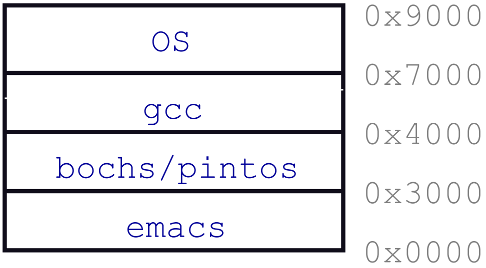
CS350: Operating Systems

Lecture 7: Virtual Memory – Hardware

Ali Mashtizadeh

University of Waterloo

Want processes to co-exist

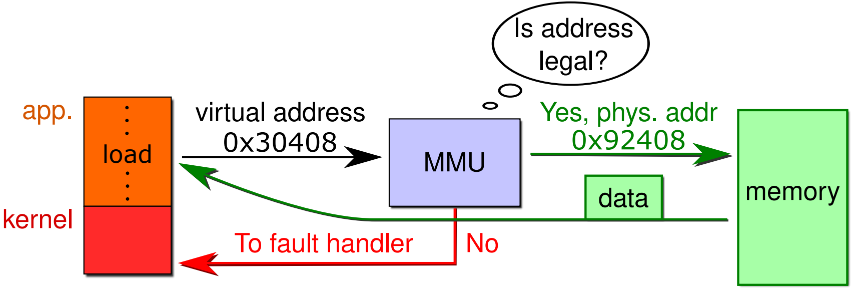


- Consider multiprogramming on physical memory
  - What happens if emacs needs to expand?
  - If emacs needs more memory than is on the machine??
  - If emacs has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs isn't using its memory?

Issues in sharing physical memory

- Protection
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process A from trashing B's memory
  - Also prevent A from even observing B's memory (ssh-agent)
- Transparency
  - A process shouldn't require particular physical memory bits
  - Yes processes often require large amounts of contiguous memory (for stack, large data structures, etc.)
- Resource exhaustion
  - Programmers typically assume machine has "enough" memory
  - Sum of sizes of all processes often greater than physical memory

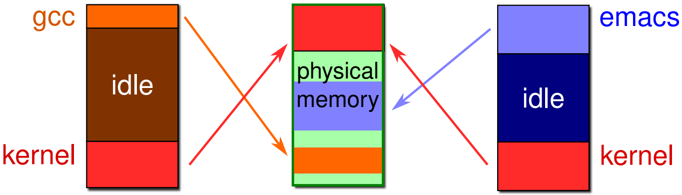
Virtual memory goals



- Give each program its own "virtual" address space
  - At run time, Memory-Management Unit relocates each load, store to actual memory... App doesn't see physical memory
- Also enforce protection
  - Prevent one app from messing with another's memory
- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk

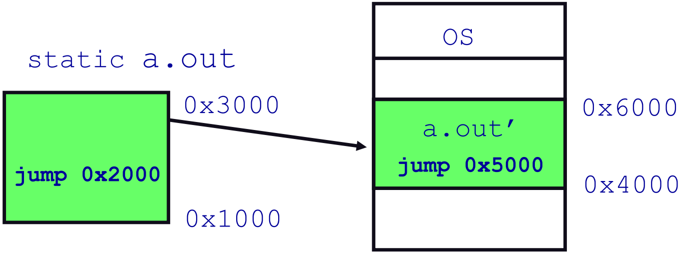
Virtual memory advantages

- Can re-locate program while running
  - Run partially in memory, partially on disk
- Most of a process's memory may be idle (80/20 rule).



