CE 440 Introduction to Operating System

Lecture 9: Virtual Memory I
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

Administrivia

This Wednesday is project hacking day

- No class, work on lab 1
- I will hold office hour in PHO210 at the lecture time

Homework 1 and 2 is released on course website

Memory Management

Next four lectures are going to cover memory management

Goals of memory management

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

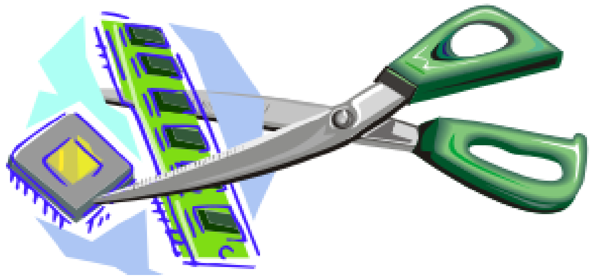
Virtual memory warm-up

Survey techniques for implementing virtual memory

- Fixed and variable partitioning
- Paging
- Segmentation

Focus on hardware support and lookup procedure

Virtualizing Resources



Different Processes/Threads share the same hardware

- Need to multiplex CPU (just finished: scheduling)
- Need to multiplex use of memory (starting today)
- Need to multiplex disk and devices (later in term)

Why worry about memory sharing?

- The working state of a process is defined by its data in memory
- Consequently, cannot just let different threads of control use the same memory
  - Two different pieces of data cannot occupy the same locations in memory
- Different processes do not have access to each other's memory

Virtual Memory

The abstraction that the OS provides for managing memory

- VM enables a program to execute with less physical memory than it “needs”

How?

- Many programs do not need all of their code and data at once (or ever)
- OS will adjust memory allocation to a process based upon its behavior
- VM requires hardware support and OS management algorithms to pull it off

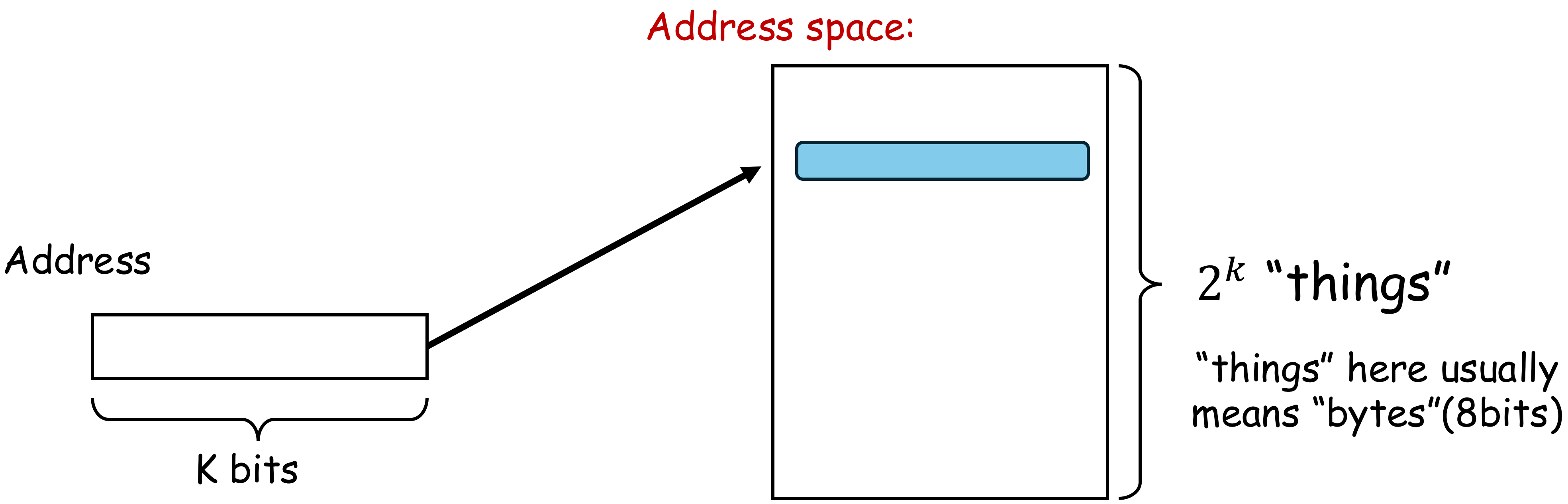
Let’s go back to the beginning…

In the beginning…

Rewind to the days of “second-generation” computers

- Programs use physical addresses directly
- OS loads job, runs it, unloads it

The Basics: Address and Address Space



What is 2^{10} bytes (where a byte is abbreviated as "B")?

- 2^{10} B = 1024B = 1 KB

How many bits to address each byte of 4KB page?

- 4KB = 4×1KB = 4× 2^{10} = 2^{12} → 12 bits

How much memory can be addressed with 20 bits? 32 bits? 64 bits?

- Use 2k

In the beginning…

Rewind to the days of "second-generation" computers

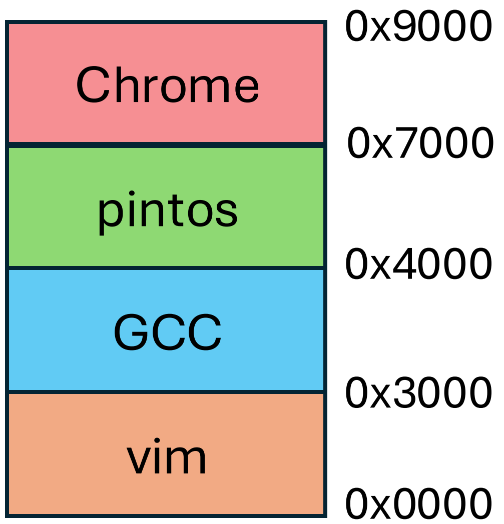
- Programs use physical addresses directly
- OS loads job, runs it, unloads it

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?



Process Virtual Address Space

Definition: Set of accessible addresses and the state associated with them

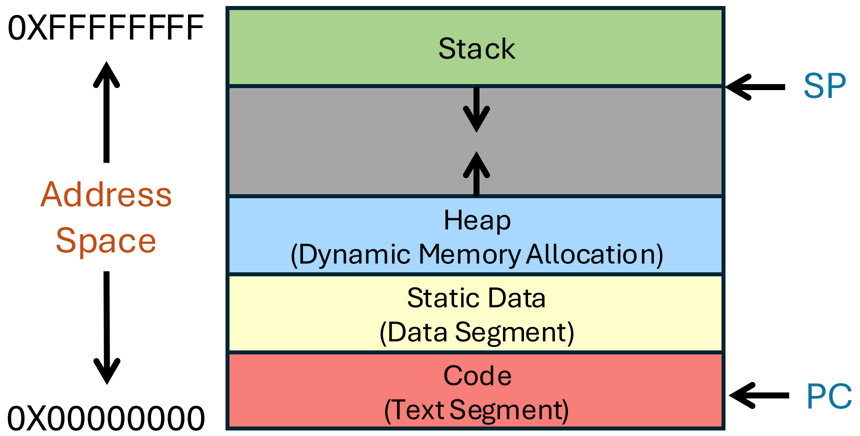
- 2^{32}= ~4 billion **bytes** on a 32-bit machine

How many 32-bit numbers fit in this address space?

- 32-bits = 4 bytes, so $2^{32}/4 = 2^{30}$ =~1 billion

What happens when processor reads or writes to an address?

- Perhaps acts like regular memory
- Perhaps causes I/O operation
  - (Memory-mapped I/O)
- Causes program to abort (segfault)?
- Communicate with another program
- …



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)

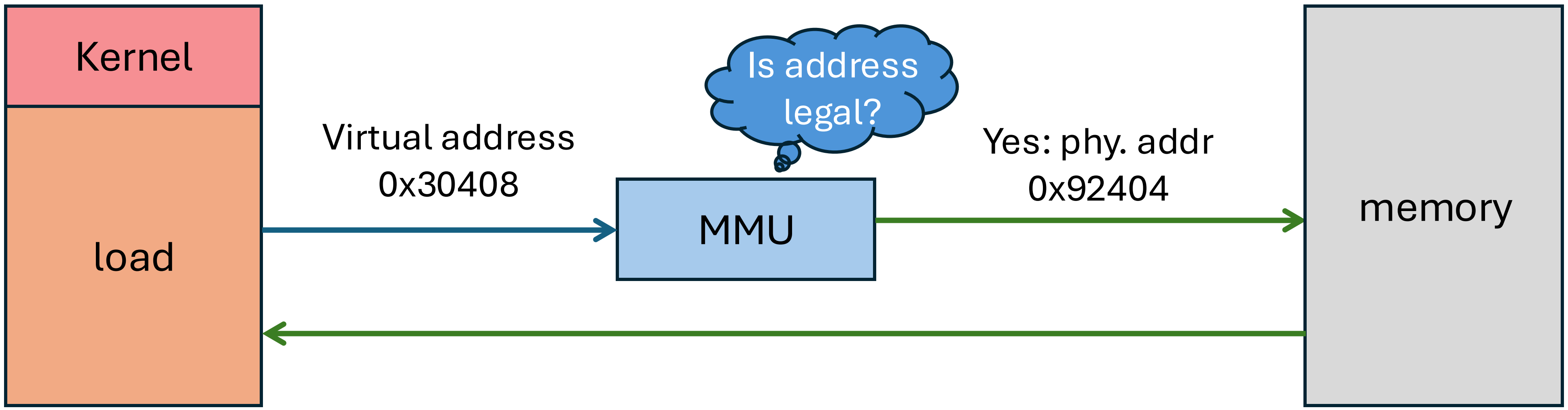
Transparency

- 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

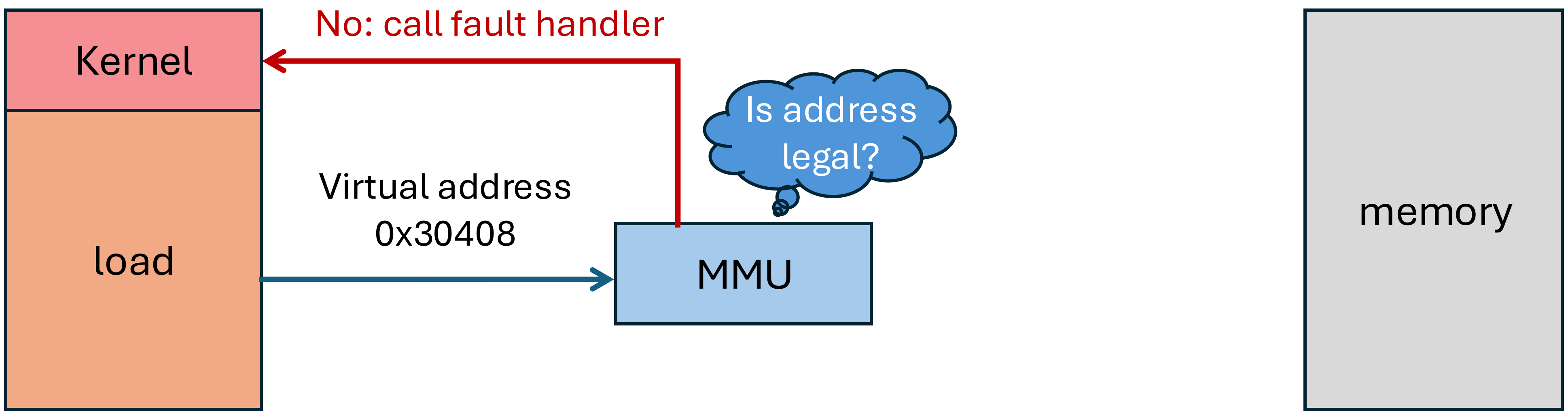
Enforce protection

- 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

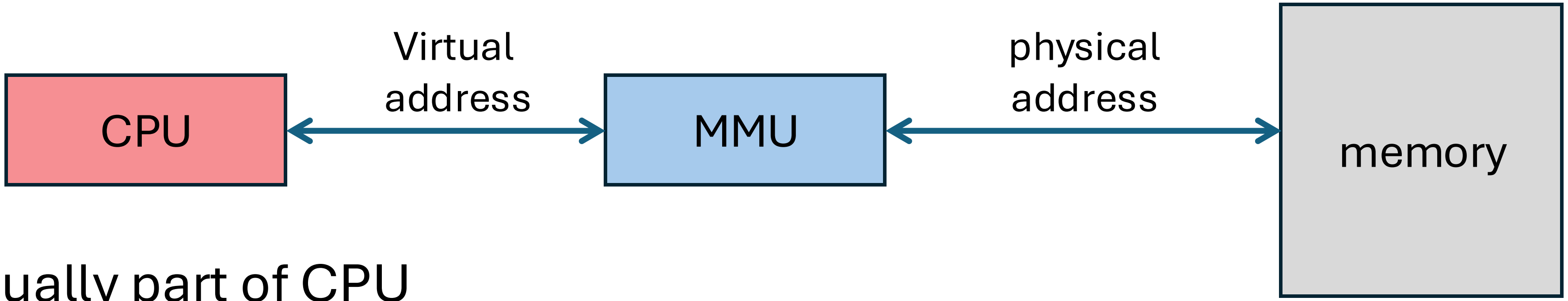
- Somehow relocate some memory accesses to disk

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
- Translates from virtual to physical addresses
- Gives per-process view of memory called address space

Virtual Memory Advantages

Can re-locate program while running

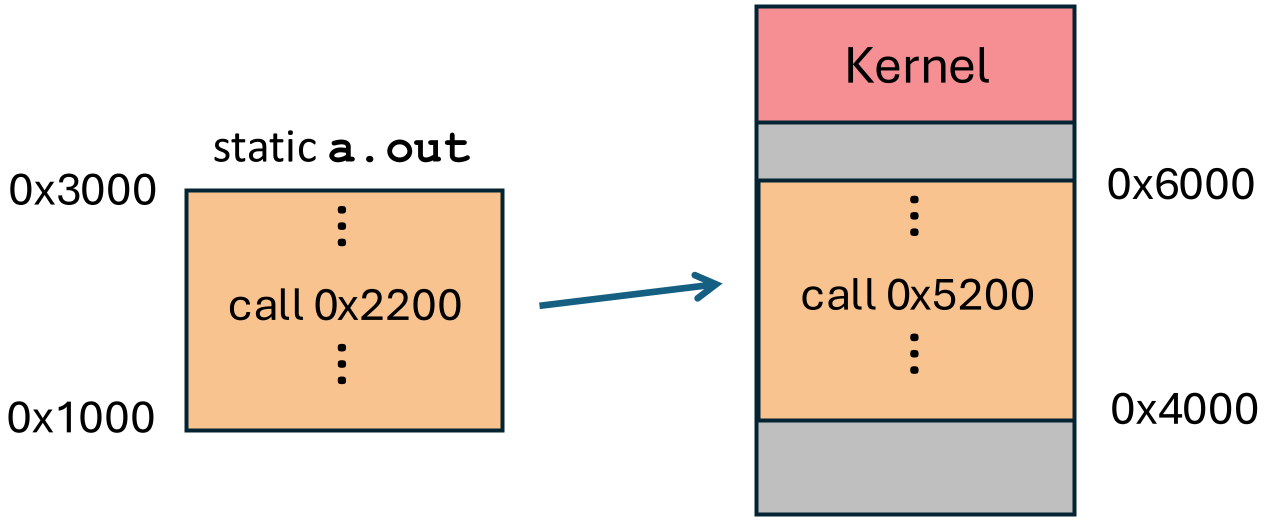
- Run partially in memory, partially on disk

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



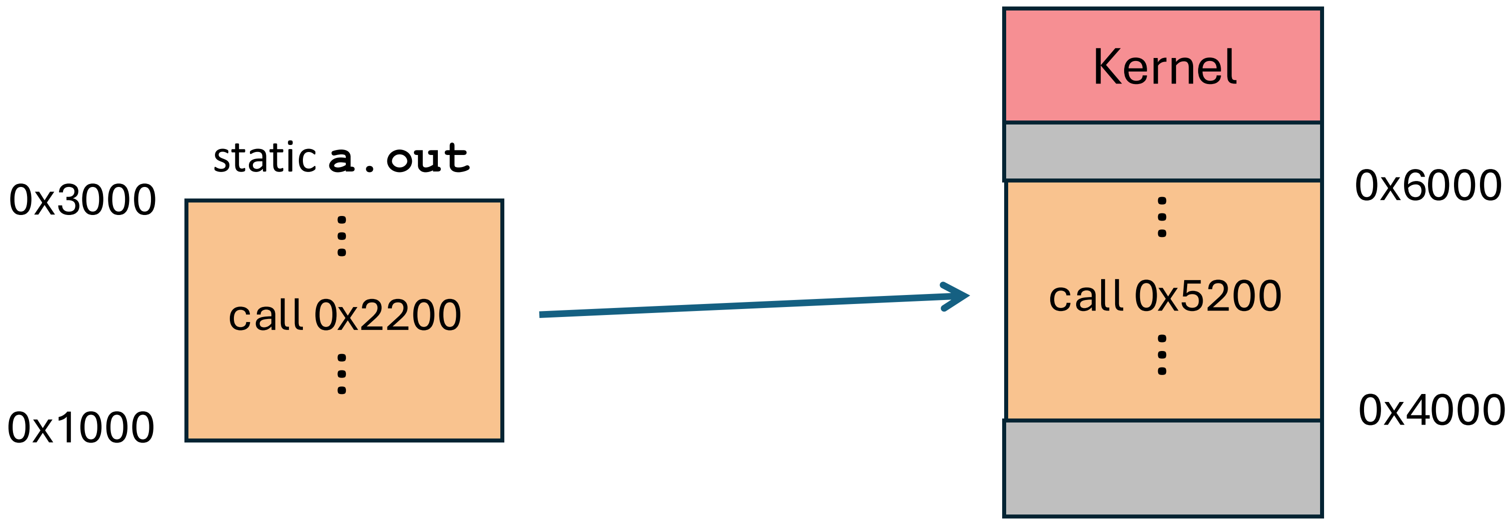
Linker patches long jump addresses (e.g., call printf**)**

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 long jump addresses (e.g., call printf**)**

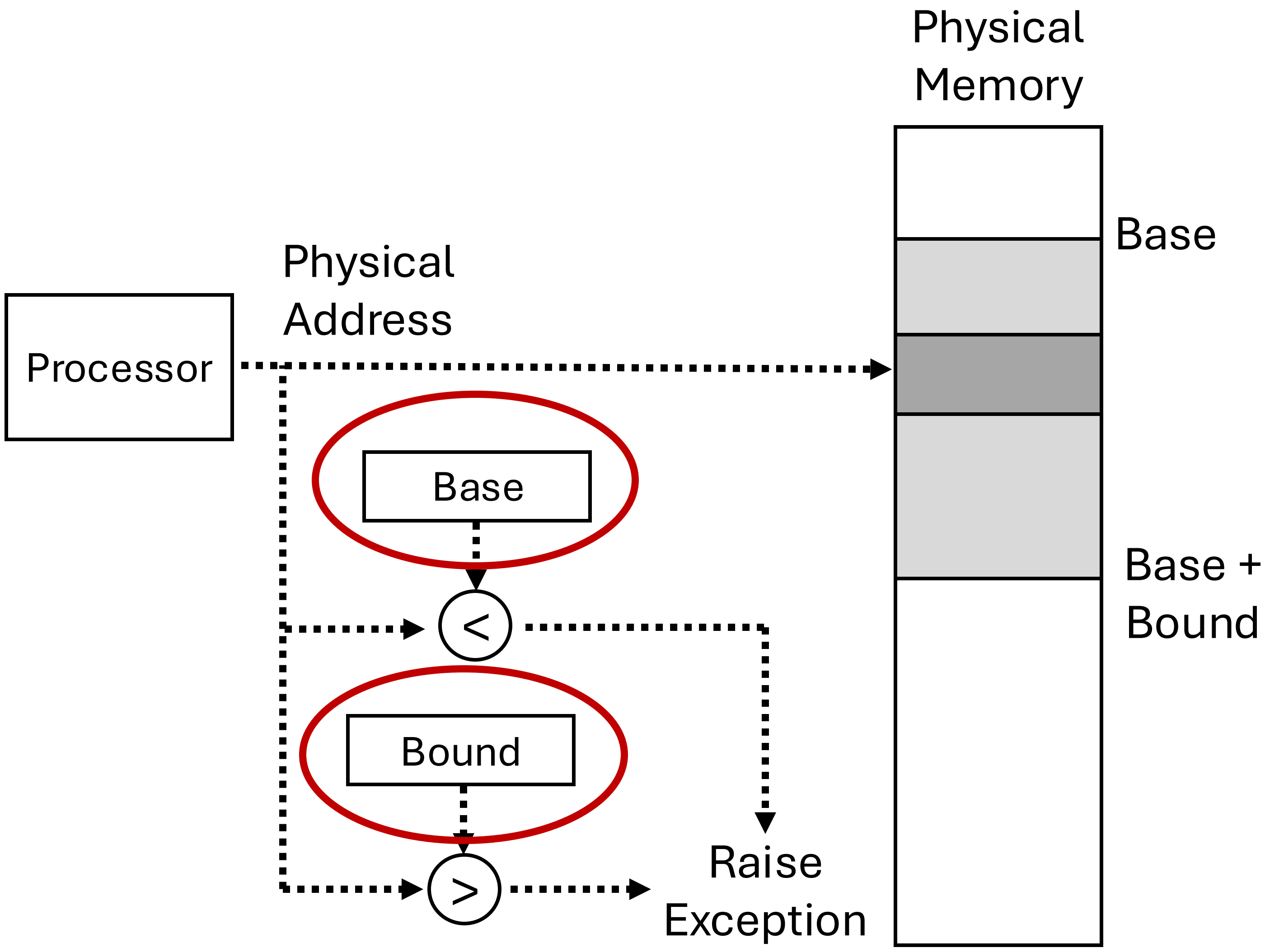
Idea: link when process executed, not at compile time

Problems?

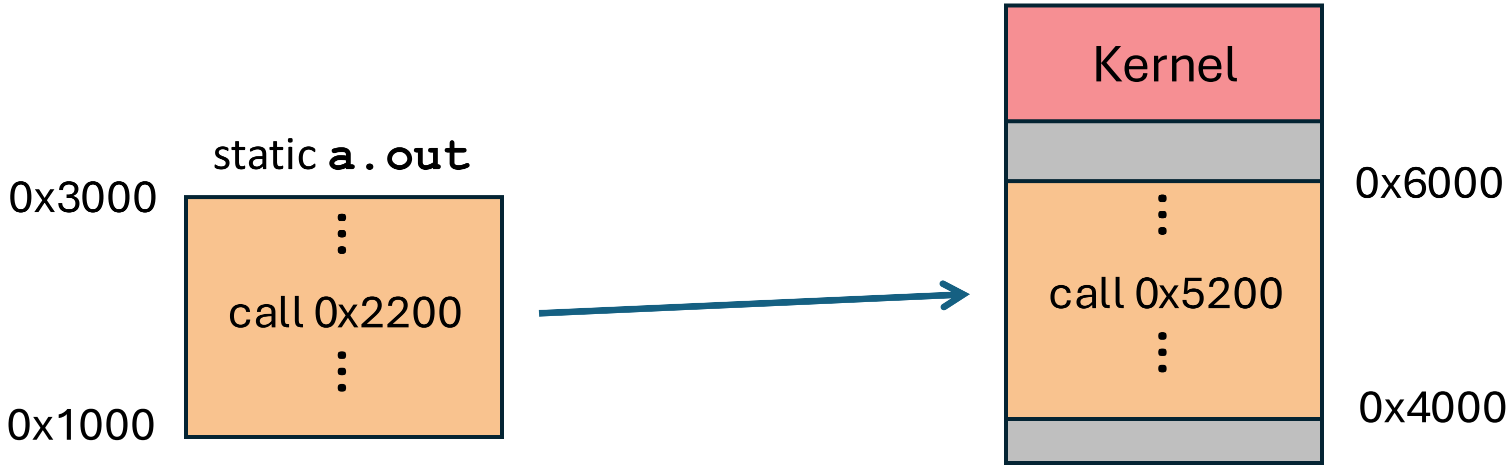
- Patching required for each run, time-consuming
- How to move once already in memory?
- What if no contiguous free region fits program?

Recall: Memory Protection

Memory access bounds check



Idea 2: Base + Bound Register



Two special privileged registers: base and bound

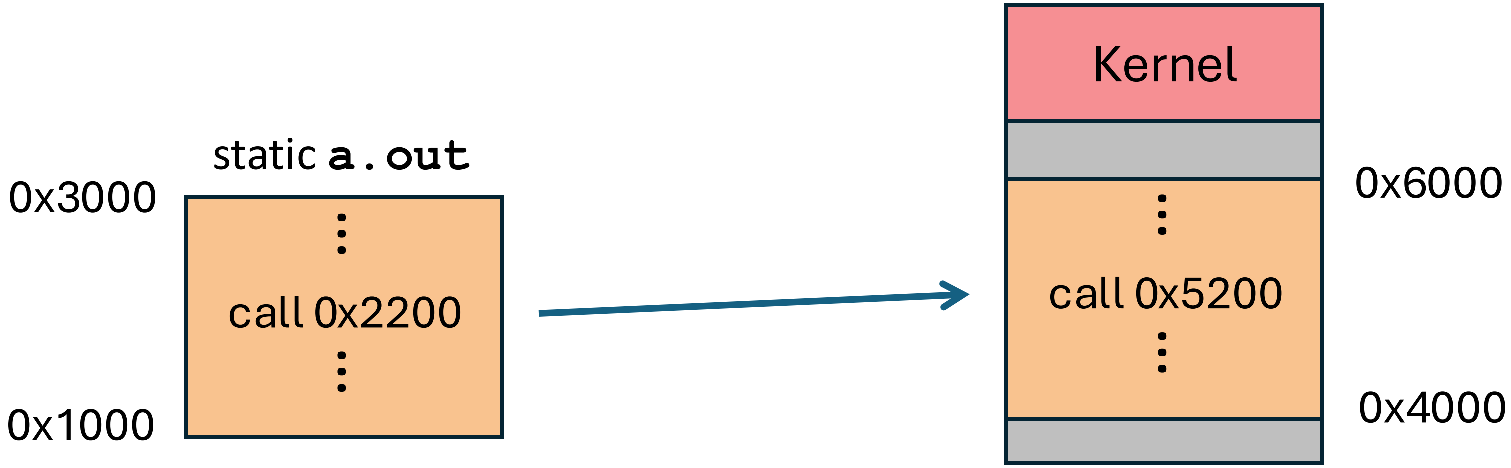
On each load/store/jump:

- 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

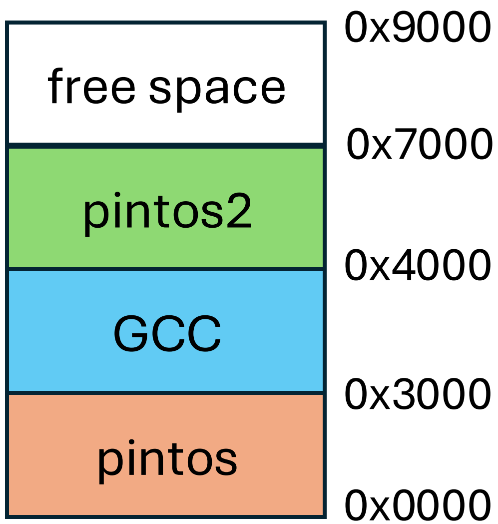
Base + Bound Trade-offs

Advantages

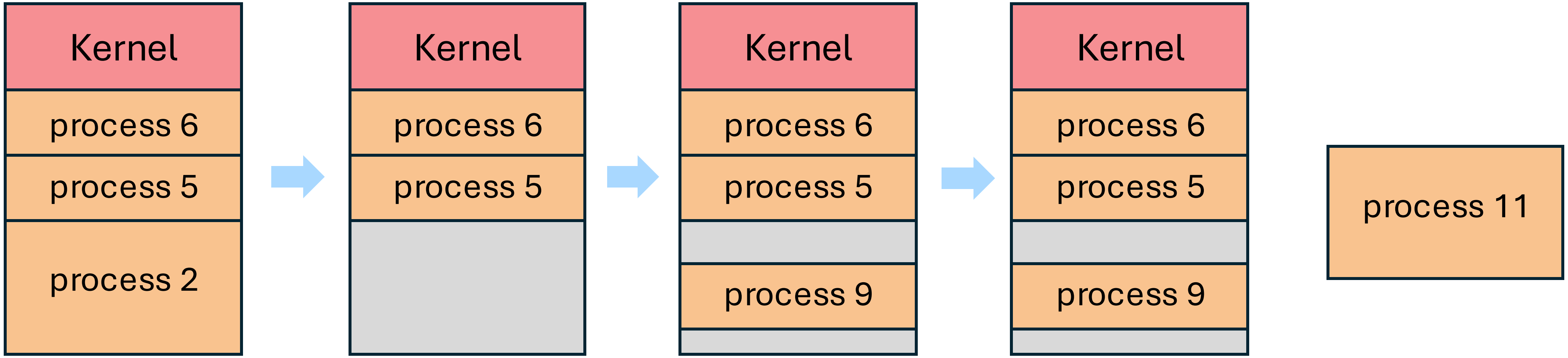
- Cheap in terms of hardware: only two registers
- Cheap in terms of cycles: do add and compare in parallel
- Examples: Cray-1 used this scheme

Disadvantages

- Growing a process is expensive or impossible
- No way to share code or data (E.g., two copies of bochs, both running pintos)



Issues with Simple Base+Bound Method



Fragmentation problem over time

- Not every process is same size → memory becomes fragmented over time

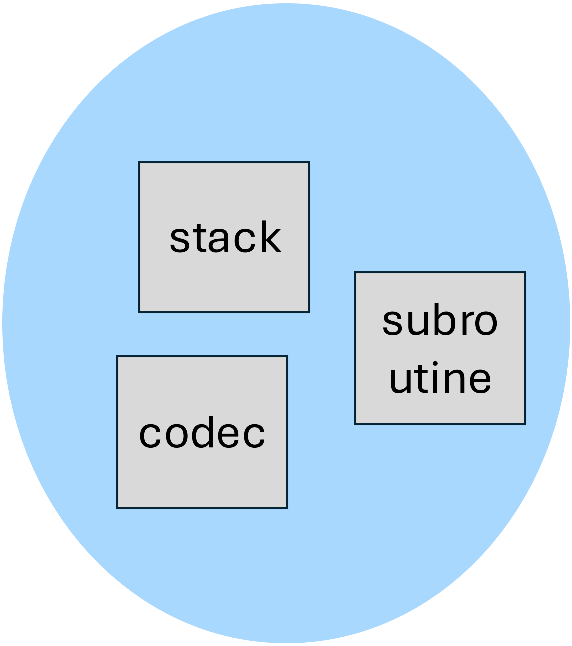
Missing support for sparse address space

- Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)

Hard to do inter-process sharing

- Want to share code segments when possible
- Want to share memory between processes
- Helped by providing multiple segments per process

More Flexible Segmentation



Logical address

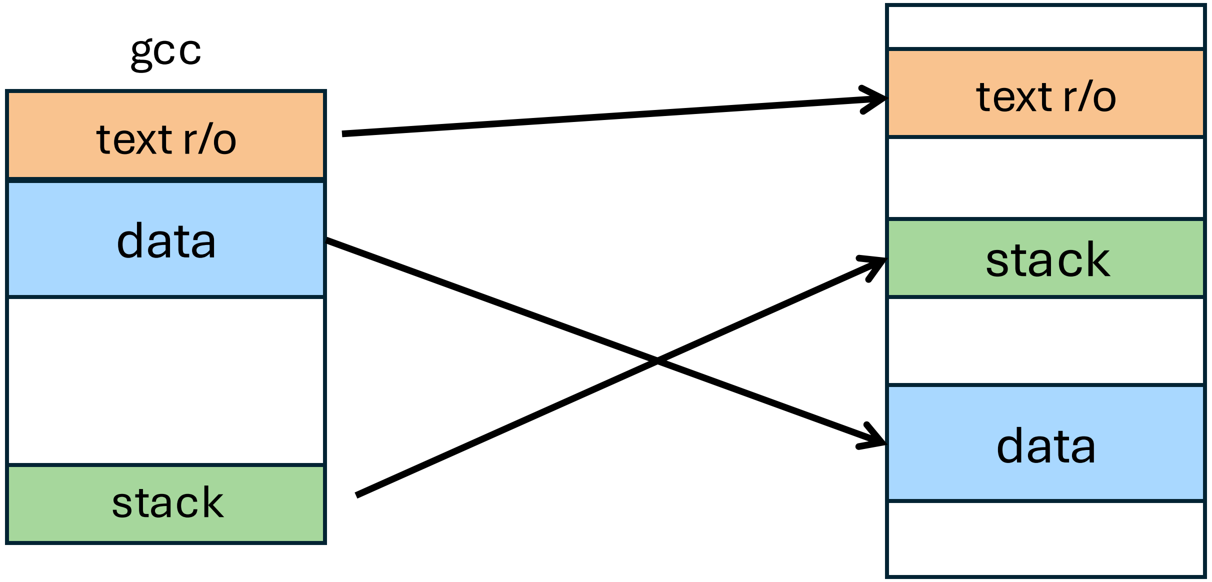
Logical View: multiple separate segments

- Typical: Code, Data, Stack
- Others: memory sharing, etc

Each segment is given region of contiguous memory

- Has a base and limit
- Can reside anywhere in physical memory

Idea3: Segmentation

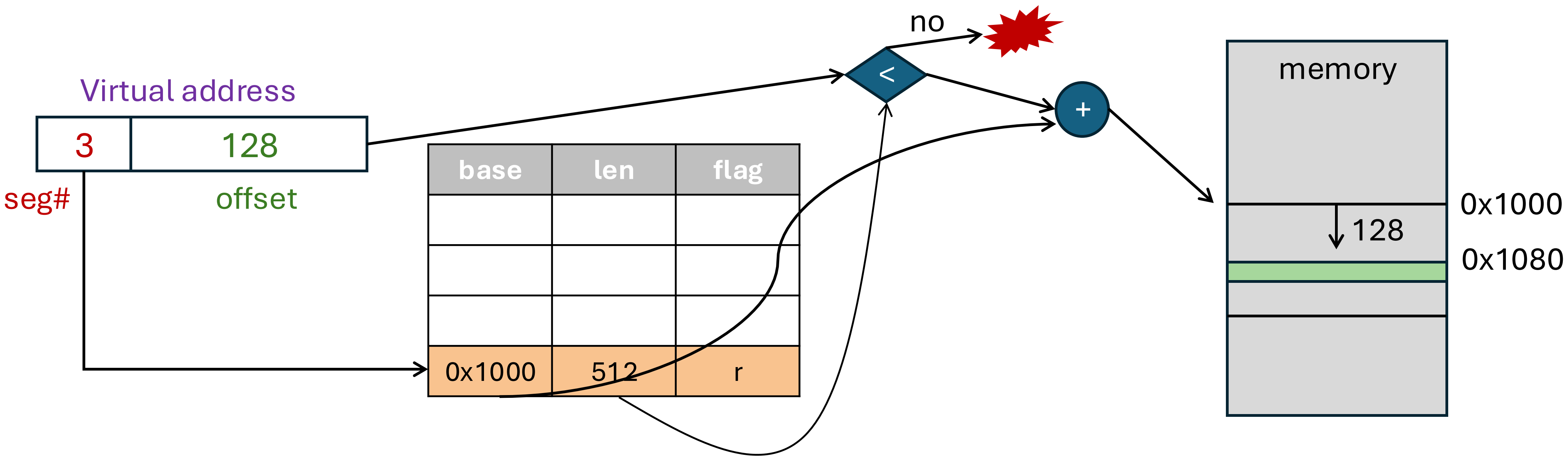


Let processes have many base/bound regs

- 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

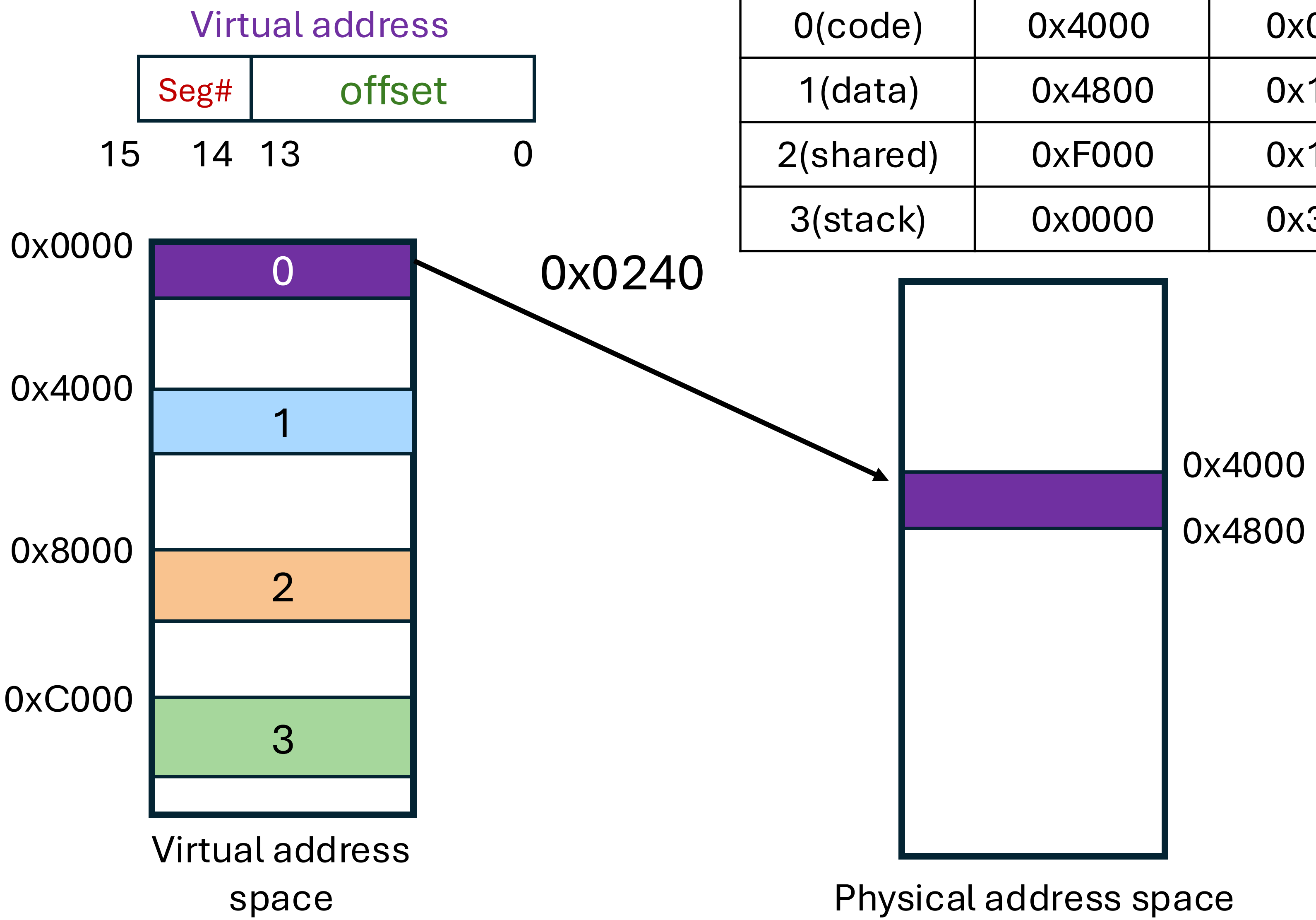
Each virtual address indicates a segment and offset:

- Top bits of addr select segment, low bits select offset
- x86 stores segment #s in registers (CS, DS, SS, ES, FS, GS)

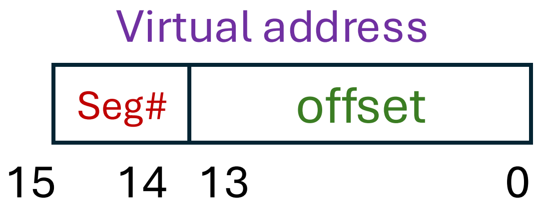
Segmentation Example

Segment	Base	Bound	RW
0(code)	0x4000	0x0800	01
1(data)	0x4800	0x1400	11
2(shared)	0xF000	0x1000	11
3(stack)	0x0000	0x3000	11

Segment table

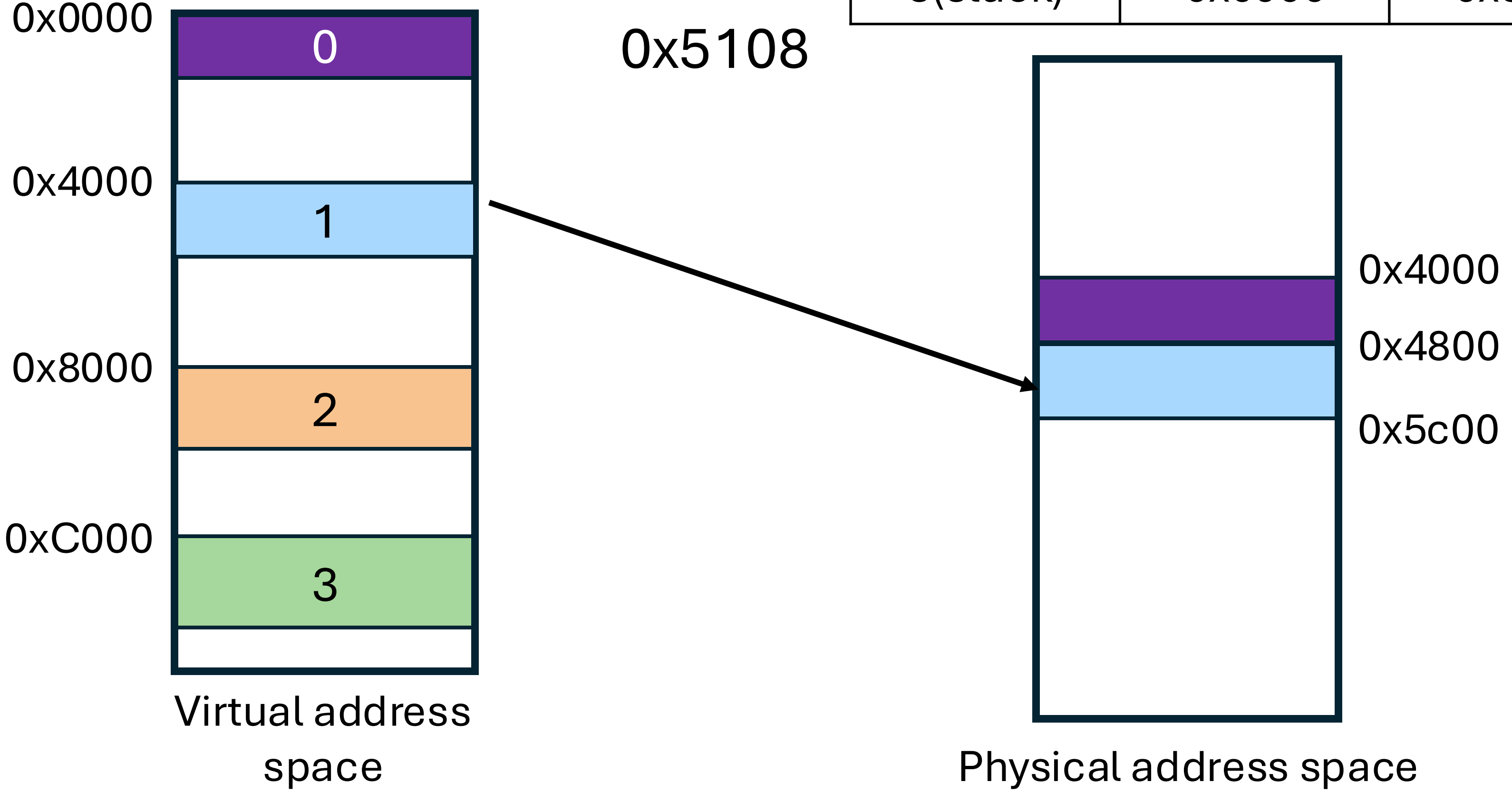


Segmentation Example



Segment	Base	Bound	RW
0(code)	0x4000	0x0800	01
1(data)	0x4800	0x1400	11
2(shared)	0xF000	0x1000	11
3(stack)	0x0000	0x3000	11

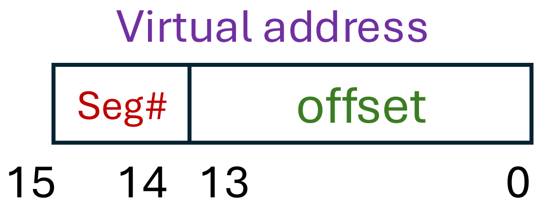
Segment table



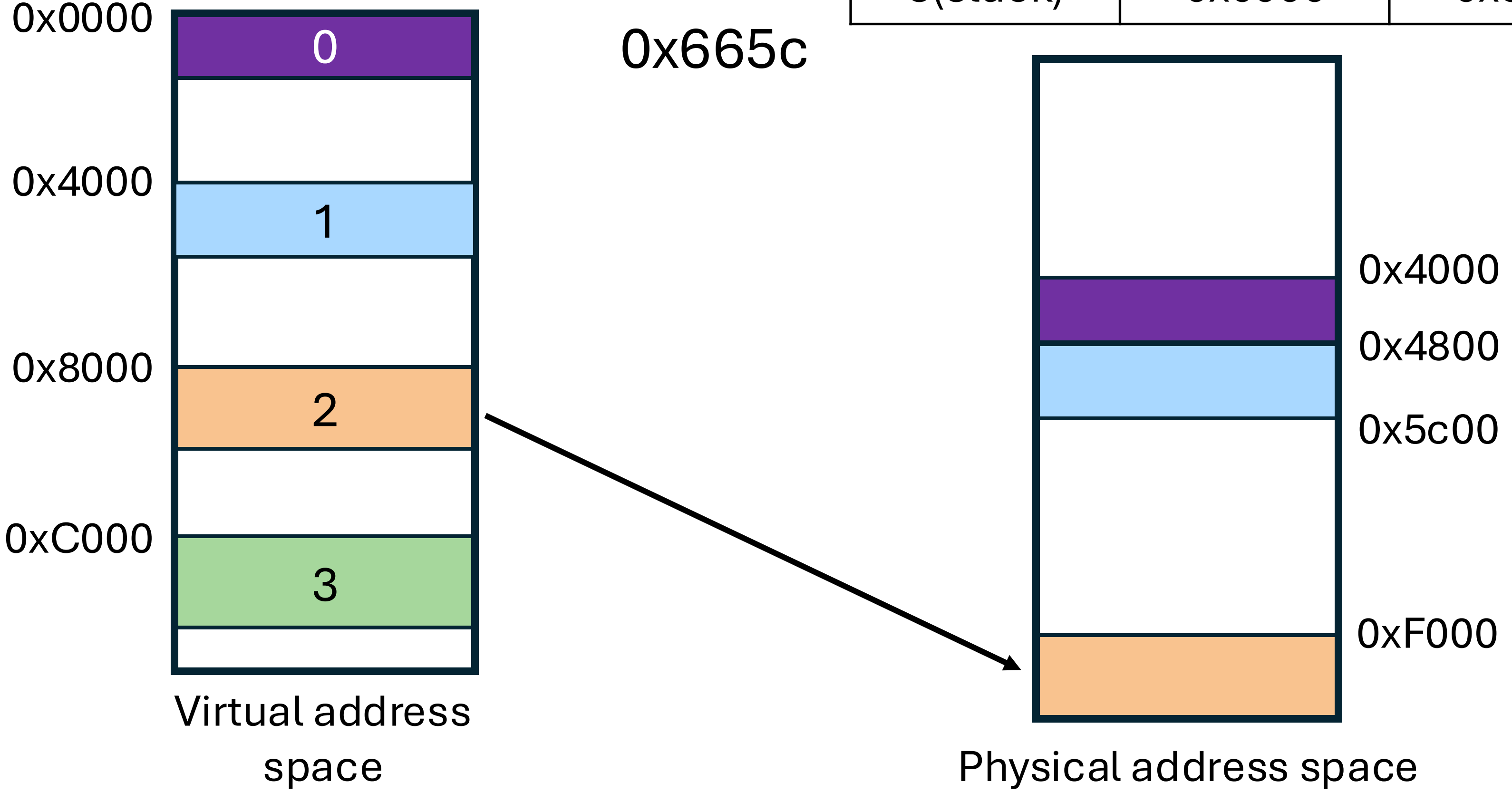
Segmentation Example

Segment	Base	Bound	RW
0(code)	0x4000	0x0800	01
1(data)	0x4800	0x1400	11
2(shared)	0xF000	0x1000	11
3(stack)	0x0000	0x3000	11

Segment table



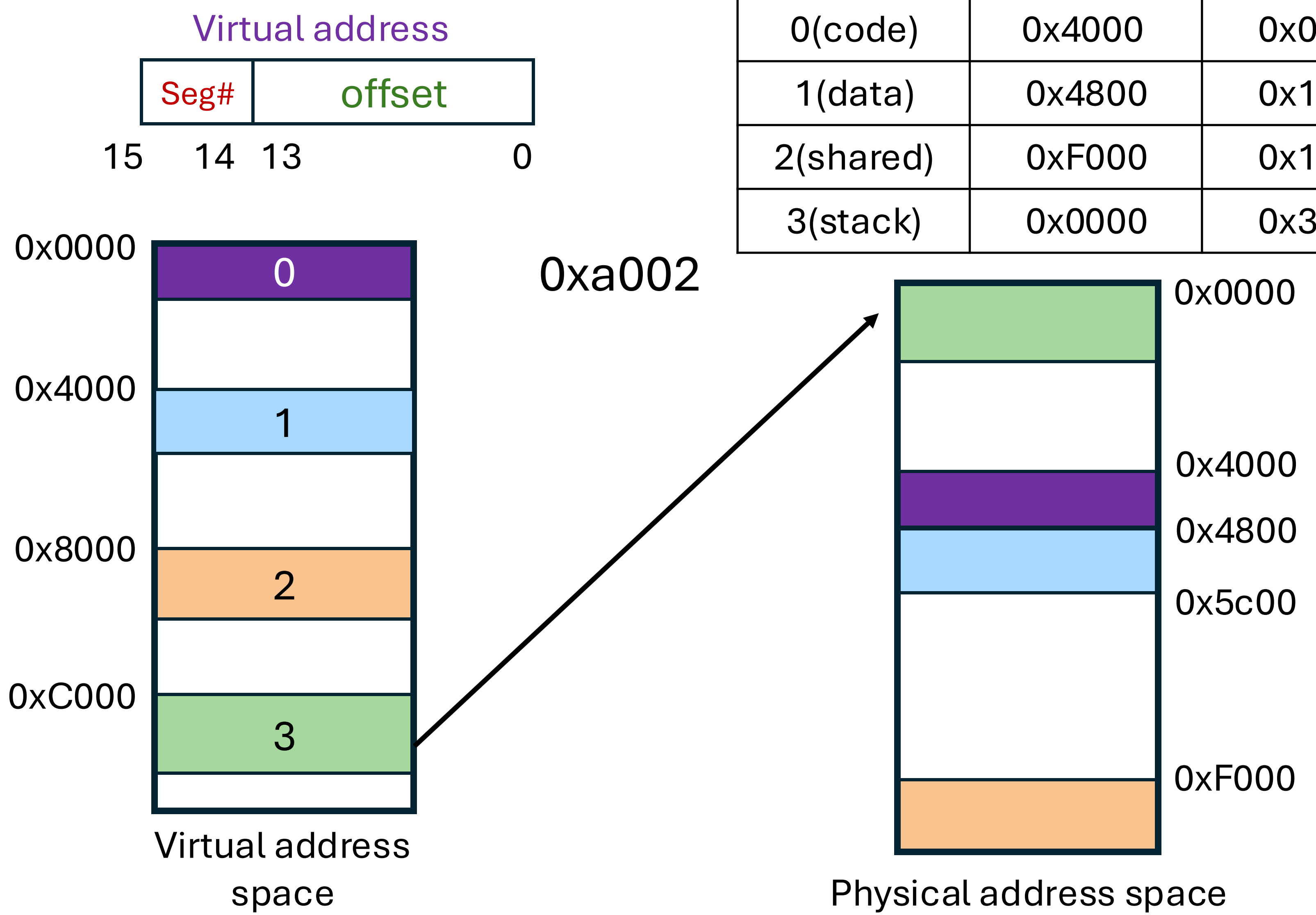
0x665c



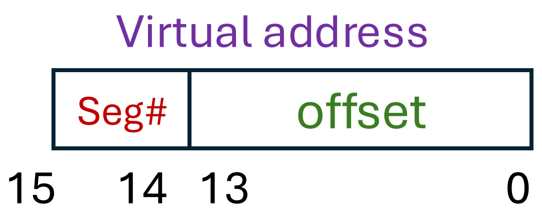
Segmentation Example

Segment	Base	Bound	RW
0(code)	0x4000	0x0800	01
1(data)	0x4800	0x1400	11
2(shared)	0xF000	0x1000	11
3(stack)	0x0000	0x3000	11

Segment table



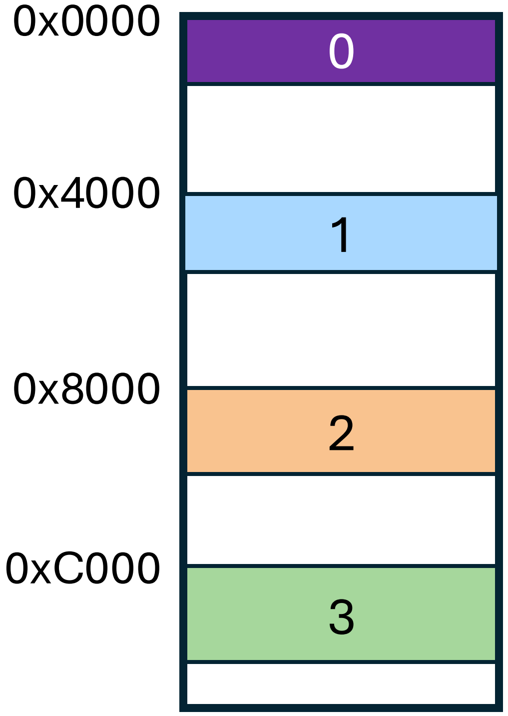
Segmentation Example



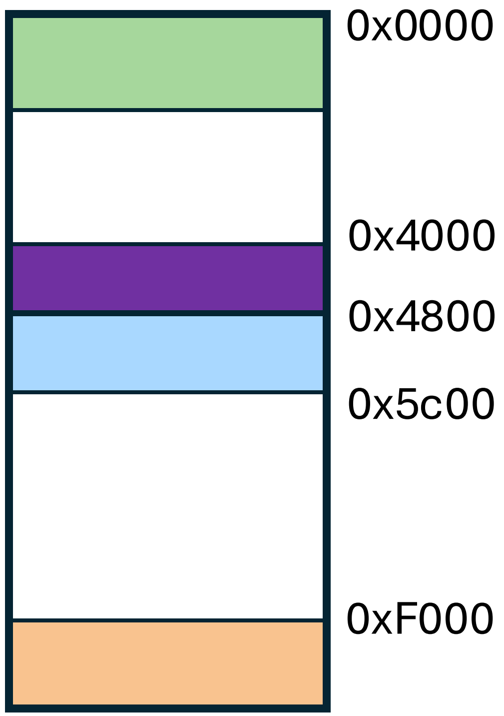
Segment	Base	Bound	RW
0(code)	0x4000	0x0800	01
1(data)	0x4800	0x1400	11
2(shared)	0xF000	0x1000	11
3(stack)	0x0000	0x3000	11

Segment table

0x1600



Virtual address space



Physical address space

Segmentation Trade-offs

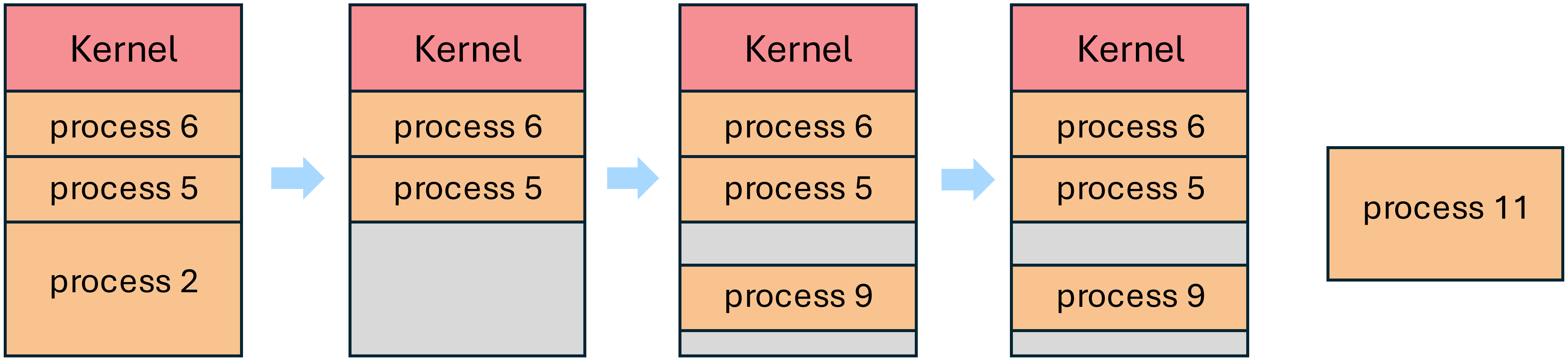
Advantages

- 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.

Recap: Fragmentation



Over time:

- many small holes (external fragmentation)
- no external holes, but force internal waste (internal fragmentation)

Idea 4: Paging

Divide memory up into fixed-size *pages*

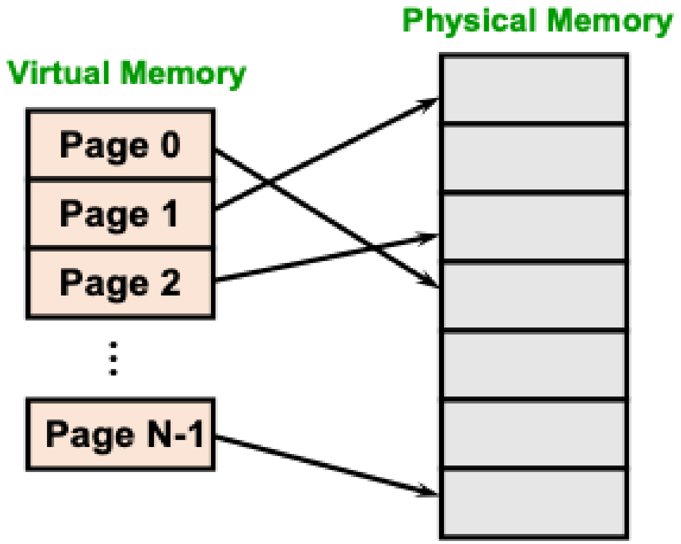
- Typical size: 4k-8k
- Eliminates external fragmentation

Map virtual pages to physical pages

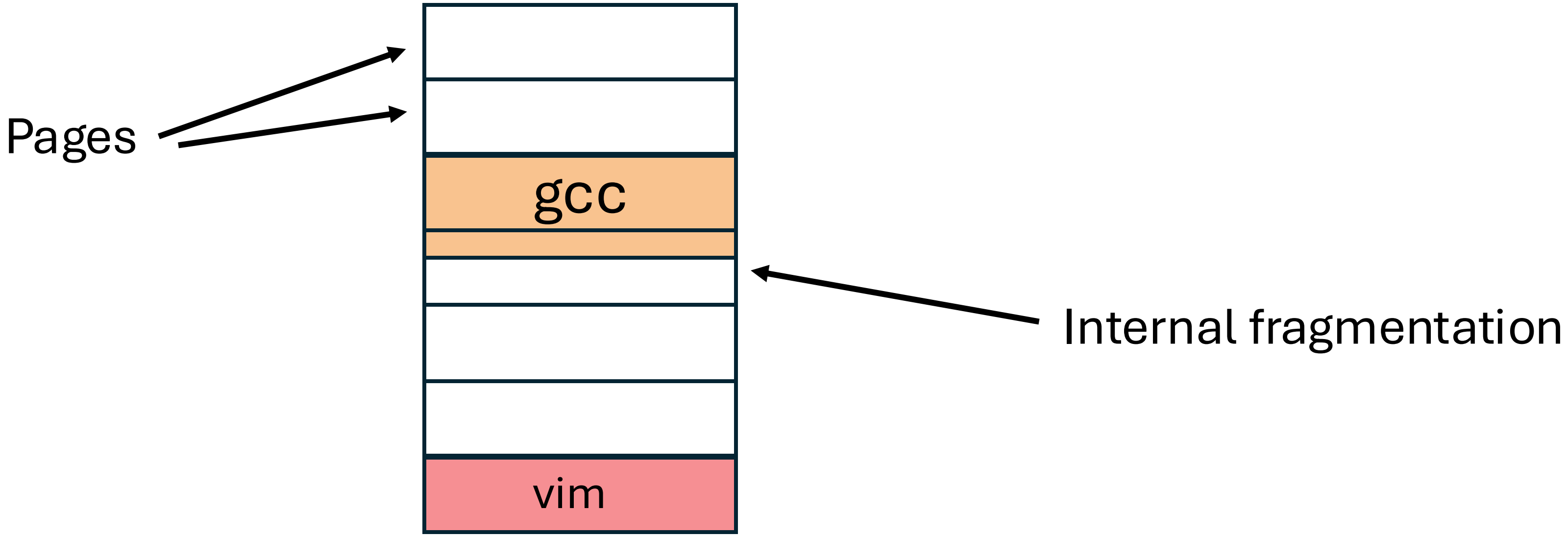
- Each process has separate mapping

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 with No External Fragmentation

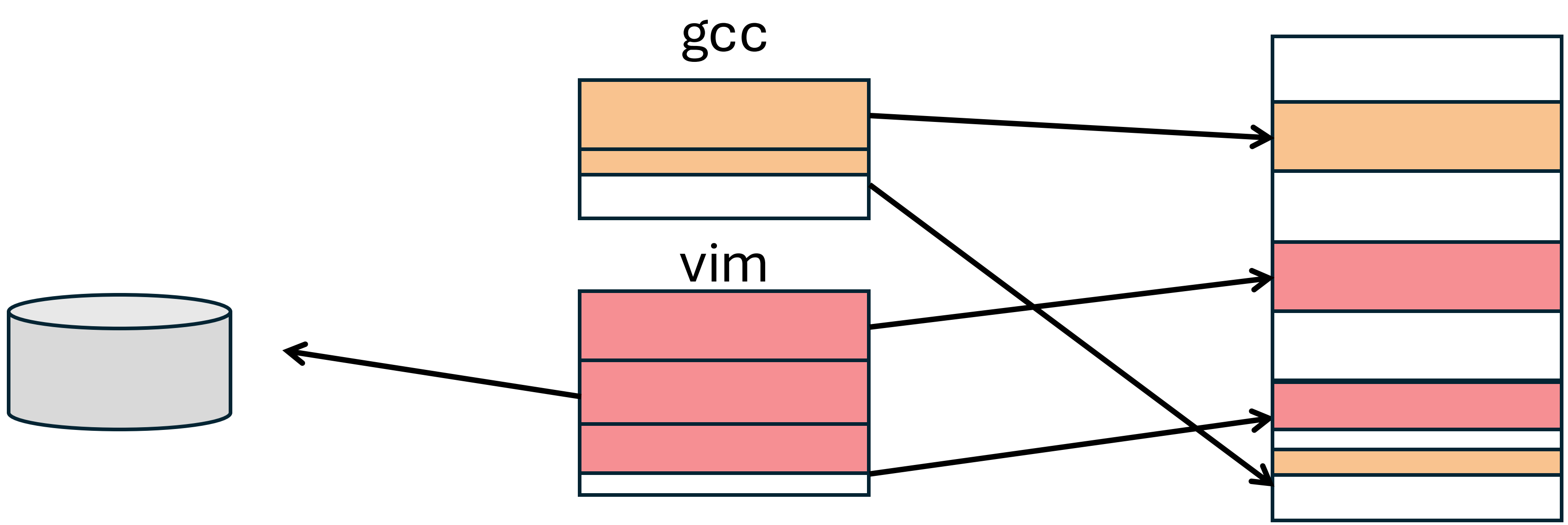


Eliminates external fragmentation

Average internal fragmentation of .5 pages per “segment”

Simplifies allocation, free, and backing storage (swap)

Simplified Allocation



Allocate any physical page to any process

Can store idle virtual pages on disk

Page and Page Tables

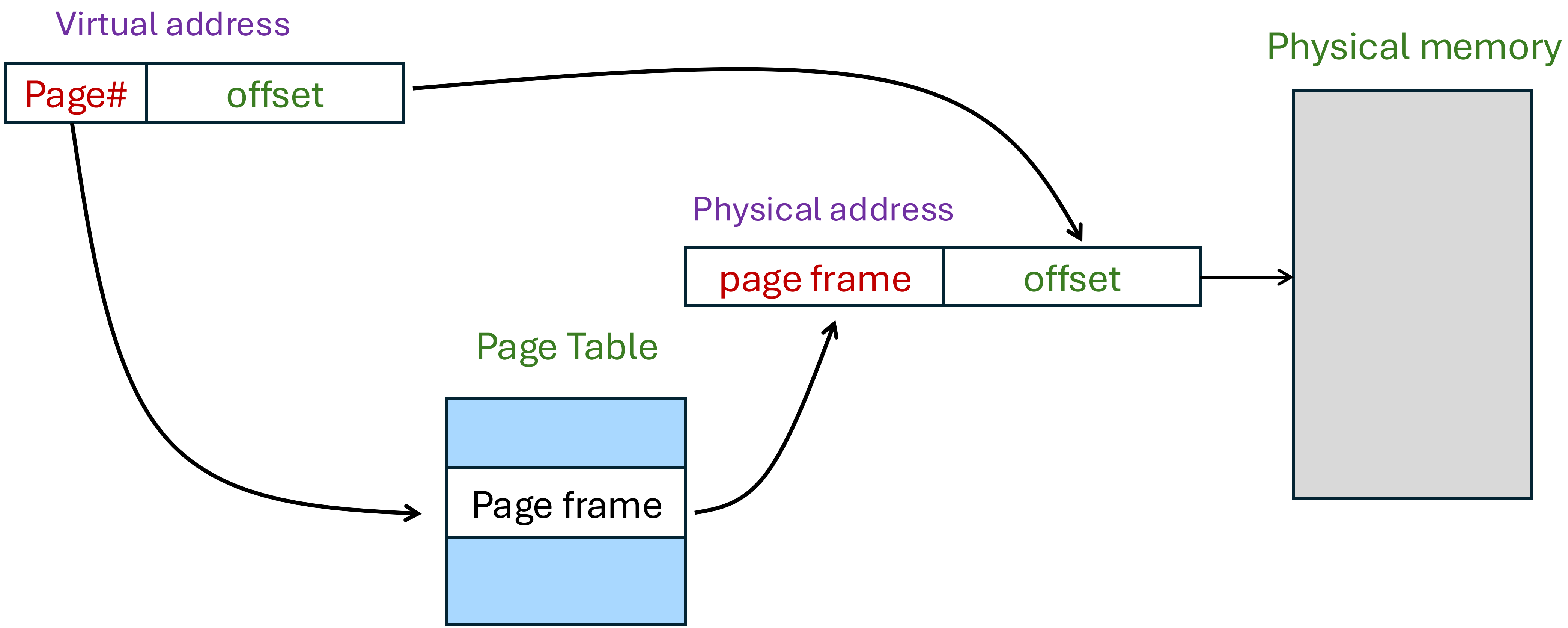
Pages are fixed size, e.g., 4K

- 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 Lookups



Page Table Entries (PTEs)

Physical Page Number	M	R	V	Protection

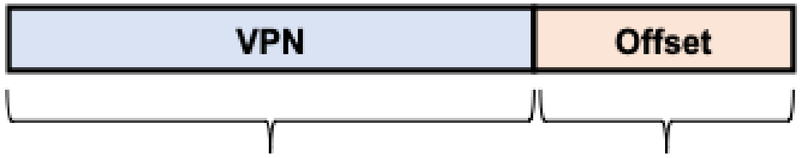
Page table entries control mapping

- The Physical page number (PPN) determines physical page
- 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

Paging Example

32-bit machines, pages are 4KB-sized

Virtual Address



What is the maximum number of VPNs?

Virtual address is `0x7468`

`0x7468`

Page Table

VPN	Prot	...
0x2	r	

Physical Address

Page 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

Page Limitation

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 – multi-level page tables (more later)

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

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

Typically all kernel text, most data at same virtual address 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?s

Summary

Virtual memory

- Processes use virtual addresses
- OS + hardware translates virtual address into physical addresses

Various techniques

- Load-time Linking – requires patching for each run
- Base + Bounds – cheap, but difficult to grow and cannot share
- Segmentation – manage in chunks from user's perspective
- Paging – use small, fixed size chunks, efficient for OS
- Combine paging and segmentation

Next Time…

Chapters 19, 20