



Memory

CS 241

February 1, 2012

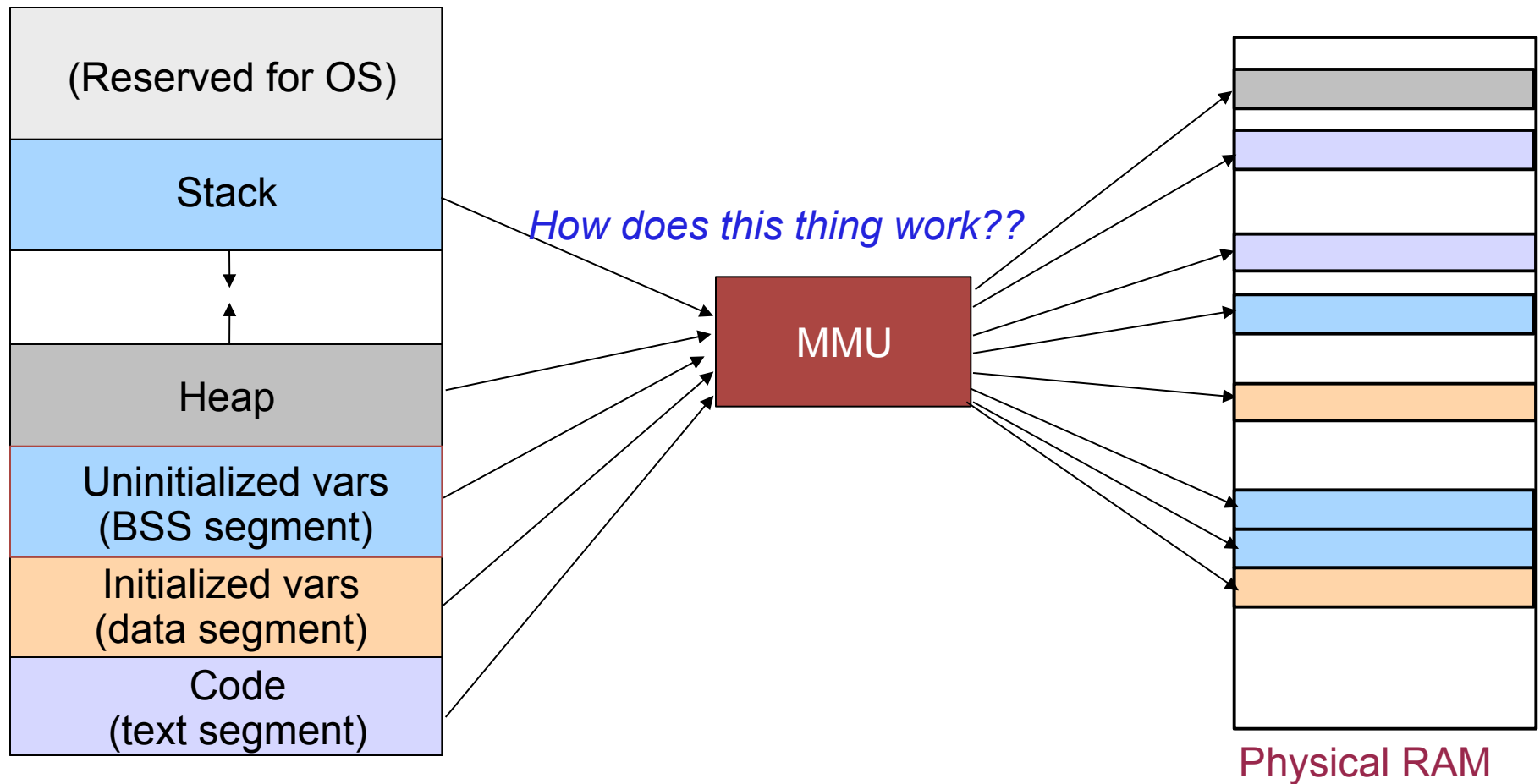
Slides adapted in part from material by
Matt Welsh, Harvard U.

