

Operating System (OS)

CS232

Memory Segmentation

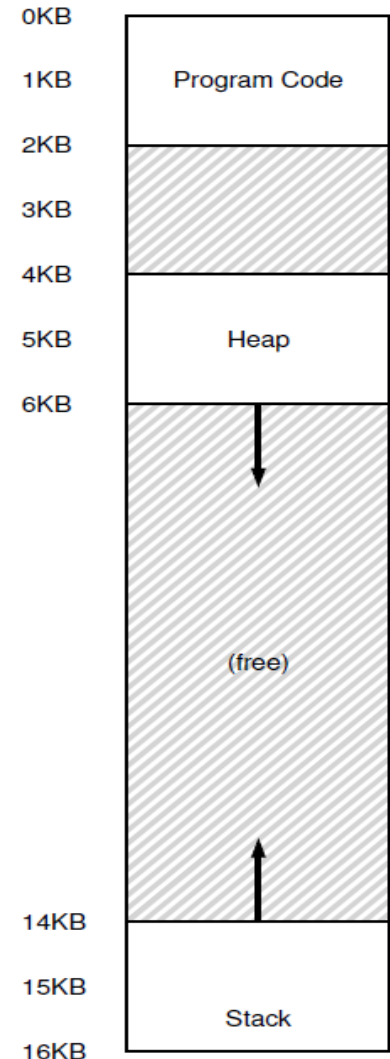
Dr. Muhammad Mobeen Movania

Outlines

- Address spaces in memory
- What is segmentation
- How do we identify segments
- Some examples of segmentation
- Protection and sharing
- OS and hardware support
- Summary

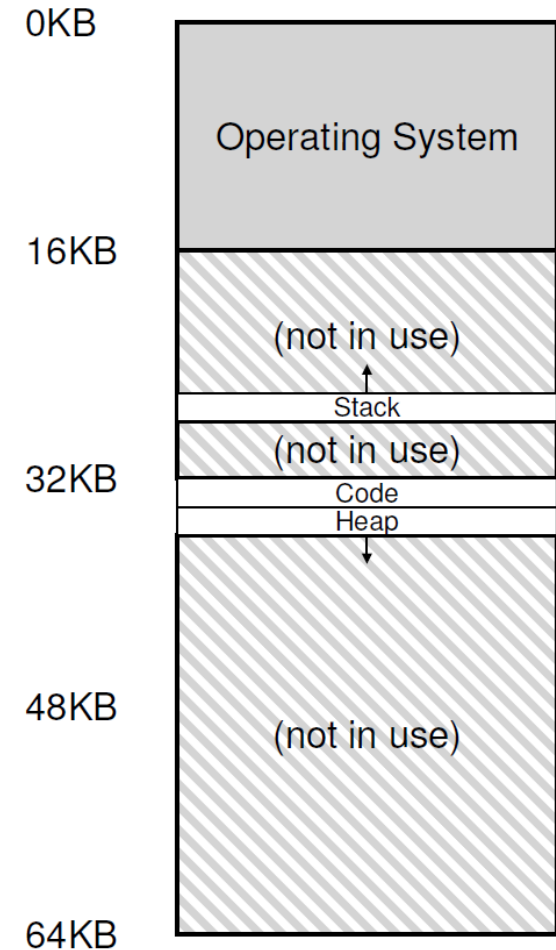
Address spaces in memory

- Using base/bounds registers to store address space is not good
 - address spaces are sparsely populated i.e. most of the space is empty due to *fragmentation*
 - Waste of RAM space as we cannot populate more processes
- What if a process address space is bigger than RAM?
 - We cant split process address space in sub-parts to make it fit in memory



Segmentation

- Key Idea
 - Think of address space to consists of logical segments
 - Instead of having one base/bound register pair, we could have per-segment base/bound pairs?
- Each segment can then be placed independently in a separate part of physical memory!



