Module 5
Memory Management

Dr.J.C.KAVITHA,
VIT, CHENNAI

Virtual Memory

In memory-management algorithms, the instructions being executed must be in physical memory and it limits the size of a program to the size of physical memory.

In fact, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions but it ever, occur in practice, and this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely.

Virtual Memory

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- Because each program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and its users.

Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory.

The advantage is that programs can be larger than physical memory.

Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory.

This technique frees programmers from the concerns of memory-storage limitations.

Virtual Memory

Virtual memory involves the separation of logical memory from physical memory.

This separation allows large virtual memory to be provided for programmers when only a smaller physical memory is available. (Figure 10.1).

Virtual memory makes the task of programming much easier, because the prog worry about the amount of physica

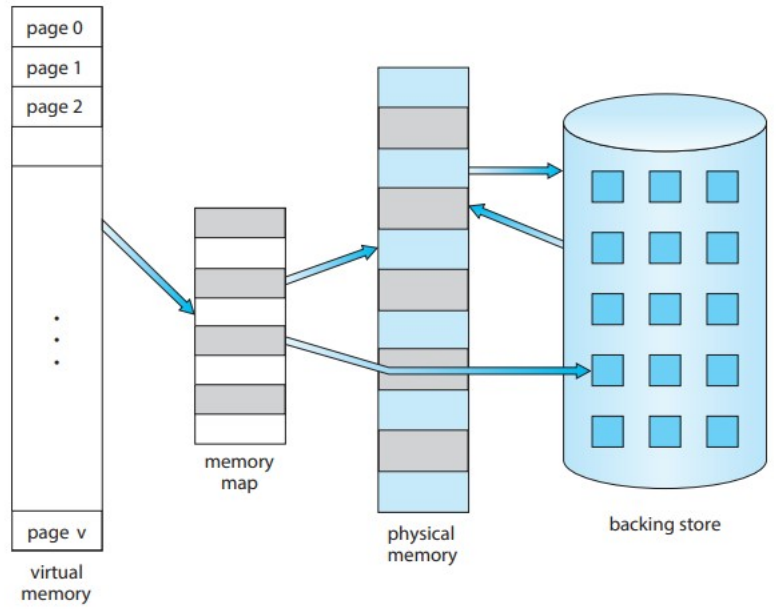


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

Virtual Memory

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.

A process begins at a certain logical address—say, address 0 and exists in contiguous memory, as shown in Figure 10.2.

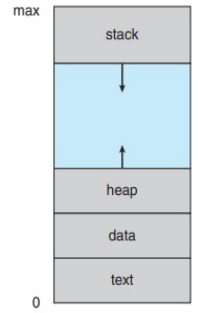
The heap is allowed to grow upward in memory as it is used for dynamic memory allocation.

Similarly, the stack is allowed to grow downward in memory through successive function calls.

The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

Virtual address spaces that include holes are known as sparse address spaces.

Using a sparse address space is
stack or heap segments grow
possibly other shared objects) du



he holes can be filled as the
ynamically link libraries (or
tion.

Figure 10.2 Virtual address space of a process in memory.

Virtual Memory

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

• System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure 10.3). Typically, a library is mapped read-only into the space of each process that is linked with it.

• Similarly, processes can share memory. Two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, muc e 10.3.

• Pages can be shared with the fork() system call, thus speeding up process crea

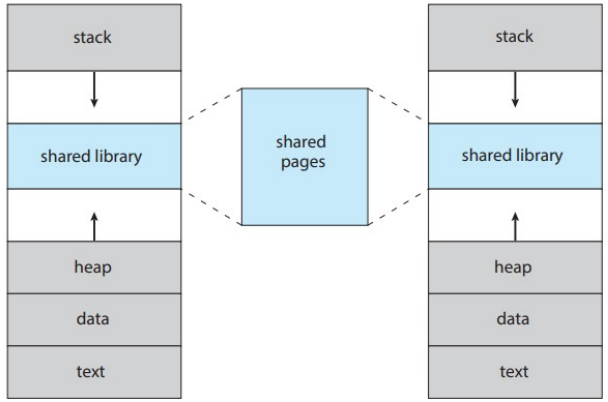


Figure 10.3 Shared library using virtual memory.

