Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Virtual Memory

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Chapter 10: Virtual Memory

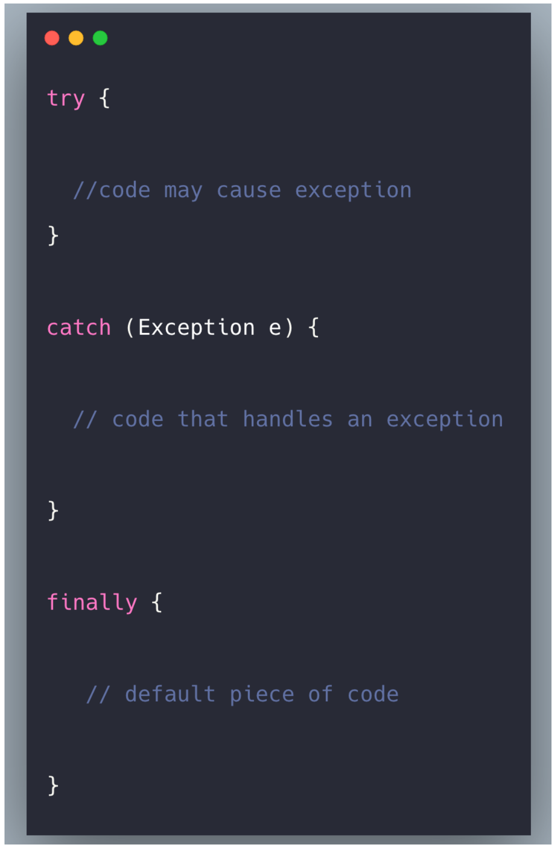
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement

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.

Background

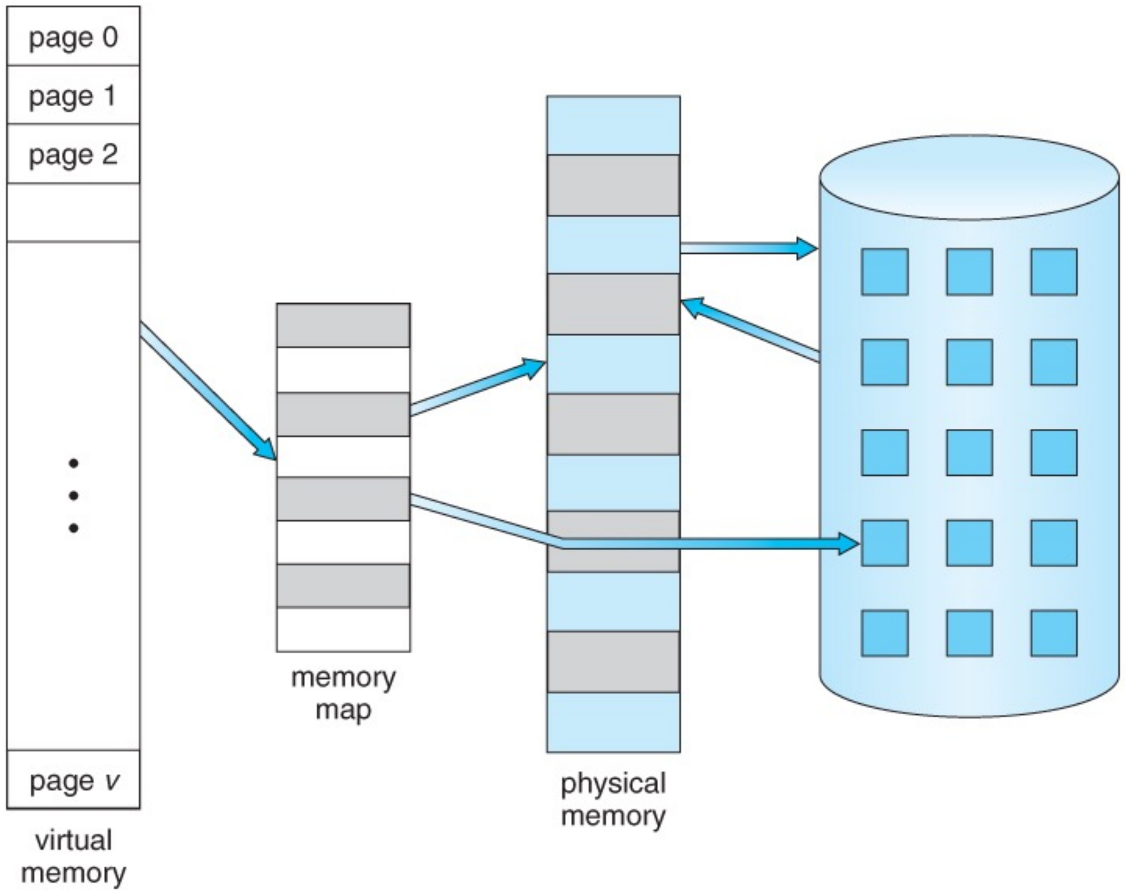
- Code needs to be in memory to execute, **but entire program rarely used**
    - Error code, unusual routines, large data structures
- Entire program code not needed at same time



```
try {

  //code may cause exception
}

catch (Exception e) {

  // code that handles an exception

}

finally {

   // default piece of code

}
```

Benefits of executing partially-load programs

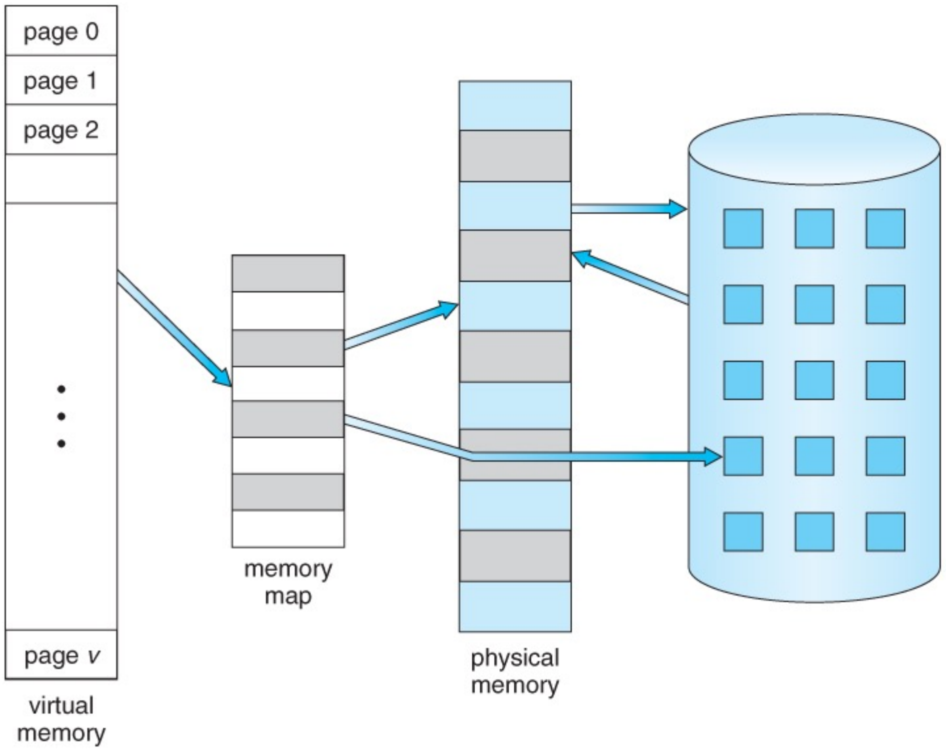
- Program no longer constrained by limits of physical memory