Segmentation

- Advantages
 - Saves memory
 - Large address spaces can be accommodated
- Needs hardware support
 - Three base/bound pairs in this case

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

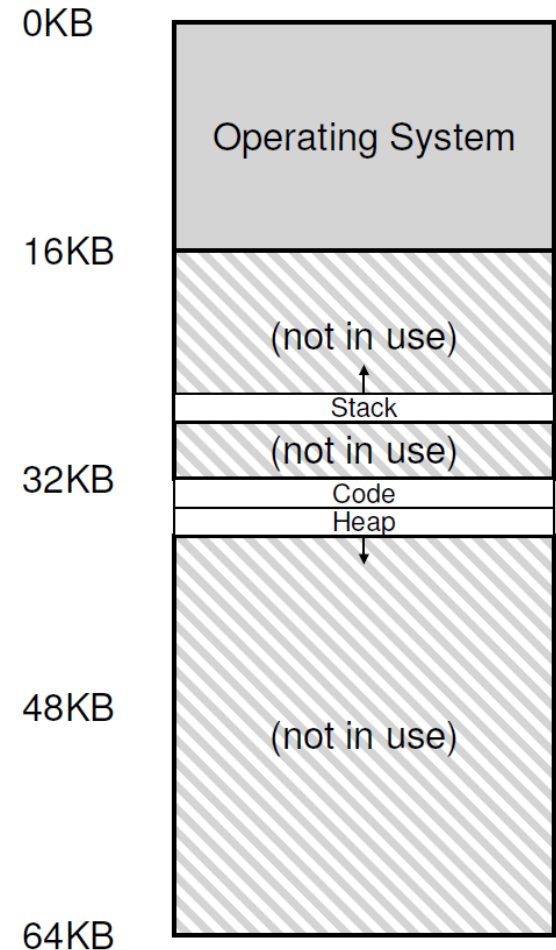


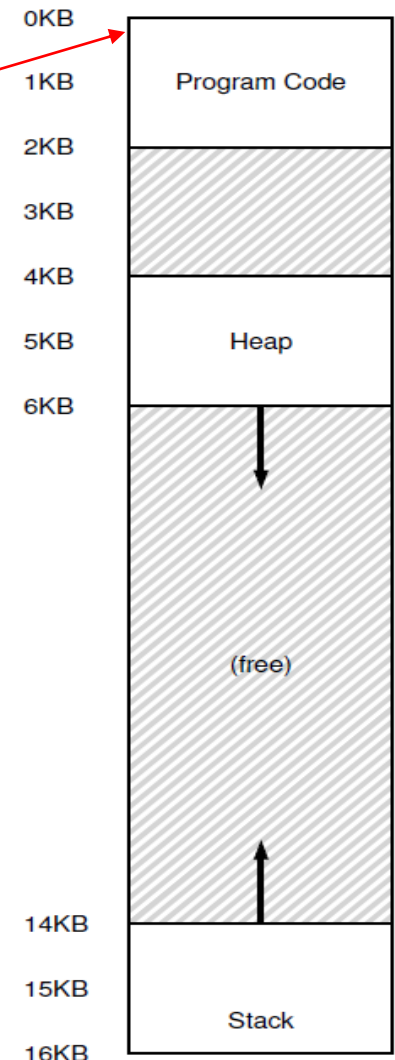
Figure 16.3: **Segment Register Values**

Segmentation: example

- Virtual address 100 (code segment)
 - Physical address = $32k + 100 = 32868$
- Crux
 - Offset is same (100) both from start of virtual address space and the start of code segment

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Figure 16.3: **Segment Register Values**



Segmentation: example

- Virtual address 4200 (heap segment)
 - Physical address = $34k + 4200 = 39016$ (wrong!)
- Why
 - Offset is not same
 - Its (4200) from the start of virtual address space and (104) from the start of the heap segment
- Remedy
 - Calculate heap offset which is the offset from the start of heap segment:
 - $4200 - 4096 = 104$!
 - Physical address = $34k + 104 = 34920$
- What if we try to refer to an address outside the segment?
 - Segmentation Fault!!

