Segmentation

Glenn Bruns
CSUMB

Lecture Objectives

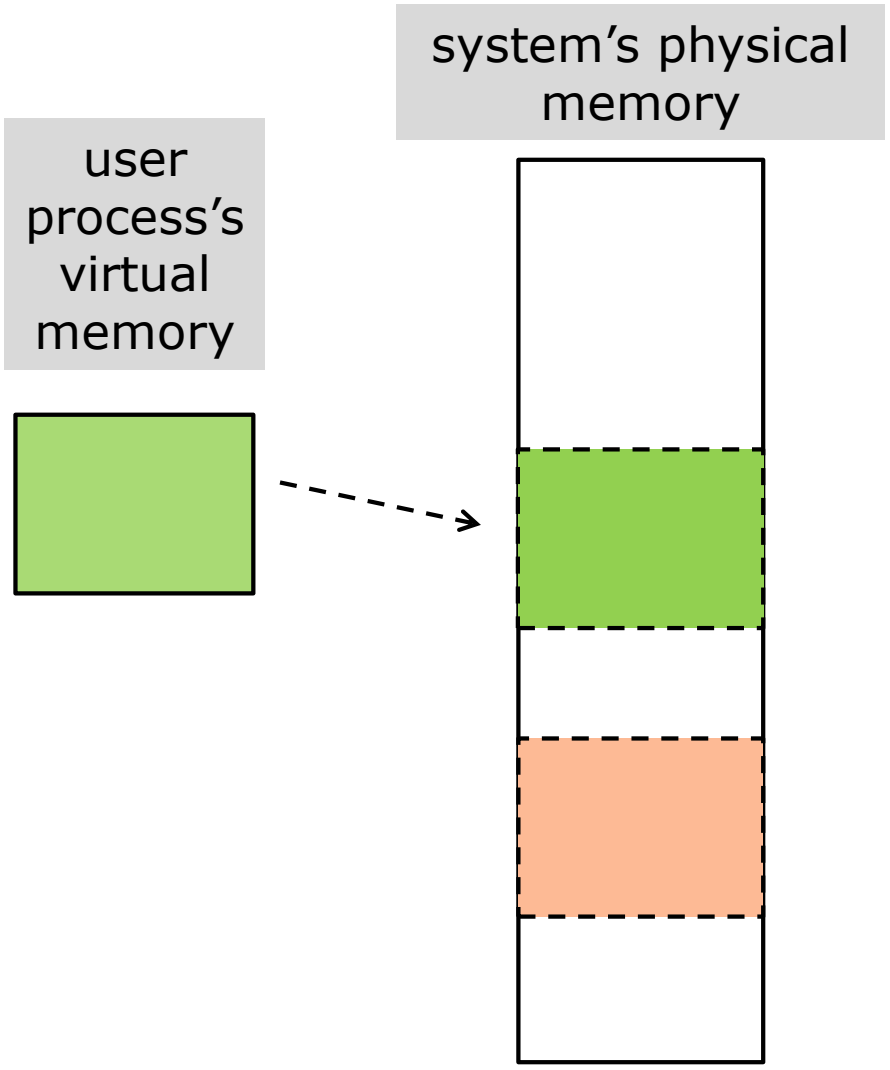
After this lecture, you should be able to:

- Explain virtual memory using segmentation
- Simulate address translation with segmentation

Base-and-bounds

With base-and-bounds, we assumed:

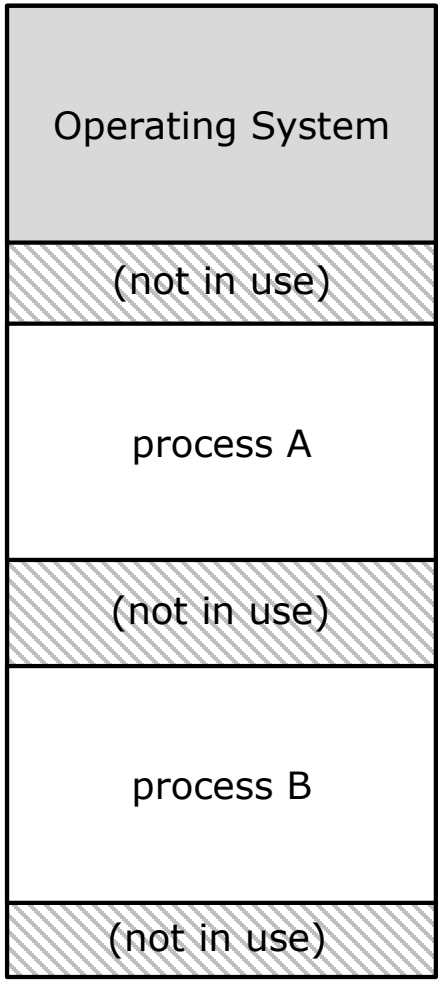
- each process's address space shall be contiguous in physical memory



