Operating Systems

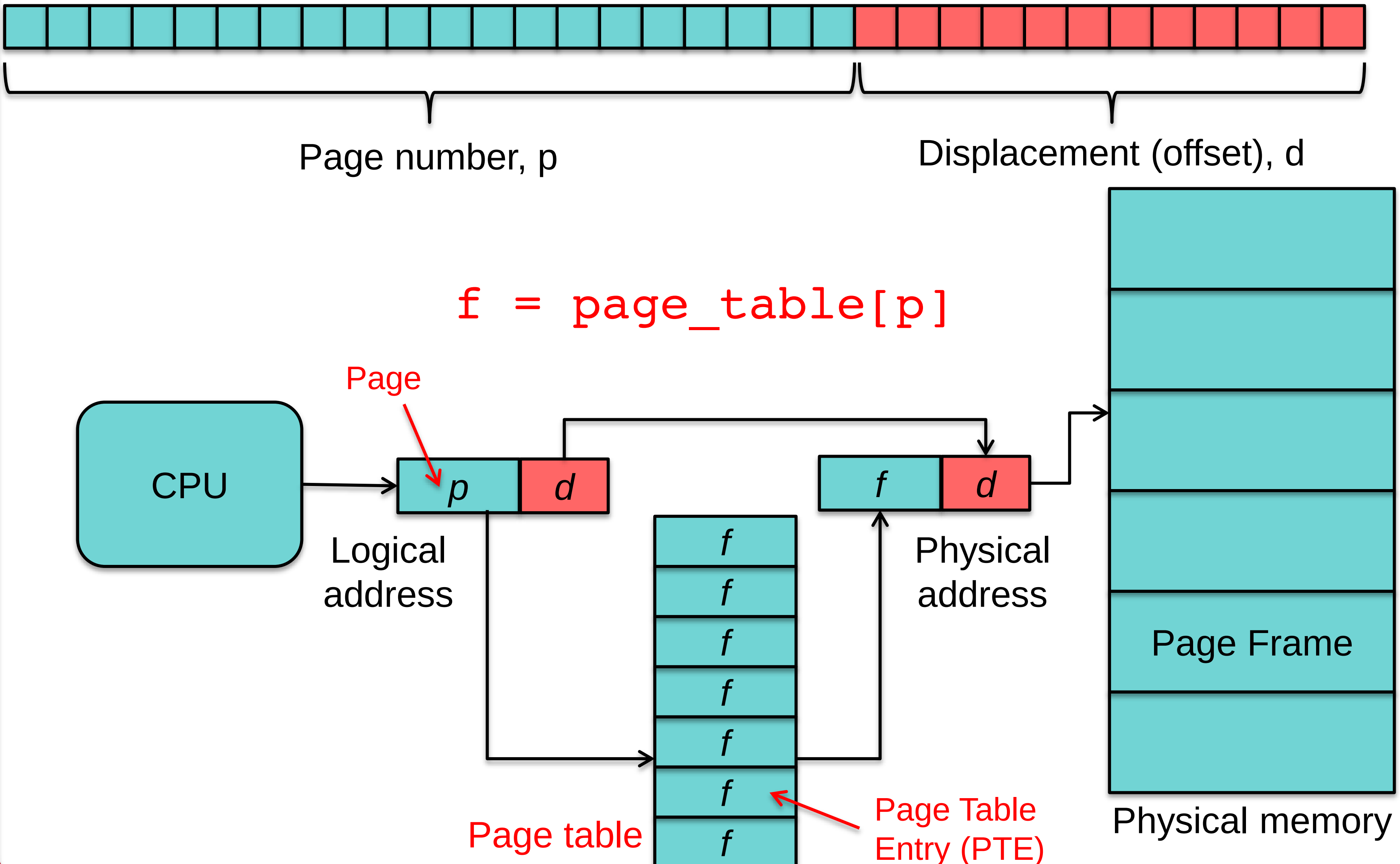
10. Memory Management – Part 2
Paging

Paul Krzyzanowski

Rutgers University

Spring 2015

Page translation



Page table

- One page table per process
  - Contains page table entries (PTEs)

- Each PTE contains
  - Corresponding page frame # for a page #
  - Permissions
    - Permissions (read-only, read-write, execute-only, privileged access only…)
  - Access flags
    - *Valid?* Is the page mapped?
    - *Modified?*
    - *Referenced?*

- Page table is selected by setting a page table base register with the address of the table

Accessing memory

- CPU starts in physical addressing mode
  - Someone has to set up page tables
  - Divide address space into user & kernel spaces
  - Switch to Virtual addressing mode
- Each process makes *virtual* address references for all memory access
- MMU converts to physical address via a per-process page table
  - Page number → Page frame number
  - Page fault trap if not a valid reference

Improving look-up performance: TLB

- Cache frequently-accessed pages
  - Translation lookaside buffer (TLB)
  - Associative memory: key (page #) and value (frame #)

- TLB is on-chip & fast … but small (64 – 1,024 entries)
- TLB miss: result not in the TLB
  - Need to do page table lookup in memory
- Hit ratio = % of lookups that come from the TLB
- Address Space Identifier (ASID): share TLB among address spaces

Page-Based Virtual Memory Benefits

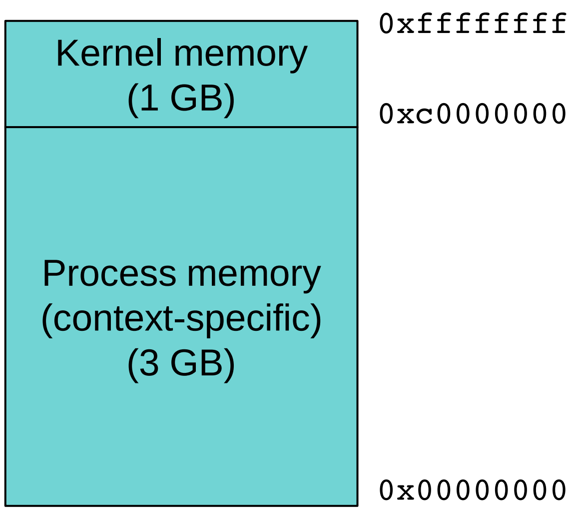
- Allow discontiguous allocation
  - Simplify memory management for multiprogramming
  - MMU gives the illusion of contiguous allocation of memory

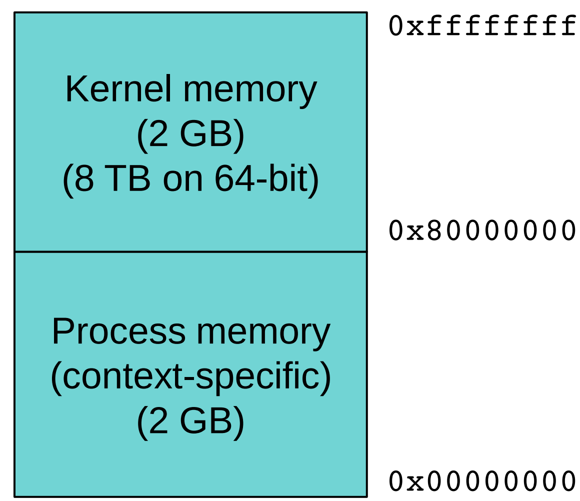
- Process can get memory anywhere in the address space
  - Allow a process to feel that it has more memory than it really has
  - Process can have greater address space than system memory

- Enforce memory Protection
  - Each process' address space is separate from others
  - MMU allows pages to be protected:
    - Writing, execution, kernel vs. user access

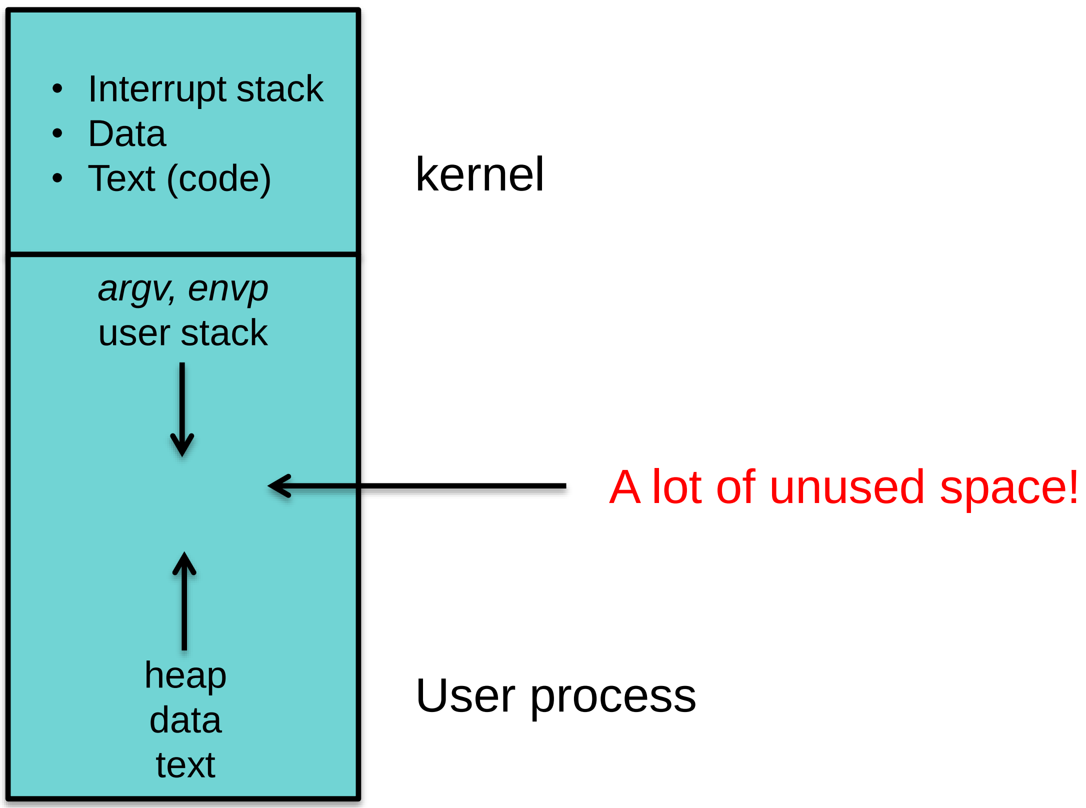
Kernel's view of memory

- 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 split into two parts
    - User part: changes with context switches
    - Kernel part: **remains constant across context switches**
  - Split is configurable:
    - 32-bit x86: PAGE_OFFSET: 3 GB for process + 1 GB kernel