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

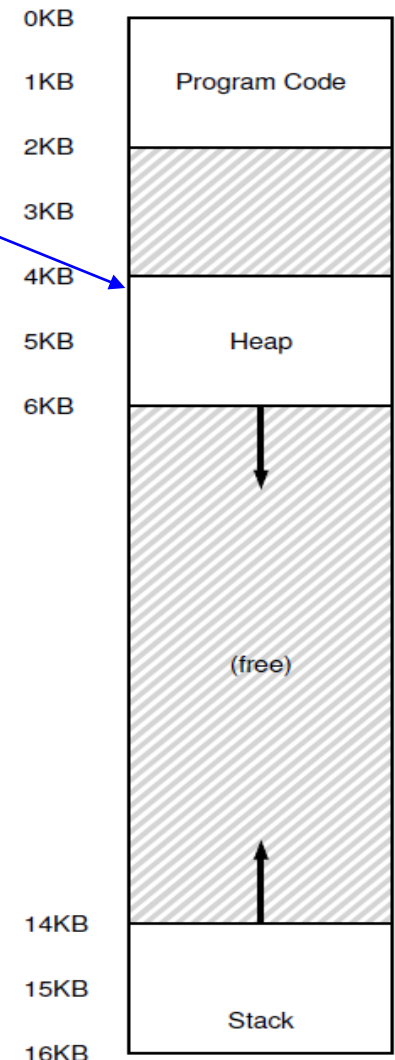
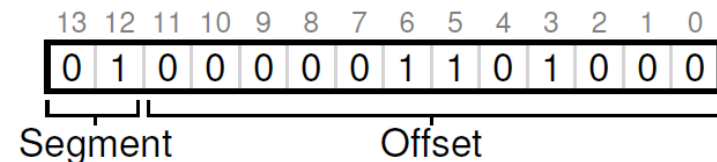
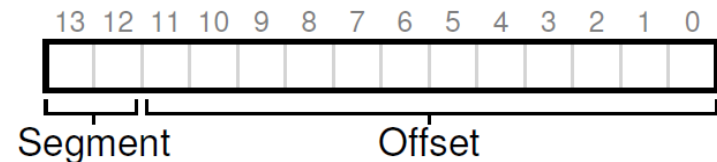


Figure 16.3: Segment Register Values

Which segment are we accessing

- How does the hardware know which base/bound pair to use?
- Two ways
 - Explicit Approach
 - Implicit Approach
- Explicit approach:
 - We know we have 3 segments
 - We can use the top two bits to identify the segment
 - If segments are placed at correct offset:
 - 00 -> code
 - 01 -> heap
 - 11 -> stack
 - Calculating offset is also easier
 - VA → 4200



Which segment are we accessing... (2)

- Explicit approach

```
1  // get top 2 bits of 14-bit VA
2  Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3  // now get offset
4  Offset  = VirtualAddress & OFFSET_MASK
5  if (Offset >= Bounds[Segment])
6      RaiseException(PROTECTION_FAULT)
7  else
8      PhysAddr = Base[Segment] + Offset
9      Register = AccessMemory(PhysAddr)
```

In our running example, we can fill in values for the constants above. Specifically, `SEG_MASK` would be set to `0x3000`, `SEG_SHIFT` to `12`, and `OFFSET_MASK` to `0xFFF`.

What's going on with those bitmasks?

- Suppose we have virtual address: 4200
- Binary of 4200 in 14 bits: 010000001101000
- What are we trying to get
 - First two bits to find the segment we are in
 - Remaining bits to get the offset
- Finding the segment we are in
 - 1) Bitwise & given address with following mask
1100000000000000 = 0x3000
 - 2) Right shift the result of step 1 by 12 bits to get the actual value of 2 bits
$$(\text{address} \ \& \ 0\text{x}3000) \gg 12$$

Finding the segment

- **(address & 0x3000)**

- 01000001101000

- 1100000000000000

- -----

- 0100000000000000

- >> 12

- -----

- 0000000000000001

Possible Segments

00 -> code

01 -> heap

11 -> stack

Finding the offset


- 1) Bitwise & given address with mask
 - 0011111111111111 = 0x0FFF = 0xFFF
- **(address & 0xFFF)**
 - 01000001101000
 - 00111111111111
 - -----
 - 00000001101000

Which segment are we accessing... (3)

- **Implicit approach:**
 - Hardware determines the segment from the provenance (origins) of VA:
- If address is generated via PC (Code segment)
- If address is generated via SP (Stack segment)
- Any thing else (Heap segment)

What about the stack?