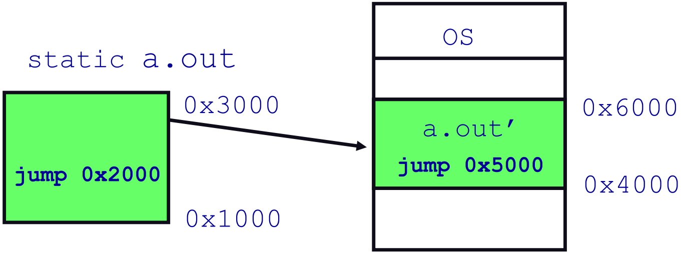
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch
    (Not using a memory region? switch it to another process)
- Challenge: VM = extra layer, could be slow

Idea 1: load-time linking



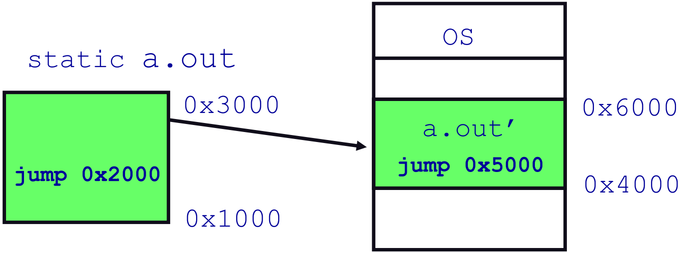
- *Linker* patches addresses of symbols like `printf`
- Idea: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?

Idea 1: load-time linking



- *Linker* patches addresses of symbols like `printf`
- Idea: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?
  - How to enforce protection
  - How to move once already in memory (Consider: data pointers)
  - What if no contiguous free region fits program?

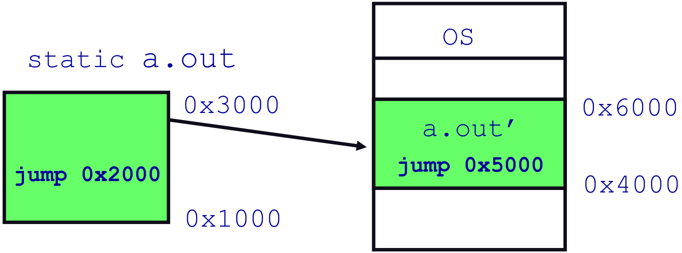
Idea 2: base + bound register



- Two special privileged registers: base and bound
- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- How to move process in memory?

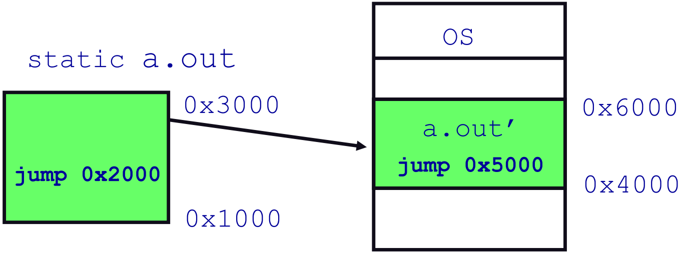
- What happens on context switch?

Idea 2: base + bound register



- Two special privileged registers: base and bound
- On each load/store:
  - Physical address $=$ virtual address $+$ base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?

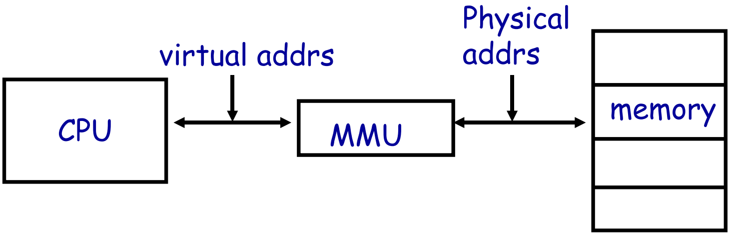
Idea 2: base + bound register



- Two special privileged registers: base and bound
- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?
  - OS must re-load base and bound register

