MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structures Group

Memo No. 31

RESOURCE ALLOCATION IN A COMPUTER UTILITY

by

Peter J. Denning

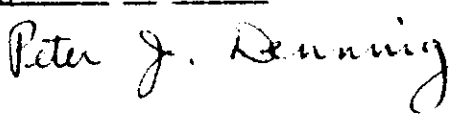
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering
Cambridge, Massachusetts

Proposal for thesis research in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Title: Resource Allocation in a Computer Utility

Submitted by: Peter J. Denning
99 Orchard St.
Somerville, Massachusetts

Signature of Author:



Date of Submission: July, 1967

Expected Date of Completion: June 1968

Brief Statement of Problem:

The problem of distributing limited resources to a user community in a computer utility is investigated. The problem of allocation is reduced to the problem of balancing demands of running programs against available equipment. To do this, a model for program behavior -- the "Working Set Model" -- is defined; starting from this model, precise mathematical meanings can be attached to the notions "demand" and "balance"; these in turn lead to a viable philosophy of allocation within a precise framework. Probabilistic analysis, applied to the models, will characterize the behavior of the computer system from the standpoint of a user and his program.

CONTENTS

CHAPTER I -- INTRODUCTION

Introduction

Distribution of limited resources to users in a computer utility -- the resource allocation problem -- has been ill-defined because of difficult-to-understand interactions among computer system components. Two major factors contributing to the absence of a general treatment are the lacks of adequate models, one for the user, the other for his program.

It is not easy to construct models for either the user or his program, because it seems that for all but the vaguest of models it is easy to contrive counterexamples. Part of the trouble arises from attempts to model all users in all kinds of computer systems. In this thesis we will define very carefully the computer system that serves as a context before beginning the modelling process; then we will find that the user and his program can be characterized quite conveniently by their demands on computer facilities.

Despite sophisticated techniques, such as "multiprogramming"[1],"segmentation"[2], "paging"[2], and "traffic control"[3], the resource allocation problem is far from solved. Terms such as these four denote techniques which are but tools, not solutions, for the problem. Without such tools resource sharing would not be possible; with them there is still no guarantee that allocation is being handled properly. One of our goals, therefore, is to evolve a general, practical philosophy regarding allocation and sharing, with well-defined objectives. Only then will we be able to say that a computer system is achieving allocation aims "properly".

Basically, then, our aim in this thesis is to set up a framework within which we can formulate reasonable models for users and programs; applying techniques of probability theory to a system whose computational activities are regarded in the light of our models, we hope to characterize important behavioral parameters.

Approach

Our philosophy will be concerned throughout with two concepts, "demand" and "balance". Roughly speaking, the demand presented by a program is its probable fractional resource consumption during the next few time units. Balance is a condition that will exist when the sum total of demands by running programs just consumes (balances) available resources. Resource allocation then becomes the question: "Which programs should be run in order to keep the computer system in balance?" Balance has been chosen as a criterion rather than criteria such as "maximum equipemnt utilization" or "user satisfaction" primarily because of mathematical simplicity and secondarily because, with modifications, it encompasses these other, somewhat vaguer, criteria.

In the remainder of this chapter we will examine the responsibilities of the management personnel (hereafter called the "administration") with regard to maintaining balance using available equipment and, through correct interpretation of demand statistics, predicting new equipment needs. In Chapter II we will describe carefully the nature of the computer system whose computational activity we intend to model. In Chapter III we begin the modelling process with a detailed investigation of the "working set model" for program behavior. This model enables us to decide which information is in use by a running program and which is not; this knowledge is vital in dynamic memory management. In addition, this model enables us to determine, at any time, the demand by a running program on the memory resources of the computer. In Chapter IV we study in detail the concepts "demand" and "balance". These are used to formulate resource allocation as a minimization problem, whose objective function attains a minimum whenever a balance condition exists. We introduce a possible allocation algorithm whose overhead in scheduling depends not on the total demand,

but only on the degree of imbalance. A policy of this nature is vitally important, for it assures that the ideas apply to any size computer system, that scheduling overhead is nearly the same in a large system as in a small one.

In the fifth chapter we will discuss the intended course of the thesis research. Of particular interest will be an analysis of a balanced computer system from the viewpoint of a user and his program. Another important aspect is generalization of the ideas to include computer systems having multilevel memory hierarchies and computer systems where a great deal of information is shared.

Administrative Aspects

In this section we will touch briefly on the economic structure underlying our thinking. We do not, however, intend to become involved with issues of computer-utility economics, a field all its own.

The goals of a computer utility[4,5,6] -- to which an allocation strategy must conform -- are these:

1. Distribution of "computing power" to a community of users, who individually would not be in a position to afford the full services of a computer system, who collectively can pay the costs.
2. Accumulation of a vast storehouse of publicly-available shared procedures and programming language systems.

To be consistent with the first of these two goals, an allocation policy should be efficient, "fair", and should not behave foolishly under extremely heavy or extremely light loading conditions. Yet the first of these two goals is not attainab only within the confines of a utility, for the day of the inexpensive desk-top

computer is not far off. The first goal, therefore, is not the essence of a computer utility. It is the second goal that is at once the most basic and the most troublesome. Techniques for dealing with communicating computations are still in the formative stages; there still is no "nice" approach to this problem of "protection while sharing". We know how to handle computations that share equipment but not computations that share information.

Allocation is partly under human control, partly under machine control; each makes decisions on his own time scale, humans handling long-term (hours, days) decisions, the machine making short-term (milliseconds, seconds) decisions. Although our prime concern is with decision procedures for use by the machine, we cannot ignore the tremendous influence the pricing structure has on a human user and thence on the behavior of his program. From a user's standpoint the pricing structure is supply-and-demand oriented, rates for equpment use being lower in "off" hours than during peak periods, and in general varying with the demand.

The basic structure is this: each user is allotted so many chips (akin to "message units" in a telephone utility) per billing period. The rates at which a chip is expended on different types of resource (e.g., processor, memory) depend partly on the user himself and partly on the total demand by other users for these same resources. It will, however, be an individual's perogative how he uses his chips; one user might consume his allotment paying for space in storage media, another might spend his chips paying processor costs. When there is a "fad" (a trend toward exceptional demand) for the use of a particular type of equipment, the administration can counter by raising the rate of chip consumption for that equipment, thereby discouraging excessive use. This would imply that there is some kind of price schedule (via a system command?) available to users indicating current rates for each type of resource. Thus there is feedback between administration and users via the pricing structure;

however here we are concerned neither with questions of stability nor with questions of time lag before price changes are felt.

An essential component of price structure is the ability for a user to bid. If he desires improved service (at correspondingly higher costs), a user may outbid his fellows. If he is unconcerned with the quality of service, a user may underbid, obtaining poorer service at reduced cost. Needless to say there must be some sort of automatic limiting mechanism that prevents some customer with unlimited chips from pre-empting the entire system. This need not imply limited chip allotments; it implies only that the wealthy subscriber be somehow inhibited automatically from keeping subscribers of modest means from obtaining service. Moreover, by assuming the existence of a bidding mechanism we can ignore the delicate questions surrounding "user dissatisfaction", hoping that unhappy users can raise bids if needed.

Many refinements of this basic structure are possible. The most notable of these is the "group structure" -- chips are allotted to groups, whose individual members obtain chips as needed from a group leader. Such refinements do not concern us here.

There are two miscellaneous responsibilities the administration has:

1. Since it is a goal of a utility to provide service to all customers, the administration ought not be efficiency-oriented. Some sacrifice in efficiency is necessary to assure good service.
2. Billing periods ought to be staggered uniformly throughout the month, so that the same number of customers have their billing periods expire the same day. This will avoid the problem arising if everyone should receive his monthly allowance at the same time -- for a week, while customers were affluent, the demand on the system could be staggering.

There are two schools of thought concerning the administration's posture toward the question of total demand from the user community, a total that over a period of weeks or months may rise relentlessly:

1. The administration should exercise control over the total demand at any time by limiting the number of chips outstanding. In some ways this resembles the policies used by parking lot officials, who allocate 150 stickers to fill 100 spaces, on the grounds that (on the average) only 100 cars at a time will show up. That is to say, the administration can ration chips, based on careful interpretation of demand statistics, hoping that the number of users logged in will present a total demand only slightly larger than whatever is required to achieve balance.
2. The administration should be willing to meet customer demand backed by money, adding whatever equipment is needed to expand the capacity of the system, so long as there is anyone willing to pay for it.

Rather than take a stand on this issue, we prefer to develop a strategy acceptable to both schools. This means we must define overload and specify how to detect it; then the administration can decide whether to ration chips or to purchase new equipment. Overload can be defined as follows. First, set tolerance limits on service, such as maximum allowable response time, or minimum allowable processing rate (i.e., fraction of available memory cycles received by a user per unit time). Overload is said to exist when the probability that service does not fall within these limits exceeds some specified number. [This probability would be measured as the fraction of time that service is poor.] We expect that the model we will set up, using balance notions, will be able to predict overload conditions, and enable us to specify measurements that can be made to decide when overload is imminent.

CHAPTER II -- FRAMEWORK

Introduction

We begin this section by discussing a very important assumption of computer utility system design: no advance information on allocation will be available to assist the operating system make scheduling decisions. Then we turn attention to important technical details and terminology.

The models we will set up are intended to model the behavior of computations in the general-purpose computer, the computer utility. For this reason we assume that the operating system must on its own determine the behavior of programs it runs; it cannot count on outside help. Two commonly proposed sources of externally-supplied allocation information are the user and the compiler. We claim neither is adequate.

Because resources are multiplexed, each user is given the illusion that he has a complete computing system at his sole disposal: a virtual computer. For our purposes, the basic elements of a virtual computer are its virtual processor and an "infinite" one-level virtual memory. Dynamic "advice" regarding resource requirements cannot be obtained successfully from users for several reasons:

1. A user may build his program on the work of others, frequently sharing procedures whose time and storage requirements may be either unknown or, because of data dependence, indeterminate.
2. It is not clear what sort of "advice" might be solicited. Nor is it clear how the operating system should use it, for overhead incurred by using advice could well negate any advantages attained.
3. Any advice acquired from a user would be intended (by him) to optimize the environment for his own program. Configuring resources to suit individual may interfere with overall good service to the community of users.

Thus it seems inadvisable at the present time to permit users, at their discretion, to advise the operating system of their needs.

Likewise, compilers cannot be expected to supply information, extracted from the structure of the program*, regarding resource requirements:

1. Programs will be modular in construction; information about other modules may be unavailable at compilation time. Because of dependence on data there may be no way to decide (until run time) just which modules will be included in a computation.
2. Compilers cluttered with extra machinery to preduct memory needs will be slower in operation. Many users are less interested in whether their program operates efficiently than whether it operates at all, and so are concerned with rapid compilation. Furthermore, the compiler is an often-used component of the operating system; if slow and bulky, it can be a serious drain on system resources.

Therefore in this thesis we are advocating mechanisms that monitor the behavior of a computation, making allocation decisions on the basis of currently observed characteristics. Only a machanism that oversees the behavior of a program in operation can cope with arbitrary interconnections of arbitrary modules having arbitrary characteristics.

---

*Ramamoorthy[7] has put forth a proposal for automatic segmentation of programs during compilation.

The Basic Computer System

Although we assume that the reader is already familiar with the concepts of a computer utility[4,5,6], of segmentation and paging[1,2], of program and address structure[8], and of a process and its states[1,3], we will review these topics here.