[Recap: Virtual Addresses]

- A **virtual address** is a memory address that a process uses to access its own memory
 - Virtual address \neq actual physical RAM address
 - When a process accesses a virtual address, the MMU hardware **translates** the virtual address into a physical address
 - The OS determines the mapping from virtual address to physical address
- Benefit: Isolation
 - Virtual addresses in one process refer to **different** physical memory than virtual addresses in another
 - Exception: shared memory regions between processes (discussed later)
- Benefit: Illusion of larger memory space
 - Can store unused parts of virtual memory on disk temporarily
- Benefit: Relocation
 - A program does not need to know which physical addresses it will use when it's run
 - Can even change physical location while program is running



[Mapping virtual to physical addresses]



[Translating virtual to physical]

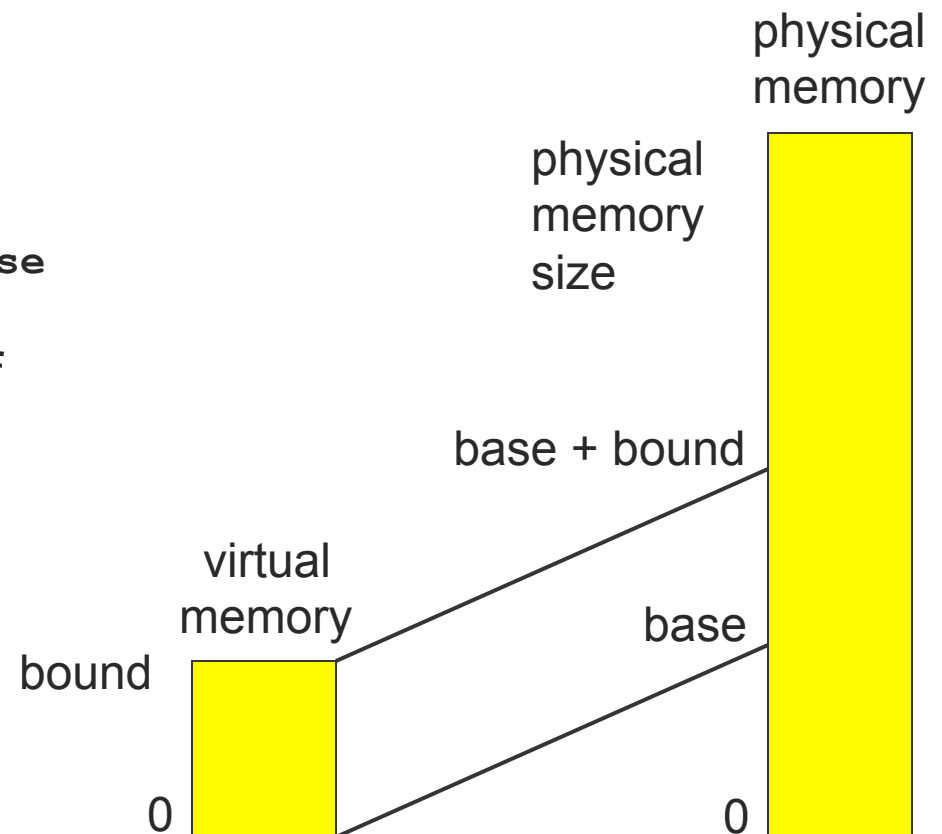
- Can do it almost any way we like
- But, some ways are better than others...
- Strawman solution from last time:
base and bound