Demand Paging

Consider how an executable program might be loaded from secondary storage into memory. One option is to load the entire program in physical memory at program execution time.

However, a problem with this approach is that we may not initially need the entire program in memory.

Suppose a program starts with a list of available options from which the user is to select.

Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user.

An alternative strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems.

With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.

Pages that are never accessed are thus never loaded into physical

Demand Paging

Demand paging is the process of loading the pages only when they are demanded by the process during execution.

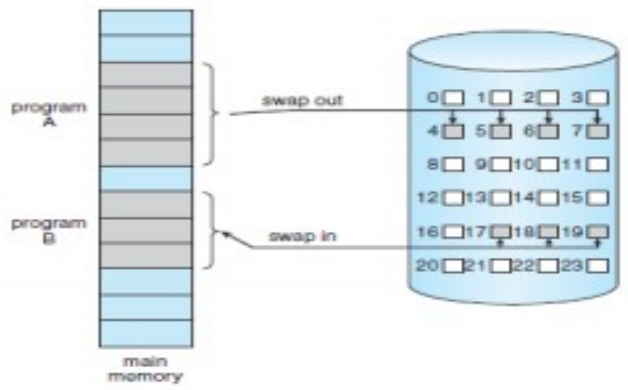
Pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory.

When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory we use a lazy swapper that never swaps a page into memory unless that page will be needed.

Lazy swapper is termed to as pager in demand paging.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pa



Demand Paging

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.

Thus, we need some form of hardware support to distinguish between the two.

The valid – invalid bit scheme can be used for this purpose.

This time, however, when the bit is set to “valid,” the associated page is both legal and in memory.

If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage.

The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid.

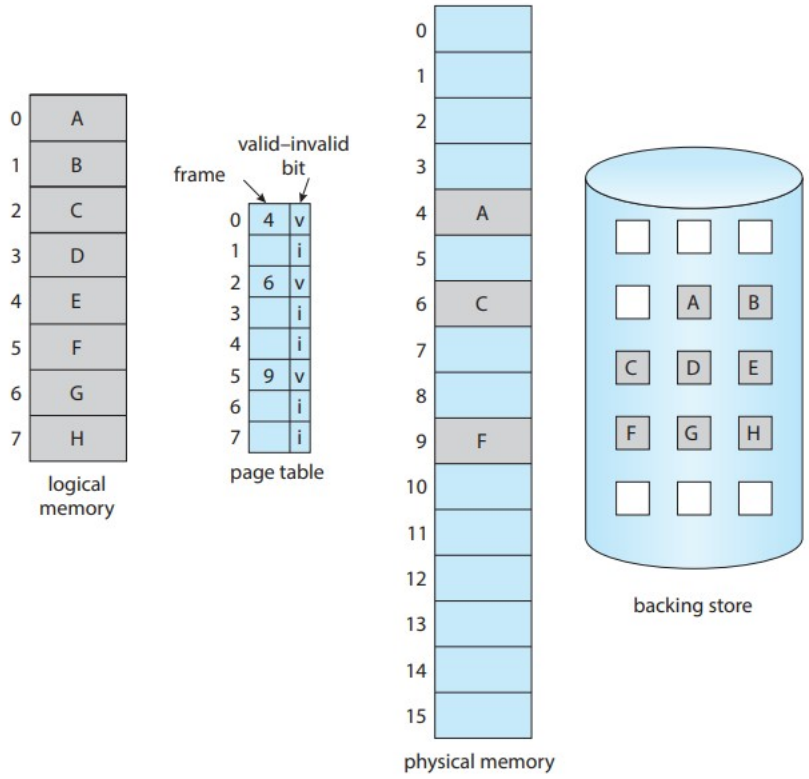
Demand Paging

This situation is depicted in Figure 10.4. (Notice that marking a page invalid will have no effect if the process never attempts to access that page.)

