Address Translation

Chapter 8 OSPP

Part I: Basics

Important?

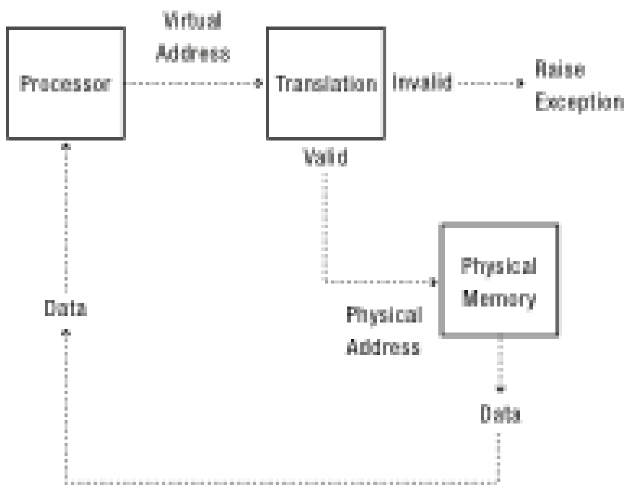
- Process isolation
- IPC
- Shared code
- Program initialization
- Efficient dynamic memory allocation
- Cache management
- Debugging
- Efficient I/O
- Memory mapped files
- Virtual memory
- Checkpoint/restart
- …

All problems in computer science can be solved by another level of indirection!

Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Base and bound
