Announcements

- Assignment 2 is online; due Monday 3/22.
- No office hours on Thursday:
  - Ask assignment 2 questions during Friday office hours.
  - Or call on Sunday at home from 5p-7p at 829 5639.
- Final examination is open book.
- Questions from last lecture?
- Revisit Segmentation with Paging.

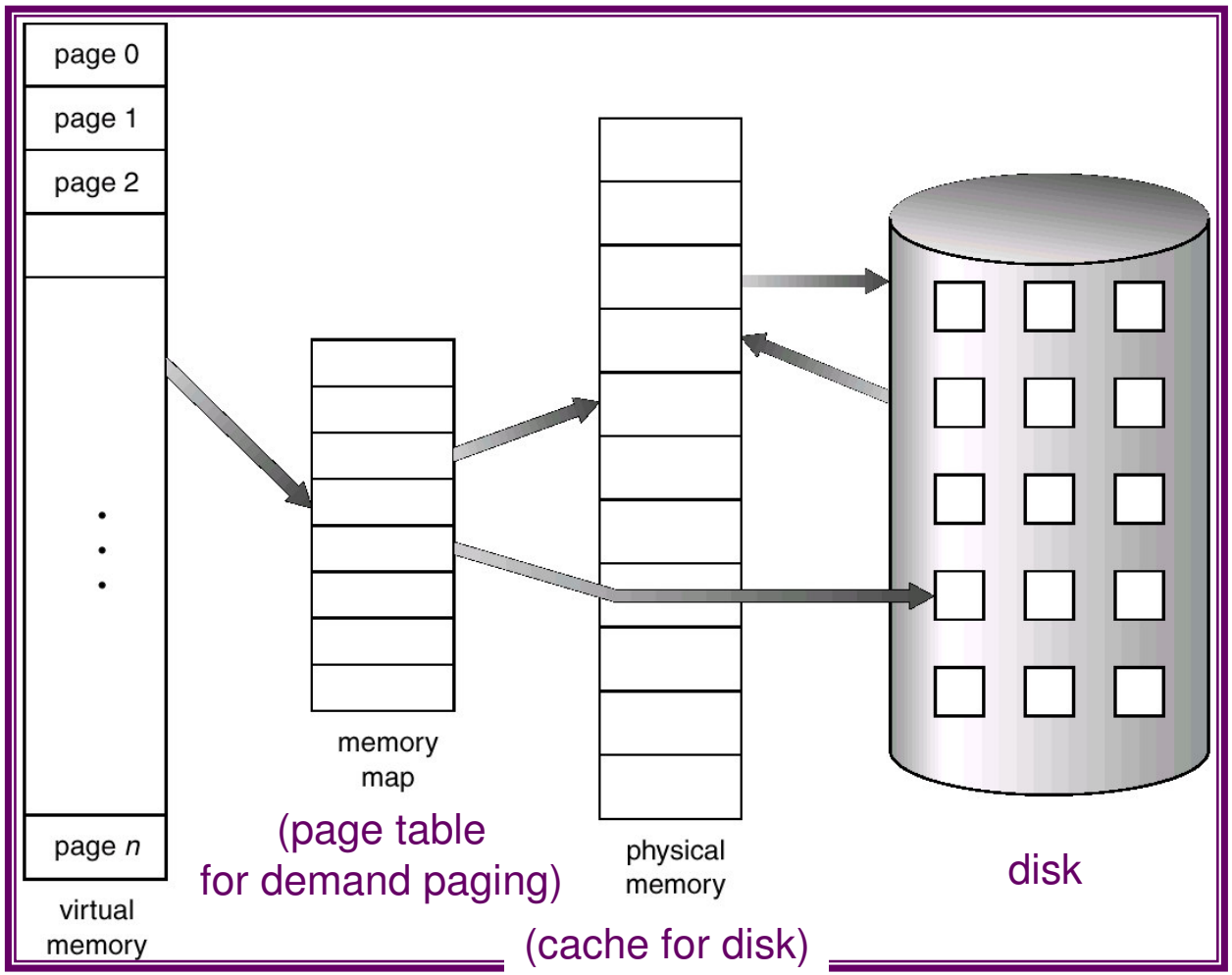
Chapter 10: Virtual Memory

- Background.
- Demand Paging.
- Page Replacement.
- Allocation of Frames.
- Thrashing/Working Set.
- Other Considerations.

Background

- *Virtual memory:* separation of user logical memory from physical memory.
  - Intuition:
    - CPU cache existence transparent to programmer; we think all operations go to RAM, but they don't.
    - Similarly, pretend all operations go to disk. RAM then becomes transparent cache for disk.
    - Practical difference: hardware manages CPU cache, OS manages RAM caching of disk.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - More efficient process creation: give process minimal memory to get started.
- Virtual memory can be implemented via:
  - Demand segmentation (OS/2).
  - Demand paging (this lecture).

Virtual Memory Larger Than Physical Memory



virtual memory size limited by disk capacity only

Demand Paging

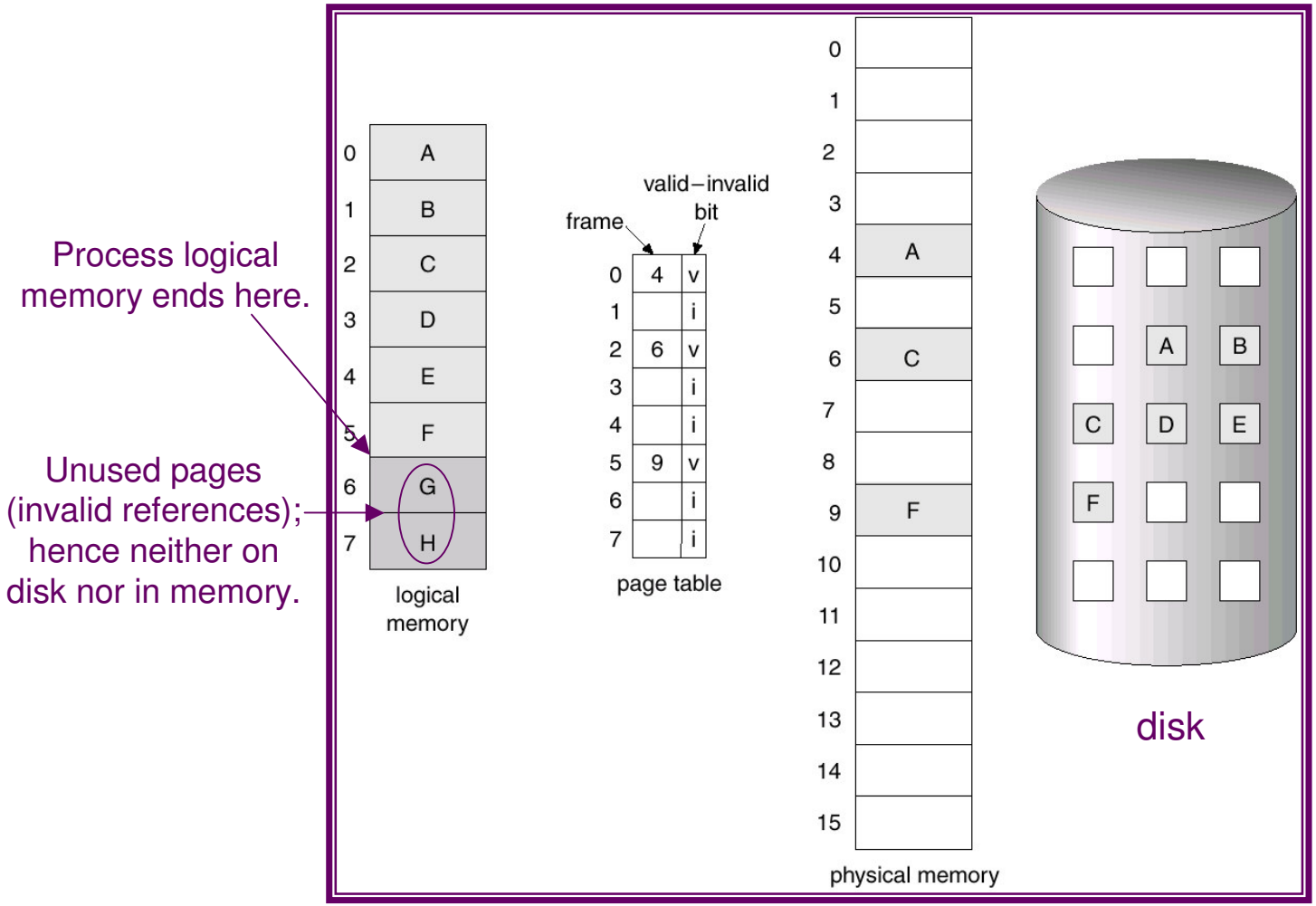
- Bring a page into memory only when it is needed:
  - Like *lazy swapper* only at level of pages not whole processes (*pager*).
  - Less I/O needed: unused pages not moved.
  - Less memory needed: unused pages not in memory.
  - Faster response: process starts as soon as minimal pages are in memory.
  - More users/processes: less memory per process, more processes.
  - When is a page needed? When process refers to it.
- In general, process reference is one of three types:
  - Invalid reference (seg fault) $\Rightarrow$ abort/signal the process.
  - Reference to page in memory $\Rightarrow$ access memory.
  - Reference to page on disk only $\Rightarrow$ bring to memory.

