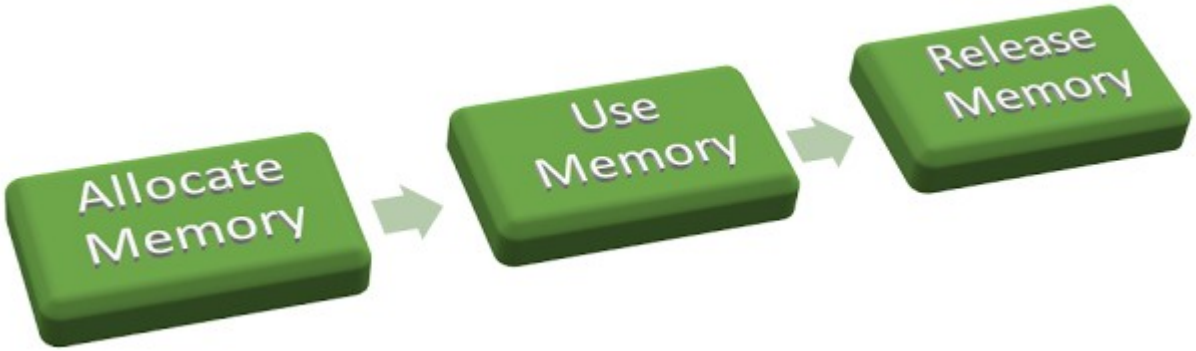
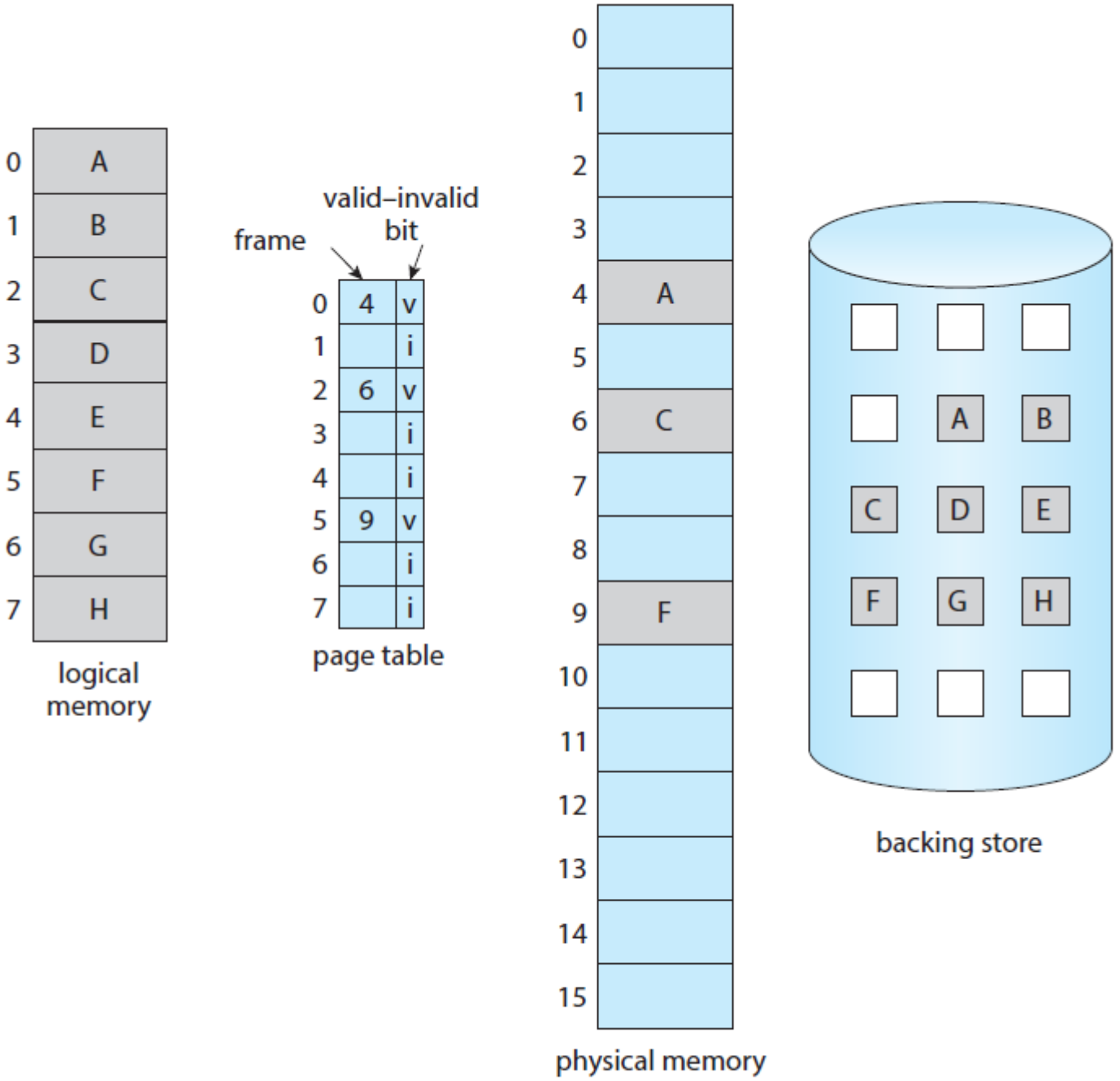
Memory Management. Part 3



MOS4E

https://dzone.com/articles/memory-management-and-garbage-collection-in-javasc



The general concept behind demand paging is to load a page in memory only when it is needed.

As a result,while a process is executing, **some pages** will be in memory, and some will be in secondary storage.

Figure 10.4 Page table when some pages are not in main memory.

OSC10E

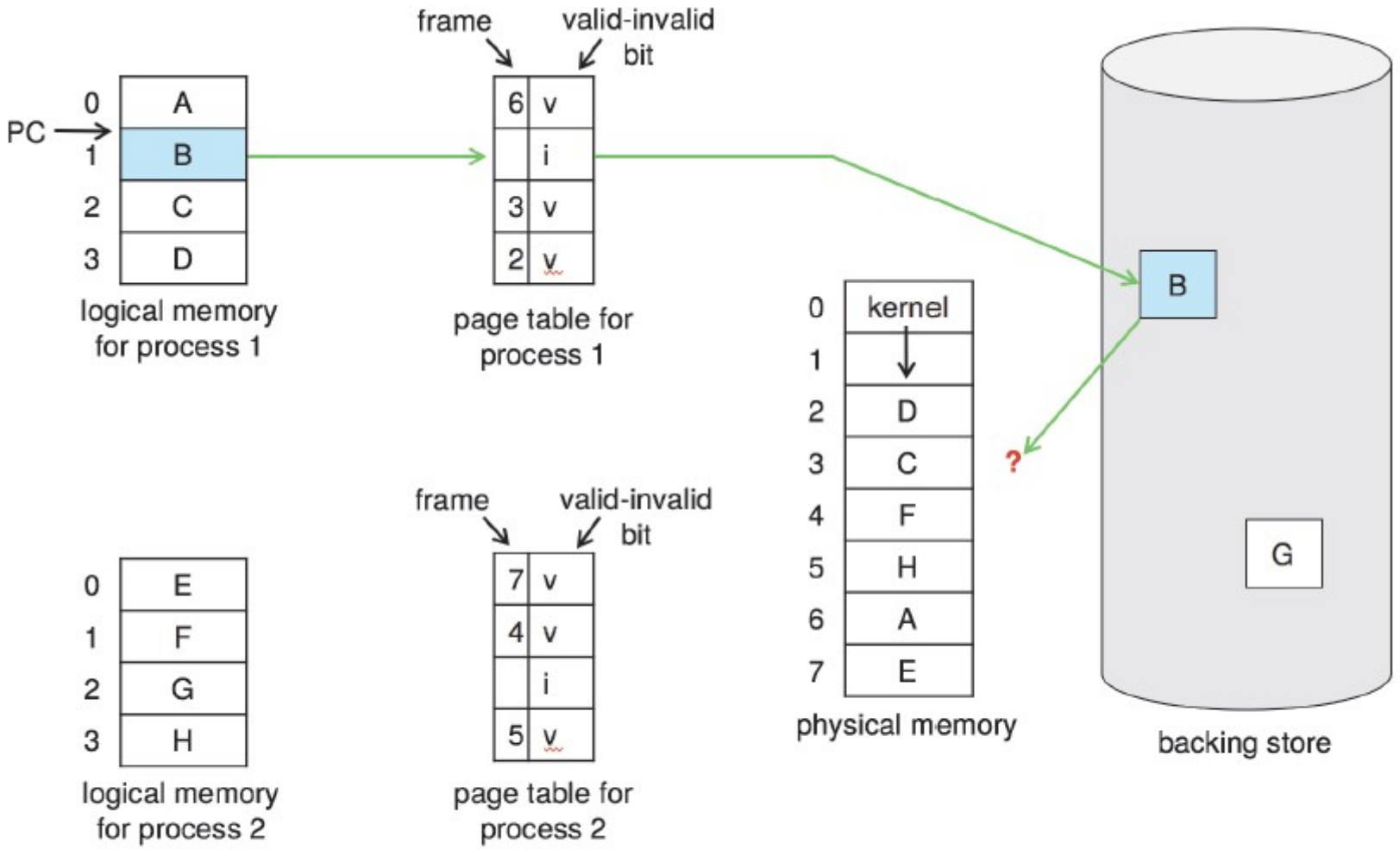


Figure 10.9 Need for page replacement.

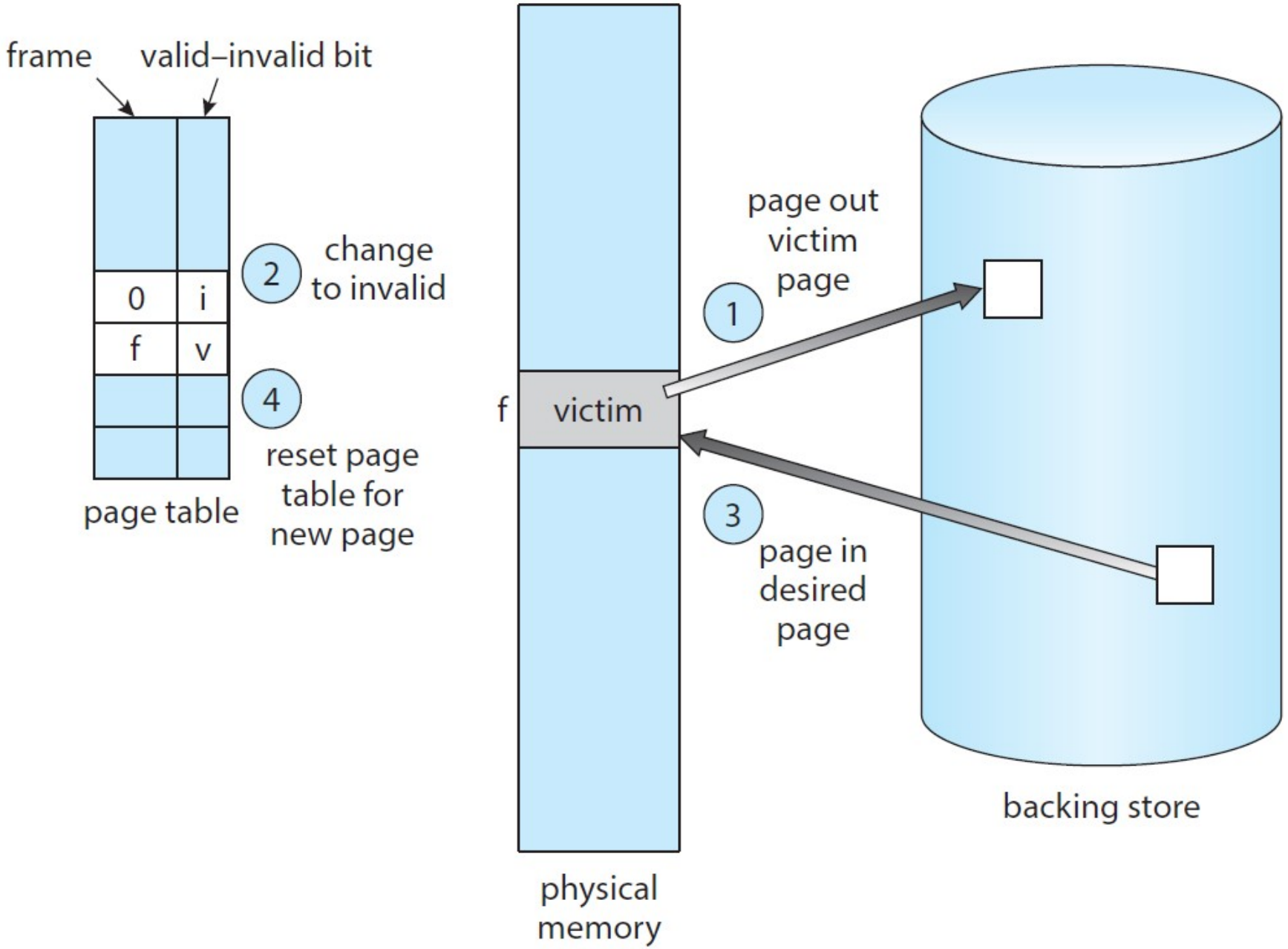


Figure 10.10 Page replacement.

