Obtain from
Room 831

MEMORY ALLOCATION IN MULTIPROGRAMMED COMPUTERS

by

Peter J. Denning

MAC-M-401

Machine Structures Group Memo No. 24

March, 1966

ABSTRACT

The problem of memory allocation in multiprogrammed computer systems is investigated. Optimal Allocation requires judicious decisions about which pages are to be deleted from main memory. A "criterion of goodness", the number of pages transferred per unit time, is used to compare three page turning techniques: (1) delete at random; (2) delete the oldest unused page [this scheme has been suggested for MULTICS]; (3) delete by detecting and predicting loops in programs. Policy (3) is shown to be much superior to either (1) or (2), but appears to involve so much overhead that (2) is preferable. The concept of a "Working Set of Information" is introduced. Allocation by working sets is shown to be easily implementable [even in the MULTICS System], to be as efficient as Scheme (3), and to incorporate automatic look-ahead. Working set size is describeable by a Markov Process: the analysis points out ways in which the system can adjust to the load. The most important result of the study is that an intimate relationship between Scheduling and Allocation has been uncovered.

1. STATEMENT OF THE PROBLEM

In a large-scale multiprogrammed computer system the problem of scheduling the system resources is enormous. At least two scheduling functions are apparent: (1) Scheduling processes in time, making a decision about which process is next to run; and (2) Allocation, a decision concerning which data are to occupy main memory. In present computing systems scheduling has received much attention, and allocation appears to have been considered almost an afterthought. It is not unnatural that scheduling should have received prime attention, because, for example, in the present MAC system allocation has been no problem -- only one user program ever occupied core memory at a time. It is clear that if the scheduling algorithm is to achieve fast average response, there must be discrimination against long jobs. It is for this purpose that the multilevel queue was proposed by Corbato [1]; its implementation has been discussed by Scherr [11], and Dennis [3]. The purpose of a multilevel queue is to "learn" the size of a job by observing how much processor time a job has consumed. The job is given a priority inversely proportional to the time it has consumed, so that it gets less and less service as time goes on. In this way short jobs, which are in the majority, get preference and the mean response time is smaller.

With the advent of multiprogrammed computer systems ("MCS" for short) it is clear that allocation is no longer of secondary importance. Indeed it is at least equally as important as scheduling. For example, what good is a scheduler if it schedules processes whose working information is not present in main memory? Such processes will be continually delayed waiting for the required information to be transferred in to main memory; if there is much of this lack of coordination in system activity, the service of the system will be degraded seriously. Thus it is apparent that there is some basic connection, or <u>interface</u>, that must exist between the scheduling and allocation functions. One of the purposes of this paper is understanding this interface. We will show that the notion of a "Working Set of Information" leads to such an understanding.

Present allocation schemes can be classed as "page turning algorithms". For practical reasons all of main memory and secondary memory (discs, drums tapes) is divided into blocks of a standard size (for example 64 or 1024 words), called <u>pages</u>. The page acts as a unit of transmission and storage. Page turning algorithms attempt to decide judiciously what pages should be removed from main memory to make way for new pages. If the choice is made properly, a minimum number of transfers will take place, and system efficiency will tend to be maximized.

In order that a page turning algorithm make "intelligent" decisions, it must look for patterns in program behavior. Just what pattern of behavior to seek is dependent upon what model for the program is assumed.

It has been proposed that one might think of the page-use behavior of a program as a Markov Process, in the following manner. The pages are numbered 1,2,...,N. Let the state of the system be described by which page was last used, either for execution or for reference. Then there will be transition probabilities $p_{jk}(t)$ from the j^{th} to the k^{th} page at time t, and the times between transitions will occur with some inter-transition time density function p_t. The trouble with this model is that it assumes that whether or not page k is next to be used depends only on whether page j is presently in use. One can think of examples in which this is not the case (for example, a big loop throught the N pages in succession), and hence this model does not appear to be realistic. Furthermore it is not clear how to determine the transition probabilities, which may vary widely from program to program. About the best one might model with a Markov model is the number of pages presently in use by a process, but beyond that it is difficult to say much (such as exactly which page is being referenced). The conclusion is there is no relatively simple, yet general enough model for program behavior which can be stated a priori. Therefore we must require the page turning algorithm to detect patterns of behavior for itself, then take action appropriate for the situation at hand.

In this paper, then, we will survey and analyze present page turning techniques, and show that they are generally unsatisfactory because they detect the wrong kind of behavior;

then we briefly consider pattern detection as a "signal in noise" problem, and show that this too is unsatisfactory, they we introduce the concept of a "Working Set of Information" and demonstrate that it leads to a most satisfactory method of controlling allocation; finally we consider the implications of this method, how it implies an interface between scheduling and allocation, and what it may lead to in the future.

II. SURVEY OF PAGE TURNING TECHNIQUES

A segment is a unit of naming, an ordered group of words. A page is the unit of storage and transmission. A segment can be composed of many pages. A segment will be defined by the user or one of his processes, while a page will be defined by the system when it stores the segment. Hence the notion of "segment turning" might occur to us in addition to that of "page turning"; segment turning would mean that allocation decisions would be made on a segment basis, rather than a page-by-page basis. However, since the system hardware deals with pages, it is natural to think of page turning rather than segment turning. Moreover, since segments are groups of pages, segment turning is nothing more that a special case of page turning. Therefore we will discuss only page turning here.

A Criterion of Goodness: The Swap Rate

If we are to compare various page turning schemes we will need some criterion on which to base the results. Suppose that an unlimited supply of main memory were available. Under such circumstances there would be no need for swapping or for secondary storage. If the main memory is finite in extent, then certain pages must be removed temporarily to make way for the creation of new pages or the return of previously displaced pages. It is useful to define an exit rate E and a return rate R, where E is the number of pages per unit time being removed from main memory to allow for other pages, and R is the number of pages per unit time returning because they are needed again. Pages which are never to be used again may simply be deleted from existence, and therefore will not be a component of E. Similarly pages which appear in the system for the first time will be created in main memory by some process and will not be a component of R. Therefore E and R are rates depending to some degree on the rate at which processes are creating or reusing data, and to a very large degree on the policy used to select pages for removal from main memory. For example a policy which consistently chooses for removal pages which are about to be referenced will cause R to be high, whereas a policy which chooses for removal pages unlikely to be referenced soon will result in low R. In a steady state situation, E = R, and consequently the number of transfers per unit time, or <u>swap</u> <u>rate</u> S, is given by

$$S = E + R = 2R .$$

Therefore an efficient page turning scheme will minimize the number of transfers per unit time; that is, it will <u>maximize the time between transfers</u>. We will use this as the criterion of goodness.

<u>Description of Three Page Turning Methods</u>

We will discuss three algorithms for page turning. When it is time to bring a new page into main memory a choice must be made to <u>delete</u> an existing page from main memory (that is, move it to secondary memory). The deletion schemes we consider are:

(1) choose the page to be deleted at <u>random</u>.

(2) choose the page to be deleted to be the one which has been out of use the longest; we will call this the oldest unused method.

(3) look for cyclic behavior of page use, and make decisions based on prediction of reuse. Since this scheme was proposed for use on the ATLAS computer we will call it the <u>Atlas</u> scheme. [8]

The philosophy behind a random deletion policy would be simply that no computational overhead would be expended on a decision, and hence its lack of elegance would be offset by its simplicity.

The thinking behind the oldest unused policy is that a processor tends to take its next instruction from the same region of code. Thus if a page was just referenced the probability is higher that it will be referenced next rather than another page. Under this assumption, the longer a page is unused, the less is the probability it will be used again. Therefore it is a wiser decision to choose the oldest unused page. Unfortunately it is not clear whether this scheme is superior to a random scheme because of the decision-making computation that must be carried out. However, it is possible to implement simple hardware which keeps track of the oldest unused page; then the decision would not have to be made by software, and this policy would be superior to a random policy.

The thinking behind the Atlas policy is as follows. Programs (apparently) spend a lot of time in loops. Therefore

If the loops can be detected, wise deletion decisions can be made. For if a program is in a cycle, pages will be referenced at regular intervals (on a program time scale). Therefore the periods of time between references will be identical. Let the length of the last period of inactivity of some page be T. Let the length of the present inactive period be τ. See Figure 1. The rules for deletion are: delete the page for which

$$\begin{array}{lll} & (1) & \tau > T + 1 \quad \text{if } T \neq 0; \\ \text{or else} & (2) & (T - \tau) \text{ is a maximum, if some } \tau \neq 0; \\ \text{or else} & (3) & T \text{ is a maximum, if all } \tau = 0. \end{array}$$

T is set to zero when a page is first loaded into memory, and is saved whenever a page is swapped out of memory. Rule 1 will throw out a page that has fallen out of use. Rule 2 will be used if it appears that no page has yet fallen out of use, and will choose the page which is least likely to be used again in the immediate future. Rule 3 is used if every page is active, and no good decision can be made; choosing T_{max} will cause T to be reset to zero and ensure that the same mistake will not be made again. The policy requires that records be kept of active and inactive periods; frequent interupts will be needed to keep the data up to date. Furthermore, the act of decision clearly requires a good deal of computation, which might be so prohibitive as to render the Atlas scheme useless. A more complete discussion of its drawbacks will be given shortly after we have analyzed the three methods.

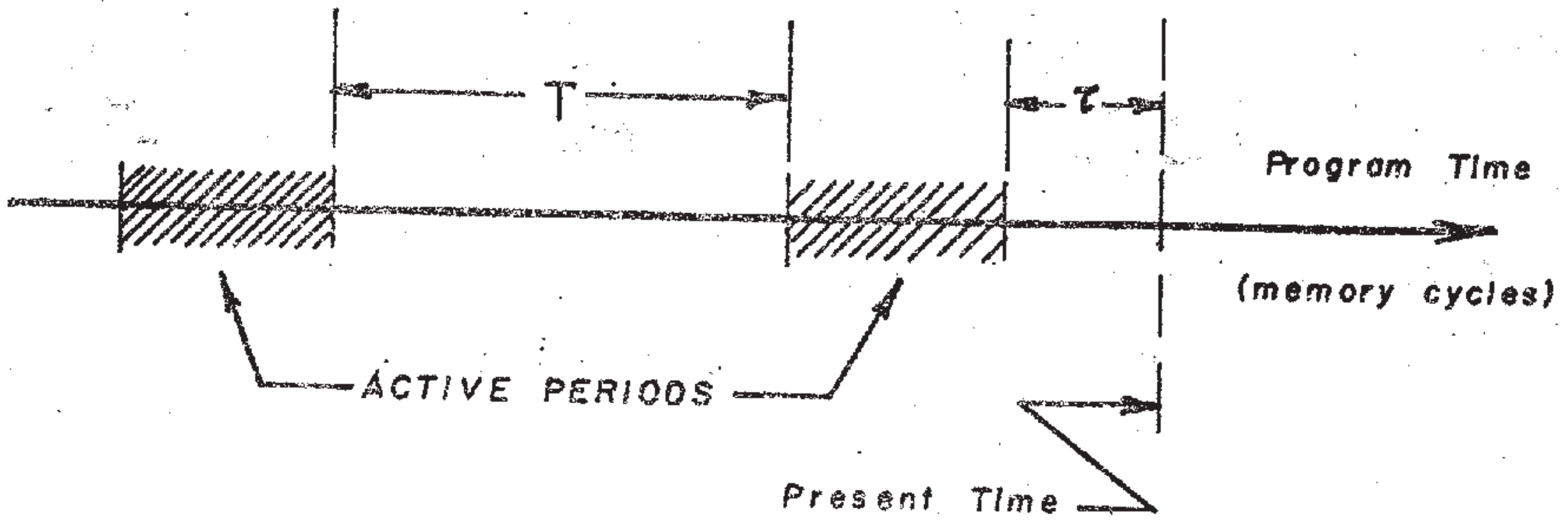

FIGURE I. History of page use.

