5-20-1986

An analysis of variation in response time of CPU scheduling algorithms as a function of load: a simulation

John DeHority

Rochester Institute of Technology
School of Computer Science and Technology

An Analysis of Variation in Response Time of CPU Scheduling Algorithms as a Function of Load: A Simulation

by: John W.B. DeHority
May 20, 1986
Rochester Institute of Technology

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Masters of Science in Computer Science.

Approved by:

Andrew Kitchen

Prof. Andrew Kitchen

Warren R. Carithers

Prof. Warren Carithers

Walter Wolf

Prof. Walter Wolf

Title: An Analysis of Variation in Response Time of CPU Scheduling Algorithms as a Function of Load: A Simulation

I, John W. B. DeHority, prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

May 20, 1986

Abstract

This thesis analyzes a group of cpu scheduling algorithms on the basis of the variation in response time that results from changes in the system load. The results of this study quantify the differential degradation of performance across job categories. The job categories include short-burst interactive jobs as well as cpu intensive jobs. For each job type, measurements were made of average job turn-around time, weighted average turn-around time, and worst case response time. Additional statistics gathered include: ready-to-run queue size, cpu utilization and throughput. The three cpu scheduling algorithms compared are round-robin, shortest-job-first, and a multi-queue priority scheduler. The analysis utilizes a model encoded in 'C' which simulates an interactive time-sharing user community. The model allows scheduling algorithms to be measured with a controlled workload. The workload is varied by selecting the number of simulated users who are sharing the cpu.

Computing Review Subject Codes

The following subject classification codes from *The New (1982) Computing Reviews Classification System*, published by the ACM apply to this project:

C.4 Performance of Systems
- Measurement Techniques
- Modeling Techniques
- Performance attributes

D.4 Operating Systems
- D.4.1 Process Management
  - Scheduling

Acknowledgments

Top billing must go to my wife, Carolyn, for her unfailing support and encouragement, as well as her keen eye in proof reading. I owe her my sincerest thanks for seeing me through the whole process.

Thanks are also due to the three members of my thesis committee, Andrew Kitchen, the chairperson; Warren Carithers and Walter Wolf. Their analysis, guidance and support has been valuable in giving shape and direction to the project.

I also want to thank my colleagues in the KEEPS department at Eastman Kodak for contributing their "histories" to the data gathered for the calibration of the model.

Table of Contents

An Analysis of Variation In Response Time Of CPU Scheduling Algorithms As A Function of Load: A Simulation

1. Introduction and Background

1.1. The Problem

Since the advent of time-sharing in computer systems, the topic of scheduling the central processing unit (cpu) has been one of continuing interest. For my masters thesis topic I propose to model and analyze a group of cpu scheduling algorithms on the basis of the variation in response time as a function of system load. Algorithms for scheduling the cpu have been analyzed in many different ways, but upon reviewing the literature it is evident that variation in response time is one dimension of performance measurement that has not been thoroughly studied or quantified.

Variation in response time is a relevant issue in a time-sharing environment. I am using response time to mean the elapsed time from the job submission time to its completion time. Anyone who has used the VAXes at R.I.T. near the end of the quarter can attest to the reality of variation in system performance and to the relevance of this issue. The question has been asked, but not answered, elsewhere.

> It has also been suggested that for interactive systems (such as time-sharing systems), it is more important to minimize the *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 little work done on cpu scheduling algorithms to minimize variance.
> [Peterson and Silberschatz 1983]

1.2. History and Theory

The performance of the computer systems we use day in and day out has been a subject of intense interest for computer programmers, engineers, buyers, managers and users almost from the beginning. The measurement and optimization of computer system performance has consequently received a great deal of attention and analysis. Someone from each of the groups just listed has at one time or another set out to analyze the performance of their computer system. Each of these groups is likely to offer a different assessment of the same system because they each have different requirements and aims. They may each choose different indices of performance reflecting their various perspectives on the problem.

The greatest concern of a computer engineer may be the total throughput of the computer system that is being designed. Throughput is a measure of the number of jobs that are completed per unit of time. A manager of a large computing center, on the other hand, might voice concern about cpu utilization. Utilization is calculated as the percentage of the time that the cpu is actually busy processing job requests. This concern over maximizing utilization was more likely to be raised in an earlier period when cpu time was very expensive, and it was highly desirable to maximize cpu utilization. It is still a valid performance criterion. Many a buyer, for lack of attention to utilization, has purchased a system which either fails to meet the demands of the organization, causing great backlogs of work, or has purchased a system that far exceeds the organization's needs. The latter buyer now owns a high performance system in which the cpu is idle a large part of the time, even when there are users on the system. The response time of this last buyer's system will deteriorate more gradually during the initial increase in workload.

The need for, and interest in, short-term job scheduling was likely born the day someone noted that a computer's central processing unit sat idle a good portion of the time if it was only given a single job to work on during any time interval. The processor would experience frequent idle periods while the current job was waiting for input or output. It became immediately apparent that total system throughput could be improved and cpu utilization could simultaneously be improved by overlapping jobs in such a way that one job could be running while another was waiting for input or output. Charles Sauer and Mani Chandy describe the consequences of this discovery:

> A major aspect of most modern computing systems is the *sharing of resources*. The classic illustration is the multiprogrammed operating system. Put simply, the objective of multiprogramming is to have one program's use of a processor overlap with other programs' use of I/O devices so that several programs may share the machine, with each making progress similar to the progress it would make if it had sole use of the machine. This sharing of resources reduces the cost attributed to each program, i.e., if a program has sole use of the machine, it must be charged for the idle time of resources as well as the busy time. In the idealized multiprogramming system, programs are only charged for the time spent using resources. However, the sharing of resources inherently causes *contention for resources*; if two programs need the processor, one must wait.
> [Sauer and Chandy 1981]

The advent of interactive time-sharing computer systems focused attention sharply upon this contention for resources as one or more of the users on a system were forced to wait for their results. The daily user of an interactive computer system is likely to choose minimal response time or uniform response time as the most desirable or significant performance criterion. The need for predictable uniform response time has been recognized as an important performance criterion in the design of interactive computer systems. Bill Joy, of Sun Microcomputers, in his address at the Rochester Institute of Technology on February 18, 1986 stated

the design goals for interactive workstation architectures being implemented between now and 1992. One of the primary performance goals he highlighted was the need for "predictable response time."

The interactive user's first choice would be to achieve that ideal condition Sauer and Chandy [1981] described, where each job makes progress as though it had sole use of the machine. Realizing that this is unlikely, the second choice would be for the system to respond in a uniform manner across time to the repetition of similar requests. This uniformity of response is another way of describing a minimal variation in response time. Minimizing variation in response time has been noted as an important goal for designers of interactive time-sharing systems.

> Consider a MIVRS [multi-programmed interactive virtual-memory reference system] in which response times to light commands vary from 1 to 15 seconds (a total deviation of 14 seconds), with a mean value of 4 seconds and a standard deviation of 6 seconds. The high variability of response time in this installation lowers the user satisfaction level, and the system is likely to be used ineffectively. When the standard deviation appears too large with respect to the mean, a thorough analysis of the system's performance should be carried out, since the system is likely to be loaded in an anomalous way or to be inadequately sized.
> [Ferrari et. al. 1983]

Algorithms for scheduling the cpu have been analyzed on the basis of most of the criteria that have already been mentioned: throughput, cpu utilization, and average response time. But upon reviewing the literature it is evident that variation in response time is one dimension of performance measurement that has not been thoroughly studied or quantified.

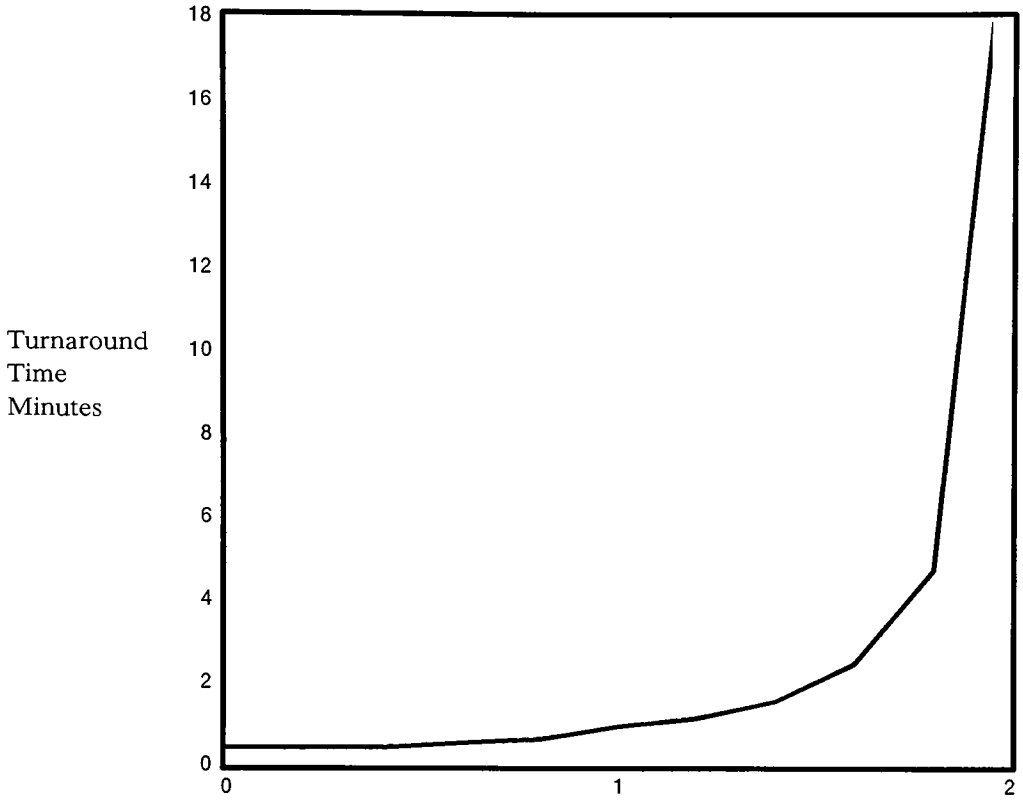
Numerous scheduling algorithms have been analyzed on the basis of these standard criteria. For example, it is well documented that the

"shortest-remaining-time-first" (S.R.T.) cpu scheduling algorithm yields the minimum average waiting time. For example consider the simple case of just two arriving jobs to be scheduled. Moving the shorter job ahead of the longer job will decrease the waiting time of the shorter job more than it will increase the waiting time of the longer job. Thus the *average* waiting time has been reduced. At least in terms of average waiting time, scheduling the job with the shortest remaining need first is provably optimal. How to implement that algorithm is another matter entirely. In the real world the job scheduler rarely knows in advance the service needs of an arriving job. Numerous heuristic methods of estimating the service needs of arriving jobs have been developed, but computers have not yet been taught omniscience. The shortest-remaining-time first algorithm remains a benchmark for algorithms which seek to minimize average response time. Most performance measures reported in the literature are ". . . average values (e.g. average response time) rather than distributional information (e.g. the 90 th. percentile of response times). Thus the word average should be understood even if it is omitted." [Peterson and Silberschatz 1983]

What happens to that average waiting time in a time-sharing environment as the load on the system increases? This is an important, yet unexamined question. The initial answer to this question is intuitively obvious. As the load on a system is increased the average waiting time will inevitably increase at some point. Again, note the *average* waiting time will increase. It is more difficult to determine what will happen to the distribution of waiting times, particularly by job type. Consider a very lightly loaded system, where the cpu utilization is so low that the cpu is idle a large percent of the time. To keep this initial analysis simple, let's further assume that we have a homogeneous job mix--that is, the jobs have very similar characteristics and demands. We would logically anticipate that a certain number

of jobs could begin to share the cpu and introduce little or no simultaneous contention for use of the processor. In the absence of contention we would expect throughput to increase, and cpu utilization to increase also, with almost no change in the response time for any of the jobs. However, as soon as the multi-programming level reaches the threshold where jobs are simultaneously contending for the use of the cpu, then waiting time is introduced and the average response time will inevitably begin to increase. The function that relates the changes in response time to the system load (i.e. the number of jobs), would describe the variation in response time for this homogeneous population. Graph I represents the change in response time we would typically find in this simple case. [Ferrari et. al. 1983]

Response Time



[Ferrari et. al. 1983]

Graph 1

This simple case, involving a single job type, could be analyzed through an application of Little's Law. [Little 1961] If N represents the average number of requests in the system, R represents the average system residence time per request, and X represents throughput, then Little's Law stated that N = XR. The average number of requests in the system is equal to the product of the throughput and the average time spent in the system by a request. In the simple illustration given above, it was speculated that the multi-programming level would initially increase as load increased, which is to say the number of processes co-resident in the system would be increased. The result, in performance terms, according to Little's

formula, would be seen as an increase in throughput. The formula would lead us further to the conclusion that when throughput levels off at or near the system's maximum capacity the average system residence time per request will increase, along with a proportional increase in the average number of requests in the system. In other words the system's queues will grow and response time along with it.

Little's Law has given rise to a number of corollaries useful in the analysis of system performance. One of these corollaries useful in the analysis of interactive computer systems is the Response Time Law, [Lazowska et. al. 1984] which says that the response time (R) is equal to the number of users on the system (N) divided by the system's throughput (X) minus the average time that each user spends thinking between job requests (appropriately named 'Z').

The Response Time Law

$$R = \frac{N}{X - Z}$$

Suppose that we initially have 40 users on the system, which can complete 2 jobs per second, and that each user spends an average of 15 seconds thinking between jobs. The Response Time Law tells us that the users will have to wait an average of 5 seconds for the system to respond and complete their job. If the number of users is doubled this law tells us that the response time will increase to 25 seconds! This analysis utilizes a time averaged value for throughput, that represents some number of completed jobs in a period of time. The analysis grows more complex than these simple illustrations when we look at examples that are more representative of modern time-sharing systems. In order to improve response time many modern systems tolerate the overhead of very low facility utilization

under light to moderate loads. In other words, the cpu and I/O subsystem sit idle a significant portion of the time under these reduced loads. This allows the system to respond to the initial increase in workload by an increase in facility utilization and a corresponding increase in total system throughput. The example just given held the throughput constant, which led to a five-fold increase in response time when the number of users doubled. If job throughput also doubled then there would be no net change in average response time.

As processor time becomes less and less expensive it will become increasingly common to size systems so that cpu utilization is low under the typical expected operating load. Thus the dramatic increase in response time that accompanies an increase in the number of users will be partially offset by an increase in throughput. But it is not realistic to assume that throughput will, or can, double to match every increase in the user population, or in job demands. At some point in every system the addition of more users, or an increase in service demands for the jobs submitted, will lead to full utilization of one facility or another within the system. This point of full utilization of a facility is called the saturation point. When that saturation point is reached, throughput levels off and response times begin to increase rapidly as the queued backlog of jobs grows at this critical facility. This facility is then labeled the "bottleneck" in the system. Traditionally system performance analysis and optimization involves a search for this "bottleneck" and its abolition or reduction. Performance analysis has been described as the abolition or reduction in backlogs caused by peak loads and the postponement of time when workload increases will saturate the system forcing its expansion or replacement. The systems analyst attempts to identify these "bottlenecks" and either "widen" them or eliminate them. The system will then be capable of handling a larger load, but at some point a subsequent or secondary

bottleneck will define the new limits of system performance.
[Ferrari et. al. 1983]

The analysis thus far has treated all jobs equally, without making any differentiations of job categories. It should come as no surprise that in many systems the jobs are not uniform in the demands they place upon the system. When we make the determination that various job types should be treated differently within the computer system on the basis of the differential demands for service, then we have drastically increased the complexity of the analysis.

> We have a relatively good understanding of the performance of systems designed to meet specifically described loading conditions, particularly those loads having small variation. On the other hand, those systems required to meet unspecified loads, with large variations in the loading conditions, have proved to be difficult to build. With respect to these latter systems we find that we are in a relative state of ignorance. [Lynch 1972]

There have been many scheduling algorithms developed over the years to meet varying needs of the computing community. They have ranged from the very simple first-come-first-serve algorithms to very sophisticated multi-level feedback and aging algorithms. These algorithms have all been thoroughly analyzed on the basis of cpu utilization, maximum throughput, average turn-around time and average response time. Peterson and Silberschatz [1983] stated "little work had been done on scheduling algorithms to minimize variance." A review of the computer science literature revealed that this void observed by Peterson and Silberschatz has not yet been filled. An analysis of scheduling algorithms on the basis of how response time changes as a function of system load has not been reported in the literature.

The performance of a number of the simpler traditional algorithms can be predicted through a reasoned analysis. The simplest of these algorithms, the

first-come-first-serve algorithm, gives the use of the cpu over to the first job that arrives. The performance of this algorithm is known to be widely variable and entirely dependent upon the order in which jobs arrive. Smaller jobs will tend to form a "convoy" behind the larger jobs in the queue. Average response time is certainly not minimal, and can vary substantially depending upon the order of arrival. The average response time of this algorithm varies widely even before variations in system load are taken into consideration. First-come-first-serve scheduling, while it may be the easiest algorithm to encode, is probably the least satisfactory.

The excellent performance of the shortest-job-first algorithm has already been reported. "It is intuitively optimal with respect to mean response time since it maximizes the number of response times completed in a given interval of time." [Sauer and Chandy 1981] It will consistently yield the minimum possible average response time. The good news is that this quality would apply under a wide range of loading conditions. The bad news is that, "at the cpu (short-term) scheduling level it is unimplementable."

> There is no way to know the length of the next cpu burst. One approach is to try to approximate SJF scheduling. We may not *know* the length of the next cpu burst, but we may be able to *predict* its value. We would expect that the next cpu burst will be similar in length to the previous ones. Thus by computing an approximation of the length of the next cpu burst, we can pick the job with the shortest predicted cpu burst.
> [Peterson and Silberschatz 1983]

In a batch environment, where the historical time demands of regularly submitted jobs are well known, it is common practice to use a shortest-job-first ordering of submitted jobs. At the level of short-term cpu scheduling a number of formulas have been applied in attempts to predict accurately the length of the next

cpu burst. By keeping track of the lengths of the most recent past bursts for a particular job a reasonable prediction of the next burst can be made by taking an exponential average of the previous bursts.

In the controlled world of a model we calculate events in advance so that when a job arrives at the scheduler its needs are already known. Thus the modeler has the artificially constructed opportunity to study the performance of a system under the ideal conditions of *knowing* the needs and not having to *predict* them through some heuristic mechanism. The cpu bursts of highly interactive jobs, such as file editing, tend to be of very short duration consisting in large part of processing I/O interrupts. Shortest–job–first algorithms should yield a differential degradation of performance across job–types as system load increases. The increase in response time should be greatest for large compute–bound jobs, such as compilation or "number crunching" (like running computer models!), and smaller for short–burst interactive jobs like editing.

Since we know empirically from the Response Time Law and Little's Law that the average response time is going to increase as system load increases the whole quest for minimizing variance in response time may seem at first like the search for the holy grail. We know that something, or someone, has got to give when the system load increases. No algorithm can avoid this hard reality. Different scheduling algorithms can change the differential degradation in performance across job types. The operating system designer can implement administrative policy that will give preference to certain job categories over others when there is contention for the cpu. So to some extent the question that this thesis raises is not purely a technical one, but an administrative policy one as well. Who gets the processor when the load gets heavy? It is possible to answer this question for any

scheduling algorithm. These questions of policy are rarely addressed in the analysis of scheduling algorithms.

First–come–first–serve, with the addition of time quantum interrupts becomes round–robin scheduling. This algorithm doesn't yield optimal performance, but it is fair. As the load increases, everybody is penalized equally in terms of the response time they experience. But such a simple algorithm naively ignores the human reality of how people respond to the machines they use. Users are more likely to tolerate (maybe even react favorably to) a system that they experience as having uniform and predictable response time, even if that predictable level of response is not the optimum that could be achieved under ideal conditions with a different algorithm. An interactive computer system builds up a certain pattern of responses that people come to expect. As performance deviates more widely from that expectation, then the users' evaluation of the system becomes increasingly negative. Users of a system that is slower, but predictable, may in moments of impatience wish for a faster system, but they will not develop the antagonism that begins when a job that "usually takes a few seconds", suddenly takes "much longer."

What algorithms might best satisfy these human engineering factors? The most likely candidates will be algorithms that assign priorities to job types. Let me venture into the realm of policy for a few moments, and state my opinion. From an experiential perspective, I would assert that nothing is more frustrating than having slow response to simple common actions. Anyone who has ever been typing and found that they were suddenly many characters ahead of what is displayed on the screen knows this frustration. My bias would be toward scheduling interactive jobs, and short, simple, common commands, ahead of the larger more "batch like" jobs. It has been asserted, and I believe it is true, that users are highly adaptive to the

characteristics of the machines they use. Whether its the loosening steering of an aging car or the response time curve of a computer system, we change our usage to adjust to the performance of these systems. So if larger jobs are going to run at a lower priority, we would expect users to find ways to make their jobs smaller. Making one's programs more modular, so that only small portions need to be recompiled, rather than recompiling entire segments of a program that are largely unchanged, is one desirable change that might result in these circumstances.

