The Working Set Model for Program Behavior

Peter J. Denning

Massachusetts Institute of Technology, Cambridge, Massachusetts

Probably the most basic reason behind the absence of a general treatment of resource allocation in modern computer systems is an adequate model for program behavior. In this paper a new model, the "working set model," is developed. The working set of pages associated with a process, defined to be the collection of its most recently used pages, provides knowledge vital to the dynamic management of paged memories. "Process" and "working set" are shown to be manifestations of the same ongoing computational activity; then "processor demand" and "memory demand" are defined; and resource allocation is formulated as the problem of balancing demands against available equipment.

KEY WORDS AND PHRASES: general operating system concepts, multiprocessing, multiprogramming, operating systems, program behavior, program models, resource allocation, scheduling, storage allocation

CR CATEGORIES: 4.30, 4.32

1. Introduction

Resource allocation is a tricky business. Recently there has been much dialog about process scheduling and core memory management, yet development of techniques has progressed independently along both these lines. No one will deny that a unified approach is needed. Here we show that it is possible to develop a unified approach. Starting from the observation that every running program places demands jointly on all system resources, particularly processor and memory, we eventually define "system demand"; the allocation problem will consist of balancing demands against available resources.

We regard a *computation* as being the fundamental activity in a computer system; in this paper, a computation consists of a single process together with the information available to it. (For a complete discussion of the meaning of "computation," see Dennis and Van Horn [1].) The usual notion "process" is one manifestation of a computation, in the form of a demand for a processor (a "processor demand"). The notion "working set of information" introduced here is another manifestation of a computation, in the form of a demand for memory (a "memory demand"). A computation's "system demand" will consist jointly of its processor and memory demands.

Probably the most basic reason for the absence of a general treatment of resource allocation is the lack of an adequate model for program behavior. In this paper we develop a new model, the *working set model*, which embodies certain important behavioral properties of computations operating in multiprogrammed environs, enabling us to decide which information is in use by a running program and which is not. We do not intend that the proposed model be considered "final"; rather, we hope to stimulate a new kind of thinking that may be of considerable help in solving many operating system design problems.

The working set is intended to model the behavior of programs in the general purpose computer system, or computer utility. For this reason we assume that the operating system must determine on its own 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, indeterminable. Therefore he cannot be expected to estimate processor-memory needs.

(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 individuals may interfere with overall good service to the community of users. Thus it seems inadvisable at the present time to permit a user, at his discretion, to advise the operating system of his needs.

Likewise, compilers cannot be expected to supply information extracted from the structure of the program regarding resource requirements:[1]

(1) Programs will be modular in construction; information about other modules may be unavailable at compilation time. Because of data dependence there may be no

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tenn., October 1-4, 1967; revised November, 1967. Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Projects Research Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

[1] There have been attempts to do this. Ramamoorthy [2], for example, has put forth a proposal for automatic segmentation of programs during compilation.

way to decide (until run time) just which modules will be included in a computation.

(2) Compilers cluttered with extra machinery to predict memory needs will be slower in operation. Many users are less interested in whether their programs operate efficiently than whether they operate at all, and are therefore 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 we are recommending mechanisms that monitor the behavior of a computation, basing allocation decisions on currently observed characteristics and not on advisories from programmers or compilers. Only a mechanism that oversees the behavior of a program in operation can cope with arbitrary interconnections of arbitrary modules having arbitrary characteristics.

Our treatment proceeds as follows. As background, we define the type of computer system in which our ideas are developed and discuss previous work with problems of memory management. We define the working set model, examine its properties, and then outline a method of implementing memory management based on this model. Finally, we show how "memory demand" is defined by a working set, how "processor demand" is defined by a process, and how resource allocation is a problem of balancing demands against equipment.

2. Background

We assume that the reader is already familiar with the concepts of a computer utility [3–5], of segmentation and paging [1, 6], and of program and addressing structure [1, 7–9]; so we only mention these topics here. Briefly, each process has access to its own private, segmented name space; each segment known to the process is sliced into equal-size units, called pages, to facilitate mapping it into the paged main memory. Associated with each segment is a page table, whose entries point to the segment's pages. An "in-core" bit in each page table entry is turned ON whenever the designated page is present in main memory[2]; an attempt to reference a page whose "in-core" bit is OFF causes a page fault, initiating proceedings to secure the missing page. A process has three states of existence: running, when a processor is assigned to it; ready, when it would be running if only a processor were available; or blocked, when it has no need of a processor (for example, during a page fault or during a console interaction). When talking about processes in execution, we will have to distinguish between "process-time" and "real-time." Process-time is time as seen by a process unaware of suspensions; that is, as if it executed without interruptions.

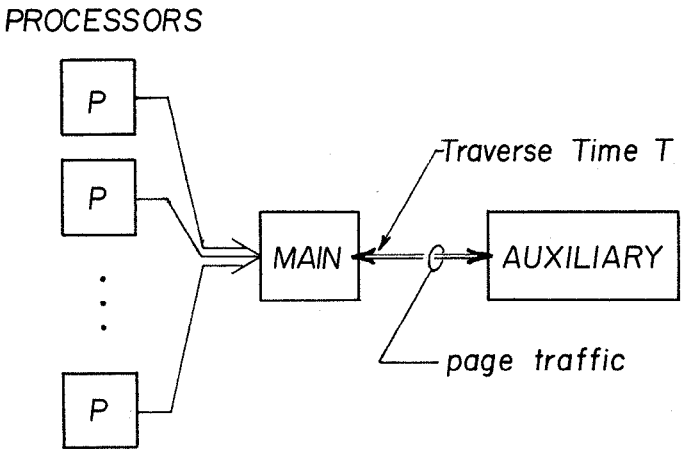

FIG. 1. Two-level memory system

We restrict attention to a two-level memory system, indicated by Figure 1. Only data residing in main memory is accessible to a processor; all other data resides in auxiliary memory, which we regard as having infinite capacity. There is a time T, the *traverse time*, involved in transferring a page between memories, which is measured from the moment a page fault occurs 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 a page is moved.