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html

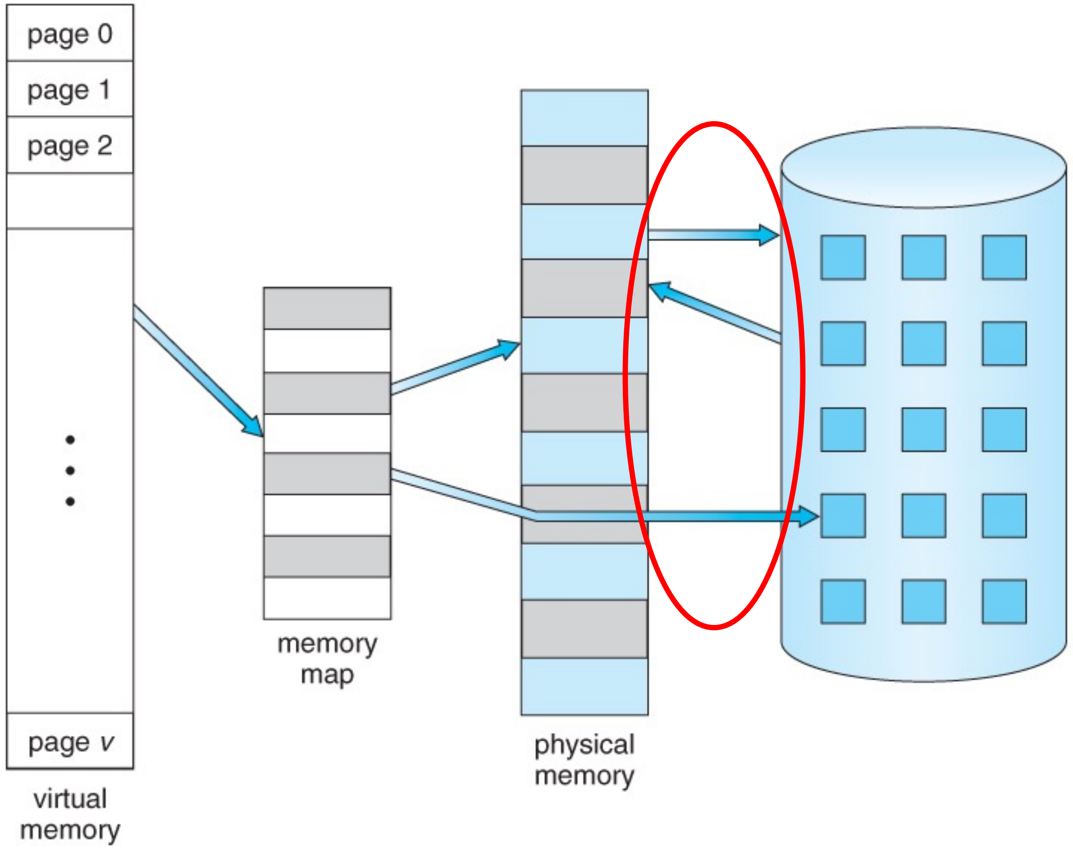
Benefits of executing partially-load programs

- 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.



Benefits of executing partially-load programs

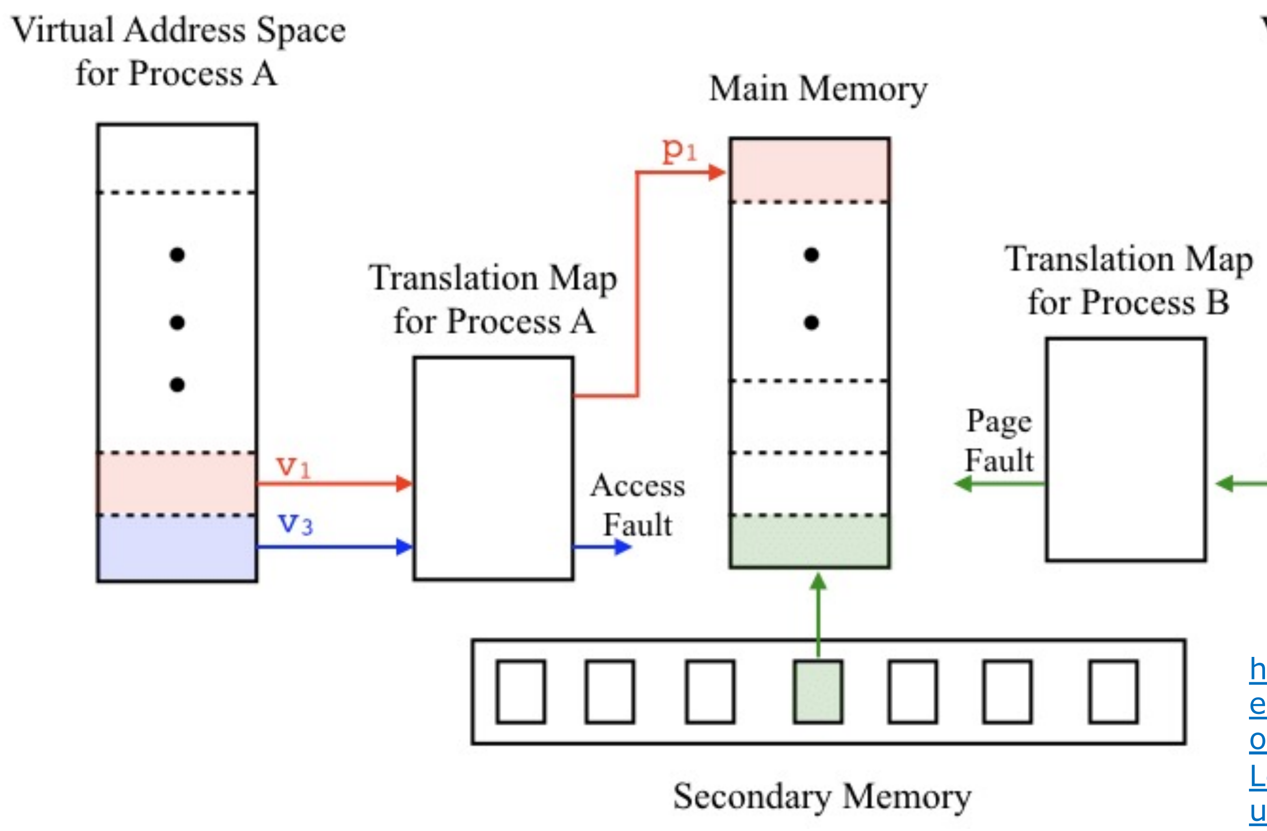
- Less I/O needed to load or swap programs into memory -> **each user program runs faster.**