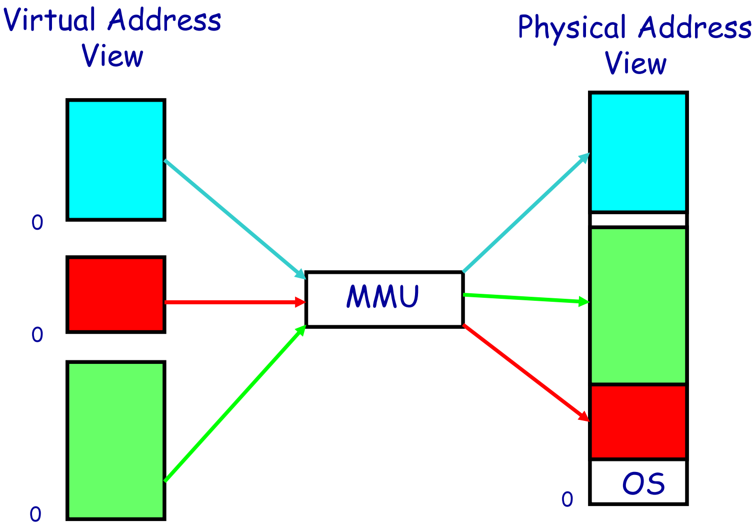
Definitions

- Programs load/store to virtual (or logical) addresses
- Actual memory uses physical (or real) addresses
- VM Hardware is Memory Management Unit (MMU)



  - Usually part of CPU
  - Accessed w. privileged instructions (e.g., load bound reg)
  - Translates from virtual to physical addresses
  - Gives per-process view of memory called address space

Address space



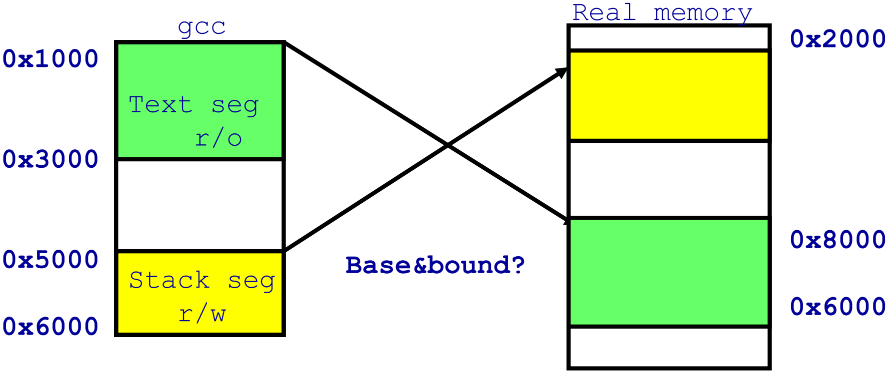
Base+bound trade-offs

- Advantages
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme
- Disadvantages
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two copies of bochs)
- One solution: Multiple segments
  - E.g., separate code, stack, data segments
  - Possibly multiple data segments

Outline

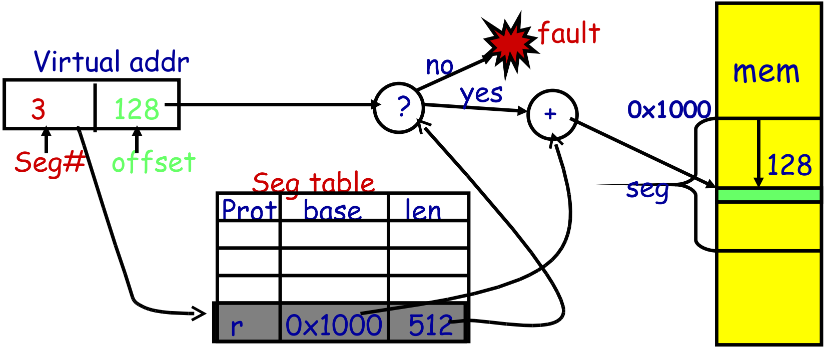
1. Segmentation
2. Paging
3. MIPS: Software Managed MMU
4. Intel x86: Hardware MMU

Segmentation



- Let processes have many base/bound regs
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- Must specify segment as part of virtual address

