CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2016 Lecture 31

Virtual Memory

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Questions from last time

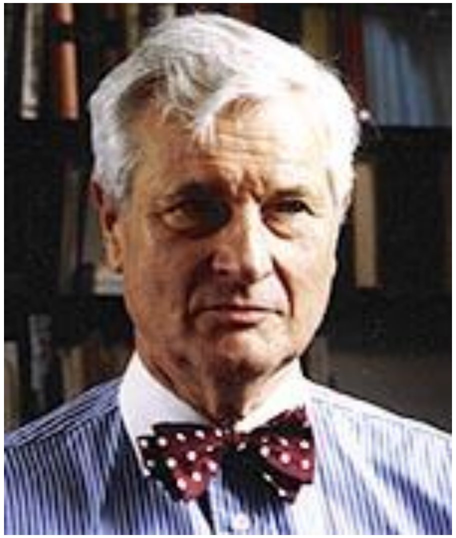
- Where are page tables for the processes stored? Memory
- Are large processes slower?
- How are page tables started? Allocation by OS
- Why multi-level page tables?
- Does searching a TLB with n entries takes O(n) time? In parallel, hardware implemented associative memory (content addressable). Expensive
- Inverted page table: one sequential entry for each frame, stores PID, page no. Used in PowerPC. Linus Torvalds comment.
- PA5: see Help Session recording
- Best fit, worst fit, first fit comparison statistical distributions. Bin packing.

Virtual Memory: Objectives

"You say we went out and I never called? I can't remember. My virtual memory must be low!"

- A virtual memory system
- Demand paging, page-replacement algorithms, allocation of page frames to processes
- Threshing, the working-set model
- Memory-mapped files and shared memory and
- Kernel memory allocation

Fritz-Rudolf Güntsch



Fritz-Rudolf Güntsch (1925-2012) at the Technische Universität Berlin in 1956 in his doctoral thesis, *Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation*.

First used in Atlas, Manchester, 1962

PCs: Windows 95

Chapter 9: 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

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
  - Program no longer constrained by limits of physical memory
  - Each program uses 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

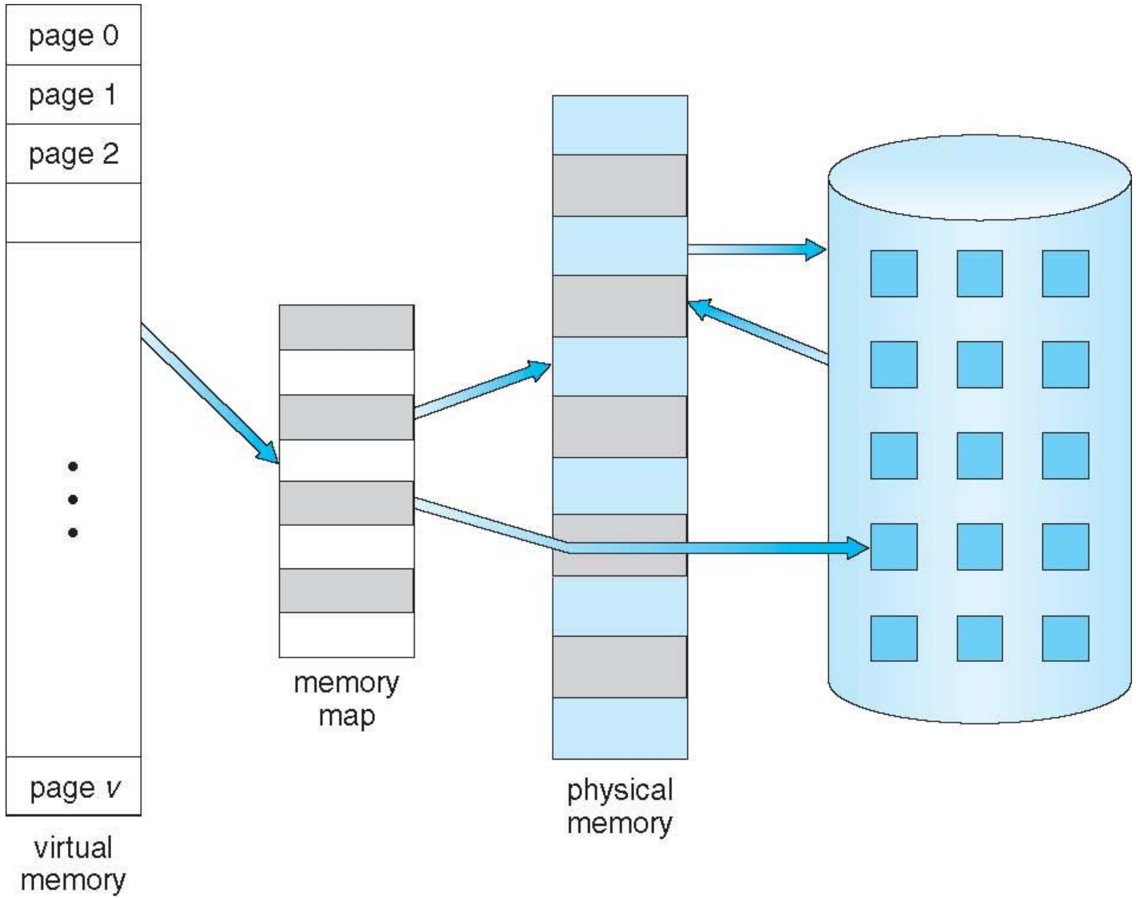
Background (Cont.)

