# MMU Types

- **Memory Management Units (MMU) come in two flavors**
- **Hardware Managed**
  - Hardware reloads TLB with pages from a page tables
  - Typically hardware page tables are Radix Trees
  - Requires complex hardware
  - Examples: x86, ARM64, IBM POWER9+
- **Software Managed**
  - Simplier hardware and asks software to reload pages
  - Requires fast exception handling and optimized software
  - Enables more flexiblity in the TLB (e.g. variable page sizes)
  - Examples: MIPS, Sun SPARC, DEC Alpha, ARM and POWER

# Today's Lecture

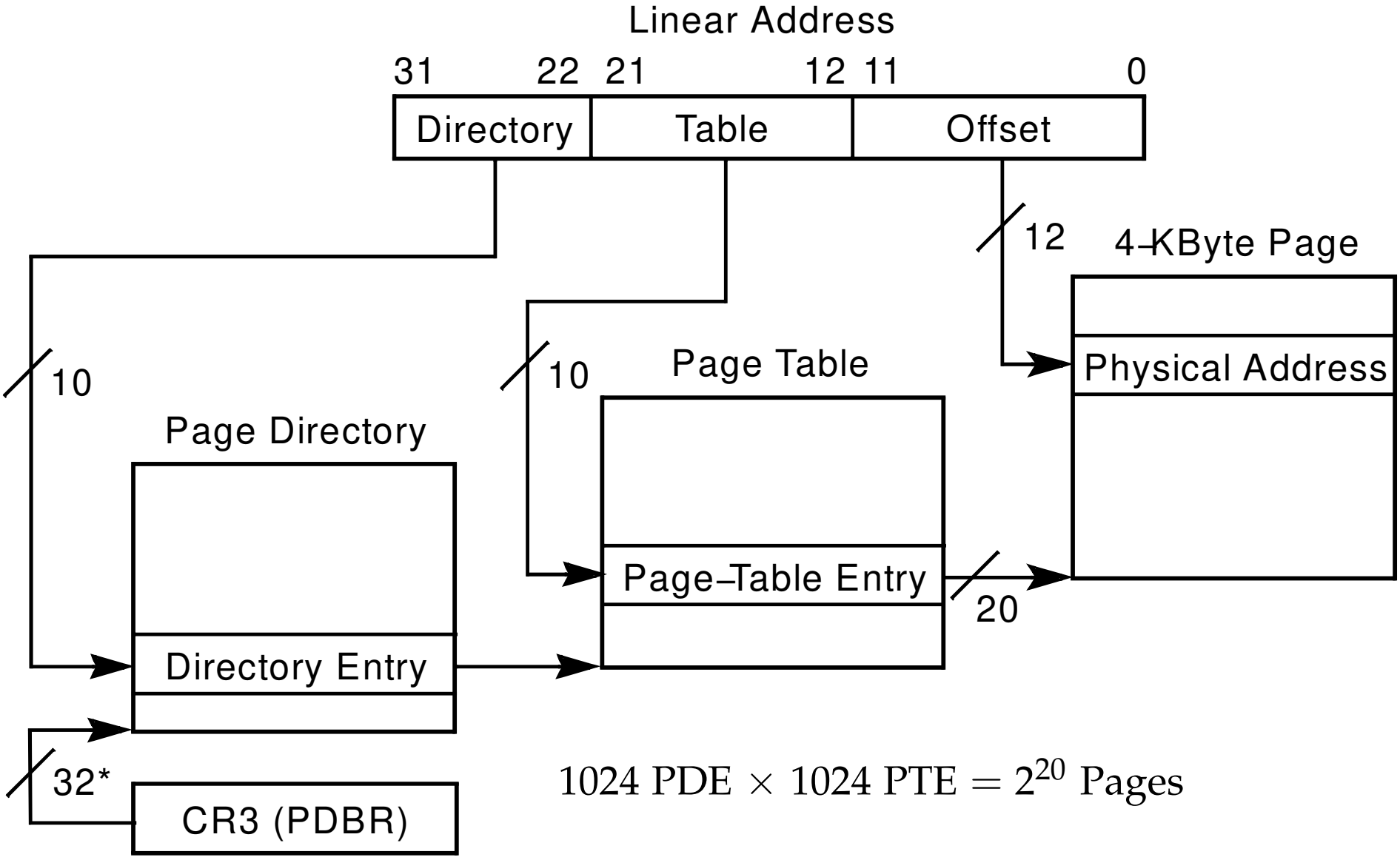
- **x86 Hardware Managed MMU**
- **MIPS Software Managed MMU**
  - In your assignment you will implement a Radix tree like x86 for MIPS!

