MODELLING AND ANALYSIS OF DISTRIBUTED SOFTWARE SYSTEMS

B. Kumar
Timothy A. Gonsalves

Computer Systems Laboratory
Stanford University
Stanford, California 94305

ABSTRACT

The problem of modelling and analysing the performance of software structures for distributed computer systems is addressed. A modelling technique is proposed which has many striking analogies with current techniques for evaluating hardware systems and yet focuses attention on the system software. The analysis of such models will draw heavily on the substantial work already done in the analysis of hardware models.

To illustrate the modelling technique proposed, it is applied to an investigation of the trade-offs associated with the configuration of critical sections in a distributed software system. Simple queueing techniques are used to model a number of alternative configurations. The study throws some light on the regions of optimum decomposition, and the impact on performance of some of the important design variables.

1. INTRODUCTION

There seems to be no abatement in the rate at which the price/performance ratio of computer system hardware continues to drop. Consequently, with powerful processing hardware being available at such low cost, distributed systems are being increasingly seen as the wave of the future. However, it is also being recognized that the most significant problem with the successful synthesis of such systems is the complexity of the system software needed to support and manage distributed computations. This paper therefore addresses the problem of analysing and characterizing the performance aspects of software structures for distributed systems. A technique for the modelling of distributed software systems is introduced and techniques for the analysis of such models are described. We also present a simple example of the application of such techniques.

Section 2 of the paper lays out the problem and raises some of the many questions that arise in the design of distributed software systems. Section 3 surveys some of the previous work done in the area. Section 4 describes the modelling technique, the analysis techniques and draws the analogies between this technique and the standard techniques used by computer system analysts. Section 5 presents a simple example illustrating the application of the technique to the characterization of the performance of one aspect of a distributed software system. Section 6 summarizes and offers suggestions for further work.

Support for this work was provided by the Joint Services Electronics Program grant DAAG-29-79-C-0047.

2. STATEMENT OF THE PROBLEM

In this section we discuss the importance of analysing software system performance as distinct from hardware system performance and raise some of the questions that may arise during the design of software for distributed computer systems.

2.1 Terminology

In the rest of this paper, we will use the following terminology. We will distinguish the hardware configuration of a computer system, viz., the complex of hardware resources - CPU's, channels, I/O devices, etc., from the software structure i.e., the operating system and application processes that execute on it. We will use the term module to loosely refer to a software unit which accomplishes a single logical function, eg., operating system routines such as schedulers, interrupt handlers, etc.

2.2 Need for Software System Performance Analysis

Computer system performance measures commonly used today fall under one of three categories: productivity (eg., throughput, CPU capacity), responsiveness (eg., response time, turnaround time) and utilization (eg., CPU and I/O channel utilization). While some of these measures relate in obvious ways to the characteristics of the system hardware, the relation of most of these measures to the system software is not immediately perceivable. Moreover, in practice the design of software is usually done at a higher level of abstraction than the system hardware level. Thus, for this design to be done in a coherent manner, it is essential that the performance aspects of the system software be brought to light and be somewhat separated from hardware and overall system performance considerations. This is all the more essential if the same software structure is to be transported to different hardware configurations.

2.2.1 Modular Structure of Software Systems

It is a commonly asserted opinion that techniques such as structured programming and modular implementations are sound engineering methods for the construction of large software systems. In order to choose between different modular implementations, the performance characteristics of such techniques need to be understood. For example, some multiprocessor configurations have experienced serious performance degradation due to processor contention at the single lock guarding a single large critical section that manipulates all the

system's resources. Splitting this critical section into simpler ones and thus allowing concurrent manipulation of disjoint portions of the resource management data base may yield a performance increase, but may also increase the overhead due to the necessity of deadlock-free management of these critical sections. Similarly, performance and cost can be traded when deciding whether many processors should share a single copy of a module as opposed to each executing a private copy. Analysis of the performance characteristics of the difference approaches will enable a sound engineering approach to resolving such trade-offs.

2.2.2 Hardware Organization Parameters and Implementation Trade-offs

While it is important to focus on the performance aspects of the software structures, it is also important to understand how the underlying hardware organization will impact the performance of the software system being designed. For example, a heterogeneous architecture with processors of different capacities, and maybe special capabilities, may impose different performance constraints on the modular structure of the software system than a homogenous architecture. Similarly, different methods of processor interconnection, memory-sharing, I/O port configuration, etc., may have significantly different impacts on the software system structure. These features need to be parameterized in the cost-performance models of software systems.

