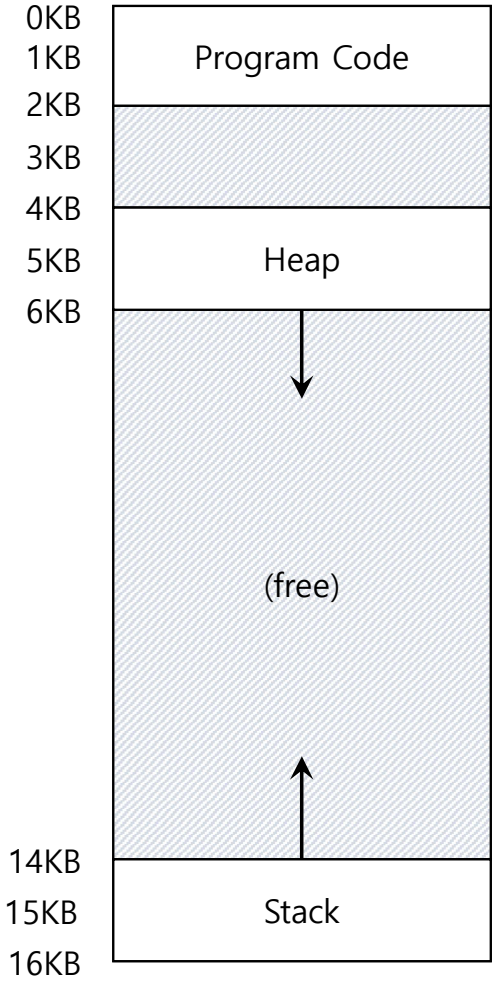
21CS2109AA

Operating Systems

Session 13

Segmentation

Inefficiency of the Base and Bound Approach



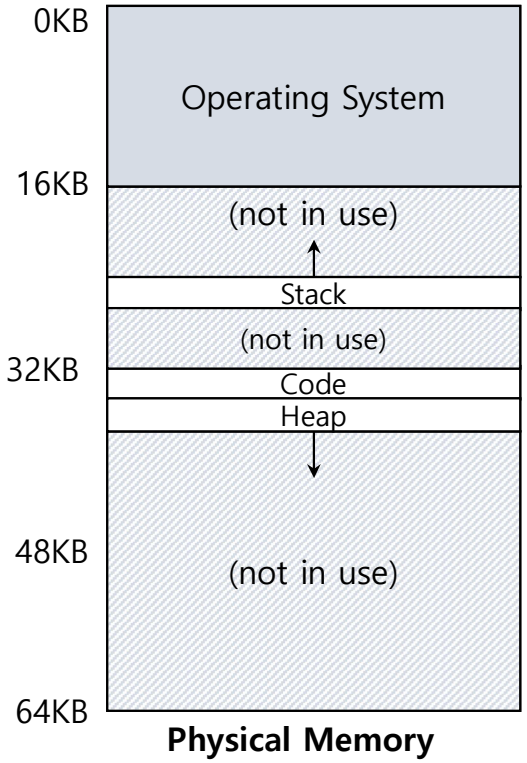
- **Big chunk of "free"** space right in the middle, between the stack and the heap.
- although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory;
- It also makes it quite hard to run a program when the entire address space doesn't fit into memory;
- Thus, base and bounds is not as flexible as we would like.

Segmentation

- Segment is just **a contiguous portion** of the address space of a particular length.
  - Logically-different segment: code, stack, heap

- Each segment can be **placed** in **different part of physical memory**.
  - **Base** and **bounds** exist **per each segment**.

