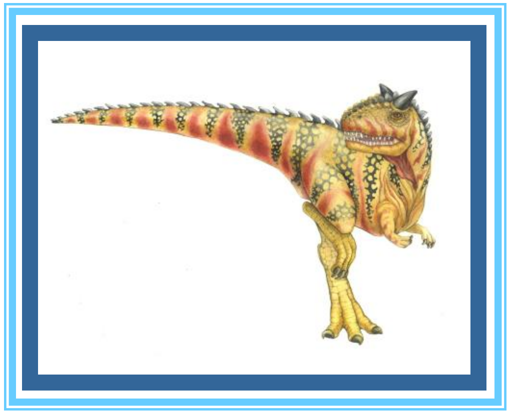
Chapter 9: Virtual Memory



- **Slides mostly borrowed from Silberschatz & Galvin**
  - With occasional modifications from us

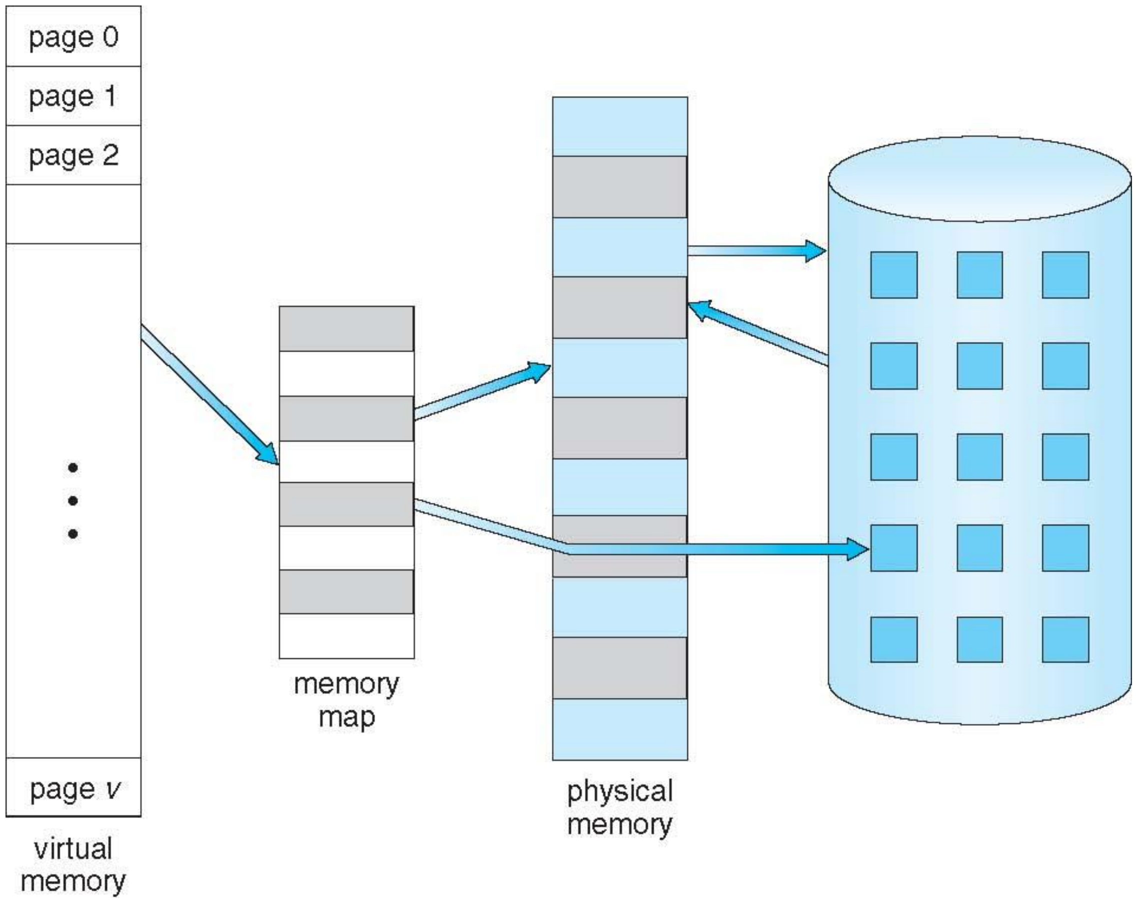
Background

- 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 the same time
- Consider ability to execute partially-loaded program
- What would this ability imply?
  - 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
  - Less I/O needed to load or swap programs into memory --> each user program runs faster