Even if we have succeeded in defining the "optimum" modular structure of the software system given performance and cost constraints, a number of issues related to the actual implementation have to be addressed. The performance aspects of various features of system programming languages, such as procedure calling sequences, inter-process communication, validity-checking, protection, etc., need to be understood. Further, the cost-performance trade-offs of providing hardware or firmware support for critical sections of the operating system need to be investigated.

3. PREVIOUS WORK

A number of multiprocessors have been designed and built, the more interesting ones being C.mmp [21], PRIME [3], PLURIBUS [9] and a local network-cum-multiprocessor Cm* [19]. However, very little work has been done in the investigation of the general principles underlying the design of software systems for these distributed architectures. The two exceptions are the HYDRA operating system for C.mmp [20] and the STAROS operating system for the Cm* [11]. These studies dealt with the specification and implementation of aspects such as process structuring, protection, message passing, task dispatching and kernel structure. In the former system, reference is made to some analytical studies conducted to investigate the impact of the number of critical sections in the scheduler on system performance. Apart from this, very little work has been done in analysing the performance aspects of design decisions regarding the system structure and the scheduling and coordinating policies embodied in the system.

Some interesting performance statistics related to the degradation due to lock contention, storage contention and interprocessor communication on the multiprocessor Michigan Terminal System (MTS) have been reported by Srodawa [18]. A statistical study of the operating system of the UNIVAC 1108 multiprocessor was done by Feeley [7]. We believe that the techniques proposed in this paper will make possible systematic studies of such systems.

4. A TECHNIQUE FOR MODELLING SOFTWARE SYSTEMS

In this section, we introduce the modelling technique and discuss methods of analysis of such models. We particularly stress the analogies between these techniques and the well-known approaches to hardware system evaluation. We believe that the application of tested techniques to a new problem should result in a greater likelihood of success.

4.1 Software System Models

The approach that we propose is related to, and yet quite different from, the conventional approach to computer system performance evaluation. All the performance models thus far have viewed computer systems as configurations of static hardware resources, such as CPU's, memories, I/O channels and devices, and user jobs or tasks as dynamic entities that flow through these configurations. Thus the system software is incorporated into these models at one of two levels:

1. the behavioural level: in the description of arrival patterns and resource demands of the job,
2. the algorithmic level: in the description of the scheduling and arbitration algorithms associated with resource management.

Since our purpose is to emphasize the software part of the system, we will take a diametrically opposite view. Our models will view a computer system as a configuration of static software modules, and the processors that execute the software as the dynamic entities that flow through this configuration. Whereas in a hardware system model a job occupies a resource for the time needed by the resource to process the job, in a software system model the processor occupies a module for the time needed for it to execute the code in that module. Further, the path of a job in a hardware system model is determined by the sequence of resources that it needs to complete its processing requirements. The path of a processor in a software system model is determined by the sequence of modules that it executes. Performance measures of interest to the software designer such as the utilization of various software modules can be obtained from the model using the analytic techniques described later.

4.2 An Example of a Software System Model

To illustrate the concepts described above, consider the following sequence of events in a time-shared, single-CPU system with virtual memory.

Time	Event
T_1	User U_1 incurs a page fault.
T_2	User U_1 is suspended and user U_2 is given a quantum of time on the CPU.
T_3	The CPU is interrupted by a command from the terminal of user U_3.
T_4	The CPU resumes the interrupted processing of user U_2's task.

Now consider a model of the software system that incorporates the module configuration of Figure 1. The above sequence of events can be modelled by the following sequence of actions of the processor through this model:

1. At T_1 the processor moves from USER to PFH, where is executes for $T_a - T_1$ sec., to process the page fault for U_1. It then moves to FMS where it executes for $T_b - T_a$ sec., to initiate the page transfer from auxiliary memory. It then moves to TD, where it executes for $T_2 - T_b$ sec., deciding which task from the READY queue is to receive the next CPU time slice. This task is U_2.
2. At T_2 the processor moves to USER where it executes U_2's task for $T_3 - T_2$ sec.
3. At T_3, it moves from USER to TIH, where it receives and parses the command from U_3's terminal for $T_c - T_3$ sec. This may lead to a decision to schedule U_3 to join the READY queue, so the processor moves to JS to accomplish this scheduling in $T_4 - T_c$ sec.
4. At T_4 the processor moves to USER to resume the execution of U_2's task.