A basic allocation problem, "core memory management," is that of deciding just which pages are to occupy main memory. The fundamental strategy advocated here—a compromise against a lot of expensive main memory—is to minimize *page traffic*.[3] There are at least three reasons for this:

(1) The more data traffic between the two levels of memory, the more the computational overhead will be deciding just what to move and where to move it.

(2) Because the traverse time T is long compared to a memory cycle, too much data movement can result in congestion on the channel bridging the two memory levels.

(3) Too much data traffic can result in serious interference with processor efficiency on account of auxiliary memory devices "stealing" memory cycles.

Roughly speaking, a working set of pages 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.

PREVIOUS WORK. In this section we review strategies that have been set forth in the past for memory management; the interested reader is referred to the literature for detail.

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

[3] Since data is stored and transmitted by pages, we can (without ambiguity) refer to data movement between memories as "page traffic."

We regard management of paged memories as operating 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. Nearly every strategy pages in on demand; that is, no action is taken to load a page into memory until some process attempts to reference it. There have been few proposals to date 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 preload pages: there is no guarantee that 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 to decide which pages to load, but which pages to *remove*. 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) because erroneously removed pages add to the traffic of returning pages. In the following we use this as a basis of comparison for several strategies.

Random Selection. Whenever a fresh page of memory is needed, a page to be replaced is selected at random. Although utterly simple to implement, this method frequently removes useful pages (which must be recalled), and results therefore in high page traffic.

FIFO (First-In/First-Out) Selection. Whenever a fresh page of memory is needed, the page least recently paged in is retired and another page is brought in to fill the now vacant slot. Implementation is simple. The pages of main memory are ordered in a cyclic list; suppose the M pages of memory are numbered $0, 1, \cdots, (M-1)$ and a pointer k indicates that the kth page was most recently paged in. When a fresh page of memory is needed, $[(k+1) \bmod M] \rightarrow k$, and page k is retired. This method is based on the assumption that programs tend to follow sequences of instructions, so that references in the immediate future will most likely be close to present references. So 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 intermodule calls, will indeed exhibit sequential instruction fetch patterns. The thread of control will not string pages together linearly; rather, it will entwine them intricately. Fine et al. [10] and Varian and Coffman [11] have experimental evidence to support this, namely, that references will be scattered over a large collection of pages. Second, this algorithm is subject to overloading when used in multiprogrammed memories. When core demand is too heavy, one cycle of 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.

Least Recently Used (LRU) Selection. Each page-table entry contains a "use" bit, set to ON each time the page is referenced. At periodic intervals all page-table entries are searched and usage records 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 the simple case of a computer where there is exactly one process whose pages cannot all fit into main memory. In this case a very reasonable choice for a page to replace is the least recently used page. Unfortunately, this method is also susceptible to overloading when many processes compete for main memory.

ATLAS Loop Detection Method. The Ferranti ATLAS computer [12] had proposed a page-turning policy that attempted to detect loop behavior in page reference patterns and then to minimize page traffic by maximizing the time between page transfers, that is, by removing pages not expected to be needed for the longest time. It was only successful for looping programs. Performance was unimpressive for programs exhibiting random reference patterns. Implementation was costly.

Various studies concerning behavior of paging algorithms have appeared. Fine et al. [10] have investigated the effects of demand paging and have seriously questioned whether paging is worthwhile at all. We cannot agree more with their data, nor agree less with their conclusion. Their experiments, as well as those of Varian and Coffman [11], confirm that should there not be enough core memory to contain most of a program, considerable paging activity will interfere with efficiency. The remedy is not to dismiss paging; it is to provide enough core memory! Put another way, there should be enough core memory to contain a program's working set. Paging is no substitute for real core.

Belady [13] 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 FIFO 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 Section 3 we begin investigation of the working set concept. Even though the ideas are not entirely new [9, 14, 15], there has been no detailed documentation publicly available.

3. Working Set Model

From the programmer's standpoint, the working set of information is the smallest collection of information that must be present in main memory to assure efficient execution of his program. We have already stressed that there will be no advance notice 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 pages are 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 information referenced by the process during the process time interval $(t - \tau, t)$.

Thus, the information a process has referenced during the last τ seconds of its execution constitutes its working set (Figure 2). τ will be called the *working set parameter*. We

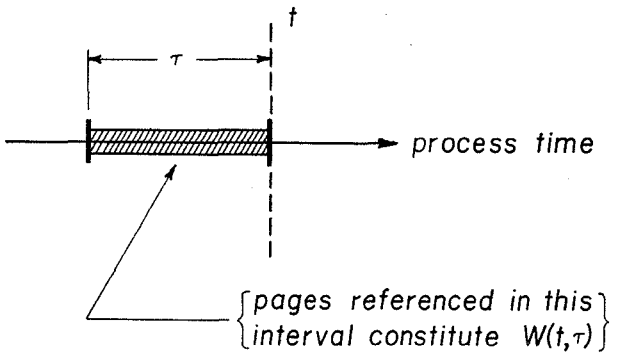

FIG. 2. Definition of $W(t, \tau)$

regard the elements of $W(t, \tau)$ as being pages, though they could just as well be any other named units of information. The *working set size* $\omega(t, \tau)$ is

$$\omega(t, \tau) = \text{number of pages in } W(t, \tau). \quad (1)$$

Let the random variable x denote the process-time interval between successive references to the same page; let $F_x(\alpha) = \Pr[x \leqq \alpha]$ be its distribution function; let $f_x(\alpha) = dF_x(\alpha)/d\alpha$ be its density function; and $\bar{x}$ denote its mean:

$$\bar{x} = \int_0^\infty \alpha f_x(\alpha)\, d\alpha. \quad (2)$$

These interreference intervals x are useful for expressing working set properties.

A working set $W(t, \tau)$ has four important, general properties. All are properties of typical programs and need not hold in special cases. During the following discussion of these properties, assume that $W(t, \tau)$ is continuously in main memory, that its process is never interrupted except for page faults, that a page is removed from main memory the moment it leaves $W(t, \tau)$, and that no two working sets overlap (there is no sharing of information).

