TDIU11

Operating systems

Virtual Memory

[SGG7/8/9] Chapter 9

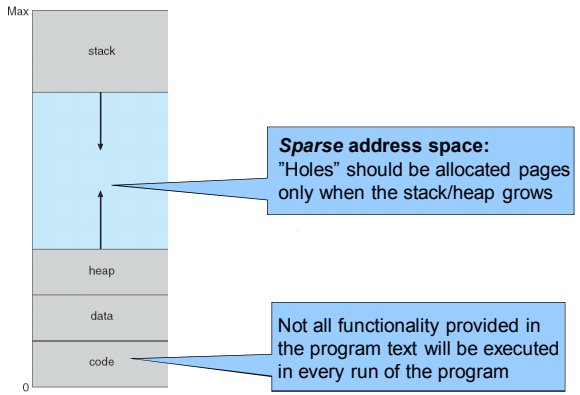
Christoph Kessler, IDA,
Linköpings universitet.

Overview: Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing and Data Access Locality

- Appendix: Additional Slides on Virtual Memory
  - Process Creation: Copy-on-Write
  - I/O Interlock

Virtual address space of a process



Virtual Memory

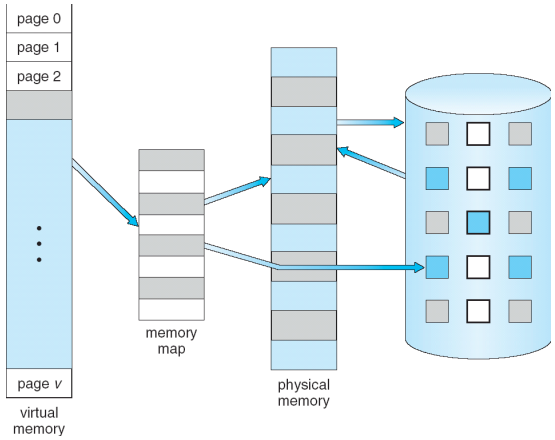
- Previously achieved:
  Separation of user logical memory from physical memory
  - ++ protection, reuse
  - Still, the whole program and data must be loaded in memory

- **Virtual memory**

  Idea: Only part of the program needs to be in memory for execution.
  - Throw out pages currently not used (to secondary memory)
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

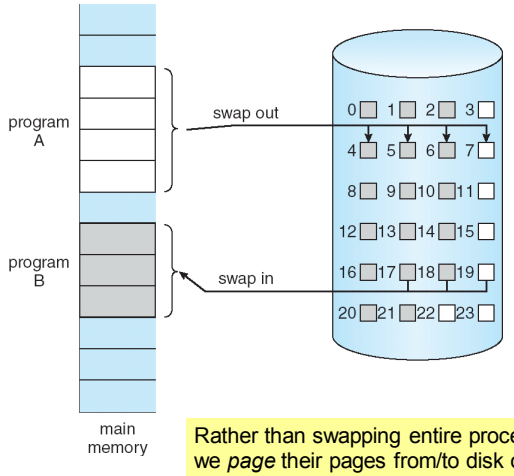


Demand Paging

[Kilburn *et al.* 1961]

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed if referenced (load/store, data/instructions)
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Demand paging ("lazy paging", "lazy swapping"): Transfer of a Paged Memory to Contiguous Disk Space



Rather than swapping entire processes (cf. *swapping*), we *page* their pages from/to disk only when first referenced.

Valid-Invalid Bit

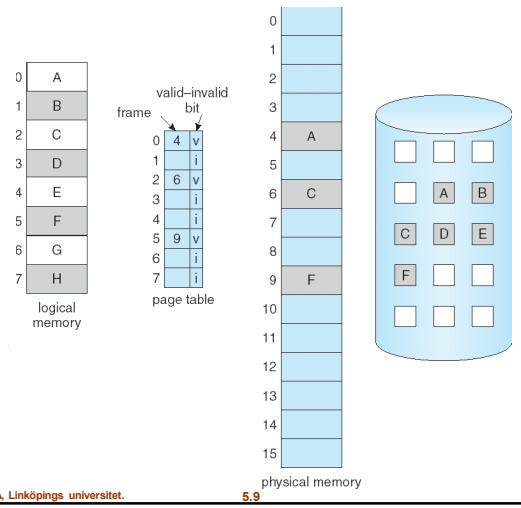
- With each page table entry, a **valid–invalid bit** is associated (1 ⇒ page in memory, 0 ⇒ not-in-memory)
- Initially, valid–invalid bit is set to 0 on all entries
- Example of a page table snapshot:

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

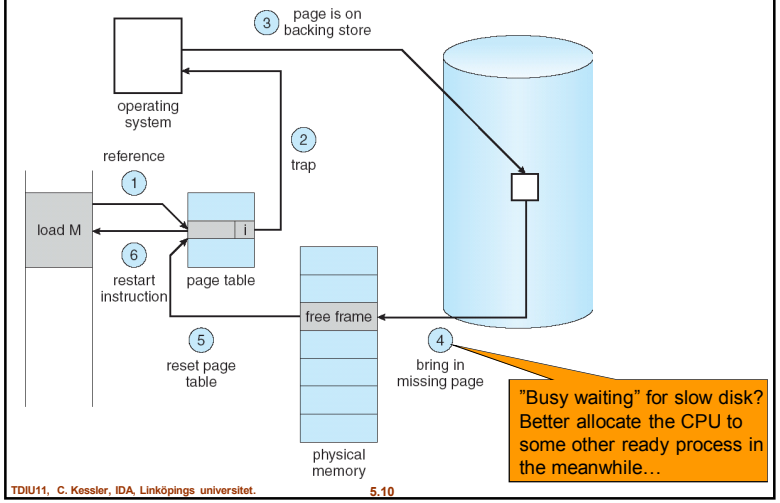
page table

- 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
(Case: a free frame exists)



What happens if there is no free frame?

- ***Page replacement*** –
  find some page in memory, but not really in use,
  swap it out
  - Write-back only necessary if victim page was modified
  - Same page may be brought into memory several times
  - Various algorithms
  - Performance
    - want an algorithm that minimizes number of page faults

Performance of Demand Paging

- ***Page Fault Rate*** p $\quad 0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- Memory access time t

- **Effective Access Time** (EAT)

$$\begin{aligned} EAT &= (1 - p)\, t \\ &\quad + p \cdot (\text{page fault overhead} \\ &\qquad + \text{time to swap page out, if modified} \\ &\qquad + \text{time to swap new page in} \\ &\qquad + \text{restart overhead} \\ &\qquad + t \ ) \end{aligned}$$

Demand Paging Example

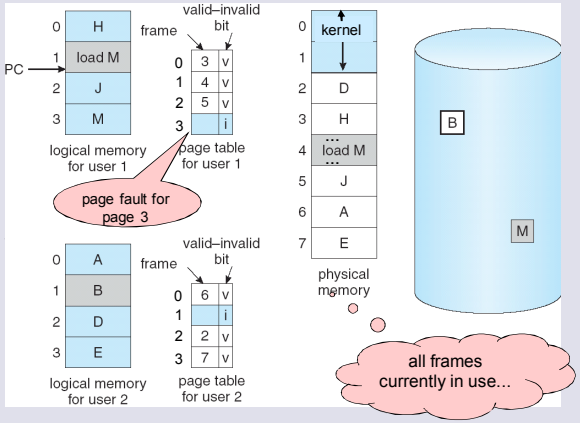
- ***Write-back rate*** *w* *0 <= w <= 1*
  % of page faults where page replacement is needed and the victim page has been modified so it needs to be swapped out

- **Example**:
  - Memory access time = 1 microsecond (μs)
  - Time for swapping a page = 10 ms = 10,000 μs
  - Write-back rate *w* = 0.5 = 50%

  → expected swap time per page fault = (1+*w*) * 10,000 μs

  → EAT = (1 – *p*) · 1μs + *p* (15,000 μs)
  = 1 μs + *p* · 14999 μs

Example: Need For Page Replacement



Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

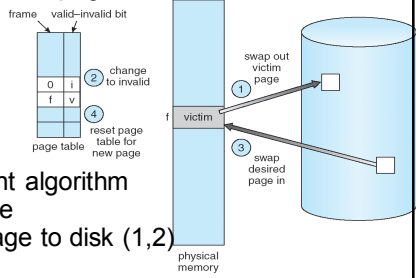
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

Extended page-fault service routine:

- Find the location of the desired page on disk

- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim** frame and, if dirty, write its page to disk (1,2)



- Read the desired page into the (newly) free frame (3) Update the page table (4)

- Restart the process

How to compare algorithms for page replacement?