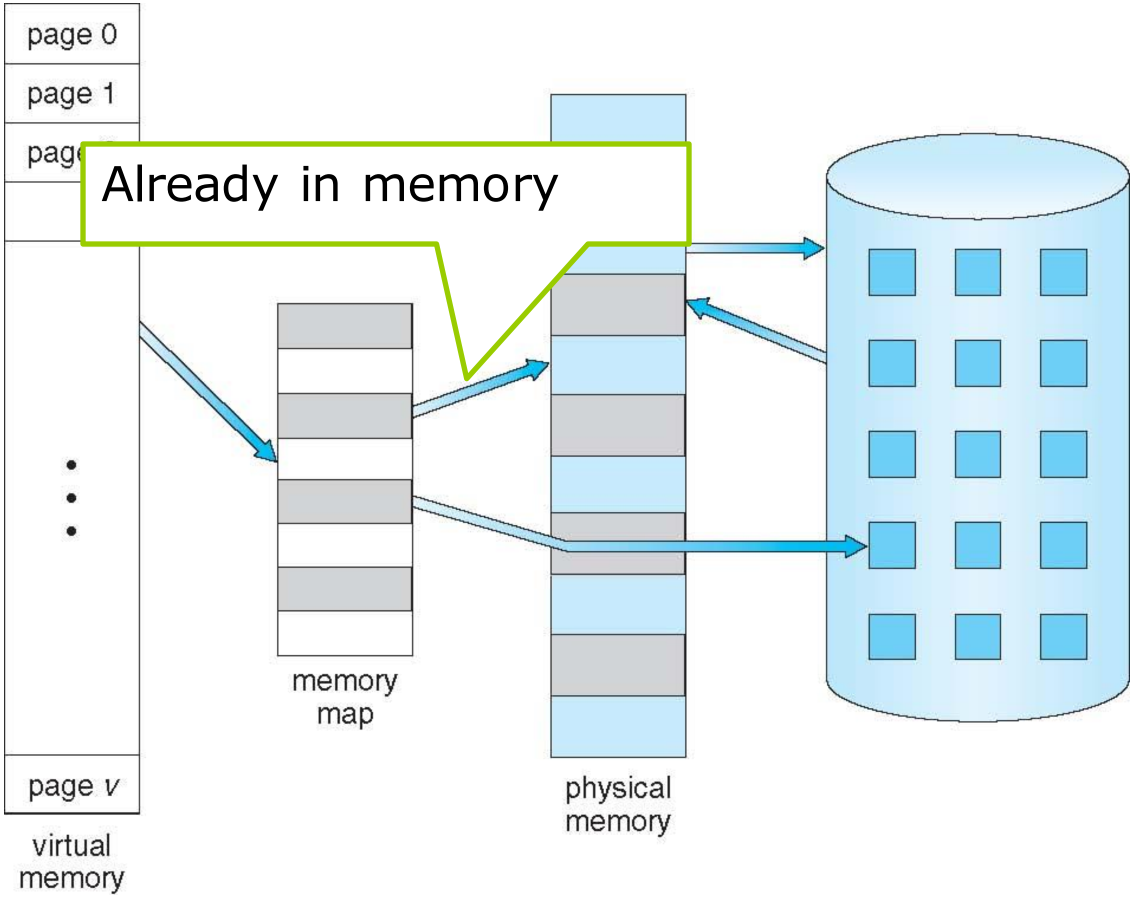
Virtual memory

- **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 be much larger than physical address space

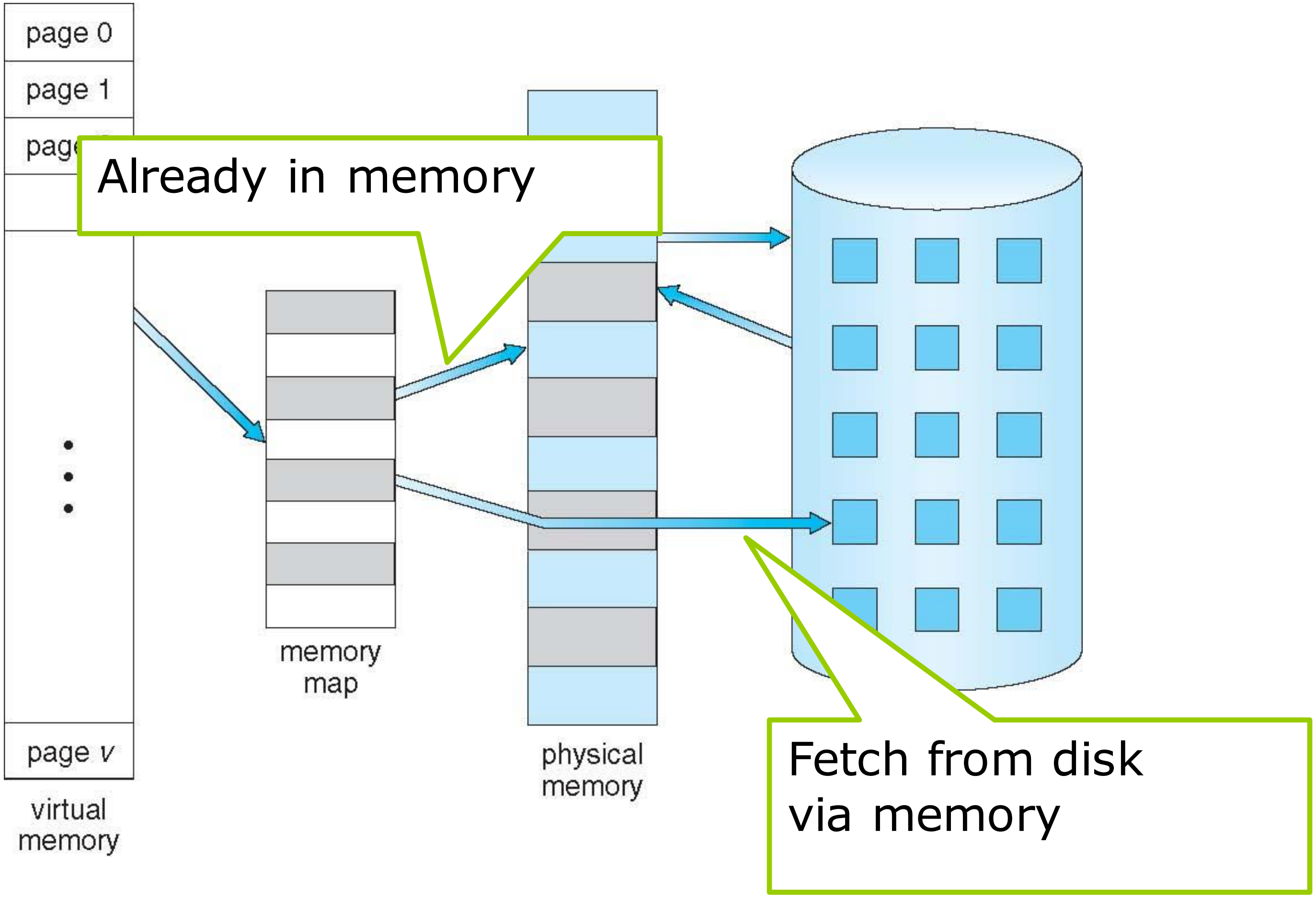
Virtual Memory That is Larger Than Physical Memory



