CSC 4320/6320: Operating Systems

Chapter 10: Virtual Memory-I

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman

Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

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
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

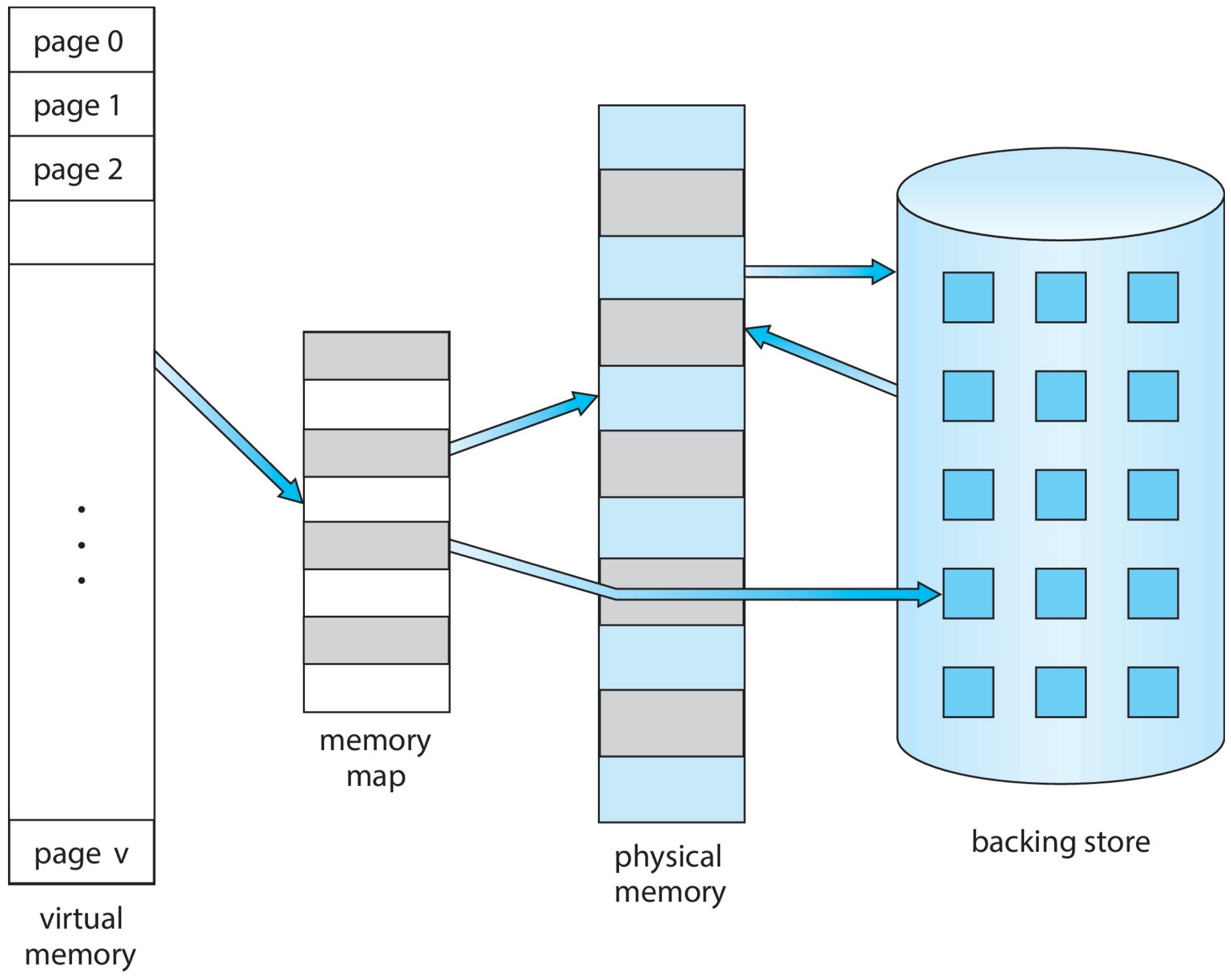
Virtual memory

- **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 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

Virtual memory (Cont.)

