Virtual memory
Paging

M1 MOSIG – Operating System Design

Renaud Lachaize

Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (most slides/figures directly adapted from those of the CS140 class)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Textbooks (Silberschatz et al., Tanenbaum)

Flashback
Remember the last lecture

- Virtual memory is required to enforce:
  - Protection/isolation: a process should only mess with its own memory
  - Transparency: memory references and size need to be dynamically adjusted ; give each process its own address space
  - Resource exhaustion management: (efficiently) handle situations where there is not enough memory to fit all processes
- The MMU is here to help
  - Hardware support for address translation
- Segmentation
  - A first approach suffering from a significant drawback: fragmentation, caused by:
    - Size heterogeneity
    - Isolated deaths (lifecycle heterogeneity)
    - Time-varying behavior

Outline

- Basic principles for address translation
- Data structures
- Case study: x86 page translation (architected page tables)
- Making paging fast (TLB)
- Another example: MIPS (architected TLB)

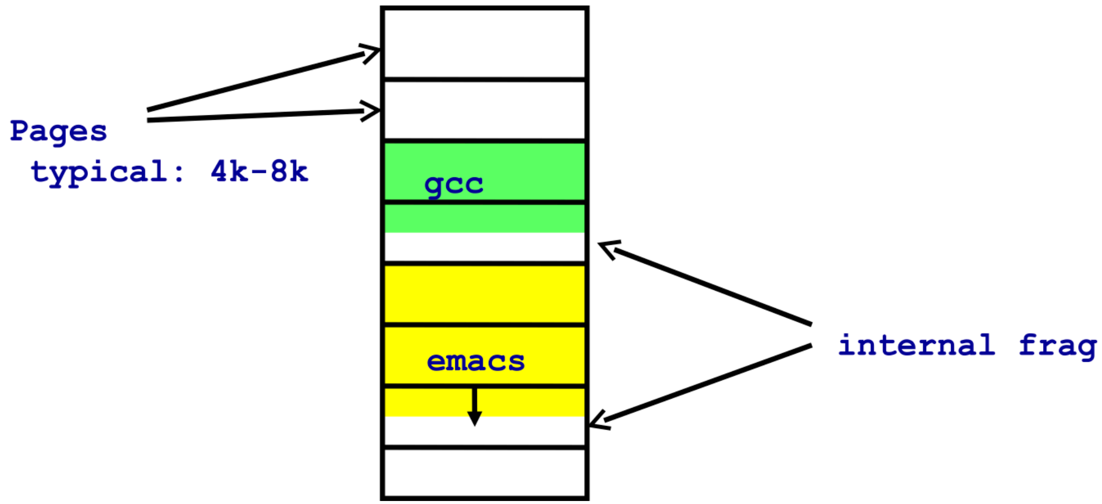
Paging

- Divide memory into fixed-size pages
- Map virtual pages to physical pages (a.k.a. “frames”)
  - Each process has separate mapping configuration

- Allow OS to gain control on certain operations
  - Write attempt to a read-only page triggers trap to OS kernel
  - (Read or write) attempt to invalid page triggers trap to OS kernel
  - OS can change mapping and resume application

