CS 318 Principles of Operating Systems

Fall 2018

Lecture 9: Virtual Memory

Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Slides adapted from Geoff Voelker's and David Mazières' lectures

Administrivia

- **Lab 2 out**
  - Does not depend on Lab 1:
    - You can either build on your lab1 submission: `git checkout -b lab2-handin`
    - Or start from beginning: `git checkout -b lab2-handin 213ffab`
  - Content mostly about syscalls
    - Only requires very basic knowledge about Virtual Memory (Lab 3 is on VM), start now
  - Due Thursday 10/18 11:59 pm
- **Lab 2 review session**
  - Wednesday (10/03) from 3:30pm to 5:00pm in Malone G33/G35
- **Homework 3 out**
  - Exercises to practice with synchronization lectures

Memory Management

Next few lectures are going to cover memory management

- **Goals of memory management**
  - To provide a convenient abstraction for programming
  - To allocate scarce memory resources among competing processes to maximize performance with minimal overhead
- **Mechanisms**
  - Physical and virtual addressing (1)
  - Techniques: partitioning, paging, segmentation (1)
  - Page table management, TLBs, VM tricks (2)
- **Policies**
  - Page replacement algorithms (3)

Lecture Overview

- **Virtual memory warm-up**
- **Survey techniques for implementing virtual memory**
  - Fixed and variable partitioning
  - Paging
  - Segmentation
- **Focus on hardware support and lookup procedure**
  - Next lecture we'll go into sharing, protection, efficient implementations, and other VM tricks and features

Virtual Memory

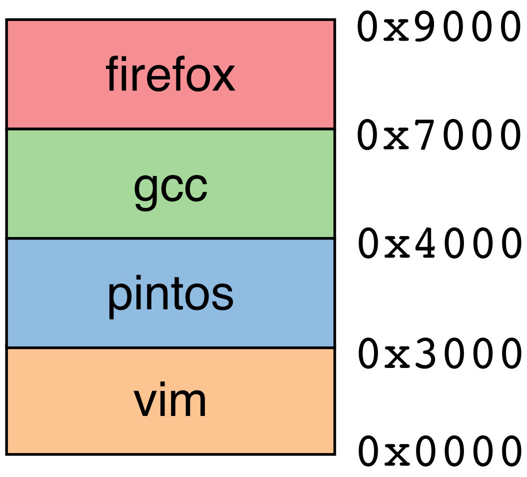
- **The abstraction that the OS provides for managing memory**
  - VM enables a program to execute with less physical memory than it “needs”
    - Can also run on a machine with “too much” physical memory
  - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
  - OS will adjust memory allocation to a process based upon its behavior
  - VM requires hardware support and OS management algorithms to pull it off
- **Let’s go back to the beginning…**

In the beginning…

- **Rewind to the days of "second-generation" computers**
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it

- **Multiprogramming changes all of this**
  - Want multiple processes in memory at once

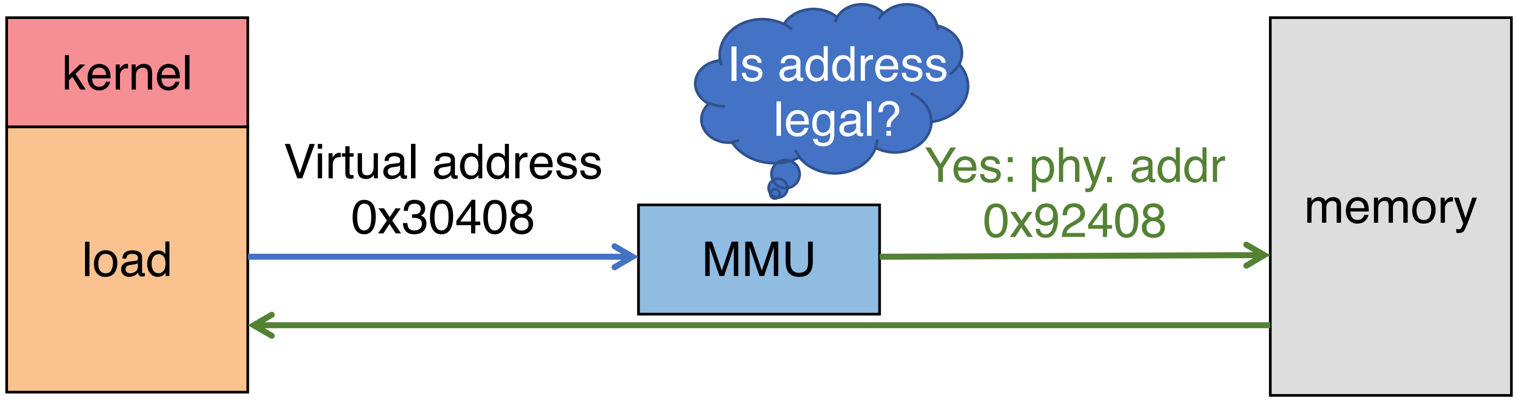
- **Consider multiprogramming on physical memory**
  - What happens if pintos needs to expand?
  - If vim needs more memory than is on the machine?
  - If pintos has an error and writes to address `0x7100`?
  - When does gcc have to know it will run at `0x4000`?
  - What if vim isn't using its memory?



Issues in Sharing Physical Memory

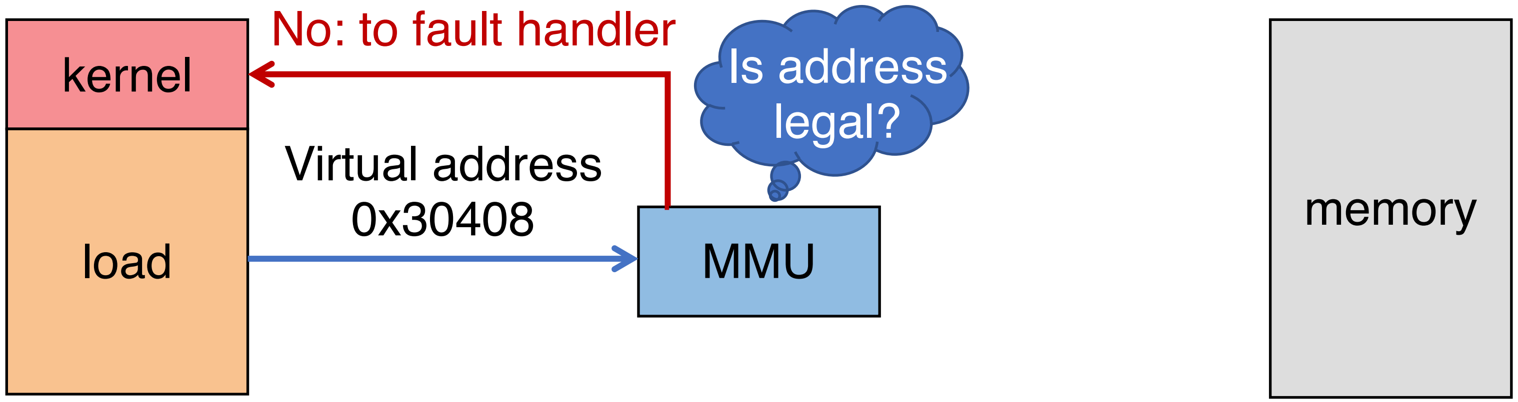
- **Protection**
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process *A* from trashing *B*'s memory
  - Also prevent *A* from even observing B's memory (ssh-agent)
- **Transparency**
  - A process shouldn't require particular physical memory bits
  - Yet processes 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 runtime, *Memory-Management Unit* (MMU) relocates each load/store
  - Application doesn't see physical memory addresses
- **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 Goals

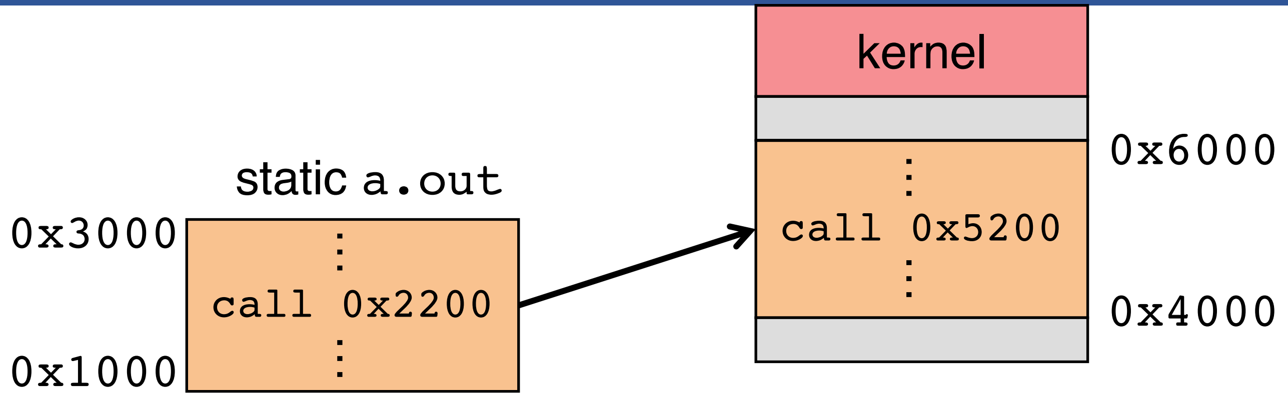


- **Give each program its own virtual address space**
  - At runtime, *Memory-Management Unit* (MMU) relocates each load/store
  - Application doesn't see physical memory addresses
- **Enforce protection**
  - Prevent one app from messing with another's memory
- **And allow programs to see more memory than exists**
  - Somehow relocate some memory accesses to disk

Virtual Memory Advantages

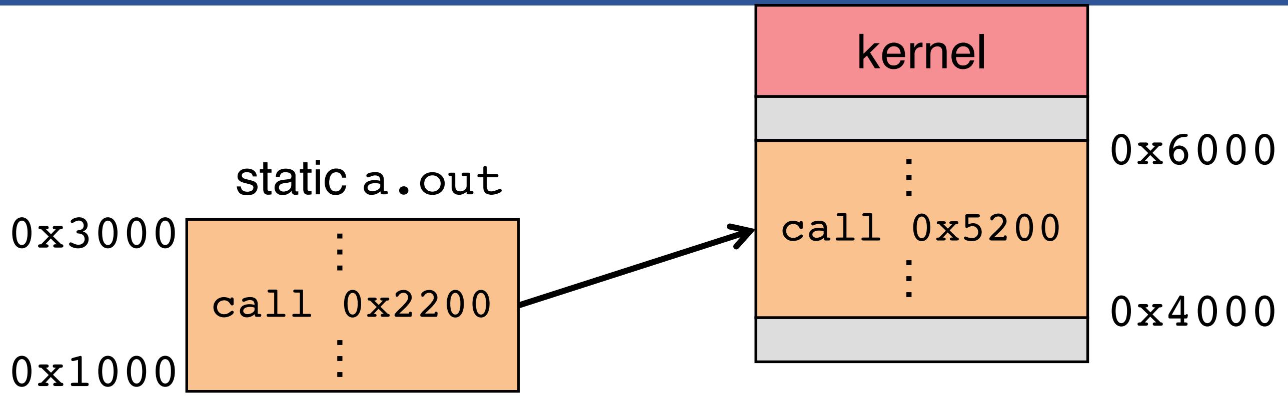
- **Can re-locate program while running**
  - Run partially in memory, partially on disk
- **Most of a process's memory may be idle (80/20 rule)**
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch (Not using a memory region? switch it to another process)
- **Challenge: VM = extra layer, could be slow**

Idea 1: Load-time Linking



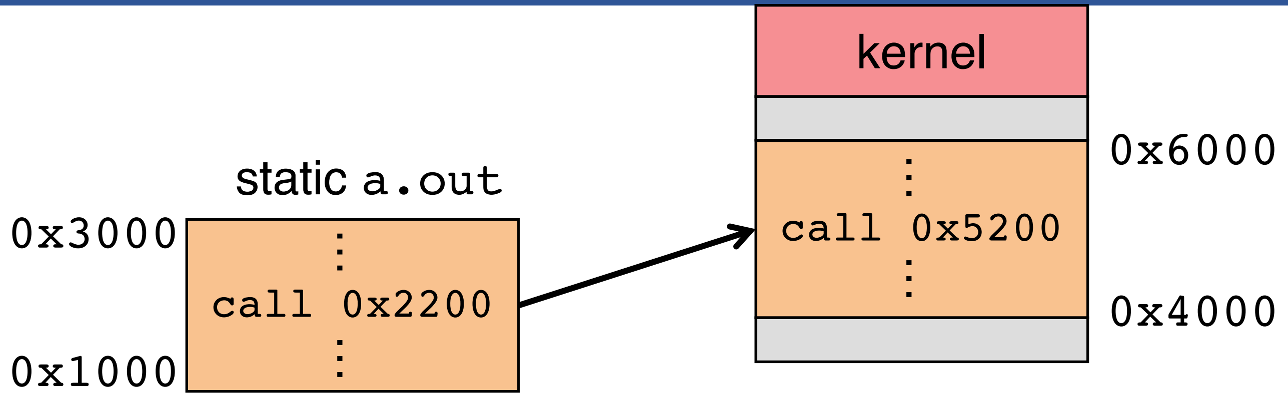
- **Linker patches addresses of symbols like `printf`**
- **Idea: link when process executed, not at compile time**
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- **Problems?**

Idea 1: Load-time Linking



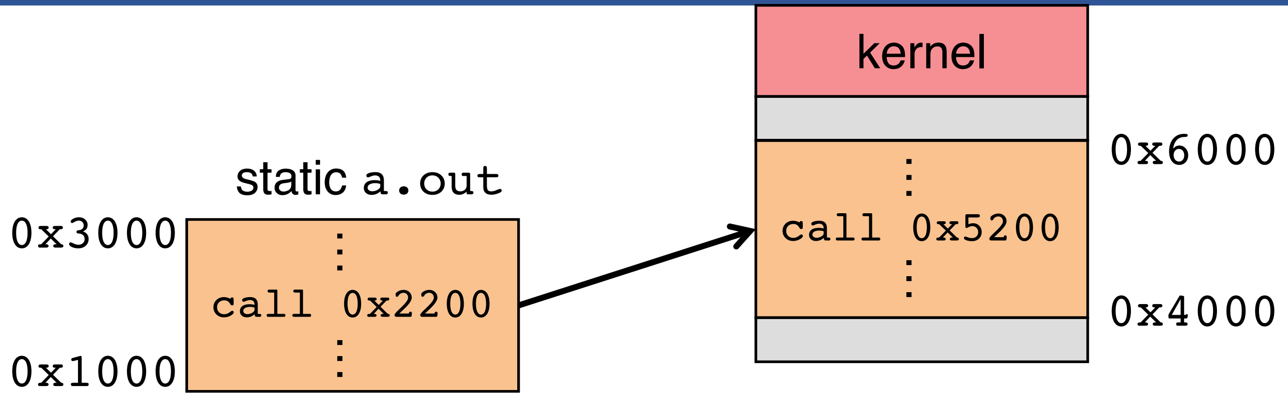
- **Linker patches addresses of symbols like `printf`**
- **Idea: link when process executed, not at compile time**
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- **Problems?**
  - How to enforce protection?
  - How to move once already in 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/jump:**
  - Physical address = virtual address + base
  - Check 0 ≤ virtual address < bound, else trap to kernel
- **How to move process in memory?**
- **What happens on context switch?**

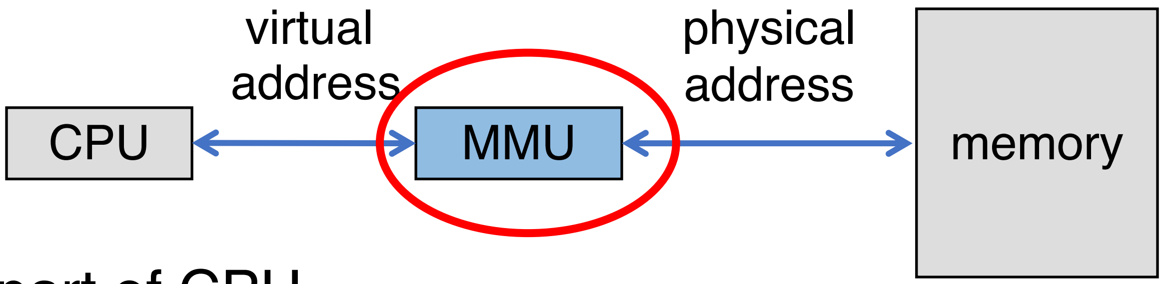
Idea 2: Base + Bound Register



- **Two special privileged registers: base and bound**
- **On each load/store/jump:**
- **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 addresses**
- **Actual memory uses physical addresses**
- **VM Hardware is Memory Management Unit (MMU)**



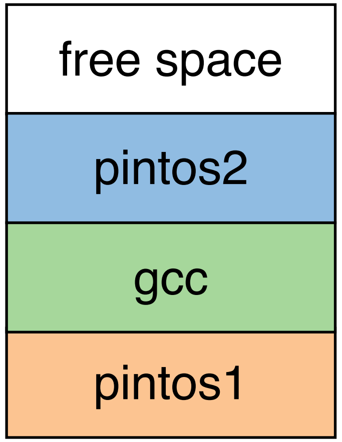
- Usually part of CPU
  - Configured through privileged instructions (e.g., load bound reg)
- Translates from virtual to physical addresses
- Gives per-process view of memory called address space

Base + Bound Trade-offs

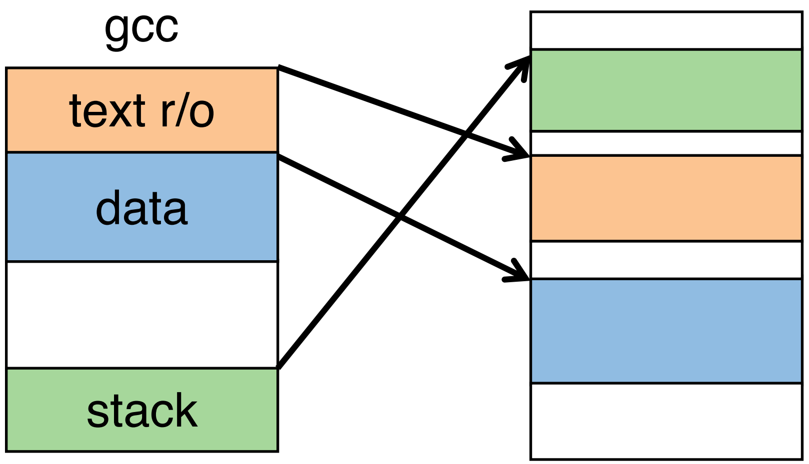
- **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme
- **Disadvantages**

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, both running pintos)
- **One solution: Multiple segments**
  - E.g., separate code, stack, data segments
  - Possibly multiple data segments

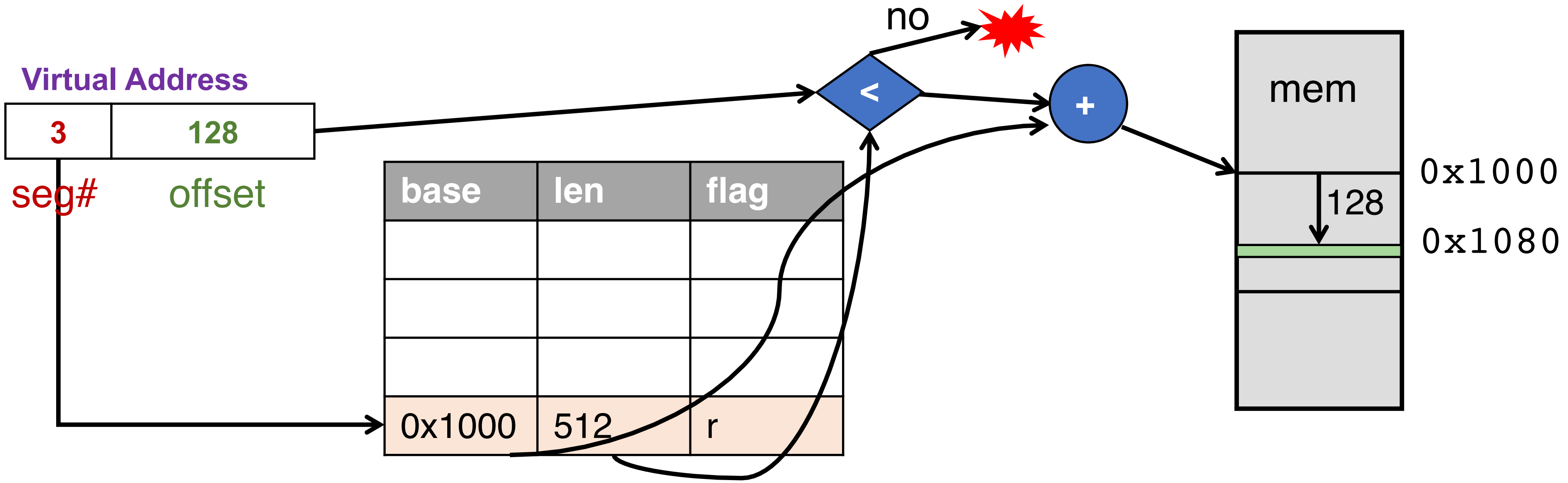


Segmentation



- **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**

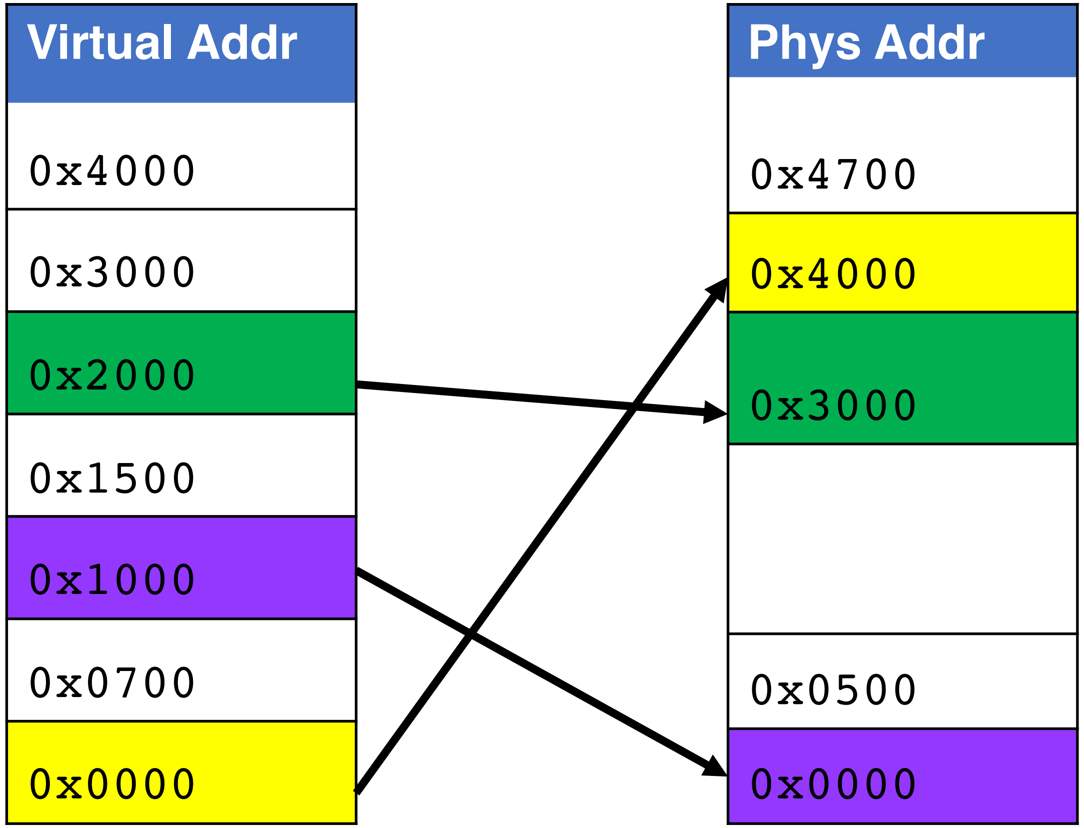
Segmentation Mechanics



- **Each process has a segment table**
- **Each VA indicates a segment and offset:**
  - Top bits of addr select segment, low bits select offset
  - x86 stores segment #s in registers (CS, DS, SS, ES, FS, GS)

Segmentation Example

Segment	Base	Bound	RW
0	`0x4000`	`0x6ff`	`10`
1	`0x0000`	`0x4ff`	`11`
2	`0x3000`	`0xfff`	`11`
3			`00`



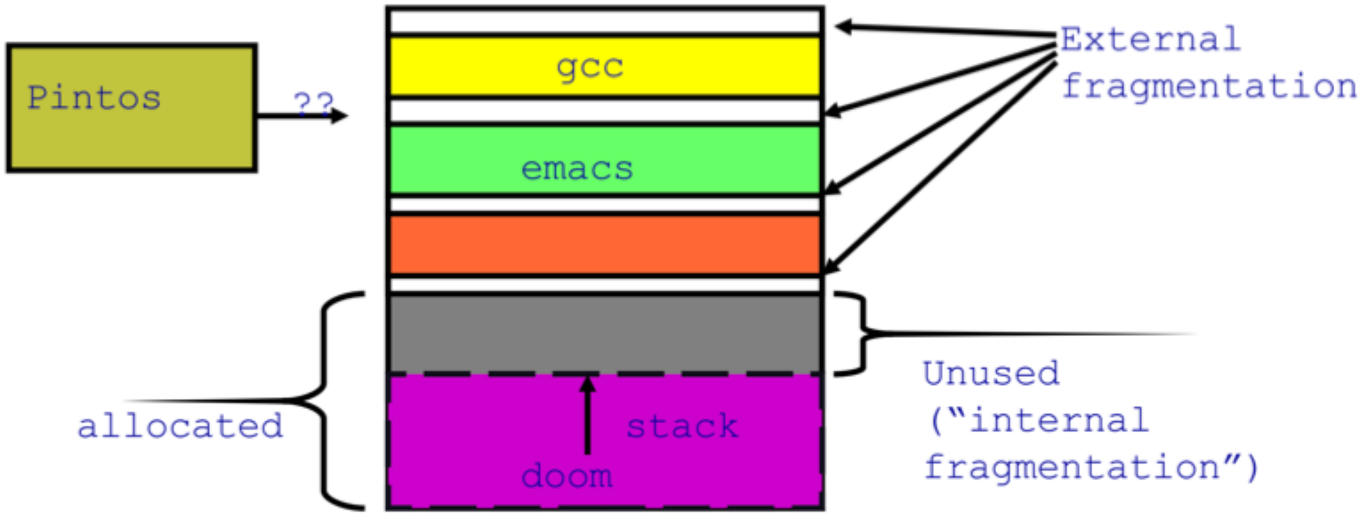
- **2-bit segment number (1st digit), 12 bit offset (last 3)**
  - Where is `0x0240`? `0x1108`? `0x265c`? `0x3002`? `0x1600`?

Segmentation Trade-offs

- **Advantages**
  - Multiple segments per process
  - Can easily share memory! (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)



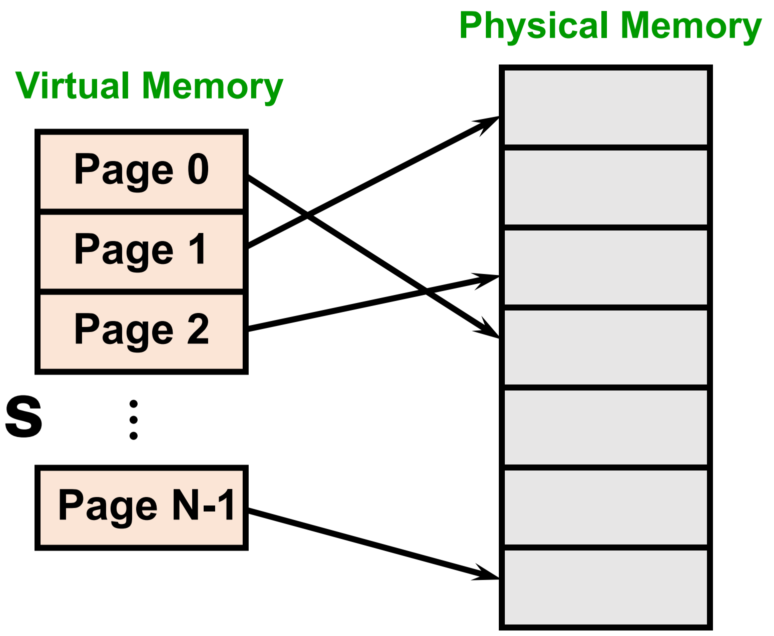
Alternatives to Hardware MMU

- **Language-level protection (Java)**
  - Single address space for different modules
  - Language enforces isolation
  - Singularity OS [1] does this
- **Software fault isolation**
  - Instrument compiler output
  - Checks before every store operation prevents modules from trashing each other
  - Google Native Client [2] does this

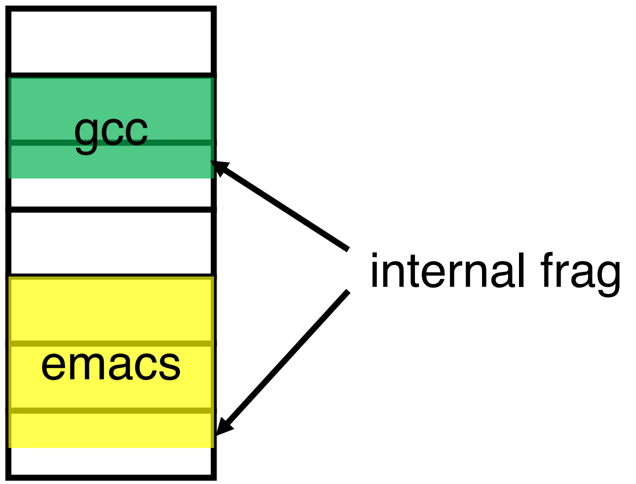
[1]: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/10/tr-2005-135.pdf
[2]: https://developer.chrome.com/native-client

Paging

- **Divide memory up into fixed-size *pages***
  - Eliminates external fragmentation
- **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



Paging Trade-offs



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

Paging Data Structures

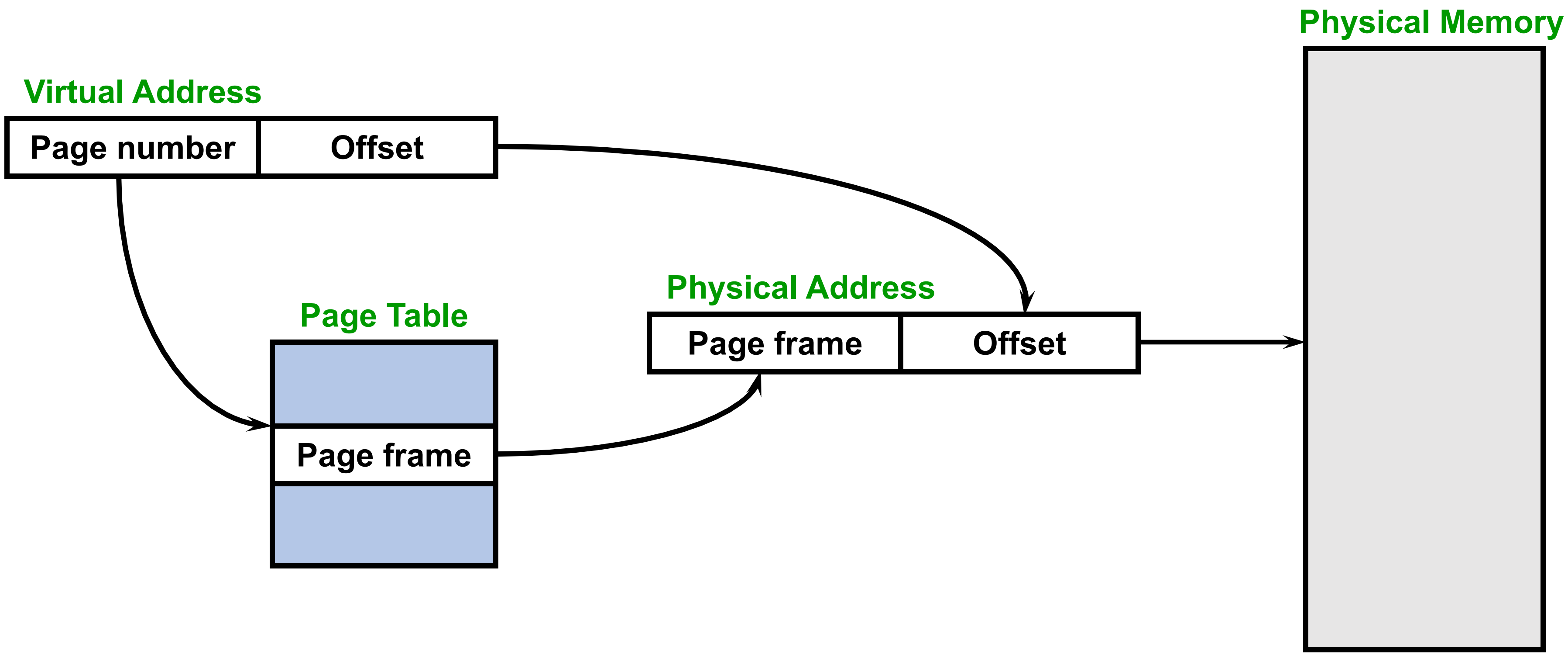
- **Pages are fixed size, e.g., 4K**
  - Virtual address has two parts: virtual page number and offset
  - Least significant 12 ($log_2 4k$) bits of address are page offset
  - Most significant bits are *page number*
- **Page tables**
  - Map virtual page number (VPN) to physical page number (PPN)
    - VPN is the index into the table that determines PPN
    - PPN also called page frame number
  - Also includes bits for protection, validity, etc.
  - One page table entry (PTE) per page in virtual address space

Page Table Entries (PTEs)

1	1	1	2	20
M	R	V	Prot	Physical Page Number

- **Page table entries control mapping**
  - The Modify bit says whether or not the page has been written
    - It is set when a write to the page occurs
  - The Reference bit says whether the page has been accessed
    - It is set when a read or write to the page occurs
  - The Valid bit says whether or not the PTE can be used
    - It is checked each time the virtual address is used
  - The Protection bits say what operations are allowed on page
    - Read, write, execute
  - The Physical page number (PPN) determines physical page

Page Lookups



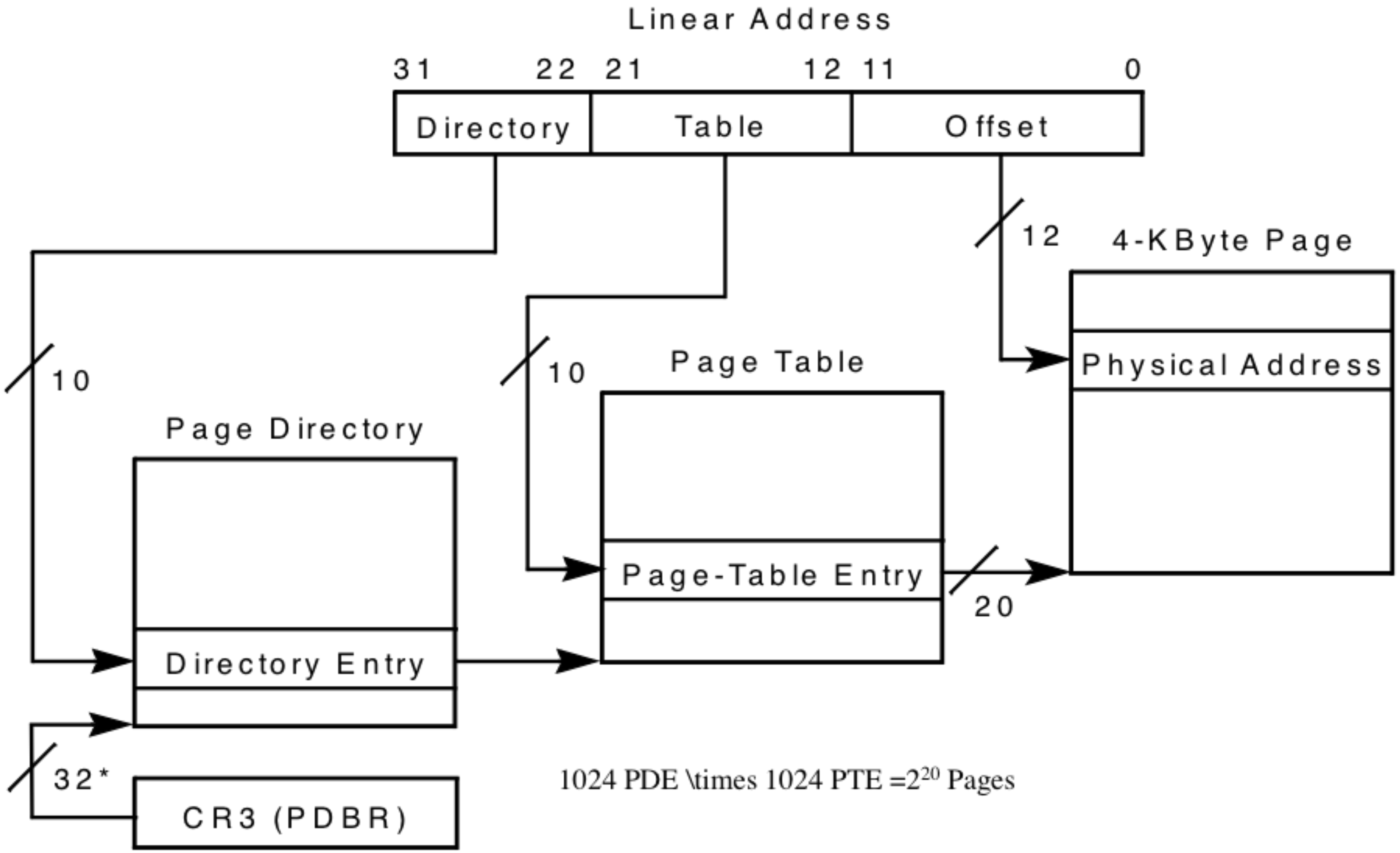
Paging Example

- **Pages are 4K**
  - VPN is 20 bits (2^{20} VPNs), offset is 12 bits
- **Virtual address is** `0x7468`
  - Virtual page is `0x7`, offset is `0x468`
- **Page table entry** `0x7` **contains** `0x2`
  - Physical page number is `0x2`
  - Seventh virtual page is at address `0x2000` (2nd physical page)
- **Physical address =** `0x2000` **+** `0x468` **=** `0x2468`

x86 Paging

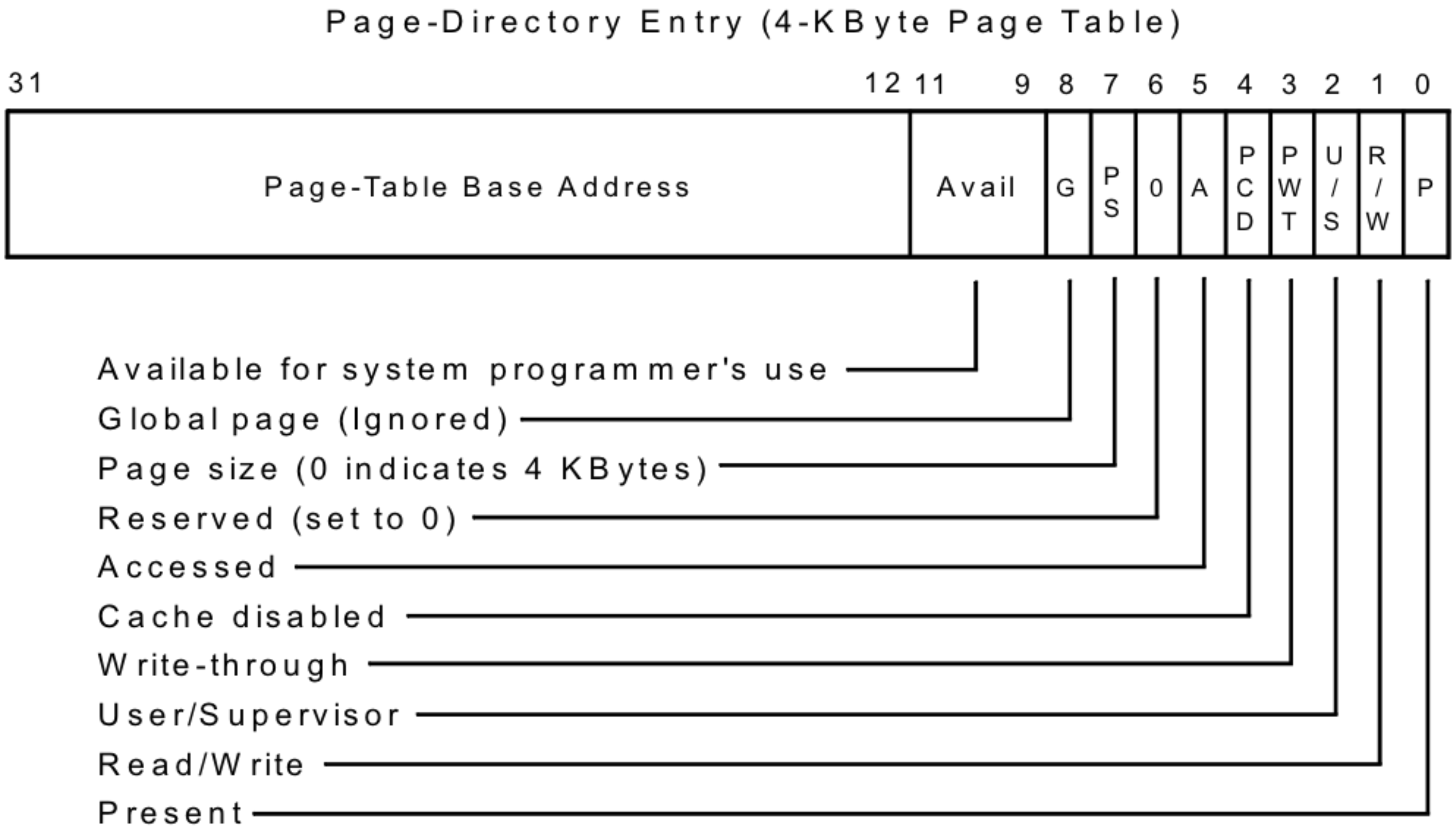
- **Paging enabled by bits in a control register (`%cr0`)**
  - Only privileged OS code can manipulate control registers
- **Normally 4KB pages**
- **`%cr3`: points to 4KB page directory**
  - See `pagedir_activate()` in Pintos `userprog/pagedir.c`
- **Page directory: 1024 PDEs (page directory entries)**
  - 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 old Intel manual for simplest explanation**

x86 Page Translation

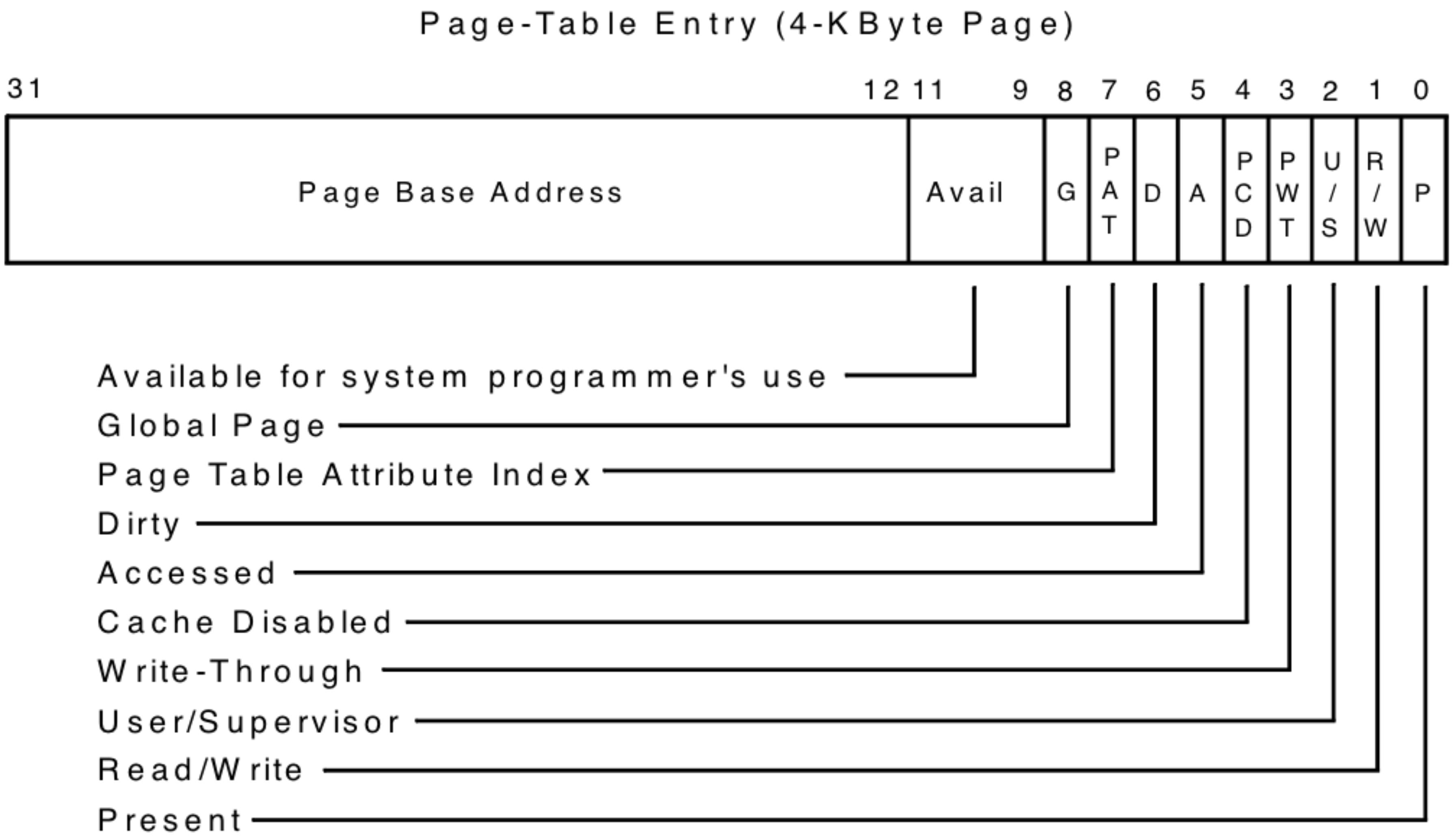


*32 bits aligned onto a 4-KByte boundary

x86 Page Directory Entry



x86 Page Table Entry



Paging Advantages

- **Easy to allocate memory**
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
- **Easy to swap out chunks of a program**
  - All chunks are the same size
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size

Paging Limitations

- **Can still have internal fragmentation**
  - Process may not use memory in multiples of a page
- **Memory reference overhead**
  - 2 or more references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (more later)
- **Memory required to hold page table can be significant**
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - Solution – page the page tables (more later)

x86 Paging and Segmentation

- **x86 architecture supports both paging and 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?**

Why Want Both Paging and Segmentation?

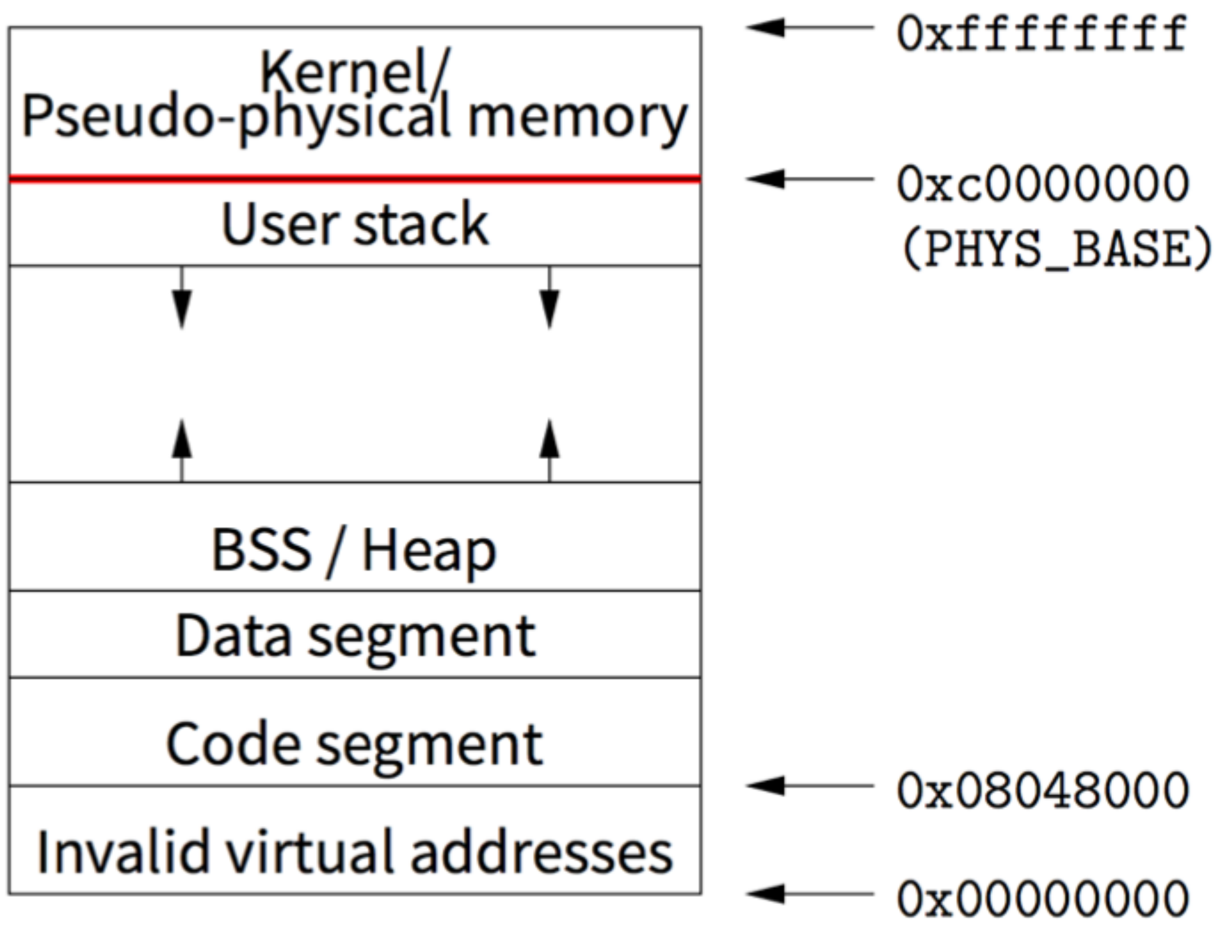
- **Short answer: You don't – just adds overhead**
  - Most OSes use "flat mode" – set base = 0, bounds = `0xffffffff` in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support
- **Long answer: Has some fringe/incidental uses**
  - Use segments for logically related units + pages to partition segments into fixed size chunks
    - Tend to be complex
  - VMware runs guest OS in CPL 1 to trap stack faults

Where Does the OS Live in Memory?

- **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
  - Recent Spectre and Meltdown CPU attacks force OSes to reconsider this [1]
- **Typically all kernel text, most data at same VA in every address space**
  - On x86, must manually set up page tables for this
- **Questions to ponder**
  - Does the kernel have to use VAs during its execution as well?
  - If so, how can OS setup page tables for processes?

[1]: https://lwn.net/Articles/743265/

Pintos Virtual Memory Layout



Summary

- **Virtual memory**
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses
- **Various techniques**
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user's perspective
  - Combine paging and segmentation – not really needed

Next time…

- **Chapters 19, 20**