OPERATING SYSTEMS

Topic - Virtual Memory

Virtual Memory

Virtual Memory allows execution of processes that may not be completely reside in the physical memory. The main advantage of this scheme is that program can be larger than the available physical memory.

In fact, an examination of real programs shows us that, in many cases, the entire program is not needed

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need.
- Certain options and features of a program may be used rarely.
- Even in those cases where the entire program is needed, it may not all be needed at the same time.

Benefits

The ability to execute a program that is only partially in memory would give 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 user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

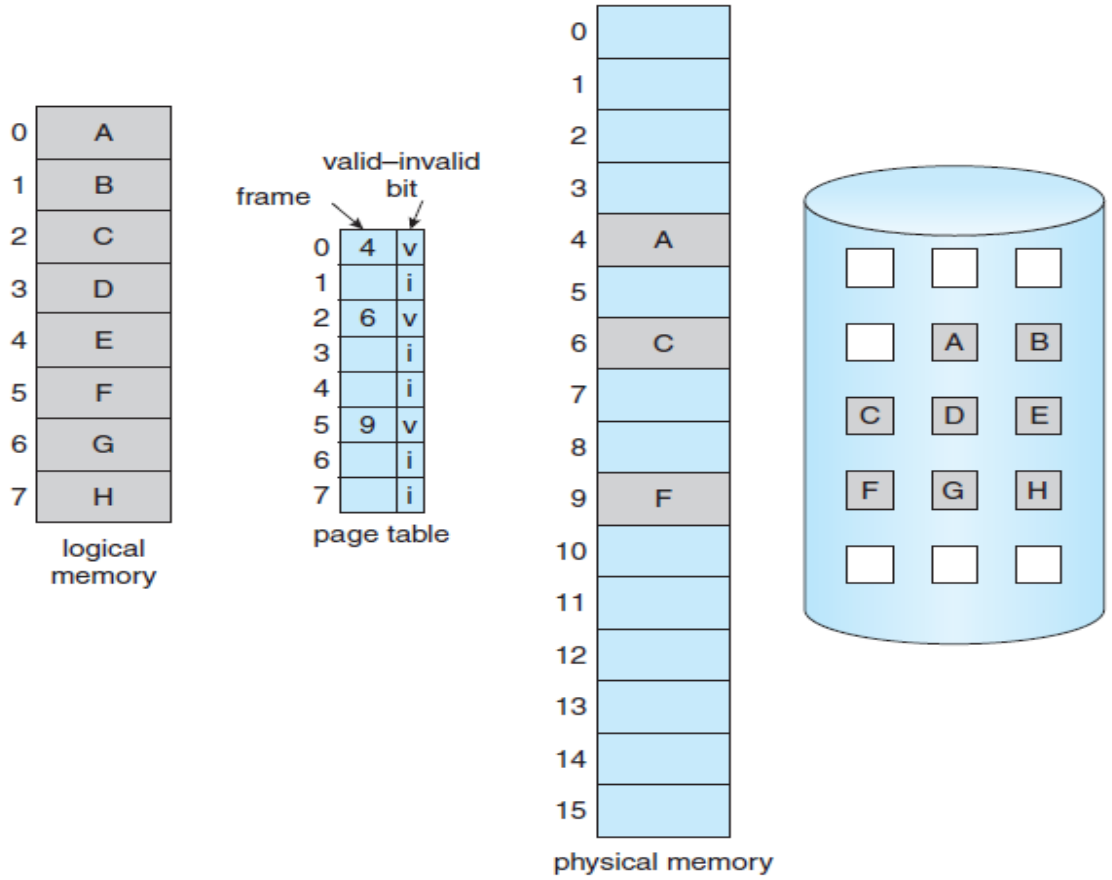
Virtual Memory Implemented by Demand Paging

Virtual memory involves the separation of user view of logical memory from physical memory and commonly implemented by *demand paging*.

