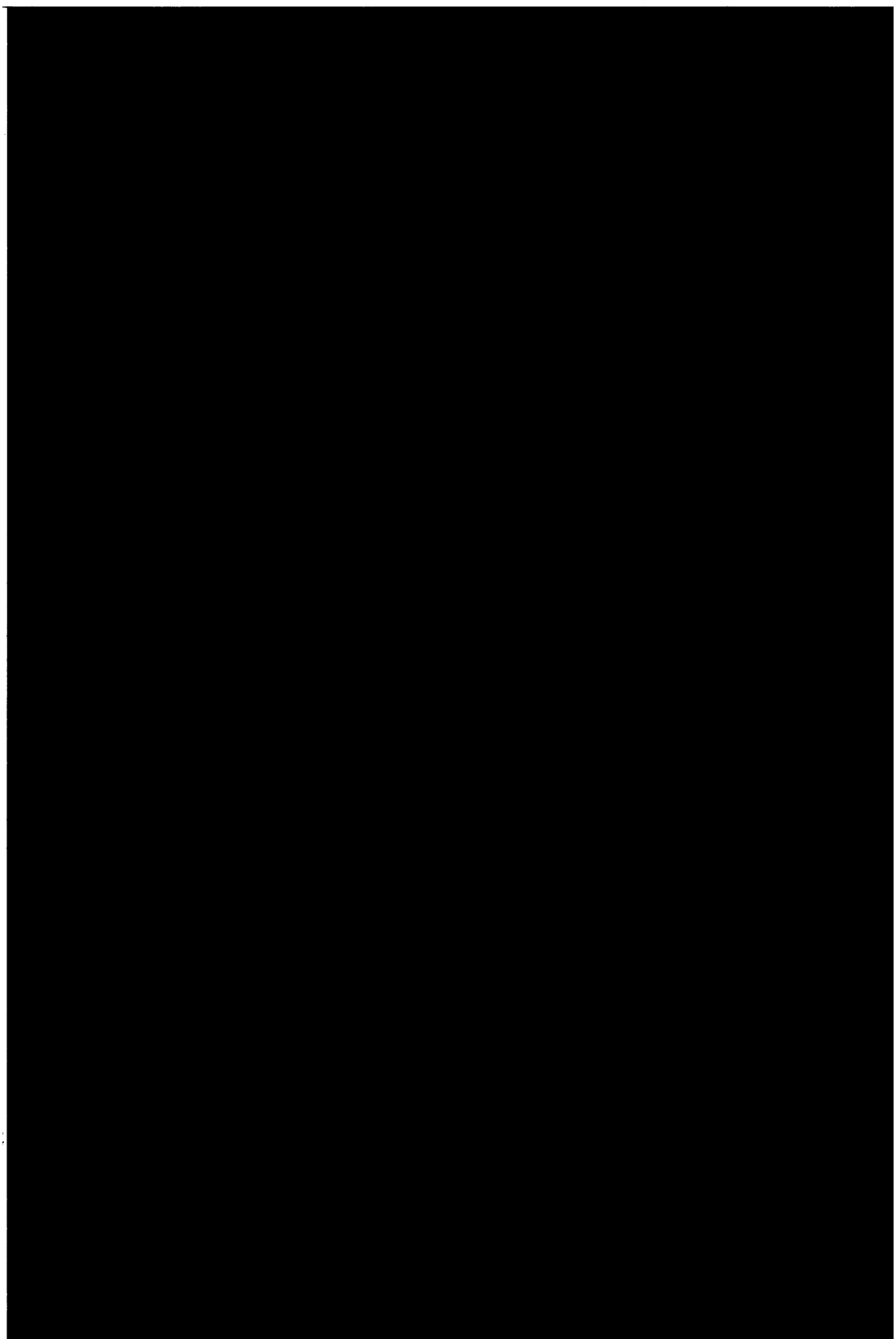


This blank page was inserted to preserve pagination.

RESOURCE ALLOCATION IN MULTIPROCESS COMPUTER SYSTEMS

by

PETER JAMES DENNING

B.E.E., Manhattan College
(1964)

S.M., Massachusetts Institute of Technology
(1965)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1968

Signature of Author Peter James Denning
Department of Electrical Engineering, May 10, 1968

Certified by Jack B. Dennis
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

This empty page was substituted for a blank page in the original document.

RESOURCE ALLOCATION IN MULTIPROCESS COMPUTER SYSTEMS

by

PETER JAMES DENNING

Submitted to the Department of Electrical Engineering
on May 10, 1968, in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy

ABSTRACT

The dynamic allocation of limited processor and main memory resources among the members of a user community is investigated as a supply-and-demand problem. The work is divided into four phases.

The first phase is the construction of the working set model for program behavior. This model is based on locality, the concept that, during any interval of execution, a program favors a subset of its information; a computation's working set is a dynamic measure of this set of favored information. A working set storage management policy is one that allocates processors to a computation if and only if there is enough uncommitted space in main memory to contain its working set. Under such a policy, a computation acquires and releases storage as needed, independently of other computations; because computations are thus made statistically independent, it is possible to derive many detailed properties of such policies, both in shared and unshared situations.

The second phase is to define and study the properties of system demand. A computation is regarded as the basic demand-making entity, placing demands jointly on processor and main memory resources. Its system demand is a pair (processor demand, memory demand), where its processor demand represents its immediate processor requirement (intensity and duration), and its memory demand represents its immediate main memory requirement (its working set size).

The third phase is to define and study the properties of system balance. Computations that demand resources are segregated into two classes: the first class, called the standby set, is temporarily denied the use of system resources; the second class, called the balance set, is granted the use of system resources. The system is balanced when the total system demand of the balance set matches the system capacity. A balance policy is a resource allocation policy that regulates membership in the balance set so that balance is maintained. Balance policies are formulated as mathematical programming problems whose solutions are found dynamically by the scheduler.

The fourth phase is to apply all these ideas to the design and administration of multiprocess computer systems. A relation describing the equipment configuration is derived; suggestions for processor and multilevel memory system design are made. Performance measures are discussed.

This work is intended to be a new approach to modelling the behavior of ongoing computations. It is intended to be a general, unified philosophy about allocation and sharing. It is intended to spark new thinking about the design and administration of multiprocess computer systems.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Associate Professor of Electrical Engineering

This empty page was substituted for a blank page in the original document.

ACKNOWLEDGEMENT

Most of all I must thank Jack Dennis, my advisor, for his continuing confidence in me, and for his assistance through all the various stages of doctoral study at M.I.T.

Especial thanks go to R. M. Fano and J. H. Saltzer, who were my thesis readers, for many discerning editorial comments.

Discussions with Donald Slutz and Fred Luconi (of M.I.T.), with Ed Coffman (of Princeton), and with Les Belady (of IBM Research Center at Yorktown, New York) led to many valuable insights which I might not otherwise have obtained.

Through it all, the patient understanding of my wife, Anne, has eased the burden, making it possible for me to embark on and complete my doctoral studies, and to enter the career of my choice.

May, 1968 Peter J. Denning

This empty page was substituted for a blank page in the original document.

TABLE OF CONTENTS

SECTION	PAGE
Abstract	i
Acknowledgement	ii
Table of Contents	iv
Table of Illustrations	ix
Notation	xii
CHAPTER 1 -- THE RESOURCE ALLOCATION PROBLEM	
1.0. Introduction to the Resource Allocation Problem	1
1.1. Plan of the Thesis	6
1.2. The Problem and Its Constraints	8
1.3. Why Balance?	13
1.4. Supply-and-Demand Economic Principles	16
1.4.1. Demand Curves	17
1.4.2. Priorities	22
1.4.3. Bidding	23
CHAPTER 2 -- THE ENVIRONMENT	
2.0. Introduction	26
2.1. The Basic System	27
2.2. Multiprocess Computer System Concepts	31
2.3. Summary	38
CHAPTER 3 -- THE WORKING SET MODEL FOR PROGRAM BEHAVIOR	
3.0. Introduction	39
3.1. Locality and Working Sets	40
3.1.1. Definition and Justification	40
3.1.2. Pictorial Representations	45
3.1.3. Interactions	50

3.2. Convexity 53
3.3. Working Set Size 56
3.4. Storage Management Policies 62
3.5. Working Set Strategies are Optimum 69
3.5.1. The Cost of a Strategy 69
3.5.2. Single Program Case 73
3.5.3. Multiprogrammed Case 79
3.5.4. Use of Biased Replacement Rules 83
3.6. Thrashing 85
3.6.1. The Causes 85
3.6.2. The Cures 89
3.7. Survey of the Literature 93
3.8. Summary 95

CHAPTER 4 -- FURTHER PROPERTIES OF THE WORKING SET MODEL

4.0. Introduction 97
4.1. The Relation between Program Sizes and Interreference Intervals 100
4.2. Missing-Page Probability 105
4.3. Paging Rate 110
4.4. Working Set Size 112
4.5. Duty Factor 118
4.6. τ-sensitivity 119
4.7. Choosing τ 121
4.8. Prediction 125
4.9. Example 128
4.10. Working Sets and Parachors 129
4.11. Implementation of Working Set Memory Management 130
4.12. Summary 136

CHAPTER 5 -- MULTIPROCESS INFORMATION SHARING

5.0. Introduction 137

5.1. Sharing 139

5.1.1. General Aspects 139

5.1.2. Refinement of the Working Set Definition 141

5.1.3. Implementation 143

5.2. The Model to be Analyzed 146

5.3. Shared Working Set Size and Memory Costs 150

5.4. Missing-Page Probability 156

5.5. Paging Rate 159

5.6. Duty Factor 161

5.7. Variable Number of Participants 163

5.8. Summary 165

CHAPTER 6 -- DEMANDS AND BALANCE

6.0. Introduction 166

6.1. Memory Demand 168

6.2. Processor Demand 170

6.2.1. Multiprocess Computations 172

6.2.2. Single-process Computations 172

6.3. System Demand 177

6.4. System Balance 178

6.5. Balance Policies 182

6.5.1. Demand and Usage Spaces 182

6.5.2. Properties of a Balance Policy 186

6.6. Survey of the Literature 188

6.7. Summary 190

CHAPTER 7 -- IMPLEMENTATION OF BALANCE POLICIES

7.0. Introduction 191

7.1. Analysis of a Single-Server Queue 193

7.2. Organization of the Queues 200

7.2.1. An Almost-Continuous System of Queues 200

7.2.2. The Logarithmic Queue 207

7.3. Mathematical Programming Problem, One-Dimensional Case 212

7.3.1. The Problem 212

7.3.2. The Objective Function 214

7.3.3. Choice of Quanta 216

7.3.4. Solution to the Memory Balance Problem 218

7.3.5. On the Optimality of the Solution 222

7.4. Mathematical Programming Problem, Two-Dimensional Case 223

7.5. Summary 227

CHAPTER 8 -- APPLICATIONS TO COMPUTER SYSTEM ORGANIZATION

8.0. Introduction 228

8.1. Toward Better Programming and System Design 229

8.2. The Equipment Configuration 231

8.2.1. Choosing the Balance Parameters α and β 231

8.2.2. How Much Resource Slack? 237

8.2.3. Relation Among Processors, Memory, Traverse Time 239

8.3. Pooling 243

8.4. Multilevel Memory Systems 250

8.4.1. Managing the Main Levels 254

8.4.2. Managing the Auxiliary Levels 257

8.4.3. What About Pre-paging? 258

8.5. The Environment Graph Information Structure 261

8.6. Summary 267

CHAPTER 9 -- PERFORMANCE MEASURES AND ACCOUNTING PROCEDURES

9.0. Introduction 268

9.1. What to Measure and Why 269

9.2. Charging for Resource Use 274

CHAPTER 10 -- CONCLUSIONS 276

BIBLIOGRAPHY 281

BIOGRAPHIC NOTE 285

This empty page was substituted for a blank page in the original document.

LIST OF ILLUSTRATIONS

FIGURE		PAGE
1-1.	Tradeoff between waiting time and idleness.	14
1-2.	A demand curve.	18
2-1.	Basic system, with two-level memory.	29
2-2.	States of a process.	34
3-1.	Evidence supporting locality.	42
3-2.	Definition of a working set.	43
3-3.	Association of working set with process.	46
3-4.	Time movement of a working set.	48
3-5.	Working sets of multiprocess computation C.	49
3-6.	Organization of an interactive modular program.	52
3-7.	Illustrating convexity theorem.	55
3-8.	For the proof of Theorem 3.2.	57
3-9.	Expected working set size.	59
3-10.	Missing-page probability for two strategies.	71
3-11.	Cost per unit virtual time for two strategies.	71
3-12.	Conceptual experiment to compare strategies.	74
3-13.	Missing-page probability for the strategies.	74
3-14.	Experiment to compare LRU and WS.	77
3-15.	Comparison of LRU and WS.	77
3-16.	Behavior of missing-page probability	81
3-17.	Duty factor for various T.	86
3-18.	Three-level memory system.	91

4-1. A simple program model. 103
4-2. Illustrating the meaning of re-entry rates. 106
4-3. Sequence of references to page i. 107
4-4. Interreference interval containing time t. 113
4-5. Using $s(\tau)$ to choose τ. 120
4-6. Cost per unit virtual time $H(\tau)$. 123
4-7. Hardware implementation of memory management. 133
4-8. Software implementation of memory management. 134

5-1. Implementation of shared memory management. 144
5-2. Experiment to investigate sharing. 147

6-1. Probability density function $f_q(u)$. 174
6-2. Conditional expectation function $Q(\gamma)$. 174
6-3. Job flow in balanced computer system. 180
6-4. Demand and usage spaces. 183
6-5. The path effect. 185

7-1. Single-Server queue. 194
7-2. Sorting jobs into size classes in standby set. 201
7-3. The logarithmic queue. 208
7-4. Demand space as a continuous two-dimensional queue. 224

8-1. Memory Usage. 238
8-2. Desired processor-memory ratio. 241
8-3. Relation among processors, memory, traverse time. 241
8-4. Pooled vs. private resource supplies. 244

8-5. Full interconnection of processors and memories. 248

8-6. Organization of multilevel memory. 251

8-7. An environment graph. 263

8-8. Representation of a linear sequence of instructions. 263

8-9. Multilevel memory for use with environment graph. 265

This empty page was substituted for a blank page in the original document.

NOTATION

To prevent confusion, we list here the major notational conventions we have used in this thesis.

Notation	Explanation
$x \varepsilon A$	x is an element of the set A.
$A = \{x \mid P\}$	A comprises all elements x having the property P.
$\lvert A \rvert$	the number of elements in A.
$A \subseteq B$	the set A is contained in the set B; i.e., every element of A is also an element of B.
$B = \bigcup_{i \varepsilon I} A_i = \{x \mid x \varepsilon A_i, \text{ some } i \varepsilon I\}$	definition of the union of sets.
$B = \bigcap_{i \varepsilon I} A_i = \{x \mid x \varepsilon A_i, \text{ all } i \varepsilon I\}$	definition of the intersection of sets.
$Pr[A]$	probability of the event A.
$Pr[A \mid B]$	probability of the event A, conditioned on the occurrence of the event B.
$F_x(u) = Pr[x \leq u]$	probability distribution function for the random variable x.
$f_x(u) = \frac{d}{du} F_x(u)$	probability density function for the random variable x.
$\bar{x} = \int u \, f_x(u) \, du$	the mean, or expectation, of the random variable x.
$\overline{x^2} = \int u^2 \, f_x(u) \, du$	the second moment of the random variable x.
$\sigma_x^2 = \overline{x^2} - \bar{x}^2$	the variance of the random variable x.
$\overline{g(x)} = \int g(u) \, f_x(u) \, du$	expectation of the function g(x) of the random variable x.
$\overline{g(x,y)}^x = \int g(u,y) \, f_x(u) \, du$	expectation with respect to x of the function g(x,y) of two random variables x and y.

This empty page was substituted for a blank page in the original document.

CHAPTER 1

The Resource Allocation Problem

1.0. Introduction to the Resource Allocation Problem

The desire for a general purpose community computing facility -- a computer utility -- has motivated recent trends in computer design. Just as electric power is distributed to the members of a community to satisfy their electromechanical needs, so-called computing power can be distributed to the members of a community to satisfy their information processing needs.

The essence of a computer utility can be captured in one word: sharing. By sharing computing resources, the users distribute the costs, and each pays less. By sharing information, one user may build on the work of others, and advance more rapidly in his own work. Sharing benefits the system, too, for the system may select from a wide range of instantaneous demands those that tend to improve its efficiency. Resource allocation is the problem of distributing limited resources among members of the community.

In recent years we all have watched the evolution of sophisticated techniques for sharing of equipment and information, techniques such as multiprogramming, multiprocessing, multi-

accessing [C7,C8,D7,P2], segmentation and paging [D8], and traffic control [S2]. Computer systems using these techniques have not always met expectations. For example, it has been observed that the efficiency of paged memory systems has often been much less than anticipated. There have even been instances of unexpected behavior. For example, it has been observed that it is possible to be processing a set of programs using all the available memory and processor resources; introducing an additional, average-sized program into memory can trigger a total collapse of service efficiency, leaving almost all the processors idle. This phenomenon, known as thrashing, at first defies our intuition, which would instead lead us to expect gradual degradation of service as additional programs are squeezed into memory.

What causes thrashing? When multiprogramming a memory, what is the smallest subset of each program that ought to reside there? What (perhaps unwanted) interactions take place among programs that compete for the same equipment? Given a set of programs, what should be the configuration of processor and memory resources to serve them best? What is the best scheduling policy? The best storage management policy? How does one predict the resource requirements of a program when nothing is known about it beforehand? How can one tell if the system is behaving properly?

The lack of answers to some of these questions, the intense debate over others, and the existence of yet unasked questions, lead inescapably to this simple conclusion: we do not understand the behavior of ongoing computations.

Thus, multiprogramming, multiprocessing, and all the other techniques, are not solutions to the resource allocation problem; they are but tools by which a solution may be implemented.

It is the purpose of this thesis to start filling the gap, to develop new approaches to modelling the behavior of computations, to spark a new way of thinking about programs in execution, to evolve a general, unified philosophy about resource sharing and allocation.

I felt that an interesting and useful solution to the resource allocation problem should be based on the ideas of supply-and-demand economics in a free-enterprise market; and this thinking underlies my work. I wanted to formulate resource allocation as the problem of selecting fairly from all user demands a subset whose total demand balances the supply; I wanted the solution to be applicable across a wide range of computer systems, large and small, existing and proposed, from Multics [C8], to Dijkstra's harmonious society of cooperating sequential processes [D11], to the highly parallel machines of Dennis [D10] and Slotnick [S6]; I wanted the solution to be unified in the sense that processor and memory allocation are handled together, not in two separate decisions. To accomplish these goals, I approached the problem in four phases.

The first phase was the construction of an abstract model for program behavior. This model, the working set model, makes it possible to decide which information is in use by a single computation or set of computations; intuitively, a computation's working set of information is the smallest collection of information that must be present in main memory for it to operate efficiently. The working set model is based on the concept of locality, the idea that a computation will, during an interval of time, favor a subset of the information available to it; the working set is a dynamic measure of this set of favored

information. A working set memory management policy is one that guarantees a computation shall receive the use of processors if and only if its working set is present in main memory. Under such a policy, computations are made independent, the memory acquisitions of one computation being unaffected by those of another; thus, unwanted interactions among computations arising from competition for memory and processor resources may be eliminated. Under such a policy a computation acquires more or less memory in accordance with its needs.

The second phase was to define demand. Observing that a computation jointly demands the use of processor and memory resources, we defined a computation's system demand to be a pair

(processor demand, memory demand)

where a processor demand represents the computation's immediate processor requirement (intensity and duration), and a memory demand demand represents the computation's main memory requirement (its working set size).

The third phase was to investigate system balance. We will say that the system is balanced when the sum total of the demands of active computations matches the available equipment. This set of active computations will be called the balance set. A balance policy is a resource allocation policy that regulates membership in the balance set so that the balance set, regarded as a super-computation, has known characteristics; its total demand is maintained within close tolerance of whatever is required to match the equipment. We have been able to formulate the problem of deciding which computations are to be members of the balance set as a mathematical programming problem, whose solution is found dynamically by the scheduler.

The fourth phase was to apply all these ideas to the design and administration of computer systems. A particularly important result is: the proper ratio of processor to memory (that is, the equipment configuration) that achieves some efficiency level is determined not only by the statistics of program size and duration, but also by the access time of auxiliary storage devices. We are also able to make suggestions about processor design, multi-level memory system design, and performance measurements.

1.1. Plan of the Thesis

The thesis is organized into four parts. Chapters 1 and 2 review the concepts with which we want the reader to be familiar; Chapters 3, 4, and 5 deal with the working set model; Chapters 6 and 7 investigate demands, balance, and balance policies; and Chapters 8 and 9 look into implications the models have on system design and administration.

The remainder of the discussion here in Chapter 1 falls into two categories: constraints and economics. The most important constraint within which we assume a solution to the resource allocation problem must function, programming generality, is the independence of an algorithm description from the environment in which it operates. One of the consequences of this constraint is that the computer system must predict, without outside assistance, the demands of the computations it executes. A discussion of basic supply-and-demand economic theory is included to illustrate how pricing policies can be used to regulate demands. Chapter 2 reviews basic multiprocess computer system concepts.

In Chapter 3 we define the working set model for program behavior and show that a working set memory management policy is the optimum of all policies that must operate without knowledge of future reference patterns made by computations. In Chapter 4 the working set model is refined and a great many of its properties are derived. The discussion of Chapters 3 and 4 is restricted to the case in which no information is shared; accordingly we examine in Chapter 5 the effects of sharing. We show how dramatically sharing can improve efficiency and reduce the resource usage costs attributed to a particular user.

In Chapter 6 we present the formal definitions of demand and balance and discuss basic aspects of balance policies. Chapter 7 is devoted to formulating balance policies as mathematical programming problems. Such formulations have dual advantage: first, we need not find explicit solutions for the balance policies as long as we can convince ourselves that the scheduler is dynamically finding them; and second, we are assured that the policies are optimum since the objective functions are clearly stated.

Chapter 8 deals with applications to computer system design. We derive a relation specifying processor-memory configuration, we show that pooling of hardware at a fine level of detail can achieve the effect of a large number of processors with a small amount of hardware, and we discuss organization and management of multilevel memory systems in light of generalized working set models.

Chapter 9 deals with performance measures. Given the models and formulation of the solution to the resource allocation problem, the performance measures are determined, so Chapter 9 merely collects together the major measured discussed in earlier chapters.

The reader who merely wants to get a detailed overview of the major work of this thesis, without having to dig through the detailed properties of our models, need only read Chapters 1, 3, 6, and 8, for that is where the main thread lies.

1.2. The Problem and Its Constraints

We have formulated the problem in the context of a multiprocess computer system; we presume that the reader is already familiar with mulitprocess computer system objectives, the particular details of which may be found in references [C8,F1,P2,V2]. Specific implementation concepts will be reviewed in Chapter 2. The properties that constrain and complicate the solution to the resource allocation problem are discussed below.

The specific problem toward which this thesis work has been directed is:

> To formulate behavior models of computations in multiprocess computer systems; then, using the models, formulate a unified approach to dynamic allocation of processor-memory resources among computations, balancing supply against demand under appropriate criteria of fairness.

We have omitted discussion of input-output allocation for three reasons. First, we assume the time rate at which a user interacts with his computation is relatively very much slower than the time rate at which execution proceeds; we are interested primarily in dynamic resource allocation in the intervals between interactions. Second, we feel that the models are general enough so that generalization to resource types beyond processor and memory will be straightforward. Third, we feel that all the rich complexity of the resource allocation problem can be found entirely in the processor-memory problem.

We assume the existence of two kinds of constraints: limited equipment and programming generality.

The limited equipment constraints center on the existence of only a fixed, finite amount of processor and memory resources. There are N identical processors, each of which can deliver

information references at the rate of one per unit time; since the processing rate is bounded, the duration of a program's execution enters the problem. We assume the standard unit of information storage and transmission is a page, and that the capacity of directly-addressable, main memory is M pages. Whenever we talk of the equipment, or the resources, we specifically mean the N processors and the M pages of main memory.

The remaining constraints center on the issue of programming generality [D10], which is the independence of an algorithm description from the environment in which it operates. Programming generality includes

1. The ability to move a program between installations, either manually or automatically (e.g., via computer networks).
2. The ability to use a program, without changes, despite changes to the hardware or to the hardware configuration.
3. The ability to use one program in the construction of another -- to build on the work of others, and to share information dynamically.

This third aspect implies that programs will be modular in construction (i.e., programs will be segmented [D8]). Once compiled, a program module should be usable without recompilation as a building block of any program whatever. To exhibit programming generality, the computer system must permit a program module to:

1. Create data structures of arbitrary size unknown prior to execution.
2. Call on further procedures unknown to the caller (which may call on still other procedures, etc.)
3. Transmit arbitrarily complex data structures as arguments.

These last three points, centering on data dependence, imply that a module's resource requirements not only will be unknown prior to its execution but also will be indeterminable. Thus, the programming generality requirement places these constraints on the resource allocation problem:

1. The computer system, not the programmer or the compiler, must decide for itself where in the memory hierarchy information is to reside [D4,D10].
2. Algorithms must be configuration independent. Information references must be made by means of a location-independent addressing mechansim.
3. Information flows upward in the memory hierarchy only on demand, being moved into main memory only when it is referenced by a computation. Information flows downward in the memory hierarchy as it falls out of use.
4. Arbitrary collections of programs will demand to share arbitrary sets of data. Many programs will reside simultaneously in main memory (multiprogramming) and many processes will be active concurrently (multiprocessing).

In order to be consistent with programming generality, we have assumed the no-advance-information constraint, namely that programmers and compilers will, because of data dependence, be

unable to make reliable advance estimates about the resource needs of their own programs[1]. In addition, any advice that is obtained from a programmer cannot necessarily be regarded as useful advice even if it may be reliable: a user would intend to optimize the environment for his own program -- configuring resources to suit an individual may interfere with overall good service to the community. In order to guard against dishonest users who attempt to secure better service by misrepresenting their needs, the system must monitor program behavior, and impose penalties for bad estimates. The additional overhead to do this may not be worth the cost.

Since it is not at all clear that advice obtained from programmers or compilers can be of any real value, we have chosen to formulate a solution to the resource allocation problem in the case where there is no advice, where the computer system must discover for itself how programs behave. Clearly there will be situations in which advice can be useful, but these are not of interest to us here.

In the interest of programming generality we make the following distinction between the tools and the methods of resource allocation:

1. The mechanisms, or machinery, of assigning and releasing equipment must operate on a low level in that they deal directly with the hardware features of the system. Some of these tools include multiprogramming, multiprocessing, segmentation, paging, interprocess communication, etc.

[1] There have been attempts to do this. Ramamoorthy [R1] for example, has a proposal for automatic segmentation of programs during compilation.

2. The policies of resource allocation operate on a higher level, in that criteria used to determine when equipment is to be allocated to a computation can be machine-independent. They are machine-independent inasmuch as no detailed knowledge of machine organization is necessary or even relevant.

Such a functional separation permits changing policies without changing machinery, if the machinery is properly defined.

Since many of the mechanical aspects of resourde sharing already have substantial solutions, we begin investigation at the machine-independent level. Resource allocation policies may be grouped into classes:

1. Short-term policies, which must be handled by the computer system, since decisions must be made in a time scale far faster than human response.
2. Long-term policies, primarily economic, which control demands over long periods of time.

In our work here, short-term policies are concerned with matching the demand to the supply, long-term policies with matching the supply to the demand.

The bulk of the thesis is concerned with models that show how to define the short-term, balance policies. After a detailed discussion in the next section of why balance was chosen as a resource allocation goal, we turn attention to a discussion of the long-term, economic policies.

1.3. Why Balance?

There were good reasons to choose balance as the objective of resource allocation policies, rather than other criteria such as maximum equipment utilization or minimum response time.

The most important reason, already stated, is our desire to be consistent with the ideas of supply-and-demand economics.

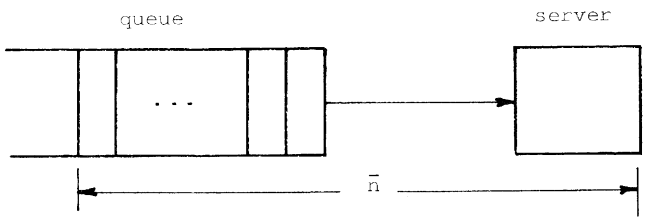
The remaining reasons are the results of this thesis. We state them here, although many of their justifications will not come completely to light until Chapter 7.

First, it is conceptually simple and mathematically tractable, and it insures a reasonable policy with respect to criteria such as maximum equipment utilization or minimum response time.

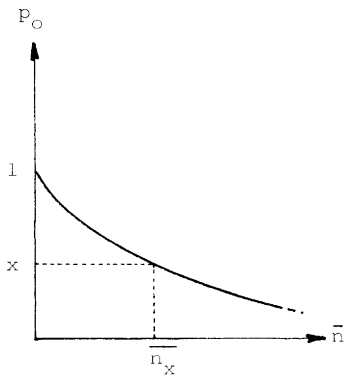
Second, we will show that its relative simplicity not only makes performance testing and evaluation straightforward but also makes clear which parameters are important. Moreover, its relative simplicity makes implementation easy.

Third, we will show that balance exercises control over the factors that cause thrashing; recall that thrashing denotes the sudden collapse of service efficiency that may occur when too many programs are squeezed into main memory.

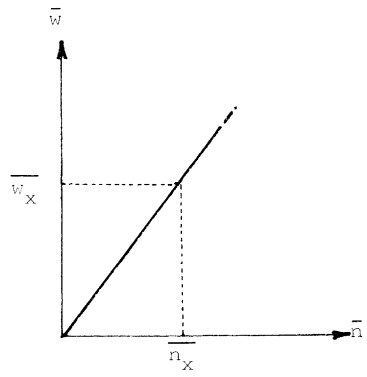
Fourth, balance compromises between the conflicting objectives of fast fair service and low equipment idleness. We illustrate the dichotomy. Figure 1-1a shows a server, before which is a queue of demands for its use, the average number in the system being $\bar{n}$; we regard $\bar{n}$ as a measure of the demand for use of the server. When there are no demands in the system, the server is idle, an event occurring with probability p_o. The average wait in the system is $\bar{w}$. Figure 1-1b shows how p_o varies with $\bar{n}$.



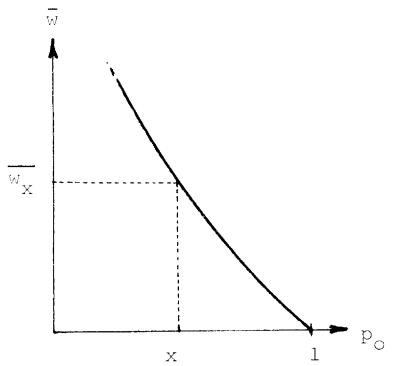
(a). Queue and server.



(b). Idleness vs. demand.



(c). Waiting time vs. demand.



(d). Waiting time vs. idleness.

Figure 1-1. Tradeoff between waiting time and idleness.

Figure 1-1c illustrates that under a fair service discipline (i.e., one in which waiting time depends only on order of arrival) the expected wait varies linearly with $\bar{n}$ (see reference [S1], p. 42). Figure 1-1d, showing directly the relation between $\bar{w}$ (service) and p_o (idleness), is constructed by choosing various x and finding the corresponding $\overline{w_x}$. In general, as p_o decreases, $\bar{w}$ increases: there is an inverse relation between fast fair service and low equipment idleness. As we will see, balance exercises control over this relation.

Fifth, we will show that a balance policy can be implemented in a relatively load-independent way, the amount of work needed to maintain balance depending on the distance of an actual load point to a desired load point.

Sixth, the abstract model of a balanced computer system will show the relation between equipment configuration, the auxiliary memory access time, and balance.

1.4. Supply-and-Demand Economic Principles

This section surveys aspects of the economic structure underlying our thinking. We consider here a form of supply-and-demand computer system economics.

One motivation for a multiprocess computer system is economic: there is a community of users, who individually would be unable to afford the full services of a computer system, but who collectively can pay the costs. This goal -- cheap computing -- is not attainable solely within a mutliprocess computer system. The ability of one user to share and build on the work of others is a far more compelling motivation. Yet sharing complicates, among other things, the problem of charging users for resource consumption, because now the cost of a shared resource must be attributed to the participants in accordance with their degrees of participation.

The perhaps overworked term computer utility can be misleading, for it is not entirely analogous to the public utilities as we know them. Contemporary public utilities are rather large economic systems where the average demand is known to vary slowly; for the immediate future, computer utilities will be rather small economic systems, subject to fast-changing demand. Public utilities are relatively much larger than computing systems; in a computer utility, any user can easily demand every resource. Public utilities have physical limits on the quantity of service a customer can obtain (a 150 ampere main circuit breaker in his house, a 3-inch water main, or 2 telephones), and this need not be the case for computer utilities.

From now on we shall refer to the management personnel of the computer system as the administration. It is the responsibility

of the administration to properly manage the system, deciding who is to use it, how prices are to be set, what additional equipment is to be purchased, what services are to be offered. Nevertheless, the burden of managing the system lies mostly on the system itself; for example, it must provide automatic metering of resource usage and maintain data on demands. The models we set up will make it clear what should be metered and how, and what demand distributions should be determined and how.

1.4.1. Demand Curves

The administration can exercise economic controls over the demands of the user community by means of the prices it charges.

Figure 1-2 shows an elementary demand curve, typifying the relation between price per unit resource and the total demand from the community. We observe that the higher the price per unit resource the less is the total community demand. Point A is the intersection between the amount R of resource currently provided by the system and the demand curve. If the price is less than p_A the user community demand will exceed the supply R. If the administration wishes to hold some resource in reserve, leaving only a fraction α of the R available, it must raise the price to p_B. We do not wish to consider issues such as how to set price to maximize profit, what to do if the demand curve is time varying, how long it is until a price change is felt, or whether instability will result from feedback between demand

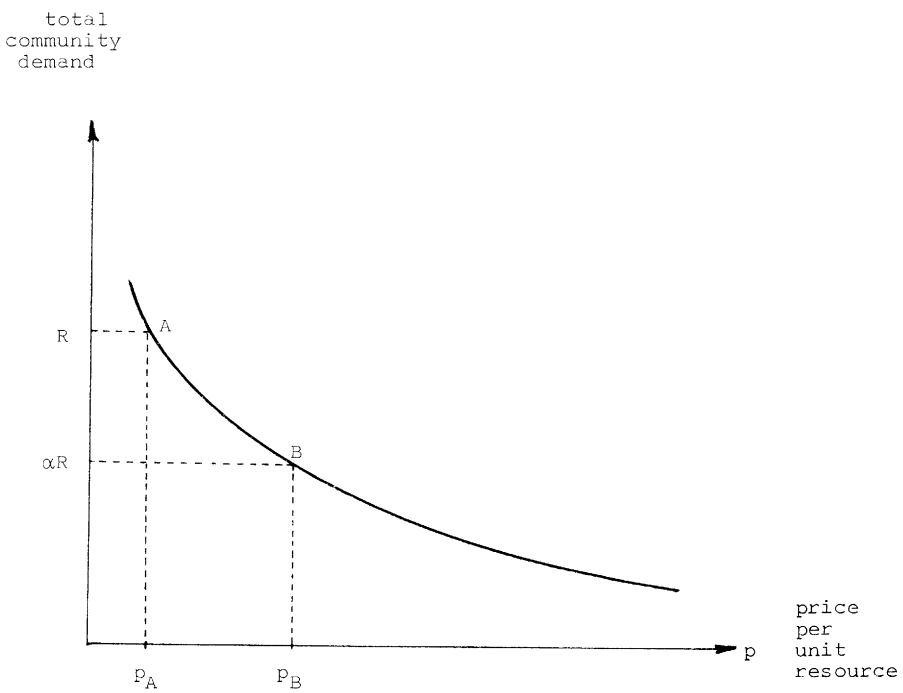


Figure 1-2. A demand curve.

and price. The point is: price is a lever for controlling the total community demand.

The demand curve of Figure 1-2 represents the behavior of some economic community in statistical steady state; therefore no claim can be made that at any particular time the demand curve is reliable. This further motivates the idea of dynamic balance: at any time the demand is closely regulated, known to be within close tolerance of the desired level.

An essential component of a supply-and-demand pricing policy is the ability for a user to bid. Should a user desire improved service (at correspondingly higher prices) he may outbid his fellows. Should a user be unconcerned with the quality of service, he may underbid, obtaining poorer service at reduced price. By assuming the existence of a bidding mechanism, we may ignore certain delicate questions surrounding the issue of user dissatisfaction; that is, we will not attempt to model dissatisfaction, hoping that unhappy users will raise their bids, or leave. We shall discuss details of bidding mechanisms in Section 1.4.3.

Such an atmosphere of free enterprise, incorporating supply-and-demand resource allocation and competitive bidding for priority, can quite possibly wreak havoc with computer system economics, there being a serious threat of inflation. In terms of Figure 1-2, bidding gradually forces the demand curve up and to the right. There are two extremes of thought concerning the administration's posture toward inflation:

1. Do nothing. Just as other public utilities do, meter resource usage, but allow users as much as they need. This means that the administration must be willing to expand the system, adding new equipment so long as

someone is willing to pay for it. It also means that the administration must be able to detect trends in the community demand so that it can decide far enough in advance to order new equipment.

2. Tight controls. The administration should exercise control over the total demand by allocating resource quotas to users, and by limiting the total number of users. This in some ways resembles the policies of parking lot officials, who allocate 150 stickers to fill 100 spaces, on the grounds that (on the average) only 100 cars will show up. The quotas allocated will depend on careful interpretation of demand statistics, and should be set so that the number of users trying to use the system at one time will present a total demand only slightly larger than system capacity.

By itself, the first alternative is not workable because there is a physical limit to how much a particular installation may be expanded, and users seem always to manage to find problems that consume the capacity of the system, no matter how large it is. By itself, the second alternative is not workable because it implies gradual degradation of service, since the system cannot meet the needs of the existing community. A truly flexible posture is a compromise between the two extremes: the administration must be prepared both to enforce controls and to expand capacity. [But who is to insure that the administration indeed takes such steps, when it is the one who profits by the inflation?]

In order to implement the compromise, the administration must monitor performance and detect overload. Overload may be defined as follows. First set tolerance limits on service, such

as maximum allowable response time, or minimum allowable service rate (that is, the fraction actually received of the resource demanded). Overload exists when the probability that service is not within the set limits exceeds some specified number; this probability is measured as the fraction of time service is poor.

Even if the administration might want to decide against a quota system, the users may still desire some such system, for self-protection. By having a self-imposed quota, a user can protect his pocketbook from a beserk computation. Should one of his programs run amuck, a quota would be exceeded and execution interrupted, the user being asked to decide whether to continue. Moreover, there should be some means whereby a user controls distribution, among his own computations, of whatever resources have been allocated to him. This is particularly useful if the user supervises some project and desires to control spending by subordinates.

What is to be done when total demand temporarily exceeds capacity? Should all jobs be given equally poor service? Or, should jobs be divided into two classes, one to receive good service, the other to receive no service at all? As we shall see, the first alternative results in a high rate of resource multiplexing and can easily cause thrashing; the second alternative may result in some jobs receiving no service. Balance can be used as a compromise: the balance set is that subset that receives all the service, but the membership in it is constantly changing.

1.4.2. Priorities

In general, the higher a job's priority, the better the service it obtains. Basically, there are three classes of priority used in today's computer systems [C3]:

1. Bought, by paying extra for better service. An example is the bidding mechanism discussed in the next section.
2. Acquired, by displaying favorable or unfavorable characteristics during execution. An example is the CTSS multilevel queue [C6, S3] in which long jobs receive little attention.
3. Deserved, by displaying favorable characteristics in advance of execution. An example, again, is CTSS [C6, S3] which gives jobs of small memory requirement better treatment than those of large memory requirement.

A given computer system may employ a combination of these three types of priority.

In our work here we shall consider only bought priority, and ignore acquired and deserved priorities. We ignore acquired priority because we deal with only completely fair resource allocation policies. We ignore deserved priority because we assume there is to be no advance allocation information.

1.4.3. Bidding

We shall adopt the point of view that the bidding mechanism is a method by which a user purchases priority from the computer system (Kleinrock uses the more colorful term bribing [K2]). The cost of priority will be added to a user's resource-consumption costs. If a user buys higher than average priority, the cost is positive (his bill is increased); if he buys lower than average priority, the cost is negative (his bill is reduced). By adopting this view, we insure that inflation due to bidding is on the cost of the priority and not on the cost of the resources themselves.

Let (p_1,p_2) be an interval of the real line; any point p in (p_1,p_2) is a possible priority. If a user takes no action to obtain priority, he is assigned some standard priority p_o. Otherwise he selects some priority p from (p_1,p_2). There is a cost-of-priority function G(p,t) satisfying

$$\begin{aligned} &G(p,t) > 0 \quad \text{if } p \in (p_o,p_2) \text{ at time } t \\ &G(p,t) = 0 \quad \text{if } p = p_o \text{ at time } t \\ &G(p,t) < 0 \quad \text{if } p \in (p_1,p_o) \text{ at time } t \end{aligned} \tag{1.4.1}$$

Let $C_k(I)$ represent the resource-consumption cost for user k in the real time interval I; then user k would be billed

$$C_k(I) + \int_I G(p,t)\,dt \tag{1.4.2}$$

There is clearly an incentive for a user to underbid his fellows, and a restraint against his overbidding.

Let $q_1,\ldots,q_n$ be the priorities of each of the n users at a certain time, and define the average priority to be

$$\bar{q} = \frac{q_1 + \ldots + q_n}{n} \tag{1.4.3}$$

An interesting example of a cost function is

$$(1.4.4) \qquad G(p,t) = G(p) = C_o A^{h(p)} \log_A h(p) , \quad h(p) = \frac{p}{\bar{q}}$$

for suitable constants C_o and A. The reader can verify that G(p) satisfies properties (1.4.1). Since G(p) increases exponentially with the deviation from the mean $\bar{q}$, it is possible to penalize a user severely for large deviations. This discourages those who would outbid the entire community and pre-empt all service for themselves. Observe, however, that if everyone bids high, $\bar{q}$ increases and the relative cost of a high bid is less. Thus inflation can be a serious problem (but note: the inflation is on the cost of priority, not on the cost of resources, and is not as serious as inflation of the resource costs themselves). The administration can control inflation by replacing $\bar{q}$ with p_o in eq. 1.4.4, and making p_o smaller than the existing $\bar{q}$.

We do not want the purchased priority to modify the demand of a job, for a simple reason. Should priority be allowed to modify a job's demand, operation of a balance policy would collapse: the scheduler would fail to keep the balance set demand at the desired level because the demands of its members were not accurately reported.

The position of a job within its queue depends on the particular interpretation of the priority p it possesses. The two possibilities are:

1. Fixed priority. An incoming job of priority p is placed ahead of any job with priority less than p, but behind any job with priority greater than or equal to p.

2. Percentile priority. If the priority range (p_1,p_2) is taken to be (0,1), then p may be interpreted as a percentile. That is, the user wishes to be always ahead of 100p per cent of the jobs. An incoming job of priority p, arriving to a queue of length n, is placed a distance (1-p)n from the front of the queue.

The fixed priority interpretation will in general mean that a user experiences different degrees of improved (or degraded) service, depending on the instantaneous demand of his job. For example, suppose his job oscillates between two demand classes, designated A and B, and there is a separate queue for each class. Let p_A denote the largest priority of a class A job, p_B the smallest priority of a class B job, and $p_A<p_B$. Suppose the user happens to choose his priority to be p such that $p_A<p<p_B$. When in class A, he receives the best of service; when in class B, the worst. The percentile method circumvents this difficulty, always giving the user the same improvement (or retardation) relative to other users.

We conclude by noting an interesting way to implement bidding. Each console is provided with a potentiometer, calibrated on the range (p_1,p_2); the user may continuously adjust his priority. This can be enhanced by supplying a meter, also calibrated on the range (p_1,p_2), which indicates the current average priority $\bar{q}$ across all users, and the particular user can adjust his own priority with respect to the average. The existence of such a meter constitutes instantaneous feedback between an economic system and the competitors: some very interesting inflation and deflation effects could occur, perhaps even resulting in conditions very similar to those in the stock market in 1929.

CHAPTER 2

The Environment

2.0. Introduction

The environment, consisting of the hardware and the software operating system plays an important role in the resource allocation problem. The reader should already be familiar with the concepts of virtual computer, of segmentation and paging [D8], of program and address-mechanism structure [A1], of a process and parallel processes [D9], and of virtual time. We shall review these concepts here in order to establish the complete picture (as we see it) of a multiprocess computer system.

2.1. The Basic System

For ease of understanding both operation and design, it is usual to view the processing function and memory function separately in a computer system. The processing function performs transformations on information stored by the memory function. The processing function is usually implemented by one or more processors, and the memory function by one or more memory modules.

To satisfy the system objectives requiring expandability, reliability, and continuous availability, modular hardware construction is common: the processing function becomes a pool of identical processors with free and unrestricted access to a pool of identical memory modules. Removing (adding) a device from (to) a pool reduces (increases) the capacity of the pool. Within a pool each device is anonymous, there being no a priori assignment of any particular task to any particular device.

The high cost of directly-addressable memory forces memory systems to consist of at least two levels:

1. main memory. No information can be processed unless it is present in main memory. Main memory is usually a magnetic core memory, though it could just as well be any other directly-addressable storage device, such as a thin-film memory. Other terms for main memory are primary memory and execution store.
2. auxiliary memory. Information which for one reason or another cannot be stored in main memory stored in auxiliary memory. Examples of auxiliary memory are drums, disks, and tapes, although a slow-speed core memory might also be used for this purpose. Other terms for auxiliary memory are secondary memory and backup store.

Main memory has relatively high cost, but also has rapid access time; auxiliary memory has low cost, but also has slow access time.

Initially we shall restrict attention to a computer with a two-level memory system, indicated by Figure 2-1. After having studied program models, we shall generalize to multilevel memory systems; this will be done in Chapter 8.

We assume that the unit of information storage and transfer is the page. We suppose the capacity of main memory is M pages, and the capacity of auxiliary memory is infinite.

The N processors and M main memory pages will be called the equipment. For generality we assume that only a fraction α, for $0 \leq \alpha \leq 1$, of the N processors are available, and that only a fraction β, for $0 \leq \beta \leq 1$, of the M memory pages are available. The αN processors and the βM memory pages constitute the available equipment. It is against the available equipment that we want to balance demand.

We suppose that each processor can deliver one reference per unit time, and that each item in main memory can be referenced no more than once per unit time, so that the processor and main memory speeds are matched. This unit of time will be called a virtual time unit (vtu).

There is a time T, the traverse time, involved in moving one page between memory levels. T is measured from the moment a page is found to be missing from main memory until the moment the missing page has been placed in main memory ready for use. T is actually the expectation of a random variable composed of waits in queues, access times, mechanical positioning delays, and transmission times. We shall regard the traverse time T as being the same regardless of which direction a page is moved.

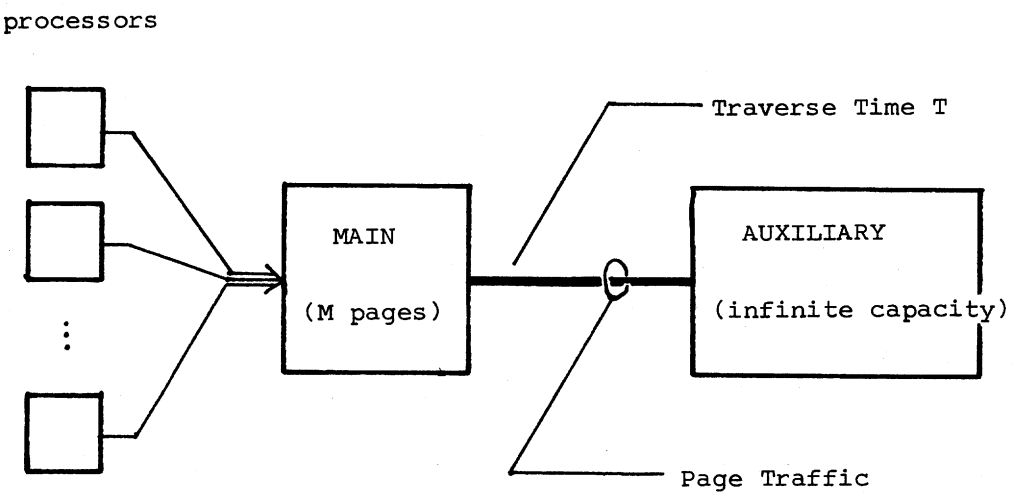


Figure 2-1. Basic System, with two-level memory.

