

## Chapter 10: Virtual Memory

- Background
- Demand Paging
- Performance of Demand Paging
- Page Replacement
- Page-Replacement Algorithms

## Background

- In Chapter 8, we discussed various memory-management strategies.
- All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming.
- Virtual memory is a technique that allows the execution of processes that may not be completely in memory.
- Advantage of virtual memory is that programs can be larger than physical memory.
- In many cases, the entire program is not needed to be in physical memory (examples):
  - Programs have code to handle unusual error conditions.
  - Arrays, lists, and tables are often allocated more memory than they actually need.
  - Certain options and features of a program may be used rarely.

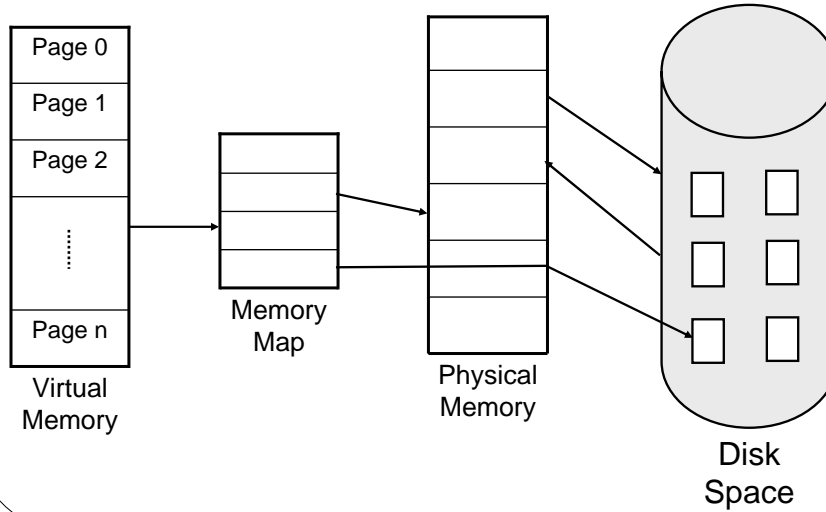
## Background (Cont.)

- The ability to execute a program that is only partially in memory would have many benefits:
  - A program would no longer be constrained by the amount of physical memory that is available.
  - Because each user program could take less physical memory, more programs could be run at the same time, with 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 each user program into memory, so each user program would run faster.

## Background (Cont.)

- Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Need to allow pages to be swapped in and out.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

### Diagram: virtual memory larger than physical memory



Operating System Concepts

10.5

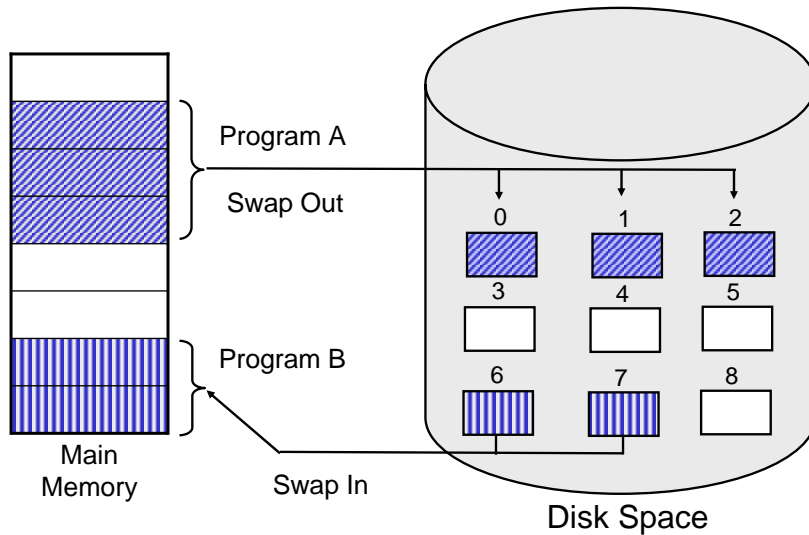
### Demand Paging

- A demand-paging system is similar to a paging system with swapping.
- We use a lazy swapper, a lazy swapper never swaps a page into memory unless that page will be needed.
- Since we swap pages not entire process we call it pager instead of swapper.
- Since the pager swaps only the pages are needed to be in physical memory (not entire process), this will yield to:
  - Less swap time
  - Less I/O needed
  - Less amount of physical memory needed
  - More users
  - Faster response time