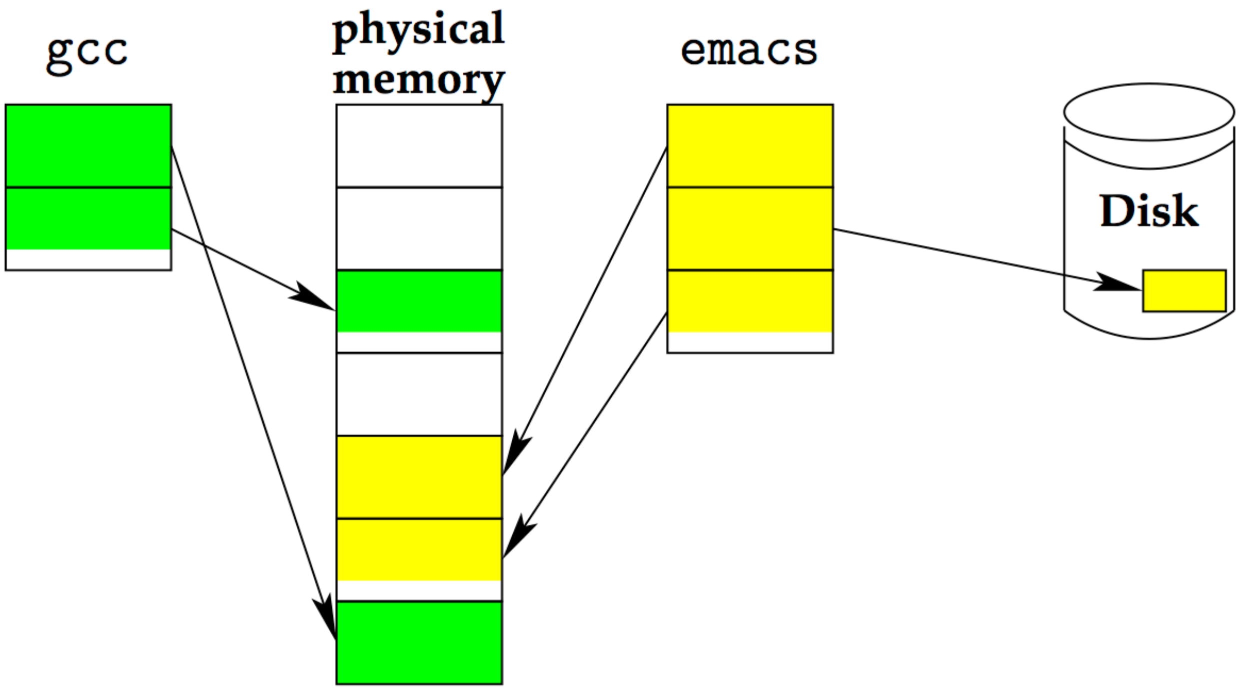
- Other features sometimes found
  - Hardware can set “accessed” and “dirty” bits
  - Control page execute permission separately from read/write
  - Control caching of page

Paging trade-offs



- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of 0.5 pages per segment

Simplified allocation



- Allocate any physical page to any process
- Can store idle virtual pages on disk

Paging data structures

- Pages are fixed size, e.g., 4 kB
  - Least significant bits (e.g., 12 bits for 4kB-pages) of address form the *page offset*
  - The most significant bits form the *page number*

