Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1974

Comments on a Linear Paging Model

Peter J. Denning

Report Number:

74-123

COMMENTS ON A LINEAR PAGING MODEL*

Peter J. Denning
Purdue University**

CSD-TR-123

Abstract: The linear approximation relating mean time between page transfers between levels of memory, as reported by Saltzer for Multics, is examined. It is tentatively concluded that this approximation is untenable for main memory, especially under working set policies; and that the linearity of the data for the drum reflects the behavior of the Multics scheduler for background jobs, not the behavior of programs.

1. Introduction

Saltzer reported recently [1] measurements taken on the Multics system, according to which the mean time between page transfers between a memory level M and the next lower level in the memory hierarchy appeared linear in m, the size of M; the linear behavior for claimed for M being either the main memory or the paging drum. Specifically, the reader is asked to believe two propositions:

P1 The mean time between requests to transfer a page from drum to main memory (i.e., the mean time between system page faults) is linear in the size of the main memory.

P2 The mean time between requests to transfer a page from disk to drum is linear in the size of the drum.

The more I pondered the paper, the less successful was I in reconciling these claims (especially P1) against completely the opposite conclusions one is led to by considering measurement data reported throughout the literature. Proposition P1 is, I believe, simply incorrect. The data

*This work was supported in part by NSF Grant GJ-41269.

**Computer Sciences Dept., West Lafayette, Indiana 47907 USA.

presented in the paper in support of P1 are unconvincing, and it appears that a bizarre scheduling algorithm would be required to cause P1 to hold. Proposition P2 appears to result from the operation of the Multics scheduler and seems at best weakly correlated to program behavior. The extent to which it is satisfied in other systems will depend on the extent to which their schedulers share certain (as yet unknown) properties with Multics. In the following pages I shall attempt to share with you the reasoning which led me to these conclusions.

I should point out that I am dissatisfied less with the conclusions of Saltzer's paper than with its orientation and philosophy. We are being asked to regard the memory system as a black box, to ignore completely its internal structure and organization. We are asked to believe that the external behavior of the system is more or less independent of numerous internal factors over which the system designer has control -- factors such as the policies of scheduling and memory management, page sharing, and the individual behaviors of programs -- without being offered a shred of evidence whether in fact the external behavior is independent of these factors. We are thus asked to sacrifice a great deal of what we know about controlled experiments and the scientific method, as well as a substantial amount of intellectual curiosity. I shall show below that factors such as those listed above are indeed critical, a fact which becomes clear only after one discards the black box philosophy and permits himself the privilege for a moment of peering within the system.

2. Definitions

The acronym LRU signifies "least recently used." An LRU stack over p pages is a time-dependent vector of the form $\underline{S}(t) = (s_1,\ldots,s_p)$ in which each page appears exactly once, t=1,2,3... counts page references, and $i<j$ implies s_i was referenced more recently than s_j. Page x is at distance i in $\underline{S}(t)$ if $s_i=x$. If page x is referenced at time t+1 and is at distance i in $\underline{S}(t)$, it is moved to the first position in $\underline{S}(t+1)$ and the intervening pages pushed down one place; that is

$$\underline{S}(t+1) = (x,s_1,\ldots,s_{i-1},s_{i+1},\ldots,s_p).$$

The importance of an LRU stack is that the first m elements of it are precisely the contents of a memory space of size $m \leq p$ managed under demand fetching and using LRU replacement. A page will be missing from the memory space at time $t+1$ if and only if its distance in $\underline{S}(t)$ exceeds m.

Let a_i denote the frequency of references having distance i in the stack over some measurement period. Let $A_i = a_1+\ldots+a_i$ denote the cumulative frequency distribution, and take $A_0=0$. For a memory space of size m pages, $1-A_m$ can be interpreted as the rate of references to pages not in the space, and $L(m) = 1/(1-A_m)$ as the expected number of references between two for missing pages.

The mean number of references between two missing-page references is called the lifetime [2,3,4], and the function L(m) is called the lifetime function. The mean real time between two missing-page references has been called the headway [1]; it is given by

$$H(m) = TL(m), \tag{2.1}$$

where T is the mean time between references to the store for which L(m) is the lifetime function. The linear paging assumption states that there exists a constant c such that $H(m) = cm$.