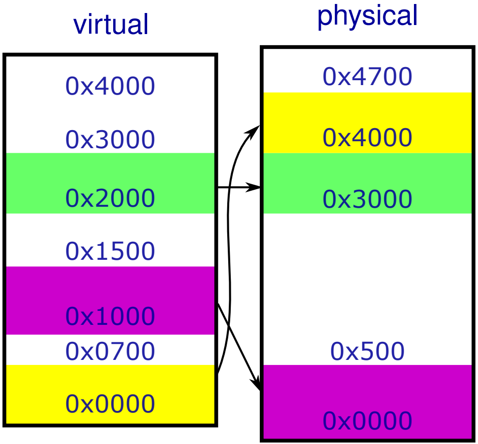
Segmentation mechanics



- Each process has a segment table
- Each VA indicates a segment and offset:
  - Top bits of addr select segment, low bits select offset (PDP-10)
  - Or segment selected by instruction or operand (means you need wider "far" pointers to specify segment)

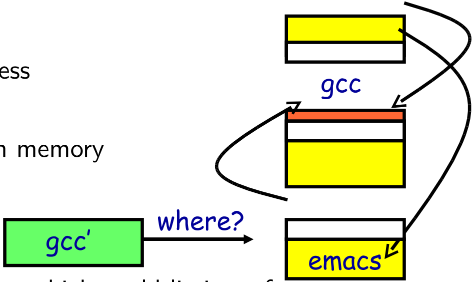
Segmentation example

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



- 2-bit segment number (1st digit), 12 bit offset (last 3)
  - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

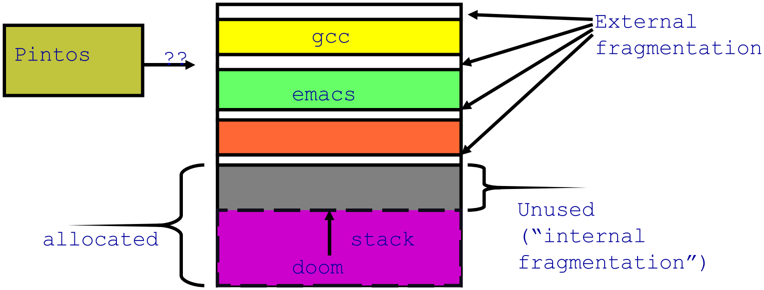
Segmentation trade-offs



- Advantages
  - Multiple segments per process
  - Allows sharing! (how?)
  - Don't need entire process in memory
- Disadvantages
  - Requires translation hardware, which could limit performance
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
  - *n* byte segment needs *n contiguous* bytes of physical memory
  - Makes *fragmentation* a real problem.

Fragmentation

- Fragmentation → Inability to use free memory
- Over time:
  - Variable-sized pieces = many small holes (external fragmentation)
  - Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)