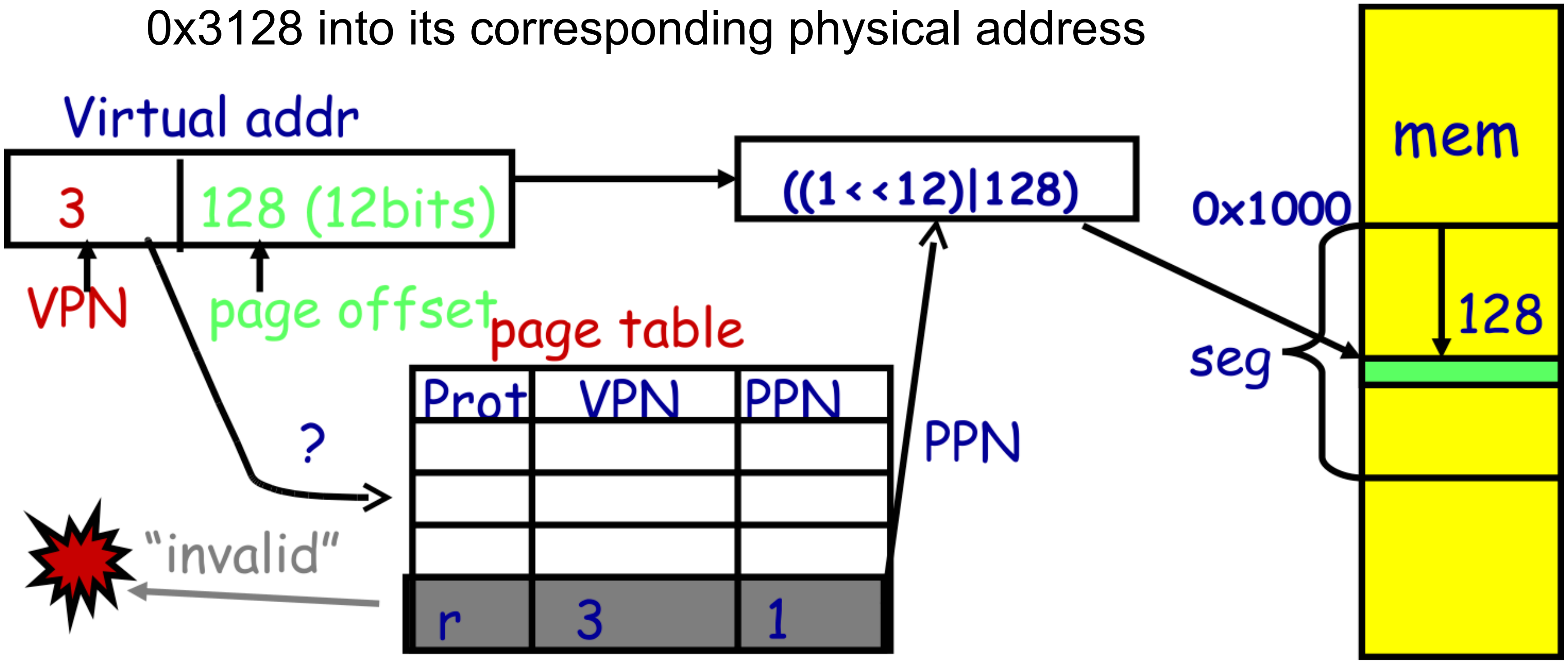
- Each process has a page table
  - Maps *virtual page number (VPN)* to *physical page number (PPN)*
  - Also includes bits for protection, validity, etc.

- On memory access:
  - Translate VPN to PPN, then add offset

Paging data structures (continued)

In this example:

- We assume a (fixed) page size of 4 kB (4096 bytes)
- We consider an attempt to convert virtual address 0x3128 into its corresponding physical address



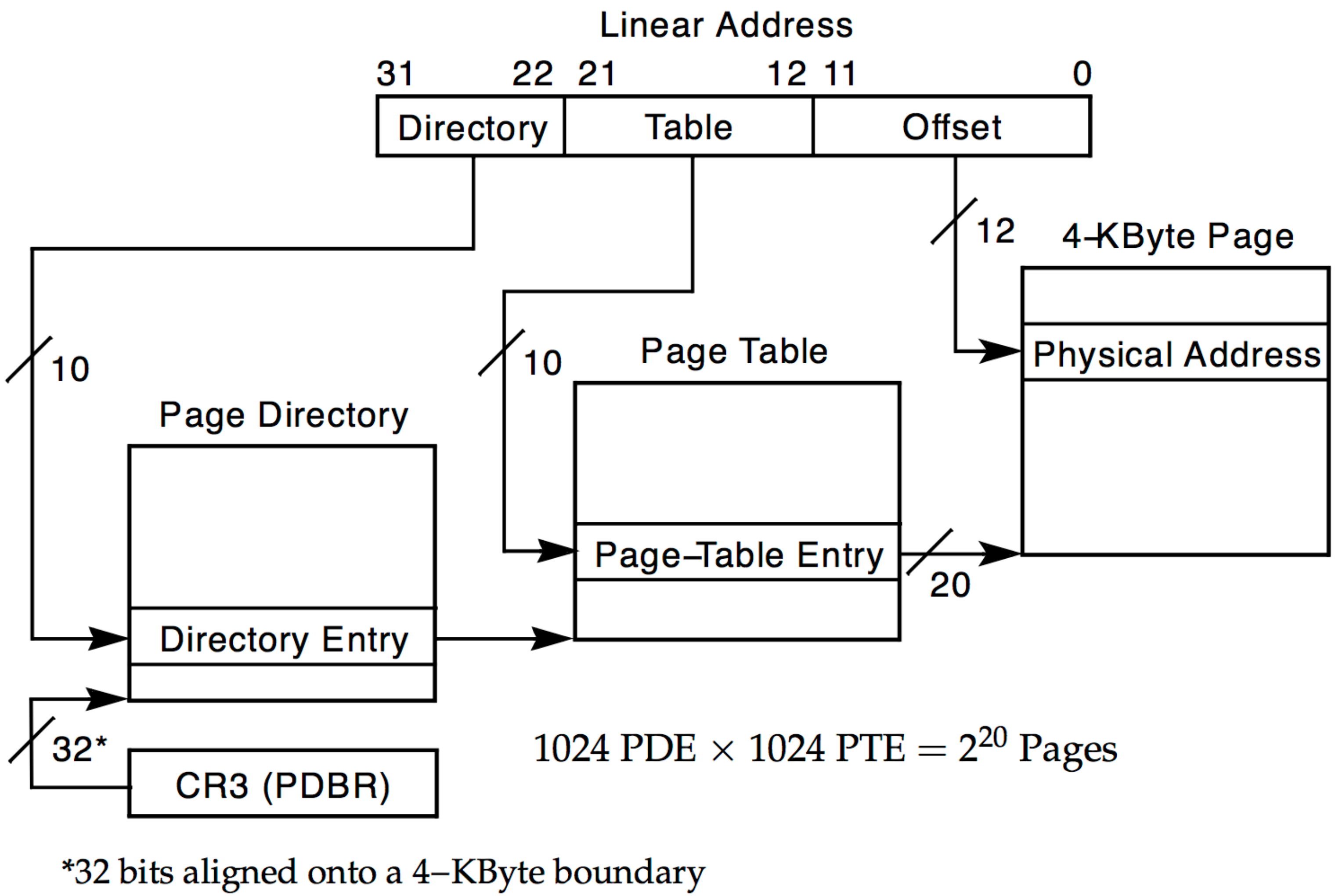
What happens in the case of a read access? And for a write access?