There are inherent risks in the implementation of priority schedulers. The most significant risk is starvation. Low priority jobs may be totally denied access to the processor by a steady stream of higher priority jobs. If it appears that starvation of lower priority jobs is occurring then remedial actions may be necessary. One option for solving the problem of starvation is to blend the priority scheduling with job aging. Aging algorithms dynamically change the priority of jobs so that as jobs remain in the system queues for long periods of time their priority gradually increases with age. An alternative response to the problem of starvation is a total reassessment of the utilization of the system, which may well conclude that the system is inadequately sized to support the current number of users. The required remedy may be to reduce the number of available terminals on that system and transfer some of the workload to another system. Another alternative might be to delay the submission of these low priority jobs until lightly loaded periods, such as the middle of the night. This alternative is particularly practical with jobs that require no user interaction to complete.

Starvation of low priority jobs may also be an indication that the categories chosen for assigning job priorities may need reassessment. The blend of jobs in the workload may have too many jobs that fall into the scheduling algorithm's high priority job category. Lowering the scheduling priority of some of these high

frequency job types may cause users to alter their patterns of system use, so that fewer of these jobs are submitted in the future.

The workload placed on an interactive computer system is very dynamic. The community of users is continually finding new ways to utilize their computers. Every time we re-tune the performance characteristics of the computer system in response to the user community's new workload pattern, we set off another reverberating set of ripples through the utilization habits of the user community. The interactions between the computer system and the user community are very dynamic and constantly evolving. While cpu job scheduling may be an old topic that has been studied and debated many times, it is not a closed one. Algorithms have been written, and will continue to be written, which very adequately solve the needs of a particular computer system, under a particular set of workload characteristics. But systems and workloads continue to evolve. New processor technologies are evolving, massive parallel processing is appearing in new systems, and tightly coupled networks are sharing workloads. There is a continual need to re-evaluate the algorithms which parcel out the utilization of these evolving computer systems. The ultimate all purpose scheduling algorithm has not been written, and probably never will be. Several authors before this one have concluded that performance evaluation and tuning is as much an art as it is a science.

2. Project Description

2.1. General Considerations

Central processor job scheduling in a multi-user, multi-tasking environment is the primary focus of this project. There are three criteria that must be satisfied for an adequate testing environment. First, the testing environment must allow easy modification of the job scheduling algorithm in effect at any given time. Second, it must allow for systematic control and variation of the workload imposed upon the system. And thirdly, it must provide for gathering detailed, accurate feedback of the system's performance under these varying circumstances. It must be possible to obtain these latter measurements without affecting or coloring the performance of the system under study. It is often the case that when performance monitors are added to an existing system they generate an additional workload on that system which affects the level of the true workload that the monitor is seeking to measure. This is an undesirable side effect of studying actual existing systems which is difficult to avoid.

The ideal testing environment for such a study is not readily available, if indeed it exists at all. Existing multi-user installations are not often accessible for kernel modifications, and were such modifications possible there would remain the difficulties of controlling the workload and measuring the system's performance. The difficulties of controlling these variables seem to outweigh the benefits of conducting the study on an actual existing system.

The most viable alternative is the construction of a computer simulation model that satisfies these same criteria. The goal then is to arrive at a carefully conceived and constructed qualitative model that will allow several job scheduling algorithms to be tested under a variety of controlled loads. The model should allow

the gathering of data and the generation of statistics measuring the "system's" performance without affecting that performance. A model can isolate the operation of the target system from the monitors which measure its performance. This is rarely possible in an actual system.

2.2. The Advantages and Limitations of A Model

A model approximates, not replicates, the behavior of an actual system. The modeler must discern the essential parameters and characteristics which affect a system's operation and capture those aspects in the model's design. "A model is an abstraction of a system: an attempt to distill, from the mass of details that is the system itself, exactly those aspects that are essential to the system's behavior." [Lazowska et. al. 1984] It is this very abstraction that is both the strength and the weakness of the model. Due to this abstraction process the model does not exactly replicate the behavior of the real system it is derived from. Instead the model seeks to provide qualitative representation of the actual system. The essential behavior characteristics of the original system must be present in the model for it to be a valid representation of the original system. The model should "reflect the general behavior of a system but not necessarily specific values of its performance measures." [Lazowska et. al. 1984]

The model can provide an efficiency, flexibility and adaptability that were not possible in the original system. These benefits are derived at the expense of precision. While a model should provide results that are qualitatively correct, they may be quantitatively imprecise. These qualitative results can provide the guide for further investigations which implement in an actual system the findings of the model. The role and purpose of the model will have been well met if it has allowed a range of possibilities to have been checked with efficiency and care. The model maker gives up quantitative precision in order to pursue and efficiently qualify

hunches and intuitions about a system's behavior. Bear in mind that the primary goal of this study is to evaluate cpu scheduling algorithms, not any particular computer system. The model should prove a valuable means of gaining insight into the relative behavior and performance of various scheduling algorithms under a wide range of loading conditions.

2.3. A Functional Description

The model must satisfy the three criteria established above. It should allow different scheduling algorithms to be exchanged easily and tested with similar workloads. The model should generate and present the necessary statistics to assess the effects of workload on response time and system performance. There are several discrete functions that must be performed by this model. The implementation of the model should ideally reflect the separation of these functions. For example, the functions having to do with the generation and reporting of statistics should probably be gathered together into one module of code. Similarly, those procedures which create and parameterize the user workload should be gathered together. Organization of the code at this level will greatly facilitate the debugging, tuning and validation of the model.

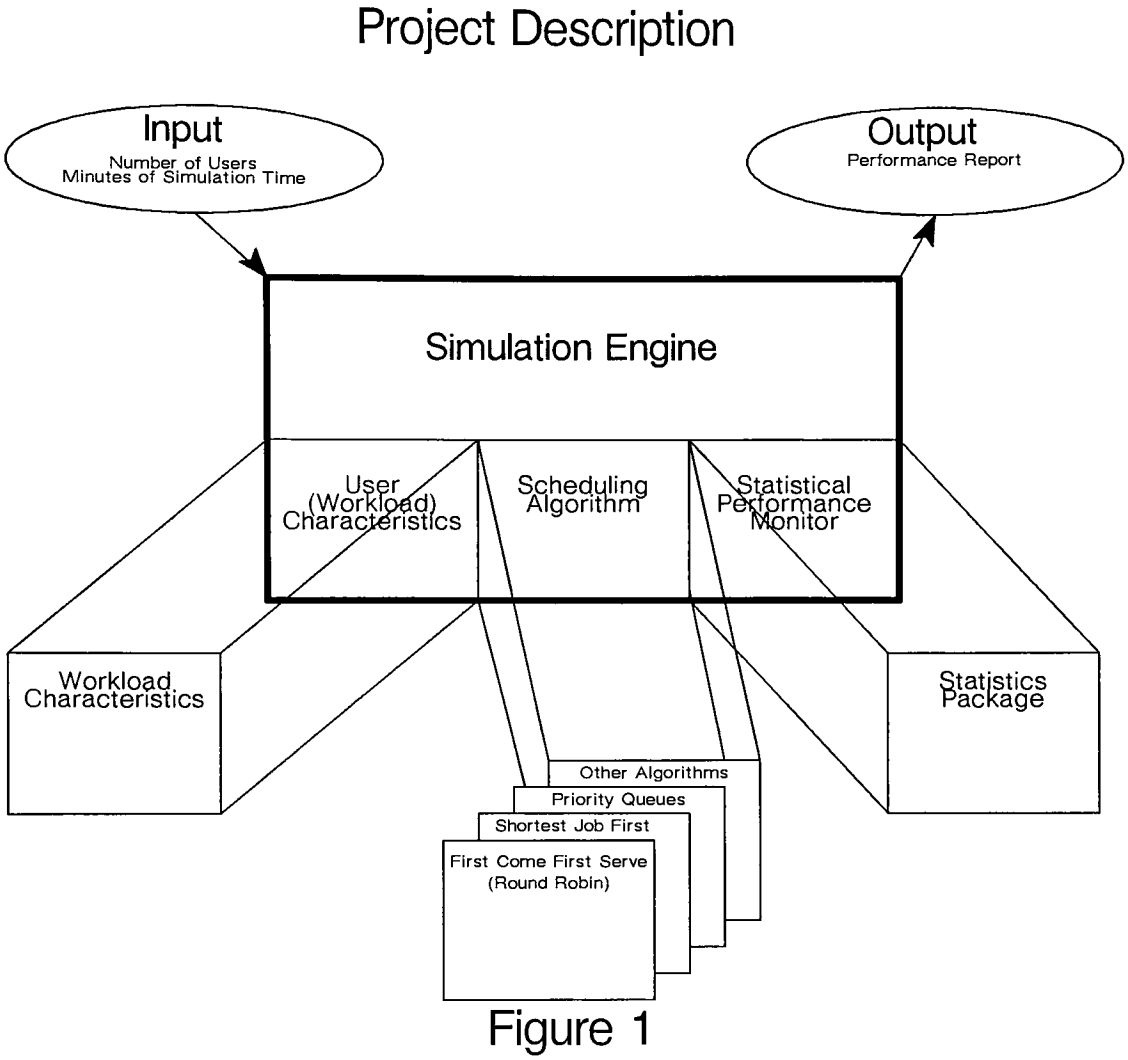


Figure 1

2.3.1. The Workload

The first function of the model is to generate an artificial workload that simulates the activity of a specified number of users. For each user a series of "jobs" must be generated which simulate the cycle of commands that a typical user might issue (edit, list files, edit, compile, edit some more, compile, run, print, etc.). The workload imposed on the system will be controlled by adding and removing "users" and examining the variations in response time that occur. Thus it should be possible to specify how many "users" are on the system during any

particular measurement interval. One of the assumptions that the model makes is that user behavior is homogeneous.

To parameterize the workload, and further define the behavior of a typical user, data was gathered from an existing time-sharing user community. A group of programmers working together on a software development project agreed to save their "history" files which record the sequence of commands which they executed during their work session. These programmers are working on Sun Workstations, which are Motorola MC-68010 based UNIX system. This information was gathered by having each programmer provide a record of the system load, their history of commands, and the time demands of their work session. This information was obtained by placing several commands in the user's ".logout" file.

The "uptime" command indicates the number of users on the system and the relative job load. This command can be used to put a time-stamp on the growing master history file and also indicate the number of users on the system as well as the relative job load. The "history" command appends the last one hundred commands executed by each user before logging off of the system. The "time" command records the total amount of user, system and real time utilized by that user since the creation of their login shell.

From this master collection of work history files a data file of typical command execution sequences and relative frequencies was created. This listing of executed commands was analyzed to determine the relative frequency of occurence of each command. This was further analyzed to plot the relative order of command execution as a transition matrix. If the current job is "A", what percent of the time is the subsequent job "B", "C" or "D"? There are typical sequences of commands that users frequently execute. For example after changing working directories, ("cd" command in UNIX), it is common to list the contents of the new directory,

("ls" in UNIX). Similarly, there are sequences of commands that are not common. A user who has just completed compilation of a program is not likely to recompile immediately without an intervening session of further editing or testing. From this analysis a data file was generated to represent these typical transitions. This data file is used by the functions which simulate "typical users" to generate a workload with a similar sequence and frequency of commands. The module that generates the user workload exhibits behavior similar to the actual user behavior, as just described.

One of the early limitations of using "history" files was that only the last twenty-five to thirty commands were being saved. It was quickly perceived that there was a gradual evolution in the sequence of commands that a programmer executes as the end of a work session is approached. The true sequence of commands executed was not being captured in its totality. To correct this problem the users later maintained and reported a much larger personal history of one hundred commands.

When a user's job is created by the workload simulation module certain things need to be known about that job besides its arrival time. It is necessary to determine the service requirements of this job based on the job type. To parameterize this aspect of the simulated workload the UNIX "time" function was used to measure the service requirements of a large sample of each of the job types that were included in the job sequence and frequency database. The UNIX "time" function returns the following information: the amount of user time, system time, and real time used in the execution of the command; the amount of memory used, the number of I/O interrupts; and the amount of paging and swapping activity associated with the timed process. This data was recorded for many samples of each of the included job types. A second database created from these

values allows typical service demands to be generated for each of the created user jobs.

2.3.2. The "Simulation Engine"

The second important function of the model could be called the "simulation engine". The overall control of the simulation, the initialization of the model, and the termination of the simulation are all controlled by this "simulation engine". For example, the model begins by determining how many users are to be simulated in this iteration and how many minutes of simulated time the model will continue to execute before printing out the accumulated statistics. The simulation engine has to maintain a "system clock" so that events can be scheduled, timed and recorded.

The other major responsibility of this portion of the model is maintaining and manipulating the internal queues of jobs in various states which constitute the inner workings of the model. The first of these internal queues contains the "arriving jobs" that are being submitted to our simulated processor for execution. The model must maintain a queue of "future events" that will represent the jobs being generated by the simulated users of the system. The module that simulates the user workload is responsible for generating the jobs that are entered in this queue.

The second of these internal queues is the "ready-to-run" queue of jobs that are ready and waiting for cpu service. Jobs are moved from the queue of "future events" into the ready-to-run queue when their designated "arrival time" is reached by the inner clock of the model which keeps track of simulated time. The job to be scheduled is placed into the ready-to-run queue according to the scheduling algorithm that is currently being tested. The cpu job scheduler is a

component of the model that is easily changed or exchanged to implement a different scheduling algorithm.

The "simulation engine" removes jobs from the ready-to-run queue and "runs" them. As jobs move from one state to the next, time-stamps are maintained for each job which allows the gathering of statistics for response time, execution time, and waiting time. When a job finishes running these statistics are gathered from the time-stamps before the job "leaves" the model. The termination of a job will trigger the generation of the "next job" for that particular user which is then be placed in the queue of future events.

In multi-process time-sharing computer systems jobs share the cpu with other jobs. When a job begins service at the cpu it does not necessarily finish before it must give up use of the cpu to other waiting jobs. The quantum of time allocated by the system may expire before the job is completed, in which case the job must be placed back in the ready-to-run queue according to the current scheduling algorithm. The "simulation engine" is responsible for maintaining this sharing of the cpu time, removing jobs from the running state that have exceeded the designated time quantum and are not yet finished, or are blocked waiting for interactive keyboard input. Jobs in the "blocked" waiting state are rescheduled as future events due to arrive at the time of their next calculated "input".

The first scheduling algorithm implemented is a simple round-robin scheduler where the jobs are placed in the ready-to-run queue on a first-come-first-serve basis. The second cpu scheduling algorithm implemented is an optimal scheduler that can be represented in a model but not in reality--this being the shortest-remaining-time-first scheduler. Since time-stamps are being maintained on each job it will be known at all times what the job's total need is, and what its remaining un-met needs are. Having this information, which one

would never have in a real computer system, it is possible to schedule the jobs in order of their total unmet need--in other words the shortest job first. A third algorithm to be examined maintains two ready-to-run queues, one for "interactive jobs" and a second for "background" or "batch" jobs. Interactive jobs will be run first, and when that queue is empty jobs will be run from the queue of batch jobs. The flexibility of the model allows for additional scheduling algorithms to be implemented and tested with relative ease, but shortest-job-first, first-come-first-serve and priority queues are the three algorithms that will be analyzed here.

2.3.3. Coming To Terms

The model attempts to represent a user's perspective on the world. Therefore the performance parameter that this thesis focuses most closely on is system response time. I am using response time to mean the elapsed time from the job submission time to its completion time. I believe this is the best criterion for measuring interactive performance and most closely represents the user's perception of system performance.

There is another definition of response time which measures the elapsed time from the submission of the job until the first appearance of response at the terminal. Often the first appearance of a response will precede the completion of the job by a significant interval. The first appearance of response can be very reassuring to the user that the system is indeed working on the submitted request, particularly when the final completion may take significant fractions of a minute or longer. Some initial early appearance of response is reassuring and helps prevent impatient users from re-submitting the job or banging on the keys to see if the system is still up or responding. I have chosen not to use this second definition of response time, but focus instead on the final completion of the submitted task.

In the inner workings of the model, response time is measured as the difference between a job's arrival time and its departure time. A job is assigned an arrival time as it is placed in the future-events queue. The job "arrives" and is moved into the ready-to-run queue when the simulation clock is equal to or greater than its scheduled arrival time. If a job is currently running when the new job is scheduled to arrive, the arriving job will have to wait until the currently active job completes its current cpu burst. Departure time is marked when the job leaves the cpu for the last time. Departure should occur when the total service received is equal to the total service need that was determined when the job was created by the "User and Job Demands Module."

The model also keeps track of cpu utilization and total throughput. Total cpu utilization is the percentage of the total simulated time that the cpu was actively running a user's job. Since the system overhead associated with a user's job is combined with the user time needed for a job, this cpu utilization figure could also be described as the percentage of "non-idle" time. The throughput is kept as a tally of the total number of jobs completed during the period of simulation. This total can be divided by the number of minutes of simulation to arrive at a jobs completed per minute figure. Throughput and cpu utilization are good measures of system performance, however response time is a better indicator of how the user would experience or evaluate the system's interactive performance.

2.3.4. Statistics--System and Job Related

The last function of the model is to gather two categories of statistics. Processor utilization statistics are kept so that a system saturation workload level can be identified. Statistics for the ready-to-run queue (or queues) are maintained. What is the maximum size that this queue reaches? What is its average size?

Histograms describing the population sizes of the ready-to-run queue provide a graphical representation of the size of the waiting queue over time. The model has been designed to report that for x amount of time there was one job waiting to run, and for y amount of time there were two jobs waiting to run, etcetera. The model reports system population statistics, so that a system population of zero corresponds to the amount of time that the processor was idle, and a system population of one corresponds to those times when there was no job waiting to run while a current job was being processed. This makes it possible to calculate and report an average system population. This additional average system population statistic is printed out along with the data and histogram representations of system population statistics.

The second category of statistics are job related statistics. What is the user's perceived response time? This user response time is further subdivided by job type. One of the special interests of this study is to determine how response times of certain job types can be optimized (namely interactive jobs such as editing), and what effect that optimization will have upon response times for other job types. It is also important to detect whether any job types are being starved out under high loads with any of these algorithms.

In the area of job-related statistics an additional measure of job-related system performance is reported. Utilizing average response times alone does not allow meaningful comparisons to be made between the changes in response time under increasing loads of two different job categories which have very different typical system demands. For example, an editing job has an average response time of .042 seconds when there is a single user and .261 seconds when there are thirty users on a system which utilizes round-robin scheduling. How does one compare those numbers with execution of a large compilation request whose response time

averages 36.68 seconds for a single user and increases to 364.2 seconds when there are thirty users sharing the system? The difference between the system demands are so different for these job categories that it is difficult and perhaps even meaningless to compare the differential changes in response time between these two job types on the basis of average response time.

While average response time is still an interesting statistic to look at, weighted turn–around time is the measure of choice for comparing how jobs of different types fare in system performance. [Ferrari, et. al. 1983] Weighted turn–around time is the ratio between turn–around time and actual processing time. On a system with a single user this ratio should be very nearly equal to one for all job types. The time required by the system to process the job is divided into the total turn–around time to yield a ratio that can be meaningfully compared across job categories. These job types may differ widely in the demands they typically place upon the system.

The use of weighted turn–around time is of further help in diminishing the impact of variations in system demands within a single job category. All compilations do not have the same processing demands. Weighted turn–around time yields a measure of system performance which takes these variations in processing demands into account.

The model reports statistics on weighted turn–around time. When the model is running in the "verbose" mode the weighted turn–around time for each job is reported as it completes execution. The summary job statistics report includes the mean weighted turn–around time for each job type. The mean weighted turn–around time was found to be the most useful measure of system performance for the comparison of the different scheduling algorithms. This measure of system performance was used extensively for the graphs depicting the relative

performance of scheduling algorithms. It should be noted that weighted turn-around time is particularly sensitive to changes in shorter duration job types.

2.4. Limitations of the Model

The model is likely to be insensitive to some of the interactions between components of an actual computer system. In particular, the model will not replicate some of the thrashing phenomena that have been observed in paged virtual memory computer systems as they approach saturation and there is a dramatic increase in page faults. The present design of the model does not directly represent memory contention, swapping or paging. The cpu overhead associated with these activities is captured in the "system time" component of the UNIX time command that is being used to calibrate this model. So while these activities are not directly represented, their impact upon system performance is taken into account.

The model does not separately represent the queuing which occurs for disk I/O. The times associated with this queuing are contained in the user and system times generated by the UNIX time command. The model factors these queuing delays into the total delays that the user perceives as response time. Questions about the interaction between a given cpu scheduling algorithm and the intensity of I/O activity cannot be answered by this model. An avenue for future exploration and enhancement of the model might be to take disk I/O queuing into account directly. It would then be possible to examine queuing algorithms which give differential treatment to I/O bound and compute bound jobs. It is likely possible that further control and tuning of response time could be obtained by differential scheduling of these two job types.

The current model is designed to explore the differential scheduling of so called "interactive" and "batch" jobs. A question that can not be answered by the model with any certainty, is the relationship between what could be described from a user's perspective as an "interactive job", and from the system's perspective be described as an I/O intensive job. These may in fact be two closely related descriptions of the same thing viewed from different analytical perspectives.

2.5. Input to the Model

As the model begins it reads from standard input to determine the number of users on the system during this iteration of execution. It also reads in the number of minutes to be simulated, and a number representing which scheduling algorithm to utilize. These are entered as three integers. They can either be entered interactively when prompted, or placed in a data file which is redirected to standard input.

The initialization process sets pointers to functions that perform the job scheduling and fetching the next job from the ready to run queue. These function pointers are set on the basis of the scheduling algorithm selected. This allows scheduling algorithms to be selected from among the available set without recompiling the model.