We will restrict attention to a two-level memory system, indicated by Figure 2.1, because all the complexity of the allocation problem can be found even in this simple system. Only data residing in main memory is accessible to a processor; all other data reside in auxiliary memory, which we regard to have infinite capacity. There is a time T, the traverse time, involved in transferring a unit of data (to be called a page) between memories. T is measured from the moment a page is found to be missing from main memory until the moment the missing page is in main memory ready for use. T is actually the expectation of a random variable composed of waits in queues and mechanical positioning delays. Though it usually takes less time to store into auxiliary memory than to read from it, we shall regard the traverse time T to be the same regardless of which direction the page is moved.

Two basic principles in the design of general puspose computer systems are the abstractions of the notions "process" from "processor" and "address space" from "memory". As we stated earlier, each user is to be given the illusion that he has the entire computing facility at his disposal; accordingly he is given a virtual processor (pseudo-processor) which has most of the capabilities of a real processor, and a virtual memory which has many times the capacity of the real memory. It is the job of the operating system to run a real processor occasionally on behalf of a given virtual processor, giving the user the illusion that his private virtual processor is running at a reduced rate, a rate (under normal load conditions) quite sufficient

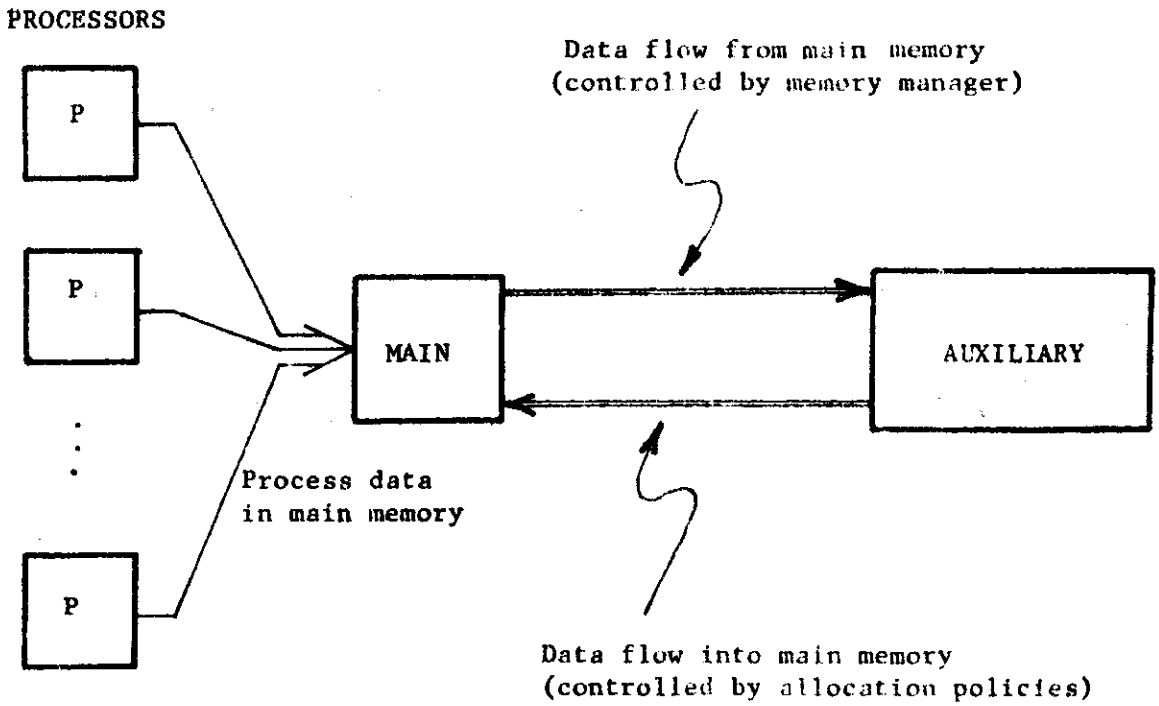


FIGURE 2.1. Two-level Memory System.

for his needs. There is a mechanism (in Multics called the traffic-controller[3]) that handles the assignment of processors to pseudo-processors.

The notion of process is a program in execution by a pseudo-processor. The states of a process are these:

1. running: the pseudo-processor is currently in execution by a real processor.
2. ready: the process is suspended because there is no real processor currently available.
3. blocked: the process could not use a processor even if one were available (perhaps it is awaiting a response from a user or the arrival of data from auxiliary memory).

When talking of processes in execution, we will have to distinguish between "process time" and "real time". Process time is time as seen by a process; that is, as if it ran without interruptions.

The second abstraction -- "address space" -- can also be called "virtual memory" or "name space". The virtual memory is the set of names (addresses) available to a pseudo-processor for use as data identifiers. There is no a priori relation between a name in name space and the location of the corresponding datum in physical memory. 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 the name of a word within S. The address-mapping mechanism establishes the correspondence between a name in name space and the physical location of a data item in memory. For convenience (to the system) in mapping segments of arbitrary size into a memory of fixed size, segments and real memory are sliced into equal-length blocks, called pages. The page, invisible to the programmer, is the standard unit used by the system for information storage and transmission. Hereafter we can (without ambiguity) refer to movement of data as page traffic.

Associated with each pseudo-processor is a segment table listing each segment known to the process. Associated with each segment is a page table listing each page of the segment. If a page is not present in main memory an "in-core" bit* of the corresponding page table entry is OFF; an attempt to reference such a page automatically causes a page fault, which interrupts the process (blocking it) and initiates action to secure the missing page from auxiliary memory. Once the page is loaded in main memory 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 the process leaves the blocked state and enters the ready state. Later on, when the page is removed from main memory, the "in-core" bit of its page table entry is again turned OFF.

A basic allocation problem, "core memory management", is that of deciding just which pages are to occupy main memory. The strategy advocated here -- a compromise against a lot of expensive main memory -- is to minimize page traffic. There are two reasons for this:

1. The more the data traffic between the two levels of memory, the more the computational overhead involved in deciding just what to move and where to move it.
2. Because the traverse time T is long compared to a main memory cycle, too much data movement can result in congestion and serious interference with processor efficiency.

Roughly speaking, a working set of pages ("working set" for short) is the minimum collection of pages that must be loaded in main memory for a process to operate efficiently, without "unnecessary" page faults. According to our definitions, a "process" and its "working set" are but two manifestations of the same ongoing computational activity.

*Consistent with current usage, we will use the terms "core memory" and "main memory" interchangeably.

Summary

This chapter has been a brief review of major computer system concepts; its main purpose is to be sure that the reader and the author both attach the same meanings to terms. We stressed that in a truly general-purpose computer system allocation information can be obtained only by dynamic measurements of ongoing computations, not by soliciting user advice or compiler counsel. We exhibited the basic two-level memory system that will command much of our attention. We discussed the notions of a process and its states; of address spaces, segments, pages; of address-mapping mechanisms, segment and page tables, page faults, and page traffic.

CHAPTER III -- THE WORKING SET MODEL FOR PROGRAM BEHAVIOR

Introduction

In this chapter we investigate in detail the working set model for program behavior, which embodies certain important behavioral properties of programs operating in the utility environment. We precede discussion of the model with a brief review of proposals for techniques of memory management.

Previous work

In this section we outline strategies that have been set forth in the past for memory management; the interested reader will be referred to the literature for detail.

We regard management of paged memories to operate in two stages:

1. paging in: locate the required page in auxiliary memory, load it into main memory, turn the "in-core" bit of the appropriate page table entry ON.
2. paging out: remove some page from main memory, turn the "in-core" bit of the appropriate page table entry OFF.

Management algorithms can be classified according to their methods of paging in and paging out. It is a common characteristic of nearly every strategy that paging in is done on demand; that is, no action is taken to load a page into memory until some process attempts to reference it. To date there have been no proposals recommending look-ahead, or anticipatory page-loading, because (as we have stressed) there is no reliable advance source of allocation information be it the programmer or the compiler. Although the working set is the desired information, it might still be futile to pre-load pages: there is no guarantee a process will not block shortly after resumption, having referenced only a fraction of its working set. The operating system could devote its already precious time to activities more rewarding than loading pages which may not be used. Thus we will assume that paging in is done on demand only, via the

page fault mechanism.

The chief problem in memory management is not deciding which pages to load; it is deciding which pages ought to be removed. For if the page with the least likelihood of being reused in the immediate future is retired to auxiliary memory, the best choice has been made. Nearly every worker in the field has recognized this. Debate has arisen over which strategy to employ for retiring pages; that is, which page-turning, or replacement, algorithm to use. A good measure of performance for a paging policy is page traffic (the number of pages per unit time being moved between memories), since erroneously removed pages add to the traffic of returning pages. In the following we will use this as a basis of comparison for several strategies.

random selection. Whenever a fresh page of memory is needed, a page is selected at random to be replaced. Although utterly simple to implement, this method frequently removes useful pages and results in high page traffic.

cyclic selection. The pages of main memory are ordered in a cyclic list. Suppose the M pages of main memory are numbered $0,1,\ldots,(M-1)$ and that a pointer k indicates that the k^{th} page was most recently paged in. Whenever a fresh page of memory is needed, $[(k+1) \bmod M] \rightarrow k$, page k is retired, and another page brought in to fill the now vacant slot. This method -- also utterly simple to realize -- is based on the principle that programs tend to follow sequences of instructions, so that references in the immediate future will most likely be close to present references. Assuming there is this tendency for page references to cluster, and assuming some kind of uniformity in scheduling techniques, the page which has been in memory longest is least likely to be reused; hence the cyclic list. We see two ways in which this algorithm can fail.

First we question its basic assumption. It is not at all clear that modular programs, which execute numerous inter-module calls, will indeed exhibit sequential instruction fetch patterns. The thread of control will not string pages together; rather, it will entwine them intricately. Second, this algorithm is subject to overloading when used in multiprogrammed memories. When core demand is too heavy, one cycle through the list completes rapidly and the pages deleted are still needed by their processes. This can create a self-intensifying crisis. Programs, deprived of still-needed pages, generate a plethora of page faults; the resulting traffic of returning pages displaces still other useful pages, leading to more page faults, and so on.

oldest unused selection. Each page table entry contains a "use" bit, set ON each time the page is referenced. At periodic intervals all the page table entries are searched and usage records are updated. When a fresh page of memory is needed, the page unreferenced for the longest time is removed. One can see that this method is intrinsically reasonable by considering it acting in a computer where there is exactly one process whose pages cannot all fit into main memory. In this case the most reasonable choice for a page to replace is the oldest unused page. Unfortunately this method too is susceptible to overload when many processes compete for main memory.

Atlas loop detection method. The Ferranti Atlas computer[9] had proposed a page-turning policy that attempted to detect loop behavior in page reference patterns, then minimize page traffic by removing pages not expected to be needed for the longest time. It was successful -- only for looping programs. Performance was unimpressive for programs exhibiting random reference patterns. Implementation was costly.

Various studies have apperared concerning behavior of paging algorithms. Fine, McIssac, and Jackson[10] have investigated the effects of demand paging policies and have questioned whether paging is beneficial. We do not feel that their conclusion applies to the kind of multiprogrammed environment we have described. They studied fixed-size programs, that quickly acquired and retained a large fraction of their pages. Highly interactive, modular programs are likely to behave differently. Not only may program size vary dynamically (according to data dependencies), but also such programs should be using a small fraction of their pages at any one time, and the membership in this set of working pages should be changing constantly.

