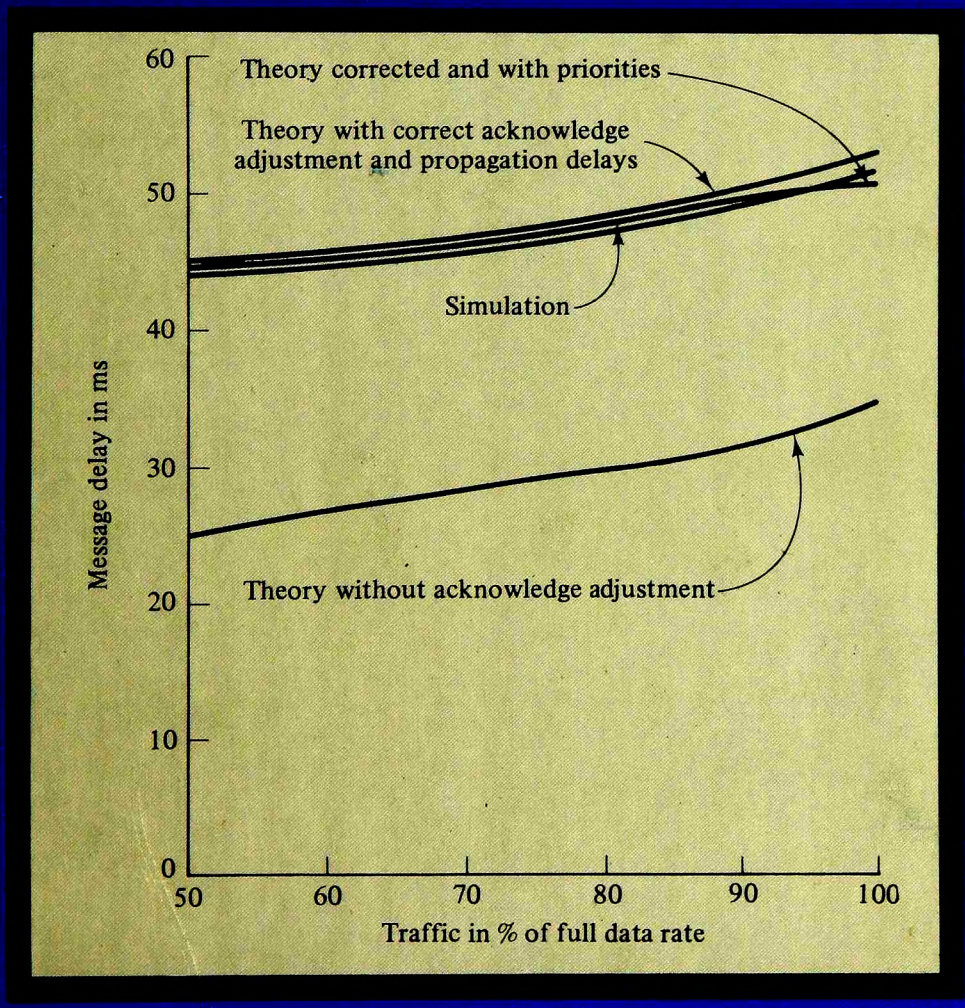
# COMPUTER SYSTEMS PERFORMANCE MODELING

Charles H. Sauer / K. Mani Chandy





**This book was previously published by Pearson Education, Inc**

# COMPUTER SYSTEMS PERFORMANCE MODELING

Charles H. Sauer
IBM Thomas J. Watson Research Center

K. Mani Chandy
The University of Texas at Austin

**Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632**

*Library of Congress Cataloging in Publication Data*

Sauer, Charles H.
Computer systems performance modeling.

(Prentice-Hall series in advances in computing science and technology)
Bibliography: p.
Includes index.
1. Electronic digital computers—Evaluation.
2. Queueing theory. I. Chandy, K. Mani, joint author.
II. Title. III. Series.
QA76.9.E94S28 001.64 80-21439
ISBN 0-13-165175-7

**Prentice-Hall Series in Advances in Computing Science and Technology**
**Raymond T. Yeh, editor**



Printed in the United States of America

10 9 8 7 6 5 4 3 2

Editorial production/supervision: Nancy Milnamow
Cover design: Edsal Enterprises
Manufacturing buyer: Joyce Levatino

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall of Canada, Ltd., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

To Caroline and Elizabeth, Jean and Christa

# CONTENTS



*Asterisks indicate sections are of a relatively advanced nature. These sections may be skipped without significant loss of understanding of the remaining material.

*7*



*8*



*9*



*10*

# PREFACE

If a computer system is to be used as intended, it must have "acceptable" performance, e.g., response times must be "small." It is common, but unfortunate, that performance is not seriously considered until the later stages of system evolution and that many systems have unacceptable performance when completed. By then, there will be relatively few avenues available to improve performance and the most frequently chosen will be to acquire additional hardware. Though the cost of computing has dropped sharply, both in the last few years and the time since electronic computers became available, computer systems are not free and almost certainly never will be.

If performance is to be considered in the design and development stages of a system, *modeling* must be used because the system is not yet operational, and thus its performance is not measurable. Though modeling is relatively well understood by researchers in the area and those with the mathematical background required by much of the literature, it is not *widely* understood. The purpose of this book is to make modeling methodology accessible to a wide audience of system designers, system developers and others who would benefit from modeling. As far as possible, we avoid sophisticated mathematics and deal with modeling on an intuitive basis. Some portions of the book require familiarity with elementary calculus and linear algebra. We do assume the reader has a thorough understanding of computing systems and programming.

The basic format of the book is to introduce the concepts of modeling in Chapters 2-8 and then to examine the practical application of these concepts in a series of six case studies in Chapter 9. In addition to more specific motivation for and description of modeling, the introductory chapter gives a brief overview of each of these case studies. Chapter 2 discusses general principles required for understanding of the remainder of the book. Modeling is heavily dependent on probability theory, and this chapter introduces some elementary concepts from probability theory as well as some other concepts fundamental to modeling. Chapter 3 is devoted to Markov processes, a basic tool for algebraically or numerically obtaining performance measures from models. A dominant factor in computer systems performance is *queueing* for system resources. Thus we focus our attention of models which consist of a *network* of queues (for system resources). Queueing networks are introduced in Chapter 3. Chapter 4

focuses on queues in isolation and on a particularly simple kind of networks, *open* networks. In open networks, the number of "jobs," the term used for the entities contending for resources, is potentially infinite. Though that may be realistic for some systems, particularly communication systems, *closed* networks, which have a limited population of jobs, are usually more appropriate as computer system models. Closed networks are more complex than open networks because of the stronger interactions between queues. Chapter 5 discusses both the mathematics required for obtaining performance measures for closed networks and the appropriate computational algorithms. In Chapters 3-5, the queueing networks have certain restricting assumptions which are necessary for exact solutions for performance measures. When these assumptions are unacceptably unrealistic, we must either use approximations (Chapter 6) or simulation (Chapter 7) to obtain performance estimates. The chapters prior to Chapter 8 assume that the input parameters to the models are given. In Chapter 8 we briefly cover methods for obtaining input parameters for models. Chapter 10 covers the management of modeling activities.

In addition to being a guide for the audience of practicing systems designers and developers, this book is intended as a text for an introductory (senior or first-year graduate) course in performance modeling and as supplementary reading for graduate systems courses. Portions of the material in this book have been used at the University of Texas at Austin in courses titled "Systems Modeling I" and "Advanced Operating Systems." In an introductory modeling course it is not necessary, or intended, that the book be covered strictly sequentially. In particular, the simulation material (Chapter 7) should be spread out over as much of the course as possible (concurrently with other material) once the first two chapters have been covered. Some of the case studies do not depend strongly on the preceding chapters. For example, the multiprocessor system study (Section 9.2) can be covered once Chapter 3 is finished, and the batch system study (Section 9.1) is primarily dependent on Chapter 3 and the first part of Chapter 8.

An instructor using this book is encouraged to skip sections or add supplementary material, depending on the particular situation and students. Section 3.5, Sections 5.1-5.5, Sections 5.7.2 and 5.7.3, Chapter 6, Sections 7.4 and 7.5 and some of the case studies are likely candidates for omission. An instructor may wish to supplement Chapters 2 and 3 with material from texts in probability theory such as DRAK67 or FELL68. The exercises given are fairly closely tied to the material discussed. Other exercises inspired by the students' background, e.g., local computing systems, are strongly encouraged. Except in Chapters 3 and 7, we have not included exercises involving computer implementation, partly to avoid language dependent exercises. However, such exercises will often be appropriate with some of the other chapters, particularly 4 and 5.

In Chapters 3 and 7 we have used PASCAL programs [JENS74] to illustrate numerical and simulation techniques. (These programs should be understandable to those familiar with other block structured languages, e.g., PL/I, even if they are not familiar with PASCAL.) We have used "standard" PASCAL entirely, and have tested these programs on both IBM 370 series and CDC 6000 series equipment. The program listings have been machine generated to avoid typographical errors. Most of the example output of the programs was the same for either the 370 or 6000 series equipment. However, in a few of the simulations the differences in numerical precision have a noticeable effect. The reader may not be able to precisely duplicate the program output, but should be able to obtain similar results.

Though we have covered the material which we consider most important in performance modeling of computer systems, and suggest further reading on these topics, there are a number of topics we have ignored. Most of these are covered in *Current Trends in Programming Methodology Volume III: Software Modeling,* edited by K.M. Chandy and R.T. Yeh and also published by Prentice-Hall. In discussing queueing network models we have taken a fairly traditional approach, neglecting the more recent and somewhat controversial "operational" approach. The operational approach is still in its infancy and not nearly as general as more traditional ("stochastic") approaches, at least at present. We refer readers interested in operational analysis to DENN78 and SEVC79.

## ACKNOWLEDGEMENTS

We are grateful for the support of the Computer Sciences Departments of the University of Texas at Austin and the IBM Thomas J. Watson Research Center in preparing this book. We would like to thank K.V. Karlstrom for his initial suggestion that we write the book and for his continuing editorial support.

A number of persons have contributed corrections and suggestions for improvement, especially Y. Bard, E.A. MacNair, D.F. Towsley and R.T. Yeh. We are also grateful to past and present colleagues and students for their collaboration in work discussed here. Thanks are also due to M.B. Bollard, B. Brown, D. Davis and B.A. Smalley for their typing of portions of the manuscript.

Finally, we thank the following authors and publishers for granting permission for republication of the cited material.

Cover figure, pages 11 and 12 and Section 9.6 based on L. Kleinrock, "Analytic and Simulation Methods in Computer Network Design" *Proc. Spring Joint Computer Conference 36* (AFIPS Press, Montvale, New Jersey, 1970), 569-579.

Pages 5 and 6 and Section 9.1 based on W. Chiu, D. Dumont and R. Wood, "Performance Analysis of a Multiprogrammed Computer System," *IBM J. of Research and Development 19* (May 1975), 263-271. Copyright 1975 by International Business Machines Corporation; reprinted with permission.

Pages 7 and 21-24 and Section 9.2 based on C.H. Sauer and K.M. Chandy, "The Impact of Distributions and Disciplines on Multiple Processor Systems," *Communications of the ACM 22* (January 1979), 25-34. Copyright 1979, Association for Computing Machinery, Inc., reprinted by permission.

Pages 7 and 8 and Section 9.3 based on J.C. Browne, K.M. Chandy, R.M. Brown, T.W. Keller, D.F. Towsley and C.W. Dissley, "Hierarchical Techniques for Development of Realistic Models of Complex Computer Systems," *Proc. IEEE 63,* (June 1975), 966-975. Copyright © 1975 by the Institute of Electrical and Electronics Engineers, Inc.

Pages 9 and 10 and Section 9.4 based on R.M. Brown, J.C. Browne and K.M. Chandy, "Memory Management and Response Time," *Communications of the ACM 20,* 153-165 (March 1977). Copyright 1977, Association for Computing Machinery, Inc., reprinted by permission.

Page 11 and Section 9.5 based on Y. Bard, "The VM/370 Performance Predictor," *Computing Surveys 10,* (September 1978), 333-342. Copyright 1978, Association for Computing Machinery, Inc., reprinted by permission.

Pages 124-163 based on K.M. Chandy and C.H. Sauer, "Computational Algorithms for Product Form Queueing Networks," *Communications of the ACM 23,* 10 (October 1980). Copyright 1980, Association for Computing Machinery, Inc., reprinted by permission.

Chapter 6 and Section 9.5 based on K.M. Chandy and C.H. Sauer, "Approximate Methods for Analysis of Queueing Network Models of Computer Systems," *Computing Surveys 10,*

(September 1978) 263-280. Copyright 1978, Association for Computing Machinery, Inc., reprinted by permission.

Quotation on page 196 from D.E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms,* (Addison-Wesley Publishing Co., Reading, Massachusetts, 1969), p. 5.

# *CHAPTER 1*

# INTRODUCTION

## 1.1 NEED FOR PERFORMANCE EVALUATION

Our most important concern with a computer system is that it correctly perform its intended functions. By "computer system" we may mean an entire computer facility or some subset of that facility, e.g., an operating system, a data base system or a particular application program. Thus "intended function" may be quite general or fairly specific.

Usually our second concern with a computer system is that it have "adequate" performance at a "reasonable" cost. (We may have roughly comparable concern for reliability, security and other aspects of the system, depending on intended functions.) Definitions of "adequate performance" and "reasonable cost" may be explicitly or implicitly given, and the definitions will usually be determined largely by the intended functions. Someone using a text editor or a word processing system may reasonably expect "instantaneous" response to all but a few commands. A user of a data base system, e.g., for travel reservations, may tolerate somewhat slower response but will find the system unacceptable if typical response times are more than say 10 seconds and unusable if response times are much longer than that. On the other hand, a programmer requiring compilation of a significant program may well be willing to wait several minutes without complaint.

It is unfortunately, and unnecessarily, the case that performance and cost are given little consideration until late in the development of most computer software systems. The developers' thoughts may be that "If the performance isn't adequate, we can always use a faster CPU or more memory or more disks to get adequate performance." There are numerous examples of systems which have gone through their entire development with little consideration for performance and which had totally unacceptable performance when complete. If one ends up with unacceptable performance with a reasonable amount of hardware, then the only options are to abandon the system entirely (which happens too frequently) or to go through redesign and redevelopment phases until the system is acceptable. Either of these options is much more expensive than a design and development process which explicitly considers performance.

The objective of this book is to describe performance estimation methods that can be used throughout the evolution of a computer system: to reject infeasible alternatives in the early design stages, to guide the develop-

ment process, to suggest hardware and software configurations when the system is complete and to guide redesign and redevelopment when functional changes are required. We cannot hope to describe the integration of performance evaluation in each of these system evolution phases. What we will do is describe methods that can be used in all of these phases and then examine a number of published studies of the application of the methods to computer systems in various stages of evolution.

## 1.2 PERFORMANCE EVALUATION METHODS

The obvious approach to evaluation of system performance is to directly *measure* that performance, either with hardware dedicated to measurement or with code embedded in software to obtain performance estimates or a combination of hardware and software. Direct measurement is both *accurate* and *credible*. We distinguish between accuracy and credibility because the approaches we discuss are capable of sufficient accuracy for their intended use but may not be credible to those who do not understand them. This lack of credibility is perhaps the principal liability of the approaches we advocate. Hopefully, this book will make these approaches accessible to a broad audience and thus make them more credible.

There are two major problems with measurement: First, measurement is not feasible in the design and development stages of the system; the system is not measurable if it is not operational. Second, measurement of most systems is a complex activity which involves considerable human and machine costs. How do we set up the measurement experiments? Do we use a "live" workload, not knowing whether that workload will be representative of the typical operating conditions, or do we try to use a controlled workload, e.g., a *benchmark* or *synthetic workload*, which we think represents typical usage? How do we attach the measurement tools to the system to get our desired estimates? In any case we are likely to need a large amount of dedicated time on the computer system and may require significant processing of the output of the measurement data to get it into meaningful terms, especially if we are using a software tool for measurement. We will discuss measurement in more detail in Chapter 8. We should point out now that measurement may well be the most appropriate approach, in fact a necessary approach, if we are trying to "tune" a system for "optimal" performance once we have obtained adequate performance.

*Modeling* should be used when measurement is intractable. We devise a model that captures the main factors determining system performance, determine performance measures in the model and use these measures from the model as estimates of performance of the actual system. Depending on how we plan to determine performance measures in the model, the model may seem very abstract relative to the actual system or may be a very

detailed representation of the actual system. Generally, the more detailed the model, the less manageable it is and the more human and machine expense will go into obtaining its performance measures. However, as we will see, very abstract and seemingly simplistic models can provide relatively accurate estimates of system performance.

In the design of a system we do not need very accurate estimates; we are much more interested in rejecting terrible designs (from a performance viewpoint) than in selecting a "best" design. In the design stages we cannot produce a very detailed model because we do not know the details of the design. However, we may find very simple models helpful in the design process. These models may be so simple that we can construct them and obtain performance measures by hand; certainly the computational costs are negligible and the principal cost is the learning necessary to be able to construct the model. The cost savings are enormous if we can determine at the design stage that a particular design cannot give adequate performance rather than waiting until the system is operational to make this same determination.

(We do not mean to imply that model performance measures are usually determined by hand or even by writing a new program for each model. Usually one would use existing software written for modeling purposes. We will describe the algorithms peculiar to such software in sufficient detail, including example programs, that the reader will be able to construct modeling software. For discussion of existing software, see SAUE78a.)

As we proceed from the design stages to development of the system, we obtain more detailed designs. From these new designs we can develop a more detailed model and can use this model to gradually eliminate inappropriate designs until we settle on the final design. If the system of interest is an operating system or major piece of system software, then we may use this model in configuring specific installations of the software, i.e., to determine hardware requirements, file placement, etc. Once the system is operational, we may use measurements to correct deficiencies in the model. The model can then be used to configure other installations with greater confidence (and credibility) and, if functional changes are required, the model can be used in redesign and redevelopment.

The evolution process we have described may seem somewhat idealized, but it is not unrealistic; these stages can be recognized whether or not performance is considered. There are likely to be iterations through the stages as functional specifications change and as performance problems are encountered. Incorporation of performance evaluation, particularly performance modeling, can greatly decrease both the number of iterations

caused by performance problems and the expense of discarding designs which have unacceptable performance. Once the methodology is understood, incorporating modeling in the evolution process is inexpensive, especially when one considers the potential savings.

## 1.3 SURVEY OF CASE STUDIES

In Chapter 9 we will study in some detail six published examples of computer system performance models; Chapters 2-8 will provide the necessary background for full understanding of these models. Of the six models, the results of five have been carefully compared with measurements and shown to give sufficient accuracy for the intended use of the model. (The definition of "sufficient accuracy" depends on the stage of evolution of the system, as we suggested before.) The sixth model was used to evaluate some hypothetical multiprocessor systems proposed in the literature. Of the five models of implemented systems, two of the models were constructed in the design stages of the systems and used throughout the evolution of the systems. Of the remaining three, all of which were constructed after the systems were operational, two were constructed as research efforts in modeling methodology; the third is widely used in configuring specific installations of the system.

We are about to give a brief summary of the characteristics of the modeled systems and of the structural characteristics of the model, but first a few comments about characteristics common to these models and most models used in practice. A major aspect of most modern computing systems is the *sharing of resources.* The classic illustration is the multiprogrammed operating system. Put simply, the objective of multiprogramming is to have one program's use of a processor overlap with other programs' use of I/O devices so that several programs may share the machine, with each making progress similar to the progress it would make if it had sole use of the machine. This sharing of resources reduces the cost attributed to each program, i.e., if a program has sole use of the machine, it must be charged for the idle time of resources as well as the busy time. In the idealized multiprogramming system, programs are only charged for the time spent using resources. However, the sharing of resources inherently causes *contention for resources*; if two programs need the processor, one must wait. In a well designed system, the gains due to sharing more than make up for the losses due to contention. But the contention for resources is usually a very significant factor in performance and the most difficult performance factor to quantify. Relatively speaking, if there is no resource contention, then performance evaluation is usually simple. Our attention is thus focused on resource contention, and most of our performance models may be characterized as *networks of queues* or *queueing networks.* The models principally consider the queues associated with resources and the interactions between

resources and their queues. We define a queue associated with a resource to be the entities (e.g., the programs) waiting for or using the resource. (Note that some resources, e.g., memory, consist of many homogeneous units and many entities may simultaneously use units of the resource.)

Our other general comment is that *performance evaluation is an art,* not a science. This is true with measurement as well as modeling. In constructing a model, particularly in deciding which system characteristics to consider and which to ignore (it is usually impractical to consider all system characteristics), we must rely heavily on intuition and use methods which are not particularly rigorous. Unlike sciences where we strive for very precise characterizations, we must recognize that the complexity of the systems precludes great precision within practical limitations of cost and time. Our model structures may very well be influenced by pressures from those who will use its results, e.g., designers, implementors, purchasers, administrators, and users of the computer system. Though we will devote most of our attention to rigorous methods for solving a model, i.e., obtaining its performance measures, there are situations where rigorous solution methods are impractical and/or prohibitively expensive, especially when we need to solve a model repeatedly for different parameter values. In Chapter 6 our attention will be focused on methods appropriate to this common situation. These methods are practical and inexpensive, and are inspired by rigorous results, but are heuristic in actual application.

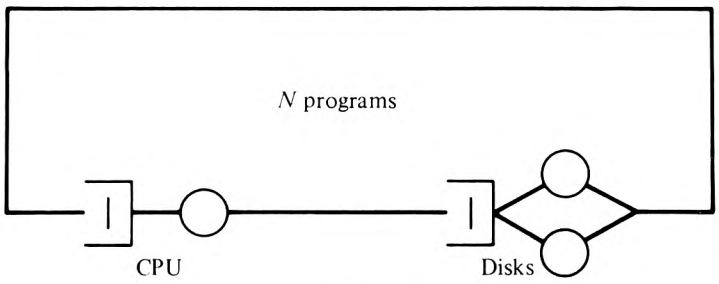



Figure 1.1

### 1.3.1 A Simple Batch System Model

Figure 1.1 depicts a queueing network model which can be used effectively to estimate system throughput and device utilizations of simple batch systems. We will refer to this network as a *cyclic queue model* because of the cycling of programs between the two queues representing the CPU and the disks. Such networks have been used as models of non-computer systems for decades. An early proposal of this network as a computer system model is found in GAVE67. We will look at the cyclic queue model as used by Chiu *et al* to evaluate the performance of an IBM 360/75

running the standard OS/MVT operating system at the University of California, Santa Barbara [CHIU75]. Only four parameters, the degree of multiprogramming, the average CPU service time (between a program's successive I/O activities), the number of disks, and the average disk service time (positioning and transfer), were used to describe this model in CHIU75, yet the model results agreed well with measurements. There are a number of assumptions implicit in the structure of the model and the choice of parameters and these assumptions are not immediately justifiable, but most of these assumptions have little effect on model results. Some of the assumptions are that (1) The degree of multiprogramming is essentially constant and determined by memory contention. Thus whenever a program finishes, it is replaced immediately by another. (2) The programs are homogeneous, i.e., we cannot distinguish their behavior. (3) The disks are equivalent, with a single queue for all accesses. (4) Scheduling may be treated as if it were First-Come-First-Served. (5) Successive CPU service times are independent and may be characterized by the *exponential probability distribution* (probability distributions are defined in Chapter 2). Similarly for disk service times. CPU and disk times are independent. (6) Programs do not overlap CPU and disk activities. (7) There is no memory interference between the CPU and disks. The list of assumptions could go on and on, depending on the level of detail we wish to consider.

The solution of this model depends on its characterization as a *Markov process.* Markov processes are key to most of our solution methods; they will be introduced in Chapter 3. We characterize the model by the possible combinations of programs at the CPU and disks. If the degree of multiprogramming is $N$, then there are $N+1$ such combinations: $N$ programs at the CPU and 0 programs at the disks, $N-1$ programs at the CPU and 1 program at the disks, etc. From these combinations we can obtain a set of $N+1$ linear equations which can be solved to obtain the probability of each combination, i.e., the fraction of time that the system has that combination of programs at the CPU and disks. Performance measures can then be obtained from these probabilities. For example, the CPU utilization is the sum of the probabilities of the combinations with at least one program at the CPU. Thus, if the degree of multiprogramming is not enormous, obtaining performance measures for this model is trivial. Further, we don't really have to numerically solve the linear equations in this case because of some algebraic simplifications which result in a *product form solution.* As we will see in Chapters 4 and 5, the product form solution exists for some very complex models. Where the product form solution exists, we can find simple algorithms for obtaining performance measures, even when the number of equations is enormous.

### 1.3.2 An Evaluation of Multiprocessor Systems

Though multiprocessor systems have been moderately popular for years, there has been an enormous increase in interest in multiprocessing because of the increasing availability of inexpensive small processors. There have been many published studies of uniprocessor system performance, but few of multiprocessor systems. In SAUE77b we examined performance effects of multiprocessor systems as compared to uniprocessor systems. Our objective was to study the effects of a variety of parameters and assumptions, particularly the number of processors, the degree of multiprogramming, relative balance between CPU and I/O services times, scheduling algorithms (including priorities) and characterization of service times by various probability distributions. Most of the models we used were variations on the cyclic queue model of Figure 1.1. Most of the models could be represented as Markov processes and solved as such. For one scheduling algorithm this was not possible and we used *simulation.* In a simulation solution, one constructs a program which behaves like the model and measures the behavior of that program. We will look at simulation from a queueing network point of view in Chapter 7.

### 1.3.3 A Data Management System Model

The Advanced Logistics System is a very large data management system developed by the United States Air Force. During the design of the system, Browne, Chandy and four other consultants developed two models which were used to guide the development and hardware selection [BROW75]. The primary model was a complex hierarchical queueing network model; its results were initially corroborated by a companion simulation model (well before the system was operational). At the time the model was constructed, the planned hardware included two CDC Cyber 70 mainframes, each with CPU, central memory and peripheral (I/O) processors. The two mainframes shared over a million 60 bit words of memory, approximately 100 disks for the data bases, eight disks for system and scratch files and 24 tape drives. The queueing network model began with four submodels: one for the CPU's, one for the central memories and shared memory, one for the data base disks, and one for the system/scratch disks and for the tape drives. The heuristic aggregation of the results of these models yielded the queueing network model of Figure 1.2. The modeling process is termed *hierarchical* because of the two levels of models. Though the network of Figure 1.2 is structurally more complex than the cyclic queue model, particularly in the detail of the I/O systems, there is much in common between the two models, both in terms of assumptions made and solution methods. The numerical solution of the network of Figure 1.2 has negligible computational cost. This model predicted that the proposed system was inadequate because of insufficient capacity in the

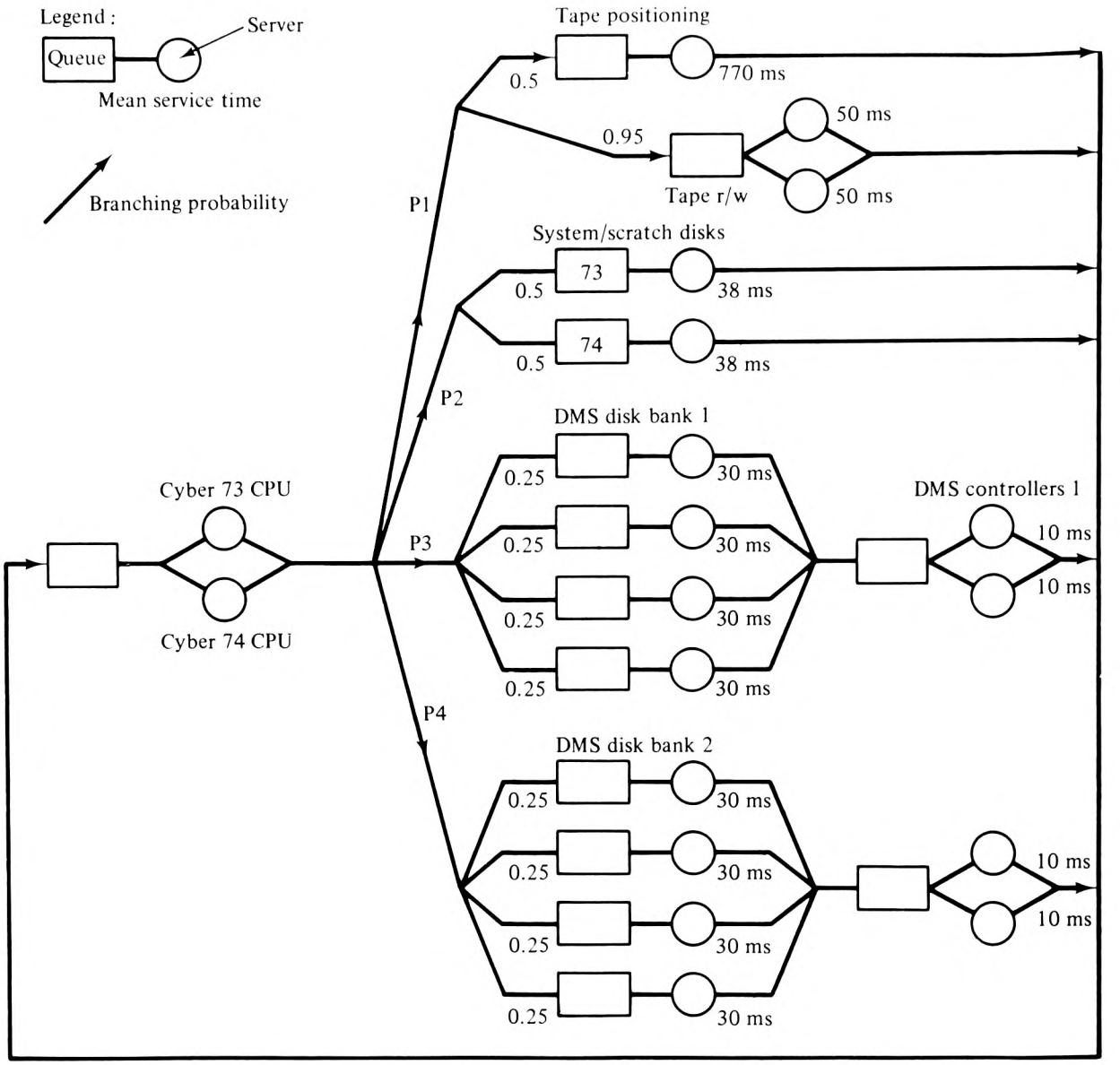



Figure 1.2

system/scratch disk subsystem, and that, if sufficient system/scratch disk capacity were obtained, performance would still be inadequate because of insufficient CPU capacity. Both of these predictions were confirmed by subsequent operational experience and measurements. The entire modeling effort, including construction, solution and documentation of both the queueing network and simulation models, required two months for the six consultants to complete.

### 1.3.4 A Model of an Interactive System

The queueing network model of Figure 1.3 is representative of a variety of models which have been used to evaluate the performance of general purpose interactive systems [BRAN74, BOYS75, BROW77]. We will look at the model in BROW77 of an interactive system at the University of Texas at Austin. At the time of the modeling effort, the system consisted of a CDC 6400 running a locally written operating system and using a large, non-executable core memory as a swapping device. Though the cyclic queue model and similar networks can be used successfully to estimate utilizations of processors and disks and to estimate throughput, it is difficult to use such a model to estimate response time. Such models provide estimates of the time a program spends in memory, but cannot provide estimates of the time spent waiting for memory, a significant portion of response time. The model of Figure 1.3 considers contention for memory explicitly rather than implicitly, and also considers the varying load on the system due to the time users spend thinking and typing.

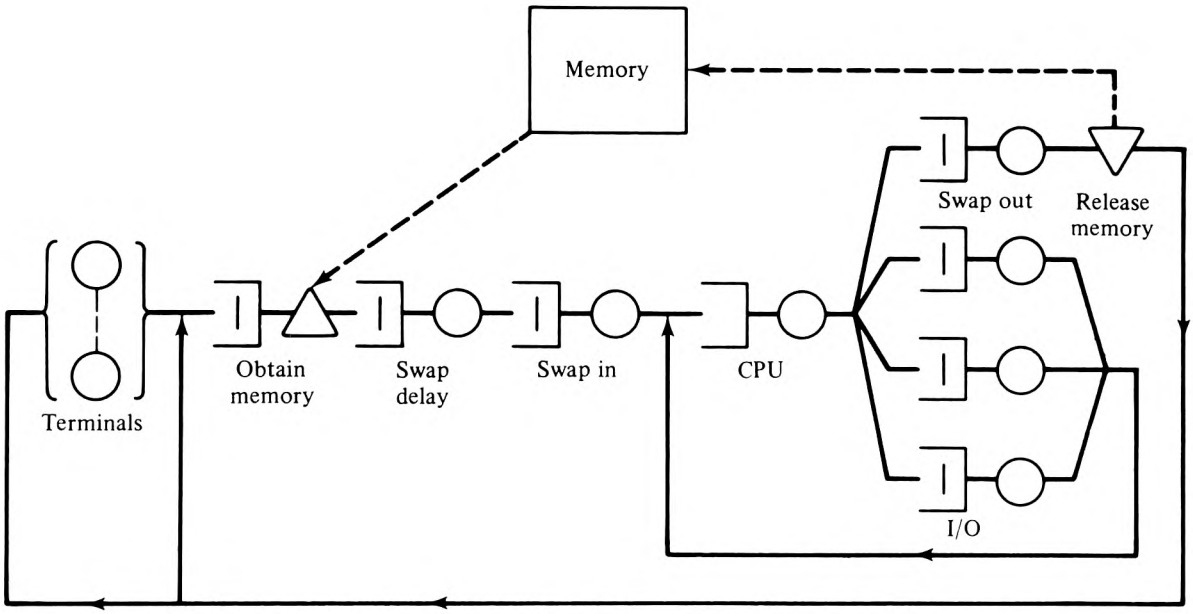



Figure 1.3

There is no convenient algebraic solution of the linear equations for this network and the number of equations will be quite large, even when the number of users is moderate, say 30. Exact numerical solution of the equations is impractical because of the memory and processing required. However, without the memory contention, a product form solution would exist. Using results from product form networks, we can obtain a conven-

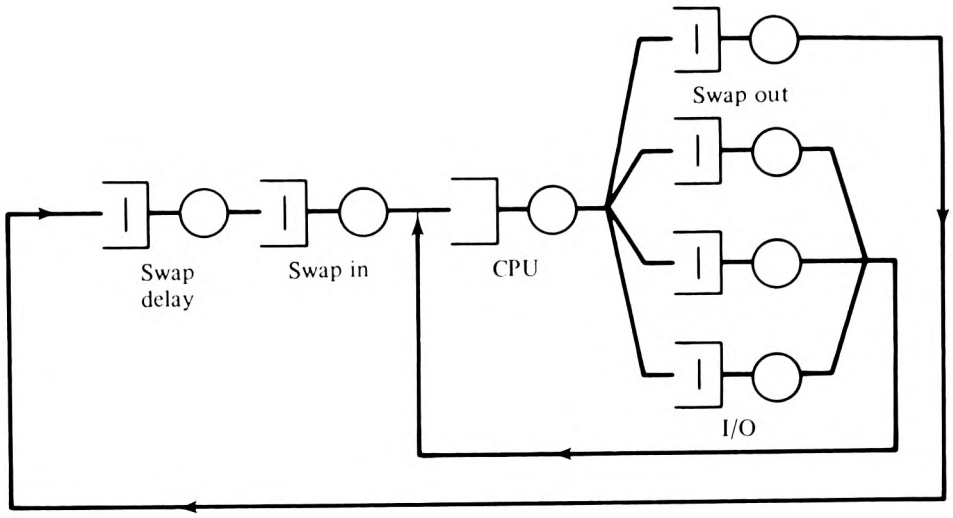



Figure 1.4

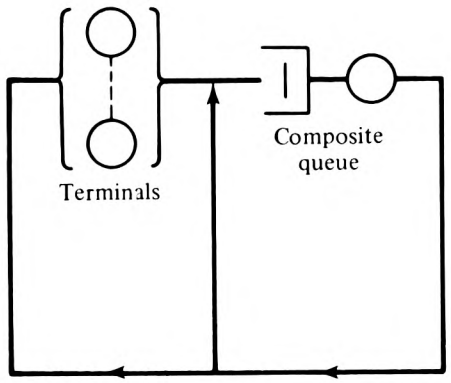



Figure 1.5

ient solution to the network of Figure 1.4. The solution of that network can be used in a hierarchical (heuristic) manner to obtain a network of the form of Figure 1.5, where the service times at the "composite queue" depend on the queue length. The solution of that network is also convenient and will give response time estimates similar to those of the network of Figure 1.3 at negligible computational cost.

### 1.3.5 The VM/370 Performance Predictor

VM/370 is a general purpose operating system for the IBM 370 series of computers. VM/370 provides a Virtual Machine Monitor [BUZE73] which is a special kind of operating system which gives each user the impression of having their own 370. Each user may run their own copy of the CMS uniprogramming interactive system or a conventional operating system (e.g., OS/MVS) in their "virtual 370". The VM/370 Performance Predictor [BARD77b, BARD78a] is a software package based on a queueing network model similar to that of Figure 1.3. Some of the principal differences are that each user may have entirely different characteristics (unlike the homogeneity of users assumed in BROW77) and that paging must be considered. The solution approach is similar to that of BROW77 but is a three level process, with the third level considering the I/O systems, and involves iterative solution of the three levels until certain consistency criteria are satisfied. (Since hierarchical solutions are usually heuristic, we may observe, and attempt to correct, inconsistencies in the performance measures.)

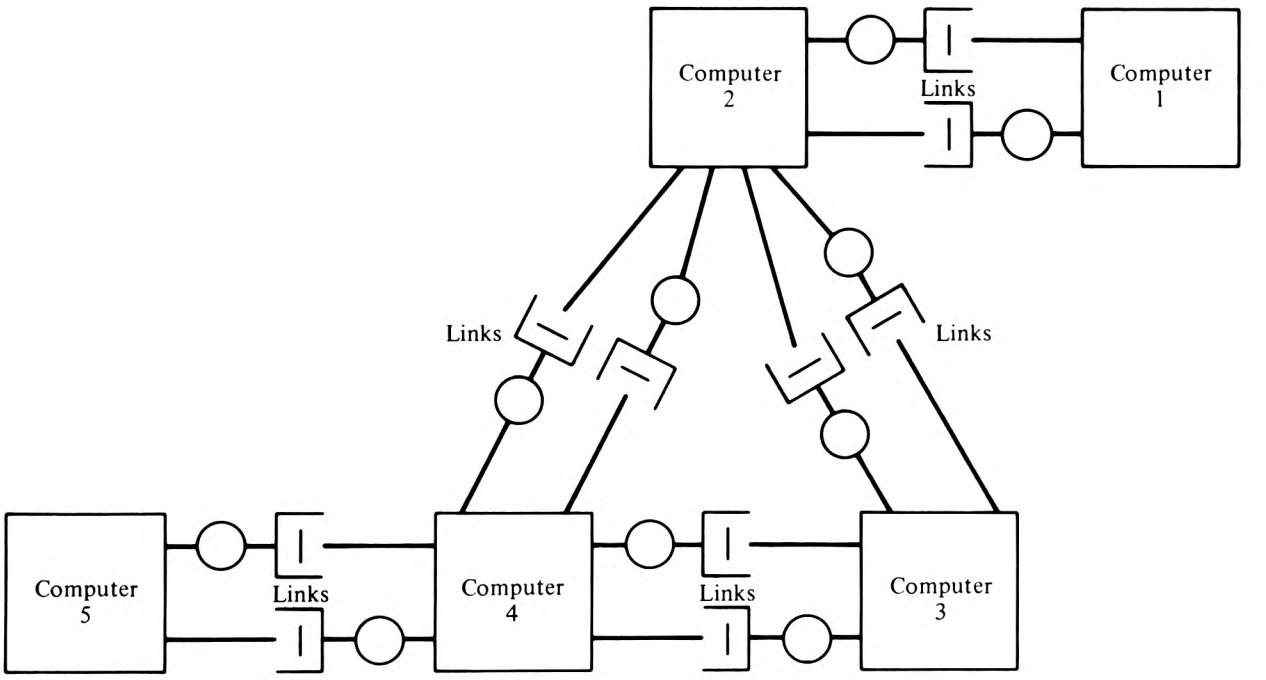



Figure 1.6

### 1.3.6 Computer Communication Models

One of the most influential computer networks is the ARPANET linking facilities at universities and research centers in North America, Hawaii and Europe. In studying the performance of the ARPANET, queueing models have played a central role [KLEI70]. In addition to evaluating the performance of the individual computer systems, we must consider the effects of messages sent between computer systems. When a user is physically connected to one computer but logically using another, a principal part of the user's response time will be the transmission delay of messages sent between the computers. In addition to such "user traffic" there will also be significant traffic of messages used to control the network and maintain coordination of the computers. The cost of the communication links will be a significant portion of the cost of using the network. A principal use of the queueing network model will be to ensure appropriate utilization of the communication links. Figure 1.6 depicts a queueing network model of a hypothetical subset of ARPANET. As with the cyclic queue model, a number of strong assumptions are usually made to allow a convenient solution, but the results of the model for link utilizations and average response time agree well with measurements.

# *CHAPTER 2*

# GENERAL PRINCIPLES

## 2.1 SERVICE TIME DISTRIBUTIONS

In modeling many resources of a computer system, we may be principally interested in the service times of programs which use the resources. (Aside — From an operating systems point of view we might use "process" or "task" instead of "program." From a queueing theory point of view we might use "customer" or "job" instead of "program." Generally we will assume all of these terms are synonymous and use "job.") For example, if the resource is a processor, a program's service time will consist of execution of instructions and the amount of time spent will be determined (principally) by the particular instructions executed, the processor times for these instructions, I/O requirements and memory management, if the system has virtual memory. If the resource is a moving head disk, then a program's service time will consist of a positioning time and a transfer time, and the amount of time will be determined (principally) by the distance (if any) the arm must move, the mechanical speed and the amount of information to be transferred.

All of the determining factors in the above examples are measurable (or observable) and in a sense deterministic. Thus we could include these factors directly in a model, at least conceptually. Such an approach would usually be totally impractical. We will pursue the issue of practicality in future chapters, especially in Section 3.1. It is usually appropriate and practical to characterize service times as random phenomena. The purpose of this section is to informally defend the claimed appropriateness and to describe the characterizations we will use in all of our models.

If we observe service times without directly observing the determining factors of those times, then the service times will usually appear to be random, i.e. unpredictable. For example, if we are observing disk service times, the initial position of the arm which holds the heads and rotating platters is unpredictable (without very detailed knowledge). The required position of the arm and platters is, perhaps, more predictable, but still unpredictable without detailed knowledge. The amount of data transferred is especially unpredictable (unless transfers are always a full buffer of fixed size). For processor times, the instruction paths will usually depend heavily on (unpredictable) data. (In a virtual memory system, the processor times will also depend heavily on memory management which may, in turn,

depend heavily on the behavior of the entire multiprogramming set of programs.)

Though service times are not predictable, that does not mean we cannot characterize them. For example, we can observe many service times and compute the average service time. This simple characterization is sufficient for many of our models. However, for proof of this statement and for some models we must consider more detailed characterizations.

The most detailed characterization we consider is that of a *probability distribution*, an assignment of probabilities to possible values or continuous intervals of possible values. We assume that distinct service times are *independent* and *identically distributed*. Though this is not always realistic, it is usually reasonable. Consideration of dependencies is beyond the scope of this book. When we say "identically distributed", we do not mean that CPU and I/O service times have the same distribution nor do we necessarily mean that all jobs have the same CPU service time distribution, etc.

Let us first consider the case where there is a finite set of possible values. In this case we can simply enumerate the possible values and perhaps display them graphically as in Figure 2.1.

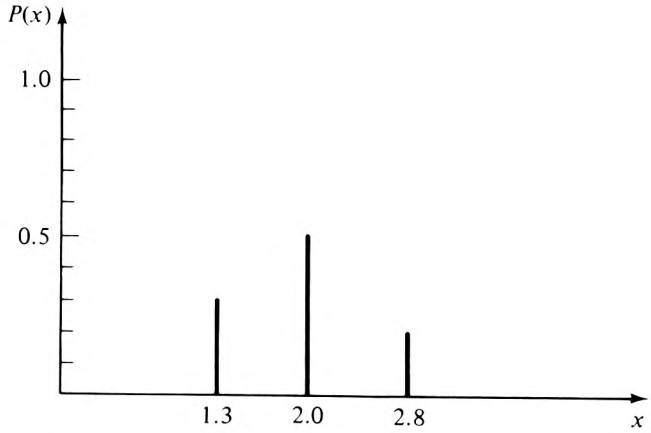



Figure 2.1

Here $P(x)$ is the probability of value $x$. (Informally we may define "probability" as "relative frequency." There is a close formal relationship between the two terms.) We must have $0 \leq P(x) \leq 1$ for all $x$ and $\sum_x P(x) = 1$.

Usually it is inconvenient to work with distributions directly; we prefer simple characterizations. The most important of these is the *mean* or *expected value* which corresponds to the average value in less formal terminology. The mean is written $E[x]$ and defined by

$$E[x] = \sum_x xP(x)$$

For the distribution of Figure 2.1

$$E[x] = 1.3 \times .3 + 2 \times .5 + 2.8 \times .2 = 1.95.$$

A generalization of the mean is the mean of some function of the service times. The most important cases are the *moments* and *central moments*. The $n^{th}$ moment is the expected value of the service time raised to the $n^{th}$ power, i.e.,

$$E[x^n] = \sum_x x^n P(x).$$

The mean and first moment are identical. The second moment is of particular interest. For the distribution of Figure 2.1

$$E[x^2] = (1.3)^2 \times .3 + (2)^2 \times .5 + (2.8)^2 \times .2 = 4.075$$

The $n^{th}$ central moment is the expected value of the $n^{th}$ power of the difference between the service time and the mean, i.e.,

$$E[(x - E[x])^n] = \sum_x (x - E[x])^n P(x). \tag{2.1}$$

The first central moment is identically zero. The second central moment is called by a special name, the *variance*. The square root of the variance is called the *standard deviation*. The Greek letter "$\sigma$" is often used as a symbol for the standard deviation. For the distribution of Figure 2.1

$$\sigma_x = \sqrt{(1.3-1.95)^2 \times .3 + (2-1.95)^2 \times .5 + (2.8-1.95)^2 \times .2}$$

$$= \sqrt{.2725} \approx .522.$$

Rather than use the definition (2.1) for the variance, it is often more convenient to use

$$\sigma_x^2 = E[x^2] - (E[x])^2. \tag{2.2}$$

The mean gives us an indication of the magnitude of the service time; the variance gives us an indication of the variability. However, we would often like a more direct indication of variability that is independent of the mean. For this purpose we use the *coefficient of variation*, $C_x$, defined by

$$C_x = \frac{\sigma_x}{E[x]}.$$

Processor service time distributions in general purpose computer systems are usually highly variable; values of $C_x$ of 10 or more are not unusual. I/O service time distributions are much less variable; values of $C_x$ much less than 1 are typical. (Note that $C_x = 0$ implies that all services times are the same.)

If the number of possible values is infinite (and this is usually the case) then we cannot depend on simple enumeration. If the possible values are *discrete*, i.e., countable, then we may be able to provide a function describing the probability of each possible value. For our purposes, the most important example is the *geometric distribution* with parameter $p$ such that $0 < p < 1$ and

$$P(x) = (1 - p)^{x-1}p, \; x = 1,2,3,....$$

By using the relationship

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}, \; 0 < |a| < 1, \tag{2.3}$$

we can easily show that $\Sigma P(x) = 1$. By repeated use of (2.3) we can also show that

$$E[x] = \frac{1}{p} \tag{2.4}$$

and

$$E[x^2] = \frac{2-p}{p^2}.$$

(Obtaining $E[x^2]$ directly is very tedious; indirect approaches using *transforms* or *generating functions* are usually more convenient [DRAK67,FELL68].) Thus we also have

$$\sigma_x = \frac{\sqrt{1-p}}{p}$$

and $C_x = \sqrt{1-p}$ for the geometric distribution. We usually will not use the geometric distribution to represent service times. However, what we say about service times also applies to other behavior, i.e., we can characterize *random variables* by probability distributions. In Chapter 4, we will introduce the use of the geometric distribution in regard to other system characteristics.

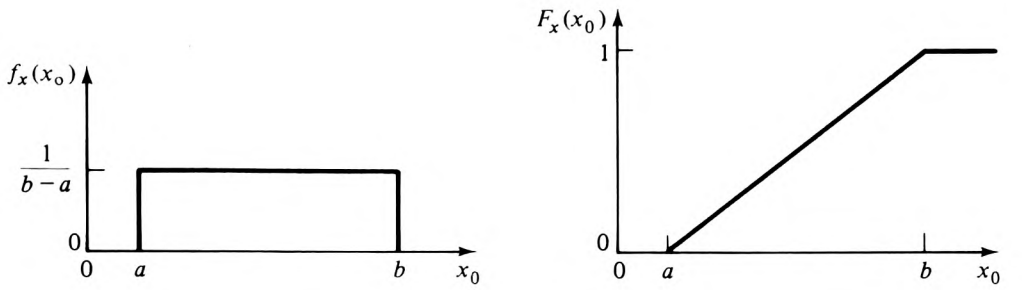



Figure 2.2

In considering service times we are usually interested in continuous portions of the positive real line. With a continuous range of possible values, the number of values is not countable and we must use alternate characterizations of the distribution. In particular, we are not interested in the probability of a particular value (which will always be zero for distributions we consider) but in the probability of a range of values. There are two common, complementary approaches: *probability density functions* and *probability distribution functions*. We will use the notation $F_x(x_0)$ for the distribution function. $F_x(x_0)$ is defined as the probability that a value x is less than or equal to $x_0$. It is required that $F_x(-\infty) = 0$, $F_x(\infty) = 1$, and $F_x(a) \leq F_x(b)$ for $a < b$. The density function is the derivative (where it exists) of the distribution function. We use the notation $f_x(x_0)$ for the density function. It is required that

$$f_x(x_0) = \frac{dF_x(x_0)}{dx_0},$$

$$F_x(x_0) = \int_{-\infty}^{x_0} f_x(x_0)dx_0,$$

and

$$\int_{-\infty}^{\infty} f_x(x_0)dx_0 = 1.$$

An important example is the *uniform* distribution on the interval $(a,b)$. With the uniform distribution each value in the interval $(a,b)$ is equally likely. (Though what we say may seem self-contradictory, zero probability does not necessarily mean that a particular value is impossible.) Figure 2.2 shows the density and distribution functions for the uniform distribution.

Our definitions of mean, moment, variance, etc. all apply to continuous distributions if we replace $P(x)$ with $f_x(x_0)$ and replace summation with integration. For example,

$$E[x] = \int_{-\infty}^{\infty} x f_x(x_0)dx_0,$$

$$E[x^2] = \int_{-\infty}^{\infty} x^2 f_x(x_0)dx_0,$$

and

$$\sigma_x^2 = \int_{-\infty}^{\infty} (x - E[x])^2 f_x(x_0)dx_0.$$

For the uniform distribution

$$f_x(x_0) = \begin{cases} \dfrac{1}{b-a}, & a \le x_0 \le b, \\ 0, & \text{otherwise}, \end{cases}$$

$$F_x(x_0) = \begin{cases} 0, & x_0 < a, \\ \dfrac{x_0 - a}{b-a}, & a \le x_0 \le b, \\ 1, & x_0 > b, \end{cases}$$

$$E[x] = \frac{a+b}{2},$$

$$E[x^2] = \frac{a^2 + ab + b^2}{3},$$

$$\sigma_x^2 = \frac{(b-a)^2}{12},$$

and

$$C_x = \frac{b-a}{(b+a)\sqrt{3}}.$$

If we want the probability that $a < x \leq b$, we can obtain this from the density function as

$$\text{Prob}[a < x \leq b] = \int_a^b f_x(x_0)dx_0$$

or from the distribution function as

$$\text{Prob}[a < x \leq b] = F_x(b) - F_x(a).$$

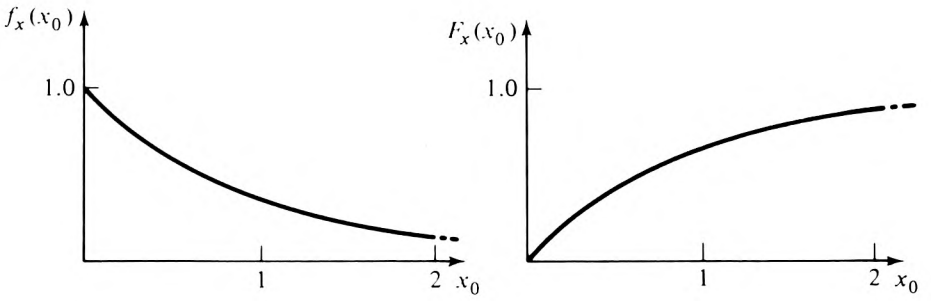



Figure 2.3

Perhaps more important in our work is the *(negative) exponential* distribution with "rate" $a$. For the exponential distribution

$$f_x(x_0) = \begin{cases} 0, & x_0 < 0, \\ ae^{-ax_0}, & x_0 \geq 0, \end{cases}$$

$$F_x(x_0) = \begin{cases} 0, & x_0 < 0, \\ 1 - e^{-ax_0}, & x_0 \geq 0, \end{cases}$$

$$E[x] = \frac{1}{a}, \tag{2.6}$$

$$E[x^2] = \frac{2}{a^2},$$

$$\sigma_x^2 = \frac{1}{a^2},$$

and

$$C_x = 1.$$

Figure 2.3 shows the density and distribution functions for the exponential distribution with $a = 1$.

The use of the exponential distribution is crucial to mathematical models of computer systems. However, as we discuss in detail in the next chapter it is possible to represent essentially arbitrary distributions with the *method of exponential stages*. Modelers often categorize distributions by their variability, relative to the exponential distributions. A class of distributions with greater variability than the exponential is known as *hyperexponential*. Similarly, some distributions with less variability are known as *hypoexponential*. Figure 2.4 shows density and distribution functions for some hyperexponential distributions with mean 1, and Figure 2.5 shows some hypoexponential distributions with mean 1.

## 2.2 SCHEDULING ALGORITHMS

Besides the *mean* service time of a program using a resource, the most important characteristic of a resource is the *scheduling algorithm* which decides how the resource is to be allocated to the competing programs. (We will often use *queueing discipline* synonymously with "scheduling algorithm.")

The simplest, and occasionally, most appropriate scheduling algorithm is *First-Come-First-Served* (FCFS). However, as we discuss graphically in the next section, FCFS is often inappropriate for resources with highly variable service time distributions. Thus there has been much research on scheduling algorithms, particularly for processors [COFF68, SHER72, SAUE77b].

There are two principle questions in scheduling: (1) Which job should be currently served? and (2) If the job in service is not the one we would now choose (because of a change in the system state) should we allow the job in service to finish or should we preempt it?

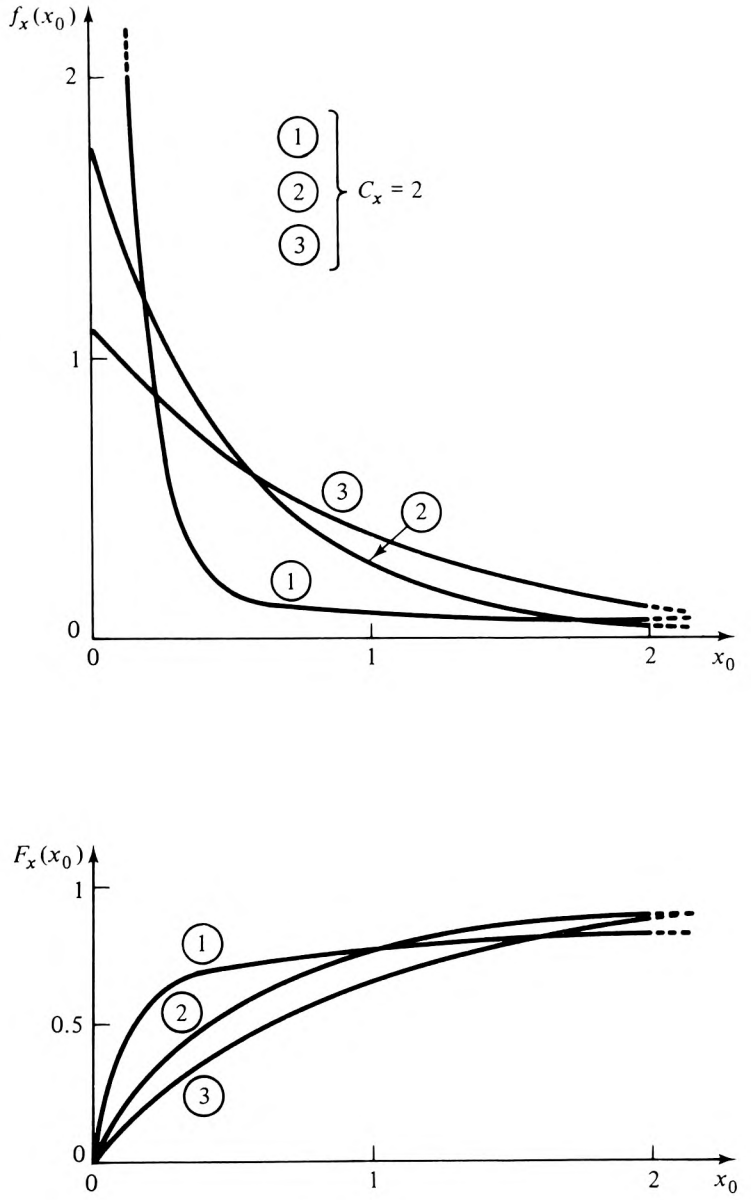



Figure 2.4

With FCFS the choice of job to be served is made according to time of arrival; the job with the earliest arrival time is served. The preemption question does not arise since a change of system state (e.g., an arrival) cannot change the choice.

The opposite of FCFS is *Last-Come-First-Served* (LCFS). With LCFS every arrival changes the choice, so we have to decide about preemption. A

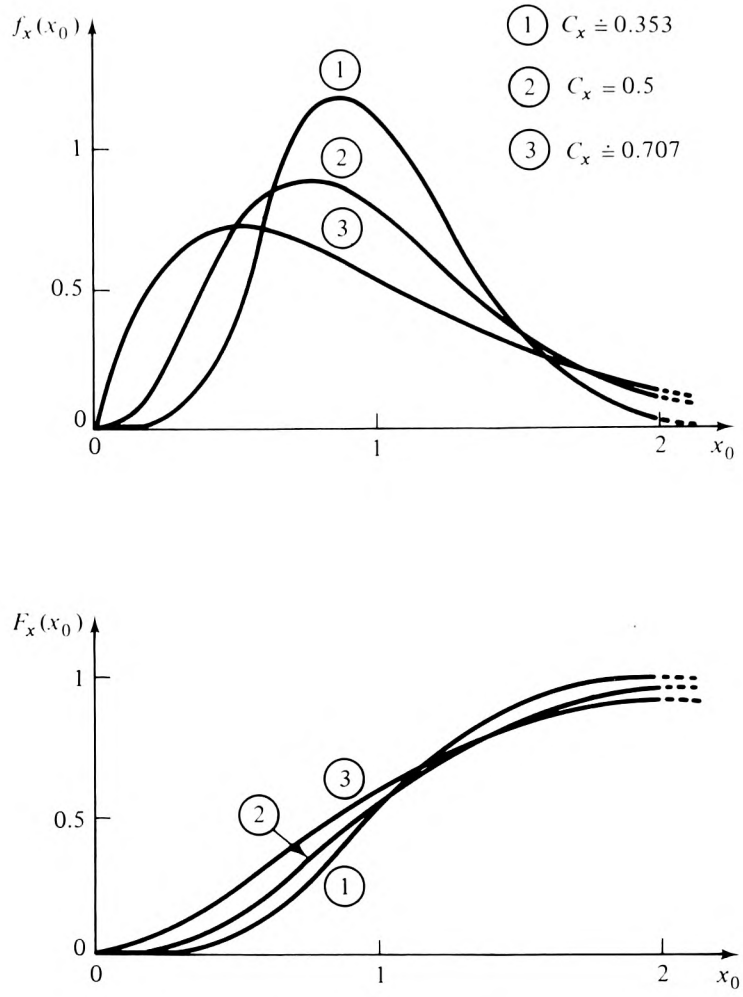



Figure 2.5

principal sub-question with regard to preemption is whether it is possible to resume the preempted job where it left off (when the choice is made to reservice that job) or whether the job's service must begin anew. Processors are usually designed to facilitate resumption of the preempted job (by providing for saving the values of the program counter and other registers). With an I/O device it is usually necessary to reposition the device to service the preempting job; the preempted jobs's service is restarted in the future. Thus preemptive scheduling of I/O devices is unusual. *Last-Come-First-Served-Preemptive-Resume* (LCFSPR), which is strictly preemptive, has some desirable theoretical characteristics and has been used for processor scheduling in interactive systems.

Among the most common processor scheduling algorithms are *round-robin* (RR) and more complex algorithms based on RR. (RR is sometimes referred to as "time-slicing.") RR is defined with respect to an interval of time called the *quantum* (or "time-slice"). Jobs are served in first-come-first-served order as long as their service times do not exceed the quantum. When a job's current service time (ignoring previous quanta) reaches the quantum, the job is preempted (with state information saved for future resumption) and the job is placed at the end of the queue (as if it had just arrived). Thus the job's service time is broken up into several quanta, all but the last of which are of fixed length. The processor is effectively shared among the jobs; a job with short service time is not forced to wait for the completion of service times of jobs ahead of it. The principal sub-question with RR is the choice of quantum size. There will be *overhead* (processor time used in implementing the scheduling and preemptions). If the overhead is large relative to the quantum, then performance will suffer. This overhead is usually at least a hundred microseconds per preemption. A common rule of thumb is to choose a quantum roughly two orders of magnitude larger than the overhead for a preemption. (As the quantum becomes large, RR becomes the same as FCFS and the sharing effects are lost.)

The *processor sharing* (PS) discipline can not be actually implemented, but is valuable in modeling RR scheduling. PS is defined as the limiting case of the RR algorithm with zero overhead as the quantum goes to zero. PS is usually a reasonable representation of RR when the quantum is very large with respect to the overhead and small with respect to the mean service time.

In terms of minimizing mean response times, the optimal scheduling algorithm is *shortest-remaining-time-first* (SRTF). SRTF always chooses to serve the job with the smallest remaining service time. If an arriving job has a smaller service time than the job in service, the arriving job preempts. (SRTF is intuitively optimal with respect to mean response time since it maximizes the number of response times completed in a given interval of time.)

SRTF assumes service times are known in advance, and this is not usually the case with processors. However, in modeling we can use SRTF as a standard for comparison with practical algorithms. It is often possible to use past behavior as a predictor of future behavior and approximate SRTF [SHER72]. For I/O devices we may very well be able to determine service times in advance (although we cannot use preemption). For drum (-like) devices *shortest-latency-time-first* (SLTF) is often optimal [FULL75]. For moving head disks, *shortest-seek-time-first* (SSTF) is often considered optimal but there are exceptions [TEOR72, WILH76].

We have assumed that jobs are not given *external priorities* (e.g. by the system administration). If this is not the case, then the above algorithms can be applied within a priority group after jobs are classified by priorities. Preemption may be applied across priority groups as well as within groups. (We use the term "external priority" since almost all algorithms have implicit *internal priorities*, e.g., FCFS has internal priorities based on arrival times.)

## 2.3 PERFORMANCE RELATIONSHIPS BETWEEN DISTRIBUTIONS AND SCHEDULING

In order to make our discussions more specific, in this section we will assume a cyclic queue model of the type used by Chiu *et al* discussed in the last chapter. The conclusions we draw apply in much more general circumstances, including all of the closed networks discussed in this book. Similar conclusions apply in open networks.

In determining performance, our measure will be throughput, the mean number of jobs passing through a point in the cyclic network per unit of time. As we will see in the next section, maximum throughput in this model coincides with maximum resource utilization and minimum mean response time, assuming that workload remains unchanged.

If service times are highly variable, i.e., $C_x > 1$, then most of the service times will be much smaller than the mean, but a few will be much much larger. (For the exponential distribution, which has $C_x = 1$, about 22% of the times will be less than one-fourth of the mean, about 39% of the times will be less than half the mean and about 63% of the times will be less than mean.) This variability causes poor performance with FCFS scheduling, since one job with long service time will delay many other jobs with short service times once it begins service. This effect is compounded by effects on other resources — if all of the jobs pile up at the CPU queue, then the I/O devices become idle and the performance benefit of multiprogramming is lost.

With RR, the effect of high variability in service times is much less noticeable. A job with a long service time is preempted after reaching the quantum and jobs with short service times proceed with very little interference from the long job. As we will show in Chapters 4 and 5, the limiting case of RR, PS, is insensitive to all distribution characteristics other than the mean (and gives the same performance as FCFS with exponential service times).

Let us assume that our cyclic queue model has a single processor, five I/O devices and five jobs. The mean CPU service is 10 ms. and the mean

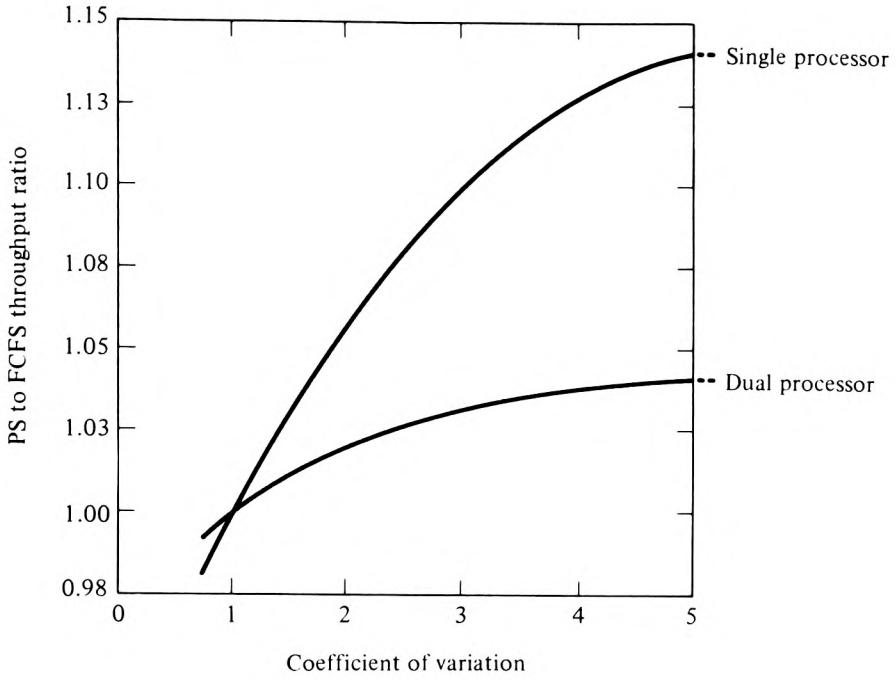



Figure 2.6

I/O service is 50 ms. (Thus the effective rate of the I/O system is the same as the processor when all devices are active.) Regardless of distribution characteristics other than the mean, with PS the throughput will be roughly .072 jobs/ms., the processor utilization will be 72% and each I/O will have 72% utilization. These values are easily obtained by the methods of Chapter 5. Regardless of I/O distribution forms, with FCFS these measures will be higher for $C_x < 1$, the same for $C_x = 1$ and lower for $C_x > 1$. (The FCFS results can be obtained, with more effort, by the methods of Chapter 3.) Figure 2.6 compares the PS throughputs to FCFS throughputs for processor service times with $.75 \leq C_x \leq 5$. Figure 2.6 also gives similar results with the single processor replaced by two processors which are half as fast, i.e., the mean CPU service becomes 20 ms. (The PS and FCFS exponential results give a throughput of .067 and utilizations of 67%.) Notice that the difference between scheduling algorithms is much smaller in the dual processor case. With two processors, a single job with long service time is much less disruptive in the FCFS case; other jobs can use the second processor. Figure 2.7 is similar to Figure 2.6 but compares SRTF to FCFS. We will pursue this discussion again in Chapter 9.

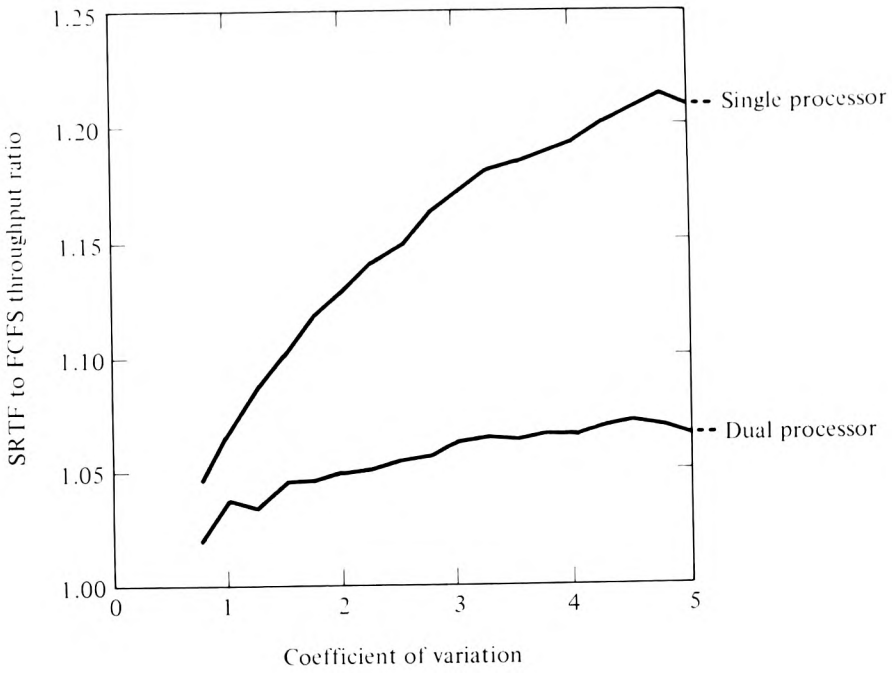



Figure 2.7

## 2.4 RELATIONSHIPS BETWEEN PERFORMANCE MEASURES

In this section we discuss two very simple relationships between performance measures. More complex relationships will be evident from results in the remaining chapters.

### 2.4.1 Throughput, Utilization and Mean Service Time

Let us assume (as we almost always will) that the system has attained equilibrium, in particular, that the flow of jobs out of a queue (for a resource) is equal to the flow into the queue. We call the flow the throughput. Suppose we have a single resource at the queue and the effective service rate of the server is independent of queue length. (This last assumption will not hold for most preemptive schedules with non-zero overhead.) Since the mean service time per job is $E[x]$, the mean service rate is $1/E[x]$. Call the utilization, the fraction of time the resource is busy, $U$. The throughput must be equal to the service rate of the resource, when it is busy, times the fraction of time it is busy. In other words

$$\text{throughput} = \frac{U}{E[x]}. \tag{2.7}$$

If there are $k$ identical units of the resource, $U$ is the utilization of each unit, and the other assumptions above are valid, then

$$\text{throughput} = \frac{kU}{E[x]}. \tag{2.8}$$

Similar results can be obtained from the queue length distribution in more complex situations. (The methods of subsequent chapters can be used to obtain the queue length distribution.)

### 2.4.2 Throughput, Mean Queue Length and Mean Response Time

Let us define the queue length as the number of jobs waiting for or using a resource. Let us define queueing time (response time) as the time between arrival of a job at a (queue for a) resource and completion of use of the resource. The mean queue length can be obtained as

$$L \equiv E[l] = \sum_{l=1}^{\infty} lP(l)$$

where $P(l)$ is the probability of queue length $l$. The mean queueing time can be obtained as

$$Q \equiv E[q] = \int_0^{\infty} q_0 f_q(q_0) dq_0$$

where $f_q(q_0)$ is the density function for the queueing times. Though not immediately obvious, it is generally true that

$$L = \lambda Q \tag{2.9}$$

where $\lambda$ is the throughput. This result was first formally proved by J.D.C. Little [LITT61] and is known as "Little's Rule."

We will intuitively justify (2.9) for a finite period of observed time and FCFS scheduling. Neither of these assumptions are necessary for formal proof; Little's Rule holds for all models in this book and under most other circumstances. (Since Little's proof, simpler proofs have been devised; see KOBA78, for example, for a formal proof and a statement of sufficient conditions for (2.9).)

Consider Figure 2.8, which shows queue length versus time. The area under $l(t)$ is partitioned and numbered according to the sequence of the

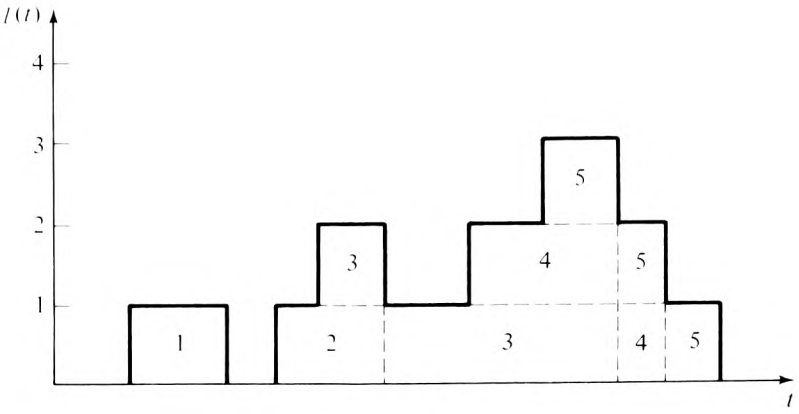



Figure 2.8

jobs' arrivals and their queue positions (assuming FCFS). A job is in service when its sequence number is in a rectangle adjacent to the horizontal axis.

Assume we are observing for a period starting at time 0 and ending at time $T$. Let $n(T)$ be the sequence number of the last job to complete service by time $T$. Thus

$$\lambda = \frac{n(T)}{T}.$$

Let $A(T)$ be the area under $l(t)$ up to time $T$, i.e.,

$$A(T) = \int_0^T l(t)dt.$$

We can easily show that

$$L = \frac{A(T)}{T} \tag{2.10}$$

and that

$$Q = \frac{A(T)}{n(T)} \tag{2.11}$$

Thus

$$L = \frac{A(T)}{T} = \frac{n(T)}{T}\frac{A(T)}{n(T)} = \lambda Q.$$

Little's Rule (2.9) also holds if we define $L$ only with respect to jobs not in service and $Q$ as time waiting for service. Further, we can apply (2.9) to sub-groups of jobs, e.g., priority groupings. Finally, we can apply (2.9) to subsystems, e.g., an entire computer system, if $L$ is defined with respect to the number of jobs in the sub-system and $Q$ is defined with respect to total time in the subsystem!

## 2.5 FURTHER READING

A more introductory and thorough treatment of the material of Section 2.1 can be found in DRAK67 and FELL68. Further discussion in general is found in KOBA78 and more discussion of individual topics is found in the cited references.

## 2.6 EXERCISES

2.1 Derive (2.2) from (2.1)
2.2 Derive (2.4) from repeated application of (2.3)
2.3 Derive (2.5)
2.4 Derive (2.6)
2.5 Justify (2.10) and (2.11)

## 2.7 SUMMARY OF CHAPTER NOTATION

| | |
|---|---|
| $P(x)$ | Discrete distribution: probability of value $x$ |
| $E[x]$ | Mean (expected) value of random variable $x$ |
| $E[x^n]$ | $n^{th}$ moment of random variable $x$ |
| $\sigma_x$ | Standard deviation of random variable $x$ |
| $C_x$ | Coefficient of variation of random variable $x$ |
| $F_x(x_0)$ | Continuous distribution: probability distribution function of random variable $x$, i.e., probability $x \leq x_0$ |
| $f_x(x_0)$ | Continuous distribution: probability density function of random variable $x$ |
| $U$ | Utilization of a (unit of a) resource |
| $L$ | Mean queue length for a resource, including jobs in service |
| $Q$ | Mean queueing time for a resource, including service time |

# *CHAPTER 3*

# MARKOVIAN QUEUEING MODELS OF COMPUTER SYSTEMS

Markov processes are extremely powerful tools which can be used to provide accurate, yet mathematically tractable, models of computing systems performance. This chapter will provide an informal description of a subset of Markov processes which is sufficient to describe very general queueing network models of computing systems. (This subset is also sufficient for description of many other computing system models, but we will restrict attention to queueing network models.) There are three issues we will face: (1) definition of Markov processes, (2) mapping computer system models to Markov processes, and (3) solution of Markov processes.

Performance models are usually used to estimate the performance of computing systems over a period of time. The time period may be explicit for some performance measures and implicit for others. The two most important measures, *throughput* and *response time*, represent explicit and implicit time periods, respectively. Throughput is measured in the amount of work, e.g., the number of batch programs or interactive commands, handled during a time period. Though we might wish to estimate the response time for an individual command (or the turnaround time for an individual batch job), usually we will be content to estimate the mean or some other measure of the response time distribution. (We might estimate the fraction of response times that exceed 3 seconds, for example.) In this second case we usually have a period of time in mind, whether or not it is well defined. The period may be the entire day, or a portion of a morning when the system is lightly loaded, or late in the afternoon when everyone is trying to get finished and go home, etc.

## 3.1 DIFFICULTY OF FINDING TRACTABLE REPRESENTATIONS OF COMPUTING SYSTEMS

Since a multiprogrammed computer exhibits very dynamic behavior, and since the behavior at a particular time depends strongly on contention for and sharing of resources, a model of a given system must attempt to represent the internal state of the system. However, a detailed representation of internal state will usually leave us without solution methods other than direct simulation. For example, towards the extreme of detailed representation, if we paid attention in our state representation to the in-

struction streams of individual programs and the data for those programs, then the most appropriate solution method would be to use the system itself!

The representation of internal state gives us memory of the current and past condition of the system. If this memory includes too little detail then we will have difficulty estimating the behavior of the system. However, if we allow very much memory we will find numerical solution impractical.

Let us consider some simplified representations which still preclude tractable solutions. (We claim, without proof, that these representations *usually*, but not always, preclude tractable solution other than simulation.) Since we have ruled out consideration of specific instruction and data streams, a less detailed representation would be one which only considered system and workload *timings*, e.g., the time until a program using a CPU relinquishes the CPU in order to perform I/O, the times between page faults for a given program and memory policy, or the times between the running of a system scheduler. If we look at the specific times then we will still be overwhelmed with information.

Another simplification is to (1) represent the timings by probability distributions, i.e., a distribution for CPU time used between I/O operations, another for the times between page faults, another for the times between scheduler activations, etc., and (2) assume that successive timings are independent with the respective distributions. However, if we allow arbitrary probability distributions as defined in Chapter 2, we will still be overwhelmed with information. Suppose we wish to represent the time until completion for a request arriving at the CPU when the CPU is busy doing something else. If we wish to estimate the distribution for this period of time, then we will have to determine the distribution for the sum of the time for the request plus the time until the CPU is given the request. This latter time will depend on the time already spent on work ahead of the arriving request and that time would be included in our state representation. Thus our state space will in general be infinite and not countable. If we cannot enumerate the state space of our representation, then we can only hope for a solution under very restricted conditions.

We have an apparent impasse — we want to study the time dependent behavior of the computer system but we cannot consider time in our representation of the system. The Markov process representation allows us to consider time in a very controlled manner and thus overcome the apparent obstacle.

## 3.2 MARKOV PROCESSES

### 3.2.1 The Exponential Distribution

The key to the Markov process representation is a very special probability distribution, the (negative) exponential distribution defined in Chapter 2. The exponential probability distribution function has the form $F_x(x_0) = 1 - e^{-ax_0}$, for $x_0 \geq 0$. The parameter $a$ is referred to as the "rate" of the distribution, for reasons which will soon be evident. The mean of the distribution is $1/a$. See Figure 3.1.

The exponential distribution is unique among continuous distributions (those that allow values along continuous portions of the real line) because it has a so called "memoryless" property. The memoryless property is that if we know that a random variable has an exponential distribution, and we know that the value of the random variable is at least some other value, say $t$, then the distribution for the remaining value of the variable (e.g., the difference between the total value and $t$) has the same exponential distribution as the total value. For example, if we know that a program's CPU times between I/O activities have an exponential distribution, and we know that a given CPU time has already lasted 10 milliseconds, then the remainder of the current CPU time will have the same exponential distribution as the total CPU time.

Though timings in computer systems usually do not have exponential distributions, 1) we can often assume the timings do have exponential distributions and get accurate results in spite of the incorrect assumption, and 2) we can use combinations of exponential distributions, as we shall see later in this chapter.

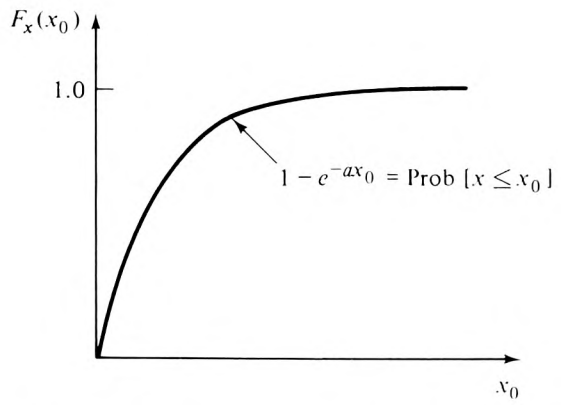



Figure 3.1 - Exponential Distribution

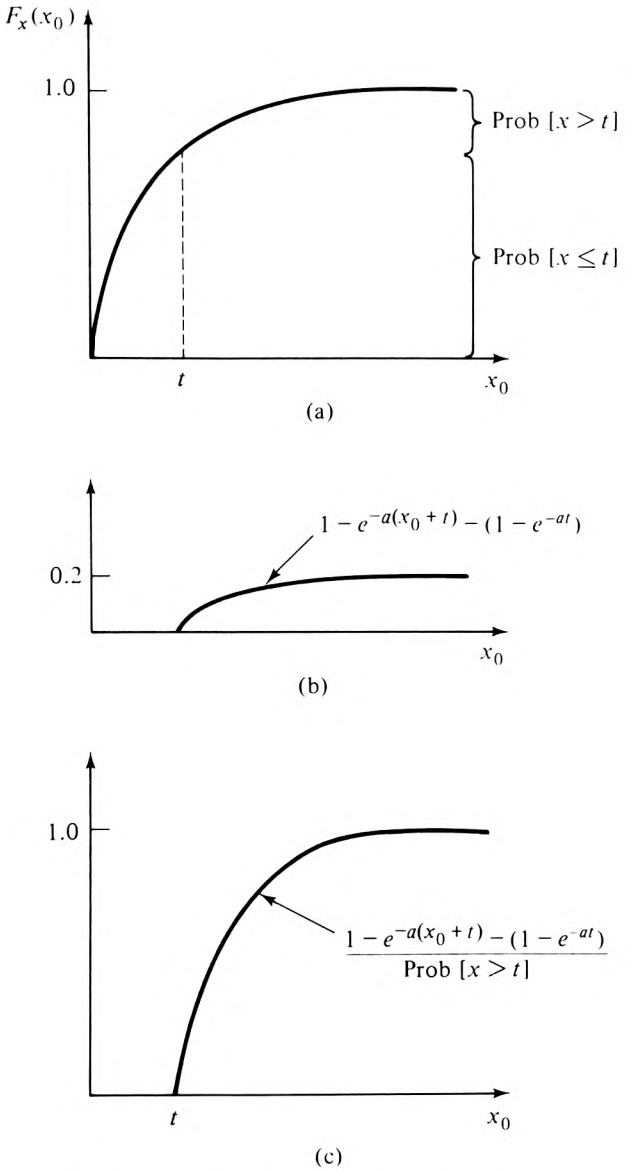



Figure 3.2 - Memoryless Property

We prove that the exponential distribution has the memoryless property as follows. We want to show that Prob[$x \leq x_0 + t$ given $x > t$] = Prob [$x \leq x_0$].

A probability of one event given the occurrence of another event, e.g., Prob[Event A given Event B], is known as a *conditional probability*. We

define, for Prob[Event $B$]>0,

$$\text{Prob[Event } A \text{ given Event } B] = \frac{\text{Prob[Event } A \text{ and Event } B]}{\text{Prob[Event } B]}.$$

We have informally used the term *independence* before. Formally, events A and B are independent if

$$\text{Prob[Event } A \text{ given Event } B] = \text{Prob[Event } A],$$

or equivalently, if

$$\text{Prob[Event } A \text{ and Event } B] = \text{Prob[Event } A]\text{Prob[Event } B].$$

From the definition of conditional probability we know that

$$\begin{aligned}
&\text{Prob}[x \le x_0 + t \text{ given } x > t] \\
&\quad = \frac{\text{Prob}[x < x_0 + t \text{ and } x > t]}{\text{Prob}[x > t]} \\
&\quad = \frac{1 - e^{-a(x_0+t)} - (1 - e^{-at})}{1 - (1 - e^{-at})} \\
&\quad = 1 - e^{-ax_0} \\
&\quad = \text{Prob}[x \le x_0].
\end{aligned}$$

Figure 3.2 illustrates this proof.

### 3.2.2 The Poisson Process

Suppose the times between events in a stream of events are independent and the durations of the inter-event times have the exponential distribution $F_t(t_0) = 1 - e^{-at_0}$. For example, the events might be completion of service at a CPU, when the CPU is busy. Since the mean time between events is $1/a$, the rate of occurrence of events will be $a$. It can be shown that the events form a *Poisson* process.

The Poisson process is defined as follows:

1. Occurrences of events during non-overlapping intervals of time are independent.
2. For a sufficiently small interval of time, $\Delta t$, the probability of zero events occurring during the interval is $1-a\Delta t$, the probability of one event occurring during the interval is $a\Delta t$, and the probability of more than one event during the interval is negligible.

Note that part 1 of the definition gives the Poisson process a memoryless property; occurrence of events during a current interval of time is independent of occurrences in previous intervals. It is equivalent to say that "events form a Poisson process" and that "inter-event times are independent with identical exponential distributions."

Now suppose we have two independent streams, each forming a Poisson process with rates $a_1$ and $a_2$, respectively. Let us consider the merging of the two streams, i.e. the stream of events consisting of all events from both streams. We would like to show that the merged stream is also a Poisson process. Since each stream satisfies part 1 of the definition and since the streams are independent, the merged stream also satisfies part 1 of the definition and since the streams are independent, the merged stream also satisfies part 1 of the definition. The probability of zero arrivals in the merged stream in interval $\Delta t$ is

$$(1 - a_1\Delta t)(1 - a_2\Delta t) = 1 - (a_1 + a_2)\Delta t + a_1a_2\Delta t^2$$

which is $1 - (a_1 + a_2)\Delta t$ for sufficiently small $\Delta t$. The probability of one arrival in the merged stream in interval $\Delta t$ is

$$\begin{aligned}(1 - a_1\Delta t)a_2\Delta t + a_1\Delta t(1 - a_2\Delta t) &= a_2\Delta t - a_1a_2\Delta t^2 + a_1\Delta t - a_1a_2\Delta t^2 \\ &= (a_1 + a_2)\Delta t - 2a_1a_2\Delta t^2\end{aligned}$$

which is $(a_1 + a_2)\Delta t$ for sufficiently small $\Delta t$. The probability of two arrivals in the merged stream in interval $\Delta t$ is

$$a_2\Delta t a_2\Delta t = a_1a_2\Delta t^2$$

which is negligible for sufficiently small $\Delta t$. Thus the merged stream forms a Poisson process with rate $a_1 + a_2$. Further, the inter-event times will have the exponential distribution $F_t(t_0) = 1 - e^{-(a_1+a_2)t_0}$. This last observation is also key to the Markov processes we discuss.

If we observe an event in the merged stream, then with probability

$$\frac{a_1\Delta t}{a_1\Delta t + a_2\Delta t} = \frac{a_1}{a_1 + a_2}$$

it is from stream 1 and with probability

$$\frac{a_2}{a_1 + a_2}$$

it is from stream 2.

Also note that if we accept only certain events, with fixed probability $P$, and reject the other events, then the stream of accepted events is a Poisson process with rate $Pa$.

### 3.2.3 Markovian States

Let us assume that we can represent the possible conditions of a system by a set $S_i$: $\{1 \leq i \leq n\}$ of mutually exclusive, collectively exhaustive *states*. Further, the future behavior of the system is dependent only on the current state of the system, i.e., it is independent of previous states of the system. Finally, the "holding times" for state $S_i$, i.e., the times between corresponding entrances to and departures from state $S_i$, are independent and identically exponentially distributed with rate $a_i$. Then the states for this system are *Markovian*.

We can define a *Markov process* to be a set of Markov states $\{S_i: 1 \leq i \leq n\}$ and a set of transition probabilities $\{q_{ij}: 1 \leq i \leq n, 1 \leq j \leq n, \sum_j q_{ij} = 1\}$. See Figure 3.3.

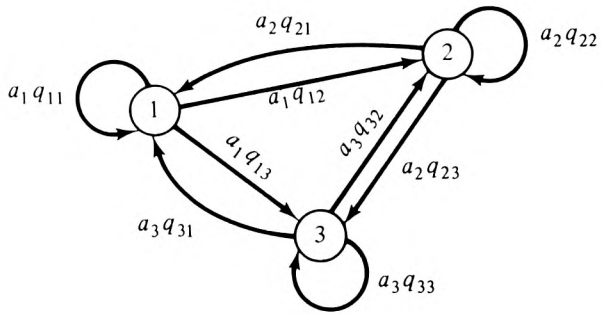



Figure 3.3

The circles in Figure 3.3 represent states of the process and the arcs represent state transitions. If a system is in state $S_i$ then it will make state transitions at rate $a_i$. The rate of transitions from state $S_i$ to $S_j$ given that the system is in state $S_i$, will be $a_iq_{ij}$.

With the above definitions, we note that the Markov process has a very limited amount of memory; the only memory is the current state. Yet time is included in our representation and we can represent very complex, time dependent systems.

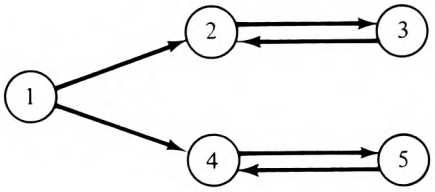



Figure 3.4

### 3.2.4 State Probabilities and Balance Equations

Let us assume that there exists a probability $P_i(t)$ that the system is in state $S_i$ at time $t$. Further, a system equilibrium exists such that

$$\lim_{t \to \infty} P_i(t) = P_i , \quad 1 \leq i \leq n$$

Note that this implies that there is only one possible equilibrium. Figure 3.4 illustrates a Markov process with two equilibria.

Let us also assume that $P_i > 0$ for $1 \leq i \leq n$. Since an equilibrium exists, we expect that the rate of transitions out of state $S_i$ is equal to the rate of transitions into state $S_i$. The rate of transitions out of state $S_i$ will be $P_i a_i$. The rate of transitions into state $S_i$ will be

$$\sum_j P_j \, a_j \, q_{ji} .$$

Thus we have $n$ equations of the form

$$P_i \, a_i = \sum_j P_j \, a_j \, q_{ji} , \; 1 \leq i \leq n.$$

These may be rewritten as

$$-P_i \, a_i \; + \sum_j P_j \, a_j \, q_{ji} \; = 0, \; 1 \leq i \leq n. \tag{3.1}$$

Note that the $n^{th}$ equation is redundant, i.e., given any $n-1$ of the equations we can derive the remaining one. We also know that $\sum_i P_i = 1$. We can solve a set of linear ("balance") equations

$$-P_i a_i + \sum_j P_j a_j q_{ji} = 0, \quad 1 \leq i \leq n-1$$

$$\sum_j P_j = 1$$

to obtain $P_i$, $1 \leq i \leq n$. From these equilibrium probabilities we can obtain performance measures such as mean response time, throughput, utilization, etc.

### 3.2.5 An Example

Consider the closed queueing network of Figure 3.5. In Chapter 1 we described this network as a model of a batch computer system. We now represent this network as a Markov process.

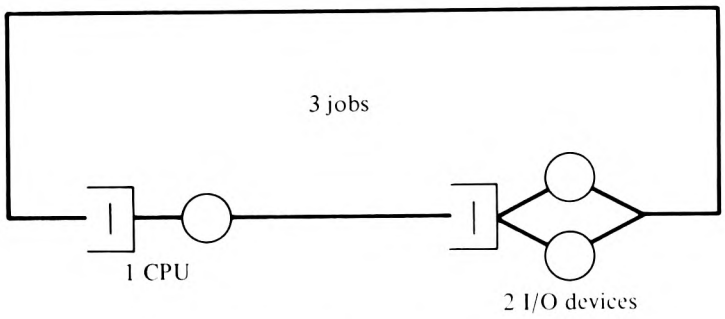



Figure 3.5 - Cyclic Queue Model

Let us assume that there is one CPU, there are two identical I/O devices and the (fixed) degree of multiprogramming is three. As in the work of Chiu *et al*, we assume that both queues have FCFS scheduling disciplines and that service times are independent and identically distributed with exponential distributions. Let the rate of the CPU distribution be $b_1$ and the rate of the I/O distribution be $b_2$.

Given these assumptions, we can define a set of Markov states of this system as {(3,0), (2,1), (1,2), (0,3)} where the couple $(i,j)$ means that there are $i$ jobs at the CPU queue and $j$ jobs at the I/O queue. (We know that $j = 3 - i$, but we include $j$ for clarity.)

It is obvious that these states are mutually exclusive and collectively exhaustive. We can show that the holding time for each state is exponential. For example, the holding time for state (1,2) is exponential with rate $b_1 + 2b_2$ since all three jobs are in service and have exponential service time distributions.

Though we could have other Markov process representations of this system with more states, we must have at least four states in any Markov process description of this system. The first part of this claim is easy to demonstrate by example. On the other hand, if we have fewer than four states for this system we will lose the "independent of previous states" property.

Given a transition out of state (1,2), the probability of entering (0,3) is

$$\frac{b_1}{b_1 + 2b_2}$$

and the probability of entering (2,1) is

$$\frac{2b_2}{b_1 + 2b_2}.$$

Thus when the system is in state (1,2) the rate of flow into (0,3) is

$$(b_1 + 2b_2) \qquad \frac{b_1}{b_1+2b_2} \qquad = b_1.$$

$$\begin{array}{ll} a_{(1,2)} & q_{(1,2),(0,3)} \\ \text{(rate)} & \text{(probability)} \end{array}$$

Similarly, the rate of flow into (2,1) is $2b_2$. The full state transition diagram is given in Figure 3.6.

Using the notation $P(i,j)$ for the equilibrium probability of state $(i,j)$, we have the following equations equating the flow in and out of each state:

$$\begin{array}{lccccc} \text{state} & \text{equation} & & & & \\ (3,0) & -b_1P(3,0) & +b_2P(2,1) & & & = 0 \\ (2,1) & b_1P(3,0) & -(b_1+b_2)P(2,1) & +2b_2P(1,2) & & = 0 \\ (1,2) & & b_1P(2,1) & -(b_1+2b_2)P(1,2) & +2b_2P(0,3) & = 0 \\ (0,3) & & & b_1P(1,2) & -2b_2P(0,3) & = 0 \end{array}$$

Notice that any one of these may be obtained from the other three. We can replace any one equation by

$$P(3,0) + P(2,1) + P(1,2) + P(0,3) = 1$$

and solve for each probability. Doing so we obtain

$$P(3,0) = b_1^{-3}/G \tag{3.2}$$

$$P(2,1) = b_1^{-2}b_2^{-1}/G \tag{3.3}$$

$$P(1,2) = \frac{1}{2}b_1^{-1}b_2^{-2}/G \tag{3.4}$$

$$P(0,3) = \frac{1}{4}b_2^{-3}/G \tag{3.5}$$

where

$$G = b_1^{-3} + b_1^{-2}b_2^{-1} + \frac{1}{2}b_1^{-1}b_2^{-2} + \frac{1}{4}b_2^{-3}.$$

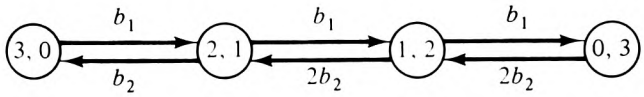



Figure 3.6

This solution is easily verified by substitution. Note that $G$ explicitly forces the probabilities to sum to 1; it is called a "normalizing" constant. From these probabilities we can easily obtain performance measures of interest. Some interesting measures are

CPU utilization = $P(3,0) + P(2,1) + P(1,2)$

CPU throughput = $b_1(P(3,0) + P(2,1) + P(1,2))$

CPU mean queue length (including job in service)
= $3P(3,0) + 2P(2,1) + P(1,2)$

CPU mean queueing time (including service)
= (CPU mean queue length)/(CPU throughput)

I/O utilization (of each identical device)
= $.5P(2,1) + P(1,2) + P(0,3)$.

As we shall show in Chapter 5, this model belongs to the class of models with a "product form" solution. For models with product form solution we need not explicitly solve for state probabilities to obtain many interesting performance measures, including the ones above. However, many interesting models will not have product form solutions. For these we must use numerical solution, approximation (Chapter 6) or simulation

(Chapter 7). The remainder of this chapter will consider methods for numerical solution.

