Memory management 4

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

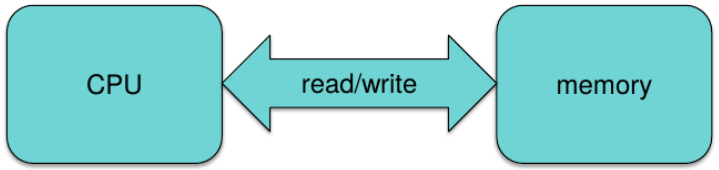
Previous lecture (s)

- Memory allocation
- Paging
  - Paging divides the memory into pages/page frames
  - Enables mapping logical addresses into physical addresses at the page granularity
    - High-degree of flexibility for OSs
    - Causes more memory accesses
  - Relies on hardware (MMU) and in-memory configuration structure (page table)

Recall: CPU access to memory

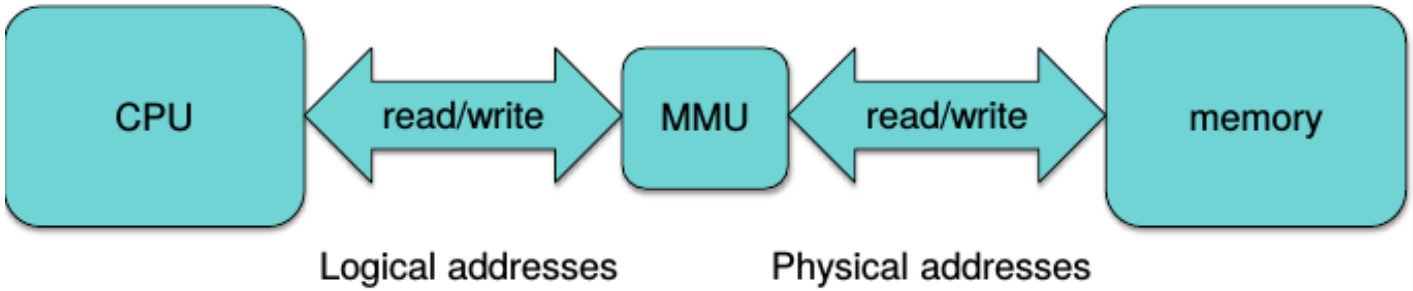
- The CPU reds instructions and reads/writes data from/to memory
- Functional interface:

  - `value = read(address)`
  - `write(address, value)`



Recall: Logical addressing

- Memory management unit (MMU()
  - Real-time, on demand translation between logical (virtual) and physical addresses