Analysis of the Three Methods

We suppose that a certain MCS has a main memory of capacity K pages, which are in use by some number of processes. The i th page, after undergoing an active period, will be used again with probability p_o, in which case it enters a period of inactivity T_i. For a given page, it is assumed that all the periods of inactivity are of the same length. For the ensemble of K pages there will be an ensemble of values $[T_i]$. Thus we can define a random variable T on the ensemble of pages, taking on the values $T_1, T_2, \ldots, T_K$. Assuming that K is a large number, we can make the approximation that T is a continuous random variable. For the purposes of this discussion we will assume that the density function for T is

$$(1) \qquad f_T(t) = c\, e^{-ct} \qquad t \geq 0$$

It is important to note that this density function is in reality a conditional density function, conditional on the fact that a page is to be reused. The distribution function for T is

$$(2) \qquad F_T(t) = Pr[T \leq t] = 1 - e^{-ct} \qquad t \geq 0$$

and $E[T] = 1/c$. We suppose further that there is a probability of reuse associated with each page. Since a processor tends to take its next instruction from the same region of code, it seems reasonable to assume that the probability of reuse should depend on the actual period of inactivity, decreasing as the period becomes longer. For the purposes of this discussion we let

$$(3) \qquad Pr[\text{reuse}] = p_o = Pr[T > t/a] = e^{-(c/a)t}$$

where t is the present length of the inactive period, and a is a scale factor.

As we have mentioned previously, we can define an exit rate E of pages per unit time from main memory, and a return rate R of pages per unit time into main memory. The total number of transfers per unit time, or swap rate S, is given in the steady state by

$$S = E + R = 2R. \qquad (4)$$

S will be greatly dependent on the page-turning policy. It is necessary to choose the policy for which S is minimized.

We start with the random case. When a deletion occurs, the expected time until the page is reused is just $E[T] = 1/c$, since T is an exponentially distributed random variable. Due to the symmetry of the exponential distribution, the expected time back to the beginning of the inactive period is also 1/c;* therefore the probability of reuse is

$$p_o = e^{-1/a} \qquad (5)$$

Consequently the return rate for this policy is

$$R_1 = p_o \frac{1}{E[T]} = c\, e^{-1/a} . \qquad (6)$$

*See Feller [5], p. 10 ff.

Next we consider the "oldest unused" policy. There will be some subset N of the K pages inactive at deletion time, on which a decision will be made. In choosing the oldest, this policy is maximizing a set of exponential random variables. From Appendix 1, the expected value of this maximum (that is, the expected age of the page actually chosen for deletion) is approximated by (ln N)/c. Putting this into equation (3) we have for the probability of reuse:

(7) $$p_o = \exp\left[- \frac{c}{a} \frac{\ln N}{c} \right] = \left(\frac{1}{N}\right)^{1/a}$$

Since the expected time till reuse is still $E[T] = 1/c$, it follows that R for this policy is

(8) $$R_2 = c\, p_o = c \left(\frac{1}{N}\right)^{1/a}$$

Finally we consider the Atlas case. The average length of a cycle is $E[T] = 1/c$. Hence with probability $(1 - p_0)$, the probability of not reuse, Rule 1 is applied (assuming that the correct decision is made), and Rule 2 is applied with probability of reuse

$$p_o = e^{-1/a}$$

Because Rule 2 is maximizing a set of exponential random variables (time till reuse), by Appendix 1 the time till reuse is approximately (ln N)/c, where N is number of pages involved in the decision. Hence the return rate under the Atlas policy is

$$R_3 = \frac{c}{\ln N} [\text{probability of Rule 2}][\,p_o\,]$$

$$R_3 = \frac{c}{\ln N} (p_o)^2$$

$$= \frac{c}{\ln N} (e^{-1/a})^2$$

(9) $$R_3 = \frac{c}{\ln N} e^{-2/a}$$

A Numerical Example

We suppose that time is measured in memory cycles. Suppose also that pages are 64 words in length. Then we expect that if no reference has been made for 64 memory cycles that this page is not very likely to be used again, yet there is still a good possibility. Therefore, a conservative estimate is that after 64 memory cycles the probability of reuse is e^{-1}. From equation (3) this gives

(10) $$p_o = e^{-1} = e^{-t/a}\Big]_{t=64}$$

Hence we set $a = 2^6$. Suppose also that the average cycle time of a page's inactive period is 32 memory cycles, so that $c = 2^{-5}$ per memory cycle. Suppose there are 50 inactive pages at the time when a deletion is to be made, so that N = 50. Then from Appendix 1, $E[T_{max}(50)] = \frac{4.5}{c}$. This gives

(11) $$R_1 = c\ e^{-1/a} = \frac{1}{32} e^{-1/64} = 3.1 \times 10^{-2} \text{ / cycle}$$

(12) $$R_2 = c\ (\tfrac{1}{N})^{1/a} = \frac{1}{32} e^{-(4.5/64)} = 2.9 \times 10^{-2} \text{ / cycle}$$

(13) $$R_3 = \frac{c}{\ln N} e^{-2/a} = \frac{1}{32}\frac{1}{4.5} e^{-1/32} = 0.7 \times 10^{-2} \text{ / cycle}$$

Therefore the Atlas method is superior, but apparently not by as much as we might have expected for all the trouble it went to. We will show that in the more general case, when conditions are not as stylized as in this example, that Atlas can be far superior.

The alert reader should be concerned with the fact that R_2 decays as $\frac{1}{N}$, while R_3 decays as $\frac{1}{\ln N}$, so that for large enough N, the Altas scheme will be inferior. We will show now that R_3 is lower than R_2 for all N of interest, by finding the value of N for which $R_2 = R_3$; that is, the value of N for which

$$\frac{R_3}{R_2} = 1.$$

Consider

(14) $$\lim_{N \to \infty} \frac{R_3}{R_2} = \lim_{N \to \infty} \frac{c}{\ln N} e^{-2/a} \frac{1}{c} N^{1/a}$$

using L'Hopital's Rule,

(15) $$= e^{-2/a} \lim_{N \to \infty} \frac{\frac{1}{a} N^{(1/a - 1)}}{N^{-1}}$$

so that for large N, $\frac{R_3}{R_2}$ behaves as

$$\frac{1}{a} e^{-2/a} N^{1/a}$$

setting this equal to 1, we can solve for N:

$$\frac{R_3}{R_2} = 1 = \frac{1}{a} e^{-2/a} N^{1/a}$$

(16) $$N = (a\ e^{2/a})^a$$

For the above example a = 64 so that

(17) $$N = (64)^{64} e^2 \approx 10^{117}$$

Clearly no memory of such size will be built in the immediate future. Figure 2 illustrates the relative behavior of R_1, R_2, and R_3 with N. It is important to note that R_2 and R_3 diminish with increasing N, while R_1 remains constant. It is also noteworthy that although 10^{117} is the actual value of N giving equality of R_2 and R_3, that these two rates will be closely equal for N considerably less than 10^{117}.

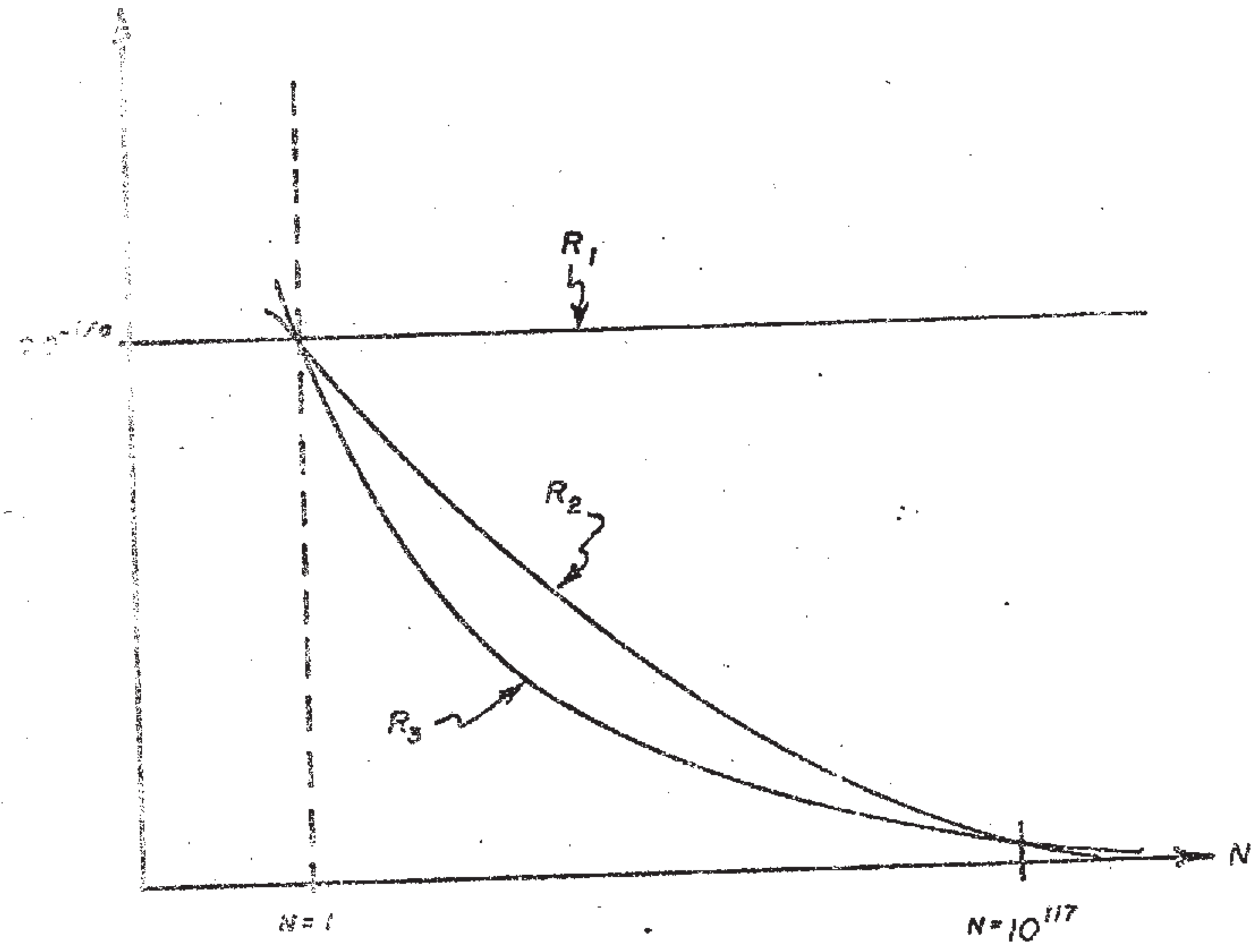

FIGURE 2. Relative behavior of the three schemes.