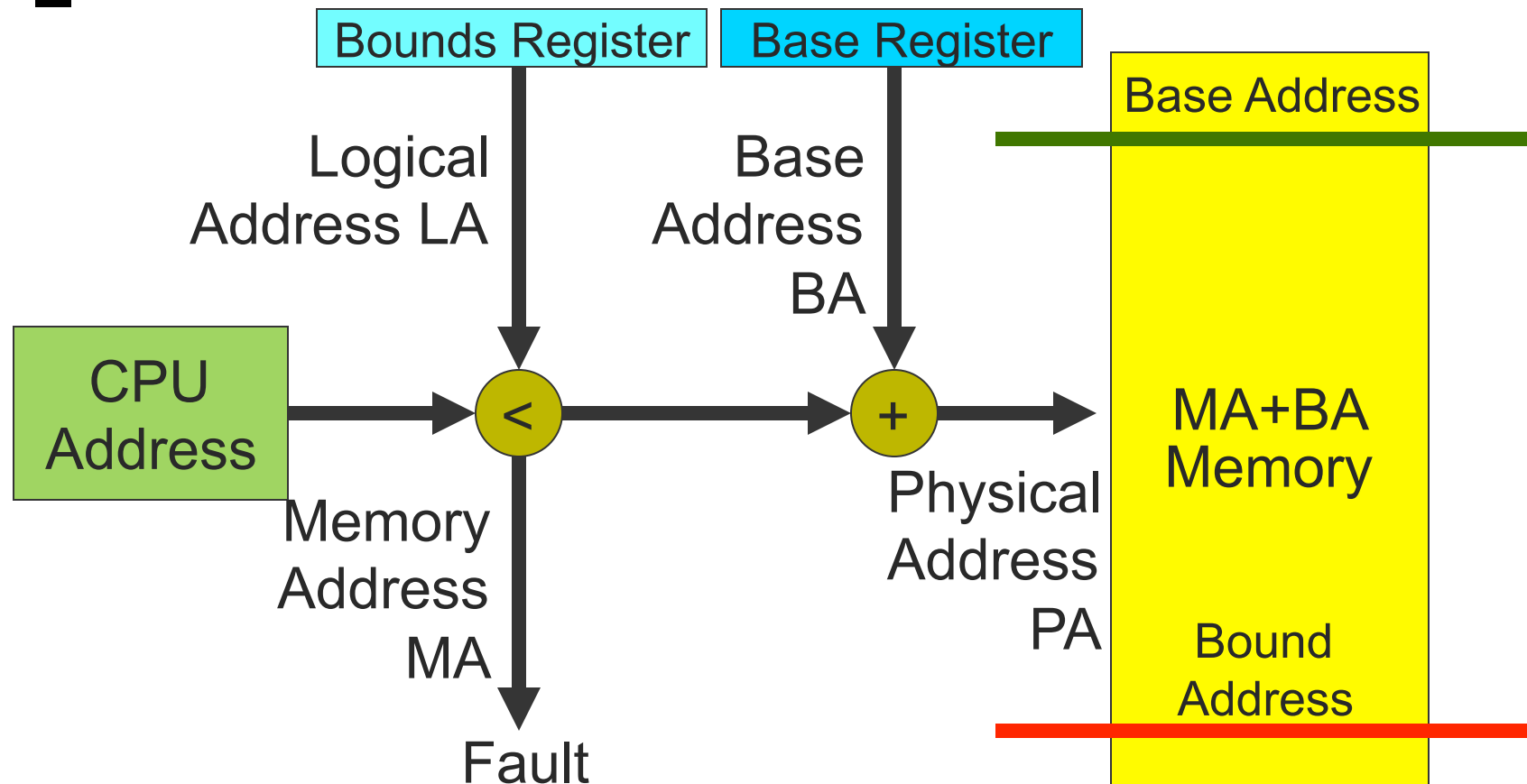
[Base and bound]

```
if (virt addr > bound)
    trap to kernel
else
    phys addr = virt addr + base
```

- Process has the illusion of running on its own dedicated machine with memory $[0, \text{bound})$
- Provides protection from other processes also currently in memory



