Virtual Memory

5

5.1 Introduction

Historically, the conservation of main storage has been an important consideration in program design. With only a few thousand bytes available, saving a few hundred could mean the difference between successfully executing a program and failing because too little memory was available. Programmer-designed overlay structures reused portions of storage for different purposes at different points during execution. Hardware advances have made large amounts of inexpensive memory possible, but memory management considerations have not disappeared. And, inevitably, the magnitude of problems for which computer solutions are sought has grown. In place of specific overlay structures for each program, a general-purpose virtual memory system is often provided to users. *Virtual memory* is an illusion supported by system hardware and software that a user has a vast linear expanse of useful storage. In fact, however, a much smaller *real memory* is used to hold portions of a user's program during execution. The efficient preservation of this illusion is the subject of this chapter.

We first review some fundamental concepts. Then we cover a class of virtual memory management methods called *stack algorithms,* which provide solutions to two important problems. This class is also relevant to the remainder of the chapter. An approach to managing storage by linking process management with memory management, called *working sets* is then presented. An introduction to models of virtual memory follows. Next, we discuss *clock algorithms*; noteworthy results have recently been published concerning them. Finally, we introduce the idea of restructuring programs to improve their performance in virtual memory systems.

5.2 Background and Review

In addition to the mythical virtual memory and the actual real memory just mentioned, a third component, the *auxiliary memory,* plays an important role. The operating system and hardware divide virtual memory into *pages* of identical size and divide real memory into *page frames.* Pages reside in auxiliary memory and may, in addition, occupy a page frame in real memory. When an executing program refers to a particular item in virtual memory, the reference proceeds normally if the item

is also *resident* in a page frame. Otherwise, a *page-fault interrupt* occurs to enable the operating system to adjust the contents of real memory to make a retry of the reference successful. A *page table* holds information about the whereabouts of each page; either it supplies the page frame address or it gives the auxiliary memory address. Conceptually, the page table is a vector indexed by the left bits of a virtual address. The right bits provide an offset into the identified page. Both system hardware and system software cooperate to make virtual memory work.

5.2.1 Hardware Support

The most important hardware component in a virtual memory system is an *address translation device* that accepts an effective address generated by instruction execution for a byte in virtual memory and produces a real memory address to that byte in some page frame. If the page table were an ordinary vector stored in real memory itself, the address translation device would be intolerably slow. Instead, special associative registers and caches hold parts of page tables so that the translation is overlapped with other CPU activity to make the translation time acceptable. If the required page is not resident in some frame, the translation device causes the page-fault interrupt to initiate software activity.

Two additional hardware components that are not strictly required are bit vectors with an entry for each page frame in real memory. The first, called the *use bits,* are set whenever pages in the associated frames are referenced. The second, the *dirty bits,* are set whenever pages in the associated frames are targets of memory-modification instructions. Operating system software can use these bits to improve its management activities. If the use bit of a particular frame has remained off for an interval, the content of that frame is a page to which the program has made no recent reference. If the dirty bit is off, then the copy of the page in auxiliary memory is still up-to-date, and the contents of the frame need not be written back. While these bit vectors are not necessary, some discussions in this chapter will assume their existence.

5.2.2 Page-Fault Rate and Principle of Locality

During the course of executing a program, the processor generates a sequence of virtual page references

$$\omega = r_1\, r_2 \ldots r_t \ldots r_T$$

called the *reference string*. To remove the effects of other processes, we view this string as being generated in *virtual time* or *process time,* the time seen by the individual process. We define a *virtual time unit (vtu)* to be the average amount of time required to make a reference to main memory. Let F denote the average amount of time (*vtus*) to service a page fault. If p is the probability of a page fault, then the *effective access time* of the process is defined by

$$EAT = 1 + p * F$$

and the *duty factor* of a process is defined to be

$$DF = \frac{|\omega|}{|\omega| * 1 + p * |\omega| * F} = \frac{1}{1 + p * F}$$

EAT measures the effective speed at which a process executes and *DF* measures the ability of a process to use the processor. For small values of p, *DF* and *EAT* are approximate reciprocals of each other. As an example, suppose that a *vtu* is 1 μs and disk access time is 10 ms. Then $F = 10^4$ and $EAT \simeq 1 + 10^4 * p$. If p is in the range $(10^{-4}, 10^{-3})$, then *EAT* is in the range (2, 11) and *DF* is in the range $(\frac{1}{11}, \frac{1}{2})$.

Reference strings typically range from tens of thousands to millions of references in length and may involve many distinct pages. Based on current technology, a very low page-fault rate p is required for acceptable performance. Even in a multiprogrammed environment, the degree of multiprogramming is typically not large enough to compensate for large values of p. Suppose we require a process to execute in a main memory (or a memory partition) smaller than the program size in virtual memory. Even then small values of p occur in practice because of a property exhibited by most programs called *locality of reference*. This property says that, at any instant, there is a favored subset of all pages that have a high probability of being referenced in the near future. The following example illustrates the *principle of locality*.

Suppose the program statement

for i **in** 1..n **do** A[i] : = B[i] + C[i] **endfor**

is executed in a memory with page size of 1000 words. Let n = 1000. Using a register machine (with registers denoted by Ri), the machine language version of this program, loaded in main memory starting at address 4000, might appear as:

4000	(R1) ← ONE	Establish index register for i
4001	(R2) ← n	Establish n in R2
4002	compare R1,R2	Test i > n
4003	branch greater 4009	
4004	(R3) ← B(R1)	Access B[i] using index register R1
4005	(R3) ← (R3) + C(R1)	Add C[i] using index register R1
4006	A(R1) ← (R3)	Store sum in A[i] using index register R1
4007	(R1) ← (R1) + ONE	Increment i
4008	branch 4002	
	...	
6000-6999	storage for A	
7000-7999	storage for B	
8000-8999	storage for C	
9000	storage for ONE	
9001	storage for n	

The reference string generated by this loop is

$$494944(47484649444)^{1000}$$

consisting of over 11,000 references (but involving only five distinct pages).

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

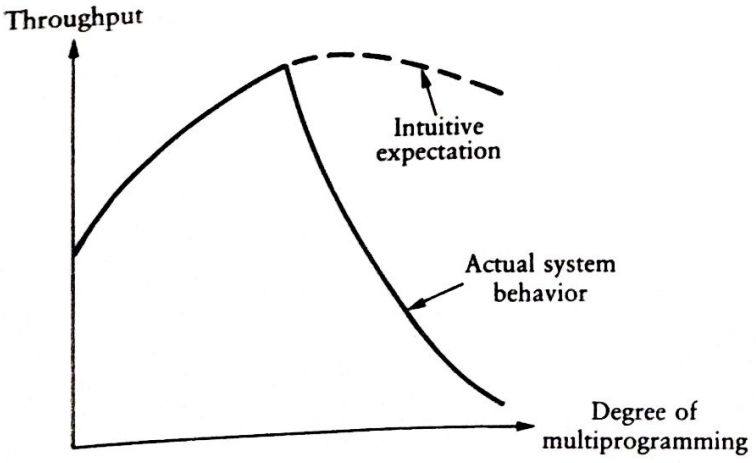


Figure 5.1 Throughput vs. Multiprogramming Level with Thrashing.

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.

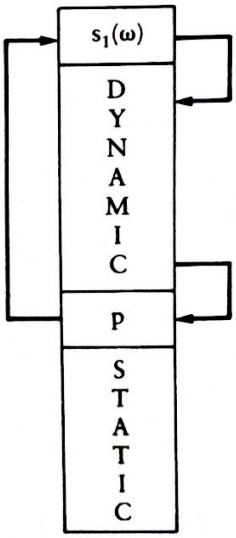


Figure 5.4 Stack Updating Procedure: General Motion.

Priority algorithms are stack algorithms. If memory of size m + 1 contains pages M ∪ {y} (for any page y), then the page to be removed upon a fault is min[M ∪ {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 ω = 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(ω) 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