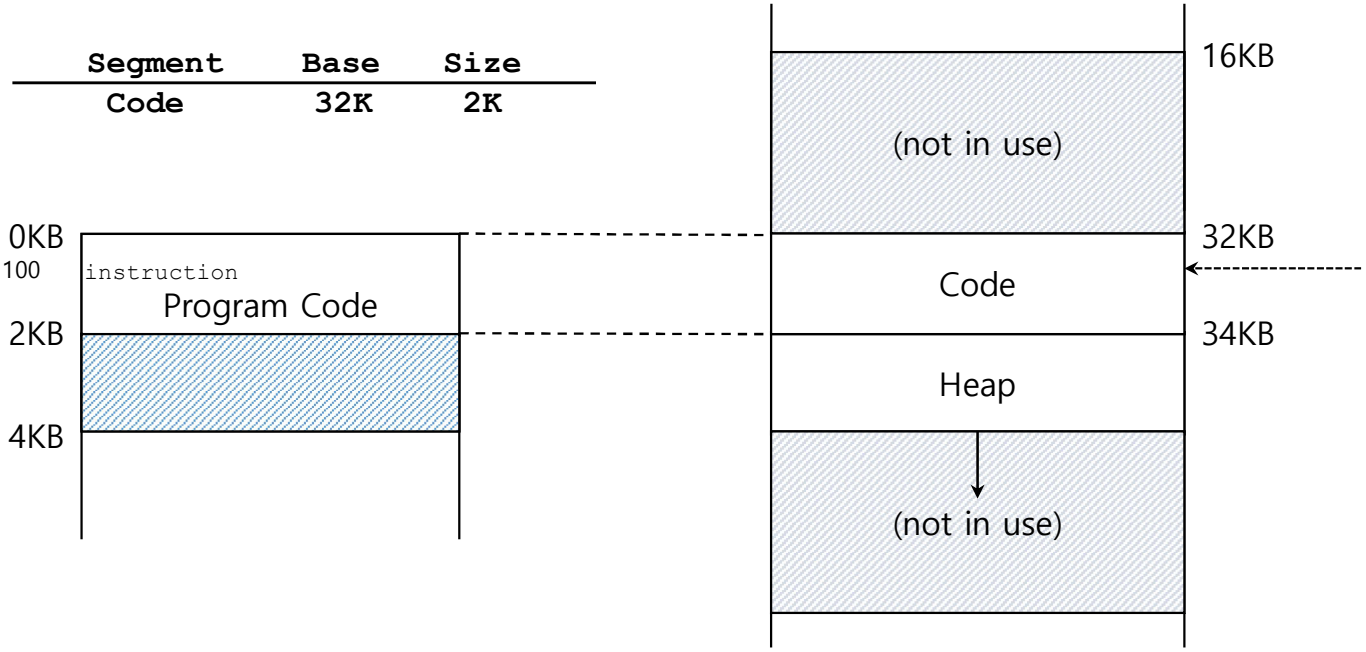
Placing Segment In Physical Memory



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

Address Translation on Segmentation

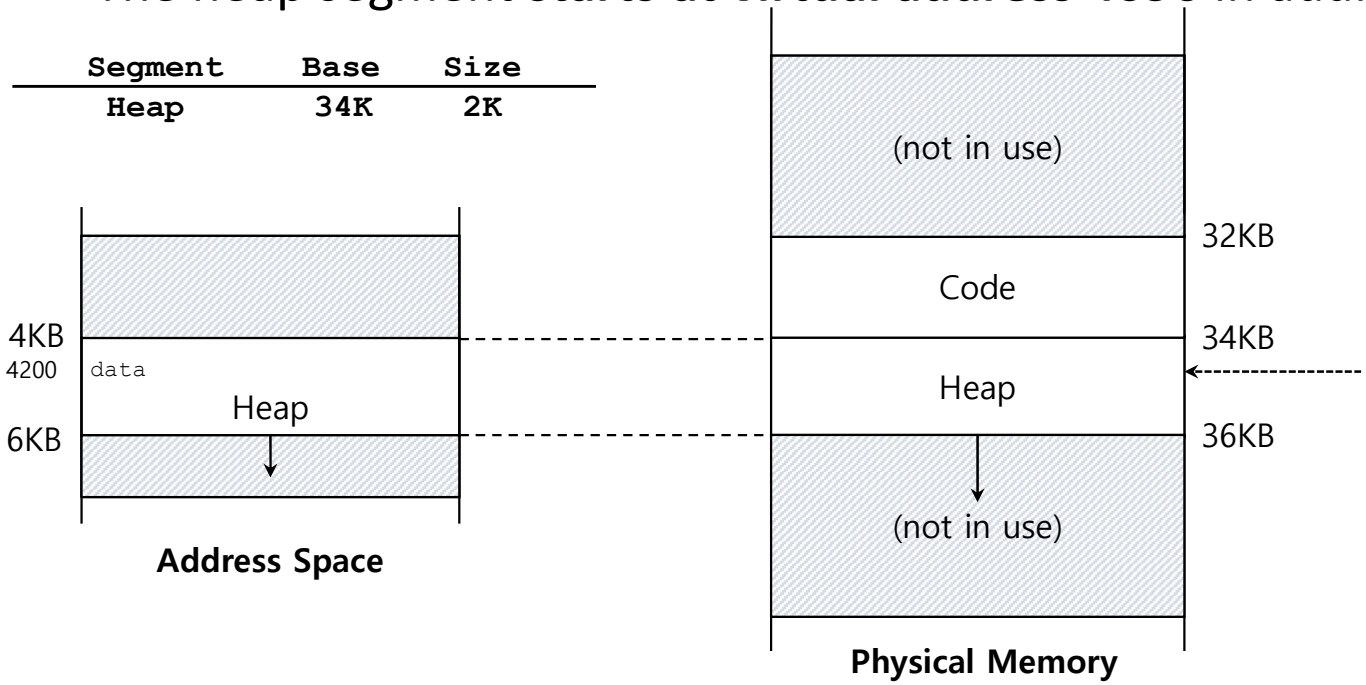
- The `offset` of virtual address 100 is `100`.
  - The code segment **starts at virtual address 0** in address space.



