Address Translation

OS/Distributed Systems Companies at the Job Fair

- Addepar
- Adobe
- Amazon
- Arista
- Clustrix
- Cray
- Dropbox
- eBay
- EMC Isilon
- Extrahop
- F5
- Facebook
- Google
- Hulu
- Intel
- Intermec
- Lawrence Livermore Labs
- Microsoft
- NetApp
- OpenMarket
- Qualcomm
- Twitter
- VMware
- Yahoo!

Last Time

- Multiprocessor scheduling
  - Affinity scheduling
  - Per-processor data structures to avoid locking
  - Space sharing vs. time sharing
- Queueing Theory
  - Predict change in response time due to changes in CPU speed, request rate, disk speed, application complexity

Overload Management

- What if arrivals occur faster than service can handle them
  - If do nothing, response time will become infinite
- Turn users away?
  - Which ones? Average response time is best if turn away users that have the highest service demand
- Degrade service?
  - Compute result with fewer resources
  - Example: CNN static front page on 9/11
  - Counterexample: highway congestion

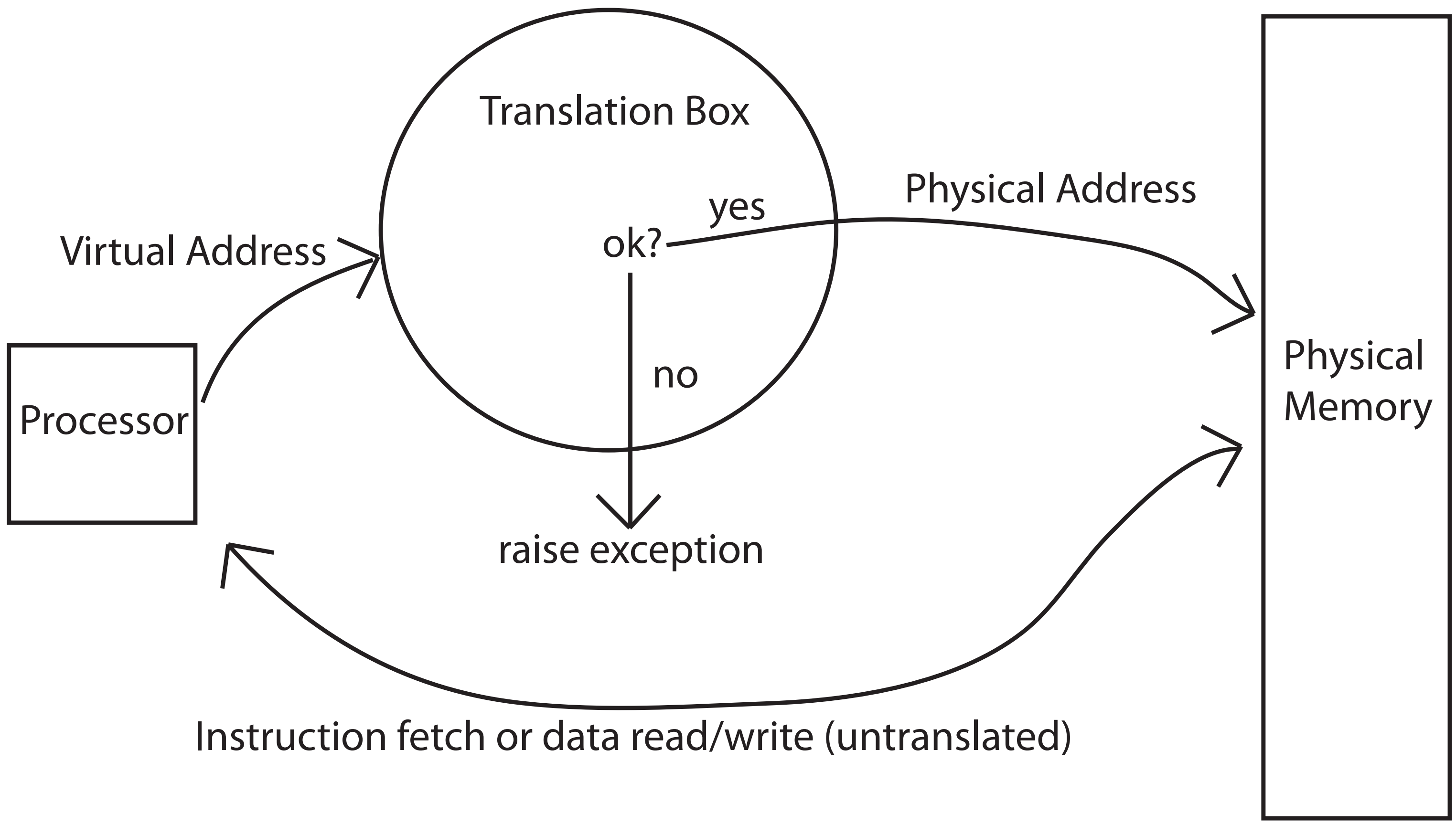
Why Do Metro Buses Cluster?