## 3.3 SPARSE MATRIX SOLUTIONS

A set of equations for the previous example may be represented in matrix notation as

$$Bp = e$$

where $p$ is a column vector $(P(3,0), P(2,1), P(1,2), P(0,3))^T$ (T stands for transpose), $e$ is a column vector $(0,0,0,1)^T$ and $B$ is a matrix

$$\begin{bmatrix} -b_1 & b_2 & 0 & 0 \\ b_1 & -(b_1+b_2) & 2b_2 & 0 \\ 0 & b_1 & -(b_1+2b_2) & 2b_2 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

(We are omitting the equation equating flow in and out of state (0,3).)

For this model we may use any one of a variety of methods to obtain a solution for the vector $p$. However, more complex models may result in very large sets of states (possibly infinite). For the purposes of this section let us assume that the number of states is on the order of a few thousand.

An immediate observation is that it is impractical to store the state transition matrix (e.g., $B$) as a two dimensional array. If there are a few thousand states, then such an array would require several million words of storage. Fortunately, for queueing models of computer systems, the state transition matrix is usually *sparse*, i.e., most of the elements are zero. Thus we can use a data structure which only stores the non-zero elements. For example, we can use a table of triples (*row*, *col*, *value*) where *row* and *col* are the subscripts of a non-zero element. For the matrix $B$ we should have

| | *row* | *col* | *value* |
|---|---|---|---|
| 1. | 1 | 1 | $-b_1$ |
| 2. | 1 | 2 | $b_2$ |
| 3. | 2 | 1 | $b_1$ |
| 4. | 2 | 2 | $-(b_1+b_2)$ |
| | etc. | | |

(Though $B$ is not sparse, note that if we increase the degree of multiprogramming then the corresponding matrix will have roughly $3N$ non-zero elements, where $N$ is the degree of multiprogramming. Thus the fraction of

non-zero elements is roughly $3/N$.)

A direct approach to solving the balance equations, such as Gaussian elimination, will be inappropriate because it will change the elements of the matrix, making many elements non-zero which previously had been zero. The increase in storage will often be prohibitive. There are other disadvantages of direct solutions which we will ignore.

The interesting alternatives to direct solution are iterative solutions and specialized recursive algorithms. We will defer discussion of the recursive algorithms until the end of the chapter. The iterative solutions are generally applicable provided the number of states is not too large.

Let $S$ be a matrix such that

$$S_{ij} = \begin{cases} a_j q_{ji}, & i \neq j \\ -a_i (1 - q_{ii}), & i = j \end{cases}$$

i.e., $S$ is the matrix corresponding to the redundant set of equations (3.1) without replacing one by $\Sigma P_i = 1$. Then $Sp=0$ where $p$ is the column vector $(P_1,P_2...,P_n)^T$ and 0 is the column vector containing all zeroes. Then we also have that $\Delta Sp = 0$ where $\Delta$ is an arbitrary scalar. Further $\Delta Sp + p = p$ so that $(\Delta S + I)p = p$, where $I$ is the identity matrix consisting of ones on the diagonal and zeroes elsewhere.

The last equation suggests the iterative formula

$$\hat{p}^{k+1} = (\Delta S + I)\hat{p}^k$$

where $\hat{p}^k$ is the estimate for $p$ on the $k^{th}$ iteration. It can be shown that, with appropriate choice of $\Delta$, this iteration will converge to $p$ with any initial estimate $\hat{p}^0$. As discussed in WALL66, a value of $\Delta$ which is usually appropriate is

$$\frac{.99}{\max_i |S_{ii}|}$$

Note that the iteration leaves $\Delta S + I$ unchanged.

As a numerical example consider the model of Section 3.2.5 with mean CPU service 6.67 ms. and mean I/O service 10 ms. Thus $b_1 = .15$ jobs/ms. and $b_2 = .1$ job/ms. From (3.2-3.5) we know that $P(3,0) = .224$, $P(2,1) = .336$, $P(1,2) = .252$ and $P(0,3) = .189$. (Thus the CPU utilization is 81% and the I/O utilization is 61%.)

Figure 3.7 shows a PASCAL program which applies this iterative solution method to this model. Figure 3.8 shows the output of this program. The program is written to allow an arbitrary number of jobs in the system and an arbitrary number of I/O devices. Note that mechanically generating the matrix $S$ is non-trivial for more complex systems. The iteration terminates when the error estimate (the sum of the magnitudes of the differences between the elements of $\hat{p}^k$ and $\hat{p}^{k+1}$) is small. Note that the algorithm converges in spite of $\hat{p}^0$ being so poor. ($\hat{p}^0(3,0) = 1$, $\hat{p}^0(2,1) = \hat{p}^0(1,2) = \hat{p}^0(0,3) = 0$.)

```
PROGRAM ITERATE(OUTPUT);
(*PROGRAM TO ITERATIVELY SOLVE BALANCE EQUATIONS
  FOR A CYCLIC QUEUE MODEL OF A COMPUTER SYSTEM.  ASSUMES
  FIXED DEGREE OF MULTIPROGRAMMING, EXPONENTIAL CPU TIMES
  WITH MEAN 1/B1, ONE CPU, EXPONENTIAL I/O TIMES WITH MEAN
  1/B2 AND NIO I/O DEVICES*)
CONST B1=0.15; B2=0.1; NIO=2;
      (*NSTATES=DEG. OF M.P. + 1
        MATSIZE=3*NSTATES - 2*)
      NSTATES=4; MATSIZE=10;
      TOLERANCE=0.001;
TYPE  PVECTOR=ARRAY[1..NSTATES] OF REAL;
VAR   MATRIX: ARRAY[1..MATSIZE] OF
        RECORD
          ROW, COL: 1..NSTATES;
          VALUE: REAL
        END;
      OLDP, NEWP: PVECTOR;
      MAXDIAG, DELTA: REAL;
      I, ITERATION: INTEGER;
FUNCTION MIN(V1,V2:INTEGER):INTEGER;
  BEGIN
    IF V1<V2 THEN MIN:=V1
    ELSE          MIN:=V2
  END; (*MIN*)
FUNCTION NORM(V1, V2: PVECTOR): REAL;
  VAR I: INTEGER;
      TEMP: REAL;
  BEGIN
    TEMP:=0.0;
    FOR I:=1 TO NSTATES DO
      TEMP:=TEMP+ABS(V1[I]-V2[I]);
    NORM:=TEMP
  END; (*NORM*)
BEGIN
```

```
(*BUILD TRANSITION MATRIX.  STATE I HAS NSTATES-1-(I-1)
  JOBS AT CPU, I-1 JOBS AT I/O*)
WITH MATRIX[1] DO
  BEGIN ROW:=1; COL:=1; VALUE:=-B1 END;
WITH MATRIX[2] DO
  BEGIN ROW:=1; COL:=2; VALUE:=B2 END;
FOR I:=2 TO NSTATES-1 DO
  BEGIN
    WITH MATRIX[3*I-3] DO
      BEGIN ROW:=I; COL:=I-1; VALUE:=B1 END;
    WITH MATRIX[3*I-2] DO
      BEGIN ROW:=I; COL:=I; VALUE:=-B1-MIN(I-1,NIO)*B2
                                                    END;
    WITH MATRIX[3*I-1] DO
      BEGIN ROW:=I; COL:=I+1; VALUE:=MIN(I,NIO)*B2 END
  END;
WITH MATRIX[MATSIZE-1] DO
  BEGIN ROW:=NSTATES; COL:=NSTATES-1; VALUE:=B1 END;
WITH MATRIX[MATSIZE] DO
  BEGIN ROW:=NSTATES; COL:=NSTATES; VALUE:=-MIN(NSTATES-1,
                                          NIO)*B2 END;

(*DETERMINE DELTA*)
MAXDIAG:=0.0;
FOR I:=1 TO MATSIZE DO
  WITH MATRIX[I] DO
    IF ROW=COL THEN
      IF ABS(VALUE)>MAXDIAG THEN
        MAXDIAG:=ABS(VALUE);
DELTA:=0.99/MAXDIAG;
(*MULTIPLY TRANSITION MATRIX BY DELTA AND ADD IDENTITY
  MATRIX*)
FOR I:=1 TO MATSIZE DO
  WITH MATRIX[I] DO
    BEGIN
      VALUE:=DELTA*VALUE;
      IF ROW=COL THEN VALUE:=VALUE+1.0
    END;
(*INITIAL ESTIMATE OF STATE PROBABILITIES*)
NEWP[1]:=1.0;
FOR I:=2 TO NSTATES DO
  NEWP[I]:=0.0;
ITERATION:=0;
WRITE('  ':11,'P');
  FOR I:=1 TO NSTATES DO
```

```
      WRITE(I:7);
    WRITELN;
  (*ITERATIVE REFINEMENT OF ESTIMATE*)
  REPEAT
    ITERATION:=ITERATION+1;
    FOR I:=1 TO NSTATES DO
      BEGIN
        OLDP[I]:=NEWP[I];
        NEWP[I]:=0.0
      END;
    FOR I:=1 TO MATSIZE DO
      WITH MATRIX[I] DO
        NEWP[ROW]:=NEWP[ROW] + VALUE*OLDP[COL];
    WRITE('ITERATION',ITERATION:3);
      FOR I:=1 TO NSTATES DO
        WRITE(NEWP[I]:7:4);
      WRITELN(' ERROR',NORM(OLDP,NEWP):7:4)
  UNTIL NORM(OLDP,NEWP)<TOLERANCE
END.
```

Figure 3.7

```
            P       1       2       3       4
ITERATION   1 0.5757 0.4242 0.0     0.0     ERROR 0.8485
ITERATION   2 0.4514 0.3685 0.1800 0.0     ERROR 0.3600
ITERATION   3 0.3641 0.4013 0.1581 0.0763 ERROR 0.2183
ITERATION   4 0.3231 0.3615 0.2150 0.1002 ERROR 0.1615
ITERATION   5 0.2883 0.3646 0.2122 0.1347 ERROR 0.0753
ITERATION   6 0.2691 0.3491 0.2330 0.1485 ERROR 0.0692
ITERATION   7 0.2537 0.3483 0.2345 0.1634 ERROR 0.0325
ITERATION   8 0.2445 0.3423 0.2425 0.1704 ERROR 0.0301
ITERATION   9 0.2376 0.3412 0.2441 0.1769 ERROR 0.0160
ITERATION  10 0.2333 0.3388 0.2473 0.1804 ERROR 0.0133
ITERATION  11 0.2301 0.3381 0.2483 0.1833 ERROR 0.0077
ITERATION  12 0.2281 0.3371 0.2496 0.1849 ERROR 0.0060
ITERATION  13 0.2267 0.3367 0.2501 0.1862 ERROR 0.0036
ITERATION  14 0.2258 0.3363 0.2507 0.1870 ERROR 0.0027
ITERATION  15 0.2251 0.3361 0.2510 0.1876 ERROR 0.0017
ITERATION  16 0.2247 0.3359 0.2512 0.1879 ERROR 0.0012
ITERATION  17 0.2244 0.3359 0.2514 0.1882 ERROR 0.0007
```

Figure 3.8

## 3.4 EXPONENTIAL STAGES REPRESENTATIONS OF DISTRIBUTIONS

When an exponential distribution is both unrealistic *and* unsatisfactory for representing service (or inter-arrival) time distributions then the usual approach is to use the "method of (exponential) stages." This method is both general and compatible with definition of Markov processes. It is general in that we can represent arbitrary distributions arbitrarily closely. It is compatible with Markov processes because the only memory introduced is the distribution stage. To accommodate this additional memory we merely refine our state definition.

Let us define a service time to consist of visits to one or more of $k$ subservers (stages), each visit having an exponential distribution with rate associated with the subserver. When a job is visiting a subserver, all other jobs are prevented from visiting any subservers of that server. Figure 3.9 illustrates this general description where we place no restriction on the routing of the job among subservers.

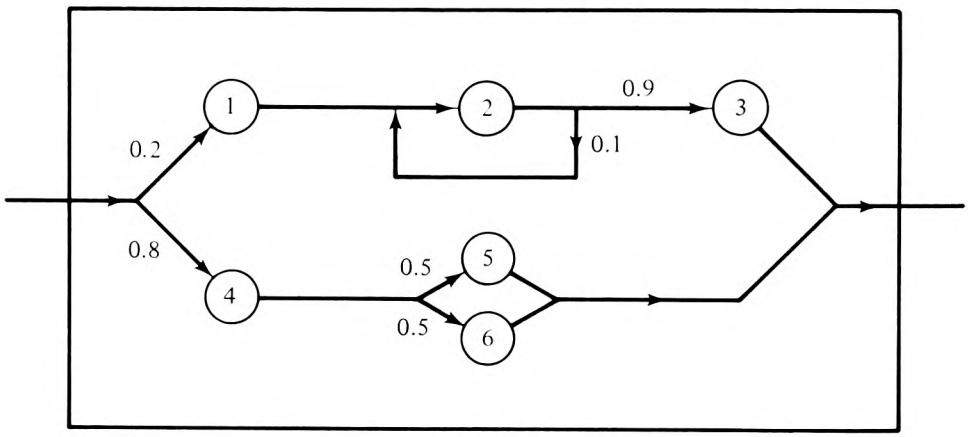



Figure 3.9 - Method of Stages

Only one job is allowed inside the rectangle at a time. A job initially enters subserver (stage) $i$ with probability $V_{0i}$, a job leaving subserver $i$ visits subserver $j$ with probability $V_{ij}$, and a job leaving subserver $i$ departs the entire server with probability $V_{i0}$. For Figure 3.9 all of these probabilities are zero except as follows: $V_{01} = .2$, $V_{04} = .8$, $V_{12} = 1$, $V_{23} = .9$, $V_{22} = .1$, $V_{45} = .5$, $V_{46} = .5$, $V_{30} = 1$, $V_{50} = 1$ and $V_{60} = 1$. Each subserver has an exponential visit time with rate $a_i$.

Some well known special cases of this representation are

1. Erlang: $a_1 = a_2 = \ldots = a_k$, $V_{01} = 1$, $V_{k0} = 1$, $V_{12} = V_{23} = \ldots V_{k-1,k} = 1$.
2. Hypo-exponential: same as Erlang but equality of rates not required.
3. Hyper-exponential: $V_{ij} = 0$ unless i or j but not both are zero.

The reader may find it helpful to draw diagrams analogous to Figure 3.9 for these cases.

There is very little generality lost if we restrict ourselves to the so called "branching Erlang" case of Figure 3.10. We do not require equality of the rates of the visit times. (It can be rigorously shown that the branching Erlang is as general as our original description if we allow the artifice of complex values for the rates and probabilities. We will not pursue this artifice.)

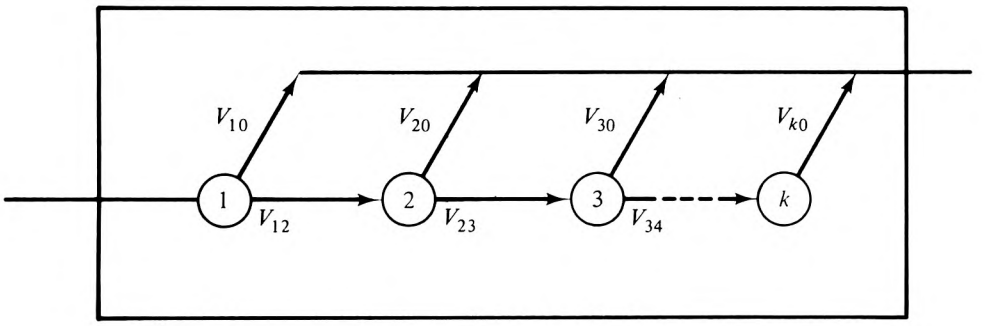



Figure 3.10 - Branching Erlang

It is immediately apparent that the Erlang and hypo-exponential cases are included in the branching Erlang cases. It is not apparent, but still true, that many other special cases of our original description are equivalent to the branching Erlang with judicious choice of the probabilities. For example, consider the hyper-exponential case with $k = 2$. Let us assume that $a_1 \neq a_2$, $V_{01} > 0$ and $V_{02} > 0$ since we would simply have the exponential case without these assumptions. Further, the labeling of the subservers is unimportant for the hyperexponential distribution, so we can assume $a_1 > a_2$. The distribution function for the hyper-exponential case is

$$F_x(x_0) = 1 - V_{01}e^{a_1 x_0} - V_{02}e^{-a_2 x_0}$$

and the distribution function for the branching Erlang case is

$$F_x(x_0) = 1 - \frac{V_{10}a_1 - V_{12}a_2}{a_1 - a_2}e^{-a_1x_0} - \frac{V_{12}a_1 - a_2V_{10}}{a_1 - a_2}e^{-a_2x_0}$$

for $a_1 \neq a_2$. If we define $V_{10}$ for the branching Erlang to be $V_{01} + (1 - V_{01})a_2/a_1$, then it is easily shown that these functions are identical. However, we cannot represent an arbitrary branching Erlang form with the hyper-exponential. The branching Erlang is most convenient for the solution algorithms of the next section.

Usually we will be satisfied to find a method of stages representation which matches the mean and variance of an observed or assumed distribution. The exponential distribution allows us to match the mean but the variance will always be the square of the mean. Thus we will usually not be able to match the mean and the variance with the exponential.

For the branching Erlang form the mean is given by

$$E[x] = \sum_{i=1}^{k} V_{12}V_{23}V_{34}\cdots V_{i-1,i}\ V_{i0}\sum_{j=1}^{i}\frac{1}{a_j},$$

the second moment is given by

$$E[x^2] = \sum_{i=1}^{k} V_{12}V_{23}V_{34}\cdots V_{i-1,i}\ V_{i0}\left[\sum_{j=1}^{i}\frac{1}{a_j^2} + \left(\sum_{j=1}^{i}\frac{1}{a_j}\right)^2\right].$$

and the variance is given by

$$\sigma_x^2 = E[x^2] - (E[x])^2.$$

The complexity of these expressions is a direct result of the generality of the form. However, since we are only interested in the mean and variance, we can simplify things considerably by making arbitrary restrictions on the values of the distribution parameters, i.e., $k$, $a_i$ and $V_{ij}$.

Our discussion will also be simplified if we consider the mean and the coefficient of variation. Since the coefficient of variation is defined to be the standard deviation divided by the mean, i.e., $C_x = \sigma_x/E(x)$, if we match the mean and coefficient of variation then we have also matched the variance. Note that $C_x$ is always 1 for an exponential distribution.

An inherent limitation of the method of stages is that $C_x \geq 1/\sqrt{k}$. Thus many stages are required for small $C_x$. Fortunately, when $C_x$ is small, small changes in its value have very little effect on the results of queueing models. Thus we can arbitrarily enforce a reasonable upper bound on $k$.

Returning to restricted forms of the branching Erlang let us consider two separate cases, $1/\sqrt{k} \leq C_x \leq 1$ and $C_x > 1$. For the first of these a traditional approach is to let $a_1 = a_2 = \ldots = a_k$ and allow $V_{i0}$ and $V_{0i}$ to be nonzero for at most one fixed value of $i \neq k$. (The other traditional approach is the hypoexponential form.)

For example, let us say that only $V_{10}$ and $V_{k0}$ may be non-zero. ($V_{k0}$ must be identically one for the branching Erlang form.) Let $k = \text{ceil}(1/C_x^2)$, i.e., the smallest integer at least equal to $1/C_x^2$. Then $V_{10}$ is uniquely determined by

$$V_{10} = \frac{2kC_x^2 + k - 2 - \sqrt{(k^2 + 4 - 4kC_x^2)}}{2(C_x^2 + 1)(k-1)}$$

and $a_1 = a_2 = \ldots = a_k = (k - V_{10}(k-1))/E[x]$. Though this special case has minor efficiency advantages in simulations there are no compelling reasons for this choice.

For the case $C_x > 1$ we only need for $k$ to be at least 2. With $k = 2$ we have three free parameters, $V_{10}$, $a_1$, and $a_2$, so we may make additional constraints. One possible constraint is that each subserver make an equal contribution to the mean. The following choices accomplish this:

$$V_{10} = C_x^2\left(1 - \sqrt{1 - \frac{2}{1+C_x^2}}\right)$$

$$a_1 = \frac{\left(1 + \sqrt{1 - \frac{2}{1+C_x^2}}\right)}{E[x]}$$

$$a_2 = \frac{\left(1 - \sqrt{1 - \frac{2}{1+C_x^2}}\right)}{E[x]}.$$

Again we point out the choice of additional constraints is reasonable but arbitrary. Alternate choices might be more appropriate in a particular situation.

## 3.5 RECURSIVE SOLUTION METHODS

In many cases a direct numerical solution of the balance equations, e.g., Gaussian elimination, will require excessive memory and computation. This may also be true of the iterative method. So called "recursive" solutions may usually be applied to a fixed class of models, e.g., the model of Figure 3.5 with an arbitrary number of jobs, an arbitrary branching Erlang distribution at the CPU, FCFS scheduling at both queues and an arbitrary number of I/O devices has a single solution method which we will describe. If we modify some of the characteristics of the class of models, then we usually have to devise a new (though usually similar) solution method. For example, if we changed scheduling disciplines in the above characterization, then we would have to change the solution method. The recursive solutions are usually much more efficient in use of memory and computation than direct or iterative solutions. The price of this efficiency is a lack of flexibility.

These solution methods are termed "recursive" because of the form of the equations used. The actual algorithms are usually iterative rather than recursive.

As an example, let us consider the class of models described above (i.e., Figure 3.5) with the additional restriction that the service times at the CPU are also exponential. Assuming the CPU service times have rate $b_1$, the I/O service times have rate $b_2$, the number of I/O's is $L$, and the number of jobs is $N$, then we have the following state transition diagram (state $i$ indicates $i$ jobs at the CPU):

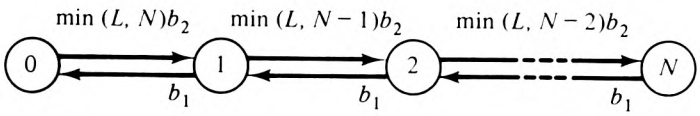



Figure 3.11

For $2 \leq n \leq N$ we can write the balance equation for state $n-1$ as

$$(b_1 + \min(L,N-(n-1))b_2)P(n-1) = b_1P(n) + \min(L,N-(n-2))b_2P(n-2).$$

This may be rewritten as

$$P(n) = ((b_1 + \min(L,N-(n-1))b_2)P(n-1) - \min(L,N-(n-2))b_2P(n-2))/b_1.$$

Thus $P(n)$ is recursively defined in terms of $P(n-1)$ and $P(n-2)$. Similarly we can use the balance equation for state 0 to obtain

$$P(1) = \min(L,N)b_2P(0))/b_1.$$

Thus we can compute the probabilities of all states in terms of $P(0)$. We can use the knowledge that the probabilities of all states sum to one to determine $P(0)$.

The following algorithm uses these recursive formulas to determine CPU utilization, throughput, mean queue length and mean queueing time. It assumes $P(0) = 1/G$ and determines $G$ to determine these performance measures. Note that after $P(n)$ is determined the storage for $P(n-2)$ may be reclaimed. Thus the memory required is very small, regardless of $N$.

**Algorithm 3.1**

$P(0) = 1$
$G = 1$
$P(1) = \min(L,N)b_2/b_1$
$Q = P(1)$
$G = G + P(1)$
For $n = 2$ to $N$
    $P(n) = ((b_1 + \min(L,N-(n-1))b_2P(n-1)$
        $- \min(L,N-(n-2))b_2\mathrm{P}(n-2))/b_1$
    $Q = Q + nP(n)$
    $G = G + P(n)$
utilization $= 1 - 1/G$
throughput $= b_1 \times$ utilization
mean queue length $= Q/G$
mean queueing time = mean queue length/throughput

Let us now assume that $L = 1$ and generalize to allow the branching Erlang distribution at the CPU. We can define the system states by the pair $(n,i)$ where $n$ is the number of jobs at the CPU and the job in service is at subserver $i$. Figure 3.12 gives the transition diagram for $N = 3$ and $k = 2$. (When $n = 0$ $i$ is not meaningful. We use the pair (0,1) for notational convenience in expressing the algorithms.) Let the CPU distribution have rates $a_1$ and $a_2$ and probabilities $V_{10}$ and $V_{12}$. Let the I/O times be exponential with rate $b_2$.

We present two algorithms for this case. The first is the preferred algorithm because of lower memory requirement and complexity. The second algorithm illustrates a technique which is useful in other circumstances, e.g., where both queues have the branching Erlang discipline. (A subsequent priority model also utilizes this technique.)

We can write the balance equation for (1,2) as

$$(a_2 + b_2)P(1,2) = V_{12}a_1P(1,1)$$

which yields

$$P(1,2) = V_{12}a_1P(1,1)/(a_2 + b_2). \tag{3.6}$$

We can write the equation for (0,1) as

$$b_2P(0,1) = V_{10}a_1P(1,1) + a_2P(1,2).$$

Substituting (3.6) yields

$$b_2P(0,1) = V_{10}a_1P(1,1) + V_{12}a_1a_2P(1,1)/(a_2 + b_2)$$

which yields

$$P(1,1) = b_2P(0,1)/(V_{10}a_1 + V_{12}a_1a_2/(a_2 + b_2)).$$

Similar arguments yield recursive expressions for $P(n,1)$ and $P(n,2)$, $2 \leq n \leq N - 1$, and for $P(N,1)$ and $P(N,2)$. The expressions are included in Algorithm 3.2 (Note that the algorithm omits calculations of the desired performance measures; the addition of these is straightforward.)

**Algorithm 3.2** (assumes $N \geq 2$)

$P(0,1) = 1$
$G = 1$
$P(1,1) = b_2P(0,1)/(V_{10}a_1 + V_{12}a_1a_2/(a_2 + b_2))$
$P(1,2) = V_{12}a_1P(1,1)/(a_2 + b_2)$
$G = G + P(1,1) + P(1,2)$
For $n = 2$ to $N-1$

$$P(n,1) = ((a_1 + b_2)P(n-1,1) - b_2P(n-2,1) - a_2b_2P(n-1,2)/(a_2 + b_2))/(V_{10}a_1 + V_{12}a_1a_2/(a_2 + b_2))$$

$$P(n,2) = (V_{12}a_1P(n-1) + b_2P(n-1,2))/(a_2 + b_2)$$

$$G = G + P(n,1) + P(n,2)$$
$$P(N,1) = ((a_1 + b_2)P(N-1,1) - b_2P(N-2,1) - b_2P(N-1,2))/a_1$$
$$P(N,2) = (V_{12}a_1P(N,1) + b_2P(N-1,2))/a_2$$
$$G = G + P(N,1) + P(N,2)$$
$$P(0,1) = 1/G$$

Suppose we wish the recursion (iteration) to proceed in ascending (descending) values of $n$. By the balance equation for $(N,1)$ we can obtain

$$P(N-1,1) = a_1P(N,1)/b_2,$$

and from the balance equation for $(N,2)$ we obtain

$$P(N-1,2) = (a_2/b_2)P(N,2) - (V_{12}a_1/b_2)P(N,1).$$

However, these equations depend on both $P(N,1)$ and $P(N,2)$ and there is no straightforward expression of one of these in terms of the other. We can proceed to define $P(N-2,1)$ and $P(N-2,2)$ in terms of $P(N,1)$ and $P(N,2)$ and so forth and eventually determine values for $P(N,1)$ and $P(N,2)$. Algorithm 3.3 does this by representing state probabilities as two element vectors. We use the notation $p(n,i)$ for such a vector. We will also use two element vectors $g$ and $d$. The elements of these vectors will be referred to as $g_1$, $g_2$, $d_1$ and $d_2$.

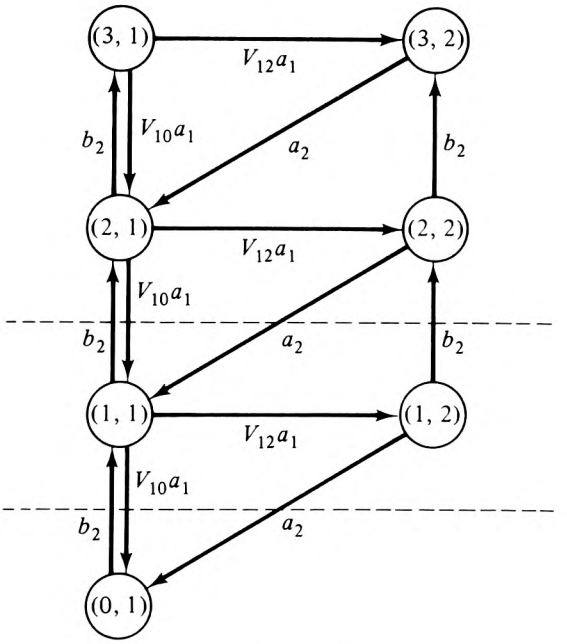



Figure 3.12

**Algorithm 3.3** (assumes $N \geq 2$)

$p(N,1) = (1,0)$

$p(N,2) = (0,1)$

$q = N(p(N,1) + p(N,2))$

$g = p(N,1) + p(N,2)$

$p(N-1,1) = (a_1/b_2)\, p(N,1)$

$p(N-1,2) = (a_2/b_2)\, p(N,2) - (V_{12}a_1/b_2)p(N,1)$

$q = q + (N-1)\,(p(N-1,1) + p(N-1,2))$

$g = g + p(N-1,1) + p(N-1,2)$

For $n = N-2$ down to 1

$p(n,1) = (1 + a_1/b_2)\, p(n+1,1) - (V_{10}a_1/b_2)p(n+2,1)$
$\quad - (a_2/b_2)p(n+2,2)$

$p(n,2) = (1 + a_2/b_2)\, p(n+1,2) - (V_{12}a_1/b_2)p(n+1,1)$

$q = q + n(p(n,1) + p(n,2))$

$g = g + p(n,1) + p(n,2)$

$p(0,1) = (1 + a_1/b_2)p(1,1) - (V_{10}a_1/b_2)p(2,1)$
$\quad - (a_2/b_2)p(2,2)$

$g = g + p(0,1)$

$d = p(1,2) - (V_{12}a_1/(a_2 + b_2))p(1,1)$

Solve the following equations for $P(N,1)$ and $P(N,2)$

$d_1P(N,1) + d_2P(N,2) = 0$

$g_1P(N,1) + g_2P(N,2) = 1$

mean CPU queue length $= q_1P(N,1) + q_2P(N,2)$

Note that the vector $d$ is the difference of the value of $p(1,2)$ determined from the balance equation for (2,2) and the value of $p(1,2)$ as determined by the balance equation for (1,2). Thus $d_1P(N,1) + d_2P(N,2) = 0$.

This algorithm illustrates how the recursive method may be used to greatly reduce the number of linear equations to be solved. Let us apply this method to do the following priority model: There are $N_1$ high priority jobs and $N_2$ low priority jobs. CPU times for high priority jobs are exponential with rate $b_{11}$ and CPU times for low priority jobs are exponential with rate $b_{12}$. High priority jobs have a dedicated I/O device with exponential rate $b_{21}$. Similarly, low priority jobs have a dedicated I/O device with exponential rate $b_{22}$. See Figure 3.13.

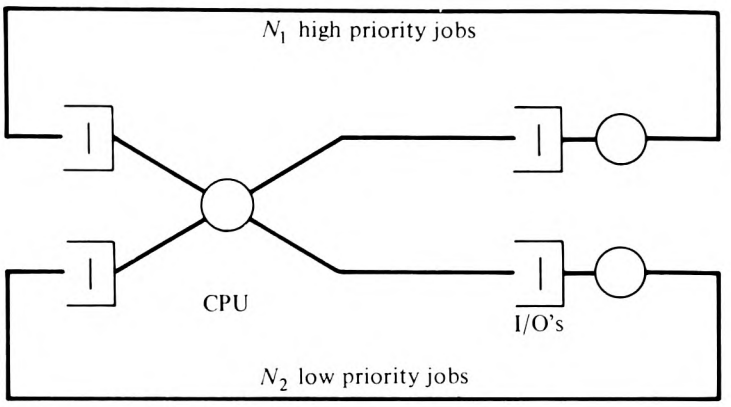



Figure 3.13

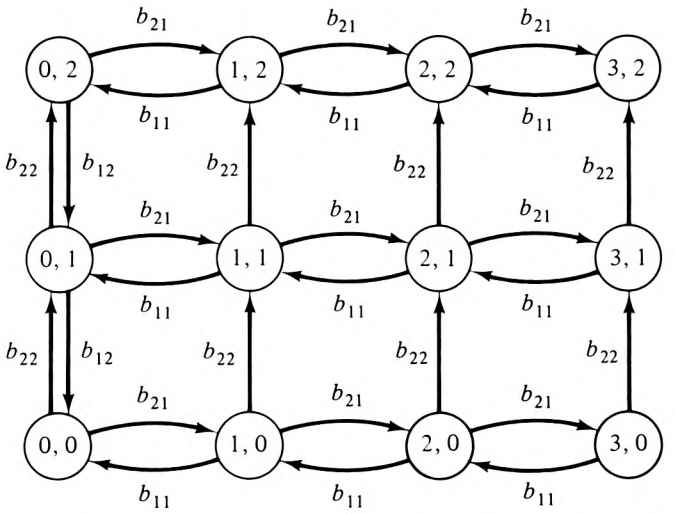



Figure 3.14

Because of the memoryless property of the exponential distribution, the service time of a low priority job is independent of preemptions. Thus we can represent a state of the system by the pair $(n_1,n_2)$ where $n_1$ is the number of high priority jobs at the CPU and $n_2$ is the number of low priority jobs at the CPU. Figure 3.14 gives the state transition diagram for $N_1 = 3$ and $N_2 = 2$.

Algorithm 3.4 uses the recursive method to solve for the probabilities of states $(N_1,0),(N,1),\ldots,(N_1,N_2)$. The notation $e_i$ is used for the vector

with the $i^{th}$ element equal to 1 and all other elements equal to 0. All vectors have length $N_2 + 1$.

**Algorithm 3.4** (assumes $N_1 > 0, N_2 > 0$)

For $n_2 = 0$ to $N_2$

$p(N_1,n_2) = e_{n_2+1}$

$p(N_1-1,0) = ((b_{11} + b_{22})/b_{21})p(N_1,0)$

For $n_2 = 1$ to $N_2-1$

$p(N_1-1,n_2) = ((b_{11} + b_{22})/b_{21})p(N_1,n_2) - (b_{22}/b_{21})p(N_1,n_2-1)$

$p(N_1-1,N_2) = (b_{11}/b_{21})p(N_1,N_2) - (b_{22}/b_{21})p(N_1,N_2-1)$

$g = (0,0,...,0)$

For $n_1 = N_1-1$ to $N_1$

For $n_2 = 0$ to $N_2$

$g = g + p(n_1,n_2)$

For $n_1 = N_1-2$ down to 0

$p(n_1,0) = ((b_{11}+b_{22})/b_{21})p(n_1+1,0) - (b_{11}/b_{21})p(n_1+2,0)$

For $n_2 = 1$ to $N_2-1$

$$p(n_1,n_2) = ((b_{11}+b_{22})/b_{21})p(n_1+1,n_2) - (b_{22}/b_{21})p(n_1+1,n_2-1) - (b_{11}/b_{21})p(n_1+2,n_2$$

$$p(n_1,N_2) = (b_{11}/b_{21})p(n_1+1,N_2) -(b_{22}/b_{21})p(n_1+1,N_2-1) -(b_{11}/b_{21})p(n_1+2,N_2)$$

For $n_2 = 0$ to $N_2$

$g = g + p(n_1,n_2)$

For $n_2 = 1$ to $N_2-1$

$$d_{n_2} = p(0,n_2) - (b_{22}/(b_{12} + b_{21} + b_{22}))p(0,n_2-1) - (b_{11}/(b_{12} + b_{21} + b_{22}))p(1,n_2) - (b_{12}/(b_{12} + b_{12} + b_{22}))p(0,n_2+1)$$

$$d_{N_2} = p(0,N_2) - (b_{22}/(b_{12} + b_{21}))p(0,N_2-1) - (b_{11}/(b_{12} + b_{21}))p(1,N_2)$$

Solve

$$\begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ \cdot \\ d_{N_2} \\ g \end{bmatrix} \begin{bmatrix} P(N_1,0) \\ P(N_1,1) \\ \cdot \\ \cdot \\ \cdot \\ P(N_1,N_2-1) \\ P(N_1,N_2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \\ 1 \end{bmatrix}$$

We have used the notation $d_i$ for the $i^{th}$ row of a matrix $D$ with $N_2$ rows and $N_2+1$ columns.

## 3.6 FURTHER READING

A more thorough introduction to Markov processes can be found in DRAK67 and FELL68.

A discussion of a variety of iterative solution methods is found in STEW78.

The complete generality of the method of stages is found in COX55. Additional description of our more restricted form is found in SAUE75a. Further discussion of matching distributions is found in BUX77, LAZO77 and SEVC77b.

Further examples of application of the recursive solutions are found in HERZ75, SAUE75a and SAUE77b. An alternate recursive solution approach, which avoids potential numerical problems of the approach we have discussed, is given in MARI80.

## 3.7 EXERCISES

3.1 Generalize the program of Figure 3.7 to allow the branching Erlang distribution at the CPU.

3.2 Generalize the program of Figure 3.7 to allow the branching Erlang distribution at the I/O devices.

3.3 Generalize the program of Figure 3.7 to allow $M$ queues with probabilities $p_{ij}$ that a job leaving queue $i$ goes to queue $j$.

3.4 Generalize Algorithm 3.2 to allow two CPU's.

3.5 Combine Algorithms 3.2 and 3.3 to allow the branching Erlang distribution at the I/O device as well as the CPU.

3.6 Construct an algorithm for the model of Figure 3.13 without preemption.

3.7 Construct an algorithm for a model similar to that of Figure 3.13 except that the two disjoint groups of jobs have equal priority and CPU scheduling is FCFS.

## 3.8 SUMMARY OF CHAPTER NOTATION

| | |
|---|---|
| $a$ | Rate of an exponential distribution |
| $S_i$ | State of a Markov process |
| $q_{ij}$ | Probability of transition from state $i$ to state $j$ given a transition out of state $i$. |
| $P_i$ | Equilibrium probability of state $i$ of a Markov process |
| $G$ | Normalizing constant |
| $p$ | A column vector consisting of $P_1, P_2, \ldots, P_N$, where $N$ is the number of states of the Markov process (Section 3.3) |
| $\hat{p}^k$ | $k^{th}$ estimate of $p$ |
| $V_{ij}$ | Probability of visiting subserver (stage) $j$ after visiting subserver $i$ in a distribution consisting of exponential stages |

# *CHAPTER 4*

# ISOLATED QUEUES AND OPEN NETWORKS OF QUEUES

Though the numerical methods of the last chapter are very general, in the sense that they apply conceptually to very complex system models, these methods by themselves are of limited practical application. They are limited because they require solution of a set of linear equations and because that set may be enormous. For the queueing models we consider in this chapter, the set of equations is infinite. Thus we cannot hope to numerically solve the linear equations nor can we cope with performance measures which require numerical values for all of the state probabilities. We must first seek algebraic simplification of the problem.

Fortunately, a convenient algebraic simplification, the *product form solution*, applies to a very large group of queueing network models. We defer formal definition of the product form solution, but informally it is expressed as follows

$$P(S_1,S_2,...,S_M) = \frac{P_1(S_1)P_2(S_2)...P_M(S_M)}{G} \tag{4.1}$$

where $P(S_1,S_2,...,S_M)$ is the probability of a network state in a network of M queues (for example, $S_m$ might be the queue length of queue $m$), $P_m(S_m)$ is a factor reflecting the probability that queue $m$ is in state $S_m$ and $G$ is a normalizing constant. ($G$ explicitly forces the probabilities of the network states to 1.) We have already seen this form in (3.2 - 3.5). The formal definitions we give will be associated with specific groups of networks.

Most software packages for numerical solution of queueing networks depend on product form solutions [SAUE78a]. Most of this chapter and all of Chapter 5 will be devoted to queueing networks with product form solution. When a network does not have a product form solution and has too many states for numerical solution for the state probabilities, then our alternatives are approximation, to be covered in Chapter 6, and simulation, to be covered in Chapter 7. Even approximation techniques are dependent on results for product form networks.

All of the models of this chapter will assume that there is an infinite *source* of jobs arriving at the network (or queue in Section 4.1) with exponential interarrival times with mean $1/R$. (In other words, arriving jobs

form a Poisson process with an average arrival rate of $R$ jobs per unit time.) There is also a *sink* which jobs enter when they leave the network. The assumption of a potentially infinite population of jobs in the network is usually not reasonable in computer system models. This is because there is usually some resource which limits the total population of jobs in the network. For example, in the simple batch system model of Chiu *et al*, the number of jobs at the CPU and I/O queues is kept small by contention for memory. In a timesharing system model such as the ones of Brown *et al* and Bard, the number of jobs is limited by the number of terminals (or terminal ports). However, the infinite job population assumption is not unreasonable in communication system models where the number of jobs may be very large. Also, the infinite population assumption is very important in the history of queueing models and results in a simpler solution for the networks of this chapter in comparison with those of the next chapter.

## 4.1 ISOLATED QUEUES

Throughout this section we will be principally interested in systems as depicted in Figure 4.1. We will look at such systems with differing service time distributions and scheduling algorithms.

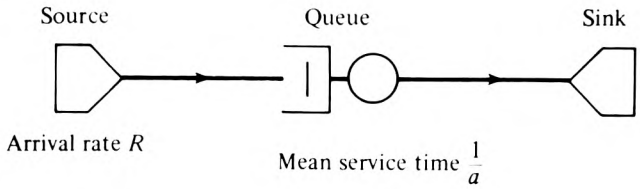



Figure 4.1

### 4.1.1 Exponential Service Times

Consider the system of Figure 4.1 with FCFS scheduling and a single server. In classical queueing notation, this system is the M/M/1 queue, where the first symbol indicates the interarrival time distribution, the second symbol indicates the service time distribution and the third symbol indicates the number of servers. M, for "Markov," indicates exponential distributions. We can define the Markov states of this system according to the number of jobs at the queue. See Figure 4.2.

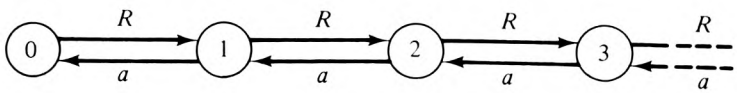



Figure 4.2

In state 0, no service is in progress, so the only transition corresponds to a job leaving the source. In all other states there is a transition for a job leaving the server as well as the source. The balance equation for state 0 is

$$RP(0) = aP(1),$$

so

$$P(1) = \frac{R}{a}P(0).$$

For the system to be stable the arrival rate $R$ must be less than the service rate a; otherwise the queue will become infinitely long. As we will show below the utilization $U = R/a$. (This can also be immediately observed from (2.7) and holds for single server queues of the form of Figure 4.1 regardless of distributions.) The equation for state $n$, $n = 1, 2, 3, \ldots$, is

$$(R + a)P(n) = RP(n - 1) + aP(n + 1).$$

However, this equation can be simplified to

$$RP(n) = aP(n + 1) \tag{4.2}$$

as follows. For state 1,

$$(R + a)P(1) = RP(0) + aP(2)$$

implies

$$RP(1) + a\frac{R}{a}P(0) = RP(0) + aP(2)$$

implies

$$RP(1) = aP(2)$$

and

$$P(2) = \frac{R}{a}P(1).$$

This derivation can be repeated for states 2, 3, ... to obtain (4.2). From (4.2),

$$P(n) = \frac{R}{a}P(n - 1), \quad n = 1, 2, 3, \ldots$$

and

$$P(n) = \left(\frac{R}{a}\right)^n P(0), \; n = 0, 1, 2, \ldots \tag{4.3}$$

We also know that

$$\sum_{n=0}^{\infty} P(n) = 1,$$

so

$$\sum_{n=0}^{\infty} \left(\frac{R}{a}\right)^n P(0) = 1.$$

Applying (2.3) we have

$$\frac{1}{1 - \frac{R}{a}} P(0) = 1$$

and

$$P(0) = 1 - \frac{R}{a}. \tag{4.4}$$

Since the server is idle in state 0 and busy in all other states,

$$U = 1 - P(0) = 1 - \left(1 - \frac{R}{a}\right) = \frac{R}{a}. \tag{4.5}$$

Substituting (4.4) and (4.5) in (4.3)

$$P(n) = U^n(1 - U), \; n = 0, 1, 2, \ldots \tag{4.6}$$

Note that $P(n)$ has the geometric distribution (starting at 0 instead of 1) with parameter $1-U$. Thus indirectly from the geometric distribution starting at 1 or by repeatedly applying (2.3),

$$\begin{aligned} L &= \sum_{n=1}^{\infty} nP(n) \\ &= \sum_{n=1}^{\infty} nU^n(1 - U) \\ &= \frac{U}{1 - U} \end{aligned} \tag{4.7}$$

Note that (4.7) is consistent with intuition; as $U = R/a$ goes to zero, the mean queue length goes to zero, and as $U$ goes to one, the mean queue length goes to infinity. We can rewrite (4.7) as

$$L = \frac{U^2}{1 - U} + U \tag{4.8}$$

where the first term is the mean number of jobs waiting for service and $U$ is interpreted as the mean number of jobs in service.

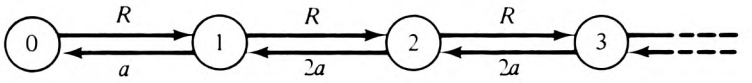



Figure 4.3

Since the queue is not saturated, i.e., $U < 1$, the throughput must be R. By Little's Rule (2.9), $L = \lambda Q = RQ$, $Q = L/R$ and the mean queueing (response) time

$$Q = \frac{\frac{U}{1 - U}}{R} = \frac{\frac{R}{a}}{\frac{1 - U}{R}} = \frac{\frac{1}{a}}{1 - U} \tag{4.9}$$

$$= \frac{1}{a - R}. \tag{4.10}$$

The form corresponding to (4.8) is

$$Q = \frac{U\frac{1}{a}}{1 - U} + \frac{1}{a} \tag{4.11}$$

where the first term is the mean waiting time and the second term is the mean service time.

Let us now consider the M/M/2, system, i.e., the same system but with two servers. The states are identified as before; Figure 4.3 gives the transitions. We now must have $R < 2a$ for stability; by (2.8) $U = R/2a$. By arguments similar to the ones we just used,

$$P(1) = \frac{R}{a}P(0) \tag{4.11}$$

and

$$P(n) = \left(\frac{R}{2a}\right)^{n-1}\frac{R}{a}P(0),\ n = 2, 3, \ldots. \tag{4.12}$$

Then

$$P(0) + \sum_{n=1}^{\infty} \left(\frac{R}{2a}\right)^{n-1} \frac{R}{a} P(0) = 1,$$

$$P(0) + \frac{\frac{R}{a}P(0)}{1 - \frac{R}{2a}} = 1,$$

and

$$P(0) = \frac{2a - R}{2a + R} \tag{4.13}$$

In state 0 both servers are idle. In state 1, one server is idle, so

$$U = 1 - P(0) - \frac{1}{2}P(1) = \frac{R}{2a}.$$

As before, by repeated application of (2.3)

$$\begin{aligned} L &= \sum_{n=1}^{\infty} nP(n) \\ &= \sum_{n=1}^{\infty} n\left(\frac{R}{2a}\right)^{n-1} \frac{R}{a} P(0) \\ &= \frac{4aR}{4a^2 - R^2}. \end{aligned} \tag{4.14}$$

By Little's Rule (2.9)

$$Q = \frac{4a}{4a^2 - R^2} \tag{4.15}$$

We can proceed similarly for 3,4,5,... servers; the algebra is more tedious, but there are no real problems. Of special interest is the limiting case where there is an infinite supply of servers, i.e., the M/M/∞ "queue." Of course, there is never any waiting for a server and thus scheduling is irrelevant. Since there is never any waiting,

$$Q = \frac{1}{a},$$

and by Little's Rule

$$L = \frac{R}{a}.$$

This is also the mean number of busy servers. (Note that, by our definition, $U = 0$.) Note that these results apply to any service time distribution with mean $1/a$. With general (arbitrary) service distribution, this is known as the M/G/$\infty$ queue.

Now consider a single server queue with PS scheduling. The Markov states may again be defined by the number of jobs in the queue. With 1 job in the queue, that job's rate of completion is $a$. With two jobs in the queue each job is getting one half of the effective rate of the server. Thus each job completes at rate $a/2$ and their combined completion rate is $a$. Similarly for the other states with non-zero queue lengths, the total completion rate is $a$. Thus the state diagram of Figure 4.2 is still valid and so the PS results are the same as the FCFS results. Since both FCFS and PS give the same results, any RR discipline with zero overhead will give these results also. Consideration of SRTF is beyond the scope of this book.

### 4.1.2 General Service Times

Let us consider a single server, FCFS scheduling and general service time distribution, i.e., we consider the M/G/1 queue. If we represent the distribution by the method of stages, e.g., the branching Erlang distribution (Figure 3.10), then we can describe the states by the current queue length (counting the job in service, if any) and the current distribution stage of the job in service. Thus our state diagram would be essentially the same as Figure 3.12 but with additional states for queue lengths 4,5,6,... Solution of such a Markov process is quite difficult and requires methods beyond the scope of this book.

However, we can obtain $U$, $L$ and $Q$ rather easily. As before,

$$U = RE[x]$$

where $E[x]$ is the mean service time. We obtain the mean queue length and queueing time by first obtaining the mean waiting time (excluding the service time) which we call $W$. A randomly arriving job will have

$$W = (L - U)E[x] + UE[x'] \tag{4.16}$$

where $E[x']$ is the mean *remaining* service time of a job in service at the arrival. The arriving job finds a job in service with probability $U$; thus it expects to wait $UE[x']$ time units for a job in service to complete. The

arriving job expects to find $L - U$ waiting jobs and also must wait for their service times. By Little's Rule we also know that

$$L - U = RW. \tag{4.17}$$

Substituting (4.17) into (4.16) and solving for $W$ we obtain

$$W = \frac{U}{1 - U} E[x']. \tag{4.18}$$

Thus we only need to obtain $E[x']$ to get $W$ and thus $Q$ and $L$.

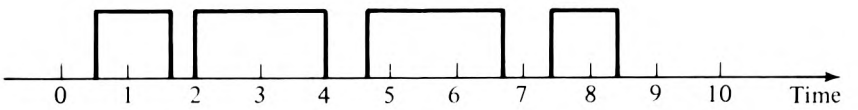



Figure 4.4

Suppose the service times have a discrete distribution $P(x)$. Though $P(x)$ is the probability a job has a service time $x$, an arriving job finding a job in service does not necessarily find the service time of the job in progress to be $x$ with probability $P(x)$. This is because the arriving job is more likely to arrive during a long service time than a short one. Consider the distribution

$$P(x) = \begin{cases} .5, & x = 1 \text{ or } x = 2, \\ 0 & \text{otherwise.} \end{cases}$$

and Figure 4.4. The rectangles represent service times. Even though the service times are equally likely, it is twice as likely that a job arriving during a service time arrives during a service time of 2. In general, the probability a job arrives during service time $x$ is proportional to $xP(x)$ and thus equal to

$$\frac{xP(x)}{\sum_x xP(x)} = \frac{xP(x)}{E[x]}. \tag{4.19}$$

Given that a job arrives during a service time $x$, it is equally likely that the job arrives at any time during that service time and thus the expected remaining time is $x/2$. So summing over all possible service times,

$$E[x'] = \frac{\sum_x \frac{x}{2} xP(x)}{E[x]} = \frac{E[x^2]}{2E[x]}. \tag{4.20}$$

A similar argument for continuous distributions gives

$$E[x'] = \frac{\int_0^\infty \frac{x}{2} x f_x(x_0) dx_0}{E[x]} = \frac{E[x^2]}{2E[x]}. \tag{4.21}$$

Substituting into (4.18) we get

$$W = \frac{UE[x']}{1 - U} = \frac{RE[x^2]}{2(1 - U)}. \tag{4.22}$$

Since $E[x^2] = (E[x])^2(1 + C_x^2)$,

$$W = \frac{UE[x](1 + C_x^2)}{2(1 - U)}. \tag{4.23}$$

Then

$$Q = W + E[x] = \frac{UE[x](1 + C_x^2)}{2(1 - U)} + E[x] \tag{4.24}$$

and by Little's Rule

$$L = \frac{U^2(1 + C_x^2)}{2(1 - U)} + U.$$

These last four equations are all variations on what is known as the *Pollaczek-Khintchine Formula*. Our information derivation of this formula is based on the derivation given in WOLF70. Alternate derivations and derivations of other M/G/1 characteristics such as the queue length distribution can be found in KLEI76 and KOBA78.

Note that (4.25) reduces to (4.7) for exponential service times, i.e., for $C_x = 1$. Figure 4.5 shows $L$ versus $U$ for $C_x = 0$, 1, 2 and 5. Notice the sharp rise in $L$ with $U$, regardless of $C_x$, and the dramatic differences between the curves at the larger utilizations.

Now consider a single server queue with PS scheduling and the branching Erlang distribution with 2 stages (Figure 3.10). Let the branching Erlang parameters be $a_1$, $a_2$, $V_{10}$ and $V_{12}$, as before. (Recall that $V_{10} + V_{12} = 1$.) Every job in the queue is in service, so the distribution

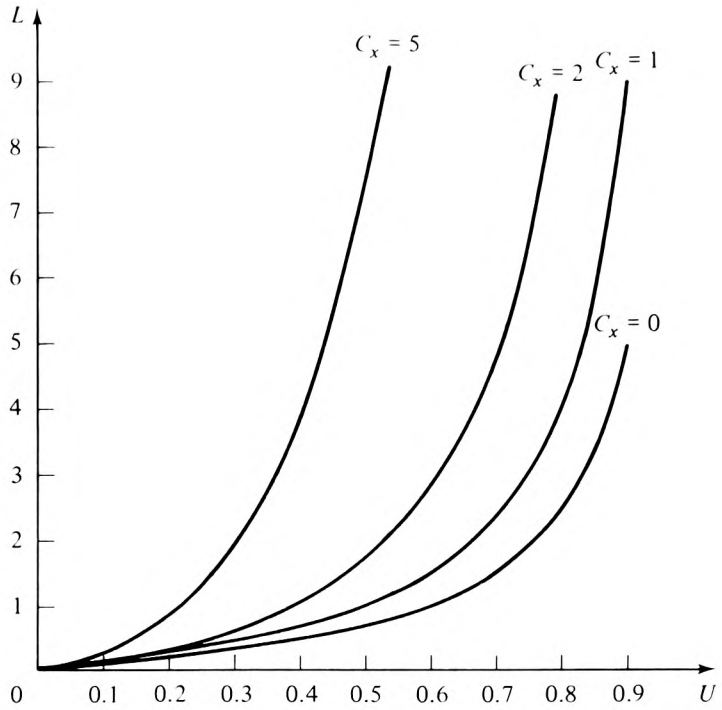



Figure 4.5

stage of each job must be included in the state description. Let the states be defined by the pair $(n_1,n_2)$ where there are $n_1$ jobs in the first stage and $n_2$ jobs in the second stage. See Figure 4.6.

In state $(n_1,n_2)$ each job gets $1/(n_1 + n_2)$ of the server. The $n_1$ jobs in stage 1 each have a completion rate of $a_1/(n_1 + n_2)$ thus the transition rate to $(n_1 - 1,n_2)$ is $n_1V_{10}a_1/(n_1 + n_2)$ and the transition rate to $(n_1 - 1,n_2 + 1)$ is $n_1V_{12}a_1/(n_1 + n_2)$. Similarly, the transition rate to $(n_1,n_2 - 1)$ is $n_2a_2/(n_1 + n_2)$.

Some of the balance equations are, for states (0,0), (1,0) and (0,1),

$$RP(0,0) = V_{10}a_1P(1,0) + a_2P(0,1) \tag{4.26}$$

$$(R + a_1)P(1,0) = RP(0,0) + V_{10}a_1P(2,0) + \frac{a_2}{2}P(1,1) \tag{4.27}$$

$$(R + a_2)P(0,1) = V_{12}a_1P(1,0) + \frac{V_{10}a_2}{2}P(1,1) + a_2P(0,2) \tag{4.28}$$

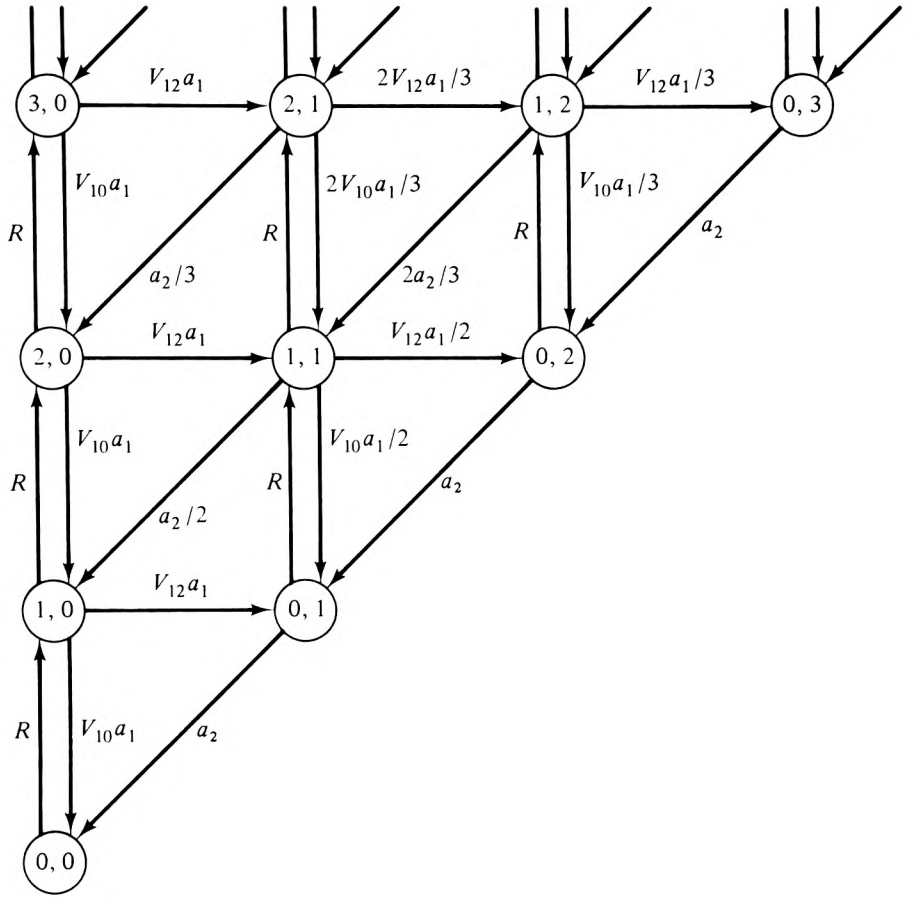



Figure 4.6

There is no obvious solution to the collection of equations, but a fairly simple solution exists. How do we find it? Suppose that

$$a_1 P(1,0) = RP(0,0) \tag{4.29}$$

and

$$RP(1,0) = V_{10}a_1 P(2,0) + \frac{a_2}{2} P(1,1). \tag{4.30}$$

A solution of both (4.29) and (4.30) must also satisfy (4.27). (A solution of an equation, e.g., (4.27), need not necessarily satisfy equations obtained by partitioning that equation, e.g., (4.29) and (4.30).)

If (4.29) holds, then

$$P(1,0) = \frac{R}{a_1}P(0,0) \tag{4.31}$$

and by (4.26)

$$P(0,1) = \frac{R(1 - V_{10})}{a_2}P(0,0) = \frac{RV_{12}}{a_2}P(0,0). \tag{4.32}$$

In, fact (4.29) holds, as will be apparent when we give a general expression for $P(n_1,n_2)$. The balance equations which consider all transitions, e.g., (4.27), are called *global balance* equations. Equations (4.29) and (4.32) are termed *local balance* (or *independent balance* or *separable balance*) equations. A subset of local balance equations or equivalently, the global balance equations, are used to verify that the proposed solution is correct.

How do we obtain the local balance equations? For example, why did we not suggest that

$$RP(1,0) = RP(0,0)$$

from equation (4.27)? The rule is to *equate flow into a state due to flow into a distribution stage to flow out of that state due to flow out of that distribution stage*. For equation (4.29), $RP(0,0)$ is the flow into (1,0) due to a job entering the first service stage and $a_1P(1,0)$ is the flow out of (1,0) due to a job leaving the first service stage. (Note that (4.2) for the M/M/1 queue is analogously obtainable.) Here the job "entering the distribution stage" begins service, but this is not always necessary, e.g., for FCFS queues with exponential service local balance applies, but jobs entering a distribution stage do not necessarily begin service until later because there is no distinction between jobs in service and jobs waiting, as far as remaining service time is concerned. We may also apply the rule to sources if we consider jobs going to a sink to join the infinite population at the source. For (4.30), $V_{10}a_1P(2,0) + (a_2/2)P(1,1)$ may be viewed as the flow into (1,0) because of a job entering a *source* distribution stage (though the interarrival time for the job does not begin until later) and $RP(1,0)$ the flow out of (1,0) due to a job leaving a source distribution stage. Similarly, for equation (4.28) we can obtain

$$RP(0,1) = \frac{V_{10}a_1}{2}P(1,1) + a_2P(0,2) \tag{4.33}$$

and

$$a_2P(0,1) = V_{12}a_1P(1,0). \tag{4.34}$$

Equation (4.34) provides no new information. From state (2,0) we can get

$$a_1 P(2,0) = RP(1,0) \tag{4.35}$$

so

$$P(2,0) = \frac{R}{a} P(1,0) = \left(\frac{R}{a}\right)^2 P(0,0). \tag{4.36}$$

Similarly, from state (1,1)

$$RP(0,1) = \frac{a_1}{2} P(1,1) \tag{4.37}$$

and

$$P(1,1) = \frac{2R}{a_1} P(0,1) = 2\frac{R}{a_1}\frac{RV_{12}}{a_2} P(0,0). \tag{4.38}$$

Now using (4.33) we have

$$P(0,2) = \frac{RV_{12}}{a_2} P(0,1) = \left(\frac{RV_{12}}{a_2}\right)^2 P(0,0). \tag{4.39}$$

Proceeding in this manner we find that

$$P(n_1,n_2) = \frac{(n_1+n_2)!}{n_1!n_2!}\left(\frac{R}{a_1}\right)^{n_1}\left(\frac{RV_{12}}{a_2}\right)^{n_2} P(0,0). \tag{4.40}$$

Summing over all states such that $n_1 + n_2 = n$, we find

$$P(n) = \left(\frac{R}{a_1} + \frac{RV_{12}}{a_2}\right)^n P(0,0). \tag{4.41}$$

But notice that the mean service time

$$E[x] = \frac{1}{a_1} + \frac{V_{12}}{a_2}$$

and

$$\frac{R}{a_1} + \frac{RV_{12}}{a_2} = RE[x] = U.$$

Thus

$$P(n) = U^n P(0,0).$$

(Of course, $P(0,0) = 1-U$.)

So the PS queue with this arbitrary two stage branching Erlang distribution has the same queue length distribution as the PS queue with exponential service times and the same mean service! Since PS and FCFS give the same queue length distribution with exponential service times, this PS queue has the same queue length distribution as the M/M/1 queue. This derivation extends directly to any distribution represented by the method of stages. Thus for essentially arbitrary service distributions: *The PS queue lengths and mean queueing times depend only on the mean service time and the arrival rate; other characteristics of the service distribution are irrelevant. These PS performance measures are the same as those of a FCFS queue with exponential service time with the same mean service time and arrival rate.* This result also applies to multiple server PS queues. It can be derived by other means for service time distributions which cannot be exactly represented by the method of stages [CHAN77b].

With this result in mind, refer back to Figure 4.5. The curve for $C_x = 1$ also applies to PS with any distribution. Notice the enormous improvement of PS over FCFS with $C_x > 1$. With $C_x < 1$, PS is worse than FCFS, but the difference is small. (With $C_x = 0$, FCFS and SRTF are identical, so FCFS is optimal.)

### 4.1.3 Job Classes

So far we have been assuming that all jobs are homogeneous, that they have the same behavior and characteristics. The usual way to eliminate this assumption is to partition the jobs into *classes*. Within a class all jobs are homogeneous, but different job classes may have different service time distributions, priorities, routing, etc. (Some restrictions on class distinctions are necessary for a product form solution to exist.) In networks with finite population we can have enough classes so that there is only one job per class; thus if we go to the effort we can consider jobs individually.

With FCFS we must consider orderings of the jobs when they have different characteristics. With PS we can ignore orderings and then the derivations are much simpler. Consider Figure 4.7. Here an arriving job joins class $c$ with probability $p_{0c}$, $c = 1, 2, \ldots, C$. (An alternative representation would be to have a separate source for each class. By the characteristics of merging and splitting Poisson event streams discussed in Section 3.2.2., the two representations are equivalent as long as we have Poisson sources. The representation we use is simpler notationally. Notation is,

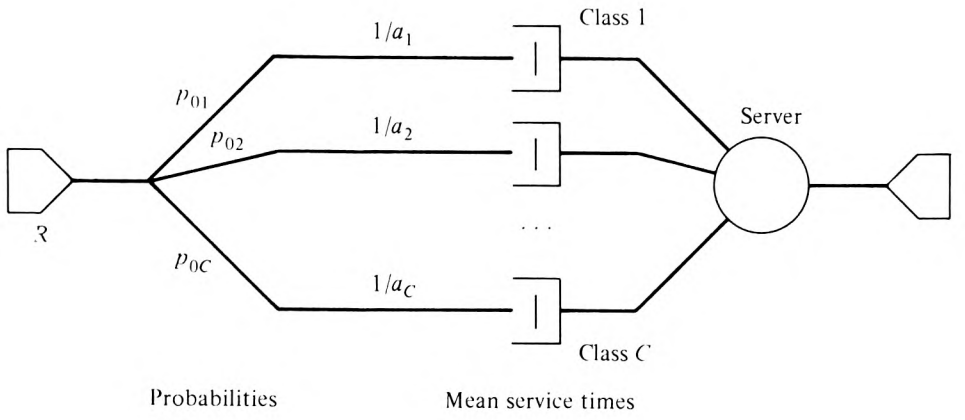



Figure 4.7

perhaps, the most difficult aspect of job classes in networks with product form solution.) We assume PS and, for the moment, exponential service distributions. With $C = 2$ we have the state diagram of Figure 4.8. Here state $(n_1,n_2)$ indicates that there are $n_1$ class 1 jobs and $n_2$ class 2 jobs at the queue.

Note that this state diagram is exactly the same as the one for PS with a two stage hyperexponential distribution (see Section 3.4) if we consider $p_{01}$ and $p_{02}$ to be $V_{01}$ and $V_{02}$, respectively, and if we consider $1/a_1$ and $1/a_2$ to be the means of the exponential stages 1 and 2! Thus, by the local balance arguments of the last section,

$$P(0,0) = 1 - R\left(\frac{p_{01}}{a_1} + \frac{p_{02}}{a_2}\right)$$

and

$$P(n_1,n_2) = \frac{(n_1 + n_2)!}{n_1!n_2!}\left(\frac{Rp_{01}}{a_1}\right)^{n_1}\left(\frac{Rp_{02}}{a_2}\right)^{n_2}P(0,0). \qquad (4.42)$$

In other words, with two different system characterizations, we discover that the underlying Markov process is the same.

In general, with $C$ classes, $P(n_1,...,n_C)$ is given by

$$\frac{(n_1 + ... + n_C)!}{n_1!...n_C!}\left(\frac{Rp_{01}}{a_1}\right)^{n_1}...\left(\frac{Rp_{0C}}{a_C}\right)^{n_C}P(0,...,0) \qquad (4.43)$$

where

$$P(0,...,0) = 1 - R\left(\frac{p_{01}}{a_1} + ... + \frac{p_{0C}}{a_C}\right). \tag{4.44}$$

Further, the overall queue length distribution is

$$P(n) = R^n\left(\frac{p_{01}}{a_1} + ... + \frac{p_{0C}}{a_C}\right)^n P(0,.,0), \tag{4.45}$$

by simple summations of (4.43). Since the parenthesized expression is the mean service time of all jobs, this is the same as (4.41).

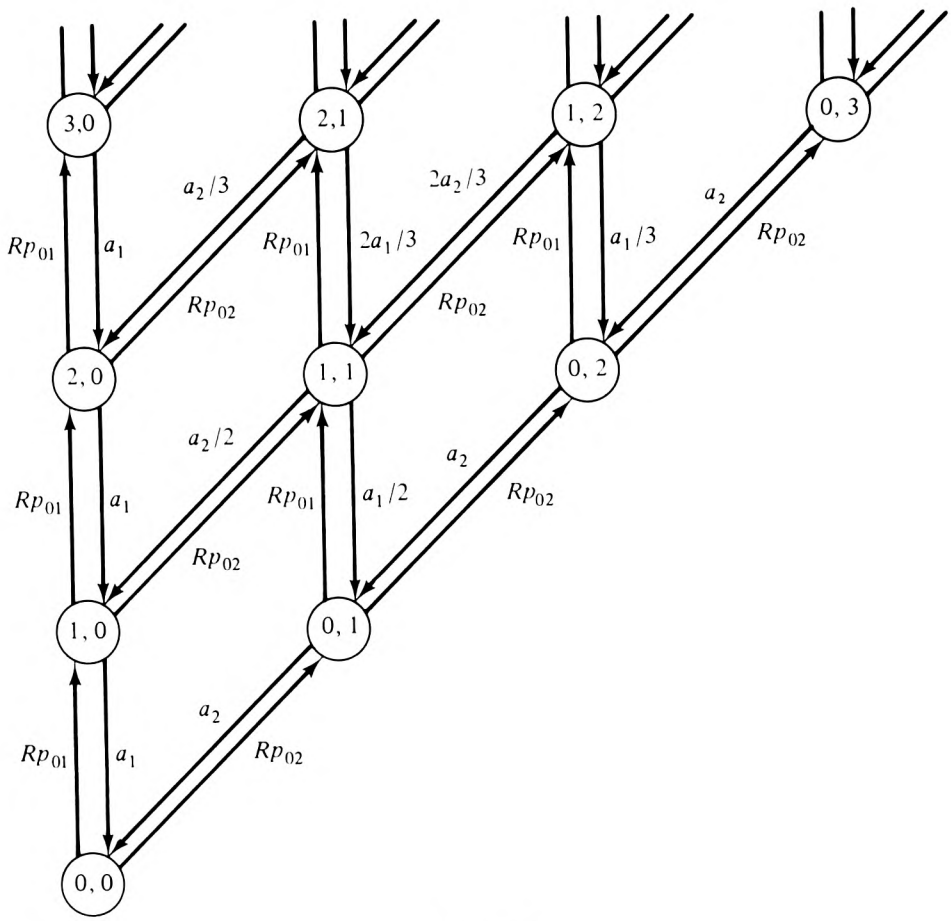



Figure 4.8

We can extend these results to general service time distributions as before; the only new problem is notation. If the mean service at class $c$ is $1/a_c$, $c = 1, 2, ..., C$, then (4.43 - 4.45) are valid for general service times.

By arguments and algebra similar to before we can show that

$$U_c = \frac{Rp_{0c}}{a_c}, \tag{4.46}$$

$$L_c = \frac{U_c U}{1 - U} + U_c \tag{4.47}$$

and

$$Q_c = \frac{U_c\left(\frac{p_{01}}{a_1} + \ldots + \frac{p_{0C}}{a_C}\right)}{1 - U} + \frac{1}{a_c} \tag{4.48}$$

Where $U_c$ , $c = 1, 2, \ldots, C$, is the utilization of the server by class $c$ jobs, $L_c$ is the mean queue length of class $c$ jobs and $Q_c$ is the mean queueing time for class $c$ jobs.

There are two problems with FCFS: the ordering problem already mentioned and the requirement that all classes have the *same, exponential* service time distribution for the product form solution to be valid. Assuming $a_1 = \ldots = a_C$ and exponential distributions, then the ordering problem becomes principally a problem of notation. For a given number of jobs of each class at the queue $(n_1,\ldots,n_C)$ each possible ordering has the same probability and (4.43 - 4.48) are valid.

It is of some consolation in computer system models that I/O devices (which often have FCFS scheduling) often have the same service time distributions for all jobs and that the actual distributions are usually only slightly less variable than the exponential distribution. In networks with general FCFS queues we must use approximations (Chapter 6) or simulation (Chapter 7).

If we eliminate the exponential distribution requirement or the requirement that all classes have the same distributions, then the FCFS queue will not have a simple solution for the queue length distribution nor can we incorporate such queues into product form networks. However, the Pollaczek-Khintchine Formula is easily extended. If $E[x_c]$ and $E[x_c^2]$ are the mean and second moment, respectively, for class $c$ service times, then

$$E[x] = \sum_{c=1}^{C} p_{0c} E[x_c],$$

$$E[x^2] = \sum_{c=1}^{C} p_{0c} E[x_c^2],$$

$$C_x^2 = \frac{E[x^2] - (E[x])^2}{(E[x])^2},$$

$$U_c = R p_{0c} E[x_c],$$

$$U = RE[x],$$

$$W = \frac{UE[x](1 + C_x^2)}{2(1 - U)}, \tag{4.49}$$

$$Q_c = \frac{UE[x](1 + C_x^2)}{2(1 - U)} + E[x_c] \tag{4.50}$$

and

$$L_c = \frac{U^2 p_{0c}(1 + C_x^2)}{2(1 - U)} + U_c . \tag{4.51}$$

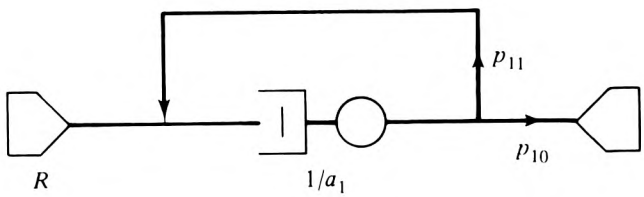



Figure 4.9

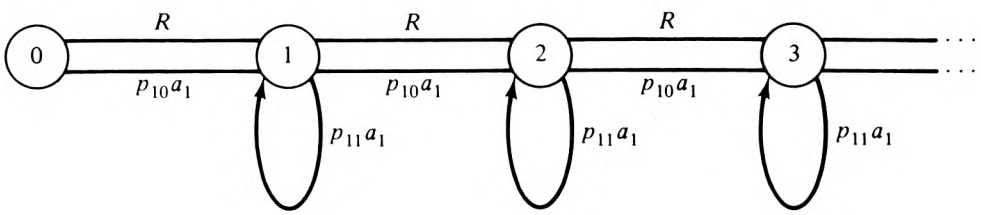



Figure 4.10

### 4.1.4 Multiple Visits (Loops)

Usually a job in a computer system model will visit a queue several times with intervening visits to other queues. For example, in the batch system model of Chiu et al, a job's processing includes many alternating CPU-I/O cycles. Before proceeding to networks of queues, let us consider isolated queues with multiple visits to the queue per job. Consider Figure 4.9. Assume FCFS and exponential service with mean $1/a_1$. A job departing the queue (from class 1) rejoins the queue (in class 1) with probability $p_{11}$. It goes to the sink with probability $p_{10}$. Figure 4.10 gives the Markov state diagram. Since we can immediately drop the $p_{11}a_1$ transitions from our balance equations, we obtain

$$P(n) = \left(\frac{R}{p_{10}a_1}\right)^n P(0), \tag{4.52}$$

$$P(0) = 1 - \frac{R}{p_{10}a_1}, \tag{4.53}$$

$$U = \frac{R}{p_{10}a_1}, \tag{4.54}$$

and

$$L = \frac{U^2}{1-U} + U. \tag{4.55}$$

Notice that these are precisely the same expressions as for an M/M/1 queue with arrival rate $R/p_{10}$. The mean queueing time is

$$Q = \frac{UR}{1-Ua_1} + \frac{1}{a_1}, \tag{4.56}$$

which is not $Q$ for the M/M/1 queue, but the mean response time is $Q/p_{10}$, which is the mean queueing time for the M/M/1 queue with arrival rate $R/p_{10}$.

It is of some interest in obtaining solutions for networks by hand or with a calculator that the number of visits to the queue has a geometric distribution with parameter $p_{10}$, i.e.,

$$\text{Prob[1 visit]} = p_{10},$$

$$\text{Prob[2 visits]} = p_{11}p_{10},$$

$$\text{Prob}[3 \text{ visits}] = p_{11}^2 p_{10},$$

...

$$\text{Prob}[n \text{ visits}] = p_{11}^{n-1} p_{10} = (1 - p_{10})^{n-1} p_{10}.$$

It turns out that a critical parameter in the solution of a network is the expected number of visits a job makes to a class. We call this the *relative throughput* of the class. Though we can find the relative throughputs by solution of a (small) set of linear equations, if we are solving the model without a program, then we would like to avoid the extra solution step.

## 4.2 OPEN NETWORKS

Consider the network of Figure 4.11. There are two queues in series, with each queue consisting of a single class. Assume that both queues are FCFS with exponential service with respective means $1/a_1$ and $1/a_2$. If we let state $(n_1,n_2)$ be the state with $n_1$ jobs at queue (class) 1 and $n_2$ jobs at queue (class) 2, then we have the state transition diagram of Figure 4.12.

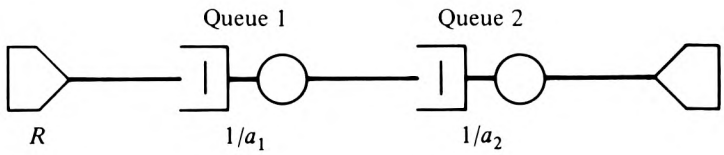



Figure 4.11

We can see that $U_1 = R/a_1$ and $U_2 = R/a_2$. Using local balance or by solving the global (full) balance equations we get

$$P(n_1,n_2) = U_1^{n_1}(1 - U_1)U_2^{n_2}(1 - U_2) \tag{4.57}$$

$$= P_1(n_1)P_2(n_2). \tag{4.58}$$

where $P_i(n_i)$ is the queue length distribution for an M/M/1 queue with arrival rate $R$ and mean service $1/a_i$. This is not a particularly surprising result since we can show that the two queues are independent of each other and that the arrivals at the second queue are Poisson with rate $R$.

Now consider Figure 4.13 which is the same as Figure 4.11 with an added loop. With the same assumptions as before, this produces the state diagram of Figure 4.14.

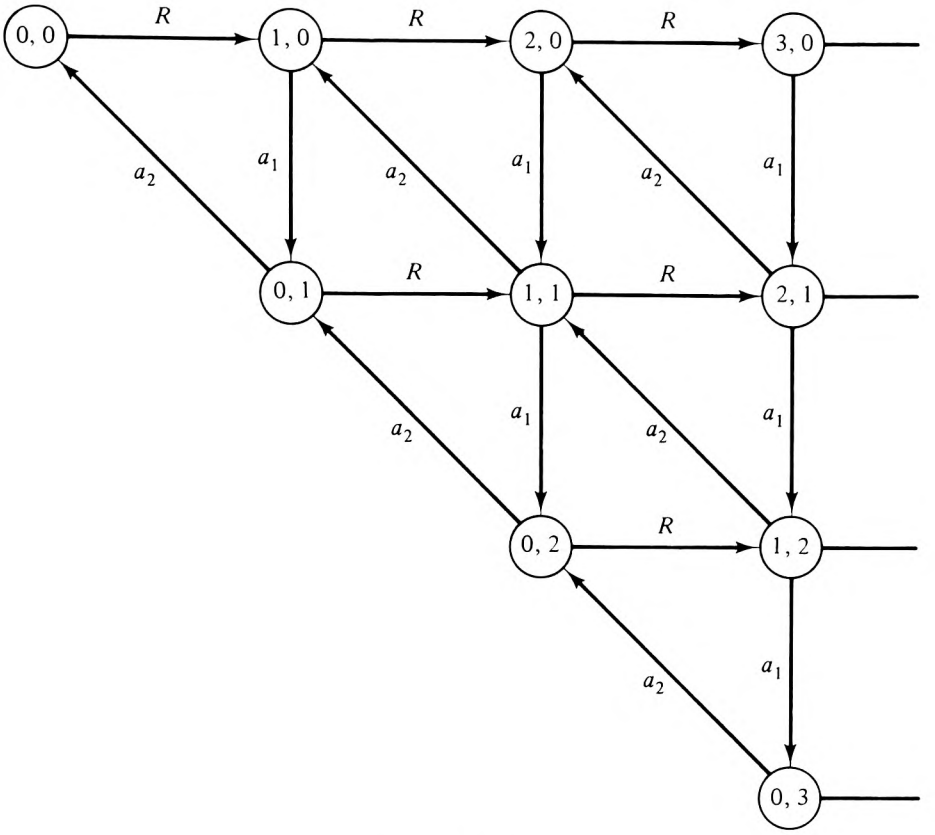



Figure 4.12

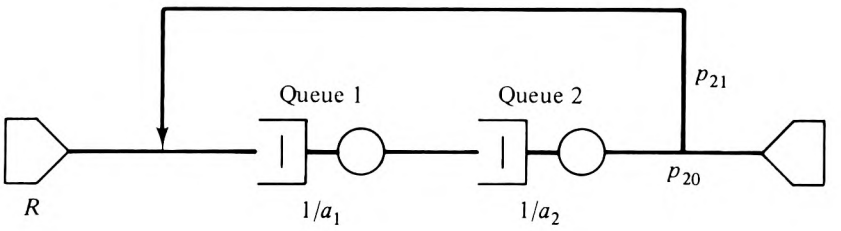



Figure 4.13

Let $r_1$ be the expected number of visits a job makes to queue 1 and $r_2$ be the expected number of visits a job makes to queue 2. The number of visits has a geometric distribution with parameter $p_{20}$, so by (2.4)

$$r_1 = r_2 = \frac{1}{p_{20}}. \tag{4.59}$$

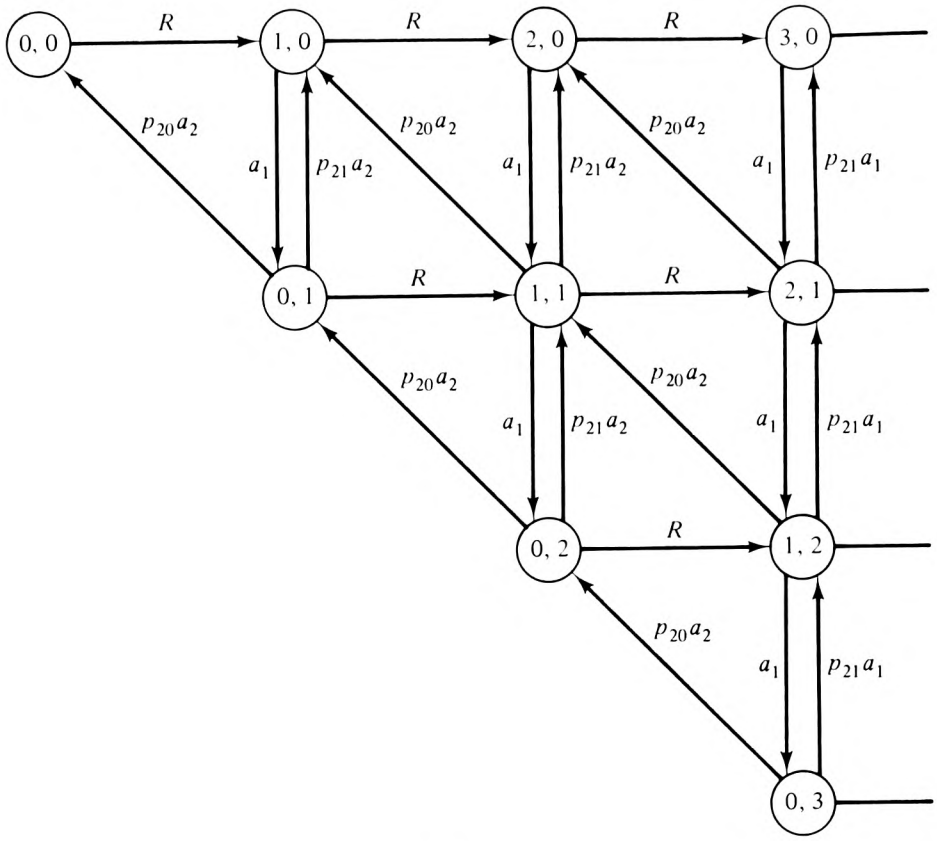



Figure 4.14

Assuming that neither queue is saturated, i.e., $U_1 < 1$ and $U_2 < 1$, then by equation (2.7)

$$U_1 = \frac{Rr_1}{a_1}$$

and

$$U_2 = \frac{Rr_2}{a_2}.$$

Using these values for $U_1$ and $U_2$, equations (4.57) and (4.58) are still valid!

This is a restricted form of a result known as *Jackson's Theorem* because Jackson was first to recognize this product form solution [JACK63]. Jackson's Theorem allows us to say that in an open network of queues with

Poisson arrivals (from outside the network), FCFS queues with exponential service times and no saturated queues,

1. Each individual queue may be treated as an M/M/1 queue with arrival rate equal to the throughput to obtain its queue length distribution.
2. In a network of $M$ queues, the joint queue length distribution of the network is the product of the queue length distributions of each of the queues, i.e.,

$$P(n_1,n_2,...,n_M) = P_1(n_1)P_2(n_2)...P_M(n_M).$$

Since Jackson's work, this result has been extended to include all of the queues of Section 4.1 *except FCFS queues with non-exponential or class dependent service time distributions.* If we look at (4.1) and assume $S_m$ , $m = 1, 2, ..., M$ is a state description we have used for an isolated queue other than the excluded FCFS queues, then (4.1) holds for $G = 1$. This result extends to other kinds of queues, e.g., queues with LCFSPR scheduling [CHAN72, BASK72, CHAN75a, REIS75, CHAN77b].

For an individual queue of the network, we can apply all of our isolated queue results (except for those for the excluded FCFS queues) once we have determined the arrival rate at the queue. The most interesting network measures, e.g., mean network population and response time, can be determined without ever dealing with probabilities of network states. The only remaining problem is determining the relative throughputs, i.e., the expected number of visits a job makes to a queue (or a class at that queue). The arrival rate at the queue (or class) will be $Rr$ where $r$ is the relative throughput. For many systems we may be able to determine the relative throughputs by inspection as in Figure 4.13. We now give a general approach.

Let us assume that there are $C$ classes and $M$ queues in the network; each queue has at least one class. Let $p_{0c}$ be the probability an arriving job from the source first joins class $c$, let $p_{c0}$ be the probability a job leaving class $c$ goes to the sink, and let $p_{cd}$ be the probability a job leaving class $c$ goes to class $d$, $1 \le c \le C$, $1 \le d \le C$. Let $r_c$ be the relative throughput for class $c$. We must have

$$r_c = p_{0c} + \sum_{d=1}^{C} r_d p_{dc}. \tag{4.60}$$

The first term of (4.60) is the direct contribution of the source to $r_c$. Each term of the summation is the contribution of class d to $r_c$. By letting c range from 1 to $C$, (4.60) gives us a system of $C$ linear equations in $C$ unknowns; $C$ is usually small enough that a numerical solution is trivial.

For the example of Figure 4.13, $p_{01} = 1$, $p_{02} = 0$, $p_{10} = 0$, $p_{11} = 0$, $p_{12} = 1$ and $p_{22} = 0$. From (4.60)

$$r_1 = 1 + r_1 \times 0 + r_2 p_{21} \tag{4.61}$$

and

$$r_2 = 0 + r_1 \times 1 + r_2 \times 0. \tag{4.62}$$

By (4.62), $r_1 = r_2$, so by (4.61)

$$\begin{aligned} r_1 &= 1 + r_1 p_{21} \\ &= \frac{1}{1 - p_{21}} \\ &= \frac{1}{p_{20}}. \end{aligned}$$

To summarize, to obtain solutions for individual queues of an open, product form network of single server queues, we

1. Obtain $r_c$, $c = 1, 2, \ldots, C$ by (4.60).
2. Obtain $U_c$, $c = 1, 2, \ldots, C$ as $Rr_c E[x_c]$.
3. Let $\mathscr{C}_{(m)}$ be the set of classes at queue $m$ and denote these classes as $\mathscr{C}_{(m)} = \{c_1, c_2, \ldots, c_{C_{(m)}}\}$ where $C_{(m)}$ is the number of classes in $\mathscr{C}_{(m)}$. Then for queue $m$

$$U_{(m)} = U_{c_1} + U_{c_2} + \ldots + U_{c_{C_{(m)}}}.$$

4. Obtain for class $c$ in $\mathscr{C}_{(m)}$

$$L_c = \frac{U_{(m)} U_c}{1 - U_{(m)}} + U_c.$$

5. Obtain for queue $m$

$$L_{(m)} = \frac{U_{(m)}^2}{1 - U_{(m)}} + U_{(m)}.$$

6. Obtain for class $c$ in $\mathscr{C}_{(m)}$

$$Q_c = \frac{U_{(m)} E[x_c]}{1 - U_{(m)}} + E[x_c] = \frac{L_c}{Rr_c}.$$

7. Obtain for queue $m$

$$r_{(m)} = r_{c_1} + r_{c_2} + \ldots + r_{c_{C_{(m)}}},$$

$$E[x_{(m)}] = \frac{r_{c_1}E[x_{c_1}] + r_{c_2}E[x_{c_2}] + \ldots + r_{c_{C_{(m)}}}E[x_{c_{C_{(m)}}}]}{r_{(m)}}$$

$$Q_{(m)} = \frac{U_{(m)}E[x_{(m)}]}{1 - U_{(m)}} + E[x_{(m)}] = \frac{L_{(m)}}{Rr_{(m)}}.$$

8. Obtain the mean network population

$$\text{Population} = \sum_{c=1}^{C} L_c.$$

9. Obtain the mean network response time for a job

$$\text{Response Time} = \frac{\text{Population}}{R}.$$

There are two cautions:

1. The results are not valid if $U_{(m)}$ is greater than or equal to one for any queue $m$.
2. The results are not valid if any FCFS queue has non-exponential or class dependent service times.

## 4.3 FURTHER READING

We will look at open networks as models of communication systems in Chapter 9. For a more thorough treatment of queues in isolation and open queueing networks, see KLEI75.

## 4.4 EXERCISES

4.1 Obtain $U$, $L$ and $Q$ for a FCFS queue with 3 servers and exponential service time, i.e., the M/M/3 queue.

4.2 Obtain the queue length distribution for the M/M/$\infty$ queue. Hint: for $-\infty < x < \infty$,

$$\sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x.$$

4.3 Verify (4.40).

4.4 Derive (4.41) from (4.40).

4.5 Obtain the queue length distribution, $U$, $L$ and $Q$ for a PS queue with two servers and service times with the branching Erlang distribution with two stages.

4.6 Repeat 4.5 with an infinite number of servers.

4.7 Define and obtain state probabilities for a Markov process representing a single server PS queue with two classes. Each class has a branching Erlang distribution with two stages and class dependent probabilities and means.

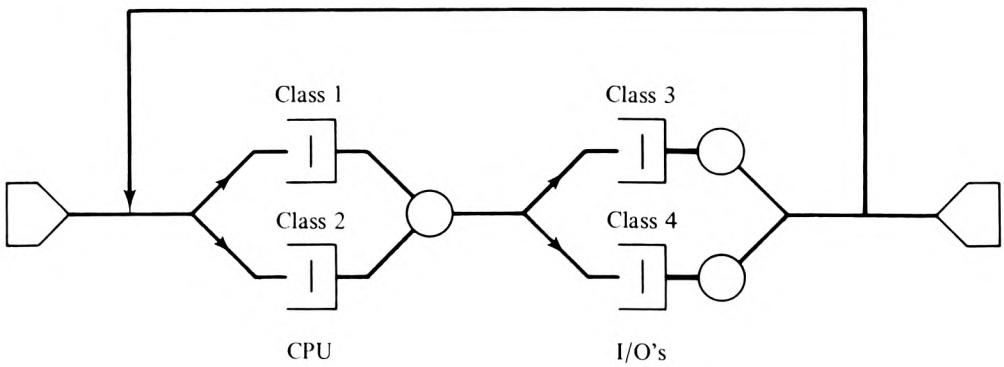
4.8 Define and obtain state probabilities for a Markov process representing a single server FCFS queue with two classes. Each class has the same exponential service time distribution. Obtain the queue length distribution.

4.9 Obtain the queue length distribution for an M/M/1 queue with queue dependent service times, i.e., with queue length $n$, the remaining service time is exponential with mean $1/a(n)$. Assume that $a(n)$ is constant for $n \geq N$, i.e., $a(N) = a(N+1) = \ldots$ .

4.10 Obtain the queue length distribution for an M/M/1 queue with queue dependent arrival rates, i.e., with queue length $n$, the arrival rate is $R(n)$. Assume that $R(n)$ is constant for $n \geq N$, i.e., $R(N) = R(N+1) = \ldots$ .

4.11 Repeat 4.10 for the network of Figure 4.11, where the arrival rate depends only on the *total* network population of jobs. You need not solve for $P(0,0)$.

4.12 Obtain the mean network population and response time for the following model. You may assume exponential service times at all queues and PS at the CPU. I/O scheduling is FCFS. Use the notation we have used, i.e., $R$ for the arrival rate, $1/a_c$ for the mean service at class $c$ etc.



4.13 Generalize the algorithm for networks of single server queues at the end of Section 4.3 to allow product form networks with an arbitrary number of servers at each queue.

## 4.5 SUMMARY OF CHAPTER NOTATION

| | |
|---|---|
| $M$ | Number of queues |
| $G$ | Normalizing constant |
| $R$ | Poisson arrival rate (throughput if queue not saturated) |
| $a$ | Mean service rate, i.e., $1/a$ is mean service time |
| $U$ | Utilization |
| $L$ | Mean queue length |
| $Q$ | Mean queueing time |
| $W$ | Mean waiting time |
| $C_x$ | Coefficient of variation of random variable $x$ |
| $V_{ij}$ | Probability of visiting stage $j$ of branching Erlang distribution after leaving stage $i$ |
| $p_{ij}$ | Probability of visiting class $j$ after visiting class $i$ |
| $C$ | Number of classes |
| $r$ | Relative throughput |

# *CHAPTER 5*

# CLOSED PRODUCT FORM QUEUEING NETWORKS

We next discuss the concept of local balance which provides the theoretical underpinning for many of the results of queueing networks. The concept is simple, the mathematics is straightforward and the algebra is minimal. We assume no background of the reader other than the earlier chapters.

The reader who is primarily interested in applications may skip Sections 5.1 through 5.5 and go directly to Section 5.6. Readers who study these sections will find some of the concepts reviewed in Section 5.6.

## 5.1 THE THEORY OF LOCAL BALANCE

We shall briefly review some of the concepts of Chapter 4 which are necessary to the understanding of local balance. Local balance is a characteristic of some Markov processes. We will speak of "queueing systems satisfying local balance" when we mean that the underlying Markov processes satisfy local balance. Typically, a queueing system will satisfy local balance if the queues have queueing disciplines, service time distributions and arrival processes compatible with local balance in the underlying Markov process.

Consider a queue with $C$ customer classes fed by a Poisson source, where the arrival rate of class $c$ customers is $R_c$, $c = 1,...,C$. ($R_c = Rr_c$ where $R$ and $r_c$ are defined as before.) See Figure 5.1. Assume that the service time for class $c$ is an independent exponential random variable, $c = 1,...,C$. We shall next define queueing disciplines which satisfy local balance. We shall illustrate the definition by considering two disciplines: LCFSPR and FCFS. In this section we restrict attention to single queues (in isolation) fed by Poisson sources as shown in Figure 5.1, and we assume that this system reaches equilibrium.

### 5.1.1 Feasible States for the Single Queue Case (Figure 5.1)

Let $S$ be any feasible state and let there be $n$ jobs in the queue. In the LCFSPR case a state is a stack $(c_1,...,c_n)$ where $c_i$ is the class of the $i^{th}$ job in the stack, and the first job is on top of the stack. The job on top of the

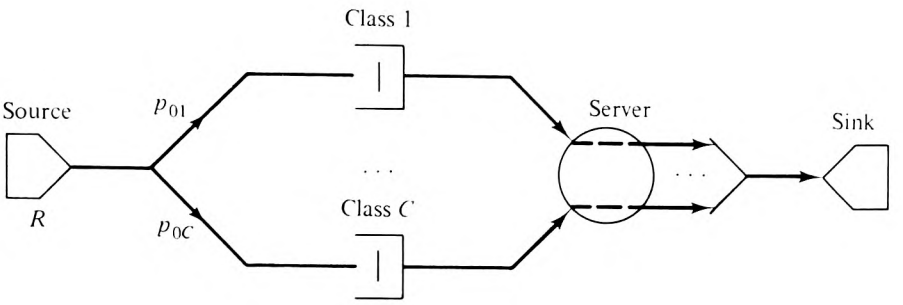



Figure 5.1

stack is currently being served. When it finishes, it is popped off the stack, and the new state becomes $(c_2,...,c_n)$ and the next job on top of the stack (with class $c_2$) begins service. If a job in class $c_0$ arrives while the system is in state $(c_1,...,c_n)$, the job at the top of the stack is preempted in favor of the new job and the new state becomes $(c_0,c_1,...,c_n)$. A state in the FCFS case is a queue $(c_1,c_2,...,c_n)$ where $c_i$ is the class of the job in the $i^{th}$ position in the queue. Only the job at the head of the queue (in position 1) is served. New jobs join the tail of the queue.