Discussion of Results

Admittedly the assumptions and numbers used in the analysis are not realistic. But they were chosen to serve as a basis on which to prove a point. If we knew what the use-patterns were we could do a good job of minimizing swaps. A more general and more realistic formulation of the porblem is as follows. Let $[T_i, i=1,2,\ldots,N]$ be the set of inactive intervals for N pages where $T_1 < T_2 < \ldots < T_N$. The random policy will on the average cause the expected time until reuse, $E[T]$, to be

(18) $$E_1[T] = \frac{1}{2} \frac{1}{N} \sum_{i=1}^{N} T_i$$

because the deletion may occur anywhere in the intervals $[0 < T \leq T_i, i=1,2,\ldots,N]$. The oldest unused policy will produce the same result, but now it is adjusted by the probability of reuse:

(19) $$E_2[T] = \frac{1}{2} \frac{1}{N} \frac{1}{p_o} \sum_{i=1}^{N} T_i$$

On the other hand, the Atlas policy explicitly seeks to maximize the expected interval until reuse. Appendix 2 shows that for a "typical" set $[T_i]$, $E[T_{max}]$ is very likely to be

(20) $$E[T_{max}] \approx 5 \frac{1}{N} \sum_{i=1}^{N} T_i$$

Thus it is very likely that the Atlas inter-swap interval will be

(21) $$E_3[T] \approx \frac{5}{p_o} \frac{1}{N} \sum_{i=1}^{N} T_i \geq 10\ E_2[T]$$

Consequently the swap rate will be considerably lower for this policy. In the example given above the assumption of exponential inactive intervals tended to de-emphasize the spread of values, since large values are unlikely. This made the numerical results tend closer together. In the more realistic situation described here we expect Atlas to be far superior by at least an order of magnitude, and the oldest unused to be perhaps half an order of magnitude better than the random policy.

Hardware Implementations

We have already pointed out that a simple hardware scheme for implementing the oldest unused policy is available, requiring a minimum of computational overhead. The scheme is simply to have an array of bits, one for each page of main memory. Wherever a page reference occurs, the corresponding bit is set. Every T seconds the bits are read and reset by a Supervior program. This program will generate a list ordering the pages, starting with the oldest first, which can be consulted by the swapping program when needed. The computation time for such a scheme is minimal and would probably be worth the saving over the random policy. Such a scheme is planned for use in the Multics System [7,12].

With the Atlas policy, matters are no longer simple. There must be an interupt every T seconds for a checkup of the hardware storage map of page use. Then records of presently inactive pages and their inactive times must be compiled. When a deletion is to occur, possibly three searches of the information must be performed to test each rule; these searches could be long if the memory is large. More than likely Rule 2 will be applied most of the time, so that two searches must be made on the average. Kilburn et al [8] mention that in the ATLAS computer, a single processor system, the required computation was performed during the swap time, while the main program was unable to run. This computation took most of the 2 msec swap time. In a multiprogrammed system it would be more important to give the processor over to tasks other than examining page-use tables. If, every swap time, 2 msec was consumed deciding what page was next, the system service would be degraded seriously. A further difficulty with this policy is that measurements must be made in program time, not in real time. In a multiprogrammed system there is not a one-one correspondence between real time and program time, so that special provisions would have to be made to secure measurements only on pages belonging to running processes.

The conclusion is simple. Of the three page turning policies of interest, only the oldest unused policy seems acceptable for use in an MCS. Nevertheless, the knowledge that a much better method exists is disturbing. There is one other conclusion that is important, namely that to minimize swaps it is necessary to maximize the expected time until the deleted page is reused. We will use this idea later on in a scheme which we believe is superior to any of the three page turning schemes discussed.

One final point: if as many pages as possible are written in pure procedure form (that is, execution of their code in no way modifies their contents) then a large number of transfers out of memory can be eliminated. Such pages only need to be read in.

[illegible] A PREDICTION PROGRAM?

It had been proposed in connection with the ATLAS system [8] that the program which detected the use-patterns be modified to predict when pages would be reused. Thus in the event of a deletion, the page would be scheduled for swap-in at some future time, so that it would be available when it was to be needed again. However no action had been taken on this idea. We mention it here to point out that although appearing reasonable, it will be more trouble than it is worth. In order to implement it, at least the previous <u>two</u> periods of inactivity would have to be recorded. Call these T_1 and T_2. Then if

(22) $$|T_1 - T_2| < \delta$$

for some small integer δ (T_1 and T_2 are numbers of memory cycles) the page would be deemed to be in a cycle, and a "prediction bit" in the main memory storage map would be set. If that page were then selected for deletion, the time until reuse would be computed, and the page number, together with the scheduled time until swap-in, would be entered in a swap job list.

It is clear that if the record-keeping associated with just one previous period of inactivity involves much overhead, then keeping the additional records and making the additional decisions, and generating the job lists, would be even more expensive. Only if the record-keeping and decision-making were relegated to hardware could the scheme hope to be practical. The idea of building special complicated hardware to do a job which intuitively seems to have an easier solution is not satisfying. Hence a scheme which automatically provides look-ahead while requiring a minimum of record-keeping and decision-making is needed.

IV. SIGNAL-IN-NOISE PROBLEM

It has been proposed that the problem of detecting page-use patterns be thought of as a signal-in-noise problem. The "signal" is the periodic use of a page, and the "noise" is a random variable of wait introduced by the processes using the page being in the queue waiting for service. If the processes were never interrupted, the periods of use would be easily observable. But since processes are interupted at random times and enter the scheduling queues for random times, the periodicity may be masked by the queue cycle time. The problem is then: given the statistics of the queue waiting time, can the periodicity of use (if present) of a page be detected?

We formulate the problem as follows. Events happen periodically with period T. Before they are observed, however, a random variable X is added to the period so that the interval t between observed events is

(23) $$t = T + X$$

Suppose X has some density function $p_X(u)$, and that $p_X(u) = 0$ for X less than 0. X is independent of T. Therefore

(24) $$E[t] = T + E[X]$$

and

(25) $$\text{var}[t] = \text{var}[X]$$

Thus an estimate of T would be

(26) $$T_{est} = E[t] - E[X]$$

This estimate might be obtained by measuring the waiting time until the nth event observed, W_n, and dividing by n. Thus

$$(27) \qquad E[t] \approx \frac{W_n}{n}$$

for large n. The expected halue of queue wait, $E[X]$, could be obtained by monitoring the activity of the queue, such that $E[X]$ and var[X] are good estimates for the present time of day and would vary from time to time as the system load varied.

Unfortunately it is pointless to formulate the problem in this manner for several reasons. First, a program-time cycle, T, is likely to be short compared to the wait in the queue. Furthermore the wait in the queue is likely to have a large variance, so that there will be much uncertainty in the estimate. For example the estimate might be "10 msec ± 2 seconds". This is hardly reliable. If we introduce a sample mean and variance

$$(28) \qquad E[T_{est}] = \frac{1}{N} \sum_{i=1}^{N} [T_{est}]_i = E[t] - E[X] = \frac{W_n}{n} - E[X]$$

then

$$(29) \qquad \text{var}[T_{est}] = \frac{1}{N} \text{var}[X]$$

It may take many page references to make the variance of the estimate become small. The page might not be in use long enough for this to become meaningful. Furthermore a period T might not exist at all. The conclusion is: in real time no reliable measurement can be obtained as an estimate of a program time use-pattern. It is the large mean and variance of queue wait compared to the actual cycle time of a page loop that lead to the unreliability of the estimate.

Therefore if the Allocator were attempting to predict when a page were to be used again, it would be better off if it predicted the time to next use to be the wait in the queue, than if it tried to perform sophisticated measurements to estimate program-time periods. This important conclusion will be used shortly. The only good means, therefore, of detecting cycles is to measure them on a program-time scale; that is, take measurements only when the process using a given page is running.

V. ALLOCATION OF WORKING SETS

It was apparent from the discussion of present-day bumping techniques that use patterns are really helpful only for predicting whether or not a page is to be reused. If the Allocator is in a position to predict it can select the page which well be unneeded for the longest period of time, thereby reducing the number of swaps per unit time. But it is necessary to make measurements on a program-time scale, and to maintain fairly extensive records. We saw also that page-use cycles in program time are likely to be small compared to cycles around the queue; therefore, in Real Time a page may exhibit "bursts of activity", each burst occurring when the associated process is running, and each inter-burst interval of inactivity occurring while the process is either blocked in an i/o wait or waiting in the queues. Therefore we can think of page use as being associated with the fact that a process is running rather than with how the process is using pages when it is running. This leads us to thinking in terms of working sets of information.

The Working Set of Information is the set of pages a computation is presently referencing.* If we know the pages belonging to the working set it is a simple matter to be sure such pages are loaded into main memory before a process is restarted. In fact we might even feel free to delete all of

*The concept of a working set as the minimum amount of information a process must have in main memory in order to operate efficiently was first proposed by J.B. Dennis.

process' working set when it is not running, provided only that those pages are restored to main memory just prior to restarting the process.

In the next few pages we will discuss possible deletion policies that would be useful if we know the working set, and choose one of these as most practical. Then we discuss a means to detect the working set, and show how it might be implemented in the context of present ideas on multiprogrammed computations [4,7]. We also show that the size of the working set of information is described by a Markov Model, and how this might be used to control the system loads. Finally we discuss the implications of this thinking.

Allocation Schemes if the Working Set is Known

It is clear that the first pages eligible for deletion are thos no longer in any working set. The second choice is any read-only pages belonging to the working set of a computation not expected to run in the near future. We list some alternatives for the third choice, used in case there are no pages covered by either of the first two:

(1) choose a page at random from the largest working set.
(2) choose a page at random from the smallest working set.
(3) choose the most used page from the lowest priority non-running process.
(4) choose the least used page from the lowest priority non-running process.
(5) choose pages in order to distribute deletions over all processes. Let n_i be the size of the i^{th} working set, and N be the capacity of main

[illegible] the smallest number of [illegible]

$$\sum_{i=1}^{k} a_i \geq N$$

So delete

$$\sum_{i=1}^{k} x_i \geq N$$

[illegible] each of the N working sets.
[illegible] pages from the lowest priority non-running process.

The [illegible] lowest priority process is the one at the end of the [illegible] in the queue, and consequently has the longest [illegible] in the queue.

[illegible] (1) and (2) are unacceptable for two reasons. [illegible] discriminating against computations of a particular [illegible] necessarily bad) they would require that working sets of [illegible] computations be distinguished. It is desirable to measure the working set of the entire system, since this is [illegible] measurement to make. Moreover, pages deleted by (1) or (2) might come from a running computation. Proposals (3) and (4) are unacceptable because additional data must be kept to keep track of how often used, rather than whether or not used. Proposal (5) is unacceptable because it would involve a great deal of computation, and also might delete pages belonging to a currently running process. Proposal (6) is the most satisfactory because it utilizes the lessons learned in the first part of this paper. It chooses pages from the process that is not expected to be using them for the longest period of time, thus maximizing the expected time between swaps. Also since working sets exist in [illegible] there need be no special provision made for translating program time into real time. Proposal (6) meets these conditions. Therefore the rules for deletion are :

(1) delete pages no longer belonging to any working set.

or, (2) delete read-only pages belonging to the lowest priority process not running.

or, (3) delete other pages belonging to the lowest priority process not running.

The lowest priority process can be determined simply by probing the Scheduler's Queues.

The Scheme

We present first what we call a conceptual scheme because implementing it directly in hardware would not be economical. Then we show how it fits into the context of Multiprogrammed Systems such as presented by Dennis and van Horn [4].

