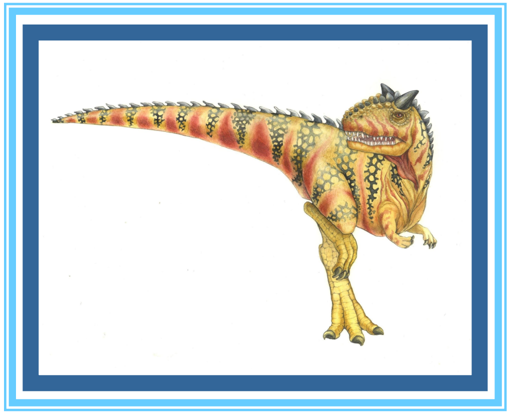
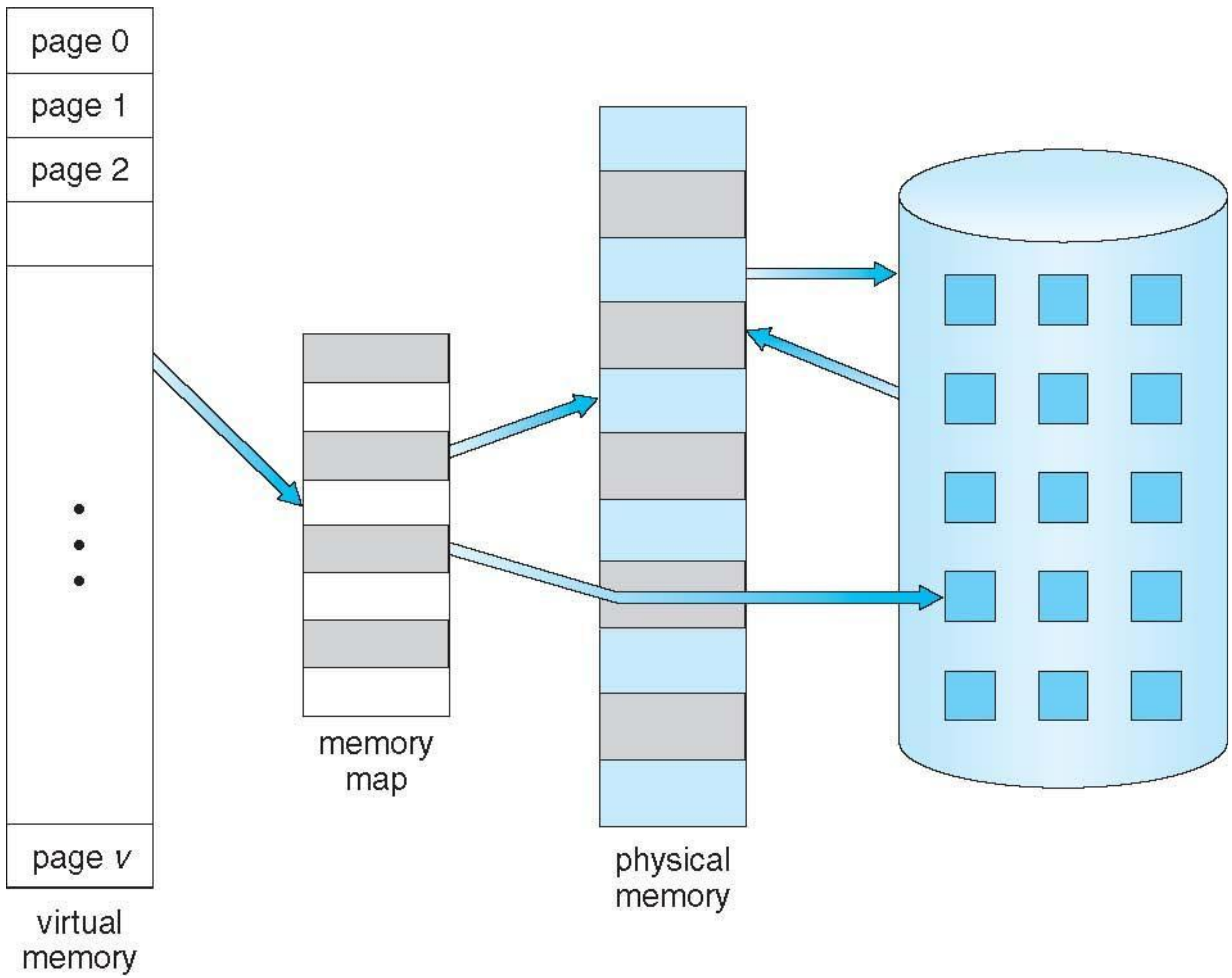
Virtual Memory



Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time

Virtual Memory That is Larger Than Physical Memory



Background (Cont.)

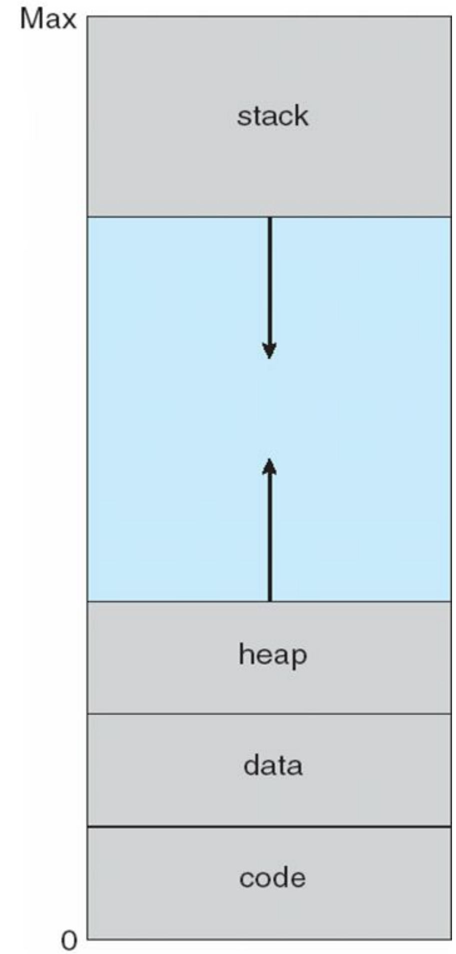
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Background (Cont.)