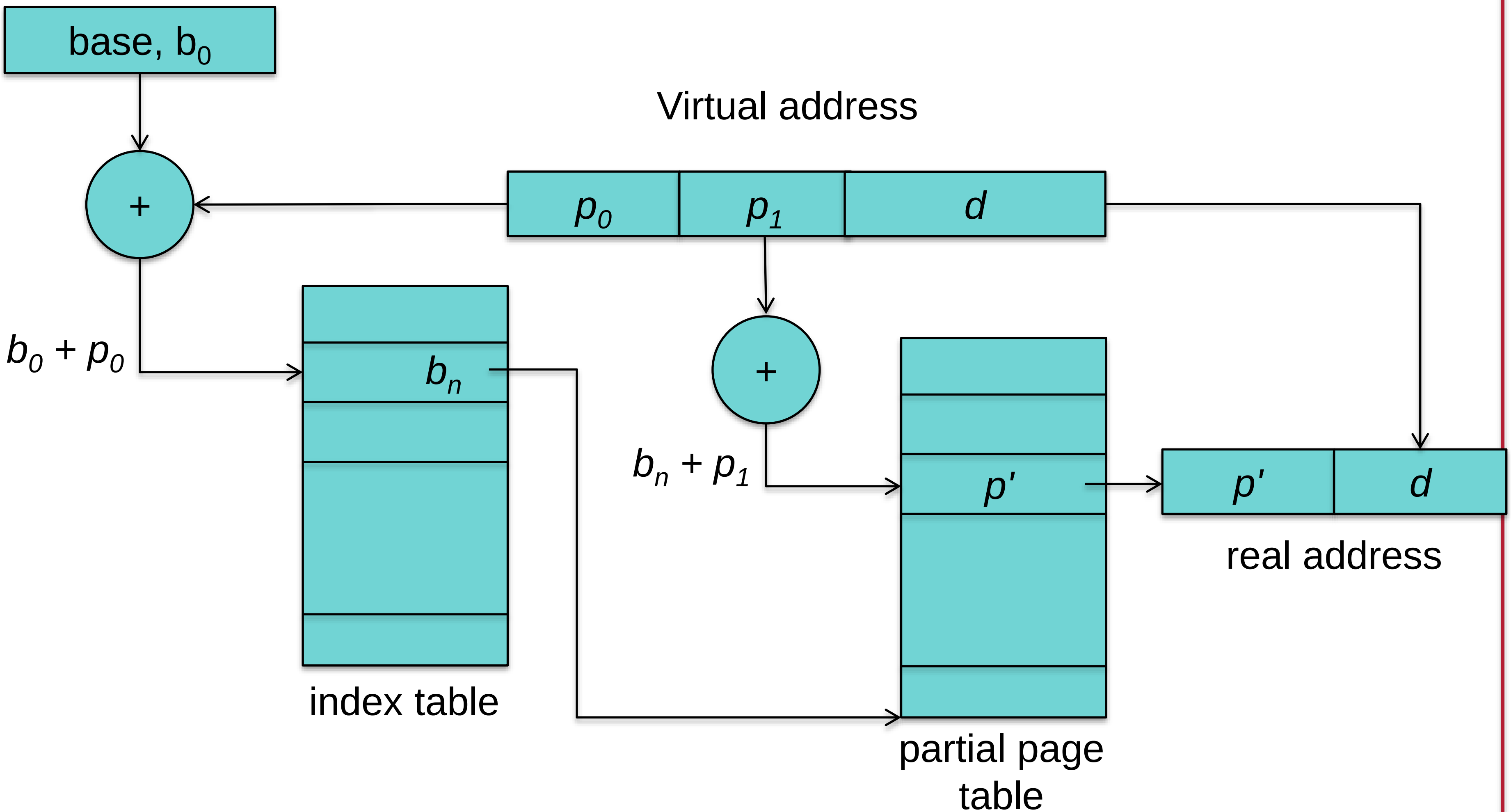
Sample memory map per process



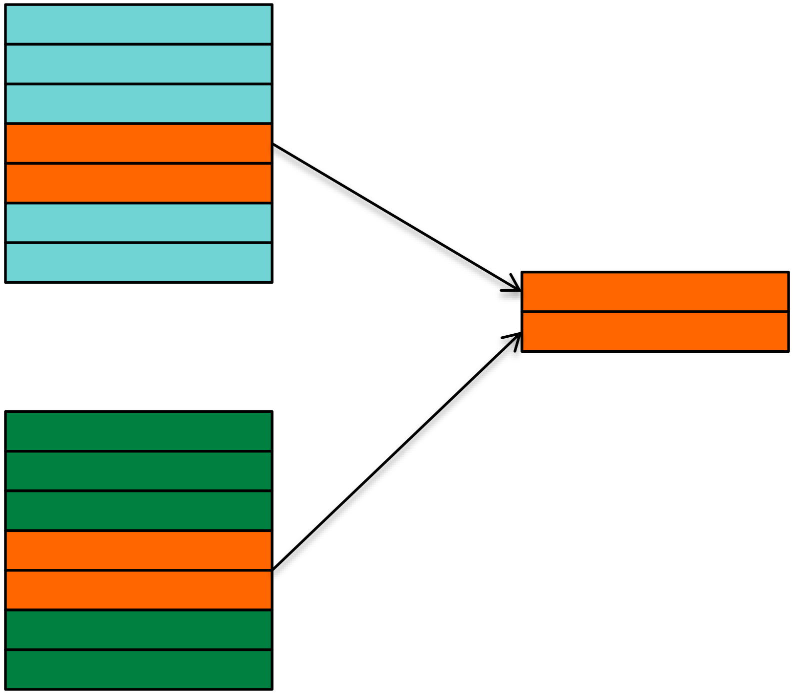
Multilevel (Hierarchical) page tables

- Most processes use only a small part of their address space
- Keeping an entire page table is wasteful
  - 32-bit system with 4KB pages: 20-bit page table
    $\Rightarrow 2^{20} = 1,048,576$ entries in a page table

Multilevel page table



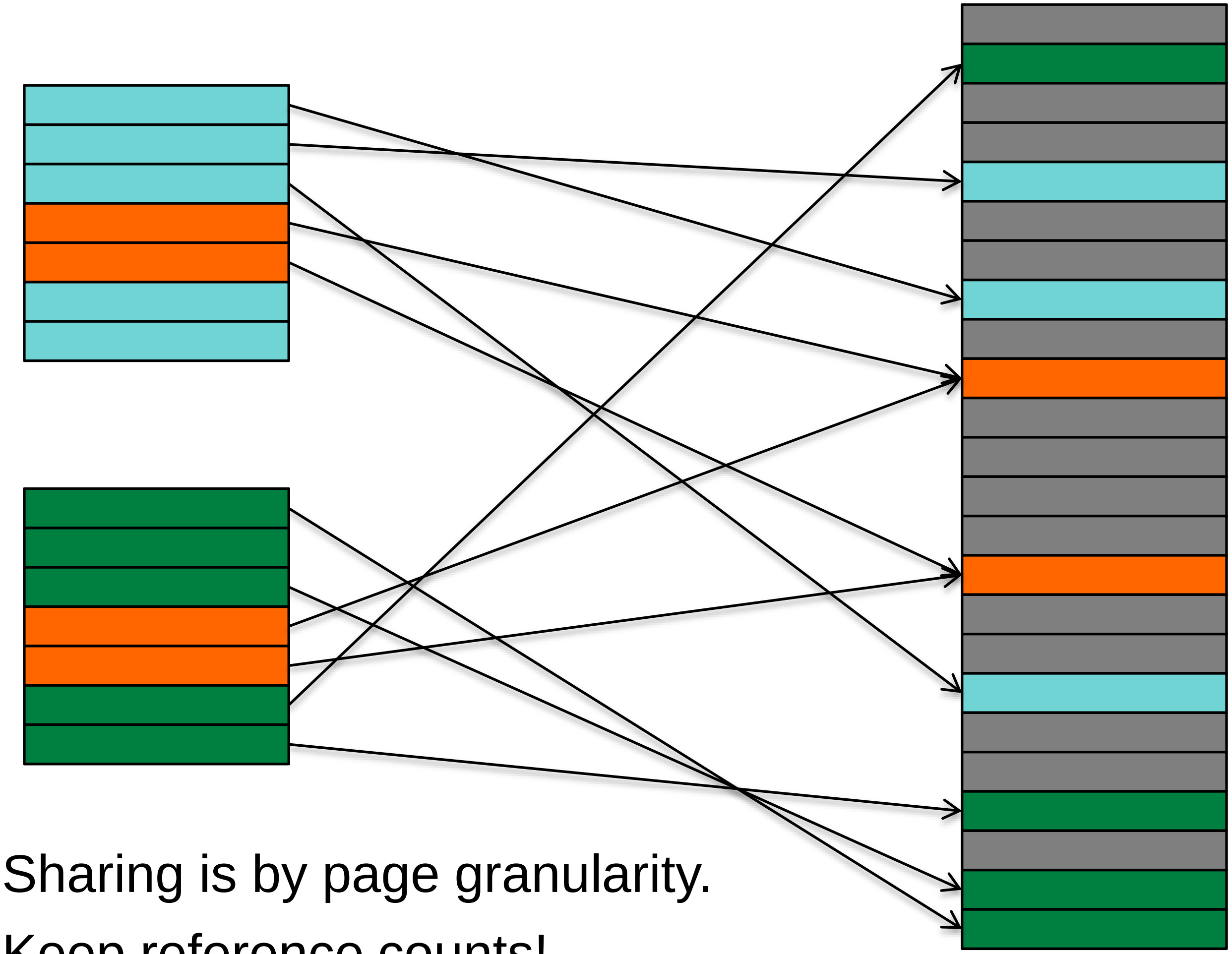
Virtual memory makes memory sharing easy



Shared library or
Shared memory

Sharing is by page granularity

Virtual memory makes memory sharing easy



Sharing is by page granularity.