Dividing memory into two levels creates the first allocation problem: storage management, the problem of deciding which information is to reside in main memory, which is not. Generally, the least-used information must be stored in auxiliary memory; the most-used information must be ready for use in main memory. When a processor makes a reference to a page not in main memory, a page fault occurs, initiating action to secure the missing page from auxiliary memory. We thus assume pages are brought into main memory on demand only. Because not every useful page may reside in main memory, there will be a flow of information -- called page traffic -- along the channel bridging the two levels. The activity of moving pages in and out of main memory is called page-turning, or simply paging.

Nowhere in Figure 2-1 have we indicated the existence of input-output equipment, the media used by programs to communicate with the outside world, because we are not concerned with this type of allocation in this thesis (see Section 1.2).

2.2. Multiprocess Computer System Concepts

Two basic principles in the design of multiprocess computer systems are the abstractions of the notions name space from memory, and process from processor. In the interest of programming generality, a user is given the illusion that he is dealing with a (configuration-independent) virtual computer. The virtual computer comprises one or more virtual processors each having most of the capabilities of a real processor, and a virtual memory having many times the capacity of the real memory. Because the virtual memory has so large a capacity, the user sees no auxiliary memory; for this reason virtual memory is often called a one-level store [K1]. It is the task of the operating system both to simulate virtual memory by paging information into real memory, and to simulate virtual processors with real processors. In Multics, the traffic controller mechanism [S2] handles assignment of real processors to virtual processors, and communication among virtual processors.

The first abstraction, name space, is the set of names (addresses) available to a virtual processor for use as data identifiers.

For convenience (to the user) the name space is divided into segments, of arbitrary size. To reference a datum, a two-component address (S,W) is given, S being the name of a segment, and W being the name of a word within S. Because names have two components, the name space is often called two-dimensional. There is no a priori relation between a name in name space and the location of the corresponding datum in physical memory; this correspondence is established dynamically by the address-mapping mechanism [A1].

For convenience (to the system) in mapping segments of arbitrary size into a memory of fixed size, segments and real memory are divided into equal-size blocks, called pages. The page, invisible to the programmer, is the standard unit of information storage and transmission. We may thus regard the name space as being sliced into equal-size regions.

Associated with each segment is a page table (itself a page) listing each page of the segment. If a page is not in main memory, an in-core bit of the corresponding page table entry is OFF; an attempt by a virtual processor to reference such a page automatically causes a missing page fault, which interrupts execution of the virtual processor and initiates action to secure the missing page from auxiliary memory. After a lapse of at least one traverse time (T) the page has been placed in main memory and is ready for use; the proper page table entry is set to point to the physical memory location of the start of the page, the in-core bit is turned ON, and execution of the interrupted virtual processor is resumed. Later on, when the page is removed from main memory, the in-core bit of the corresponding page table entry is again turned OFF.

It is apparent that pages are on a lower level of abstraction than segments. The operating system should not attempt to have each page of a segment in main memory; it should instead attempt to have each useful page in main memory. For it is possible that only some of a segment's pages are in use, and there is no need to strain main memory resources by keeping useless pages there. Roughly speaking, a working set of pages is the smallest collection of pages that must be present in main memory for a program to operate efficiently. Storage allocation should

attempt to keep at least the working set of each running program in main memory.

The second abstraction, process, is the notion of a program in execution by a virtual processor. In our work here, we use the equivalent definition: a process is an ordered sequence of references to information in name space, under the control of an instruction stream. A process is sometimes referred to as a thread of control through an instruction sequence. A process has four states of existence in real time:

1. running, meaning that is is receiving the use of a real processor; alternatively, that a real processor is assigned to its virtual processor.
2. ready, meaning that it is demanding, but not receiving, the use of a real processor; alternatively, it is suspended only because no real processor is currently assigned to its virtual processor.
3. page wait, meaning that it is temporarily syspended because a page is missing from main memory. Execution is resumed as soon as the missing page has been placed in main memory and a processor is available. We take the duration of a page wait to be the traverse time T.
4. blocked, meaning it has no use for a real processor because it is awaiting the occurrence of some (expected) external event, such as a message or signal from another process, from a device, or from a user at a console.

Figure 2-2 illustrates the possible transitions among these states. The transition from running to ready under a pre-emption means that the operating system has required the real processor for some other use, for example to execute another process. The transition from ready to running under go-ahead means that the

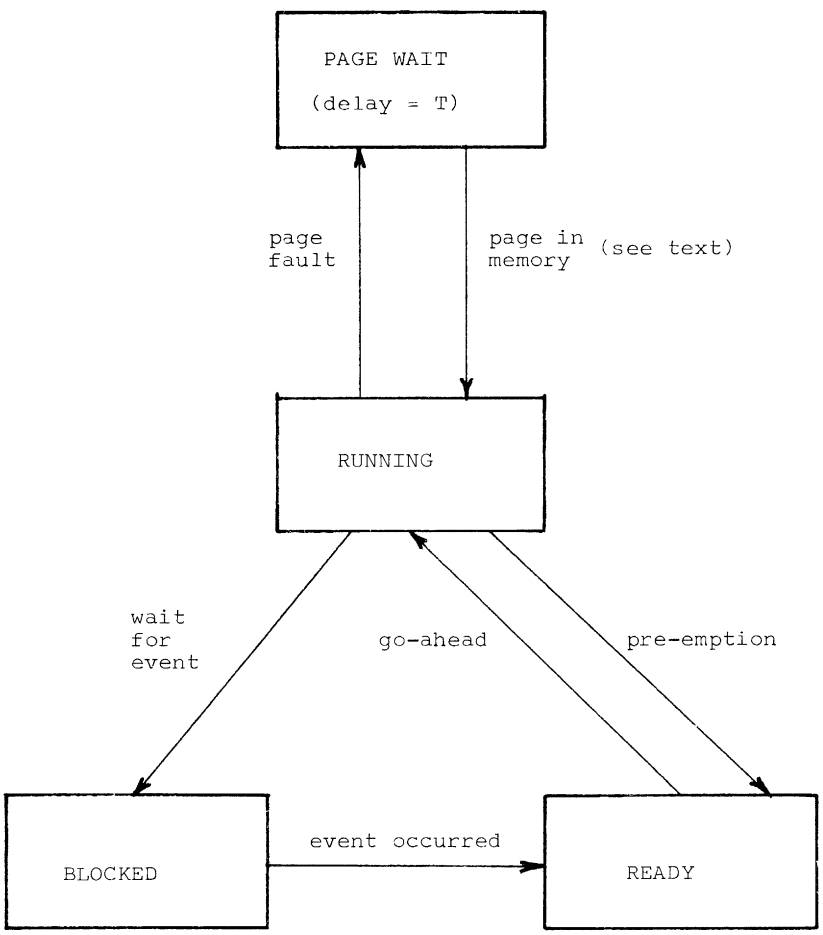


Figure 2-2. States of a process.

operating system has decided to return the processor to this process.

In Figure 2-2 we have indicated a transition from page wait directly back to running, when in fact this need not be the case. It is the case if a processor is dedicated to it, being immediable to resume execution when the process returns from page wait. But, if the page wait time T is larger than the time it takes to switch the processor to another process, it is uneconomical to dedicate a processor to a single process, and in this case a process returns to running status via ready status. In our work here, we assume a sufficiency of processor resources, so that at worst a negligible delay is experienced by a process as it passes through ready to running. This is the justification for the direct page-wait to running transition shown in Figure 2-2.

When talking about processes we shall make a distinction between virtual time (vt) and real time. Virtual time is time as seen by a process as if it were never interrupted; that is, the total accumulated time in the running state. Virtual time, also called execution time or process time, is measured in virtual time units (vtu), usually memory cycles. Put another way, a virtual time unit is the interval between any two of the successive information references that constitute a process. We shall usually regard virtual time as being continuous, even though it is actually finely divided into small units. Finally, real time is virtual time with page wait, blocked, and ready delays inserted appropriately.

When we talk about the virtual time interval $(t-\tau,t)$ we shall mean the τ information references prior to the real time instant t.

Because a process is an ordered sequence of information references it is often called a sequential process [D11]. In a multiprocess computer system, many processes may be executed concurrently, or in parallel. Thus, we may speak of parallel sequential processes.

We define a computation to be a collection of mutually cooperating processes and information, all operating in the same name space. In Multics [C8, S2, V2] every computation is a single-process computation, since there is a one-to-one correspondence between a process and a name space; however a programmer can execute a program with parallel processes by setting up a collection of single-process computations with isomorphic name spaces. IBM System 360 [R3], RCA Spectra 70 [O2], and THE-Multiprogrammed System [D11] are other examples of systems using single-process computations. The Illiac IV [S6] is an example of a system using multiprocess computations.

The constraints among the member processes of a computation are, from the resource allocation viewpoint, unspecified and must be considered arbitrary. For the very same reasons that compilers and programmers cannot specify before hand the resource needs of their programs (because of arbitrary timing of parallel processes and data dependence), compilers and programmers cannot predict the constraints among parallel processes.

By the term contemporary computer system we shall mean a Multics-like system, characterized by single-process computations. Such systems are not geared for a high degree of intra-computation parallel programming because in them the tables specifying a computation are so ponderous that the cost of spawning new processes is prohibitive.

We assume that the operating system allocates resources to computations, rather than to processes individually. Thus a commitment must be made to grant a computation all the processors and all the memory it needs. In a contemporary computer system, the notion of scheduling a process is the same as this more general notion of scheduling a computation.

2.3. Summary

We have reviewed the basic concepts of the computing environment, presuming familiarity with such common notions as segments, pages, demand paging, page traffic, virtual computer, virtual processor, and virtual memory. Terms whose meaning is important in this thesis are:

1. process: a sequence of information references.
2. states of a process: running, ready, page wait, blocked.
3. virtual time: time seen by a running process.
4. computation: a family of cooperating processes and information within the same name space.

We turn attention in the next chapters to the definition and characterization of the working set model for program behavior.

CHAPTER 3

The Working Set Model for Program Behavior

3.0. Introduction

We introduce and justify here the most basic concept in this thesis: the locality of information references. This is the property of program behavior that, during any interval of execution, the program favors a subset of its information. A working set of information dynamically measures this set of favored pages. A working set memory allocation strategy guarantees each running process that its working set shall be present in main memory. We shall show that working set strategies are optimum in two senses: minimum cost and minimum sensitivity to thrashing.

First, we will say that a strategy is optimum when it produces minimum cost (the product of memory space and time). After discussing various strategies, we show that working set strategies result in minimum cost. The proof is based on certain convexity properties, which follow from locality, of the cost function.

Second, we investigate the causes of thrashing, and show that working set strategies minimize the possibility of thrashing.

We conclude the chapter with a survey of the literature, best done in the light of the working set model.

3.1. Locality and Working Sets

3.1.1. Definition and Justification

Throughout this thesis we shall assume that locality is a fundamental property of program behavior. Locality is the property that, during any interval of execution, a process will favor some of its pages more than others; during disjoint virtual time intervals, the set of favored pages may be different. Put another way, if one observes a process's reference pattern for some virtual time interval, he will see that the process does not scatter its references uniformly across its information. There are at least five factors motivating this assumption:

1. Sequential instruction steams. Both programmers and compilers tend to organize sequentially the instructions that direct the activity of a process; this is especially true in single-address machines (i.e., those with a program counter). If a process fetches an instruction from a given page, it is highly probable that it will soon fetch another instruction, in sequence, from the same page.
2. Functional modularity. Program modules are organized and executed by function.
3. Content-related data organization. Information is usually grouped by content into segments, and is normally referenced that way; thus, references will occur in clusters to a content-related region in name space.
4. Looping. Programs often loop within a set of pages.
5. People. Realizing that their programs will run on a paged machine and that page transfers are costly, pro-

grammers tend to organize their algorithms so that activity is localized within subsets of their information. Moreover, people have been studying methods of minimizing interpage references at execution time; see references [B1,B2,C5,M1,O1,R1].

Experimental evidence suggests that this assumption, locality, is a very good assumption. Suppose to the contrary that, during every virtual time interval, a process scatters its references uniformly over its information. Suppose that a fraction s ($0 \leq s \leq 1$) of its pages have been placed in main memory. Let $\mu(s)$ be the fraction of its references the process makes to the set of pages <u>not</u> in memory; since the references are uniformly scattered, it follows that

$$\mu(s) = 1 - s$$

Experimental evidence, illustrated in Figure 3-1, contradicts this [B1,V1]. As measured, $\mu(s)$ actually follows some curve that lies below the curve $\mu(s)=1-s$. It has been observed that there is some number s_o and constant $k>1$, such that if $s \leq s_o$ then $\mu(s)=1-ks$; that is, the process is scattering its references uniformly over only a subset of its information. The numbers s_o and k depend on the particular program and the particular storage management rule used to decide what information is to reside in main memory.

We will therefore assume that locality is a property of program behavior.

We define the <u>working set</u> of information $W_p(t,\tau)$ of process p at time t to be the set of pages that process p has referenced during the virtual time interval $(t-\tau;t)$. The idea is illustrated in Figure 3-2.

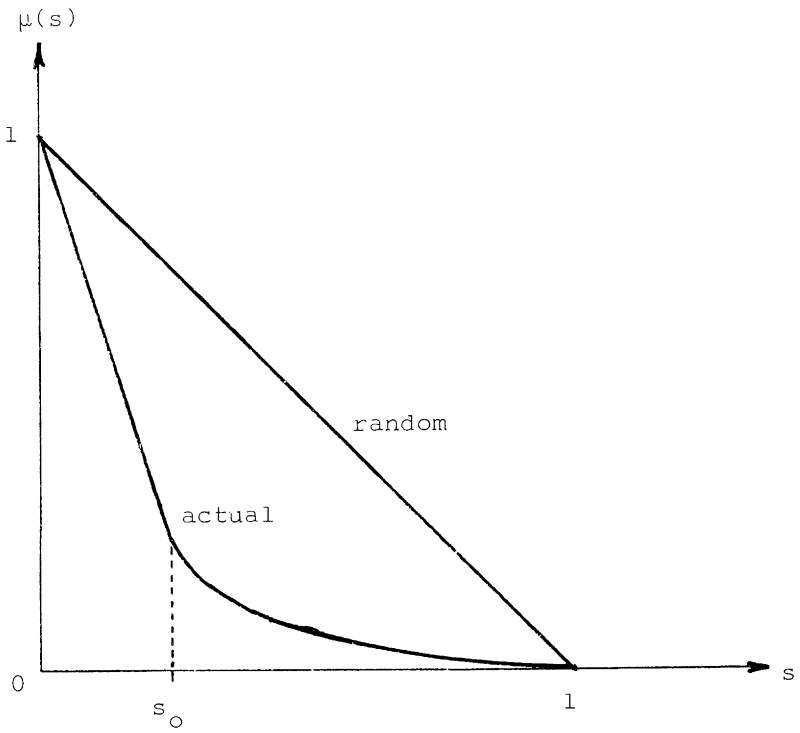


Figure 3-1. Evidence supporting locality.

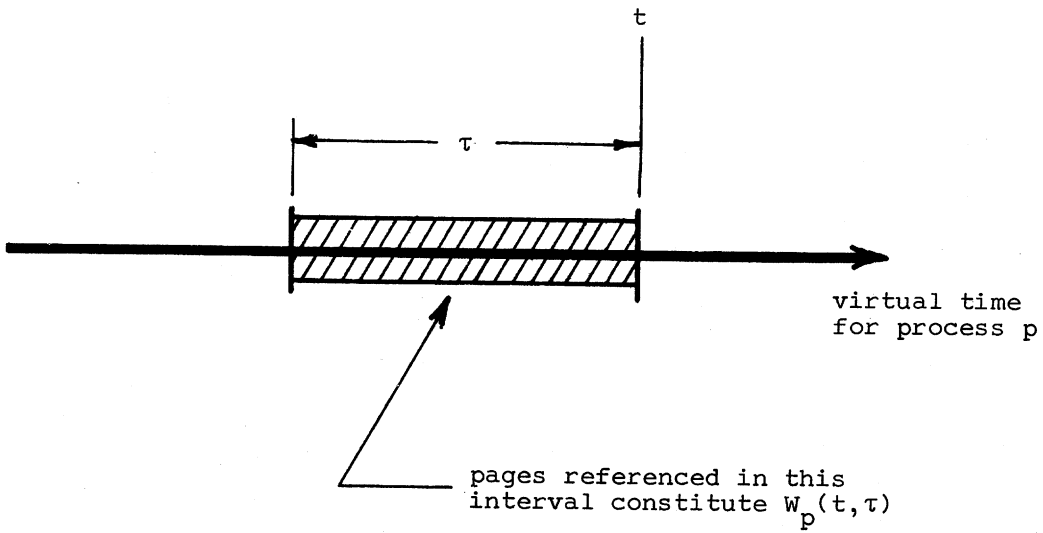


Figure 3-2. Definition of a working set.

The validity of the working set model rests on the concept of locality. A working set $W_p(t,\tau)$ measures the set of pages process p is favoring at time t. Assuming that process p is not likely to abruptly change its set of favored pages, the working set $W_p(t,\tau)$ constitutes a reliable estimate of p's immediate memory needs. To put it another way, we are assuming that, on the average,

$$\Pr[\text{page i referenced next} \mid i \in W_p(t,\tau)] > \Pr[\text{page i referenced next} \mid i \notin W_p(t,\tau)]$$

The working set parameter τ should be chosen as small as possible, and yet assure that $W_p(t,\tau)$ contains p's favored pages. Thus, τ may vary from program to program, and from time to time. We shall discuss details of choosing τ in Chapter 4.

We assume that the page size (i.e., the number of words in a page) is chosen small enough so that the working set $W_p(t,\tau)$ always consists of at least several pages. Indeed, if in a particular computer system we observed that working sets often consisted of only one or two pages, we would begin to suspect that a smaller page size might result in smaller working sets and in smaller memory requirements for programs.

Intuitively, a working set is the smallest set of information that ought to reside in main memory so that a process can operate efficiently. A working set memory management policy is one that permits a process to be running if and only if there is enough uncommitted space in main memory to contain its working set.

Define the random variable x to be the virtual time interval between successive references to the same page. These inter-reference intervals x are useful for describing certain program

properties, which we will do in detail in Chapter 4. Let $F_x(u) = Pr[x \leq u]$ denote its distribution function and let $\bar{x}$ deonte its mean. A working set is the collection of a process's pages whose current interreference intervals (in virtual time) satisfy $x \leq \tau$.

By a program we mean the set of information to which a process directs its references. There is a relation between the size of a program and the lengths of the interreference intervals to its component pages. Let process 1 be associated with program P_1 and process 2 be associated with program P_2, and let P_1 be larger than P_2. Then process 1 has to scatter its references across a wider range of pages than process 2, and we expect that the interreference intervals x_1 of process 1 will be longer than the interreference intervals x_2 of process 2. That is, P_1 bigger than P_2 implies $\bar{x}_1 > \bar{x}_2$.

3.1.2. Pictorial Representations

It is useful to develop some pictorial representations for the notions of working set and locality. Let C be a computation and $\underset{\sim}{M}$ be the name space used by C; we may imagine that elements of $\underset{\sim}{M}$ have been grouped together, by pages. We may associate with C a process space $\underset{\sim}{P}$ whose elements are the processes (sequences of information references) of C. If C is a single-process computation, $\underset{\sim}{P}$ contains just one sequence. If C is a multiprocess computation, $\underset{\sim}{P}$ contains several sequences. In Figure 3-3 we show a process p in $\underset{\sim}{P}$; the directed line suggests the ordering of the

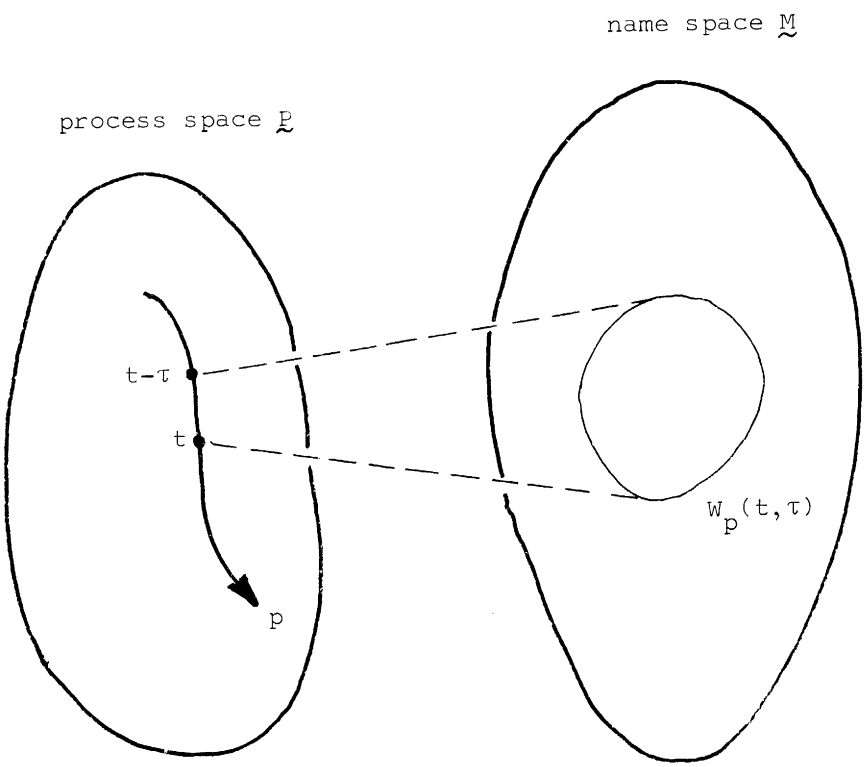


Figure 3-3. Association of working set with process.

information references constituting p; two of the information references, one at time t, the other at time $(t-\tau)$, have been singled out. We may imagine that $W_p(t,\tau)$ is a projection of the virtual time interval $(t-\tau,t)$ into $\underset{\sim}{M}$. Adjacencies indicated in $\underset{\sim}{M}$ (i.e., the content of $W_p(t,\tau)$)should not be construed as adjacencies of address values; they are simply adjacencies of references in virtual time.

Figure 3-4 depicts the assumption that the content of $W_p(t,\tau)$ is not fast-changing. For small time separations α, we expect a large intersection between $W_p(t,\tau)$ and $W_p(t+\alpha,\tau)$. For large time separations ß (with ß>>α and ß>>τ) we do not expect an intersection between $W_p(t,\tau)$ and $W_p(t+ß,\tau)$ because p has had ample opportunity to finish the work of time t by time (t+ß). Put another way, we expect a working set $W_p(t,\tau)$ to be a reliable estimate of p's memory needs only over a short interval.

Figure 3-5 illustrates the situation for a multiprocess computation C. Let P(C,t) denote the processes in C at time t that are running or in page wait (i.e., receiving the use of resources). The information that should be in main memory is

$$W_C(t,\tau) = \bigcup_{p\varepsilon P(C,t)} W_p(t,\tau)$$

Note that some of the working sets may overlap, because processes may share information.

Note further that P(C,t) may be regarded as a <u>working set of processes</u> in the process space $\underset{\sim}{P}$.

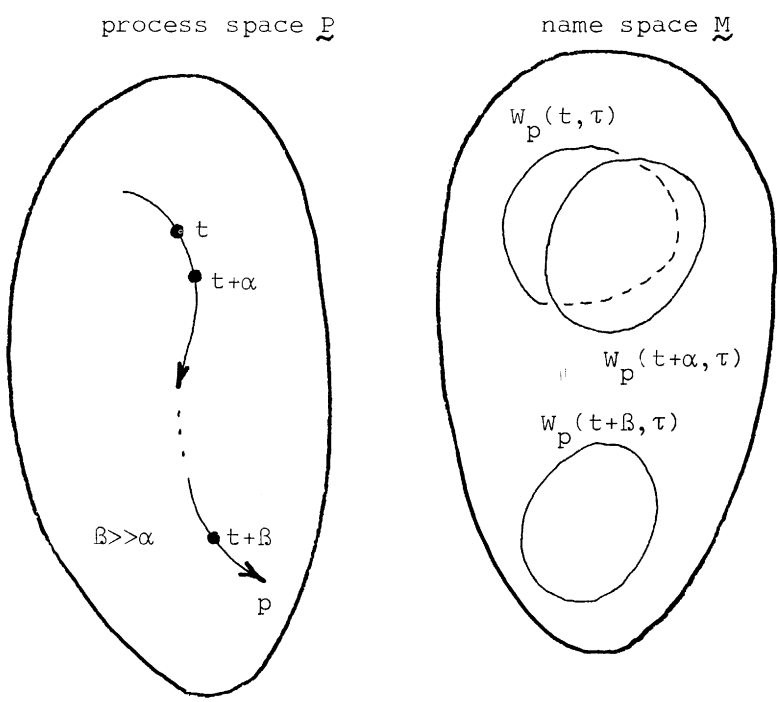
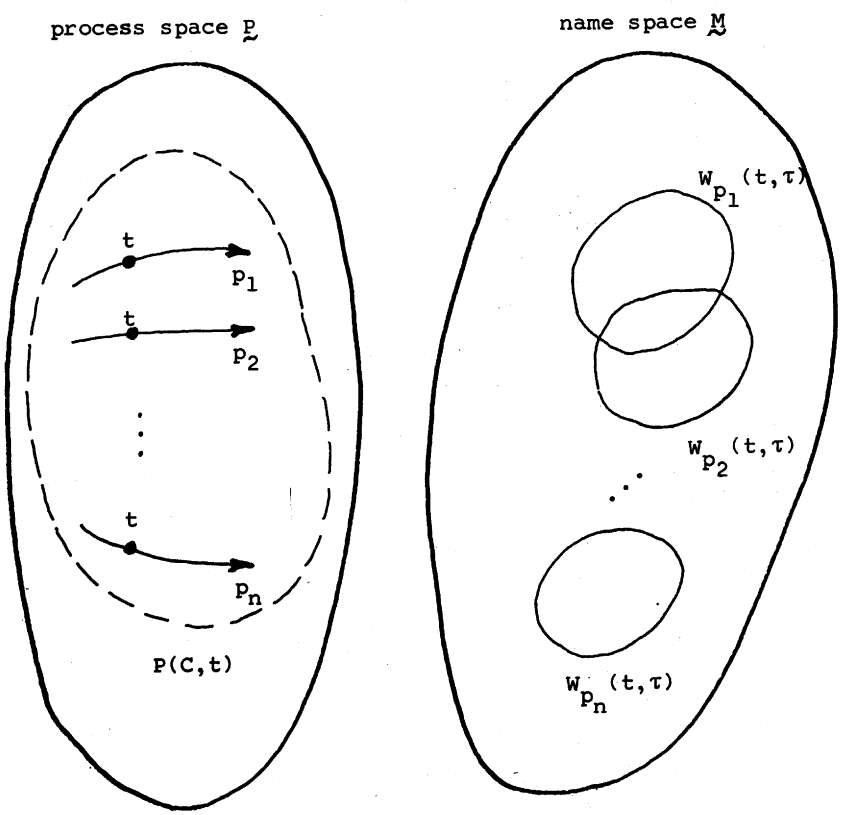


Figure 3-4. Time movement of a working set.



$$W_C(t,\tau) = \bigcup_{i=1}^{n} W_{p_i}(t,\tau)$$

Figure 3-5. Working sets for multiprocess computation C.

3.1.3. Interactions

The foregoing discussion deals with locality concepts during virtual time intervals that contain no interactions. An *interaction* is an instant in virtual time at which the process stops to wait for a message. What happens to our definitions if the virtual time interval contains an interaction?

When a process stops (blocks) for an interaction, it seeks a message or signal from another process, for a device, or from a user at a console. An interaction has two properties of interest to us:

1. The process enters the blocked state where it may remain, unpredictably, for a long time.
2. The message received by the process may affect its behavior following the interaction.

This second property means that, if t_i is an interaction instant, the working set $W_p(t_i,\tau)$ may not be a good estimate of any working set $W_p(t,\tau)$ for $t>t_i$, because the message may seriously alter p's behavior.

What we will do is assume that the working sets before and after an interaction intersect, though not completely. We believe that the expected size of the intersection will tend to decrease with long blocked-intervals, because in longer time intervals, for example, a user will have more opportunity to change his mind and alter the behavior of his process. Conversely, the shorter the duration of a blocked-interval, the greater the expected size of the intersection between the working set before and after the interaction.

An example is helpful. Figure 3-6 illustrates a program organization likely to be typical of modular, interactive programs. The user sends requests to the interface procedure A; having interpreted the request, A calls on one of the procedures $B_1,\ldots,B_n$ to perform an operation on the data D. The called B-procedure then returns to A for the next user request. Interactions occur whenever the process enters A to await a message. A program organization such as this might be used (for example) in an editing program. Just before the interaction, the working set will contain A, D, and one B-procedure. Just after the interaction, the working set will contain A. The intersection is just A.

A study of intersections of working sets before and after interactions is needed in order to assess the value of look-ahead when a process unblocks.

Because we are not interested in input-output allocation in this thesis, we will no longer be concerned with the effects of interactions on program behavior; from now on we assume that virtual time intervals contain no interactions. This problem has been studied in reference [D5].

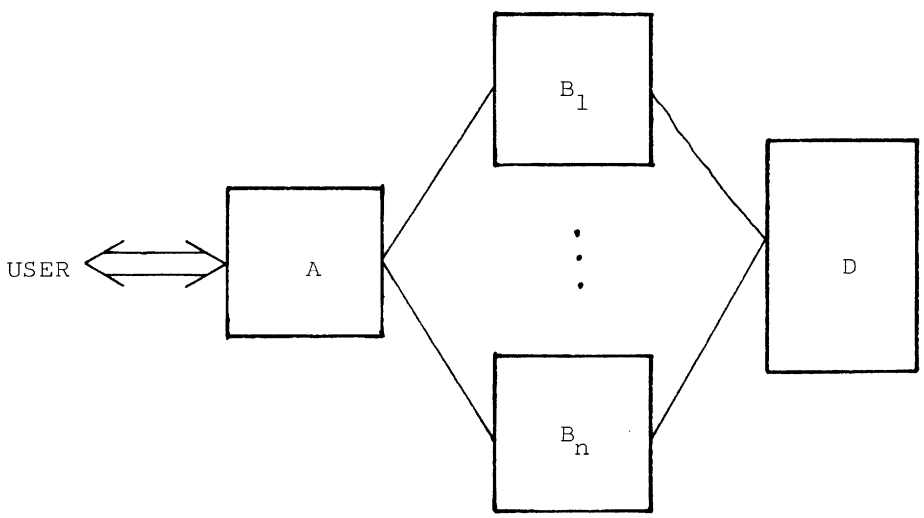


Figure 3-6. Organization of an interactive modular program.

3.2. Convexity [W2, p.563]

We shall prove a theorem about convex functions which will be of great importance in the following sections.

A function f(x) is strictly convex on an interval I if its second derivative is negative:

$$f''(x) < 0 \qquad x \in I$$

and f(x) is strictly concave if its second derivative is positive. If the second derivative is zero on an interval, we regard the function as being either convex or concave on that interval. Since f is convex if and only if -f is concave, we may restrict attention to convex functions. Figure 3-7 shows a strictly convex function; note that every line segment connecting two points on the curve lies below the curve.

Theorem 3.1. Suppose x is a random variable on an interval I, where its probability density function $p_x(u)$ satisfies

$$\int_I p_x(u)\, du = 1$$

$$\int_I u\, p_x(u)\, du = \bar{x}$$

Suppose also that f is a strictly convex function on I. Then

$$\overline{f(x)} \leq f(\bar{x})$$

and equaltiy holds if and only if $p_x(u) = \delta(u-\bar{x})$ (the impulse function).

Proof. Since $\bar{x}$ is a fixed number we may expand f(x) around the point $\bar{x}$ using Taylor's expansion:

$$f(x) = f(\bar{x}) + (x-\bar{x})\, f'(\bar{x}) + \frac{(x-\bar{x})^2}{2} f''(z) \qquad \text{some } z \in I$$

Since $f''(z)<0$ for $z \varepsilon I$,

$$f(x) \leq f(\bar{x}) + (x-\bar{x})f'(\bar{x})$$

Taking expectations on both sides, and noting that $f(\bar{x})$ and $f'(\bar{x})$ are constant,

$$\overline{f(x)} \leq f(\bar{x}) + \overline{(x-\bar{x})}\, f'(\bar{x})$$

but since $\overline{(x-\bar{x})}=0$, we have

$$\overline{f(x)} \leq f(\bar{x})$$

The equality clearly holds if and only if $x=\bar{x}$ with probability 1.

QED.

In Figure 3-7 we show a geometric interpretation of the theorem for the simple case

$$p_x(u) = \begin{cases} 1/2 & u = \bar{x}\pm\varepsilon \\ 0 & \text{otherwise} \end{cases}$$

and it is clear that

$$\overline{f(x)} = \frac{f(\bar{x}+\varepsilon) + f(\bar{x}-\varepsilon)}{2} \leq f(\bar{x})$$

Observe from the definition and the figure that, for f to be convex on an interval I, it is sufficient that

$$f(x+\varepsilon) + f(x-\varepsilon) \leq 2\, f(x)$$

for all choices of x and ε such that $(x+\varepsilon)$ and $(x-\varepsilon)$ are in I.

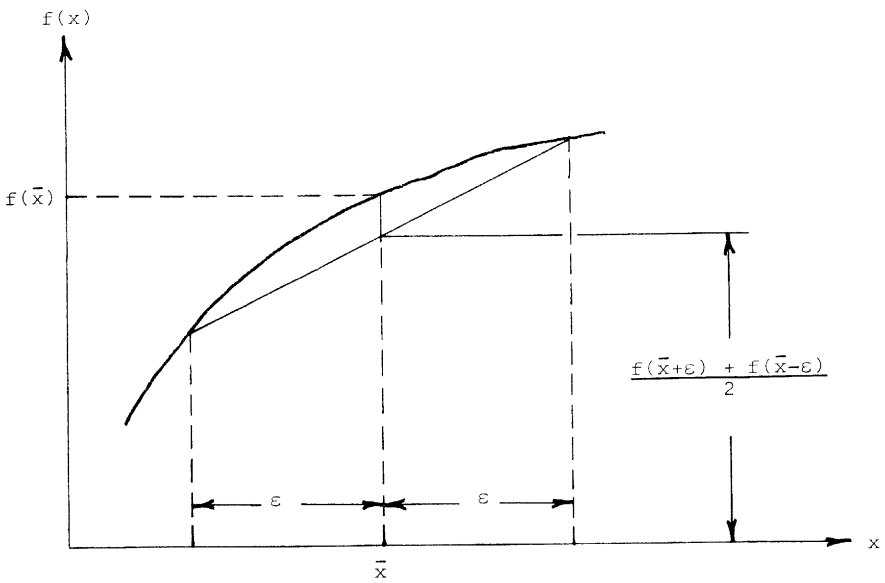


Figure 3-7. Illustrating convexity theorem.

3.3. Working Set Size

Let $W(t,\tau)$ be a working set. Define the working set size $\omega(t,\tau)$ to be:

$$\omega(t,\tau) = \text{Number of pages in } W(t,\tau) = |W(t,\tau)|$$

We assume that the working set size $\omega(t,\tau)$ is a stationary stochastic process. so that the time expectation $\overline{\omega(t,\tau)}^t$ is independent of t, and we may write

$$w(\tau) = \overline{\omega(t,\tau)}$$

where we understand overbar to mean time expectation.

Theorem 3.2. The expected working set size $w(\tau)$ has these properties:

1. $w(\tau) \leq \tau$
2. $w(0) = 0$
3. $w(\tau+\lambda) \geq w(\tau) \quad \lambda \geq 0$ (non-decreasing)
4. $w(\tau)$ is convex.

Proof: Since the maximum number of distinct references that can occur in τ vtu is τ, we have $\omega(t,\tau) \leq \tau$ and hence $w(\tau) \leq \tau$. That $w(0)=0$ is clear since no pages can be referenced in zero time. That $w(\tau+\lambda) \geq w(\tau)$ is also clear since more pages can be referenced in longer intervals. To show that $w(\tau)$ is convex, we will show that for all choices of τ and ε such that $(\tau-\varepsilon) \geq 0$,

$$2\,w(\tau) \geq w(\tau+\varepsilon) + w(\tau-\varepsilon)$$

So, let τ and ε be arbitrarily given, with $(\tau-\varepsilon) \geq 0$. Refer to Figure 3-8. Observe that

$$W(t,\tau) \cup W(t-\tau,\tau) = W(t,\tau-\varepsilon) \cup W(t-(\tau-\varepsilon),\tau+\varepsilon)$$

Using $|X \cup Y| = |X| + |Y| - |X \cap Y|$ for any sets X and Y, we have

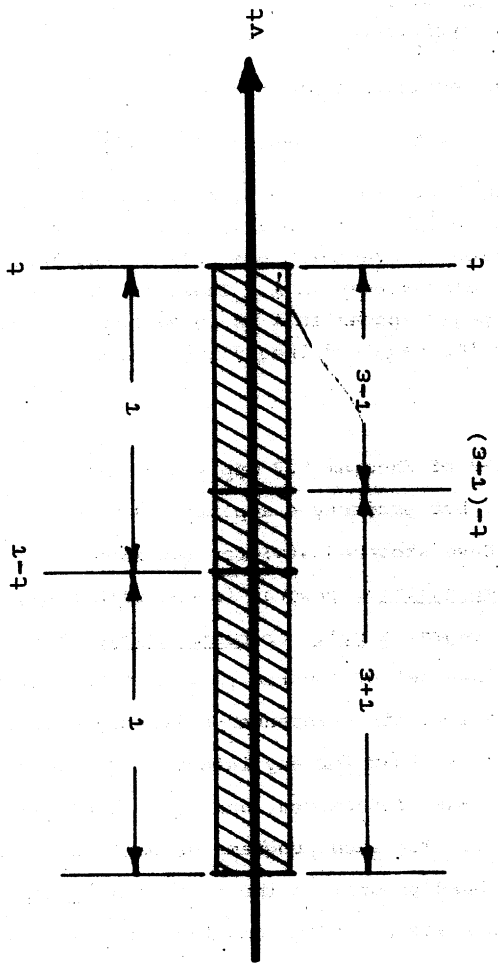


Figure 3-8. For the proof of Theorem 3.2.

$$\omega(t,\tau) + \omega(t-\tau,\tau) = \omega(t,\tau-\varepsilon) + \omega(t-(\tau-\varepsilon),\tau+\varepsilon) + |A| - |B|$$

where

$$A = W(t,\tau) \cap W(t-\tau,\tau)$$
$$B = W(t,\tau-\varepsilon) \cap W(t-(\tau-\varepsilon),\tau+\varepsilon)$$

taking the time expectation on both sides,

$$w(\tau) + w(\tau) = w(\tau-\varepsilon) + w(\tau+\varepsilon) + \overline{|A|} - \overline{|B|}$$

We claim $\overline{|A|} - \overline{|B|} \geq 0$. To see this, note that $|A| \leq w(\tau)$ and $|B| \leq w(\tau-\varepsilon)$, so $|A|$ is potentially bigger than $|B|$. During the operation of averaging over all t, any page that appears in B must also appear in A. Thus, on the average at least as many pages appear in A as in B; hence we have $\overline{|A|} - \overline{|B|} \geq 0$ and the required inequality follows.

QED.

Note that properties 1-3 of Theorem 3.2 apply also to the random variable $\omega(t,\tau)$ itself, but property 4 applies only to $w(\tau)$.

In Figure 3-9 we have sketched $w(\tau)$ for two kinds of program. A hard, or incompressible, program is one with a well-defined set of favored pages; a soft, or compressible program is one with a fuzzily-defined set of favored pages. A hard program tends to scatter most all of its references uniformly over some set of τ_o favored pages, so that for any interval $\tau \leq \tau_o$ we expect to see mostly distinct pages referenced, and $w(\tau)$ increases (almost) linearly with τ. For such programs we want to choose $\tau \geq \tau_o$. An example of a hard program is the so-called stream-processing program, whose algorithm is contained wholly in a set of τ_o pages, and occasional references are made to a sequence of data pages; after only a few references, each data page is discarded forever. If $\tau >> \tau_o$, the working set will contain many

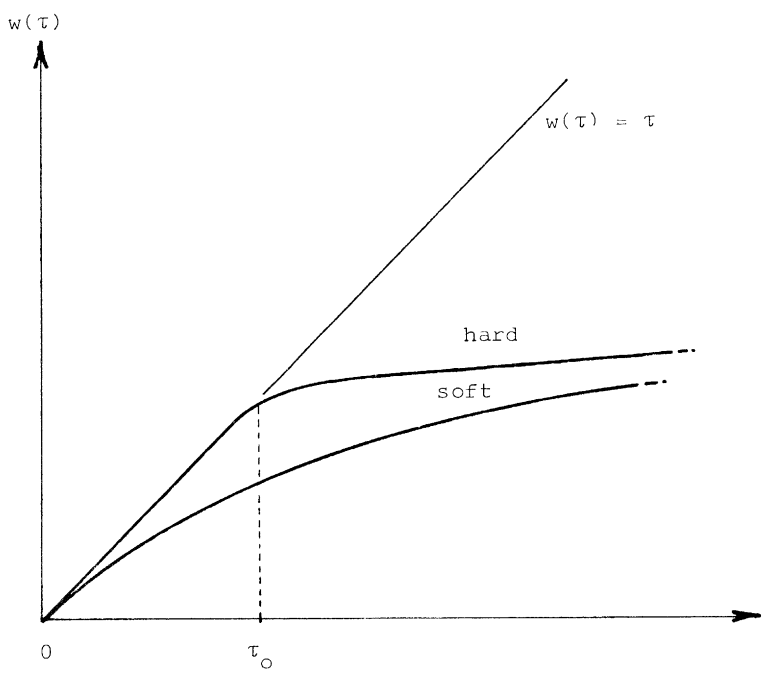


Figure 3-9. Expected working set size.

useless pages. Choosing τ close to τ_o will not affect the program's operating efficiency, but will diminish the amount of memory it occupies.

Recall the definition of the interreference intervals x (the vt intervals between successive references to the same page), with distribution function $F_x(u)=Pr[x\leq u]$, and density function $f_x(u)=\frac{d}{du}F_x(u)$. We shall assume $F_x(u)$ is convex. This is not unreasonable since it requires only that $f_x(u)$ be decreasing, which is consistent with the concept of locality. In fact, there is strong evidence that that this type of interarrival distribution is modelled nicely by a hyperexponential distribution [C4,F4], which is convex.

We define the missing-page probability $\lambda(\tau)$ to be the probability that a process directs its next reference to a page not in the working set $W(t,\tau)$; under a working set memory allocation strategy, such a page may be missing from main memory.

Theorem 3.3. Let $\lambda(\tau)$ = Pr[process references a page not in $W(t,\tau)$]. Then $\lambda(\tau) = 1-F_x(\tau)$.

Proof: The probability the page referenced is not in $W(t,\tau)$ is just the probability its most recent interreference interreference interval satisfies $x>\tau$, so $\lambda(\tau)=Pr[x>\tau]=1-F_x(\tau)$.

QED.

We will need the following two theorems to prove that a working set strategy is optimum.

Theorem 3.4. Suppose τ is varied on some interval, with mean $\bar{\tau}$. Then the average missing-page probability is increased:

$$\overline{\lambda(\tau)} \geq \lambda(\bar{\tau})$$

Proof: Since we assume $F_x(u)$ is convex, $\lambda(\tau) = 1-F_x(\tau)$ is concave, and by Theorem 3.1, we have $\overline{\lambda(\tau)} \geq \lambda(\bar{\tau})$.

QED.

Theorem 3.5. Suppose τ is varied on some interval, with mean $\bar{\tau}$. Then the average working set size is decreased:

$$\overline{w(\tau)} \leq w(\bar{\tau})$$

Proof: By Theorem 3.2, $w(\tau)$ is convex. By Theorem 3.1, we have $\overline{w(\tau)} \leq w(\bar{\tau})$.

QED.

Varying τ increases the probability that a missing page will be referenced, as well as diminishing the average memory share held by a process. That is, varying τ with mean $\bar{\tau}$ on some interval has the same effect as holding τ fixed at some $\tau_o < \bar{\tau}$ such that $w(\tau_o) = \overline{w(\tau)}$.

3.4. Storage Management Policies

Storage management policies for multiprogrammed memories may be regarded as operating in two provinces:

1. Fetching (page in): Locate the required page in auxiliary memory, and load it into main memory; turn the in-core bit of the corresponding page table entry ON.
2. Replacement (page out): Remove some page from main memory, turn the in-core bit of the corresponding page table entry OFF. The policy rule that decides which page to remove is called the replacement rule.

Management algorithms may be classified according to their methods of fetching and replacement.

Fetch strategies may load pages before they are needed (pre-paging), at the moment they are needed (demand paging), or even later. Many strategies use demand paging; that is, no action is taken to bring a page into main memory until some process attempts a reference to it. Demand paging is usually preferred to pre-paging because it is much cheaper to implement, and because it is not clear that pre-paging improves performance significantly. As we have stressed, advance information is often non-existent because there is no reliable source of allocation information. In fact the only major argument favoring pre-paging is the possibility of moving large contiguous blocks of pages from auxiliary memory so that the accumulated traverse time is reduced in the long run. Although traverse time reduction is (in some sense) a valid argument for pre-paging, we feel that it is also a more powerful argument for better, faster, auxiliary memories.

It may be argued that a working set, a supposedly reliable estimate of a process's immediate memory needs, is the ideal

set of pages to pre-load. Because the records required to keep track explicitly of which pages belong to which working sets may easily become so complicated that any benefits resulting from pre-paging may be lost, we prefer to assume that fetching occurs on demand only, via the page fault mechanism.

The major problem in memory management is not deciding which pages to load; it is deciding which pages to replace[1]. A storage management policy should attempt to keep in main memory the pages most likely to be used. Thus, the best choice for replacement is the page with the least likelihood of being reused immediately. Debate has arisen over which replacement, or page-turning, strategy is best.

The cost of operating a program under a given strategy will be defined (Section 3.5.1) to be the amount of memory used times the duration of such use. We will say that the optimum strategy is the one that results in the lowest cost. In Section 3.5.1 we will show that low missing-page probability is equivalent to low cost. We shall therefore use the missing-page probability as a measure of performance for a paging policy; this will be done in Sections 3.5.2 and 3.5.3.

---

[1] If a page has been modified since being placed in main memory, replacing it involves transferring it into auxiliary memory; an unmodified page is simply overwritten, provided there is a copy in auxiliary memory.

Allocation of pages in multiprogrammed memories can be handled on either a fixed or variable memory basis:

1. Fixed share. Before being run, a program is granted a share of the memory for its private use.
2. Variable share. Programs are allowed to compete freely for memory space. In principle, more aggressive programs should be able to obtain a greater share of the memory. In principle, as a program expands or contracts, its share increases or decreases accordingly.

In Section 3.5.2 we shall prove that variable-share strategies yield smaller missing-page probabilities than fixed-share strategies, all other things being equal. Policy rules for replacement (which may be used with either fixed or variable share basic strategies) fall into the following three classes, ordered in terms of the intrinsic increase in the logic required to implement:

1. Static rules, which use no information about page use; these rules are very simple to implement.
2. Usage rules, which use information about page use, generally measuring time intervals since the last reference to each page.
3. Demand rules, which attempt to predict, on the basis of recent reference patterns, the set of pages most likely to be used immediately. A program is given more or less space according to its demand for space.

We shall show in Section 3.5.2 that the static rules lead to the highest missing-page probabilities, the demand rules the lowest.

There are two static rules of interest:

1. Random (RAND). Whenever a fresh page of memory is needed, a page is selected at random to be replaced. Implementation is simple, requiring only a random-number generator.
2. First-in, First-out (FIFO). Whenever a fresh page of memory is needed, the page least recently paged in is retired and another page brought in to fill the newly vacated slot. Whereas RAND requires a random number generator, FIFO requires only a counter, and implementation is even simpler, as follows. The pages of main memory are regarded as a cyclic group; suppose the M pages of main memory are numbered $0,1,\ldots,(M-1)$ and a pointer k indicates that the k^{th} page was most recently paged in. When a fresh page is needed, $[(k+1) \bmod M] \rightarrow k$, and page k is retired.