Virtual memory

separation of user logical memory from physical memory

Only part of the program needs to be in memory for execution

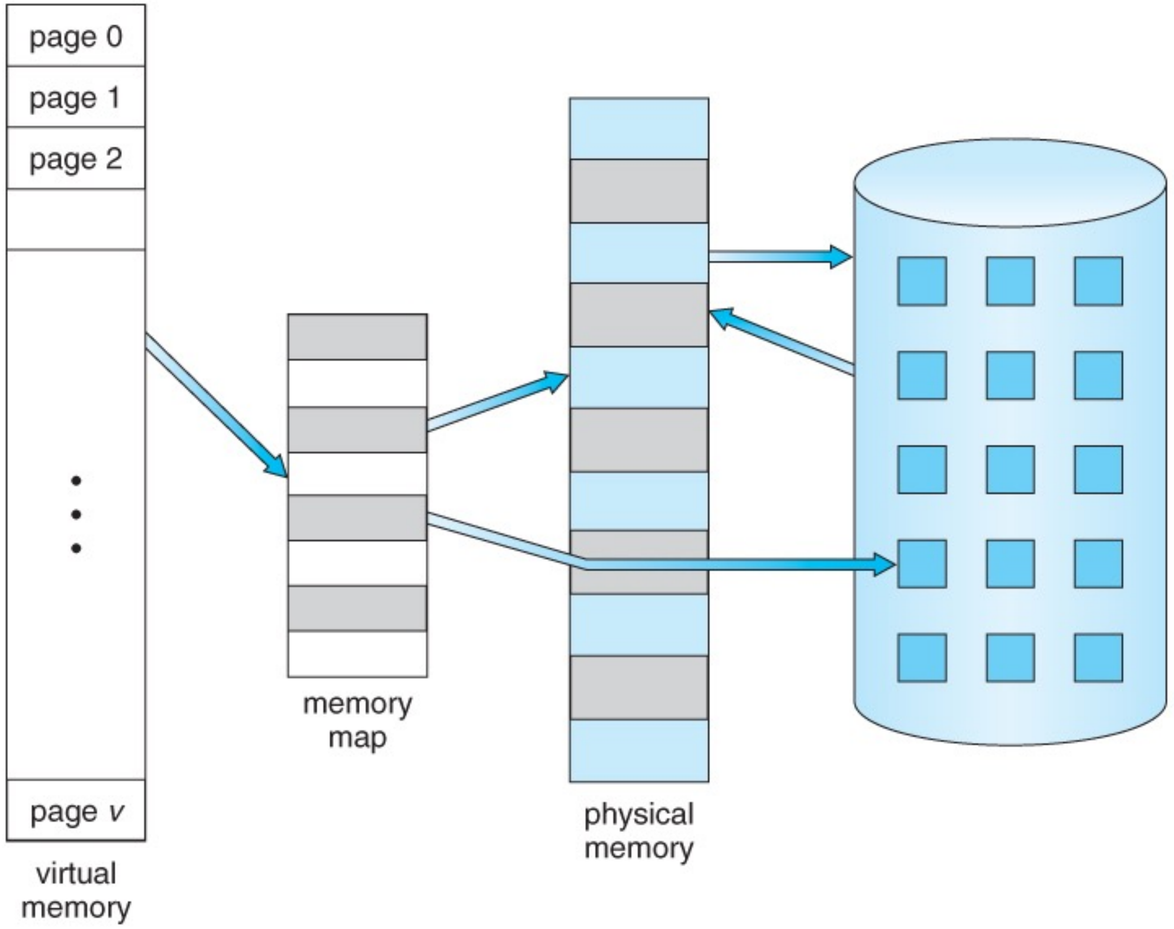


Virtual Address Space for Process A
Main Memory
Virtual Address Space for Process B
p1
Translation Map for Process A
Translation Map for Process B
v1
v3
Access Fault
Page Fault
v10
Secondary Memory

https://www.researchgate.net/publication/334895925_Low-Level_Program_Verification_under_Cached_Address_Translation/figures?lo=1

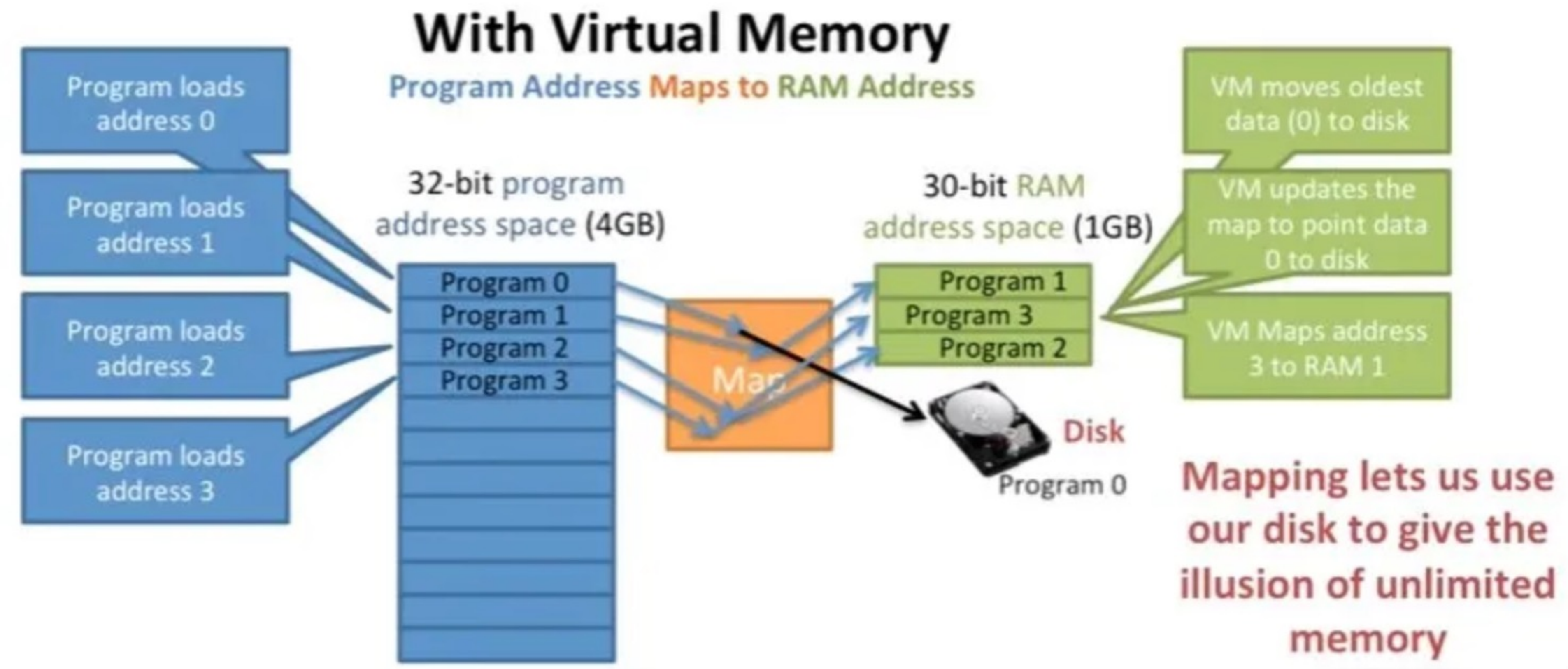
Benefits of Virtual memory

- Logical address space can be much larger than physical address space



Benefits of Virtual memory

- Allows address spaces to be shared by several processes


With Virtual Memory
Program Address Maps to RAM Address
Program loads address 0
Program loads address 1
Program loads address 2
Program loads address 3
32-bit program address space (4GB)
Program 0
Program 1
Program 2
Program 3
Map
30-bit RAM address space (1GB)
Program 1
Program 3
Program 2
Disk
Program 0
VM moves oldest data (0) to disk
VM updates the map to point data 0 to disk
VM Maps address 3 to RAM 1
Mapping lets us use our disk to give the illusion of unlimited memory