- **Virtual address space** – logical view of how a 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 memory can be implemented via:
  - Demand paging
  - Demand segmentation

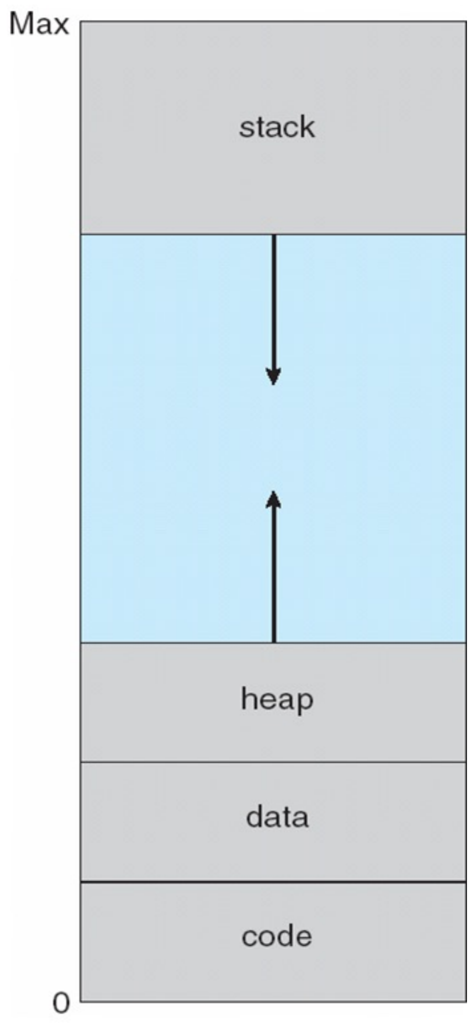
Virtual Memory That is Larger Than Physical Memory



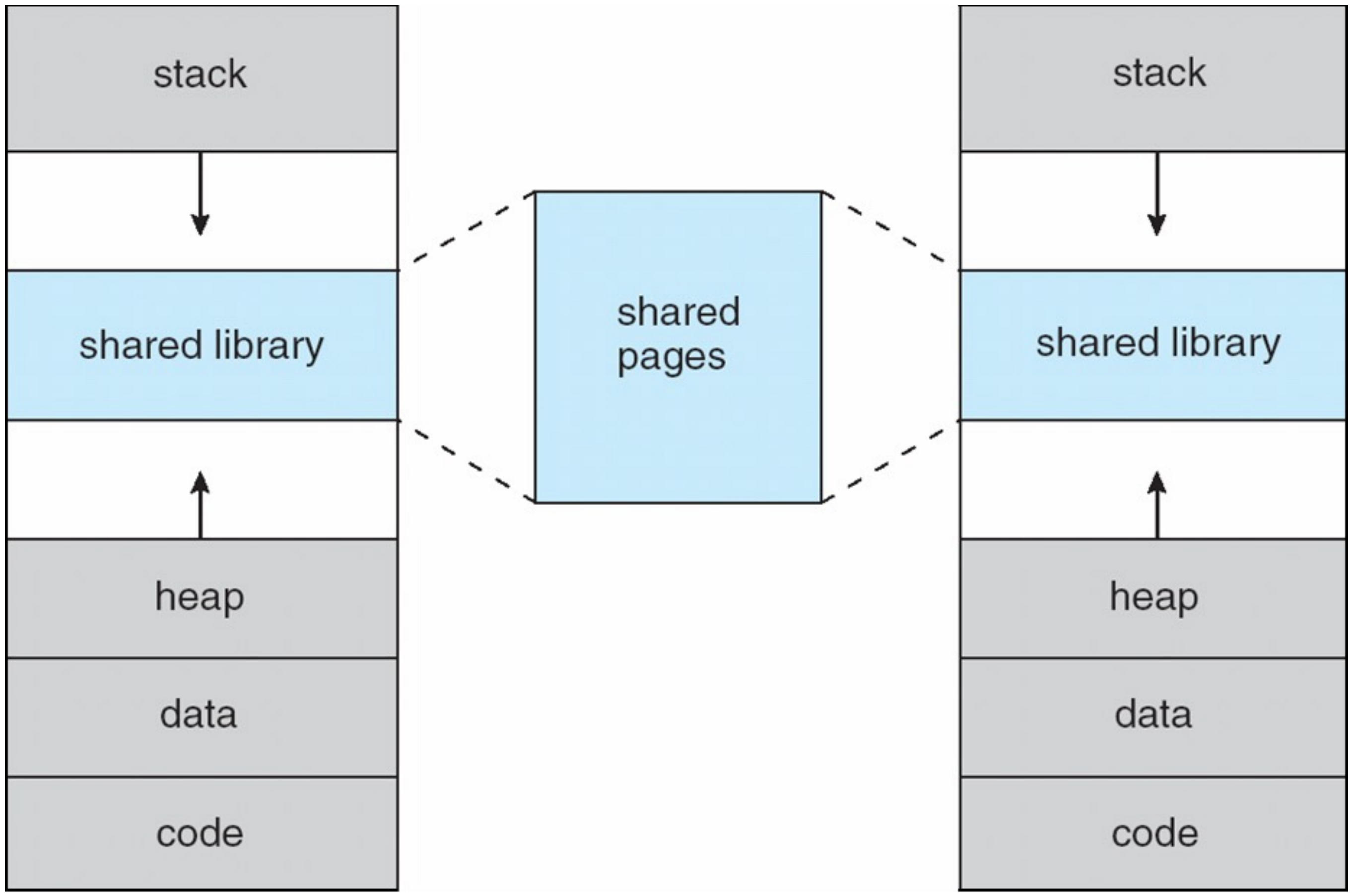
- Virtual memory makes the task of programming much easier
- the programmer no longer needs to worry about the amount of physical memory available;
- can concentrate instead on programming the problem that is to be solved.