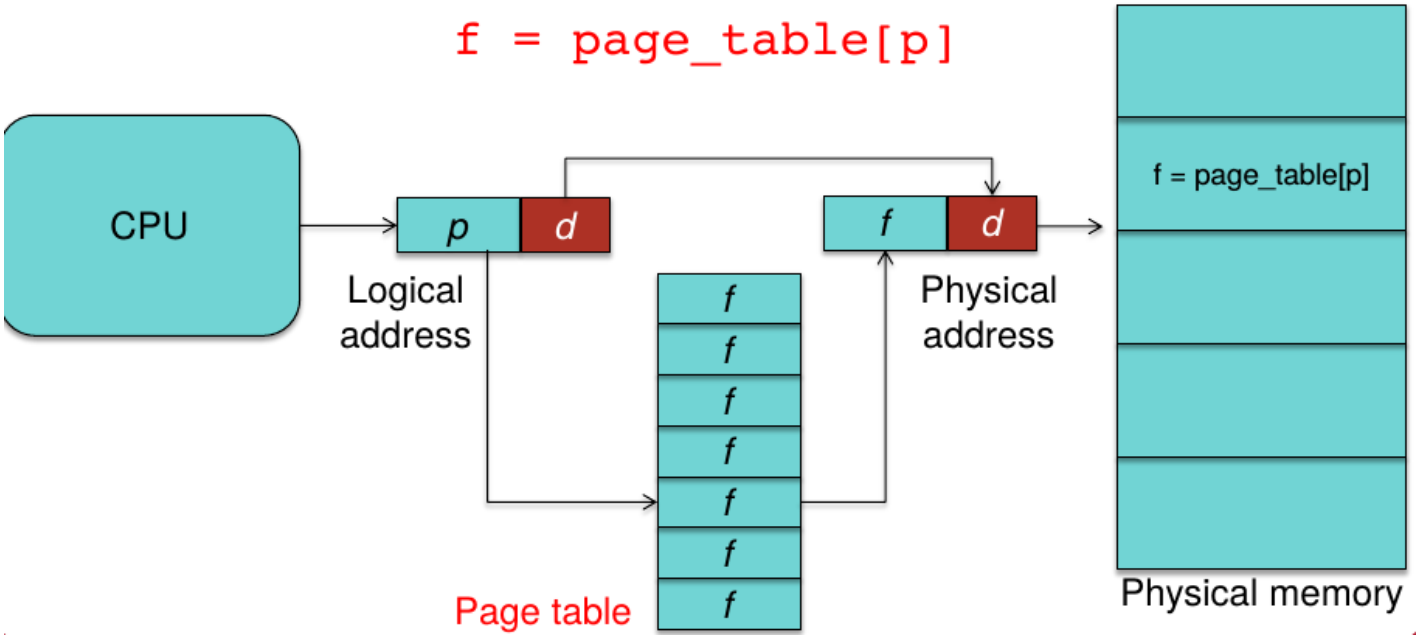
Recall: Paging

- Memory management scheme
  - Physical space can be non-contiguous
  - No fragmentation problems
  - No need for compaction
- Paging is implemented by the MMU (in the CPU)

Recall: Paging

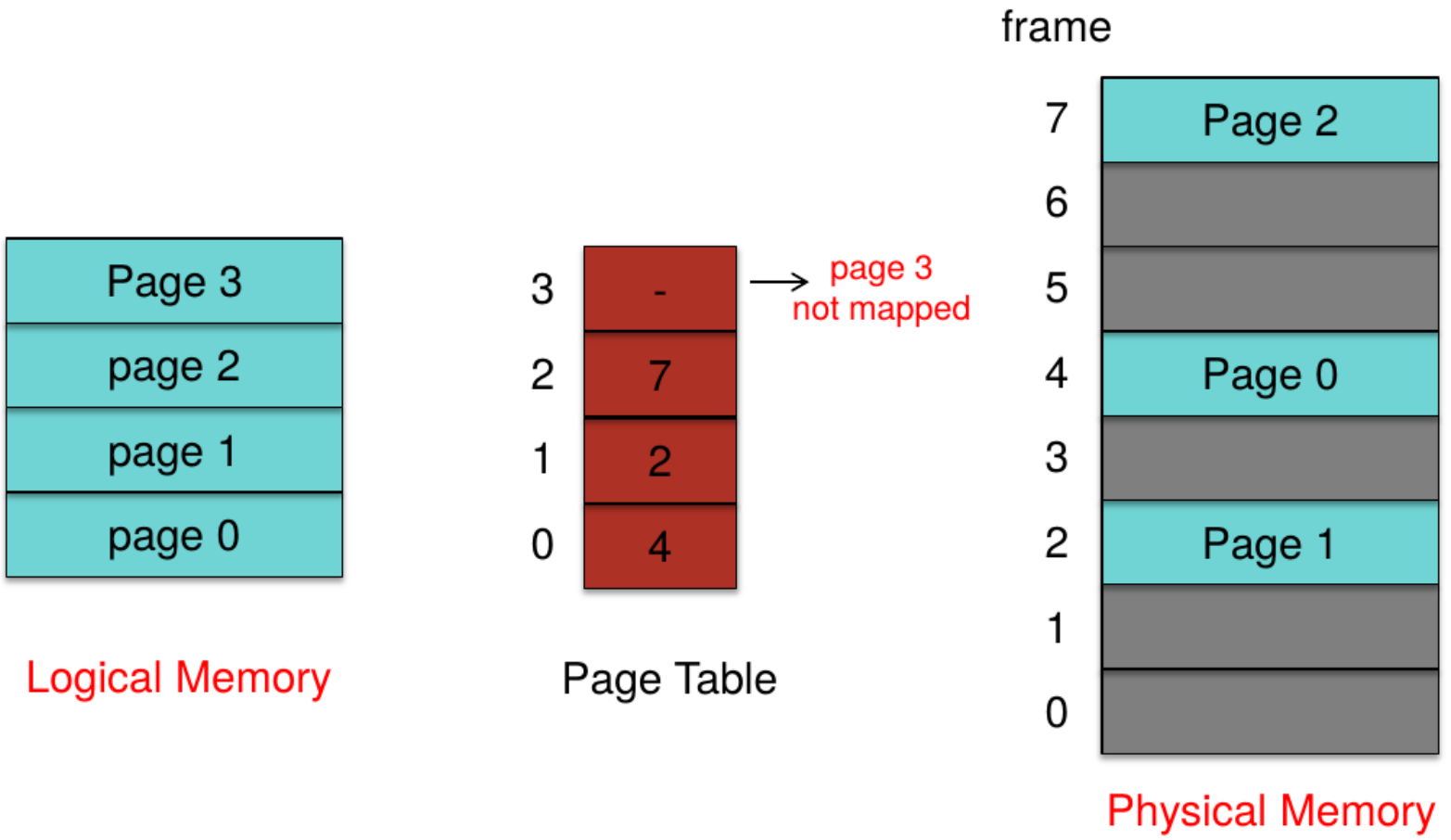
- Translation:
  - Divide physical memory into fixed-size blocks: **page frames**
  - A logical address is divided into blocks of the same size: **pages**
  - All memory accesses are translated:
    - page (virtual) -> page frame (physical)
    - The mapping is stored in a **page table**
- Example:
  - 32-bit address, 4KB page size:
    - Top 20 bits identify the **page number**
    - Bottom 12 bits identify offset with the page or page frame

