

OPERATING SYSTEMS

Memory Management

Chandravva Hebbi

Department of Computer Science

OPERATING SYSTEMS

Virtual Memory

Chandravva Hebbi

Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course



- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:
 1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
 2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
 3. Some presentation transcripts from A. Frank – P. Weisberg
 4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

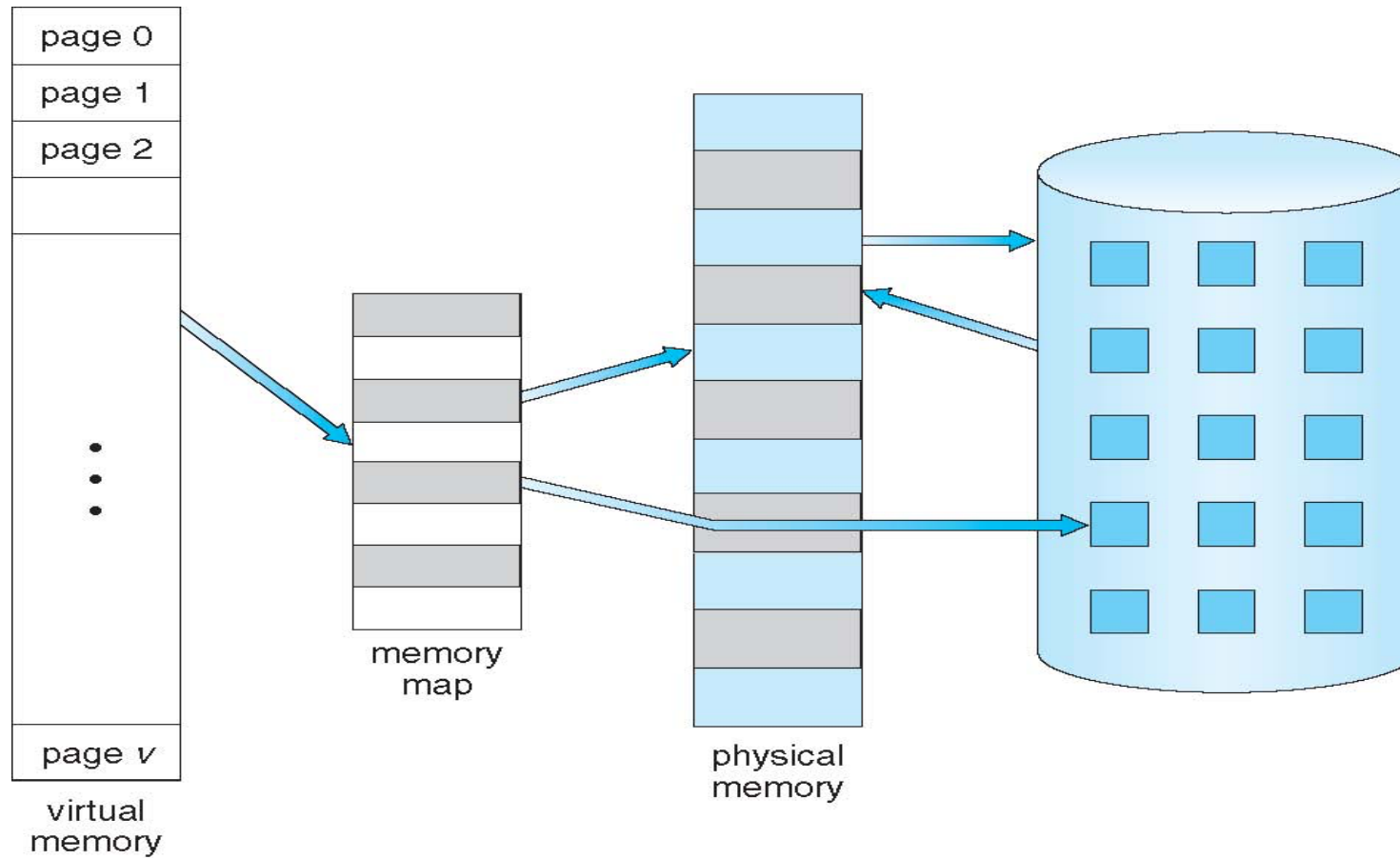
- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory

- 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

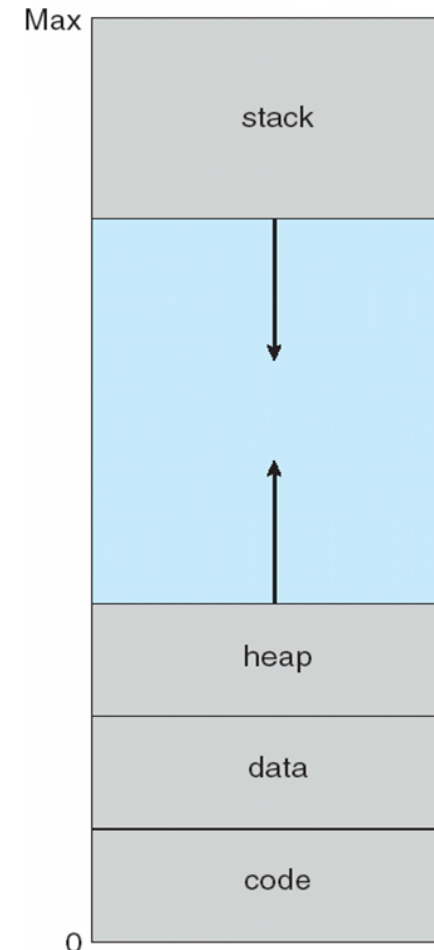
- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes

- **Virtual 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 Memory That is Larger Than Physical Memory

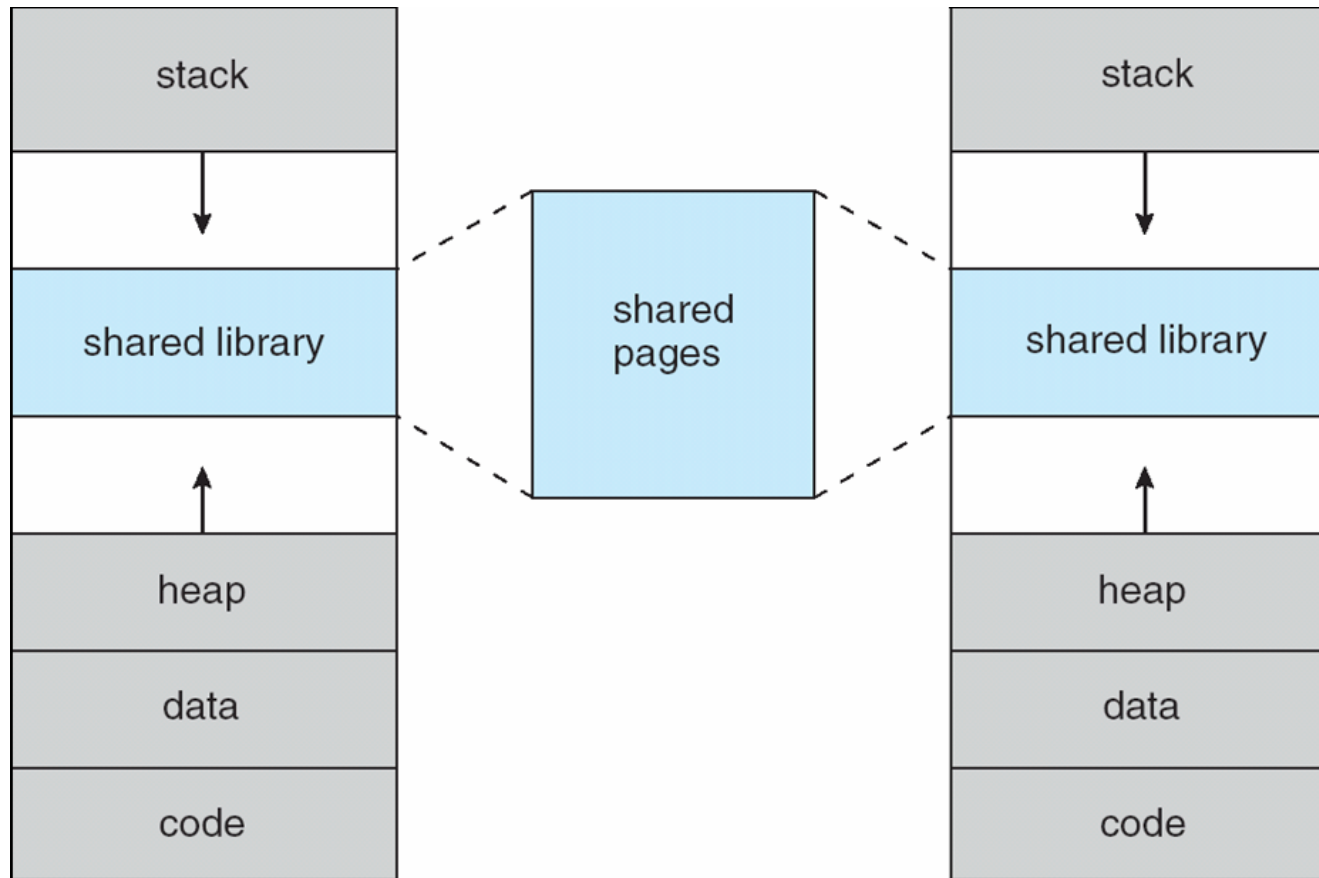


- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
 - Maximizes address space use
 - Unused address space between the two is hole
 - ▶ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address 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