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 for holes 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. One process can create a region of memory and share with another process.
- Pages can be shared during `fork()`, thus speeding process creation



Shared Library Using Virtual Memory



virtual memory allows files and memory to be shared by two or more processes through page sharing. For example, sharing standard C library

Questions

1. Which of the following is a benefit of allowing a program that is only partially in memory to execute?

A) Programs can be written to use more memory than is available in physical memory.
B) CPU utilization and throughput is increased.
C) Less I/O is needed to load or swap each user program into memory.
D) All of the above ✓

2. In systems that support virtual memory, ____.

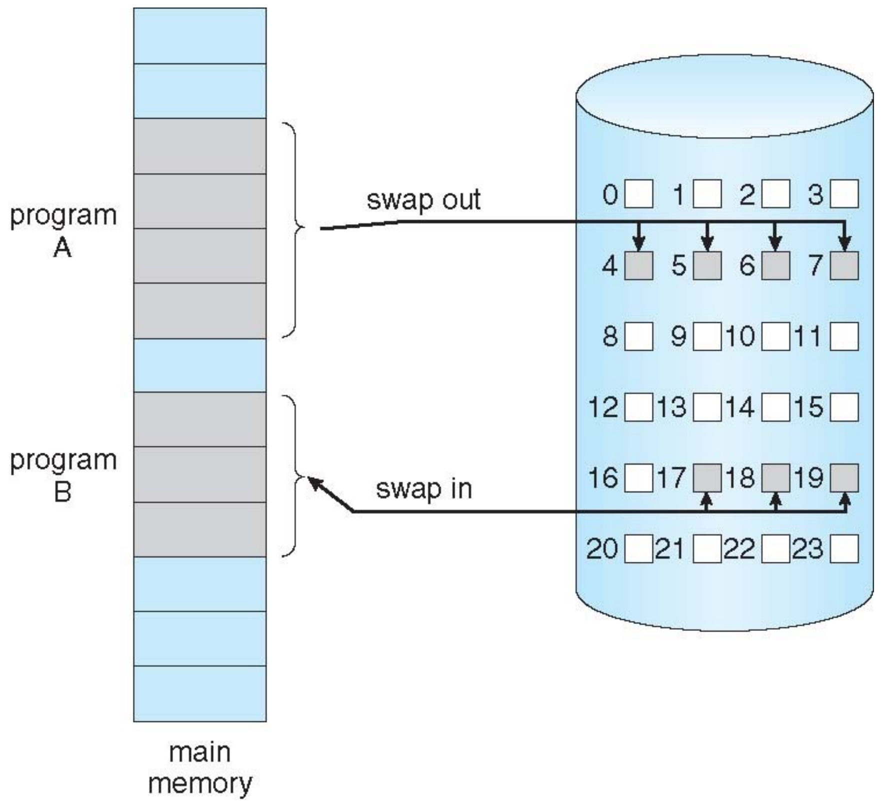
A) virtual memory is separated from logical memory.
B) virtual memory is separated from physical memory.
C) physical memory is separated from secondary storage.
D) physical memory is separated from logical memory. ✓