Keep reference counts!

Copy on write

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

MMU Example: ARM

ARMv7-A architecture

- Cortex-A8
  - iPhone 3GS, iPod Touch 3G, Apple A4 processor in iPhone 4 & iPad, Droid X, Droid 2, etc.)
- Cortex-A9
  - Multicore support
  - TI OMAP 44xx series, Apple A5 processor in iPad 2
- Apple A6
  - 32-bit AMD Cortex-A15 processor
  - Used in iPhone 5, 5C, 4^{th} gen iPad
- Apple A7
  - 64-bit ARMv8-A architecture
  - Used in iPhone 5S, 2^{nd} gen iPad mini, iPad Air

Pages

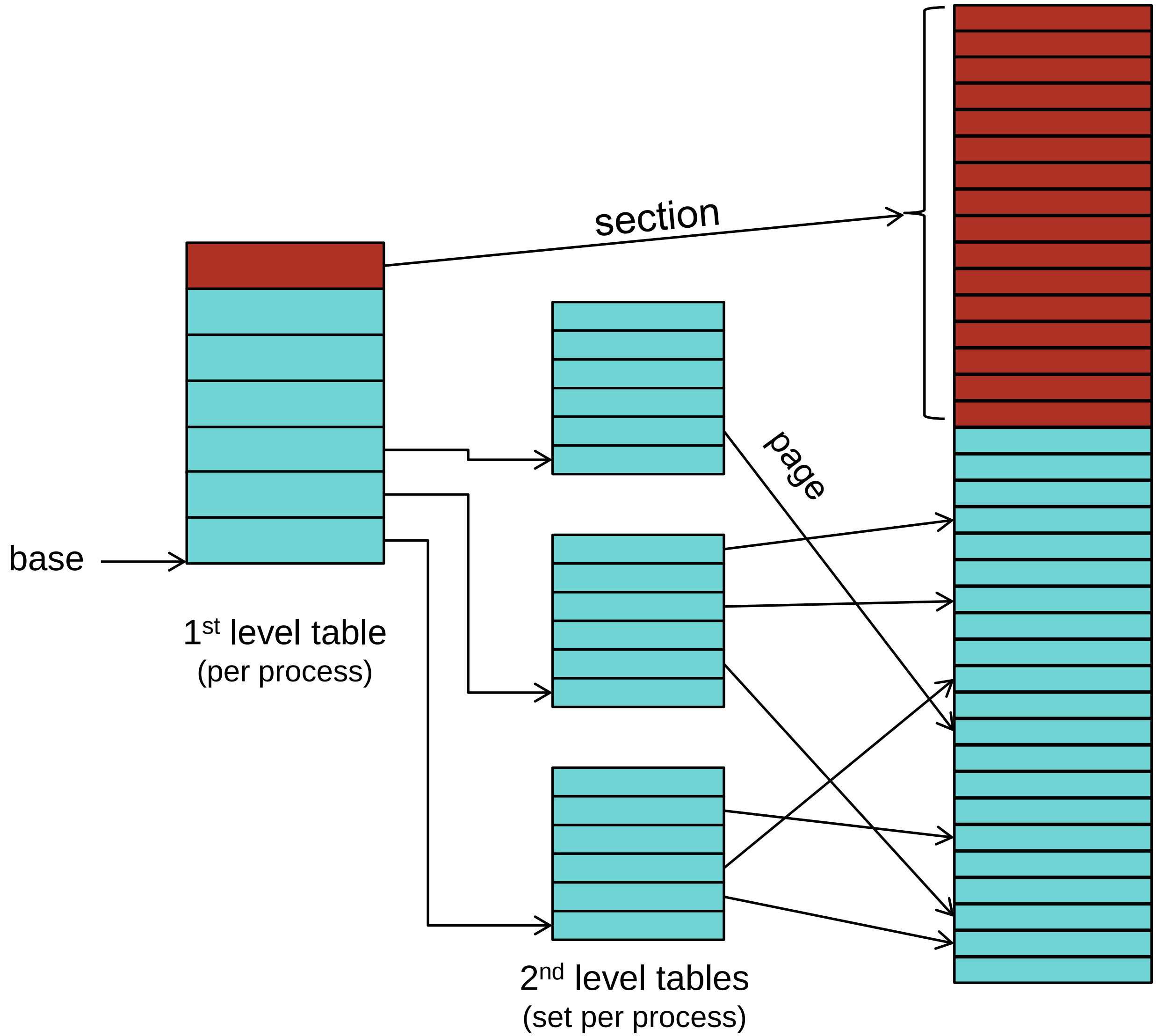
Four page (block) sizes:

- Supersections: 16MB memory blocks
- Sections: 1MB memory blocks
- Large pages: 64KB memory blocks
- Small pages: 4KB memory blocks

Two levels of tables

- First level table (*aka* translation tables)
  - Base address, descriptors, and translation properties for sections and supersections (1 MB & 16 MB blocks)
  - Translation properties and pointers to a second level table for large and small pages (4 KB and 64 KB pages)
- Second level tables (*aka* page tables)
  - Each contains base address and translation properties for small and large pages
- Benefit: a large region of memory can be mapped using a single entry in the TLB (e.g., OS)

ARM Page Tables

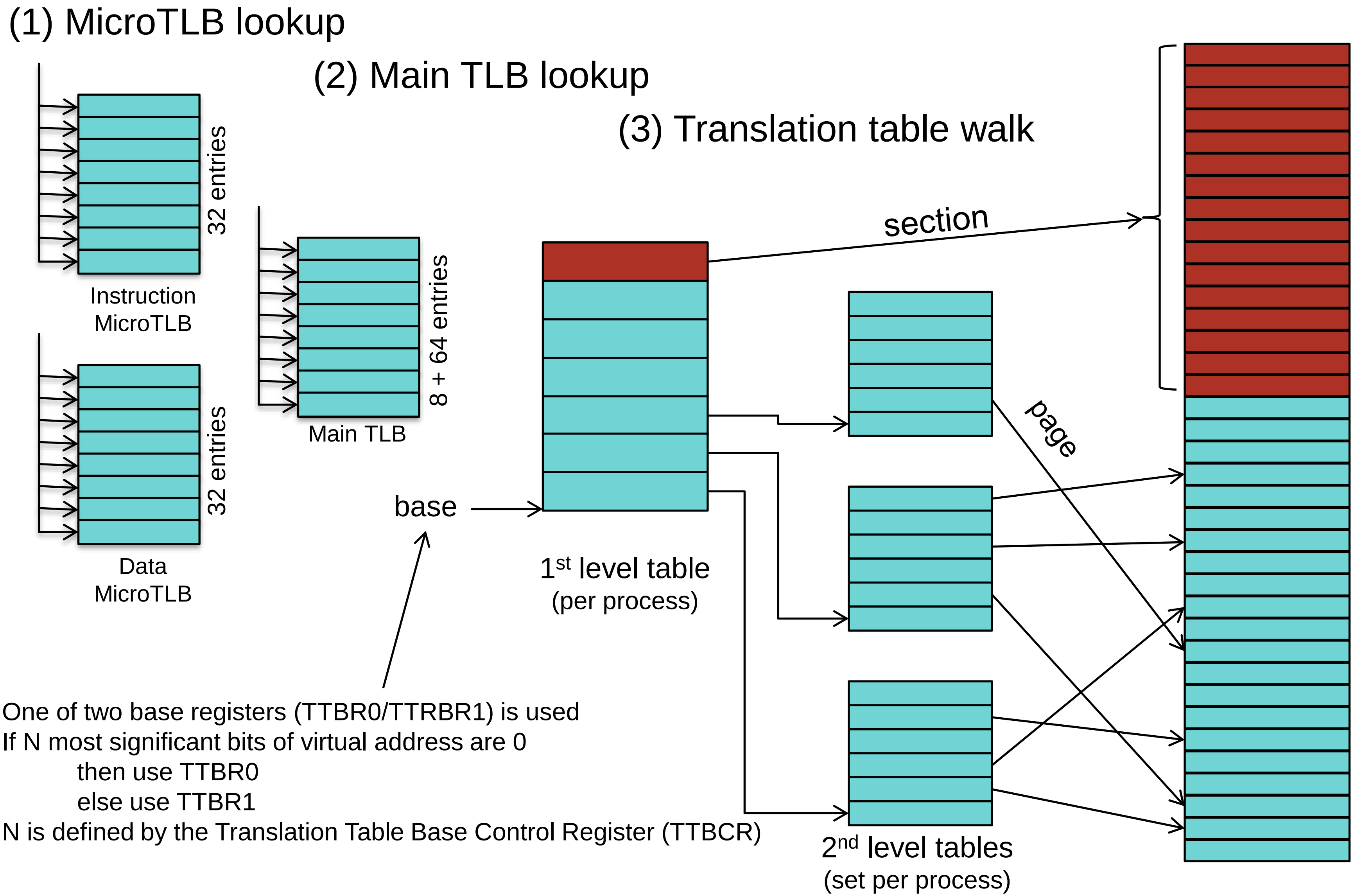


TLB

- 1st level: MicroTLB – one each for instruction & data sides
  - 32 entries (10 entries in older v6 architectures)
  - Address Space Identifier (ASID) [8 bits] and Non-Secure Table Identifier (NSTID) [1 bit]; entries can be global
  - Fully associative; one-cycle lookup
  - Lookup checks protection attributes: may signal Data Abort
  - Replacement either Round-Robin (default) or Random