Benefits of Virtual memory

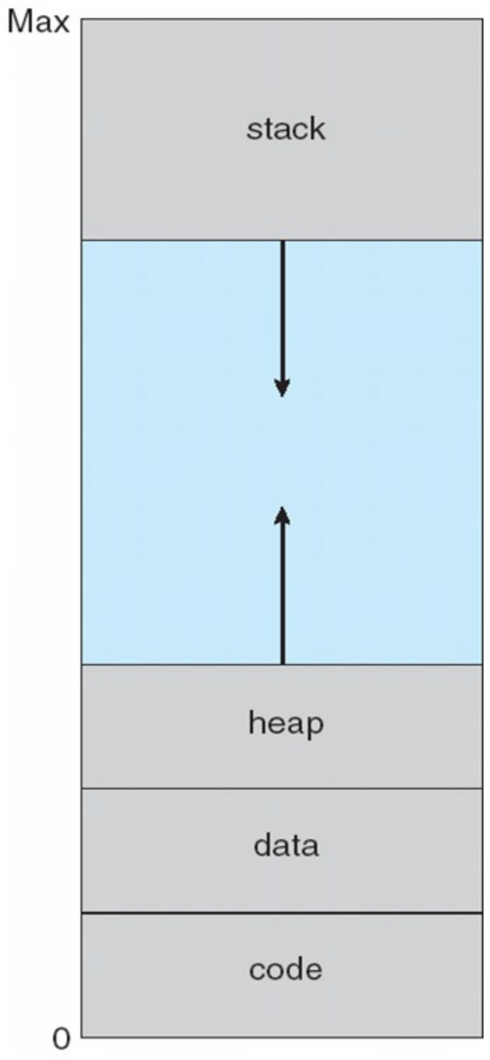
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes

Virtual memory (cont.)

- **Virtual address space:**
  - Logical view of how 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**