Valid-Invalid Bit

- Page table entry bit states whether page is in memory:
  - 1: in-memory; 0: not-in-memory.
  - Also called valid-invalid bit, but it's logically separate bit than the one marking page table entries that are in use vs. unused ones (invalid references).
  - Can be same hardware bit meaning "trap into OS", and then OS looks up at parallel page table to distinguish used vs. unused from memory vs. disk.
- Initially bit set to 0 on all entries (user program is on disk).
- During address translation, if bit is
  - 1: access memory.
  - 0: *page fault* (bring from disk); **simplified** algorithm:
    - Get available free frame.
    - Copy/swap page into frame.
    - Update page table (set valid bit).
    - Update free frame list.
    - Restart instruction.

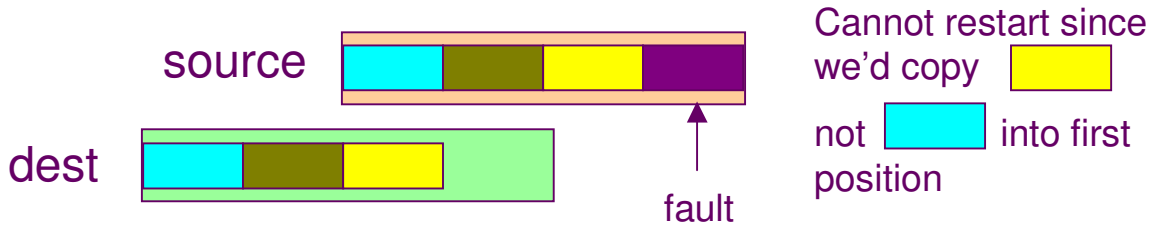
Page Table With Pages Are Not in Main Memory

- Page starts on disk.
- *Copied* (not moved) to memory, only if needed.



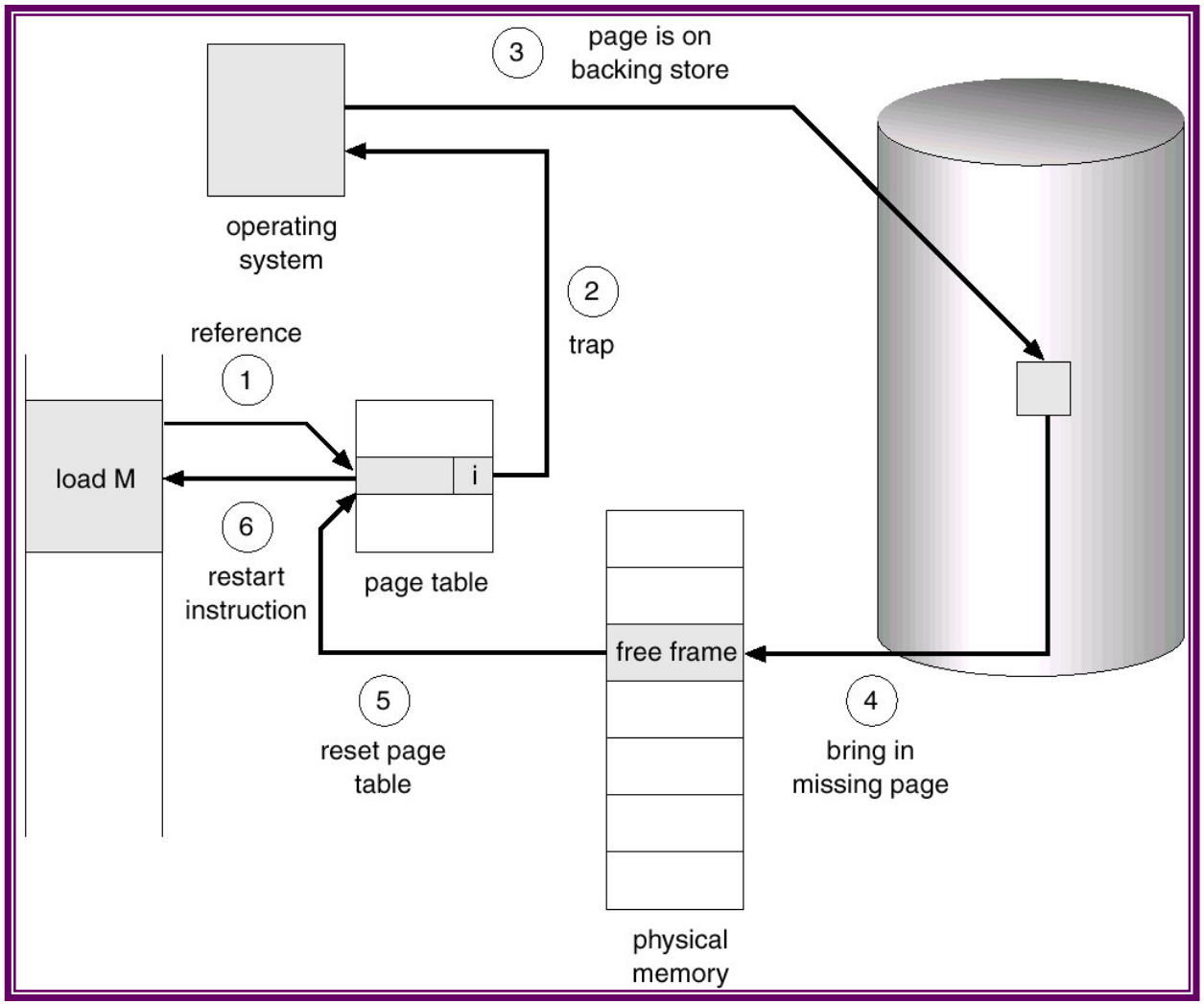
Instruction Restart

- Restart sometimes not easy:
  - ADD A, B, C: just repeat until both operands can be read and result can be stored.
  - Block move: single instruction copies lots of data that span page boundary; blocks overlap.



  - Auto increment/decrement location: MOV (R2)+,-(R3).
    - Like `*(R2++)=*(--R3)` in C.
    - Should not repeat `R2++` and `--R3` if `*R3` fails.
- Conclusion: hardware architecture should be helpful (RISC).