OPERATING SYSTEMS

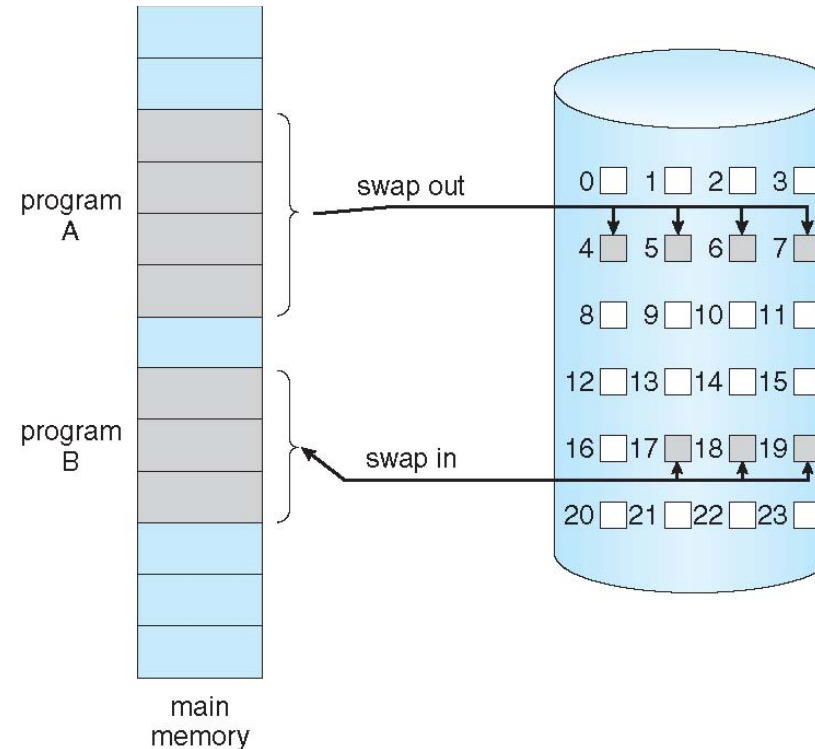
Shared Library Using Virtual Memory



OPERATING SYSTEMS

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)
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**



- 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
- 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

- 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:

Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

page table

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

OPERATING SYSTEMS

Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

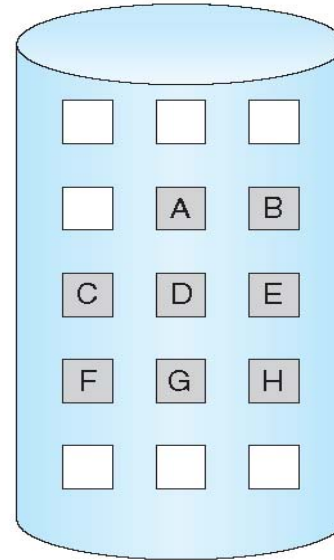
logical
memory

valid-invalid bit		
frame		bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