Belady[11] has compared some of the algorithms mathematically. His most important conclusion is that the "ideal" algorithm should possess much of the simplicity of random or cyclic selection (for efficiency) and some, though not much, accumulation of data on past reference patterns. He has shown that too much "historical" data can have adverse effects (witness Atlas).

In the next section we begin investigation of the working set concept. Even though the ideas are not entirely new[12,13,14], there has been no detailed documentation publicly available.

The Working Set Model

From the programmer's standpoint, the working set of information is the smallest collection of procedure and data items that must be present in main memory to assure efficient operation of his program. We have already stressed that there will be no advance information from either the programmer or the compiler regarding what information "ought" to be in main memory. It is up to the operating system to determine on the basis of page reference patterns whether a page is in use. Therefore the working set of information associated with a process is, from the system standpoint, the set of most recently referenced pages.

We define the working set of information $W(t,\tau)$ of a process at time t to be the collection of data items referenced by the process during the process time interval $(t-\tau,t)$.

Thus, the data items a process has referenced during the last τ seconds of its execution comprise its working set. τ will be called the working set parameter. We will regard the data items in $W(t,\tau)$ as being pages, although they could just as well be any other named data objects. The working set size $\omega(t,\tau)$ is

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

A working set $W(t,\tau)$ has two important, general properties. Both are properties of typical programs, and need not hold in special cases.

P1. Size. It should be clear immediately that the working set size $\omega(t,0) = 0$ since no page reference can occur in zero time. It should also be clear that $\omega(t,\tau)$ as a function of τ is monotonically increasing, since more pages can be referenced in longer process time intervals. Because a process will reference its more-needed pages

rapidly and its less-needed pages slowly, we expect $\omega(t,\tau)$ as a function of τ to have a steep initial slope which diminishes to a more gradual slope. The general character of $\omega(t,\tau)$ is suggested by the idealized curve of Figure 3.1.

P2. Correlation. Program modularity enables us to say something about the correlation between working set sizes at two times, t and $(t+\beta)$. Correlation is useful in constructing storage allocators, for the higher the correlation between $\omega(t,\tau)$ and $\omega(t+\beta,\tau)$, the better a prediction $\omega(t,\tau)$ is of $\omega(t+\beta,\tau)$. In modular programs, control passes randomly from one module to another, in such a way that a working set is more likely to change smoothly, less likely to change abruptly. Thus, for small time separations β, $\omega(t,\tau)$ and $\omega(t+\beta,\tau)$ are highly correlated, meaning that a measurement of $\omega(t,\tau)$ will be a good estimate for the memory requirement during the process time interval $(t,t+\beta)$. For large time separations β, control will have passed through a great many modules during the interval $(t,t+\beta)$; thus $\omega(t,\tau)$ gives little information about $\omega(t+\beta,\tau)$, and so $\omega(t,\tau)$ and $\omega(t+\beta,\tau)$ have much less correlation than for small β. This behavior is suggested in Figure 3.2.

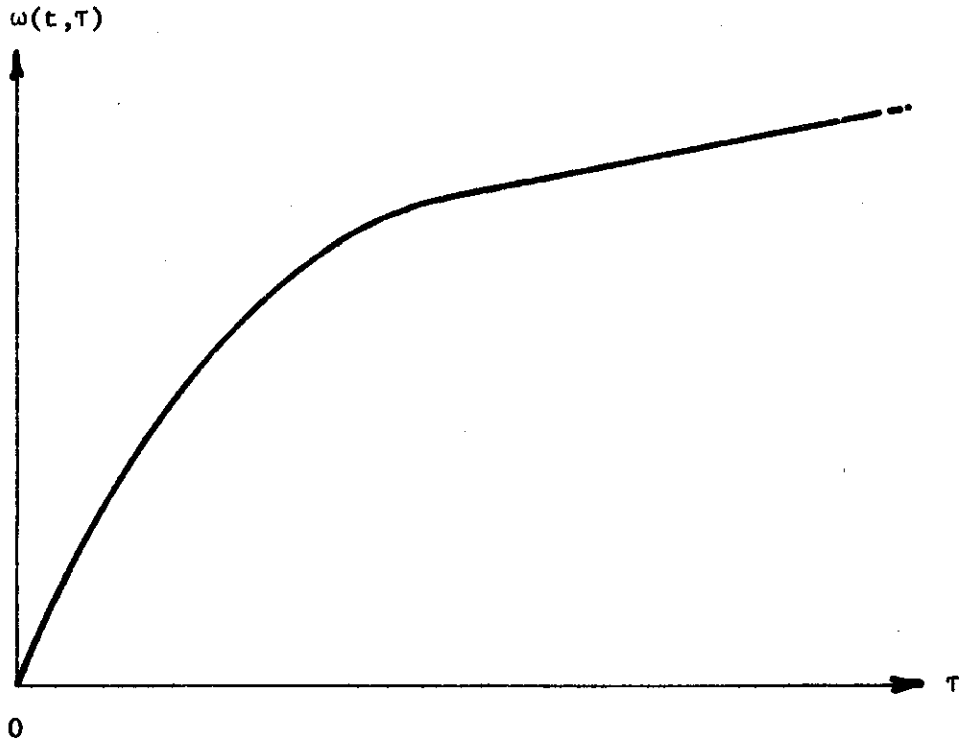


FIGURE 3.1. Behavior of $\omega(t,\tau)$.

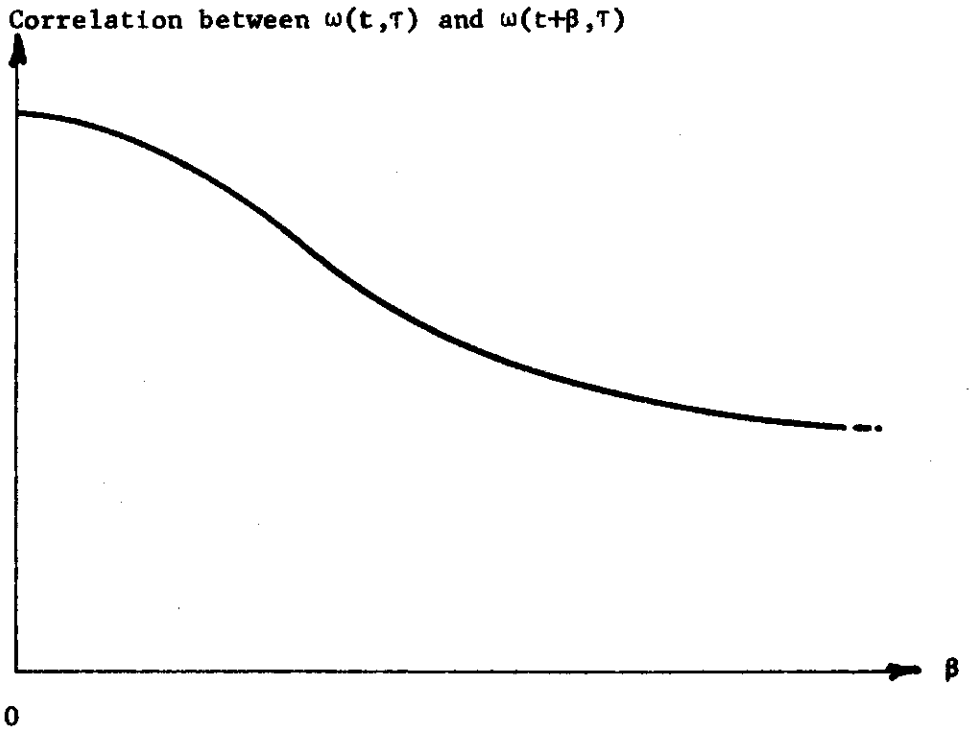


FIGURE 3.2. Correlation between working set sizes.

Choice of τ

The value utimately selected for τ will reflect efficiency requirements and will be influenced by system parameters such as core memory size and memory traverse time. For example, if τ is too small, pages may be removed from main memory while they are still useful, and high page traffic may result. If τ is too large, pages may remain in main memory long after being used last, and wasted main memory may result. Thus the value of τ will have to represent a compromise between too much page traffic and too much wasted memory space.

The following considerations lead us to recommend for τ a value comparable to the memory traverse time T (Figure 2.1). Assuming that memory allocation procedures balk at removing from main memory any page in a working set, once a page has entered a working set $W(t,\tau)$ it will remain in main memory for at least τ seconds. Under the very worst of page-shuffling conditions, a page could be dispatched to auxiliary memory and be recalled immediately; the time for this round trip is two traverse times, 2T. Therefore a highly-shuffled page would spend roughly $\tau/2T$ of its time in main memory. So, for example, if we wished to insure that a page is available in main memory (when needed) not less than 50 per cent of the time, we would have to choose $\tau \approx 2T$.

Use of Working Sets for Memory Allocation

In our discussion so far we have seen two alternative quantities of possible use in storage allocation: the working set $W(t,\tau)$ and the working set size $\omega(t,\tau)$.

Complete knowledge of $W(t,\tau)$, page for page, would be needed if look-ahead were contemplated. We have already discussed why past paging policies have shunned look-ahead, due to the strong possibility that pre-loading could be futile. A program organization likely to be typical of interactive modular programs, shown in Figure 3.3, fortifies our previous argument against look-ahead. 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. Each time the process of this program blocks, the working set $W(t,\tau)$ may change radically -- sometimes only A may be in $W(t,\tau)$, at other times one of the B-procedures and D may be in $W(t,\tau)$. We can see that when the process blocks for an interaction with the user the pages of $W(t,\tau)$ are likely to be different after blocking from before blocking. Thus, the fact that a process blocks for an interaction (not page faults) can be a strong indication of a change in $W(t,\tau)$. Therefore the look-ahead, most often used just after a process unblocks, would probably load pages not likely to be used.

Knowledge of $\omega(t,\tau)$ with demand paging suffices to manage memory well. Before running a process we insure that there are enough pages of memory free to contain its working set $W(t,\tau)$, pages of $W(t,\tau)$ filling free slots on demand. By implication, enough free storage is reserved so that no page of another working set is displaced by a page of $W(t,\tau)$ [as can be the case with the random, cyclic, or oldest-unused policies]. Accordingly we will use the working set size $\omega(t,\tau)$ as a measure of memory demand for storage allocation.