OSC10E

“There are many different page-replacement algorithms.

Every operating system probably has its own replacement scheme.

How do we select a particular replacement algorithm?

In general, we want the one with the lowest page-fault rate.”

OSC10E

“We evaluate a page-replacement algorithm by running it on a particular string of memory references and computing the number of page faults.

The string of memory references is called a reference string.”

OSC10E

“We can generate reference strings artificially (by using a random-number generator, for example),

or we can trace a given system and record the address of each memory reference.

The latter choice produces a large number of data (on the order of 1 million addresses per second).

To reduce the number of data, we use two facts.”

OSC10E

“First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address.

Second, if we have a reference to a page *p*, then any references to page *p* that *immediately* follow will never cause a page fault. Page *p* will be in memory after the first reference, so the immediately following references will not fault.”

OSC10E

“For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1”

OSC10E

“To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available.

Obviously, as the number of frames available increases, the number of page faults decreases.

For the reference string considered previously, for example, if we had three or more frames, we would have only three faults—one fault for the first reference to each page.
”

OSC10E

“In contrast, with only one frame available, we would have a replacement with every reference, resulting in eleven faults.

In general, we expect a curve such as that in Figure 10.11. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.”

OSC10E

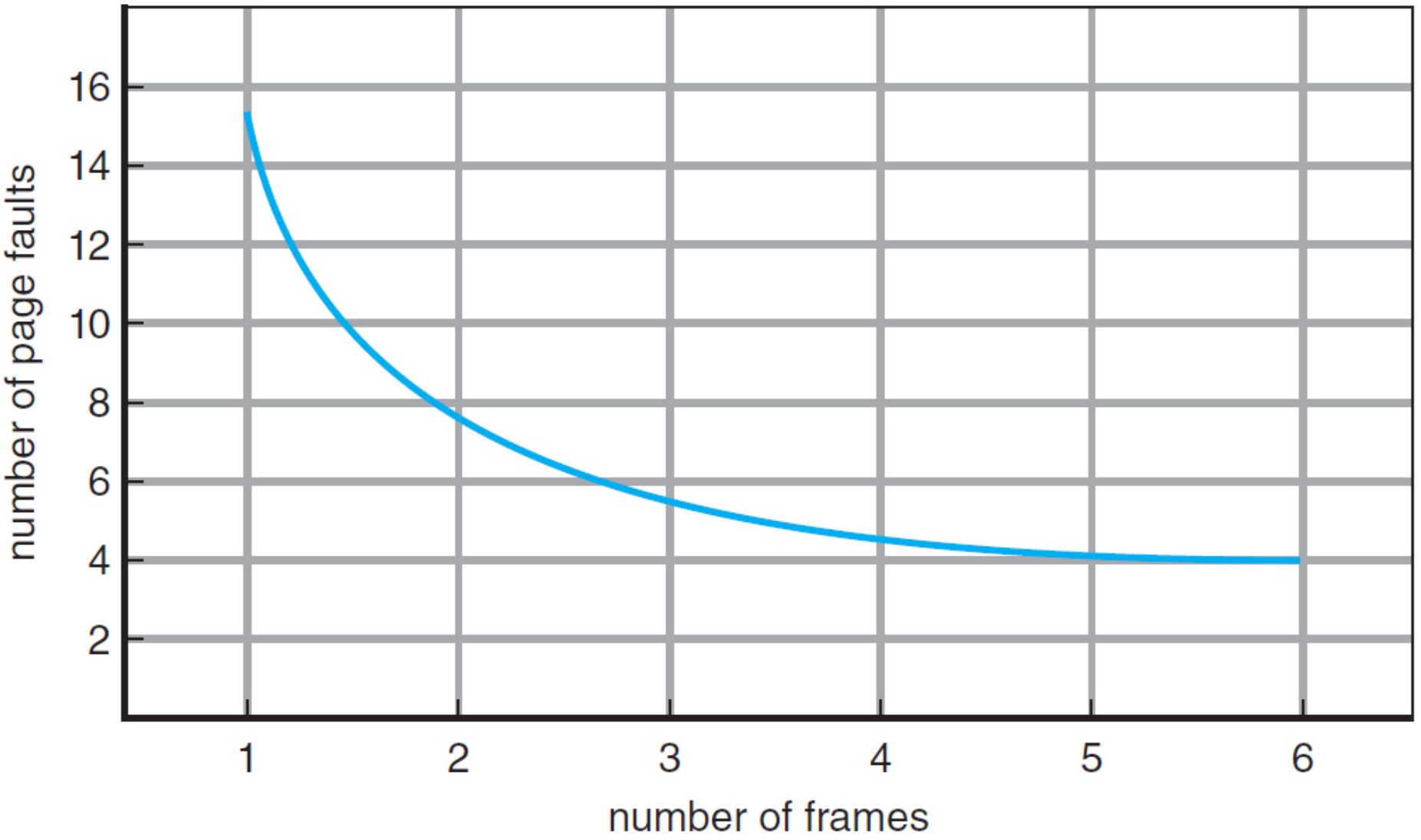


Figure 10.11 Graph of page faults versus number of frames.

The Optimal Page Replacement Algorithm:

Replace the page that will not be used for the longest period of time.

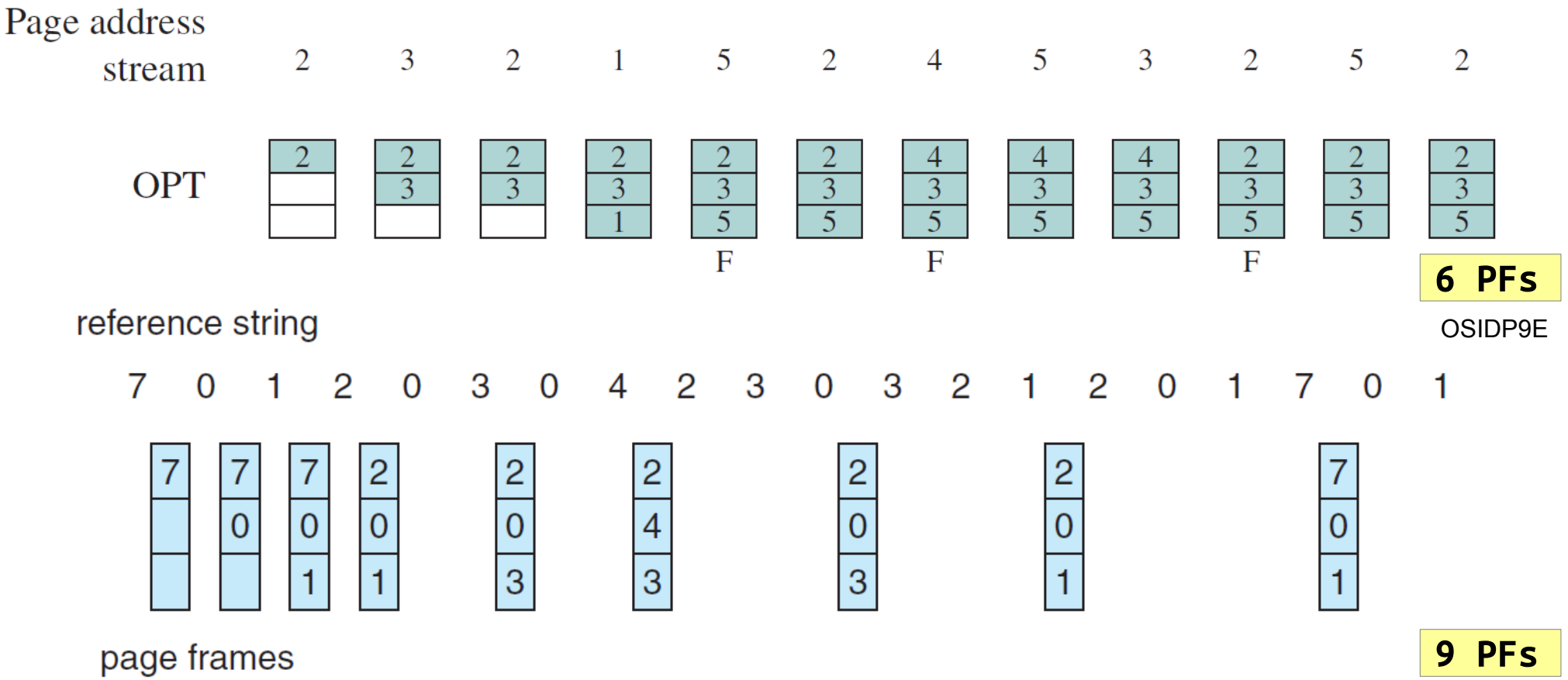


Figure 9.14 Optimal page-replacement algorithm.

OSIDP9E

OSC10E

The Not Recently Used Page Replacement Algorithm

In order to allow the operating system to collect useful page usage statistics, most computers with virtual memory have two status bits, *R* and *M*, associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it.

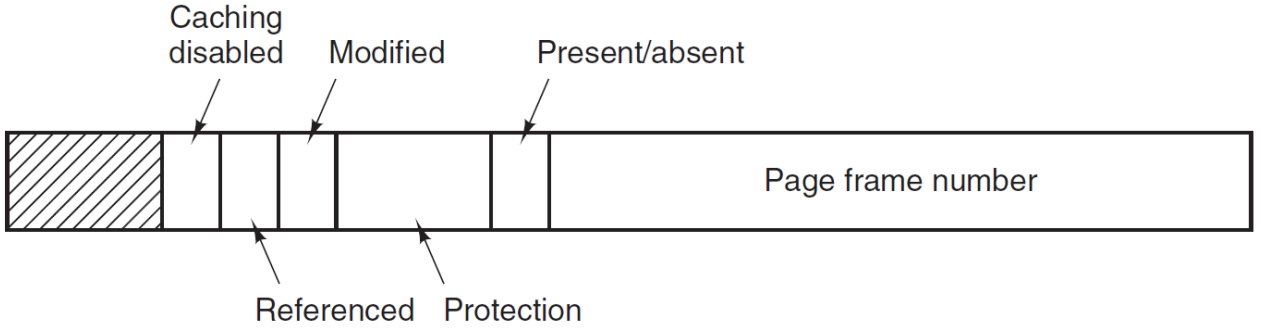


Figure 3-11. A typical page table entry.

MOS4E

The Not Recently Used Page Replacement Algorithm

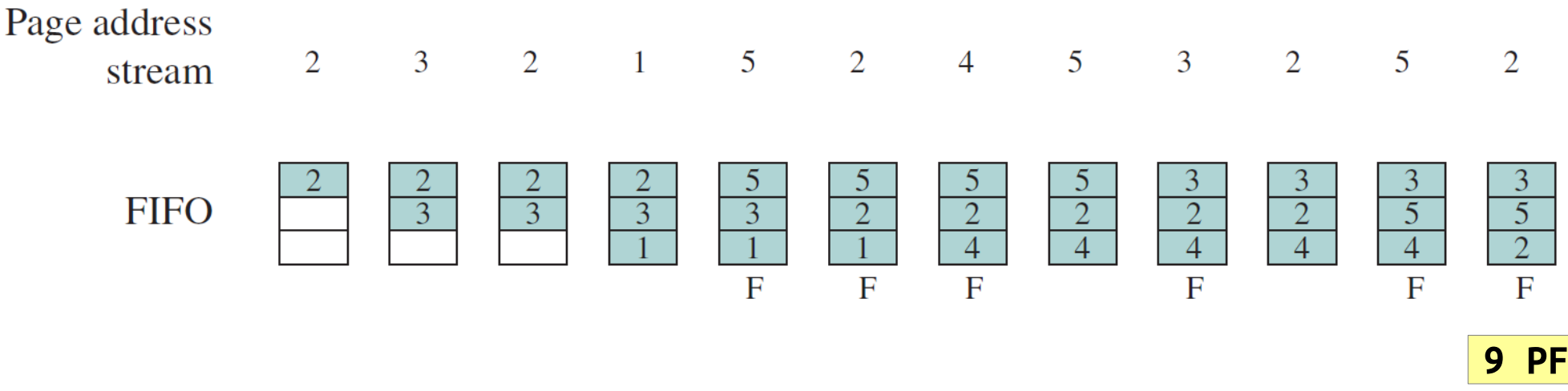
The *R* and *M* bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that have not been referenced recently from those that have been. When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.