Operating System Concepts

10.6

## Transfer of a Pages Memory to Contiguous Disk Space



## Valid-Invalid bit

- We need hardware support to distinguish between those pages that are in memory and those pages are on the disk.
- The valid-invalid bit scheme can be used:
  - Valid: indicates that the associated pages is both legal and in memory.
  - Invalid: indicates that the page either is not valid (not in logical address space) or is valid but is currently on the disk.
- What happens if the process tries to use a page that was not brought into memory?
- Access to a page marked invalid causes a page-fault trap.

## Valid-Invalid Bit (Cont.)

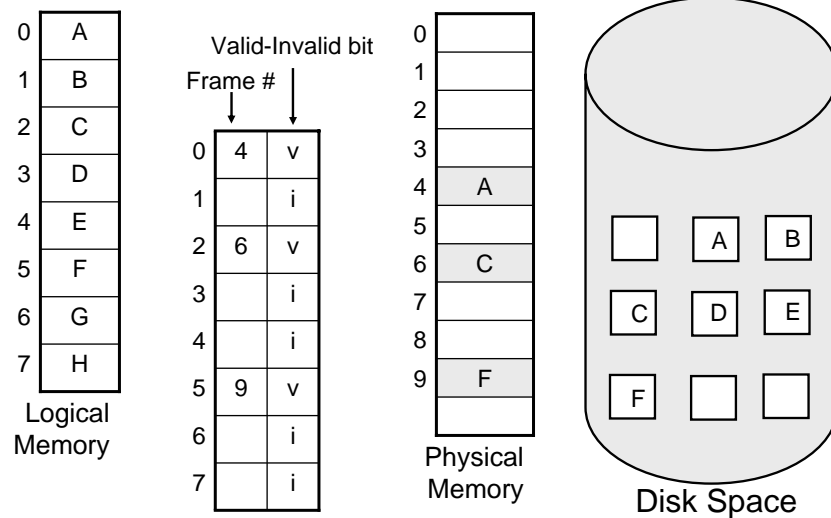
- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid bit is set to 0 on all entries.
- Example of a page table snapshot.

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

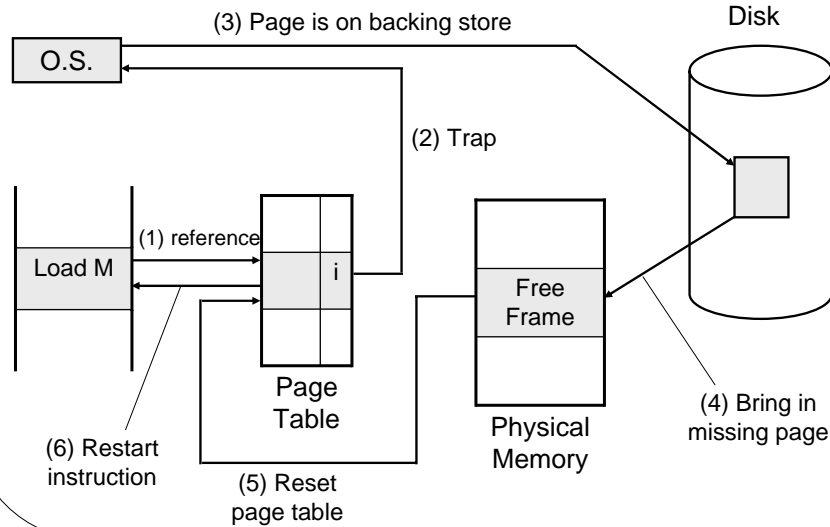
page table

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault.