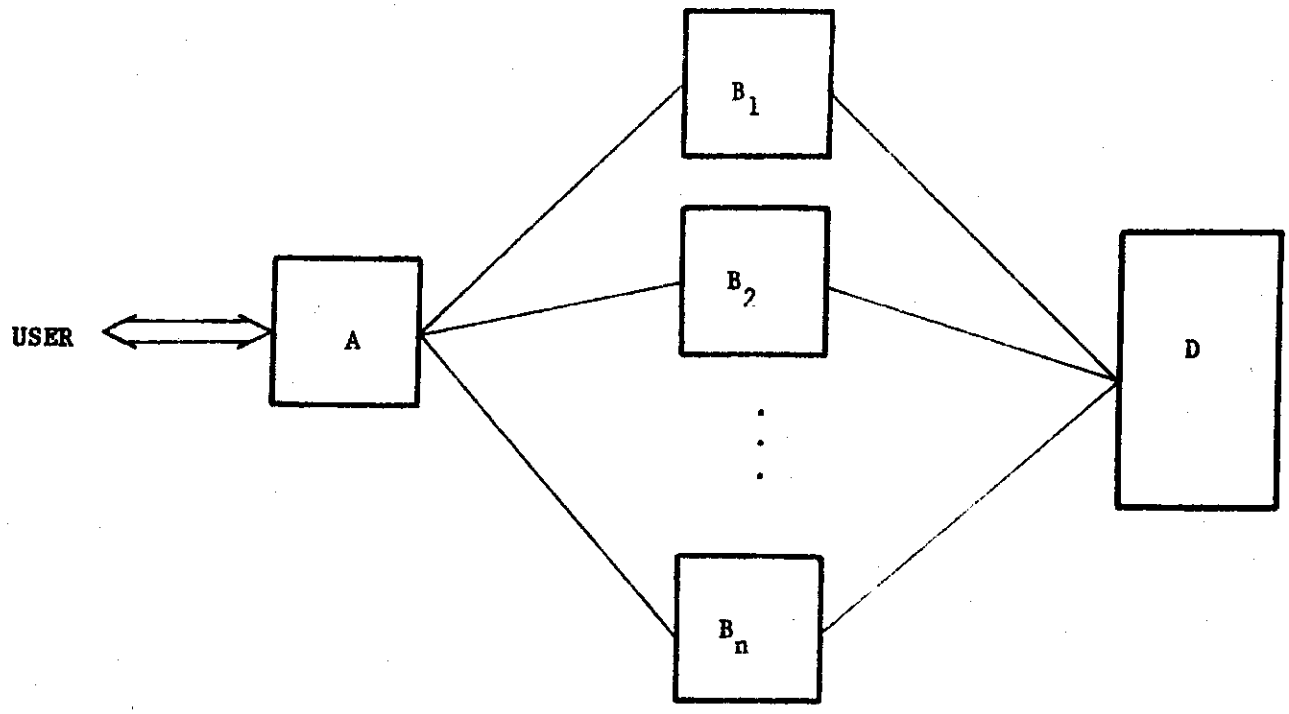


FIGURE 3.3. Organization of a Program.

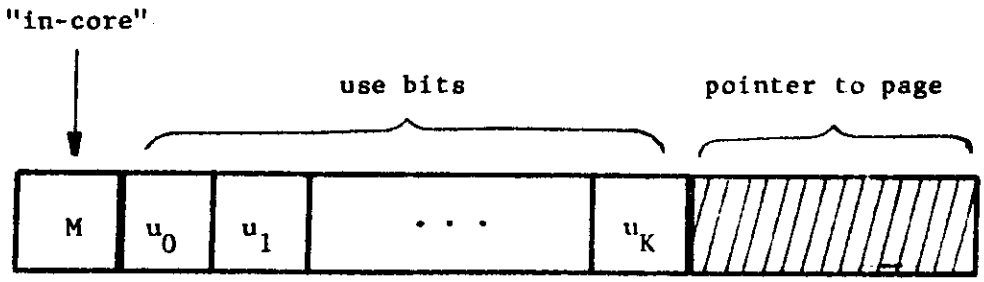
Detecting $W(t,\tau)$

According to our definition, $W(t,\tau)$ is the set of its pages a process has referenced within the last τ seconds. This suggests that memory management can be controlled by hardware mechanisms, by associating with each page of main memory a timer. Each time 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 the page for removal whenever the space is needed. In the appendix we describe such a hardware memory management mechanism, hardware that can be housed within the memory boxes. The mechanism has two interesting features:

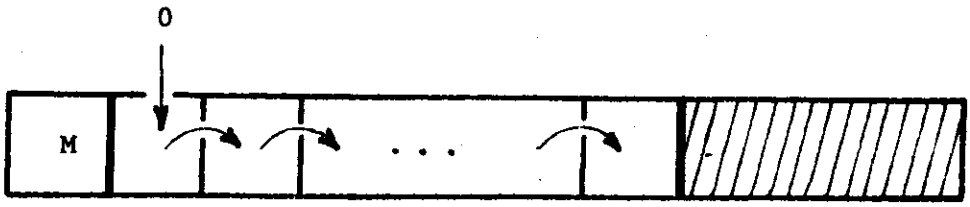
1. It operates asynchronously and independently of the supervisor, whose only responsibility in memory management is handling page faults. Quite literally, memory manages itself.
2. Analog devices such as capacitative timers could be used to measure time intervals.

Unfortunately it is not practical to add on hardware to existing systems. We seek a method of handling memory management with the software. The procedure we propose here samples the page table entries of pages in core memory at intervals of σ seconds (σ is called the sampling interval) where $\sigma = \tau/K$, K an integer constant being chosen to make the sampling intervals as "fine grain" as desired. On the basis of page referenced during each of the last K sampling intervals, the working set $W(t,K\sigma)$ can be determined, as follows.

As indicated by Figure 3.4, each page table entry contains an "in-core" bit M, where M=1 if and only if the page is present in main memory. It also contains a string of use bits $u_0,u_1,\ldots,u_K$. Each time a page reference occurs, $1 \rightarrow u_0$. At the end of each sampling interval σ, the bit pattern contained in $u_0,u_1,\ldots,u_K$ is shifted one position, a 0 enters u_0, and u_K is discarded:



TYPICAL PAGE TABLE ENTRY



SHIFT AT END OF SAMPLING INTERVAL σ

FIGURE 3.4. Page table entries used to detect $W(t,K\sigma)$.

$$
\begin{aligned}
u_{K-1} &\rightarrow u_K \\
&\vdots \\
u_0 &\rightarrow u_1 \\
0 &\rightarrow u_0
\end{aligned}
\tag{3.2}
$$

Then the logical sum U of the use bits is computed:

$$U = u_0 + u_1 + \ldots + u_K \tag{3.3}$$

so that U=1 if and only if the page has been referenced during the last K sampling intervals, that is, if and only if it is in $W(t,K\sigma)$. If U=0 when M=1 the page is no longer in a working set and may be removed from main memory.

Implementation

The previous discussion has indicated a skeleton for implementing memory management using working sets. Now we will fill in the flesh.

If the working set ideas are to be consistent with our stated allocation aims, an implementation should have these properties:

1. Since there is such an intimate relation between a process and its working set, memory management and process scheduling must be closely related activities. One cannot take place independently of the other.
2. Efficiency should be of prime importance. When sampling of page tables is done, it should be only on pages in currently changing working sets, and it should be done as infrequently as possible.
3. The mechanism ought to be capable of providing measurements of current working set sizes and processor time requirements for each process.

Figure 3.5 displays an implementation having the desired properties. Each rectangular box represents a delay. The solid arrows indicate the paths that may be followed by a process-identifier while it traverses the network of queues. The dashed boxes and arrows show when operations are to be performed on the time-used variable t_i associated with process i; processor time used by process i since it was last blocked (page faults excluded) is recorded in t_i. We shall follow a single process through the system to see what transpires:

1. When process i is created, an identifier for it is placed in the ready list[1,3], which is a list of all processes in the ready state. Process are selected from the ready list to enter service according to the prevailing priority rule.
2. Once selected from the ready list, process i is assigned a quantum q_i, which upper-bounds its time in the running list. This list is a cyclic queue; process i cycles through repeatedly, receiving bursts o

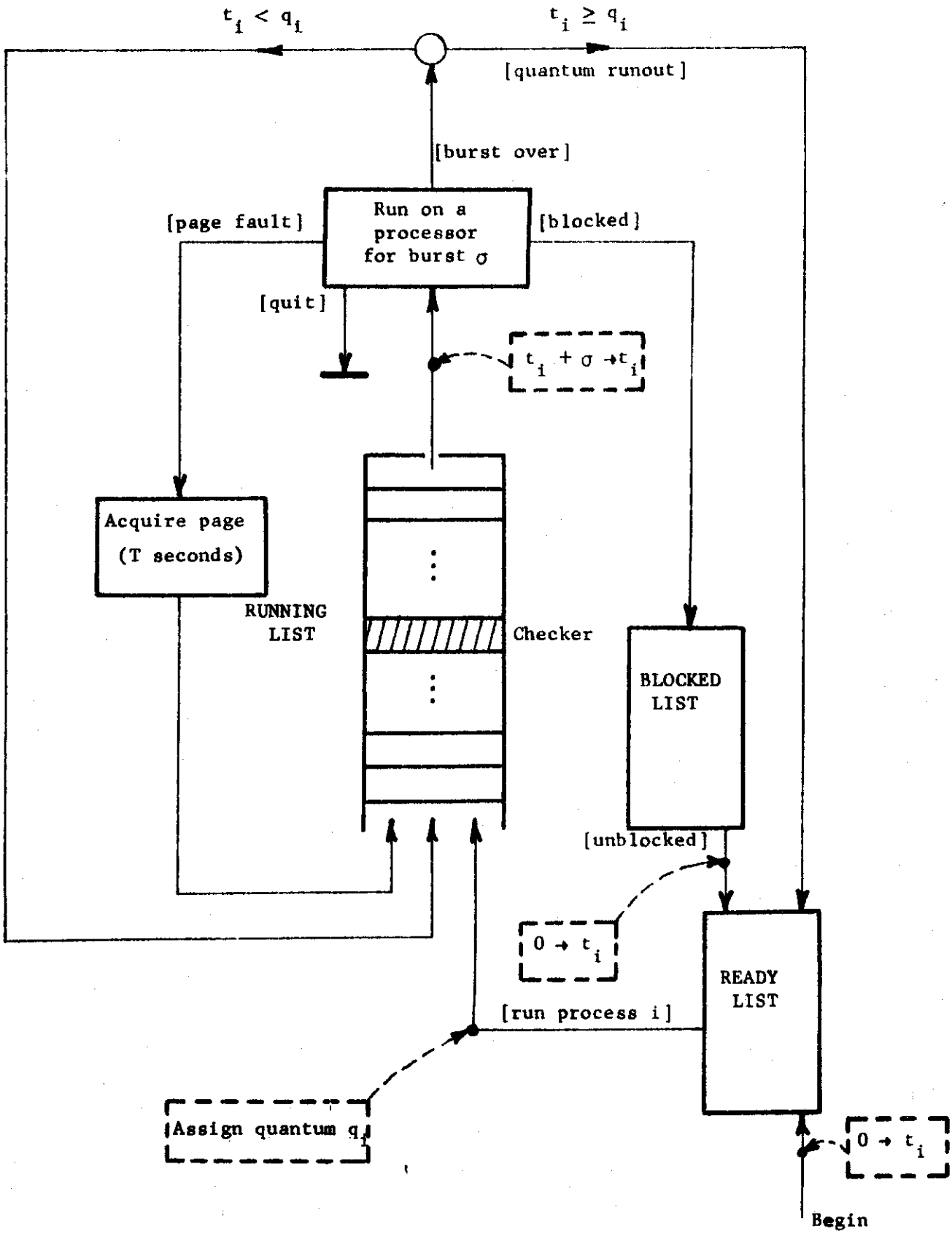


FIGURE 3.5. Implementation of Scheduling.

of processor time until it blocks or exhausts its quantum q_i. Note that the processor burst σ is also the sampling interval.

3. If process i blocks, its identifier is placed in the blocked list, where it remains until the process unblocks; it is then re-entered in the ready list.