But what happens if the process tries to access a page that was not brought into memory?

Access to a page marked invalid causes a page fault.

The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's f memory.



Demand Paging

This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 10.5):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example)
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The

Demand Paging

The process of executing a program with no pages in main memory is called as **pure demand paging**. This never brings a page into memory until it is required. The hardware to support demand paging is the same as the hardware for paging and swapping.

Page table: This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.

Secondary memory: This memory holds those pages that are not present in main memory. The secondary memory is [illegible] as the swap device, and the section of disk [illegible] swap space.

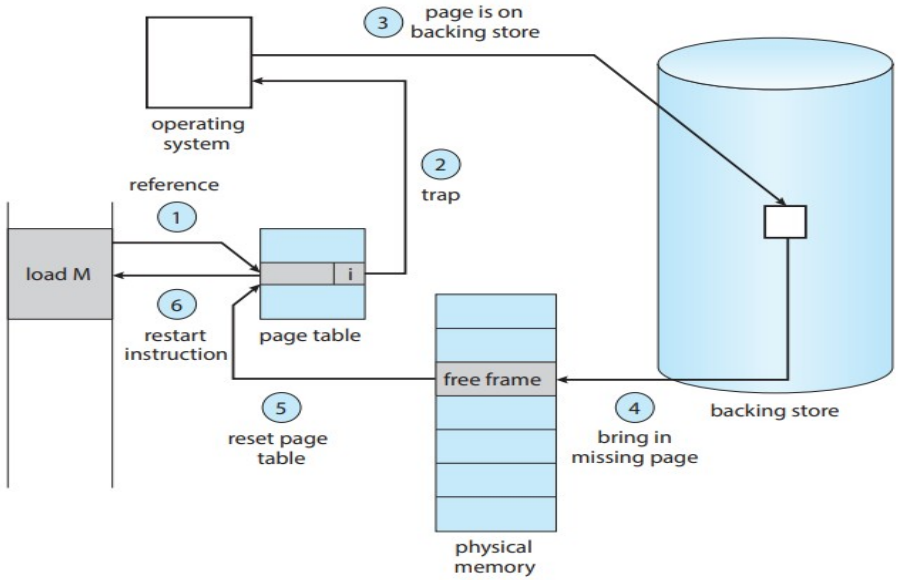


Figure 10.5 Steps in handling a page fault.

Performance of Demand Paging

Demand paging can affect the performance of a computer system.
The effective access time for a demand-paged memory is given by

$$\text{Effective access time} = (1 - p) \times ma + p \times \text{page fault time}$$

The memory-access time, denoted ma, ranges from 10 to 200 nanoseconds.
If there is no page fault then the effective access time is equal to the memory access time.
If a page fault occurs, first read the relevant page from disk and then access the desired word.
There are three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process

Ex: With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{Effective access time} &= (1 - p) \times (200) + p\ (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

Effective access time is directly proportional to the page-fault rate

NEED FOR PAGE REPLACEMENT

- If a page requested by a process is in memory, then the process can access it. If the requested page is not in main memory, then it is page fault.
- When there is a page fault the OS decides to load the pages from the secondary memory to the main memory.
- It looks for the free frame. If there is no free frame then the pages that are not currently in use will be swapped out of the main memory, and the desired page will be swapped into the main memory.
- The process of swapping a page out of main memory to the swap space and swapping in the desired page into the main memory for execution is called as **Page Replacement.**

