5.2.3 Software Components

Both operating system software and user processes behave in ways that make virtual memory efficient. The former implements explicit policies, and the latter have implicit characteristics that follow expected patterns.

Operating System Policies

Several alternatives exist for the operating system designer for handling page-fault interrupts and other aspects of virtual memory management. Each choice enforces a component of the set of policies for memory management. We briefly review them here.

FETCH POLICY The determination about *when* a page migrates from auxiliary memory to a page frame constitutes the fetch policy. There are two alternatives. The simpler is called *demand fetching* because a page comes into real memory only when a page-fault interrupt results from its absence. The characteristics of auxiliary memory (such as disks with seek and rotational delays preceding information transfer) encourage forms of *prepaging* in which pages other than the one demanded by a page fault are brought into page frames. The utility of this scheme has not been established. However, when a process is initiated (or reactivated) it may be advantageous to immediately preload some pages to prevent frequent page faults.

PLACEMENT POLICY Determining *where* in real memory a page is to reside is irrelevant in a *paged-memory system* because the address translation hardware will use any element of the page table with equal efficiency. *Placement policies* are important, however, in *pure-segmentation systems,* where the parts of a program are variable in size and the minimization of external fragmentation in main memory is an issue. Recent systems that support segmentation also provide paging at a lower level so that segments need not reside in contiguous memory locations, once again rendering a placement policy unimportant. Because modern segmented memory is supported by paging, we do not discuss pure segmentation.

REPLACEMENT POLICY Replacement policy has been much studied over the last two decades. The decision about which page to *replace* when all frames have resident pages can have a great effect on the frequency of subsequent page faults. General-purpose approaches rely on the locality of reference to make decisions. Since the correlation between recent referencing history and near-future referencing patterns is high, realizable replacement policies try to predict future references from past behavior. The set of pages to which accesses are being made changes relatively slowly; if the operating system can accurately determine what this subset is and keep it in real memory, page faulting can be greatly reduced. Both program and data structures contribute to this tendency. The desirable subset that ought to be in real memory is sometimes called a *resident set* or a *locality*.

Several common replacement policies will be useful in this chapter. All are *demand replacement* policies: No prepaging is done, and replacement occurs only

when real memory is full. For any pair of fetch and replacement policies, the processing of virtual memory references results in a sequence of memory states. Specifically, corresponding to a reference string ω is a sequence of real memory states

$$M_0, M_1, \ldots, M_t \ldots, M_T$$

where M_0 is the initial state, usually empty. Real memory has m page frames. A reference to r_t causes a transition from M_{t-1} to M_t:

$$M_t \leftarrow M_{t-1} + X_t - Y_t$$

where X_t is the set of pages fetched because of the reference to r_t, and Y_t is the set of pages removed from real memory. A demand algorithm defines the transition of memory states by

$$M_t \leftarrow \begin{cases} M_{t-1} \text{ if } r_t \text{ in } M_{t-1} \\ M_{t-1} + r_t \text{ if } r_t \text{ not in } M_{t-1} \text{ and } | M_{t-1} | < m \\ M_{t-1} + r_t - y \text{ for some y in } M_{t-1} \text{ if } r_t \text{ not in } M_{t-1} \text{ and } | M_{t-1} | = m \end{cases}$$

Replacement policies vary in how the page y is selected for removal from real memory:

- *Least recently used:* LRU replaces the page in real memory whose last reference was the longest in the past. A faithful implementation of LRU requires hardware support to maintain a stack of pages referenced, an expensive prospect. However, a policy based on examining use bits and dirty bits approximates LRU well. By dividing all frames into four classes according to the value of the pair (use bit, dirty bit), the page to replace is chosen from the first nonempty class in [(0,0), (0,1), (1,0), (1,1)]. All use bits must be periodically reset [Shaw, 1974].
- *Optimal replacement:* OPT selects for replacement the page that will be referenced next the longest time in the future. If a page is not referenced again, its future reference time is ∞. Several pages may have value ∞; any can be selected and optimality is preserved. Although this is not a realizable policy, it is valuable in simulations to determine the best possible paging behavior for a reference string.
- *First-in, first-out:* A FIFO policy is extremely easy to implement with a pointer rotating circularly among the page frames. Unfortunately, the performance of FIFO is inferior to that of LRU.
- *Last-in, first-out:* LIFO ensures that a page will be removed as soon as another page is referenced. This may have utility in specific cases of sequential references.
- *Least frequently used:* LFU replaces the page whose total usage count is the smallest. Implementing LFU accurately requires expensive hardware, and other methods generally outperform it.
- *Most recently used:* MRU replaces the most recently accessed page.

A distinction can be made between replacement policies that consider *all* pages in real memory for replacement and those that choose only among the pages of the process that generated the fault. The former are *global* policies and the latter are

local. Page stealing is possible with a global policy; a page of one process may be replaced in a frame with the page for another process. The dichotomy between local and global policies is well summarized by Carr [1984]:

> Local algorithms have gained considerable popularity among the research community because they are easier to analyze and use in multiprogram system models. Global algorithms are more popular with operating system designers because of their simplicity of implementation and minimal overhead.

MAIN STORAGE ALLOCATION POLICY Allocation policies can be divided into two categories based on how main memory is allocated. A *fixed-allocation policy* gives a process a fixed number of pages in which to execute, while a *variable-allocation policy* allows the page frames held by a process to vary over the lifetime of the process. The former implies a local page replacement policy while the latter allows either a local or global policy. If a process has a fixed number of page frames, difficulties arise if the allocation is too small. Difficulties may also arise if the program makes a *phase transition* from one locality to another. Two properties characterize a reasonable page-replacement algorithm:

1. If a program is executing in a locality whose size exceeds the size of the fixed partition, then the algorithm should do a good job of making the best of a bad situation; that is, on the average it should do a respectable job in attempting to minimize the number of page faults.
2. The algorithm should quickly adapt to a phase transition.

If the amount of main memory exceeds the size of the largest locality, then property 1 is not a factor and one may argue the merits of various page-replacement algorithms. For example, consider the MRU and LFU policies previously mentioned. Both are poor choices for satisfying property 2. The following reference string (for a fictitious program) shows the execution of two program loops:

$$\omega = (123)^r(456)^s$$

If the program executes in a partition of three page frames, then both MRU and LFU have difficulties. MRU incurs $3(1 + s)$ faults and LFU incurs $3(1 + \min(r,s))$ faults. LIFO also fails to adapt to a phase transition. On the other hand, the LRU and FIFO policies each incur the minimum number of page faults (six), readily adapting to the phase transition. Most page replacement algorithms used in practice are variants or enhancements of the LRU or FIFO algorithms.