Let there be a data map M which is an array of all data objects known to all processes belonging to all computations. Let T be a sampling interval for reading the map. Whenever a data object is referenced the corresponding bit in the map is set to 1. When the map is read at the end of the sample interval T, bits of any running processes are reset. Any processes which are not running do not have their bits reset. Thus if a process is suspended its working set is static in time and the corresponding bits are "frozen" in the map. Therefore the map M is an accurate record of all operands working during the last sample interval,

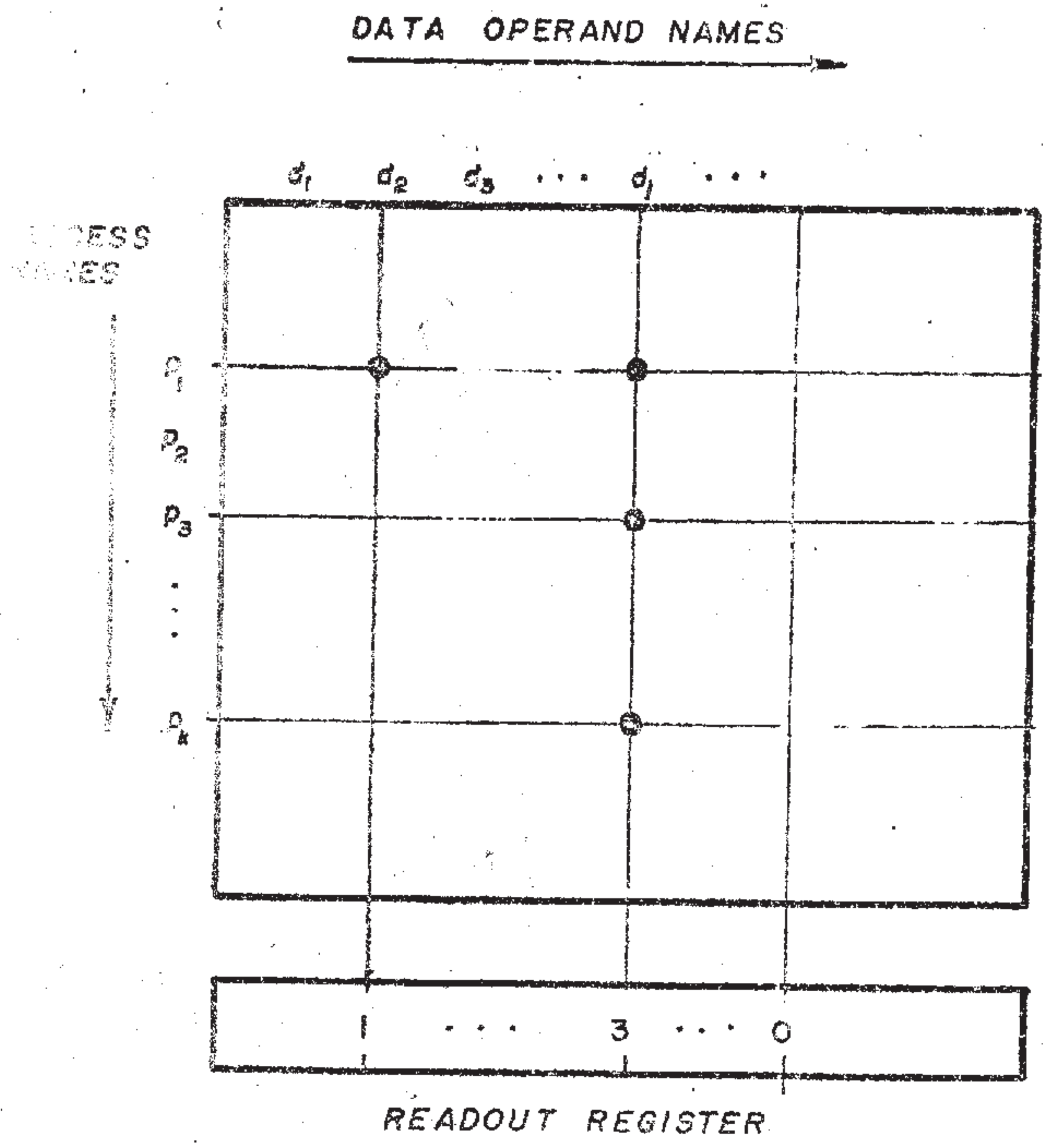

FIGURE 3. The Map M.

regardless of whether the associated process is currently running. Refer to Figure 3.

In M, the $(i,j)^{th}$ position is set whenever process p_i makes reverence to data object d_j. The j^{th} position of the readout register will contain the __sum__ of all the bits set in the j^{th} column -- a __count__ of how many processes have used a particular object. This count will be used to determine the priority of deletion, the higher the count, the lower the priority. Thus shared objects will tend not to be deleted. Now there must be another map M' which has bits set indicating which data objects are actually in main memory. This map will be used to mask the readout register so that only data objects in main memory will be considered for deletion. The masked output will be a string of symbols from the set [Ø,0,1,...,k,...] with the following meaning. Ø means "not in main memory or a working set"; 0 means "in main memory but in no working set"; k means "in main memory and in k working sets". Data objects of priority 0 are first in line for deletion. Objects of priority 1 are next, and so forth. (Always read-only objects within each priority are first.)

There is one problem: if a process should be suspended just after the map was sampled, its bits will have been reset and its working set will appear to be empty, only because it did not have an opportunity to use its objects. Therefore we must think of an additional map M" which acts as a backup for M.

At each sample, a copy of M is placed in M". Then if some process is suspended, M" is OR'd into M for that process, so that no page in that process working set is accidentally deemed no longer working and permanently deleted. These ideas are depicted in Figure 4.

The Sample Interval

The sample interval T at which the map M is sampled cannot be too short or else the process will not have a chance to reference every page in its working set at least once. It cannot be too long or some pages will fall out of the working set undetected, and clutter the memory. It seems desirable that from one sample to the next the average size of the working set should not increase or decrease by more than one page. In this way any error due to a process being suspended during a sample interval (resulting in M" being OR'd into M for that process) cannot exceed one or two pages. Thus the memory will have a minimal number of unneeded pages present. In order to do this, the sample interval must be in the order of a drum transfer time. However, it seems clear that the sample interval should not be fixed -- it should be allowed to vary. It is reasonable that a large working set would require a longer sample interval to allow every page to be reference at least once; conversely a small working set would require a short sample interval. Therefore the System should be able to adjust the sample interval

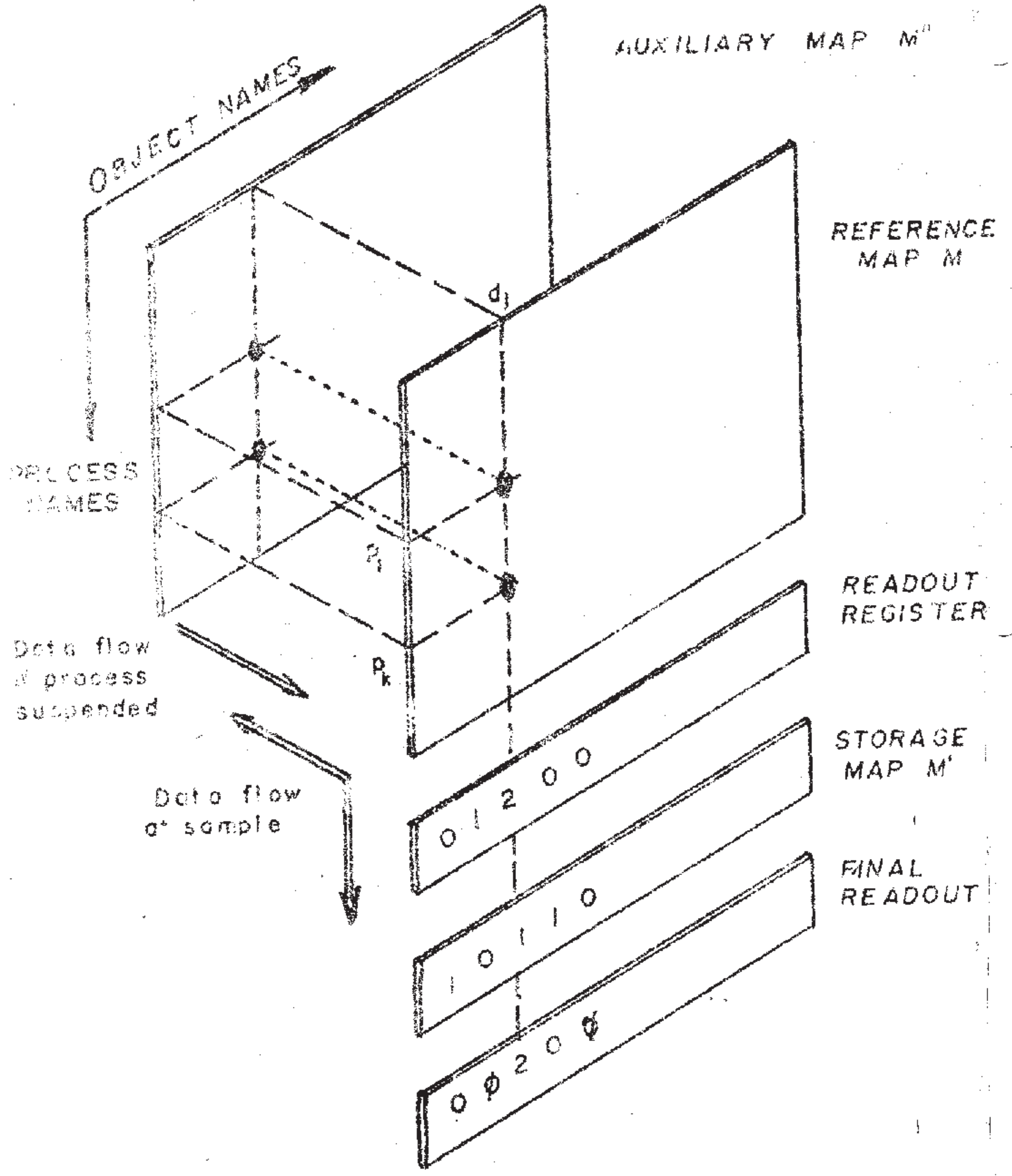

FIGURE 4. Organization of Working Set Detector.

to meet present conditions. If the sample interval is too long the size of the working set will change significantly from one sample to the next, invalidating the assumption that the average working set size does not change much from one sample to the next. Exactly how to determine the sample interval optimally is not yet clear.

The Look-Ahead

We have seen that the Allocator can access the Scheduler's queues, and so discover the lowest priority process, then make the deletion decision among its operands. In this way the time until reuse is maximized. Furthermore the queues should be undergoing enough change so that the lowest priority process is different from one sample to the next, and a given process will not have many pages deleted. But since it will have some pages deleted, it is necessary to restore these pages before restarting the process. It is a simple matter to probe the queues and find out which process is second or third in line, then interrogate M (through another readout register) to discover which pages belong to the working set of that process. The result of this probe is compared with the storage map M', and any missing pages can be restored.

It is clear that this look-ahead scheme can possibly be made highly efficient, and therefore an arbitrarily large number of pages could belong to the working set of the system (for example, many users logged in). This implies that long queue cycle times are possible without sacrificing system efficiency. Hence some limit must be set on the size of the system working set, or on the maximum response time a user must experience, or both. This will be discussed more later.