The principal argument for these two rules is their simplicity of implementation. Yet the experimental evidence [B1,B2,V1] indicates that usage rules, despite higher overhead, significantly outperform the static rules.

There are two usage rules of interest, LRU and FINUFO:

3. Least recently used (LRU). Whenever a fresh page of memory is needed, the page unreferenced for the longest time is removed. Each page table entry contains a use bit, set ON each time the page is referenced. At periodic intervals, all page table entries are searched, use bits reset, and usage records updated.

Unfortunately, implementation of an LRU rule may become complicated, and it is not clear whether an overall improvement would

result. A very interesting rule[1] combines the simplicity of FIFO with the sophistication of LRU:

4. First-in, Not-used, first-out (FINUFO). Implementation is almost exactly that of FIFO, but now the use bits come into play. Let k be the pointer that cycles through the M pages of memory. Whenever a fresh page is needed, k is incremented until a page is found with use bit OFF; this page is retired. When k passes a page with use bit ON, the use bit is turned OFF.

It is interesting to note that FINUFO is much closer to a demand rule than to a usage rule, because when demand for main memory is high, FINUFO will have difficulty in finding a page to remove (many use bits ON). Whereas FIFO and LRU will always find a page to remove, FINUFO may not. It is therefore a stable rule.

Another usage rule, primarily of academic interest, is:

5. ATLAS loop-detection. The Ferranti ATLAS computer [K1] had a paging strategy that attempted to detect loop behavior in page reference patterns, then minimize page traffic by maximizing the time between page transfers; that is, by removing pages not expected to be needed for the longest time. Performance was satisfactory for programs exhibiting loop behavior; unsatisfactory for programs exhibiting aperiodic reference patterns, because the algorithm attempted to predict loops when there were none. Implementation was costly.

Two kinds of demand rules warrant investigation, biased rules and working set rules:

---

[1] Reported (by J. H. Saltzer) to be used in Multics.

6. Biased replacement rules. In round-robin fashion, each program is favored for an interval of time. During its favored interval, none of its[1] pages are removed and it may acquire new pages without hindrance. After its favored interval, it will be forced to give up pages in deference to other programs. When a page is to be retired, any of the rules discussed above may be applied to the non-favored pages.

Belady [B2] reports that a biased FIFO rule on the M44/44X computer improved performance significantly. The arguments given in Section 3.5.4 may be used to show that biased rules will perform better than non-biased rules (except the working set rule). Intuitively this makes sense, because large programs will have opportunity to expand into memory shares more matched with their needs.

7. Working set (WS). Guarantees that a computation receives the use of processor if and only if there is enough uncommitted space in memory to contain its working set pages. Thus, every page belonging to the working set of some running process must be kept in main memory. Pages in no working set are subject to removal, though need not be removed until the space is needed. A computation acquires more or less memory in accordance with fluctuations in its working set size. Should the totality of working sets exceed memory, some program (perhaps the one present there for the longest time) is removed in order to clear space.

---

[1] This complicates implementation, because now identification of pages by program is required.

These seven are a portfolio of the most interesting and most important rules.

In the next section we will compare all these strategies and show that WS is optimum.

3.5. Working Set Strategies are Optimum

The following demonstration that working set strategies are optimum is based on the concept of locality, because we shall rate strategies by their ability to retain a process's favored pages in main memory.

First we show that the missing-page probability is a valid measure of performance for a paging policy. Then, using the convexity properties of working set size and missing-page probability (Theorems 3.4 and 3.5) we show that the working set strategies have the lowest missing-page probabilities. For ease in discussion, we start by studying the algorithms operating on one program in a cramped memory. We then generalize to the case that many programs reside together in memory.

3.5.1. The Cost of a Strategy

Suppose h(t) is the number of pages of memory held by a certain program at time t. Define the cost C(I) for memory usage over the real time interval I to be

$$C(I) = \int_I h(t)\, du \tag{3.5.1}$$

which is the space-time product of memory usage. We will say that the best strategy is the one that produces minimum cost. C(I) includes page wait times in the interval I; even though the process is not running, its information still occupies space during page waits.

For convenience, we shall deal with the cost per unit virtual time, G, which we define to be

(3.5.2) $$G = \frac{C(I)}{v(I)}$$

where v(I) is the amount of virtual time contained in I. We are interested in the paging policy with smallest G.

We consider a certain program consisting of r pages, which is operating in a space of s memory pages. The missing-page probability $\mu(s)$ is the probability that the process references a page not in main memory, when s pages are in main memory; clearly, $\mu(s)$ depends on the paging algorithm[1].

It is important to note that $\mu(s)$ is an average. To measure $\mu(s)$ experimentally, one would run a program in a space s, using a given paging algorithm, for a virtual time interval of length V. If he observed R references to pages not in memory, he would assign $\mu(s)=\frac{R}{V}$. Thus, $\mu(s)$ is also the rate at which page faults occur, for in a virtual time interval of length V, we expect $V\mu(s)$ page faults.

We have sketched $\mu(s)$ in Figure 3-10 for two strategies, which we shall call 1 and 2 (cf. Figure 3-1).

The cost per unit virtual time G(s) of a strategy, as a function of the number s of pages in main memory, is related to $\mu(s)$ as follows. Suppose the program has executed for V vtu, and suppose $\mu(s)$ is constant over this interval. The expected number of page waits is $V\mu(s)$, and so the total elapsed real time is

(3.5.3) $$t_r = V + V\mu(s)T = V(1+\mu(s)T)$$

where each page wait costs one traverse time T. The memory

---

[1] When the missing-page probability is a function of memory space s, we will write it as $\mu(s)$. When it depends only on the work-set parameter τ, we will write it as $\lambda(\tau)$.

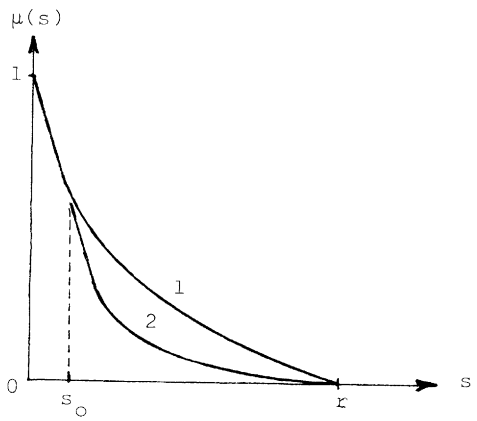


Figure 3-10. Missing-page probability for two strategies.

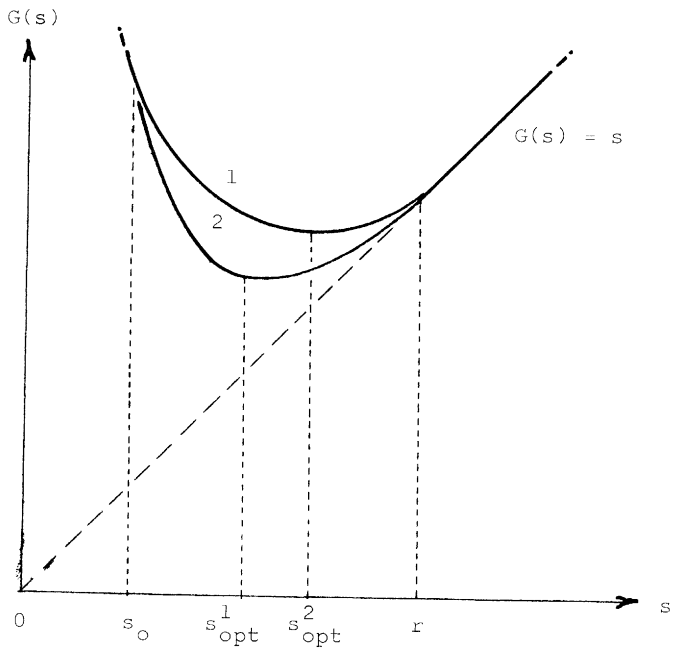


Figure 3-11. Cost per unit virtual time for two strategies.

space s is constant for this interval, so the cost is

(3.5.4) $$s\,t_r = sV(1+\mu(s)T)$$

and the cost per unit virtual time is

(3.5.5) $$G(s) = \frac{st_r}{V} = s(1+\mu(s)T)$$

We have sketched G(s) for the two strategies, 1 and 2, in Figure 3-11. Because $\mu(s)$ for strategy 2 is flatter than for strategy 1, the optimum memory size s_{opt} is smaller for strategy 2 than for strategy 1. Moreover, because

(3.5.6) $$\mu_2(s) \leq \mu_1(s)$$

it follows that

(3.5.7) $$G_2(s) \leq G_1(s)$$

We therefore obtain two important conclusions. First, the smaller the average missing-page probability, the cheaper is the policy. Missing-page probability is therefore a valid performance measure. Second, the smaller the average missing-page probability, the smaller is the optimum memory space s_{opt}. Hence, under better strategies, more programs can be placed in memory.

If the working set parameter τ is properly chosen (Chapter 4), it is possible to cause a working set strategy to operate at or near its current value of s_{opt}.

3.5.2. Single Program Case

Imagine an experiment (depicted in Figure 3-12) in which the same program is executed in memories A, B, C, and D; memory A uses RAND replacement, B uses FIFO, C uses LRU, and D uses FINUFO. Let $\mu(s)$ denote the missing-page probability when a fraction s ($0 \leq s \leq 1$) of the program is in memory. We claim that

$$\mu_A(s) \geq \mu_C(s)$$
$$\mu_B(s) \geq \mu_C(s)$$
$$\mu_D(s) \geq \mu_C(s)$$

Since we assume locality is a basic program property, the question is: How well do RAND, FIFO, LRU, and FINUFO keep a program's favored pages in memory?

To answer the question we imagine that we are trying to measure $\mu_A(s)$, $\mu_B(s)$, $\mu_C(s)$, and $\mu_D(s)$ by observing the rate at which page faults occur.

Since a process references its favored pages most often, we expect that the least recently referenced pages in memory are the least favored; thus, LRU tends to retain favored pages. RAND may very easily select a favored page, even one that LRU would not; thus, we expect RAND to induce more page faults over an execution interval than LRU, and so $\mu_A(s) \geq \mu_C(s)$. Under FIFO, it is certain that every page will eventually be removed; thus, we expect FIFO to induce more page faults over an execution interval than LRU, and so $\mu_B(s) \geq \mu_C(s)$.

We make no claim that one or the other of RAND and FIFO is better. On the one hand, there are cases in which RAND is better than FIFO (e.g., an unchanging set of favored pages --

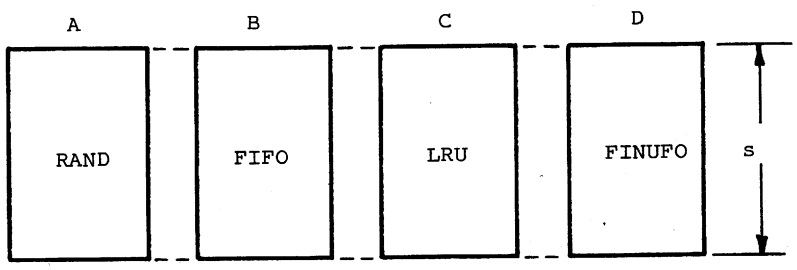


Figure 3-12. Conceptual experiment to compare strategies.

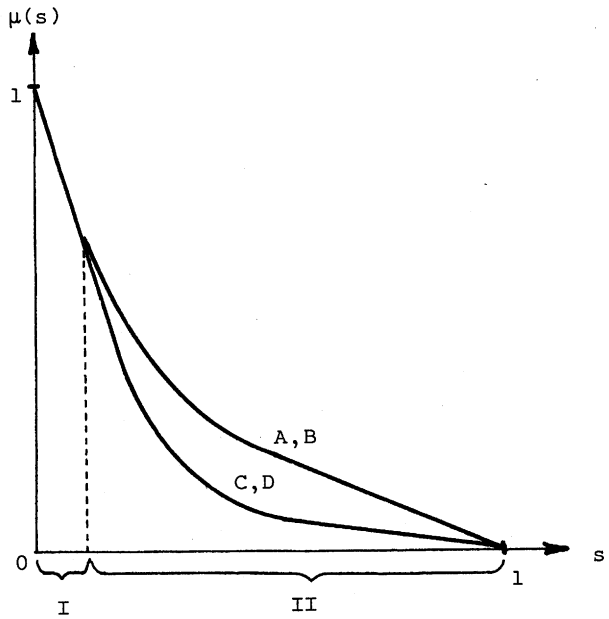


Figure 3-13. Missing-page probabilities for the strategies.

FIFO eventually removes every one of them, whereas RAND may not). On the other hand, there are cases for which FIFO is better than RAND (e.g., a process which changes its set of favored pages completely before FIFO completes a cycle -- FIFO removes the old pages first, whereas RAND may select some of the new favored pages). FIFO, of course, is cheaper to implement.

The FINUFO algorithm operates nearly the same as LRU, there being little difference, except in cost of implementation. If all the use bits have been set, FINUFO will do worst than LRU for the following reason. On its first cycle through memory, FINUFO finds all use bits ON, and clears them; since the process is in page wait, the use bits remain OFF, so on its second cycle through memory, FINUFO will select the first page whose use bit it cleared on the first cycle. Thus, FINUFO essentially selects a page at random. Assuming correlation between age and usefulness, we expect that there are situations in which LRU induces fewer page faults during an execution interval than FINUFO, and so on the average $\mu_D(s) \geq \mu_C(s)$.

In Figure 3-13 we show $\mu(s)$ sketched for memories A, B, C, and D in our conceptual experiment. In Region I, all three policies behave equally poorly, because too few pages are in memory. In Region II, the differences become apparent. At s=1, all three policies are again the same, since the program is entirely present in memory.

To complete the discussion, we must show that WS is better than LRU.

We imagine a second experiment, to compare LRU and WS, shown in Figure 3-14. Memory A, of size M, is run under LRU. Memory B, of variable size, runs under WS with τ fixed at τ_o such that the average working set size is $w(\tau_o) = M$. Note that LRU and WS are very similar in operation: LRU keeps the M most recently used pages in memory, whereas WS keeps the $\omega(t,\tau_o)$ most recently used pages in memory.

Figure 3-15 compares the behavior of the two policies. Figure 3-15a shows τ fixed at τ_o; at two times t_1 and t_2 the working set size is $\omega(t_1,\tau_o) = w_1$ and $\omega(t_2,\tau_o) = w_2$, and so the size of memory B varies at least over the range (w_1,w_2). Figure 3-15b shows that memory A does not vary, is fixed at M. Hence at the times t_1 and t_2 memory A is operating at τ_1 and τ_2, respectively. That is, we may hold the working set size fixed at M by varying τ so that the working set is always exactly contained in memory A.

Thus, memory A is simulated by a working set strategy with τ varying around mean τ_o on the range (τ_1,τ_2), and memory B has τ fixed at τ_o. Writing the missing-page probability for WS as

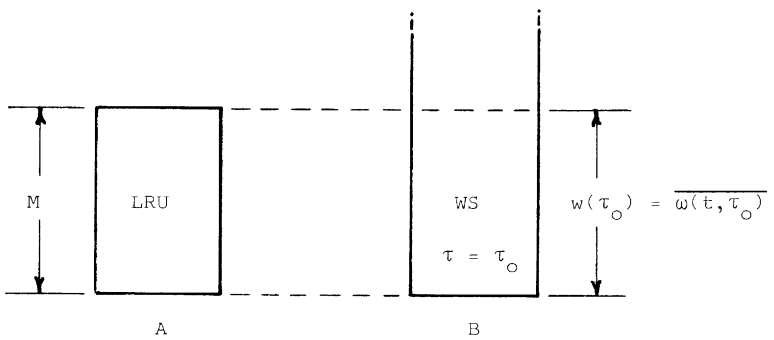


Figure 3-14. Experiment to compare LRU and WS.

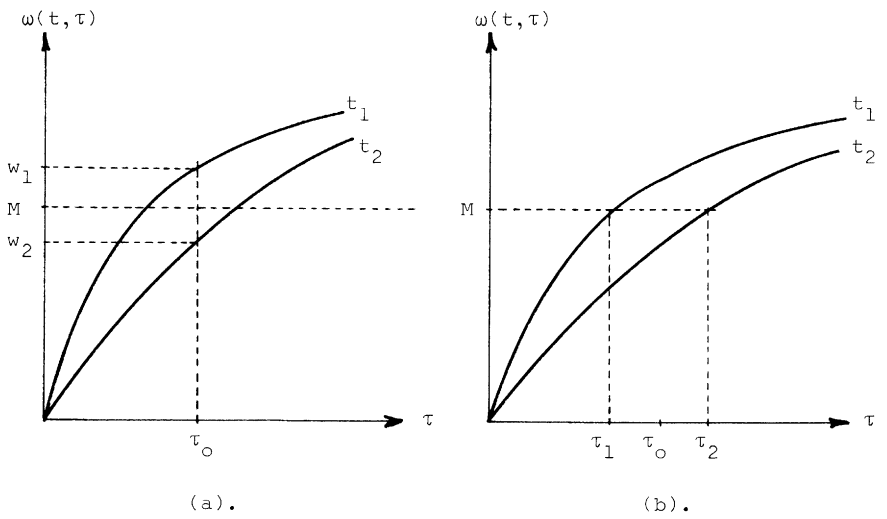


Figure 3-15. Comparison of LRU and WS.

$\lambda(\tau)$, and recalling Theorem 3.4,

$$\mu_A(M) = \overline{\lambda(\tau)} \geq \lambda_B(\tau_o)$$

and so WS is at least as good as LRU.

Intuitively this makes sense. Suppose the program is constant at size M throughout execution except for a single reference to the $(M+1)^{st}$ page. If it is in memory A, the reference to the $(M+1)^{st}$ page displaces some other page in A, which must be recalled.

Note that we have also shown that a variable size share of memory is superior to a fixed size share of memory. This makes sense since:

1. As we have stressed, advance knowledge of program size is often non-existent, and indeterminable. The share cannot be chosen optimally in advance of execution.
2. If each program gets a fixed share of memory, we cannot guarantee that memory is densely packed with the most useful information. A small program operating in too large a space is occupying space it does not need, space which could and should be given over to a large program operating in too small a space. Using variable shares permits allocating space on the basis of need.

If the programs in question do not satisfy locality, the arguments above fall apart. Consider, for example, the case of an (n+1)-page program which cycles endlessly through the (n+1) pages; operate this program in a memory of size n. Clearly, the least recently used page is the one about to be referenced, so LRU makes the worst possible decision. Similarly, FIFO and FINUFO

remove a page just before it is referenced. Only RAND has non-zero probability of not making a mistake, and is the best of the four. If the working set parameter is $\tau \leq n$, the working set never contains the page next to be referenced, and so WS is poor in this case too.

3.5.3. Multiprogrammed Case

How do programs interact with each other, if at all, under each of these strategies? How can the memory demands of one program interfere with the execution of another? We can obtain answers to these questions by examining the missing-page probability.

The missing-page probability μ is the probability that a process makes a reference to a page not in main memory. In the multiprogrammed case, we expect it to be a function of the program size r (r is the number of pages in the program), of the number n of programs simultaneously resident in main memory, and on the main memory size M:

(3.5.8) (missing-page probability) = $\mu(n,r,M)$

In the following discussion we assume that locality is a basic behavior property.

Suppose there are n programs in main memory; intuitively we expect that if the totality of working sets does not exceed the main memory size M, then no program loses its favored pages

to the expansion of another[1]. That is, as long as

$$(3.5.9) \qquad \sum_{i=1}^{n} \omega_i(t,\tau_i) \leq M$$

there will be no interaction among programs, and we expect the missing-page probability to be small. But when n exceeds some critical number n_o, the totality of working sets exceeds M, the expansion of one program displaces the favored pages of another, and so the missing-page probability increases sharply with n. Thus, we have

$$(3.5.10) \qquad \mu(n_1,r,M) > \mu(n_2,r,M) \qquad \text{if } n_1 > n_2$$

This is illustrated in Figure 3-16. In other words, it costs more to operate a program in a crowded memory than to operate it in a roomy memory.

If a paging algorithm operates in the range $n > n_o$, we will say it is <u>saturated</u>.

Next we want to show that the RAND, FIFO, LRU, and FINUFO algorithms have the property that

$$(3.5.11) \qquad \mu(n,r_1,M) > \mu(n,r_2,M) \qquad \text{if } n>n_o \text{ and } r_1>r_2$$

That is, a large program is more likely to lose pages than a small program, when the algorithm is saturated. Put another way, it costs more per page to operate a large program in a crowded memory than to operate a small program in a crowded memory.

To see that this is true for RAND, observe that a large program occupies more space in memory than a small program, and so

---

[1] Though it may lose favored pages because of foolish decisions by the replacement rule; for example, RAND or FIFO.

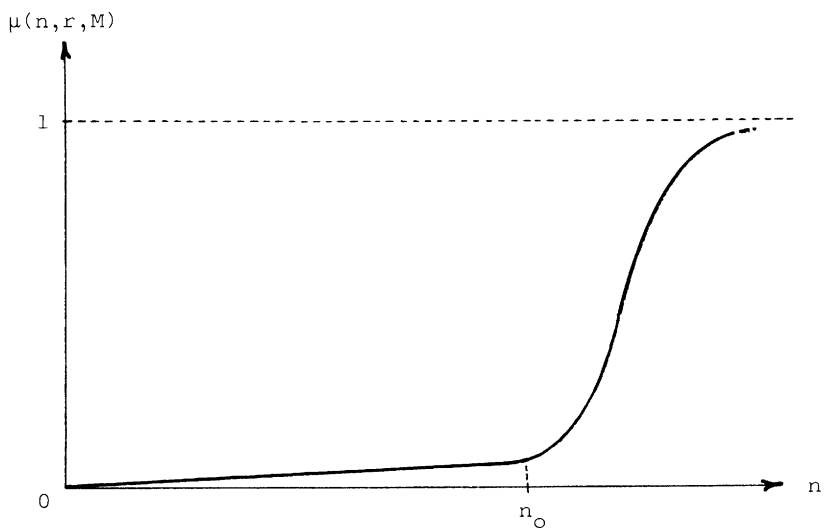


Figure 3-16. Behavior of missing-page probability.

has more pages as candidates for random selection to choose from. To see that this is true for FIFO, observe that a large program tends to execute longer than a small one, and is thus more likely to be still in execution when FIFO gets around to replacing its pages. To see that this is true under LRU, recall that if program P_1 is bigger than P_2, then the interreference intervals satisfy $\bar{x}_1 > \bar{x}_2$ -- the large programs are the ones that tend to reference the least recently used pages. To see that this is true for FINUFO is more difficult. If $n > n_o$, all the use bits are ON, until some program stops for a page wait; since it can no longer set its use bits, such a program will tend to lose all its pages. Inasmuch as large programs have more pages, a large program will suffer more when it enters page wait.

By definition, a WS algorithm makes the missing-page probability independent of n and M, since eq. 3.5.9 is assumed to be satisfied. In fact, Theorem 3.3 shows that the missing-page probability depends only on τ in this case: $\lambda(\tau) = 1-F_x(\tau)$.

Thus, the RAND, FIFO, LRU, and FINUFO policies result in higher costs when the memory is crowded. By avoiding crowding, WS results in lower cost.

If we ask the question: How well does each strategy fare in keeping the working set of a process in memory? We again see that FIFO and RAND are worse than LRU, which in in turn comparable to FINUFO. If we regard the entire memory contents as a large multiprocess computation, the same arguments of the preceding section show that WS results in lower missing-page probability than LRU. If FINUFO is kept away from saturation, it should perform nearly as well as WS. (FINUFO is nearing sat-

uration when it cycles once through the memory in a time comparable to the traverse time T.)

It might be argued that our comparison of LRU and WS (at least in the single program case) is not strictly valid because WS operates with surplus memory. That is, a larger memory is needed in order to provide buffer space to absorb working set expansions. This is quite true of one program only is using memory. In the multiprogrammed case WS shows its superiority:

1. WS makes programs independent in the sense that the expansion of one program cannot displace working set pages of another.
2. When the size of the memory becomes large, the fractional requirement for buffer space to absorb working set expansions becomes small. This is shown in Chapter 8.
3. If τ is properly chosen, each program operates in the vicinity of its optimum cost size (s_{opt} in Figure 3-11) -- thus it is possible to fit more programs cheaply into memory under a WS strategy than under any other strategy.

3.5.4. Use of Biased Replacement Rules

Belady has shown that biasing the FIFO rule (see Section 3.4) on the M44/44X computer improved performance significantly [B2]. We wish to show that this is true in general: by slowly varying the memory share of a program the probability of referencing a

missing page is reduced. A WS strategy is still superior because it varies the memory share exactly in accordance with need, whereas the biased rules do not guarantee that the memory share enlarges at the times the program would like to see it enlarged.

We shall show that biasing the LRU rule improves performance, but not to the point of WS. Since LRU is better than RAND or FIFO, it follows that biasing RAND or FIFO produces corresponding improvements. Since a non-saturated FINUFO rule behaves very much like a WS rule, there is little point to biasing it.

To show biasing improves LRU we shall show biasing increases the average value of τ for each value of the random variable $\omega(t,\tau)$. Suppose $\omega(t,\tau)$ is known; choose $\tau=\tau_o$ such that $\tau_o = \omega^{-1}(t,M)$, where M is the memory size and ω^{-1} is the inverse function of $\omega(t,\tau)$ with respect to τ. Now let the memory size be s, and let s vary such that $\bar{s}=M$. Since $\omega^{-1}(t,\tau)$ is concave (see Figure 3-9), we have from Theorem 3.1

$$\overline{\omega^{-1}(t,s)}^{s} \geq \omega^{-1}(t,M)$$

that is,

$$\bar{\tau} \geq \tau_o$$

Since the average value of τ has been increased, the missing-page probability $\lambda(\tau)$ must decrease:

$$\lambda(\bar{\tau}) \leq \lambda(\tau_o)$$

Of course, since the memory variation is out of phase with the variation of $\omega(t,\tau)$, a pure WS strategy is better.

3.6. Thrashing

It has been observed that even the slightest attempt to overuse memory may trigger a total collapse of service efficiency, rather than the moderate degradation that might be expected. This phenomenon is known as thrashing. We show that thrashing is caused by the large value of the traverse time T.

In this section we write μ for the missing-page probability.

3.6.1. The Causes

Suppose that a certain process has executed for a virtual time interval of length V and that the missing-page probability μ is constant over this interval. The expected number of page waits is then $(V\mu)$, each costing one traverse time T. We define the duty factor $\eta(\mu)$ to be:

$$\eta(\mu) = \frac{\text{(elapsed virtual time)}}{\text{(elapsed virtual time) + (elapsed page wait time)}}$$

$$\eta(\mu) = \frac{V}{V + V\mu T} = \frac{1}{1 + \mu T} \tag{3.6.1}$$

$\eta(\mu)$ measures the ability of a process to use a processor.

Figure 3-17 shows $\eta(\mu)$ sketched for five values of T:

$$T = 1, 10, 100, 1000, 10000 \text{ vtu}$$

If 1 vtu is taken to be 1 microsecond, and the rotation of the fastest existing rotating auxiliary storage devices is taken to be 10 milliseconds, then T=10000 vtu may be regarded as typical for existing computer systems.

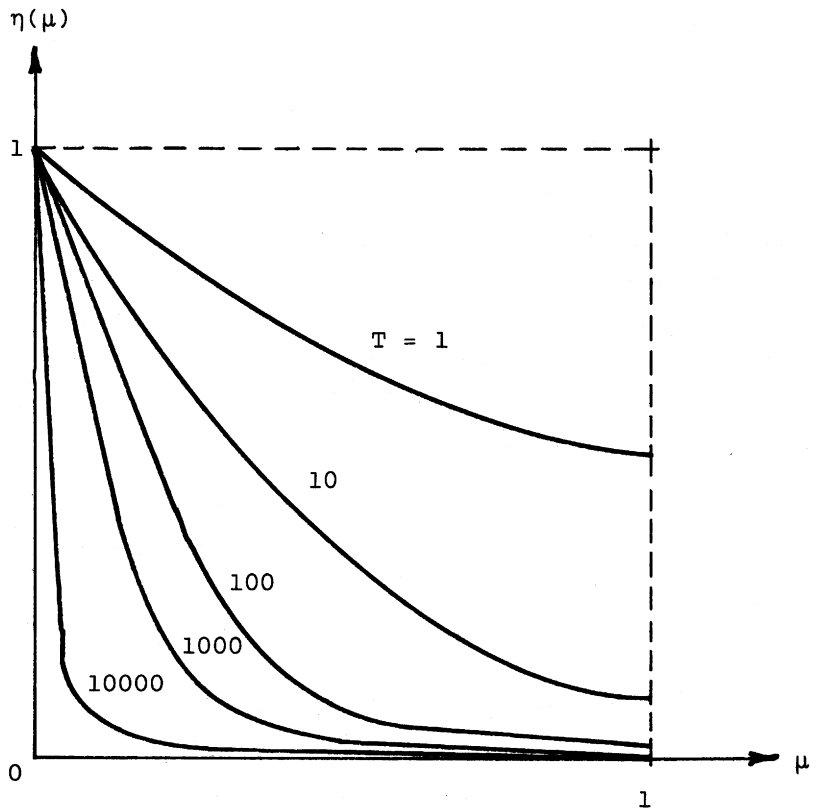


Figure 3-17. Duty factor for various T.

The slope of $\eta(\mu)$ is

$$\eta'(\mu) = \frac{d}{d\mu}\eta(\mu) = -\frac{T}{(1+\mu T)^2} \tag{3.6.2}$$

which for small μ and $T \gg 1$, is extremely sensitive to a change in μ. It is this extreme sensitivity of $\eta(\mu)$ to changes in μ for large T that is responsible for thrashing.

To show how the slightest attempt to over use memory can wreck processing efficiency, we perform the following conceptual experiment. We imagine a set of (n+1) identical programs, n of which are initially operating together in a memory, at the verge of saturation (i.e., $n=n_o$ in Figure 3-10) with no sharing. Then we examine the effect of introducing the $(n+1)^{st}$ program.

Let $1,2,\ldots,(n+1)$ be this set of (n+1) programs, each of size r. Initially, n of them occupy the memory, so that the memory size is M=nr. Let μ_o denote the missing-page probability under these circumstances, assume $\mu_o \ll 1$, and that $\eta(\mu_o)$ is reasonable (i.e., it is <u>not</u> true that $\eta(\mu_o) \ll 1$). Then the expected number of busy processors (ignoring the cost of switching a processor between processes) is:

$$\alpha = \sum_{i=1}^{n} \eta_i(\mu_o) = \frac{n}{1+\mu_o T} \tag{3.6.3}$$

Now introduce the $(n+1)^{st}$ program. The missing-page probability increases to $(\mu_o+\delta)$ and the expected number of busy processors becomes

$$\beta = \sum_{i=1}^{n+1} \eta_i(\mu_o+\delta) = \frac{n+1}{1+(\mu_o+\delta)T} \tag{3.6.4}$$

Now, if nr pages consume the memory and we squeeze another size r program into memory, the resulting increase in missing-page probability is

$$\delta = \frac{r}{(n+1)r} = \frac{1}{n+1} \tag{3.6.5}$$

since we are assuming that the paging algorithm acquires the additional r pages by displacing r pages uniformly from the (n+1) programs now resident in memory. The fractional number of busy processors after introduction of the $(n+1)^{st}$ program is

$$\frac{\beta}{\alpha} = \frac{n+1}{n} \frac{1 + \mu_o T}{1 + (\mu_o+\delta)T} \tag{3.6.6}$$

Now, assume T>>n>>1. We argue that $\delta = \frac{1}{n+1} >> \mu_o$. To see this, suppose to the contrary that $\delta \approx \mu_o$; then

$$\eta(\mu_o) \approx \frac{1}{1 + \delta T} = \frac{1}{1 + \frac{T}{n+1}} = \frac{n + 1}{n + 1 + T} << 1 \tag{3.6.7}$$

which contradicts our assumption that, in the non-saturated operating region, efficiency is reasonable. Thus, when T>>n>>1 and $\delta >> \mu_o$, it is easy to show that

$$\frac{\beta}{\alpha} \approx \frac{n}{T} + n\mu_o << 1 \tag{3.6.8}$$

The presence of one additional program has caused a complete collapse of service.

The sharp difference between the two cases at first defies the intuition, which might lead us to expect a gradual degradation of service. The large value of the traverse time T is the

root cause. It is interesting to note that Smith [S7] has warned of this behavior.

3.6.2. The Cures

To cure or prevent thrashing, we must do two things: first, we must prevent the missing-page probability μ from fluctuating; and second, we must reduce the traverse time T.

In order to prevent μ from fluctuating, we must be sure that the number n of programs residing in main memory satisfies $n \leq n_o$ (Figure 3-16), which is equivalent to the condition that

$$\sum_{i=1}^{n_o} \omega_i(t,\tau_i) \leq M \tag{3.6.9}$$

where $\omega_i(t,\tau_i)$ is the working set size of program i. In other words, there must be space enough in memory for each program's working set. This strongly suggests that a working set strategy be used.

In order to get the largest number of programs in memory, that is, to maximize n_o, we want to choose τ as small as possible and yet be sure that $W(t,\tau)$ contains a process's favored pages. Programmers can cooperate in this effort by designing algorithms to operate locally on data, consciously keeping the working set small and not moving about too rapidly. A programmer is rewarded for this effort, because not only does he achieve a high operating duty factor, he also pays less for use of memory.

With the FIFO, RAND, and LRU algorithms, it is very difficult to ascertain n_o, and therefore difficult to control the possible μ-fluctuations. The FINUFO algorithm displays some natural tendency to refuse to run more than n_o programs (the extra ones tend to be completely unloaded).

The problem of reducing the traverse time T is more difficult. Recall that T is the expectation of a random variable composed of queue waits, and mechanical delay factors. Using optimum scheduling techniques on disk and drum auxiliary storage devices [C2,D3,F3], together with parallel data channels, we can effectively remove all but the mechanical delays from T; accordingly, T may be made comparable to a disk arm seek time or to half a drum revolution time. To reduce T further would require reduction of the rotation tome of the device (for example, a 40,000 rpm drum).

A much more promising solution is to dispense altogether with a rotating device as the second level of memory. A three-level memory system (Figure 3-18) would be a possible solution, where between the main level and the drum we have introduced a slow speed bulk core storage. The analysis of Section 3.6.1 suggests that speed ratios in the order of 1:100 (i.e., $T \approx 100$ vtu) between adjacent devices would lead to much less sensitivity to traverse times and permit tighter control over the factors that cause thrashing. For example:

level	type of memory device	access time
0	thin film	200 ns.
1	slow speed core	20 μs.
2	very high speed drum	2 ms.

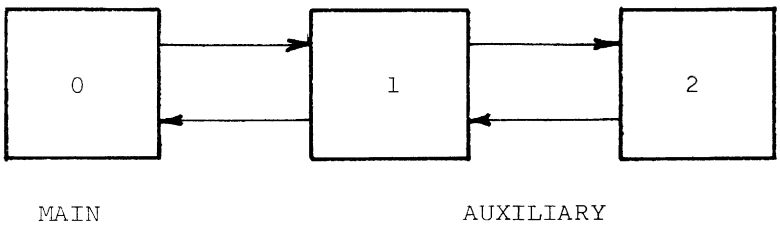


Figure 3-18. Three-level memory system.

We cannot overemphasize, however, the importance of a sufficient supply of main memory, enough to contain the desired number of working sets. Paging is no substitute for real main memory.

3.7. Survey of the Literature

Various studies concerning the behavior of paging algorithms have appeared. The earliest published study, by Fine *et al.* [F6], investigates the effects of demand paging and seriously questions whether paging is worthwhile at all. Their experiments, as well as the more discerning experiments of Varian and Coffman [V1], confirm this: if a program is forced to operate in a space smaller than its working set, considerable paging activity may seriously interfere with efficiency. The remedy is not to dismiss paging, it is to provide enough main memory. Paging is no substitute for real memory.

Experience with the M44/44X computer has yielded important insights into program behavior [O1]. Belady and his colleagues, noting the concavity of efficiency vs. core-share curves, were able to improve efficiency significantly by artificially varying a program's core share; this led to the biased replacement rules [B2]. Belady has defined a unit of storage allocation, the *parachor* [B2], which is that amount of information that must be loaded in main memory for the program to spend no more than half its time in page wait. We shall discuss in Chapter 4 the relation between parachor and working set. Belady has also compared some of the paging algorithms mathematically [B1]. His most important conclusion in this area is that an ideal replacement rule should have much of the simplicity of RAND or FIFO (for efficiency) and some, though not much accumulation of data on past reference patterns.

Randell and Kuehner [R2] have a good survey of all the techniques commonly used to handle multiprogrammed memory allocation, ranging from various name space concepts, across look

ahead and replacement rules, to problems of optimum page size.

Oppenheimer and Weizer [02] report on simulations of the RCA Spectra 70/46 Time-Sharing Operating System when memory allocation is based on a strategy related to working sets. Their experiments indicate that this type of allocation markedly improves performance.

3.8. Summary

Starting from the assumption that locality is a basic program behavior property, we developed the working set model for program behavior. Locality is the property that, during any virtual time interval, a process favors only a subset of the pages available ot it; a working set is a dynamic measure of this set of pages. Locality manifests itself as convexity in the working size and as concavity in the missing-page probability. Experimental evidence suggests that locality is a very good assumption. There is every reason to beleive that a programmer who keeps in mind a working set concept can make this property strong in his programs.

A good performance measure for paging policies is the missing page probability, since lower missing-page probabilities result in lower memory usage costs. We showed that working set strategies achieve the lowest missing-page probabilities and operate dynamically in a memory space close to that which achieves minimum cost.

We also showed that thrashing is directly traceable to the large value of the traverse time T. By minimizing the possibility of fluctuations in the missing-page probability, a working set strategy can markedly decrease sensitivity to thrashing.

Thus, a working set strategy has three advantages. First, it results in lowest costs of operating programs in memory. Second, it reduces sensitivity to thrashing. Third, it makes programs independent of one another, in the sense that memory acquisitions of one program do not interfere with the working set holdings of another. Because of this, analysis will be simple.

In the next Chapter we refine the working set model and derive detailed properties, in the case of no sharing. In Chapter 5 we give attention to the case of sharing, when working sets overlap. The reader who is interested in the ideas of demand and balance should turn directly to Chapter 6.

CHAPTER 4

Further Properties of the Working Set Model

4.0. Introduction

Having seen the basic concepts beneath the working set model, we are in a position to investigate its properties more thoroughly. Here in this chapter we shall refine the working set model in the very simplest case: single-process computations with no information sharing. In Chapter 5 we shall investigate the additional complications that arise from multiprocess computations and overlapping working sets.

One of the properties of a working set memory management policy is the statistical independence of working sets: the expansion of one working set cannot displace pages of another. Because of this, we may analyze the behavior of a single process and its working set, and then extend the results in a simple way to collections of independent processes with non-overlapping working sets.

The quantities we shall derive here in this chapter are described in the following table.

quantity	symbol	description
missing-page probability	$\lambda(\tau)$	probability that a process, when making an information reference, directs the reference to a page not in the working set $W(t,\tau)$.
paging rate	$\rho(\tau)$	number of pages per unit real time re-entering a working set $W(t,\tau)$.
expected working set size	$w(\tau)$	expected number of pages in working set $W(t,\tau)$.
variance of working set size	$\sigma_\omega^2(\tau)$	- -
duty factor	$\eta(\tau)$	fraction of time a running or page wait process spends running.
τ-sensitivity	$s(\tau)$	rate of increase of missing page probability to decrease in τ.

The interreference distribution $F_x(u)$ plays a key role in the analysis, since all these quantities may be expressed in terms of $F_x(u)$.

We begin by deriving an important result: the mean interreference interval is also the mean program size. An interesting consequence of this is that the expected working set size depends only on the interreference distribution. We derive, one by one, the quantities listed in the table above; then we show how each of these quantities is useful in determining the allowable range of τ-values to be used; we discuss the problem of predicting working set sizes; and finally we discuss how a working set memory allocation strategy might be implemented.

During the remainder of this chapter we assume:

1. No sharing. That is, working sets do not overlap.
2. Single-process computations.
3. Only working-set pages are in memory. Any page which leaves a working set is automatically removed from main memory.
4. Unlimited processor-memory resources. But, since only a finite number of processes, each with a finite working set, are active, only a finite amount of each resource type is in use.

The third assumption is a worst-case assumption, in the sense that a working set strategy would not normally retire a non-working-set page until there was need for the space it occupied. The fourth assumption allows us to ignore for the time being whatever additional problems arise from lack of equipment. We shall discuss these problems in Chapter 8, when we examine the equipment configuration.

4.1. The Relation between Program Sizes and Interreference Intervals

As before, we define the random variable x to be the virtual time interval between successive references to the same page. Thus, these intervals x are the interarrival times between references.

The distribution function is $F_x(u) = \Pr[x \leq u]$. The density function is $f_x(u) = \frac{d}{du} F_x(u)$. The mean is

$$\bar{x} = \int_0^\infty u\, f_x(u)\, du = \int_0^\infty (1-F_x(u))\, du \tag{4.1.1}$$

where this latter integral can be verified by integrating the former by parts[1]. The second moment is

$$\overline{x^2} = \int_0^\infty u^2\, f_x(u)\, du \tag{4.1.2}$$

and the variance is $\sigma_x^2 = \overline{x^2} - \bar{x}^2$. We assume both $\bar{x}$ and $\overline{x^2}$ are finite (that $\bar{x}$ is finite is shown shortly in Theorem 4.1).

---

[1] The formula is: $\int y\, dz = yz - \int z\, dy$. In eq. 4.1.1 let y=u, dy=du, $dz=f_x(u)\, du$, $z=F_x(u)$. We integrate from 0 to α, and let α tend to infinity when we are done. Then

$$\int_0^\alpha u\, f_x(u)\, du = yz\Big|_0^\alpha - \int_0^\alpha z\, dy = uF_x(u)\Big|_0^\alpha - \int_0^\alpha F_x(u)\, du - \alpha + \int_0^\alpha du$$

where we have added and subtracted $\alpha = \int_0^\alpha du$. Noting that $uF_x(u)\big|_0^\alpha = \alpha F_x(\alpha)$ and regrouping terms,

$$\int_0^\alpha u\, f_x(u)\, du = \int_0^\alpha (1-F_x(u))\, du - \alpha(1-F_x(\alpha))$$

To complete the proof, we must show $\alpha(1-F_x(\alpha))$ tends to 0 as α tends to infinity, when x has finite mean. Now, $\alpha(1-F_x(\alpha)) \to 0$ if and only if $(1-F_x(\alpha))/(1/\alpha) \to 0$ if and only if (L'Hospital's Rule) $f_x(\alpha)/(1/\alpha^2) \to 0$ which is exactly the condition that $\bar{x}$ be finite.

By a <u>program</u> Z(t) at time t we mean the set of pages toward which a process directs its information references[1]. The <u>program size</u> is $z(t) = |Z(t)|$. We assume that z(t) is a stationary stochastic process, so that we may write z instead of z(t), and $\bar{z}$ instead of $\overline{z(t)}^t$. The program size distribution is $F_z(u)$, across the ensemble of all programs.

We must be careful not to confuse a program Z(t) with a working set $W(t,\tau)$. A working set $W(t,\tau)$ is related to a program Z(t), thus:

$$W(t,\tau) \subseteq \bigcup_{s\varepsilon(t-\tau,t)} Z(s) = Z(t,\tau) \qquad (4.1.3)$$

where $(t-\tau,t)$ is a virtual time interval. Because of our assumption of locality, we assume also that the content of Z(t) does not change appreciably over intervals of length τ, so that the size of $Z(t,\tau)$ is described by the random variable z. Thus, we assume

$$|Z(t,\tau)| = z \qquad (4.1.4)$$

Recalling the definition of the working set size $\omega(t,\tau)$, we have

$$\omega(t,\tau) \leq z \qquad (4.1.5)$$

and hence

$$w(\tau) \leq \bar{z} \qquad (4.1.6)$$

---

[1] A more detailed view would note that a program contains the instruction stream that directs the activity of a process, together with the data used. However, we do not require this much detail in our analysis.

Theorem 4.1. Let x be the interreference intervals, with mean $\bar{x}$. Let z be the program sizes, with mean $\bar{z}$. Then $\bar{x} = \bar{z}$.

Proof: Refer to Figure 4-1, where we have shown a set of z pages consitituing a certain program. Let

$$p_i = \Pr\left[\begin{array}{l}\text{process references page i,}\\ \text{when program size is z}\end{array}\right] \tag{4.1.7}$$

then,

$$\sum_{i=1}^{z} p_i = 1 \tag{4.1.8}$$

Now, let

$$(x|z)_i = \left[\begin{array}{l}\text{interreference interval to page i,}\\ \text{when program size is z.}\end{array}\right] \tag{4.1.9}$$

In a sequence of independent trials, with Pr[success]=p_i, the expected waiting time until success is $1/p_i$. Thus,

$$\overline{(x|z)_i} = \frac{1}{p_i} \tag{4.1.10}$$

For the entire program,

$$\overline{(x|z)} = \sum_{i=1}^{z} \overline{(x|z)_i}\, p_i = \sum_{i=1}^{z} \frac{1}{p_i}\, p_i = z \tag{4.1.11}$$

Now, taking the expectation on z,

$$\bar{x} = \overline{\overline{(x|z)}^{z}} = \bar{z} \tag{4.1.12}$$

QED.

Corollary 4.1. Let Z_1 and Z_2 be programs; let x_1 be the interreference intervals to Z_1 and x_2 be the interreference intervals to Z_2. If Z_1 is bigger than Z_2, then $\bar{x}_1 > \bar{x}_2$.

Proof: Let $z_1 = Z_1$ and $z_2 = Z_2$. We have

$$\bar{x}_1 = \overline{(x|z_1)} = z_1 > z_2 = \overline{(x|z_2)} = \bar{x}_2$$

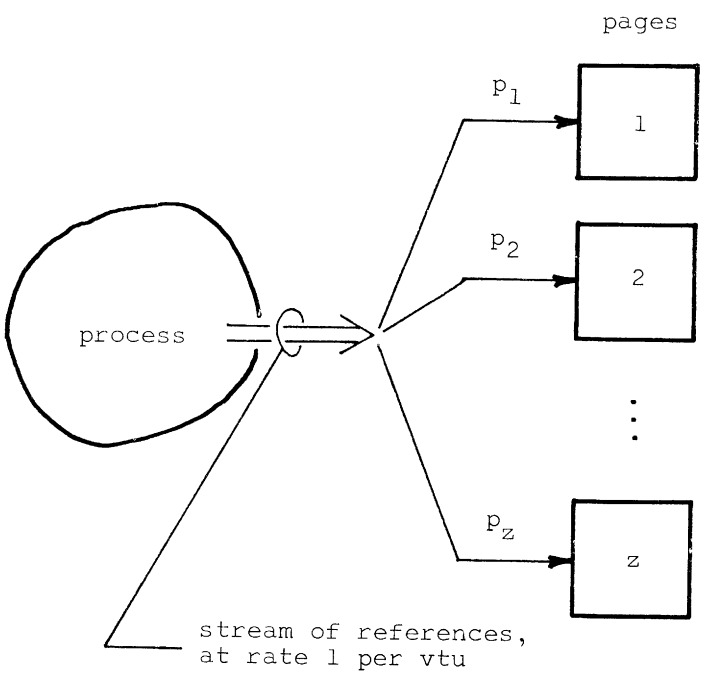


Figure 4-1. A simple program model.

Note, however, that we cannot also claim $F_z(u)=F_x(u)$; i.e., that the interreference distribution is also the program size distribution. In a given computer system, there will be some largest program size z_o. It is unreasonable to assume

(4.1.13) $$\Pr[x > z_o] = 0$$

because even the largest program may contain pages it uses only rarely. Thus, we expect that the variance σ_x^2 of interreference intervals will be greater than the variance σ_z^2 of program size.

Since $F_x(u)$ and $F_z(u)$ describe the ensemble of all programs, we can make no claim that any given program is reliably described by $F_x(u)$ or $F_z(u)$. We can, however, claim that a balance set of programs, being large, is representative of the ensemble; thus the quantities we shall derive in the next sections, expressed in terms of $F_x(u)$, are applicable to balance sets of programs.

A question which may have occurred to the reader is: How does page size (number of words to a page) enter into our considerations? Page size is accounted for implicitly in the definitions of the interreference intervals x and the traverse time T. On the one hand, halving the page size makes the same program comprise twice as many pages; from Theorem 4.1, we see that the interreference intervals become twice as long. That is, smaller pages are referenced less often. On the other hand, the traverse time T contains a component due to page transmission time, which depends on page size.