Rather than hard-coding the behavior of users and time demands of jobs into the model, the data for scaling these quantities is placed in external files. During initialization the model will open a file named "user.data" which contains a square matrix of integers. For each job type represented in the model there is an entry in the "user.data" matrix giving the likelihood of executing each of the other job types next. For example, if a user is currently editing, the matrix allows the model

to calculate what percent of the time the next command issued will be to edit further, or compile, or print.

The file "job.demands" is also read into the model at initialization. This file contains a typical distribution of the time demands for each job type that is represented in the model. As a job of any given type is generated the model randomly selects the time to associate with that job from the distribution of typical times for that job type which are contained in the "job.demands" file.

By placing this data in external files it is not necessary to recompile new data into the model during the process of calibrating and validating the model. Placing this information in separate data files where it can be easily edited allows this model to apply what is already known about job time demands. Re-editing a file places much lower demands upon a system than does recompiling a program. Copies of these input files are found in Appendix B.

2.6. Output from the Model

The model prints out a running record of user activity. As each job is completed a line is printed to standard output recording the user's id, the job type, arrival time, departure time, total service needed, response time, and waiting time, cpu utilization and size of the ready to run queue at time of departure. This verbose output can be suppressed by recompiling the statistics module with the "VERBOSE" flag undefined.

```
                time (reported in hundredths of seconds)

type  user  depart arrive  need Res Tm Wait Tm %CPU Q-size

vi       2    2200   1600   300     600      300 0.12    3
ls       3    2300   1700    90     270      180 0.25    2
make     1    6900   1650  2300    3100      800 0.84    4
```

These quantities are also internally accumulated so that statistics for average response time by job type can be calculated at the conclusion of the simulation. When the model has finished simulating the specified number of minutes, then summary statistics are printed on the standard output. These summary statistics include a listing of total cpu utilization and throughput, and a listing of how many jobs of each type were executed. For each job type the model calculates the average response time, the standard deviation in that response time, the weighted turn-around time, and the worst case for response time of that job type. The model also accumulates and prints statistics describing the population size of the ready-to-run queue. The output of the statistics produced by the model appear as follows:

```
********** System Stats ************

Number of users: 3
Total minutes simulated: 15
Total seconds simulated:  900
CPU seconds utilized: 791
CPU utilization  87.99%

Job Type Number Percent

 vi          30  44.12
pwd           0   0.00
cd            3   4.41
ls            5   7.35
more          3   4.41
diff          2   2.94
lpr           4   5.88
run           8  11.76
cc            8  11.76
make          5   7.35
Total jobs completed: 68

******** Queue Population Stats ******

size         millisec.       minutes
0            97583   ***************
1            25821   ****
2            11115   *
```

********** Job Stats **********

Job Type	Number	Avg.Resp.	Std.Dev.	Worst Case	WTAT
vi	30	18	8.37	89	1.10
cd	3	0	75.91	0	1.36
ls	5	1	26.21	2	1.41
more	3	3	75.38	6	1.31
diff	2	15	143.69	20	1.85
lpr	4	4	43.74	13	1.48
run	8	4	27.62	12	1.50
cc	8	69	2.83	116	1.88
make	5	48	46.77	63	4.65

2.7. Validation and Verification of the Model

An important step in the design and implementation of a model is the verification and validation of the model and its output. Validation is the process of building confidence in the model and its output. Verification is the process of component testing which assures that every part of the model is functioning according to its specified design. There is no one single test for the validity of a model, however the process of validating a model may include any of the following factors:

First, a valid model provides results which are reasonable and consistent with common sense. We would immediately reject the output of a model which showed an improvement in response time as loads increased. A second way of stating this criteria for validity is that the output generated by the model should be a good representation of the real world as it is perceived.

The observation of continuity and consistency in the output of the model increases our confidence in the validity of the model.

Continuity in the output over a range of input conditions is important. To be valid a model must generate reasonable results with multiple sets of input conditions. This leads us to the criteria of consistency. Assuming that a steady-state has been reached the number of jobs that are completed by the model should not have any affect. The model should not yield one set of results when run for one time interval and another set of results if that time interval is increased.

The model should provide an acceptable level of detail in its output to be of any use to the modeler. It must give us sufficient information to be able to make judgments regarding its activity and behavior.

The validity of a model is measured in terms of its suitability for a particular purpose. Validity is to some extent contextual, it depends upon the initial expectations and assumptions brought to the modeling process.

A valid model emulates events in the real world. A model should emulate both the interactions and the cause and effect relationships that exist in the real world. This observation must include the caveat that, the exact mechanisms that control the behavior or a real system or of a model may be elusive. There may be surprisingly subtle interactions between components of a system. The creation and interpretation of a model is part art and part science!

An important component in the process of validation is determining that the model is in a steady-state at the point when we begin

making measurements and gathering statistics. When a model begins running its internal queues are empty. This would bias any statistics regarding average queue size or average response time. For this reason it is necessary to allow a model to run for a calculated period of time to "prime the pump". Following this priming it is good modeling practice to reinitialize all of the statistics being gathered but not to change the queues. The model is allowed to run for a period of time, referred to as T_b, after which the model resets all gathered statistics. The value of T_b is usually calculated by running the model a number of times and observing the point at which the population of its internal queues have entered a steady state.

An analysis of how successfully the implementation of the model satisfies these validation criteria will be provided in a later section. The validation process begins in the early design phases, and continues through the implementation, running and analysis of output. If we are to have confidence in the output of the computer simulation we must first have confidence in the conceptual model and design. The description of the model's design lays the foundation for the assessment of validity. The conceptual foundation must be sound.

Once a valid conceptual design is complete the process of implementation, verification and further validation can proceed. Verification of the implementation is an important part of the modeling process. The Queue Management Module contains a set of tools that are central to the functioning of the entire model. A series of small drivers were written to test the procedures which add and remove jobs from various queues in the model. In particular, the procedure to insert jobs into queues in a sorted order was carefully tested outside of the model to verify

that the conceptual design was correctly implemented. Similar tests were made of the random number generation procedure.

Additional steps have been taken in the implementation to aid in the verification process. For example, as the "next job" matrix file is read into memory the cumulative percentages are checked to be certain that each job category adds up to 100 percent. If a job category does not add up correctly then an error message is printed and the model aborts. A number of other error conditions have been identified and appropriate error traps for these conditions have been placed in the model. Being an event driven model the ready-to-run queue and the future events queue should never both be empty at the same time. If one job has completed running then its successor should already be in either the future events queue or the ready-to-run queue. Error conditions such as this one are trapped and will cause the model to abort giving diagnostic messages.

3. System Specification

3.1. Two Perspectives To Consider

This project is a software simulation of central processor job scheduling in a multi-user time sharing environment. Consequently when we begin to discuss the architecture and data flow we must be careful to differentiate between the structure of the model and the structure of the simulated system that is represented within the model. Failure to make this distinction can lead to a great deal of confusion.

3.1.1. The Modeled System

The system that is being modeled is relatively simple. It is pictorially represented in the System Data Flow diagram (Figure 2). A group of users are independently submitting jobs of various types to a central processor. When their job is completed they are prompted at their terminal and they submit another job. Inside this shared processor there is an arbitration mechanism that submits each of their jobs in turn to the processor for execution. Time sharing is implemented through a system clock that insures that no one job monopolizes the processor.

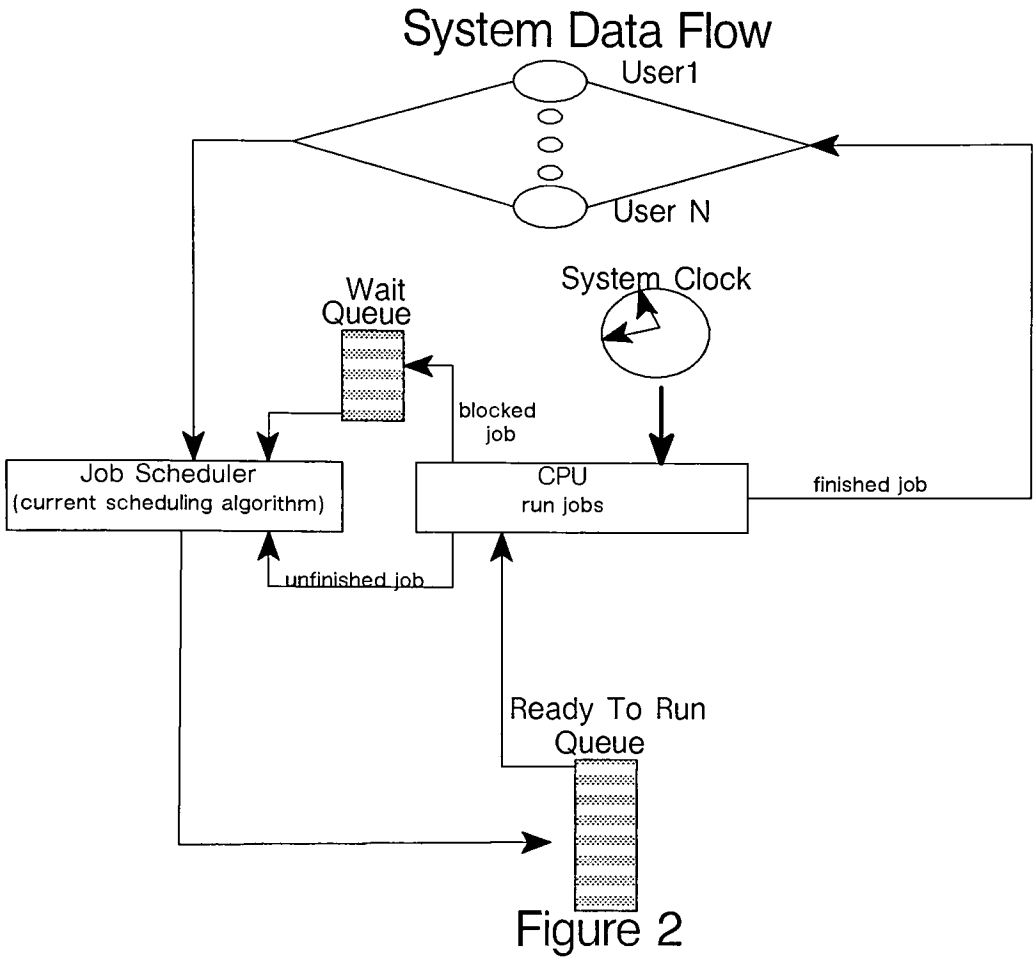


Figure 2

The central processing unit's time will be apportioned among the waiting jobs in a variety of ways depending upon the current job scheduling strategy that is in effect. The goal is to be able to change the job scheduling strategy that is in effect, and have that same number of users submit a similar mix of jobs to the processor so that we can observe changes in response time from the users' perspective.

3.1.2. The Simulation

All of the components of the modeled system must be represented inside the simulation. Not only must the processor that handles all of the submitted jobs be represented with all of its internal details, but there must also be a representation of the users who are generating the workload. In addition we want the model we

construct to provide us with statistical data about its functioning. From a descriptive perspective the modeled system is put into motion and and we sit on the outside and monitor its performance for a period of time. At the end of that interval we intervene to vary the number of users generating a workload, or to change the scheduling algorithm of the system and set it back in motion again.

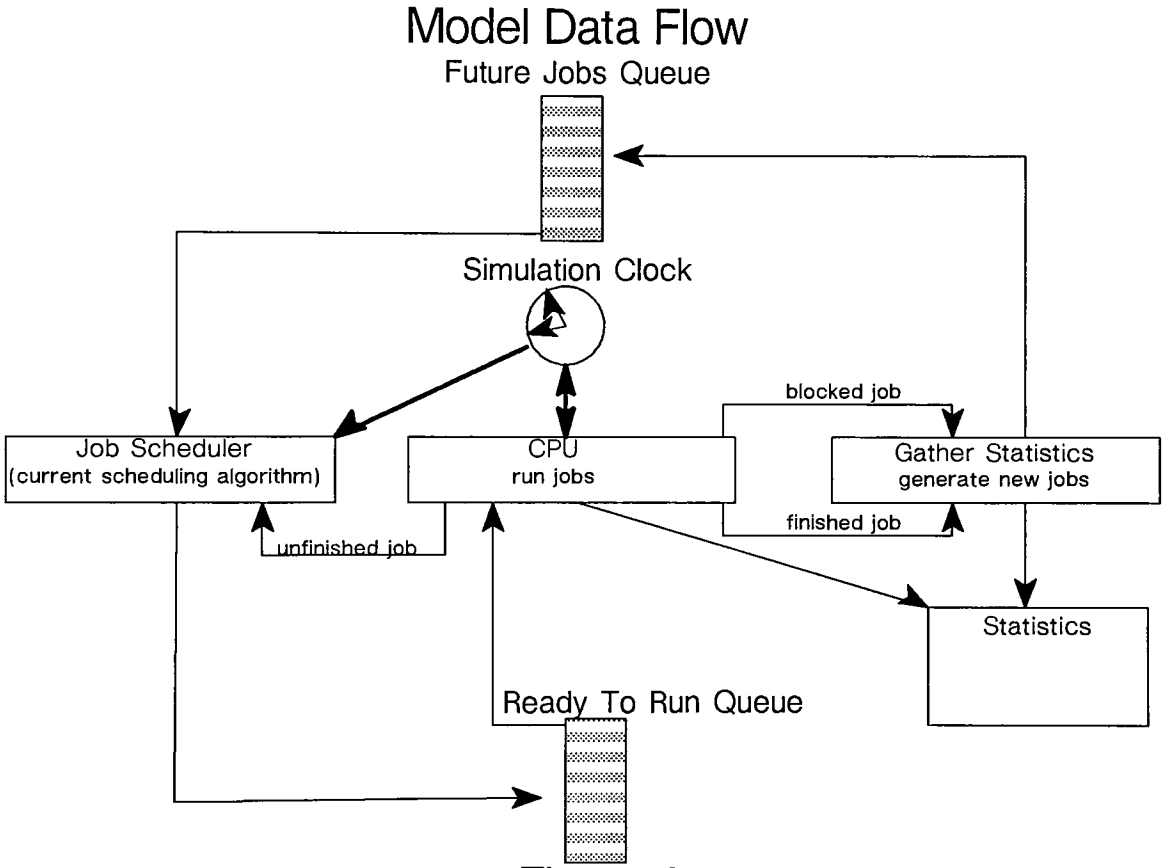


Figure 3

The Model Data Flow diagram, shown in figure 3, looks very similar to the System Data Flow diagram, shown in figure 2. In the place of a group of users there is a queue of future jobs that would be submitted by those users if they existed. It is a convenient modeling technique to maintain a queue of future events and allow the model to digest those events and create more future events as its notion of the present advances. There is an additional box that hangs off of the

side in the model that isn't present in the System Data Flow diagram. This box takes the finished jobs and gathers system performance data from them before disposing of the job and creating the next new job to place in the Future Jobs Queue.

The two central entities within the model are jobs and queues. The model maintains two queues of jobs. There is a future jobs queue that represents the jobs that would be created by a specified number of users on the system that is being modeled. The second queue is the ready-to-run queue maintained by the job scheduler. When the simulation clock reaches the time when one of the future jobs was due to arrive then the job scheduler takes that job out of the future jobs queue and places it in the ready-to-run queue on the basis of the current scheduling strategy.

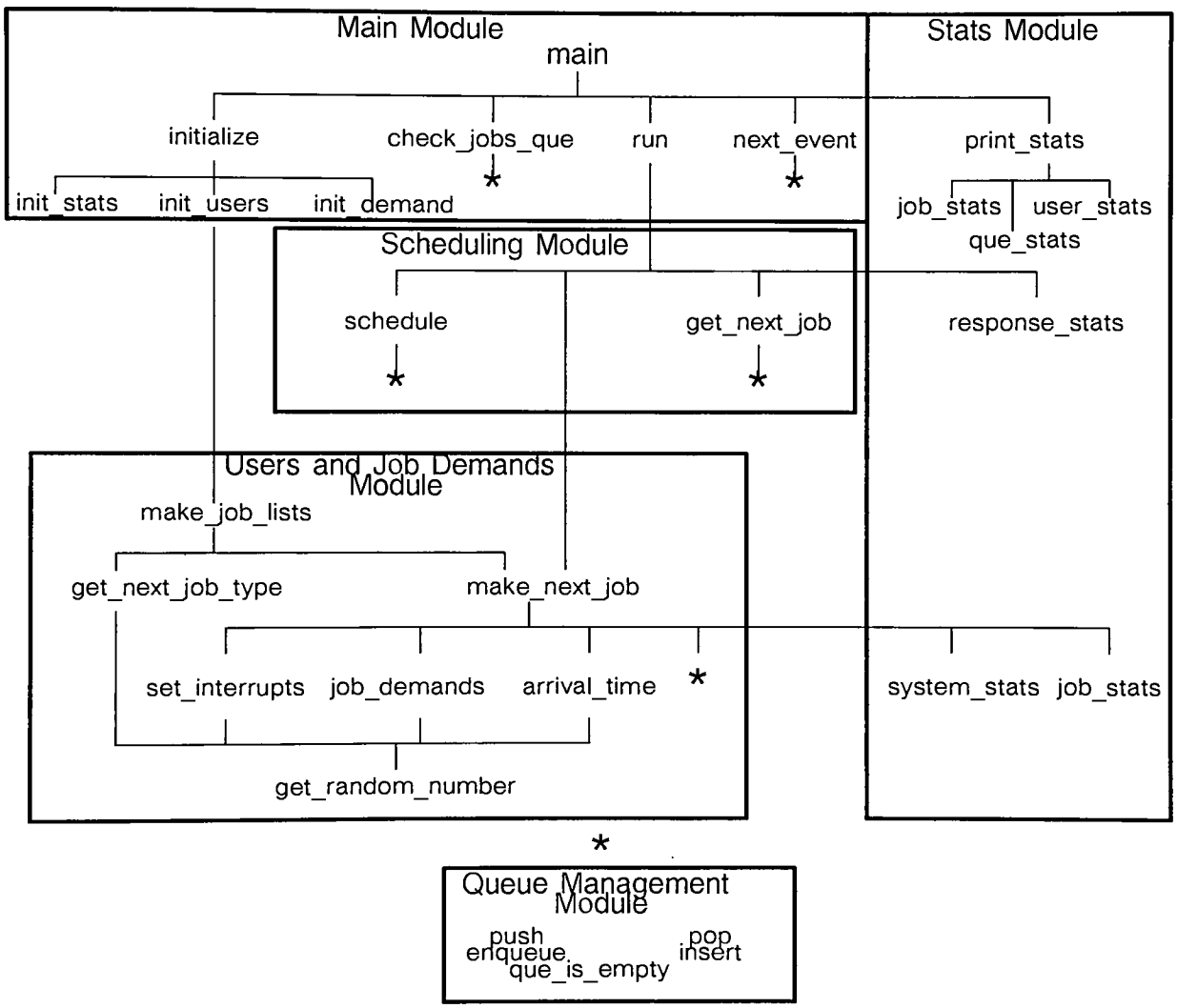
To "run a job" it is removed from the head of the ready-to-run queue and the simulation clock is advanced by either the system time quantum or the unmet service needs of the current job, which ever is less. If a job is then finished, statistics can be gathered from it. Otherwise, it must be returned to the ready-to-run queue by the job scheduler.

3.2. Organization of the Model

From the perspective of the modeler, the simulation is little more than a network of inter-connected queues containing job descriptions. For this reason one of the first tasks in implementing a model is to develop a Queue Management Module containing a set of tools for manipulating and managing queues. At the bottom of the Functional Hierarchy diagram is a Queue Management Module containing these tools. Most of the remaining modules in the model make use of the queue management tools provided in this module. The clustering of queue

manipulation functions in this one module allows data hiding to be implemented and enforced. The complexity of the queue structures and their manipulation is all hidden away inside this module. By passing a queue name to these queue manipulation functions the same functions can be used to operate on any of the queues needed in the model.

Function Hierarchy Diagram



The Main Module controls the flow and function of the entire model. After the Main Module determines how many users to simulate the model can be initialized with the first job for each user placed in the Future Jobs Queue. Once

the model is initialized the simulated target system can begin "running" until we reach the time limit set for the simulation at which point the statistics gathered are printed out.

The Scheduling Module contains two functions which are needed by the Main Module as it "runs" jobs. These two functions are a matched pair which implement a particular scheduling algorithm. This allows further information hiding. Jobs can be run without concern for how the scheduling is being implemented. Waiting jobs may in some algorithms be kept in multiple queues. Only the functions to schedule a job and to get the next job to be run need to know of the inner details of how the jobs wait to run.

3.3. Implementing the Model

For some academic and also very pragmatic reasons I have chosen to implement the model using the 'C' programming language. It is important to maintain good modularity in the implementation of this model. In particular I wanted to be able to keep the Scheduling Module as separated and autonomous as possible. This separation was viewed as critical to the success of the project so that alternative scheduling algorithms could be easily plugged into the model and evaluated. It is important to be able to change the scheduling algorithm without that having any effect on any other part of the model. The 'C' programming language provided the capability of maintaining separately compiled modules that even have local unshared data.

Secondly, I wanted to avoid the computational overhead of recompiling all of the code for the entire model each time a change was made in one isolated segment of the model. I knew there would be a need to make changes in isolated portions of the model, particularly in the segment that scheduled jobs. No matter

what computer system I implemented the model on I wanted to avoid the costly overhead of recompiling large segments of code, the majority of which was unchanged since the previous compilation. I have already taken to heart the reality of soaring response time as real systems are loaded with large compilation jobs.