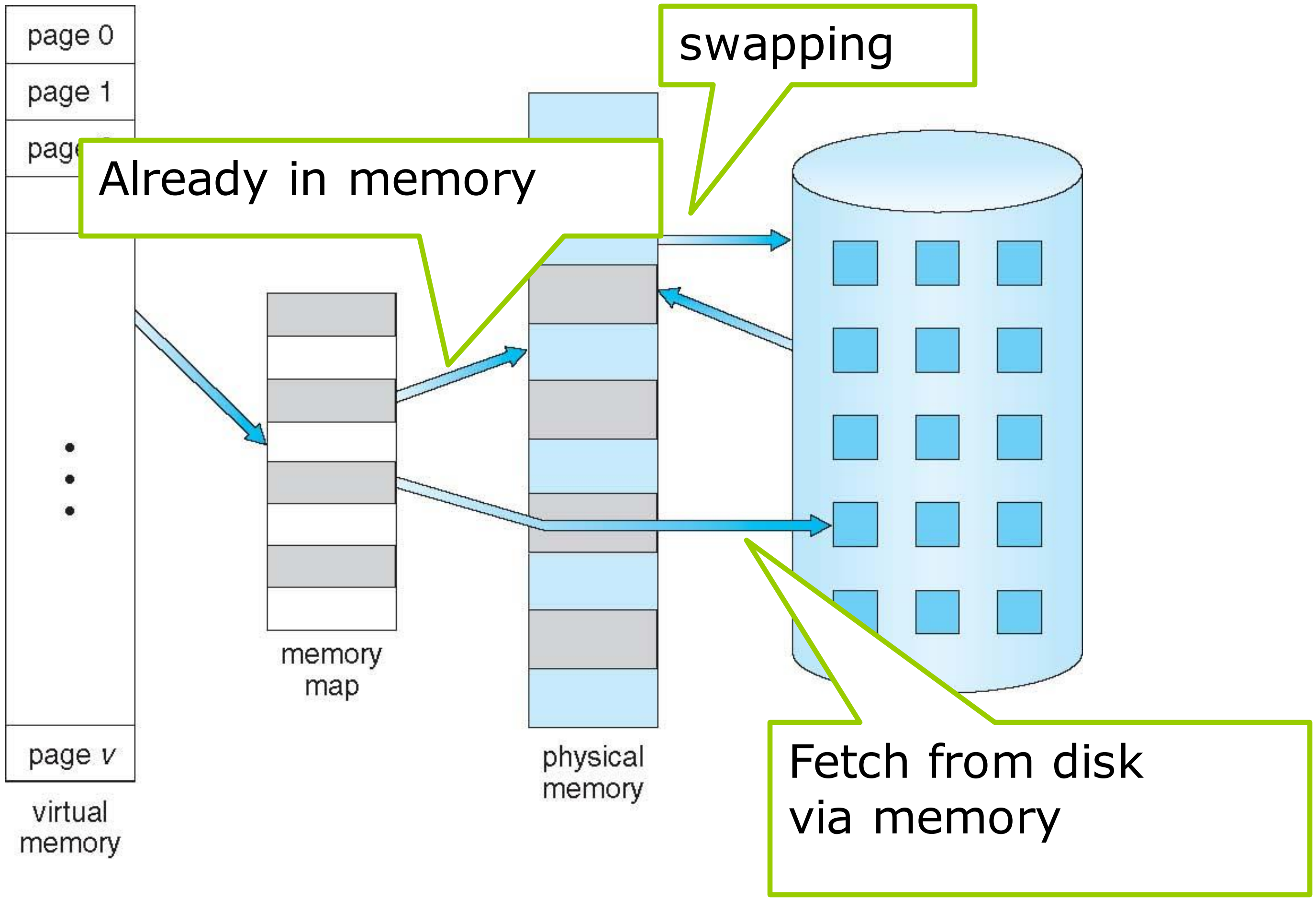
Virtual Memory That is Larger Than Physical Memory