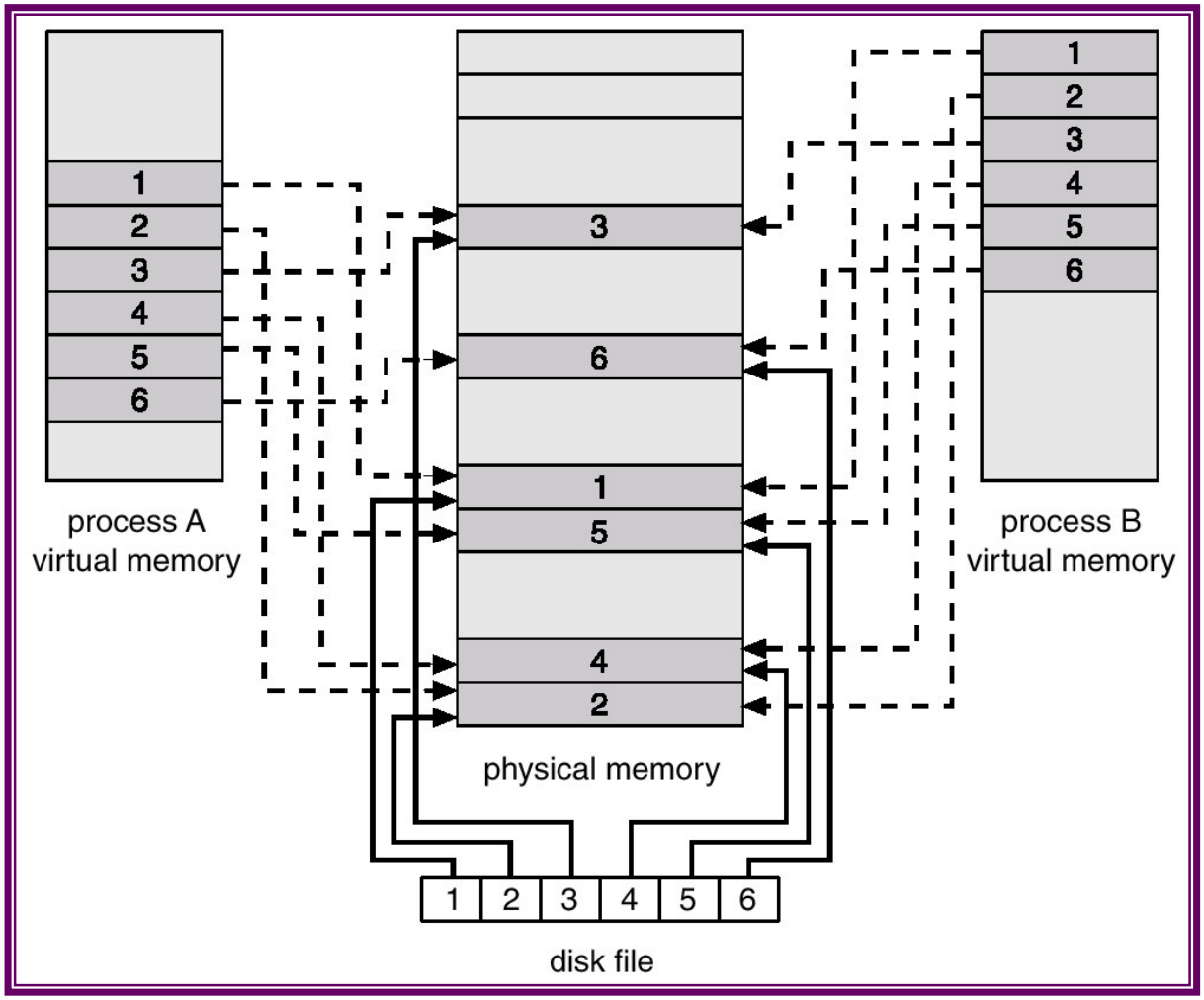
Steps in Handling a Page Fault



More Benefits

- Copy-on-Write (COW):
  - Allows both parent and child processes to initially *share* the same pages in memory.
  - Child process starts much faster (fast `fork()`).
  - If either process modifies a shared page, only then is the page copied.
  - Benefit of general paging, not just Virtual Memory.
- Memory-mapped file I/O:
  - File is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
  - Simpler file access by routing file I/O through memory rather than `read()`, `write()` system calls. Many OSs transparently replace those system calls with memory-mapped I/O.
  - No system call overhead for each byte read/written.
  - Processes that use the same file share the pages.

Memory Mapped Files



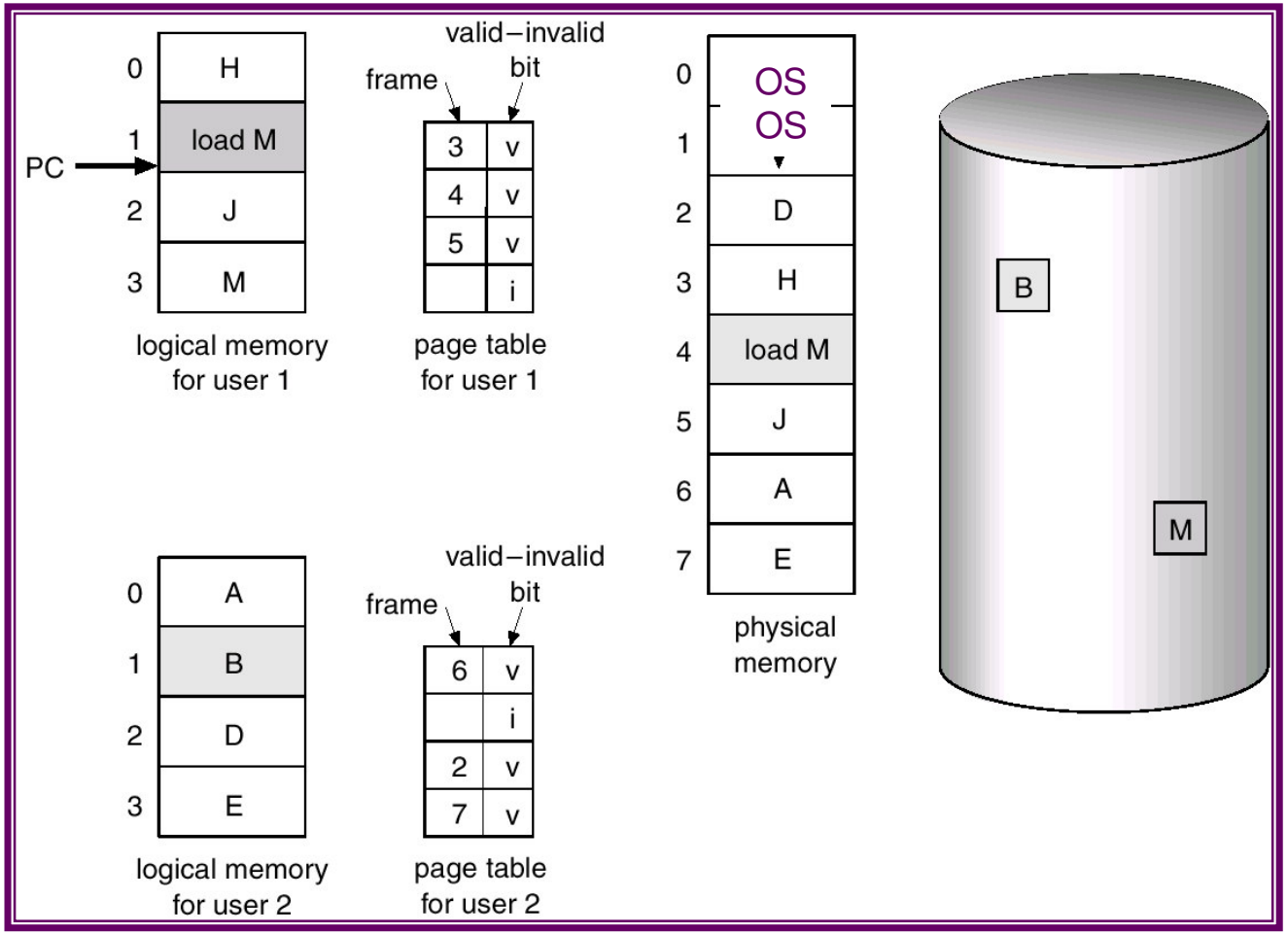
No Free Frame?

