MAC TR-103

PERFORMANCE EVALUATION OF MULTIPROGRAMMED TIME-SHARED COMPUTER SYSTEMS

Allan Scherr

SEPTEMBER 1972

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 1095, and was monitored by ONR under Contract No. N00014-70-A-0362-0006.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified

1. ORIGINATING ACTIVITY (Corporate author) MASSACHUSETTS INSTITUTE OF TECHNOLOGY PROJECT MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP NONE

3. REPORT TITLE

PERFORMANCE EVALUATION OF MULTIPROGRAMMED TIME-SHARED COMPUTER SYSTEMS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

INTERIM SCIENTIFIC REPORT

5. AUTHOR(S) (First name, middle initial, last name)

AKIRA SEKINO

6. REPORT DATE SEPTEMBER 1972	7a. TOTAL NO OF PAGES 280	7b. NO OF REFS 59
8a. CONTRACT OR GRANT NO N00014-70-A-0362-0006 b. PROJECT NO c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) MAC TR-103 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) NONE	

10. DISTRIBUTION STATEMENT

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

11. SUPPLEMENTARY NOTES PH.D. THESIS, DEPT. OF ELECTRICAL ENGINEERING, AUGUST 1972	12. SPONSORING MILITARY ACTIVITY OFFICE OF NAVAL RESEARCH

13. ABSTRACT

This thesis presents a comprehensive set of hierarchically organized modular analytical models developed for the performance evaluation of multiprogrammed virtual-memory time-shared computer systems using demand paging. The hierarchy of models contains a user behavior model, a secondary memory model, a program behavior model, a processor model, and a total system model. This thesis is particularly concerned with the last three models. The program behavior model developed in this thesis allows us to estimate the frequency of paging expected on a given processing system. The processor model allows us to evaluate the throughput of a given multi-processor multi-memory processing system under multiprogramming. Finally, the total system model allows us to derive the response time distribution of an entire computer system under study.
Since all major factors (such as various system overhead times and idle times) which may decrease a system's computational capacity available for users' useful work are explicitly considered in the analyses using the above models, the performance predicted by these analyses is very realistic. A comparison of the performance of an actual system, the Multics system of M.I.T., and the corresponding performance predicted by these analyses confirms the accuracy of performance prediction by these models. Then, these analyses are applied to the optimization of computer systems and to the selection of the best performing system for a given budget. The framework of performance evaluation using these hierarchically organized analytical models guides human intuition in understanding the actual performance problems and provides us with reliable answers to basic performance questions on system throughput and response time.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0102-014-6600

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer System Models						
Performance Evaluation						
Time-Shared Computer Systems						
Operating System						
System Response Time						
System Throughput						
Program Behavior						
Virtual Memory						
Multiprogramming						
Multiprocessing						
Multics						
Operations Research						

MAC TR-103

PERFORMANCE EVALUATION OF MULTIPROGRAMMED TIME-SHARED COMPUTER SYSTEMS

Akira Sekino

September 1972

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095, and was monitored by ONR under Contract No. N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

3

PERFORMANCE EVALUATION OF MULTIPROGRAMMED TIME-SHARED COMPUTER SYSTEMS

by

AKIRA SEKINO

Submitted to the Department of Electrical Engineering
on August 28, 1972, in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy

ABSTRACT

This thesis presents a comprehensive set of hierarchically organized modular analytical models developed for the performance evaluation of multiprogrammed virtual-memory time-shared computer systems using demand paging. The hierarchy of models contains a user behavior model, a secondary memory model, a program behavior model, a processor model, and a total system model. This thesis is particularly concerned with the last three models. The program behavior model developed in this thesis allows us to estimate the frequency of paging expected on a given processing system. The processor model allows us to evaluate the throughput of a given multi-processor multi-memory processing system under multiprogramming. Finally, the total system model allows us to derive the response time distribution of an entire computer system under study.

Since all major factors (such as various system overhead times and idle times) which may decrease a system's computational capacity available for users' useful work are explicitly considered in the analyses using the above models, the performance predicted by these analyses is very realistic. A comparison of the performance of an actual system, the Multics system of M.I.T., and the corresponding performance predicted by these analyses confirms the accuracy of performance prediction by these models. Then, these analyses are applied to the optimization of computer systems and to the selection of the best performing system for a given budget. The framework of performance evaluation using these hierarchically organized analytical models guides human intuition in understanding the actual performance problems and provides us with reliable answers to most of the basic quantitative performance questions concerning throughput and response time of actual modern large-scale time-shared computer systems.

THESIS SUPERVISOR: Fernando J. Corbató
TITLE: Professor of Electrical Engineering

ACKNOWLEDGEMENT

I would like to express my appreciation to my thesis supervisor, Professor Fernando J. Corbató, for the time and effort he spent discussing this research with me, and in particular for his helpful comments which greatly improved the presentation of this thesis.

I am also indebted to my readers, Professors J. H. Saltzer and R. C. Larson, for their constructive review and comments.

Thanks are also due to my colleagues, D. H. Hunt and L. L. Scheffler, for their critical reading of earlier drafts of this thesis, and to my friends, H. Matsumoto for her assistance in typing this thesis and M. Watanabe for his assistance in drawing figures of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

THE PERFORMANCE EVALUATION PROBLEM

1.1. Motivation

Almost ten years have passed since the appearance of the first general-purpose time-sharing computer system marked by the CTSS system [C9]. This system is still in operation at M.I.T. although it is being replaced by later time-sharing computer systems because of its limitations on supporting sophisticated users. Many other systems have been developed in the effort to correct the deficiencies of earlier systems. Notably, the Multics system [C10,C12], the successor of CTSS, and the IBM 360 model 67 system [A3,L1] incorporated many elaborate ideas into their design to efficiently provide sophisticated users with their own virtual computers. These large-scale systems have stimulated the development of a number of smaller systems such as GE 235, RCA Spectra model 46, CDC 3300, DEC PDP-10, and XDS 940 and Sigma 7.

As pointed out by Rosen [R2], especially large-scale time-sharing

systems (the Mutlics system and the 360/67 system) have fallen far short of the performance anticipated in the initial design stage. The elaborate ideas incorporated into these systems, such as paging, segmentation, multiprogramming, multi-processing, and memory hierarchy, combine to create an enormously complex system and therefore it was hard to estimate how efficient such an ambitious system would be. Moreover, it was difficult to estimate the impact of the behavior of sophisticated user programs in this environment upon the system. In fact, it is not easy even now to predict the performance (e.g., throughput, response time, etc.) of such complex time-shared computer systems; it is generally believed that the most accurate prediction of the performance of a system in question is obtained by extrapolating the observed performance of "similar" existing systems. But a little investigation often reveals that each system is very different in hardware, software, and user characteristics from others. Therefore, other existing systems do not directly provide us with a reliable performance projection of a system, although an examination of performance differences of similar systems may be very interesting. Some examples of difficult questions on the performance of these modern large-scale time-shared computer systems are listed for reference in Table 1-1.

It is obvious that we need some sort of <u>comprehensive theory of performance evaluation</u> that is capable of answering these quantitative performance questions concerning hardware, software, and user characteristics of these computer systems.

Table 1-1 Some Difficult Performance Questions

- What is the best configuration to provide time-sharing service for 200 sophisticated users at a place such as M.I.T.?
- What is the maximum number of users that can be supported by a certain system of a given configuration, without discouraging them with excessively slow responses?
- What are the future procurement plans to improve the throughput of a certain system if that system must evolve? Should we purchase another processor, another unit of primary memory, or a faster secondary memory device?
- How sensitive is the system performance to the changing user characteristics?
- Does sharing affect the cost/performance of a given system very much?
- How much does multiprogramming improve the performance of a certain system?
- How can we keep the system performing optimally by tuning system parameters when a certain system overhead is halved?
- What is the optimum page size for a proposed system? How sensitive is the performance to the page size?

1.2. Computer Systems to be Studied

This thesis will be concerned with performance evaluation of these complex systems, i.e., multiprogrammed virtual-memory time-shared computer systems using demand paging. Before beginning a discussion of the performance evaluation problems, a brief description of such computer systems to be studied is given in this section, in order to avoid ambiguities associated with the complex structure of these systems.

A typical system is schematically depicted in Figure 1-1. The entire system is composed of a processing system and a finite population of interactive users at their terminals. Each interactive user of the computer system thinks for a while and then requests a computation (hereafter called a job) to be performed by the processing system, by typing a command line at his terminal. The job thus requested is received and first placed in the memory queue Q_m by the processing system.

The processing system is assumed to have more than one processor, and a two-level virtual-memory consisting of multi-unit primary memory (usually, core) as the first level and large secondary memory as the second, as shown in Figure 1-2. This multi-processor multi-memory processing system is assumed to have a two-dimensional address translation mechanism using both segmentation and paging [D2]. When this processing system finds some space in primary memory that can be allocated to a new user job, one of the jobs in Q_m is moved, under the First-Come First-Served (FCFS) discipline, to the processor queue Q_p. The jobs in Q_p are scheduled one by one, (usually) under the FCFS discipline, for a processor's service when one of the processors becomes available, and then each job is executed (in the running state of

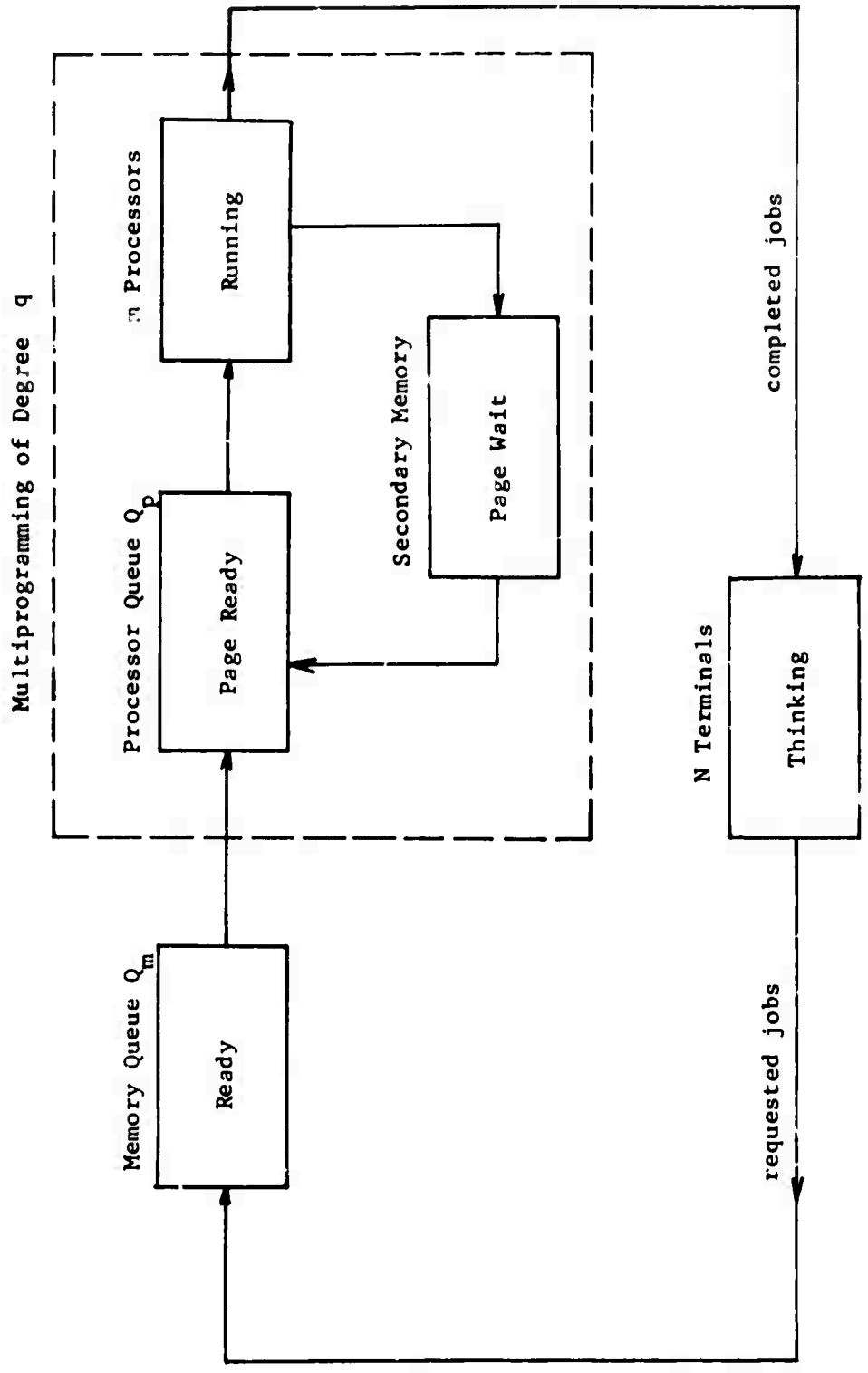


Figure 1-1 Multiprogrammed Virtual-Memory Time-Shared Computer System Using Demand Paging

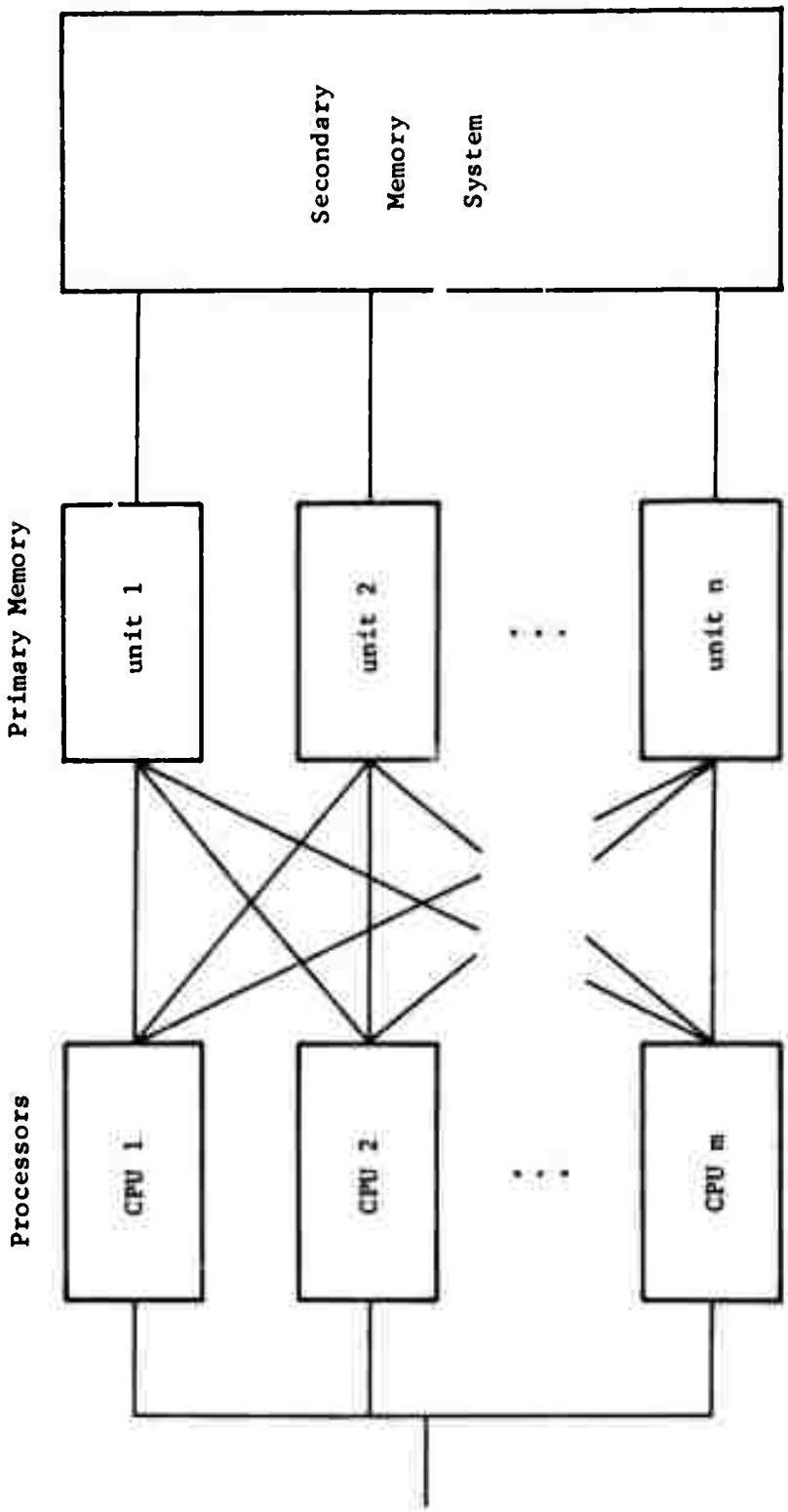


Figure 1-2 Multi-Processor Multi-Memory Processing System

Figure 1-1) by the processor until it encounters a part (page), of its program, that is missing in primary memory (a missing page fault is said to have occurred). Control of the processor is transferred from the user job to a supervisor module, named the page fault handler, which then requests the missing page to be brought from secondary memory for that user job, under a demand paging strategy. At this point, the processor becomes available for another job in Q_p. On the other hand, the page-faulted user job enters the page-wait state, waiting for the requested page. When it is eventually transferred from secondary memory to primary memory, the user job reenters the processor queue Q_p (or the page-ready state), becoming ready for another service by a processor. Because there usually exist several jobs competing for service by processors, each job is executed in an interleaved fashion, cycling through the above three states many times. When the job is eventually completed, the result is returned, as a system response, to the terminal user who requested this job, and he then starts thinking about which job to request next.

A series of jobs being requested by a user is generally called a process. It is assumed that a user can request a job only after he receives the system's response to his preceeding job request. If a user's job is located within the section of Figure 1-1 surrounded by a broken line, his process is said to be eligible because it is eligible for a processor's service; otherwise it is said to be ineligible. The number of eligible processes existing at any instant should be determined by considering, at least, their demand for primary memory available to eligible user processes, in order to avoid thrashing,

i.e., excessive competition for primary memory space leading to a less than optimum use of system resources. The computer system to be considered is actually assumed to have a simple (static) mechanism to avoid thrashing, called multiprogramming of degree q, which allows a maximum of q (a constant) processes to be simultaneously eligible.

Summing up, the computer system to be studied is characterized by the parameters shown in Table 1-2. This table lists only major system parameters which are believed to be important in performance evaluation. The precise definition of these system parameters will be given as they are introduced in the thesis.

Table 1-2 Major System Parameters

System Hardware:

number of processors
size of primary memory
number of primary memory units
size of secondary memory*
speed of processors and primary memory
speed of secondary memory
channel organization and capacity*
memory cycle interference

System Software:

scheduling algorithm*
multiprogramming algorithm (degree of multiprogramming)
page replacement algorithm
page size
paging overhead
miscellaneous overheads
data-base lockout

Users and Their Programs:

number of interactive users
types of interactive users*
user's think time
execution time required by each user interaction (job)
program size
reference pattern of programs
degree of sharing among user processes

---

* Comments will be given in Section 1.6.

1.3. Nature of the Problem

As the complexity of time-shared computer systems grew, the performance evaluation of these costly systems became vital. Therefore, many performance evaluation techniques have been developed. Lucas [L4] recently classified these techniques, according to their purposes of performance evaluation, into the following three categories.

(A) System selection techniques

(B) Performance projection techniques

(C) Performance monitoring techniques

The first category of techniques is intended to select a particular system from various systems available from many manufacturers when system performance is a major criterion to make a purchase order. Lucas suggests that synthetic programs (a comprehensive set of benchmark programs, so to speak) are most appropriate in this category. The second category of techniques are intended to estimate the performance of a system that does not yet exist or only partially exists. Within this category, Lucas claims that simulation is most powerful because of its relative flexibility in modeling complex systems. The last category of techniques is intended to collect data on the actual performance of an existing system. These data are used to identify the operating condition of the system so as to forecast the impact of changes in the system, possibly with the help of the techniques of the second category. Monitoring uses both hardware and software methods.

According to this classification, the performance evaluation techniques to be presented in this thesis belong to the second category

and are especially concerned with the following three general performance problems of modern large-scale time-shared computer systems.

(1) performance prediction for a given configuration

(2) performance optimization for given (hardware) configuration

(3) configuration selection for a given budget

Performance prediction means a functional expression of system performance (e.g., throughput, response time, etc.) in terms of various system parameters concerning hardware, software, and user characteristics of the system; a performance prediction technique estimates the performance of a given (hardware-software-user) configuration. Performance optimization deals with the problem of how to improve the performance of a given system without changing its hardware: the system performance is optimized with respect to certain adjustable parameters of the operating system such as the degree of multiprogramming, the page size, and various resource allocation algorithms, without changing the hardware cost. Lastly, configuration selection[1] is the problem of deriving the optimum system configuration which attains the best performance, constrained by a given purchase budget; this involves an optimization of hardware configuration as well as that of the operating system.

---

[1]The configuration selection problem is simpler than the system selection problem because we need not be concerned with various differences of different systems in machine structure such as word length, width of data transfer paths, machine instruction repertory, etc.

Unfortunately, simulation approaches, recommended for this category of problems by Lucas, require enormous amount of development effort and operating cost. This becomes especially apparent if a simulation model includes many micro and macro operations in evaluating the performance of various possible configurations of a certain system. Moreover, simulation approaches tend to lack the capability of yielding a general insight into the cause-and-effect relationship of performance problems. To compensate these weaknesses of simulation approaches, this thesis explores the possibility of using analytical models in tackling the above three general problems of computer system performance.

1.4. Review of Analytical Computer Models

Because analytical models will be developed in this thesis to project the performance of large-scale time-shared computer systems, this section briefly reviews and examines the analytical computer models developed as performance projection techniques until now.

These models are stochastic in nature and their analysis usually involves queuing theory. Because it is generally difficult to include many mutually related system parameters (see Table 1-2) in a mathematically tractable model, most of these analytical models are concerned with subsystem behaviors. They may be classified into:

(a) processor scheduling models

(b) secondary memory models

(c) multiprogramming models

(d) program behavior models

Processor scheduling models usually include a single processor, an infinite (sometimes, finite) user population, and a scheduling algorithm (e.g., First-Come First-Served, Round-Robin, Processor Sharing, Feed-Back), and aim to study the effect of a scheduling parameter (e.g., quantum length) upon response time of a user job (conditioned upon its execution time). These models are most abundant among the above four classes and were extensively surveyed by McKinney [M2]. For example, general analyses by RR, PS, and FB algorithms were presented respectively by Chang [C2], Baskett [B2], and Schrage [S5].

Secondary memory models are intended to estimate the (average) time to fetch a block of information from a secondary memory device (e.g.,

disk, drum, bulk core, etc.) used as part of virtual memory. Fewer papers have been published in this area. Coffman's work [C5,C6] is, however, worthy of a special note. He derived the average time required to fetch a page of information from a sector drum under demand paging, using an embedded Markov chain technique [C8].

Multiprogramming models are relatively newer class of models and were analyzed by Smith [S7], Wallace and Mason [W2], Buzen [B6], Moore [M3], Rice [R1], et al. These models combine processor scheduling models (with a FCFS discipline) and secondary memory models, making a significant step toward a "system model". They include several (a fixed number) jobs under multiprogramming, each of which is serviced by a processor and then by one of the secondary memory devices in a cyclic fashion, as schematically shown in Figure 1-3. Service time of each server(processor, I/O devices) is usually assumed to be exponentially distributed and branchings are specified by constant (unconditonal) probabilities (e.g., p_i of Figure 1-3). This class of models aims to evaluate the effect, upon the server utilizations, of the hardware configuration as well as of the number of jobs under multiprogramming, i.e., the degree of multiprogramming. Smith and Wallace et al formulated their problems as a Markov process and numerically obtained the perfomance of these models using a powerful queue analysis program called RQA [W1]. On the other hand, Buzen, Moore, Rice, and others explicitly analyzed the performance of their multiprogramming models using Gordon and Newell's method [G2] or Jackson's decomposition thorem concerning queuing networks [C7,J1]. The effect of non-exponential service times

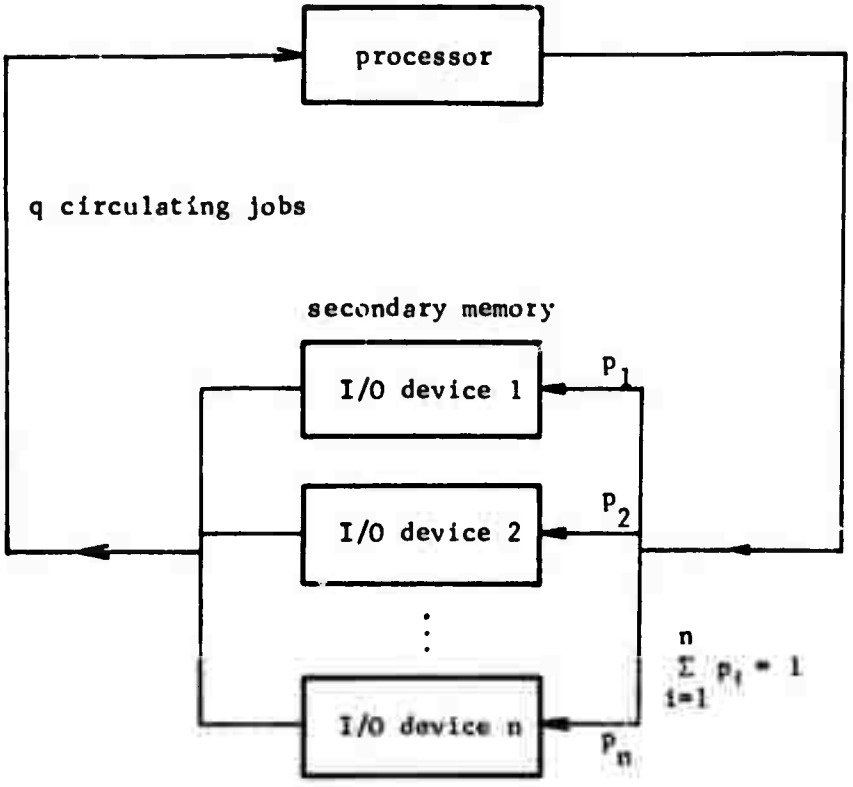


Figure 1-3 Typical Multiprogramming Model

was examined by Gaver [G1]. The problem of optimizing the degree of multiprogramming under a simple assumption was studied by Wallace and Mason. The effect of system overhead was carefully analyzed by Lewis and Schedler [L2].

Finally, there exists another class of analytical models, i.e., program behavior models. These models are intended to study the paging behavior of programs being executed by a processor within a limited amount of primary memory. There seem to be at least two different popular approaches in modeling program behavior, Markovian program models [K1] and Denning's working-set program models [D1, D3]. The paging behavior of programs is modeled by a Markov chain in the first approach, while the paging behavior is measured by the number of distinct pages referenced during a given time interval in the second approach. Both of these attempt to derive a success function, i.e., the probability that a reference is made to a page already in primary memory expressed as a function of the primary memory size, for a certain page replacement algorithm. However, no practically useful results have been obtained about the dynamic paging behavior of programs.

Although many analytical models have been developed and analyzed, there still exists a general consensus especially among system designers that these models are not good enough to answer the performance questions that they face (see Table 1-1). This is probably due not only the system designers' insufficient efforts to try to use studied models in understanding the behavior of an actual system, but also to certain important defects of these analytical models; most of the analytical models developed until now have serious weaknesses in at least some of

the following aspects.

(1) System behavior versus subsystem behavior; almost all analytical models attempt to represent only a subsystem[1], and this is the very reason why many system analysts abandon an analytical approach and choose a costly simulation approach. Perhaps, we need another model which is capable of combining all subsystem models meaningfully so as to estimate the performance of the entire system.

(2) Paging activities and multiprogramming capability; this aspect of the modern large-scale time-shared computer system is not considered in the processor scheduling models, but is usually considered in the multiprogramming models. Therefore, we should favor the latter models. The program behavior models developed until now are not good enough to consider a multiprogrammed situation.

(3) Space-domain considerations; the size of primary memory is not considered in the multiprogramming models. Moreover, these models consider only a single-processor system, in spite of the increasing interest in multi-processor multi-memory systems.

(4) Overhead considerations; almost all models do not consider the overhead of system programs, although a major implication

---

[1]There are some analytical models of an entire system [S4,T1]. However, these models are apparently too simple to reasonably represent multiprogrammed time-shared computer systems using demand paging.

of modern time-shared computer systems with paged memory is considered to be system overhead.

(5) Good choice of performance measure; the improvement of utilization factors of system resources is not an ultimate goal. Better performance measures are the system throughput, namely, the amount of user computation accomplished by the entire system per unit time, and the system response time experienced by interactive terminal users.

We will attempt to correct these defects of the existing analytical models in this thesis research so that we can realistically attack performance prediction problems, performance optimization problems, and configuration selection problems.

1.5. Approach to be Taken

Modeling of multiprogrammed virtual-memory time-shared computer systems using demand paging must include many system parameters, as seen in Table 1-2. It is actually this multiplicity of interacting system parameters that has prevented analytical computer models from being a reasonable representation of these systems as their entirety; a straightforward inclusion of all important system parameters in an analytical model would not allow mathematical tractability in its analysis. Interestingly, it has been observed also in simulation approaches that a simulation model which attempts to account for both "macro-operations" (e.g., user's think time) and "micro-operations" (e.g., state transitions of eligible processes, dynamic program behavior) can be prohibitively expensive to run [B4]. On the contrary, the results obtained from the oversimplified analytical models cannot convince the system designers of their practical value as a design aid. Therefore, we are in a dilemma of mathematical tractability and multiplicity of interacting system parameters.

However, a solution to this dilemma may be found by noting that all system parameters do not necessarily interact and that the system behavior involves various activities of different time scales. For example, user behavior (e.g., think time, jobs to be requested, etc.) tends to be logically independent of dynamic program behavior unless the user is encouraged to be particulary careful about the program behavior. We find that the activities of the entire system may be categorized into "macro-activities" (e.g., memory queuing of user jobs, terminal user behavior---activities on the order of seconds), "micro-activities" (e.g , state transitions of eligible processes---millisecond

activities). Thus, we can obtain several semi-independent activities of the entire system which interact only through their prime inter-relationships. When all the semi-independent activities are connected according to their prime inter-relationships, we generally obtain hierarchically organized system activities. One such result is shown in Figure 1-4. The approach to be taken in analyzing the behavior of the entire computer system in this thesis is to develop a separate model for each semi-independent activity of this hierarchy in such a way that any model has, as its input parameters, at least one system parameter produced (as an output) by each immediate lower-level model, if such models exist, as well as its own set of system parameters. A system parameter produced as an output of a model can be regarded as a performance index of that model. If such a performance index is not the output of the top-level model, it is called an intermediate performance measure. Now it is clear that the entire computer system is represented by a <u>set of hierarchically organized modular models</u>.

We will use a particular hierarchy of modular models depicted by Figure 1-4. It includes the following five modular models.

(1) program behavior model(s)

(2) secondary memory model

(3) user behavior model

(4) processor model

(5) total system model

The program behavior model(s), to be developed for the behavior of

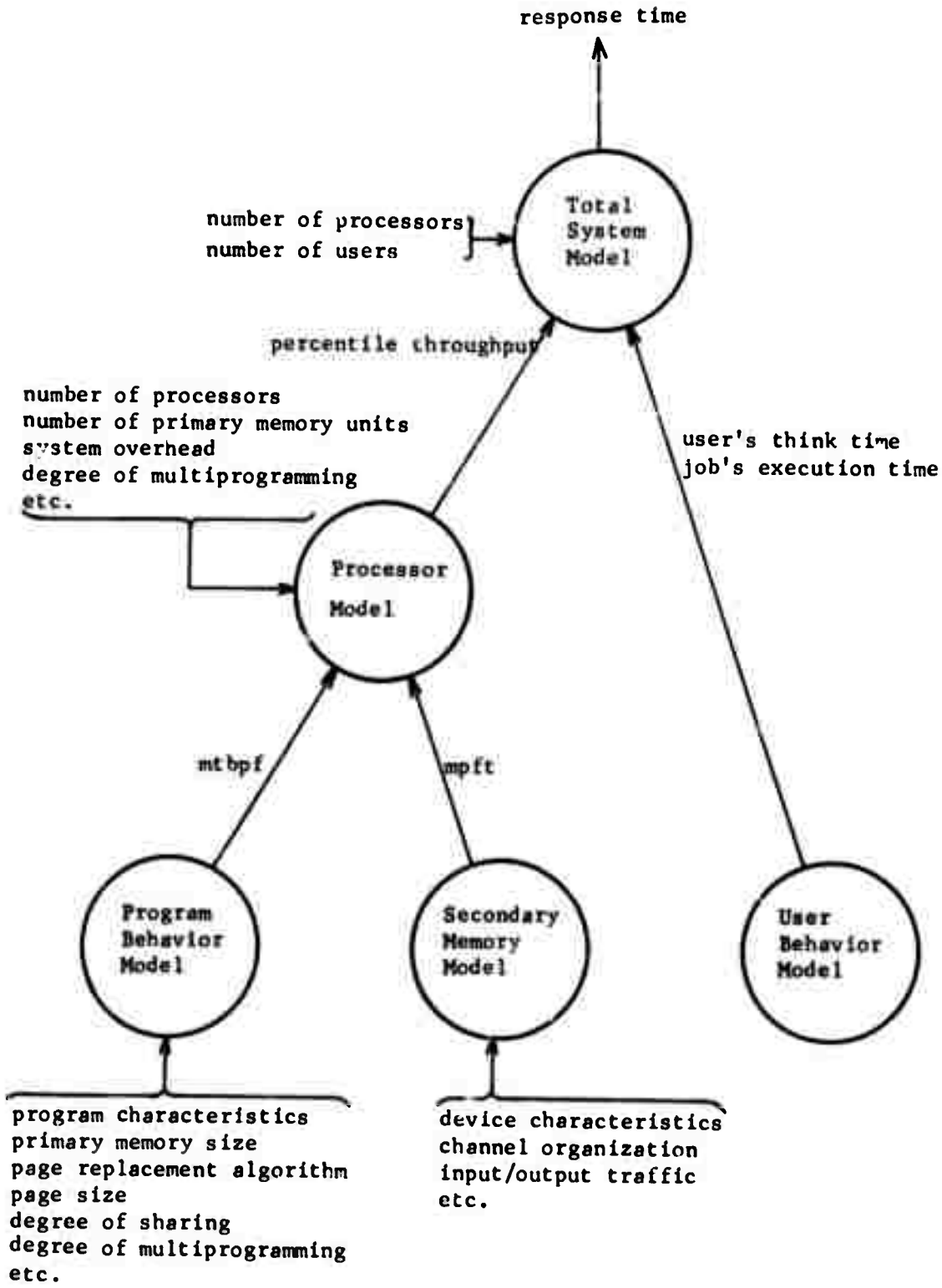


Figure 1-4 Hierarchical Organization of Modular Models

programs being executed on the processing system, aim(s) to derive the mean length of program execution before a page fault (that is, the mean time between page faults denoted as "mtbpf") as a function of various system parameters such as program characteristics, the primary memory size, the page replacement algorithm being used, the page size, the degree of sharing among eligible processes, the degree of multiprogramming, etc. The secondary memory model aims to derive the mean length of time required to fetch a page of information from the secondary memory system (that is, the mean page fetch time denoted as "mpft") as a function of memory device characteristics, the channel organization, the input/output traffic intensity, etc. Because the performance of the secondary memory system (disk, drum, bulk core, or a combination of these) can be reasonably predicted by a simple queuing model like an M/G/1 queue [C8], or by the fairly complex drum model of Coffman [C5, C6], the thesis will not develop a new model in this area. The user behavior model exists for a user's function to think and request a computation called a user job. In this thesis, we simply assume that think time, which a user needs to decide and request a job, and execution time required by such a user job are both exponentially distributed because they are known to be roughly exponential [S4]; we will not be concerned with any internal mechanism of these interactive users (e.g., under which condition a user's think time tends to be exponential).

Immediately above the program behavior model and the secondary memory model, there exists the processor model. This is a multiprogramming model improved in such a way that the performance of a multi-

programming mechanism implemented on a multi-processor multi-memory processing system (see Section 1.2.) can be measured in percentile throughput, i.e., the percentage of the system's computational capacity utilized for users' useful work. This means that the objective of the processor model is to derive the percentile throughput of the system as a function of the number of processors and primary memory units, their operating speed, the speed of secondary memory, various overheads of the operating system, the degree of multiprogramming, user process behavior, multi-processor interferences (such as memory cycle interference and data-base lockout), and so on. Particularly as for the user process behavior and the speed of secondary memory, it will be assumed that the time between page faults and the page fetch time are both exponentially distributed with means predicted by the analyses of the program behavior model and the secondary memory model respectively. Because percentile throughput turns out to be linearly proportional to the throughput of the processing system, i.e., the average number of user jobs completed per unit time (this relation will be shown later in Chapter 3), the derivation of percentile throughput using the processor model is called a throughput analysis. The throughput analysis using the above multiprogramming model aims to overcome the defects of the existing multiprogramming models concerning the space-domain considerations, the overhead considerations, and the performance measure.

At the top level, there exists the total system model. This model combines all other models in the hierarchy, and aims to derive a distribution of the system's response time as an explicit function of the

number of processors, the number of interactive users, the percentile throughput predicted by the processor model, and the user characteristics such as user think time and execution time of a user job both of which are assumed to be exponentially distributed. This means that it will be possible to derive the percentile response time, i.e., the time limit which guarantees that a certain proportion (e.g., 90 percent) of response times is shorter than that limit, as a function of various system parameters describing the configuration of the computer system. Therefore, the result to be presented is much more informative than the result of the average response time, because the fluctuations of response times around the average can be accurately predicted. It seems that this is the first derivation of the distribution of response time of a multiprogrammed time-shared computer system using demand paging.

It should be pointed out that the response time and the system throughput predicted in this approach reflect not only processor hardware characteristics, system software (operating system) characteristics, and user behavior characteristics, but also secondary memory characteristics and program behavior characteristics. This was made possible by the use of a set of hierarchically organized modular models each of which involves only a few system parameters, while the entire computer system includes more than twenty important system parameters. This thesis will consider all the system parameters of Table 1-2 except those starred, in evaluating the performance of multiprogrammed virtual-memory time-shared computer systems using demand paging, explained in Section 1.2.

Aside from the fact that this hierarchical organization permits mathematical tractability and the existence of such a multiplicity of interacting system parameters, it has the following equally important aspects.

(1) This organization allows human understanding of the very complicated behavior of modern complex computer systems and therefore provides us with an insight into the cause-and-effect relationships existing in such systems.

(2) If the behavior of a subsystem becomes known through the experimental or monitoring studies of a partially existing system, the behavior of the corresponding model can be replaced by the actual behavior of that subsystem. Therefore, available information about subsystems can be usefully utilized in predicting the performance of the entire system with an increased accuracy.

(3) Change in subsystem configuration (e.g., secondary memory configuration), expected especially on evolving computer systems, does not require an overall change of the entire analysis; only the parameters of that subsystem need to be modified.

1.6. Some Comments about the Models

Comments may be in order as to why some of the system parameters of Table 1-2 are out of consideration in this thesis. In studying the paging activities under multiprogramming, this thesis assumes a two-level virtual memory consisting of primary and secondary memories. In this context, the key parameter of secondary memory is not its size but its speed, and therefore it is simply assumed in the thesis that the secondary memory system is large enough to store all the system and user programs; we do not explicitly consider the size of secondary memory. As for the channel considerations, it will be assumed for simplicity of the analysis that the processing system is not channel-limited. Therefore, the dynamic queuing delay associated with channel service will not be explicitly considered, but some fixed service time by channel may be included as part of the page fetch time of secondary memory.

Scheduling algorithms of the system described in Section 1.2 are all FCFS and any alternative algorithm will not be considered. But, as is well known, actual systems usually employ more sophisticated algorithms. For example, the Multics system uses a refined variation of FB scheduling algorithm [01] with a pre-paging and post-purging technique (rather than strict demand paging) for jobs in the memory queue. It should be realized that what we want to accomplish in this thesis is to develop a performance evaluation methodology which is capable of deriving percentile throughput (or system throughput) and response time of a "reasonably basic" system like the one described as a typical system in Section 1.2, as a function of various important system parame-

ters. Then, it becomes clear that inclusion of an alternative scheduling algorithm like a RR or FB algorithm (for jobs in the memory queue) in the proposed framework of the performance evaluation theory is not very attractive, because it is known in queuing theory [B2,C7] that (overheadless) scheduling algorithms cannot affect percentile throughput or average response time if the execution time of user jobs is only probabilistically known and follows an exponential distribution, as assumed in Section 1.2 and as observed on the Multics system, for example; scheduling algorithms however can affect the variance of response time because of their intended favoritism for certain (e.g., short) jobs at the expense of others [C7]. Percentile throughput and average response time only deteriorate if the process (job) switching overhead associated with quantum run-out[1] of the alternative scheduling algorithm is not negligible. (Note that the introduction of pre-paging and post-purging mainly aims to reduce this overhead.) Therefore, in making a hard effort to include as many "important" system parameters (those which affect percentile throughput or average response time)

---

[1]Under a RR algorithm, for example, a fixed amount of time, called a quantum, is given to each process becoming eligible. The process is allowed to remain eligible until it uses up this amount of processor time. If this process needs more processor time to complete its job, it becomes ineligible and joins the end of the memory queue. This state transition path is not included in Figure 1-1, because we assume a FCFS algorithm, i.e., the RR algorithm with a infinitely large quantum.

as possible in the framework of this approach, alternative scheduling algorithms will not be considered; such a refinement may be applied after the computer system under study is optimized with respect to percentile throughput.

For jobs in the processor queue, the Multics system uses a preemptive scheduling algorithm (to accomplish a biased primary memory allocation [B3]) with a dynamic eligibility control mechanism based on a working-set estimate of user processes [O1] (rather than the static eligibility control mechanism of Section 1.2). This elaboration does improve percentile throughput, but we consider it as a refinement of the FCFS scheduling (i.e., an unbiased primary memory allocation) with the static eligibility control mechanism. Therefore, these details will not be considered in this thesis.

Finally, it should be mentioned that many random variables are assumed to be exponentially distributed not simply because of mathematical amenability but, more importantly, because of experimental evidence. The user's think time measured by Scherr [S4] is roughly exponential, and the execution time of user jobs being monitored by a built-in meter of the Multics system is almost exponential. Page fetch time tends to be close to an exponential distribution especially if secondary memory consists of a combination of a frequently-used high-speed device (e.g., drum) and a less-frequently-used low-speed device (e.g., disk). The distribution of time between page faults has not been measured on the Multics system, but it is very probable that this distribution appears like an exponential distribution. These observations of MIT systems, of course, do not necessarily lessen the importance of extending the

exponential assumptions to more general ones. In fact, the execution time of user jobs of some other systems is reported to be more like a hyper-exponential distribution or a Weibull distribution, possessing a large coefficient of variation [A2].

1.7. Thesis Organization

This thesis takes a bottom-up approach in describing the models developed for hierarchically organized subsystems. Each chapter of this thesis presents results obtained for a different subsystem.

Chapter 2 is devoted to the studies of dynamic program behavior, and presents several program behavior models evaluating mtbpf as a function of various system parameters. Chapter 3 is dedicated to the studies of the processing system, and presents a result of the throughput analysis using the processor model, i.e., a model of the multiprogramming mechanism implemented on the multi-processor multi-memory processing system. Chapter 4 is devoted to the studies of the response time characteristics of the entire computer system, and presents the result of the response time analysis using the total system model formulated as a queuing process. Thus, Chapter 2 through Chapter 4 collectively consider the performance prediction problems, i.e., the problems of expressing computer system performance as a function of various system parameters describing its configuration.

Chapter 5 presents many numerical results obtained by these analytical models as well as the actual performance data collected from the Multics system of M.I.T. The validity of these models is first examined and then the effect of various system parameters upon the overall system performance (i.e., the system throughput and the system response time) is numerically evaluated using these models. The problem of optimizing a given computer system with respect to certain operating system parameters (the degree of multiprogramming and the page size) and the problem of deriving the best performance system for a given budget

are realistically considered. This chapter tackles all the performance evaluation problems mentioned in Table 1-1 numerically.

The last chapter, i.e., Chapter 6, summarizes the problems solved in this thesis and those that still remain to be solved. Throughout the thesis, a stochastic modeling approach is used. This thesis therefore represent an application of some known techniques (or their slight variations) in Markov process theory and queuing theory to performance evaluation problems of computer systems. The emphasis, however, is on the identification of the actual performance problems and the development of a framework of performance evaluation methodology.

A reader who is intere ted simply in finding out the application of the framework of performance evaluation developed in this thesis to the performance problems of actual computer systems is suggested to read Chapter 5 immediately after this chapter. If a reader is interested in understanding modeling techniques for computer systems, he should read Chapter 2 through Chapter 4 carefully. Each of these three chapters can be read individually without much trouble. If he decides to read in this way, the notation and glossary included at the end of this thesis may be helpful. Chapter 6 is useful to those who are searching for research topics in the area of computer system performance evaluation.

CHAPTER 2

PROGRAM BEHAVIOR ANALYSIS

2.1. Introduction

Virtual memory computer systems have enabled the memory system to appear to their users (programmers) as if it is virtually infinite in size. Therefore, the nasty problem of carefully overlaying programs within a relatively limited primary memory space has been removed from the user's programming considerations. Virtual memory is assumed, throughout this thesis, to be implemented by "segmentation" and "paging" [D2], on a two-level physical memory system consisting of primary memory and secondary memory.

These computer systems have a mechanism for translating program-generated addresses into the correct physical memory addresses [D2]. The set of program-generated addresses is called the virtual address space (virtual memory) and the set of physical memory addresses the physical address space (physical memory). Segmentation organizes the

virtual address space into blocks, called segments, of arbitrary size. By allocating each program to its own segment, programs can have their own "linear" virtual address space within themselves. This means that a processor accesses each word of a program residing at a certain location of the virtual address space, using a two-component address (two-dimensional address) consisting of a segment name (segment number) and a word name (word number). Paging further organizes each segment into blocks, called pages, of a fixed size (usually 1,024 or 512 words). This means that a word name is represented by a page number and an offset. Each segment usually has several pages of information (procedure and data).

Correspondingly, the physical memory is organized into equal-size blocks of locations, known as page-frames, which serve as sites of residence for pages of segments. Because a processor can execute only that portion of a program (segment) which resides within primary memory which is relatively limited in size, the operating system must exercise a special algorithm, called a paging algorithm, to keep only the pages being needed for a progress of program execution in primary memory at all times, by transferring pages of the program back and forth between primary and secondary memories. The paging algorithm decides when to fetch a page from secondary memory and which page to be removed from primary memory when one of the pages in there must be replaced by the page to be fetched. If a page is fetched only on demand, i.e., only after that page is found "missing" in primary memory in the course of program execution (a page fault is said to have occurred), then the

fetch rule is said to be demand paging; otherwise, it is said to be pre-paging. The rule used to select a page for removal is called a page replacement algorithm. These two functions (page replacement and fetching) of the paging algorithm are carried out by a supervisor program, named a page fault handler, which is a part of the "resident" supervisor.

In fact, the processing of a page fault requires some other things such as the bookeeping of the page table, the initiation of a channel program, the handling of a paging interrupt, etc. [C11]. These supervisory operations required to process a page fault are collectively called the paging overhead. From the above explanation, it should be clear that each burst of continuous program execution, i.e., the running state of each elegible process (see Section 1.2.), consists of (at least) user program execution and paging overhead execution. The length of this continuous program execution in the running state is called a time between page faults (abbreviated as tbpf) and the length of user program execution within tbpf is called a headway between page faults (abbreviated as hbpf). The means of these variables are respectively called the mean time between page faults (mtbpf) and the mean headway between page faults (mhbpf).

As explained in Section 1.2., the computer system to be studied is under multiprogramming of degree q. This means that q eligible (user) processes compete for service by a processor and for the use of page-frames of primary memory. Because segmentation enables any segment to be shared (a single copy of a segment in virtual memory can be simultaneously used by different users), some of these page-frames are physically shared by these eligible processes. Therefore, the efficiency of

primary memory is heightened by segmentation. However, some part of the primary memory space is not available to non-resident programs (programs whcih are not always resident in primary memory) invoked by eligible processes, because resident supervisor programs stay at certain physical memory addresses at all times. This situation is schematically depicted in Figure 2-1. The total primary memory space with M page-frames is divided into two areas, i.e., the area with m_r page-frames for resident supervisor programs and the area with m_n page-frames for non-resident programs (user programs and non-resident supervisor programs). Eligible user processes execute not only non-resident programs but also resident (supervisor) programs, and therefore they can potentially utilize any part of the primary memory space.

Dynamic paging behavior of user processes in this environment is known to have a great impact on the system's overall performance [K2] and therefore it has been extensively studied by many researchers. It is believed that at least the following system parameters are significant in determining the performance of programs, which is usually measured by the mhbpf of user processes.

(1) program characteristics (size, reference pattern)

(2) primary memory size (m_r, m_n, M)

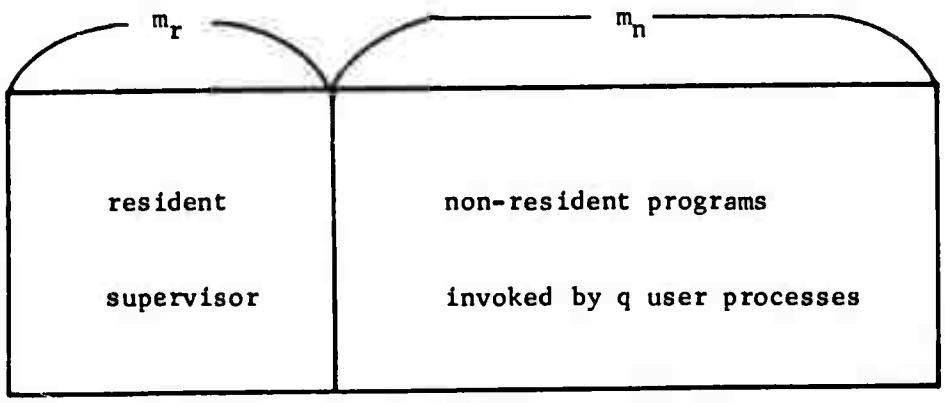
(3) paging algorithm

(4) page size

(5) degree of sharing among user processes

(6) degree of multiprogramming (q)

Belady [B3], Brawn and Gustavson [B5], Coffman and Varian [C4], Hatfield



$$M = m_r + m_n$$

Figure 2-1 Allocation of Primary Memory Space

and Gerald [H1], Tsao, Comeau, and Margolin [T2], and Baer and Sager [B1], among others, have experimentally investigated the effect of various system parameters upon the mhbpf (or its equivalent) of user processes. On the other hand, Denning [D1,D3], Mattson, Gecsei, Slutz and Traiger [M1], Chang [C1], Aho, Denning and Ullman [A1], King [K1], Woolf [W3], and others have carried out analytical studies of the dynamic behavior of user processes.

Research performed by Denning, Woolf, and King represents three different analytical (probabilistic) approaches to the problem considered in this chapter, and therefore some comments about their results are in order. Denning defined the working set, $W(t,\tau)$, of a user process at time t to be the set of pages that the process has referenced during the time interval $(t-\tau,t)$, and he has demonstrated the usefulness of his model using the working set as an aid for guiding one's intuition in understanding program behavior and possibly as a basis for further program behavior analysis. Woolf considered program behavior to be an execution of a series of loops and successfully evaluated the effects of program characteristics, primary memory size, and page size. However, this elaborate model has not been validated against actual program behavior. On the other hand, King formulated program behavior as a Markov process in evaluating the effect of paging algorithms as well as those of program characteristics and primary memory size. These analytical studies do explain various behaviors of programs experimentally observed, but none of these has proved to serve as a practical design aid to predict a program's quantitative performance (e.g., mhbpf) as a function of the system parameters mentioned above. In particular, the

effects of sharing and multiprogramming have never been fully studied.