Virtual Memory That is Larger Than Physical Memory

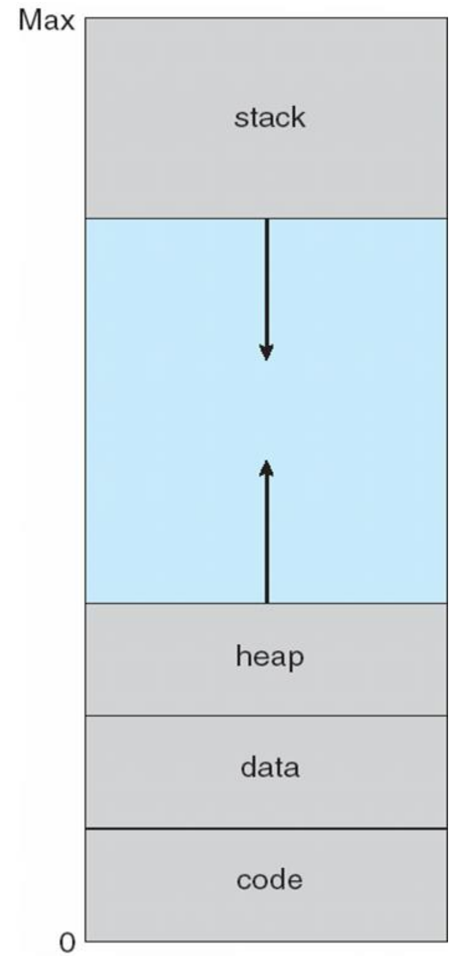


Virtual Memory That is Larger Than Physical Memory



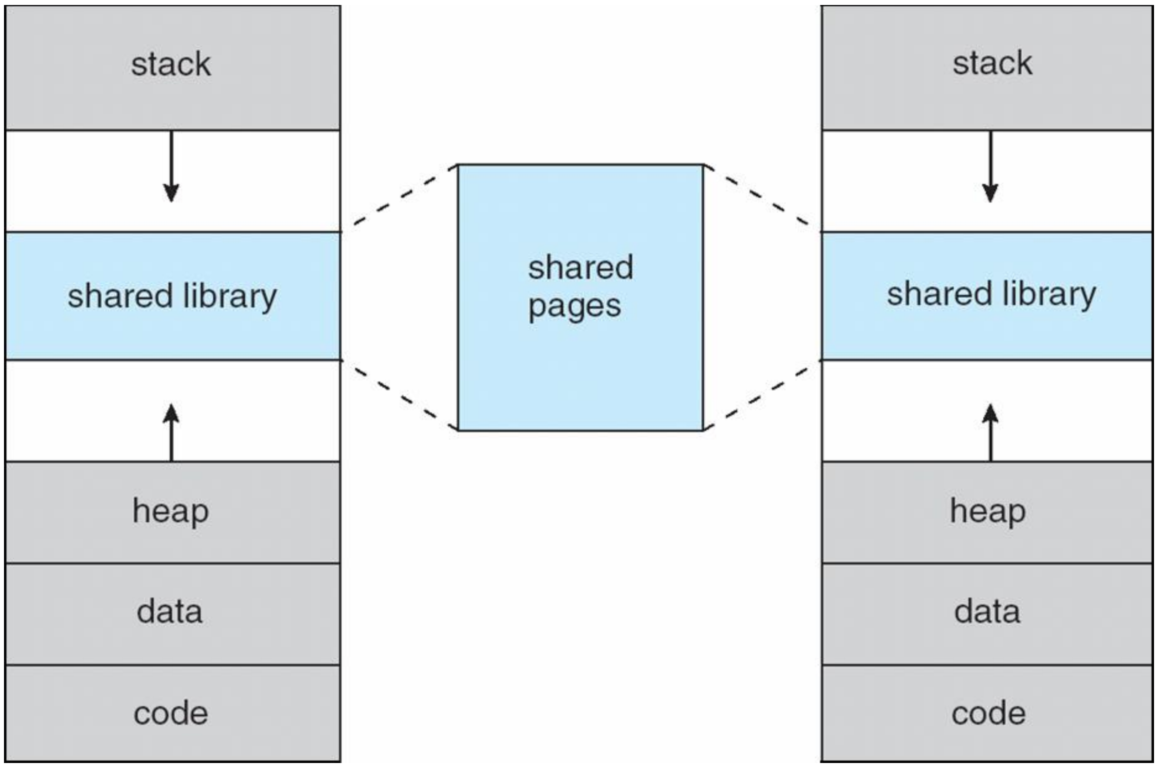
Virtual address space

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space (Max)
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical address to physical address
- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Unused address space between the two is hole
  - No physical memory needed until heap or stack grows to a new page



Virtual address space

- What are the advantages?
  - 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



Virtual memory: advantages

- Virtual memory
  - Separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution

- Advantages
  - Allows a large virtual memory to be provided to programmers even when a smaller physical memory is available
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual memory: implementation