- **Goal:** find algorithm with lowest page-fault rate.
- **Method:** Simulation.
  - Assume initially empty page table
  - Run algorithm on a particular string of memory references (reference string – page numbers only)
  - Count number of page faults on that string.
- In all our examples, the reference string is

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

First-In-First-Out (FIFO) Algorithm

- Use a time stamp or a queue
- *Victim* is the "oldest page"
- Assume table size = 3 frames / process
  (3 pages can be in memory at a time per process)
  and reference string:

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

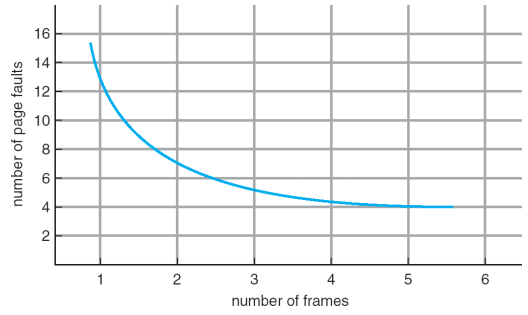
1	1	1	**4**	4	4	**5**	5	5	5	5	*5*
	2	2	2	**1**	1	1	*1*	1	**3**	3	3
		3	3	3	**2**	2		*2*	2	**4**	4

A total of 9 page faults

After page 3 is loaded, page 1 is the oldest of them all (gray box)

The fact that we re-use an existing page does not alter who has been in there the longest...

Expected Graph of Page Faults Versus Number of Frames



Generally, more frames => Less page faults **?**

Same FIFO: More frames = better ?

- 4 frames/process
  (4 pages can be in memory at a time per process)
- Reference string:

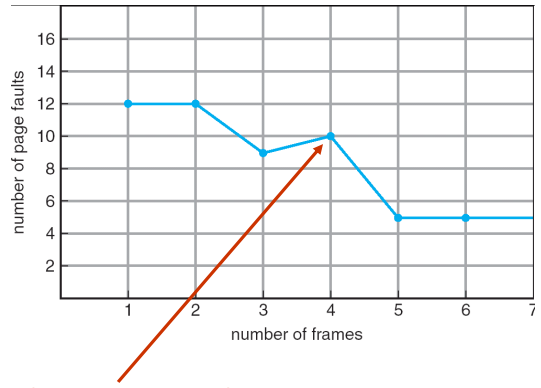
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	*1*	1	**5**	5	5	5	**4**	4
	2	2	2	2	2	2	**1**	1	1	1	**5**
		3	3	3	3	3	3	**2**	2	2	2
			4	4	4	4	4	4	**3**	3	3

A total of 10 page faults

- FIFO Replacement – **Belady's Anomaly**
  - more frames with more page faults – possible!

FIFO illustrating Belady's Anomaly



An Optimal Algorithm [Belady 1966]

- **"optimal":**
  - has the lowest possible page-fault rate (NB: still ignoring dirty-ness)
  - does not suffer from Belady's anomaly
- **Belady's Algorithm:** ***Farthest-First, MIN, OPT***
  - Replace page that ***will*** *not be used for the longest period of time....*
  - How do you know this?
- Example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	*1*	1	1	*1*	1	1	**4**	1	
	2	2	2	2	2	2	2	2	2	2	2	
		3	3	3	3	3	3	3	*3*	3	3	
			4	4	4	**5**	5	5	5	5	*5*	

A total of 6 page faults

Remark: Belady's algorithm is only optimal if there are no dirty write-backs. Otherwise it is just a heuristic algorithm.

Theoretical algorithm! Used for measuring how well your algorithm performs, e.g., *"is it within 12% from the optimal?"*

We will need frames 1, 2 and 3 before we need frame 4 again, thus throw it out!

Least Recently Used (LRU) Algorithm

- Optimal algorithm not feasible?
  ....try using recent history as approximation of the future!
- **LRU Algorithm:**
  - Replace the page that ***has*** *not been used for the longest period of time*
- Example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	*1*	1	1	*1*	1	1	1	**5**	
	2	2	2	2	2	2	2	2	2	2	2	
		3	3	3	3	**5**	5	5	5	**4**	4	
			4	4	4	4	4	4	**3**	3	3	

A total of 8 page faults

Out of pages 1,2,3 and 4, page 3 is the one *not* used for the longest time...

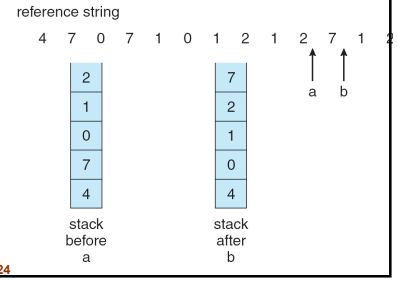
LRU Algorithm Implementations

- **Timestamp implementation**
  - The process maintains a *logical clock* (counter for number of memory accesses made)
  - Every page entry has a *timestamp field*
  - Every time a page is referenced, copy the logical clock into the timestamp field
  - When a page needs to be replaced, search for oldest timestamp

Overhead!