- Assume a reference is made to virtual address 100 (which is in the code segment).
- When the reference takes place (say, on an instruction fetch), the hardware will add the base value to the offset into this segment (100 in this case) to arrive at the desired physical address: 100 + 32KB, or 32868.
- It will then check that the address is within bounds (100 is less than 2KB), find that it is, and issue the reference to physical memory address 32868.

Address Translation on Segmentation(Cont.)

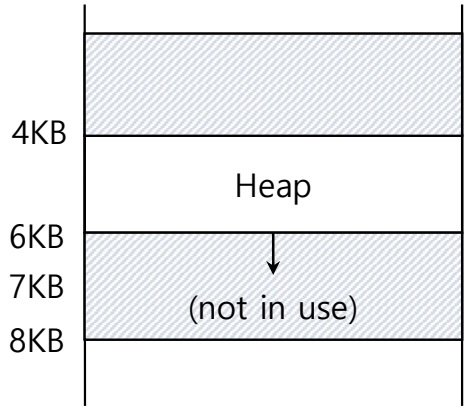
- The `offset` of virtual address 4200 is `104`.
  - The heap segment **starts at virtual address 4096** in address space.



- Now let's look at an address in the heap, virtual address 4200. If we just add the virtual address 4200 to the base of the heap (34KB), we get a physical address of 39016, which is not the correct physical address.
- What we need to first do is extract the offset into the heap, i.e., which byte(s) in this segment the address refers to.
- Because the heap starts at virtual address 4KB (4096), the offset of 4200 is actually 4200 – 4096 or 104. We then take this offset (104) and add it to the base register physical address (34K or 34816) to get the desired result: 34920.

Segmentation Fault or Violation

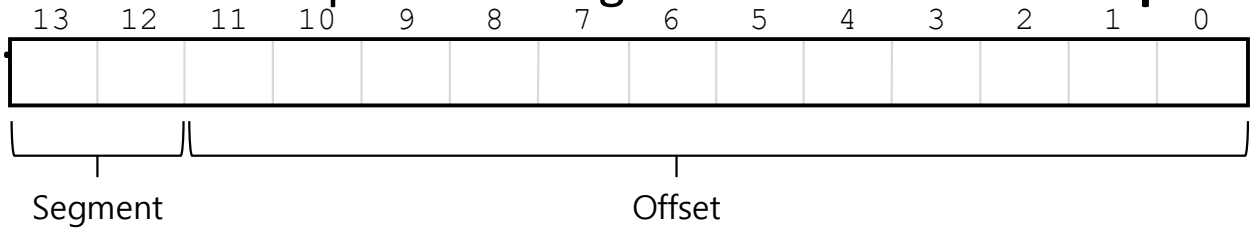
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS occurs **segmentation fault**.
  - The hardware detects that address is **out of bounds**.