CLEANING POLICY A cleaning policy is the opposite of a fetch policy. It determines *when* a page that has been modified will be written to auxiliary memory. In *demand cleaning* a dirty page will be written only when selected for removal by the replacement policy. A *precleaning* policy writes dirty pages before their page frames are needed, converting them to clean, resident pages. If a form of precleaning is implemented, the writing of a dirty page need not make it unavailable for use during the transfer. The operating system resets the dirty bit as output is initiated; if a process

modifies it during the transfer, the dirty bit is set and the page must be cleaned again subsequently. A careful balance is necessary between cleaning dirty pages and fetching needed pages. A static priority given to one or the other can result in management problems [Carr, 1984].

LOAD-CONTROL POLICY Without a mechanism to control the number of active processes, it is very easy to *overcommit* a virtual memory system. Each process is actively referencing some subset of its virtual pages. If the sum of the sizes of these subsets exceeds the number of page frames, then frequent faulting will occur. The virtual memory management overhead becomes so great that a precipitous decrease in system performance is visible as soon as the memory overcommitment occurs. A system in this state is said to be *thrashing*. Figure 5.1 shows a typical curve. As the multiprogramming level increases, one would expect that throughput would increase up to the level of system capacity and then decline slowly due to increasing overhead. Instead, throughput falls dramatically because active processes require more real memory than is available. Question 6 illustrates concretely the concept of thrashing.

Global-replacement policies are more susceptible to thrashing because they lack mechanisms to gauge the memory required by individual processes and hence the total demand. A local policy can measure the page-faulting rate of individual processes and determine if more page frames (or fewer) are required. The *working set* virtual memory management method, discussed in Section 5.4, is especially attractive in this regard. Section 5.6 includes a discussion of a global policy that shows promise in handling the load control problem well.

Process Behavior

No action by processes in execution is required to implement virtual memory. Attempts have been made to describe how code should be written or data structures arranged

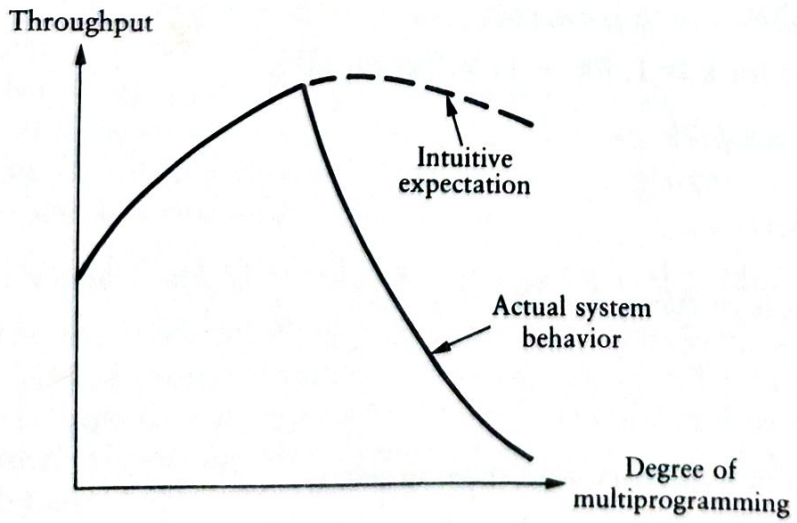

Figure 5.1 Throughput vs. Multiprogramming Level with Thrashing.

to improve virtual memory performance [Brawn, 1968; McKellar, 1969]. However, one of the advantages of automatic memory management is the freedom from explicit action by processes (or programmers). Success relies on programs exhibiting locality in memory references. An automatic procedure for improving locality is discussed in Section 5.7.

5.3 Stack Algorithms

In this section we investigate a class of replacement algorithms called *stack algorithms* [Mattson, et al., 1970; Coffman and Denning, 1973] that have the *inclusion property*. This property allows the computation of the cost of processing a particular reference string for all memory sizes of interest in only one pass over the string, and to predict the cost of executing a particular string in an expanded memory. First, we define what we mean by cost.

5.3.1 Cost Function

If the replacement policy is fixed, the *cost* of processing virtual memory references depends both on the amount of real memory (m page frames) and the particular reference string ω. This in turn determines the contents of X_t and Y_t, the successive fetch and replacement sets. Since many pages in Y_t may be clean, and page writes can be overlapped with other processing, we ignore the latter.

Let f(k) be the cost of a page-fetch operation that obtains k pages from auxiliary memory. The function is normalized by

$$Z = \text{Time}_{delay} + \text{Time}_{transfer}$$

the time required to fetch a single page, so $f(1) = 1$. In keeping with practical algorithms, we assume that

$$f(0) = 0,\ f(k) \geq f(1) \text{ for } k \geq 1,\ f(k+1) \geq f(k) \text{ for all } k$$

Then the cost $C(m,\omega)$ is given by

$$C(m,\omega) = \sum_{t=1}^{T} f(|X_t|)$$

If a demand replacement is in effect, $0 \leq |X_t| \leq 1$, and

$$C(m,\omega) = \sum_{t=1}^{T} |X_t|$$

Notice that for auxiliary memory with moving parts,

$$f(k) = \frac{\text{Time}_{delay} + k \times \text{Time}_{transfer}}{Z}$$

and so $f(k) < k \times f(1) = k$. If electronic auxiliary memory is used,

$$f(k) = k \times \frac{Time'_{delay} + Time'_{transfer}}{Z}$$

with $f(k) = k \times f(1) = k$. This leads to the following result.

THEOREM 5.1

[From Mattson, et al., 1970] *Suppose $f(k) \geq k$. Then for any page-replacement algorithm, there exists a demand algorithm with a cost function that does at least as well for* all *memory sizes and reference strings.*

Because effective prepaging algorithms are generally difficult to find and analyze, most virtual memory implementations using disks and drums use demand-paging algorithms. Further, in anticipation that electronic auxiliary memory will be common in the future, nearly all research has focused on demand algorithms.

5.3.2 Definition of a Stack Algorithm

To expand the notation for memory states, $M(m,\omega)$ is the state of real memory after processing reference string ω in m frames, with initial memory state $\emptyset$. A replacement algorithm is a *stack* algorithm if it satisfies the *inclusion property:*

$$M(m,\omega) \subseteq M(m + 1,\omega) \text{ for all m and } \omega$$

Equivalently, we can express the inclusion property as follows. Given ω, there is a permutation of the virtual pages labeled 1, 2, . . . , *n* called the *stack*

$$S(\omega) = [s_1(\omega), \ldots, s_n(\omega)]$$

so that for all m,

$$M(m,\omega) = \{s_1(\omega), \ldots, s_m(\omega)\}$$

That is, the contents of real memory consisting of m frames is always identified by the first m elements of $S(\omega)$.

