BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Virtual Memory

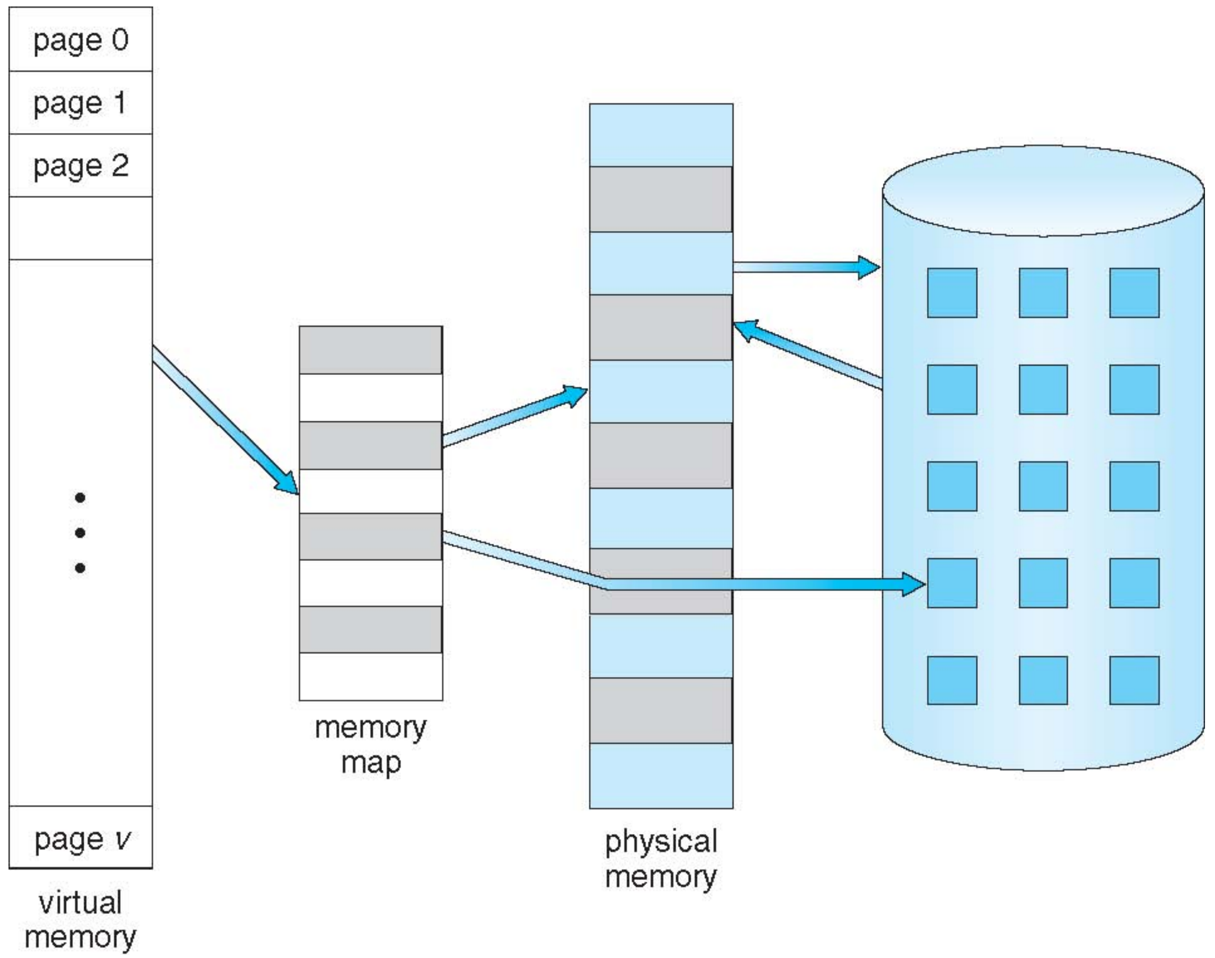
Notice: The slides for this lecture have been largely based on those accompanying an earlier edition of the course text *Operating Systems Concepts, 8th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, of the illustrations contained in this presentation come from this source. Revised by X.M. Based on Professor Perrone's notes.