4.3 Power of the Modelling Technique

The power of a modelling technique is reflected in two aspects:

1. The analysis power: is a measure of the techniques available to analyse the model and reach significant conclusions about the system.
2. The representation power: is reflected in the number and diversity of system features that can be conveniently represented in the model.

4.3.1 Analysis Power

The advantage of using the modelling technique described so far is that a number of standard techniques used in the analysis of hardware system models can be brought to bear on the analysis of such models. The specific analysis technique depends on the level of detail and accuracy required by the analysis.

For example, if the flow through the module network and the module processing times are deterministically modelled, the analysis may be conducted using trace-driven simulation. If any of the above model parameters is described by a probability distribution, stochastic simulation may be used as the analysis technique. If the above probability distributions are chosen from certain restricted classes, the well-developed methods of queueing system analysis may be used. The use of queueing system analysis imposes other restrictions. For example, past history of customers and servers and synchronization cannot be modelled exactly using standard queueing analysis.

4.3.2 Representation Power

A number of important features of software structures for distributed computer systems can be incorporated quite simply and very naturally into models of the kind proposed. We now give a few examples, emphasizing throughout the analogies with hardware system models.

1. Hardware system models usually model multiprogramming and time-sharing by having a fixed number of user jobs, equal to the degree of multiprogramming or the number of user terminals respectively, flowing through a resource configuration modelling the system hardware. The model of a software system for a uniprocessor is thus analogous to a hardware system model with the degree of multiprogramming equal to one. Similarly the model of the software system for a multiprocessor is analogous to a hardware multiprogrammed system model with the degree of multiprogramming in the latter corresponding to the number of processors in the former. The amazing degree of success in using closed queueing network techniques to model time-sharing systems ([16], [14]) and multiprogramming systems ([14], [10]) augurs well for their use in modelling software systems too.
2. In queueing models of hardware systems, the servers, i.e., the hardware resources, which are static entities, are characterized by a service discipline and a service rate. The latter is related to the processing power of the resources; commonly used measures include MIPS for CPU's, Mbps for channels, etc. User jobs, which are dynamic entities, are characterized by service demand distributions. The combination of job service demand distributions and resource service rates yields resource service time distributions. In software system models on the other hand, it seems more natural to associate the service demand with the static entity, i.e., the software module. This would be related to the length of the code along different paths through the module and would thus directly measure the implementation efficiency of that module. The service rate would now be associated with the dynamic entities, i.e., the processors.
3. Recent advances in queueing theory allow the modelling of different classes of customers with different service time distributions and different flow paths through the model [2]. This feature can be used to model two kinds of heterogeneity in distributed software systems:
   a) Heterogeneity due to the presence in the system of processors of different processing powers can be modelled by a separate class of customers for each type of processor, with different service rates for the different classes.
   b) Heterogeneity due to the dedication of certain processors to certain system functions can be modelled by assigning a unique class of customers for each dedicated processor or class of processors. Each customer class is now restricted to certain flow paths through the configuration of software modules.
4. Congestion occurs in a hardware system model when jobs contend for the use of a single hardware resource. That resource is then modelled as a single server with a queue of waiting jobs. The analogous situation in a software system model is when many processors try to execute a critical section module that

implements mutually exclusive access to some critical resource. That module can be modelled as a single server with a queue to hold waiting processors.

5. A commonly used resource scheduling discipline in multiprogramming systems is the round-robin discipline. This is modelled analytically by the processor-sharing discipline [12], in which the service rate of the server is shared by all the jobs using the server. The analogous situation in a software system model is when a number of processors execute a single copy of a module of re-entrant code. The service degradation due to many processors accessing the same physical regions of memory can be modelled as a drop in the effective service rate per processor due to module-sharing.

6. The extreme case of the processor-sharing discipline of resource scheduling is when an effectively infinite number of servers is available at a service station to process as many jobs as are at the station at any time. This is called the infinite server discipline [2]. In software systems, this may be used to model module replication - the situation where each processor has its own copy of a software module and can execute it without any interference from any other processors.