In the following sections of this chapter, four program behavior models mostly using a Markov process approach will be developed to study the effects of those six system parameters mentioned above upon the mhbpf (or its equivalent) of a program (a user process). All of these four models consider at least the effect of program characteristics and of primary memory size, but each of them concentrates on the evaluation of the effect of a particular system parameter. The models to be presented in Sections 2.2, 2.3, and 2.4 are especially concerned with the effect of the paging algorithm in use, the effect of the primary memory size, and the effect of the page size, respectively. The model to be presented in Section 2.5 considers a multiprogramming environment and is particularly concerned with the effect of the degree of multiprogramming and that of sharing among user processes under multiprogramming.

Readers who are only peripherally interested in the analysis of program behavior may skip Sections 2.2, 2.3, and 2.4, and directly move to Section 2.5, which describes a macroscopic paging performance model for multiprogramming. The result of this section will be extensively used in the succeeding chapters. Sections 2.2, 2.3, and 2.4 present detailed analyses of dynamic program behavior and can be read individually without difficulty. However, a reader can grasp the basic approach used in this chapter more thoroughly by reading them in the order of these sections.

2.2. First-order Markov Model Applied to the PRA Studies

In Section 2.2., we present a program behavior model developed to study the effect of page replacement algorithms (abbreviated as PRAs) as well as those of program characteristics and primary memory size, upon the mhbpf of a user process operating on a multiprogrammed virtual memory time-shared computer system using demand paging[1]. To simplify the analysis, we assume that the primary memory space (m_n) is partitioned into q equal-size areas, one for each eligible process, and that the PRA under study is exercised within the area reserved for each user process (this kind of PRA is called a local PRA as opposed to a global PRA [D2]). Then, the analysis of program behavior under multiprogramming reduces to that of program behavior under uniprogramming.

2.2.1. Program Behavior Model and Paging Algorithms

Consider a program consisting of a set $P=\{p_1,p_2,\ldots,p_n\}$ of n pages. As the program is executed by a processor, it generates a sequence $r_1r_2\ldots r_t\ldots$ of references to pages, where if $r_t=p_i$ we say that the program references page p_i at the t-th reference (or at time t). We assume that such a sequence, known as a page reference string, is generated by a probabilistic law associated with the program. In particular, if r_{t+1} depends only on $r_{t-k+1},\ldots,r_t$, then the probabilistic behavior of the

---

[1] The result described in Section 2.2. can be regarded as an extension of King's work [K1], although this was independently derived. He used a zeroth-order Markov model for exactly the same purpose.

program forms a k-th order Markov chain. This Markov chain actually can be reduced to a first-order Markov chain of n^k states by considering a direct product (P^k) of the set P, i.e., a state augmentation technique [H2]. Therefore, we will consider, without loss of generality, only programs that are modeled by a first-order Markov chain.

The probabilistic behavior of these programs is characterized by a transition matrix $\underline{P}$ of the following form [H2], where an element of the matrix p_{ij} represents a conditonal probability of a reference to a page p_j at time t+1, given that a page p_i was referenced at time t.

$$\underline{P} = \begin{array}{c} t \backslash t+1 \\ p_1 \\ p_2 \\ \vdots \\ p_n \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & \cdots & p_n \end{array} \\ \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{pmatrix} \end{array} \qquad (2.2.1)$$

We assume that the elements of this matrix do not depend on time t, i.e., the program behavior is <u>stationary</u>[1]. Hereafter, we call this type of program a (stationary first-order) <u>Markovian program</u>.

Suppose that primary memory consists of a set $M=\{f_1, f_2, \ldots, f_m\}$ of m page-frames where $m \leq n$. Although the most general paging algorithm may be the one which fetches (replaces) an arbitrary number of pages at an arbitrary instant of time, we will consider the algorithms which

---

[1] Lewis and Yue [L3] have pointed out that program behavior represented in (LRU stack) distance reference strings rather than page reference strings tends to be more stationary.

fetch (replace) only one page at an instant of a page fault, i.e., demand paging algorithms. In this case, the studies of paging algorithms reduce to those of PRAs.

With this much of background, we can formally describe a PRA for M and $\underline{P}$, as follows.

Definition[1] A page replacemant algorithm for M and $\underline{P}$ is a five-tuple system $\Sigma = (S, I, O, f, g)$ where

(1) S is a finite set of system states and is represented by a direct product of finite sets (S_a, S_m, S_p) of PRA control states, memory states, and program states, i.e.,

$$S = S_a \times S_m \times S_p,$$

where

(a) S_a is a finite set of PRA control states, $S_a = \{ s_a \}$, which is defined separately for each PRA later,

(b) S_m is a finite set of memory states, indicating a set of pages resident in primary memory, such that

$$S_m = \{ s_m \mid s_m \subseteq P, \mid s_m \mid \leq m \},$$

(c) S_p is a finite set of program states,

$$S_p = \{ s_p \mid s_p \subseteq P, \mid s_p \mid = 1 \}.$$

such that if $s_p = \{ p_i \}$ then page p_i of the program is being referenced,

---

[1] This definition is similar to that of Aho et al [A1] or King [K1].

(2) I is a finite input alphabet which contains all the elements of page reference strings, i.e.,

$$I = \{p_i \mid p_i \in P\} \equiv P,$$

(3) O is a finite output alphabet

$$O = \{0,1\}$$

whose meaning is defined below,

(4) f is a next state function of the mapping

$$S \times I \rightarrow S \quad \text{such that}^1$$

$$\text{if } f(s_a^t, s_m^t, s_p^t; p_i) = (s_a^{t+1}, s_m^{t+1}, s_p^{t+1})$$

$$\text{then } p_i \in s_m^{t+1} \text{ and } s_p^{t+1} = \{p_i\},$$

(5) g is an output function of the mapping

$$S \times I \rightarrow O \quad \text{such that}$$

$$g(s_a^t, s_m^t, s_p^t; p_i) = \begin{cases} 0 & \text{if } p_i \in s_m^t \\ 1 & \text{otherwise (i.e., page fault).} \end{cases}$$

(6) initial states of S_a, S_m, and S_p are empty sets (φ), i.e.,

$$s_a^0 = s_m^0 = s_p^0 = \varphi.$$

It should be noted that a PRA for M and P is a (deterministic) transition-assigned finite-state automaton (FSA) with states $S_a \times S_m \times S_p$, inputs P, and outputs O, as shown in Figure 2-2. The triple, (s_a, s_m, s_p),

---

[1] The superscript t means "at time t". For example, s_a^t means that the PRA control state at time t is s_a.

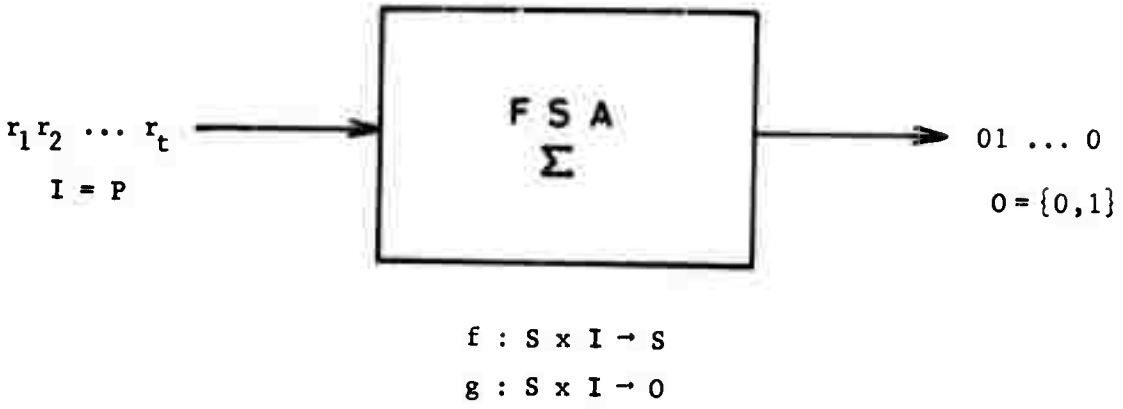


Figure 2-2 FSA Representation of PRA System

represents a state of this FSA and is called a configuration of the PRA system Σ [A1,K1]. As the system Σ receives a page reference string $r_1 r_2 \cdots r_t$, the configuration of this system changes like

$$(s_a^0, s_m^0, s_p^0), (s_a^1, s_m^1, s_p^1), \ldots, (s_a^t, s_m^t, s_p^t), (s_a^{t+1}, s_m^{t+1}, s_p^{t+1})$$

generating an output sequence of 0's and 1's. The output sequence, which indicates when a page fault takes place, depends only on a page reference string, the PRA in use, and primary memory size (m). This implies that once a PRA is chosen for a computer system under study with a fixed amount (m) of primary memory, the probabilistic property of the occurence of "1" in the output sequence, i.e., the occurrence of page faults, is completely determined by the probabilistic property of the corresponding input sequence, i.e., the probabilistic law given by the $\underline{P}$ matrix.

2.2.2. Behavior of the PRA System

We will proceed to determine the PRA control states left unspecified in the previous section and enumerate the number of distinct states that the PRA system Σ can assume for each of various PRAs. Then, noting that the probability associated with the transition from a certain current state to another is completely determined by that pair of states and the $\underline{P}$ matrix, we will see that the probabilistic behavior of the system Σ can be viewed as a Markov chain defined over these states.

First of all, note that for any demand paging PRA

$$f(s_a, s_m, s_p; p_i) = \begin{cases} (s'_a, s_m, \{p_i\}) & \text{if } p_i \in s_m \\ (s'_a, s_m + \{p_i\}, \{p_i\}) & \text{if } p_i \notin s_m,\ |s_m| < m \\ (s'_a, s_m + \{p_i\} - \{p_j\}, \{p_i\}) & \text{if } p_i \notin s_m,\ |s_m| = m,\ p_j \in s_m \end{cases} \qquad (2.2.2)$$

where p_j is the page replaced by page p_i and s'_a is a state which is generally different from s_a. Observing that the PRA is exercised only when $|s_m| = m$ and that the program is assumed to operate indefinitely within primary memory of m page-frames, we will neglect all the system states for which $|s_m| < m$; we will now enumerate the number (N_s) of distinct states that the system Σ can assume, for each of the following several PRAs, assuming that $|s_m| = m$.

Random PRA The page to be replaced is chosen randomly from the pages currently contained in primary memory. Therefore, the PRA itself need not remember any kind of priorities associated with resident pages i.e., the PRA is "memoryless", and N_s is determined by the number of different combinations of only s_p and s_m. That is to say,

$$N_s(\text{Random}) = m \binom{n}{m} \qquad (2.2.3)$$

LRU PRA This well-known algorithm chooses the least-recently-used page (i.e., the page with the longest time since the last reference) and removes it from primary memory. This algorithm must maintain an ordered list (called a LRU stack [M1]) of resident pages to remember how recently each resident page was referenced in the past. Noting that there exist m! different ordered lists and that each ordered list

automatically determines s_p, N_s is obtained as

$$N_s(\text{LRU}) = m!\binom{n}{m} \tag{2.2.4}$$

FIFO PRA This easy-to-implement algorithm chooses the page which stayed in primary memory for the longest time for removal. This algorithm must maintain an ordered list of resident pages to remember in which order each resident page was fetched into primary memory. Noting that there exist m! different ordered lists and that s_p cannot be implied by each of those lists in any way, N_s is obtained as

$$N_s(\text{FIFO}) = m!\, m\binom{n}{m} \tag{2.2.5}$$

LET PRA This algorithm removes the page with the longest expected time (LET) until next reference [M1,A1,D2]. Because the expected time until next reference to each page can be analytically calculated only from s_p and the $\underline{P}$ matrix (as will be shown below), the PRA itself is memoryless. Therefore, N_s is obtained as

$$N_s(\text{LET}) = m\binom{n}{m} \tag{2.2.6}$$

Another similar PRA removes the page with the longest expected time since the last reference. Because this PRA uses information about the backward distance (into the past) from the faulted page to each resident page, we call this PRA the backward LET PRA (B-LET). N_s is given by Eq.(2.2.6). We call the former LET PRA the forward LET PRA (F-LET). It should be noted that the LRU PRA, an approximation to the B-LET PRA, is often used on an actual computer system in the hope that the B-LET PRA is fairly close to the F-LET PRA for typical (actual) programs.

Now we are ready to consider a Markov chain model of the behavior of the system Σ for each of these PRAs. For this purpose, we borrow an example of a Markovian program which was used by Chang [C1] in proving that the F-LET PRA does not necessarily minimize the number of page faults for a general Markovian program. Chang's example[1] is reproduced in the form of a $\underline{P}$ matrix.

$$\underline{P} = \begin{array}{c} t \backslash t+1 \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 0 & c & 1-c \\ d & 0 & 1-d \\ 0 & 1 & 0 \end{pmatrix} \end{array} \qquad (2.2.7)$$

We assume that our primary memory has only two page-frames, i.e., $M = \{f_1, f_2\}$.

Random PRA Let $\{\textcircled{i}, j\}$ denote a state of Σ where $s_m = \{p_i, p_j\}$ and $s_p = \{p_i\}$. Then, there exist the following six states in S, for this PRA.

$$S = \{\{\textcircled{1}, 2\}, \{1, \textcircled{2}\}, \{\textcircled{2}, 3\}, \{2, \textcircled{3}\}, \{\textcircled{1}, 3\}, \{1, \textcircled{3}\}\}$$

Noting the $\underline{P}$ matrix of Eq. (2.2.7) and the random nature of this PRA, we obtain the transition matrix $\underline{Q}$ specifying the probabilities of transitions among these states.

---

[1] This example happens to have zero diagonal elements in the $\underline{P}$ matrix, but probably most actual programs tend to have fairly large diagonal elements corresponding to locality of memory references.

$$\underline{Q} = \begin{array}{c} \\ \{①,2\} \\ \{1,②\} \\ \{②,3\} \\ \{2,③\} \\ \{①,3\} \\ \{1,③\} \end{array} \begin{array}{c} \begin{array}{cccccc} \{①,2\} & \{1,②\} & \{②,3\} & \{2,③\} & \{①,3\} & \{1,③\} \end{array} \\ \begin{pmatrix} 0 & c & 0 & \frac{1-c}{2} & 0 & \frac{1-c}{2} \\ d & 0 & 0 & \frac{1-d}{2} & 0 & \frac{1-d}{2} \\ \frac{d}{2} & 0 & 0 & 1-d & \frac{d}{2} & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \frac{c}{2} & \frac{c}{2} & 0 & 0 & 1-c \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \end{pmatrix} \end{array}$$

LRU PRA Let $\{ⓘ, j\}$ similarly denote a state of Σ where $s_m = \{p_i, p_j\}$ and $s_p = \{p_i\}$; page p_j is least recently used. Then, there exist the same six states in S as above, and the $\underline{Q}$ matrix is obtained as

$$\underline{Q} = \begin{array}{c} \\ \{①,2\} \\ \{1,②\} \\ \{②,3\} \\ \{2,③\} \\ \{①,3\} \\ \{1,③\} \end{array} \begin{array}{c} \begin{array}{cccccc} \{①,2\} & \{1,②\} & \{②,3\} & \{2,③\} & \{①,3\} & \{1,③\} \end{array} \\ \begin{pmatrix} 0 & c & 0 & 0 & 0 & 1-c \\ d & 0 & 0 & 1-d & 0 & 0 \\ d & 0 & 0 & 1-d & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & c & 0 & 0 & 0 & 1-c \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

FIFO PRA Let $\{ⓘ, ⓙ\}$ denote a state of Σ where $s_m = \{p_i, p_j\}$, $s_p = \{p_i\}$, and page p_j stayed in primary memory for the longest time. Then, there exist the following twelve states:

$$S = \begin{cases} \{0, 2\}, \{0, 2\}, \{1, 2\}, \{1, 2\}, \{2, 3\}, \{2, 3\} \\ \{2, 0\}, \{2, 0\}, \{0, 3\}, \{1, 3\}, \{1, 3\}, \{1, 0\} \end{cases}$$

The state transitions are specified by the following $\underline{Q}$ matrix.

	{0,2}	{0,2}	{1,2}	{1,2}	{2,3}	{2,3}	{2,0}	{0,3}	{1,3}	{1,0}	{1,0}
{0,2}	0	0	c	0	0	0	1-c	0	0	0	0	0
{0,2}	0	0	0	c	0	0	0	0	0	0	1-c	0
{1,2}	d	0	0	0	0	0	1-d	0	0	0	0	0
{1,2}	0	d	0	0	0	0	0	0	0	0	1-d	0
{2,3}	0	0	0	0	0	0	1-d	0	0	d	0	0
{2,3}	0	d	0	0	0	0	0	1-d	0	0	0	0
{2,0}	0	0	0	0	1	0	0	0	0	0	0	0
{2,0}	0	0	0	0	0	1	0	0	0	0	0	0
{0,3}	0	0	0	0	0	c	0	0	0	0	1-c	0
{0,3}	0	0	c	0	0	0	0	0	0	0	0	1-c
{1,0}	0	0	0	0	0	1	0	0	0	0	0	0
{1,0}	0	0	1	0	0	0	0	0	0	0	0	0

LET PRA First of all, we must derive the expected time, $E(t_{ij})$, to elapse until the first reference to a page p_j starting from a page p_i $(1 \leq i, j \leq n)$ in order to establish the decision rule of this PRA. Let $\underline{P}_j$ be the matrix obtained from $\underline{P}$ by setting $p_{jk} = \delta_{jk}$ (Kronecker's delta) for all k, i.e., making page p_j an artificial trapping state

[H2], as shown below.

$$\underline{P}_j = \begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_j \\ \vdots \\ p_n \end{array} \begin{array}{c} \begin{array}{cccccc} p_1 & p_2 & \cdots & p_j & \cdots & p_n \end{array} \\ \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1j} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2j} & \cdots & p_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nj} & \cdots & p_{nn} \end{pmatrix} \end{array} \qquad (2.2.8)$$

Observing that the j-th column of $(\underline{P}_j)^k$ gives the probabilities that page p_j is referenced at the k-th reference starting from each page [H2], we see that the probabilities that page p_j is referenced for the first time at the k-th reference starting from each page is given by the j-th column of the difference matrix, $(\underline{P}_j)^k - (\underline{P}_j)^{k-1}$. (Note that page p_j is the artificial trapping state.) Therefore,

$$\sum_{k=1}^{\infty} k\{(\underline{P}_j)^k - (\underline{P}_j)^{k-1}\} = \begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_n \end{array} \begin{array}{c} \begin{array}{ccccc} p_1 & \cdots & p_j & \cdots & p_n \end{array} \\ \begin{pmatrix} \cdots & E(t_{1j}) & \cdots \\ \cdots & E(t_{2j}) & \cdots \\ & \vdots & \\ \cdots & E(t_{nj}) & \cdots \end{pmatrix} \end{array} \qquad (2.2.9)$$

where $(\underline{P}_j)^0 \equiv (0)$.

Repeating a simlilar calculation for each j ($1 \le j \le n$), we can obtain the matrix $\underline{E}(t)$ which contains $E(t_{ij})$ as its i-j element.

$$\underline{E}(t) = \begin{pmatrix} 1 & E(t_{12}) & \cdots & E(t_{1n}) \\ E(t_{21}) & 1 & \cdots & E(t_{2n}) \\ \vdots & \vdots & & \vdots \\ E(t_{n1}) & E(t_{n2}) & \cdots & 1 \end{pmatrix} \qquad (2.2.10)$$

We call $\underline{E}(t)$ the <u>distance matrix</u> and $E(t_{ij})$ the distance from a page p_i to a page p_j. It should be noted that this matrix is not necessarily symmetric.

Now we can describe the LET PRAs more clearly using the concept of distances. Suppose that a page p_i was found to be missing in primary memory when $|s_m| = m$ in the course of program execution. Then,

$$f(s_a, s_m, s_p; p_i) = (s'_a, s_m + \{p_i\} - \{p_j\}, \{p_i\})$$

$$\text{where} \begin{cases} E(t_{ij}) = \max_{p_k \in s_m} E(t_{ik}) & \text{if F-LET is used,} \\ E(t_{ji}) = \max_{p_k \in s_m} E(t_{ki}) & \text{if B-LET is used,} \\ s_a = s'_a = \varphi & \text{(memoryless algorithm).} \end{cases} \qquad (2.2.11)$$

This means that the F-LET and the B-LET PRAs coincide if the ordering of row elements of the distance matrix, with respect to the magnitude of their values, is the same as that of the corresponding column elements (or more strongly, if the distance matrix is symmetric). For this reason, the LRU PRA presumably performs well if the distance matrix of real programs tends to be symmetric.

For the $\underline{P}$ matrix of Eq. (2.2.7), with $c = 0.1$ and $d = 0.5$, we obtain the following distance matrix.

$$\underline{E}(t) = \begin{pmatrix} 1 & 1.9 & 1.16 \\ 3 & 1 & 1.58 \\ 4 & 1 & 1 \end{pmatrix}$$

Therefore, as is clear from Eq. (2.2.11), the decision rule of the F-LET PRA can be stated as follows:

s_m^t	r_t page to be removed		s_m^{t+1}
$\{p_1, p_2\}$	p_3	p_1	$\{p_2, p_3\}$
$\{p_2, p_3\}$	p_1	p_2	$\{p_1, p_3\}$
$\{p_1, p_3\}$	p_2	p_1	$\{p_2, p_3\}$

Similarly, the decision rule of the B-LET PRA can be stated as follows:

s_m^t	r_t page to be removed		s_m^{t+1}
$\{p_1, p_2\}$	p_3	p_2	$\{p_1, p_3\}$
$\{p_2, p_3\}$	p_1	p_3	$\{p_1, p_2\}$
$\{p_1, p_3\}$	p_2	p_1	$\{p_2, p_3\}$

Now, let $\{i, j\}$ denote a state of Σ where $s_m = \{p_i, p_j\}$ and $s_p = \{p_i\}$. Then, for each of LET PRAs, we have the same set of six states as obtained for the Random PRA. The transition matrix for the F-LET PRA is given by

$$\underline{Q} = \begin{array}{c} \\ \{1,2\} \\ \{1,2\} \\ \{2,3\} \\ \{2,3\} \\ \{1,3\} \\ \{1,3\} \end{array} \begin{array}{c} \begin{array}{cccccc} \{1,2\} & \{1,2\} & \{2,3\} & \{2,3\} & \{1,3\} & \{1,3\} \end{array} \\ \begin{pmatrix} 0 & c & 0 & 1-c & 0 & 0 \\ d & 0 & 0 & 1-d & 0 & 0 \\ 0 & 0 & 0 & 1-d & d & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 & 1-c \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \begin{array}{l} c=0.1 \\ d=0.5 \end{array}$$

The corresponding matrix for the B-LET PRA is given by

$$\underline{Q} = \begin{array}{c} \\ \{1,2\} \\ \{1,2\} \\ \{2,3\} \\ \{2,3\} \\ \{1,3\} \\ \{1,3\} \end{array} \begin{array}{c} \begin{array}{cccccc} \{1,2\} & \{1,2\} & \{2,3\} & \{2,3\} & \{1,3\} & \{1,3\} \end{array} \\ \begin{pmatrix} 0 & c & 0 & 0 & 0 & 1-c \\ d & 0 & 0 & 0 & 0 & 1-d \\ d & 0 & 0 & 1-d & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 & 1-c \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \begin{array}{l} c=0.1 \\ d=0.5 \end{array}$$

Thus, we have obtained the transition matrix $\underline{Q}$ which characterizes the Markovian behavior of the PRA system Σ for each PRA.

2.2.3. Evaluation of Paging Behavior of Markovian Programs

Now that we have enumerated the number of system states, each of which satisfies the Markovian property, and have shown how the state transitions can be determined when a PRA is chosen for a given program, the next natural step is to evaluate the program behavior in this environment. Therefore, this section is concerned with how often a page fault occurs as a given Markovian program is executed in a given amount of primary memory under the dynamic memory managemant of a given paging algorithm.

We will use the missing-page probability p defined below to measure the frequency of page faults.

$$p = \lim_{t \to \infty} \frac{\text{expected number of page faults observed during the period } [0,t]}{t} \qquad (2.2.12)$$

It should be noted that the missing-page probability is the inverse of mhbpf, as is clear from this definition.

As a Markovian program is executed under a given PRA, the PRA system Σ operates over a set of N_s Markovian system states making two kinds of state transitions, i.e., those each accompanied by a page fault and those not accompanied by a page fault. We call them page-fault transitions and non-page-fault transitions respectively. Then, the missing-page probability can be expressed as

$$P = \lim_{t\to\infty} \text{Prob}\{\text{occurrence of a page-fault transition at time } t\}$$

$$= \sum_{k=1}^{N_s} \left\{ \pi(k) \left(\begin{array}{l} \text{sum of the all conditional probabilities} \\ \text{of page-fault transitions from state k,} \\ \text{given that the system is now in state k} \end{array} \right) \right\}$$

(2.2.13)

where $\pi(k)$ is the steady-state probability of the k-th state of the system Σ. The value of $\pi(k)$ $(k=1,2,\ldots,N_s)$ can be determined by solving the balance equations [H2]

$$\underline{\pi} = \underline{\pi}\,\underline{Q} \qquad (2.2.14)$$

where $\underline{\pi}$ is a row vector whose elements are $\pi(k)$ $(k=1,2,\ldots,N_s)$. Thus, we have presented a method to evaluate the missing-page probability (or mhbpf) of a given Markovian program operating in a given amount of primary memory under the dynamic memory management of a given demand paging algorithm.

Next, we will give some numerical results using the example of the previous section. Let us first evaluate the performance of the three page program, operating in the two page-frame primary memory under the LRU PRA, that we discussed in Section 2.2.2. Solving Eq. (2.2.14) with the $\underline{Q}$ matrix ($c = 0.1$, $d = 0.5$) obtained before, we get

$$\underline{\pi} = (\pi(1,2), \pi(1,2), \pi(2,3), \pi(2,3), \pi(1,3), \pi(1,3))$$
$$= (0.204,\ 0.020,\ 0.388,\ 0.204,\ 0\ \ ,\ 0.184\ )$$

Then, we can calculate the missing-page probability, using Eq. (2.2.13), as follows:

$$p = 0.204\,(1-c) + 0.020\,(1-d) + 0.388\,p + 0.184$$
$$= 0.184 + 0.010 + 0.194 + 0.184$$
$$= 0.572$$

The missing-page probabilities were similarly evaluated for other PRAs considered in the previous sections. The result is summarized in Table 2-1. It should be pointed out that the result of Table 2-1 depends heavily on a particular choice of the $\underline{P}$ matrix. However, it is reasonable to expect that the B-LET and the LRU PRAs, which use the "backward" information about the program behavior, generally do not perform well for a Markovian program whose $\underline{P}$ matrix is rather asymmetric; in fact, their performance shown in Table 2-1 is significantly worse than that of others. Similarly, the FIFO PRA does not perform well. On the other hand, the F-LET PRA, which uses the "forward" information about the program behavior, outperformed other PRAs. It is however known that the F-LET PRA does not necessarily minimize the missing-page probability of a first-order Markovian program. Chang [C1] actually gave a better PRA, for this Markovian program, which has the same decision rule as that of the F-LET PRA described in Section 2.2.2. except that if a page fault occurs when $s_m = \{p_2, p_3\}$ then page p_3 is removed. For this PRA, the missing-page probability is found to be 0.388 by the above evaluation method.

Thus, it has been shown that the paging behavior of any Markovian program can be analytically evaluated as a function of (1) program characteristics, (2) the paging algorithm, and (3) primary memory size. However, the method which has been described in Section 2.2. has a

Table 2-1 Comparison of PRA Performances

PRA	Missing-Page Probability
Random	0.475
LRU	0.572
FIFO	0.565
F-LET	0.408
B-LET	0.673

serious drawback concerning the amount of required computation. To examine this problem, let us assume that the computer system uses a memoryless PRA (such as the Random or the LET PRA). The number of system states was found to be

$$N_s = m \binom{n}{m} = \frac{n!}{(m-1)!\,(n-m)!}$$

and we must solve the simultaneous equations in N_s variables, given by Eq. (2.2.14), in evaluating the missing-page probability. Although sparsity of non-zero elements in a $\underline{Q}$ matrix may help, solving such simultaneous equations in, say, three hundred variables requires much computation time. For example, if we have n=10 and m=8, then we get N_s=360. If we have a pair of more reasonable numbers, like n=50 and m=40, for a typical situation of the Multics system, then N_s becomes a tremendously large number. Therefore, if we want to study the behavior of more realistic programs, it seems that we must somehow simplify our model.

King [K1] used a simplified model, for which $p_{ij} = \beta_j$ for all i in the $\underline{P}$ matrix, in specifying a program's reference pattern. This means that the page to be referenced at time t+1 no longer depends on the page being referenced at time t. Therefore, the program behavior is completely memoryless and the third component of a state of the PRA system Σ , s_p, can be reduced to an empty set, i.e., $s_p = \varphi$. This type of program behavior model is called a <u>zeroth-order Markovian program</u> (or a multinomial process model). The number of system states N_s decreases in this simplified model by a factor of m, but the evaluation of this method still requires a large amount of computation.

2.3. Zeroth-order Markov Model Applied to the Page Size Problem

In Section 2.3, we will consider the effect of another system parameter, i.e., the page size, upon the paging behavior of programs. Because the page size must be changed to examine its effect, we must be concerned with a program's reference pattern not only over its pages but within each of those pages. We must be concerned also with how a compiler forms a program's pages by combining smaller pieces of that program, i.e., the pagination process of the compiler. The effect of page size upon a program's paging behavior has been experimentally studied by Hatfield et al [H1], Informatics, Inc. [I1], and Baer et al [B1]. This aspect of the page size problem , i.e., this aspect of the problem of determining the optimum page size, has rarely been studied in an analytical approach. So far as known to the author of this thesis, only Woolf [W3] has examined the effect of page size upon the frequency of paging analytically. The approach to be taken in this section is completely different from that of Woolf because of the way in which programs are modeled. Other aspects of the page size problem will be discussed later in Chapter 5.

2.3.1. Program Behavior Model for the Page Size Studies

A program usually possesses a fundamental property known as locality of information references [D1-3], i.e., a program's tendency to reference a subspace of its address space during any interval of its execution. We call a set of pages in such a favored subspace of its address space

a set of favored pages or a favored set[1] in this section. Control of a program usually stays in a favored set for some time and then enters another favored set. Therefore, we can evaluate the program behavior by analyzing the behavior of each successive favored set using a Markov model described in Section 2.2.

If we take a close look at each page of a favored set, we find that there are portions which are referenced frequently (e.g., loops) and protions which are not (e.g., program code following a loop, rarely used data, etc.) We roughly call each of these logically identifiable portions a program block. Thus, each page of a favored set is composed of (several) program blocks combined together by a compiler.

Now we are concerned with a program's reference pattern on the level of program blocks. In view of the computational drawback of first-order Markovian programs discussed in Section 2.2.3, we use a zeroth-order Markovian program model for the behavior of a favored set. Let b_j (j=1,2,...) and $p(b_j)$ be the j-th program block and its probability of reference in the favored set respectively. For the sake of simplicity, we assume that all the program blocks have the same size, s(b), and that only one page size, s(p), is used on the computer system. Furthermore, we assume that the block size s(b) divides the page size s(p). Denoting the i-th page of the favored set and its probability

---

[1] Roughly speaking, a "favored set" corresponds to a program's module being referenced by a process. This loosely defined concept should be distinguished from the more strictly defined "working set" of Denning [D1,D3].

of reference by $p_i (i=1,2,\ldots,n)$ and $p(p_i)$ respectively, we have

$$p(p_i) = \sum_{b_j \in p_i} p(b_j) \qquad (2.3.1)$$

$$\sum_{i=1}^{n} p(p_i) = \sum_{i=1}^{n} \sum_{b_j \in p_i} p(b_j) = 1 \qquad (2.3.2)$$

Because these program blocks generally have different probabilities of reference, the program pages of the favored set have different probabilities of reference. One of the very interesting problems in this area of study is that of determing a set of program blocks to be contained in each page of the favored set so as to minimize the expected number of page faults during its execution. This problem is called the pagination problem. The paging behavior of program is known to depend heavily on the quality of pagination [H1].

2.3.2. Pagination, Page Size, and Missing-Page Probability

We proceed to evaluate the performance of programs under paging in terms of missing-page probability, for different paginations and different page sizes. It is known that the F-LET PRA is optimum for zeroth-order Markovian programs in the sense that it minimizes the expected number of page faults [A1]. Therefore, we assume that this PRA is in use for the computer system under study.

We now consider three different paginations for the program (the favored set) described in Section 2.3.1, i.e., a zeroth-order Markovian program. Without loss of generality, we can assume that b_j and p_i are

both numbered in the order of monotonically decreasing probability; that is to say,

$$p(b_1) \geq p(b_2) \geq p(b_3) \geq \ldots \geq p(b_k) \qquad (2.3.3)$$

$$p(p_1) \geq p(p_2) \geq p(p_3) \geq \ldots \geq p(p_n) \qquad (2.3.4)$$

Best Pagination The best pagination must have the property that the missing-page probability p is minimum for any size m ($0 \leq m \leq n$ pages) of primary memory available to the program. Noting that the F-LET PRA retains m-1 most-frequently-used pages and the page which caused a page fault for a zeroth-order Markovian program, we get

$$p \simeq 1 - \sum_{i=1}^{m} p(p_i) \qquad (2.3.5)$$

Let r be the number of program blocks that can be contained in a page, i.e., $r = s(p)/s(b)$. Then, the best pagination is the one for which the most-frequently-used page p_1 contains the r most-frequently-used program blocks $b_1, b_2, \ldots, b_r$, the second most-frequently-used page p_2 contains the second r most-frequently-used program blocks $b_{r+1}, b_{r+2}, \ldots, b_{2r}$, and so on. In other words,

$$p_i = \{ b_{(i-1)r+1}, b_{(i-1)r+2}, \ldots, b_{ir} \} \quad 1 \leq i \leq n \qquad (2.3.6)$$

It is easy to prove that this pagination has the property that the missing-page probability is minimum for any m, under the F-LET PRA. Furthermore, page references are best localized by this pagination.

Worst Pagination It seems that any one pagination generally cannot guarantee that the missing-page probability given by Eq. (2.3.5) is maximum for every size m of primary memory. To see this, assume that

there exists a worst pagination with page reference probabilities $p^*(p_i)$ $(i=1,2,\ldots,n)$. Then, for an arbitrarily chosen pagination with page reference probabilities $p(p_i)$ $(i=1,2,\ldots,n)$, the following inequalities must be simultaneously satisfied;

$$\left.\begin{array}{ll} p^*(p_1) \le p(p_1) & \text{for } m=1 \\ p^*(p_1) + p^*(p_2) \le p(p_1) + p(p_2) & \text{for } m=2 \\ \vdots & \vdots \\ \sum_{i=1}^{k} p^*(p_i) \le \sum_{i=1}^{k} p(p_i) & \text{for } m=k \\ \vdots & \vdots \\ \sum_{i=1}^{n} p^*(p_i) \le \sum_{i=1}^{n} p(p_i) & \text{for } m=n \end{array}\right\} \qquad (2.3.7)$$

where $p^*(p_i)$ and $p(p_i)$ are both numbered in the non-increasing order, as given by Eq. (2.3.4). The first inequality of Eq. (2.3.7) means that the largest page reference probability is minimized by the worst pagination. There generally exists just one pagination which satisfies this inequality, but this pagination does not guarantee the rest of the inequalities of Eq. (2.3.7), in general. Therefore, we must give up searching for the "universal" worst pagination that is independent of m. Instead, we say that a given pagination is near-worst if the resulting page reference pattern is reasonably close to a uniform distribution. In fact, it can be shown that as $r = s(p)/s(b)$ approaches infinity the missing-page probability of such a near-worst pagination for a given size (m) of primary memory becomes

$$p = 1 - \sum_{i=1}^{m} p(p_i) = \frac{n-m}{n} = 1 - \frac{m}{n} \qquad (2.3.8)$$

The following simple algorithm seems to be a practical one to derive a

near-worst pagination.

Algorithm to derive a near-worst pagination Assume that the reference probabilities of program blocks of a given program (a favored set) are arranged in the non-increasing order and that we want to derive a near-worst pagination for page size 2 s(b). The derivation algorithm consists of successive mergers of program blocks; merge the most- and the least-frequently-used blocks (b_1 and b_{nr}), merge the second most- and the second least-frequently-used blocks (b_2 and b_{nr-1}), merge the third most- and the third least-frequently-used blocks, and so on, in order to form the pages which tend to have the same magnitude of reference probabilities. If the number (nr) of the program blocks is not even, an appropriate modification is necessary (e.g., adding a dummy block). If we want a near-worst pagination for page size 4 s(b), then arrange the above newly created pages of size 2 s(b) in the order of non-increasing probability and repeat the above pair-wise successive mergers.

Random Pagination The program blocks are randomly selected and merged into pages. The missing-page probability of a random pagination is bounded by those of the best and the worst paginations. Perhaps, a random pagination is fairly close to a casual pagination found in most actual user programs. The analytical evaluation of the missing-page probability of this pagination for a given program is not as easy as it seems, but the performance of this pagination could be easily evaluated by a similation. (We will not consider this pagination any further in this thesis.)

Next, we present an illustrative example of the program behavior under paging, to show

- the effect of pagination (due to a compiler's code optimization, loading order of program modules, programming style, etc.)
- the effect of page size

upon the missing-page probability (the inverse of mhbpf) as a function of the size of available primary memory.

Example We must specify the reference pattern of a program to be studied. Let us assume that s(b) is 256 words and that the distribution of $p(b_j)$ is given by Figure 2-3. Then, we can numerically evaluate the lower and the upper bounds of the missing-page probability of an arbitrary pagination for a given page size, by examining the performance of the best and the worst paginations.

Let us consider the case of s(p)=512 words, first of all. Then, we obtain, for the best pagination,

$$p(p_1)=1/16+1/16,\ p(p_2)=1/16+1/16,\ p(p_3)=1/32+1/32$$
$$\cdots\cdots\ ,\ p(p_{30})=1/128+1/128$$

and for a near-worst pagination,

$$p(p_1)=1/16+1/128,\ p(p_2)=1/16+1/128,\ p(p_3)=1/16+1/128$$
$$\cdots\cdots\ ,\ p(p_{30})=1/128+1/128$$

Noting that memory management is under the demand paging F-LET PRA, we obtain the result shown in Figure 2-4. It is seen that the region bounded

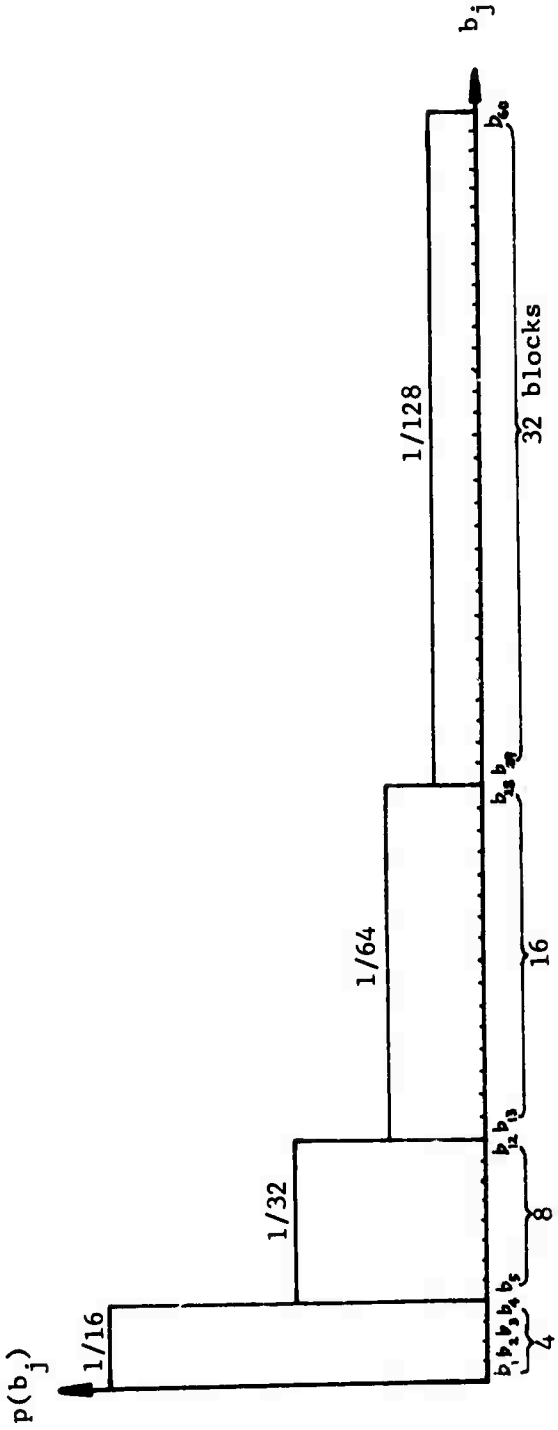


Figure 2-3 Distribution of $p(b_j)$

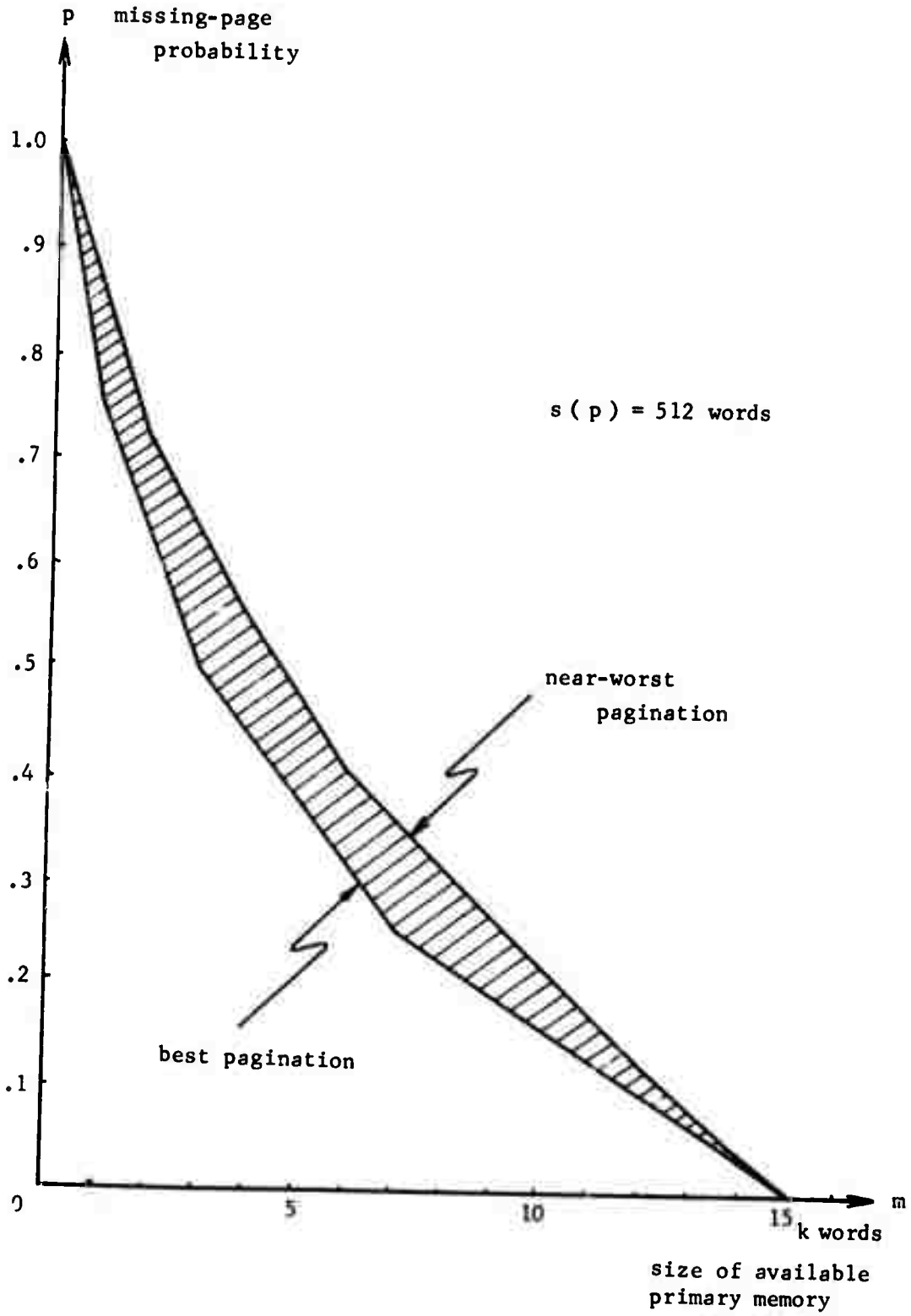


Figure 2-4 Effect of Pagination on Missing-Page Probability

by the two curves[1], which represents the variation of program performance due to pagination, is rather narrow for this choice of page size, i.e., s(p)=512 words. Similarly, the bounding curves were derived for each choice of page size (256, 512, 1024, and 2048 words) using the algorithms described above, in order to examine the effect of page size upon the missing-page probability p. This result is shown in Figure 2-5. It is clear that the bounded region becomes wider, having a common lower bound curve, as the page size increases. It should be noted that the upper bound curve for s(p)=2048 words is very close to the straight line given by Eq. (2.3.8). In summary, the following observations may be made:

(1) If a given program is optimally paginated (like the best pagination), a choice of page size does not affect the missing-page probability (or mhbpf).

(2) The choice of smaller page size tends to attain the smaller missing-page probability (or the longer mhbpf) if a given program is casually paginated.

We assumed in the above analysis that program blocks are relocatable within a program's address space. However, it is apparent that a program will behave as poorly as the one with a near-worst pagination

---

[1]The bound given by a near-worst pagination is not exactly an upper bound because it is not the worst possible pagination for each memory size m, but it is believed to be fairly close to the tightest upper bound. For this reason, it is treated as if it is the exact upper bound.

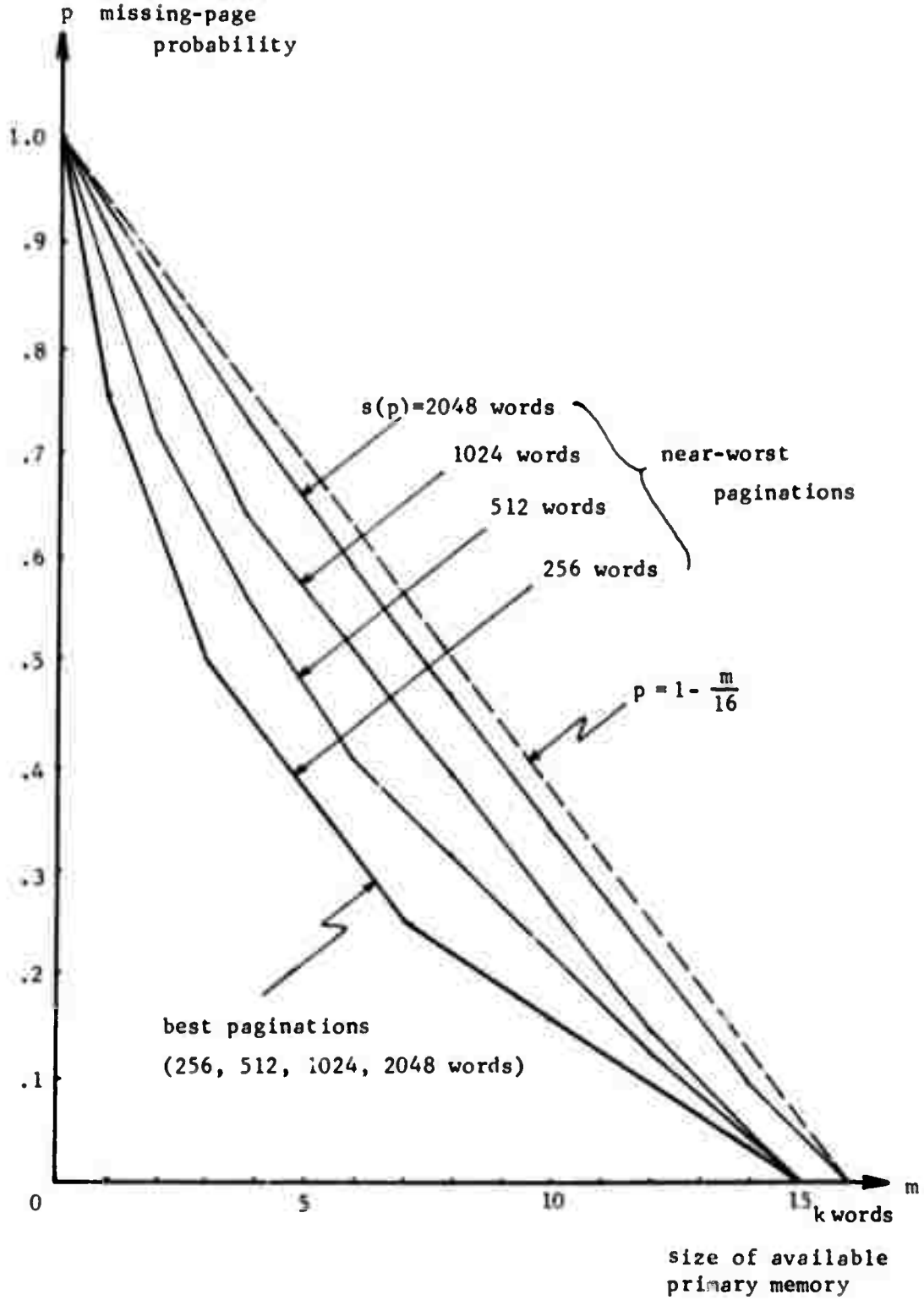


Figure 2-5 Effect of Page Size on Missing-Page Probability

even if the program blocks are not relocatable, if that program happens to possess such a bad loading order of machine instructions as the one generated by the near-worst pagination. Therefore, program behavior like that seen in this section should be universally found on any virtual memory computer system.

Finally, some comments may be in order about the drawbacks of the model which has been used in Section 2.3. First of all, it should be remembered that the model is based on the stationary zeroth-order Markov property of program behavior. This is perhaps the least accurate assumption of this model. Second, the model assumes a single program block size which divides the page size. In order to be more realistic, we must consider the probability distribution of block size and page-boundary crossover of program blocks. Third, we have not derived the exact performance of actual (casual) paginations. The lower bound of the missing-page probability given by the best pagination may be loose if a non-optimal PRA (like the LRU PRA) is in use. However, in spite of these drawbacks of this simplified model, it has successfully demonstrated the basic relationship between pagination, page size, and the steady-state paging behavior of programs. While the smaller page size was found to yield a longer mhbpf in the steady-state program behavior, it is clear that the larger page size yields a longer mhbpf in its transient-state behavior (e.g., when the primary memory is not yet filled with pages). This aspect will be further considered in the next section and later in Chapter 5 when the page size problem will be discussed from an overall viewpoint.

2.4. Random Behavior Model Applied to the Memory Size Problem

In Section 2.4., we will consider a simplified zeroth-order Markovian program, called a random behavior model, to study primarily the effect of primary memory size upon its paging behavior. We will consider both the transient-state program behavior, i.e., the initial stage of program execution generating a number of page faults, and the steady-state program behavior, i.e., the following stage of program execution where the rate of page faults is smaller because available primary memory space is fully utilized. It has been experimentally observed [B5,H1,T2] that paging behavior of programs is more sensitive to the available primary memory size than to any other parameters (e.g., a page replacement algorithm), while primary memory is usually the most expensive element of a computer system. Therefore, an evaluation of program behavior within a limited amount of primary memory is very important from a cost-performance viewpoint. We will express the mhbpf of a random behavior model of programs, as a function of program size, primary memory size, and a parameter which determines the total length of program execution time. It will be clearly seen how thrashing (i.e., excessive competition for primary memory page-frames leading to a less than optimal use of system resources) starts to occur as the primary memory size is gradually reduced.

