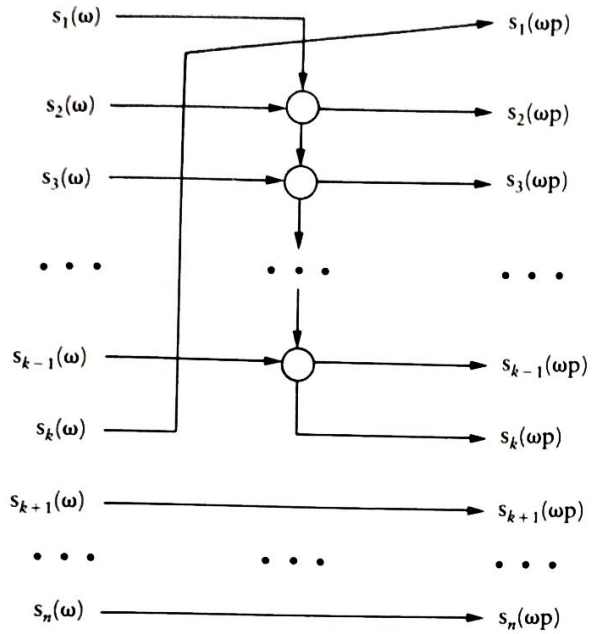


Figure 5.5 Stack Updating Procedure: Priority Operator.

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).

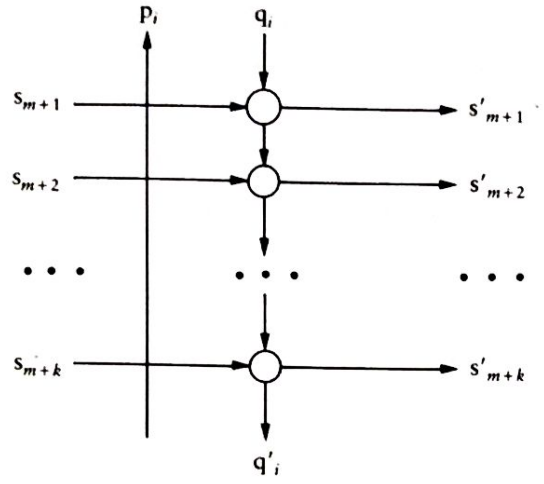


Figure 5.8(a) Extension Problem: Fault Occurs in Larger Memory.

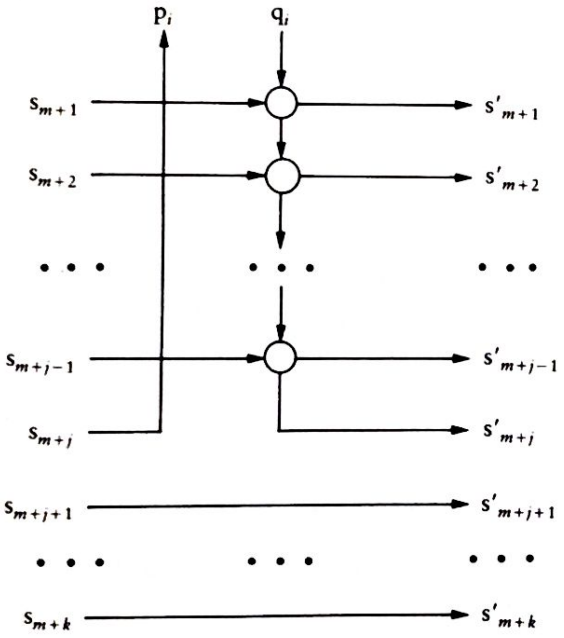


Figure 5.8(b) Extension Problem: No Fault in Larger Memory.

As an example, if we recorded the sequence from the example in Figure 5.7 along with priority lists (using three page frames), we would have obtained:

(p_i,q_i)	(1,−)	(2,−)	(3,−)	(4,3)	(3,4)
L_{i-1}	−	1	1	1	3
	−	−	2	2	2
	−	−	−	3	1
	−	−	−	−	4

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

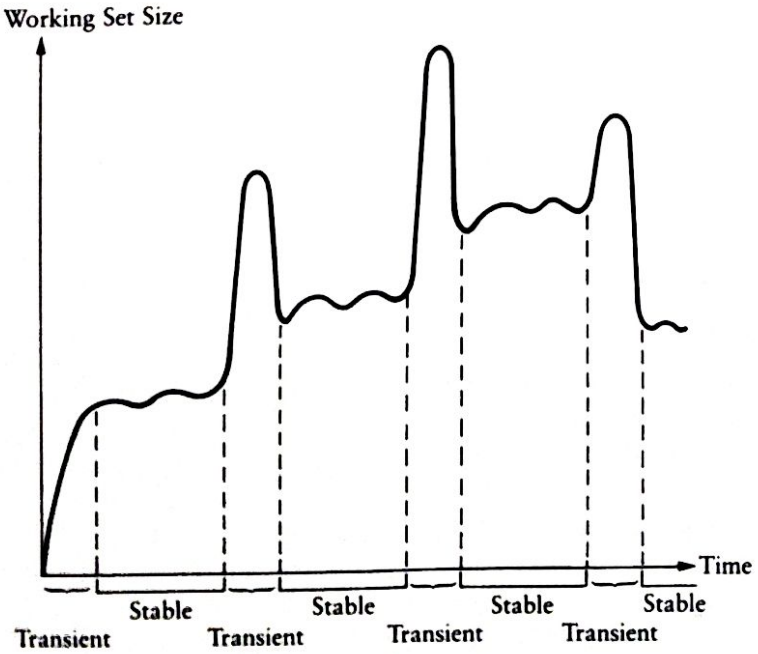


Figure 5.9 Typical Graph of Working Set Size.