- **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.)

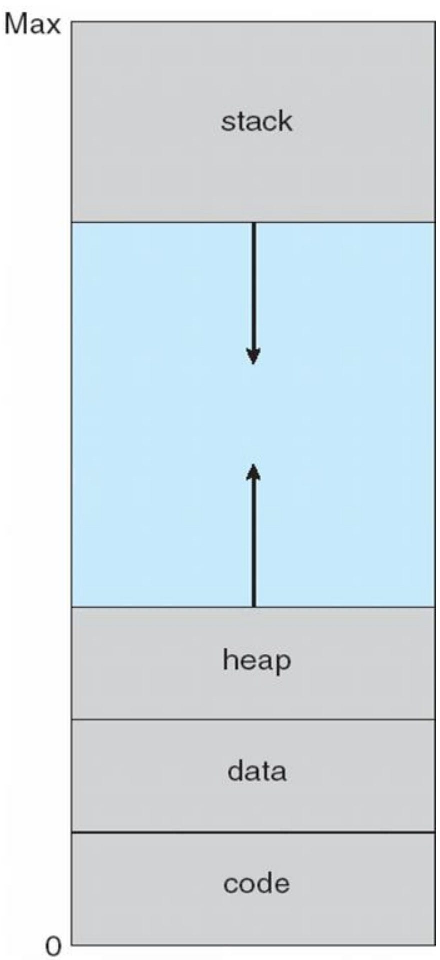
- **Virtual address space** – logical view of how process views memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