Perennially present in the running list is a special process, the checker. The checker performs core management functions (and as we will see in the next chapter, it also performs other allocation duties). It samples the page tables of each process that has received service since the last time it (the checker) was run, removing pages according to the algorithm discussed at equations 3.2. It should be clear that if the length of the running list is ℓ, sampling of page tables takes place only every $\ell\sigma$ seconds, not every σ seconds.

Associated with process i is a counter w_i giving the current working set size $\omega_i(t,\tau)$. Each time a page fault occurs a new page enters $W_i(t,\tau)$ and so w_i must be increased by one. Each time a page is removed from $W_i(t,\tau)$ by the checker, w_i is decreased by one.

Having completed its management duties, the checker replenishes vacancies in the running list by selecting jobs from the ready list according to the prevailing priority rule. More will be said about this in the next chapter.

Sharing

Sharing finds its place naturally.

When pages are shared, working sets will overlap. If Arden's[8] suggestion concerning program structure* is followed, sharing of data can be accomplished without modification of the regime of Figure 3.5. If a page is in at least one working set, the "use" bits in the page table entry will be turned on and the page will not be removed. To prevent anomalies, the checker must not be permitted to examine the same page table more than once during one of its scans. Allocation policies should tend to run two processes together in time whenever they are sharing data (symptomized by overlap of their working sets) in order to avoid unnecessary reloading of the same information.

*If a segment is shared, there will be an entry for it in the segment tables of each participating process; however, each entry points to the same page table. Each physical segment has exactly one page table describing it, but a name for the segment may appear in many segment tables.

Summary

In a synopsis of previous work on core memory management culled from varied sources we saw that memory management operates in two basic stages: page-in and page-out. Page-in should be done on demand, without look-ahead· page-out can be done in any of several ways. Page-out is the heart of the problem, for if pages least likely to be reused in the near future are removed, the traffic of returning pages is minimized. The working-set strategy, which attempts to have present in main memory every page "needed" by each running process, which balks at the idea of displacing any page in a working set from main memory, is offered as a viable solution to the problem. Choice of the working set parameter τ will depend on compromises among page traffic, amount of unused memory, and required availability of pages. The working set size is the best quantity to measure for memory allocation purposes, and leads to economical implementation of scheduling and allocation functions.

Looking at this chapter from a slightly different viewpoint, we have seen four major contenders for paging policies: Random, Cyclic, Oldest-unused, and Working-Set. For modular programs, the type ultimately expected to predominate in the utility environment, Random brings on the highest page traffic, Working-Set the lowest. Although Random and Cyclic are the most inexpensive to implement, the added cost of Working-Set is more than offset by its accuracy and its compatibility (as we shall see) with generalized allocation functions.

In the next chapter we look into notions of "balance"; there we will see how the notion of working set, operating together with the implementation proposed by Figure 3.5, blends into one decision function the heretofore independent activities of process-scheduling and memory-management.

CHAPTER IV -- BALANCE ALLOCATION

Introduction

In this chapter we are seeking to formulate an allocation policy as a minimization problem

$$\{\text{minimize } f\}$$

where the function f is at a minimum whenever a condition of "balance" exists. We begin by defining carefully the notions "demand" and "balance", and showing how demand is measured dynamically by the operating system. The last part of the chapter illustrates a possible allocation policy founded on the ideas defined; the resulting policy has the feature that overhead will depend on the "degree of imbalance" and not on load conditions or the size of the computer system.

Demand

Our purpose in this section is to define "memory demand" and "processor demand", then combine these into the single notion "demand".

We define the memory demand m_i of process i to be

$$m_i = \min\left(\frac{w_i}{M}, 1\right) \qquad 0 \le m_i \le 1 \tag{4.1}$$

where M is the number of pages of memory, and $w_i = \omega_i(t,\tau)$ is the working set count maintained by the allocation strategy of Figure 3.5. If a working set contains more than M pages (it is bigger than memory) we regard its demand to be m = 1. Presumably M is large enough so that the probability (over the ensemble of all processes) Pr[m=1] is very small.

Processor demand is difficult to define without some discussion. We want to define the processor demand p_i of process i to be the expected fractional processor requirement before the next time process i blocks (exclusive of page faults). One method of doing this is described below.

Let q be the random variable of processor time used by a process between interactions*. In general character, the probability density function for q, $f_q(x)$, is hyperexponential (for a complete discussion, see Fife[15]):

$$f_q(x) = c\,a\,e^{-ax} + (1-c)\,b\,e^{-bx} \qquad \begin{matrix} 0 < a < b \\ 0 < c < 1 \end{matrix} \tag{4.2}$$

$f_q(x)$ is diagrammed in Figure 4.1; most of the probability is concentrated toward small values of q, and $f_q(x)$ has a long exponential tail. Given that it has been γ seconds (process time) since the last interaction, the conditional density function for time until next interaction is

$$f_{q|\gamma}(x) = \frac{f_q(x+\gamma)}{\int_\gamma^\infty f_q(z)\,dz} = \frac{c\,a\,e^{-ax}e^{-a\gamma} + (1-c)\,b\,e^{-bx}e^{-b\gamma}}{c\,e^{-a\gamma} + (1-c)\,e^{-b\gamma}} \tag{4.3}$$

which is just that portion of $f_q(x)$ for $q \geq \gamma$ with its area normalized to unity. The conditional expectation is

$$Q(\gamma) = \int_0^\infty x\,f_{q|\gamma}(x)\,dx = \frac{\frac{c}{a}e^{-a\gamma} + \frac{(1-c)}{b}e^{-b\gamma}}{c\,e^{-a\gamma} + (1-c)\,e^{-b\gamma}} \tag{4.4}$$

---

*A process "interacts" when it communicates with something outside its name space, such as a user at a console, or another process.

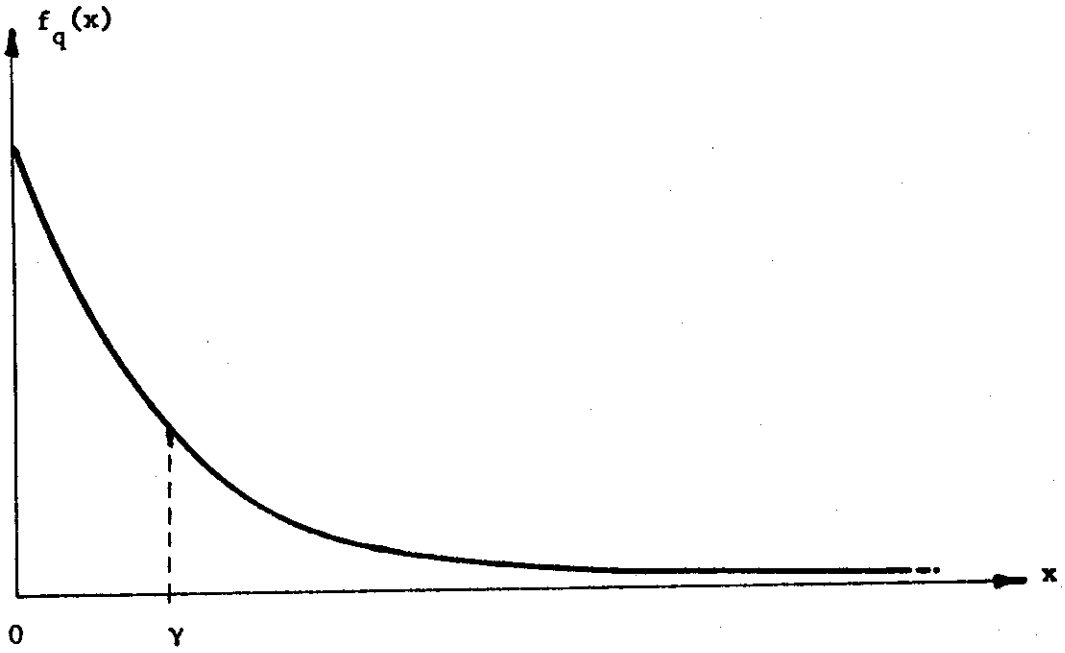


FIGURE 4.1. Behavior of $f_q(x)$.

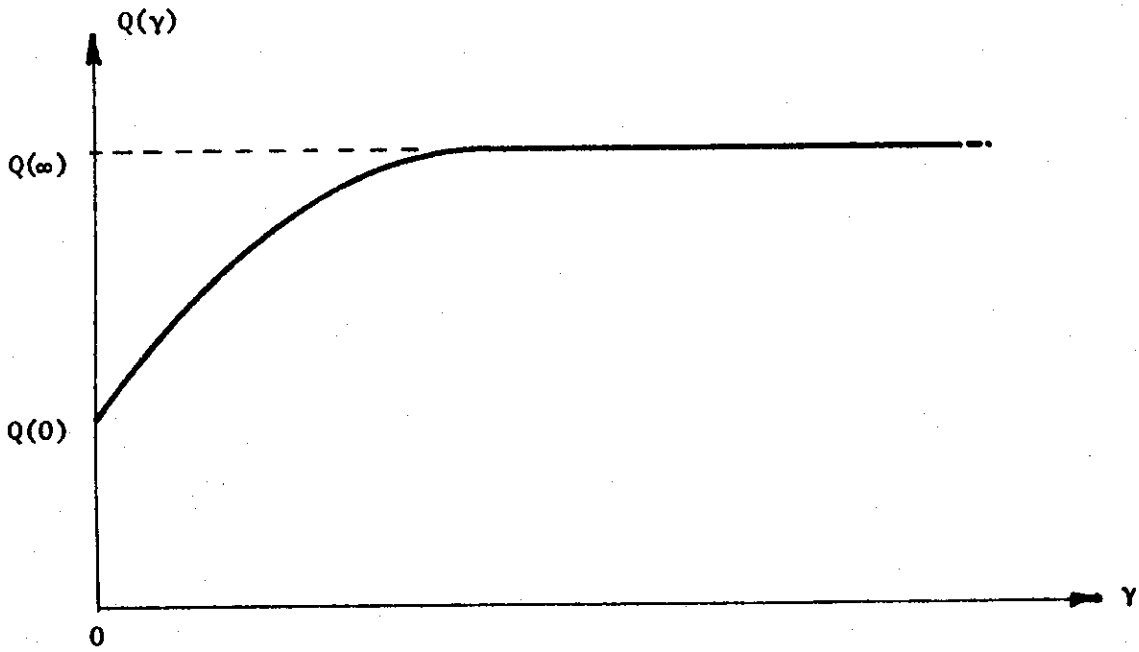


FIGURE 4.2. Conditional expectation function Q(Y)

The conditional expectation function $Q(\gamma)$ is shown in Figure 4.2. It starts at $Q(0) = \frac{c}{a} + \frac{1-c}{b}$ and rises to a constant maximum of $Q(\infty) = \frac{1}{a}$. Note that for large enough γ, the conditional expectation becomes independent of γ. The conditional expectation $Q(\gamma)$ is a useful prediction function -- if γ seconds of processor time have been used by some process since its last interaction, we may expect $Q(\gamma)$ seconds to elapse before its next interaction. It should be clear that the conditional expectation function $Q(\gamma)$ can be determined and updated automatically by the operating system.

We define the processor demand p_i of process i to be

$$p_i = \frac{Q(t_i)}{N\ Q(\infty)} \qquad p_i \leq \frac{1}{N} \tag{4.5}$$

where N is the number of processors and t_i is the time-used variable maintained by the allocation strategy of Figure 3.5.

