Thrashing: Its causes and prevention

by PETER J. DENNING

*Princeton University**
Princeton, New Jersey

INTRODUCTION

A particularly troublesome phenomenon, thrashing, may seriously interfere with the performance of paged memory systems, reducing computing giants (Multics, IBM System 360, and others not necessarily excepted) to computing dwarfs. The term *thrashing* denotes excessive overhead and severe performance degradation or collapse caused by too much paging. Thrashing inevitably turns a shortage of memory space into a surplus of processor time.

Performance of paged memory systems has not always met expectations. Consequently there are some who would have us dispense entirely with paging,[1] believing that programs do not generally display behavior favorable to operation in paged memories. We shall show that troubles with paged memory systems arise not from any misconception about program behavior, but rather from a lack of understanding of a three-way relationship among program behavior, paging algorithms, and the system hardware configuration (i.e., relative processor and memory capacities). We shall show that the prime cause of paging's poor performance is not unfavorable program behavior, but rather the large time required to access a page stored in auxiliary memory, together with a sometimes stubbon determination on the part of system designers to simulate large virtual memories by paging small real memories.

After defining the computer system which serves as our context, we shall review the working set model for program behavior, this model being a useful vehicle for understanding the causes of thrashing. Then we shall show that the large values of secondary memory access times make a program's steady state processing efficiency so sensitive to the paging requirements of other programs that the slightest attempt to overuse main memory can cause service efficiency to collapse. The solution is two-fold: first, to use a memory allocation strategy that insulates one program's memory-space acquisitions from those of others; and second, to employ memory system organizations using a non-rotating device (such as slow-speed bulk core storage) between the high-speed main memory and the slow-speed rotating auxiliary memory.

Preliminaries

Figure 1 shows the basic two-level memory system in which we are interested. A set of identical processors has access to M pages of directly-addressable, multiprogrammed *main memory*; information not in main memory resides in auxiliary memory which has, for our purposes, infinite capacity. There is a time T, the *traverse time*, involved in moving a page between the levels of memory; T is measured from the moment a missing page is referenced until the moment the required page transfer is completed, and is therefore the expectation of a random variable composed of waits in queues, mechanical positioning delays, page transmission times, and so on. For simplicity, we assume T is the same irrespective of the direction a page is moved.

Normally the main memory is a core memory, though it could just as well be any other type of directly-addressable storage device. The auxiliary memory is usually a disk or drum but it could also be a combination of slow-speed core storage and disk or drum.

We assume that information is moved into main memory only on *demand* (demand paging); that is, no attempt is made to move a page into main memory until some program references it. Information is returned from main to auxiliary memory at the discretion of the *paging algorithm*. The information movement across the channel bridging the two levels of memory is called *page traffic*.

A *process* is a sequence of references (either fetches or

*Department of Electrical Engineering. The work reported herein, completed while the author was at Project MAC, was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract No. Nonr-4102(01).