- Suppose two Metro buses start 15 minutes apart
  - Why might they arrive at the same time?

Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Base and bound
  - Segmentation
  - Paging
- Efficient Address Translation
  - Translation Lookaside Buffers

Address Translation Concept



Address Translation Goals

- Memory protection
- Memory sharing
- Flexible memory placement
- Sparse addresses
- Runtime lookup efficiency
- Compact translation tables
- Portability

Address Translation

- What can you do if you can (selectively) gain control whenever a program reads or writes a particular memory location?
  - With hardware support
  - With compiler-level support
- Memory management is one of the most complex parts of the OS
  - Serves many different purposes

Address Translation Uses

- Process isolation
  - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand

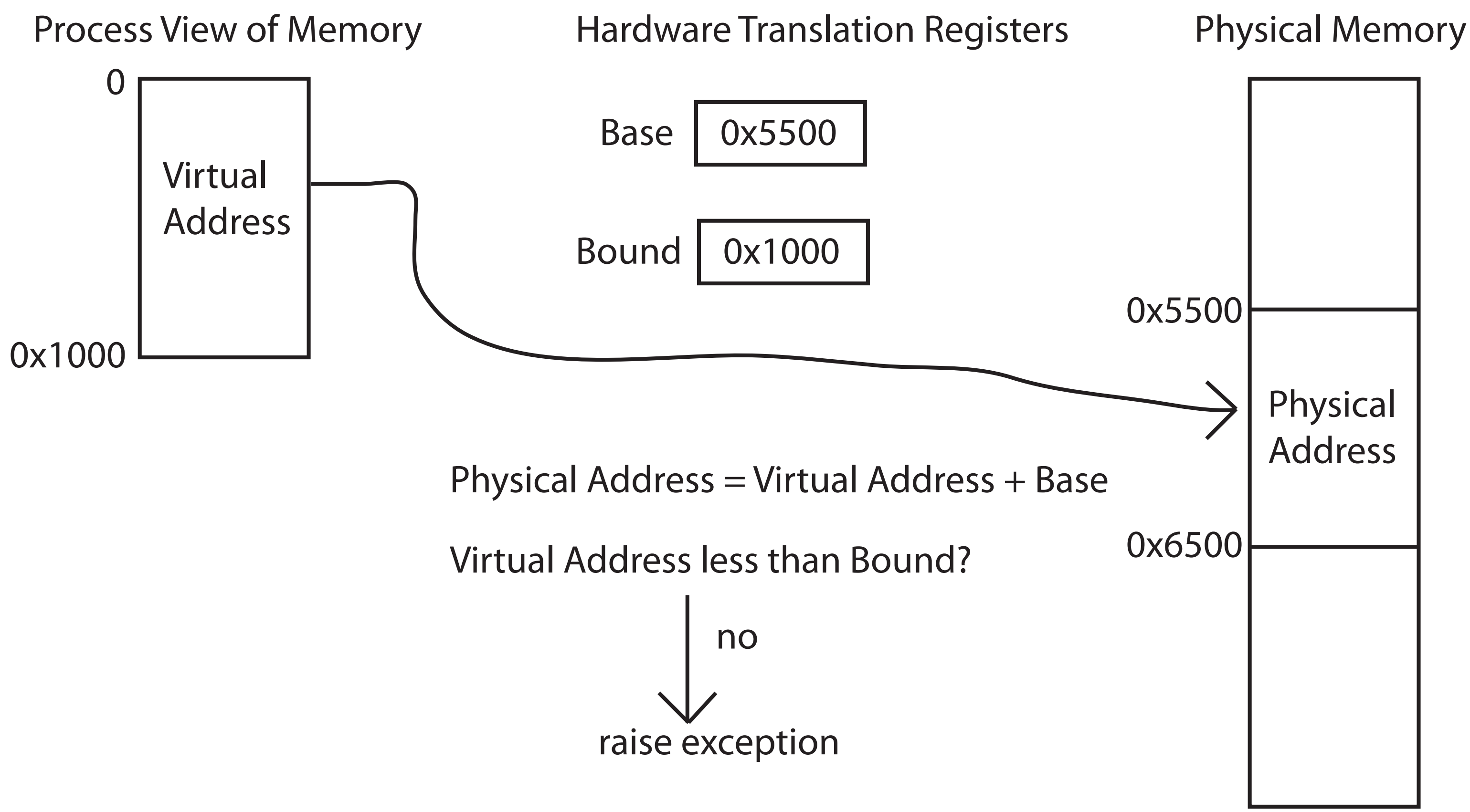
Address Translation (more)