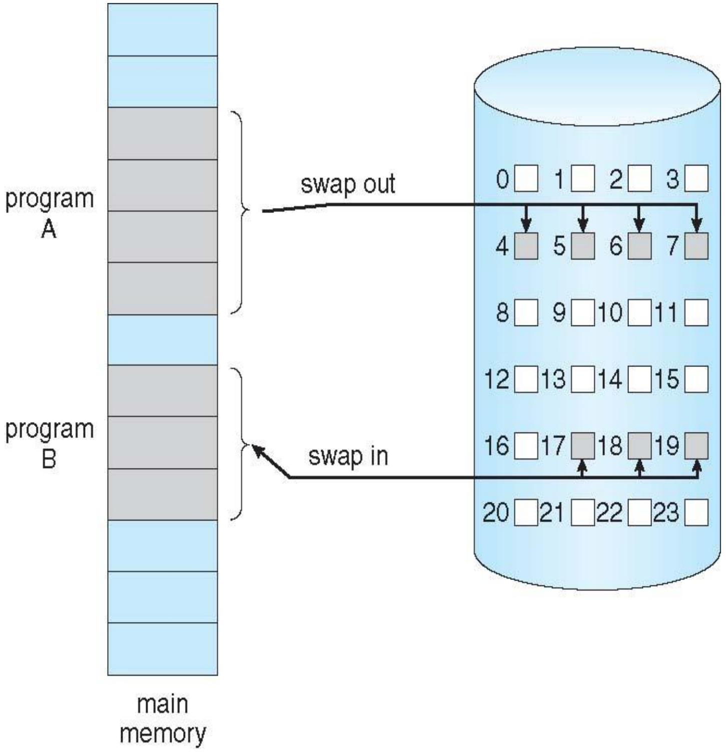
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

Demand Paging

- 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 processes, ...
- Similar to paging system with swapping

- Page is needed (CPU has referred to it)
  - invalid reference --> abort
  - Page is already **memory resident** --> no difference from non demand-paging scenario
  - Page is not memory resident -->
    - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

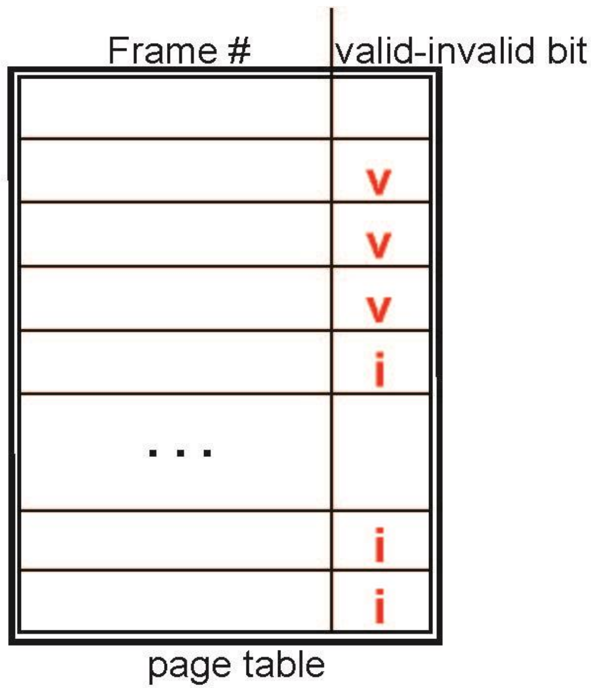


Lazy swapper (pager)

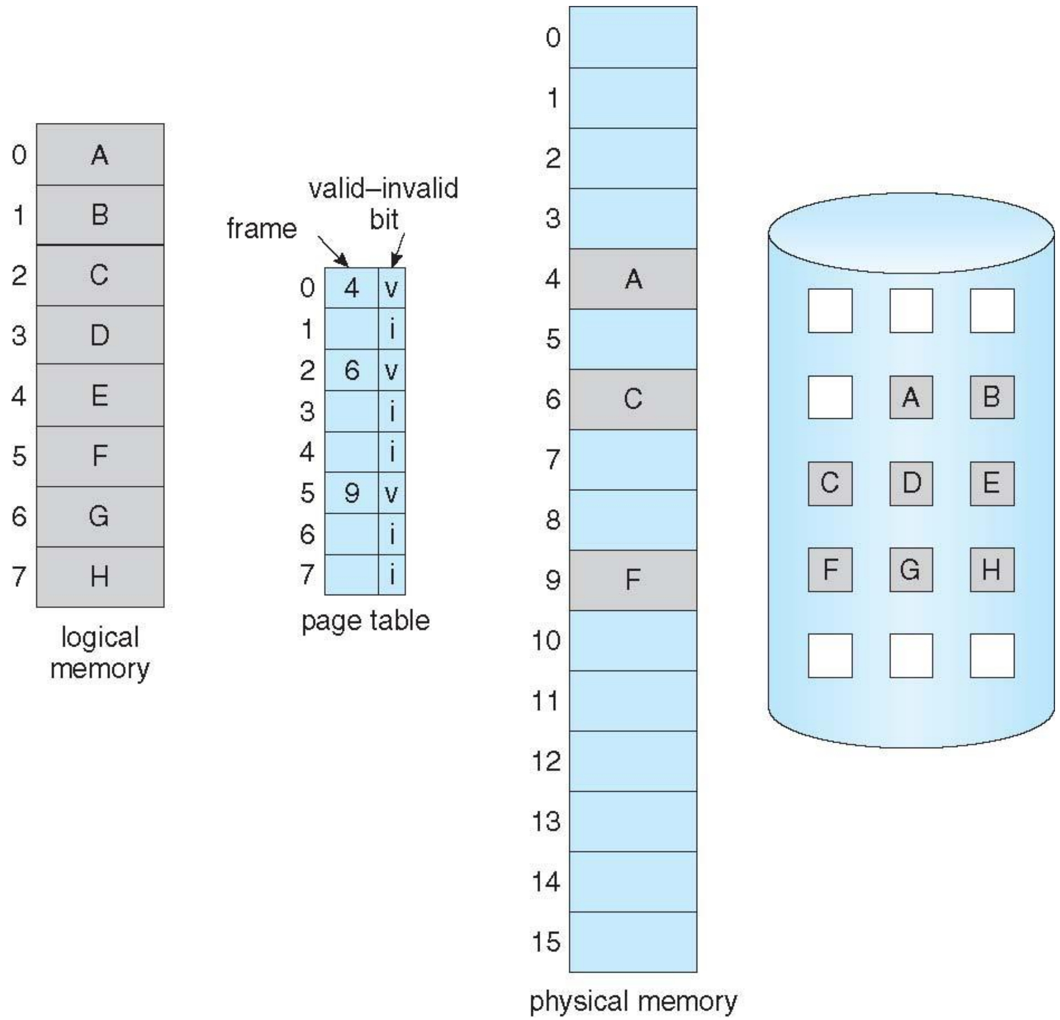
- Swapper that deals with pages is called **pager**
  - **Lazy swapper** – never swaps a page into memory unless page is needed
  - Before swapping out a process, pager guesses which pages will be used
  - Next time the process is swapped in, instead of whole process, pager brings in only those pages into memory
  - Need a mechanism to determine that set of pages