P1. *Size.* It should be clear immediately that $\omega(t, 0) = 0$ since no page reference can occur in zero time. It should be equally clear that, as a function of τ, $\omega(t, \tau)$ is monotonically increasing, since more pages can be referenced in longer intervals. $\omega(t, \tau)$ is concave downward. To see this, note that

$$W(t, 2\tau) = W(t, \tau) \cup W(t - \tau, \tau), \quad (3)$$

which implies that

$$\omega(t, 2\tau) \leqq \omega(t, \tau) + \omega(t - \tau, \tau). \quad (4)$$

Assuming statistical regularity, $\omega(t, \tau)$ behaves on the average like $\omega(t - \tau, \tau)$, so that on the average

$$\omega(t, 2\tau) \leqq 2\omega(t, \tau). \quad (5)$$

The general character of $\omega(t, \tau)$ is suggested by the smoothed curve of Figure 3.

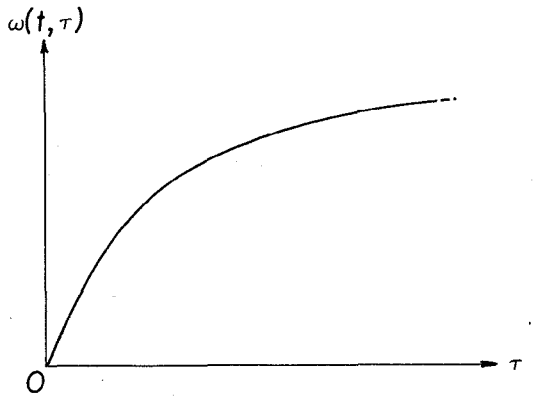

FIG. 3. Behavior of $\omega(t, \tau)$

P2. *Prediction.* We expect intuitively that the immediate past page reference behavior of a program constitutes a good prediction of its immediate future page reference behavior: for small time separations α, the set $W(t, \tau)$ is a good predictor for the set $W(t + \alpha, \tau)$. To see this more clearly, suppose $\alpha < \tau$. Then

$$W(t + \alpha, \tau) = W(t + \alpha, \alpha) \cup W(t, \tau - \alpha). \quad (6)$$

Because references to the same page tend to cluster in time, the probability

$$\Pr[W(t + \alpha, \alpha) \cap W(t, \tau) = \varphi]$$

tends to be small. Therefore some pages of $W(t, \tau)$ will still be in use after time t (i.e. pages in $W(t + \alpha, \alpha)$; since

also

$$W(t, \tau - \alpha) \subseteq W(t, \tau) \cap W(t + \alpha, \tau), \quad (7)$$

$W(t, \tau)$ is a good predictor for $W(t + \alpha, \tau)$. On the other hand, for large time separations α (say, $\alpha \gg \tau$) control will have passed through a great many program modules during the interval $(t, t + \alpha)$, and $W(t, \tau)$ is not a good predictor for $W(t + \alpha, \tau)$.

P3. *Reentry Rate.* As τ is reduced, $\omega(t, \tau)$ decreases, so the probability that useful pages are not in $W(t, \tau)$ increases; correspondingly the rate at which pages are recalled to $W(t, \tau)$ increases. We define two functions: a process-time reentry rate $\lambda(\tau)$ defined so that the mean process time between the instants at which a given page reenters $W(t, \tau)$ is $1/\lambda(\tau)$, and a real-time reentry rate $\varphi(\tau)$ defined so that the mean real-time between the instants at which a given page reenters $W(t, \tau)$ is $1/\varphi(\tau)$.

Let $\{t_n\}_{n \geq 0}$ be a sequence of instants in process time at which successive references to a given page occur. The nth interreference interval is $x_n = t_n - t_{n-1}$; but we are assuming the interreference intervals $\{x_n\}_{n \geq 1}$ are independent, identically distributed random variables, so that for all $n \geq 1$, $f_{x_n}(\alpha) = f_x(\alpha)$. A *reentry point* is a reference instant which finds the page not in $W(t, \tau)$: at such an instant the page reenters $W(t, \tau)$. The reference instant t_n is a reentry point if $x_n > \tau$, independently of other reference instants. Suppose t_0 is a reentry point; we are interested in π_n, the probability that t_n is the first reentry point after t_0. The probabilities $\{\pi_n\}_{n \geq 1}$ are distributed geometrically:

$$\pi_n = \Pr[t_n \text{ first reentry after } t_0] = \zeta^{n-1}(1 - \zeta), \quad (8)$$

where $\zeta = \Pr[x \leq \tau] = F_x(\tau)$. That is, t_n is the first reentry after t_0 if all the instants $\{t_1, \cdots, t_{n-1}\}$ are not reentry instants, and t_n is a reentry. The expected number $\bar{n}$ of reference instants until the first reentry is

$$\bar{n} = \sum_{n=1}^{\infty} n\pi_n = \frac{1}{1 - \zeta}. \quad (9)$$

Each reference interval is of expected length $\bar{x}$ [eq. (2)], so the mean time $m(\tau)$ between reentries is $m(\tau) = \bar{n}\bar{x}$. Therefore

$$m(\tau) = \frac{\bar{x}}{1 - F_x(\tau)}.$$

We define the *reentry rate* $\lambda(\tau)$ to be

$$\lambda(\tau) = \frac{1}{m(\tau)} = \frac{1 - F_x(\tau)}{\bar{x}}, \quad (10)$$

where $\lambda(\tau)$ is the average process-time rate at which one page is reentering $W(t, \tau)$.