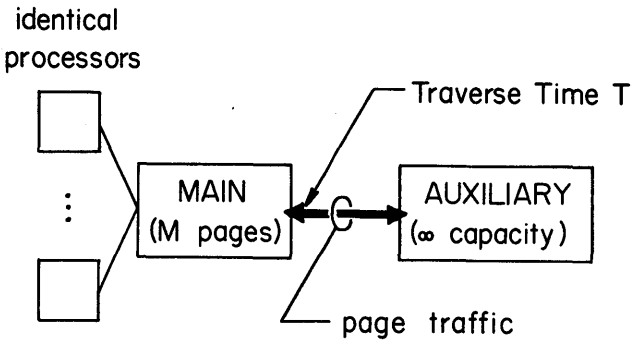

FIGURE 1—Basic two-level memory system

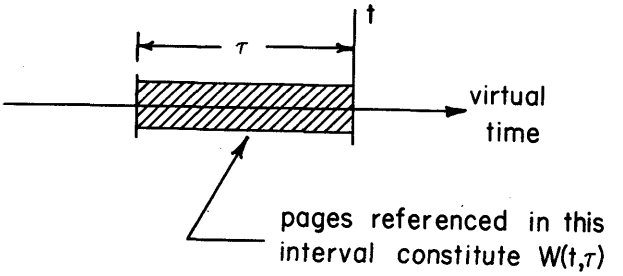

FIGURE 2—Definition of working set

stores) to a set of information called a *program*. We assume that each program has exactly one process associated with it. In this paper we are interested only in *active* processes. An active process may be in one of two states: the *running* state, in which it is executing on a processor; or the *page wait* state, in which it is temporarily suspended awaiting the arrival of a page from auxiliary memory. We take the duration of the page wait state to be T, the traverse time.

When talking about processes in execution, we need to distinguish between real time and virtual time. *Virtual time* is time seen by an active process, as if there were no page wait interruptions. By definition, a process generates one information reference per unit virtual time. *Real time* is a succession of virtual time intervals (i.e, computing intervals) and page wait intervals. A *virtual time unit* (vtu) is the time between two successive information references in a process, and is usually the memory cycle time of the computer system in which the process operates.

In this paper we take 1 vtu = 1 microsecond, since 1 microsecond is typical of core memory cycle times. The table below lists estimates of the traverse time T for typical devices, using the approximate relation

$$T = T_a + T_t$$

where T_a is the mechanical access time of the device and Tt is the transmission time for a page of 1000 words.

Storage Device	T_a	T_t (page = 1000 words)	$T = T_a + T_t$
thin film	0	10^2 vtu	10^2 vtu
core	0	10^3 vtu	10^3 vtu
bulk core	0	10^4 vtu	10^4 vtu
high speed drum	10^4 vtu	10^3 vtu	10^4 vtu
moving-arm disk	10^5 vtu	10^3 vtu	10^5 vtu

The working set model for program behavior

In order to understand the causes and cures for thrashing, it is necessary to understand some basic properties of program behavior. The working set model for program behavior, discussed in detail in reference[2], is a useful way for understanding these properties, so we review it here.

By a *program* we mean the set of pages to which a process directs its references. A basic program property, that of *locality*, is the non-uniform scattering of a process's reference across its program during any virtual time interval. That is, a process tends to favor some of its pages more than others. During disjoint virtual time intervals, the set of favored pages may be different. Locality has been observed to various degrees in existing programs,[3,4] and it can be considerably enhanced if programmers design their algorithms to operate locally on information, one region at a time.

The *working set* of information $W(t,\tau)$ associated with a process at time t is the set of pages referenced by the process during the virtual time interval $(t-\tau,t)$. The concept is illustrated in Figure 2.

The *working set size* $\omega(t,\tau)$ is the number of pages in $W(t,\tau)$. Observe that $\omega(t,\tau) \leq \tau$, since no more that τ distinct pages can be referenced in an interval of length τ; that $\omega(t,0) = 0$, since no references can occur in zero time; and that $\omega(t,\tau)$ is a non-decreasing function of τ, since more references can occur in longer intervals $(t-\tau,t)$.

The working set model owes its validity to locality. A working set measures the set of pages a process is favoring at time t; assuming that processes are not too fickle, that is, they do not abruptly change sets of favored pages, the working set $W(t,\tau)$ constitutes a reliable estimate of a process's immediate memory need.

Intuitively, a working set is the smallest set of pages that ought to reside in main memory so that a process can operate efficiently. Accordingly, τ should be chosen as small as possible and yet allow $W(t,\tau)$ to contain at least the favored pages. In principle, then, τ may vary from program to program and from time to time. A *working set memory allocation policy* is one that permits a process to be active if and only if there is enough uncommitted space in main memory to contain its working set.

Define the random variable x_s to be the virtual time interval between successive references to the same page in a program comprising s pages; these *interreference intervals* x_s are useful for describing certain program properties. Let $F_{x_s}(u) = \Pr[x_s \leq u]$ denote its distribution function (measured over all programs of size s), and let x_s denote its mean.