For a stack algorithm processing $\omega = r_1, \ldots, r_t, \ldots$, a sequence of stacks $S_1, \ldots, S_t, \ldots$ can be constructed so that the memory state sequence for each value of m is just the first m pages in the stacks. The LRU replacement policy results in a stack algorithm. It is easy to see that, for LRU,

$$M(m,\omega) = \{\text{m most recently referenced pages}\} \subseteq M(m + 1, \omega)$$

On the other hand, FIFO is *not* a stack algorithm, as can be seen from the processing of a reference string in two memory sizes. Figure 5.2(a) shows processing in three page frames; the same string is processed in four frames in Figure 5.2(b). The final memory contents in the two cases show that $M(3,\omega) = \{5,1,2\} \not\subseteq \{5,2,3,4\} = M(4,\omega)$.

Nonstack algorithms such as FIFO also exhibit an unusual property called *Belady's anomaly* [Belady, et al., 1969]. It is not always true that $C(m,\omega)$ is a nonin-

ω	1	2	3	4	1	2	5
M	1	1	1	4	4	4	5
	-	2	2	2	1	1	1
	-	-	3	3	3	2	2

(a) FIFO Processing in Three Frames.

ω	1	2	3	4	1	2	5
M	1	1	1	1	1	1	5
	-	2	2	2	2	2	2
	-	-	3	3	3	3	3
	-	-	-	4	4	4	4

(b) FIFO Processing in Four Frames.

Figure 5.2 FIFO Processing.

creasing function of m; consider Figures 5.3(a) and 5.3(b) in which a reference string is processed in three and then four page frames. The references that result in page fetches are marked with *. In the former, $C(3,\omega) = 9$, but in the latter, $C(4,\omega) = 10$.

5.3.3 Stack-Updating Procedure

We need a method of describing how to obtain the stack S_{t+1} from S_t. First we make some observations about how the movements of pages in a stack are restricted. Define the *stack distance* of a page p as its position in $S(\omega)$. Let $d_p(\omega) = k$ if $s_k(\omega) = p$. If p has not yet been referenced, it does not appear in $S(\omega)$ and $d_p(\omega) = \infty$. Note that a fault occurs in an m-frame memory on the last element p of ωp if $d_p(\omega) > m$.

ω	1*	2*	3*	4*	1*	2*	5*	1	2	3*	4*	5
M	1	1	1	4	4	4	5	5	5	5	5	5
	-	2	2	2	1	1	1	1	1	3	3	3
	-	-	3	3	3	2	2	2	2	2	4	4

Figure 5.3(a) Belady's Anomaly: Three Frames.

ω	1*	2*	3*	4*	1	2	5*	1*	2*	3*	4*	5*
M	1	1	1	1	1	1	5	5	5	5	4	4
	-	2	2	2	2	2	2	1	1	1	1	5
	-	-	3	3	3	3	3	3	2	2	2	2
	-	-	-	4	4	4	4	4	4	3	3	3

Figure 5.3(b) Belady's Anomaly: Four Frames.

Observation 1: Since the most recently referenced page p must be in memory of any size (and, in particular, for $m = 1$), we must have $d_p(\omega p) = 1$.

Observation 2: Stack algorithms are demand algorithms, so a page that is not referenced may not be brought into memory of any size, so $d_q(\omega) \leq d_q(\omega p)$ if $p \neq q$.

Observation 3: If some page q resides in the stack at position k below the referenced page p, observation 2 tells us that it cannot move up the stack. If it could move down the stack, a reference to p in a k-frame memory would result in the removal of q even though no fault had occurred. Hence, such a page does not move in the stack at all: $s_k(\omega p) = s_k(\omega)$ if $d_p(\omega) < k$.

As a result of these observations, we have a general picture in Figure 5.4 of how stack elements move. The referenced page p moves to the top of the stack, the elements from position 1 through $d_p(\omega) - 1$ arrange themselves into positions 2 through $d_p(\omega)$, and the pages below $d_p(\omega)$ are unchanged.

To more clearly define how the dynamic section of stack is actually sorted out, we define *priority algorithms*. A paging algorithm is a priority algorithm if, associated with each reference r_t in ω, there is a priority list L_t with two properties:

1. The priority list L_t is an ordering (by decreasing priority) of the distinct pages referenced so far. This list is independent of the memory size in which a process executes.
2. For all real memory sizes m, the page q removed from real memory as a result of referring to r_{t+1} is the lowest priority page (according to L_t) that is also resident. Refer to this page as $q = \min[M]$. Note that max[M] also has a reasonable meaning. We will extend the notation to select the lower or higher priority of two individual pages as well.

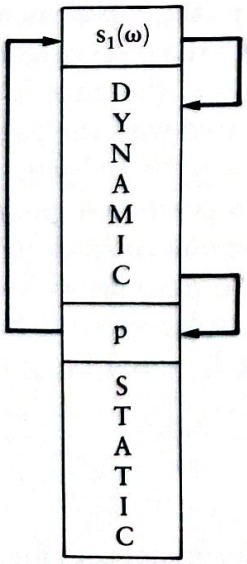

Figure 5.4 Stack Updating Procedure: General Motion.

Priority algorithms are stack algorithms. If memory of size m + 1 contains pages $M \cup \{y\}$ (for any page y), then the page to be removed upon a fault is $\min[M \cup \{y\}] = \min[\min[M], y]$, the lower priority page between that which would be removed from M and the page y. This is just an alternative definition of a stack algorithm.

Stack algorithms are also priority algorithms. The stack algorithms mentioned in this chapter all have a method of ordering pages independently of real memory size:

LRU	Orders by increasing time to last reference
OPT	Orders by increasing time to next reference
LFU	Orders by decreasing frequence of reference (any deterministic tie-breaker will do)
LIFO	Orders by increasing time of real memory entry

One may be tempted to include FIFO in this list, since it also orders pages by time of entry to real memory. However, consider processing $\omega = 123412$ in three and four page frames.After processing ω, the two alleged priority lists would be (3, 4, 1, 2) and (1, 2, 3, 4). Page 2 precedes page 4 in the former, but follows it in the latter. No single priority list suffices independently of real memory size (and FIFO orders only pages in real memory, not all pages referenced), so FIFO is not a priority algorithm and there is no contradiction.