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers (TLB)
  - Virtually and physically addressed caches

Address Translation Concept



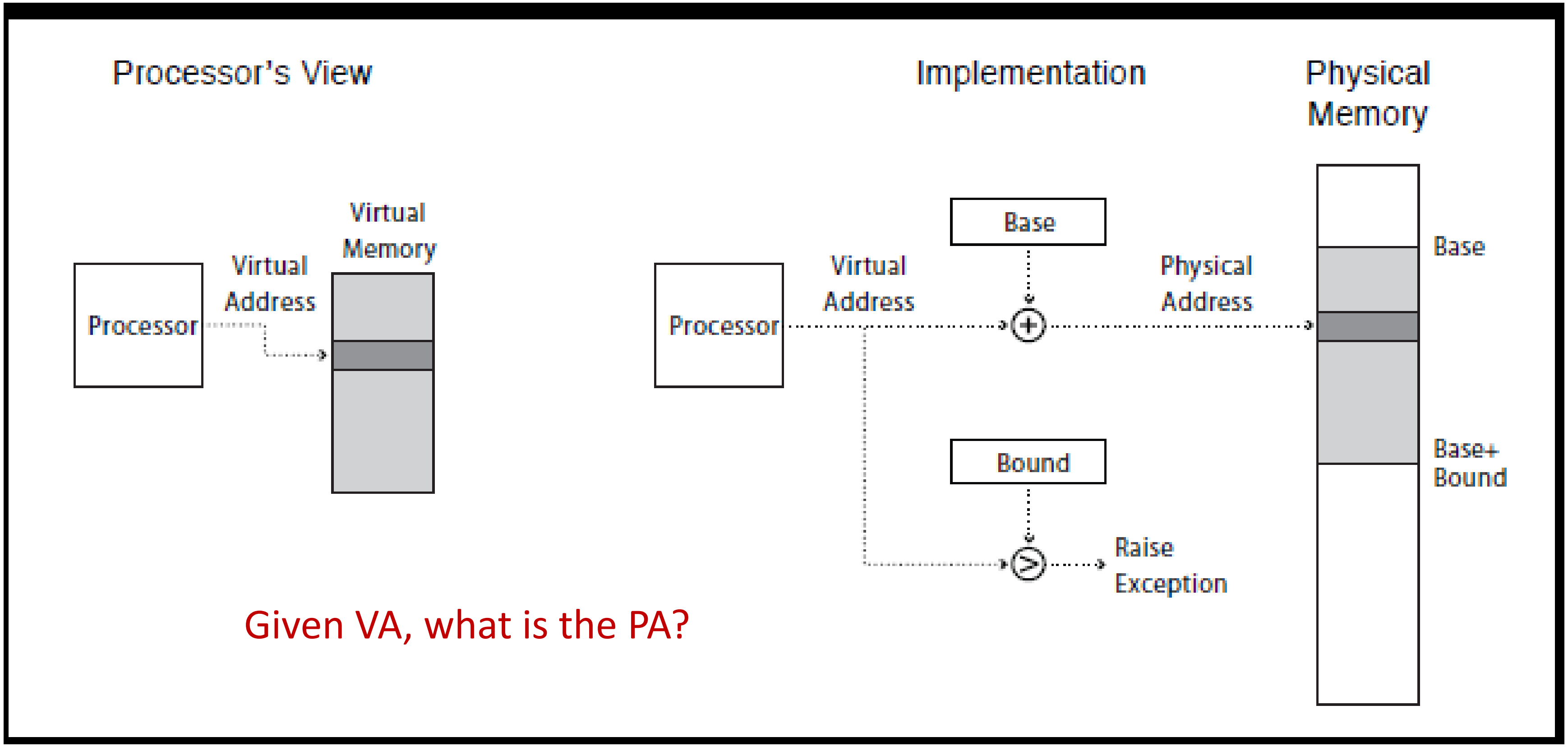
Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, shared-memory IPC
- Sparse addresses (64 bit addresses)
  - Multiple regions of dynamic allocation (heaps/stacks)
  - Allow room for growth
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability
  - OS must exploit hardware