- Page replacement: find some *victim* page in memory, but not actively in use, and swap it out.
  - Need copy back to disk only if dirty, i.e. changed since last swap-in.
  - Want algorithm which will minimize number of page faults.
- Page Fault Rate *p* (*p*=0 no page faults; *p*=1 always fault).
- Effective Access Time (EAT) =
  (1 – *p*) x memory access + *p* x (overhead
  + [possibly swap page out]
  + swap page in)

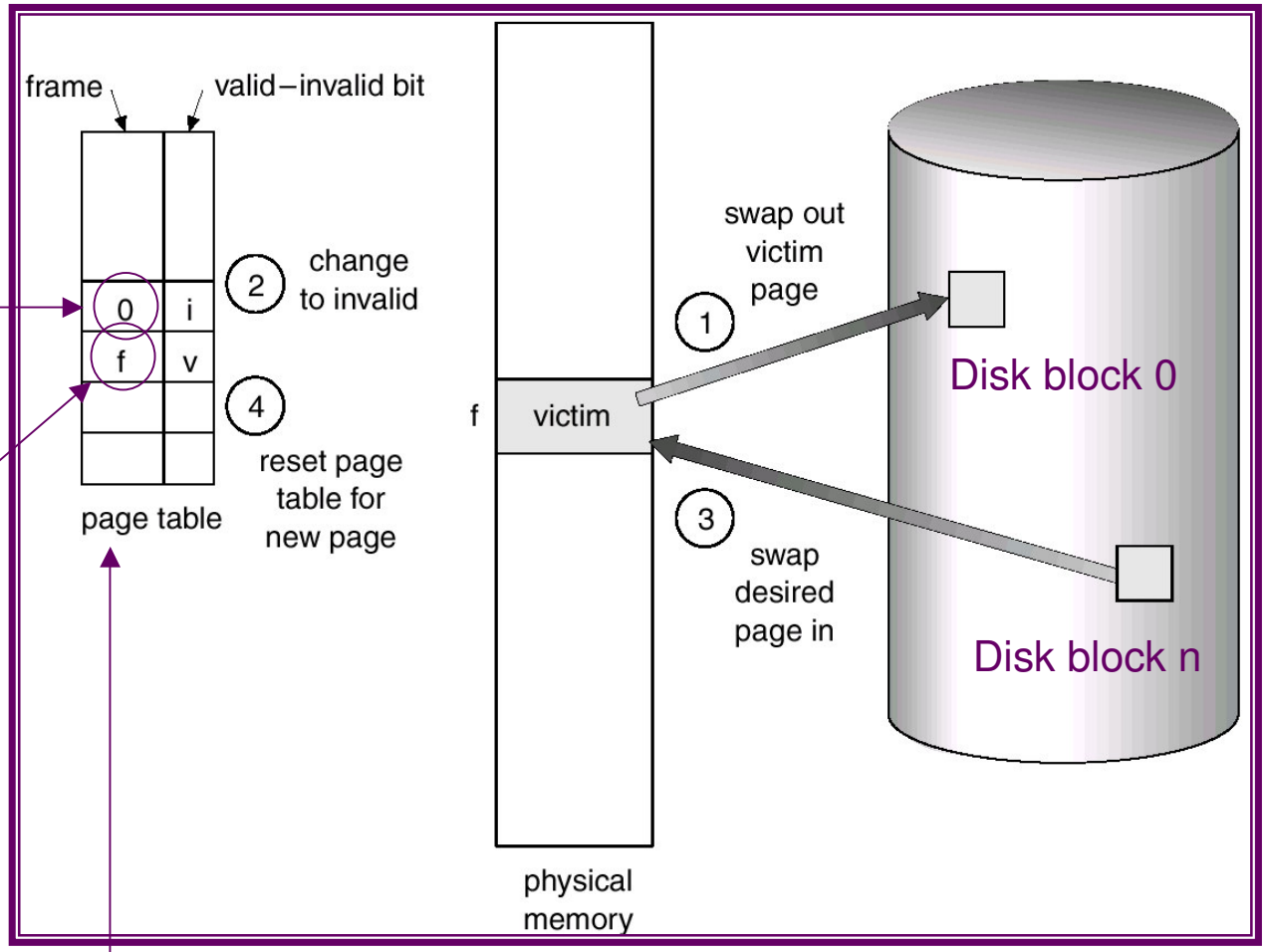
  Includes all memory, TLB accesses
- Assume:
  - Memory access time = 1 ms.
  - Swap Page Time = 10 sec = 10,000 ms.
  - 50% of the time the page dirty, hence expected swap out time is 5,000 ms.
- Then EAT ~ (1 – *p*) x 1 + *p* (15000) = 1 + 15000 *p* ms.
- Need very low fault rate *p*.

Need For Page Replacement

User/Process 1 just executed instruction which needs page M. But there is no free frame available.



Page Replacement



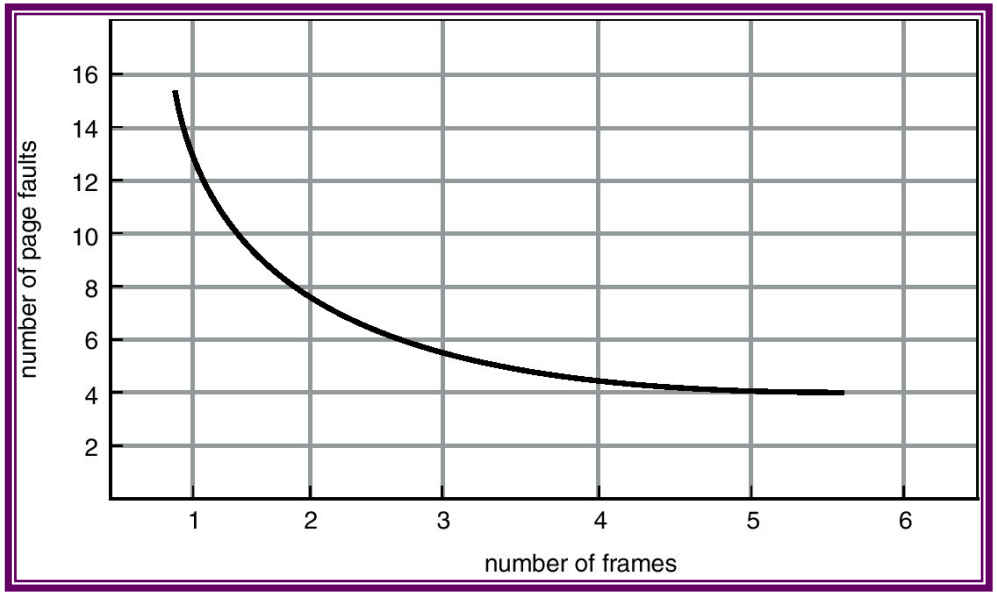
Used to contain f, now it has disk block 0.

Used to contain disk block n, now it has f.

Also common to have separate columns for frame, disk block: if a page needs to be swapped out, no disk allocation overhead. Just reuse the block it came from.

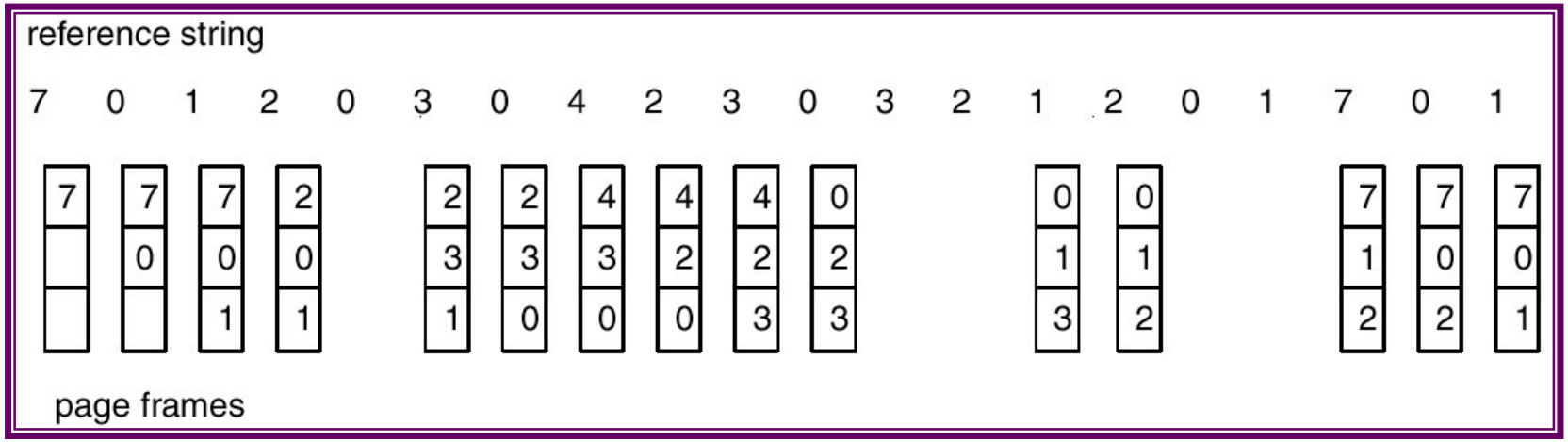
Page Replacement Algorithms

- Want lowest page-fault rate:
  - Good algorithm.
  - More frames.



- Deterministic modeling: evaluate algorithm by running it on a particular sequence of memory references and computing the number of page faults on that string.

First-In-First-Out (FIFO)



FIFO (Cont.)

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

3 frames

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

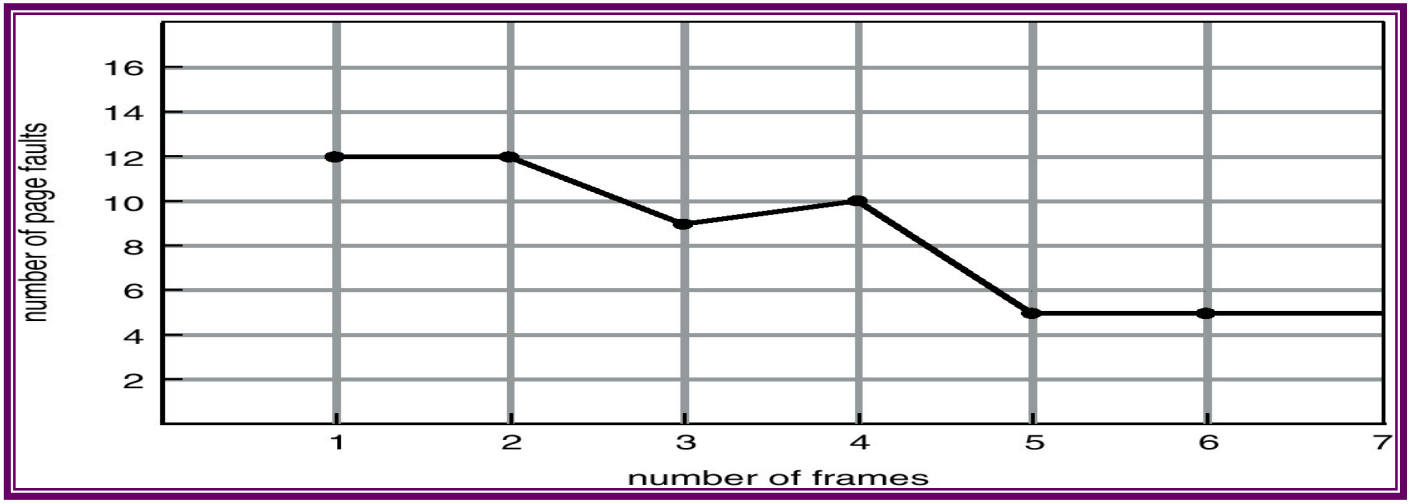
9 page faults

4 frames

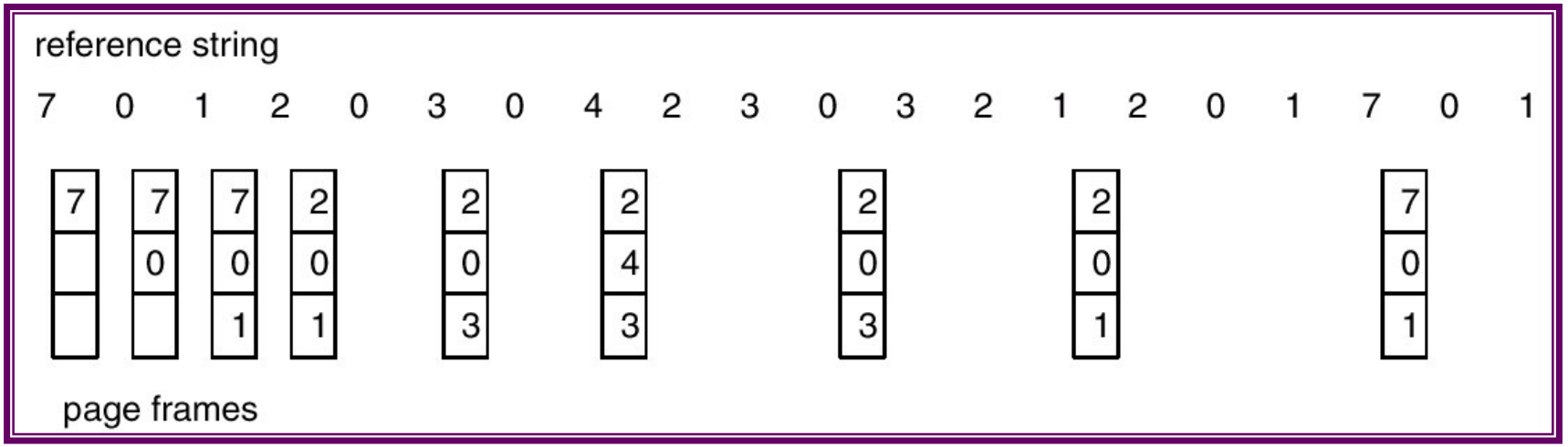
1	1				5			4			
2	2					1			5		
3	3						2				
4	4							3			

10 page faults

- Belady's Anomaly: more frames ⇒ more page faults.

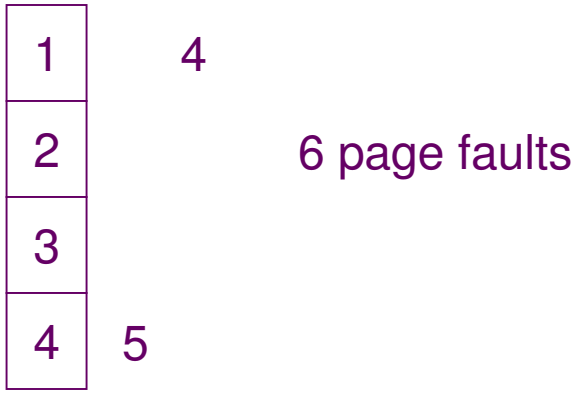


Optimal



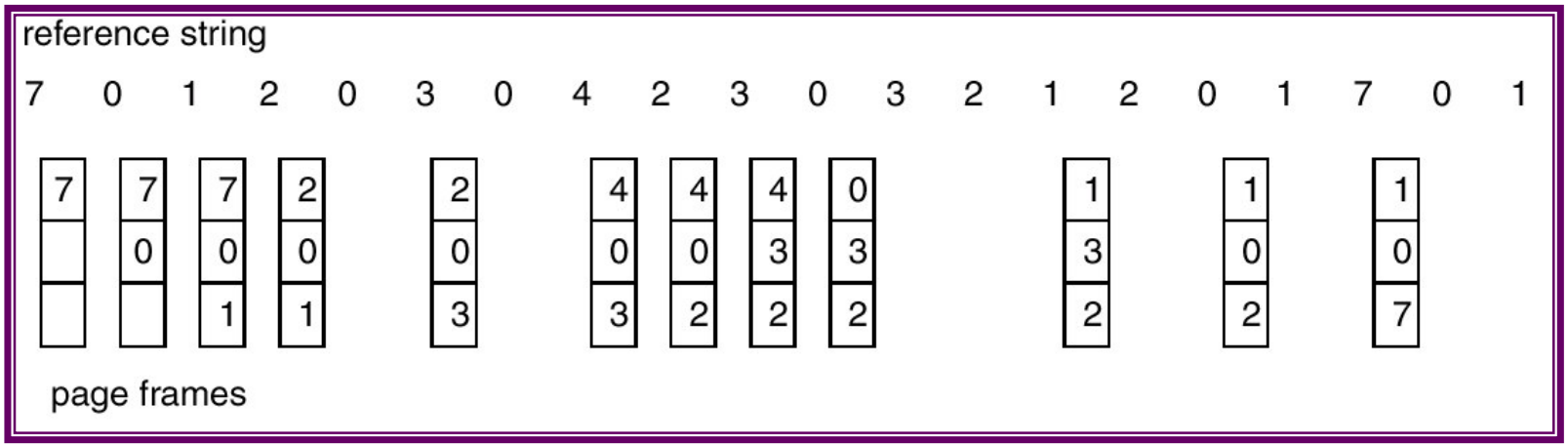
Optimal (Cont.)

- Replace page that will not be used for the longest time.
- 4 frames example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.



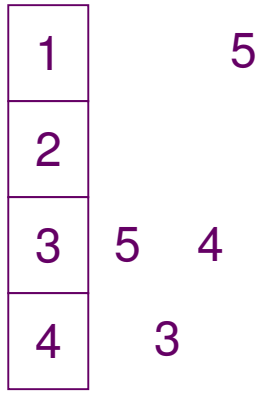
- How do you know which page *will* not be used? You don't. Algorithm is useful as yardstick (to evaluate others).

Least Recently Used (LRU)



LRU (Cont.)

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

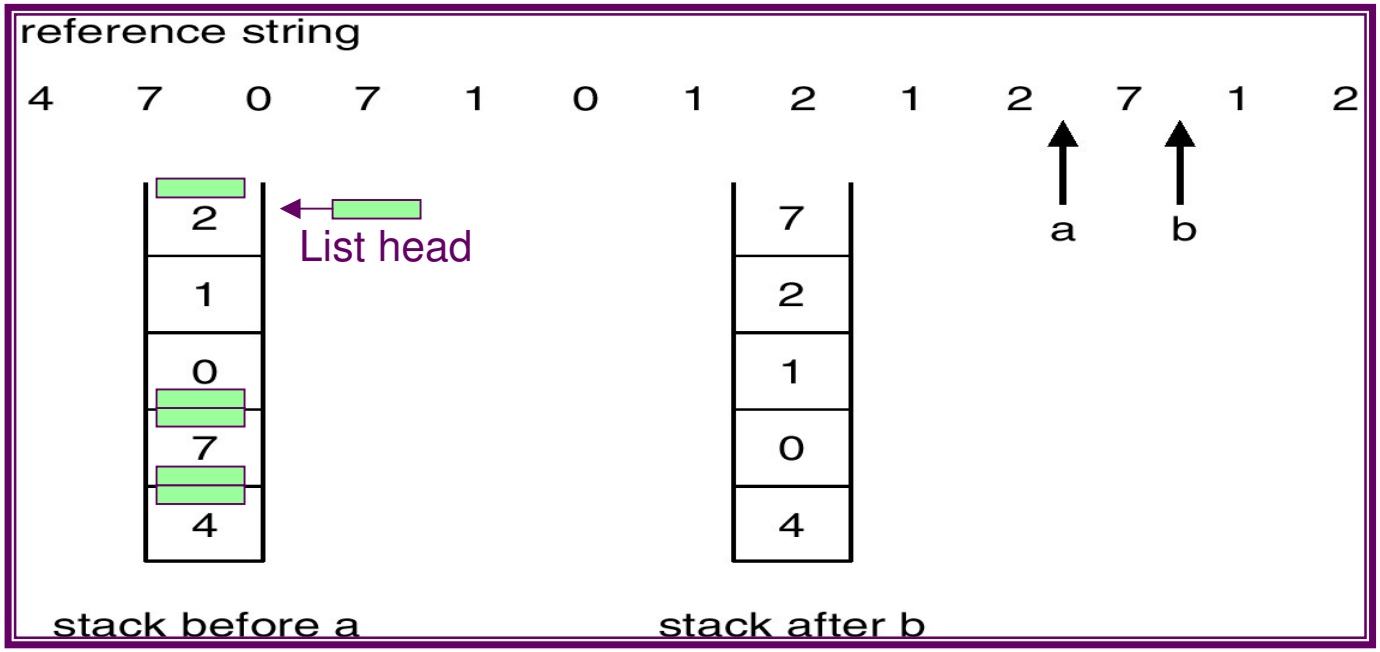


8 page faults

- Counter implementation:
  - Every page table entry has a counter field.
  - When page is referenced (through this entry), copy the clock (or global counter) into the counter.
  - When looking for victim, choose entry with earliest counter. Search needed.

LRU (Cont.)

- Stack implementation:
  - Keep a stack of page table entries in a double linked list.
  - Page referenced:
    - Move entry to the top.
    - Requires 6 pointers to be changed: my 2, the 2 that used to point to me, the 2 that now point to me.
  - No search for replacement.

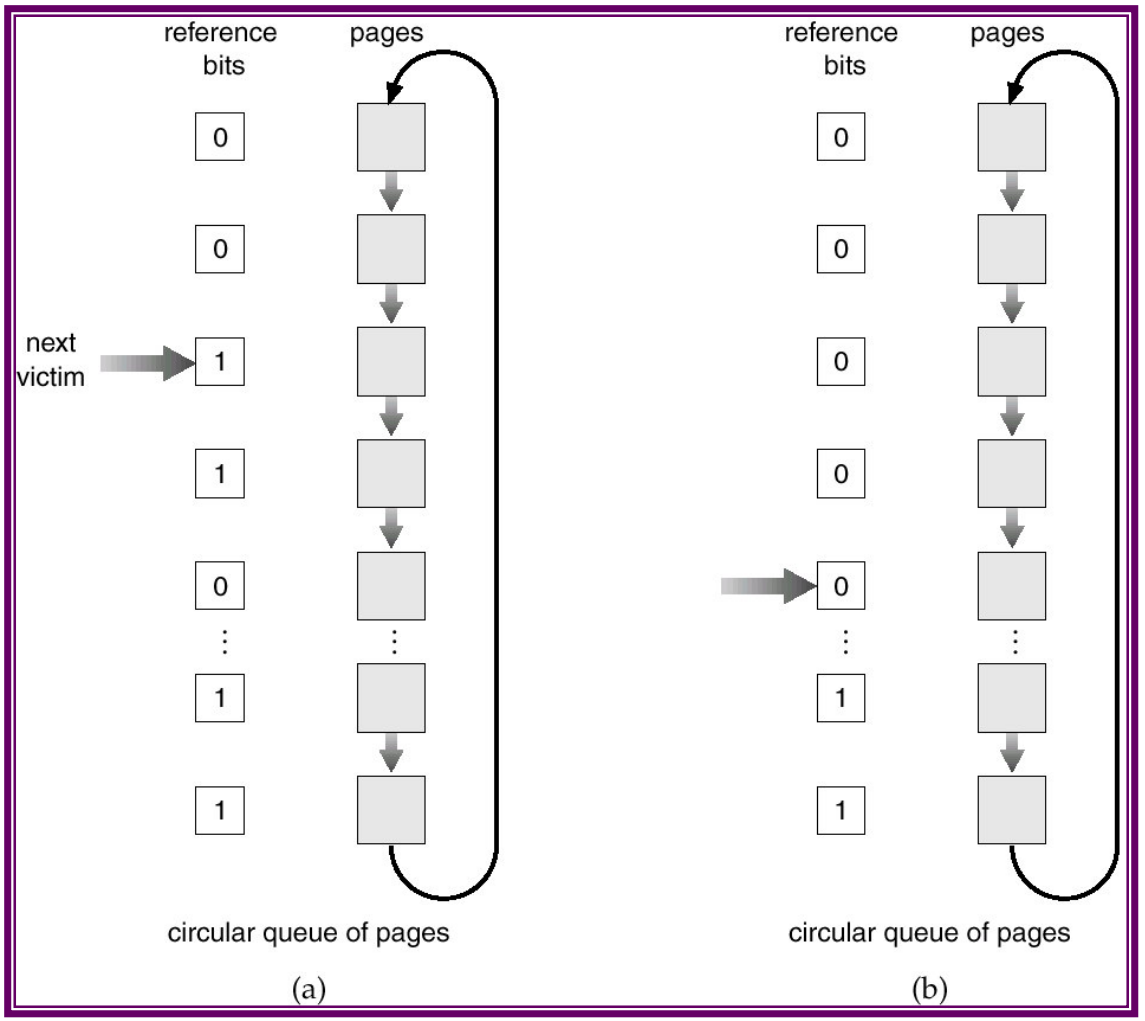


LRU Approximation

- Reference bit:
  - With each page associate a bit, initially 0.
  - When page is referenced, set bit to 1.
  - Replace page with bit=0, if one exists.
  - If all pages bit=1, which was LRU? Who sets bit to 0?
    - Keep history bits (shift left at regular timer interrupt).
- Second chance algorithm:
  - Key part of assignment 2.
  - Need reference bit.
  - A.k.a. clock replacement (but no clock involved).
  - Organize pages in *circular* queue; queue pointer points to next victim (like FIFO)...
  - ... but if pointed page has reference bit=1. then:
    - Set reference bit to 0.
    - Leave page in memory.
    - Try next page, subject to same rules.

Second-Chance (Clock)

Tip: the search for a victim always starts right after the last victim chosen. It does *not* always start at the beginning of the queue.