Only in LRU do the stack S and the priority list L coincide. In all other stack algorithms, both must be explicitly maintained in order to perform stack updating. Specifically, we define the stack-updating procedure in terms of the priority of pages. If $S(\omega)$ is a stack in which $d_p(\omega) = k$, we define each stack element $s_i(\omega p)$ by

$$s_i(\omega p) = \begin{cases} p & \text{if } i = 1 \\ \max [s_i(\omega), \min [M(i-1, \omega)]] & \text{if } 1 < i < k \\ \min [M(k-1, \omega)] & \text{if } i = k \\ s_i(\omega) & \text{if } i > k \end{cases}$$

To summarize, the referenced page (at position k) moves to the top element of the new stack, and the page displaced from the top position and the page originally in the second position are compared. The "max" page becomes the page in the second position of the new stack, and the "min" page is compared with the page in the original third position. This comparison continues until the "min" page not placed into the new stack at position $k - 1$ falls into the empty position at position k. Pages below position k do not move. Figure 5.5 gives a diagram of stack movement, with a circle, ○, used to represent the comparison of two page priorities.

An example of processing a reference string using optimal replacement is shown in Figure 5.6. At each reference r_t, the stack S_t is formed using S_{t-1} and L_{t-1}; then the new priority list L_t is obtained.

5.3.4 Calculating Cost Function

One attractive property of stack algorithms is that the cost of fetching pages for any reference string can be computed for all memory sizes in one pass over the string

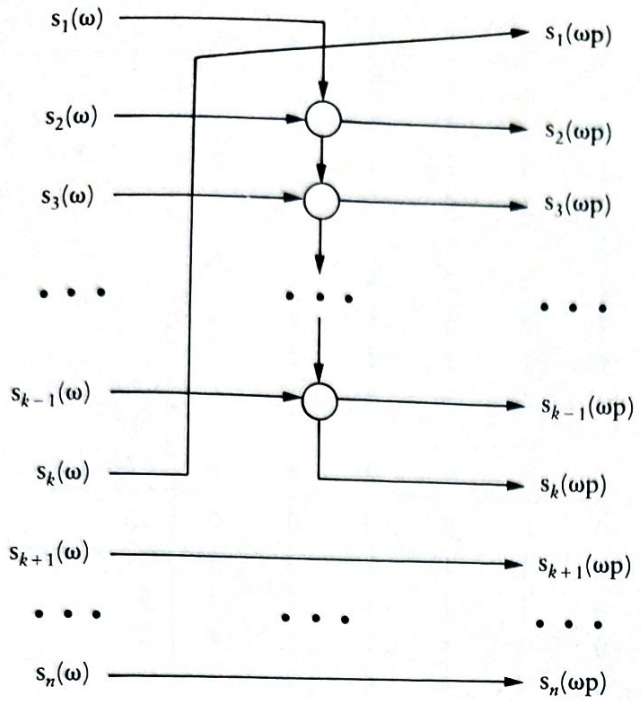

Figure 5.5 Stack Updating Procedure: Priority Operator.

by using the procedure described here. Define a vector with elements c_k = the number of times that a page p moved from position $d_p = k$ to position 1 in the stack because it was referenced. Then the cost of processing the string is

$$C(m,\omega) = \sum_{k=m+1}^{n} c_k + c_\infty$$

where c_∞ is the number of times a new page was referenced and n is the number of distinct pages in ω.

A tableau like the one exemplified in Figure 5.7 can be used to manually process small examples, and gives the method for programming a simulator that processes large reference strings.

ω	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**2**	**3**	**1**
S_t	1	2	3	4	1	2	3	2	3	1
	-	1	1	1	4	1	2	3	2	2
	-	-	2	2	2	4	1	1	1	3
	-	-	-	3	3	3	4	4	4	4
L_t	1	1	1	1	2	3	2	3	1	1
	-	2	2	2	3	2	3	1	2	2
	-	-	3	3	1	1	1	2	3	3
	-	-	-	4	4	4	4	4	4	4

Figure 5.6 Stack and Priority List Updating: OPT Replacement.

ω	1	2	3	4	1	2	3	2	3	1	
S_t	1	2	3	4	1	2	3	2	3	1	
	-	1	1	1	4	1	2	3	2	2	
	-	-	2	2	2	4	1	1	1	3	
	-	-	-	3	3	3	4	4	4	4	
L_t	1	1	1	1	2	3	2	3	1	1	
	-	2	2	2	3	2	3	1	2	2	
	-	-	3	3	1	1	1	2	3	3	
	-	-	-	4	4	4	4	4	4	4	
d_p	∞	∞	∞	∞	2	3	4	2	2	3	C(m,ω)
c_1	0	0	0	0	0	0	0	0	0	0	10
c_2	0	0	0	0	1	1	1	2	3	3	7
c_3	0	0	0	0	0	1	1	1	1	2	5
c_4	0	0	0	0	0	0	1	1	1	1	4
c_∞	1	2	3	4	4	4	4	4	4	4	-

Figure 5.7 Calculating Cost Function.

5.3.5 The Extension Problem

When a demand-paging algorithm processes a reference string ω in real memory with m frames, and only information available at the time of faults may be recorded, can we predict performance in a real memory with m + k frames? The answer to this *extension problem* is yes only if the paging algorithm is a stack algorithm. Such an algorithm allows one to predict the performance of a memory management system before actually installing additional storage by recording information only at fault times, not at each reference.

At each fault, record a pair of virtual page numbers (p_i, q_i), where p_i is the page referenced that caused the fault and will be brought into real memory, and q_i is the page that will be removed. Also record the priority lists at each fault that were used to determine which page was to be replaced. Then, with this information we can maintain the stack segment $[s_{m+1}, \ldots, s_{m+k}]$.

Given $[(p_i, q_i)]$, construct a new sequence by considering two cases as follows:

1. The stack distance for p_i exceeds m + k. Hence a fault will also occur in m + k frames. Using the priority list recorded at fault i, update the stack segment to find the page q_i' that will be removed from the larger memory. Emit (p_i, q_i') into the new sequence. See Figure 5.8(a).
2. The stack distance for p_i lies in the range [m + 1, m + k]. Then no fault will occur in the larger memory. Simply update the stack segment and emit no entry into the new sequence. See Figure 5.8(b).

To predict paging behavior in four frames, we need to maintain only the stack element in position 4, $s_4(\omega)$. Each initial fault is unchanged in the new sequence, but at the pair (3, 4) we recognize that $s_4(\omega) = 4$ and no fault results from this string in four frames. We correctly predict that four faults will occur in the larger memory.

Clearly, the stack segment is necessary for the operation of this procedure. If the paging algorithm does not have the stack property, the problem cannot be solved with only this amount of information.

5.4 Working Sets

In a multiprogramming system we have conflicting policies for process and memory management. To maximize throughput we should have as many active processes as possible to keep processor and device utilization high. To minimize page-fault overhead, each process should have as much real memory as possible. If we limit the number of active processes too much, throughput can suffer; if we do not control them enough, thrashing can result. Working sets can help determine the optimum point between policies for process and memory management.