## Page Table: when some pages are not in main memory



## Steps in Handling a Page Fault



Operating System Concepts

10.11

## Steps in Handling a Page Fault (Cont.)

1. We check an internal table for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate process. If it was valid, but we have not yet brought in that page, we now page in the latter.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk 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 illegal address trap. The process can now access the page as though it had always been in memory.

Operating System Concepts

10.12

## What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
  - Algorithm.
  - Performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

## Performance of Demand Paging

- Effective Access Time (EAT) for a demand-paged memory.
- Memory Access Time ( $m_a$ ) for most computers now ranges from 10 to 200 nanoseconds.
- If there is no page fault, then  $EAT = m_a$ .
- If there is page fault, then
$$EAT = (1 - p) \times (m_a) + p \times (\text{page-fault time}).$$

$p$ : the probability of a page fault ( $0 \leq p \leq 1$ ), we expect  $p$  to be close to zero ( a few page faults).  
If  $p=0$  then no page faults, but if  $p=1$  then every reference is a fault
- If a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

## Performance of Demand Paging (Cont.)

- We are faced with three major components of the page-fault service time:
  1. Service the page-fault interrupt.
  2. Read in the page.
  3. Restart the process.
- A typical hard disk has:
  - An average latency of 8 milliseconds.
  - A seek of 15 milliseconds.
  - A transfer time of 1 milliseconds.
  - Total paging time =  $(8+15+1)= 24$  milliseconds, including hardware and software time, but no queuing (wait) time.

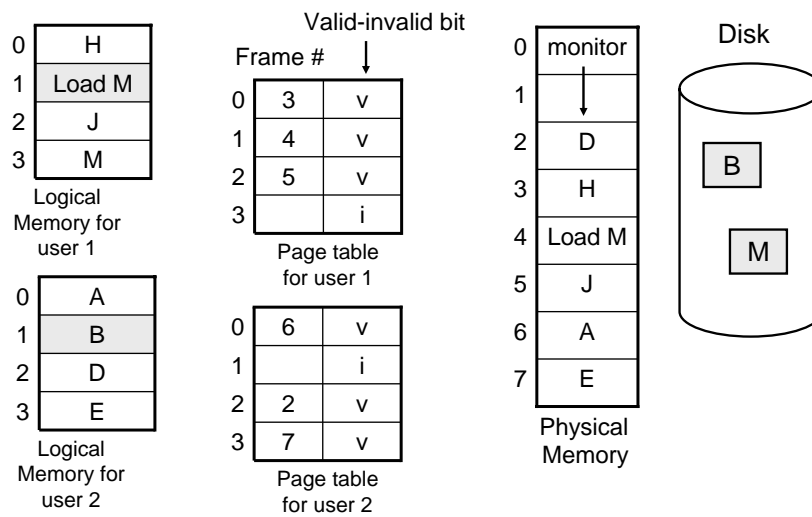
## Demand Paging Example 1

- Assume an average page-fault service time of 25 milliseconds ( $10^{-3}$ ), and a Memory Access Time of 100 nanoseconds ( $10^{-9}$ ). Find the Effective Access Time?
- Solution: Effective Access Time (EAT)
$$= (1 - p) \times (ma) + p \times (\text{page fault time})$$
$$= (1 - p) \times 100 + p \times 25,000,000$$
$$= 100 - 100 \times p + 25,000,000 \times p$$
$$= 100 + 24,999,900 \times p.$$
- Note: The Effective Access Time is directly proportional to the page-fault rate.

## Page Replacement

- Example: Assume each process contains 10 pages and uses only 5 pages.
  - If we had 40 frames in physical memory then we can run 8 processes instead of 4 processes. Increasing the degree of multiprogramming.
  - If we run 6 processes (each of which is 10 pages in size), but uses only 5 pages. We have higher CPU utilization and throughput, also 10 frames to spare (i.e.,  $6 \times 5 = 30$  frames needed out of 40 frames).
  - It is possible each process tries to use all 10 of its pages, resulting in a need for 60 frames when only 40 are available.