A detailed example of hardware support for paging: Paging on (Intel 32-bit) x86 processors

- Paging enabled by bits in control register (%cr0)
  - Only privileged OS code can manipulate control registers
- Normally 4kB pages
  - x86 uses 32-bit words => 4 GB of addressable memory
  - offset: 12 bits / page index: 20 bits => flat page table = 4 MB per process (big!)
  - Instead of a flat table, a hierarchical structure is used

- %cr3 register points to 4kB page directory (1 directory per process)
- Page directory: 1024 PDEs (page directory entries)
  - Each PDE contains the physical address of a page table
  - Table index: 10 bits
- Page table: 1024 PTEs (page table entries)
  - Each PTE contains the physical address of virtual 4k page
  - A page table covers (up to) 4 MB of virtual memory
  - Page index: 10 bits

x86 page translation



Space savings with two-level paging structures Example on x86 (1/2)

- Assume an address space with the following valid address ranges:
  - Range 1: 0x00000000 to 0x00000FFF
  - Range 2: 0x01000000 to 0x020003FFF

- How much space is required by the paging structures in the following setups?
  - Setup 1: using a flat page table
  - Setup 2: using a two-level structure (i.e., with PDE+PTEs)

Space savings with two-level paging structures Example on x86 (2/2)

- Assume an address space with the following valid address ranges
  - Range 1: 0x00000000 to 0x00000FFF
  - Range 2: 0x01000000 to 0x02003FFF

- How much space required by paging structures?
  - With flat page table: 4 MB (2^{20} entries of 32 bits each)