- Cache management
  - Page coloring
- Program debugging
  - Data breakpoints when address is accessed
- Zero-copy I/O
  - Directly from I/O device into/out of user memory
- Memory mapped files
  - Access file data using load/store instructions
- Demand-paged virtual memory
  - Illusion of near-infinite memory, backed by disk or memory on other machines

Address Translation (even more)

- Checkpointing/restart
  - Transparently save a copy of a process, without stopping the program while the save happens
- Persistent data structures
  - Implement data structures that can survive system reboots
- Process migration
  - Transparently move processes between machines
- Information flow control
  - Track what data is being shared externally
- Distributed shared memory
  - Illusion of memory that is shared between machines

Virtual Base and Bounds

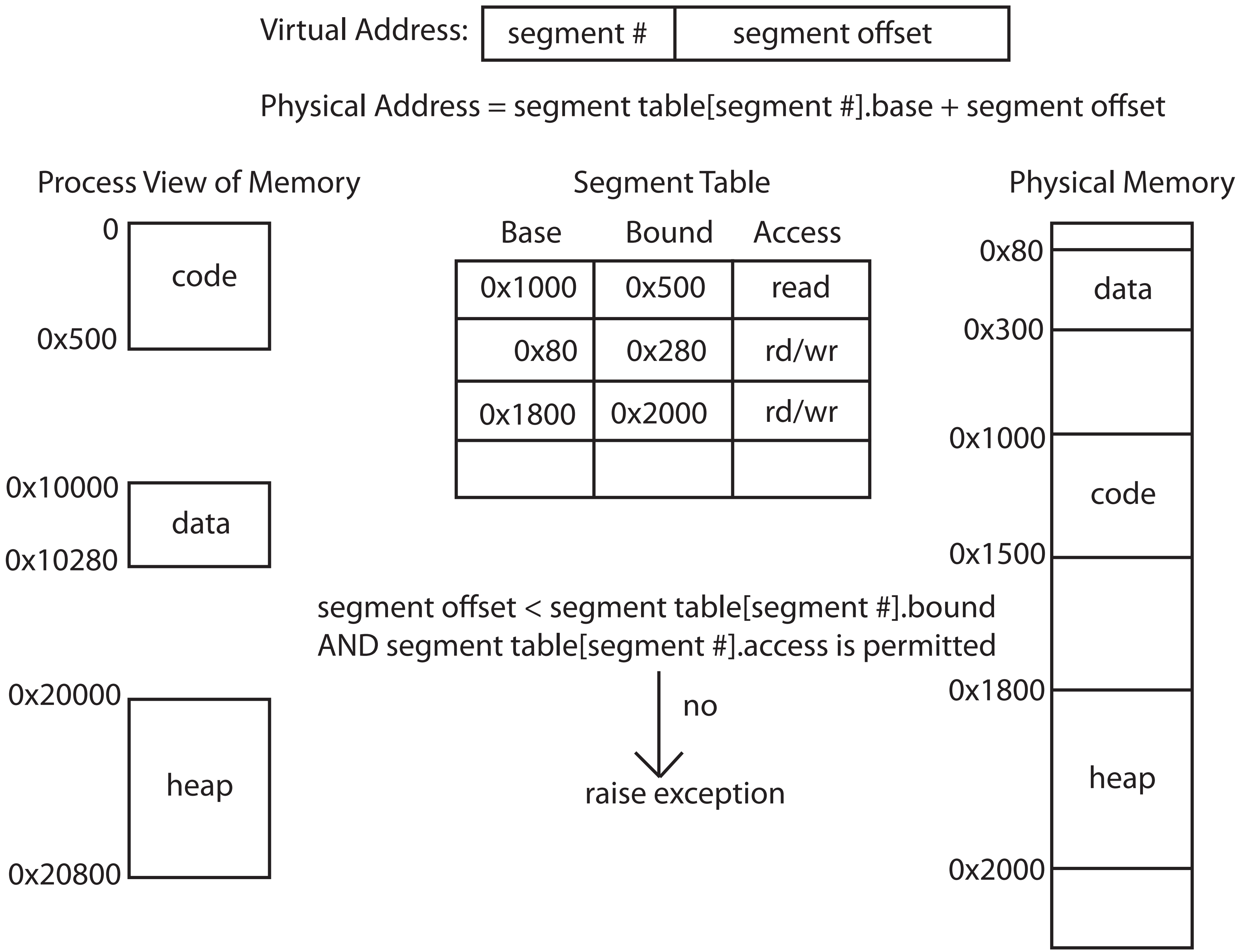


Virtual Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Can relocate in physical memory without changing process
- Cons?
  - Can’t keep program from accidentally overwriting its own code
  - Can’t share code/data with other processes
  - Can’t grow stack/heap as needed

Segmentation

- Segment is a contiguous region of memory
  - Virtual or (for now) physical memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Start
  - Length
  - Access permission
- Processes can share segments
  - Same start, length, same/different access permissions


Virtual Address:
segment #
segment offset
Physical Address = segment table[segment #].base + segment offset
Process View of Memory
0
code
0x500
0x10000
data
0x10280
0x20000
heap
0x20800
Segment Table
Base
Bound
Access
0x1000
0x500
read
0x80
0x280
rd/wr
0x1800
0x2000
rd/wr
segment offset < segment table[segment #].bound
AND segment table[segment #].access is permitted
no
raise exception
Physical Memory
0x80
data
0x300
0x1000
code
0x1500
0x1800
heap
0x2000

2 bit segment #
12 bit offset

	Segment start	length
code	0x4000	0x700
data	0	0x500
heap	-	-
stack	0x2000	0x1000

Virtual Memory

main: 240	store #1108, r2
244	store pc+8, r31
248	jump 360
24c	
…	
strlen: 360	loadbyte (r2), r3
…	…
420	jump (r31)
…	
x: 1108	a b c \0
…	