2.4.1. Random Behavior Model

We will again assume partitioned primary memory under the management of a demand-paging local page replacement algorithm, as in Sections 2.2 and 2.3. A general zeroth-order Markovian program, used in Section 2.3,

is further simplified by requiring that all pages (except a special page) be equally probable in its page reference pattern.

This program behavior model is assumed to have, besides ordinary pages, two special pages, i.e., a starting page which contains an entry point of the program and an ending page which contains an exit point of the program; it is assumed that program execution begins from the starting page p_1 and ends immediately upon entrance to the ending page p_n where n is the size of the program under study. Transfer of processor control among n pages may occur only at the multiples of a unit time t_0, i.e., $t = kt_0$ where $k = 1,2,3,\ldots$. It is furthermore assumed that the ending page can be referenced with a constant probability e at any stage of program execution. Therefore, the page reference pattern of this program behavior model, which we call a <u>random behavior model</u> or a <u>random program</u> hereafter, is characterized by the following transition matrix.

$$\underline{P} = \begin{array}{c} \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \\ p_n \end{array} \begin{array}{c} \begin{array}{cccccc} p_1 & p_2 & \cdots & p_{n-1} & p_n \end{array} \\ \begin{pmatrix} \frac{1-e}{n-1} & \frac{1-e}{n-1} & \cdots & \frac{1-e}{n-1} & e \\ \frac{1-e}{n-1} & \frac{1-e}{n-1} & \cdots & \frac{1-e}{n-1} & e \\ \vdots & \vdots & & \vdots & \vdots \\ \frac{1-e}{n-1} & \frac{1-e}{n-1} & \cdots & \frac{1-e}{n-1} & e \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \end{array} \qquad (2.4.1)$$

In other words,

$$
\begin{aligned}
p_{ij} &= p(p_j) = \frac{1-e}{n-1} && i \neq n,\ j \neq n \\
p_{in} &= p(p_n) = e && i \neq n \\
p_{nj} &= 0 && j \neq n \\
p_{nn} &= 1
\end{aligned}
\qquad (2.4.2)
$$

Thus, a random behavior model has three independent parameters, i.e., program size (n), the reference probability (e) of the ending page, and the unit time (t_0) of this discrete-time Markovian model. The program being modeled references the starting page, then some number of pages at random, and finally the ending page. The number of pages referenced during the execution of this program is a random variable. This may be interpreted in such a way that the number of pages to be referenced during a program execution depends on the arguments given to that program, but a random program represents nevertheless a very special class of programs. The essense of this artificial program consists in its capability to yield some insights into the memory size problem, i.e., the problem of evaluating the effect of primary memory size upon the dynamic paging behavior of programs. Because of the uniformness of its page reference characteristics, the paging performance of a random program does not depend on a particular demand-paging page replacement algorithm in use.

2.4.2. Evaluation of Paging Behavior of Random Programs

We will analytically evaluate the mhbpf of a random program

operating in a limited amount of primary memory under demand paging, as an explicit function of the size (m) of primary memory and the three independent parameters (n,e,and t_0) of a random program, in this section.

First of all, we should briefly consider mhbpf because there seem to exist at least two reasonable definitions of mhbpf for our problem. Suppose that we make K experiments to run a random program. Let $T_e(T_{ei})$ and $n_p(n_{pi})$ be the execution time of the random program and the number of page faults generated by this program, in the (i-th) experiment. Then, the following two definitions of mhbpf seem to be reasonable:

$$mhbpf_1 = \sum_{i=1}^{K} \left(\frac{T_{ei}}{n_{pi}}\right)/K = E\left(\frac{T_e}{n_p}\right)$$

$$mhbpf_2 = \frac{\sum_{i=1}^{K} T_{ei}}{\sum_{i=1}^{K} n_{pi}} = \frac{E(T_e)}{E(n_p)} \qquad (2.4.3)$$

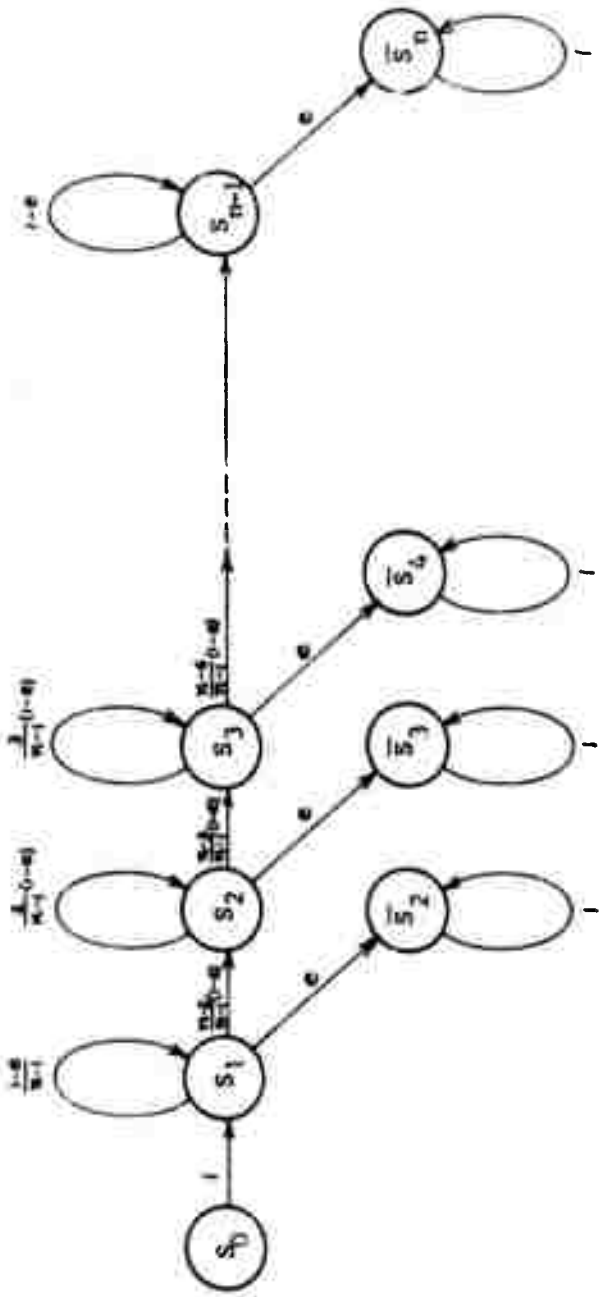
In deriving mhbpf, all experiments are equally weighted in the former definition, while all page faults are equally weighted in the latter definition. Because we usually use the latter definition in calculating mhbpf from on-line monitoring results of an actual computer system, we use the latter definition of mhbpf in evaluating the paging performance of a random program in this section.

Now we proceed to derive the mhbpf of a random program, characterized by n, e, and t_0, operating in primary memory of m page-frames. Let S_i and $\bar{S}_i$ (i=0,1,2,...) respectively denote a state of the random program where it is still running after causing exactly i page faults and a state

of the random program where it has referenced the ending page after causing exactly i page faults. There exist only three possible state transitions from S_i, i.e., a transition to itself, a transition to S_{i+1}, and a transiton to $\overline{S}_{i+1}$. Now we want to derive the probabilities associated with these three state transitions, i.e., $p_{i,i}$, $p_{i,i+1}$, and $\overline{p}_{i,i+1}$. Because each page of the program is assumed to be equally probable in a page reference string at any stage of the program execution, all that matters is the number of pages which exist in primary memory. If we assume that no page of the program was initially in primary memory, then there must be min(i,m) pages in primary memory when the program is in state S_i. Therefore, noting Eq. (2.4.1) or Eq. (2.4.2), the probabilities associated with these state transitions are found as follows:

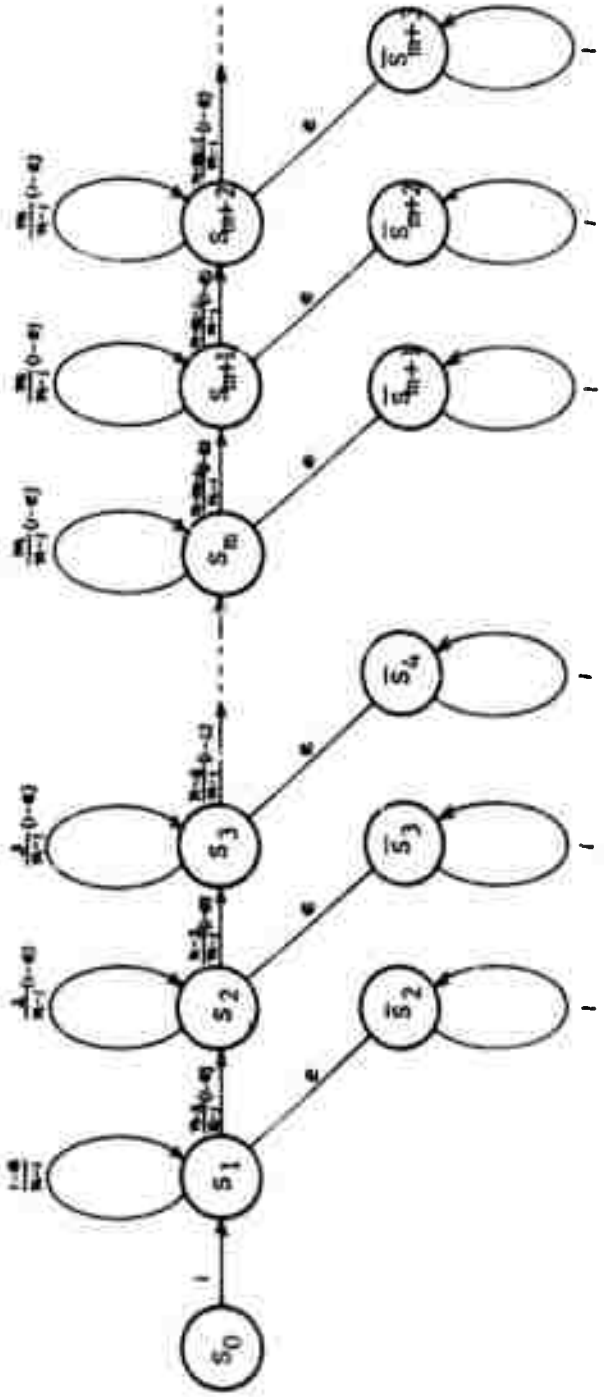
$$\begin{aligned} p_{i,i} &= \min(i,m)\frac{1-e}{n-1} \\ p_{i,i+1} &= (n-1-\min(i,m))\frac{1-e}{n-1} \\ \overline{p}_{i,i+1} &= e \end{aligned} \qquad (2.4.4)$$

Thus, the entire paging behavior of the random program is described by the state transition diagram of Figure 2-6. It should be noted that the behavior of this random program is now formulated as a discrete-time Markov process. It is seen that if the program fits into the available memory space ($n \leq m+1$) the number of page faults to be generated is at most n, but if the program does not fit into the memory space ($n > m+1$) an infinite number of page faults can occur at least theoretically. A comparison of Figure 2-6 (a) and Figure 2-6 (b) reveals that the Figure 2-6 (b) with $n = m+1$ reduces to the Figure 2-6 (a). This means that it



(a) $n \leq m+1$

Figure 2-6 State Transition Diagram of the Random Behavior Model



(b) $n \geq m+1$

Figure 2-6 State Transition Diagram of the Random Behavior Model

suffices to analyze only the latter case, i.e., $n \geq m+1$, because the result of the former case ($n \leq m+1$) can be obtained simply by setting $n = m+1$ in the analysis of the latter case ($n \geq m+1$). Therefore, we assume hereafter that $n \geq m+1$, i.e., the program does not (necessarily) fit into the primary memory space.

Let $p(n_p)$ be the probability that $n_p (2 \leq n_p)$ page faults result during the execution of this random program. This probability can be obtained by multiplying all the probabilities given on the branches leading to the state $\bar{S}_{n_p}$ of Figure 2-6 (b), with an extra consideration on looping at each intermediate state [H2]. If $2 \leq n_p \leq m+1$, we obtain

$$p(n_p) = 1 \cdot \left(\frac{1}{1-\frac{1-e}{n-1}}\right) \cdot \left[\left(\frac{n-2}{n-1}(1-e)\right) \cdot \left(\frac{1}{1-\frac{2(1-e)}{n-1}}\right)\right]$$
$$\cdot \left[\left(\frac{n-3}{n-1}(1-e)\right) \cdot \left(\frac{1}{1-\frac{3(1-e)}{n-1}}\right)\right] \cdot \cdots \cdot \left[\left(\frac{n-n_p+1}{n-1}(1-e) \cdot \left(\frac{1}{1-\frac{(n_p-1)(1-e)}{n-1}}\right)\right] \cdot e$$
$$= \frac{n-1}{n+e-2} \cdot \left[\frac{(n-2)(1-e)}{n+2e-3}\right] \cdot \left[\frac{(n-3)(1-e)}{n+3e-4}\right] \cdot \cdots \cdot \left[\frac{(n-n_p+1)(1-e)}{n+(n_p-1)e-n_p}\right] \cdot e$$
$$= \frac{e}{1-e} \prod_{i=1}^{n_p-1} \left(\frac{(n-i)(1-e)}{n+ie-(i+1)}\right) \qquad 2 \leq n_p \leq m+1 \qquad (2.4.5a)$$

If $n_p \geq m+1$, then $p(n_p)$ is similarly obtained as

$$p(n_p) = \frac{e}{1-e} \left[\prod_{i=1}^{m} \left(\frac{(n-i)(1-e)}{n+ie-(i+1)}\right)\right] \cdot \left[\frac{(n-m-1)(1-e)}{n+me-(m+1)}\right]^{n_p-m-1}$$
$$m+1 \leq n_p \qquad (2.4.5b)$$

Now let w be the ratio of the reference probability of the ending page and that of an ordinary page, i.e.,

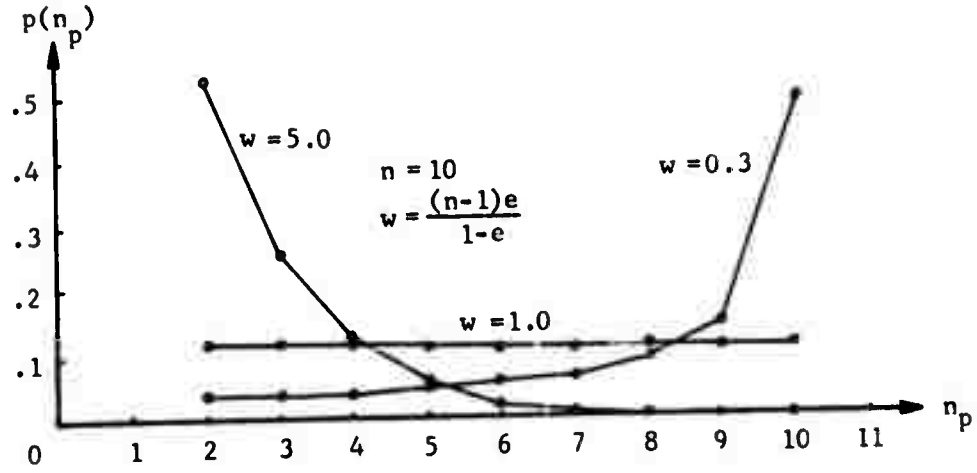
$$w = \frac{e}{(\frac{1-e}{n-1})} \qquad (2.4.6)$$

We call w the ending-page weight. If the ending-page weight is one, that is, if the reference probabilities of all the n pages of the random program are equal, Eq. (2.4.5) reduces to

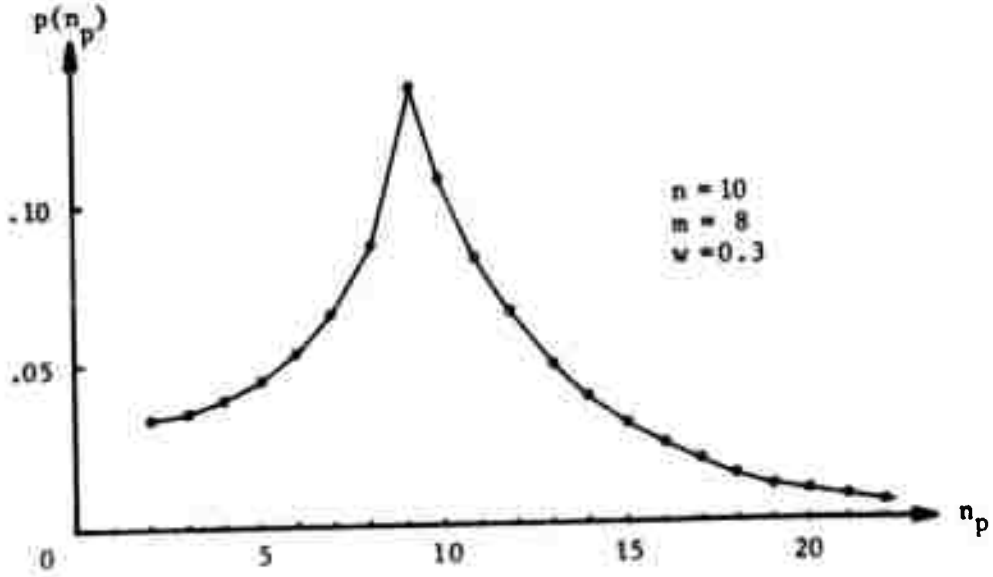
$$p(n_p) = \begin{cases} \frac{1}{n-1} & 2 \le n_p \le m+1 \qquad (2.4.7a) \\ \frac{1}{n-1}(\frac{n-m-1}{n-m})^{n_p-m-1} & m+1 \le n_p \qquad (2.4.7b) \end{cases}$$

From these results, we can readily obtain the average number, $E(n_p)$, of page faults to be caused by this program. To illustrate the shape of $p(n_p)$, some examples are given in Figure 2-7. Figure 2-7 (a) shows the effect of w on $p(n_p)$, for a random program which fits into the memory space, and therefore the value of n_p ranges from 2 to n. On the other hand, Figure 2-7 (b) shows an example of a random program which does not fit into the memory space and therefore the value of n_p ranges from 2 to infinity.

Next, we proceed to derive the average program execution time, $E(T_e)$. Noting that the program execution time is the sum of the holding times of all the intermediate states leading to the end state, we first derive the expected holding time, $E(t_i)$, of a state S_i. The holding time in S_i follows a geometric distribution with a parameter $1-p_{i,i}$, and therefore, noting Eq. (2.4.4),



(a) Effect of Ending Page Weight ($n \leq m+1$)



(b) Example of $p(n_p)$ for which $n > m+1$

Figure 2-7 Some Examples of $p(n_p)$

$$E(t_i) = t_0 \sum_{j=0}^{\infty} j \cdot (p_{i,i})^{j-1}(1-p_{i,i})$$

$$= \frac{t_0}{1-p_{i,i}}$$

$$= \begin{cases} \dfrac{n-1}{n-ie-(i+1)} & i \le m \qquad (2.4.8a) \\ \dfrac{n-1}{n-me-(m+1)} & m \le i \qquad (2.4.8b) \end{cases}$$

The general shape of $E(t_i)$ is shown in Figure 2-8. It is seen that as the number of pages in primary memory increases the expected holding time (or the mean headway between page faults) increases. From Eq. (2.4.8), we obtain the average program execution time with exactly n_p page faults, $E(T_e(n_p))$.

$$E(T_e(n_p)) = \sum_{i=0}^{n_p-1} E(t_i) \qquad (2.4.9)$$

The general shapes of the curves which show the growth of the number of pages in primary memory and that of the number of pages faults are given in Figure 2-9. Averaging $E(T_e(n_p))$ obtained above, with respect to n_p, we can obtain the desired average execution time, $E(T_e)$, of the program, as follows.

$$E(T_e) = \sum_{n_p=2}^{\infty} p(n_p)\, E(T_e(n_p)) \qquad (2.4.10)$$

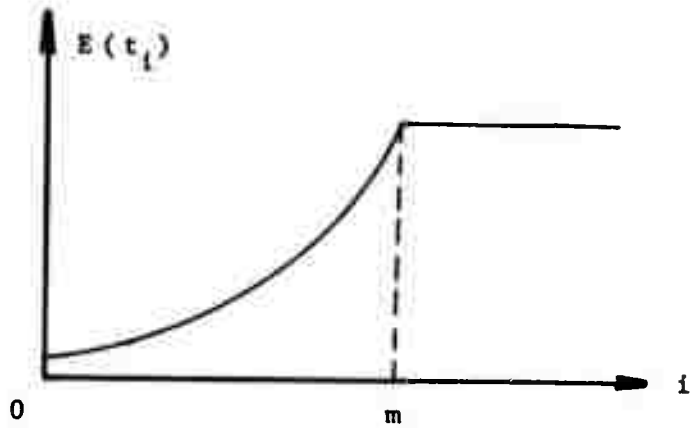


Figure 2-8 Mean Headway Between Page Faults

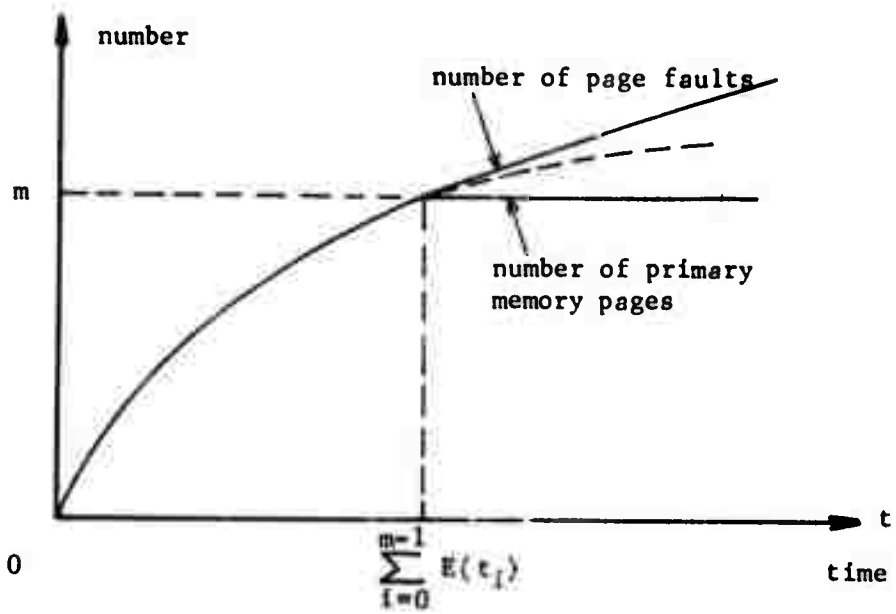


Figure 2-9 Growth of the Number of Primary Memory Pages and the Number of Page Faults

However, $E(T_e)$ may be obtained by a more direct method. Let S_s, S_r, and S_e denote respectively a starting state where the random program is about to start its execution with no page in primary memory, a running state where the program is running without ever referencing the ending page, and an ending state where the program has referenced the ending page. Noting that the reference probability of the ending page is assumed to be e (a constant) in any stage of the program execution, we find that these three states form a simple Markov chain shown in Figure 2-10 Therefore, the average execution time, $E(T_e)$, of this program is given by the first passage time to state S_e. Thus,

$$E(T_e) = \left(1 + \sum_{i=0}^{\infty} i \cdot (1-e)^{i-1} e \right) t_0$$

$$= \left(1 + \frac{1}{e} \right) t_0$$

$$= \left(\frac{n-1}{w} + 2 \right) t_0 \qquad (2.4.11)$$

It is seen that the reference probability of the ending page completely determines the average execution time of the program and that the average execution time is roughly proportional to the ratio of the program size and the ending-page weight.

Finally, we can obtain the mhbpf of this random program by combining the results of Eq. (2.4.5) and Eq. (2.4.11), as follows:

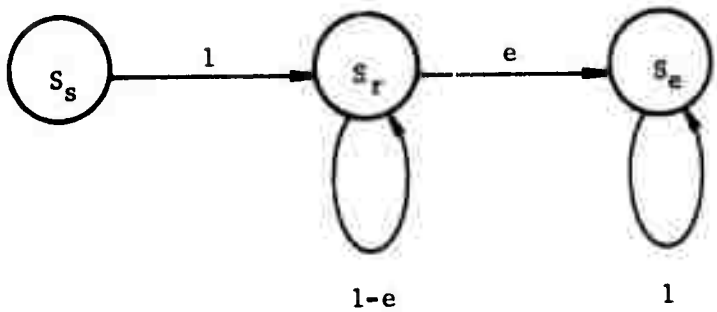


Figure 2-10 Simpler Derivation of the Average Execution Time of a Random Program

$$\text{mhbpf} = \frac{E(T_e)}{E(n_p)}$$

$$= \begin{cases} \dfrac{\left(\frac{n-1}{w} + 2\right) t_0}{\sum_{n_p=2}^{n} \left\{ \frac{e n_p}{1-e} \prod_{i=1}^{n_p-1} A(i) \right\}} & n \le m+1 \qquad (2.4.12a) \\ \dfrac{\left(\frac{n-1}{w} + 2\right) t_0}{\sum_{n_p=2}^{m+1} \left\{ \frac{e n_p}{1-e} \prod_{i=1}^{n_p-1} A(i) \right\} + \sum_{n_p=m+2}^{\infty} \left\{ \frac{e n_p}{1-e} \left(\prod_{i=1}^{m} A(i) \right) \cdot \left(\frac{(n-m-1)(1-e)}{n+me-(m+1)} \right)^{n_p-m-1} \right\}} & n \ge m+2 \qquad (2.4.12b) \end{cases}$$

where $A(i) = \dfrac{(n-i)(1-e)}{n+ie-(i+1)}$

In particular, if the reference probabilities of the ending page and an ordinary page are equal, i.e., $w = 1$, then Eq. (2.4.12) reduces to

$$\text{mhbpf} = \begin{cases} \dfrac{2(n+1)}{n+2} t_0 & n \le m+1 \qquad (2.4.13a) \\ \dfrac{2(n-1)(n+1)}{2n^2-2mn+(m+2)(m-1)} t_0 & n \ge m+2 \qquad (2.4.13b) \end{cases}$$

It can be easily verified that as m/n approaches zero, i.e., as the available memory size becomes much smaller than the program size, the mhbpf of the program approaches t_0, the unit time of this discrete-time

Markov process model.

2.4.3. Numerical Examples of Random Program Behavior

In the previous section, the average execution time, the average number of page faults, and the mhbpf of a random program were explicitly obtained as a function of the primary memory size (m), the program size (n), the ending-page weight (w), and the unit time of the discrete-time Markov process model (t_0). In this section, the effects of these parameters on the program's performance will be numerically evaluated.

As is clear from Figure 2-7 (a) and Eq. (2.4.11), the ending-page weight w has a major effect on both the average number of page faults and the program's average execution time. Therefore, the effect of w upon the program's mhbpf, which is the ratio of these two averages, was numerically evaluated in Figure 2-11. It is seen that the smaller the value of w becomes the longer mhbpf becomes. The other point to be noted is that mhbpf is almost independent of the program size so long as the program fits into the available primary memory space.

Next, we move on to the central theme of this section, that is, a numerical evaluation of the effect of primary memory size m, using Eq. (2.4.12). This result is summarized in Figure 2-12. It is observed that the curve representing the length of mhbpf branches into several downward curves each corresponding to a particular primary memory size as the program size increases. In particualr, it should be noted that as the program size exceeds the memory size the downward branch curve representing mhbpf abruptly decreases and from then on it continues to

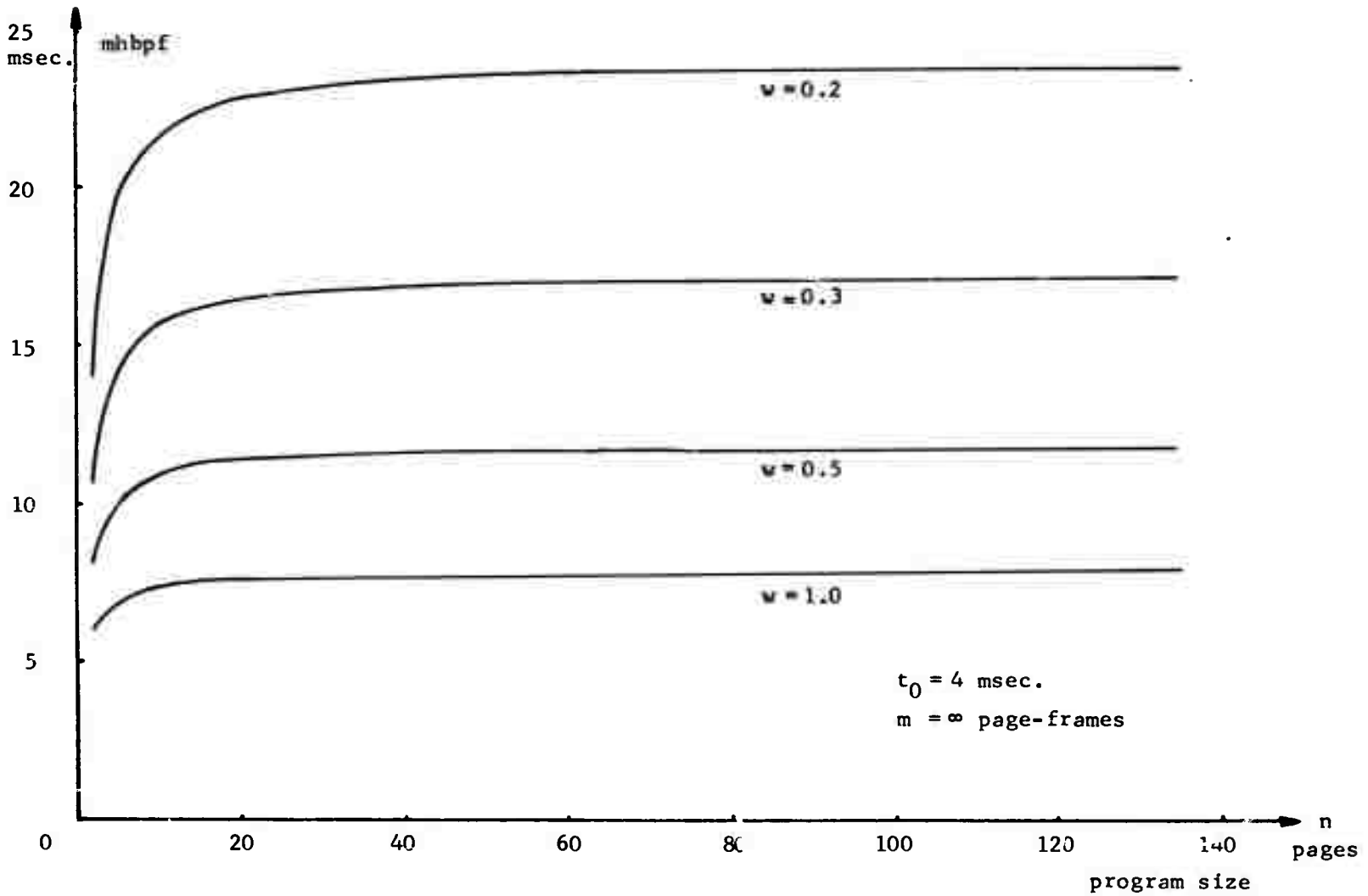


Figure 2-11 Effect of Ending-Page Weight upon mhbpf

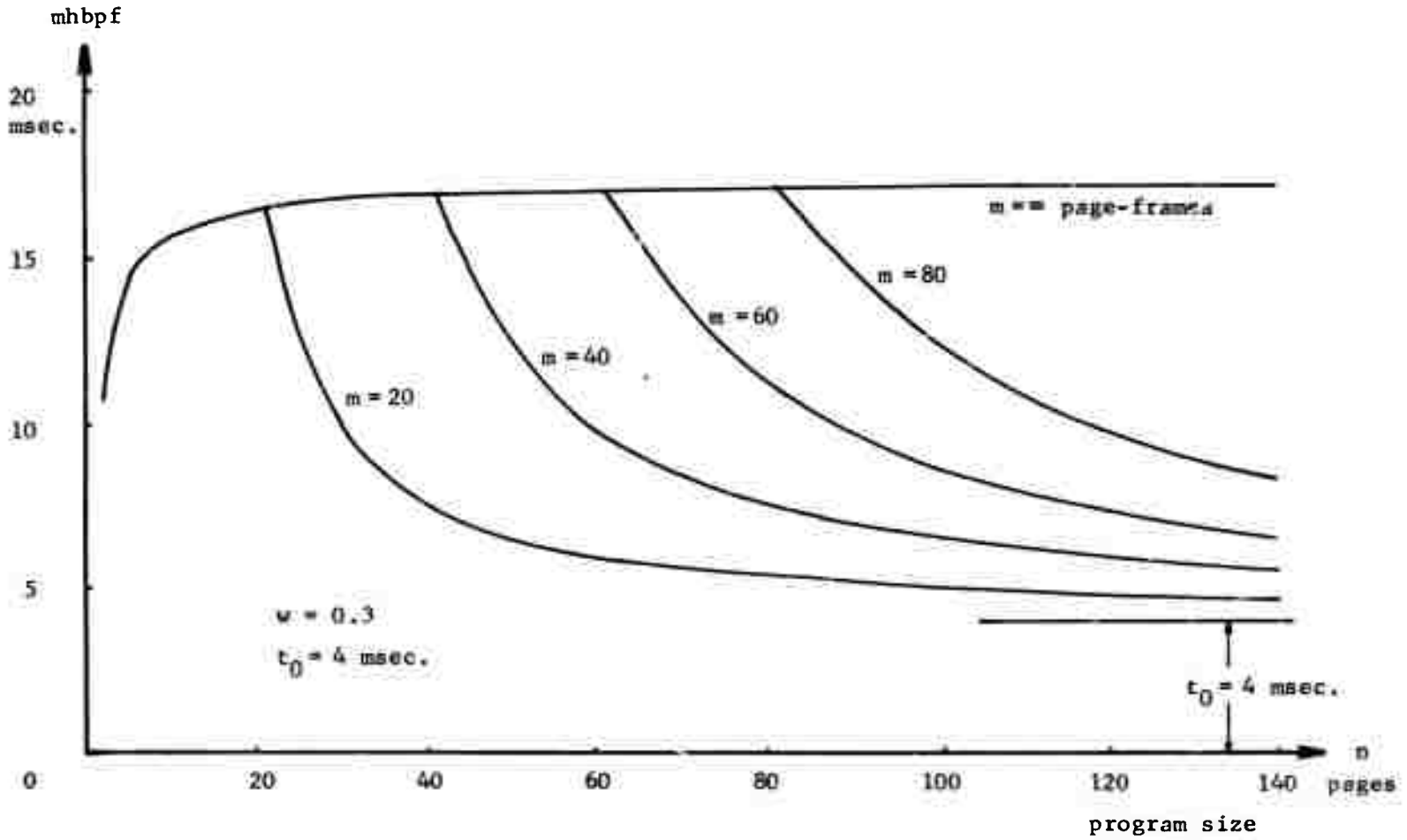


Figure 2-12 Effect of Primary Memory Size upon mhbpf

decrease only gradually, approaching the unit time t_0 of the model asymptotically. This kind of abrupt decrease of mhbpf was called thrashing by Denning [D2]. However, this abruptness shown in Figure 2-12 is somewhat exaggerated by the uniform page reference probabilities assumed in the random behavior model; if this model could include some less-frequently-used pages also this abrupt decrease of mhbpf would be moderated. The other way to interpret this figure is to fix the program size instead of the primary memory size. Then, mhbpf's of a given random program operating in the primary memory of various sizes can be obtained. For example, the mhbpf of a random program, with n = 100 pages, w = 0.3, and t_0 = 4 millisecond, operating in the primary memory of 80 page-frames is found to be about twice as long as that of the same program operating in the primary memory of 40 page-frames. The ratio of mhbpf's corresponding to two arbitrarily chosen primary memory sizes, however, depends on the chosen values of m and n.

Finally, the effect of primary size m upon the program's average total execution time was numerically evaluated, assuming that the average total execution time of a random program is given by the sum of the program's average execution time and the average total time to be spent for handling page faults. In other words, denoting the expected paging overhead time necessary for handling a page fault by $\bar{t}_p$,

$$E(\text{total execution time}) = E(T_e) + E(n_p)\cdot\bar{t}_p \qquad (2.4.14)$$

The result is shown in Figure 2-13. It is seen that the average total execution time of each random program gradually increases as the available primary memory size becomes smaller than the program size. This increase

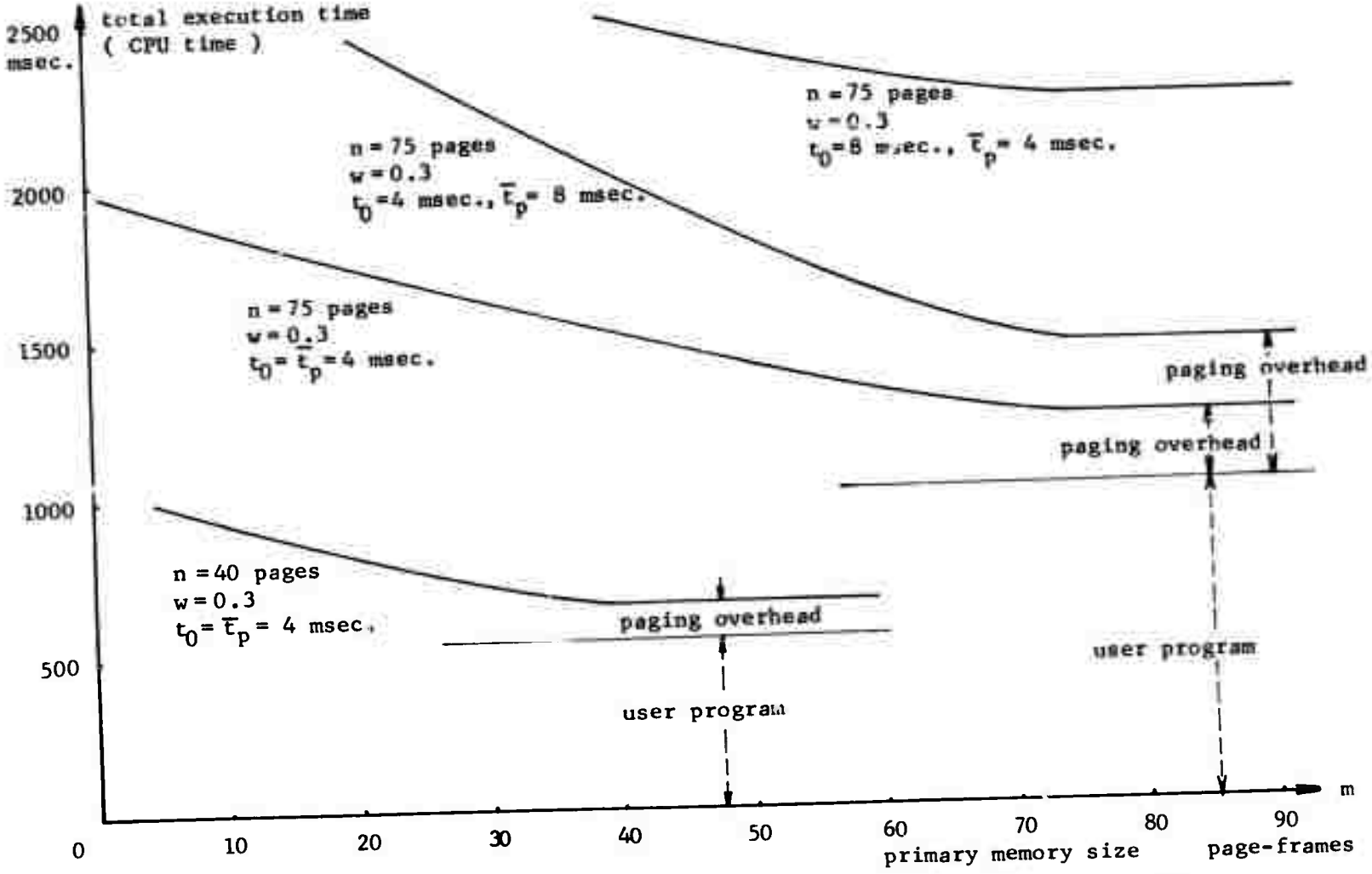


Figure 2-13 Effect of Primary Memory Size on Total Program Execution Time

in total execution time is due entirely to the increase in the number of page faults encountered during the program execution. Brawn and Gustavson [B4] measured the paging behavior of several actual programs on the IBM M44/44X system and plotted the curves corresponding to those shown in Figure 2-13. Their measurement result of actual computer programs is very similar to that of Figure 2-13, except in the following two respects. First, the increase of program execution time was observed by them when the available primary memory size became smaller than a certain fraction (typically, 60 percent rather than 100 percent seen in Figure 2-13) of the program size. This probably means that at most 60 percent of the entire program pages were referenced at one time because of locality of information references. Second, the slope of their curves representing increasing program execution time is steeper than that of those shown in Figure 2-13. This probably means that the (expected) paging overhead time of their system is larger than the values (4 and 8 msec.) that we assumed in Figure 2-13.

2.5. Macroscopic Paging Performance Model for Multiprogramming

In the previous sections, we assumed a partitioned primary memory with a local page replacement algorithm, and therefore we did not consider sharing of segments among eligible user processes. However, on actual computer systems like the one described in Section 1.2, any non-resident program as well as any resident supervisor program is sharable among eligible user processes. Therefore, we will now proceed to consider the effect of sharing of resident and non-resident programs upon the paging behavior of these user processes. First, we will develop a simple model of sharing in estimating the number of page-frames effectively available to each user process under multiprogramming. Then, in deriving the mhbpf of these user processes, we will use a linear model of paging performance discussed by Saltzer [S3] as a babis. By combining the results of these two models, it will be possible to express the mtbpf of an eligible user process as a function of the size (m_r, m_n, and M) of primary memory, the degree (q) of multiprogramming, and the degree of sharing.

2.5.1. Model of Sharing among Eligible User Processes

As we have seen in Section 2.5, the number of primary memory page-frames available to a user process plays a major role in dertermining the paging behavior of programs. In this section, we will develop a simple model of sharing and then use it to estimate the number of primary memory page-frames, including those being shared, available to each user process under multiprogramming.

We assume that the paging behavior of all user processes under multiprogramming, i.e., all eligible user processes, is probabilistically identical; we will consider only one class of user processes whose paging behavior is probabilistically specified. We assume that each eligible user process is scheduled for service by a processor under the FCFS discipline and therefore the processors are equally accessible to the processes. Thus, it is fairly reasonable to expect that each of q existing eligible user processes uses approximately the same amount of primary memory under the management of a demand-paging global page replacement algorithm. It is assumed that the entire primary memory space is fully utilized by these q eligible user processes under the memory allocation depicted in Figure 2-1.

Now we are ready to present a model of (segment) sharing among the eligible user processes.

Model of Sharing

(1) Sharing of resident programs: $a \times 10^2$ percent of m_r page-frames occupied by the resident programs is actually used by each eligible user process, on the average.

(2) Sharing of non-resident programs: $b \times 10^2$ percent of the memory space (m_0 page-frames in size) of each eligible user process, taken out of m_n page-frames reserved for non-resident programs, overlaps with the memory space (m_0 page-frames in size) of another eligible user process, on the average. Any such overlapping of memory spaces is independent of any other overlapping of memory spaces.

We will first investigate the properties of sharing of non-resident programs. For this purpose, let $P_i (i=1,2,\ldots,q)$ be the i-th eligible user process, $m_0(P_1,P_2,\ldots,P_i)$ be the average size of overlapping of i (non-resident program) memory spaces used by processes P_1 through P_i, and $m_0(P_1+P_2+\ldots+P_i)$ be the average size of (non-resident program) joint memory space of processes P_1 through P_i. Then, under the assumptions of our model of sharing, we have

$$\begin{aligned} m_0(P_1,\ P_2) &= b \cdot m_0 \\ m_0(P_1,\ P_2,\ P_3) &= b^2 \cdot m_0 \\ &\vdots \\ m_0(P_1,\ P_2,\ \ldots\ ,\ P_i) &= b^{i-1} \cdot m_0 \end{aligned} \qquad (2.5.1)$$

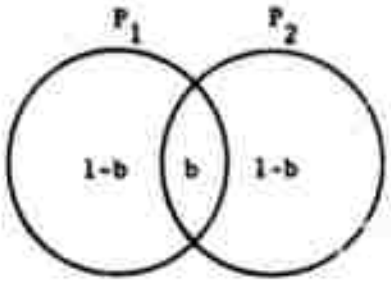
Therefore, as illustrated in Figure 2-14, we get

$$m_0(P_1+P_2) = (2-b)m_0$$

$$m_0(P_1+P_2+P_3) = (3-3b+b^2)m_0$$

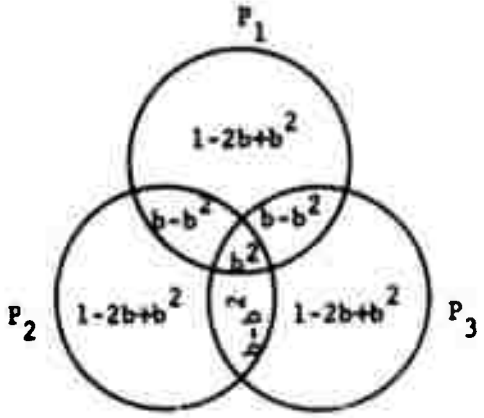
In general [F1], we obtain

$$\begin{aligned} & m_0(P_1+P_2+\ \ldots\ +P_q) \\ &= \underset{\text{all singles}}{\Sigma}\, m_0(P_\ell) - \underset{\text{all pairs}}{\Sigma}\, m_0(P_\ell,\ P_m) + \underset{\text{all triples}}{\Sigma}\, m_0(P_\ell,\ P_m,\ P_n) - \\ &\qquad \ldots + (-1)^{q-1} \underset{\text{all q-tuples}}{\Sigma}\, m_0(P_1,\ P_2,\ \ldots\ ,\ P_q) \\ &= [\tbinom{q}{1}\cdot 1 + \tbinom{q}{2}\cdot(-b) + \tbinom{q}{3}\cdot(-b)^2 + \ldots + \tbinom{q}{q}\cdot(-b)^{q-1}]\cdot m_0 \\ &= \sum_{i=1}^{q} \tbinom{q}{i}\cdot(-b)^{i-1}\cdot m_0 \\ &= [\frac{1-(1-b)^q}{b}]\cdot m_0 \end{aligned} \qquad (2.5.2)$$



$$m_0(P_1 + P_2) = (2-b)\, m_0$$

(a) two processes



$$m_0(P_1 + P_2 + P_3) = (3-3b+b^2)\, m_0$$

(b) three processes

Figure 2-14 Overlapping of Non-Resident Program Memory Spaces

Thus, we have obtained the average size of total memory space which is required by the non-resident programs invoked by a set of q eligible user processes.

This average size of primary memory space associated with the q eligible user processes can be easily proved to have the following properties:

Property 1

$$\lim_{b \to 0} m_0(P_1 + P_2 + \ldots + P_q) = q m_0 \tag{2.5.3a}$$

$$\lim_{b \to 1} m_0(P_1 + P_2 + \ldots + P_q) = m_0 \tag{2.5.3b}$$

Property 2

$$\Delta m_0(q) \equiv m_0(P_1 + P_2 + \ldots + P_{q+1}) - m_0(P_1 + P_2 + \ldots + P_q)$$

$$= (1-b)^q \cdot m_0 \tag{2.5.4}$$

Property 3

$$\lim_{q \to \infty} m_0(P_1 + P_2 + \ldots + P_q) = \frac{m_0}{b} \tag{2.5.5}$$

The dependency of $m_0(P_1 + P_2 + \ldots + P_q)$ upon the degree(b) of sharing is seen in Property 1; Eq. (2.5.3a) means that if there is not any sharing at all the total memory space required by the q user processes under multiprogramming is simply q times as large as the memory space required by each process, and Eq. (2.5.3b) means that if non-resident programs are completely shared by the q user processes under multiprogramming (consider a special-purpose computer system) the total

memory space required is equal to the space of only one process. Property 2 indicates that $m_0(P_1+P_2+\ldots+P_q)$ is a non-decreasing function of the degree of multiprogramming and that if one more user process must be made eligible without changing the effectively available memory space (m_0) of each process then the additional primary memory space to be required is given by a geometric probability distribution. Property 3 means that an infinite number of eligible user processes can coexist using a finite primary memory space because we assumed "independent" sharing (overlapping of memory spaces) of the processes' non-resident program.

Finally, we will evaluate the average number of page-frames which are available to each of these q user processes under multiprogramming, assuming that a given primary memory of M page-frames is divided into two areas, i.e., the area with m_r page-frames for resident programs and the area with $m_n(=M-m_r)$ page-frames for non-resident programs (see Figure 2-1).

From the first assumption concerning our model of sharing, the average number of page-frames storing the pages of resident programs used by an eligible process is am_r. On the other hand, the average number (m_0) of page-frames available to the eligible process for storing non-resident programs is found from Eq. (2.5.2) to be

$$m_0 = m_n \cdot \frac{b}{1-(1-b)^q} \qquad (2.5.6)$$

Therefore, the average total number (m) of page-frames available to each of these q user processes under multiprogramming (using the primary

memory depicted by Figure 2-1) is obtained as follows:

$$m = a\,m_r + \frac{b\,m_n}{1-(1-b)^q} \qquad (2.5.7)$$

The overall percentile saving of primary memory due to sharing of segments is given by 1 - (M/qm).

2.5.2. Evaluation of mtbpf in Multiprogramming Environment

In this section, we will present a simple macroscopic model of paging behavior which relates the average number (m) of page-frames effectively available to a user process under multiprogramming to the mtbpf of the user process.

Before getting into the development of this model, we must briefly consider tbpf (time between page faults) and mtbpf (mean time between page faults). It was previously stated that each burst of continuous program execution, i.e., tbpf, consists of at least user program execution and paging overhead execution. To be more realistic, tbpf includes a proportion of time (t_m) spent as miscellaneous overhead of the supervisor[1], as well as a proportion of time (t_u) spent as a user's useful work and a proportion of time (t_p) spent as paging overhead of the supervisor. That is to say,

$$tbpf = t_u + t_p + t_m \qquad (2.5.8)$$

---

[1] This complication is now necessary as a step to evaluate the amount of user-oriented computation in Chapter 3.

The first component, t_u, of the tbpf includes the execution of both user programs and non-overhead-type (user-oriented) supervisor programs (e.g., input/output control programs), and is generally called a headway between page faults (hbpf). The second component, t_p, of the tbpf includes several supervisory functions required to handle a page fault, as roughly explained in Section 2.1, and is generally called a paging overhead time. The third component, t_m, of the tbpf includes several supervisory functions required to handle miscellaneous faults, such as segment faults, protection faults, various non-paging interrupts, etc., which occur during the duration of the tbpf, and is generally called a miscellaneous overhead time. Denoting average values by barred symbols, the mean time between page faults (mtbpf) is expressed as

$$\text{mtbpf} = \bar{t}_u + \bar{t}_p + \bar{t}_m, \tag{2.5.9}$$

where $\bar{t}_u$, $\bar{t}_p$, and $\bar{t}_m$ respectively represents the mean headway between page faults (mhbpf), the mean paging overhead time, and the mean miscellaneous overhead time.