The NRU (Not Recently Used) algorithm removes a page at random from the lowest-numbered nonempty class.

MOS4E

The FIFO Page Replacement Algorithm



OSIDP9E

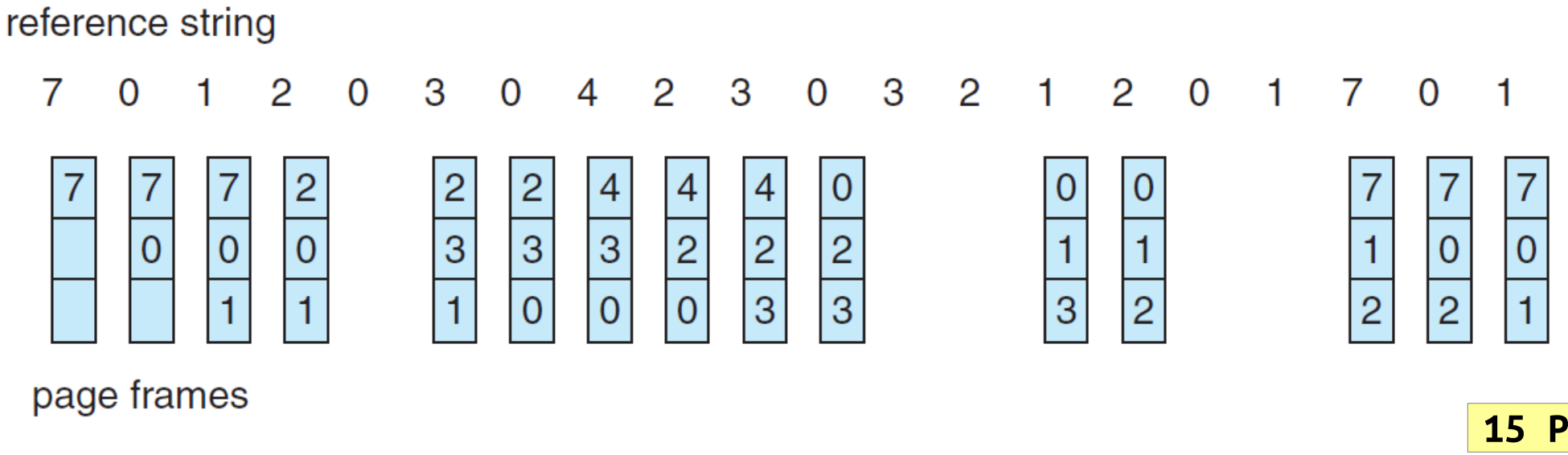


Figure 9.12 FIFO page-replacement algorithm.

OSC10E

Belady's Anomaly

Consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

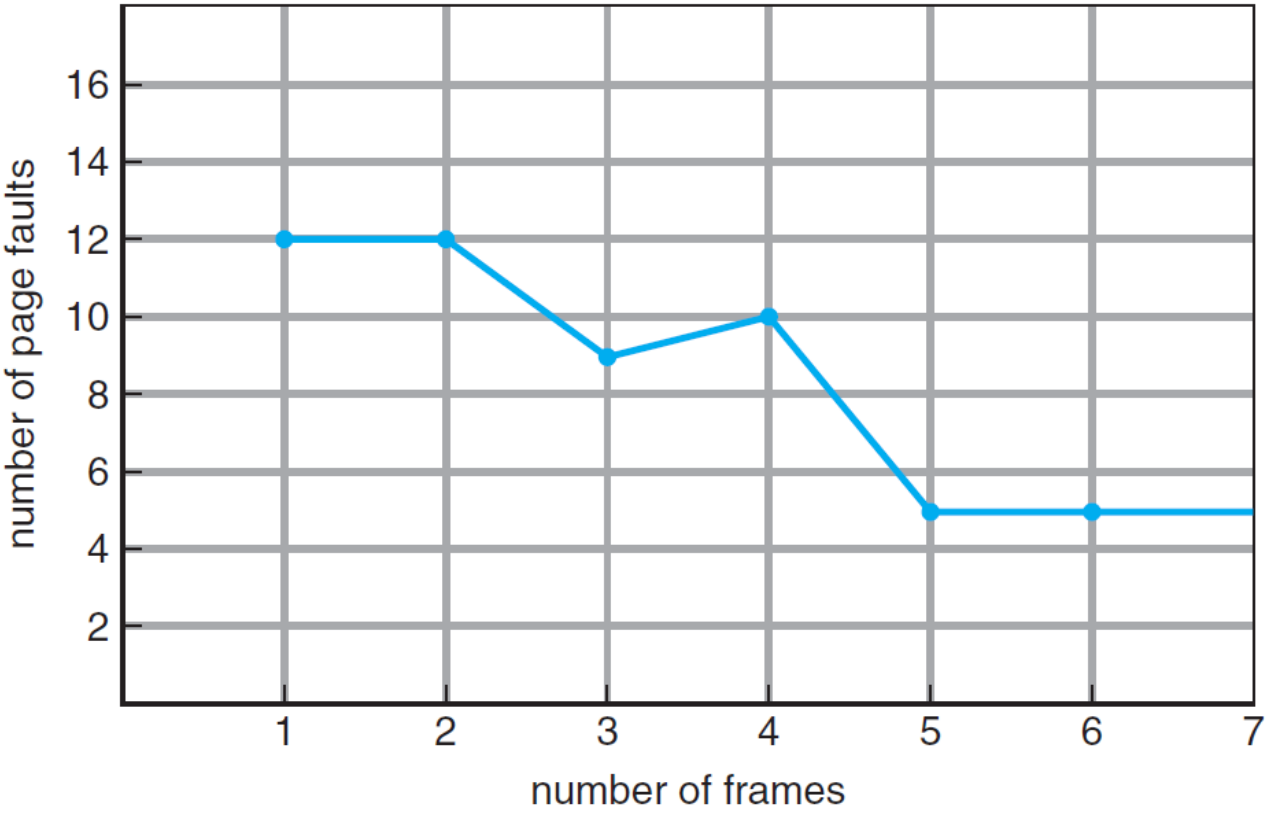


Figure 9.13 Page-fault curve for FIFO replacement on a reference string.

OSC9E

The Second-Chance Page Replacement Algorithm

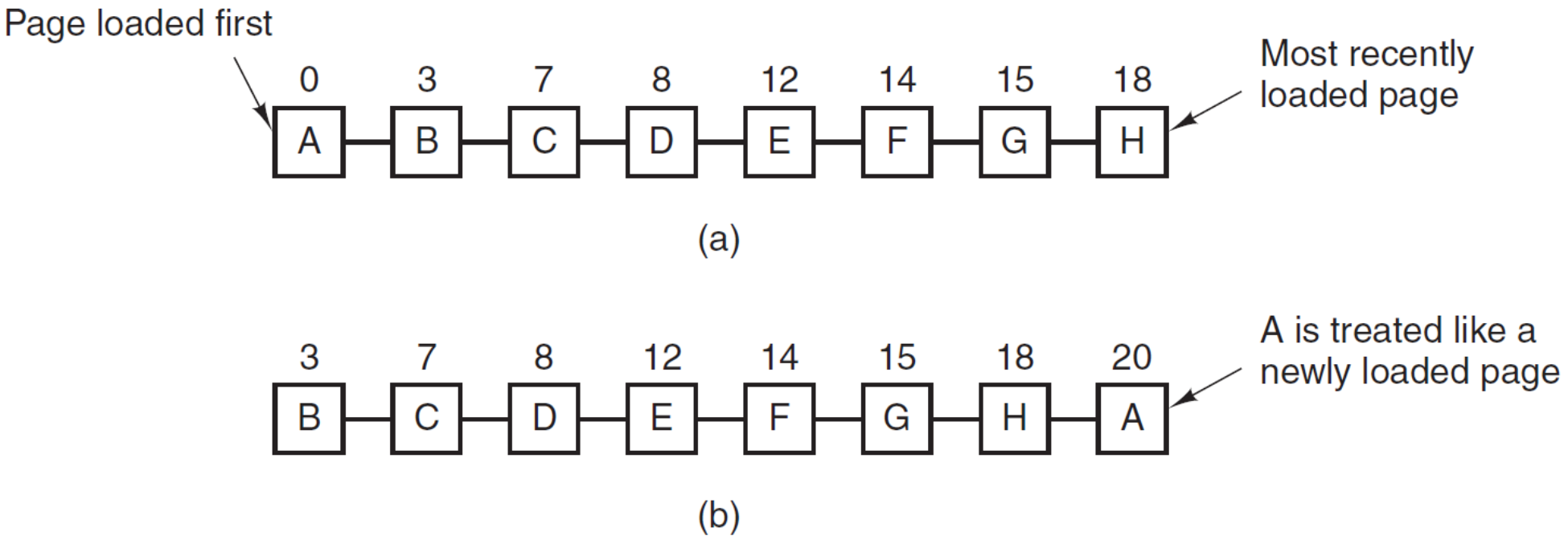


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

MOS4E

The Second-Chance Page Replacement Algorithm

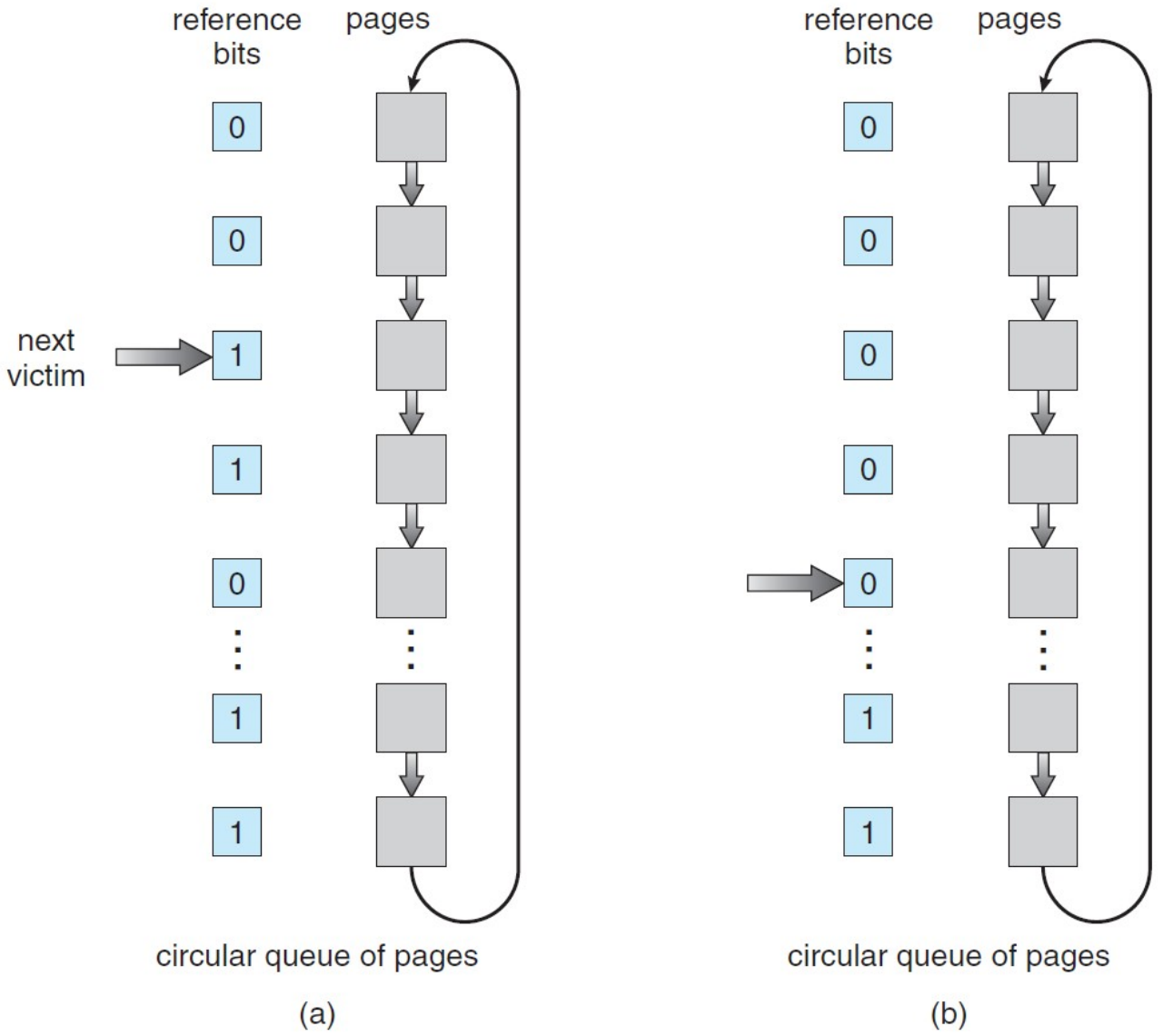
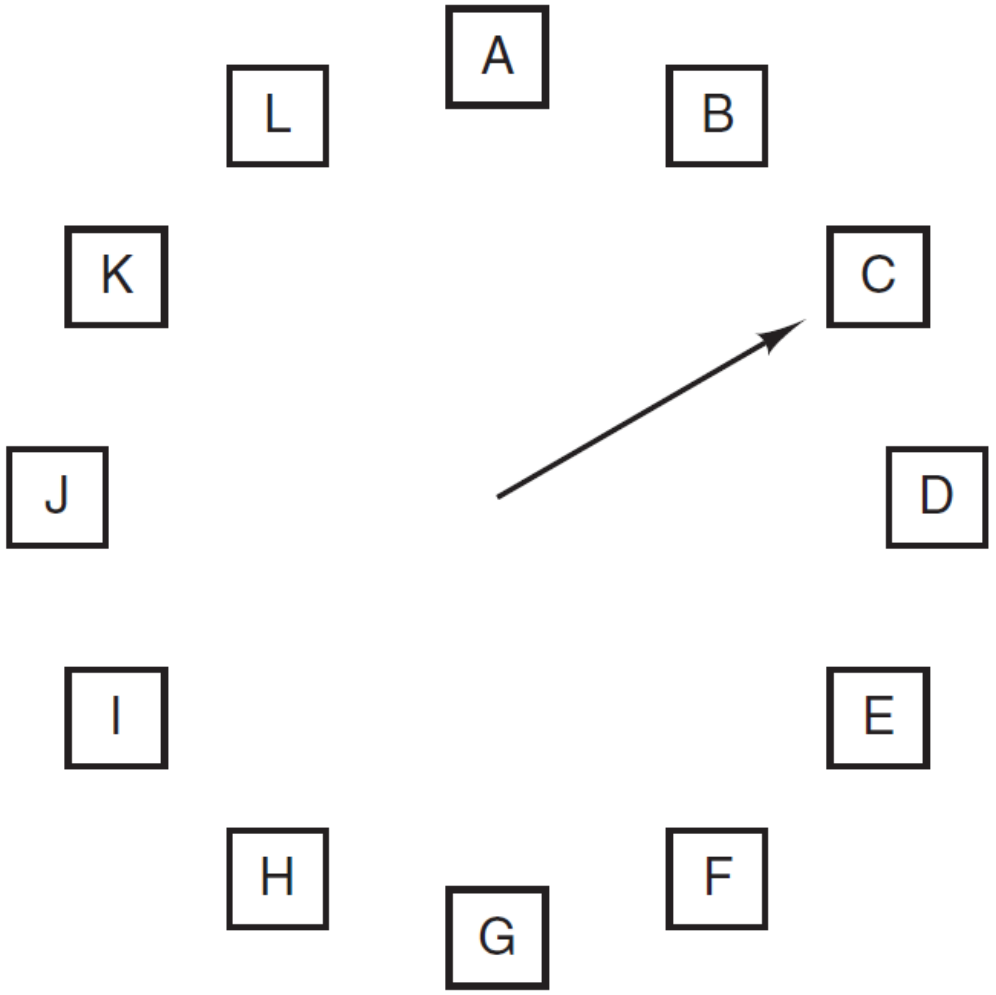