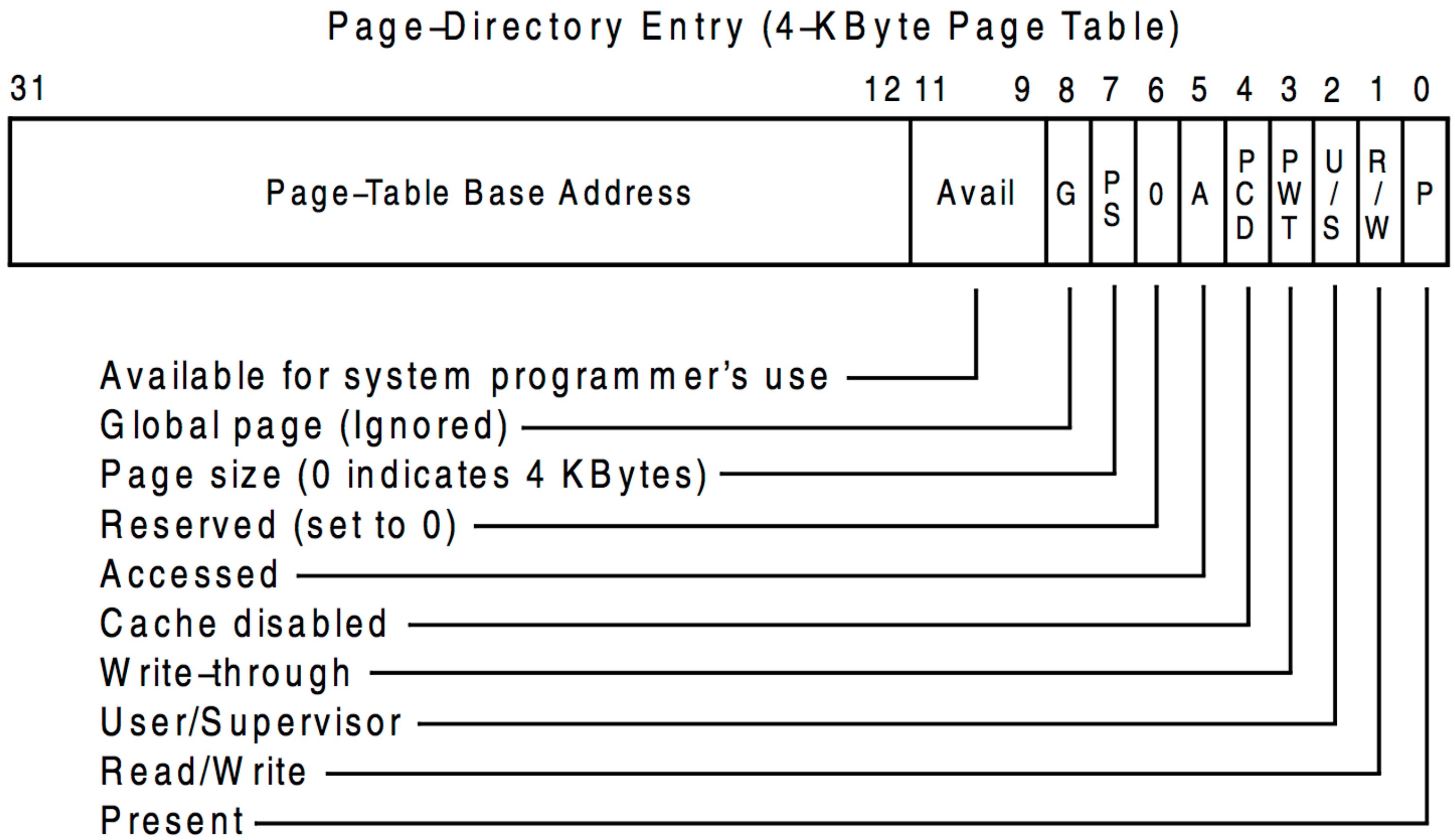
  - With PDE+PTEs: 28 kB
    - Page directory: 4 kB
    - Page tables for range 1: 1 page table (4 kB)
      - Page table 0x0
      - Because the 10 most significant bits are the same for all addresses in range 1
    - Page tables for range 2: 5 page tables (4 kB each)
      - Page tables 0x4 to 0x8
      - PT 0x4 for range 0x01000000 to 0x013FFFFF, PT 0x5 for range 0x01400000 to 0x017FFFFF, ..., PT 0x8 for range 0x02000000 to 0x02003FFF
      - Note that some of the entries of PT 0x8 are not used (only 4 are used)