5.4.1 Definition of Working Set

The *working set* with parameter Δ for a process at time t, $W(t, \Delta)$, is the set of pages that have been referenced in the last Δ time units [Denning, 1968; Denning, 1970]. Δ is the *window size*, and it is a tuning parameter. Since Δ is a time measure, and we are considering multiprogramming systems, Δ must measure process time rather than real time. If Δ is large enough so that the working set contains pages being frequently accessed, and small enough to contain no more, the working set is a reliable way to obtain a description of the needs of a process. Observe that $|W(t, \Delta)|$ can vary over time. If a process executes for Δ time units and uses only a single page, then $|W(t, \Delta)| = 1$. Working sets can also grow as large as the number of pages n of a process if many different pages are rapidly addressed.

The importance of working sets is that they link memory and process management via the *Working Set Principle:*

> A process may execute only if its working set is resident in main memory. A page may not be removed from main memory if it is in the working set of an executing process.

This means that, if the working set of a process contains those pages currently needed, frequent page faulting is eliminated. The working set of a process may not be affected by the execution of other processes. By limiting the number of active processes to a group whose working sets will fit in real memory, memory overcommitment is avoided and thrashing is eliminated. The working set strategy is a local policy that allows the number of page frames used by executing processes to vary.

If a process has run long enough to have established a working set, and is then blocked or otherwise suspended from execution, its working set can be expected to

disappear from real memory. When reactivated, the Working Set Principle dictates that all of its former working set should be prepaged back before execution resumes. If care is taken, the transfer of a working set into real memory can be much less costly than moving it using the normal faulting mechanism page by page.

5.4.2 Properties of Working Sets

Working sets exhibit some interesting properties that make them attractive for memory management. First, as mentioned, the size of a working set can vary. Specifically, $1 \leq | W(t, \Delta) | \leq \min(\Delta, n)$. This means we can avoid allocating a fixed partition of real memory for a process.

In addition, working sets are inclusive: $W(t, \Delta) \subseteq W(t, \Delta + 1)$. This is similar to the inclusive nature of stack algorithms; working set management will not exhibit Belady's anomaly.

Finally, a graph of working set size versus time will show that, for many programs, periods of relatively constant sizes alternate with periods of much larger sizes. This reflects periods of fairly constant locality followed by rapid changes to a new locality. Since Δ is constant for the process, the working set size temporarily increases as new pages rapidly join the working set. Later, as pages no longer in use leave the working set (also at a rapid pace), the working set size falls to another stable level. Figure 5.9 shows a typical graph of working set size. This phenomenon can also be seen in a plot of page number referenced versus time; see Figure 5.10 [Hatfield and Gerald, 1971] for an example. The rectangular areas show that for long periods of time (horizontal dimension) a well-defined subset (vertical dimension) of pages is being referenced. See also that the shifts to new localities occur

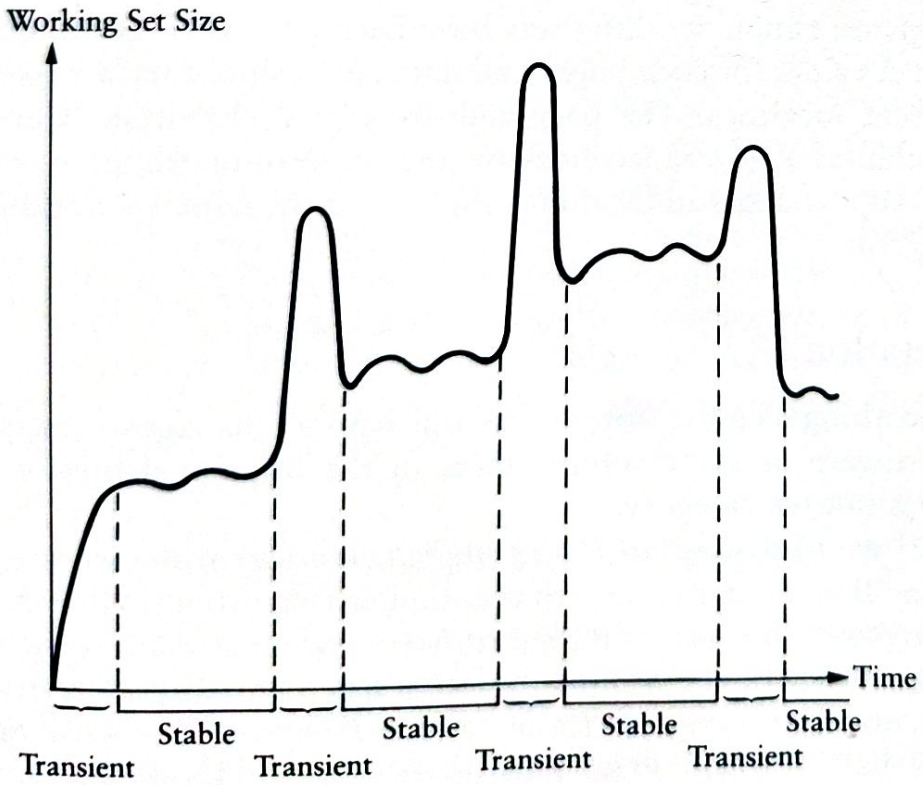

Figure 5.9 Typical Graph of Working Set Size.

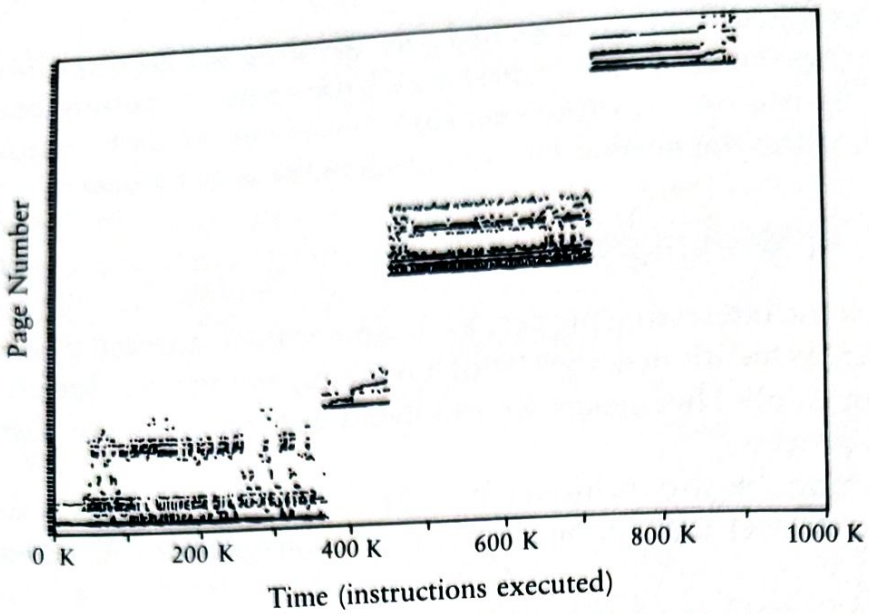

Figure 5.10 Working Sets: Page Referenced vs. Time. [Copyright © 1971 by International Business Machines. Reprinted by permission.]

quickly. In addition, the relationship between the mean working set size and the page-fault rate is established in Problem 10.