A task in the system is a member of the active set if it is eligible to receive processor service and to be allocated pages in main memory. The size of the active set is called the degree of multiprogramming. In Multics (and in many other multilevel memory systems) a task's pages reside initially on the disk. When an active task references a page for the first time, a copy of that page is placed in both main memory and on the drum. After a page has been unreferenced for a sufficient time, the main memory policy will delete it from main memory. If the page remains unreferenced for an additional period, the copy of it on the drum will also be deleted. In Multics, the drum pages are maintained in an LRU stack, the lowest page on this stack being deleted from the drum when an active task generates a page fault that causes a page to be requested from the disk. However, Multics' main memory policy is not based on an LRU stack; it uses an algorithm resembling a working set policy.

The above considerations lead to the observation that the main memory size m and the drum size M satisfy the relation $m \leq M$. It is customary in such systems to take the main memory access time as the basic time unit;

therefore the main memory headway function, which gives the mean time between page faults in the system, is given by taking T=1 in Eq. (2.1):

$$(2.2) \qquad H(m) = L(m).$$

Hereafter, the notation of (2.2) will be used for main memory lifetime and headway functions. The drum headway function $H_D(M)$ gives the mean time between requests to move a page (simultaneously into main memory and drum) from the disk. Under the assumption that every main memory page has a copy on the drum, it must be true that

$$(2.3) \qquad H(m) = H_D(m);$$

that is, the headways agree when M=m. (Saltzer's data obeys this property.) Noting that drum stack updates occur only at page fault times, the intervals between which are T=H(m), we can use eq. (2.1) to obtain for the drum headway function

$$(2.4) \qquad H_D(M) = H(m)L_D(M), \qquad m\leq M$$

where $L_D(M)$ is the lifetime function of the drum. Note that $H_D(M)$ is a function of m, explicitly because of the term H(m) on the right side and implicitly because m affects the drum stack distance frequencies and hence $L_D(M)$. Since the drum is managed according to LRU,

$$(2.5) \qquad L_D(M) = \frac{1}{1-A_M}.$$

Note that (2.3) and (2.4) imply that $L_D(m)=1$, and (2.5) implies that $A_M=0$ for M<m; this is consistent with the assumption that every main memory page has a copy on the drum.* (However, to enforce this, it is necessary to deviate slightly from the drum stack updating procedure: At each page fault, the referenced page is placed as usual on top of the stack; but the page being replaced from main memory, which will appear at some distance not exceeding m in the drum stack, must be moved directly to position m+1 in this stack.)

---

*To be honest, I do not know for sure whether the assumption that every main memory page has a copy on the drum holds in Multics, as Saltzer is not clear on this point. An obvious alternative is to have one copy of a page between the main memory and drum. In this case, the drum stack and main memory contents are disjoint. A reference to drum stack distance i implies a page fault (and a contribution to the frequency a_i), which moves the page off the drum stack and into main memory. A replacement from

.../...

Because $H_D(M)$ is undefined for $M<m$, is it tempting to construct for m given a composition of H and H_D:

$$F_m(x) = \begin{cases} H(x), & x \le m \\ H_D(x), & x > m \end{cases}$$

Because by (2.3) H and H_D agree at the point x=m, the function F_m will exhibit no discontinuities. (Saltzer's Figure 4 is a plot of this function for m=320 pages.) From the earlier discussion, you can see that H and H_D are entirely different functions with different interpretations. A plot of F_m can, therefore, be quite misleading, luring the unsuspecting beholder to the false conclusion that $H=H_D$.

3. The Main Memory Headway Function

The form of the main memory headway function and its relation to Proposition P1 will be considered in this section. By (2.1) it is sufficient to study the lifetime function directly. I must discuss first how the main memory lifetime function relates to those of individual tasks. Let $L_i(x)$ denote the lifetime function of task T_i when it has a space allocation of x pages in main memory. It has been determined that for well-behaved paging algorithms $L_i(x)$ has the S-shaped form suggested in Figure 1 [see 2,3,4], consisting of a concave up region for $x \le x_{0i}$ and a concave down region for $x > x_{0i}$ (where x_{0i} depends on the task). In the concave up region it has been discovered moreover that, approximately

(3.1) $$L_i(x) = c_i x^{k_i}, \qquad x \le x_{0i}$$

where approximately $k_i=2$. The point is, the individual lifetime functions

---

(cont.)

main memory is entered on top of the drum stack. In this case both $m \ge 0$ and $M \ge 0$; eq. (2.3) becomes $H(m) = H_D(0)$; and eq. (2.4) is unchanged. Saltzer's data obeys eq. (2.3), which leads me to suspect that my principal formulation is more accurate that the one in this footnote.

are distinctly nonlinear. Similar behavior will be observed for working set policies, where $L_i(\bar{x})$ denotes the lifetime when mean working set size is $\bar{x}$ [see 5].