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:



Page Table When Some Pages Are Not in Main Memory

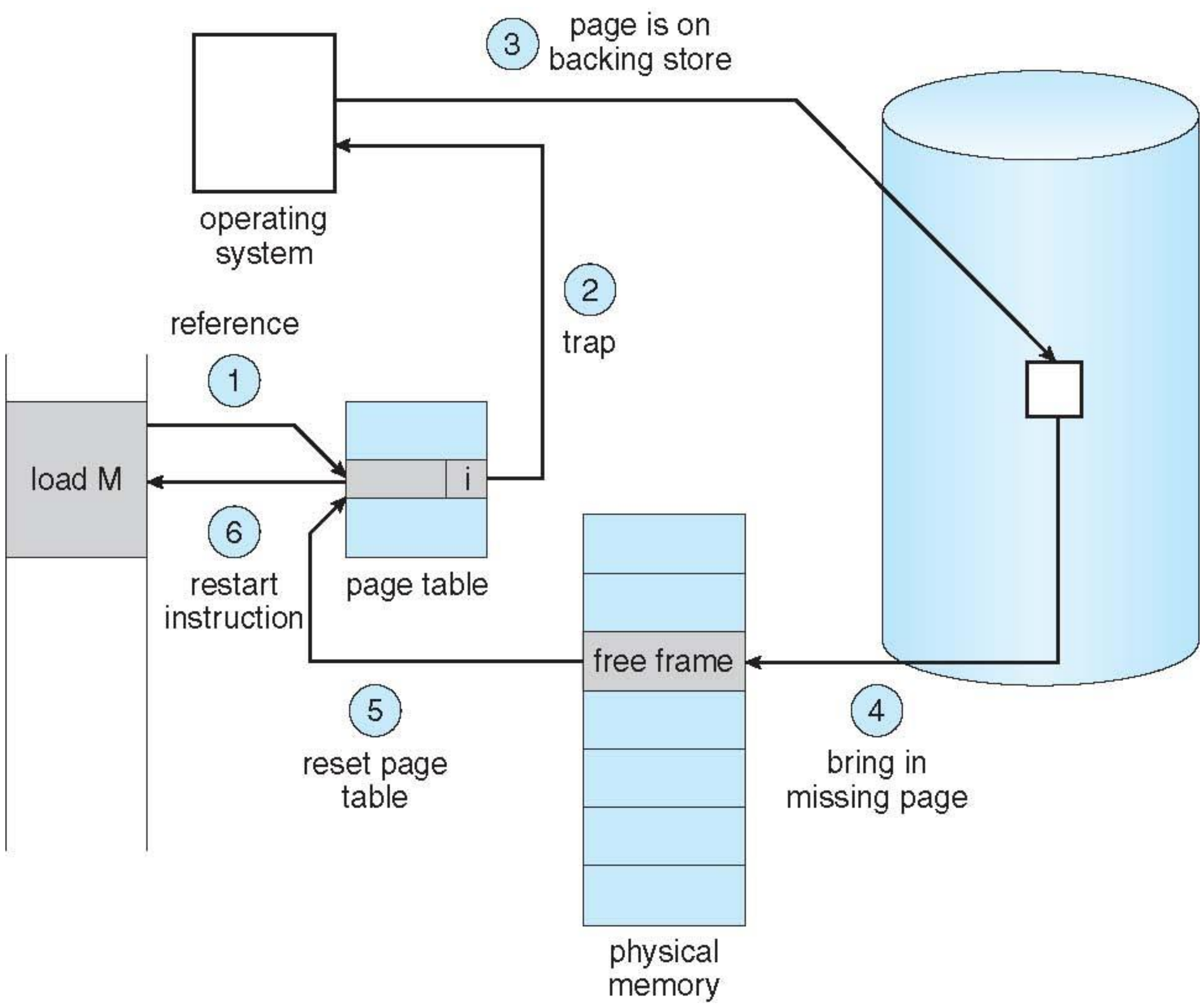


Page Fault

- Page Fault
  - During MMU address translation, if valid–invalid bit in page table entry is **i**
  - First reference to every page
  - Page Fault will trap to operating system