While general purpose simulation languages, like GPSS, provide many useful tools for setting up the queuing networks and gathering statistics, their use would also mean that the model would have to be created and executed on one of the R.I.T. UNIX systems. My previous experience with GPSS models was that they are extremely time consuming when they execute, and I was envisioning running the model many times with a variety of scheduling algorithms. Again the response time realities seemed to point away from imposing these heavy loads on any system that was shared by very many other people--for my sake and theirs!

A further reason for choosing not to use a general purpose simulation language was an awareness that in real systems scheduling algorithms were often implemented in 'C' or a language like 'C'. By using 'C' it will also be possible to observe the relative coding complexity and efficiency of the various scheduling algorithm implementations. One of my further goals is for the scheduling module contained in this model to replicate closely actual scheduling modules that could be plugged into a real operating system. The job schedulers will be manipulating pointers to job structures. The contents of those structures would be quite different for the purposes of the model, but their manipulation would be very similar in an actual operating system. The structures that are pointed to would be different, but they would be manipulated in the same way. This parallelism between the model and an actual implementation added to the appeal of using 'C' for writing the model.

Some more pragmatic reasons for selecting the 'C' programming language are accessibility and familiarity. Both at work and at home I have access to 'C' compilers. I can do all or the majority of the writing, debugging, calibration and running of the model on my home computer system--a Digital PDP 11/23 (Professional 350) operating under a VENIX operating system. The source code for the model could easily be transferred from this system to one of the R.I.T. UNIX systems, or to one of the Sun UNIX workstations at work if necessary.

The second pragmatic reason I stated was familiarity. I know 'C' better than any other programming language. I have to use 'C' regularly at work and wanted to gain further experience in designing and implementing a large modular software project in this language.

4. Validation and Verification of the Model

The validation of this model involved examination of the model itself, as well as the input, the output, and the original data gathered from real systems used as a benchmark for calibrating the model. The process of validating the model required numerous iterations of a cycle that included statistical analysis, refinement or correction of the model, and regeneration of output. Successive iterations of this cycle uncovered and corrected problems that were found in the output of early versions of the model. For example, in calculating the standard deviation of the response times, the model was accumulating the response time and the square of each job's response time as integers. Some of the standard deviations reported in the early output did not make sense intuitively. For example, some job types reported a worst case response time that was much larger than the average response time. In and of itself this wouldn't be cause for alarm, were it not for the the relatively small standard deviation that was also being reported. Upon closer examination it was determined that the sum of squares that was being accumulated was exceeding the maximum integer value that the system could store. When the internal accumulation of the sum of squares was changed to a double precision floating point value this discrepancy in the output was corrected.

4.1. The Modeled Commands

The data gathered from the Sun system included 97 different job types or commands. For obvious reasons this many different job types could not be represented within the model. An analysis of this original data revealed that 84% of the jobs consisted of a set of just 10 job types. The model uses twelve job types to represent the workload generated by a "typical user". These twelve job types include the ten job types that accounted for 84% of the real sample, plus two conglomerate job types labeled "heavy" and "light". Jobs from the original sample

were clustered into these two types on the basis of the relative demands they place upon the system. The twelve job categories selected and their characteristics are listed below.

ls — The "ls" command and its derivatives list the files and sub-directories contained within a directory. This job involves no user processor time, but includes a consistent and brief system I/O demand.

cd — The change directory command is a very short burst of system book keeping. Included with this command were essentially similar commands, pushd and popd, which change the current working directory and also manipulate a stack of directories maintained for the user's convenience.

vi — The visual editor is an interactive screen editor. The duration and demands of editing jobs varied more widely than any other command. The range of real elapsed times that these jobs recorded ranged from 30 seconds to a full hour.

"light" — The "light" category includes such UNIX commands as: pwd, rm, history, mv, set, mkdir, date, touch, chmod, and clear. All of these commands are characterized by very light demands for system resources.

make — The frequently used make command is the real processor hog in the measured user population. The UNIX Makefile system keeps track of file dependencies and recompiles object files when ever source files are changed which they depend upon. This updating process often involves compilation, linking and archiving of multiple files. As a general rule they tend to require more user processor time than many of the UNIX commands found in the other categories.

more — This command displays the contents of a selected file one screen full at a time, pausing between screens until the user requests the next section.

run — This category combines a variety of executable programs created by the users. In a software development environment this represents the execute part of the edit-compile-execute cycle that is repeated so frequently. These run modules range from simple component test drivers to very elaborate complete programs.

cp — This is the copy command. In this particular programming environment it is used much more often than the printing of hard copies. When a

module under development is completed it is copied into a central project directory rather than being printed out.

grep — This command causes a directory to be searched for occurrences of a particular pattern. It is frequently used to search through the large project directory for particular information. It is a high demand job that potentially requires significant user and system processor time.

"heavy" — The "heavy" category includes cpu intensive UNIX jobs such as: find, diff, ps, what, and df.

mail — This is the electronic mail system. In this environment most of the mail comes from automated background system processes, for example reporting the successful completion of automated nightly backups or reminders gleaned from the user's appointments calendar. Mail is usually read once immediately after logging in and not executed again until the next day.

lpr — The lpr command causes files to be queued up for printing by the system. On the systems that were measured for this study printing of hard copy is rather rare. Most of the programmers have large Sun monitors on their desks capable of displaying multiple windows. This capability significantly reduces the need for printed hard copy. One source can be displayed while another is being edited.

4.2. Validating the Input

Before the output of the model could be validated, the input to the model had to be validated. The amount of processing time that each job type would require within the model was derived from a large sample of jobs run on a Sun 2/120 workstation. A "job demands" file is read into the model at initialization time. This file contains a set of ten job demand values for each job type represented in the model. A copy of this file is included in the appendixes. If this sample of ten input values was not representative of the processor time required for these jobs on the target Sun system then there would be no hope of the models output being valid.

The 2016 jobs recorded from the target Sun system divide into the twelve job categories selected for the model's input in the following proportions:

type	percentage	average response time	standard deviation
ls	19.5	.30	.21
cd	17.2	.15	.04
vi	15.5	7.95	8.69
"light"	14.0	.295	.47
make	11.3	41.83	38.86
more	6.7	.425	.15
run	6.1	3.86	6.15
cp	3.6	.88	.89
grep	2.7	13.6	14.38
"heavy"	2.1	11.15	11.17
mail	0.8	1.8	.93
lpr	0.6	5.83	3.26

The response time data gathered from the Sun 2/120 was separated into these twelve job categories. The mean and standard deviation for the response time of the Sun 2/120 was calculated for each of the twelve job categories that are represented in the model. Ten typical values were selected from Sun data for each job category. The mean and standard deviation of each set of ten values was checked against the mean and standard deviation of the data gathered from the Sun 2/120. Different subsets of the full set of Sun 2/120 data were tried until the mean and standard deviation of the sets matched that of the Sun data. These twelve sets of ten job demand figures are found in the "job.demands" file in Appendix B which drives the job demands of the "typical user" workload within the model.

4.3. Validating the Output

With the input data calibrated against a real workload on the Sun 2/120 there was a better chance of obtaining valid and meaningful output. The next step in

validating the model was to run the model with one user under each of the scheduling algorithms to obtain verbose output of the demands for each job as it completed. The response times of the model running with one user were compared with the response times of the Sun 2/120 with one user. For non-interactive jobs which do not depend on terminal interruptions, the comparison of response times was done using t-tests. The data gathered from the Sun 2/120 reports user cpu time, system cpu time, total elapsed real time, plus additional information on memory and I/O demands. The total elapsed real time displayed by the Sun system was used as the response time value. For non-interactive jobs this represents job turn-around response time from the time the command was issued until the system was ready to accept the next command. T-tests were performed to determine whether the real system and the modeled system had the same average response times. Results of these t-tests indicate that at the p=.05 level of certainty the model and the Sun samples do not differ for non-interactive jobs.

For example, for the "ls" command a t-test was made of the difference of means between the data gathered from the Sun 2/120 and the data gathered from the model. A value of t = 0.7664 was calculated for the "ls" command. This value does not exceed the cut-off value of 1.645 for a p=.05 level of certainty. These tests confirm that the model's output closely matches that of the target Sun system. The following table sumarizes some of the calculated t-tests made to validate the output of the model:

command	calculated value for t	degrees of freedom	table cut-off for p = .05
cp	t = .1984	df = 17	1.740
make	t = .4722	df = 59	1.645
ls	t = .7664	df = 40	1.645
light	t = .0036	df = 40	1.645

For interactive jobs it was necessary to make a different comparison to verify the calibration of the model workload. The value reported by the Sun system for real elapsed time represents the total duration of the editing session, and the amounts of user and system processor time that were distributed over that span of real time. The total cpu demands for these interactive jobs was correct from the outset. It matched closely the values measured on the Sun system. However the model's editing sessions initially passed without interruptions, meaning the elapsed "real time" nearly matched the "processing time". This was an unacceptable condition, because one "typical user" was generating a tremendous load on the model of the system, consuming 80-90% of the processor's time. The model clearly needed to make provision for a waiting state where interactive jobs waited for terminal input. The model now divides the processor demand time up into a series of short job bursts. The model has an "editing think time" built into it which represents the time the user spends thinking at the terminal between these short bursts. The value of the "editing think time" was adjusted until the duration of a modeled interactive job closely matched the real time duration of the corresponding jobs in the real world. In the data gathered on the Sun 2/120 system, vi editing sessions were found to have a mean duration of 8.61 minutes of real time. The "editing think time" was adjusted until the average duration of interactive jobs matched closely that of the real users'. The average modeled editing job has a duration of 8.77 minutes. A t-test of these values yields a t of

.04912 with 44 degrees of freedom. Again, these values indicate a very close correspondence between the output of the model, and values found on the reference Sun system. With these changes made for the model's handling of interactive jobs a "typical user" generates a workload which consumes 1–5% of the available cpu time. This value agrees closely with the total cpu utilization of users on the Sun systems.

4.4. Validating the Job Mix

> "W' is a perfectly representative model of W if it demands the same physical resources in the same proportions as W." "W' is a perfectly representative model of W if it demands the same physical resources at the same rates as W." [Ferrari, et. al. 1983]

These two statements summarize Ferrari's criteria for the validation of a computer model and its workload. This project involves a simulated workload with multiple job types. A paraphrase of Ferrari's criteria for this model might be: "W' is a perfectly representative model of W's workload if it performs the same functions in the same proportions as W." The validation of the model so far has focused on the cpu service demands of the individual job types within the model. It has been shown that the model demands the same cpu resources to perform a particular job as are required to perform that job on the target system. It remains to be shown that the mix of jobs in the model matches that of the real population.

The distribution of jobs under each of the modeled scheduling algorithms was checked against the distribution of jobs in the original data gathered from the Sun 2/120 users. Chi square goodness–of–fit tests were performed on the output from each of the three scheduling algorithms to determine whether the simulated workload had a comparable workload to that measured on the target system. The chi square goodness–of–fit test for the round–robin scheduler's output yielded a

result of 10.613. The chi square cut off point for twelve cases at p = .05 is 19.675. The calculated value of 10.613 passes this goodness-of-fit test.

The results for the other scheduling algorithms were similar. A chi square goodness-of-fit value of 11.926 was calculated for the shortest-job-first scheduler. For the priority scheduler a chi square value of 14.823 was found. Both of these values indicate a satisfactory fit within the limits of sampling error at a .05 level of certainty.

5. Results and Discussion

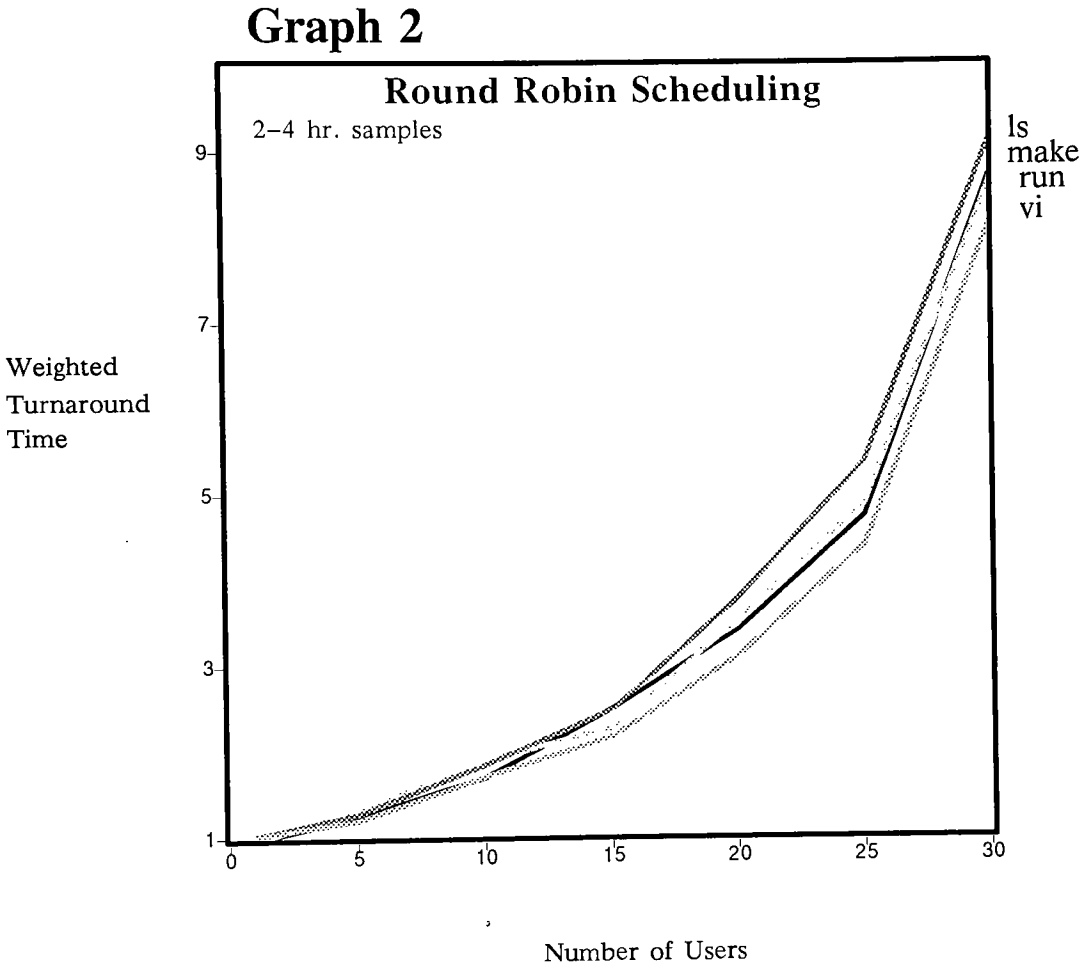
5.1. Generating The Data

After completing the previously described steps to validate the output of the model, five complete sets of data were generated using different random number seeds. Each set of data consisted of running the model using each of the three scheduling algorithms, and a series of increasing workloads. Each of the algorithms was run with increasingly heavy workloads, consisting of greater numbers of users, until the saturation point of cpu utilization was determined. Thirty-five users completely utilized the cpu with each of the three algorithms. With thirty users the cpu utilization was in the range of 94-96% for all three scheduling algorithms. Ninety-five percent utilization of a modeled facility is generally regarded by modeling standards to be the saturation point. Thirty users is the maximum workload that was used in generating the data used for any of the comparative graphs and discussion. For each of the three algorithms, the model was run with 1, 5, 10, 15, 20, 25, and 30 users. During the generation of the first three sets of data one hour of simulated time was measured for each run. For the last two sets of data the simulated time was increased to four hours. The weighted average from these final two sets of four-hour runs was used in preparing the graphs which accompany this report.

5.2. Round-robin

Simple round-robin scheduling is the "fairest" of all the algorithms. This algorithm does not assign priorities to jobs in any way; it gives each job an equal turn in using the cpu. As one would expect, and as Graph 2 indicates, all of the job categories display an equal degradation in weighted turn-around time. There is a uniform deterioration in the performance of the system across all job types. The

four job categories which were plotted on the graph have nearly identical values. With round-robin scheduling no person or job category fares any better or any worse than any other.



A further measure of the non-discriminating nature of this scheduling algorithm was the distribution of job types that were left in the ready-to-run queue when the model terminated. The ending state of the model represents one sample of the mix of jobs that might be found waiting to run at a random moment. Under lighter loads the system population statistics indicate that the ready to run queue was empty much of the time. But under the maximum load of thirty users there

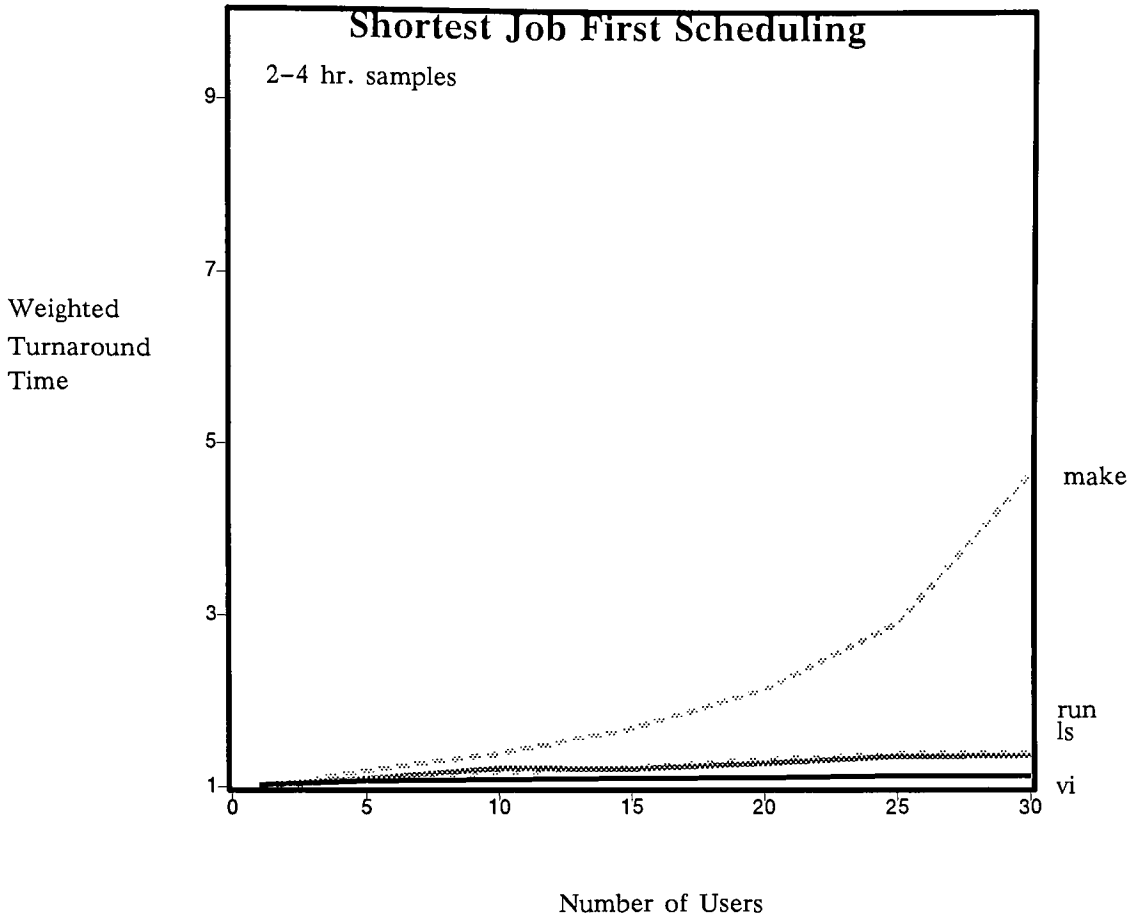
were remaining jobs in the ready-to-run queue when the model terminated. Among the 17 jobs remaining in the system, there were 13 "makes", and one each of "grep", "lpr", "vi", and "ls". One would expect to find a predominance of the lengthy and common "make" job type in the queue at any given time. It is noteworthy that a wide variety of other job types are also represented in this sample.

In terms of variation in the average response time, the round-robin scheduler was the worst of the three algorithms. The average response time varied widely. With this algorithm the response time of a job submitted at any instant depended as much upon what other jobs, and how many other jobs, were ready to run at that moment in time.

5.3. Shortest-Job-First

The shortest-job-first scheduling algorithm exhibits a very different pattern of weighted turn-around times from that found in the previous example of round-robin scheduling. This algorithm does not treat all jobs equally. There is a differential degradation in response time across job types that depends upon the cpu demands imposed by that job type. Graph 3 contains the weighted turn-around time for the shortest-job-first scheduler. Jobs with very light demands and short cpu bursts are given preferred treatment by this scheduler. The response time for the very short cpu burst of the vi editor was nearly ruler flat from 1 user all the way to 30. Other light demand jobs like "ls", which lists the contents of a directory, also showed only a slight increase in weighted turn-around time. But, heavily cpu intensive jobs, like "make", which often involves multiple compilations, bore the brunt of the burden of slowed system response. The average weighted turn-around time increased almost five-fold as the load on the system was increased from 1 to 30 users.

Graph 3



When we look at the worst case response time the penalty paid by the cpu intensive "make" job type is even more dramatic. The worst case response time climbed from a low of 97 seconds, when a single user was using the system, to a high of 57 minutes (3459 seconds). This is almost a forty-fold increase. User's perceptions of the performance of interactive computer systems is greatly colored by worst case performance. It is worst case scenarios like this one which become the horror stories which circulate around university computer labs as the end of the semester deadlines push the load on the system higher and higher.