Demand paging is a paging system with *lazy swapper*. A *lazy swapper* never swaps a page into memory unless that page will be needed

- The process image resides on secondary memory
- When a process is activated, the pager will swap the page that is needed.
- Here the process is viewed as a sequence of pages rather than a large contiguous memory.
- Access to a page marked invalid causes a page fault trap

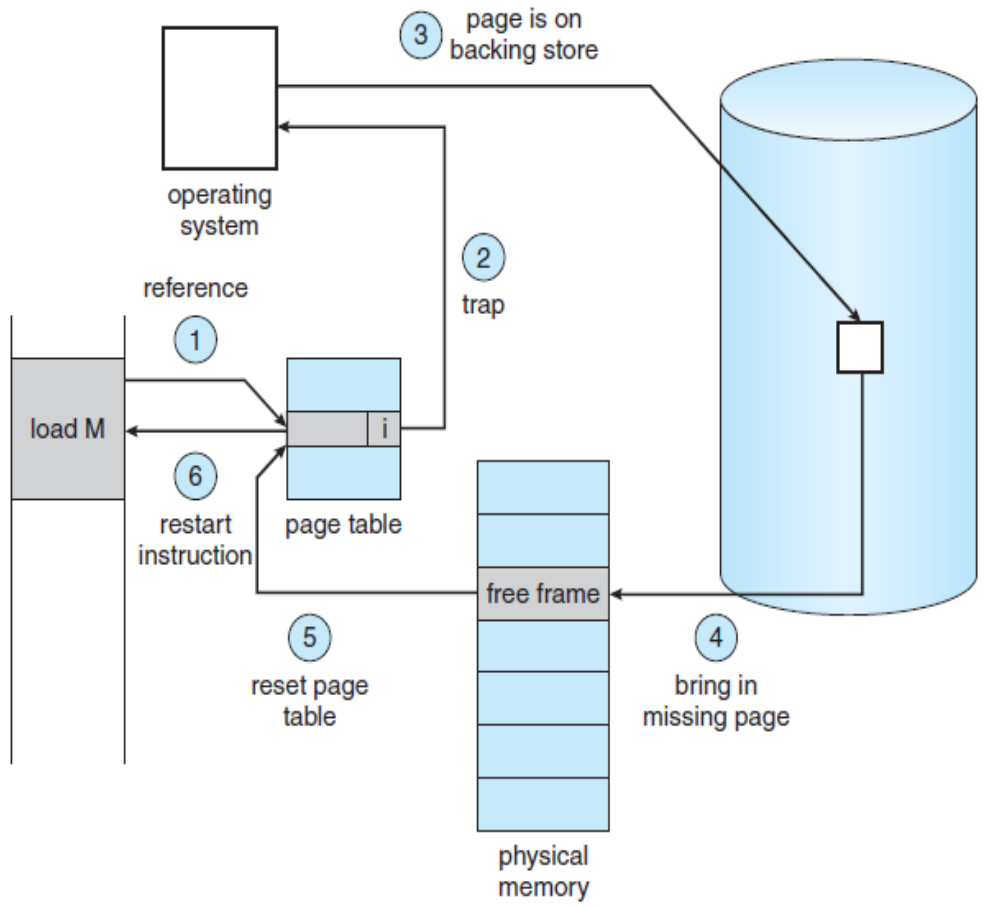
In Demand Paging where only Some Pages are in Memory



Procedure for Handling a Page Fault

1. Check an internal table (usually kept with the process control block) for this process to determine whether the reference was a legal or an illegal memory access.
2. If the reference was illegal, we terminate the process.
3. If it was legal but we have not yet brought in that page, find a free frame (by taking one from the free-frame list).
4. Schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the internal page-map table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Steps in handling a page fault.



Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. As long as there is no page faults, the effective access time is equal to the memory access time.

However, if a page fault occurs, the relevant page is read from disk and then access the desired word.

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

Where

$p \rightarrow$ the probability of a page fault ($0 \leq p \leq 1$).