We define the demand $\underline{d}_i$ of process i to be a pair

$$\underline{d}_i = (p_i, m_i) \tag{4.6}$$

where p_i is its processor demand (equation 4.5) and m_i is its memory demand (equation 4.1). That the processor demand is p_i tells us to expect process i to use p_i of the processors for the next $Q(\infty)$ seconds, before its next interaction*. That the memory demand is m_i tells us process i is most probably going to use $(m_i M)$ pages of memory during the next few time units.

---

*A reasonable choice for the quantum q_i (figure 3.5) granted to process i might be $q_i = k\ Q(t_i)$ for some suitable constant $k \geq 1$.

Balance

The computer system is said to be balanced if simultaneously

$$\sum_{\substack{\text{processes in}\\ \text{running list}}} p = \alpha \qquad 0 < \alpha \leq 1 \tag{4.7}$$

$$\sum_{\substack{\text{processes in}\\ \text{running list}}} m = \beta \qquad 0 < \beta \leq 1 \tag{4.8}$$

where p is a processor demand, m a memory demand, and α,β are constants chosen to cause any desired fraction of resource to constitute balance. If the system is balanced, the total demand presented by running list processes just consumes the available fractions of processor and memory resources. If equation 4.7 alone holds we will say that the system is balanced with respect to processor; if equation 4.8 alone holds, we will say it is balanced with respect to memory. We can write equations 4.7 and 4.8 in the more compact form

$$\sum_{\substack{\text{processes in}\\ \text{running list}}} \underline{d} = (\alpha,\beta) \qquad 0 < \alpha,\beta \leq 1 \tag{4.9}$$

where $\underline{d} = (p,m)$ is a demand, defined by equation 4.6. Balance exists when equation 4.9 holds.

Demand and Usage Spaces

To help visualize the operation of balance-seeking allocation policies, it is useful to define two metric spaces*: the demand space D and the usage space U, both having the same metric $\lambda(x,y)$. The purpose of a metric is to enable us to assign a "magnitude" to a demand. Both D and U are two-dimensional spaces; any point in either space can represent a demand (p,m). The metric $\lambda(x,y)$ might be the usual one

$$\lambda\left((p_1,m_1),(p_2,m_2)\right) = \left((p_1-p_2)^2 + (m_1-m_2)^2\right)^{1/2} \tag{4.10}$$

or perhaps

$$\lambda\left((p_1,m_1),(p_2,m_2)\right) = c_1|p_1-p_2| + c_2|m_1-m_2| \tag{4.11}$$

for some positive constants c_1 and c_2. Metric 4.10 might be chosen if it were desirable to interpret D and U as Euclidean spaces; metric 4.11 might be chosen if it were desirable to define differences between demands in terms of equipment costs. To preserve generality, we will deal in an arbitrary metric.

---

*A metric space is a set S of elements and a metric $\lambda(x,y)$ defined for all x,y in S. The metric $\lambda(x,y)$ is a distance measure between points in S, and satisfies these properties for all x,y,z in S:

1. $\lambda(x,y) \geq 0$ and $\lambda(x,y) = 0$ iff $x = y$.
2. $\lambda(x,y) = \lambda(y,x)$
3. $\lambda(x,y) \leq \lambda(x,z) + \lambda(z,y)$

In the space D there is a set of special points, the <u>demand points</u>. Each demand point stands for the demand of some process in the ready list. This set is

(4.12) $\{(p,m) \in D \mid (p,m) = (p_i,m_i),\ i \text{ some process in ready list}\}$

All this says is that the demand space D is regarded as a two-dimensional queue containing points representing processes in the ready list.

In the space U there are two special points. One is the <u>desired demand point</u>

(4.13) $$\underline{u}_0 = (\alpha,\beta)$$

where equation 4.9 defines α and β. The other is the <u>total demand point</u>

(4.14) $$\underline{u} = \sum_{\substack{\text{processes in} \\ \text{running list}}} \underline{d} \qquad \underline{d} = (p,m)$$

$\underline{u}$ is simply the left side of equation 4.9 and $\underline{u}_0$ the right side.

The <u>degree of imbalance</u> Δ is the deviation from balance:

(4.15) $$\Delta = \lambda(\underline{u},\underline{u}_0)$$

The <u>magnitude of a demand</u> $\underline{d}$ is its distance from the origin in D:

(4.16) $$\mu = \lambda(\underline{d},\underline{0}) \qquad \underline{0} = (0,0)$$

All these ideas are illustrated in Figure 4.3.

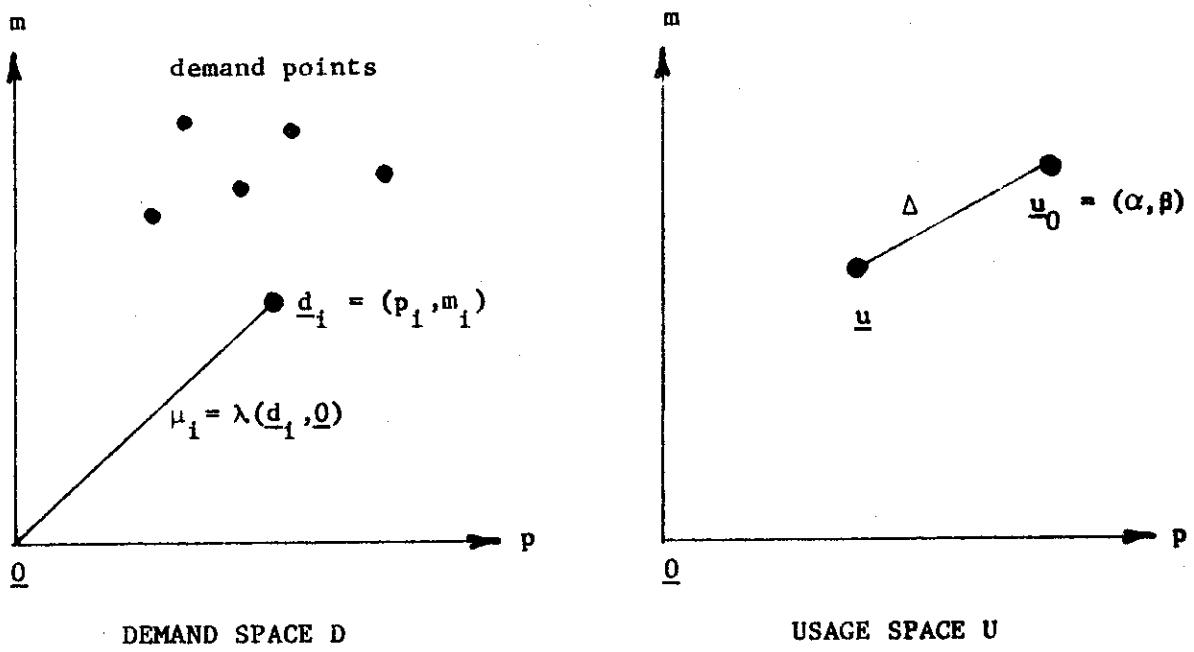


FIGURE 4.3. Demand and Usage Spaces.

Example of a Policy

The remainder of this chapter describes a possible balance-seeking allocation policy. We must emphasize that this is not the only possible policy; it illustrates an approach motivated by the desire for a load-independent, economical strategy.

Basically, the idea of the policy is: whenever the system is below balance ($\underline{u}$ is closer to the origin than $\underline{u}_0$) choose as next for service a process in the ready list whose demand is most nearly ($\underline{u}_0$-$\underline{u}$). That is, whenever $\underline{u} \neq \underline{u}_0$, attempt to move $\underline{u}$ closer to $\underline{u}_0$. Expressed as a minimization problem, the policy is

(4.17) $\{\text{minimize } \lambda(\underline{u},\underline{u}_0)\}$

The checker process (Figure 3.5) plays an important role in the realization of this policy. When run, it updates the demand (p,m) for each process run since the last check, at the same time keeping track of the total running list demand $\underline{u}$ (equation 4.14). The checker admits new processes to the running list according to the policy 4.17.

So simply stated, policy 4.17 does not satisfy one of our most basic requirements: fairness. There is discrimination against large demands. Suppose the typical demand is of magnitude μ_0, so that the running list is populated principally by processes whose demands are of magnitude μ_0. When such a process exits the running state it typically causes the degree of imbalance to become μ_0, causing a demand to magnitude μ_0 to be serviced next. Demands of magnitude much larger than μ_0 will typically receive little service. The remedy for this complicates the policy. The price for fariness will be an increase in the average degree of imbalance*, since it will be necessary to reserve resources in order to make them available to large demands.

---

*Since the total demand $\underline{u}$ is a (time-dependent) random variable, the random variable $\Delta = \lambda(\underline{u},u_\wedge)$ has a (time-dependent) expectation.

The method we have chosen to resolve the problem divides the space D into K shells and guarantees that shell j receives a fraction f_j of resources. The following discussion outlines a procedure for accomplishing this.

Figure 4.4 shows the space D divided into shells whose boundaries are of constant distance (on the metric $\lambda(x,y)$) from the origin; the center of shell j is at a distance μ_j. Corresponding to this, the ready list is divided into K levels, a demand point in shell j being listed (in increasing order of magnitude) in level j.

Define a resource cycle to be a time interval during which the sum of the magnitudes of the demands serviced is some constant ρ. Two resources cycles may not be of equal duration. During each resource cycle shell j is guaranteed to receive $f_j\rho$ units of resource. We want to find integers $n_1, n_2, \ldots, n_K$, where n_j is the number of jobs from shell j to be serviced during a resource cycle to insure its fraction f_j of resource.

Figure 4.4 shows that μ_j can be regarded as typical of the demand magnitude of demand points within shell j. If, during a resource cycle, n_j jobs are serviced from shell j we must have

