Introduction to Operating Systems

CPSC/ECE 3220
Lecture Notes
OSPP Chapter 8 – Part A

(adapted by Mark Smotherman and Lana Drachova from publisher’s slides)

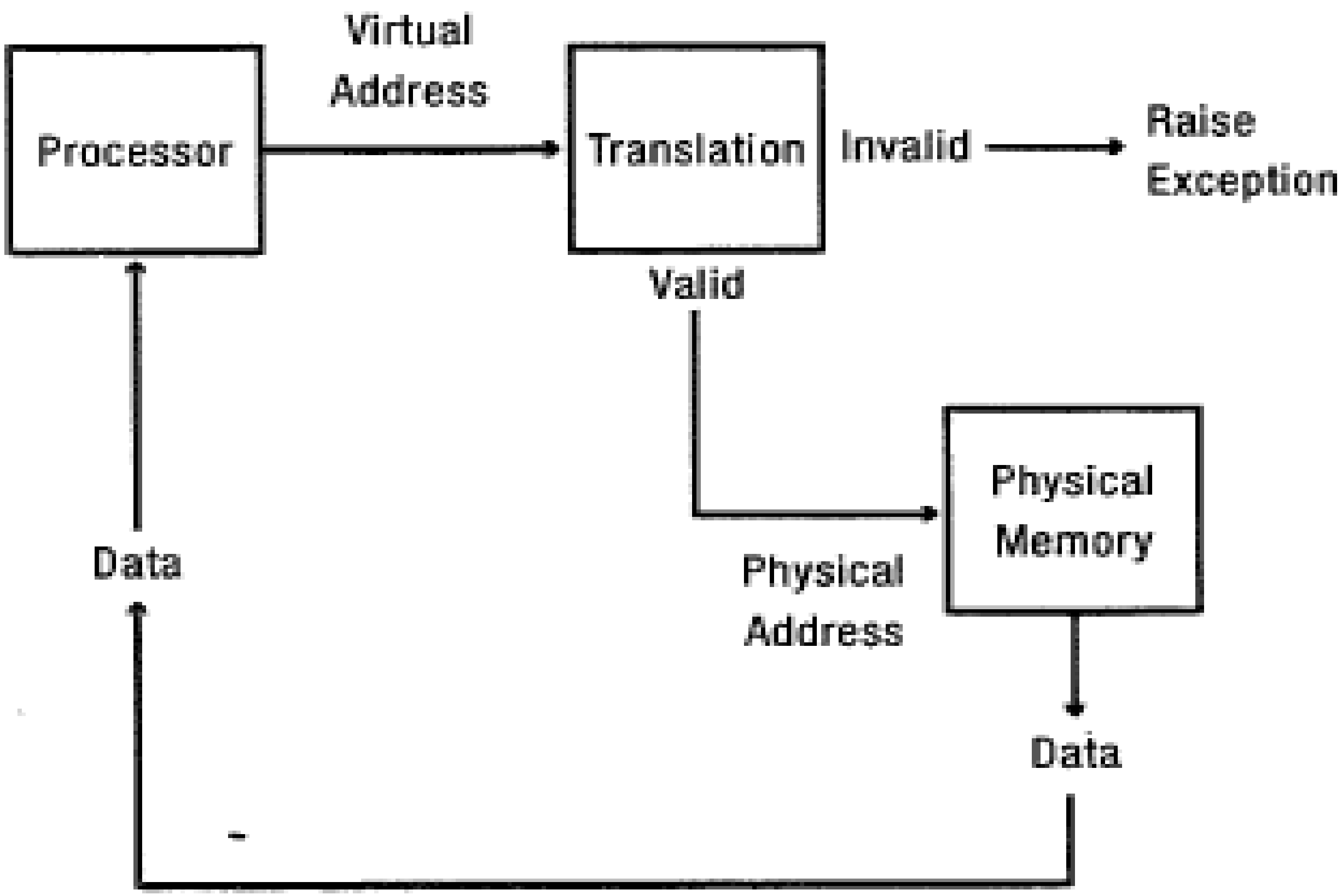
Main Points

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

Address Translation Goals

- Memory protection
- Memory sharing (libraries, code, ipc, data structures)
- Flexible Placement
- Sparse addresses (stack/heap grow as needed)
- Efficiency (let OS decide)
- Runtime lookup (every fetch and load/store)
- Compact translation tables (to reduce overhead)
- Portability