Assuming that storage management mechanisms retain in main memory only the pages of $W(t, \tau)$, every page reentering $W(t, \tau)$ must be recalled from auxiliary memory and contributes to page traffic; we here estimate this contribution. In an interval A of process time, the expected number of times a single page reenters $W(t, \tau)$ is $A\lambda(\tau)$; each reentry causes the process to enter a "page-wait" state for one traverse time T, a total of $(A\lambda(\tau)T)$ seconds spent in page wait. Therefore the total real-time spent to recall a page $A\lambda(\tau)$ times is $(A + A\lambda(\tau)T)$. The *return traffic rate* $\varphi(\tau)$ is

$$\varphi(\tau) = \frac{A\lambda(\tau)}{A + A\lambda(\tau)T},$$

that is,

$$\varphi(\tau) = \frac{\lambda(\tau)}{1 + \lambda(\tau)T}, \quad (11)$$

where $\varphi(\tau)$ estimates the average real-time rate at which one page is reentering $W(t, \tau)$. That is, the mean real-time between reentries is $1/\varphi(\tau)$.

Later in the paper we define "memory balance," a condition in which the collection of working sets residing in main memory at any time just consumes some predetermined portion β of the available M pages of main memory. That is, on the average,

$$\sum_{\substack{\text{working sets} \\ \text{in main memory}}} \omega(t, \tau) = \beta M. \quad (12)$$

In this case, the average number of pages in memory belonging to working sets is βM; we define the *total return traffic rate* $\Phi(\tau)$ to be the total reentry rate in real-time to main memory, when the working sets contained therein are not interrupted except for page waits:

$$\Phi(\tau) = \beta M\varphi(\tau) = \frac{\beta M\lambda(\tau)}{1 + \lambda(\tau)T}, \quad (13)$$

where $\Phi(\tau)$ estimates the average number of pages per unit real-time returning to main memory from auxiliary memory. Since "memory balance" is an equilibrium condition, there must also be a flow of pages $\Phi(\tau)$ from main to auxiliary memory. Therefore $2\Phi(\tau)$ measures the capacity required of the channel bridging the two memory levels.

It must be emphasized that the reentry rate functions $\lambda(\tau)$, $\varphi(\tau)$, and $\Phi(\tau)$ are estimates. The important point is: starting from the probability density function $f_x(\alpha)$ for the page interreference intervals x, it is possible to estimate the page traffic which results from the use of working sets for memory allocation.

P4. *τ-Sensitivity.* It is useful to define a sensitivity function $\sigma(\tau)$ that measures how sensitive is the reentry rate $\lambda(\tau)$ to changes in τ. We define the *τ-sensitivity* of a working set $W(t, \tau)$ to be

$$\sigma(\tau) = -\frac{d}{d\tau}\lambda(\tau) = \frac{f_x(\tau)}{\bar{x}}. \quad (14)$$

That is, if τ is decreased by $d\tau$, $\lambda(\tau)$ increases by $\sigma(\tau)\, d\tau$. It is obvious that $\sigma(\tau) \geq 0$; reducing τ can never result in a decrease in the reentry rate $\lambda(\tau)$.

CHOICE OF τ. The value ultimately selected for τ will reflect the working set properties and efficiency require-

ments and will be influenced by system parameters such as core memory size and memory traverse time. Should τ be too small, pages may be removed from main memory while still useful, resulting in a high traffic of returning pages. The return traffic functions $\lambda(\tau)$, $\varphi(\tau)$, and $\Phi(\tau)$, and the τ-sensitivity $\sigma(\tau)$, play roles in determining when τ is "too small." Should τ be too large, pages may remain in main memory long after they were used, resulting in wasted main memory. The desired number of working sets simultaneously to occupy a core memory of given size plays a role in determining when τ is "too large." Thus the value selected for τ will have to represent a compromise between too much page traffic and too much wasted memory space.

The following consideration leads us to recommend for τ a value comparable to the memory traverse time T. Define the *residency* of a page to be the fraction of time it is potentially available in core memory. Assuming that memory allocation procedures balk at removing from main memory any page in a working set, once a page has entered $W(t, \tau)$ it will remain in main memory for at least τ seconds. Letting x be the interreference interval to a given page, we have:

(1) If $x \leqq \tau$, the page will reside in main memory 100 percent of the time.

(2) If $\tau < x \leqq (\tau + T)$, the page will reside in main memory $\tau/(\tau + 2T)$ of the time: it resides in main memory for an interval of τ seconds, after which it is dispatched to auxiliary memory; while in transit it is rereferenced, so it must begin a return trip as soon as it reaches auxiliary memory, a total of two traverse times for the round trip. Therefore the page reappears in main memory $(\tau + 2T)$ seconds after the previous reference.

(3) If $x > (\tau + T)$, the page will reside in main memory $\tau/(x + T)$ of the time: it resides in main memory for an interval of τ seconds, after which it is dispatched to auxiliary memory; sometime after having reached auxiliary memory it is rereferenced, requiring T seconds for the return trip. Therefore the page reappears in main memory $(x + T)$ seconds after the previous reference.

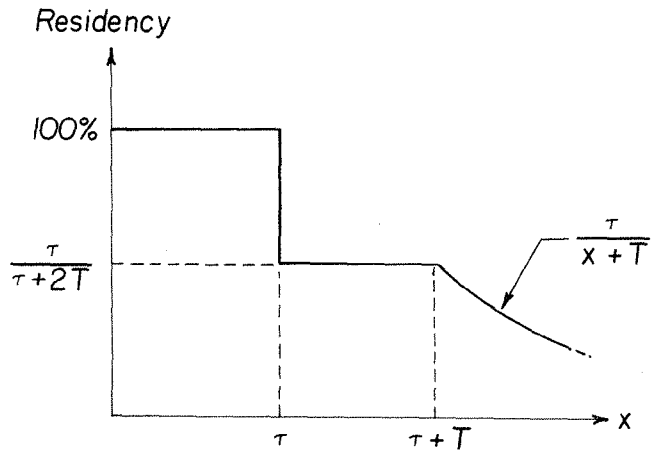

FIG. 4. Residency

Figure 4 shows the residency as a function of x. In the interest of efficiency, it is desirable that the drop from 100 percent to $\tau/(\tau + 2T)$ at $x = \tau$ is not too severe; thus, for example, should we wish to limit the drop to 50 percent, we should have to choose $\tau \approx 2T$.