Virtual Memory

- **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.

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

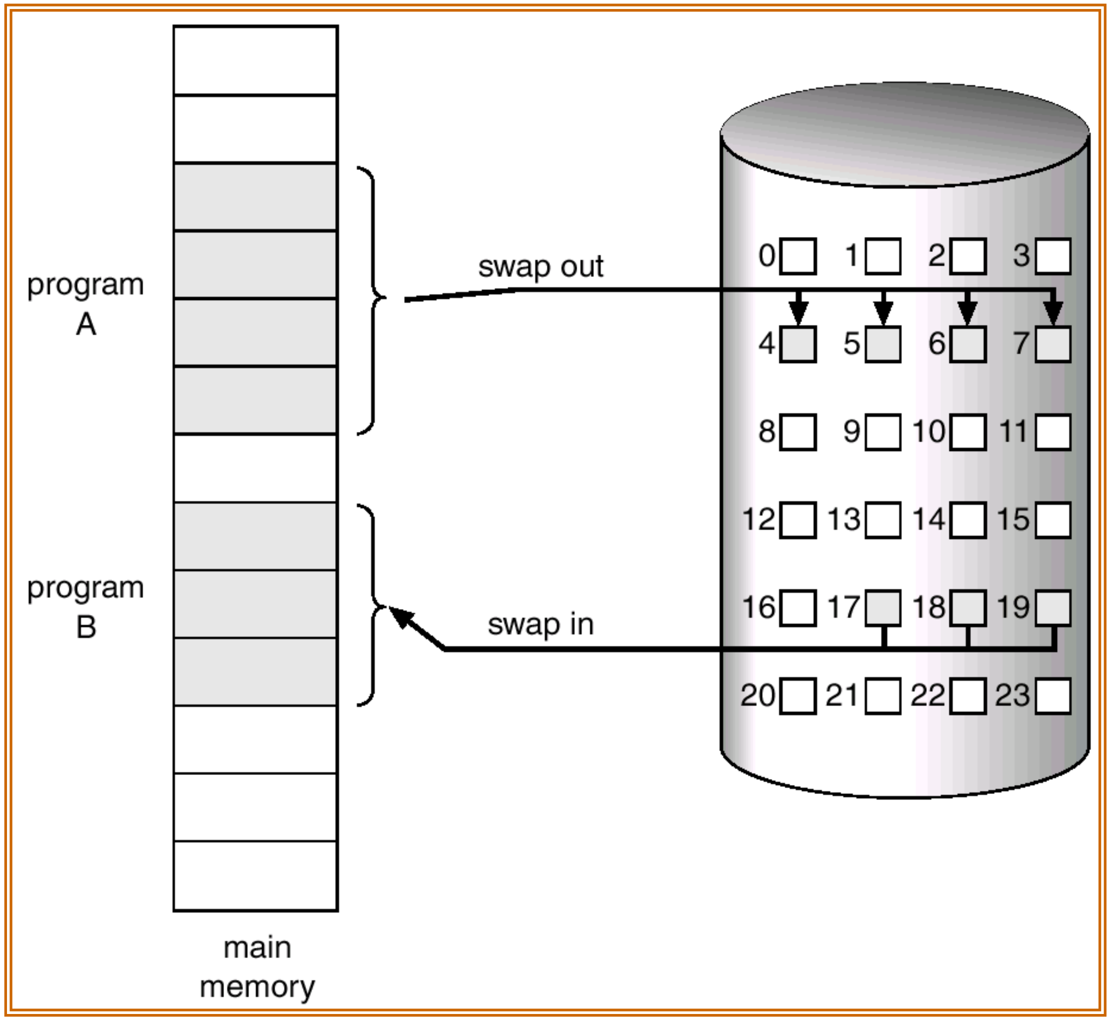
Virtual Memory That is Larger Than Physical Memory