Problem: waste

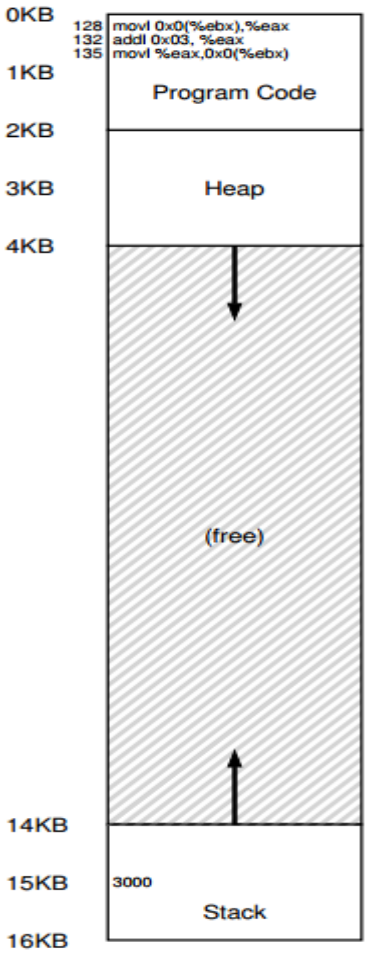
Most processes don't use their entire address space