The relation between the size of a program and the lengths of the interreference intervals to its component pages may be described as follows. Let process *1* be associated with program P_1 (of sizes s_1) and process *2* be associated with program P_2 (of size s_2), and let P_1 be larger than P_2. Then process *1* has to scatter its references across a wider range of pages than process *2*, and we expect the interreference intervals x_{s1} of process *1* to be no longer than the interreference intervals $\bar{x}_{s_2}$ of process *2*. That is, $s_1 > s_2$ implies $\bar{x}_{s_1} > \bar{x}_{s_2}$.

Memory management strategies

It is important to understand how programs can interfere with one another by competing for the same limited main memory resources, under a given paging policy.

A good measure of performance for a paging policy is the *missing-page probability*, which is the probability that, when a process references its program, it directs its reference to a page not in main memory. The better the paging policy, the less often it removes a useful page, and the lower is the missing-page probability. We shall use this idea to examine three important paging policies (ordered here according to increasing cost of implementation):

1. **First In, First Out (FIFO):** whenever a fresh page of main memory is needed, the page least recently paged in is removed.
2. **Least Recently Used (LRU):** whenever a fresh page of main memory is needed, the page unreferenced for the longest time is removed.
3. **Working Set (WS):** whenever a fresh page of main memory is needed, choose for removal some page of a non-active process or some non-working-set page of an active process.

Two important properties set WS apart from the other algorithms. First is the explicit relation between memory management and process scheduling: a process shall be active if and only if its working set is fully contained in main memory. The second is that WS is applied individually to each program in a multiprogrammed memory, whereas the others are applied globally across the memory. We claim that applying a paging algorithm globally to a collection of programs may lead to undesirable interactions among them.

How do programs interact with each other, if at all, under each of these strategies? How may the memory demands of one program interfere with the memory allocated to another? To answer this, we examine the missing-page probability for each strategy.

In a multiprogrammed memory, we expect the missing-page probability for a given program to depend on its own size s, on the number n of programs simultaneously resident in main memory, and on the main memory size M:

$$\text{(1)} \qquad \text{(missing-page probability)} = m(n, s, M)$$

Suppose there are n programs in main memory; intuitively we expect that, if the totality of their working sets does not exceed the main memory size M, then no program loses its favored pages to the expansion of another (although it may lose its favored pages because of foolish decisions by the paging algorithm). That is, as long as

$$\text{(2)} \qquad \sum_{i=1}^{n} \omega_i(t, \tau_i) \leq M$$

there will be no significant interaction among programs and the missing-page probability is small. But when n exceeds some critical number n_0, the totality of working sets exceeds M, the expansion of one program displaces working set pages of another, and so the missing-page probability increases sharply with n. Thus,

$$\text{(3)} \qquad m(n_1, s, M) > m(n_2, s, M) \qquad \text{if } n_1 > n_2$$

This is illustrated in Figure 3.

If the paging algorithm operates in the range $n > n_0$, we will say it is *saturated*.

Now we want to show that the FIFO and LRU algorithms have the property that

$$\text{(4)} \qquad m(n, s_1, M) \geq m(n, s_2, M) \qquad \text{if } s_1 > s_2$$

That is, a large program is at least as likely to lose pages than a small program, especially when the paging algorithm is saturated.

To see that this is true under LRU, recall that if program P_1 is larger than P_2, then the interreference intervals satisfy $\bar{x}_1 > \bar{x}_2$: large programs tend to be the ones that reference the least recently used pages. To see that this is true under FIFO, note that a large program is likely to execute longer than a small program, and thus it is more likely to be still in execution when the FIFO algorithm gets around to removing its pages. The interaction among programs, expressed by Eq. 4, arises from the paging algorithm's being applied globally across a collection of programs.