Base and bound

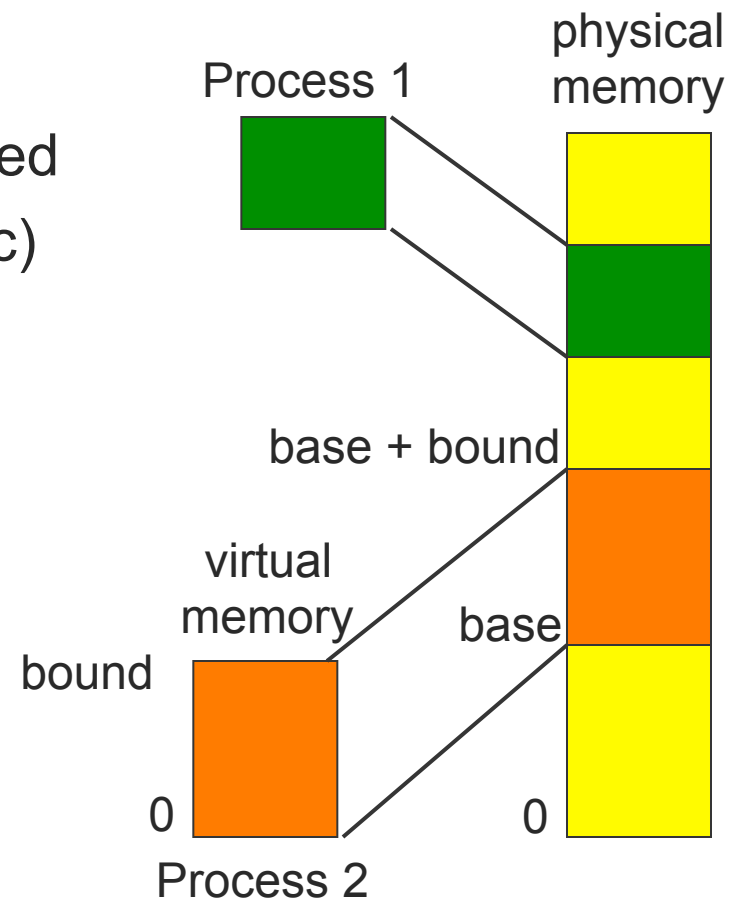


Base: start of the process's memory partition
Bound: length of the process's memory partition