Recall: Page translation



- p: upper 20-bits in a logical address, page index
- d: lower 12-bits in a logical address, offset within the page or page frame
- f: page frame index

Recall: Logical vs. physical views of memory



Recall: Hardware-assisted page translation

- Where do you keep the page table?
  - In memory
- How do we find the page table?
  - Page table base register: **CR3** register on x86 architecture
- Software-based translation can be slow!
  - To read a byte of memory, we need to read the page table first!
  - Each memory access would now be x2 (or more) times slower
- Hardware-assisted translation
  - **MMU** and **TLB**

Hardware acceleration: TLB

- Cache mappings for frequently accessed pages:
  - Translation Lookaside Buffer (TLB)
  - Associative memory: key (page #) and value (frame #)
- TLB hit vs miss
  - If there is a TLB miss => need to perform the page table lookup in RAM
- TLB is on-chip and fast, but small (from 64 to 1024 entries)
  - Locality (temporal and spatial) may help to keep good TLB hit rate

Address space identifiers: Tagged TLB

- There is only one TLB per system
  - But there can be multiple address spaces (per process)
- When we context switch, we switch address spaces
  - Some TLB entries may belong to the old address space
- Solutions:
  - Flush / invalidate the entire TLB
  - Have a Tagged TLB with and address space identifier

MMU

- Memory management unit (MMU):
  - A hardware unit (typically on-chip)
  - Walks the page table
- An MMU can also **enforce memory protection**
  - Page table stores status & protection bits per page
  - **valid/invalid:** is there a frame mapped to this page
  - **read-only**:
  - **no-execute**
  - **kernel only access**
  - **dirty**: the page has been modified since the flag was cleared
  - **accessed**: the page has been accessed since the flag was cleared