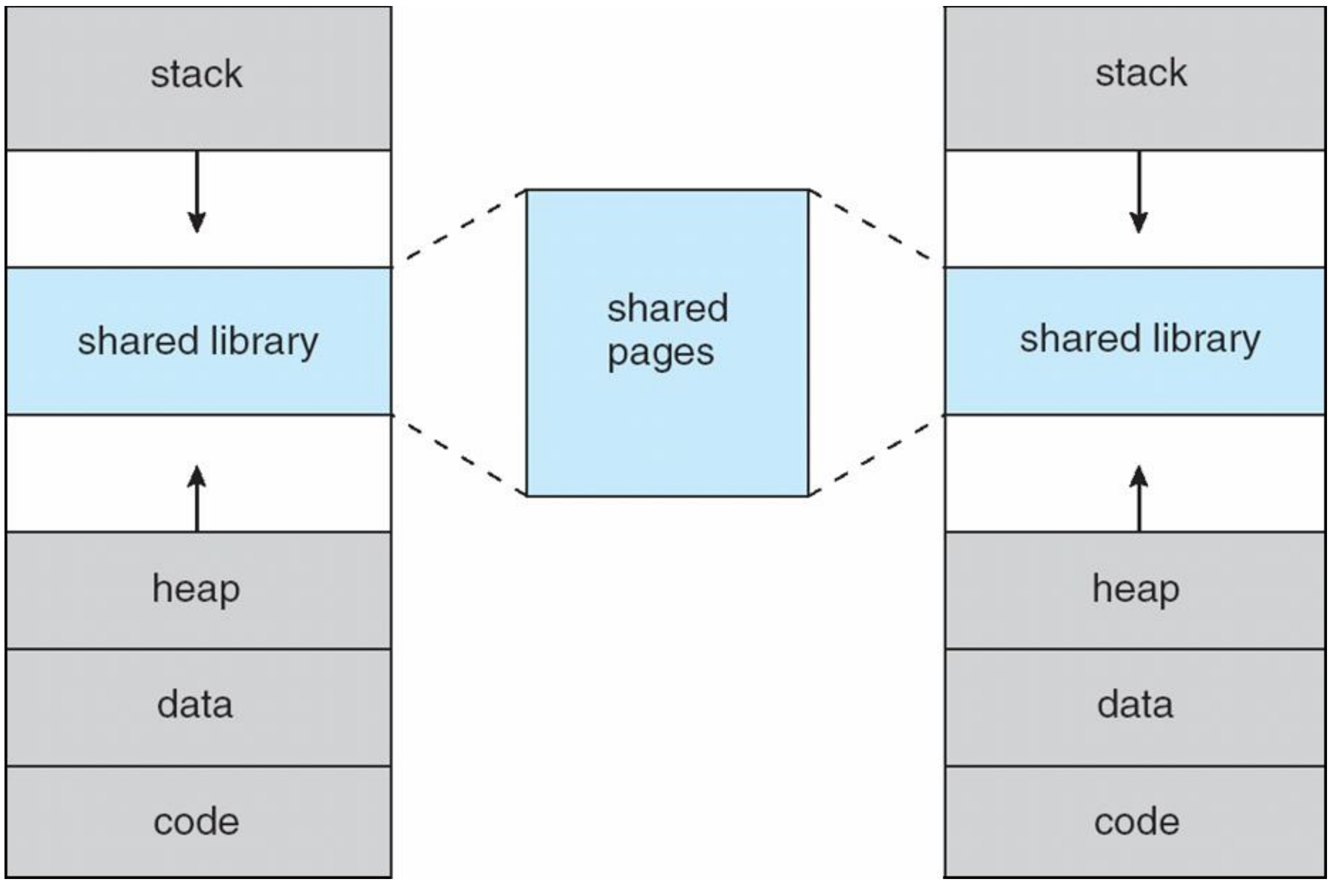
Virtual-address Space: advantages

- 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



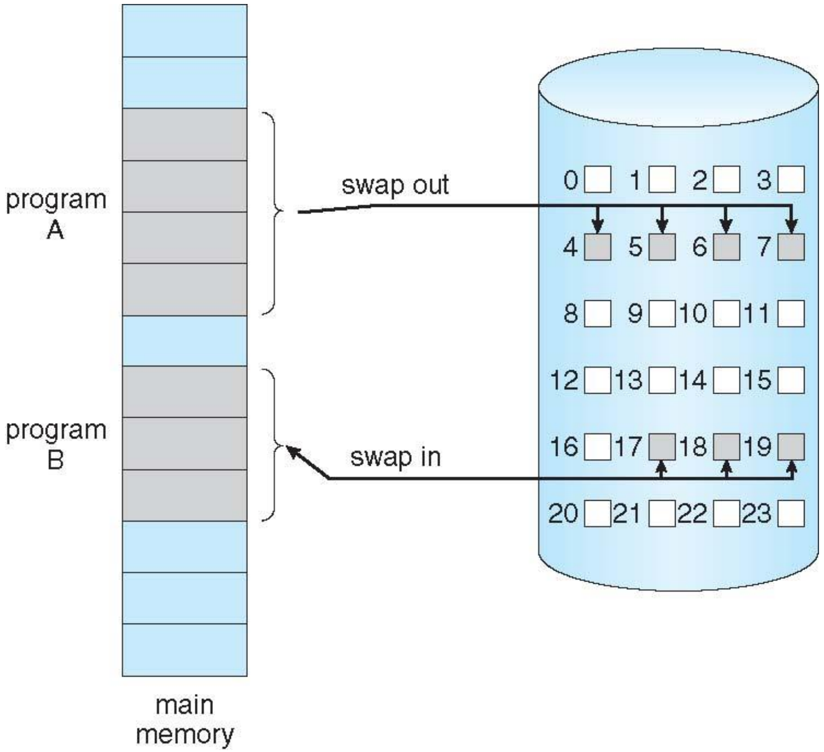
Colorado State University

Shared Library Using Virtual Memory



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 users
- 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**



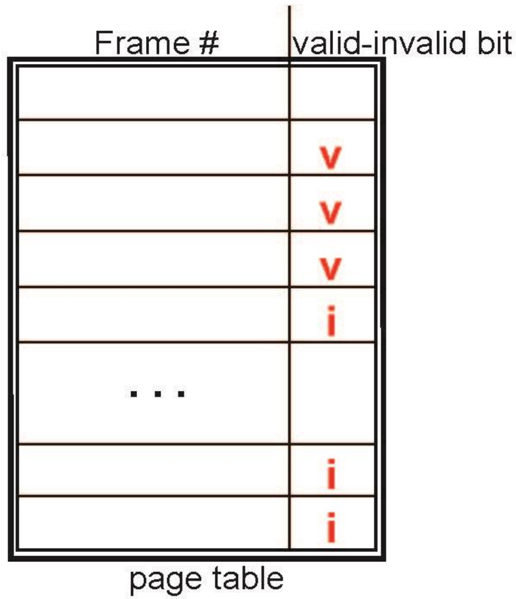