# Outline

# x86 Paging

- **Paging enabled by bits in a control register (`%cr0`)**
  - Only privileged OS code can manipulate control registers
- **Normally 4KB pages**
- **`%cr3`: points to 4KB page directory**
- **Page directory: 1024 PDEs (page directory entries)**
  - Each contains physical address of a page table
- **Page table: 1024 PTEs (page table entries)**
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MB of Virtual mem
- **See intel manual for detailed explanation**
  - Volume 2 of AMD64 Architecture docs
  - Volume 3A of Intel Pentium Manual

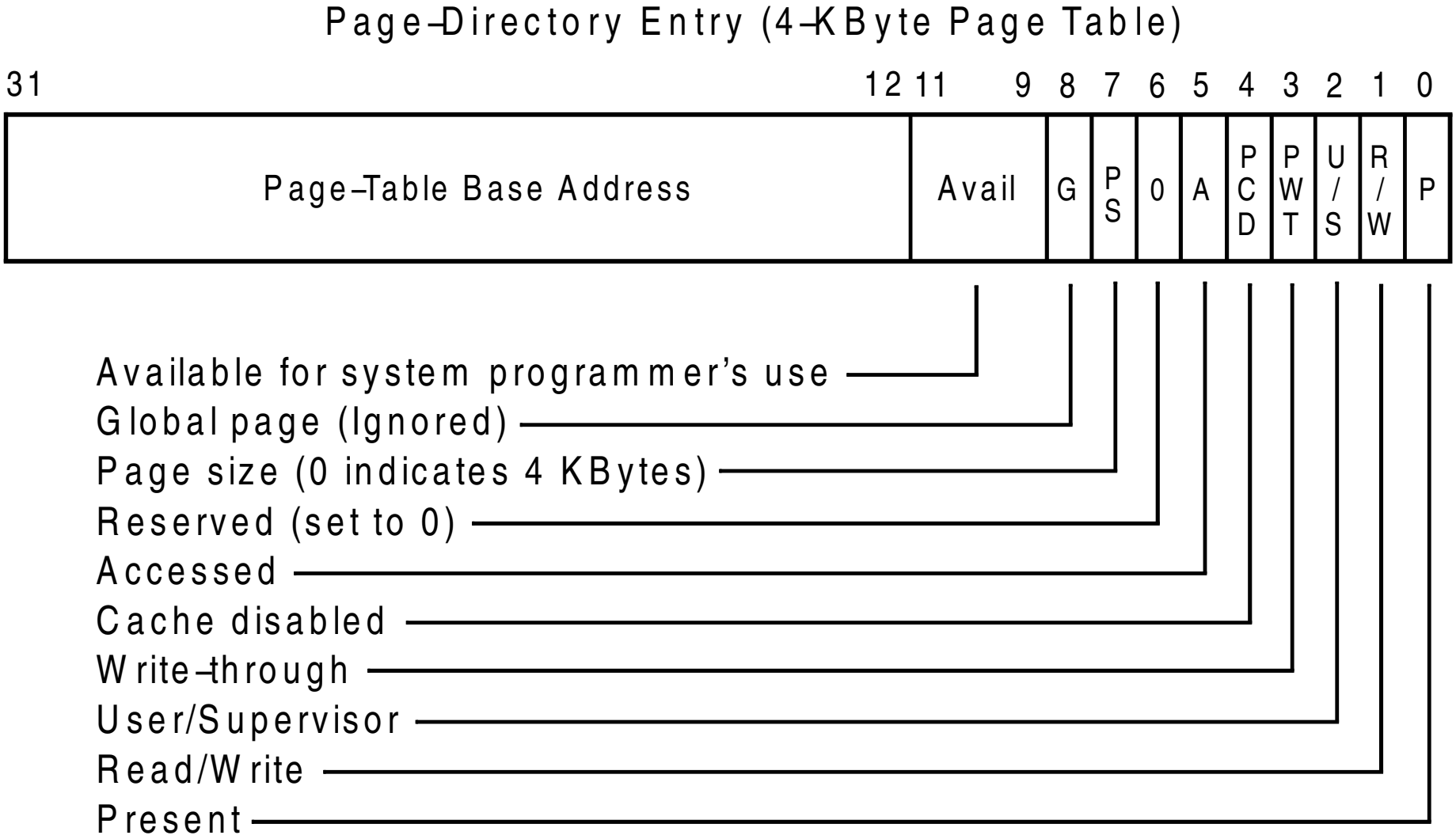
# x86 page translation



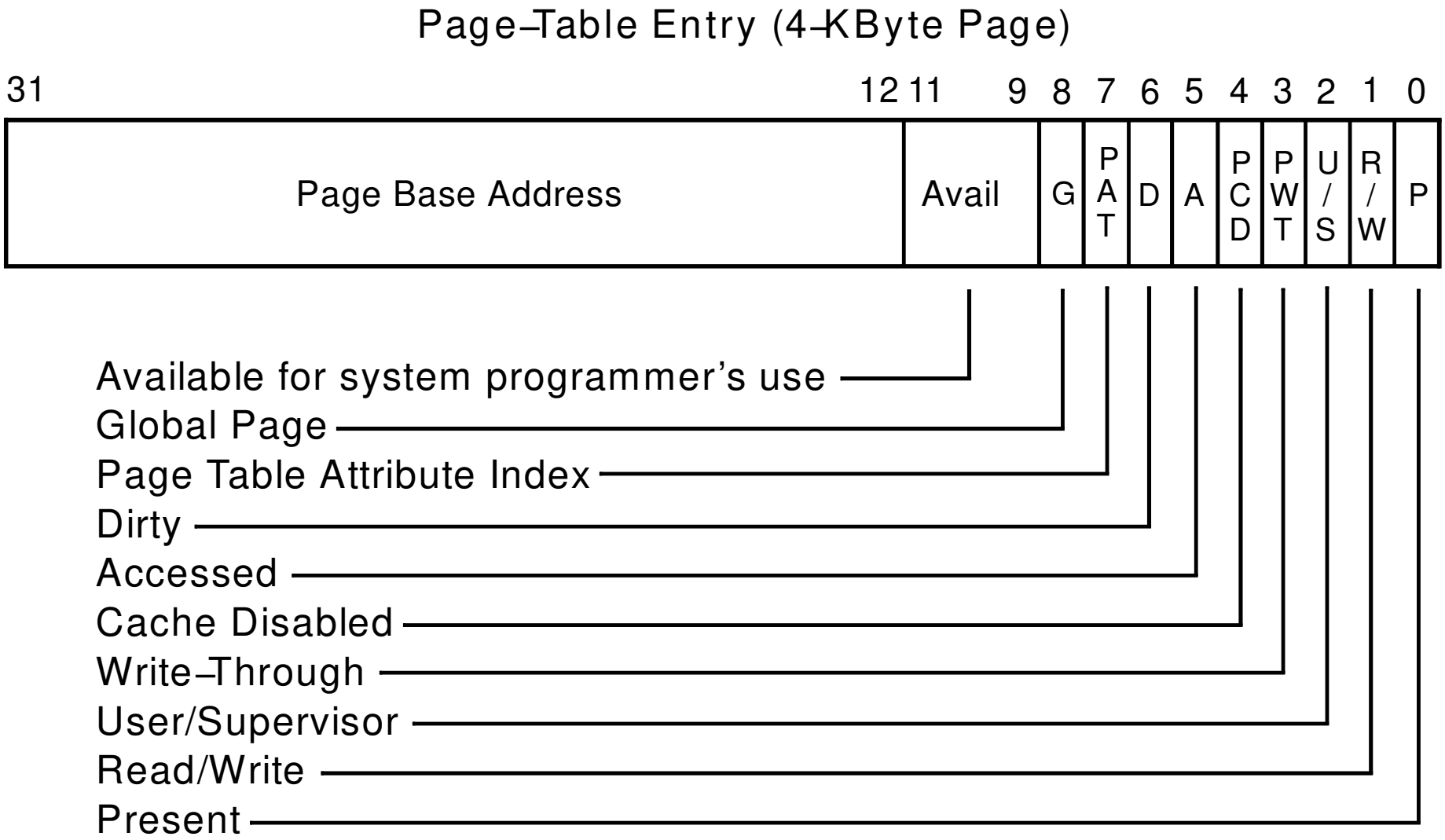

$1024 \text{ PDE} \times 1024 \text{ PTE} = 2^{20} \text{ Pages}$

*32 bits aligned onto a 4–KByte boundary

# x86 page directory entry



# x86 page table entry



# x86 hardware segmentation

- **x86 architecture *also* supports segmentation**
  - Segment register base + pointer val = *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
- **Why do you want *both* paging and segmentation?**

# x86 hardware segmentation

- **x86 architecture *also* supports segmentation**
  - Segment register base + pointer val = *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
- **Why do you want *both* paging and segmentation?**
- **Short answer: You don't – just adds overhead**
  - Most OSes use "flat mode" – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support
- **Long answer: Has some fringe/incidental uses**
  - VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for W$\wedge$X when no PTE NX bit

# Making paging fast

- **x86 PTs require 3 memory references per load/store**
  - Look up page table address in page directory
  - Look up PPN in page table
  - Actually access physical page corresponding to virtual address
- **For speed, CPU caches recently used translations**
  - 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 $\rightarrow$ PPN + protection information
- **On each memory reference**
  - Check TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time
    (Must evict some entry)