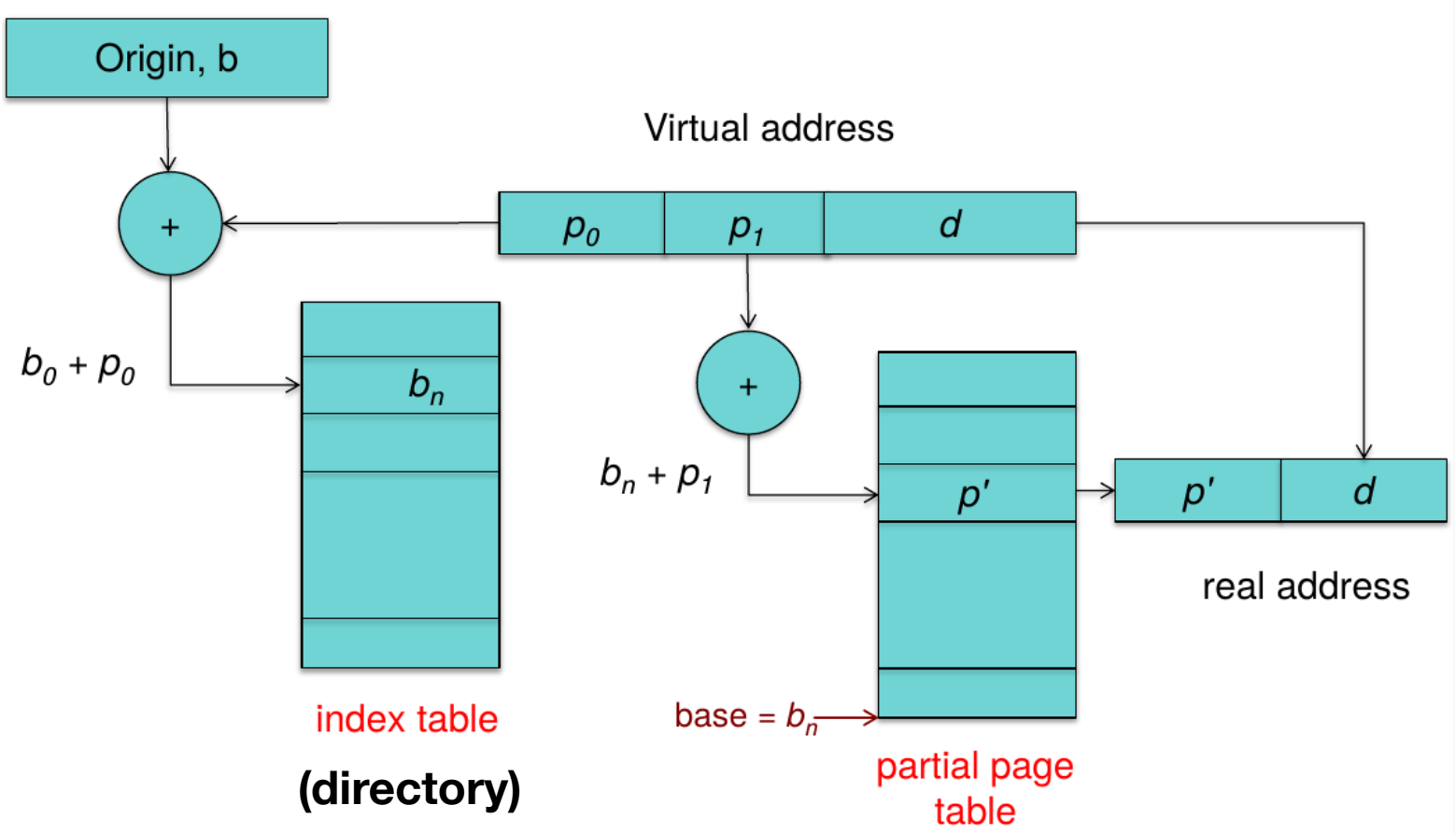
How much space used by the page table?

- Example: 32-bit system with 4KB pages: 20-bit page table
- ==> 2^20 = 1048576 entries in the page table

Multi-level (hierarchical) page tables

- Most processes use only a small part of their address space
- Keeping a (single-level) page table for the entire address space is wasteful
  - Example:
    - 32-bit system with 4KB pages: 20-bit page table
    - ==> 2^20 = 1048576 entries in the page table