Figure 9.17 Second-chance (clock) page-replacement algorithm.

OSC9E

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

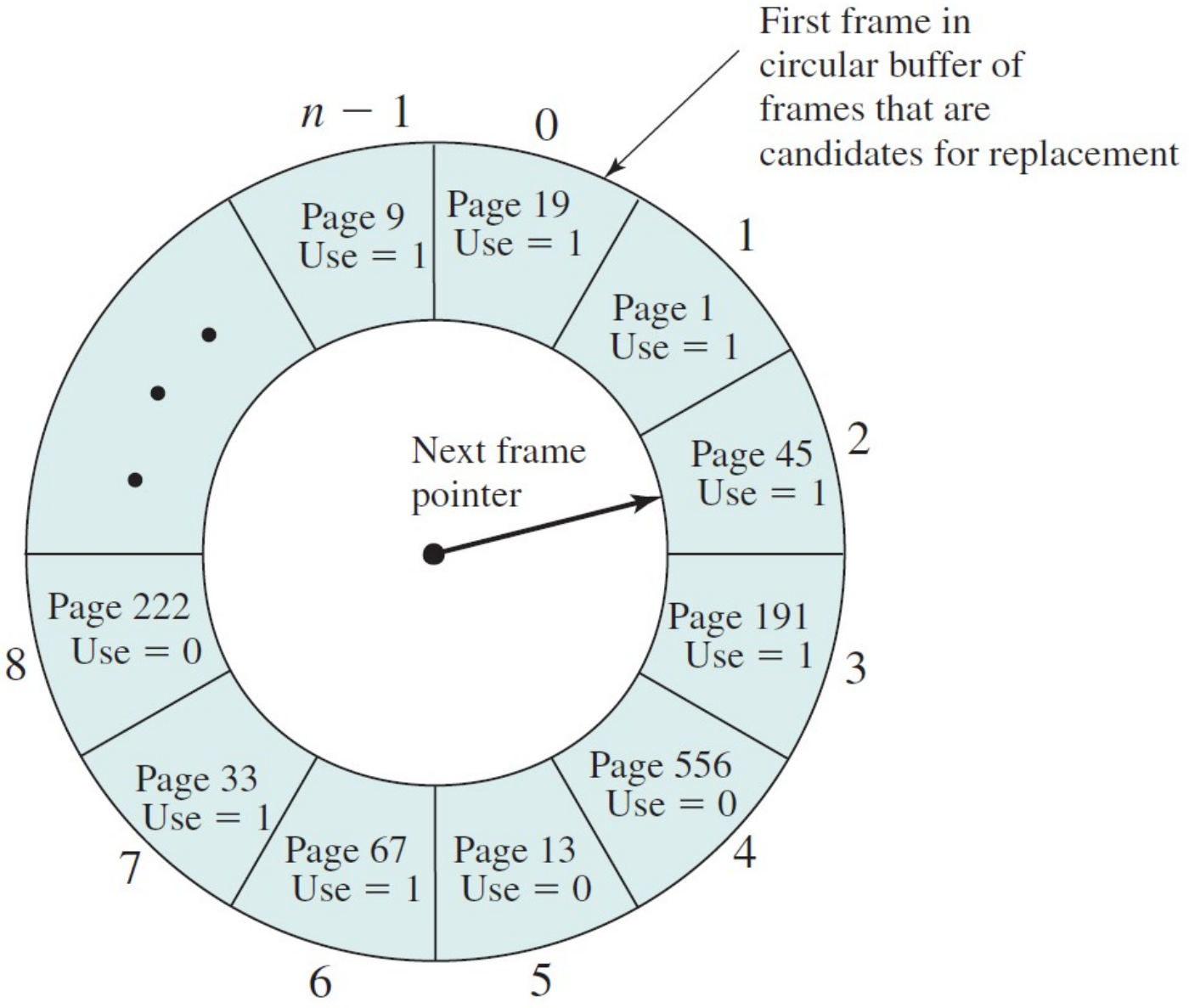
R = 0: Evict the page

R = 1: Clear R and advance hand

Figure 3-16. The clock page replacement algorithm.