The Actual Implementation

Clearly it is impossible to build a hardware counterpart of the map M because its dimensions would have to be the maximum possible number of processes by the maximum possible number of pages. Furthermore most of the time much of the map would be empty. If it were stored as a matrix in main memory, the map would have to be referenced each time a page reference was made, and this would be time consuming. Even then it would still require a great number of unused bits. It seem desirable to use a scheme which lists only those pages which actually exist. The groundwork for such a scheme has already been laid by Dennis and others [4,7,12]. It is the notion of associating with each process a descriptor segment, a list of all the data objects known to a process. Each descriptor in the Descriptor Segment can contain a "presence" bit indicating that this data item is in main memory; this corresponds to entries in the map M'. Since all data references must go through the descriptors it is easy to set a "referenced" bit: this corresponds to one element of the map M. Finally an auxiliary "referenced" bit will be present to play the role of the map M". At the end of each sample interval the descriptors of only the active processes are scanned to reset the M-bits. All inactive processes are not sampled and hence their bits remain set. When the need for a deletion arises, the queue is interrogated to find the lowest priority process; the descriptor segment of this process is scanned for mismatches between "presence" and "referenced" bits to determine pages for deletion. If there is any discrepancy between the M and M" bits, they are both set to 1. If still more pages are needed, the second process from the bottom of the queue is scanned. The look-ahead works just as simply. The descriptor of the second process (or some process near the top of the queue) is scanned, and any "working" but not "present" pages are restored to main memory. If a process quits, any pages "present" are immediately subject to deletion.

There is one major difficulty in associating with each process a descriptor segment. This arises when a data object is shared by several processes. Avoiding deletion of that object while it is use by another process is mandatory. The trouble is that there may be many descriptors pointing to it. The solution is to define all the privileges a computation enjoys in a "list of capabilities" or C-list. A C-list will point to a descriptor, and exactly one descriptor is associated with a data object. Then if several computations share an object, the descriptor for that object will contain M- and M"- counts (not bits) and references by different processes will increase the count. References by the same process do not increase the count. The count is the deletion priority. The idea is illustrated in Figure 5.

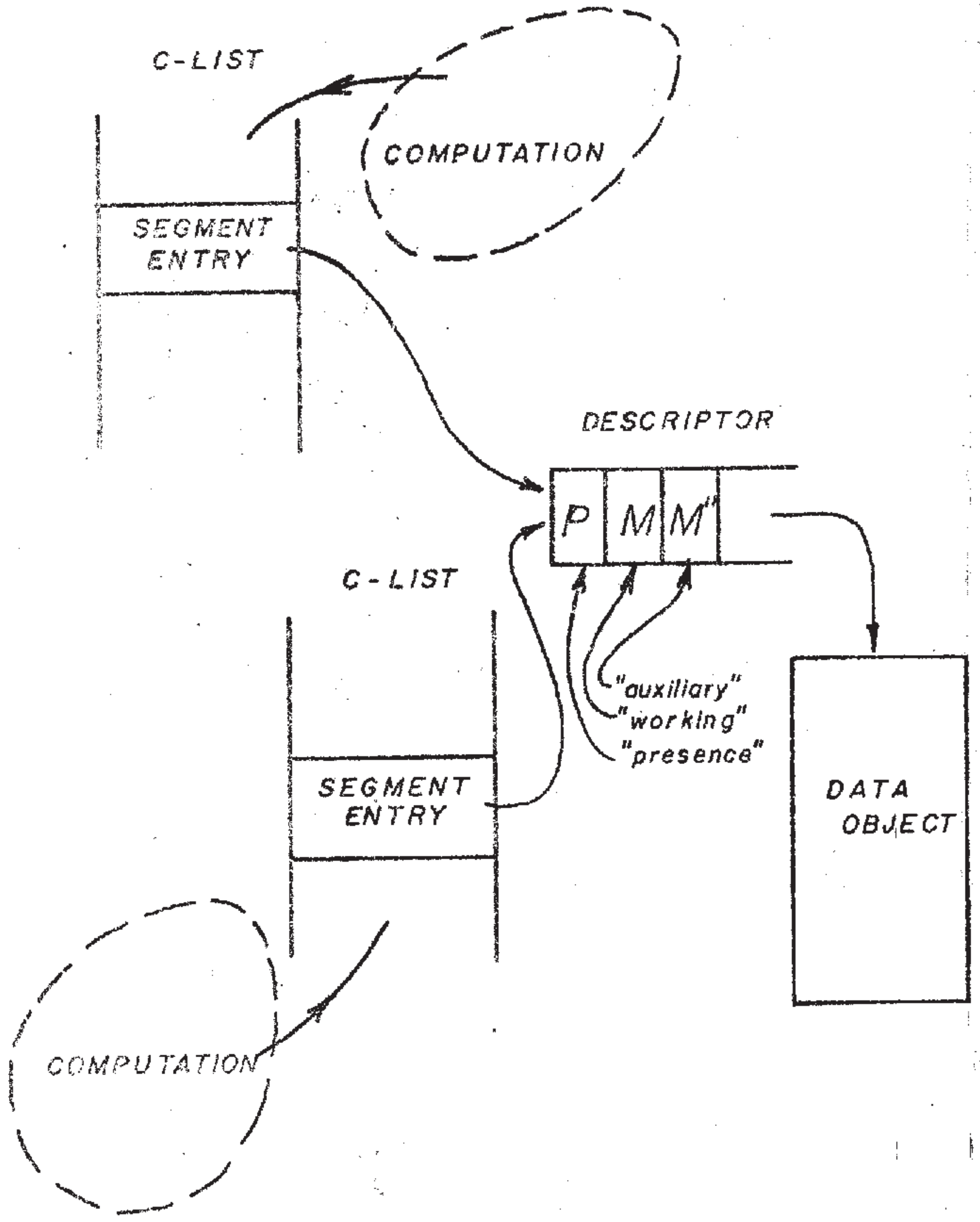
C-LIST
COMPUTATION
SEGMENT ENTRY
DESCRIPTOR
P
M
M''
C-LIST
"auxiliary"
"working"
"presence"
SEGMENT ENTRY
DATA OBJECT
COMPUTATION

FIGURE 5.

Markov Description of the Working Set

We suppose the sample interval is selected so that no working set changes size by more than one page per sample. Let the largest acceptable working set size be N pages. Let the rate at which pages enter the working set be a and that at which they leave be b. Although a and b are time-varying, we assume they do so slowly with respect to the time it takes the system to reach an equilibrium. Let n be the number of pages in the working set, and P_n be the probability that n pages are in the working set. Then the probability of a transition from state n to (n+1) in a small interval of time dt is

$$P_n \, a \, dt$$

The probability of a transition from state (n+1) to n in a small time dt is

$$P_{n+1} \, b \, dt$$

We are assuming equilibrium so that

(30) $$P_n \, a \, dt = P_{n+1} \, b \, dt$$

(31) $$P_{n+1} = \frac{a}{b} P_n$$

By recursion we have

(33) $$P_{n+1} = \left(\frac{a}{b}\right)^n P_o$$

P_o, the probability of a null working set, is found by

(34) $$\sum_{n=0}^{N} P_n = 1$$

(35) $$P_o \sum_{n=0}^{N} (\tfrac{a}{b})^n = 1$$

Now

$$\lim_{N \to \infty} \sum_{n=0}^{N} x^n = \frac{1}{1-x} \qquad \text{if } x < 1$$

and for large N

(36) $$P_o \approx 1 - r \qquad \text{where} \quad r = (\tfrac{a}{b})$$

Therefore

(37) $$P_n \approx \frac{r^n}{1 - r}$$

The moment generating function for P_n is

(38) $$P_n(z) = \sum_{n=0}^{N} z^n P_n \approx \frac{1 - r}{1 - rz}$$

Therefore the expectation of n is

(39) $$E[n] = P_n'(1) \approx \frac{r}{1 - r} \qquad r = \frac{a}{b}$$

The variance is

(40) $$\mathrm{var}[n] = P_n''(1) + P_n'(1) - [P_n'(1)]^2$$

(41) $$\mathrm{var}[n] \approx \frac{r}{(1 - r)^2}$$

Application of the Markov Model.

We will now show that the System can determine the load ration $r = a/b$ in a very simple manner, and how it can use this information to control the load it experiences. If equation (39) is solved for r we have

(42) $$r = \frac{E[n]}{E[n] + 1}$$

Now at any time it is not unreasonable to expect that the System knows the size of the working set. Hence it can determine the load ratio r at any time by putting the present value of n into (42). Since n will be large, $r \approx 1$, but not quite; this implies $a \approx b$. Therefore the total swap rate is

(43) $$S \approx a+b \approx 2r$$

We wish to know how S should compare with the service rates of the swapping equipment. The number of jobs waiting for service by a hardware swapper could be described by a process similar to the process P_n above, except that a is replaced by 2r and b by s, where s is the service rate of the swapping equipment (note that 1/s = mean service time). Clearly if

(44) $$2r < s$$

the swapping equipment can handle the load, but if $2r \approx s$ there will be a long waiting line. It can be shown [Saaty, 10] that the number in the waiting line for thw swapping equipment is

(45) $$E[k] = r + \frac{r^2 + a^2\sigma_s^2}{2(1 - r)}$$

where a is the arrival rate to the line, b is the service rate (1/b = mean service time), $r = a/b$, and σ_s^2 is the variance of the service distribution. In this case $\sigma_s^2 \approx 0$, since pages are of constant length, $b = s$, and $a = 2r$. Therefore:

$$E[k] = r' + \frac{r'^2}{2(1 - r)} \qquad r' = \frac{2r}{s} \tag{46}$$

If we solve for r' in terms of some desired number K_m of waiting jobs

$$r' = 1 + K_m - \sqrt{1 + K_m^2} \tag{47}$$

Then

$$r \leq \frac{1}{2} s \left(1 + K_m - \sqrt{1 + K_m^2}\right) \tag{48}$$

Clearly the system can check periodically on the value of r it knows; if it is getting too high, it can take action to reduce the size of the working sets (for example, by logging some users out), until the load r is again at the desired level. Similarly if r is less than the tolerated limit of equation (48) the system might permit the load to increase, and so keep itself at its most efficient level, but not allowing itself to go into saturation. Clearly the time constant for this behavior must be long so that the system will tend not to oscillate around the saturation point.

It appears that it may be possible for the swapping equipment to handle the load well, and that the saturation limit imposed by equation (48) may be high. Thus it may be possible to allocate all of memory, with look-ahead, so the problem becomes one of how long a process must wait in the queues for its next quantum. The point is that the upper limit of equation (48) may not be severe enough. A better criterion might be to keep the average response time at a certain value or to require that a certain fraction of the working pages of the system be in the main memory while one process cycles around the queue. Let us consider briefly the case in which we require at least p percent of the System Working set, E[n], to be remainin in main memory while one process cycles through the queue. This implies that at most (1 - p) E[n] pages have been deleted. Deletions are occurring approximately at the arrival rate a of new pages into the working set, if E[n] >> 1. Since a is assumed to be a Poisson arrival rate, the number k of pages to be deleted in time t is poisson distributed:

(49) $$P_k(t) = \frac{(at)^k}{k!} e^{-at}$$

If the average queue cycle time is W_q, then the number deleted is aW_q, and this must be less than (1 - p) E[n]. Therefore

(50) $$a \leq \frac{(1 - p)\ E[n]}{W_q}$$

But from equation (39), E[n] is

$$E[n] = \frac{r}{1 - r} \qquad r = \frac{a}{b}$$

putting this into (50) we have

(51) $$r \geq 1 - \frac{1 - p}{bW_q}$$

And the system could control the load accordingly.

For further discussion of File System Models and capacities, the reader is referred to Denning [2], and Fife [5].