Multilevel page table example



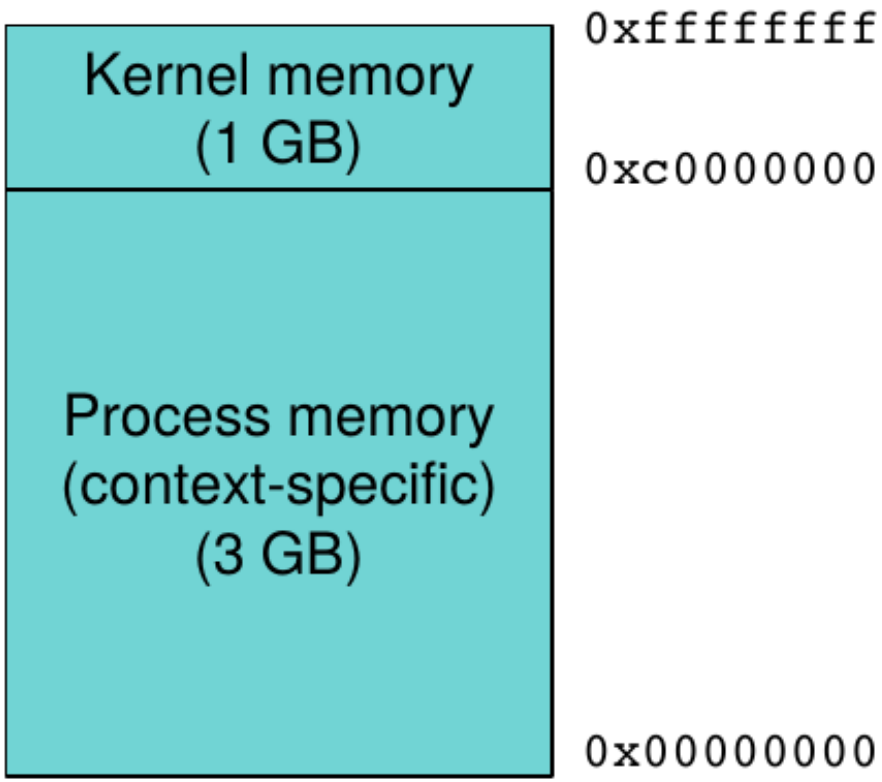
Page-based virtual memory benefits

- Allow discontinuous allocation
  - Simplify memory management for multiprogramming
  - MMU gives the illusion of contiguous allocation of memory
- Process can get memory any where in the address space
  - Allow a process to “feel” that it has more memory than it really has
  - Process can have greater address space than system memory
- Enforce memory protection
  - Each process’s address space is separated from others
  - MMU allows pages to be protected

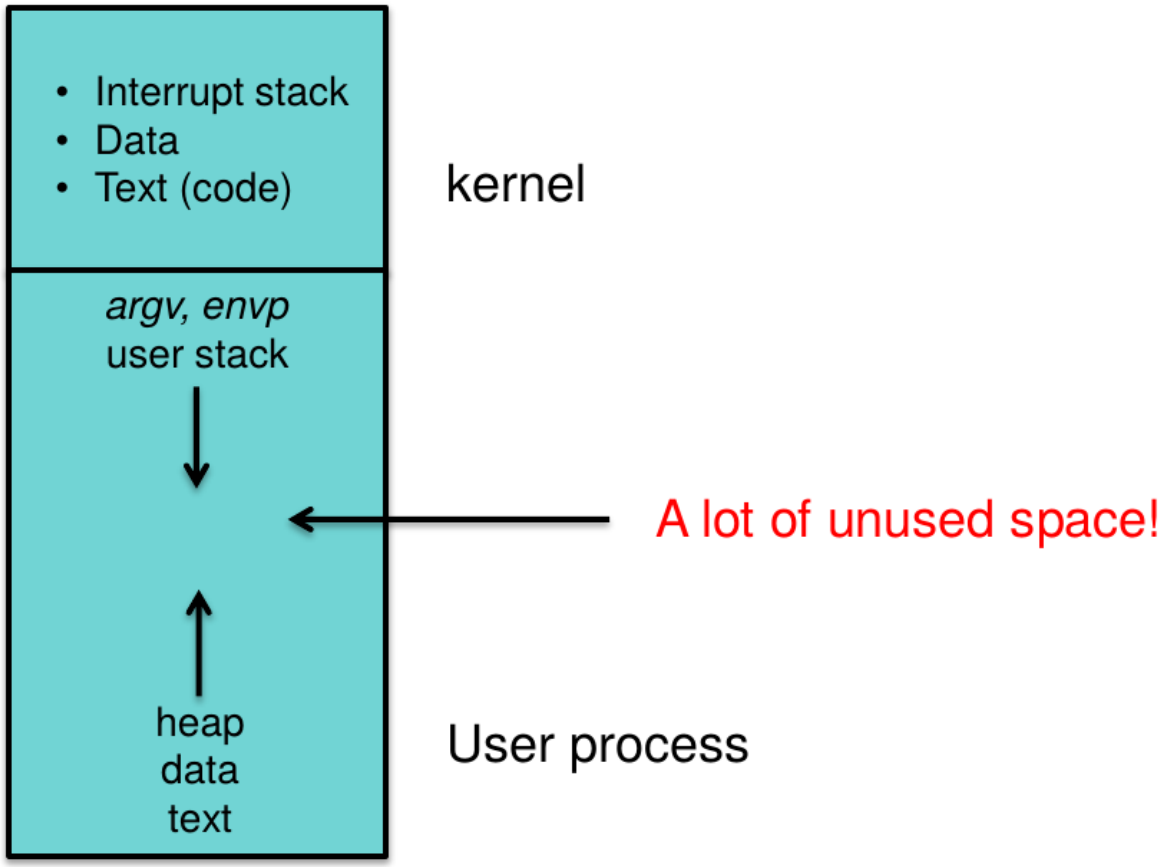
Typical kernel's view of memory

- A process sees a flat linear address space
  - Accessing regions of memory mapped to the kernel causes a page fault
- Kernel's view:
  - Address space is divided into two parts:
    - User part: changes with context switches
    - Kernel part: remains constant across context switches
  - Split is configurable:
    - 32-bit x86: 3GB for user + 1 GB for kernel