The mean main memory lifetime function $L(m)$ observed by the processor executing active tasks in a memory of size m is determined as follows. Consider a sequence of r lifetime intervals on the processor, and suppose T_{j_k} denotes the task to receive service in the k th such interval. The processor's lifetime function, which is also the main memory lifetime function, is then

$$(3.2) \qquad L(m) = \frac{1}{r}\sum_{k=1}^{r} L_{j_k}(x_{j_k})$$

where task T_{j_k} has space x_{j_k} allocated, and the total space allocated among the active tasks is m.

Suppose now that the main memory size is changed from m to Km pages; what effect does this have on the main memory lifetime function $L(m)$? (The linear paging assumption would predict that $L(Km) = KL(m)$.) To answer this, one needs to know how the multiprogramming policy responds to an increase in the main memory size. Consider two extremes:

> Case A. The degree of multiprogramming is held fixed, the extra pages being used to increase uniformly the allocation of each task.
>
> Case B. The degree of multiprogramming is multiplied by (approximately) K, under a working set policy that keeps main memory as fully allocated to working sets as possible.

For case A, the main memory lifetime function (3.2) becomes

$$(3.3) \qquad L_A(Km) = \frac{1}{r}\sum_{k=1}^{r} L_{j_k}(Kx_{j_k})$$

If we assume that each task operates in its concave up region, we have from (3.1) that approximately $L_{j_k}(Kx_{j_k}) = K^2L_{j_k}(x_{j_k})$, which with (3.3) implies $L_A(Km) = K^2L_A(m)$; in this case, the linear approximation is not even close. If we assume each task operates in its concave down region, a similar argument leads to the conclusion $L_A(Km) < KL_A(m)$, again violating the linear apprмximation. The only situation possibly favorable to the linear approximation would take m such that most tasks operate in

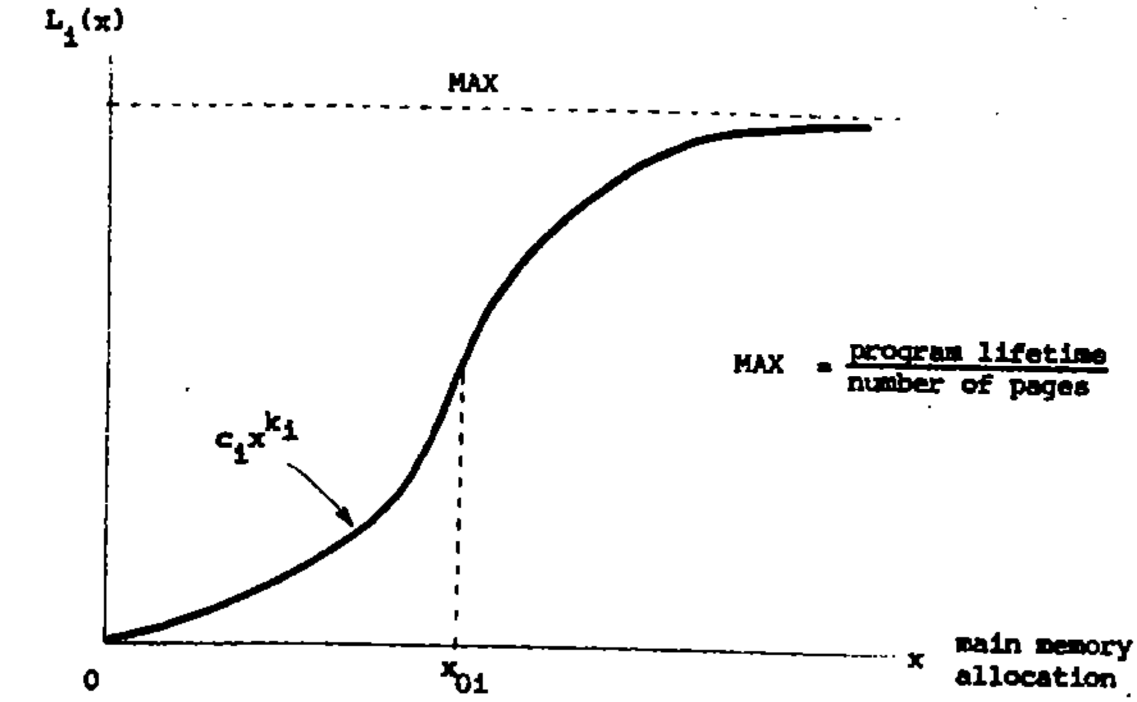

Figure 1. Typical task lifetime function.