VI. CONCLUSIONS

The most natural way to begin a study of memory allocation is to set forth a model for program behavior. When one tries to apply the most versatile probabilistic model, the Markov Model, he finds that even this model is inadequate, because future states of the system are not solely dependent on the present state. Therefore it is essential that allocation schemes be independent of a priori program models, and be able to "learn" a program's behavior, taking action appropriate for the current situation.

In our study of page turning techniques we learned that the idea of seeking patterns (specifically loops) in the execution of programs is theoretically capable of saving a great deal of swap overhead, but it is impractical. Of the three techniques proposed as page turning algorithms, the "oldest unused" policy is the most practical. Yet because it is so far behind an Atlas-type policy in theoretical value, it is not satisfying. We seek a better method. We considered briefly the idea of a prediction program; but because it is no more than an augmented version of the Atlas scheme it too is impractical. We considered the problem of detecting periodic behavior in the background noise of inter-quantum waits in queue, but found that long queue waits and large variance in the wait tend to make this approach fruitless.

From the page-turning study came two important lessons: (1) In order to minimize the number of swaps per unit time it is necessary to maximize the time until a deleted page is expected to be reused. This is the essence of the Atlas policy. (2) It is fruitless to try and detect program time cycles by any method because of large random fluctuations introduced by waiting in the queues.

We then turned our attention to the notion of a working set of information, the number of pages currently being referenced by all the processes in the system. If we know the working set it is a simple matter to perform a look-ahead and restore any pages deleted but belonging to the working set. Naturally any page that has left a working set is deleted. Since it is unreasonable to require programmers or compilers to insert declarations about working set sizes at various points in programs, the working set must be detected by the system. The problem of detecting the working set is not difficult. It is desirable that the system "learn" the working set in much the same sense that it "learns" the priority level in which to place the process in the queues, rather than requiring programs to declare their working sets from time to time. Conceptually we can think of a matrix map of all data objects belonging to all processes; bits are set when references are made to these objects and are reset when the map is read at the end of a sample interval. Bits of suspended processes are not reset because their working sets cannot change. Then, in order to maximize the time until next use, only pages belonging to the lowest priority process in the queues are considered for deletion (because that process will be the longest before it runs again). It is unnecessary to attempt to resolve page use further into individual cycles within quanta because the uses within quanta are too closely spaced compared to the quanta spacing. This is illustrated in Figure 6. From the figure it is evident that to maximize the time to next reuse, one chooses the process with the longest wait in the queue, i.e., the lowest priority process.

We showed how the conceptual scheme might be implemented in a segmented memory by using already existing data descriptors (for example in the MULTICS System [7]), but we pointed out that because data objects may be shared, it is necessary to have exactly one descriptor per operand, and not separate, duplicate, descriptors belonging to each of the sharing processes (such descriptors belonging to each process are planned for MULTICS). When a deletion was being considered, the descriptors of the lowest priority process are scanned. When a look-ahead was being considered, the descriptors of a process near the top of the queues are scanned.

Finally we showed how the rate at which pages enter the working set is related to the swapping hardware capacity and to the size of the working set. We showed how the system might monitor this rate, to determine the load on its memory resources, and adjust the load to keep within limits of saturation. We discussed two limits: one set by the capacity of the swapping

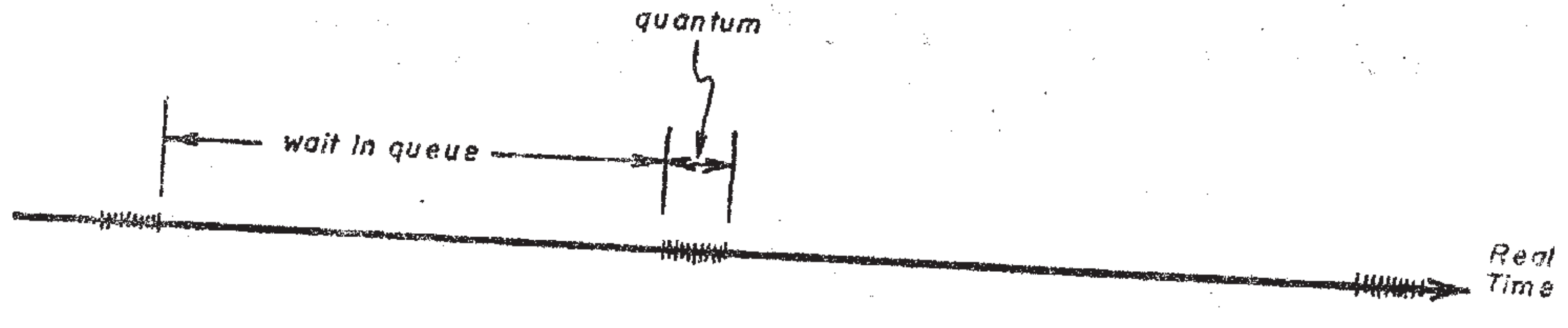

FIGURE 6. History of page use.

hardware, the other set by how much of the working set we are willing to have deleted during the average wait in the queues of one process.

The most important single conclusion we reach is that there must be a definite coordination between the Scheduler and the Allocator. There is a two-way flow of information -- from the queue to the Allocator to determine the best page to delete; from the Allocator to the queues for use in look-ahead. Realizing that this intimate interconnection exists is important. However, more work will be done to understand it. A new approach to the problem of Scheduling is available -- design the best allocator, then find a scheduler that interacts with it smoothly. Most important of all, it suggests that it might be possible to unify completely the notions of "Scheduling" and "Allocation" into one of "System Resource" -- then the scheduling problem will be to schedule the Resources (a mix of memory, processor, and time) to meet the Needs of the user community on a supply-and-demand basis. Some investigation into this viewpoint has been done by Potter [9]. It is clear that if a computing system is to operate someday as a public utility, marketplace procedures will have to be used in scheduling.

APPENDIX 1.

We seek an expression for the mean of the maximum of a set of independent identically distributed exponential random variables with mean 1/c.

Consider a set of N independent exponentially distributed random variables, $\{T_i\}$, with means 1/c and distribution function

$$P[T \leq t] = F_T(t) = 1 - e^{-ct} \qquad t \geq 0$$

Define

$$X = \max_i [T_i]$$

Then

$$\begin{aligned} P[X \leq u] &= P[T_1 \leq u, T_2 \leq u, \ldots, T_N \leq u] \\ &= P[T_1 \leq u]\, P[T_2 \leq u] \ldots P[T_N \leq u] \\ &= (F_T(u))^N \\ &= (1 - e^{-ct})^N \end{aligned}$$

Consider the mean of X,

$$E[X] = \int_0^\infty s\, p_X(s)\, ds$$

Integration by parts with $u = s$ and $dv = p_X(s)ds$ yields

$$E[X] = \int_0^\infty (1 - F_X(s))\, ds$$

Therefore

$$E[X] = \int_0^\infty [1 - (1 - e^{-ct})^N]\, ds$$

Using the Binomial Expansion on $(1 - e^{-ct})^N$;

$$(1 - e^{-ct})^N = \sum_{k=0}^{N} \binom{N}{k} (-e^{-ct})^{N-k}$$

Or,

$$(1 - e^{-ct})^N = \sum_{k=0}^{N-1} \binom{N}{k} (-e^{-ct})^{N-k} + 1$$

Therefore

$$E[X] = \int_0^\infty [- \sum_{k=0}^{N-1} \binom{N}{k} (-e^{-ct})^{N-k}] \, dt$$

$$= - \sum_{k=0}^{N-1} \binom{N}{k} (-1)^{N-k} \int_0^\infty e^{-c(N-k)t} \, dt$$

$$= \sum_{k=0}^{N-1} \binom{N}{k} (-1)^{N-k+1} \frac{1}{(N-k)} \frac{1}{c}$$

Rearranging the terms of the summation,

$$E[X] = \frac{1}{c} \sum_{k=0}^{N-1} \binom{N}{k} \frac{(-1)^{N-k+1}}{(N-k)}$$

$$= \frac{1}{c} \sum_{k=1}^{N} \binom{N}{k} \frac{(-1)^{k-1}}{k}$$

Now we wish to show that

$$S(N) = \sum_{k=1}^{N} \binom{N}{k} \frac{(-1)^{k-1}}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}$$

PROOF.*

BASIS. $S(1) = \frac{(-1)^0}{1} \binom{1}{1} = 1$

INDUCTION STEP. Show that $S(N+1) = S(N) + \frac{1}{N+1}$

*The author wishes to thank David Martin, who furnished this pro

Now
$$\binom{N+1}{k} = \binom{N}{k} + \binom{N}{k-1}$$

which follows from the fact that

$$\binom{N}{k} = \frac{N!}{(N-k)!\ k!}$$

Then

$$S(N+1) = \sum_{k=1}^{N+1} \frac{(-1)^{k-1}}{k} \left[\binom{N}{k} + \binom{N}{k-1} \right]$$

$$= \sum_{k=1}^{N} \frac{(-1)^{k-1}}{k} \binom{N}{k} + \sum_{k=1}^{N+1} \frac{(-1)^{k-1}}{k} \binom{N}{k-1}$$

since

$$\binom{N}{N+1} = 0$$

Thus we have

$$S(N+1) = S(N) + \sum_{k=1}^{N+1} \frac{(-1)^{k-1}}{k} \binom{N}{k-1}$$

Now show that

$$(N+1) \sum_{k=1}^{N+1} \frac{(-1)^{k-1}}{k} \binom{N}{k-1} = 1$$

$$\frac{N+1}{k} \binom{N}{k-1} = \frac{N+1}{k} \frac{N!}{(k-1)!\ (N-k+1)!} = \frac{(N+1)!}{k!\ (N+1-k)!} = \binom{N+1}{k}$$

$$\sum_{k=1}^{N+1} (-1)^{k-1} \binom{N+1}{k} = 1 + \sum_{k=0}^{N+1} (-1)^{k-1} \binom{N+1}{k}$$

$$= 1 + 0$$

Hence $S(N+1) = S(N) + \frac{1}{N+1}$ QED.

Now the sum $S(N)$ is tabulated for the first few values of N:

N	S(N)
1	1.0
2	1.5
3	1.8
4	2.0
5	2.3
6	2.5
7	2.6
8	2.7
9	2.8
10	2.9

We can approximate the sum

$$S(N) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} = \sum_{n=1}^{N} \frac{1}{n}$$

by the integral

$$S(N) \approx \int_{1}^{N} \frac{1}{x}\, dx = \ln N$$

In order to allow for the error for small N, the approximation

$$S(N) \approx 2.9 + \int_{10}^{N} \frac{1}{x}\, dx = 2.9 + \ln \frac{N}{10} \qquad N \geq 10$$

is quite accurate.

Finally, the expectation of the maximum of N exponentially distributed random variables, with means $1/c$, is given by

$$E[T_{max}(N)] = \frac{1}{c} \sum_{n=1}^{N} \frac{1}{n} \approx \frac{\ln N}{c}$$

which was to be shown.