Address Space

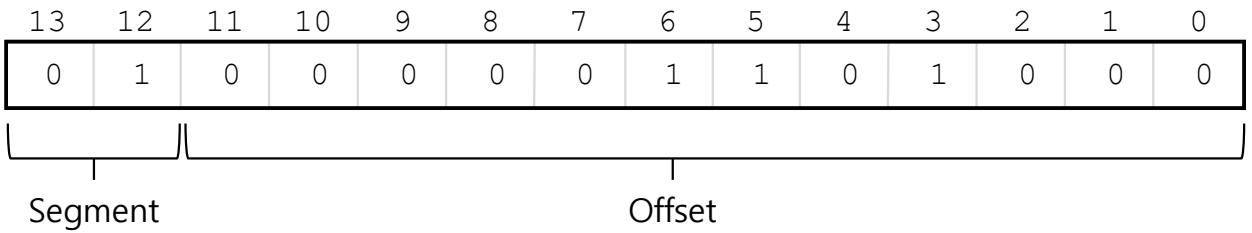
Referring to Segment

- H/W uses segment registers for translation.
- **Explicit approach**
  - Chop up the address space into segments based on the **top few bits** of virtual address.



- Example: virtual address 4200 (01000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11



Referring to Segment(Cont.)

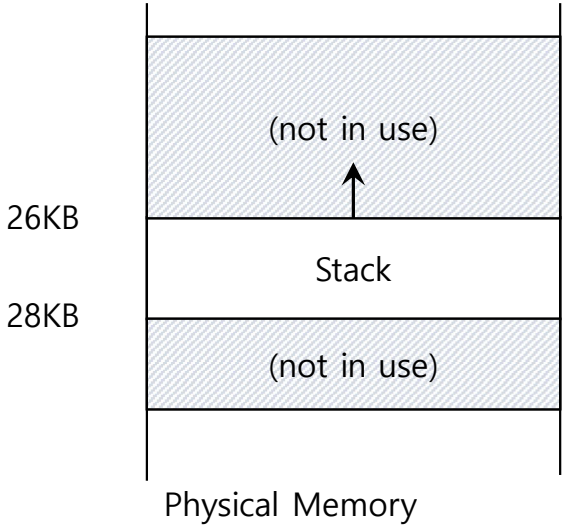
```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7   else
8        PhysAddr = Base[Segment] + Offset
9        Register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

- SEG_MASK = 0x3000(110000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (00111111111111)

- In the **implicit approach, the hardware determines** the segment by noticing how the address was formed.
- If, for example, the address was generated from the program counter (i.e., it was an instruction fetch), then the address is within the code segment;
- if the address is based off of the stack or base pointer, it must be in the stack segment; any other address must be in the heap.

Referring to Stack Segment

- Stack grows **backward**.
- **Extra hardware support** is need.
  - The hardware checks which way the segment grows.
  - 1: positive direction, 0: negative direction



Segment Register(with Negative-Growth Support)

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

Support for Sharing

- Segment can be **shared between address** space.
  - **Code sharing** is still in use in systems today.
  - by extra hardware support.
- Extra hardware support is need for form of **Protection bits.**
  - **A few more bits** per segment to indicate **permissions** of **read,** write and **execute**.

Segment Register Values(with Protection)

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

Fine-Grained and Coarse-Grained

- Segmentation as **coarse-grained,** as it chops up the address space into relatively large, coarse chunks. e.g., code, heap, stack.
- However, some early systems were more flexible and allowed for address spaces to consist of a large number smaller segments, referred to as **fine-grained segmentation**
- To support many segments, Hardware support with a **segment table** is required. Such segment tables usually support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways

OS support: Fragmentation

- **External Fragmentation**: little holes of **free space** in physical memory that make difficulty to allocate new segments.
  - There is **24KB free**, but **not in one contiguous** segment.
  - The OS **cannot** satisfy the **20KB request**.

- **Compaction**: **rearranging** the exiting segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere.
    - **Change** segment register value.

Memory Compaction