- 2nd level: Main TLB – catches cache misses from microTLBs
  - 8 fully associative entries (may be locked) + 64 low associative entries
  - variable number of cycles for lookup
  - lockdown region of 8 entries (important for real-time)
  - Entries are globally mapped or associated ASID and NSTID

ARM Page Tables



One of two base registers (TTBR0/TTRBR1) is used
If N most significant bits of virtual address are 0
then use TTBR0
else use TTBR1
N is defined by the Translation Table Base Control Register (TTBCR)

Translation flow for a section (1 MB)

Virtual address

31 … 20	19 … 0
Table index (12 bits)	Section offset (20 bits)

Physical section = *read* [Translation table base + table index]

Physical address = physical section : section offset

Real address

31 … 20	19 … 0
Physical section (12 bits)	Section offset (20 bits)

Translation flow for a supersection (16 MB)

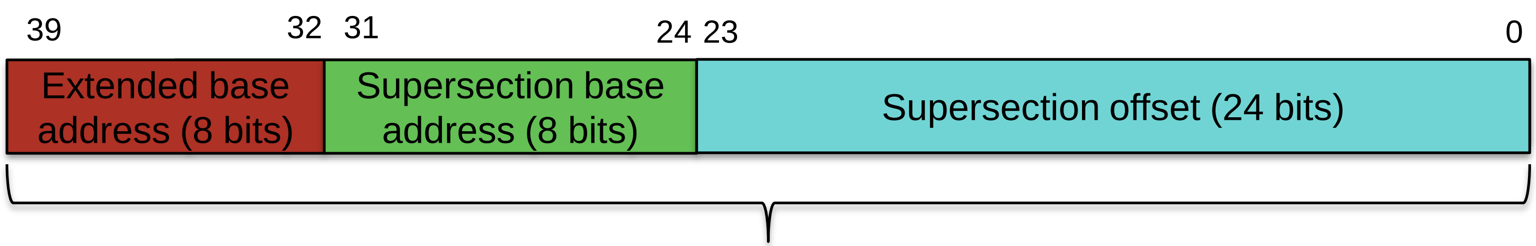
Virtual address

Supersection base address, Extended base address =

read [Translation table base + table index]

Real address = Extended base address : physical section : section offset

Real address



40 bit address
(1 TB)

Translation flow for a small page (4KB)

Virtual address	31 … 20	19 … 12	11 … 0
	First level index (12 bits)	Second-level index (8 bits)	Page offset (12 bits)

Page table address = *read* [Translation table base + first-level index]

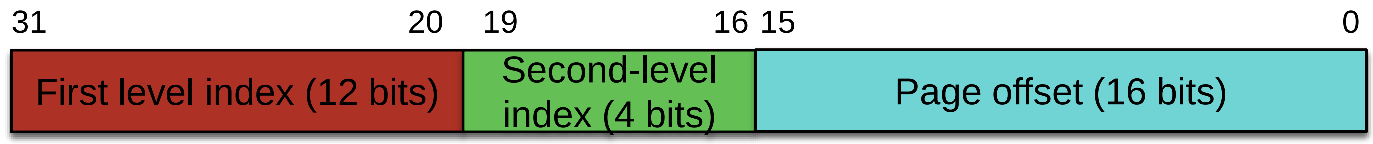
Physical page = *read*[page table address + second-level index]

Real address = physical page : page offset

Real address	31 … 12	11 … 0
	Physical page (20 bits)	Page offset (12 bits)

Translation flow for a large page (64KB)

Virtual address

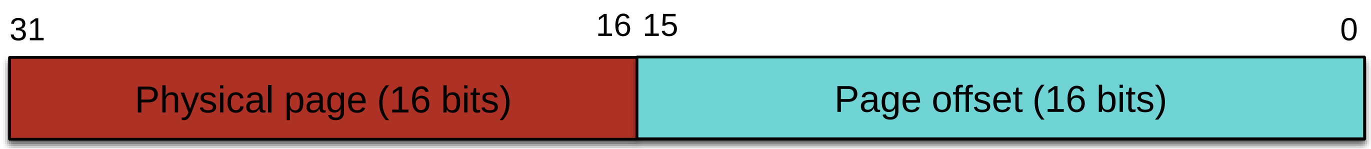


Page table address = Read [Translation table base + first-level index]

Physical page = read[page table address + second-level index]

Physical address = physical page : page offset

Real address



Memory Protection & Control

- Domains
  - Clients execute & access data within a domain. Each access is checked against access permissions for each memory block
- Memory region attributes
  - Execute never
  - Read-only, read/write, no access
    - Privileged read-only, privileged & user read-only
  - Non-secure (is this secure memory or not?)
  - Sharable (is this memory shared with other processors)
    - Strongly ordered (memory accesses must occur in program order)
    - Device/shared, device/non-shared
    - Normal/shared, normal/non-shared
- Signal *Memory Abort* if permission is not valid for access

MMU Example: x86-64

IA-32 Memory Models

- Flat memory model
  - Linear address space
  - Single, contiguous address space

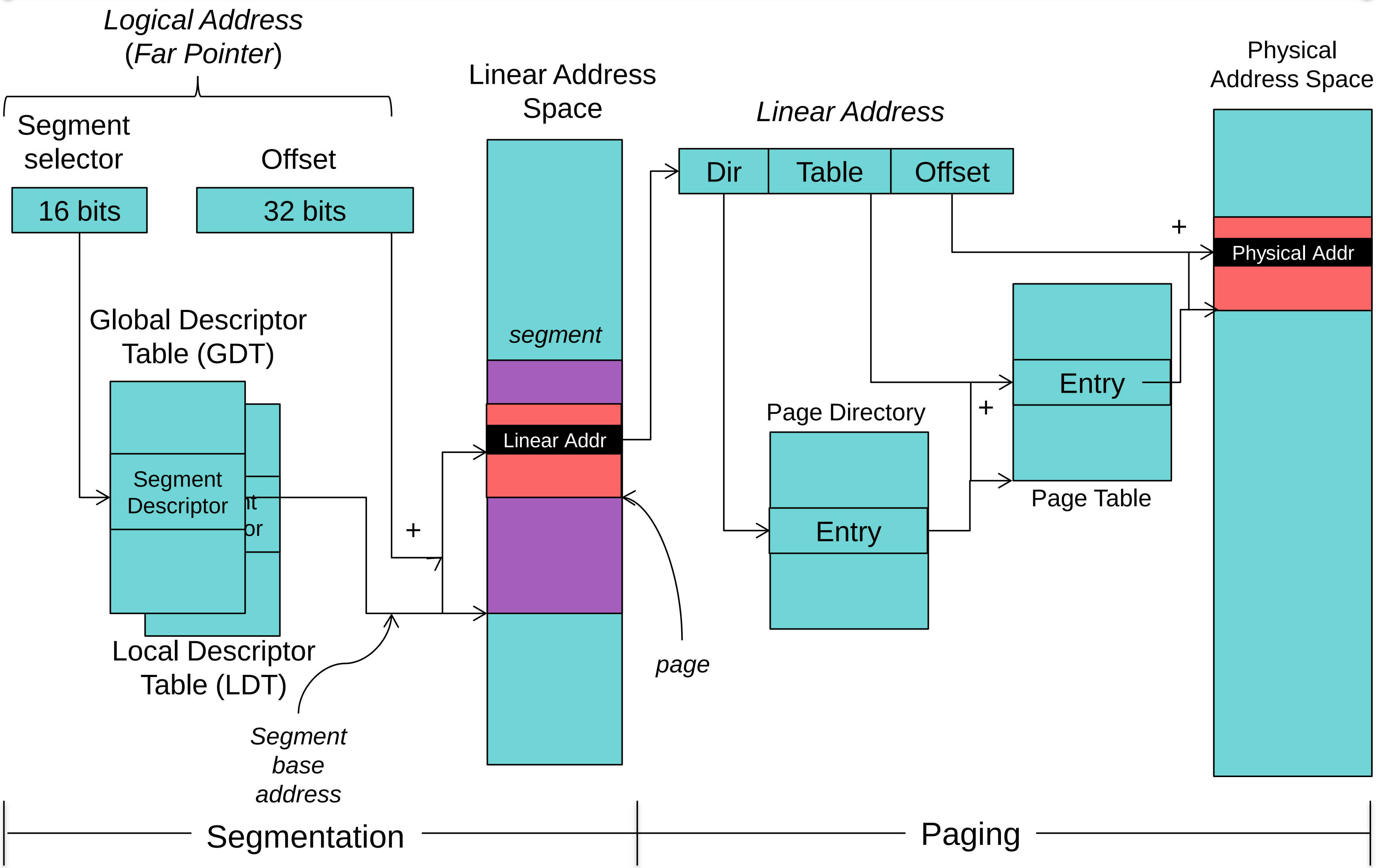
- Segmented memory model
  - Memory appears as a group of independent address spaces: segments (code, data, stack, etc.)
  - Logical address = {segment selector, offset}
  - 16,383 segments; each segment can be up to 2^{32} bytes