Bonus Feature

- What can you (OS) do if you can (selectively) gain control whenever a program reads or writes a particular virtual memory location?
- Examples:
  - Copy on write
  - Zero on reference
  - Demand paging
  - Fill on demand
  - Memory mapped files

Virtually Addressed Base and Bounds



Hardware support is minimal: base register, bound register

Question

- With virtually addressed base and bounds, what is saved/restored on a process context switch?
  - Usually just the base and bound register

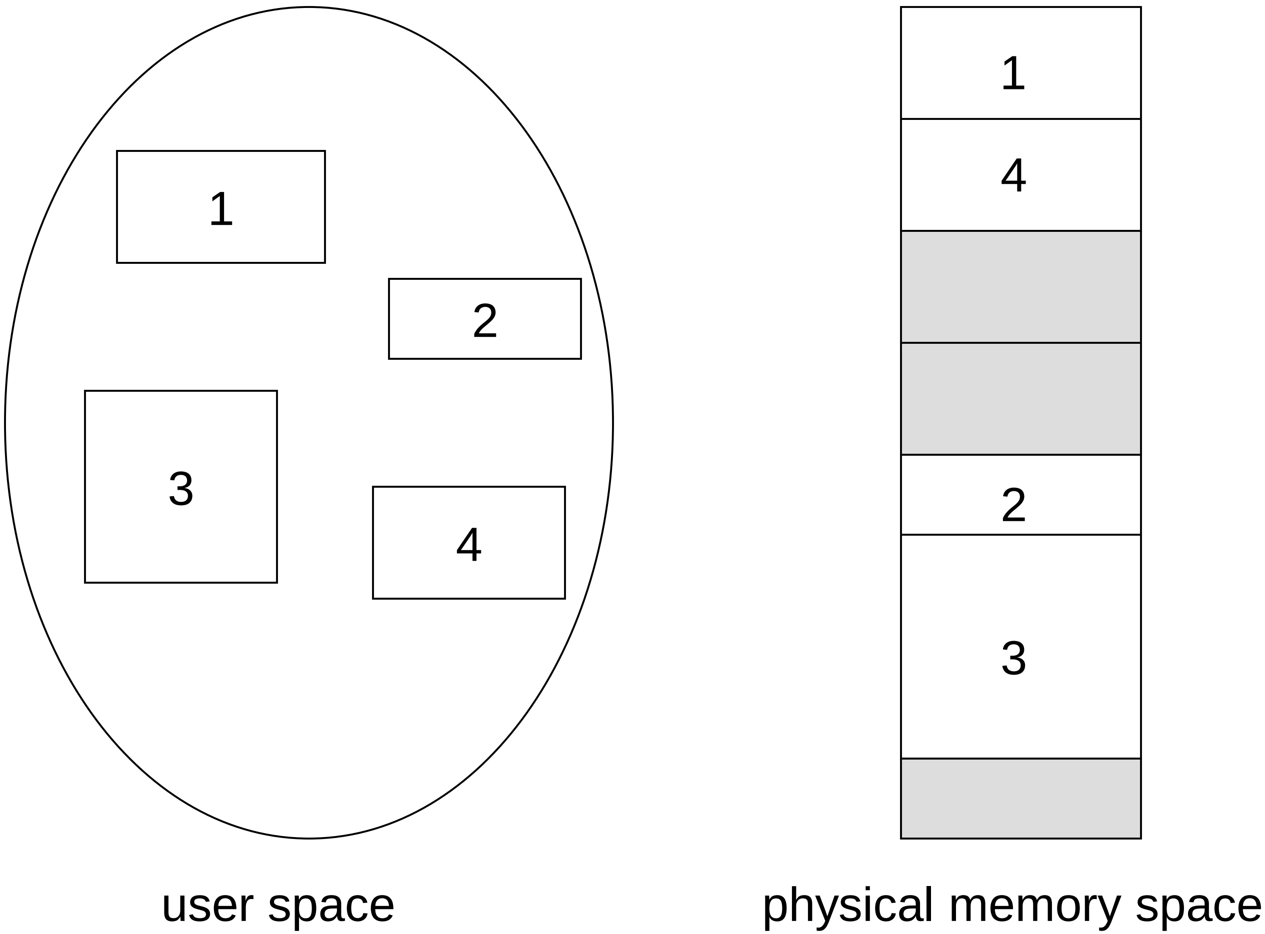
Virtually Addressed Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
- Cons?
  - Can’t share code/data with other processes
  - Can’t grow stack/heap as needed
  - Fragmentation