### 5.1.2 State Transitions for the Single Queue Case (Figure 5.1)

**5.1.2.1 Job departure.** *If* a class $c$ job can be served in state $S$, let $S-(c)$ be the state resulting from the departure of a class $c$ job from the system, when the system is in the state $S$. $S-(c)$ is undefined if class $c$ jobs are not served in state $S$. In the LCFSPR case only the job in class $c_1$ can be served in the state $S = (c_1,...,c_n)$. Hence we only define $S - (c_1) = (c_2,...,c_n)$. The same definition holds for FCFS. We restrict attention to disciplines where, for any class $c$, $S-(c)$ is either a unique state, or is undefined.

**5.1.2.2 Job arrival.** Let $S+(c)$ be the state resulting from the arrival of a class $c$ job to the system when it is in state $S$. For LCFSPR if $S = (c_1,...,c_n)$ then $S + (c_0) = (c_0,c_1,...,c_n)$. For FCFS, $S + (c_0) = (c_1,...,c_n,c_0)$. We restrict attention to disciplines where, for any class $c$, $S+(c)$ is a unique state.

### 5.1.3 The Local Balance Equation

Let $a_c(S)$ be the rate at which class $c$ jobs are served in state $S$. If class $c$ jobs are not served in state $S$ then $a_c(S) = 0$. Let $P(S)$ be the equilibrium probability of state $S$. We assume the convention that if $S$ is an infeasible state $P(S) = 0$. The local balance equation is:

$$P(S)a_c(S) = P(S-(c))R_c \tag{5.1}$$

and its dual is

$$P(S + (c))a_c(S + (c)) = P(S)R_c \tag{5.2}$$

The first (second) equation states that the rate of entry into state $S$ due to the arrival (departure) of a class $c$ job is equal to the rate of departure from state $S$ due to the departure (arrival) of a class $c$ job. We also require that the arrival of a class $c$ job, when the system is in state $S-(c)$, take the system to state $S$, and the dual of this requirement is that the departure of a class $c$ job when the system is in state $S+(c)$ take the system to $S$. In other words $(S-(c))+(c) = S$ and $(S+(c))-(c) = S$. Thus equation (5.1) states that: the rate of transaction from $S$ to $S-(c)$ equals the rate of transition from $S-(c)$ to $S$. Equation (5.2) states the dual: the rate of transition from $S+(c)$ to $S$ equals the rate of transition from $S$ to $S+(c)$. The local balance equation is depicted in Figure 5.2. The concept of local balance is indeed very simple. In a nutshell, it states that between any pair of states there should either be no transition at all or transitions should be in both directions and the rates in both directions should be equal. For LCFSPR, the reader should show that $(S+(c))-(c) = S$ for any $S$ and any $c$ and also show that $(S-(c))+(c) = S$ for any $S$ and $c$ where $S-(c)$ is defined. Hence we know that for LCFSPR, between any pair of states, there are either no transitions at all, or transitions are in both directions. The Markov diagram (Figure 5.3) shows this fact pictorially. For FCFS the reader should show that $(S+(c))-(c)$ is defined and is equal to $S$ if and only if $S = ()$ or $S = (c_1,...,c_n)$ where $c_1 = ... = c_n = c$. Similarly $(S-(c))+(c)$ is defined and equals $S$ only if $c_1 = ... = c_n = c$. Hence unless there is only one customer class, we could have the case that there is a transition from some state $S$ to a state $S'$ but no transition back from $S'$ to $S$.

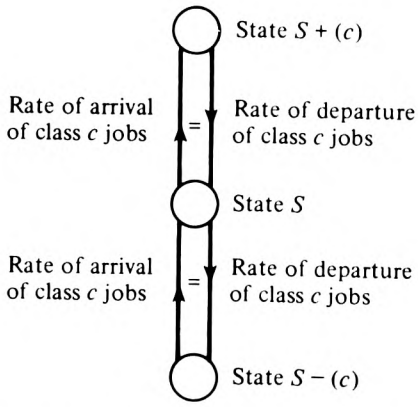



Figure 5.2 The local balance equation in pictorial form

A queue (Figure 5.1) satisfies local balance if and only if it satisfies local balance for every state $S$ and every class $c$. Then, for every pair of (feasible) states $S$ and $S'$, we have:

$$\begin{aligned} &\text{transition rate from } S' \text{ to } S \\ &= \text{transition rate from } S \text{ to } S' \end{aligned} \tag{5.3}$$

A queueing system (or a corresponding Markov process) does not need to satisfy local balance but just the balance equations introduced in Chapter 3. The balance (or equilibrium) equation for a state S is that the rate of transition into $S$ equals the transition rate out of $S$. This equation is called *global* balance to distinguish it from *local* balance. Global balance: For every (feasible) state $S$,

$$\begin{aligned} &\sum_{S'} \text{transition rate from } S' \text{ to } S \\ &= \sum_{S'} \text{transition rate from } S \text{ to } S' \end{aligned} \tag{5.4}$$

Local balance is a sufficient (though not necessary) condition for global balance because if local balance is satisfied, then summing the local balance equation (5.3) over all $S'$ gives the global balance equation, (5.4).

Let us consider whether LCFSPR satisfies local balance. Though we could consider a more general case as in CHAN77 let us construct the Markov diagram with 2 classes (Figure 5.3).

Let $U_c = R_c/a_c$. (The fraction of time the server spends on class $c$ jobs is $U_c$.) We shall show that

$$P(c_1,\ldots,c_n) = \frac{U_{c_1}\ldots U_{c_n}}{G}, \tag{5.5}$$

where $G$ is a normalization constant, satisfies the local balance equations. We must show that the transition rate from $S$ to $S'$ equals the rate from $S'$ to $S$ for every pair of states $S$, $S'$. The rate of transition from $(c_1,\ldots,c_n)$ to $(c_2,\ldots,c_n)$ is

$$P(c_1,\ldots,c_n)a_{c_1} = \frac{U_{c_2}\ldots U_{c_n}}{G} R_{c_1}. \tag{5.6}$$

The rate of transition from $(c_2,\ldots,c_n)$ to $(c_1,\ldots,c_n)$ is

$$P(c_2,\ldots,c_n)R_{c_1} = \frac{U_{c_2}\ldots U_{c_n}}{G}R_{c_1}. \tag{5.7}$$

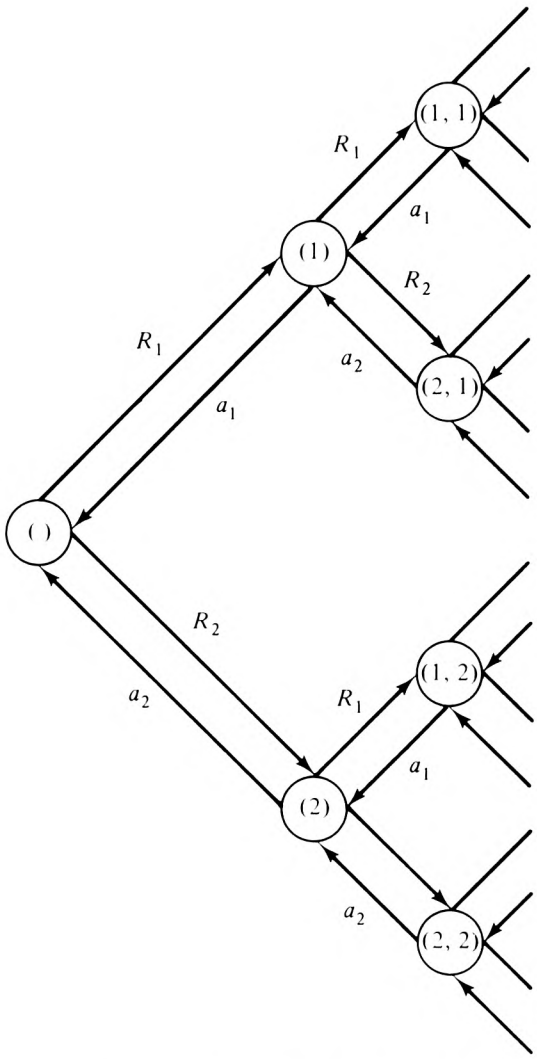



Figure 5.3 Markov diagram for LCFSPR.

Since the right hand sides of (5.6) and (5.7) match, all of the local balance equations are satisfied. Thus equation (5.5) is correct.

In the two-class FCFS case there are states $S$ such that $(S+(c))-(c)$ is not defined; hence FCFS does not satisfy local balance in this case. The

reader is encouraged to compare the Markov diagrams for the FCFS and LCFSPR cases.

## 5.2 NETWORKS

### 5.2.1 Definitions

Let there be $M$ queues in the network. Associated with queue $m$ is a set of classes, referred to as $\mathscr{C}(m)$. Let there be a total of $C$ classes in the network indexed 1,...,$C$. We shall assume that any sources and sinks belong to class 0 (Figure 5.4). The probability that a job completing service in class $i$ joins class $j$ is $p_{i,j}$, $i = 0,...,C$, $j = 0,...,C$ The service times and the disciplines for each queue are independent of all other queues in the network.

If a class $i$ job can become a class $j$ job, possibly after passing through intermediate classes, we shall say that $j$ is *reachable* from $i$. We define a *chain* $k$ to be a set of classes such that for any pair of classes $i$ and $j$ in chain $k$, $j$ is reachable from $i$ and $i$ is reachable from $j$; furthermore, there is no class $c$ in the network, where $c$ is not in chain $k$, such that $c$ is reachable from any class in chain $k$ or that any class in chain $k$ is reachable from $c$.

In Figure 5.4, classes 1, 2 and 4 belong to one chain; classes 0, 5 and 6 belong to another. Class 3 does not belong to any chain. It is easy to see that class 3 is a *transient* class, i.e., there will be no jobs in that class at equilibrium. Since we are only considering equilibrium conditions we shall ignore all classes which are not in chains because they must be transient classes. The chain which includes class 0 is said to be the *open* chain. (Note that with our definitions of class 0 and chains, it is not possible for a network to have more than one open chain. If desired it is simple to use alternate definitions and consider multiple open chains.) A chain which is not open is said to be *closed*. The number of jobs in a closed chain is constant at all times: this number is called the *population* of the chain. Let there be $A$ closed chains in the network. The *population vector* $N$ of the network is $N = (N_1,...,N_A)$ where $N_k$ is the population of chain $k$, $k = 1,...,A$. Of course, the population vector is not concerned with open chains. A network with an open chain and no closed chains is said to be an *open* network. A network with closed chains and no open chain is said to be a *closed* network. A network with both open and closed chains is said to be *mixed*.

A network in which all queues satisfy local balance in isolation is called a *local balance network*. In the following discussion we restrict attention to local balance networks.

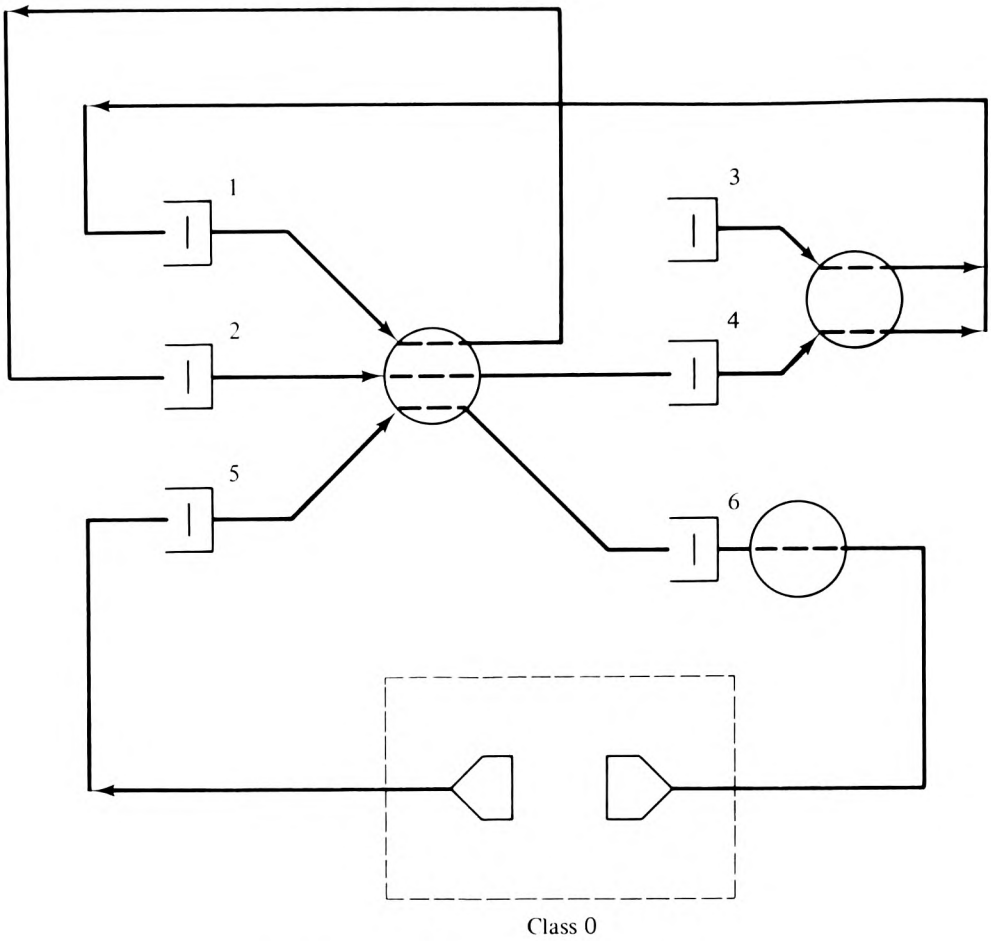



Figure 5.4 Example of a network.

Let $R_c$ be the throughput of class $c$ jobs in the network, i.e., $R_c$ is the equilibrium rate at which jobs leave class $c$. Since the rate of flow of jobs into a class must equal the rate of flow of jobs out of the class, we must have:

$$\sum_c R_c p_{c,c'} = R_{c'}. \tag{5.8}$$

$R_0$, the rate of flow of jobs from the source (or to the sink) is given.

If $c$ belongs to a closed chain $k$, we define the *relative* throughput, $r_c$, as any set of positive numbers such that

$$\sum_c r_c p_{c,c'} = r_{c'} \tag{5.9}$$

for all $c'$ in chain $k$. If we multiply the relative throughputs by a positive constant $D$, we see that equation (5.9) is still satisfied because both sides of the equation are multiplied by $D$. Thus if $\{r_c \mid c \text{ in closed chain } k\}$ is a set of relative throughputs, then so is $\{Dr_c \mid c \text{ in closed chain } k\}$. Thus we cannot solve equation (5.9) to get unique values for the relative throughputs. However, if we set the relative throughput of any class $c$ in chain $k$ to an arbitrary positive value, we can use equation (5.9) to solve for the relative throughputs of all other classes in the chain. The magnitudes of the relative throughputs are immaterial; what is material is that the relative throughputs be *consistent*, i.e., that they satisfy equation (5.9).

(What we have said here should be qualified somewhat in regard to computational algorithms. Until recently, the best available algorithms were sensitive to the magnitudes of the relative throughputs, particularly when network populations were large [REIS78b]. The *Mean Value Analysis* Algorithm of REIS78a is insensitive to the magnitudes of the relative throughputs. One of the other algorithms we present, though less sensitive to the magnitudes of the relative throughputs than the algorithms in REIS78b, is still somewhat sensitive. We will discuss the Mean Value Analysis Algorithm and numerical requirements on the choice of relative throughputs in Section 5.7.3.)

From equations (5.8) and (5.9), we have

$$R_c = r_c B(k), \; c \text{ in chain } k, \tag{5.10}$$

where $B(k)$ is a positive proportionality constant for chain $k$.

Let $L_c$ be the mean number of class $c$ jobs in the network. For any closed chain $k$:

$$\sum_{c \text{ in } k} L_c = N_k,$$

where $N_k$ is the population of chain $k$, since the number of chain $k$ jobs within the system must always be $N_k$.

### 5.2.2 The Markov Process Solution of a Local Balance Network

For any queueing network representable as a Markov process, in obtaining its solution we could take the following steps:

1. Determine the set of feasible network states. (We will use an overbar to indicate network states, e.g., $\overline{S}$ is a network state but $S$ is an individual queue state.)

2. Determine if it is possible to go from every feasible network state $\overline{S}$ to every feasible network state $\overline{S}'$ after one or more transitions with non-zero probability; if it is possible to go from $\overline{S}$ to $\overline{S}'$ with non-zero probability, then $\overline{S}'$ is said to be *reachable* from $\overline{S}$. If every state is reachable from every state, then we do not have to worry about the initial condition of the network, because we know that regardless of where the network starts every state will be traversed eventually. We assume here that this is the case.

3. Determine the global balance equations for each state.

4. Solve the global balance equations and obtain performance measures from the solution.

These steps may be difficult in general, but they are fairly simple for local balance networks.

**5.2.2.1 The set of feasible network states.** If we have no simple algorithm to construct the network state space, then network analysis is quite difficult! Fortunately, for local balance networks, there is a simple relationship between the state space of each queue *in isolation*, e.g., fed by a Poisson source as in Figure 5.1, and the network state space.

We shall consider mixed networks because they are the most general. The analysis for closed or open networks is straightforward. Let the network have $K$ chains the first $A$ of which are closed. Let $N = (N_1,...,N_A)$ be the population vector of the network. Let $S_m$ be any feasible state of queue $m$ in isolation (Figure 5.1). Let $\text{POP}(S_m)$ be an $A$ element vector whose $k^{th}$ element is the number of chain $k$ jobs in queue $m$ in state $S_m$.

A state $\overline{S}$ is a feasible network state if and only if $\overline{S} = (S_1,...,S_M)$ where $S_m$ is a feasible state of queue $m$ in isolation, and

$$\text{POP}(S_1) + ... + \text{POP}(S_M) = N$$

Furthermore, every state in the network state space is reachable from every state in the space. A transition from a network state $\overline{S}$ to another network state $\overline{S}'$ is permitted if and only if:

$$\bar{S} = (S_1,\ldots,S_M)$$

$$\bar{S}' = (S_1,\ldots,S_{i-1},S_i - (c),S_{i+1},\ldots,S_{j-1},S_j + (c'),S_{j+1},\ldots,S_M)$$

and

$$p_{c,c'} \neq 0,$$

for some $c$ and $c'$, where class $c$ belongs to queue $i$ and class $c'$ to queue $j$. The transition takes place because a job in class $c$ becomes a member of class $c'$. The reader should prove this is true. (The proof is not completely trivial; consider a network in which none of the queues have local balance in isolation such as Figure 5.5. Note that it is impossible for the system to go from state (1, 2, 3) to state (1, 3, 2) in the network shown in Figure 5.5. This shows that if the network is not a local balance network, investigation of the state space is not likely to be trivial.)

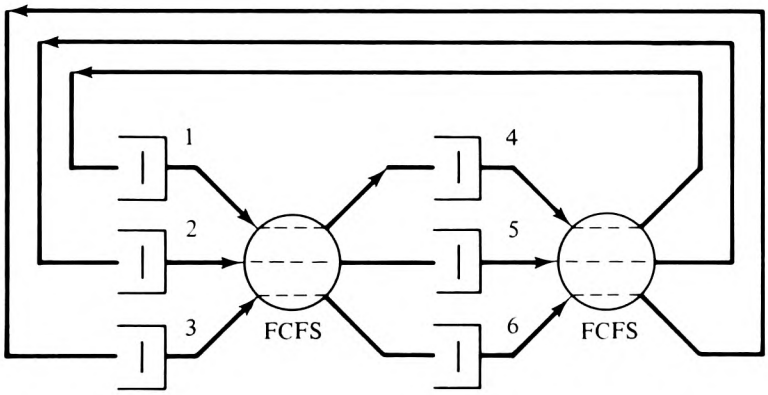



Figure 5.5 A non-local balance network

**5.2.2.2 The balance equations for a local-balance network.** Fortunately, we do not have to derive the balance equations for each network from scratch! We shall now write down the generic form for balance equations for all local balance networks. To simplify notation we shall write:

$$\overline{(S + (c) - (c'))}$$

for

$$S_1,\ldots,S_{j-1},S_j + (c),S_{j+1},\ldots,S_{i-1},S_i - (c'),S_{i+1},\ldots,S_M$$

The rate of transition from $\overline{(S + (c) - (c'))}$ to $\overline{S}$ is:

$$P(\overline{(S + (c) - (c'))}a_c(S_j + (c))p_{c,c'}. \tag{5.11}$$

This transition occurs because a class $c$ job enters class $c'$. Note that $(S + (c) - (c'))$ is a network state in equation (5.11) but that $S_j + (c)$ is an individual queue state. The net rate at which state $\overline{S}$ is entered because a job enters class $c'$ is:

$$\sum_c \text{ rate } \overline{(S + (c) - (c'))} \text{ to } \overline{S}. \tag{5.12}$$

Hence the net rate at which $S$ is entered is:

$$\sum_{c'} \sum_c \text{ rate } \overline{(S + (c) - (c'))} \text{ to } \overline{S}. \tag{5.13}$$

The rate at which class $c'$ jobs are served in state $\overline{S}$ is $a_{c'}(S_{q(c')})$, where we define $q(c')$ as the queue to which $c'$ belongs. Hence the rate at which the system departs state $\overline{S}$ because a job leaves class $c'$ is

$$P(\overline{S})a_{c'}(S_{q(c')}). \tag{5.14}$$

The net rate at which the system departs state $\overline{S}$ is

$$\sum_{c'} P(\overline{S})a_c(S_{q(c')}). \tag{5.15}$$

The global balance equation equates the rates of arrival to (5.13) and departure from (5.15) state $\overline{S}$.

**5.2.2.3 Equilibrium state probabilities.** Fortunately, we do not have to solve the global balance equations numerically for each local balance network separately to compute equilibrium state probabilities! When we were describing the state space of the network we did so by relating the network state space to the state space of each queue in isolation. We shall use the same method here: we shall obtain network state probabilities from the state probabilities of each queue in isolation. Since we can analyze a local balance queue in isolation very simply from the local balance equations, the equilibrium state probabilities for each queue in isolation are readily obtain-

able. Computing network state probabilities is then conceptually straightforward.

We now discuss the setting of the arrival rates for each job class when we analyze a queue in isolation. Recall that $r_c$ is the relative throughput of class $c$, if $c$ is a member of a closed chain, see equations (5.8-5.10). For each closed chain choose a set of relative throughputs. If class $c$ belongs to queue $m$, and class $c$ is a member of a closed chain, set the arrival rate for class $c$ to queue $m$ *in isolation*, to the relative throughput of class $c$. If class $c$ is a member of the open chain, set its arrival rate to queue $m$ *in isolation* to the *actual* throughput, $R_c$ of class $c$. The actual throughput for classes in the open chain are determined easily from equation (5.8) since we know $R_0$, the source rate. ($R_0$ is equivalent to $R$ of Chapter 4; equation (5.8) is equivalent to equation (4.60).) We assume that the arrival rates for the closed chains are small enough so that queue $m$ reaches equilibrium in isolation. (We can always choose the relative throughputs to satisfy this assumption.)

In the following proofs it is helpful to define:

$$\bar{S} = S_1,\ldots,S_M \tag{5.16}$$

$$\overline{(S - (c'))} = S_1,\ldots,S_{i-1},S_i - (c'),S_{i+1},\ldots,S_M \tag{5.17}$$

and

$$p(\bar{S}) = P(S_1)\ldots P(S_M) \tag{5.18}$$

Note that $P(S_i)$ is obtained from analyzing queues in *isolation*.

**Theorem:**

$$P(\bar{S}) = \frac{p(\bar{S})}{G} \tag{5.19}$$

if $\bar{S}$ is feasible, where $G$ is a normalization constant.

**Proof:**

We are given that all queues satisfy local balance, i.e., satisfy equations (5.1) and (5.2), in isolation. Applying local balance equation (5.2) to (5.11) and assuming (5.19) we get

$$\text{rate } \overline{(S + (c) - (c'))} \text{ to } S = \frac{P(\overline{(S - (c'))})}{G} r_c p_{c,c'}. \tag{5.20}$$

Hence from (5.12) and (5.9) the net rate at which $S$ is entered because a job enters class $c'$ is:

$$\frac{P(\overline{(S - (c'))})}{G} r_{c'}. \tag{5.21}$$

Applying local balance equation (5.1) to (5.14), the rate at which the system departs $S$ because a job leaves $c'$ is also:

$$\frac{P(\overline{(S - (c'))})}{G} r_{c'}. \tag{5.22}$$

Hence the balance equations are satisfied by $P(\bar{S})$ defined in (5.19). This completes the proof.

The *normalization constant*, $G$, can be obtained from

$$\sum_{\text{feasible } \bar{S}} P(\bar{S}) = 1. \tag{5.23}$$

Hence

$$G = \sum_{\bar{S}} p(\bar{S}) \tag{5.24}$$

where the summation is taken over all $\bar{S} = (S_1,...,S_M)$ where $S_m$ is feasible for queue $m$ in isolation and

$$\text{POP}(S_1) + ... + \text{POP}(S_M) = N. \tag{5.25}$$

We have now completed the analysis of the Markov process for local balance networks.

## 5.3 NON-EXPONENTIAL SERVICE TIMES

Up to this point we have assumed that all service times are exponential. We now consider non-exponential service times which can be represented as a network of exponential stages. An Erlang distribution and a branching

Erlang distribution are shown in Figure 5.6. Consider a local balance network in which all classes have exponential service times and in which some queue $m$ has 2 classes, say $c$ and $c'$, and suppose a job leaving class $c$ immediately enters class $c'$ (Figure 5.7). Assume that class $c$ and class $c'$ have the same mean service time. Now consider another network in which queue $m$ has an Erlang service time with two stages: each stage has the same service time as each of the classes $c$ and $c'$. It is possible to show that if queue $m$ satisfies local balance and if it has a class independent discipline (i.e., one in which jobs are not given priority based on their class or amount of service received) then the Markov process for the network with two exponential classes of Figure 5.7 is identical to that for the network in which queue $m$ has the 2 stage Erlang service distribution. The reader should check this out for the LCFSPR case. Similarly, the 2 class network of Figure 5.8 models two stage branching Erlangs. If queue $m$ has a class-independent discipline, satisfies local balance and has classes $c_1,\ldots,c_k$, then by suitably interconnecting the classes, any $k$-stage service time can be modeled. The reader should prove this to be true for the LCFSPR case. We shall hereafter ignore non-exponential service times and restrict attention to local balance networks in which all classes have exponential service times. As stated in Chapter 4, only the mean time and the number of visits to each queue are relevant.

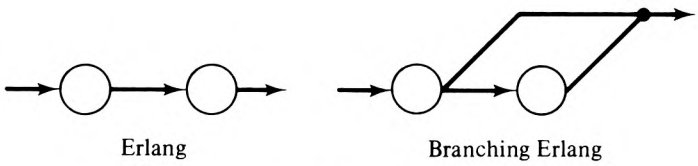



Figure 5.6

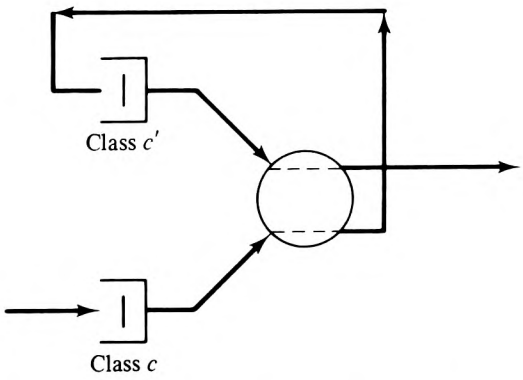



Figure 5.7 A class representation of an Erlang distribution

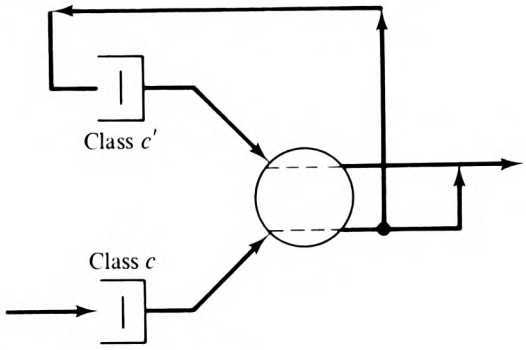



Figure 5.8
A class representation of a branching Erlang distribution

## 5.4 SOME IMPORTANT LOCAL BALANCE SYSTEMS

The reader should prove that queues with LCFSPR, PS or Infinite Servers (IS) with an arbitrary number of classes satisfy local balance. For these disciplines show that non-exponential service times can be modeled by suitably connecting classes with exponential service times. Also show that queues with FCFS and a *single* exponential class satisfy local balance. The reader can "cook-up" other local balance systems (indeed may enjoy cooking one up); however, few other local balance systems are practically meaningful. We next discuss an important, though apparently unusual, local balance system with an arbitrary number of classes. We call this system the *composite queue*. The composite queue with $C$ classes is defined by a $C$-dimensional, positive matrix $H$, called the rate matrix. The states of the queue in isolation are $C$-tuples, $(n_1,...,n_C)$ where $n_c$ is the number of class $c$ jobs in the queue. The rate at which class $c$ jobs are served in state $S = (n_1,...,n_C)$ is

$$a_c(S) = \frac{H(n_1,...,n_{c-1},n_c - 1,n_{c+1},...,n_C)}{H(S)}, \text{ for } n_c > 0. \qquad (5.26)$$

We leave it to the reader to show that

$$P(S) = \frac{H(S)R_1^{n_1}...R_C^{n_C}}{G}. \qquad (5.27)$$

Where $G$ is a normalization constant and $P(S)$ is the equilibrium probability of the queue, in isolation, when the arrival rate of class $c$ jobs is $R_c$, $c = 1,...,C$.

*Note:* Given *any* set of state probabilities $P(S)$ where $S$ is a $C$-tuple, we can find a composite queue which has the same equilibrium state probabilities by setting:

$$H(S) = \frac{P(S)}{R_1^{n_1} \ldots R_C^{n_C}} \tag{5.28}$$

This is quite remarkable! It implies that we cannot tell whether a system satisfies local balance by inspecting the numerical values of the equilibrium state probabilities.

We will find the composite queue useful when we model a complex system consisting of several queues by a single composite queue with the same equilibrium state probabilities.

**A special case.** A FCFS queue, with 2 or more classes, in which all classes have the *same* exponential service distribution behaves like a local balance queue as far as equilibrium probabilities are concerned provided all the states of the queue in isolation are also feasible states of the queue in the network, and every state is reachable from every other [BASK75]. Figure 5.5 gives an example of a network in which every state is not reachable from every other, and which does not behave like a local balance network as far as equilibrium probabilities are concerned even though all classes have the same exponential service distribution. Even though networks such as Figure 5.5 do not have the solution given in (5.19), the queue length distribution of such a network (and all performance measures derivable from the queue length distribution) will be the same as for a local balance network, provided that for each FCFS queue all classes of the queue have the same exponential service time distribution. The reader can prove these results from the Markov balance equations as in the local balance case.

## 5.5 PROPERTIES OF CLOSED LOCAL BALANCE NETWORKS

This section is restricted to closed local balance networks. The state space, state probabilities, performance measures and the normalization constant of closed networks depend on the population vector $N$. We shall show this dependence explicitly by writing $G(N)$, $R_c(N)$, $L_c(N)$, and $Q_c(N)$. Note that the relative throughput $r_c$ is independent of $N$. Define $e_k$ to be a vector with a 1 in the $k^{th}$ position and 0 elsewhere. Then $(N - e_k)$ is a population vector with one less job in chain $k$ than $N$.

### 5.5.1 Throughputs

**Throughput Theorem:**

The throughput of class $c$ customers when the population vector is $N$ is

$$R_c(N) = \frac{G(N - e_k)}{G(N)} r_c \tag{5.29}$$

where class $c$ is in chain $k$.

**Proof:**

Let class $c$ belong to queue $m$. From (5.22), the rate at which jobs leave class $c$ is

$$\sum_{\bar{S}} P(\bar{S}) a_c(S_m) = \sum_{(S - (c))} \frac{p(\overline{(S - (c))})}{G(N)} r_c. \tag{5.30}$$

From the fact that queue $m$ is in local balance in isolation, we know that the set of feasible states $S_m$ in which $a_c(S_m) > 0$ is identical to the set of feasible states $S_m - (c)$, for queue $m$ in isolation. The summation in (5.30) is taken over all feasible states $S_1,...,S_M$ such that

$$\text{POP}(S_1) + ... + \text{POP}(S_M) = N$$

and $a_c(S_m) > 0$, i.e., $S_m - (c)$ is feasible. Hence the summation is being taken over all feasible states $S_1,...,S_{m-1}$, $S_m - (c)$, $S_{m+1},...,S_M$ such that

$$\text{POP}(S_1) + ... + \text{POP}(S_m - (c)) + ... + \text{POP}(S_M) = N - e_k,$$

and the theorem follows from the definition of $G$ (see equations (5.24) and (5.25)).

### 5.5.2 Probabilities of Queue States (Marginal Probabilities)

Let $P_{(m)}(S_m \mid N)$ be the probability that queue $m$ in the network is in state $S_m$ given that the population vector is $N$. Note that $P_{(m)}(S_m \mid N)$ is concerned with queue $m$ *within* the network, whereas $P(S_m)$ is concerned with queue $m$ *in isolation*. $(P_{(m)}(S_m \mid N)$ is an example of a *marginal* probability because it considers only the state of queue $m$ and not the states of other queues in the network. The network state probabilities we have been dealing with are referred to as *compound* probabilities; a marginal probability is then the sum of compound probabilities.)

We use lower case letters to represent unnormalized probabilities; for a feasible network state $\bar{S}$ we have defined $p(\bar{S} \mid N) = P(\bar{S} \mid N)G(N)$ (see equations (5.18) and (5.19)). Define

$$p_{(m)}(S_m \mid N) = P_{(m)}(S_m \mid N)G(N). \tag{5.31}$$

The following theorem is crucial to algorithms for computing performance metrics of the models.

**The Marginal Local Balance Theorem:**

$$p_{(m)}(S_m \mid N)a_c(S_m) = p_{(m)}(S_m - (c) \mid N - e_k)r_c \tag{5.32}$$

where class $c$ belongs to chain $k$ and is in queue $m$.

**Proof:**

For notational simplicity, assume $m = 1$. By definition

$$p_1(S_1 \mid N) = P(S_1) \sum P(S_2)...P(S_M) \tag{5.33}$$

where the summation is taken over feasible $S_2,...,S_M$ (in isolation), such that

$$\mathrm{POP}(S_2) + ... + \mathrm{POP}(S_M) = N - \mathrm{POP}(S_1) \tag{5.34}$$

($P(S_m)$, $m = 1,...,M$, is still the probability that queue $m$, in isolation, is in state $S_m$.) Applying local balance equation (5.1)

$$p_1(S_1 \mid N)a_c(S_1) = r_c P(S_1 - (c)) \sum P(S_2)...P(S_M)$$

where the summation is taken over $S_2,...,S_M$ satisfying (5.34), which is equivalent to

$$\mathrm{POP}(S_2) + ... + \mathrm{POP}(S_M) = N - e_k - \mathrm{POP}(S_1 - (c)),$$

and the theorem follows from the definition of $p_1(S_1 - (c) \mid N - e_k)$ (see equation (5.33)).

**Lemma:**

$$P_{(m)}(S_m \mid N)a_c(S_m) = P_{(m)}(S_m - (c) \mid N - e_k)R_c(N) \tag{5.35}$$

The proof follows from the Marginal Local Balance and Throughput Theorems.

Equations (5.32) and (5.35) look exactly like the local balance equation except that they deal with (unnormalized and normalized) *marginal* probabilities; hence we refer to these equations as marginal local balance equations.

Repeated use of the Marginal Local Balance equation yields performance statistics. Briefly, the approach is (1) Assume that we know the unnormalized state probabilities for a population vector of $N-e_k$. (2) Use the Marginal Local Balance equation to compute the unnormalized marginal probability of queue $m$ being in state $S_m$ given a population $N$, for all states $S_m$ for which $S_m$ – $(c)$ is defined. (3) Compute the unnormalized mean queue length given a population $N$ from the unnormalized marginal probabilities given population $N$. (4) Since the sum of the mean number of jobs in chain $k$ over all queues must be $N_k$, it follows that the sum of the unnormalized mean number of chain $k$ jobs over all queues must be $G(N)N_k$. Hence compute the normalizing constant $G(N)$. (5) Normalized probabilities and mean queue lengths are obtained by dividing the unnormalized values by $G(N)$. Throughputs are obtained from the Throughput Theorem and mean queueing times from Little's Rule. (We will discuss algorithms using this approach in detail in Section 5.7.)

### 5.5.3 Marginal Probabilities of Subsystems

It is helpful to partition the set of queues 1,...,$M$ of the network into subsystems. For example, a model of a computer system may be partitioned into the processor subsystem and the I/O subsystem. Assume that the network is partitioned into $K$ subsystems: $\text{SUB}_1,...,\text{SUB}_K$. Each queue (and its classes) belongs to exactly one subsystem. Let $Z_1,...,Z_K$ be population vectors and let $P(Z_1,...,Z_K)$ be the equilibrium probability that the population of subsystem $i$ is $Z_i$, $i = 1,...,K$ given that the network population is $N = Z_1 + ... + Z_K$. Let $G_i(Z_i)$ be the normalization constant for a network containing only subsystem $i$ with a population of $Z_i$. (We may think of this network as being obtained by setting mean service times for all classes not in subsystem $i$ to zero.) In other words $G_i(Z_i)$ is the normalization constant for a network consisting of subsystem $i$ *alone*, when the population of this network is $Z_i$. Formally, if subsystem $i$ consists of queues $i(1),...,i(q)$, then:

$$G_i(Z_i) = \sum P(S_{i(1)})...P(S_{i(q)}) \tag{5.36}$$

where the summation is taken over all feasible states $S_{i(1)},...,S_{i(q)}$ of the queues in isolation, and where:

$$\text{POP}(S_{i(1)}) + ... + \text{POP}(S_{i(q)}) = Z_i$$

**Subsystem Lemma:**

$$P(Z_1,\ldots,Z_K) = \frac{G_1(Z_1)\ldots G_K(Z_K)}{G(N)} \tag{5.37}$$

The proof follows directly from the definition of $G(N)$ and $G_i(Z_i)$ by summing state probabilities over subsystems.

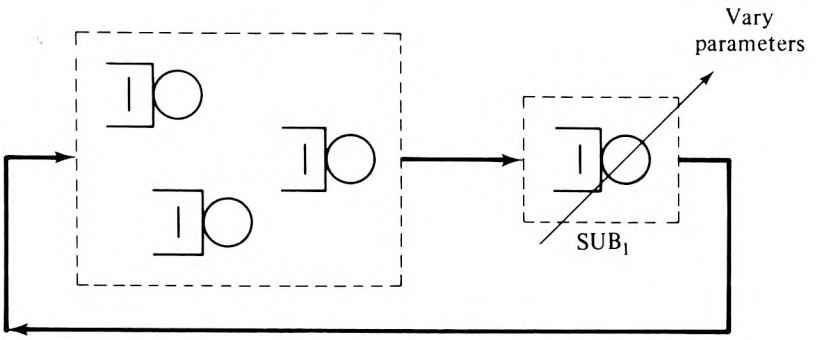



Figure 5.9

Suppose we wish to repeatedly analyze a network while we vary parameters in some subsystem $SUB_1$ as shown in Figure 5.9. The Aggregation Theorem helps us to carry out a parametric analysis of $SUB_1$ very easily.

**The Aggregation (Decomposition) Theorem (Norton's Theorem)**

For the purpose of computing statistics about $SUB_1$, all the queues, except those in $SUB_1$, can be replaced by a single composite queue whose rate matrix $H$ is computed in the following way. Let all the queues in the rest of the network (i.e., not in $SUB_1$) belong to $SUB_2$. For any population vector $Z_2$, define the matrix $H$ as:

$$H(Z_2) = G_2(Z_2)$$

where $G_2(Z_2)$ is defined in equation (5.36).

**Proof:**

The subsystem lemma states that $P(Z_1,Z_2)$ is proportional to $G_1(Z_1)G_2(Z_2)$, regardless of what $SUB_1$ is, provided the network is a local balance network. If we replace $SUB_2$ by a composite queue with rate

matrix $H = G_2$, we get the same values for $P(Z_1,Z_2)$ because the composite queue satisfies local balance, and the subsystem lemma states that $P(Z_1,Z_2)$ is proportional to $G_1(Z_1)G_2(Z_2)$.

This theorem is also referred to as a decomposition theorem and/or as Norton's Theorem. We have described the *aggregation* of queues into subsystems, but we could have equivalently described the *decomposition* of subsystems into queues. The description used is partly a matter of taste and partly dependent on particular situations; in Section 6.3 we will be focusing on decomposition. The Aggregation Theorem is analogous to Norton's Theorem for electrical circuits [CHAN75a].

### 5.5.4 Common Local Balance Disciplines

We next study the more useful local balance disciplines in detail. We shall restrict attention to the probability that there are $n_c$ jobs in class $c$, $c = 1,...,C$. Let us first consider a PS discipline where the capacity of the server varies with the number of jobs in the queue. The *total* rate at which class $c$ jobs are serviced when there are $n_c$ jobs of class $c$, $c = 1,...,C$ is $a_c \text{CAP}(n) n_c/n$, where $n = n_1 + ... + n_c$ is the total number of jobs in the queue; CAP($n$) is said to be the capacity of the queue when the queue length is $n$. We assume that the rates are normalized so that CAP(1) = 1. Our notation is simplified if we assume CAP(0) = 0.

If there is only one processor CAP($n$) = 1 for all positive $n$. If we want to model overhead in job switching, we may want CAP($n$) to decrease with $n$. In the infinite server case CAP($n$) = $n$. Define SHARE($n$) = CAP($n$)/$n$. SHARE($n$) is the fraction of processing power given to each of the $n$ jobs when there are $n$ jobs in the queue.

Since LCFSPR has the same queue length distribution as PS (see Chapter 4), we shall not continue to discuss LCFSPR separately. The following results for PS hold for LCFSPR and the special FCFS cases as well. The results hold for IS by suitably defining SHARE($n$).

For the PS case, the Marginal Local Balance equation (5.32) becomes, after simplification

$$p_{(m)}(\bar{n} \mid N) = \frac{p_{(m)}(\bar{n} - e_c \mid N - e_k) u_c}{n_c \text{ SHARE}(n)} \tag{5.38}$$

where $u_c = r_c/a_c$ and where $\bar{n}$ is defined as $n_1,...,n_c$, $n = n_1 + ... + n_c$, and class $c$ belongs to chain $k$. Recall that $p_{(m)}(n \mid N)$ is the unnormalized probability of $n$ jobs (regardless of class) in queue $m$ given population

vector $N$. Hence

$$p_{(m)}(n \mid N) = \sum_{\text{feasible } \bar{n}} p_{(m)}(\bar{n} \mid N) \tag{5.39}$$

where the summation is taken over all non-negative integral values of $n_1,\ldots,n_c$ such that

$$n_1 + \ldots + n_c = n. \tag{5.40}$$

Define $l_c(N)$ as the unnormalized mean queue length at class $c$ given population $N$, i.e.,

$$l_c(N) = L_c(N)G(N). \tag{5.41}$$

Then

$$l_c(N) = \sum_{\text{feasible } \bar{n}} n_c p_{(m)}(\bar{n} \mid N) \tag{5.42}$$

where the summation is taken over all non-negative integer values of $n_1,\ldots,n_c$ where (5.40) holds and

$$n_c \geq 1. \tag{5.43}$$

From equations (5.38) to (5.43), the unnormalized mean queue length at class $c$ is

$$l_c(N) = u_c \sum_{n=1}^{|N|} \frac{p_{(m)}(n-1 \mid N - e_k)}{\text{SHARE}(n)}, \tag{5.44}$$

where $|N| = N_1 + \ldots + N_k$ is the total job population over all chains, and from the Throughput Theorem the normalized queue length at class $c$ is

$$L_c(N) = U_c(N) \sum_{n=1}^{|N|} \frac{P_{(m)}(n-1 \mid N-e_k)}{\text{SHARE}(n)} \tag{5.45}$$

where

$$U_c(N) = \frac{R_c(N)}{a_c}. \tag{5.46}$$

Note that this $U$ is related to, but not the same as, the $U$ of the previous chapters. This $U$ would have to be divided by the number of servers to be consistent with the previous usage. After this section we will return to the

previous definition of $U$. In the *infinite server* case, SHARE($n$) = 1, hence from equation (5.45)

$$L_c(N) = U_c(N) \tag{5.47}$$

and

$$l_c(N) = u_c G(N - e_k). \tag{5.48}$$

In the *single server* case, SHARE($n$) = $1/n$, and simplifying equation (5.44)

$$L_c(N) = U_c(N)(1 + L_{(m)}(N - e_k)) \tag{5.49}$$

where $L_{(m)}(N - e_k)$ is the mean queue length of queue $m$, and

$$l_c(N) = u_c(G(N - e_k) + l_{(m)}(N - e_k)). \tag{5.50}$$

There is no closed form expression for the mean number of class $c$ jobs for a composite queue with an arbitrary rate matrix. If we are given the unnormalized probabilities for a population vector $N - e_k$, we use the Marginal Local Balance equation (5.32) to compute the unnormalized probabilities for a population vector $N$, and then compute unnormalized mean queue lengths from the unnormalized state probabilities.

Given the unnormalized mean queue lengths we can compute the normalization constant from

$$\sum_{c \text{ in } k} L_c(N) = N_k. \tag{5.51}$$

It follows from this equation that

$$G(N) = \frac{\sum_{c \text{ in } k} l_c(N)}{N_k}. \tag{5.52}$$

These equations suggest an iterative approach to computing normalizing constants. We know that $G(0) = 1$. Assume we have normalizing constants and other statistics for all population vectors less than $N$. Compute the unnormalized mean number of class $c$ jobs from the Marginal Local Balance equations and use equation (5.52) to compute $G(N)$. We will describe this approach in detail in Section 5.7.

Another set of equations is useful in determining $G(N)$ for closed networks with many (e.g., 50 or more) single or infinite server queues, especially when memory is severely constrained (e.g., when using program-

mable calculators). For notational simplicity we shall restrict attention to the 2 chain case. For chain $k$, $k = 1,2$, and queue $m$ let $u_{(k,m)} = \Sigma u_c$, for $c$ in chain $k$ and queue $m$ where $u_c = r_c/a_c$. Similarly, let $l_{(k,m)} = \Sigma l_c$. For fixed rate single server queues from equation (5.50)

$$l_{(1,m)}(N) = u_{(1,m)}(G(N_1 - 1,N_2) + l_{(m)}(N_1 - 1,N_2)) \quad (5.53)$$

and

$$l_{(2,m)}(N) = u_{(2,m)}(G(N_1,N_2 - 1) + l_{(m)}(N_1,N_2 - 1)). \quad (5.54)$$

For infinite server, queues from equation (5.48),

$$l_{(1,m)}(N) = u_{(1,m)}G(N_1 - 1,N_2) \quad (5.55)$$

and

$$l_{(2,m)}(N) = u_{(2,m)}G(N_1,N_2 - 1). \quad (5.56)$$

Let us assume that single server fixed rate queues are numbered from 1 to $I$ and that infinite server queues are numbered from $I+1$ to $J$. Define $\mathrm{CUM}(n_1,n_2)$ as follows (CUM is an abbreviation for cumulative):

$$\mathrm{CUM}(1,0) = \sum_{m=1}^{J} u_{(1,m)}, \quad (5.57)$$

$$\mathrm{CUM}(0,1) = \sum_{m=1}^{J} u_{(2,m)}, \quad (5.58)$$

and

$$\mathrm{CUM}(n_1,n_2) = \frac{(n_1 + n_2)!}{n_1!n_2!} \sum_{m=1}^{I} u_{(1,m)}^{n_1} u_{(2,m)}^{n_2} \text{ for } n_1 + n_2 > 1. \quad (5.59)$$

**Aggregate Queue Theorem:**

For $(N_1,N_2) \neq (0,0)$

$$G(N_1,N_2) = \sum_{(n_1,n_2)\neq(0,0)}^{N_1,N_2} \frac{G(N_1-n_1,N_2-n_2)\mathrm{CUM}(n_1,n_2)}{N_1 + N_2}. \quad (5.60)$$

**Proof:**

For the purpose of the proof we define a function $F_{(m)}(n_1,n_2)$ for each

queue $m$ as follows:

For a fixed rate server

$$F_{(m)}(n_1,n_2) = \begin{cases} \dfrac{(n_1+n_2)!}{n_1!n_2!} u_{(1,m)}^{n_1} u_{(2,m)}^{n_2} & \text{for } n_1 \geq 0,\ n_2 \geq 0 \\ 0 & \text{elsewhere} \end{cases}$$

and for an infinite server

$$F_{(m)}(n_1,n_2) = \begin{cases} u_{(1,m)} & \text{if } n_1 = 1,\ n_2 = 0 \\ u_{(2,m)} & \text{if } n_1 = 0,\ n_2 = 1 \\ 0 & \text{elsewhere.} \end{cases}$$

We shall now show by induction that

$$l_{(m)}(N_1,N_2) = \sum_{n_1,n_2 \neq 0,0}^{N_1,N_2} G(N_1-n_1,N_2-n_2) F_{(m)}(n_1,n_2)$$

for fixed rate and infinite servers.

This equation is obviously true for $(N_1,N_2) = (0,0)$, $(0,1)$ and $(1,0)$. Assume the equation is true for all $(K_1,K_2)$ such that either (1) $K_1 < N_1$ and $K_2 \leq N_2$ or (2) $K_1 \leq N_1$ and $K_2 < N_2$. We shall now prove the equation for $(K_1,K_2) = (N_1,N_2)$.

For a fixed rate server, from equations (5.53) and (5.54) and applying the induction assumption,

$$l_{(m)}(N_1,N_2)$$

$$= u_{(1,m)} G(N_1-1,N_2)$$

$$+ \sum_{n_1,n_2 \neq 0,0}^{N_1-1,N_2} G(N_1-1-n_1,N_2-n_2) u_{(1,m)} F_{(m)}(n_1,n_2)$$

$$+ u_{(2,m)} G(N_1, N_2 - 1)$$

$$+ \sum_{n_1,n_2 \neq 0,0}^{N_1,N_2-1} G(N_1 - n_1, N_2 - 1 - n_2) u_{(2,m)} F_{(m)}(n_1,n_2)$$

$$= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2} G(N_1 - 1 - n_1, N_2 - n_2) u_{(1,m)} F_{(m)}(n_1,n_2)$$

$$+ \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2-1} G(N_1 - n_1, N_2 - 1 - n_2) u_{(2,m)} F_{(m)}(n_1,n_2)$$

$$= \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2} G(N_1 - n_1, N_2 - n_2) u_{(1,m)} F_{(m)}(n_1 - 1, n_2)$$

$$+ \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2} G(N_1 - n_1, N_2 - n_2) u_{(2,m)} F_{(m)}(n_1, n_2 - 1)$$

$$= \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2} G(N_1 - n_1, N_2 - n_2)(u_{(1,m)} F_{(m)}(n_1 - 1, n_2) + u_{(2,m)} F_{(m)}(n_1, n_2 - 1))$$

$$= \sum_{n_1,n_2 \neq 0,0}^{N_1,N_2} G(N_1 - n_1, N_2 - n_2) F_{(m)}(n_1,n_2)$$

The very last step follows from the fact that for $(n_1,n_2) \neq (0,0)$

$$u_{(1,m)} F_{(m)}(n_1 - 1, n_2) + u_{(2,m)} F_{(m)}(n_1, n_2 - 1) = F_{(m)}(n_1,n_2).$$

When $(n_1,n_2) = (0,0)$ the left hand side of the above equation is zero (0) while the right hand side is unity (1). The proof of the above equation for $l_{(m)}(N_1,N_2)$ for the IS case follows trivially from equations (5.55) and (5.56).

The theorem follows from the fact that

$$\sum_m l_{(m)}(N_1,N_2) = (N_1 + N_2)G(N_1,N_2)$$

and

$$\sum_m F_{(m)}(N_1,N_2) = \mathrm{CUM}(N_1,N_2).$$

**Convolution**

Consider a closed network with $K$ chains consisting of 2 subsystems: 1 and 2. Assume that all queues in the network are either processor-sharing queues, with service rates which may vary with queue length, or composite queues. Let $G_1(N)$, $G_2(N)$ and $G(N)$ be the normalization constants for subsystem 1, subsystem 2 and the entire system (respectively) given a population vector of $N$. From the subsystem lemma we know that the probability of $\bar{n}_i$ jobs in subsystem $i$, $i = 1,2$, (where $\bar{n}_i$ is a vector of length $K$) given a population vector $\bar{n} = \bar{n}_1 + \bar{n}_2$ for the system is

$$P(\bar{n}_1,\bar{n}_2 \mid \bar{n}) = \frac{G_1(\bar{n}_1)G_2(\bar{n}_2)}{G(\bar{n})}.$$

Summing over all $\bar{n}_1$, $\bar{n}_2$, we have

$$\sum_{\bar{n}_1=0}^{\bar{n}} P(\bar{n}_1,\bar{n}-\bar{n}_1 \mid \bar{n}) = 1$$

Hence

$$G(\bar{n}) = \sum_{\bar{n}_1=\bar{0}}^{\bar{n}} G_1(\bar{n}_1)G_2(\bar{n} - \bar{n}_1) \tag{5.61}$$

Let $G$ be an $(N_1 + 1)\times\ldots\times(N_K + 1)$ matrix of $G(\bar{n})$, with $\bar{n}_k$ varying from 0 to $N_k + 1$, where $N_k$ are arbitrary non-negative integers for all $k$. $G_1$ and $G_2$ are defined similarly. We shall write

$$G = G_1 * G_2 \tag{5.62}$$

and * is called the convolution operation.

**Convolution Theorem:**

Consider a network consisting of $M$ subsystems, and let $G_m(\bar{n})$ be the normalization constant for the $m^{th}$ subsystem given a population vector $\bar{n}$,

$m = 1,...,M$. Let $G_m$ be a matrix of $G_m(\bar{n})$ whose indices range from $(0,...,0)$ to $(N_1,...,N_K)$, $m = 1,...,M$. Let $G(\bar{n})$ be the normalization constant for the entire network given population vector $\bar{n}$, and let $G$ be a matrix of $G(\bar{n})$, with indices ranging from $(0,...,0)$ to $(N_1,...,N_K)$.

$$G = G_1 * ... * G_M. \tag{5.63}$$

**Proof:**

This result follows by induction on the number of subsystems, since the arguments given earlier show that the theorem holds for a network consisting of two subsystems. Note that the convolution operation is both associative and commutative.

**Deleting a queue**

We now discuss the computation of normalization constants for a network consisting of queues $1,...,M-1$ from the normalization constants for a network consisting of queues $1,...,M$.

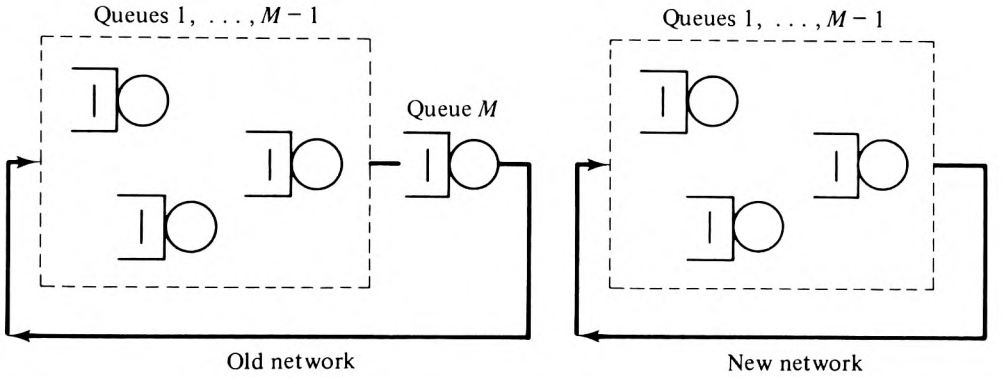



Figure 5.10

Consider a network with queues $1,...,M$. Let $G_{\text{OLD}}(\bar{n})$ be the normalization constant for this network given a population vector $\bar{n}$. Let $G_{\text{NEW}}(\bar{n})$ be the normalization constant for the network with queue $M$ removed (e.g., shorted out). From the subsystem lemma, the unnormalized probability of $n_k$ jobs of chain $k$, all $k$, in queue $M$ (in the OLD system), given a population vector $N$ is

$$p_{(M)}(\bar{n} \mid N) = X_{(M)}(\bar{n}) G_{\text{NEW}}(N - \bar{n}) \tag{5.64}$$

where $X_{(M)}(\bar{n})$ is the normalizing constant for a network consisting of queue $M$ by itself. From this equation, we have

$$p_{(M)}(0 \mid N) = G_{NEW}(N)$$

and

$$p_{(M)}(\bar{n} \mid N) = X_{(M)}(\bar{n}) p_{(M)}(0 \mid N) \tag{5.65}$$

We shall use equation (5.64) in computing $G_{NEW}$.

We now derive equations for the special case where queue $M$ is a fixed rate server. For notational simplicity we now consider a 2 chain case. The Marginal Local Balance equation can be written as

$$p_{(m)}(n_1,n_2 \mid N_1,N_2) = \begin{cases} p_{(m)}(n_1-1,n_2 \mid N_1-1,N_2) u_{(1,m)} \dfrac{n_1+n_2}{n_1}, & n_1 \neq 0, \\ p_{(m)}(n_1,n_2-1 \mid N_1,N_2-1) u_{(2,m)} \dfrac{n_1+n_2}{n_2}, & n_2 \neq 0. \end{cases}$$

From this equation, for $(n_1,n_2) \neq (0,0)$ we have

$$p_{(m)}(n_1-1,n_2 \mid N_1-1,N_2) u_{(1,m)} + p_{(m)}(n_1,n_2-1 \mid N_1,N_2-1) u_{(2,m)}$$

$$= p_{(m)}(n_1,n_2 \mid N_1,N_2)$$

where we define $p_{(m)}(i,j \mid k,l)$ to be 0 if $i$, $j$, $k$, or $l$ is negative. Summing the above equation over all $n_1,n_2$ we have

$$G_{\mathrm{OLD}}(N_1-1,N_2) u_{(1,m)} + G_{\mathrm{OLD}}(N_1,N_2-1) u_{(2,m)}$$

$$= G_{\mathrm{OLD}}(N_1,N_2) - p_{(m)}(0,0 \mid N_1,N_2)$$

where $G_{\mathrm{OLD}}$ is the normalizing matrix for the entire network, i.e., for queues $1,\ldots,M$. Recall that $G_{\mathrm{NEW}}$ is the matrix of normalizing constants for a network obtained by deleting queue $M$. Hence

$$G_{\text{NEW}}(N_1,N_2)$$

$$= p_{(m)}(0,0 \mid N_1,N_2)$$

$$= G_{\text{OLD}}(N_1,N_2)$$

$$- G_{\text{OLD}}(N_1-1,N_1)u_{(1,m)} - G_{\text{OLD}}(N_1,N_2-1)u_{(2,m)} \qquad (5.66)$$

For a fixed rate queue, we know that the normalization constant for a matrix consisting of queue $M$ by itself is

$$X_{(M)}(N_1,N_2) = \frac{(N_1+N_2)!}{N_1!N_2!} u_{(1,m)}^{N_1} u_{(2,m)}^{N_2}$$

We can use equations (5.65) and (5.66) to compute the marginal probability for queue $M$ if it is a fixed rate queue.

## 5.6 AN INTRODUCTION TO CLOSED NETWORKS

The reader who has completed Sections 5.1 through 5.5 may skip this section and go directly to Section 5.7. Some of the material of those sections is repeated in this section before we discuss computational algorithms. Readers who have skipped the earlier sections will find a nontheoretical introduction in this section.

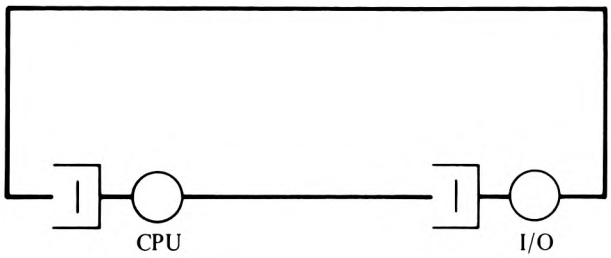



Figure 5.11 A simple closed network

Jobs neither enter nor leave a closed network. There is a constant number of jobs in a closed network at all times. This number is called the job *population* of the network. Figure 5.11 is an example of a simple two queue closed network model where one queue represents a CPU and the other a disk. In this model, when a job finishes service at the CPU it joins the disk queue, and when a job finishes service at the disk it joins the CPU

queue. The total number of jobs in the network (and hence the sum of the queue lengths of all jobs in the network) must always equal the population, $N$.

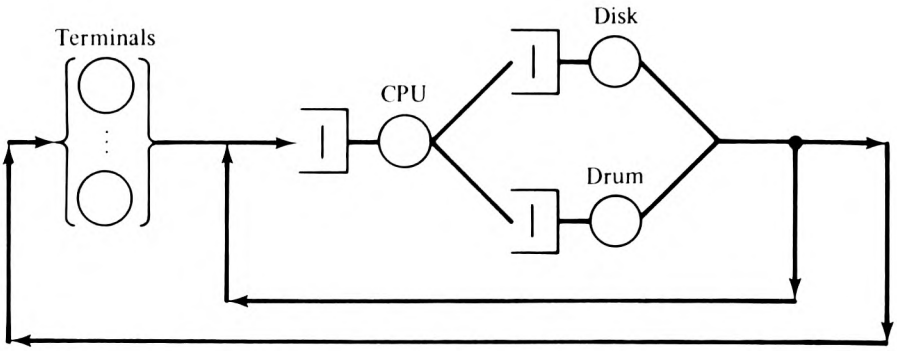



Figure 5.12 A simple model of a time-sharing system

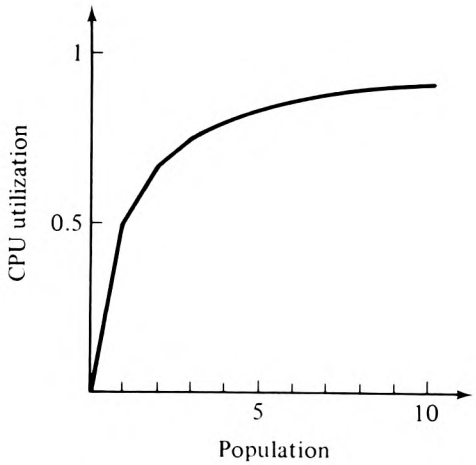



Figure 5.13 CPU utilization as a function of population

In open networks, any queue length can range from zero to infinity and no restriction is placed on the total number of jobs in the network. In most computing systems, the total number of jobs in device queues is limited by the amount of memory, or the number of control points or some other resource. For example, in a time-sharing system with $N$ users, there must be a total of $N$ jobs in the system, where the system includes the terminals as well. (Here we assume that there is at most one job per user and that there are no additional "system" jobs. Though neither assumption is necessarily correct, the statement that the total number of jobs is fixed is essen-

tially true.) In practice, $N$ is not truly constant over all time; for example, in a time-sharing system there may be 10 users for 10 minutes, and then 9 users for the next 5 minutes, and then 10 users for the next minute, and so on. It is preferable to model such a system as a closed network with a fixed population $N$, and study the behavior of the model for different values of $N$. For instance, we might want to determine the CPU utilization as a function of $N$. In computing such a function, we are really analyzing several *equilibrium* models: one in which $N = 1$, another in which $N = 2$, and so on. Strictly speaking, we should only have a single model which incorporates the *transient* behavior of $N$, i.e., the way $N$ changes with time; however, such models are difficult to analyze and we choose to approximate the true model by one in which the system is assumed to switch from one equilibrium model (and value of $N$) to another. If we wished to determine the CPU utilization for a period of time during which $N = 9$ for 50% of the time and $N = 10$ for 50% of the time, we could do so (approximately) by averaging the CPU utilizations in equilibrium closed models in which $N = 9$ and $N = 10$.

The previous discussion should convince the reader that closed network models are more appropriate than open network models for a variety of systems. There is another basic difference between open and closed models: the computation of queue throughput (the rate at which jobs complete service at the queue). As shown in Chapter 4, the throughput of a queue in an open network depends only on the rates at which the sources produce jobs and the expected number of visits that jobs make to a queue; in particular, the throughputs are independent of the service times, provided, of course, that equilibrium conditions exist. An inspection of Figure 5.11 will show that the throughput of the CPU queue must decrease with increasing CPU and disk service times. Queue throughputs in closed networks depend on *service times* and *branching probabilities* (the probability of making a transition from queue $i$ to queue $j$). There seems to be no simple way of computing throughputs in closed networks. However, it is possible to compute the throughput of one queue relative to the throughput of another quite easily as will be shown in the next paragraph; for instance, the throughput of the CPU queue in Figure 5.11 must equal the throughput of the disk queue.

We now restrict attention to closed, product form networks in which a job of any class $i$ can become a job of any other class $j$ (possibly after making transitions through other classes $k_1$, $k_2$, $k_3$,...) with non-zero probability; such networks are called *single-chain networks*. Consider the closed networks of Figure 5.14. There are 4 classes in each network. In Figure 5.14a, a job in any class $i$ can eventually become a job in any class $j$; for example, a job of class 2 can become a job of class 1 after passing through classes 3, 4 and 1. In Figure 5.14b, a job in class 1 can become a job in

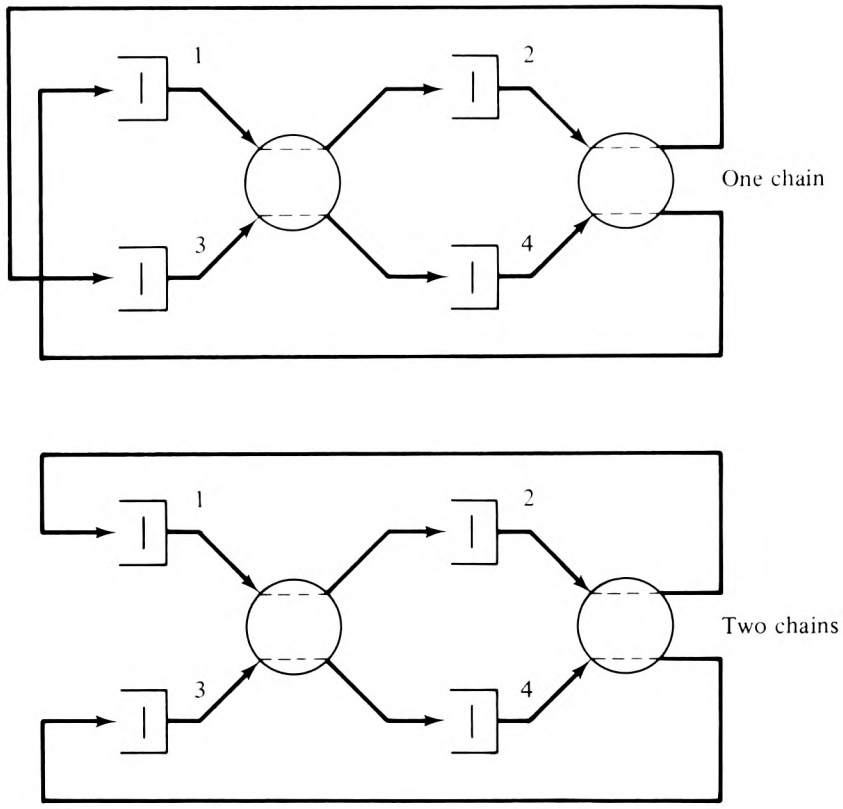



Figure 5.14 Networks with one and two chains

class 2 but can never join classes 3 and 4. We will consider networks of the form shown in Figure 5.14b (which are called multiple-chain networks) later in this chapter, but we shall ignore them for the rest of this section.

The notation used in this chapter is the same as that used in Chapter 4. It is reviewed here. Let there be $C$ classes. Let $R_c = Rr_c$ be the throughput of class $c$. Let $p_{cd}$ be the probability that a job leaving class $c$ immediately joins class $d$. Then we must have

$$R_C = \sum_{d=1}^{C} R_d p_{dc}, \; c = 1,...,C. \tag{5.67}$$

This equation states that the rate of flow of jobs into class $c$ must equal the rate of flow out of class $c$. Equation (5.67) is a system of $C$ equations, but one of the equations is linearly dependent on the other $C-1$ because the equation for any class $c$ can be obtained by summing the equations for all other classes (and then simplifying). Hence, if we knew

the throughput of any class, we could use (5.67) to find the throughput of all the other classes because we would have only $C-1$ unknowns and $C-1$ independent equations.

Let the *relative throughputs*, $r_c$, $c = 1,...,C$ be a set of numbers such that

$$r_c = DR_C\ ,\ c = 1,...,C, \tag{5.68}$$

where $D$ is any positive constant. Note that

$$\frac{R_c}{R_d} = \frac{r_c}{r_d}\ , \tag{5.69}$$

i.e., the ratio of throughputs is equal to the ratio of relative throughputs. Substituting (5.68) in (5.67) the set of $r_c$ must satisfy

$$r_c = \sum_{d=1}^{C} r_d p_{dc},\ c = 1,...,C. \tag{5.70}$$

Let $n_c$ be the number of class $c$ jobs, $c = 1,...,C$. The $n_c$ are random variables, but at all times

$$\sum_{c=1}^{C} n_c = N \tag{5.71}$$

where $N$ is the population. Taking the means of both sides of (5.71)

$$\sum_{c=1}^{C} L_c = N \tag{5.72}$$

where $L_c$ is the mean number of jobs in class $c$, i.e., $L_c$ is the mean queue length at class $c$. Let $Q_c$ be the mean queueing time at class $c$. From Little's Rule

$$Q_c = \frac{L_c}{R_c}. \tag{5.73}$$

Note that equations (5.67) through (5.73) hold regardless of the service disciplines and distributions. (These results hold for networks without product form solution.) We next consider results that only hold for special distributions or disciplines.

Consider the network of Figure 5.11. Refer to the CPU queue as queue 1 and to the disk queue as queue 2. All jobs in queue $c$ belong to

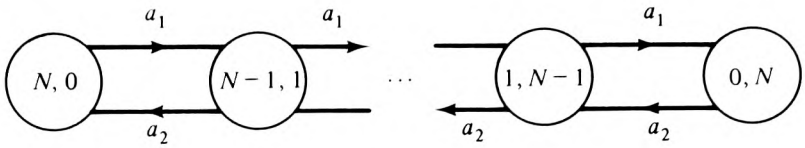



Figure 5.15 The Markov diagram for the network of Figure 5.11

class $c$, $c = 1,2$. Let $a_c$ be the service rate of class $c$, $c = 1,2$, i.e., $1/a_c$ is the mean service time at class $c$. Assume that all service times are independent exponential random variables. Let $n_c$ be the number of jobs in class $c$, and let $N$ be the total job population. The states for the Markov process for this network are $(n_1,n_2)$ where $n_1 + n_2 = N$, as shown in Figure 5.15. Note the similarity of this diagram to that of Figure 4.2. Solving the balance equations exactly as we did for the example in Figure 4.2 (the reader should perform the solution) we get

$$P(n_1,n_2) = P(0,N)\rho^{n_1}\ ,\ n_1 = 0,...N,\ n_1 = N - n_1, \tag{5.74}$$

where $\rho = a_2/a_1$ and

$$P(0,N) = \frac{1}{1 + \rho + \rho^2 + ... + \rho^N}. \tag{5.75}$$

The utilization of class 1 is $1 - P(0,N)$. Note that if we interpret $a_2$ as $R$ for the isolated queue of Section 4.1 and $\rho$ as $U$, with $\rho < 1$, the state probabilities $P(n,N - n)$ for the closed network tend to the $P(n)$ of equation 4.6, as the population $N$ gets arbitrarily large. As $N$ gets arbitrarily large, the queue length at the disk (queue 2) will get arbitrarily large since it is the slowest queue in the network, because $\rho$ is assumed to be less than 1. In the limit the utilization of queue 2 will tend to 1, in which case the inter-arrival time at queue 1 will be the service time of queue 2. Thus, in the limit, the arrival process to queue 1 will be Poisson at rate $a_2$, which allows us to use an open network model to analyze queue 1. The same analysis can be applied to general networks. Let $\mathscr{I}$ be the set of queues with the maximum value of relative utilization, i.e., where

$$u_{(m)} = \sum_{c \text{ in } \mathscr{C}_m} u_c = \sum_{c \text{ in } \mathscr{C}_m} \frac{r_c}{a_c}.$$

let

$$U_{\text{MAX}} = \text{MAX } \{u_{(m)}\}$$

and

$$\mathcal{I} = \{m \mid u_{(m)} = U_{\text{MAX}}\}$$

For any population $N$, a queue in $\mathcal{I}$ will have a greater utilization than a queue not in $\mathcal{I}$ because the ratio of queue utilizations is equal to the ratio of relative queue utilizations. As the population tends to infinity, the utilization of queues in $\mathcal{I}$ will approach 1. Hence, departures from these queues will become Poisson with rates equal to the queue service rates. In this case we can analyze every queue not in $\mathcal{I}$ by analyzing an open

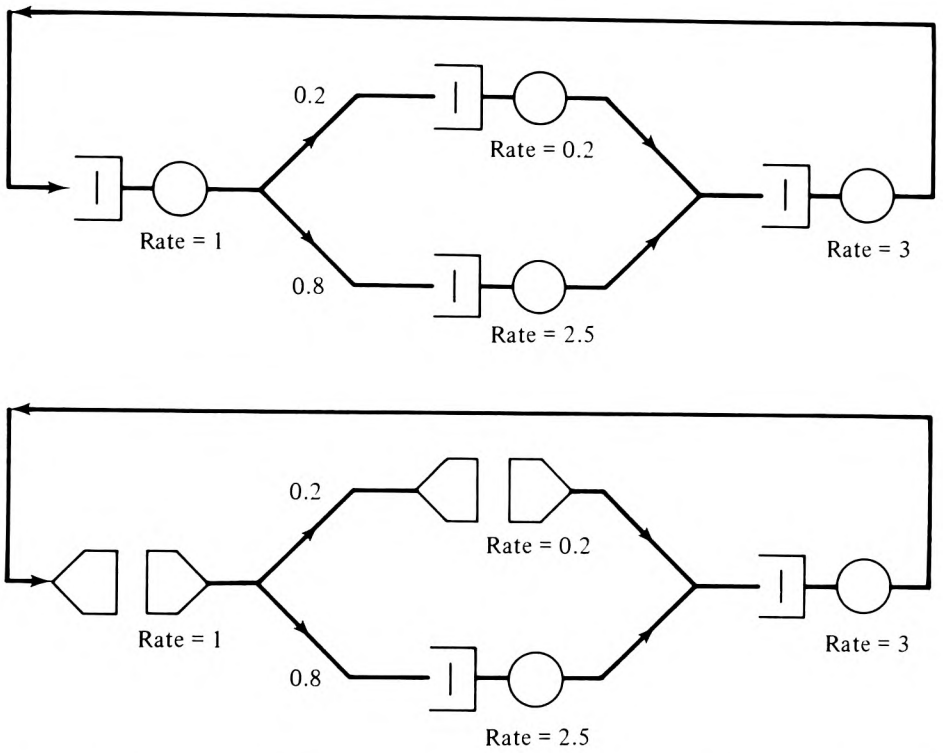



Figure 5.16 Creating the asymptotic open network from a closed one

network, which is obtained from the given closed network by replacing every queue in $\mathcal{I}$ by a source and sink pair, as shown in Figure 5.16.

Let us examine the relationship between the CPU (queue 1) utilization and job population more closely. Computing the utilization as discussed earlier, we get the graph shown in Figure 5.17. With $\rho < 1$, the CPU utilization tends to $\rho$ as $N$ tends to infinity for the reasons given earlier.

With $\rho > 1$, the CPU utilization tends to 1 as $N$ tends to infinity. When $\rho$ is much smaller or much larger than 1 the CPU utilization rises sharply and then flattens out for relatively small values of $N$, whereas the curve rises in a gentler fashion when $\rho$ is approximately 1. The reason for this is seen by examining the mean queue length of either queue as a function of the population. When $\rho = 1$, the queues are symmetric and the mean queue length must be $N/2$ for each queue. When $\rho$ is much greater than 1, each additional unit increase in $N$ will increase the mean queue length of the heavily utilized queue (queue 1) greater than the less utilized queue, until, in the limit $L_1$ asymptotically approaches a 45° line. When $\rho < 1$, each additional job added to the system spends more time in the more heavily utilized disk queue, till in the limit the mean CPU queue length approaches a constant value.

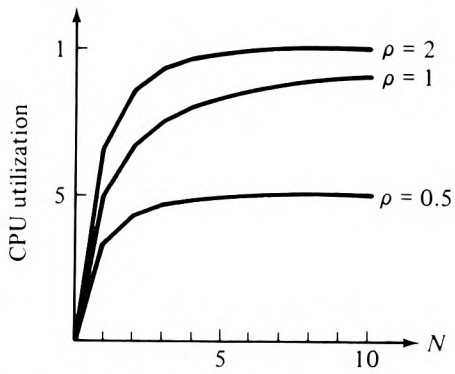



Figure 5.17 CPU utilization as a function of $N$

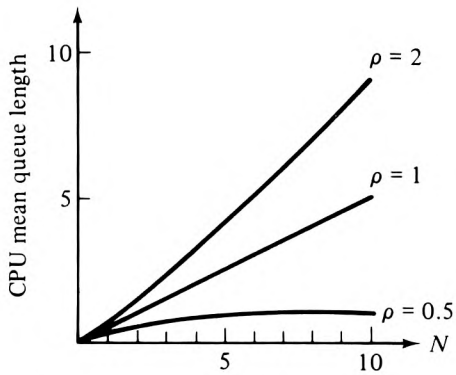



Figure 5.18 Mean queue length as a function of population

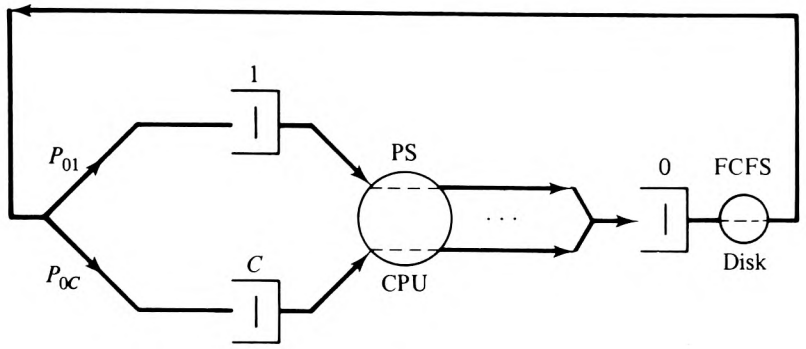



Figure 5.19

We next consider networks in which some of the queues have PS, LCFSPR or IS disciplines. All classes belonging to a queue with a FCFS discipline are assumed to have the same exponential service distribution. Consider, for example, a simple extension of Figure 5.11, shown in Figure 5.19. The notation used here is designed to fit the open network in Figure 4.7 with the disk representing the source. Let $a_c$ be the service rate of class $c$, $c = 1,...,C$ and let $R$ be the service rate of the disk (class 0).

Let $C = 2$, let $n_c$ be the number of jobs in class $c$ and let $(n_1,n_2)$ be the state of the network. The number of jobs at the disk is $N - (n_1 + n_2)$. The Markov diagram for this closed network is the same as that for the open network (see Figure 4.8) with infeasible states (those with $n_1 + n_2 > N$) removed. Calculation (as in equations (4.41-4.48)) shows that the equilibrium state probabilities are given by equation (4.42), and the probability of $n$ jobs at the disk and $N - n$ in the CPU is given by (4.45). The mean CPU service time is

$$\frac{p_{01}}{a_1} + \frac{p_{02}}{a_2}.$$

Setting $\rho$ to be the ratio of the CPU service time to the disk service time, we get

$$\rho = R\left(\frac{P_{01}}{a_1} + \frac{P_{02}}{a_2}\right).$$

Hence, the probability of $n$ jobs in the CPU and $N - n$ jobs in the disk is exactly the same as for Figure 5.10. Thus the queue length distributions and other measures we have considered in detail are dependent only on mean service time and are independent of class distinctions.

It has been shown that only the relative utilizations are relevant in computing the probability that there are $n_m$ jobs in queue $m$, $m = 1,2,...,M$ in an arbitrary network provided that (1) at each FCFS queue all classes have the same exponential service time distribution, and (2) at non-FCFS queues the service time distributions are differentiable, and the queueing disciplines are processor sharing, last come first served preemptive resume, infinite servers or other members of a special set of disciplines defined in CHAN77b. (Most of these results are in Chapter 4 and Sections 5.1 through 5.5. The exponential stages characterization of distributions we have considered is slightly more restrictive than the differentiable characterization mentioned above and used in CHAN77b.) This class of networks satisfies product form, i.e.,

$$P(S_1,...,S_M) = \frac{P_1(S_1)...P_M(S_M)}{G} \tag{5.76}$$

where $P(S_1,...,S_M)$ is the probability of a feasible network state in a network of $M$ queues, where $P_m(S_m)$ is a factor reflecting the probability that queue $m$ is in state $S_m$ and $G$ is a normalizing constant. The queue states $S_m$ for queue $m$, and the functional form of the probabilities $P_m(S_m)$ are the same as in the case where class $c$ of queue $m$ is fed by Poisson arrivals with rate equal to the relative throughput of class $c$ in the closed network. (See Figure 5.1.) The derivation of this last result is found in Section 5.4.

## 5.7 COMPUTATIONAL ALGORITHMS

There are a number of criteria that must be considered in choosing a computational algorithm for queueing network models. These include generality, computational effort, storage requirements, numerical stability and implementation effort. There are four generic types of algorithms which we find interesting. In historical order, these are the Convolution Algorithm as first discussed in BUZE71 and most refined in REIS78b, the Mean Value Analysis Algorithm of REIS78a, and two algorithms we proposed in CHAN79, the Local Balance Algorithm for Normalizing Constants (LBANC) and the Algorithm to Coalesce Computation of Normalizing Constants (CCNC). LBANC was derived from Mean Value Analysis.

**Generality.** CCNC is restricted to networks with fixed rate single server queues and infinite server queues. (It can be augmented by the Convolution Algorithm to solve networks with other queues.) The Mean Value Analysis Algorithm and LBANC allow fixed rate single server queues, infinite server queues and variable service rate queues where the service rate depends only on the total queue length, e.g., multiple chain composite queues (Section 5.4 and Chapter 6) are not allowed. Mean Value Analysis

has not been applied to *mixed* networks, i.e., networks with both open and closed chains; it seems likely that Mean Value Analysis can be easily extended to mixed networks using the approach described for LBANC, but it is not clear whether the approach would work for Mean Value Analysis with variable rate queues. Convolution is the only algorithm which has been applied to the full class of product form networks.

**Computational Effort.** Computational effort is tied to storage requirements since one may choose to recompute values to save storage. We may think of the computational effort associated with Convolution, Mean Value Analysis and LBANC to be essentially the same, depending on implementations and which performance measures are obtained. (E.g., mean queueing times are necessarily obtained by Mean Value Analysis but optionally obtained by Convolution and LBANC.) For the single chain case, the effort of LBANC for networks without variable rate queues is roughly $3MN + N + M$ additions, multiplications and divisions, which compares favorably with Convolution [REIS76] and Mean Value Analysis [REIS78a]. Computational effort is usually not a problem except when we are dealing with many closed chains and/or large closed chain populations or when we are using very slow processors, e.g., programmable calculators. Approximate methods have been proposed based on Mean Value Analysis which have much lower computational requirements [REIS78a, BARD78]. Essentially the same approaches can be used with LBANC, as we will discuss in Chapter 6.

**Storage Requirements.** It is difficult to generalize about storage requirements, since the requirements are dependent on both the problem solved and the implementation. CCNC has much lower storage requirements than the other algorithms when the number of queues is large, assuming we save the intermediate result vectors of the Convolution Algorithm, rather than recompute them. In fact, its requirements are so low that multiple chain problems can be solved on a programmable calculator using this algorithm. However, it is of limited generality and has poorer numerical properties than the Mean Value Analysis Algorithm and LBANC, so it is only appropriate when storage is at a premium. The Mean Value Analysis Algorithm and LBANC have small storage requirements as long as variable rate queues and/or queue length distributions are not considered. (In the multiple chain case the obvious looping structures for these algorithms will require large amounts of storage, but alternate structures can be used to obtain reasonable storage requirements.) The Convolution Algorithm has by far the lowest storage requirements for general variable rate queues in multi-chain networks; depending on specific variations it may have smaller storage requirements when queue length distributions are estimated.

**Numerical Stability.** The Mean Value Analysis Algorithm has remarkable numerical stability for models with only fixed rate and infinite server queues; it is the algorithm of choice for such models when extreme parameter values are considered. (Mean Value Analysis, as originally defined [REIS78a], does not retain its numerical stability for variable rate queues. Modifications have been proposed which eliminate the numerical problem but may be quite expensive in computation and storage, depending on the number of variable rate queues in the network [REIS80].) Though LBANC is closely related to the Mean Value Analysis Algorithm it may fail for extremes of parameters which can be handled by Mean Value Analysis. The Convolution Algorithm has poorer numerical properties for fixed rate and infinite server queues than those two algorithms, *but has better numerical properties for variable rate queues* when the probability of small queue lengths at those queues is small. (It also has better numerical properties for queue length distributions.)

**Implementation Effort.** Mean Value Analysis, LBANC and CCNC are very simple to implement. The Convolution Algorithm is significantly more complex, particularly because of the intermediate values involved which have little intuitive relationship to the performance measures.

None of these algorithms is entirely satisfactory. However, LBANC supplemented by the Convolution Algorithm will be satisfactory under most circumstances. For extreme parameter values the Mean Value Analysis Algorithm will be the only satisfactory choice, but it may be expensive if variable rate queues are involved. Where storage is severely constrained CCNC may be useful. CCNC seems the best choice for programmable calculator implementations when there is a significant number of queues, while LBANC or Mean Value Analysis are preferred when there is a small number of queues and a large population(s). The best approach for general purpose use is a mixture of algorithms; we would suggest that LBANC be used for the portion of the network restricted to fixed rate single server and infinite server queues, and that the Convolution Algorithm be used to complete the computation.

In Section 5.7.1 we discuss, for single chain networks, LBANC, CCNC and then the Convolution Algorithm. (We defer discussion of the Mean Value Analysis Algorithm to Section 5.7.3 since it is so similar to LBANC.) In Section 5.7.2 we extend the discussion to multiple chains. In Section 5.7.3 we consider the numerical properties and requirements of all four algorithms.

### 5.7.1 Algorithms for Single Chain Networks

We shall first restrict attention to PS queues where the total service capacity varies with the number of jobs in the queue, since LCFSPR and FCFS queues which satisfy local balance have the same queue length distribution and the same values for performance measures obtainable from the queue length distribution. (In the multiple chain case there are queues which are not equivalent to this restricted characterization.)

As motivation for queue with capacities which vary with the number of jobs in the queue, consider a single queue, say queue $m$, with two servers. When there is only one job in queue $m$, only one of the two servers can be active, whereas when there are 2 or more jobs in the queue, both servers are active. (If there are two or more jobs in queue $m$, the two servers processor-share the jobs.) Let $a_c$ be the rate at which class $c$ jobs are served when there is only one class $c$ job in queue $m$. Then, when there are 2 or more class $c$ jobs in queue $m$, and if all jobs in queue $m$ are class $c$ jobs, we may expect the total service rate for class $c$ jobs to become $2a_c$, because both servers will be busy. However, the servers may interfere with one another. For example, two processors may not be able to work twice as fast as one because of memory interference, and it is possible that the service rate with 2 or more jobs may be an arbitrary positive number times the rate for one job.

Let $\text{SHARE}_{(m)}(n)$ be the fraction of processing power given to each job when there are $n$ jobs in the queue. In a single server, PS case, $\text{SHARE}_{(m)}(n) = 1/n$ for all positive $n$, and in the infinite server case $\text{SHARE}_{(m)}(n) = 1$ for all positive $n$. In the 2 processor case discussed above $\text{SHARE}_{(m)}(n) = 2/n$ for $n \geq 2$. The service rate for a *single*, specific class $c$ job, when there are a total of $n$ jobs in queue $m$ is $a_c\text{SHARE}_{(m)}(n)$. The total service rate for *all* class $c$ jobs when there are a total of $n$ jobs in queue $m$, $n_c$ of which belong to class $c$, is $a_c\text{SHARE}_{(m)}(n)n_c$. Define $\text{CAP}_{(m)}(n) = n\text{SHARE}_{(m)}(n)$. For a single server PS queue $\text{CAP}_{(m)}(n) = 1$ for positive $n$. For an IS queue $\text{CAP}_{(m)}(n) = n$ for positive $n$. For a two server PS queue $\text{CAP}_{(m)}(1) = 1$ and $\text{CAP}_{(m)}(n) = 2$ for $n \geq 2$.

#### LBANC

With the exception of the Mean Value Analysis Algorithm of REIS78a, all efficient algorithms for performance metrics require computation of the normalizing constant, $G$. LBANC iteratively applies the Marginal Local Balance equation (5.32) to obtain the normalizing constant. The normalization constant, mean queue length and other statistics depend on the population. We shall show this dependence explicitly by writing $G(n)$ and $L_{(m)}(n)$

for the normalization constant and mean queue length of queue $m$, respectively, given a population of $n$. Let $l_{(m)}(N)$ be the unnormalized mean queue length of queue $m$ given population N, where

$$l_{(m)}(N) = L_{(m)}(N)G(N) \tag{5.77}$$

The use of (5.32) in LBANC yields unnormalized mean queue lengths as intermediate results. Thus the mean queue lengths are immediately available once $G$ is determined. From the fact that $\sum_m L_{(m)}(n) = n$, we get

$$G(n) = \frac{\sum_m l_{(m)}(n)}{n}. \tag{5.78}$$

Applying local balance we have shown (equations (5.48) and (5.50)) that for fixed rate queues (i.e., $\text{CAP}_{(m)}(n) = 1$ for all positive $n$)

$$l_{(m)}(n) = u_{(m)}(G(n-1) + l_{(m)}(n-1)) \tag{5.79}$$

and for infinite server queues

$$l_{(m)}(n) = u_{(m)}G(n-1). \tag{5.80}$$

The queue length distribution is necessarily obtained for queue length dependent queues other than IS queues by LBANC. For queues other than single server and infinite server (and to obtain the queue length distribution for fixed rate and IS queues) we use equation (5.38),

$$p_{(m)}(i \mid n) = \frac{p_{(m)}(i-1 \mid n-1)u_{(m)}}{\text{CAP}_{(m)}(i)} \quad \text{for } i > 0, \tag{5.81}$$

to compute the queue-length distribution given population $n$, and then compute $l_{(m)}(n)$ from

$$l_{(m)}(n) = \sum_{i=1}^{n} i p_{(m)}(i \mid n). \tag{5.82}$$

We next compute

$$p_{(m)}(0 \mid n) = G(n) - \sum_{i=1}^{n} p_{(m)}(i \mid n). \tag{5.83}$$

LBANC is simply the application of these equations, $l_{(m)}(0) = 0$ and $p_{(m)}(0 \mid 0) = 1$. We can obtain throughputs directly from $G$ and the relative throughputs by the Throughput Theorem (5.29), i.e.,

$$R_{(m)} = r_{(m)} \frac{G(N-1)}{G(N)}.$$

The mean queueing times can then be obtained from Little's Rule. Utilizations can be obtained from (2.7) for most cases; in general it may be necessary to use the queue length distribution to determine utilizations, but this is still conceptually trivial.

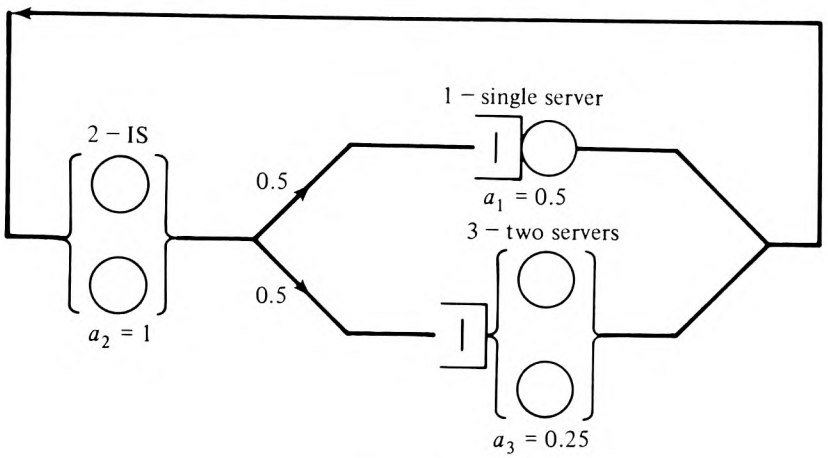



Figure 5.20 An Example

We will now give a more program-like definition of LBANC. Let us assume that queues 1,...,$I$ are single server fixed rate queues, that queues $I+1$,...,$J$ are IS queues and that queues $J+1$,...,$M$ have general CAP functions. We assume that we are primarily interested in metrics for population $N$ and will reuse variables along the way, i.e., we will drop the $n$ subscripts from all of the above variables, except $G$.

$G(0) = 1$
For $m$=1 to $M$
    $l_{(m)} = 0$
    $p_{(m)}(0) = 1$
For $n$=1 to $N$ {Iterate over populations}
    $G(n) = 0$
    For $m$=1 to $I$ {Fixed rate queues}
        $l_{(m)=}u_{(m)}(G(n-1) + l_{(m)})$
        $G(n) = G(n) + l_{(m)}$
    For $m$=$I$+1 to $J$ {Infinite server queues}

$l_{(m)} = u_{(m)}G(n-1)$
$G(n) = G(n) + l_{(m)}$
For $m=J+1$ to $M$ {Variable rate queues}
$l_{(m)} = 0$
For $i=n$ down to 1 {Iterate over queue lengths}
$p_{(m)}(i) = p_{(m)}(i-1)u_{(m)}/\text{CAP}_{(m)}(i)$
$l_{(m)} = l_{(m)} + i \times p_{(m)}(i)$
$G(n) = G(n) + l_{(m)}$
$G(n) = G(n)/n$ {Finished computing $G(n)$}
For $m=J+1$ to $M$ {Variable rate queues}
$p_{(m)}(0) = G(n)$
For $i=1$ to $n$
$p_{(m)}(0) = p_{(m)}(0) - p_{(m)}(i)$
For $m=1$ to $M$
$L_{(m)} = l_{(m)}/G(N)$
For $m=1$ to $M$
$R_{(m)} = r_{(m)}G(N-1)/G(N)$
$Q_{(m)} = L_{(m)}/R_{(m)}$

Note that we do not have to save $G(0)$, ..., $G(N-2)$. However, if we do save these values, then we can readily recompute measures for individual queues for populations less than $N$, *without recomputing measures for other queues*. This is especially significant when using machines with very limited memory, e.g., programmable calculators and home computers. LBANC has been implemented on two popular programmable pocket calculators [REYN80]. This appears to be an advantage of LBANC over Mean Value Analysis in such memory limited situations. Note that LBANC and Mean Value Analysis can be used with pocket calculators with large single chain populations because the storage required is independent of the populations (assuming we do not save $G(0)$, ..., $G(N-2)$ with LBANC).

We illustrate LBANC with the example of Figure 5.20. Since each queue has exactly one class, we will not parenthesize the queue subscripts. We have $a_1 = .5$, $a_2 = 1$, $a_3 = .25$, $\text{CAP}_3(1) = 1$, $\text{CAP}_3(2) = \text{CAP}_3(3) = 2$, $p_{21} = .5$, and $p_{23} = .5$.

1. Compute relative throughputs.
   Let $r_2 = 1$.
   Then $r_1 = r_2 \times 0.5 = 0.5$. Also $r_3 = r_2 \times 0.5 = 0.5$.
2. Compute relative utilizations.

$u_1 = r_1/a_1 = 1.0$, $u_2 = 1.0$, $u_3 = 2.0$.

3. Computation for $N = 1$

$G(0)$ is defined to be 1.