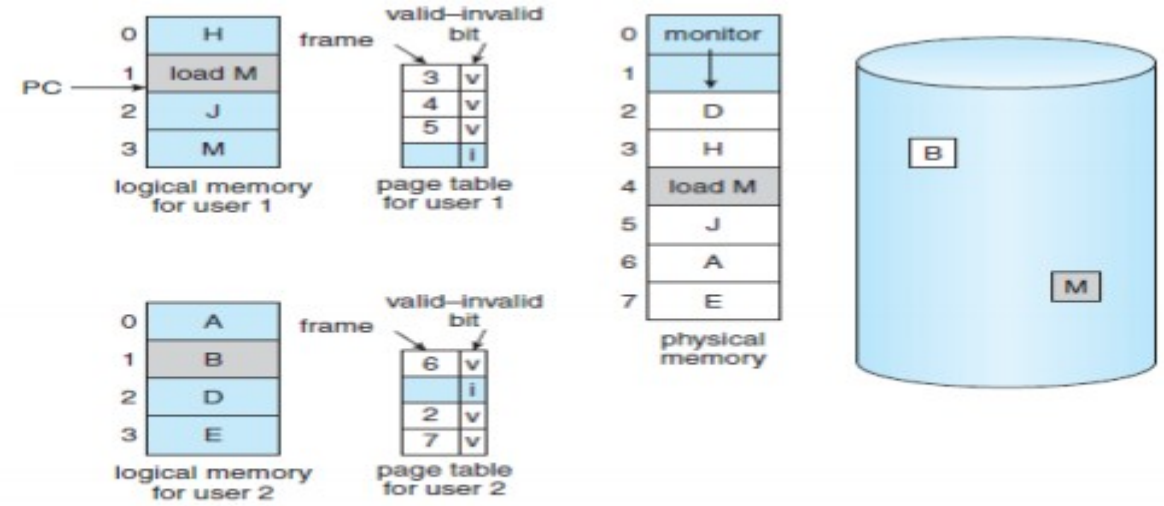


Figure 9.9 Need for page replacement.

STEPS IN PAGE REPLACEMENT

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user p ed.

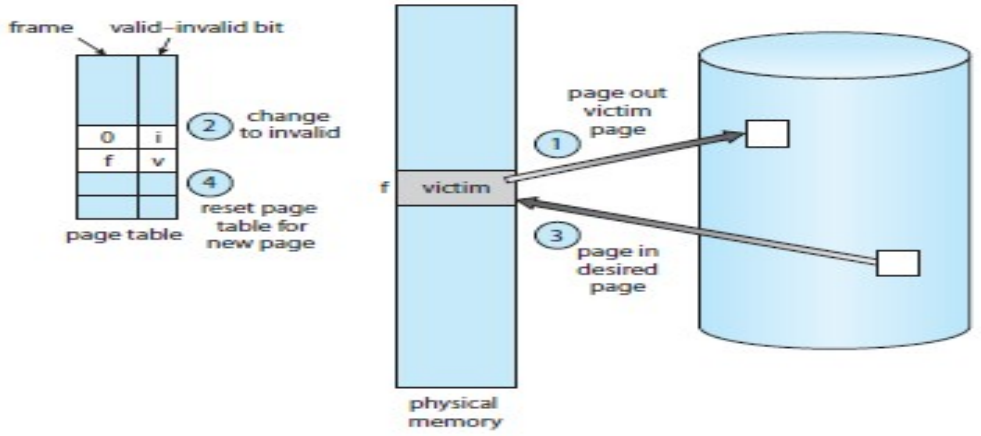


Figure 9.10 Page replacement.

PAGE REPLACEMENT

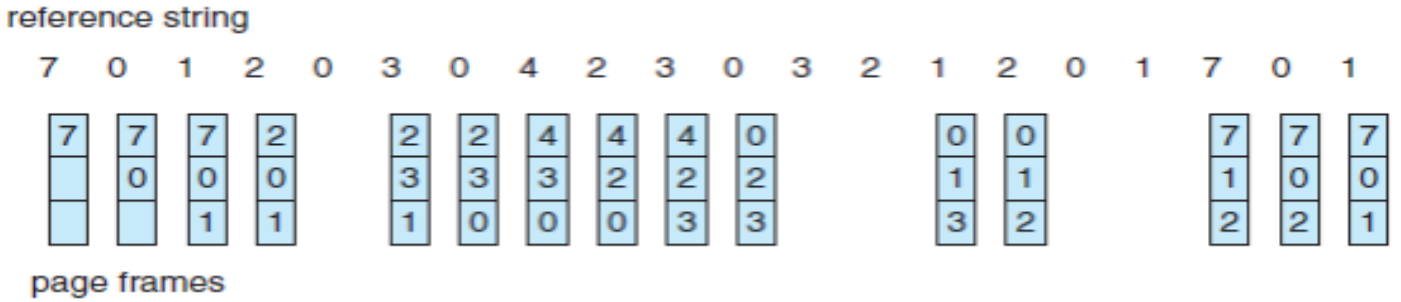
- If no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
- This overhead can be reduced by using a modify bit (or dirty bit).
- When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
- **MODIFY BIT:** The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.

PAGE REPLACEMENT ALGORITHMS

- If we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.
- The string of memory references made by a process is called a reference string.
- There are many different page-replacement algorithms that includes
  - FIFO page Replacement
  - Optimal Page Replacement
  - LRU Page Replacement
  - LRU Approximation page Replacement algorithm
  - Counting Based Page Replacement Algorithm
  - Page Buffering Algorithm

FIFO PAGE REPLACEMENT

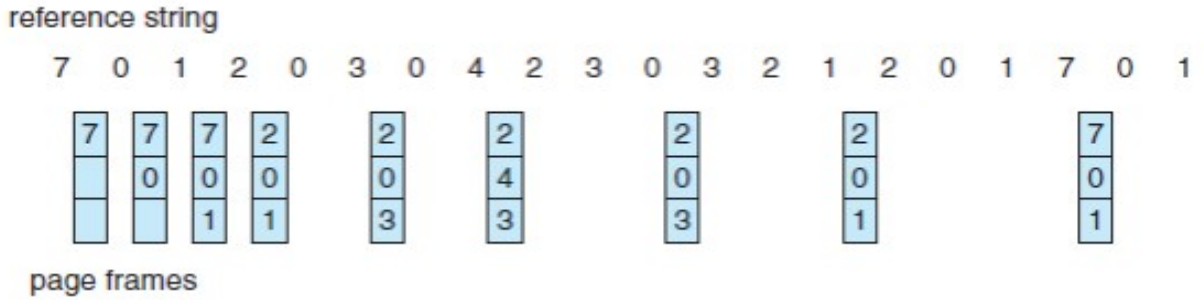
- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm replaces the oldest page that was brought into main memory.
- EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames



- The three frames are empty initially.
- The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
- The algorithm has 15 faults.
- **Disadvantages:** It Suffers from Belady's Anomaly.
- **BELADY'S ANOMALY:** The page fault increases as the number of allocated memory frame increases. This is called as Belady's Anomaly.

OPTIMAL PAGE REPLACEMENT

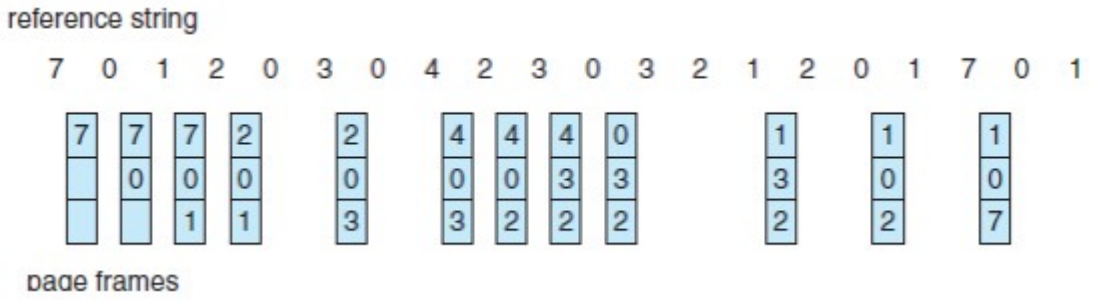
- This algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Optimal page replacement algorithm replace the page that will not be used for the longest period of time.
- EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