- Real mode
  - 8086 model
  - Segments up to 64KB in size
  - maximum address space: 2^{20} bytes

Segments

- Each segment may be up to 4 GB
- Up to 16 K segments per process
- Two partitions per process
  - Local: *private to the process*
    - Up to 8 K segments
    - Info stored in a Local Descriptor Table (LDT)
  - Global: *shared among all processes*
    - Up to 8 K segments
    - Info stored in a Global Descriptor Table (GDT)
- Logical address is (*segment selector, offset)*
  - Segment selector = 16 bits:
    - 13 bits segment number + 1 bit LDT/GDT ID + 2 bits protection

IA-32 Segmentation & Paging



Segment protection

- S flag in segment descriptor identifies *code* or *data* segment
- Accessed (referenced)
  - has the segment been accessed since the last time the OS cleared the bit?
- Dirty
  - Has the page been modified?
- Data
  - Write-enable
    - Read-only or read/write?
  - Expansion direction
    - Expand down (e.g., for stack): dynamically changing the segment limit causes space to be added to the bottom of the stack
- Code
  - Execute only, execute/read (e.g., constants in code segment)
  - Conforming:
    - Execution can continue even if privilege level is elevated

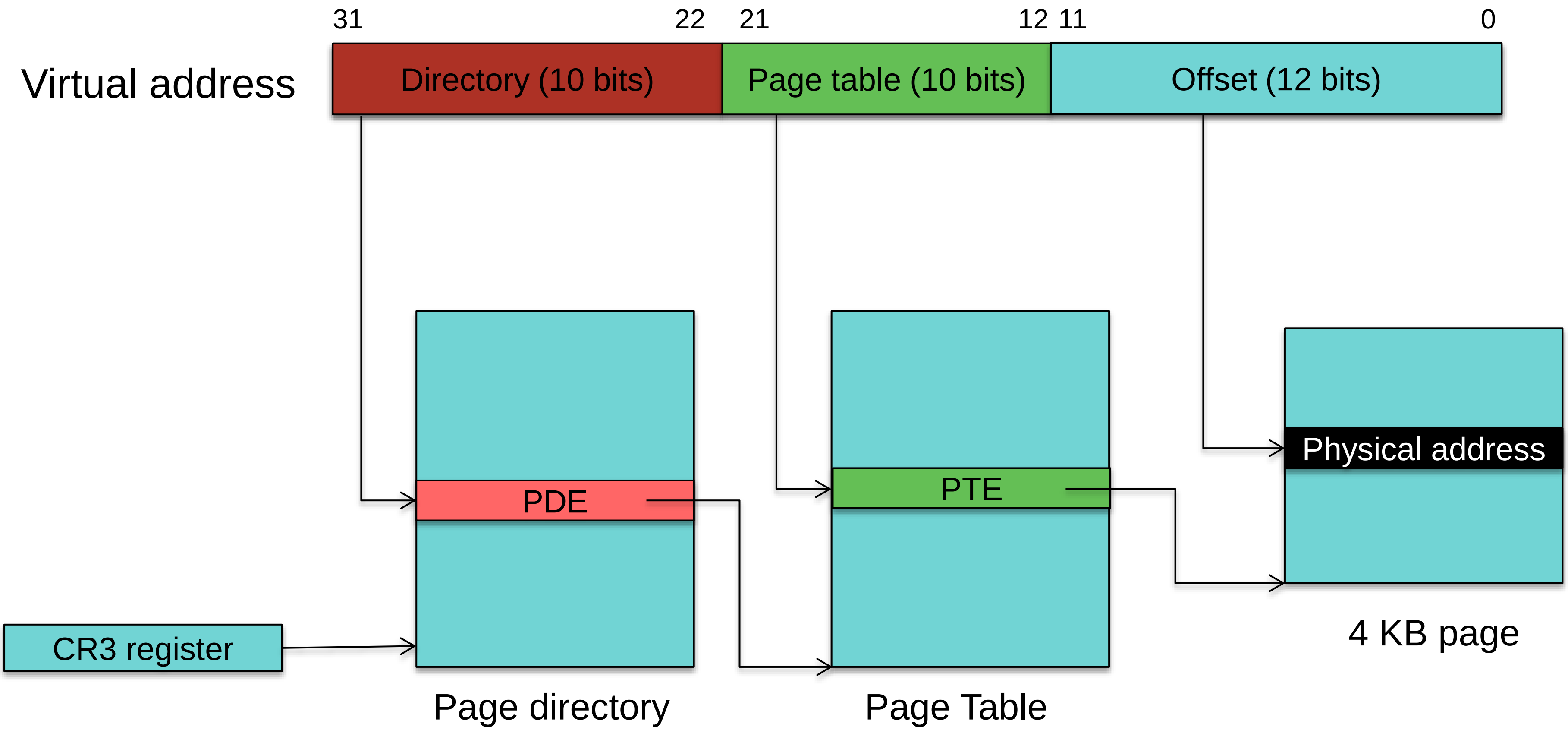
IA-32 Paging

- 32-bit registers, 36-bit address space (64 GB)
  - Physical Address Extension (PAE)
    - Bit 5 of control register CR4
    - 52 bit physical address support (4 PB of memory)
    - Only a 4 GB address space may be accessed at one time
  - Page Size Extensions (PSE-36)
    - 36-bit page size extension (64 GB of memory)
  - Supports up to 4 MB page size

Intel 64-bit mode

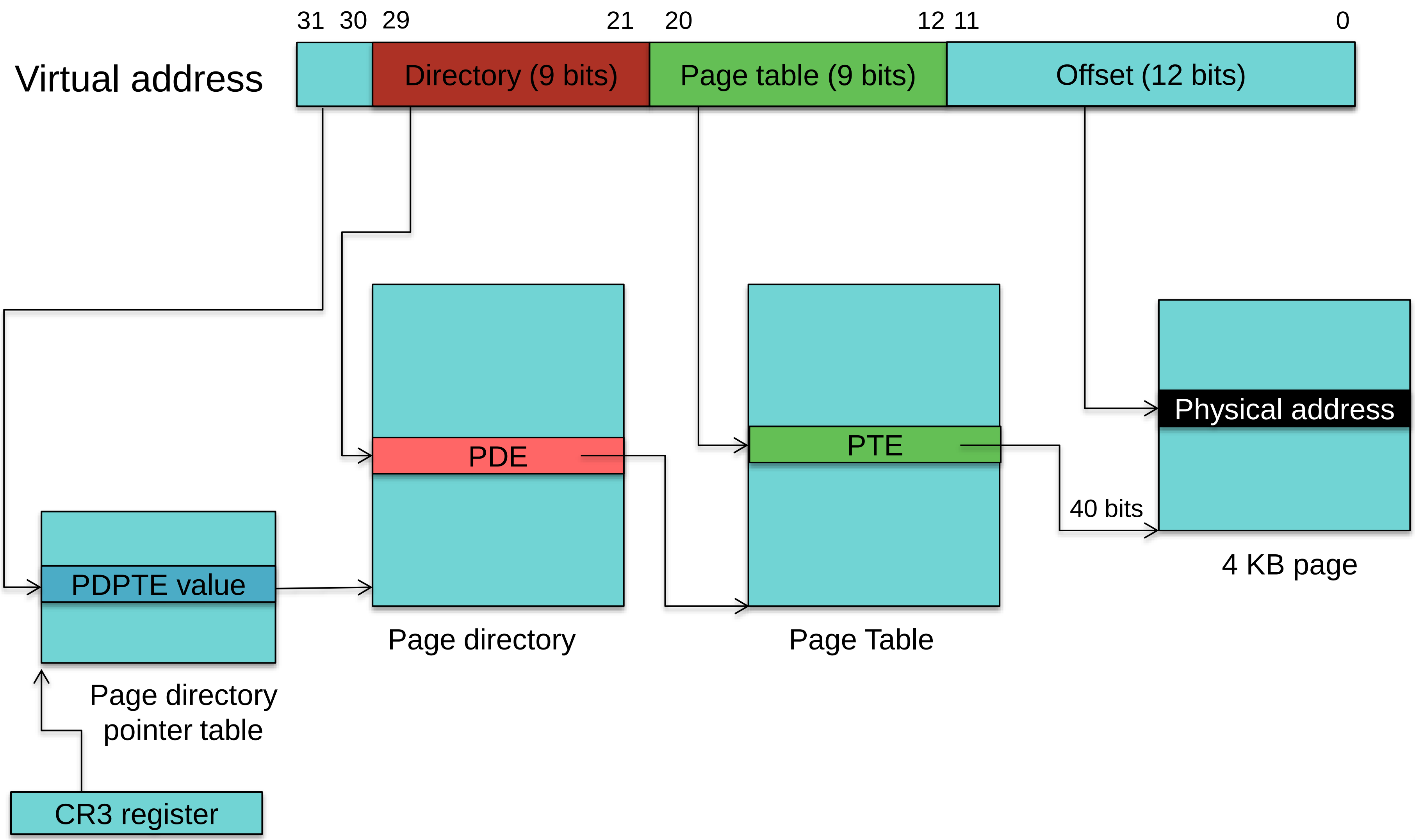
- Segments supported only in IA-32 emulation mode
  - Mostly disabled for 64-bit mode
    - 64-bit base addresses where used
- Three paging modes
  - 32-bit paging
    - 32-bit virtual address; 32-40 bit physical address
    - 4 KB or 4 MB pages
  - PAE
    - 32-bit virtual addresses; up to 52-bit physical address
    - 4 KB or 2 MB pages
  - IA-32e paging
    - 48-bit virtual addresses; up to 52-bit physical address
    - 4 KB, 2 MB, or 1 GB pages