MOS4E

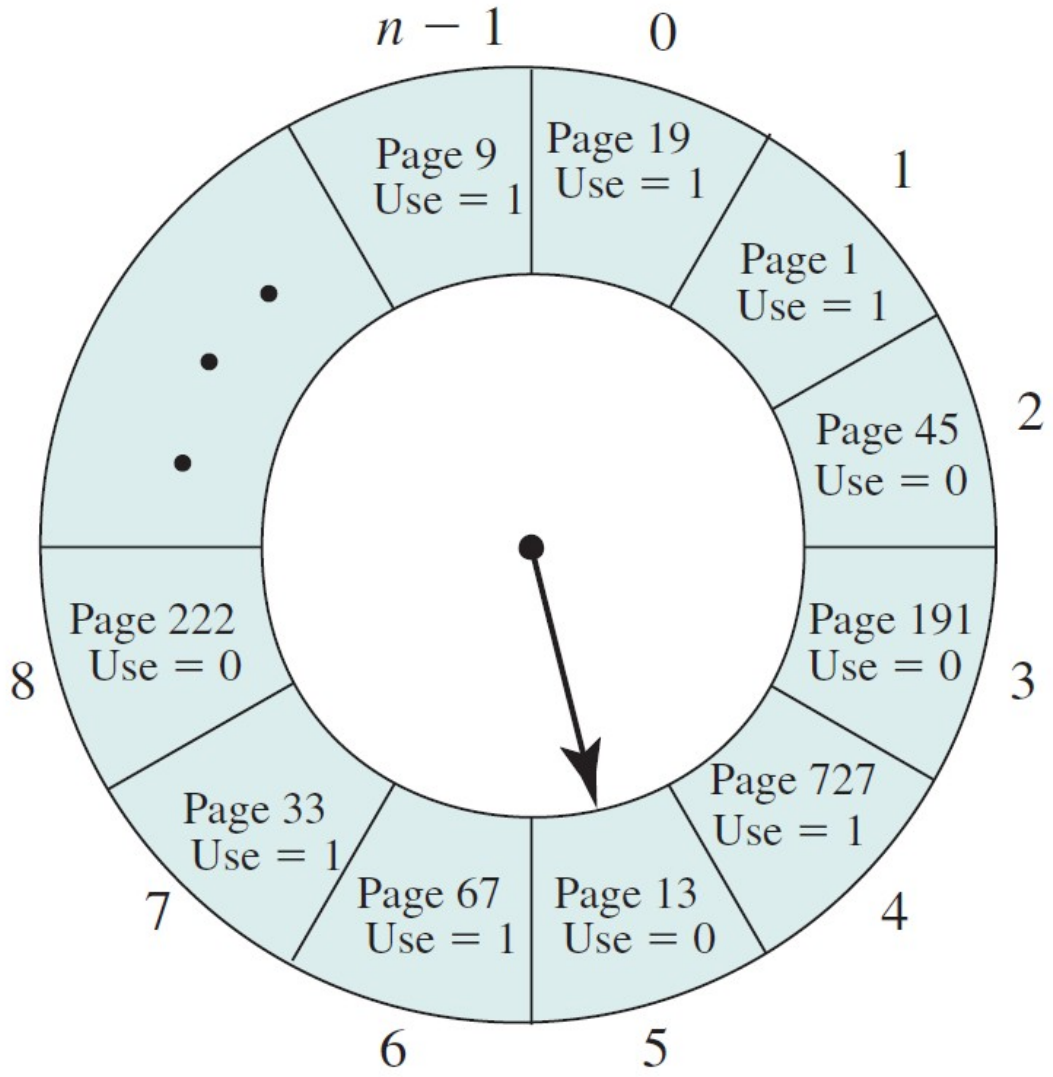
The Clock Page Replacement Algorithm



(a) State of buffer just prior to a page replacement

OSIDP9E

The Clock Page Replacement Algorithm

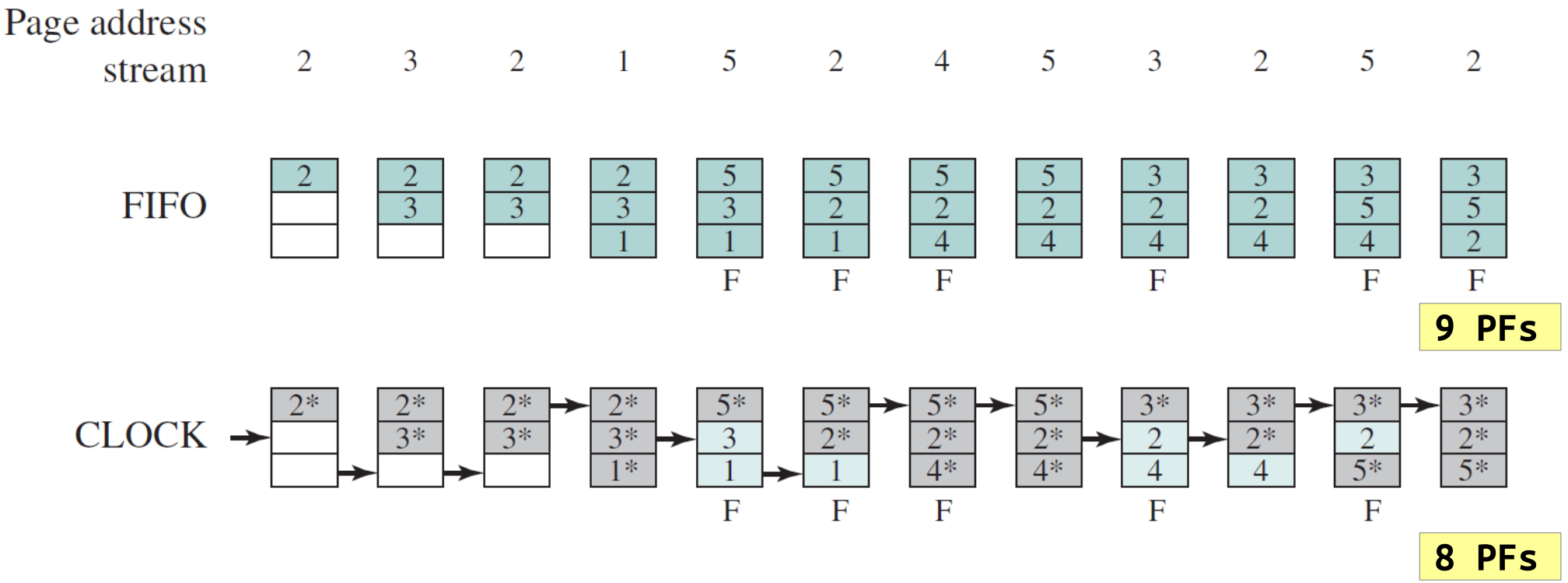


(b) State of buffer just after the next page replacement

Figure 8.15 Example of Clock Policy Operation

OSIDP9E

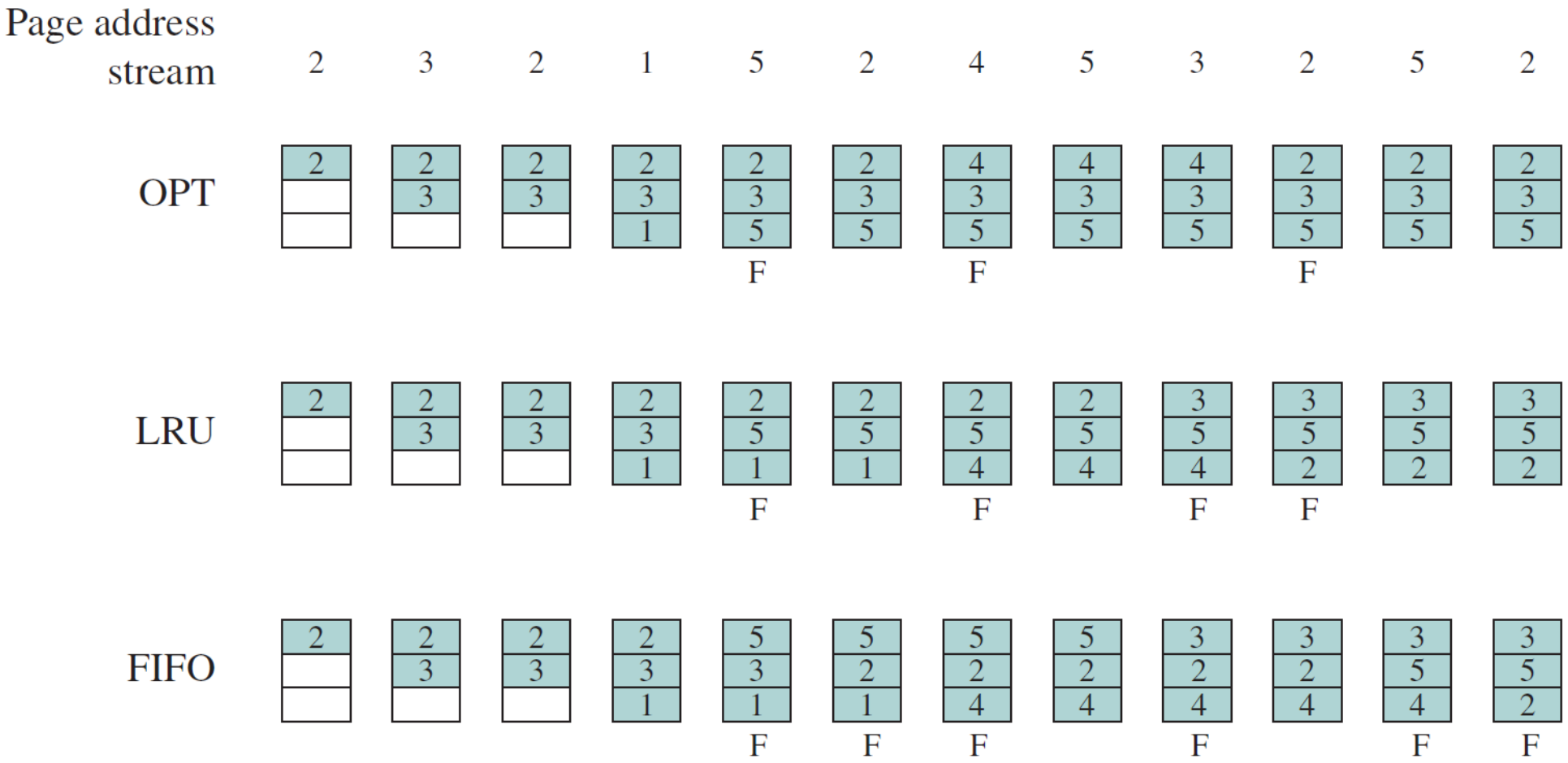
The Clock Page Replacement Algorithm



OSIDP9E

The Least Recently Used Page Replacement Algorithm

Replace the page that *has not been used* for the longest period of time.



OSIDP9E

The Least Recently Used Page Replacement Algorithm

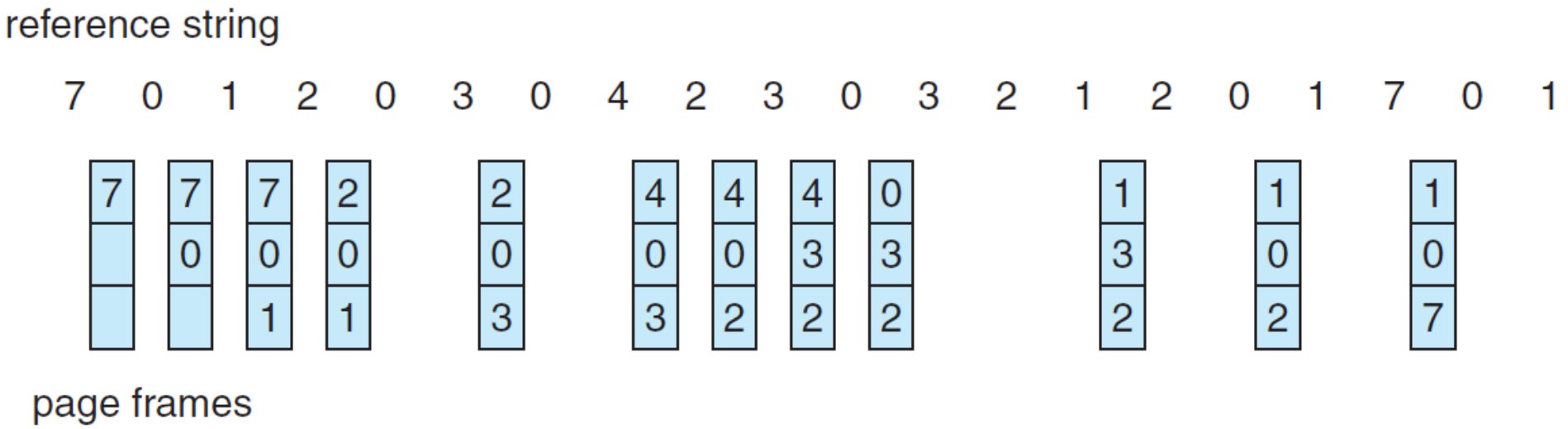


Figure 9.15 LRU page-replacement algorithm.

LRU: 12 PFs
vs
FIFO: 15 PFs

OSC9E

The LRU policy is often used as a page-replacement algorithm and is considered to be good.

The major problem is *how* to implement LRU replacement.

An LRU page-replacement algorithm may require substantial hardware assistance.

The problem is to determine an order for the frames defined by the time of last use.

OSC10E

Two implementations are feasible:

- **Counters. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value.**

  This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

OSC10E

- **Stack. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom (Figure 10.16). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.**

OSC10E

The LRU Implementation

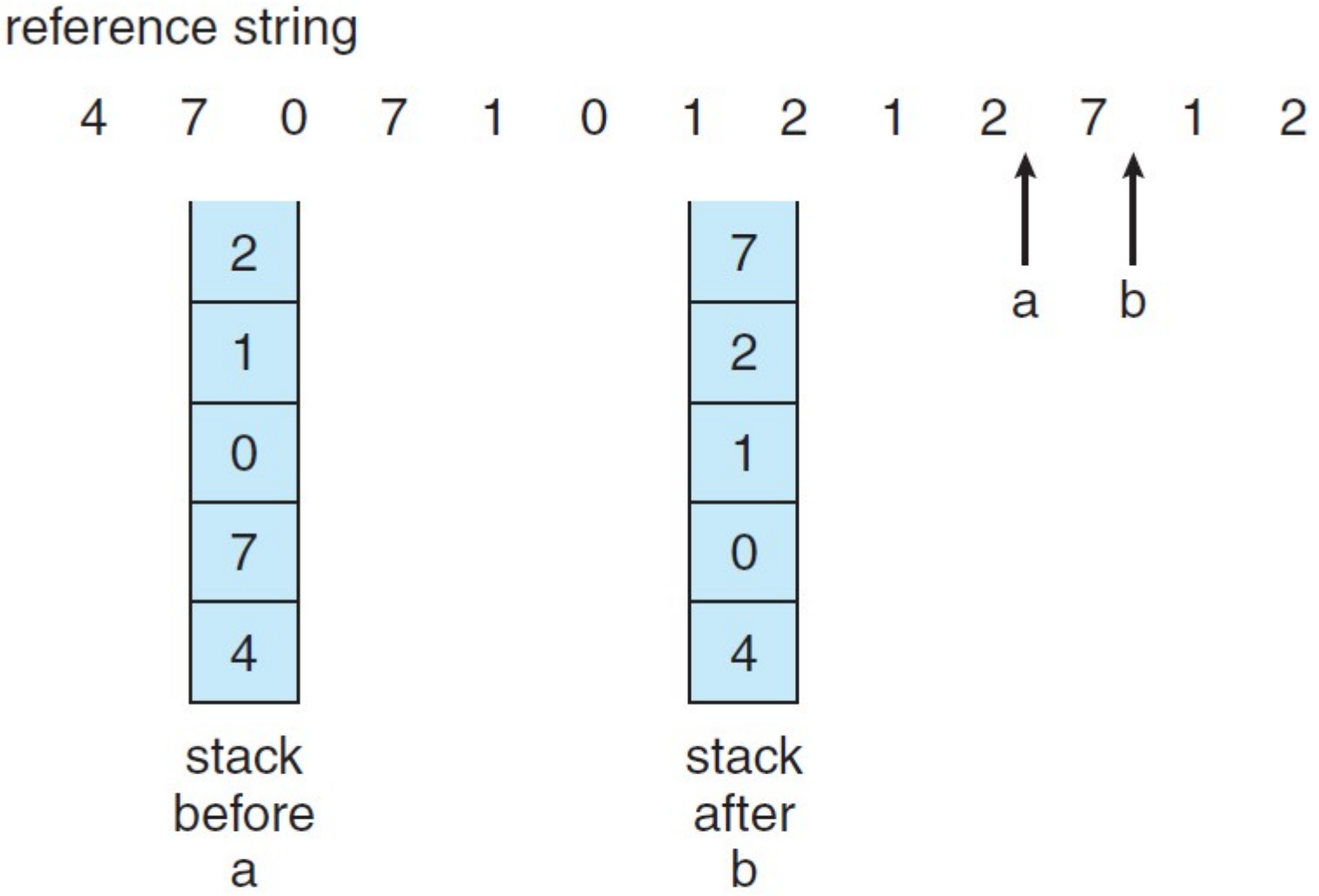


Figure 9.16 Use of a stack to record the most recent page references.

OSC9E

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's anomaly.

A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n* frames is always a subset of the set of pages that would be in memory with *n* + 1 frames.

For LRU replacement, the set of pages in memory would be the *n* most recently referenced pages. If the number of frames is increased, these *n* pages will still be the most recently referenced and so will still be in memory.

OSC10E

Simulating LRU in Software (NFU)

One possibility is called the NFU (Not Frequently Used) algorithm.

It requires a software counter associated with each page, initially zero.