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 of its execution. This suggests that memory management can be controlled with 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 a hardware memory management mechanism that could be housed within the memory boxes. It 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 intervals.

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

As indicated by Figure 5, 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* $u_0, u_1, \cdots, u_K$. Each time a page reference occurs $1 \rightarrow u_0$. At the end of each sampling interval σ, the bit

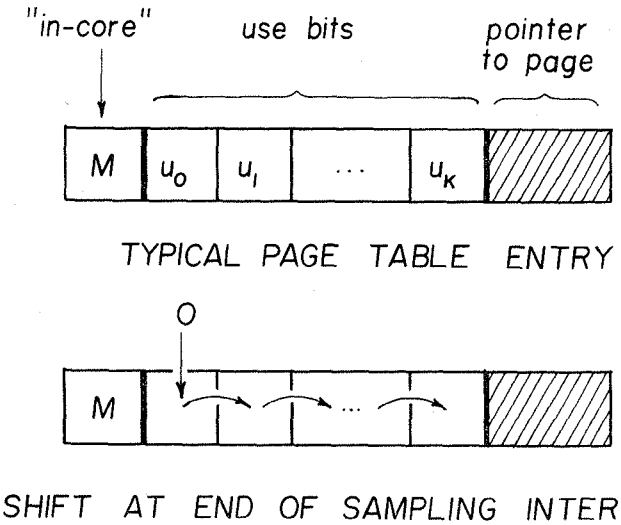

FIG. 5. Page table entries for detecting $W(t, K\sigma)$

pattern contained in u_0, u_1, $\cdots$, u_K is shifted one position, a 0 enters u_0, and u_K is discarded:

$$\begin{aligned} u_{K-1} &\rightarrow u_K \\ &\vdots \\ u_0 &\rightarrow u_1 \\ 0 &\rightarrow u_0 . \end{aligned} \tag{15}$$

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

$$U = u_0 + u_1 + \cdots + u_K , \tag{16}$$

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

MEMORY ALLOCATION. The basic assumption in memory allocation is that a program will not be run unless there is space in memory for its working set.

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

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 eschewed look-ahead: the strong possibility that preloading could be futile. A program organization likely to be typical of interactive, modular programs, shown in Figure 6, 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, $\cdots$, 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 $W(t, \tau)$ is likely to change radically—sometimes only A may be in $W(t, \tau)$, at other times one of the B-procedures and D may be in $W(t, \tau)$. The pages of $W(t, \tau)$ are likely to be different. Thus, that a process blocks for an interaction (not page faults) can be a strong indication of an imminent change in $W(t, \tau)$. Therefore the look-ahead, which is most often used just after a process unblocks, would probably load pages not likely to be used.

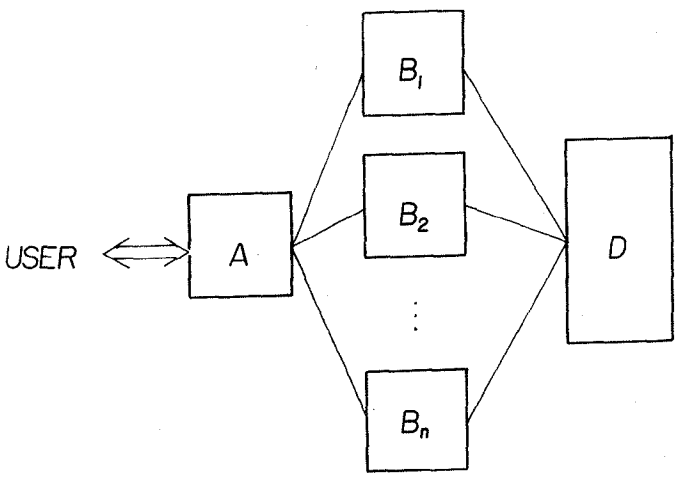

FIG. 6. Organization of a program

Knowledge of only $\omega(t, \tau)$ 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 $W(t, \tau)$, whose pages fill free slots upon demand. By implication, enough free storage has been reserved so that no page of a working set of another program is displaced by a page of $W(t, \tau)$, as can be the case with other page-turning policies. $\omega(t, \tau)$ is a good measure of memory demand.

4. Scheduling

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

If the working set ideas are to contribute to good service, an implementation should at least 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 consumptions for each process.

Figure 7 displays an implementation having these properties. Each solid 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. Let us trace a single process through this system:

(1) When process i is created, an identifier for it is placed in the *ready list*, which lists all the processes in the ready state. Processes are selected from the ready list 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 σ of processor time until it blocks or exceeds 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. 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 eqs. (15) and (16). It should be clear that if the length of the running list is l and there are N processors, sampling of page tables occurs about every $l\sigma/N$ seconds, not every σ seconds.

Associated with process i is a counter w_i giving the current size of its working set. 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 must be 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. This is discussed in more detail below.

5. Sharing

Sharing finds its place naturally.

When pages are shared, working sets will overlap. If Arden's [7] suggestion concerning program structure[4] is followed, sharing of data can be accomplished without modification of the regime of Figure 7. If a page is in at least one working set, the "use bits" in the page table entry will be 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 samples. Allocation policies should tend to run two processes together in time whenever they are sharing information (symptomized by overlap of their working sets), in order to avoid unnecessry reloading of the same information. How processes should be charged for memory usage when their working sets overlap is still an open question, and is under investigation.