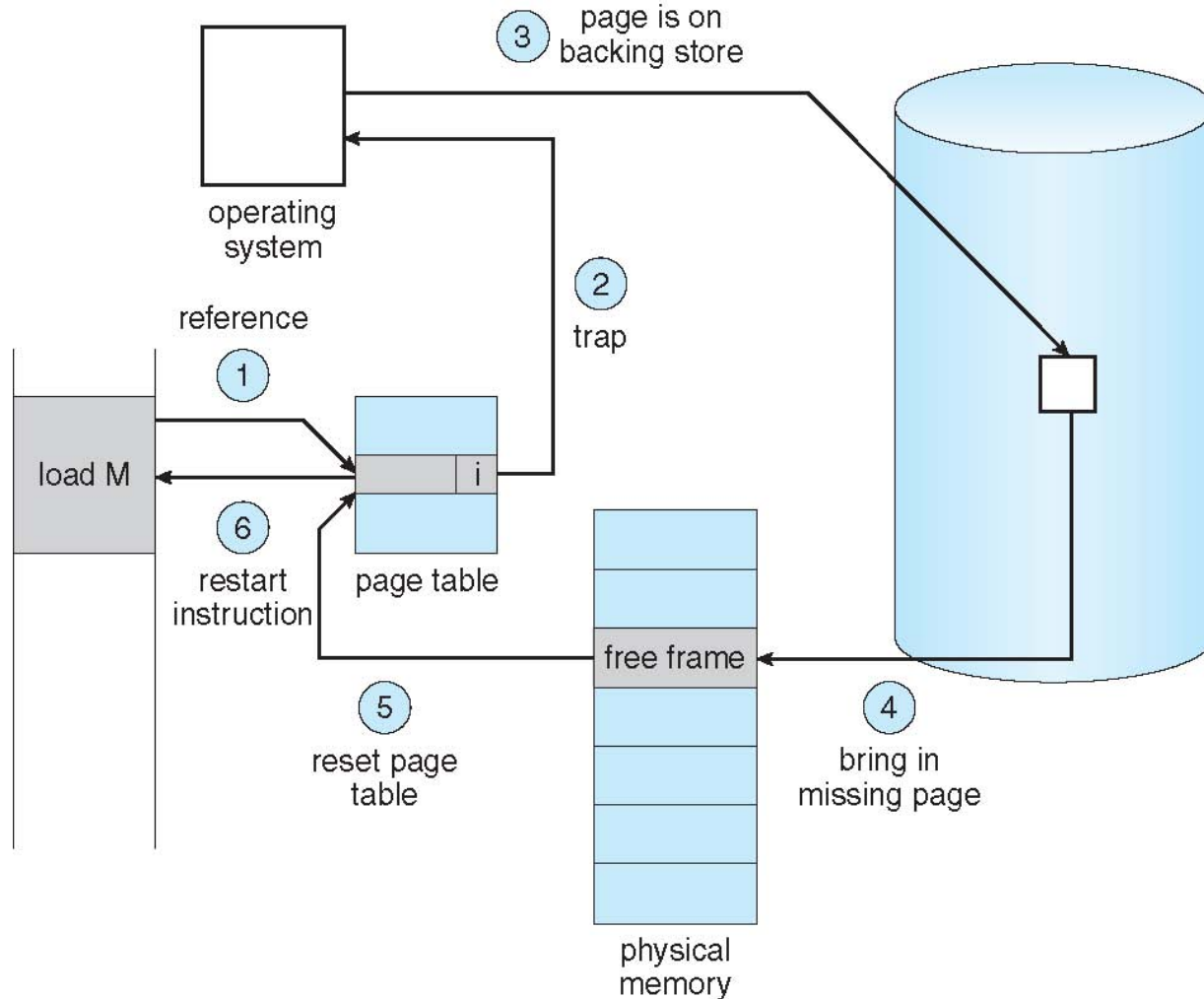
page table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

physical memory



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 \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



- 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** - Process migrates from one locality (i.e., a set of pages that are actively used together) to another
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with **swap space**)
 - Instruction restart

- Consider an instruction that could access several different locations
(Ex: move some bytes from one location to another possibly overlapping location)
 - Source and destination blocks overlap i.e straddle a page boundary
 - ▶ Page fault might occur after the move is partially done
 - ▶ Source block may have been modified so we cannot simply restart the instruction
 - In one solution, the microcode computes and attempts to access both ends of both blocks.
 - Page fault can occur before anything is modified
 - The other solution uses temporary registers to hold the values of overwritten locations.
 - If Page fault occurs, all the old values are written back into memory before the trap occurs

■ 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

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

■ 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 $0 \leq p \leq 1$ (p is the probability of a page fault)

- if $p = 0$ no page faults
- if $p = 1$, every reference is a fault

■ Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 - p) \times \text{memory access} \\ &\quad + p \times \text{page-fault service time} \end{aligned}$$

OPERATING SYSTEMS

Demand Paging Example



- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$
 $= (1 - p) \times 200 + p \times 8,000,000$
 $= 200 + p \times 7,999,800$
- EAT is directly proportional to page-fault rate
- If one access out of 1,000 causes a page fault i.e $p = 0.001$, then
EAT = 8200 ns or 8.2 microseconds.
This is a slowdown by a factor of 40!! (i.e. $8200/200$)
- If we 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

- Swap space I/O faster than file system I/O even if on the same device
 - Swap space is allocated in larger blocks, 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) from applications
 - ▶ if memory becomes constrained and demand page such data from file system later if needed



THANK YOU

Chandravva Hebi

Department of Computer Science Engineering

chandravvahebbi@pes.edu