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

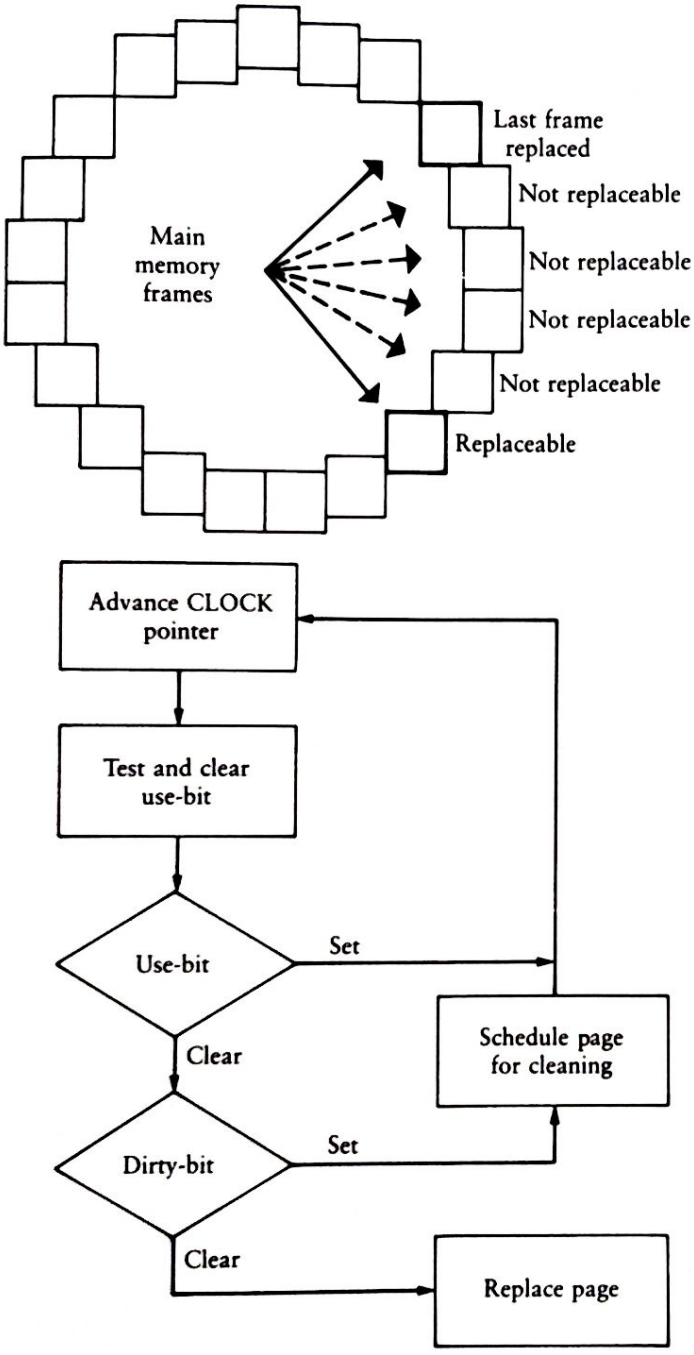


Figure 5.12 Clock Algorithm [From Carr and Hennessy, 1981. Copyright © 1981 by Association for Computing Machinery. Reprinted by permission.]

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.

LT/RT Control

The parameter τ should not be too small, or a process will be deemed running before its resident set has been acquired; if τ is too large, efficient processing continues, so the LT/RT control mechanism is robust in that regard.

The number L of loading processes allowed was optimal when L was set to the number of auxiliary memory paging devices in use. This was limited to L = 1 or 2 in the simulations.

Clock Load Control

The mechanism proved very insensitive to the value of the parameters δ, α, and ϕ after tuning the simulation model to an appropriate value for C_0. This does not mean it was ineffective, because it often caused changes in multiprogramming level. Instead, the LT/RT mechanism had a larger effect.

Relative Performance

The WSClock and Clock mechanisms were compared with each other and with an exact implementation of a working set policy. All were independently tuned for their best performance. Carr's simulation showed that all three performed almost exactly as well in maintaining a high utilization (with similar mean multiprogramming levels), as shown in Table 5.1. Several comments will be helpful here. First, the Clock method performed much worse before tuning than the untuned WSExact and WSClock methods. This may indicate that careful tuning is more important for Clock, because its feedback mechanism is less direct than those based on working sets. Second, the Clock algorithm is simpler to implement and should use less system capacity to execute than the others, making it attractive to operating system designers. Third, WSClock performs as well as an exact working set algorithm, and is simpler to implement.

5.7 Program Restructuring

Users of systems that provide virtual memory facilities are encouraged to view virtual memory as a seamless, linear extent of memory without internal structure. However, it is easy to see that the arrangement of program modules in virtual memory determines the page coresidence of the modules, which explains the potential for page

Table 5.1 Comparison of Working Set, WSClock, and Clock Algorithms

Algorithm	Utilization	Mean MPL
WSExact	.6211	5.25
WSClock	.6164	5.01
Clock	.6177	5.27