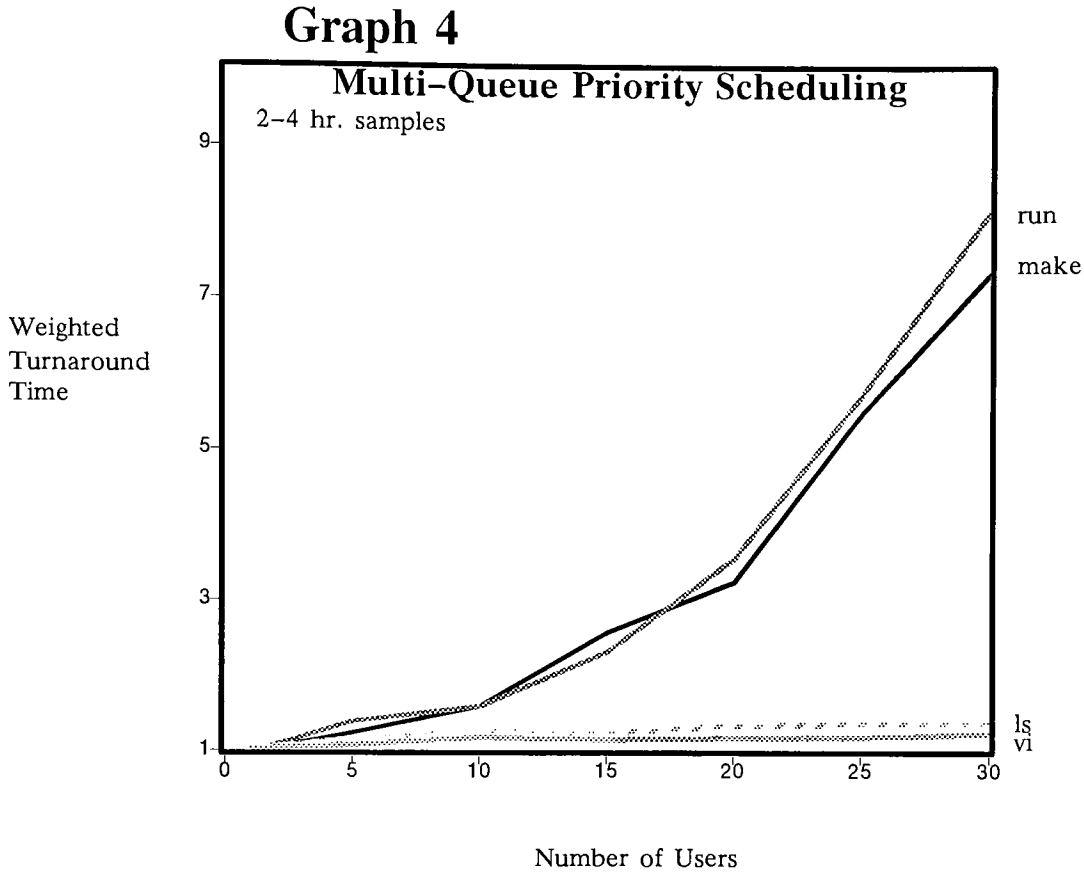
Ironically, this scheduling algorithm is a "procrastinator" just like a number of those students who fall prey to the declining performance of the system at semester's end. This algorithm delays tackling the big, hard (cpu intensive) tasks, in favor of working on more of the short simple ones. Evidence of this "procrastinating" behavior can be found by examining the jobs left in the ready-to-run queue when the model terminates. With the maximum load of thirty users on the system there were 3 "makes" and 1 "vi" job left in the queue. Needless to say, the "vi" job was the first in line in the queue. Similarly with 25 users the queue had 6 jobs left it, 5 "makes" and 1 "run". With 20 users the model ended with 4 "makes" in the queue. A closer examination of these jobs remaining in the ready-to-run queue under heavy system loads reveals that these jobs are nearing starvation. They are receiving service, but at a very slow rate.

While the shortest-job-first scheduling algorithm may appear to be a "procrastinator" in terms of completing those jobs with heavy demands, it is "productive". It was observed that under heavy loads the total job throughput for this algorithm was consistently higher than the round-robin and priority schedulers. The shortest-job-first algorithm's throughput was about 3-4% greater than the other two algorithms. This was generally a consistent trend, though it was not statistically validated. It completes more jobs per hour than other algorithms because it continually focuses its efforts on completing the short, easy ones first.

5.4. Priority Queueing

The multi-queue priority scheduler gave priority to interactive, and short burst jobs such as "vi", "ls", "more", "cd" and "light". These jobs were placed in a higher priority queue which was serviced first. When this queue was empty then other job types placed in the lower priority queue would receive service. The performance characteristics of this scheduler are interesting. The interactive, high

priority jobs experienced minimal degradation in turn-around time, as Graph 4 shows. The lower priority jobs like, "make" and "run" show a very pronounced increase in turn-around time. The "run" job type saw an increase from an ideal weighted turn-around time of 1.00 for a lone user, to a high of 8.19 when thirty users were on the system. One could think of weighted turn-around time as the ratio of how long job execution actually takes to how long it should take under ideal circumstances. The degradation in weighted turn-around time was worse for this "run" job type than for the more demanding "make" job category. That is to say this job category was penalized more than its needs for cpu time would lead one to expect that it should have been. This is understandable given how the priority scheduler treated the "run" job type. The "run" job type was thrust into the same low priority queue with the very demanding "make" job type, yet in terms of its total cpu demands it falls into an intermediate category, between the high and low priority queues. Perhaps in a multi-level queueing algorithm that offered a finer granularity it would not have suffered such a severe degradation in performance.



Given the fact that this is a priority scheduler, one would expect to find the lower priority job types left in the ready-to-run queue when the model terminates. With 25 users there were 2 jobs left in the queue, and with 30 users there were 5 jobs left in the ready-to-run queue. All were "make" jobs. This algorithm postpones work on these demanding jobs, which are going to take longer to complete under any circumstances because they require more processing, in favor of working on the short, less demanding jobs. In this regard it is like the shortest-job-first algorithm.

6. Comparing The Scheduling Algorithms

For the purposes of comparing the various scheduling algorithms the same data was regrouped on the basis of job type, rather than scheduling algorithm as was done in the previous set of graphs. This regrouped data is plotted in Graphs 5 through 8. Accompanying these four graphs are four tables containing the data that is plotted in the graphs depicting weighted turn-around time for each of the three algorithms. In addition, a third set of graphs was prepared which compares the worst case response time of the various algorithms within each job type (Graphs 9-12). Note, that the scales for worst case response time are different from one graph to the next. Bear in mind that a worst case response time statistic represents a single value out of the many samples collected. It is not a powerful statistic in terms of characterizing the performance of these algorithms, however, it is an interesting statistic to keep track of in a comparison of these algorithms in two regards. First, it is a good indicator of the onset and severity of starvation that a particular job type may be experiencing. Secondly, as has been noted, real users seem to be very sensitive to this measure of system performance in their subjective evaluation of a system's performance. If the worst case far exceeds the average response time then user's reported satisfaction declines markedly.

Graph 5

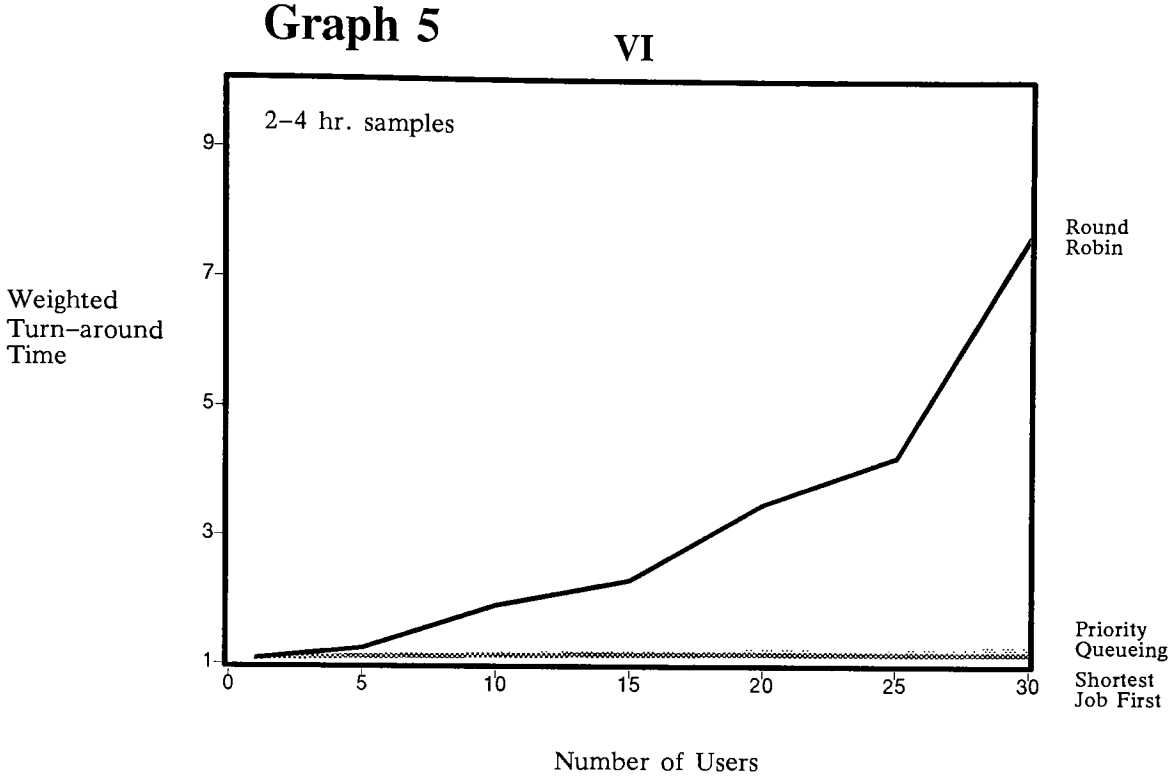


Table 1

users	round–robin	shortest–job–first	priority queueing
1	1.11	1.11	1.11
5	1.27	1.14	1.14
10	1.94	1.16	1.18
15	2.33	1.17	1.20
20	3.52	1.18	1.23
25	4.25	1.18	1.22
30	7.69	1.19	1.29

weighted turn–around time for vi

Graph 6

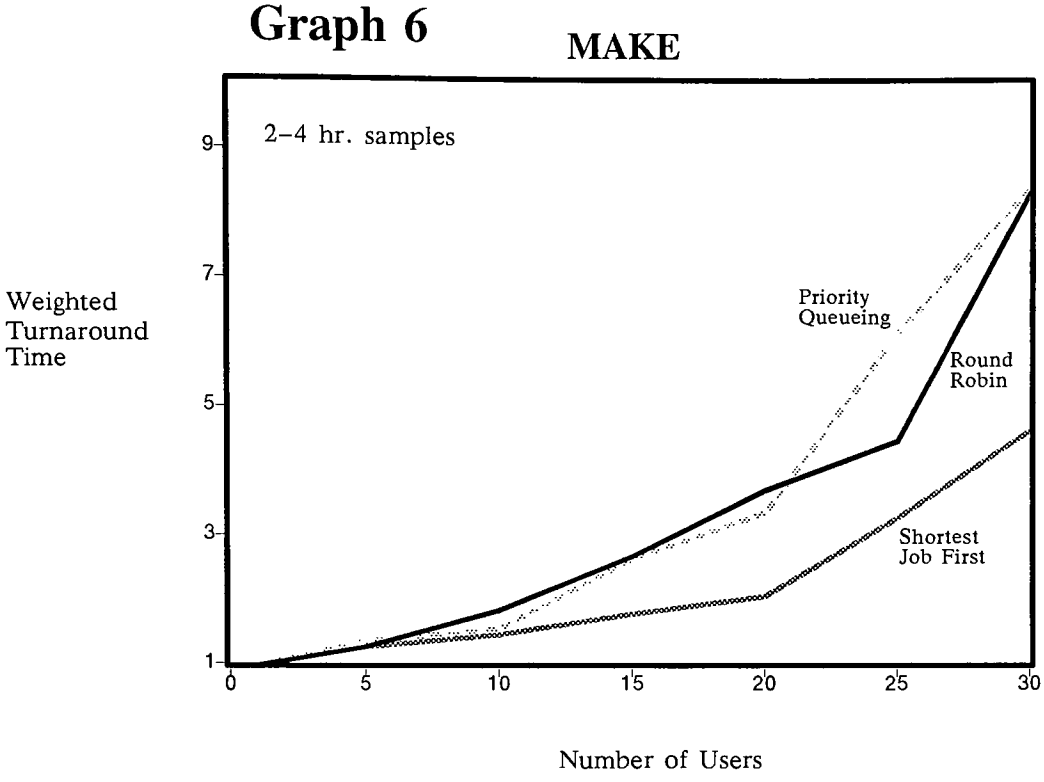


Table 2

users	round robin	shortest–job–first	priority queueing
1	1	1	1
5	1.28	1.28	1.37
10	1.85	1.47	1.56
15	2.69	1.79	2.65
20	3.72	2.07	3.37
25	4.49	3.28	6.17
30	8.35	4.65	8.40

weighted turn–around time for make

Graph 7

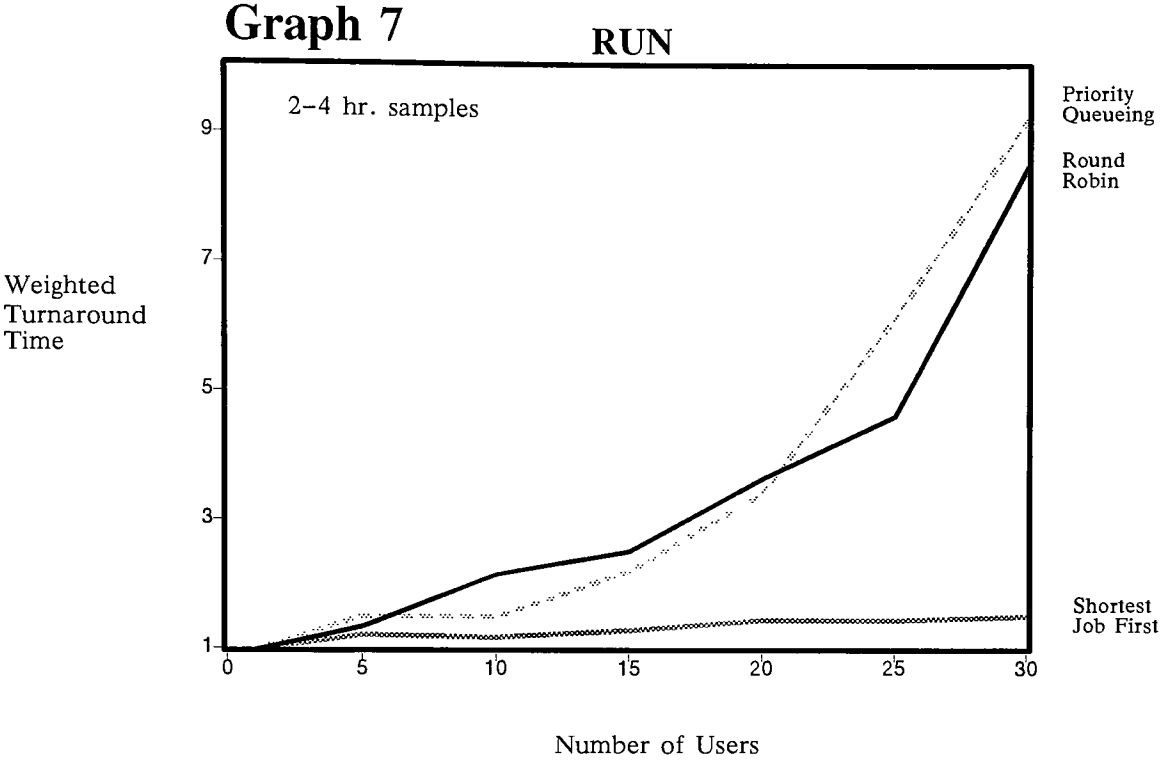


Table 3

users	round-robin	shortest-job-first	priority queueing
1	1	1	1
5	1.36	1.23	1.53
10	2.17	1.19	1.51
15	2.53	1.30	2.22
20	3.66	1.46	3.44
25	4.64	1.46	6.17
30	8.52	1.52	9.27

weighted turn-around time for run

Graph 8 LS

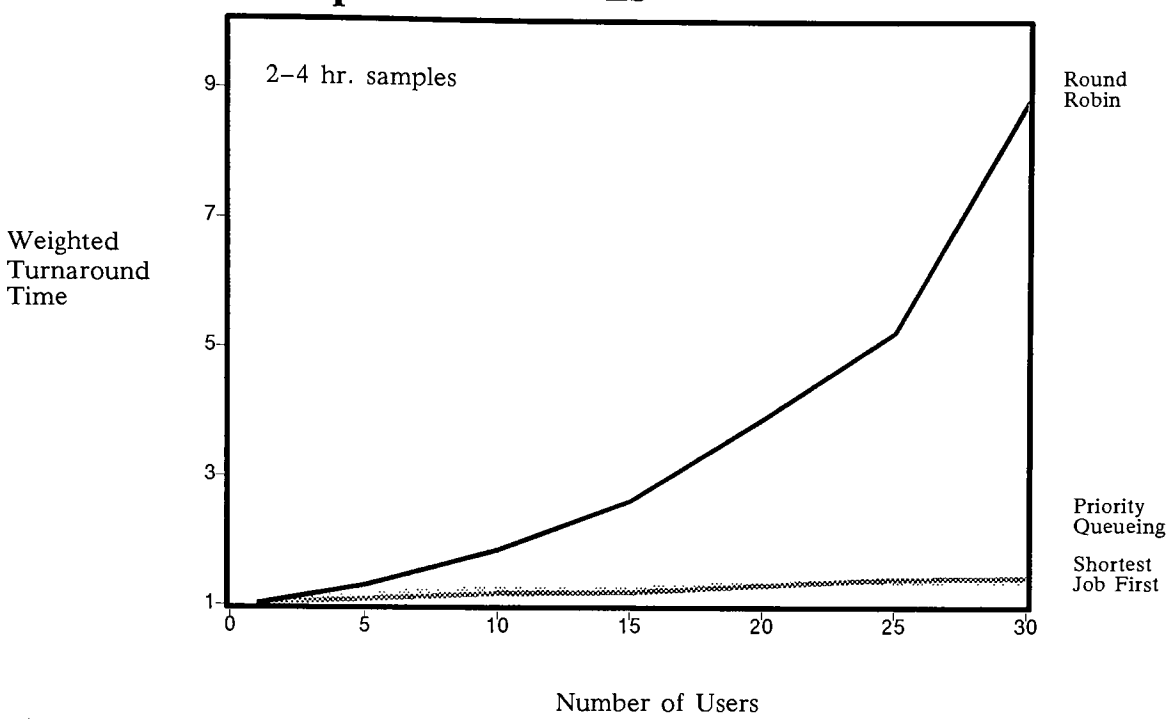


Table 4

users	round-robin	shortest-job-first	priority queueing
1	1.05	1.05	1.05
5	1.34	1.12	1.20
10	1.89	1.21	1.29
15	2.65	1.23	1.29
20	3.93	1.34	1.38
25	5.28	1.44	1.40
30	8.86	1.46	1.39

weighted turn-around time for ls

Graph 9 VI

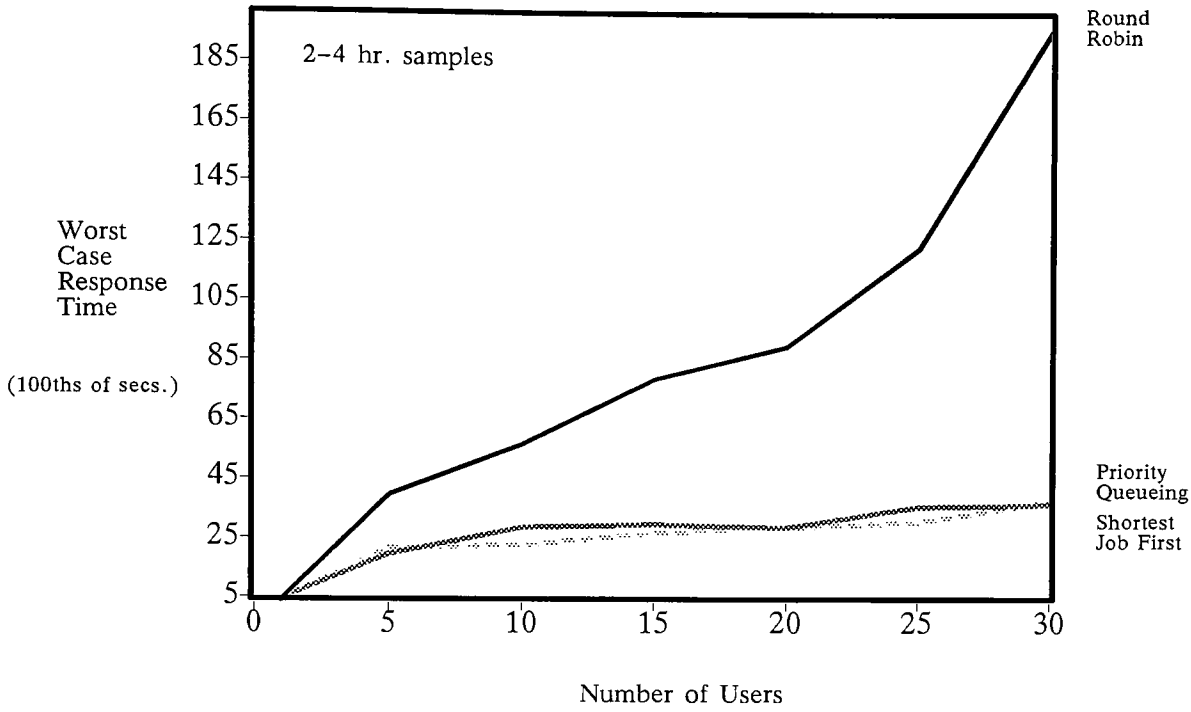


Table 5

	Round Robin	Shortest-Job-First	Priority Scheduler
1	5	5	5
5	40	20	22
10	57	29	23
15	79	30	27
20	90	29	29
25	123	36	31
30	196	37	38