A process' virtual space consumes a big contiguous piece of memory

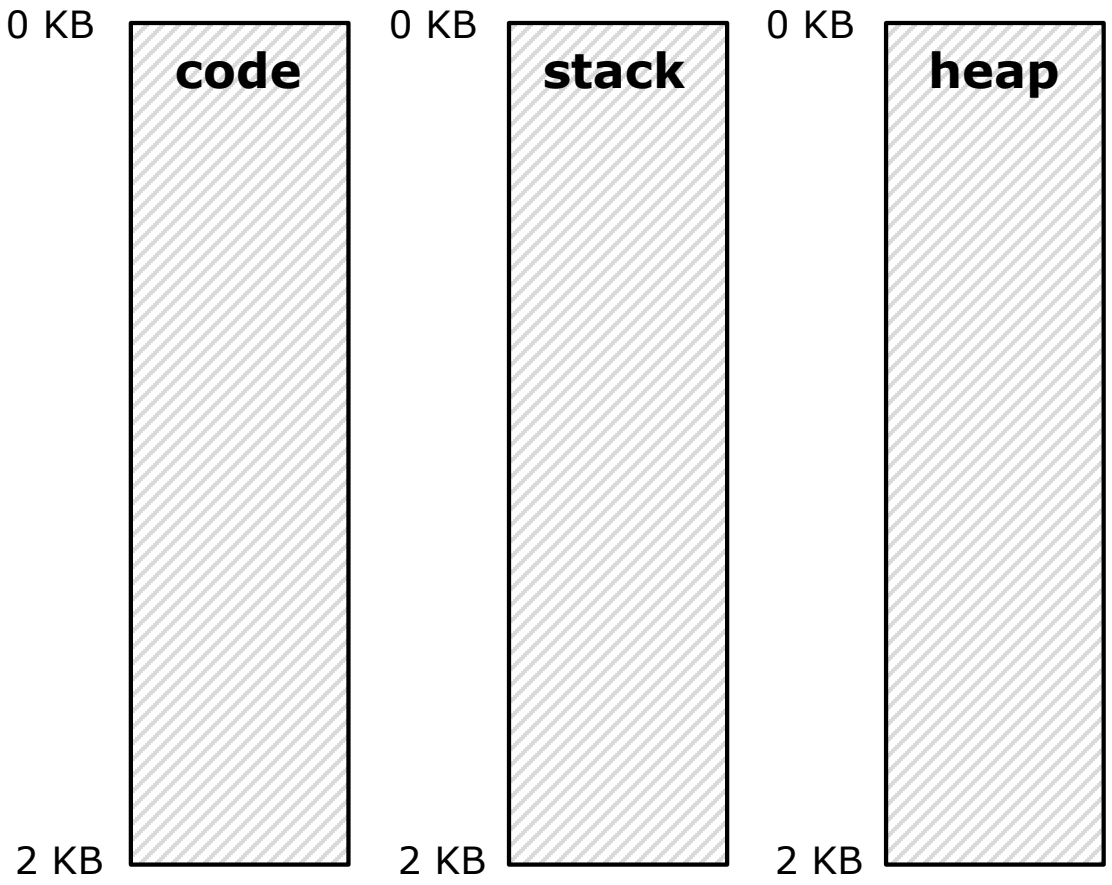


Idea: Segmented virtual memory

old concept

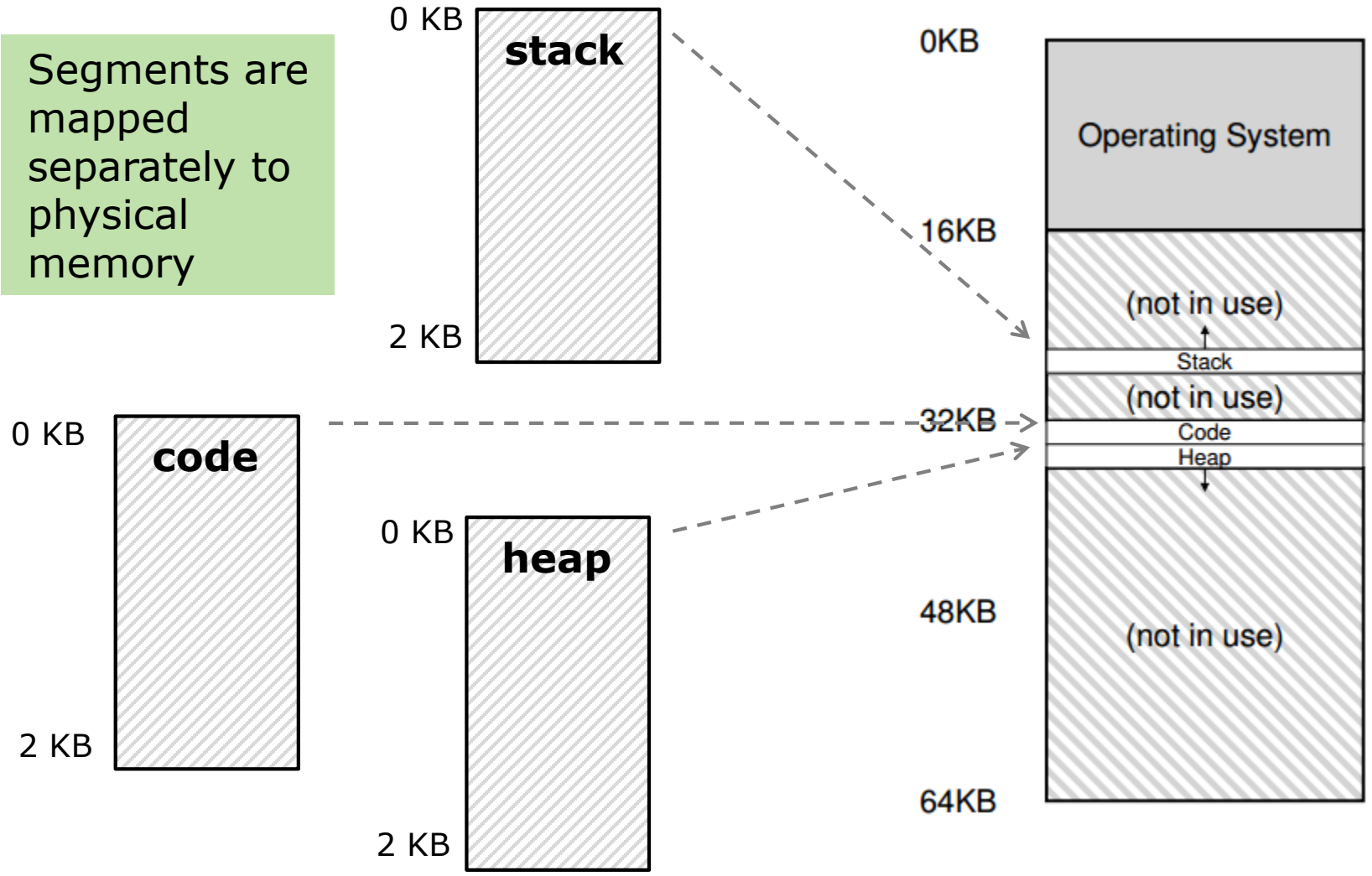


new concept



Mapping segments into phys. memory

Segments are mapped separately to physical memory



Address Translation

A virtual address now has two parts:

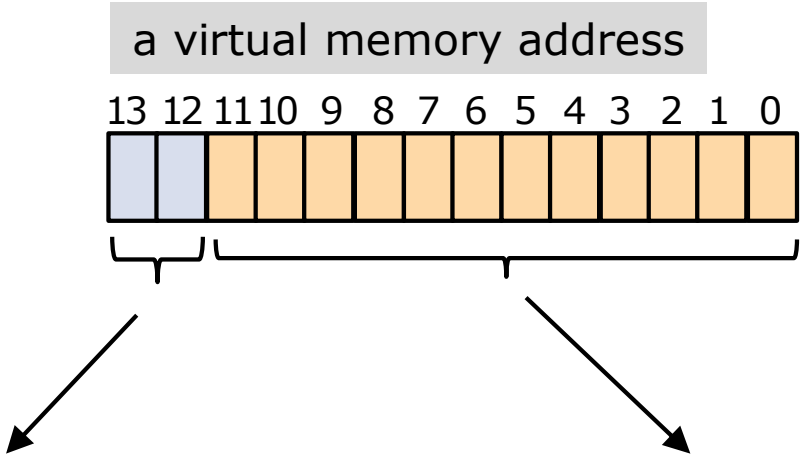
(segment, offset)

To translate we need a base and bounds for each segment

Address translation:

```
phys_addr(segment, offset) {
    if (offset ≥ bounds[segment])
        error
    else
        return(base[segment] + offset)
}
```

Example: Virtual Memory ala VAX



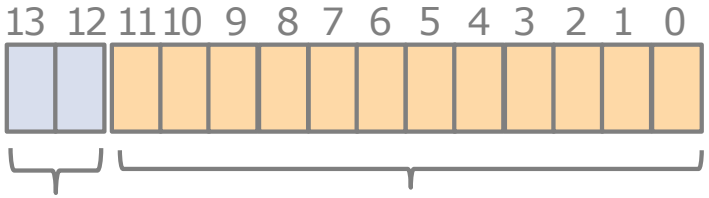
top two bits = segment:
00: code
01: heap
10: unused
11: stack

bottom 12 bits = offset

- 3 pairs of base/bounds registers used for address translation
- Stack grows backwards!

Viewing virtual address space

a virtual memory address:

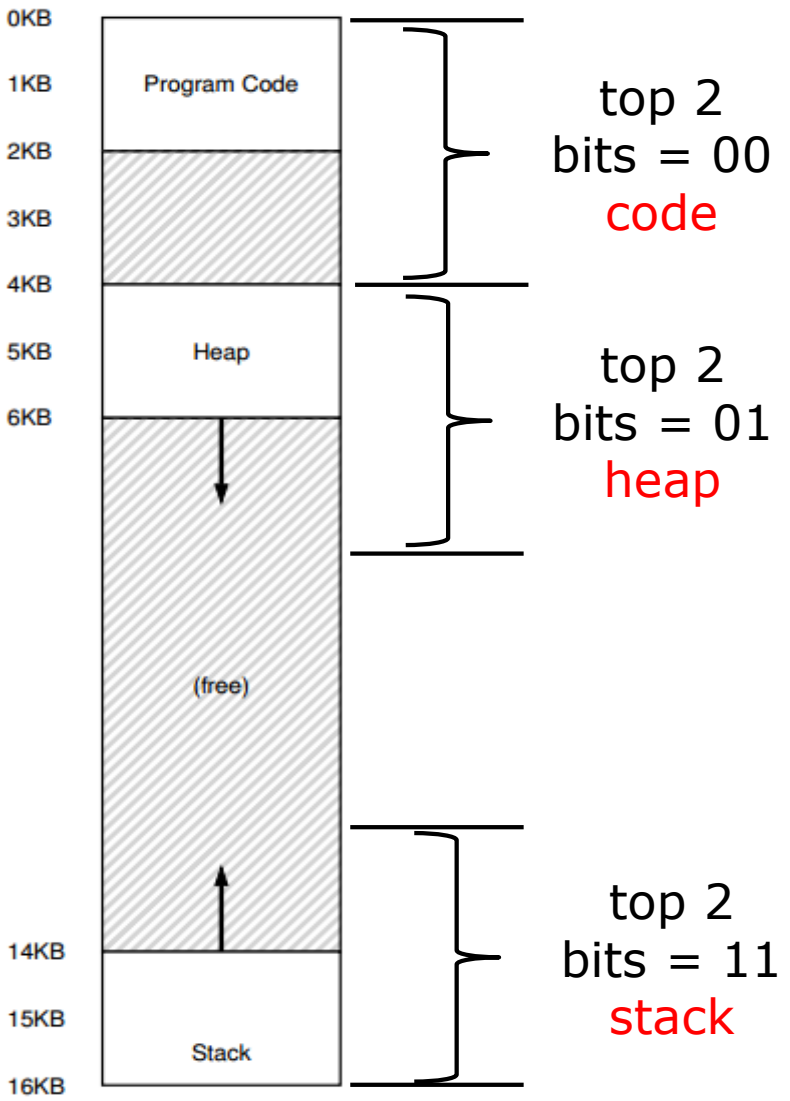


two bits for segment: (00 = code, 01= heap,…)

12 bits for offset

Size of this virtual address space is 16 KB.
Why? Because with 14 bits the biggest address is 16536.