$ma \rightarrow$ the memory access time

To compute the effective access time, we must know how much time is needed to service a page fault.

Computing the Page Fault Time

A page fault causes the following sequence of steps to occur

1. Trap to Operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
    a. Wait in a queue for this device until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame

Computing the Page Fault Time (cond.)

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Computing the Page Fault Time (cond.)

The three major components of the page-fault service time:

- ✓ Service the page-fault interrupt.
- ✓ Read in the page.
- ✓ Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each.

The second task involves a read from hard disk with

- ➢ a seek time of 5 milliseconds
- ➢ an average latency of 3 milliseconds,, and
- ➢ a transfer time of 0.05 milliseconds.

Thus, the total paging time is about 8 milliseconds.

Page Fault Time Rate

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

Effective Access Time = $(1 - p) \times (200 \text{ nanoseconds}) + p\ (8 \text{ milliseconds})$

$= (1 - p) \times 200 \text{ nanoseconds} + p \times 8{,}000{,}000 \text{ nanoseconds}$

$= 200 \text{ nanoseconds} + 7{,}999{,}800 \times p \text{ nanoseconds}.$

The effective access time is directly proportional to the **page-fault rate.** If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$220 > 200 + 7{,}999{,}800 \times p,$

$20 > 7{,}999{,}800 \times p,$

$p < 0.0000025.$

- This allow fewer than one page-fault out of every 399,990 memory access ,

So it is very important to keep the page fault rate low in demand paging system. The page fault is not a serious problem because each page faults at most once, when it is first referenced (provided enough number of free frames are available).

Page Replacement

In case a process actually uses 50% of the total pages, the demand paging will save the I/O necessary to load the remaining 50% of the pages. As a result the degree of multiprogramming could be increased by running twice as many processes. (over-allocating memory). This increases CPU utilization and throughput.

It may suddenly happen for some data set the process needs more pages and no free frames are available in memory. [This situation is unlikely unless the degree o multiprogramming is increased beyond certain limit(average memory usage is close to the available physical memory)].

The options to deal with this situation are:

- ✖ Terminate the user process (not the good choice)
- ✖ Reduce the degree of multiprogramming
- ✓ Page Replacement

Page Fault Service Routine with Page Replacement

1. Find the location of the desired page on the disk.
2. Find a free frame:
   - If there is a free frame, then use it.
   - else, use a page-replacement algorithm to select a victim frame and Write the victim frame to the disk.
   - Read the desired page into the newly freed frame; modify PMT.
3. Continue the user process from where the page fault occurred.

Modify Bit (or Dirty Bit)

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 (also called dirty 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. If the modify bit is not set, then the page has *not been modified since it was read into memory and* there is no need to write the page back in to memory.

This reduce time by half if the victim page is not modified

Page-replacement Algorithm.

Two major problems to implement demand paging:

- ✓ Frame-allocation Algorithm and
- ✓ Page-replacement Algorithm.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. The selection of a particular replacement algorithm depends on the selection one with the lowest page-fault rate.

A page-replacement algorithm is evaluated 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.**

Reference String

The reference strings can be generated :

1. Artificially by using a random-number generator or
2. Recording the address of each memory reference in actual execution of a process
   - This choice produces a large number of data (on the order of one million addresses per second). To reduce the number of data, we use two facts.
     - First, for a given page, we need to consider only the page number, rather than the entire address.
     - 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.

Reference String (cont.)

Suppose we trace a particular process and record the following address sequence with 100 bytes per page :

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

This sequence is reduced reference string: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

A page-replacement algorithm and the number of available page-frames to the process are needed to determine the number of page faults for a particular reference string .

- To illustrate several page-replacement algorithms, the following reference string is used with three frames.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

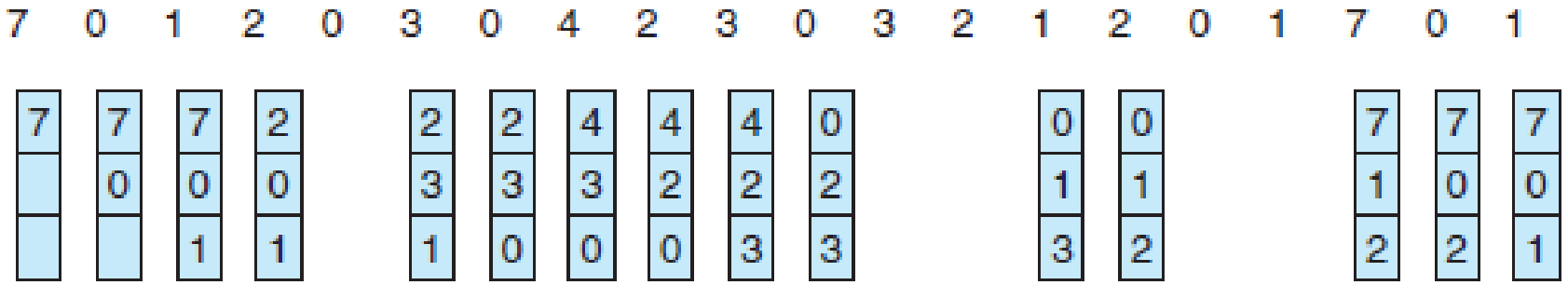
FIFO Page Replacement Algorithm

The first-in, first-out (FIFO) algorithm is the simplest page-replacement algorithm to implement and to understand.

- Principle: A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.
- Implementation: Notice that it is not strictly necessary to record the time when a page is brought in. A FIFO queue can be created to hold all pages in memory and the page at the head of the queue will be replaced. When a page is brought into memory, it will be added at the tail of the queue.

Illustration of finding the number of page-faults using FIFO Page Replacement Algorithm

When three page-frames are available to the process



The number of page-faults is 15.

Belady's Anomaly

Consider the situation that, if we select for replacement a page that is in active use, everything still works correctly.

- When an active page is replaced with a new one, a fault may occurs almost immediately to retrieve the active page and for this some other victim page must be replaced to bring the active page back into memory.
  - So, a bad replacement choice increases the page-fault rate and slows process execution, however, It does not cause incorrect execution.
- For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases and this most unexpected result is known as **Belady's anomaly**

Illustration of Belady's Anomaly

To illustrate the **Belady's anomaly** that are possible with a FIFO page-replacement algorithm,

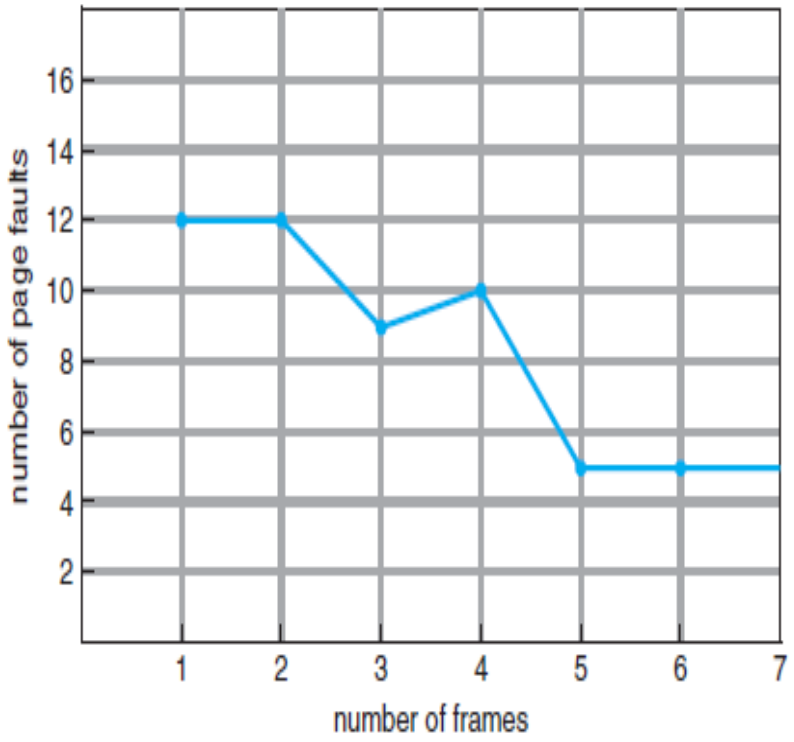
Consider the following reference string:

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

- The right side figure shows the curve of page faults for this reference string versus the number of available frames

! **Notice that *the ten page-faults for four frames* is greater than *nine page-faults for three frames* (Belady's Anomaly)**

Page-Fault Curve



Second Chance Page Replacement Algorithm (Additional-Reference-Bits Algorithm)

It is a modified FIFO page-replacement algorithm.

- Strategy: The reference bit is checked for the page selected by FIFO, if it is zero then OS will proceed with this page replacement else the R-bit will be reset and its arrival time is set to the current time. After that move on to the next page selected by FIFO.

  If a page is used enough it will be never be replaced.

- Implementation
  - FIFO Queue is used
  - Clock Algorithm uses a circular queue where a pointer indicates the page which is to be replaced next.
  - If R-bit is set, it will be reset and advance to next in queue. This is repeated until OS indicates a page with 0 reference bit.

Enhanced Second Chance Page Replacement Algorithm (Additional-Reference-Bits Algorithm)

This algorithm first selects the page encountered in the following lowest non-empty class for replacement.

Classification of Pages

Class	R-bit	M-bit	
I	(0,	0)	Neither recently used nor modified
II	(0,	1)	Not recently used but modified (*Not good choice because the victim page has to be written back*)
III	(1,	0)	Recently used but clean modified (*Probably will be used again*)
IV	(1,	1)	Recently used and modified

Optimal Page-Replacement Algorithm

The optimal page-replacement algorithm is easy to describe but impossible to implement (because it requires the future knowledge of the reference string)

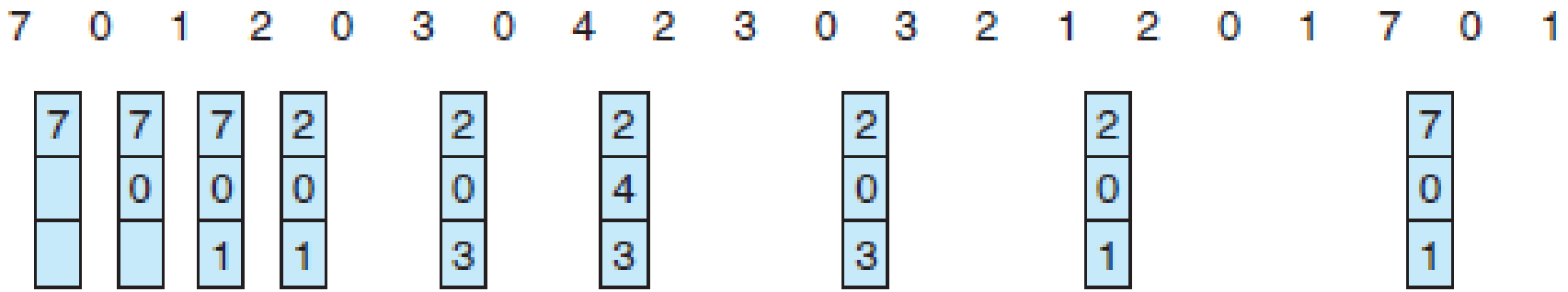
Strategy: Replace the page that will not be used for the longest period of time.

Some pages will be referenced on the very next instruction while other pages may not be referenced until 10, 100 or even 1000 instruction later. This algorithm will select the page which will be referenced after highest number instructions.

- The optimal page-replacement algorithm guarantees the lowest possible page-fault rate and mainly used for comparison studies.
- The optimal page-replacement algorithm will never suffers from Belady's anomaly
- It is possible to implement this algorithm on the second run by using the page reference information collected during the first run.

Illustration of finding the number of page-faults using Optimal Page-Replacement Algorithm

When three page-frames are available to the process



The number of page-faults is 9.

Least Recently Used(LRU) Algorithm

Difference between FIFO and Optimal page replacement algorithm

FIFO Algorithm uses the time when a page was brought in to memory

Optimal Algorithm uses the time when a page is to be used.

↓

LRU Algorithm: If we use the recent past as an approximation of the near future, then we can replace the page that **has not been used for the longest period of time. This approach is the least recently used (LRU) algorithm.**

- LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

Implementation of LRU Algorithm

Two implementations are feasible:

I. Counter Method

- Each page-table entry have an additional field to record time-of-use field.
- Here a logical clock or counter is used. The logical clock is incremented for every memory reference and this clock value is kept in a CPU register.
- 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, PMT has the "time" of the last reference to each page.
- When a page is to be replaced, the page with the smallest value in the time of use field will be replaced. This requires a search of all relevant entries in the page table.

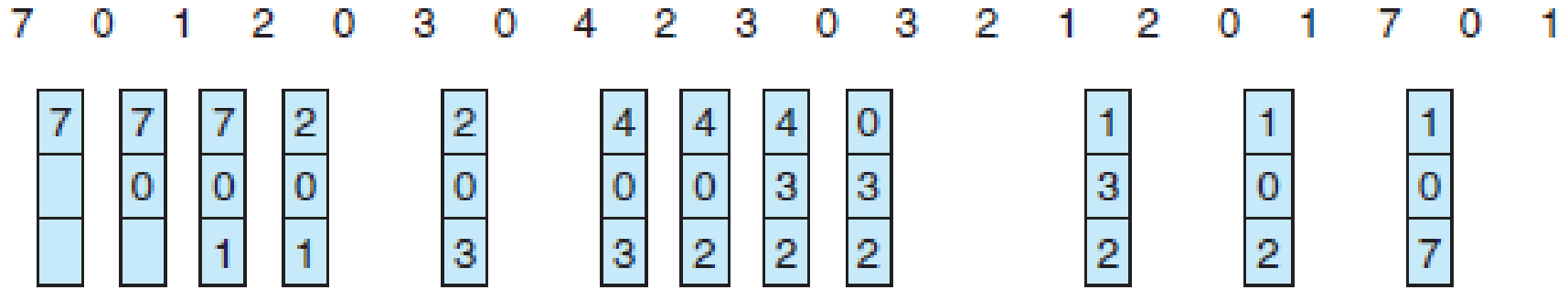
Implementation of LRU Algorithm (cont.)

II. Stack Method

- **Stack** of page numbers is used.
- Whenever a page is referenced, it is removed from the stack and put on the top.
- The top of the stack is always the most recently used page and the least recently used (LRU) page is always at the bottom.
- The stack is implement by using a doubly linked list with a head pointer and a tail pointer because entries must be removed from the middle of the stack.
- Each update is a little more expensive. In the worst case removing a page and putting it on the top of the stack requires changing six pointers. at worst., but there is no search for a replacement.

Illustration of finding the number of page-faults using Least Recently Used(LRU) Algorithm

When three page-frames are available to the process



The number of page-faults is 12.

Stack Algorithms

- Optimal replacement and LRU replacement do 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 will still be in memory.

Counting Algorithms

These algorithms are based on the number of references have been made to each page.

I. **Least frequently Used (LFU) Algorithm**

- The least frequently used (LFU) page-replacement 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.
- A problem arises when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage.

Counting Algorithms (cont.)

II. Most Frequently Used(MFU) Algorithm

- The most frequently used (MFU) page-replacement algorithm replace the page with highest **count**.
- The reason for this selection is that the page with the smallest count was probably just brought in and has yet to be used.