Documentation about x86 paging

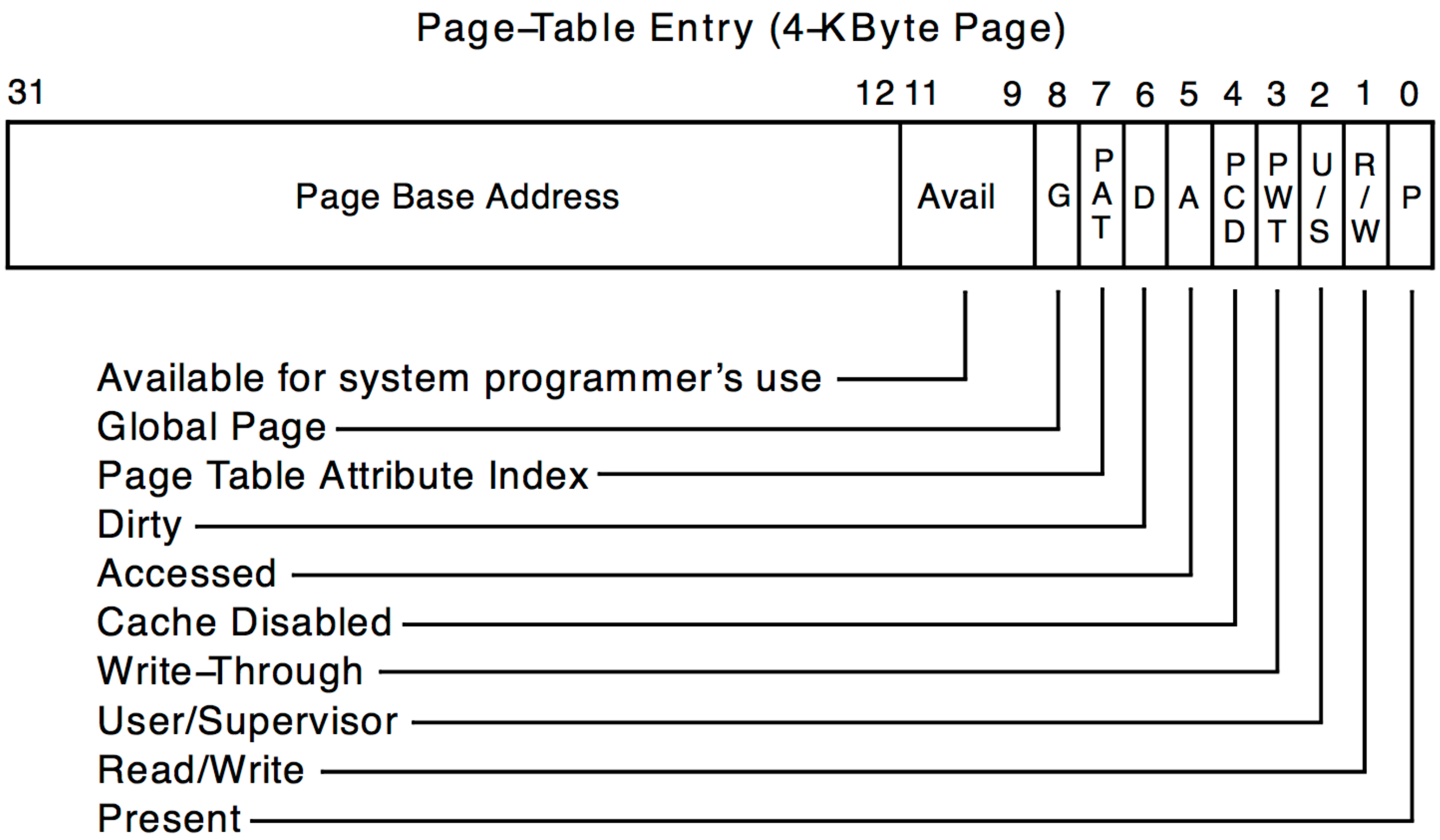
- Old Intel manual (simplest explanation): Intel 80386 Programmer's reference manual, 1987
  - http://www.scs.stanford.edu/05au-cs240c/lab/i386/toc.htm
  - http://www.scs.stanford.edu/05au-cs240c/lab/386htm09.tar.gz