Single server: $l_1(1) = u_1 = 1.0$

Infinite server: $l_2(1) = u_2 = 1.0$

Variable rate: $p_3(1 \mid 1) = u_3 = 2.0$

$l_3(1) = 1.0 \times p_3(1 \mid 1) = 2.0$

$G(1) = l_1(1) + l_2(1) + l_3(1) = 4$

$p_3(0 \mid 1) = G(1) - p_3(1 \mid 1) = 2.0$

$L_1(1) = 1/4 = .25$ , $L_2(1) = .25$, $L_3(1) = .5$

$R_1(1) = r_1 G(0)/G(1) = .125$, $R_2(1) = .25$, $R_3(1) = .125$

$Q_1(1) = .25/.125 = 2.0$, $Q_2(1) = 1.0$, $Q_3(1) = 4.0$

$U_1(1) = .125/.5 = .25$, $U_2(1) = .25/\infty = 0$, $U_3(1) = .25$

4. Computation for $N = 2$

Single server: $l_1(2) = u_1(G(1) + l_1(1)) = 1(4+1) = 5$

Infinite server: $l_2(2) = u_2 G(1) = 4$

Variable rate:

$$p_3(2 \mid 2) = p_3(1 \mid 1)u_3/\mathrm{CAP}_3(2) = (2 \times 2)/2 = 2$$

$$p_3(1 \mid 2) = p_3(0 \mid 1)u_3/\mathrm{CAP}_3(1) = 2 \times 2 = 4$$

$$l_3(2) = 2 \times p_3(2 \mid 2) + 1 \times p_3(1 \mid 2) = 8$$

$$G(2) = \frac{l_1(2) + l_2(2) + l_3(2)}{2} = 8.5$$

$p_3(0 \mid 2) = G(2) - p_3(2 \mid 2) - p_3(1 \mid 2) = 2.5$

$L_1(2) = 10/17$, $L_2(2) = 8/17$, $L_3(2) = 16/17$

$R_1(2) = 4/17$, $R_2(2) = 8/17$, $R_3(2) = 4/17$

$Q_1(2) = (10/17)/(4/17) = 2.5$, $Q_2(2) = 1.0$, $Q_3(2) = 4.0$

$U_1(2) = (4/17)/.5 = 8/17$, $U_2(2) = 0$, $U_3(2) = 8/17$

5. Computation for $N = 3$

Single server: $l_1(3) = 1(8.5+5) = 13.5$

Infinite server: $l_2(3) = 8.5$

Variable rate:

$$p_3(3 \mid 3) = (2 \times 2)/2 = 2$$

$$p_3(2 \mid 3) = (4 \times 2)/2 = 4$$

$$p_3(1 \mid 3) = (2.5 \times 2)/1 = 5$$

$$l_3(3) = (3 \times 2) + (2 \times 4) + (1 \times 5) = 19$$

$$G(3) = \frac{13.5 + 8.5 + 19}{3} = \frac{41}{3}$$

$$p_3(0 \mid 3) = 41/3 - (5 + 4 + 2) = 8/3$$
$$L_1(3) \approx .988\ ,\ L_2(3) \approx .622,\ L_3(3) \approx 1.390$$
$$R_1(3) \approx .311,\ R_2(3) \approx .622,\ R_3(3) \approx .311$$
$$Q_1(3) \approx 3.176,\ Q_2(3) = 1.0,\ Q_3(3) \approx 4.471$$
$$U_1(3) \approx .622,\ U_2(3) = 0,\ U_3(3) \approx .622$$

**The CCNC Algorithm**

The name arises from the fact that all fixed rate and infinite server queues coalesce into a single (composite) queue. CCNC is principally intended for use with programmable calculators, though it certainly is not restricted to calculator implementations. It applies only to queues with fixed rate servers or infinite servers; it can be used in conjunction with the Convolution Algorithm for networks with queues with variable rates of service. It takes advantage of exponentiation and factorial operations usually provided as machine instructions and is trivial to implement to obtain normalizing constants. Other performance measures would typically be obtained by the unnormalized mean queue length expressions of LBANC (equations (5.79) and (5.80)) and other standard expressions, but on a *queue by queue* basis, since $G$ has already been obtained.

As before, let queues $1,\ldots,I$ have fixed rate servers and let queues $I+1,\ldots,J$ be infinite server queues. Define CUM($n$) as follows (CUM is an abbreviation for cumulative):

$$\text{CUM}(1) = \sum_{m=1}^{J} u_{(m)}$$

$$\text{CUM}(n) = \sum_{m=1}^{I} u_{(m)}^{n} \text{ for } n > 1.$$

Then from the Aggregate Queue Theorem (equation (5.60))

$$G(n) = \frac{G(n-1)\text{CUM}(1) + \ldots + G(0)\text{CUM}(n)}{n}.$$

Since the algorithm which follows from these equations is trivial, we shall not present a program-like description.

**Using CCNC with the pocket calculator.** The difficulty with using LBANC on pocket calculators is that there may not be enough registers to store the $\{u_{(m)}\}$ for all queues if there is a large number (say, over 15) of queues. (Similar, if not more severe, problems arise in using convolution or Mean Value Analysis on pocket calculators.) With CCNC, after the user enters any single $u_{(m)}$, the calculator program computes the partial sums of the CUM array. When the user enters $u_{(m+1)}$, it can be placed in the register which previously held $u_{(m)}$. After all the $\{u_{(m)}\}$ are entered, the $G$ array is computed directly from the CUM array using equation (5.60). To compute metrics for any queue $m$, the user reenters $u_{(m)}$, and metrics are computed from $u_{(m)}$ and the $G$ array using the LBANC equations. The difficulty with using CCNC with large populations is that there may not be enough registers to store the CUM array. In this case LBANC is preferable.

### The Convolution Algorithm

We assume that either LBANC or CCNC has been used to obtain the normalization constant for a network consisting of fixed rate and infinite server queues 1,...,$J$. Note that such queues could be directly considered in the following algorithm, but this would usually not be appropriate. Let the normalization constant for queues 1,...,$J$ be $G_J(n)$ given a population of $n$. The remainder of the $M$ queues, i.e., queues $J+1,...,M$ are variable rate queues. For queue $m$, $m = J+1,...,M$, define a vector $X_{(m)}$ of length $N+1$ ($N$ is the population), where $X_{(m)}(0)$ is defined to be 1 and

$$X_{(m)}(n) = \frac{u_{(m)}^n}{\mathrm{CAP}_{(m)}(n)...\mathrm{CAP}_{(m)}(1)}, \quad n > 0.$$

$X_{(m)}(n)$ is the normalization constant for a network consisting *only* of queue $m$ given a population of $n$.

We define a Convolution operator * as follows: if $X$ and $Y$ are vectors of length $N+1$, then $Z = X*Y$ is also a vector of length $N+1$ where

$$Z(n) = \sum_{i=0}^{n} X(i)\,Y(n-i), \quad n = 0,...,N.$$

After computing $G_J$ (i.e., the normalization constants for queues with fixed rate or infinite servers), the Convolution Theorem (5.63) tells us that we may compute G for the entire network as

$$G = G_J * X_{(J+1)} * ... * X_{(M)}.$$

Though we can still apply the Throughput Theorem, for other measures we must obtain the queue length distribution. This can be done most efficiently by applying equations (5.81) and (5.83). However, it may be numerically more appropriate to use the following (see Section 5.7.3). Let $G_{M-(m)}$ be the normalization constant vector for the network with queue $m$ *omitted*. Then

$$P_{(m)}(n \mid N) = \frac{X_{(m)}(n) G_{M-(m)}(N-n)}{G_M(N)} \quad \text{for } n = 0,...,N. \tag{5.84}$$

Though $G_{M-(m)}$ could be obtained from $G_M$ and $X_{(m)}$, such an approach has poorer numerical properties than computing $G_{M-(m)}$ directly. (Such direct computation will require significant additional storage and/or redundant computational effort.) Utilizations and mean queue lengths can be directly obtained from the queue length distribution and mean queueing times from Little's Rule.

We now repeat the example of Figure 5.20 with CCNC and the Convolution Algorithm. Again we do not parenthesize subscripts. We first compute normalization constants for the network consisting of the fixed rate and infinite server queues, i.e., queues 1 and 2.

$$\text{CUM}(1) = u_1 + u_2 = 2 \quad \text{(only fixed rate and IS queues)}$$

$$\text{CUM}(2) = u_1^2 = 1 \text{ (only fixed rate queues)}$$

$$\text{CUM}(3) = u_1^3 = 1 \text{ (only fixed rate queues)}$$

$G_2(n)$ is the normalization constant for a network consisting of queues 1 and 2 given a population of $n$.

$$\begin{aligned} G_2(0) &= 1 \qquad \text{(Initial condition)} \\ G_2(1) &= G_2(0)\text{CUM}(1) = 2 \\ G_2(2) &= (G_2(1)\text{CUM}(1) + G_2(0)\text{CUM}(2))/2 = 2.5 \\ G_2(3) &= (G_2(2)\text{CUM}(1) + G_2(1)\text{CUM}(2) + G_2(0)\text{CUM}(3))/3 \\ &= 8/3 \end{aligned}$$

For queue 3 in the example we have $\text{CAP}_3(1) = 1$ and $\text{CAP}_3(n) = 2$ for $n > 1$ because the service capacity of the queue with 2 or more jobs is twice that of the service with only one job. Hence

$$X_3(1) = u_3^1/\text{CAP}_3(1) = 2$$

$$X_3(2) = u_3^2/(\text{CAP}_3(2) \times \text{CAP}_3(1)) = 2^2/2 = 2$$

$$X_3(3) = u_3^3/(\text{CAP}_3(3) \times \text{CAP}_3(2) \times \text{CAP}_3(1)) = 2$$

Thus we have

$$G_2 = G_2(0),\ldots,G_2(3) = (1\ 2\ 2.5\ 8/3)$$

and

$$X_3 = X_3(0),\ldots,X_3(3) = (1\ 2\ 2\ 2).$$

Then $G = G_2 * X_3$, i.e.,

$$\begin{aligned} G(0) &= G_2(0)X_3(0) = 1 \\ G(1) &= G_2(0)X_3(1) + G_2(1)X_3(0) = 4 \\ G(2) &= G_2(0)X_3(2) + G_2(1)X_3(1) + G_2(2)X_3(0) \\ &= 8.5 \\ G(3) &= G_2(0)X_3(3) + \ldots + G_2(3)X_3(0) = 41/3 \end{aligned}$$

All performance measures could be computed exactly as before (in the LBANC version of the example). The computation of the queue length distribution for queue 3 by equation (5.84) would be

$$\begin{aligned} P_3(0 \mid 1) &= (X_3(0)*G_2(1))/G(1) = 2/4 = .5 \\ P_3(1 \mid 1) &= (X_3(1)*G_2(0))/G(1) = 2/4 = .5 \\ P_3(0 \mid 2) &= (X_3(0)*G_2(2))/G(2) = 2.5/8.5 = 5/17 \\ P_3(1 \mid 2) &= (X_3(1)*G_2(1))/G(2) = 4/8.5 = 8/17 \\ P_3(2 \mid 2) &= (X_3(2)*G_2(0))/G(2) = 2/8.5 = 4/17 \\ P_3(0 \mid 3) &= (X_3(0)*G_2(3))/G(3) = (8/3)/(41/3) = 8/41 \\ P_3(1 \mid 3) &= (X_3(1)*G_2(2))/G(3) = 5/(41/3) = 15/41 \\ P_3(2 \mid 3) &= (X_3(2)*G_2(1))/G(3) = 4/(41/3) = 12/41 \\ P_3(3 \mid 3) &= (X_3(3)*G_2(0))/G(3) = 2/(41/3) = 6/41 \end{aligned}$$

**Performance Metrics by Class**

Let $c$ be a class in $\mathscr{C}_m$, the set of classes of queue $m$, and let $q_c$ be the probability that a random job in queue $m$ is in class $c$. Then

$$q_c = u_c / u_{(m)} \tag{5.85}$$

The conditional probability of $n_c$ jobs of class $c$ in the queue, *given* a total of $n$ jobs in queue $m$ is the multinomial

$$\frac{n!}{\prod_{c \text{ in } \mathscr{C}_m} n_c!} \prod_{c \text{ in } \mathscr{C}_m} q_c^{n_c}. \tag{5.86}$$

We can use this conditional probability to compute queue length distributions by class from the queue length distribution for all jobs. Mean queue lengths are readily available from

$$L_c = q_c L_{(m)}. \tag{5.87}$$

Throughputs can be determined from relative throughputs, i.e.,

$$R_c(N) = r_c \frac{G(N-1)}{G(N)}. \tag{5.88}$$

Mean queueing times can then be determined from Little's rule.

**Retrieving Discarded Measures**

Because of storage limitations, one may discard performance estimates and then discover that they are needed. For example, in programming LBANC one can discard the measures for smaller populations once the measures for the current population are obtained. We now discuss the computation of performance metrics for a population of $N-1$ from the metrics obtained for a population of $N$. To compute queue length distributions we rewrite equation (5.38) as

$$p_{(m)}(n \mid N-1) = p_{(m)}(n+1 \mid N)\mathrm{CAP}_{(m)}(n+1)/u_{(m)}. \tag{5.89}$$

For fixed rate servers equation (5.79) can be rewritten

$$l_{(m)}(n-1) = (l_{(m)}(n)/u_{(m)}) - G(N-1). \tag{5.90}$$

For infinite servers, we can use equation (5.80).

**Changing the Relative Utilization of a Queue**

Suppose we change the relative utilization of queue $m$ from $u_{\mathrm{OLD}}$ to $u_{\mathrm{NEW}}$. Let $P_{\mathrm{OLD}}(\mathrm{n} \mid \mathrm{N})$ and $P_{\mathrm{NEW}}(\mathrm{n} \mid \mathrm{N})$ be the equilibrium probability of $n$ jobs in queue $m$ given a population of $N$, when the relative utilization is $u_{\mathrm{OLD}}$ and $u_{\mathrm{NEW}}$, respectively. Then, from equation (5.64)

$$p_{\mathrm{NEW}}(n \mid N) = p_{\mathrm{OLD}}(n \mid N)\left(\frac{u_{\mathrm{NEW}}}{u_{\mathrm{OLD}}}\right)^n. \tag{5.91}$$

Let $G_{\mathrm{NEW}}(N)$ be the normalization constant for the new model. Then

$$G_{NEW}(N) = \sum_{n} p_{NEW}(n \mid N). \tag{5.92}$$

(Now we can compute the queue length distribution for queue $m$ and the normalization constant for a population $N-1$ as discussed above. We can repeat this process for all populations from $N-1$ down to 1.) This method can be extended to the case where the relative utilizations of several queues are altered.

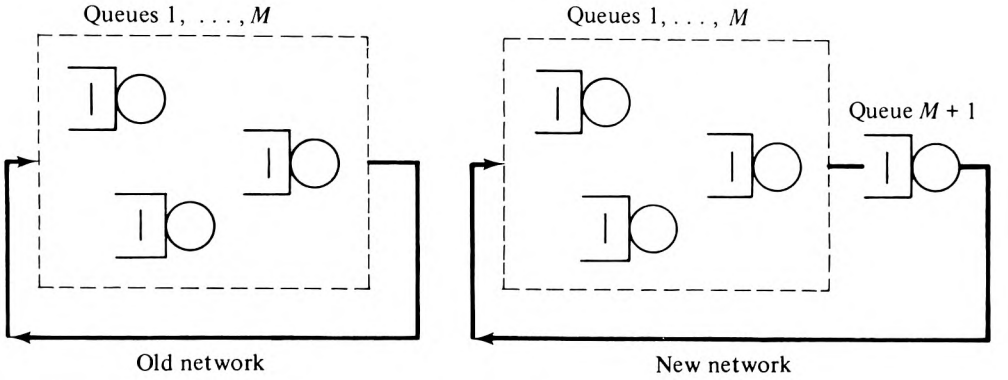



Figure 5.21

**Adding a Queue**

Suppose we wish to add a queue, $M+1$, along a path of the current system, which we call the OLD system, to get an altered system, the NEW system. Let $G_{OLD}(n)$ and $G_{NEW}(n)$ be the values of the normalization constant for the old and new systems. From the Convolution Theorem, equation (5.63)

$$G_{NEW} = G_{OLD} * X_{(M+1)} \tag{5.93}$$

Note that this applies to fixed rate and infinite server queues as long as we define $X_{(M+1)}$ appropriately.

**Deleting a Queue**

Suppose we delete queue $m$ from the old (i.e., current) system to get a NEW system. We wish to compute statistics for the NEW system from the old one. Let $P_{OLD}(n \mid N)$ be the probability of $n$ jobs in queue $m$ given a population of $N$, in the OLD system. Then (see equation (5.64))

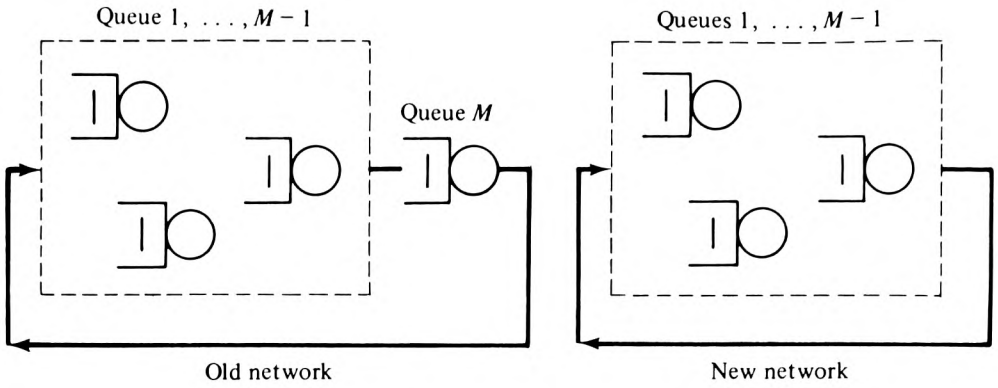



Figure 5.22

$$P_{\text{OLD}}(n \mid N) = \frac{X_{(m)}(n) G_{\text{NEW}}(N - n)}{G_{\text{OLD}}(N)}. \tag{5.94}$$

Hence

$$G_{\text{NEW}}(n) = \frac{P_{\text{OLD}}(N - n \mid N) G_{\text{OLD}}(N)}{X_{(m)}(N - n)}. \tag{5.95}$$

These methods can be used to handle the case where several queues are added and deleted.

### 5.7.2 Algorithms for Multiple Chain Networks

#### 5.7.2.1 Multiple Closed Chain Networks

We now consider multiple chain closed networks (see Figure 5.14). We will follow the same development as in the single chain case.

Equation (5.67) relating the throughputs of all classes holds independent of the number of chains:

$$R_c = \sum_{d=1}^{C} R_d p_{dc} \quad \text{for } c = 1,\ldots,C. \tag{5.96}$$

Note that $p_{dc} = 0$ if $d$ and $c$ belong to different chains. Hence, we can rewrite (5.96) as:

$$R_c = \sum R_d p_{dc} \tag{5.97}$$

where the summation is taken over all classes $d$ in the same chain as class $c$.

Let $r_c$, $c = 1,...,C$ be a set of numbers such that for all classes $c$ in chain $j$,

$$r_c = D_j R_c, \text{ for all chains } j, \tag{5.98}$$

where $D_j$ is *any* positive constant for chain $j$. Note that $D_j$ can be set independently of $D_i$ if $j \neq i$ thus we could have different proportionality factors relating the relative throughput $r_c$ to the true throughputs $R_c$ for different chains.

Further,

$$\frac{R_c}{R_d} = \frac{r_c}{r_d} \quad \text{if } c \text{ and } d \text{ are in the same chain,} \tag{5.99}$$

but the above equation usually will not hold if $c$ and $d$ are not in the same chain. Clearly, we cannot solve for the true throughputs from equation (5.96) because any set of relative throughputs will satisfy the equation (see equation (5.97)). However, if we arbitrarily *fix* the relative throughput of any class in a chain, we can compute the relative throughputs of all classes in that chain from (5.97).

Let $n_c$ be the number of jobs in class $c$, for any $c$. The $n_c$ are random variables, but at all times:

$$\sum_{c \text{ in chain } k} n_c = N_k \tag{5.100}$$

where $N_k$ is the population of chain k jobs. Taking the means of both sides of (5.100)

$$\sum_{c \text{ in chain } k} L_c = N_k \tag{5.101}$$

where $L_c$ is the average number of jobs in class $c$. Note that equations (5.96) through (5.101) hold regardless of service disciplines and distributions.

As before we restrict ourselves to local balance networks in which all queues have the PS discipline and where the service capacity may vary with the total number of jobs in the queue.

The exposition that follows assumes only two chains for notational simplicity. The theory (see Sections 5.1-5.5) is general and uses a general

notation.

**LBANC**

Consider a network with two chains. Let $N_k$ be the population of chain $k$, $k = 1,2$. Let $G(N_1,N_2)$ be the normalizing constant given populations of $N_1$ and $N_2$ for chains 1 and 2 respectively. Let $l_c(N_1,N_2)$ and $L_c(N_1,N_2)$ be the unnormalized and true (i.e. normalized) class $c$ mean queue lengths respectively, given populations of $N_1$ and $N_2$ for chains 1 and 2 respectively, i.e.,

$$L_c(N_1,N_2) = l_c(N_1,N_2)/G(N_1,N_2). \tag{5.102}$$

Indeed, all normalized mean queue lengths and queue length probabilities are found from the corresponding unnormalized values by dividing by the normalization constant.

Let $l_{(m)}(N_1,N_2)$ and $L_{(m)}(N_1,N_2)$ be the unnormalized and normalized mean queue length of queue $m$ given populations of $N_1$ and $N_2$ for chains 1 and 2 respectively.

*For fixed rate queues* we have shown that (see equation (5.50)) if class $c$ belongs to chain 1 and queue $m$

$$l_c(N_1,N_2) = \begin{cases} u_c(G(N_1 - 1,N_2) + l_{(m)}(N_1 - 1,N_2)) \text{ if } N_1 > 0 \\ 0 \text{ if } N_1 = 0 \end{cases} \tag{5.103}$$

and if class $c$ belongs to chain 2 and queue $m$

$$l_c(N_1,N_2) = \begin{cases} u_c(G(N_1,N_2 - 1) + l_{(m)}(N_1,N_2 - 1)) \text{ if } N_2 > 0 \\ 0 \text{ if } N_2 = 0 \end{cases} \tag{5.104}$$

For IS queues we have shown that (see equation 5.50) if class $c$ belongs to chain 1

$$l_c(N_1,N_2) = \begin{cases} u_c G(N_1 - 1,N_2) \text{ if } N_1 > 0 \\ 0 \text{ if } N_1 = 0 \end{cases} \tag{5.105}$$

and if class $c$ belongs to chain 2

$$l_c(N_1,N_2) = \begin{cases} u_c G(N_1,N_2 - 1) \text{ if } N_2 > 0 \\ 0 \text{ if } N_2 = 0. \end{cases} \tag{5.106}$$

We temporarily defer the general variable rate case. Since

$$\sum_{c \text{ in chain } 1} L_c(N_1,N_2) = N_1 \tag{5.107}$$

it follows (see equation 5.52) that

$$G(N_1,N_2) = \sum_{c \text{ in chain } 1} l_c(N_1,N_2)/N_1. \tag{5.108}$$

Similarly

$$G(N_1,N_2) = \sum_{c \text{ in chain } 2} l_c(N_1,N_2)/N_2 \tag{5.109}$$

and

$$G(N_1,N_2) = \frac{\sum_c l_c(N_1,N_2)}{N_1 + N_2}. \tag{5.110}$$

We can easily extend LBANC to multiple chains using the above equations. However, to do so would require storing $l_c(N_1,N_2)$ for all classes $c$. We can improve the speed of the algorithm and reduce the amount of memory required, in some cases, if we store information by chains rather than classes. We can then determine individual class values in a manner analogous to the one used with the single chain algorithms.

Let $l_{(k,m)}(N_1,N_2)$ and $L_{(k,m)}(N_1,N_2)$ be the unnormalized and normalized average number of chain $k$ jobs in queue $m$, respectively, given a population vector $(N_1,N_2)$. Thus

$$l_{(k,m)}(N_1,N_2) = \sum_{c \text{ in chain } k \text{ and queue } m} l_c(N_1,N_2) \tag{5.111}$$

and $L_{(k,m)}(N_1,N_2)$ is defined similarly. Let $u_{(k,m)}$ be the relative utilization of chain $k$ jobs in queue $m$, i.e.,

$$u_{(k,m)}(N_1,N_2) = \sum_{c \text{ in chain } k \text{ and queue } m} u_c. \tag{5.112}$$

Summing equation (5.103) and (5.104) over all c in chain 1 and queue $m$ we get for fixed rate servers

$$l_{(1,m)}(N_1,N_2) = u_{(1,m)}(G(N_1 - 1,N_2) + l_{(m)}(N_1 - 1,N_2)) \quad (5.113)$$

for $N_1 > 0$ and $l_{(1,m)}(N_1,N_2) = 0$, otherwise. Similarly,

$$l_{(2,m)}(N_1,N_2) = u_{(2,m)}(G(N_1,N_2 - 1) + l_{(m)}(N_1,N_2 - 1)) \quad (5.114)$$

for $N_2 > 0$ and $l_{(2,m)}(N_1,N_2) = 0$, otherwise. Similarly, for IS queues,

$$l_{(1,m)}(N_1,N_2) = \begin{cases} u_{(1,m)}G(N_1 - 1,N_2) \text{ if } N_1 > 0 \\ 0 \text{ if } N_1 = 0 \end{cases} \quad (5.115)$$

and

$$l_{(2,m)}(N_1,N_2) = \begin{cases} u_{(2,m)}G(N_1,N_2 - 1) \text{ if } N_2 > 0 \\ 0 \text{ if } N_2 = 0. \end{cases} \quad (5.116)$$

For variable rate queues let $p_{(m)}(n_1,n_2 \mid N_1,N_2)$ be the unnormalized probability of $n_1,n_2$ jobs at queue $m$ given populations $N_1,N_2$. Similarly, let $p_{(m)}(n \mid N_1,N_2)$ be the probability of $n$ jobs at queue $m$ given populations $N_1,N_2$. From equation (5.38), for $(n_1,n_2) \neq (0,0)$, $p_{(m)}(n_1,n_2 \mid N_1,N_2)$

$$= \begin{cases} \dfrac{p_{(m)}(n_1 - 1,n_2 \mid N_1 - 1,N_2)u_{(1,m)}}{n_1 \text{SHARE}_{(m)}(n_1+n_2)} \text{ if } n_1,\ N_1 > 0 \\ \dfrac{p_{(m)}(n_1,n_2 - 1 \mid N_1,N_2 - 1)u_{(2,m)}}{n_2 \text{SHARE}_{(m)}(n_1+n_2)} \text{ if } n_2,\ N_2 > 0 \\ 0 \text{ if } N_1 \text{ and } N_2 = 0. \end{cases} \quad (5.117)$$

We can then determine

$$l_{(1,m)}(N_1,N_2) = \sum_{n_1=1}^{N_1} \sum_{n_2=0}^{N_2} n_1 p_{(m)}(n_1,n_2 \mid N_1,N_2) \quad (5.118)$$

and

$$l_{(2,m)}(N_1,N_2) = \sum_{n_1=0}^{N_1} \sum_{n_2=1}^{N_2} n_2 p_{(m)}(n_1,n_2 \mid N_1,N_2). \tag{5.119}$$

Alternatively, if we are only interested in *chain independent* values for the variable rate queues, we can simply determine $l_{(m)}(N_1,N_2)$ from the following equations. By appropriate summations we can show that

$$p_{(m)}(n \mid N_1,N_2) = \frac{p_{(m)}(n-1 \mid N_1 - 1,N_2)u_{(1,m)}}{n\text{SHARE}_{(m)}(n)} + \frac{p_{(m)}(n-1 \mid N_1,N_2 - 1)u_{(2,m)}}{n\text{SHARE}_{(m)}(n)} \tag{5.120}$$

if $n$, $N_1$ and $N_2 > 0$. If either $N_1$ or $N_2$ is zero, then $p_{(m)}(n \mid N_1,N_2)$ can be determined from equation (5.117). Then

$$l_{(m)}(N_1,N_2) = \sum_{n=0}^{N_1+N_2} np_{(m)}(n \mid N_1,N_2). \tag{5.121}$$

Rewriting equation (5.108) we have

$$G(N_1,N_2) = \sum_{m=1}^{M} l_{(1,m)}(N_1,N_2)/N_1. \tag{5.122}$$

Similarly,

$$G(N_1,N_2) = \sum_{m=1}^{M} l_{(2,m)}(N_1,N_2)/N_2 \tag{5.123}$$

and

$$G(N_1,N_2) = \sum_{m=1}^{M} \frac{l_{(m)}(N_1,N_2)}{N_1 + N_2} \tag{5.124}$$

where $l_{(m)}(N_1,N_2)$ is, of course, $l_{(1,m)}(N_1,N_2) + l_{(2,m)}(N_1,N_2)$. Finally, for the variable rate queues we can use

$$p_{(m)}(0,0 \mid N_1,N_2) = G(N_1,N_2) - \sum_{n_1+n_2 \neq 0} p_{(m)}(n_1,n_2 \mid N_1,N_2) \tag{5.125}$$

or

$$p_{(m)}(0 \mid N_1,N_2) = G(N_1,N_2) - \sum_{n=1}^{N_1+N_2} p_{(m)}(n \mid N_1,N_2). \quad (5.126)$$

The extension of LBANC to multiple chains is simply the application of an appropriate subset of equations (5.111) through (5.126). We iterate on increasing values of $N_1$ and $N_2$, computing the unnormalized mean queue lengths and then computing $G(N_1,N_2)$ from the unnormalized mean queue lengths. We can then determine normalized mean queue lengths. From the Throughput Theorem (equation (5.29), we can determine

$$R_{(1,m)}(N_1,N_2) = r_{(1,m)} \frac{G(N_1 - 1,N_2)}{G(N_1,N_2)} \quad (5.127)$$

and

$$R_{(2,m)}(N_1,N_2) = r_{(2,m)} \frac{G(N_1,N_2 - 1)}{G(N_1,N_2)} \quad (5.128)$$

where $r_{(k,m)}$ is the sum of $r_c$ such that class $c$ is in chain $k$ and queue $m$. We can then obtain utilizations from equation (2.7) and mean queueing times from Little's Rule.

As an example consider a three queue network in which queues 1 and 2 are fixed rate servers and queue 3 is an infinite server queue. Let there be 2 chains. The relative utilization of queues by chains is:

$$u_{(1,1)} = 1 \; u_{(1,2)} = 2 \; u_{(1,3)} = 1$$
$$u_{(2,1)} = 1 \; u_{(2,2)} = 1 \; u_{(2,3)} = 2$$

Suppose we wish to compute the $G$ matrix for a population vector $(N_1,N_2) = (2,1)$. We may proceed as follows.

1. Initialization:

   $l_{(k,m)}(0,0) = 0$ for all $k,m$

   and $G(0,0) = 1$

2. $N_1 = 1, N_2 = 0$:

   Fixed Rate Server: Queue 1, Chain 1

   $$l_{(1,1)}(1,0) = u_{(1,1)}(G(0,0) + l_{(1)}(0,0)) = 1(1 + 0) = 1$$

Fixed Rate Server: Queue 2, Chain 1

$$l_{(1,2)}(1,0) = u_{(1,2)}(G(0,0) + l_{(2)}(0,0)) = 2(1 + 0) = 2$$

Infinite Server: Queue 3, Chain 1

$$l_{(1,3)}(1,0) = u_{(1,3)}G(0,0) = 1$$

$l_{(2,m)}(1,0) = 0$ and hence $l_{(m)}(1,0) = l_{(1,m)}(1,0)$ for all $m$.

$$G(1,0) = \frac{l_{(1)}(1,0) + l_{(2)}(1,0) + l_{(3)}(1,0)}{1} = 4$$

3. $N_1 = 0, N_2 = 1$

Fixed Rate Server, Queue 1, Chain 2

$$l_{(2,1)}(0,1) = u_{21}(G(0,0) + l_{(1)}(0,0)) = 1(1 + 0) = 1$$

Fixed Rate Server, Queue 2, Chain 2

$$l_{(2,2)}(0,1) = u_{(2,2)}(G(0,0) + l_{(2)}(0,0)) = 1(1 + 0) = 1$$

Infinite Server, Queue 3, Chain 2

$$l_{(2,3)}(0,1) = u_{(2,3)}G(0,0) = 2$$

$l_{(1,m)}(0,1) = 0$ and hence $l_{(m)}(0,1) = l_{(2,m)}(0,1)$ for all $m$.

Normalizing Constant

$$G(0,1) = \frac{l_{(21)}(0,1) + l_{(22)}(0,1) + l_{(23)}(0,1)}{1} = 4$$

4. $N_1 = 2, N_2 = 0$

Fixed Rate Server: Queue 1, Chain 1

$$l_{(1,1)}(2,0) = u_{(1,1)}(G(1,0) + l_{(1)}(1,0)) = 1(4 + 1) = 5$$

Fixed Rate Server: Queue 2, Chain 1

$$l_{(1,2)}(2,0) = u_{(1,2)}(G(1,0) + l_{(2)}(1,0)) = 2(4 + 2) = 12$$

Infinite Server, Queue 3, Chain 1

$$l_{(1,3)}(2,0) = u_{(1,3)}G(1,0) = 4$$

$l_{(2,m)}(2,0) = 0$ and hence $l_{(m)}(2,0) = l_{(1,m)}(2,0)$ for all $m$.

Normalizing Constant

$$G(2,0) = \frac{l_{(1)}(2,0) + l_{(2)}(2,0) + l_{(3)}(2,0)}{2} = \frac{21}{2}$$

5. $N_1 = 1, N_2 = 1$

Fixed Rate Server, Queue 1, Chain 1

$$l_{(1,1)}(1,1) = u_{(1,1)}(G(0,1) + l_{(1)}(0,1)) = 1(4 + 1) = 5$$

Fixed Rate Server, Queue 2, Chain 1

$$l_{(1,2)}(1,1) = u_{(1,2)}(G(0,1) + l_{(2)}(0,1)) = 2(4 + 1) = 10$$

Infinite Server, Queue 3, Chain 1

$$l_{(1,3)}(1,1) = u_{(1,3)}G(0,1) = 1 \times 4 = 4$$

Fixed Rate Server, Queue 1, Chain 2

$$l_{(2,1)}(1,1) = u_{(2,1)}(G(1,0) + l_{(1)}(1,0)) = 1(4 + 1) = 5$$

Fixed Rate Server, Queue 2, Chain 2

$$l_{(2,2)}(1,1) = u_{(2,2)}(G(1,0) + l_{(2)}(1,0)) = 1(4 + 2) = 6$$

Infinite Server, Queue 3, Chain 2

$$l_{(2,3)}(1,1) = u_{(2,3)}G(1,0) = 2 \times 4 = 8$$

Mean Queue Lengths - Jobs of Both Chains

$$l_{(1)}(1,1) = l_{(1,1)}(1,1) + l_{(2,1)}(1,1) = 5 + 5 = 10$$

$$l_{(2)}(1,1) = l_{(1,2)}(1,1) + l_{(2,2)}(1,1) = 10 + 6 = 16$$

$$l_{(3)}(1,1) = l_{(1,3)}(1,1) + l_{(2,3)}(1,1) = 4 + 8 = 12$$

Normalizing Constant

$$G(1,1) = \frac{l_{(1)}(1,1) + l_{(2)}(1,1) + l_{(3)}(1,1)}{N_1 + N_2} = 19$$

6. $N_1 = 2$, $N_2 = 1$

Fixed Rate Server, Queue 1, Chain 1

$$l_{(1,1)}(2,1) = u_{(1,1)}(G(1,1) + l_{(1)}(1,1)) = 1(19 + 10) = 29$$

Fixed Rate Server, Queue 2, Chain 1

$$l_{(1,2)}(2,1) = u_{(1,2)}(G(1,1) + l_{(2)}(1,1)) = 2(19 + 16) = 70$$

Infinite Server, Queue 3, Chain 1

$$l_{(1,3)}(2,1) = u_{(1,3)}(G(1,1) = 1 \times 19 = 19$$

Fixed Rate Server, Queue 1, Chain 2

$$l_{(2,1)}(2,1) = u_{21}(G(2,0) + l_{(1)}(2,0)) = 1(10.5 + 5) = 15.5$$

Fixed Rate Server, Queue 2, Chain 2

$$l_{(2,2)}(2,1) = u_{(2,2)}(G(2,0) + l_{(2)}(2,0)) = 1(10.5 + 12) = 22.5$$

Infinite Server, Queue 3, Chain 2

$$l_{(2,3)}(2,1) = u_{(2,3)}(G(2,0) = 2 \times 10.5 = 21$$

Mean Queue Lengths - Jobs of Both Chains

$$l_{(1)}(2,1) = l_{1,1}(2,1) + l_{(2,1)}(2,1) = 29 + 15.5 = 44.5$$

$$l_{(2)}(2,1) = l_{1,2}(2,1) + l_{(2,2)}(2,1) = 70 + 22.5 = 92.5$$

$$l_{(3)}(2,1) = l_{1,3}(2,1) + l_{(2,3)}(2,1) = 19 + 21 = 40$$

Normalizing Constant

$$G(2,1) = \frac{l_{(1)}(2,1) + l_{(2)}(2,1) + l_{(3)}(2,1)}{N_1 + N_2} = 59$$

Hence the matrix of normalization constants is:

$G(0,0) = 1$ $\quad G(0,1) = 4$
$G(1,0) = 4$ $\quad G(1,1) = 19$
$G(2,0) = 10.5$ $\quad G(2,1) = 59$

**CCNC**

We next discuss CCNC for networks of fixed rate and infinite server queues with multiple chains. This algorithm is based on equation (5.60). As before, let queues 1,...,$I$ be fixed rate single server queues and let queues $I+1$,...,$J$ be infinite server queues. Define CUM($n_1$,$n_2$) as follows:

$$\text{CUM}(1,0) = \sum_{m=1}^{J} u_{(1,m)} \tag{5.129}$$

$$\text{CUM}(0,1) = \sum_{m=1}^{J} u_{(2,m)} \tag{5.130}$$

$$\text{CUM}(n_1,n_2) = \frac{(n_1+n_2)!}{n_1!n_2!} \sum_{m=1}^{I} u_{(1,m)}^{n_1} u_{(2,m)}^{n_2} \text{ for } n_1 + n_2 > 1. \tag{5.131}$$

Then

$$G(N_1,N_2) = \sum_{(n_1,n_2)\neq(0,0)} \frac{G(N_1 - n_1, N_2 - n_2)CUM(n_1,n_2)}{N_1 + N_2} \tag{5.132}$$

for $(N_1,N_2) \neq (0,0)$.

We apply CCNC to the example just solved with LBANC.

1. CUM($n_1$,$n_2$)

$$\text{CUM}(1,0) = u_{(1,1)} + u_{(1,2)} + u_{(1,3)} = 4$$
$$\text{CUM}(0,1) = u_{(2,1)} + u_{(2,2)} + u_{(2,3)} = 4$$
$$\text{CUM}(1,1) = \frac{(1+1)!}{1!1!}(u_{(1,1)}u_{(2,1)} + u_{(1,2)}u_{(2,2)}) = 6$$
$$\text{CUM}(2,0) = u_{(1,1)}^2 + u_{(1,2)}^2) = 5$$
$$\text{CUM}(2,1) = \frac{(2+1)!}{2!1!}(u_{(1,1)}^2 u_{(2,1)}^1 + u_{(1,2)}^2 u_{(2,2)}^1) = 15$$

2. G($n_1$,$n_2$)

$$G(1,0) = \frac{G(0,0)\text{CUM}(1,0)}{1} = 4$$

$$G(2,0) = \frac{G(0,0)\text{CUM}(2,0) + G(1,0)\text{CUM}(1,0)}{2} = 10.5$$

$$G(0,1) = \frac{G(0,0)\text{CUM}(0,1)}{1} = 4$$

$$G(1,1) = \frac{G(0,0)\text{CUM}(1,1) + G(1,0)\text{CUM}(0,1) + G(0,1)\text{CUM}(1,0)}{1+1}$$

$$= 19$$

$$G(2,1) = \frac{G(0,0)\text{CUM}(2,1) + G(1,0)\text{CUM}(1,1) + G(2,0)\text{CUM}(0,1)}{2+1}$$

$$+ \frac{G(0,1)\text{CUM}(2,0) + G(1,1)\text{CUM}(1,0)}{2+1} = 59$$

**The Convolution Algorithm**

As in the single chain case, we assume that either LBANC or CCNC has been used to obtain the normalization constant for a network consisting of fixed rate and infinite server queues 1,...,$J$. Note that such queues could be directly considered in the following, but this would usually not be appropriate. Let the normalization constant for queues 1,...,$J$ be $G_J(n_1,n_2)$ given a population of $n_1,n_2$. The remainder of the $M$ queues, i.e., queues $J+1,...,M$ are variable rate queues. For queue $m$, $m = J+1,...,M$, define a matrix $X_{(m)}$ of dimension $N_1+1$ by $N_2+1$, where $X_{(m)}(0,0)$ is defined to be 1 and

$$X_{(m)}(n_1,n_2) = \frac{\dfrac{(n_1+n_2)!}{n_1!n_2!}u_{(1,m)}^{n_1}u_{(2,m)}^{n_2}}{\text{CAP}_{(m)}(n_1+n_2)...\text{CAP}_{(m)}(1)}, \quad n_1,n_2 \neq (0,0).$$

($CAP_{(m)}(n) = nSHARE_{(m)}(n)$ is the service capacity when there are $n$ jobs in the queue.) As before, $X_{(m)}$ corresponds to the normalization constant for a network consisting *only* of queue $m$. However, a more general definition of $X_{(m)}$ is possible such that $X_{(m)}$ is the normalizing constant matrix for *an entire subnetwork!* In this case, $X_{(m)}$ would have the form

$$X_{(m)}(n_1,n_2) = H(n_1,n_2)r_{(1,m)}^{n_1}r_{(2,m)}^{n_2} \tag{5.133}$$

where the matrix $H$ is as defined in equations (5.26-5.28).

We redefine the Convolution operator * as follows: if $X$ and $Y$ are matrices of dimension $N_1+1$ by $N_2+1$, then $Z = X*Y$ is also a matrix of dimension $N_1+1$ by $N_2+1$ where

$$Z(n_1,n_2) = \sum_{i_1=0}^{n_1} \sum_{i_2=0}^{n_2} X(i_1,i_2)Y(n_1 - i_1,n_2 - i_2).$$

After computing $G_J$ (i.e., the normalization constants for queues with fixed rate or infinite servers), the Convolution Theorem (5.63) tells us that we may compute $G$ for the entire network as

$$G = G_J * X_{(J+1)} * \ldots * X_{(M)}.$$

Notice that the Convolution operator is both commutative and associative. Thus we can apply the Convolution to the $X$ matrices for several queues and then treat the resulting matrix as the $X$ vector for a composite queue. This may be appropriate computationally in parametric analysis of networks. It is extremely important as a basis for aggregation approximations, as discussed in Chapter 6.

Again as before, we can apply the Throughput Theorem, but for other measures we must obtain the queue length distribution. Let $G_{M-(m)}$ be the normalization constant matrix for the network with queue *m omitted*. Then equation (5.84) becomes

$$P_{(m)}(n_1,n_2 \mid N_1,N_2) = \frac{X_{(m)}(n_1,n_2)G_{M-(m)}(N_1 - n_1,N_2 - n_2)}{G_M(N_1,N_2)}. \quad (5.134)$$

Utilizations and mean queue lengths can be directly obtained from the queue length distribution and mean queueing times from Little's Rule.

#### 5.7.2.2 Mixed Networks

We have focused our attention on closed networks because they are most important in computer system modeling. Open networks are important in communication system modeling. With rare exceptions, published applications of queueing network models have not used mixed networks. However, mixed networks can be reasonably proposed as models of computer communication systems and other systems. We will indicate how the previous discussion of algorithms may be extended to consider mixed networks. We will assume the mixed network has exactly one closed chain and exactly one open chain and consists entirely of fixed rate single server and infinite server queues. Extension to mixed networks with multiple closed chains is trivial. Extension to mixed networks with variable rate queues is straightforward but algebraically tedious. (The tedium is directly related to the complexity of the capacity function and comparable to that of dealing with variable rate queues in isolation.)

Queues which are not visited by jobs of the closed chain are not affected by the closed chain and may be treated as queues in isolation as in open networks. Thus we assume that all $M$ queues are visited by the closed chain. We also assume that the network would not be saturated if the closed chain had zero population. (As observed in REIS75, the pioneering work on algorithms for mixed networks, closed chains do not affect the *stability* of mixed networks.) For the moment, let us assume that only queue $m$ is visited by the open chain and that queue $m$ is fixed rate single server. Let $P_{(m)}(n_{\mathrm{OP}(m)},n_{\mathrm{CL}(m)} \mid N)$ be the distribution of open and closed chain jobs at queue $m$, given the closed chain has population $N$. Then it is easy to show from earlier results (see equations (5.18), (5.19), (5.24) and (5.84)) that for $n_{\mathrm{OP}(m)} = 0,\ldots,\infty$, and $n_{\mathrm{CL}(m)} = 0,\ldots,N$,

$$P_{(m)}(n_{\mathrm{OP}(m)},n_{\mathrm{CL}(m)} \mid N) = \tag{5.135}$$

$$\frac{(n_{\mathrm{OP}(m)}+n_{\mathrm{CL}(m)})!}{n_{\mathrm{OP}(m)}!n_{\mathrm{CL}(m)}!} U_{\mathrm{OP}(m)}^{n_{\mathrm{OP}(m)}} u_{\mathrm{CL}(m)}^{n_{\mathrm{CL}(m)}} G_{M-(m)}(N-n_{\mathrm{CL}(m)})$$

$$\frac{}{\tilde{G}_M(N)}$$

where $u_{\mathrm{CL}(m)}$, $U_{\mathrm{OP}(m)}$, and $u_{\mathrm{OP}(m)}$ are, respectively, the closed chain relative utilization, the open chain actual utilization and the open chain relative utilization and $\tilde{G}_M(N)$ is the normalizing constant for the mixed network with closed chain population $N$. Then

$$\begin{aligned}
&\tilde{G}_M(N) \\
&= \sum_{n_{\mathrm{OP}(m)}=0}^{\infty} \sum_{n_{\mathrm{CL}(m)}=0}^{N} P_{(m)}(n_{\mathrm{OP}(m)},n_{\mathrm{CL}(m)} \mid N) \\
&= \frac{1}{1-U_{\mathrm{OP}(m)}} \sum_{n_{\mathrm{CL}(m)}=0}^{N} \frac{u_{\mathrm{CL}(m)}^{n_{\mathrm{CL}(m)}}}{(1-U_{\mathrm{OP}(m)})^{n_{\mathrm{CL}(m)}}} G_{M-(m)}(N-n_{\mathrm{CL}(m)}) \\
&= \frac{1}{1-U_{\mathrm{OP}(m)}} \sum_{n_{\mathrm{CL}(m)}=0}^{N} \tilde{u}_{\mathrm{CL}(m)}^{n_{\mathrm{CL}(m)}} G_{M-(m)}(N-n_{\mathrm{CL}(m)})
\end{aligned} \tag{5.136}$$

The derivation of (5.136) and many of the following equations is straightforward if we use the relationship

$$\sum_{i=0}^{\infty} \frac{(i+j)!}{i!j!}\rho^i = \frac{1}{(1-\rho)^{(j+1)}} \quad \text{for } \rho < 1 \text{ and } j = 0,1,2,\ldots .$$

Equation (5.136) has an intuitive explanation. The first term is simply the inverse of the probability that queue $m$ is empty if the closed chain were ignored. $\tilde{u}^{n_{CL(m)}}_{CL(m)}$ as defined in (5.136) is what $X_{(m)}(n_{CL(m)})$ would be if the effective closed chain mean service rate were used, where by "effective" we mean the rate after taking away the time spent on open chain jobs. The open chain has a very localized impact on the network, affecting only our characterization of queue $m$ and its contribution to the normalizing constant.

In general,

$$\tilde{G}_M = \tilde{X}_{(1)} * \ldots * \tilde{X}_{(M)} \tag{5.137}$$

where for fixed rate single server queues

$$\tilde{X}_{(m)}(n) = \frac{u^n_{CL(m)}}{(1 - U_{OP(m)})^{n+1}} \tag{5.138}$$

and for infinite server queues

$$\tilde{X}_{(m)}(n) = \frac{u^n_{CL(m)}}{e^{-u_{OP(m)}}\, n!}. \tag{5.139}$$

(Equation (5.139) uses the probability that an infinite server queue is empty as obtained in Exercise 4.2. Equation (5.137) also holds for variable rate queues with appropriate definition of $\tilde{X}_{(m)}(n)$.)

We can rewrite (5.84) for the closed chain queue length distribution as

$$P_{CL(m)}(n \mid N) = \frac{\tilde{X}_{(m)}(n)\tilde{G}_{M-(m)}(N-n)}{\tilde{G}_M(N)} \quad \text{for } n = 0,\ldots,N. \tag{5.140}$$

Equations (5.138-5.140) are the ones relevant to the Convolution Algorithm; LBANC can be easily applied to mixed networks, also. We can rewrite equations (5.79) and (5.80) as

$$l_{CL(m)}(N) = \tilde{u}_{CL(m)}(\tilde{G}_M(N-1) + l_{CL(m)}(N-1)) \tag{5.141}$$

and

$$l_{\mathrm{CL}(m)}(N) = u_{(m)}\tilde{G}_M(N-1). \tag{5.142}$$

Letting

$$\tilde{G}_M(0) = \tilde{X}_{(1)}(0)\ldots\tilde{X}_{(M)}(0) \tag{5.143}$$

and rewriting (5.78) as

$$\tilde{G}_M(N) = \frac{\sum_m l_{\mathrm{CL}(m)}(N)}{N} \tag{5.144}$$

is all we need to complete revision of LBANC for the closed chain for our restricting assumptions (single closed chain, single server and infinite server queues). Closed chain throughput is obtained using $\tilde{G}_M$, i.e.,

$$R_{\mathrm{CL}(m)}(N) = r_{\mathrm{CL}(m)}\frac{\tilde{G}_M(N-1)}{\tilde{G}_M(N)} \tag{5.145}$$

and closed chain utilization is obtainable from this throughput by equation (2.7).

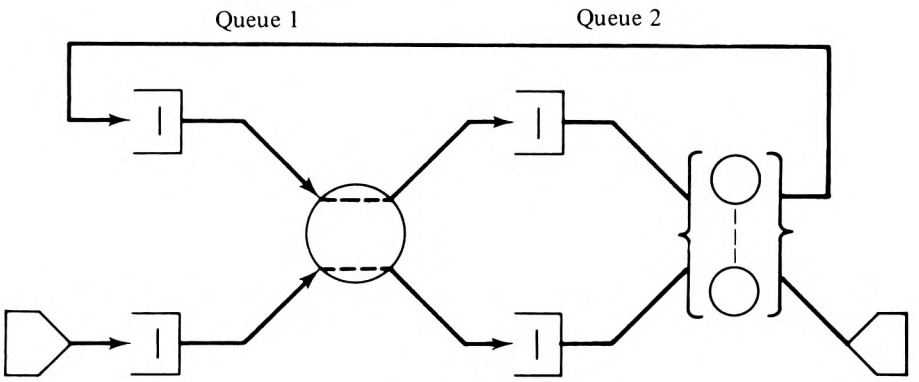



Figure 5.23

For the open chain, for the single server queues,

$$P_{OP(m)}(n \mid N) = \frac{\sum_{i=0}^{N} \frac{(n+i)!}{n!i!} U_{OP(m)}^{n} u_{CL(m)}^{i} \tilde{G}_{M-(m)}(N-i)}{\tilde{G}_{M}(N)}, \quad (5.146)$$

and

$$L_{OP(m)}(N) = \frac{\frac{U_{OP(m)}}{1-U_{OP(m)}} \sum_{i=0}^{N} (i+1)\tilde{X}_{(m)}(i)\tilde{G}_{M-(m)}(N-i)}{\tilde{G}_{M}(N)}$$

$$= \frac{U_{OP(m)}}{1 - U_{OP(m)}} \frac{l_{CL(m)}(N+1)}{\tilde{u}_{CL(m)}\tilde{G}(N)}. \quad (5.147)$$

For the infinite server queues the closed chain has no effect on the open chain and we can use the standard formulas for an infinite server queue in isolation.

Let us consider the network of Figure 5.23 with the following parameters: $N = 2$, $U_{OP(1)} = .6$, $u_{CL(1)} = 2$, $u_{OP(2)} = 3$, and $u_{CL(2)} = 3$. Then

$$\tilde{X}_{(1)}(0) = \frac{1}{1-.6} = 2.5, \tilde{X}_{(2)}(0) = e^3 \approx 20.1$$

$$\tilde{G}_2(0) = \frac{e^3}{1-.6} \approx 50.2$$

$$l_{CL(1)}(1) = \frac{2}{1-.6}\frac{e^3}{1-.6} \approx 251$$

$$l_{CL(2)}(1) = 3\frac{e^3}{1-.6} \approx 151$$

$$\tilde{G}_2(1) = l_{CL(1)}(1) + l_{CL(2)}(1) \approx 402$$

$$L_{OP(2)} = 3.$$

$$l_{CL(1)}(2) = \frac{2}{1-.6}(402 + 251) \approx 3263$$

$$L_{OP(1)}(1) = \frac{.6}{1-.6}\frac{3263}{\frac{2}{1-.6}402} \approx 2.44$$

$$l_{CL(2)}(2) \approx 3(402) = 1206$$

$$\tilde{G}_2(2) \approx \frac{3263 + 1206}{2} \approx 2235$$

$$l_{CL(1)}(3) = \frac{2}{1-.6}(2235 + 3263) \approx 27490$$

$$L_{OP(1)}(2) = \frac{.6}{1-.6}\frac{27490}{\frac{2}{1-.6}2235} \approx 3.69.$$

### 5.7.3 Numerical Properties of Computational Algorithms

So far we have essentially ignored the numerical properties of the computational algorithms. For most models which are used in practice, all of the algorithms are fairly stable. However, for some quite reasonable parameter values, some or all of the algorithms will experience numerical difficulties. Except for Reiser's thorough treatment of the Convolution Algorithm [REIS76], there has been little formal analysis of numerical properties. We will not attempt a formal analysis of the numerical properties of the algorithms. Rather, we will informally indicate some of the difficulties likely to be encountered and methods for coping with these difficulties. There are two basic difficulties that we know of: (1) Algorithms that rely on normalizing constant vectors may fail because the normalizing constant exceeds the floating point range for some populations. (2) The recursive expressions for queue length distributions used by mean value analysis and LBANC may fail for relatively small populations; thus these algorithms may not be able to handle variable rate queues for some networks. We will focus our attention on single chain networks, but will consider multiple chains as appropriate.

**Mean Values: Fixed Rate and Infinite Server Queues**

Let us first consider LBANC. Further, let us assume that all queues are either fixed rate single server or infinite server and that we do not estimate queue length distributions. The principal problem in this case is that the normalizing constant, $G(N)$, may become too small or too large. For some models with performance measures near unity and some choices of the set of relative throughputs, $\{r_c\}$, $G(N)$ may exceed the limits of floating point representations.

For example, suppose that in Figure 5.12 we number the queues as follows: CPU - queue 1, Disk - queue 2, Drum - queue 3 and Terminals - queue 4. Let us suppose the mean CPU time is 20 ms., the mean disk time is 44 ms., the mean drum time is 8 ms. and the mean think (terminal) time is 15 sec. Thus $a_1 = 1/.02$, $a_2 = 1/.044$, $a_3 = 1/.008$ and $a_4 = 1/15$. Let us further suppose that servicing of each terminal command requires an average of 5 CPU-I/O cycles and that 20% of the I/O accesses are to the disk. We can interpret this statement as $p_{2,1} = p_{3,1} = .2$, $p_{2,4} = p_{3,4} = .8$, $p_{1,2} = .2$ and $p_{1,3} = .8$. (All other probabilities are either 0 or 1, as indicated by the figure.) This completely specifies the model except for the population, i.e., the number of terminals. The modeled system is lightly loaded, and none of the queues becomes saturated until the population nears 200. If we let $r_4 = 1$, then $r_1 = 5$, $r_2 = 1$ and $r_3 = 4$. With these choices we have $u_1 = .1$, $u_2 = .044$, $u_3 = .032$ and $u_4 = 15$. Then $G(1) = 15.176$ and $G(N)$ increases with $N$ until $G(15) \approx 401382$. For $N > 15$ $G(N)$ decreases with increasing $N$ until $G(143) \approx 10^{-78}$. Attempting to compute $G(144)$ exceeds the floating point range of the IBM 360/370 series of computers, and computation terminates on such machines. (We could illustrate the same phenomenon for machines with larger floating point ranges without unreasonable choice of parameters.) Suppose, alternatively, we let $r_4 = 10$. Then $G(1) = 151.76$ and for $N \leq 200$ $G(N)$ is *apparently* monotonically non-decreasing with $N$, with an *apparently* limiting value of approximately $3.66 \times 10^{65}$. (Actually, $G(N)$ decreases slowly for $N > 200$, but it is still quite large with $N = 3000$. Since the CPU queue saturates with $N$ near 200, larger values of $N$ are not very interesting with this model.) In particular $G(200)$ is comfortably within the floating point range of the 360 and 370. So with proper choice of $\{r_c\}$ we have no numerical difficulties with this model with LBANC. Algebraically, the absolute values of $\{r_c\}$ are irrelevant, so we are free to make the choice to alleviate (and hopefully eliminate) numerical problems.

How do we make this choice? The following method is suggested in REIS78b. We can try to choose $\{r_c\}$ so that $G(N)$ is always large but within the floating point range of our machine (e.g., less than $10^{75}$ for the 360/370 series.) Certainly,

$$G(N) < \sum_{n=0}^{\infty} G(n), \tag{5.148}$$

so if we can make the right hand side of (5.148) large but not out of range then we will not have overflow in computing $G(N)$, for any $N$. This is very easily done if we make the following interpretation of the right hand side of (5.148).

Let us assume that we are using the queue numbering scheme described before, i.e., queues 1 to $I$ are fixed rate single server and queues $I+1$ to $J$ are infinite server. (Since we are only considering those two queue types for the moment, $M = J$.) Let us further assume that we have initially chosen $\{r_c\}$ so that $u_{(i)} < 1$, for $i = 1,...,I$. Then we can speak of a corresponding open network with a source and sink and the same values for $\{r_c\}$ and $\{a_c\}$ as the closed network. This open network will be stable for $R = 1$, where $R$ is the arrival rate of jobs from the source. The right hand side of (5.148) is simply the inverse of the probability that the open network is empty! I.e., if $P(\ )$ is the probability the open network is empty, from equation (5.24)

$$P(\ ) = \frac{G(0)}{\sum_{n=0}^{\infty} G(n)} = \frac{1}{\sum_{n=0}^{\infty} G(n)}. \tag{5.149}$$

This suggests an intuitive explanation of the situation where overflow may occur in computing $G(N)$: *Overflow is a potential problem when the corresponding open network is saturated!* So to avoid overflow, we need simply choose $\{r_c\}$ so that $P(\ )$ is not too small, e.g., for the 360/370 we might choose $\{r_c\}$ so that $P(\ )$ is greater than $10^{-75}$. Fortunately, $P(\ )$ is trivial to calculate. By Jackson's Theorem,

$$P(\ ) = P_{(1)}(\ )...P_{(M)}(\ ), \tag{5.150}$$

where $P_{(m)}(\ )$, $m = 1,...,M$, is the probability that queue $m$ in isolation is empty, i.e., from equation (4.4) $P_{(i)}(\ ) = 1 - u_{(i)}$, $i = 1,...,I$, and from Exercise 4.2 $P_{(j)}(\ ) = e^{-u_{(j)}}$, $j = I+1,...,J$.

The following algorithm will *scale* the relative throughputs and utilizations so that overflow cannot occur in computing $G(N)$. It assumes $I > 0$. $\delta$ is an arbitrary constant near but less than 1, e.g., .99. $\varepsilon$ is chosen so that $1/\varepsilon$ is near the limit of the floating point range of the computer being used (e.g., $\varepsilon = 10^{-75}$ could be used for the 360/370 series.)

Choose an arbitrary, algebraically valid $\{r_c\}$.

Determine $\{u_c\}$ and $\{u_{(m)}\}$.

Let $D = 1/\max(u_{(1)},\ldots,u_{(I)})$.

Repeat

$$D = \delta D$$

$$P = (1 - Du_{(1)})\ldots(1 - Du_{(I)})\exp(-D(u_{(I+1)} + \ldots + u_{(J)}))$$

Until $P \geq \varepsilon$

For $c = 1$ to $C$: $r_c = Dr_c$

Redetermine $\{u_c\}$ and $\{u_{(m)}\}$.

We leave it to the reader to verify that the algorithm terminates, to modify the algorithm to allow $I = 0$ and to modify the algorithm to allow variable rate queues. We point out that it is not usually necessary to even execute this algorithm as given, i.e., without executing the loop one can predict with sufficient accuracy the final value of $D$ from the initial value of $D$, $\delta$ and $\ln(-\varepsilon)$. The prediction method is also left to the reader. (Notice that for the initial choice of $\{r_c\}$ in the example, $\delta = .99$ and $\varepsilon = 10^{-75}$, the final value of $D$ is 9.9.) There is some freedom in extending the algorithm to multiple chains. The approach used in REIS78b is to require that

$$\max(u_{(1,1)},\ldots,u_{(1,M)}) = \ldots = \max(u_{(K,1)},\ldots,u_{(K,M)}),$$

where $u_{(k,m)}$ is the sum of $u_c$ such that class $c$ is in chain $k$ and queue $m$. We can enforce this requirement in initially determining $\{r_c\}$, and then, *for the purposes of scaling only*, determine $u_{(m)} = u_{(1,m)} + \ldots + u_{(K,m)}$ before initially determining $D$. The remainder of the algorithm is then the same.

Unfortunately, for extremes of parameter values in otherwise reasonable models, it is not possible to keep $G$ within floating point range for all values of $N$. (This statement assumes we use a single set of relative throughputs for all populations. If we are willing to use different relative throughputs for different populations, then we can keep $G$ within floating point range for all populations [LAM80].) The most common situation is when service times at IS queues very strongly dominate service times at all other queues. For example, if this is true and there is exactly one IS queue, then $G(N) \approx X_{(J)}(N)$ where

$$X_{(J)}(N) = \frac{u_{(J)}^{N}}{N!}.$$

Regardless of what we do, for some values of $N$ the numerator will be much larger than the denominator and we must be concerned about overflow, and for other values of $N$ the reverse is true and we must be concerned about underflow. If in the example the terminal think time were 300 sec., and the other parameters the same, we could not avoid both overflow and underflow on a 360 or 370. Using the above scaling algorithm, $G(N)$ increases with $N$ until $G(171) \approx 6.6 \times 10^{72}$ but then $G(N)$ decreases with $N$ until $G(618) \approx 1.06 \times 10^{-78}$. Yet no queue in the model is saturated until $N$ approaches 3000!

**Mean Value Analysis**

Thus any solution method which depends on $G(N)$ must fail for some potentially interesting models. The Mean Value Analysis Algorithm of Reiser and Lavenberg [REIS78a] does not depend on $G(N)$ and is most suited to extremes of parameter values in our current context. (There is a numerical problem with both LBANC and Mean Value Analysis in more general situations; we discuss this below.) As we said before, Mean Value Analysis is very similar to LBANC; LBANC was inspired by the Reiser and Lavenberg algorithm. Let us normalize equation (5.79):

$$L_{(m)}(N) = \frac{l_{(m)}(N)}{G(N)}$$

$$= U_{(m)}(N)(1 + L_{(m)}(N - 1)). \tag{5.151}$$

Applying Little's Rule,

$$Q_{(m)}(N) = \frac{1}{a_{(m)}}(1 + L_{(m)}(N - 1)), \tag{5.152}$$

where

$$a_{(m)} = \frac{r_{(m)}}{\sum_{c \text{ in } \mathscr{C}_m} \frac{r_c}{a_c}}.$$

(Of course, for IS queues $Q_{(m)} = 1/a_{(m)}$.) Further,

$$R_{(m)}(N) = r_{(m)} \frac{N}{\sum_{i=1}^{M} r_{(i)} Q_{(i)}(N)}. \tag{5.153}$$

This is simply Little's Rule applied to the mean cycle time (the denominator) which is the mean time for a job to make a complete cycle around the network. Thus we can get $Q_{(m)}(N)$ and $R_{(m)}$ from the model parameters and $L_{(m)}(N-1)$. Assuming we know $L_{(m)}(N-1)$, we can then get $L_{(m)}(N)$ from Little's Rule. Since we know that $L_{(m)}(0) = 0$ for all $m$, we can iteratively determine these *mean values* for $N$ from 1 up to the desired population. As long as these mean values have reasonable magnitude, there will be no numerical difficulties. In the multi-chain case, Mean Value Analysis will require approximately twice the storage of LBANC since both queueing times and queue lengths are stored.

For the example problem as stated initially, LBANC and Mean Value Analysis give the same results to three significant digits for $U$, $L$, $R$ and $Q$ for all interesting values of $N$ when the computation is performed on an IBM 370. (By interesting $N$ we mean at most 200; we have not made the comparison for larger $N$.) For the modified example problem (think time 300 sec.) run on a 370, both methods agree for $N$ up to 616. For $N = 617$ and 618 they disagree slightly, and for larger $N$ LBANC cannot be used because $G(N)$ is too small.

With CCNC and the Convolution Algorithm we have an additional problem: even though $G(N)$ is quite large, some of the intermediate values used in its computation may be quite small, small enough to cause underflow. The intermediate values we speak of are $u_{(m)}^n$ (and products of such values for several queues). In the (initial) example problem, with $u_3 = .3168$ after scaling, computation of $u_3{}^n$ will cause underflow on a 360 or 370 for $N$ in the vicinity of 155. Fortunately, since there are no negative values involved, if we replace the small intermediate values by zero, we get satisfactory results for $G(N)$ except for extreme parameter values. For the initial example problem run on a 370, LBANC, CCNC and the Convolution Algorithm agree to three significant digits for $U$, $L$, $R$ and $Q$ for $N$ up to 200, and they agree to *seven* significant digits for $G(N)$ for $N$ up to 200. But when we do have extreme parameter values, we may get grossly inaccurate results with no warning! (This assumes we proceed upon occurrence of underflow. LBANC fails very gracefully, without misleading results, when it fails.) For the modified example problem run on a 370, LBANC, CCNC and the Convolution Algorithm agree on the above values for $N$ up to 607. However, CCNC and the Convolution Algorithm behave very poorly for $N$ between 608 and 618; *at* $N = 618$, *these algorithms underestimate* $G(N)$ *by an order of magnitude and estimate* $L_4(618)$ *as* 842!

**Queue Length Distributions and Variable Rate Queues**

Before we make any hasty conclusions about the overall numerical properties of these algorithms, let us consider queue length distributions and variable rate queues. With only slight modification of the above examples, we can illustrate the less stable characteristics of LBANC and Mean Value Analysis. We focus on LBANC, but the discussion applies directly to Mean Value Analysis. With either of these algorithms it is necessary to estimate the queue length distribution at variable rate queues in order to handle such queues. In LBANC we do this with equations (5.81) and (5.83). (In Mean Value Analysis we can use the normalized equivalents of those two equations. To perform the Mean Value Analysis we also need an equation for mean queueing time of variable rate queues, such as equation (2.14) of REIS78a, which would be used in place of equation (5.152).) This will be reasonably stable *except when the probability of small queue lengths at a queue is very small.* To be more specific, as $p_{(m)}(0 \mid N)$ tends to zero LBANC will fail very gracelessly. Because of the recursive nature of (5.81), as $p_{(m)}(0 \mid N - 1)$ tends to zero we will severely underestimate $p_{(m)}(n \mid N)$ for $n > 0$. Thus we will severely *overestimate* $p_{(m)}(0 \mid N)$, and subsequently, $p_{(m)}(n \mid N + 1)$. Chaotic behavior ensues, with *negative* estimates of probabilities for small populations!

In the following discussion we assume double precision arithmetic on an IBM 370 which yields about 16 decimal digits of significance. The behavior would be somewhat worse with less precision, and somewhat better with more precision, but basically the same. Let us consider a hypothetical queueing network, not necessarily a computer system model, which is the same as the initial example network above except that the infinite server queue is replaced by one with 10 servers. If we were not forewarned of potential trouble, we would expect the same behavior observed before for the various algorithms for $N$ not much larger than 10. In fact, LBANC (and Mean Value Analysis) have trouble with $N = 10$ if we do not recognize the terminals queue as an infinite server queue but treat it in the more general context of variable rate queues. Then we get a *negative* estimate for $p_4(0 \mid 10)$. (With 370 single precision, approximately 6 significant digits, we get a negative estimate for $p_4(0 \mid 5)$. With 370 extended precision, approximately 33 significant digits, we first get a negative estimate for $p_4(0 \mid 22)$.) Though at first (with increasing $N$) there is no noticeable effect on the mean performance measures, eventually these suffer severely, as well. Queue 4 becomes saturated for $N = 12$, but if we continue the algorithm for larger $N$, at $N = 50$ we see a *decrease* in $U_4$ from 1 to approximately .998. Further increases in $N$ see $U_4 > 1$ and other impossible performance estimates.

With the Convolution Algorithm we have none of these difficulties, i.e, as long as $G(N)$ and the terms which contribute to it are of reasonable magnitude, we have no problems. This is because variable rate queues are treated in the same way as fixed rate and infinite server queues by the Convolution Algorithm. Further, there are no additional numerical problems in estimating queue length distributions if equation (5.84) is used and $G_{M-(m)}$ is calculated directly. (This direct calculation may be quite expensive in processing and/or memory.)

Of course, if one was careful, one might notice the peculiar behavior of LBANC (or Mean Value Analysis) for a particular model with variable rate queues. In the above example, one does not really care to estimate performance measures for $N > 13$, *but one may not realize that if one simply picks N (larger than 13) in advance.* There are also obvious heuristics for dealing with these difficulties, such as replacing negative probabilities by zero and scaling $p_{(m)}(n \mid N)$ so that they sum to $G(N)$. However, by the time a negative probability is discovered, it is likely too late to properly deal with the problem. The reader may object that this last example network is not relevant to computer system modeling, but similar behavior can be observed in other models with a large variation in $\mathrm{CAP}_{(m)}(n)$ with $n$, for example models of multiprocessor systems such as C.mmp (see Section 9.2 and WULF72).

Reiser has proposed a modification of Mean Value Analysis which avoids this problem [REIS80]. The modification requires the additional solution of the network with the queue of interest (the one for which we desire the queue length distribution or which has variable service rates) removed. Solving that network properly with Mean Value Analysis may in turn require solution of networks with other queues removed because those queues also have variable service rates. Thus the computational effort may be dramatically increased. A similar modification can be proposed for LBANC. Instead of estimating $p_{(m)}(0 \mid n)$ using equation (5.83), we can use

$$p_{(m)}(0 \mid n) = G_{M-(m)}(n). \tag{5.154}$$

This will be numerically appropriate provided that $G_{M-(m)}$ does not exceed the floating point range of our computer and provided that $G_{M-(m)}$ is computed in a numerically appropriate way (this will require solution of a network or networks with a queue removed, as with Reiser's modifications to Mean Value Analysis).

In summary, the approach we suggest is to use LBANC for the fixed rate and infinite server queues to obtain $G_J(N)$, the normalizing constant for the network of queues 1,...,$J$, and then to finish computation with the

convolution algorithm. (Note that if we wish to use convolution in conjunction with Mean Value Analysis we must compute $G_J$ from the throughputs determined by Mean Value Analysis, and thus lose all the advantages of Mean Value Analysis over LBANC.) If we *really* need queue length distributions, then they should be obtained by equation (5.84) as previously described.

## 5.8 EXERCISES

5.1 For LCFSPR, show that $(S + (c)) - (c) = S$ for any $S$ and any $c$ and also show that $(S - (c)) + (c) = S$ for any $S$ and $c$ where $S - (c)$ is defined.

5.2 For FCFS show that $(S + (c)) - (c)$ is defined and is equal to $S$ if and only if $S = ()$ or $S = (c_1,...,c_n)$ where $c_1 = ... = c_n = c$. Similarly show that $(S - (c)) + (c)$ is defined only if $c_1 = ... = c_n = c$.

5.3 Prove the feasibility and reachability statements in Section 5.2.2.1.

5.4 Prove that if queue $m$ has a LCFSPR discipline, satisfies local balance and has classes $c_1,...,c_k$, then by suitably interconnecting the classes, any $k$ stage service time can be modeled.

5.5 Prove that queues with LCFSPR, PS or IS with an arbitrary number of classes satisfy local balance.

5.6 Show that queues with FCFS and a *single* exponential class satisfy local balance.

5.7 Suggest a discipline which satisfies local balance other than the ones discussed here (it need not be a practical discipline).

5.8 Verify equation (5.27).

5.9 Show that a network such as the one of Figure 5.5 has the same queue length distribution as for a local balance network, provided that for each FCFS queue all classes of the queue have the same exponential service time distribution.

5.10 Verify equation (5.74).

5.11 In the multiple chain version of LBANC (and Mean Value Analysis) if we use the obvious sequence of chain population values (i.e., nested loops over the chain populations) then the storage required is roughly proportional to the product of the chain populations if we don't allow variable rate queues. How might we use a different sequence of chain populations to obtain storage requirements roughly proportional to the sum of the chain populations?

5.12 Extend the results of Section 5.7.2.2 to mixed networks with multiple closed chains.

5.13 Provide the definition of $\tilde{X}_{(m)}$ for queues with two fixed rate servers with equal rates.

5.14 Provide the definition of $\tilde{X}_{(m)}$ for variable rate queues.

5.15 How may we handle mixed networks with Mean Value Analysis? Restate equation (5.147) in a form suitable for Mean Value Analysis.

5.16 Verify that the Scaling Algorithm of Section 5.7.3 terminates.

5.17 Modify the Scaling Algorithm of Section 5.7.3 to allow $I = 0$ and to allow variable rate queues.

5.18 How would you predict the final value of $D$ without running the Scaling Algorithm of Section 5.7.3?

5.19 Suppose we are not going to use LBANC in conjunction with another algorithm. How can we then eliminate the overflow and overflow problem which would occur with the example model with think time 300 seconds?

## 5.9 SUMMARY OF CHAPTER NOTATION

| | |
|---|---|
| $M$ | Number of queues |
| $G$ | Normalizing constant |
| $R$ | Poisson arrival rate (throughput if queue not saturated) |
| $a$ | Mean service rate, i.e., $1/a$ is mean service time |
| $U$ | Sections 5.1-5.5: Throughput times mean service time. Sections 5.6-5.7: utilization |
| $L$ | Mean queue length |
| $l$ | Unnormalized mean queue length |
| $Q$ | Mean queueing time |
| $p_{ij}$ | Probability of visiting class $j$ after visiting class $i$ |
| $C$ | Number of classes |
| $r$ | Relative throughput |
| $S$ | System state |
| $A$ | Number of closed chains in a mixed network |
| $\mathscr{C}$ | Set of classes |
| $N$ | Chain population |
| $e_k$ | Vector with 1 in $k^{th}$ position, 0 elsewhere |
| $p(S)$ | Unnormalized probability of state $S$ |
| CAP | Service capacity function |
| SHARE | A job's share of service capacity function |
| $X$ | Normalization constant for queue by itself |

# *CHAPTER 6*

# APPROXIMATION

Though product form queueing networks are quite useful, detailed computer system models will often have characteristics which violate product form conditions. If a model does not have a product form solution, and if it has too many Markov states for a numerical solution to be feasible, then we must use approximations or simulation (Chapter 7). The most important approach to approximation is *aggregation.* We have already discussed some aspects of aggregation from the point of view of product form networks. In this chapter we will further discuss aggregation from that point of view and consider its application to approximate solution. We will also discuss two other approaches to approximation, *diffusion* and heuristic extensions to LBANC and the mean value analysis algorithm discussed in the last chapter. We will be principally concerned with closed networks, but much of what we say can be applied to open networks.

## 6.1 INTRODUCTION

In aggregation we replace a subnetwork by a single queue with queue length dependent service rates (or, equivalently, service times). This *composite* queue is intended to behave as the entire subnetwork in its interaction with the remaining queues of the network. (This is the motivation for our discussion of composite queues in Section 5.4. Our discussion here is compatible with that one, but may *seem* different because we will be using an alternate representation for a special case. The seeming difference should disappear with our discussion in Section 6.3.2.) What we are trying to do is make the composite queue *flow-equivalent* to the subnetwork in the sense that job flow through the composite queue is equivalent to job flow through the subnetwork. See Figure 6.1. The resulting network, i.e., the network with the composite queue, will usually have fewer states and may have few enough states to be solved numerically. If not, the aggregation process can be repeated until the resulting network has a tractable solution. This aggregation process can be performed exactly for product form networks [CHAN75, VANT78, COUR78, SAUE79a] and limiting cases of some other networks [COUR75, COUR77]. As such, aggregation is useful in parametric analysis. For example, if we wish to study a wide range of CPU parameters, it will take less effort to repeatedly solve the second network of Figure 6.1 than the first. However, our principal interest in aggregation is in approximate solution methods. These approximate methods are strictly *heuristic* in the sense that we cannot defend them formally.

By basing our approximations on methods which are exact for product form networks (and limiting cases of some other networks), we can expect little error to be introduced for networks which are "similar" to those networks. However, it is exceedingly difficult to characterize the error since characterization of the error implies a solution for a (presumably) unsolvable network. In general, we can only provide empirical evidence that the error is small. Approximations are still very attractive because they are usually computationally inexpensive. (Simulation is usually computationally expensive.)

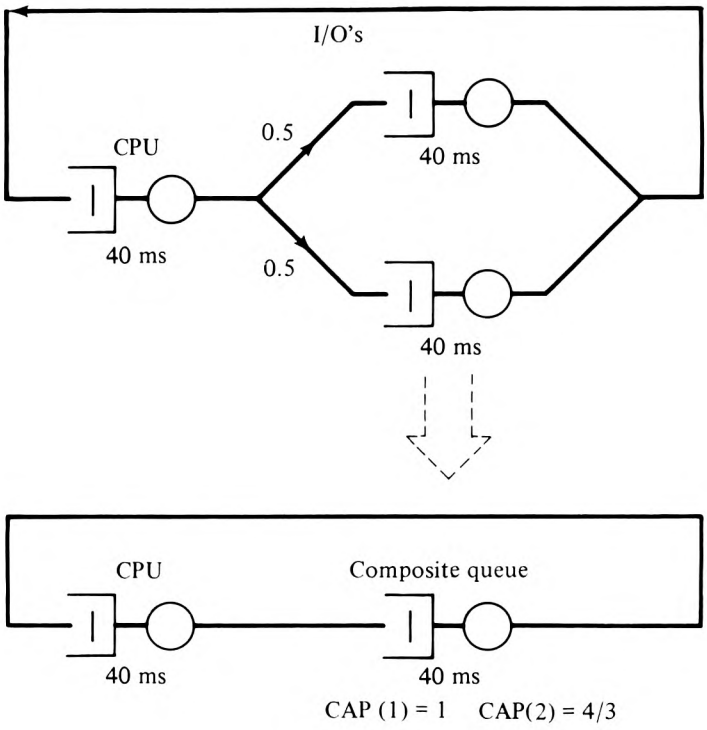



Figure 6.1

If the network of Figure 6.1 satisfies product form, then with two jobs in the network the CPU utilization will be 72.7%. However, if the CPU scheduling is FCFS and the service time distribution has greater variance than the exponential distribution, then we would expect lower CPU utilization. How much lower? If the coefficient of variation is 5, and we use the branching Erlang form of Section 3.4 to represent the distribution, then the CPU utilization of the first network is 67.1%. Thus even for this trivial network, if we assume product form to simplify the solution the relative error is over 8%. However, if we use aggregation and find the CPU utilization of the second network, the result is 67.1%; there is no error in the first three digits. (In general, e.g., for $N > 2$, we would expect FCFS with non-exponential service times to cause a greater deviation from product

form results, and would expect aggregation to be only an approximation, i.e., to cause a more noticeable deviation from the exact results.)

How do we obtain the composite queue characteristics? We could give it a service time equal to the expected time spent in service in the subnetwork it replaces. However, this estimate may be much too high if several jobs may be in service simultaneously in the subnetwork. We could propose alternative estimates but would quickly recognize that any characterization of the composite queue which ignores congestion is likely to be unsatisfactory. One characterization which is exact for product form networks is to let the composite queue have a service rate which depends on the queue length. The service rate for a given queue length is the throughput in a network with corresponding population, where the network is obtained by connecting the output of the subnetwork to its input. See Figure 6.2. (We state this without proof; the proof should be trivial for the reader who has understood Sections 5.1-5.5.) For our example, the throughput with one job in Figure 6.2 is 25 jobs per second and the throughput with two jobs is 33.3 jobs per second. Using our characterization of queue length dependent queues from the last chapter, i.e., characterizing a queue in terms of mean service rate a for queue length one and a capacity function CAP($n$), we obtain the parameters for the composite queue of Figure 6.1, $a^{-1}$=40 ms., CAP(1)=1 and CAP(2)=4/3.

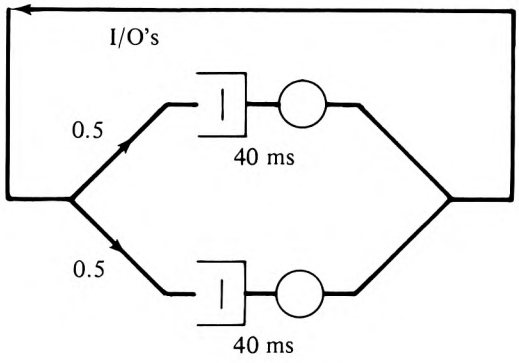



Figure 6.2

Of course, for either model of Figure 6.1, with two jobs the numerical solution is computationally trivial. However, if we increase the number of I/O's and/or the number of jobs while retaining the non-exponential service times and FCFS scheduling, then the number of Markov states quickly exceeds the limitations of the numerical methods of Section 3.3 for the first model. (Recall that we said the iterative methods were only practical for a few thousand states or less. The recursive methods of Section 3.5 do not easily apply to networks where a job can visit more that two queues.) The

| I/O's | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Jobs |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | |
| 2 | 4 | 9 | 16 | 25 | 36 | 49 | 64 | 81 | |
| 3 | 5 | 14 | 30 | 55 | 91 | 140 | 204 | 285 | |
| 4 | 6 | 20 | 50 | 105 | 196 | 336 | 540 | 825 | |
| 5 | 7 | 27 | 77 | 182 | 378 | 714 | 1254 | 2079 | |
| 6 | 8 | 35 | 112 | 294 | 672 | 1386 | 2640 | 4719 | |
| 7 | 9 | 44 | 156 | 450 | 1122 | 2508 | 5148 | 9867 | |
| 8 | 10 | 54 | 210 | 660 | 1782 | 4290 | 9438 | 19305 | |

Table 6.1
Numbers of States for Hyperexponential Central Server Model

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Jobs |
|---|---|---|---|---|---|---|---|---|
| 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | |

Table 6.2
Numbers of States for Aggregation of Hyperexponential C.S.M.

model resulting from aggregation remains trivial, regardless of the number of I/O's, even for very large numbers of jobs. See tables 6.1 and 6.2.

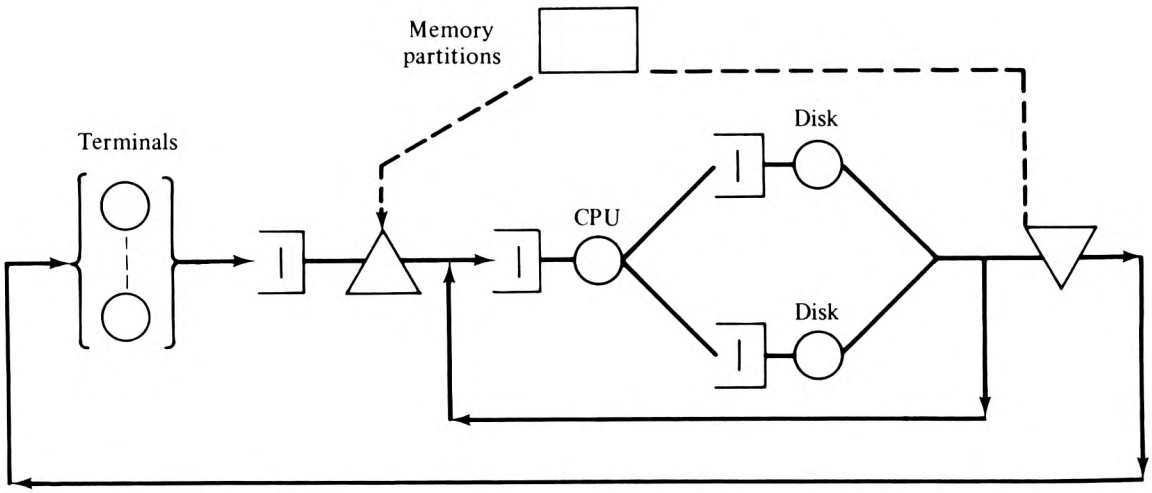



Figure 6.3

As another example, consider the queueing network model of Figure 6.3. This represents an interactive computer system. A user at a terminal submits a command. Processing of the command requires memory. Once memory is allocated, the processing requires alternate CPU and I/O activity until processing is complete, memory can be released, the user reacts to the response and the user submits a new command. This is a simplification of a

very common computer system model as we suggested in Chapter 1 and will discuss in detail in Chapter 9; for other examples see BRAN74 and BOYS75. The inclusion of the memory queue in this network will violate product form except in limiting cases (e.g., where there is only enough memory for one command's processing, where there is no memory contention or where the number of CPU-I/O cycles tends to infinity).

Let us consider a model as suggested by Figure 6.3 with homogeneous jobs, where each job requires exactly one memory partition in order to use the CPU or a disk. Even if we make the strongest assumptions possible, i.e., FCFS scheduling and exponential service time distributions at all queues, the number of states will be quite large if we have more than a few jobs, partitions and/or disks. Table 6.3 gives the numbers of states under these assumptions for a system with 4 disk queues. The combinatorial explosion of the set of states is much worse with more disk queues. Clearly, an exact solution for this model is not reasonable.

| Jobs | 2 | 4 | 6 | 8 | 10 | 12 Partitions |
|---|---|---|---|---|---|---|
| 5 | 66 | 196 | 252 | 252 | 252 | 252 |
| 10 | 141 | 546 | 1302 | 2277 | 3003 | 3003 |
| 15 | 216 | 896 | 2352 | 4752 | 8008 | 11648 |
| 20 | 291 | 1246 | 3402 | 7227 | 13013 | 20748 |
| 25 | 366 | 1596 | 4452 | 9702 | 18018 | 29848 |
| 30 | 441 | 1946 | 5502 | 12177 | 23023 | 38948 |
| 35 | 516 | 2296 | 6552 | 14652 | 28028 | 48048 |
| 40 | 591 | 2646 | 7602 | 17127 | 33033 | 57148 |
| 45 | 666 | 2996 | 8652 | 19602 | 38038 | 66248 |
| 50 | 741 | 3346 | 9702 | 22077 | 43043 | 75348 |

Table 6.3
Numbers of States for Extended Central Server Model

The usual approach to aggregation of this model is to collect the CPU and I/O queues together into a composite queue. See Figure 6.4. Solution of the network after aggregation is then much simpler than solution of the original network. (Error will usually be introduced in the aggregation process. However, empirical studies [BOYS75, KELL76, BROW77, SAUE80a] and analytic studies [BRAN74] show the error to usually be small.) The solution of the network of Figure 6.4 will be computationally inexpensive except in extreme cases [SAUE80b]. In the case where all jobs (commands) are considered homogeneous, one can modify the composite queue service rates to reflect the memory contention. The resulting two queue network (Figure 6.5) may then satisfy product form.

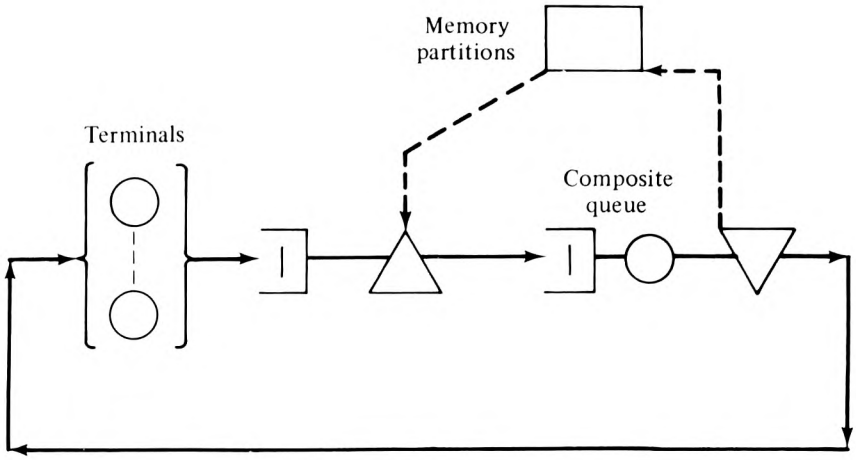



Figure 6.4

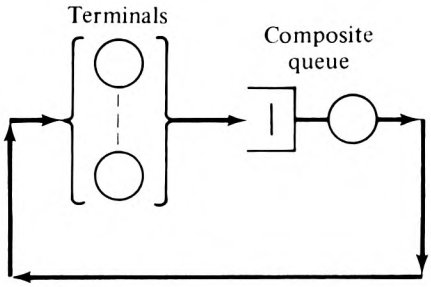



Figure 6.5

How do we obtain the composite queue characteristics? Consider the network of Figure 6.6, which is in a sense the complement of Figure 6.4 with respect to Figure 6.3, i.e., it is obtained by removing from Figure 6.3 all queues to be represented by the composite queue in Figure 6.4, i.e., the CPU and I/O queues. The flow-equivalent approach would solve the network of Figure 6.6 *for each possible population of jobs in the corresponding subnetwork of the original network*. The throughput of jobs through the outer loop of Figure 6.6 (corresponding to the loop through the terminals and memory queues of Figure 6.3) for a given population is used as the service rate of the composite queue when it has the corresponding number of jobs at the queue. It should be apparent that this is the same procedure we went through for the model of Figure 6.1. As we said before, these steps, plus the steps below, would provide an exact solution of the network of Figure 6.3 in the limiting cases, e.g., no memory contention.

A solution of the network of Figure 6.4 or Figure 6.5 will usually yield at least the throughput of jobs through that network and the queue length distributions. Using Little's Rule we can obtain the mean queueing times as

the mean queue lengths divided by the throughput. The mean queueing time for the memory queue in Figure 6.4 or the composite queue in Figure 6.5 will correspond to mean response time in the modelled system. We can obtain the throughputs and utilizations for the individual queues of Figure 6.6 from the throughput through the composite queue and the knowledge of relative throughputs and service times in the network of Figure 6.6. We can obtain the queue length distribution for the individual queues of Figure 6.6 as weighted sums of the queue length distributions for each possible population in that network, with the weights being the queue length distribution of the composite queue of Figure 6.4 (or an appropriate function of the queue length distribution of the composite queue of Figure 6.5). We will discuss these steps in detail in Section 6.3.3.

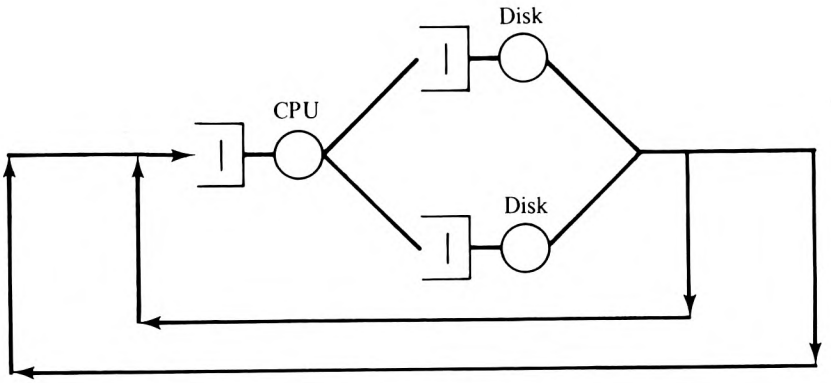



Figure 6.6

| Queue | $U$ | $R$ | $L$ | $Q$ |
|---|---|---|---|---|
| Terminals | .49 | 1.65 | 4.93 | 3.00 |
| Memory | .90 | 1.65 | 5.07 | 3.08 |
| CPU | .82 | 16.4 | 1.98 | .12 |
| Disk | .49 | 8.22 | .82 | .10 |
| Disk | .49 | 8.22 | .82 | .10 |

Table 6.4

As a numerical example consider the network of Figure 6.3 with two disks with the following parameters: $a_{\text{TERMINALS}}^{-1} = 3$, $a_{\text{CPU}}^{-1} = .05$, $a_{\text{DISK}}^{-1} = .06$, $p_{\text{CPU,DISK}} = .5$, $p_{\text{DISK,TERMINALS}} = .1$, and $p_{\text{DISK,CPU}} = .9$. All times are in seconds. The disks are identical. There are 10 jobs and 4 memory partitions. Each job requires one partition to use the CPU or a disk. All queueing disciplines are FCFS and all service times are exponential. (The parameters are such that there are only 125 Markov states for the model, so approximation is not necessary. However, since this model violates product form conditions, it is more convenient to use flow-

equivalent approximation or simulation to obtain a solution than to obtain an exact numerical solution. The parameters are chosen partly so that the approximate solution may be performed by hand, if the reader wishes.) The terminals queue is an infinite server queue. The model of Figure 6.6 satisfies product form conditions, and thus its solution is trivial by the methods of Chapter 5. From these methods we determine the throughput through the outer loop of Figure 6.6 is 0.909 with 1 job in that network. For 2 jobs the outer loop throughput in Figure 6.6 is 1.341, for 3 jobs 1.583 and for 4 jobs 1.730. For the flow-equivalent approximation the following values are used for the "composite queue" of Figure 6.5: $a = 0.909$, and

$$\text{CAP}(n) = \begin{cases} 1, & n = 1, \\ 1.475, & n = 2, \\ 1.741, & n = 3, \\ 1.903, & 4 \leq n \leq 10. \end{cases}$$

($a\text{CAP}(n)$, $1 \leq n \leq 4$, is the throughput through the outer loop of Figure 6.6 with n jobs in that network. Since no more than four jobs can be in memory at once, we heuristically consider the memory contention by letting $\text{CAP}(n) = \text{CAP}(4)$ for $n \geq 4$.) Table 6.4 gives the results from the aggregation approximations. All values are mean values. Utilizations for the terminals and memory are for each terminal and partition, respectively. Simulation results for this model will be given in Chapter 7; we note now that all results agree well with simulation.

## 6.2 SYSTEM CHARACTERISTICS WHICH SUGGEST APPROXIMATE SOLUTION

In previous chapters we have outlined the class of models which can be solved exactly. In this chapter we attempt to motivate the reader to consider the use of approximations. Models which represent systems very realistically often do not have exact solutions. There are at least two immediate questions: (1) Are the performance estimates of the more realistic model significantly different than those of the less realistic model? (2) If so, is the error introduced by approximate solution less than the error introduced by simplistic assumptions? These questions can only be answered empirically. However, if a model ignores the existence of a resource, that model cannot be used to design or schedule the resource. For example, if memory contention is ignored in a computer system model, we cannot use the model to determine the effect on performance of the amount of memory available. If we wish to evaluate memory contention effects, then they must be considered in our model.

The following are system characteristics which suggest the use of approximation because they are likely to significantly affect performance

and because models with these characteristics have been solved by approximations with acceptable accuracy.

### 6.2.1 Multiple Resource Holding

A job may hold more than one resource at a time as in our second example above. When a job is holding more than one resource, usually one resource dominates the other resources held in the sense that the job's activity with the dominant resource determines how long the other resources are held. In the example, the CPU and I/O activity dominates possession of memory (though a job does not simultaneously hold the CPU and I/O resources in this model). In a situation such as this the dominant resources are referred to as *active* resources and the others are referred to as *passive* resources. (We will refer to active and passive *queues* according to the type of resource associated with the queue. For example, we refer to the memory queue of Figure 6.3 as a passive queue and to all the other queues of that Figure as active queues.) Flow-equivalent approximation may be used hierarchically when a job holds several passive resources. A principal difference between active and passive resources in models is that active resources have service time distributions associated with them but passive resources do not. Channels, controllers and peripheral processors in I/O systems are examples of resources often treated as passive.

### 6.2.2 Blocking

The model above allows arbitrarily long queues for resources. However in some systems there are bounds on queue lengths such that a job which no longer needs a resource holds it anyway because it is *blocked* from joining a queue which has reached its bound. Blocking is common in communication systems [KLEI76, KOBA78]. Blocking is a difficult problem to deal with except in simple networks; we will not attempt to consider solution of networks with blocking. For an example of a blocking problem solved by approximation, see LAM76 and CHAN78.

### 6.2.3 Parallelism

Many operating systems allow a job to spawn subservient processes which progress in parallel with the spawning job and may require additional (possibly entirely different) resources. For example, a job may attempt to overlap CPU and I/O activity by spawning a task to perform I/O while it continues computation. Flow-equivalence approximation techniques are relatively easily applied to such systems once one has a solution method for the model corresponding to Figure 6.5. Since these problems are usually not considered to have significant impact on general purpose computer systems, we refer the reader to TOWS78. In a communication system a

message may be split into several *packets* which are transmitted independently, perhaps on different communication links. For discussion of this problem, see KLEI76.