Demand Paging

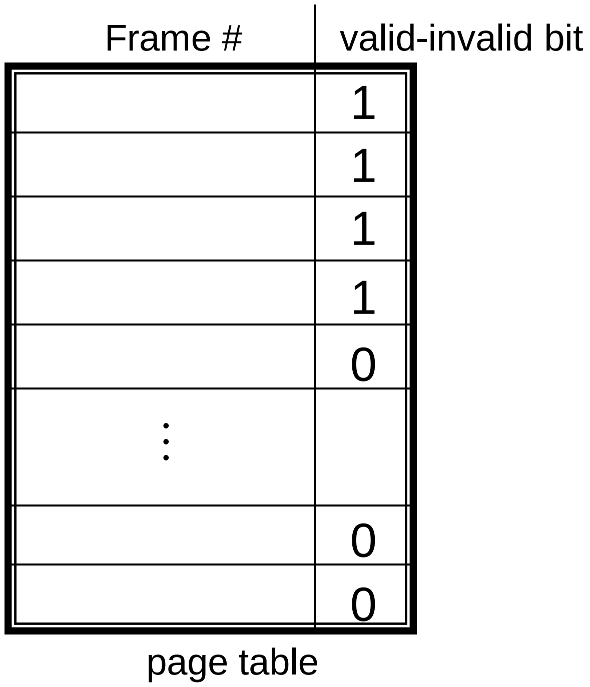
- Bring a page into memory only when it is needed.
  - Less I/O needed.
  - Less memory needed.
  - Faster response.
  - More users.
- When a page is needed (there is a reference to it):
  - invalid reference ⇒ abort.
  - not-in-memory ⇒ bring to memory.
- **Lazy swapper** – never swaps a page into memory unless page will be needed