3. In general, virtual memory decreases the degree of multiprogramming in a system.

True
False ✓

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



Demand Paging

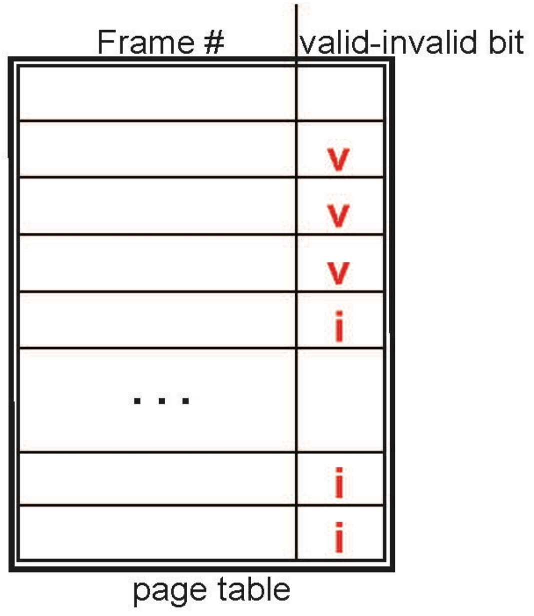
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort (after checking PCB)
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed (demand paging)
  - Swapper that deals with pages is a **pager (OS module)**

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
  - While swapping out, keep pages that are likely to be used soon.
- Instead, in demand paging, pager brings in only those pages into memory for which page fault occurs.
- 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