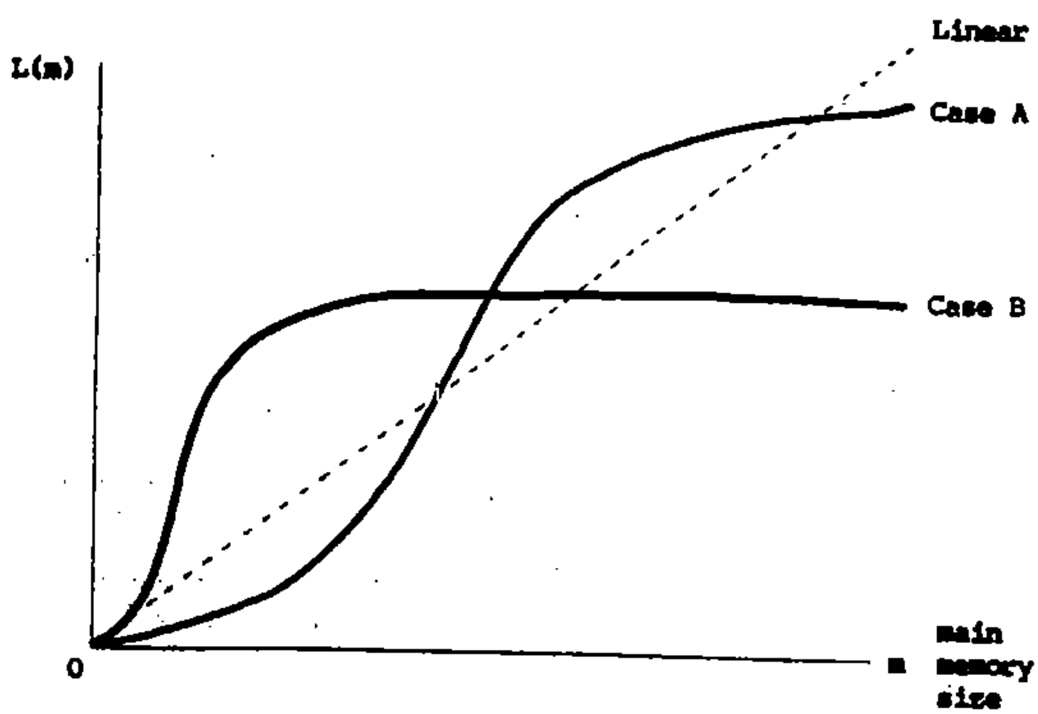

Figure 2. Possible main memory lifetime functions.

their concave up regions, and Km such that they operate in their concave down regions. It is plain, however, that in this case linearity would hold only over a limited range of values of m and K.* Case A multiprogramming policies appear incapable of exhibiting linear behavior over any significant range of memory sizes.

For Case B, the situation is simple indeed. Assuming that Case B follows the working set principle, it will tend to allocate each program the minimum space in which the rate of paging (of that program) does not exceed some predetermined maximum, which implies that $L_{j_k}(x_{j_k}) = L$ (approximately) for all k and some L in (3.3). This yields, approximately,

$$L_B(m) = L \tag{3.4}$$

for all m, in serious violation of the linear assumption. Case B multiprogramming policies necessarily violate the linear assumption for nearly all values of m.

The above conclusions on Case A and Case B multiprogramming policies are summarized in Figure 2. It appears that a multiprogramming policy would have to employ a fortuitous combination of Case A and Case B effects to maintain the linear approximation over any appreciable range of memory sizes. Inasmuch as working set policies are used increasingly in multiprogramming, most systems will exhibit the distinctly nonlinear, controlled Case B behavior.