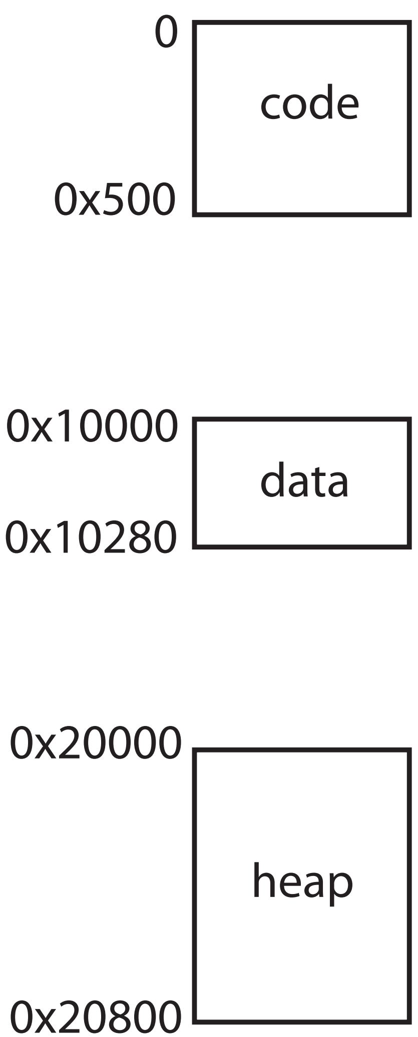
Physical Memory

x: 108	a b c \0
…	
main: 4240	store #1108, r2
4244	store pc+8, r31
4248	jump 360
424c	
…	…
strlen: 4360	loadbyte (r2),r3
…	
4420	jump (r31)
…	

UNIX fork and Copy on Write

- UNIX fork
  - Makes a complete copy of a process
- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment, will trap into kernel
    - make a copy of the segment and resume