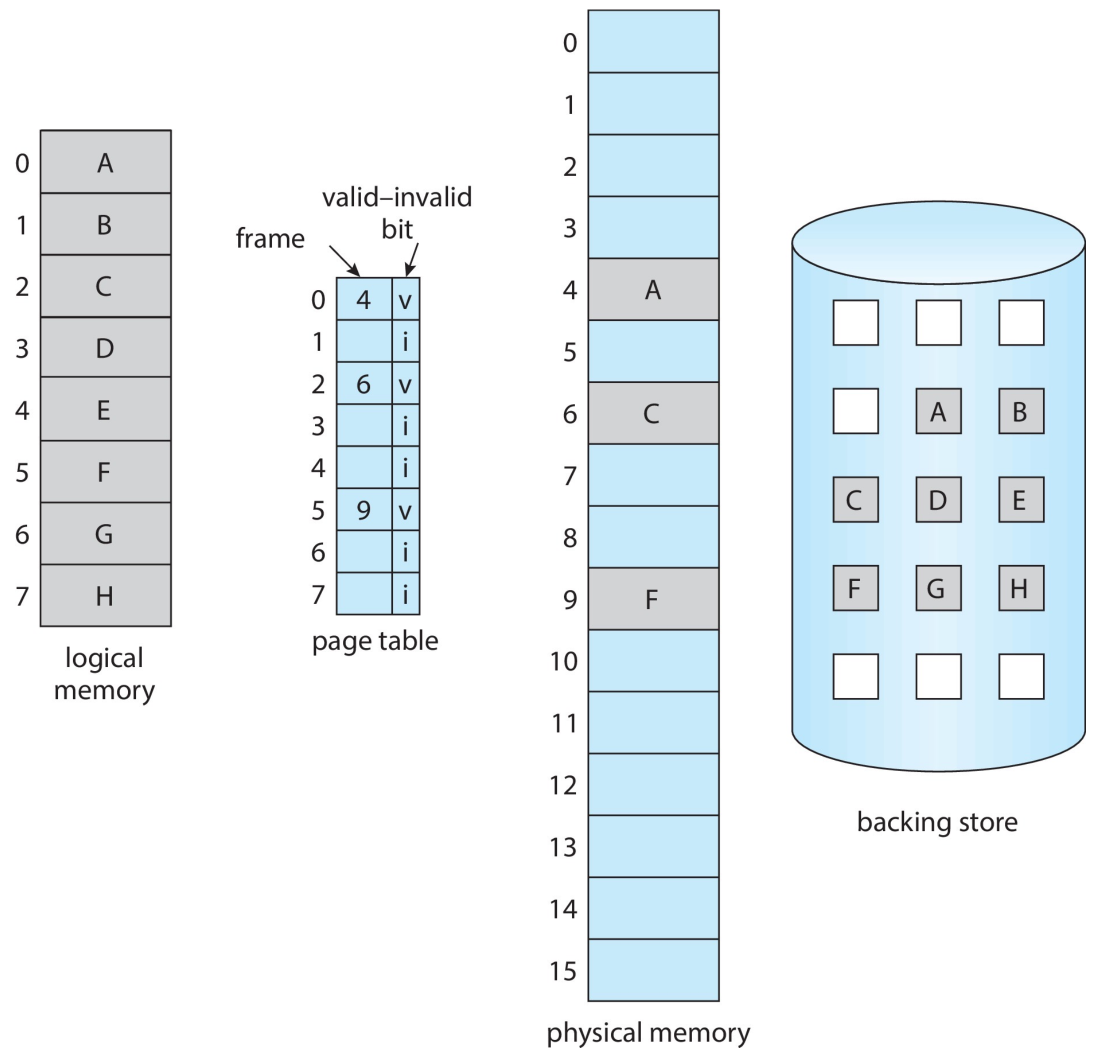
Valid-Invalid Bit

- 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

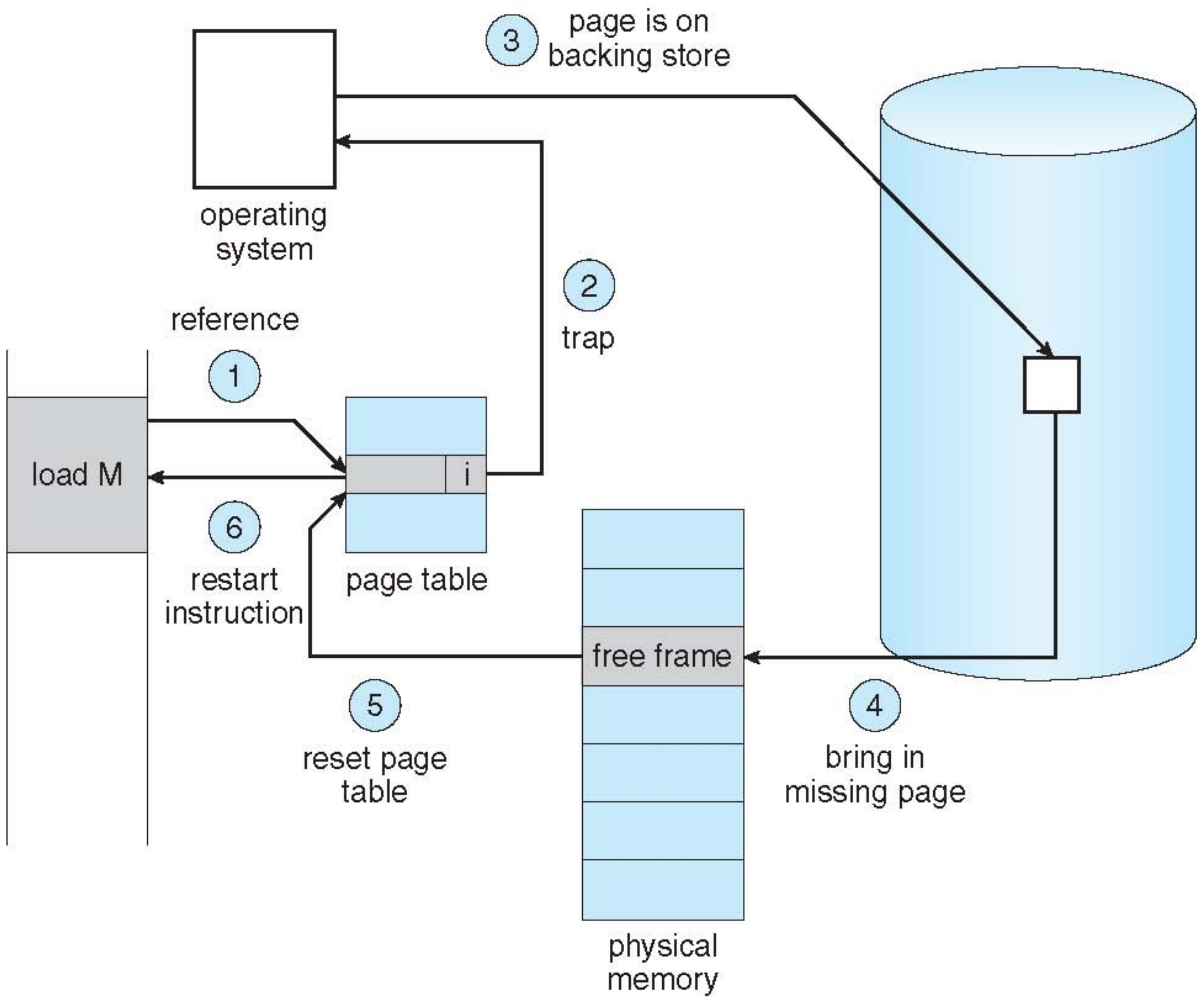
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table (in PCB) to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
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 (Cont.)