1. Operating system decides:
   - If invalid memory reference: abort
   - If just page not in memory, do steps 2--5
2. Find free frame in memory
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory: Set validation bit = **v**
5. 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
  - **Pure Demand Paging**
  - OS sets PC to first instruction of process; Page not in memory --> page fault
  - Similarly, page fault for every other page on first access
- A given instruction can access multiple pages --> ***multiple page faults***
  - E.g., consider fetch and execution of instruction that adds two numbers from memory and stores the result back to memory
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart (Re-execute instruction that resulted in page fault)

Performance of Demand Paging

Stages in Demand Paging (worse case)

1. **Say process Pi causes Page Fault -->** Trap to the operating system
2. Save the user registers and process state of Pi
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 disk
5. Issue a read from the disk to a free frame [state of Pi changed to waiting]
6. While Pi is waiting, allocate the CPU to some other process Pj
7. Receive an interrupt from the disk I/O subsystem (***I/O completed***)
8. Save the registers and process state for the other process Pj (which was running)
9. Determine that the interrupt was from the disk
10. Correct the page table of Pi and other tables to show page is now in memory
11. Wait for the CPU to be allocated to process Pi again
12. When Pi allocated CPU again: Restore the user registers, process state, and new page table of Pi, 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; just a small amount of time
  - **Read the page** – lots of time
  - **Restart the process** – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access time

  + p (page fault overhead + swap page out + swap page in)

Demand Paging :: Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 nsec + p (8 msec)

  = (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!!
- If want performance degradation < 10 %

  220 > 200 + 7,999,800 x p

  or, 20 > 7,999,800 x p

  or, p < .0000025

  < one page fault in every 400,000 memory accesses

Optimizations of Demand Paging

Dirty bit

- Associate a "dirty bit" with every page to note if the page has been modified while in memory
- Can decide whether a page needs to be swapped out before replacing it

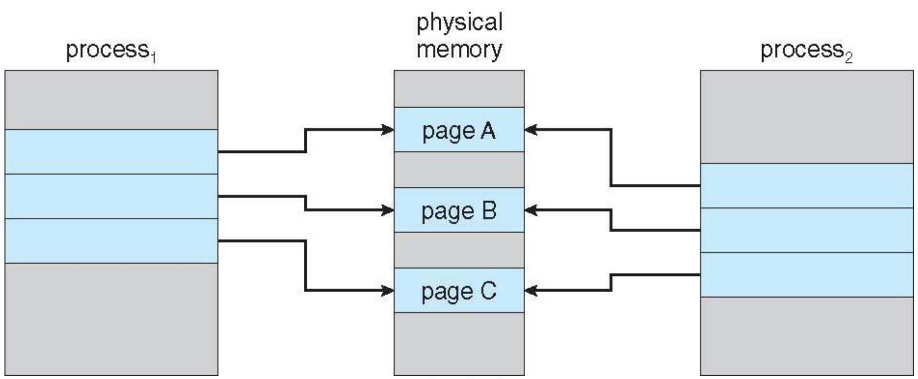
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\text{EAT} = (1 - p) \times \text{memory access time} + p\,(\text{page fault overhead} + \textbf{\textit{swap page out}} + \text{swap page in})$$

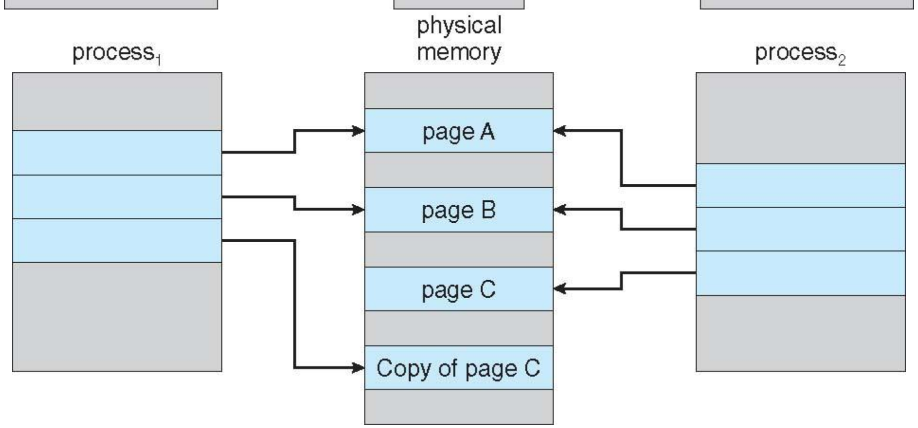
Only if the replaced page is dirty

Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - Read-only pages (e.g., code) can be shared by parent and child
  - If either process modifies a shared page, only then is the page copied
  - Allows more efficient process creation as only modified pages are copied



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Copy-on-Write (cont.)