Thus, provided that pages are sufficiently small that working sets contain several pages, all our results are independent of the page size.

4.2. Missing-Page Probability

We showed in Theorem 3.3 that the missing-page probability depends on $F_x(u)$:

(4.2.1) $\lambda(\tau) = 1-F_x(\tau)$

and is just the probability that the page referenced satisfies $x>\tau$. The next theorem shows that $\lambda(\tau)$ may be regarded as the rate, in virtual time, at which pages re-enter the working set $W(t,\tau)$; that is, $1/\lambda(\tau)$ is the expected virtual time interval between references to a page not in $W(t,\tau)$. See Figure 4-2.

Theorem 4.2. Let $\lambda(\tau)=1-F_x(\tau)$ be the missing-page probability. Then $\lambda(\tau)$ is also the number of pages per unit virtual time re-entering $W(t,\tau)$.

Proof: Let Z(t) be a program. We consider first the behavior of a typical page in in Z(t) and then obtain the behavior of Z(t) by summing the behaviors of its component pages.

Let $\{t_n\}_{n\geq 0}$ be a sequence of virtual time instants at which references to page i occur (Figure 4-3). The n^{th} interreference interval is

(4.2.2) $x_n = t_n-t_{n-1}$

Now, we assume the interreference intervals $\{x_n\}_{n\geq 1}$ are statistically independent[1], so that for all $n\geq 1$:

(4.2.3) $f_{x_n}(u) = f_x(u)$

A re-entry point is a reference instant that finds the page not in $W(t,\tau)$: at such an instant the page re-enters $W(t,\tau)$. Observe

---

[1]This assumption does not contradict the assumption of locality. Locality implies only that the favored pages have short interreference intervals.

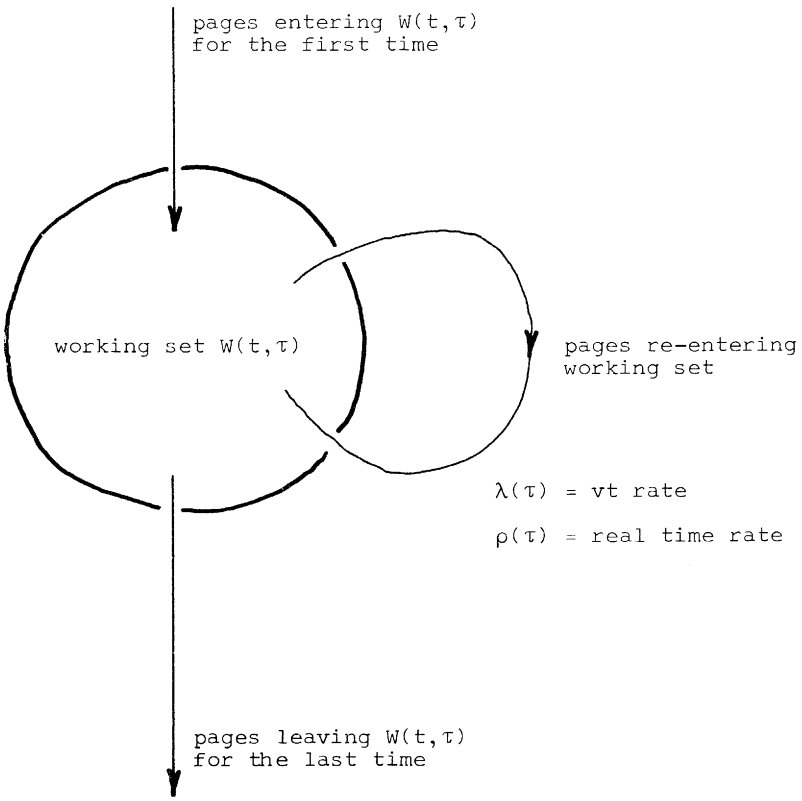


Figure 4-2. Illustrating the meaning of re-entry rates.

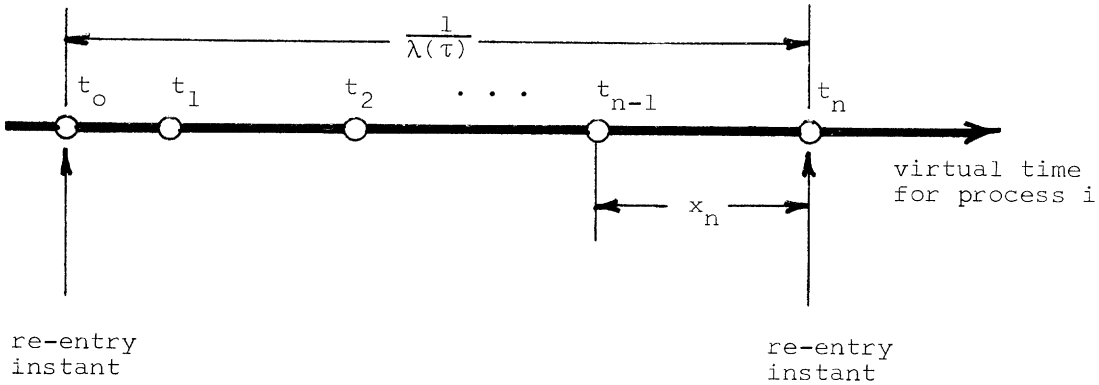


Figure 4-3. Sequence of references to page i.

that t_n is a re-entry instant if and only if $x_n > \tau$, independent of other reference instants. Suppose t_o is a re-entry; we are interested in π_n, the probability

(4.2.4) $$\pi_n = \Pr[t_n \text{ is first re-entry after } t_o]$$

The probabilities $\{\pi_n\}_{n\geq 1}$ are distributed geometrically:

(4.2.5) $$\pi_n = (F_x(\tau))^{n-1}(1-F_x(\tau))$$

That is, t_n is the first re-entry after t_o if and only if each of the intervals $x_1,\ldots,x_{n-1}$ satisfies $x_1 \leq \tau,\ldots,x_{n-1} \leq \tau$ and x_n satisfies $x_n > \tau$. The expected number of references until the re-entry is

(4.2.6) $$\bar{n} = \sum_{i=1}^{\infty} n\,\pi_n = \frac{1}{1 - F_x(\tau)}$$

Each interreference interval x is of expected length $\bar{x}$, so the expected time between re-entries is

(4.2.7) $$\bar{n}\,\bar{x} = \frac{\bar{x}}{1 - F_x(\tau)}$$

Let us define the <u>virtual time re-entry rate</u> $\lambda_i(\tau)$ for page i in Z(t) to be:

(4.2.8) $$\lambda_i(\tau) = \frac{1 - F_x(\tau)}{\bar{x}}$$

Next, suppose Z(t) contains z pages. Given z, the total re-entry rate for Z(t) is:

(4.2.9) $$(\lambda(\tau)\,|\,z) = \sum_{i=1}^{z} \lambda_i(\tau) = z\,\frac{1 - F_x(\tau)}{\bar{x}}$$

Then, taking the expectation on z,

(4.2.10) $$\lambda(\tau) = \overline{(\lambda(\tau)\,|\,z)^z} = \bar{z}\,\frac{1 - F_x(\tau)}{\bar{x}}$$

But, since $\bar{z} = \bar{x}$ (Theorem 4.1), we obtain finally

(4.2.11) $$\lambda(\tau) = 1 - F_x(\tau)$$

QED.

Since $F_x(\tau)$ is a non-decreasing function of τ, $\lambda(\tau)$ is a non-increasing function of τ. Thus, decreasing τ can never result in a decrease in the missing-page probability.

4.3. Paging Rate

Assuming that the memory management mechansim guarantees that a page resides in main memory if and only if it is in a working set, every re-entry point in virtual time corresponds to a page wait in real time. Thus, every page re-entering $W(t,\tau)$ must be recalled from auxiliary memory, and contributes to page traffic.

Define the paging rate $\rho(\tau)$ to be the number of pages per unit real time re-entering the working set $W(t,\tau)$. That is, $1/\rho(\tau)$ is the expected real time between re-entries. See Figure 4-2.

Theorem 4.3. Let $\rho(\tau)$ be the paging rate. Then

$$\rho(\tau) = \frac{\lambda(\tau)}{1 + \lambda(\tau)T} \tag{4.3.1}$$

where T is the traverse time, and $\lambda(\tau)$ is the missing-page probability.

Proof: The expected virtual time between re-entries is $1/\lambda(\tau)$, by Theorem 4.2. Then the expected real time between re-entries is

$$\frac{1}{\rho(\tau)} = \frac{1}{\lambda(\tau)} + T \tag{4.3.2}$$

so that

$$\rho(\tau) = \frac{\lambda(\tau)}{1 + \lambda(\tau)T}$$

QED.

Observe that $\rho(\tau)$ may be interpreted as

$$\rho(\tau) = \frac{\text{number of re-entries}}{\text{elapsed real time}} \tag{4.3.3}$$

because, in a virtual time interval of length V, there are $V\lambda(\tau)$ re-entires; each costs one traverse time T, so the elapsed real time must be $(V + V\lambda(\tau)T)$.

With a balanced memory (i.e., the totality of working sets constituting the balance set B does not exceed memory), process j in the balance set B contributes $\rho_j(\tau)$ to the <u>total returning page traffic</u> $\Psi(\tau)$:

$$\Psi(\tau) = \sum_{j \in B} \rho_j(\tau) \qquad (4.3.4)$$

so that $\Psi(\tau)$ estimates the total traffic of pages being recalled to memory, and is therefore a lower bound on the capacity required of the channel bridging the two levels of memory.

The rate $\Psi(\tau)$ does not include page traffic resulting from:

1. computations entering and leaving the balance set B;
2. pages being referenced for the first or last time by processes in B (see Figure 4-2).

Given the rate at which each of these occurs, one can estimate the true total paging rate. These adjustments are straightforward, so we shall not pursue the matter further.

We must emphasize that the rates $\rho(\tau)$ and $\Psi(\tau)$ are estimates of steady-state behavior, under the assumptions of Section 4.1. The important point is: starting from the interreference distribution $F_x(u)$ and the definition of $W(t,\tau)$, it is possible to estimate these rates.

4.4. Working Set Size

Let Z(t) be a program, and let z(t) be the random variable of the size of Z(t). Starting from the assumption that z(t)=z is a stationary stochastic process, we shall derive expressions for the mean and variance of working set size $\omega(t,\tau)$. The importance of these results is that a program's main memory requirement is completely determined by its page interreference activity.

Theorem 4.4. Let $w(\tau) = \overline{\omega(t,\tau)}$ be the expected working set size. Then

$$w(\tau) = \int_0^\tau (1-F_x(u))\,du = \int_0^\tau \lambda(u)\,du \tag{4.4.1}$$

where $\lambda(u)$ is the missing page probability.

Proof: Refer to Figure 4-4, where we have shown an interval in virtual time for a typical page in $W(t,\tau)$. Define the random variable

$$y = \left[\begin{array}{l}\text{length of the interreference interval}\\ \text{containing the time instant t}\end{array}\right] \tag{4.4.2}$$

Thus, if we choose a point t at random on the virtual time axis, y is the length of the interval in which t lies. The density function $f_y(u)$ for y is <u>not</u> the same as that of the interreference intervals x because, even though long intervals are less likely than short intervals, they occupy a larger fraction of the virtual time axis. A little thought should convince the reader that the probability that t is contained in an interval of length u is just the fraction of the time axis occupied by intervals of length u:

$$f_y(u) = \frac{u\, f_x(u)}{\bar{x}} \tag{4.4.3}$$

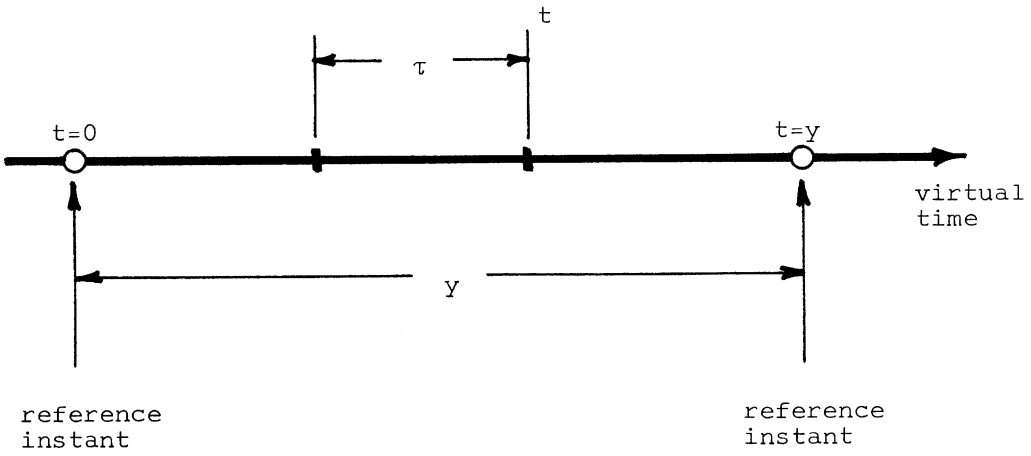


Figure 4-4. Interreference interval containing time t.

For a complete discussion of this property, see Feller [F2,Vol.2, p.10ff]. Let i denote a typical page in the program Z(t). Define the binary random variable

$$(4.4.4) \qquad \gamma_i = \begin{cases} 1 & \text{if } i \varepsilon W(t,\tau) \\ 0 & \text{otherwise} \end{cases}$$

Refer again to Figure 4-4, and use t=0 as the left end of the interval. Suppose y=u; then

$$\Pr[\gamma_i=0 \mid y=u] = \Pr[u>\tau \text{ and } t\varepsilon(\tau,u)]$$

$$\Pr[\gamma_i=0] = \int_{u>\tau} \Pr[\gamma_i=0 \mid y=u]\, f_y(u)\, du$$

$$(4.4.5) \qquad \Pr[\gamma_i=0] = \int_{u>\tau} \Pr[u>\tau \text{ and } t\varepsilon(\tau,u)]\, f_y(u)\, du$$

Now, t may fall randomly on the interval (0,y), so

$$(4.4.6) \qquad \Pr[u>\tau \text{ and } t\varepsilon(\tau,u)] = \frac{u-\tau}{u}$$

then

$$(4.4.7) \qquad \Pr[\gamma_i=0] = \int_\tau^\infty \frac{u-\tau}{u} f_y(u)\, du = \int_\tau^\infty \frac{(u-\tau)\, f_x(u)\, du}{\bar{x}}$$

carrying out this integration (by parts) we obtain

$$(4.4.8) \qquad \Pr[\gamma_i=0] = 1 - \frac{1}{\bar{x}} \int_o^\tau \lambda(u)\, du$$

where $\lambda(u)=1-F_x(u)$. Then

$$(4.4.9) \qquad \Pr[\gamma_i=1] = \frac{1}{\bar{x}} \int_o^\tau \lambda(u)\, du$$

Now, observe that

$$(4.4.10) \qquad \omega(t,\tau) = \sum_{i\varepsilon Z(t)} \gamma_i$$

Suppose $|Z(t)| = z$. Then, given z, the expected working set size is

$$(4.4.11) \quad (w(\tau) \mid z) = \overline{\omega(t,\tau)} = \sum_{i=1}^{z} \overline{\gamma_i} = \sum_{i=1}^{z} \Pr[\gamma_i=1] = z\Pr[\gamma_i=1]$$

Then, taking expectation on z,

$$(4.4.12) \qquad w(\tau) = \overline{(w(\tau)\,|\,z)}^{z} = \frac{\bar{z}}{\bar{x}} \int_0^{\tau} \lambda(u)\, du$$

Finally, from Theorem 4.1 we have $\bar{z} = \bar{x}$, so that

$$w(\tau) = \int_0^{\tau} \lambda(u)\, du$$

QED.

We should verify that the properties of Theorem 3.3 are satisfied:

1. $w(\tau) \leq \tau$
2. $w(0) = 0$
3. $w(\tau+s) \geq w(\tau) \qquad s \geq 0$
4. $w(\tau)$ convex.

Since $\lambda(u) = 1 - F_x(u) \leq 1$,

$$(4.4.13) \qquad w(\tau) = \int_0^{\tau} \lambda(u)\, du \leq \int_0^{\tau} du = \tau$$

and properties 1 and 2 are satisfied. Since $\lambda(u) \geq 0$,

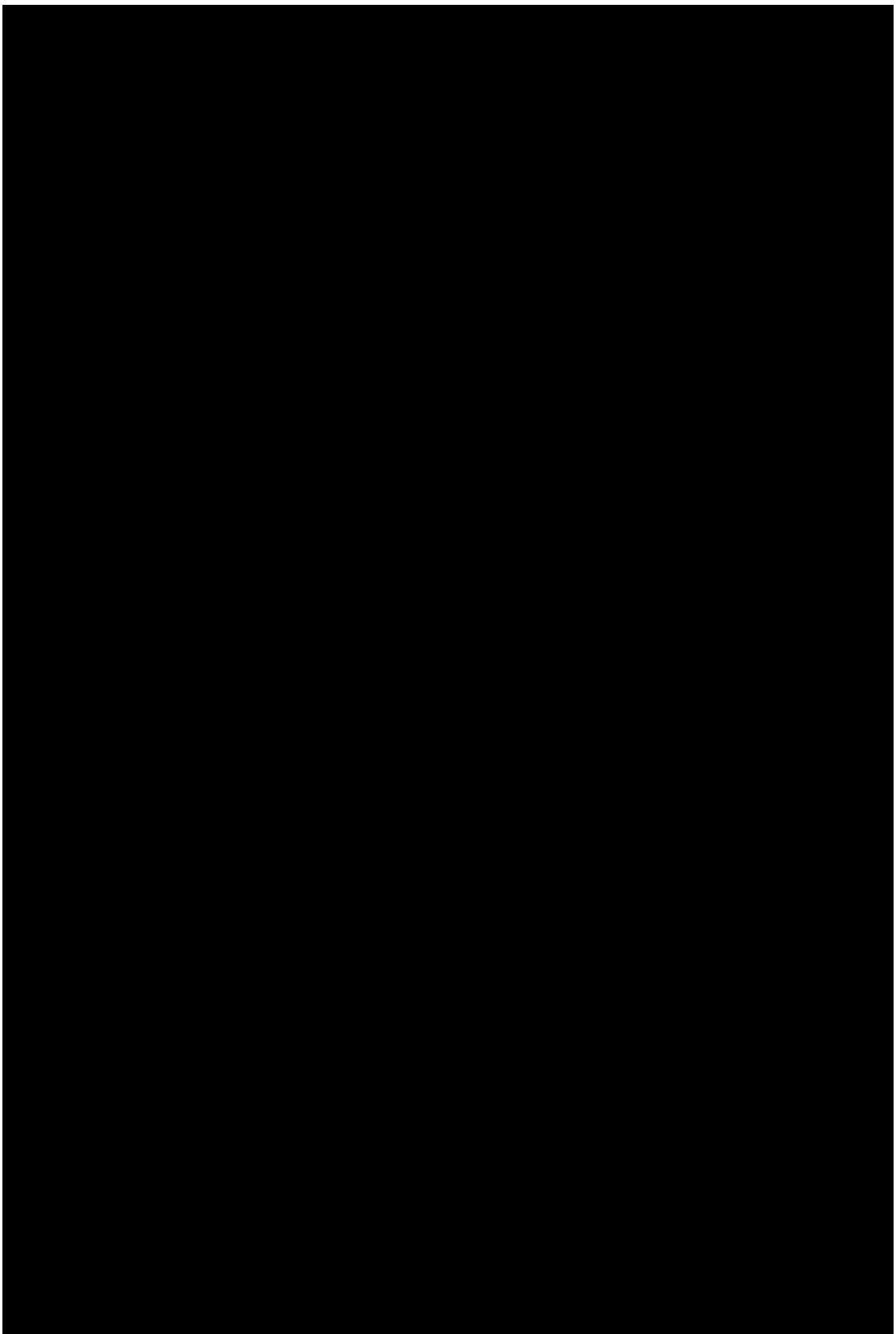
$$(4.4.14) \qquad w(\tau+s) = \int_0^{\tau+s} \lambda(u)\, du \geq \int_0^{\tau} \lambda(u)\, du = w(\tau)$$

and property 3 is satisfied. To verify property 4, we show that the second derivative of $w(\tau)$ is non-positive:

$$(4.4.15) \qquad \begin{aligned} \frac{d}{d\tau} w(\tau) &= \lambda(\tau) = 1 - F_x(\tau) \\ \frac{d^2}{d\tau^2} w(\tau) &= -f_x(\tau) \leq 0 \quad \text{since } f_x(\tau) \geq 0. \end{aligned}$$

Comparing the theorem statement with eq. 4.1.1, we observe also that

$$(4.4.16) \qquad \lim_{\tau \to \infty} w(\tau) = \bar{x} = \bar{z}$$



And, since $\bar{z} = \bar{x}$,

$$\overline{\omega^2(t,\tau)} = w(\tau) + \frac{\overline{x^2}}{\bar{x}^2} w^2(\tau) - \frac{w^2(\tau)}{\bar{x}} \tag{4.4.22}$$

then

$$\sigma_\omega^2(\tau) = \overline{\omega^2(t,\tau)} - w^2(\tau) = \overline{\omega^2(t,\tau)} - \frac{\bar{x}^2}{\bar{x}^2} w^2(\tau) \tag{4.4.23}$$

$$= w(\tau) + \frac{\overline{x^2}-\bar{x}^2}{\bar{x}^2} w^2(\tau) - \frac{w^2(\tau)}{\bar{x}}$$

$$= w(\tau) + \frac{w^2(\tau)}{\bar{x}^2}(\sigma_x^2-\bar{x})$$

QED.

Corollary 4.5. The variance $\sigma_\omega^2(\tau)$ is lower-bounded by

$$\sigma_\omega^2(\tau) \geq w(\tau)\left(1 - \frac{w(\tau)}{\bar{x}}\right) \tag{4.4.24}$$

Proof: For any random variable x, $\sigma_x^2 \geq 0$, so put $\sigma_x^2 = 0$ into the expression above for $\sigma_\omega^2(\tau)$.

QED.

Observe, from eq. 4.4.16 and 4.4.17, that

$$\lim_{\tau \to \infty} \sigma_\omega^2(\tau) = \sigma_x^2 \tag{4.4.25}$$

and that $\sigma_\omega^2(\tau)$ attains a maximum value for some $\tau > 0$.

4.5. Duty Factor

The duty factor $\eta(\tau)$ of a process is the fraction of time it is able to spend computing:

$$(4.5.1) \quad \eta(\tau) = \frac{\text{(elapsed virtual time)}}{\text{(elapsed virtual time) + (elapsed page wait time)}}$$

$\eta(\tau)$ measures the ability of a process to use a processor.

Theorem 4.6. The duty factor $\eta(\tau)$ is given by

$$(4.5.2) \quad \eta(\tau) = \frac{1}{1 + \lambda(\tau)T}$$

where $\lambda(\tau)$ is the missing-page probability, and T is the traverse time.

Proof: Suppose the process has executed for V vtu, with no interruptions other than page waits. The time spent in page wait is then $(V\lambda(\tau)T)$ and so

$$\eta(\tau) = \frac{V}{V + V\lambda(\tau)T} = \frac{1}{1 + \lambda(\tau)T}$$

QED.

The duty factor has already appeared in Section 3.6, on thrashing.

We may interpret $\eta(\tau)$ as the probability that, if we look at a process at some random time, we find it running.

4.6. τ-sensitivity

It is useful to define a sensitivity function $s(\tau)$ that measures how sensitive is the re-entry rate $\lambda(\tau)$ to changes in τ. We define the τ-sensitivity $s(\tau)$ of a working set $W(t,\tau)$ to be

$$(4.6.1) \qquad s(\tau) = -\frac{d}{d\tau}\lambda(\tau) = f_x(\tau)$$

That is, if τ is decreased by $d\tau$, the resulting increase in re-entries to $W(t,\tau)$ is $s(\tau)\,d\tau$. It is obvious that $s(\tau)\geq 0$: reducing τ can never reduce the page traffic.

Observe that $s(\tau)$ is the negative second derivative of $w(\tau)$, and is therefore a measure of the convexity of $w(\tau)$.

$s(\tau)$ may be useful in deciding how small a value of τ to choose. If $f_x(\tau)$ has the shape shown in Figure 4-5, curve A, a good choice for τ is $\tau=\tau_A$ since $\tau>\tau_A$ has little effect on reducing $s(\tau)$. If $f_x(\tau)$ has the shape of curve B we should have to choose $\tau=\tau_B>\tau_A$ in order to have the same τ-sensitivity. There is good reason to believe that in practice $f_x(\tau)$ is approximately hyperexponential, in which case curve A is more representative than curve B.

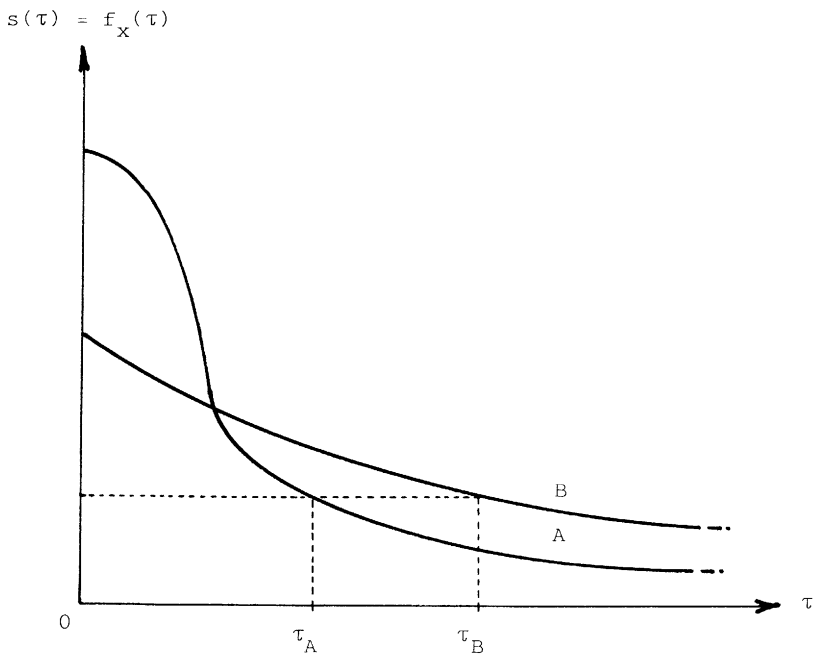


Figure 4-5. Using $s(\tau)$ to choose τ.

4.7. Choosing τ

Ideally, the working set parameter τ should be chosen as small as possible and yet assure that the working set $W_p(t,\tau)$ of process p contains p's favored pages. In principle, then, τ should be variable, from process to process and from time to time.

In practice, it will be necessary to choose non-ideal values for τ, because the optimum value for τ may be indeterminable, or because too much mechanism may be needed either to decide on the required value of τ or else to vary τ dynamically as required. Thus, system parameters, as well as program parameters, will play roles in choosing τ.

Should τ be too small, the favored pages of a process will be removed, resulting in high missing-page probability, high memory-usage costs, high page traffic, and low efficiency (i.e., duty factors). Should τ be too large, pages may remain in memory long after last being used, thus wasting memory and again resulting in high memory-usage costs.

We shall attempt to clarify the nature of the tradeoffs among all these factors.

Strange as it may seem, there may be a <u>worst</u> value for τ. Suppose the process in question has executed for V vtu. The expected number of page waits is $V\lambda(\tau)$, the expected time spent in page wait is $V\lambda(\tau)T$, and the expected elapsed real time is

$$V + V\lambda(\tau)T = V\,(1 + \lambda(\tau)T) \tag{4.7.1}$$

During this interval V, the expected working set size is $w(\tau)$, so that the expected cost per unit virtual time is

$$H(\tau) = \frac{w(\tau)\,V\,(1 + \lambda(\tau)T)}{V} = w(\tau)(1 + \lambda(\tau)T) \tag{4.7.2}$$

In Figure 4-6 we have sketched $w(\tau)$, $(1+\lambda(\tau)T)$, and the product $H(\tau)$. It is clear that $H(\tau)$ attains a maximum for some $\tau_o>0$, if $(1+T)>\bar{x}$. If $T>>1$ it is not hard to see that τ_o is very small, and the value of τ chosen to permit inclusion of at least the favored pages will satisfy $\tau>>\tau_o$. There are values of T satisfying $(1+T)\leq\bar{x}$ such that $H(\tau)$ has no maximum for finite τ, in which case we need not worry about a worst value of τ.

Note that $H(\tau)$ has a maximum at τ_o, whereas the cost function $G(s)$ of Section 3.5.1, as a function of memory space s, has a minimum. The apparent discrepancy is resolved if we note that $H(\tau)$ cannot account for a memory holding larger than the expected working set size $w(\tau)$, whereas $G(s)$ can. The functions $G(s)$ and $H(\tau)$ are not the same cost function.

The remaining tradeoff issues fall into classes: those that depend on the behavior of the program, and those that depend on system requirements. The program-dependent considerations are:

1. Hard vs. Soft Programs (cf. Section 3.3, Figure 3-9). Choose τ as small as possible, yet allow $W(t,\tau)$ to contain the favored pages. A hard program has a well-defined minimum value of τ, whereas a soft program does not.
2. τ-sensitivity (cf. Section 4.6). τ can be chosen so that $s(\tau)$ is at some desired level, or that τ is at the start of a flat region of the $s(\tau)$ curve.

The system-dependent considerations are:

1. Paging rate (cf. Sections 4.2, 4.3). τ can be chosen so that the virtual time between page faults is comparable to T; that is so that $1/\lambda(\tau) = T$. This is

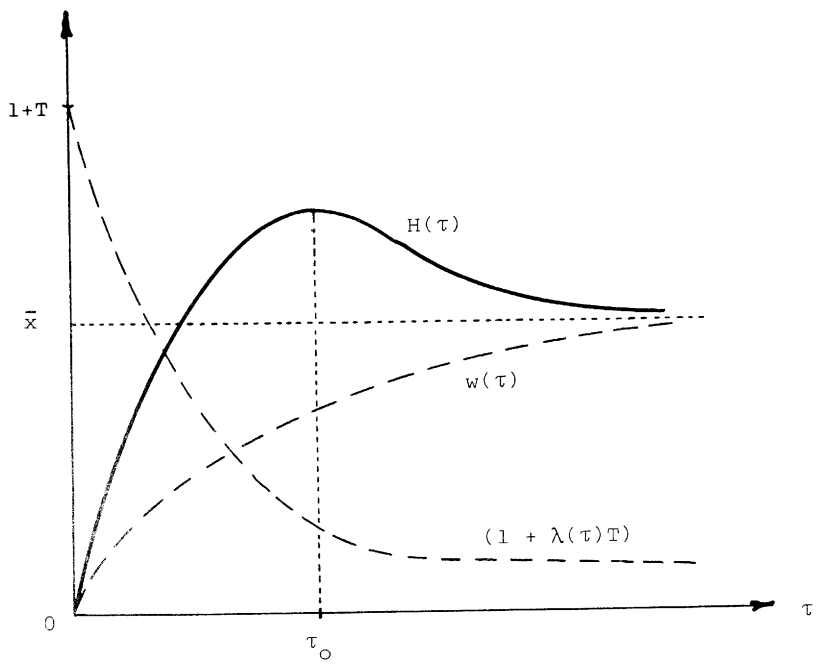


Figure 4-6. Cost per unit virtual time $H(\tau)$.

equivalent to the condition

$$\rho(\tau) = \frac{\lambda(\tau)}{1 + \lambda(\tau)T} = \frac{1}{2T}$$

2. Duty factor (cf. Section 4.5). τ can be chosen so that a given duty factor $\eta(\tau)$ is attained for each computation.

Of course, τ should never be chosen less than whatever is required to satisfy the program-dependent criteria, conditions 1 and 2. In most contemporary systems T is so large ($T \approx 10000$ vtu = 10 ms.) that τ must be chosen to satisy the system-dependent criteria, conditions 3 and 4; this will generally cause τ to be an order of magnitude or more greater than program-dependent criteria would require.

Ideally we should like to have the flexibility to choose τ according to the program-dependet criteria, without regard to the system-dependent criteria. It should be clear that this is achievable only when T becomes much smaller than is normal in contemporary systems; for example, T less than 100 vtu. The use of bulk core storage or some other non-rotating device for the second level of memory can achieve this. We shall return to these issues in Chapter 8.

4.8. Prediction

On several occasions we have noted that a working set $W(t,\tau)$ is a reliable prediction of the working set $W(t+\alpha,\tau)$, if α is not too large; similarly the working set size $\omega(t,\tau)$ is a reliable prediction of the working set size $\omega(t+\alpha,\tau)$, if α is not too large. Without going into great detail we want to indicate how these ideas can be made more precise.

The prediction problem for working set sizes is:

given: $\omega(t,\tau)$ for $t\varepsilon I$

want to estimate: $\omega(t+\alpha,\tau)$ for $\alpha\notin I$

the estimate is to be: $\hat{\omega}(t+\alpha,\tau) = G(\omega(s,\tau))$ for $s\varepsilon I$

Here, I is a set of time points at which the value of $\omega(t,\tau)$ is known. This set I could consist of one or more distinct points, of a time interval (t_1,t_2), or even the entire time history since $t=0$. The transformation G is to be chosen so that $\hat{\omega}(t+\alpha,\tau)$ is an optimum (in terms of a given criterion) estimate of $\omega(t+\alpha,\tau)$.

We assume that $\omega(t,\tau)$ is a stationary stochastic process; hence we can write its expectation independent of time:

$$(4.8.1) \qquad w(\tau) = \overline{\omega(t,\tau)}$$

and we can define the autocorrelation function between $\omega(t,\tau)$ and $\omega(t+u,\tau)$ to be

$$(4.8.2) \qquad R(u,\tau) = \overline{\omega(t,\tau)\ \omega(t+u,\tau)}$$

depending only on the separation u of the two times.

The most common form of prediction is least mean square prediction, used because it is particularly easy to analyze. Define the error of the estimate to be

thus,

(4.8.3) $\quad e(\alpha) = \omega(t+\alpha,\tau) - G(\omega(u,\tau)) \quad u\varepsilon I$

The problem is to choose the transformation G such that the mean square error, $\overline{e^2(\alpha)}$, is minimum. Clearly, the smallest mean sqaure error can be obtained if we impose no restrictions on G (non-linear mean square prediction). This, however, leads to practical and analytic difficulties, so G is usually restricted to be a linear operator. When G is a linear operator, the mean square error $\overline{e^2(\alpha)}$ is minimum if and only if the error $e(\alpha)$ is orthogonal to all the given data (this result is well known; see, for example, reference [P1, p. 389]); that is,

(4.8.4) $\quad \overline{[\omega(t+\alpha,\tau)-G(\omega(u,\tau))]\ \omega(v,\tau)^u} = 0 \quad$ for each $v\varepsilon I$

The most convenient linear operator is a linear combination of a finite number of data. That is, $\omega(t,\tau)$ is known at the time instants $t_1,\ldots,t_n$, and the estimate is to be a linear combination

(4.8.5) $\quad \hat{\omega}(t+\alpha,\tau) = A_1\omega(t_1,\tau)+\ldots+A_n\omega(t_n,\tau)+A_{n+1}$

The constants $A_1,\ldots,A_{n+1}$ must be chosen to satisfy eq. 4.8.4; that is, so that

(4.8.6) $\quad \overline{[\omega(t+\alpha,\tau)-(A_1\omega(t_1,\tau)+\ldots+A_n\omega(t_n,\tau)+A_{n+1})]\ \omega(t_i,\tau)} = 0$

for $i=1,\ldots,n$, and

(4.8.7) $\quad \overline{[\omega(t+\alpha,\tau)-(A_1\omega(t_1,\tau)+\ldots+A_n\omega(t_n,\tau)+A_{n+1})]\ A_{n+1}} = 0$

If one expands these for each i, one obtains (n+1) equations in (n+1) unknowns (the A_i) with coefficients of the form

$$(4.8.8) \qquad \overline{\omega(t_i,\tau)\ \omega(t_j,\tau)} \;=\; R(t_i-t_j,\tau)$$

which follows from eq. 4.8.2.

We hope to have indicated with this overview how the problem of predicting working set sizes might be made more precise, and how an error can be determined for a given estimate and time separation α. We refer the reader to the literature for further detail [P1, p. 385ff.].

4.9. Example

It is interesting to examine the results of the previous sections in the case of exponentially distributed interreference intervals:

$$F_x(u) = 1 - e^{-Bu} \qquad B = \frac{1}{\bar{x}}$$

we have, for the major expressions:

Name	symbol	Result, exponential case
missing-page probability	$\lambda(\tau)$	$e^{-B\tau}$
mean vt interval between re-entries	$\frac{1}{\lambda(\tau)}$	$e^{B\tau}$
mean real time interval between re-entries	$\frac{1}{\rho(\tau)}$	$e^{B\tau} + T$
duty factor	$\eta(\tau)$	$\frac{1}{1 + Te^{-B\tau}}$
expected working set size	$w(\tau)$	$\bar{x}(1-e^{-B\tau})$

Now, suppose we have chosen τ so that

$$\frac{1}{\lambda(\tau)} = T$$

then

$$\tau = \bar{x} \ln T$$

For this choice of τ we have

$$\rho(\tau) = \frac{1}{2T}$$

$$\eta(\tau) = \frac{1}{2}$$

$$w(\tau) = \bar{x}\left(\frac{T-1}{T}\right)$$

4.10. Working Sets and Parachors

Belady has defined a unit of storage allocation, the parachor [B2], which is that amount of information that must be in main memory so that a program spends no more than half its time in page wait. If we choose τ so that

$$\frac{1}{\lambda(\tau)} = T$$

we find that the duty factor is $\eta(\tau) = \frac{1}{2}$. Hence the expected working set size for this value of τ corresponds to one parachor. In the exponential case, this is

$$w(\tau) = \bar{x}\,(\frac{T-1}{T})$$

Allocating one parachor to each program is the same as allocating enough space for its expected working set size. The parachor is a static unit of allocation, whereas the working set size $\omega(t,\tau)$ is a dynamic unit of allocation. Our results in Chapter 3 show that working set strategies should perform better than parachor strategies (a parachor strategy is one that runs a process if and only if there is at least one uncommitted parachor of main memory).

4.11. Implementation of Working Set Memory Management

According to our definition, $W(t,\tau)$ is the set of pages a process has referenced within the last τ vtu of its execution. This suggests that memory management can be controlled with hardware mechanisms, by associating with each page-block of main memory a timer. Whenever a page is referenced, its timer is set to τ and begins to run down; if the timer succeeds in running down, a flag is set to mark to page for removal from main memory whenever the space is needed.

Unfortunately matters are not so simple. According to the definition of $W(t,\tau)$, the timers must run down in virtual time. Virtual time coincides with real time only when the process is running. More precisely, the timer behavior should be as follows, for each process state:

1. running. A timer may run down in real time.
2. page wait. Since the process is temporarily suspended, all timers on its working set pages must be stopped, else they amy run down and working set pages may be removed during a page wait.
3. ready and blocked. If a process is pre-empted by the operating system, or blocks, its timers may continue to run down in real time; then, within τ vtu, the memory it formerly occupied will be freed.

We can see that it is the page wait state that gives the trouble. Whenever a process enters page wait its page timers must stop until the new page is acquired. For other process states, the page timers may run in real time. Therefore we shall associate with each page-block in main memory the name of the process that has most recently referenced it; when the process enters

page wait, all its pages can have their timers suspended.

The following procedure is useful in a software as well as a hardware implementation, and is therefore potentially applicable in contemporary systems. The procedure we propose here samples use bits associated with each page; these use bits may be part of a page table entry or part of a hardware register. Sampling occurs at intervals of σ vtu, σ being called the sampling interval, where $\sigma=\tau/K$, and K is an integer constant chosen to make the sampling intervals as fine as desired (K=2 or 3 should be sufficient). On the basis of page references during each of the last K sampling intervals, the working set $W(t,K\sigma)$ can be determined.

There is a sequence of use bits $u_o,u_1,\ldots,u_K$ associated with each page. Whenever a reference occurs, $1\rightarrow u_o$. At the end of each sampling interval, the bit pattern contained in $u_o,u_1,\ldots,u_K$ is shifted one position, a 0 enters u_o, and u_K is discarded:

$$\begin{aligned} u_{K-1} &\rightarrow u_K \\ &\vdots \\ u_o &\rightarrow u_1 \\ 0 &\rightarrow u_o \end{aligned}$$

Then the logical sum U of the use bits

$$U = u_o + u_1 + \ldots + u_K$$

is U=1 if and only if the page in question has been referenced during the last K sampling intervals; of all the pages associated with a process, those with U=1 constitute its working set $W(t,K\sigma)$.

Figure 4-7 shows how this idea might be implemented in hardware. If process j is currently using the page, the π-field of the page register contains an identifier to j. The PW bit is 0 if and only if j is in page wait. The PT-field points to the page table entry designating this page. The σ-bus is pulsed once every σ vtu; these pulses cause a shift in the use bits if and only if PW=1 (the process is not in page wait). Whenever the logical sum U of use bits becomes 0, a mechanism (not shown) may (not must) remove the page from main memory; this mechanism will dispatch the page to auxiliary memory (unless it has not been modified and there is a spare copy already in auxiliary memory), and then (using PT) find the page table entry for this page and set the in-core bit OFF. All this is done without troubling the operating system.

This mechanism maintains a count of the working set size for each process as follows. Whenever a fresh page of process π is loaded by the operating system (in response to a page fault), increment a counter for the process π. Whenever the logical sum U of use bits becomes 0 for some page marked as belonging to process π, decrement the counter for process π.

It is interesting to note that $\tau=K\sigma$ may be varied if desired by varying σ. The operating system thus has control over the current value of τ.

This basic scheme can also be realized in software, as suggested by Figure 4-8. All processes in the running state are identified in the running list. Upon entry to the running state, process i is assigned some quantum q_i. A process cycles through the list, receiving a burst σ (σ is the sampling interval) at each pass; the quantity γ_i records its time used. There is

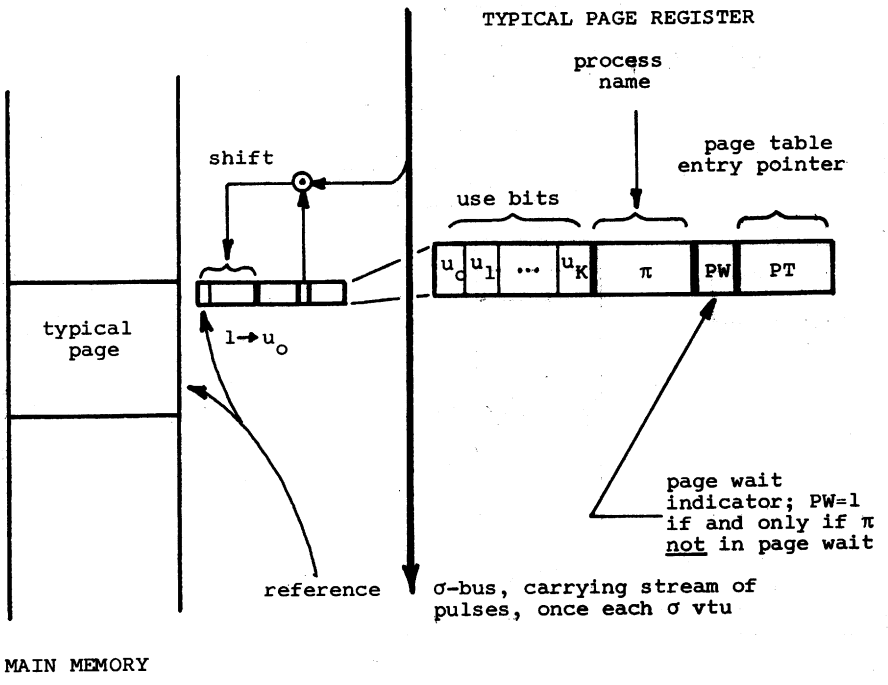


Figure 4-7. Hardware implementation of memory management.

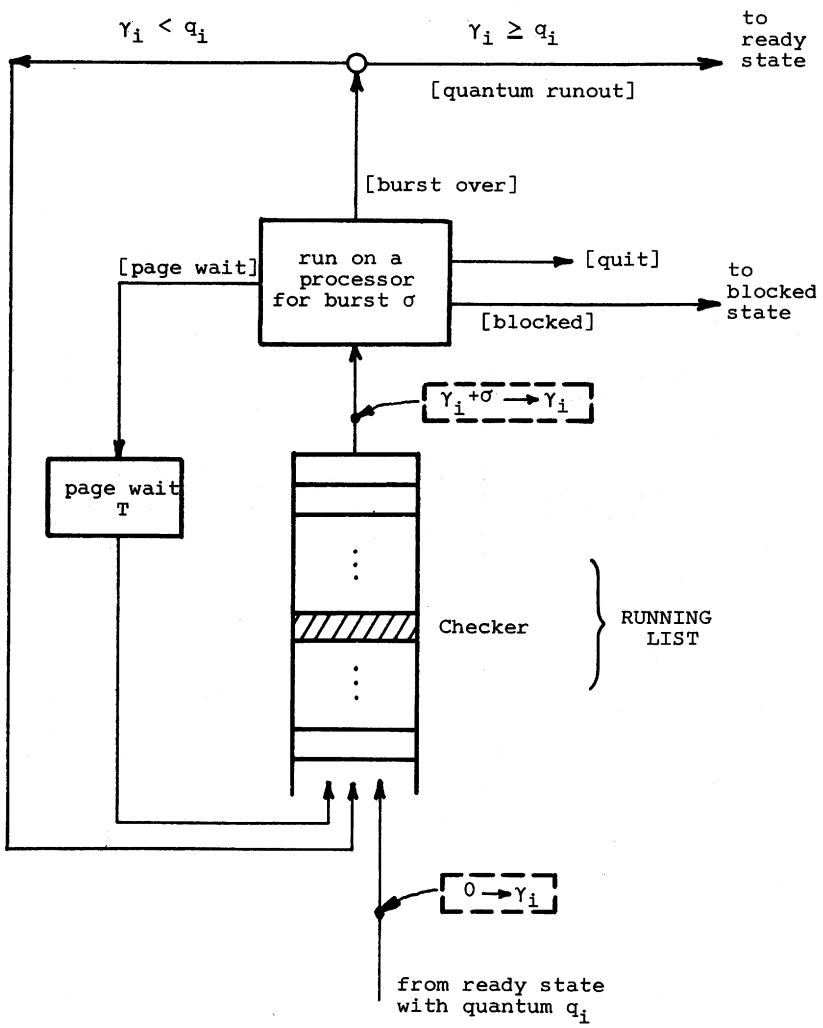


Figure 4-8. Software implementation of memory management.

a special process, the checker; whenever run, the checker looks at the page tables of processes run since the last time it was run, and performs the use-bit shift discussed above (the use bits $u_o, u_1, \ldots, u_K$ are stored in the page tables, so the π, PW, and PT fields of Figure 4-7 are no longer necessary).

Associated with each process is a counter w_i giving its current working set size. At each page fault for process i, w_i is increased by one. If the checker observes a page leave the working set of process i, w_i is decreased by one.

It should be clear that, if the length of the running list is n, the checker samples page use bits only every $n\sigma$ seconds, not every σ seconds.

This implementation is also discussed in reference [D4].

4.12. Summary

We refined the working set model, deriving expressions for the missing-page probability $\lambda(\tau)$, the paging rate $\rho(\tau)$, the expected working set size $w(\tau)$, the variance of the working set size $\sigma_\omega^2(\tau)$, the duty factor $\eta(\tau)$, and the τ-sensitivity $s(\tau)$. Each of these depends on the interreference distribution $F_x(u)$, and some of them depend also on the traverse time T. We showed how each of these plays a role in selecting a value for τ.

We discussed the problem of prediction, showing general methods whereby errors may be determined precisely for a given mode of estimation.

We discussed implementation of working set memory allocation strategies, both for hardware and software.

All this was done in the absence of sharing. In the next chapter we investigate the effects of sharing and show quantitatively that great benefits are attainable.

CHAPTER 5

Multiprocess Information Sharing

5.0. Introduction

In this chapter we complete the characterization of the working set model, by investigating the effects of sharing. Intuition already tells us that sharing should produce an all-around inprovement. Our purpose here is to give quantitative justification to this well known premise.