Address Translation Concept



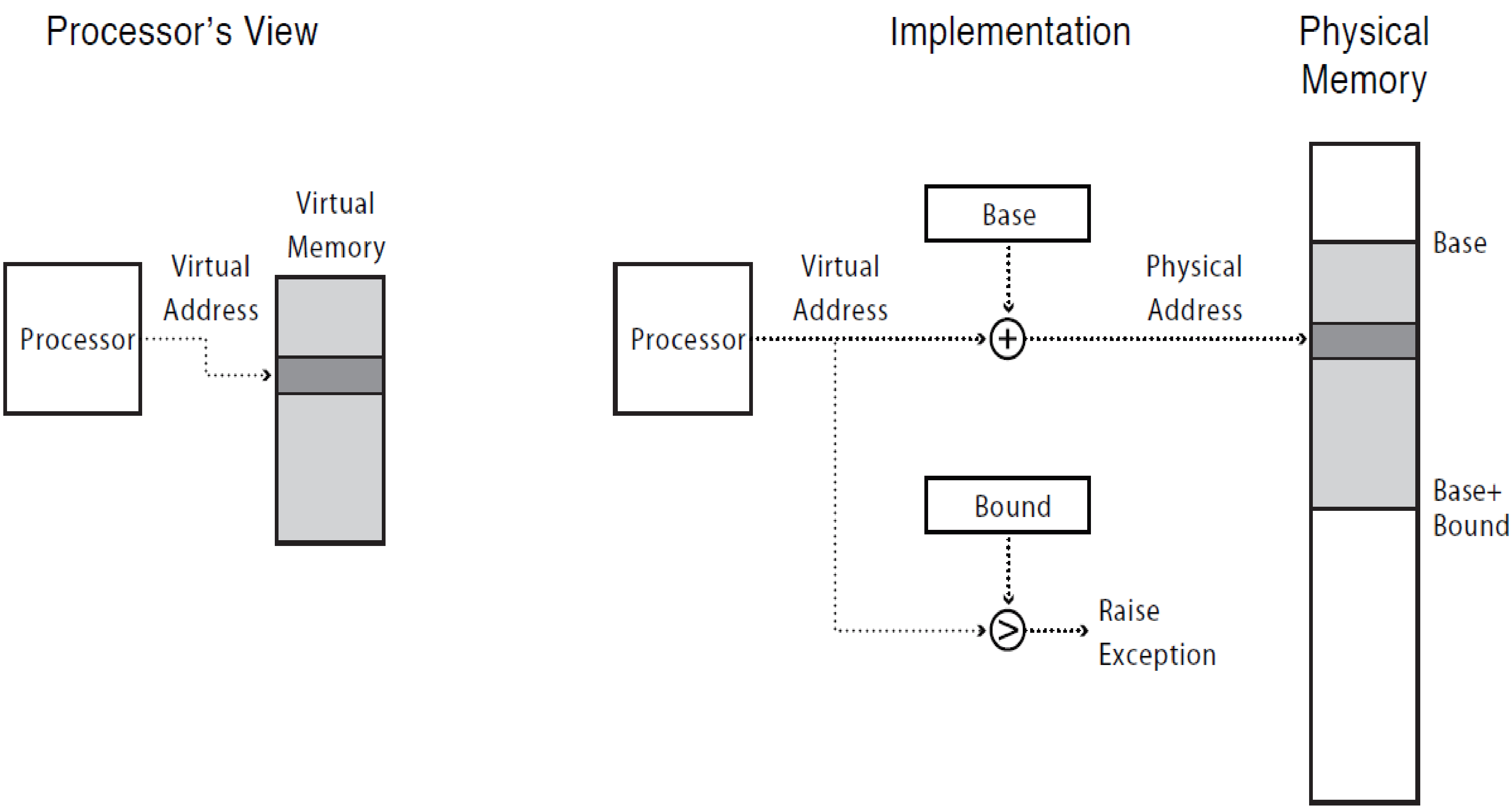
Base and Bounds Registers

- Compiled program contains addresses starting at 0
- Program is loaded into physical memory to run
- One continuous chunk of memory is allocated for that process to run

- *Base* – start of the region of physical memory
- *Bounds* – the extent of that region

- *Virtual address range* – from 0 to bounds
- *Physical address range* – from base to base+bounds

Virtually Addressed Base and Bounds



Virtually Addressed Base and Bounds

- Pros
  - Simple, safe, fast (2 registers, adder, comparator)
  - Can relocate in physical memory without changing process

- Cons
  - Course-grained protection @ level of the entire process
  - Can’t keep program from overwriting its code
  - Can’t share code/data with other processes
  - Can’t grow stack/heap as needed