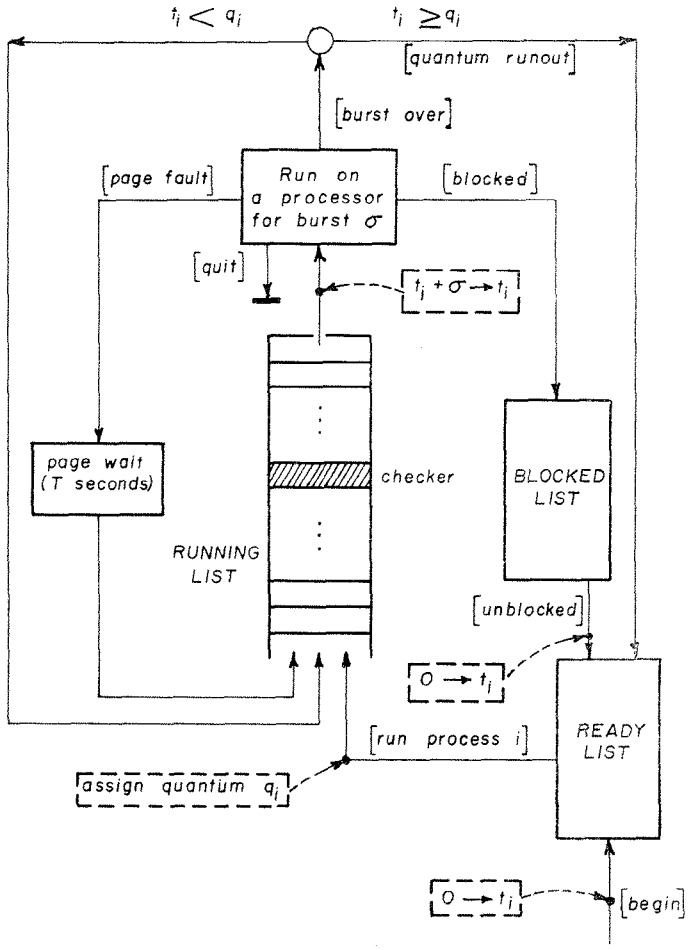

FIG. 7. Implementation of scheduling

6. Resource Allocation: A Balancing Problem

We have already pointed out that a computation places demands jointly on the processor and memory resources of a computer system. A computation's processor demand manifests itself as a process; its memory demand manifests itself as a working set. In this section we show how notions of "demand" can be made precise and how resource allocation can be formulated as a problem of balancing processor and memory demands against available equipment.

DEMAND. Our purpose here is to define "memory demand" and "processor demand," then combine these into the single notion "system demand."

We define the *memory demand* m_i of computation i to be

$$m_i = \min\left(\frac{w_i}{M}, 1\right), \qquad 0 \leq m_i \leq 1, \qquad (17)$$

where M is the number of pages of main memory, and $w_i = \omega_i(t, \tau)$ is the working set count, such as maintained by the scheduler of Figure 7. If a working set contains more than M pages (it is bigger than main 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 more difficult to define. Just as memory demand is in some sense a prediction of memory requirements in the immediate future, so processor demand should be a prediction of processor requirements for the

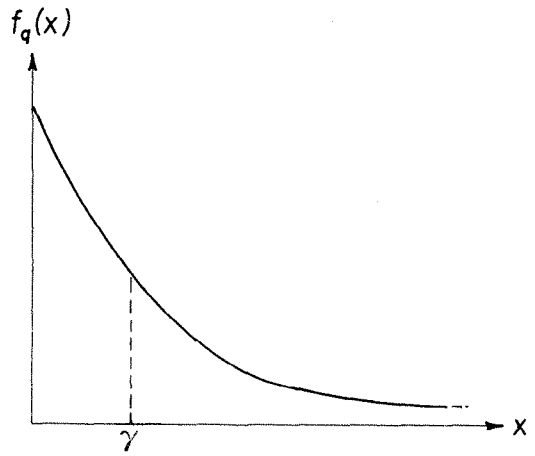

FIG. 8. Probability density function for q

[4] 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.

near future. There are many ways in which processor demand could be defined. The method we have chosen, described below, defines a computation's processor demand to be the fraction of a standard interval a process is expected to use before it blocks.

Let q be the random variable of processor time used by a process *between interactions*. (A process "interacts" when it communicates with something outside its name space, e.g. with a user, or with another process.) In general character, $f_q(z)$, the probability density function for q, is hyper-exponential (for a complete discussion, see Fife [16]):

$$f_q(z) = cae^{-az} + (1 - c)be^{-bz}, \qquad \begin{matrix} 0 < a < b, \\ 0 < c < 1, \end{matrix} \tag{18}$$

where $f_q(z)$ is diagrammed in Figure 8; most of the probability is concentrated toward small q (i.e. frequently interacting processes), but $f_q(z)$ has a long exponential tail.

Given that it has been γ seconds (process time) since the last interaction, the conditional density function for the time beyond γ until the next interaction is

$$\begin{aligned} f_{q|\gamma}(\gamma) &= \frac{f_q(z + \gamma)}{\int_\gamma^\infty f_q(y)\, dy} \\ &= \frac{cae^{-a(z+\gamma)} + (1 - c)be^{-b(z+\gamma)}}{ce^{-a\gamma} + (1 - c)e^{-b\gamma}}, \qquad z \geqq 0, \end{aligned} \tag{19}$$

which is just that portion of $f_q(z)$ for $q \geqq \gamma$ with its area normalized to unity. The conditional expectation of q, given γ, is

$$\begin{aligned} Q(\gamma) &= \int_0^\infty z f_{q|\gamma}(z)\, dz \\ &= \frac{(c/a)e^{-a\gamma} + [(1 - c)/b]e^{-b\gamma}}{ce^{-a\gamma} + (1 - c)e^{-b\gamma}}. \end{aligned} \tag{20}$$

The conditional expectation function $Q(\gamma)$ is shown in Figure 9. It starts at $Q(0) = c/a + [(1 - c)/b]$ and rises toward a constant maximum of $Q(\infty) = 1/a$. 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 consumed by a process since its last interaction, we may expect $Q(\gamma)$ seconds of process time 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.

In order to make a definition of processor demand "reasonable," it is useful to establish symmetry between space and time. Just as we are unwilling to allocate more than M pages of memory, so we may be unwilling to allocate processor time for more than a standard interval A into the future. A can be chosen to reflect the maximum tolerable response time to a user: for if processor time is allocated to some set of processes such that their expected times till interactions total A, no process in that set expects to wait more than A time units before its own interaction.