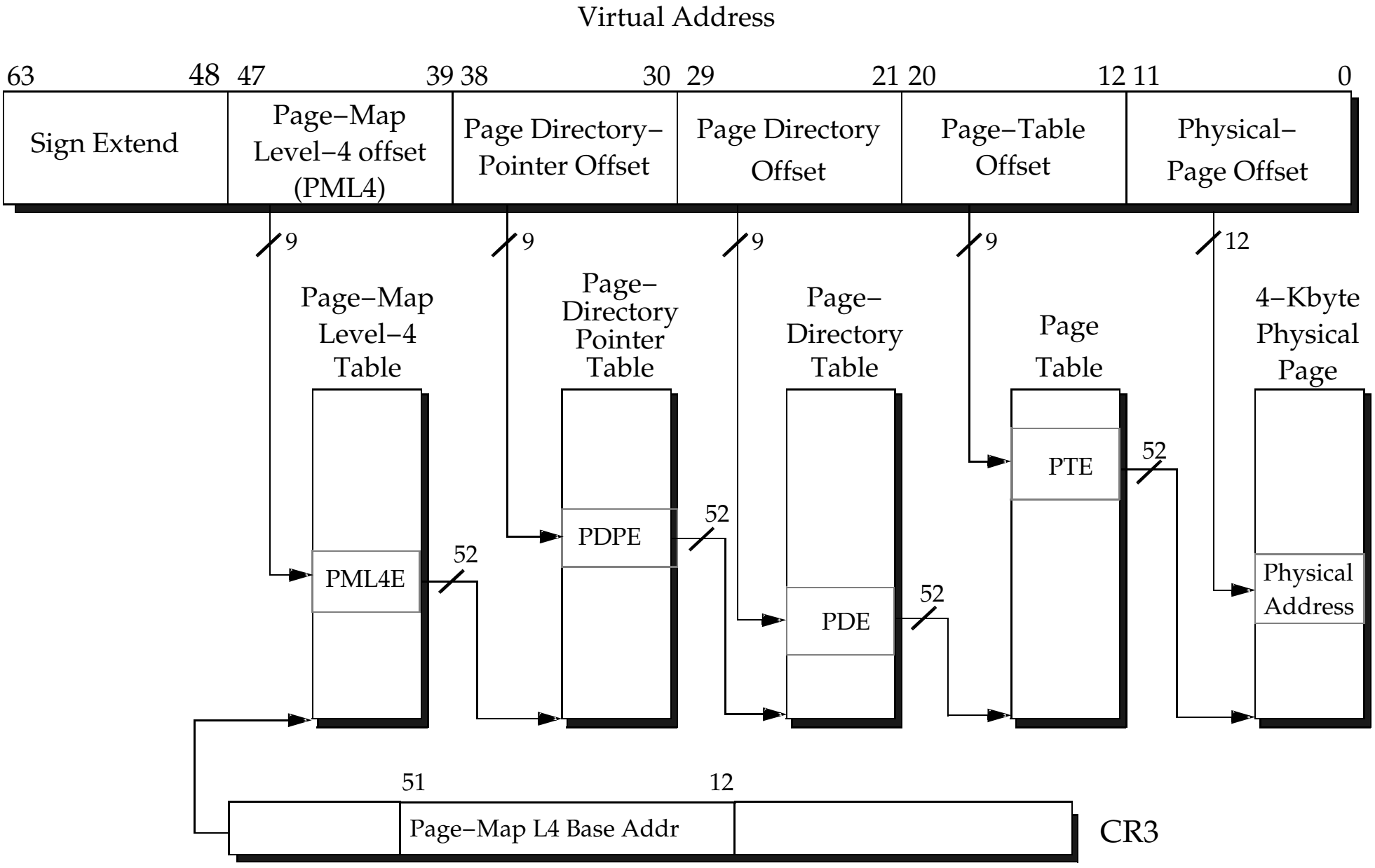
# TLB details

- **TLB operates at CPU pipeline speed $\Longrightarrow$ small, fast**
- **Complication: what to do when switch address space?**
  - Flush TLB on context switch (e.g., old x86)
  - Tag each entry with associated process's ID (e.g., MIPS)
- **In general, OS must manually keep 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
- **More Complex on a multiprocessor (TLB shootdown)**

# x86 Paging Extensions

- **PSE: Page size extensions**
  - Setting bit 7 in PDE makes a 4MB translation (no PT)
- **PAE Page address extensions**
  - Newer 64-bit PTE format allows 36 bits of physical address
  - Page tables, directories have only 512 entries
  - Use 4-entry Page-Directory-Pointer Table to regain 2 lost bits
  - PDE bit 7 allows 2MB translation
- **Long mode PAE**
  - In Long mode, pointers are 64-bits
  - Extends PAE to map 48 bits of virtual address (next slide)
  - Why are aren't all 64 bits of VA usable?

# x86 long mode paging



# Where does the OS live?

- **In its own address space?**
  - Can't do this on most hardware (e.g., syscall instruction won't 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 writing kernel
- **Typically all kernel text, most data at same VA 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 physical memory
  - Some hardware puts physical memory (kernel-only) somewhere in virtual address space