Segmentation

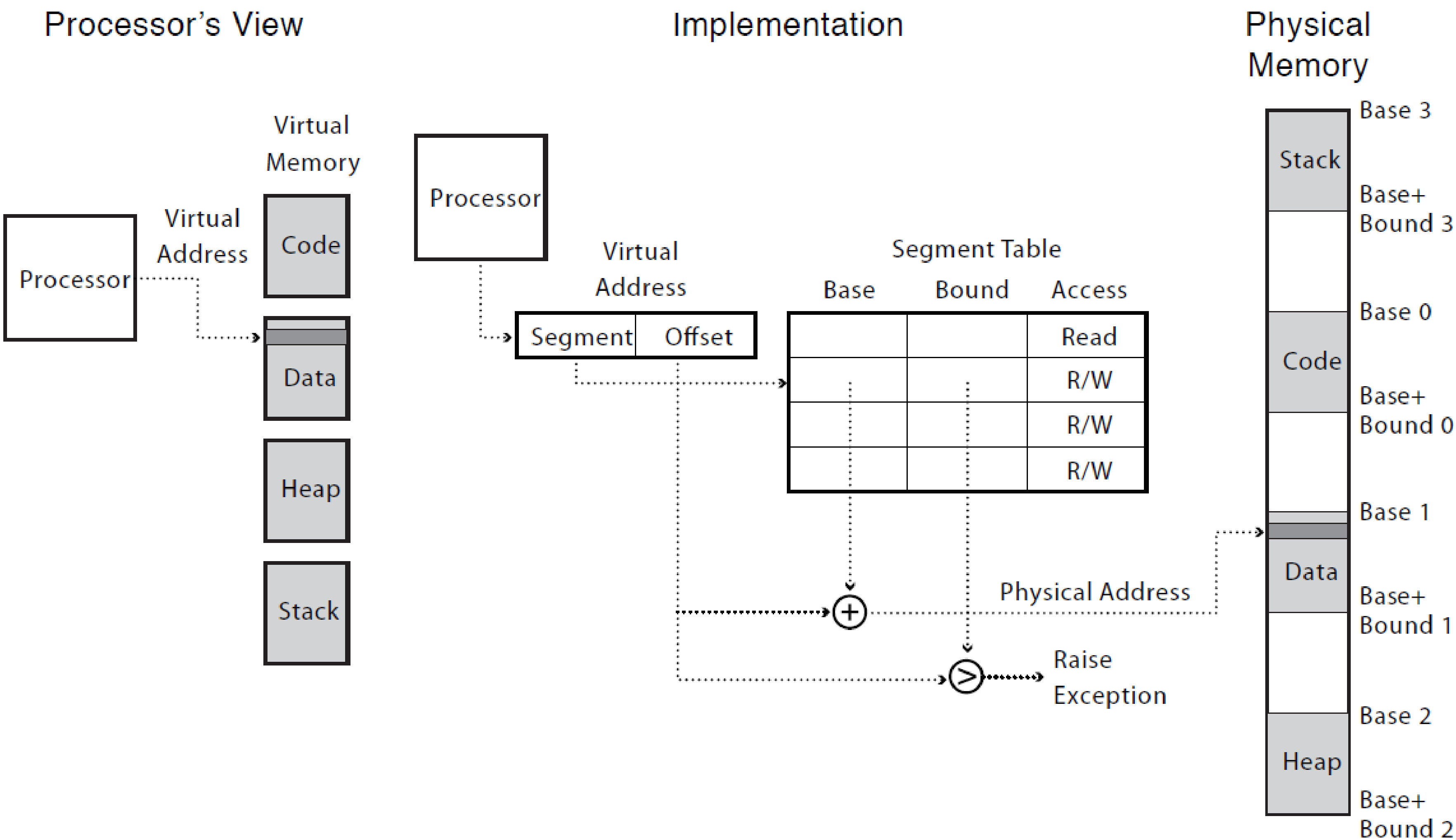
- An array of base/bounds pairs, for each part of process
- Each segment is a *contiguous* region of *virtual* memory
- Segments vary in size
- Each process needs a segment table (in hardware)