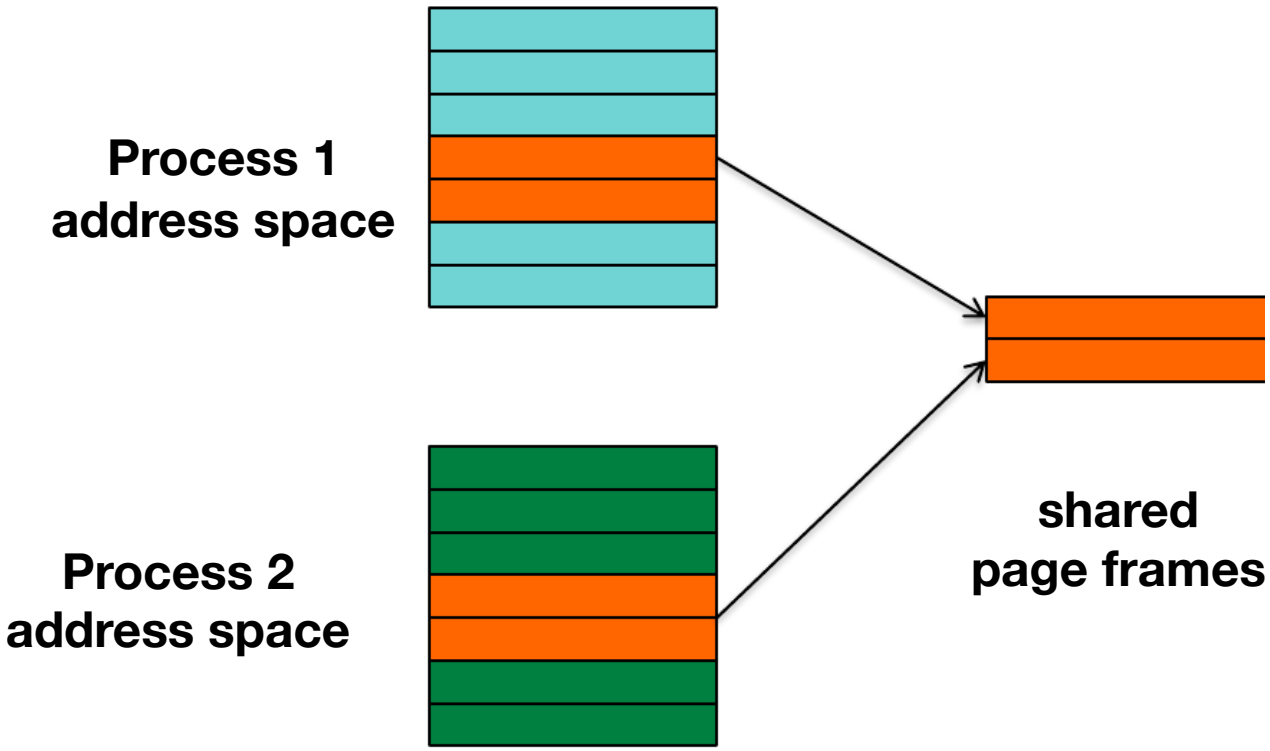
Example


Kernel memory
(1 GB)
Process memory
(context-specific)
(3 GB)
0xffffffff
0xc0000000
0x00000000

Example:
Sample per-process memory map



Virtual memory makes sharing easy



Copy on write

- Share until a page gets modified
- Q: When would this be useful?

Copy on write use case example

- Example: fork()
  - Set all pages to read-only
  - Trap on write
  - If legitimate write:
    - Allocate a new page and copy contents from the original