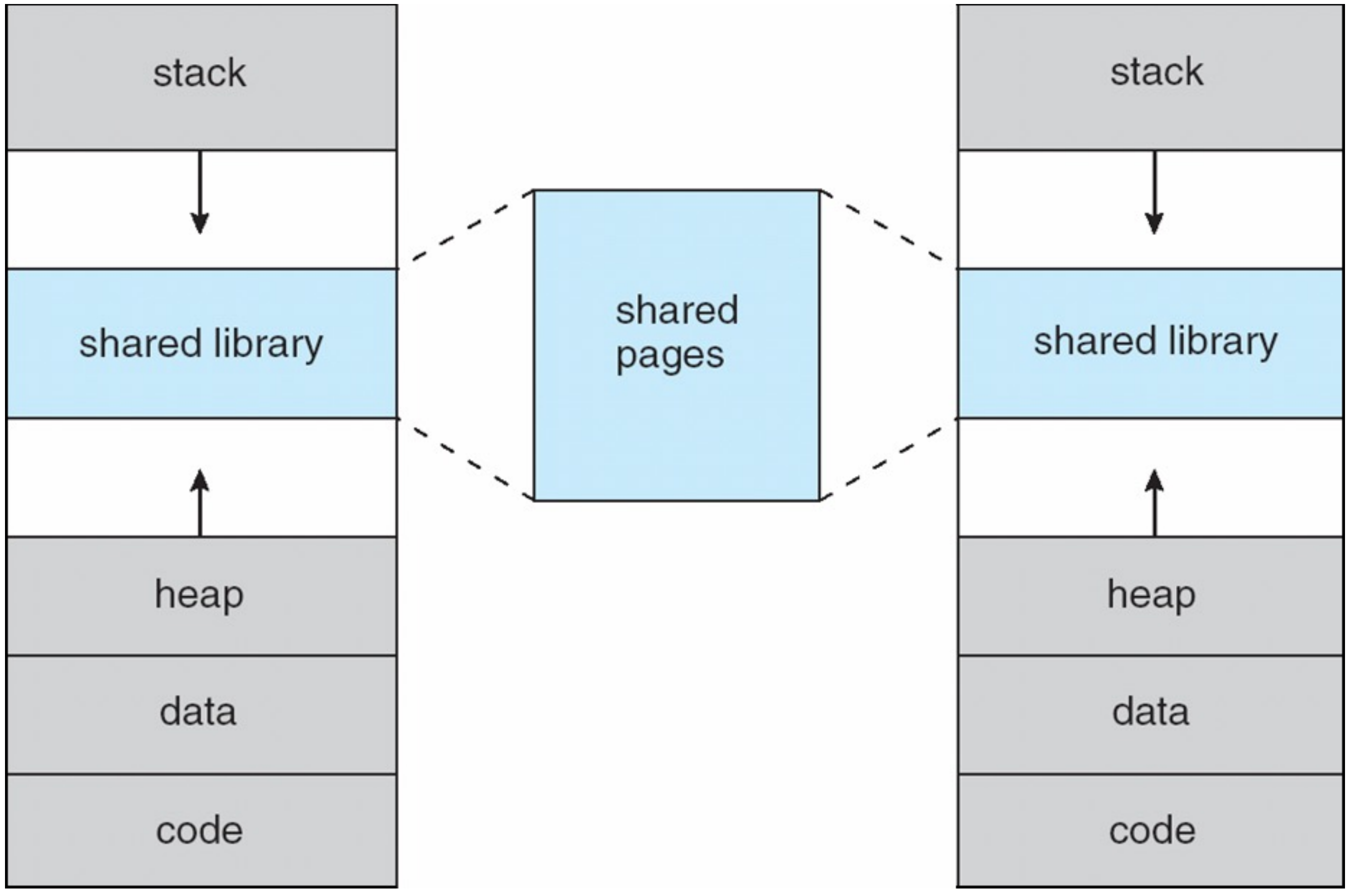
Virtual-address Space

- 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



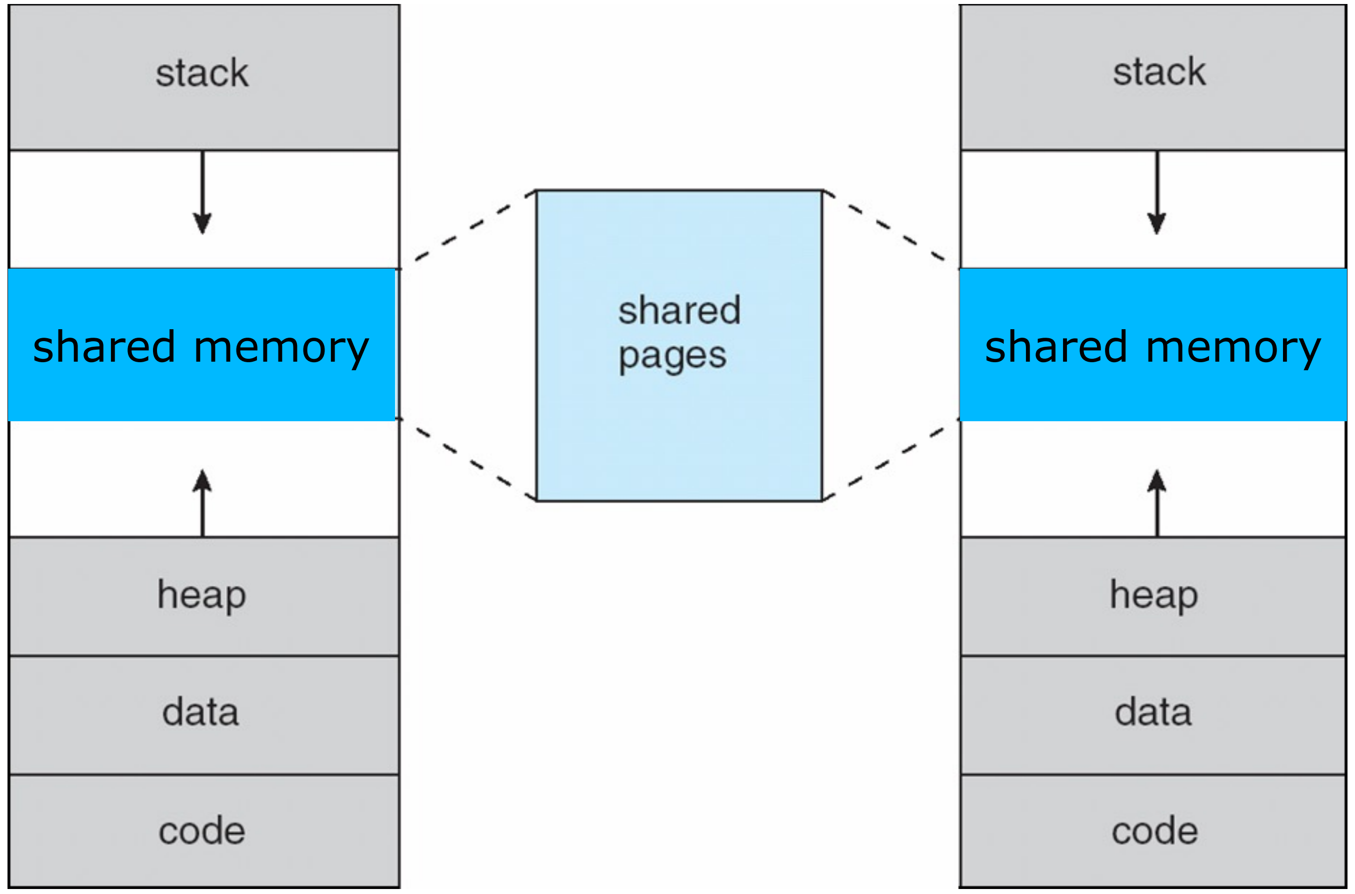
Virtual-address Space

- System libraries shared via mapping into virtual address space



Virtual-address Space

- Shared memory by mapping pages read-write into virtual address space



Virtual-address Space

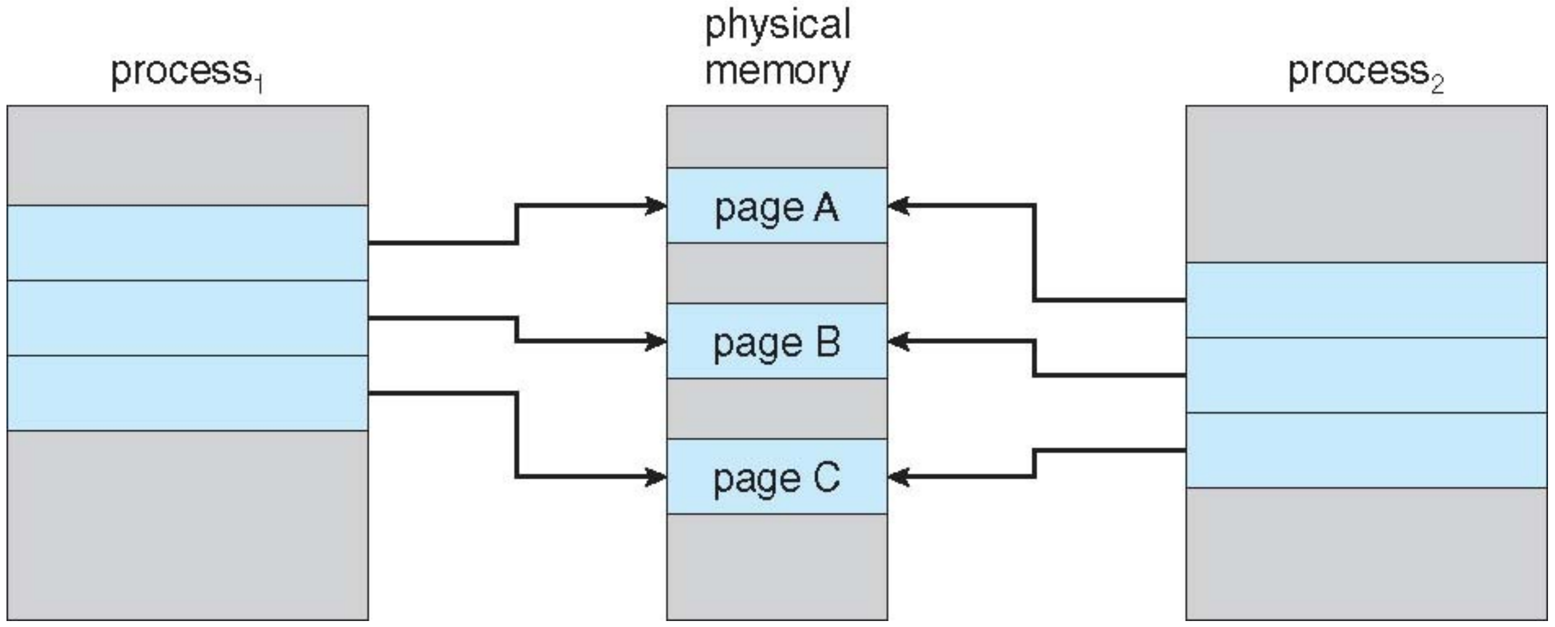
- Pages can be shared during `fork()`, speeding process creation

- How?