Process View of Memory



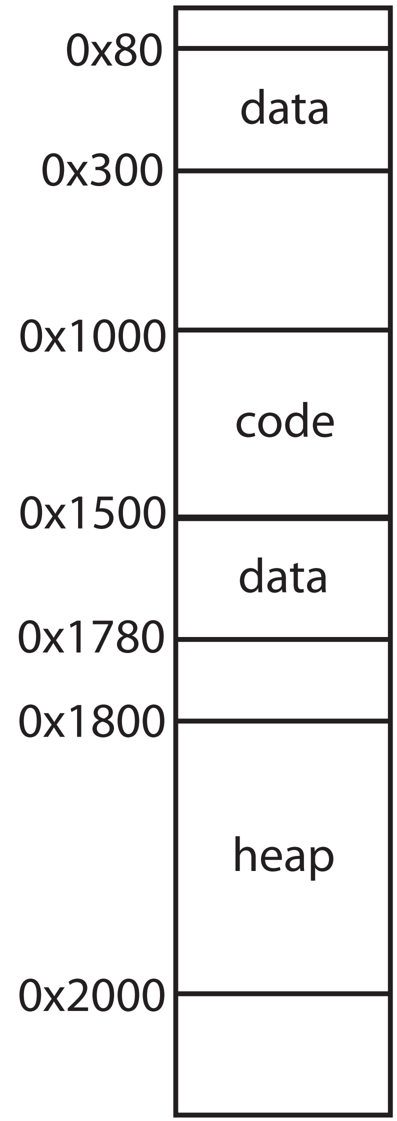
Process 1 Segment Table

Base	Bound	Access
0x1000	0x500	read
0x80	0x280	rd/wr
0x1800	0x2000	rd/wr

Process 2 Segment Table

Base	Bound	Access
0x1000	0x500	read
0x1500	0x280	rd/wr
0x2000	0x2800	rd/wr

Physical Memory



Zero-on-Reference