Frame Allocation Algorithms

Equal Allocation Scheme

The easiest way to split m frames among n processes is to give everyone an equal share, $\lfloor m/n \rfloor$ frames (ignoring frames needed by the operating system for the moment).

- For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool. This scheme is called equal allocation.

Proportional Allocation Scheme

In this scheme the available memory will be allocated to each process according to its size. Let the size of the virtual memory for process p_i *be* s_i *, and define* $S = \sum s_i$.

Then, if the total number of available frames is *m, we allocate* a_i *frames to* process p_i,

$$a_i = s_i/S \times m.$$

we must adjust each a_i *to be an integer that is greater than the* minimum number of frames required by the instruction set, with a sum not exceeding *m*.

Global versus Local Allocaton

An another factor that influence the way frames are allocated to the various processes in page-replacement.

Global Page Replacement

One process can take frames from another

High-priority process can select from low-priority process

This change the number of frames allocated to processes

- Increase in the number of frames allocated to high priority process
- decrease in the number of frames allocated to low-priority process

Local Page Replacement

Process selects victim frame from its own set of allocated frames

Thrashing

If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called **thrashing.**

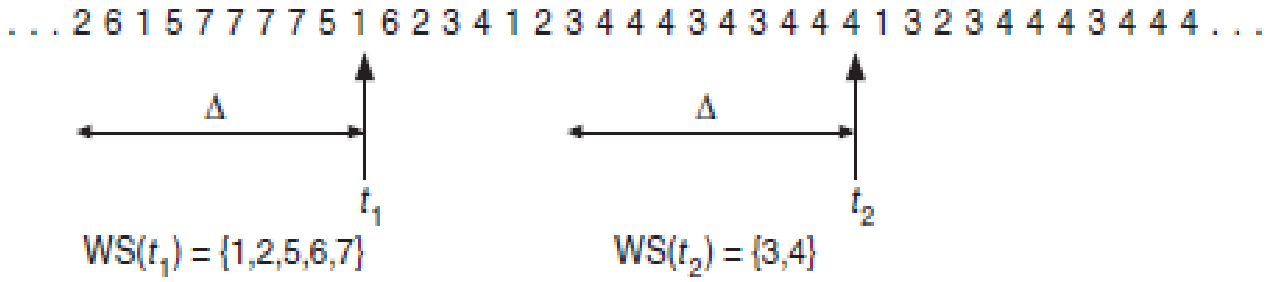
A process is thrashing if it is spending more time paging than executing.

Working-Set Model

This working-set model is based on the assumption of *locality of reference* and uses a parameter **'Δ'** to define the working-set window. The idea is to examine the most recent page references.

Working Set = The set of pages in the most* recent *page references

If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set time units after its last reference. Thus, the working set is an approximation of the program's locality. For example, given the sequence of memory references as shown in Figure with Δ = 10 memory references, then the working set at time *t1 is* **WS**(t_1) = {1, 2, 5, 6, 7}. By the time *t2,* the working set has changed to **WS**(t_2) = {3, 4}.



Working-Set Model (cont.)

The accuracy of the working set depends on the selection of working set size Δ.

If *is too* small,

it will not encompass the entire locality;

if *is too large,*

it may overlap several localities.

if *is infinite,*

the working set is the set of pages touched during the process execution.

So, the most important property of the working set is its size. To compute the working-set size, WSS_i , *for each process in the system, we can* consider that

$$D = WSS_i$$

Where

$D \rightarrow$ *the total demand for frames.*

Working-Set Model (cont.)

Each process is actively using the pages in its working set. Thus, i^{th} process needs WSS_i frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have sufficient frames.

Once *has been selected, use of the working-set model is simple. The* operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped out), and its frames are reallocated to other processes. The suspended process can be restarted later time.

Working-Set Model (cont.)

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window.

At each memory reference, a new reference appears at one end, and the oldest reference drops off the other end.

A page is in the working set if it is referenced anywhere in the working-set window.