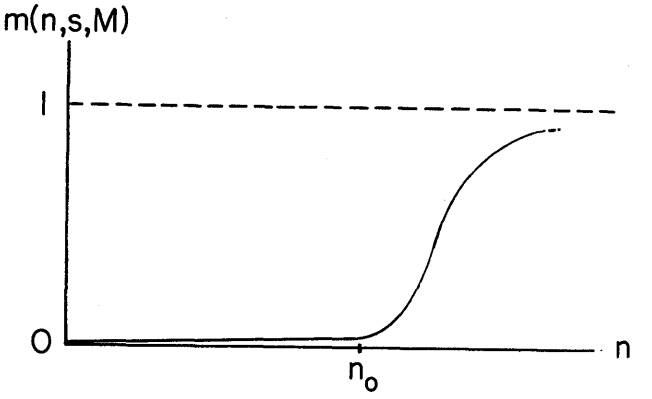

FIGURE 3—Missing-page probability

Finally, we note that under a WS algorithm, the missing-page probability is independent of n and M since eq. 2 is always satisfied. The missing-page probability depends only on the choice of τ; indeed,

$$m_s(\tau) = \Pr[\text{missing page is referenced in size } s \text{ program}]$$

$$= \Pr[\text{page referenced satisfies } x_s > \tau] \tag{5}$$

$$m_s(\tau) = 1 - F_{x_s}(\tau)$$

where $F_{x_s}(u) = \Pr[x_s \leq u]$ has already been defined to be the interreference distribution. Therefore, the WS algorithm makes programs independent of each other. We shall show shortly that this can prevent thrashing.

From now on, we write m instead of $m(n,s,M)$.

Steady state efficiency and thrashing

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

$$e(m) = \frac{\text{(elapsed virtual time)}}{\text{(elapsed virtual time)} + \text{(elapsed page wait time)}} \tag{6}$$

Then,

$$e(m) = \frac{V}{V + VmT} = \frac{1}{1 + mT} \tag{7}$$

Clearly, $e(m)$ measures the ability of an active process to use a processor.

Figure 4 shows $e(m)$ for five values of T:

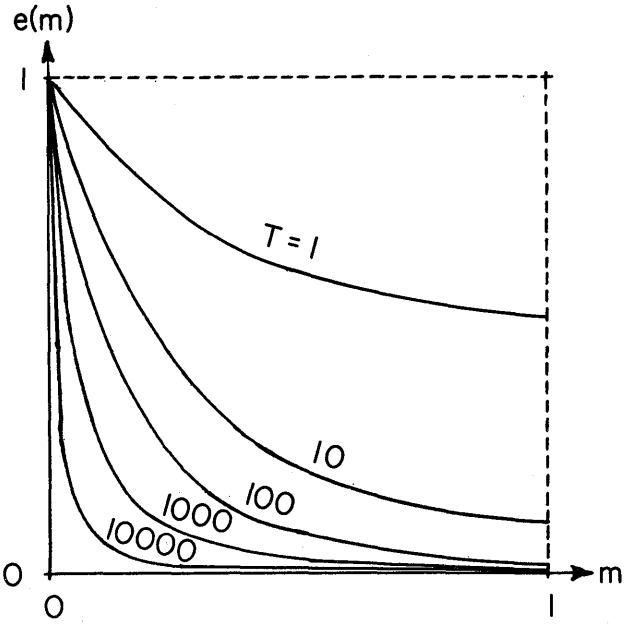

FIGURE 4—Efficiency

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

where $T = 10000$ vtu may be regarded as being typical of the fastest existing rotating auxiliary storage devices.

The slope of $e(m)$ is

$$e'(m) = \frac{d}{dm} e(m) = \frac{-T}{(1 + mT)^2} \tag{8}$$

which means that, for small m and $T>>1$, $e(m)$ is extremely sensitive to a change in m. It is this extreme sensitivity of $e(m)$ to m-fluctuations for large T that is responsible for thrashing.