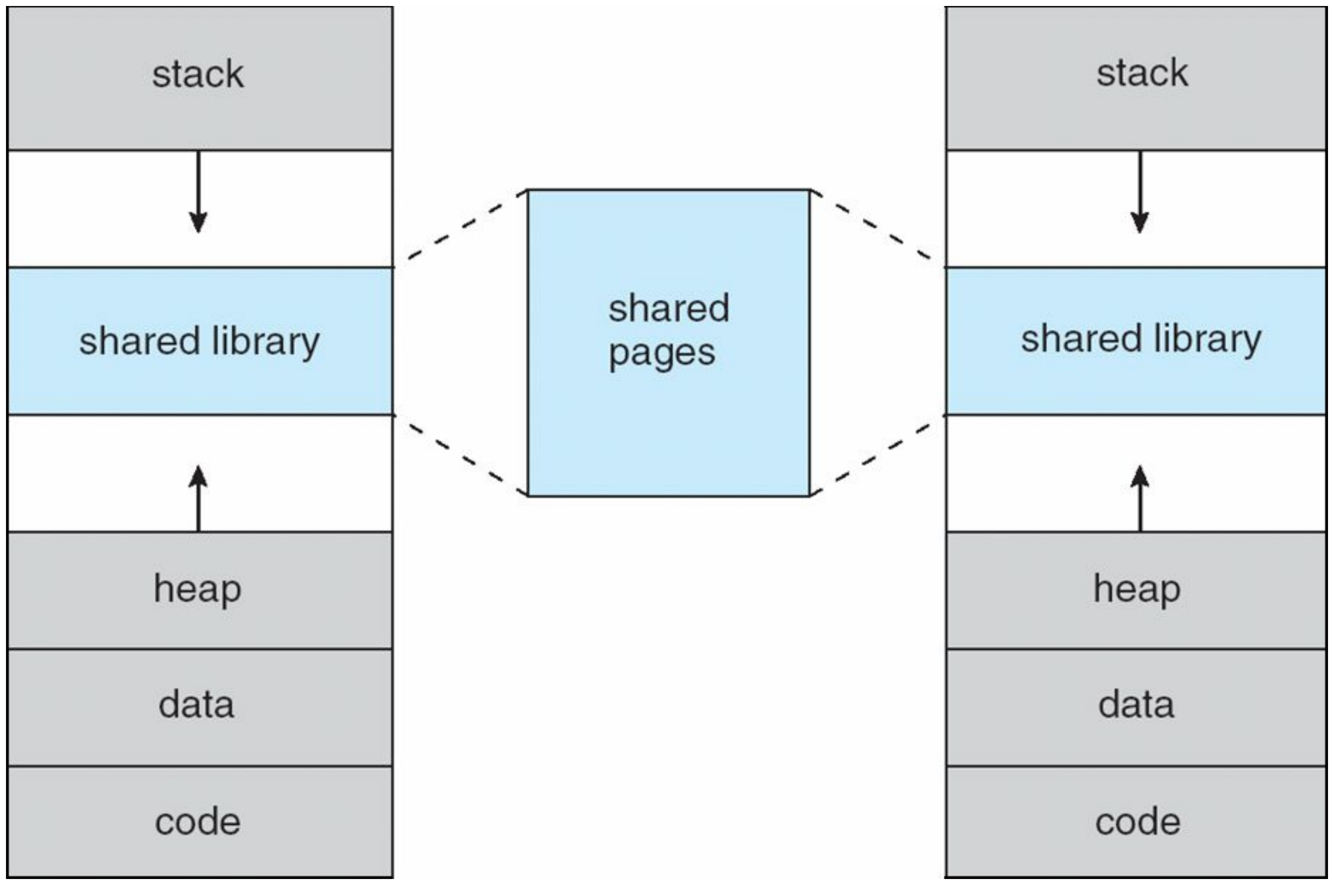
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical

Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory



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

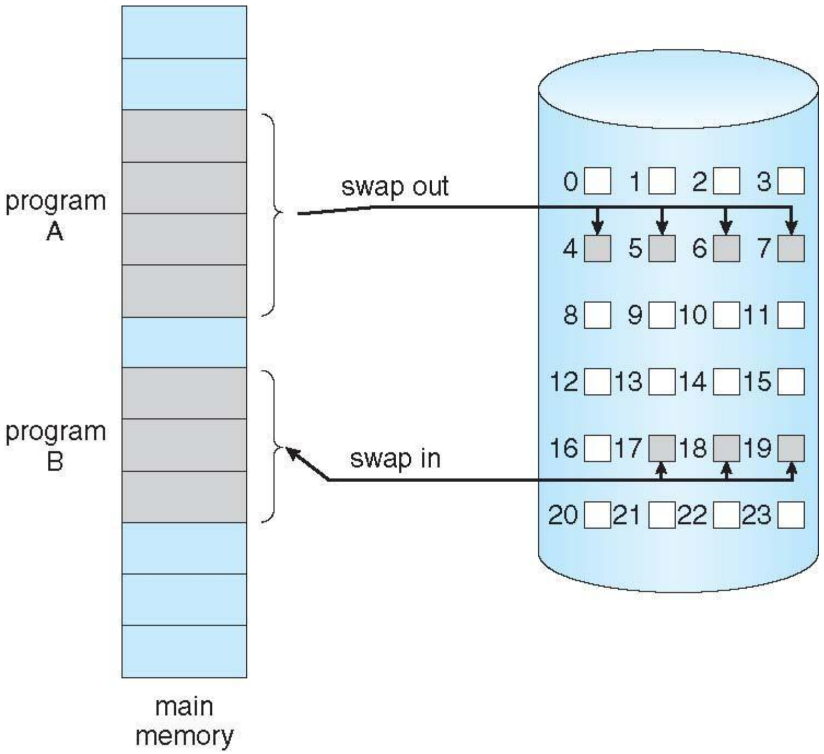
DEMAND PAGING

Demand Paging

- Executable program need to be loaded from disk into memory
  - Could bring entire process into memory at load time
  - Or bring a page into memory only when it is needed [Demand Paging]
    - Less I/O needed, no unnecessary I/O
    - Less memory needed
    - Faster response
    - More users

Demand Paging

- Similar to paging system with swapping
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

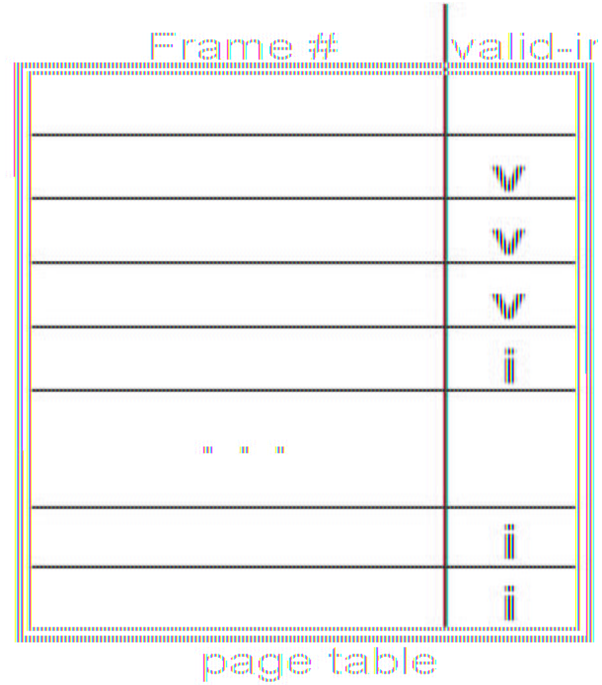


Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- The pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