  - Entry in table = segment
  - Table management is overhead (higher bits = segment, lower bits are offset)

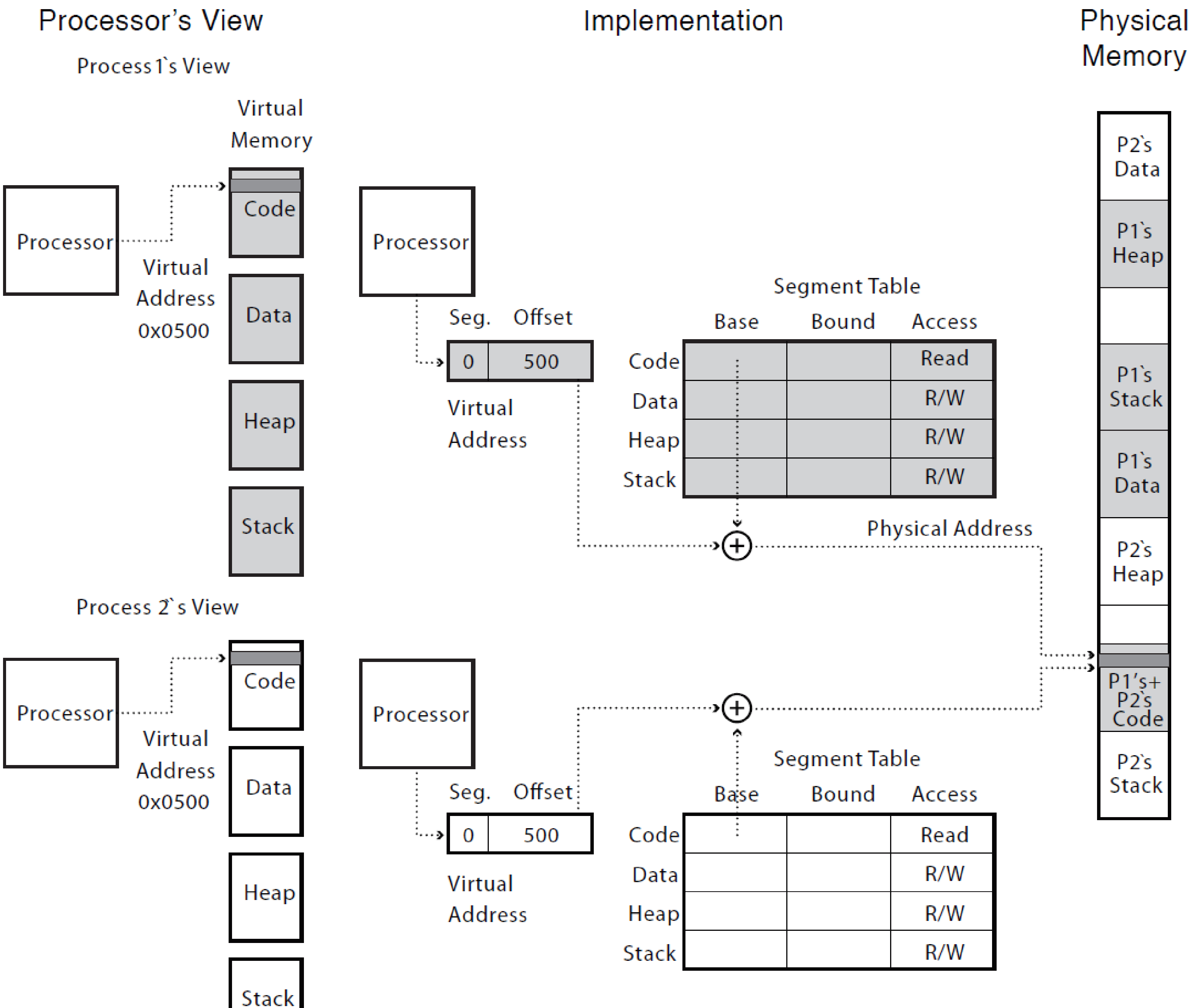
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission.
  - Segmentation fault is access outside memory regions