Under the existing definition, working sets will overlap when their processes share information. This complicates the problem of charging for main memory usage of shared information, because the number of overlaps among shared working sets (there can be as many as (2^n-1) overlaps among n working sets), their sizes, and their contents, may be unknown or at best exceedingly difficult to determine. A minor modification of the definition makes working sets disjoint, thereby relieving these difficulties.

We consider a simple conceptual experiment: n independent processes referencing n identical programs compared to n independent processes referencing one program. We derive expressions for working set size, missing-page probability, paging rate, and duty factor. We show that sharing produces improvement in each of these quantities.

The discussion here in this chapter is not intended to solve the problems of sharing information. We only hope to shed light on the difficulties of the problem and to give insights into possible solutions.

5.1. Sharing

5.1.1. General Aspects

The smallest unit of information that can be shared is, from a process's point of view, a segment, because the protection mechanism operates on a segment level. The smallest unit of information that can be shared is, from the system's point of view, a page, because memory allocation is handled on a page level.

We follow Arden's suggestions for program structure [A1]. If a segment is shared, there will be an entry for it in the segment table of each participating process. Each such entry need not assign the same name to the segment. Each such entry, however, points to the same page table. Thus, each physical segment has exactly one page table describing it.

The problem of charging participants for the use of shared information can be handled at two levels: shared information in main memory, and shared information not in main memory.

Ideally, we should like each participant in sharing of information which resides in main memory to be charged in accordance with his degree of participation. Even though this may not be easy to implement, an extension (Section 5.1.2) of the working set concept can give insights into how this might be done, and how an implementation can approximate this ideal.

When working sets overlap, the existing working-set definition leads to the following difficulty. Suppose computation C contains two processes, designated 1 and 2, which are sharing information. Then

$$W_1(t,\tau) \cap W_2(t,\tau) \neq \emptyset$$

where $\emptyset$ is the empty set. Then the joint working set for computation C is

$$W_C(t,\tau) = W_1(t,\tau) \cup W_2(t,\tau)$$

and the working set size of C is

$$\omega_C(t,\tau) \leq \omega_1(t,\tau) + \omega_2(t,\tau)$$

Thus, measuring the individual working set sizes of the component processes of a computation will lead to an overestimate of the true joint working set size. When there is much sharing, working sets will be very nearly coincident; summing the sizes of each process's working set will grossly overestimate the true joint working set size. This can seriously complicate the problem of attributing memory-usage charges to the participants.

In the next section we shall introduce an alternative working set definition that facilitates the accounting and billing procedures by making working set always disjoint.

The method most frequently proposed for handling charges on the non-main-memory shared information is based on a concept of ownership. Each segment is assigned exactly one owner. Anyone wishing to use another's segment must make arrangements to do so with the owner. The owner is charged for use of the segment, regardless of who is actually using it; he is in turn paid royalties by borrowers, these fees fixed to defray those expenses charged to him because borrowers have used his segment[1].

[1] The owner method of charging for sharing very much resembles copyrights. A similar problem is the so-called proprietary software problem, in which a firm or user may lease programs to other firms or users.

One of the chief motivations for this method is simplicity of implementation: because an arbitrarily large and unpredictably varying number of processes may wish to share a single segment, it can become unbelievably complex to keep track of, and attribute charges to, every participant. If this method is used for information shared in main memory, the following inequity will result. Two users sharing the same segment both pay the same fee to the owner (there is no way to determine in advance how much a given user will use it); yet one user may use it sparingly, the other heavily. Thus, costs of sharing may not be distributed fairly.

It is apparent that, if the owner method is used, it must be augmented in order to distribute main memory costs more equitably.

5.1.2. Refinement of The Working Set Definition

The basic idea we use here is: rather than associate with each process the pages it has most recently referenced, we associate each page with the process that has most recently referenced it.

Page i belongs to the working set $W_p(t,\tau)$ of process p if and only if:

1. p has referenced i most recently at time s in its virtual time interval $(t-\tau,t)$.

and

2. no other process has referenced i in p's virtual time interval $(t-s,t)$.

Thus,

$$W_p(t,\tau) = \left\{ i \;\middle|\; \begin{array}{l}\text{the most recent reference to i originated} \\ \text{from process p, in p's vt interval } (t-\tau,t)\end{array} \right\}$$

This definition has two consequences:

1. Disjoint working sets. A page is in at most one working set. Therefore if $\omega_p(t,\tau)$ is the working set size for each process p, and if Q is any collection of processes, then the size of their joint working set is

$$\omega_Q(t,\tau) = \sum_{p\varepsilon Q} \omega_p(t,\tau)$$

We may therefore compute the memory demand of any collection of processes simply by adding their working set sizes.

2. Fair distribution of costs. Suppose processes $p_1,\ldots,p_r$ have been sharing page i, independently, for some interval of time, and let n_j denote the number of references process p_j made to page i. Then, on the average, page i spends a fraction

$$f_j = \frac{n_j}{n_1+\ldots+n_r} \qquad (5.1.1)$$

of its time in the working set $W_{p_j}(t,\tau)$, and has contributed f_j to the size of p_j's working set. Thus, a participant is charged in accordance with his degree of participation.

This last relation, eq. 5.1.1, holds only if $p_1,\ldots,p_r$ behave independently. If the shared information is modifiable and protected by interlocks, then the likelihood of correlation is very high. In general, there is no easy way to determine how an interlocked, modifiable piece of data will affect whatever processes attempt to use it, because of data dependence and arbitrary timing.

5.1.3. Implementation

The implementation of Section 4.11 and Figure 4-7 remains unchanged. Refer now to Figure 5-1. The π-field shown there in the page register, whose contents designate the process whose working set contains the page, now is loaded at each reference with the name of the process making the reference.

To be more precise, an information reference is a pair (i,p) where i is the name of the page being referenced and p is the name of the process making the reference. The only modification of Figure 4-7 is simply that p is loaded into the π-field of the page register as the reference is made.

If $\tau<T$, the following difficulty arises. If the process named in the π-field enters page wait, we must be sure that another process does not borrow the page and then discard it before the π-field process completes its page wait. For example, suppose at time t process 1 enters page wait. If process 2 (which is not in page wait during the interval $(t,t+T)$) references a page in process 1's working set just once in the interval $(t,t+T-\tau)$, the page will exit process 2's working set before process 1 terminates page wait, and will not be available for use by process 1.

There is no easy solution to this difficulty. One possibility is to choose $\tau>T$; but if T depends on the rotation time of a device, this may result in undesirably large values of τ. Another possibility (shown in Figure 5-1) is to prevent a change in the contents of the π-field when the process named there is in page wait; but then other processes may obtain references without paying.

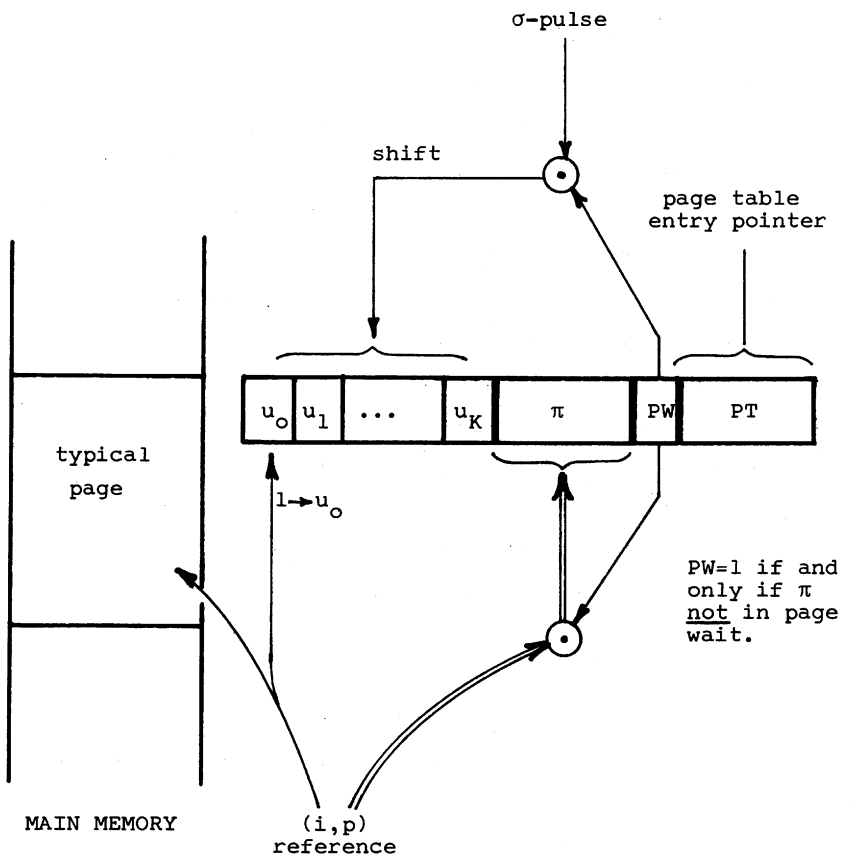
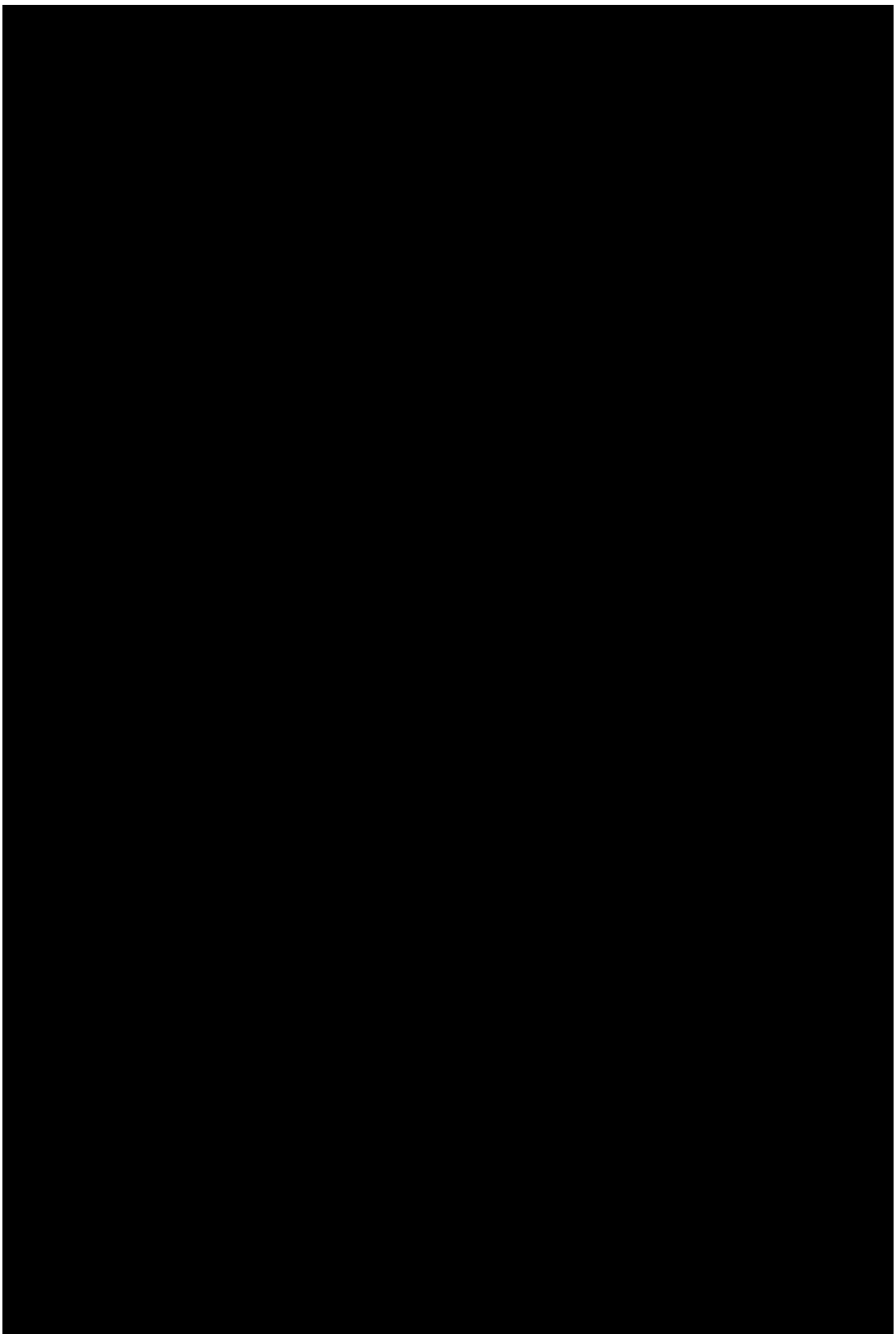
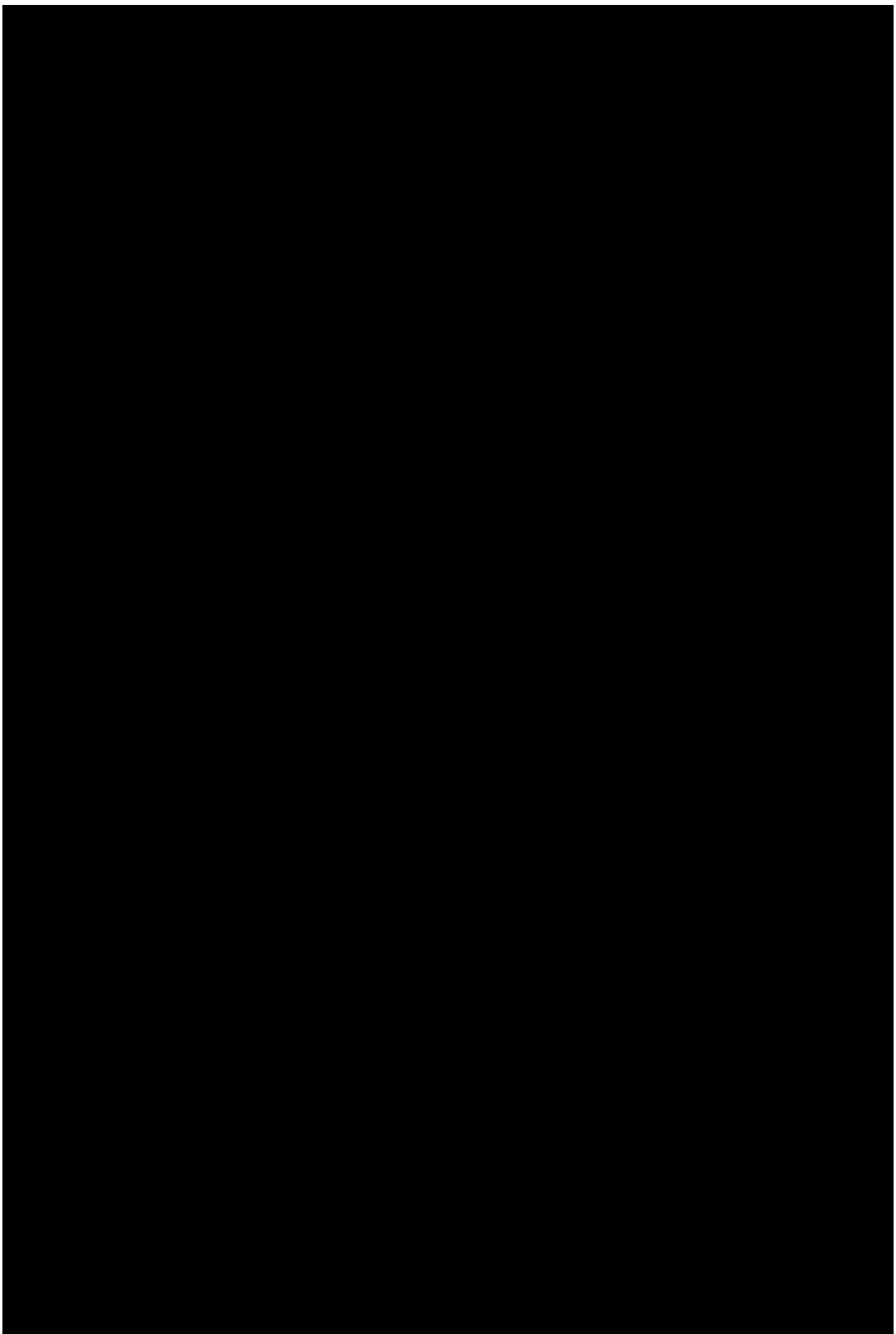
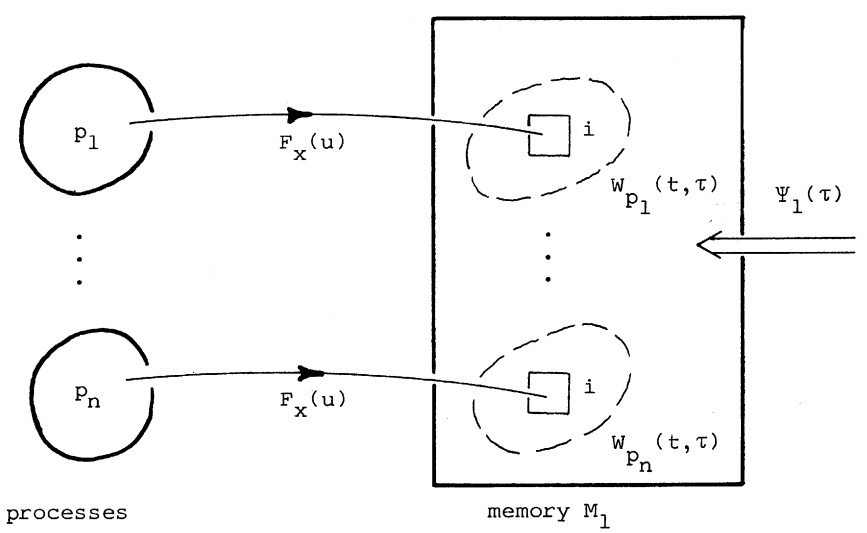


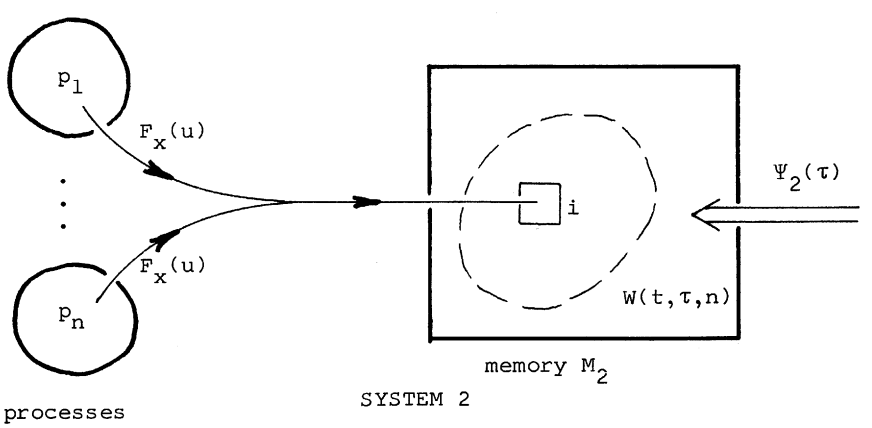
Figure 5-1. Implementation of shared memory management.







SYSTEM 1



SYSTEM 2

Figure 5-2. Experiment to investigate sharing.

System 1 represents the completely unshared case, and is the poorest performance situation. System 2 represents the completely shared case, the best performance situation.

In the following sections, we evaluate the quantities described in the following table.

quantity	symbol	description
expected working set size	$w(\tau,n)$	expected number of pages in the joint working set $W(t,\tau,n)$ of n processes.
missing-page probability	$\lambda(\tau,n)$	probability that a process references a page not in the joint working set $W(t,\tau,n)$.
paging rate	$\rho(\tau,n)$	number of pages per unit real time re-entering $W(t,\tau,n)$ on behalf of one process.
duty factor	$\eta(\tau,n)$	fraction of time a running or page wait process is running, when (n-1) others share the program Z with it.

In each case we show that sharing is an improvement. That is, for $n>1$ and $\tau>0$ we show that

$$\lambda(\tau,n) < \lambda(\tau,1)$$

$$\rho(\tau,n) < \rho(\tau,1)$$

$$\eta(\tau,n) > \eta(\tau,1)$$

and

$$\frac{w(\tau,n)}{n} < w(\tau,1)$$

This last relation differs slightly from the others for the following reason. There may exist a shared page that no one references often enough to keep continuously in main memory,

but that, together, the n processes reference often enough to keep it continuously in main memory. Thus, the joint working set will be larger than any single working set: $w(\tau,n) > w(\tau,1)$. In the shared case, however, each process pays for $\frac{1}{n}$ of the memory used, so his expected cost depends on $\frac{w(\tau,n)}{n}$. By showing $\frac{w(\tau,n)}{n} < w(\tau,1)$ we show that sharing reduces costs.

5.3. Shared Working Set Size and Memory Costs

In Figure 5-2, let $W(t,\tau,n)$ denote the joint working set of the n processes, and define

(5.3.1) $\quad w(\tau,n) = \text{expected size of } W(t,\tau,n)$

When n=1, $w(\tau,1)$ is exactly the expected working set size discussed in Section 4.4. We shall obtain an expression for $w(\tau,n)$ and show that the expected memory cost $\frac{w(\tau,n)}{n}$ for one process is diminished for increased n.

Theorem 5.1. Let n statistically independent processes be simultaneously sharing the same program Z, whose size is fixed at z. The interreference distribution $F_x(u)$ is the same for each process, and is unchanging. Define the integral

(5.3.2) $\quad I(\tau) = \frac{1}{\bar{x}} \int_0^\tau (1-F_x(u))\, du$

Then the expected size of the joint working set of the n processes is

(5.3.3) $\quad w(\tau,n) = z\,[1 - (1 - I(\tau))^n]$

Discussion: Note that, for n=1, we have

$$w(\tau,1) = z\,[1-(1-I(\tau))] = z\,I(\tau) = \int_0^\tau (1-F_x(u))\, du$$

where we have used $z=\bar{x}$ from eq. 5.2.1 (and Theorem 4.1). Thus, the expression reduces to that of Theorem 4.4, in the unshared case.

Proof of Theorem 5.1: We follow an argument similar to that of Theorem 4.4, in which we derived the expected working set size for one process. Let $W_{p_j}(t,\tau)$ be the working set of process p_j, and let $\omega_{p_j}(t,\tau)$ be its size. Define the binary random variable

$$\gamma_i = \begin{cases} 1 & \text{if page } i \text{ in } W_{p_j}(t,\tau) \\ 0 & \text{otherwise} \end{cases} \tag{5.3.4}$$

Observe that $\gamma_i=1$ if and only if p_j referenced page i in $(t-\tau,t)$. Then

$$\omega_{p_j}(t,\tau) = \sum_{i\varepsilon Z} \gamma_i \tag{5.3.5}$$

and

$$w(\tau) = \overline{\omega_{p_j}(t,\tau)} = \sum_{i\varepsilon Z} \overline{\gamma_i} = z \Pr[\gamma_i=1] \tag{5.3.6}$$

Now, from eqs. 4.4.8 and 4.4.9 we have

$$\begin{aligned} \Pr[\gamma_i=0] &= 1 - I(\tau) \\ \Pr[\gamma_i=1] &= I(\tau) \end{aligned} \tag{5.3.7}$$

where $I(\tau)$ stands for

$$I(\tau) = \frac{1}{\bar{x}} \int_o^\tau (1-F_x(u))\, du = \frac{w(\tau)}{\bar{x}} \tag{5.3.8}$$

Now, let $W(t,\tau,n)$ stand for the joint working set of the n processes, and $\omega(t,\tau,n)$ denote its size. Define the binary random variable

$$\delta_i = \begin{cases} 1 & \text{if page } i \text{ in } W(t,\tau,n) \\ 0 & \text{otherwise} \end{cases} \tag{5.3.9}$$

Observe that $\delta_i=1$ if and only if some one of the processes $p_1,\ldots,p_n$ has referenced page i in the interval $(t-\tau,t)$. Then

(5.3.10) $$\omega(t,\tau,n) = \sum_{i \in Z} \delta_i$$

and

(5.3.11) $$w(\tau,n) = \overline{\omega(t,\tau,n)} = \sum_{i \in Z} \overline{\delta_i} = z \Pr[\delta_i=1]$$

We must find $\Pr[\delta_i=1]$. Now, the n processes are statistically independent; then

$$\Pr[\delta_i=0] = \Pr\left[\begin{array}{l}\text{no reference from any of n processes}\\ \text{to page i in the vt interval } (t-\tau,t)\end{array}\right]$$

$$= \left(\Pr\left[\begin{array}{l}\text{no reference from one process to}\\ \text{page i in the vt interval } (t-\tau,t)\end{array}\right]\right)^n$$

$$= \left(\Pr[\gamma_i=0]\right)^n$$

(5.3.12) $$\Pr[\delta_i=0] = (1 - I(\tau))^n$$

thus,

(5.3.13) $$\Pr[\delta_i=1] = 1 - (1 - I(\tau))^n$$

putting this into eq. 5.3.11, we obtain

$$w(\tau,n) = z\,[1 - (1 - I(\tau))^n]$$

QED.

Define the <u>expected memory usage</u> of one of the n processes to be

(5.3.14) $$m(\tau,n) = \frac{w(\tau,n)}{n}$$

so that $m(\tau,n)$ measures the expected cost per unit virtual time attributed to one of the n processes.

Theorem 5.2. Let $m(\tau,n) = \frac{w(\tau,n)}{n}$ be the expected memory usage of one of the n processes, where $w(\tau,n)$ is given by Theorem 5.1, and depends only on the interreference distribution $F_x(u)$. Then for $\tau>0$ and $n>1$,

(5.3.15) $$m(\tau,n) < m(\tau,1)$$

Discussion: Theorem 5.2 asserts that sharing reduces costs. This result is very strong, for it depends only on the arbitrarily given distribution $F_x(u)$. In other words, whenever two or more processes are sharing the same program Z: provided that Z remains fixed, that $F_x(u)$ remains fixed, and that the processes are run concurrently, the shared expected memory usage costs are always less than the unshared memory usage costs. Put another way, sharing is always an improvement under the stated conditions, regardless of program behavior[1].

---

[1]The reader might think there are counterexamples. For example, let the n processes share an interlocked section. A process tests the interlock: if the interlock is ON, the process creates an enormous amount of data; if the interlock is OFF, the process turns it ON and works in the interlocked section. It is clear that n distinct copies require less memory than one shared copy, because in the shared case (n-1) processes will find the interlock ON, whereas in the unshared case no process finds the interlock ON. This violates the assumptions of the theorem, because the program size is not fixed in both cases, and because the interlock violates the assumption of statistical independence. Thus, this is not a counterexample.

Proof of Theorem 5.2: To prove $m(\tau,n) < m(\tau,1)$ we must show that

$$\frac{w(\tau,n)}{n} < w(\tau,1) \tag{5.3.16}$$

from Theorem 5.1, this is the same as showing

$$\frac{1 - (1 - I(\tau))^n}{n} < I(\tau) \qquad \text{all } \tau>0,\ n>1 \tag{5.3.17}$$

This requires that

$$1 - \frac{1 - (1 - I(\tau))^n}{n\, I(\tau)} > 0 \tag{5.3.18}$$

or equivalently that

$$1 - \frac{1 - (1 - I(\tau))^n}{n(1-(1-I(\tau)))} > 0 \tag{5.3.19}$$

This expression is of the form

$$1 - \frac{1-A^n}{n(1-A)} > 0 \qquad A = 1-I(\tau) < 1 \tag{5.3.20}$$

Now, using the fact that

$$\frac{1-A^n}{1-A} = 1 + A + \ldots + A^{n-1} < n \tag{5.3.21}$$

which follows from $A<1$, we have

$$1 - \frac{1-A^n}{1-A} > 1 - \frac{n}{n} = 0 \tag{5.3.22}$$

and the inequality is proved.

QED.

In Figure 5-2, define M_1 to be the total expected memory requirement in system 1, and M_2 to be the total expected memory requirement in system 2. We have the following rather obvious corollary to Theorem 5.2.

Corollary 5.2. $M_1 > M_2$.

Proof: By Theorem 5.2,

$$M_1 = n\, w(\tau,1) > w(\tau,n) = M_2$$

QED.

Corollary 5.2 asserts simply that sharing reduces the overall memory usage, resulting in more memory for other programs to use.

5.4. Missing-Page Probability

Define the missing-page probability to be:

(5.4.1) $\lambda(\tau,n) = \Pr\left[\begin{array}{l}\text{a given process references a missing page,}\\ \text{when (n-1) others share the same program}\end{array}\right]$

Thus, $\lambda(\tau,n)$ is the probability that a process directs a reference to a page not in the joint working set $W(t,\tau,n)$. Using reasoning similar to that of Theorem 4.2, we can see that $\lambda(\tau,n)$ may be regarded as the number of pages per unit virtual time re-entering the joint working set $W(t,\tau,n)$, on behalf of one of the n processes. That is, the expected virtual time interval between the page faults of one process is $1/\lambda(\tau,n)$.

Theorem 5.3. Let $\lambda(\tau,n)$ be the missing-page probability as just defined, and let $F_x(u)$ be the interreference distribution. Then

(5.4.2) $$\lambda(\tau,1) = 1 - F_x(\tau)$$

and

(5.4.3) $$\lambda(\tau,n) = \lambda(\tau,1)(1 - I(\tau))^{n-1}$$

where

(5.4.5) $$I(\tau) = \frac{1}{\bar{x}} \int_0^\tau (1-F_x(u))\, du$$

Proof: From Theorem 3.3, the single-process, unshared missing-page probability is $\lambda(\tau) = 1-F_x(\tau) = \lambda(\tau,1)$. To find $\lambda(\tau,n)$ we note

$$\lambda(\tau,n) = \Pr\left[\begin{array}{l}\text{given process, say p, references page}\\ \text{at time t and finds it missing}\end{array}\right]$$

$$= \Pr\left\{\begin{array}{l}\text{p's most recent reference interval x to}\\ \text{the page in question satisfies } x>\tau, \underline{\text{and}}\\ \text{the (n-1) other processes made no refer-}\\ \text{to the page in question during } (t-\tau,t).\end{array}\right\}$$

Since the processes are statistically independent, this last probability becomes

$$\lambda(\tau,n) = \Big(\Pr[x>\tau]\Big)\Big(\Pr[\text{no reference to page in } (t-\tau,t)]\Big)^{n-1}$$

$$= (1-F_x(\tau))(1-I(\tau))^{n-1}$$

$$= \lambda(\tau,1)(1-I(\tau))^{n-1}$$

where the probability $(1-I(\tau))$ is obtained from the arguments given in Theorem 5.1.

QED.

The next theorem asserts that sharing reduces the missing-page probability.

Theorem 5.4. Under the given assumptions ($F_x(u)$ unchanging, processes running concurrently) the missing-page probability $\lambda(\tau,n)$ is reduced by sharing:

(5.4.7) $\qquad \lambda(\tau,n) < \lambda(\tau,1) \qquad$ if $\tau>0$, $n>1$

Proof: Since $(1-I(\tau)) < 1$, if $\tau>0$ it follows that $(1-I(\tau))^{n-1} < 1$, and thence $\lambda(\tau,n) = \lambda(\tau,1)(1-I(\tau))^{n-1} < 1$.

QED.

Recalling the discussion of Chapter 3, in which we showed that lower missing-page probabilities are equivalent to lower memory-usage costs, Theorem 5.4 verifies that sharing reduces memory-usage costs. Indeed, in many circumstances it will be true that

$$\lambda(\tau,n) \ll \lambda(\tau,1)$$

that is, sharing is a pronounced improvement.

5.5. Paging Rate

Define the real time paging rate to be:

(5.5.1) $\rho(\tau,n)$ = [real time paging rate of one process when (n-1) others are sharing the same program]

That is, one process expects to see a real time interval of length $1/\rho(\tau,n)$ between page waits.

Theorem 5.5. Let $\rho(\tau,n)$ be the real time paging rate as defined above. Then:

$$\rho(\tau,n) = \frac{\lambda(\tau,n)}{1 + \lambda(\tau,n)T} \tag{5.5.2}$$

where $\lambda(\tau,n)$ is the missing-page probability, defined in Theorem 5.3, and T is the traverse time.

Proof: In a virtual time interval of length V, one process generates V information references and expects to encounter $V\lambda(\tau,n)$ page waits. Therefore:

$$\rho(\tau,n) = \frac{\text{(number of page waits)}}{\text{(virtual time) + (page wait time)}} = \frac{V\lambda(\tau,n)}{V + V\lambda(\tau,n)T}$$

QED.

If n=1 we obtain $\rho(\tau,1) = \rho(\tau)$, must like Theorem 4.3.

Let us compare the total page traffic in the two cases. In Figure 5-2, let the total paging rates be denoted by

$$\begin{aligned} \Psi_1(\tau) &= n\,\rho(\tau,1) \\ \Psi_2(\tau) &= n\,\rho(\tau,n) \end{aligned} \tag{5.5.3}$$

We wish to show that sharing reduces page traffic.

Theorem 5.6. Let $\Psi_1(\tau) = n\rho(\tau,1)$ be the unshared total page traffic, and let $\Psi_2(\tau) = n\rho(\tau,n)$ be the shared page traffic. Then

$$\Psi_2(\tau) < \Psi_1(\tau) \tag{5.5.4}$$

Proof: We must show $\rho(\tau,1)-\rho(\tau,n) > 0$ if $n>1$. Consider

$$\rho(\tau,1) - \rho(\tau,n) = \frac{\lambda(\tau,1)}{1 + \lambda(\tau,1)T} - \frac{\lambda(\tau,n)}{1 + \lambda(\tau,n)T}$$

$$= \frac{\lambda(\tau,1) - \lambda(\tau,n)}{(1 + \lambda(\tau,1)T)(1 + \lambda(\tau,n)T)}$$

$$> 0$$

where we have used $\lambda(\tau,1)-\lambda(\tau,n) > 0$ from Theorem 5.4.

QED.

5.6. Duty Factor

Define the duty factor to be

(5.6.1) $\eta(\tau,n)$ = [duty factor of one process when it is sharing its program with (n-1) others]

Recall that $\eta(\tau,n)$ is the fraction of time a process is in the running state as opposed to the page wait state; thus, $\eta(\tau,n)$ measures the ability of a process to use a processor.

Theorem 5.7. Let $\eta(\tau,n)$ be the duty factor, as defined above. Then

$$\eta(\tau,n) = \frac{1}{1 + \lambda(\tau,n)T} \tag{5.6.2}$$

where $\lambda(\tau,n)$ is the missing-page probability (Theorem 5.3), and T is the traverse time.

Proof: In a virtual time interval of length V, the process encounters $V\lambda(\tau,n)$ page waits. Then

$$\eta(\tau,n) = \frac{\text{(virtual time)}}{\text{(virtual time) + (page wait time)}}$$

$$= \frac{V}{V + V\lambda(\tau,n)T}$$

QED.

If n=1, we have $\eta(\tau,1) = \eta(\tau)$, just like Theorem 4.6.

Theorem 5.8. Sharing increases the duty factor:

$$\eta(\tau,n) > \eta(\tau,1) \qquad \text{if } n>1,\ \tau>0 \tag{5.6.3}$$

Proof: Since $\lambda(\tau,1) > \lambda(\tau,n)$ we must have

$$1 + \lambda(\tau,1)T > 1 + \lambda(\tau,n)T$$

hence

$$\frac{1}{1 + \lambda(\tau,1)T} < \frac{1}{1 + \lambda(\tau,n)T}$$

QED.

Again, under many circumstances $\lambda(\tau,1) >> \lambda(\tau,n)$ and it is not difficult to obtain $\eta(\tau,n) \approx 1$, even for small n. Thus, sharing can result in markedly increased processing efficiency.

5.7. Variable Number of Participants

It is often the case that the number of participants in a sharing problem is not fixed; instead, the number is a random variable. The convexity theorem (Theorem 3.1) enables us to obtain bounds on $w(\tau,n)$, $m(\tau,n)$, $\lambda(\tau,n)$, $\rho(\tau,n)$, and $\eta(\tau,n)$ when the average value $\bar{n}$ of n is known but the distribution of n is not. These bounds are summarized in the following table.

quantity	symbol	convexity (in n)	bound
expected working set size	$w(\tau,n)$	convex	$\overline{w(\tau,n)} \leq w(\tau,\bar{n})$
one-process memory demand (pages)	$m(\tau,n)$	convex	$\overline{m(\tau,n)} \leq m(\tau,\bar{n})$
missing-page probability	$\lambda(\tau,n)$	concave	$\overline{\lambda(\tau,n)} \geq \lambda(\tau,\bar{n})$
paging rate	$\rho(\tau,n)$	concave	$\overline{\rho(\tau,n)} \geq \rho(\tau,\bar{n})$
duty factor	$\eta(\tau,n)$	convex	$\overline{\eta(\tau,n)} \leq \eta(\tau,\bar{n})$

In the most general n-process sharing problem, information can be in use by any combination of processes, and each possible combination will be sharing different subsets of information. Suppose Z is the program in use by processes $p_1,\ldots,p_n$. We can partition Z into as many as 2^n-1 blocks, such that exactly some subset of $p_1,\ldots,p_n$ is using each block. Each block associated with just one process behaves as system 1 in Figure 5-2. Each block associated with more than one process behaves as system 2 in Figure 5-2, having higher per-process efficiency, lower per-process memory-usage costs, and lower paging rates. Therefore

the net effect across the program Z is better than the situation when $p_1,\ldots,p_n$ share nothing at all. Thus, system 1 represents the worst case behavior and system 2 represents the best case behavior; any actual system would fall in between these two extremes.

5.8. Summary

When working sets are defined so that a page belongs only to one working set at a time, namely the one of the process that most recently referenced it, memory usage costs tend to be distributed among participants in accordance with their degrees of participation. Implementation is straightforward.

Using a simple model with complete sharing we were able to obtain strong results that quantitatively verify intuitive ideas: providing processes are run concurrently and the interreference distribution is unchanging, sharing always improves performance, regardless of the particular interreference distribution. In many situations the improvements can be very pronounced.

Processes sharing information must be run concurrently (requiring multiple processors) whenever they are not blocked because

1. If run at widely separated intervals, the same information must (unnecessarily) be reloaded.
2. It is only when references are arriving concurrently to shared information that the benefits obtain.

The results obtained here apply to a collection of n statistically independent processes, without regard to whether they are components of multiprocess computations or single-process computations. Thus, it should be clear that multiprocess computations, by permitting interprocess information sharing, can be very efficient, provided that processor-switching time is small and there are enough processors to permit the parallel operation of many processes.

CHAPTER 6

Demands and Balance

6.0. Introduction

We regard a computation, a collection of mutually cooperating processes and information operating within a common name space, as being the fundamental demand-making entity in a computer system. A computation manifests itself by demanding the joint use of processor and memory resources.

Because we want a computation to operate effectively as a unit, we believe it is necessary to allocate resources to a computation as a unit. We therefore assume that the entities being scheduled for service are computations. This is a generalization of existing scheduling philosophies, which call for scheduling of processes.

Let C be a computation, with working set $W_C(t,\tau)$. The working set size $\omega_C(t,\tau)$ will be used to define C's memory demand.

If, on the one hand, C is a single-process computation, its expected running time beyond the present will be used to define

its processor demand. If, on the other hand, C is a multiprocess computation, the number of active component processes will be used to define its processor demand. We make this distinction because in multiprocess computations the number, rather than the duration, of processes is important, whereas in single-process computations the duration of the process is important.

Each computation will be assigned a system demand consisting jointly of its processor and memory demands. Computations requiring the use of system resources will be segregated: those in the standby set temporarily receive no service, whereas those in the balance set receive service. The system is balanced when the total demand of the balance set matches the available equipment[1]. A balance policy is a resource allocation policy that regulates membership in the balance set so that the system remains balanced.

We shall study all these concepts in more detail, then examine general properties of balance policies, and conclude the chapter with a survey of the pertinent literature.

---

[1]Recall that the N processors and M pages of main memory constitute the equipment. Because we may wish to hold some equipment in reserve, we assume that constants α and β have been given (we shall discuss how in Chapter 8), where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$, and we will say that αN processors and βM main memory pages constitute the available equipment.

6.1. Memory Demand

We have defined a working set memory management policy to be one that permits a process to run if and only if there is enough uncommitted space in main memory to accommodate its working set. Working set pages fill these uncommitted slots on demand. Thus, the working set size $\omega(t,\tau)$ is a useful measure of memory demand.

Let C be a computation consisting of processes $p_1,\ldots,p_n$. If $W_{p_j}(t,\tau)$ is the working set of process p_j, then the working set of C is

$$W_C(t,\tau) = \bigcup_{j=1}^{n} W_{p_j}(t,\tau) \tag{6.1.1}$$

If we use the working set definition given in Section 5.1 (a page is in the working set of whatever process most recently referenced it), working sets will be disjoint, and their sizes add:

$$\omega_C(t,\tau) = \sum_{j=1}^{n} \omega_{p_j}(t,\tau) \tag{6.1.2}$$

We define the memory demand $m_C(t)$ of computation C at time t to be:

$$m_C(t) = \min\left(1, \frac{\omega_C(t,\tau)}{M}\right) \qquad 0 \le m_C(t) \le 1 \tag{6.1.3}$$

where M is the number of pages comprising main memory, and $\omega_C(t,\tau)$ is the size of C's working set.

Clearly, $m_C(t)$ represents the fraction of the memory resource demanded by C at time t. If C's working set $W_C(t,\tau)$ contains more than M pages (it exceeds memory) we regard its

memory demand as being $m_C(t)=1$ because it is demanding the entire resource. Presumably M is large enough so that the probability $Pr[m_C(t)=1]$ (over the ensemble of all computations) is very small.

The definition of memory demand applies to any computation, whether it be single-process or multiprocess.

6.2. Processor Demand

We assume that the processor demand of a multiprocess computation depends on the number of processes, whereas the processor demand of a single-process computation depands on the duration of the process.

In contemporary computer systems, the page wait time T depends mostly on the rotation time of a device. Because the switching time of a processor is relatively much smaller than T, it is worthwhile to switch a processor to a second process during a page wait of the first.

To be more precise, let S represent the time required to switch a processor from one process to another. S is actually the expectation of a random variable composed of electronic switching times and scheduling delays. If T>S, it is not economical to dedicate a processor to a process during a page wait, whereas, if $T \leq S$, it is economical to do so.

Define the binary random variable $\pi(t)$ for a given process at time t to be:

$$\pi(t) = \begin{cases} 1 & \text{if a processor is assigned to the process at time t} \\ 0 & \text{otherwise} \end{cases} \tag{6.2.1}$$

This quantity $\pi(t)$ is related to the processor demand of a process. The relationships among process states, memory demand, $\pi(t)$, the traverse time T, and the processor switching time S, are summarized in the following table.

process state	memory demand	processor demand
blocked	$m(t) = 0$	$\pi(t) = 0$
ready	$m(t) > 0$	$\pi(t) = 0$
running	$m(t) > 0$	$\pi(t) = 1$
page wait during $(t,t+T)$	$m(t) > 0$ and $m(t+T) = m(t)+\frac{1}{M}$	$\pi(u) = 1$ if $T \leq S$ or $\pi(u) = 0$ if $T > S$ } $u \varepsilon (t,t+T)$

We have assumed that the entities to be scheduled (hereafter called jobs) are computations -- specific sets of processes rather than individual processes. Thus, we assume that all a computation's non-blocked processes are running (or page wait), or else all such processes are ready. We define the states of a computation to be:

1. enabled: all non-blocked processes are running or page wait.
2. standby: all non-blocked processes are ready.
3. disabled: all processes are blocked.

In our work here, only these states are permitted.

Correspondingly, we define the working set of processes $P(C,t)$ of a computation C at time t to be:

$$(6.2.2) \quad P(C,t) = \begin{cases} \{\text{non-blocked processes in C at time t}\} & \text{if C enabled or standby} \\ \emptyset & \text{if C disabled} \end{cases}$$

where $\emptyset$ is the empty set. Note that $P(C,t)$ is well-defined even if C is a single-process computation.

Using these ideas, we shall define processor demand in both the single-process and multiprocess computation cases.

6.2.1. Multiprocess Computations

In this case, the processor demand is concerned with the number of processes in a computation, because the computer system must know how many processing units to assign.

Let C be a multiprocess computation. We define C's processor demand $p_C(t)$ at time t to be:

$$(6.2.3) \qquad p_C(t) = \min\left(1, \frac{|P(C,t)|}{N}\right) \qquad 0 \leq p_C(t) \leq 1$$

where N is the number of processors, and P(C,t) is the working set of processes in C (eq. 6.2.2).

It is clear that $p_C(t)$ represents the fraction of the processor resources needed by C at time t. Presumably N is large enough so that the probability $Pr[p_C(t)=1]$ (over the ensemble of all computations) is very small.

Note the symmetry between the definitions of processor and memory demand (eqs. 6.1.3 and 6.2.3), in the case of multiprocess computations.

6.2.2. Single-process Computations

In the case of single-process computations, P(C,t) contains at most one process; so we must be concerned with its duration in order to know how long to assign a processor to it. Thus, computer systems in which single-process computations predominate (systems such as Multics or IBM System 360) must use a somewhat different definition of processor demand. Because of this, we are unable to completely preserve the symmetry between the definitions of processor and memory demand.

We should like to define processor demand in the single-process case so that a processor demand has a meaning in the time domain analogous to the meaning of a memory demand in the space domain. A useful method (by no means the only one) is described in the following paragraphs.

Let the random variable q denote the virtual time interval between interactions. It has been found [C4,F4] that the probability density function for q, $f_q(u)$, may be modelled by a hyperexponential distribution:

$$f_q(u) = cae^{-au} + (1-c)be^{-bu} \qquad \begin{matrix} 0 < a < b \\ 0 < c < 1 \end{matrix} \tag{6.2.4}$$

$f_q(u)$ is diagrammed in Figure 6-1; most of the probability is concentrated toward small q (i.e., frequently interacting processes), but $f_q(u)$ has a long exponential tail.

Given that it has been γ vtu since the last interaction, the conditional density function for the time beyond γ until the next interaction is

$$f_{q|\gamma}(u) = \frac{f_q(u+\gamma)}{\int_\gamma^\infty f_q(v)\,dv} \qquad u \geq 0 \tag{6.2.5}$$

which is just that portion of $f_q(u)$ for $q \geq \gamma$ with its area normalized to unity. The conditional expectation function $Q(\gamma)$ is

$$Q(\gamma) = \int_o^\infty u\, f_{q|\gamma}(u)\,du \tag{6.2.6}$$

$$= \frac{\frac{c}{a}e^{-a\gamma} + \frac{1-c}{b}e^{-b\gamma}}{ce^{-a\gamma} + (1-c)e^{-b\gamma}}$$

$Q(\gamma)$ is the expected time beyond γ until the next interaction; it is illustrated in Figure 6-2. It starts at

$$Q(0) = \frac{c}{a} + \frac{1-c}{b} \tag{6.2.7}$$

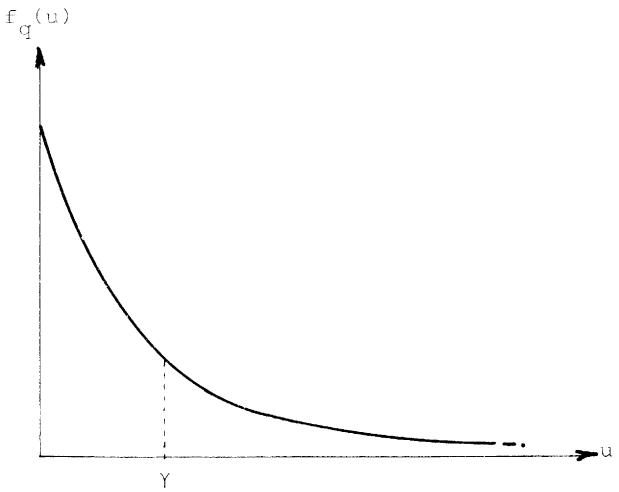


Figure 6-1. Probability density function $f_q(u)$.

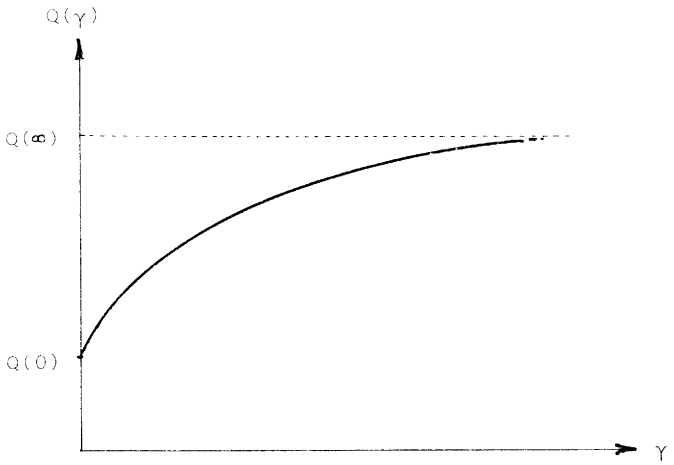


Figure 6-2. Conditional expectation function $Q(\gamma)$.

and rises toward a constant maximum

$$(6.2.8) \qquad Q_{max} = Q(\infty) = \frac{1}{a}$$

Note that, for large γ, the conditional expectation $Q(\gamma)$ becomes independent of γ.

The conditional expectation function $Q(\gamma)$ is a useful prediction function -- if a process has consumed γ vtu since its last interaction, we may expect it to consume an additional $Q(\gamma)$ vtu before its next interaction.

A reasonable choice of quantum to allocate a process might be $kQ(\gamma)$ for some suitable constant $k \geq 1$. It should be clear that $Q(\gamma)$ can be measured and updated from time to time by the computer system.

We should point out that this notion -- a conditional expectation function to predict processor usage -- is very useful and quite independent of the hyperexponential distribution hypothesis[1]. We have formulated it in terms of the hyperexponential because the hyperexponential is a good model and because the hyperexponential has the interesting property that the prediction function $Q(\gamma)$ becomes independent of γ for large γ.

Just as we are unwilling to commit more than M pages of memory, so we may be unwilling to commit processor time for more than a standard interval A into the future. This interval A can be chosen to reflect the maximum tolerable response time to a user: for if the set of processes receiving service has total

---

[1] That is, other prediction functions might be used. The CTSS scheduler [C6,S3], for example, happens to use the prediction function $Q(\gamma)=\gamma$.

expected time consumption not exceeding A, then no process in this set expects to wait longer than A before its own interaction. Just as M is a space constraint, so A is a time constraint.

We define the processor demand $p_C(t)$ of a single-process computation C at time t to be:

$$(6.2.9) \qquad p_C(t) = \frac{Q(\gamma_C)}{N\,A} \qquad \frac{Q(0)}{NA} \leq p_C(t) \leq \frac{Q_{max}}{NA}$$

where γ_C is the time used by C's process since its last interaction.

Since N processors have NA units of time to be committed among them, $p_C(t)$ is the fraction of this total that C is expected to need before its next interaction. Note that this definition of processor demand is just the previous definition (eq. 6.2.3), with $|P(C,t)| = 1$, multiplied by the expected duration of processor use. It is no longer symmetric with the definition of memory demand (eq. 6.1.3).

6.3. System Demand

We define the system demand $\underset{\sim}{d}_C(t)$ of a computation C at time t to be a pair

(6.3.1) $$\underset{\sim}{d}_C(t) = (p_C(t), m_C(t))$$

where $p_C(t)$ and $m_C(t)$ are the processor and memory demands of C.

That the processor demand is $p_C(t)$ tells us to expect C's immediate processor need to be $Np_C(t)$ processors[1]. That the memory demand is $m_C(t)$ tells us to expect C's immediate memory need to be $Mm_C(t)$ pages.

This definition applies to C being either a multiprocess or a single-process computation. It expresses the dual manifestation of C, as a demand for processors and as a demand for memory. $\underset{\sim}{d}_C(t)$ must be considered as a two-dimensional random variable, with unknown correlations between $p_C(t)$ and $m_C(t)$.

---

[1] If C is a single-process computation, then $p_C(t)$ tells us to expect C to require one processor for $NAp_C(t)$ vtu.

6.4. System Balance

Let numbers α and β be given, where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$, and let $\underset{\sim}{D}(t)$ represent the total demand presented by enabled computations:

$$\underset{\sim}{D}(t) = \sum_{C \in B} d_C(t) \qquad B = \left\{ \begin{matrix} \text{enabled} \\ \text{computations} \end{matrix} \right\} \tag{6.4.1}$$

The computer system is said to be balanced at time t if

$$\underset{\sim}{D}(t) = (\alpha, \beta) \tag{6.4.2}$$

The system is processor-balanced if

$$\sum_{C \in B} p_C(t) = \alpha \tag{6.4.3}$$

The system is memory-balanced if

$$\sum_{C \in B} m_C(t) = \beta \tag{6.4.4}$$

That the system is balanced means that the total resource requirement of enabled computations is simultaneously for αN processors and for βM main memory pages.

The resource allocation problem is to decide dynamically which computations to enable so that balance is maintained. This set of enabled computations at time t will be called the balance set B. In general, the system will not be balanced at each instant of time; instead there will be a sequence of instants, called the decision points, at which the demand of B is made to return to the desired demand (α, β) by admitting or removing computations from the balance set B.

One of the chief advantages of balance is simply that the balance set B presents (at least at decisions points) a known demand; that is,

$$\left.\begin{aligned} p_B(t) &= \alpha \\ m_B(t) &= ß \end{aligned}\right\} t \text{ a decision point} \tag{6.4.5}$$

A major design problem, one we shall discuss in Chapter 8, is that of determining the balance parameters α and ß. These parameters will be chosen so that, just before a decision point t, the probabilities

$$\begin{matrix} \Pr\left[\sum_{C\varepsilon B} p_C(t-\delta) \geq 1\right] \\ \Pr\left[\sum_{C\varepsilon B} m_C(t-\delta) \geq 1\right] \end{matrix} \qquad \delta > 0 \tag{6.4.6}$$

are as small as desired.

In Figure 6-3 we have diagrammed the flow of jobs (i.e., computations) among the states enabled, standby, and disabled. New jobs enter the standby set. The scheduler regulates membership in the balance set so that balance is maintained. If a computation becomes disabled, it enters the disabled set. These points should be noted:

1. Each job in the standby set has its demand associated with it. When a new job enters the standby set, an estimate of demand must be associated with it. In the absence of reliable predictive information, the best estimate is $(\bar{p},\bar{m})$, where $\bar{p}$ is the average processor demand over all computations, and $\bar{m}$ is the average memory demand over all computations.

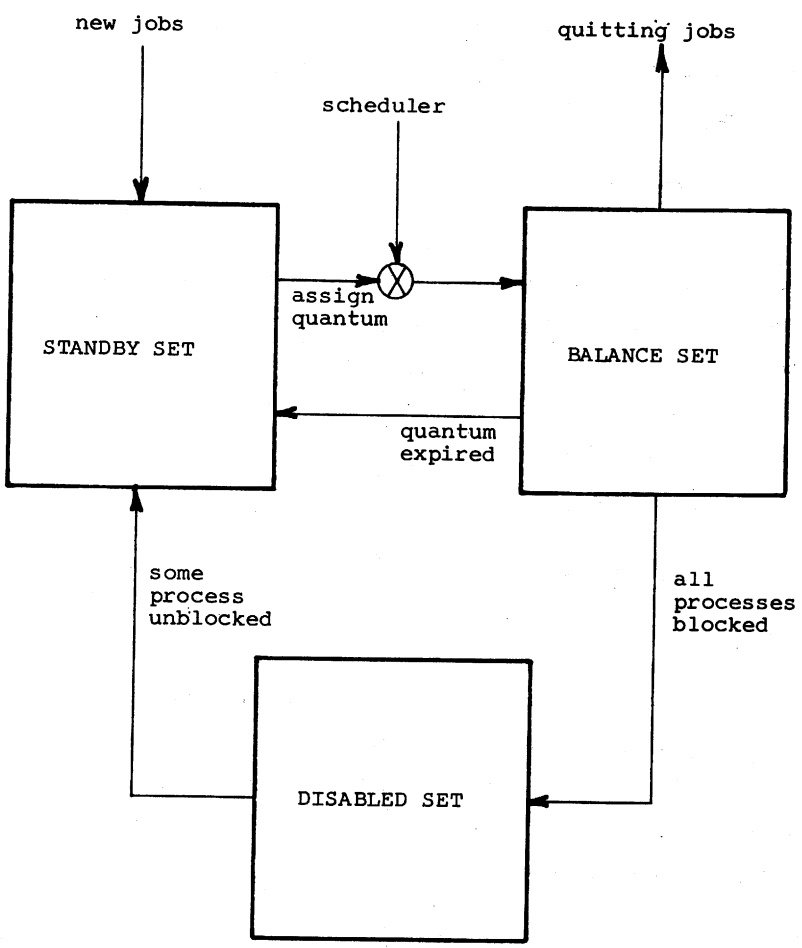


Figure 6-3. Job flow in balanced computer system.

2. In general, when the scheduler admits a new job to the balance set, it allocates a quantum q to the job, where q represents the total virtual-time processor consumption permitted to the job's processes. If q expires (at time t), the job is returned to the standby set, with its current demand (p(t),m(t)).
3. The balance set contains a mixture of running and page wait processes, together with their working sets. Memory management follows a working set strategy.

In the case of single-process computations, the following terminology (from Multics) is often used. Any process in the standby set is said to be ready, and so the standby set may be called the ready list. Any process in the disabled is blocked, and so the standby set may be called the blocked list. Any process in the balance set is either running or page-wait, and so the balance set may be called the running list. There is, however, a very important difference with Multics: here, a page-wait process remains a member of the balance set; in Multics, a page-wait process is regarded as being blocked.

6.5. Balance Policies

A balance policy is a resource allocation policy that keeps the computer system balanced. It is implemented by the scheduler shown in Figure 6-3, which regulates membership in the balance set. Expressed as a minimization problem, a balance policy is:

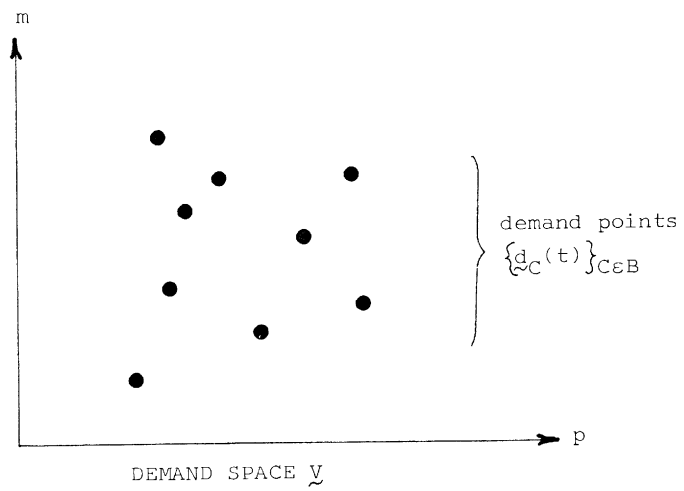
(6.5.1) $\{\text{minimize } |\underset{\sim}{D}(t)-(\alpha,\beta)|\}$

where $|\underset{\sim}{D}(t)-(\alpha,\beta)|$ stands for componentwise minimization.

6.5.1. Demand and Usage Spaces

To help visualize the operation of a balance policy, it is useful to define two spaces: the demand space $\underset{\sim}{V}$ and the usage space $\underset{\sim}{U}$. We regard $\underset{\sim}{V}$ and $\underset{\sim}{U}$ as being two-dimensional: a typical point (p,m) in either space is capable of representing the demand of some computation. The demand space $\underset{\sim}{V}$ contains a set of specially designated points, the demand points, one representing the demand $\underset{\sim}{d}_C(t)$ of each enabled computation C in the balance set. The demand points are time-varying in position. The usage space $\underset{\sim}{U}$ contains two specially designated points: the actual demand point $\underset{\sim}{D}(t)$ and the desired demand point (α,β). A balance policy tries to move the actual demand point, along some path, closer (in the sense of eq. 6.5.1) to the desired demand point. These ideas are illustrated in Figure 6-4.

Unfortunately we must be careful not to interpret the spaces $\underset{\sim}{V}$ and $\underset{\sim}{U}$ as metric spaces, because the path $\underset{\sim}{D}(t)$ follows when it moves toward (α,β) affects system behavior. If these spaces were metric spaces we would be able to assign a magnitude, say $\mu(\underset{\sim}{d})$, to a demand $\underset{\sim}{d}$, which would in turn imply that system performance



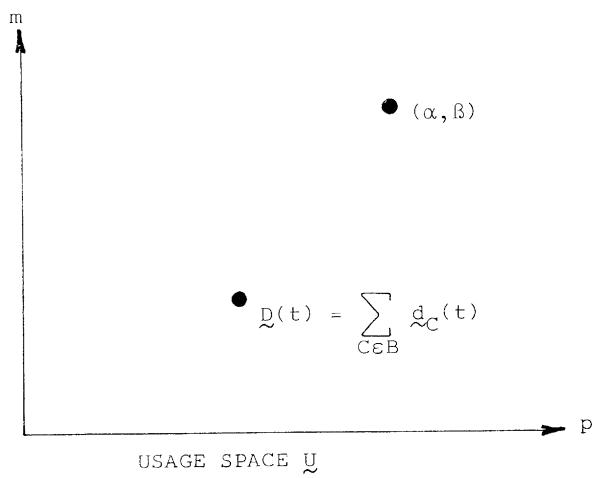


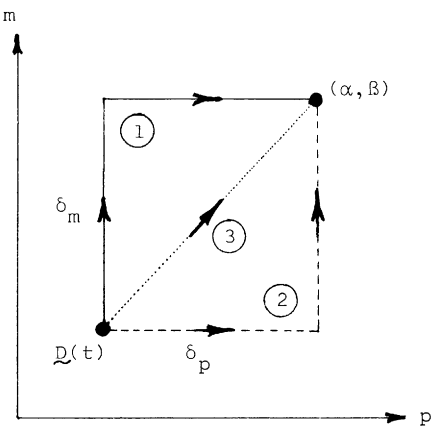
Figure 6-4. Demand and Usage spaces.

would depend only on the magnitude of the imbalance, $\mu(\underset{\sim}{D}(t)-(\alpha,\beta))$. The following argument shows that this is not the case.

Figure 6-5 shows how system performance will depend on the path. The paths shown are:

1. path 1. (first balance memory, then processor.) First examine the standby set for a subset of computations with memory demand δ_m; second, select from this subset a computation with processor demand δ_p. In the first step the whole standby set is examined, so it is highly probable a computation with memory demand δ_m will be found. In the second step only a subset of computations (those with memory demand δ_m) is examined, so it is less likely a computation with processor demand δ_p will be found. The result is that memory usage is tightly distributed about βM, whereas processor usage is loosely distributed about αN.
2. path 2. (first balance processor, then memory.) This has exactly the opposite effect as path 1, the memory usage being loosely distributed about βM, the processor usage being tightly distributed about αN.
3. path 3. Balance both processor and memory simultaneouly by examining the standby set for a computation whose demand is exactly (δ_p,δ_m). This has an effect intermediate between those of path 1 and path 2, the pro- and memory usage tending to be equally distributed about the desired points.

Now in itself the path effect need not interfere with performance. But in computer systems in which the traverse time is large and



USAGE SPACE $\underset{\sim}{U}$

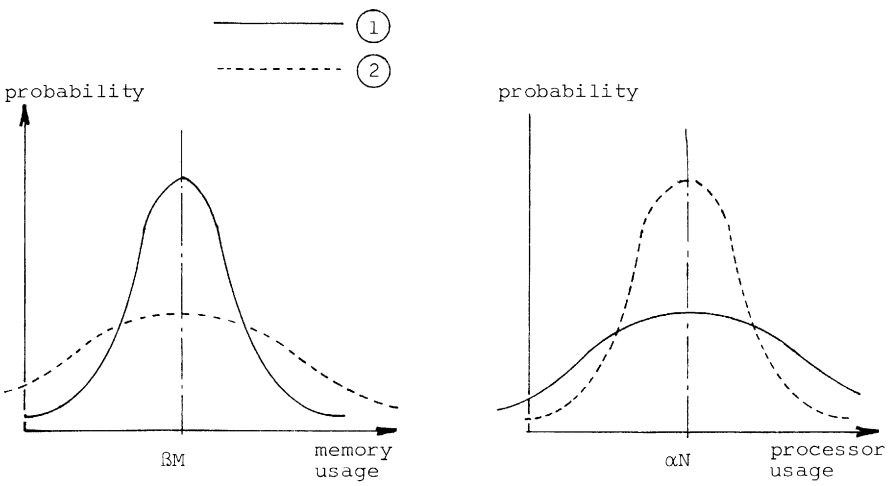


Figure 6-5. The path effect.

computations are single-process, it is extremely important to balance memory properly in order to avoid thrashing, and to avoid accumulating too many traverse times from returning pages. In such systems path 1 is the best path. In such systems we can tolerate higher imbalance in processor usage, because (see eq. 6.2.8) the standard interval A is a delay or response constraint and is therefore a value judgment, whereas the memory size M is a physical constraint.

Conversely, in computer systems in which the traverse time is not large and computations are multiprocess, path 3 is the best path because we are equally concerned with processor and memory balance.

6.5.2. Properties of a Balance Policy

Although we shall defer detailed discussion about implementing balance policies until the next chapter, it is nonetheless useful to point out certain properties the implementation should or will have, consistent with the objectives of a multiprocess computer system.

First, the balance criterion is not necessarily an equipment utilization criterion. If (α,β) are set close to (1,1) then certainly equipment is fully utilized. If (α,β) are set much less than (1,1), the service to users is improved because, as we discussed in Chapter 1, there is an inverse tradeoff between utilization and service. Therefore α and β can be regulated by the administration to meet its current objectives.

Second, balance significantly diminishes the possibility of thrashing because, by proper selection of the balance parameters α and β, the probability that the system actually enters an overload condition can be made arbitrarily small.

Third, we shall see in Chapter 7 that an implementation of a balance policy can be made to have the property that the relative computational overhead required to restore balance depends only on the degree of imbalance (i.e., (δ_p,δ_m) in Figure 6-5) and not on the size of the total demand $\underline{D}(t)$. This guarantees that balance is configuration independent in the sense that the same basic strategy scales over a very wide range of loads.

Fourth, the reader will recall that we have assumed fairness should be built in to a balance policy. We will say that a policy is fair if a job's waiting time depends only on its order of arrival relative to jobs of comparable demand, and not on its order of arrival relative to jobs of different demand. Many existing scheduling philosophies, which tend to stall long jobs in deference to shorter jobs, are not fair by this definition. In the next chapter we shall show how to incorporate fairness into a balance policy.

Fifth, balance makes it possible to make jobs independent of one another, in the sense that an increase in the demand of one will not interfere with the resources in use by another. If the balance parameters α and β are set less than unity, then there will be a slack of $(1-\alpha)N$ processors and $(1-\beta)M$ memory pages to absorb just such demand fluctuations. This paves the way to tractable analysis.

Sixth, a balance policy should tend to run two processes concurrently whenever they are sharing information, in order to reap the benefits of sharing.

6.6. Survey of the Literature

Much of the literature is devoted exclusively to problems of processor scheduling or to memory allocation; little of it is devoted to a unified treatment of both.

By far the greatest part of the literature addresses itself to scheduling; a myriad of algorithms and analyses have appeared. Estrin and Kleinrock [E1] and then Coffman and Kleinrock [C3] have very good surveys of the important algorithms. In reference [C3] it is demonstrated that all the algorithms are susceptible to countermeasures: because most algorithms favor small tasks (both in size and duration) either explicitly or implicitly, a user may significantly improve service to himself by subdividing his job into a sequence of small tasks (provided no one else does this too). This does overall efficiency no good.

A variety of papers report on memory allocation [B1,B2,D2, D4,P4,R2]; we have already discussed these in Chapter 3.

Not much reported work deals with interactions between processor and memory. The approaches most often used are either to regard scheduling as the primary allocation function, memory management as the secondary allocation function, or else to regard them both as being independent. It should be obvious by now that in existing systems the problem of memory management is of far greater importance than that of scheduling, on account of the very large traverse time and the serious possibility of thrashing. Therefore, if memory is properly managed, almost any reasonable scheduling algorithm will function well. Conversely, if memory is mismanaged or overloaded, the particular scheduling algorithm will be of little consequence.

No work is reported that gives insight into how scheduling problems are compounded by information sharing. For example, it is clear that scheduling algorithms should tend to run two processes together in time whenever they are sharing information. However no study reports on the extent to which a scheduling algorithm should tend to do this, or whether it should tend to do this explicitly at all.

There has been some work on system balance. It falls into two classes: static balance, the problem of determining an optimum equipment configuration for a given program mix; and dynamic balance, the problem of dynamically adjusting the load to the existing equipment. Nielsen [N1,N2] has reported on simulation work for static balance which has been of considerable help in configuring the Stanford version of IBM System 360. Saltzer [S2] describes some rule-of-thumb performance measurements that may be used to test a system to decide whether or not it is statically balanced or whether it is thrashing.

The most interesting work of all concerns dynamic balance. Oppenheimer and Weizer [O2] report that their simulation of the RCA Spectra 70/46 Time Sharing Operating System verify conclusively that even relatively primitive notions of dynamic memory balance result in markedly improved performance. O'Neill, Belady, and colleagues [O1] have been experimenting with a load-leveler on the M44/44X computer, and have been very pleased with the results.

6.7. Summary

The important concepts introduced here in this chapter all center around the idea of supply-and-demand allocation in large computer systems. Memory demand is based on working set size. Processor demand is based on the intensity or the duration of processor requirements. System demand is a composite of these two types of demand.

Computations requiring resources are divided into two classes: standby set computations, which are temporarily denied use of system resources; and balance set computations, which are granted tne use of system resources. The system is balanced just when the total demand of the balance set matches the available equipment.

A balance policy is a resource allocation policy that regulates membership in the balance set so that system balance is maintained. The demand space and usage space were introduced as conceptual aids to understanding properties of balance policies. An important property, the path effect, is the dependence of performance on the order in which the processor and memory resources are balanced.

We distinguished two aspects of balance. The first aspect, static balance (controlled by the administration) is the problem of matching the equipment configuration to the total demand of the user community. We return to this in Chapter 8. The second aspect, dynamic balance (controlled by the scheduler) is the problem of matching the demand of the balance set to the existing equipment. We direct attention to this in the next chapter.

CHAPTER 7

Implementation of Balance Policies

7.0. Introduction

The general structure and basic properties of a balance policy have been given in Chapter 6. It remains to show how a balance policy can be realized.

The three most important things we are requiring from a balance policy are: first, that it keep the system balanced; second, that it be fair; and third, that it assure reasonable policies with respect to other criteria such as minimum response time.

We distinguish two cases: the one-dimensional case is applicable to contemporary computer systems, in which the threat of thrashing makes memory balance so much more important than processor balance; the two-dimensional case is applicable to future computer systems, in which both processor and memory

balance will be equally important. In one-dimensional cases, we explicitly balance only one resource type and try to achieve reasonable balance of the other resource type, whereas in two-dimensional cases we explicitly balance both resource types simultaneously.

The most important result of this chapter is: we formulate mathematical programming problems whose solutions, found dynamically by the scheduler, are almost-optimum balance policies.

In Section 7.1 we present an analysis of a single-server, first-come, first-served queue, because this can act as a worst case analysis for the behavior of the queue structures we propose. In Section 7.2 we study properties of queue structures that guarantee fair policies, and we find bounds on the processor and memory requirements needed to act as servers to the queues. In Section 7.3 we formulate the one-dimensional mathematical programming problem, and give a simple algorithm that finds the optimum solution in the particular case of memory balance. In Section 7.4 we formulate the two-dimensional mathematical programming problem, but we do not attempt to give solutions.

7.1. Analysis of a Single-Server Queue

The statistics of a first-come, first-served (FCFS) single server queue can be used to obtain the worst case behavior of a balanced computer system.

The queueing system under consideration is shown in Figure 7-1. Job interarrival times are exponentially distributed with mean $\frac{1}{a}$. That is, if $\{t_n\}$ is the sequence of instants at which jobs arrive, the interarrival times $\gamma_n = t_n - t_{n-1}$ are identically distributed according the the density function

$$f_{\gamma_n}(u) = f_{\gamma}(u) = ae^{-au} \qquad u \geq 0 \tag{7.1.1}$$

Similarly, the job service times are exponentially distributed, with mean $\frac{1}{b}$. The rate (a) of job arrivals remains fixed, regardless of the number of jobs in the system; in other words, we regard the source population as being infinite.

Our use of exponential interarrival and service times, and an infinite source population, requires justification.

We are directing the analysis toward large systems, in which a large source population generates the service requests. In these systems, exponential interarrival and service distributions are good models for at least two reasons. First, it is well known that, when a large population generates service requests, the times between arrivals from the population tend to be exponentially distributed, even though the times between arrivals from a particular member of the population do not. The telephone system, for which it has been found that the interarrival and service distributions are very nearly exponential [P3,p.281], is an excellent example of this behavior. Second, there is considerable evidence to indicate that many interarrival and

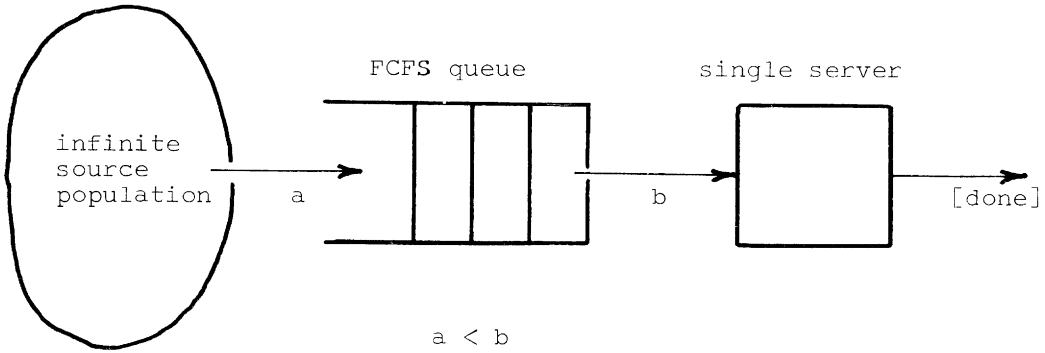


Figure 7-1. Single-server queue.

service distributions are approximately hyperexponential in the case of not too large populations [C4,F4]; these distributions have exponential tails. By assuming exponential arrival and service distributions, we are modelling the tails of the actual distributions, thereby providing a worst case analysis.

Furthermore, the exponential case, interesting in its own right, can yield insights perhaps not obtainable from protracted analysis.

When the source population is finite and the server is saturated, jobs pile up in the queue and, there being fewer requestors remaining in the source population, the arrival rate slackens. Because we are interested in the unsaturated behavior of a computer system our use of infinite source populations (in which the arrival rate is independent of queue length) is not unreasonable. Assuming that balance policies keep the computer system out of saturation, a job will not be seriously delayed in the queues, and the source population will not be seriously depleted. Thus, the arrival rate will not significantly slacken. Indeed, experience has shown that infinite population models approximate finite population systems with surprisingly little error, for population sizes as small as 20 [F2,Vol.1, p.143ff].

Nevertheless, since we are in fact making approximations to actual behavior by using these assumptions, we can only interpret the results as being system averages.

When (n-1) jobs are in the queue and 1 job is in service, the system is in state n. Let π_n denote the steady state probability of state n.

Theorem 7.1. In the queueing system described above, the steady state probability of state n is

$$\pi_n = r^n(1-r) \qquad n = 0,1,2,\ldots \tag{7.1.2}$$

where $r = \frac{a}{b}$. The mean and variance of n are

$$\bar{n} = \frac{r}{1-r}$$
$$\sigma_n^2 = \frac{r}{(1-r)^2} \tag{7.1.3}$$

Proof: In a small time interval dt, the probability of a transition from state (n-1) to n is (a dt); in the same interval dt, the probability of a transition from n to (n-1) is (b dt). Therefore in the steady state we must have

$$\pi_{n-1}\, a\, dt = \pi_n\, b\, dt$$

which means

$$\pi_n = \frac{a}{b}\,\pi_{n-1}$$

Letting $r = \frac{a}{b}$, we have

$$\pi_n = r^n\,\pi_o \qquad r = \frac{a}{b}$$

The generating function for n is

$$G(z) = \sum_{n=0}^{\infty} \pi_n z^n = \pi_o \frac{1}{1-rz} \qquad r < 1$$

Since G(1) = 1, we have $\pi_o = 1-r$ and thus

$$\pi_n = r^n(1-r)$$

which verifies eq. 7.1.2. The mean and variance of n are given

by the usual expressions:

$$\bar{n} = G'(1) = \frac{r}{1-r}$$

$$\sigma_n^2 = G''(1) + G'(1) - \bar{n}^2 = \frac{r}{(1-r)^2}$$

which verify eqs. 7.1.3.

QED.

Theorem 7.2. Let the random variable y denote the time a job waits in the queue described above until it enters service. Then the probability density function $f_y(u)$ is given by

$$f_y(u) = \begin{cases} \frac{b-a}{b} & u=0 \\ \frac{a}{b}(b-a)e^{-(b-a)u} & u>0 \end{cases} \tag{7.1.4}$$

and the mean and variance of y are given by

$$\begin{aligned} \bar{y} &= \frac{a}{b}\frac{1}{b-a} \\ \sigma_y^2 &= \frac{a}{b}\frac{1}{(b-a)^2} \end{aligned} \tag{7.1.5}$$

Proof: Observe that

$$y = \begin{cases} 0 & \text{if } n=0 \qquad \Pr[y=0] = \pi_o = 1-r \\ \sum_{i=1}^{n} s_i & \text{otherwise} \end{cases} \tag{7.1.6}$$

Here, s_i is the random variable of service time for one job, whose density function satisfies

$$f_{s_i}(u) = be^{-bu} \qquad u \geq 0$$

for each subscript i. Eq. 7.1.6 is obtained by noting: if the job arrives to find n already in the system it must wait for all n to complete service.

Suppose $n \geq 1$. We want to find $f_{y,n}(u)$, the density function for y when n are in the system. Observe that

$$f_{y,n}(u)\,du = \Pr[(n-1) \text{ events in } u \text{ and } 1 \text{ event in } (u,u+du)]$$

$$= \frac{(bu)^{n-1}}{(n-1)!} e^{-bu} (b\,du)$$

so that

$$f_{y,n}(u) = b \frac{(bu)^{n-1}}{(n-1)!} e^{-bu}$$

To find $f_{y,n\geq 1}(u)$:

$$f_{y,n\geq 1}(u) = \sum_{n=1}^{\infty} f_{y,n}(u)\,\pi_n = \sum_{n=1}^{\infty} b \frac{(bu)^{n-1}}{(n-1)!} e^{-bu} r^n (1-r)$$

$$f_{y,n\geq 1}(u) = r\,b\,(1-r)\,e^{-b(1-r)u}$$

$$f_{y,n\geq 1}(u) = \frac{a}{b}(b-a)e^{-(b-a)u}$$

since $r = \frac{a}{b}$. Finally,

$$f_{y,0}(u) = \pi_o = \frac{b-a}{b} \qquad \text{at } u=0$$

Dropping the use of the second subscript,

$$f_y(u) = \begin{cases} \frac{b-a}{b} & u=0 \\ \frac{a}{b}(b-a)e^{-(b-a)u} & u>0 \end{cases}$$

which verifies eq. 7.1.4.

The mean and variance of waiting time are

$$\bar{y} = \int_0^\infty u\, f_y(u)\, du = \frac{a}{b}\frac{1}{b-a}$$

$$\sigma_y^2 = \overline{y^2} - \bar{y}^2 = \frac{a}{b}\frac{1}{(b-a)^2}$$

which verify eqs. 7.1.5.

QED.

Theorems 7.1 and 7.2 can be used to find bounds on the number in the system and on the waiting time. A bound on the number n in the system can be obtained from

(7.1.7) $$Pr[n>u] = \sum_{n=u+1}^{\infty} \pi_n = r^{u+1}$$

and a bound on the waiting time y from

(7.1.8) $$Pr[y>u] = \int_u^\infty f_y(v)\, dv = \frac{a}{b}e^{-(b-a)u}$$

7.2. Organization of the Queues

The systems of queues described below are embedded in the standby set. They are organized so that the scheduler can quickly locate jobs of whatever demand it seeks. They are specifically intended for use in contemporary computer systems, in which the grave danger of thrashing makes it so overridingly important to balance memory. They are the queues for use in the one-dimensional case.

The following discussion illustrates how fairness can be incorporated into a balance policy. It also establishes bounds on the total processor and memory requirements needed to accommodate the balance set.

7.2.1. An Almost-Continuous System of Queues

Figure 7-2 illustrates a very general, one-dimensional queueing structure. We assume that jobs (i.e., computations) are arriving at random, interarrival times exponential with mean $\frac{1}{a}$. Job (working set) sizes are integers s, $s\varepsilon[1,s_o]$, s_o being the size of the largest working set. The job size distribtuion (of incoming jobs) is

$$\Pr[s=i] = f_s(i) \tag{7.2.1}$$

and

$$\sum_{i=1}^{s_o} f_s(i) = 1 \tag{7.2.2}$$

Let $Q = \{1,\ldots,k,\ldots,s_o\}$ denote the set of queues. A job of size k is placed at the end of the k^{th} queue.

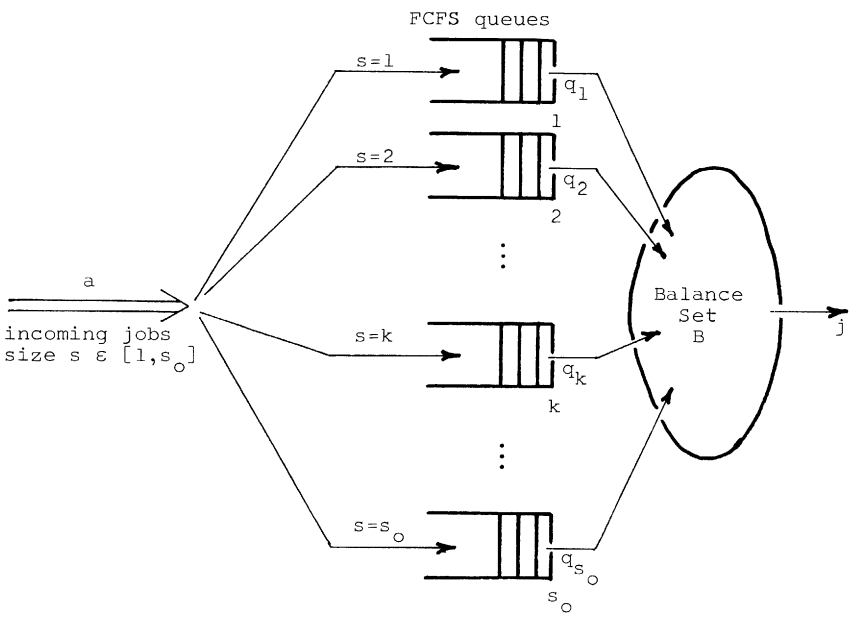


Figure 7-2. Sorting jobs into size classes in standby set.

In our work here, we require that a balance policy be fair. To accomplish this we have made the individual queues FCFS, and we will require that the scheduler keep at least one job from each queue in service. Thus, a job's waiting time will depend only on its order of arrival relative to jobs of the same size, and not on jobs of different size.

Associated with the k^{th} queue is a quantum q_k. The quantum q_k is assumed to be the same for each job in the k^{th} queue, regardless of its past.

The scheduler controls membership in the balance set B so that balance is maintained; that is, so that the balance set demand (p_B, m_B) is kept within close tolerances of the desired (α, β). Because each job is assigned a quantum, its time in B is bounded, so the scheduler need only control entries to B (cf. Figure 6-3).

A job (of size j) may exit the balance set B for one of three reasons:

1. Its quantum expired, in which case it is entered at the end of the j^{th} queue.
2. It disabled, in which case it enters the disabled set.
3. It quit.

In general, a job will fluctuate in size during execution. Thus, if it is of size k at entry to B, it may be of size $j \neq k$ upon exit from B. We assume a condition of statistical equilibrium, so that, on the average, a job of size k entering B implies that, within q_k, <u>some</u> job of size k will exit B.

The arrival rate to the k^{th} queue is

(7.2.3) $$a_k = a\, f_s(k)$$

and, from eq. 7.2.2,

(7.2.4) $$\sum_{k\varepsilon Q} a_k = \sum_{k\varepsilon Q} a\, f_s(k) = a$$

We may assume that $\frac{1}{a_k}$ is the mean of an exponential distribution, because $\frac{1}{a}$ is the mean of an exponential distribution and jobs are statistically independent. The service rate for the k^{th} queue is b_k, and $\frac{1}{b_k}$ is the average service time for jobs in the k^{th} queue. We assume that $\frac{1}{b_k}$ is the mean of an exponential distribution.

The analysis of Section 7.1 can be used to estimate the behavior of each queue when each receives service independently of the others and one job at a time is serviced from each. In reality, more than one job from each queue may be in service, in which case the analysis of Section 7.1 must be interpreted as the worst-case behavior.

Theorem 7.3. Suppose B acts as a single server to each of the s_o queues in Figure 7-2. Let a_k be the arrival rate to the k^{th} queue, b_k be the service rate of the k^{th} queue, and let $r_k = a_k/b_k$. Then the expected processor and memory demand of the balance set B are:

$$\bar{p}_B = \sum_{k\varepsilon Q} r_k$$

(7.2.5)

$$\bar{m}_B = \sum_{k\varepsilon Q} k\, r_k$$

Proof: Using the result of Theorem 7.1, let

$$\pi_{ok} = \Pr[k^{th} \text{ subsystem is empty}] = 1 - \frac{a_k}{b_k} = 1 - r_k$$

where the k^{th} subsystem comprises the k^{th} queue and the single job from it in service. Define the random variable γ_k:

$$\gamma_k = \begin{cases} 1 & \text{if } k^{th} \text{subsystem non-empty: } \Pr[\gamma_k=1] = 1-\pi_{ok} \\ 0 & \text{otherwise} \qquad\qquad\qquad\quad \Pr[\gamma_k=0] = \pi_{ok} \end{cases}$$

Thus,

$$\Pr[\gamma_k=1] = r_k$$
$$\Pr[\gamma_k=0] = 1 - r_k$$

Then the random variable p_B of processor demand is

$$p_B = \sum_{k\varepsilon Q} \gamma_k$$

and the random variable m_B of memory demand is

$$m_B = \sum_{k\varepsilon Q} k\, \gamma_k$$

The expectations are:

$$\bar{p}_B = \sum_{k\varepsilon Q} \overline{\gamma_k} = \sum_{k\varepsilon Q} Pr[\gamma_k=1] = \sum_{k\varepsilon Q} r_k$$

$$\bar{m}_B = \sum_{k\varepsilon Q} k\,\overline{\gamma_k} = \sum_{k\varepsilon Q} k\,Pr[\gamma_k=1] = \sum_{k\varepsilon Q} k\,r_k$$

QED.

There is an interesting special case, in which the running time of a job of size k is inversely proportional to the probability $f_s(k)$:

(7.2.6) $$(\text{mean running time})_k = \frac{1}{b_k} = \frac{1}{b\,f_s(k)}$$

for some constant b. This behavior may in fact occur in some real situations. For example, the quantum q_k could be chosen to be:

(7.2.7) $$q_k = \frac{1}{b\,f_s(k)}$$

In this case,

(7.2.8) $$r_k = \frac{a_k}{b_k} = \frac{a\,f_s(k)}{b\,f_s(k)} = \frac{a}{b} = r$$

That is, $r_k=r$ is constant for all the queues.

Theorem 7.4. Suppose the conditions of Theorem 7.3 and eq. 7.2.7 hold. Then the expected processor and memory demands of the balance set are:

(7.2.9) $$\bar{p}_B = s_o r$$
$$\bar{m}_B = \frac{s_o(s_o+1)}{2}\,r$$

Furthermore, when demand is not too high, that is $r \ll 1$ (the queues are sparsely populated), then

$$\bar{m}_B \ll \frac{s_o^2}{2} \tag{7.2.10}$$

Proof: Eqs. 7.2.9 follow directly from eqs. 7.2.5 with $r_k = r$. If $r \ll 1$, then $rs_o \ll s_o$, and we have

$$\bar{m}_B = \frac{s_o(s_o+1)}{2} r = \frac{(s_o r)(s_o+1)}{2} \ll \frac{s_o^2}{2}$$

QED.

It is interesting to note that, when s_o is large, the distribution $f_s(i)$ may be approximated by a continuous density function $f_s(u)$ if we regard the range $[1, s_o]$ of job sizes as being continuous. In this case we may regard the set of queues, Q, as being a continuum of queues, and use the notation Q(u) to denote the queue into which jobs of size u are arriving. The arrival rate to Q(u) is:

$$a_u = a\, f_s(u) \qquad u \varepsilon [1, s_o] \tag{7.2.11}$$

Let b_u denote the service rate of size u jobs, and then

$$r_u = \frac{a_u}{b_u} \tag{7.2.12}$$

By analogy with eqs. 7.2.5,

$$\begin{aligned} \bar{p}_B &= \int_o^{s_o} r_u \, du \\ \bar{m}_B &= \int_o^{s_o} u \, r_u \, du \end{aligned} \tag{7.2.13}$$

Again, if $r_u = r$ is constant, we obtain the same results as eqs. 7.2.9.

7.2.2. The Logarithmic Queue

When the size s_o of the largest job is large, it may become impractical to implement a large network of queues, such as that of Figure 7-2. Indeed, when demand is not too high, the probability that queue k is non-empty is small; such a queue structure would comprise a mostly-empty set of queues.

A general approach to the problem of reducing the number of empty queues is to establish classes of comparable-size jobs, and to sort jobs entering the standby set into a system of FCFS queues, one for each class. Suppose the number of classes is chosen to be K. Then we must choose K size-intervals (s_{k-1},s_k) in order to define the classes S_k:

$$S_k = \left\{ s \;\middle|\; \begin{array}{l} s \text{ is the size of some job} \\ \text{in the interval } (s_{k-1},s_k) \end{array} \right\} \tag{7.2.14}$$

Figure 7-2 may be regarded as s_o classes with $S_k = \{k\}$, and if we choose $K<s_o$, it is not hard to see that the total expected balance set demands $(\bar{p}_B,\bar{m}_B)$ will be smaller (under the conditions of Theorem 7.3), because more work will be allowed to pile up in the (smaller number of) queues.

One method for choosing the boundaries of the classes S_k is to make the arrival rate into each class be the same:

$$a_k = \frac{a}{K} = \sum_{i \varepsilon S_k} a\, f_s(i) \tag{7.2.15}$$

where $f_s(i)$ is the probability Pr[s=i]. A much more interesting method of sorting, the logarithmic queue, has particularly useful properties.

The structure of the logarithmic queue is shown in Figure 7-3. Jobs are sorted by size into one of $[\log_2 s_o]$ FCFS queues (here,

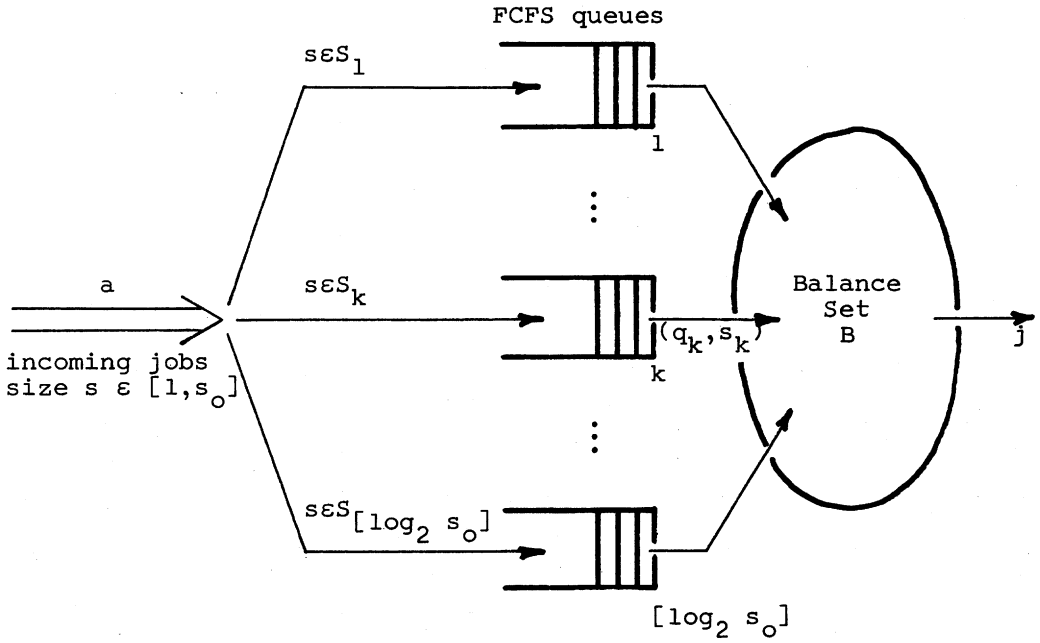


Figure 7-3. The logarithmic queue.

the notation [x] means the greatest integer $i \leq x$). The classes are defined to be

$$S_k = \{s \mid s \in (2^k, 2^{k+1})\} \qquad k \in Q \tag{7.2.16}$$

where $Q = \{1,\ldots,k,\ldots,[\log_2 s_o]\}$ is the set of queues. When a job of size s enters the standby set, it is placed at the end of queue $k = [\log_2 s]$.

With the k^{th} queue is associated a pair (q_k, s_k), q_k being the time quantum and s_k being the typical size of a queue k job.

The probability distribution $f_{s_k}(i)$ for jobs in class S_k is

$$f_{s_k}(i) = \begin{cases} \dfrac{f_s(i)}{\sum_{j \in S_k} f_s(j)} & \text{if } i \in S_k \\ 0 & \text{otherwise} \end{cases} \tag{7.2.17}$$

The average job size in class S_k is:

$$\bar{s}_k = \sum_{i \in S_k} i \, f_{s_k}(i) \tag{7.2.18}$$

and we may regard $\bar{s}_k$ as being typical of the jobs in class S_k.

The arrival rate to the k^{th} class is:

$$a_k = \sum_{i \in S_k} a \, f_s(i) \qquad \text{and} \qquad \sum_{k \in Q} a_k = a \tag{7.2.19}$$

Again, $\frac{1}{a_k}$ is assumed to be the mean of an exponential distribution. The service rate for the k^{th} class is:

$$b_k = \sum_{i \in S_k} b(i) \, f_{s_k}(i) \tag{7.2.20}$$

which is the average over S_k of the rate b(i) of each job size

i in S_k. Again we make the approximation that $\frac{1}{b_k}$ is the mean of an exponential distribution, so that we may use Theorems 7.1 and 7.2 to provide upper bounds on queue lengths and waiting times.

Theorem 7.5. Suppose the logarithmic queue structure described above is used, and that one job from each class is in service. Then the processor and memory demands of the balance set are bounded by:

$$p_B \leq [\log_2 s_o] \qquad \text{(single-process computations)}$$

$$m_B < 2s_o \tag{7.2.21}$$

where s_o is the size of the largest job.

Proof: If one single-process computation at a time is in B from each class, then at most $[\log_2 s_o]$ processes can be demanding a processor. In class S_k the largest job is of size $(2^{k+1}-1)$; then,

$$m_B \leq \sum_{k=1}^{\log_2 s_o} (2^{k+1}-1) < \sum_{k=1}^{\log_2 s_o} 2^{k+1} = 4 \sum_{k=0}^{(\log_2 s_o)-2} 2^k$$

or,

$$m_B < 4\left(2^{(\log_2 s_o)-2} - 1\right) < 2s_o$$

QED.

Thus, a memory of size $2s_o$, a logarithmic queue, and $[\log_2 s_o]$ processors are sufficient to guarantee service to one job from each class.

The advantages of the logarithmic queue are:

1. Fairness. One job from each class S_k is guaranteed service, and jobs from each class are serviced in order of arrival.
2. Ability to scale. The boundaries of the classes S_k are invariant to s_o, except for the upper boundary of class S_K, $K = [\log_2 s_o]$.
3. Small number of classes. Unless s_o is small, it is true that $[\log_2 s_o] << s_o$.
4. Small processor and memory requirements. From Theorem 7.5, no more than $2s_o$ pages of main memory are needed, and no more than $[\log_2 s_o]$ processors are needed, to accommodate the balance set B.
5. Flexibility. Suppose an imbalance of size s appears in the balance set memory demand, and that queue j is the queue in which size s jobs reside. If the scheduler finds queue j empty, it may still satisfy the imbalance with 2 jobs from queue (j-1), or 4 jobs from queue (j-2), etc.

7.3. Mathematical Programming Problem, One-Dimensional Case

We shall formulate mathematical programming problems whose solutions are balance policies.