- It grows backwards!
 - VA: 16KB – 14KB → PA: 28KB – 26KB
- Need hardware support, need 1 bit
- Process addresses:
 - VA: 15KB. 11 1100 0000 0000 (hex 0x3C00) VA offset = 3KB
 - To obtain physical offset we've to subtract max segment size from this offset i.e. 3KB – 4KB = -1 →
Physical address = 28KB – 1KB = 27KB



Segment	Base	Size (max 4K)	Grows Positive?
Code ₀₀	32K	2K	1
Heap ₀₁	34K	3K	1
Stack ₁₁	28K	2K	0

Protection and Sharing

- Protection
 - We can mark different segments as read only, writeable, executable
- Sharing
 - Code segment can be marked read only and shared b/w processes

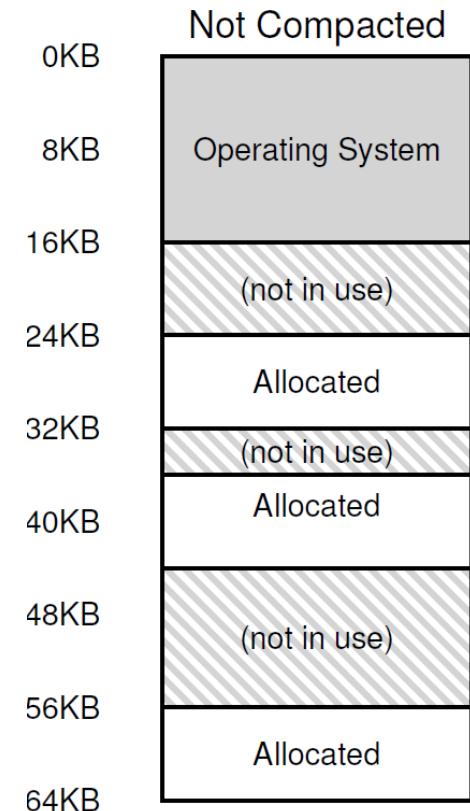
Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Coarse grained vs Fine grained segmentation

- We only had 3 segments → coarse grained
- Some systems provide support for 1000s of segments
 - via Segment Tables

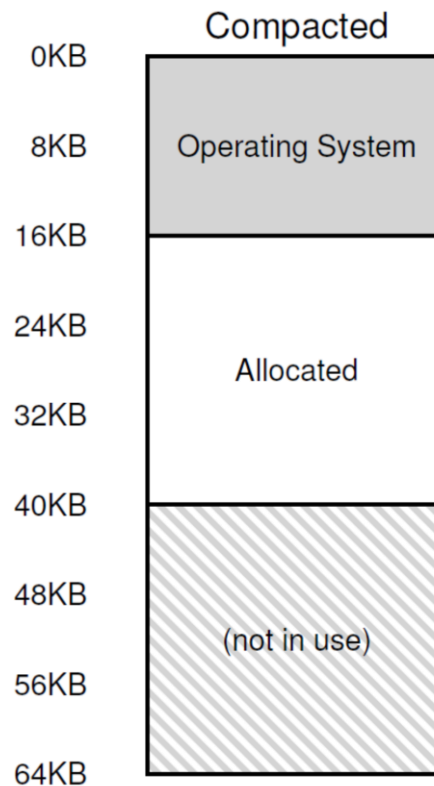
Segmentation: OS Support

- Set up the segment registers when a process runs
- Save and load segment registers on context switch
- Free space management
 - OS has to find space for segments of a new process
 - Segments may be of unequal sizes
 - Processes come and go
 - Memory may become full of small holes
 - This is called External Fragmentation



Segmentation: OS support

- Fragmentation can render RAM unusable
- Compaction is one solution
 - It is expensive



Summary

- We learned what segmentation is.
- It can better support sparse address spaces, by avoiding the huge potential waste of memory
- Allows possibility of code sharing
- Variable sized fragments result in external fragmentation
- Defragmentation can help reduce external segmentation but it is costly.