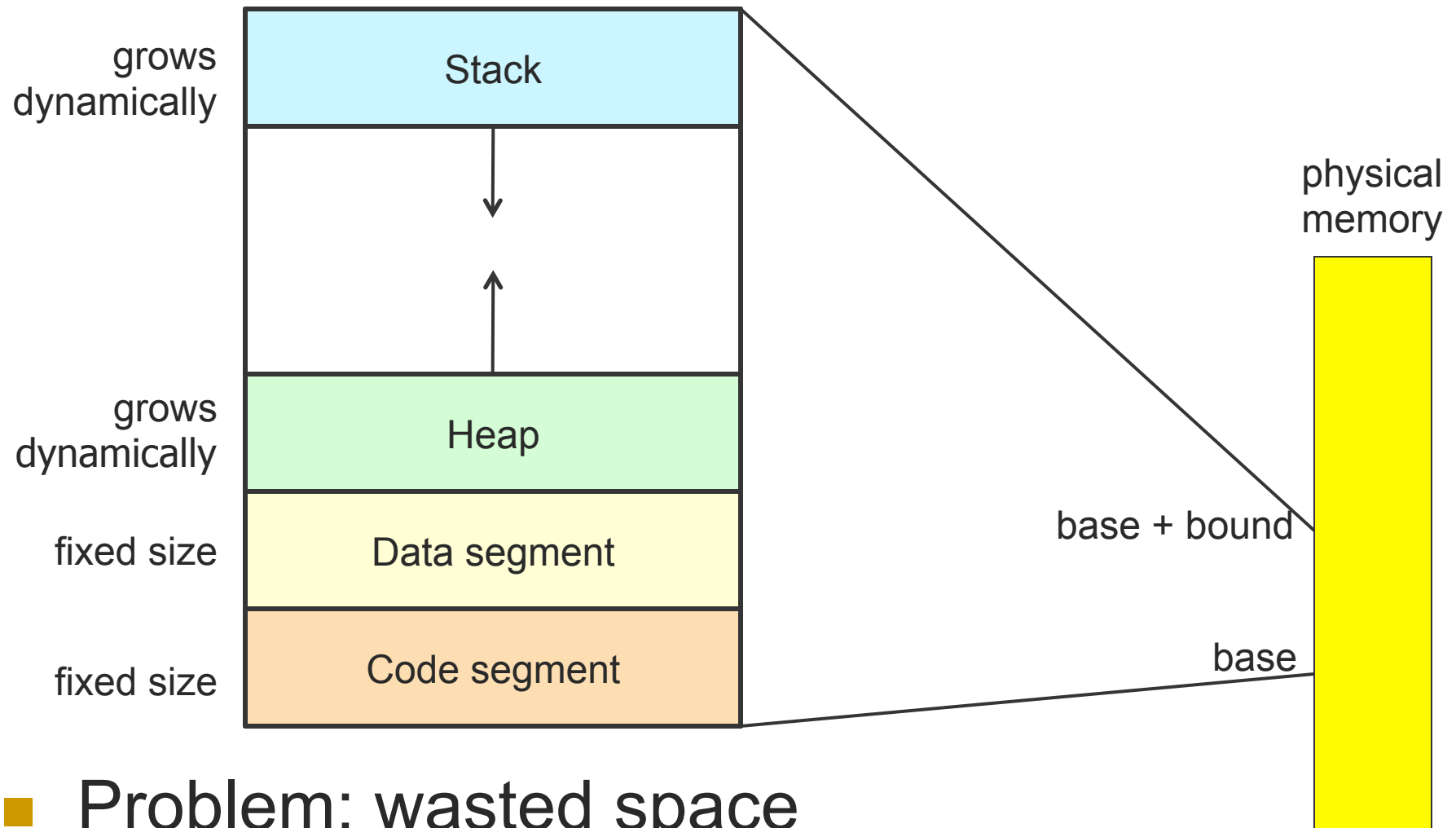


[Base and bounds]

- Problem: Process needs more memory over time
 - Stack grows as functions are called
 - Heap grows upon request (malloc)
 - Processes start and end
- How does the kernel handle the address space growing?
 - **You are the OS designer**
 - **Design strategy for allowing processes to grow**



[But wait, didn't we solve this?]



■ Problem: wasted space

- And must have virtual mem \leq phys mem

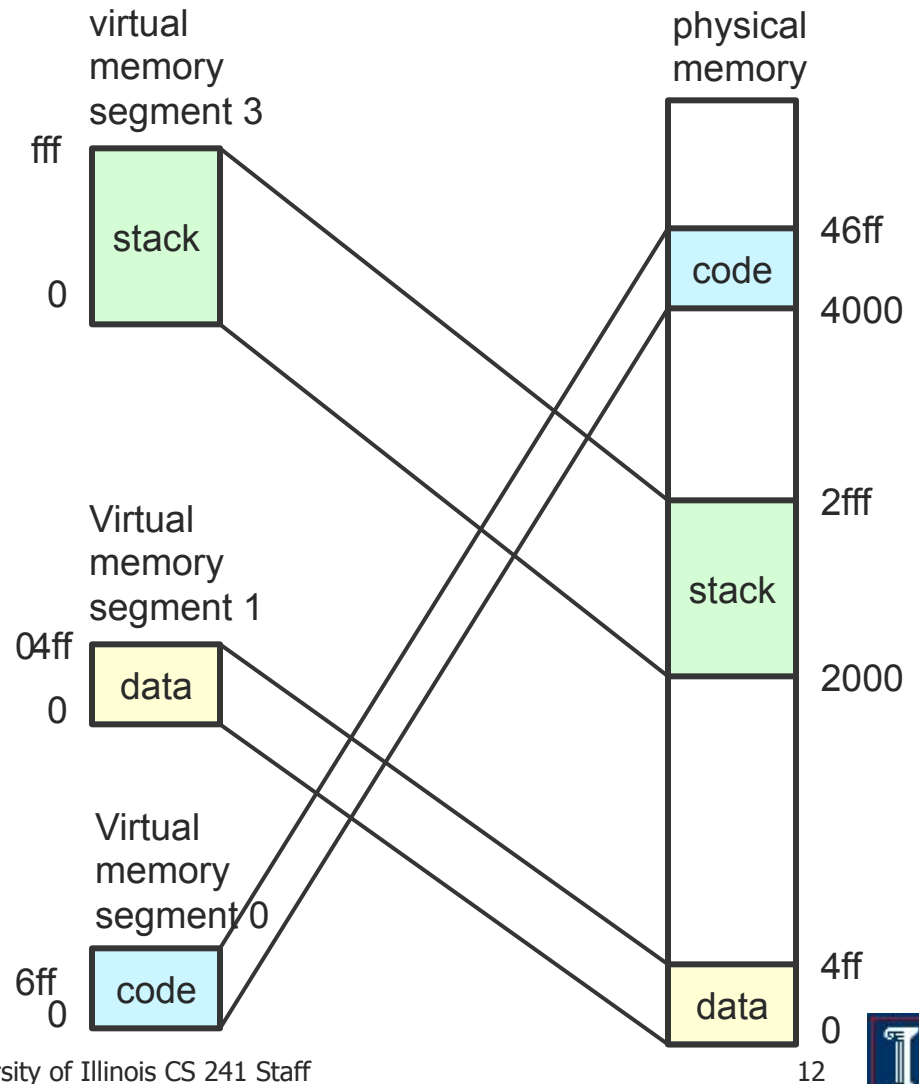
[Another attempt: segmentation]