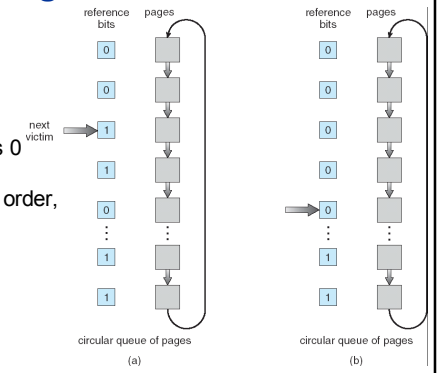
Linear search!

- **Stack implementation**
  - keep a stack of page numbers in doubly linked form:
  - page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement



LRU Approximation Algorithms

- **Reference bit for each page**
  - Initially = 0 (by OS)
  - When page is referenced, bit set to 1
  - Replace a page whose bit is 0 (if one exists).
  - We do not know the access order, however.
  - May improve precision by using several bits.



- **Second chance algorithm**
  - Clockwise replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Not common in today's operating systems

Allocation of Frames

- Each process needs a *minimum* number of pages
  - Example:
    IBM 370 – 6 pages to handle SS MOVE instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle *from*
    - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- *Equal allocation*
  - Example: if there are 100 frames and 5 processes, give each process 20 frames.
- *Proportional allocation*
  - Allocate according to the size of process

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

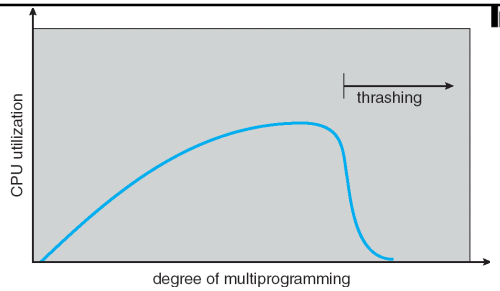
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority

Global vs. Local Allocation

- **Global replacement**
  - process selects a replacement frame from the set of all frames;
  - one process can take a frame from another
- **Local replacement**
  - each process selects from only its own set of allocated frames

Thrashing



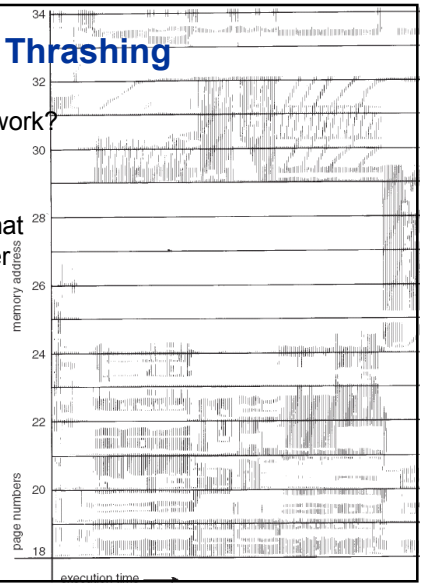
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Demand Paging and Thrashing

- Why does demand paging work?

  Locality model
  - *Locality* = set of pages that are actively used together
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ sizes of localities > total memory size

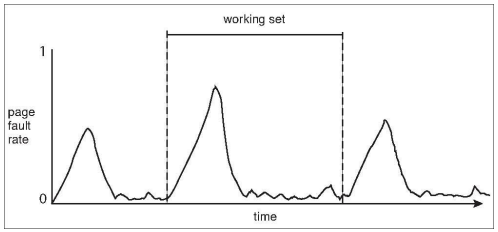


Working-Set Model [Denning, 1968]

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$



Working-Set Model [Denning, 1968]

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

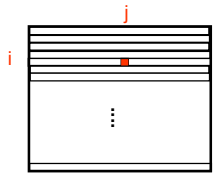
$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

- $\Delta \equiv$ **working-set window** $\equiv$ a fixed number of page references
- $WS_i = WS_i(t)$ = **working set** of process P_i at current time t = set of pages referenced in the most recent Δ accesses
  - if Δ too small, WS will not encompass entire locality
  - if Δ too large, WS will encompass several localities
  - if $\Delta = \infty \Rightarrow$ WS will encompass entire program
- $WSS_i = |WS_i|$ = **working set size** of process P_i at current time = total number of pages referenced in the most recent Δ accesses
- $D = \Sigma_i WSS_i \equiv$ total demand frames
  - if $D > m \Rightarrow$ Thrashing
  - Policy: if $D > m$, then suspend one of the processes

WS can be derived e.g. by simulation, by sampling or tracing (with hardware support).