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. How do we allocate frames?
  - ☞ Assignment 2.

Allocation of Frames

- Each process needs **minimum** number of pages in memory (i.e. frames allocated to process).
- Example: IBM 370 – 6 pages to handle a single SS MOVE instruction:
  - Instruction is 6 bytes long, might span 2 pages.
  - Source data can span 2 pages (even just 2 adjacent bytes but on different pages).
  - Destination data can span 2 pages.
- Two major allocation schemes.
  - Fixed allocation (static and local):
    - Equal allocation.
    - Proportional allocation (see next slide).
  - Priority allocation (dynamic and global):
    - If high priority process generates a page fault, select victim from a lower priority process.

Proportional Allocation

- Bigger process $\Rightarrow$ more pages, or
- Higher priority process $\Rightarrow$ more pages.

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

$S = \sum_{processes} s_i$

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

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

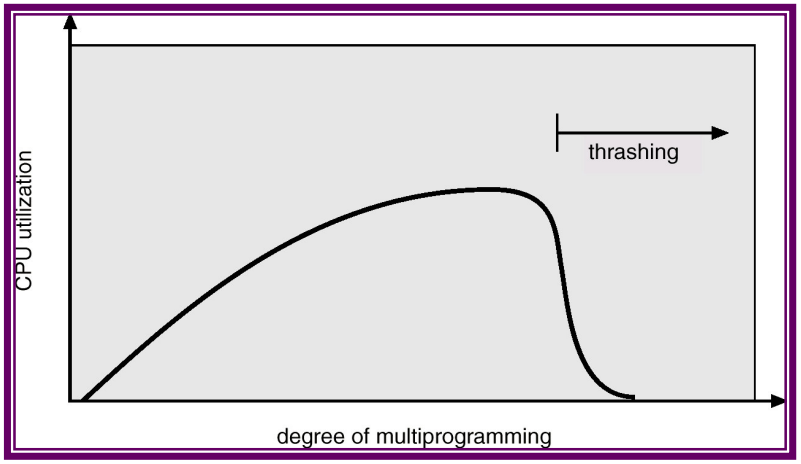
$S = 137$

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

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

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - Low CPU utilization (lots of time spent doing I/O).
  - Operating system thinks that it needs to increase the degree of multiprogramming hence...
    - It creates more processes...
    - so processes get less memory...
    - so page-fault rate increases!

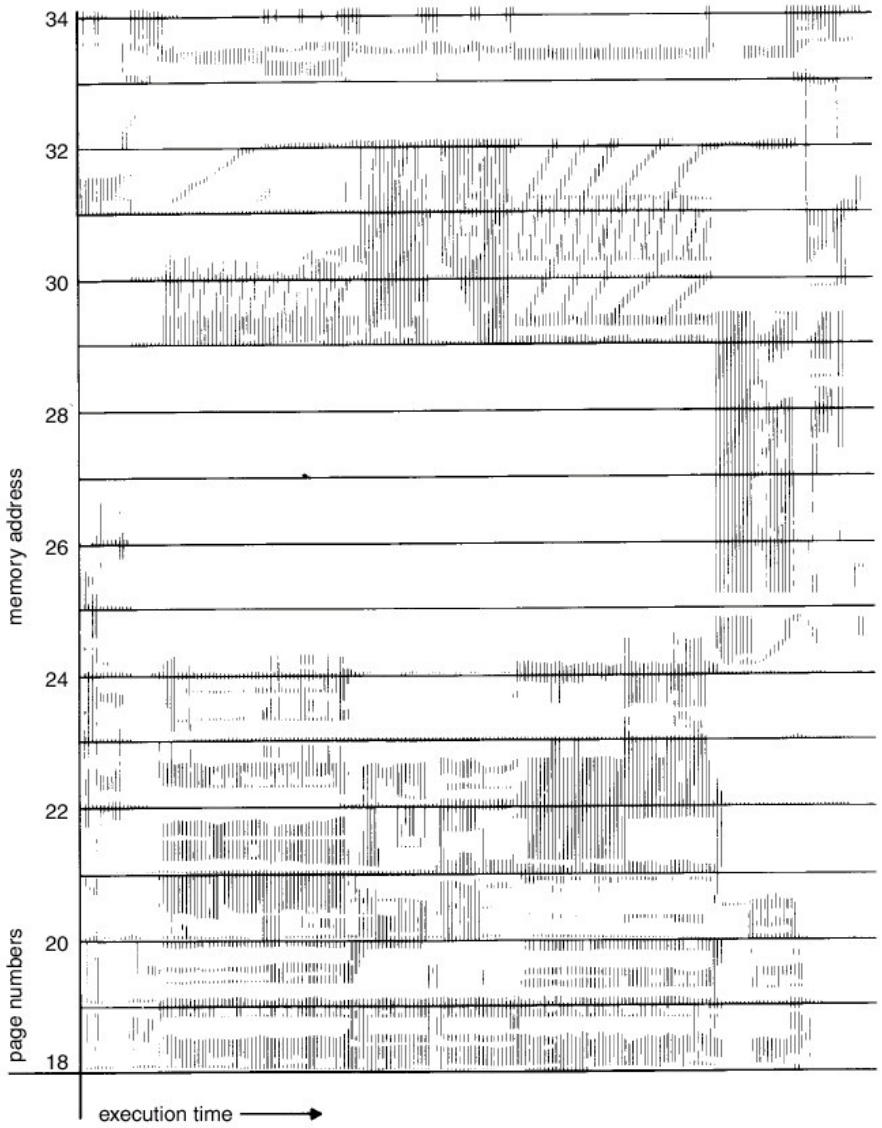


  - Medium-term scheduler should swap out processes.
- *Thrashing*: system more busy swapping pages in and out than letting processes use CPU.
- How many pages are “enough”? Depends on program. See later slide with integer array example.

Locality In A Memory-Reference Pattern

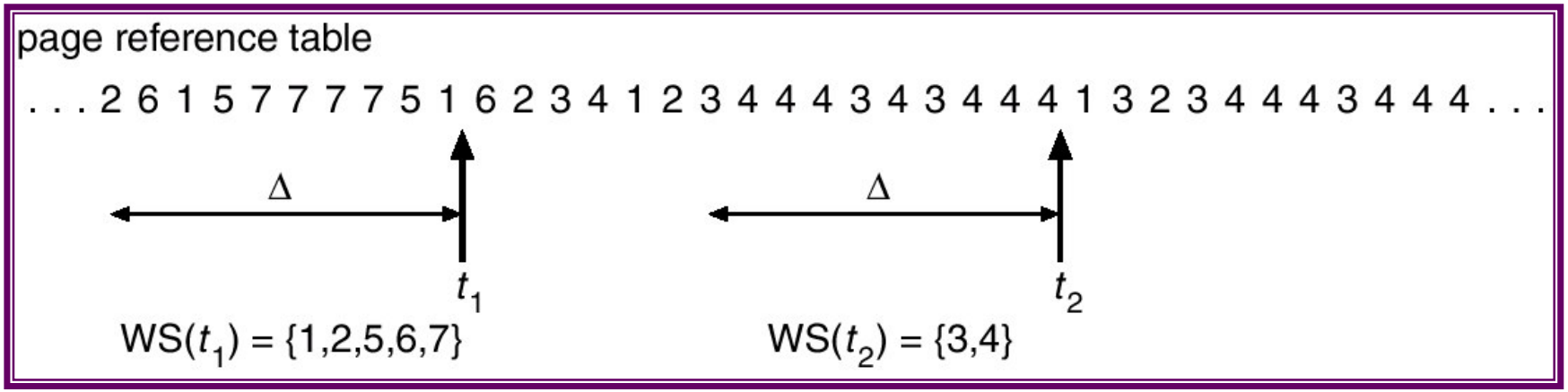
- Paging works due to *locality* model:
  - Locality is small subset of pages in active use.
  - Process migrates from one locality to another.
- Why does thrashing occur?
  Σ locality size > memory size.