- Segment
 - Region of contiguous memory
- Segmentation
 - Generalized base and bounds with support for multiple segments at once



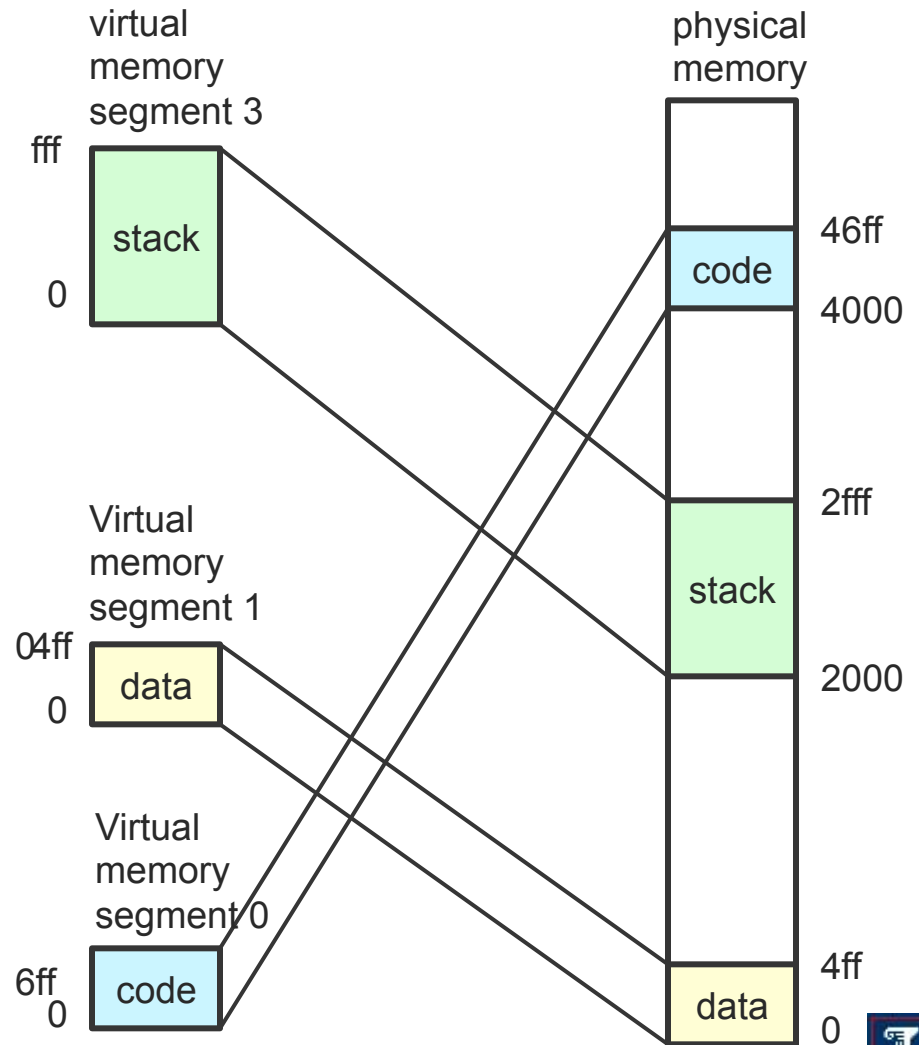
[Segmentation]