# Outline

1. Intel x86: Hardware MMU

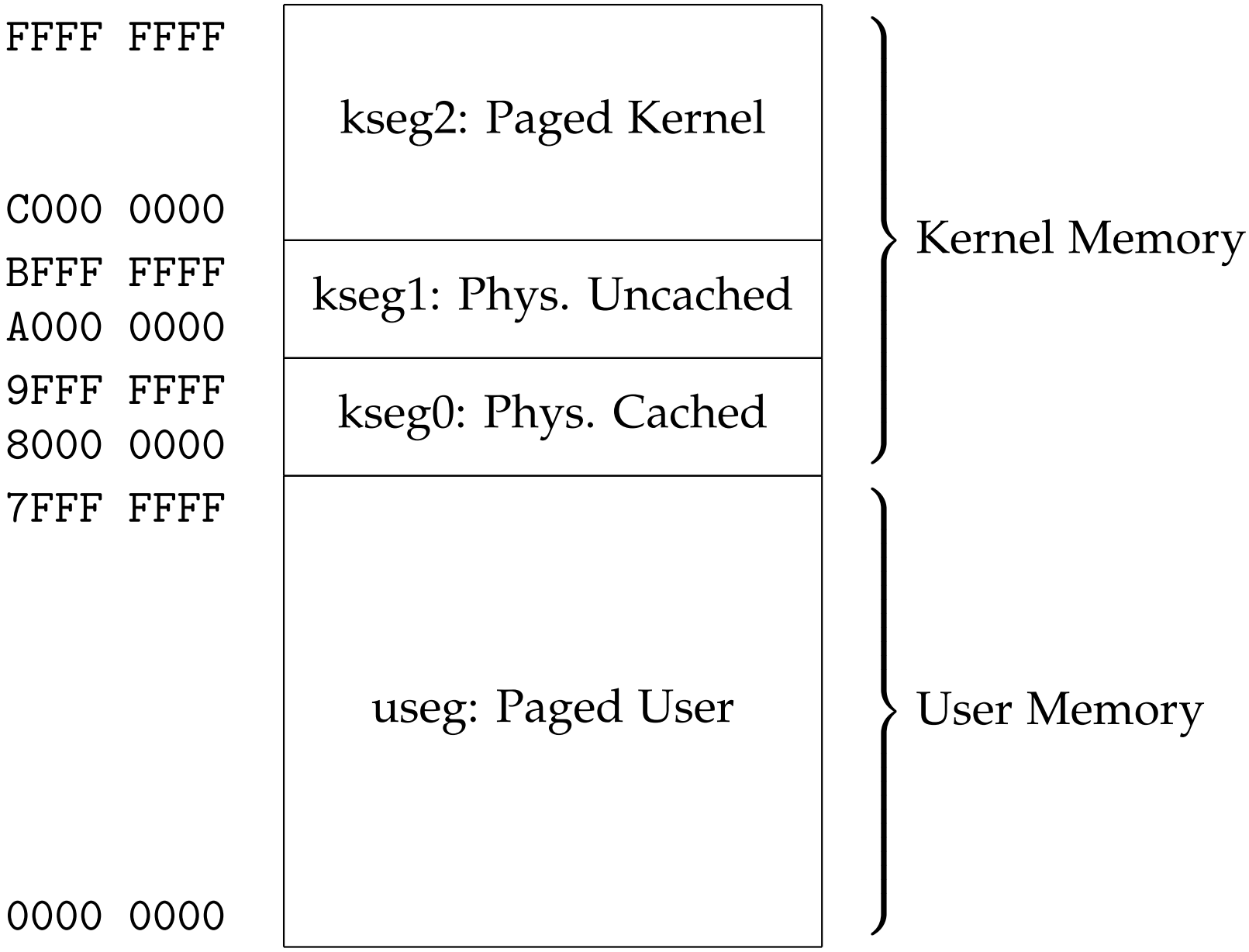
2. MIPS: Software Managed MMU

# Very different MMU: MIPS

- **Hardware has 64-entry TLB**
  - References to addresses not in TLB trap to kernel
- **Each TLB entry has the following fields:**

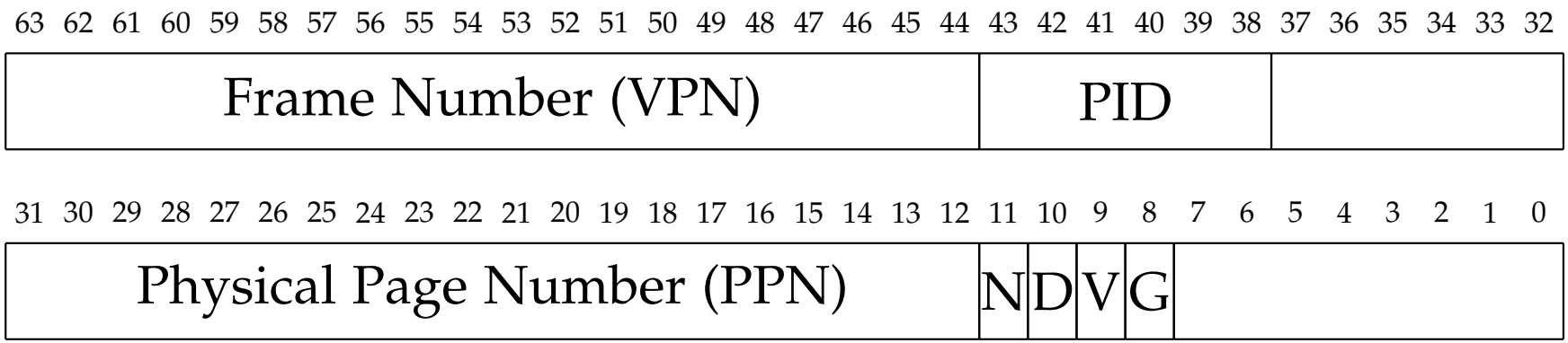
  Virtual page, Pid, Page frame, NC, D, V, Global
- **Kernel itself unpaged**
  - All of physical memory contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses
- **User TLB fault hander very efficient**
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables
- **OS is free to choose page table format!**