5. AN APPLICATION OF THE MODELLING TECHNIQUE

In this section we apply the techniques developed in section 4 to investigate the trade-offs involved in the configuration of critical sections in a distributed operating system. We analyse the models developed by simple queueing techniques. We show that the results from even such a simplistic modelling approach can provide some insight into the above trade-offs.

5.1 The Alternative Configurations

We consider the design of an operating system for a homogeneous distributed system consisting of n identical processors. The software operating system is assumed to consist of a number of modules among them being several critical sections. Each critical section, by definition, can be executed by only one processor at a time.

We compare two alternative configurations of the critical sections. In the first configuration, there is a single lock that allows only one processor at a time to obtain exclusive access to all the critical sections. This scheme is simple to implement, but may suffer from a performance degradation due to lack of concurrency in accessing the critical sections. In the second configuration, the critical sections are divided into logically related subsets with a lock provided for each subset. Each subset may be accessed by only one processor at a time, but different subsets may be executed concurrently by different processors. However, a processor may need to obtain exclusive control of more than one subset to accomplish certain functions. Thus, the allocation order of usage of locks by processors must be carefully controlled to prevent deadlocks. Thus additional overhead is necessitated in the form of a deadlock avoidance module that is responsible for the granting of all locks to the processors.

An example of the above trade-off may be found in the design of the OS/VS2 operating system for the System 370. Release 1 of OS/VS2 [17] which was not designed to support multiprocessing, has a single lock for all the critical sections of the system. Release 2 of OS/VS2, which supports multiprocessing, has about 13 separate locks with a locking hierarchy defined to solve the deadlock problem discussed above [1].

5.2 The Queueing Models

We assume that there are m independent critical section modules. Each of these may in fact consist of several related critical sections. For simplicity, we assume that the execution time of each of these modules is an exponentially distributed random variable with mean $1/\mu$. The entire software system is assumed to consist of these m modules plus an arbitrary number of other non-critical modules with arbitrary characteristics.

In our model, after spending some time in the non-critical modules, a processor moves to the set of critical section modules. It executes, on the average, c of these modules before returning to the non-critical modules to begin another such cycle. The distribution of execution times of the various modules and the routing probabilities of processors moving between the various modules are assumed to be the same for all the n processors.

5.2.1 Single Global Lock Scheme

In the single global lock scheme, all the m critical sections can be represented by a single Markovian server of rate μ/c. We assume that the time spent in locking is negligible compared to the execution time in the critical sections. This model is shown in Figure 2.

5.2.2 Multiple Lock Scheme

In the multiple lock scheme we assume that there is one lock for each of the m critical section modules. Further, we assume that the overhead introduced by the deadlock avoidance module is also exponentially distributed with a mean of $1/\lambda(m)$. The rationale for making this rate a function of m is that the complexity of the deadlock avoidance algorithm can be expected to increase with the number of modules to be allocated. The system can be modelled by the queueing network shown in Figure 3. After execution in the non-critical modules, a processor moves to server 0, the deadlock avoidance module, before it can gain access to any of the critical section modules. It then moves to one of the critical section modules at random. After execution in that module, with probability q it executes another critical section module, and with probability $p=1-q$, it returns to the non-critical modules for another such cycle. Thus the mean number of critical section modules executed per cycle is $c=1/p$.

Notice that the above model is only an approximation in that after a processor has been allocated its required set of critical section modules, it executes each of them sequentially, but does not lock other processors out of its allocated set. This is because multiple-resource holding behaviour cannot be modelled by queueing techniques. Thus, since the above model allows more concurrent usage of modules than actually possible, it yields an upper bound on performance. The service rates of the critical section modules could be appropriately reduced to approximately compensate for this increased

concurrency.

In the model of multiple critical sections used by McCredie [13] each processor executes all the critical sections in a static sequence. Our model is more powerful in that each processor may execute only a subset of the critical sections and the sequence may vary.

5.2.3 Analysis of the Models

Since we are primarily interested in the impact of the critical section modules on system performance, we can apply Norton's theorem for queueing networks [6] to the two queueing models developed above. This is done by effectively "shorting" out the non-critical modules and studying the throughput of the resultant networks. Specific systems with specific configurations of non-critical modules can then be studied by replacing the critical section modules by servers equivalent to the above networks.