A study of page faults under the working set discipline disclosed some interesting phenomena [Kahn, 1976]. Stable phases covered nearly all the process time (98%). Nearly half of the faults occurred during the other 2% of process time while phase transitions were in progress. Fault rates during transitions were 100 to 1000 times higher than during stable phases. Finally, the observed phases were relatively insensitive to the choice of working set window size Δ.

Several variations on simple working sets have been proposed. Denning proposed using different Δ values for code pages and data pages since they are likely to exhibit totally different localities. The page-fault-frequency algorithm [Chu and Opderbeck, 1972] achieves a reverse feedback by altering Δ to match a target fault rate. Dealing with abrupt changes in locality is the goal of the damped working set algorithm [Smith, 1976].

5.4.3 Implementation

The realization of a working set policy for process and memory management requires either additional hardware or some compromises in the original definition. We discuss representatives in each category.

In Morris [1972] we find a design for an implementation of the working set concept in the Maniac II computer system that is similar to a proposal in Denning [1968]. In addition to page table base and limit registers and an associative memory to implement paging efficiently, the Maniac II design has a *working set register*, a *page-frame register* associated with each frame, and a *T-register*. The working set register is a bit map that identifies which page frames hold pages of the running process. Each page-frame register contains a counter that is periodically incremented

if the associated working set register bit is set; the counter associated with a nonrunning process's page is not incremented. When a counter in a page-frame register overflows, an *alarm bit* in the page-frame register is set to indicate that the page in this frame is no longer in the working set of the process. Each time a page in a frame is referenced, the counter in that page-frame register is reset to zero (the alarm bit is also reset). The T-register controls the frequency of updating counters. At a page fault, the frames with alarm bits set indicate pages eligible for removal from real memory. Process switching requires replacing the contents of the working set register but not alteration of any page-frame registers.

An approximation to the working set policy as described here was implemented in an extant time-sharing system [Rodriguez-Rosell and Dupuy, 1973]. The IBM 360/67 computer system provided use bits with each page frame. The quantum of processor time is divided into subintervals of T time units corresponding to a working set window size. At the beginning of an entire quantum all use bits are reset. After each subinterval pages in frames not referenced are marked as candidates for replacement by setting the *absent bit* in the appropriate page-table entries, and then all use bits are again reset. If the process subsequently references such a page before its actual removal, a page fault results but the only effect is the return of the page to the working set of the process. Accurate accounting for the sizes of working sets is complicated by the special handling of information in shared pages outside of this mechanism. However, the installation of this scheme successfully eliminated the thrashing phenomenon.

5.5 Models of Virtual Memory

Models that represent the behavior of programs in virtual memory have been of interest to researchers since the development of virtual memory itself. They have value in testing proposed modifications to systems, for capturing essential characteristics of workloads, and for understanding the effects of virtual memory management policies. A good model is easier to work with than long traces of actual executions. In Spirn [1977] we see a dichotomy between classes of models. One class, called *extrinsic models,* uses information about observed behavior to construct a description that is representative of more particular instances. The other class, *intrinsic models,* uses what is known about the internal nature of program behavior to build a mathematical structure that displays results similar to observed behavior. If one approach is used, the other can be used to validate the first model. We will discuss one virtual memory model of each class.

5.5.1 An Extrinsic Model—Lifetime Curves

This simple description of a program's behavior is generated by processing a reference string for a range of memory sizes m to obtain *mean lifetimes* L(m). A lifetime is a period of execution that is uninterrupted by a page fault, measured in process time or in number of main memory references. If the residence set has a variable size, as in the working set method, other parameters (such as the working set window

size Δ) are manipulated to vary mean residence set size (see Problem 9). The curve L(m) versus m is the *lifetime curve* and the *fault rate* is 1/L(m).

Ideally, a lifetime curve is S-shaped, as shown in Figure 5.11. For small memory sizes, faulting is frequent and lifetimes are short. As available real memory increases, lifetimes increase as expected. However, above a certain threshold the increase in mean lifetime grows less rapidly. The "primary knee" of the concave-down portion of the curve is the point where L(m)/m is maximal. The knee phenomenon has been attributed to two sources [Spirn, 1977]:

1. A reasonable explanation is the concept of locality. After a sufficient number of frames is allocated, a further increase in real memory has a small effect on fault rates and hence on lifetimes. It has been proposed [Denning and Kahn, 1975] that the primary knee corresponds to the average size of localities that span stable phases in residence set composition, and that this real memory allocation is optimal.
2. When a page is first referenced, it always causes a fault. For a reference string of length k that touches n distinct pages, the lifetime curve L(n) is approximated by k/n. When real memory allocation approaches n page frames, the mean lifetime is strongly influenced by the length of the reference string. This may lead one to conclude that, beyond the primary knee, a lifetime curve is an unreliable model of program behavior. Although we may not be very interested in this area of the curve, studies have shown that initial page loads can be a major contributor to the existence of the primary knee [Carr, 1984].

Nevertheless, a lifetime curve is easy to generate from an actual program trace and therefore it has the advantage of being derived from real data.

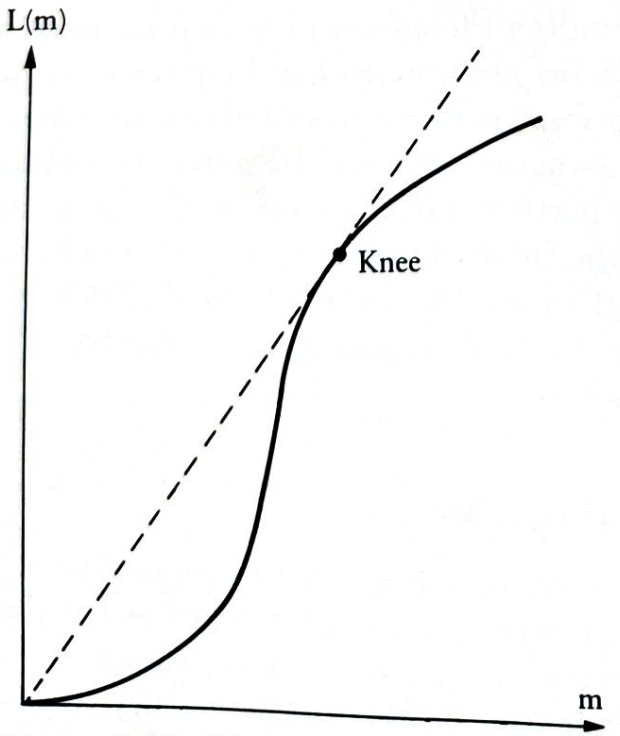

Figure 5.11 Ideal Lifetime Curve.