Seg #	Base	Bound	Description
0	4000	700	Code segment
1	0	500	Data segment
2	Unused		
3	2000	1000	Stack segment



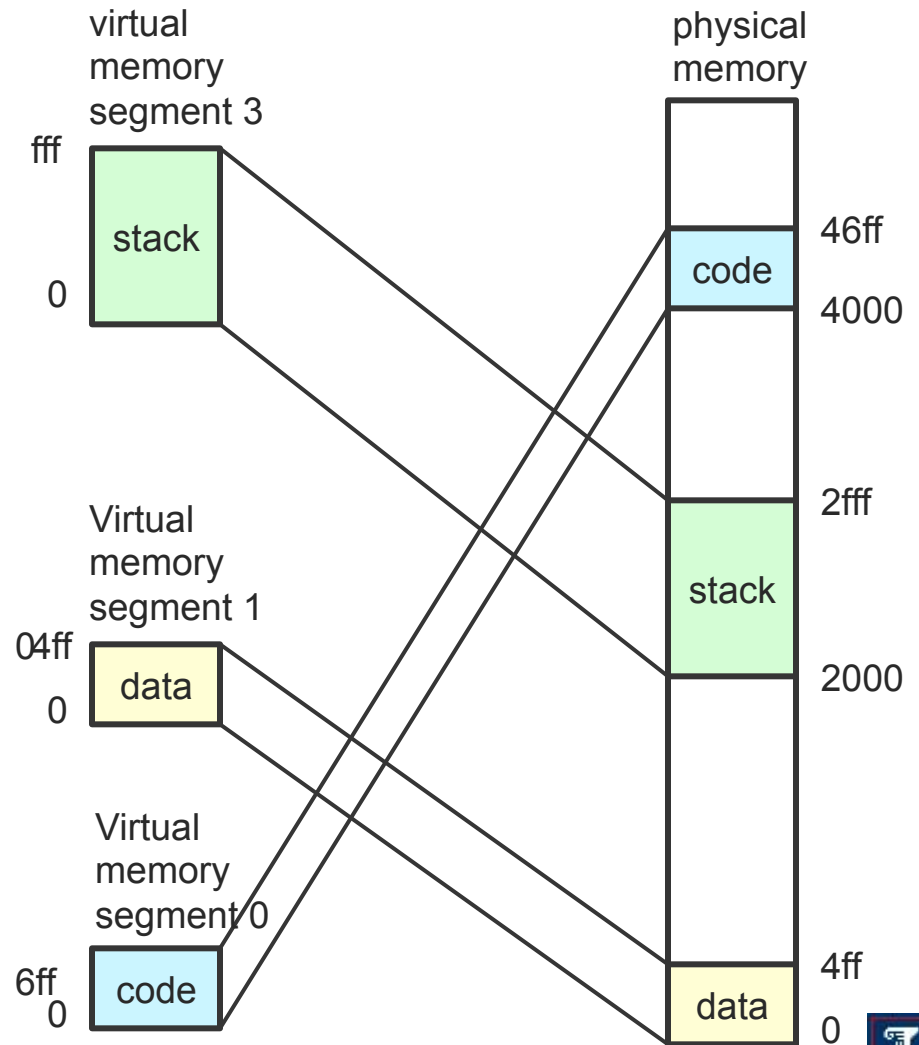
[Segmentation]

- Segments are specified many different ways
- Advantages over base and bounds?
- Protection
 - Different segments can have different protections
- Flexibility
 - Can separately grow both a stack and heap
 - Enables sharing of code and other segments if needed



[Segmentation]

- Segments are specified many different ways
- What are the advantages over base and bounds?
- What must be changed on context switch?
 - Contents of your segmentation table
 - A pointer to the table, expose caching semantics to the software (what x86 does)



[Recap: mapping virtual memory]

- **Base & bounds**

- Problem: growth is inflexible
- Problem: external fragmentation
 - As jobs run and complete, holes left in physical memory

- **Segments**

- Resize pieces based on process needs
- Problem: external fragmentation
- Note: x86 used to support segmentation, now effectively deprecated with x86-64

- **Modern approach: Paging**