The single server model of the global lock system can be solved using standard queueing techniques [12]. The multiple lock model can be solved using the mean-value analysis technique developed by Reiser [15]. This technique avoids the computation of the normalization constant and the resultant loss of numerical accuracy found in algorithms such as in Buzen [5].

5.3 Discussion of Results

Using the models and analysis techniques described above, the throughput of the multiple lock scheme is compared to that of the single lock scheme with the following choice for the service function of the deadlock avoidance module: $\lambda(m)=\lambda/(m-1)^{\alpha}$, where α is a variable parameter in the analysis. λ is a measure of the service rate of the deadlock avoidance module independent of the algorithmic complexity, and is expressed relative to $\mu=1$. It may thus measure the efficiency of the module implementation, higher values indicating higher efficiency and vice versa.

Figure 4 is a plot of the throughput of the multiple lock scheme relative to that of the single lock scheme as a function of m for various values of λ and α, and a 10-processor system. If the overhead is high ($\lambda=1$), the multiple lock scheme does worse regardless of the value of α. If $\alpha=2$, for $\lambda=10$ or 20, the multiple lock scheme does better by a factor of 2 to 2.5. The region of substantial performance improvement is rather narrow - m=2 to 4. This may be the case for a complex algorithm such as the banker's algorithm [8]. For a linear deadlock avoidance algorithm ($\alpha=1$), the multiple lock scheme does better by a factor of 2.5 to 3, and the decomposition choice is larger, - m=2 to 10. This would be the case if a simple algorithm, such as using a hierarchy of locks, is used.

Figure 5 is a plot of the same throughput increase for various values of n and α. It can be seen that the region of optimum decomposition is quite insensitive to the number of processors and is much more dependent on α. The actual performance itself is sensitive to n only in the region of optimum decomposition. It should be noted that in a real system, the performance may be more sensitive to the non-critical modules than the critical sections. Thus, analysing the model for a real system will involve taking the cross-section of the above curves at the design value of m. This will characterize the throughput of the Norton's equivalent of the set of critical sections as a function of the number of processors currently executing in them.

The above analysis may be summarized as below:

1. The multiple lock scheme may do at most 2 or 3 times better than the single lock scheme.
2. For the parameter ranges examined, the optimum decomposition is m=3 to 6.
3. The implementation efficiency of the deadlock avoidance module plays a crucial role in determining performance.

6. CONCLUSION

We have introduced a technique for the modelling of complex software structures for distributed systems. Application of such modelling and analysis techniques will yield a better understanding of the performance characterisitics of the various design alternatives in the construction of such systems. We have presented a simple example of the technique, which has yielded some insight into the problem of critical section configuration for maximizing throughput.

A great deal of work needs to be done to establish the technique as a viable tool. Analytical models such as the one in section 5 need to be better calibrated and validated. The impact of the underlying hardware configuration on the performance of such software systems needs to be understood. The performance issues of software implementation techniques need to be studied. We hope that this study will fuel research in these areas.

REFERENCES

1. Arnold, J.S., Casey, D.P., and McKinstry, R.H. Design of tightly-coupled multiprocessing programming. *IBM Sys. J. 13*, 1 (1974), 60-87.
2. Baskett, F., Chandy, K.M., Muntz, R.R., and Palacios, F. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM 22*, 2 (Apr. 1975), 248-260.
3. Baskin, H.B., Borgerson, B., and Roberts, R. PRIME - a modular architecture for terminal-oriented systems. Proc. AFIPS 1972 SJCC, Vol.40, AFIPS Press, Montvale, N.J., pp. 431-437.
4. Buzen, J.P. Queueing network models of multiprogramming. Ph.D. thesis, Division of Engineering and Applied Sciences, Harvard University, 1971.
5. Buzen, J.P. Computational algorithms for closed queueing networks with exponential servers. *Comm. ACM 16*, 9 (Sep. 1973), 527-531.
6. Chandy, K.M., Herzog, U. and Woo, L. Parametric analysis of queueing networks. *IBM J. of R. and D. 19*, 1 (Jan. 1975), 36-42.
7. Feeley, J.M. A computer performance monitor and Markov analysis for multiprocessor system evaluation. In *Statistical Computer Performance Evaluation*, W. Freiberger, Ed., Academic Press, New York, 1972.
8. Habermann, N. *Introduction to Operating System Design*. Science Research Associates, Inc., 1976.
9. Heart, F.E., Ornstein, S.M., Crowther, W.R., and Barker, W.B. A new minicomputer/multiprocessor for the ARPA network. Proc. AFIPS 1973 NCC, Vol.