5.2.2 An Intrinsic Model—LRU Stack

The *LRU stack model* of program behavior arose because the LRU replacement policy was seen to give good performance. It is a relatively simple intrinsic model, and the stack-updating procedure is easy to understand (see Section 5.3.3). Its origins are found in Shemer and Shippey [1966], and in Shemer and Gupta [1969].

In this model we assume that successive stack positions p_{t-1} (r_t) at which the references r_t occur are independent random variables drawn from the probability distribution

$$\text{Prob}[p_{t-1}(r_t) = i] = a_i$$

with cumulative distribution

$$A_i = \text{Prob}[p_{t-1}(r_t) \leq i] = \sum_{j=1}^{i} a_j$$

Given an initial stack s_0 (often assumed to be $[1, 2, \ldots, n]$), we obtain a sequence of stack distances from the distribution above and, from that, generate a reference string. Strings generated this way are called *LRU reference strings*.

LRU reference strings can be constructed that exhibit statistical behavior similar to that of actual programs. If $a_k = 1$, $a_i = 0$ for $i \neq k$, a looping reference string results:

$$\omega = k, k-1, k-2, \ldots, 2, 1, k, k-1, \ldots$$

If $a_1 \geq a_2 \geq \cdots a_n$, then stack positions near the top are favored. In this case, we are not surprised that the LRU replacement policy is optimal [Coffman and Denning, 1973]. One could empirically obtain estimates of these probabilities by processing reference strings of interest and observing stack positions.

The LRU stack model, along with some other models, was validated in Spirn [1977] by comparing its behavior with that observed when actual reference strings were processed with a working set method. The needed probabilities for stack positions were obtained empirically by observing those that actually occurred. The reference strings were all short enough so that they do not include phase transitions. Within a single locality, the LRU stack model was seen to be the most acceptable model of the program's virtual memory behavior.

5.6 Clock Algorithms

Versions of replacement algorithms have long existed that resemble the simple versions of the methods in this section. Recent work has shown that they can be enhanced to provide good alternatives to purely local policies [Carr and Hennessey, 1981; Carr, 1984]. This section is based on that research. Clock algorithms implement forms of global-replacement policies. Envision the page frames of real memory arranged circularly, as on a clock face. A pointer travels clockwise among the frames. Whenever a page must be selected for replacement, the pointer is advanced to the

next frame containing a page that satisfies some particular criterion. If no criterion at all is used, the next page in turn is selected for replacement, implementing the global FIFO method. By using the following scheme, global LRU is approximated. When a page frame is examined, its use bit is checked and then cleared. If the page has been referenced, the pointer is advanced to the next frame. If not, the page is replaceable, and the pointer is left at the following frame. If frames remain unused with equal probability for a given length of time over all frames, the clock algorithm will find the highest density immediately ahead of the pointer, so the pointer should not have far to travel.

When a page has been deemed replaceable by the criterion in effect, it is processed differently if its dirty bit is set. In that case the page is queued for cleaning and the clock scan proceeds to the next page. Usually the page has been cleaned by the time it is considered again, and is then replaced. This generates a relatively even stream of requests to clean pages and avoids long waits for page writes followed by reads. See Figure 5.12 [Carr and Hennessey, 1981]; we refer to this as the Clock algorithm.

5.6.1 A Working Set Approximation—WSClock

The clock algorithm framework can be supplied with different criteria in order to approximate the replacement methods described in this chapter; we describe one here, the *WSClock algorithm*. This algorithm approximates a working set method for pages of processes in combination with global replacement by updating information about pages as the pointer passes over them. As a frame is considered, its use bit is tested and reset as usual. If the bit was already set, its time of last reference is assumed to be the current process time PT and is stored in association with that frame in LR[f]. If the bit was not set, the page is replaceable if PT − LR[f] > T, the working set window parameter.

This method has two inaccuracies. First, the estimate of the last reference is more accurate when the pointer is rapidly moving among the frames. Second, the algorithm operates on resident pages (frames, actually) instead of all pages. It is, however, a low-overhead approximation to the standard working set method.

Standard working set schemes do not distinguish among replaceable pages. By approximating it in the clock algorithm framework, the replaceable pages are ordered in a manner that approximates LRU. Also, no data structure is needed to keep track of pages not in working sets; they are found as needed.

5.6.2 Load-Control Methods

Since the clock algorithm in all its forms is still a global replacement scheme, load control is important to avoid memory overcommitment. The method in the next section applies to both WSClock and Clock mechanisms. In the sections after this, specific techniques for each are described.

LT/RT Load Control

Carr describes a method called *loading-task/running-task* load control (LT/RT). This method distinguishes between *loading tasks*, which have few resident pages that will