To show how the slightest attempt to overuse memory can wreck processing efficiency, we perform the following conceptual experiment. We imagine a set of $(n + 1)$ identical programs, n of which are initially operating together, without sharing, in memory at the verge of saturation (that is, $n = n_o$ in Figure 3); then we examine the effect of introducing the $(n + 1)$-st program.

Let *1,2,*,..., $(n + 1)$ represent this set of $(n + 1)$ identical programs, each of average size s. Initially, n of them fully occupy the memory, so that the main memory size is $M = ns$. Let m_o denote the missing-page probability under these circumstances; since there is (on the average) sufficient space in main memory to contain each program's working set, we may assume $m_o << 1$ and that $e(m_o)$ is reasonable (i.e., it is *not* true that $e(m_o) << 1$). Then, the expected number of busy processors (ignoring the cost of switching a processor) is:

$$(9) \qquad p = \sum_{i=1}^{n} e_i(m_o) = \frac{n}{1 + m_o T}$$

Now introduce the $(n + 1)$–st program. The missing-page probability increases to $(m_0 + \Delta)$ and the expected number of busy processors becomes

$$(10) \qquad p' = \sum_{i=1}^{n+1} e_i(m_o + \Delta) = \frac{n + 1}{1 + (m_o + \Delta)T}$$

Now if the pages of n programs fully occupy the memory and we squeeze another program of average size s into memory, the resulting increase in the missing-page probability is

$$(11) \qquad \Delta = \frac{s}{(n + 1)s} = \frac{1}{n + 1}$$

since we assume that the paging algorithm obtains the additional s pages by displacing s pages uniformly from the $(n + 1)$ identical programs now resident in main memory. The fractional number of busy processors after introduction of the $(n + 1)$–st program is

$$(12) \qquad \frac{p'}{p} = \frac{n + 1}{n} \frac{1 + m_o T}{1 + (m_o + \Delta)T}$$

We assume that the traverse time T is very large; that is, T (in vtu) $>> n >>$. We argue that

$$\Delta = \frac{1}{n + 1} >> m_o$$

To show this, we must show that neither $\Delta \approx m_o$ nor $\Delta << m_o$ is the case. First, $\Delta \approx m_o$ cannot be the case, for if it were, we would have (recalling $T >> n >> 1$):

$$(13) \qquad e(m_o) \approx e(\Delta) = \frac{1}{1 + \Delta T}$$

$$= \frac{1}{1 + \dfrac{T}{n + 1}}$$

$$= \frac{n + 1}{n + 1 + T} << 1$$

which contradicts the original assumption that, when n programs initially occupied the memory, it is not true that $e(m_o) << 1$. Second, $\Delta << m_o$ cannot be the case; for if it were, then we would have (from Eqs. 7 and 13)

$$1 >> e(\Delta) = e\left(\frac{1}{n + 1}\right) >> e(m_o)$$

once again contradicting the original assumption that, when n programs initially occupied the memory, it is not true that $e(m_o) << 1$. Thus, we conclude that $\Delta \gg m_o$.

When $T \gg n \gg 1$ and $\Delta = \dfrac{1}{n + 1} \gg m_o$, it is easy to show that

$$(14) \qquad \frac{p'}{p} \approx \frac{n + 1}{T} + (n + 1)m_o \ll 1$$

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

The sharp difference between the two cases at first defies intuition, which might lead us to expect a gradual degradation of service as new programs are introduced into crowded main memory. The excessive value of the traverse time T is the root cause; indeed, the preceding analysis breaks down when it is not true $T >> n$.

The recognition that large traverse times may interfere with system performance is not new. Smith,[5] for example, warns of this behavior.

Relations among processor, memory, traverse time

We said earlier that a shortage of memory space leads to a surplus of processor time. In order to verify this statement, we shall answer the question: "Given p, what is the smallest amount of main memory needed to contain enough programs to busy an average of p processors?" We define $Q(p)$ to be this quantity of memory, and then show that p may be increased if and only if $Q(p)$ is increased, all other things being equal.