Colorado State University

Demand paging: Basic Concepts

- Demand paging: pager brings in only those pages into memory what are needed
- 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

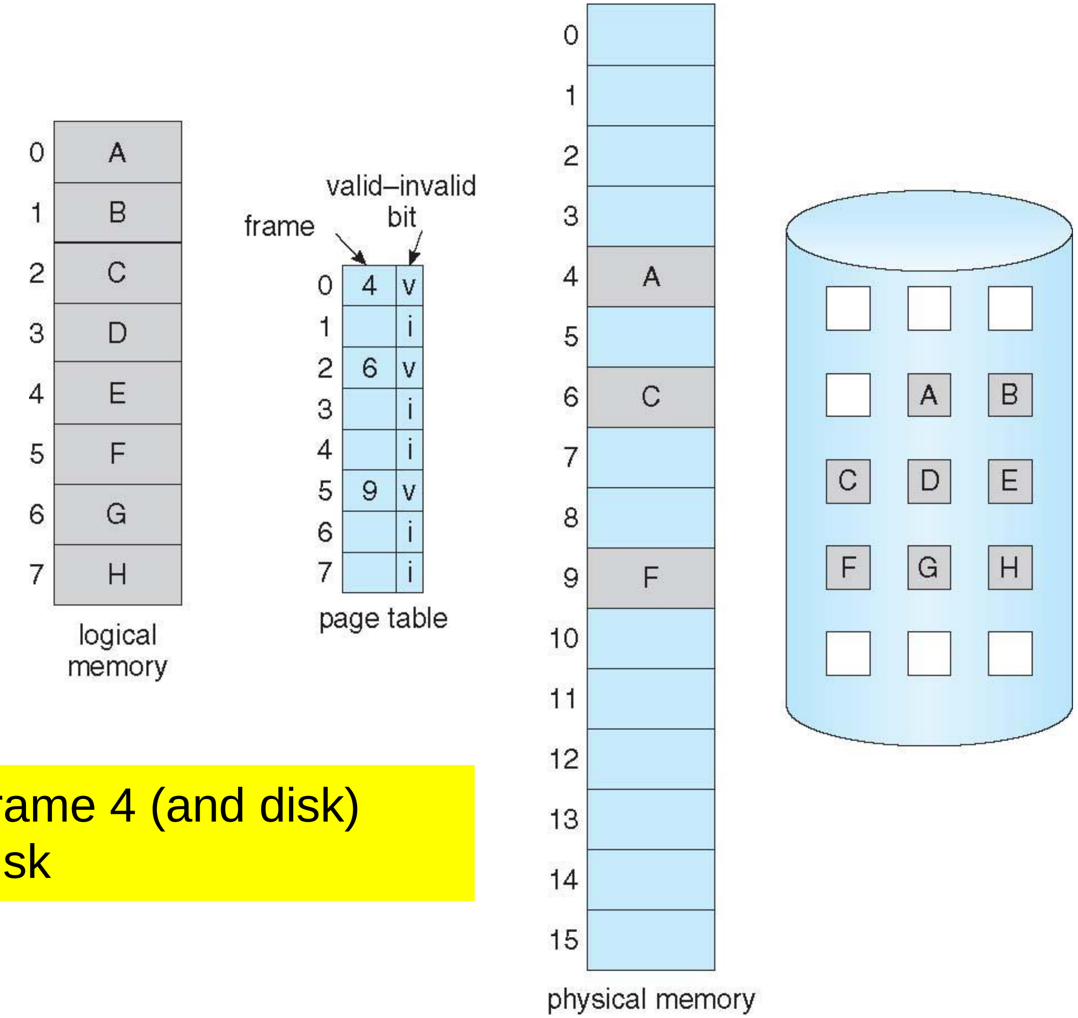
- With each page table entry a valid–invalid bit is associated (**v** $\Rightarrow$ in-memory – **memory resident**, **i** $\Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



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

Colorado State University

Page Table When Some Pages Are Not in Main Memory



Page 0 in Frame 4 (and disk)
Page 1 in Disk

Colorado State University

Page Fault

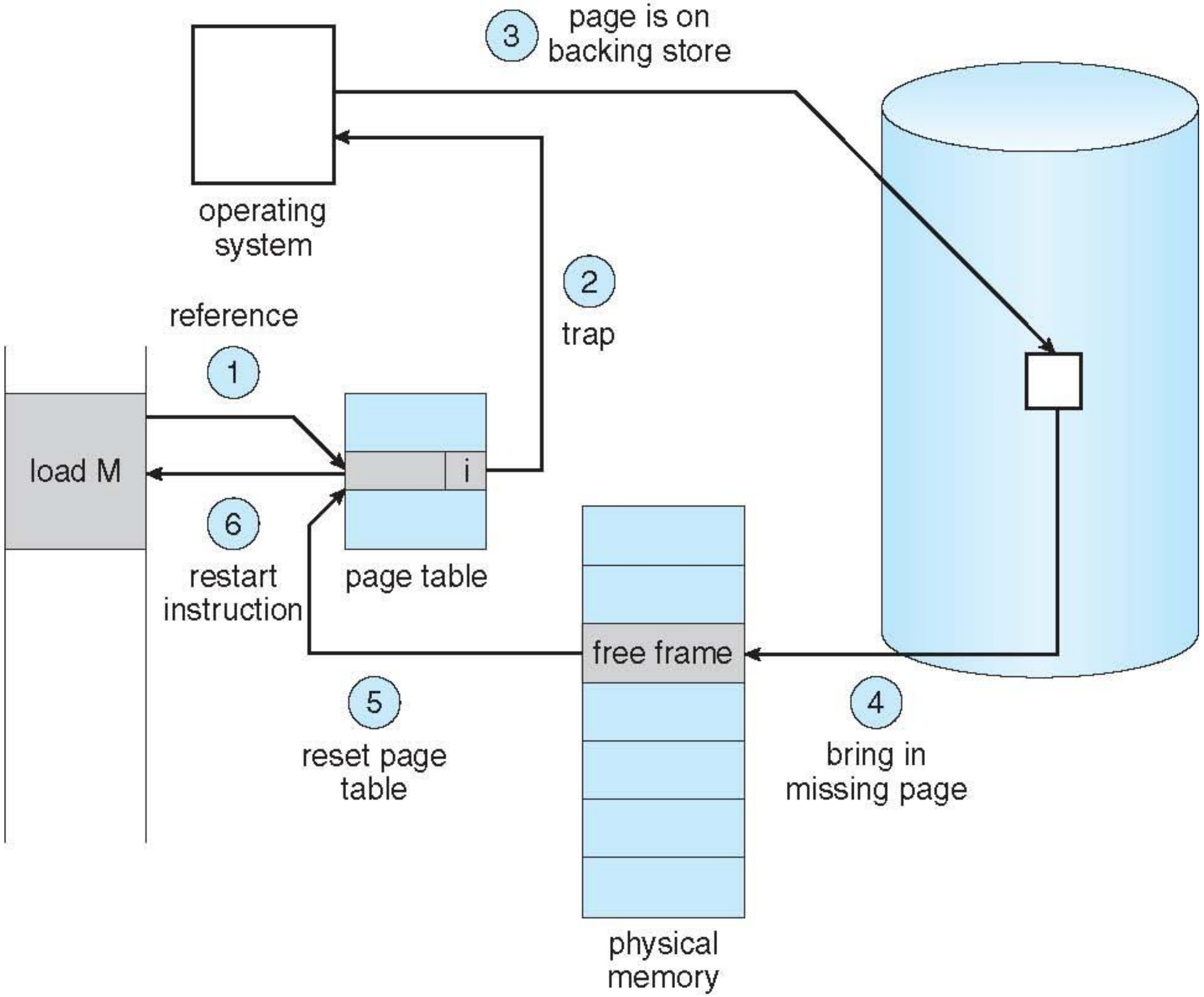
- If there is a reference to a page, first reference to that page will trap to operating system: Page fault