32-bit paging with 4 KB pages

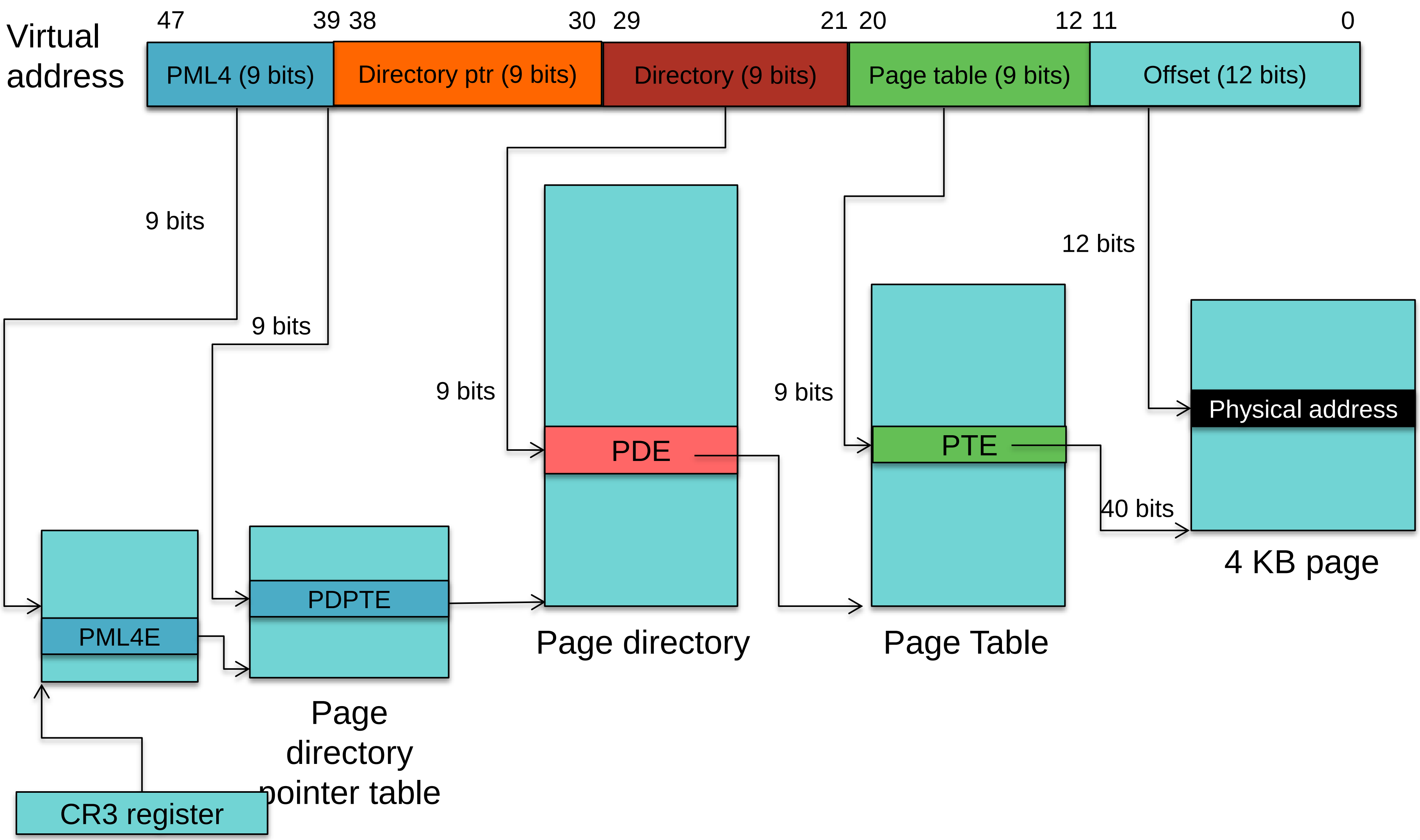


32-bit paging with 4 MB pages

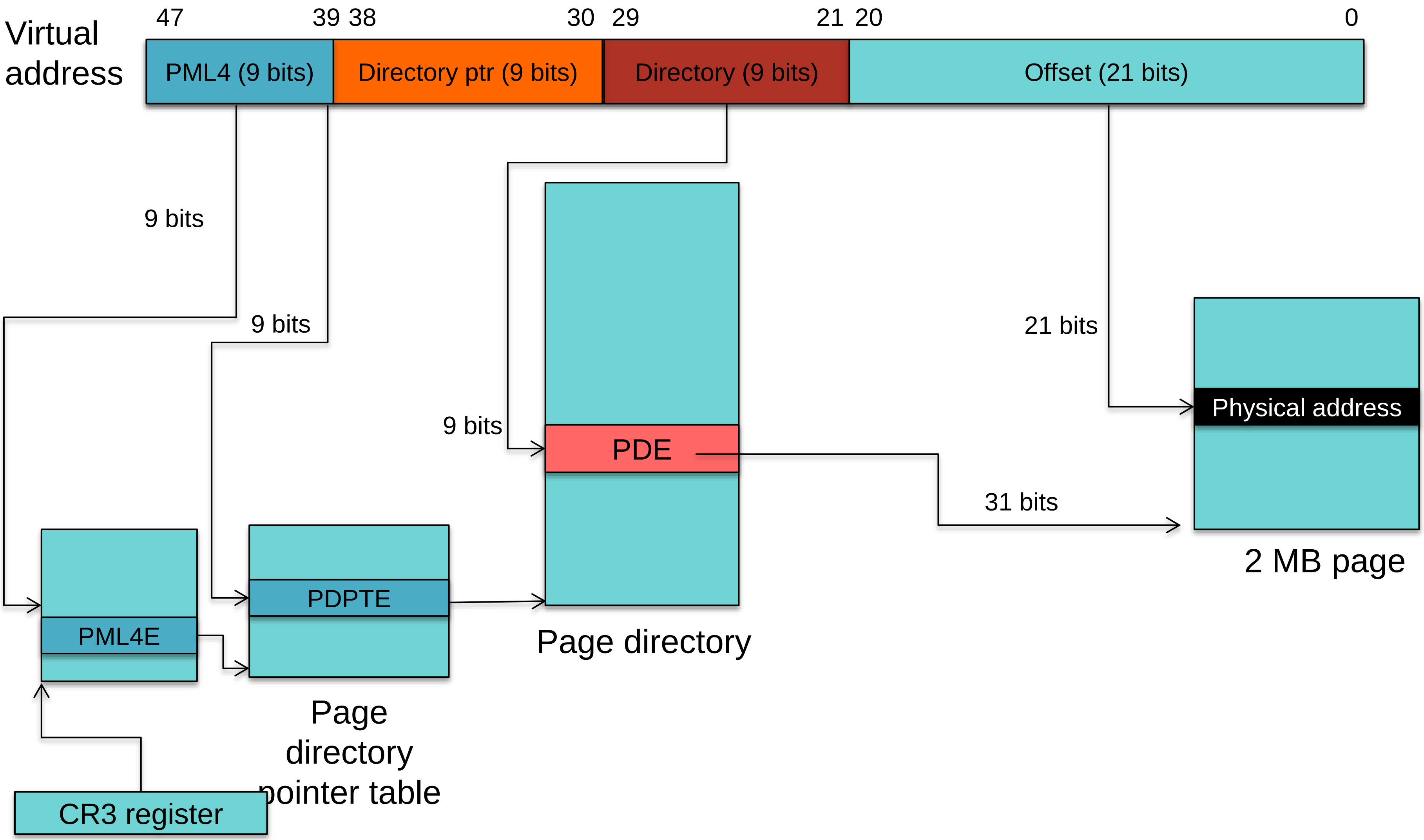
32-bit paging with 4 KB pages & PAE paging