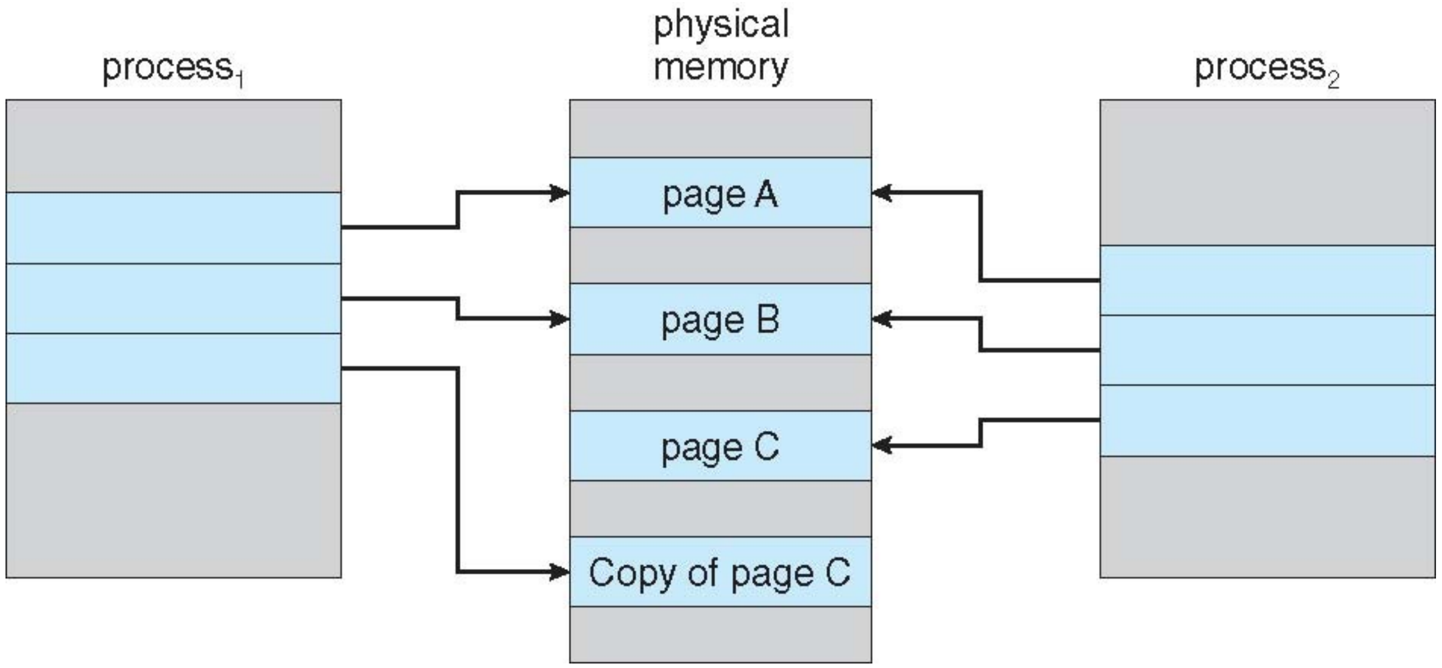
Copy-on-Write

- **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.
- Furthermore, often exec is called immediately after fork.

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Demand Paging

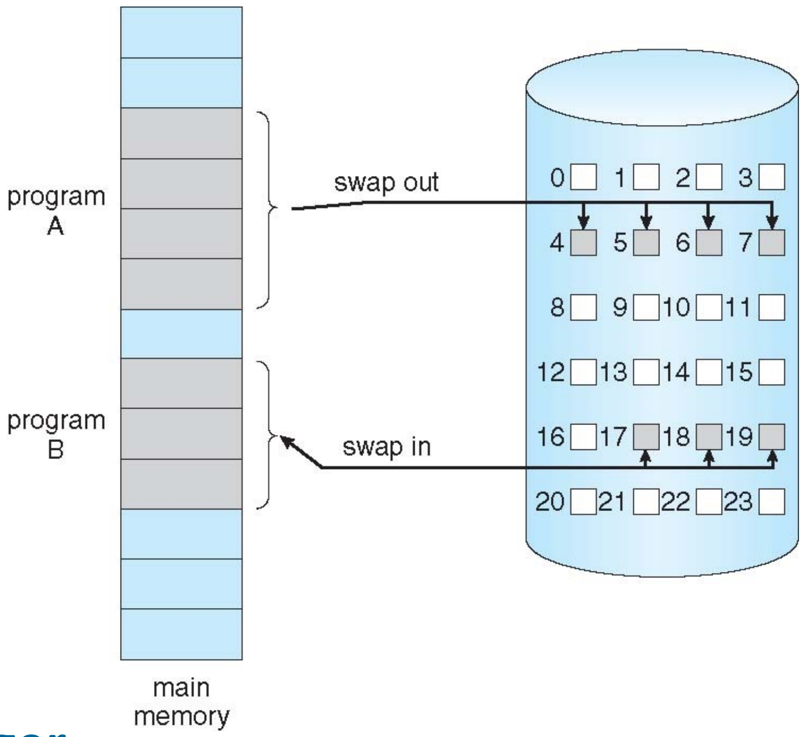
- Could bring entire process into memory at load time
- Or bring **a page into memory only when it is needed**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

Demand Paging (Cont.)

- Similar to paging system **with swapping** (diagram on right)

- 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
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need **new MMU functionality** to implement demand paging

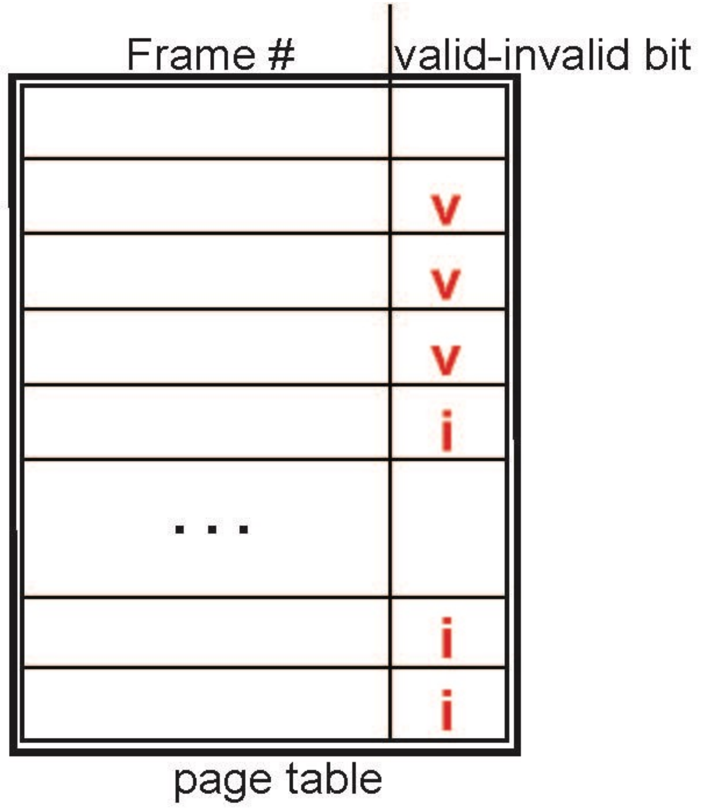
Basic Concepts (Cont.)

- 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

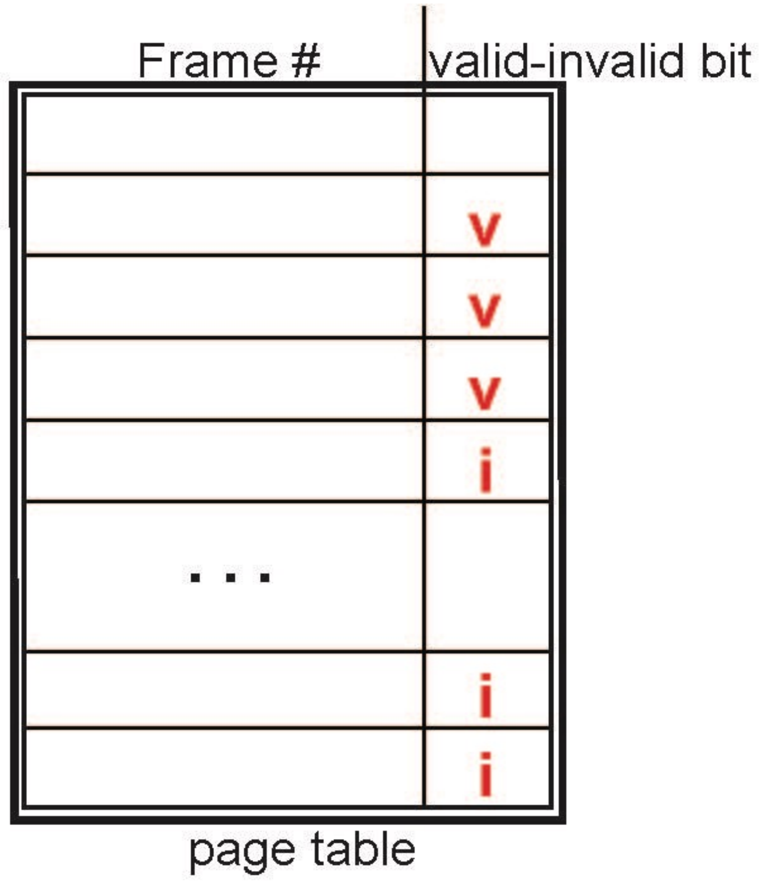
Valid-Invalid Bit

- 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:

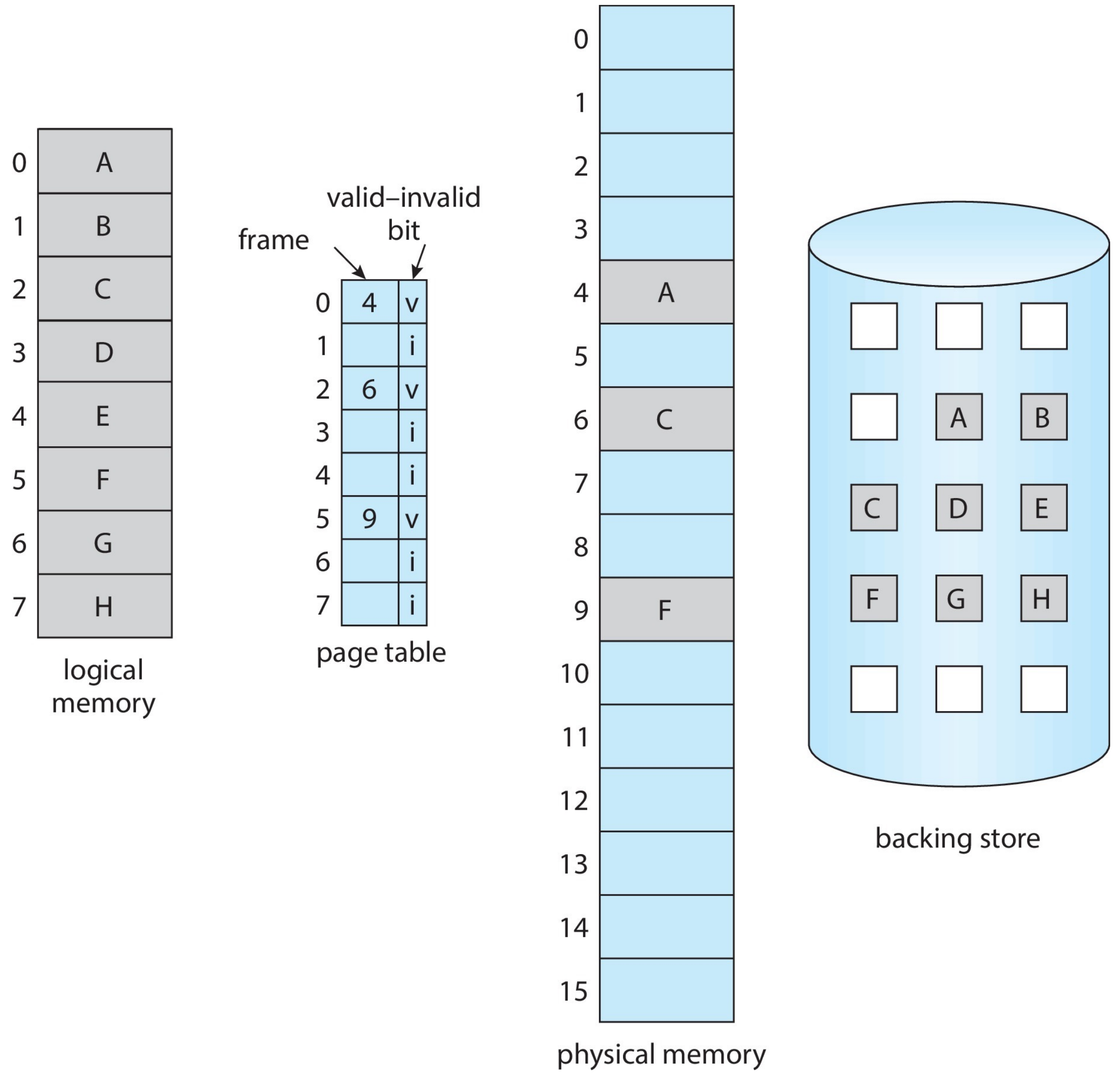


Valid-Invalid Bit

- During MMU address translation, if valid–invalid bit in page table entry is **i** $\Rightarrow$ 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 to decide:
   - Invalid reference ⇒ abort
   - Just not in memory

…

Steps in Handling Page Fault

…

3. Find free frame

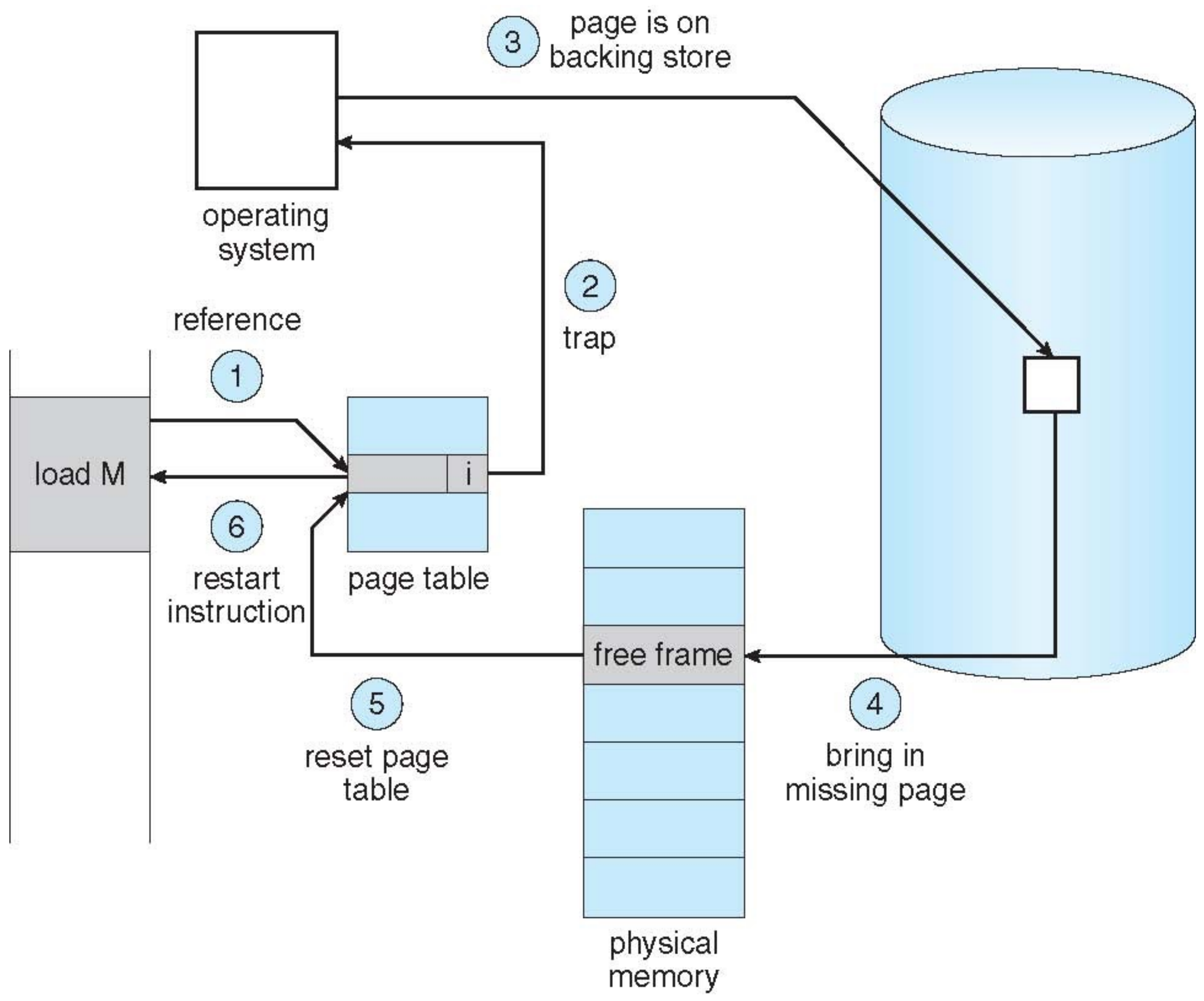
4. Swap page into frame via scheduled disk operation