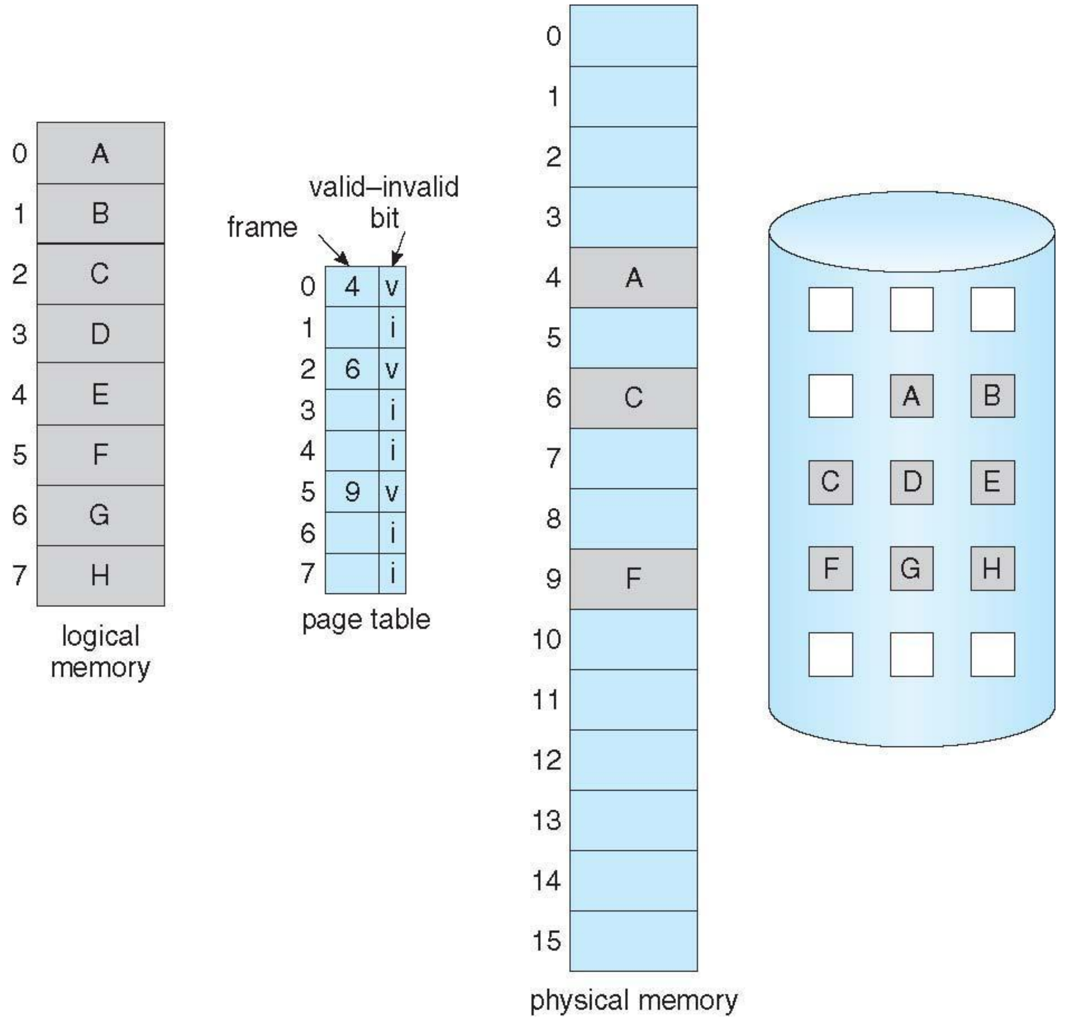
Valid-Invalid Bit

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



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ **page fault**

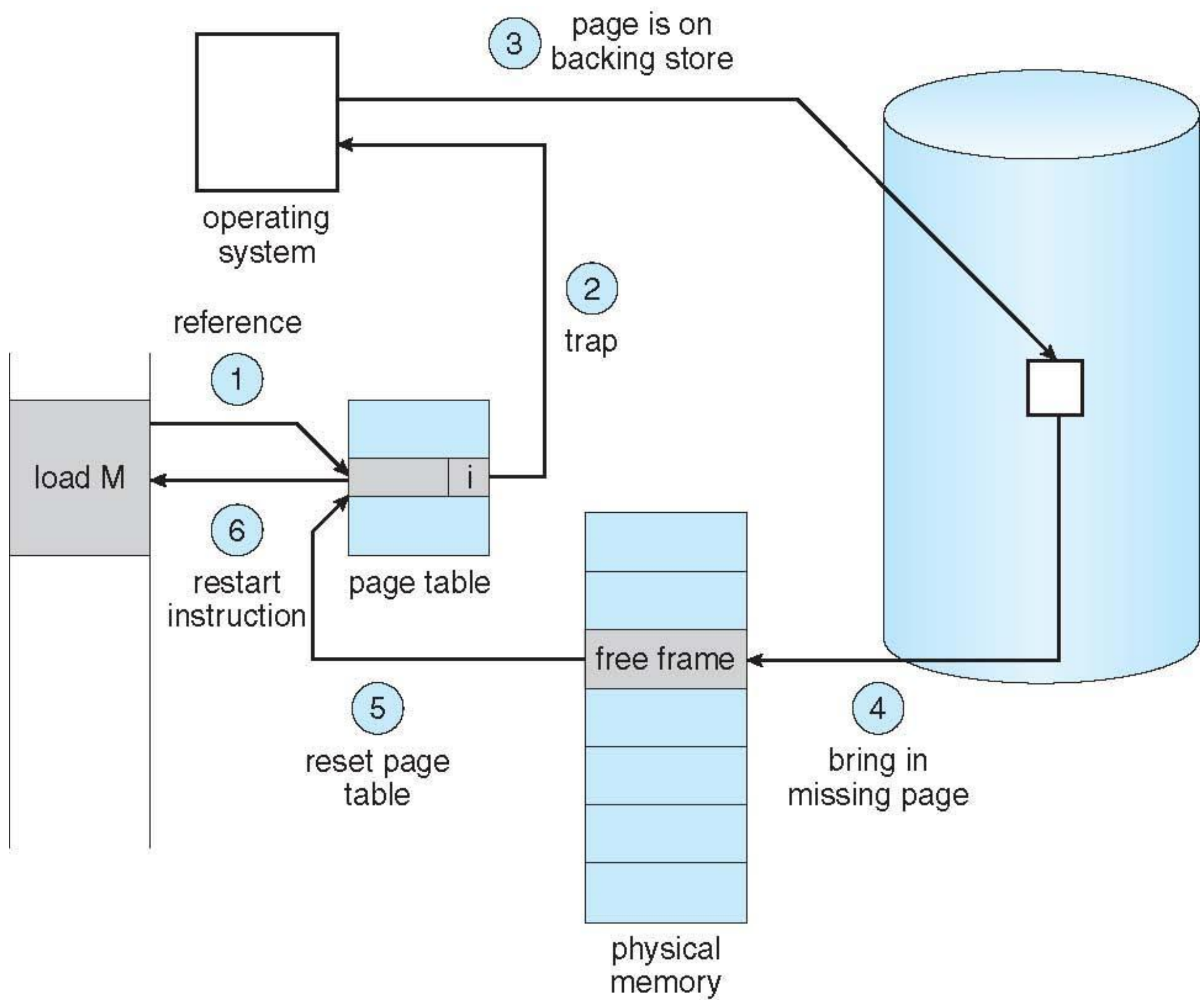
Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**
- The procedure to handle page fault:
  1. Operating system looks at internal table to check whether the reference is valid or invalid
  2. if
     - Invalid reference ⇒ abort
     - Valid, just not in memory ⇒ page it in
  3. Find free frame
  4. Swap page into frame via scheduled disk operation
  5. Reset tables to indicate page now in memory
     Set validation bit = **v**
  6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**

Aspects of Demand Paging

- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Performance of Demand Paging (Cont.)