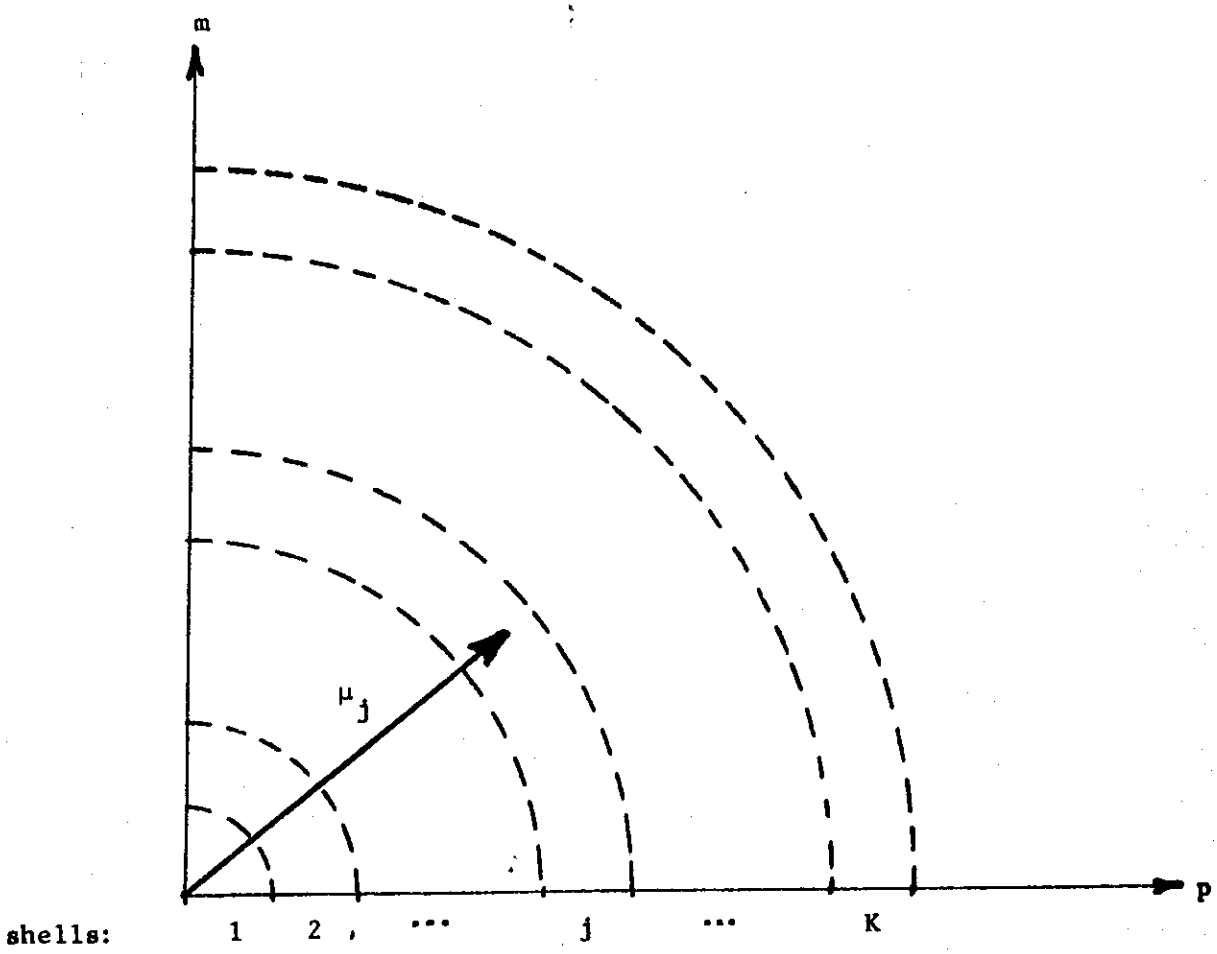
$$f_j = \frac{n_j\mu_j}{n_1\mu_1 + \ldots + n_K\mu_K} \qquad j=1,2,\ldots,K \tag{4.18}$$

however

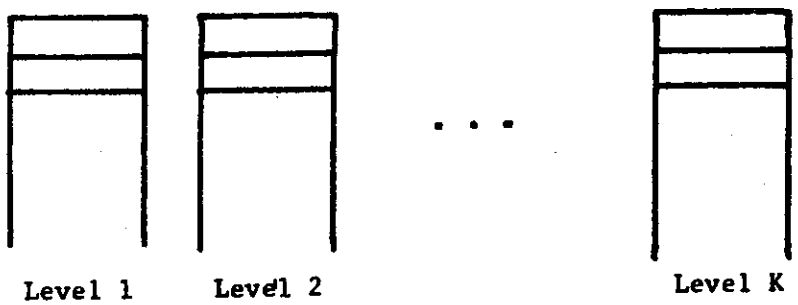
$$\rho = n_1\mu_1 + \ldots + n_K\mu_K \tag{4.19}$$

so

$$n_j = \frac{f_j\,\rho}{\mu_j} \qquad j=1,2,\ldots,K \tag{4.20}$$



DEMAND SPACE D



READY LIST LEVELS

FIGURE 4.4. Shells in Demand Space.

Figure 4.5 shows the algorithm used by the checker to implement policy 4.17. Until run, the checker is in the blocked state; after completing its duties, it re-enters the blocked state. In the diagram there are two blocks deserving comment, labelled A and B. If level ℓ is an "underserviced level", then fewer than n_ℓ jobs have been serviced from it during the current resource cycle; that is, $N_\ell > 0$. Suppose n_j jobs have already been serviced from level j (so that $N_j = 0$); Block A is entered. If there are jobs waiting in a level higher than j it is necessary to reserve resources: the system is left unbalanced so that the next time the checker is run the degree of imbalance may be large enough to accomodate the waiting demand. If there are no such higher-level jobs waiting, a job of smaller demand might as well be serviced, so Block B is entered. A job in an underserviced level ℓ is eligible for service if it satisfies these two requirements:

1. Its demand $\underline{d} = (p,m)$ is not larger than $(\underline{u}_0 - \underline{u})$ [vector inequality] so that if it is serviced, the total demand point $\underline{u}$ is moved closer to, but not beyond, the desired demand point $\underline{u}_0$.
2. Its demand magnitude $\mu = \lambda(\underline{d},\underline{0})$ is the largest of all demands in level ℓ. [If the jobs are ordered by demand magnitude, the required job is the first one in level ℓ satisfying requirement 1.]

The algorithm operates iteratively; that is, additional jobs will be serviced, one by one, until $\underline{u}$ is so close to $\underline{u}_0$ that no new job can be serviced without moving $\underline{u}$ beyond $\underline{u}_0$. Therefore the computation time required to restore balance depends roughly on the degree of imbalance Δ and not on the total demand $\underline{u}$. This means that this balance policy will operate with about the same overhead regardless of the total demand $\underline{u}$.

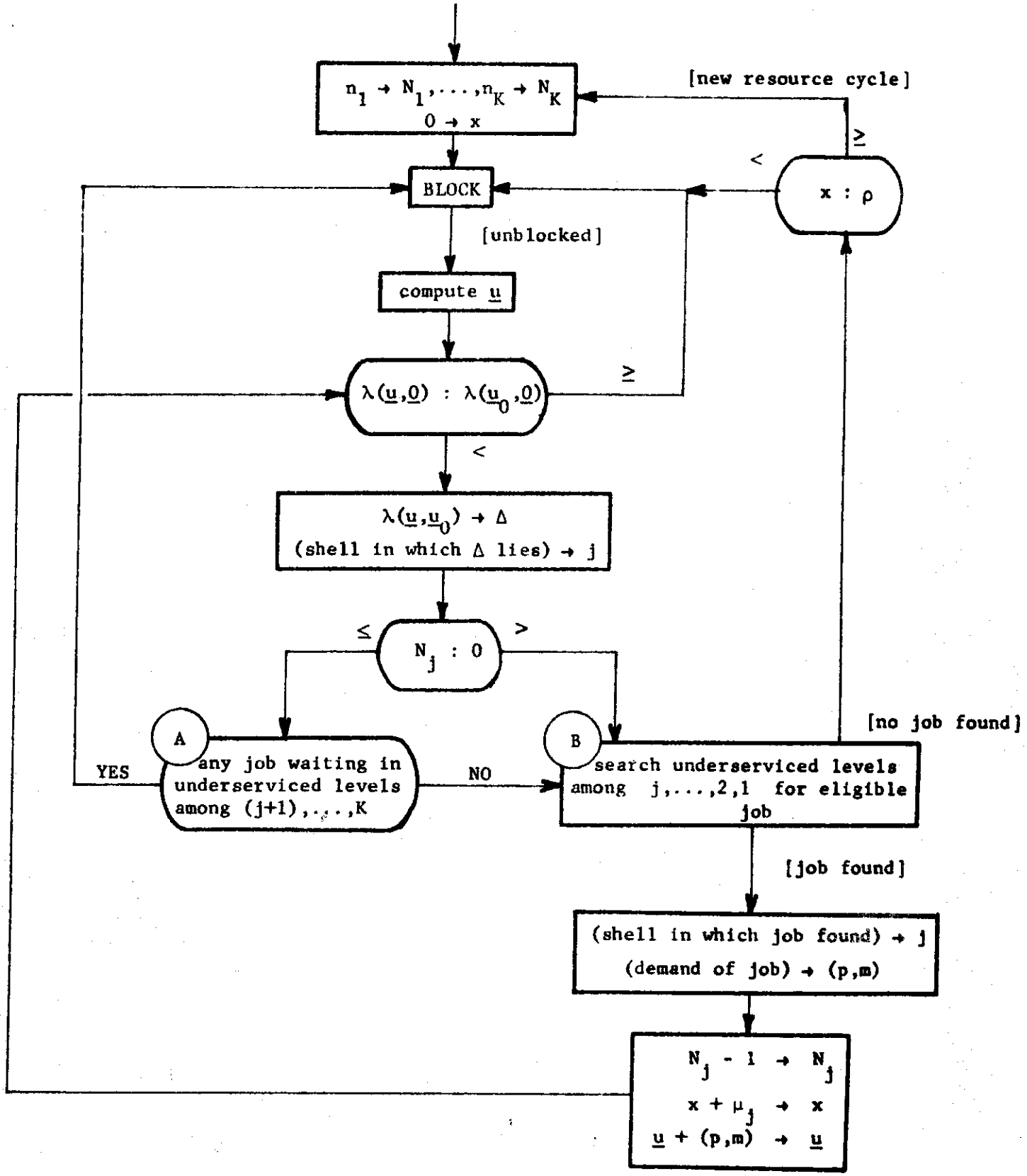


FIGURE 4.5. Balance policy used by Checker Process.

Summary

The demand of a process, its most probable resource usage during the immediate future, has been carefully defined. When the total demand by running processes equals the available resource, balance exists. A "balance policy" strives to allocate resources to maintain balance. Demand and usage spaces were introduced to serve as conceptual aids to understand the operation of balance policies. We described in detail a possible policy, which attempts to move the total demand point toward the desired demand point in usage space. An important feature of this policy is that overhead depends primarily on the degree of imbalance, not on the total demand or the size of the computer system. Another important feature of this policy is that all jobs receive service, not just those having "typical" demands; but the price for fairness is increased aberage imbalance.

CHAPTER V -- FUTURE WORK

It is quite clear that, even with the simplest processor-memory computer system, the allocation problem is far from trivial. This thesis proposal has devoted nearly all its time to isolating issues, clarifying concepts, and making models. The thesis itself will be devoted primarily to analyzing these models. More specifically:

1. We propose to analyze the balanced-computer-system model in order to discover how it looks to a process. To do this, we will have to make reasonable assumptions about demand probability distributions; fortunately there is considerable data available in the literature showing the general form of such distributions. One very promising method of analysis, similar to that used by Scherr[16], is this: it is difficult to model a single demand, not so difficult to model a whole collection of demands. Starting from a few known properties of the whole collection of demands, we can postulate a hypothetical entity, the "virtual demander", which has (in some sense) 1/n of the properties of a collection of n "virtual demanders". No claim can be made that a virtual demander is like any real demander. However we can establish confidence levels describing the accuracy with which a virtual demander models a real one. Using the analysis we can predict such quantities as response times, processing rates and chip consumptions. As described in Chapter I we can also define and predict overload. To date no model of a multi-process computer system has been set forth that can can be "analyzed" by means other than simulation. Such an analysis would constitute a significant contribution.

2. We have yet to turn a perspicacious eye toward allocation problems arising from sharing -- that is, when working sets of different processes overlap. Of particular interest are questions like: Is the allocation problem complicated (or simplified) by sharing? What is the meaning of "memory demand" when it includes shared pages? How are processes to be charged for memory that is shared? Is there anything to be gained by attempting to run two process concurrently when they are sharing information?