Page fault

1. Operating system looks at a table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory, but in backing storage, ->2
2. Find free frame
3. Get 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

Page fault: context switch because disk access is needed

Steps in Handling a Page Fault



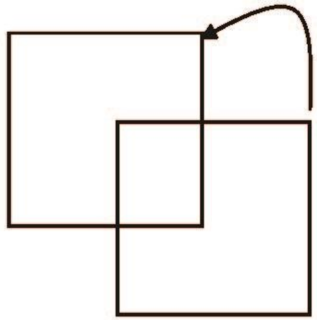
Aspects of Demand Paging

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

Instruction Restart: Complications

- Consider an instruction that could access several different locations
  - block move in some ISAs:

    Either block straddles page-boundary
  - Restart the whole operation?
    - What if source and destination overlap?
    - Source overwritten – instruction cannot restar
  - One solution: obtain both ends of the block
    - If a page fault were to happen: it will at this point. Nothing has been partially modified.
    - After fault servicing, block transfer completes
  - Use temporary registers
    - Track overwritten values



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
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
- 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)

Hopefully p <<1

EAT = $(1 – p)$ x memory access time

+ p (page fault overhead

+ swap page out + swap page in)

Colorado State University

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 nanosec.
  = 200 + p x 7,999,800

Linear with page fault rate

- 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

Issues: Allocation of physical memory to I/O and programs

- Memory used for holding **program** pages
- **I/O buffers** also consume a big chunk of memory
- Solutions:
  - Fixed percentage set aside for I/O buffers
  - Processes and the I/O subsystem compete

Demand paging and the limits of logical memory

- Without demand paging
  - All pages of process **must be** in physical memory
  - Logical memory **limited** to size of physical memory
- With demand paging
  - All pages of process **need not be** in physical memory
  - Size of logical address space is **no longer constrained** by physical memory
- Example
  - 40 pages of physical memory
  - 6 processes each of which is 10 pages in size
    - Each process only needs 5 pages as of now
  - Run 6 processes with 10 pages to spare

Higher degree of multiprogramming

Colorado State University

Coping with over-allocation of memory

Example

- Physical memory = 40 pages
- 6 processes each of which is of size 10 pages
  - But are using 5 pages each as of now
- What happens if each process needs all 10 pages?
  - 60 physical frames needed
- **Terminate** a user process
  - But paging should be transparent to the user ⊗
- **Swap out** a process
  - Reduces the degree of multiprogramming ⊗
- **Page replacement:** selected pages. Policy?