- Demand paging can significantly affect the performance of a computer system???
- Let's compute Effective Access Time (EAT)

EAT = (1 – *p*) x ma + *p* x page fault time

  - p is probability of a page fault ($0 \le p \le 1$)
    - if $p = 0$ no page faults
    - if $p = 1$, every reference is a fault
  - ma is memory access time (10 to 200 ns)
  - Page fault time??

Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the 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:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame

Performance of Demand Paging

- Stages in Demand Paging (contd….)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show 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

Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time

Demand Paging Example

- Memory access time, ma = 200 ns
- Average page-fault service time = 8 ms = 8,000,000ns
- EAT = (1 – *p*) x ma + *p* x page fault time

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then

  EAT = 8199.8 ns

  = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

Demand Paging Example

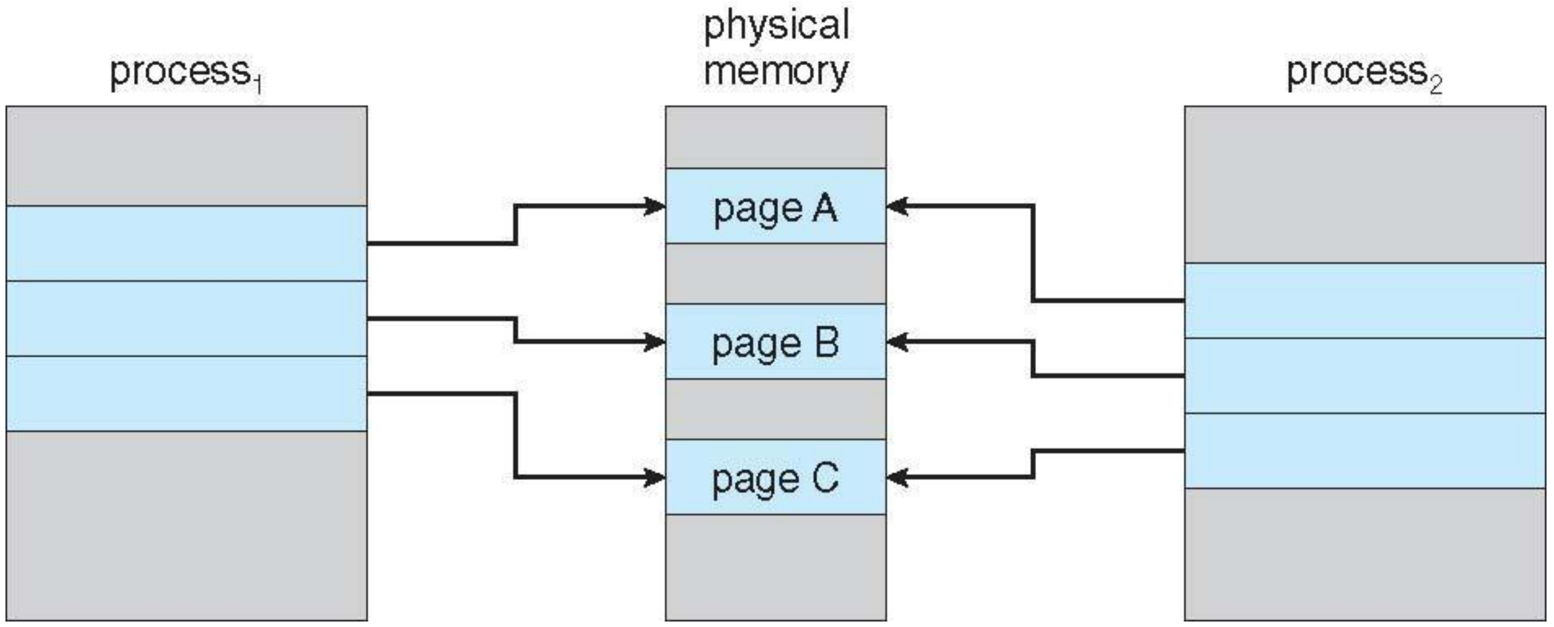
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p

    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Copy-on-Write

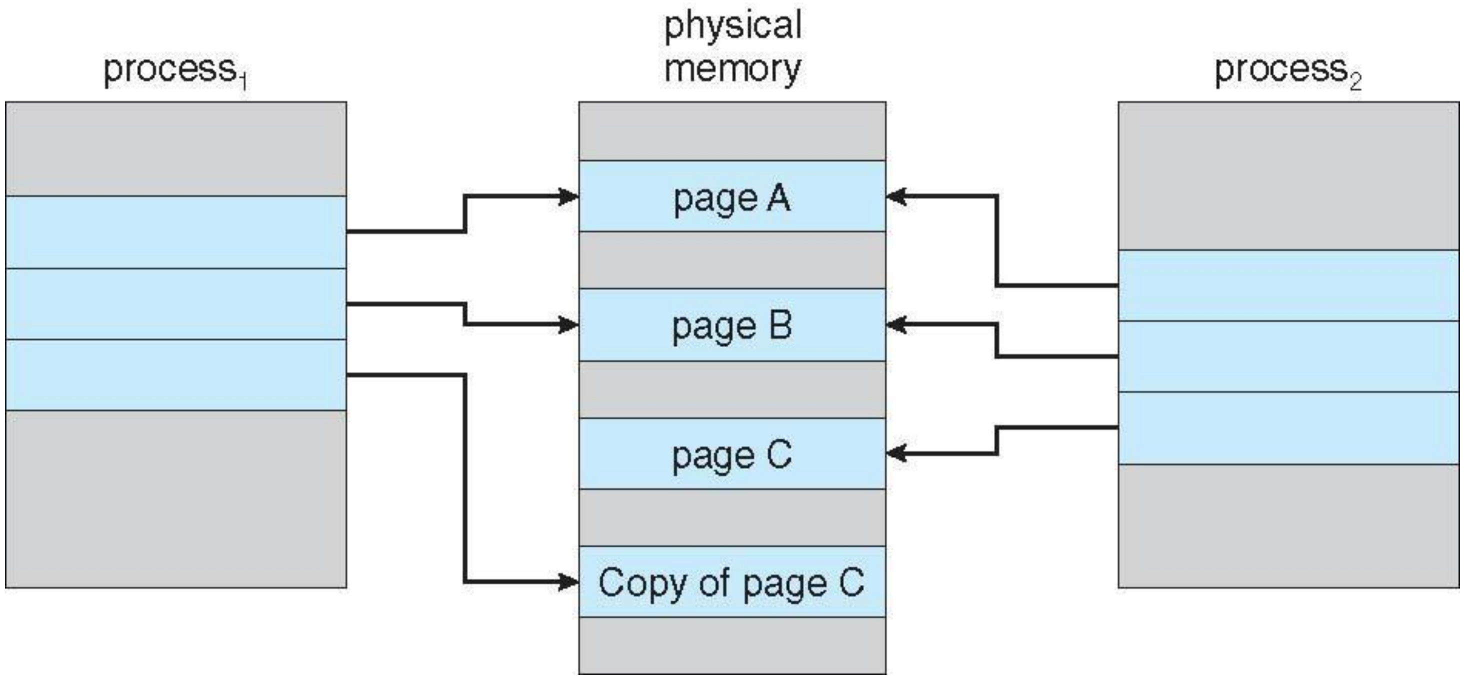
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?

Copy-on-Write



Before Process 1 Modifies Page C

Copy-on-Write

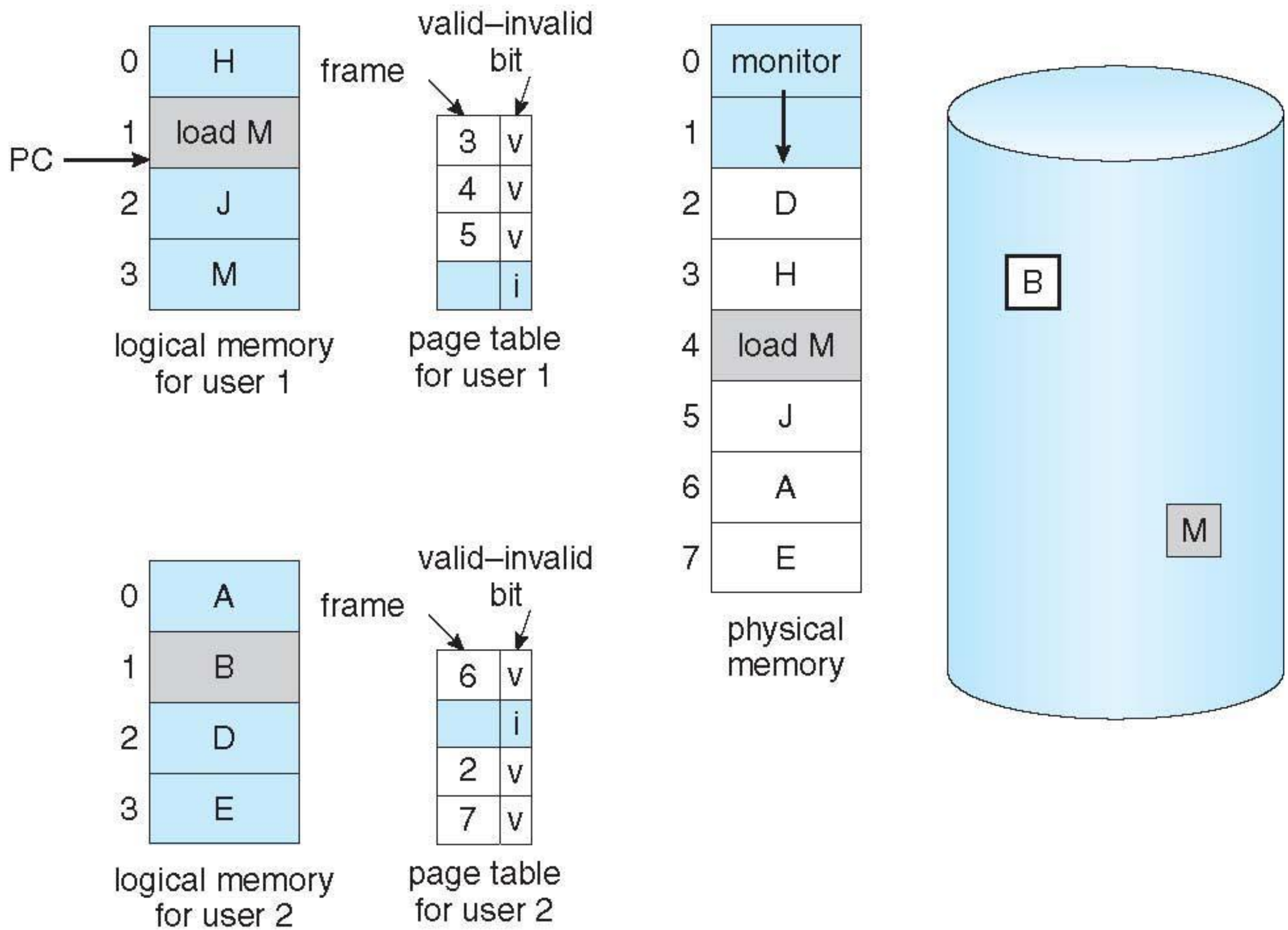


After Process 1 Modifies Page C

What happens if there is **no Free Frame**?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc

Need For Page Replacement



What Happens if There is no Free Frame?

- Terminate? swap out? replace the page?
- **Page replacement** – find some page in memory, but not really in use, page 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

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

1. Find the location of the desired page on disk

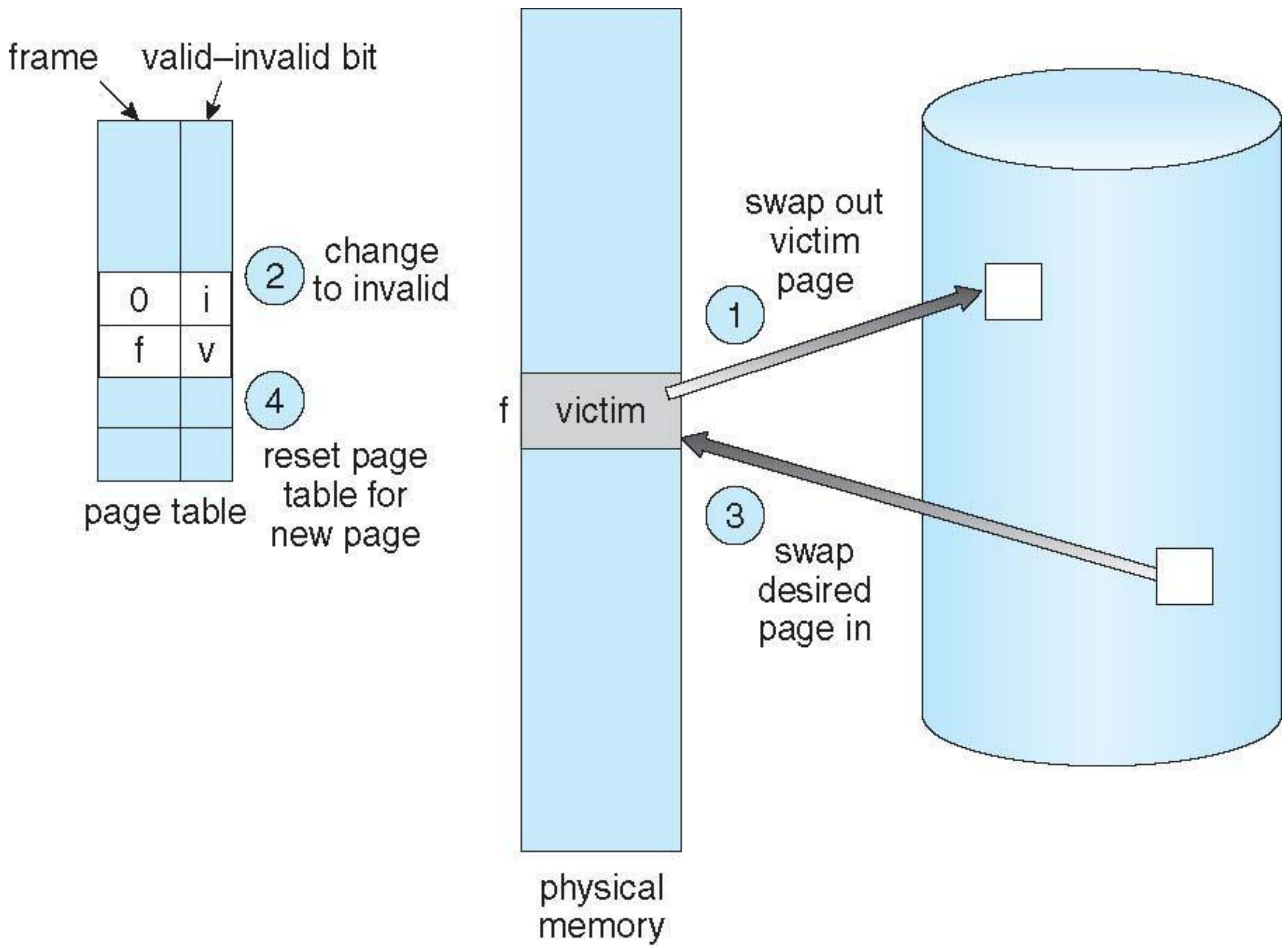
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