We are require three things of balance policies: maintenance of balance, fairness, and the ability to satisfy other objectives such as minimum response time. Maintenance of balance is achieved by the constraints in the problems, fairness is achieved by making the mathematical programming problem operate in conjunction with queue structures of the types discussed in Section 7.2, and other objectives may be expressed as objective functions in the programming problems. We leave the particular objective function unspecified, the final choice being up to the policy designer.

In the remainder of this section we formulate the problem, review alternatives for the objective function, prove a theorem that constrains the choice of quanta, present a solution in the case of memory balance with minimum-response-time objective, and finally discuss briefly why the formulations cannot lead to completely optimum solutions.

7.3.1. The Problem

A decision point is a real time instant at which the scheduler is called on the rebalance the system. Suppose that the balance set demand is (p_B, m_B) at a decision point. Define the imbalance in B to be:

$$(\delta_p^B, \delta_m^B) = (\alpha, ß) - (p_B, m_B) = (\alpha - p_B, ß - m_B) \tag{7.3.1}$$

We assume that the scheduler is called on to admit jobs to B, never to remove jobs from B. On the one hand, since a job's quantum bounds its time in B, the demand (p_B, m_B) must eventually fall below (α, β). On the other hand, the parameters α and β are chosen to leave $(1-\alpha)N$ processors and $(1-\beta)M$ memory pages available for unanticipated expansions.

When do the decision points occur? This is basically a decision to be made by the policy designer. Possibilities are: decision points occur at regular, clocked intervals; they occur whenever a job exits the balance set B; or they occur whenever the imbalance exceeds some threshold.

Define the following parameters:

Q = $\{1,2,\ldots,K\}$ is the set of job-class indices.

n_k = number of jobs from class S_k selected by the scheduler to enter B.

n_k^B = number of jobs from class S_k already in B.

s_k = typical size of job in class S_k.

q_k = quantum assigned to jobs in class S_k.

α, β = balance parameters.

N, M = number of processors, number of main memory pages.

A = standard interval used to define processor demand in the single-process case (Section 6.2.2).

η = minimum tolerable duty factor for each computation.

Problem definition. Find integers $\{n_k\}_{k\varepsilon Q}$ such that the objective

$$F(n_1,\ldots,n_K)$$

is extremized, and the constraints

$$\{n_k + n_k^B \geq 1\}_{k\varepsilon Q} \tag{7.3.2}$$

$$\sum_{k\varepsilon Q} n_k s_k \leq M \delta_m^B \tag{7.3.3}$$

$$\sum_{k\varepsilon Q} n_k q_k \leq \frac{N A}{\eta} \delta_p^B \tag{7.3.4}$$

are satisfied.

The choice of objective function $F(n_1,\ldots,n_K)$ is discussed below. The constraint of eq. 7.3.2 means that at least one job from each class shall be in service. The constraint of eq. 7.3.3 asserts that the total memory requirement of jobs admitted to B shall not exceed the imbalance of $M\delta_m^B$ pages of memory. The constraint of eq. 7.3.4 asserts that the total processor requirement of jobs admitted to B shall not exceed the imbalance of $\frac{N A}{\eta}\delta_p^B$ processors. We have divided by the duty factor η because if each job has minimum duty factor η, then N processors may appear as $\frac{N}{\eta}$ processors; see Section 7.3.3.

7.3.2. The Objective Function

The objective function $F(n_1,\ldots,n_K)$ which is to be extremized (i.e., maximized or minimized) is to be specified by the policy designer. Some possibilities are, in order of complexity:

1. Minimize total waiting time. At a decision point, let N_k denote the number of jobs in the k^{th} queue. Then the wait of the job at the end of the k^{th} queue (before entering service) is $(N_k-n_k)q_k$. The objective becomes

$$\text{minimize } F(n_1,\ldots,n_K) = \sum_{k\varepsilon Q} (N_k-n_k)q_k$$
$$= \sum_{k\varepsilon Q} N_k q_k - \sum_{k\varepsilon Q} n_k q_k$$

but $N_k q_k$ is a constant at a decision point, so we have the simpler objective

(7.3.5) $$\text{maximize } \sum_{k\varepsilon Q} n_k\ q_k$$

We shall use eq. 7.3.5 as the expression of a minimum response time objective.

2. Minimize weighted sum of waiting times. Let $c_1,\ldots,c_K$ be a set of weights (relative importances) of the waiting times in each queue. Then (by analogy with eq. 7.3.5) the objective becomes

(7.3.6) $$\text{maximize } \sum_{k\varepsilon Q} c_k\ n_k\ q_k$$

3. Minimize weighted sum of functions of waiting times. Let $g_1,\ldots,g_K$ be a set of (cost) functions associated with the wait of the job at the end of each queue, and $c_1,\ldots,c_K$ are weights. Then the objective is

(7.3.7) $$\text{minimize } \sum_{k\varepsilon Q} c_k\ g_k(N_k-n_k)$$

These alternatives are meant only to illustrate possibilities, not to exhaust them.

7.3.3. Choice of Quanta

Efficiency requirements place an important constraint on the value of the quanta that may be chosen.

Theorem 7.6. Suppose η_o is given, where $0 \leq \eta_o \leq 1$, and we want the duty factor η to satisfy $\eta \geq \eta_o$ for all jobs, regardless of size. Then the quantum must be at least linearly proportional to the job size. Furthermore, not every value of η_o in the interval [0,1] is attainable for a given choice of the working set parameter τ.

Proof: Let s_k be the size of jobs in the k^{th} class, and q_k be their quantum. Let $\lambda(\tau)$ be the missing-page probability (Sections 4.2 and 5.4). In a virtual time interval of length q_k, the process encounters $\lambda(\tau)q_k$ page waits. In addition, at the start of the quantum q_k, the working set must be demand-paged into main memory (assuming it is not already there), requiring an additional s_k pages waits. Therefore the duty factor across the quantum must satisfy:

$$\eta = \frac{q_k}{q_k + \lambda(\tau)q_k T + s_k T} \geq \eta_o \qquad (7.3.8)$$

Solving eq. 7.3.8 for q_k we find

$$q_k \geq \frac{s_k \eta_o T}{1 - \eta_o - \eta_o \lambda(\tau) T}$$

For the given η_o, in order that q_k be finite, we must have

$$1 - \eta_o - \eta_o \lambda(\tau) T > 0 \qquad (7.3.9)$$

(For example, $\eta_o = 0.5$ requires $\lambda(\tau)T > 1$, and we must have

$\lambda(\tau)T>>1$ if q_k is to be reasonably small). Once η_o and τ have been fixed, the quantity

$$C_o = \frac{\eta_o T}{1 - \eta_o - \eta_o \lambda(\tau) T}$$

is fixed, and we have

$$q_k \geq C_o s_k$$

which was to be shown. In other words, if $q_k < C_o s_k$, the actual value of η (eq. 7.3.8) cannot satisfy $\eta \geq \eta_o$.

To show that not every value of η_o in the interval [0,1] is attainable for a given choice of τ, let τ be given and solve eq. 7.3.9 for η_o:

$$\eta_o < \frac{1}{1 + \lambda(\tau) T}$$

Thus, η_o is upper bounded. Compare with the result of Theorem 4.6, which gives this expression as the steady state duty factor when quantum starts and expirations are ignored.

QED.

Theorem 7.6 tells us that if we wish to achieve a certain level of processing efficiency, we must be willing to associate larger quanta with larger jobs.

7.3.4. Solution to the Memory Balance Problem

The most important one-dimensional case is the memory-balance case, because it finds application immediately in contemporary computer systems. In this case we place much emphasis on balancing memory (to avoid thrashing) and little emphasis on balancing processor. If the objective function is to minimize response time, there is a rather elegant algorithm to find the solution to the mathematical programming problem.

The linear programming problem presented in Section 7.3.1 very much resembles a classic problem, the knapsack problem [D0]. We are given a collection of objects, each having a certain weight and a certain value, and we are to pack them into a knapsack such that a given weight limit is not exceeded and the total value of objects packed is maximum. The solution to this problem gives insight into the nature of the solution to the memory-balance problem. Formally stated, the knapsack problem is:

The Knapsack Problem. Let Q be a class of object types, N_k be the number of objects of type k, w_k be the weight of a type k object, and v_k be the value of a type k object. We are to find integers $\{n_k\}_{k\varepsilon Q}$ such that the objective

$$\text{maximize} \sum_{k\varepsilon Q} n_k \, v_k$$

is achieved and the constraints

$$\{0 \leq n_k \leq N_k\}_{k\varepsilon Q}$$

$$\sum_{k\varepsilon Q} n_k \, w_k \leq W$$

are satisfied, where W is a given positive number.

Theorem 7.7. Suppose in the knapsack problem we have ordered the elements of Q such that

$$\frac{v_1}{w_1} > \frac{v_2}{w_2} > \ldots > \frac{v_k}{w_k} > \ldots \qquad k\varepsilon Q$$

and suppose no two classes have the same $\frac{v_k}{w_k}$ ratio. Then the optimum solution is:

for k=1,2,3,... do:

choose the largest n_k such that $0 \leq n_k \leq N_k$ and

$$\sum_{k\varepsilon Q} n_k w_k \leq W$$

Proof: The details can be found in Dantzig [D0], but the idea is very simple. The ratios $\frac{v_k}{w_k}$ may be interpreted as the value per pound weight of each object. The algorithm merely attempts to reach the weight limit W by packing in all the objects with the highest values per unit weight.

QED.

In the general case, we would have to assume that the ratios satisfy

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_k}{w_k} \geq \ldots \qquad k\varepsilon Q$$

instead of the strictly decreasing situation given in Theorem 7.7. This complicates the algorithm. Since we are about to apply it to the memory balance problem, which will satisfy a constraint similar to that in Theorem 7.7, we shall not concern ourselves further with refinements of the algorithm. Additional solution methods are found in Dantzig [D0].

The Memory Balance Problem. Let Q be a set of queues, as discussed earlier. Let s_k be the typical size of jobs of type k, q_k be the quantum associated with type k jobs, and N_k be the number in the k^{th} queue at a decision point. We are to find integers $\{n_k\}_{k\varepsilon Q}$ such that the objective

$$\text{maximize} \sum_{k\varepsilon Q} n_k \, q_k$$

is achieved and these constraints are satisfied:

$$\{0 \leq n_k \leq N_k\}_{k\varepsilon Q}$$

$$\{n_k + n_k^B \geq 1\}_{k\varepsilon Q}$$

$$\sum_{k\varepsilon Q} n_k \, s_k \leq M \, \delta_m^B$$

where M is the main memory size, n_k^B is the number of type k jobs in the balance set B, s_k is the size of a type k job, and δ_m^B is the memory imbalance of B.

The objective function, which minimizes response time, is the same as eq. 7.3.5. There is one constraint more than in the knapsack problem, namely that the balance set B shall contain at least one object of each class. There is no explicit processor balance condition because we assume a sufficiency of processors. We shall see in Chapter 8 that we can properly match processor and memory resources to the given program mix; thus we may suppose there are enough processors as long as we abide by the memory constraint and do not change the program mix.

The solution to the memory balance problem is given in the next theorem.

Theorem 7.8. The solution to the memory balance problem is as follows. Let $Q = \{1,2,\ldots,K\}$ be the set of queue indices. Choose the quanta q_k to satisfy

$$\frac{q_K}{s_K} > \ldots > \frac{q_2}{s_2} > \frac{q_1}{s_1}$$

Let $R = \{k \varepsilon Q \mid n_k^B = 0\} = \{k_1, k_2, \ldots, k_r \mid k_1 > k_2 > \ldots > k_r\}$. Then:

1. for $j=1,2,\ldots,r$ do:

$$\text{if } \sum_{i=1}^{j} s_{k_i} > M\delta_m^B \text{ then goto step 3 else } n_{k_j} = 1;$$

2. for $k=K,\ldots,2,1$ do: choose the largest n_k such that

$$0 \leq n_k \leq N_k$$

$$\sum_{k \varepsilon Q} n_k s_k \leq M\delta_m^B - \sum_{j \varepsilon R} n_j s_j$$

3. done.

Proof: Follows at once by analogy with Theorem 7.7.

QED.

In words, Theorem 7.8 says: first satisfy the one-job-from-a-class constraint, then keep on admitting the largest jobs possible until memory is full. If step 1 fails to admit jobs to the balance set, step 2 is bypassed; thus, resources are reserved for these jobs, in readiness for the next decision point.

7.3.5. On the Optimality of the Solutions

At each decision point the scheduler finds a solution to the mathematical programming problem; but this solution is optimum only with respect to the decision point at which it was made.

Put another way, if the scheduler had a complete listing of balance set program sizes, together with their completion times, it might very well want to make a decision different from that which satisfies the mathematical programming problem. A decision which appears to satisfy the objective function at time t_1 may turn out to be poorer across an interval (t_1,t_2) than a decision which appears not to satisfy the objective function at time t_1.

Thus, all we can claim about these mathematical programming formulations of balance policies is that they produce solutions which are optimum (with respect the the given objective function) across short time intervals, but not necessarily across long time intervals.

We do not feel that this is a serious difficulty. The main function of these policies is to keep the computer system balanced under the given criteria of fairness. The objective functions incorporated into the mathematical programming problems are there to accomplish ancillary objectives, namely those beyond balance and fairness. Thus, it is not of major importance that the policy is only locally optimum with respect to the objective function.

Of far greater import is: it is possible, under fair balance policies, to establish reasonable policies with respect to criteria such as minimum response time, without requiring exorbitant amounts of processor and memory resources.

7.4. Mathematical Programming Problem, Two-Dimensional Case

We briefly generalize the ideas of the previous section, formulating the mathematical programming problem for the two-dimensional case, in which it is important to balance both processor and memory equally well. The formulation is very general and is presented in the continuous case. Each job in the standby set is now a multiprocess computation.

Consider again the demand space $\underset{\sim}{V}$, illustrated in Figure 7-4, where the region $\underset{\sim}{Q}$ in the unit square is regarded as being a continuous two-dimensional queue. A demand is a point (u,v); demands may appear only in the region $\underset{\sim}{Q}$. The demand density function $f_{pm}(u,v)$ is two-dimensional; that is, the probability a demand (p,m) falls in a differential region of area (du dv) at the point (u,v) is given by $(f_{pm}(u,v)\ du\ dv)$, and

$$\iint_{\underset{\sim}{Q}} f_{pm}(u,v)\ du\ dv = 1 \qquad (7.4.1)$$

Again, a is the (exponential) arrival rate of demands into the standby set. The rate to the queue at the point (u,v) is

$$a(u,v) = a\ f_{pm}(u,v) \qquad (7.4.2)$$

The rate at which jobs leave the queue at the point (u,v) is

$$b(u,v) \qquad (7.4.3)$$

Therefore

$$\text{Pr[queue (u,v) non-empty]} = r(u,v) = \frac{a(u,v)}{b(u,v)} \qquad (7.4.4)$$

following Theorem 7.1. Assuming that each queue is treated independently, under a FCFS policy, the expected processor demand

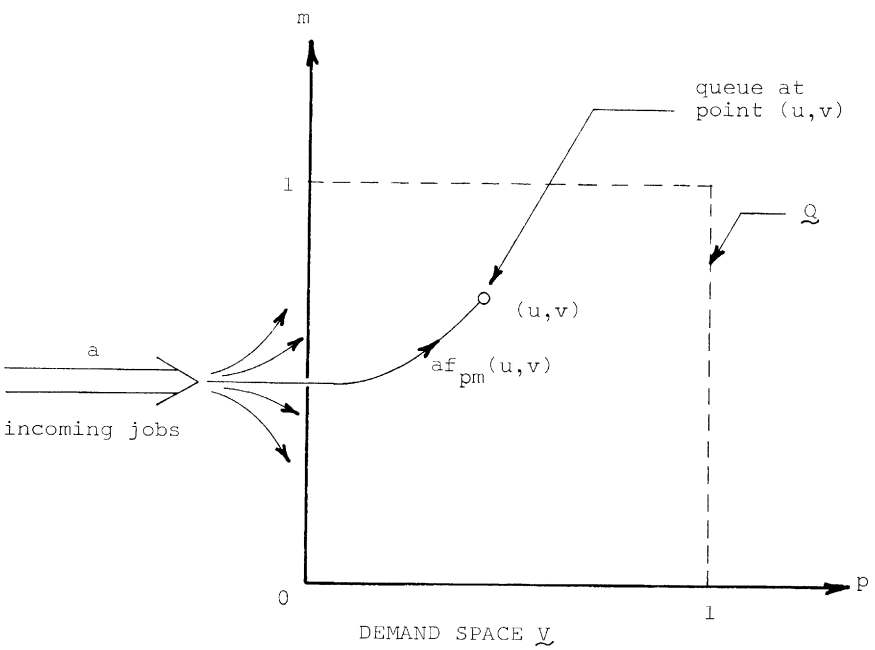


Figure 7-4. Demand space as a continuous two-dimensional queue.

of the balance set is

(7.4.5) $$\bar{p}_B = \iint_{\underset{\sim}{Q}} u\, r(u,v)\, du\, dv$$

and the expected memory demand of the balance set is

(7.4.6) $$\bar{m}_B = \iint_{\underset{\sim}{Q}} v\, r(u,v)\, du\, dv$$

These equations are obtained by noting the expected contribution to processor demand from the queue at (u,v) is

(7.4.7) $$u\left(\Pr[\text{queue } (u,v) \text{ non-empty}]\right) = u\, r(u,v)$$

and the expected contribution to memory demand from the queue (u,v) is

(7.4.8) $$v\left(\Pr[\text{queue } (u,v) \text{ non-empty}]\right) = v\, r(u,v)$$

In the special case that $r(u,v)=r$ is constant everywhere in $\underset{\sim}{Q}$:

(7.4.9) $$\bar{p}_B = \bar{m}_B = \frac{r}{2}$$

To set up the mathematical programming problem, we define the following quantities:

- $n(u,v)$ = number of jobs to be chosen from queue (u,v) at a decision point to enter the balance set B. Note that $n(u,v) \geq 0$ is a continuous distribution.
- $q(u,v)$ = quantum to be allocated to a job from queue (u,v). We assume q(u,v) depends only on (u,v). Since a job at (u,v) is a multiprocess computation, q(u,v) represents the <u>total</u> virtual time allocated, in a pool, to all the processes in the computation.
- u = processor demand at queue (u,v).
- v = memory demand at queue (u,v).
- $w(u,v)$ = waiting time of job at end of queue (u,v) until it enters service.

$g(u,v,x)$ = cost function associated with queue (u,v) when the waiting time there is x.

(δ_p^B, δ_m^B) = $(\alpha,\beta)-(p_B,m_B)$ is the degree of imbalance.

The Problem. Let (p_B,m_B) be the balance set demand at a decision point. We are to find a distribution $n(u,v)$ of jobs to enter B, such that the objective

$$\iint_{\underset{\sim}{Q}} n(u,v)\, g(u,v,w(u,v))\, du\, dv$$

is minimized, and the constraints

$$\iint_{\underset{\sim}{Q}} n(u,v)\, u\, du\, dv \leq N\, \delta_p^B$$

$$\iint_{\underset{\sim}{Q}} n(u,v)\, v\, du\, dv \leq M\, \delta_m^B$$

are satisfied.

We shall not attempt to discuss implementation issues here as we have done for the one-dimensional programming problem in Section 7.3.4. The solution $n(u,v)$ to this problem is complicated by the path effect, discussed in Section 6.5.1. We leave this as an area of future research.

7.5. Summary

The major result of this chapter is: the balance policy to be used the scheduler can be expressed as the solution to a mathematical programming problem.

The solutions produced by these formulations are optimum with respect to the objective function across short time intervals but not necessarily across long time intervals. These policies are supposed primarily to be equitable balance policies, secondarily to insure reasonable policies toward criteria such as minimum response time; thus, these policies meet the objectives of this thesis work.

We showed that there exists a simple, elegant algorithm, which finds the optimum set of jobs to admit to the balance set at a decision point, which is to be used in the memory balance case together with a minimum response time objective function. This algorithm is applicable in contemporary computer systems, when it is important to prevent thrashing. It is based on an analogy with the knapsack problem, a classic linear programming problem.

CHAPTER 8

Applications to Computer System Organization

8.0. Introduction

One aspect of the study of the resource allocation problem has been to set up behavior models for computations in order to provide a framework within which we can understand misunderstood problems. An equally important aspect of the study is to examine how programmers, system designers, and the computer system itself might all cooperate in allocating resources.

We shall discuss three seemingly disparate aspects of computer system organization. The first, the equipment configuration, is the relationship among the program mix, the amount of processor, and the amount of memory. The second, equipment pooling, is effecting large processor-memory capacity by sharing equipment at the finest hardware level. The third, multilevel memories, is showing how to make better use of memory resources. The relation among these three aspects is: each is concerned with matching the equipment to the work load.

8.1. Toward Better Programming and System Design

There is every reason to believe that programmers can, by careful programming, create programs that run with small, compact working sets. They can do this, for example, by designing algorithms to work locally on information, and by employing data structures which induce highly local reference patterns. Programmers who cooperate in this way will be rewarded, for their working sets will be smaller, memory-usage costs lower, and running times shorter. Thus, a first guiding principle, for programmers, is to design programs to have small working sets.

The remaining guiding principles, for system designers, are applications of programming generality (Section 1.2) and of our results here.

Perhaps the single most degrading factor in contemporary computer systems is the inability to manipulate small quantities of information easily. Ideally, the unit of information storage and transfer, universally used throughout the entire computer system from the highest level of memory to the lowest, should be the word. This is simply not feasible in contemporary systems on account of the high cost of accessing an item in auxiliary storage. The commonly-used compromise is that of paging: each page comprises a block of words, the page size being chosen to represent a compromise among wasted memory, complexity of record-keeping (i.e., page tables, memory usage map), and cost of transferring a page into main memory. And yet, the traverse-time cost is still so high that paged memory systems have been beseiged with poor performance. Thus, a second guiding principle for the system designer must be to make it convenient to manipulate small

quantities of information [let us aim for the word]. To do this, he must take recourse to parallelism in the data channels, and in the addressing and accessing mechanisms.

Our behavior models have verified quantitatively the intuitively obvious fact that sharing of equipment becomes more and more successful when there are more and more participants. This generates a need for a great number of processors and for a great deal of main memory (much of which is wasted in contemporary systems because the large traverse time induces so many protracted page waits). On the one hand, multiprogramming has made it possible to effectively share the memory resource among many computations. On the other hand, however, it is not yet possible to achieve anywhere near complete utilization of processors. Instead, a processor is dedicated to a single process, only one instruction at a time being executed, and most of the equipment (adders, multipliers, etc.) in a processor is idle[1]. If, instead, the individual hardware components of several processors were placed in pools (adders, multipliers, etc.), it would be possible to overlap a great many operations. By making it accessible from pools on demand, the same equipment contained in one modern processor could be used to service simultaneously a surprisingly large number of processes. Thus, a third guiding principle for the system designer is to permit pooling of small hardware units.

In summary, the computer system designer must at the very least be guided by these principles: programming generality, small-working-set programs, ability to manipulate small quantities of information, and ability to pool small hardware units.

---

[1] Some processors, such as the CDC 6600 and the IBM 360/91, attempt to overlap operations by looking ahead a short distance in the instruction stream; but the monosequential nature of instruction streams makes it difficult to overlap more than a few operations.

8.2. The Equipment Configuration

By the program mix we mean the collection of possible computations. By the equipment configuration we mean the proper relative choices of the number N of processors and the number M of main memory pages to achieve static balance. We shall show that, once any two of {program-mix,M,N} are arbitrarily given, the third is determined.

The following is meant to indicate the kind of procedure that may be used to determine the equipment configuration; it is not meant to be the only possible approach.

We assume that all jobs are single-process computations, that they are statistically independent, and that their working sets do not overlap.

8.2.1. Choosing the Balance Parameters α and β

The statistical properties of the program mix are the working set size and the duty factor.

We assume that the working set size $\omega(t,\tau)$ is a stationary random process (cf. Sections 3.3 and 4.4), and we let τ be understood. Thus, we may write ω instead of $\omega(t,\tau)$. The mean $\bar{\omega}$ and the variance σ_ω^2 have already been derived in Theorems 4.4 and 4.5.

The duty factor η depends on the choice of working set parameter τ, the size ω of a job's working set, a job's quantum q, and the traverse time T, as follows. If a job is assigned a quantum q, it generates q information references. The steady state missing-page probability is $\lambda(\tau)$, so the job expects to encounter $q\lambda(\tau)$ page waits due to pages re-entering its working set. In addition, its working set must be demand-paged into

memory at the start of its quantum, requiring an additional ω page waits, one for each page. The total expected page wait time is $(q\lambda(\tau)+\omega)T$. The duty factor is

$$\eta = \frac{q}{q + (q\lambda(\tau)+\omega)T} = \frac{1}{1 + (\lambda(\tau)+\frac{\omega}{q})T} \tag{8.2.1}$$

Let

$$\gamma = \lambda(\tau) + \frac{\omega}{q} \tag{8.2.2}$$

so that

$$\eta = \frac{1}{1 + \gamma T} \tag{8.2.3}$$

For simplicity in the following discussion we assume that η is the same for all jobs (thus, $q=C_o\omega$ for some constant C_o; cf. Theorem 7.6).

We assume that, whenever a job in the balance set is not in page wait, it is running. In order that this be a good assumption, there must be sufficient processor resources that the probability

$$\Pr\left[\begin{matrix}\text{no processor available when}\\ \text{a process exits page wait}\end{matrix}\right]$$

is arbitrarily small. We shall see shortly that this is the case. In this case, we may regard η as the probability that a process is running.

We now define two random variables: W is the total working set size of the balance set and P is the total processor requirement of the balance set. We suppose there are n jobs in the balance set. From our discussion in Chapter 7, if K is the number of standby set queues, then n must satisfy $n \geq K$.

Let ω_i be the working set size of the i^{th} job in the balance set. Then the total working set size W of the balance set is

$$W = \sum_{i=1}^{n} \omega_i \tag{8.2.4}$$

Since the jobs are statistically independent and identically distributed, we have

$$\begin{aligned} \bar{\omega}_i &= \bar{\omega} \\ \sigma^2_{\omega_i} &= \sigma^2_{\omega} \end{aligned} \qquad i = 1,2,\ldots,n \tag{8.2.5}$$

Then also

$$\begin{aligned} \bar{W} &= n\bar{\omega} \\ \sigma^2_W &= n\sigma^2_{\omega} \end{aligned} \tag{8.2.6}$$

Define the binary random variable

$$\pi_i = \begin{cases} 1 & \text{if the } i^{th} \text{ job is running} \\ 0 & \text{otherwise} \end{cases} \tag{8.2.7}$$

From the discussion above,

$$\begin{aligned} \Pr[\pi_i=1] &= \eta \\ \Pr[\pi_i=0] &= 1-\eta \end{aligned} \tag{8.2.8}$$

where η is the duty factor. Since the jobs are statistically independent and identically distributed,

$$\begin{aligned} \bar{\pi}_i &= \bar{\pi} = \eta \\ \sigma^2_{\pi_i} &= \overline{\pi^2} - \bar{\pi}^2 = \eta(1-\eta) \end{aligned} \qquad i = 1,2,\ldots,n \tag{8.2.9}$$

The total processor requirement P of the balance set is

$$P = \sum_{i=1}^{n} \pi_i \tag{8.2.10}$$

and

$$\begin{aligned} \bar{P} &= n\bar{\pi} = n\eta \\ \sigma_P^2 &= n\sigma_\pi^2 = n\eta(1-\eta) \end{aligned} \tag{8.2.11}$$

Now, let numbers ε_N and ε_M be given, with $0 \leq \varepsilon_N \leq 1$ and $0 \leq \varepsilon_M \leq 1$. These numbers ε_N and ε_M represent the allowable processor and memory overflow probabilities. That is, we want to choose M and N such that

$$\begin{aligned} \Pr[W > M] &\leq \varepsilon_M \\ \Pr[P > N] &\leq \varepsilon_N \end{aligned} \tag{8.2.12}$$

We must proceed carefully, because M and N are not independent. But before proceeding, we must indicate how Pr[W>M] and Pr[P>N] might be determined.

The Central Limit Theorem tells us that the sum of n identically distributed, statistically independent random variables becomes normally distributed for large n. We may therefore approximate the distributions of W and P by normal distributions (these approximations are surprisingly good, even for n as small as 10 or 20; see Feller [F2, Vol. 1, p. 168ff]). That is, we approximate $f_W(u)$ and $f_P(u)$ by

$$\begin{aligned} f_W(u) &= \frac{1}{\sqrt{2\pi}\,\sigma_W} \exp\left[-\frac{(u-\bar{W})^2}{2\,\sigma_W^2}\right] \\ f_P(u) &= \frac{1}{\sqrt{2\pi}\,\sigma_P} \exp\left[-\frac{(u-\bar{P})^2}{2\,\sigma_P^2}\right] \end{aligned} \tag{8.2.13}$$

and then

$$\Pr[W > M] = \int_M^\infty f_W(u)\,du$$

(8.2.14)

$$\Pr[P > N] = \int_N^\infty f_P(u)\,du$$

Therefore, given ε_M and ε_N we can find M and N (using standard tables for the normal distribution, such as [F2, Vol. 1, p. 167]) such that

$$\int_M^\infty f_W(u)\,du = \varepsilon_M$$

(8.2.15)

$$\int_N^\infty f_P(u)\,du = \varepsilon_N$$

and so relate M to ε_M and N to ε_N[1]. It is now a simple matter to choose M and N.

Let the memory size M be given. Then choose the largest n such that[2]

(8.2.16) $$\Pr[W > M] = \Pr[\sum_{i=1}^{n} \omega_i > M] \leq \varepsilon_M$$

Using this value of n, find the smallest N such that

(8.2.17) $$\Pr[P > N] = \Pr[\sum_{i=1}^{n} \pi_i > N] \leq \varepsilon_N$$

---

[1] The normal approximation is not the only way. For example, the more powerful Chernoff Bound [W2, p.67ff] shows that, given random variables $z_i \geq 0$ with common density function $f_z(u)$, there exists a decreasing function h(A) depending only on $f_z(u)$ such that

$$\Pr[\sum_{i=1}^{n} z_i \geq nA] \leq (h(A))^n \quad \text{with } 0 \leq h(A) \leq 1$$

[2] In Chapter 7, we required $n \geq K$, where K is the number of standby set queues. Here we assume that M and N are large enough so that the largest values of n satisfying eqs. 8.2.16 and 8.2.17 also satisfy $n \geq K$.

It should be clear from eqs. 8.2.16 and 8.2.17 that, once any one of the three quantities $\{n,M,N\}$ has been arbitrarily given, the other two are uniquely determined (all other things being equal). Thus, it makes no difference which of $\{n,M,N\}$ is chosen first.

To choose α and β, let $\{n,M,N\}$ be chosen as above, and set β such that

$$\beta M = \bar{W} = n\bar{\omega} \tag{8.2.18}$$

and set α such that

$$\alpha N = \bar{P} = n\bar{\pi} = n\eta \tag{8.2.19}$$

The procedure discussed above is a worst-case procedure, for the following reason. In a real computer system, the values of α and β so selected are lower bounds on the actual values that may be used without violating the probabilities ε_M and ε_N. That is, the values of α and β actually used may satisfy

$$\begin{aligned} \frac{n\eta}{N} &\leq \alpha \leq 1 \\ \frac{n\bar{\omega}}{M} &\leq \beta \leq 1 \end{aligned} \tag{8.2.20}$$

The reason for this is: the scheduler carefully regulates the membership of the balance set, dynamically maintaining W within close tolerance of βM and P within close tolerance of αN. The procedure just described takes no account of this additional certainty, that W is close to βM and P is close to αN. Thus, the actual variance of W is less than σ_W^2 (eq. 8.2.6) and the actual variance of P is less than σ_P^2 (eq. 8.2.11). There is more freedom to choose larger α and β.

8.2.2. How Much Resource Slack?

We refer to the reserve $(1-\beta)M$ pages of memory as the slack memory, and the reserve $(1-\alpha)N$ processors as the slack processor. We want to show that, as M and N are increased and ε_M and ε_N are held fixed, that the relative amounts of slack resources become negligible.

Theorem 8.1. Suppose ε_N and ε_M are given, α and β are determined according to the procedure above, and we let the number n of jobs in the balance set increase without bound (appropriately adjusting M and N to satisfy ε_M and ε_N). Then

$$\alpha \rightarrow 1$$
$$\beta \rightarrow 1$$

Proof: We show that $\beta \rightarrow 1$, since the proof for $\alpha \rightarrow 1$ is exactly the same. Since $0 \leq \beta \leq 1$, it is enough to show:

$$\frac{1-\beta}{\beta} \rightarrow 0$$

Refer to Figure 8-1, where we have plotted memory usage W, showing it to be normally distributed with mean $\beta M = n\bar{\omega}$ and standard deviation $\sigma_W = \sqrt{n}\sigma_\omega$. It is well known that, given ε_M, the probability $Pr[W>M]$ depends only on the distance between M and βM. That is, there exists a fixed constant $b>0$ such that

$$Pr[W>M] = Pr[\beta M + b\sigma_W > M]$$

Then, as $n \rightarrow \infty$

$$\frac{1-\beta}{\beta} = \frac{(1-\beta)M}{\beta M} = \frac{b\sigma_W}{n\,\bar{\omega}} = \frac{b\sigma_\omega}{\bar{\omega}}\frac{\sqrt{n}}{n} = \frac{b\sigma_\omega}{\bar{\omega}}\frac{1}{\sqrt{n}} \rightarrow 0$$

QED.

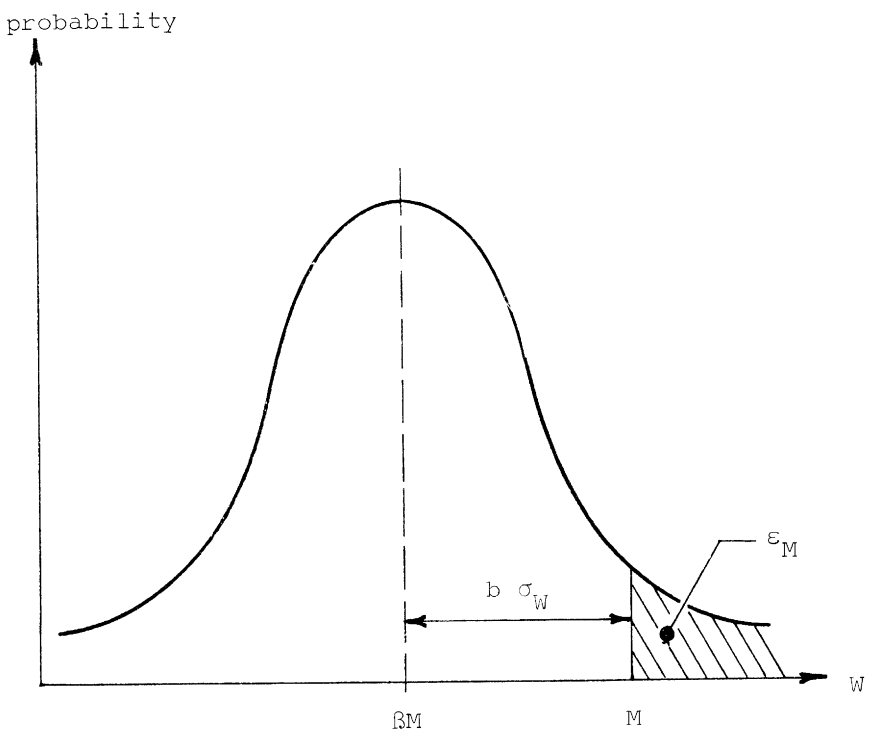


Figure 8-1. Memory usage.

8.2.3. Relations Among Processors, Memory, Traverse Time

There is an important three-way relationship among the number N of processors, the number M of main memory pages, and the traverse time T. If any two of these quantities are given, the third is determined.

Theorem 8.2. Define ßM to be the expected amount of memory to match αN processors. Then

(8.2.21) $$ßM = \alpha N\bar{\omega}(1+\gamma T)$$

where $\bar{\omega}$ is the expected working set size, T is the traverse time, and

(8.2.22) $$\gamma = \lambda(\tau) + \frac{\omega}{q} = \lambda(\tau) + \frac{1}{C_o}$$

has been discussed at eqs. 8.2.2 and 8.2.3.

Proof: Let {n,M,N} be chosen as discussed in Section 8.2.1. Then

$$\bar{W} = ßM = n\bar{\omega}$$

$$\bar{P} = \alpha N = n\eta = \frac{n}{1 + \gamma T}$$

where η is the duty factor, given by eq. 8.2.3. Eliminating n between these two equations, we have

$$n = \alpha N(1+\gamma T)$$

so that

$$ßM = n\bar{\omega} = \alpha N\bar{\omega}(1+\gamma T)$$

QED.

Since Theorem 8.2 depends on average value arguments, it is only an approximation to the actual behavior. Put another way, we may only regard the relation $\beta M=\alpha N\bar{\omega}(1+\gamma T)$ as stating a necessary, but not sufficient, condition on the hardware configuration. However, the discussion in Section 8.2.1 shows that we can regard ε_M and ε_N as confidence levels for this result.

The relation $\beta M=\alpha N\bar{\omega}(1+\gamma T)$ gives further insight into the causes of thrashing (Section 3.6). Recall that large values of T make the duty factor (and hence the attainable processing efficiency) very sensitive to small changes in the missing-page probability (here, represented by γ). In Figure 8-2 we have indicated the behavior of the processor-memory ratio:

$$R = \frac{\beta M}{\alpha N\bar{\omega}} = (1+\gamma T) \tag{8.2.23}$$

for T=1,10,100,1000,and 10000 vtu. It is clear that, when γ is small and T is large, the slope of R is quite steep. Small fluctuations of γ can result in wild fluctuations in R. Thus, if $R_d=\frac{\alpha}{\beta}\bar{\omega}(1+\gamma T)$ is regarded as representing the desired processor-memory ratio, and $R_a=\frac{M}{N}$ is regarded as representing the actual processor-memory ratio, then these fluctuations in γ cause R_d to be seriously mismatched to R_a.

In Figure 8-3 we have shown that the expected amount βM of memory grows linearly. Indeed, for large values of T,

$$\beta M \approx \alpha N\bar{\omega}T \tag{8.2.24}$$

Were we to reduce T by an order or two of magnitude, we could also reduce the main memory requirement by as much as an order or two of magnitude, without sacrificing efficiency.

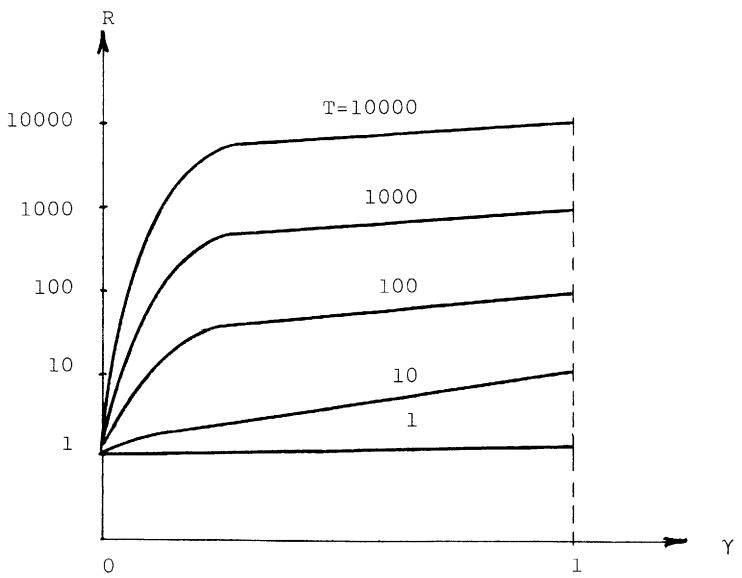


Figure 8-2. Desired processor-memory ratio.

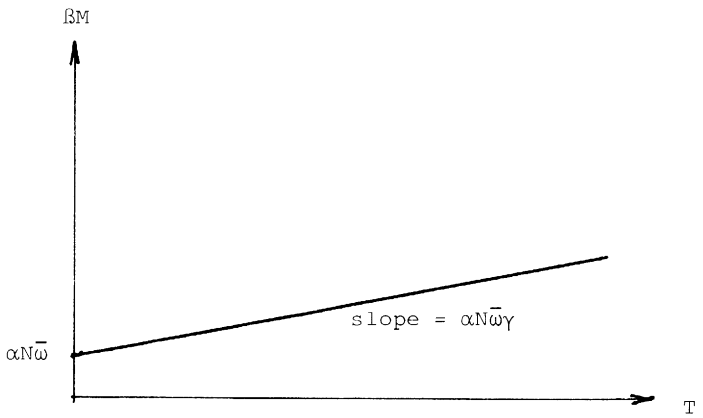


Figure 8-3. Relation among processors, memory, traverse time.

The relation $\beta M=\alpha N\bar{\omega}(1+\gamma T)$ shows that βM can increase if and only if αN increases, all other things being equal. Thus, if $\alpha_o N<\alpha N$ processors are available, then for some $\beta_o<\beta$, $\beta_o M=\alpha_o N\bar{\omega}(1+\gamma T)$, and $(\beta-\beta_o)M$ main memory pages stand idle (that is, they are in no working set). Similarly, if $\beta_o M<\beta M$ main memory pages are available, then for some $\alpha_o<\alpha$, $\beta_o M=\alpha_o N\bar{\omega}(1+\gamma T)$, and $(\alpha-\alpha_o)N$ processors stand idle. A shortage in one resource type inevitably results in a surplus of the other.

All our claims that large traverse times degrade performance and strain system resources can be substantiated. Fikes, et al. [F5] and Lauer [L1] report on their experience with the IBM 360/67 Time Sharing System at Carnegie University, in which they replaced the drum auxiliary store with large (bulk) core storage. For their system, the drum traverse time is about 10 times larger than the large core storage transfer time. They report that throughput was increased by about a factor of 10 when the large core replaced the drum. This supports our remarks concerning eq. 8.2.24. Since a considerable amount of main memory space was reserved for use as buffers for the drum, removing the drum made a large quantity of additional memory available.

Lauer [L1] points out that the equipment rental and maintenance costs were about 10 per cent higher after the large core storage was introduced. Since, however, the system capacity is effectively 10 times greater with large core storage than without it, the increased size of the market (user community) more than offsets the increased cost.

8.3. Pooling

Equipment pooling can remarkably enhance throughput. To verify this, we consider the conceptual experiment shown in Figure 8-4.

Theorem 8.3. Suppose n requestors with identically distributed demands seek to use a given type of resource. Compare two cases: first, each requestor is given a private supply of resource; second, each requestor draws on demand from a pooled supply of the resource. In order than the probability

$$\pi = \Pr[\text{given requestor fails to obtain required resource}]$$

be the same in both cases, at least $\sqrt{n}$ times as much resource is needed to provide private supplies as is needed to provide a pooled supply.

Proof: Each of the n requestors requests a random variable $y_i = y$ of the resource, independently of the others. For simplicity assume $\bar{y}=0$. Then

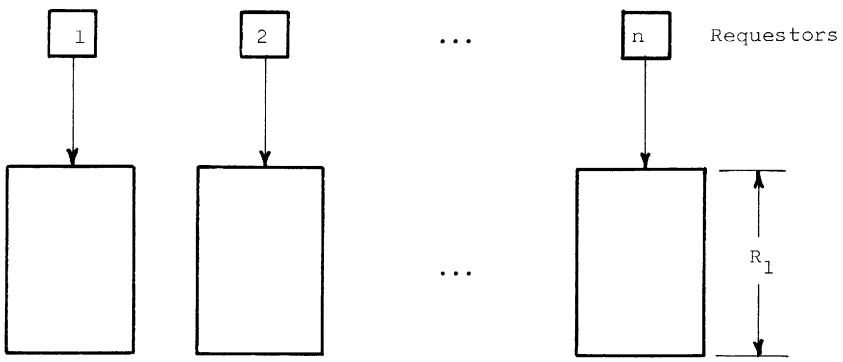
$$\sigma_y^2 = \overline{y^2} - \bar{y}^2 = \overline{y^2} = \int_{|y| \geq 0} u^2 f_y(u)\, du$$

but

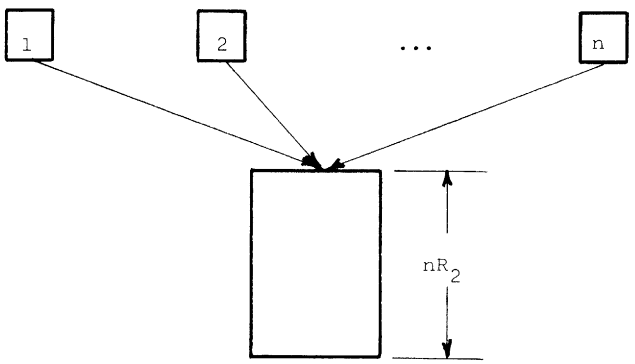
$$\int_{|y| \geq 0} u^2 f_y(u)\, du \geq \int_{|y| \geq R} u^2 f_y(u)\, du \geq R^2 \int_{|y| \geq R} f_y(u)\, du$$

thus,

(8.3.1) $$\sigma_y^2 \geq R^2 \Pr[\,|y| \geq R]$$



CASE 1. Private supplies of resource.



CASE 2. Pooled resource.

Figure 8-4. Pooled vs. Private resource supplies.

If

$$Y = \sum_{i=1}^{n} |y_i|$$

then

$$\sigma_Y^2 = n\sigma_y^2$$

and eq. 8.3.1 becomes

$$\Pr[Y \geq R] \leq \frac{n\sigma_y^2}{R^2} \tag{8.3.2}$$

Let $\varepsilon^2>0$ be given. In Case 1 (Figure 8-4), the probability π of the theorem statement becomes, using eq. 8.3.1,

$$\pi = \Pr[|y| \geq R_1] \leq \frac{\sigma_y^2}{R_1^2} = \varepsilon^2$$

or,

$$R_1 = \frac{\sigma_y}{\varepsilon}$$

and

$$\text{(total resource in Case 1)} = nR_1 = \frac{n\sigma_y}{\varepsilon}$$

In Case 2, suppose (n-1) requestors have made their requests, and then the n^{th} request arrives, his request being $y_n=u$. Then he fails to obtain his request just when

$$\sum_{i=1}^{n-1} |y_i| \geq nR_2 - |u|$$

Therefore the probability π of the theorem statement becomes

$$\pi = \int \left(\Pr\left[\sum_{i=1}^{n-1} |y_i| \geq nR_2 - |u| \right] \right) f_y(u)\, du$$

$$= \int \left(\Pr\left[\sum_{i=1}^{n-1} |y_i| + |u| \geq nR_2 \right] \right) f_y(u)\, du$$

$$= \Pr\left[\sum_{i=1}^{n-1} |y_i| + |y_n| \geq nR_2\right]$$

$$= \Pr\left[\sum_{i=1}^{n} |y_i| \geq nR_2\right] \leq \frac{n\,\sigma_y^2}{(nR_2)^2} = \varepsilon^2$$

from eq. 8.3.2. Then

$$R_2 = \frac{1}{\sqrt{n}} \frac{\sigma_y}{\varepsilon}$$

and

$$\text{(total resource in Case 2)} = nR_2 = \sqrt{n}\,\frac{\sigma_y}{\varepsilon}$$

Finally,

$$\frac{\text{(total resource in Case 1)}}{\text{(total resource in Case 2)}} = \frac{n\,\frac{\sigma_y}{\varepsilon}}{\sqrt{n}\,\frac{\sigma_y}{\varepsilon}} = \sqrt{n}$$

QED.

It is therefore quite clear that sharing and pooling can significantly increase the usable capacity of a given amount of equipment, especially when n is large.

It is one matter to realize that pooling at the finest level of detail is beneficial, but it is quite another matter to implement it. For pooling to operate without unnecessary loss of speed, it is necessary to dispense with a centrally clocked computer system and to rely wholly on asynchronous logic. Luconi [L2] has studied some of the rather delicate issues attending this problem.

Existing techniques make pooling of memory resources possible, but they have not yet made pooling of processing hardware possible at an equivalent level of detail -- many programs can reside in one memory unit, but only one process can use a processing unit at a time.

The problem of pooling the memory resources can be very effectively solved by paging. The smaller the page size, the better the pooling. Unfortunately the long traverse times that predominate in contemporary computer systems make it just as expensive to move a small page as a large one. These systems, therefore, have been forced into using large page sizes and have not always performed as well as expected. Since physical limitations make it impossible to reduce access times of rotating storage devices to the required levels, we must turn to non-rotating storage devices and rely increasingly on parallel data channels and asynchronous logic in order to effect completely successful memory pooling.