Segmentation

- Processes can share some memory regions, but not others (shared libraries)
  - Share code segment – their segment table points to the same area, same base and bounds
- Segments can be used for inter-process comm (need R/W permissions)
- Efficient in managing dynamically allocated memory (zero-on-reference for heap and stack, if need more – exception, zero, and move bounds)

Segmentation





Memory Area Sharing

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

Zero on Reference

- How much physical memory is needed for stack or heap?
  - It is initially “dirty”.
  - Only small part is clean (zeroed out).
- When program uses memory beyond clean end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
  - Zeros the memory
    - Avoid accidentally leaking information!
    - Avoid using garbage values!
  - Modify segment table
  - Resume process

Segmentation

- **Pros**
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write

- **Cons**
  - Complex memory management
    - Need to find free chunk of a particular size
  - May need to rearrange memory for new or growing segments
  - Various memory compaction schemes exist

    - External fragmentation: wasted space between chunks

External fragmentation

- Memory is divided into variable size segments allocated to different processes.

- External Fragmentation – wasted memory *between* segments.

- May be unable to run a process, even though the amount of total free memory is > than the process needs

- Need memory compaction algorithm (overhead)

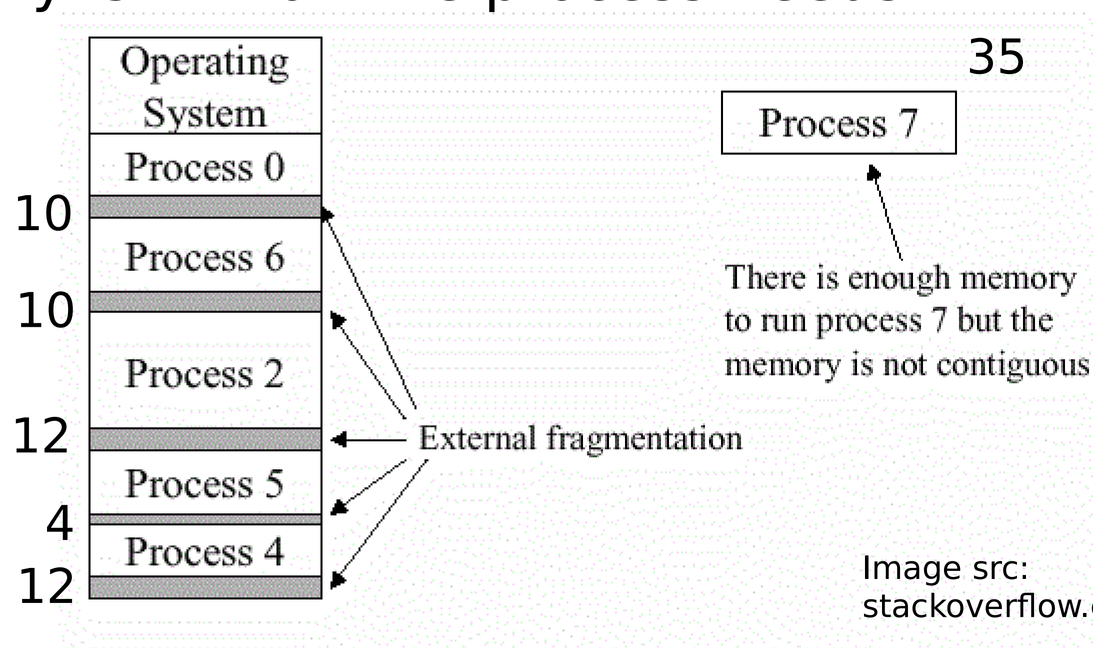
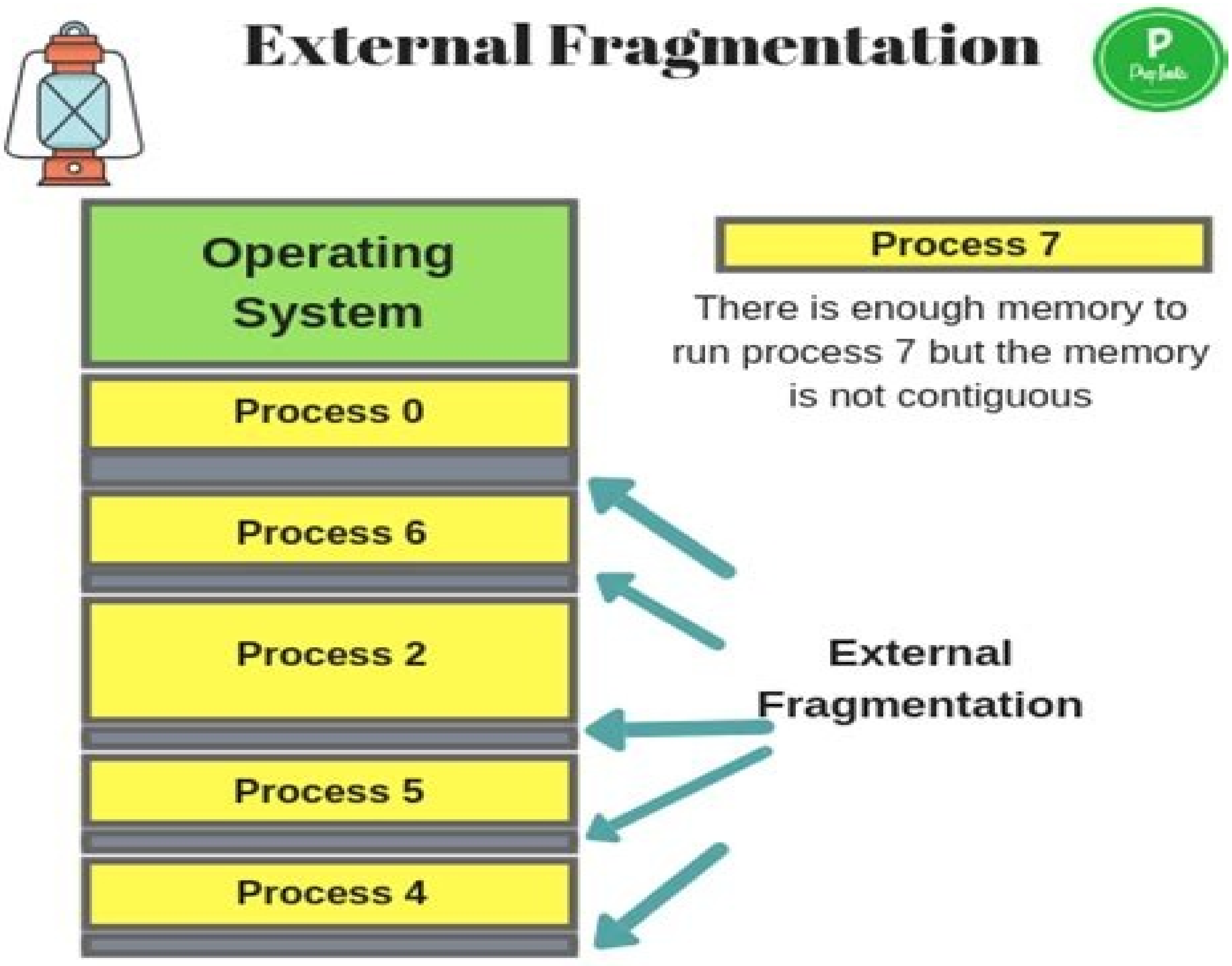


Image src: stackoverflow.com

External Fragmentation (img src: prepinsta.com)



Paged Translation

- Virtual memory is divided into fixed sized units – pages
- Physical memory is divided into fixed size units - page frames
- No need for a bounds register
  - The unit size is fixed
- Virtual page = physical page = disk sector

    Why do you think they are of equal size?