Worst Case Response Time for vi job type
(in 100ths of seconds)

Graph 10 MAKE

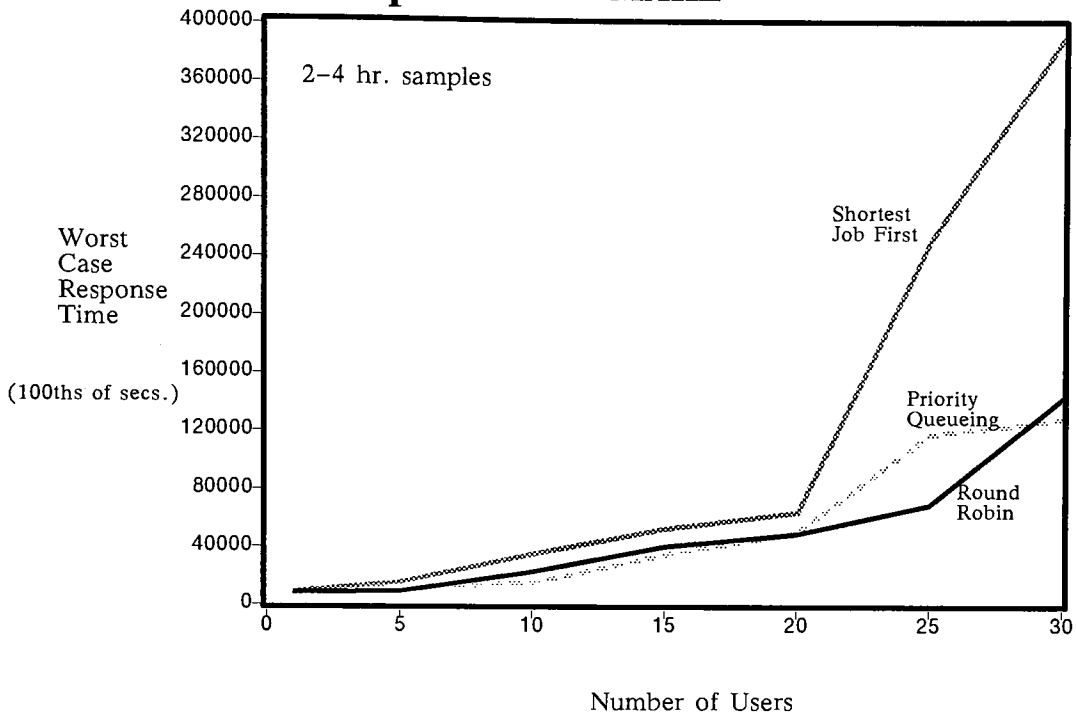


Table 6

	Round Robin	Shortest-Job-First	Priority Scheduler
1	9760	9760	9760
5	10168	16628	13255
10	23916	36146	15917
15	41427	53624	35672
20	50154	65018	50796
25	70048	252278	118903
30	144187	392957	129762

Worst Case Response Time for make job type
(in 100ths of seconds)

Graph 11

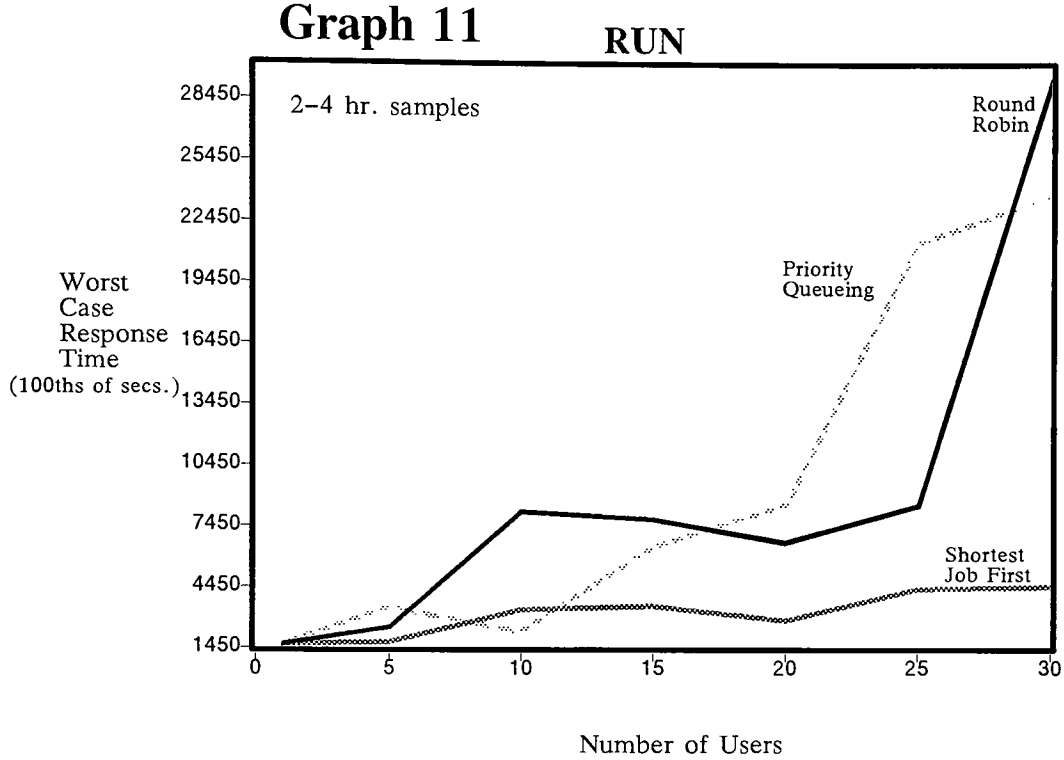


Table 7

	Round Robin	Shortest-Job-First	Priority Scheduler
1	1730	1730	1730
5	2539	1810	3540
10	8248	3391	2309
15	7866	3580	6499
20	6734	2865	8600
25	8545	4435	21500
30	29557	4557	23875

Worst Case Response Time for run job type

(in 100ths of seconds)

Graph 12 LS

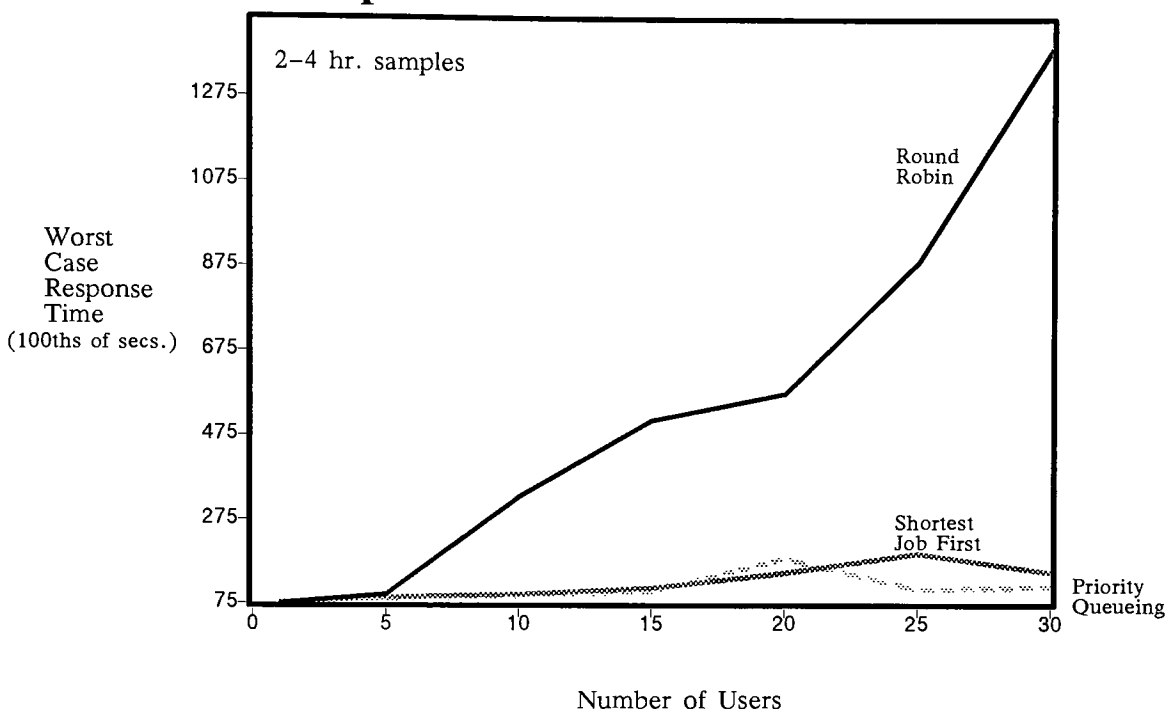


Table 8

	Round Robin	Shortest-Job-First	Priority Queueing
1	80	80	80
5	100	92	86
10	331	100	96
15	513	115	108
20	577	151	184
25	889	197	113
30	1394	152	117

Worst Case Response Time for ls job type
(in 100ths of seconds)

A lot of weight shouldn't be given to any comparisons based upon worst case response time. However, it is interesting to note the relative performance of the priority scheduling algorithm based upon this measure. For the short burst, interactive type jobs the worst case reports for the priority scheduler closely track those of shortest-job-first scheduler. But, interestingly, when we look at the "make" job type, the worst case performance for the priority scheduler does not deteriorate nearly as badly as the shortest-job-first worst case. For these lower priority job types the performance of the priority queueing algorithm more closely matches the round-robin scheduler than it does the shortest-job-first scheduler. This isn't a total surprise, since under the hood the priority scheduler is implementing a round-robin scheduling algorithm among the jobs of the same priority. Based upon worst case performance alone the priority scheduler would fare well in a comparison of these three algorithms. It maintains relatively uniform response for short-burst interactive jobs, without starving the worst "make" job as severely as the shortest-job-first scheduler.

The safer ground for making a comparison of the relative performance of these algorithms is the weighted turn-around time. On the basis of this performance criteria we can see that the shortest-job-first algorithm consistently performs better than the other algorithms in every job category. In most of the job categories the priority scheduling algorithm turned in a remarkable performance. One notable exception to this good performance rating were those job types which had moderate cpu demands such as the "run" job type. The priority scheduling algorithm penalized the "run" job type by placing it in the low priority queue, where it was forced to share time with some jobs that were much more demanding that it was. The shortest-job-first algorithm, by contrast, was able to keep the "run" job type's execution time more in line with its real needs. Jobs that are just

over the cut off for the next higher priority will be most severely penalized in these priority based schedulers.

An analogy would be the experience of heading for the checkout lines at the grocery store and discovering that you have eleven items in your cart. The limit for express checkout is ten items. Being one item over the limit forces you into line behind the parent with two mounded carts full of groceries. Your weighted turn-around time will be higher than the parent with two carts full under the priority queueing scheme at that grocery store. That's a statistician's way of saying that you will spend a greater portion of time just waiting in line than you will actually checking out. For the person with the two full carts, the opposite is true, this person will spend proportionately more time actually checking out than waiting in line. This ratio of the productive part of the delay to the unproductive part of the total delay is what weighted turn-around time represents.

For the person who uses a computer to edit files or perform other interactive tasks the choice of scheduling algorithms is easy. This person will clearly favor one of the algorithms that offers differential degradation in system performance. In order to retain uniformly quick response while editing files this person would accept a greater increase in the processing of large compilations and other cpu intensive jobs.

The first choice algorithm would be the shortest-job-first scheduler. But, this algorithm is not implementable at a short-term cpu scheduling level. It was possible in the context of a controlled model, but it can only be approximated in a real system. The round-robin scheduler is clearly the least satisfactory--its response time for interactive jobs deteriorates markedly, and its variation in response time was the greatest. Round-robin has little to recommend it beyond its simplicity.

The multi-queue priority scheduler turned in a surprisingly good performance overall, particularly given the relative simplicity of the algorithm. Its handling of short burst interactive jobs nearly matched that of the ideal shortest-job-first. And for the cpu intensive jobs its worst case performance was closer to that of the round-robin scheduler. The one exception noted to its good performance was those job types with intermediate demands. The area of multi-level queueing algorithms would be a fruitful one for further exploration and development. A multi-queueing algorithm which offered a finer granularity in separating jobs by need would be a promising candidate. Other algorithms worth evaluating would be multi-level feedback queues with aging. Each job would receive an initial high priority quantum of cpu time, and those jobs which didn't complete in the initial quantum would be moved down to a lower priority queue. After spending a period of time in the second level queue a job might even be moved to yet a lower priority. However when the job approaches a certain age it would be rewarded with a return to one of the higher priority queues. This sounds like a fair and workable approach to job scheduling, but it is a costly algorithm to implement in terms of the operating system overhead it introduces.

6.1. The Cost Of Scheduling

Every scheduling algorithm introduces a certain amount of overhead into the operating system. It takes a certain number of machine cycles to execute every algorithm. It is easy to envision more sophisticated scheduling algorithms, which may offer better rates of differential performance degradation across job categories, such as the multi-level feedback algorithms mentioned earlier. With these more sophisticated algorithms comes the cost of increased operating system overhead. Job scheduling occurs at every context switch, which means that this scheduling code is executed a great many times.

One of the limitations of the model was that it did not adjust the amount of system overhead associated with executing a job based upon the complexity of the scheduling algorithm that was currently in place. However during the actual execution of the model it was possible to gain some sense of the relative efficiency of the various scheduling algorithms. To show the differences in the computational complexity of the various algorithms I measured the model's execution time for each of the algorithms tested. Execution time of each iteration of the model was collected for one full set. The following table summarizes the execution time of the model for the various scheduling algorithms.

Model Execution Times
(minutes:seconds)

Number of Users Simulated	Round Robin	Shortest Job First	Priority Queueing
5	1:13	1:26	1:26
10	2:22	2:27	3:05
15	3:22	4:20	3:40
20	3:57	4:33	4:02
25	5:03	5:55	5:25
30	6:05	6:44	6:18

The differences in execution time between the three algorithms can be directly attributed to their differing degrees of computational complexity. The shortest-job-first algorithm while yielding the best response time performance was computationally the most costly of the algorithms. Every time a job was scheduled an insertion sort of the ready-to-run queue was required. This is far slower to execute than queueing the job at the end of the line, as was done with both the round-robin and priority schedulers. The insertion algorithm that was used included an additional heuristic test to handle more efficiently the rather common case where the job belonged at the end of the line, without walking through the

entire queue to find that out. The shortest-job-first algorithm was the most complex algorithm, and required the most cpu time to execute the model. The round-robin scheduler was the simplest algorithm, and required the least cpu time to execute in the model. The priority scheduler was of intermediate complexity, and execution times.

Based on the differences in execution time of the model we could be talking in terms of a 7-10% increase in system overhead for cpu job scheduling to go from simple round-robin scheduling to shortest-job-first. The overhead of the priority queueing algorithm is a better bargain in terms of the improvements in performance it gives in return.

7. Discrepancies and Shortcomings of the Model

The present study is not without its limitations. First, the user population that the model was based upon is involved in the development of a large software project. Individual programmers are frequently linking newly developed modules into the entire system under development. This linking is done quite frequently, and is executed through a large "makefile" shared by everyone that results in an executable file that is about .75 megabytes in size. The execution of this master makefile is a genuine time hog. This accounts for the relatively high frequency of very cpu intensive "makes" in this programming environment. The model is currently calibrated to simulate the job mix and cpu demands of a group of programmers working in this software production environment. In an academic computing environment the size and complexity of the programs generated by the student population would not begin to approach this scale. It would be an interesting follow-up to this study to gather a similar set of calibration data from an academic computing environment and run the model with this different workload.

A second limitation of the model is in the area of background jobs. The user's option of putting a job into the background and executing another job in the foreground is not represented in this model, although it could be. A very low incidence of background processes was found in the software development environment. There are several possible explanations for this infrequent use of background processes. First, there is a tremendous abundance of hardware available, so that usually only one or two programmers, share the same workstation and processor. This abundance of processors keeps the response time, even for the execution of the master makefile, reasonably quick. The second factor contributing to the low incidence of background processes is the extensive

availability of multi-windowing display consoles. Instead of putting the job in the background, work can simply continue in another window at the same terminal. Since response time is generally very good, it is a common practice to execute makefiles and compilations in one of the windows and wait for their completion. If further editing is required this is done in the another adjacent window so that error messages can be left displayed on the screen. Incidentally, this availability of multi-windowing made the gathering and analysis of data more difficult. The data for the order of job execution had to be taken from those programmers who didn't have, or didn't use multi-windowing consoles.

My experience from working in the computer labs of the academic computing environment is that background processes are used much more frequently. This is likely due to the greater number of users sharing the system, and to fact that the terminals available couldn't simultaneously display multiple foreground processes. If the model was to be recalibrated to reflect the behavior of another user population, it might also need to be enhanced to allow it to represent the creation of background processes by users, particularly the placement of large compilations and executions in the background.

8. Conclusion

This cpu scheduling model has provided a useful tool for evaluating the changes in performance of various cpu scheduling algorithms under varying system loads. The model's modular design and calibration that is driven from external data files extends its possible usefulness beyond the current study. Other cpu scheduling algorithms, other target processors, and other user populations could be simulated in a future study by making relatively minor changes to the input files and scheduling modules of the model. The information that can be derived from the output of the model could be useful in projecting the computing needs of many different groups. A particular level of system response time may be selected as an acceptable maximum limit. It is then possible to determine how many users could share a particular system before reaching that limit of acceptable response time. The beauty of computer simulations, once they are calibrated and validated, is that many "what if" scenarios can be explored through the lens of the model. Another entire project could involve a continuation of the work begun in this project. A series of "what if" scenarios could be defined and explored utilizing the model that was developed for this project. A whole new class of scheduling algorithms, designed for the multi-processor parallel architectures now being developed, would make another interesting avenue for future exploration.

9. Bibliography

[Coffman 1968] E. G. Coffman. "Analysis of two timesharing algorithms designed for limited swapping." *JACM* 15, 3 (1968), 341-353.

[Coffman and Denning 1973] E. G. Coffman and P. J. Denning. *Operating System Theory*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[Coffman and Kleinrock 1968] E. G. Coffman and L. Kleinrock. "Feedback Queuing Models for Time-Shared Systems" *JACM* 15, 4 (October 1968), 549-576.

[Ferrari et. al. 1983] Domenico Ferrari, Giuseppe Serazzi and Alessandro Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1983.

[Kameda 1984] H. Kameda. "Optimality of a Central Processor Scheduling Policy for Processing a Job Stream." *ACM* 2, 1 (February 1984), 78-80.

[Kimbleton 1972] S. R. Kimbleton. "The Role of Computer System Models in Performance Evaluation". *CACM* 15, 7 (July 1972), 586-590.

[Kleinrock 1975] L. Kleinrock. *Queuing Systems, Vol. II.: Computer Applications*. Wiley-Interscience, New York, N.Y., 1975.

[Kleinrock and Coffman 1967] L. Kleinrock and E. G. Coffman. "Distribution of attained service in time-shared systems." *Journal of Computer & System Sciences* 1, 3 (1967), 287-298.

[Lazowska et. al. 1984] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

[Little 1961] J. D. C. Little, "A Proof of The Queuing Formula L = λ W", *Operations Research*, 9 (October 1961), 383-387.

[Lucas 1971] H. C. Lucas. "A User Oriented Approach to System Design". *Proc. 1971 ACM Annual Conference*, 325-338.

[Lynch 1972] W.C. Lynch. "Operating System Performance". *CACM* 15,7 (July 1972), 579-586.

[Peterson and Silberschatz 1983] J. Peterson and A. Silberschatz, *Operating System Concepts*. Addison-Wessley, Reading, Mass., 1983.

[Salza and Lavenberg 1981] S. Salza and S. Lavenberg. "Approximating response time distributions in closed queuing network models of computer performance." *Performance 1981*. Proceedings of 8th. Intl. Symposium on Computer Performance Modeling, Measurement and Evaluation.

[Sauer and Chandy 1981] Charles H. Sauer and K. Mani Chandy. *Computer Systems Performance Modeling*. PrenticeHall, Englewood Cliffs, N.J., 1981.

Appendix A -- Qualifications

Personal Background

My formal course work at R.I.T. has included two double course sequences which have provided a firm foundation for the completion of this thesis project. The first sequence was a two quarter examination of operating systems. The high point of this sequence was the implementation of memory management in a UNIX–like operating system running on an Integrated Solutions MC–68010 microprocessor. That project had a very primitive job scheduler, it placed jobs in the ready–to–run queue on a first–come–first–served basis and allowed each scheduled job to run to completion. This was a first–come–first–served scheduler without any interrupts which would have made it a round–robin scheduler. This project introduced me to useful data structures for representing processes and queues of jobs in an operating system.