Or, you can think of it as 4 segments, each with max size 4 KB, giving a total of 16 KB.

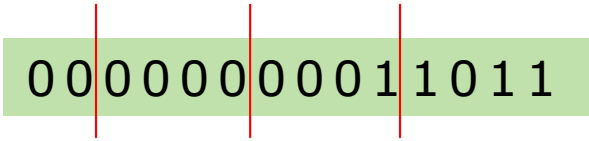


Address translations

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Note: authors write 32K or 32KB to mean 32 * 1024

Logical address: 25 (i.e. 0x0019)
- top two bits are 00, rest (offset) is 25
- 25 < 2K, so no error
- addr = 32K + 25 = 32768 + 25 = 32793



Logical address: 4202 (i.e. 0x106a)
- top two bits are 01, offset is 106
- 106 < 2K, so no error
- addr = 34K + 106 = 34816 + 106 = 34922

max_segment_size is 2^{12}, because 12 offset bits.

Logical address: 15833 (= 0x3DD9)
- top two bits are 11, offset is 3545
- neg. offset = 3545 – max_segment_size = 3545 - 4096 = -551
- abs(-551) < 2K, so no error
- addr = 28K – 551 = 28121

Segmentation and Protection

- If we segment memory, then we can assign permissions to each segment:
  - heap, stack: read-write
  - code: read-execute
- This can be done with a few bits in hardware
- Better protection!

Sharing code

- With read-only permission on code, we can share code between processes
- Two processes running the same app can have their code segments map to the same physical memory

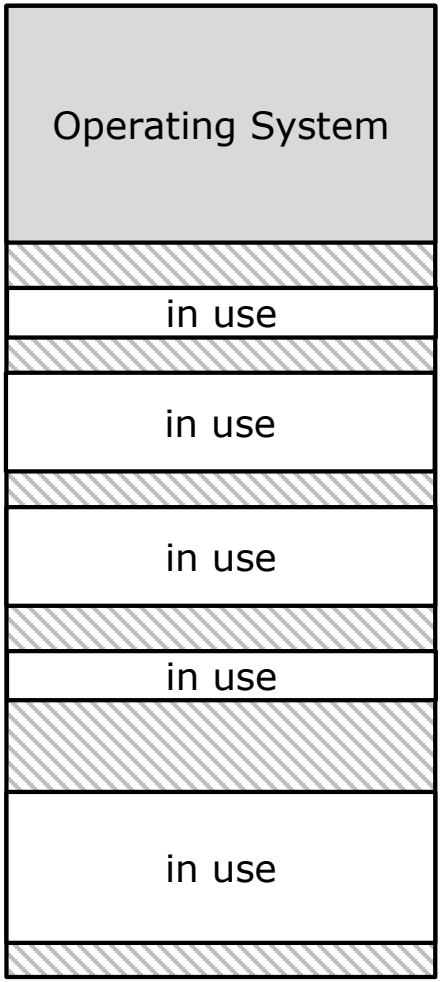
Segmentation as memory model

- Segmentation isn't just an implementation concept
- Segmentation is also a memory abstraction concept – it simplifies handling of memory
- A system could organize a process's virtual memory space into hundreds of segments

Exercise: what might be one hardware-related problem in this?

Fragmentation

- Segments can vary in size
- As processes are created and later terminate, memory will become fragmented
- Solutions:
  - "burping"
  - better initial allocation



Summary

- Segmentation treats virtual memory as logically separate parts called “segments”
- Segments are mapped to physical memory separately
- Base-and-bounds idea is still used, but for each segment
- Protection is improved as segments can be given permissions independently

Address translation with segmentation

If segment grows positive:

```
if offset > bounds(segment)
    segmentation fault
else
    physical_address = offset + base(segment)
```

If segment grows negative:

```
neg_offset = offset – max_segment_size
if abs(neg_offset) > bounds(segment)
    segmentation fault
else
    physical_address = offset + neg_offset
```

Example: if offset is 12 bits, then max segment size is 2^{12}