Transfer of a Paged Memory to Contiguous Disk Space



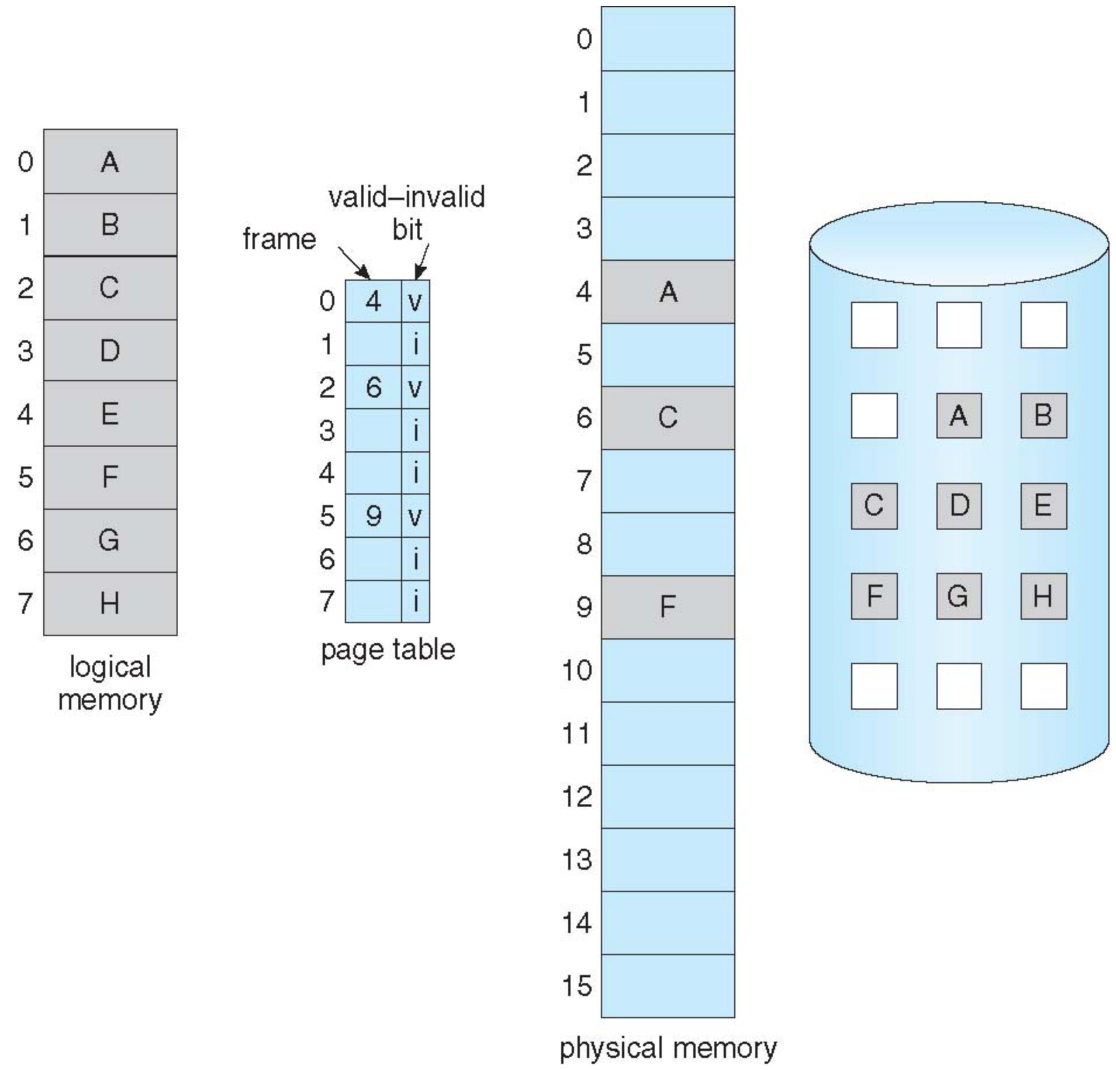
Valid-Invalid Bit

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



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

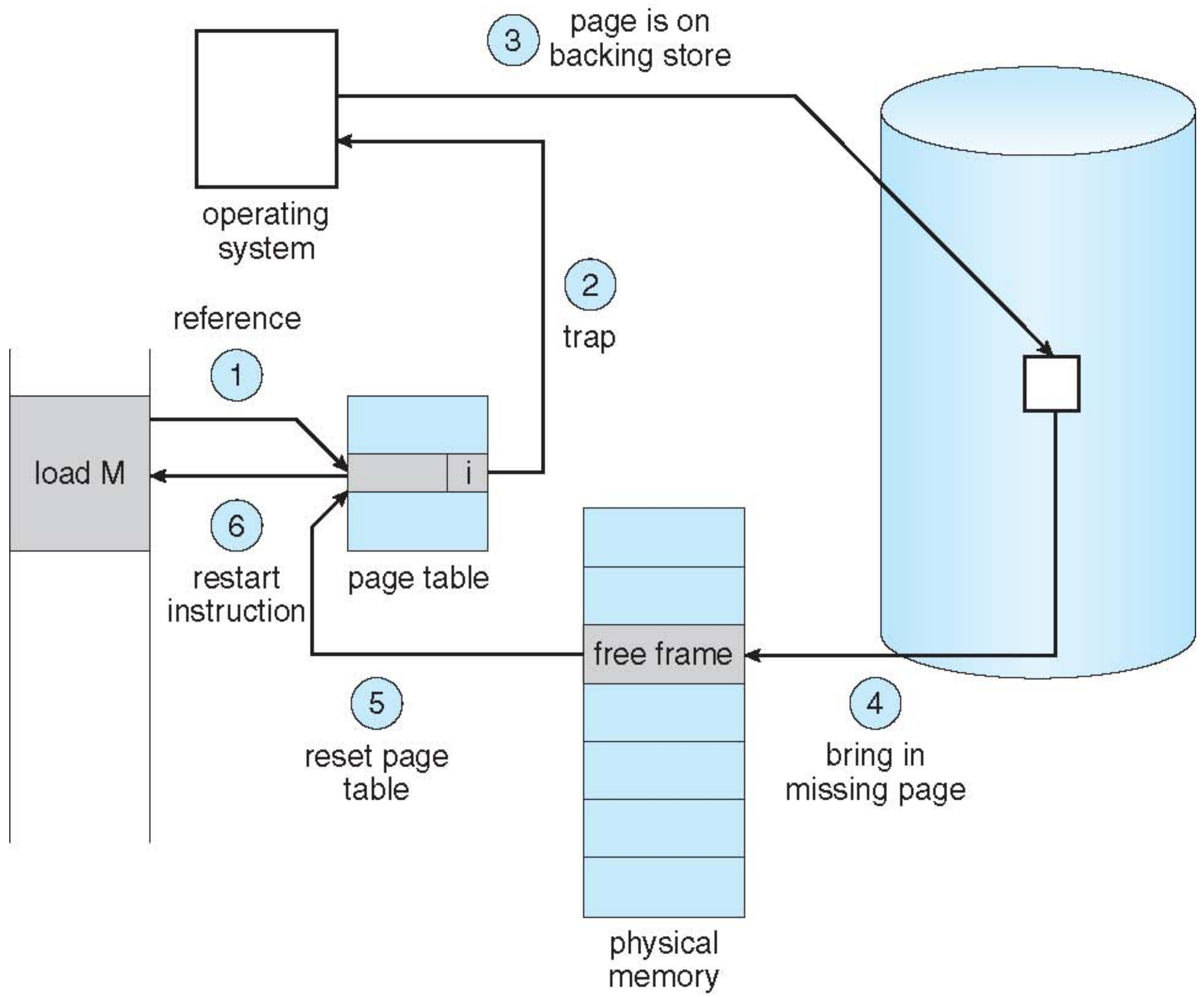
Page Table When Some Pages Are Not in Main Memory



Page Fault and Its Handling

- If there is ever a reference to a page, first reference will trap to OS $\Rightarrow$ page fault.
- OS looks at page table and page limit table to decide:
  - If it was an invalid reference $\Rightarrow$ abort.
  - If it was a reference to a page that is not in memory, continue.
- Get an empty frame from the free-list.
- Bring the page content from disk into frame.
- Correct the page table and make validation bit = 1.
- Restart the instruction that caused the page fault.

Steps in Handling a Page Fault



No free frame: now what?

- **Page replacement:** Are all those pages in memory being referenced? Choose one to swap back out to disk and make room to load a new page.
  - **Algorithm:** How you choose a victim.
  - **Performance:** Want an algorithm that will result in **minimum** number of page faults.

- Side effect: The same page may be brought in and out of memory several times.

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**
  - Peter Denning's *Work Set Model* (more later)

Hardware Support for Demand Paging

- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart
    - Add $s1, $t1, $t2 # easy
    - Mov +4($sp), $t0 # challenge, side effect

Performance of Demand Paging

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

- **Effective Access Time (EAT):**

  EAT = [(1 – p) (memory access)] + [p (page fault overhead)]

where:

page fault overhead = [swap page out] + [swap page in]
+ [restart overhead]

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (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 = 8.2 microseconds.
  This is a slowdown by a factor of 40!! (in comparison to 200 ns)
- 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

Improve Performance

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don’t support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