- In general, free frames for the COW pages are allocated from a pool of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution (used for duplication of COW pages, or when stack/heap of a process has to expand)
  - Why zero-out a page before allocating it?

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

Page Replacement

What Happens if There is no Free Frame?

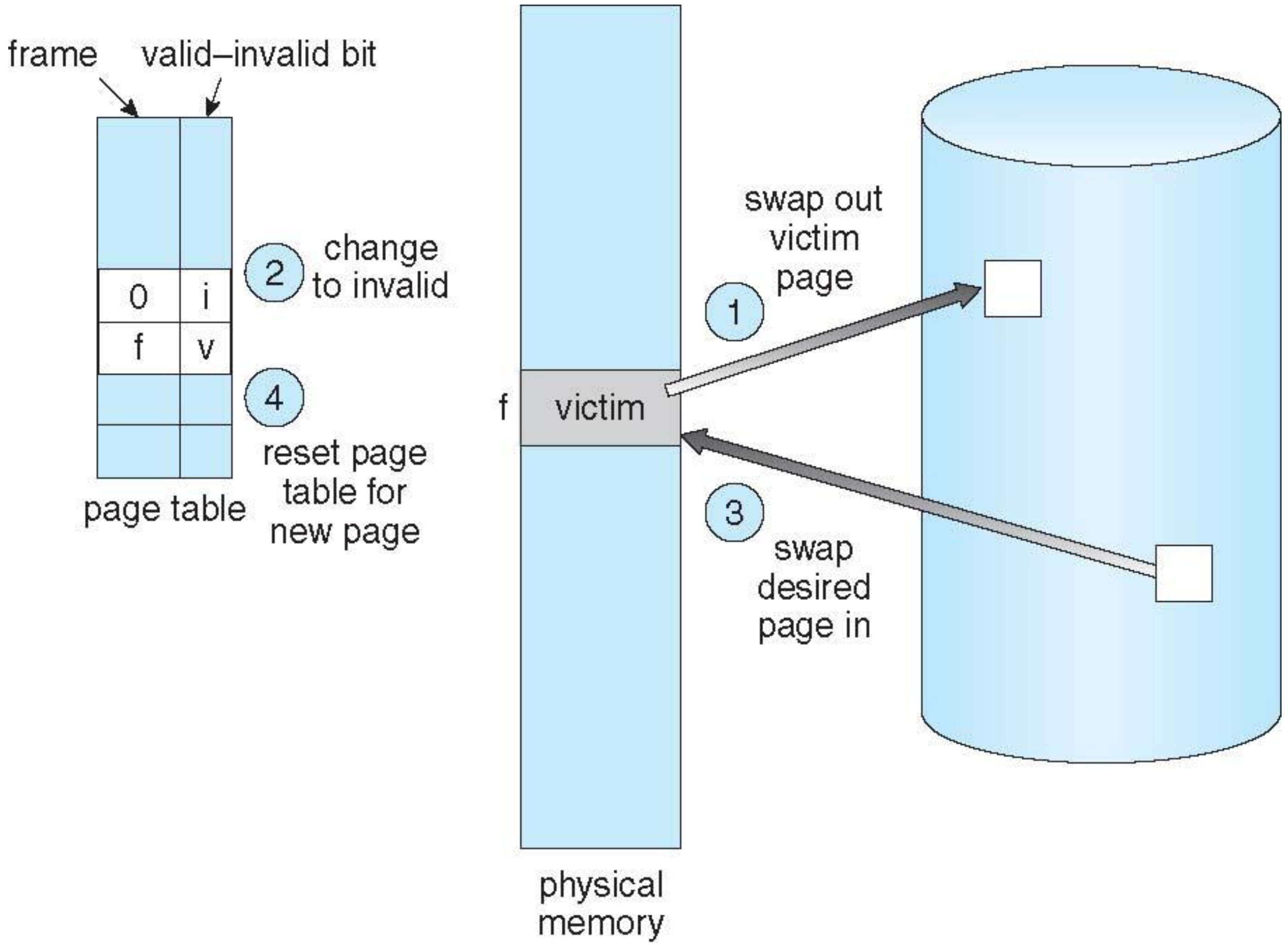
- All pages are used up by the processes / kernel, I/O buffers, etc. BUT a page that is not in memory needs to be accessed by CPU
- **Page replacement** – find some page in memory, but not *really* in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Advantage - Prevent ***over-allocation*** of memory by modifying page-fault service routine to include page replacement
  - Too many frames should not be allocated to a process
- Optimization: Use ***modify (dirty) bit*** to reduce overhead of page transfers
  - Only modified pages (marked as dirty) are written to disk before replacement

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



A Simple Calculation

- Let p : probability of page fault

  δ : probability that victim page is dirty

  t_{MM} : access time of main memory

  t_{swap} : time required to swap-in or swap-out a page

- Effective memory access time

  $$\mathbf{EAT = (1 - p)(t_{MM} + t_{MM}) + p\,[\{(1 - \delta)\,t_{swap} + \delta\,(2\,t_{swap})\} + t_{MM}]}$$

The above expression does not consider the speed-up due to TLB.

Modify the expression considering that TLB is also present.