### 6.2.4 Distributions and Disciplines

If service times in the model do not have an exponential probability distribution and the scheduling discipline is one such as FCFS which does not result in product form solutions with non-exponential distributions, then we may notice significant differences in performance measures if exponential distributions and/or product form scheduling disciplines are assumed. This is illustrated even in the trivial example at the beginning of this chapter. Usually CPU service times are quite different from the exponential distribution and CPU scheduling is complex with time-slicing and priority effects. However, results for product form disciplines and other simple disciplines (e.g., FCFS) can often be used to bound the effects of a more complex scheduler. For example, consider a round robin fixed quantum scheduler. As the quantum gets arbitrarily large, the scheduling becomes FCFS. If there is no switching overhead, then the limiting case as the quantum tends to zero is processor sharing, a product form discipline. Thus FCFS and processor sharing can be used to bound the effects of a round robin scheduler, provided the switching overhead is negligible. We have seen in our first example how aggregation may be used to reduce the complexity of solving models with queues which violate product form (i.e., the CPU queue in the example).

### 6.2.5 Routing

Most queueing network models assume that the probability a job will join class $j$ after leaving class $i$ is a constant $p_{ij}$, independent of the state of the system. However, there are systems in which the route that a job takes through a network of queues is designed to depend on the state of the system. For instance, a job requiring the use of a computing system in a multi-computer network may be allowed to use any one of a pool of computers. In this case a reasonable scheduling policy is to direct the job toward the computer with the least expected delay; thus the job's path depends upon the relative congestion at different computers. Systems with such *dynamic routing* strategies (also referred to as *load balancing* strategies) sometimes satisfy product form [TOWS75], but usually do not. Diffusion approximations have been successfully applied to some dynamic routing problems, but general approaches to the problem have yet to be devised.

## 6.3 FLOW-EQUIVALENT AGGREGATION

We have illustrated the application of flow-equivalents in two examples and have discussed the system characteristics which suggest approximation. The flow-equivalent technique is conceptually quite simple, i.e., we replace a subnetwork by a queue with queue dependent service rates as in the examples. The principal remaining question is "What heuristics do we use in particular cases?" In this section we try to answer the question by example, by further developing the application of flow-equivalence to models with passive queues and models with non-product form distribution and/or discipline assumptions.

### 6.3.1 Single Chain Equivalents

As usual, things are simple when there is only one closed chain, so we consider that case first. (The transition from single to multiple chains is generally more difficult in networks without product form solution.)

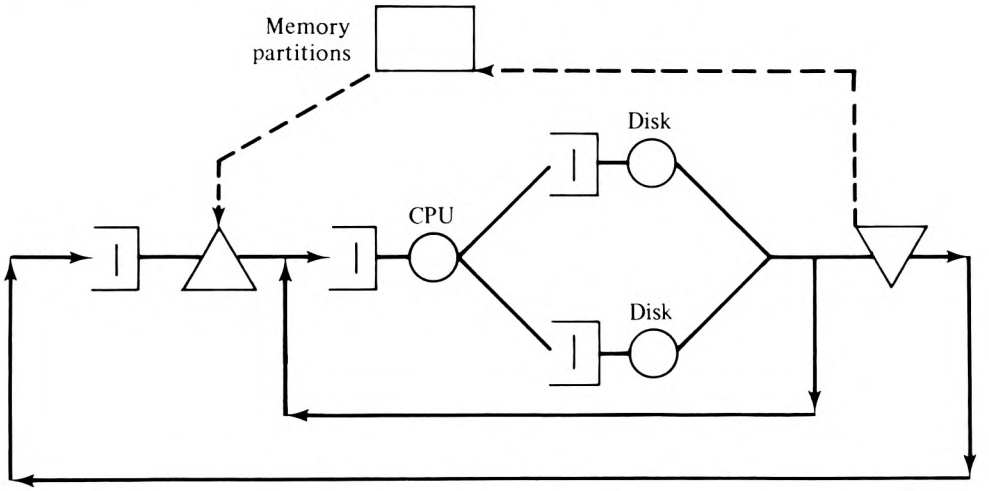



Figure 6.7

**6.3.1.1 Passive Queues.** We assume that each job has a fixed demand for the passive resource, e.g., each job always requires one memory partition. (This assumption is usually made because of the great difficulties encountered without it. However, in the study of BROW77 in Section 9.4 we will not make this assumption.) With these assumptions (single chain, fixed demand), the basic heuristic for passive queues is the one of the numerical example: *If the passive resource limits the population of the subnetwork represented by the composite queue to T, then let* $\mathrm{CAP}(n)=\mathrm{CAP}(T)$ *for* $n > T$. This heuristic allows us to eliminate the passive queue from the reduced model, e.g., we can solve the network of Figure 6.5 rather than the

one of Figure 6.4. An alternate way to view this heuristic is suggested by Figure 6.7. Figure 6.7 is the complement of Figure 6.5 in the same sense that Figure 6.6 is the complement of Figure 6.4. The throughput through the passive queue is limited by the amount of passive resource available; once the resource is fully utilized, adding jobs to the network does not affect the throughput.

As we said, we can repeat the aggregation as necessary. Consider the model of a CDC 6000 series system in Figure 6.8. In addition to the memory contention of Figure 6.3, we also have contention for peripheral (I/O) processors (PP's). A job must have a PP continuously while doing I/O. The PP's are identical; typical operating systems reserve a pool of PP's for user I/O commands. We can begin the solution of this model by solving the model of Figure 6.9 to obtain the throughput through the passive queue. For populations not greater than the number of PP's, we can ignore the PP's and the solution will be trivial if the disk queues satisfy product form. For populations greater than the number of PP's, the additional jobs do not affect the throughput and we use the throughput for the number of jobs equal to the number of PP's. Having the throughputs for the PP/Disk subnetwork, we can (heuristically) replace that by a (composite) queue with $a = R_{PP/Disk}(1)$ and $CAP(n) = R_{PP/Disk}(n)/a$, $n=1,...,N$. We then proceed to the network of Figure 6.10. The process is essentially the same as before. We can determine the throughputs through the memory queue easily if that passive queue is the only characteristic violating product form. Then we can characterize the composite queue of Figure 6.5 to obtain a solution of the aggregate model. The results of this model can then be used to obtain performance estimates of the CPU, PP and Disk queues, as we discussed briefly before and will discuss in detail in Section 6.3.3.

**6.3.1.2 Distributions.** Another major use of aggregation is with models where one or more queues have non-exponential service time distributions and non-local balance queueing disciplines (e.g., FCFS). When such queues are to be explicitly considered in the model after aggregation, as in our first example, then the principal difficulty is in obtaining the numerical solution for the non-product form network, e.g., the second network of Figure 6.1. (That may not be difficult at all, as in the example.) However, when such queues are to coalesce into a composite queue, the situation is conceptually difficult. A number of heuristics have been proposed for this situation, but none of these have such an intuitive defense as the passive queue heuristic above. At least three issues must be faced:

1. How do we estimate the throughputs in the subnetwork to be replaced by the composite queue? This seems to be the most crucial issue.

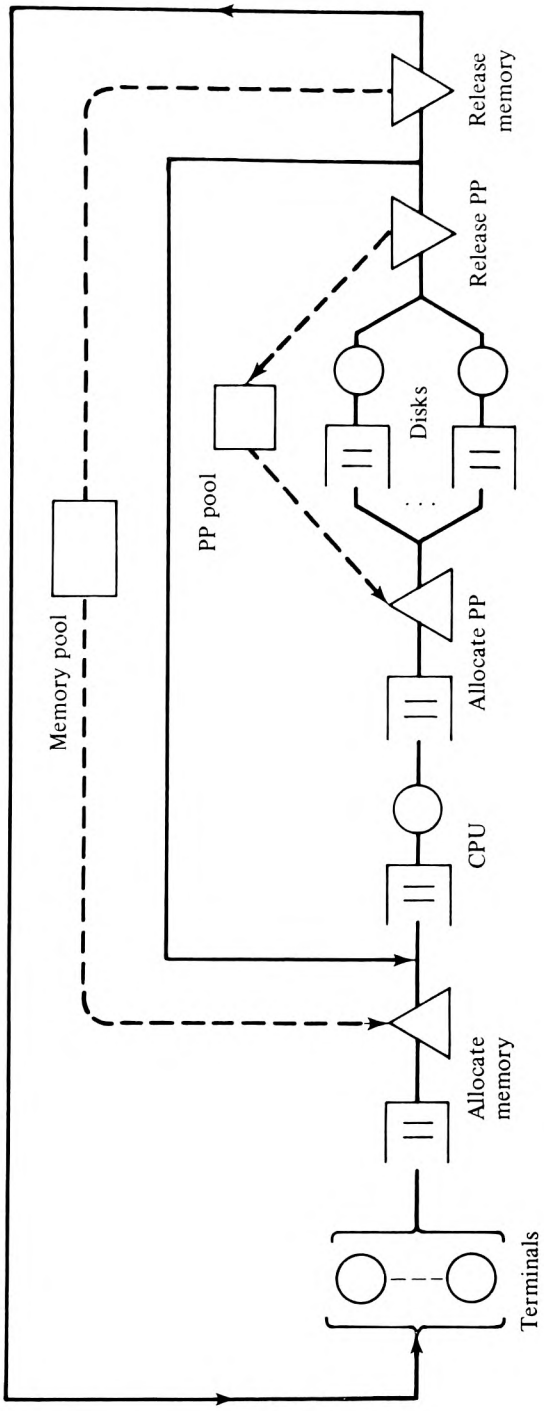

Terminals
Allocate memory
CPU
Allocate PP
Disks
PP pool
Release PP
Release memory
Memory pool

Figure 6.8

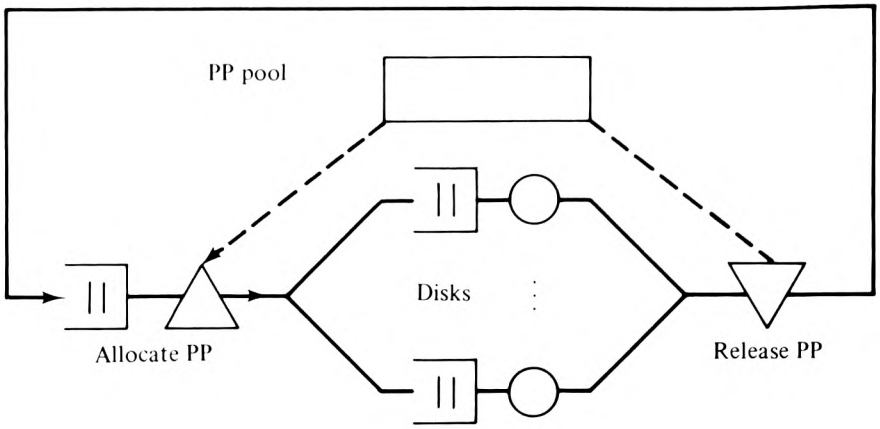



Figure 6.9

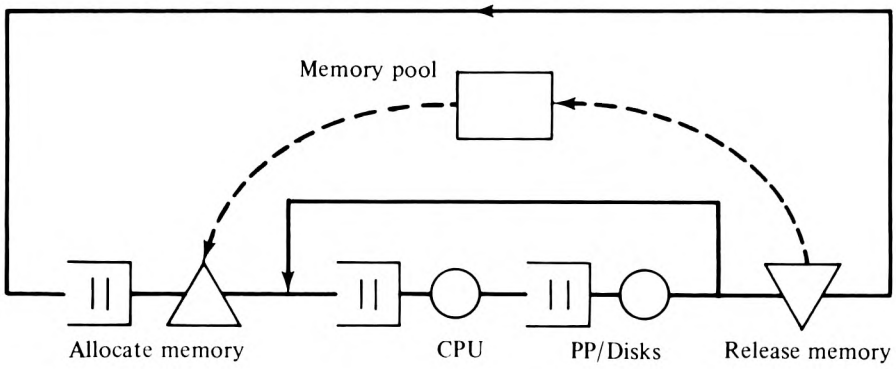



Figure 6.10

2. How do we characterize the service distributions and capacities of the composite queue?
3. How do we characterize the queueing discipline of the composite queue?

**Issue 1:** Since the subnetwork does not satisfy product form and may be large, we must consider alternatives to direct application of the numerical methods of Chapter 3. An obvious, crude alternative is to assume the subnetwork satisfies product form even though it doesn't. The accuracy of this approach depends both on the characteristics of the subnetwork and of the network outside of the subnetwork. In the past iterative refinements have been proposed to overcome the inaccuracy of this approach [CHAN75b]. However, there is no guarantee that such refinements will converge nor that even if the iteration does converge that it will converge to the correct values. Empirical results on single chain networks were promising, but extensions to multiple chain networks would fail unpredictably

[CHAN75b, MACN75]. A more sophisticated approach is to simply use aggregation repeatedly to make the problem tractable, as suggested in ZAHO77. The obvious result is that we recursively face the three issues again, but at least the first issue is easier to face. For example, if we have a numerical solution program for an arbitrarily connected network of two composite queues, then we can partition the subnetwork into pairs of queues, solve each of those resulting subnetworks, replace each of them by composite queues, group the new composite queues in pairs and so on until we have a solution for the original subnetwork. If we can efficiently deal with more than two queues at a time, we may save both computation and accuracy by doing so.

**Issue 2:** This issue is tied to the third issue, for it is only material if we choose a non-local balance discipline (e.g., FCFS) for the composite queue. Assuming this is the case, then we may want to characterize the composite queue by more than just the mean service time ($a^{-1}$) and capacity (CAP($n$)). To be most general it would be appropriate to (re)evaluate remaining service times for all jobs whenever jobs arrive at or depart from the composite queue. However, this has been considered too complex to attempt. Note that the memoryless property of the exponential distribution enormously simplifies (eliminates) the reevaluation of the remaining service times. An attempt at improvement over assuming exponential distributions has been to characterize the distribution of service times independent of congestion and then use a capacity function to consider congestion. Even this is not straightforward, so additional assumptions are made, e.g., that the times will be represented by exponential stages (with the rates of the stages determined in part by the capacity function), that only the mean and coefficient of variation should be considered and that the coefficient of variation is determined by the coefficient of variation of the subnetwork delay when there is only one job in the network. A more thorough treatment of this issue is given in SEVC77b and CHAN78.

**Issue 3:** The queueing discipline of the composite queue does affect the network which contains it. Unfortunately, there has been very little work in the area of selecting queueing disciplines. Queueing disciplines have been selected more to reduce computational complexity than to better represent the subnetwork. Note that a discipline such as processor sharing will likely discard any efforts to characterize distribution form (issue 2); however, a discipline such as FCFS forces a sequencing of jobs which may not have been present in the subnetwork.

### 6.3.2 Multiple Chain Flow Equivalents

Consideration of multiple chains greatly increases complexity. This is not only because the size of the problem grows quickly with the additional

detail (of distinguishing between jobs) but also because of new problems and limitations. The composite queue characterization is a problem; the applicability of the "passive queue heuristic" above is limited to cases where chains are treated differently by the passive queues (or visit separate passive queues).

**6.3.2.1 Composite queue characterization.** The characterization which we have been using for composite queues, $a$ and CAP($n$), is not sufficiently general for multiple chain problems. However, the rate matrix $H$ of Section 5.4 is not directly usable for non-product form networks; i.e., it would be difficult to define the Markov process for a (non-product form) queueing network containing a composite queue with rate matrix $H$. What is usually used is a characterization of the service rates for each class given a specific state of the composite queue, i.e., what we really want is $a_c(S)$ of equation (5.26), the rate at which class $c$ jobs are served when the composite queue is in state $S$. It is important that we recognize that each class of jobs is receiving service *simultaneously*. This is necessary for aggregation to be exact in multiple chain product form networks and considerably restricts our freedom in choosing queueing disciplines for the composite queue, i.e., the scheduling of one class of jobs cannot affect the scheduling of another class. (This is of no consequence in product form networks; in other networks it may be significant. The discussion of issue 3 of the previous Section applies here.)

How do we represent $a_c(S)$ for approximation purposes and how do we obtain its values? Assume that there is exactly one class per chain at the composite queue, as is usually appropriate. Let $a^k(n)$ be the rate at which chain $k$ jobs are served when the composite queue has population vector $n = n_1,\ldots,n_K$, where $K$ is the number of chains. Let $R^k(n)$ be the chain $k$ throughput in the subnetwork to be replaced by the composite queue when the population vector of the subnetwork is $n$. Then $a^k(n) = R^k(n)$. (Proof of this is immediate from equations (5.26-5.29).)

We can now extend the single chain characterization if we let $a^k$ be a scalar equal to $R^k(e_k)$, where $e_k$ is the vector with a 1 in the $k^{th}$ position and 0 elsewhere, and let $\text{CAP}^k(n) = R^k(n)/a^k$, where $\text{CAP}^k(n)$ is the capacity function for chain $k$ jobs when the population vector is $n$.

**6.3.2.2 Passive Queues.** Suppose we wish to obtain an aggregation approximation for a network similar to Figure 6.3 but with two chains. Further suppose that there are effectively separate FCFS memory queues for each chain, e.g., that there is memory dedicated to jobs of each chain. Let $T_k$, $k = 1,2$, be the maximum number of jobs in memory for each chain and let us assume the $a^k$ and $\text{CAP}^k(n)$ representation of the last paragraph. Then, because of the dedication of the passive resources to each chain we

can extend the single chain passive queue heuristic as $CAP^1(n_1,n_2) = CAP^1(\min(n_1,T_1),n_2)$ and $CAP^2(n_1,n_2) = CAP^2(n_1,\min(n_2,T_2))$. However, suppose that the passive resource is not so dedicated, that there is simply a limit T on the number of jobs in memory, still with FCFS scheduling. Then there is no obvious way to successfully extend the single chain heuristic. We know that $CAP^k(n_1,n_2)$ seems reasonable for $n_1 + n_2 \leq T$, but when $n_1 + n_2 > T$ there is no obvious choice for $CAP^k(n_1,n_2)$. I.e., do we choose $CAP^k(0,T)$ or $CAP^k(1,T-1)$ or ... or $CAP^k(T,0)$ or some function of these? The most appropriate approach seems to be not to attempt to use the "passive queue heuristic" but instead to ignore the passive queue in the subnetwork, e.g., use the aggregation suggested by Figures 6.4 and 6.6 rather than Figures 6.5 and 6.7. This approach can be used successfully, but it may be computationally expensive for an approximation [SAUE80a, SAUE80b].

**6.3.2.3 Aggregation of Chains.** Even after aggregation of queues ultimately resulting in a two queue network, with multiple chains the numerical solution of that network may still be expensive or infeasible. For example, if one of the queues is not a composite queue but simply a FCFS queue, then an exact numerical solution would require a state for each possible ordering of jobs at the queue. Thus the number of states might be enormous even with two chains with large populations and certainly would be so with several such chains. One reason for considering multiple chains is priority scheduling; with priority scheduling there are fewer possible orderings, but the state space may still be unwieldy with moderate numbers of chains (e.g., five). In such cases we may attempt aggregation of chains as well as aggregation of queues. In aggregation of queues one replaces several queues by a composite queue that is approximately flow-equivalent as far as the other queues are concerned. In aggregation of chains one replaces several chains by a composite queue that is approximately equivalent as far as the remaining chains are concerned.

Unfortunately, aggregation of chains is not exact even for product form networks except in limiting cases, so there has been no formal basis for aggregation of chains. The population of the "composite chain" is simply the sum of the populations of the component chains, but how do we choose service time distributions, routing probabilities and priorities? One approach is to use weighted sums. In SAUE75b the weights were obtained from throughputs in a product form network as similar as possible to the given network. MacNair and Woo report better results from simply using the relative populations, i.e., the weight for a given chain is its population divided by the composite chain population [MACN75]. Some of the approaches of REIS78a and REIS78c provide an alternative to aggregation of chains; these approaches may also be used with LBANC.

### 6.3.3 Individual Queue Performance Measures

Now that we have dealt with the principal question, we look at the *decomposition* side of the problem, i.e., once we have obtained the aggregate solution, how do we obtain solutions for individual queues? We describe an approach that is exact for product form networks and extends directly to general networks. We assume, without loss of generality, that the aggregation process has yielded the solution of a two queue network, of which one queue was queue $m$ in the original network and the other is a composite queue representing the remaining queues of the original network. For example, such networks would be the second network of Figure 6.1 with queue $m$ being the CPU queue or the network of Figure 6.5 with queue $m$ being the terminals queue. Extension to networks such as the one of Figure 6.4 or with multiple composite queues is straightforward and left to the reader.

As usual, we consider single chain networks first. We assume that we are given the queue length distributions for queue $m$ in the two queue network $P_{(m)}(n \mid N)$ (which is also the queue length distribution for queue $m$ in the original network). Also, we have $R_{(m)}(N)$, the throughput through queue $m$ in the two queue network and the original network and $L_{(i)}^{M-(m)}(N)$, $i \neq m$, the mean queue length for queue $i$ in the network with queue $m$ omitted, e.g., the network of Figure 6.2, with population $N$.

For throughputs we know that

$$R_{(i)}(N) = \frac{r_{(i)}}{r_{(m)}} R_{(m)}(N), \; i = 1,...,\mathrm{M}. \tag{6.1}$$

The extension to multiple chains is straightforward. Having throughputs, we can usually apply equation (2.7) to obtain utilizations.

To obtain utilizations when equation (2.7) does not apply it is necessary obtain queue length distributions. For this reason, and to justify our method for obtaining mean queue lengths, we show how to obtain the queue length distribution for queue $i$ in the original network. Note that this is only necessary when the distribution is directly required or equation (2.7) does not apply. In this case we also need $P_{(i)}^{M-(m)}(n \mid N)$, $i \neq m$, the queue length distribution for queue $i$ in the network with queue $m$ omitted. Then we have the following theorem.

**Theorem 6.1:**

$$P_{(i)}(n \mid N) = \sum_{j=n}^{N} P_{(m)}(N - j \mid N) P_{(i)}^{M-(m)}(n \mid j), \; 0 \leq n \leq N. \tag{6.2}$$

Note that $P_{(m)}(N - j \mid N)$ is the probability the number of jobs in the composite queue is $j$ and is thus the probability that the total number of jobs in queues other than m is $j$.

**Proof:**

Using equation (5.88) and letting $G_{M-(m),(i)}$ be the normalizing constant vector for the network with both queues $m$ and $i$ omitted, we have

$$\sum_{j=n}^{N} P_{(m)}(N - j \mid N) P_{(i)}^{M-(m)}(n \mid j)$$

$$= \sum_{j=n}^{N} \frac{X_{(m)}(N - j) G_{M-(m)}(j)}{G_M(N)} \frac{X_{(i)}(n) G_{M-(m),(i)}(j - n)}{G_{M-(m)}(j)}$$

$$= \frac{X_{(i)}(n) \sum_{j=n}^{N} X_{(m)}(N - j) G_{M-(m),(i)}(j - n)}{G_M(N)}$$

$$= \frac{X_{(i)}(n) G_{M-(i)}(N - n)}{G_M(N)}$$

$$= P_{(i)}(n \mid N).$$

Thus we can obtain the queue length distributions for an individual queue in the subnetwork represented by the composite queue as a weighted sum of the distributions for each possible population, where the weights are the composite queue length distribution.

However, the following result allows us to bypass obtaining the individual queue length distributions unless we really want them.

**Theorem 6.2:**

$$L_{(i)}(N) = \sum_{n=1}^{N} P_{(m)}(N - n \mid N) L_{(i)}^{M-(m)}(n). \tag{6.3}$$

In words, the mean queue length in the original network is obtained as the weighted sum of the queue lengths in the network with queue $m$ omitted.

**Proof:**

$$\sum_{n=1}^{N} P_{(m)}(N - n \mid N) L_{(i)}^{M-(m)}(n)$$

$$= \sum_{n=1}^{N} \frac{X_{(m)}(N - n) G_{M-(m)}(n)}{G_M(N)} \sum_{j=1}^{n} j \frac{X_{(i)}(j) G_{M-(m),(i)}(n - j)}{G_{M-(m)}(n)}$$

$$= \sum_{n=1}^{N} \sum_{j=1}^{n} \frac{j X_{(i)}(j) X_{(m)}(N - n) G_{M-(m),(i)}(n - j)}{G_M(N)}$$

$$= \sum_{j=1}^{N} \sum_{n=j}^{N} \frac{j X_{(i)}(j) X_{(m)}(N - n) G_{M-(m),(i)}(n - j)}{G_M(N)}$$

$$= L_{(i)}(N).$$

Both of these theorems extend directly to multiple chains; most importantly, equation (6.3) becomes

$$L_{(k,i)}(N) = \sum_{n_1=0}^{N_1} \cdots \sum_{n_K=0}^{N_K} P_{(m)}(N_1 - n_1, \ldots, N_K - n_K \mid N_1, \ldots, N_K) L_{(k,i)}^{M-(m)}(n_1, \ldots, n_K). \tag{6.4}$$

We leave it to the reader to state and prove the multiple chain version of Theorem 6.1 and to prove equation (6.4).

Having the throughputs and mean queue lengths for the individual queues, we use Little's Rule to obtain the mean queueing times.

### 6.3.4 Limiting Case Justifications for Aggregation

The primary justification we have used for aggregation approximation is the exact aggregation of product form networks. This justification becomes less credible as the network to be solved becomes "less similar" to a product form network. Examples include the number of queues violating product form conditions (e.g., distributions and disciplines) increasing, an individual queue tending to deviate greatly from product form conditions (e.g., service times at a FCFS queue having very small or very large coefficients of variation), and conditions such as multiple resource holding becoming dominant. There is another justification for aggregation, weakly

coupled subnetworks [COUR75, COUR77], which is independent of product form conditions. The product form justification holds regardless of the degree of coupling of subnetworks; the weakly coupled justification holds regardless of product form conditions.

The essence of the weakly coupled justification is that in the limiting case of disjoint subnetworks, aggregation must be exact because the subnetworks do not interact. So if a subnetwork is essentially independent of the remainder of the network, i.e., its internal events are much more frequent and much more dominant in its behavior than its interactions with the rest of the network, then we should introduce little error in replacing it by a composite queue. As an example, consider the network of Figure 6.3. As the number of CPU-I/O cycles tends to infinity, the CPU-I/O subsystem becomes independent of the remainder of the system. Thus if we replace the CPU-I/O subsystem by a composite queue, we expect little error to be introduced. Taking this point of view it is possible to characterize the error introduced by aggregation [COUR77].

## 6.4 APPROXIMATION EXTENSIONS TO LOCAL BALANCE ALGORITHMS

We have emphasized aggregation approximations because they have been fairly widely used and empirically justified, because they are relatively easy to apply and because they are fairly general. Diffusion approximations have been used for quite some time but are principally successful in open networks with heavy traffic. A third, and relatively untested, approach to approximation has recently appeared. Along with the mean value analysis algorithms for product form networks, Reiser and Lavenberg proposed heuristic extensions for non-product form networks in REIS78a. Subsequently, Bard proposed additional extensions for other characteristics violating product form [BARD78b]. These extensions are often much simpler to program and less computationally expensive than the aggregation approximations we have discussed. However, we emphasize that there has been very little empirical justification for these methods. The extensions may be applied to LBANC as well as mean value analysis. Since we prefer LBANC, we will translate some of the extensions of REIS78a, REIS78c and BARD78b to that algorithm. Other mean value analysis extensions, e.g., for reducing computational costs for networks with many closed chains, also translate directly to LBANC. We will discuss another extension, for multiple resource holding, in Section 9.5. We note without further discussion that some of the aggregation approaches may be used together with this approach.

Let us assume that queue $m$ has FCFS scheduling, a single fixed rate server, a single class and a non-exponential service time distribution with

mean $a_{(m)}^{-1}$ and coefficient of variation $\mathrm{CV}_{(m)}$. Using the same arguments as in Section 4.1.2, we would expect the mean queueing time to be

$$\begin{aligned} Q_{(m)}(N) &= a_{(m)}^{-1}(1 + L_{(m)}(N - 1) - U_{(m)}(N - 1)) \\ &\quad + U_{(m)}(N - 1)a_{(m)}^{-1}\frac{1+\mathrm{CV}_{(m)}^2}{2}. \end{aligned} \tag{6.5}$$

The first term represents the expected service times for the jobs not yet served; the second term represents the expected service time for a job in service. Notice that equation (6.5) is equivalent to equation (5.103) for exponential service time distributions. To apply this equation to LBANC, we want an expression for the unnormalized mean queue length $l_{(m)}(N)$. Applying Little's rule, multiplying by G(N) and simplifying, we have

$$l_{(m)}(N) = u_{(m)}\Big( G(N-1) + l_{(m)}(N-1) + u_{(m)}\frac{\mathrm{CV}_{(m)}^2 - 1}{2}\Big). \tag{6.6}$$

Similar arguments for the multiple chain case with one class per chain yield (for $K = 2$)

$$\begin{aligned} &l_{(1,m)}(N_1,N_2) \\ &= u_{(1,m)}G(N_1-1,N_2) + l_{(m)}(N_1-1,N_2)) \\ &\quad + u_{(1,m)}\Big( u_{(1,m)}\frac{\mathrm{CV}_{(1,m)}^2-1}{2} + u_{(2,m)}\frac{\mathrm{CV}_{(2,m)}^2-1}{2}\Big) \end{aligned} \tag{6.7}$$

and

$$\begin{aligned} &l_{(2,m)}(N_1,N_2) \\ &= u_{(2,m)}G(N_1,N_2-1) + l_{(m)}(N_1,N_2-1)) \\ &\quad + u_{(2,m)}\Big( u_{(1,m)}\frac{\mathrm{CV}_{(1,m)}^2-1}{2} + u_{(2,m)}\frac{\mathrm{CV}_{(2,m)}^2-1}{2}\Big). \end{aligned} \tag{6.8}$$

Note that these reduce to our previous equations for FCFS queues where all classes have the same exponential distributions. We emphasize that these tantalizing expressions are *approximations*. For example, it can be shown that characteristics of the service time distribution other than the mean and the coefficient of variation have some effect on performance in closed networks [PRIC76], but equation (6.5) ignores such other characteristics.

## 6.5 DIFFUSION APPROXIMATION

We may think of a diffusion process as a Markov process with a continuous state space. Diffusion approximations use the theory of diffusion processes to analyze queueing problems. The apparently difficult mathematics discourages most analysts from using diffusion approximations. However, it is possible to use the diffusion approximation formulae without understanding their derivation in detail. It is not necessary to understand the derivations provided that empirical evidence justifies the use of the formulae. Diffusion approximations are principally successful in open networks with heavy traffic. For such networks numerical methods are not feasible and, as we shall discuss in Chapter 7, simulations may be quite expensive, thus the importance of diffusion. The only cases where an analyst really needs a thorough understanding of the mathematics of diffusion are when attempting to develop better approximations or to extend the approximations to new problems. We will first discuss diffusion processes from an informal point of view. Then we discuss mapping of queueing problems to diffusion processes.

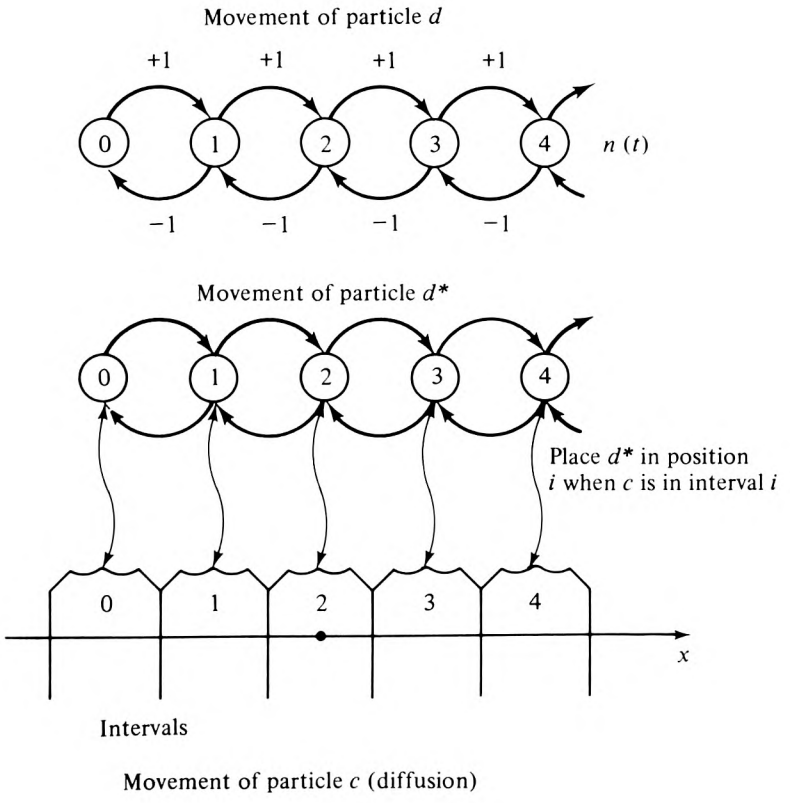



Figure 6.11

Consider a FCFS GI/G/1 queue, i.e., a single server queue fed by a source where service and interarrival times are independent random variables having general distributions. Even mean response time for this system cannot be expressed simply [KLEI75]. Let $n(t)$ be the number of jobs in the queue at time $t$. $n(t)$ can take on the values 0,1,2,... . We may think of $n(t)$ as the position of a particle $d$ (for discrete) that makes a jump (of $+1$) to the right when a job arrives and a jump (of $-1$) to the left when a job departs (see Figure 6.11). In general, the probability that $d$ will make a jump of $+1$ or $-1$ in the next time interval depends upon the past behavior of $d$. For example, the length of time $d$ will stay in its current place depends (partly) upon the time since the last arrival or departure.

Our goal is to deduce the probability of the future behavior of $d$ given its past behavior. The difficulty with predicting the behavior of particle $d$ is that it has memory in addition to its current position, i.e., the state of the system is not merely the particle's current position. (In the GI/G/1 systems we must remember remaining service time and remaining time until the next arrival.) In the diffusion approximation we represent the behavior of particle $d$ approximately by the behavior of a particle $c$ (for continuous) which *has no memory.*

Whereas $d$ can only take on the values 0,1,2,..., we let $c$ take on all values on the non-negative real line. Let us decide how $c$ should move along the real line. Since we are used to dealing with discrete state spaces we shall treat the continuous state space of $c$ as the limiting case of a discrete state space. The informal treatment here is based on the discussion in COX65.

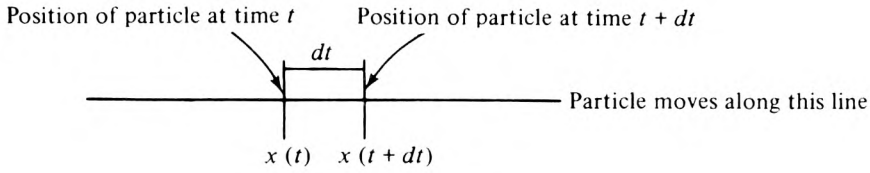



Figure 6.12

Assume that particle $c$ can only move at times 0, $T$, $2T$, $3T$, ..., where $T$ is some small constant time interval. $c$ can only take small steps of magnitude $M$. Thus if we take the limit as $T$ and $M$ approach 0, we see that $c$ moves continuously along the real line. In each time interval $T$ we assume that the particle takes a step $z$ where $z = +M$ with probability $p$ and $z = -M$ with probability $1 - p$. In time $iT$, the total displacement of the particle will be the sum of $i$ independent, identically distributed random variables, each with the same distribution as $z$. As $i$ gets large, the distribution of the displacement approaches that of a *normal* or *Gaussian* random variable. (The normal or Gaussian random variable is extremely important

in simulation. Since some readers will likely skip diffusion approximations, we defer a more detailed discussion of the normal random variable until Section 7.2. The reader may wish to defer reading this section until having read Section 7.2, but this should not be necessary to get an informal understanding of diffusion approximations.)

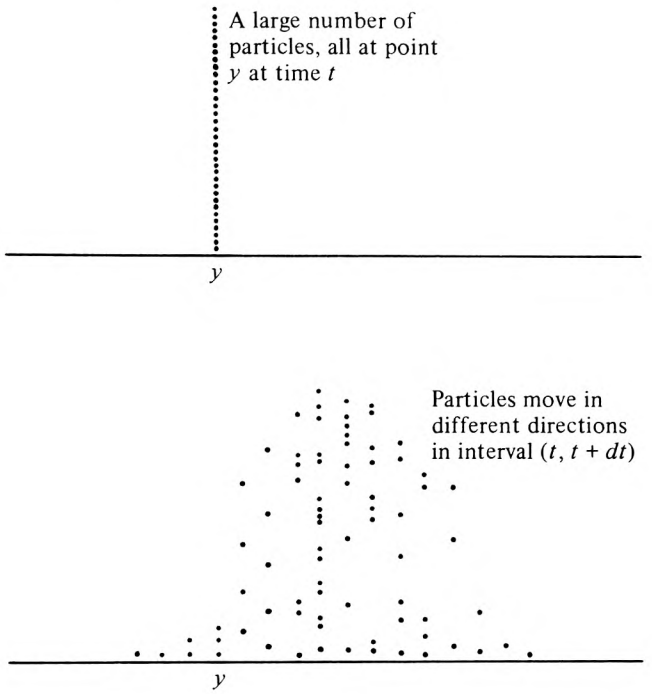



Figure 6.13

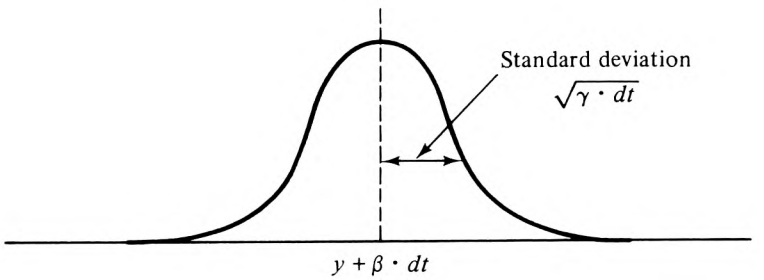



Figure 6.14

Let the position of $c$ at time $t$ be $x(t)$ (see Figure 6.12). Let the displacement of the particle in the interval $[t,t + dt]$ be $dx(t)$ where

$$dx(t) = x(t + dt) - x(t). \tag{6.9}$$

Assume that $dx(t)$ is normally distributed with mean $\beta dt$ and variance $\gamma dt$. To help picture the process, imagine an arbitrarily large number of particles that move according to the above assumptions. Suppose that all the particles are at a point $y$ at time $t$ (see Figure 6.13). Then at time $t + dt$ the particles would have moved, some one way and some in the opposite direction. The function showing the density of particles around a given point at time $t + dt$ has the familiar bell shape of the normal distribution with a mean at $y + \beta dt$ and variance $\gamma dt$. The particle is memoryless in the sense that its future displacement, relative to its current position, is independent of the past. (The particle *does* have memory in the sense that it doesn't cross the boundary into the negative queue length region.) Let $p(x_0,x\ t)$ be the density function for the process $x(t)$ given that $x(0) = x_0$. The density function of the particle has been studied in depth and methods exist for computing it [COX65].

We wish to deduce the behavior of particle $d$ from the behavior of particle $c$. To do this we shall simulate the behavior of $d$ by a particle $d^*$ that also jumps between points 0, 1, 2, ... but whose movement is driven by the movement of $c$ in the following way. Partition the real line into intervals; place $d^*$ in position $i$ when $c$ is in the $i^{th}$ interval. When $c$ moves into the $i - 1^{th}$ (or $i + 1^{th}$) interval, move $d^*$ to position $i - 1$ (or $i + 1$). Statistics regarding $d^*$ are said to be *diffusion approximations* of the corresponding performance measures regarding $d$.

The accuracy with which $d^*$ models $d$ depends upon:

1. How values are assigned to the parameters $\beta$ and $\gamma$ that characterize the diffusion process (and hence characterize the movement of the particle $c$).
2. How the real line is partitioned into intervals.
3. How we place a boundary condition on the diffusion process. Typically we want $c$ to move on the non-negative real line just as $d$ does. There are different conditions we might place at the boundary $x = 0$ to ensure $x(t) \geq 0$ for all $t$. These boundary conditions affect the behavior of $c$ and thus the behavior of $d^*$.

We now consider these issues in turn.

**Setting $\beta$ and $\gamma$**

To make the computation of $\beta$ and $\gamma$ tractable we shall make the (invalid) assumption that the queue is never empty. This assumption is more reasonable in heavy traffic (when the queue approaches saturation) and thus the approximation gives better results under heavy traffic conditions.

Consider a time interval $[t,t + dt]$. Let $n(t)$ be the queue length at time $t$. During this interval the expected number of jobs to arrive is $Rdt$ where $R$ is the arrival rate and the expected number of departures is $adt$ where $a$ is the service rate. Hence

$$E[n(t + dt) - n(t)] = (R - a)dt. \tag{6.10}$$

We want to position $x(t)$ of particle $c$ to reflect the queue length $n(t)$. Note from the earlier discussion that the displacement $x(t + dt) - x(t)$ is a random variable with mean $\beta dt$. Hence, it is reasonable to set

$$\beta = R - a. \tag{6.11}$$

By similar (though more complex) arguments we set

$$\gamma = CV_A^2 R + CV_S^2 a \tag{6.12}$$

where $CV_A$ and $CV_S$ are the coefficients of variation of the interarrival and service times, respectively.

**Selecting Intervals**

A reasonable heuristic is to place $d^*$ in the $i^{th}$ position when $c$ is between $i$ and $i + 1$ as depicted in Figure 6.11. Using this method of selecting intervals,

$$p^*(n_0,n;t) = \int_n^{n+1} p(x_0,x;t)dx. \tag{6.13}$$

**Boundary Conditions**

The *reflective barrier* is the boundary condition normally used [COX65]. This boundary condition states that the particle $c$ must always be on the non-negative portion of the real line. The density function for particle $c$ with this boundary condition is known and we can compute $p^*(n_0,n\ t)$. We are primarily interested in the equilibrium queue length distribution $p^*(n) = p^*(n_0,n\ \infty)$. Using the methods described for setting $\beta$ and $\gamma$ and for selecting intervals, with this boundary condition we get

$$p^*(n) = (1-\hat{U})\hat{U}^n,\ n = 0,1,2,... \tag{6.14}$$

where

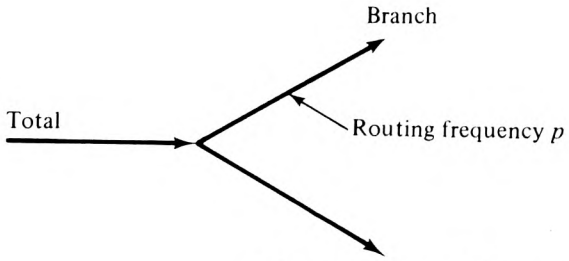
$$\hat{U} = e^{-2(1-U)/(CV_S^2+UCV_A^2)} \tag{6.15}$$

and $U = R/a$ is the utilization. Let $p(n)$ be the (true) equilibrium probability of $n$ jobs in the queue. We know that the fraction of the time the server is idle is $p(0)=1 - U$, whereas $p^*(0) = 1 - \hat{U}$, which is erroneous. A heuristic to deal with this is to let

$$\hat{p}(n) = \begin{cases} 1 - U, & n = 0, \\ U(1 - \hat{U})\hat{U}^{n-1}, & n > 0. \end{cases} \tag{6.16}$$

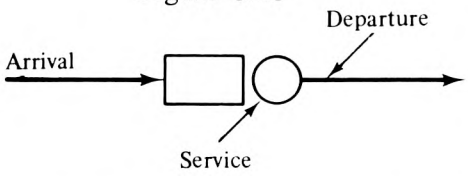
Other boundary conditions have been proposed. For example, Gelenbe reports improved accuracy with a boundary condition where the particle reaching the boundary sticks there for an exponentially distributed time and then jumps back into the region $x > 0$ according to some distribution [GELE75]. For example, the particle might jump from $x = 0$ to $x = 1$, representing an arrival of a new job. The exponentially distributed time the particle is stuck has the same mean as the interarrival distribution.

**Networks**



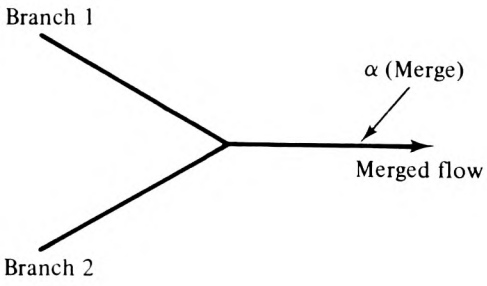

$\alpha$ (Branch) = $p \cdot \alpha$ (TOTAL)

Figure 6.15




$\alpha$ (Departure) = Utilization$^2 \cdot \alpha$ (Service) + (1 − Utilization$^2$) $\cdot \alpha$ (Arrival)

Figure 6.16




$p_i$ = (Branch $i$ flow)/(Merged flow)

$\alpha$ (Merge) = $\Sigma\, p_i^2 \alpha$ (Branch $i$)

Figure 6.17

In an open network of GI/G/1 queues, we can use the diffusion approximation as follows [KOBA74]. We essentially assume that the product form holds, i.e.,

$$\hat{p}(n_1,\ldots,n_M) = \prod_{m=1}^{M} \hat{p}_m(n_m). \tag{6.17}$$

We treat the queues as independent except for determining the coefficient of variation of the interarrival time for each queue. To do that, we first determine the mean interarrival time for each queue from the source rate and the relative throughputs. Then we determine the utilization at each queue. Then we use the following procedure proposed in SEVC77b based in part on earlier work [DISN74, GELE76, KOBA74, REIS74]. For simplicity we assume exactly one class per queue. Let $CV_{A(m)}$ be the coefficient of variation of the interarrival times at queue $m$, and let $\alpha_m = CV^2_{A(m)} - 1$. We determine $\alpha_m$ for each queue according the the equations in Figures 6.15, 6.16 and 6.17 and then determine $CV_{A(m)}$.

Similar approaches have been used for closed networks, but are less successful because of the difficulty in determining throughputs (which would then be used for interarrival distributions). Two ways to determine the throughputs are to either assume product form [KOBA74] or to assume one queue is saturated.

## 6.6 FURTHER READING

For further discussion of aggregation approximations, see CHAN78, COUR77, KOBA78, and MARI79. For further discussion of approximation extensions for mean value analysis see REIS78a, REIS78c and BARD78b. For further discussion of diffusion approximations, see COX65, KOBA74 and FOSC77.

## 6.7 EXERCISES

6.1 In the discussion of Section 6.3.2.2 it is assumed that the passive resource scheduling is FCFS. What if the scheduling was preemptive priority? Non-preemptive priority? Which is more likely to be realistic for passive resource scheduling?

6.2 How would Theorems 6.1 and 6.2 be applied to networks such as the one of Figure 6.4 or with multiple composite queues?

6.3 State and prove the multiple chain version of Theorem 6.1. Prove equation (6.4).

6.4 Justify equations (6.7) and (6.8).

6.5 Restate equations (6.7) and (6.8) for multiple classes per chain.

# *CHAPTER 7*

# SIMULATION

The most popular approach to the solution of a computer system model is to simulate it, i.e., to use a program which behaves like the model and observe the behavior of the program. The principal advantage of simulation is its great generality. There are three main problems with simulation: the expense of constructing a simulation program, the computational expense of running the program, and the statistical analysis of the program behavior. We will give little direct attention to the computational expense of simulation. There exist specific techniques for reducing this expense but the techniques are of a relatively advanced nature. (Some of these specialized techniques are closely related to the flow-equivalent approximations of Chapter 6.)

When a model does not have a product form solution (e.g., because it has some of the characteristics described in Chapter 6 as precluding a product form solution) and is of sufficient size that memory and computational costs of numerical solution are excessive, then the reasonable alternatives are approximations and simulation. Approximations have the advantage of low computational costs but may introduce an unknown amount of error and may be difficult to apply. Simulation will usually be more expensive computationally, but with sufficient computational expense, the error can be made very small and the application will be relatively straightforward. Note that simulation results are usually *not* exact; the error can be reduced by additional computational time in most situations.

Even though the Markov process formalism of Chapters 3-5 is not necessary for simulation, it is helpful to keep this formalism in mind. For one thing, the queueing network representations provide a convenient framework for model formulation. Having an appropriate formulation of the model will help eliminate programming errors, a major source of error in simulation. Perhaps more importantly, the Markov process representation allows us to take advantage of rigorous statistical methods. The most important of these is the *regenerative* method for confidence intervals, which we discuss later in this chapter.

Some of the characteristics violating product form solution conditions are of little consequence with respect to simulation. For example, simulation techniques will be essentially the same whether or not distributions are exponential. (The use of the method of exponential stages may be neces-

sary if we wish to apply the regenerative method.) Use of scheduling disciplines which violate product form introduces little additional complexity. Other characteristics, e.g., simultaneous resource possession, can easily be simulated, but product form queueing networks provide no framework for representing these characteristics. In these cases we find it appropriate to define extensions to the queueing network representation such as the "passive" queues discussed in Chapter 6. We will discuss passive queues from a simulation point of view toward the end of this chapter, and also discuss other extensions to queueing network representations. Our objective is to treat extended queueing networks as a unified approach to modeling, regardless of solution method.

## 7.1 CONSTRUCTION OF SIMULATION PROGRAMS

Our objectives in this section are (1) to give a thorough introduction to random number generation and event list mechanisms, the two programming techniques relatively unique to simulation, (2) to give a very brief introduction to estimation of performance measures, and (3) to give an example simulation program for the cyclic queue model which was solved numerically in Chapter 3.

### 7.1.1 Random Number Generation

One of the central aspects of the models we consider in this text, and most simulation models, is that of random variables characterized by probability distributions. These are used to represent service times, interarrival times and other system characteristics. However, computer systems are designed to be deterministic with respect to individual programs, so we can hardly expect a program to have truly random behavior. We can devise methods for programming apparently random behavior, behavior which appears to be "random" as far as we can determine from statistical tests. Further, we can program this behavior so that the random variables appear to have the intended probability distributions. Since the numbers are actually deterministic, but appear to be random, the term "pseudo-random" is often used.

We are given $F_x(x_0)$, defined to be the probability that a given value of the random variable $x$ is not greater than $x_0$, and we wish to deterministically generate a sequence of values ("samples") which have this probability distribution defined so that every value in the interval (0,1) is equally likely. This is known as the *uniform* distribution (Chapter 2) for this interval and has the probability distribution function

$$F_u(u_0) = \begin{cases} 0, & u_0 \leq 0 \\ u_0, & 0 < u_0 < 1 \\ 1, & u_0 > 1 \end{cases}$$

We are arbitrarily eliminating the endpoints of the interval to avoid taking the logarithm of zero, which is undefined. Given the capability to obtain samples from this distribution and the inverse of the distribution function for random variable $x$, $F_x^{-1}(\ )$, we can obtain a sample of the random variable $x$ as $F_x^{-1}(u_0)$ where $u_0$ is a sample from the above uniform random variable. This is depicted in Figure 7.1 for an exponential random variable. We will return to this general sampling problem after showing how we may obtain samples from the above uniform distribution.

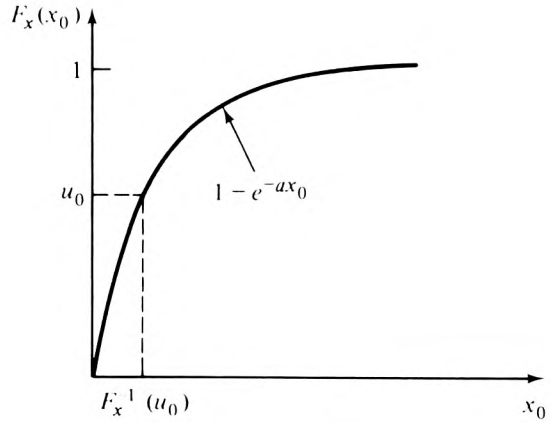



Figure 7.1

There are many possible approaches to obtaining the samples from the uniform distribution. Nearly all known approaches define some function which will determine a new uniform value based on the previously obtained values (or some chosen initial values). Experience has repeatedly demonstrated that one must be very careful in choosing such a "random number generator." As Knuth has said, "... random numbers should not be generated with a method chosen at random." [KNUT68] The chosen function should have been carefully selected and then subjected to rigorous statistical testing to ensure that it has the desired characteristics. We will not attempt to discuss the extensive theoretical foundations of the selection process nor the many statistical tests which have been proposed and applied. We will describe in detail one of the most highly regarded generators, one which can be efficiently implemented on most known computers. In doing so we will try to mention a few of the significant theoretical considerations.

For a variety of reasons, including theoretical tractability and computational efficiency, most random number generators utilize integer representations and arithmetic. Given a random integer in the interval $[1,m-1]$, $m$ an arbitrary positive integer, such that each value in this interval is equally likely, then we can obtain the desired value in the interval (0,1) by dividing by $m$.

The generator we use will be based on a specific case of the following general approach: Given a positive integer $a$, a non-negative integer $b$ and a positive integer initial value $Z_0$, subsequent values are obtained from the expression

$$Z_i = (aZ_{i-1} + b) \text{ modulo } m, \; i = 1,2,3,...$$

(The modulo operation gives the remainder of integer division by $m$.) As we have tried to suggest, the choice of $a$, $b$ and $m$ is critical. Consider $a = b = 1$. The sequence produced will obviously not be satisfactory. However, other choices may be worse, though not *obviously* unsatisfactory. Given appropriate choices of $a$, $b$ and $m$, the choice of $Z_0$ may be more or less arbitrary.

We would like for $m$ to be very large, so a common criterion is that $m$ be nearly equal to the maximum representable integer for a given computer. If there are $p$ bits in the representation of an integer, not counting the sign bit, then usually the maximum integer will be $2^p - 1$ except that double word products and dividends may be allowed. If these double word values are allowed, then $2^p$ will be a very convenient choice for $m$. Suppose $p = 5$ and we choose $m = 2^5 = 32$. Consider $a = 7$, $b = 0$ and $Z_0 = 1$. Then we will have $Z_1 = 7$, $Z_2 = 17$, $Z_3 = 23$, $Z_4 = 1$, $Z_5 = 7$, etc., an obviously unsatisfactory situation. We say that this sequence has a *period* of four. Clearly we would like to obtain a period relatively close to $m$. If we modify the generator to have $a = 5$, then we would get the sequence 1, 5, 25, 29, 17, 21, 9, 13, 1, 5, ... , which has a period of 8, a definite improvement. In fact it is the best we can do with $b = 0$ and $m = 32$. In general, it can be shown that with $p$ greater than 2, $m = 2^p$ and $b = 0$, the maximum period obtainable is $m/4$. If we make a better choice of $b$, then we can obtain the period $m$, but, given our definition of random variable $u$ on the interval (0,1) we will have to discard somehow the $Z$'s $= 0$. Though generators with $m = 2^p$ may have adequate characteristics, we reject this choice of $m$ because the least significant bits of the generated integers will not be very random and for reasons of convenience not related to the generators' statistical properties. We prefer to let $m$ be the largest prime less than $2^p$. For $p = 5$ this would be $m = 31$. For prime $m$, the maximum period will be $m - 1$, with the missing value conveniently being zero, if we make the proper choice of $a$, and if $b = 0$. If a given value of $a$ produces the maxi-

mum period, then $a^c$ will also produce the maximum period, provided that $c$ is less than $m$ and and $m$ is not divisible by $c$. For $m = 31$, $a = 3$ gives the full period. ($3^2$ and $3^3$ do not, but $3^4$ does, also.) The reader may wish to use this choice of $a$ and $m$ to see the period $m - 1$ is obtained, and might also try another value of $a$, say 5, to see that a smaller period is obtained.

There are many other considerations in the choice of $a$ and $m$, most of which are related to properties under statistical tests, and we shall ignore these considerations. The generator we will use was very carefully chosen and has performed well under thorough statistical testing. It was originally designed for the IBM 360 family of computers, but can be efficiently implemented on nearly any computer. We know of no other generator which has been as thoroughly tested, and the great transportability of the algorithm allows us to get similar simulation results on quite different computers.

For the 360 we have $p = 31$. Conveniently, the largest prime less than $2^{31}$ is $2^{31} - 1 = 2147483647$. For this choice of $m$, an appropriate choice of $a$ is $7^5 = 16807$. Note that the computation

$$Z_i = (16807Z_{i-1}) \text{ modulo } 2147483647, \; i = 1,2,3,... \qquad (7.1)$$

can be performed efficiently on any machine allowing 48 bit products and dividends. Even the CDC 6600 and its successors, which have unusual integer arithmetic, allow efficient implementation of this computation.

The generator of (7.1) is widely used and has been implemented in a variety of software packages, including non-IBM software for non-IBM machines. However, like any random number generator, some subtle deviations from desired behavior are indicated by some statistical tests. (For many generators we would have to omit "subtle" from this statement.) A technique that can be used to improve upon almost any generator is called "shuffling," analogous to picking a card from a shuffled deck of cards. A table with $n$ entries ($n$ approximately 100) is initialized with $Z_1, Z_2, ..., Z_n$. Then when the generator is called, we obtain $Z_{n+i}$, use this value to select a table entry and return that table entry. The table entry is replaced by $Z_{n+i}$. (A variation on this approach is to use two sub-generators, one for filling the table entries and one for picking the table entry.) The generator LLRANDOM [LEAR73] uses $n = 128$ and uses the least significant 7 bits of $Z_{n+i}$ to select the table entry. Figure 7.2 shows a PASCAL function for this generator and the statements to initialize the table. (We should point out that random number generators are often implemented in assembly language, and that it is possible to avoid relatively expensive division instructions if this is done. It is doubtful that there is any noticeable saving in

the simulations we discuss, but the savings may be noticeable in other applications of random number generators.

```
CONST M=2147483647.0; A=16807.0;
TYPE RANDINT=1..2147483646;
VAR Z: RANDINT;
    TABLE: ARRAY[0..127] OF RANDINT;
    I: INTEGER;
FUNCTION RANDOM(VAR Z: RANDINT): REAL;
  BEGIN
    (*Z:=(A*Z) MOD M*) Z:=TRUNC(A*Z - (TRUNC((A*Z)/M)*M));
    RANDOM:=TABLE[Z MOD 128]/M;
    TABLE[Z MOD 128]:=Z
  END;  (*RANDOM*)
  ...
  Z:=314159;           (*AN ARBITRARY VALUE*)
  FOR I:=0 TO 127 DO   (*INITIALIZE TABLE  *)
    BEGIN
      (*Z:=(A*Z) MOD M*) Z:=TRUNC(A*Z - (TRUNC((A*Z)/M)*M));
      TABLE[I]:=Z
    END;
  ...
```

Figure 7.2

Given a generator for uniform random variables in the interval (0,1), we return to the generation of other random variables. Consider the uniform random variable on the interval $(a,b)$ where $a$ and $b$ are arbitrary real numbers such that $a$ is less than $b$. The distribution function is $F_x(x_0) = (x_0 - a)/(b - a)$ for $x_0$ in the interval $(a,b)$, as shown in Figure 7.3. Given a value $u_0$ in the interval (0,1) we would like to obtain the appropriate value of $x_0$. Starting with $u_0 = (x_0 - a)/(b - a)$, we can directly obtain $x_0 = (b - a)u_0 + a$.

Suppose we wish to obtain a sample from the exponential distribution shown in Figure 7.1. For positive $x_0$, $F_x(x_0) = 1 - e^{-ax_0}$. Thus we have

$$u_0 = 1 - e^{-ax_0},$$

$$1 - u_0 = e^{-ax_0},$$

$$\ln (1 - u_0) = \ln e^{-ax_0} = -ax_0 \ln e = -ax_0,$$

and

$$x_0 = -(1/a) \ln (1 - u_0).$$

Notice that $1 - u_0$ has the same distribution as $u_0$, so a program would actually use $x_0 = -(1/a) \ln u_0$.

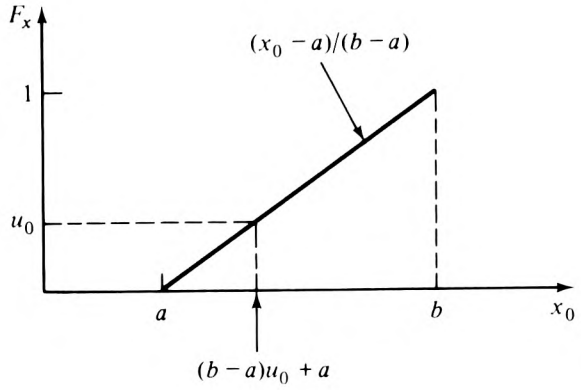



Figure 7.3

Suppose we have a discrete distribution with $n$ possible values, $a_1$, $a_2$, ..., $a_n$, with corresponding probabilities $p_1$, $p_2$, ..., $p_n$, which sum to 1. If we define $q_i$ to be the cumulative probability for the $i^{th}$ value, i.e., $q_i = p_1 + p_2 + \ldots + p_i$, then we sample from this distribution by choosing $a_i$ where $i$ is the smallest value such that $u_0 \leq q_i$. Figure 7.4 illustrates this procedure for $n = 3$ and $a_1 < a_2 < a_3$. Note that this latter condition is required only for clarity of the figure. Usually we would choose the subscripts so that $p_i \geq p_j$ for $i < j$, to allow the procedure to inspect the fewest values in determining a given sample.

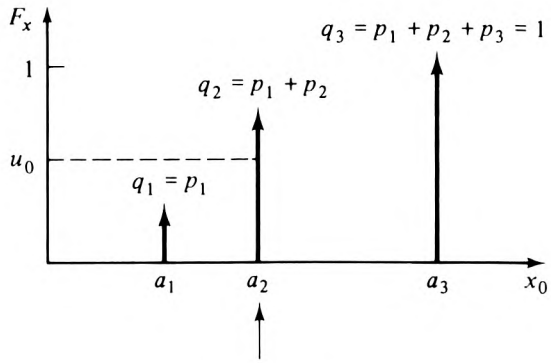



Figure 7.4

These three distributions, the uniform, the exponential and the discrete, along with the branching Erlang distribution discussed in Chapter 3, are the

only ones we will need for our simulations. The above techniques can be used in a straightforward way to sample from the branching Erlang distribution; the details are left as exercises. The inverse distribution approach we have used can be applied whenever we have a convenient expression or efficient algorithm for obtaining the inverse of the distribution function. However, we should point out that alternate approaches are available. For example, there are alternate approaches to sampling from the exponential distribution which are more computationally efficient but are more complex and may require more memory. These approaches for the exponential distribution are entirely reasonable; we ignore them because of their complexity. Further, there are interesting distributions for which there is no practical way to obtain the inverse distribution function. In these cases we must use alternative approaches. Regardless of the specific approach we will need a generator for the uniform distribution on the (0,1) interval.

### 7.1.2 Event List Mechanisms (Simulated Time)

Perhaps the principal difference between simulation programs and other programs is that is that the simulation program must provide the timing mechanism for the simulated system and take simulated time into consideration in its actions. The usual approach to this problem is to identify significant *events* in the simulated system, i.e., times when noticeable changes occur. It is at those points in simulated time that the simulation program must take action. Each event is described by the time it is to occur and by the action that takes place. For a queueing network model, a typical event is the completion of a job's service time. The simulation program maintains a list of events ordered by time of occurrence. The program cycles through the following steps: (1) Select the event with the earliest time. (2) Set the simulated clock to this time. (3) Perform the action.

Suppose we wish to simulate the cyclic queue model which we have solved numerically. Assuming FCFS scheduling, the only events we need to consider are the service completions. While jobs are in service, the simulation program does not need to take any action. However, when a job finishes service the program must reassign the server to a waiting job, if there is one, and the program must move the job to the other queue and possibly initiate service for the job. The program can view simulated time as moving forward in discrete leaps, with leaps ending because of CPU or I/O completions. If we assume exponential service times in our queue model, then the probability of two simultaneous events (service completions) is negligible. With nonexponential service times or other models, simultaneous events may occur frequently. See Figure 7.5. In these cases we must have a rule for determining which of two simultaneous events to handle first; for our purposes we will arbitrarily choose the event which was placed on the event list first.

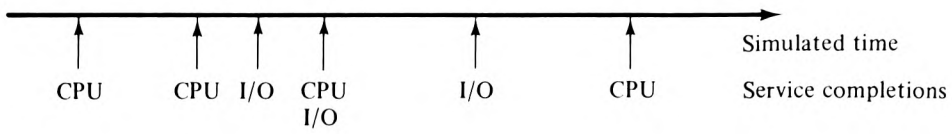



Figure 7.5

There are a variety of ways in which we can store and manipulate the event list in our simulation program. Since we must keep the list ordered by event times, since we will make insertions (and possibly, deletions) anywhere in the list, and since the list will vary in size, an array or similar table will be inappropriate unless the number of events on the list is always small. This is because of the cost of moving many elements when a change is made. The most common, and often the most appropriate, representation is a simple linked list. Each list element consists of (at least) the time of the event, data associated with the event (e.g., the queue for completion events in the cyclic queue model) and a pointer to the next event in the ordering. In addition to list elements, there will be a pointer to the first element in the list, and it is convenient to have a pointer to the last element in the list. See Figure 7.6.

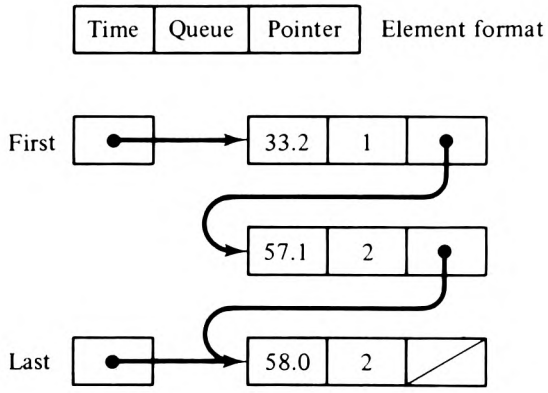



Figure 7.6

```
TYPE ELEMPTR: ↑ELEMENT;
     ELEMENT=RECORD
                TIME: REAL;
                QUEUE: INTEGER;
                NEXT: ELEMPTR
              END;
VAR FIRST, LAST, AVAIL: ELEMPTR;
```

Figure 7.7a

```
PROCEDURE INSERTEVENT(T: REAL; Q: INTEGER);
  (*INSERTEVENT ADDS EVENT AT TIME T FOR QUEUE Q TO LIST*)
  VAR TEMP, L: ELEMPTR;
  BEGIN
    IF AVAIL=NIL THEN
      NEW(TEMP)
    ELSE
      BEGIN (*PREVIOUSLY USED STORAGE AVAILABLE*)
        TEMP:=AVAIL;
        AVAIL:=AVAIL↑.NEXT
      END;
    TEMP↑.TIME:=T;
    TEMP↑.QUEUE:=Q;
    IF FIRST=NIL THEN
      BEGIN (*LIST WAS EMPTY*)
        FIRST:=TEMP;
        LAST:=TEMP;
        TEMP↑.NEXT:=NIL
      END
    ELSE IF T<FIRST↑.TIME THEN
      BEGIN (*INSERT AT BEGINNING OF LIST*)
        TEMP↑.NEXT:=FIRST;
        FIRST:=TEMP
      END
    ELSE IF T≥LAST↑.TIME THEN
      BEGIN (*INSERT AT END OF LIST*)
        LAST↑.NEXT:=TEMP;
        LAST:=TEMP;
        TEMP↑.NEXT:=NIL
      END
    ELSE
      BEGIN (*INSERT SOMEWHERE IN MIDDLE OF LIST*)
        L:=FIRST;
        WHILE T≥L↑.NEXT↑.TIME DO
          L:=L↑.NEXT;
        TEMP↑.NEXT:=L↑.NEXT;
        L↑.NEXT:=TEMP
      END;
  END;  (*INSERTEVENT*)
```

Figure 7.7b

```
PROCEDURE REMOVEFIRSTEVENT(VAR T:REAL; VAR Q: INTEGER);
  (*REMOVEFIRSTEVENT RETURNS TIME T AND QUEUE Q OF FIRST
    EVENT*)
  VAR TEMP: ELEMPTR;
  BEGIN
    IF FIRST=NIL THEN
      BEGIN
        WRITELN('REMOVEFIRSTEVENT -- EMPTY LIST');
        HALT
      END
    ELSE
      BEGIN
        T:=FIRST↑.TIME;
        Q:=FIRST↑.QUEUE;
        TEMP:=FIRST;
        FIRST:=FIRST↑.NEXT;
        IF FIRST=NIL THEN
          LAST:=NIL;
        TEMP↑.NEXT:=AVAIL;
        AVAIL:=TEMP
      END
  END;  (*REMOVEFIRSTEVENT*)
  ...
  FIRST:=NIL;
  LAST:=NIL;
  AVAIL:=NIL;
  ...
```

Figure 7.7c

Figures 7.7a, 7.7b and 7.7c show PASCAL procedures for inserting an element in this linked list representation and for removing the first element. PASCAL provides the NEW procedure for obtaining storage for elements, but provides no complementary procedure for returning storage. So the procedures maintain an auxiliary list of previously used elements with the AVAIL pointer and only calls NEW when this list is empty. The representation of the figures will be adequate for our simulations, but eventually we will want to add a backward pointer to the elements so that we can efficiently remove events in the middle of the list.

Many other organizations can be used for the event list but these will only be appropriate when the list usually has many (more than 30) elements. For our cyclic queue model the maximum list length will be the number of servers (the CPU and the I/O devices).

### 7.1.3 Basic Performance Estimates

We have presented most of the mechanics of a simulation of the cyclic queue model except for the procedures for handling the completion event. These procedures are fairly simple and will be described in Section 7.1.4. The data structures and details of those procedures will depend to a certain extent on what performance measures we wish to obtain, so we provide a brief discussion of performance estimates first. We cannot overemphasize that we obtain only *estimates* for the performance measures of the model, in much the same sense that the approximations of Chapter 6 provide only estimates. In either approximation or simulation, there will usually be some error in the estimates. With approximations we usually cannot estimate the error. With simulation we can provide estimates of the variability of the basic estimates, and can make the error "small" if the simulation run is "long" enough. We will return to these topics in later sections.

For now we will be content with simple estimates for utilization, throughput, mean queue length and mean queueing time. As before, we will count jobs in service as part of the queue length and service time as part of the queueing time. Let us assume the simulation stops at simulated time $T$. Since utilization is defined as the fraction of time the server is busy, we can estimate the utilization by summing the busy times of the server during the simulation run and dividing by $T$ when the run is over. If there are $k$ identical servers, then we can accumulate the busy times for the $k$ servers together, and then estimate the utilization of each one by dividing this aggregate busy time by $kT$. There are two obvious ways to accumulate the busy time for a server. One is to simply add the service times at some appropriate point, e.g., when the sample is obtained, when a job begins service or when a job ends service. However, this may become tricky when we need to include or exclude partial service times at the end of the run. Further, this does not easily generalize to certain cases of interest, so we take a more direct approach. When a server becomes busy, we record the time for future use. Then, when the server becomes idle or the simulation terminates, we take the difference of the current time and this recorded time and add this difference to our sum of busy times. See Figure 7.8. Note that, except for possible numerical differences, we can break the busy periods into discrete subperiods and add the length of the subperiod to our accumulated busy time. This is more convenient with multiple identical servers and allows us to combine the utilization estimation process with the queue length estimation process described below.

For throughput, we need only count the number of jobs going through the queue and divide by $T$ to get our estimate. Again, we have a problem with jobs in service at the end of the run. We will arbitrarily omit these jobs in our throughput estimates.

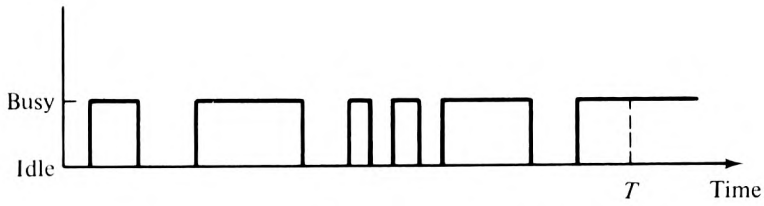



Figure 7.8

There are at least two ways we can estimate the mean queue length. One would be to estimate the queue length distribution and then use that estimate to obtain the mean queue length. This approach is expensive in memory if the potential queue length is large and so is only appropriate if we want the distribution estimate as well. An alternative is closely related to our discussion of Little's Rule in Chapter 2. We can look at queue length as a function of time (Figure 7.9) and estimate the mean queue length as the integral of that function divided by $T$. In other words we estimate the mean queue length by the enclosed area in Figure 7.9 divided by $T$. We can estimate this area easily, in a manner analogous to our preferred approach to estimating busy time. Each time the queue length changes, we record the time for future use. We subtract the previously recorded time from the current time, multiply this difference by the previous queue length and add that product to our summation of the area.

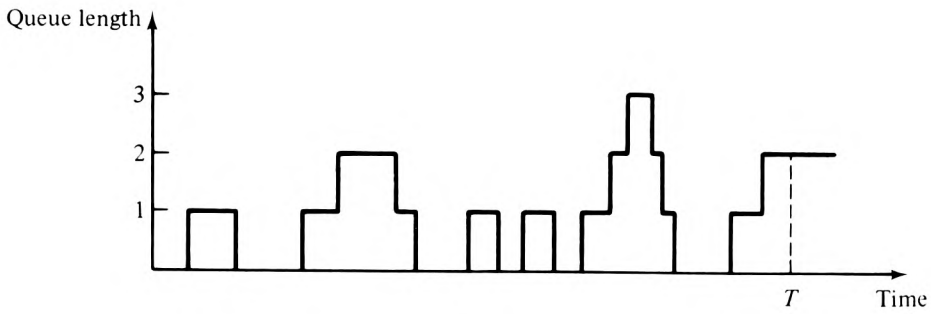



Figure 7.9

There are also two obvious ways to estimate the mean queueing time. We could simply observe the queueing times and use their average as our estimation of the mean. However, this effort is unnecessary if we are only interested in the mean. By Little's Rule we know the mean queueing time is equal to the mean queue length divided by the throughput. Using our estimators for the queue length and throughput, we can eliminate $T$ from numerator and denominator and use the integral of the queue length func-

tion (the area of Figure 7.9) divided by the number of completions as our estimate of mean queueing time.

### 7.1.4 Cyclic Queueing Network Simulator

We are now prepared to present a complete simulation program for the simple cyclic queueing network model described in earlier chapters. The only missing components are the data structures for the queues and the mechanics of handling the completion events. If we are only interested in the above performance measures, if the jobs are homogeneous, and if we assume FCFS scheduling, then a counter giving the length of the queue is all we need to represent each queue. In Section 7.3 we will consider more general data structures for the queue. So the mechanics of the completion event will be to (1) decrease the counter for the queue where the completion occurs, (2) schedule another completion event for that queue if there is a waiting job, (3) increase the counter for the other queue, and (4) schedule a completion event for the other queue if there is an idle server. Since the following generalizations add no complexity and slightly simplify the code, we will allow the number of queues in series to be arbitrary, and we allow multiple identical servers at each queue. We will actually simulate the same model that we solved using the iterative numerical method in Chapter 3. For this model we only obtained state probabilities in Chapter 3. Figure 7.10 gives the numerical values for the above measures for this model.

| Queue | $U$ | $R$ | $L$ | $Q$ |
|---|---|---|---|---|
| 1 | 0.812 | 0.122 | 1.596 | 13.082 |
| 2 | 0.609 | 0.122 | 1.404 | 11.508 |

Figure 7.10

We will run the simulation three times, for 100, 1000 and 10000 events. Figures 7.11a, 7.11b and 7.11c show the complete simulation program except for the bodies of procedures previously defined. The handling of the completion events is separated into two procedures, one for the queue where the completion occurs and one for the queue where the job arrives. This definition of procedures allows easy generalization to networks with other routing paths between queues. Figure 7.12 shows the output from the simulation program.

We arbitrarily initialized the simulated system with all jobs at the first queue. As we look at this model further in Section 7.2, it will be apparent that this choice is of little consequence. This is not to say that choice of initial state is irrelevant for more complex models.

```
PROGRAM CYCLIC(OUTPUT);
(*PROGRAM TO SIMULATE A CYCLIC MODEL WITH NQ QUEUES AND NJ
  JOBS*)
CONST M=2147483647.0; A=16807.0;
      NQ=2; NJ=3; B1=0.15; B2=0.1; NIO=2;
TYPE RANDINT=1..2147483646;
     ELEMPTR: ↑ELEMENT
     ELEMENT=RECORD
                TIME: REAL;
                QUEUE: INTEGER;
                NEXT: ELEMPTR
              END;
VAR Z: RANDINT;
    TABLE: ARRAY[0..127] OF RANDINT;
    I: INTEGER;
    FIRST, LAST, AVAIL: ELEMPTR;
    CLOCK: REAL;
    QUEUES: ARRAY[1..NQ] OF
              RECORD
                NUMBERSERVERS: INTEGER;
                MEANSERVICE: REAL;
                LENGTH: INTEGER;
                TIMELENGTHCHANGED: REAL;
                SUMTIMELENGTH: REAL;
                SUMBUSYTIME: REAL;
                NUMBERCOMPLETIONS: INTEGER
              END;
    RUN, NUMBEREVENTS, EVENTLIMIT: INTEGER;
FUNCTION RANDOM(VAR Z: RANDINT): REAL;
  ...
PROCEDURE INSERTEVENT(T: REAL; Q: INTEGER);
  ...
PROCEDURE REMOVEFIRSTEVENT(VAR T:REAL; VAR Q: INTEGER);
  ...
FUNCTION MIN(V1,V2:INTEGER):INTEGER;
  ...
```

Figure 7.11a

The three runs required roughly 33 ms., 330 ms. and 3.3 seconds, respectively of CPU time on a CDC 6400. These certainly are inexpensive runs, but then this model is trivial for nearly any solution method. Using the methods of Chapter 5, one can obtain these results easily by hand, with appropriate use of a minimal calculator. Even the brute force iterative solution of Chapter 3 requires a few milliseconds on a CDC 6400. For this

```
PROCEDURE COMPLETE(Q: INTEGER);
(*HANDLES COMPLETION OF A JOB AT QUEUE Q*)
  BEGIN
    WITH QUEUES[Q] DO
      BEGIN
        (*STATISTICS*)
        NUMBERCOMPLETIONS:=NUMBERCOMPLETIONS+1;
        SUMTIMELENGTH:=SUMTIMELENGTH+(CLOCK
                        -TIMELENGTHCHANGED)*LENGTH;
        SUMBUSYTIME:=SUMBUSYTIME+(CLOCK-TIMELENGTHCHANGED)
                      *MIN(LENGTH,NUMBERSERVERS);
        TIMELENGTHCHANGED:=CLOCK;
        (*MECHANICS*)
        LENGTH:=LENGTH-1;
        IF LENGTH≥NUMBERSERVERS THEN
          INSERTEVENT(CLOCK-MEANSERVICE*LN(RANDOM(Z)),Q)
      END
  END;  (*COMPLETE*)
PROCEDURE ARRIVE(Q: INTEGER);
(*HANDLES ARRIVAL OF A JOB AT QUEUE Q*)
  BEGIN
    WITH QUEUES[Q] DO
      BEGIN
        (*STATISTICS*)
        SUMTIMELENGTH:=SUMTIMELENGTH+(CLOCK
                        -TIMELENGTHCHANGED)*LENGTH;
        SUMBUSYTIME:=SUMBUSYTIME+(CLOCK-TIMELENGTHCHANGED)
                      *MIN(LENGTH,NUMBERSERVERS);
        TIMELENGTHCHANGED:=CLOCK;
        (*MECHANICS*)
        LENGTH:=LENGTH+1;
        IF LENGTH≤NUMBERSERVERS THEN
          INSERTEVENT(CLOCK-MEANSERVICE*LN(RANDOM(Z)),Q)
      END
  END;  (*ARRIVE*)
```

Figure 7.11b

trivial model, even the longest simulation run does not produce results correct to three significant digits. These statements emphasize a principal liability of simulation: *a relatively long simulation run may produce relatively inaccurate results.* We are assuming that the longer the simulation run the more likely it is that the results will be accurate. This will usually be a correct assumption.

```
BEGIN
  (*INITIALIZATION*)
  Z:=314159;           (*AN ARBITRARY VALUE*)
  FOR I:=0 TO 127 DO  (*INITIALIZE TABLE  *)
    BEGIN
      (*Z:=(A*Z) MOD M*) Z:=TRUNC(A*Z - (TRUNC((A*Z)/M)*M));
      TABLE[I]:=Z
    END;
  AVAIL:=NIL;
  EVENTLIMIT:=10;
  FOR RUN:=1 TO 3 DO
    BEGIN
      FIRST:=NIL;
      LAST:=NIL;
      CLOCK:=0.0;
      NUMBEREVENTS:=0;
      EVENTLIMIT:=10*EVENTLIMIT;
      FOR I:=1 TO NQ DO
        WITH QUEUES[I] DO
          BEGIN
            LENGTH:=0;
            TIMELENGTHCHANGED:=0.0;
            SUMTIMELENGTH:=0.0;
            SUMBUSYTIME:=0.0;
            NUMBERCOMPLETIONS:=0
          END;
      QUEUES[1].NUMBERSERVERS:=1;
      QUEUES[1].MEANSERVICE:=1.0/B1;
      QUEUES[1].LENGTH:=NJ;
      INSERTEVENT(CLOCK-QUEUES[1].MEANSERVICE
                  *LN(RANDOM(Z)),1);
      QUEUES[2].NUMBERSERVERS:=NIO;
      QUEUES[2].MEANSERVICE:=1.0/B2;

      (*RUN*)
      WHILE (FIRST<>NIL) AND (NUMBEREVENTS<EVENTLIMIT) DO
        BEGIN
          NUMBEREVENTS:=NUMBEREVENTS+1;
          REMOVEFIRSTEVENT(CLOCK,I);
          COMPLETE(I);
          ARRIVE(I MOD NQ + 1)
        END;

      (*PRINT STATISTICS*)
      WRITELN;
```

```
      WRITELN('NUMBER OF EVENTS:',NUMBEREVENTS:8,
              ' SIMULATED TIME:',CLOCK:10:3);
      WRITELN;
      WRITELN(
 'QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME');
      FOR I:=1 TO NQ DO
        WITH QUEUES[I] DO
          IF NUMBERCOMPLETIONS>0 THEN
            BEGIN
              SUMTIMELENGTH:=SUMTIMELENGTH+
                     (CLOCK-TIMELENGTHCHANGED)*LENGTH;
              SUMBUSYTIME:=SUMBUSYTIME+
                             MIN(LENGTH,NUMBERSERVERS)*
                             (CLOCK-TIMELENGTHCHANGED);
              WRITELN(I:5,
                     SUMBUSYTIME/(NUMBERSERVERS*CLOCK):12:3,
                     NUMBERCOMPLETIONS/CLOCK:11:3,
                     SUMTIMELENGTH/CLOCK:13:3,
                     SUMTIMELENGTH/NUMBERCOMPLETIONS:14:3)
            END;

      (*PUT LEFTOVER EVENTS ON AVAIL LIST*)
      IF FIRST<>NIL THEN
        BEGIN
          LAST↑.NEXT:=AVAIL;
          AVAIL:=FIRST
        END
    END
END.
```

Figure 7.11c

The reader may also notice that the programming effort to produce this simulation is significantly greater than the effort to produce a numerical program to solve this model. However, this is not an entirely fair comparison. Some of the procedures, i.e., those for random number generation and event list manipulation, can be used with little or no modification in very general simulation programs. This program can be used essentially without modification to simulate networks with many queues and varying numbers of servers per queue. This certainly is not true of the iterative or recursive techniques of Chapter 3, though it is true of the numerical techniques of Chapter 5. Those techniques are based on a product form solution and will not be applicable with FCFS scheduling and non-exponential service times. The simulation program, on the other hand, requires relatively trivial modification to consider other distribution functions. It is difficult to numerically

```
NUMBER OF EVENTS:      100 SIMULATED TIME:    397.416

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
    1       0.807      0.128        1.407         10.964
    2       0.700      0.123        1.593         12.920

NUMBER OF EVENTS:     1000 SIMULATED TIME:   3995.699

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
    1       0.797      0.125        1.535         12.238
    2       0.631      0.125        1.465         11.735

NUMBER OF EVENTS:    10000 SIMULATED TIME: 40884.621

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
    1       0.805      0.122        1.567         12.814
    2       0.619      0.122        1.433         11.717
```

Figure 7.12

obtain the queueing time distributions for this model, and it would be virtually impossible to do so if we were to increase the number of queues and allow non-exponential service times. The simulation program, on the other hand, requires relatively trivial modification to consider other distribution functions. It is difficult to numerically obtain the queueing time distributions for this model, and it would be virtually impossible to do so if we were to increase the number of queues and allow non-exponential service times. With a few modifications to our data structures for the queues and additions to the completion and arrival procedures, the simulation program could be used to estimate the queueing time distributions. These extensions to the simulation program would cause a modest increase in the program's computational requirements, and might require significantly longer runs to obtain sufficiently accurate results, depending on the nature of the service time and queueing time distributions.

The discussion so far has, hopefully, given the reader a feel for the problems cited in the introduction of this chapter. There is another problem worth mentioning. As a consequence of the generality of simulation, simulation models often contain excessive detail and are unwieldy for this reason. We have emphasized, and will continue to emphasize, some of the system characteristics with the most impact on performance. Our intent is to help the reader to avoid excessive detail in models, especially simulation models. The reader should be aware of our intent, and be prepared to consider other characteristics with respect to models of particular systems.

## 7.2 STATISTICAL ANALYSIS OF SIMULATION RESULTS

We have emphasized the variability and potential for error in performance estimates obtained by simulation. In this section we will take a somewhat more formal view of this estimation problem. Our ultimate objective in this section is to show how "confidence interval" estimates may be obtained and interpreted with respect to model performance measures. We will find it most convenient to focus our discussion on estimating the mean queueing time. The same approaches can be applied to most other performance measures of interest, and we will do so in the program examples.

### 7.2.1 Sample Means and Laws of Large Numbers

A deterministic sequence $x_1, x_2, x_3, \ldots$ is said to converge to a limit $C$ if for all $a > 0$ there exists a finite number $n_0$ such that $|x_n - C| < a$ for $n > n_0$. This is written

$$\lim_{n \to \infty} x_n = C.$$

We can similarly define convergence of a sequence of random variables. A sequence $y_1, y_2, y_3, \ldots$ of random variables is said to converge in probability ("stochastically converge") to $C$ if for all $a > 0$,

$$\lim_{n \to \infty} \text{Prob}[\,|y_n - C| > a] = 0.$$

In general, if the value of a performance measure is well defined, we want to say that the simulation estimator converges in probability to this value. This will usually be true, but we must be careful not to read too much into such a statement of convergence.

Let us consider the average of the queueing times as an estimator of the mean queueing time. (The alternative estimator used in the program of Section 7.1.4 will give the same numerical values except for effects of queueing times in progress at the end of the run. These effects should be negligible if the number of queueing times in progress is small relative to the total number.) In traditional terminology, the collection of observed queueing times would be called a "sample," and their average would be called the "sample mean." The sample mean of $n$ observed queueing times, $n = 1,2,3,\ldots$, will be our random variable, $y_n$, and it can be shown under fairly mild assumptions that if the mean queueing time is well defined, the sequence of sample means, $y_1, y_2, y_3, \ldots$, converges in probability to the mean queueing time. This statement of convergence would be known as a "law of large numbers."

This law of large numbers allows us to say that if out sample size is large enough (i.e., our simulation run is long enough) then the sample mean (i.e., the average queueing time) is probably very close to the expected value (i.e., the mean queueing time). This does *not* allow us to say that the sample mean cannot be far from the expected values, but only that the probability of this occurring is small.

Similar statements can be made about the other performance measures we have considered, but the statements are more awkward because the sample size (i.e., simulated time) is not discrete. The reader may more easily imagine these statements if we were to measure the sample size in numbers of events.

### 7.2.2 The Normal Distribution and Central Limit Theorems

We would like to make a probabilistic statement about the potential error in our simulation estimates. Nearly all approaches to this problem are dependent on a very special probability distribution, the *normal* distribution, or on distributions closely related to the normal distribution. (The normal distribution is also known as the *Gaussian* distribution.) The normal distribution has distribution function

$$F_x(x_0) = \int_{-\infty}^{x_0} \frac{1}{\sqrt{2\pi}} e^{-(x-m)^2/2\sigma^2} dx. \tag{7.2}$$

The mean of this distribution is $m$ and the standard deviation is $\sigma$. Notice that the distribution is completely specified by these two parameters. Given a normal distribution of the form (7.2) we can use the transformation $z_0 = (x_0 - m)/\sigma$ to obtain a normal distribution with mean 0 and standard deviation 1. This transformed distribution is known as the *standard* (or *unit*) normal distribution. There is no simple expression for the normal distribution (7.2) so we usually depend on a numerical characterization. There are extensive tables of the standard normal distribution, and we can use these tables in combination with the above transformation to obtain numerical values for an arbitrary normal distribution. Figure 7.13 shows both the density function and the distribution function for the standard normal distribution. Note the symmetry of the density function around the mean.

There are some very remarkable properties associated with the normal distribution. We are most interested in the following one. If $x_1, x_2, \ldots, x_n$ are independent random variables with identical distributions (not necessarily normal) and finite mean and variance, then the distribution of their sum tends toward a normal distribution as $n$ becomes large. This is one version of a class of results known as *central limit theorems*. Such a result may be

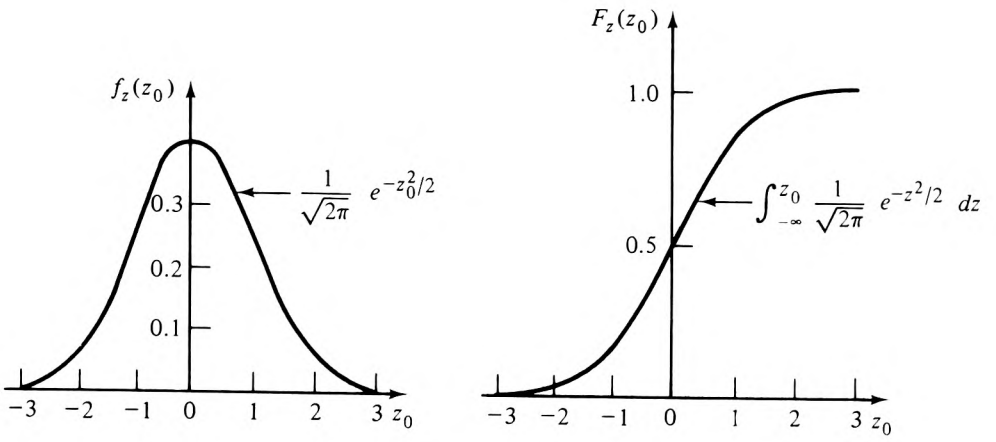



Figure 7.13

accurate for fairly small values of $n$, e.g., 10 to 20, depending on the specific distributions involved. (Aside — The normal distribution is one of the ones we alluded to in Section 7.1.1 as one where an alternative approach to random variable generation is appropriate because of difficulty in characterizing the inverse distribution. One approach to generating normal random variables is to use the sum of 12 uniform random variables on the (0,1) interval and appropriately standardize this sum with the transform described above. Though this approach by itself is rather crude, it can be refined to produce a usable method.)

### 7.2.3 Confidence Intervals

Suppose $z_0$ is a random variable with the standard normal distribution. Let $F_z^{-1}(\alpha)$ be the inverse of $F_z(z_0)$, i.e., $\text{Prob}[z_0 \leq F_z^{-1}(\alpha)] = \alpha$, $0 \leq \alpha \leq 1$. From the symmetry of the density function (Figure 7.13), it is clear that $\text{Prob}[0 \leq z_0 \leq F_z^{-1}(\alpha)] = \alpha - .5$, $.5 \leq \alpha \leq 1$, and $\text{Prob}[-F_z^{-1}(\alpha) \leq z_0 \leq F_z^{-1}(\alpha)] = 2\alpha - 1$. Thus,

$$\text{Prob}[-F_z^{-1}((1 + a)/2) \leq z_0 \leq F_z^{-1}((1 + a)/2)] = a, 0 \leq a \leq 1. \quad (7.3)$$

Tables of $F_z^{-1}(\alpha)$ are readily available. For example, for $a = .9$, $F_z^{-1}(.95) = 1.645$ and we say that $\text{Prob}[-1.645 \leq z_0 \leq 1.645] = .9$. Notice that if we have a *known* value $z_0$, either it is contained in the interval [-1.645,1.645] or it is not. The probabilistic statement only makes sense if $z_0$ is unknown. However, if we obtain many values from the distribution $F_z$, we would expect 90% of them to be contained in the interval [−1.645,1.645].

Now suppose we have random variables $x_1, x_2, \ldots, x_n$ which are independent and identically distributed, each with mean $m$ and variance $\sigma^2$. Then $x_1/n, x_2/n, \ldots, x_n/n$ are also independent and identically distributed, each with mean $m/n$ and variance $(\sigma/n)^2$. Let us take a sample consisting of one value from each random variable $x_1, x_2, \ldots, x_n$. Let us consider the sample mean (the average of these values) and call it $y_n$. Then $y_n = (x_1/n) + (x_2/n) + \ldots + (x_n/n)$ has mean $n(m/n) = m$. Further, since $y_n$ is the sum of independent random variables, its variance is the sum of the individual variances, i.e., $n(\sigma/n)^2 = \sigma^2/n$. If $n$ is large enough, we can reasonably assume that $y_n$ has a normal distribution and that $(y_n - m)\sqrt{n}/\sigma$ has the standard normal distribution. From (7.3) we have

$$\text{Prob}[-F_z^{-1}((1 + a)/2) \leq (y_n - m)\sqrt{n}/\sigma \leq F_z^{-1}((1 + a)/2)] = a,$$

and simple algebra allows us to write

$$\text{Prob}[y_n - d \leq m \leq y_n + d] = a,$$

where

$$d = F_z^{-1}((1 + a)/2)\sigma/\sqrt{n}.$$

Notice that $m$ is *not* a random variable, but $[y_n - d, y_n + d]$ is a *random interval*, i.e., its endpoints are random variables. We must be very careful in our interpretation of this interval with respect to $m$. The interpretation is similar to our statements with respect to the interval obtained in (7.3), but here the interval is random while there it was not, and here $m$ is fixed but there $z_0$ was random. Before obtaining a sample from $x_1, x_2, \ldots, x_n$, we can plan to construct the above interval and say that the interval will contain $m$ with probability $a$. Once we have obtained the sample, either $m$ is contained in the interval or it is not; we should not make a probabilistic statement. However, if we repeat this process many times, we would expect that $a \times 100\%$ of the intervals would contain $m$. The interval $[y_n - d, y_n + d]$ is called a *confidence interval* for $m$; $a \times 100\%$ is called the *confidence level*. Typical confidence levels are 90%, 95% and 99%. We will always use 90% in our examples.

Finally, notice that the variance, $\sigma^2$, of the individual random variables is known. In practice, it is unlikely that we would know the variance without also knowing the mean, in which case we would have no use for the confidence interval. Again assuming $n$ is large, we could estimate the variance by using the sample variance $s^2$, where

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - y_n)^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - ny_n^2\right). \qquad (7.4)$$

Since we are using an estimate of the variance in producing the interval, the interval is properly called a confidence interval *estimate*, but we will just use the term "confidence interval."

Having stated the basic results, we will now discuss the two most theoretically sound methods for producing confidence intervals in simulation, *independent replications* and the *regenerative* method.

### 7.2.4 Independent Replications

In most of our discussion so far, we have implicitly or explicitly assumed that (1) the modeled system appears to reach an equilibrium condition and (2) that we are interested in estimating system performance, given that the system has reached this equilibrium. This will be true for most of the models we have considered, with the obvious exception being open networks where one or more queues has an arrival rate greater than its service rate. Hopefully this will also be true of the modeled system as well, provided that we consider appropriate time periods and are aware of our assumptions in interpreting performance measures.

We may also be interested in *transient* behavior of the system, whether or not the system reaches equilibrium. For example, we may be interested in knowing the mean time until all jobs are at the CPU queue, given the current location of jobs. Or we may be interested in the mean response time for an interactive command, given the current state of the system. In both of these examples the system may be an equilibrium condition, but this is irrelevant. Suppose we wish to estimate the mean number of terminals during the day. If we are considering a single day as our measurement period, then it is probably not reasonable to consider the system as being in equilibrium, though we might reasonably consider a different period of time and make equilibrium assumptions.

For the sorts of models we have considered, numerical solutions for transient behavior are much more difficult than numerical solutions for equilibrium behavior. We have ignored such solutions for that reason. However, with simulation, estimation of transient behavior is no more difficult, in general, than estimation of equilibrium behavior. In fact, estimation of transient behavior is a simpler problem.

Consider our simulation of cyclic queueing networks. If our objective is to estimate the mean utilization of the servers, given that we initially have

all jobs at the first queue and observe the system for some number of service completions, say 100, then we have already given in Figure 7.12 the results of a single experiment with these specifications. One difficulty with using the results of that figure for equilibrium estimates is that the system was not initially in an equilibrium condition, or at least we have not justified our assumption that it was in equilibrium. If our interest is in the above transient measure, or something similar, then we do not have to justify an equilibrium assumption. However, we must still be aware that we are dealing with random processes and that the results of a single experiment may or may not be close to the desired measure. The obvious step is to repeat the experiment many times and use the average of the experimental results as our final estimate. In other words, we *replicate* the experiment.

We now have all we need to provide confidence intervals for our desired performance measure, using the method known as *independent replications*. If we make identical replications of our experiment, i.e., we make identical simulation runs except that we do not reinitialize the random number generator, then we can reasonably assume that the distributions of the performance measures have finite mean and finite variance, then if the number of replications is large enough, we can reasonably assume that the average over the replications has a normal distribution and we can estimate confidence intervals as described in the last section. (We can almost certainly assume a finite mean for the measures of interest if the system has an equilibrium, and this may be a safe assumption otherwise. The finite variance assumption may be harder to justify, but will usually be correct for the cases we are interested in.)

So we essentially know all we need to estimate confidence intervals for transient behavior. We can also use the method of independent replications to estimate equilibrium behavior, but we must make some additional assumptions, and these assumptions may be difficult to justify. Essentially we must assume that each replication accurately reflects the equilibrium behavior of the system. We may separate this assumption into the following two: First, the system is an equilibrium for most of each replication. Second, the results are not significantly affected by the choice of initial conditions. Notice that we would probably be making these assumptions even if we were attempting point estimates only, as in the simulation program of Figure 7.11. Figures 7.14a and 7.14b show the modifications to use the method of independent replications with the program of Figure 7.11. Each replication is 500 events long, and otherwise is the same as the previous runs that we made. Figure 7.15 gives the results of this program for 20 replications.

We can see that all of the known values given in Figure 7.10 are contained in the corresponding confidence intervals. We emphasize that it is not meaningful at this point to say that the probability that the utilization at

```
  ...
VAR Z: RANDINT;
    TABLE: ARRAY[0..127] OF RANDINT;
    I: INTEGER;
    FIRST, LAST, AVAIL: ELEMPTR;
    CLOCK: REAL;
    QUEUES: ARRAY[1..NQ] OF
              RECORD
                NUMBERSERVERS: INTEGER;
                MEANSERVICE: REAL;
                LENGTH: INTEGER;
                TIMELENGTHCHANGED: REAL;
                SUMTIMELENGTH: REAL;
                SUMBUSYTIME: REAL;
                NUMBERCOMPLETIONS: INTEGER;
                SUMUTIL: REAL;
                SUMSQUTIL: REAL;
                SUMTPUT: REAL;
                SUMSQTPUT: REAL;
                SUMQL: REAL;
                SUMSQQL: REAL;
                SUMQT: REAL;
                SUMSQQT: REAL
              END;
    RUN, NUMBEREVENTS, EVENTLIMIT: INTEGER;
    UTIL, TPUT, QL, QT: REAL;
    DUTIL, DTPUT, DQL, DQT: REAL;
  ...
```

Figure 7.14a

queue 1, .812, is contained in the interval (.806,.828) is .9; we know the number .807 is in that interval. Of course, we would not have used the simulation at all if we knew the utilization was .807, but knowing or not knowing the true value does not change the appropriate interpretation of the confidence interval.

If we use independent replications and find that the confidence intervals are larger than we had desired, then we can simulate further to try to obtain narrower intervals. If we are interested in transient behavior, then it is clear that we do not want to make the replications longer, so we simply run more replications. If we have retained the summary results from our previous replications, e.g., SUMUTIL, SUMSQUTIL, etc., then we can use the results from previous replications in producing our new estimates. If we are interested in equilibrium results, then the problem is somewhat more

```
BEGIN
  (*INITIALIZATION*)
  Z:=314159;           (*AN ARBITRARY VALUE*)
  FOR I:=0 TO 127 DO   (*INITIALIZE TABLE  *)
    BEGIN
      (*Z:=(A*Z) MOD M*) Z:=TRUNC(A*Z - (TRUNC((A*Z)/M)*M));
      TABLE[I]:=Z
    END;
    FOR I:=1 TO NQ DO
      WITH QUEUES[I] DO
        BEGIN
          SUMUTIL:=0;
          SUMSQUTIL:=0;
          SUMTPUT:=0;
          SUMSQTPUT:=0;
          SUMQL:=0;
          SUMSQQL:=0;
          SUMQT:=0;
          SUMSQQT:=0
        END;
  AVAIL:=NIL;
  EVENTLIMIT:=500;
  FOR RUN:=1 TO NREP DO
    BEGIN
      ... (*ONE REPLICATION OF THE SIMULATION. *)
      FOR I:=1 TO NQ DO
        WITH QUEUES[I] DO
          IF NUMBERCOMPLETIONS>0 THEN
            BEGIN
              SUMTIMELENGTH:=SUMTIMELENGTH+
                     (CLOCK-TIMELENGTHCHANGED)*LENGTH;
              SUMBUSYTIME:=SUMBUSYTIME+
                             MIN(LENGTH,NUMBERSERVERS)*
                             (CLOCK-TIMELENGTHCHANGED);
              UTIL:=SUMBUSYTIME/(NUMBERSERVERS*CLOCK);
              SUMUTIL:=SUMUTIL+UTIL;
              SUMSQUTIL:=SUMSQUTIL+UTIL*UTIL;
              TPUT:=NUMBERCOMPLETIONS/CLOCK;
              SUMTPUT:=SUMTPUT+TPUT;
              SUMSQTPUT:=SUMSQTPUT+TPUT*TPUT;
              QL:=SUMTIMELENGTH/CLOCK;
              SUMQL:=SUMQL+QL;
              SUMSQQL:=SUMSQQL+QL*QL;
              QT:=SUMTIMELENGTH/NUMBERCOMPLETIONS;
              SUMQT:=SUMQT+QT;
```

```
            SUMSQQT:=SUMSQQT+QT*QT
          END;

    (*PUT LEFTOVER EVENTS ON AVAIL LIST*)
    IF FIRST<>NIL THEN
      BEGIN
        LAST↑.NEXT:=AVAIL;
        AVAIL:=FIRST
      END
  END;
  WRITELN;
  WRITELN('REPLICATIONS:',NREP:4,
          ' EVENTS PER REPLICATION:',EVENTLIMIT:6);
  WRITELN;
  WRITELN(
'QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME');
  FOR I:=1 TO NQ DO
    WITH QUEUES[I] DO
      BEGIN
        UTIL:=SUMUTIL/NREP;
        DUTIL:=1.645
          *SQRT((SUMSQUTIL-SUMUTIL*UTIL)/((NREP-1)*NREP));
        TPUT:=SUMTPUT/NREP;
        DTPUT:=1.645
          *SQRT((SUMSQTPUT-SUMTPUT*TPUT)/((NREP-1)*NREP));
        QL:=SUMQL/NREP;
        DQL:=1.645
              *SQRT((SUMSQQL-SUMQL*QL)/((NREP-1)*NREP));
        QT:=SUMQT/NREP;
        DQT:=1.645
              *SQRT((SUMSQQT-SUMQT*QT)/((NREP-1)*NREP));
        WRITELN('UPPER',UTIL+DUTIL:12:3,TPUT+DTPUT:11:3,
                QL+DQL:13:3,QT+DQT:14:3);
        WRITELN(I:5,UTIL:12:3,TPUT:11:3,QL:13:3,QT:14:3);
        WRITELN('LOWER',UTIL-DUTIL:12:3,TPUT-DTPUT:11:3,
                QL-DQL:13:3,QT-DQT:14:3);
      END
END.
```

Figure 7.14b

REPLICATIONS: 20 EVENTS PER REPLICATION: 500

| QUEUE | UTILIZATION | THROUGHPUT | QUEUE LENGTH | QUEUEING TIME |
|---|---|---|---|---|
| UPPER | 0.828 | 0.125 | 1.651 | 13.763 |
| 1 | 0.817 | 0.123 | 1.606 | 13.186 |
| LOWER | 0.806 | 0.120 | 1.562 | 12.609 |
| UPPER | 0.622 | 0.125 | 1.438 | 11.776 |
| 2 | 0.605 | 0.122 | 1.394 | 11.443 |
| LOWER | 0.588 | 0.119 | 1.349 | 11.110 |

Figure 7.15

complex, at least conceptually. If we are satisfied with the assumption that the replications are adequate examples of equilibrium behavior, then we can proceed as in the transient case. However, this assumption is difficult to justify, so it would be more appropriate to lengthen each replication rather than to increase the number of replications. There are two obstacles, if we wish to use the data from the replications already made. First, we will have difficulty resuming replications where they ended unless we have anticipated doing so and have recorded the state of the system, e.g., current queue lengths, pending event times, etc., and the values of the statistics accumulator variables, e.g., TIMELENGTHCHANGED, SUMTIMELENGTH, etc. Even with advance planning the programming effort and memory required may make this impractical since we must do this for every replication. Second, we will be in an awkward position with respect to our random number generator. We would like it to continue for each replication where it was left at the previous end of the replication. However, we have already used these values for other replications, so we must use some alternative if we wish the replications to be independent. Though this second obstacle can be overcome, we will usually find that the first one is intractable and be forced to choose between discarding our existing results and using longer replications on the one hand, or saving our previous results and making more replications of the original length, on the other hand.

There is a less rigorous method for estimating confidence intervals for equilibrium behavior which is similar to the method of independent replications and is known as *batch means*. This method uses *batches* which are treated analogously to replications but which are obtained in a different manner. Rather than reinitializing the system at the beginning of each batch, one batch begins in the state in which the previous batch ended. Thus our "independent and identically distributed" assumption will be difficult to defend. However, for some systems this is a defensible assumption, *if the batches are long enough*, and this method has some advantages of convenience. We can reasonably use batch means for our cyclic queue

model with the parameters as before. Figure 7.16 shows that the results are similar to those of Figure 7.15.

```
BATCHES:   20 EVENTS PER BATCH:    500

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER        0.827      0.125        1.641         13.640
    1        0.816      0.122        1.601         13.136
LOWER        0.805      0.120        1.561         12.632
UPPER        0.623      0.125        1.439         11.738
    2        0.608      0.122        1.399         11.444
LOWER        0.592      0.120        1.359         11.150
```

Figure 7.16

### 7.2.5 The Regenerative Method

We know from our discussion in Chapter 3 that the future behavior of a Markov process is dependent only on the current state of the process, then each time the process enters that state the process will have the same expected future behavior. We must emphasize "expected"; the actual future behaviors will be different. We may say that the Markov process *regenerates* each time it enters the specified (*regeneration*) state and calls the periods between successive entrances to the state regeneration *cycles*. (We should be a little more careful. We can define Markov processes which never return to a specified state, and thus are not regenerative. However, all Markov processes of the subset defined in Chapter 3 are regenerative. There are also regenerative processes which are not Markovian, but we will ignore these.)

We can take advantage of the regenerative structure of a simulation model to estimate confidence intervals for equilibrium behavior, provided we can determine a regeneration state which is entered sufficiently frequently, i.e., the regeneration cycles are sufficiently short. (We can define "frequently" and "short" in pragmatic terms. Though theoretical restrictions exist, they will usually be weaker restrictions than those imposed by practical considerations.) The regenerative method for confidence intervals is independent of our ability or lack of ability to obtain numerical solutions for the model. It is principally dependent on our ability to observe enough regeneration cycles that we may apply the results of an appropriate central limit theorem. Note that this second condition is similar to our requirement in the method of independent replications that the number of replications be large. (Aside — it is possible to apply the method of independent replications with a few replications using different assumptions about distributions.

This approach will be the same as the one we use for sufficiently large numbers of replications.)

A principal advantage of the regenerative method, given the above conditions, is that if we initialize the simulation in a regeneration state, then we can reasonably assume we have initialized the simulation in an equilibrium condition! Observing regeneration cycles will then be observing periods of equilibrium behavior. We can formally justify the least supportable assumption of the method of independent replications for equilibrium behavior, that each replication accurately represents equilibrium behavior of the system. (Aside — if we know that several regeneration cycles occur during each replication, then this is strong support for this equilibrium behavior assumption. Our knowledge of the regenerative structure of the cyclic queue model was part of the basis for our choice of replication and batch lengths in the previous section.)

Besides recognizing the entrances to the regeneration state, the regenerative method is somewhat more complex for our estimators because the regeneration cycles are of random length. Consider estimators for mean queueing time. For each replication we estimated mean queueing time as SUMTIMELENGTH/NUMBERCOMPLETIONS, and our final estimate was simply the average of these values. This was reasonable because each replication had a fixed length (measured in events) and was long enough that we could reasonably assume that NUMBERCOMPLETIONS had essentially the same value for each replication. However, the number of completed queueing times during a regeneration cycle may be very small and/or highly variable. Thus if we take the average of SUMTIMELENGTH/NUMBERCOMPLETIONS over all of the regeneration cycles, we may get a quite different result from the value of SUMTIMELENGTH/NUMBERCOMPLETIONS taken over a single long run and ignoring regeneration cycles. This is unsatisfactory, of course, so we take a more careful approach. Rather than use the average of SUMTIMELENGTH/NUMBERCOMPLETIONS over the regeneration cycles, we use the average of SUMTIMELENGTH divided by the average of NUMBERCOMPLETIONS. In other words, we use the quotient of the averages rather than the average of the quotient. It is easy to see that this is algebraically equivalent to what we would do if we were ignoring regeneration cycles, i.e., SUMTIMELENGTH/NUMBERCOMPLETIONS equals (SUMTIMELENGTH/$n$)/(NUMBERCOMPLETIONS/$n$) where SUMTIMELENGTH, NUMBERCOMPLETIONS and $n$ are taken over the entire run and $n$ is the number of regeneration cycles. In programming this estimate, we will not actually do the divisions by $n$ when we are obtaining point estimates. Thus with respect to point estimates there is no difference mechanically whether we consider regeneration cycles or not.

However, the computations are more complex with respect to confidence intervals as compared with our previous methods. For notational convenience, let us call the value of SUMTIMELENGTH for the $i^{th}$ regeneration cycle $u_i$, and the value of NUMBERCOMPLETIONS for the $i^{th}$ cycle, $v_i$, $i = 1, 2, \ldots, n$. Because of regenerative structure of our system, $u_1, u_2, \ldots, u_n$ are independent and identically distributed. Similarly, $v_1, v_2, \ldots, v_n$ are independent and identically distributed. Further, the pairs $(u_1,v_1), (u_2,v_2), \ldots, (u_n,v_n)$ are independent and identically distributed. If we define $w_n$ as the average of $u_1, u_2, \ldots, u_n$, $x_n$ as the average of $v_1, v_2, \ldots, v_n$ and $y_n$ as $w_n/x_n$, then we can obtain a law of large numbers to show that $y_n$ converges to $m$, where $m$ is our mean queueing time. Further, we can prove a central limit theorem and eventually produce the confidence interval estimate $[y_n - d, y_n + d]$ where

$$d = \frac{F_z^{-1}((1+a)/2)s}{x_n\sqrt{n}},$$

$$s^2 = s_u^2 - 2y_n s_{uv} + y_n^2 s_v^2,$$

$$s_u^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} u_i^2 - nw_n^2\right),$$

$$s_{uv} = \frac{1}{n-1}\left(\sum_{i=1}^{n} u_i v_i - nw_n x_n\right),$$

and

$$s_v^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} v_i^2 - nx_n^2\right).$$

Thus our simulation program must recognize when the $i^{th}$ cycle has ended and maintain sums of $u_i$, $u_i^2$, $u_i v_i$, $v_i$, and $v_i^2$, for $i = 1, 2, \ldots, n$. For mean queueing time, we will maintain these sums in the variables TL, TLSQ, TLXNC, NC and NCSQ, respectively. Figures 7.17a, 7.17b and 7.17c show the modifications to the program of Figure 7.11, and Figure 7.18 shows the output of this program.

In this program we use the initial state that we have used before: all jobs at the first queue. As discussed in Chapter 3, the Markov states for the cyclic queue model are uniquely specified by the number of jobs at each queue, assuming that service times are exponential. Thus our initial state is also a regeneration state, and is used as the regeneration state for determining confidence intervals. We could use other choices for our regeneration

```
VAR Z: RANDINT;
    TABLE: ARRAY[0..127] OF RANDINT;
    I: INTEGER;
    FIRST, LAST, AVAIL: ELEMPTR;
    CLOCK: REAL;
    QUEUES: ARRAY[1..NQ] OF
              RECORD
                NUMBERSERVERS: INTEGER;
                MEANSERVICE: REAL;
                LENGTH: INTEGER;
                TIMELENGTHCHANGED: REAL;
                SUMTIMELENGTH: REAL;
                SUMBUSYTIME: REAL;
                NUMBERCOMPLETIONS: INTEGER;
                (*SUMS OF CYCLE VALUES*)
                (*BT=BUSYTIME;
                  SQ=SQUARED;
                  X=TIMES;
                  CL=CYCLELENTGH;
                  NC=NUMBERCOMPLETIONS;
                  TL=TIMELENGTH;*)
                BT: REAL;
                TL: REAL;
                NC: REAL;
                BTSQ: REAL;
                BTXCL: REAL;
                NCSQ: REAL;
                NCXCL: REAL;
                TLSQ: REAL;
                TLXCL: REAL;
                TLXNC: REAL
              END;
    RUN, NUMBEREVENTS, EVENTLIMIT, EVENTMAX: INTEGER;
    NOEVENTSDURINGCYCLES, NUMBERCYCLES, NOCYCM1: INTEGER;
    TIMECYCLESTARTED, CYCLELENGTH,
       SUMCL, SUMCLSQ, VARCL, DCL: REAL;
    UTIL, DUTIL, VARBT, COVARBTCL, VART: REAL;
    TPUT, DTPUT, VARNC, COVARNCCL: REAL;
    QL, DQL, VARTL, COVARTLCL: REAL;
    QT, DQT, COVARTLNC: REAL;
```

Figure 7.17a

```
FUNCTION ENDCYCLE: BOOLEAN;
(*DETERMINES WHETHER AT END OF REGENERATION CYCLE.
  IF SO, ENDCYCLE UPDATES ACCUMULATORS.*)
  VAR Q: INTEGER;
  BEGIN
    IF (QUEUES[1].LENGTH=NJ) AND (NUMBEREVENTS>0) THEN
      BEGIN
        ENDCYCLE:=TRUE;
        NOEVENTSDURINGCYCLES:=NUMBEREVENTS;
        NUMBERCYCLES:=NUMBERCYCLES+1;
        CYCLELENGTH:=CLOCK-TIMECYCLESTARTED;
        TIMECYCLESTARTED:=CLOCK;
        SUMCL:=SUMCL+CYCLELENGTH;
        SUMCLSQ:=SUMCLSQ+SQR(CYCLELENGTH);
        FOR Q:=1 TO NQ DO
          WITH QUEUES[Q] DO
            BEGIN
              SUMTIMELENGTH:=SUMTIMELENGTH
                          +(CLOCK-TIMELENGTHCHANGED)*LENGTH;
              SUMBUSYTIME:=(SUMBUSYTIME
                       +(CLOCK-TIMELENGTHCHANGED)
                   *MIN(LENGTH,NUMBERSERVERS))/NUMBERSERVERS;
              TIMELENGTHCHANGED:=CLOCK;
              (*BT=BUSYTIME;
                SQ=SQUARED;
                X=TIMES;
                CL=CYCLELENTGH;
                NC=NUMBERCOMPLETIONS;
                TL=TIMELENGTH;*)
              BT:=BT+SUMBUSYTIME;
              TL:=TL+SUMTIMELENGTH;
              NC:=NC+NUMBERCOMPLETIONS;
              BTSQ:=BTSQ+SQR(SUMBUSYTIME);
              BTXCL:=BTXCL+SUMBUSYTIME*CYCLELENGTH;
              SUMBUSYTIME:=0.0;
              NCSQ:=NCSQ+SQR(NUMBERCOMPLETIONS);
              NCXCL:=NCXCL+NUMBERCOMPLETIONS*CYCLELENGTH;
              TLSQ:=TLSQ+SQR(SUMTIMELENGTH);
              TLXCL:=TLXCL+SUMTIMELENGTH*CYCLELENGTH;
              TLXNC:=TLXNC+SUMTIMELENGTH*NUMBERCOMPLETIONS;
              NUMBERCOMPLETIONS:=0;
              SUMTIMELENGTH:=0.0
            END
      END
    ELSE
```

```
        ENDCYCLE:=FALSE
    END;  (*ENDCYCLE*)
```

Figure 7.17b

state, but it can be shown that the expected width of the confidence intervals is independent of the choice of regeneration state, given the same simulated time [CRAN74]. Our main criteria in choosing the regeneration state are that we can easily identify entrances to the state and that the number of regeneration cycles not be too small. Procedure ENDCYCLE is used to determine whether or not the system is in the regeneration state after such event. Notice that the completion events correspond exactly to state transitions of the Markov process. Thus if we find all of the jobs at the first queue after handling an event, we know the system has just entered the regeneration state.

We would like to have the simulation end at the end of a regeneration cycle, for both practical and theoretical reasons. For this reason, the run lengths are specified by both a "soft" limit (EVENTLIMIT) and a "firm" limit (EVENTMAX). If a cycle end does not occur between these two limits, then the program produces confidence interval and point estimates based only on the completed cycles. (If there is only one completed cycle, or there are no completed cycles, then only point estimates are produced. The user of the program should disregard the confidence interval estimates if the number of cycles is small.)

The three runs described in Figure 7.18 correspond to the three runs of Figure 7.12, with the difference (besides the confidence intervals) being that the first and third runs were extended so that the last cycle would be complete. Essentially the same computational effort went into the third runs of these Figures, the 20 replications of Figure 7.25 and the 20 batches of Figure 7.16.

Notice that the confidence intervals for mean queue length from the first run of Figure 7.18 do not include the expected queue lengths given in Figure 7.10. Recall that we would expect the confidence intervals for a particular value for a particular model to contain the expected value for 90% of the runs if we made a larger number of runs, assuming a 90% confidence level. Notice that the confidence intervals from a given run are strongly dependent on each other and thus we should not attempt a similar statement for the set of confidence intervals from a run. (Notice that the queue length intervals for one queue are directly obtainable from the queue length intervals for the other queue, the throughput intervals are identical for the two queues, etc.) It happens that all the other intervals in Figure

```
BEGIN
  (*INITIALIZATION*)
  Z:=314159;           (*AN ARBITRARY VALUE*)
  FOR I:=0 TO 127 DO   (*INITIALIZE TABLE  *)
    BEGIN
      (*Z:=(A*Z) MOD M*) Z:=TRUNC(A*Z - (TRUNC((A*Z)/M)*M));
      TABLE[I]:=Z
    END;
  AVAIL:=NIL;
  EVENTLIMIT:=10;
  FOR RUN:=1 TO 3 DO
    BEGIN
      FIRST:=NIL;
      LAST:=NIL;
      CLOCK:=0.0;
      NUMBEREVENTS:=0;
      NUMBERCYCLES:=0;
      TIMECYCLESTARTED:=0.0;
      SUMCL:=0.0;
      SUMCLSQ:=0.0;
      EVENTLIMIT:=10*EVENTLIMIT;
      EVENTMAX:=2*EVENTLIMIT;
      FOR I:=1 TO NQ DO
        WITH QUEUES[I] DO
          BEGIN
            LENGTH:=0;
            TIMELENGTHCHANGED:=0.0;
            SUMTIMELENGTH:=0.0;
            SUMBUSYTIME:=0.0;
            NUMBERCOMPLETIONS:=0;
            BT:=0.0;
            TL:=0.0;
            NC:=0.0;
            BTSQ:=0.0;
            BTXCL:=0.0;
            NCSQ:=0.0;
            NCXCL:=0.0;
            TLSQ:=0.0;
            TLXCL:=0.0;
            TLXNC:=0.0
          END;
      QUEUES[1].NUMBERSERVERS:=1;
      QUEUES[1].MEANSERVICE:=1.0/B1;
      QUEUES[1].LENGTH:=NJ;
      INSERTEVENT(CLOCK-QUEUES[1].MEANSERVICE*LN(RANDOM(Z)),
```

```
                                                        1);
      QUEUES[2].NUMBERSERVERS:=NIO;
      QUEUES[2].MEANSERVICE:=1.0/B2;

      (*RUN*)
      WHILE (FIRST<>NIL) AND (NUMBEREVENTS<EVENTMAX)
          AND ((NUMBEREVENTS<EVENTLIMIT) OR NOT ENDCYCLE) DO
        BEGIN
          NUMBEREVENTS:=NUMBEREVENTS+1;
          REMOVEFIRSTEVENT(CLOCK,I);
          COMPLETE(I);
          ARRIVE(I MOD NQ + 1)
        END;
      (*PRINT STATISTICS*)
      WRITELN;
      WRITELN('NUMBER OF EVENTS:',NUMBEREVENTS:8,
              ' SIMULATED TIME:',CLOCK:10:3);
      WRITELN;
      WRITELN(
'QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME');
      IF NUMBERCYCLES>1 THEN
        (*PRODUCE CONFIDENCE INTERVAL ESTIMATES*)
        BEGIN
          CYCLELENGTH:=SUMCL/NUMBERCYCLES;
          NOCYCM1:=NUMBERCYCLES-1;
          VARCL:=(SUMCLSQ-SQR(SUMCL)/NUMBERCYCLES)/NOCYCM1;
          FOR I:=1 TO NQ DO
            WITH QUEUES[I] DO
              IF NC>0 THEN
                BEGIN
                  UTIL:=BT/SUMCL;
                  VARBT:=(BTSQ-SQR(BT)/NUMBERCYCLES)
                              /NOCYCM1;
                  COVARBTCL:=(BTXCL-BT*SUMCL/NUMBERCYCLES)
                              /NOCYCM1;
                  DUTIL:=1.645*SQRT((VARBT-2*UTIL*COVARBTCL
                         +SQR(UTIL)*VARCL)/NUMBERCYCLES)
                         /CYCLELENGTH;
                  TPUT:=NC/SUMCL;
                  VARNC:=(NCSQ-SQR(NC)/NUMBERCYCLES)
                              /NOCYCM1;
                  COVARNCCL:=(NCXCL-NC*SUMCL/NUMBERCYCLES)
                              /NOCYCM1;
                  DTPUT:=1.645*SQRT((VARNC-2*TPUT*COVARNCCL
                         +SQR(TPUT)*VARCL)/NUMBERCYCLES)
```

```
                    /CYCLELENGTH;
                QL:=TL/SUMCL;
                VARTL:=(TLSQ-SQR(TL)/NUMBERCYCLES)
                    /NOCYCM1;
                COVARTLCL:=(TLXCL-TL*SUMCL/NUMBERCYCLES)
                    /NOCYCM1;
                DQL:=1.645*SQRT((VARTL-2*QL*COVARTLCL
                    +SQR(QL)*VARCL)/NUMBERCYCLES)
                    /CYCLELENGTH;
                QT:=TL/NC;
                COVARTLNC:=(TLXNC-TL*NC/NUMBERCYCLES)
                    /NOCYCM1;
                DQT:=1.645*SQRT((VARTL-2*QT*COVARTLNC
                    +SQR(QT)*VARNC)/NUMBERCYCLES)
                    /(NC/NUMBERCYCLES);
                WRITELN('UPPER',
                    UTIL+DUTIL:12:3,TPUT+DTPUT:11:3,
                    QL+DQL:13:3,QT+DQT:14:3);
                WRITELN(I:5,UTIL:12:3,TPUT:11:3,
                    QL:13:3,QT:14:3);
                WRITELN('LOWER',
                    UTIL-DUTIL:12:3,TPUT-DTPUT:11:3,
                    QL-DQL:13:3,QT-DQT:14:3)
            END;
        WRITELN;
        WRITELN('NUMBER OF CYCLES:',NUMBERCYCLES:8);
        IF NOEVENTSDURINGCYCLES<>NUMBEREVENTS THEN
          WRITELN('NUMBER OF DISCARDED EVENTS:',
                NUMBEREVENTS-NOEVENTSDURINGCYCLES:8);
        WRITELN('AVERAGE NUMBER OF EVENTS:',
                NOEVENTSDURINGCYCLES/NUMBERCYCLES:10:3);
        DCL:=1.645*SQRT(VARCL/NUMBERCYCLES);
        WRITELN('AVERAGE LENGTH:',CYCLELENGTH:10:3,
                ' C.I.:(',CYCLELENGTH-DCL:10:3,',',
                CYCLELENGTH+DCL:10:3,')')
      END
    ELSE
      (*PRODUCE POINT ESTIMATES ONLY*)
      FOR I:=1 TO NQ DO
        WITH QUEUES[I] DO
          IF NUMBERCOMPLETIONS+TRUNC(NC)>0 THEN
            BEGIN
              SUMTIMELENGTH:=SUMTIMELENGTH+TL;
              SUMBUSYTIME:=SUMBUSYTIME+BT*NUMBERSERVERS;
              NUMBERCOMPLETIONS:=NUMBERCOMPLETIONS
```

```
                                    +TRUNC(NC);
                SUMTIMELENGTH:=SUMTIMELENGTH+
                        (CLOCK-TIMELENGTHCHANGED)*LENGTH;
                SUMBUSYTIME:=SUMBUSYTIME+
                                MIN(LENGTH,NUMBERSERVERS)*
                                (CLOCK-TIMELENGTHCHANGED);
                WRITELN(I:5,
                     SUMBUSYTIME/(NUMBERSERVERS*CLOCK):12:3,
                     NUMBERCOMPLETIONS/CLOCK:11:3,
                     SUMTIMELENGTH/CLOCK:13:3,
                     SUMTIMELENGTH/NUMBERCOMPLETIONS:14:3)
              END;

    (*PUT LEFTOVER EVENTS ON AVAIL LIST*)
    IF FIRST<>NIL THEN
      BEGIN
        LAST↑.NEXT:=AVAIL;
        AVAIL:=FIRST
      END
  END
END.
```

Figure 7.17c

7.18 contain the expected values. We would tend to question the intervals from the first run because of the small number of cycles.

If the confidence intervals obtained are wider than we would like, then it is relatively easy to have the program continue the simulation for additional cycles until the intervals are satisfactory.

In comparing the regenerative method and the method of independent replications for equilibrium behavior, we prefer the regenerative method because the assumptions made are relatively easy to justify. The principal difficulty with the regenerative method is in finding a frequently occurring regeneration state. This is easy for simple models, but may be quite difficult for complex models. The principal problem with the method of independent replications is in justifying the assumption that the replications represent equilibrium behavior. As with the regenerative method, this is easy for simple models but may be quite difficult otherwise.

In the following sections, we will principally discuss the mechanics of simulating more complex systems. However, as appropriate, we will discuss choice of regeneration states for these more complex models.

```
NUMBER OF EVENTS:      104 SIMULATED TIME:   423.132

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.896      0.143        1.583         13.368
    1       0.812      0.123        1.395         11.354
LOWER       0.727      0.103        1.208          9.341
UPPER       0.779      0.143        1.792         16.073
    2       0.708      0.123        1.605         13.057
LOWER       0.638      0.103        1.417         10.042

NUMBER OF CYCLES:       10
AVERAGE NUMBER OF EVENTS:    10.400
AVERAGE LENGTH:    42.313 C.I.:(    26.193,    58.433)

NUMBER OF EVENTS:     1000 SIMULATED TIME:  3955.809

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.828      0.134        1.641         13.366
    1       0.795      0.126        1.523         12.046
LOWER       0.761      0.119        1.404         10.726
UPPER       0.680      0.134        1.596         12.656
    2       0.636      0.126        1.477         11.689
LOWER       0.592      0.119        1.359         10.722

NUMBER OF CYCLES:      120
AVERAGE NUMBER OF EVENTS:     8.333
AVERAGE LENGTH:    32.965 C.I.:(    27.571,    38.359)

NUMBER OF EVENTS:    10006 SIMULATED TIME: 40899.099

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.817      0.124        1.604         13.241
    1       0.805      0.122        1.567         12.812
LOWER       0.793      0.120        1.530         12.384
UPPER       0.632      0.124        1.470         12.022
    2       0.619      0.122        1.433         11.713
LOWER       0.605      0.120        1.396         11.403

NUMBER OF CYCLES:     1312
AVERAGE NUMBER OF EVENTS:     7.627
AVERAGE LENGTH:    31.173 C.I.:(    29.724,    32.622)
```

**Figure 7.18**

## 7.3 SIMULATION OF GENERAL QUEUEING NETWORKS

It is the objective of this section and the accompanying exercises to extend the program of Figure 7.17 to allow simulation of queueing networks with any or all of the characteristics discussed in Chapters 3, 4 and 5, e.g., networks with sources and sinks, general queueing disciplines, non-exponential service times, job classes, probabilistic routing, etc. Section 7.4 and the exercises will further extend the program to consider some of the characteristics cited in Chapter 6 as precluding exact solution, e.g., simultaneous resource possession, state dependent routing, overlapped job activities, etc. (Notice that some of the characteristics of this section may also preclude exact solution for networks with moderate size, e.g., FCFS scheduling with non-exponential service times.)

In order that the programs of this section be understandable, we will proceed in two separate steps. First we will modify the earlier program to allow sources and sinks, an essentially trivial modification. Then we will ignore sources and sinks, but consider simulation of a close network with the FCFS, LCFSPR and PS queueing disciplines, Erlang service times, job classes and probabilistic routing. The combination of these two steps, as well as other refinements, will be left as exercises. In both steps we will illustrate use of the regenerative method for confidence intervals.

### 7.3.1 Sources and Sinks (Open Networks)

Consider a network similar to the cyclic network we have been considering, but in which there is a source of jobs which arrive at the first queue and in which jobs leaving the last queue leave the network rather than returning to the first queue. See Figure 7.19. We assume that the queues are characterized as before, both externally and in the internal data structures. The number of jobs in the network is variable and potentially infinite. We will keep track of the number of jobs with the variable TOTAL-LENGTH. Jobs arrive from the source in a Poisson manner, i.e., the interarrival times are exponential. We are not proposing this network as a computer system model. However, networks with sources and sinks often are appropriate models of communication systems and sometimes are appropriate models of computer systems.

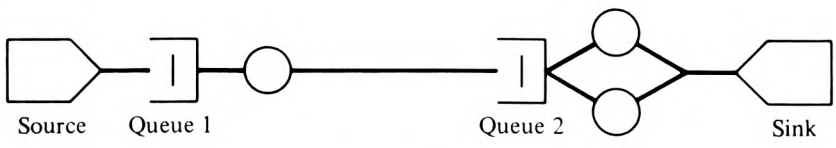



Figure 7.19

| Queue | $U$ | $R$ | $L$ | $Q$ |
|---|---|---|---|---|
| 1 | 0.833 | 0.125 | 5.000 | 40.000 |
| 2 | 0.625 | 0.125 | 2.051 | 16.410 |

Figure 7.20

```
CONST M=2147483647.0; A=16807.0;
      NQ=2; NJ=3; B1=0.15; B2=0.1; NIO=2;
      MEANINTERARRIVAL=8.0;
TYPE RANDINT=1..2147483646;
     ELEMPTR: ↑ELEMENT
     ELEMENT=RECORD
               TIME: REAL;
               PARAM: INTEGER;
               NEXT: ELEMPTR
             END;
VAR Z: RANDINT;
  ...
    TOTALLENGTH: INTEGER;
  ...
PROCEDURE INSERTEVENT(T: REAL; Q: INTEGER);
  (*INSERTEVENT ADDS EVENT AT TIME T FOR PARAM Q TO LIST*)
  VAR TEMP, N, L: ELEMPTR;
  BEGIN
    IF AVAIL=NIL THEN
      NEW(TEMP)
    ELSE
      BEGIN (*PREVIOUSLY USED STORAGE AVAILABLE*)
        TEMP:=AVAIL;
        AVAIL:=AVAIL↑.NEXT
      END;
    TEMP↑.TIME:=T;
    TEMP↑.PARAM:=Q;
  ...
```

Figure 7.21a

The principal changes required in the program are with respect to definition and handling of events and with respect to definition of the regeneration state. In addition to the completion event, we also need an event for arrivals from the source. There will always be exactly one such event in the event list assuming one source, and in general there will be an arrival event in the list for each source if there is more than one source. Each time an arrival event occurs we add another job to the network (i.e.,

```
  ...
PROCEDURE REMOVEFIRSTEVENT(VAR T:REAL; VAR Q: INTEGER);
  (*REMOVEFIRSTEVENT RETURNS TIME T AND PARAM Q OF FIRST
    EVENT*)
  VAR TEMP: ELEMPTR;
  BEGIN
    IF FIRST=NIL THEN
      BEGIN
        WRITELN('REMOVEFIRSTEVENT -- EMPTY LIST');
        HALT
      END
    ELSE
      BEGIN
        T:=FIRST↑.TIME;
        Q:=FIRST↑.PARAM;
  ...
FUNCTION ENDCYCLE: BOOLEAN;
(*DETERMINES WHETHER AT END OF REGENERATION CYCLE.
  IF SO, ENDCYCLE UPDATES ACCUMULATORS.*)
  VAR Q: INTEGER;
  BEGIN
    IF (TOTALLENGTH=0) AND (NUMBEREVENTS>0) THEN
  ...
```

Figure 7.21b

increment TOTALLENGTH), place that job at the first queue and schedule the next arrival event at the current time plus a sample from the interarrival time distribution. We need no event for sinks; each time a job leaves the last queue we remove it from the network (i.e., decrement TOTALLENGTH). Since the only information for the source arrival event is its time and type, and since the variable QUEUE with each event will never be zero for a completion event, we can use the value 0 to indicate a source arrival event. However, the old variable name is misleading, so we rename it PARAM and interpret an event as a source arrival event if PARAM is zero and otherwise interpret an event as a completion event for the queue identified by PARAM.

Since the number of jobs in the network is variable, we must consider the number of jobs in our regeneration state definition. As along as no queue is saturated, i.e., has an arrival rate greater than its service rate, the state where no jobs are in the network will be a regeneration state. Further, this state will be one of the most frequently occurring for many networks. To be more specific, the expected time between entrances to a Markov state is inversely proportional to the state's probability. Thus we

```
...
    QUEUES[1].MEANSERVICE:=1.0/B1;
    QUEUES[2].NUMBERSERVERS:=NIO;
    QUEUES[2].MEANSERVICE:=1.0/B2;
    TOTALLENGTH:=0;
    INSERTEVENT(-MEANINTERARRIVAL*LN(RANDOM(Z)),0);

    (*RUN*)
    WHILE (FIRST<>NIL) AND (NUMBEREVENTS<EVENTMAX)
        AND ((NUMBEREVENTS<EVENTLIMIT) OR NOT ENDCYCLE) DO
      BEGIN
        NUMBEREVENTS:=NUMBEREVENTS+1;
        REMOVEFIRSTEVENT(CLOCK,I);
        IF I=0 THEN
          BEGIN
            TOTALLENGTH:=TOTALLENGTH+1;
            ARRIVE(1);
            INSERTEVENT(CLOCK
                        -MEANINTERARRIVAL*LN(RANDOM(Z)),0)
          END
        ELSE
          BEGIN
            COMPLETE(I);
            IF I<>NQ THEN
              ARRIVE(I+1)
            ELSE
              TOTALLENGTH:=TOTALLENGTH-1
          END
      END;

    (*PRINT STATISTICS*)
...
```

Figure 7.21c

can loosely say that the expected frequency of occurrence of a Markov state is directly proportional to the state probability. For a queue with exponential interarrival times, exponential service times and service rate independent of queue length, i.e., a single fixed rate server, the queue length distribution is given by $P(N) = (1 - U)U^N$, $N = 0, 1, 2, \ldots$, where $U$ is the server utilization. Since $U$ is strictly less than 1, $P(0)$ must be the most probable queue length. This same result holds for LCFSPR and PS queues with a single fixed rate server and arbitrary service time distributions. (Note that the empty system will be a Markov state for such a system and that the probability of any other Markov state for such a system can be no greater

```
NUMBER OF EVENTS:      147 SIMULATED TIME:    412.487

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.878      0.132        2.488        22.435
    1       0.722      0.119        1.652        13.910
LOWER       0.567      0.106        0.817         5.384
UPPER       0.945      0.132        5.314        47.800
    2       0.810      0.119        3.398        28.607
LOWER       0.674      0.106        1.483         9.413

NUMBER OF CYCLES:        2
AVERAGE NUMBER OF EVENTS:     73.500
AVERAGE LENGTH:    206.244 C.I.:(   107.965,    304.522)

NUMBER OF EVENTS:     1362 SIMULATED TIME:   3400.368

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
    1       0.973      0.134       15.838       118.622
    2       0.627      0.134        1.986        14.874

NUMBER OF EVENTS:    10200 SIMULATED TIME: 27031.674

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.886      0.129       13.178       103.849
    1       0.843      0.126        8.026        63.810
LOWER       0.801      0.122        2.874        23.770
UPPER       0.658      0.129        2.280        17.854
    2       0.629      0.126        2.078        16.525
LOWER       0.600      0.122        1.877        15.196

NUMBER OF CYCLES:      129
AVERAGE NUMBER OF EVENTS:     79.070
AVERAGE LENGTH:    209.548 C.I.:(   143.271,    275.824)

NUMBER OF EVENTS:   100290 SIMULATED TIME: 266918.844

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.847      0.126        5.670        45.107
    1       0.837      0.125        5.131        40.969
LOWER       0.828      0.124        4.592        36.830
UPPER       0.633      0.126        2.142        17.042
    2       0.626      0.125        2.073        16.551
LOWER       0.618      0.124        2.004        16.060

NUMBER OF CYCLES:     1229
```

```
AVERAGE NUMBER OF EVENTS:     81.603
AVERAGE LENGTH:    217.184 C.I.:(    199.888,    234.480)
```

Figure 7.22

than $P(0)$.) This result also holds for queues with job classes. We can also show that the empty system is the most probable state for other types of queues. However, for queues with variable rate, e.g., multiserver queues, some other state may have higher probability. For any network satisfying Jackson's Theorem, i.e., that the probability of a network state is the product of the probabilities of the states of the individual queues, the most frequent state must be the one where each queue is in its most frequent state, e.g., the empty state. (See Chapter 4 for Jackson's Theorem.) The empty state will also be the most frequent state for some networks not satisfying Jackson's Theorem.

Using the parameters for queues 1 and 2 as before and a mean interarrival time of 8, the most probable queue length for queue 1 is 0 with a probability of $1/6 \approx 0.167$. For queue 2, $P(0) \approx 0.231$, $P(1) \approx 0.288$, and all other states have smaller probabilities. Thus the most frequently occurring network state will be the one with 0 jobs at queue 1 and 1 job at queue 2. But the empty network state has nearly the same expected frequency and is easier to test for. Thus we let ENDCYCLE determine the system is in the regeneration state if TOTALLENGTH = 0.

Figure 7.20 shows the expected values of the performance measures for this network. Even though the throughputs and utilizations are essentially the same as for our corresponding closed network, the queue lengths and queueing times are much higher. In general this system is much more variable than the corresponding closed system because of the variability in the total number of jobs in the system. Figures 7.21a, 7.21b and 7.21c show the modifications to the program of Figure 7.17. Figure 7.22 shows the output of this program for EVENTLIMIT = 100, 1000, 10000 and 100000 and EVENTMAX twice EVENTLIMIT. The first run consisted of only 2 regeneration cycles and its confidence intervals should be ignored. The second run consisted of only a single regeneration cycle and thus no confidence interval estimates were produced. The third run consisted of 129 cycles and thus its confidence interval estimates may be considered valid but useless because of their great width, at least for queue 1. For example, the queue 1 queueing time interval, (23.77,103.85), contains the expected value but has a width of 125% relative to the point estimate, and even greater width relative to the expected value. Only the final run, consisting of 1229 cycles, has usefully narrow confidence intervals for queue 1. For example, the relative width of the queueing time interval is 20%, with respect to the point estimate. (In comparing the runs for this model with those of the

closed model, we should remember that only 2/3 of the open network events are completions.)

### 7.3.2 Disciplines, Distributions, Classes and Routing

Up until now we have been content without explicit representations of the jobs and the queues. An explicit representation is also necessary if we wish to estimate characteristics of the queueing time (and response time) distributions other than the means, and an explicit representation may also be desirable for other reasons. Our data structure will be a simple linked list for each queue, with the elements of the lists representing the jobs. The elements contain information about the job specific to the current queue, for example the current job class, an indicator stating whether a sample has been taken from the service time distribution, the remaining service time if a sample has been taken, a pointer to the next job in the queue and a pointer to a pending event for this job, if one has been scheduled. The variables for this information will be called CURRENTNODE, REQUESTGRANTED, REQUEST, NEXTJOB, and EVENT, respectively. See the definition of type JOBELEMENT in Figure 7.23. (JOBELEMENT also has a variable SUBSERVER to be described shortly and other variables to be defined in Section 7.4.) The elements are moved from list to list as the jobs move from queue to queue. Each queue has a pointer to the beginning of the list, FIRSTINQUEUE, and a pointer to the end of the list, LASTINQUEUE. Each list is maintained in an order appropriate to the queueing discipline. For FCFS and LCFSPR, the appropriate orders are increasing time of arrival, and decreasing time of arrival, respectively. We do not need to record the time of arrival for this purpose, though we would want to do so if we were estimating characteristics of the queueing time distribution other than the mean. For PS the appropriate order is increasing values of REQUEST, as we will describe shortly.

Since the elements representing jobs now contain much of the important dynamic information in the program, the information associated with an event (other than the time) will be a pointer to the job which the event affects. (A second kind of event will be defined in Section 7.4.) Since we want to implement preemptive queueing disciplines, we need to be able to efficiently cancel a pending service completion event. We add a pointer to the previous event to the event elements, modify INSERTEVENT accordingly, and replace REMOVEFIRSTEVENT with a new procedure REMOVEEVENT. (See Figure 7.26.)

Thus far we have only used exponential distributions in our simulation examples. Incorporation of other distributions is a simple matter, providing we have an algorithm for obtaining samples from the distribution and providing we are not using the regenerative method for confidence intervals.

```
CONST M=2147483647.0; A=16807.0;
      NN=2; NQ=2; NJ=3; B1=0.15; B2=0.1; NIO=2;
TYPE RANDINT=1..2147483646;
     EVENTTYPE=(COMPLETION, NODEDEPARTURE);
     JOBPTR: ↑JOBELEMENT;
     EVENTPTR: ↑EVENTELEMENT;
     EVENTELEMENT=RECORD
                    KINDOFEVENT: EVENTTYPE;
                    TIME: REAL;
                    JOB: JOBPTR;
                    NEXT: EVENTPTR;
                    PREVIOUS: EVENTPTR
             END;

     JOBELEMENT=RECORD
                  CURRENTNODE: 1..NN;
                  REQUEST: REAL;
                  REQUESTGRANTED: BOOLEAN;
                  SUBSERVER: INTEGER;
                  NEXTJOB: JOBPTR;
                  TOKENHOLDER: JOBPTR;
                  PARENT, CHILD: JOBPTR;
                  EVENT: EVENTPTR
                END;

     ROUTINGELEMENT=RECORD
                      DESTINATION: 1..NN;
                      PROBABILITY: REAL;
                      NEXTROUTING: ↑ROUTINGELEMENT
                    END;
     REGENELEMENT=RECORD
                    NODEREGEN: 1..NN;
                    LENGTHREGEN: INTEGER;
                    NEXTREGEN: ↑REGENELEMENT
                  END;
```

Figure 7.23 - Constant and Type Declarations

We must take distributions into consideration with the regenerative method, and it will be most convenient if we restrict ourselves to distributions represented by the method of exponential stages described in Chapter 3. As we discussed in Chapter 3, the method of stages is very general in the sense that we can represent other distributions closely. But using this method we can still describe our system as a Markov process by incorporating the distribution stage in our state definition, and we can use a state of

```
VAR Z: RANDINT;
    TABLE: ARRAY[0..127] OF RANDINT;
    I: INTEGER;
    FIRSTEVENT, LASTEVENT, AVAILEVENT: EVENTPTR;
    CLOCK: REAL;
    QUEUES: ARRAY[1..NQ] OF
              RECORD
                DISCIPLINE: (FCFS,LCFSPR,PS);
                NUMBERUNITS: INTEGER;
                NUMBERSUBSERVERS: INTEGER;
                MEANSUBSERVICE: REAL;
                FIRSTINQUEUE: JOBPTR;
                LASTINQUEUE: JOBPTR;
                LENGTH: INTEGER;
                TIMELENGTHCHANGED: REAL;
                SUMTIMELENGTH: REAL;
                SUMBUSYTIME: REAL;
                NUMBERCOMPLETIONS: INTEGER;
                (*SUMS OF CYCLE VALUES*)
                (*BT=BUSYTIME;
                  SQ=SQUARED;
                  X=TIMES;
                  CL=CYCLELENTGH;
                  NC=NUMBERCOMPLETIONS;
                  TL=TIMELENGTH;*)
                BT: REAL;
                TL: REAL;
                NC: REAL;
                BTSQ: REAL;
                BTXCL: REAL;
                NCSQ: REAL;
                NCXCL: REAL;
                TLSQ: REAL;
                TLXCL: REAL;
                TLXNC: REAL
              END;
    RUN, NUMBEREVENTS, EVENTLIMIT, EVENTMAX: INTEGER;
    NOEVENTSDURINGCYCLES, NUMBERCYCLES, NOCYCM1: INTEGER;
    TIMECYCLESTARTED, CYCLELENGTH,
        SUMCL, SUMCLSQ, VARCL, DCL: REAL;
    UTIL, DUTIL, VARBT, COVARBTCL: REAL;
    TPUT, DTPUT, VARNC, COVARNCCL: REAL;
    QL, DQL, VARTL, COVARTLCL: REAL;
    QT, DQT, COVARTLNC: REAL;
    AVAILJOB, TEMPJOB: JOBPTR;
```

```
TEMPKIND: EVENTTYPE;
FIRSTREGEN, AVAILREGEN: ↑REGENELEMENT;
AVAILROUTING: ↑ROUTINGELEMENT;
NODES: ARRAY[1..NN] OF
         RECORD
           KINDOFNODE: (CLASS,ALLOCATE,RELEASE,
                        FISSION,FUSION);
           QUEUE: INTEGER;
           LENGTHNODE: INTEGER;
           FUSIONPTR: JOBPTR;
           ROUTINGPTR, CHILDROUTING: ↑ROUTINGELEMENT
         END;
```

Figure 7.24 - Variable Declarations

that process as our regeneration state. (If we use other distribution forms we will not be able to describe our system as a Markov process. If there are recurring system states such that no nonexponential times are in progress, then we may be able to use one of these as a regeneration state.) In Chapter 3 we discussed the method of stages in terms of subservers, with each subserver visit time having an exponential distribution. Thus each service time is a sum of one or more visits to a set of subservers, with each subserver visit time having an exponential distribution. Thus each service time is a sum of one or more exponential values, the selection of which may itself have been a random process. This is a very natural characterization from a simulation point of view. In fact, it is the standard characterization for simulation purposes because of the relative difficulty of obtaining the inverse distribution functions for these distributions. For example, the usual way to obtain a value from an Erlang distribution with k stages is to obtain k independent exponential values and sum them. Though the usual practice is to obtain the exponential values together, this is not necessary. Since we must include the jobs' current distribution stages in our regeneration state definition, we redefine our interpretation of the completion event to be the completion of a distribution stage, i.e., the completion of a visit to a subserver. The COMPLETE procedure (Figure 7.27) and the ARRIVE procedure (Figure 7.29) assume that service distributions at each queue are independent of job class and have an Erlang distribution with NUMBERSUBSERVER stages. The variable SUBSERVER in type JOBELEMENT keeps track of the current distribution stage of the job. The generalization to the branching Erlang distribution and class dependent distributions is left to the reader. An obvious efficiency improvement is also left as an exercise.

Since we did not provide class dependent service distributions, the job classes principally serve as nodes in the routing. (The allocate, release, fission and fusion nodes are defined in Section 7.4.) Still, we must consider

```
PROCEDURE INSERTEVENT(K: EVENTTYPE; T: REAL; J: JOBPTR);
  (*INSERTEVENT ADDS EVENT OF KIND K AT TIME T FOR JOB J TO
    LIST*)
  VAR TEMP, L: EVENTPTR;
  BEGIN
    IF AVAILEVENT=NIL THEN
      NEW(TEMP)
    ELSE
      BEGIN (*PREVIOUSLY USED STORAGE AVAILABLE*)
        TEMP:=AVAILEVENT;
        AVAILEVENT:=AVAILEVENT↑.NEXT
      END;
    TEMP↑.KINDOFEVENT:=K;
    TEMP↑.TIME:=T;
    TEMP↑.JOB:=J;
    J↑.EVENT:=TEMP;
    IF FIRSTEVENT=NIL THEN
      BEGIN (*LIST WAS EMPTY*)
        FIRSTEVENT:=TEMP;
        LASTEVENT:=TEMP;
        TEMP↑.NEXT:=NIL;
        TEMP↑.PREVIOUS:=NIL
      END
    ELSE IF T<FIRSTEVENT↑.TIME THEN
      BEGIN (*INSERT AT BEGINNING OF LIST*)
        TEMP↑.NEXT:=FIRSTEVENT;
        TEMP↑.PREVIOUS:=NIL;
        FIRSTEVENT↑.PREVIOUS:=TEMP;
        FIRSTEVENT:=TEMP
      END
    ELSE IF T≥LASTEVENT↑.TIME THEN
      BEGIN (*INSERT AT END OF LIST*)
        LASTEVENT↑.NEXT:=TEMP;
        TEMP↑.PREVIOUS:=LASTEVENT;
        LASTEVENT:=TEMP;
        TEMP↑.NEXT:=NIL
      END
    ELSE
      BEGIN (*INSERT SOMEWHERE IN MIDDLE OF LIST*)
        L:=FIRSTEVENT;
        WHILE T≥L↑.NEXT↑.TIME DO
          L:=L↑.NEXT;
        TEMP↑.NEXT:=L↑.NEXT;
        L↑.NEXT:=TEMP;
        TEMP↑.PREVIOUS:=L;
```

```
        TEMP↑.NEXT↑.PREVIOUS:=TEMP
      END;
  END;   (*INSERTEVENT*)
```

Figure 7.25 - INSERTEVENT

the number of jobs at each class in our regeneration state definition. Our new definition of ENDCYCLE makes some arbitrary restrictions on the choice of regeneration state; relaxation of these restrictions is left as an exercise. ENDCYCLE assumes that the regeneration state can be detected by checking the number of jobs at each class on the list of classes pointed to by FIRSTREGEN and by checking that each job which has begun service is in the first stage of the distribution, i.e., SUBSERVER = 1. Note that the first part of this assumption excludes regeneration states where more than one class has jobs at a FCFS or LCFSPR queue, since we would have to take the ordering of jobs as part of the regeneration state definition. The procedure ADDREGEN is used to add a class to the list of classes to be checked and to initially place jobs at those classes.

We have described the principal modification and additions to the data structures, with emphasis on the aspects relevant to the closed cyclic model we have previously simulated. We are about to discuss the aspects relevant to the other characteristics which motivated the revisions, but first let us consider the rest of the program for the previous cyclic model. There are two procedures for the routing to be described in detail later. NEXTNODE determines which node a job should go to next and ADDESTINATION is used to add a node to the list of jobs leaving another node. Figure 7.32 shows the rest of the program relevant to this model. Notice that there are no changes with respect to performance estimates. The output of this program is identical to that shown in Figure 7.18 for the previous version of the program. We have changed the variable NUMBERSERVERS to NUMBERUNITS because of an alternative meaning for passive queues in Section 7.4. Since a job may or may not be ready to leave a queue after a subservice completion, COMPLETE will change its job pointer parameter to NIL only if the job is not ready to leave. The procedures defined in Section 7.4 for other kinds of nodes will follow this convention indicating whether jobs can leave those nodes or not. (For this model and the remaining models of this section, we have assumed queue 1 consists only of class 1, queue 2 consists only of class 2, etc. This will not be true of the last model in Section 7.4.)

**The LCFSPR Queueing Discipline. (Last Come First Served Preemptive Resume).** As we said in Chapter 2, this discipline is principally of theoretical interest even though it has been used for CPU scheduling. It is a good example for our purposes because it illustrates the handling of preemption in

```
PROCEDURE REMOVEEVENT(E: EVENTPTR; VAR K: EVENTTYPE;
                      VAR T: REAL; VAR J: JOBPTR);
  (*REMOVEEVENT RETURNS KIND K, TIME T AND JOB J
    OF EVENT E*)
  VAR TEMP: EVENTPTR;
  BEGIN
    IF FIRSTEVENT=NIL THEN
      BEGIN
        WRITELN('REMOVEEVENT - EMPTY LIST');
        HALT
      END
    ELSE IF E=FIRSTEVENT THEN
      BEGIN
        K:=FIRSTEVENT↑.KINDOFEVENT;
        T:=FIRSTEVENT↑.TIME;
        J:=FIRSTEVENT↑.JOB;
        TEMP:=FIRSTEVENT;
        FIRSTEVENT:=FIRSTEVENT↑.NEXT;
        IF FIRSTEVENT=NIL THEN
          LASTEVENT:=NIL
        ELSE FIRSTEVENT↑.PREVIOUS:=NIL;
        TEMP↑.NEXT:=AVAILEVENT;
        AVAILEVENT:=TEMP
      END
    ELSE IF E=LASTEVENT THEN
      BEGIN
        K:=LASTEVENT↑.KINDOFEVENT;
        T:=LASTEVENT↑.TIME;
        J:=LASTEVENT↑.JOB;
        TEMP:=LASTEVENT;
        LASTEVENT:=LASTEVENT↑.PREVIOUS;
        LASTEVENT↑.NEXT:=NIL;
        TEMP↑.NEXT:=AVAILEVENT;
        AVAILEVENT:=TEMP
      END
    ELSE
      BEGIN
        TEMP:=FIRSTEVENT;
        WHILE (TEMP<>E) AND (TEMP<>NIL) DO
          TEMP:=TEMP↑.NEXT;
        IF TEMP<>E THEN
          BEGIN
            WRITELN('REMOVEEVENT - EVENT NOT FOUND');
            HALT
          END
```

```
        ELSE (*E IS BETWEEN FIRSTEVENT AND LASTEVENT*)
          BEGIN
            K:=TEMP↑.KINDOFEVENT;
            T:=TEMP↑.TIME;
            J:=TEMP↑.JOB;
            TEMP↑.NEXT↑.PREVIOUS:=TEMP↑.PREVIOUS;
            TEMP↑.PREVIOUS↑.NEXT:=TEMP↑.NEXT;
            TEMP↑.NEXT:=AVAILEVENT;
            AVAILEVENT:=TEMP
          END
      END
  END;   (*REMOVEEVENT*)
```

Figure 7.26 - REMOVEEVENT

an otherwise simple mechanism. In the single server case, whenever a job arrives at a queue and another job is in service, the job in service is preempted, the new job is placed at the beginning of the queue and the new job is assigned to the server. In procedure ARRIVE, the preemption is effected by a call to REMOVEEVENT. The remaining service time for the preempted job is the difference between the (future) time of the scheduled event and the current time. This time is stored in REQUEST for future use when the preempted job is reassigned to the server. The mechanism in COMPLETE is the same as for FCFS except that a new sample from the service time distribution is *not* obtained when a job is assigned to the server; the value stored in REQUEST by ARRIVE is used. The multiple server case is different only in that a job is preempted only if a server is not available and in that if a preemption does occur, the job in service which has been in the queue the longest is the one preempted.

**The PS Queueing Discipline. (Processor Sharing).** In Chapter 2 we defined PS as the limiting case of the Round Robin (RR) discipline without switching overhead as the quantum goes to zero. PS is used in numerically solved models in place of RR because of the resulting tractability. This substitution will usually have little effect on the performance estimates if the quantum actually used is large in comparison with the switching overhead and small in comparison with the mean service time. Though there are no difficult problems in implementing RR in a simulation, if the quantum is small relative to the mean service time, there will be a large number of preemptions resulting in significant computational expense.

A few observations lead to a fairly simple simulation implementation of PS. We assume a single observer and then generalize to the multiple server case. Since the server is shared equally among all jobs in the queue, if a job has a service time of $X$ and there are $L$ jobs in the queue the time to serve

```
PROCEDURE COMPLETE(VAR J: JOBPTR);
(*HANDLES COMPLETION OF SUBSERVER FOR JOB J.
  IF SERVICE COMPLETE, J REMAINS UNCHANGED.
  OTHERWISE J BECOMES NIL.*)
  VAR LENG: INTEGER; L: JOBPTR; T: REAL;
  BEGIN
    WITH QUEUES[NODES[J↑.CURRENTNODE].QUEUE] DO
      BEGIN
        IF J↑.SUBSERVER<NUMBERSUBSERVERS THEN
          BEGIN
            J↑.SUBSERVER:=J↑.SUBSERVER+1;
            IF (DISCIPLINE IN [FCFS,LCFSPR]) OR (LENGTH=1)
              THEN
              BEGIN
                J↑.REQUEST:=-MEANSUBSERVICE*LN(RANDOM(Z));
                INSERTEVENT(COMPLETION,CLOCK+J↑.REQUEST,J);
                J:=NIL
              END
            ELSE (*DISCIPLINE=PS*)
              BEGIN
                T:=J↑.REQUEST;
                J↑.REQUEST:=-MEANSUBSERVICE*LN(RANDOM(Z));
                FIRSTINQUEUE:=FIRSTINQUEUE↑.NEXTJOB;
                UPDATEPSQUEUE(NODES[J↑.CURRENTNODE].QUEUE,T,
                              J);
                INSERTEVENT(COMPLETION,
                            CLOCK+FIRSTINQUEUE↑.REQUEST*
                            LENGTH/MIN(LENGTH,NUMBERUNITS),
                            FIRSTINQUEUE);
                J:=NIL
              END
          END
        ELSE
          BEGIN
            (*STATISTICS*)
            NUMBERCOMPLETIONS:=NUMBERCOMPLETIONS+1;
            SUMTIMELENGTH:=SUMTIMELENGTH
                          +(CLOCK-TIMELENGTHCHANGED)*LENGTH;
            SUMBUSYTIME:=SUMBUSYTIME
                        +(CLOCK-TIMELENGTHCHANGED)
                        *MIN(LENGTH,NUMBERUNITS);
            TIMELENGTHCHANGED:=CLOCK;
            (*MECHANICS*)
            NODES[J↑.CURRENTNODE].LENGTHNODE:=
                    NODES[J↑.CURRENTNODE].LENGTHNODE-1;
```

```
LENGTH:=LENGTH-1;
IF (DISCIPLINE IN [FCFS,LCFSPR]) OR (LENGTH=0)
  THEN
  BEGIN
    IF J=FIRSTINQUEUE THEN
      BEGIN
        FIRSTINQUEUE:=FIRSTINQUEUE↑.NEXTJOB;
        IF FIRSTINQUEUE=NIL THEN
          LASTINQUEUE:=NIL
        ELSE
          BEGIN
            LENG:=1;
            L:=FIRSTINQUEUE
          END
      END
    ELSE
      BEGIN
        L:=FIRSTINQUEUE;
        LENG:=2;
        WHILE J<>L↑.NEXTJOB DO
          BEGIN
            LENG:=LENG+1;
            L:=L↑.NEXTJOB
          END;
        IF J↑.NEXTJOB=NIL THEN
          LASTINQUEUE:=L;
        L↑.NEXTJOB:=J↑.NEXTJOB;
        L:=L↑.NEXTJOB
      END;
    IF LENGTH≥NUMBERUNITS THEN
      BEGIN
        WHILE LENG<NUMBERUNITS DO
          BEGIN
            L:=L↑.NEXTJOB;
            LENG:=LENG+1
          END;
        IF NOT L↑.REQUESTGRANTED THEN
          BEGIN
            L↑.REQUEST:=-MEANSUBSERVICE
                        *LN(RANDOM(Z));
            L↑.REQUESTGRANTED:=TRUE
          END;
        INSERTEVENT(COMPLETION,CLOCK+L↑.REQUEST,
                    L)
      END
```

```
            END
          ELSE (*DISCIPLINE=PS*)
            BEGIN
              T:=J↑.REQUEST;
              FIRSTINQUEUE:=FIRSTINQUEUE↑.NEXTJOB;
              L:=FIRSTINQUEUE;
              WHILE L<>NIL DO
                BEGIN
                  L↑.REQUEST:=L↑.REQUEST-T;
                  L:=L↑.NEXTJOB
                END;
              INSERTEVENT(COMPLETION,
                          CLOCK+FIRSTINQUEUE↑.REQUEST*
                          LENGTH/MIN(LENGTH,NUMBERUNITS),
                          FIRSTINQUEUE)
            END
        END
    END
END;   (*COMPLETE*)
```

Figure 7.27 - COMPLETE

that job will be $LX$. (This assumes the queue length does not change during the job's service.) Further, the job with the smallest remaining service time must be the first to leave the queue. Thus we do not need to have events pending for each job in the queue, but rather can have an event pending for each job in the queue with the smallest service time and interpret this event as one for all of the jobs in the queue. When the event occurs, all of the jobs have received service equal to the request of the job with the smallest request. Thus that amount of time may be subtracted from the request of each job to obtain that job's remaining request. The job with the smallest request, which has had its request satisfied, leaves the queue and, if there are jobs still in the queue, a new event is scheduled at the current time plus $LX$, where $L$ is the new queue length and $X$ is the new smallest request. This implementation is the basis for our earlier statement that the list of jobs at the queue should be kept in order of increasing service requests.

There is one complication which arises when a job arrives at the queue when an event is already pending. That event was scheduled based on the queue length before the arrival, but now the jobs will progress at a slower rate. Thus we must cancel the pending event and subtract from each of the old jobs the service already received. This service already received is $X - (T - C)/L$, where $X$ is the smallest old service request, $T$ is the scheduled time of the pending event, $C$ is the current time and $L$ is the old

```
PROCEDURE UPDATEPSQUEUE(Q: INTEGER; T: REAL; J: JOBPTR);
(*SUBTRACTS T FROM REQUEST FOR JOBS CURRENTLY IN QUEUE Q.
  THEN INSERTS J IN THE QUEUE ACCORDING TO J↑.REQUEST*)
  VAR TEMP: JOBPTR;
  BEGIN
    WITH QUEUES[Q] DO
      BEGIN
        TEMP:=FIRSTINQUEUE;
        WHILE TEMP<>NIL DO
          BEGIN
            TEMP↑.REQUEST:=TEMP↑.REQUEST-T;
            TEMP:=TEMP↑.NEXTJOB
          END;
        IF J↑.REQUEST<FIRSTINQUEUE↑.REQUEST THEN
          BEGIN
            J↑.NEXTJOB:=FIRSTINQUEUE;
            FIRSTINQUEUE:=J
          END
        ELSE IF J↑.REQUEST≥LASTINQUEUE↑.REQUEST THEN
          BEGIN
            LASTINQUEUE↑.NEXTJOB:=J;
            J↑.NEXTJOB:=NIL;
            LASTINQUEUE:=J
          END
        ELSE
          BEGIN
            TEMP:=FIRSTINQUEUE;
            WHILE J↑.REQUEST≥TEMP↑.NEXTJOB↑.REQUEST DO
              TEMP:=TEMP↑.NEXTJOB;
            J↑.NEXTJOB:=TEMP↑.NEXTJOB;
            TEMP↑.NEXTJOB:=J
          END
      END
  END;  (*UPDATEPSQUEUE*)
```

Figure 7.28 - UPDATEPSQUEUE

order and a new event is scheduled based on what is now the smallest service request and the new queue length.

With the distribution sampled by stages, a similar action takes place upon completion of a subservice other than the last one for the job. The service received by the job with the completed stage is subtracted from all of the jobs, the service time for the next stage is obtained, and the job is reinserted in the queue. Then a new event is scheduled. Because of this

```
PROCEDURE ARRIVE(VAR J: JOBPTR; C: INTEGER);
(*HANDLES ARRIVAL OF A JOB J AT CLASS C.  J BECOMES NIL*)
  VAR DUMMYKIND: EVENTTYPE; T: REAL; DUMMYJOB, TEMP: JOBPTR;
      LENG: INTEGER;
  BEGIN
    J↑.CURRENTNODE:=C;
    J↑.SUBSERVER:=1;
    J↑.REQUESTGRANTED:=FALSE;
    WITH QUEUES[NODES[C].QUEUE] DO
      BEGIN
        (*STATISTICS*)
        SUMTIMELENGTH:=SUMTIMELENGTH
                       +(CLOCK-TIMELENGTHCHANGED)*LENGTH;
        SUMBUSYTIME:=SUMBUSYTIME+(CLOCK-TIMELENGTHCHANGED)*
                                  MIN(LENGTH,NUMBERUNITS);
        TIMELENGTHCHANGED:=CLOCK;
        (*MECHANICS*)
        IF (DISCIPLINE=FCFS) OR (FIRSTINQUEUE=NIL) THEN
          BEGIN
            J↑.NEXTJOB:=NIL;
            IF FIRSTINQUEUE=NIL THEN
              FIRSTINQUEUE:=J
            ELSE
              LASTINQUEUE↑.NEXTJOB:=J;
            LASTINQUEUE:=J;
            NODES[C].LENGTHNODE:=NODES[C].LENGTHNODE+1;
            LENGTH:=LENGTH+1;
            IF LENGTH≤NUMBERUNITS THEN
              BEGIN
                J↑.REQUEST:=-MEANSUBSERVICE*LN(RANDOM(Z));
                J↑.REQUESTGRANTED:=TRUE;
                INSERTEVENT(COMPLETION,CLOCK+J↑.REQUEST,J)
              END
          END
        ELSE IF DISCIPLINE=LCFSPR THEN
          BEGIN
            IF LENGTH=NUMBERUNITS THEN
              BEGIN (*PREEMPT LASTINQUEUE*)
                REMOVEEVENT(LASTINQUEUE↑.EVENT,
                            DUMMYKIND,T,DUMMYJOB);
                LASTINQUEUE↑.REQUEST:=T-CLOCK
              END
            ELSE IF LENGTH>NUMBERUNITS THEN
              BEGIN (*PREEMPT LAST JOB IN SERVICE*)
                LENG:=1;
```

```
                TEMP:=FIRSTINQUEUE;
                WHILE LENG<NUMBERUNITS DO
                  BEGIN
                    LENG:=LENG+1;
                    TEMP:=TEMP↑.NEXTJOB
                  END;
                REMOVEEEVENT(TEMP↑.EVENT,
                             DUMMYKIND,T,DUMMYJOB);
                TEMP↑.REQUEST:=T-CLOCK
              END;
            J↑.NEXTJOB:=FIRSTINQUEUE;
            FIRSTINQUEUE:=J;
            NODES[C].LENGTHNODE:=NODES[C].LENGTHNODE+1;
            LENGTH:=LENGTH+1;
            J↑.REQUEST:=-MEANSUBSERVICE*LN(RANDOM(Z));
            J↑.REQUESTGRANTED:=TRUE;
            INSERTEVENT(COMPLETION,CLOCK+J↑.REQUEST,J)
          END
        ELSE (*DISCIPLINE=PS*)
          BEGIN
            REMOVEEVENT(FIRSTINQUEUE↑.EVENT,
                        DUMMYKIND,T,DUMMYJOB);
            T:=FIRSTINQUEUE↑.REQUEST-(T-CLOCK)
               *MIN(LENGTH,NUMBERUNITS)/LENGTH;
            J↑.REQUEST:=-MEANSUBSERVICE*LN(RANDOM(Z));
            J↑.REQUESTGRANTED:=TRUE;
            UPDATEPSQUEUE(NODES[C].QUEUE,T,J);
            NODES[C].LENGTHNODE:=NODES[C].LENGTHNODE+1;
            LENGTH:=LENGTH+1;
            INSERTEVENT(COMPLETION,
                        CLOCK+FIRSTINQUEUE↑.REQUEST*LENGTH
                        /MIN(LENGTH,NUMBERUNITS),
                        FIRSTINQUEUE)
          END
      END;
    J:=NIL
  END;   (*ARRIVE*)
```

Figure 7.29 - ARRIVE

similarity we define a procedure UPDATEPSQUEUE which is used by both ARRIVE and COMPLETE.

The multiple server case is identical to the single server case except that events are scheduled at the current time plus $LX/\min(L,K)$, where $K$ is

```
FUNCTION ENDCYCLE: BOOLEAN;
(*DETERMINES WHETHER AT END OF REGENERATION CYCLE.
  IF SO, ENDCYCLE UPDATES ACCUMULATORS.*)
  VAR RESULT: BOOLEAN; TEMP: JOBPTR; L,Q: INTEGER;
      RTEMP: ↑REGENELEMENT;
  BEGIN
    IF FIRSTEVENT=NIL THEN
      BEGIN
        WRITELN('ENDCYCLE - EVENT LIST EMPTY');
        ENDCYCLE:=FALSE
      END
    ELSE
      BEGIN
        IF FIRSTEVENT↑.KINDOFEVENT=COMPLETION THEN
          RESULT:=TRUE
        ELSE
          RESULT:=FALSE;
        RTEMP:=FIRSTREGEN;
        WHILE RESULT AND (RTEMP<>NIL) DO
          BEGIN
            IF NODES[RTEMP↑.NODEREGEN].LENGTHNODE
                <>RTEMP↑.LENGTHREGEN THEN
              RESULT:=FALSE;
            RTEMP:=RTEMP↑.NEXTREGEN
          END;
        IF RESULT THEN
          BEGIN
            Q:=1;
            WHILE RESULT AND (Q≤NQ) DO
              BEGIN
                WITH QUEUES[Q] DO
                  IF LENGTH>0 THEN
                    IF NUMBERSUBSERVERS>1 THEN
                      BEGIN
                        IF DISCIPLINE=FCFS THEN
                          BEGIN
                            TEMP:=FIRSTINQUEUE;
                            L:=1;
                            WHILE RESULT AND
                              (L≤MIN(LENGTH,NUMBERUNITS))
                              DO
                              BEGIN
                                IF TEMP↑.SUBSERVER<>1 THEN
                                  RESULT:=FALSE;
                                L:=L+1;
```

```
                                TEMP:=TEMP↑.NEXTJOB
                              END
                          END
                        ELSE
                          BEGIN
                            TEMP:=FIRSTINQUEUE;
                            L:=1;
                            WHILE RESULT AND (L≤LENGTH) DO
                              BEGIN
                                IF TEMP↑.SUBSERVER<>1 THEN
                                  RESULT:=FALSE;
                                L:=L+1;
                                TEMP:=TEMP↑.NEXTJOB
                              END
                          END
                      END;
                Q:=Q+1
              END
          END;
        IF NUMBEREVENTS=0 THEN
          IF NOT RESULT AND
            (FIRSTEVENT↑.KINDOFEVENT=COMPLETION) THEN
            BEGIN
              WRITELN(
          'ENDCYCLE - NOT INITIALLY IN REGENERATION STATE');
              HALT
            END
          ELSE
            ENDCYCLE:=FALSE
        ELSE IF RESULT THEN
          BEGIN
            ENDCYCLE:=TRUE;
...
```

Figure 7.30 - ENDCYCLE

the number of servers. Similarly, when a pending event must be rescheduled, the amount of received service is $X - (T - C)\min(L,K)/L$.

Let us consider the cyclic queue model with LCFSPR at queue 1 and PS at queue 2. Further, let the mean service times be as before but the distributions be two stage Erlang. From our discussion in Chapter 5, we know that for the performance measures we are estimating the FCFS exponential results and the results from this revised model have the same expected values, those given in Figure 7.10. This is because the LCFSPR

```
PROCEDURE ADDREGEN(N,L: INTEGER);
(*INITIALIZES L JOBS AT NODE L.  SETS REGENERATION
  STATE DESCRIPTION TO HAVE L JOBS AT NODE L*)
  VAR TEMP: ↑REGENELEMENT; J: JOBPTR; I: INTEGER;
  BEGIN
    FOR I:=1 TO L DO
      BEGIN
        IF AVAILJOB=NIL THEN
          NEW(J)
        ELSE
          BEGIN
            J:=AVAILJOB;
            AVAILJOB:=AVAILJOB↑.NEXTJOB
          END;
        J↑.TOKENHOLDER:=NIL;
        J↑.PARENT:=NIL;
        J↑.CHILD:=NIL;
        ARRIVE(J,N)
      END;
    IF AVAILREGEN=NIL THEN
      NEW(TEMP)
    ELSE
      BEGIN
        TEMP:=AVAILREGEN;
        AVAILREGEN:=AVAILREGEN↑.NEXTREGEN
      END;
    TEMP↑.NODEREGEN:=N;
    TEMP↑.LENGTHREGEN:=L;
    TEMP↑.NEXTREGEN:=FIRSTREGEN;
    FIRSTREGEN:=TEMP
  END; (*ADDREGEN*)
```

Figure 7.31 - ADDREGEN

and PS results are dependent only on the mean service time and not on other distribution characteristics. We are using this network to empirically verify this result; we are not proposing it as a computer system model. Figure 7.33 shows the model specific parameters for this network. Since there are two events for each service completion, we doubled the event limits for the runs of this model. The results of these runs are shown in Figure 7.34. We ignore the confidence intervals for the first run because of the small number of regeneration cycles. The confidence intervals from the second and third runs contain the expected values in Figure 7.10, so these empirical results support the discussion of Chapter 5.

```
BEGIN
  (*INITIALIZATION*)
  Z:=314159;           (*AN ARBITRARY VALUE*)
  FOR I:=0 TO 127 DO   (*INITIALIZE TABLE  *)
    BEGIN
      (*Z:=(A*Z) MOD M*) Z:=TRUNC(A*Z - (TRUNC((A*Z)/M)*M));
      TABLE[I]:=Z
    END;
  AVAILEVENT:=NIL;
  AVAILJOB:=NIL;
  AVAILROUTING:=NIL;
  AVAILREGEN:=NIL;
  EVENTLIMIT:=10;
  FOR RUN:=1 TO 3 DO
    BEGIN
      FIRSTEVENT:=NIL;
      LASTEVENT:=NIL;
      CLOCK:=0.0;
      NUMBEREVENTS:=0;
      FIRSTREGEN:=NIL;
      NUMBERCYCLES:=0;
      TIMECYCLESTARTED:=0.0;
      SUMCL:=0.0;
      SUMCLSQ:=0.0;
      EVENTLIMIT:=10*EVENTLIMIT;
      EVENTMAX:=2*EVENTLIMIT;
      FOR I:=1 TO NQ DO
        WITH QUEUES[I] DO
          BEGIN
            DISCIPLINE:=FCFS;
            NUMBERSUBSERVERS:=1;
            NUMBERUNITS:=1;
            FIRSTINQUEUE:=NIL;
            LASTINQUEUE:=NIL;
            LENGTH:=0;
            TIMELENGTHCHANGED:=0.0;
            SUMTIMELENGTH:=0.0;
            SUMBUSYTIME:=0.0;
            NUMBERCOMPLETIONS:=0;
            BT:=0.0;
            TL:=0.0;
            NC:=0.0;
            BTSQ:=0.0;
            BTXCL:=0.0;
            NCSQ:=0.0;
```

```
        NCXCL:=0.0;
        TLSQ:=0.0;
        TLXCL:=0.0;
        TLXNC:=0.0
      END;
  FOR I:=1 TO NN DO
    WITH NODES[I] DO
      BEGIN
        KINDOFNODE:=CLASS;
        QUEUE:=I;
        LENGTHNODE:=0;
        ROUTINGPTR:=NIL;
        FUSIONPTR:=NIL;
        CHILDROUTING:=NIL
      END;
  (*PARAMETERS SPECIFIC TO THIS MODEL*)
  ADDDESTINATION(1,2,1.0,FALSE);
  QUEUES[1].MEANSUBSERVICE:=1.0/B1;
  ADDDESTINATION(2,1,1.0,FALSE);
  QUEUES[2].NUMBERUNITS:=NIO;
  QUEUES[2].MEANSUBSERVICE:=1.0/B2;
  ADDREGEN(1,NJ);

  (*RUN*)
  WHILE (FIRSTEVENT<>NIL) AND (NUMBEREVENTS<EVENTMAX)
    AND ((NUMBEREVENTS<EVENTLIMIT) OR NOT ENDCYCLE) DO
    BEGIN
      REMOVEEVENT(FIRSTEVENT,TEMPKIND,CLOCK,TEMPJOB);
      IF TEMPKIND=COMPLETION THEN
        BEGIN
          NUMBEREVENTS:=NUMBEREVENTS+1;
          COMPLETE(TEMPJOB)
        END;
      WHILE TEMPJOB<>NIL DO
        BEGIN
          I:=NEXTNODE(TEMPJOB);
          CASE NODES[I].KINDOFNODE OF
            CLASS: ARRIVE(TEMPJOB,I);
            ALLOCATE: ALLOC(TEMPJOB,I);
            RELEASE: RELEAS(TEMPJOB,I);
            FISSION: FISS(TEMPJOB,I);
            FUSION: FUS(TEMPJOB,I)
          END
        END
    END;
```

```
      (*PRINT STATISTICS*)
  ...
      (*PUT LEFTOVERS ON AVAIL LISTS*)
      IF FIRSTEVENT<>NIL THEN
        BEGIN
          LASTEVENT↑.NEXT:=AVAILEVENT;
          AVAILEVENT:=FIRSTEVENT
        END;
      FREEJOBS;
      FREEROUTING;
      FREEREGEN
    END
END.
```

Figure 7.32 - Program Body

```
      (*PARAMETERS SPECIFIC TO THIS MODEL*)
      ADDDESTINATION(1,2,1.0,FALSE);
      QUEUES[1].DISCIPLINE:=LCFSPR;
      QUEUES[1].NUMBERSUBSERVERS:=2;
      QUEUES[1].MEANSUBSERVICE:=0.5/B1;
      ADDDESTINATION(2,1,0.5,FALSE);
      QUEUES[2].DISCIPLINE:=PS;
      QUEUES[2].NUMBERUNITS:=NIO;
      QUEUES[2].NUMBERSUBSERVERS:=2;
      QUEUES[2].MEANSUBSERVICE:=0.5/B2;
      ADDREGEN(1,NJ);
```

Figure 7.33

**Routing**. We would like to simulate networks where a job leaving class $i$ is routed to class $j$ with probability $p_{ij}$. Conceptually this is the problem of sampling from a discrete distribution discussed in Section 7.1.1 and depicted in Figure 7.4. There we said that the selection would be the smallest value $j$ such that $u_0 \leq q_{ij}$, where $u_0$ is a sample from the uniform distribution on the (0,1) interval and $q_{ij}$ is the cumulative probability, i.e., $p_{i1} + p_{i2} + \ldots + p_{ij}$. The method used in function NEXTNODE (see Figure 7.35) is algebraically equivalent, somewhat more convenient, and somewhat less efficient. It is used with the assumption that the number of possible destinations is small enough that the efficiency loss is negligible. Rather than obtain the cumulative probability in preparation for calls to NEXTNODE, we subtract the individual probabilities from $u_0$ until we find j such that $u_0 \leq p_{ij}$. (Notice that the PARENT variable for a job will always be NIL for jobs created by ADDREGEN. This variable will have other values only for jobs created at fission nodes, as defined in Section 7.4.)

```
NUMBER OF EVENTS:      240 SIMULATED TIME:    517.452

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.904      0.133        2.015         17.611
    1       0.814      0.115        1.702         14.683
LOWER       0.724      0.098        1.390         11.756
UPPER       0.672      0.133        1.609         14.784
    2       0.555      0.115        1.297         11.188
LOWER       0.439      0.098        0.984          7.593

NUMBER OF CYCLES:        8
AVERAGE NUMBER OF EVENTS:     30.000
AVERAGE LENGTH:     64.681 C.I.:(     32.898,     96.464)

NUMBER OF EVENTS:     2096 SIMULATED TIME:   4234.877

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.831      0.127        1.608         13.095
    1       0.810      0.123        1.550         12.531
LOWER       0.789      0.120        1.492         11.967
UPPER       0.650      0.127        1.507         12.287
    2       0.629      0.123        1.449         11.714
LOWER       0.609      0.120        1.391         11.140

NUMBER OF CYCLES:       28
AVERAGE NUMBER OF EVENTS:     74.857
AVERAGE LENGTH:    151.245 C.I.:(     99.635,    202.855)

NUMBER OF EVENTS:   20124 SIMULATED TIME: 41292.096

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.821      0.123        1.619         13.385
    1       0.814      0.121        1.597         13.113
LOWER       0.807      0.120        1.575         12.842
UPPER       0.616      0.123        1.424         11.723
    2       0.608      0.121        1.402         11.508
LOWER       0.599      0.120        1.380         11.294

NUMBER OF CYCLES:      350
AVERAGE NUMBER OF EVENTS:     57.497
AVERAGE LENGTH:    117.977 C.I.:(    106.016,    129.937)
```

Figure 7.34

Procedure ADDDESTINATION of Figure 7.36 is used to build the list of possible destinations for each class. (The parameter C is only true for possible destinations of jobs created at fission nodes.)

```
FUNCTION NEXTNODE(J: JOBPTR): INTEGER;
(*FINDS THE NEXT NODE FOR JOB J TO GO TO*)
  VAR PROB: REAL; TEMP: ↑ROUTINGELEMENT;
  BEGIN
    IF (NODES[J↑.CURRENTNODE].KINDOFNODE=FISSION) AND
      (J↑.PARENT<>NIL) THEN
      TEMP:=NODES[J↑.CURRENTNODE].CHILDROUTING
    ELSE
      TEMP:=NODES[J↑.CURRENTNODE].ROUTINGPTR;
    IF TEMP=NIL THEN
      BEGIN
        WRITELN('NEXTNODE - UNDEFINED ROUTING FROM NODE',
                J↑.CURRENTNODE);
        HALT
      END;
    IF TEMP↑.PROBABILITY<1.0 THEN
      BEGIN
        PROB:=RANDOM(Z);
        WHILE (PROB>TEMP↑.PROBABILITY) AND
          (TEMP↑.NEXTROUTING<>NIL) DO
          BEGIN
            PROB:=PROB-TEMP↑.PROBABILITY;
            TEMP:=TEMP↑.NEXTROUTING
          END
      END;
    NEXTNODE:=TEMP↑.DESTINATION
  END;  (*NEXTNODE*)
```

Figure 7.35 - NEXTNODE

We have considered several models with such probabilistic routing in earlier chapters. Figure 7.37 shows a model similar in structure to the model Brown *et al* used as a model of an interactive computer system. The figure is simpler than their model in several aspects, most notably in that memory contention is ignored. Let us suppose this is a model of a small system used for rather simple purposes, e.g., text editing. There are ten users at terminals. Each user thinks for a moment, keys in a command and waits for a response. Upon receiving a response, the user repeats this cycle. The mean time for thinking and keying has an exponential distribution with mean 3 seconds. Each command requires an average of ten cycles of alternating CPU-I/O activity. CPU scheduling is PS and the CPU service times

```
PROCEDURE ADDDESTINATION(I,J: INTEGER; P:REAL; C: BOOLEAN);
(*ADDS DESTINATION NODE J TO ROUTING LIST
  FOR NODE I WITH PROBABILITY P.
  IF C THEN ROUTING IS FOR CHILD, OTHERWISE PARENT*)
  VAR TEMP: ↑ROUTINGELEMENT;
  BEGIN
    IF AVAILROUTING=NIL THEN
      NEW(TEMP)
    ELSE
      BEGIN
        TEMP:=AVAILROUTING;
        AVAILROUTING:=AVAILROUTING↑.NEXTROUTING
      END;
    TEMP↑.PROBABILITY:=P;
    TEMP↑.DESTINATION:=J;
    IF C THEN
      BEGIN
        TEMP↑.NEXTROUTING:=NODES[I].CHILDROUTING;
        NODES[I].CHILDROUTING:=TEMP
      END
    ELSE
      BEGIN
        TEMP↑.NEXTROUTING:=NODES[I].ROUTINGPTR;
        NODES[I].ROUTINGPTR:=TEMP
      END
  END;   (*ADDDESTINATION*)
```

Figure 7.36 - ADDDESTINATION

are exponential with mean 50 ms. Each disk is chosen with probability 0.5, disk scheduling is FCFS and the mean disk service is 60 ms. Figure 7.38 shows the expected values for the individual queue measures. The expected length of a think-key-response cycle can be obtained by use of Little's Rule, i.e., 10/1.72 = 5.81 seconds. Figure 7.39 gives the model specific statements for this network (with NN = NQ = 4) and Figure 7.40 shows the program output with EVENTLIMIT = 40000. (Note that scheduling is irrelevant at queue 1 because there is always a server for each job. We use PS there to keep the event list small.)

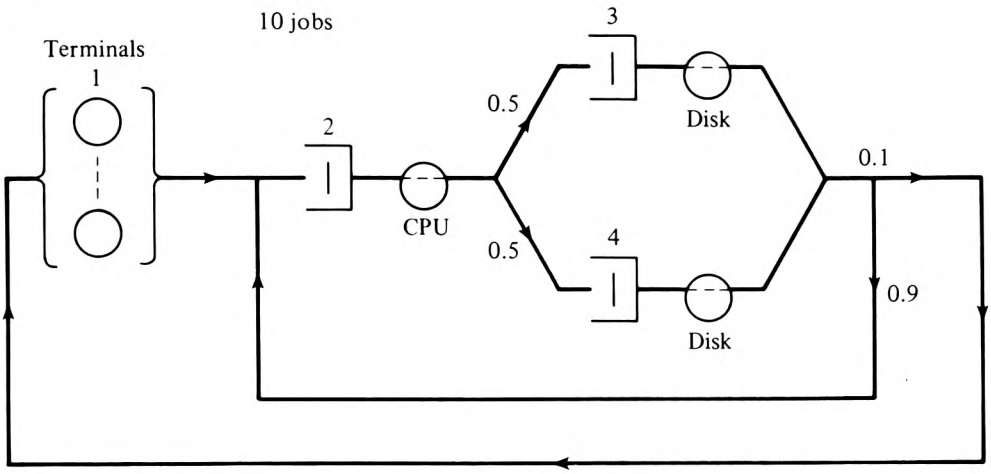



Figure 7.37 - Central Server Model with Terminals

| Queue | *U* | *R* | *L* | *Q* |
|---|---|---|---|---|
| 1 | 0.517 | 1.72 | 5.17 | 3.00 |
| 2 | 0.861 | 17.2 | 2.91 | 0.17 |
| 3 | 0.517 | 8.61 | 0.96 | 0.11 |
| 4 | 0.517 | 8.61 | 0.96 | 0.11 |

Figure 7.38 - Expected Values

```
(*PARAMETERS SPECIFIC TO THIS MODEL*)
ADDDESTINATION(1,2,1.0,FALSE);
QUEUES[1].DISCIPLINE:=PS;
QUEUES[1].NUMBERUNITS:=10;
QUEUES[1].MEANSUBSERVICE:=3.0;
ADDDESTINATION(2,3,0.5,FALSE);
ADDDESTINATION(2,4,0.5,FALSE);
QUEUES[2].DISCIPLINE:=PS;
QUEUES[2].MEANSUBSERVICE:=0.050;
ADDDESTINATION(3,1,0.1,FALSE);
ADDDESTINATION(3,2,0.9,FALSE);
QUEUES[3].MEANSUBSERVICE:=0.060;
ADDDESTINATION(4,1,0.1,FALSE);
ADDDESTINATION(4,2,0.9,FALSE);
QUEUES[4].MEANSUBSERVICE:=0.060;
ADDREGEN(1,10);
```

Figure 7.39

```
NUMBER OF EVENTS:    41076 SIMULATED TIME:   1117.410

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.523      1.778       5.236         3.030
    1       0.504      1.721       5.046         2.930
LOWER       0.485      1.665       4.855         2.831
UPPER       0.872     17.790       3.108         0.176
    2       0.859     17.519       2.958         0.168
LOWER       0.846     17.247       2.808         0.160
UPPER       0.535      8.952       1.040         0.117
    3       0.521      8.755       0.985         0.112
LOWER       0.507      8.557       0.931         0.107
UPPER       0.538      8.940       1.046         0.118
    4       0.528      8.764       1.009         0.115
LOWER       0.517      8.587       0.972         0.112

NUMBER OF CYCLES:       42
AVERAGE NUMBER OF EVENTS:    978.000
AVERAGE LENGTH:     26.605 C.I.:(     18.321,     34.888)
```

Figure 7.40

## 7.4 DEFINITION AND SIMULATION OF EXTENDED QUEUEING NETWORKS

The principal advantage of simulation is its generality; with enough investment of human and machine resources we can produce very realistic models of systems. This generality leads to one of simulation's greatest liabilities, the difficulty of deciding which system characteristics to try to represent and which to ignore. With computer system models, the queueing networks we have discussed provide a reasonable, relatively abstract approach to this model formulation problem. In Chapter 6 we discussed some of the limitations of the usual characteristics of queueing networks with respect to computer system models. This section will discuss extensions to queueing networks which overcome some of these limitations. The exercises will deal with other extensions which can be used to overcome the other limitations we cited. We are proposing extended queueing networks as a unified approach to computer systems. We choose to discuss two extensions which we consider both useful and relatively tricky to implement: passive queues and fusion nodes.

### 7.4.1 Passive Resources

In Chapter 6 we termed some resources (and their associated queues) as "passive" because the holding of these resources is determined by activi-

ties at other ("active") resources. An active resource and one or more passive resources may be held simultaneously by a job. The job needs the passive resources to use the active resources; time of holding the passive resources to use the active resources, routing, etc. A natural example of a passive resource in a computer system is primary memory. A job needs memory to use a CPU or I/O device; the job's time in memory is largely determined by CPU and I/O times.