Program Structure Matters!

- Example: Matrix initialization
  - int data [128][128];
    - Each row is stored in one page
  - Assume page size = 128 ints
  - Assume < 128 frames available, LRU repl.



- Program 1:
```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)      // traverse column-wise
        data[i][j] = 0;
```
-> 128 x 128 = 16,384 page faults

- Program 2:
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)      // traverse row-wise
        data[i][j] = 0;
```
-> 128 page faults

Data access locality is an important issue for programmers and optimizing compilers

TDIU11

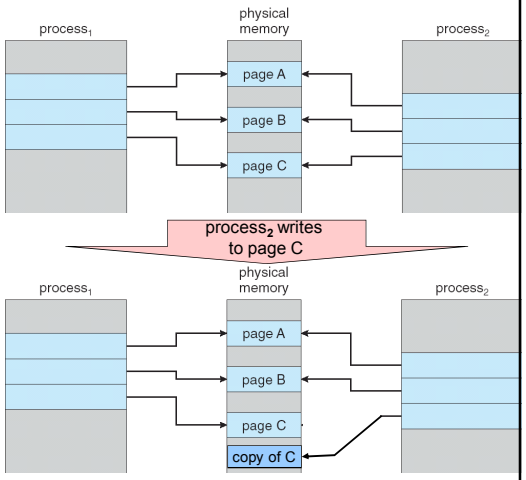
Operating systems

Additional Slides on Virtual Memory

Christoph Kessler, IDA, Linköpings universitet.

Process Creation: Copy-on-Write

- **fork**(): Child process gets a copy of parent process' address space
- *Lazy copying*

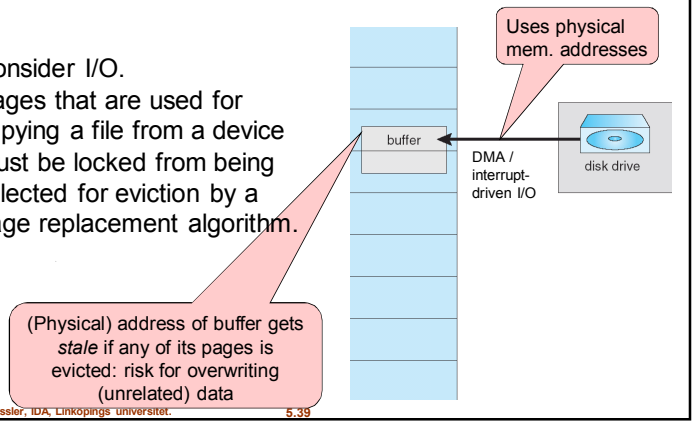


Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then the page is copied
  - Free pages are allocated from a pool of zeroed-out pages
  - More efficient process creation as only modified pages are copied
  - No improvement if child process does **exec** (overwrites all)
- Alternatively: **vfork()** in Unix, Solaris, Linux:
  - Light-weight **fork** variant: sleeping parent and immediate **exec**() by child process

I/O interlock

- **I/O Interlock**
  – Pages must sometimes be locked in memory
- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

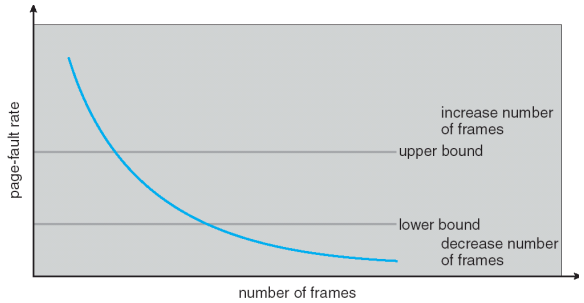


Data used by Page Replacement Algorithms

- All algorithms need extra data in the page table, e.g., one or several of the following:
  - A *reference bit* to mark if the page has been used
  - a *modify* (*dirty*) *bit* to mark if a page has been written to (changed) since last fetched from SM
  - Additional mark bits
  - Counters or time stamps (used for theoretical algorithms – must often be converted into use of mark bits)
  - A queue or stack of page numbers...

Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



References

- Belady, L.: A Study of Replacement of Algorithms for a Virtual Storage Computer. *IBM Systems Journal* 5 (1966) 78–101
- Peter J. Denning: The working set model for program behavior, *Communications of the ACM* 11(5):323-333, May 1968
- Peter J. Denning: The Locality Principle. *Communications of the ACM* 48(7):19-24, July 2005.