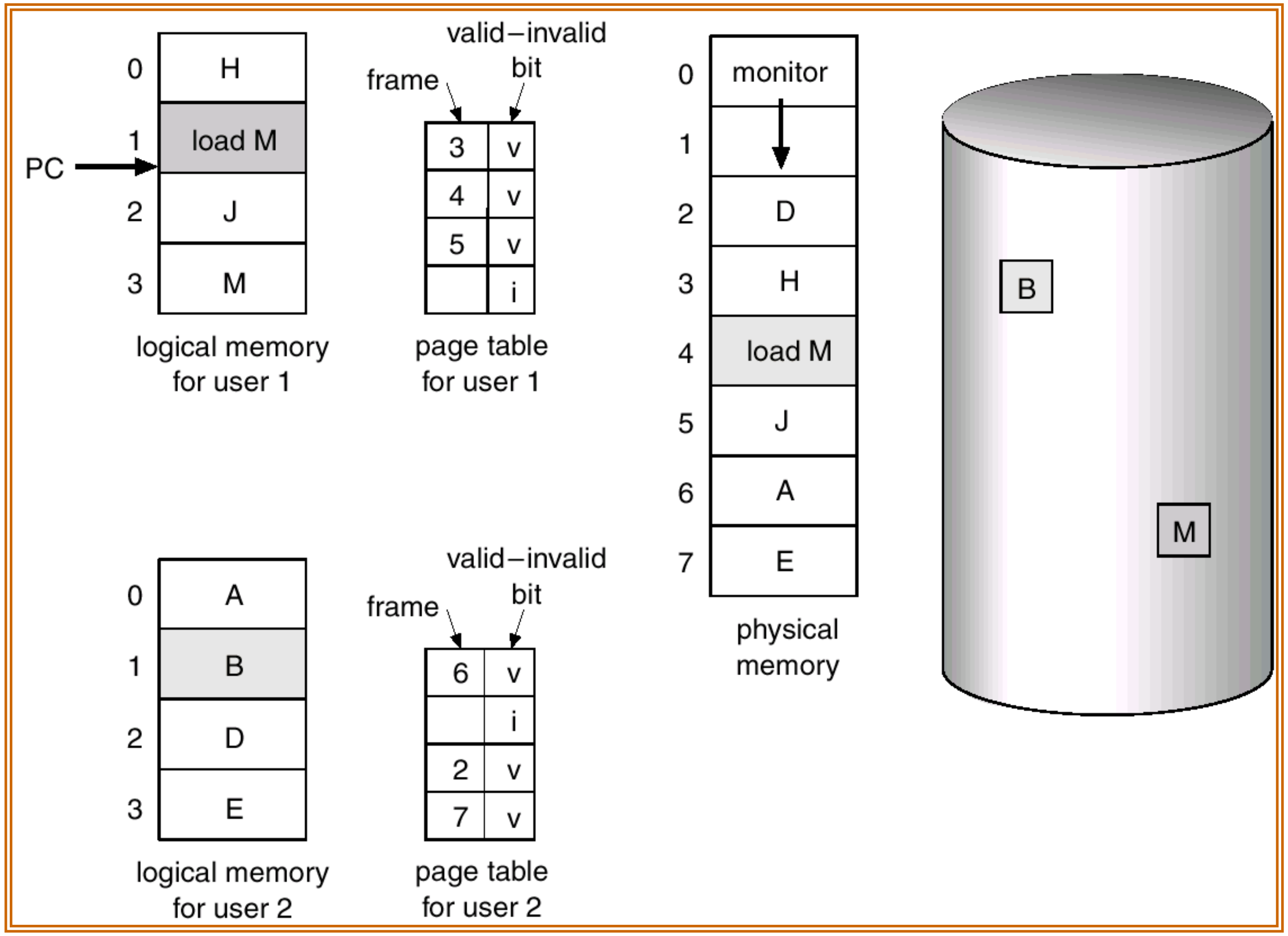
Copy-on-Write

- **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?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

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.

Need For Page Replacement



0
H
1
load M
PC
2
J
3
M
logical memory for user 1
frame
valid–invalid bit
3 v
4 v
5 v
i
page table for user 1
0
A
1
B
2
D
3
E
logical memory for user 2
frame
valid–invalid bit
6 v
i
2 v
7 v
page table for user 2
0 monitor
1
2 D
3 H
4 load M
5 J
6 A
7 E
physical memory
B
M