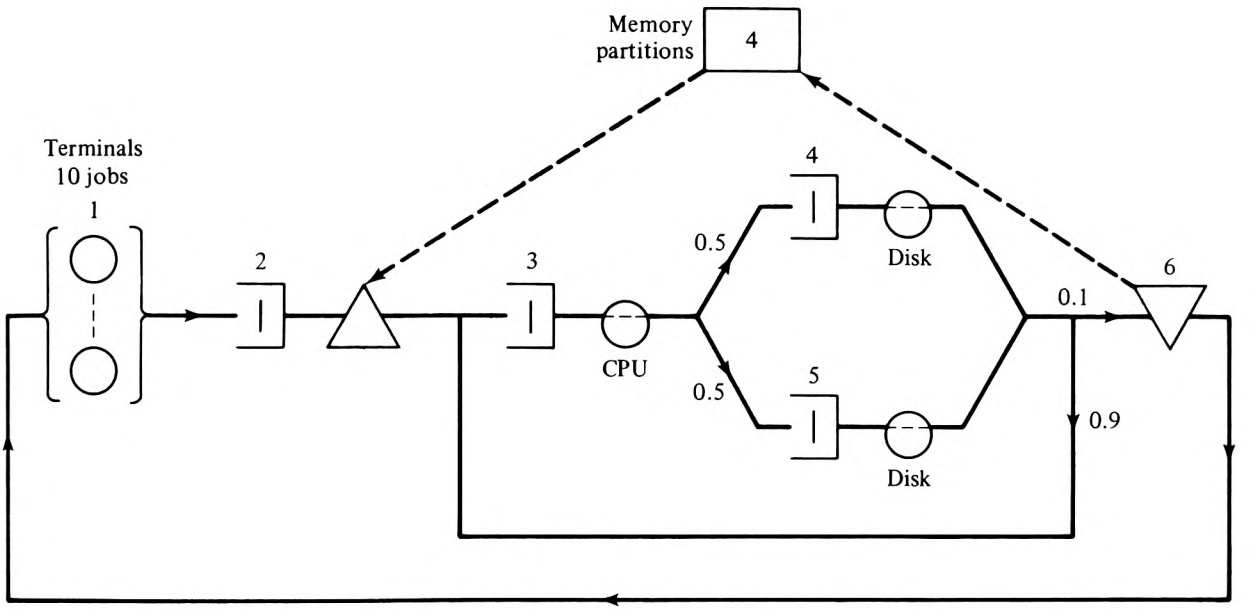



Figure 7.41
Central Server Model with Terminals and Memory

We define a passive queue as a set of *tokens* analogous to the servers of an active queue, a set of *allocate nodes* and a set of *release nodes*. A job wanting a token goes to an allocate node. If a token is available, the job receives the token and proceeds without delay. Tokens are allocated in FCFS order. When a job is through with a token, it proceeds to a release node where it returns the token and proceeds without delay. (This is a very restricted definition. A general definition is given in [SAUE77c].) Figure 7.41 shows our interactive computer system model with a passive queue added to represent memory contention. It is assumed that memory is divided into a fixed number of partitions and that a token represents a partition. Node 2 is an allocate node and node 6 is a release node. Figure 7.42 gives performance estimates for this model obtained by a flow-equivalence approximation discussed in Chapter 6. (Queue 2 is the passive queue.) The queueing time for a passive queue is defined as the time of request for tokens to the time of release of tokens. Thus the queueing time

for the passive queue in this model is also our estimate of mean response time, and this estimate is noticeably higher than our 2.81 estimate without memory contention.

| Queue | *U* | *R* | *L* | *Q* |
|---|---|---|---|---|
| 1 | 0.493 | 1.64 | 4.93 | 3.00 |
| 2 | 0.904 | 1.64 | 5.04 | 3.08 |
| 3 | 0.822 | 16.4 | 1.98 | 0.12 |
| 4 | 0.493 | 8.22 | 0.82 | 0.10 |
| 5 | 0.493 | 8.22 | 0.82 | 0.10 |

Figure 7.42 - Approximation Values

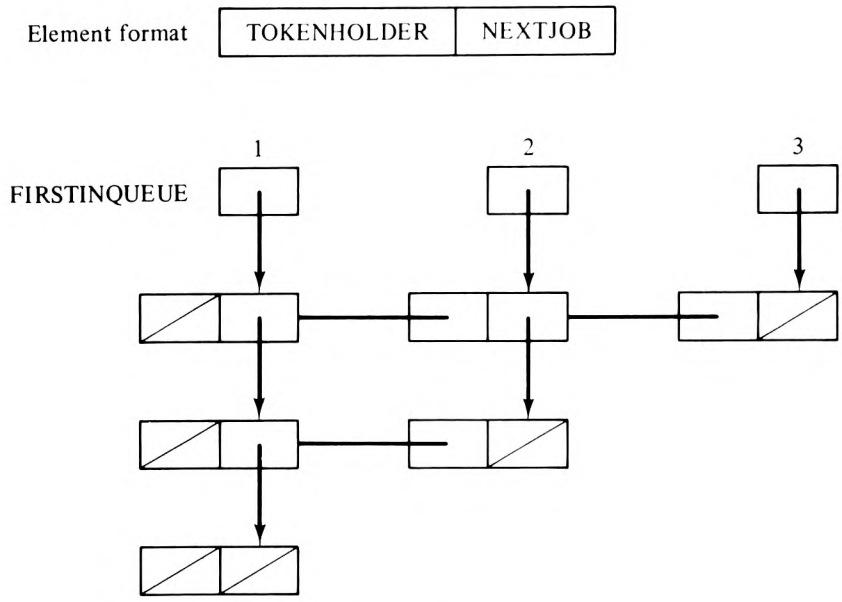



Figure 7.43 - Tokenholders

The principal problem with implementation of passive queues is that a job must be in several queues at the same time, i.e., in our data structure the job must appear in several lists. To accomplish this, we have several instances of type JOBELEMENT which collectively represent the job. There is exactly one entry in a queue list for each resource (active or passive) that the job has requested or possesses. These list elements collectively representing the job are also linked to each other with the TOKENHOLDER variable. We refer to the elements at the passive queues where the job holds tokens as "tokenholders" for the job. When a job is to be allocated a token, a new job element is obtained, this new element is placed in the queue where the job was, the new element is put in the job's list of

```
PROCEDURE ALLOC(VAR J: JOBPTR; A: INTEGER);
(*HANDLES ARRIVAL OF JOB J AT ALLOCATE NODE A*)
  VAR TH: JOBPTR;
  BEGIN
    J↑.CURRENTNODE:=A;
    WITH QUEUES[NODES[A].QUEUE] DO
      BEGIN
        (*STATISTICS*)
        SUMTIMELENGTH:=SUMTIMELENGTH
                       +(CLOCK-TIMELENGTHCHANGED)*LENGTH;
        SUMBUSYTIME:=SUMBUSYTIME+(CLOCK-TIMELENGTHCHANGED)*
                                    MIN(LENGTH,NUMBERUNITS);
        TIMELENGTHCHANGED:=CLOCK;
        (*MECHANICS*)
        NODES[A].LENGTHNODE:=NODES[A].LENGTHNODE+1;
        LENGTH:=LENGTH+1;
        IF LENGTH≤NUMBERUNITS THEN
          BEGIN
            IF AVAILJOB=NIL THEN
              NEW(TH)
            ELSE
              BEGIN
                TH:=AVAILJOB;
                AVAILJOB:=AVAILJOB↑.NEXTJOB
              END;
            TH↑.CURRENTNODE:=A;
            TH↑.REQUESTGRANTED:=TRUE;
            TH↑.TOKENHOLDER:=J↑.TOKENHOLDER;
            J↑.TOKENHOLDER:=TH;
            IF FIRSTINQUEUE=NIL THEN
              FIRSTINQUEUE:=TH
            ELSE
              LASTINQUEUE↑.NEXTJOB:=TH;
            LASTINQUEUE:=TH;
            TH↑.NEXTJOB:=NIL
          END
        ELSE
          BEGIN
            IF FIRSTINQUEUE=NIL THEN
              FIRSTINQUEUE:=J
            ELSE
              LASTINQUEUE↑.NEXTJOB:=J;
            LASTINQUEUE:=J;
            J↑.NEXTJOB:=NIL;
            J↑.REQUESTGRANTED:=FALSE;
```

```
            J:=NIL
          END
      END
END;   (*ALLOC*)
```

Figure 7.44 - ALLOC

tokenholders, and the job proceeds. Figure 7.43 shows the queue and tokenholder lists for a hypothetical state of a hypothetical system. The job at queue 3 holds tokens at both queues 1 and 2. The second job at queue 2 is waiting for a token there but holds a token at queue 1. The third job at queue 1 holds no tokens. Figure 7.44 shows the ALLOC procedure for allocate nodes. ALLOC sets its parameter J to NIL if a token is not allocated. (Refer back to Figure 7.42.) Figure 7.45 shows the RELEAS procedure for release nodes. If the job possesses no tokens from the release node's queue, there is no effect on the job. RELEAS searches the tokenholder list to find a tokenholder for its queue. If a tokenholder is found, the token is returned to the queue. RELEAS checks to see if a job is waiting for a token, and if so, reuses the tokenholder for that job. Otherwise the tokenholder is put on the AVAILJOB list.

The implementation of the release node encounters another problem: we may have more than one job moving around the network at the same simulated time, i.e., both the releasing job and the job just allocated a token. (If the releasing job then releases a token at another queue, or if we have a more general definition of passive queues, there may be more than two jobs simultaneously moving.) There are a variety of ways we could deal with this, e.g., by keeping a list of jobs in motion or by recursive procedure calls, but the most convenient way is to define a new kind of event, "nodedeparture" and to schedule such an event for each job set in motion because of the activities of another job. Though this may increase the size of our event list, this approach is easily generalized, is likely to be more efficient than recursive procedure calls, and can be applied to other situations such as the fission nodes about to be defined. We would like to maintain the property that our events correspond directly to state transitions of our implicit Markov process, so we will not think of nodedeparture events as "real" events and will not count them in NUMBEREVENTS.

Figure 7.46 show this model specific statements for the model of Figure 7.41, and Figure 7.47 shows the program output with EVENTLIMIT = 40000, as before. We are using state with all jobs at the terminals as the regeneration state, as we did before. Even though this is certainly not the most frequently occurring state, we do observe a reasonable number of regeneration cycles. Though the results for this model in Figure 7.42 are

```
PROCEDURE RELEAS(VAR J: JOBPTR; R: INTEGER);
(*HANDLES ARRIVAL OF JOB J AT RELEASE NODE R*)
  VAR FOUND: BOOLEAN; TH, TEMP, TEMPNEXT: JOBPTR;
  BEGIN
    J↑.CURRENTNODE:=R;
    FOUND:=FALSE;
    TEMP:=J;
    WHILE NOT FOUND AND (TEMP↑.TOKENHOLDER<>NIL) DO
      IF NODES[TEMP↑.TOKENHOLDER↑.CURRENTNODE].QUEUE=
        NODES[R].QUEUE THEN
        FOUND:=TRUE
      ELSE
        TEMP:=TEMP↑.TOKENHOLDER;
    IF FOUND THEN
      BEGIN
        TH:=TEMP↑.TOKENHOLDER;
        TEMP↑.TOKENHOLDER:=TH↑.TOKENHOLDER;
        WITH QUEUES[NODES[R].QUEUE] DO
          BEGIN
            (*STATISTICS*)
            NUMBERCOMPLETIONS:=NUMBERCOMPLETIONS+1;
            SUMTIMELENGTH:=SUMTIMELENGTH
                          +(CLOCK-TIMELENGTHCHANGED)*LENGTH;
            SUMBUSYTIME:=SUMBUSYTIME
                        +(CLOCK-TIMELENGTHCHANGED)
                        *MIN(LENGTH,NUMBERUNITS);
            TIMELENGTHCHANGED:=CLOCK;
            (*MECHANICS*)
            NODES[TH↑.CURRENTNODE].LENGTHNODE:=
                    NODES[TH↑.CURRENTNODE].LENGTHNODE-1;
            LENGTH:=LENGTH-1;
            IF TH=FIRSTINQUEUE THEN
              BEGIN
                FIRSTINQUEUE:=FIRSTINQUEUE↑.NEXTJOB;
                IF FIRSTINQUEUE=NIL THEN
                  LASTINQUEUE:=NIL
              END
            ELSE
              BEGIN
                TEMP:=FIRSTINQUEUE;
                WHILE TH<>TEMP↑.NEXTJOB DO
                  TEMP:=TEMP↑.NEXTJOB;
                TEMP↑.NEXTJOB:=TH↑.NEXTJOB;
                IF TEMP↑.NEXTJOB=NIL THEN
                  LASTINQUEUE:=TEMP
```

```
            END;
          IF LENGTH≥NUMBERUNITS THEN
            IF NOT FIRSTINQUEUE↑.REQUESTGRANTED THEN
              BEGIN
                (*MAKE TH A TOKENHOLDER FOR FIRSTINQUEUE*)
                TH↑.NEXTJOB:=FIRSTINQUEUE↑.NEXTJOB;
                IF TH↑.NEXTJOB=NIL THEN
                  LASTINQUEUE:=TH;
                TH↑.TOKENHOLDER:=
                        FIRSTINQUEUE↑.TOKENHOLDER;
                TH↑.REQUESTGRANTED:=TRUE;
                TEMP:=FIRSTINQUEUE;
                FIRSTINQUEUE:=TH;
                TEMP↑.TOKENHOLDER:=TH;
                INSERTEVENT(NODEDEPARTURE,CLOCK,TEMP)
              END
            ELSE
              BEGIN (*MAKE TH A TOKENHOLDER FOR TEMPNEXT*)
                TEMP:=FIRSTINQUEUE;
                WHILE TEMP↑.NEXTJOB↑.REQUESTGRANTED DO
                  TEMP:=TEMP↑.NEXTJOB;
                TEMPNEXT:=TEMP↑.NEXTJOB;
                TEMP↑.NEXTJOB:=TH;
                TH↑.NEXTJOB:=TEMPNEXT↑.NEXTJOB;
                IF TH↑.NEXTJOB=NIL THEN
                  LASTINQUEUE:=TH;
                TH↑.TOKENHOLDER:=TEMPNEXT↑.TOKENHOLDER;
                TH↑.REQUESTGRANTED:=TRUE;
                TEMPNEXT↑.TOKENHOLDER:=TH;
                INSERTEVENT(NODEDEPARTURE,CLOCK,TEMPNEXT)
              END
          ELSE
            BEGIN
              TH↑.NEXTJOB:=AVAILJOB;
              AVAILJOB:=TH
            END
        END
    END
END;  (*RELEAS*)
```

Figure 7.45 - **RELEAS**

```
(*PARAMETERS SPECIFIC TO THIS MODEL*)
ADDDESTINATION(1,2,1.0,FALSE);
QUEUES[1].DISCIPLINE:=PS;
QUEUES[1].NUMBERUNITS:=10;
QUEUES[1].MEANSUBSERVICE:=3.0;
NODES[2].KINDOFNODE:=ALLOCATE;
ADDDESTINATION(2,3,1.0,FALSE);
QUEUES[2].NUMBERUNITS:=4;
ADDDESTINATION(3,4,0.5,FALSE);
ADDDESTINATION(3,5,0.5,FALSE);
QUEUES[3].DISCIPLINE:=PS;
QUEUES[3].MEANSUBSERVICE:=0.050;
ADDDESTINATION(4,3,0.9,FALSE);
ADDDESTINATION(4,6,0.1,FALSE);
QUEUES[4].MEANSUBSERVICE:=0.060;
ADDDESTINATION(5,3,0.9,FALSE);
ADDDESTINATION(5,6,0.1,FALSE);
QUEUES[5].MEANSUBSERVICE:=0.060;
NODES[6].KINDOFNODE:=RELEASE;
NODES[6].QUEUE:=2;
ADDDESTINATION(6,1,1.0,FALSE);
ADDREGEN(1,10);
```

Figure 7.46

approximate, the two sets of results are in close agreement. We can reasonably conclude that memory contention may be significant in this system.

### 7.4.2 Fission and Fusion Nodes

In some computer systems a job may consist of several concurrent processes which are simultaneously active with different resources, e.g., simultaneously performing computations and I/O transfers, simultaneously performing several I/O transfers, etc. In communication systems a message may be divided into several "packets" which are transmitted simultaneously, often on different communication lines. We extend our definition of queueing networks so that a job may go through a *fission* node to produce a second job associated with the first. We refer to the first job as the "parent" and the second job as a "child." It would be difficult to make the child identical to the parent because the parent may already possess tokens. So the asymmetry is intentional and not easily avoided. The passage of the parent through the fission node is instantaneous and the child departs immediately. We again take advantage of the node departure event in our implementation of procedure FISS (see Figure 7.48).

```
NUMBER OF EVENTS:    43582 SIMULATED TIME:   1256.610

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.505      1.678        5.054         3.052
    1       0.484      1.631        4.842         2.968
LOWER       0.463      1.583        4.631         2.884
UPPER       0.926      1.678        5.368         3.370
    2       0.910      1.631        5.157         3.161
LOWER       0.893      1.583        4.945         2.952
UPPER       0.836     16.773        2.051         0.123
    3       0.825     16.525        2.002         0.121
LOWER       0.814     16.276        1.953         0.119
UPPER       0.494      8.271        0.820         0.100
    4       0.484      8.140        0.792         0.097
LOWER       0.475      8.009        0.765         0.093
UPPER       0.516      8.538        0.876         0.103
    5       0.504      8.384        0.845         0.100
LOWER       0.492      8.230        0.815         0.098

NUMBER OF CYCLES:       31
AVERAGE NUMBER OF EVENTS:  1405.870
AVERAGE LENGTH:    40.535 C.I.:(    25.164,    55.907)
```

Figure 7.47

Our definitions of fission and fusion nodes allow a parent at most one child in existence at the same time and do not allow a child to have children. More general definitions are found in [SAUE76, SAUE77c], but the former is rather awkward and the latter is not as general as it could be.

A fusion node complements a fission node (or nodes) by providing a place for parents to wait for their children and vice versa. As soon as both a parent and its child are at a fusion node together, the child is destroyed and the parent leaves the node immediately. The variable FUSIONPTR is used to keep a list of waiting jobs. See Figure 7.49.

Our definition of fission nodes provides a separate list of possible destinations for the children. The last parameter of ADDDESTINATION is used to indicate whether the destination is to be added to the parent's list or the child's list.

To illustrate the use of fission and fusion nodes to represent CPU-I/O overlap, let us suppose that in our interactive computer system model that before 50% of the CPU services a job is able to produce another job which can do I/O while the creating job is still at the CPU. After either of these

```
PROCEDURE FISS(J: JOBPTR; F: INTEGER);
(*HANDLES ARRIVAL OF JOB J AT FISSION NODE F*)
  VAR TEMP: JOBPTR;
  BEGIN
    J↑.CURRENTNODE:=F;
    IF (J↑.CHILD<>NIL) OR (J↑.PARENT<>NIL) THEN
      BEGIN
        WRITELN('FISS - MULTIPLE GENERATIONS ATTEMPTED');
        HALT
      END
    ELSE
      BEGIN
        IF AVAILJOB=NIL THEN
          NEW(TEMP)
        ELSE
          BEGIN
            TEMP:=AVAILJOB;
            AVAILJOB:=AVAILJOB↑.NEXTJOB
          END;
        TEMP↑.CHILD:=NIL;
        J↑.CHILD:=TEMP;
        TEMP↑.PARENT:=J;
        TEMP↑.CURRENTNODE:=F;
        TEMP↑.TOKENHOLDER:=NIL;
        INSERTEVENT(NODEDEPARTURE,CLOCK,TEMP)
      END
  END;  (*FISS*)
```

Figure 7.48 - FISS

(hopefully) overlapped activities is finished, the process which is finished is forced to wait for the other to finish. Then another CPU-I/O cycle (possibly again with overlapped activities) begins for the job if the command is not finished. Figure 7.50 shows the addition of fission node 7, fusion node 11 and classes 8, 9 and 10 which belong to queues 3, 4 and 5, respectively.

Figure 7.51 shows the model specific statements for this network. Again the regeneration state has all of the jobs at the terminals. Figure 7.52 shows the output of the program with EVENTLIMIT=40000.

Notice that the CPU-I/O overlap has an apparently dramatic effect on response times in this system. We chose the parameters that this might happen, but it is not unlikely that a widely used program such as a text editor would be designed to achieve such overlap. We would not expect

```
PROCEDURE FUS(VAR J: JOBPTR; F: INTEGER);
(*HANDLES ARRIVAL OF JOB J AT FUSION NODE F*)
  VAR L, TEMP: JOBPTR; FOUND: BOOLEAN;
  BEGIN
    J↑.CURRENTNODE:=F;
    IF (J↑.PARENT<>NIL) OR (J↑.CHILD<>NIL) THEN
      IF (J↑.PARENT<>NIL) AND
        ((J↑.CHILD<>NIL) OR (J↑.TOKENHOLDER<>NIL)) THEN
        BEGIN
          WRITELN(
        'FUS - MULTIPLE GENERATIONS OR CHILD HOLDS TOKENS');
          HALT
        END
      ELSE
        WITH NODES[F] DO
          BEGIN
            FOUND:=FALSE;
            IF FUSIONPTR=NIL THEN
              BEGIN
                J↑.NEXTJOB:=NIL;
                FUSIONPTR:=J;
                LENGTHNODE:=1;
                J:=NIL
              END
            ELSE (*THERE ARE WAITING JOBS*)
              BEGIN
                IF J↑.PARENT=NIL THEN
                  IF J↑.CHILD=FUSIONPTR THEN
                    BEGIN
                      FOUND:=TRUE;
                      TEMP:=J↑.CHILD;
                      FUSIONPTR:=FUSIONPTR↑.NEXTJOB
                    END
                  ELSE  (*J↑.CHILD<>FUSIONPTR*)
                    BEGIN
                      L:=FUSIONPTR;
                      WHILE NOT FOUND AND (L↑.NEXTJOB<>NIL)
                        DO
                        IF L↑.NEXTJOB=J↑.CHILD THEN
                          FOUND:=TRUE
                        ELSE
                          L:=L↑.NEXTJOB;
                      IF FOUND THEN
                        BEGIN
                          TEMP:=J↑.CHILD;
                          L↑.NEXTJOB:=L↑.NEXTJOB↑.NEXTJOB
                        END
```

```
                END
              ELSE (*J↑.PARENT<>NIL*)
                IF J↑.PARENT=FUSIONPTR THEN
                  BEGIN
                    FOUND:=TRUE;
                    FUSIONPTR:=FUSIONPTR↑.NEXTJOB;
                    TEMP:=J;
                    J:=J↑.PARENT
                  END
                ELSE  (*J↑.PARENT<>FUSIONPTR*)
                  BEGIN
                    L:=FUSIONPTR;
                    WHILE NOT FOUND AND (L↑.NEXTJOB<>NIL)
                      DO
                      IF L↑.NEXTJOB=J↑.PARENT THEN
                        FOUND:=TRUE
                      ELSE
                        L:=L↑.NEXTJOB;
                    IF FOUND THEN
                      BEGIN
                        L↑.NEXTJOB:=L↑.NEXTJOB↑.NEXTJOB;
                        TEMP:=J;
                        J:=J↑.PARENT
                      END
                  END;
              IF FOUND THEN
                BEGIN
                  LENGTHNODE:=LENGTHNODE-1;
                  J↑.CHILD:=NIL;
                  TEMP↑.PARENT:=NIL;
                  TEMP↑.NEXTJOB:=AVAILJOB;
                  AVAILJOB:=TEMP
                END
              ELSE  (*NOT FOUND*)
                BEGIN
                  LENGTHNODE:=LENGTHNODE+1;
                  J↑.NEXTJOB:=FUSIONPTR;
                  FUSIONPTR:=J;
                  J:=NIL
                END
            END
        END
END;  (*FUS*)
```

Figure 7.49 - FUS

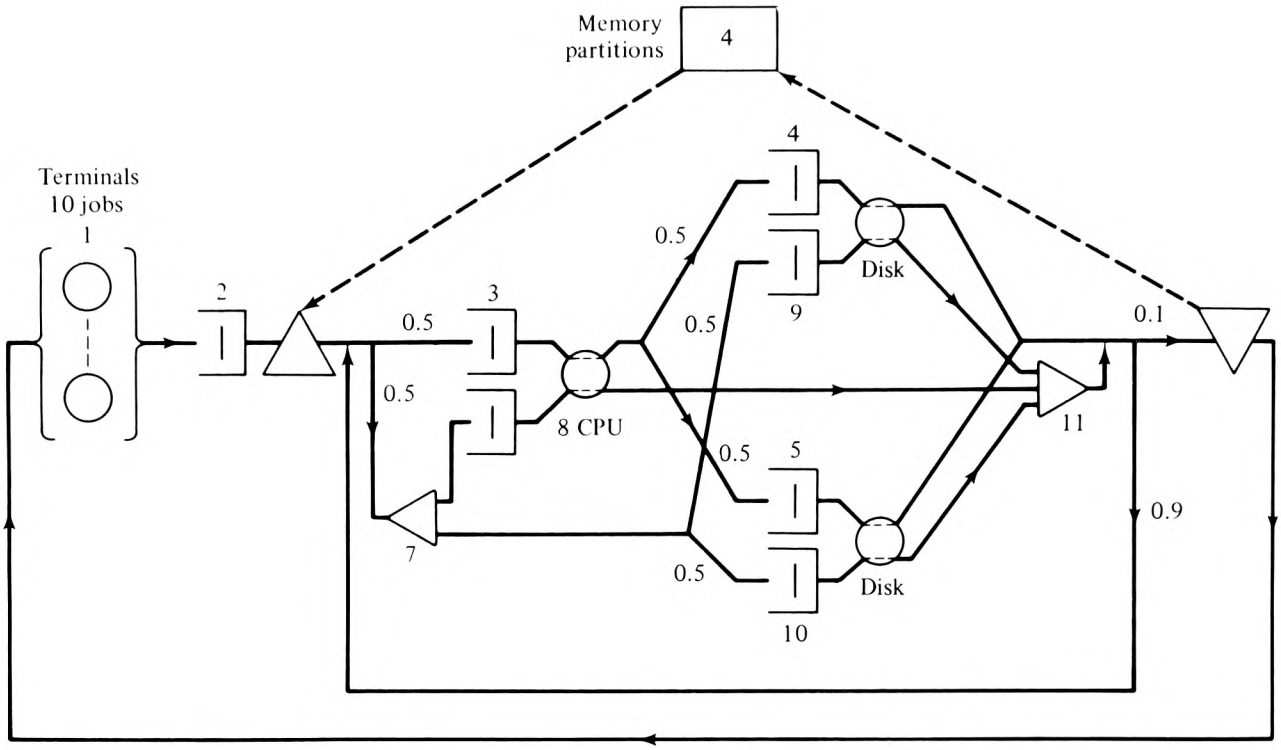



Figure 7.50

such design effort would be put into most programs, and measurements of general purpose computer systems usually show a small degree of CPU-I/O overlap.

Also notice that the mean response time (queue 2 queueing time) estimate is near to the response time estimate of Section 7.3, 2.81, which was based on a much simpler model. The increase in mean response time due to memory contention and the decrease in mean response time due to CPU-I/O overlap roughly negate each other to make the estimates of the relatively unrealistic model fairly accurate. This is one explanation of the success of some queueing network models which appear to be unacceptably simplistic: the models capture the contention for the principal resources while the effects of secondary resources and other characteristics are small *when considered together*. If we have the time and money to investigate the effects of secondary resources and characteristics, and if we need a high degree of accuracy in our estimates, then we should pursue these effects. On the other hand, if we are limited in human and machine resources and can tolerate a fair amount of error, then we may well get by on our intuition and simplistic models.

```
(*PARAMETERS SPECIFIC TO THIS MODEL*)
ADDDESTINATION(1,2,1.0,FALSE);
QUEUES[1].DISCIPLINE:=PS;
QUEUES[1].NUMBERUNITS:=10;
QUEUES[1].MEANSUBSERVICE:=3.0;
NODES[2].KINDOFNODE:=ALLOCATE;
ADDDESTINATION(2,3,0.5,FALSE);
ADDDESTINATION(2,7,0.5,FALSE);
QUEUES[2].NUMBERUNITS:=4;
ADDDESTINATION(3,4,0.5,FALSE);
ADDDESTINATION(3,5,0.5,FALSE);
QUEUES[3].DISCIPLINE:=PS;
QUEUES[3].MEANSUBSERVICE:=0.050;
ADDDESTINATION(4,3,0.45,FALSE);
ADDDESTINATION(4,6,0.1,FALSE);
ADDDESTINATION(4,7,0.45,FALSE);
QUEUES[4].MEANSUBSERVICE:=0.060;
ADDDESTINATION(5,3,0.45,FALSE);
ADDDESTINATION(5,6,0.1,FALSE);
ADDDESTINATION(5,7,0.45,FALSE);
QUEUES[5].MEANSUBSERVICE:=0.060;
NODES[6].KINDOFNODE:=RELEASE;
NODES[6].QUEUE:=2;
ADDDESTINATION(6,1,1.0,FALSE);
NODES[7].KINDOFNODE:=FISSION;
ADDDESTINATION(7,8,1.0,FALSE);
ADDDESTINATION(7,9,0.5,TRUE);
ADDDESTINATION(7,10,0.5,TRUE);
NODES[8].QUEUE:=3;
ADDDESTINATION(8,11,1.0,FALSE);
NODES[9].QUEUE:=4;
ADDDESTINATION(9,11,1.0,FALSE);
NODES[10].QUEUE:=5;
ADDDESTINATION(10,11,1.0,FALSE);
NODES[11].KINDOFNODE:=FUSION;
ADDDESTINATION(11,3,0.45,FALSE);
ADDDESTINATIION(11,6,0.1,FALSE);
ADDDESTINATION(11,7,0.45,FALSE);
ADDREGEN(1,10);
```

Figure 7.51

```
NUMBER OF EVENTS:    41073 SIMULATED TIME:   1133.542

QUEUE UTILIZATION THROUGHPUT QUEUE LENGTH QUEUEING TIME
UPPER       0.533      1.739        5.328         3.148
    1       0.515      1.684        5.147         3.056
LOWER       0.497      1.629        4.965         2.964
UPPER       0.904      1.739        5.035         3.063
    2       0.885      1.684        4.853         2.882
LOWER       0.866      1.629        4.672         2.701
UPPER       0.881     17.524        2.369         0.137
    3       0.870     17.275        2.303         0.133
LOWER       0.859     17.026        2.237         0.130
UPPER       0.522      8.790        0.892         0.102
    4       0.509      8.630        0.855         0.099
LOWER       0.496      8.471        0.819         0.096
UPPER       0.517      8.828        0.873         0.100
    5       0.505      8.645        0.840         0.097
LOWER       0.493      8.461        0.807         0.095

NUMBER OF CYCLES:       50
AVERAGE NUMBER OF EVENTS:    821.460
AVERAGE LENGTH:     22.671 C.I.:(     15.707,     29.635)
```

Figure 7.52

## 7.5 RESPONSE TIME DISTRIBUTIONS

One apparently intractable problem with almost all interesting queueing network models is to numerically obtain response time distributions. Exact solutions have been found for some simple central server models [CHOW78] and restricted open networks [TAKA63, WONG78a] and approximate solutions have been proposed for other networks [YU77].

Yet we are often very interested in response time distributions. Users of a computer system are likely to prefer a distribution with a larger mean if they are compensated by less variability in the response times, i.e., if they are better able to predict the response they will get. Vendors are often asked to make statements about system response times such as "95% of the response times will be less than 5 seconds".

A common heuristic used with respect to this last situation is to assume the response time distribution has a convenient form, e.g., exponential or Erlang, and make a statement based on the mean [MART67]. For example, if the mean is 3 seconds and we assume an exponential distribution then we can say that $a \times 100\%$ of the response times will be less than

$-3\ln(1 - a)$, e.g., 95% of the response times will be less than 8.99 seconds. The exponential assumption is correct for a FCFS single server queue with exponential interarrival times and exponential service times, i.e., the M/M/1 queue. The Erlang assumption is thus correct for some open networks. One of Chow's results was that certain limiting cases of central server model cycle times have an Erlang distribution [CHOW78].

With simulation, response time distribution estimation is no more difficult, in principle, than estimation of the measures we have been considering. We have seen in the last two simulations that the queueing times for the passive queue were also the computer system response times. This is no coincidence, but a natural consequence of the meaning of the resource and our definition of queueing time for passive queues. We can use passive queues to measure response times in arbitrary subnetworks, whether or not there is a corresponding actual (physical or logical) resource. For example, suppose our interactive computer system of the previous simulations has an abundance of memory and no memory contention. We can still use the passive queue for the response time purpose by providing an "infinite" number of tokens. (For that system 10 tokens would be sufficient.)

Thus estimating response times is reduced to estimating queueing times. To estimate queueing times other than the mean we must record them in some way. We record the arrival time when a job arrives at a queue so that we can determine the time the job spent in the queue when it leaves. Then if we want to estimate the variance or higher moments we can apply standard formulas based on these individual queueing times. (For the variance this would simply be (7.4).) If we want to estimate the fraction of queueing times less than 5 seconds, we need only count the number of times less than 5 seconds and divide by the total number. We can do this for several values of interest, or if we want an estimate of the entire distribution, we can do this for selected values throughout the range of possible queueing times. (If we want the actual distribution of the individual queueing times, then we should maintain a sorted list of the observed times and calculate the cumulative frequency. This will be computationally expensive, but conceptually simple.)

Estimating confidence intervals with respect to the queueing time distribution may or may not be more difficult depending on the measure of interest and the confidence interval method. The entire cumulative frequency will cause difficulty regardless of confidence interval method, as will measures based on it. For most other measures we can apply independent replications in a straightforward manner. (Some measures will be much more variable than the mean values we have estimated — thus much more computational effort will be needed for narrow intervals to be obtained.)

With the regenerative method some measures, in particular the variance, require more complex estimators, but other measures, for example the fraction of times less than a specified value, use exactly the same methods we applied to the measures in the previous sections. However, there is one very important consideration in regards to choice of regeneration state. That consideration is that we cannot assume the queueing time processes during different regeneration cycles are independent and identically distributed if there are queueing times of interest in progress in that state. (There is no problem with respect to the mean queueing time since we use a "Little's Rule" approach to its estimation.) The Markov state which is a regeneration state for the population process, i.e., the queue lengths, is not necessarily a regeneration state for the queueing time process. However, if the queue is empty in the regeneration state, then no queueing times can be in progress and we have a regeneration state for the queueing time process. This is part of the basis for our choice of the state with all jobs at the terminals in our interactive system model; that state is a regeneration state for the response time process for the computer system. This consideration is another argument in favor of the empty system state in simulation of open networks.

## 7.6 FURTHER READING

A more thorough and general treatment of simulation can be found in [FISH73,FISH78]. A general discussion of random number generation, including testing of uniform generators and methods for obtaining non-uniform random variables, is given by [KNUT68]. A comparison of several popular generators is provided by [LEAR73].

Discussion of more efficient event list mechanisms can be found in [FRAN77] and [MEAR79].

An excellent introduction to the regenerative method is given in [CRAN77]. Some more advanced material is presented in [IGLE78a] and [FISH78]. Some solutions to the "queueing times in progress" problem are proposed by [IGLE78b] and [FISH79]. A heuristic approach for systems with infrequent regeneration states is studied empirically in [SAUE77a].

We have entirely ignored programming languages specifically designed for simulation. Such languages provide random number generators, event list mechanisms, and other features such as specialized list processing facilities and statistics gathering. An introduction to such languages is found in the Fishman texts. Discussion of specialized queueing languages is found in [SAUE78a].

More examples of extended queueing network models of computer and communication systems, including use of the regenerative method, are given in [SAUE77c,SAUE78b].

## 7.7 EXERCISES

7.1 Modify the data structures, ARRIVE and COMPLETE to allow the branching Erlang distribution for service times. Provide a procedure that allows specification of the standardized forms in Chapter 3 by giving the mean and coefficient of variation.

7.2 We have omitted the procedures FREEJOBS, FREEROUTING and FREEREGEN which are used in Figure 7.32. Provide definitions of each of these.

7.3 With our definitions of INSERTEVENT and REMOVEEVENT we would expect that we would have to search half of the list to insert an event or to remove an event other than the first or the last. Provide a third pointer, in addition to FIRSTEVENT and LASTEVENT, so that we would only expect to search one fourth of the list. Can you generalize this for further efficiency? (At what point do the additional pointers become a burden?)

7.4 Revise the data structures, ARRIVE and COMPLETE to allow class dependent service times.

7.5 Revise the data structures, ARRIVE and COMPLETE to avoid the inefficiency of scheduling completion events for each stage for jobs at classes which are empty in the regeneration state.

7.6 The simulation program as presented allows us to specify run length in terms of numbers of events. If the confidence intervals are "too wide," we must run the simulation again with a larger number of events. Provide the following rule as an alternative: We wish to stop the simulation every $k$ cycles and determine whether, for a specified queue's mean queueing time, and relative width of the confidence interval, i.e., $2d/y_n$, is less than $g$. If so we terminate the run, otherwise we continue for another k cycles. We always use the data from all simulated cycles in our estimates. (Formal and empirical justification for this rule is given in [LAVE77].)

7.7 Implement a non-preemptive priority scheduling algorithm for active queues, as described in Chapter 3. Each class is to be assigned a priority for scheduling purposes.

7.8 Repeat 7.7 with preemptive priority.

7.9 Implement a Round Robin scheduling algorithm for active queues.

7.10 Implement the Shortest Remaining Time First scheduling algorithm for active queues.

7.11 Implement composite queues (Chapter 6) assuming each class belongs to a different chain.

7.12 Revise the data structures, ALLOC and RELEAS so that the number of tokens a job requests may random with a finite discrete distribution, i.e., with probability $p_1$ a job requests $t_1$ tokens, with probability $p_2$ a job requests $t_2$ tokens, etc.

7.13 Provide new nodes for passive queues: A *destroy* node which throws away a job's tokens rather than returning them to the queue, and a *create* node which adds to the total number of tokens at a queue. The number of tokens added has a finite discrete distribution. The create node affects only the queue; it has no direct effect on the job going through it.

7.14 Allow regeneration states which have jobs at allocate nodes. Be sure that you do not allow states which are not a regeneration state to be counted as such. (You may want to separate initialization of jobs from the other functions of ADDREGEN.)

7.15 Provide performance estimates for individual nodes as well as for queues.

7.16 Provide estimates of the number of queueing times less than or equal to specified values. Be careful about queueing times in progress with respect to confidence intervals.)

7.17 Provide sources and sinks with the new data structures. Provide for efficient determination of regeneration state by partitioning nodes into chains.

7.18 Implement *split* nodes which are similar to fission nodes except that there is no future association between the created job and the creating job.

7.19 In addition to routing based on probabilities, provide routing based on *predicates*, e.g., a job may select a node to go to based on queue lengths, number of tokens available, etc.

7.20 Allow jobs to carry data with them from node to node to be used (if desired) in routing decisions and service time calculations. Provide nodes to define and alter this data.

# *CHAPTER 8*

# MEASUREMENT AND PARAMETER EVALUATION

In the preceding chapters we have focused our attention on solution of models and on system characteristics we consider most likely to significantly impact performance. In doing so, we have assumed that numerical parameters are given to use or are readily available. Now we turn our attention to measuring and estimating model parameters.

Obtaining model parameters usually requires a combination of approaches, with the particular approaches and combinations strongly dependent on the particular system and its current evolution stage. We will consider the most important approaches and combination strongly dependent on the particular system and its current evolution stage. We will consider the most important approaches and try to discuss them in as general a context as possible; as a result, we will omit some details relevant (and required) in particular situations. These details will usually require system specific knowledge and/or specific knowledge of measurement tools used.

We assume that we are formulating a model of modifications to an existing system in order to conveniently cover the entire range of possibilities suggested in Chapter 1. In the limiting case of an entirely new system, the "modifications" are actually the entire system, while in the limiting case of modeling an existing system, there are no modifications. Thus we need to characterize the parameters of the existing system and we need to characterize the parameters of the modifications or new system. We will consider the existing system parameters first.

## 8.1 MEASUREMENT AND RELATED METHODS FOR EXISTING SYSTEM PARAMETERS

There are three main sources of measured data: existing *accounting software, hardware monitors* attached to the system, and *software monitors*, i.e., software added to the system specifically for measurement. Measurements are usually obtained principally by a hardware or software monitor supplemented by accounting software and hardware specifications. Depending on particular situations and parameters, we may be able to get the parameter values directly or we may have to obtain the parameter values from intermediate values which we can obtain directly. This latter case is

very similar to methods used in simulation to obtain performance measures, for example, the "Little's Rule" approach to mean queueing times (Section 7.1.3).

### 8.1.1 Accounting Software and Hardware Specifications

Most computing systems of significant size include software to determine users' resource usage so that they may be billed for their activities (or at least discouraged from wasting resources in an environment without charges for system use). Though nearly every different site will have different accounting policies, and correspondingly different accounting software, there is information such as CPU time per user which will almost always be gathered. Other readily available information may include the number of I/O accesses (perhaps by device), memory required, number of page faults, connect time for interactive users, turnaround time for batch jobs and active time for batch jobs. (More information is likely to be available than is actually used for accounting purposes.) In addition to such user specific data, general information such as the average degree of multiprogramming may be obtainable from the accounting data.

A principal problem with using accounting data for model parameters is that accounting data often excludes resource usage by the operating system not directly invoked by user programs. (Since such resource usage cannot be attributed to individual users, it is usually not directly charged to users.)

If the operating system resource usage not included in the accounting data is negligible, then accounting data and intuitive use of system and hardware specifications may be sufficient to determine parameters for simple models (for existing system portions). For example, if we assume negligible overlap of CPU and I/O activity by individual programs, then mean CPU service time can be estimated by total CPU time divided by the number of I/O accesses. (This can be done for an individual user, a grouping of users, operating system components, users and operating system components together, etc.) If we are willing to make assumptions about cylinder access patterns for the disks, then we can easily estimate seek times from the hardware specifications. (See WILH76 for detailed discussion of such calculations and the effects of various assumptions.) We can reasonably assume that the latency will be uniformly distributed between zero and one revolution. If we know the buffer size and assume that most transfers consist of a full buffer, then the mean transfer time can be estimated by the buffer size divided by the transfer rate. We may estimate the mean disk service time then as the sum of the mean seek, latency and transfer times. (With position sensing devices and several devices per channel and/or controller, we may need to consider these disk service time components individually. We will pursue this further in Section 9.3.) Similar approaches

may be used for drums, tapes and other I/O devices. Thus we have the essential parameters (mean degree of multiprogramming, mean CPU service time and mean I/O time) for the cyclic queue model used in CHIU75 and SAUE77b. If we need to consider several I/O queues then we can obtain the branching probabilities from the relative frequency of access to the devices of each queue.

### 8.1.2 Hardware Monitors

Since accounting data and hardware specifications are usually not sufficient for supplying model parameters, it is usually necessary to use a hardware or software monitor to supplement these other sources. A hardware monitor is simply a collection of digital circuitry which is attached to the hardware of the computer system. Existing hardware monitors range from devices consisting of a few circuits to complete computer systems including disks and other peripherals. Hardware monitors are widely used to estimate system performance directly (without modeling); there are commercially available hardware monitors for most significant computer systems. Though hardware monitors may be quite complex, they are fairly simple from our point of view.

We are primarily interested in the *probes* and *accumulators* which may be associated with the probes and the *probe points* in the system where we attach the probes. The probe points are places where we can measure voltage levels corresponding to system states such as CPU busy (or idle), channel busy, (a particular) memory location being referenced, etc. By attaching the probes to these points we can observe system behavior and use the counters to record the number of changes in system state during an observation period and the accumulators to record the amount of time spent in interesting system states. This measurement process will usually be *transparent* to the observed system, i.e., there will be no difference in system performance with or without the monitor attached. This is one of the principal advantages of hardware monitors over software monitors.

A principal difficulty in using hardware monitors is knowing where to place the probes, e.g., knowing where is the probe point which corresponds to the CPU being busy. Fortunately, the system manufacturers and the monitor manufacturers have developed libraries of probe points for many common architectures.

Hardware monitors are principally limited by the available probe points and our ability to interpret the available data in terms of the interesting system parameters. For example, suppose we wish to estimate CPU service times. Further, there is a probe point which is "on" whenever the CPU is switched to another process. Thus we can determine the number of times

this probe point changes state, and we can obtain the CPU busy time by accumulating the time this point is on. The CPU busy time will be the sum of the service times (regardless of scheduling algorithm, if we ignore switching overhead) and we can obtain the mean service time by dividing CPU busy time by the number of CPU *services*. However, if the CPU scheduling is preemptive, e.g., Round Robin, then the count from this probe point will include preemptions as well as the number of services. In general, it is unlikely that there will be a probe point which can give us a count of the number of services and we must obtain the number of services from some other source. Such a source would be the accounting data, as described in the previous section. In estimating mean CPU times, hardware monitors and accounting data complement each other well; the hardware monitor can give us the sum of the CPU times *including* operating system activities and the accounting data can give us the number of CPU services, *excluding* the preemptions.

Hardware monitors can provide very accurate estimates for other parameters of importance, such as mean seek times (without making assumptions about cylinder access patterns). However, hardware monitors are inherently limited to measuring information we can interpret at the hardware level without knowledge of operating system activities. This means that we cannot easily obtain CPU service times by process, for example, and implies that we cannot obtain distribution estimates other than the mean for parameters such as CPU service times. For these reasons, hardware monitors by themselves will usually not be sufficient for our purposes, but hardware monitors supplemented by accounting data, hardware specifications and educated guesses (e.g., "The CPU service time distribution form doesn't matter since scheduling is similar to Processor Sharing.") may well be sufficient.

### 8.1.3 Software Monitors

A software monitor is a collection of pieces of code embedded in the operating system to gather performance data. (Depending on the architecture and operating system, some of the code may be run at user level processes.) There are two major approaches to the design of such a monitor, the *event* approach and the *sampling* approach. Any software monitor will perturb system performance somewhat, since it requires system resources to execute. The event approach has the advantages of maximum flexibility and generality, but it is likely to perturb performance more than a sampling monitor. In addition to providing more control over the overhead introduced by the monitor, in the sampling approach it may be more convenient to add a monitor to an existing operating system.

In the event approach, the designer or user of the monitor must define significant events, e.g., CPU is switched from one process to another, I/O request issued, etc. The operating system modules which effect such events must be modified so that when the events occur, the module records (e.g., writes to a tape file) the type of event, the time it occurred, and any other important data associated with the event. The event records are processed after the measurement period to obtain the desired parameters. (Though data reduction might be done during the measurement period, this usually is avoided because of the additional perturbation of system performance.) This approach allows us to obtain very detailed information. For example, from the event trace we can observe the duration of each individual CPU service time (including preemptions and resumptions) and accumulate estimates of the distribution form as well as the mean. The principal limitation on detail is that of being able to observe and record the appropriate events.

With the sampling approach, monitor code is enabled periodically to determine whether the CPU is busy, what a particular device is doing, etc. From this information we can directly estimate performance metrics such as CPU utilization, and thus indirectly estimate total CPU service time. The sampling approach is much less general and flexible than the event approach, e.g., there are parameters such as variances of service times which cannot be feasibly estimated with the sampling approach. The advantages of the sampling approach are potentially simpler implementation and the ability to directly reduce system perturbation by reducing sampling frequency. However, if we reduce sampling frequency we must compensate for the loss of data by lengthening the measurement period. We prefer the event approach because of its generality and flexibility.

Depending on the particular system and monitor, a software monitor may consume 20% or more of the system resources (particularly CPU and channel time) and thus produce very questionable results. By appropriate definition of events and implementation, this overhead may be kept to roughly 5%, and the software monitor results should be sufficiently accurate for our purposes.

Besides the inherent perturbation of system performance, there are two other significant problems with software monitors. First, the amount of data produced by the event trace may be overwhelming, particularly in terms of reducing the trace data after the measurement period. (It is not unusual for the measurement period to be limited to a fairly short period of time, say twenty minutes, by the capacity of a reel of tape [SHER72b].) Second, unlike hardware monitors, software monitors must be specifically designed for particular architectures and operating systems. Thus there are no commercially available software monitors for many significant computer systems. Further, implementing a software monitor for an operat-

ing system that already exists may require a significant amount of effort and expertise.

Before leaving measurement we should point out that any of the above sources of parameters may also be very valuable in providing data for *validation* of our models. For example, we can estimate the CPU utilization by the sum of CPU times divided by the length of the measurement period. If our model estimate of CPU utilization agrees well with our measurement estimate for the system without modifications, then we can place more faith in the model estimates for the modified system.

## 8.2 PARAMETERS FOR SYSTEM MODIFICATIONS

As we said in Chapter 1, we cannot measure a system unless it is operational. In particular, if we are adding a new subsystem to an existing system or building a new system entirely, then we cannot measure it during the design and development stages of its implementation. To obtain numerical parameters for our models we must use our knowledge of the existing system and the planned modifications to produce estimates of the numerical parameters. Early in the evolution process it will be difficult to produce accurate estimates, but since we are only interested in rejecting poor designs at that time, we can make our estimates intentionally pessimistic. As long as this is done in a reasonable manner, we should not incorrectly reject good designs but we may unnecessarily reject a marginal design. This should not be of concern as long as we have better designs left.

For example, let us consider the resource demands of module X. We discuss X with its designer and are told that a call to X will "probably" result in the execution of 400,000 instructions but the designer is confident that no more than 1,000,000 instructions will be executed. The designer's estimate of the working set of X is 5 pages, the estimated number of page faults (including initial loading) is 7, and the estimated number of I/O operations is 3. We also know that the CPU executes instructions at a rate of 1.5 MIPS (million instructions per second), so we can estimate the total CPU time of module X as .667 seconds. Similarly, from our discussion with the designer we come up with estimates of which files will be accessed and the amount of data transferred. From this information we can determine which devices will be involved and estimate I/O times for module X. If X makes calls on the operating system which will cause significant resource demands, then these demands should also be included as part of the demands of X. We proceed in this manner for all of the modules of the system. If the number of modules is small, then we may be able to include this information directly in our model, using a separate class for each module and reflecting the module execution order by the class transitions. If the number of modules is large then we must first aggregate modules and then

represent the aggregate modules as classes. (Though we could aggregate all of the modules and avoid class distinctions, class dependent performance estimates can be used to determine which modules are likely to be bottlenecks, and thus are candidates for redesign.)

As the system evolves, we will be able to get more accurate, less pessimistic estimates of resource demands and thus get more accurate performance predictions. When the system is operational, we can use measurements in place of some of our resource demand estimates, and we can use measurements of performance metrics to validate our model or suggest improvements.

## 8.3 FURTHER READING

We have given a very superficial treatment of measurement and parameter estimation, partly because there are so many books devoted to these topics, particularly measurement. See DRUM73 and FERR78, for example.

A survey of measurement tools and practices for many popular computing systems is given in ROSE78. That article discusses these topics from a queueing network point of view.

For a more detailed discussion of parameter estimation during the design process and a specific example, see SMIT79.

# *CHAPTER 9*

# CASE STUDIES

We now discuss in detail the six modeling examples of Chapter 1.

## 9.1 A SIMPLE BATCH SYSTEM MODEL

### 9.1.1 The Modeled System

The hardware studied by Chiu *et al* consists of an IBM 360/75 with 512 kilobytes of high speed core memory, two megabytes of slower core memory, two selector channels, each with 8 2314 disk drives, and a multiplexor channel controlling printers, tape drives and other peripherals. During the modeling project, the slower core memory was changed from one with an eight microsecond cycle time to one with a 1.8 microsecond cycle time (each with the same two megabyte capacity).

The operating system is the standard IBM OS/MVT with HASP, modified to support a locally implemented time sharing system. The time sharing uses a small amount of the CPU capacity (approximately 8% with the 1.8 microsecond slow core) and very little of the remaining resources. Thus the focus of the modeling effort is the batch workload, but the effect of the time sharing system is taken into consideration.

### 9.1.2 The Model

The principal model used is the cyclic queue model we have discussed frequently. In addition, a similar *central server model* [BUZE71], is used for comparative purposes. See Figure 9.1. The difference between the between the structures of the cycle queue model and the central server model is that the central server model has more than one queue for the I/O devices.

In using this model, it is assumed that there is a sufficient backlog of jobs and there is sufficient memory contention that the degree of multiprogramming is essentially constant. This is not strictly true, and so the average degree of multiprogramming is usually not an integer. In using the model, one interpolates between the two integer degrees of multiprogramming containing the average. Chiu *et al* obtained the average degree of multiprogramming from accounting software, the standard IBM SMF (System Management Facility) package.

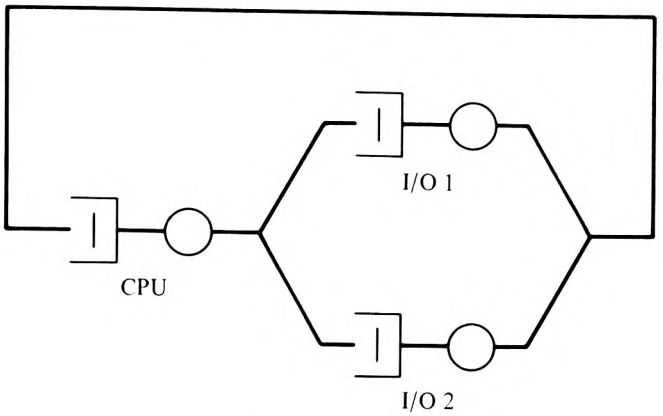



Figure 9.1

The CPU service time distribution was measured (using unstated methods, presumably a special purpose software monitor) and observed to have a coefficient of variation greater than one. Figure 9.2 shows the measured service time distribution and a hyperexponential distribution which could have been used to fit that distribution. However, an exponential distribution is used in the model. This is principally justified by the effects of the heuristic CPU scheduling algorithm which attempts to approximate SRTF. Figure 9.3 shows the *effective* CPU service time distribution in the sense that the CPU times between I/O requests have this distribution (This effective distribution has lowercoefficient of variation because long CPU bursts are broken into smaller ones by the scheduler. One time between I/O requests may be attributed to several jobs service.) Figure 9.3 also shows this effective distribution fitted by an exponential distribution. (Note that the distributions in these figures are scaled to have mean one.) The authors concluded from this and other evidence that they could treat the CPU as if it had FCFS scheduling and an exponential distribution. This consistent with our previous observations that some scheduling disciplines, such as PS and LCFSPR, give the same performance measures with essentially arbitrary distributions as FCFS does with exponential distributions.

In modeling the I/O system, several observations are used to simplify the model. First, all I/O to or from the slow speed peripherals is negligible. Second, there is minimal use of the tape drives. So the tape drives and slow peripherals are ignored in the model. Finally, though it is feasible for more than one disk drive on a channel to be active, this is rare, so in the model each channel and its disk drives are treated as a single I/O device, i.e., a single high capacity disk.

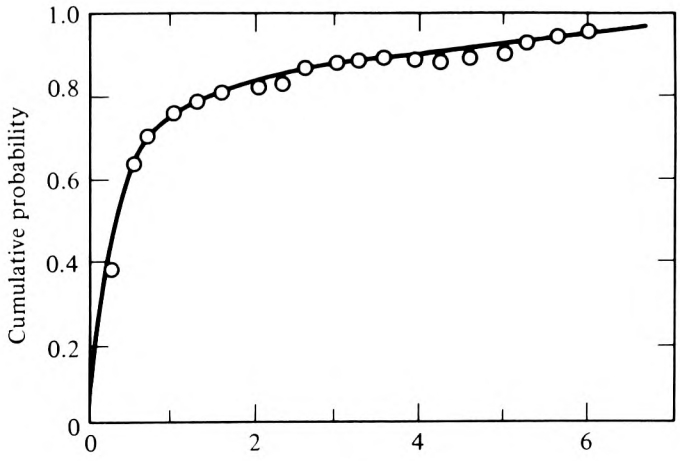



Figure 9.2

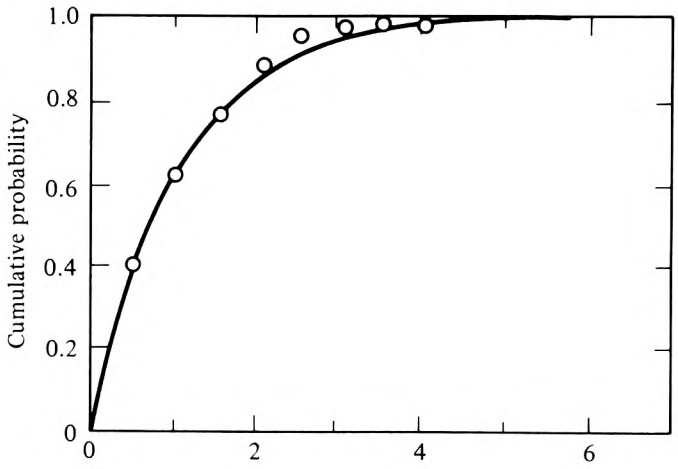



Figure 9.3

With the 2314 disk it is possible to perform the seek operation without the channel attached, once the seek has been initiated by the channel, but the channel must be attached for rotational positioning and transfer operations. (If more than one disk per channel were simultaneously active, then the model would have to consider possible overlap of seek with other operations. We will discuss this problem in Section 9.3.) The authors measured the channel busy time distribution (where channel busy time consists of rotational positioning and transfer) but were unable to measure the distributions of seek time or total service time. The busy time distribution has a coefficient of variation of one, like the exponential distribution, but does not closely fit the exponential distribution function. It is quite reasonable to assume the seek time has a uniform distribution [TEOR72]. Based

on results for sums of random variables, one can easily conclude that the coefficient of variation of the total I/O service time is significantly less than one. The authors assumed an exponential distribution for I/O times. This is not unreasonable since the effect of distribution form is relatively small when the coefficient of variation is less than one. (Refer back to Figure 4.5.) Also, the parameters used in this work were such that there was little queueing for I/O; we know from Chapters 4 and 5 that distribution form does not have an effect when there is no queueing. The authors assumed FCFS scheduling at the I/O queue (queues in the central server model).

The mean CPU service time is obtained as the CPU busy time (obtained from a hardware monitor) divided by the number of disk transfers. The mean I/O service time is obtained as the sum of the selector channel busy times and the total seek time (obtained by a hardware monitor) divided by the number of disk transfers.

### 9.1.3 Experiments with the System and Model

The authors conducted a number of experiments comparing model results with measurements, using a controlled workload for some experiments and eight "live" measurement sessions. We discuss the controlled workload experiments first.

The controlled workload consisted of fifty jobs selected as representative of the daily submissions. Since one of the objectives of these experiments was a reproducible environment, the time sharing system was disabled. The experiments were used to verify the ability of the model to predict performance while varying the slow core used (8 microsecond vs. 1.8 microsecond) and the number of initiators. (In the MVT operating system an "initiator" was required for each active job. Thus the number of initiators enforces an upper bound on the degree of multiprogramming.)

In the first experiment, with the 8 microsecond slow core and five initiators, the estimated degree of multiprogramming from SMF was 2.2. (The low degree of multiprogramming was due to memory contention.) From the hardware monitor, the CPU busy time was 1746 seconds (out of an elapsed time of 2580 seconds), one selector channel was busy 691 seconds, the selector channel was busy 928 seconds, the total seek time was 1246 seconds and the number of transfers was 60802. Thus the mean CPU time was 28.7 ms. and the mean I/O time was 46.8 ms. Using $N = 2$, the state probabilities are then .353, .412 and .336 for 2, 1 and 0 jobs at the CPU, respectively. Thus the CPU utilization is 66.5% and the I/O utilization is 54.2%. With $N = 3$, the state probabilities are .198, .323, .264 and .215 for 3, 2, 1 and 0 jobs at the CPU, respectively, the CPU utilization is 78.5% and the I/O utilization is 64.1%. Interpolating for $N = 2.2$, we get

the CPU utilization as 68.9% and the I/O utilization as 56.2%. From the measurements above, the actual CPU utilization was 67.7% and the actual I/O utilization was 55.1%. The model estimate of CPU throughput is .689/28.7 = .024 jobs per ms. and of system CPU throughput is .689/28.7 = .024 jobs per ms. and of system throughput is 24/(60802/50) = .0197 jobs per second. The actual system throughput was 50/2580 = .0194 jobs per second. Thus the model estimates for both utilization and throughput are very close to the measured values.

Repeating the experiment with the 1.8 microsecond core, the estimated degree of multiprogramming was again 2.2. The CPU busy time was 1225 seconds out of 2160 elapsed seconds, the channel one busy time was 834 seconds, the channel two busy time 760 seconds, and the seek time and number of transfers were the same as before. Thus the mean CPU time was 20.1 ms. and the mean I/O time was 46.7 ms. With $N = 2$ the model estimated CPU utilization is 55.2% and the model I/O utilization is 64.1%. With $N = 3$ the corresponding values are 65.8% and 76.4%. Interpolation gives CPU utilization 57.3% and I/O utilization 66.6%. The throughput estimate is .0234 jobs per second. The corresponding values from measurements were 56.7%, 65.7% and .0231 jobs per second; the agreement is even better.

| Initiators | $N$ | Measurement | CQM | CSM |
|---|---|---|---|---|
| 1 | 1.00 | 60% | 56% | 56% |
| 2 | 1.93 | 78% | 80% | 74% |
| 3 | 2.82 | 85% | 90% | 84% |
| 4 | 3.14 | 91% | 91% | 86% |
| 5 | 3.99 | 92% | 95% | 89% |

Figure 9.4

The remaining controlled workload experiments we consider were intended to study the effect of varying the degree of multiprogramming, principally by varying the number of initiators from one to five. Two others system changes were made: the jobs were executed in the larger, slower core (1.8 microsecond) to reduce the effect of memory contention, and memory scheduling was FCFS rather than the HASP algorithm. The estimated degrees of multiprogramming were 1, 1.93, 2.82, 3.14 and 3.99 for 1, 2, ..., 5 initiators, respectively. The elapsed times were 3445, 2662, 2435, 2278 and 2258, respectively, the I/O times (total) were 1611, 1890, 1978, 2123 and 2243, respectively, and the numbers of transfers were 45050, 47440, 47528, 47749 and 47665, respectively. Figure 9.4 shows the CPU utilizations from measurements, from the cyclic queue model (CQM) and from the central server model (CSM). (For the central service model the

branching probabilities to the first I/O queue are .556, .526, .544, .528 and .521, respectively.) Except for the one initiator case, the cyclic model seems to slightly overestimate the utilization and the central server model slightly underestimates the utilization.

The authors made measurements during 8 periods of the system running in its normal production environment, 2 with the 8 microsecond slow core and 6 with the 1.8 microsecond core. The authors were unable to measure the total seek time because of too few probes in the hardware monitor (the controlled experiments were run through the system twice to overcome this) and used the mean seek time from the first controlled experiment, 20.5 ms., for the model. Apparently for sake of convenience, the authors also used 2.2 as the degree of multiprogramming. For the session with the highest CPU utilization, 80%, the mean CPU time (including the time sharing load as overhead) was 49.7 ms. and the mean I/O time was 49.9 ms. The cyclic model estimate of CPU utilization is then 82.1%.

Note that all of the above models can be represented by Markov processes with at most fifteen states. Further, the models have product form solutions, so *the results can be trivially obtained with a hand calculator,* if necessary.

## 9.2 AN EVALUATION OF MULTIPROCESSOR SYSTEMS

Multiple CPU systems have been available for some time, but have only recently achieved popularity. Bell and Newell [BELL71] suggested that the range of performance spanned by the IBM 360 family of single CPU systems could also be spanned by a smaller product line and the use of multiple CPU systems may be cost-effective in comparison with single CPU systems; for a detailed discussion see Bell and Newell [BELL71] and Fuller [FULL76].

The economics of large scale integrated circuits has made multi-miniprocessing units an alternative to systems with single large central processing units. The decreasing costs of hardware relative to software and the growing importance of security and reliability may tend to result in simpler software and relatively underutilized or redundant hardware. This trend could result in multi-miniprocessing systems with simplified system scheduling strategies. In SAUE77b we compared performance metrics for different architectures for a variety of scheduling strategies and work loads. An important objective was to study the impact of CPU service service distributions and disciplines on the behavior of multiprocessing systems. The primary goal of that work was to use modeling techniques to make quantitative analyses of possible trends in architecture and their impact on operating

systems. The multiple processor systems considered were essentially restricted to those with tightly coupled homogeneous CPU's, for example C.mmp [WULF72], where the CPU's share main memory and most other resources.

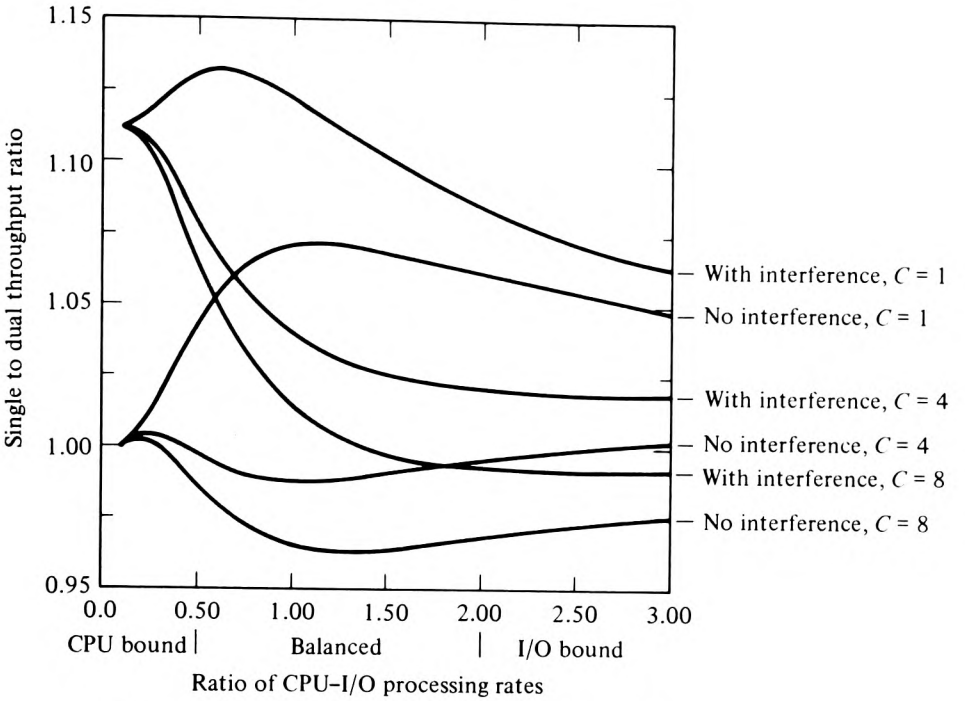
A multiple CPU system may have multiprogramming and/or multitasking. In multiprogramming, two or more independent programs reside in main memory and are processed in parallel. In multitasking, a program is decomposed into a partially ordered set of tasks where each task may be processed in parallel subject to precedence constraints. The interdependence between tasks causes the time required to process a program by $N$ CPU's, ($N$ at least 2) to be significantly greater than $1/N$ times the time required to process the same program by one processor. (Multitasking is desirable in multiprocessor systems to avoid idle processing capacity when only one job needs a CPU.)

Both multiprogramming and multitasking create contention for CPU's. With multiple processors we are likely to have memory interference (memory interference also occurs between channels and processors in single processor systems, but the amount of interference is usually small). Memory interference has been analyzed by Baskett and Smith [BASK76], Burnett and Coffman [BURN75] and Bhandarkar and Fuller [BHAN73], among others. Fuller [FULL76] says that memory interference studies consistently predict degradation factors of less than 10% for actual and proposed C.mmp configurations. Our paper did not study multitasking or memory interference in detail; however, results from these areas were used to suggest parameter values for our models.

### 9.2.1 First Come First Served CPU Scheduling

We shall compare, for different service distributions, a single CPU system with a dual CPU system where each CPU of the dual processor system has half the rate of the CPU of the single processor system. Degradation due to memory interference will be considered by reducing the rate of each CPU in a multiple CPU system. Multiple CPU systems allow for graceful degradation in service. However, the metric of interest in this subsection is system throughput, when the system is functioning normally.

**9.2.1.1. Impact of CPU service distributions.** It is reasonable to expect the single processor system to perform better than the multiprocessor system, since some of the CPU's in a multiprocessor system will be idle when there is only one program ready for CPU service. However, there is a compensating factor in favor of the multiprocessing system when the CPU service time has a high coefficient of variation. A single program requiring very long CPU service can bottleneck the CPU of a single processor system, while other programs queue up for CPU service, whereas it will monopolize

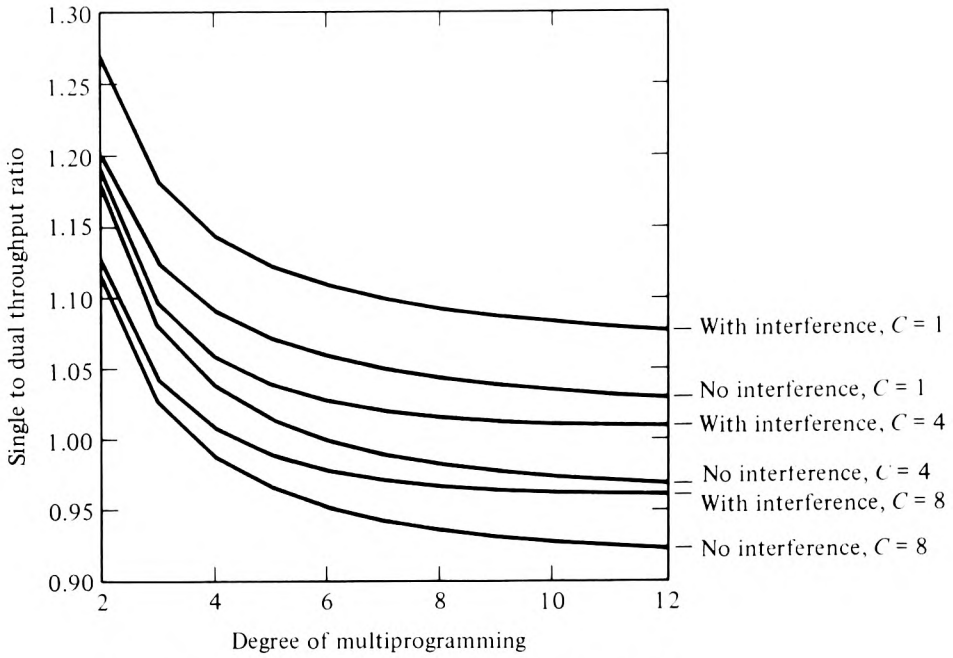



Single processor throughput/Dual processor throughput
Degree of multiprogramming = 5; Number of I/O's = 5
No memory interference; Dual processor memory interference

Figure 9.5

only one of the processors in a multiprocessor system allowing other jobs to go through remaining processors. Note that for small coefficients of variation (one or less) of CPU service times, the compensating factor in favor of multiprocessor systems does not generally apply. This is because with very large coefficients of variation it is more likely that there will be many short service times, and a few very long ones which bottleneck the single CPU. We should expect multiprocessor systems to perform better than single processor systems when CPU service time coefficients of variation are large.

**9.2.1.2. Impact of multiprogramming level.** When the level of multiprogramming is very small, the probability is also very small that there are enough jobs requiring CPU service to keep all CPU's busy in a multiprocessing system. Thus we should expect multiprocessing systems to have smaller throughputs than single processor systems for low degrees of multiprogramming. As the level of multiprogramming increases, the average number of busy CPU's in a multiprocessing system goes up, thus exploiting parallelism in the system. We should therefore expect the throughput of multiprocessing systems to improve more than that of single processor systems with increased levels of multiprogramming. This intuitive notion is



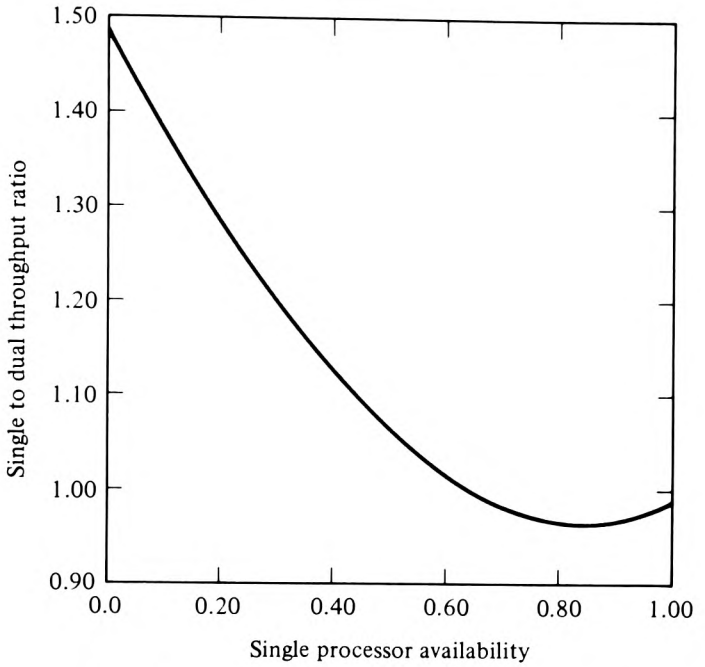

Single processor throughput/Dual processor throughput
Equal CPU and I/O processing rates: Number of I/O's = 5
No memory interference: Dual processor memory interference

Figure 9.6

supported by results from the models for all CPU service distributions studied.

**9.2.1.3. Models.** Two sets of models were used to analyze these systems. A cyclic queue model was used and recursive techniques (see Chapter 3) were used to analyze the model. I/O service times were assumed to be exponential, and it was assumed that all $L$ I/O devices shared a common queue. Three different CPU distributions were compared: (a) exponential (coefficient of variation ($C$) one), (b) hyperexponential with coefficient of variation 4, and (c) hyperexponential with coefficient of variation 8. Central server models with several I/O queues were analyzed using flow equivalence approximations and regenerative simulation techniques; this analysis yielded results similar to the cyclic queue model. We consider models with and without memory interference.

**9.2.1.4. Results for the interference free case.** The results obtained from the cyclic queue model are graphed in Figures 9.5 and 9.6; Similar results (not shown) were obtained for the central server model. The service rate of each CPU in the dual processor case was set to half that of the




Single processor throughput/Dual processor throughput
Equal CPU and I/O processing rates; Number of I/O's = 5
Degree of multiprogramming = 5, $C = 5$

Figure 9.7

single processor CPU. When the system is CPU-bound, (i.e. when the mean CPU queue length is much greater than the I/O queue length) the single processor system does not perform much better than the dual processor system since there are usually a sufficient number of programs desiring CPU service to keep all processors utilized in the multiprocessor system. When the system is I/O bound, changes in CPU processing rates do not significantly affect system throughput. Significant differences in system throughput between single and dual processor systems occur only when the system is well-balanced. The ratio of single to dual processor throughputs decreases with increase in the coefficient of variation of CPU service time (Figure 9.5) and with the degree of multiprogramming (Figure 9.6), as expected. Since CPU service time coefficients of variation are generally larger than 1, and often 8 or greater, we may conclude that no substantial reduction in throughput (more than 5 ) occurs by replacing single CPU systems by dual CPU systems with moderate or high levels of multiprogramming (i.e. greater than 3). Note that dual processor systems may be better than single processor systems in some cases!

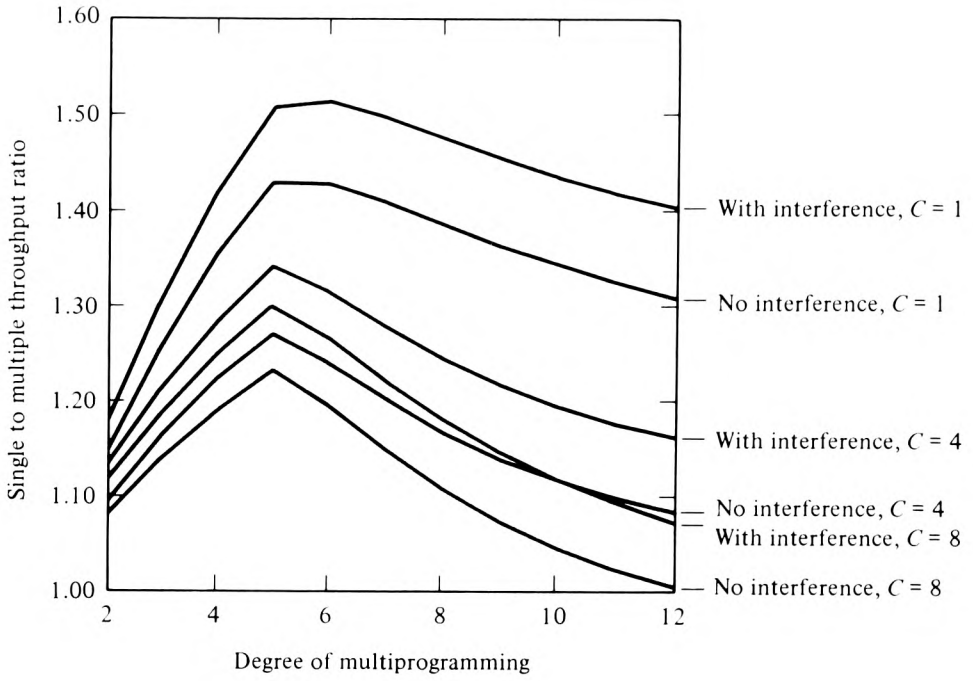
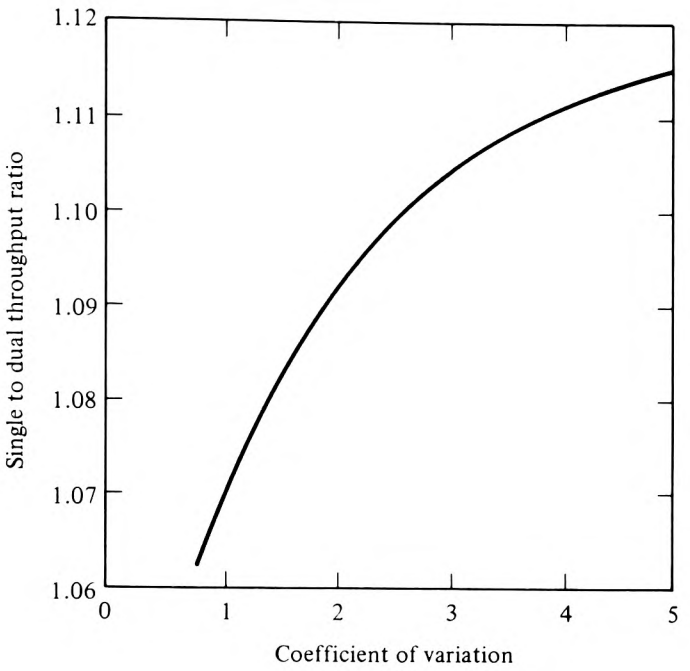



Figure 9.8

**9.2.1.5. Results for the with-interference case.** Fuller's analysis [FULL76] shows that an estimate of 10% degradation in CPU service rates due to memory interference is pessimistic for most C.mmp configurations. The degree of memory interference at any given time depends on the number of active CPU's at that time. A realistic model of multiprocessor systems with memory interference is to make the effective CPU service rate for each CPU decrease with the number of active CPU's; thus the effective service rate reflects the expected amount of memory interference. Our goal here is to obtain a clearly pessimistic estimate; the behavior of models with a realistic degree of memory interference will lie between the optimistic estimate of no interference and the pessimistic estimate. A clearly pessimistic estimate is to use Fuller's worst-case estimate and set the service rate for each processor in an n-processor system to 0.9/times the service rate of the CPU in a single processor system, independent of the number of busy CPU's.

As expected, CPU interference degrades the performance of multiprocessor systems with reference to single processor systems, especially when the system is CPU-bound, as shown in Figure 9.5. When the CPU service has an exponential distribution ($C = 1$) the ratio of single processor to multiprocessor throughput is larger for balanced systems than for CPU bound systems for the same reason that this effect is observed in the



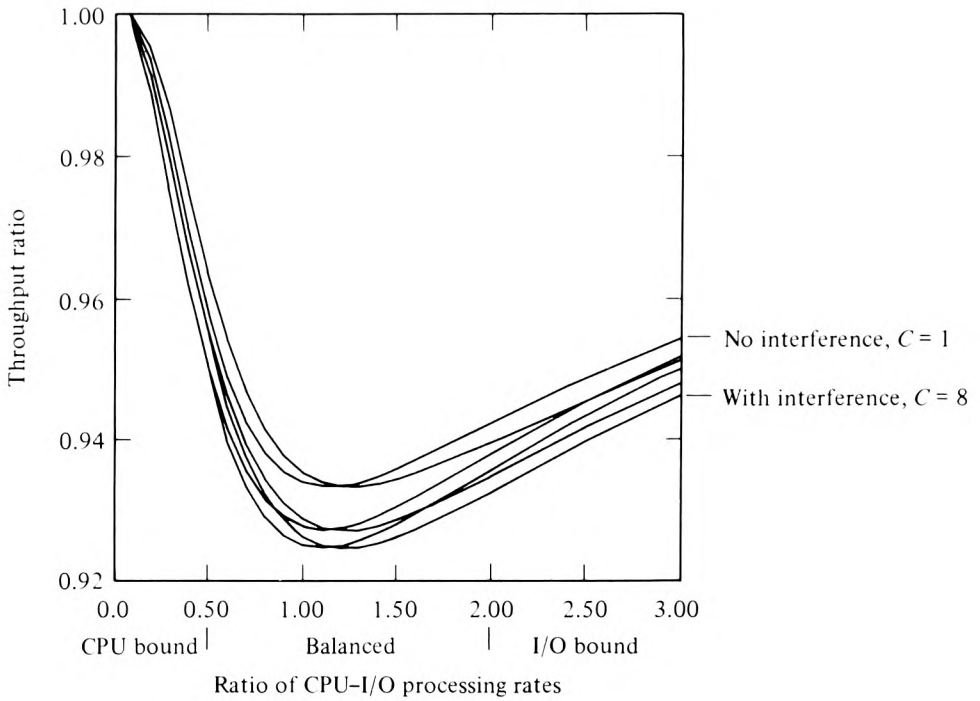

PS single processor throughput/FCFS dual processor throughput
Equal CPU and I/O processing rates; Number of I/O's = 5
Degree of multiprogramming = 5

Figure 9.9

interference-free case: When the system is CPU bound both processors in the dual processor system are busy all the time resulting in a combined throughput of 0.9 times that of the single processor system, resulting in a throughput ratio of 1/0.9. As the system gets more balanced the fraction of time that there are sufficient jobs to keep all CPU's in the dual processor system busy decreases, thus increasing the throughput ratio. However, for coefficients of variation of 4 and 8, the throughput ratio *decreases* as the system gets balanced because single processor systems are more likely to get bottlenecked by jobs with very long CPU bursts. For balanced systems, with typical CPU coefficients of variation the difference between single and dual processor throughputs is not substantial, even for pessimistic estimates of the degree of memory interference.

The ratio of single to dual processor throughputs decreases with increasing levels of multiprogramming as in the interference-free case.

**9.2.1.6. Impact of CPU availability.** It is instructive to study the impact of CPU availability on system performance. Assume that the fast CPU (in the single CPU system) and the slow CPU's (in the multiple CPU system)

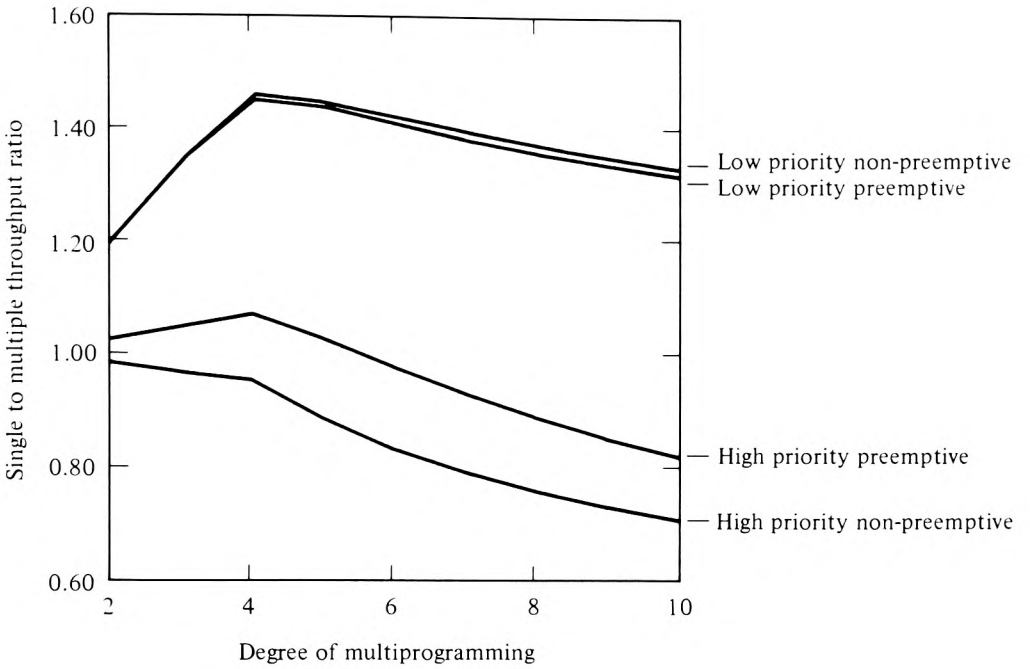



Throughput without multitasking/Throughput with multitasking
Degree of multiprogramming = 5; Number of I/O's = 5
No memory interference; Multitasking memory interferenc

Figure 9.10

have the same up-time and repair time distributions, and assume further that these times are independent random variables. If only one of the CPU's in the multiprocessor system is down, the system continues to function in degraded mode. We make the conservative assumption that if both machines are down, only one of them is repaired at a time. As Figure 9.7 shows, for certain ranges of single CPU system availability the multiprocessor system has a significantly better overall throughput. Additional discussion is found in [SAUE76b].

**9.2.1.7. Several processor configurations.** Configurations with a large number of CPU's may become more common as the cost of hardware decreases. Consider systems in which there are as many processors as there are jobs (degree of multiprogramming); in the cyclic model there are $N$ processors, each with $1/N^{th}$ the processing rate of the uniprocessor, where $N$ is the degree of multiprogramming. Each job in this system is assigned a dedicated processor. Figure 9.8 shows that the reduction in (CPU) resource sharing generally results in poor performance; however the throughputs of multiprocessing systems improve in comparison with uniprocessor systems



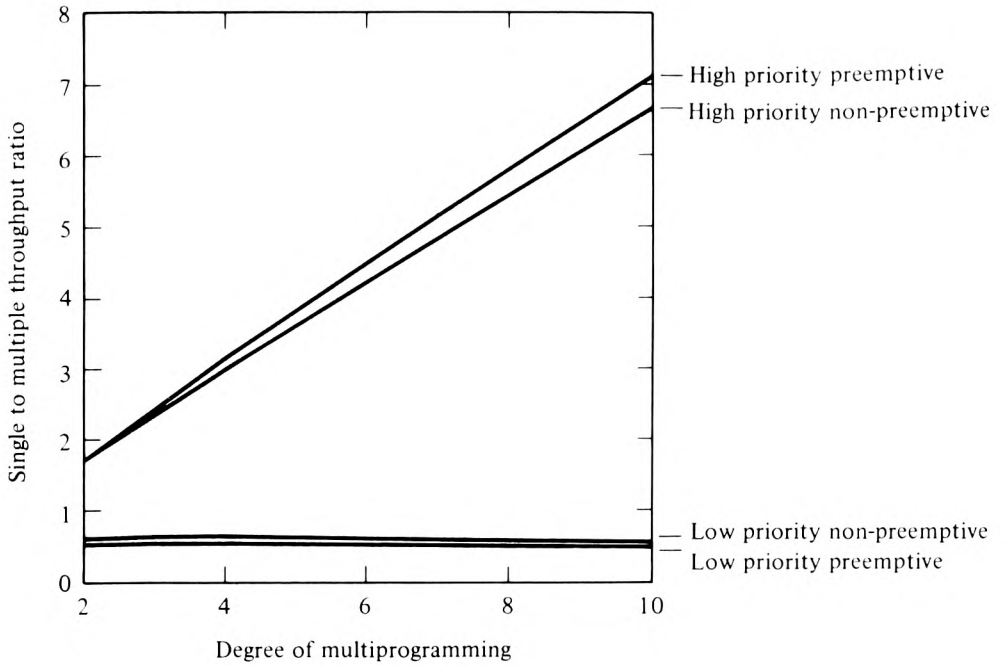

Single processor throughput/Multiprocessor throughput; Fast high priority jobs
Mean I/O time for all jobs = 40; Number of I/O's = 4
Single processor mean CPU time for high priority = 1
Single processor mean CPU time for low priority = 10
Preemptive priority; Non-preemptive priority

Figure 9.11

with increasing CPU service time coefficient of variation and with increasing levels of multiprogramming. (The latter part of this statement assumes the processors do not become more numerous and slower with increasing levels of multiprogramming, unlike Figure 9.8.) For certain cases [SAUE76b] multiprocessor systems may have greater throughput than uniprocessor systems.

In summary, high CPU service time coefficients of variation and levels of multiprogramming improve the performance of multiprocessor systems in comparison to single processor systems; in some case multiprocessor systems have throughputs competitive with single processor systems.

We may take the idea of reducing resource sharing to an extreme, as suggested by Martin and Frankel [MART75] and study a multiprocessing system in which each job is assigned a dedicated CPU and a dedicated scratch disk; jobs only share the permanent file system. As expected [SAUE76b] reduced resource sharing may result in substantial (up to 67%) reduction in throughput.




Single processor throughput/Multiprocessor throughput; Slow high priority jobs
Mean I/O time for all jobs = 40; Number of I/O's = 4
Single processor mean CPU time for high priority = 100
Single processor mean CPU time for low priority = 10
Preemptive priority; Non-preemptive priority

Figure 9.12

### 9.2.2. Other Disciplines

CPU service disciplines have been studied intensively with respect to single processor systems. We shall attempt to extend the study to multiprocessor systems. If the CPU discipline is changed from first come first served to round robin fixed quantum, the performance of single processor systems improves more than that of multiprocessor systems for typical hyperexponential CPU service distributions. Figure 9.9 compares the throughput ratios of a single processor system with a processor sharing discipline to a dual processor system with a first come first served discipline as a function of the coefficient of variation of CPU service time. Note that the dual processor system is never better than the single processor system in this case, and the single/dual throughput ratio may exceed 1.1. Generally speaking, multiprocessor systems are less sensitive to scheduling disciplines than uniprocessing systems as illustrated by Figure 2.6. Figure 2.6 shows the ratio of the throughput of a system with processor sharing CPU discipline to that of a system with a first come first served discipline for the single processor and dual processor systems as a function of CPU coeffi-

cient of variation. FCFS is the simplest scheduling discipline to incorporate into the operating system, and it has less overhead than Round Robin. The curves suggest that first come first served is satisfactory for dual processor systems for typical CPU coefficients of variation and degrees of multiprogramming; the same is not true for single processor systems.

CPU scheduling has been an important part of system performance tuning and a great deal of work has been done in this area; see [SHER72a] for instance. The literature contains discussions of various schemes to predict future job behavior on the basis of past behavior using statistical estimators such as moving point averages and exponential smoothing, and to schedule CPU's on the basis of predicted job behavior. Let us make the optimistic assumption that all future service times can be predicted with total accuracy and that the *Shortest-Remaining-Time-First* (SRTF) discipline is used to schedule CPU's. Figure 2.7 shows the ratio of throughputs in the SRTF and FCFS cases. The percentage improvement in going to the SRTF discipline from the FCFS discipline is small for dual CPU systems. An even smaller gain in throughput would accrue from using statistical predictors of job behavior such as exponential smoothing. We conclude that CPU scheduling has much less impact on multiple CPU systems than on single CPU systems. Equivalently, multiple CPU systems favor simplicity in scheduling strategies.

### 9.2.3. Multitasking

A great deal of work has been carried out on scheduling two or more processors to concurrently process a single program. We next address the question: How effective is multitasking in improving system throughput, and in particular, what impact do CPU service distributions have on multitasking? We compare two multiprocessing systems, one which allows multitasking and the other which does not. We shall use very optimistic models of multitasking with a view towards getting an upper bound on the improvement in throughput due to multitasking; more realistic models are considered later. We assume that in the multitasking system two or more CPU's cooperate on a single program if, at any given time, there are fewer programs requiring CPU service than there are CPU's. However, if there are at least as many programs requiring CPU service as there are CPU's, then each CPU works on an independent program. Thus CPU bound systems do not offer much opportunity for multitasking. Multitasking of I/O's is not permitted in this model. If the system is I/O bound, multitasking the CPU does not impact performance. Thus we would expect the greatest percentage increase in system throughput with a well balanced system.

Consider the cyclic queue model. Assume that when two processors cooperate on a single program, the time required to complete CPU service

for that program is $1/K$ times the time required to complete CPU service on a single CPU, where $K$ is between 1 and 2. As shown in Figure 9.10, the maximum benefit from multitasking is obtained when the system is well balanced and the CPU coefficient of variation is high. Similar results are obtained for central-server models with multiple I/O queues. Note that the results of Figure 9.10 are very optimistic, since we assume perfect cooperation between processors ($K = 2$). The benefit of multitasking increases with the variance of CPU service time because the ratio of the probability of exactly one job in the CPU queue increases with CPU variance. In other words, increasing CPU variance results in greater opportunity for multitasking.

The actual benefits of multitasking will be significantly less than indicated in the figures due to the overhead involved in the multitasking process and due to the interference between processors. Ramamoorthy and Gonzalez [RAMA69] suggest that $K$ is generally less than 1.5 and that values of 1.1 are not atypical. For $K = 1.1$ and the other parameters as in Figure 9.10, the improvement obtained by multitasking is less than 1%. We may conclude that multitasking does not have substantial impact at reasonable degrees of multiprogramming. Once again our analysis suggests that multiprocessor systems favor simplicity in scheduling strategies if the objective is to maximize throughput.

The concept of multitasking is closely related to that of CPU-I/O overlap, where a program splits itself into two subtasks, with one subtask requiring CPU service and the other requiring I/O service. Price [PRIC75] and Towsley [TOWS75] have shown that the benefit of CPU-I/O overlap also decreases with increasing levels of multiprogramming.

### 9.2.4. Priority Disciplines

It can be argued that though fast single CPU systems are not a greater deal better than systems with several slow processors when the metric of interest is overall throughput, there is an important metric for which fast single CPU systems are clearly better. There are some environments where it is crucial that a small number of special jobs have very short turn-around times while the bulk of jobs are not time-critical. Intuitively, one expects fast single processor systems to yield shorter turn-around times of high priority jobs because the entire CPU resource can be devoted to a single high priority job. The models show that though this assumption is generally true, it does not always hold!

Once again let us compare a single CPU system with an $N$ CPU system where each CPU is $1/N$ times as fast. We may consider memory interference by reducing the rates of each CPU in the multiprocessor system by an

appropriate amount. Let the degree of multiprogramming be $N$, and assume that one of the $N$ jobs is a high priority job while the remaining $N - 1$ jobs are low priority. Consider the case where the single CPU has a priority scheduling policy, either preemptive resume or nonpremptive. Figures 9.11 and 9.12 show the ratios of the throughputs of uniprocessing and multiprocessing systems for the high priority (and low priority) jobs. The figures differ in the relative processing requirement of the high and low priority jobs. We assume exponential CPU service times, but we expect performance trends to be the same with realistic CPU service distributions. The single processor system behaves considerably better if the metric of interest is the throughput (or response time) of the high priority program and the I/O subsystem is not heavily loaded. As shown in Figure 9.11, the multiprocessor system may actually perform *better* for high priority jobs when the I/O subsystem is heavily loaded and the degree of multiprogramming is high! This is because the uniprocessor system has much higher throughput for low priority jobs and these jobs cause high priority jobs to spend more time queueing for I/O. (Of course this effect could be decreased or eliminated if a priority discipline is used for I/O.) When the I/O subsystem is not saturated, or when the CPU requirements of the high priority jobs are high relative to the I/O requirements, these effects do not occur, as shown in Figure 9.12. If the CPU discipline is non-preemptive priority, the advantage of the single CPU system is not as great.

In summary, the interaction between CPU priorities and multiprocessing is complex; care must be used in exercising intuition in such cases.

## 9.3 A DATA MANAGEMENT SYSTEM MODEL

The Advanced Logistics System (ALS) modeled by Browne *et al* is considerably more complex than the systems of the previous two sections. Yet, the final model used in the ALS study is not all that different from the simpler queueing network models of these systems. There are several explanations for this situation: First, the modeling project was hierarchical in both models and personnel. The "final model" was the top level of a two level hierarchy of models. Second, the purpose of the model was to estimate system capacity (throughput) and device utilizations, not response times. Third, the queueing network model and companion simulation model were to be constructed in a very short period of time.

Because the Cyber 73 and 74 mainframes share the Extended Core Storage (ECS) of one million words and use it in many ways analogous to primary memory (with the private memories of the 73 and 74 analogous to cache memory) and because the 100 Data Management System (DMS) disk drives are shared between the two machines, it is reasonable to use a variation on the central server model as the top level model of the system.

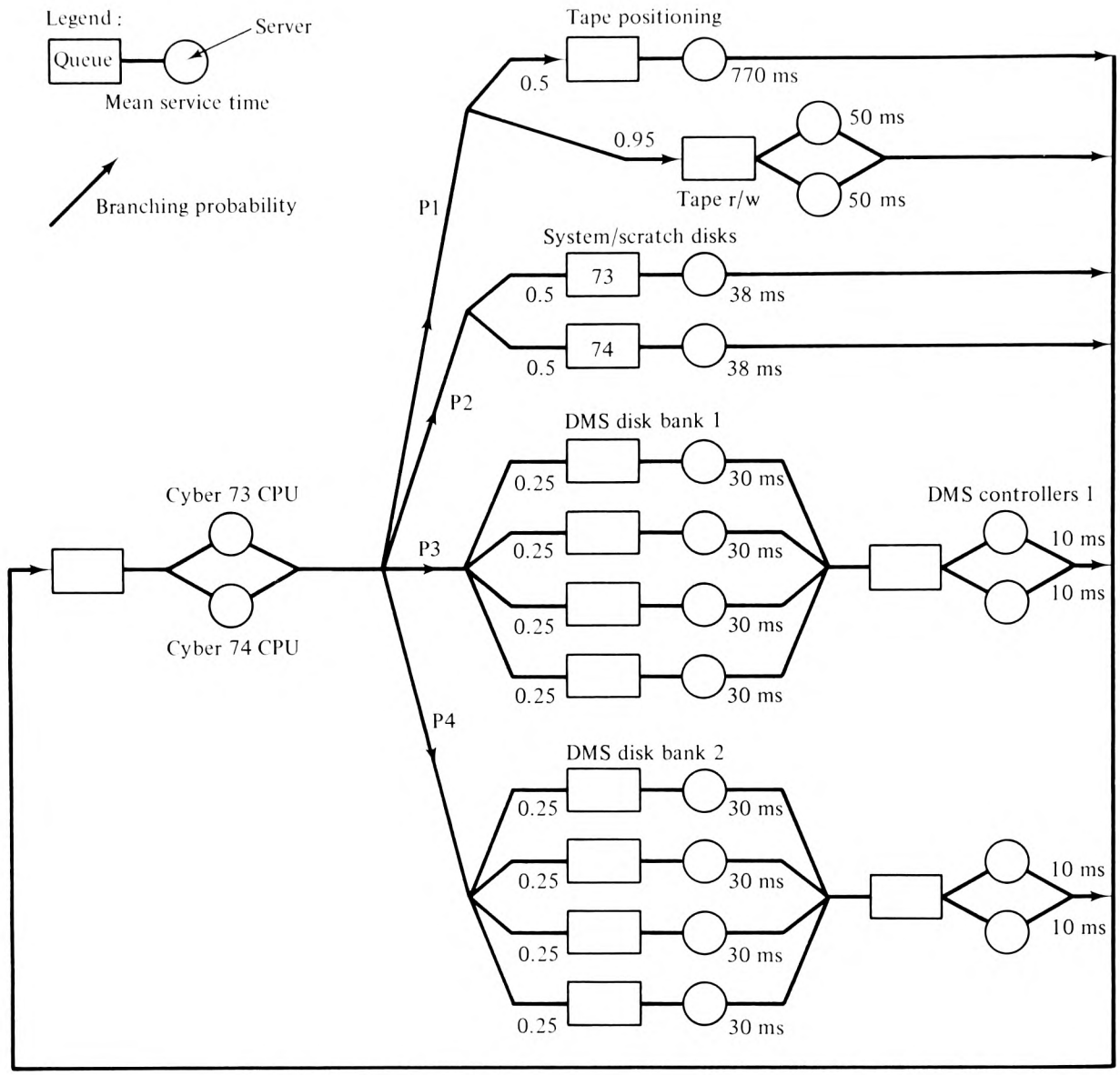



Figure 9.13

(Figure 9.13, a copy of Figure 1.2, shows the top level model.) Though not *physically* tightly coupled, it is reasonable to consider the CPU's to be *logically* tightly coupled. Figure 9.13 gives the model parameters except for the degree of multiprogramming, the CPU service times and the branching probabilities P1, P2, P3 and P4. As is common in models of this sort, model results are obtained for all possible degrees of multiprogramming, from 1 to 17 in this case. The CPU service times are determined to be 8.3 ms. for the

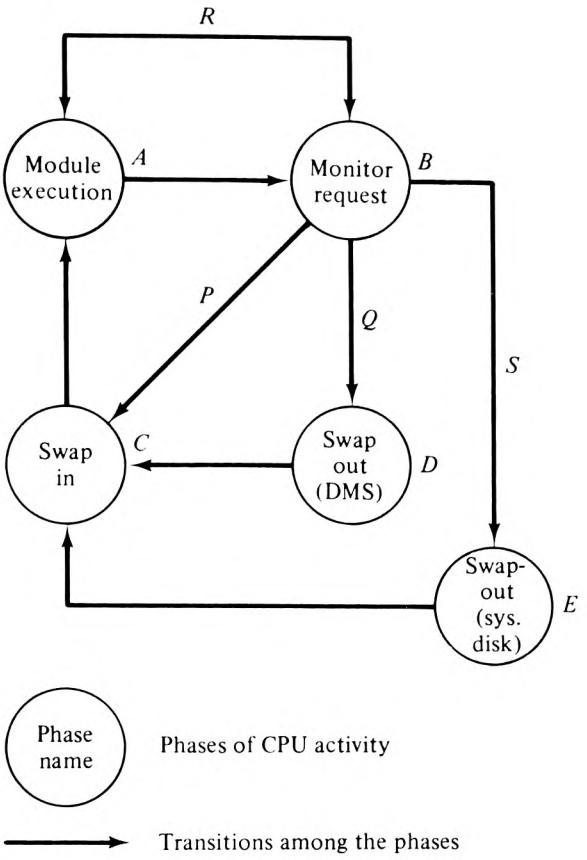



Figure 9.14

Cyber 84 and 13.3 for the Cyber 73, as described below. (All service times are treated as if exponential.) The remaining branching probabilities depend on details of the I/O system.

### 9.3.1 CPU, Central Memory and ECS Submodel

In the basic mode of operation, a transaction is serviced by a collection of modules in central memory of one of the Cybers. Usually a portion of only one transaction's modules will be resident in one of the central memories. A module executes until it issues a monitor call to either call upon some other module or to carry out an I/O activity. In the first case, the module may either already be in memory or need to be swapped in from disk or need to be swapped in from ECS. In the second case the I/O access may be either to the DMS or to the system disk or tape. In the second case, the module is swapped out to ECS, that transaction's I/O is handled, a module for another transaction is swapped in and that transaction's execu-

tion begins (or continues after the completion of I/O). Note that the CPU carries out swapping functions. Figure 9.14 shows the phases of CPU activity and the transitions between phases. The variables $A$ through $E$ identify mean CPU times for each phase and the variables $P$ through $S$ identify probabilities of results of monitor calls. From Figure 9.14 we can derive the probability of an access to the DMS given a swap out as

$$P_{\text{DMS}} = Q/(Q + S). \tag{9.1}$$

Similarly, the probability of an access to the system disk or tape given a swap out is

$$P_{\text{SYS}} = S/(Q + S). \tag{9.2}$$

The mean time between swap outs (mean CPU service time) is obtained as

$$T_{\text{CPU}} = A + B + C + (R(A + B) + P(A + B + C) + QD + SE)/(Q + S) \tag{9.3}$$

by treating Figure 9.14 as a Markov state diagram. Figure 9.15 gives these values for each CPU, with times in microseconds.

| CPU | $A$ | $B$ | $C$ | $D$ | $E$ | $P$ | $Q$ | $R$ | $S$ | $P_{\text{DMS}}$ | $P_{\text{SYS}}$ | $T_{\text{CPU}}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 74 | 113 | 152 | 765 | 2350 | 600 | .05 | .025 | .9 | .025 | .5 | .5 | 8305 |
| 73 | 200 | 293 | 850 | 2850 | 727 | .05 | .025 | .9 | .025 | .5 | .5 | 13349 |

Figure 9.15

The discussion so far has assumed that the two CPU's do not interfere with each other. Actually, the ECS is divided into two independent segments. As long as both CPU's do not try to access the same segment simultaneously, there is no interference. However, if both attempt simultaneous access to the same segment, one must wait. Browne et al used a five state Markov model to estimate the amount of interference. In the worst case it was estimated that the CPU service times would be increased by 8.3% and 10.6% for the 73 and 74, respectively, because of this interference in the worst case.

### 9.3.2 Tape and System/Scratch Disk Submodels

The tape system needs little consideration because of the small fraction of I/O accesses (about 3 ) to it. For our purposes Figure 9.13 is enough.

The system/scratch disks are separated into two banks of four, with one bank dedicated to each CPU and a single controller for each bank. Because of the software characteristics, there is no concurrency within a bank. Thus each bank may be treated as a single server queue with service time equal to the sum of the mean seek, latency and transfer times. The model treats the banks as if they were accessible by either Cyber for simplicity. This gives the effect of slightly more capacity than actually available. As we said in Chapter 1, the model shows the system/scratch disk system to be the most severe bottleneck, even with this optimistic assumption.

### 9.3.3 DMS Submodel

The Data Management System consists of roughly 100 disks, separated into two banks. Each bank has a channel and a controller attached to each Cyber 70 mainframe. The disks have position sensing capabilities and only need be connected to the channel and controller during transfer operations. A disk access consists of the following stages, once the disk is acquired: Initiation – The controller is acquired and used to initiate positioning; the controller is then free to service some other disk. This stage is very sort, i.e. less than 1 ms. long. Positioning – The arm is moved to the necessary cylinder and the disk rotates to the necessary sector for transfer. This stage has a mean length of 30 ms. Transfer – The controller is reacquired and information is transferred to or from the disk. This stage has a mean length of 10 ms. Figure 9.16 illustrates these stages.

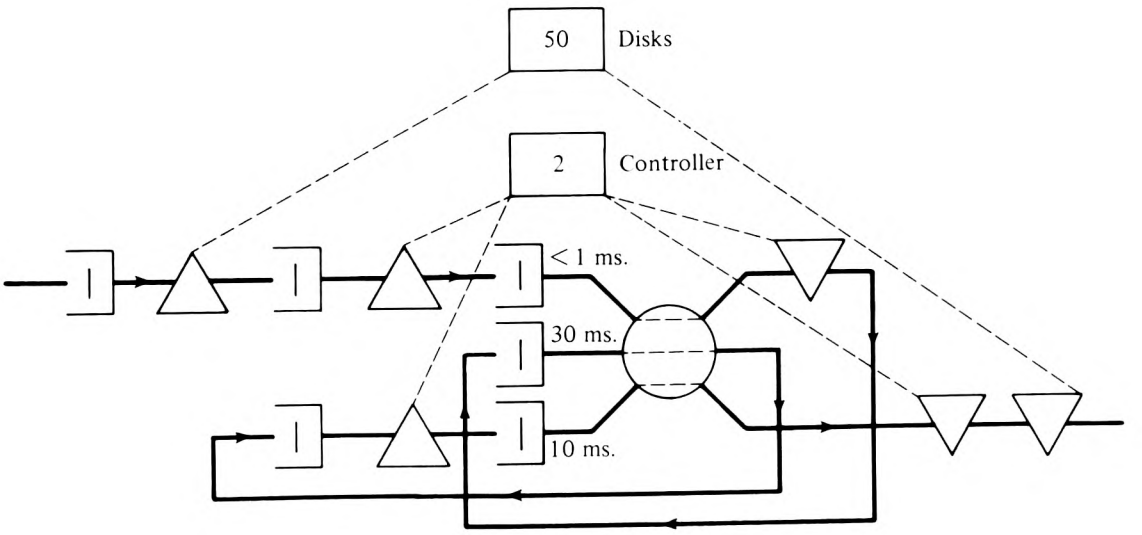



Figure 9.16

This figure is based on figures and discussion in BROW75 but did not appear there. The figure and discussion above ignore potential performance effects of the actual system, for example, if the controller cannot be quickly reacquired after the positioning stage, then a repositioning stage (another rotation to get back to the sector) may be necessary. Still, a model as detailed as Figure 9.16 as part of the top level model would preclude exact numerical solution of the top level model (Figure 9.13). (With large populations, exact numerical solution of the submodel of Figure 9.16 would also be impractical. Regenerative simulation of such a model is feasible [SAUE77c].) The authors decided that the most important aspect of the DMS disks was the possible simultaneous positioning and transfer stages of two (or more) disks in the same bank. This led to the representation in Figure 9.13, with separate queues for positioning and transfer. Note that Figure 9.13 allows some impossible situations in the actual system, e.g., simultaneous transfer and positioning on the same disk. After studying six such inaccuracies of the representation of Figure 9.13, using submodels, the authors concluded that the inaccuracies had little effect on the model results. In the figure there are four queues (per bank) labelled "Disk Bank." These represent the positioning stage. The initiation stage is ignored. The number four was chosen because it was highly unlikely that more than four positioning stages would occur concurrently, based on system data. The two server queue labelled "Controllers" represents the transfer stage.

### 9.3.4 Simulation Model, Validation and Predictions

The model as described was intended to be used to study the effects of a large number of system parameters, so that model results could be used to guide development and configuration of the system. However, since the system was not yet operational, there was no way to directly assess the accuracy of the model. A more detailed simulation model was constructed, both to convince the analysts that their model was sufficiently accurate, and to convince the designers and others that the model was valid. The simulation model was also constructed in a hierarchical manner, with four analysts constructing relatively independent components and a fifth constructing the control and interface portions of the simulation program. This division of labor allowed the entire program to be completed in approximately one and a half months. (As we said in Chapter 1, the entire modeling effort took about 2 months for the six analysts to complete.) The published utilization estimates as produced by the two models were within 5% agreement.

As we said in Chapter 1, the (numerically solved) model was used to make two major predictions: First, that the system/scratch disk subsystem would be a major bottleneck. Second, that if that subsystem were redesigned to increase scratch disk capacity, then performance of the system

would be unacceptable because of insufficient CPU capacity. Both of these predictions were confirmed by subsequent operating experience.

We have necessarily omitted many details of the modeling effort, particularly in details of the submodels. The reader is referred to the original paper [BROW75] for a more thorough discussion.

## 9.4 A MODEL OF AN INTERACTIVE SYSTEM

In the cyclic queue model, the central server model and the modified central server model of BROW75, memory contention is indirectly considered by restricting the population of jobs in the model. This is usually sufficient to get estimates of utilizations and throughputs, but there is no attempt to estimate times spent waiting for memory and thus there is no attempt to estimate response times. (Though our attention is now on interactive systems, the same statements apply to turnaround times in batch systems.) The approach used in BROW77 is an excellent example of a general approach used to consider memory contention and estimate response time. Though the model is of a nonpaged system, the approach is also suitable for paged systems, as we will see in Section 9.5.

The modeled system consists of a CDC 6600 and a CDC 6400 at the University of Texas at Austin. The operating system (UT-2D) is locally implemented with its roots in early CDC operating systems. The usual mode of operation has the interactive service on the 6400 and the batch service on the 6600. 500,000 words of Extended Core Storage (ECS) are used as a swapping device to the 64,000 words of central memory of the 6400. There are also 4 CDC 808 disks and 8 CDC 841 disks.

Figure 9.17 describes the activity phases of an interactive job in the system. After completion of terminal input (block 1) the job is ready to run. It waits for the memory scheduler to be run (block 2), which does not occur until there is a change in status of a job already in memory. When the scheduler runs (block 3) the job may or may not be allocated memory. After the scheduler allocates memory to the job it must wait for resources needed to swap the job into memory (block 4). The resources required are a contiguous block of memory (compaction may be necessary to make available memory contiguous) and a peripheral processor to initiate the swap-in. After the swap-in (block 5), the job can perform its computation while resident in memory (block 6). The job will leave memory either because it completes its computation or because its memory is being preempted. (In the UT-2D system, unfinished interactive jobs are preempted from memory after one second of memory residence. Memory scheduling is partially based on a round robin strategy with this one second quantum.) After completion of computation, the job again waits for a scheduler run (block

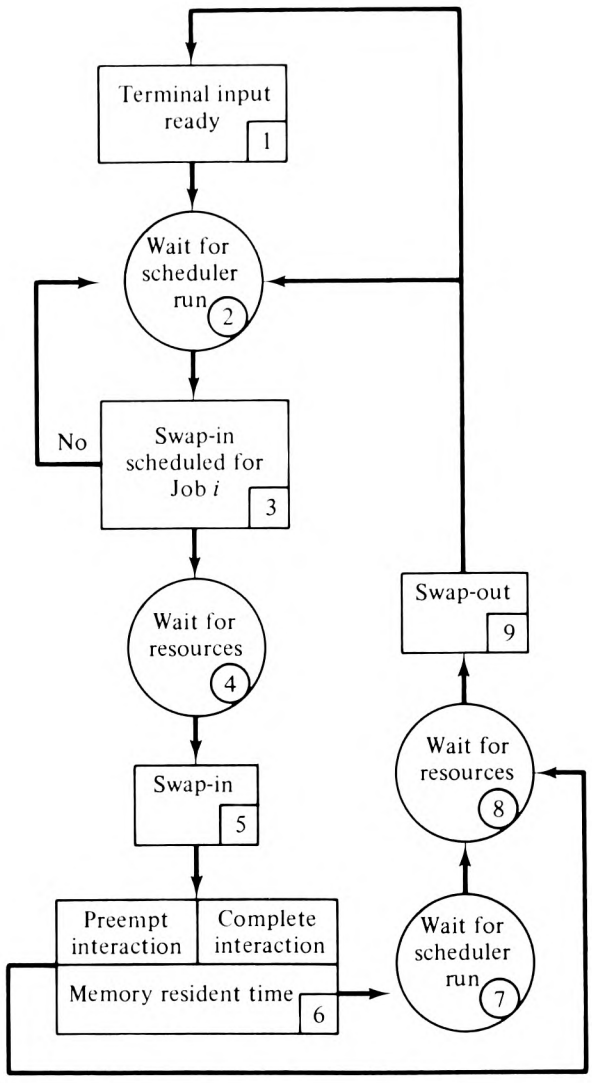



Figure 9.17

7) until the scheduler recognizes the job's status has changed. The job then waits for a peripheral processor to initiate the swap-out (block 8). After the swap-out (block 9) the job returns to wait for the scheduler, if it has been preempted, or to wait for completion of terminal input, otherwise.

Figure 9.18 is the queueing network model corresponding to Figure 9.17. (Figures 9.18, 9.19 and 9.20 are copies of Figures 1.3, 1.4 and 1.5, respectively.) In Figure 9.18 the times spent waiting for scheduler runs (blocks 2 and 7 of Figure 9.17) are included as part of other times which are explicitly shown. The "swap delay" queue corresponds to block 4. The

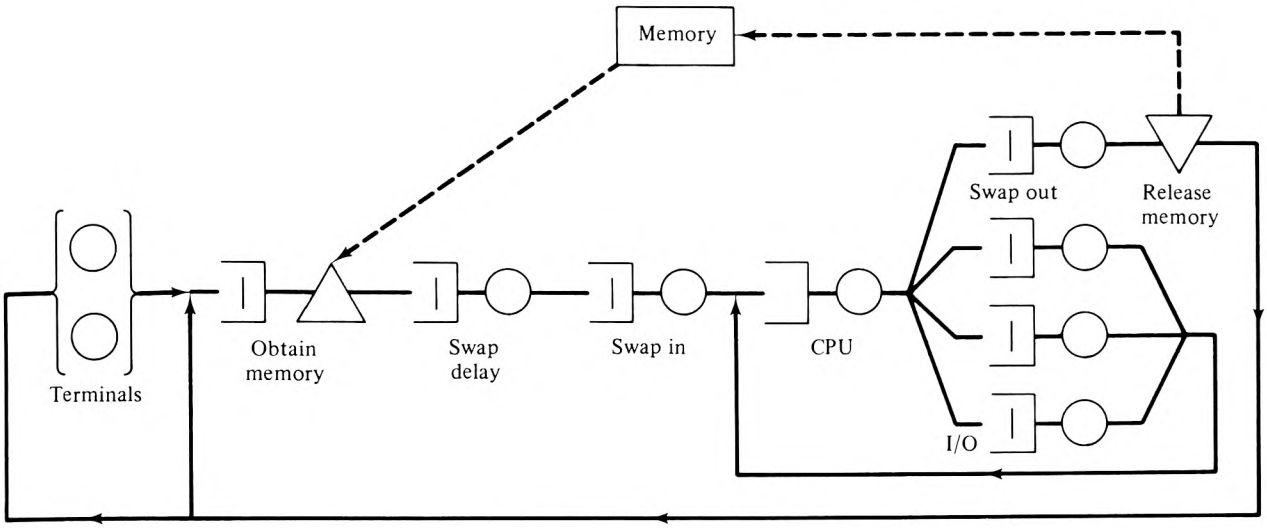



Figure 9.18

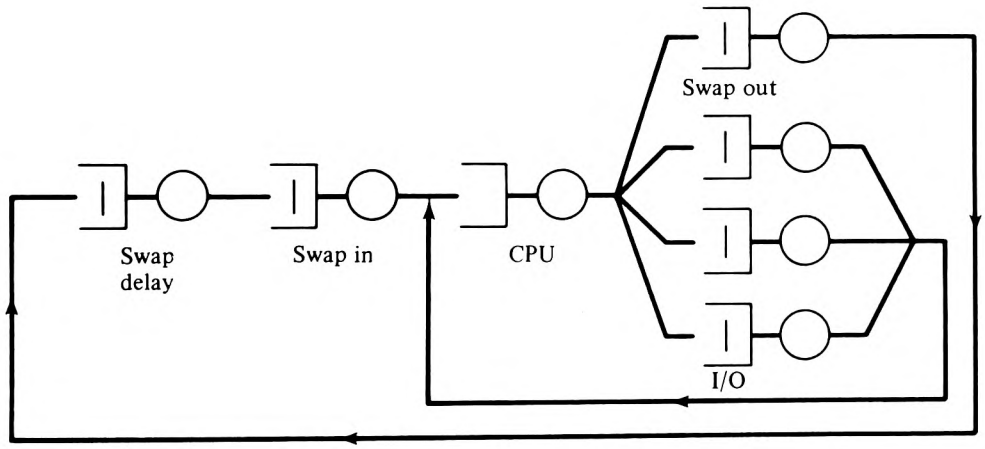



Figure 9.19

contention for peripheral processors is not directly considered in the model, but the mean time to wait for a peripheral processor for a swap-in is included in the swap delay queue service time. The mean time to wait for compaction is also included in that service time. A central server model is embedded within the model to represent CPU and I/O activity.

Numerical parameters for the model are obtained from the software monitor built-in to UT-2D. The measurements indicate that it is reasonable to consider all service times except the CPU times to be exponential. CPU

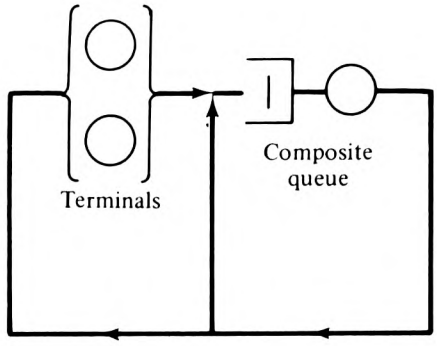



Figure 9.20

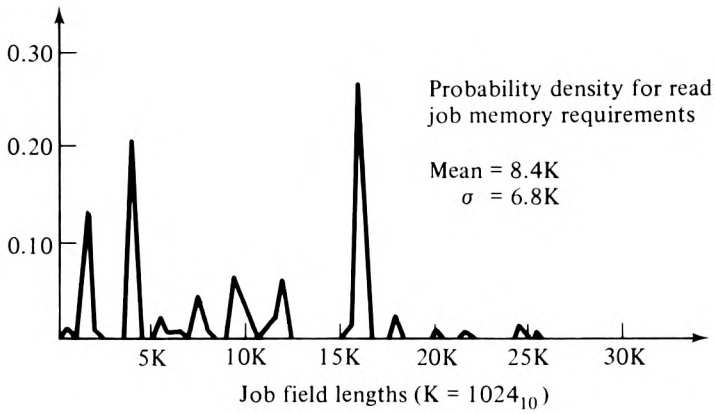



Figure 9.21

scheduling is round robin with a 16 ms. quantum and about 0.5 ms. switching overhead. The mean interactive CPU time is slightly less than the quantum, contrary to our criteria for representing round robin as processor sharing in Chapter 2, but it is not unreasonable to represent the CPU scheduling as processor sharing since the criteria are nearly satisfied. Thus the model of Figure 9.18 would satisfy product form conditions if it were not for the memory queue.

From our discussion in Chapter 6, a flow-equivalence approximation is appropriate for this model. By eliminating the terminals and memory from Figure 9.18, we get the network of Figure 9.19, which does satisfy product form and is easily solved. The throughputs from Figure 9.19 for the possible populations, along with the characterization of the memory requirements, can be used to produce a composite queue characterization in Figure 9.20.

That figure's model also will have a product form solution and can be easily solved. The one difficult aspect of this approximation which we have not considered before is the more detailed memory contention representation in BROW77.

Before, we have assumed that each job required the same amount of memory (in Chapters 6 and 7). However, even though jobs may have fairly homogeneous behavior, the amount of memory they require fluctuates from time to time. It is more reasonable to assume that there is a probability distribution characterizing a job's memory requirements. Figure 9.21 shows a probability density function observed by Brown on the CDC 6400. The UT-2D system limits interactive jobs to 32K of memory. Note the three high density spikes in the distribution. These correspond to frequently used utilities and systems (editors, Basic, etc.).

Let us assume that there are $N$ jobs in the system. We desire a value $a$ and a function CAP($n$) such that $a$CAP($n$) is the service rate of the composite queue with length $n$, $n = 1,2,...,N$. We have obtained, by solution of the model of Figure 9.19, $R(n)$, the throughput in that model with a population of $n$ jobs, $n = 1,2,...,N$. Let us assume we can find a function $h(i \mid n)$ which is defined as the probability there are $i$ jobs in memory given $n$ ready jobs, $i = 1,2,...,n$, $n = 1,2,...,N$. Then

$$a\text{CAP}(n) = \sum_{i=1}^{n} R(i)h(i \mid n), \; n = 1,2,...,N. \tag{9.4}$$

So our problem is to find $h(i \mid n)$. This can be fairly easily done under two assumptions made in BROW77, for certain scheduling algorithms. Except for the cases considered in BROW77, this problem has received little attention and remains unsolved (except for simulation).

It is assumed that (1) with each scheduler run, the memory requirement of each job is determined from the distribution (e.g. Figure 9.21) without regard to the job's previous memory requirement, and (2) with each scheduler run a *fresh* decision is made with regard to each job, regardless of whether it is currently allocated memory or not. (Thus a job holding memory may have that memory preempted regardless of other scheduling policies, e.g., the one second round robin policy of UT-2D.) The scheduler most like the UT-2D scheduler is *First Fit* (FF) which allocates memory to jobs in FCFS order as far as possible. However, if a job cannot fit in the available memory, and another job with a later arrival time can fit in the available memory, then that job is allocated memory, in violation of FCFS order.

Let there be $C$ units of memory, let $p(c)$ be the probability that a job requires $c$ units of memory, $c = 1,2,...,C$, and let $P(c)$ be the cumulative function of $p(c)$, i.e., $P(c) = p(1) + p(2) + ... + p(c)$. Consider the scheduler as it decides on the jobs of the queue, in order. Let $g(c,n \mid l)$ be the probability that the scheduler has allocated $c$ units of memory to $n$ jobs given that it has considered $l$ jobs, $c = 0,1,2,...,C$, $n = 0,1,2,...,N$, $l = 0,1,2,...,N$. From the definition,

$$g(c,n \mid l) = \begin{cases} 1, & \text{for } c = n = l = 0, \\ 0, & \text{for } c > 0,\ n = l = 0, \\ 0, & \text{for } n > l, \text{ for all } c. \end{cases} \tag{9.5}$$

In general,

$$\begin{aligned} & g(c,n \mid l+1) \\ & = g(c,n \mid l)(1 - P(C - c)) + \sum_{i=0}^{c} g(i,n - 1 \mid l)p(c - i), \end{aligned} \tag{9.6}$$

for $n = 1,2,...,N$, $l = 1,2,...,N - 1$. The first term of (9.6) corresponds to the $l+1^{th}$ job requiring more memory than is available. ($c$ units have been allocated to the $n$ jobs after consideration of $l$ jobs. $P(C - c)$ is the probability the $l+1^{th}$ job requires at most $C - c$ units, so the probability the $l+1^{th}$ job requires more than $C - c$ units is $1 - P(C - c)$.) The remaining terms correspond to $i$ units having been allocated to $n - 1$ jobs after consideration of $l$ jobs. In each of these cases the $n^{th}$ job allocated memory must require $c - i$ units. Using (9.5) and (9.6) we can obtain $g(c,n \mid l)$ for all required values.

Having $g(c,n \mid l)$ we can determine

$$h(i \mid n) = \sum_{c=0}^{C} g(c,i \mid n). \tag{9.7}$$

Equations (9.4), (9.5) and (9.7) apply to all scheduling algorithms considered by Brown et al. Schedulers other than FF will result in different equations (possibly more than one per scheduler) corresponding to equation (9.6).

One of the objectives of the modeling effort was to demonstrate that a systems designer could effectively use such a model. In such usage, some parameters will vary but the designer will be unable to predict the variation. Thus the designer is forced to use fairly fixed estimates. In comparing the model with measured results, some parameters were held fixed even though more accurate values for the parameters were known. In particular: (1) The

amount of memory available to user programs ($C$) was assumed to be 33K. The remainder of memory is used for the operating system and for terminal buffers. The actual amount of available memory fluctuated from about 32K to 34K depending on the number of logged on users. (2) The mean user think time was assumed to be 18.7 seconds. (3) In the system, user activity accounts for only a small fraction of CPU activity. The model CPU times are obtained by dividing the measured CPU times by this fraction. This fraction was assumed to be .165. The actual values for three measurement periods ranged from .15 to .18. (4) The mean disk service times were assumed to be 100 ms. (5) The swap delay, swap in and swap out queue mean service times were assumed to be 60, 20 and 30 ms., respectively.

The input parameters to the model were (1) the number of users, (2) the probability that a job releasing memory has been preempted (the probability of going from block 6 to block 8 in Figure 9.17, or, equivalently, the probability of going from the release node to the allocate node in Figure 9.18), (3) the job memory requirement distribution, $p(c)$, (4) the probability a job is swapped out after finishing the CPU queue, and (5) the mean user CPU time.

| | | CPU Util. | | Mean Resp. | | Deg. of M.P. | |
|---|---|---|---|---|---|---|---|
| Period | Users | Meas. | Mod. | Meas. | Mod. | Meas. | Mod. |
| 1 | 52 | 0.93 | 0.80 | 1.32 | 1.27 | 3.67 | 3.47 |
| 2 | 30 | 0.93 | 0.86 | 1.24 | 1.60 | 3.30 | 3.99 |
| 3 | 49 | 0.83 | 0.71 | 0.97 | 0.74 | 2.35 | 3.13 |

Figure 9.22

Figure 9.22 shows some of the performance metrics as measured and predicted for three measurement periods. Though the agreement is not as close as we might hope, such accuracy should be sufficient in the design and development stages of a system. Note that this is a very simple model of a fairly complex system, and recall that *several model parameters are intentionally fixed beforehand* rather than based on the measured data. In particular, the authors report much better agreement between measured and predicted values when the measured fraction of CPU time attributed to users is used rather than the fixed value (.165).

## 9.5 THE VM/370 PERFORMANCE PREDICTOR

Vendors of computer systems need methods for predicting performance of the computer systems they provide. Otherwise, they may either (1) underestimate the resources required and provide a system with unacceptable performance, or (2) overestimate the resources required and lose the customer to a competitor with a lower bid. The VM/370 Performance

Predictor [BARD77b,BARD78a] is used by IBM personnel to estimate performance of System 370 computer systems using the VM/370 operating system. It is principally intended for estimating performance of existing VM/370 installations which are being reconfigured and of new VM/370 installations.

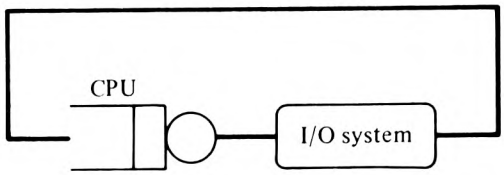



Figure 9.23

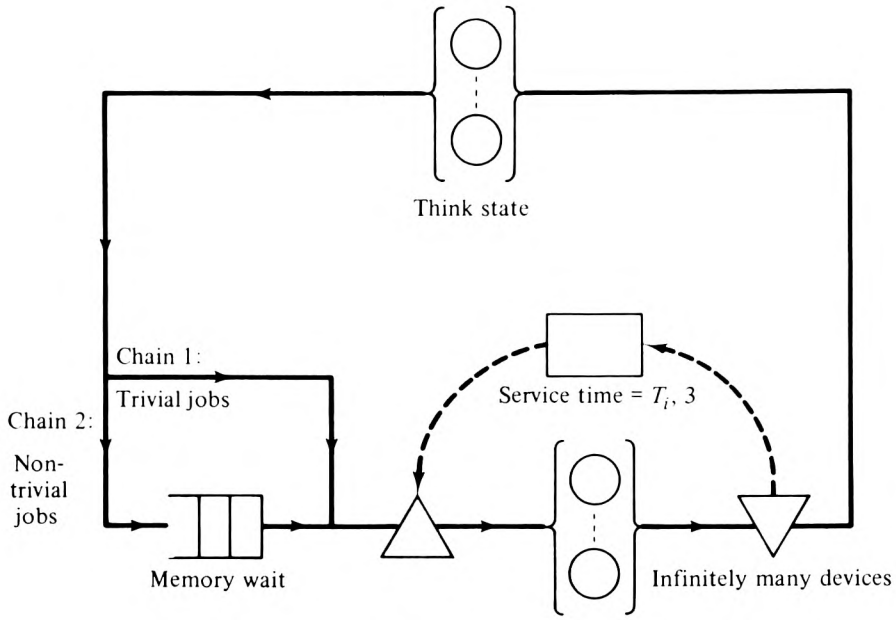



Figure 9.24

We are principally interested here in the model portion of the Predictor, but we should point out that it includes facilities for determining the model parameters. The VM/370 operating system includes its own software monitor which can be selectively enabled. The Predictor includes a Fortran program for producing the model inputs from the software monitor output. (When estimating performance of a new installation, the model input must be provided from other sources.)

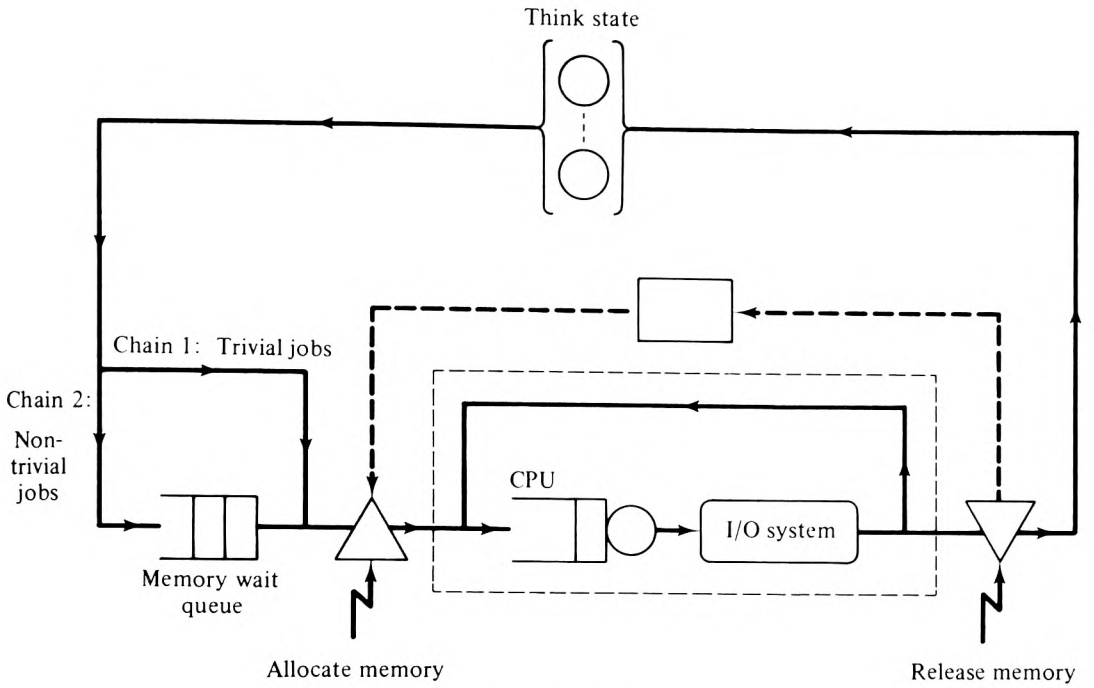



Figure 9.25

A principal difference between the model in the Predictor and almost all of the other models considered in this chapter is that it considers jobs in the system to be heterogeneous. Both the VM/370 schedulers and the model classify jobs as "trivial" or "non-trivial" according to their demonstrated resource requirements. In addition, the model allows arbitrary partitioning of jobs into separate chains. Our discussion will assume that only two chains, trivial and non-trivial, are being considered.

The input to the model is partitioned into categories, the system description and the workload description. The system description gives the CPU model, the main storage size, the number and types of channels and secondary storage devices and the assignment of paging and file data sets to the secondary storage devices. The workload description is given by chains; the number of users, the mean think time and the mean resource demands are given for each chain. The predictor transforms the workload description appropriately when it is based on a different system than the one specified, e.g., if different CPU's are involved, the CPU time is multiplied by the ratio of CPU speeds. Details of the workload description and transformation are found in BARD77a.

The model consists of a three level hierarchy: the I/O subsystem model, the active (i.e., multiprogrammed) set model and the transaction flow model (i.e., the entire system). The I/O subsystem model is an open queueing network model which represents details of the channel-disk architecture such as those discussed in Section 9.3.3 in regard to the ALS model. We will not discuss this model but refer the interested reader to WILH77 for discussion of this type of model. In addition to the system and workload description described above, the I/O subsystem model requires estimates of the arrival rate of requests for each data set. The output of the I/O subsystem model consists of the mean response times by data set. The active set model is depicted in Figure 9.23. It is a cyclic queue model with an infinite server queue for the I/O system; the response times from the I/O subsystem model are used as the service times at the infinite server queue. (The dynamic behavior of the system might be captured more accurately if the I/O subsystem model and its interface with the active set model considered state dependent behavior instead of mean values. However, the approach used is more convenient and *seems to be sufficiently accurate in practice.*) The active set model satisfies product form and can be easily solved by the methods of Chapter 5. The degree of multiprogramming (by chain) is required as input to this model; the model output is the mean memory residence time (by chain). The transaction flow model (Figure 9.24) uses the mean memory residence times from the active model as service times in an infinite server queue. Rather than go to the effort of attempting an exact solution of the transaction flow model, a solution is obtained by the mean value arguments given below. The final solution of the hierarchy of models is interpreted as a solution of the overall model shown in Figure 9.25.

Hopefully, the reader is wondering "Which came first, the chicken or the egg?" in regard to the model inputs and outputs for the three models of the hierarchy. The I/O system model requires arrival rates at the data sets which should be obtained from the transaction flow model, the active set model requires the response times from the I/O system model and the transaction flow model requires residence times from the active set model. The answer is that we start with a *guess* for the residence times. Then we can solve the transaction flow model, the I/O system model and the active set model, in that order. The residence times from the solution of the active set model will usually be different from our initial guess. We can use the new value and repeat the cycle. We continue to repeat the cycle until there is little change in the residence times. There is no guarantee that such convergence will be achieved, and there is no guarantee that the results will be a correct solution for the model(s), but in practice convergence usually is achieved within five to ten cycles and the model results agree well with measurement values [BARD78].

| CPU Model | Logged Users | CPU Util. Meas. | CPU Util. Pred. | Trivial Resp. (seconds) Meas. | Trivial Resp. (seconds) Pred. | Non-trivial Resp. (seconds) Meas. | Non-trivial Resp. (seconds) Pred. |
|---|---|---|---|---|---|---|---|
| 135 | 4 | 17.1 | 17.2 | 0.70 | 1.00 | 19.0 | 24.1 |
| 145 | 8 | 84.0 | 84.8 | 0.25 | 0.24 | 3.9 | 3.1 |
| 145 | 15 | 96.6 | 97.4 | 0.51 | 0.44 | 26.6 | 19.7 |
| 155-II | 20 | 22.2 | 22.2 | 0.05 | 0.06 | 1.1 | 1.1 |
| 155-II | 23 | 36.9 | 35.7 | 0.08 | 0.11 | 2.8 | 3.6 |
| 158 | 37 | 59.2 | 55.4 | 0.21 | 0.26 | 21.8 | 18.4 |
| 158 | 46 | 70.3 | 69.0 | 0.14 | 0.12 | 2.5 | 1.6 |
| 158 | 24 | 68.8 | 71.3 | 0.07 | 0.09 | 6.1 | 5.3 |
| 168 | 72 | 36.0 | 35.2 | 0.13 | 0.11 | 7.8 | 6.7 |
| 168 | 117 | 96.3 | 99.7 | *0.46 | 0.41 | 8.0 | 9.7 |
| | | | | *0.48 | 0.53 | 13.9 | 10.7 |
| | | | | *0.55 | 0.58 | 19.2 | 19.2 |
| | | | | *0.83 | 0.73 | 28.3 | 26.0 |

*These response times refer to four separate user classes. Classification was based on ratio of trivial to nontrivial transaction counts.

Figure 9.26

Let us now consider the solution of the transaction flow model (Figure 9.24). A user is assumed to make transitions between states in a cyclic fashion: THINK - MEMORY WAIT - ACTIVE - THINK - ..., or equivalently, 1, 2, 3, 1, ... . Let us designate the trivial jobs as chain 1 jobs and the non-trivial jobs as chain 2 jobs. A trivial job is assumed to be immediately admitted to service, i.e., it is assumed to spend no time in the memory-wait state. Thus its state transitions are THINK - ACTIVE - THINK, ... or 1,2,1, ... . A non-trivial job may have to spend time in the memory-wait state.

We are given the numbers of jobs $N_i$ in chain $i$, the mean think times $T_{i,1}$ (i.e., time in state 1 for chain $i$ jobs), the mean active times (residence times) $T_{i,3}$, the mean main memory requirements (resident set sizes) $W_i$ and the total main memory available $S$. We are required to compute the throughputs of each chain (so that we can determine the arrival rates at the data sets for the I/O subsystem model) and the mean number of jobs of each chain in the active state (which is used as the degree of multiprogramming in the active set model. We will determine $N_{i,k}$ and $T_{i,k}$, the mean number of jobs of each chain in each state and the mean time spent by jobs of each chain in each state, respectively, for chain $i = 1, 2$ and state $k = 1, 2, 3$. $N_{i,3}$ will be the input to the active set model. The input to the I/O subsystem model can be determined from $N_{i,3}$ and $T_{i,3}$ using Little's rule. We can also obtain desired performance measures, e.g., utilizations and response times, by appropriate use of these values and the model inputs.

The probability of $p_{i,k}$ that a random chain $i$ job ($i = 1, 2$ for trivial, non-trivial) is in state $k$ ($k = 1, 2, 3$ for think, memory-wait, and active) is proportional to the time spent by that job in that state, i.e.,

$$p_{i,k} = \frac{T_{i,k}}{T_{i,1} + T_{i,2} + T_{i,3}} \quad \text{for all } i,k. \tag{9.8}$$

Hence the mean number of chain i jobs in state k is

$$N_{i,k} = N_i p_{i,k} = N_i \frac{T_{i,k}}{T_{i,1} + T_{i,2} + T_{i,3}}. \tag{9.9}$$

Note that we are given $N_i, T_{i,1}$ and $T_{i,3}$. Further, we have assumed that $T_{1,2} = 0$. Thus we only need to obtain $T_{2,2}$.

The mean amount of main storage used by chain $i$ jobs $S_i$, is simply given by

$$S_i = N_{i,3} W_i. \tag{9.10}$$

Thus we can immediately obtain $S_1$ from equations (9.9) and (9.10). If $S_1$ is less than S then there is enough main storage to accommodate trivial jobs, *on the average*, and our assumption that $T_{1,2} = 0$ is reasonable; if $S_1$ is greater than S the system is saturated by trivial jobs and our solution terminates unsuccessfully. These equations and arguments are not necessarily correct because they are based on mean values rather than distributions. (Recall that the mean value analysis of Chapter 5 rests on formal derivations and the underlying Markov processes.) Strictly speaking, a trivial job will experience memory wait if all of main storage is filled with other trivial jobs; the only case where trivial jobs *never* experience memory wait is when $S$ is at least $N_1 W_1$. However, our goal here is not formal analysis but effective performance prediction. The methods may be heuristic, but they are reasonable and are supported by extensive empirical results.

To obtain $T_{2,2}$, let us first assume it is zero. Then we can obtain $S_2$ from equations (9.9) and (9.10). If $S_2$ is less than $S - S_1$, then we conclude that $T_{2,2} = 0$. Otherwise, we assume that storage is saturated, i.e., $S_2 = S - S_1$, and we have

$$\begin{aligned} S - S_1 \\ = S_2 = W_2 N_{2,3} \\ = W_2 N_2 \frac{T_{2,3}}{T_{2,1}+T_{2,2}+T_{2,3}} \end{aligned} \tag{9.11}$$

or equivalently

$$T_{2,2} = \frac{W_2 N_2 T_{2,3}}{S - S_1} - T_{2,1} - T_{2,3}. \tag{9.12}$$

Having determined $T_{2,2}$, we are through with the solution of the transaction flow model. We can proceed to the I/O subsystem model and then the active set model. Then, if necessary, we repeat the cycle through the models.

The Predictor is widely used within IBM (it is not expected to become available outside of IBM) and has been validated with measurements from a range of systems and workloads. See Figure 9.26 for some sample results with live workloads. The Predictor is a satisfying example of a fairly simple queueing network model being used effectively to estimate the performance of complex computer systems.

## 9.6 COMPUTER COMMUNICATION MODELS

When several geographically separate computers are connected in a network, or when terminals are not located near their computer system, a substantial portion of response times and a substantial portion of system cost will be due to communication between these entities. Queueing network models have played an important role in estimating the performance of computer and communication networks. We provide here a brief summary of some of the results in one of the early papers about the ARPANET [KLEI70].

For the purposes of that paper, the ARPANET consists of nineteen "host" computers located at universities and research centers through the United States. (This description was out of date even at the time that paper was written, as acknowledged by the author.) Associated with each host is an Interface Message Processor (IMP), a minicomputer which handles all network dealings for the host. The IMP's are connected by leased telephone lines with bandwidths of 50K bits per second. (A few lines are of different bandwidths.) However, there is not a direct connection between every pair

of IMP's. This would be rather expensive and unnecessary. A message sent from one host to another will typically pass through several intermediate IMP's. So that the communication lines will not be monopolized by large messages and for other reasons messages are divided into *packets* with a maximum length of 1000 bits. Different packets of a message may take different paths to the same destination, thus the term "packet switching network." The network is also referred to as "store-and-forward" because the IMP's store copies of the packets they forward until they receive *acknowledgement* messages saying that the packets have been successfully received by the subsequent IMP's of the packets' paths.

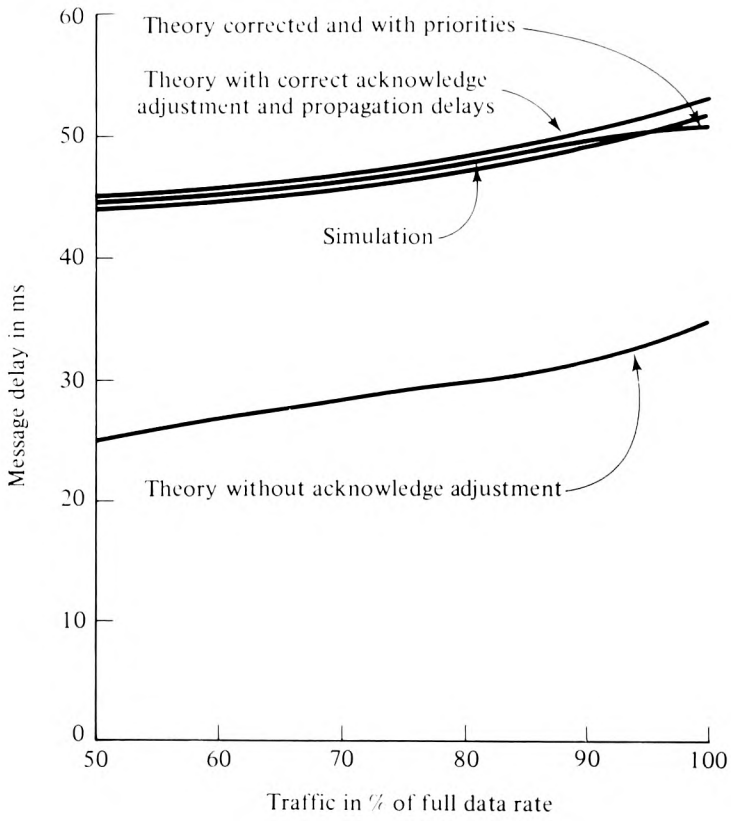



Figure 9.27

The response time for a packet will be the sum of several delays in transmission from IMP to IMP. The delays in transmission will typically consist of (1) an IMP processing time of roughly 1 ms., (2) a waiting time until the communication line is available, (3) a propagation delay for the first bit to travel from the sender to receiver, and (4) a service time depending on the packet length and line capacity.

An open queueing network is a reasonable model of the network because the number of packets in the network may be quite large. It is fairly reasonable to assume that messages arrive from the hosts in a Poisson manner. However, there are several problems if we wish to use a product form network as our model. The packet lengths (and thus service times) have a more regular distribution than the exponential, while scheduling is typically first come first served. The length of a packet remains constant as it goes from IMP to IMP, and thus a packet's successive service times are not at all independent. Choice of routing paths for a packet may depend on congestion of the possible paths. None of these problems seem to be troublesome in this case. Kleinrock assumes exponential packet lengths, that routing may be specified probabilistically and that an *independence assumption* holds, i.e., successive service times for a packet are independent. With these assumptions our model is simply a network of FCFS queues with exponential service times and the results of Chapter 4 apply directly.

The average packet length is 560 bits. For each packet there is (hopefully) an acknowledgement with average length 140 bits. So the mean service time is

$$\frac{\frac{560+140}{2}}{50000} = .007 \text{ seconds} = 7 \text{ ms.}$$

We can determine $R$ and $r_{(m)}$, $m = 1,...,M$ by measuring the traffic flow in the network, including acknowledgements. Then $U_{(m)} = 7Rr_{(m)}$ (assuming $R$ is expressed in traffic per ms.) and

$$Q_{(m)} = \frac{7U_{(m)}}{1-U_{(m)}} + 7 \text{ ms.}$$

Then the mean response time is

$$\sum_{m=1}^{M} r_{(m)} Q_{(m)}. \tag{9.13}$$

This analysis, unfortunately, is insufficient in that it severely underestimates the response time as estimated by a detailed simulation. Figure 9.27 shows response time estimates for various fractions of $R$ up to 100%. The curve labelled "theory without acknowledgement adjustment" is obtained from equation (9.13).

There are several things we can do to improve the estimate, though they will violate product form conditions so our analysis is not rigorously defensible, particularly in regard to assumed independence of the queues.

First, though the mean service time for all traffic is 7 ms., the mean service time for packets is 11.2 ms., so we should increase $Q_{(m)}$ by 4.2 ms. Second, we have ignored the propagation delay and should add this to $Q_{(m)}$. The propagation delay will depend on the distance traveled. Third, we need to add the IMP processing time for every time the packet is transmitted, plus one more for the destination. With all of these changes our estimate of mean response time for a packet is

$$\sum_{m=1}^{M} r_{(m)}(Q_{(m)} + 4.2 + D_{(m)} + 1) + 1, \tag{9.14}$$

where $D_{(m)}$ is the propagation delay on line $m$. This estimate agrees very well with the simulation; it is labelled "theory with correct acknowledge adjustment and propagation delays" in Figure 9.27. (The remaining curve is for an analysis which considers the priority given to acknowledgements.)

This model allows much room for variation to consider special characteristics of particular networks. In addition, if we wish to vary the capacities of the lines to improve the cost effectiveness of the model, it is possible to determine optimal capacity assignments given cost and/or performance constraints [KLEI70, CHAN77a]. Many other queueing network models have been proposed for computer communication systems. For further discussions see KLEI76, SAUE77c, SAUE78b, SCHW77, and WONG78b.

## 9.7 EXERCISES

9.1 Derive the values in Figure 9.4.

9.2 Using algorithm 3.2, the algorithm of exercise 3.4 and the results of Chapter 5, duplicate Figures 9.5 and 9.6.

9.3 Using algorithm 3.4, the algorithm of exercise 3.6 and the results of Chapter 5, duplicate Figures 9.11 and 9.12.

9.4 Justify equations 9.1, 9.2, and 9.3.

9.5 Determine the equation corresponding to equation 9.6 for First Fit scheduling with a bound on the number of jobs in memory.

9.6 Determine the equations corresponding to equation 9.6 for First Come First Served scheduling.

(Exercises 9.5 and 9.6 assume that the previously stated assumptions of BROW77 apply.)

9.7 Show that expression (9.13) gives the same response time result as the algorithm at the end of section 4.2.

# *CHAPTER 10*

# MANAGEMENT OF MODELING PROJECTS

This chapter is based upon our experience working and consulting with industry and government on performance problems. We address typical questions that arise in the practice of performance modeling such as (1) How does one manage a performance prediction project? (2) What methods should be used for performance predictions? Our viewpoint is pragmatic. We begin with the fundamental premise that unless the money earned from a performance group exceeds the money spent on it, there is no point in having the group. The acid test of an investment in a performance group is the same as in any other investment: is the cost/benefit ratio satisfactory?

## 10.1 THE MANAGER'S VIEWPOINT

Performance groups are used in the phases of system evolution referred to in chapter 1: design and development, configuration and tuning. In system design and development we are concerned with new systems (as opposed to configuring existing systems to meet specific needs). Configuration is concerned with selection of hardware and software components from the sets of available components. Tuning usually consists of making relatively minor modifications (e.g., changing scheduling policies) to an existing system to improve performance, perhaps to dramatically improve performance. The objectives to be set for a performance group depend on whether its primary function is to participate in design, development, configuration or tuning.

### 10.1.1 Consequences of Decisions

A wrong recommendation made by a performance group participating in a system design may have disastrous consequences, affecting a large group of potential users as well as the suppliers of system components. An incorrect decision in system configuration may be severe, but is not likely to be as catastrophic as a wrong decision in design because fewer users will be affected. System tuning decisions can be altered quite easily and hence the cost associated with wrong tuning decisions is relatively small. The importance of a decision (measured in money lost in making a wrong one) plays a key role in determining the modeling technology to be used.

### 10.1.2 Frequency of Use of Technique

There are many more cases of system configuration than there are of systems design and development because there are normally several installations of each system design. A given configuration may be tuned several times in its lifetime. Vendors and companies specializing in performance find it profitable to develop configuration and tuning tools because there are a large number of potential users/customers for these tools. Individual customers usually find it more cost-effective to purchase configuration and tuning tools than to develop their own.

An organization which does not develop computing systems will typically obtain configuration and tuning tools from external sources. Performance groups in such organizations must understand the tools they obtain and be able to validate predictions made by using these tools against measurements made in their own installations. The management of such performance groups is very different from the management of performance groups in organizations which do develop computing systems. (These latter performance groups will likely develop their own performance tools.)

### 10.1.3 Number of Alternatives

The space of design alternatives is vast. Performance models used in systems design must be flexible since very different aspects of the system may have to be modeled in the design and development stages. The number of alternatives that needs to be considered in system configuration is considerably smaller. The alternatives in system tuning are even more well-defined and limited. The modeling techniques to be used depend heavily on the size of the parameter space.

### 10.1.4 Validation

The credibility of a tuning model is usually demonstrated by showing that it has successfully predicted the effects of previous modifications. A user's faith in a good tuning model is bound to increase if it successfully predicts the behavior of his or her modifications. Similarly, the validity of a configuration model may be demonstrated by comparison with measurements obtained from previous installations.

However, it is difficult to have faith in a model during design and development because it may be some time after the modeling effort before the system is operational. It is difficult to gain credibility for a design or development *model* by showing that the same *technique* has been applied on earlier systems, because it is not self-evident that the same technique will work satisfactorily for a radically different system. Even if a

designer/developer has faith in a modeling *technique*, he or she may not have faith in a particular *model* because some aspects of the system are ignored. For these reasons it is *critical* that the designers/developers have faith in the modeling *team*. Further, the modeling team must demonstrate that all important aspects of the system are considered in the model.

### 10.1.5 Workload Definition

The workload definition provides the input to the model and thus determines its output. The workload is relatively well defined in tuning models and relatively poorly defined in designing and developing models. It is difficult, but necessary, for designers and modelers to agree on a set of workload scenarios. A clear definition of workload is necessary for a successful modeling project.

### 10.1.6 Summary of Performance Group Objectives

#### 10.1.6.1 Design and development groups.

(1) Demonstrate the credibility of the performance *group* by successfully predicting the performance of designs.
(2) Work closely with designers/developers to understand the intricacies of each design. Understanding and explaining the interactions between different aspects of a system is the most important contribution a performance group can make.
(3) Make the best possible predictions, given the limited data on hand, and justify the predictions. A prediction that is to have impact on a design must be made early in the design/development cycle. Such predictions are necessarily made with incomplete data. It is tempting for a performance group to protect itself by refusing to make predictions until measurements can be made. Indeed, performance teams can (and often do) play totally safe by restricting their activities to measurement. Predictions may be wrong. However, in the long run, good performance groups can save their organizations a great deal of money by recognizing poor design decisions early in the design/development cycle.
(4) Set up performance goals for each system. Each system, however novel, must meet certain performance goals to be useful. It is the duty of the performance group to work closely with marketing groups (or corresponding groups in non-profit organizations) in determining suitable ranges for key performance measures. Far too often, systems are implemented without stated performance goals. It is safer to abdicate the responsibility of stating performance goals until the system is operational. However, avoiding the statement of performance requirements is costly because poor

designs can be allowed to develop into poor systems, precisely because goals are nebulous.

(5) Make measurements on prototypes and early systems. Validate models as much as possible. If necessary, explain to the designers/developers why the system does not behave in the manner that they expected it to behave.

(6) Use the experience from the design/development models in helping to build configuration models.

**10.1.6.2 Configuration: Vendor's Performance Group.**

(1) Determine the range of configuration alternatives that customers want considered.

(2) Validate the model by comparison with measurements for all (or several) points in the range of alternatives.

(3) Suggest guidelines for choosing the best configuration for a given customer.

**10.1.6.3 Configuration: Customer's Performance Groups.**

(1) Understand the modeling techniques used by the vendor and capacity planning consultants. Far too often, a customer treats a capacity planning program offered by a consulting firm as a black box encapsulating magic or incomprehensible mathematics. Modeling techniques are generally very simple. Certainly, every reader of this book should be able to understand the technology underlying capacity planning models. It is necessary to understand the programs one is buying because one must (a) choose between competing capacity planning programs and (b) know the fallibility of the programs one is purchasing.

(2) Understand measurement tools, and the data they report.

(3) Quantify anticipated workloads by extrapolating from current measurements.

(4) Use configuration tools to predict the performance of proposed configurations with anticipated workloads.

**10.1.6.4 Tuning.** The goals for groups working on tuning are similar to the goals of groups on configuration, except that (1) tuning efforts place greater emphasis on measurement and (2) tuning "models" are likely to be guidelines or decision rules. The analyst should have an intuitive understanding of these rules.

## 10.2 EVALUATION OF MODELING TECHNOLOGY

### 10.2.1 Measurement

Measurement is conceptually simple. To determine performance, measure the behavior of a system running a representative workload. To determine how changes to the system will impact performance, implement the changes and measure the changed system running the representative workload. Measurement is a necessary aspect of modeling techniques. However, reliance on measurement exclusively is short-sighted because (1) it may not be possible to implement a proposed system and (2) it may be prohibitively expensive to develop a detailed synthetic workload to represent an anticipated workload. Measurement is attractive because it deals with real, tangible things. The drawbacks to measurement are that (1) predictions based on this approach alone may come too late to be of any use and (2) the approach may be prohibitively expensive.

### 10.2.2 Simulation

Simulation is a flexible and powerful approach. A simulation model can be arbitrarily detailed, representing all system characteristics. Abstract models are made of the anticipated workload and of the proposed system, and measurements are taken from the simulation program. Simulation has advantages of security in the applicability of the technique, of comprehensibility, of comprehensiveness (particularly with respect to time dependent behavior) and of application to software development. Simulation has disadvantages of potentially containing programming errors, of time and cost for simulation program development, of unavailable input data and of computational expense.

**Security.** A performance group can embark on a major project knowing that the simulation approach will work provided there is sufficient time to write and run the simulation. The performance team can be secure in their choice of method since the limitations are known, at least qualitatively, in advance. The same cannot be said about mathematically solved queueing models.

**Comprehensibility.** Most computer professionals have some understanding of simulation; the concepts are simple. Most computer professionals do not understand queueing-theoretic models and as a consequence may feel threatened by proponents of such models. (This is partly a result of the jargon and notation often associated with queueing theory.) Readers of this book understand queueing theory, but many computer professionals are suspicious of performance methods other than measurement and simulation. To convince a manager or designer that the results of a model should be

taken seriously one must first convince him or her that the modeling technique is reasonable.

**Comprehensiveness and time dependent behavior.** Besides the obvious generality of simulation, there is the ability to accurately capture transient behavior. Decisions are made in the design of some systems (e.g., transaction oriented data base systems) to allow certain kinds of deadlocks to occur in the expectation that these kinds of deadlocks will not occur frequently (and the knowledge that recovery is not prohibitively expensive). Simulation can be used to determine the frequency of deadlocks, and may uncover unanticipated deadlocks. Queueing theoretic models are very limited in applicability to transient behavior.

**Application to software development.** Algorithms and programs used in a simulation model may be used, in some cases, in the actual system with only minor modification. Such transfer from model to system is facilitated by use of a common language.

**Correctness.** Simulation models, like any complex program, are likely to contain errors. Even if a model is conceptually correct (i.e., specified correctly) it's simulation implementation may have errors which cause very poor performance estimates. This is a severe problem.

**Development time and expense.** Simulation models take a significant amount of time to develop, especially when constructed in general purpose (non-simulation) programming languages. Detailed simulation development consumes a critical resource in system development: programmer time.

**Availability of input data.** A detailed simulator requires detailed input data, and such detailed data may not be available. A grievous mistake made by some purchasers of simulation models is to insist on an excruciating level of detail, in the hope that the more detailed the model the better its predictions. The purchaser may find too late that he or she only has guesses for the required input data. There is no point in detail if there is not a corresponding level of *credible* detail in the input data.

**Computational expense.** Credible simulation results may depend on fairly long simulation runs. For this reason, simulation may be infeasible for studying the sensitivity of a system to a number of parameters.

### 10.2.3 An Engineering Approach to Queueing Theory

This approach is based on the tenet that all significant aspects of a system must be represented, regardless of mathematical tractability. Some submodels may be solved by numerical methods, others by simulation and

others by heuristic approximations, with the overall result being a heuristic approximation to the solution of a detailed model. The approach attempts to develop prediction methods which are "reasonable" and appeal to the intuition of its users.

**Advantages.** This approach is flexible in that almost any system characteristic can be represented (by use of simulation for a submodel, if necessary). The computational time necessary may be dramatically less than that for a simulation or an exact solution. The memory required may be dramatically smaller, as well.

**Disadvantages.** This approach may produce poor predictions either because the conceptual model is inaccurate or because the intuitively plausible solution fails. It is very difficult to estimate the error or to defend the approach from attacks on its credibility. The time to develop approximate solutions and to empirically validate them (e.g., by comparison with simulation) may be prohibitive. One is more likely to be able to find staff for a simulation model, i.e., programmers, than for a heuristically solved model.

### 10.2.4 The Formal Queueing Theory Approach

Formal, numerically solved models may not suffer the credibility problems of heuristic approaches if system characteristics are adequately represented. However, a numerical solution is likely to be intractable if the model does not satisfy product form.

**Advantages.** The algorithms for product form models are very simple and have been implemented for pocket calculators as well as larger machines. Many of the programs for product form models have been used extensively, and experience has established the credibility of the programs. Thus the credibility of the model can only be attacked by faulting the fidelity of the model itself. The computational requirements for solution of product form networks are negligible unless there are many closed chains. Thus one can interactively evaluate a parameter space of models.

**Disadvantages.** The principal disadvantage is that one may have to ignore important system characteristics to obtain a mathematically tractable model. If the model does not have a product form solution, even though its solution may be computationally tractable the implementation cost of programming the solution may be significant. Finally, there is a problem of skepticism of those unfamiliar with queueing models. The only practical way to establish the credibility of a queueing model before a skeptical computer professional is empirical validation.

### 10.2.5 The Method of Choice

The best method to use depends upon the stage of evolution of the system being analyzed (i.e., design/development, configuration/tuning) and the affiliation of the performance group. We consider the important classifications.

**Design/development.** A design analyst should be capable of carrying out "back of the envelope" calculations to determine whether a design in its early stages is reasonable and worthy of further consideration. Since there are interactive performance modeling packages on the market, it is preferable to do "front of the terminal" calculations instead of "back of the envelope" calculations, both for sake of analyst efficiency and modeling accuracy. If an analyst does not have access to a performance analysis package, he or she can implement the algorithms we have given (on a machine as small as a programmable calculator for some of the algorithms). The design analyst should understand the queueing network models and the approximation techniques described in this book so that these techniques may be used to cull potentially good design ideas from the definitely bad ones. The design analyst does not need much mathematics but must *understand the models,* i.e., what is represented in detail, what is being simply represented and what is being ignored.

Some designers say that they would rather rely on intuition and trivial "paper and pencil" calculations than on modeling. These people do not realize that even trivial paper and pencil calculations deal with models, albeit simplistic ones. Since queueing network model technology has progressed to the point of providing packages which allow users to (1) define sophisticated models interactively, (2) solve the models in real time and (3) obtain reports of model estimates, there is no longer any excuse for designers to avoid queueing network models.

Design analysts must also develop (relatively detailed) simulation models to help detect performance problems. Though the cost of developing simulations may be significant, the cost of redesign will likely be even higher. In our experience, designers are more likely to be convinced by estimates from simulations than from queueing theoretic results. Further, issues of vital interest to designers (such as frequency of deadlock) may not be feasibly considered without simulation.

**Configuration/tuning - vendor/consultant.** The performance team for a vendor or performance consulting firm should be capable of developing relatively detailed models to be configured or tuned. Such detail will usually force the use of approximate solutions or simulation or both in a hybrid solution. The team should be capable of carrying out intensive measure-

ments on the system being marketed and the models *must be validated with measurements*. Validation is the most important aspect of configuration and tuning models.

Configuration and tuning models should (ideally) run while the analyst is sitting at a terminal. The requirement for short execution times will usually preclude complex models and detailed simulations. The designer of a configuration model does not (normally) need to know more mathematics than there is in this book. However, designers of such models should have a strong grasp of how approximate models may be constructed from simple submodels.

The natural questions asked by a buyer of a configuration/tuning model are "What am I paying for? If the mathematics is simple and the models are not complex then the cost of the modeling package should be small. Why shouldn't I develop my own model for less money than the vendor or consultant demands?" The answers are "Validation! Validation!" A vendor or performance consultant has much better opportunity for obtaining measurements from a large number of systems and carrying out intensive validation. Gathering and reducing the data to drive the model and human engineering of the modeling software is also a sizable cost.

**Configuration/tuning - customer.** The performance group of a computer system user has less stringent requirements than groups developing models, unless there are no satisfactory models for the user's system. The primary responsibility of a computer system user's group is to understand the system itself and the performance tools marketed by the system vendor and performance consultants. There are many different performance tools on the market. Most of these tools are concerned with measurement though some are modeling tools. It is important to have some understanding of how the tools work so that the analyst will place the correct amount of credibility (too much or too little can be dangerous) on the reports generated by the tools. These reports predict performance measures but are not directive; the reports do not suggest changes to the system. Hence the analyst must understand the system and the reports thoroughly so that correct changes can be made.

## 10.3 ORGANIZATIONAL STRUCTURE

What effect does the organizational relationship of the performance group with other groups have on the quality of performance evaluation? We address this question, first considering design and development and then configuration and tuning.

### 10.3.1 Design and Development

Peformance analysts are used in the design of computing systems in two modes:

1. Performance analysts are members of the design team. The analysts report to the same managers as other members of the design team. The analysts are responsible for helping designers in evaluating design strategies.
2. A single performance group services the entire organization or an entire division. Analysts do not report to the same managers as designers. There are separate chains of command for analysts and designers which meet at relatively high levels of management.

There are advantages with each mode of organization. We consider the advantages and disadvantages of the first mode (analysts are part of the design team) in comparison with the second mode (distinct performance group).

**10.3.1.1 Advantages.** Designers have to choose between alternate designs. They will do so one way or another, usually by using intuition, past experience or models. The designer must make choices quickly even though this results in wrong choices. In many cases the designer will be able to make choices without the aid of an analyst; however, when the designer needs an analyst he or she usually needs a quick analysis and a (subjectively) convincing argument that the analysis is correct. The designer does not require the analyst to use the most sophisticated modeling techniques available.

An analyst who is part of a design team is more likely to be responsive to a designer's needs than an analyst who belongs to a totally different group. Being responsive to the needs of a design group implies coming up with quick analyses even though the analyses may not always be correct. It is tempting for an analyst to be "absolutely sure" that his or her analysis is correct before reporting it. Many analysts are unwilling to use their best judgement; they would rather not give any analysis at all. Certainty can only be achieved by extensive measurement and designers can not afford to wait for measurement.

The business of making predictions is inherently risky. The analyst is hired to make *better* predictions in a manner which is timely and responsive to the needs of the designers. If an analyst has a different reporting chain than the design team, he or she is more likely to avoid risks by avoiding timely predictions because (1) Performance groups have a time honored excuse: "There isn't enough data to carry out this analysis." Note that

design groups are not permitted to make this same excuse! (2) Performance groups are (unfortunately) evaluated by the accuracy of their predictions rather than by the successes of the entire design effort. Hence, they may have less commitment than they ought to in helping make design choices.

An analyst's most important role is to gather all the facts relevant to a design and to relate different aspects of a design (in a quantitative manner where possible). The model serves as a framework and a discipline for gathering facts. *The mathematics of the model is much less important than the discpline engendered by the modeling process.* It is less important whether a model is based on numerical or simulation solution than that modeling be carried out. Once the facts are gathered together, the consequences of the facts are often intuitively obvious, i.e., the facts speak for themselves. The discipline of modeling helps one arrange facts.

An analyst is likely to (1) find it easier to gather the relevant facts and (2) have greater success in explaing the results of the modeling effort if he or she is a member of the design team. When analysts and designers belong to different groups it is likely that an adversary relationship will develop between the two groups. The design team will very likely consider an analyst to be a person who evaluates and passes judgement on their design. Designers are less likely to spend time interacting with the analyst because they see no direct benefit to themselves. As one design group manager told us: "As far as the design groups are concerned the modeling project is all 'give'; we get nothing in return." It is equally dangerous for the analyst to think of himself or herself as a person who "certifies" a design. Ideally, such a counter-productive adversary relationship can be avoided if the analyst belongs to the design group.

**10.3.1.2 Disadvantages.** Given the time pressures on design teams it is not surprising that capable analysts on design teams are often subborned into becoming full time designers and giving up their role of analytic service to other designers. Recall that the primary role of an analyst is to gather and piece together facts from different designers. Thus an analyst should have a more comprehensive view of the overall design than most designers. It is tempting for the group to use the analysts knowledge in a design capacity.

It is helpful to centralize the experience gained from analyzing several designs into a single performance analysis group. It is easier for a single performance group to develop and keep abreast of the most advanced techniques than for several analysts dispersed among several groups. In-house training of analysts is also easier when there is a centralized performance analysis group.

### 10.3.2 Configuration and Tuning

Performance groups dealing with configuration and tuning may not be organizationally tied to systems engineering and sales groups. However, even though the performance tools are *developed* in other organizations it is necessary that there be significant expertise in the use of these tools within systems engineering and sales groups. In practice, performance analysts usually report to the same management as other information systems professionals except in very large organizations where performance groups may be centralized. The tradeoffs between organizational structures are much the same as in the design case.

## 10.4 IN-HOUSE TRAINING

At present, performance analysts are in short supply. Thus organizations will often have to train analysts in order to obtain them. It is not difficult to train analysts. A sophisticated mathematical background is usually not necessary. The critical prerequisite for performance analysis is an understanding of systems. Thus systems programming is an ideal background; an experienced systems programmer can be converted into a performance analyst relatively easily. Unfortunately, systems programmers are also in short supply!

The first thing to teach a budding analyst is the detailed organization of system hardware and software. The second thing an analyst ought to be taught is measurement. Third, the analyst should learn simulation; a programmer can easily learn to write simulation programs. Currently, training for most analysts stops at this stage. An analyst can do an excellent job on configuration and tuning problems with a thorough knowledge of systems and measurement tools and a reasonable knowledge of simulation. For design and development problems and for maximum efficiency in configuration and tuning problems, an understanding of the fundamentals of queueing models (as provided in this book) is necessary. Only a small minority of analysts will need more sophisticated modeling techniques.

The best training in modeling is experience. After an intensive (perhaps two week) course based on this book, supplemented with additional material on measurement, an analyst should be given on-the-job training. Analysts learn best when faced with real problems and real deadlines.

# *BIBLIOGRAPHY*


BARD77a Y. Bard, "A Characterization of VM/370 Workloads," *Modeling and Performance Evaluation of Computer Systems,* H. Beilner and E. Gelenbe (Eds.), North-Holland, Amsterdam (1977) pp. 35-56.

BARD77b Y. Bard, "The Modeling of Some Scheduling Strategies for an Interactive Computer System," in *Computer Performance,* K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 113-138.

BARD78a Y. Bard, "The VM/370 Performance Predictor," *Computing Surveys 10,* 3 (Sept. 1978), 333-342.

BARD78b Y. Bard, "Some Extensions to Multiclass Queueing," G320-2124, IBM Cambridge Scientific Center, Cambridge, Mass. (Nov. 1978). In M. Arato, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems,* North Holland, Amsterdam, pp. 51-61 (1979).

BARD80a Y. Bard and C.H. Sauer, "IBM Contributions to Performance Modeling and Simulation," IBM Research Report RC-8364, Yorktown Heights, NY (July 1980).

BARD80b Y. Bard, "A Model of Shared DASD and Multipathing," *CACM 23,* 10 (October 1980).

BASK75 F. Baskett, K.M. Chandy, R.R. Muntz, and F. Palacios-Gomez, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *JACM 22,* 2 (April 1975).

BASK76 F. Baskett and A.J. Smith, "Interference in Multi-processor Systems with Interleaved Memory," *CACM 19,* 6 (June 1976).

BELL71 C.G. Bell and A. Newell, *Computer Structures: Readings and Examples,* McGraw-Hill (1971).

BHAN73 D.P. Bhandarkar and S.H. Fuller, "A Survey of Techniques for Analyzing Memory Interference in Multiprocessor Systems," Carnegie-Mellon University Technical Report, Pittsburgh, Pa. (April 1973).

BOYS75 J.W. Boyse and D.R. Warn, "A Straightforward Model for Computer Performance Prediction," *Computing Surveys 7,* 2 (1975).

BRAN74 A. Brandwajn, "Equivalence and Decomposition Methods with Application to a Model of a Time-sharing Virtual Memory System," *Proceedings International Symposium Rocquencourt* (April 1974).

BROW75 J.C. Browne, K.M. Chandy, R.M. Brown, T.W. Keller, D.F. Towsley and C.W. Dissley, "Hierarchical Techniques for Devel-

opment of Realistic Models of Complex Computer Systems," *Proc. IEEE 63,* 6 (June 1975), 966-975.

BROW77 R.M. Brown, J.C. Browne and K.M. Chandy, "Memory Management and Response Time," *Communications of the ACM 20,* 3 pp. 153-165 (March 1977).

BURN75 G.J. Burnett and E.G. Coffman, Jr., "Analysis of Interleaved Memory Systems Using Blockage Buffers," *CACM 18,* 2 (Feb. 1975).

BUX77 W. Bux and U. Herzog, "The Phase Concept: Approximation of Measured Data and Performance Analysis," in *Computer Performance,* K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 23-28.

BUZE71 J.P. Buzen, "Queueing Network Models of Multiprogramming," Ph.D. Thesis, Harvard University, Cambridge, Mass. (1971).

BUZE73 J.P. Buzen and U.O. Gagliardi, "The Evolution of Virtual Machine Architecture," *AFIPS Conf. Proc. 42* (1973 NCC), pp. 291-301.

BUZE79 J.P. Buzen and P.J. Denning, "Operational Treatment of Queue Distributions and Mean Value Analysis," CSD-TR-309, Purdue University (August 1979).

CHAN72 K.M. Chandy, "The Analysis and Solutions for General Queueing Networks," *Proc. 6th Annual Princeton Conf. on Information Science and Systems,* (1972) pp. 224-228.

CHAN75a K.M. Chandy, U. Herzog and L.S. Woo, "Parametric Analysis of Queueing Networks," *IBM J. of Research and Development 19,* 1 pp. 43-49 (January 1975).

CHAN75b K.M. Chandy, U. Herzog and L.S. Woo, "Approximate Analysis of General Queueing Networks," *IBM J. of Research and Development 19,* 1 pp. 50-57 (January 1975).

CHAN77a K.M. Chandy, J. Hogarth and C.H. Sauer, "Selecting Capacities in Computer Communication Systems," *IEEE Trans. on Software Eng. SE-3,* 4 (July 1977) pp. 290-295.

CHAN77b K.M. Chandy, J.H. Howard and D.F. Towsley, "Product Form and Local Balance in Queueing Networks," *JACM 24,* 2 pp. 250-263 (April 1977).

CHAN78 K.M. Chandy and C.H. Sauer, "Approximate Methods for Analysis of Queueing Network Models of Computer Systems," *Computing Surveys 10,* 3 pp. 263-280 (September 1978).

CHAN79 K.M. Chandy and C.H. Sauer, "Computational Algorithms for Product Form Queueing Networks," RC-7950, IBM Research, Yorktown Heights, N.Y. (November 1979). *CACM 23,* 10 (October 1980).

CHIU75 W.W. Chiu, D. Dumont and R. Wood, "Performance Analysis of a Multiprogrammed Computer System," *IBM J. of Research and Development 19,* 3 (May 1975) pp. 263-271.

CHIU78 W.W. Chiu and W-M. Chow, A Performance Model of MVS," *IBM Systems Journal 17,* 4 (1978) pp. 444-462.

CHOW78 W-M. Chow, "The Cycle Time Distribution of Exponential Central Server Models," *AFIPS Conf. Proc. 43* (1978 NCC).

COFF78 E.G. Coffman and L. Kleinrock, "Computer Scheduling Methods and their Countermeasures," *AFIPS Conf. Proc. 32* (1968 SJCC) pp. 11-25.

COUR75 P.J. Courtois, "Decomposability, Instabilities and Saturation in Multiprogramming Systems," *Communications of the ACM 18,* 7 pp. 371-368 (July 1975).

COUR77 P.J. Courtois, *Decomposability: Queueing and Computer System Applications,* Academic Press, Inc., New York (1977).

COUR78 P.J. Courtois, "Exact Aggregation in Queueing Networks," *Proc. First Meeting AFCET-SMF,* Paris (September 1978).

COX55 D.R. Cox, "A Use of Complex Probabilities in the Theory of Stochastic Processes," *Proc. Cambridge Philos. Soc. 51,* (1955), pp. 313-319.

COX65 D.R. Cox and H.D. Miller, *The Theory of Stochastic Processes,* Wiley, New York, (1965).

CRAN74 M.A. Crane and D.L. Iglehart, "Simulating Stable Stochastic Systems II: Markov Chains," *JACM 21* (Jan. 1974) pp. 114-123.

CRAN77 M.A. Crane and A.J. Lemoine, *An Introduction to the Regenerative Method for Simulation Analysis,* Springer-Verlag, New York (1977).

DENN78 P.J. Denning and J.P. Buzen, "The Operational Analysis of Queueing Network Models," *Computing Surveys 10,* 3 (Sept. 1978) pp. 225-261.

DISN74 R.L. Disney and W.P. Cherry, "Some Topics in Queueing Network Theory," *Mathematical Methods in Queueing Theory,* A.B. Clarke (Ed.), Springer-Verlag, New York (1974).

DRAK67 A.W. Drake, *Fundamentals of Applied Probability Theory,* McGraw-Hill, New York (1967).

DRUM73 M.E. Drummond, Jr., *Evaluation and Measurement Techniques for Digital Computer Systems,* Prentice-Hall, Englewood Cliffs, N.J. (1973).

FELL68 W. Feller, *An Introduction to Probability Theory and Its Implications,* Wiley, New York (1968).

FERR78 D. Ferrari, *Computer System Performance Evaluation,* Prentice-Hall, Englewood Cliffs, N.J. (1978).

FISH73 G.S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation,* Wiley, New York (1973).

FISH78 G.S. Fishman, *Principles of Discrete Event Simulation,* Wiley, New York (1978).

FISH79 G.S. Fishman and L.R. Moore, "Estimating the Mean of a Correlated Binary Sequence," *JACM 26,* (Jan. 1979) pp. 82-94.

FOSC77 G.J. Foschini, "On Heavy Traffic Diffusion Analysis and Dynamic Routing in Packet Switched Networks," *Computer Performance,* K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 419-514.

FOST74 D.V. Foster, P.F. McGehearty, C.H. Sauer and C.N. Waggoner, "A Language for Analysis of Queueing Models," *Proc. Fifth Annual Pittsburgh Modeling and Simulation Conf.* (1974) pp. 381-386.

FRAN77 W.R. Franta and K. Maly, "An Efficient Data Structure for the Simulation Event Set," *CACM 20,* 8 (Aug. 1977) pp. 596-602.

FULL75 S.H. Fuller and F. Baskett, "An Analysis of Drum Storage Units," *JACM 22,* 1 (Jan. 1975) pp. 83-105.

FULL76 S.H. Fuller, "Price/Performance Comparison of C.mmp and the PDP/10," *Third Annual Symp. on Comp. Architecture, Computer Architecture News 4,* 4 (January 1976) pp. 195-202.

GAVE67 D.P. Gaver, "Probability Models of Multiprogramming Computer Systems," *JACM 14,* 3 (1967) pp. 423-428.

GAVE68 D.P. Gaver, "Diffusion Approximations and Models for Certain Congestion Problems," *J. Appl. Prob. 5,* (1968) pp. 607-623.

GAVE76 D.P. Gaver and G. Humfeld, "Multitype Multiprogramming: Probability Models and Numerical Procedures," *Computer Performance,* Elsevier North-Holland, New York (1976) pp. 38-43.

GELE75 E. Gelenbe, "On Approximate Computer System Models," *JACM 22,* 2 (April 1975) pp. 261-269.

GORD67 W.J. Gordon and G.F. Newell, "Closed Queueing Networks with Exponential Servers," *Operations Research 15* pp. 244-265 (1967).

HERZ75 U. Herzog, L.S. Woo and K.M. Chandy, "Solution of Queueing Problems by a Recursive Technique," *IBM J. of Research and Development 19,* 3 (May 1975) pp. 295-300.

IGLE78a D.L. Iglehart, "The Regenerative Method for Simulation Analysis," in K.M. Chandy and R.T. Yeh, editors, *Current Trends in Programming Methodology, Volume III: Software Modeling and Its Impact on Performance.* Prentice-Hall (1978).

IGLE78b D.L. Iglehart and G.S. Shedler, "Regenerative Simulation of Response Times in Networks of Queues," *JACM 25,* 3 (July 1978) pp. 449-460.

JACK63 J.R. Jackson, "Jobshop-like Queueing Systems," *Management Science 10,* pp. 131-142 (1963).

JENS74 K. Jensen and N. Wirth, *PASCAL User Manual and Report,* Springer-Verlag, New York (1974).

KELL73 T.W. Keller, *ASQ User's Manual,* TR-27, Dept. of Computer Sciences, Univ. of Texas at Austin (1973).

KELL76 T.W. Keller, "Computer Systems Models with Passive Resources," Ph.D. Thesis, Univ. of Texas at Austin (1976).

KIEN79a M.G. Kienzle and K.C. Sevcik, "A Systematical Approach to the Performance Modeling of Computer Systems," *Performance of Computer Systems,* (M. Arato, A. Butrimenko and E. Gelenbe, Editors), North-Holland (1979).

KIEN79b M.G. Kienzle and K.C. Sevcik, "Survey of Analytic Queueing Models of Computer Systems," *Conference on Simulation, Measurement and Modeling of Computer Systems,* Boulder, CO (August 1979).

KLEI70 L. Kleinrock, "Analytic and Simulation Methods in Computer Network Design" *AFIPS Conf. Proc.36* (1970 SJCC) pp. 569-579.

KLEI75 L. Kleinrock, *Queueing Systems Volume I: Theory,* Wiley, New York (1975).

KLEI76 L. Kleinrock, *Queueing Systems Volume II: Computer Applications,* Wiley, New York (1976).

KNUT69 D.E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms,* Addison-Wesley, Reading, Mass. (1969).

KOBA74 H. Kobayashi, "Application of the Diffusion Approximation to Queueing Networks I: Equilibrium Queue Distributions," *JACM 21,* 2 (April 1974) pp. 316-328.

KOBA75 H. Kobayashi and M. Reiser, "On Generalization of Job Routing Behavior in a Queueing Network Model," IBM Research Report RC-5252 (1975).

KOBA76 H. Kobayashi, "A Computational Algorithm for Queue Distributions via Polya Theory of Enumeration," RC-6154, IBM Research, Yorktown Heights, N.Y. (August 1976).

KOBA78 H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology,* Addison-Wesley, Reading, Mass. (1978).

LAM76 S.S. Lam, "Store-and-Forward Buffer Requirements in a Packet Switching Network," *IEEE Trans. Communication 24,* (April 1976), pp. 394-403.

LAM77 S.S. Lam, "Queueing Networks with Population Size Constraints," *IBM J. of Research and Development 21,* 4 (July 1977) pp. 370-378.

LAM80 S.S. Lam, "Behavior of the Normalization Constant and a Scaling Algorithm for Product Form Queueing Networks," Technical Report TR-148, Department of Computer Sciences, University of Texas at Austin (July 1980).

LAVE75 S.S. Lavenberg and D.R. Slutz, "Introduction to Regenerative Simulation," *IBM J. of Research and Development 19,* (Sept. 1975) pp. 458-463.

LAVE77 S.S. Lavenberg and C.H. Sauer, "Sequential Stopping Rules for the Regenerative Method of Simulation," *IBM J. of Research and Development 21,* (Nov. 1977) pp. 545-558.

LAZO77 E.D. Lazowska, "The Use of Percentiles in Modeling CPU Service Time Distributions," *Computer Performance,* K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 53-66.

LEAR73 G.P. Learmonth and P.A.W. Lewis, "Statistical Tests of Some Widely Used and Recently Proposed Uniform Random Number Generators," *Proc. of Computer Science and Statistics: 7th Annual Symp. on the Interface,* Iowa State Univ. (Oct. 1973) pp. 163-171.

LITT61 J.D.C. Little, "A Proof of the Queueing Formula $L = \lambda W$," *Operations Research 9,* pp. 383-387 (1961).

MACN75 E.A. MacNair and L.S. Woo, private communication, 1975.

MARI79 R.A. Marie, "An Approximate Analytical Method for General Queueing Networks," *IEEE Transactions on Software Engineering SE-5,* 5 (September 1979).

MARI80 R.A. Marie, "Calculating Equilibrium Probabilities for $\lambda(n)/C_k/1/N$ Queues," *Performance Evaluation Review 9,* 2 (Summer 1980).

MART67 J. Martin, *Design of Real-Time Computer Systems,* Prentice-Hall, Englewood Cliffs, N.J. (1967).

MART75 R.R. Martin and H.D. Frankel, "Electronic Disks in the 1980's," *Computer 8,* 2 (Feb. 1975).

MEAR79 C.E. Mear and C.H. Sauer, "A Simple and Robust Data Structure for the Simulation Event Set," IBM Research Report RC-8001, December 1979.

MOOR72 F.R. Moore, "Computational Model of a Closed Queueing Network with Exponential Servers," *IBM Journal of Research and Development 16,* 6 pp. 567-572 (June 1972).

NEUS80 D. Neuse and K.M. Chandy, "A Method for Approximate Analysis of General Queueing Networks," Technical Report, Department of Computer Sciences, University of Texas at Austin.

PRIC75 T.G. Price, "Models of Multiprogrammed Computer Systems with I/O Buffering," *Proc. Fourth Texas Conf. on Computing Systems,* (Nov. 1975).

PRIC76 T.G. Price, "A Note on the Effect of the Central Processor Service Time Distribution on Processor Utilization in Multiprogrammed Computer Systems," *JACM 23,* 2 (April 1976) pp. 342-346.

RAMA69 C.V. Ramamoorthy and M.J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," *AFIPS Conf. Proc. 35,* (1969 FJCC) pp. 1-17.

REIS74 M. Reiser and H. Kobayashi, "Accuracy of the Diffusion Approximation for Some Queueing Systems," *IBM J. of Research and Development 18,* (1974).

REIS75 M. Reiser and H. Kobayashi, "Queueing Networks with Multiple Closed Chains: Theory and Computational Algorithms," *IBM J. of Research and Development 19,* 3 (May 1975).

REIS76 M. Reiser, "Numerical Methods in Separable Queueing Networks," IBM Research Report RC-5842, Yorktown Heights, NY (February 1976).

REIS78a M. Reiser and S.S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," IBM Research Report RC-7023, Yorktown Heights, NY (March 1978). *JACM 27,* 2 (April 1980) pp. 313-322.

REIS78b M. Reiser and C.H. Sauer, "Queueing Network Models: Methods of Solution and their Program Implementation," in K.M. Chandy and R.T. Yeh, editors, *Current Trends in Programming Methodology, Volume III: Software Modeling and Its Impact on Performance.* Prentice-Hall (1978) pp. 115-167.

REIS78c M. Reiser, "A Queueing Network Analysis of Computer Communication Networks with Window Flow Control," RC-7218, IBM Research, Yorktown Heights, N.Y. (July 1978).

REIS80 M. Reiser, "Mean-Value Analysis and Convolution Method for Queue-Dependent Servers in Closed Queueing Networks," to appear as an IBM Research Report (Zurich).

REYN80 P.F. Reynolds, "Queueing Network Algorithms on Programmable Pocket Calculators," to appear as a technical report, Department of Computer Sciences, University of Texas at Austin.

ROSE78 C.A. Rose, "A Measurement Procedure for Queueing Network Models of Computer Systems," *Computing Surveys 10,* 3 (Sept. 1978) pp. 263-280.

SAUE75a C.H. Sauer, "Configuration of Computing Systems: An Approach Using Queueing Network Models," Ph.D. Thesis, Univ. of Texas at Austin (May 1975).

SAUE75b C.H. Sauer and K.M. Chandy, "Approximate Analysis of Central Server Models," *IBM J. of Research and Development 19,* 3 (May 1975) pp. 301-313.

SAUE76a C.H. Sauer, "Characterization and Simulation of Generalized Queueing Networks," RC-6057, IBM Research, Yorktown Heights, N.Y. (May 1976).

SAUE76b C.H. Sauer and K.M. Chandy, "Parametric Modeling of Multiminiprocessor Systems," RC-5978, IBM Research, Yorktown Heights, N.Y. (April 1976).

SAUE76c C.H. Sauer, L.S. Woo and W. Chang, "Hybrid Analysis/Simulation: Distributed Networks," RC-6341, IBM Research, Yorktown Heights, N.Y. (June 1976).

SAUE77a C.H. Sauer, "Confidence Intervals for Queueing Simulations of Computer Systems," RC-6669, IBM Research, Yorktown Heights, N.Y. (July 1977). *Performance Evaluation Review 8,* (Spring-Summer 1979) pp. 45-55.

SAUE77b C.H. Sauer and K.M. Chandy, "The Impact of Distributions and Disciplines on Multiple Processor Systems," RC-6621, IBM Research, Yorktown Heights, N.Y. (July 1977). *CACM 22,* (Jan. 1979) pp. 25-34.

SAUE77c C.H. Sauer and E.A. MacNair, "Computer/Communication System Modeling with Extended Queueing Networks," RC-6654, IBM Research, Yorktown Heights, N.Y. (July 1977).

SAUE78a C.H. Sauer and E.A. MacNair, "Queueing Network Software for Systems Modeling," RC-7143, IBM Research, Yorktown Heights, N.Y. (May 1978). *Software-Practice and Experience 9,* 5 (May 1979).

SAUE78b C.H. Sauer, "Passive Queue Models of Computer Networks," *Computer Networking Symp.,* Gaithersburg, Maryland (December 1978).

SAUE79a C.H. Sauer, "Some Results on Queue Lengths in Queueing Networks Solved by Aggregation," RC-7607, IBM Research, Yorktown Heights, N.Y. (May 1979).

SAUE79b C.H. Sauer and K.M. Chandy, "Approximate Solution of Queueing Models of Computer Systems," RC-7785, IBM Research, Yorktown Heights, N.Y. (July 1979). *Computer 13,* 4 (April 1980) pp. 25-32.

SAUE79c C.H. Sauer, E.A. MacNair and S. Salza, "A Language for Extended Queueing Networks," IBM Research Report RC-7996, December 1979. *IBM J. of Research and Development 24,* 6 (November 1980).

SAUE80a C.H. Sauer, "Approximate Solution of Queueing Networks with Simultaneous Resource Possession," to appear as an IBM Research Report.

SAUE80b C.H. Sauer, "Numerical Solution of Some Multiple Chain Queueing Networks," to appear as an IBM Research Report.

SCHW77 M. Schwartz, *Computer-Communication Network Design and Analysis,* Prentice-Hall (1977).

SCHW78 H.D. Schwetman, "Hybrid Simulation Models of Computer Systems," *CACM 21,* 9 (Sept. 1978) pp. 718-723.

SEKI71 A. Sekino, "Performance Evaluation of Multiprogrammed Time-Shared Computer Systems," Proj. MAC TR-103, Mass. Inst. of Tech., Cambridge, Mass. (Sept. 1971).

SEVC77a K.C. Sevcik, "Priority Scheduling Disciplines in Queueing Network Models of Computer Systems," *Proc. IFIP Congress 77,* pp. 565-570.

SEVC77b K.C. Sevcik, A. Levy, S.K. Tripathi and J.L. Zahorjan, "Improved Approximations of Aggregated Queueing Network Subsystems," *Computer Performance,* K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 1-22.

SEVC79 K.C. Sevcik and M.M. Klawe, "Operational Analysis Versus Stochastic Modelling of Computer Systems," *Proc. Computer Science and Statistics: 12th Annual Symposium on the Interface,* University of Waterloo, May 1979.

SHER72a S.W. Sherman, F. Baskett and J.C. Browne, "Trace Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System," *CACM 15,* (1972) pp. 1063-1069.

SHER72b S.W. Sherman, J.H. Howard and J.C. Browne, "A Study of Response Time under Various Deadlock Algorithms and Job Schedulers," *ACM 1974 National Conf.*

SHUM77 A. Shum and J.P. Buzen, "The EPF Technique: A Method for Obtaining Approximate Solutions to Closed Queueing Networks with General Service Times, *Measuring Modeling and Evaluating Computer Systems,* H. Beilner and E. Gelenbe (Eds.) North-Holland, Amsterdam (1977) pp. 201-220.

SIMO61 H.A. Simon and A. Ando, "Aggregation of Variables in Dynamic Systems," *Econometrica 29,* 2 pp. 111-138 (April 1961).

SMIT66 J.L. Smith, "An Analysis of Time Sharing Computer Systems Using Markov Models," *AFIPS Conf. Proc. 28,* (1966 SJCC) pp. 87-95.

SMIT79 C.U. Smith and J.C. Browne, "Performance Specifications and Analysis of Software Designs," *Conference on Simulation, Measurement and Modeling of Computer Systems,* Boulder, CO (August 1979).

STEW78 W.J. Stewart, "A Comparison of Numerical Techniques in Markov Modeling," *CACM 21,* (Feb. 1978) pp.144-151.

TAKA63 L. Takacs, "A Single Server Queue with Feedback," *Bell Sys. Tech. J.* (March 1963) pp. 505-519.

TEOR72 T.J. Teorey and T.B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *CACM 15,* 3 (March 1972) pp. 177-183.

TOWS75 D.F. Towsley, "Local Balance Models of Computer Systems," Ph.D. Thesis, Univ. of Texas at Austin (Dec. 1975).

TOWS78 D.F. Towsley, J.C. Browne and K.M. Chandy, "Models for Parallel Processing Within Programs: Application to CPU:I/O and I/O:I/O Overlap," *CACM 21,* 10 (October 1978) pp. 821-831.

TOWS80 D.F. Towsley, "Queueing Network Models with State-Dependent Routing," *JACM 27,* 2 (April 1980) pp. 323-337.

VANT78 H. Vantilborgh, H. "Exact Aggregation in Exponential Queueing Networks," *JACM 25,* 4 (October 1978).

WALL66 V.L. Wallace and R.S. Rosenberg, "Markovian Models and Numerical Analysis of Computer System Behavior," *AFIPS Conf. Proc. 28,* (1966 SJCC) pp. 141-148.

WILH76 N.C. Wilhelm, "An Anomaly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling Using an Empirical Model for Disk Access," *CACM 19,* (Jan. 1976) pp. 13-17.

WILH77 N.C. Wilhelm, "A General Model for the Performance of Disk Systems," *JACM 24,* (Jan. 1977) pp. 14-31.

WOLF70 R.W. Wolff, "Work Conserving Priorities," *J. Appl. Prob. 7,* (1970) pp. 327-337.

WOLF77 R.W. Wolff, "The Effect of Service Time Regularity on System Performance," *Computer Performance,* K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 297-304.

WONG78a J.W. Wong, "Distribution of End-to-End Delay in Message-Switched Networks," *Computer Networks 2,* 1 (Feb. 1978) pp. 44-49.

WONG78b J.W. Wong, "Queueing Network Modeling of Computer Communication Networks," *Computing Surveys 10,* 3 (Sept. 1978) pp. 343-352.

WULF72 W. Wulf and C.G. Bell, "C.mmp, a Multi-miniprocessor," *AFIPS Conf. Proc. 41,* (1972 FJCC) pp. 765-777.

YU77 P.S. Yu, "Passage Time Distributions for a Class of Queueing Networks: Closed, Open or Mixed, with Different Classes of Customers, with Application to Computer System Modeling," SEL-77-017, Stanford Electronics Laboratories, Stanford, Calif. (March 1977).

ZAHO77 J.L. Zahorjan, "Iterative Aggregation with Global Balance," Proj. SAM Notes, Univ. of Toronto, Toronto, Ontario, Canada (Feb. 1977).

ZAHO79 J.L. Zahorjan, "Computational Algorithms for Queueing Networks with Product Form Solutions," *Topics in Performance Evaluation* (G.S. Graham, editor), CSRG-100, Computer Systems Research Group, University of Toronto, Toronto, Ontario, Canada (July 1979).

# INDEX

# COMPUTER SYSTEMS PERFORMANCE MODELING

Charles H. Sauer / K. Mani Chandy

COMPUTER SYSTEMS PERFORMANCE MODELING was written to help designers, developers, and others achieve optimum system effectiveness at the lowest lifetime cost. "It is common, but unfortunate," authors Sauer and Chandy point out, "that performance is not seriously considered until the later stages of system evolution" when remedies, such as the acquisition of additional hardware, can be costly.

An attractive alternative to after-the-fact modifications is the mathematical model that estimates performance early in the design and development stages and permits appropriate adjustments before the system becomes operational.

This comprehensive volume introduces the key concepts of modeling, their practical application, and their management. Well written, detailed chapters cover general principles, Markovian queuing models of computer systems, isolated queues and open networks of queues, closed product form queuing networks, approximation, simulation, measurement and parameter estimation, case studies, and management of modeling projects.

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632