The second relevant sequence of courses was in modeling and simulation. The second quarter of this sequence was spent working on a project at Computer Consoles Incorporated. The production process of integrated circuit boards was simulated in a large GPSS model. The production forecasts derived from this model matched the company's actual monthly production statistics within a 5% tolerance. From this project I learned techniques in designing and managing a large simulation. I also gained an appreciation of the relative merits of the GPSS simulation package. Some interactions of that large GPSS model of board production ran for more than four hours, requiring in excess of 60 minutes of cpu time.

Currently I am employed by Eastman Kodak Company as a Product Software Engineer. The project group I am presently assigned to is developing the user interface for an office publishing system. The majority of this project is written in C, which has greatly enhanced my command of this language.

This topic of CPU scheduling and the approach of evaluating algorithms through a software simulation of the job scheduler in an operating system present a natural blending of these two areas of interest and prior study. It is a topic of considerable interest to me personally, and I suspect of interest to those who must routinely share their computing resources with other people.

Courses Taken

Introductory and Bridge Courses

CTDP 200	Intro to Microcomputers
CTDP 208	Intro to Programming
CTDS 230	Discrete Structure
ICSS 701	Programming I
ICSS 702	Programming II

Core Computer Science Courses

ICSS 706	Foundations of Computing Theory
ICSS 708	Computer Organization and Programming
ICSS 709	Programming Language Theory
ICSS 720	Computer Architecture

Elective and Advanced Topics in Computer Science

ICSS 770	Fundamentals of Computer Graphics
ICSS 836	Data Base Systems
ICSS 890	Artificial Intelligence Seminar
ICSS 730	Modeling and Simulation I
ICSS 731	Modeling and Simulation II
ICSS 809	Operating Systems I
ICSS 810	Operating Systems II

It is the last four courses in the above list which are the most relevant to this thesis project. The proposed model of job scheduling represents a very natural blending of the issues and methods introduced in these four courses.

Appendix B -- Model Input

10	10	10	15	20	20	15	20	10	10
20	20	20	30	50	80	20	20	20	20
550	3020	130	440	300	670	480	370	620	1300
30	30	40	70	50	70	40	30	30	30
10	10	10	18	150	60	10	10	10	10
2470	1460	3490	360	9760	3320	5000	4630	9440	1180
20	230	60	210	220	20	20	10	60	50
110	110	130	190	210	380	130	110	210	190
10	40	30	70	110	1730	1100	300	330	100
3380	110	2320	550	1250	1570	1230	240	200	150
710	590	960	600	980	130	110	610	550	580
70	50	370	1490	690	200	2120	1550	3710	880

```
7 54 4 6 6 13 1 0 3 3 0 3
25 18 14 8 22 1 2 0 3 4 2 1
7 8 13 3 13 47 1 1 4 0 0 3
14 25 18 18 4 0 4 3 7 7 0 0
27 13 12 8 20 10 3 0 0 2 0 5
21 9 21 2 2 2 8 0 33 2 0 0
0 0 47 6 12 0 23 0 12 0 0 0
47 11 16 0 21 0 0 5 0 0 0 0
3 21 39 11 17 0 0 3 0 3 0 3
23 8 23 23 0 0 8 0 0 15 0 0
0 0 34 0 33 0 0 0 0 0 33 0
36 0 9 0 37 0 9 0 0 0 0 9
```

Appendix C -- Sample Output

```
Enter number of active users:  30
Enter minutes of simulation:  240

Enter number of desired Scheduling Algorithm
        (1) First Come First Serve -- Round Robin
        (2) Shortest Job First
        (3) Priority Queuing--(Interactive/Batch)
        Choice :  1

Scheduling Algorithm is First Come First Serve
Enter seed for random numbers:  1979

**********   System Stats   **********

Number of users: 30
Total minutes simulated:  240
Total seconds simulated:  14400
CPU seconds utilized:  13593
CPU utilization  94.40%
```

Job Type	Number	Percent
cd	320	17.34
ls	362	19.62
vi	333	18.05
more	114	6.18
light	207	11.22
make	237	12.85
cp	54	2.93
mail	10	0.54
run	117	6.34
grep	44	2.38
lpr	19	1.03
heavy	28	1.52
Total jobs completed:		1845

******** RTR Queue Ending State ******

type	user	arrive	depart	need	RTR-SZ
make	8	1529039	1620000	9760	16
grep	1	1596475	1620000	3380	15
make	24	1614786	1620000	360	14
make	19	1550086	1620000	9760	13
make	15	1591282	1620000	9760	12
make	16	1591575	1620000	9760	11
make	2	1615180	1620000	3320	10
lpr	25	1618537	1620000	110	9
make	7	1593315	1620000	5000	8
make	21	1527339	162000000		7
vi	28	1071871	1620000	3020	6
make	9	1559316	1620000	9440	5
ls	27	1619549	1620000	30	4
make	13	1591995	1620000	4630	3
make	11	1583647	1620000	5000	2
make	22	1579459	1620000	3490	1
make	3	1592486	1620000	3320	0

******** SYSTEM POPULATION STATS ******

Average population: 5.01

```
size  milisec.          minutes
   0  182878**********************************
   1  102178*******************
   2  107160********************
   3  105184*******************
   4  106282*******************
   5  123568***********************
   6  135265*************************
   7  142753**************************
   8  141210**************************
   9  127068************************
  10   97507******************
  11   65696************
  12   37231*******
  13   18742***
  14    9876*
  15    8013*
  16    9017*
  17   10519*
  18    7015*
  19    3465
  20    1142
  21     350
  22      41
  23      18
```

********** Job Stats **********

Job Type	Number	Avg. Resp. Time	Std. Dev.	Worst Case	Wted TAT
cd	320	90.74	53.85	326	8.78
ls	362	193.81	175.55	1394	8.86
vi	333	20.92	19.87	196	7.69
more	114	19.51	18.22	123	8.28
light	207	175.23	292.67	1852	9.44
make	237	25948.03	26142.44	144187	8.35
cp	54	634.20	786.16	2730	9.37
mail	10	1209.30	973.16	3088	7.38
run	117	2654.70	4530.66	29554	8.52
grep	44	7662.95	10373.08	56238	8.80
lpr	19	3486.79	3121.67	11266	9.70
heavy	28	12551.57	14097.41	64641	9.77

```
Enter number of active users:  30
Enter minutes of simulation:  240

Enter number of desired Scheduling Algorithm
        (1) First Come First Serve -- Round Robin
        (2) Shortest Job First
        (3) Priority Queuing--(Interactive/Batch)
        Choice :  2

Scheduling Algorithm is Shortest Job First
Enter seed for random numbers:  1979

**********    System Stats    **********

Number of users: 30
Total minutes simulated:  240
Total seconds simulated:  14400
CPU seconds utilized:  13930
CPU utilization  96.74%


        Job Type  Number Percent
              cd     323   17.50
              ls     373   20.21
              vi     312   16.90
            more     116    6.28
           light     226   12.24
            make     234   12.68
              cp      48    2.60
            mail      13    0.70
             run     117    6.34
            grep      41    2.22
             lpr      16    0.87
           heavy      27    1.46
Total jobs completed:        1846

******** RTR Queue Ending State ******

type    user    arrive  depart  need    RTR-SZ
vi      17      1392033 1620000 3020    3
make    24      1615052 1620000 2470    2
make    14      1456521 1620000 9760    1
make    11      1495018 1620000 9760    0
```

******** SYSTEM POPULATION STATS ******

Average population: 3.01

```
size  milisec.            minutes
   0  179019*******************************
   1  132031***********************
   2  195800**********************************
   3  219560**************************************
   4  234622****************************************
   5  185695********************************
   6  109310******************
   7   63626**********
   8   74618************
   9  103371*****************
  10   54778*********
  11   16509**
  12    2761
  13     307
  14      24
```

********** Job Stats **********

Job Type	Number	Avg. Resp. Time	Std. Dev.	Worst Case	Wted TAT
cd	323	16.66	6.48	47	1.36
ls	373	39.32	26.81	152	1.46
vi	312	3.34	2.53	37	1.19
more	116	4.15	3.31	21	1.31
light	226	42.61	61.17	325	1.45
make	234	27161.31	55976.03	392957	4.65
cp	48	111.79	119.97	352	1.55
mail	13	231.31	125.06	461	1.21
run	117	423.61	770.54	4557	1.52
grep	41	1596.56	1807.52	6271	1.95
lpr	16	714.69	421.86	1373	1.48
heavy	27	2333.74	4179.18	17760	1.88

```
Enter number of active users:  30
Enter minutes of simulation:  240

Enter number of desired Scheduling Algorithm
        (1) First Come First Serve -- Round Robin
        (2) Shortest Job First
        (3) Priority Queuing--(Interactive/Batch)
        Choice :  3

Scheduling Algorithm is Priority Queueing
Enter seed for random numbers:  1979

**********   System Stats   **********

Number of users: 30
Total minutes simulated:  240
Total seconds simulated:  14400
CPU seconds utilized:  13672
CPU utilization  94.95%


        Job Type  Number Percent
              cd     336   18.35
              ls     375   20.48
              vi     309   16.88
            more     121    6.61
           light     202   11.03
            make     234   12.78
              cp      46    2.51
            mail      10    0.55
             run     105    5.73
            grep      45    2.46
             lpr      22    1.20
           heavy      26    1.42
Total jobs completed:        1831

******** RTR Queue Ending State ******

type    user    arrive  depart  need    RTR-SZ
make    12      1607395 1620000 3320    4
make    15      1592528 1620000 9760    3
make    30      1601355 1620000 4630    2
make    23      1573633 1620000 9760    1
make    10      1614255 1620000 9760    0
```

******** SYSTEM POPULATION STATS ******

Average population: 4.11

```
size  milisec.          minutes
   0  204261************************************
   1  131498***********************
   2  161093****************************
   3  161189****************************
   4  150716**************************
   5  145198*************************
   6  130332***********************
   7   90680****************
   8   89162***************
   9   71892************
  10   55885*********
  11   54619*********
  12   54463*********
  13   55712*********
  14   13170**
  15    1454
  16     169
  17       5
```

********** Job Stats **********

Job Type	Number	Avg. Resp. Time	Std. Dev.	Worst Case	Wted TAT
cd	336	16.84	6.12	38	1.34
ls	375	37.16	25.45	117	1.39
vi	309	3.64	2.51	38	1.29
more	121	3.81	2.99	36	1.31
light	202	36.50	53.43	219	1.45
make	234	29908.24	30182.25	129762	8.40
cp	46	569.07	690.63	2521	10.85
mail	10	925.30	319.85	1326	7.06
run	105	2185.90	4111.07	23875	9.27
grep	45	6868.98	9880.35	46106	9.66
lpr	22	3803.86	1941.54	7051	6.99
heavy	26	7436.12	11308.31	50630	9.47