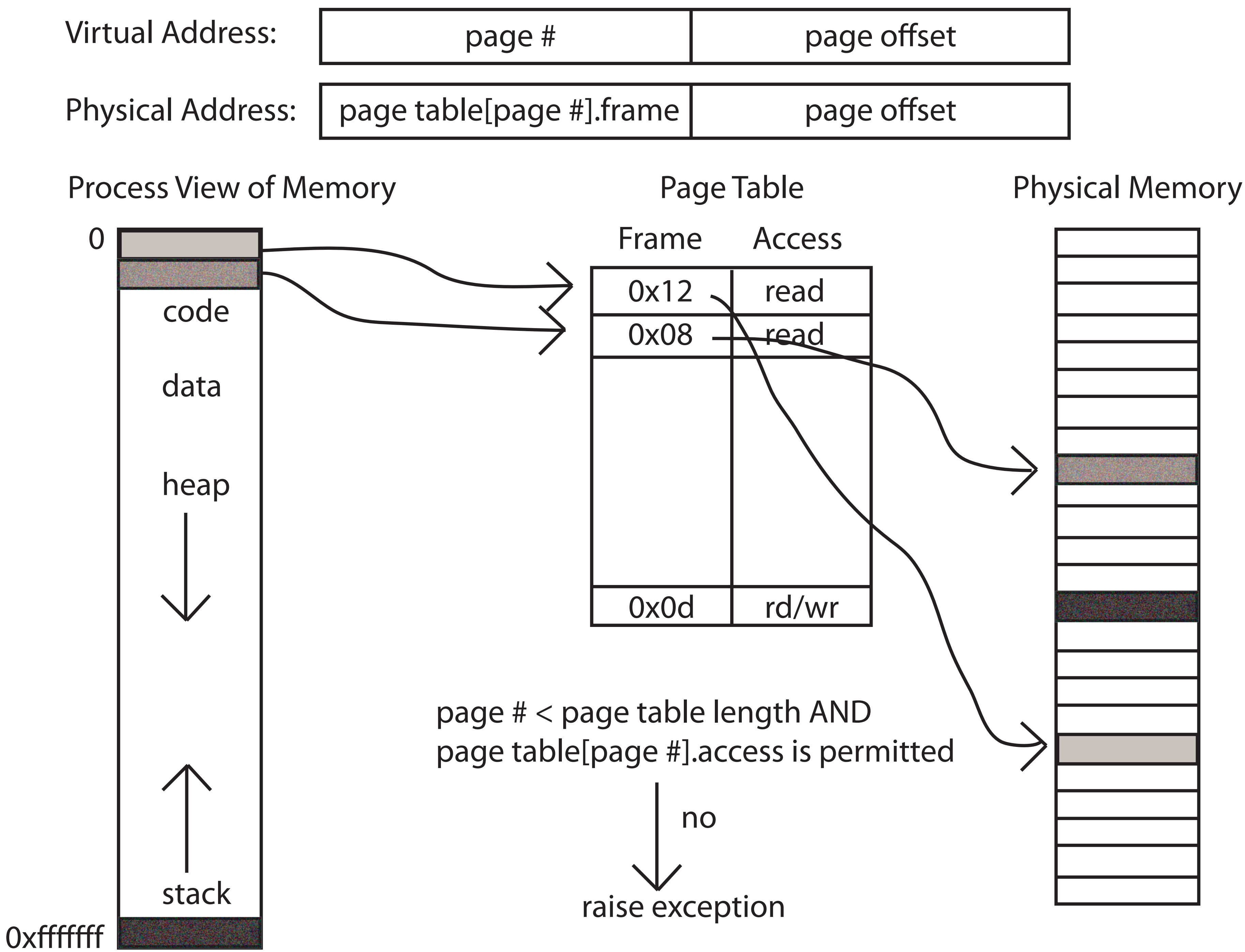
- How much physical memory do we need to allocate for the stack or heap?
  - Zero bytes!
- When program touches the heap
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Zeros the memory
    - avoid accidentally leaking information!
  - Restart process