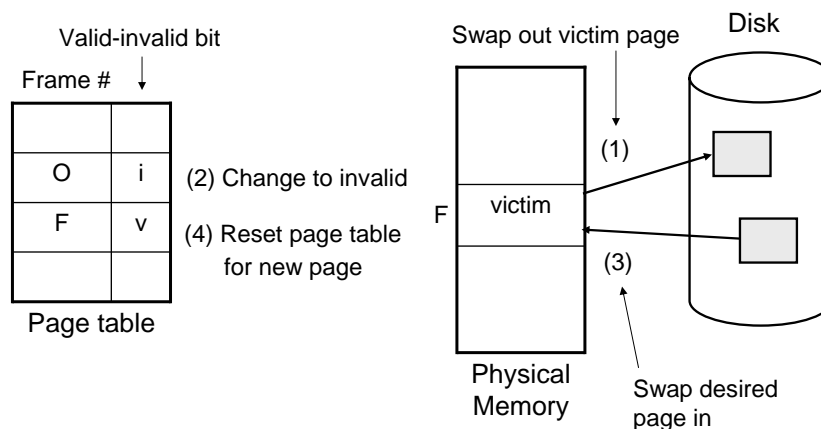
## Need for Page Replacement



## The Operating System has Several Options

1. Terminate user process.
2. Swap out a process, freeing all its frames, and reducing the level of multiprogramming.
3. Page replacement takes the following approach:
  - If no frames is free, find one that is not currently being used and free it.
  - We can free a frame by writing its contents to swap space, and changing the page table to indicate that the page is no longer in memory.

## Page Replacement



## Page Replacement (Cont.)

- The page-fault service time is now modified to include page replacement:
  1. Find the location of the desired page on the disk.
  2. Find a free frame:
    - If there is a free frame use it.
    - Otherwise, use a page-replacement algorithm to select a victim frame.
    - Write the victim page to the disk; change the page and frame tables accordingly.
  3. Read the desired page into the newly free frame; change the page and frame tables.
  4. Restart the user process.

## Page Replacement (Cont.)

- Note: If no frames are free, two page transfers (one out and one in) are required. This doubles the page-fault service time and will increase the effective access time accordingly.
- This overhead can be reduced by the use of a modify (dirty) bit.
- Each page or frame may have a modify bit associated with it in the hardware.
- To implement demand paging, we must develop:
  - Frame-allocation algorithm, to decide how many frames to allocate to each process.
  - Page-replacement algorithm, to select the frames that are to be replaced.

## Page-Replacement Algorithms

- We want a page replacement algorithm with the lowest page-fault rate.
- We evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- The string of memory references is called a reference string.
- We can generate reference strings by tracing a given system and recording the address of each memory reference.
- In our examples, the reference string is

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

## Page-Replacement Algorithms (Cont.)

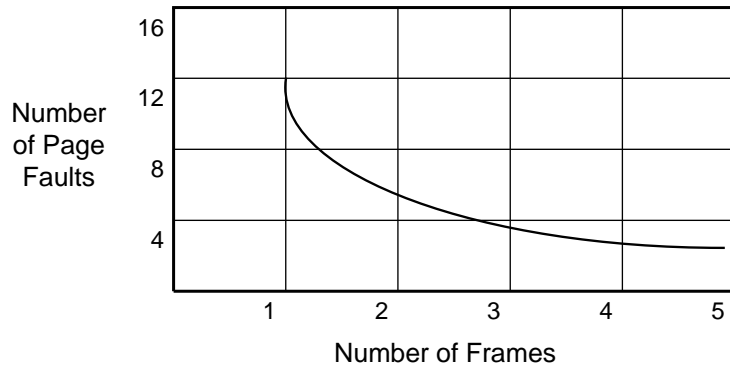
- Example: If we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0102, 0609, 0601, 0612  
↓     ↓                    ↓                    ↓  
1     4                    1                    6

This is reduced to the following reference string: 1, 4, 1, 6

- As the number of frames available increases, the number of page faults will decrease.
- From the above example: If we had 3 or more frames, we would have only 3 faults, one fault for the first reference to each page. If we had only one frame, we would have a replacement with every reference resulting in 4 faults.