42, AFIPS Press, Montvale, N.J., pp. 529-537.

10. Hughes, P.H., and Moe, G. A structural approach to computer performance analysis. Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 109-120.
11. Jones, A., Chansler Jr., R.J., Durham, I., Feiler, P., and Schwans, K. Software management of Cm* - a distributed multiprocessor. Proc. AFIPS 1977 NCC, Vol. 46, AFIPS Press, Montvale, N.J., pp. 657-663.
12. Kleinrock, L. *Queueing Systems, Vol. 1 and 2.* Wiley-Interscience, 1975.
13. McCredie, J. Analytic models as aids in multiprocessor design. Dept. of Computer Science, Carnegie-Mellon U., Pittsburgh, Pa., 1972.
14. Moore, C.G. Network models for large-scale time-sharing systems. Tech. Rpt. 71-1, Dept. of Industrial Engineering, University of Michigan, 1971.
15. Reiser, M. Mean value analysis of queueing networks, a new look at an old problem. Res. Rep. RC 7228, IBM Thomas J. Watson Res. Ctr., Yorktown Heights, N.Y., 1978.
16. Scherr, A.L. *An analysis of time-shared computer systems.* MIT Press, 1967.
17. Scherr, A.L. The design of OS/VS2 Release 2. Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 387-394.
18. Srodawa, R.J. Positive experiences with a multiprocessing system. *Comptg. Surveys 10,* 1 (Mar. 1978), pp. 73-82.
19. Swan, R.J., Fuller, S.H., and Siewiorek, D.P. Cm* - a modular multi-microprocessor. Proc. AFIPS 1977 NCC, Vol. 46, AFIPS Press, Montvale, N.J., pp. 657-663.
20. Wulf, W.A. HYDRA: the kernel of a multiprocessor operating system. *Comm. ACM 17,* 6 (June 1974), pp. 337-345.
21. Wulf, W.A., and Bell, C.G. C.mmp - A multi-miniprocessor. Proc. AFIPS 1972 FJCC, Vol. 41 part II, AFIPS Press, Montvale, N.J., pp. 765-777.

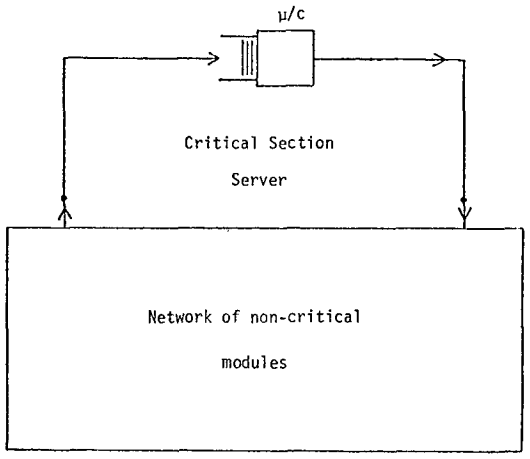
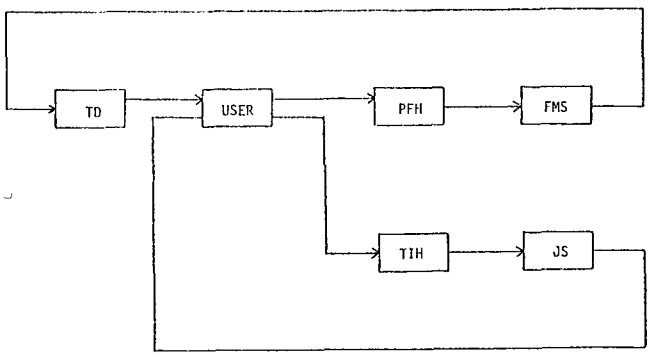


Figure 2. Queueing Model of a Single Lock System



TD: Task Dispatcher - chooses next task to use CPU from the READY queue.

USER: All user programs that use the CPU.

PFH: Page Fault Handler - invoked when a user task incurs a page fault.