---

*To verify this, I experimented numerically with simple functions of the form

$$L_i(x) = \begin{cases} x^2, & x \leq x_{0i} \\ x_{0i}^2 + B_i(1 - \exp(-b_i(x-x_{0i})), & x > x_{0i} \end{cases}$$

Every simple example I constructed (by arbitrary but seemingly reasonable choices of the parameters x_{0i}, B_i, and b_i) yielded a function L(m) which was itself S-shaped (as in Fig. 1) and sometimes double S-shaped. None had any appreciable range of linearity. Though examples prove nothing, they did suggest the difficulty of parameter sets occurring "naturally" which produce linear L(m).

The interested reader will find in the paper by Brandwajn et al. [6] a queueing network model in which, given a memory size m, the degree of multiprogramming n is chosen to maximize processing efficiency. The analysis shows that, except for small values of m, the mean time between processor page faults is independent of m. Inasmuch as this is the ideal Case B policy, it further supports my conclusions above.

The diagrams offered in Saltzer's paper to support the linear approximation of main memory lifetime (his Figures 2 and 3) do not in fact support the linear assumption at all. At the very least, they are unconvincing (one contains four data points, the other two). The interested reader will find that an S-shaped curve (such as Figure 1 of this paper) fits this meager data better than straight lines do.* With respect to Multics, therefore, we require more data and more information about the multiprogramming policy before we can conclude anything useful about its main memory lifetime function.

4. The Drum Headway Function

Saltzer's data in support of Proposition P2 shows that the mean time between requests to move a page from disk to drum, referred to here as the drum headway function, is approximated by

(4.1) $$H_D(M) = cM$$

for drum size M in the range

$$320 \leq M \leq 2048$$

and approximately

$$c = 20 \text{ time units/page.}$$

*The data in Saltzer's Figure 2 is taken from Schroeder [7, p239] and is as shown. I remain to be convinced that this is reasonably approximated by a straight line graph. Actually, Schroeder's data defines an "associative memory lifetime function" giving the mean time between two "no-match" events in address translation. Since the associative memory contains both SDWs (segment descriptor words) and PTWs (page table words), and since it is frequently cleared, it is difficult to see the relation between this data and the main memory lifetime function.

m	L(m)
0	0.0
4	9.4
8	34.3
16	80.1

Since the drum is maintained by an LRU stack, we can use the equations (2.4) and (2.5) together with (4.1),

$$(4.2) \qquad H_D(M) = \frac{H(m)}{1-A_M} = cM$$

to study the properties of the drum stack distance distribution A_M. Letting $d = c/H(m)$, we observe that the drum lifetime function $L_D(M) = dM$ also satisfies a linear assumption, and hence

$$(4.3) \qquad A_M = 1 - \frac{1}{dM}$$

$$(4.4) \qquad a_M = A_M - A_{M-1} = \frac{1}{dM(M-1)} = \frac{1}{dM^2}$$

(the last equality is an approximation). It remains to deduce what if anything this implies about scheduler and program behavior.

It is interesting to observe that the working set pages of some task T_i will tend to precede those of T_j in the drum stack, if T_i has been a member of the active set more recently than T_j. In other words, the drum stack can be partitioned as shown in Figure 3; atop the stack are pages belonging to the working sets of active tasks, while farther down the stack can be partitioned into the working sets of tasks $T_{j_1},\ldots,T_{j_r}$ in order of increasing time since last deactivation. (This is of course an approximate description, since nonworking set pages of active and formerly active tasks will be intermingled with the neatly-grouped working set pages.) The pages of the active tasks atop the stack are not likely to be partitioned as neatly as those of inactive tasks, since the drum stack is updated only at page fault times and the processor is cycled among active tasks. The average distance D at which the partitioning begins is $D = \bar{n}\bar{w}$, where $\bar{n}$ is the mean degree of multiprogramming and $\bar{w}$ is the mean working set size. (Saltzer provides no data on $\bar{n}$ or $\bar{w}$, so I have no idea what portion of the drum stack is consumed by pages of active tasks in Multics.) The main point is, the positioning of pages in the drum stack is dominated by the scheduling policies of the system. A program's reference pattern is less significant, for it determines only the relative positions of its pages within the group corresponding to its working set.