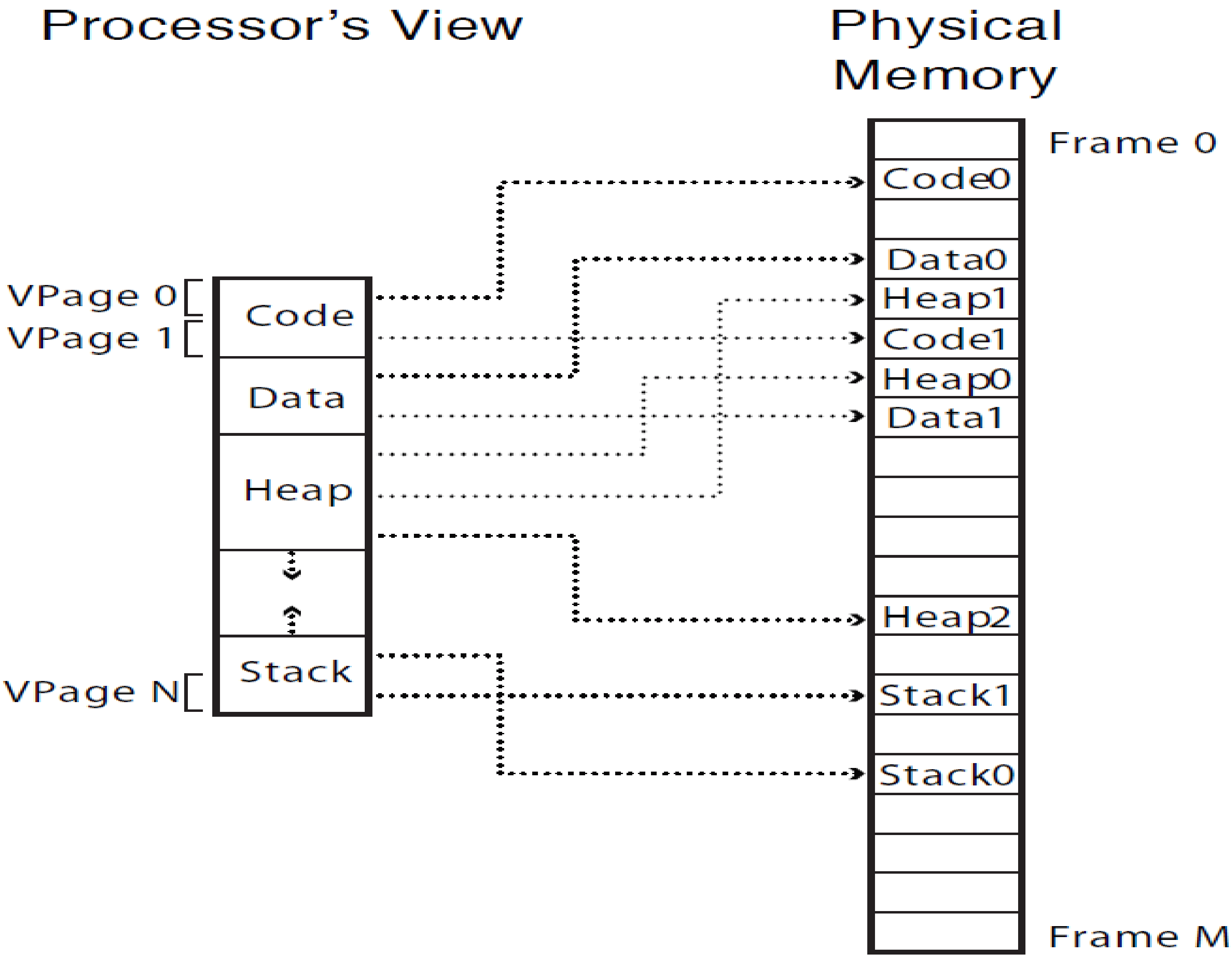
Paged Translation

- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
  - 0 = free and 1 = occupied

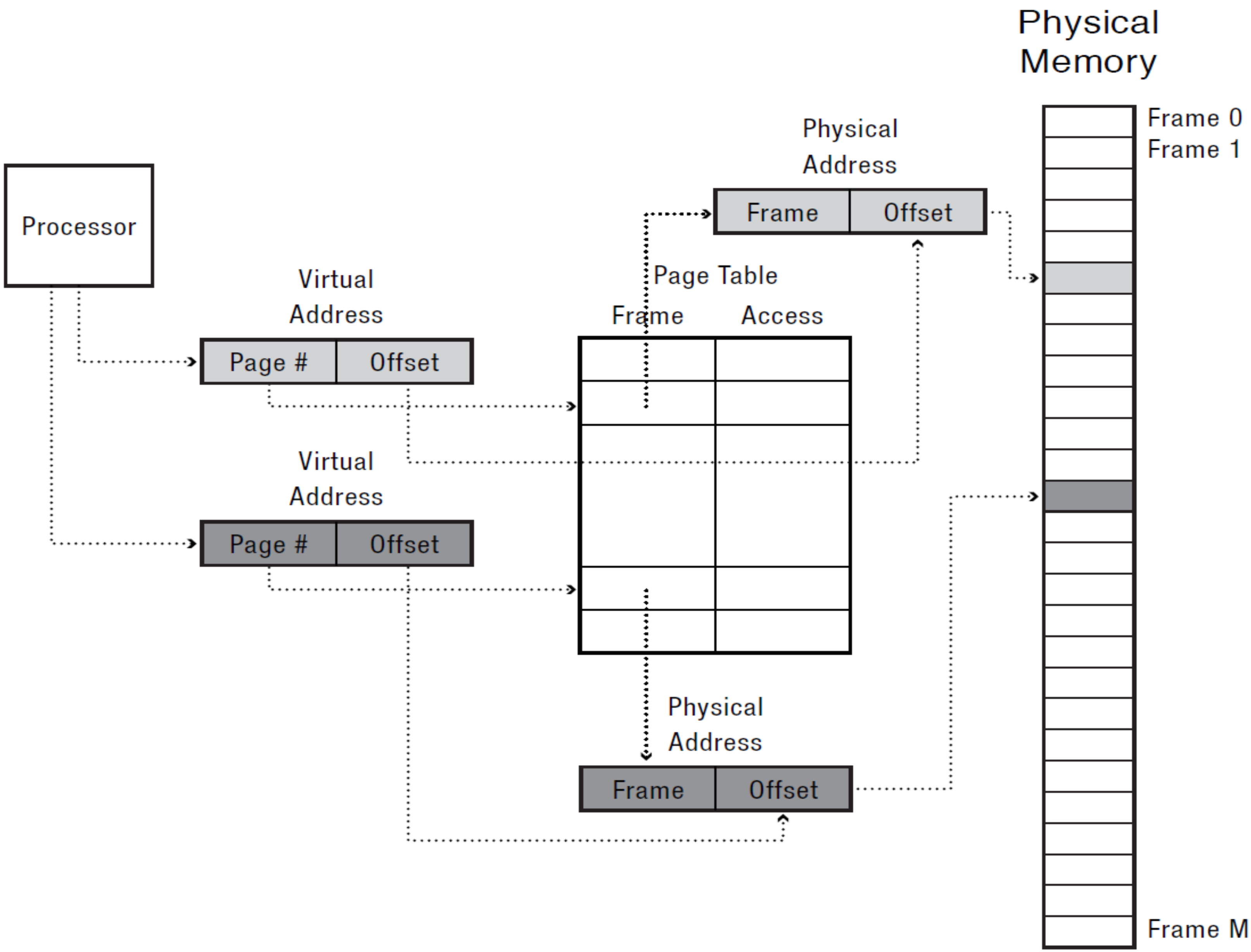
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - Pointer to page table start and page table length

      Do we have external fragmentation here?

Paged Translation (Abstract)



Paged Translation (Implementation)



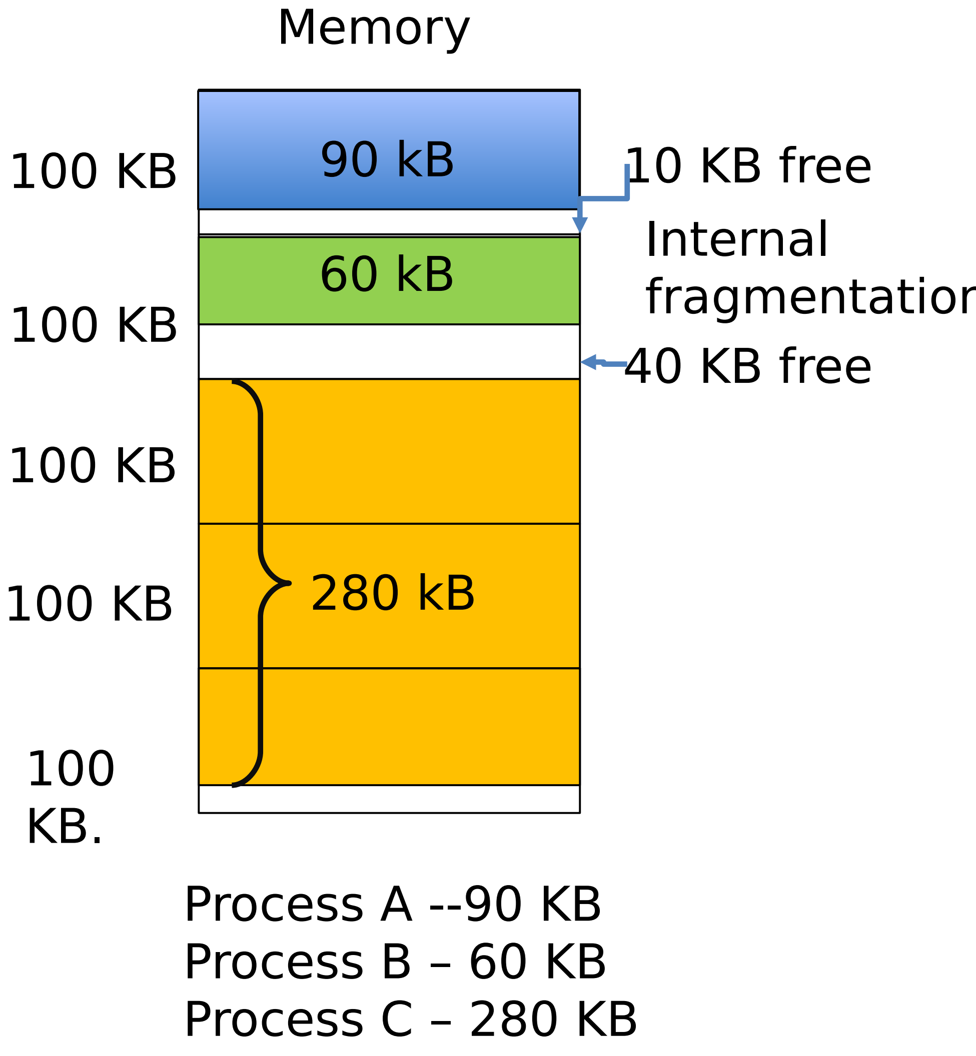
Paging Questions

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table and size of page table are loaded into privileged registers
  - 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

Internal Fragmentation

- The size of a process may not be a multiple of page frame sizes.
- *Internal Fragmentation* is a wasted memory in the last page frame only.
- Can be managed by smart page frame size selection.



Paging and Copy on Write

- Can we share memory between processes?
  - Set entries in both page tables to point to the same page frames
  - Need *core map* of page frames to track which processes are pointing to which page frames (e.g., reference count)

- 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 in parent)
  - Copy page and mark both as writeable
  - Resume execution

Fill On Demand (Demand paging)

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid (not present)
  - When a page is referenced for first time, page fault
  - Kernel brings page in from disk into memory
  - Resume execution
  - Remaining pages can be transferred in the background while program is running

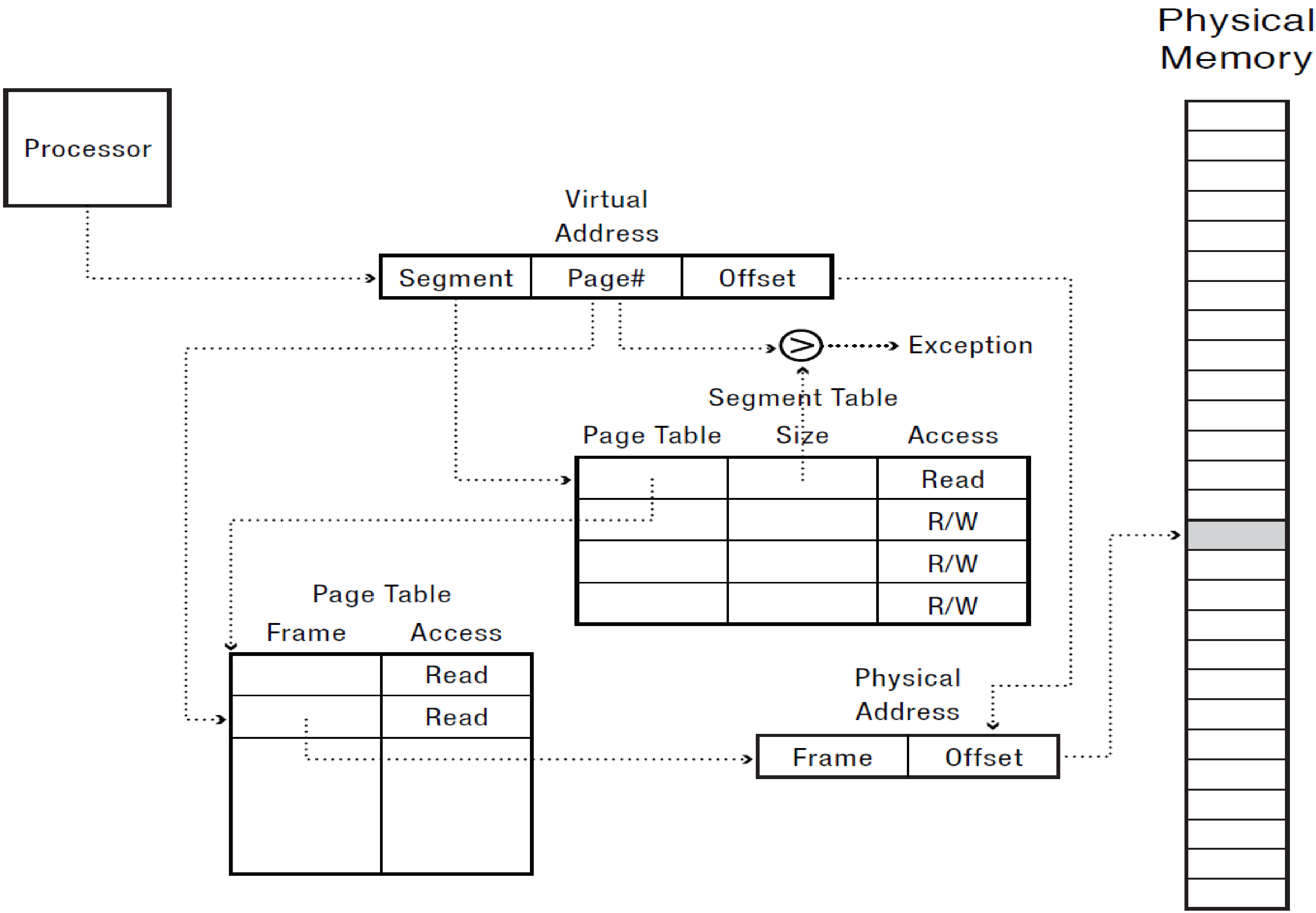
Multi-Level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient disk transfers (fixed size units)
    - page = page frame = sector
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
  - Variable granularity for protection/sharing

Paged Segmentation

- Process memory is segmented
- Segment is multiple of pages
- Segment table entry (in hardware):
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry (in memory):
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Paged Segmentation



Multi-Level Translation

- Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Space overhead: one pointer per virtual page
  - Two (or more) lookups per memory reference