Suppose there are n identical programs in main memory, each of average size s and efficiency $e_i(m_i) = e(m)$. The expected number of busy processors is to be

$$(15) \qquad p = \sum_{i=1}^{n} e_i(m_i) = n\, e(m)$$

so that

$$(16) \qquad n = \frac{p}{e(m)} = p(1 + mT)$$

Then the expected memory requirement is

$$(17) \qquad Q(p) = ns = ps(1 + mT)$$

This relationship between memory requirement and traverse time is important. If for some reason the pag-

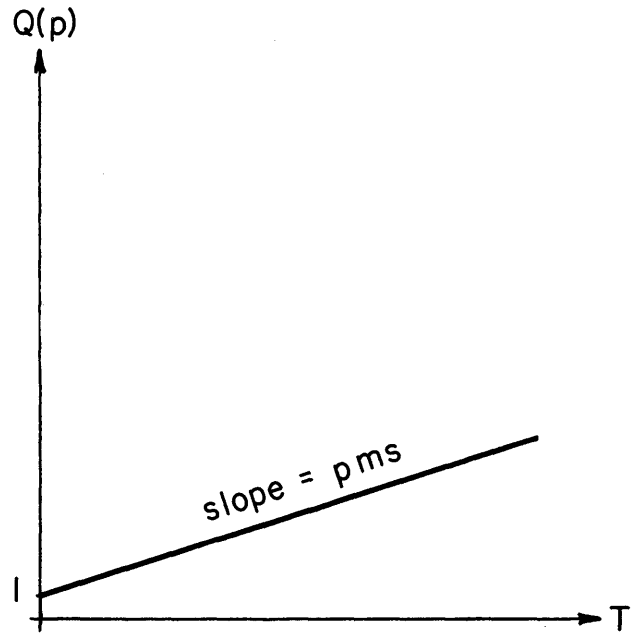

FIGURE 5—Relation between memory size and traverse time

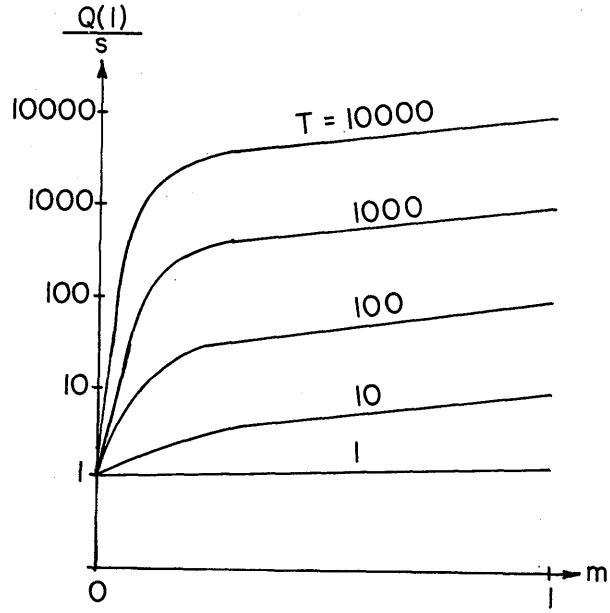

FIGURE 6—Single-processor memory requirment

ing algorithm does not make m sufficiently small, then $mT >> 1$ because $T >> 1$. In this case we have $Q(p) \approx psmT$, *almost directly proportional to the traverse time* T (see Figure 5).

Reducing T by a factor of 10 could reduce the memory requirement by as much as 10, the number of busy processors being held constant. Or, reducing T by a factor of 10 could increase by 10 the number of busy processors, the amount of memory being held constant.

This is the case. Fikes *et al.*[6] report that, on the IBM 360/67 computer at Carnegie-Mellon University, they were able to obtain traverse times in the order of 1 millisecond by using bulk core storage, as compared to 10 milliseconds using drum storage. Indeed, the throughput of their system was increased by a factor of approximately 10.

In other words, it is possible to get the same amount of work done with much less memory if we can employ auxiliary storage devices with much less traverse time.