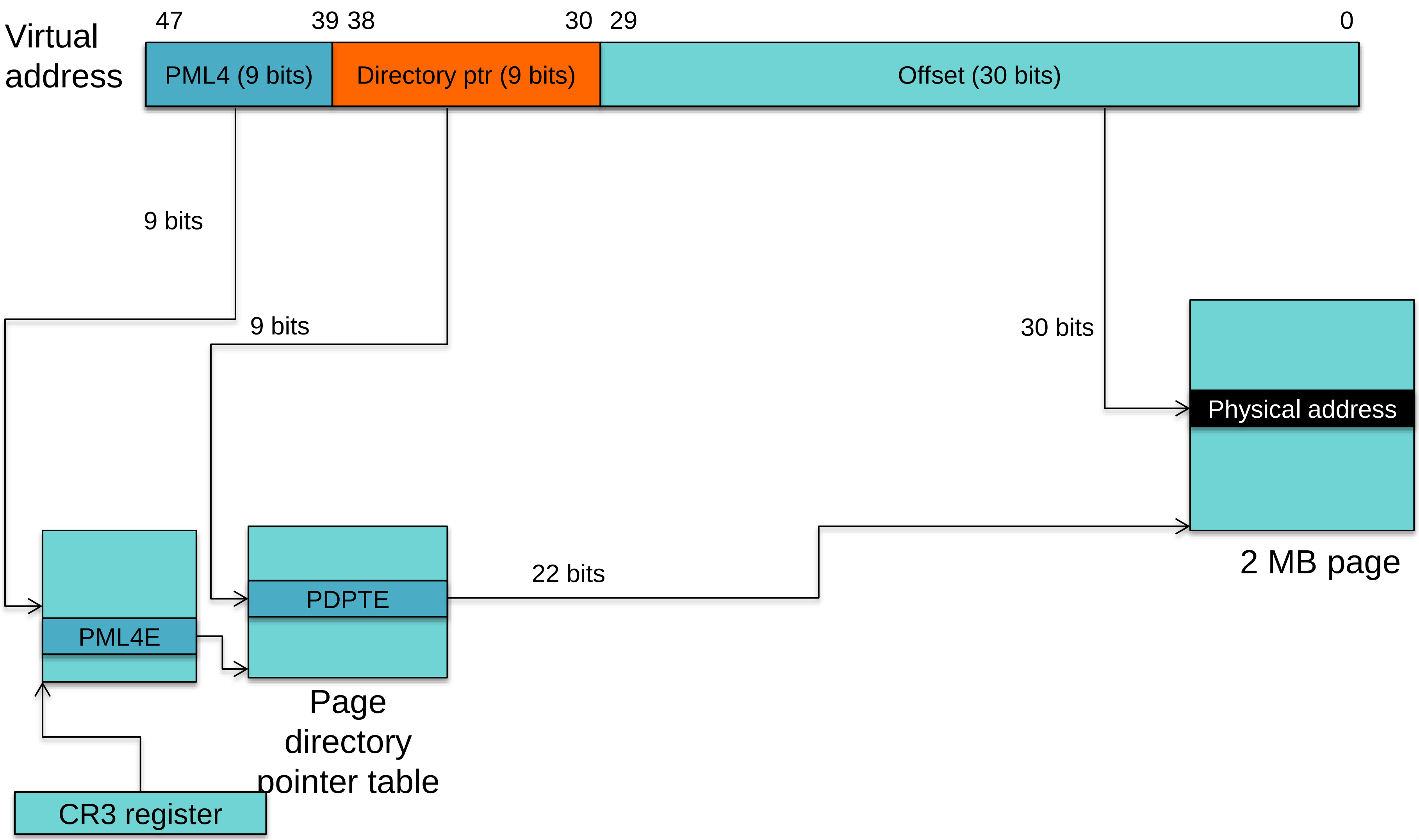
IA-32e paging with 4 KB pages



IA-32e paging with 2 MB pages



IA-32e paging with 1 GB pages



Example: TLBs on the Core i7

- 4 KB pages
  - Instruction TLB: 128 entries per core
  - Data TLB: 64 entries
    - Core 2 Duo: 16 entries TLB0; 256 entries TLB1
    - Atom: 64-entry TLB, 16-entry PDE
- Second-level unified TLB
  - 512 entries

Managing Page Tables

- Linux: architecture independent (mostly)
  - Avoids segmentation (only Intel supports it)
- Abstract structures to model 4-level page tables
  - Actual page tables are stored in a machine-specific manner

Recap

- Fragmentation is a non-issue
- Page table
- Page table entry (PTE)
- Multi-level page tables
- Segmentation
- Segmentation + Paging
- Memory protection
  - Isolation of address spaces
  - Access control defined in PTE

Demand Paging

Executing a program

- Allocate memory + stack
- Load the entire program from memory
  (including any dynamically linked libraries)
- Then execute the loaded program

Executing a program

- Allocate memory + stack
- Load the entire program from memory (including any dynamically linked libraries)
- Then execute the loaded program

This can take a while!

There's a better way…

Demand Paging

- Load pages into memory only as needed
  - On first access
  - Pages that are never used never get loaded

- Use *valid* bit in page table entry
  - Valid: the page is in memory (“valid” mapping)
  - Invalid: out of bounds access or page is not in memory
    - Have to check the process’ memory map in the PCB to find out

- Invalid memory access generates a *page fault*

Demand Paging: At Process Start

- Open executable file
- Set up memory map (stack & text/data/bss)
  – But don't load anything!
- Load first page & allocate initial stack page
- Run it!

Memory Mapping

- Executable files & libraries must be brought into a process' virtual address space
  - File is *mapped* into the process' memory
  - As pages are referenced, page frames are allocated & pages are loaded into them

- `vm_area_struct`
  - Defines regions of virtual memory
  - Used in setting page table entries
  - Start of VM region, end of region, access rights
- Several of these are created for each mapped image
  - Executable code, initialized data, uninitialized data

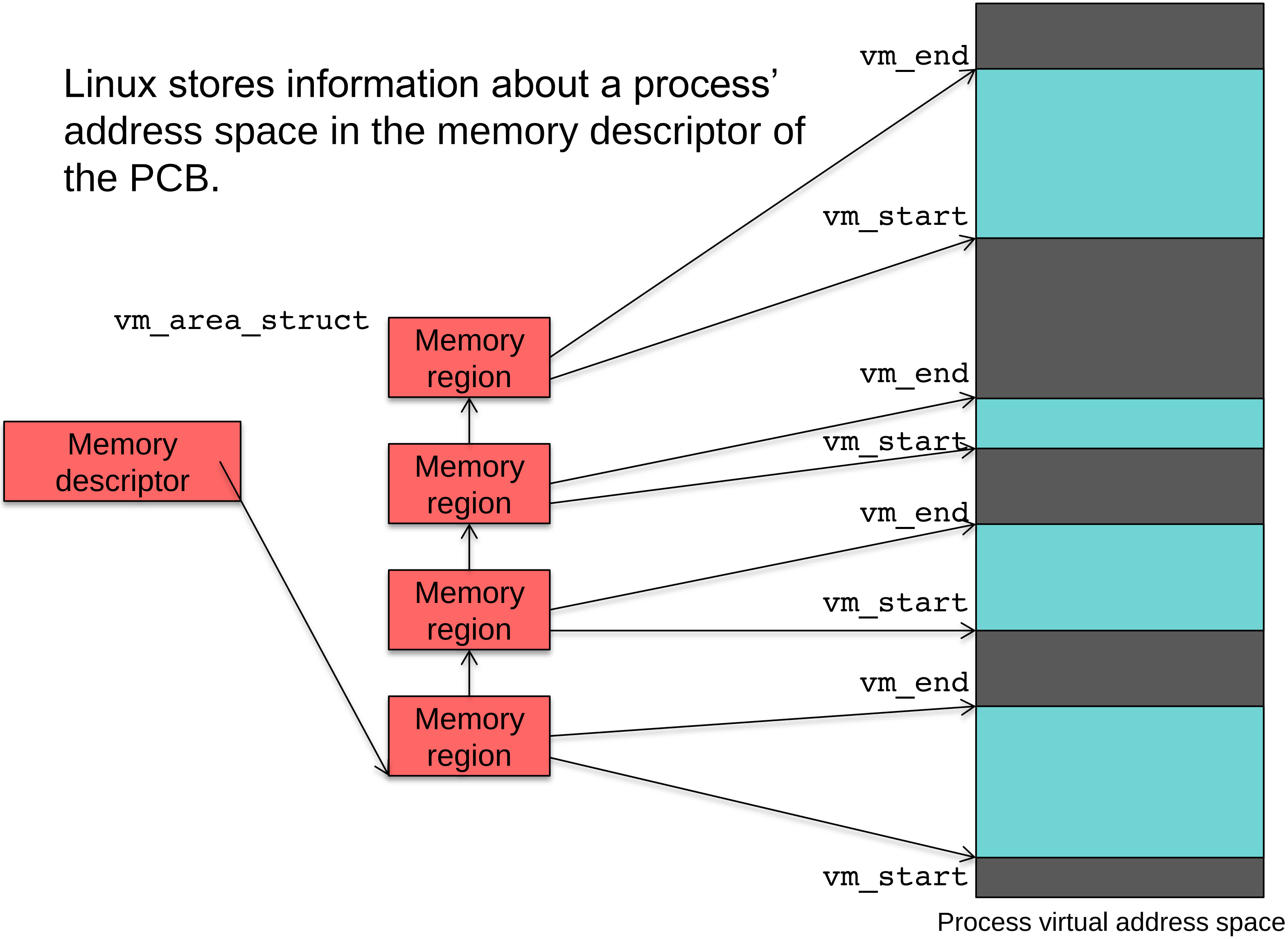
Demand Paging: Page Fault Handling

- Eventually the process will access an address without a valid page
  - OS gets a page fault from the MMU

- What happens?
  - Kernel searches a tree structure of memory allocations for the process to see if the faulting address is valid
    - If not valid, send a SEGV signal to the process
  - Is the type of access valid for the page?
    - Send a signal if not
  - We have a valid page but it’s not in memory
    - Go get it from the file!

Keeping track of a processes' memory region

Linux stores information about a process' address space in the memory descriptor of the PCB.