Page and Frame Replacement Algorithms

Two problems need to be solved in demand paging

1. **Frame-allocation algorithm** determines
   - How many frames to give each process
2. **Page-replacement algorithm**
   - Which frames to replace

- Want lowest page-fault rate on both first access and re-access

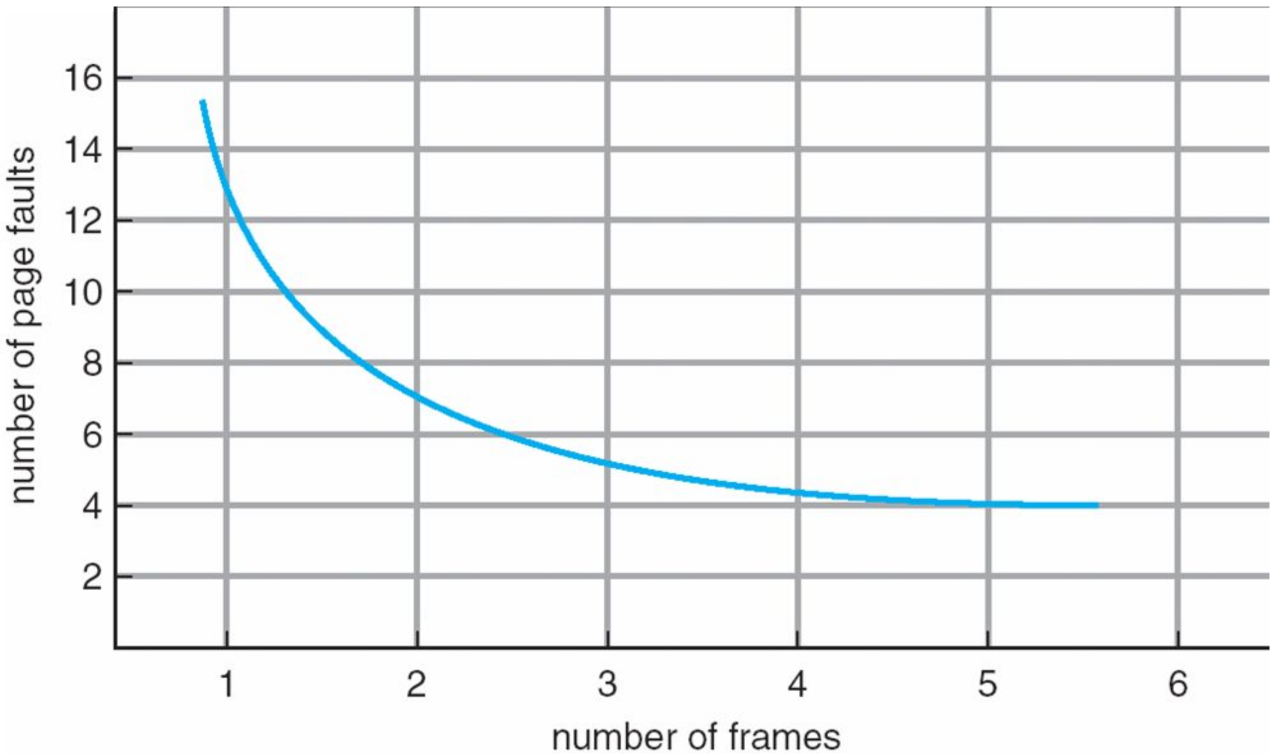
PAGE REPLACEMENT ALGORITHMS

Page and Frame Replacement Algorithms

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

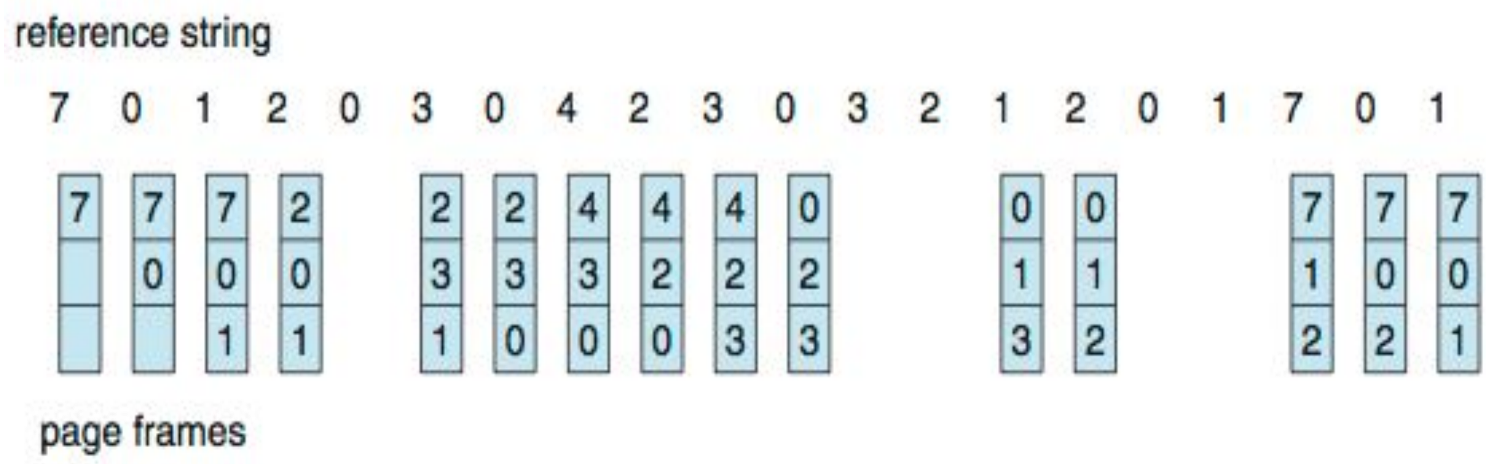
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