[illegible]

Given a set of uniformly distributed random variables $[t_i, i=1,2,\ldots,N]$, with density functions

$$f_i(t_i) = \frac{1}{T_i} \qquad 0 < t_i \leq T_i$$

where $0 < T_1 < T_2 < \ldots < T_N$. Define

$$X = \max_i [t_i]$$

We want to find an expression for $E[X]$. First, the distribution function $F_X(u)$ is

$$F_X(u) = P[X \leq u] = \begin{cases} \prod_{i=1}^{N} P[t_i \leq u] & 0 < u \leq T_1 \\ \prod_{i=2}^{N} P[t_i \leq u] & T_1 < u \leq T_2 \\ \vdots & \\ P[t_N \leq u] & T_{N-1} < u \leq T_N \end{cases}$$

Then

$$F_X(u) = \sum_{k=1}^{N} \prod_{i=k}^{N} \frac{u}{T_i} [u_{-1}(T_i) - u_{-1}(T_{i-1})]$$

$$F_X(u) = \sum_{k=1}^{N} u^{N-k+1} \prod_{i=k}^{N} \frac{1}{T_i} [u_{-1}(T_i) - u_{-1}(T_{i-1})]$$

where u_{-1} is the unit step function. Define

$$A_k = \prod_{i=k}^{N} \frac{1}{T_i} [u_{-1}(T_i) - u_{-1}(T_{i-1})] \quad , T_0 = 0$$

$$B_k = \prod_{i=k}^{N} \frac{1}{T_i}$$

Then $$p_X(u) = \frac{d}{du} F_X(u) = \sum_{k=1}^{N} (N-k+1)\, u^{N-k} A_k$$

And $$E[X] = \int_{o}^{T_N} \sum_{k=1}^{N} (N-k+1) u^{N-k+1} A_k \, du$$

$$= \sum_{k=1}^{N} \frac{N-k+1}{N-k+2} A_k \, u^{N-k+2} \Big]_{o}^{T_N}$$

Therefore $$E[X] = \frac{N}{N+1} B_1 \, T_1^{N+1} + \frac{N-1}{N} B_2 \, [T_2^N - T_1^N] + \dots$$

$$\dots + \frac{1}{2} B_N \, [T_N - T_{N-1}]$$

If all the $[T_i]$ are the same, i.e., $T_1 = T_2 = \dots = T_N$, then $A_2 = A_3 = \dots = A_N = 0$, and

$$E[X] = \frac{N}{N+1} \prod_{i=1}^{N} \frac{1}{T_i} \, T^{N+1} = \frac{N}{N+1} T_N$$

For all the $[T_i]$ the same, the distribution function is shown in Figure A2.1. Figure A2.2 shows the case when all the $[T_i]$ are not the same. Since we are interested in $E[X]$ we note that

$$E[X] = \int_{o}^{\infty} (1 - F_X(u))\, du$$

$E[X]$ is just the area under the curve $1 - F_X(u)$ shown in Figure A2.3. Let us consider the ratio

$$\frac{E[X]}{\frac{1}{N} \sum_{i=1}^{N} T_i} = r$$

If the $[T_i]$ are uniformly spaced, as indicated in Figure A2.4, then $\frac{1}{N} \sum_{i=1}^{N} T_i = \frac{1}{2} T_N$ and $r \approx 2$. However in a typical situation we might expect that some of the $[T_i]$ will be bunched around smaller values, with occasional large values of T_N, as indicated in Figure A2.5.

If T_N is considerably larger than $\frac{1}{N}\sum_{i=1}^{N} T_i$, r will be larger than 2. In fact it is not unreasonable to expect that $r \approx 5$.

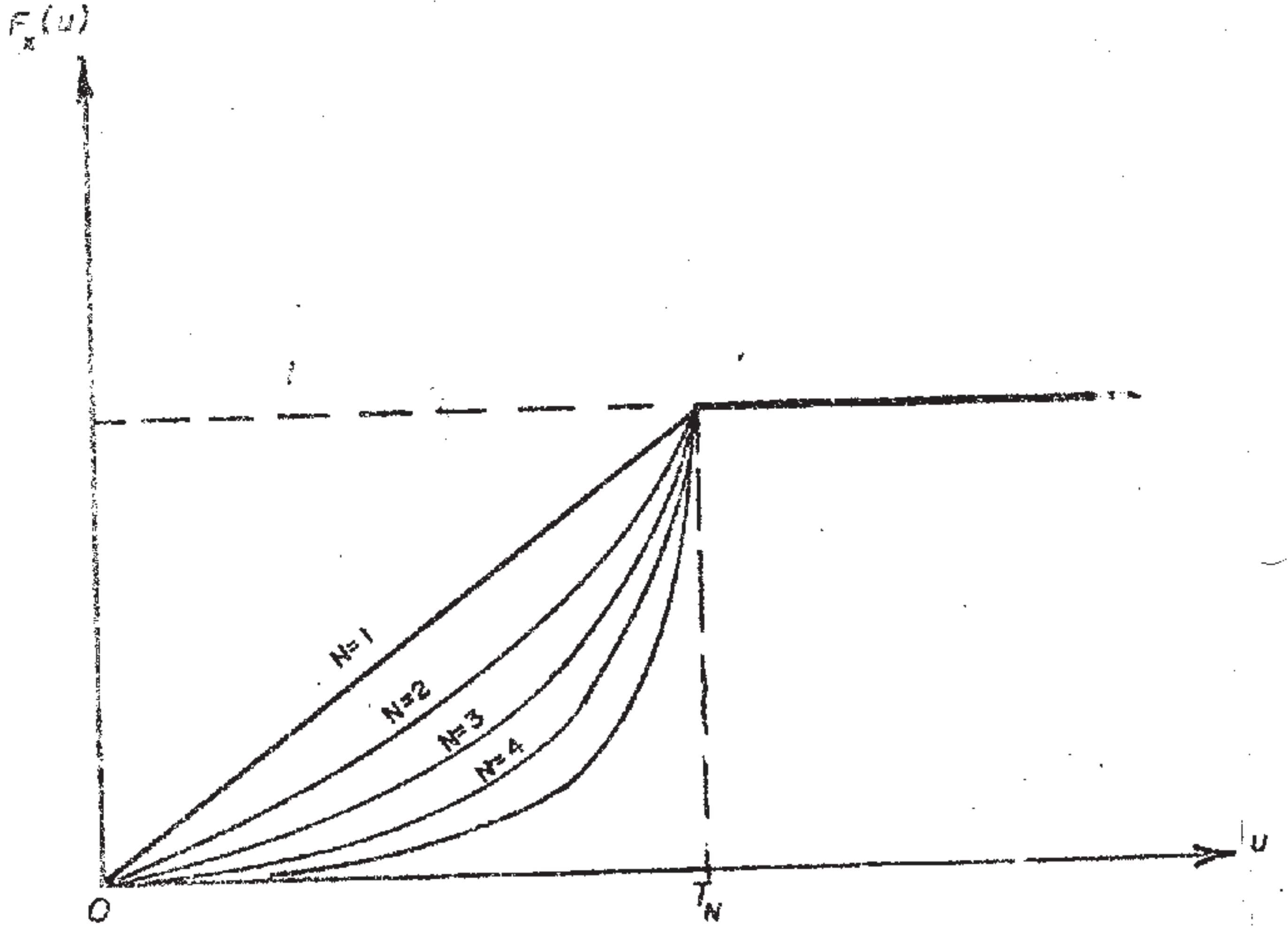

FIGURE A2.1

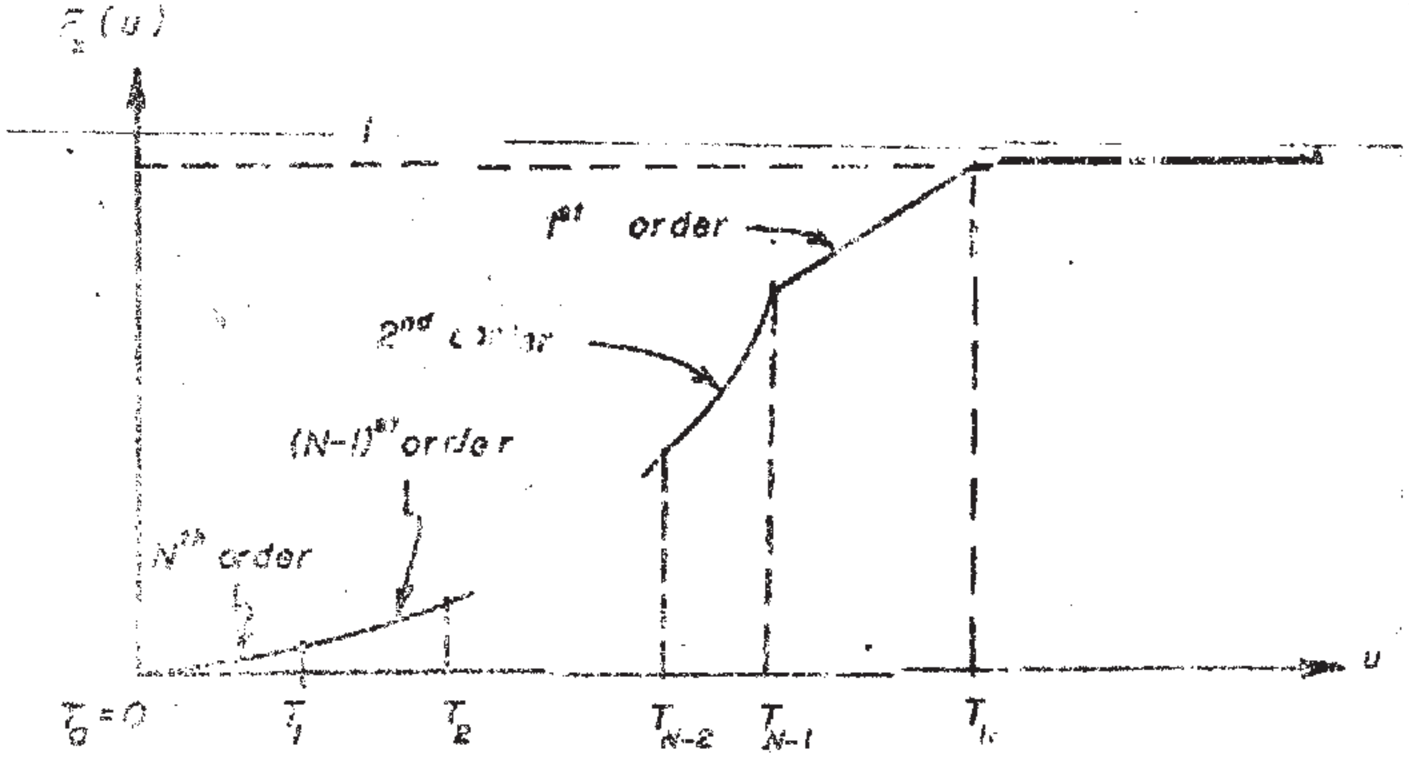

FIGURE A2.2

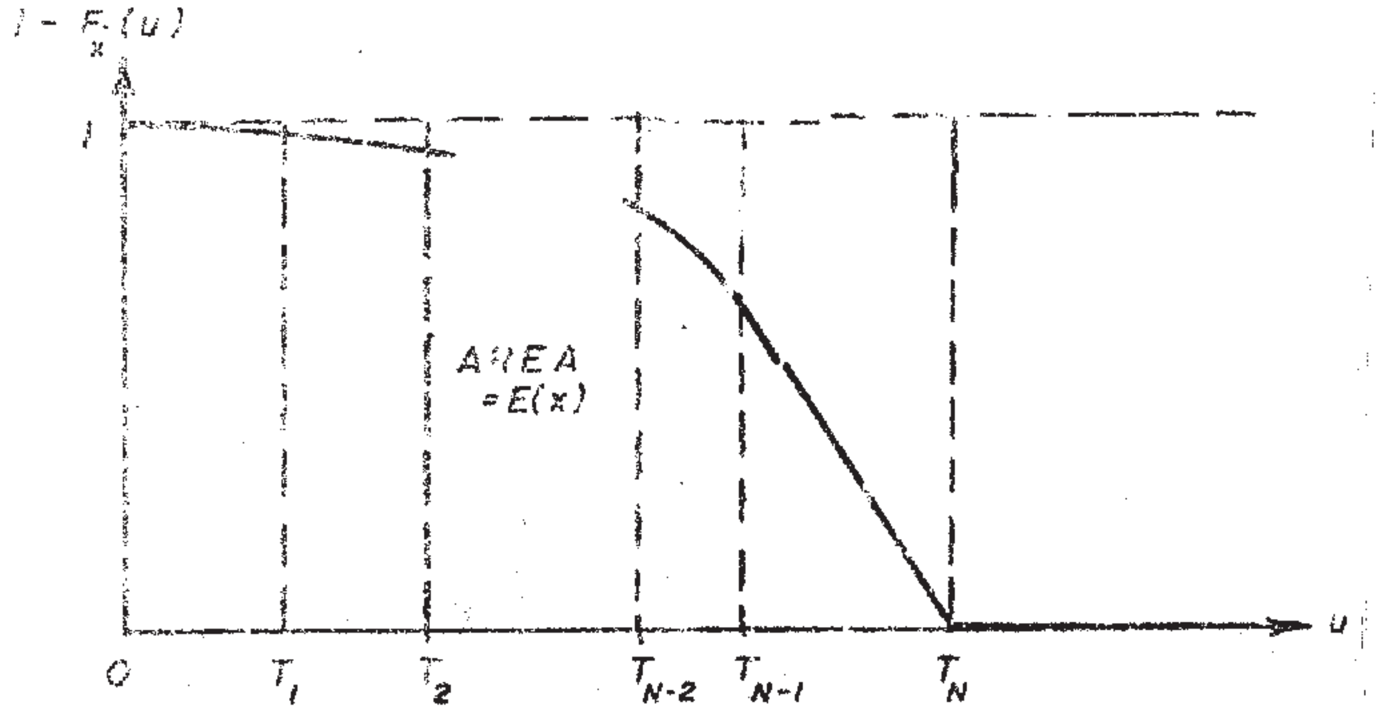

FIGURE A2.3

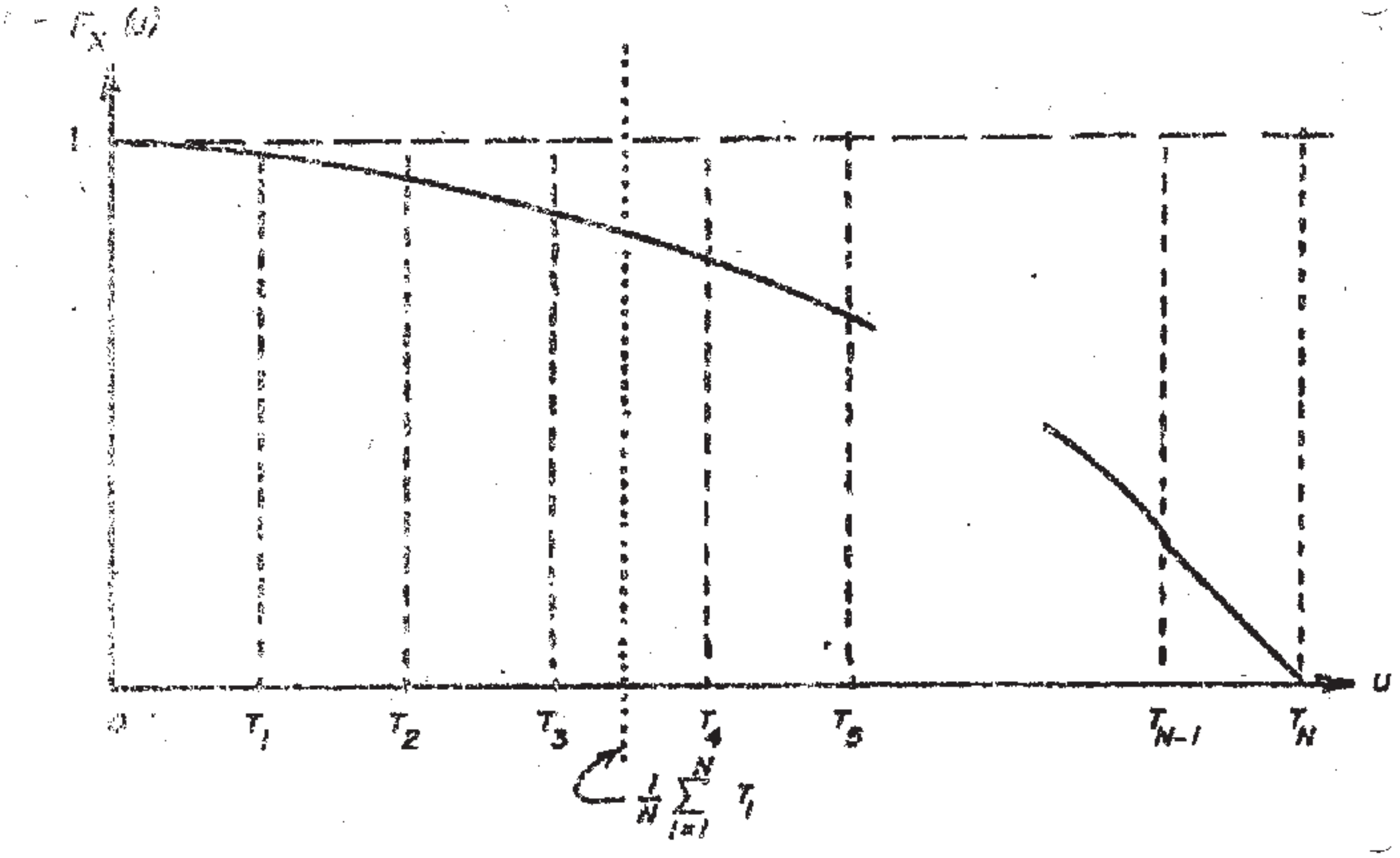

FIGURE A2.4

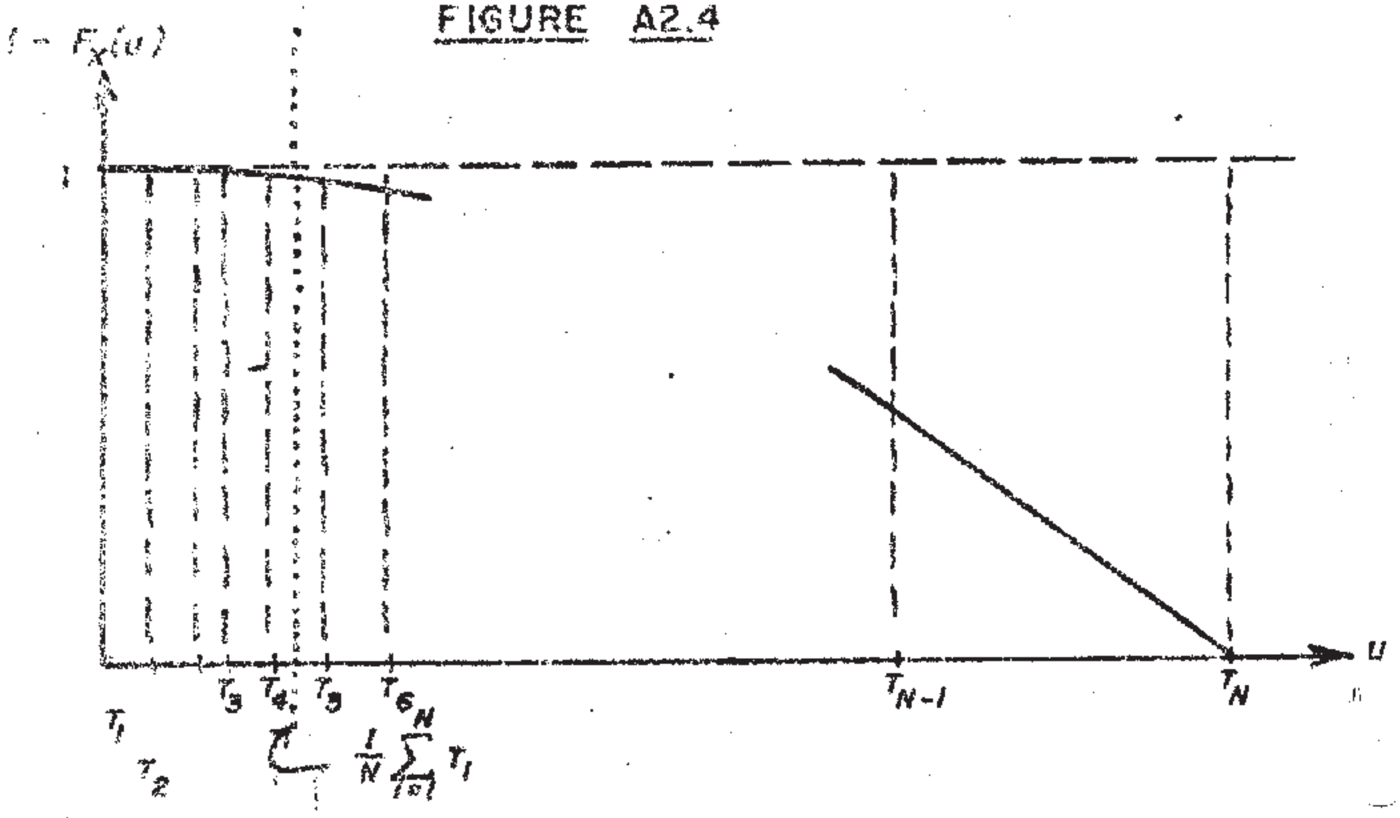

FIGURE A2.5

REFERENCES

1. Corbato, F.J. "System Requirements for Multiple-Access, Time-Shared Computers". M.I.T. Project MAC Technical Report MAC-TR-3.

2. Denning, P.J. "Queueing Models for File Memory Operation", M.I.T. Project MAC Technical Report MAC-TR-21, October, 1965.

3. Dennis, J.B. "Automatic Scheduling of Priority Processes", M.I.T. Project MAC Memorandum MAC-M-189.

4. Dennis, J.B., and van Horn, E.C. "Programming Semantics for Multiprogrammed Computations", M.I.T. Project MAC Technical Report MAC-TR-23.

5. Feller, W. Probability Theory and its Applications. Vol II. New York, John Wiley, 1966.

6. Fife, D.W., and Smith, J.L. "Transmission Capacity of Disc with Concurrent Arm Positioning." IEEE Transactions on Electronic Computers, Vol. EC-14, August 1965.

7. Glaser, E.L., et al, "System Design for Time-Sharing Applications". AFIPS Conference Procedings, Vol. 27, Part 1, 1965 Fall Joint Computer Conference, Spartan Books.

8. Kilburn, et al. "One-Level Storage System", IRE Transactions on Electronic Computers, Vol. EC-11, No. 2, April, 1962.

9. Potter, R.L. "The Optimal Allocation of Resources in a Time-Shared Computer". M.I.T. Course VI S.M. Thesis, June 1965.

10. Saaty, T.L. Elements of Queueing Theory. New York, McGraw-Hill, 1961.

11. Scherr, A.L. "An analysis of Time Shared Computer Systems". M.I.T. Project MAC Technical Report MAC-TR-18, August 1965.

12. Vyssotsky, V.A., et al. "Structure of the MULTICS Supervisor". AFIPS Conference Proceedings, Vol. 27, Part 1, 1965 Fall Joint Computer Converence, Spartan Books.