Figure 6, showing $Q(p)/ps$ sketched for p = 1 and T = 1, 10, 100, 1000, 10000 vtu, further dramatizes the dependence of memory requirement on the traverse time. Again, when m is small and T is large, small m-flucatuations (as might result under saturated FIFO or LRU paging policies) can produce wild fluctuations in $Q(p)$.

Normally we would choose p so that $Q(p)$ represents some fraction f of the available memory M:

$$(18) \qquad (Q)p = fM \qquad 0 < f \leq 1$$

so that $(1\text{-}f)M$ pages of memory are held in reserve to allow for unanticipated working set expansions (it should be evident from the preceding discussion and from Eq. 4 that, if $Q(p) = M$, an unanticipated working set expansion can trigger thrashing). Eq. 18 represents a condition of *static balance* among the paging algorithm, the processor memory configuration, and the traverse time.

Eq. 16 and 17 show that the amount of memory $Q(p) = fM$ can increase (or decrease) if and only if p increases (or decreases), providing mT is constant. Thus, if $p' < p$ processors are available, then $Q(p') < Q(p) = fM$ and $fM\text{-}Q(p')$ memory pages stand idle (that is, they are in the working set of no active process). Similarly, if only $f'M < fM$ memory pages are available, then for some $p' < p$, $Q(p') = f'M$, and $(p\text{-}p')$ processors stand idle. A shortage in one resource type inevitably results in a surplus of another.

If must be emphasized that these arguments, being average-value arguments, are only an approximation to the actual behavior. They nevertheless reveal certain important properties of system behavior.

The cures for thrashing

It should be clear that thrashing is caused by the extreme sensitivity of the efficiency $e(m)$ to fluctuations in the missing-page probability m; this sensitivity is directly traceable to the large value of the traverse time T. When the paging algorithm operates at or near saturation, the memory holdings of one program may interfere with those of others: hence paging strategies must be employed which make m small and indepen-

dent of other programs. The static balance relation $Q(p) = fM$ shows further that:

1. A shortage in memory resource, brought about the onset of thrasing or by the lack of equipment, results in idle processors.
2. A shortage in processor resources, brought about by excessive processor switching or by lack of equipment, results in wasted memory.

To prevent thrashing, we must do one or both of the following: first, we must prevent the missing-page probability m from fluctuating; and second, we must reduce the traverse time T.

In order to prevent m from fluctuating, we must be sure that the number n of programs residing in main memory satisfies $n \leq n_o$ (Figure 2); this is equivalent to the condition that

$$\sum_{i=1}^{n_o} \omega_i(t, \tau_i) \leq M \tag{19}$$

where $\omega_i(t, \tau_i)$ is the working set size of program i. In other words, there must be space in memory for every active process's working set. This strongly suggests that a working set strategy be used. In order to maximize n_o, we want to choose τ as small as possible and yet be sure that $W(t, \tau)$ contains a process's favored pages. If each programmer designs his algorithms to operate locally on data, each program's set of favored pages can be made surprisingly small; this in turn makes n_o larger. Such programmers will be rewarded for their extra care, because they not only attain better operating efficiency, but they also pay less for main store usage.

On the other hand, under paging algorithms (such as FIFO or LRU) which are applied globally across a multiprogrammed memory, it is very difficult to ascertain n_o, and therefore difficult to control m-fluctuations.

The problem of reducing the traverse time T is more difficult. Recall that T is the expectation of a random variable composed of queue waits, mechanical positioning times, and page transmission times. Using optimum scheduling techniques[7] on disk and drum, together with parallel data channels, we can effectively remove the queue wait component from T; accordingly, T can be made comparable to a disk arm seek time or to half a drum revolution time. To reduce T further would require reduction of the rotation time of the device (for example, a 40,000 rpm drum).