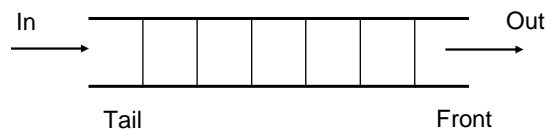
## Graph of Page Faults vs. Number of Frames



As you can see, as the number of frames increases, the number of page faults drops.

## First-In-First-Out (FIFO) Algorithm

- A FIFO 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.
- Or we can use a FIFO queue to hold all pages in memory.
- We replace the page at the head of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.



## Example: FIFO Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Let 3 frames are initially empty (3 pages can be in memory at a time per process).
- The first 3 references (1, 2, 3) cause page faults, and are brought into these empty frames.

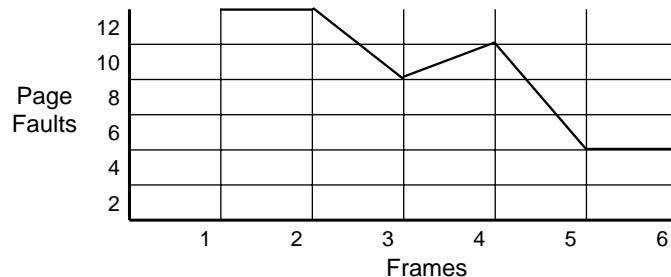
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

## Graph: Curve of page faults vs. number of available frames.

- FIFO algorithm is easy to understand and to programs.
- A bad replacement choice increases the page-fault rate and slows process execution.



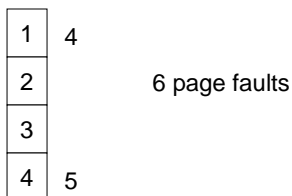
- Belady's anomaly: For some page-replacement algorithms, the page fault rate may increase as the number of allocated frames increases.

## Optimal (OPT) Algorithm

- An optimal algorithm has the lowest page-fault rate of all algorithms.
- An optimal algorithm will never suffer from Belady's anomaly.
- Replace the page that will not be used for the longest period of time.
- This algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.
- The optimal algorithm is difficult to implement, because it requires future knowledge of the reference string.
- Similar situation with Shortest-Job-First in CPU scheduling.

## Example: OPT Algorithm

- Initially 4 frames empty.
- Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



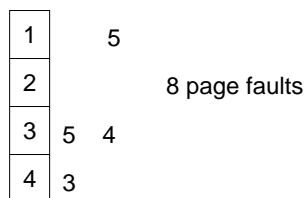
- How do you know this?
- Used for measuring how well your algorithm performs.

## Least Recently Used (LRU) Algorithm

- The key distinction between FIFO and OPT algorithms is that FIFO uses the time when a page was brought into memory; the OPT uses the time when a page is to be used (future).
- LRU algorithm uses the time when a page has not been used for the longest period of time (Past).
- LRU replacement associates with each page the time of that page's last use.

## Example: LRU Algorithm

- Looking backward in time.
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- Note: Number of page faults (on same reference string) using:
  - LRU is 8.
  - FIFO is 10.
  - OPT is 6.

## LRU Algorithm Implementations

- LRU algorithm may require hardware assistance; two implementations:
  1. Counter implementation:
    - 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 content of the clock register is copied to the time-of-use field in the page table for that page.
    - We replace the page with the smallest time value.
    - Requires a search of the page table to find LRU page and a write to memory for each memory access.
    - In summary, every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
    - When a page needs to be changed, look at the counters to determine which are to change.