Appendix D -- Model Source Code

```
#include <stdio.h>
#include <ctype.h>

#define         GLOBAL
#define         LOCAL   static
#define         BOOL    char

#define         TRUE    1
#define         FALSE   0

#define         void    int
#define         VOID    int
```

```
/*
**      queues.h
**
**      type job
**              includes fields for linking jobs into queues,
**              scheduling jobs by various criteria,
**              and gathering performance stats
*/

struct  job_box{
        long    arrival_time;
        long    departure_time;
        long    origin_time;
        short   interrupts;
        short   total_need;
        short   unmet_need;
        short   burst_size;
        short   service_received;
        char    *job_type;
        char    user_id;
        struct  job_box *next_job;
        struct  job_box *prev_job;
        };

struct  queue{
        struct  job_box *head;
        struct  job_box *tail;
        short   population;
        };

#define BY_TIME         0
#define BY_UNMET_NEED   1
```

```
#define         TIME_UNITS      100     /* 100ths of a second */
#define         QUANTUM         1       /* 1/100ths of a second */

#define         T_SUB_B         180000  /* 30 minutes in 100ths of secs. */
```

```
#define         MAX_USERS       64

#define         THINK_TIME      17

#define         TYPE_TIME       8

#define         MAX_JOBS        256

#define         MAX_TASK_TYPES  12

#define         BG_TYPE         5

#define         CD_TASK         0
#define         LS_TASK         1
#define         VI_TASK         2
#define         MORE_TASK       3
#define         LIGHT_TASK      4
#define         MAKE_TASK       5
#define         COPY_TASK       6
#define         MAIL_TASK       7
#define         RUN_TASK        8
#define         GREP_TASK       9
#define         LPR_TASK        10
#define         HEAVY_TASK      11


#define         CD_STR          "cd"
#define         LS_STR          "ls"
#define         VI_STR          "vi"
#define         MORE_STR        "more"
#define         LIGHT_STR       "light"
#define         MAKE_STR        "make"
#define         COPY_STR        "cp"
#define         MAIL_STR        "mail"
#define         RUN_STR         "run"
#define         GREP_STR        "grep"
#define         LPR_STR         "lpr"
#define         HEAVY_STR       "heavy"
#define         OTHER           "other"
```

```
#include        "gendef.h"
#include        "queues.h"

GLOBAL  struct  queue   job_que;

extern  long            loc_clock;
extern  struct  job_box *pop();
extern  BOOL            que_is_empty();
extern  VOID            (*schedule)();

GLOBAL  VOID
check_jobs_que()
{

        struct  job_box *ptr;

        while ( ( !que_is_empty( &job_que ) ) &&
                ( job_que.head->arrival_time <= loc_clock )){

                ptr = pop( &job_que );
                (*schedule)( ptr );
        }
}

GLOBAL  long
next_event()
{
        if ( que_is_empty( &job_que ) ){
                printf( "ERROR   job_que is empty. TIME: %ld\n", loc_clock);
                exit(1);
        }
        else {
                return ( job_que.head->arrival_time );
        }
}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "units.h"
#include        "users.h"
#include        <math.h>

#define         USER_DATA_FILE  "user.data"

GLOBAL  struct  job_box *(*get_next_job)();
GLOBAL  int             num_users;
GLOBAL  BOOL            (*rtr_is_empty)();
GLOBAL  short           (*rtr_size)();
GLOBAL  VOID            (*schedule)();

extern  struct  queue   job_que;
extern  struct  queue   rtr_que;
extern  long            loc_clock;
extern  long            quiting_time;
extern  int             user_data[MAX_TASK_TYPES][MAX_TASK_TYPES];

extern  VOID            init_demand();
extern  VOID            init_stats();
extern  VOID            make_job_lists();
extern  VOID            srand();

LOCAL   VOID
init_users()

{

        int     cum;
        FILE    *fd;
        int     i;
        int     j;
        struct job_box  *ptr;
        int     num;

        /*
        **      initialize user_data stats from file
        */

        if ( ( fd = fopen( USER_DATA_FILE, "r")) == NULL ){
                printf("FATAL ERROR opening user.data file \n");
                exit(1);
        }

        for (i = 0; i < MAX_TASK_TYPES; i++){

                cum = 0;
                for ( j = 0; j < MAX_TASK_TYPES; j++){

                        if ( fscanf( fd, "%d", &num) == 0 ){
                                printf("ERROR reading user.data  i=%d j=%d\n",
                                 i, j);
                                exit (1);
                        }
                        cum += num;
```

```
                        user_data[i][j] = cum;
                }

                if ( cum != 100 ){
                        printf("ERROR cum[%d]=%d \n", i, cum );
                        exit(1);
                }
        }

        fclose( fd );


        /*
        **      make initial task for each user
        */

        make_job_lists( num_users );

}


GLOBAL  VOID
initialize()
{

        long                    minutes;
        int                     rand_seed;
        int                     algorithm_choice;
        extern  VOID            fcfs_schedule();
        extern  VOID            sjf_schedule();
        extern  VOID            pq_schedule();
        extern  struct  job_box *fcfs_get_next();
        extern  struct  job_box *sjf_get_next();
        extern  struct  job_box *pq_get_next();
        extern  BOOL            fcfs_que_is_empty();
        extern  BOOL            sjf_que_is_empty();
        extern  BOOL            pq_que_is_empty();
        extern  short           fcfs_rtr_size();
        extern  short           sjf_rtr_size();
        extern  short           pq_rtr_size();

        printf("Enter number of active users:  ");
        scanf("%d", &num_users);
        printf("%d\n", num_users);

        printf("Enter minutes of simulation:  ");
        scanf("%ld", &minutes);
        printf("%ld\n", minutes);

        printf("\nEnter number of desired Scheduling Algorithm\n");
        printf("\t(1) First Come First Serve -- Round Robin\n");
        printf("\t(2) Shortest Job First\n");
        printf("\t(3) Priority Queuing--(Interactive/Batch)\n");
        printf("\tChoice :  ");

        scanf("%d", &algorithm_choice);
        printf("%d\n\n", algorithm_choice);
```

```
        switch ( algorithm_choice ){
                case 1:         schedule = fcfs_schedule;
                                get_next_job = fcfs_get_next;
                                rtr_is_empty = fcfs_que_is_empty;
                                rtr_size = fcfs_rtr_size;
                                printf("Scheduling Algorithm is First Come First
                                break;
                case 2:         schedule = sjf_schedule;
                                get_next_job = sjf_get_next;
                                rtr_is_empty = sjf_que_is_empty;
                                rtr_size = sjf_rtr_size;
                                printf("Scheduling Algorithm is Shortest Job Fir
                                break;
                case 3:         schedule = pq_schedule;
                                get_next_job = pq_get_next;
                                rtr_is_empty = pq_que_is_empty;
                                rtr_size = pq_rtr_size;
                                printf("Scheduling Algorithm is Priority Queuein
                                break;
                default:        printf("Invalid Scheduler Choice!!!\n");
                                exit(2);
                                break;
        }
        quiting_time = ( minutes * 60 * TIME_UNITS ) + T_SUB_B;

        printf("Enter seed for random numbers:  ");
        scanf("%d", &rand_seed);
        printf("%d\n", rand_seed);

        srand( (int)rand_seed );

        loc_clock = 0;

        init_demand();

        init_users();

        init_stats();
}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "units.h"

extern          VOID    check_jobs_que();
extern          VOID    initialize();
extern          VOID    init_stats();
extern          long    next_event();
extern          BOOL    (*rtr_is_empty)();
extern          VOID    run();

GLOBAL          long    loc_clock;
GLOBAL          long    quiting_time;
GLOBAL  struct  queue   rtr_que;

main()
{
        initialize();

        while ( loc_clock < T_SUB_B ){
                check_jobs_que();

                if ( ! (*rtr_is_empty)() )
                        run();
                else
                        loc_clock = next_event();
        }

        init_stats();   /* reset stats after Tb */

        while ( loc_clock < quiting_time ){
                check_jobs_que();

                if ( ! (*rtr_is_empty)() )
                        run();
                else
                        loc_clock = next_event();
        }

        print_stats();

}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "units.h"

extern  long            cpu_useage;
extern  struct  job_box *(*get_next_job)();
extern  long            loc_clock;
extern  VOID            make_next_job();
extern  long            queue_stats[];
extern  VOID            response_stats();
extern  short           (*rtr_size)();
extern  VOID            (*schedule)();

GLOBAL  VOID
run()

{

        struct  job_box *cur_job;
        extern  struct  queue   *rtr_que;

        cur_job = (*get_next_job)();

        if ( cur_job->unmet_need <= QUANTUM ){  /*  finishable job */

                loc_clock += (long)cur_job->unmet_need;
                cpu_useage += (long)cur_job->unmet_need;
                queue_stats[ (*rtr_size)() ] += (long)cur_job->unmet_need;

                cur_job->service_received += cur_job->unmet_need;
                cur_job->unmet_need = 0;
                cur_job->departure_time = loc_clock;

                response_stats( cur_job );
                make_next_job( cur_job );
        }

        else {
                loc_clock += (long)QUANTUM;
                cur_job->service_received += QUANTUM;
                cur_job->unmet_need -= QUANTUM;
                queue_stats[ (*rtr_size)() ] += (long)QUANTUM;

                cpu_useage += (long)QUANTUM;

                (*schedule)( cur_job );
        }
}
```

```
#include        "gendef.h"
#include        "queues.h"


extern  BOOL    que_is_empty();
extern  VOID    push();
extern  VOID    enqueue();

/*
**      que is a sorted linked-list of jobs
**      the job_que is generally sorted on the basis of arrival time
**      the rtr_que may be sorted on the basis of arrival time
**      (first-come-first-serve) or least un-met need ( shortest-job-
**      first )
**
**      insert can sort on the basis of two criteria--arrival time or unmet need
*/

GLOBAL VOID
insert( job, que, criteria )

struct  job_box *job;
struct  queue   *que;
int     criteria;
{

        struct  job_box *leader,
                        *follower;

        if ( que_is_empty( que ) ){
                push ( job, que );
                return;
        }

        if ( criteria == BY_TIME ){

/*
**      If it belongs at the end or the beginning, put it there.
**      Otherwise walk through the waiting jobs to insert it.
*/
                if ( job->arrival_time < que->head->arrival_time ){
                        push( job, que );
                        return;
                }
                if ( job->arrival_time >= que->tail->arrival_time ){
                        enqueue( job, que );
                        return;
                }

                leader = que->head;
                while ( (job->arrival_time >= leader->arrival_time )
                        &&      (leader->next_job != NULL) ){
                        leader = leader->next_job;
                }
                follower = leader->prev_job;
                follower->next_job = job;
                job->prev_job = follower;
```

```
                job->next_job = leader;
                leader->prev_job = job;
                ++que->population;
        }

/*
**      Insert job into que on the basis of least remaining unmet need.
**
**
**      If it belongs at the end, put it there.
**      Otherwise walk through the waiting jobs to insert it.
*/
        else if ( criteria == BY_UNMET_NEED ){
                if ( job->unmet_need < que->head->unmet_need ){
                        push( job, que );
                        return;
                }
                if ( job->unmet_need >= que->tail->unmet_need ){
                        enqueue( job, que );
                        return;
                }

                leader = que->head;
                while ( (job->unmet_need >= leader->unmet_need )
                        &&      (leader->next_job != NULL) ){
                        leader = leader->next_job;
                }
                follower = leader->prev_job;
                follower->next_job = job;
                job->prev_job = follower;
                job->next_job = leader;
                leader->prev_job = job;
                ++que->population;
        }
}
```

```
#include        "gendef.h"
#include        "queues.h"


GLOBAL  BOOL
que_is_empty( que )

struct queue    *que;
{

        if (que->head)
                return (FALSE);
        else
                return (TRUE);
}

GLOBAL  VOID
push( job, que )

struct job_box  *job;
struct queue    *que;
{

/*
**      Push this job onto the front of the queue and adjust the pointers.
*/

        if ( que_is_empty( que ) ){
                que->head = que->tail = job;
                job->next_job = job->prev_job = NULL;
        }
        else {
                job->next_job = que->head;
                job->prev_job = NULL;
                que->head->prev_job = job;
                que->head = job;
        }
        ++que->population;
}

GLOBAL VOID
enqueue( job, que )

struct job_box  *job;
struct queue    *que;
{

/*
**      Enqueue this job at the end of the queue and adjust the pointers.
*/

        if (que_is_empty( que ) ){
                que->head = que->tail = job;
                job->next_job = job->prev_job = NULL;
        }
        else {
                que->tail->next_job = job;
```

```
                job->next_job = NULL;
                job->prev_job = que->tail;
                que->tail = job;
        }
        ++que->population;
}


GLOBAL struct job_box *
pop( que )

struct queue    *que;
{

        struct job_box  *ptr;

/*
**      Take first element out of que,
**      adjust pointers, and
**      return a pointer to the poped element.
*/

        if ( que_is_empty ( que ) )
                return ( NULL );
        else {
                ptr = que->head;
                if ( que->head->next_job ) {
                        que->head->next_job->prev_job = NULL;
                        que->head = que->head->next_job;
                }
                else{
                        que->tail = NULL;
                        que->head = NULL;
                }
                ptr->next_job = NULL;
                --que->population;
                return ( ptr );
        }
}

GLOBAL VOID
print_que( que )

struct queue *que;
{

        struct job_box  *ptr;

        ptr = que->head;

        printf( "head ->" );

        while ( ptr ){

                printf( " %d ->", ptr->user_id );
                ptr = ptr->next_job;
        }
```

```
        printf( " tail  size:%d\n", que->population );
}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "users.h"

/*
**      Priority Scheduler
*/

GLOBAL  struct  queue   batch_que;

extern  VOID            enqueue();
extern  struct  job_box *pop();
extern  BOOL            que_is_empty();
extern  struct  queue   rtr_que;

GLOBAL  BOOL
pq_que_is_empty()
{
        return(  que_is_empty( &rtr_que ) && que_is_empty( &batch_que) );
}


GLOBAL  short
pq_rtr_size()

{
        return( batch_que.population + rtr_que.population );
}


GLOBAL  VOID
pq_schedule( job )

struct  job_box *job;
{
        if ( *job->job_type < BG_TYPE )
                enqueue ( job, &rtr_que );
        else
                enqueue ( job, &batch_que );
}

/*
**      This version of *get_next_job is for priority queues scheduling
**      and gets the next job from the head of the highest priority
**      queue which is not empty.
*/

GLOBAL  struct  job_box *
pq_get_next()

{
        if ( !que_is_empty( &rtr_que ) )
                return  ( pop( &rtr_que ) );
        else
                return  ( pop( &batch_que ));
}
```

```
#include        "gendef.h"
#include        "queues.h"

/*
**      Round Robin Scheduler
*/

extern  VOID            enqueue();
extern  struct  job_box *pop();
extern  BOOL            que_is_empty();
extern  struct  queue   rtr_que;

GLOBAL  BOOL
fcfs_que_is_empty()
{
        return(  que_is_empty( &rtr_que ) );
}


GLOBAL  short
fcfs_rtr_size()

{
        return( rtr_que.population );
}


GLOBAL  VOID
fcfs_schedule( job )

struct  job_box *job;
{
        enqueue ( job, &rtr_que );
}

/*
**      This version of *get_next_job is for round robin scheduling
**      and gets the job from the head of a single
**      first-come-first-serve rtr-que.
*/

GLOBAL  struct  job_box *
fcfs_get_next()

{
        return  ( pop( &rtr_que ) );
}
```

```
#include        "gendef.h"
#include        "queues.h"

/*
**      Shortest-Job_First Scheduler
*/

extern  VOID            insert();
extern  struct  job_box *pop();
extern  BOOL            queue_is_empty();
extern  struct  queue   rtr_que;

GLOBAL  BOOL
sjf_que_is_empty()
{
        return(  que_is_empty( &rtr_que ) );
}


GLOBAL  short
sjf_rtr_size()

{
        return( rtr_que.population );
}


GLOBAL  VOID
sjf_schedule( job )

struct  job_box *job;
{
        insert ( job, &rtr_que, BY_UNMET_NEED );
}

/*
**      This version of *get_next_job is for shortest-job-first
**      scheduling and gets the job from the head of a single rtr-que.
*/

GLOBAL  struct  job_box *
sjf_get_next()

{
        return  ( pop( &rtr_que ) );
}
```

```
#include        "gendef.h"

extern  VOID    init_job_stats();
extern  VOID    init_queue_stats();
extern  VOID    init_system_stats();

GLOBAL  VOID
init_stats()

{

        init_job_stats();
        init_queue_stats();
        init_system_stats();
}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "units.h"
#include        "users.h"
#include        <math.h>

extern  long    loc_clock;

LOCAL   double  std_dev();
LOCAL   double  r_t_num[ MAX_TASK_TYPES ];
LOCAL   double  r_t_sum[ MAX_TASK_TYPES ];
LOCAL   double  r_t_squared[ MAX_TASK_TYPES ];
LOCAL   int     total_jobs[ MAX_TASK_TYPES ];
LOCAL   double  weighted_ta_time[ MAX_TASK_TYPES ];
LOCAL   long    worst_case[ MAX_TASK_TYPES ];

GLOBAL  char
*job_type( id )

int     id;
{

        switch ( id ) {

                case CD_TASK:           return( CD_STR );
                case LS_TASK:           return( LS_STR );
                case VI_TASK:           return( VI_STR );
                case MORE_TASK:         return( MORE_STR );
                case LIGHT_TASK:        return( LIGHT_STR );
                case MAKE_TASK:         return( MAKE_STR );
                case COPY_TASK:         return( COPY_STR );
                case MAIL_TASK:         return( MAIL_STR );
                case RUN_TASK:          return( RUN_STR );
                case GREP_TASK:         return( GREP_STR );
                case LPR_TASK:          return( LPR_STR );
                case HEAVY_TASK:        return( HEAVY_STR );

                default:                return( OTHER );

        }
}

GLOBAL  VOID
job_header()

{
        printf("\n%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\n",
          "type",
          "user",
          "arrive",
          "depart",
          "need",
          "Res Tm",
          "Wtd TAT",
          "%CPU",
          "RTR-SZ" );
}
```

```
GLOBAL  VOID
init_job_stats()

{
        int     i;

        for ( i = 0; i < MAX_TASK_TYPES; i++ ){
                r_t_num[i] = 0;
                r_t_sum[i] = 0;
                r_t_squared[i] = 0;
                total_jobs[i] = 0;
                worst_case[i] = 0;
        }
#ifdef  VERBOSE
        job_header();
#endif
}


GLOBAL  VOID
show_a_job( cur_job )

struct  job_box *cur_job;
{
        extern  short           (*rtr_size)();

/*
** job_type  user  departure   arrival  total_need   weighted_ta_time %CPU QUE
*/
        printf("%s\t%d\t%ld\t%ld\t%d\t%5.2f\t%ld\t%5.2f\t%d\n",
          job_type( *cur_job->job_type ),
          cur_job->user_id,
          cur_job->origin_time,
          cur_job->departure_time,
          cur_job->total_need,

          (double)( ( (double)cur_job->departure_time -
                (double)cur_job->origin_time ) / (double)TIME_UNITS ),

          ( (cur_job->departure_time -          /* weighted turn-arnd time*/
                cur_job->origin_time  ) /
                cur_job->service_received),


          (double)cur_job->total_need /
                (double)( ( (double)cur_job->departure_time -
                (double)cur_job->origin_time )),

          (*rtr_size)() );
}


GLOBAL  VOID
response_stats( cur_job )
```

```
struct  job_box *cur_job;
{
        long                    resp_time;
        double                  wt_ta_time;

        resp_time = cur_job->departure_time - cur_job->arrival_time;

        if ( resp_time > worst_case[ *cur_job->job_type ] ){
                worst_case[ *cur_job->job_type ] = resp_time;
        }

        wt_ta_time = (double)resp_time / (double)cur_job->burst_size;

        weighted_ta_time[ *cur_job->job_type ] += wt_ta_time;

        r_t_num[ *cur_job->job_type ] += 1;
        r_t_sum[ *cur_job->job_type ] += (double)resp_time;

        r_t_squared[ *cur_job->job_type ] += (double)( (double)resp_time *
        (double)resp_time );

}

GLOBAL  VOID
job_stats( cur_job )

struct  job_box *cur_job;
{
        total_jobs[ *cur_job->job_type ]++;

#ifdef  VERBOSE
        show_a_job( cur_job );
#endif
}



GLOBAL  VOID
pr_job_stats()

{

        int     i;
        double  avg;
        double  dev;
        double  wt_ta_time;

        printf("\n**********   Job Stats   **********\n\n");

        printf("%10s%8s%17s%11s%12s%10s\n",
          "Job Type", "Number", "Avg. Resp. Time", "Std. Dev.",
          "Worst Case", "Wted TAT" );
        for ( i = 0; i < MAX_TASK_TYPES; i++){
                if ( total_jobs[ i ] ){
                        avg = r_t_sum[ i ] / r_t_num[ i ];
                        dev =  std_dev( i );
                        wt_ta_time = ( weighted_ta_time[i] /
```

```
                                        r_t_num[i] );
                }
                else{
                        avg = 0;
                        dev = 0;
                        wt_ta_time = 0;
                }
                printf("%10s%8d%17.2f%11.2f%12ld%10.2f\n",
                  job_type( i ), total_jobs[i], avg, dev, worst_case[i],
                  wt_ta_time );
        }


}

LOCAL   double
std_dev( i )

int     i;
{
        double  ans;
        double  divisor;
        double  numerator;
        double  square;

        if ( ! total_jobs[ i ] )
                return ( 0 );

        numerator = ( r_t_num[i] * r_t_squared[i] )
                        - ( r_t_sum[i] * r_t_sum[i] );

        if ( (divisor = ( r_t_num[i] * ( r_t_num[i] - 1 ) )) == 0)
                return( 0 );

        square = numerator / divisor;

        ans = sqrt(square);

        return ( ans );
}

GLOBAL  VOID
dump_remaining_jobs()
{

        struct  job_box         *ptr;
        extern  BOOL            (*rtr_is_empty)();
        extern  struct job_box  *(*get_next_job)();

        loc_clock += T_SUB_B;

        printf("\n******** RTR Queue Ending State ******\n");
        job_header();
        while ( ! (*rtr_is_empty)() ){
                ptr = (*get_next_job)();
                ptr->departure_time = loc_clock;
                show_a_job( ptr );
```

```
        }
        printf("\n\n");

        loc_clock -= T_SUB_B;
}
```

```
#include        "gendef.h"

extern  VOID    pr_job_stats();
extern  VOID    pr_queue_stats();
extern  VOID    pr_system_stats();

GLOBAL  VOID
print_stats()
{
        pr_system_stats();
        dump_remaining_jobs();
        pr_queue_stats();
        pr_job_stats();
}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "units.h"
#include        "users.h"

GLOBAL  long    cpu_useage;
GLOBAL  long    queue_stats[ MAX_USERS ];
LOCAL   int     job_count[ MAX_TASK_TYPES ];
LOCAL   int     total_jobs;

extern  char    *job_type();
extern  long    loc_clock;
extern  int     num_users;

GLOBAL  VOID
init_queue_stats()

{

        int     i;

        for ( i = 0; i < MAX_USERS; i++ )
                queue_stats[ i ] = 0;
}

GLOBAL  VOID
init_system_stats()

{
        int     i;

        total_jobs = 0;
        cpu_useage = 0;

        for ( i = 0; i < MAX_TASK_TYPES ; i++ )
                job_count[ i ] = 0;
}

GLOBAL  VOID
system_stats( cur_job )

struct  job_box *cur_job;
{

        job_count[ *cur_job->job_type ] += 1;
        total_jobs += 1;

}

GLOBAL  VOID
pr_system_stats()

{

        int     i;

        /*
```

```
        ** adjust loc_clock for Tb
        */

        loc_clock -= T_SUB_B;

        printf("\n***********   System Stats   ***********\n\n");

        printf("Number of users: %d\n", num_users );
        printf("Total minutes simulated:  %ld\n", loc_clock / 60 / TIME_UNITS );
        printf("Total seconds simulated:  %ld\n", loc_clock / TIME_UNITS );
        printf("CPU seconds utilized:  %ld\n", cpu_useage / TIME_UNITS);
        printf("CPU utilization  %5.2f%%\n",
                (double)( ( (double)cpu_useage / (double)loc_clock ) * 100) );

        printf("\n\n");
        printf("        %8s%8s%8s\n", "Job Type", "Number", "Percent");
        for ( i = 0; i < MAX_TASK_TYPES; i++){
                printf("        %8s%8d%8.2f\n",
                        job_type( i ),
                        job_count[i],
                        (double)(((double)job_count[i]
                                / (double)total_jobs ) * 100 ));
        }

        printf("Total jobs completed:%11d\n", total_jobs );

}

LOCAL   double
avg_queue_size()

{
        long    a;
        double  avg;
        long    b;
        long    cum;
        int     i;
        long    middle;
        long    que_cum[ MAX_USERS ];

        i = 0;
        cum = 0;
        middle = loc_clock / 2;

        do{
                cum +=  queue_stats[i];
                que_cum[i] = cum;
                i++;
        } while( cum < middle );

        if ( i == 1 ){
                avg = (double)( (double)middle /
                        (double)queue_stats[0] );
        }
        else{
                a = que_cum[i] - que_cum[i-1];
                b = que_cum[i] - middle;
```

```
                avg = (double)( (double)(i) -
                        (double)( (double)(b)/(double)(a) ));
        }

        return ( avg );
}


GLOBAL VOID
pr_queue_stats()

{
        register        int     i;
        register        int     j;
        long            *m;
        long            *n;
        long    convert;

        /*
        ** Adjust queue_stats array for job that is running when
        **      samples were gathered.
        */

        n = m = queue_stats + ( MAX_USERS * sizeof(long) );
        --m;
        do {
                --n;
                --m;
                *n = *m;
        }
        while ( m > queue_stats );

        /*
        ** Add cpu idle time to queue_stats[0]
        */

        queue_stats[0] += (loc_clock - cpu_useage);

        convert = 60 * TIME_UNITS;

        /* form feed */
        fflush( stdout );
        printf("
```

```
");

        printf("\n\n******** SYSTEM POPULATION STATS ******\n\n");

        printf("Average population:  %6.2f\n\n", avg_queue_size() );

        printf("size  milisec.\t\t minutes\n");

        for ( i=0; i < MAX_USERS; i++ ){
                if ( queue_stats[ i ] ){
                        printf("%5d  %6ld", i, queue_stats[ i ]);
                        for ( j=0; j < ( queue_stats[ i ] / convert ); j++ ){
                                putchar('*');
                        }
                        printf("\n");
                }
        }
        printf("\n\n");
}
```

```
#include        "gendef.h"
#include        "users.h"

#define         DEMAND_DISTRIBUTION     10
#define         DEMAND_FILE     "job.demands"

extern  int     get_rand();

LOCAL   short   job_demands[ MAX_TASK_TYPES ][ DEMAND_DISTRIBUTION ];

GLOBAL  short
demand( this_job )

char    this_job;
{
        int     rnum;

        rnum =  get_rand( DEMAND_DISTRIBUTION );
        return ( job_demand[this_job][rnum] );
}

GLOBAL  VOID
init_demand()
{
        FILE    *fd;
        int     i;
        int     j;
        short   num;

        /*
        **      initialize demand_data stats from file
        **
        **      each line in demand_data file will contain a
        **      distribution of typical times for that job_type
        */

        if ( ( fd = fopen( DEMAND_FILE, "r")) == NULL ){
                printf("FATAL ERROR opening demand file \n");
                exit(1);
        }

        for (i = 0; i < MAX_TASK_TYPES; i++){

                for ( j = 0; j < DEMAND_DISTRIBUTION; j++){

                        if ( fscanf( fd, "%d", &num) == EOF ){
                                printf("ERROR reading job.demands i=%d j=%d\n",
                                 i, j);
                                exit (1);
                        }
                        job_demands[i][j] = num;
                }
        }
        fclose( fd );
}
```

```
#include        "gendef.h"
#include        <math.h>

extern  rand();

/*
**      get_rand returns a random number between 0 and (range - 1)
**               the number is derived from the c-library subroutine
**               rand() which returns an integer between 0 and (2^15 - 1)
*/

GLOBAL  int
get_rand( range )

int     range;
{

        return( rand() % range );

}
```

```
#include        "gendef.h"
#include        "queues.h"
#include        "users.h"
#include        "units.h"

extern  struct  queue   job_que;
extern  long            loc_clock;
extern  int             get_rand();
extern  int             demand();
extern  VOID            insert();
extern  VOID            job_stats();
extern  VOID            system_stats();

GLOBAL  int     user_data[MAX_TASK_TYPES][MAX_TASK_TYPES];
LOCAL   int     a_tenth;

LOCAL   char
get_type_of_next_job ( this_job )

char    this_job;
{
        char    i;
        int     rnum;

        rnum = get_rand( 100 );
        for ( i = 0; i < MAX_TASK_TYPES; i++ )
        {
                if ( rnum <= user_data[ this_job ][ i ] ){
                        return ( i );
                }
        }
}


GLOBAL  VOID
make_job_lists( num_users )

int     num_users;
{
                        char            i;
        register        char            *ip;
                        char            *jp;
                        struct  job_box *ptr;
                        char            this_job;

        a_tenth = TIME_UNITS / 10;

        for ( i = 1; i <= num_users; i++ ){

                if ( ( ptr = (struct job_box *)malloc( sizeof (struct job_box))
                {
                        printf("FATAL_ERROR:  job_box malloc failed.\n");
                        exit(1);
                }

                if ( (ip = malloc( MAX_JOBS )) == NULL ) {
                        printf("FATAL_ERROR:  JOB_ARRAY malloc failed\n");
```

```
                exit(1);
            }
            ptr->job_type = ip;
            ptr->user_id = i;
            ptr->interrupts = 0;
            this_job = MAIL_TASK;



            for ( jp = ip + MAX_JOBS; ip < jp; ip++ ){
                *ip = get_type_of_next_job( this_job );
                this_job = *ip;
            }
            make_next_job( ptr );
        }
}



LOCAL   VOID
set_interrupts( cur_job )

struct  job_box *cur_job;
{
        cur_job->interrupts =
          ( cur_job->total_need / a_tenth ) * ( get_rand(5) + 1);

        if ( cur_job->interrupts ){
                cur_job->unmet_need = cur_job->burst_size =
                        cur_job->total_need / cur_job->interrupts;
        }

#ifdef VERBOSE
        printf("\t%s\tTotal need: %d\tBurst size: %d\tInterrupts: %d\n",
          job_type( *cur_job->job_type ),
          cur_job->total_need, cur_job->burst_size,
          cur_job->interrupts );
#endif
}

GLOBAL  VOID
make_next_job( cur_job )

struct  job_box *cur_job;
{

        /*
        ** reschedule an interactive job blocked for terminal input
        */

        if ( cur_job->interrupts ){
                cur_job->unmet_need = cur_job->burst_size;
                --cur_job->interrupts;
                cur_job->arrival_time =
                  loc_clock + ( get_rand( TYPE_TIME * TIME_UNITS ) );
                insert( cur_job, &job_que, BY_TIME );
                return;
```

```
        }

        /*
        ** gather stats from finished job
        */

        if ( loc_clock ) {
                job_stats( cur_job );
                system_stats( cur_job );
        }

        /*
        ** create new job
        */

        cur_job->job_type++;

        cur_job->arrival_time = cur_job->origin_time =
          loc_clock + ( get_rand(THINK_TIME * 2) * TIME_UNITS );
        cur_job->departure_time = 0;
        cur_job->unmet_need = cur_job->burst_size =
          cur_job->total_need = demand( *cur_job->job_type );
                /* demand returned in 100ths of seconds */
        cur_job->service_received = 0;
        cur_job->next_job = cur_job->prev_job = NULL;

        /*
        ** Interrupt initialization for interactive jobs
        */

        if ( ( *cur_job->job_type == VI_TASK ) ||
                ( *cur_job->job_type == MORE_TASK ) )
                set_interrupts( cur_job );
        else
                cur_job->interrupts = 0;

        insert( cur_job, &job_que, BY_TIME );

}
```