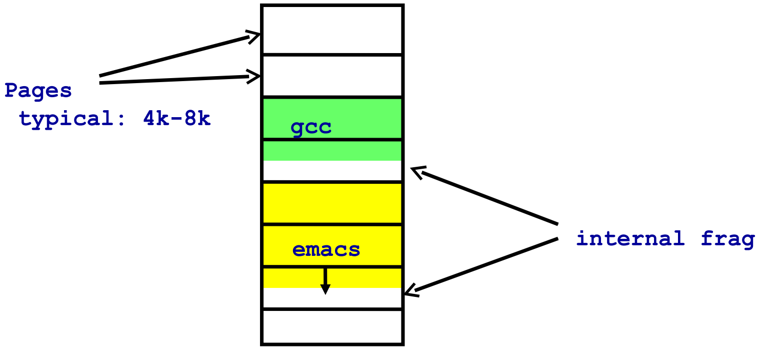
Outline

1. Segmentation
2. Paging
3. MIPS: Software Managed MMU
4. Intel x86: Hardware MMU

Paging

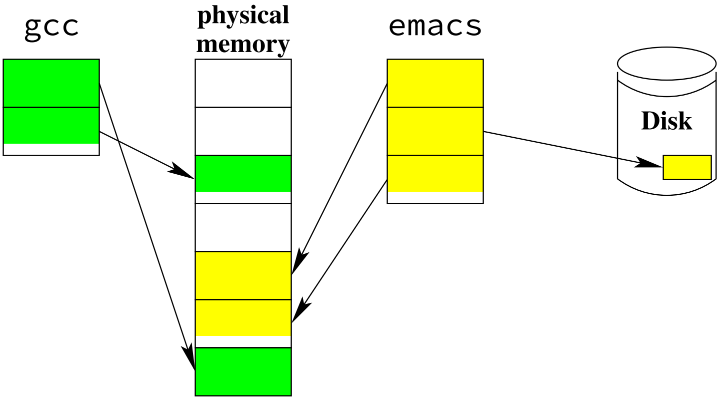
- Divide memory up into small *pages*
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set "accessed" and "dirty" bits
  - Control page execute permission separately from read/write
  - Control caching or memory consistency of page

Paging trade-offs



- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per “segment”

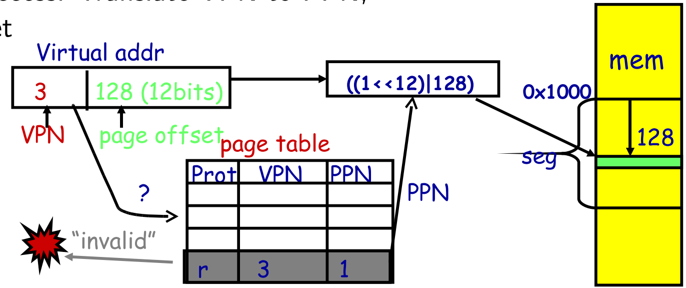
Simplified allocation



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

Paging data structures

- Pages are fixed size, e.g., 4K
  - Least significant 12 ($\log_2$ 4K) bits of address are *page offset*
  - Most significant bits are *page number*
- Each process has a *page table*
  - Maps *virtual page numbers* (VPNs) to *physical page numbers* (PPNs)
  - Also includes bits for protection, validity, etc.
- On memory access: Translate VPN to PPN, then add offset



Example: Paging on PDP-11

- 64K virtual memory, 8K pages
  - Separate address space for instructions & data
  - I.e., can't read your own instructions with a load
- Entire page table stored in registers
  - 8 Instruction page translation registers
  - 8 Data page translations
- Swap 16 machine registers on each context switch

MMU Types

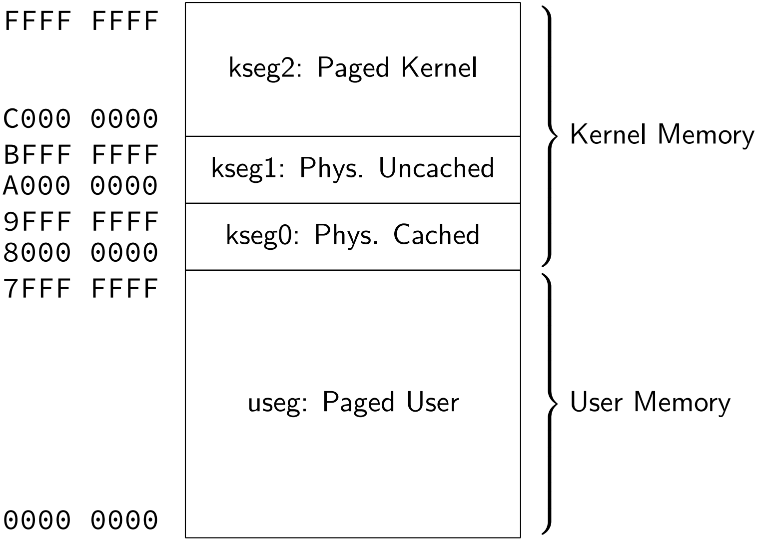
- Memory Management Units (MMU) come in two flavors
- Software Managed
  - Simplier hardware and asks software to reload pages
  - Requires fast exception handling and optimized software
  - Enables more flexiblity in the TLB (e.g. variable page sizes)
  - Examples: MIPS, Sun SPARC, DEC Alpha, ARM and POWER
- 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+

Outline

Software Managed 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!
  - Combination of hash tables, trees and list of VM regions (next lecture)

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

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Frame Number (VPN)																				PID											

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number (PPN)																				N	D	V	G								

- 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
  - COP0 provides most privileged functionality
- Four instructions:
  - `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

1. If most significant bit (MSB) is 1 and in user mode → address error exception
2. If no VPN match → TLB/UTLB miss exception
3. If PID mismatches and global bit not set → TLB/UTLB miss
4. If valid bit not set → TLB/UTLB miss
5. Write to read-only page → TLB mod(ification) exception
6. If N bit is set directly access device memory (disable cache)

Outline

1. Segmentation
2. Paging
3. MIPS: Software Managed MMU
4. Intel x86: Hardware MMU

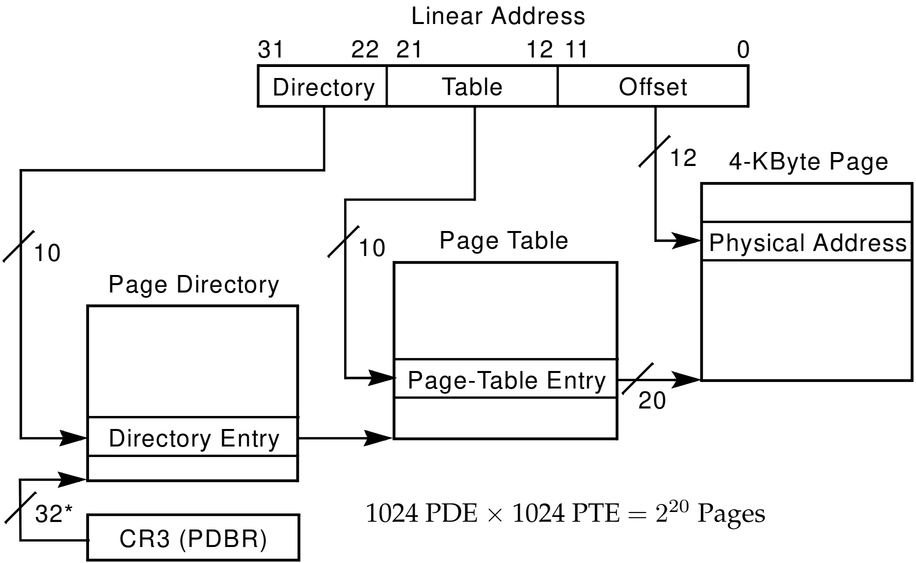
Hardware Managed MMU: x86

- TLB Managed by Hardware and Microcode
  - Two levels of TLBs each acting as a cache
  - Typical: a 1K entry TLB and 512 entry TLB
  - TLB acts as a cache page table structure
  - TLB automatically reloaded from page table
  - Missing in the page tables result in page faults
- OS builds a Radix-tree describing memory layout
  - Control register `%cr3` points to radix-tree root
- 32-bit mode uses two level radix tree
  - 1024 entries per level with page sizes 4 KiB or 4 MiB
- 64-bit mode
  - 512 entries per level with page sizes of 4 KiB, 2 MiB, 1 GiB
  - Four levels by default, newer chips support 5 levels

x86 Paging

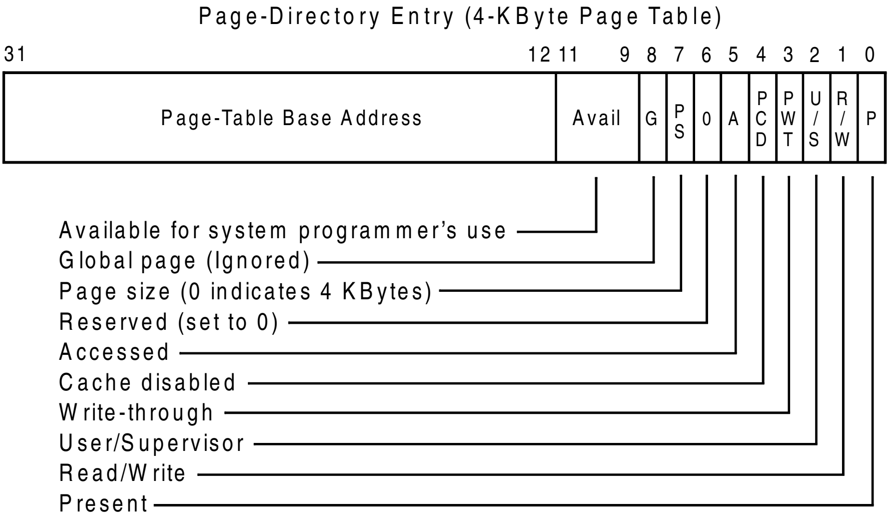
- Paging enabled by bits in a control register (`%cr0`)
  - Only privileged OS code can manipulate control registers
- Normally 4KB pages
- `%cr3`: points to 4KB page directory
- Page directory: 1024 PDEs (page directory entries)
  - Each contains physical address of a page table
- Page table: 1024 PTEs (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

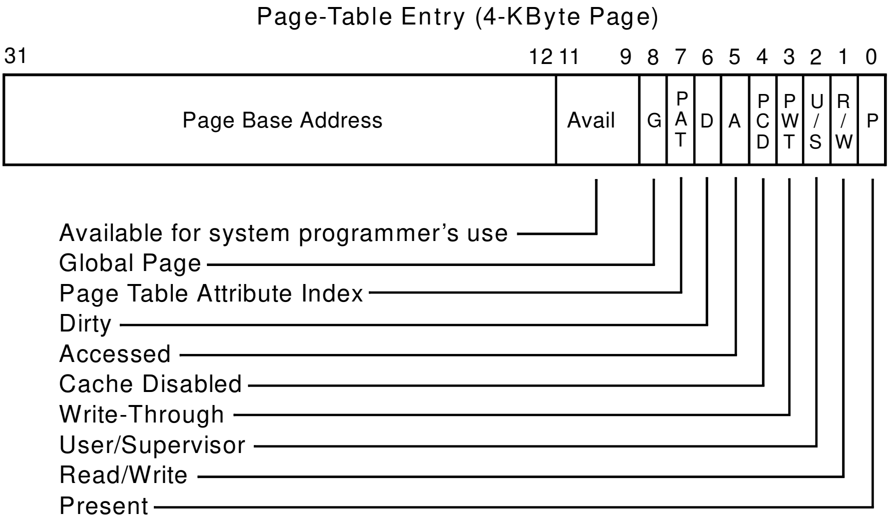


$$1024 \text{ PDE} \times 1024 \text{ PTE} = 2^{20} \text{ Pages}$$

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
- Implementation Details
  - Segments defined through descriptors (similar IDTs)
  - Two descriptor tables: GDT (global) and LDT (local – usually per process)
  - Bonus: TSS used by interrupts is also a descriptor in the GDT
  - Segment registers: `%cs` (code), `%ds` (data), `%ss` (stack), `%fs`/`%gs` (cpu/thread local data)
- x86-64 keeps segmentation offsets for `%fs`/`%gs`
  - Early AMD64's kept segmentation for virtualization
  - Now: Offset set using model specific registers `MSR_FSBASE`/`MSR_GSBASE`

Why have segmentation and paging?

- Why do you want *both* paging and segmentation?

Why have segmentation and paging?

- 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 → 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)

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

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 operating system side