# MIPS Memory Layout



# MIPS Translation Lookaside Buffer

- **TLB Entries: 64 - 64-bit entries containing:**
  - PID: Process ID (tagged TLB)
  - N: No Cache - disables caching for memory mapped I/O
  - D: Writeable - makes the page writeable
  - V: Valid
  - G: Global - ignores the PID during lookups




- **Page Sizes: Multiples of 4 from 4 kiB–16 MiB**
  - 4 kiB, 16 kiB, 64 kiB, 256 kiB, 1 MiB, 4 MiB, 16 MiB

# TLB PID and Global Bit

- **Process ID (PID) allows multiple processes to coexist**
  - We don't need to flush the TLB on context switch
  - By setting the process ID
  - Only flush TLB entries when reusing a PID
  - Current PID is stored in `c0_entryhi`
- **Global bit**
  - Used for pages shared across all address spaces in kseg2 or useg
  - Ensures the TLB ignores the PID field
  - Typically in most hardware a TLB flush doesn't flush global pages

# TLB Instructions

- **MIPS co-processor 0 (COP0) provides the TLB functionality**
- `tlbwr`: **TLB write a random slot**
- `tlbwi`: **TLB write a specific slot**
- `tlbr`: **TLB read a specific slot**
- `tlbp`: **Probe the slot containing an address**
- **For each of these instructions you must load the following registers**
  - `c0_entryhi`: high bits of TLB entry
  - `c0_entrylo`: low bits of TLB entry
  - `c0_index`: TLB Index

# Hardware Lookup Exceptions

- **TLB Exceptions:**
  - UTLB Miss: Generated when the accessing useg without matching TLB entry
  - TLB Miss: Generated when the accessing kseg2 without matching entry
  - TLB Mod: Generated when writing to read-only page
- **UTLB handler is seperate from general exception handler**
  - UTLBs are very frequent and require a hand optimized path
  - 64 entry TLB with 4 kiB pages covers 256 kiB of memory
  - Modern machines have workloads with far more memory
  - Require more entries (expensive hardware) or larger pages