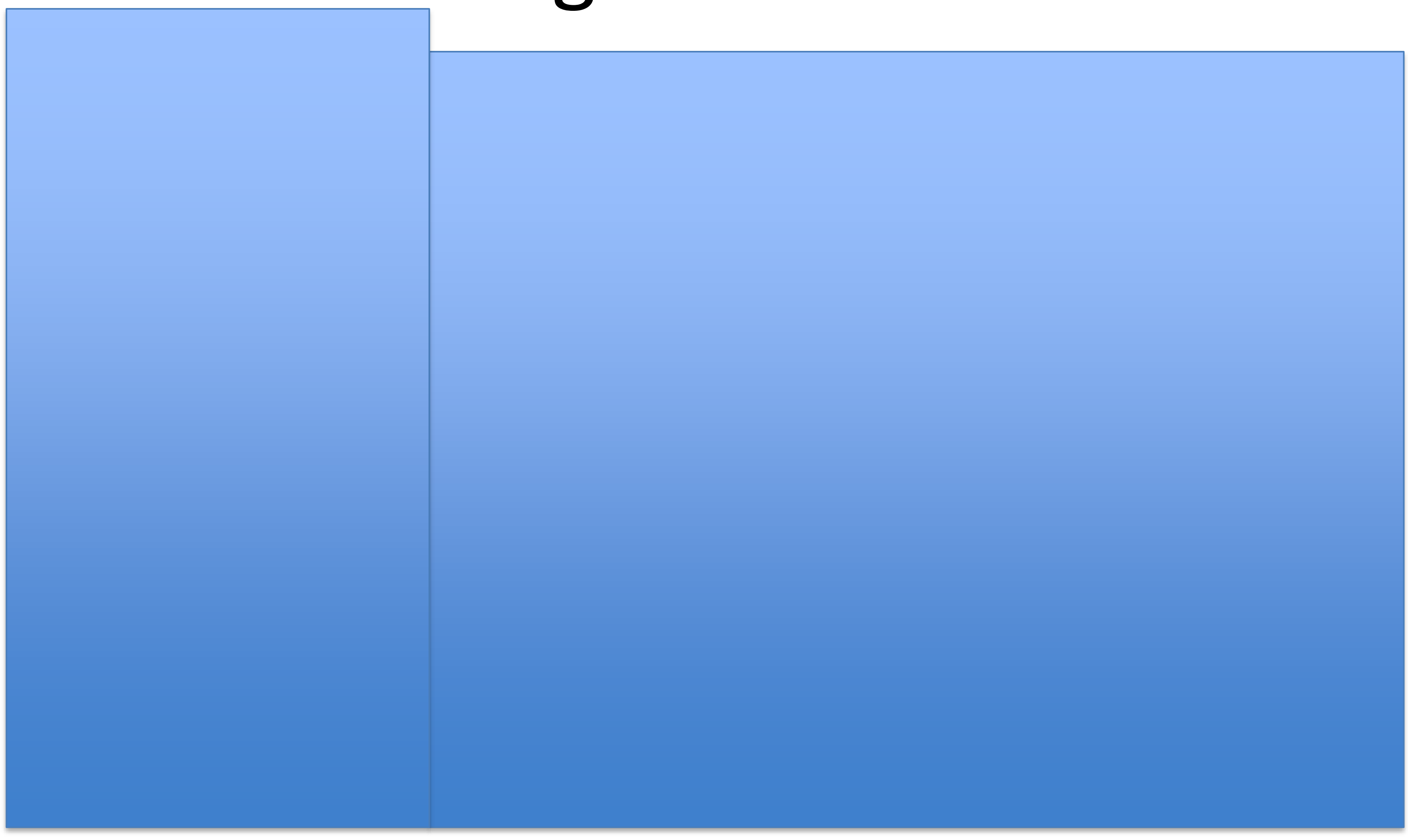
Segmentation

- Segment is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware or mem)
  - Entry in table for each segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions
  - Great for shared libraries