At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter. The counters roughly keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

MOS4E

Simulating LRU in Software (Aging)

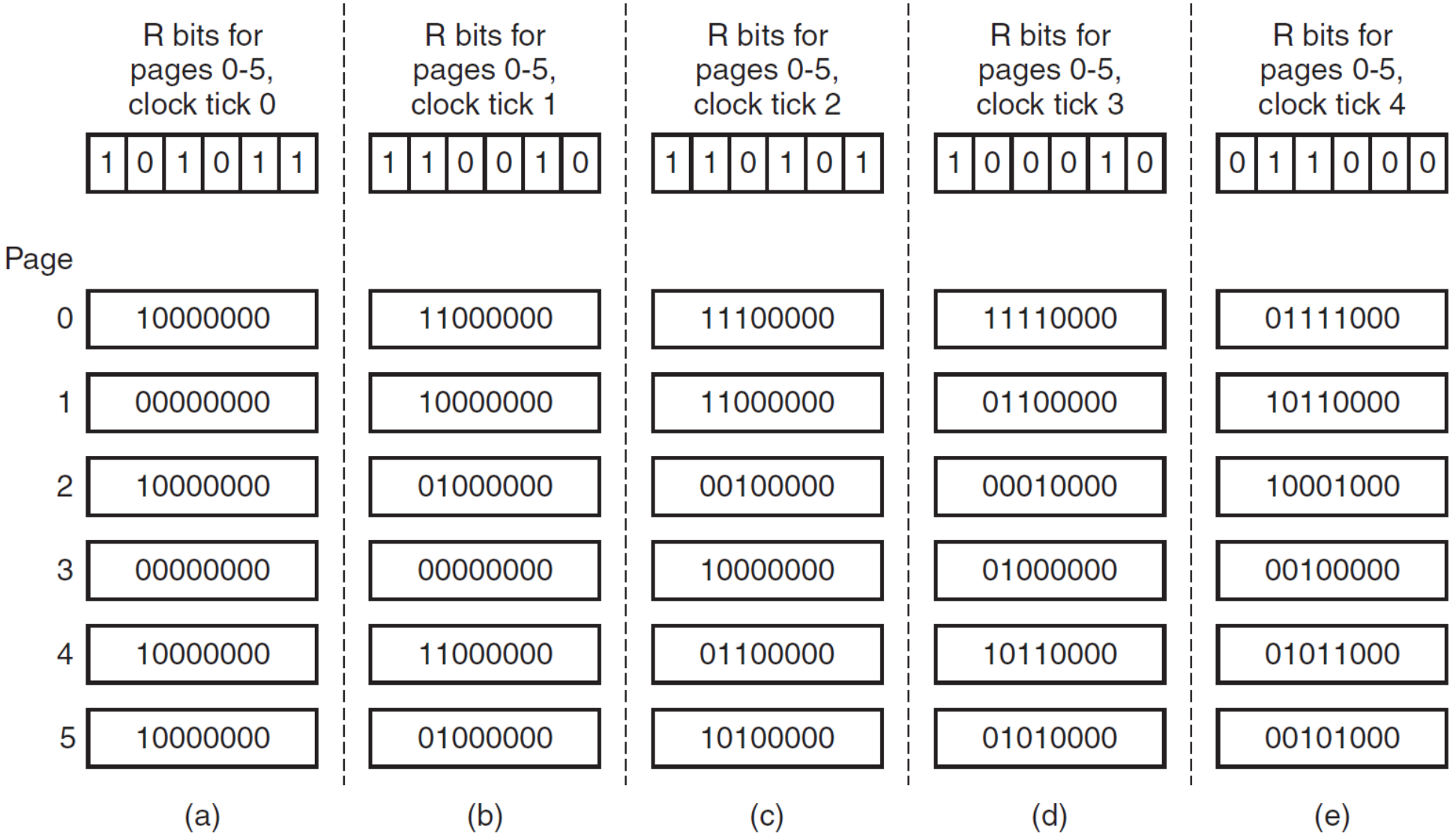


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

MOS4E

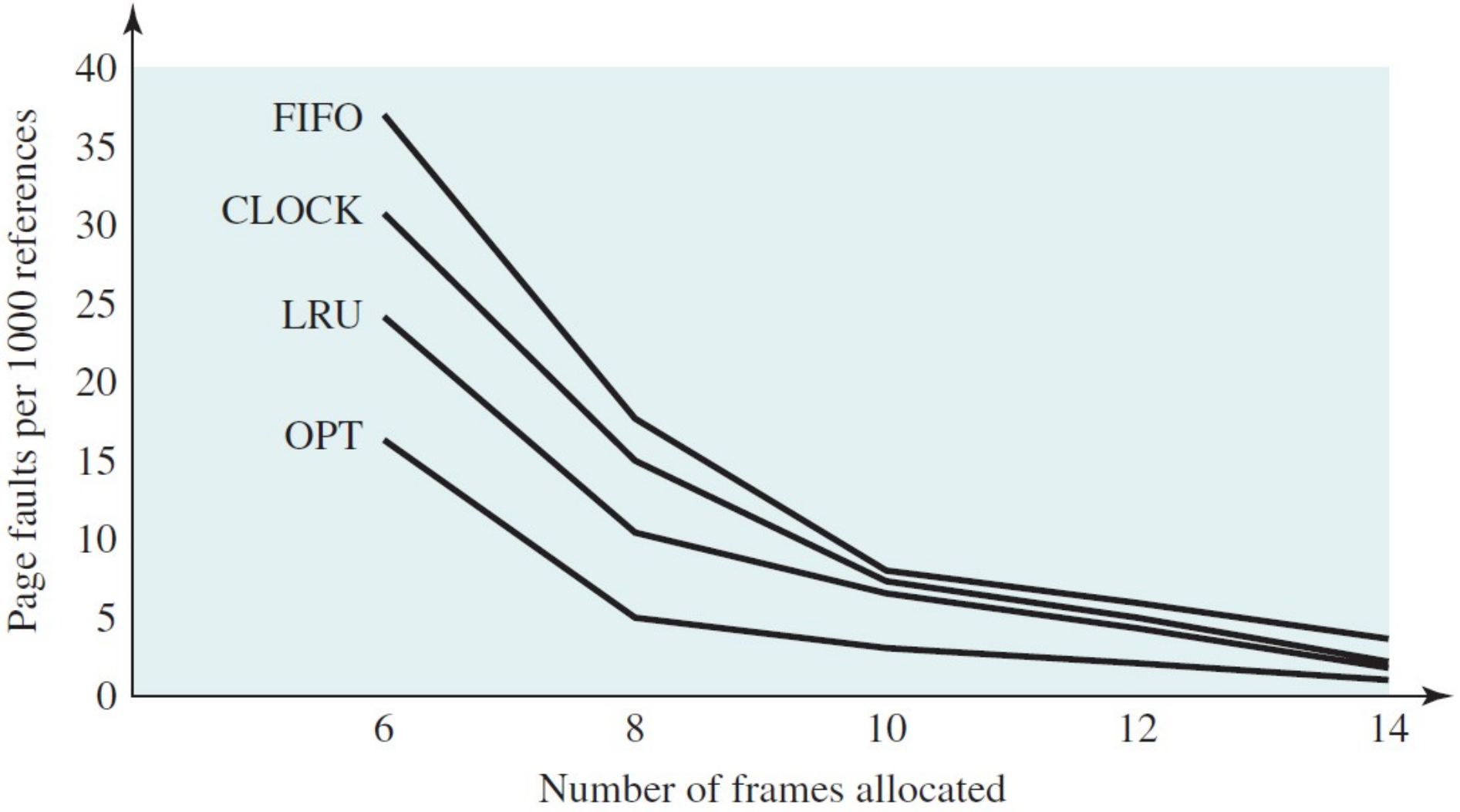


Figure 8.16 **Comparison of Fixed-Allocation, Local Page Replacement Algorithms**

OSIDP9E

The locality model of process execution states that, as a process executes, it moves from locality to locality.

A locality is a set of pages that are actively used together.

A running program is generally composed of several different localities, which may overlap. For example, when a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

OSC10E

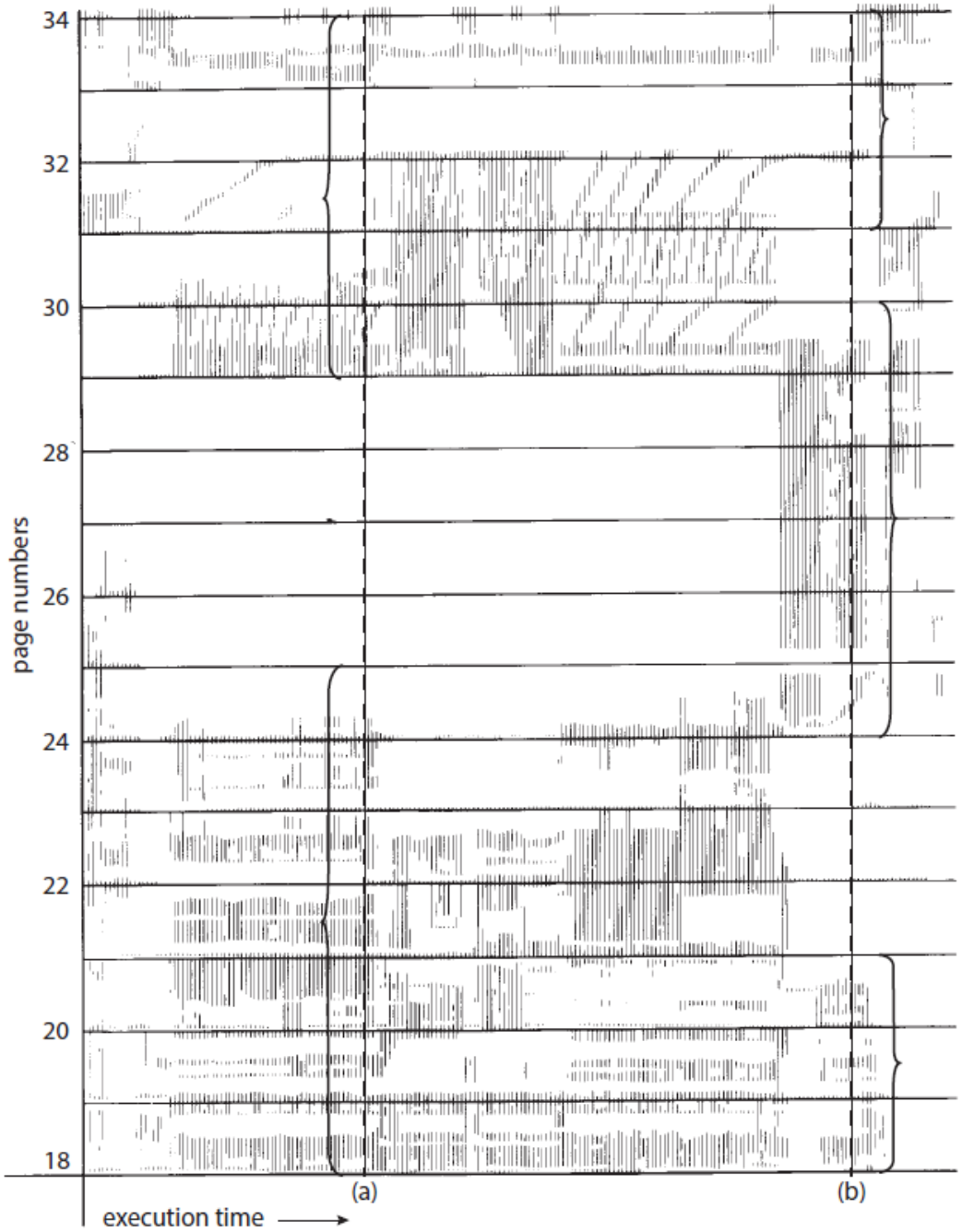


Figure 10.21 Locality in a memory-reference pattern.

Figure 10.21 illustrates the concept of locality and how a process's locality changes over time.

At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}.

At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}.

Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

OSC10E

The locality model of process execution states that, as a process executes, it moves from locality to locality.

A locality is a set of pages that are actively used together.

A running program is generally composed of several different localities, which may overlap. For example, when a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

OSC10E

The Working Set Model

At any instant of time, *t*, there exists a set consisting of all the pages used by the *k* most recent memory references. This set, *w*(*k*, *t*), is the working set.

Because the *k* = 1 most recent references must have used all the pages used by the *k* > 1 most recent references, and possibly others, *w*(*k*, *t*) is a monotonically nondecreasing function of *k*.

The limit of *w*(*k*, *t*) as *k* becomes large is finite because a program cannot reference more pages than its address space contains, and few programs will use every single page.