## LRU Algorithm Implementations (Cont.)

2. Stack implementation:
  - Keep a stack of page numbers.
  - Whenever a page is referenced, it is removed from the stack and put on the top.
  - Top of the stack is always the mostly recently used page and the bottom is the LRU page.
  - Because entries must be removed from the middle of the stack; it is implemented by a double linked list with a head and tail pointer.
  - The tail pointer points to the bottom of the stack (which is LRU page).
- Neither OPT nor LRU suffers from Belady's anomaly.



## Reference Bit

- Reference bit for a page is set by hardware.
  - With each entry in the page table associate a bit, initially = 0.
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists).
  - We do not know the order, however.

## Additional-Reference-Bits Algorithm (8-bits).

- Use 8-bits (byte) for each page in the table.
- Example1: 0000 0000 means this page not been used for 8 periods of time. So, it is LRU.
- Example2: 1111 1111 means this page have been used (referenced) 8 times.
- Note: 1100 0000 > 0011 1111.
- The numbers are not unique
- We can either replace (swap out) all pages with smallest value (have same value), or use a FIFO selection among them.

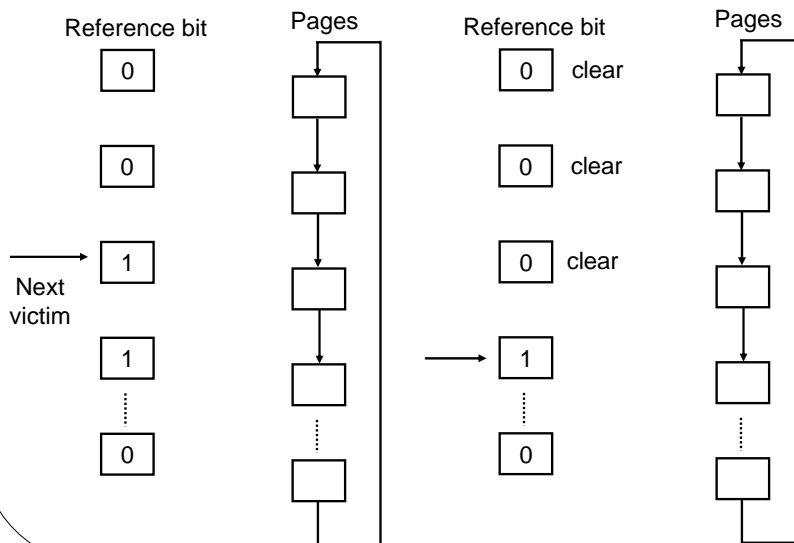
## **Second-Chance Algorithm (one-bit).**

- The basic algorithm of second-chance is a FIFO.
- When a page has been selected, inspect its reference bit:
  1. If the value is 0 replace the page.
  2. If the value is 1, we give that page a second chance and move on to select the next FIFO page.
- When a page gets a second chance:
  1. Its reference bit is cleared.
  2. Its arrival time is reset to the current time.
  3. Will not be replaced until all other pages are replaced or given second chance.

## **Second-Chance Algorithm's Implementation.**

- 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 reference 0.
  - As it advances, it clears the reference bits.
  - Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position.

### Example: Second-Chance Algorithm's Implementation.



Operating System Concepts

10.41

### Enhanced Second-Chance Algorithm (2-bits).

- Using the reference bit and the modify bit as an ordered pair.
- With 2-bits we have the following four possible classes:
  1. (0,0)– neither recently used nor modified (best page to replace).
  2. (0,1)– not recently used but modified the page will need to be written out before replacement.
  3. (1,0)– recently used but clean (probably will be used again).
  4. (1,1)– recently used and modified probably will be used again and write out will be needed before replacing it.
- This algorithm is used in Macintosh virtual-memory-management.

Operating System Concepts

10.42

## Counting Algorithms

- Keep a counter of the number of references that have been made to each page, and develop the following two schemes:
  1. LFU (Least Frequently Used) Algorithm:
    - Replaces page with smallest count.
    - Suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.
  2. MFU (Most Frequently Used) Algorithm:
    - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.