FMS: File Management System routines.

TIH: Terminal Interrupt Handler.

JS: Job Scheduler - schedules a job for processing by placing it in the READY queue.

Figure 1. A Software System Model

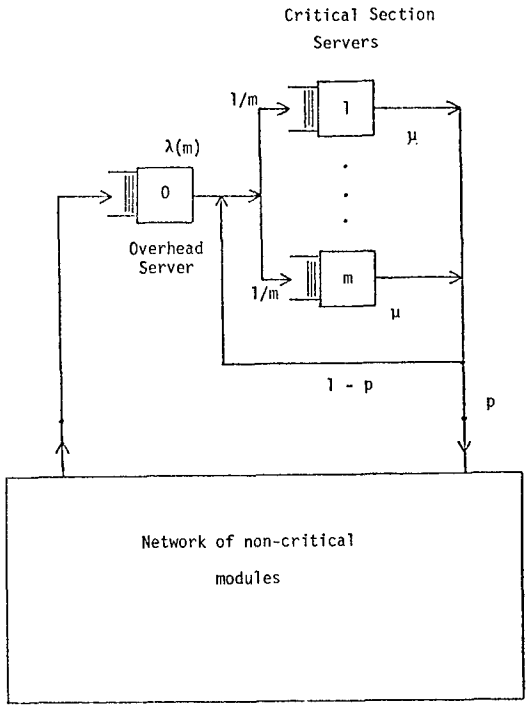


Figure 3. Queueing Model of a Multiple Lock System

FIG. 4 MULTIPLE LOCKS VS. SINGLE LOCK

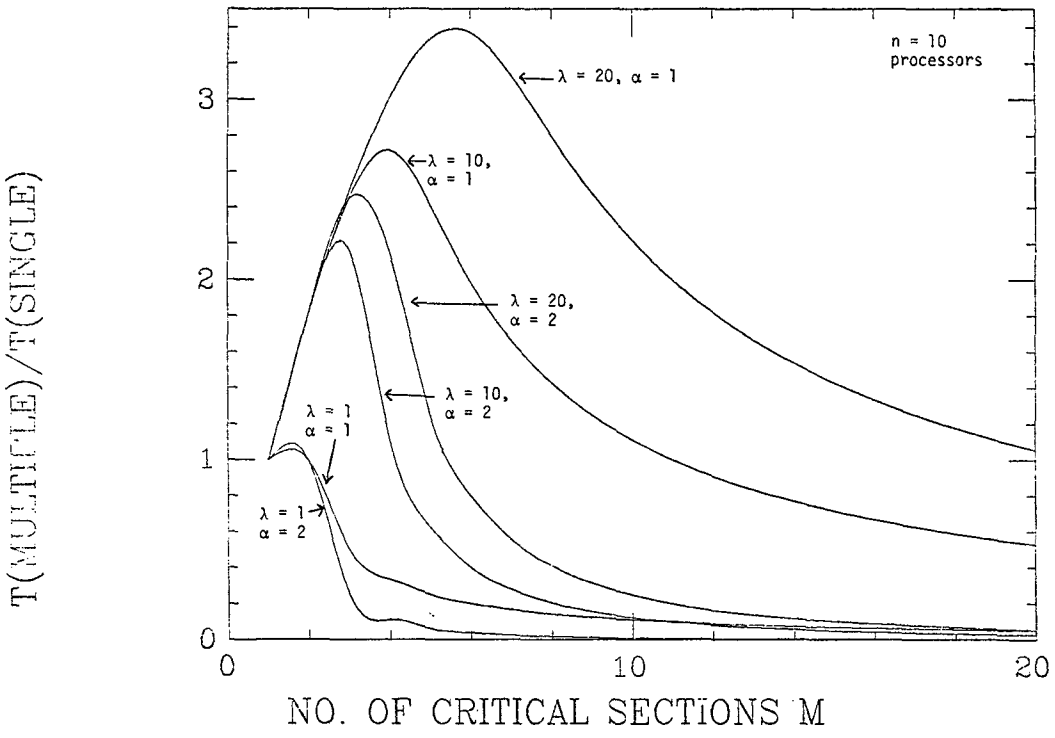


FIG. 5 MULTIPLE LOCKS VS. SINGLE LOCK