- The Optimal replacement algorithm produces nine faults.
- The first three references cause faults that fill the three empty frames.
- **Advantage**: Optimal replacement is much better than a FIFO algorithm **Disadvantage:** The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU PAGE REPLACEMENT

- The Least Recently used algorithm replaces a page that has not been used for a longest period of time.
- LRU replacement associates with each page the time of that page's last use.
- It is similar to that of Optimal page Replacement looking backward in time, rather than forward.
- EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



- The LRU algorithm produces twelve faults.
- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 7 has not been used for a longest period of time, when we look backward.
- The Reference to page 3 replaces page 1, because page 1 has not been used for a longest period of time.
- When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
- Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

LRU PAGE REPLACEMENT

- Advantages:
  - The LRU policy is often used as a page-replacement algorithm and is considered to be good.
  - LRU replacement does not suffer from Belady's anomaly.
- Disadvantage:
  - The problem is to determine an order for the frames defined by the time of last use.
  - Two implementations are feasible:

❖ Counters: We associate with each page-table 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. So we can find the time of the last reference to each page.

❖ **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.

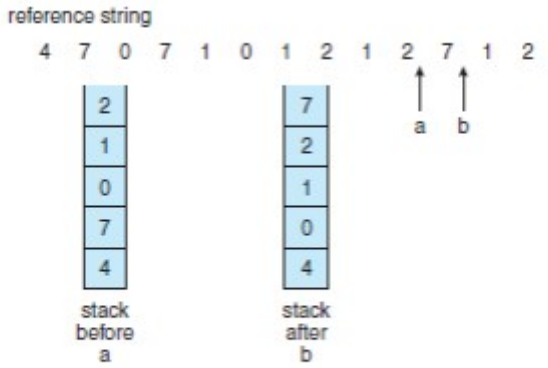


Figure 9.16 Use of a stack to record the most recent page references.

LRU APPROXIMATION PAGE REPLACEMENT ALGORITHM

- The system provides support to the LRU algorithm in the form of a bit called Reference bit.
- REFERENCE BIT: The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).
- Reference bits are associated with each entry in the page table.
- Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- After some time, we can determine which pages have been used and which have not been used by examining the reference bits.
- This information is the basis for many page-replacement algorithms that approximate LRU replacement.

LRU APPROXIMATION PAGE REPLACEMENT ALGORITHM

a) Additional-reference-bits algorithm:

- The additional ordering information can be gained by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods.
- A page that is used at least once in each period has a shift register value of 11111111.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
- Thus the page with the lowest number is the LRU page, and it can be replaced.

LRU APPROXIMATION PAGE REPLACEMENT ALGORITHM

b) Second-Chance Algorithm:

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm.
- When a page has been selected, we inspect its reference bit.
- If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced.
- One way to implement the second-chance algorithm is as a circular queue.
- A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

COUNTING-BASED PAGE REPLACEMENT

- We can keep a counter of the number of references that have been made to each page. This method includes two schemes:
- Least frequently used (LFU) page-replacement: The least frequently used (LFU) pagereplacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.
- Most frequently used (MFU) page-replacement algorithm: The most frequently used (MFU) pagereplacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

PAGE BUFFERING ALGORITHM

- Systems commonly keep a pool of free frames.
- When a page fault occurs, a victim frame is chosen and the desired page is read into a free frame from the pool before the victim is written out.
- This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.
- Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset.
- This scheme increases the probability that a page will be clean when it is selected for replacement.