- See also:
  - Volume 2 of AMD64 Documentation:
    - http://www.scs.stanford.edu/05au-cs240c/lab/amd64/AMD64-2.pdf
  - Volume 3A of Intel Pentium Manual:
    - http://www.scs.stanford.edu/05au-cs240c/lab/ia32/IA32-3.pdf
  - Volume 3A of Intel (IA-32 and Intel 64) manual
    - http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

x86 page directory entry



x86 page table entry



x86 hardware segmentation

- x86 also supports segmentation
  - Segment register base + pointer value = linear address
  - Page translation happens on linear addresses
- Two levels of protection and translation check
  - Segmentation model has four privilege levels (CPL 0-3)
  - Paging only two, so 0-2 = kernel, 3 = user

x86 hardware segmentation (continued)

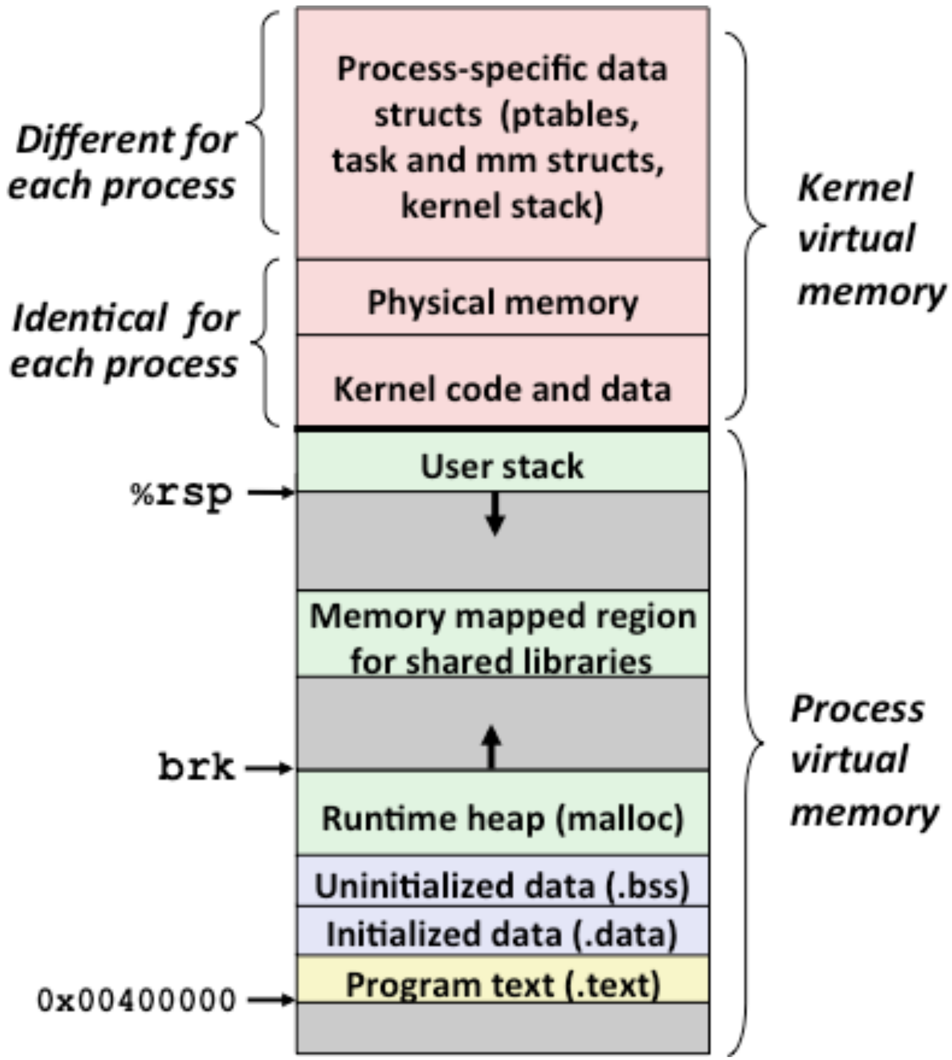
- Why do you want both paging and segmentation?
- Short answer: you don't
  - Most operating systems use "flat mode" – set base = 0, bounds = 0xFFFFFFFF in all segments registers, then forget about it
  - The x86-64 architecture removes most segmentation support
- Long answer: may be useful
  - Used in some operating systems for refining protection checks

Where does the OS kernel live?