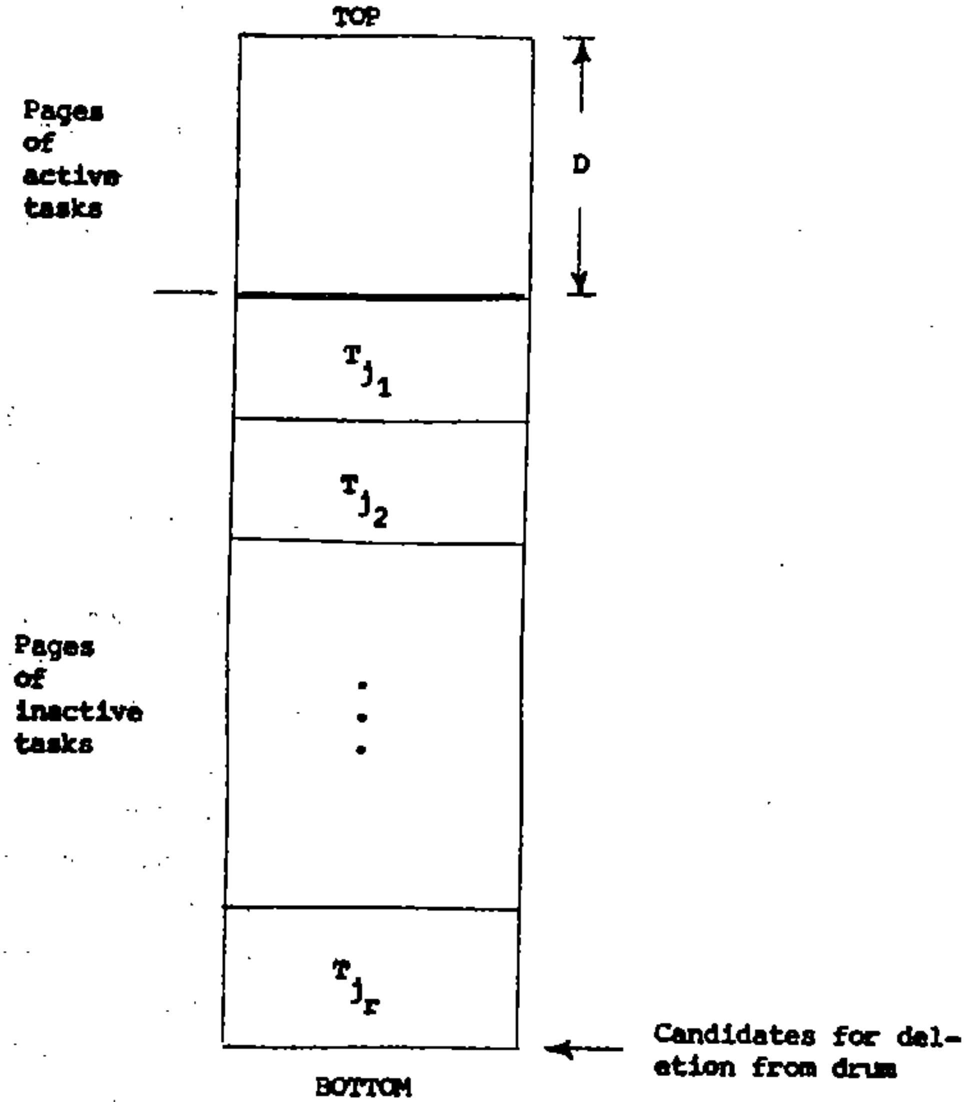

Figure 3. Effect of scheduler on page position in drum LRU stack.

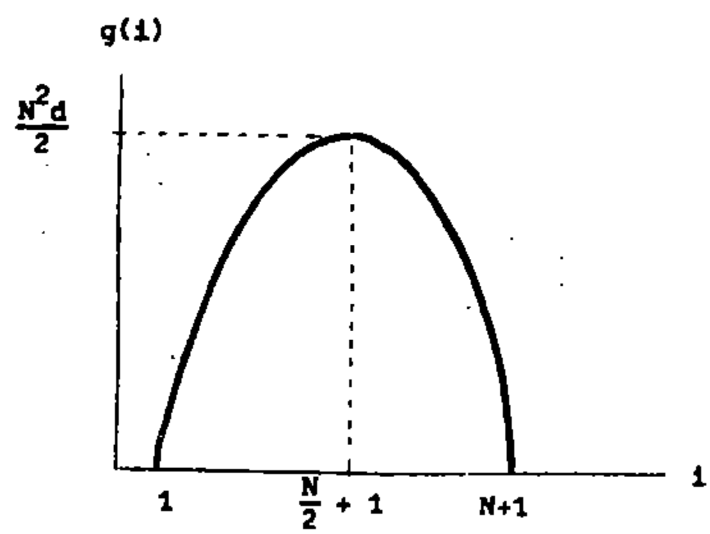

Figure 4. Expected time till reactivation of job in stack position i, for linear lifetime assumption.