processes



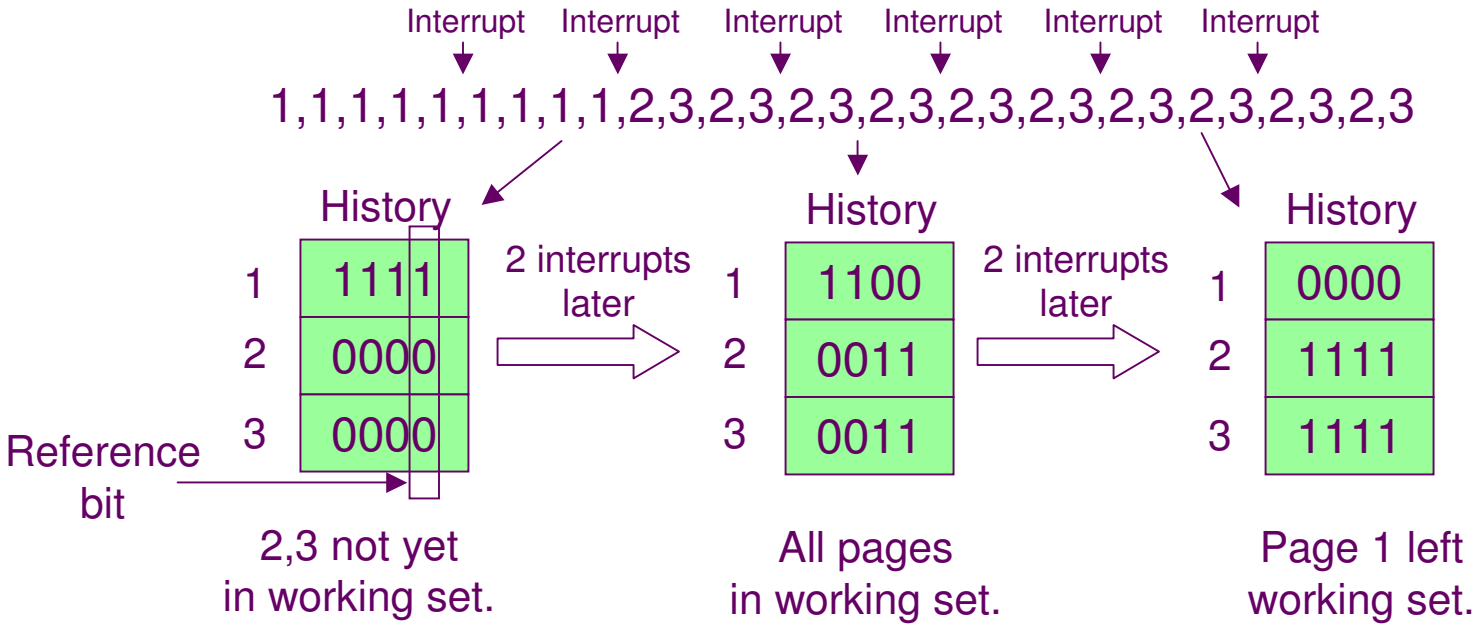
Working-Set Model

- $\Delta \equiv$ working-set window: a fixed number of page references.
- WSS_i (working set size of process P_i) = total number of pages referenced in the most recent Δ.
  - If Δ too small, it will not encompass entire locality.
  - If Δ too large, it will encompass several localities.
  - If $\Delta = \infty$, it will encompass entire program.
- $D = \sum_{\text{processes}} WSS_i \equiv$ total demand frames.
- If $D >$ total memory frames $\Rightarrow$ thrashing, hence suspend one or more processes or increase allocation of single thrasher.

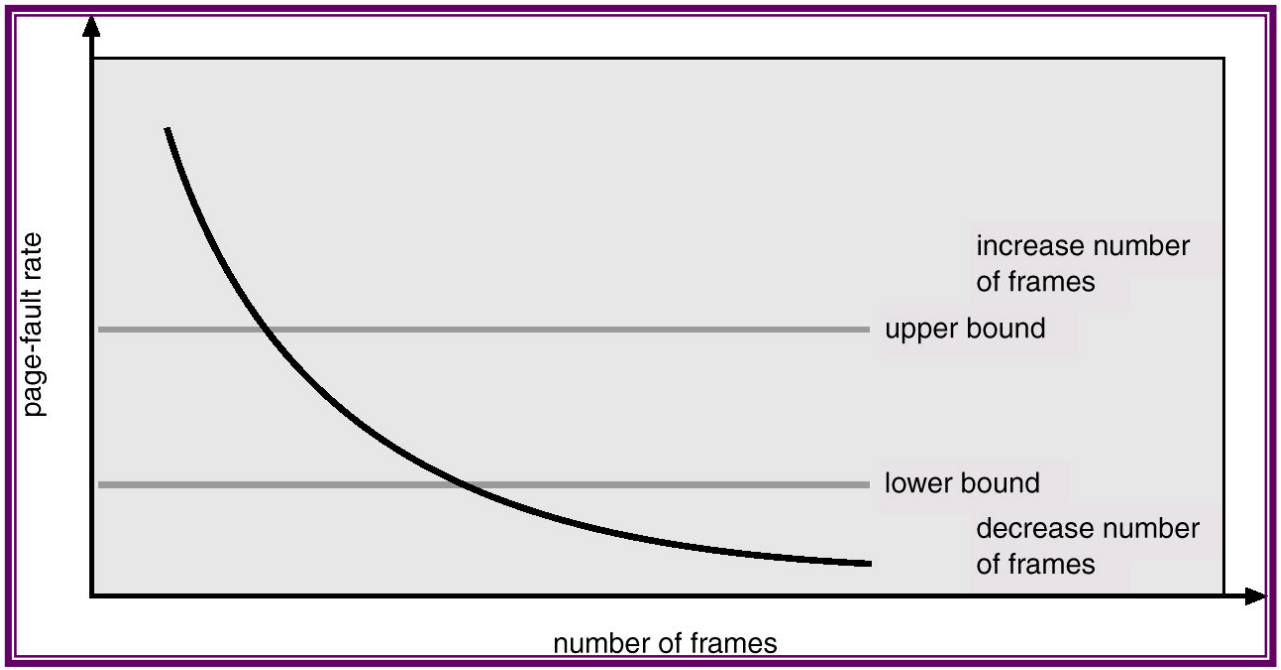


Keeping Track of the Working Set

- Working set related to LRU: pages in working set are those in on-going/recent use.
- Approximate with reference bit and history bits. If a page has non-zero history counter, then page is in working set.



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.

Other Considerations

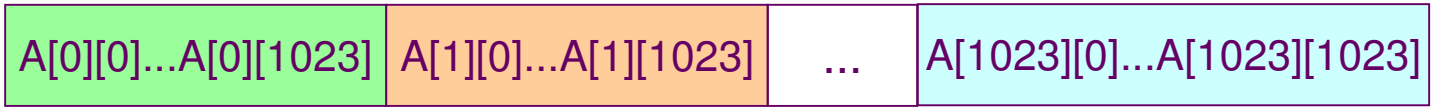
- Prepaging: when process suspends then resumes, it must reload its working set. Rather than do many separate I/Os for each page as it faults, bring in full working set before restarting.
- Page size selection:
  - Internal fragmentation: want smaller pages.
  - Table size: want large pages.
  - I/O overhead (transfer time): large pages reduce *number* of pages loaded, which reduces seek time. But...
  - Locality (working set) wants smaller pages to focus only on memory actually used. Hence less total I/O because we are not wasting memory to store data we don't actually use; less page faults, less data transferred.

Other Considerations (Cont.)

- *TLB Reach:* the amount of memory accessible via TLB.
  - Equal to (TLB Size) X (Page Size).
- Ideally, the working set of each process is stored in the TLB. Otherwise low TLB hit ratio f; EAT increases (see last lecture).
- Increasing reach:
  - Increase TLB size: high cost.
  - Increase page size: different apps are allowed to have different page sizes.

Other Considerations (Cont.)

- Program structure:
  - **int A[][] = new int[1024][1024];**
  - Assume only 1 frame available (for data; ignore code page).
  - If page size 4KB, then each row is stored in one page:



  - Program 1

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

    1024 x 1024 page faults.

  - Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

    1024 page faults.
  - Compiler usually does loop reordering.

Other Considerations (Cont.)

- *I/O Interlock*: pages must sometimes be locked into memory.
  - Same issue as with process relocation (last lecture).
  - Pages containing buffers for pending I/O must not be chosen for eviction by a page replacement algorithm, or...
  - ... do all I/O in OS buffers and then copy (costly).