- In its own address space?
- Cannot do this on most hardware
  - e.g., syscall instruction will not switch address spaces
  - Also would make it harder to parse syscall arguments passed as pointers
- So in the same address space as process
  - Use protection bits to prohibit user code from accessing kernel code/data
- Typically all kernel code and most kernel data are mapped at same virtual address in every address space
  - On x86, must manually set up page tables for this
  - Usually, just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous memory
  - Some hardware/systems put physical memory (kernel-only) somewhere in virtual address space

Where does the OS kernel live?

Example: Virtual address space layout of a Linux process



Making paging fast

- Motivating example: x86 (2-level) paging structures require 3 memory references per load/store instruction
  - (1) Look up page table address in page directory
  - (2) Look up PPN in page table
  - (3) Actually access physical page corresponding to virtual address

- For speed, the CPU caches the recently used translations
  - Special cache called a *translation lookaside buffer* or TLB
  - Typical: 64-2k entries, 4-way to fully associative, 95% hit rate
  - Each TLB entry maps a VPN to PPN + protection information

- On each memory reference
  - CPU checks TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time (must evict some entry - we will discuss eviction soon)

TLB principle

- A TLB is a fast (small) associative memory which can perform a parallel search
- It acts as a cache for the page table
- TLB management can either be one at the hardware or software level (depending on the design choice for a given processor architecture)

TLB effective access time

- The percentage of successful lookups in the TLB is called the TLB hit ratio

- Typical TLB stats:
  - Size: from a few to a few hundreds (max. thousands) of entries
  - Hit time: 0-1 clock cycle
  - Miss penalty: 10-100 clock cycles
  - Miss rate: depends on the workload

- Example:
  - If a TLB hit takes 1 clock cycle, a miss takes 30 cycles and the miss rate is 1%, the effective memory cycle rate is an average of 1x0.99 + (1+30)x0.01 = 1.3 clock cycles per instruction
  - A 10% miss rate would lead to 4 cycles

TLB details

- TLB operates at CPU pipeline speed (fast)

- Complication: what to do when switching address space?

  - Flush TLB on context switch (e.g., on x86 processors until recently)

  - Another design: Tag each entry with associated process ID: ASIDs (address space IDs)
    - E.g., MIPS, and recent extensions to the x86 architecture

TLB details (continued)

- In general, the OS must manually keep the TLB valid
  - E.g., x86 **`invlpg`** instruction
    - Invalidates a page translation in TLB
    - Must execute after changing a possibly used page table entry
    - Otherwise, hardware will miss page table change
  - On a multiprocessor, more complex
    - Every core has its own TLB
    - Maintaining consistency is non-trivial (TLB shootdown coordination software protocol)

An example of a very different MMU: MIPS

- Hardware has 64-entry TLB
  - References to addresses not found in TLB triggers trap to kernel
  - In other words: TLB misses are handled by the (OS) software rather than by the hardware

- Specific processor instructions for the manipulation of the TLB entries
  - Because the contents of the TLB must be software managed

- A different trade-of compared to x86
  - Pros: Simpler hardware, flexibility for OS design (the OS is free to choose the page table format)
  - Cons: More things to be performed in software: performance, additional complexity (must avoid infinite chain of TLB misses)

References

- Remzi & Andrea Arpaci-Dusseau. OSTEP textbook (http://www.ostep.org).
  - (Chapter 15: Address translation)
  - Chapter 18: Introduction to paging
  - Chapter 19: Paging – Faster translations (TLBs)
  - Chapter 20: Paging – Smaller tables

- Bruce Jacob and Trevor Mudge. Virtual memory: issues of implementation. IEEE Computer, June 1998.

- AMD and Intel documentations (see previously mentioned links)