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

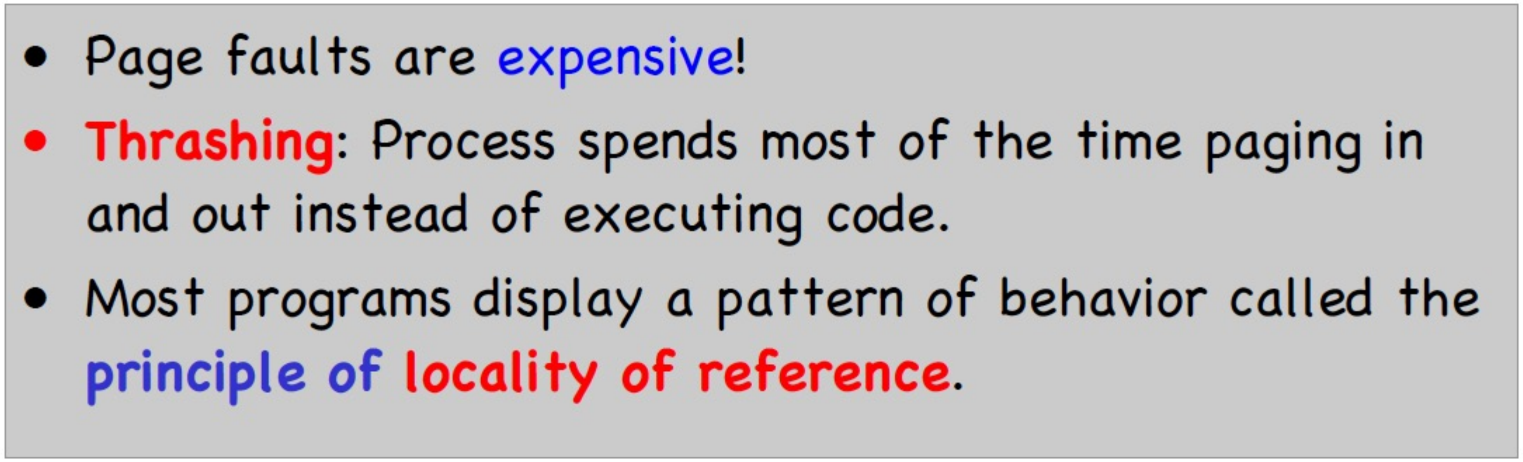
- Extreme case – start process with ***no* pages in memory**
  - OS sets instruction pointer to first instruction of process:

    non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**

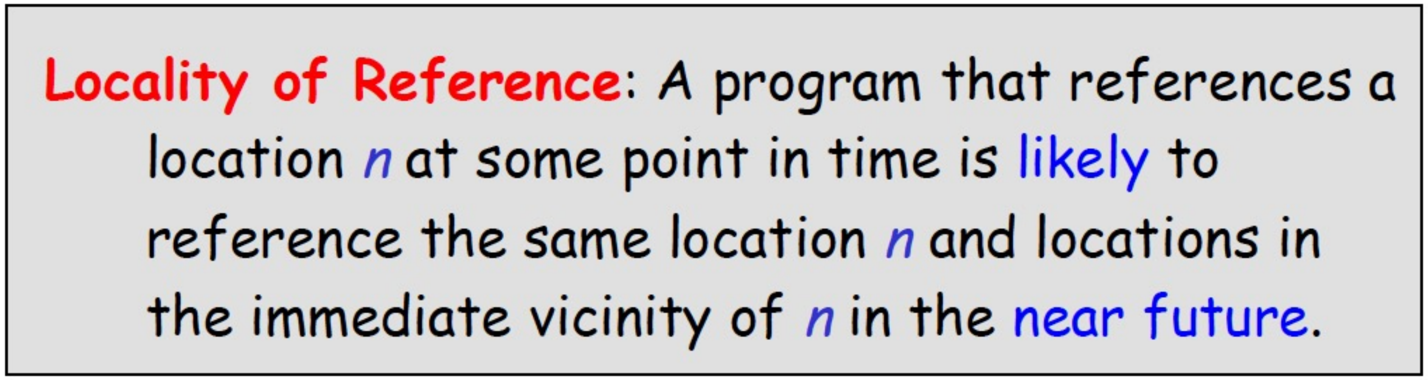
Aspects of Demand Paging (cont.)

- …
- 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**

Locality of reference



- Page faults are expensive!
- Thrashing: Process spends most of the time paging in and out instead of executing code.
- Most programs display a pattern of behavior called the principle of locality of reference.



Locality of Reference: A program that references a location *n* at some point in time is likely to reference the same location *n* and locations in the immediate vicinity of *n* in the near future.

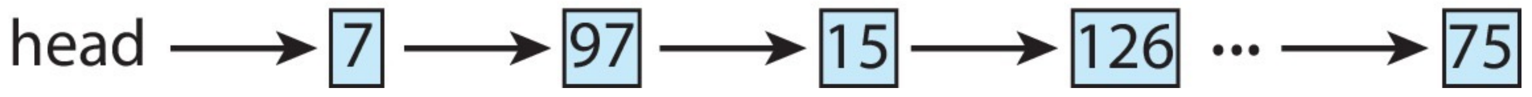
Source: https://people.engr.tamu.edu/bettati/Courses/410/2017A/Slides/virtmemory.pdf

Aspects of Demand Paging

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

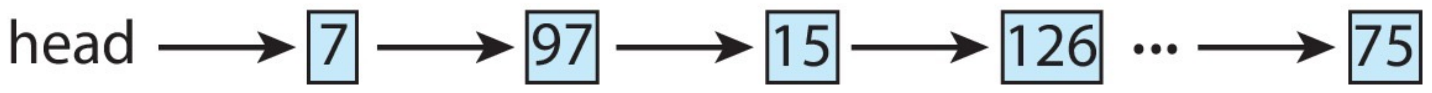
Free-Frame List

- 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.



Free-Frame List (cont.)

- ..



- 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

….

Stages in Demand Paging – Worse Case

….

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
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)

…

Stages in Demand Paging (cont.)

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

Performance of Demand Paging (cont.)

- …
- 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

  + 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 + 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!!

Demand Paging Example (cont.)

- ....
- If want performance degradation < 10 percent
  - $220 > 200 + 7{,}999{,}800 \times p$

    $20 > 7{,}999{,}800 \times p$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses