Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 12: Address Translation

Prof. Ryan Huang

Administration

Course web server was flaky due to some repos being too large

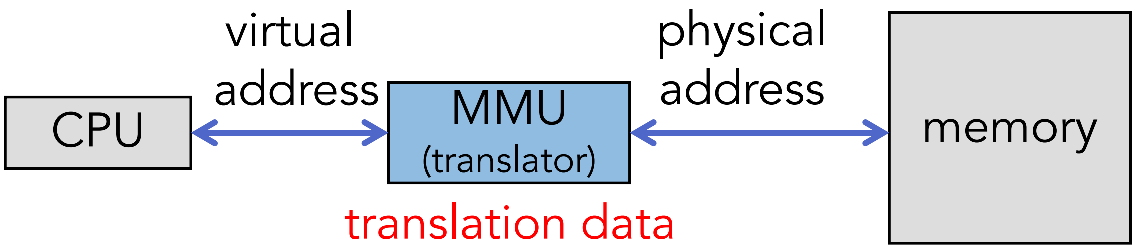
- These repos contain object files, etc.

Thanks to Prof. Chen, the problem is fixed

Follow git best practices!

- Don't commit the *results* of compilation (object files, executable, etc.) into git history
- Use .gitignore to ignore such files

Recap: address translation



MMU uses translation data

- Each process has its own translation data
- Changing address spaces → changing translation data

Many ways (data structures) to implement translator

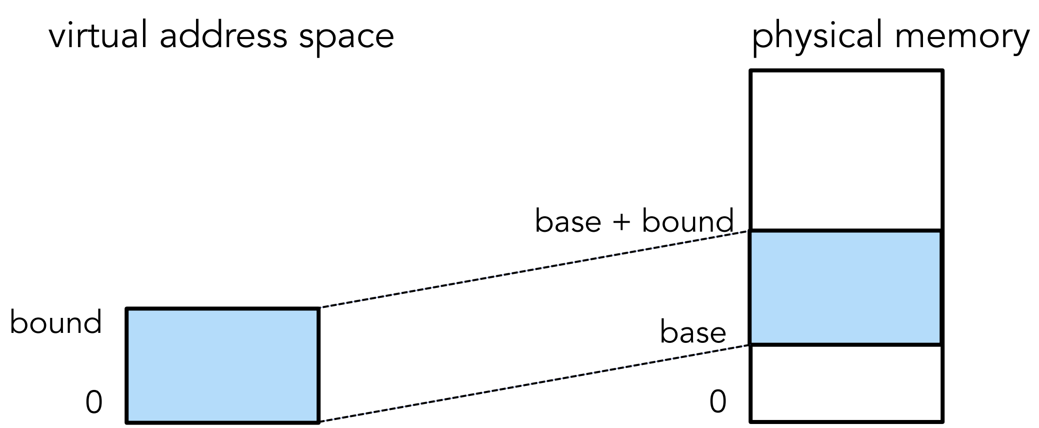
- Speed of translation
- Size of data needed to support translation
- Flexibility (sharing, growth, large address space)

Idea 1: Base + bound

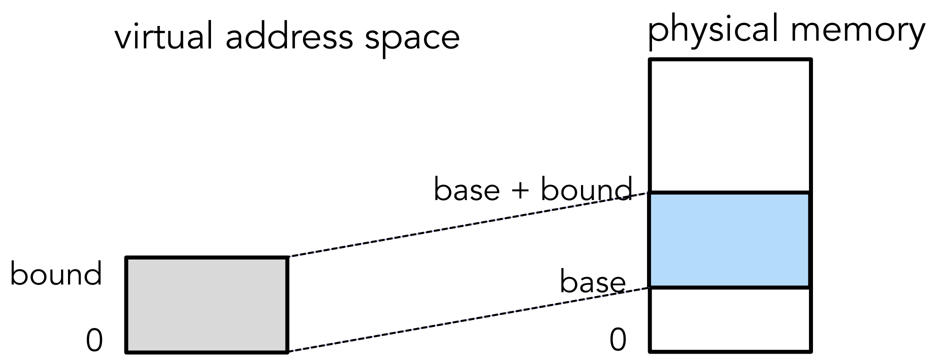
Load each process into a single contiguous region of physical memory

Two special privileged registers:

- Base register: starting physical address
- Bound register: size of region



Translation algorithm



On each load/store/jump:

```
if (virtual address >= 0 && virtual address < bound) {
    physical address = virtual address + base
} else {
    trap to kernel; OS kills process or tries to fix fault & retry access
}
```

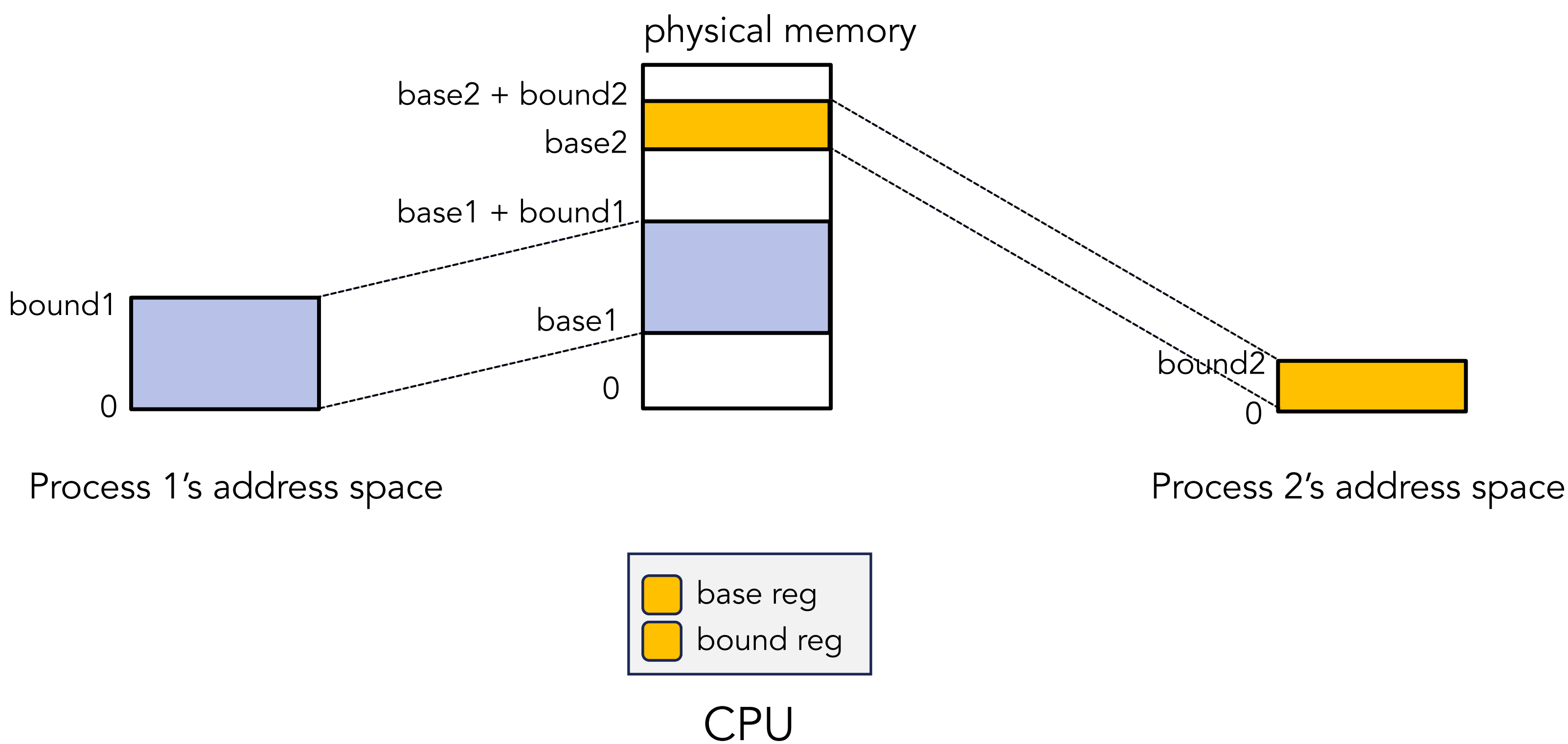
How to move a process in memory?

- Change the base register

How to switch address spaces?

- Reload the base and bound registers

Switching address spaces



Base + bound trade-offs

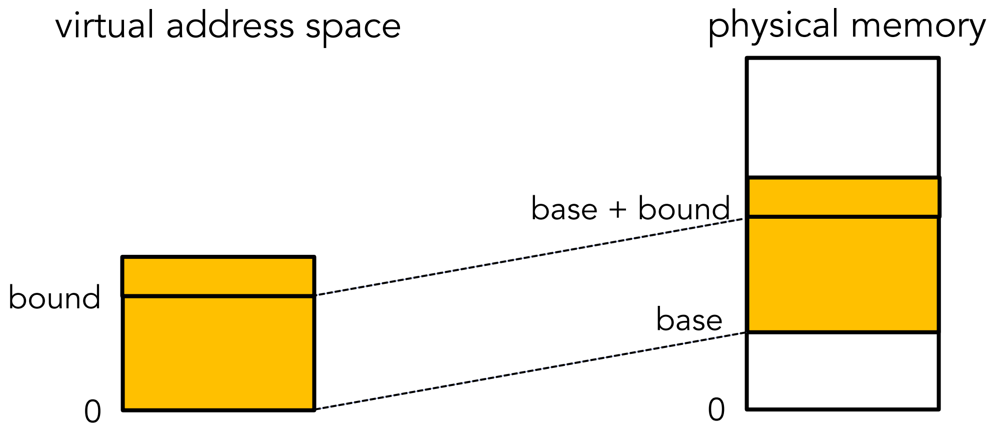
Advantages:

- 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

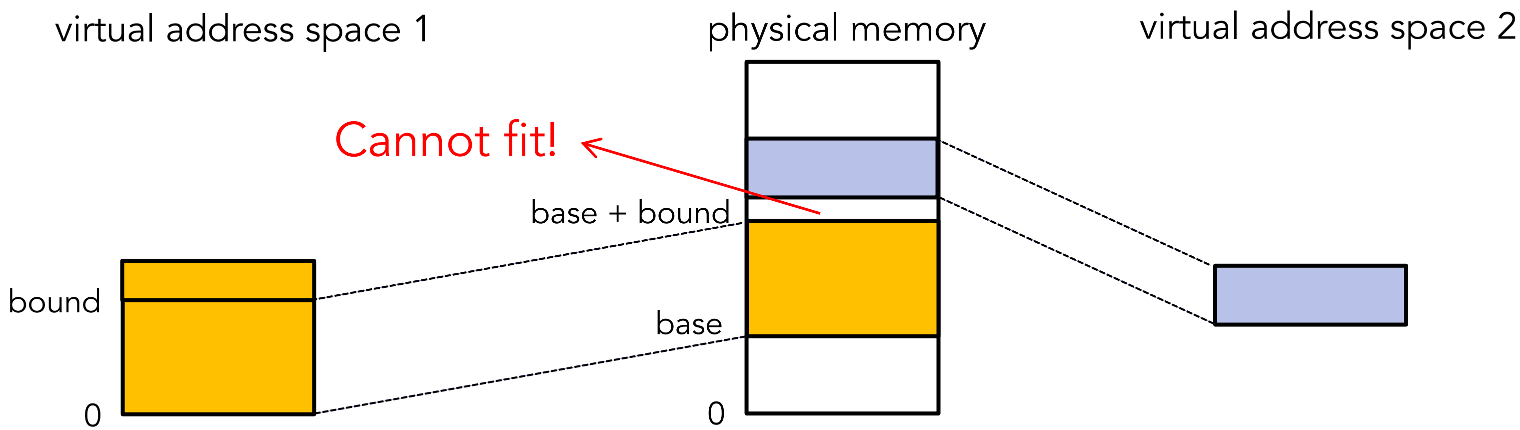
Disadvantages?

- Address independence?
- Protection?
- Large address space?
- Flexibility?

How to grow an address space?

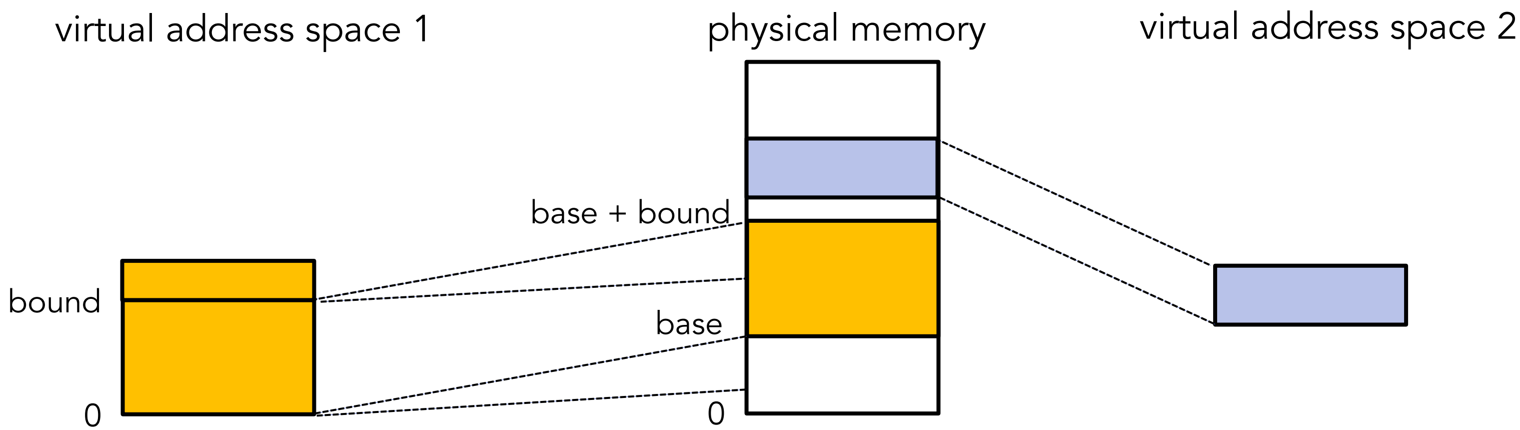


How to grow an address space?



What to do?

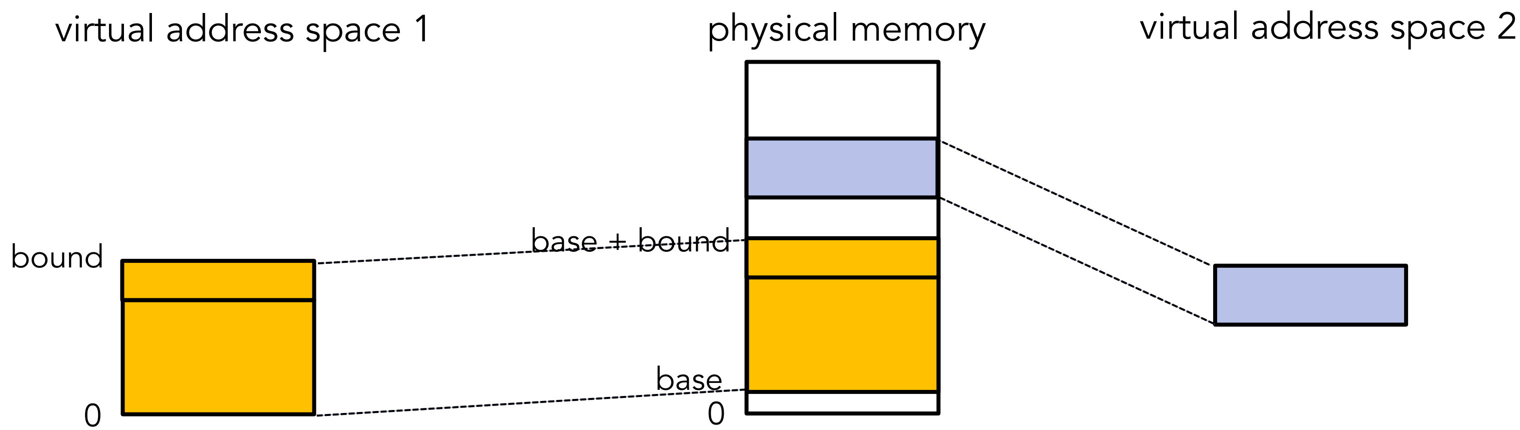
How to grow an address space?



What to do?

Move around, then grow

How to grow an address space?



Is the process aware of this move?

- No: Transparent
- Yes: Slow! (mem copy)

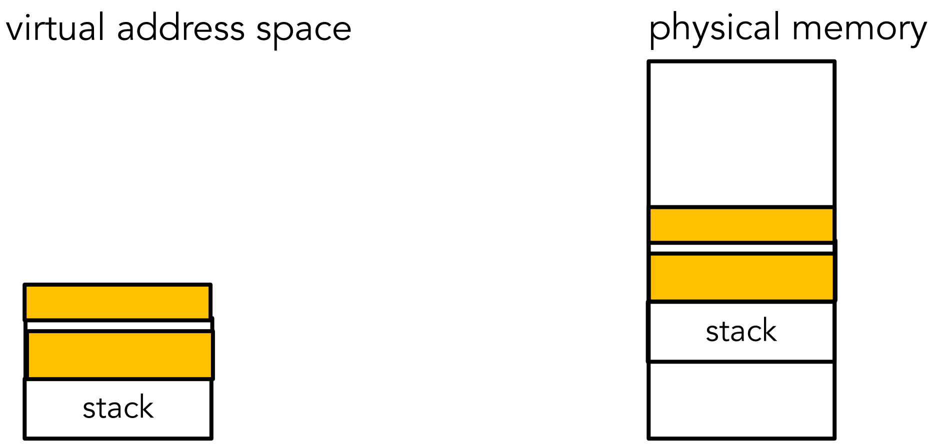
May not always be possible to move

How to grow multiple regions in an address space?

Easier to grow top region (e.g., heap)

How to grow lower region (e.g., stack)?

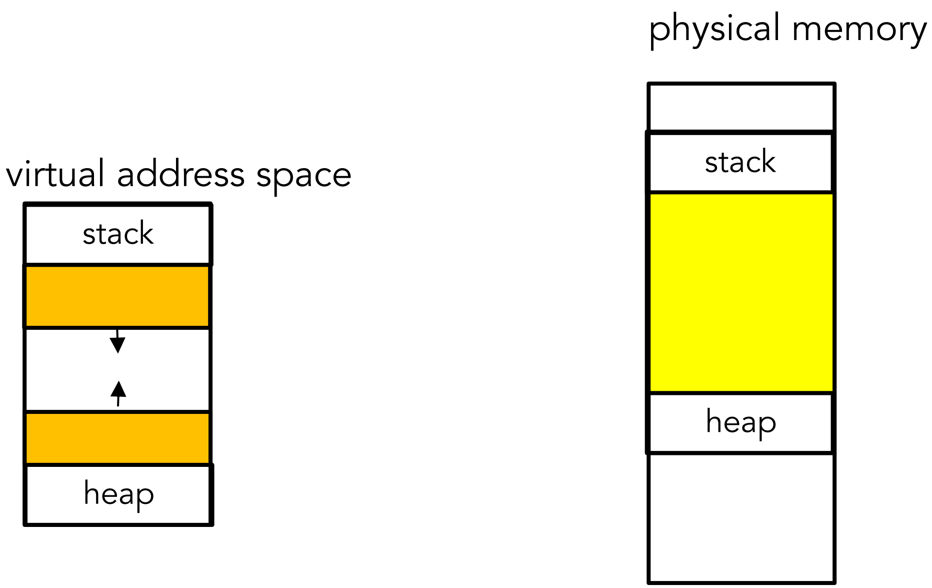
- Problem?



How to grow multiple regions in an address space?

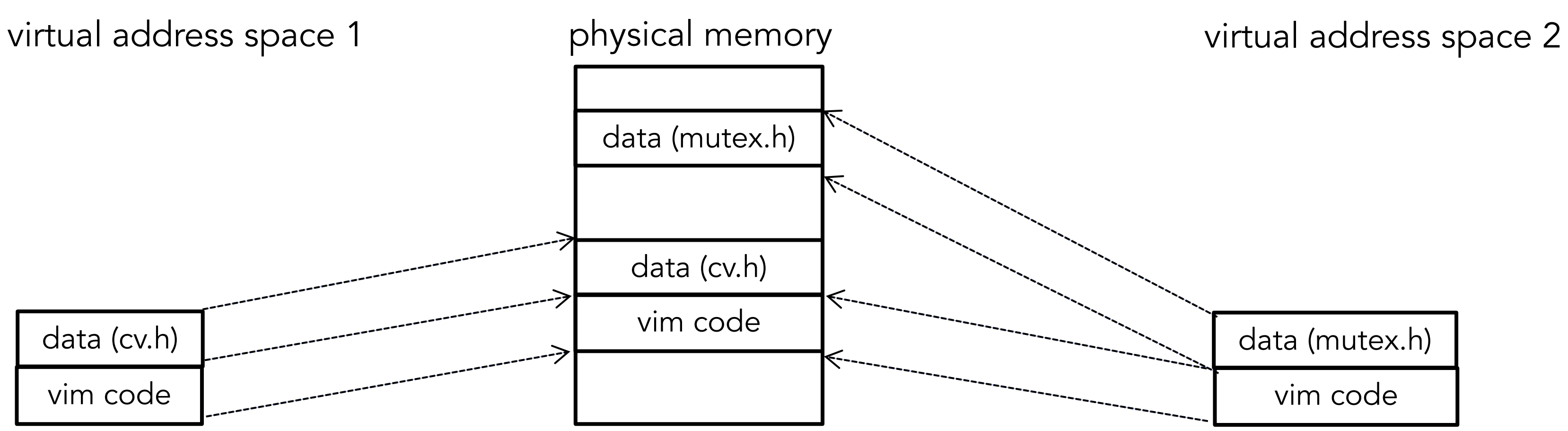
Must reserve space for each region in virtual address space

Problem?



Base and bound: sharing

Can't share part of an address space between processes



Base and bound

Disadvantages?

- Does not support large address spaces
- Growing address space may be slow or impossible
- Need to reserve space to allow multiple regions to grow
- Can’t share part of address space

Root cause: requires entire address space to be contiguous in memory

Idea 2: Segmentation

Let a process have multiple base/bound regs

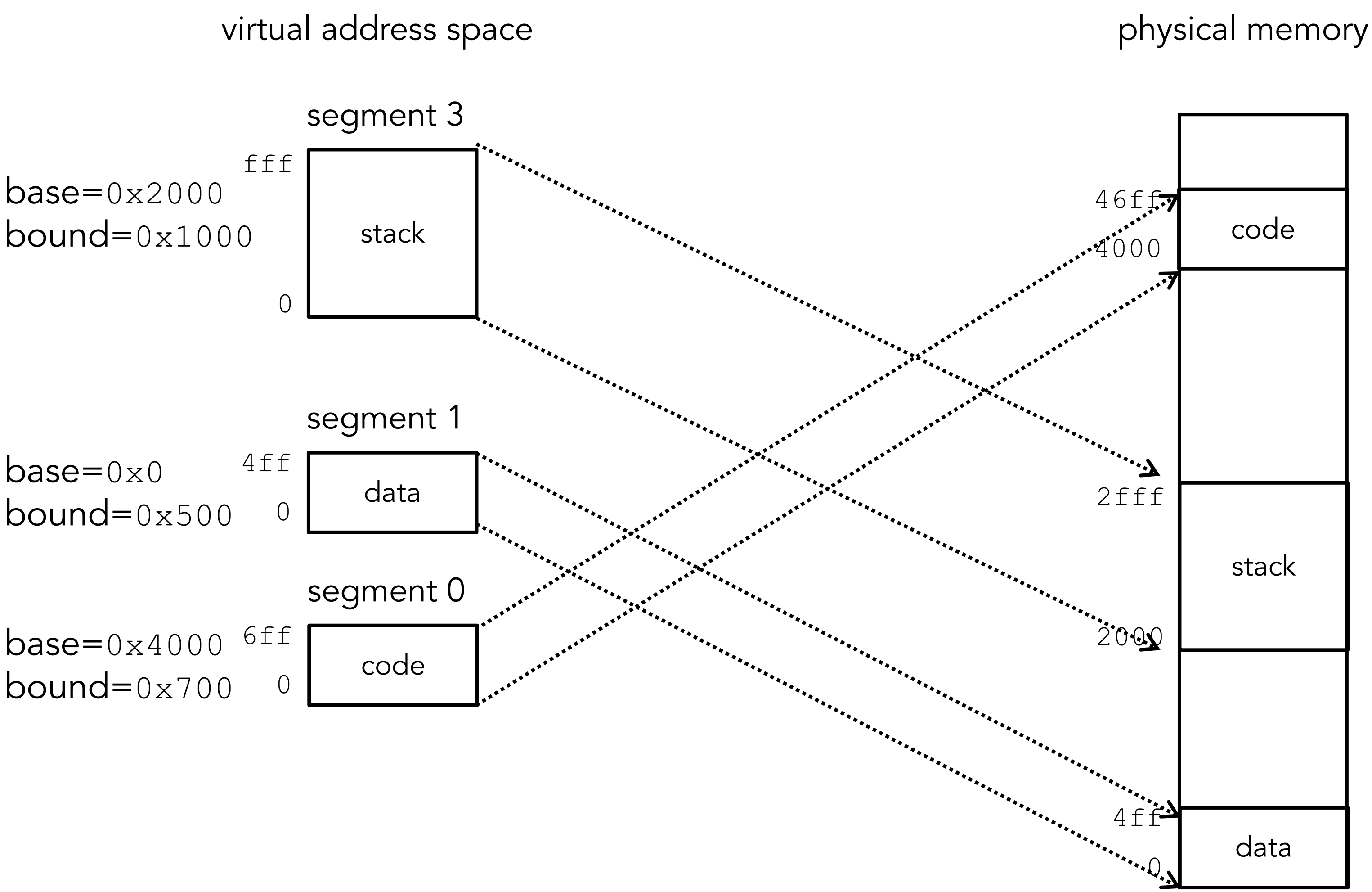
Divide address space into multiple segments

- Can share/protect memory at segment granularity

Segment: region of address space that is:

- Contiguous in physical memory
- Contiguous in virtual address space
- Of variable size

Segmentation



Segmentation

Segment #	Base	Bound	
0	0x4000	0x700	code segment
1	0	0x500	data segment
2	n/a	0	unused
3	0x2000	0x1000	stack segment

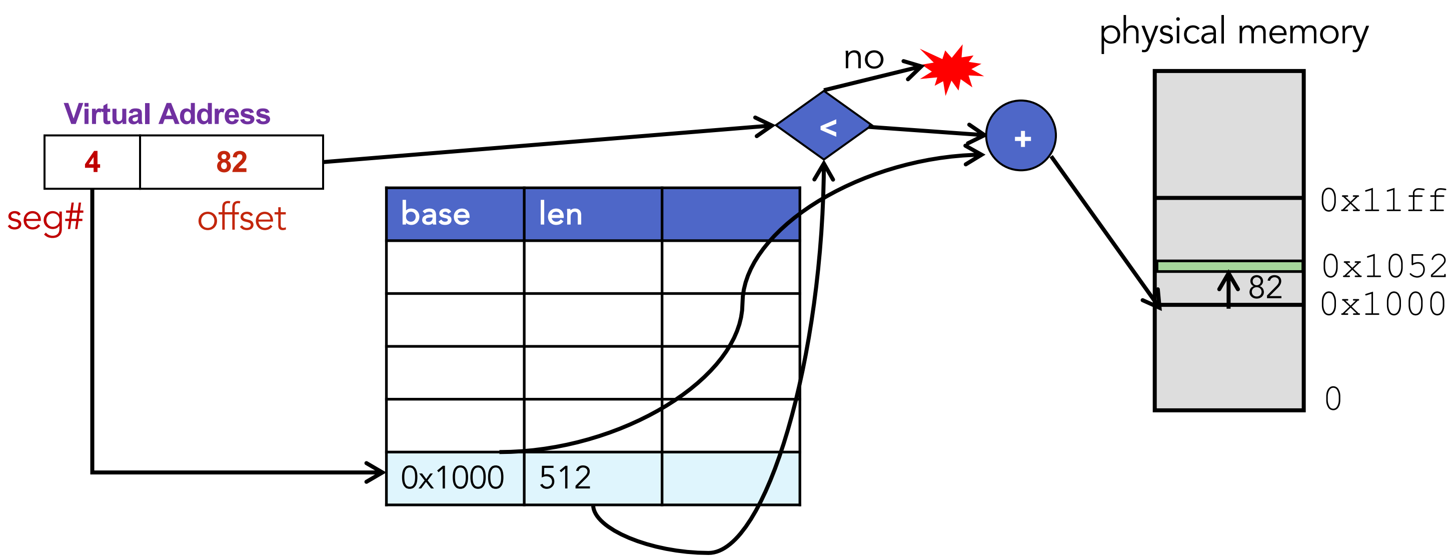
Sets up a segment table for each process

- Each row describes one segment (segment descriptor)

Virtual address → (segment #, offset)

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

Segmentation: translation



1. Break down a virtual address into (seg #, offset)

2. Physical address = seg_table[segment #].base + offset

Translation example

Segment #	Base	Bound
0	`0x4000`	`0x700`
1	`0`	`0x500`
2	`n/a`	`0`
3	`0x2000`	`0x1000`

Assume max 16 segments

- With 20-bit addresses: 4 bits for seg #, 16 bits for offset

Virtual address —(seg #, offset)→ **Physical address**

`0x30100` → `(3,100)` → `0x2000+0x100`?= `0x2100`

`0xff` → `(0,ff)` → `0x4000+0xff = 0x40ff`

`0x200ff` → `(2,ff)` illegal! (segmentation fault)

`0x105ff` → `(1,5ff)` illegal! (segmentation fault)

Valid vs. invalid addresses

Not all virtual addresses are valid

- Valid → address is part of virtual address space
- Invalid → virtual address is illegal to access
  - Accessing invalid address causes trap to OS

In segmentation, a virtual address can be invalid due to:

- Invalid segment number
- Out of bound offset

Segmentation: protection

Segment #	Base	Bound	Flag
0	`0x4000`	`0x700`	`r/o`
1	`0`	`0x500`	`r/w`
2	`n/a`	`0`	
3	`0x2000`	`0x1000`	`r/w`

Can protect at segment granularity

- Segment descriptor contains protection info (access rights)

Check access rights for each memory access

- Segmentation faults can occur for valid virtual address
  - E.g., write to `0x100`