We define the *processor demand* p_i of computation i to be

$$p_i = \frac{Q(t_i)}{NA}, \qquad \frac{Q(0)}{NA} \leqq p_i \leqq \frac{Q(\infty)}{NA}, \tag{21}$$

where N is the number of processors and t_i is the time-used quantity for process i, such as maintained by the scheduler of Figure 7.

The (system) *demand* $\mathbf{D}_i$ of computation i is a pair

$$\mathbf{D}_i = (p_i, m_i), \tag{22}$$

where p_i is its processor demand [eq. (21)] and m_i its memory demand [eq. (17)].

That the processor demand is p_i tells us to expect computation i to use p_i of the processors for the next A units of execution time, before its next interaction.[5] That the memory demand is m_i tells us to expect computation i to use $(m_i M)$ pages of memory during the immediate future.

BALANCE. Let constants α and β be given. The computer system is said to be balanced if simultaneously

$$\sum_{\substack{\text{processes in} \\ \text{running list}}} p = \alpha, \qquad 0 < \alpha \leqq 1; \tag{23}$$

and

$$\sum_{\substack{\text{processes in} \\ \text{running list}}} m = \beta, \qquad 0 < \beta \leqq 1, \tag{24}$$

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 processes just consumes the available fractions of processor and memory resources. Equation (23) defines "processor balance" and eq. (24) defines "memory balance." We can write eqs. (23) and (24) in the more compact form

$$\mathbf{S} = \sum_{\substack{\text{processes in} \\ \text{running list}}} \mathbf{D} = (\alpha, \beta), \quad \mathbf{D} = (p, m), \tag{25}$$

so that balance exists whenever eq. (25) holds; that is, whenever $\mathbf{S} = (\alpha, \beta)$.

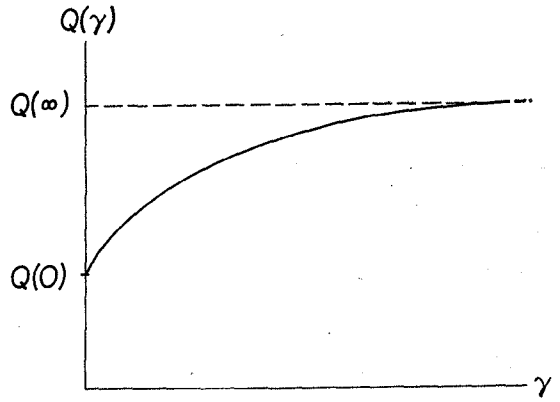

FIG. 9. Conditional expectation function for q

[5] A reasonable choice for the quantum q_i (Fig. 7) granted to computation i might be $q_i = kQ(t_i)$, for some suitable constant $k \geqq 1$.

Whenever the system is balanced, it means that (βM) pages of memory have been committed to running computations, and that (αNA) units of processor time have been committed to running computations.

Dynamic maintenance of $\mathbf{S} = \sum \mathbf{D}$ is straightforward. Whenever a process of demand (p, m) is admitted to the running list, $\mathbf{S} + (p, m) \rightarrow \mathbf{S}$. Whenever a process of demand (p, m) exits the running list (except for page faults), $\mathbf{S} - (p, m) \rightarrow \mathbf{S}$. Therefore $\mathbf{S}$ always measures the current total running list demand.

BALANCE POLICIES. A "balance policy" is a resource allocation policy whose objective is to keep the computer system in balance. Expressed as a minimization problem it is:

$$\{\text{minimize } (\mathbf{S} - (\alpha, \beta))\}. \tag{26}$$

Instead of a priority in the ready list, a process has associated its demand $\mathbf{D}$. In the event of imbalance, the next job (or set of jobs) to leave the ready list should be that whose demand comes closest to restoring balance. [Means of formulating this type of policy are currently under investigation as part of doctoral research into the whole problem of resource allocation.]

We do not wish to venture further here into the alluring problems of allocation policies; our aim is primarily to stimulate new thinking by sowing seeds of ideas. There are, however, three points we want to stress about policy (26):

(1) It is quite clear that system performance is particularly sensitive to overcommitment of memory: when too many working sets occupy main memory, each is displacing another's pages in an attempt to have its own pages present. This phenomenon, known as "thrashing," can easily result in a large number of programs stalled in the page-wait state, implying sticky congestion on the channel to auxiliary memory and serious degradation of service. It is, therefore, highly desirable first to balance memory, then to balance processor. That is, in the event of imbalance, the next job selected from the ready list should be the one that comes closest to restoring memory balance; if there are several jobs available to accomplish this purpose, the tie can be broken by selecting the one that comes closest to restoring processor balance.

(2) The balance criterion is basically an equipment utilization criterion. It is well known that equipment utilization and good response to users are not mutually-aiding criteria. As it stands, policy (26) will tend to favor jobs of small demand and discriminate against jobs of large demand; but with modifications and the methods suggested by Figure 7, together with proper adjustment of the "balance constants" α and β, it is possible to maintain "almost-balance" along with good service to users.

(3) Even with intuitively simple strategies such as balance, the allocation problem is far from trivial—interactions such as those between process and working set, and between balance and good service, are not yet fully understood.

7. Conclusions

Starting from the observation that "process" and "working set" are two manifestations of a computation, we have shown that it is possible to define precise notions of "processor demand," of "memory demand," and of "system demand." Resource allocation is then the problem of "balancing" memory and processor demands against equipment. A "balance policy" strives to maintain balance by judiciously selecting jobs to run. The notions "demand" and "balance" can play important roles in understanding the complex interactions among computer system components. It is quite clear that even with these intuitively simple notions, interactions are exceedingly complex.

In order to arrive at notions of memory demand, we had to define a model for program behavior. The working set model affords a convenient way to determine which information is in use by a computation and which is not; it enables simple determination of memory demands.

It is interesting that Oppenheimer and Weizer [17] have used notions related to "working set" and "memory balance" in their simulations of the RCA Spectra 70/46 Time-Sharing Operating System; their evidence indicates that system performance can be improved markedly through these techniques.