As an approximation, therefore, we can map the linearity of the drum lifetime function into that of an <u>LRU job stack</u>, in which T_i precedes T_j if T_i was activated more recently than T_j. The motion of jobs in the job stack corresponds directly to the policy by which the scheduler activates tasks. If $T_1, T_2, \ldots, T_p$ are the tasks (in order) in the job stack, then the drum stack contains the working sets (in the same order) of these tasks, as suggested in Figure 3.

The job stack lifetime function $L(i)$ denotes the mean number of references to the job stack for jobs at positions $i+1$ or greater. Let the mean time between two references to the job stack be signified by t_0, the mean time between two activations; then the mean time between activations of jobs at positions $i+1$ or greater in the job stack is $L(i)t_0$. Now, the linearity of the drum stack lifetime function implies (under suitable assumptions of equilibrium) that of the job stack lifetime function; that is, there exists a constant d such that $L(i) = di$. Comparing against (4.3) and (4.4), we find $A_i = 1-1/di$ and $a_i = 1/di^2$ as defining the frequencies of referencing job stack positions.

In the appendix, we prove expressions for two quantities: $g(i)$ represents the expected number of jobs to be activated before a given job in position i of the job stack; and $h(i)$ represents the expected number of jobs which were activated since the one in position i was deactivated. In other words, $g(i)t_0$ estimates the time until next reactivation of the job in position i, and $h(i)t_0$ estimates the time since last activation of the job in position i. We have

$$(4.5) \qquad g(i) = \frac{N-i+1}{1 - A_{i-1}}$$

$$(4.6) \qquad h(i) = \sum_{k=0}^{i-1} \frac{1}{1-A_k}$$

where N is the capacity of the job stack. Comparing (4.5) and (4.6) with (4.2), we find

$$(4.7) \qquad g(i) = (N+i+1)L(i-1) = d(N-i+1)(i-1)$$

$$(4.8) \qquad h(i) = \sum_{k=0}^{i-1} L(i) = d\frac{i(i-1)}{2}$$

Eq. (4.7) has the form shown in Figure 4. It is interesting that jobs in the middle of the stack have the maximum expected time until reactivation. One explanation for the decrease in g(i) as i approaches N might be that the scheduler compensates jobs which have been inactive for an undue period by forcing them to become active. Or, it might simply be a manifestation of system equilibrium, according to which every job gets run, eventually.

At this point, I have no other interpretations for the above equations to offer. I present them merely as approximations, the utility of which is in question because important effects such as page sharing, initial loading of tasks, and the presence of absentee jobs (background work) have not been accounted for. It is best to regard this as an example of how a scheduler can affect the drum lifetime function, rather than as a definitive explanation for the behavior reported by Saltzer.

In connection with the linear approximation of the drum lifetime function, one other point is important. According to the definition of $L_D(M)$ as $1/(1-A_M)$, the frequency distribution of references to the various positions of the drum stack determines the behavior of $L_D(M)$. Since the frequency a_i measures only the fraction of drum stack updates which moved a page from stack position i to stack position 1, any page which was referenced only once will not affect this data: for that page will be moved immediately from the disk to the top of the drum, then will drift gradually down the drum stack, and finally will return to the disk. The pages of short, terminal-oriented tasks may well satisfy this, as such a task will be activated, bring its working set pages into main memory, and subsequently terminate before the scheduler deactivates it. If the majority of tasks run on Multics are of this type (and I understand that they are), then the data accumulated in A_M (and therefore in $L_D(M)$) may well reflect only the behavior of absentee jobs, and certainly not the majority of work processed by this system. This suggests that the observed drum lifetime function may give us little basis for concluding anything about the majority of Multics' workload: it may be nothing more than noise, the manifestation of, literally, a background effect.

5. Conclusions

One should take great care with experiments designed to measure, as if a black box, the external behavior of a mechanism with controllable internal parameters, for it is easily neglected that the results may depend critically on the parameter settings. There is no apparent theoretical basis for the Multics' claim of linear main memory headway function, and on closer inspection the data put forth to explain the claim appears irrelevant to it. The data put forth to explain a similar claim of linearity for the drum headway function appears on closer inspection to reflect little more than the manner in which the scheduler treats background jobs, which are a small fraction of the work performed by the system. The Multics results appear of limited utility, even to Multics.

Acknowledgement

I am grateful to Jeffrey Buzen and Don Hatfield for useful criticisms offered while I was preparing this paper. My students Kevin Kahn and Rich Simon also provided valuable input.

References

1. Saltzer, J. H. "A simple linear model of demand paging performance." Comm. ACM 17, 4 (April 1974), 181-185.

2. Belady, L. A. and C. J. Kuehner, "Dynamic space sharing in computer systems." Comm. ACM 12, 5 (May 1969), 282-288.

3. Chamberlin, D. D., S. H. Fuller, and L. Y. Liu. "A page allocation strategy for multiprogramming systems with virtual memory." IBM T. J. Watson Research Center Report RC 3848 (May 1972).

4. Ghanem, M. Z. "The lifetime function shape and the optimal memory allocation." IBM T. J. Watson Research Center Report (Sept 1973).

5. Spirn, J. R. "Program locality and dynamic memory management." Ph.D. Thesis, Dept. Elec. Engrg., Princeton Univ., March 1973.

6. Brandwajn, A., J. Buzen, E. Gelenbe, and D. Potier, "A model of performance for virtual memory systems." Proc. 1974 SIGMETRICS Symp. (Oct 1974).

7. Schroeder, M. D. "Performance of the GE-645 associative memory while Multics is in operation." Proc. ACM Wkshp. on Syst. Perf. Eval. (April 1971), 227-245.

APPENDIX

Consider an LRU stack with positions $i = 1,\ldots,N$. Assume that each reference to the stack is independent of past and future references, and that each causes the entry at some distance i to be moved to the top and the intervening entries moved down one place. Let a_i denote the relative frequency of distance i, and A_i its cumulative distribution.

To study the motion of a particular entry in this stack, it is useful to define a Markov chain $X(t)$, in which $X(t)=i$ if and only if the given entry is at position i in the stack at time t, where t counts the number of references to the stack. Initially, $X(0)=1$. Let P_{ij} denote the transition probability $P_{ij} = \Pr[X(t+1)=j / X(t)=i]$. It is easily verified that $P_{ij}=0$ except for these cases:

$$P_{i1} = a_i, \quad P_{ii} = A_{i-1}, \quad P_{i,i+1} = 1-A_i$$

That $P_{ii} = A_{i-1}$ follows from the observation that $X(t+1)=X(t)=i$ if and only if the stack position referenced at time t is one of $1,2,\ldots,i-1$. That $P_{i,i+1} = 1-A_i$ follows from the observation that $X(t+1) = X(t)+1 = i+1$ if and only if the stack position at time t is one of $i+1,\ldots,N$.

Define $g(i)$ to be the mean forward passage time from state i to state 1 — i.e., $g(i)$ is the mean value of k, where k is the smallest integer such that $X(t)=i$ and $X(t+k)=1$. Define $h(i)$ to be the mean backward recurrence time since the most recent exit from state 1 — i.e., $h(i)$ is the mean value of k, where k is the smallest integer such that $X(t-k)=1$ and $X(t)=i$. Now, $g(i)$ is 1 if the transition $(i,1)$ is followed; it is $g(i)+1$ if the transition (i,i) is followed; and it is $g(i+1)+1$ if the transition $(i,i+1)$ is followed. This gives the recursion relations

$$g(i) = a_i + (g(i)+1)A_{i-1} + (g(i+1)+1)(1-A_i), \quad 1\le i<N$$

$$g(N) = a_N + (g(N)+1)A_{N-1}$$

It is easily proved by induction that the solution is

$$g(i) = \frac{N-i+1}{1-A_{i-1}} .$$

which is shown as eq. (4.5) in the main text.

Now, $h(i)$ is $h(i)+1$ if the transition (i,i) was most recently used; it is $1+h(i-1)$ if the transition $(i-1,i)$ was most recently used. This leads to the recurrence relations

$$h(i) = (h(i)+1)A_{i-1} + (h(i-1)+1)(1-A_{i-1}), \quad 1<i\le N$$

$$h(1) = 1$$

It is easily proved by induction that the solution is

$$h(i) = \sum_{k=0}^{i-1} \frac{1}{1-A_k}$$

where $A_0=0$. This is shown as eq. (4.6) in the main text.