# Hardware Lookup Algorithm

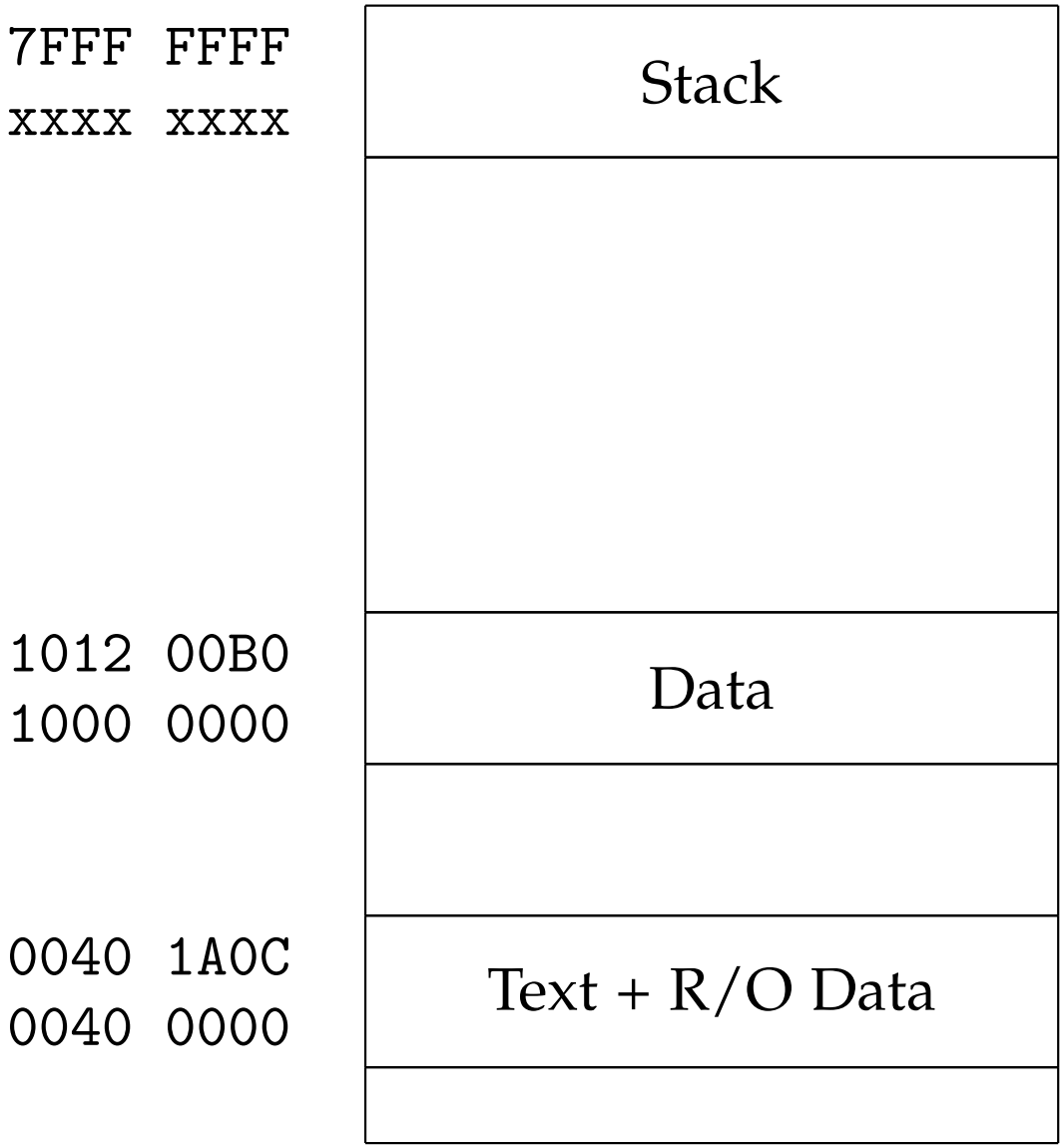
- **If most significant bit (MSB) is 1 and in user mode $\rightarrow$ address error exception.**
- **If no VPN match $\rightarrow$ TLB miss exception if MSB is 1, otherwise UTLB miss.**
- **If PID mismatches and global bit not set $\rightarrow$ generate a TLB miss or UTLB miss.**
- **If valid bit not set $\rightarrow$ TLB miss.**
- **Write to read-only page $\rightarrow$ TLB mod exception.**
- **If N bit is set directly access device memory (disable cache)**

# OS/161 Assembly Wrappers

- `tlb_random`: **Write random TLB entry**
- `tlb_write`: **Write specific TLB entry**
- `tlb_read`: **Read specific TLB entry**
- `tlb_probe`: **Lookup TLB entry**
- **Currently the OS implements segments using paging hardware**
- **In a later assignment you will implement a Radix tree (like x86)**

# OS/161 Memory Layout

- **Example Memory Layout: user/testbin/sort**



# Paging in day-to-day use

- **Paging Examples**
  - Demand paging
  - Growing the stack
  - BSS page allocation
  - Shared text
  - Shared libraries
  - Shared memory
  - Copy-on-write (`fork`, `mmap`, etc.)
- **Next time: detailed discussion on MIPS**