MOS4E

The Working Set Model

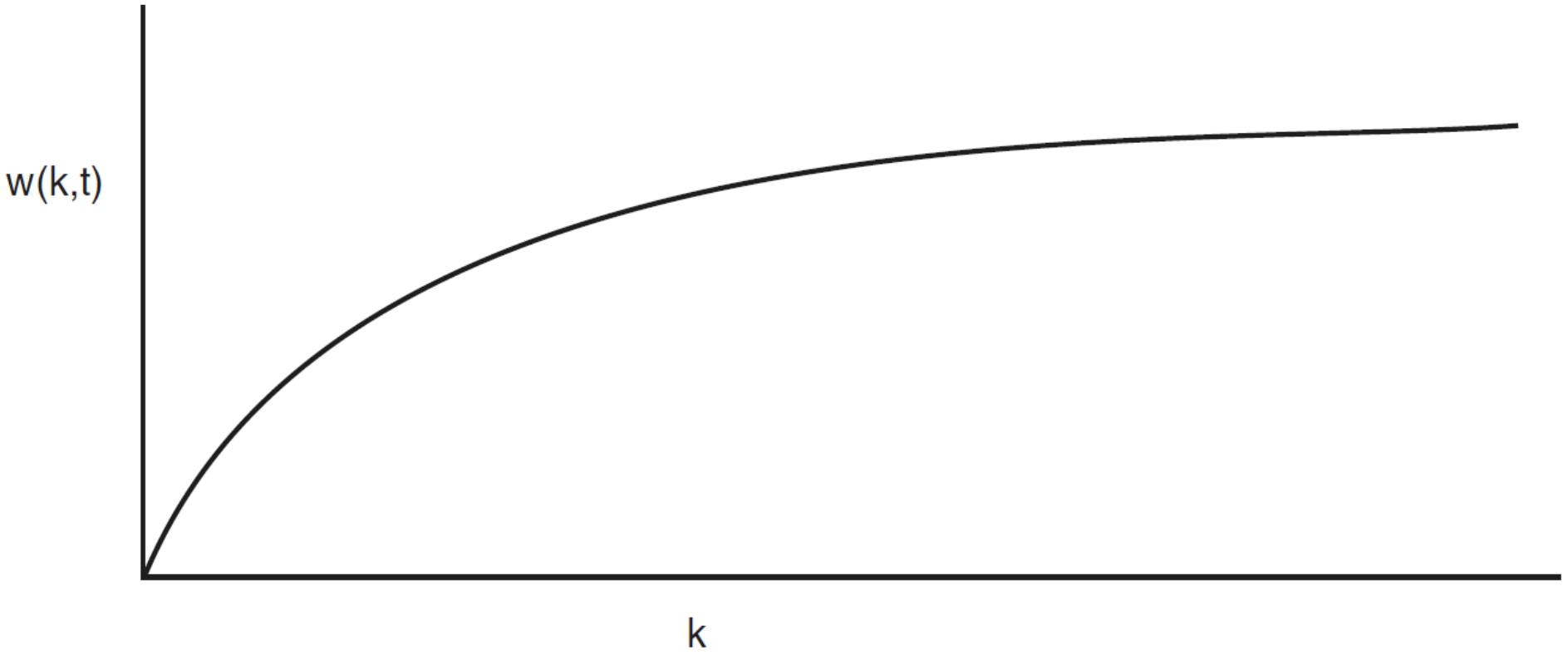


Figure 3-18. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

MOS4E

The Working Set Model

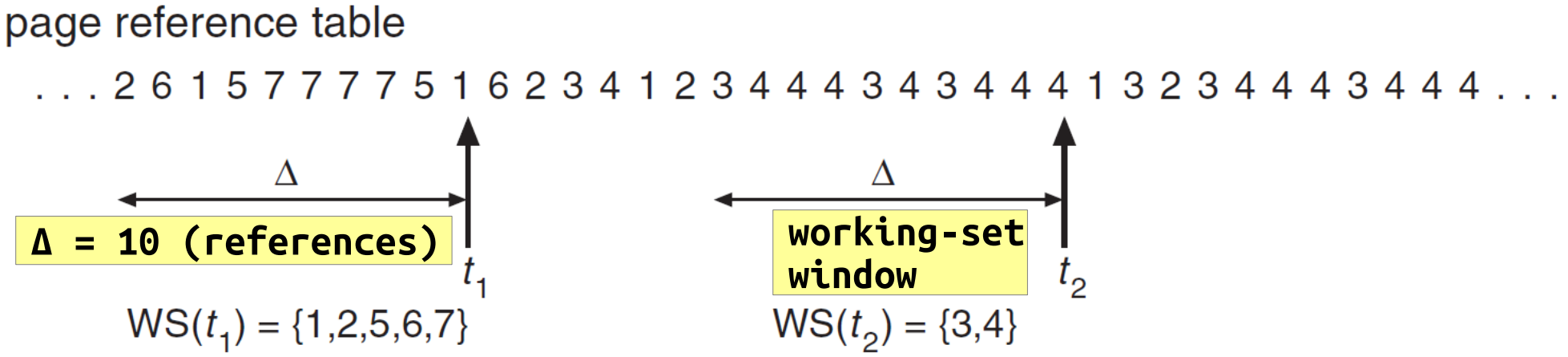


Figure 9.20 Working-set model.

Sequence of Page References	Window Size, Δ			
W	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figure 8.17 **Working Set of Process as Defined by Window Size**

OSIDP9E

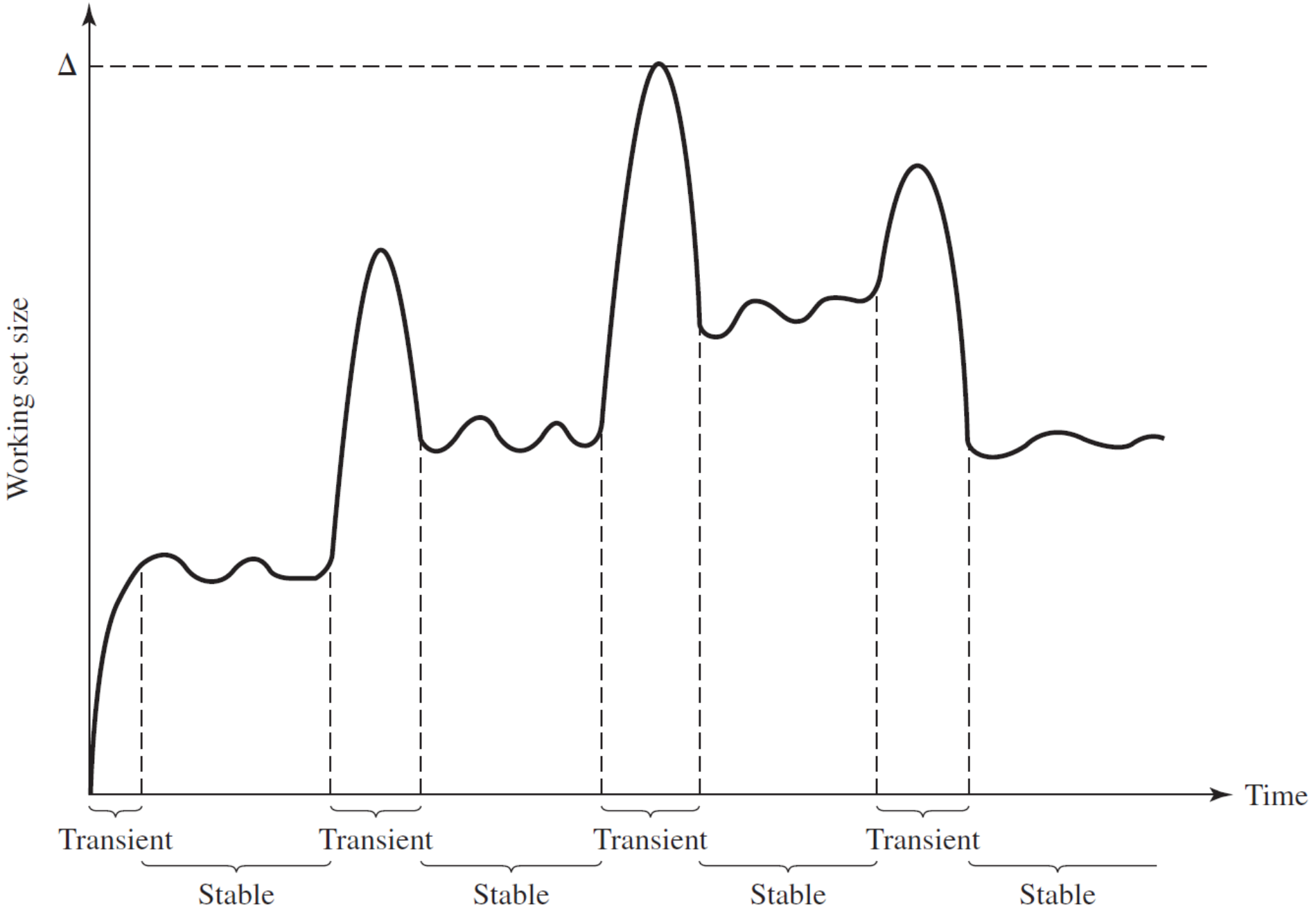
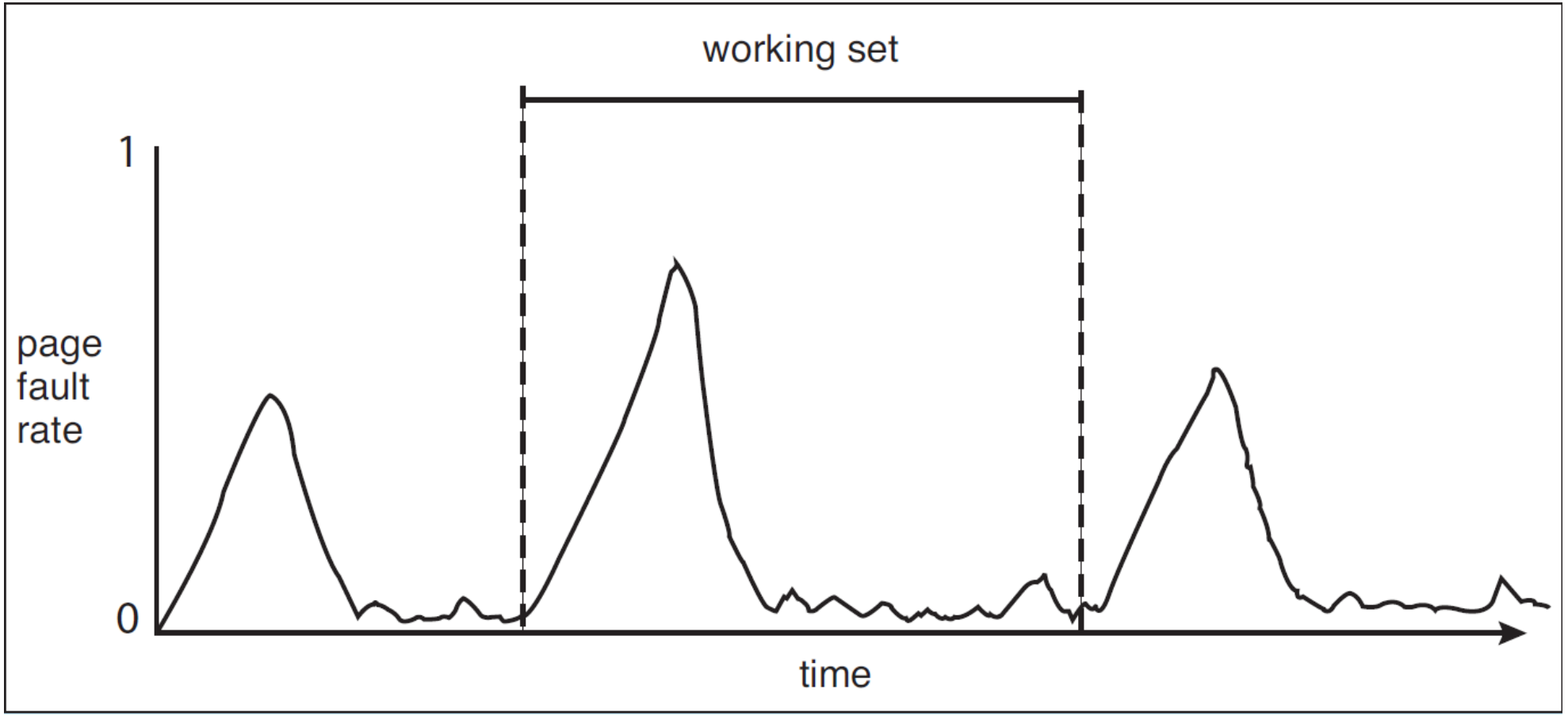


Figure 8.18 **Typical Graph of Working Set Size [MAEK87]**

OSIDP9E

The Working Set Model



OSC10E

The Working Set Algorithm

Instead of defining the working set as those pages used during the previous 10 million memory references, we can define it as the set of pages used during the past 100 msec of execution time. In practice, such a definition is just as good and much easier to work with. Note that for each process, only its own execution time counts. Thus if a process starts running at time *T* and has had 40 msec of CPU time at real time *T* + 100 msec, for working set purposes its time is 40 msec.