Therefore, the potential for effective pooling of memory resources already exists in contemporary computer systems.

It is not the case in contemporary systems that a potential exists for achieving the degree of processor pooling needed. There are three reasons for this.

The first reason is complexity of interconnection. In order to satisfy objectives of reliability, expandability, and programming generality, it has been standard practice to allow each of the (say) n processors free and unrestricted access to each of the (say) m main memory modules, as indicated in Figure 8-5. It is not hard to see that the complexity of the interconnection grows exponentially as (mn), whereas the processor-memory capacity

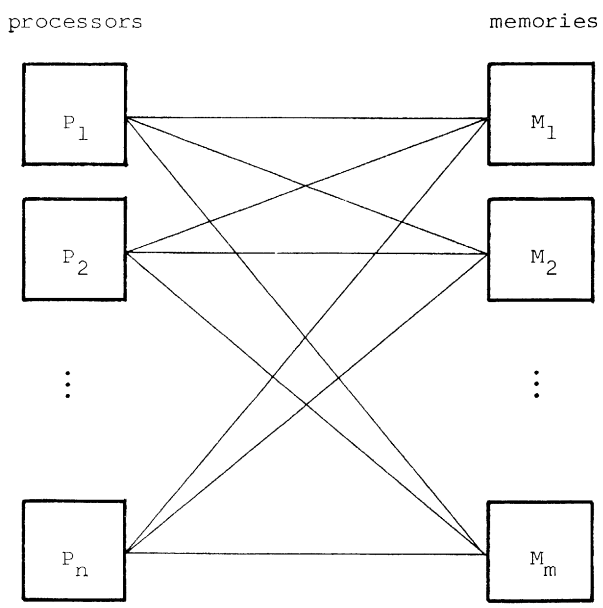


Figure 8-5. Full interconnection of processors and memories.

grows linearly as (m+n). Indeed, realizing the required large number of processors and memory units may not, using full interconnection, be at all feasible. Pooling will have to occur at a much finer level.

The second reason is the large amount of information needed to specify a computation. In Multics, for example, a myriad of tables and lists are needed in order to completely specify a process's name space and to allow it to be interrupted at arbitrary times and yet be properly restarted. These tables have two deleterious effects. First, it is expensive to switch a processor between processes, partly because of all the information that must be loaded into the processor registers, partly because of operating system scheduling functions. Second, the tables that must be loaded into main memory while a process is active occupy considerable space and reduce the memory space available to a program's working set. Unless all this software complexity is rooted out, it will remain impractical to implement pooling of hardware at a fine level. At the end of this chapter we shall disucss a highly organized name-space information structure that may one day produce a solution to these problems.

The third reason is lack of parallelism in the hardware. Processor pooling implies considerable process activity, which in turn implies considerable information movement. Parallelism on the data channels between levels of memory and in the addressing hardware is needed if the memory system is to be capable of handling the information flows induced by busy processor hardware.

Therefore, although there is much work to be done on memory system organization and the structuring information, there is even more work to be done on basic hardware design so that the required degree of processing can be achieved.

8.4. Multilevel Memory Systems

A memory hierarchy, or multilevel memory, is a sequence of increasingly capacious and successively slower-access memory devices. Its general organization is shown in Figure 8-6.

There are n main levels, $M_1,\ldots,M_n$, and m auxiliary levels, $A_1,\ldots,A_m$. Each main level device may be addressed directly by a processor. Information residing in auxiliary devices must be moved into a main level (namely, M_n) before it can be referenced. We assume that information can migrate only between adjacent levels. It is also possible that each auxiliary device (such as B) feeds directly into M_n, rather than into another auxiliary device.

By splitting main memory into several levels, we intend to model computer systems using large core storage in addition to the high speed execution store. In these systems, we would have n=2; for generality, we allow arbitrary n. The auxiliary devices may be drums, disks, tapes, etc.

Each main level device M_k has an access time a_k, representing the time required to reference one word in level M_k. The access times satisfy

$$a_1 < a_2 < \ldots < a_n$$

We take the access time a_1 of the fastest memory M_1 to be one virtual time unit (vtu).

Define T_{ij} to be the traverse time from device i to device j. Here,

$$T_{ij} = \sum_{k=i}^{j-1} T_{k,k+1} \qquad i < j$$

We assume $T_{ij} = T_{ji}$, and that

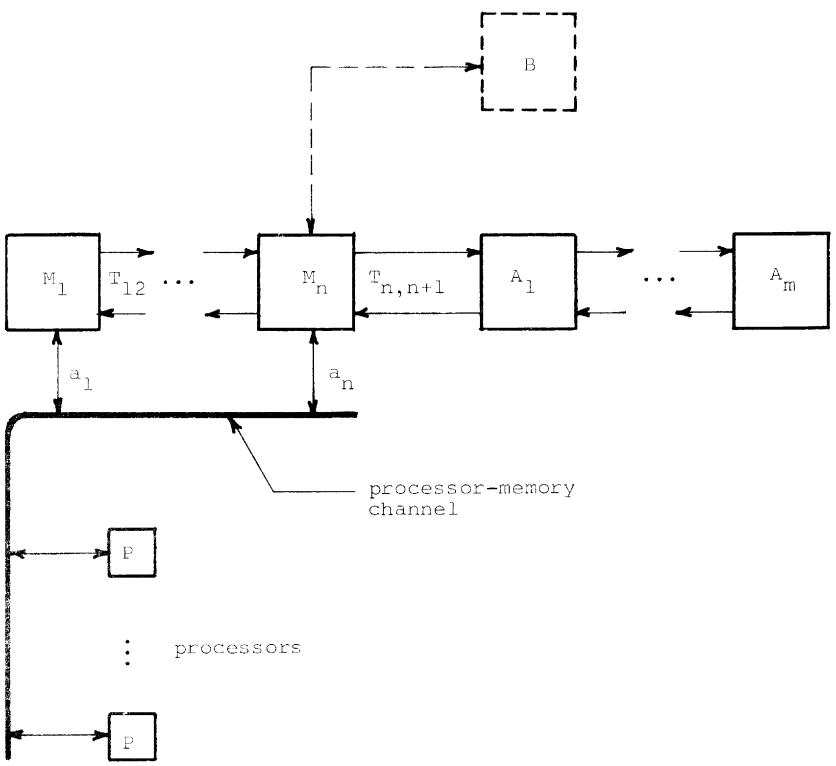


Figure 8-6. Organization of multilevel memory.

$$T_{12} < \cdots < T_{n-1,n} < T_{n,n+1} < \cdots < T_{n+m-1,n+m}$$

These traverse times include queue delays, mechanical positioning times (if any), access times, and page transmission times. The traverse times to auxiliary devices usually depend on rotation times, and so queue delays and transmission times are usually negligible components in them. The traverse times between main levels are composed mostly of transmission times, since access times are small. Typical traverse times, using 1 vtu = 1 microsecond, are:

type of device	access time	traverse time (page=1K words)
thin film	0.1 vtu	100 vtu (0.1 ms.)
high speed core	1 vtu	1000 vtu (1 ms.)
slow speed core	8 vtu	8000 vtu (8 ms.)
high speed drum	10^4 vtu	10^4 vtu (10 ms.)
moving-arm disk	10^5 vtu	10^5 vtu (100 ms.)

When the traverse times depend on the rotation time of a device, we assume that shortest-access-time scheduling techniques, known to be optimum [C2,D3], are used. We may thus assume that each such traverse time is as small as physically possible.

The cost of storing one word for one unit of time is less at lower levels, M_1 being the most expensive and A_m least expensive. The total storage capacity is assumed sufficient for system needs.

The combined capacity of the main levels should certainly be sufficient to contain the balance set. But, in order that

lower traverse times may be effective, we strongly recommend that the main levels be sufficiently capacious that standby set jobs may also have their working sets present in the main levels. Thus, a job re-entering the balance set need not experience paging delays at the start of its quantum, and much higher processing efficiency is possible.

We assume information moves upward only on demand, and downward as it falls out of use.

We assume that the unit of storage in the main levels is the page, and the page size is the same in each main level. The unit of transfer between main levels is the page. We assume the unit of storage in the auxiliary levels is the segment. The unit of transfer between auxiliary levels, and between M_n and A_1, is also the segment. Since information must reside in a main level to be addressable, a reference to information in an auxiliary level must always involve the transfer of a segment into M_n before the reference can be completed.

The basic strategy we adopt for managing multilevel memories is to place information at whatever level results in the least memory-usage cost (space-time product).

There are three questions we must answer:

1. How are the main levels to be managed?
2. How are the auxiliary levels to be managed?
3. What is the role of pre-paging?

We shall use notions of locality and notions of cost to develop guidelines for strategies in each of these areas.

8.4.1. Managing the Main Levels

In a multilevel memory, a working set allocation policy guarantees a computation the use of processors if and only if there is enough uncommitted space among the main levels to contain its working set. Thus, if $W(t,\tau)$ is the working set of some computation, it resides somewhere in

$$M_1 \cup M_2 \cup \ldots \cup M_n$$

We must refine the working-set definition in order to decide at which level each page of $W(t,\tau)$ shall reside.

It should be apparent that we should use a value for τ that permits most of a program to reside in the main levels, because space is more abundant than in a single-main-level system, and because we want to assure higher processing efficiency. For example, given ε, we can choose τ such that the missing-page probability satisfies

$$\lambda(\tau) = 1-F_x(\tau) \leq \varepsilon \tag{8.4.1}$$

where $F_x(\tau)$ is the interreference distribution.

Let **Z** be a program. For each page i in Z we define the reference density $\rho_i(t,\tau)$ at time t to be

$$\rho_i(t,\tau) = \frac{\text{number of references to i in } (t-\tau,t)}{\tau} \tag{8.4.2}$$

Then the working set of the computation using Z is

$$W(t,\tau) = \{i \varepsilon Z \mid \rho_i(t,\tau) > 0\} \tag{8.4.3}$$

Let $\Theta_o,\Theta_1,\ldots,\Theta_n$ be a set of thresholds, where

$$1 = \Theta_o > \Theta_1 > \ldots > \Theta_n = 0 \tag{8.4.4}$$

Then we partition $W(t,\tau)$ into n subsets, where

$$W^k(t,\tau) = \{i \varepsilon Z \mid \Theta_{k-1} \geq \rho_i(t,\tau) > \Theta_k\} \tag{8.4.5}$$

$$W(t,\tau) = \bigcup_{k=1}^{n} W^k(t,\tau) \tag{8.4.6}$$

and $W^k(t,\tau)$ is the set of pages to reside in level M_k.

This definition is based on a refinement of locality, the concept that, during any execution interval, a process favors some of its pages. The reference density $\rho_i(t,\tau)$ measures the degree to which page i is being favored. We assume that the set of favored pages (measured by $W(t,\tau)$) is not likely to change abruptly. In addition, we assume that the reference densities $\rho_i(t,\tau)$ are not likely to change abruptly.

The capacity of each main level can be determined by suitable generalizations of the procedures already discussed in Section 8.1.

The thresholds Θ_k represent tradeoffs between the cost of not having a page in level M_k and running more slowly, versus having a page in level M_k, running more quickly, and paying the overhead of moving the page.

One method for setting the thresholds Θ_k is as follows. Let $\bar{q}$ be the average quantum, over all jobs, and let S be the page size. We wish to decide whether to move page i from level M_k into M_{k-1}. If page i is moved, the saving in running time during $\bar{q}$ is:

$$\bar{q}(a_k - a_{k-1})\rho_i(t,\tau)$$

where a_k is the access time to level M_k. The time required to

move the page is

$$Sa_k$$

since the page transfer proceeds at the slower of the two access times a_k and a_{k-1}. The page should be moved if

(8.4.7) $$\bar{q}(a_k - a_{k-1})\rho_i(t,\tau) > Sa_k$$

That is, whenever

(8.4.8) $$\rho_i(t,\tau) > \theta_k = \frac{Sa_k}{\bar{q}(a_k - a_{k-1})}$$

Hardware not presently commercially available would be needed to implement automatic memory management using these ideas[1]. Whenever the reference density of a page in level M_k exceeds θ_{k-1}, the page is moved into M_{k-1}. Whenever the reference density of a page in level M_k falls below θ_k, the page is a candidate for removal to M_{k+1}. The least recently used non-working set pages in M_n are candidates for removal to A_1.

---

[1] For example, we could associate a τ-bit shift register with each page-block of main memory. The bit pattern in the register is shifted once every time unit. A 1 is entered into the register if the page is referenced, 0 otherwise. The number of 1's contained in the register can be used as a measure of the reference density.

8.4.2. Managing the Auxiliary Levels

Because the high access times make it expensive to reference information stored in auxiliary levels, we assume the the unit of information storage and transfer among the auxiliary levels is the segment. Moreover, an entire segment is moved from A_1 into M_n whenever one of its pages is referenced.

The best strategy for managing auxiliary levels is the least-recently-used strategy. As a segment falls out of use, it finds its way into the lowest levels. A segment is moved upward only when it is referenced.

The reason this strategy is best follows from a locality concept, though not exactly the same concept we have been using for program behavior. The locality concept of interest here is locality in people's behavior and actions. The longer it has been since a person used a certain segment, the more likely it is that he has forgotten about it or that he no longer cares about it, and so the less likely it is that the segment is of immediate use to him.

8.4.3. What About Pre-Paging?

When, if at all, is it worthwhile to load a job's information into main storage prior to its execution?

The chief argument for pre-paging is as follows [L1]. Suppose it requires a traverse time T to acquire a page from a drum auxiliary memory, and that we wish to demand-page an n-page working set into memory. What is the space-time product (cost) of this operation? For k=1,...,n, paging in the k^{th} page results in k pages standing idle in memory, at a cost of kT. The total cost is

$$\sum_{k=1}^{n} kT = \frac{n(n+1)}{2} T \qquad (8.4.9)$$

On the other hand, by careful drum management, it is possible to write out the n-page working set as a contiguous block and read it back in as a contiguous block, the readin operation requiring about one traverse time T (since the page transmission times are so much less than the rotation time). The cost of the operation is nT, since n pages of memory must be reserved before the paging operation can begin. Let C denote the additional cost of identifying working set pages (so that they can be paged out as a block) and carrying out the page-out operation. Then pre-paging is better if

$$nT + C < \frac{n(n+1)}{2} T \qquad (8.4.10)$$

It is usually possible to make the cost C small enough to satisfy eq. 8.4.10. Apparently, then, pre-paging is worthwhile when used to obtain information from a rotating device.

If the information to be pre-paged resides in a large core storage, where the traverse time depends only on page transmission time, pre-paging is not worthwhile. With rotating devices, it takes about as much time to move a block of n pages as it does to move one page, whereas with non-rotating devices it takes one traverse time to move each page. There is clearly no gain from pre-paging information stored in a non-rotating device.

Nevertheless, we do not believe pre-paging is worthwhile in the multiprocess computer systems we have described. The argument given to derive eq. 8.4.10 depended on there being no sharing of information. If a working set is to be paged out in a block, we must exclude the shared pages from this operation. But then, when paging the working set back in, these shared pages may not be available, and additional effort is needed to locate them. The costs C (eq. 8.4.10) of identifying pages, of careful drum management, of handling the page-out operation, and of recovering the missing shared pages, can easily outweigh the potential savings. Other arguments agains pre-paging have been presented in Section 3.4.

We do not, therefore, subscribe to pre-paging working set pages in multiprocess computer systems, unless no sharing is possible. Furthermore, if the multilevel memory system of Figure 8-6 is used, it is unlikely that a working set of a standby set computation will leave the main levels, so there is no need for pre-paging.

We do, however, feel that it is possible to anticipate that an item will be referenced even before it is in a working set, and begin moving it into higher levels beforehand. This requires a new concept of information structures, which we discuss in the next section.

8.5. The Environment Graph Information Structure

Recent work by Dennis [D10] on the design of highly parallel computer systems has produced interesting concepts which can greatly simplify the solution to the resource allocation problem. The most important concept is that of the *environment graph information structure*.

A *naming scheme* is a set of rules for relating occurrences of identifiers to items represented in the computer's memory system. We assume here that the same naming scheme is used throughout the entire memory system from the lowest level to the highest, and throughout the entire execution of a process from its first reference to the last. This means that all references can be handled within the hardware, and in particular that levels of memory can communicate directly with one another without having to consult an operating system procedure[1].

The *environment graph* is a generalization of the file directory structure [D1] to a level of detail so fine that every word has a named position. The environment graph is a directed

---

[1] Multics-like systems do not have a uniform naming scheme. All user information is embedded in a system-wide file directory structure [D1]. A program makes its first reference to a segment by means of a *tree name*, which may incur many costly references to little-used file directories stored in auxiliary levels before the desired segment is located. Once located, a segment is assigned a *segment number*; subsequent references take place using segment numbers and are handled automatically by hardware. The dreadful inefficiency of referencing information buried in the file directory structure makes it necessary to have a second, more efficient naming scheme that streamlines later information references.

acyclic graph having a single root node from which there is at least one directed path to every other node in the structure. Every node has a label. Figure 8-7 shows an example of an (unlabelled) environment graph. All paths are assumed to be directed downward.

If v_1 and v_2 are nodes, and there is a directed path from v_1 to v_2, then v_2 is a descendent of v_1. A subgraph is any node v together with all its descendents; subgraphs represent data structures, such as files, arrays, procedures, etc. Figure 8-8 shows how the linear sequence of instructions

$$V = (v_1, v_2, \ldots, v_n)$$

would be represented. The leaf nodes (those with no descendents) represent actual data values, whereas internal nodes represent named information structures (an internal node is always interpreted as the root of a subgraph).

A data value or structure is identified by selecting a path to it from the root node. A special data type, the pointer, may designate some internal node as being the most recent reference point of a process. The process makes new references with respect to its pointer, not with respect to the root.

Define the k-orbit of a node v to be the set of nodes that are connected to v by a shortest undirected path of length k. The k-sphere of a node v is the set of j-orbits for $1 \leq j \leq k$. If a process has its pointer at node v it will generally make its next reference to some node in the 1-sphere of v, its second reference to some node in the 2-sphere of v, and its k^{th} reference to some node in the k-sphere of v. Therefore the environment

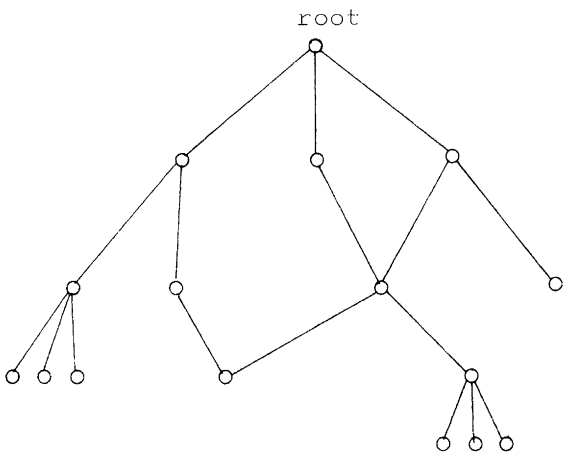


Figure 8-7. An environment graph.

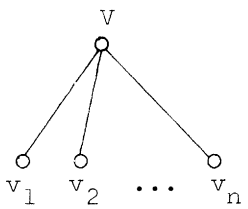


Figure 8-8. Representation of a linear sequence of words.

graph can be used to anticipate references: if we observe the pointer at node v, we can say reliably that the next k references will occur within the k-sphere of v.

Consider the multilevel memory system shown in Figure 8-9. Suppose a node v residing in level M_k is referenced, requiring it to be moved into level M_1. In anticipation of future references, v's 1-orbit should be moved into level M_1, its 2-orbit into level M_2, and so on: when a node is moved upward k levels, its (n-k)-sphere is moved upward k levels. The k-orbit of any node v in level M_1 should not lie below level M_k in the memory system.

Since the file directory structure used in many contemporary computer systems may be regarded as an environment graph whose nodes are segments (a node is a directory segment if and only if it is an internal node), a similar procedure might be used to anticipate segment references. Keep all of a segment on one level. If we observe a directory at level k is consulted, we bring it into level M_1, all its contents to level M_2, etc.

By using the environment graph information structure together with a uniform naming scheme and highly parallel automatic memory management hardware, these goals are met:

1. There is sufficient detail in the environment graph to specify a process, so that little more than a pointer is needed to remember where the last reference took place. This eliminates complex auxiliary tables needed to specify a computation, conserves memory, and permits rapid inter-process switching.

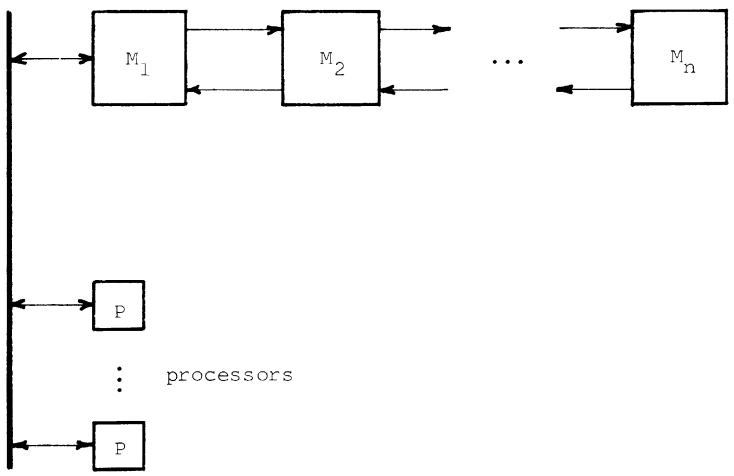


Figure 8-9. Multilevel memory for use with environment graph.

2. Sharing is natural to implement. User A, whose local root node is v_A, can share subgraph v_B of user B simply by introducing the edge (v_A, v_B) into the environment graph (with B's permission).
3. Protection is natural to implement. User A cannot reference any node v_B for which there is no path (v_A, v_B); and the path cannot be established with permission from user B.
4. Locality is implied by the k-spheres. Given that a process has referenced node v, its next k references will generally fall within the k-sphere of v. In managing multilevel memories, the k-orbit of any node v in level M_1 should not lie below level M_k. Working set concepts can be used to decide when a node is to be moved downward.

8.6. Summary

Programmers and system designers should keep in mind certain guidelines, where applicable:

1. locality.
2. programming generality.
3. uniform naming schemes.
4. pooling of equipment at the finest level of detail.
5. parallelism.
6. ability to manipulate small quantities of information.

The equipment configuration can be described analytically. Relations among program properties, processor-memory resources, and traverse times were derived. There is strong evidence favoring the use of large core storage at the upper levels of memory.

In order to utilize equipment fully and to obtain the required capacity, it is necessary to pool small hardware units. If this is done successfully, it is possible to obtain many times the capacity with little more equipment than is currently used in computer systems.

Management of multilevel memories can be handled using generalized working set concepts. The environment graph information structure provides a method for anticipating information references.

CHAPTER 9

Performance Measures and Accounting Procedures

9.0. Introduction

Once we have accepted the working set model and the ideas of demand and balance as being valid and useful approaches to the resource allocation problem, the set of performance measures is more clearly defined. We shall review the relevant probability distributions and indicate how their measurement is useful, not only for proper regulation of the computer system, but also for assisting the administration in setting its operating policies. We shall complete the discussion begun in Chapter 1 regarding metering of resource usage and attributing of charges; of particular interest are methods for charging for shared information.

9.1. What to Measure and Why

The measures fall into three classes, according to their purpose:

1. Working set measures. The distribution of the page interreference intervals, the working set size distribution, and the autocorrelation function for working set size, are needed for the proper (program-dependent) determination of the working set parameter τ, and for better understanding of program behavior.
2. System control measures. The joint demand distribution, the job running-time distribution, and the queue length distribution are needed to decide what equipment is needed and to arrive at a solution for the balance policy from the mathematical programming problem equations. Here, the efficiency, the missing-page probability, and the traverse time serve three purposes: first, to determine sensitivity to thrashing; second, to determine the equipment configuration; and third, to provide additional (non-program-dependent) criteria for selecting the working set parameter τ. Finally, the variation of the balance set demand (p_B, m_B) about (α, β) is useful for deciding on the choices of the balance parameters α and β.
3. Policy-determining measures. The queue-length distribution (equivalently, the distribution of unserviced demands in the standby set) provides indicators to the administration when user community demand is outstripping supply. The relationships among total community demand, bidding, and price, will have to be measured in order to be able to set prices.

We discuss each measure separately and indicate how it applies to each of these three categories.

(1). Interreference Distribution $F_x(u)$. (cf. Section 4.1). The page interreference intervals x, so intimately related to working set properties, have appeared again and again in our discussions. Although we defined them in virtual time, because virtual time renders invisible the vagaries of paging and arbitrary sequencing of scheduled jobs, we can also define them in real time and obtain directly the real-time working set properties. In order that we can do this with the assurance that the derivations are correct, we must first convince ourselves that page waits and scheduling interrupts are distributed uniformly among the jobs. Since working set memory management strategies assure statistical independence among jobs, and since the scheduler is assumed fair, we may be assured of non-distorted measures.

(2). Working set size distribution $F_\omega(u)$. (cf. Section 4.4). Measurements of individual working set sizes are needed to obtain more insights into the behavior of programs, answering questions such as: How strong is locality across the range of program types?, How does $\omega(t,\tau)$ vary across the execution of a program?, How successful can programmers who attempt to design programs with small, compact working sets be?

(3). Correlation function $R_\omega(u,\tau)$. (cf. Section 4.8). The correlation between working set size at two times is invaluable not only for examing locality, but also for assisting in the

proper choice of τ and evaluating the predictive ability of a measurement of the working set size. To determine $R_\omega(u,\tau)$, proceed as follows. Let $\{t_n\}_{n=0}^{N}$ be a long sequence of N equally-spaced instants at which the size $\omega(t,\tau)$ is sampled, and let $\omega_i=\omega(t_i,\tau)$ denote the size at time t_i. Then the value of $R_\omega(u,\tau)$ at time spacing u_i is:

$$R_\omega(u_i,\tau) = \frac{1}{N-i} \sum_{k=0}^{N-i} \omega_k \, \omega_{k+i}$$

Two things must be noted: the number N of samples must be large so that i may become large enough, with N>>i, to make the samples ω_k and ω_{k+i} statistically independent; and $R_\omega(u,\tau)$ depends on τ, and so it will have to be measured for a family of τ-values.

(4). Working Set Intersections. (cf. Section 3.1.3). A study of the size of the intersection between the working sets $W(t,\tau)$ and $W(t+\gamma,\tau)$ of a certain process, as a function of γ, would provide insight into the predictive ability of working sets. Also of interest is the effect of an interaction during $(t,t+\gamma)$ on the intersection, as a function of the duration of the interaction.

(5). The running-time distribution $F_q(u)$. (cf. Section 6.2.2). In the case of single-process computations, this distribution is useful for determining processor demand. This distribution may not be particularly valuable in the case of multiprocess computations, in which we are more interested in the number, rather than the duration, of component processes. Moreover, since q is defined to be interval between successive interactions,

the distribution $F_q(u)$ tells how often a process will be blocked.

(6). Joint demand distribution $F_{pm}(u,v)$. (cf. Section 7.4). Knowledge of this distribution is needed to obtain a solution (a balance policy) from the mathematical programming problem described in Sections 7.3 and 7.4. Assuming a fair balance policy, $F_{pm}(u,v)$ is easily measured by taking samples of the jobs in the standby set queues. Knowledge of this distribution is also invaluable for assisting the administration in setting prices and deciding when to purchase equipment. If it is observed that either of Pr[p=1] or Pr[m=1] is not small, then either price controls must be enforced to reduce demand or more equipment must be purchased.

(7). Queue length distributions $F_{n|pm}(y|u,v)$. This gives the length n of the queue at the point (u,v) in the standby set demand space, Section 7.4. This is again useful for finding the optimum balance policy and for indicating to the administration when the total demand is high enough to warrant new equipment.

(8). Duty factor $\eta(\tau)$. (cf. Sections 4.5 and 5.6). Defined as the fraction of time in the balance set a process is not in page wait, the duty factor is useful for determining sensitivity to thrashing (Section 3.6) and for determining the equipment configuration (Section 8.2) and for estimating processing efficiency.

(9). Missing-page probability $\lambda(\tau)$. (cf. Sections 4.2 and 5.4). Useful for determining sensitivity to thrashing, and for estimating paging rates. It can be measured in a time interval I as

$$\lambda(\tau) = \frac{\text{number of page faults in I}}{\text{number of references in I}}$$

(10). Traverse time T. It is useful to know how often references are made to slower, lower levels of memory, for purposes of determining sensitivity to thrashing and memory system requirements.

(11). Variation of balance set demand. For a given balance policy, we can perform experiments to observe the variation of balance set demand (p_B, m_B) about the desired $(\alpha, ß)$. Doing this for a family of $(\alpha, ß)$ values will yield information useful for determination of $(\alpha, ß)$.

(12). Demand vs. cost curves. (cf. Section 1.4.1). The steady state curves discussed in Chapter 1 relating cost per unit resource to total community demand would be valuable for assisting administration officials set prices. These curves can be composed from the joint demand distribution $F_{pm}(u,v)$ resulting from particular price settings. The administration may have to experiment with prices in order to determine the general character of the curve.

(13). Bidding and inflation. (cf. Section 1.4.3). Assuming the existence of a bidding mechanism, it is necessary to know whether inflation is a problem.

9.2. Charging for Resource Use

Given that memory management at all levels of the memory hierarchy is controlled by means of working set or related strategies, we observe that there may be no need to explicitly bill for processor usage, because a process receives service from a processor if and only if its working set is in main memory. We merely charge an account for the size and duration of its main memory usage; in so doing we implicitly obtain processor usage.

Thus, let $\omega_p(t)$ denote the number of pages in main memory at time t, belonging to process p. If $\omega_p(t)=0$ we understand that p has no pages in main memory (i.e., it is neither running nor page wait). The cost $C_p(I)$ to process p during a real time interval I for main memory usage is

$$C_p(I) = c_o \int_I \omega_p(t)\, dt \qquad \text{some } c_o>0 \tag{9.2.1}$$

and $C_p(I)$ implies both processor and memory usage.

We do not mean to imply that processor usage ought not be metered. We only mean to point out that the same mechanism that meters memory usage can be used to infer processor usage costs.

When there is sharing, we follow the ideas of Section 5.1, letting a page in main memory belong to the working set of the process that most recently referenced it. In this case $\omega_p(t)$ still measures the number of pages belonging to process p at time t, and the cost is still given by eq. 9.2.1. The problem of attributing pages to processes is an implementation problem, and has already been discussed in Section 5.1.

Eq. 9.2.1 can be extended easily to memory usage costs in multilevel memories, where now an <u>owner</u> (Section 5.1) has pages or segments stored at various levels. Let $\omega_k^j(t)$ denote the number of pages held by owner j at level M_k, and suppose c_k is the cost per unit time to store one page at level M_k. Then, during an interval I, owner j is charged

$$C_j(I) = c_o \int_I \sum_{k=1}^{n} c_k \, \omega_k^j(t) \, dt \qquad \text{some } c_o > 0 \tag{9.2.2}$$

for his resource usage.

CHAPTER 10

Conclusions

By constructing abstract behavior models for ongoing multiprocess computations, we have intended to build a framework within which we can understand misunderstood problems, answer unanswered questions, and foresee unforeseen difficulties.

Perhaps more important than the particular models is the basic approach. Every one of the models is based on an appropriate locality concept.

For a variety of reasons it is natural to suppose that, during any interval of execution, the majority of programs will favor a subset of their information, exhibiting locality in their reference patterns.

A process's working set of information -- the pages it has referenced during the last τ units of execution -- is a measure of the set of favored pages. Main memory allocation strategies that grant processors to processes if and only if their working

sets are present in main memory can minimize both memory usage costs and the possibility of thrashing. By defining a page's reference density -- the fraction of the last τ references it received -- we can refine the notion of a working set for memory systems having several levels of directly-addressable memory. Pages with the highest reference densities reside in the highest levels, and pages with the lowest reference densities reside in the lowest levels.

The locality concept behind these working set models assumes that a process is unlikely to abruptly change either its favored pages or its reference densities.

It is quite clear that resource allocation can be very effective if programs do in fact exhibit the locality properties we assume. Indeed, the more pronounced the locality behavior, the more successful the resource allocation. Because the concept of a working set is defined independently of a computer system, it is perfectly reasonable to encourage programmers to construct their programs to have small, compact working sets. There is no need to resort to absurdities, like a declare working set statement in PL/I; all that is necessary is that a programmer get organized, avoid unnecessary jumping from region to region in name space, and employ algorithms and data structures that induce highly local reference patterns.

The definition of system demand is another application of locality concepts, for we assume that it is possible to measure, and act on, a computation's demand before the demand can change significantly.

The definition of system demand can be extended from two resource types -- processor and memory -- to n resource types. One simply defines an n-tuple, whose i^{th} position contains a measure of demand for the i^{th} resource type. Demand for resource types beyond processor and memory can be defined once the appropriate locality concept has been recognized.

Thus, by first asking ourselves the question: What is the locality concept applicable here?, we have been able to construct useful models for program behavior. We suspect that this sort of approach to constructing behavior models may be useful in other areas as well.

The model of a balanced computer system has given insights into the causes of thrashing, into the equipment configuration problem, into means of satisfying other scheduling objectives beyond balance, and into methods of analysis.

When the computer system is continuously balanced, the demand of the balance set is tightly distributed about the desired demand. Although we cannot accurately predict the demand of an arbitrarily given computation, we can accurately predict the demand of the balance set. For this reason it is possible not only to avoid thrashing, but also to effect the proper equipment configuration and be confident that it is correctly matched to the work load.

Balance policies are flexible. By formulating a mathematical programming problem whose objective function is arbitrary, whose constraints enforce both balance and fairness, and whose solution is the set of jobs to be admitted to the balance set, we showed that it is possible to establish reasonable

policies with respect to other, arbitrarily given criteria (such as minimum response time).

The model of a balanced computer system has shown that analysis is possible because computations can be made independent of one another, inasmuch as resource acquisitions of one computation do not interfere with resources in use by another. This model has also shown that processor and memory demands cannot be treated independently: resource allocation decisions must account for both demands at the same time.

The model of a balanced computer system has many applications to contemporary and future problems of computer system organization. This model gives quantitative justification to many intuitive ideas; for example, the intuitive notion of a working set, or the benefits obtainable by sharing information and pooling equipment, or the dependence of thrashing on memory traverse times. This model affords possible solutions to problems for which we have no previous answers, such as the equipment configuration problem or the thrashing problem. This model makes clear which program behavior parameters are important, and what performance measures ought to be used. The model suggests better system organizations, better resource allocation policies. The model can make system designers and administrators feel confident that there is theoretical justification to their decisions. Finally, the model has shown that we are only starting on the long road to understanding the complex behavior of computations and other information-processing activities.

If we have answered some questions, we have raised others. Many of these have already been indicated throughout the text.

The most troublesome problems arise when information is shared. In our work here, we have made processes statistically independent, an assumption that is valid only if processes do not communicate or if shared data is not interlocked. Clearly, many interesting questions concern non-independent processes. It is evident that we want two processes to run concurrently whenever they share information. Ideally, we want to use scheduling mechanisms and policies that somehow automatically group processes together, according as they share information. More work is needed in this area.

We defined a computation to be a collection of mutually cooperating processes and information operating in the same name space, so that a computation is behaviorally well-defined. Might a vaguer definition lead to even more useful models? Can we define degrees of cooperation among processes and let the membership of a computation vary dynamically, according to degrees of cooperation? More work is needed in this area.

Another direction the work can be extended is into the so-called distributed data problem. What locality and working set concepts are important when the data is geographically scattered, as might be the case in a computer network? Is there any way to anticipate, on the basis of present or past behavior, when information should be moved from one geographic location to another?

We have intended to devise new approaches to modelling computations, to spark a new kind of thinking about dynamic information processing activities, and to develop new philosophies about resource sharing and allocation. We sincerely hope we have raised more questions than we have answered.

BIBLIOGRAPHY

The following abbreviations are used:

CACM	Communications of the ACM
JACM	Journal of the ACM
IEEETEC	IEEE Transactions on Electronic Computers

A1. Arden, B. W., et al. Program and Address Structure in a Time Sharing Environment. JACM 13, 1 (Jan. 1966),1-16.

B1. Belady, L. A. A Study of Replacement Algorithms for a Virtual Storage Computer. IBM Systems Journal 5, 2, 1966, 78-101.

B2. Belady, L. A. Biased Replacement Algorithms for Multiprogramming. IBM Thomas Watson Research Center Research Note NC697 (21 March 1967).

C1. Coffman, E. G. Stochastic Models of Multiple and Time-Shared Computer Operations. UCLA Report No. 66-38 (Ph.D. Thesis).

C2. Coffman, E. G. Analysis of a Drum Input/Output Queue under Scheduled Operation in a Paged Computer System. (to be published.)

C3. Coffman, E. G., and Kleinrock, L. Computer Scheduling Methods and their Countermeasures. AFIPS Conf. Proc. 32 (1968 SJCC).

C4. Coffman, E. G., and Wood, R. C. Interarrival Statistics for Time Sharing Systems. CACM 9, 7 (July 1966), 500-503.

C5. Comeau, L. A Study of the Effect of User Program Optimization in a Paging System. ACM Symp. on Op. Sys. Princ. (Gatlinburg, Tenn., Oct 1967).

C6. Corbato, F. J., et al. An Experimental Time Sharing System. AFIPS Conf. Proc. 21 (1962 SJCC), 335-344.

C7. Corbato, F. J. System Requirements for a Multiple-Access, Time-Shared Computer. Project MAC Report MAC-TR-3 (1964).

C8. Corbato, F. J., and Vyssotsky, V. A. Introduction and Overview of the Multics System. AFIPS Conf. Proc. 27 (FJCC 1965)

D0. Dantzig, G. B. Linear Programming and Extensions. Princeton University Press (1963), p. 517ff.

D1. Daley, R. C., and Neuman, P. G. A General Purpose File System for Secondary Storage. AFIPS Conf. Proc. 27 (1965 FJCC).

D2. Denning, P. J. Memory Allocation in Multiprogrammed Computers. M.I.T. Project MAC Computation Structures Group Memo No. 24 (March 1966).

D3. Denning, P. J. Effects of Scheduling on File Memory Operations. AFIPS Conf. Proc. 30 (1967 SJCC), 9-21.

D4. Denning, P. J. The Working Set Model for Program Behavior. CACM 11, 5 (May 1968).

D5. Denning, P. J. A Statistical Model for Console Behavior in Multiuser Computers. (to appear in CACM, 1968).

D6. Denning, P. J. Thrashing: Its Causes and Prevention. (to be published).

D7. Dennis, J. B. Program Structure in a Multi-Access Computer. M.I.T. Project MAC Report MAC-TR-11.

D8. Dennis, J. B. Segmentation and the Design of Multiprogrammed Computer Systems. JACM 12, 4 (Oct 1964), 589-602.

D9. Dennis, J. B., and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. CACM 9, 3 (March 1966), 143-155.

D10. Dennis, J. B. Programming Generality, Parallelism and Computer Architecture. IFIPS Conf. Proc. (1968).

D11. Dijkstra, E. W. The Structure of THE-Multiprogrammed System. CACM 11, 5 (May 1968).

E1. Estrin, G., and Kleinrock, L. Measures, Models and Measurements for time-shared Computer Utilities. Proc. ACM Nat'l Conf. (1967), 85-96.

F1. Fano, R. M., and David, E. E. On the Social Implications of Accessible Computing. AFIPS Conf. Proc. 27 (1965 FJCC), 243-247.

F2. Feller, W. Introduction to Probability Theory and its Applications. (Vol. I, 1950; Vol. II, 1966). New York: Wiley.

F3. Fife, D. W., and Smith, J. L. Transmission Capacity of Disk Storage Systems with Concurrent Arm Positioning. IEEETEC EC-14, 4 (August 1965).

F4. Fife, D. W. An Optimization Model for Time-Sharing. AFIPS Conf. Proc. 28 (1966 SJCC).

F5. Fikes, R. E., et al. Steps Toward a General Purpose Time Sharing System Using Large Capacity Core Storage and TSS/360. Carnegie-Mellon University Technical Report (1968). Also, Proc. 23 Nat'l Conf. ACM (1968).

F6. Fine, G. H, et al. Dynamic Program Behavior under Paging. Proc. 21 Nat'l Conf. ACM (1966).

K1. Kilburn, T., et al. One-Level Storage System. IRE Trans. on Elec. Comp. EC-11, 2 (April 1962).

K2. Kleinrock, L. Optimum Bribing for Queue Position. To appear in Journal of Operations Research.

L1. Lauer, H. A. Bulk Core in a 360/67 Time Sharing System. Computer Design 7, 4 (April 1968), 94-101.

L2. Luconi, F. Asynchronous Computation Structures. M.I.T. Ph.D. Thesis (Dept. of E.E.), January, 1968.

M1. McKellar, A., and Coffman, E.G. The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Evnironment. Princeton University Report TR-59 (Feb. 1968).

N1. Nielsen, N. R. The Analysis of General Purpose Computer Time Sharing Systems. Stanford University School of Bus. Adm. Ph.D. Thesis, 1967.

N2. Nielsen, N. R. The Simulation of Time Sharing Systems. CACM 10, 7 (July 1967).

O1. O'Neill, R. W. Experience Using a Time Sharing Multiprogramming System with Dynamic Address Relocation Hardware. AFIPS Conf. Proc. 30 (1967 SJCC), 611-621.

O2. Oppenheimer, G., and Weizer, N. Resource Management for a Medium Scale Time-Sharing Operating System. CACM 11, 5 (May, 1968).

P1. Papoulis, A. Probability, Random Variables, and Stochastic Processes. New York: McGraw-Hill (1965).

P2. Parkhill, D. The Challenge of the Computer Utility. Addison-Wesley (1966).

P3. Parzen, E. Stochastic Processes. San Francisco: Holden-Day (1962).

P4. Progress Report III. M.I.T. Project MAC (1965-1966), 63-66.

R1. Ramamoorthy, C. V. The Analytic Design of a Dynamic Look-Ahead and Program Segmenting System for Multiprogrammed Computers. Proc. 21 Nat'l Conf. ACM (1966).

R2. Randell, B., and Kuehner, C. Dynamic Storage Allocation Systems. CACM 11, 5 (May, 1968).

R3. Rosen, S. (ed.). Programming Systems and Languages. New York: McGraw-Hill (1967), p. 598.

S1. Saaty, T. L. Elements of Queueing Theory. New York: McGraw-Hill (1961).

S2. Saltzer, J. H. Traffic Control in a Multiplexed Computer System. M.I.T. Project MAC Report MAC-TR-30 (Ph.D. Thesis), July, 1966.

S3. Scherr, A. L. An Analysis of Time Shared Computer Systems. M.I.T. Project MAC Report MAC-TR-18 (Ph. D. Thesis), June 1965.

S4. Selwyn, L. L. The Information Utility. Industrial Management Review, 7, 2 (Spring 1966).

S5. Shemer, J., and Shippey, G. Statistical Analysis of Paged and Segmented Computer Systems. IEEETEC EC-15, 6 (Dec. 1966).

S6. Slotnick, D. Achieving Large Computing Capabilities Through an Array Computer. AFIPS Conf. Proc. 30 (1967 SJCC), 477-482.

S7. Smith, J. L. Multiprogramming under a Page on Demand Strategy. CACM 10, 10 (Oct. 1967), 636-646.

V1. Varian, L., and Coffman, E. G. An Empirical Study of the Behavior of Programs in a Paging Environment. CACM 11, 5 (May 1968).

V2. Vyssotsky, V. A., et al. Structure of the Multics Supervisor. AFIPS Conf. Proc. 27 (1965 FJCC).

W1. Wegner, P. Programming Languages, Information Structures, and Machine Organization. New York: McGraw-Hill (1968).

W2. Wozencraft, J. M., and Jacobs, I. M. Principles of Communications Engineering. New York: Wiley (1965).

BIOGRAPHIC NOTE

Peter James Denning was born January 6, 1942, in New York City. He resided in New York City until 1945, then in Darien, Connecticut, until 1960. He graduated (1960) from Fairfield College Preparatory School, Fairfield, Connecticut; from Manhattan College (1964), New York City, with a Bachelor of Electrical Engineering Degree; from M.I.T., Cambridge, Massachusetts, with a Master of Science (1965) and with a Doctor of Philosophy (1968). During his four years at M.I.T., he has been associated with Project MAC. For the first two of these four years he held a National Science Foundation Fellowship; for the third year he held a National Science Foundation Traineeship; and for the last year he worked as a Research Assistant.

During his summer, Mr. Denning has worked for Bell Laboratories (1963), IBM Corporation (1964), Project MAC (1965 and 1967), and Aerospace Corporation (1966). His three-semester experience teaching M.I.T. undergraduate subject, Theoretical Models for Computation, developed a strong desire to teach, and so he has joined the faculty of Princeton University as Assistant Professor of Electrical Engineering.

Mr. Denning is a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, and the Association for Computing Machinery.

Mr. Denning married the former Anne DeMarco, of New York City, in August, 1964; his first child, Anne Catherine, was born in February, 1968.

Mr. Denning has the following publications:

1. Queueing Models for File Memory Operation. (S.M. Thesis). Project MAC report MAC-TR-21 (Oct. 1965).

2. Protected Service Routines and Intersphere Communication. Project MAC Computation Structures Group Memo No. 20 (Feb. 1966).

3. Memory Allocation in Multiprogrammed Computer Systems. Project MAC Computation Structures Group Memo No. 24 (March 1966).

4. Effects of Scheduling on File Memory Operations. AFIPS Conf. Proc. 30 (SJCC 1967), 9-21.

5. The Working Set Model for Program Behavior. Comm. ACM 11, 5 (May 1968).

6. A Statistical Model for Console Behavior in Multiuser Computers. (To appear in Comm. ACM during 1968).

7. Thrashing: Its Causes and Prevention. (To be published in 1968).

8. Machines, Languages, and Computation. Textbook co-authored with Jack B. Dennis, to be published by Prentice-Hall, Inc.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 3/27/96

Report # LCS-TR-50

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 304 (311-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): FOLLOWS TITLE PAGE, iii, v, xi, xv, xvii,

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

	Description :	Page Number:
(A)	IMAGE MAP: (1-304) UNNUMBERED TITLE PAGE, UNNUMBERED BLANK, iii, UNNUMBERED BLANK, v, UNNUMBERED BLANK, vii-xi, UNNUMBERED BLANK, xiii-xv, UNNUMBERED BLANK, xvii, UNNUMBERED BLANK, 1-285, UNNUMBERED BLANK (305-311) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)	
(B)	NOTE: SIZE OF MATERIAL SCANNED WAS 5¾" x 9".	

Scanning Agent Signoff:
Date Received: 3/27/96 Date Scanned: 4/2/96 Date Returned: 4/4/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

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

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP None

3. REPORT TITLE

Resource Allocation in Multiprocess Computer Systems

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

Ph.D. Thesis, Department of Electrical Engineering, May 1968

5. AUTHOR(S) *(Last name, first name, initial)*

Denning, Peter James

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1968	297	57

8a. CONTRACT OR GRANT NO. Office of Naval Research, Nonr-4102(01) b. PROJECT NO. NR 048-189 c. RR 003-09-01 d.	9a. ORIGINATOR'S REPORT NUMBER(S) MAC-TR-50 (THESIS) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301

13. ABSTRACT

The dynamic allocation of limited processor and main memory resources among members of a user community is investigated as a supply-and-demand problem. The work is divided in four phases. First is the construction of the working set model for program behavior based on locality; a computation's working set is a dynamic measure of this set of favored information. The second phase is the definition and study of properties of system demand. A computation is the basic demand-making entity, placing demands jointly on processor and main memory resources. Its system demand is a pair (processor demand, memory demand). The third phase is the definition and study of the properties of system balance. Computations that demand resources are segregated into two classes — the standby set, which is temporarily denied the use of system resources, and the balance set, which is granted the use of system resources. The system is balanced when the total system demand matches the system capacity. The fourth phase is to apply all these ideas to the design and administration of multiprocess computer systems.

14. KEY WORDS

Computers	Machine-aided cognition	Real-time computers
Resource allocation	Multiple-access computers	Time-sharing
Multiprocess computers	On-line computers	Time-shared computers

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/02/1996

M.I.T. Libraries
Document Services