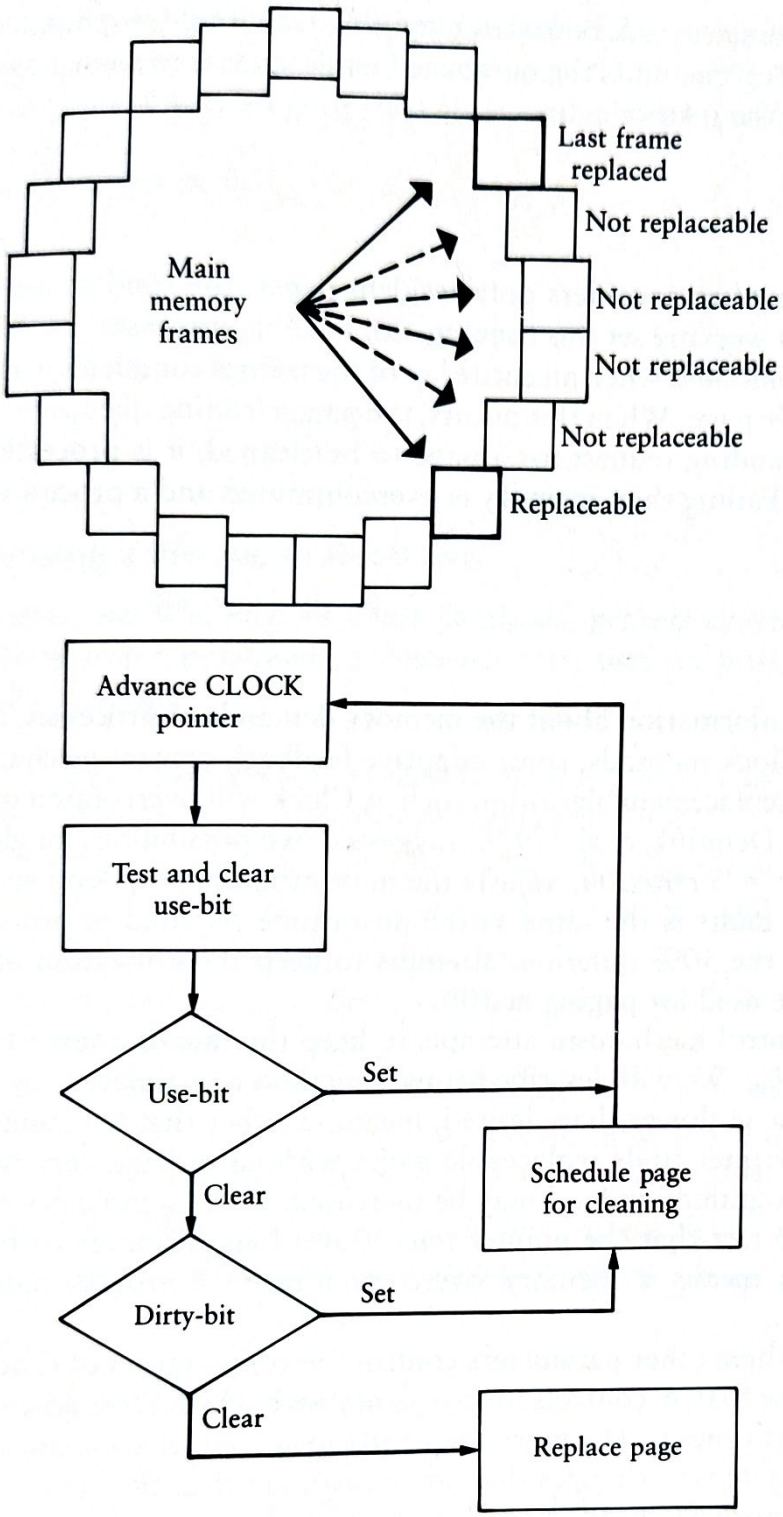

Figure 5.12 Clock Algorithm [From Carr and Hennessy, 1981. Copyright © 1981 by Association for Computing Machinery. Reprinted by permission.]

fault frequently as needed pages are referenced, and *running tasks*, which have needed pages resident and fault infrequently. It is used instead of prepaging the resident sets of newly activated tasks. LT/RT uses a parameter τ to distinguish between the classes. A newly activated task is a loading task initially. As long as its lifetimes (intervals between successive faults) do not exceed τ, it remains a loading task. Whenever a

lifetime longer than τ occurs, the task becomes a running task until it terminates or is deactivated. The LT/RT regime limits the number of loading tasks to some number L, a second parameter of the method.

WSClock Load Control

Since the WSClock mechanism considers only resident pages, the resident set of a process is defined to be its working set (for running, not loading, processes). WSClock detects *memory overcommitment* when an entire lap of the frames completes without encountering a replaceable page. When this occurs, the page-cleaning queue is examined. If there is an outstanding request for a page to be cleaned, it is processed to yield a replaceable page. Failing that, memory is overcommitted and a process must be deactivated.

Clock Load Control

In the absence of direct information about the memory demands of processes, as in the working set or WSClock methods, some adaptive feedback control mechanism is necessary, or a global replacement algorithm such as Clock will overcommit memory and cause thrashing. Denning, et al. [1976] suggested two possibilities for global policies. The first, the $L = S$ *criterion,* adjusts the multiprogramming level so that the mean time between faults is the same as the mean time required to process a page fault. The second, the *50% criterion,* attempts to keep the utilization of the auxiliary memory device used for paging at 50%.

The Clock load-control mechanism attempts to keep the rate of pointer travel C near an optimal rate C_0. We will describe below a method to estimate C by $\hat{C}$. If $\hat{C} < C_0$, the pointer rate is slower than desired, meaning either that few faults are occurring or that the pointer finds replaceable pages without moving very far. In either case, the multiprogramming level may be increased. $\hat{C} > C_0$ indicates either a high fault rate or the fact that the pointer must travel long distances to find a replaceable page. This means a memory overcommitment; a process must be deactivated.

In addition to C_0, three other parameters control the computation of $\hat{C}$ and its comparison with C_0. The first, δ, controls the frequency with which the load-control mechanism is invoked at times t_i. The more frequently load control is executed, the more responsive the system is to changes that are needed, but then the load-control algorithm itself causes more overhead. A small δ also introduces more variability in successive estimates. Each time, an estimate of clock pointer rate, called c_i, is computed. The second parameter, α, is the exponential smoothing weight used to combine c_i with $c_{i-1}, c_{i-2}, \ldots$ into a current average $\hat{C}_i$. A large α puts more weight on the latest c_i, allowing faster response to changes, but it also contributes to variation in the estimate. The third parameter, ϕ, governs how the comparison between C_0 and $\hat{C}_i$ is done. $\hat{C}_i$ is acceptable if it lies in $[C_0 - \phi, C_0 + \phi]$; no changes in multiprogramming level are necessary. A small ϕ results in quicker system response but incurs the cost of frequent process activation and deactivation.

An exponential smoothing method is used to compute $\hat{C}_i$ because it incorporates all previous interim measures c_i and requires no explicit storage of previous measures. At each execution of the Clock load-control mechanism, compute

$$Y_i = c_i + \alpha Y_{i-1}$$

and

$$Z_i = 1 + \alpha Z_{i-1}$$

with $Y_0 = Z_0 = 0$. Then

$$\hat{C}_i = Y_i / Z_i.$$

Choosing a Process to Deactivate

In either the WSClock or Clock methods, *process deactivation* must occur when memory overcommitment is detected. Carr lists six possibilities (in addition to a random choice):

1. The *lowest priority process* [Denning, 1980]: This implements a policy decision and is unrelated to performance issues.
2. The *faulting process* [Fogel, 1974]: This is an intuitive idea, since the faulting process may not have its working set resident, and deactivating it eliminates the need to satisfy this page request.
3. The *last process activated*: This is the process least likely to have its working set resident.
4. The *smallest process*: This will require the least future effort to reload.
5. The *largest process*: This obtains the most free frames in an overcommitted memory, making additional deactivations unlikely soon.
6. The *largest remaining quantum process*: This approximates a *shortest processing time first* scheduling discipline.

5.6.3 Simulation Results

In Carr [1984], we find an extensive description of a simulation model for virtual memory designed to compare the many alternative management policies. The reference strings from executing eight heavily used programs were captured and processed for simulator input. We report results germane to the discussion in this section.

Process Deactivation Policies

No differences were observed among selecting the process with the smallest resident set, the one most recently activated, or the one with the largest remaining quantum. The other alternatives exhibited inferior performance.