- Q: Other related use-cases for memory protection?
  - Zero-on-reference

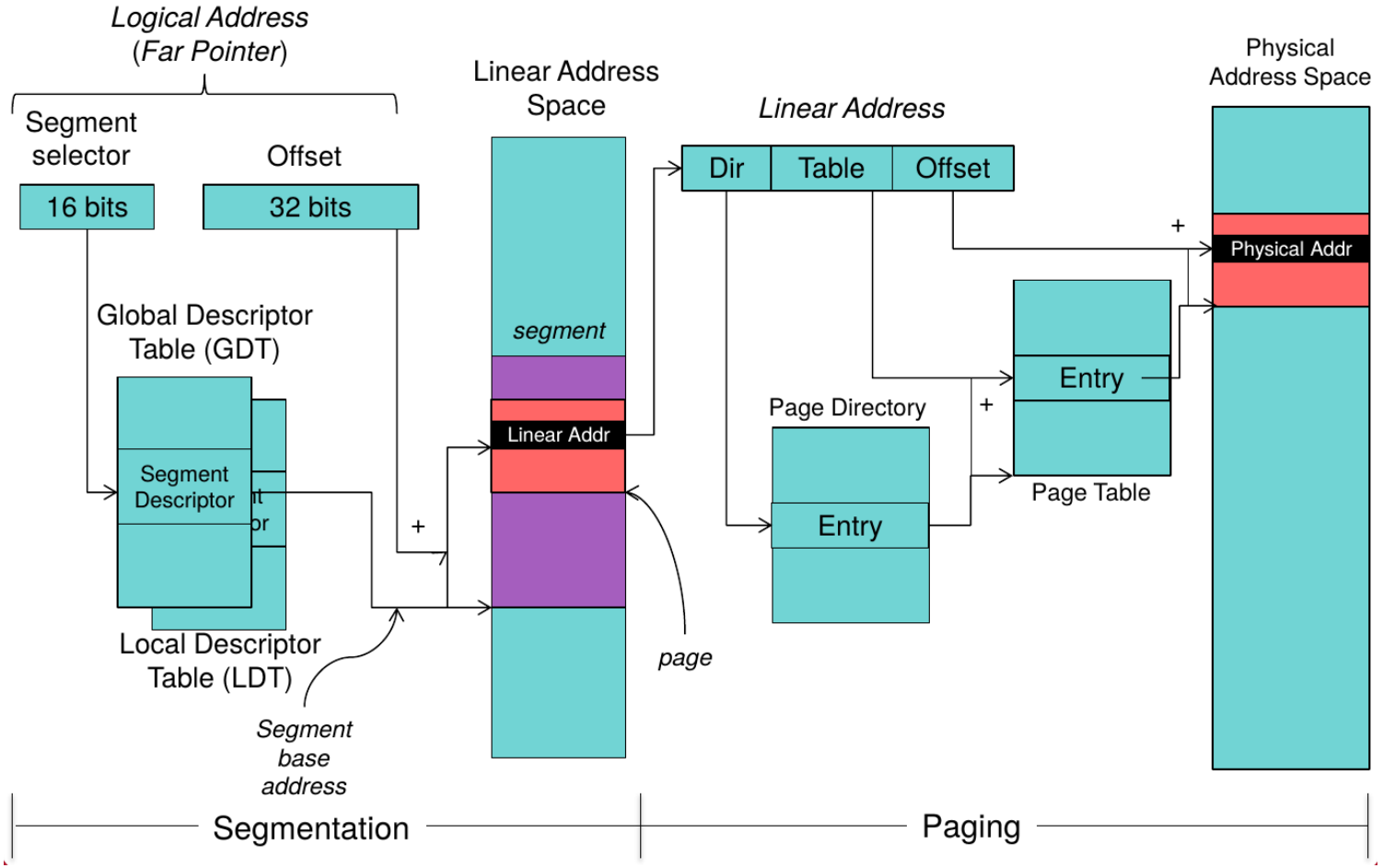
x86 microprocessor modes

- Real mode (IA-32):
  - Memory model: “**Segmentation**”
  - Logical address is expressed as A:B
    - **A**: Segment (value **in** segment register)
    - **B**: Offset within the segment
  - Physical address = A * 0x10 + B
  - Segment registers:
    - CS for code, DS for data, SS for stack, ES, FS, and GS
    - Stores the base address of the segment
  - Real mode is rarely used nowadays, apart from boot

x86 microprocessor modes

- Protected mode (IA-32):
  - Memory model: **Segmentation + Paging**
  - Logical address is expressed as A:B
    - A: Segment selector (Index of Global Descriptor Table)
    - B: Offset within the segment (32-bits)
  - Logical address = Segment base address (specified by GDT[A]) + B
  - The paging mechanism can be optionally enabled

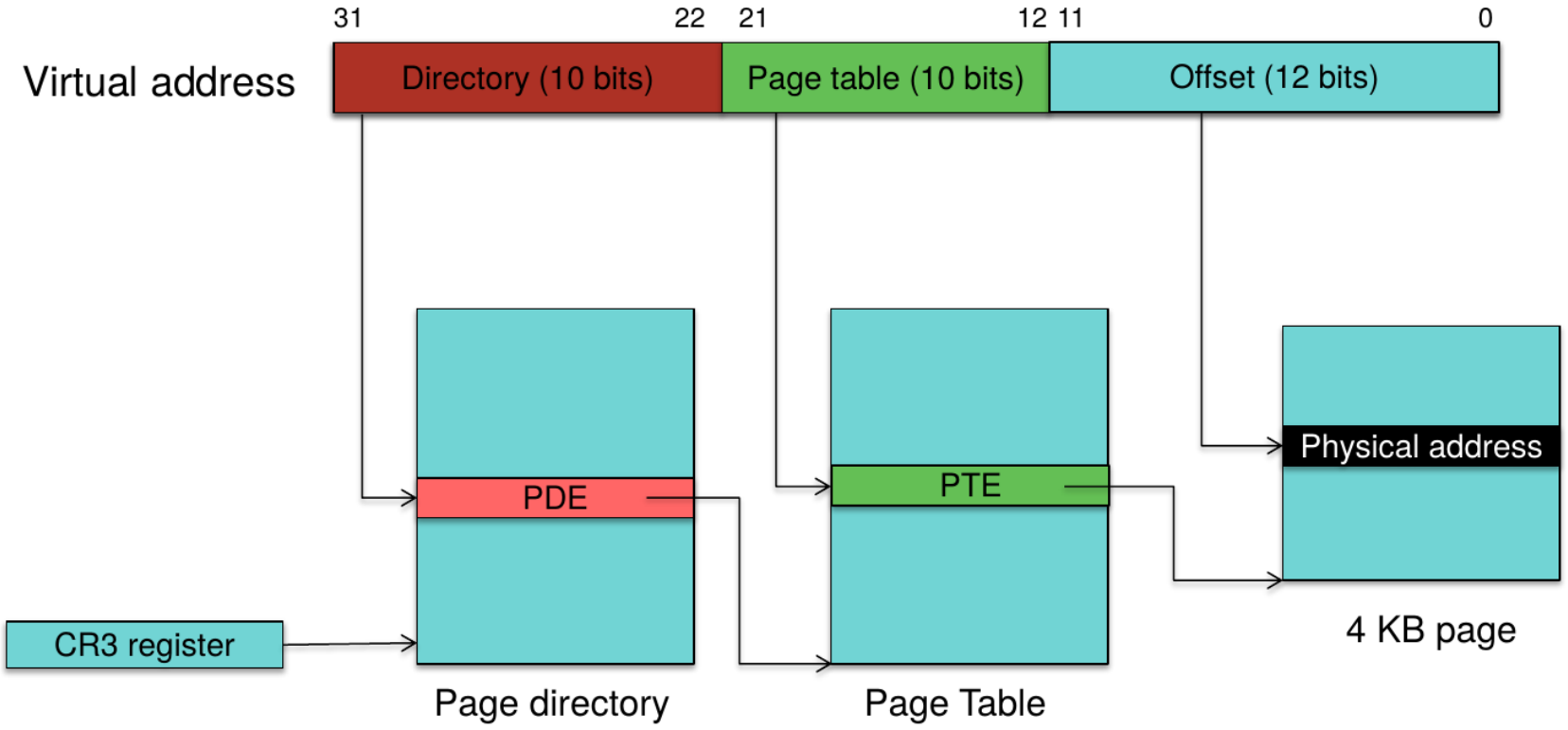
IA-32 segmentation + paging



x86 microprocessor modes

- Long mode (x86-64):
  - The memory model that most of you x86-64 computers run on
  - Segmentation is almost obsolete:
    - Except for FS and GS
    - Use flat memory model (i.e., descriptor base = 0, limit = max)
  - Check more details:
    - http://wiki.osdev.org/Segmentation
    - http://wiki.osdev.org/X86-64

32-bit paging with 4KB pages



Example: TLBs on the Core i7

- 4 KB pages:
  - Instruction TLB: 128 entries per core
  - Data TLB: 64 entries
- Old CPUs: Cache and TLB specs core 2 Duo

64-byte Prefetching
Data TLB: 4-KB Pages, 4-way set associative, 256 entries
Data TLB: 4-MB Pages, 4-way set associative, 32 entries
Instruction TLB: 2-MB pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
Instruction TLB: 4-KB Pages, 4-way set associative, 128 entries
L1 Data TLB: 4-KB pages, 4-way set associative, 16 entries
L1 Data TLB: 4-MB pages, 4-way set associative, 16 entries

http://www.cpu-world.com/sspec/SL/SLB9L.html