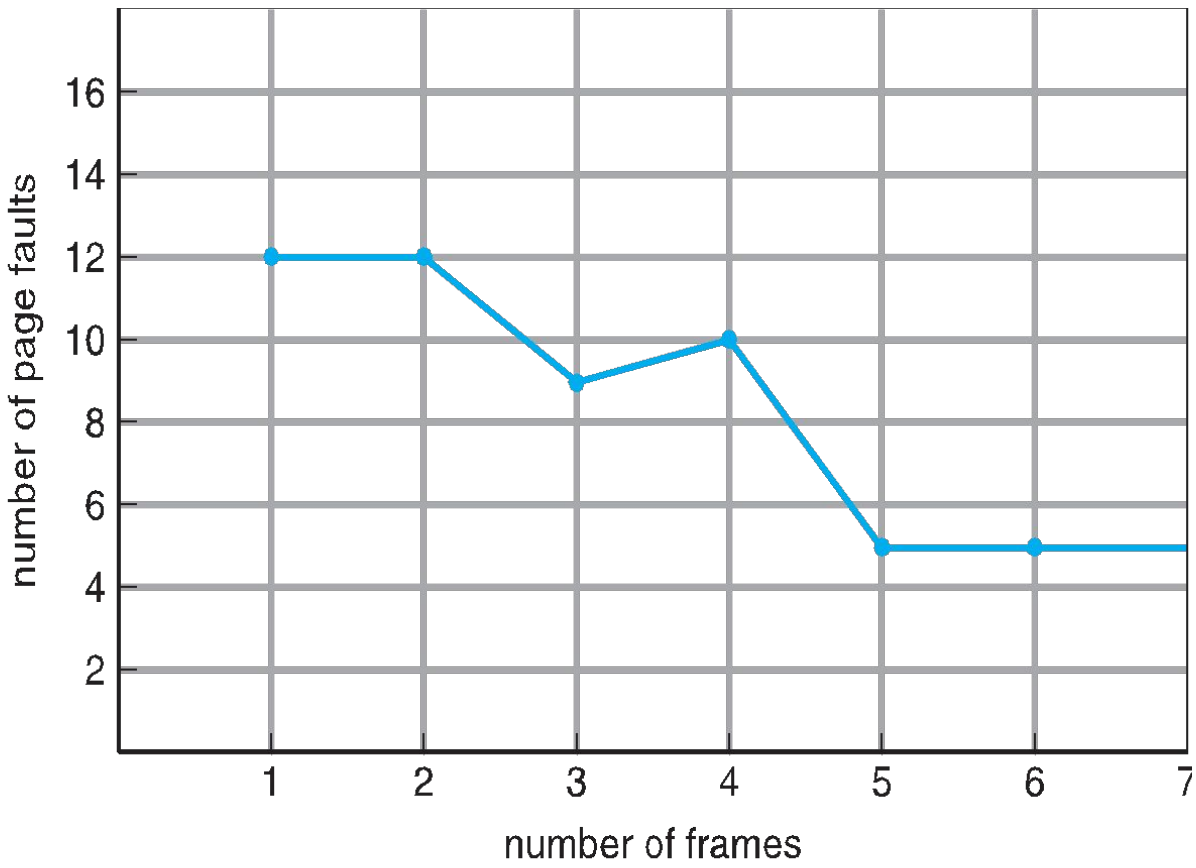


15 page faults

First-In-First-Out (FIFO) Algorithm

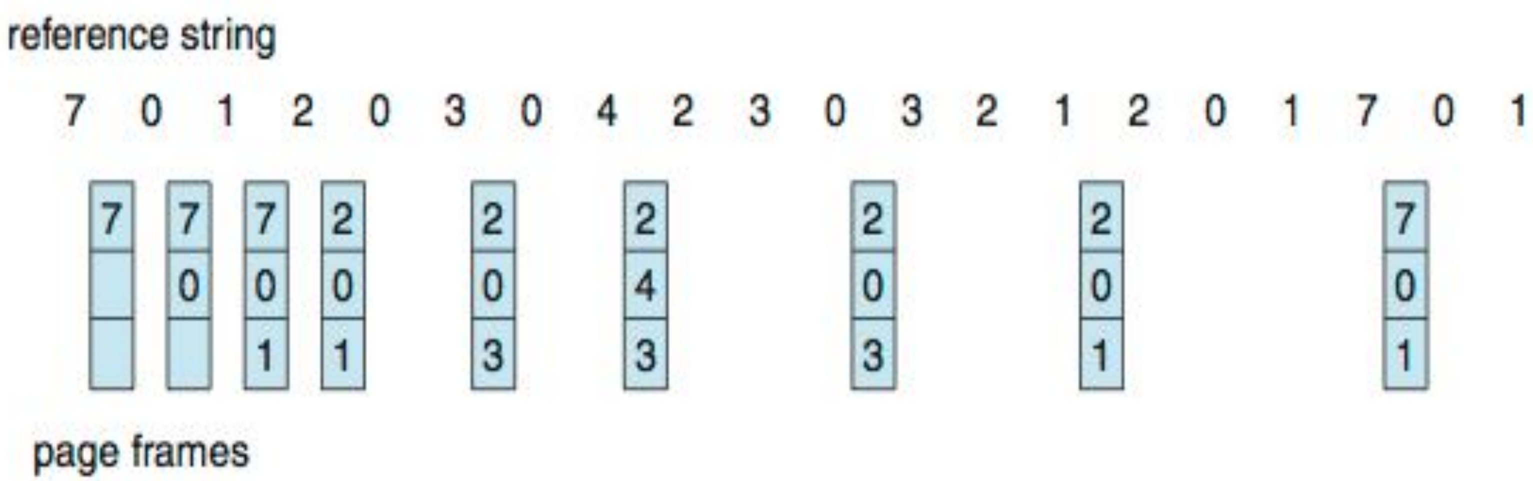
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
- How to track ages of pages?
  - Just use a FIFO queue
- Adding more frames can cause more page faults!
  - **Belady's Anomaly**

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time



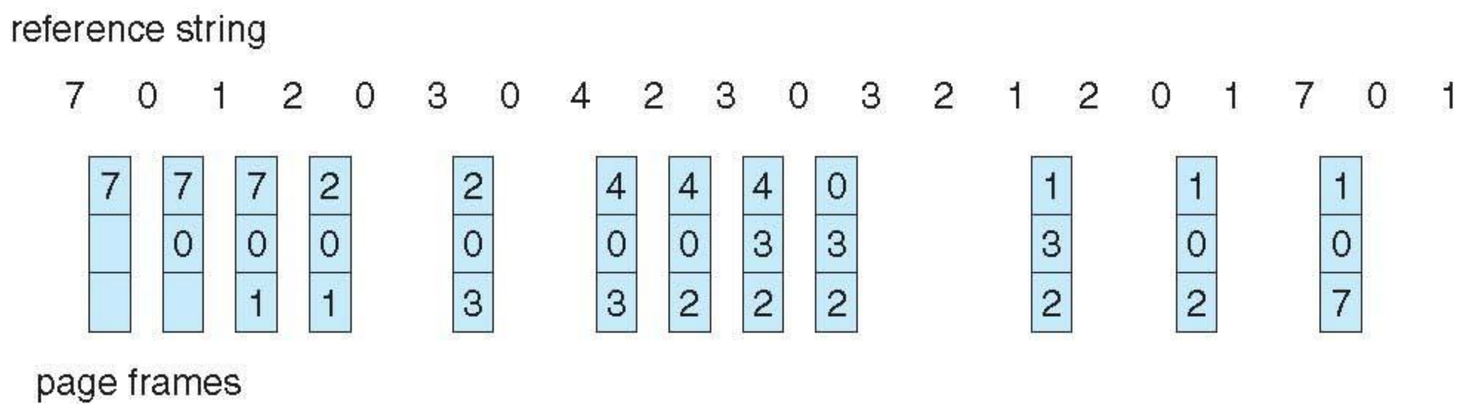
- 9 page faults, and this is optimal for the example