A much more promising solution is to dispense altogether with a rotating device as the second level of memory. A three-level memory system (Figure 7) would be a solution, where between the main level (level O) and the drum or disk (level 2) we introduce a bulk core storage. The discussion following Eq. 17 suggests that it is possible, in today's systems, to reduce the traverse time T by a factor of 10 or more. There are two important reasons for this. First, since there is no mechanical access time between levels O and 1, the traverse time depends almost wholly on page transmission time; it is therefore economical to use small page sizes. Second, some bulk core storage devices are directly addressable,[6] so that it is possible to execute directly from them without first moving information into level O.

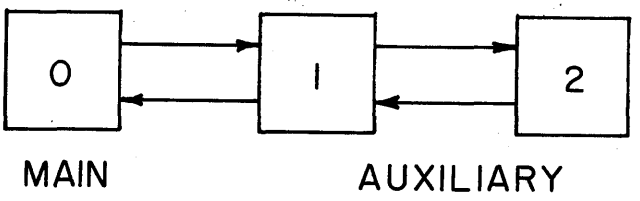

FIGURE 7—Three-level memory system

As a final note, the discussion surrounding Figures 4 and 5 suggests that speed ratios in the order of 1:100 between adjacent levels would lead to much less sensitivity to traverse times, and permit tighter control over thrashing. For example:

Level	Type of Memory Device	Access Time
0	thin film	100 ns.
1	slow-speed core	10μs.
2	very high-speed drum	1 ms.

CONCLUSIONS

The performance degradation or collapse brought about by excessive paging in computer systems, known as thrashing, can be traced to the very large speed difference between main and auxiliary storage. The large traverse time between these two levels of memory makes efficiency very sensitive to changes in the missing-page probability. Certain paging algorithms permit this probability to fluctuate in accordance with the total demand for memory, making it easy for attempted overuse of memory to trigger a collapse of service.

The notion of locality and, based on it, the working set model, can lead to a better understanding of the problem, and thence to solutions. If memory allocation strategies guarantee that the working set of every active process is present in main memory, it is possible to make programs independent one another in the sense that the demands of one program do not affect the memory acquisitions of another. Then the missing-page probability depends only on the choice of the working

set parameter τ and not on the vagaries of the paging algorithm or the memory holdings of other programs.

Other paging policies, such as FIFO or LRU, lead to unwanted interactions in the case of saturation: large programs tend to get less space than they require, and the space acquired by one program depends on its "aggressiveness" compared to that of the other programs with which it shares the memory. Algorithms such as these, which are applied globally to a collection of programs, cannot lead to the strict control of memory usage possible under a working-set algorithm, and they therefore display great susceptibility to thrashing.

The large value of traverse time can be reduced by using optimum scheduling techniques for rotating storage devices and by employing parallel data channels, but the rotation time implies a physical lower bound on the traverse time. A promising solution, deserving serious investigation, is to use a slow-speed core memory between the rotating device and the main store, in order to achieve better matching of the speeds of adjacent memory levels.

We cannot overemphasize, however, the importance of a sufficient supply of main memory, enough to contain the desired number of working sets. Paging is no substitute for real memory. Without sufficient main memory, even the best-designed systems can be dragged by thrashing into dawdling languor.

REFERENCES

1 G H FINE et al
Dynamic program behavior under paging
Proc 21 Nat l Conf. ACM 1966

2 P J DENNING
The working set model for program behavior
Comm ACM 11 5 May 1968 323–333

3 L A BELADY
A study of replacement algorithms for virtual-storage computers
IBM Systems Journal 5 2 1966

4 J S LIPTAY
The cache
IBM Systems Journal 7 1 1968

5 J L SMITH
Multiprogramming under a page on demand strategy
Comm ACM 10 10 Oct 1967 636–646

6 R E FIKES H C LAUER A L VAREHA
Steps toward a general prupose time sharing system using large capacity core storage and TSS/360
Proc 23 Nat'l Conf ACM 1968

7 P J DENNING
Effects of scheduling on file memory operations
AFIPS Conf Proc 30 1967 SJCC