Logical View



Segmentation



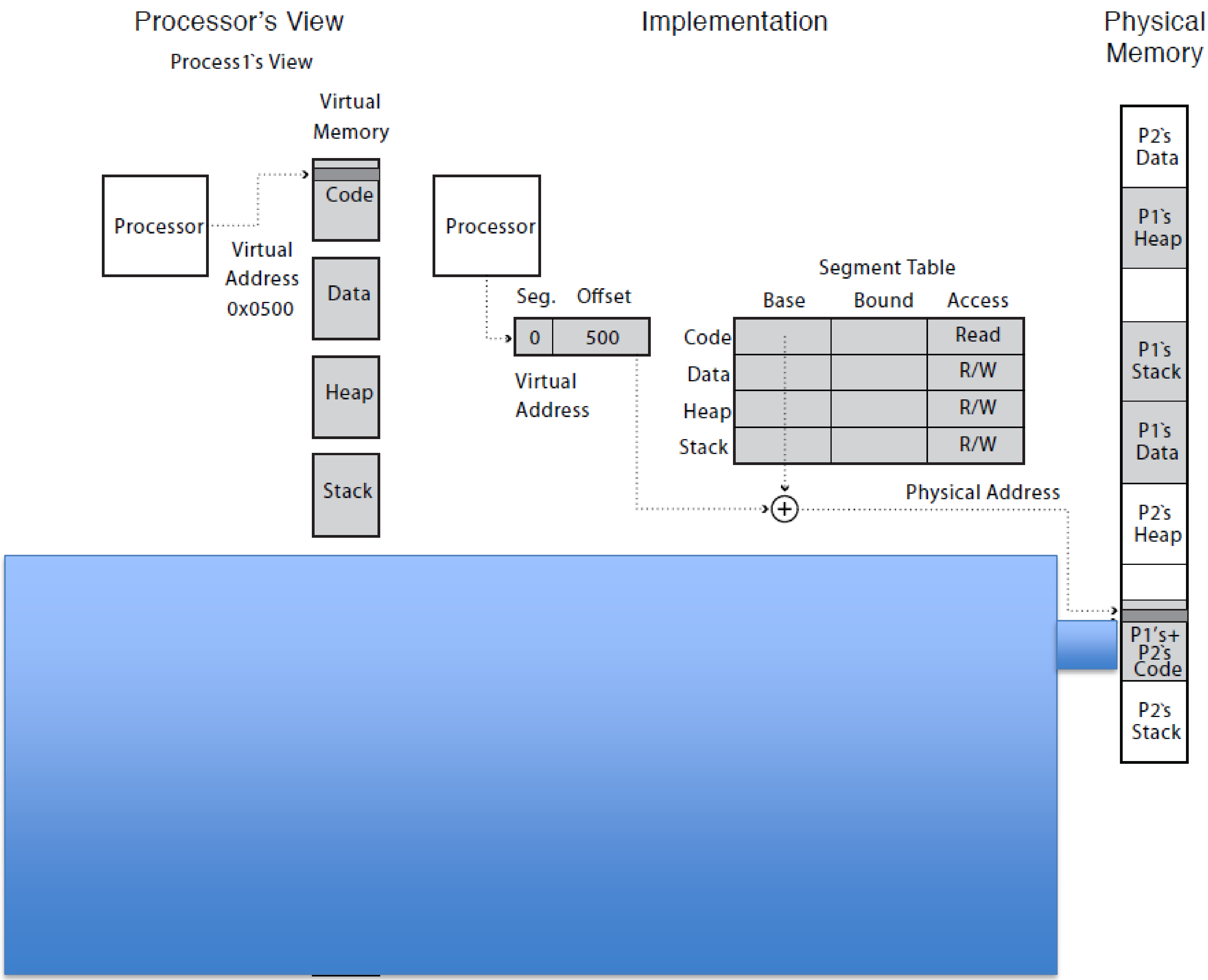
Hardware support: segment table start and length register (# segs)

Question

- With segmentation, what is saved/restored on a process context switch?
  - assuming segment table is in memory, just the segment table start and end pointer registers
  - if segment table fits in registers (small), then would need to save/restore the entire table

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 (ex: stack, heap)
    - trap into kernel
    - make a copy of the segment and resume


Processor's View
Process1's View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Implementation
Processor
Seg. Offset
0 500
Virtual Address
Segment Table
Base Bound Access
Code Read
Data R/W
Heap R/W
Stack R/W
Physical Address
Physical Memory
P2's Data
P1's Heap
P1's Stack
P1's Data
P2's Heap
P1's+ P2's Code
P2's Stack

Dynamic Segments & Zero-on-Reference

- Dynamic segments: not all impl. allow this
  - When program uses memory beyond bound (e.g. end of stack)
  - Segmentation fault into OS kernel
  - Kernel can then allocate some additional memory
    - How much?
- Zeros the memory
  - idea: set segment bound (i.e. stack) artificially low
  - at seg fault, kernel zeros the memory
  - avoid accidentally leaking information!
- Modify segment table
- Resume process

More on zero’ing

- If data is so sensitive, why not have programs zero their own memory?
  - `bzero` system call

- Background: when CPU is idle, we can zero memory not currently allocated

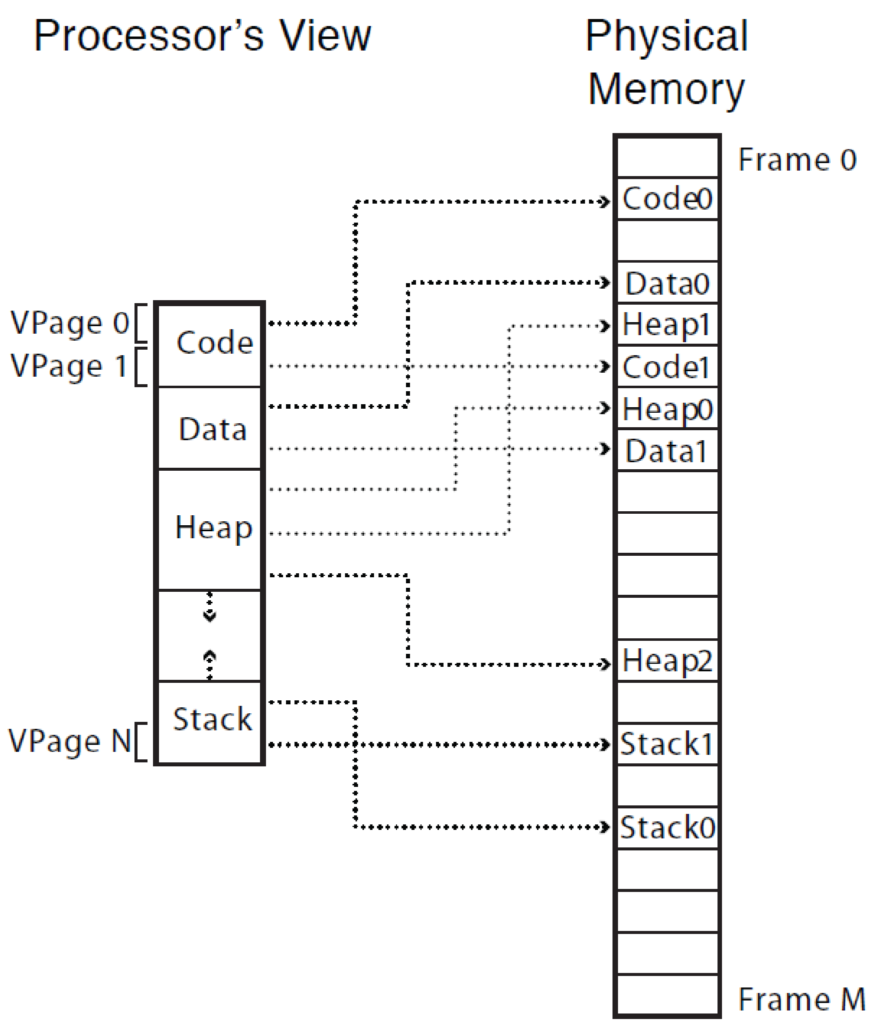
Segmentation

- Pros?
  – Can share code/data segments between processes
  – Can protect code segment from being overwritten
  – Can transparently grow stack/heap as needed - maybe
  – Can detect if need to copy-on-write/zero-on-ref
  – Matches programmer view with memory view
- 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

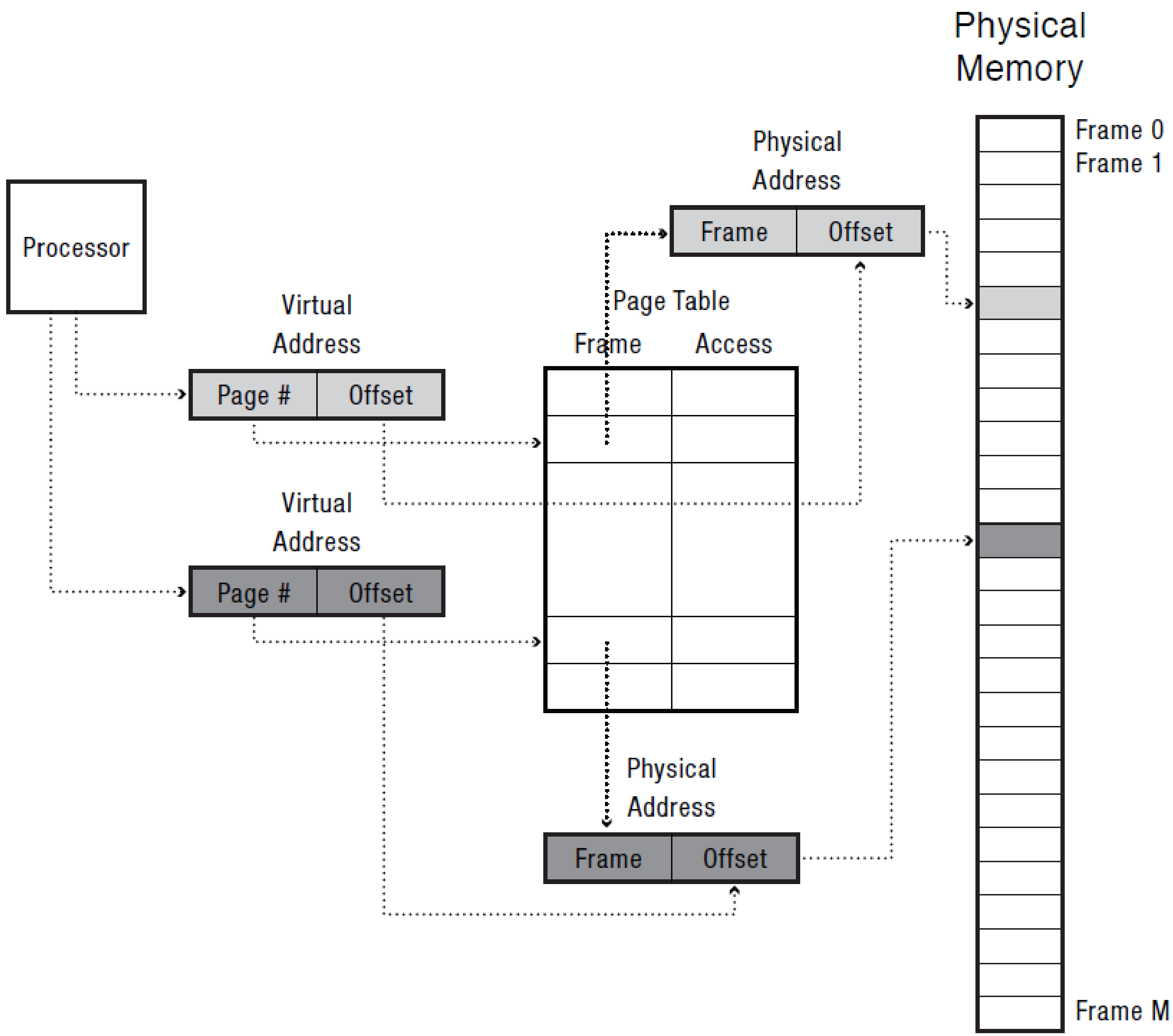
Solve Fragmentation: Paged Translation

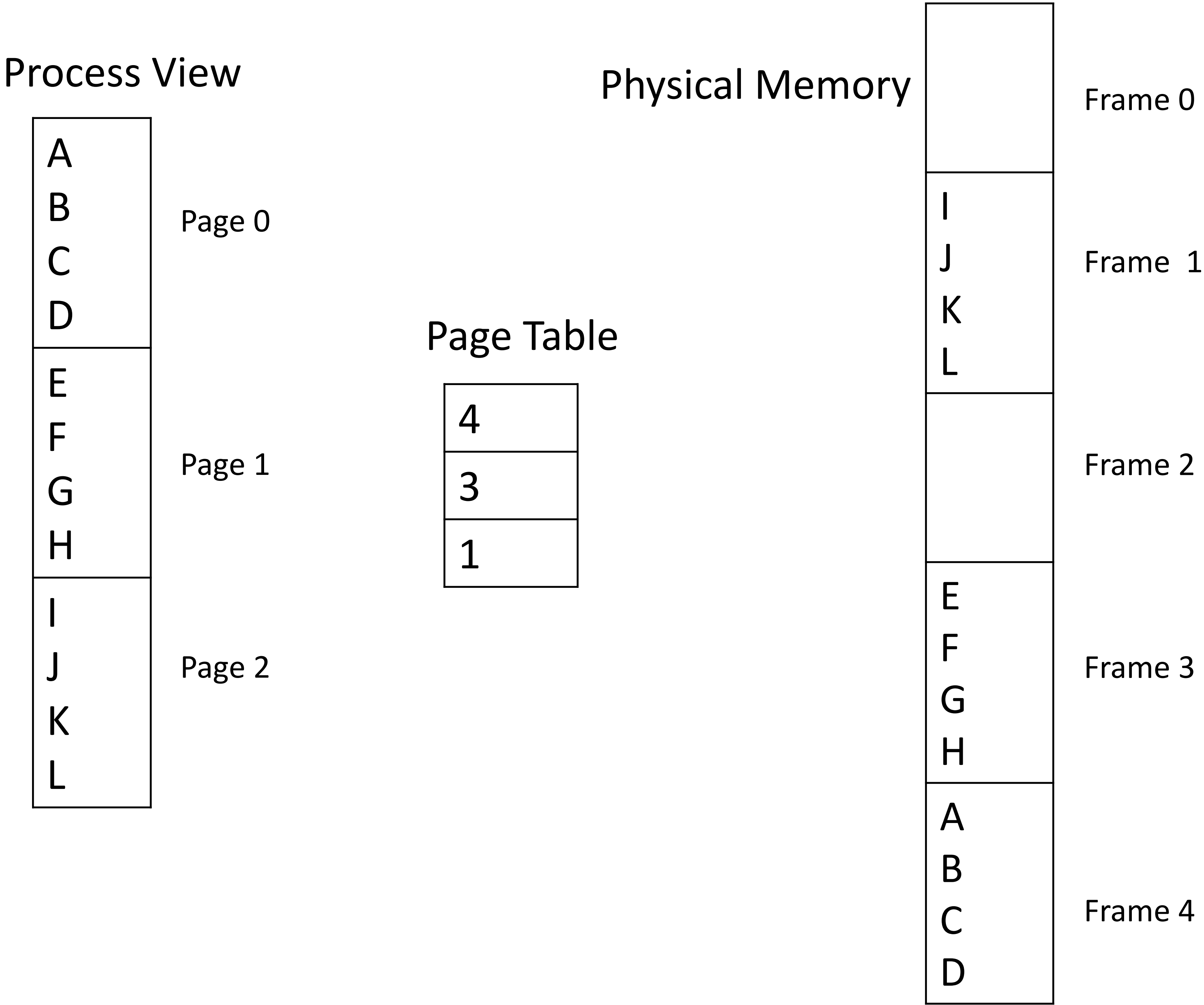
- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
- Hardware registers
    - pointer to page table start
    - page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

Paged Translation (Abstract)



Paged Translation (Implementation)




Process View
A
B
C
D
Page 0
E
F
G
H
Page 1
I
J
K
L
Page 2
Page Table
4
3
1
Physical Memory
Frame 0
I
J
K
L
Frame 1
Frame 2
E
F
G
H
Frame 3
A
B
C
D
Frame 4

Example

Comparison

- Like segmentation, paging adds a level of indirection
- Page size is smaller than segment size generally
- What about translation overhead?
- What about memory overhead (size) of paging vs. segmentation?

Paging Questions

- With paging, what is 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?
  - Big page tables, lots of I/O (as we will see)
- 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 entries in both page tables to point to same page frames
  - Need *core map* of page frames to track which processes are pointing to which page frames (e.g., reference count): why?

- UNIX fork with copy on write
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Demand Paging/Fill On Demand

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid
  - When a page is referenced for first time, kernel trap, “page fault”
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running

Data Breakpoints

- Please trace variable A
- Mark page P containing A as read-only
- If P is changed, trap into kernel, and see if A actually changed?
- Why is this better with paging vs. segmentation?

Page Table Issue

- 64 bit machines
- Page table(s) can get huge
- Need to address this
- 16 bit page size, 50 bits for pages, 2^50 entries in PT PER process!

Next Week

- Chapter 9 virtual memory
- Chapter 8; multi-level translation