Page Replacement

- A process can run without having all of its memory allocated
  – It's allocated on demand
- If the
  {address space used by all processes + OS} ≤ physical memory
  then we're ok
- Otherwise:
  – Make room: discard or store a page onto the disk
  – If the page came from a file & was not modified
    - Discard … we can always get it
  – If the page is dirty, it must be saved in a page file (aka swap file)
  – Page file: a file (or disk partition) that holds excess pages
    - Windows: pagefile.sys
    - Linux: swap partition or swap file
    - OS X: multiple swap files in `/private/var/vm/swapfile*`

Demand Paging: Getting a Page

- The page we need is either in the a mapped file (executable or library) or in a page file
  - If PTE is not valid but page # is present
    - The page we want has been saved to a swap file
    - Page # in the PTE tells us the location in the file
  - If the PTE is not valid and no page #
    - Load the page from the program file from the disk
- Read page into physical memory
  1. Find a free page frame (evict one if necessary)
  2. Read the page: This takes time: context switch & block
  3. Update page table for the process
  4. Restart the process at the instruction that faulted

Cost

- Handle page fault exception: ~ 400 usec
- Disk seek & read: ~ 10 msec
- Memory access: ~ 100 ns
- Page fault degrades performance by around 100,000!!
- Avoid page faults!
  - If we want < 10% degradation of performance, we must have just one page fault per 1,000,000 memory accesses

Page replacement

We need a good replacement policy for good performance

FIFO Replacement

First In, First Out

- Good
  - May get rid of initialization code or other code that's no longer used
- Bad
  - May get rid of a page holding frequently used global variables

Least Recently Used (LRU)

- Timestamp a page when it is accessed
- When we need to remove a page, search for the one with the oldest timestamp

- Nice algorithm but…
  - Timestamping is a pain – we can't do it with the MMU!

Not Frequently Used Replacement

Approximate LRU behavior

- Each PTE has a reference bit
- Keep a counter for each page frame
- At each clock interrupt:
  - Add the reference bit of each frame to its counter
  - Clear reference bit
- To evict a page, choose the frame with the lowest counter
- Problem
  - No sense of time: a page that was used a lot a long time ago may still have a high count
  - Updating counters is expensive

Clock (Second Chance)

- Arrange physical pages in a logical circle (circular queue)
  - Clock hand points to first frame
- Paging hardware keeps one *reference* bit per frame
  - Set *reference* bit on memory reference
  - If it's not set then the frame hasn't been used for a while
- On page fault:
  - Advance clock hand
  - Check *reference* bit
    - If 1, it's been used recently – clear & advance
    - If 0, evict this page

Enhanced Clock

- Use the *reference* and *modify* bits of the page
- Choices for replacement – (reference, modify):
  - (0, 0): not referenced recently or modified
    - Good candidate for replacement
  - (0, 1): not referenced recently but modified.
    - The page will have to be saved before replacement
  - (1, 0): recently used.
    - Less ideal – will probably be used again
  - (1, 1): recently used and modified
    - Least ideal – will probably be used again AND we'll have to save it to a swap file if we replace it.
- Algorithm: like clock but replace the first page in the lowest non-empty class

Kernel Swap Daemon

- *kswapd* on Linux
- Anticipate out-of-memory problems
- Decides whether to shrink caches if page count is low
  - Page cache, buffer cache
  - Evict pages from page frames

Demand paging summary

- Allocate page table
  - Map kernel memory
  - Initialize stack
  - Memory-map text & date from executable program (& libraries)
    - But don't load!
- Load pages on demand (first access)
  - When we get a page fault

Summary: If we run out of free page frames

- Free some page frames
  - Discard pages that are mapped to a file

    or
  - Move some pages to a page file

- Clock algorithm

- Anticipate need for free page frames
  - *kswapd* – kernel swap dæmon

Paging: Multitasking Considerations

Supporting multitasking

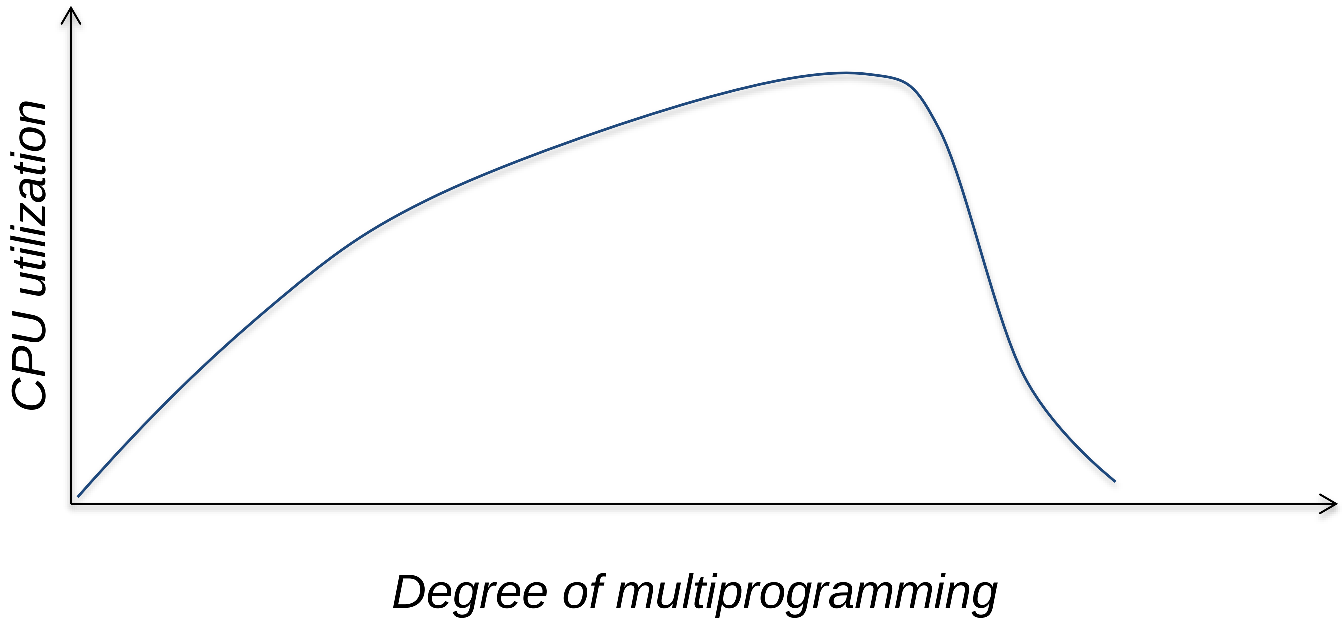
- Multiple address spaces can be loaded in memory
  - Each process sees its own address space
  - Illusion is created by the page table
- A CPU page table register points to the current page table
- OS changes the register set when context switching
  - Includes page table register
- Performance increased with Address Space ID in TLB
  - Can cache *page number* → *page frame number* caching
  - Avoid the need for page table lookups

Working Set

- Keep active pages in memory
- A process needs its working set in memory to perform well
  - Working set =
    Set of pages that have been referenced in the last window of time
  - Spatial locality
  - Size of working set varies during execution
- More processes in a system:
  - *Good*
    Increase throughput; chance that some process is available to run
  - *Bad*
    Thrashing: processes do not have enough page frames available to run without paging

Thrashing

- Locality
  - Process migrates from one working set to another
- Thrashing
  - Occurs when sum of all working sets > total memory
  - There is not enough room to hold each process' working set



Resident Set Management

- Resident set = set of a process' pages in memory
- How many pages of a process do we bring in?
- Resident set can be fixed or variable
- Replacement scope: global or local
  - Global: process can pick a replacement from *all* frames
- Variable allocation with global scope
  - Simple
  - Replacement policy may not take working sets into consideration
- Variable allocation with local scope
  - More complex
  - Modify resident size to approximate working set size

Working Set Model

Approximates locality of a program

- Δ: *working set window*:
  - Amount of elapsed time while the process was actually executing (e.g., count of memory references)

- WSS_i : working set size of process P_i
  - WSS_i = set of pages in most recent Δ page references

- System-wide demand for frames

$$D = \sum WSS_i$$

- If $D >$ *total memory size*, then we get thrashing

Page fault frequency

- Too small a working set causes a process to thrash
- Monitor page fault frequency per process
  - If too high, the process needs more frames
  - If too low, the process may have too many frames

Dealing with thrashing

If all else fails …

- Suspend a process(es)
  - Lowest priority, Last activated, smallest resident set, …?
- Swapping
  - Move an entire process onto the disk: no pages in memory
  - Process must be re-loaded to run
  - Not used on modern systems (Linux, Windows, etc.)
  - Term is now often used interchangeably with *paging*

Real-Time Considerations

- Avoid paging time-critical processes
  - The pages they use will sit in memory
- Watch out for demand paging
  - Might cause latency at a bad time
- Avoid page table lookup overhead
  - Ensure that process memory is mapped in the TLB
    - Pin high-priority real-time process memory into TLB (if possible)
  - Or run CPU without virtual addressing

Memory-mapped files

- Use the virtual memory mechanism to treat file I/O as memory accesses
  - Use memory operations instead of *read* & *write* system calls
- Associate part of the virtual address space with a file
  - Initial access to the file
    - Results in page fault & read from disk
  - Subsequent accesses
    - Memory operations

  mmap system call
- Multiple processes may map the same file to share data

Allocating memory to processes

- When a process needs more memory
  - Pages allocated from kernel
    - Use page replacement algorithms (e.g., clock, enhanced clock, …)
- When do processes need more memory?
  - Demand paging (loading in text & static data from executable file)
  - Memory mapped files via *mmap* (same as demand paging)
  - Stack growth (get a page fault)
  - Process needs more heap space
    - *malloc* is a user-level library: reuses space on the heap
    - *brk* system call: change the data segment "break point"
      malloc requests big chunks to avoid system call overhead
    - More recently, use *mmap* to map "anonymous" memory – memory not associated with a file

The End