Regarding this paper from a slightly different point of view, we have seen four major contenders for page-turning policies for use in memory management: random, first-in/first-out (FIFO), least recently used (LRU), and working set. Each of these policies pages in on demand, believing that paging out is the heart of the problem, for if pages least likely to be reused in the near future are removed from main memory, the traffic of returning pages is minimized. Random brings on the highest page traffic, working set the lowest. Although Random and FIFO are the easiest to implement, the added cost of working set is more than offset by its accuracy and compatibility with generalized allocation strategies.

Acknowledgment. I thank Jack B. Dennis and Donald R. Slutz for many helpful criticisms.

APPENDIX. Hardware Implementation of Memory Management

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

Each process, upon creation, is assigned an identification number, i, which is used to index the *process table*. The ith entry in the process table contains information about the ith 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 in readiness for the next page fault. The hardware selects a page not in any working set and dispatches it directly to auxiliary memory, without troubling 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 help locate the page in auxiliary memory.

Figure A 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 to be used for τ 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 the timer t runs out. Operation proceeds as follows:

(1) When a page is loaded into main memory, π is set to point to the memory location of 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 is modified: $\tau \rightarrow t$ 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 $A = 1$. 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 $A = 1$; it is only *subject* to removal. This means a page may leave and later reenter 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 7, 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 σ), and continues to receive bursts β until its running-list quantum is exhausted. If on a particular cycle there are n entries in the list and N processors in service, a given process will be unable to reference any of its pages for about $n\beta/N$ seconds, the time to complete a cycle through the queue. That is, one unit of process time elapses for a program about each n units of real-time. So the supervisor should be able to set the contents of the t-register to some multiple of $n\tau$, for otherwise management hardware will begin removing pages of working sets of running processes. However the t-register contents should never be less than some multiple of the traverse time T, for otherwise when a process interrupts for a page fault its working set may disappear from core memory.

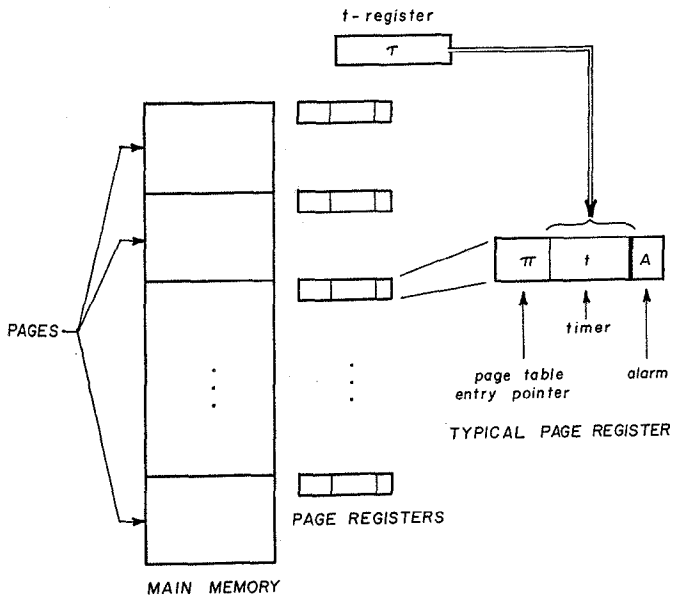

FIG. A. Memory management hardware

REFERENCES

1. DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Comm. ACM 9* (Mar. 1966), 143–155.
2. RAMAMOORTHY, C. V. The analytic design of a dynamic look ahead and program segmenting system for multiprogrammed computers. Proc. ACM 21st Nat. Conf. 1966. Thompson Book Co., Washington, D.C., pp. 229–239.
3. FANO, R. M., AND DAVID, E. E. On the social implications of accessible computing. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 2. Thompson Book Co., Washington, D.C., pp. 243–247.
4. SELWYN, L. L. The information utility. *Indust. Man. Rev. 7*, 2 (spring, 1966).
5. PARKHILL, D. *The Challenge of the Computer Utility*. Addison-Wesley, Reading, Mass., 1966.
6. DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (Oct. 1965), 589–602.
7. ARDEN, B. W., ET AL. Program and address structure in a time-sharing environment. *J. ACM 13*, 1 (Jan. 1966), 1–16.
8. SALTZER, J. H. Traffic control in a multiplexed computer system. M.I.T. Project MAC Tech. Rep. MAC-TR-30, M.I.T., Cambridge, Mass., July 1966.
9. DENNIS, J. B. Program structure in a multi-access computer. Project MAC Tech. Rep. MAC-TR-11, M.I.T., Cambridge, Mass.
10. FINE, G. H., MCISAAC, P. V., AND JACKSON, C. W. Dynamic program behavior under paging. Proc. ACM 21st Nat. Conf. 1966. Thompson Book Co., Washington, D.C., pp. 223–228.
11. VARIAN, L., AND COFFMAN, E. An empirical study of the behavior of programs in a paging environment. Proc. ACM Symp. on Operating Principles, Gatlinburg, Tenn., Oct. 1967.
12. KILBURN, T., ET AL. One-level storage system. *IRE Trans. EC-11*, 2 (Apr. 1962).
13. BELADY, L. A. A study of replacement algorithms for a virtual storage computer. *IBM Systems J. 5*, 2 (1966), 78–101.
14. Progress Report III, M.I.T. Project MAC, 1965–66, pp. 63–66.
15. DENNING, P. J. Memory allocation in multiprogrammed computers. Project MAC Computation Structures Group Memo 24, M.I.T., Cambridge, Mass., Mar. 1966.
16. FIFE, D. W. An optimization model for time-sharing. Proc. AFIPS 1966 Spring Joint Comput. Conf., Vol. 28. Spartan Books, New York, pp. 97–104.
17. OPPENHEIMER, G., AND WEIZER, N. Resource management for a medium scale time-sharing operating system. *Comm. ACM 11*, 5 (May 1968), 313–322.