3. We have yet to show how the pricing structure -- chips, bidding -- affects allocation decisions. In particular, we shall show how our model may be interfaced with an economic structure such as we have described; this has two aspects:

   A. Description of how user bids are employed, how his account is charged.

   B. Description of how the administration acquires and interprets data on demand and usage.

4. We want to generalize the structure of the computer system of Figure 2.1 where memory has two levels to that of Figure 5.1 where memory has n levels. The main problem here is the traverse time from main to auxiliary memory is no longer simple to determine. It may require generalizations of the notion "working set" and affect the behavior of the allocator with respect to handling page faults. The n-level memory system is expected to become increasingly prevalent in computer systems.

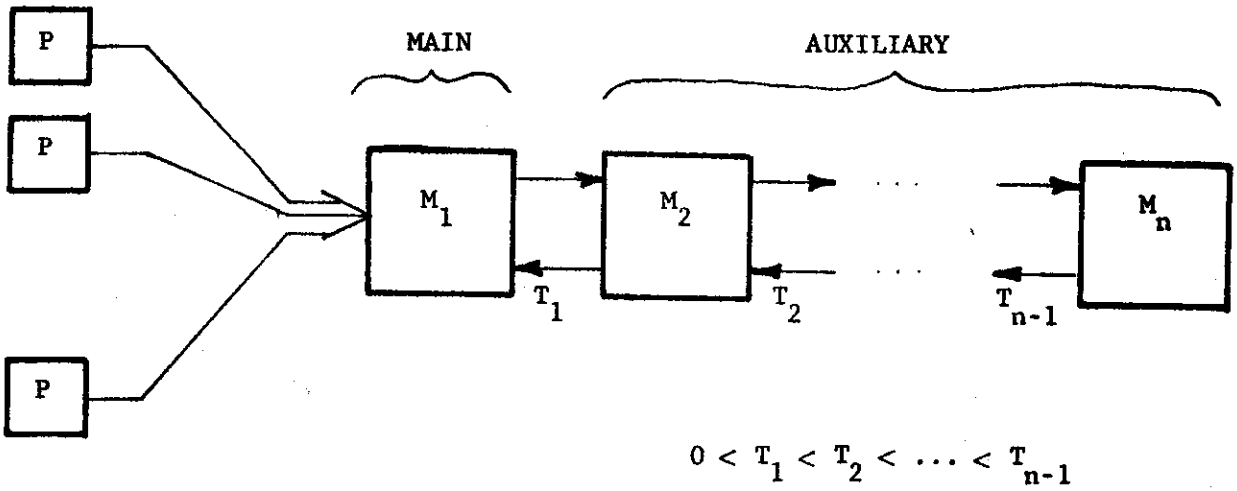


FIGURE 5.1. n-level Memory Hierarchy.

APPENDIX A -- HARDWARE IMPLEMENTATION OF MEMORY MANAGEMENT

Just as hardware is used to streamline the address-mapping mechanism, so too hardware can be used to streamline memory management. The hardware described here associates a timer with each physical page of main memory to measure the working set parameter τ.

Each process, upon creation, is assigned an identification number, i, which is used to index the process table. The i^{th} entry in the process table contains information about the i^{th} process, including its current demand (p_i, m_i). Because this demand information is stored in a common place, the memory hardware can update the memory demand m_i without calling the supervisor. Whenever a page fault occurs, the new page is located in auxiliary memory and transferred to main memory; then a signal is sent to the management hardware to free a page of main memory. The hardware selects a page not in any working set and dispatches it directly to auxiliary memory, without bothering the supervisor. This hardware modifies the page table entry pointing to the newly deleted page, turning the "in-core" bit OFF and leaving a pointer to the new location of the page in auxiliary memory.

Figure A.1 indicates that with each page of memory there is associated a page register, having three fields:

1. π-field: π is a pointer to the memory location of the page table entry pointing to this page. A page table cannot be moved or removed without modifying π.
2. t-field: t is a timer to measure off the interval τ. The value of τ to be used is found in the t-register. The supervisor modifies the contents of the t-register as discussed below.
3. A-field: A is an alarm bit, set to 1 if t runs out.

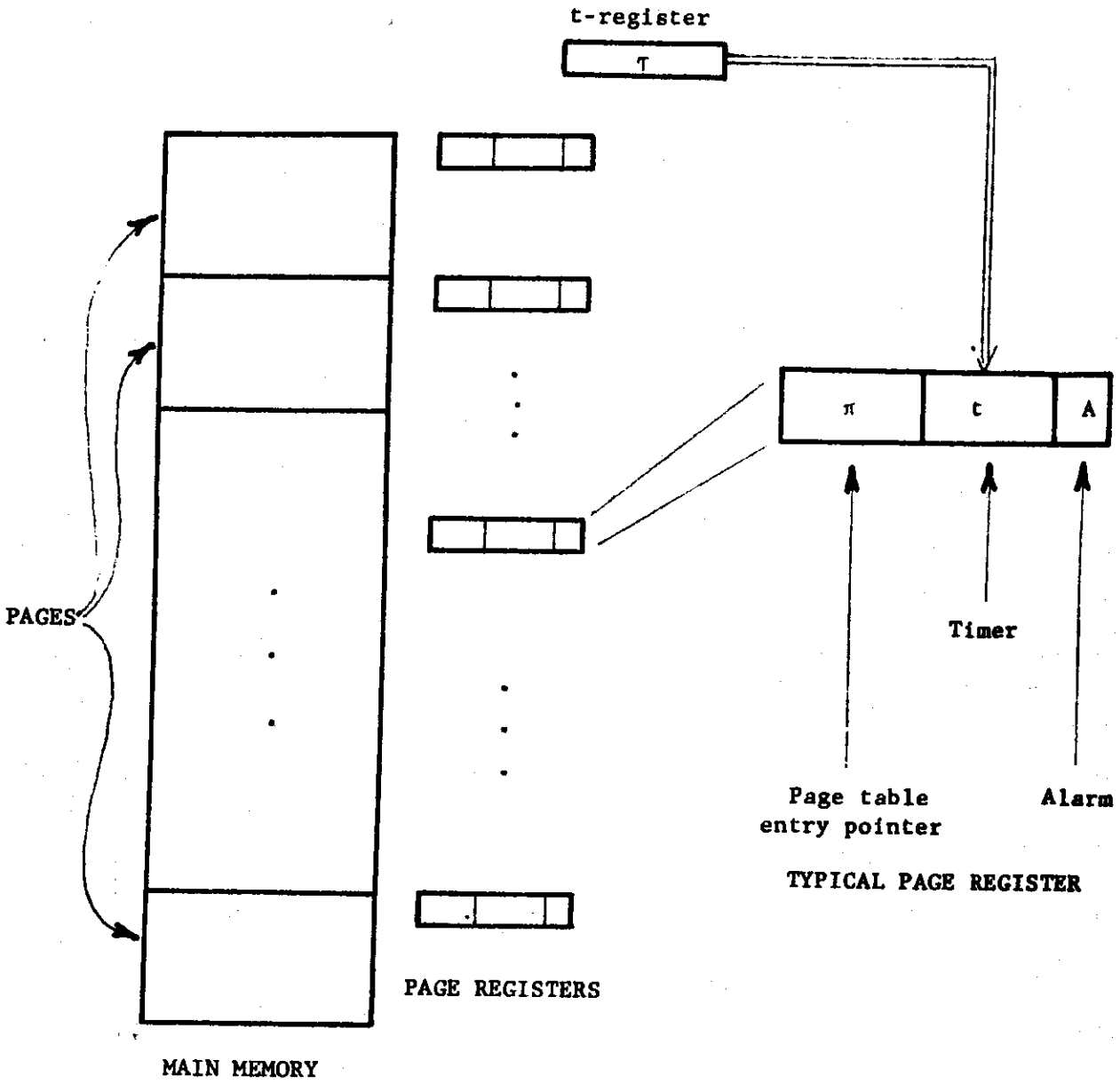


FIGURE A.1. Memory Management Hardware.

Operation proceeds as follows:

1. When a page is loaded into main memory, π is set to point to the correct page table entry. The "in-core" bit of that entry is turned ON.
2. Each time a reference to some location within a page occurs, its page register has $\tau \rightarrow t$ and $0 \rightarrow A$. The timer t begins to run down (in real time), taking τ seconds to do so.
3. If t runs down, $1 \rightarrow A$. Whenever a fresh page of memory is needed, the supervisor sends a signal to additional memory hardware (not shown) which scans pages looking for a page with A=0. Such a page is dispatched directly to auxiliary memory. π is used to find the page table entry, turn the "in-core" bit OFF, and leave information there to permit future retrieval of the page from auxiliary memory. Note that a page need not be removed when A=0; it is only subject to removal. This means that a page may leave and later re-enter a working set without actually leaving main memory.

The timers t are running down in real time. The value in the t-register must be modifiable by the supervisor for the following reason. As in Figure 3.5 the running list is cyclic, except now we suppose that each process is given a burst β of processor time (β need not be related to the sampling interval), and continues to receive bursts β until its running-list quantum is exhausted. If, on a particular cycle, there are n entries in the running list and N processors in service, a given process will be unable to reference any of its pages for about $\frac{n\beta}{N}$ seconds. So the supervisor should be able to set τ to some multiple of $\frac{n\beta}{N}$, for otherwise management hardware will begin removing pages of running processes. However τ should never be less than some multiple of the traverse time T, (Figure 2.1), otherwise when a process interrupts for a page fault its working set may disappear from core memory.

REFERENCES

1. J.B.Dennis and E.C.Van Horn. "Programming Semantics for Multiprogrammed Computations." Comm ACM 9 (March 1966), 143-155.

2. J.B.Dennis. "Segmentation and the Design of Multiprogrammed Computer Systems." JACM 12, 4 (Oct 1965), 589-602.

3. J.H.Saltzer. "Traffic Control in a Multiplexed Computer System." MAC-TR-30, Project MAC, July 1966.

4. R.M.Fano and E.E.David. "On the Social Implications of Accessible Computing." AFIPS Conf. Proc. 27 (Nov 1965). Baltimore: Spartan Books, 243-247.

5. L.L.Selwyn. "The Information Utility." Industrial Management Review 7, 2, Spring 1966.

6. D.Parkhill. The Challenge of the Computer Utility. Addison-Wesley, 1966.

7. C.V.Ramamoorthy. "The Analytic Design of a Dynamic LookAhead and Program Segmenting System for Multiprogrammed Computers." Proc. 21 Nat'l Conf. ACM (1966).

8. B.W.Arden, et al. "Program and Address Structure in a Time-Sharing Environment." JACM 13, 1 (Jan 1966), 1-16.

9. T.Kilburn, et al. "One-Level Stroage System." IRE Trans. on Elec. Computers. Vol EC-11, 2, April 1962.

10. G.H.Fine, P.V.McIssac, C.W.Jackson. "Dynamic Program Behavior Under Paging." Proc. 21 Nat'l Conf. ACM (1966).

11. L.A.Belady. "A Study of Replacement Algorithms for a Virtual-Storage Computer." IBM Systems Journal, Vol 5, 2, 1966. 78-101.

12. P.J.Denning. "Memory Allocation in Multiprogrammed Computers." MIT Project MAC Machine Structures Group Memo No. 24, March 1966.

13. Progress Report III, MIT Project MAC (1965-1966), 63-66.

14. J.B.Dennis. "Program Structure in a Multi-Access Computer." MAC-TR-11, Project MAC.

15. D.W.Fife. "An Optimization Model for Time-Sharing." Proc. 1966 SJCC.

16. A.L.Scherr. "An Analysis of Time-Shared Computer Systems." MAC-TR-18, Project MAC, June 1965.