Segmentation

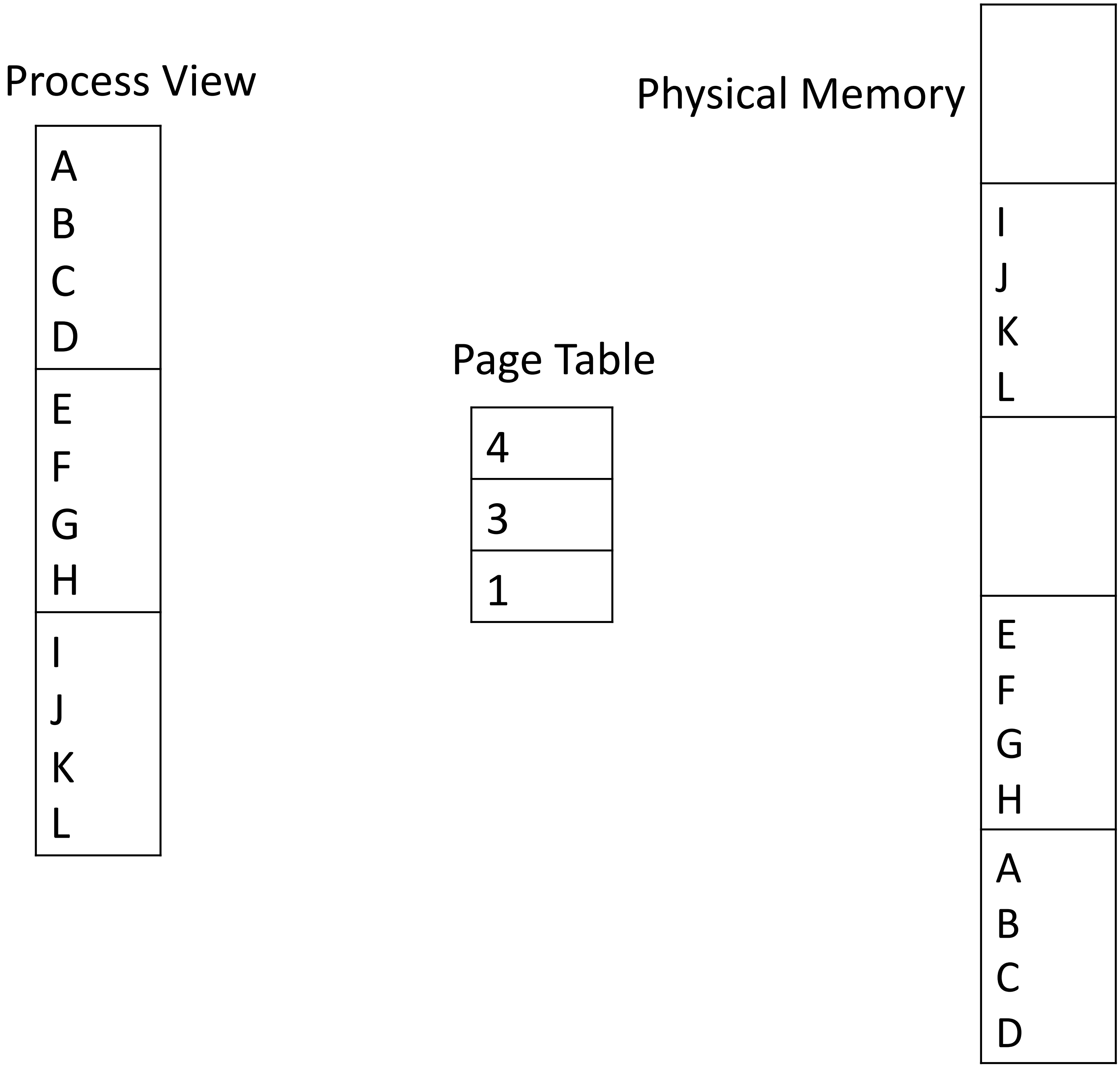
- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can grow stack/heap as needed
  - Can detect if need to copy-on-write
- Cons?
  - Complex memory management
    - Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
    - External fragmentation: wasted space between chunks

Paging

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware needs registers to hold pointer to page table, page table length



Virtual Address:
page #
page offset
Physical Address:
page table[page #].frame
page offset
Process View of Memory
0
code
data
heap
stack
0xffffffff
Page Table
Frame
Access
0x12
read
0x08
read
0x0d
rd/wr
Physical Memory
page # < page table length AND
page table[page #].access is permitted
no
raise exception


Process View
A
B
C
D
E
F
G
H
I
J
K
L
Page Table
4
3
1
Physical Memory
I
J
K
L
E
F
G
H
A
B
C
D

Paging Questions

- What must be saved/restored on a process context switch?
  - Pointer to page table/size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

Paging and Copy on Write

- Can we share memory between processes?
  - Set both page tables to point to same page frame
  - Need core map of page frames to track which processes are pointing to which page frames
- UNIX fork with copy on write at page granularity
  - Copy page table entries to new process
  - Mark all pages as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page and resume execution

Paging and Fast Program Start

- Do we need to have all of a program in physical memory before we start it running?
  - Set all page table entries to invalid
  - When page is referenced for first time
    - Trap to OS kernel
    - OS kernel brings in page
    - Resumes execution
  - Remaining pages can be transferred in the background while program is running

Sparse Address Spaces

- What if virtual address space is sparse?
  - On UNIX, code starts at 0
  - Stack starts at 2^31
  - 1KB pages => 2M page table entries