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 (tendency to reference memory in patterns rather than randomly)**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Worst-case on instruction restart

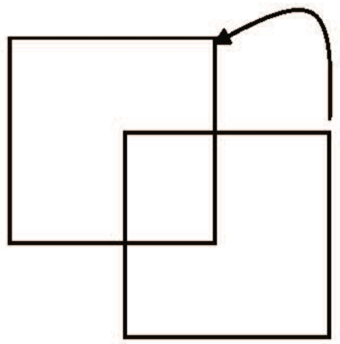
Consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If fault occurs when we try to store in C, we bring the desired page in, correct the page table, and restart the instruction. The restart will require start over at step 1.

Instruction Restart

- Consider an instruction that could access several different locations
  - Block move



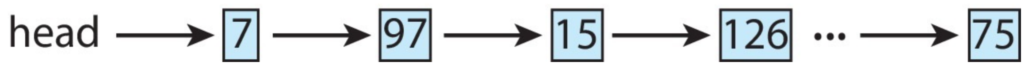
  - Auto increment/decrement location
    - E.g. in-place operations on locations
  - Restart the whole operation?
    - What if source and destination overlap?

Solution for block move:

- The microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified.
- Register-based move for overlapped locations (registers hold pre-modified data only for overwritten locations). In case of very large overwritten locations (more than register capacity), OS may also use kernel memory.

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

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:
    a) Wait in a queue for this device until the read request is serviced
    b) Wait for the device seek and/or latency time
    c) Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

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

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

$$\begin{aligned} \text{EAT} = {} & (1 - p) \text{ x memory access} \\ & + p \text{ (page fault overhead} \\ & + \text{swap page out} \\ & + \text{swap page in)} \end{aligned}$$

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!!
- 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

Questions

1. On a system with demand-paging, a process will experience a high page fault rate when the process begins execution.

[✓ True]

False

2. A page fault occurs when

A) a page in memory get corrupted.

B) the size of a process is larger than the size of the physical memory.

[✓ C) a process tries to access a page that is not loaded in memory.]

D) the page table is not large enough to include all page table entries.

3. If memory access time is 250 nanoseconds and average page fault service time 10 milliseconds, the probability of page faults must be less _______ to keep the performance degradation less than 20%.

A) 0.0000025

B) 0.000005

C) 0.0000075

D) 0.00001

Do it yourself