We assume that the mhbpf (or $\bar{t}_u$) of a user process under multiprogramming can be determined by the average number (m) of page-frames that are effectively available to the user process and, in particular, that the mhbpf is generally expressed as

$$\begin{aligned} \bar{t}_u &\equiv \bar{t}_u(m) \\ &= c_0 + c_1 m + c_2 m^2 + c_3 m^3 + \ldots \end{aligned} \tag{2.5.10}$$

where $c_i (i=0,1,2,\ldots)$ are constant coefficients. On the other hand, the miscellaneous faults (segment faults, protection faults, non-paging

interrupts, etc.) are likely to occur uniformly at any instant during the execution of a user's useful work. Therefore, it is reasonable to assume that the miscellaneous overhead time is linearly proportional to the mhbpf. That is to say,

$$\bar{t}_m \equiv \delta \, \bar{t}_u \tag{2.5.11}$$

where δ (a constant) is called the miscellaneous overhead coefficient. It has been experimentally observed on the Multics system that the paging overhead time is roughly independent of the primary memory size, although the execution times of certain supervisory functions included in the paging overhead (e.g., the execution of a page replacement algorithm) may depend on the primary memory size. Therefore, we assume that the paging overhead time, $\bar{t}_p$, does not depend on the primary memory size. Thus, the mtbpf of an eligible user process which effectively uses m page-frames of the primary memory is given by

$$\begin{aligned} \text{mtbpf} &\equiv \text{mtbpf}(m) \\ &= (\bar{t}_u + \bar{t}_m) + \bar{t}_p \\ &= (1+\delta) \cdot \bar{t}_u(m) + \bar{t}_p \end{aligned} \tag{2.5.12}$$

On the other hand, Saltzer [S3] describes a linear model of paging performance, which was developed from performance measurements of the Multics system. He observed that the mhbtf of an eligible user process is approximately linearly proportional to the size of

primary memory[1], for a wide range of the primary memory size. This observation justifies (at least partially) the approximation of $\bar{t}_u$ with only the second term of Eq. (2.5.10). Thus,

$$t_u(m) = c_1 \cdot m \tag{2.5.13}$$

We call this simple model of mhbpf a linear paging model and we expect it to serve as a macroscopic model of paging behavior of user processes.

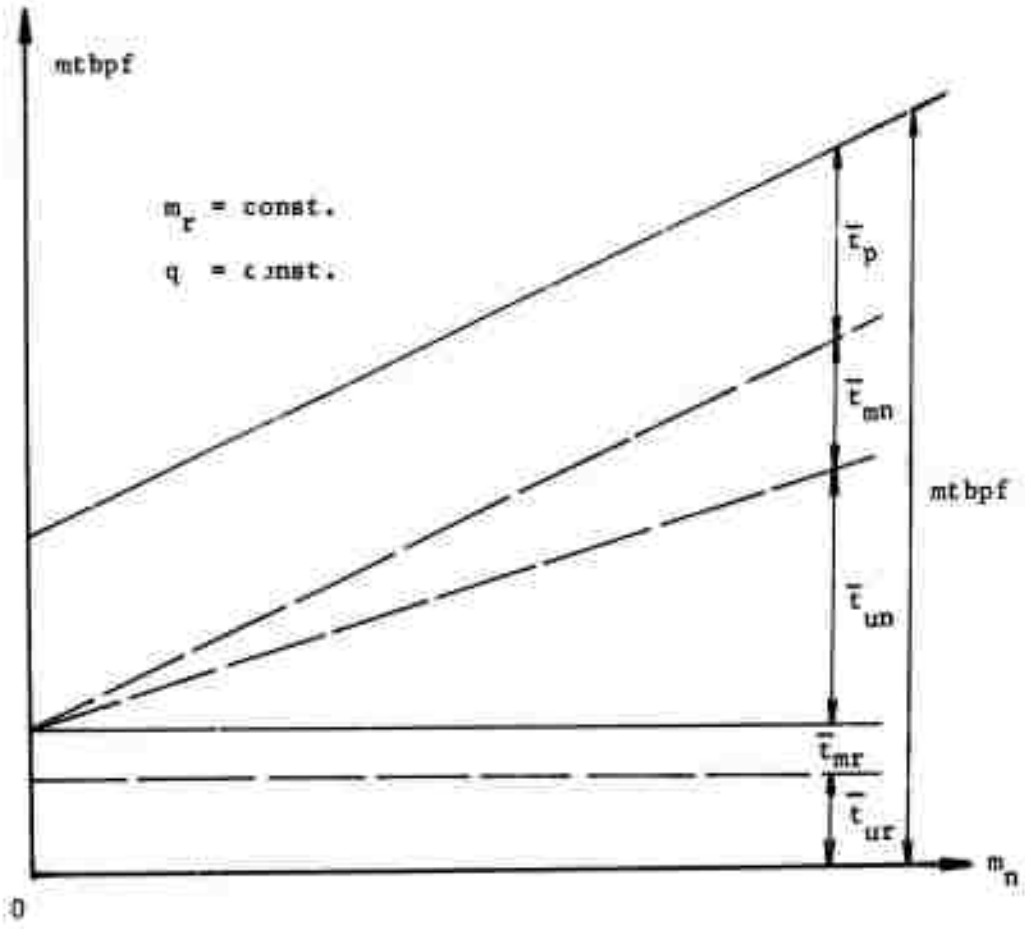
Therefore, combining this linear paging model and the model of sharing developed in the previous section, we obtain the following expression for the mtbpf of an eligible user process.

$$\begin{aligned} \text{mtbpf} &= (1+\delta) \cdot \bar{t}_u(m) + \bar{t}_p \\ &= (1+\delta) \cdot c_1 m + \bar{t}_p \\ &= (1+\delta) \cdot c_1 \cdot \left(a m_r + \frac{b m_n}{1-(1-b)^q} \right) + \bar{t}_p \end{aligned} \tag{2.5.14}$$

The behavior of the above three components ($\bar{t}_u$, $\bar{t}_m$, and $\bar{t}_p$) of the mtbpf is explained as a function of the primary memory space (m_n) available to non-resident programs in Figure 2-15. The gross behavior of the mtbpf as a function of m_n and the degree (q) of multiprogramming is

---

[1]This observation was made about the averaged behavior of an aggregate of programs invoked by an eligible user process. It does not necessarily represent the behavior of a particular program.



$\bar{t}_{ur}$; mean headway between page faults made in resident programs

$\bar{t}_{mr}$; mean miscellaneous overhead time spent in resident programs

$\bar{t}_{un}$; mean headway between page faults made in non-resident programs

$\bar{t}_{mn}$; mean miscellaneous overhead time spent in non-resident programs

$$\bar{t}_u = \bar{t}_{ur} + \bar{t}_{un} , \quad \bar{t}_m = \bar{t}_{mr} + \bar{t}_{mn} , \quad \text{mtbpf} = \bar{t}_u + \bar{t}_m + \bar{t}_p .$$

Figure 2-15 Behavior of Various Components of mtbpf

similarly shown in Figure 2-16. It should be noted that the mtbpf of an eligible user process under multiprogramming of a given degree increases linearly with the available primary memory size. Finally, to evaluate the effect of sharing upon the mtbpf of eligible user processes under multiprogramming, three cases concerning sharing (no sharing, ten percent sharing, twenty percent sharing of non-resident programs) were considered. This result is given in Table 2-2. It is observed that the effect of sharing upon mtbpf becomes more evident as the number of user processes that must compete with each other within a fixed amount of primary memory increases. Later in Chapter 5, the gain in overall system performance (i.e., the system throughput and the system response time) due to sharing of programs will be numerically evaluated.

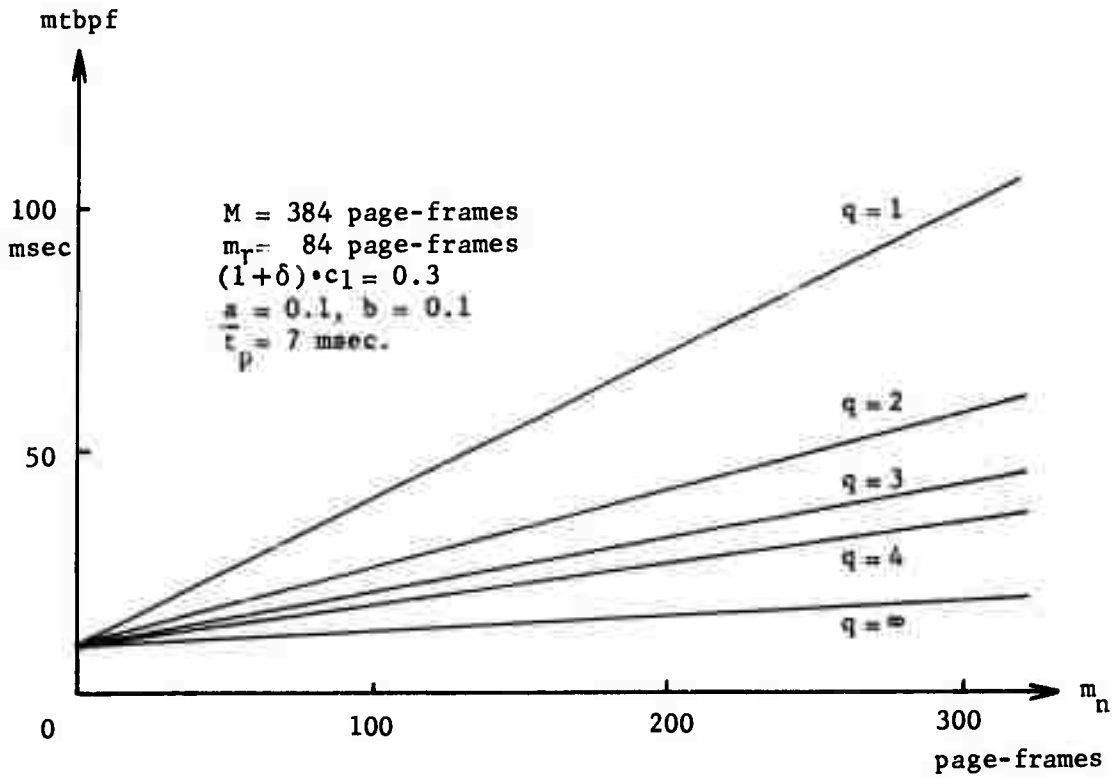


Figure 2-16 Effect of Primary Memory Size and Degree of Sharing upon mtbpf

Table 2-2 Effect of Multiprogramming and Sharing upon mtbpf

Degree of Multiprogramming q	Degree of Sharing of Non-Resident Programs b = 0	b = 0.1	b = 0.2
1	99.5 msec	99.5 msec	99.5 msec
2	54.5	56.9	59.5
3	39.5	42.7	46.4
4	32.0	35.7	40.0
5	27.5	31.5	36.3
6	24.5	28.7	33.9
7	22.3	26.8	32.3
8	20.7	25.3	31.1
.	.	.	.
.	.	.	.
.	.	.	.
∞	9.5	18.5	27.5

M = 384 page-frames, $m_r = 84$ page-frames, $m_n = 300$ page-frames, $(1+\delta)\cdot c_1 = 0.3$, $a = 0.1$, $\bar{t}_p = 7$ msec.

2.6. Summary

Because program behavior is so complex that we have developed several models each of which studies only a particular aspect of program behavior, rather than attempting to develop a single "universal" program behavior model. The first three models respectively attack the PRA (page replacement algorithm) studies, the page size problem, and the memory size problem. The analyses of these models respectively allows us to evaluate numerically the possible differences in program performance (the mean time between page faults) due to the PRA (page replacement algirithm) in use, the page size being used, and the size of primary memory available to a user process. The macroscopic paging performance model described in Section 2.5 is based on a model of sharing and Saltzer's linear paging model, and allows us to evaluate the mtbpf (mean time between page faults) of user processes under multiprogramming of a given degree.

CHAPTER 3

THROUGHPUT ANALYSIS

3.1. Introduction

Having studied program behavior within a limited amount of primary memory, we shall proceed to evaluate the performance of a processing system consisting of processors, primary and seconday memories, and user and supervisor programs (see Figure 1-4). The hardware is, in general, assumed to possess multiple processors and multiple primary memory units, with secondary memory system as part of the virtual memory (see Fig. 1-2). The operating system is assumed to allow multiple jobs to possess some of their pages in primary memory using a multiprogramming mechanism.

Our ultimate goal of this thesis is to develop a methodology to optimize such multi-processer multi-memory multiprogrammed virtual-memory computer system in cost-performance. As the first step towards this goal, Chapter 3 presents a method to evaluate the performance of

a processing system of a given hardware configuration (number of processors, number of primary memory units, speed of processors, speed of memory system, size of memory system, etc.), a given software configuration (size of resident supervisor, various overhead times, degree of multiprogramming, etc.) and given user program characteristics (mhbpf, etc.).

We choose system throughput, i.e., the average number of jobs that can be completed within a unit time, as our performance measure of the processing system. In evaluating the throughput of the processing system of a given configuration, various wastages of the system's computational capacity (various idle times and system overhead times to be defined) will be considered. Therefore, the system's effective computational power can be evaluated accurately. The result of this analysis then allows us to optimize the degree of multiprogramming realistically in such a way that throughput of the system is maximized. It will be seen in Chapter 4 that this optimization approach leads to the minimization of the average response time experienced by each interactive user of the computer system.

3.2. Preliminary Considerations

Before getting into a mathematical anaysis of the system throughput of a multi-processor multi-memory multi-programmed virtual-memory computer, some preliminary considerations will be given in this section in order to put the following analysis in proper perspective.

We have seen in Chapter 2 how the three components ($\bar{t}_u$, $\bar{t}_m$, $\bar{t}_p$) of mtbpf depend on the primary memory size. We now consider how each of these can be affected by a multi-processor multi-memory configuration of the processing system. A program's execution time is generally prolonged on a multi-processor computer system because of multi-processor interference such as memory cycle interference and data-base lockout. The former is the interference caused by the occasional conflict of multiple processors for a memory cycle of a particular primary memory unit and the latter is the interference caused by the occasional conflict of multiple processors for the use of a particular shared, writable data-base which cannot allow simultaneous accesses of multiple processors. Because a memroy cycle of a primary memory unit or a shared writable data-base can be used only serially, only one of the conflicting processors can use it immediately and others must wait for their turn to use it. Therefore, mtbpf is prolonged on the multi-processor computer system.

Table 3-1 summarizes the way in which the dependency of the three components of mtbpf, i.e.,

$\bar{t}_u$ (mean headway between page faults),
$\bar{t}_m$ (mean miscellaneous overhead time), and
$\bar{t}_p$ (mean paging overhead time),

upon the multi-processor interference as well as upon the primary memory size will be considered in later mathematical analyses. It is assumed that all fault handlers (page fault handler, segment fault handler, protection fault handler, etc.) of the supervisor are either permanently resident or almost always resident in primary memory by virtue of the fact that these most-frequently-used programs are likely to be shared by many user processes. On the other hand, non-resident programs such as user programs are assumed to be initially in the secondary memory system and to be brought into primary memory under demand paging, as stated in Chapter 2. Under these assumptions, we will use the macroscopic paging performance model described in Section 2.5; $\bar{t}_p$ is independent of primary memory size but $\bar{t}_u$ and $\bar{t}_m$ depend on (are roughly linearly proportional to) the size of primary memory effectively available to each user process under multiprogramming, as shown in the first row of Table 3-1.

It is well-known that the execution of a page fault handler involves some shared, writable data-bases (i.e., resource tables and resource queues [S1]) which may significantly delay the execution of the page fault handler on the multi-processor computer system, while the execution of users' useful work or of miscellaneous fault handlers normally does not involve such frequently-used shared writable data-bases. Therefore, only $\bar{t}_p$ is assumed to be significantly affected by data-base lockout, as shown in Table 3-1. On the other hand, the execution of any program is subject to memory cycle interference on the multi-processor computer system. Therefore, all three components of mtbpf are assumed to be affected by memory cycle interference, as shown in Table 3-1. It has been actually found on the Multics system (to be reported in

Table 3-1 Does Each Processor Time Depend on the Following Factors?

factors \ processor time	mean headway between page faults $\bar{t}_u$	mean miscellaneous overhead time $\bar{t}_m$	mean paging overhead time $\bar{t}_p$
effective size of primary memory	yes	yes	no
memory cycle interference of multi-processor configuration	yes	yes	yes
data-base lockout of multi-processor configuration	no	no	yes

Section 5.2.1) that a noticeable amount of processor time is lost due to the multi-processor interference.

In order to consider quantitatively the effect of this multi-processor interference, let t_{ui}, t_{mi}, and t_{pi} denote the t_u, t_m, and t_p of an i-processor (i=1,2) computer system. Letting K_m ($1 \leq K_m \leq 2$) be the slow-down factor due to memory cycle interference on a dual processor system, we have the following relationship between the t_u of a single processor system and that of a dual processor system with the same configuration.

$$\bar{t}_{u2} = K_m \bar{t}_{u1} \qquad (3.2.1)$$

where the value of K_m will be determined in a later analysis. Assuming the same degree of memory cycle interference for the handling of miscellaneous faults, we have

$$\bar{t}_{m2} = K_m \bar{t}_{m1} \quad . \qquad (3.2.2)$$

On the other hand, the relation between the paging overhead time of a single processor system and that of a dual processor system is more complex because of the additional interference caused by data-base lockout. We assume that the execution time of the page fault handler on a dual processor system is stretched by a factor of K_ℓ ($1 \leq K_\ell \leq 2$) because of data-base lockout and is further prolonged by a factor of K_m because of memory cycle interference. That is to say,

$$\bar{t}_{p2} = K_m \cdot K_\ell \cdot \bar{t}_{p1} \qquad (3.2.3)$$

Under these conditions, the linear relation between $\bar{t}_{m1}$ and $\bar{t}_{u1}$ of Eq. (2.5.11) is now generalized to

$$t_{mi} = \delta \bar{t}_{ui} \qquad i = 1,2 \tag{3.2.4}$$

It should be noted that the miscellaneous overhead coefficient δ is independent of the number (i) of processors. Thus, mtbpf of the single and dual processor systems can be expressed as:

$$\begin{aligned} \text{mtbpf(1 CPU)} &= \bar{t}_{u1} + \bar{t}_{m1} + \bar{t}_{p1} \\ &= (1+\delta)\bar{t}_{u1} + \bar{t}_{p1} \\ &\equiv \text{mtbpf}_1 \end{aligned} \tag{3.2.5}$$

$$\begin{aligned} \text{mtbpf(2 CPUs)} &= \bar{t}_{u2} + \bar{t}_{m2} + \bar{t}_{p2} \\ &= K_m \bar{t}_{u1} + K_m \bar{t}_{m1} + K_m K_\ell \bar{t}_{p1} \\ &= K_m \cdot \left((1+\delta)\bar{t}_{u1} + K_\ell \bar{t}_{p1} \right) \\ &\equiv K_m \cdot (\text{mtbpf}_2) \end{aligned} \tag{3.2.6}$$

Finally, we are ready to give an expression for the percentage of the system's computational capacity spent as users' useful work (to be called the <u>percentile throughput</u> of the system), taking into consideration all of the above system overheads and the possible processor idle time. Let u be a utilization factor, of the processors of the processing system, which will be analytically obtained by the throughput analysis of this chapter as a function of various system parameters such as

(1) the number of processors,

(2) the number of primary memory units,

(3) the degree of multiprogramming,

(4) various system overheads,

(5) $mtbpf_i$ $(i=1,2)$

(6) mpft (the mean page fetch time) of the secondary memory system.

Then, the desired percentile throughput θ of the multi-processor computer system can be expressed as:

$$\theta = u \cdot \frac{\bar{t}_{ui}}{mtbpf(i \text{ CPUs})} \qquad (3.2.7)$$

Now let T be an arbitrary length of time. Then, the total time that can be spent as users' useful work by the i processors during T is $i\theta T$. Let $T_{e1}, T_{e2}, \ldots, T_{ek}$ be the execution times of k user jobs (excluding all the system overhead times) which were completed during T. Then, assuming that the computer system was under a full load, the system throughput $\underline{\theta}$ can be obtained as follows:

$$\begin{aligned}
\underline{\theta} &\equiv \lim_{T\to\infty} \frac{k}{T} \\
&= \lim_{T\to\infty} \left(\frac{k}{T} \cdot \frac{i\theta T}{T_{e1}+T_{e2}+ \cdots +T_{ek}} \right) \\
&= \lim_{T\to\infty} \frac{i\theta}{\frac{k}{(\sum_{j=1} T_{ej})/k}} \\
&= \frac{i\theta}{\bar{T}_e} \qquad (3.2.8)
\end{aligned}$$

where $\bar{T}_e$ is the average execution time required for a job's useful work. Therefore, it has been shown that the percentile throughput is linearly proportional to the system throughput.

3.3. Single Processor System

In this section, we will be concerned with the performance of a single processor processing system under multiprogramming of a fixed degree (q). It is assumed that the time between page faults (tbpf) of each user process follows a certain stationary probability distribution with a (constant) mean determined by the size of primary memory available to the job. (In detailed reality, tbpf or hbpf depends on the size of primary memory actually being used at the moment rather than the amount of memory that can be used, as seen in Chapter 2, but this initial microscopic transient behavior of programs is only macroscopically considered hereafter.) It is also assumed that the computer system under study is fully loaded with user jobs; there always exist at least q executable user jobs on the computer system. This means that q user processes are being multiprogrammed at all times. Therefore, in modeling the behavior of the multiprogramming mechanism described in the section surrounded by a broken line of Figure 1-1, we need not explicitly consider the completions of job executions or the entries of new eligible user processes. Furthermore, because the processing system has only one processor we need not consider any multi-processor interference in this section.

3.3.1. Single Processor Multiprogramming Model

A model of multiprogramming described here involves the following resources of the computer system.

(1) a processor queue (Q_p) which contains page-ready processes, i.e., the eligible user processes waiting for a service by a processor of the processing system

(2) a processor (CPU) which can service one eligible user process at a time

(3) a secondary memory system (SM) from which a missing-page is brought into primary memory on demand for the eligible user process that has requested the page.

As explained in Section 1.2, if an eligible user process is waiting for a processor's service in the processor queue Q_p, the process is said to be in the page-ready state. If the process is being serviced by the processor, it is said to be in the running state. If the process is waiting for a missing-page, it is said to be in the page-wait state. The (mean) length of time in the running state exactly corresponds to the (mean) time between page faults and the (mean) length of time in the page-wait state exactly corresponds to the (mean) page fetch time. All the eligible user processes simultaneously cycle around these three states in our conceptual model of multiprogramming, as shown in Figure 3-1.

We usually assume that the jobs in Q_p are scheduled under the FCFS discipline for the processor's service when the processor is available. A preemptive priority scheduling discipline is also considered briefly. We assume that a service by the secondary memory system to locate and transfer a missing-page of a process into primary memory is always immediately initiated and carried out in parallel

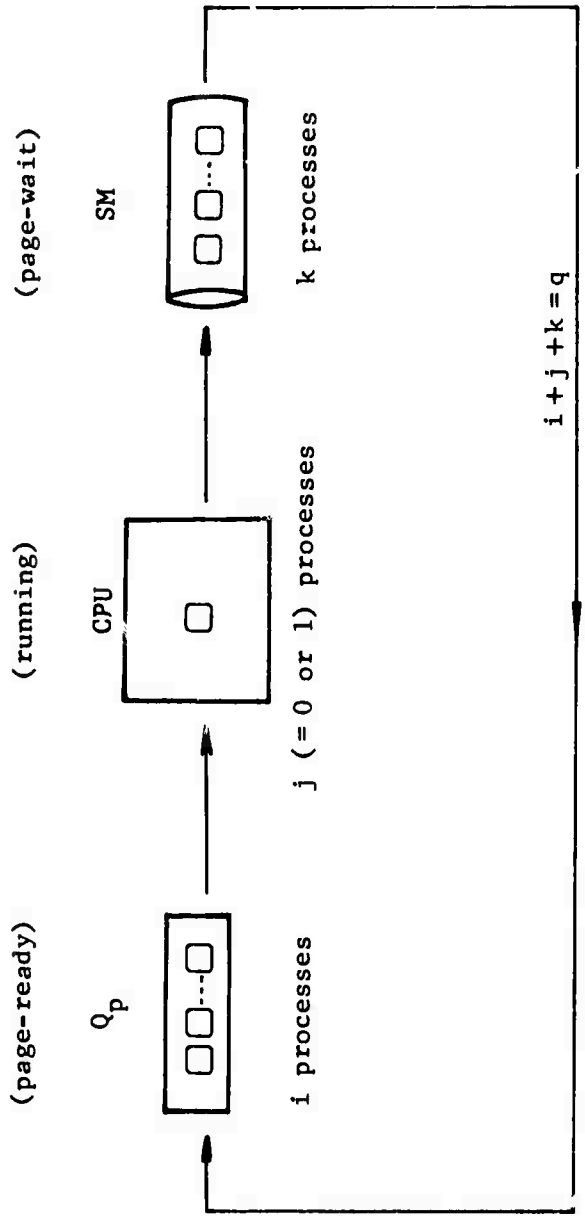


Figure 3-1 Locations of Eligible Processes

with services by the same secondary memory system for other page-wait processes; the input/output channel is assumed not to limit the rate of page transfers between primary and secondary memories.

Finally, we must specify the probabilistic property of both the time between page faults (tbpf) and the page fetch time (pft). We assume that tbpf independently follows an exponential distribution with a mean of $1/\alpha$ which is determined by the macroscopic paging performance model of Section 2.5. That is to say,

$$\left.\begin{aligned} & f(t) = \alpha e^{-\alpha t} \qquad t \geq 0 \\ & \text{where} \\ & \text{mtbpf}_1 = \frac{1}{\alpha} = \bar{t}_{u1} + \bar{t}_{m1} + \bar{t}_{p1} = (1+\delta)\bar{t}_{u1} + \bar{t}_{p1} \end{aligned}\right\} \qquad (3.3.1)$$

Similarly, we assume that pft independently follows an exponential distribution with a mean of $1/\beta$. That is to say,

$$g(t) = \beta e^{-\beta t} \qquad t \geq 0 \qquad (3.3.2)$$

where $\text{mpft} = 1/\beta$ should be determined by a separate analysis using a

secondary memory model (see Figure 1-4)[1]. This assumption would be reasonable especially if secondary memory consists of a combination of frequently-used high-speed devices and less-frequently-used low-speed devices.

3.3.2. Multiprogramming with FCFS Scheduling

We will proceed to evaluate the system throughput of the single processor processing system described in the previous section, assuming that the FCFS discipline is used in scheduling page-ready eligible processes [C3]. We assume that tbpf of each eligible user process follows the identical probability distribution of Eq. (3.3.1).

Then, a state of our model of the processing system is characterized by the number (i) of page-ready processes (waiting in Q_p), the number (j)

---

[1] Coffman's result concerning a paging drum [C5,C6] or the well-known result of M/G/1 queues [C8] may be used in this area. In particular, the built-in meters of the Multics system show that Coffman's result, reproduced below, is accurate enough to predict the mpft of a paging drum in practice. Letting T,N, and ρ respectively be the drum revolution time, the number of sectors, and the sector utilization factor,

$$\text{mpft} = \frac{(T/2)}{1-\rho} + \frac{T}{N} \quad ,$$

where the first and the second terms respectively represent the mean access time, which is subject to a queuing delay, and the constant page transfer time.

of running processes (using the CPU), and the number (k) of page-wait processes (associated with the SM), where $i+j+k=q$ and $j \leq 1$, as shown in Figure 3-1. Therefore, each (i,j,k) tuple defines a state of the model, as given in Table 3-2. Noting that the holding time of each state is exponentially distributed, we see that the behavior of this model can be regarded as a continuous time Markov process, in fact, a simple birth and death process known as the machine repairman model (or the machine interference problem) [F1,J2], defined over these $q+1$ states of Table 3-2. All possible state transitions and the corresponding transition rates[1] are shown in the state transition diagram of Figure 3-2.

The probability π_i $(i=0,1,\ldots,q)$ that one finds the system in S_i at a randomly chosen instant after the system has been in operation for a long time, i.e., the steady-state probability π_i of a state S_i, can be obtained by solving the balance equation [F1,H2]

$$\alpha \pi_i = (q-i+1)\,\beta\, \pi_{i-1} \qquad 1 \leq i \leq q \qquad (3.3.3)$$

under the condition $\sum_{i=0}^{q} \pi_i = 1$. Thus,

---

[1] If the transition rate associated with an $S_i \rightarrow S_j$ transition is α, then this means that the conditional probability that this transition will occur in the next Δt, given that the present state is S_i, is given by $\alpha \Delta t$ for $j \neq i$ and suitably small Δt.

Table 3-2 State Table of the FCFS Single Processor Multiprogramming Model

State Name	Q_p i	CPU j	SM k
S_0	0	0	q
S_1	0	1	q-1
S_2	1	1	q-2
S_3	2	1	q-3
...	.....		
S_{q-1}	q-2	1	1
S_q	q-1	1	0

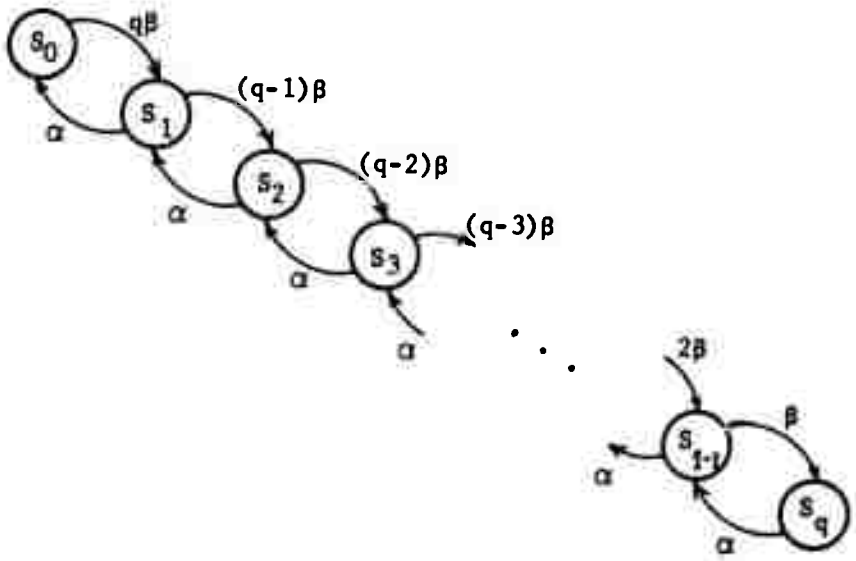


Figure 3-2 State Transition Diagram of the FCFS Single Processor Multiprogramming Model

$$\pi_i = \frac{\sigma^{q-i}/(q-i)!}{\sum_{j=0}^{q} (\sigma^j/j!)} \qquad 0 \le i \le q \qquad (3.3.4)$$

where $\sigma = \alpha/\beta$. In particular, the processor idle probability π_0 is given by

$$\pi_0 = \frac{\sigma^q/\sigma!}{\sum_{j=0}^{q} (\sigma^j/j!)} \qquad (3.3.5)$$

Therefore, the percentile throughput of this system is obtained from Eq. (3.2.7) as

$$\theta = (1-\pi_0) \cdot \frac{\bar{\tau}_{ul}}{(1+\delta)\bar{\tau}_{ul} + \bar{\tau}_{pl}} = (1-\pi_0)/\bar{\tau}_{ul} \qquad (3.3.6)$$

The corresponding system throughput can be readily obtained from Eq. (3.2.8).

3.3.3. Multiprogramming with Preemptive Priority Scheduling

We will briefly consider the system throughput of the same single processor computer system, assuming that a preemptive priority (PP) discipline like that of the Multics system [01] is used in scheduling page-ready eligible user processes. When this discipline is used, all the eligible user processes have their own distinct priorities upon which preemption decisions are solely based. Suppose that the super-

visor decides to allow a ready process to become eligible at a certain instant when there exist q' eligible user processes (see Figure 1-1). This entering process is immediately awarded a priority q' + 1 ($\leq$ q), the lowest priority (i.e., largest in value) among existing q' + 1 eligible user processes. Whenever a process becomes ineligible by completing the execution of the requested user job, the process loses its priority and accordingly changes of priority of other eligible user processes take place. Suppose that the leaving process has a priority q". Then, if the priority of a remaining process is lower than that of the leaving process (i.e., larger than q"), then the priority is heightened by one (i.e., decreased by one in value); otherwise, the priority is left unchanged. Thus, this algorithm gives a higher priority to an older eligible user process, maintaining a set of distinct priorities for the existing eligible user processes. The page-ready processes in Q_p are always ordered according to their distinct priorities and the process with the highest priority is scheduled for the processor's service when the processor is available. What is more, if the priority of this process is higher than that of the running process, this process is allowed to preempt the processor by suspending the progress of the running process as soon as possible. Therefore, an eligible process with a higher priority has a greater chance to use both the processor and the primary memory. Thus, an older eligible user process tends to own more pages in primary memory than newer eligible user processes, under a global page replacement algorithm. We assume again that the computer system is under a full load.

Instead of using the tbpf distribution of Eq. (3.3.1), we assume the following set of equations for the tbpf distributions of user processes under the PP discipline.

$$\left.\begin{aligned} f_1(t) &= \alpha_1 e^{-\alpha_1 t} \quad \text{for the highest priority process} \\ f_2(t) &= \alpha_2 e^{-\alpha_2 t} \quad \text{for the priority 2 process} \\ &\cdots \\ f_q(t) &= \alpha_q e^{-\alpha_q t} \quad \text{for the lowest priority process} \\ \text{where} \quad & \alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_q \quad \text{and} \quad t \geq 0. \end{aligned}\right\} \tag{3.3.7}$$

The fact that an older process, i.e., a higher priority process, owns a greater number of pages in primary memory is reflected by a longer mtbpf, $1/\alpha_i$, of a higher priority process. As for the pft distribution, we still assume Eq. (3.3.2).

What characterizes a state of this modified model is not just the number of processes in each of those three states of eligible user processes but the combination of priorities of processes in each of those three states. Then, it is again possible to model the behavior of this model by a continuous time Markov process. Let an (i,j,k) tuple represent a state of this model, where i is a (particular) combination of priorities of the page-ready processes, j is a priority of the running process, and k is a combination of priorities of the page-wait processes. Then, it can be easily proved that the total number (N_s) of these (Markovian) states of this model is given by

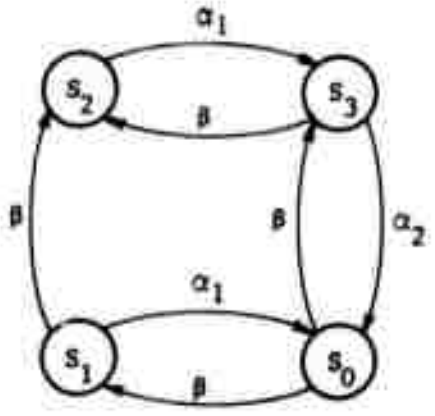
$$N_s = \sum_{k=0}^{q} \binom{q}{k} = 2^q \tag{3.3.8}$$

Unfortunately, a general expression for π_i, like Eq. (3.3.4) of the FCFS scheduling model, is too complicated to write out (although π_i can be easily evaluated numerically) and therefore we will simply give an illustrative example.

Example Case of $q = 2$

Because the supervisor allows only two eligible processes in primary memory there exist only priority 1 and 2 processes. Defining four possible states as shown below, we obtain a state transition diagram shown also below.

	Q_p	CPU	SM
S_0			(1,2)
S_1		(1)	(2)
S_2	(2)	(1)	
S_3		(2)	(1)



Letting $\underline{\pi} = (\pi_0, \pi_1, \pi_2, \pi_3)$ be the steady-state probabilities of states (S_0, S_1, S_2, S_3), these steady-state probabilities can be obtained by solving the balance equation

$$\underline{\pi} \cdot \underline{A} = \underline{0}$$

under the condition $\sum_{i=0}^{3} \pi_i = 1$, where $\underline{A}$ is the transition rate matrix [H2] and takes the following form for this example.

$$\underline{A} = \begin{pmatrix} -2\beta & \beta & 0 & \beta \\ \alpha_1 & -\alpha_1-\beta & \beta & 0 \\ 0 & 0 & -\alpha_1 & \alpha_1 \\ \alpha_2 & 0 & \beta & -\alpha_2-\beta \end{pmatrix}.$$

The solutions are:

$$\pi_0 = \frac{(\beta+\alpha_1)\alpha_1\alpha_2}{2\beta^3+3\beta^2\alpha_1+\beta^2\alpha_2+\beta\alpha_1^2+2\beta\alpha_1\alpha_2+\alpha_1^2\alpha_2}$$

$$\pi_1 = \frac{\beta\alpha_1\alpha_2}{2\beta^3+3\beta^2\alpha_1+\beta^2\alpha_2+\beta\alpha_1^2+2\beta\alpha_1\alpha_2+\alpha_1^2\alpha_2}$$

$$\pi_2 = \frac{\beta^2(2\beta+\alpha_1+\alpha_2)}{2\beta^3+3\beta^2\alpha_1+\beta^2\alpha_2+\beta\alpha_1^2+2\beta\alpha_1\alpha_2+\alpha_1^2\alpha_2}$$

$$\pi_3 = \frac{(2\beta+\alpha_1)\beta\alpha_1}{2\beta^3+3\beta^2\alpha_1+\beta^2\alpha_2+\beta\alpha_1^2+2\beta\alpha_1\alpha_2+\alpha_1^2\alpha_2}$$

The processor idle probability is given by π_0. Therefore, the percentile throughput can be obtained by inserting π_0 into Eq. (3.3.6) of the previous section. The priority 1 (higher priority) process uses the processor with a probability $\pi_1+\pi_2$ and the priority 2 (lower priority) process with a probability π_3. If $\alpha_1=\alpha_2=\alpha$, then we see that the π_0 obtained from the above equation agrees with the π_0 given by Eq. (3.3.5) of the previous model. This corresponds to the fact

that a preemption scheme does not improve the processor utilization if $\alpha_1 = \alpha_2 = \ldots = \alpha_q$, under the assumption of exponential distributions. Finally, a numerical example of processor time breakdown is shown in Table 3-3, assuming that $\alpha_1 = \alpha_2 = \alpha$. It is clearly seen that the priority 1 process uses the processor much more than the priority 2 process because of the preemptive priority scheduling discipline.

Table 3-3 Processor Time Breakdown for the PP Scheduling Model

Processor Time Breakdown	$\sigma = \alpha/\beta$ 0.5	1.0	1.5
Priority 1 Process	0.667	0.500	0.400
Priority 2 Process	0.256	0.300	0.290
Idle Time	0.077	0.200	0.310

3.4. Dual Processor System

The model of a single processor processing system under multiprogramming of Section 3.3 is now generalized to a dual processor multimemory processing system under multiprogramming. To specify the gross configuration of the system, we use a notation (q; m, n) where

(1) q is the degree of multiprogramming,

(2) m is the number of processors ($m \leq q$),

(3) n is the number of primary memory units, each of which can be accessed through its own memory controller,

as schematically shown in Figure 3-3. We will be concerned with the system throughput of the (q; 2, n) configuration system in Section 3.4.

Although the system of this configuration is generally expected to have a much larger computational capacity than a single processor system, this system will encounter new problems which may seriously limit its computational power. These problems are caused by multiprocessor interference such as memory cycle interference and data-base lockout. These interference sources effectively reduce the speed of the processing system. We will therefore develop a model of the (q; 2, n) configuration processing system and evaluate the throughput of this system. The effective loss of the system's computational capacity due to multi-processor interference, various system overheads, and the multiprogramming (processor) idle time (to be defined) will also be evaluated.

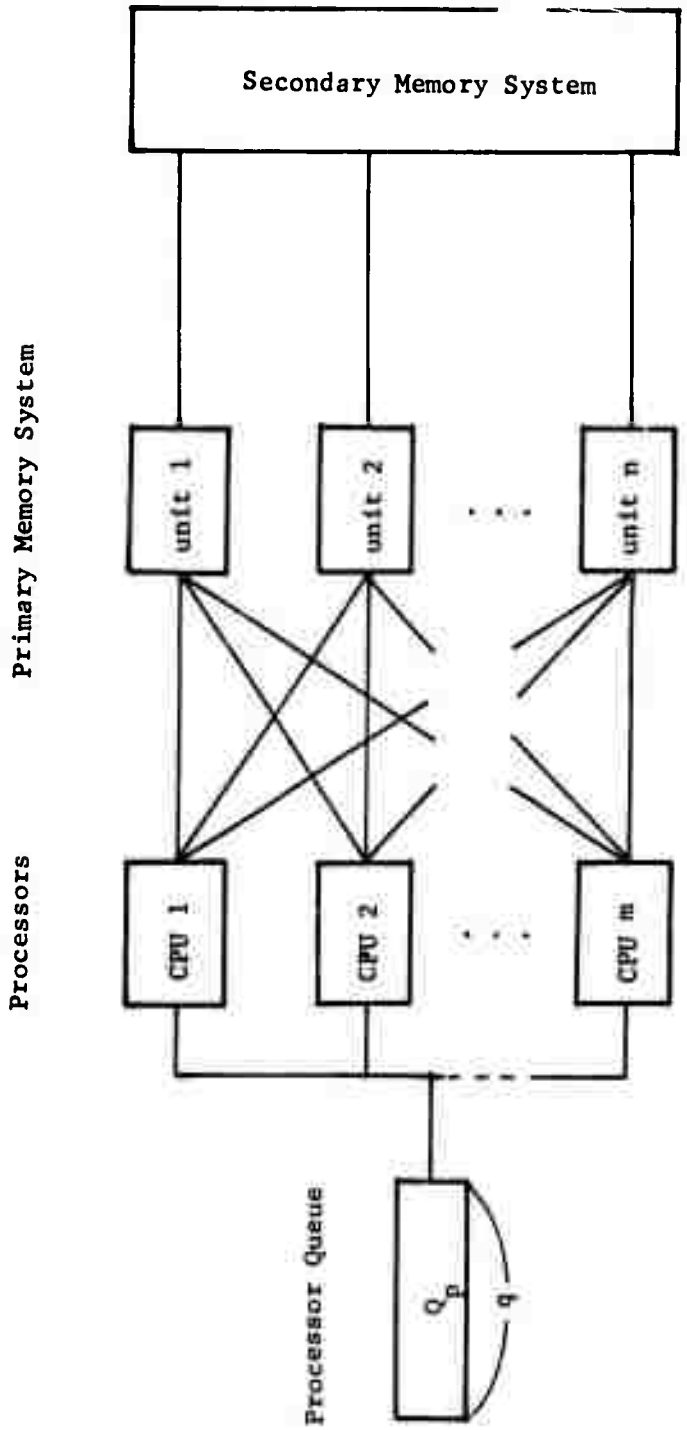


Figure 3-3 (q ; m , n) Configuration

3.4.1 Dual Processor Multiprogramming Model

A model of multiprogramming on the (q; 2, n) configuration system is based on the following assumptions:

(1) The virtual memory is paged. A given page of the primary memory system is found in a memory unit i ($1 \leq i \leq n$) with a probability 1/n (uniformness assumption).

(2) The length of in-page operation of a program (the length of time during which processor control remains within a particular page without making any reference to other pages) follows an exponential distribution with a mean of $\bar{t}_{in} = 1/\alpha_0$.

$$f_0(t) = \alpha_0 e^{-\alpha_0 t} \qquad t \geq 0 \qquad (3.4.1)$$

(3) A page fault occurs with a probability p (missing-page probability) when processor control is transferred to another page. The missing-page probability is common to all user programs.

(4) A missing-page will be fetched from the secondary memory system. The length of time required to complete a page fetch, or, the page fetch time (pft), follows an exponential distribution with a mean of $1/\beta$.

$$g(t) = \beta e^{-\beta t} \qquad t \geq 0 \qquad (3.4.2)$$

(5) The processor queue (Q_p) is common to two processors. Moreover, the processing system is assumed to have more than q jobs at all times to ensure the existence of q eligible processes under multiprogramming of degree q (full load assumption).

(6) If both processors direct their accesses to the same unit of primary memory at all times, the execution time of a program on each processor would be stretched by a factor of γ ($1 \leq \gamma \leq 2$), on the average[1]. γ is called the memory (cycle) interference coefficient. We say that a memory unit has no interference if $\gamma = 1$ and that a memory unit has a complete interference if $\gamma = 2$.

First, we will prove that tbpf approximately follows an exponential distribution under these assumtions. Noting that hbpf consists of successive in-page operations (see Section 2.4.2), we see that

$$f_1(t) \equiv \text{Prob}\{\ \text{hbpf} = t\ \}$$

$$= \sum_{i=1}^{\infty} \left[\text{Prob}\left\{ \begin{matrix} \text{i successive in-} \\ \text{page operations} \\ \text{before a page fault} \end{matrix} \right\} \cdot \text{Prob}\left\{ \begin{matrix} \text{total length of} \\ \text{i in-page operations} \\ = t \end{matrix} \right\} \right] \tag{3.4.3}$$

The probability that i in-page operations take place before a page fault occurs is given by a geometric distribution with a parameter of p, as is clear from the assumption (3). The sum of i exponentially distributed (identical) variables is known to follow an Erlang distribution of phase i, i.e., the i-fold convolution of identical exponential distributions [F1]. Therefore, Eq. (3.4.3) becomes

---

[1] The value of the effective slow-down factor due to memory cycle interference (K_m) for a given configuration will be calculated using the value of γ.

$$f_1(t) = \sum_{i=1}^{\infty} \left[\{(1-p)^{i-1} p\} \cdot \left\{ \frac{\alpha_0^i t^{i-1} e^{-\alpha_0 t}}{(i-1)!} \right\} \right]$$

$$= \alpha_0 p\, e^{-\alpha_0 t} \cdot \sum_{i=0}^{\infty} \frac{\{\alpha_0 t(1-p)\}^i}{i!}$$

$$= \alpha_0 p\, e^{-\alpha_0 t} \cdot e^{\alpha_0 t(1-p)}$$

$$= \alpha_0 p\, e^{-\alpha_0 p t} \qquad (3.4.4)$$

Thus, hbpf has been proved to follow an exponential distribution of a mean of $1/\alpha_0 p$. This means that the sum of hbpf (t_u) and the miscellaneous overhead time (t_m) approximately follows an exponential distribution with a mean of $(1+\delta)/\alpha_0 p$. Noting that the paging overhead time is considerably shorter than the sum of hbpf and t_m, we can expect that tbpf roughly follows

$$\left.\begin{aligned} & f(t) = \alpha\, e^{-\alpha t} \qquad t \geq 0 \\ & \text{where} \\ & \text{mtbpf}_2 = \frac{1}{\alpha} = (1+\delta)\bar{t}_{u1} + K_\ell \bar{t}_{p1} \simeq \frac{1+\delta}{\alpha_0 p} \equiv \frac{1}{\alpha_1 p} \end{aligned}\right\} \qquad (3.4.5)$$

Thus, we have seen that the model analyzed in Section 3.3 can be regarded as a special case of the model developed in this section (see Eq. (3.3.1)); the previous model actually corresponds to a (q; 1, n) configuration system with the above formulation.

Under these assumptions, each eligible user process cycles around the three states, i.e., the page-ready state, the running state, and

the page-wait state, as shown in Figure 3-4. It should be noted that if two processors happen to use the same primary memory unit for which $\gamma \neq 1$ then the rate $\alpha_1 \equiv \alpha_0/(1+\delta)$ of Figure 3-4 must be replaced by α_1/γ. It is important to note that non-resident programs (e.g., user programs) and resident programs (e.g., fault handlers) are treated as if they are equally susceptible of page faults in this formulation. This formulation was adopted for the sake of simplicity of the analysis. Finally, we will finish this section by defining some terminology to be used in the following analysis.

Multiprogramming idle time: the processor time lost when processors do not have executable eligible user jobs, i.e., when at least q - 1 jobs are simultaneously in the page-wait state.

Memory interference idle time: the processor time lost due to the unavailability of memory cycles when two processors simultaneously access the same unit of primary memory (memory cycle interference).

Data-base lockout idle time: the processor time lost due to the unavailability of certain shared writable data-bases (data-base lockout).

Besides these various idle times, we lose a non-negligible fraction of the system's computational capacity because of paging overhead and miscellaneous overhead of the supervisor.

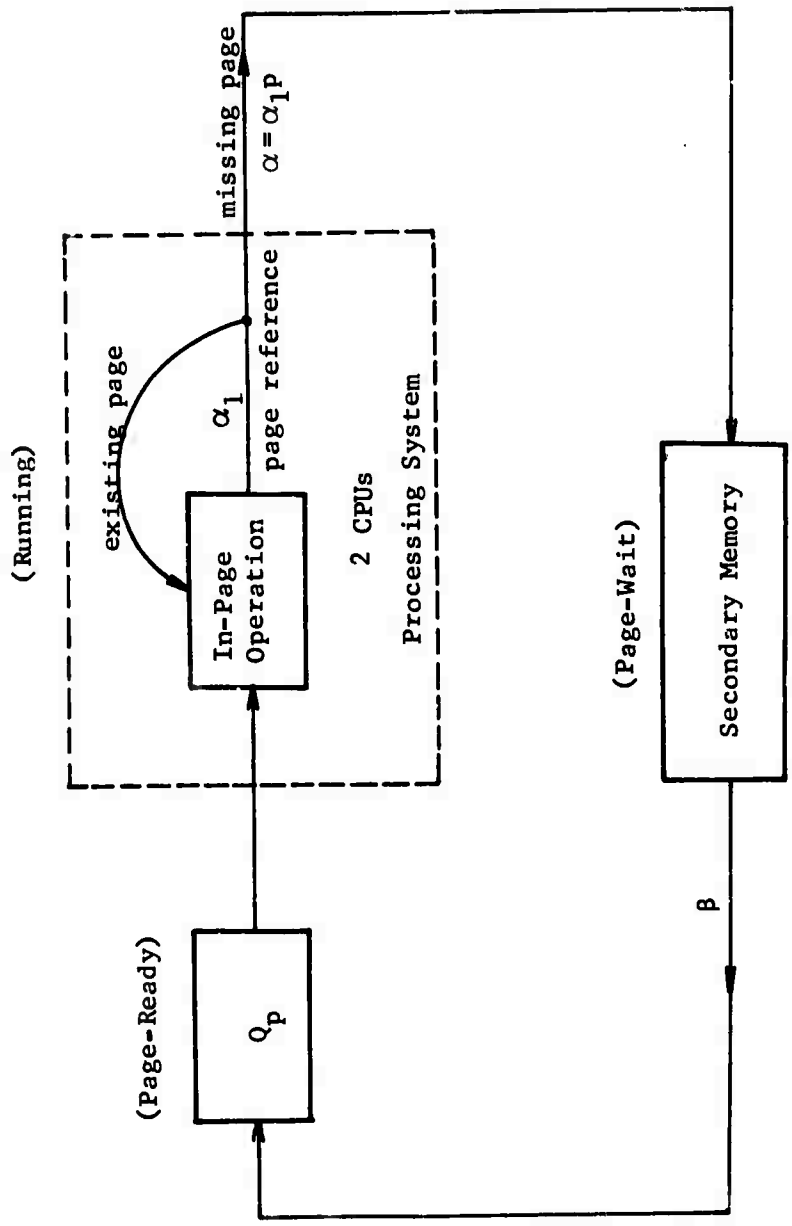


Figure 3-4 Behavior of an Eligible Process ($\gamma = 1$)

3.4.2. Multiprogramming with FCFS Scheduling

We will now evaluate the system throughput of the (q; 2, n) configuration system as well as the multiprogramming idle time, the memory interference idle time, the date-base lockout idle time, and other capacities wasted as various system overheads, under the assumptions stated in the previous section.

In analyzing the behavior of the (q; 2, n) configuration system, we should first note that a state of the system is characterized by the number (i) of page-ready processes (waiting in Q_p), the number (j) of running processes (using the CPUs) the number (k) of page-wait processes (associated with the SM), and a variable (s) which indicates the existence of memory cycle interference (s = 1 if interference exists and s = 0 otherwise). Then, it follows that each (i,j,k,s) tuple defines a state $\left({}_sS_j^k\right)$ of the system, as shown in Table 3-4. The number of states thus defined is found to be 2q. We see that these states are mutually related by the transitions with transition rates indicated in Figure 3-5, and therefore the behavior of the system can be modeled by a continuous time Markov process defined over these states.

Let ${}_s\pi_j^k$ be the steady-state probability of ${}_sS_j^k$. Then, the steady-state probabilities $\underline{\pi}$ can be obtained by solving the balance equation $\underline{\pi} \cdot \underline{A} = \underline{0}$, i.e., the following set of simultaneous equations given by Eq. (3.4.6) through Eq. (3.4.14).

Table 3-4 State Table of the FCFS Dual Processor Multiprogramming Model

state name	Q_p	CPU	SM	Memory Interference
	i	j	k	s
${}_0S_0^q$	0	0	q	0
${}_0S_1^{q-1}$	0	1	q-1	0
${}_0S_2^{q-2}$	0	2	q-2	0
${}_1S_2^{q-2}$	0	2	q-2	1
${}_0S_2^{q-3}$	1	2	q-3	0
${}_1S_2^{q-3}$	1	2	q-3	1
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
${}_0S_2^1$	q-3	2	1	0
${}_1S_2^1$	q-3	2	1	1
${}_0S_2^0$	q-2	2	0	0
${}_1S_2^0$	q-2	2	0	1

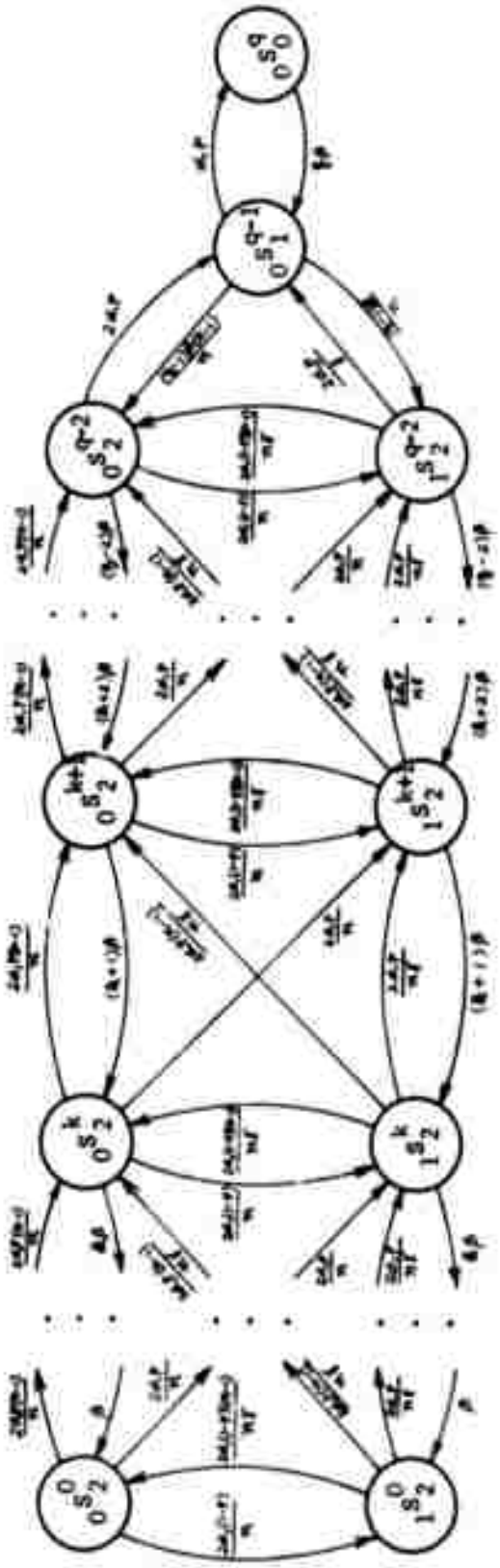


Figure 3-5 State Transition Diagram of the FCFS Dual Processor Multiprogramming Model

$$\frac{2\alpha_1 p(n-1)}{n}({}_0\pi_2^{k-1})+\frac{2\alpha_1 p(n-1)}{n\gamma}({}_1\pi_2^{k-1})+\frac{2\alpha_1(1-p)(n-1)}{n\gamma}({}_1\pi_2^{k})+(k+1)\beta({}_0\pi_2^{k+1})$$

$$=(k\beta+\frac{2\alpha_1(1-p)}{n}+\frac{2\alpha_1 p}{n}+\frac{2\alpha_1 p(n-1)}{n})({}_0\pi_2^{k}) \qquad 1\le k\le q-3 \qquad (3.4.6)$$

$$\frac{2\alpha_1 p}{n\gamma}({}_1\pi_2^{k-1})+\frac{2\alpha_1 p}{n}({}_0\pi_2^{k-1})+\frac{2\alpha_1(1-p)}{n}({}_0\pi_2^{k})+(k+1)\beta({}_1\pi_2^{k+1})$$

$$=(k\beta+\frac{2\alpha_1(1-p)(n-1)}{n\gamma}+\frac{2\alpha_1 p(n-1)}{n\gamma}+\frac{2\alpha_1 p}{n\gamma})({}_1\pi_2^{k}) \qquad 1\le k\le q-3 \qquad (3.4.7)$$

Left Boundary Conditions

$$\frac{2\alpha_1(1-p)(n-1)}{n\gamma}({}_1\pi_2^{0})+\beta({}_0\pi_2^{1})=(\frac{2\alpha_1(1-p)}{n}+\frac{2\alpha_1 p}{n}+\frac{2\alpha_1 p(n-1)}{n})({}_0\pi_2^{0}) \qquad (3.4.8)$$

$$\frac{2\alpha_1(1-p)}{n}({}_0\pi_2^{0})+\beta({}_1\pi_2^{1})=(\frac{2\alpha_1(1-p)(n-1)}{n\gamma}+\frac{2\alpha_1 p(n-1)}{n\gamma}+\frac{2\alpha_1 p}{n\gamma})({}_1\pi_2^{0}) \qquad (3.4.9)$$

Right Boundary Conditions

$$\frac{2\alpha_1 p(n-1)}{n}({}_0\pi_2^{q-3})+\frac{2\alpha_1 p(n-1)}{n\gamma}({}_1\pi_2^{q-3})+\frac{2\alpha_1(1-p)(n-1)}{n\gamma}({}_1\pi_2^{q-2})+\frac{(q-1)(n-1)\beta}{n}({}_0\pi_1^{q-1})$$

$$=((q-2)\beta+\frac{2\alpha_1(1-p)}{n}+2\alpha_1 p)({}_0\pi_2^{q-2}) \qquad (3.4.10)$$

$$\frac{2\alpha_1 p}{n\gamma}({}_1\pi_2^{q-3})+\frac{2\alpha_1 p}{n}({}_0\pi_2^{q-3})+\frac{2\alpha_1(1-p)}{n}({}_0\pi_2^{q-2})+\frac{(q-1)\beta}{n}({}_0\pi_1^{q-1})$$

$$=((q-2)\beta+\frac{2\alpha_1(1-p)(n-1)}{n\gamma}+\frac{2\alpha_1 p}{\gamma})({}_1\pi_2^{q-2}) \qquad (3.4.11)$$

$$2\alpha_1 p({}_0\pi_2^{q-2})+\frac{2\alpha_1 p}{\gamma}({}_1\pi_2^{q-2})+q\beta({}_0\pi_0^{q})=(\frac{(q-1)(n-1)\beta}{n}+\frac{(q-1)\beta}{n}+\alpha_1 p)({}_0\pi_1^{q-1}) \qquad (3.4.12)$$

$$\alpha_1 p\,({}_0\pi_1^{q-1}) = q\beta\,({}_0\pi_0^{q}) \qquad (3.4.13)$$

Normalization Conditon

$$({}_0\pi_0^q) + ({}_0\pi_1^{q-1}) + \sum_{k=0}^{q-2} ({}_0\pi_2^k) + \sum_{k=0}^{q-2} ({}_1\pi_2^k) = 1 \qquad (3.4.14)$$

Because there are 2q independent equations in 2q unknown variables, the ${}_s\pi_j^k$'s can be uniquely determined.

Now we are ready to express several interesting performance measures of the system's computational capacity in terms of the steady-state probabilities obtained above. It is assumed that the system's computational capacity (2C), measured in the total number of instructions that can be performed by the system's two processors under the ideal condition, breaks down into the capacity (C_{mi}) wasted as multiprogramming idle time, the capacity (C_{ii}) wasted as memory interference idle time, the capacity (C_p) wasted as the paging overhead including the capacity (C_{di}) wasted as data-base lockout idle time, the capacity (C_m) wasted as the miscellaneous overhead, and the rest, i.e., the capacity (C_u) used as users' useful work. These capacities are evaluated as follows:

Capacity wasted as multiprogramming idle time

$$\begin{aligned} C_{mi} &= 2C\ ({}_0\pi_0^q) + C\ ({}_0\pi_1^{q-1}) \\ &= \{2({}_0\pi_0^q) + ({}_0\pi_1^{q-1})\} \cdot C \end{aligned} \qquad (3.4.15)$$

Capacity wasted as memory interference idle time

$$C_{ii} = \{\sum_{k=0}^{q-2} ({}_1\pi_2^k)\} \cdot (1-\frac{1}{\gamma}) \cdot 2C$$

$$= \frac{\gamma-1}{\gamma} \cdot \{\sum_{k=0}^{q-2} ({}_1\pi_2^k)\} \cdot 2C \qquad (3.4.16)$$

Capacity wasted as the paging overhead

$$C_p = (2C - C_{mi} - C_{ii}) \cdot \frac{\bar{t}_{p2}}{\text{mtbpf (2 CPUs)}}$$

$$= (2C - C_{mi} - C_{ii}) \cdot \frac{K_\ell \bar{t}_{p1}}{\text{mtbpf}_2}$$

$$= (2C - C_{mi} - C_{ii}) \cdot \alpha K_\ell \bar{t}_{p1} \qquad (3.4.17)$$

where $\frac{1}{\alpha} = \text{mtbpf}_2 = (1+\delta)\,\bar{t}_{u1} + K_\ell \bar{t}_{p1} = \frac{1}{\alpha_1 p}$

$$C_{di} = (K_\ell - 1) \cdot C_p \qquad (3.4.18)$$

Capacity wasted as the miscellaneous overhead

$$C_m = (2C - C_{mi} - C_{ii}) \cdot \frac{\bar{t}_{m2}}{\text{mtbpf (2 CPUs)}}$$

$$= (2C - C_{mi} - C_{ii}) \cdot \frac{\bar{t}_{m1}}{\text{mtbpf}_2}$$

$$= (2C - C_{mi} - C_{ii}) \cdot \alpha\delta\, \bar{t}_{u1} \qquad (3.4.19)$$

Capacity used as users' useful work

$$C_u = (2C - C_{mi} - C_{ii}) \cdot \frac{\bar{t}_{u2}}{\text{mtbpf}\,(2\ \text{CPUs})}$$

$$= (2C - C_{mi} - C_{ii}) \cdot \frac{\bar{t}_{u1}}{\text{mtbpf}_2}$$

$$= (2C - C_{mi} - C_{ii}) \cdot \alpha\, \bar{t}_{u1} \qquad (3.4.20)$$

Memory interference slow-down factor

$$K_m = \frac{\left[\sum_{k=0}^{q-2} \{ ({}_0\pi_2^k) + \gamma ({}_1\pi_2^k) \} \right] + \frac{1}{2} ({}_0\pi_1^{q-1})}{\left[\sum_{k=0}^{q-2} \{ ({}_0\pi_2^k) + ({}_1\pi_2^k) \} \right] + \frac{1}{2} ({}_0\pi_1^{q-1})} \qquad (3.4.21)$$

Percentile throughput

$$\theta = u \cdot \frac{\bar{t}_{u2}}{\text{mtbpf}\,(2\ \text{CPUs})}$$

$$= \frac{2C - C_{mi} - C_{ii}}{2C} \cdot \frac{\bar{t}_{u1}}{\text{mtbpf}_2}$$

$$= \frac{C_u}{2C} \qquad (3.4.22)$$

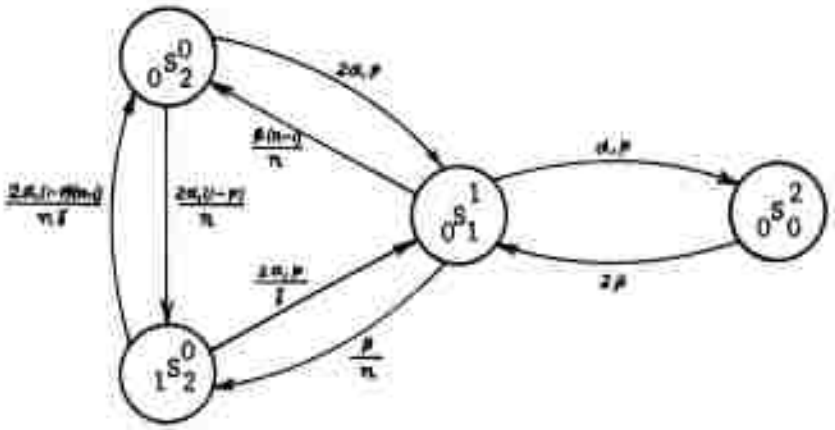
It is seen from Eq. (3.4.22) that the percentile throughput of the (q;2,n) configuration system is expressed as a function of user program characteristics $(\bar{t}_{u1}, p)$, the system overhead structure $(\bar{t}_{p1}, \delta, K_\ell)$, and hardware characteristics (n, β, γ). The system throughput $\underline{\theta}$ of this configuration can be obtained by substituting Eq. (3.4.22) into Eq. (3.2.8).

An expression for ${}_s\pi_j^k$ is generally too complicated to write out and therefore the expressions given by Eq. (3.4.15) through Eq. (3.4.22) can only be numerically evaluated (this will be done in Chapter 5),

except for some simple cases. However, one of those simple cases for which ${}_s\pi_j^k$ is not too complicated may be worth being explicitly analyzed as an illustrative example.

Example A (2;2,n) Configuration System

The state transition diagram of Figure 3-5 now reduces to the four state transition diagram given below.



The equations given by Eq. (3.4.6) through Eq. (3.4.14) correspondingly reduce to:

$$\frac{2\alpha_1(1-p)(n-1)}{n\gamma}\,({}_1\pi_2^0) + \frac{\beta(n-1)}{n}\,({}_0\pi_1^1) = \left\{\frac{2\alpha_1(1-p)}{n} + 2\alpha_1 p\right\}({}_0\pi_2^0)$$

$$\frac{2\alpha_1(1-p)}{n}\,({}_0\pi_2^0) + \frac{\beta}{n}\,({}_0\pi_1^1) = \left\{\frac{2\alpha_1(1-p)(n-1)}{n\gamma} + \frac{2\alpha_1 p}{\gamma}\right\}({}_1\pi_2^0)$$

$$2\alpha_1 p\,({}_0\pi_2^0) + \frac{2\alpha_1 p}{\gamma}\,({}_1\pi_2^0) + 2\beta\,({}_0\pi_0^2) = (\beta + \alpha_1 p)\,({}_0\pi_1^1)$$

$$\alpha_1 p\,({}_0\pi_1^1) = 2\beta\,({}_0\pi_0^2)$$

$$({}_0\pi_2^0) + ({}_1\pi_2^0) + ({}_0\pi_1^1) + ({}_0\pi_0^2) = 1$$

Solving these equations, we get

$$0\pi_0^2 = \frac{n\sigma^2}{n(1+\sigma)^2+\gamma-1}$$

$$0\pi_1^1 = \frac{2n\sigma}{n(1+\sigma)^2+\gamma-1}$$

$$0\pi_2^0 = \frac{n-1}{n(1+\sigma)^2+\gamma-1}$$

$$1\pi_2^0 = \frac{\gamma}{n(1+\sigma)^2+\gamma-1}$$

where $\sigma = \alpha_1 p/\beta = \alpha/\beta$.

Thus, from Eqs. (3.4.15) and (3.4.16), we obtain the following result concerning the capacities wasted as multiprogramming idle time and as memory interference idle time.

$$C_{mi} = \frac{n\sigma(1+\sigma)}{n(1+\sigma)^2+\gamma-1}\cdot 2C = \frac{\sigma(1+\sigma)}{(1+\sigma)^2+(\frac{\gamma-1}{n})}\cdot 2C$$

$$C_{ii} = \frac{\gamma-1}{n(1+\sigma)^2+\gamma-1}\cdot 2C = \frac{(\frac{\gamma-1}{n})}{(1+\sigma)^2+(\frac{\gamma-1}{n})}\cdot 2C$$

The system's computational capacities used as various overheads (C_p, C_m) and as users' useful work (C_u) respectively follow from Eqs. (3.4.17), (3.4.19), and (3.4.20). The memory interference slow-down factor is obtained from Eq. (3.4.21).

$$K_m = \frac{(n-1)+\gamma^2+n\sigma}{(n-1)+\gamma+n\sigma}$$

Finally, the percentile throughput is obtained from Eq. (3.4.22) as

$$\theta = \frac{1+\sigma}{(1+\sigma)^2 + (\frac{\gamma-1}{n})} \cdot \alpha \bar{t}_{u1}$$

It should be noted that as $(\gamma - 1)/n$ approaches zero (either $n \to \infty$, i.e., an infinite number of memory units, or $\gamma \to 1$, i.e., no memory cycle interference) the effect of memory cycle interference disappears. Furthermore, the multiprogramming (processor) idle probability of this dual processor system under this limit is $C_{mi}/2C = \sigma/(1+\sigma)$. Although the multiprogramming (processor) idle probability of a (1; 1, n) configuration system was also found to be $\sigma/(1+\sigma)$ from Eq. (3.3.5) of the previous section, the split system (q; 1, n) generally gives a larger multiprogramming idle probability than the (combined) dual processor (2q; 2, n) configuration system, under the no memory interference condition mentioned above. Needless to say, this does not necessarily mean that the (2q; 2, n) system is more efficient than the (q; 2, n) system; the comparison must be made in terms of percentile throughput rather than multiprogramming idle probability.

3.5. Some Extensions

We have been concerned with the throughput of the multi-processor multi-memory processing system under multiprogramming. In particular, we have presented an analytical method to evaluate the throughput of a single processor and a dual processor systems under multiprogramming. In this section, we will give brief remarks on some extensions of the results obtained in this chapter.

The study of the system throughput of this chapter has been mainly confined to the FCFS scheduling model; a brief investigation of the PP (preemptive priority) scheduling model for a single processor system left the value of the mtbpf of each process with a distinct priority unspecified. It is perhaps possible to determine the values of these mtbpf's and then evaluate the throughput of such a model, by assuming that the processor time breakdown among eligible user processes is proportional to the primary memory space usage breakdown among them. Then, it would be possible to evaluate the performance gain of the PP scheduling model over the FCFS scheduling model quantitatively, using the macroscopic paging performance model of Section 2.5. As for the dual processor system, even the analysis of the FCFS discipline model was rather complicated. Inclusion of the PP scheduling would increase the number of Markovian states of this model and necessarily complicate its analysis. Therefore, no attempt to study the dual processor PP scheduling model is made in this thesis.

Processing systems involving more than three processors have not been considered. In order to examine the complexity of such systems,

let us consider a general (q; m ,n) configuration system. Speaking of memory cycle interference alone, there exist many kinds of memory cycle interferences on this system, such as interference between two processors, that among three processors,..., that among m processors. Therefore, we must define a different γ (memory interference coefficient) for each kind of memory cycle interference. Such a complicated formulation of the problem, however, tends to reduce the mathematical tractability, and therefore it seems necessary to consider the effect of memory cycle interference indirectly much like the effect of data-base lockout on the (q; 2, n) configuration system was considered in this chapter.

Finally, let us briefly consider the optimization of the multiprocessor multi-memory processing system under multiprogramming, with respect to the degree of multiprogramming. The method of evaluating the system throughput developed in this chapter enables us to investigate the effect of the degree (q) of multiprogramming upon the system throughput of a given hardware configuration. If the system allows only a few eligible user processes the percentile throughput would be small because of relatively large multiprogramming idle time. On the contrary, if the system allows too many eligible user processes the percentile throughput would be again small because of relatively frequent paging overhead execution. Therefore, we expect a modal curve for the percentile throughput like the one shown in Figure 3-6; the percentile throughput is maximized when the degree of multiprogramming has a certain value (q*). This optimization will be carried out numerically in Chapter 5, and therefore detailes will not be given in this section.

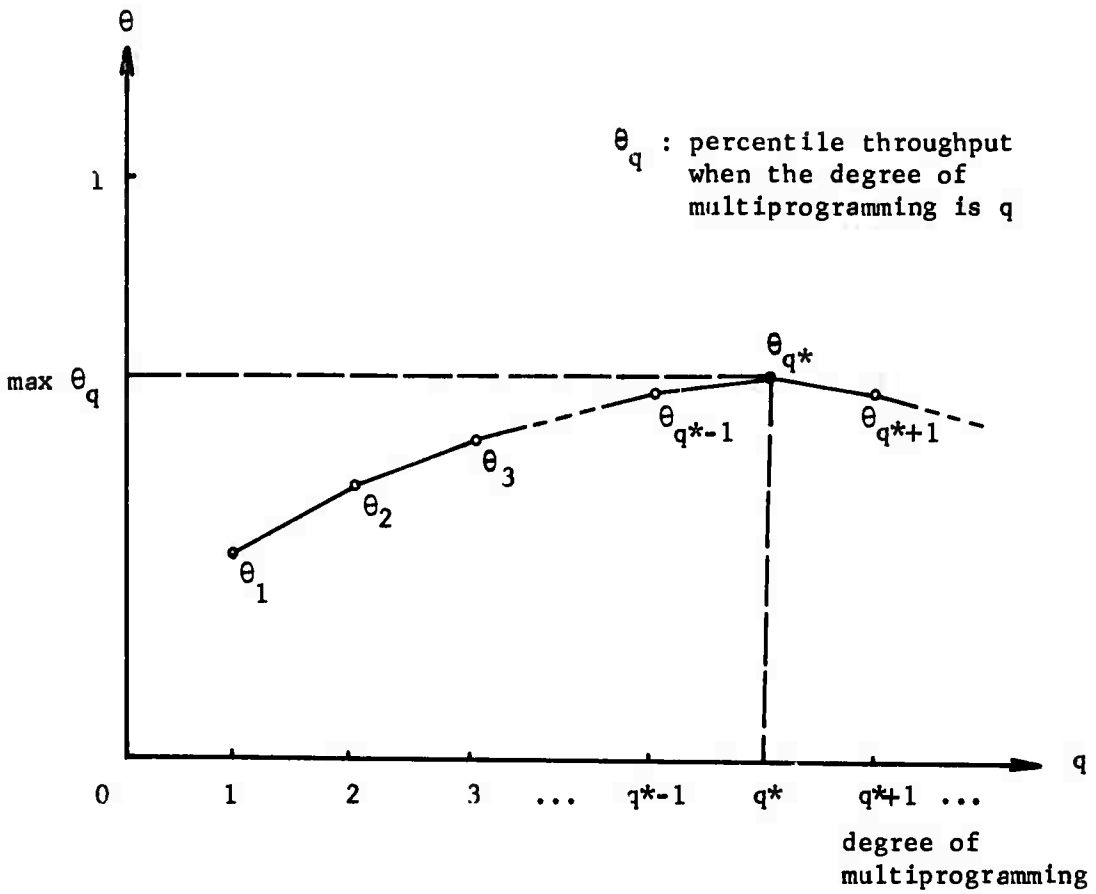


Figure 3-6 Percentile Throughput as a Function of the Degree of Multiprogramming

CHAPTER 4

RESPONSE TIME ANALYSIS

4.1. Introduction

Now that a method to evaluate the throughput (or the percentile throughput) of a multiprogrammed virtual-memory computer system of a given multi-processor multi-memory configuration has been developed, we shall proceed in this chapter to evaluate the response time experienced by interactive users of such a time-shared computer system. From a system manager's viewpoint efficient utilization of system resources, or, more concretely, the maximization of the system throughput is most important. On the other hand, service of good quality such as fast system response or high system reliability is most desirable from a user's viewpoint. As will be described in this chapter, the system response time and the system throughput are closely related. In fact, the response time analysis of this chapter is firmly based on the system's percentile throughput that was evaluated by the throughput

analysis of Chapter 3.

It will be seen that the maximization of the system throughput generally leads to the minimization of the system response time. Therefore, the computer system under study is assumed to be optimized with respect to the degree (q) of multiprogramming; i.e., the system is under multiprogramming of degree q* (q= q*), where the percentile throughput (or system throughput) of the system of a given configuration is maximized by the throughput analysis of Chapter 3 when q is equal to q* (see Figure 3-6). It is clear that if interactive users impose a full load on the system the percentile throughput of the system (or system throughput) is independent of the actual number of users (because of the load leveling of the multiprogramming mechanism).

On the other hand, the response time is clearly an increasing function of the number of users. Therefore, one may ask how many interactive users can be supported by such an optimized computer system without discouraging them with excessively slow system responses.

Scherr [S4] considered the problem of analyzing the performance of a CTSS-like computer system and proposed a simple queuing model to predict the average response time. Because the computer system under study in this thesis is much more sophisticated, the model must reflect those sophisticated features of the system such as paging, multiprogramming, etc. Therefore, we will extend the queuing model used in Scherr's analysis in the following two directions. First, his model (developed for uniprogrammed non-paging computer systems like the CTSS system [C9]) will be generalized to multiprogrammed paging computer

systems. Second, in view of the fact that interactive terminal users are more concerned with something like the worst response time out of ten trials rather than the average response time, the analysis to be given in this chapter will derive a probability distribution of response times. (Scherr derived only the average response time of a simpler system.) Thus, we can evaluate the moment of any order (average, variance, etc) of response times, and, perhaps more importantly, we can derive the percentile system response time, i.e., the time limit which guarantees that a certain proportion of response times is shorter than this limit (e.g., the 90 percentile system response time).

The analysis of this chapter makes it possible to express the distribution of the response time of a multiprogrammed time-shared virtual-memory computer system of a given configuration, as a function of various system parameters such as

(1) the number of processors

(2) the percentile throughput θ of the processing system

(3) the number of (interactive) users

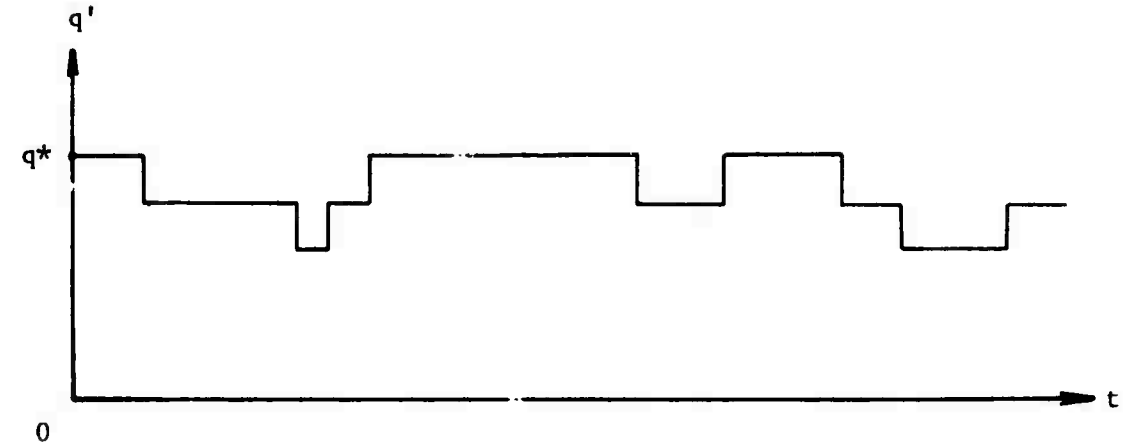
(4) think time of each user

(5) execution time required by each user's job

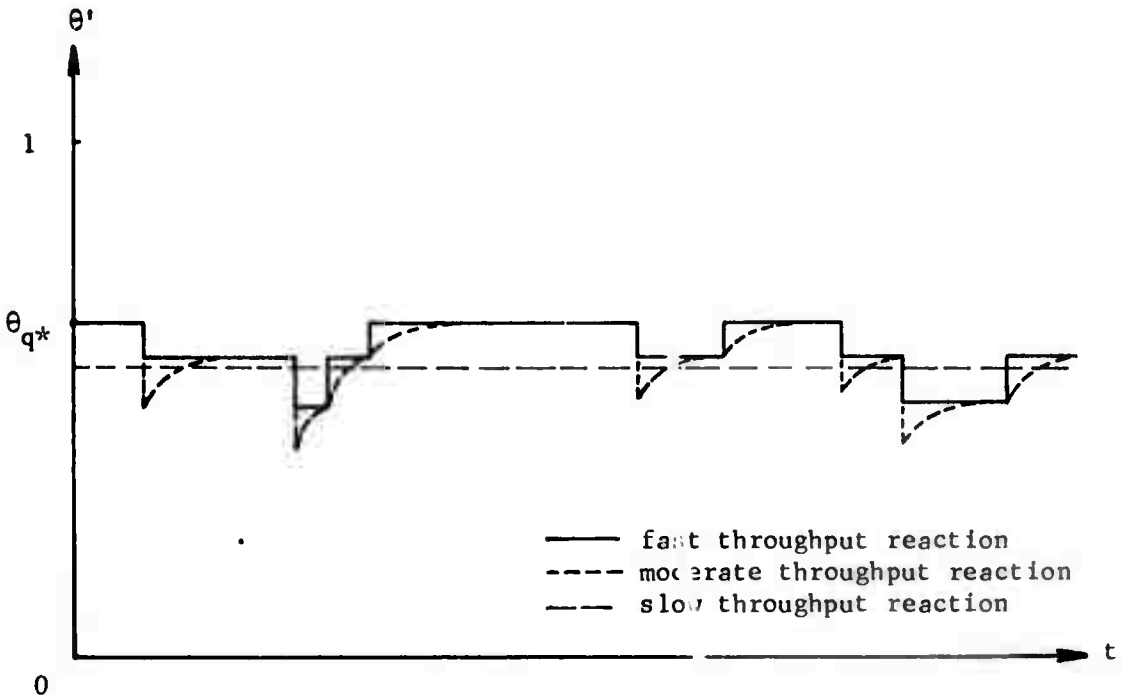
This implies that it now becomes possible to determine the number of interactive terminal users that can be supported by the optimized computer system of a given hardware configuration with the assurance that, for example, the 90 percentile system response time is shorter than five seconds.

4.2. Total System Model

Let us consider an m ($1 \leq m \leq \infty$) processor multiprogrammed virtual-memory computer system optimized with respect to the degree of multiprogramming, with N interactive terminal users. When this system is in operation, the existence of N interactive users however does not assure the existence of q* (q*= optimum degree of multiprogramming) jobs that can be multiprogrammed on the system. The actual number (q') of jobs (user processes) under multiprogramming may become smaller than q* from time to time if the total number (N) of interactive users is not large enough to impose a full load upon the system, as shown in Figure 4-1 (a). Correspondingly, the actual percentile throughput (θ') should become smaller than θ_{q*} (θ_{q*}= optimized percentile throughput) at least from time to time, under the same loading condition. However, we do not know exactly how θ' reacts to a sudden change of q'. If θ' can almost instantaneously react to a change of q' (the case of fast throughput reaction), then θ' as a function of time appears like a step-wise function for which the height of each step is simply given by the steady-state solution for θ_q (the percentile throughput when there exist q eligible user processes) obtained in Chapter 3, as shown by the solid line in Figure 4-1 (b); otherwise, θ' should show a humping curve as shown by the broken line beside the step-wise solid line in the same figure. If the reaction of θ' is very slow (the case of slow throughput reaction), θ' may be reasonably approximated by a constant function [S6], as shown by the horizontal line in Figure 4-1 (b); the value (height) of this function may be given by the average of the percentile through-



(a) Number of User Processes under Multiprogramming



(b) Percentile Throug put

Figure 4-1 Sample Behavior of a Typical Computer System

put of the system in operation, which we call the effective percentile throughput $\theta(N)$. That is to say,

$$\theta(N) = \left\{\left(\sum_{i=1}^{q^*-1} \bar{\pi}_i \theta_i\right) + \left(\sum_{i=q^*}^{N} \bar{\pi}_i \theta_{q^*}\right)\right\} \Big/ (1-\bar{\pi}_0) \qquad (4.2.1)$$

where $\bar{\pi}_i$ is the probability that i user jobs are either waiting or being serviced on the actual processing system, $\bar{\pi}_0$ is the probability that no jobs are being serviced (i.e., the system is completely idle), and θ_i $(i=1,2,\ldots,q^*,\ldots)$ is the percentile throughput of the system under multiprogramming of degree $q=i$. It should be noted that $\bar{\pi}_i$ is a function of N (the number of users) as well as of other system parameters. This means that the effective percentile throughput of the system under multiprogramming is dependent on the number of users unlike that of the system under uniprogramming. ($\theta(N)$ will be often denoted simply as θ hereafter.)

We shall approximate the behavior of a multi-processor processing system of such an actual computer system with that of a hypothetical multi-processor processing system of Figure 4-2, for which the system's computational capacity can be fully utilized for users' useful work, but its execution speed is θ $(0 \leq \theta \leq 1)$ times as slow as the actual processing system. We assume that each processor of the hypothetical system services user jobs one by one up to completion, under the FCFS scheduling discipline. A discussion on the possible modeling errors will be collectively given later in this chapter and therefore that is not our current concern.

Finally, in order to characterize the behavior of interactive

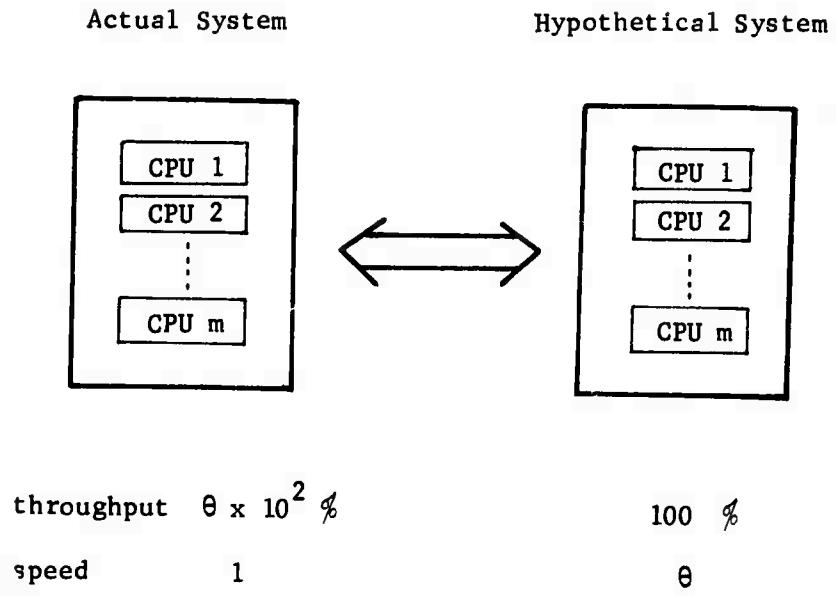


Figure 4-2 Tradeoff of Execution Speed and Percentile Throughput

terminal users, we assume the following:

(1) A user's think time (T_t), i.e., the length of time required by an interactive user to decide and request the next job from a terminal, independently follows an exponential distribution with a mean of $\overline{T}_t = 1/\mu$.

$$f(t) = \mu e^{-\mu t} \qquad t \geq 0 \qquad (4.2.2)$$

(2) The execution time (T_e) of each user's job (excluding all the system overhead times to be added) independently follows an exponential distribution with a mean of $\overline{T}_e = 1/\lambda_0$, on the actual multiprocessor computer system.

$$g_0(t) = \lambda_0 e^{-\lambda_0 t} \qquad t \geq 0 \qquad (4.2.3)$$

Available measurement results of actual computer systems[e.g., S4] indicate that both of these distributions are roughly exponential[1]. Therefore, these two assumptions seem to be fairly reasonable. The second assumption obviously implies that the execution time of each user's job, when executed on the hypothetical system, follows an exponential distribution with a mean of $\overline{T}_e/\theta = 1/\theta\lambda_0 \equiv 1/\lambda$. That is to say,

$$\left.\begin{aligned} g(t) &= \lambda e^{-\lambda t} \qquad t \geq 0 \\ \text{where} \quad \lambda &= \theta \lambda_0 \end{aligned}\right\} \qquad (4.2.4)$$

---

[1]The effect of non-exponential distributions upon the response time characteristics of a similar model is considered by Jaiswal[J2] and D'Avanzo[D4].

Thus, the behavior of an actual multiprogrammed virtual-memory computer system depicted in Figure 1-1 is now approximated with that of the hypothetical system, called the total system model, like the one shown in Figure 4-3.

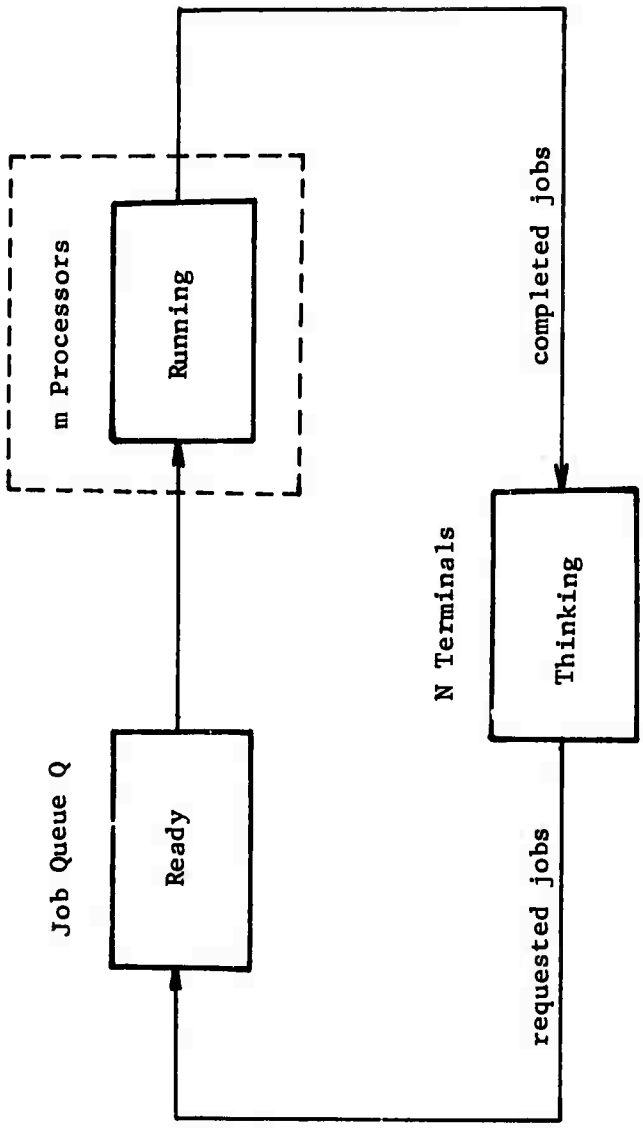


Figure 4-3 Scherr's Model for Uniprogrammed Non-Paging Time-Shared Computer Systems

4.3. Analysis of the Model Behavior

It should be noted that the total system model developed in the previous section is basically the multi-server machine repairman model or an $M/M/S$[1] queue with a finite population [C8,F1,J2]. We shall proceed to derive a response time distribution of the computer system under study using the total system model, step by step, in this section.

4.3.1. Derivation of Queue Length Distribution

As the first step in deriving a response time distribution, this section is concerned with the derivation of a distribution of the queue length, i.e., the number of user jobs either waiting or being serviced (executed) on the processing system.

Noting, as before, that a state of the model is characterized by the number of interactive users currently thinking, the number of interactive users (jobs) queuing in front of the hypothetical processing system, and the number of interactive users (jobs) being serviced by the processors (see Figure 4-3), let S_i $(0 \leq i \leq N)$ denote a state of the model where i user jobs requested by the corresponding i interactive users are either being serviced or waiting for their turn to be serviced

---

[1]The first, second, and third components of this notation specify respectively the type of the arrival process, the type of the service process, and the number of servers of the queuing system. In this case, both the arrival and service processes are Markovian (i.e., exponential) and the queuing system has S servers.

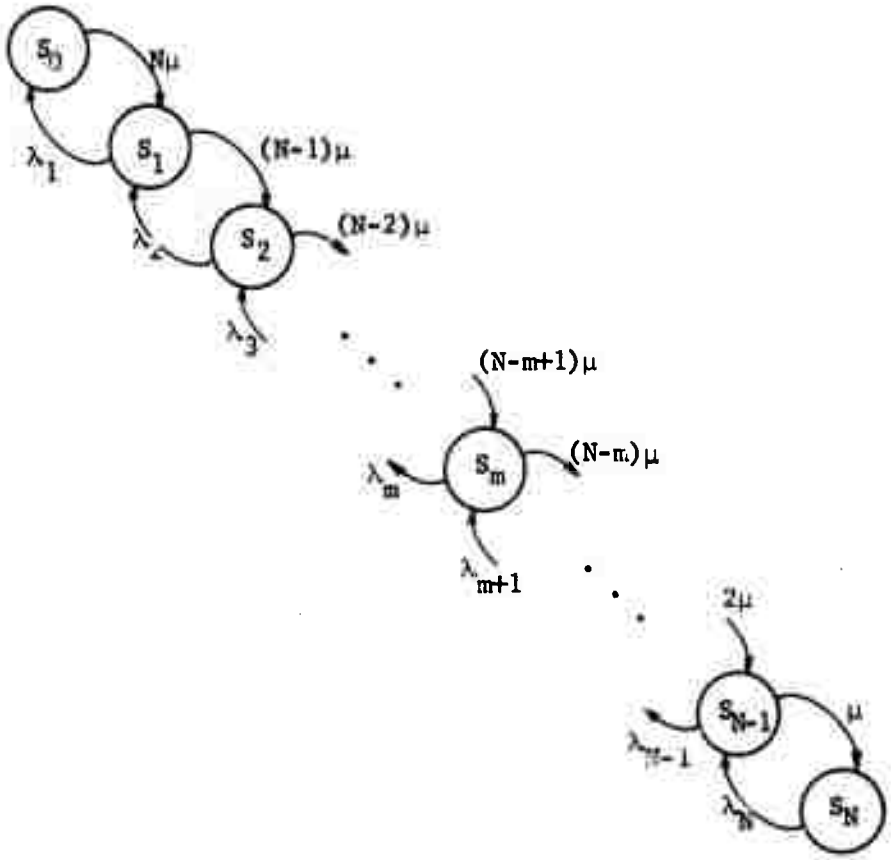
on the processing system, i.e., the queue length is i. (We assume N to be constant and that the service for one of the waiting user jobs starts as soon as a processor becomes available.) Then, the behavior of this stochastic process can be schematically described by a state transition diagram shown in Figure 4-4. Because the effective execution speed of the processors depends on the value of θ' (the "actual" percentile throughput) at that moment, the rates λ_i (the rate with which one of the user processes on the processing system completes its execution when there exist i ($1 \le i \le N$) user processes on the processing system) depends on how θ' reacts to the changes of the actual number (q') of user processes under multiprogramming. Therefore, the rates λ_i are specified for two cases, i.e., the case of fast throughput reaction and that of slow throughput reaction, in Figure 4-4.

Now let π_i ($0 \le i \le N$) be the steady-state probability of S_i. Then, the steady-state probabilities can be obtained [F1] by solving

$$(N-i)\,\mu\,\pi_i = \lambda_{i+1}\pi_{i+1} \qquad 0 \le i \le N-1 \qquad (4.3.1)$$

under the condition $\sum_{i=0}^{N} \pi_i = 1$. Thus, π_i is obtained by recursion, for the case of slow throughput reaction, as follows:

$$\left.\begin{aligned} \pi_i &= \begin{cases} \binom{N}{i} \rho^i \pi_0 & 1 \le i \le m \\ \dfrac{N!}{m!\,(N-i)!} \cdot \dfrac{\rho^i}{m^{i-m}} \cdot \pi_0 & m \le i \le N \end{cases} \\ \text{where} \quad \rho &= \frac{\mu}{\lambda} = \frac{\mu}{\theta\lambda_0} = \frac{\bar{T}_e}{\theta \bar{T}_t} \\ \pi_0 &= \left[\sum_{i=0}^{m-1} \binom{N}{i} \rho^i + \sum_{i=m}^{N} \frac{N!}{m!\,(N-i)!} \cdot \frac{\rho^i}{m^{i-m}} \right]^{-1} \end{aligned}\right\} \qquad (4.3.2)$$



(Fast Throughput Reaction)

$$\lambda_1=\theta_1\lambda_0,\ \lambda_2=2\theta_2\lambda_0,\ \lambda_3=3\theta_3\lambda_0,\ \cdots,\ \lambda_m=m\theta_m\lambda_0,$$
$$\lambda_{m+1}=m\theta_{m+1}\lambda_0,\ \cdots,\ \lambda_{q^*}=\lambda_{q^*+1}=\cdots=\lambda_N=m\theta_{q^*}\lambda_0.$$

(Slow Throughput Reaction)

$$\lambda_1=\theta(N)\lambda_0,\ \lambda_2=2\theta(N)\lambda_0,\ \lambda_3=3\theta(N)\lambda_0,\ \cdots$$
$$\lambda_m=\lambda_{m+1}=\cdots=\lambda_q=\cdots=\lambda_N=m\theta_q\lambda_0.$$

Figure 4-4 State Transition Diagram of the Behavior of the Total System Model

Note that π_i is the probability that one finds i user jobs either waiting or being serviced on the processing system at a randomly chosen instant after the system has operated for a long time. Therefore, π_i gives a steady-state probability distribution of the queue length found at a randomly chosen instant.

The analysis of the case of fast throughput reaction is exactly analogous to that of the above case, and therefore we will consider only the case of slow throughput reaction hereafter.

Now let P_i $(0 \le i \le N)$ be the steady-state probability that an arriving user job (one of the N users) finds i jobs either waiting or being serviced on the processing system at the instant it joined the job queue Q. In other words, P_i is the distribution of the queue length at an arriving instant of a user job. Then, noting that the expected number of arriving users who find exactly i user jobs ahead of themselves, during a duration of time T, is given by $(N - i)\mu \pi_i \cdot T$, we find that P_i is expressed as

$$\left.\begin{aligned} P_i &= \frac{(N-i)\mu \pi_i T}{\sum_{j=0}^{N} (N-j)\mu \pi_j T} = \frac{N-i}{N-\bar{Q}} \cdot \pi_i \\ \text{where } \bar{Q} &= \sum_{j=0}^{N} j \pi_j \end{aligned}\right\} \qquad (4.3.3)$$

It should be emphasized that P_i is different from π_i in nature; for example, P_N is zero because no user can request a new job when jobs of all interactive users are either waiting or being serviced on the processing system, while π_N is obviously non-zero.

4.3.2. Derivation of Effective Percentile Throughput[1]

Now it should be noted that the queue length distributions, π_i and P_i, of the previous section involve $\theta(N)$, as clear from Eq. (4.3.2), whose value is still unknown because $\bar{\pi}_i$ is not known (see Eq. (4.2.1)). Therefore, we proceed to determine the value of $\theta(N)$ in this section.

It is reasonable to expect that π_i (of the hypothetical system) gives a fair estimate of $\bar{\pi}_i$ (of the actual system) if $\theta(N)$ can be accurately estimated. This argument suggests that one solve Eq. (4.3.2) and

$$\theta(N) = \left\{\left(\sum_{i=1}^{q^*-1} \pi_i \theta_i\right) + \left(\sum_{i=q^*}^{N} \pi_i\right) \theta_{q^*}\right\} \Big/ (1-\pi_0) \qquad (4.3.4)$$

simultaneously. ($\bar{\pi}_i$ of Eq. (4.2.1) was replaced by π_i, in Eq. (4.3.4).) Therefore, we must solve these non-linear simultaneous equations. However, if we note that any distribution of π_i obtained from Eq. (4.3.2), assuming a certain value for $\theta(N)$, gives an estimate of $\theta(N)$ which is bounded by θ_1 and θ_{q^*}, then we find that the value of $\theta(N)$ may be iteratively determined, as flollows.

---

[1]The analysis of this section is required only for the case of slow throughput reaction.

Procedure to determine the value of $\theta(N)$

Step 1 Set the variable $\hat{\theta}$ to θ_{q^*}, i.e., $\hat{\theta} = \theta_{q^*}$.

Step 2 Assuming that $\theta(N) = \hat{\theta}$, evaluate π_i using Eq. (4.3.2).

Step 3 Compute $\hat{\theta} = \left\{\left(\sum_{i=1}^{q^*-1} \pi_i \theta_i\right) + \left(\sum_{i=q^*}^{N} \pi_i\right) \theta_{q^*}\right\} \Big/ (1 - \pi_0)$.

Step 4 If $|\theta(N) - \hat{\theta}| \leq \epsilon$, then go to the next step. Otherwise, return to Step 2. (The value of ϵ should be small enough to assure the desired accuracy of the estimation.)

Step 5 $\hat{\theta}$ thus obtained is an estimate of $\theta(N)$.

The initial overestimation of $\theta(N)$ in step 1 leads to the underestimation of $\sum_{i=q^*}^{N} \pi_i$ in step 2, which in turn leads to the underestimation of $\theta(N)$ in step 3. (See Figure 4-5) The condition of step 4 is usually not satisfied and therefore we must return to step 2. The underestimation of $\theta(N)$ then leads to the overestimation of $\sum_{i=q^*}^{N} \pi_i$ in step 2 this time, which in turn leads to the overestimation of $\theta(N)$ in step 3. But the overestimated value of $\theta(N)$ is certainly smaller (closer to the $\theta(N)$ being estimated) than the initially overestimated value of $\theta(N)$, i.e., θ_{q^*}. Repeating this kind of oscillation around the value being estimated, as shown in Figure 4-5, the value of $\theta(N)$ can be determined by the above iterative procedure. In deriving a numerical example to be

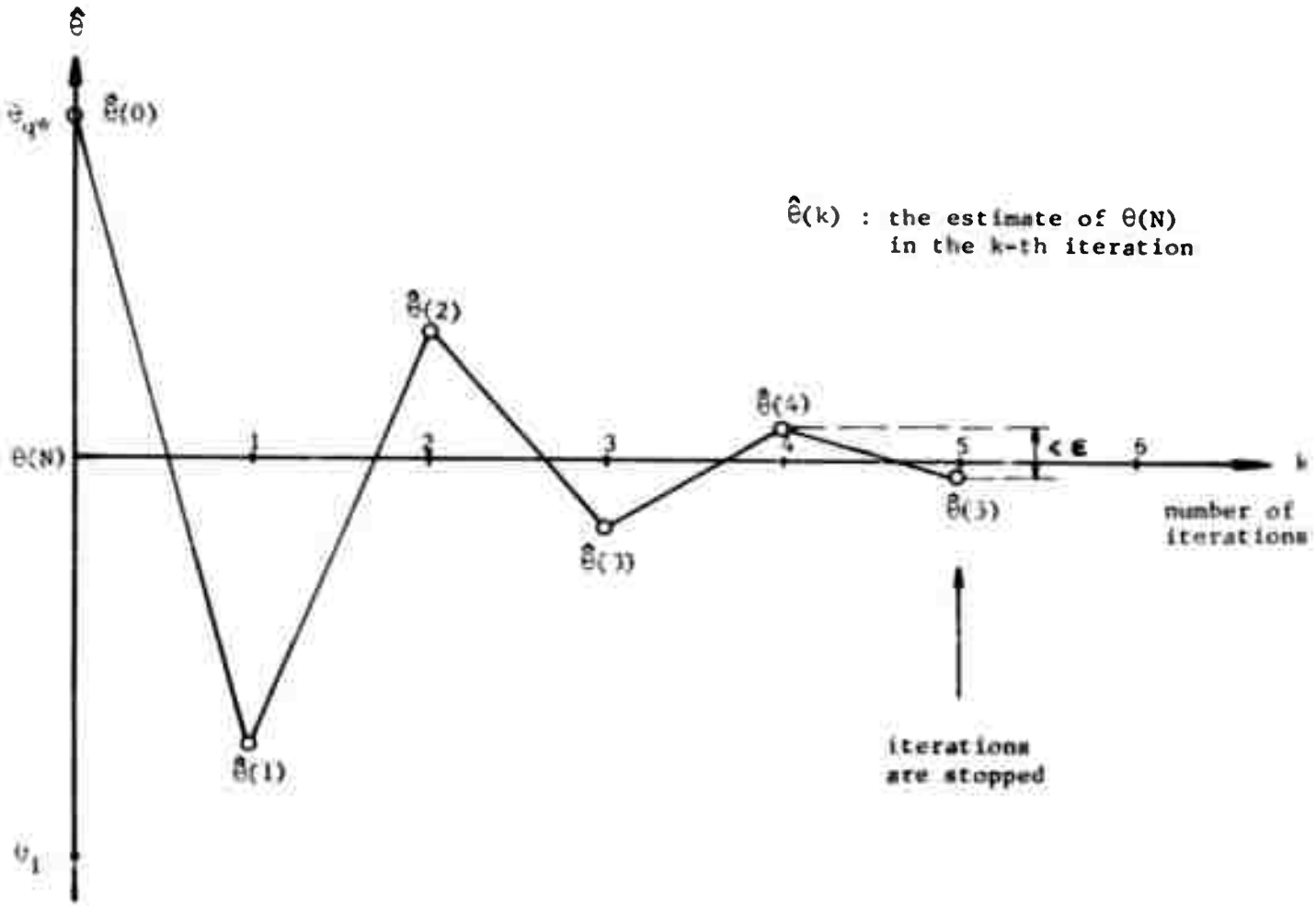


Figure 4-5 Iterative Estimation of Effective Percentile Throughput

given in Chapter 5, it was observed that $\hat{\theta}$ converges within several iterations when ϵ was chosen to be about 0.2 percent of $\theta(N)$. If the number of users (N) is large enough to impose a heavy load upon the system, then $\sum_{i=q^*}^{N} \pi_i \simeq 1$ and $\hat{\theta} \simeq \theta_{q^*}$. Therefore, the above procedure converges without need for iterations. It was furthermore observed that the use of θ_1 (rather than θ_{q^*}) as the initial estimate for $\theta(N)$ in step 1 leads to the convergence to the same value of $\hat{\theta}$. Therefore, it is felt that this iterative procedure uniquely determines the value of $\theta(N)$.

4.3.3. Derivation of Response Time Distribution

The system response time is generally defined as the time elapsed between the receipt by the system of a user's specified job request and the satisfaction of that request at the terminal. For our total system model, we define the (system) response time (T_r) to be the total length of time which is spent on the processing system by each user job. Therefore, the system response time consists of both the waiting time and the execution time of the job.

Let $p\{T_r=t\}$ and $P(s)$ be the probability density of the system response time and the corresponding Laplace transform. Then,

$$P(s) = \int_0^\infty p\{T_r = t\}\, e^{-st}\, dt$$

$$= \int_0^\infty \left[\sum_{i=0}^{N} P_i\, p\{T_r = t \mid i \text{ jobs}\}\, e^{-st} \right] dt$$

$$= \sum_{i=0}^{N} \left[P_i \int_0^\infty p\{T_r = t \mid i \text{ jobs}\}\, e^{-st}\, dt \right] \qquad (4.3.5)$$

where $p\{T_r = t \mid i \text{ jobs}\}$ denotes a conditional probability density of the system response time given that i jobs were found ahead of an arriving job on the processing system. Noting that the m-processor ($1 \le m \le \infty$) processing system schedules arriving jobs (whose execution times are exponentially distributed) according to the FCFS discipline, we see that

$$\left.\begin{aligned} &\int_0^\infty p\{T_r = t \mid i \text{ jobs}\}\, e^{-st}\, dt \\ &= \begin{cases} \dfrac{\lambda}{s+\lambda} & 0 \le i \le m-1 \\ \left(\dfrac{m\lambda}{s+m\lambda}\right)^{i+1-m} \cdot \left(\dfrac{\lambda}{s+\lambda}\right) & m \le i \le N \end{cases} \end{aligned}\right\} \qquad (4.3.6)$$

where the first component, $\left(\frac{m\lambda}{s+m\lambda}\right)^{i+1-m}$, of the latter case represents the Laplace transform of the waiting time and the following second component, $\left(\frac{\lambda}{s+\lambda}\right)$, represents the Laplace transform of a job's execution time. Substituting Eq. (4.3.6) into Eq. (4.3.5), we get

$$P(s) = \sum_{i=0}^{m-1} \left\{ P_i \frac{\lambda}{s+\lambda} \right\} + \sum_{i=m}^{N} \left\{ P_i \left(\frac{m\lambda}{s+m\lambda}\right)^{i+1-m} \cdot \left(\frac{\lambda}{s+\lambda}\right) \right\} \qquad (4.3.7a)$$

$$= \frac{N!\,\pi_0}{N-\bar{Q}} \frac{\lambda}{s+\lambda} \left[\sum_{i=0}^{m-1} \frac{\rho^i}{i!(N-i-1)!} + \sum_{i=m}^{N-1} \left\{ \frac{\rho^i}{m!(N-i-1)!\,m^{i-m}} \left(\frac{m\lambda}{s+m\lambda}\right)^{i+1-m} \right\} \right]$$

(4.3.7)

We can thus derive the k-th moment, $E((T_r)^k)$, of the system response time, using a well-known technique of differentiating P(s) and then setting s to zero [H2].

$$E\left((T_r)^k\right) = (-1)^k \frac{d^k P(s)}{ds^k}\bigg|_{s=0} \qquad k = 1, 2, 3, \ldots \quad (4.3.8)$$

In particular, the average response time is obtained as follows.

$$\begin{aligned}
&E(T_r) \\
&= -\frac{dP(s)}{ds}\bigg|_{s=0} \\
&= \frac{N!\pi_0}{N-\bar{Q}}\left(\sum_{i=0}^{m-1}\frac{\rho^i}{i!(N-i-1)!}\right)\cdot\frac{\lambda}{(s+\lambda)^2}\bigg|_{s=0} \\
&\quad + \frac{N!\pi_0}{N-\bar{Q}}\left(\sum_{i=m}^{N-1}\left\{\frac{\rho^i}{m!(N-i-1)!m^{i-m}}\left(\frac{m\lambda}{s+m\lambda}\right)^{i+1-m}\right\}\right)\cdot\frac{\lambda}{(s+\lambda)^2}\bigg|_{s=0} \\
&\quad + \frac{N!\pi_0}{N-\bar{Q}}\,\frac{\lambda}{s+\lambda}\sum_{i=m}^{N-1}\left\{\frac{\rho^i}{m!(N-i-1)!m^{i-m}}\cdot(i+1-m)\cdot\left(\frac{m\lambda}{s+m\lambda}\right)^{i-m}\cdot\frac{m\lambda}{(s+m\lambda)^2}\right\}\bigg|_{s=0} \\
&= \frac{N!\pi_0}{N-\bar{Q}}\left(\frac{1}{\lambda}\right)\left[\sum_{i=0}^{m-1}\frac{\rho^i}{i!(N-i-1)!} + \sum_{i=m}^{N-1}\left\{\frac{\rho^i}{m!(N-i-1)!}\,\frac{i+1}{m^{i-m+1}}\right\}\right] \\
&= \frac{\displaystyle\sum_{i=0}^{m-1}\frac{i\rho^i}{i!(N-i)!} + \sum_{i=m}^{N}\frac{i\rho^i}{m!(N-i)!m^{i-m}}}{\displaystyle\sum_{i=0}^{m-1}\frac{\rho^i}{i!(N-i-1)!} + \sum_{i=m}^{N-1}\frac{\rho^i}{m!(N-i-1)!m^{i-m}}}\cdot\bar{T}_t \qquad (4.3.9) \\
&\qquad\qquad 1 \le m \le N
\end{aligned}$$

This result agrees with Scherr's result [S4] if $\theta(N) \equiv 1$. Some special cases of Eq. (4.3.9) may be worth mentioning.

$$E(T_r) = \begin{cases} \dfrac{\sum_{i=0}^{N} \dfrac{i\rho^i}{(N-i)!}}{\sum_{i=0}^{N-1} \dfrac{\rho^i}{(N-i-1)!}} \cdot \bar{T}_t & (m = 1) \\ \dfrac{\bar{T}_e}{\theta} & (m = N) \end{cases} \qquad (4.3.10)$$

Obviously, the result for m=N means that if a processor can always start its service for a user's job as soon as the job is submitted by an interactive user the average response time is given by the average execution time of the job.

Now we proceed to invert P(s) to obtain the probability density of the system response time $p\{T_r = t\}$. Let

$$P_1(s) = \frac{\lambda}{s+\lambda}, \qquad P_2(s) = \frac{\lambda}{s+\lambda}\left(\frac{m\lambda}{s+m\lambda}\right)^{i+1-m} \qquad (4.3.11)$$

Then, we must invert $P_1(s)$ and $P_2(s)$ in order to invert P(s) given by Eq. (4.3.7). Denoting the inverse Laplace transform operator by $\mathcal{L}^{-1}$,

$$\mathcal{L}^{-1}[P_1(s)] = \mathcal{L}^{-1}\left[\frac{\lambda}{s+\lambda}\right] = \lambda e^{-\lambda t} \qquad t \geq 0 \qquad (4.3.12)$$

Next, we perform a partial fraction expansion for $P_2(s)$ as the first step to invert $P_2(s)$, assuming temporarily that $m \neq 1$.

$$P_2(s) = \frac{\lambda}{s+\lambda}\left(\frac{m\lambda}{s+m\lambda}\right)^{i+1-m}$$

$$= \frac{c_0}{s+\lambda} + \frac{c_1}{s+m\lambda} + \frac{c_2}{(s+m\lambda)^2} + \ldots + \frac{c_{i+1-m}}{(s+m\lambda)^{i+1-m}} \qquad m \neq 1 \qquad (4.3.13)$$

where c_k $(0 \leq k \leq i+1-m)$ are the unknown constants to be determined. These constants are easily found to be

$$
\left.
\begin{aligned}
c_0 &= (s+\lambda)\, P_2(s)\Big|_{s=-\lambda} = \lambda\left(\frac{m}{m-1}\right)^{i+1-m} \\
c_{i+1-m-j} &= \frac{1}{j!}\,\frac{d^j}{ds^j}\left[(s+m\lambda)^{i+1-m} P_2(s)\right]\Big|_{s=-m\lambda} \\
&= \frac{1}{j!}\,(-1)^j \frac{j!\lambda}{(s+\lambda)^{j+1}}\,(m\lambda)^{i+1-m}\Big|_{s=-m\lambda} \\
&= \frac{-(m\lambda)^{i+1-m}}{(m-1)^{j+1}\,\lambda^j} \qquad 0 \le j \le i-m,\ m \neq 1
\end{aligned}
\right\} \tag{4.3.14}
$$

Now, noting that

$$
\mathcal{L}^{-1}\left[\frac{1}{(s+a)^k}\right] = \frac{t^{k-1}\,e^{-at}}{(k-1)!} \qquad k = 1, 2, 3, \ldots \tag{4.3.15}
$$

$P_2(s)$ specified by Eqs. (4.3.13) and (4.3.14) can be inverted.

$$
\begin{aligned}
&\mathcal{L}^{-1}\left[P_2(s)\right] \\
&= \lambda\left(\frac{m}{m-1}\right)^{i+1-m} \cdot e^{-\lambda t} - \sum_{j=0}^{i-m} \frac{(m\lambda)^{i+1-m}}{(m-1)^{j+1}\,\lambda^j} \cdot \frac{t^{i-m-j}\,e^{-m\lambda t}}{(i-m-j)!} \qquad m \neq 1
\end{aligned} \tag{4.3.16}
$$

Using the results of Eqs. (4.3.12) and (4.3.16), the probability density of the response time of the m-processor system ($m \neq 1$) is obtained as follows:

$$p\{ T_r = t \}$$

$$= \mathcal{L}^{-1}\left[P(s)\right]$$

$$= \frac{N!\pi_0}{N-\bar{Q}}\left(\sum_{i=0}^{m-1}\frac{\rho^i}{i!(N-i-1)!}\right)\lambda e^{-\lambda t}$$

$$+\frac{N!\pi_0}{N-\bar{Q}}\sum_{i=m}^{N-1}\left\{\frac{\rho^i}{m!(N-i-1)!m^{i-m}}\left(\lambda\left(\frac{m}{m-1}\right)^{i+1-m}\cdot e^{-\lambda t}-\sum_{j=0}^{i-m}\left(\frac{(m\lambda)^{i+1-m}}{(m-1)^{j+1}\lambda^j}\,\frac{t^{i-m-j}e^{-m\lambda t}}{(i-m-j)!}\right)\right)\right\}$$

$$=\frac{\lambda e^{-\lambda t}}{N-\bar{Q}}\,\pi_0\sum_{i=0}^{m-1}\left\{\binom{N}{i}(N-i)\rho^i\right\}$$

$$+\frac{m\lambda}{N-\bar{Q}}\left(\frac{N!}{m!}\right)\rho^m\pi_0\sum_{i=0}^{N-m-1}\left\{\frac{\rho^i}{(N-m-1-i)!(m-1)^{i+1}}\left(e^{-\lambda t}-e^{-m\lambda t}\sum_{j=0}^{i}\frac{\{(m-1)\lambda t\}^j}{j!}\right)\right\}$$

$$m\neq 1 \qquad (4.3.17)$$

For the single processor system ($m = 1$), Eq. (4.3.7) reduces to

$$P(s)=\sum_{i=0}^{N}P_i\left(\frac{\lambda}{s+\lambda}\right)^{i+1}=\frac{1}{N-\bar{Q}}\cdot\frac{\sum_{i=0}^{N-1}\frac{\left(\frac{\lambda}{s+\lambda}\right)^{i+1}}{(N-i-1)!}\rho^i}{\sum_{j=0}^{N}\frac{\rho^j}{(N-j)!}} \qquad (4.3.18)$$

Therefore, the probability density of the response time of the single processor system is obtained, using Eq. (4.3.15), as follows:

$$p\{T_r = t\}$$

$$= \mathcal{L}^{-1}\left[P(s)\right]$$

$$= \frac{1}{N-\bar{Q}} \cdot \frac{\sum_{i=0}^{N-1}\left\{\frac{\rho^i}{(N-i-1)!}\,\frac{t^i e^{-\lambda t}\lambda^{i+1}}{i!}\right\}}{\sum_{j=0}^{N}\frac{\rho^j}{(N-j)!}}$$

$$= \frac{\lambda e^{-\lambda t}(1+\mu t)^{N-1}}{\sum_{i=0}^{N-1}\frac{(N-1)!\,\rho^i}{(N-i-1)!}} \qquad (4.3.19)$$

The probability distribution function, $P\{T_t \le t\}$, of the system response time can be obtained by integrating the probability density given by Eq. (4.3.17) or Eq. (4.3.19) from 0 to t. For $m \neq 1$, noting first that

$$\int_0^t t^j e^{-m\lambda t}\,dt$$

$$= \left[-\frac{e^{-m\lambda t}}{m\lambda}\sum_{k=0}^{j}\frac{j!\,t^{j-k}}{(j-k)!\,(m\lambda)^k}\right]_0^t$$

$$= \frac{1}{m\lambda}\cdot\frac{j!}{(m\lambda)^j} - \frac{e^{-m\lambda t}}{m\lambda}\sum_{k=0}^{j}\frac{j!\,t^k}{k!\,(m\lambda)^{j-k}} \qquad j = 1, 2, \ldots \qquad (4.3.20)$$

the integration of Eq. (4.3.17) gives

$$P\{T_r \le t\}$$

$$= \frac{1-e^{-\lambda t}}{N-\bar{Q}} \pi_0 \left\{ \sum_{i=0}^{m-1} \binom{N}{i} (N-i)\, \rho^i \right\}$$

$$+ \frac{m}{N-\bar{Q}} \left(\frac{N!}{m!}\right) \rho^m \pi_0 \sum_{i=0}^{N-m-1} \frac{\rho^i (1-e^{-\lambda t})}{(N-m-1-i)!\,(m-1)^{i+1}}$$

$$- \frac{m\lambda}{N-\bar{Q}} \left(\frac{N!}{m!}\right) \rho^m \pi_0 \sum_{i=0}^{N-m-1} \left[\frac{\rho^i}{(N-m-1-i)!\,(m-1)^{i+1}} \cdot \frac{1}{m\lambda} \sum_{j=0}^{i} \left\{ \left(\frac{m-1}{m}\right)^j \left(1 - e^{-m\lambda t} \sum_{k=0}^{j} \frac{(m\lambda t)^k}{k!}\right) \right\} \right]. \tag{4.3.21}$$

Then, noting that

$$\sum_{j=0}^{i} \left\{ \left(\frac{m-1}{m}\right)^j \sum_{k=0}^{j} \frac{(m\lambda t)^k}{k!} \right\}$$

$$= \sum_{j=0}^{i} \left\{ \left(\frac{m-1}{m}\right)^i \cdot \left(\frac{1 - \left(\frac{m}{m-1}\right)^{i+1-j}}{1 - \frac{m}{m-1}} \right) \cdot \left(\frac{(m\lambda t)^j}{j!} \right) \right\}$$

$$= m \left[\sum_{j=0}^{i} \frac{\{(m-1)\lambda t\}^j}{j!} \right] - \left(\frac{m-1}{m}\right)^i \cdot (m-1) \left[\sum_{j=0}^{i} \frac{(m\lambda t)^j}{j!} \right], \tag{4.3.22}$$

the third (last) term of Eq. (4.3.21) becomes

$$- \frac{m}{N-\bar{Q}} \left(\frac{N!}{m!}\right) \rho^m \pi_0 \sum_{i=0}^{N-m-1} \left[\frac{\rho^i}{(N-m-1-i)!\,(m-1)^{i+1}} - \frac{\rho^i}{(N-m-1-i)!\,m^{i+1}} \right.$$

$$\left. - \frac{\rho^i e^{-m\lambda t}}{(N-m-1-i)!\,(m-1)^{i+1}} \sum_{j=0}^{i} \frac{\{(m-1)\lambda t\}^j}{j!} + \frac{\rho^i e^{-m\lambda t}}{(N-m-1-i)!\,m^{i+1}} \sum_{j=0}^{i} \frac{(m\lambda t)^j}{j!} \right] \tag{4.3.23}$$

Therefore, the probability distribution function of the system response time for $m \neq 1$ is obtained as

$$P\{ T_r \leq t \}$$

$$= \frac{1-e^{-\lambda t}}{N-\bar{Q}} \pi_0 \left\{ \sum_{i=0}^{m-1} \binom{N}{i} (N-i)\, \rho^i \right\}$$

$$+ \frac{m}{N-\bar{Q}} \left(\frac{N!}{m!}\right) \rho^m \pi_0 \sum_{i=0}^{N-m-1} \left\{ \frac{\rho^i}{(N-m-1-i)!\, m^{i+1}} \left(1 - e^{-m\lambda t} \sum_{j=0}^{i} \frac{(m\lambda t)^j}{j!} \right) \right\}$$

$$- \frac{m}{N-\bar{Q}} \left(\frac{N!}{m!}\right) \rho^m \pi_0 \sum_{i=0}^{N-m-1} \left\{ \frac{\rho^i e^{-\lambda t}}{(N-m-1-i)!\,(m-1)^{i+1}} \left(1 - e^{-(m-1)\lambda t} \sum_{j=0}^{i} \frac{\{(m-1)\lambda t\}^j}{j!} \right) \right\}$$

where

(4.3.24)

$$\frac{\pi_0}{N-\bar{Q}} = \frac{1}{N! \left\{ \sum_{i=0}^{m-1} \frac{\rho^i}{i!(N-i-1)!} + \sum_{i=m}^{N-1} \frac{\rho^i}{m!(N-i-1)!\, m^{i-m}} \right\}} \qquad m \neq 1$$

For $m = 1$, the integration of Eq. (4.3.19) similarly gives

$$P\{ T_r \leq t \}$$

$$= \frac{1}{\left(\sum_{i=0}^{N-1} \frac{\rho^i}{(N-i-1)!} \right)} \cdot \sum_{i=0}^{N-1} \left[\frac{\mu^i}{(N-i-1)!\, i!} \left(\frac{i!}{\lambda^i} - e^{-\lambda t} \cdot \sum_{j=0}^{i} \frac{i!\, t^{i-j}}{(i-j)!\, \lambda^j} \right) \right]$$

$$= 1 - e^{-\lambda t} \left(\frac{\sum_{i=0}^{N-1} \left\{ \frac{\rho^i}{(N-i-1)!} \sum_{j=0}^{i} \frac{(\lambda t)^j}{j!} \right\}}{\sum_{i=0}^{N-1} \frac{\rho^i}{(N-i-1)!}} \right) \qquad (4.3.25)$$

Thus, we have derived the expressions for the probability density and the probability distribution function of the response time of a multiprogrammed multi-processor time-shared virtual-memory computer system. The result of the probability distribution function of the system response time enables us to evaluate the percentile response time numerically, as will be carried out in Chapter 5.

Finally, the last part of this section presents two somewhat peripheral but related results; a result of the response time distribution conditioned on a job's execution time and a result of the first-time response time distribution (to be defined). Each of these results can be obtained by a slight modification of the analysis of this section. We have assumed until now that execution times of all user jobs are only probabilistically (exponentially) known. Now we assume that the execution time of a particular arriving job is completely known, but the execution time of other user jobs already on the processing system are only probabilistically known as above. Then, the response time (T_r) to be experienced by a user who has requested a job requiring an execution time of T_e seconds (excluding all the overhead times to be added) is given by the sum of the waiting time (T_w) in the job queue Q and its own execution time T_e. Letting the Laplace transform of the waiting time T_w be $P_w(s)$, we have (see Eqs. (4.3.5),(4.3.6),and (4.3.7)) therefore the following expression.

$$P_w(s) = \int_0^\infty p\{ T_w = t \}\, e^{-st}\, dt$$

$$= \sum_{i=0}^{N} \left(P_i \int_0^\infty p\{ T_w = t \mid i \text{ jobs} \}\, e^{-st}\, dt \right)$$

$$= \sum_{i=0}^{m-1} \left(P_i \int_0^\infty \delta(t)\, e^{-st}\, dt \right) + \sum_{i=m}^{N} \left(P_i \int_0^\infty p\{ T_w = t \mid i \text{ jobs} \}\, e^{-st}\, dt \right)$$

$$= \sum_{i=0}^{m-1} P_i + \sum_{i=m}^{N} P_i \left(\frac{m\lambda}{s+m\lambda} \right)^{i-m+1} \qquad (4.3.26)$$

where $\delta(t)$ is a delta function at $t = 0$. Using Eq. (4.3.15), this can be readily inverted as follows:

$$p\{ T_w = t \}$$

$$= \left(\sum_{i=0}^{m-1} P_i \right) \delta(t) + \sum_{i=m}^{N} P_i\, (m\lambda)^{i-m+1}\, \frac{t^{i-m} e^{-m\lambda t}}{(i-m)!}$$

$$= \frac{N!\pi_0}{N-\bar{Q}} \left(\sum_{i=0}^{m-1} \frac{\rho^i}{i!(N-i-1)!} \right) \delta(t) + \frac{m\lambda e^{-m\lambda t}}{N-\bar{Q}} \left(\frac{N!}{m!} \right) \rho^m \pi_0 \sum_{i=0}^{N-m-1} \frac{(\mu t)^i}{(N-m-i-1)!\, i!}$$

(4.3.27)

Therefore, the probability density of the system response time conditioned on a job's execution time, $p\{T_r = t \mid T_e\}$, is given by

$$p\{T_r = t \mid T_e\}$$

$$= p\{T_w = t - T_e\}$$

$$= \frac{N!\pi_0}{N-\bar{Q}}\left(\sum_{i=0}^{m-1}\frac{\rho^i}{i!(N-i-1)!}\right)\delta(t-T_e) + \frac{m\lambda e^{-m\lambda(t-T_e)}}{N-\bar{Q}}\left(\frac{N!}{m!}\right)\rho^m\pi_0\sum_{i=0}^{N-m-1}\frac{\{\mu(t-T_e)\}^i}{(N-m-1-i)!\,i!} \tag{4.3.28}$$

Thus, if the execution time of a job is known in advance, the results given by Eqs. (4.3.17) and (4.3.19) must be modified, as given by Eq. (4.3.28). The corresponding probability distribution function and its moments can be easily obtained from the above result.

We have been concerned with the response time to be experienced by each of N interactive terminal users who are currently using the time-shared computer system. One may ask a question about the system response time to be experienced by a new user who is about to join the existing N user population by typing the "login" command at his terminal. This <u>first-time response time</u> should be slightly different from the one with which we have been concerned because such a new user joins the system independently of the system state (e.g., the number of queuing users). Let $T_{e\ell}$ (i.e., $T_e = T_{e\ell}$) be the execution time (excluding all the overhead times to be added) of the "login" command. Then, noting this new user's random arrival to the system, we see that the Laplace transform $P_w(s)$ of the probability density of the waiting time T_w experienced by this user is given by Eq. (4.3.26) with P_i replaced by π_i.

Therefore, the probability density of the response time to be experienced by the login command, $p\{T_r(\text{login}) = t \mid T_{e\ell}\}$, is obtained as follows:

$$p\{T_r(\text{login}) = t \mid T_{e\ell}\}$$

$$= p\{T_w(\text{login}) = t - T_{e\ell}\}$$

$$= \sum_{i=0}^{m-1} \binom{N}{i} \rho^i \pi_0 \cdot \delta(t-T_{e\ell}) + m\lambda e^{-m\lambda(t-T_{e\ell})} \left(\frac{N!}{m!}\right) \rho^m \pi_0 \sum_{i=0}^{N-m} \frac{\{\mu(t-T_{e\ell})\}^i}{(N-m-i)!\,i!} \qquad (4.3.29)$$

Thus, the probability density of the first-time response time has been obtained. The corresponding probability distribution function and its moments can be readily derived from the above result.

4.3.4. Relationship with an Infinite Population Model

In developing queuing models for time-shared computer systems, one must make a fundamental choice between the use of an infinite or finite population model. It is clearly stated in [M3] that the finite population model is much more appropriate as a model of time-shared computer systems. The basic difference resulting from the size of the population consists in the fact that the arrival rate of the finite population model is dependent on the state of the system (e.g., the number of queuing users) while the arrival rate of the infinite population model is not; the arrival rate of the finite population model increases (decreases) if the number of queuing users decreases

(increases), showing the existence of a negative feedback mechanism to stabilize the number of queuing users, but the arrival rate of the infinite population model is always constant. In view of the fact that the infinite population model is however often used in modeling computer systems, we will make a brief remark about the relationship of these two kinds of models in this section.

We will particularly discuss the relationship between the M/M/m queue with N users with which we have been concerned and the M/M/m queue with an infinite population [C8,F1] whose arrival rate is μ_0(a constant). For such an infinite population model, the steady-state probability P(i) of finding i jobs on the system at the instant that a new job joins the waiting queue is obtained [C8,F1] as

$$P(i) = \left\{ \begin{array}{ll} \dfrac{\rho^i}{i!} P(0) & 1 \le i \le m \\ \dfrac{\rho^i}{m! m^{i-m}} P(0) & m \le i < \infty \end{array} \right\} \qquad (4.3.30)$$

where $\rho = \mu_0 / \lambda$, $P(0) = \left[\left(\sum_{i=0}^{m-1} \dfrac{\rho^i}{i!} \right) + \dfrac{m\rho^m}{m!(m-\rho)} \right]^{-1}$

It can be easily shown [F1] that as $N \to \infty$ and $\mu \to 0$ in our finite population model in such a way that $N\mu$ remains a constant μ_0, π_i given by Eq. (4.3.2) approaches P(i) given by Eq. (4.3.30). But for the series $\Sigma(P(i)/P(0))$ to converge we must have

$$\frac{\mu_0}{m\lambda} = \text{"utilization"} < 1 \qquad (4.3.31)$$

Noting that $\bar{Q}$ is finite under this condition,

$$\lim_{\substack{N\to\infty \\ \mu\to 0 \\ N\mu=\mu_0}} P_i = \lim_{\substack{N\to\infty \\ \mu\to 0 \\ N\mu=\mu_0}} \frac{N-i}{N-\bar{Q}}\,\pi_i = \lim_{\substack{N\to\infty \\ \mu\to 0 \\ N\mu=\mu_0}} \pi_i = P(i) \qquad (4.3.32)$$

Therefore, both P_i and π_i approaches $P(i)$ as $N\to\infty$ and $\mu\to 0$ under the constraint $N\mu = \mu_0$. This implies that the system response time obtained for the finite population model approaches the system response time of the infinite population model, under this limit. For example, the probability density of the system response time given by Eq. (4.3.19) approaches the corresponding solution of the infinite population model, i.e.,

$$p\{T_r = t\} = (\lambda-\mu_0)\, e^{-(\lambda-\mu_0)t} \qquad t \geq 0 \qquad (4.3.33)$$

In the case of the infinite population model, the system response time is exponentially distributed, as above. In contrast, the probability density of the system response time obtained using the finite population model of this section (see Eqs. (4.3.17) and (4.3.19) will be numerically found in Chapter 5 to be much closer to a normal distribution density if the number (N) of interactive users is large. This tendency, usually observed on actual computer systems [S6], is a result of the finiteness of the user population.

4.4. Some Remarks on Modeling Errors

We will finish the response time analysis of the multiprogrammed time-shared computer system by examining the modeling errors which may be introduced to this analysis. Looking back at what we have been doing, we have developed a queuing theoretic model of the computer system whose characteristics were specified in Section 4.2 and have analyzed the behavior of such a model to investigate the system response time of the computer system under study. We will not consider what happens if a given (actual) system is slightly different from the one considered in Section 4.2 (the reader who is interested in this subject should read D'Avanzo [D4]), but will be concerned with the errors possibly introduced in approximating such an actual computer system with our queuing model called the total system model.

We will particularly discuss three approximations being used in the analysis. The first one is the approximation to use the hypothetical system, of Figure 4-2, whose execution speed is traded for the percentile throughput. As stated in the previous chapter, $(1-\theta) \times 10^2$ percent of the computational capacity of the actual computer system cannot be utilized for users' useful computation; this proportion of the system's capacity is wasted either by the system overhead operations or by processor idle time. If the length of each burst of these wastages is comparable to the length of the execution time of each job, then the speed-capacity tradeoff will not give a good approximation. (Consider the variance of the system response time, for example.) But fortunately, on actual computer systems, these wastages (e.g., paging overhead) usually have much shorter lengths than the job execution times and

moreover tend to occur uniformly in time. Therefore, this approximation does not seem to introduce a significant error in the analysis.

The second approximation consists in the use of the effective percentile throughput $\theta(N)$. Basically, the variable θ system (actual system) is approximated by the constant θ system. This implies that if θ' of the actual system (N = fixed) varies very much in time this approximation tends to underestimate, for example, the 90 percentile response time. If that is the case, the use of θ_1 for $\theta(N)$, i.e., the use of the largest lower bound for $\theta(N)$, will give a good upper bound estimate of the 90 percentile response time. This kind of underestimation may be significant especially in the medium load range of N because of relatively frequent fluctuations of θ' (see Figure 4-1); the estimated 90 percentile response time of the heavy load or the light load range of N tends to be accurate.

The third approximation consists in the fact that user jobs are executed up to completion under the FCFS discipline on the hypothetical system while the jobs are not executed strictly in this way on an actual multiprogrammed system using demand paging even if all the scheduling disciplines (the ones associated with the memory queue and the processor queue) are FCFS. To see this, consider a situation where a large job and a small job enter the processor queue respectively at time t and $t+\Delta t$. It is very likely that the small job which arrived at the processing system later than the large one will be completed earlier than the large one, because the small one is likely to demand a smaller number of missing-pages than the other. This means that a job which arrives at the system later than another can be completed earlier on the actual

multiprogrammed system under study. Therefore, we must consider the effect of this kind of favoritism for short jobs upon the system response time. The average response time is not affected because the execution time is assumed to follow the (memoryless) exponential distribution, but its variance or 90 percentile response time is. The variance of the system response time is basically proportional to the sum of the variances of job execution times because execution times are assumed to be independent of each other. Considering that if the system is under a heavy load the number of multiprogrammed jobs is constant (q*) but the number of jobs waiting in the memory queue varies in time, we see that the variance of the response time of the system under a heavy load is determined mainly by the varying number of the jobs in the FCFS memory queue. Therefore, this argument suggests that the 90 percentile system response time tends to be accurate as the number of jobs queuing on the processing system increases. on the contrary, if there are only moderate number of jobs on the processing system , the variance of the response time would be affected by the favoritism for short jobs; the variance (or 90 percentile response time) predicted by the total system model is somewhat smaller (shorter) than that of the corresponding actual system because the genuine FCFS scheduling discipline, used in the model, is known to attain the smallest variance of response times

[C7]

CHAPTER 5

MODEL VALIDATION, PERFORMANCE PREDICTION AND OPTIMIZATION, AND CONFIGURATION SELECTION

5.1. Introduction

We have finished the development and analyses of all the hierarchically organized modular models shown in Figure 1-4 that we intended to describe in this thesis. It is time to examine the validity of these models and to consider if the performance questions raised in Table 1.1 of Chapter 1 can be answered by a series of these analyses given in Chapters 2, 3, and 4.

We will first examine the validity of the processor model and the total system model by comparing the system performance predicted by these two models and the available statistics of an actual computer system, the Multics system of M.I.T. This validity examination of the models is intended to present a rough idea about the accuracy of the system performance that can be predicted by these models. Then, noting that the performance questions of Table 1.1 can be classified into

performance prediction problems (the second through the fifth questions), performance optimization problems (the sixth through the eighth questions), and a configuration selection problem (the first question), we shall proceed to consider each class of problems quantitatively one by one.

5.2. Model Validation

In this section, the validity of the models developed in the preceeding chapters will be examined by comparing the performance of an actual large-scale time-shared computer system, the Multics system of M.I.T. [C10,C12], with the performance result that can be obtained using these models. The instrumentation used in the development of the Multics system and the performance statistics obtained from this running system will be first described and then the details of the statistical results will be presented.

5.2.1. Instrumentation and Multics Performance

The Multics system is a large-scale time-shared computer system developed as a cooperative effort involving the Bell Telephone Laboratories (from 1965 to 1969), the computer department of the General Electric Company (subsequently acquired by Honeywell Information Systems Inc.), and Project MAC of M.I.T. The system has all the features of modern large-scale time-shared computer systems such as paging, segmentation, multiprogramming, multi-processing, memory hierarchy, and so on.

The current standard configuration of the Multics processing system at M.I.T. includes two processors, three primary memory units (128 k thirty-six bit words per unit), and the secondary memory system consisting of drum and disk memories, with five or six eligible user processes under multiprogramming (see Figure 5-1). Occasionally, a processor, a primary memory unit, and/or some part of secondary memory

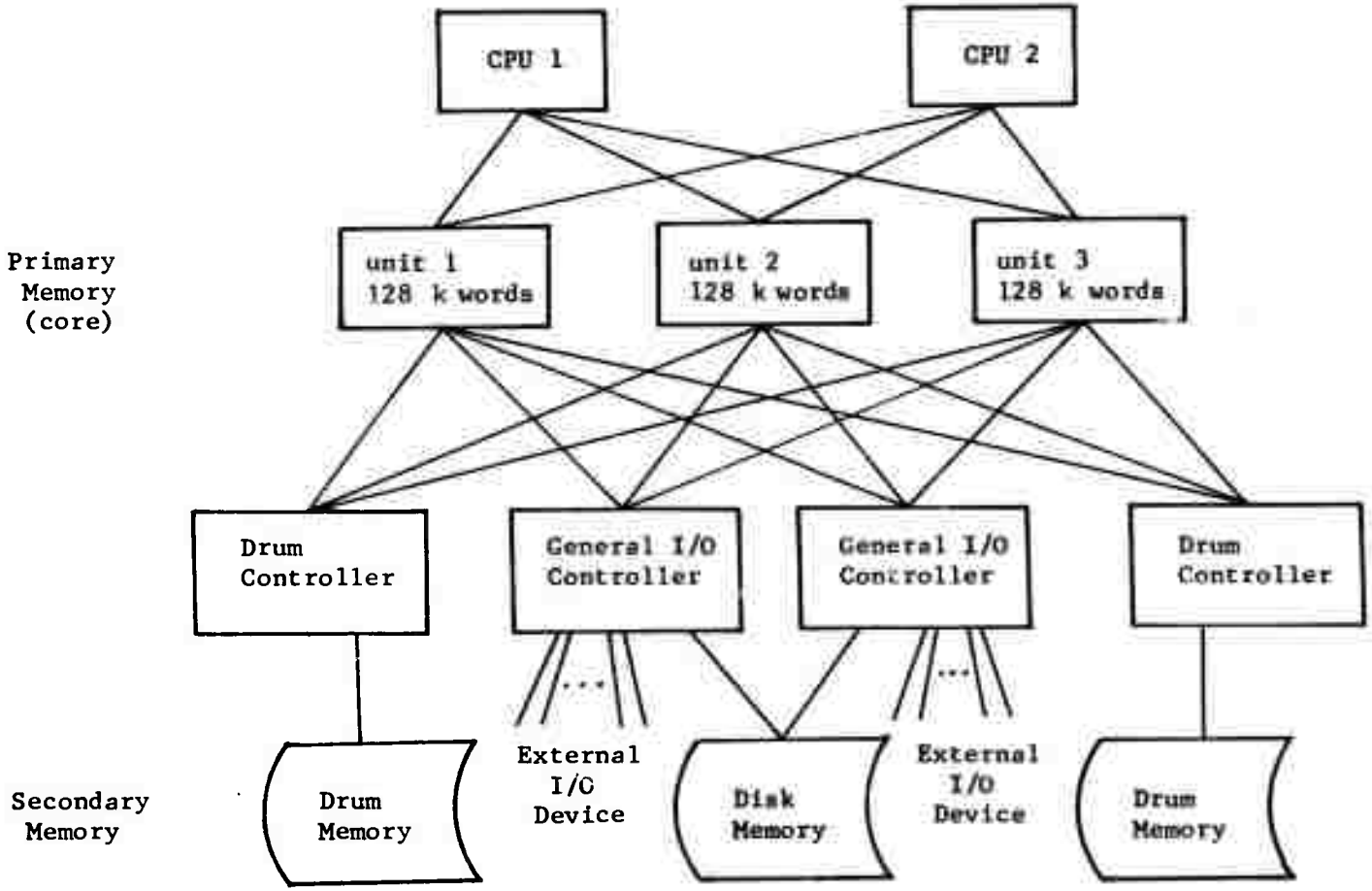


Figure 5-1 A Typical Multics Processing System Configuration

is removed from the servicing system, for maintenance or to create another system for debugging a new version of the operating system [C12]. Therefore, the Multics system has been running with one of the following hardware configurations during the past year.

(1) large-scale configuration ... standard configuration[1]
(two-processor three-primary-memory-unit system)

(2) medium-scale configuration
(one-processor three-primary-memory-unit system)

(3) small-scale configuration
(one-processor two-primary-memory-unit system)

In this way, every effort has been made to support the continued operation of the system.

Early in the design stage of this system, it was felt that the cost of maintaining well-organized instrumentation can be made low and the payoff in being able to "look at the meters" any time a performance problem is suspected is very high. This initial conviction resulted in a comprehensive set of system metering commands that can be used from any terminal [S2] and the use of a DEC PDP-8 system as a peripheral computer to test and monitor the operation of the Multics system [G3,G4]. The metering commands have proved to be extremely useful in measuring the paging performance of programs, the secondary memory performance, the processor time usage (among various system overhead times, idle

---

[1] Whenever there was not a need to reduce the configuration, this standard configuration of the Multics system has been used.

times, and users' useful computation time), and user behavior characteristics. The use of the PDP-8 computer as a simulator of interactive users [G3] has provided system developers with a convenient tool to test the performance of a newly installed version of the operating system; a standard benchmark (or a series of commands interspersed by think times representing a typical debugging user process) has been run from time to time during the past four years to measure the execution time, the paging performance, and the response time of each command included in the benchmark. Thus, these tools have enabled system analysts to obtain the operational statistics of the Multics performance which are essential in improving the succeeding versions of the operating system.

Many performance statistics concerning the Multics system have been accumulated using these measurement tools. Three sample results each representing the performance of a different hardware configuration were randomly chosen from these performance statistics to show the typical performance of each of these three hardware configurations. These sample results are given in Table 5-1. Each sample result was collected from the system during a thirty-minute session either in the morning or afternoon of a normal working day when the system was operating normally and was fully loaded with interactive terminal users. Although the numbers given in Table 5-1 are subject to statistical fluctuations, it is felt that they are representative of typical performance of the Multics system and that they can be used for the purpose of examining the validity of the models developed in this thesis.

Table 5-1 Typical Performance of the Multics System under a Full Load

Configuration	Small 1 CPU 2 PM units	Medium 1 CPU 3 PM units	Large 2 CPUs 3 PM units
average number of eligible processes	5.7	5.3	5.6
mtbpf (msec)	16.5	29.7	31.9
mpft (msec)	32.4	33.2	30.9
mean paging overhead time (msec)	5.4	6.1	9.0
processor time breakdown			
multiprogramming idle	2.9 %	1.8 %	8.8 %
memory interference idle	0	0	5.7
paging overhead	28.7	18.4	22.9
miscellaneous overhead	16.6	15.9	12.6
users' useful computation	51.8	63.9	49.9
number of users	41	43	48
response time characteistics			
average queue length	16.7	14.9	14.9
average response time to the PDP8 user simulator (sec)	not measured	7.9	6.0

July 1971

Using a set of metering commands, the mean paging overhead times (all the supervisory operations necessitated by a page fault are included) of a single processor system and a dual processor system are respectively found to be approximately 6 and 9 milliseconds long.[1] On the other hand, the value of the miscellaneous overhead coefficient (δ) is found to be between 0.2 and 0.35, with a typical value of 0.25. The average think time of interactive users is found to vary from session to session (e.g., 12 to 35 seconds), but it is typically 15 to 25 seconds long. These numbers represent shorter think times than observed for the CTSS system by Scherr [S4]. The average execution time required by an interactive job (excluding all the overhead times to be added) is roughly 400 milliseconds long.

In a special measurement of multi-processor interference, it was found that if two processors direct their accesses to a particular primary memory unit at all times the execution time of a typical program is stretched by about twenty percent. This means that the value of the memory interference coefficient of the Multics system is approximately 1.2, i.e., $\gamma = 1.2$. On the other hand, it has been generally observed on the large-scale configuration (with three primary memory units) under a heavy load that the system typically loses 5 to 6 percent of each processor's processing time because of memory cycle

---

[1] Most of the increase of the paging overhead time on a dual processor system is due to the data-base lockout. As described in Table 3-1 of Chapter 3, the data-base lockout is frequent enough to prolong significantly program execution time only in the page fault processing.

interference[1] and another 5 to 6 percent of each processor's processing time because of data base lockout. Most of the loss due to data-base lockout is being caused by the lockout of the page table.

5.2.2. Validation of Processor Model

In this section, we examine the validity of the processor model of Chapter 3 by comparing the performance of the above three configurations of the Multics system to be predicted by the processor model and the actual performance of the same system summarized in Table 5-1.

For this purpose, a program was written in PL/I to derive the performance of the processing system under investigation using the processor model. We call this program the throughput analysis program and it is included in Appendix A together with an explanation about how to use it and a sample console session using the program. This program derives the processor time breakdown into various system overhead times, idle times, and users' useful computation time (i.e., percentile throughput), upon specification of the configuration of the processing system under study.

In predicting the performance of each of the above three hardware configuration of the Multics system, all the input parameters (except the degree of multiprogramming) were set fairly close to the corresponding measured values given in Table 5-1. The degree of multiprogramming

---

[1] This observation will be found consistent with the above measurement result, $\gamma = 1.2$, numerically in Section 5.2.2.

was then changed to see the effect on the resulting processor time breakdown. The result is shown in Table 5-2 (a), (b) and (c). It is clearly seen that as the degree of multiprogramming approaches the measured value of the average number of eligible processes of Table 5-1 the resulting processor time breakdown becomes very similar to the measured processor time breakdown of Table 5-1. The closest result analytically obtained for each configuration is enclosed by a broken line in Table 5-2. For example, it is seen that the processor time breakdown of the small-scale configuration predicted by the processor model becomes very close to the measured performance of this configuration given in Table 5-1 when the degree of multiprogramming is fixed at either 5 or 6. The performance of the two-processor configuration (i.e., the large-scale configuration) is also predicted fairly accurately by the processor model, as seen in Table 5-2 (c).

As a result of comparing the predicted performance of Table 5-2 and the actually measured performance of the Multics system, we conclude that the processor model can be used as a practical tool of performance prediction despite its simplified abstraction of complex structure of actual computer systems (see Section 1.6).

5.2.3. Validation of Total System Model

In this section, we examine the validity of the total system model developed in Chapter 4 by comparing the performance of the three configurations of the Multics system to be predicted by the total system model and the actual performance of the corresponding configura-

Table 5-2 Validation of the Processor Model

(a) one-processor two-primary-memory-unit configuration, i.e., small-scale configuration

degree of multiprogramming	2	3	4	5	6
multiprogramming idle	39.4	20.5	9.2	3.5	1.1
memory interference idle	0	0	0	0	0
paging overhead	19.8	26.0	29.7	31.6	32.4
miscellaneous overhead	9.9	13.0	14.8	15.7	16.1
users' useful computation	30.9	40.5	46.3	49.2	50.4

mtbpf = 16.5 msec, mpft = 32.4 msec, $\bar{t}_{pl} = 5.4$ msec, $\delta = 0.32$

(b) one-processor three-primary-memory-unit configuration, i.e., medium-scale configuration

degree of multiprogramming	1	2	3	4	5
multiprogramming idle	53.8	23.9	8.5	2.4	0.6
memory interference idle	0	0	0	0	0
paging overhead	9.4	15.5	18.6	19.8	20.2
miscellaneous overhead	7.4	12.1	14.6	15.5	15.8
users' useful computation	29.4	48.5	58.3	62.2	63.4

mtbpf = 30 msec, mpft = 35 msec, $\bar{t}_{pl} = 6.1$ msec, $\delta = 0.25$

(c) two-processor three-primary-memory-unit configuration, i.e., large-scale configuration

degree of multiprogramming	1	2	3	4	5
multiprogramming idle	74.6	48.4	27.2	13.0	5.2
memory interference idle	0	1.7	3.5	4.9	5.7
paging overhead	7.2	14.1	19.6	23.2	25.1
miscellaneous overhead	3.6	7.2	10.0	11.8	12.8
users' useful computation	14.6	28.7	39.8	47.2	51.2

mtbpf = 31.9 msec, mpft = 30.9 msec, $K_{\ell}\bar{t}_{pl} = 9$ msec, $\delta = 0.25$, $\gamma = 1.2$, $p = 0.1$*

*It has been numerically found that the resulting performance is rather insensitive to a particular set of values of $\bar{t}_{in}$ and p if $\bar{t}_{in}/p$ = mtbpf is constant (see Section 3.4.1). Therefore, the value of p was arbitrarily chosen to be 0.1. (An explanation of these input variables can be found, for example, in Appendix A.)

tions summarized in Table 5-1.

For this purpose, a program was written in PL/I to evaluate the response time characteristics of the entire computer system under investigation using the total system model. We call this program the response time analysis program and it is included in Appendix B together with an explanation about how to use it and a sample console session using the program. The program has two phases: the first phase which determines the effective percentile throughput and the second phase which derives the resulting response time distribution.

Unfortunately, all of the data required to validate the entire total system model is not available on the Multics system. For instance, a response time distribution is not measured on the Multics system; one of the metering commands is capable of measuring only the average queue length as an indicator of the average system response time experienced by a population of interactive users. On the other hand, the PDP-8 user simulator can measure the response time distribution for a particular benchmark. Therefore, the average response time measured by the user simulator represents the average response time experienced by a population of all users only roughly. As for the input parameters needed by the response time analysis program, the values of θ_q(percentile throughput of the system under multiprogramming of degree $q(1 \leq q \leq q^*)$) are not all measured; only the value of $\theta(N)$ can be measured by a metering command (note that $\theta(N) \simeq \theta_{q^*}$ under a heavy user load). Therefore, we can at best examine the validity of the latter phase of the total system model partially.

In predicting the response time characteristics of each of those

three hardware configurations of the Multics system, the actual values of the percentile throughput and the number of users were supplied as input parameters to the response time analysis program. The average execution time needed for useful computation of a user's job was chosen to be 400 milliseconds long. The average think time of users was changed to see the effect upon the resulting average queue length (of user jobs) and average response time, for each of the above hardware configurations. The result is shown in Table 5-3 (a), (b), and (c). The predicted performance which is closest to the actual performance given in Table 5-1 is enclosed by a broken line for each configuration. It is observed that a close match between the predicted performance and the actual performance is obtained when the average think time of users is chosen to be 14 to 20 seconds long. This result of user think time is consistent with a general observation that the typical think time of the Multics system is approximately 15 to 25 seconds long. However, unavailability of a metering command for the measurement of users' think time for each experiment prevents us from examining more details of the validity of the total system model.

Table 5-3 Validation of the Total System Model

(a) one-processor two-primary-memory-unit configuration, i.e., small-scale configuration.

average think time (sec)	16	18	20	22
average queue length	20.3	17.7	15.1	12.7
average response time (sec)	15.7	13.7	11.7	9.8

percentile throughput = 51.8 % , average execution time of a job = 0.4 sec, number of users = 41.

(b) one-processor three-primary-memory-unit configuration, i.e., medium-scale configuration.

average think time (sec)	16	18	20	22
average queue length	17.5	14.3	11.4	8.9
average response time (sec)	10.9	9.0	7.2	5.7

percentile throughput = 63.9 % , average execution time of a job = 0.4 sec, number of users = 43.

(c) two-processor three-primary-memory-unit configuration, i.e., large-scale configuration.

average think time (sec)	14	16	18	20
average queue length	13.3	9.4	6.6	4.8
average response time (sec)	5.4	3.9	2.9	2.2

percentile throughput = 49.9 % , average execution time of a job = 0.4 sec, number of users = 48.

5.3. Performance Prediction

It is now fairly reasonable to expect that the models developed in this thesis can serve as a practical tool to predict the performance of a system in question whose configuration is specified. Therefore, we shall proceed to use these models in evaluating the effect of several important system parameters upon the processor time breakdown and the system response time.

5.3.1. Effect of System Parameters upon Throughput

In this section, we consider the probelm of improving the percentile throughput of the large-scale (standard) configuration of the Multics system, i.e., the two-processor three-primary-memory-unit configuration, as an example of investigating the effects of several important system parameters upon its percentile throughput. We assume, as our starting point, that the percentile throughput of this configuration (under multiprogramming of degree 4) is 47.2 percent, as shown in Table 5-2 (c). For the sake of simplicity, we do not change the degree of multiprogramming in this section.

There are several possible approaches to improve the percentile throughput of this configuration. For example,

(a) Addition of one more 128 k word primary memory unit

(b) Enhancement of secondary memory speed

(c) Reduction of system overhead time

Approach (a) aims to increase the mtbpf of user processes under multi-

programming by increasing the amount of primary memory available to each eligible user process. The resident Multics supervisor programs (including the I/O buffer, the memory space required by the page table, etc.) occupy approximately 90 k words of primary memory space. Then, using the linear paging model of Section 2.5.2., we can roughly expect the mtbpf to become about 45 milliseconds long if 128 k words are added to the current 384 k word primary memory. The longer mtbpf naturally decreases the percentage of both multiprogramming idle time and paging overhead time, and therefore increases the percentile throughput.

Approach (b) also aims to reduce the multiprogramming idle time, by having a shorter mpft. It is similarly possible to improve the percentile throughput of the system. The shorter mpft is usually attained by replacing the existing secondary memory device by a faster device. Another way which is applicable to a rotating device like a drum is to create multiple copies of each file on the device; the first copy accessed by the device head is read and transferred to primary memory, saving some part of the device's access time.[1] We assume that the current mpft can be somehow halved.

On the other hand, approach (c) aims to increase the percentile throughput not by decreasing the multiprogramming idle time as in the above two approaches but by reducing the system overhead time; the multiprogramming idle time will not be affected in this approach. We

---

[1] By creating multiple copies on the device we decrease the memory space of the device. This time-space tradeoff approach is being tested on the Multics drum for the purpose of reducing the mpft of the secondary memory system.

assume that the slow-down factor (K_ℓ) due to data-base lockout can be reduced from (the current value of) 1.5 to 1.2 by shortening each locking period of shared writable data-bases; this means that the mean paging overhead time ($K_\ell \bar{t}_{pl}$) would be approximately 7 ($\simeq 1.2 \times 6$) milliseconds long. The shorter mean paging overhead time may also be attained by reprogramming the page fault handler. Furthermore, we assume that the miscellaneous overhead coefficient (δ) can be reduced from the current value (0.25) to 0.2 by reprogramming the miscellaneous fault handlers.

The system performance resulting from each of these three approaches was then evaluated using the throughput analysis program. The result is summarized in Table 5-4. The expected change in processor time usage is clearly seen in each approach. It should be noted that approach (a) attains a nearly 10 percent improvement in percentile throughput (really a relative improvement of about 20 percent), while other approaches attain a 5 to 6 percent improvement. However, approach (a) may involve the highest cost; the decision of choosing the right approach must be made in consideration of the cost-performance. This aspect of configuration selection will be discussed later in Section 5.5.

Finally, it should be mentioned that there are other approaches to improve the system throughput. One could divide the primary memory into smaller memory units in order to decrease the percentage of memory interference idle time; the effect of having more memory units is seen even in approach (a) where the number of memory units is increased only by one. Another approach may be to add another processor to the existing processing system. The more attractive approach which does not

Table 5-4 Alternative Approaches to Improve System Throughput

Configuration	Current System	Approach (a)	Approach (b)	Approach (c)
No. of Processors	2	2	2	2
No. of PM units	3	4	3	3
Degree of Multiprogramming	4	4	4	4
mtbpf (msec)	31.9	45.0	31.9	31.9
mpft (msec)	30.9	30.9	15.5	30.9
Mean Paging Overhead Time (msec)	9	9	9	7
Misc. Overhead Coeff.	0.25	0.25	0.25	0.20
Memory Interference Coeff.	1.2	1.2	1.2	1.2
* processor time usage *				
Multiprogramming Idle	13.0 %	7.0 %	3.2 %	13.0 %
Memory Interference Idle	4.9	4.2	5.9	4.9
Paging Overhead	23.2	17.8	25.7	18.0
Miscellaneous Overhead	11.8	14.2	13.1	10.7
Users' Useful Computation	47.2	56.9	52.2	53.4

involve any increase of system cost is to optimize the degree of multiprogramming. This will be studied in detail in Section 5.4.1.

5.3.2. Effect of Percentile Throughput upon Response Time

In this section, we examine the quantitative effect of the percentile throughput upon the response time characteristics of a system so that we can determine the number of interactive users that can be supported by the system with a given configuration.

First, in order to present a rough idea about the response time characteristics, we evaluate only the average response times of the three configurations of the Multics system, for the range of the number of users that impose a heavy load upon the system. For this purpose, we need to specify only the value of the optimized percentile throughput (θ_{q*}) of each configuration. We assume therefore that the small-scale, the medium-scale, and the large-scale configurations of the Multics system have respectively the optimized percentile throughputs of 50, 65, and 50 percent. Then, noting that $\theta(N) \simeq \theta_{q*}$ in the above range of the number of users (see Section 4.3.2.), the average response time can be obtained by Eq. (4.3.9). As shown in Figure 5-?. the result shows that the average response time ($\bar{T}_r$) increases almost linearly with the number (N) of users. In fact, the asymptote for the average response time can be directly obtained from the result derived by Scherr [S4] for his simpler model, as follows:

$$\bar{T}_r \simeq \frac{N}{m} \cdot \frac{\bar{T}_e}{\theta_{q*}} - \bar{T}_t$$

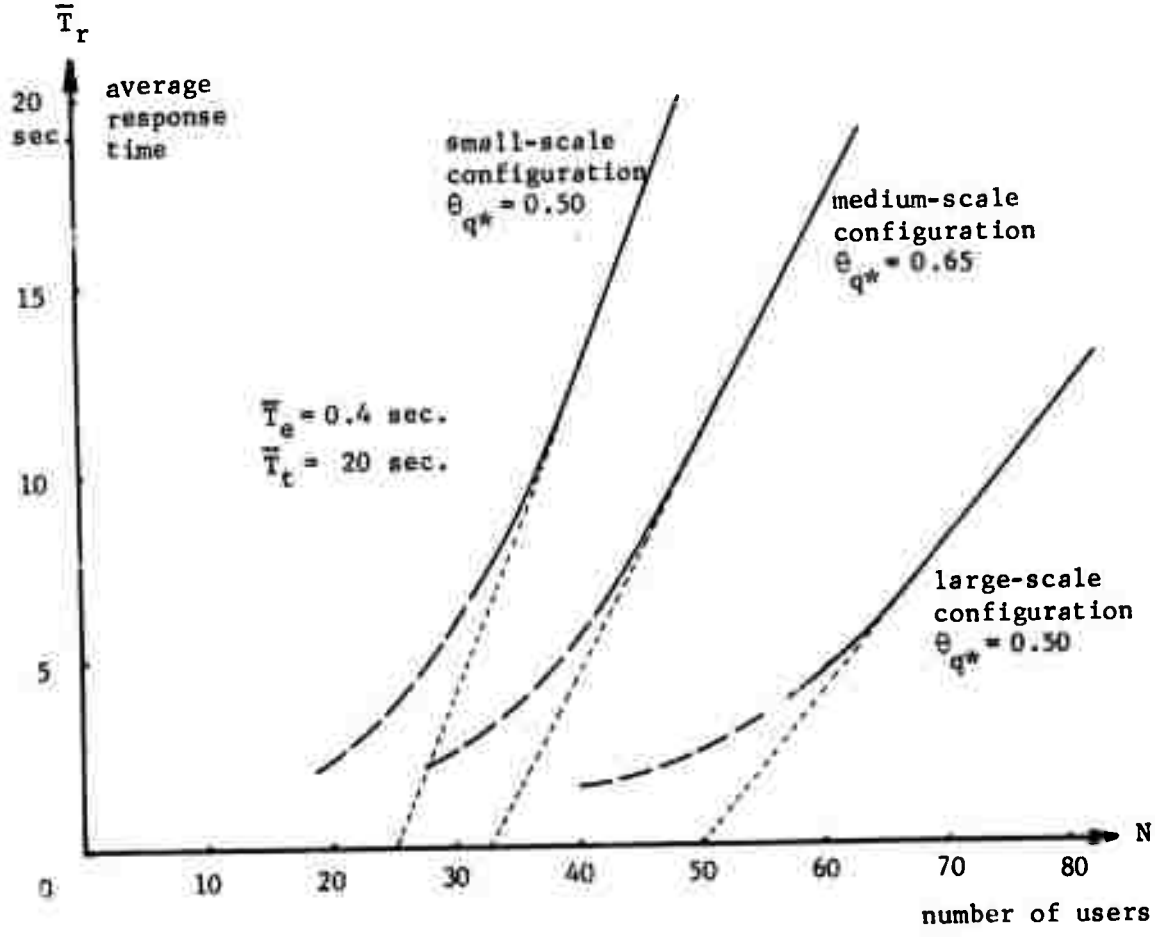


Figure 5-2 Effect of Percentile Throughput upon Average Response Time

---

The broken line part of each curve represents a lower-bound estimate of the average response time. Without specifying the values of θ_q for all q $(1 \leq q \leq q^*)$, the average reponse time cannot be accurately determined for the medium or light load range of N.

where m, $\bar{T}_e$, and $\bar{T}_t$ are respectively the number of processors, the average execution time of each user's job (excluding all the system overhead times), and the average think time of each terminal user. It must be noted that the slope of the asymptote is inversely proportional to the optimized percentile throughput. This means that maximization of optimized percentile throughput leads to minimization of the average response time.

Next, we proceed to evaluate the distribution of response time for the entire range of the number of users. For this purpose, we need to specify all the values of θ_q $(1 \leq q \leq q^*)$ of the system under study. As an example, we consider the medium-scale configuration of the Multics system, assuming that

$$q^* = 4$$

$$(\theta_1, \theta_2, \theta_3, \theta_4) = (0.55, 0.63, 0.65, 0.65)$$

$$\bar{T}_e = 0.4 \text{ seconds.} \quad \bar{T}_t = 20 \text{ seconds.}$$

Then, using the first phase of the response time analysis program, the effective percentile throughput $\theta(N)$ was calculated, as shown in Table 5-5. It is seen that $\theta(N) \simeq \theta_1$ under a light load, but as interactive users begin to impose a heavier load upon the system $\theta(N)$ gradually approaches θ_{q^*}. In Figure 5-3, the average, 10, and 90 percentile response times obtained using the second phase of the response analysis program are shown as a function of the number of interactive users. It is observed in this example that the 90 percentile response time is about twice as long as the corresponding average response time. If 90 percentile response time must be kept under ten seconds, the medium-

Table 5-5 Predicted Effective Percentile Throughput

Number of Users N	1	10	20	30	40	50
Effective Percentile Throughput $\theta(N)$	.550	.576	.604	.629	.645	.650

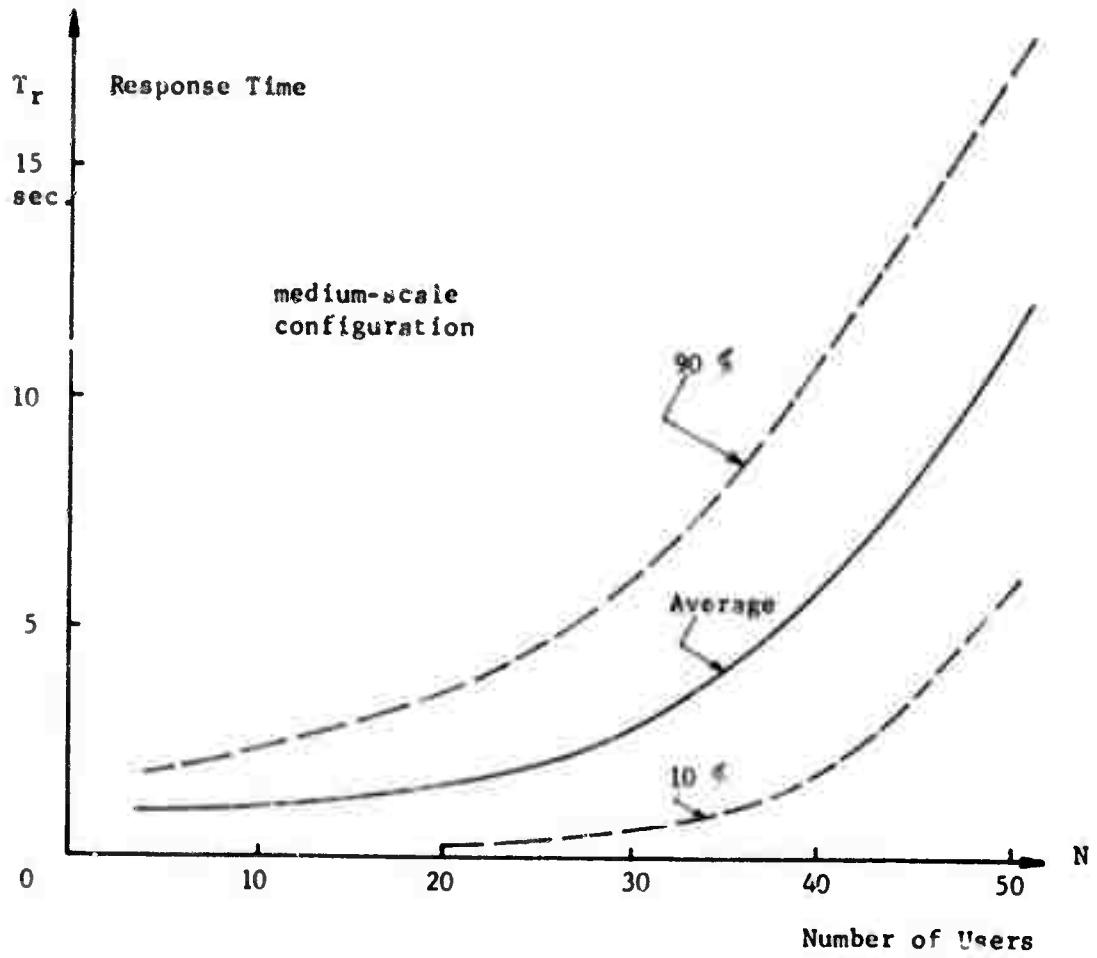


Figure 5-3 Average, 10, 90 Percentile Response Times

scale Multics configuration can support at most 39 interactive users[1].

Finally, the probability density and the probability distribution function obtained by the response time analysis program are shown respectively in Figures 5-4 and 5-5. Figure 5-4 clearly shows that the probability density gradually flattens out as the number of interactive users increases; if the number of users is very small (e.g., ten users) the response time distribution is similar to the distribution of a job's execution time (i.e., an exponential distribution) in shape, but if the number of users is large enough to impose a heavy load upon the system (e.g., fifty users) the response time distribution is more like a normal distribution. This tendency of the response time distribution of the Multics system has been actually measured by the PDP-8 user simulator also. Figure 5-5 gives the percentile response times for each value of the number (N) of users. For example, it is seen that if the system has 40 interactive users 80 percent of the response times fall between 1.4 seconds and 10.2 seconds (or below 8.5 seconds) with 5.1 seconds as a median. These results predicted by the response time analysis are generally consistent with the casually observed performance of the Multics system.

---

[1]The actual system of this configuration currently (1971-72) supports up to 45 users. But these numbers may change with time as the system characteristics change.

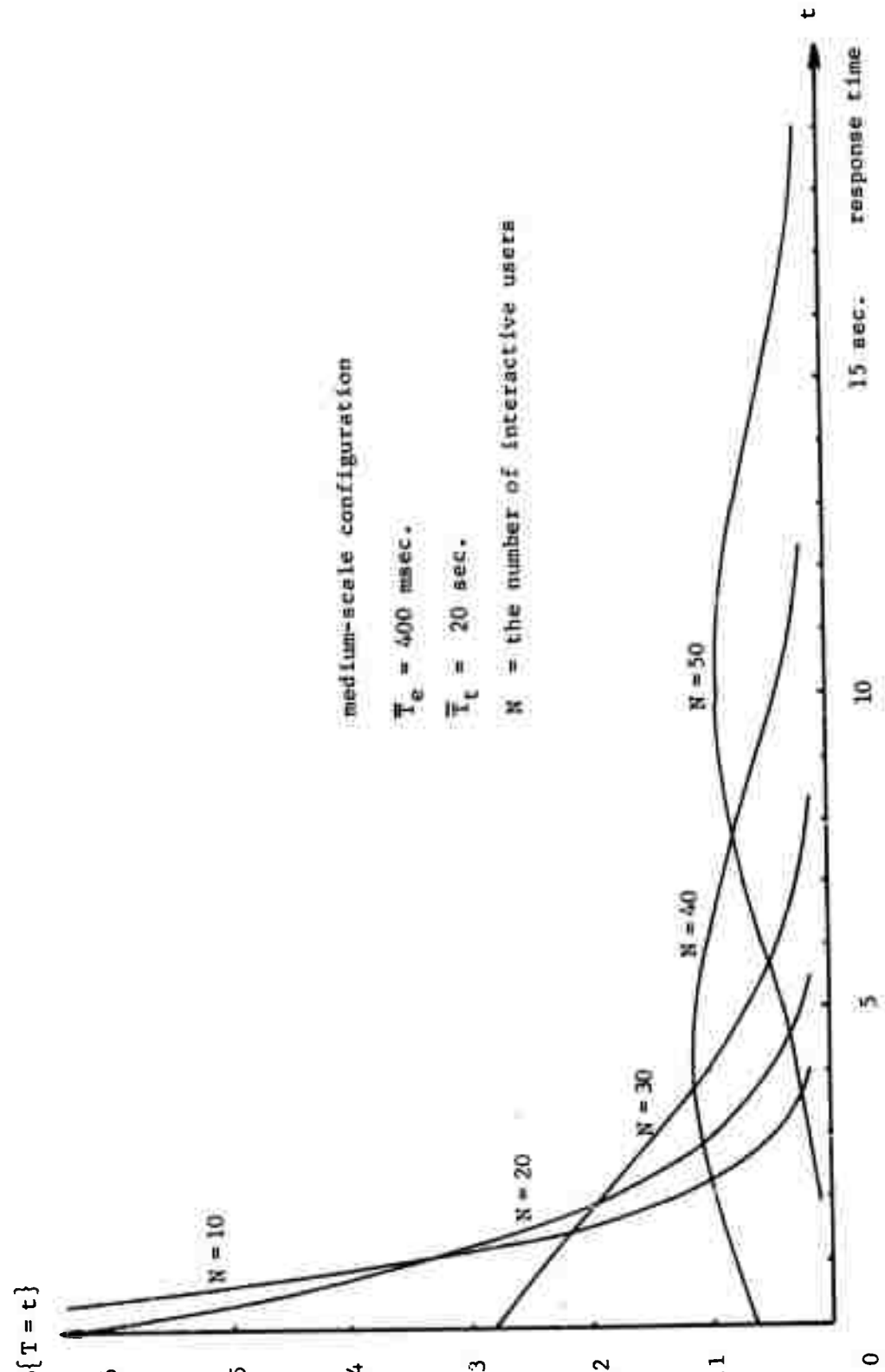


Figure 5-4 Probability Densities of Response Times

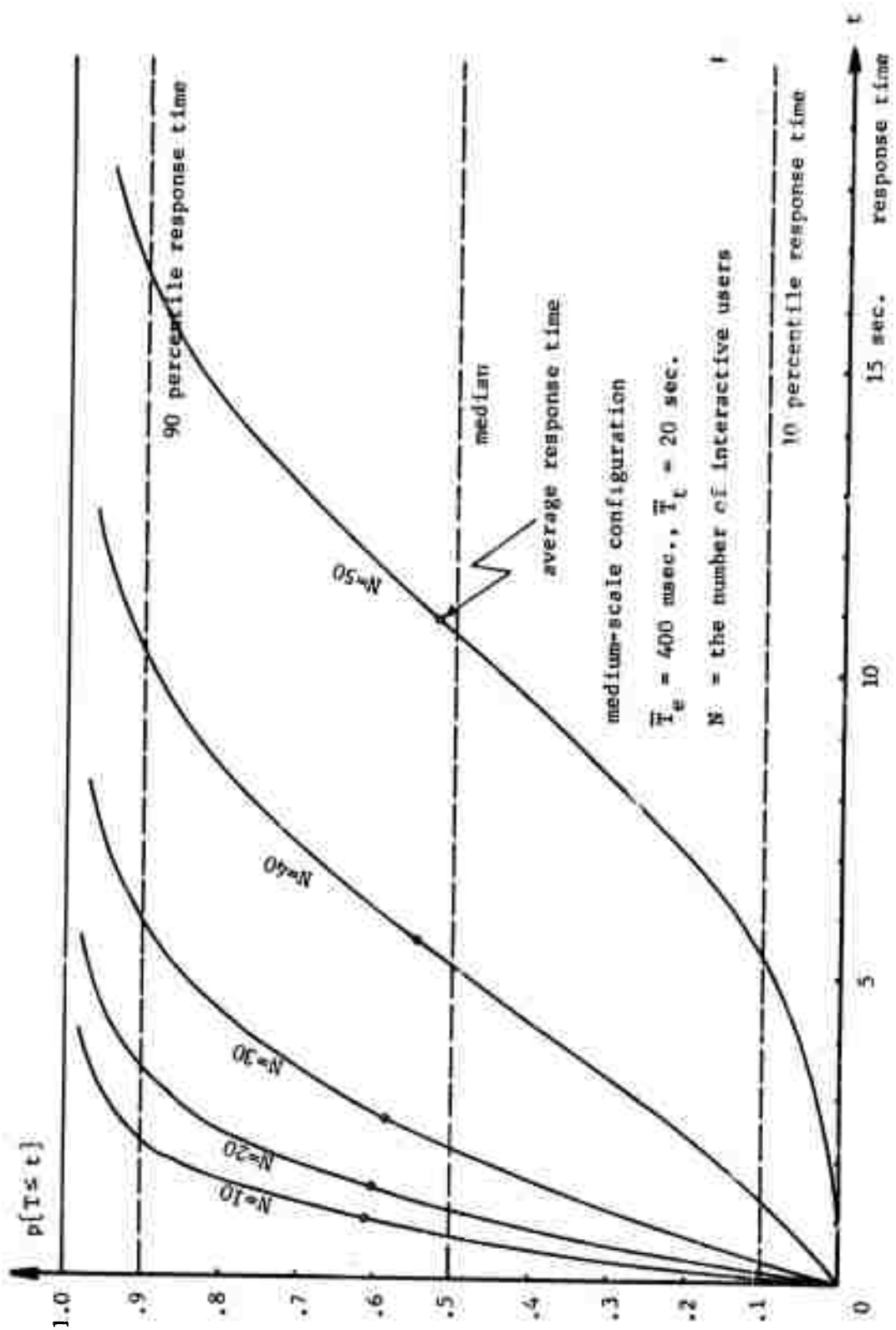


Figure 5-5 Probability Distribution Functions of Response Times

5.3.3. Effect of User Characteristics upon Response Time

An interaction cycle of a terminal user consists of the user's think time (T_t), a waiting time experienced by the user's job in the job queue of the computer system ($T_w = T_r - T_e$), and an execution time required by the user's job (T_e). Therefore, the user's need for processor time per unit real time is given by $T_e/(T_t + T_r)$. The system response time T_r is a complicated function of various system parameters (including T_t and T_e) as seen in Chapter 4, but T_t and T_e are (purely) user characteristics and are both prime factors determining a user's need for processor time per unit real time.

We will therefore examine the effect of these two parameters concerning user characteristics upon the average response time, for the range of the number of users that impose a heavy load upon the system. In particular, we will consider the medium-scale and large-scale configurations of the Multics system, assuming that these configurations have 65 and 50 percent as their optimized percentile throughput respectively.

The result obtained by the response time analysis program is graphically summarized in Figures 5-6 and 5-7. From these figures, it becomes clear that a twenty-five percent change in the value of each of these user parameters produces more than several users as a change in the number of interactive users that can be supported by the system (use five-second average response time as a criterion, for example). Furthermore, it should be noted that in this heavy load range of N the response time is very sensitive to these two parameters (T_t, T_e) and the number of users. This indicates the importance of a dynamic load controller on time-shared computer system with interactive users.

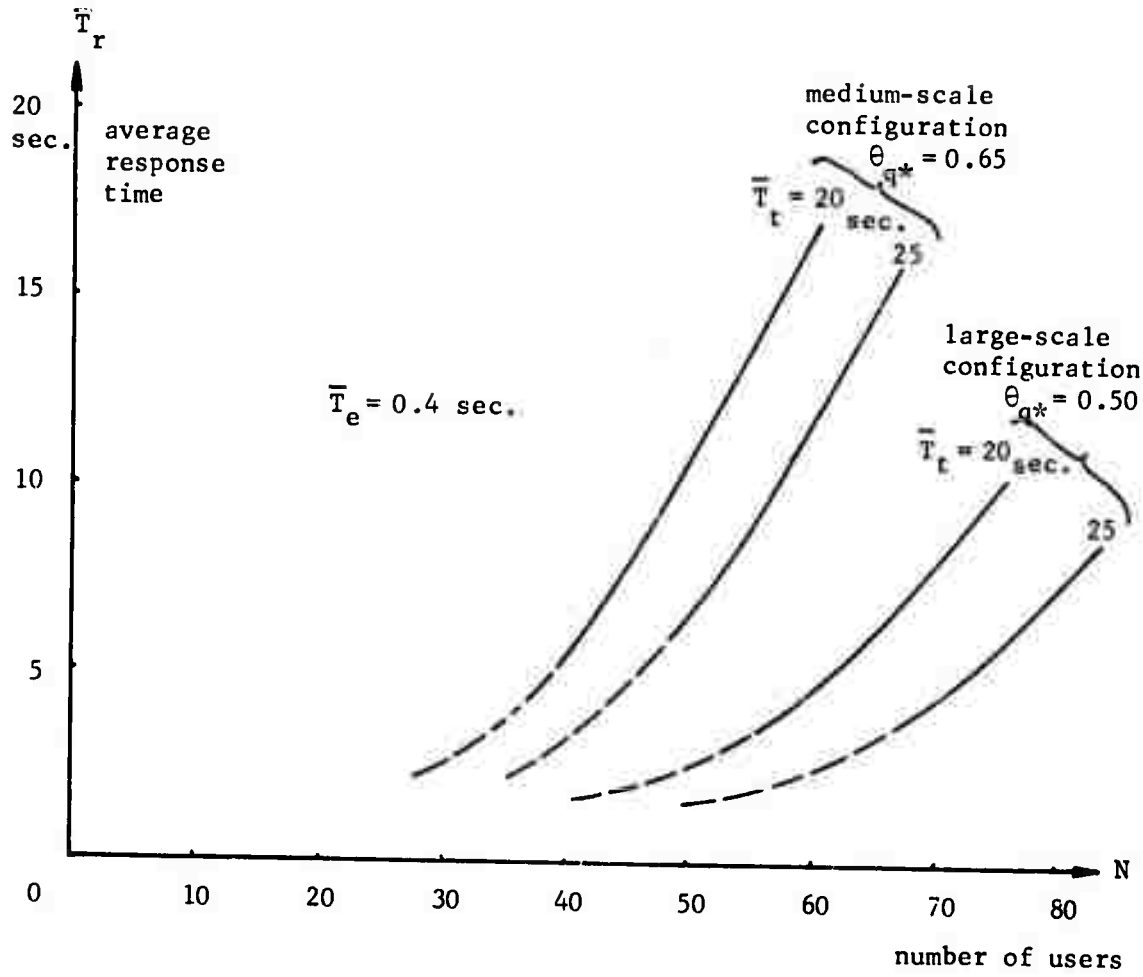


Figure 5-6 Effect of User's Think Time upon Average Response Time

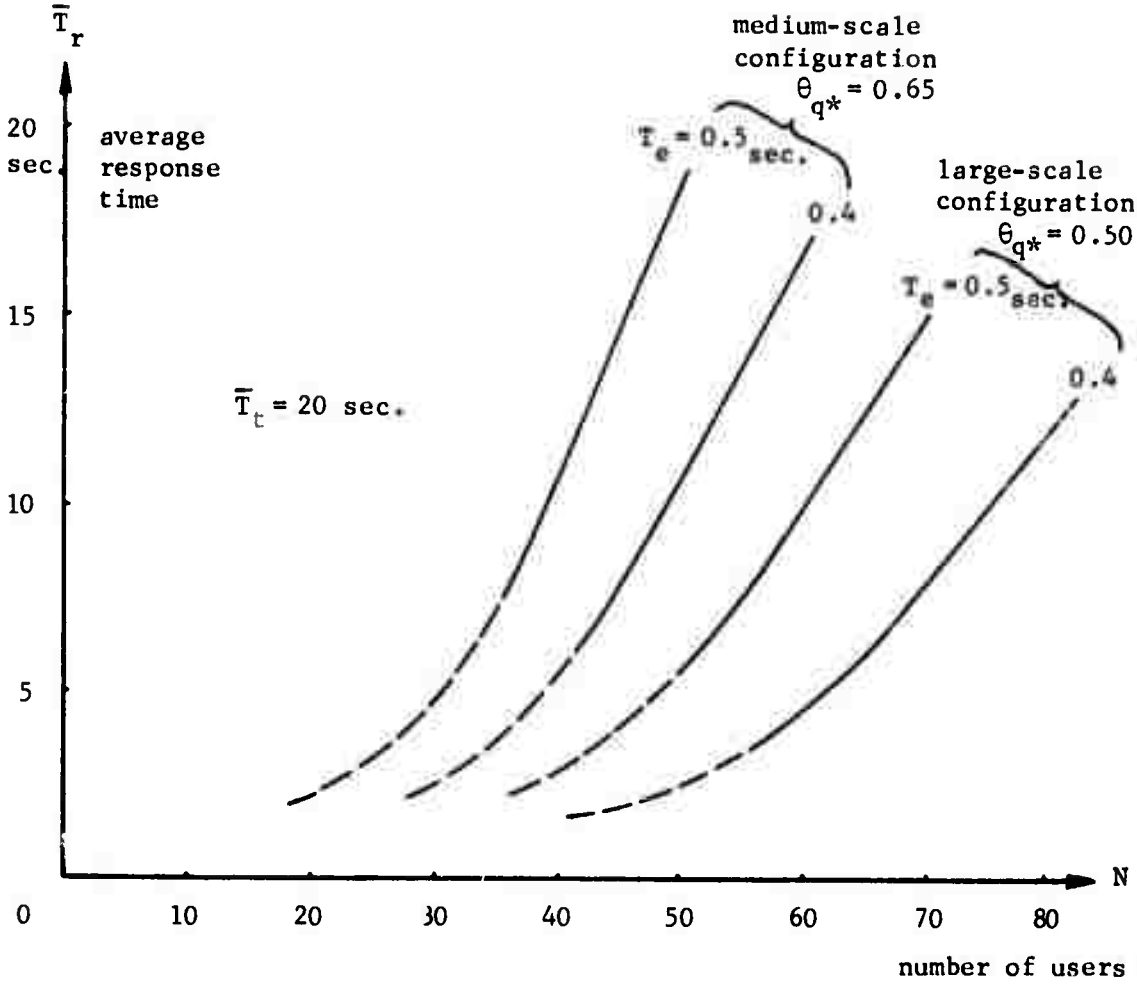


Figure 5-7 Effect of Execution Time of User's Job upon Average Response Time

5.4. Performance Optimization

Now we shall proceed to consider the performance optimization problems, i.e., the problems of optimizing the throughput of a given configuration with respect to certain adjustable parameters of the operating system, without changing the hardware configuration of the system (the system cost is held constant). In particular, this section presents the result of performance optimization of the degree of multiprogramming and then proposes a promising approach to the page size problem (the problem of determining the optimum page size).

5.4.1. Optimization of Multiprogramming Algorithm

Assuming that a hardware configuration of the system under study is given, this section is concerned with the problem of optimizing the degree of multiprogramming in such a way that the throughput of this configuration is maximized under a heavy load. Because the system throughput is linearly proportional to the percentile throughput, the above problem is equivalent to that of maximizing the percentile throughput by choosing the optimum degree of multiprogramming.

As suggested in Section 3.5., it is reasonable to expect that the percentile throughput as a function of the degree of multiprogramming shows a uni-model curve like the one shown in Figure 3-6; if the degree of multiprogramming is too small the multiprogramming idle time would dominate the processor time breakdown and if it is too large the paging overhead operations could dominate the processor time breakdown. We will investigate the performance of a particular configuration, i.e.,

a two-processor three-primary-memory-unit configuration with the following characteristics.

mean page fetch time (mpft) = 35 msec

mean paging overhead time ($K_{\ell}\bar{t}_{pl}$) = 7 msec

miscellaneous overhead coefficient (δ) = 0.25

memory interference coefficient (γ) = 1.2

missing-page probability (p) = 0.1

We assume a particular linear paging behavior of programs obtained in Table 2-2 of Chapter 2. All of the three cases concerning sharing included in Table 2-2 are considered. Those three cases are:

Case I: no sharing of non-resident programs
($a = 0.1$, $b = 0$)

Case II: 10 percent sharing of non-resident programs
($a = 0.1$, $b = 0.1$)

Cast III: 20 percent sharing of non-resident programs
($a = 0.1$, $b = 0.2$)

Changing the degree (q) of multiprogramming, the processor time breakdown of the above configuration was repeatedly computed using the throughput analysis program, for each of these three cases concerning sharing.

The result is summarized in Table 5-6. It is clearly seen in each case that the percentile throughput as a function of q shows a uni-model curve. The multiprogramming idle time and the paging overhead time are the two major components determining this uni-modal tendency of the percentile throughput. For case I, the optimum degree of multi-

Table 5-6 Optimization of Percentile Throughput of a Dual Processor Configuration

Case I: No Sharing of Non-Resident Programs

degree (q) of multiprogramming	2	3	4	(5)	6	7	8
(1)	7.6	12.6	17.3	21.5	25.2	28.4	31.1
(2)	10.3	11.7	12.4	12.6	12.6	12.4	12.1
(3)	38.1	24.6	15.9	10.0	6.0	3.5	1.9
(4)	2.4	3.7	4.6	5.2	5.6	5.8	6.0
(5)	41.4	47.1	49.6	[50.5]	50.4	49.7	48.7

Case II: 10 Percent Sharing of Non-Resident Programs

degree (q) of multiprogramming	2	3	4	5	(6)	7	8
(1)	7.4	12.0	16.0	19.3	22.0	24.0	25.7
(2)	10.5	12.2	13.1	13.5	13.6	13.6	13.4
(3)	37.1	22.6	13.2	7.3	3.7	1.7	0.7
(4)	2.4	3.9	4.8	5.4	5.8	6.0	6.1
(5)	42.3	49.1	52.6	54.2	[54.6]	54.4	53.8

Case III:20 Percent Sharing of Non-Resident Programs

degree (q) of multiprogramming	2	3	4	5	6	(7)	8
(1)	7.2	11.3	14.7	17.1	18.9	20.1	21.0
(2)	10.8	12.8	13.8	14.3	14.5	14.5	14.4
(3)	36.0	20.5	10.8	5.1	2.2	0.8	0.2
(4)	2.5	4.1	5.1	5.6	6.0	6.1	6.2
(5)	43.3	51.2	55.4	57.5	58.2	[58.3]	57.9

mpft = 35 msec, mean paging overhead time ($K_\ell \bar{t}_{pl}$) = 7 msec, $\delta = 0.25$, $\gamma = 1.2$, $p = 0.1$.

processor time breakdown: (1) Paging Overhead, (2) Miscellaneous Overhead, (3) Multiprogramming Idle, (4) Memory Interference Idle, (5) Users' Useful Computation (all in percent).

programming is obtained to be five ($q^* = 5$) with the maximized percentile throughput amounting to 50.5 percent ($\theta_{q^*} = 0.505$). It is seen that the percentile throughput can be improved by nearly 10 percent ($\simeq 50.5\text{-}41.4$) by using the optimum multiprogramming mechanism instead of the uni-programming mechanism (i.e., $q = 2$). By comparing the results of the above three cases concerning sharing, the effect of sharing upon the optimized percentile throughput is clearly seen; the gain due to twenty-percent sharing is approximately 8 percent in absolute percentile throughput. It is also observed that as the degree of sharing increases the optimum degree of multiprogramming tends to increase. This tendency is due to the fact that the larger degree of multiprogramming will not shorten the mtbpf of the eligible user processes too much if there exists a considerable amount of sharing among these processes. Thus, the optimum degree of multiprogramming for the above two-processor configuration was found to be five to seven, for the assumed program behavior. This result is surprisingly consistent with the fact that the Multics system with the same hardware configuration is tuned to allow five to six user processes to be eligible with its dynamic multi-programming algorithm.

Finally, in order to examine the effect of the number of processors upon the optimization of the multiprogramming algorithm, the performance of a single processor configuration was evaluated. The system characteristics were similarly specified as mpft = 35 msec, $\delta = 0.25$, $\gamma = 1.2$, and $K_\ell \bar{t}_{pl} = \bar{t}_{pl} = 6$ msec. The same linear paging behavior of programs, with a=b= 0.1 (ten-percent sharing), was assumed. The result is

summarized in Table 5-7. It is seen that the optimized percentile throughput amounting to 65.5 percent is attained when the degree of multiprogramming is chosen to be three. Comparing this result with the performance of the dual processor configuration that we have considered, we find that an addition of a processor to this single processor configuration would make the system throughput 1.67 (= 2 x 0.546 / 0.655) times as large as that of the original single processor configuration; apparently, the system throughput cannot be doubled because there exists multi-processor interference (note that the size of primary memory is 384 k words for both configurations.). All of these observations are again surprisingly consistent with the actual measurement results of the Multics system both qualitatively and quantitatively.

However, a comment is in order about the optimum degree of multiprogramming. A considerable difference is seen between the optimum degrees of multiprogramming of the two corresponding (single processor and dual processor) configurations. This means that the number of processors plays an important role in optimizing the multiprogramming algorithm. On the other hand, the current Multics operating system determines the degree of multiprogramming using only a (crude) working-set estimate of user processes against the available size of primary memory; the Multics system is currently tuned to allow as many as five or six user processes under multiprogramming on both the large-scale and the medium-scale configurations. (Note that both configurations have the same amount of primary memory.) Therefore, our observation about the role of the number of processors in optimizing the system throughput

Table 5-7 Optimization of Percentile Throughput of a Single Processor Configuration

degree (q) of multiprogramming	1	2	(3)	4	5	6	7	8
mtbpf (msec)	98.5	55.9	41.7	34.7	30.5	27.7	25.8	24.3
(1)	4.4	9.5	13.7	17.0	19.5	21.6	23.2	24.6
(2)	13.8	15.9	16.3	16.2	15.9	15.6	15.3	15.0
(3)	26.2	10.7	4.3	1.5	0.5	0.1	0.0	0.0
(4)	0	0	0	0	0	0	0	0
(5)	55.4	63.7	65.5	65.1	63.9	62.5	61.3	60.2

mpft = 35 msec, mean paging overhead time ($\bar{t}_{pl}$) = 6 msec, $\delta = 0.25$
processor time breakdown: (1) Paging Overhead, (2) Miscellaneous Overhead, (3) Multiprogramming Idle, (4) Memory Interference Idle, (5) Users' Useful Computation (all in percent).

suggests a possible change of the Multics multiprogramming algorithm such that the number of processors, i.e., a source of raw computing power, is also considered in its working-set strategy in determining the degree of multiprogramming.

Thus, we have evaluated the percentile throughput of a given hardware configuration, for each degree of multiprogramming, in this section. This result makes it possible to derive the response time distribution of this configuration, as the next step. In fact, we have already evaluated the response time distribution of the optimized single processor configuration that we have just analyzed, earlier in Section 5.3.2. (See Figures 5-3, 5-4, and 5-5.)

5.4.2. Optimization of Page Size

In this section, we will consider the page size problem, i.e., the problem of determining the optimum page size. This problem is well discussed in Denning's tutorial paper on virtual memory [D2]. The author claims that a choice of page size should be made considering memory fragmentation and efficiency of page-transfer operations between primary and secondary memories. Consideration of memory fragmentation suggests the use of small page size (e.g., 45 words) and that of page-transfer efficiency suggests the use of large page size (e.g., 10^3 - 10^4 words). The author however does not present any method to find a trade-off between these two conflicting factors which may give the optimum page size. It is apparant that there existed no vehicle to accomodate

these two conflicting factors in a unifying framework which allows the determination of the optimum page size. However, the framework of performance evaluation developed in this thesis fortunately provides us with such a vehicle.

We assume that the configuration of the system is completely specified except the page size, and attempt to evaluate the effect of page size upon the percentile throughput of this system so that we can derive the page size which maximizes the throughput of the system under study.

Two types of memory fragmentation exist on a virtual memory computer system using paging: internal fragmentation and table fragmentation. The former represents the wasted space in the last page of each segment (the memory requirement of each segment must be rounded up to an integral number of pages) and the latter represents the space required by storing the page table in the area of primary memory reserved for the resident supervisor. Memory fragmentation reduces the size of primary memory available to non-resident programs (see Figure 2-1), and therefore paging activities of user processes may be intensified. Given the page size, it is straightforward to estimate the total wastage of primary memory space due to both types of memory fragmentation.

The effect of page size upon page-transfer operations can be measured in the page fetch time associated with each secondary memory device. For rotating devices like drums and disks the effect of page size upon the page fetch time is rather slight because of their relatively long access time, but for a bulk core memory used as secondary memory the page fetch time is linearly proportional to the page size [D2].

On the other hand, it is clear from the result of Section 2.3 of Chapter 2 that programs' paging behavior is also affected by the page size. Therefore, we must consider the effect of page size upon the mhbpf (or mtbpf) of user processes, besides those two factors mentioned by Denning. It was seen in Section 2.3 that the smaller page size yields a longer mhbpf in a program's steady-state behavior. However, in its transient-state, i.e., while the program has not yet fetched pages that are necessary for a sound progress of its computation into primary memory, the smaller page size would yield a shorter mhbpf. For example, if we measure the mhbpf of a process during the period which terminates at the moment when all the page-frames available to the process become occupied by its pages in a partitioned primary memory, then the mhbpf on the system with the halved page size would be roughly half of the mhbpf on the original computer system. This argument suggests that the larger page size would be favorable to programs with relatively short execution times which can operate comfortably in a relatively large primary memory space; on the other hand, the smaller page size would be favorable to programs with relatively long execution times which must operate within a relatively small primary memory space.

Unfortunately, none of the program models developed in this thesis can quantitatively evaluate the effect of page size upon the overall mhbpf of user processes. The experimental result obtained by Baer [B1] indicates that mhbpf is maximum when the page size is 128 to 256 words. However, interpretation of "numbers" obtained in experimental studies is (at best) very tricky because all the conditions of these studies are not explicit; it is much more difficult to derive a parameterized

expression of mhbpf (which can be applied to any other system) as a function of page size, memory size, and program characteristics, from the experimental studies. Therefore, a comprehensive analytical model is much desired in this field. In this respect, a program behavior model of Woolf [W3] is worthy of attention, but the validity of this model has not been examined.

Thus, if the overall mhbpf (or mtbpf) can be properly evaluated as a function of those system parameters, we can evaluate the effect of page size upon the percentile throughput of the processing system, using the analysis results of mpft (of secondary memory) and memory fragmentation as well as that of the overall mhbpf of user processes. Therefore, it would be possible to determine the page size which maximizes the throughput of the computer system under study.

5.5. Configuration Selection

Configuration selection is the problem of selecting the configuration which attains the best performance for a given (purchase or rental) budget. By now, a solution to this problem may be clear.

The process of selecting the best performance configuration within a given budget is schematically depicted in Figure 5-8. This process essentially contains the following three stages:

(1) Choice of a hardware configuration

(2) Performance optimization for this configuration

(3) Application of a decision rule to the cost-performance of each configuration

In the first stage, a particular hardware configuration is chosen from a finite set of possible hardware configurations of the processing system, as a candidate for the best configuration. Different hardware configurations represent different number of hardware components (processors and primary memory units), different size of primary memory, different speed of hardware components (processors, primary memory, and secondary memory) and so on. In the second stage, the percentile throughput of the chosen hardware configuration is optimized with respect to certain adjustable parameters of the operating system (e.g., the degree of multiprogramming, the page size, etc.). The optimized (hardware-software) configuration as well as its performance is recorded as the best performance which can be obtained from that hardware configuration. Repeating these two stages for each hardware configuration whose cost is within the budget, we can obtain the maximized performance

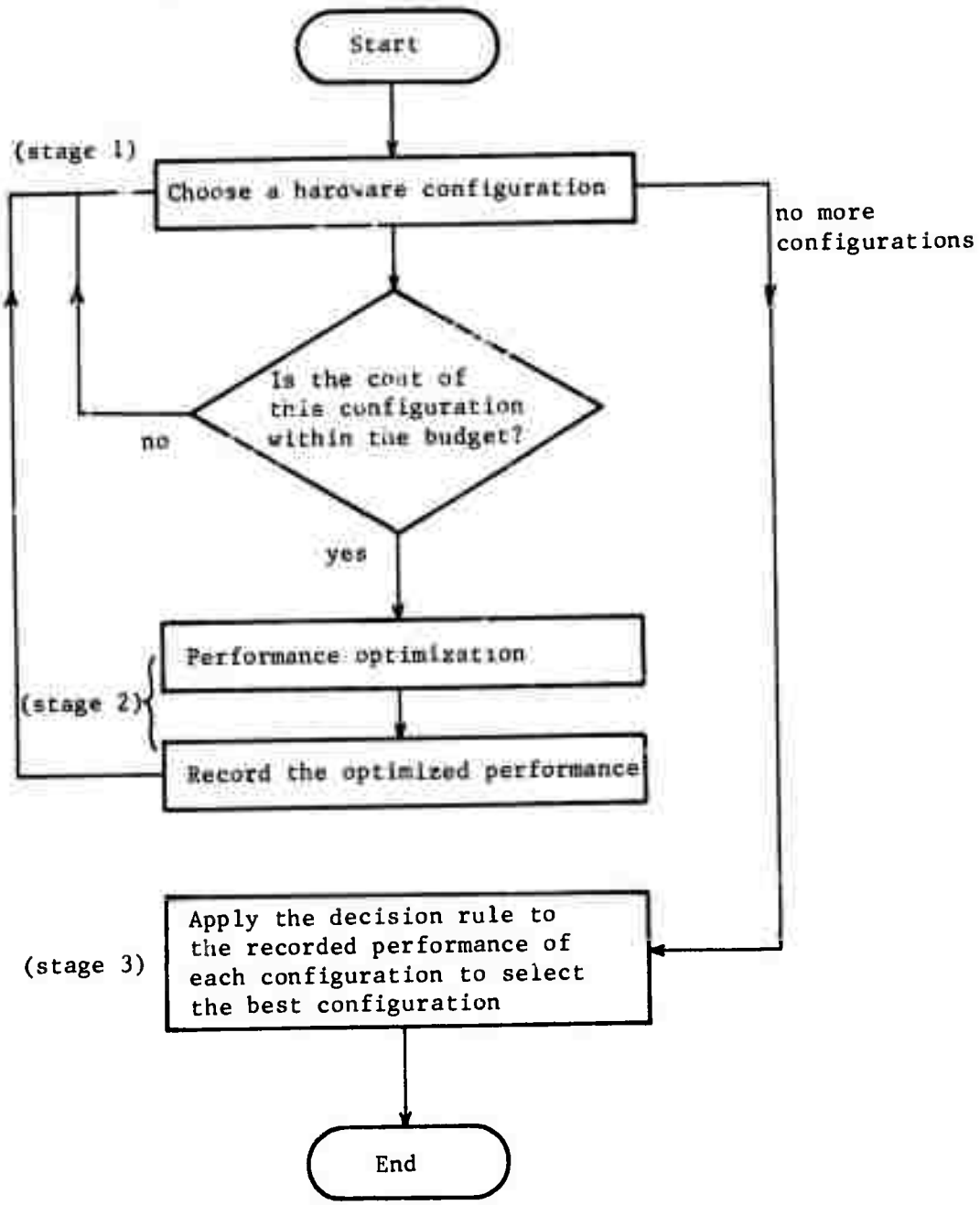


Figure 5-8 Configuration Selection Process

for each candidate hardware configuration of the processing system.

When all the candidate configurations of the processing system are evaluated in the above optimization stage, the result is passed over to the third and last stage of the configuration selection process. In this stage, the best configuration is selected according to a decision maker's formula concerning the cost-performance of the system; the decision maker may use a simple ratio of cost and performance or his own complicated formula concerning cost and performance. This decision process is entirely up to the decision maker. When he has selected a particular configuration in this way, he obtains the best configuration of the system under study which is within a given budget.

CHAPTER 6

CONCLUSIONS

Modern large-scale time-shared computer systems have become so complicated in their structure and performance that human intuition often cannot foresee the impact of a small change in their structure upon their performance. This thesis has focused its attention on the statistical performance of these systems; the system throughput and the system response time have been selected as the performance measures of these systems. Architects and designers of these computer systems have encountered many performance questions that they found very difficult to answer quantitatively (some examples are shown in Table 1-1). They did not have a useful tool to tackle these performance questions; they definitely needed a comprehensive structured framework of performance evaluation which serves as an aid to guide their intuition in understanding variety of performance evaluacion problems; they wanted a methodology of performance evaluation which can quantitatively answer performance

questions such as those given in Table 1-1.

For variety of good reasons, we have decided to explore the possibility of using analytical models in attacking these performance evaluation problems earlier in this thesis. Because these modern large-scale time-shared computer systems involve many important features which are believed to influence their statistical performance significantly, we faced a dilemma of two basically conflicting factors concerning analytical models of these computer systems: a multiplicity of important system parameters and the mathematical tractability of these models. As a solution to this dilemma, we have presented an approach using a set of hierarchically organized modular models (see Figure 1-4); by developing a model for each subsystem of the entire computer system and then combining these models together through their inter-relationships, we can realistically model actual modern computer systems. In this approach, we can evaluate the effect of a small change in one subsystem (e.g., secondary memory) upon the overall performance measures of these systems (throughput and response time distribution); we can consider all the major features of these modern computer systems such as paging, segmentation, multiprogramming, multi-processing, memory hierarchy, etc.; we can consider several kinds of system overhead times and idle times which reduce the computational capacity of these computer systems. In modeling these modern computer systems, we have however abstracted the actual structure of these systems in a certain way; because this thesis is concerned with how a system configuration (concerning hardware, system programs, user programs, and users) affects its overall performance, only the system parameters which are believed to influence the

system throughput and the average response time were considered in tackling performance projection problems (the problems of estimating the performance of a system which does not yet exist) of these computer systems.

Performance projection problems were classified into (1) performance prediction problems, (2) performance optimization problems, and (3) configuration selection problems, in this thesis. In attacking the performance prediction problems of these computer systems with an approach using a set of hierarchically organized modular models, the thesis has proposed a particular hierarchy of modular models depicted in Figure 1-4. The hierarchy contains (1) a user behavior model, (2) a secondary memory model, (3) a program behavior model, (4) a processor model, and (5) a total system model. Because the last three models were particularly felt to be underdeveloped, this thesis developed stochastic models in these three areas. Chapter 2 described several program behavior models which evaluate the effect of various important system parameters affecting a program's paging behavior (measured in the mean time between page faults). In praticular, the macroscopic paging performance model of Section 2.5 turned out to be useful in the sense that the analysis result of this model can be easily combined into the analyses of other subsystems. Chapter 3 presented the throughput analysis of the multi-processor multi-memory processing system, i.e., the derivation of the processor time breakdown for a processing system of a given configuration; because the system throughput is linearly proportional to the percentile throughput of the system (the percentage of the system's computational capacity used for users' useful work), as

shown in Section 3.2, the system throughput can be directly obtained from the result of the processor time breakdown. Chapter 4 then proceeded to evaluate the response time characteristics of the entire computer system, using the results obtained by the analyses of all other models in the hierarchy of models. The analysis presented in this chapter has explicitly derived the probability distribution of response time of these computer systems; from this result, we can determine the number of interactive users that can be supported by the system of a given configuration with an assurance that the 90 percentile response time is less than ten seconds, for example.

Finally in Chapter 5, the validity of the processor model and the total system model was examined by comparing their behavior with the behavior of an actual system, i.e., the Multics system of M.I.T. This comparison has shown that the performance predicted by these analytical models is consistent with the actual performance of three configurations of the Multics system. Then, various performance prediction problems and performance optimization problems such as those mentioned in Table 1-1 were numerically studied. The predicted results look very reasonable and the result of optimization concerning the degree of multiprogramming suggests a possible change of the current Multics multiprogramming algorithm. At the end of Chapter 5, the configuration selection problem (the problem of selecting the best configuration for a given budget) was finally considered.

The structured framework of performance evaluation presented in this thesis gives system analysts a vehicle to tackle their performance problems. When investigating the effect of a certain system parameter

upon the performance of the system under study, a general approach suggested by this thesis research is to examine its effect on all the input parameters and intermediate performance measures included in the hierarchy of models shown in Figure 1-4; system analysts can then make their intuition work in the right direction in figuring out the effect of that system parameter upon the overall system performance. A good example was presented in Section 5.4.2, when we considered how the effect of page size upon the throughput of the system under investigation can be possibly evaluated in an effort to determine the optimum page size for that system.

Many numerical results concerning processor time breakdown and response time characteristics of the computer system of various configurations presented in Chapter 5 have shown that a set of analytical models developed in this thesis is capable of providing reasonably accurate answers to the quantitative performance questions with which computer architects and designers must cope; these models can answer most of the basic questions concerning the throughput and the response time of multiprogrammed virtual-memory time-shared computer systems using demand paging, for a wide range of system configurations. Therefore, the author of this thesis believes that analytical models are extremely useful especially in tackling the performance projection problems which computer architects and designers must resolve in early design stages. Many modern large-scale computer systems continue to evolve both in performance and in supporting hardware-software structure. For these systems, analytical models may be valuable in all stages of their life. However, in studying some of the detailed problems like a comparison of

several specific resource allocation strategies or paging performance of computer programs of acutal systems, simulation approaches would be more useful in practice. Most of these detailed problems usually arise later in the design stage. It is known that the studies of these detailed problems in simulation approaches tend to be very expensive. If this is the case with a particular problem, an analytical approach like that presented in this thesis should be used in deriving some sub-optimal solutions to the problem under investigation which are to be studied by a detailed simulation approach, in order to reduce the operating cost of the simulation studies. Measurement of actual systems is almost indispensable to any problem as a source of information verifying the expected performance or as a source of information indicating a need for a design change of the computer system under study.

In short, the approach using a set of hierarchically organized modular models is usable for actual performance evaluation problems. This approach guides one's intuition to understand the cause-and-effect relationship existing in a complicated structure of modern large-scale computer systems. When the analyses using these models are applied, a quantitative solution with a reasonably accurate approximation to the performance problem concerning system configuration can be obtained. Furthermore, this modular modeling approach allows one to use partially available information (obtained by actual measurement) about system performance effectively in estimating the overall performance of the system.

Finally, we would like to turn our attention to the specific performance problems considered in this thesis to find the problems

which require further research. In the area of program behavior analysis, it is generally felt that more measurements of paging behavior of actual programs are needed. Since the primary memory size is the single most influential parameter determining the paging behavior of programs, more measurement examining the validity of the linear paging model is absolutely necessary. Measurement of sharing is virtually unexplored and needs to be carried out. To study the page size problem (the problem of deriving the optimum page size) fully, we need a comprehensive analytical model to evaluate the effect of page size upon the mean time between page faults, as discussed in Section 5.4.2. In the area of throughput analysis, it was observed in Chapter 3 that the analysis of a dual processor system is fairly complicated and that of a many-processor system (e.g., a ten-processor system) would be of staggering complexity; it is felt that we must somehow develop a model of many-processor processing systems which can realistically predict the processor time breakdown with a reasonably amount of computation, because we see the current trend of computer design moving towards many-processor large-scale computer systems. Finally, in the area of response time analysis, the accuracy of percentile response time should be examined more throughly against actual response time characteristics of real computer systems. For this purpose, more data concerning response time characteristics of actual time-shared computer systems must be collected.

Although the class of computer system configurations (see Figures 1-1 and 1-2) considered in this thesis is fairly general in structure,

it is almost certain that we must consider other classes of configurations also which will attract more attention in the future. For example, distributed (geographically separated) computers may become more attractive in near future to allow reliability and scaling up of size (on the order of more than ten). A constant effort must be made to develop modeling techniques which can realistically evaluate the performance of computer systems of today and the future.

APPENDIX A THROUGHPUT ANALYSIS PROGRAM

This appendix gives an explanation of the throughput analysis program which was extensively used in deriving the processor time breakdown for various multi-processor multi-memory processing systems under multiprogramming in Chapter 5. The program is entirely based on the processor model of Chapter 3 and uses particularly Eqs. (3.3.4) through (3.3.6) and Eqs. (3.4.6) through (3.4.22). The source program written in PL/I and a sample console session involving analyses of a single processor configuration and a dual processor configuration are also included in this appendix.

In using the throughput analysis program, the configuration of a processing system under investigation must be interactively specified from a terminal of a time-shared computer system on which this program is executed. This program needs the following input parameters[1] which specify the system configuration.

1. number of processers (m ; m = 1 or 2)
2. number of primary memory units (n)
3. degree of multiprogramming (q)
4. mean page fetch time (mpft)
5. mean length of in-page operation ($\bar{t}_{in}$)
6. missing-page probability (p)
7. mean paging overhead time ($K_{\ell}\bar{t}_{pl}$)

---

[1] Performance evaluation of a single processor processing system does not require all of these input data.

8. miscellaneous overhead coefficient (δ)

9. memory cycle interference coefficient (γ)

A few comments may be in order about some of these inputs. In specifying the values of the mean length of in-page operation ($\bar{t}_{in}$) and the missing-page probability (p), it should be remembered that these variables automatically determines the mean time between page faults, mtbpf (1 CPU), as follows.

$$\text{mtbpf (1 CPU)} = \frac{\bar{t}_{in}}{p}$$

It has been found through the experience of using this program that the resulting performance is rather insensitive to a particular set of values of $\bar{t}_{in}$ and p if their ratio, i.e., mtbpf (1 CPU), is constant. Therefore, after the value of mtbpf (1 CPU) is carefully selected, these two particular values can be fairly arbitrarily determined. The input representing the mean paging overhead time reflects a slow-down factor concerning data-base lockout. Therefore, K_ℓ should be selected in such a way that

$$K_\ell \begin{cases} = 1 & \text{for a single processor system } (m = 1), \\ \geq 1 & \text{for a dual processor system } (m = 2). \end{cases}$$

On the Multics system, the value of K_ℓ is approximately 1.5 for $m = 2$. The typical value of the memory cycle interference coefficient of the Multics system is found to be $\gamma \simeq 1.20$. The value of the miscellaneous overhead coefficient of the same system is typically $\delta \simeq 0.25$.

Upon receipt of these parameters, the program proceeds to derive and print out the processor time breakdown (into various system overhead times, idle times, and users' useful computation) and the relative

processor speed (the value of the effective slow-down factor due to memory cycle interference, K_m, for a particular configuration under study). The program also prints out the steady-state probabilities of the states of the model (for the definition of these states, see Table 3-2 or Table 3-4), as an additional information. The sample console session given in Section A.II. more concretely demonstrates how the throughput analysis program should be used. The lines typed by a user of the throughput analysis program are underlined and the lined typed by the system are not.

The throughput analysis program is written in PL/I, as shown in Section A.I. Therefore, this program can be transferred to other systems after only a few slight modifications; if the program is to be transferred to an IBM system, "call ioa_" (or "call ioa_$nnl") and "call read_list_" statements contained in the program must be replaced respectively by "put list" and "get list" statements. An external routine, "mlsq", invoked by the program solves a system of linear simultaneous equations, and is documented in "System/360 Scientific Subroutine Package (PL/I)--- Program Description and Operations Manual" (H20-0586-0) published by IBM.

A.I. Source Program

```
                    thruput.pl1          07/24/72  2305.3 edt Mon

/*                                       Thruput Analysis                  */
/*                                            of                           */
/*                                       Processing System                 */
/*                                                                         */
thruput:procedure;
          dcl (thruput1,thruput2) ext entry,
                    (ioa_,read_list_) entry,
                    m          fixed bin;
          call ioa_("");
          call ioa_("*************************************************");
          call ioa_("*                  THROUGHPUT  ANALYSIS          *");
          call ioa_("*************************************************");
          call ioa_("");
          call ioa_$nnl("number of processors = ");
          call read_list_(m);call ioa_("");
          if m=1 then call thruput1;
          if m=2 then call thruput2;
          call ioa_("");
end thruput;
```

```
/*                                                                    */
/*                          6/29/71   Thruput Analysis                */
/*                                    for 1 CPU System                */
/*                                                                    */
thruput1:procedure;
          dcl       (ioa_,ioa_$nnl,read_list_) entry,
                    mdm internal entry returns (float bin),
                    (ta,tp,tm,rho,sum,psum,util,poverhead,thruput) float bin,
                    (del,thru1,thru2,cm) float bin,
                    (q,m,i) fixed bin,
                    (t1,t2,t3) fixed bin,
                    p(0:20) float bin;
          call ioa_$nnl("degree of multiprogramming = ");
          call read_list_(q);call ioa_("");
          call ioa_$nnl("mean page fetch time (in msec) = ");
          call read_list_(tm);call ioa_("");
          call ioa_$nnl("mean time between page faults (in msec) = ");
          call read_list_(ta);call ioa_("");
          call ioa_$nnl("mean paging overhead time (in msec) = ");
          call read_list_(tp);call ioa_("");
          call ioa_$nnl("miscellaneous overhead coefficient = ");
          call read_list_(del);call ioa_("");
          rho = tm/ta;
          sum = 0;
          do i=0 to q;
                    sum = mdm(i,rho) + sum;
          end;
          psum=0;
          do i=0 to q;
                    p(i) = mdm(q-i,rho)/sum;
                    psum = psum + p(i);
          end;
```

```
            cm=0;
            poverhead = (1-p(0))*tp/ta;
            thruput = (1-p(0))*(1-tp/ta);
            thru1=(1/(1+del))*thruput;
            thru2=(del/(1+del))*thruput;
            call ioa_("          paging overhead                =¬f",poverhead);
            call ioa_("          miscellaneous overhead         =¬f",thru2);
            call ioa_("          multiprogramming idle          =¬f",p(0));
            call ioa_("          memory cycle idle              =¬f",cm);
            call ioa_("          percentile thruput             =¬f",thru1);
            call ioa_("");
            call ioa_("");
            call ioa_("***  Additional Information  ***");
            call ioa_("");
            call ioa_("   Qp CPU SM");
            do i=0 to q;
                        t1=i-1;if i=0 then t1=0;
                        t2=1; if i=0 then t2=0;
                        t3=q-i;
                        call ioa_("pi( ¬d, ¬d, ¬d) = ¬f",t1,t2,t3,p(i));
            end;
            call ioa_("sum of pi(...) = ¬f",psum);
            call ioa_("");call ioa_("");
            mdm:procedure(di,dr) returns (float bin);
                        dcl         (di,l) fixed bin,
                                    (dr,ans) float bin;
                        ans = 1;
                        if di=0 then go to dest1;
                        do l=1 to di;
                                    ans = ans*dr/l;
                        end;
                        dest1:return(ans);
            end;
end thruput1;
```

```
/*                                                                      */
/*                          3/25/71   Thruput Analysis                  */
/*                                    for 2 CPU System                  */
/*                                                                      */
thruput2:procedure;
declare (ioa_,ioa_$nnl,read_list_) entry;
declare (ta,mp,tp,nb,bi,tm,ql,del,rm,ra) binary float;
declare (m,n,k,q) binary fixed;
call ioa_$nnl("number of primary memory units = ");
call read_list_(nb);call ioa_("");
call ioa_$nnl("degree of multiprogramming = ");
call read_list_(q);call ioa_("");
call ioa_$nnl("mean page fetch time (in msec) = ");
call read_list_(tm);call ioa_("");
rm=1/tm;
call ioa_$nnl("mean length of in-page operation (in msec) = ");
call read_list_ (ta);call ioa_("");
ra=1/ta;
call ioa_$nnl("missing-page probability = ");
call read_list_ (mp);call ioa_("");
call ioa_$nnl("mean paging overhead time (in msec) = ");
call read_list_ (tp);call ioa_("");
call ioa_$nnl("miscellaneous overhead coefficient = ");
call read_list_(del);call ioa_("");
call ioa_$nnl("memory cycle interference coefficient = ");
call read_list_ (bi);call ioa_("");
ql=q;
m=2*q;n=m;k=1;
          begin;
          declare (a(m,n),b(m,n)) binary float (53);
          declare (ci,cm,ce,speed,thruput,temp1,temp2,thru1,thru2,ngov) bin float;
          declare (mt,nt,kt) binary fixed;
```

```
          declare (t1,t2,t3,t4,t5) binary fixed;
          declare mlsq ext entry;
          do mt=1 to m;do nt=1 to n;a(mt,nt)=0;end;end;
          do mt=1 to m;do kt=1 to k;b(mt,kt)=0;end;end;
/*                                                                    */
          do mt=1 to q-3;
          a(mt,2*mt-1)=2*ra*mp*(nb-1)/nb;
          a(mt,2*mt)=2*ra*mp*(nb-1)/(nb*bi);
          a(mt,2*mt+2)=2*ra*(1-mp)*(nb-1)/(nb*bi);
          a(mt,2*mt+3)=(mt+1)*rm;
          a(mt,2*mt+1)=-(mt*rm+2*ra*(1-mp)/nb+2*ra*mp/nb+2*ra*mp*(nb-1)/nb);
          end;
/*                                                                    */
          do mt=1 to q-3;
          a(q-3+mt,2*mt)=2*ra*mp/(nb*bi);
          a(q-3+mt,2*mt-1)=2*ra*mp/nb;
          a(q-3+mt,2*mt+1)=2*ra*(1-mp)/nb;
          a(q-3+mt,2*mt+4)=(mt+1)*rm;
          a(q-3+mt,2*mt+2)=-(mt*rm+2*ra*(1-mp)*(nb-1)/(nb*bi)
+2*ra*mp*(nb-1)/(nb*bi)+2*ra*mp/(nb*bi));
          end;
/*                                                                    */
          a(n-5,2)=2*ra*(1-mp)*(nb-1)/(nb*bi);
          a(n-5,3)=rm;
          a(n-5,1)=-((2*ra/nb)*(1+mp*(nb-1)));
          a(n-4,1)=2*ra*(1-mp)/nb;
          a(n-4,4)=rm;
          a(n-4,2)=-((2*ra/(nb*bi))*(nb-1+mp));
/*                                                                    */
          a(n-3,n-5)=2*ra*mp*(nb-1)/nb;
          a(n-3,n-4)=2*ra*mp*(nb-1)/(nb*bi);
          a(n-3,n-2)=2*ra*(1-mp)*(nb-1)/(nb*bi);
          a(n-3,n-1)=(q1-1)*rm*(nb-1)/nb;
          a(n-3,n-3)=-((q1-2)*rm+2*ra*(1-mp)/nb+2*ra*mp);
/*                                                                    */
```

```
          a(n-2,n-3)=2*ra*mp;
          a(n-2,n-2)=2*ra*mp/bi;
          a(n-2,n)=ql*rm;
          a(n-2,n-1)=-((ql-1)*rm*(nb-1)/nb+(ql-1)*rm/nb+ra*mp);
          a(n-1,n-1)=ra*mp;
          a(n-1,n)=-(ql*rm);
/*                                                                    */
          do nt=1 to n;
          a(n,nt)=1;
          end;
/*                                                                    */
          b(n,1)=1;
/*                                                                    */
/*        do mt=1 to m;do nt=1 to n;                    printing        */
/*                  call ioa_("~d ~d ~f",mt,nt,a(mt,nt));   of        */
/*        end;end;                                      A matrix      */
/*                                                                    */
          call mlsq(a,b,m,n,k);
/*                                                                    */
          ci=(2*b(n,1)+b(n-1,1))/2;
          temp1=0;temp2=0;
          do mt=2 to n-2 by 2;
                    temp1=temp1+b(mt,1);
                    temp2=temp2+b(mt-1,1);
          end;
          cm=temp1*(bi-1)/bi;
          ce=1-ci-cm;
          speed=(bi*temp1+temp2+(b(n-1,1))/2)/(1-b(n,1)-(b(n-1,1))/2);
          thruput=ce*(1-ra*mp*tp);
          thru1=(1/(1+del))*thruput;
          thru2=(del/(1+del))*thruput;
          pgov=ce*ra*mp*tp;
/*                                                                    */
```

```
          call ioa_("          paging overhead                =¬f",pgov);
          call ioa_("          miscellaneous overhead         =¬f",thru2);
          call ioa_("          multiprogramming idle          =¬f",ci);
          call ioa_("          memory interference idle       =¬f",cm);
          call ioa_("          percentile thruput             =¬f",thru1);
          call ioa_("");
          call ioa_("          processor slow-down factor");
          call ioa_("          due to memory interference     =¬f",speed);
          call ioa_("");
          call ioa_("");
          call ioa_("***  Additional Information  ***");
/*                                                                        */
          call ioa("");
          call ioa_("   Qp CPU SM MI");
          do mt=1 to m;
                    t1=divide(mt+3,2,3,0);
                    t5=q-t1;if mt=m-1 then t5=0;
                              if mt=m then t5=0;
                    t4=0;if divide(mt,2,3,0)*2=mt then t4=1;
                         if mt=m then t4=0;
                    t2=2;if mt=m-1 then t2=1;
                         if mt=m then t2=0;
                    t3=t1-2;if mt=m then t3=t1-1;
                    call ioa_("pi( ¬d, ¬d, ¬d, ¬d) = ¬f",t5,t2,t3,t4,b(mt,1));
          end;
          call ioa_("");call ioa_("");
end thruput2;
```

A.II. A Sample Console Session

```
thruput

**********************************************************
*                  THROUGHPUT  ANALYSIS                  *
**********************************************************

number of processors = 1

degree of multiprogramming = 4

mean page fetch time (in msec) = 35

mean time between page faults (in msec) = 30

mean paging overhead time (in msec) = 6.1

miscellaneous overhead coefficient = 0.25

          paging overhead                 =.198411580
          miscellaneous overhead          =.155476615
          multiprogramming idle           =.0242053359
          memory cycle idle               =0.
          percentile thruput              =.621906459


***  Additional Information  ***

   Qp CPU SM
pi( 0, 0, 4) = .0242053359
pi( 0, 1, 3) = .0829897244
pi( 1, 1, 2) = .213402154
pi( 2, 1, 1) = .365832269
pi( 3, 1, 0) = .313570518
sum of pi(...) = .999999993


r 2243  2.796  58+29
```

```
thruput

**************************************************************
*                  THROUGHPUT  ANALYSIS                      *
**************************************************************

number of processors = 2

number of primary memory units = 3

degree of multiprogramming = 5

mean page fetch time (in msec) = 30.9

mean length of in-page operation (in msec) = 3.19

missing-page probability = 0.1

mean paging overhead time (in msec) = 9.0

miscellaneous overhead coefficient = 0.25

memory cycle interference coefficient = 1.20

          paging overhead                  =.251299866
          miscellaneous overhead           =.127883710
          multiprogramming idle            =.0524190464
          memory interference idle         =.0568625187
          percentile thruput               =.511534840

          processor slow-down factor
          due to memory interference       =1.07200967


***  Additional Information  ***

   Qp CPU SM MI
pi( 3, 2, 0, 0) = .103385237
pi( 3, 2, 0, 1) = .0632515093
pi( 2, 2, 1, 0) = .188467521
pi( 2, 2, 1, 1) = .113935644
pi( 1, 2, 2, 0) = .171850422
pi( 1, 2, 2, 1) = .102678935
pi( 0, 2, 3, 0) = .104922063
pi( 0, 2, 3, 1) = .0613090484
pi( 0, 1, 4, 0) = .0755611220
pi( 0, 0, 5, 0) = .0146334857


r 2245  7.683  171+171
```

APPENDIX B RESPONSE TIME ANALYSIS PROGRAM

This appendix gives an explanation of the response time analysis program which was extensively used in evaluating the response time characteristics for various systems in Chapter 5. The program is entirely based on the total system model of Chapter 4 and uses particularly Eqs. (4.3.2) through (4.3.4) and Eqs. (4.3.9), (4.3.19), and (4.3.25). The source program written in PL/I and a sample console session involving an analysis of a single processor configuration are also included in this appendix.

The response time analysis program has two phases; the first phase which estimates the effective percentile throughput and the second phase which predicts the response time distribution. In using this program, the configuration of the system under study must be specified from a terminal of a time-shared computer system on which this program is executed. In particular, the first phase of this program requires the following input parameters which specify the system configuration.

1. number of processors (m)
2. optimum degree of multiprogramming (q^*)
3. percentile throughput (θ_q) for each degree ($1 \leq q \leq q^*$) of multiprogramming
4. number of users (N)
5. average execution time of a user job ($\bar{T}_e$)
6. average think time of a user ($\bar{T}_t$)

Upon specification of the system configuration, the program proceeds to

derive the effective percentile throughput of this configuration. The computed result of the effective percentile throughput is printed out as a console output. The estimated steady-state probabilities of the states of the model (see Figure 4-4, for the definition of these states) are also printed out, as an additional information.

The second phase of this program requests the following input parameters.

1. number of processors (m)
2. effective percentile throughput ($\theta(N)$)
3. number of users (N)
4. average execution time of a user job ($\bar{T}_e$)
5. average think time of a user ($\bar{T}_t$)

Upon supply of these inputs, the program proceeds to evaluate the response time distribution. It prints out both the evaluated probability density and the evaluated probability distribution function (i.e., cumulative function) of the system response time. The sample console session given in Section B.II. more concretely demonstrates how the response time analysis program should be used. The lines typed by a user of this program are underlined and the lines typed by the system are not.

The comment on program transferability of the throughput analysis program is appropriate also for this program.

B.I. Source Program

resp.pl1 07/25/72 0013.5 edt Tue

```
/*                                        Response Time Analysis                  */
/*                                             of                                 */
/*                                         Total   System                         */
/*                                                                                */
resp:procedure;
          dcl (effect,t_dis) ext entry,
              (ioa_,ioa_$nnl,read_list_) entry options (variable),
              mess char(3);
          call ioa_("");
          call ioa_("*************************************************************");
          call ioa_("*                  RESPONSE TIME ANALYSIS                   *");
          call ioa_("*************************************************************");
          call ioa_("");
          call ioa_("Do you want to determine the effective percentile");
          call ioa_("throughput?  ---  type yes or no  ---");
          call ioa_$nnl("                                 ");
          call read_list_(mess);
          if mess = "yes" then call effect;
          call ioa_("Do you want to derive the response time distribution?");
          call ioa_("                ---  type yes or no  ---");
          call ioa_$nnl("                                 ");
          call read_list_(mess);
          if mess = "yes" then call t_dis;
          call ioa_("");
end resp;
```

```
/*                                                                      */
/*                 2/15/72   Effective Percentile Throughput            */
/*                 3/9/72    Modified                                  */
/*                 3/11/72   Modified                                  */
/*                                                                      */
effect:procedure;
declare (ioa_,ioa_$nnl,read_list_) entry;
declare (tt,tp,th) binary float;
declare (m,n,mt,qop) binary fixed;
declare (tr,pb) binary float;
declare (th1,th2,th3,th4,th5,th6,thmax) binary float;     /*          added     */
put skip;
put list ("***  Estimation of Effective Percentile Throughput  ***");
put skip;
put skip;
put list ("number of processors = ");
get list (m);
put skip;
put list ("degree of multiprogramming = ");
get list (qop);
put skip;
call ioa_("percentile throughput (th) in % for each degree (q) of ");
call ioa_("multiprogramming");
put skip;
call ioa_$nnl ("if q=1 then th = ");
call read_list_(th1);
if qop<2 then go to num2;
call ioa_$nnl ("if q=2 then th = ");
call read_list_(th2);
if qop<3 then go to num3;
call ioa_$nnl ("if q=3 then th = ");
call read_list_(th3);
```

```
if qop<4 then go to num4;
call ioa_$nnl ("if q=4 then th = ");
call read_list_(th4);
if qop<5 then go to num5;
call ioa_$nnl ("if q=5 then th = ");
call read_list_(th5);
if qop<6 then go to num6;
call ioa_$nnl ("if q=6 then th = ");
call read_list_(th6);
go to num;
num2:th2=th1;
num3:th3=th2;
num4:th4=th3;
num5:th5=th4;
num6:th6=th5;
num:th=th6;
put skip;
put list ("number of users = ");
get list (n);
put skip;
put list ("average execution time of a job in seconds = ");
get list (tp);
put skip;
put list ("average user think time in seconds = ");
get list (tt);
th=th/100;
thmax=th;
th1=th1/100;th2=th2/100;th3=th3/100;th4=th4/100;th5=th5/100;
th6=th6/100;
          call1:begin;
          declare (i,j,k,ctr) binary fixed;
          declare (k1,k2,k3) binary float;
          declare p(0:n) binary float;
          declare pb(0:10) binary float;
          declare (r,t1,t2,t3,t4,t5,t6,t7,t8,t9) binary float;
```

```
declare fact ext entry;
declare (theff) binary float;
ctr=0;
put skip;
call ioa_("iteration   estimate of effective % throughput");
change:r=tp/(th*tt);ctr=ctr+1;
          call ioa ("    ¬d                ¬f",ctr,th);
          t1,t2,t3,t4,t7,t8,t9=0;
          mt=m-1;
          do i=0 to mt;
                    if i=0 then go to tend1;
                    t5=1;t6=n;
                    do j=1 to i;
                    t5=t5*t6;
                    t6=t6-1;
                    end;go to tend2;
                    tend1:t5=1;
          tend2:call fact(i,k1);
          t1=t1+i*(r**i)*t5/k1;
          p(i)=(r**i)*t5/k1;
          t9=t9+i*p(i);
          t7=t7+(r**i)*t5/k1;
          end;
          call fact(m,k1);
          do i=m to n;
                    t5=1;t6=n;
                    do j=1 to i;
                    t5=t5*t6*r/m;
                    t6=t6-1;
                    end;
          t2=t2+i*(m**m)*t5/k1;
          p(i)=(m**m)*t5/k1;
          t9=t9+i*p(i);
          t8=t8+(m**m)*t5/k1;
          end;
```

```
                t9=t9/(t7+t8);
                do i=0 to mt;
                        t5=1;t6=n;
                        do j=0 to i;
                        t5=t5*t6;
                        t6=t6-1;
                        end;
                call fact(i,k1);
                t3=t3+(r**i)*t5/k1;
                end;
                call fact(m,k1);
                do i=m to n;
                        t5=1;t6=n;
                        do j=0 to i;
                        t5=t5*t6*r/m;
                        t6=t6-1;
                        end;
                t4=t4+(m**(m+1))*t5/(k1*r);
                end;
        tr=(t1+t2)*tt/(t3+t4);
        do i=0 to n;
        p(i)=p(i)/(t7+t8);
        end;
        theff = p(1)*th1+p(2)*th2+p(3)*th3+p(4)*th4+p(5)*th5+p(6)*th6
                + (1-p(0)-p(1)-p(2)-p(3)-p(4)-p(5)-p(6))*thmax;
        theff = theff/(1-p(0));
        if abs(th - theff) <1.0e-3 then go to converge;
        th = theff;
        go to change;
        converge:t1=0;
        do i=0 to n;
        t1=t1+p(i);
        end;
        put list ("
");
```

```
          put skip;
          put list ("effective percentile throughput = ",th,"
");
          put skip;
          put list ("average queue length = ",t9,"   jobs
");
          put skip;
          put list ("average response time = ",tr,"seconds
");
          put skip;
          put skip;
          put skip;
          call ioa_("***  Additional Information  ***");
          put skip;
          put skip;
          do i=0 to n;
                    call ioa_("pi(¬d) = ¬f",i,p(i));
          end;
          call ioa_("sum of p(i) is ¬f",t1);
          put skip;put skip;
end effect;
```

fact.pl1 07/24/72 2336.5 edt Mon

```
/*                                                                    */
/*                          2/14/71   Factorial Calculator            */
/*                                                                    */
fact:procedure(k,k1);
          declare (i,k) binary fixed;
          declare (k1,fi) binary float;
          if k<1 then go to end1;
          k1=1;
          do i=1 to k;
                    fi=i;
                    k1=k1*fi;
          end;
          go to end;
          end1:k1=1;
end:end fact;
```


t_dis.pl1 07/24/72 2337.9 edt Mon

```
/*                                                                    */
/*                                                                    */
/*                    9/22/71          Response Time Distribution     */
/*                                                                    */
t_dis:procedure;
declare   (tt,tp,th,r,rp,rt,t,p,acc,ave,t1,t2,t3,t4,t5,t6,t7,t8) float bin,
          (m,n,i,j,k) fixed bin,
          (ioa_,ioa_$nnl,read_list_) entry;
```

```
call ioa_("");
call ioa_("***  Derivation of Response Time Distribution  ***");
call ioa_("");
call ioa_$nnl("number of processors = ");
call read_list_(m);call ioa_("");
if m>1 then call ioa_("***  The current version of this program can accept ");
if m>1 then call ioa_("     only a single processor system. Sorry!  ***");
if m>1 then go to end;
call ioa_$nnl("effective percentile throughput in % =");
call read_list_(th);call ioa_("");
call ioa_$nnl("number of users = ");
call read_list_(n);call ioa_("");
call ioa_$nnl("average execution time of a job in seconds =");
call read_list_(tp);call ioa_("");
call ioa_$nnl("average user think time in seconds =");
call read_list_(tt);call ioa_("");
call ioa_("");call ioa_("");
call ioa_("  time        density        cumulative");
call ioa_("");
th=th/100;
rp=th/tp;
rt=1/tt;
r=tp/(th*tt);
begin;
t=0;
chng_t:t5=0;t6=0;t7=0;t8=0;
          do i=0 to n-1;
          if i=0 then go to tend1;
          t1=1;t2=n-1;t3=i;t4=1;
                    do j=1 to i;
                    t1=t1*(t2/t3)*rt*t;
                    t4=t4*t2*r;
                    t2=t2-1;
                    t3=t3-1;
                    end;go to tend2;
```

```
          tend1:t1=1;t4=1;
          tend2:t5=t5+t1;
                t6=t6+t4;
          if t1=0 then go to con;
          if t1<1.0e-30 then go to stop;
          if t4<1.0e-30 then go to stop;
          con:if i=0 then go to tend3;
          t3=1;
                    do j=1 to i;
                              t1=1;t2=j;
                              do k=1 to j;
                              t1=t1*rp*t/t2;
                              t2=t2-1;
                              end;
                    t3=t3+t1;
                    if t1<1.0e-30 then go to stop1;
                    end;go to stop1;
          tend3:t3=1;
          stop1:t7=t7+t4*t3;
          t8=t8+(i+1)*t4;
          end;
stop:p=rp*(exp(-rp*t))*t5/t6;
acc=1-(exp(-rp*t))*t7/t6;
ave=t8*r*tt/t6;
call ioa_("t = ¬5.1f  p = ¬11.9f  cum = ¬11.9f",t,p,acc);
t=t+0.5e0;
if p>0.02e0 then go to chng_t;
if acc<0.9e0 then go to chng_t;
call ioa_("");
call ioa_("average response time = ¬f    seconds",ave);
call ioa_("");call ioa_("");
end;
end:end t_dis;
```

B.II. A Sample Console Session

```
resp

**********************************************************
*                RESPONSE TIME ANALYSIS                  *
**********************************************************

Do you want to determine the effective percentile
throughput?  ---  type yes or no  ---
                        yes

***  Estimation of Effective Percentile Throughput  ***

number of processors = 1

degree of multiprogramming = 4

percentile throughput (th) in % for each degree (q) of
multiprogramming

if q=1 then th = 55
if q=2 then th = 63
if q=3 then th = 65
if q=4 then th = 65

number of users = 30

average execution time of a job in seconds = 0.4

average user think time in seconds = 20

iteration   estimate of effective % throughput
    1              .649999999
    2              .626669958
    3              .629028954


effective percentile throughput =        6.29028954e-01

average queue length =          3.52031380e+00        jobs

average response time =         2.65887877e+00      seconds
```

```
*** Additional Information ***

pi(0) = .158077382
pi(1) = .150782293
pi(2) = .139029734
pi(3) = .123772760
pi(4) = .106254710
pi(5) = .0878376858
pi(6) = .0698200641
pi(7) = .0532783591
pi(8) = .0389617118
pi(9) = .0272533605
pi(10) = .0181969544
pi(11) = .0115714571
pi(12) = .00699038355
pi(13) = .00400067121
pi(14) = .00216242540
pi(15) = .00110007037
pi(16) = .000524651696
pi(17) = .000233538489
pi(18) = .96529748e-04
pi(19) = .36830004e-04
pi(20) = .12881125e-04
pi(21) = .40955587e-05
pi(22) = .11719660e-05
pi(23) = .29810163e-06
pi(24) = .66347070e-07
pi(25) = .12657046e-07
pi(26) = .20121563e-08
pi(27) = .25590634e-09
pi(28) = .24409656e-10
pi(29) = .15522119e-11
pi(30) = .49352636e-13
sum of p(i) is 1.00000000
```

```
Do you want to derive the response time distribution?
          ---  type yes or no  ---
                        yes

***  Derivation of Response Time Distribution  ***

number of processors = 1

effective percentile throughput in % =62.90

number of users = 30

average execution time of a job in seconds =0.4

average user think time in seconds =20
```

time		density		cumulative	
t =	0.0	p =	.281582188	cum =	0.000000000
t =	.5	p =	.262502369	cum =	.136165990
t =	1.0	p =	.240528617	cum =	.262020107
t =	1.5	p =	.216799615	cum =	.376402460
t =	2.0	p =	.192369802	cum =	.478703693
t =	2.5	p =	.168154361	cum =	.568808913
t =	3.0	p =	.144896636	cum =	.647018701
t =	3.5	p =	.123155845	cum =	.713959388
t =	4.0	p =	.103311582	cum =	.770491533
t =	4.5	p =	.085580618	cum =	.817623936
t =	5.0	p =	.070041458	cum =	.856438048
t =	5.5	p =	.056662548	cum =	.888025880
t =	6.0	p =	.045330858	cum =	.913442105
t =	6.5	p =	.035878406	cum =	.933670118
t =	7.0	p =	.028105280	cum =	.949600466
t =	7.5	p =	.021798311	cum =	.962019749
t =	8.0	p =	.016745327	cum =	.971607707

```
average response time = 2.65923250    seconds


r 2119  9.451  68+60
```

NOTATION

Following is a partial list of the symbols used in this thesis. Only those symbols which have a global meaning throughout the text are listed here alphabetically.

a - degree of sharing of resident programs.

b - degree of sharing of non-resident programs.

b_j - j-th block of a program.

f_j - j-th page-frame of primary memory.

m - size of primary memory space (Chapter 2).
number of processors (Chapters 3 -5).

m_r - size of primary memory space reserved for resident supervisor programs.

m_n - size of primary memory space reserved for non-resident programs.

n - size of a program (Chapter 2).
number of primary memory units (Chapters 3 -5).

n_p - number of page faults.

p - missing-page probability.

p_i - i-th page of a program.

p_{ij} - conditional probability of a reference to a page p_j, given that a page p_i was referenced.

q - degree of multiprogramming.

q^* - optimum degree of multiprogramming.

r_t - page reference at time t.

t_0 - unit time of a discrete-time Markov process.

t_m - miscellaneous overhead time.

t_p - paging overhead time.

t_u - headway between page faults (user's useful work within a tbpf).

u - utilization factor of a processor.

K_ℓ - effective slow-down factor of the processor speed due to data-base lockout.

K_m - effective slow-down factor of the processor speed due to memory cycle interference.

M - total size of primary memory.

N - number of interactive terminal users.

N_s - number of Markovian states.

P - set of pages of a program.

T_e - execution time of a user job.

T_r - response time.

T_t - user's think time.

T_w - waiting time in the job queue.

$1/\alpha$ - mtbpf

$1/\beta$ - mpft

γ - memory cycle interference coefficient.

δ - miscellaneous overhead coefficient.

θ - (effective) percentile throughput.

$\theta(N)$ - effective percentile throughput.

θ_q - percentile throughput of the system under multiprogramming of degree q.

$\underline{\theta}$ - throughput.

λ - $\theta(N)\lambda_0$

$1/\lambda_0$ - average execution time of a user job ($\bar{T}_e$).

$1/\mu$ - average think time of a user ($\bar{T}_t$).

π - steady-state state probability.

ρ - μ/λ

σ - α/β

GLOSSARY

This glossary is a partial list of terminology used in this thesis and is intended to remind readers of the meaning of each term appearing throughout this thesis.

data-base lockout: multi-processor interference caused by unavailability of a shared writable data-base which cannot allow simultaneous accesses of multiple processors.

data-base lockout idle time: the processor idle time caused by data-base lockout.

effective percentile throughput: the average of percentile throughput (see Eq. (4.2.1)).

fully loaded system (heavily loaded system): if the system under multi-programming of degree q has almost always q eligible jobs, the system is said to be fully (or heavily) loaded.

global PRA: a page replacement algorithm for which the page for removal may be selected globally from any area of primary memory.

local PRA: a page replacement algorithm for which the page for removal is selected from the pages of the user process which necessitated that page replacement.

LRU PRA: the PRA which chooses the least-recently-used page for removal.

LET PRA: the PRA which chooses the page with the longest expected time until next reference.

macroscopic paging performance model: the model of dynamic paging performance of programs based on the model of sharing (see Section

2.5.1) and the linear paging model (see Section 2.5.2).

memory cycle interference: multi-processor interference caused by unavailability of a memory cycle of a particular primary memory unit.

memory cycle interference coefficient: see page 142.

memory interference idle time: the processor time lost due to memory cycle interference when two or more processors simultaneously access the same unit of primary memory.

miscellaneous overhead time: the system overhead time required to handle miscellaneous faults such as segment faults, protection faults, various non-paging interrupts, etc.

miscellaneous overhead coefficient: see Eq. (2.5.11).

mpft: mean page fetch time, i.e., the mean length of time required to fetch a missing page from the secondary memory system.

mtbpf: mean time between page faults, i.e., the mean length of time during which program is executed continuously without a page fault (see Eq. (2.5.9) also).

multiprogramming idle time: the processor time lost when processors under a full load do not have executable eligible user jobs.

multiprogramming of degree q: a multiprogramming algorithm which allows a maximum of q processes to be simultaneously eligible.

multi-processor interference: interference of multiple processors such as memory cycle interference and data-base lockout.

pagination: determination of a set of program blocks to be contained in each page of a program.

paging overhead time: the collection of all the system overhead times

required to process a page fault.

partitioned memory: a memory allocation of primary memory such that one area of primary memory is solely reserved for each user process.

percentile throughput: the percentage of a system's computational capacity utilized for users' useful work.

percentile response time: the time limit which guarantees that a certain proportion (e.g., 90 percent) of response times is shorter than that limit.

PP discipline: preemptive priority discipline (see Section 3.3.3).

(q; m, n) configuration: a hardware configuration of a processing system which involves m processors and n primary memory units, with q eligible user processes under multiprogramming.

response time: the time elasped between the receipt by a computer system of a user's specified job request and the satisfaction of that request at the terminal; for the total system model of this thesis, it is defined to be the total length of time spent on the processing system by each user job.

thrashing: excessive competition among user processes for primary memory space leading to a less than optimum use of system resources.

throughput: the average number of user jobs than can be completed per unit time.

BIBLIOGRAPHY

A1 Aho, A. V., Denning, P. J. and Ullman, J. D., "Principles of Optimal Page Replacement," J. ACM, Vol. 18, No. 1, Jan. 1971, pp. 80-93.

A2 Anderson, H. A. and Sargent, R. G., "A Statistical Evaluation of the Scheduler of an Experimental Interactive Computing System," Proc. Conf. on Statist. Methods for Evaluation of Computer System Performance, Brown Univ., Nov. 1971, pp. 73-98. (Available from Academic Press, New York, N.Y.)

A3 Arden, B. W., Galler, B. A., O'Brien, T. C., and Westervelt, F. H., "Program and Addressing Structure in a Time-Sharing Environment," J. ACM, Vol. 13, No. 1, Jan. 1966, pp. 1-16.

B1 Baer, J. L. and Sager, G. R., "Measurement and Improvement of Program Behavior under Paging Systems," Proc. Conf. on Statist. Methods for Evaluation of Computer System Performance, Brown Univ., Nov. 1971, pp. 241-264. (Available from Academic Press, New York, N.Y.)

B2 Baskett, F., "The Dependence of Computer System Queues upon Processing Time Distribution and Central Processor Scheduling," Proc. Third Symposium on Operating Systems Principles, October 1971, pp. 109-113.

B3 Belady, L. A., "A Study of Replacement Algorithms for Multiprogramming," IBM Systems J., Vol. 5, No. 2, 1966, pp. 78-101.

B4 Blum, J., "Modeling, Simulation and Information System Design," Information System Science and Technology, Nov. 1966, pp. 301-306.

B5 Brawn, B. S. and Gustavson, F. G., "Program Behavior in a Paging Environment," Proc. AFIPS FJCC 33, 1968, pp. 1019-1032.

B6 Buzen, J., "Analysis of System Bottlenecks Using a Queuing Network Model," Proc. ACM SIGOPS Workshop on System Performance Evaluation, April 1971, pp. 82-103. (Available from ACM.)

C1 Chang, A., "Some Probabilistic Models of Storage Systems," Report RJ 781, IBM Research Lab., San Jose, Calif., Nov. 1970.

C2 Chang, W., "Queues with Feedback for Time-Sharing Computer System Analysis," Operations Research, Vol. 16, No. 3, May-June 1968, pp. 613-627.

C3 Chang, W., Paternot, Y. J. and Ray, J. A., "Throughput Analysis of Computer Systems---Multiprogramming versus Multiprocessing," Proc. ACM SIGOPS Workshop on System Performance Evaluation, April 1971, pp. 59-81.

C4 Coffman, E. G. and Varian, L. C., "Further Experimental Data on the Behavior of Programs in a Paging Environment," Comm. ACM, Vol. 11, No. 7, July 1968, pp. 471-474.

C5 Coffman, E. G., "Analysis of a Drum Input/Output Queue under Scheduled Operation in a Paged Computer System," J. ACM, Vol. 16, No. 1, Jan. 1969, pp. 73-90.

C6 Coffman, E. G., "Correction to "Analysis of a Drum Input/Output Queue under Scheduled Operation in a Paged Computer System"," J. ACM, Vol 16, No. 4, Oct. 1969, p. 646.

C7 Conway, R. W., Maxwell, W. L. and Miller, L. W., Theory of Scheduling, Addison-Wesley, Reading, Mass., 1967.

C8 Cooper, R. B., Introduction to Queuing Theory, MacMillan Company, New York, 1972.

C9 Corbató, F. J., Daggett, M. M. and Daley, R. C., "An Experimental Time Sharing System," Proc. AFIPS SJCC 21, 1962, pp. 335-344.

C10 Corbató, F. J. and Vyssotsky, V. A., "Introduction and Overview of the Multics System," Proc. AFIPS FJCC 27, 1965, pp. 185-196.

C11 Corbató, F. J., "A Paging Experiment with the Multics System," In Honor of P. M. Morse, MIT Press, Cambridge, Mass., 1969, pp. 217-228.

C12 Corbató, F. J., Saltzer, J. H. and Clingen, C. T., "Multics---The First Seven Years," Proc. AFIPS SJCC 40, 1972, pp. 571-583.

D1 Denning, P. J., "The Working Set Model for Program Behavior," Comm. ACM, Vol. 11, No. 5, May 1968, pp. 323-333.

D2 Denning, P. J., "Virtual Memory," Computing Surveys, Nol. 2, No. 3, Sept. 1970, pp. 153-190.

D3 Denning, P. J., "On Modeling Program Behavior," Proc. AFIPS SJCC 40, 1972, pp. 937-944.

D4 D'Avanzo, W. C., Purdue University, Ph.D. Thesis, in preparation.

F1 Feller, W., An Introduction to Probability Theory and Its Applications, Vol. 1, Third Ed., Wiley, New York, 1968.

G1 Gaver, D. P. Jr., "Probability Models for Multiprogramming Computer Systems," J. ACM, Vol. 14, No. 3, July 1967, pp. 423-438.

G2 Gordon, W. J. and Newell, G. F., "Closed Queuing Systems with Exponential Servers," Operations Research, Vol. 15, No. 2, April 1967, pp. 254-265.

G3 Greenbaum, H. J., "A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer Systems," M.S. Thesis, MIT Project MAC Report MAC-TR-58, Jan. 1969.

G4 Grochow, J. M., "The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System," M. S. Thesis, MIT Project MAC Report MAC-TR-54, Oct. 1968.

H1 Hatfield, D. J. and Gerald, J., "Program Restructuring for Virtual Memory," IBM Systems J., Vol. 10, No. 3, 1971, pp. 168-192.

H2 Howard, R. A., Dynamic Probabilistic Systems, Vol. 1, Wiley, New York, 1971.

I1 Informatics, Inc., "Experiments in Automatic Paging," Report RADC-TR-71-231, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, Nov. 1971.

J1 Jackson, J. R.,"Jobshop-Like Queuing Systems," Management Science, Vol. 10, No. 1, Oct. 1963, pp. 131-142.

J2 Jaiswal, N. K., Priority Queues, Academic Press, New York, 1968.

K1 King, W. F. III, "Analysis of Demand Paging Algorithms," Proc. IFIP Congress, TA-3, August 1971, pp. 155-159.

K2 Kuehner, C. J. and Randell, B., "Demand Paging in Perspective," Proc. AFIPS FJCC 33, 1968, pp. 1011-1018.

L1 Lett, A. S. and Konigsford, W. L., "TSS/360: A Time-Shared Operating System," Proc. AFIPS FJCC 33, 1968, pp. 15-28.

L2 Lewis, P. A. W. and Schedler, G. S., "A Cyclic-Queue Model of System Overhead in Multiprogrammed Computer Systems," Report RC 2813, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, March 1970.

L3 Lewis, P. A. W. and Yue P. C., "Statistical Analysis of Program Reference Patterns in a Paging Environment," Proc. IEEE Computer Society Conference, Sept. 1971, pp. 133-134.

L4 Lucas, H. C., "Performance Evaluation and Monitoring," Computing Surveys, Vol. 3, No. 3, Sept. 1971, pp. 79-91.

M1 Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L., "Evaluation Techniques for Storage Hierarchies," IBM Systems J., Vol. 9, No. 2, 1970, pp. 78-117.

M2 McKinney, J. M., "A Survey of Analytical Time-Sharing Models," Computing Surveys, Vol. 1, No. 2, June 1969, pp. 105-116.

M3 Moore, C. G. III, "Network Models for Large-Scale Time-Sharing Systems," Ph.D. Thesis, Technical Report No. 71-1, Dept. of Industrial Engineering, Univ. of Michigan, Ann Arbor, April 1971.

O1 Organick, E. I., The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Mass., 1972.

R1 Rice, D. R., "An Analytical Model for Computer System Performance Analysis," Ph.D. Thesis, Univ. of Florida, June 1971.

R2 Rosen, S., "Electronic Computers: A Historical Survey," Computing Surveys, Vol. 1, No. 1, March 1969, pp. 7-36.

S1 Saltzer, J. H., "Traffic Control in a Multiplexed Computer System," Sc.D. Thesis, MIT Project MAC Report MAC-TR-30, July 1966.

S2 Saltzer, J. H. and Gintell, J. W., "The Instrumentation of Multics," Proc. Second Symposium on Operating Systems Principles, October 1969, pp. 167-174. (Also in Comm. ACM, Vol. 13, No. 8, August 1970, pp. 495-500.)

S3 Saltzer, J. H., "A Simple Linear Model of Demand Paging Performance," MIT Project MAC, in preparation.

S4 Scherr, A. L., "An Analysis of Time-Shared Computer Systems," Ph.D. Thesis, Project MAC Report MAC-TR-18, June 1965, (Available also from MIT Press, Cambridge, Mass.).

S5 Schrage, L. E., "The Queue M/G/1 with Feedback to Lower Priority Queues," Management Science, Vol. 13. No. 7, March 1967, pp. 466-474.

S6 Sekino, A., "Response Time Distribution of Multiprogrammed Time-Shared Computer Systems," Sixth Annual Conference on Information Sciences and Systems, Princeton University, Princeton, New Jersey, March 1972, pp. 613-619.

S7 Smith, J. L., "Multiprogramming under Page on Demand Strategy," Comm. ACM, Vol. 10, No. 10, Oct. 1967, pp. 636-646.

T1 Tanaka, H., "Analysis of On Line Systems Using Parallel Cyclic Queues," Denshi Tsushin Gakkai Rombunshi (Trans. of Inst. of Elec. and Comm. Engrs. of Japan), Vol. 53-C, No. 10, Oct. 1970, pp. 756-764, (Japanese).

T2 Tsao, R. F., Comeau, L. W. and Margolin, B. H., "A Multi-Factor Paging Experiment: I. The Experiment and the Conclusions," Proc. Conf. on Statist. Methods for Evaluation of Computer System Performance, Brown Univ., Nov. 1971, pp. 103-134. (Available from Academic Press, New York, N. Y.)

W1 Wallace, V. L. and Rosenberg, R. S., "RQA-1 The Recursive Queue Analyzer," Systems Engineering Lab., Univ. of Michigan, Ann Arbor, Mich., Feb. 1966.

W2 Wallace, V. L. and Mason, D. L., "Degree of Multiprogramming in Page-on-Demand Systems," Comm. ACM, Vol. 12, No. 6, June 1969, pp. 305-308.

W3 Woolf, A. M., "Analysis and Optimization of Multiprogrammed Computer Systems Using Storage Hierarchies," Ph.D. Thesis, SEL Technical Report No. 53, Dept. of Electrical Engineering, Univ. of Michigan, Ann Arbor, April 1971.

BIOGRAPHICAL NOTE

Akira Sekino was born in Nagasaki, Japan, on January 9, 1942. He graduated from Asahigaoka Senior High School, Nagoya, Japan, in 1960. He studied Electrical Engineering and Electronics Engineering at Nagoya University, Nagoya, Japan, receiving the degrees of B. Eng. (Ko-Gakushi), in 1964, and of M. Eng. (Kogaku-Shushi), in 1966.

Mr. Sekino worked for the Mitsubishi Electric Corporation, Tokyo, Japan, from 1966 to 1968, where he was engaged in the development of prototype computer systems. He joined the staff of the M.I.T. Electrical Engineering department in September, 1968, as a research assistant. Since then, he has worked on the research on performance evaluation of time-shared computer systems at M.I.T. Project MAC.

Mr. Sekino is a member of Sigma Xi and the Association for Computing Machinery.

Publications

1. "Distribution of Constituent Elements of Characters," Denshi Tsushin Gakkai Zasshi (J. of Inst. of Elec. and Comm. Engrs. of Japan), Vol. 47, No. 6, June 1964 , pp. 933-934. (With K. Udagawa et al.)

2. "Evaluation of Statistical Dependence among Quantized Meshes in Character Recognition," Denshi Tsushin Gakkai Zasshi, Vol. 49, No. 8, August 1966 , pp. 1478-1485. (With K. Udagawa et al.)

3. "The CTSS Performance Measured from a Terminal," Project MAC Technical Memo MAC-M-418, July 1969.

4. "Response Time Distribution of Multiprogrammed Time-Shared Computer Systems," Sixth Annual Princeton Conference on Information Sciences and Systems, Princeton University, Princeton, New Jersey, March 1972, pp. 613-619.