The amount of CPU time a process has actually used since it started is often called its current virtual time. With this approximation, the working set of a process is the set of pages it has referenced during the past τ seconds of virtual time.

MOS4E

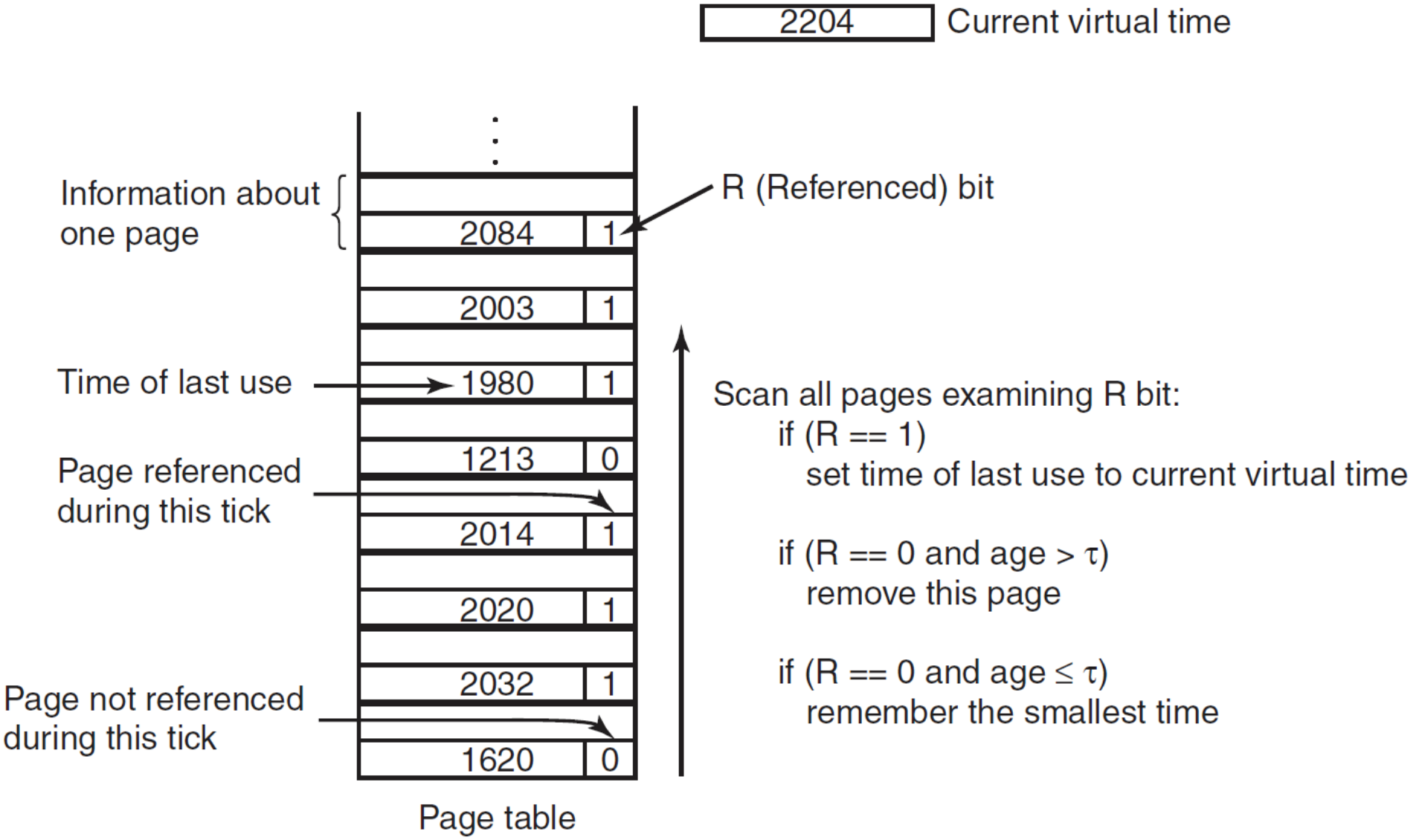


Figure 3-19. The working set algorithm.

MOS4E

The Working Set Algorithm

The hardware is assumed to set the *R* and *M* bits. Similarly, a periodic clock interrupt is assumed to cause software to run that clears the *Referenced* bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to evict.

As each entry is processed, the *R* bit is examined. If it is 1, the current virtual time is written into the *Time of last use* field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal (τ is assumed to span multiple clock ticks).

MOS4E

The Working Set Algorithm

If *R* is 0, the page has not been referenced during the current clock tick and may be a candidate for removal.

To see whether or not it should be removed, its age (the current virtual time minus its *Time of last use*) is computed and compared to τ.

If the age is greater than τ, the page is no longer in the working set and the new page replaces it. The scan continues updating the remaining entries.

MOS4E

The Working Set Algorithm

However, if *R* is 0 but the age is less than or equal to τ, the page is still in the working set.

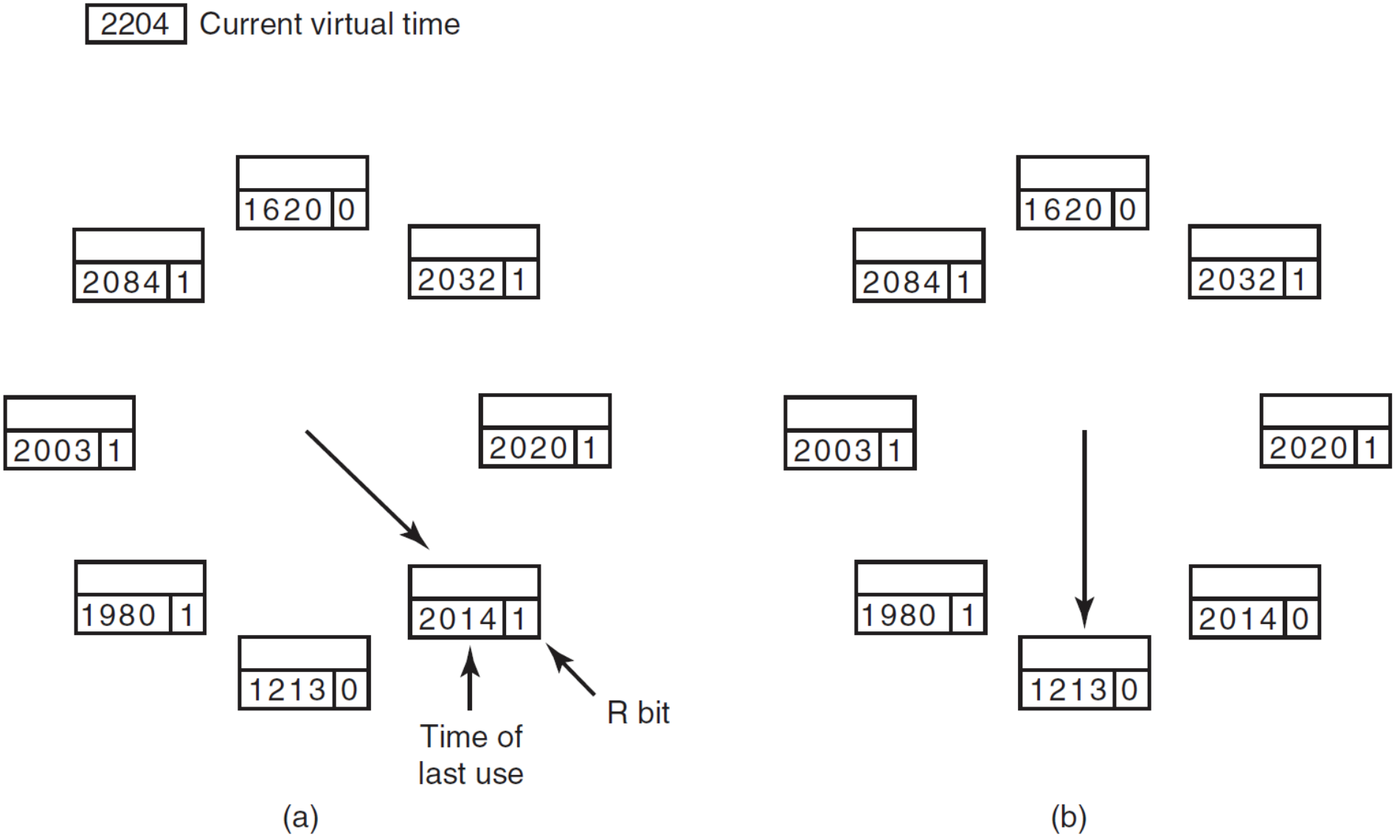
The page is temporarily spared, but the page with the greatest age (smallest value of *Time of last use*) is noted.

If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with *R* = 0 were found, the one with the greatest age is evicted.

In the worst case, all pages have been referenced during the current clock tick (and thus all have *R* = 1), so one is chosen at random for removal, preferably a clean page, if one exists.

MOS4E

The WSClock Algorithm



MOS4E

The WSClock Algorithm

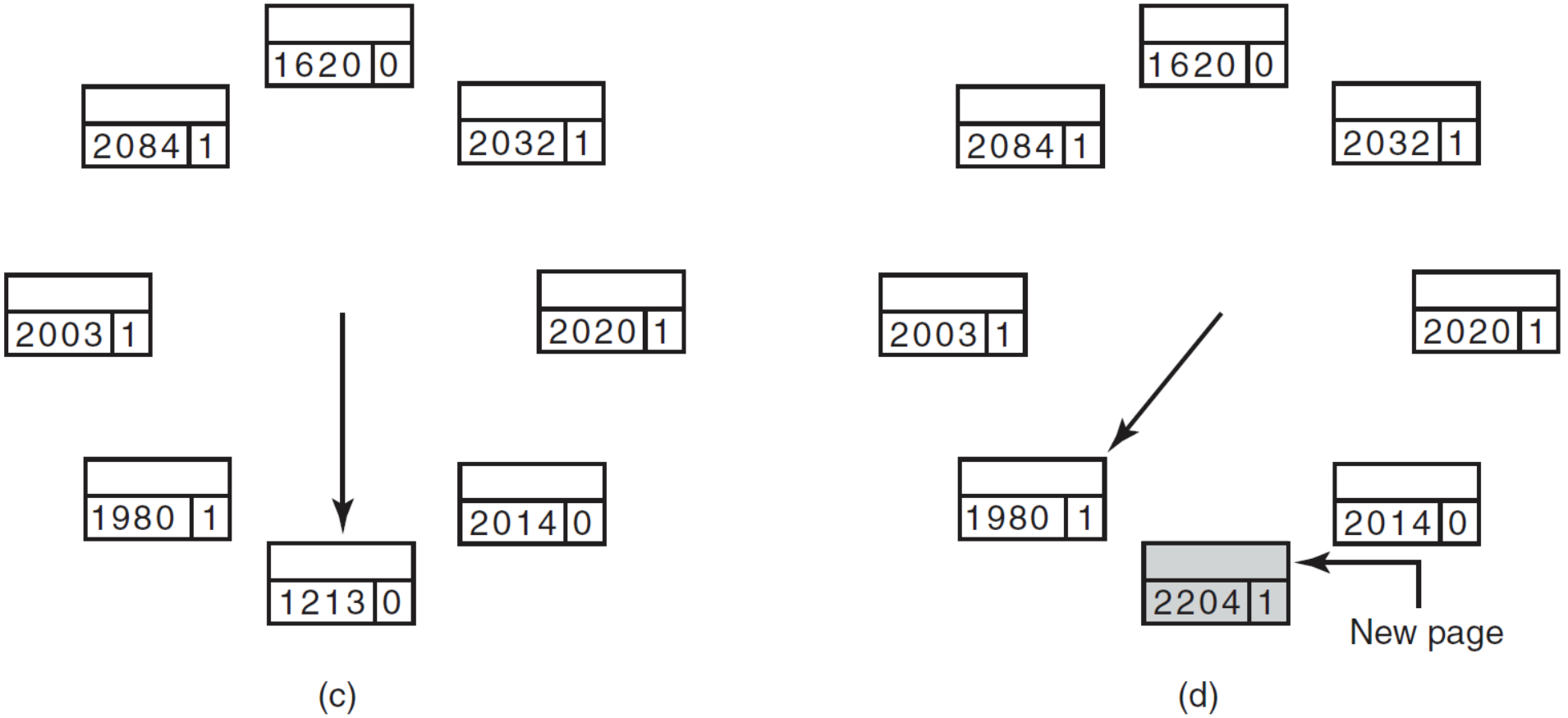


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

MOS4E

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 3-21. Page replacement algorithms discussed in the text.

MOS4E