Optimal Algorithm

- Replace page that will not be used for longest period of time
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

- Replace page that has not been used in the most amount of time (Use past knowledge rather than future)
- Associate time of last use with each page

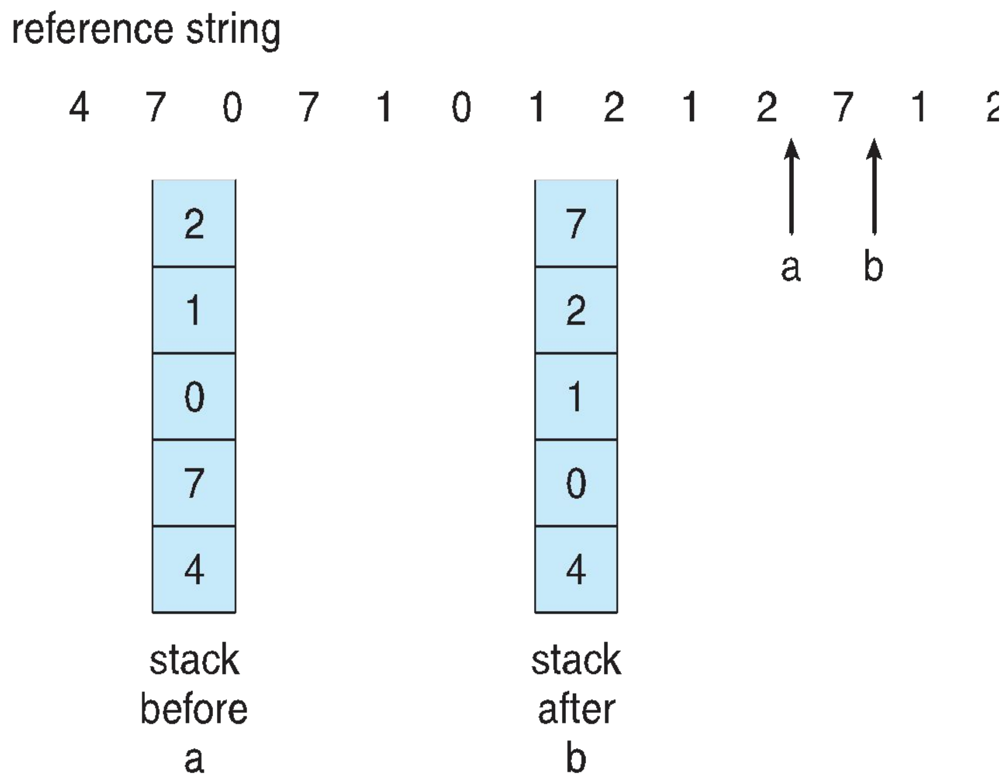


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm Implementation

- Counter implementation
  - 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 find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers
  - Page referenced: move it to the top
  - But each update more expensive

Use Of A Stack to Record Most Recent Page References



- LRU and OPT are cases of **Stack Algorithms** that don't have 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

LRU Approximation Algorithms

1. **Additional-Reference-Bits Algorithm**
2. **Second-Chance Algorithm**

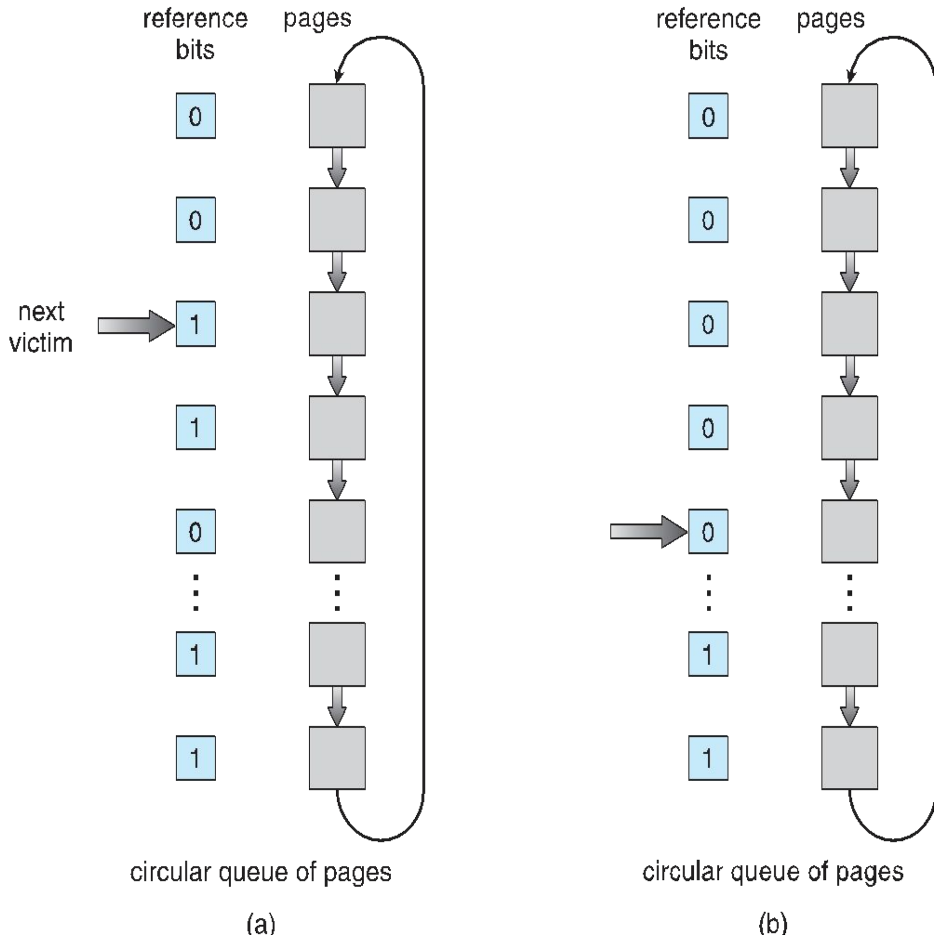
LRU Approximation Algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)

LRU Approximation Algorithms

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. **(0, 0)** neither recently used not modified – best page to replace
2. **(0, 1)** not recently used but modified – not quite as good, must write out before replacement
3. **(1, 0)** recently used but clean – probably will be used again soon
4. **(1, 1)** recently used and modified – probably will be used again soon and need to write out before replacement

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc