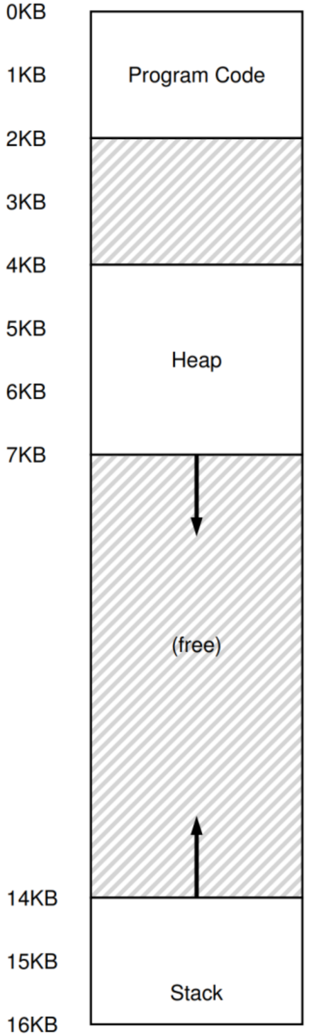
Segmentation

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Limitation of Base-Bounds Scheme

- **The base-bounds approach to virtualize memory is wasteful**
  - There is a big chunk of “free” space between heap and stack
  - The space between stack and heap is not being used by process it still takes up physical memory when relocating
  - It is quite hard to run a program when the entire address space does not fit into physical memory
  - Thus, base-bounds is not as flexible as we would like

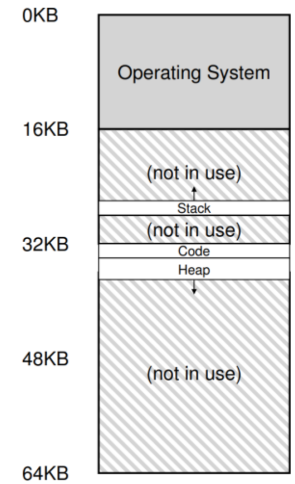


- **To solve this issue, segmentation was born**
  - Instead of having one base and bounds pair in MMU, let’s have a base and bounds pair **per** logical segment
  - A segment is just a contiguous portion of the address space of a particular length
  - We have three logically-different segments: code, stack, heap
  - The segmentation allows the OS to place each segment in different part of physical memory

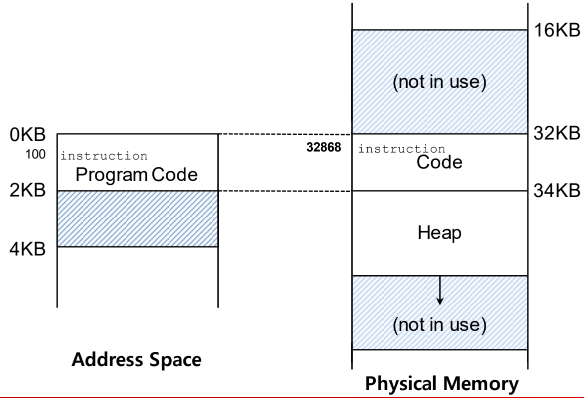
Segmentation

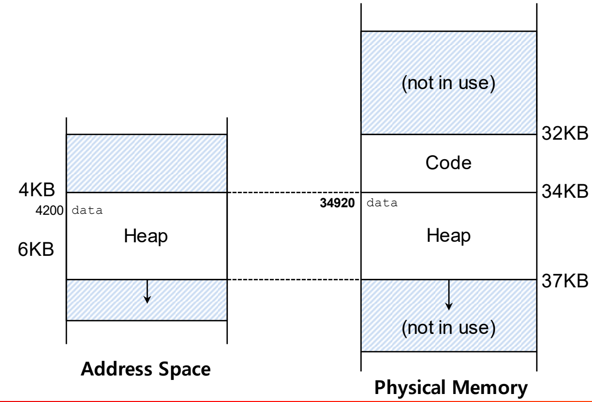
- **Let's look at an example**
  - 64KB physical memory with three segments in it
  - Only used memory is allocated space in physical
  - MMU has three base and bounds register pairs

Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K



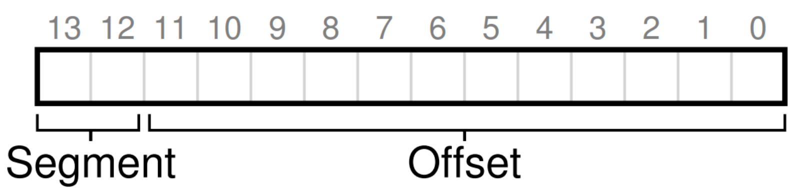
- **Address translation examples**
  - A reference is made to virtual address 100 (code segment): 32KB(32768) + 100 = 32868 and the address is within bounds
  - Virtual address 4200 (heap segment): heap starts at 4 KB (4096) and offset is 4200 – 4096 = 104; thus 34KB(34816) + 104 = 34920 (within bounds)
  - OS occurs segmentation fault if an illegal address (e.g. >7KB; beyond the end of heap) is reference; the hardware detects that this address is out of bounds

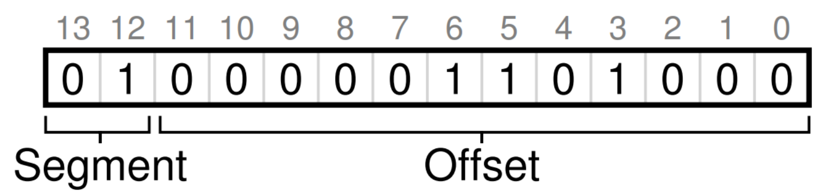




Which Segment Are We Referring To?

- **The hardware uses segment registers during translation**
  - How does it know the offset into a segment, and to which segment an address refers?
- **One common explicit approach is to chop up the address space into segments based on the top few bits of the virtual address**





  - 00: code segment, 01: heap segment, 11: stack segment, 10: unused
  - Virtual address 4200 is encoded: **01_0000_0110_1000** (heap seg. + offset 104)

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

- **SEG_MASK 0x3000**
- **SEG_SHIFT 12**
- **OFFSET_MASK 0xFFF**

  - It limits use of virtual address space (2^{14}:16KB→2^{12}:4KB) ➔ implicit approach

What About The Stack?

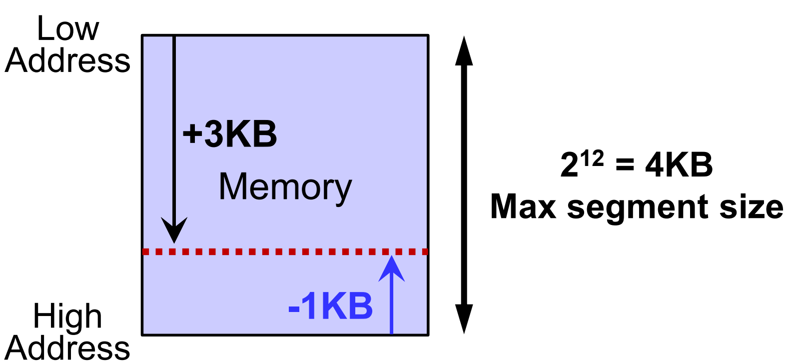
- **Stack has one critical difference: it grows backwards to lower**
  - It starts at 28KB and grows back to 26KB, virtual addresses 16KB to 14KB
- **We need a little extra hardware support**
  - The hardware needs to know which way the segment grows and translates differently
  - 1: positive direction, 0: negative direction

Segment	Base	Size (max 4K)	Grows Positive?
$Code_{00}$	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

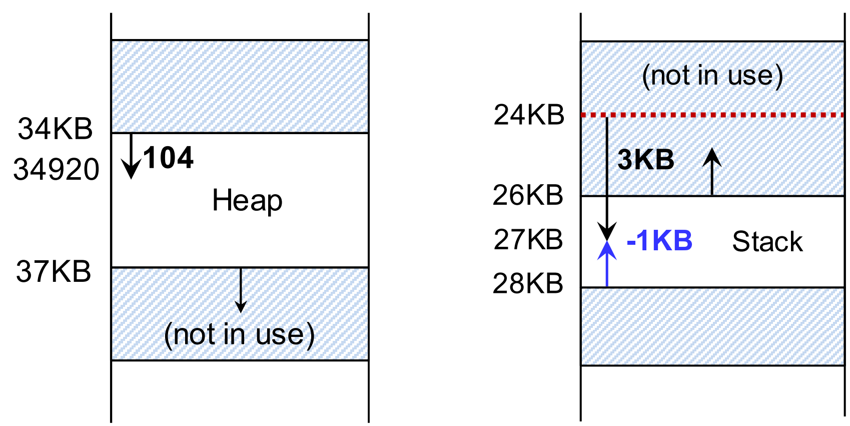
x

실제 3KB를 더하면 seg fault!

- **Example of virtual address 15KB**
  - **11_1100_0000_0000** (offset 3KB) ➔ 3KB-4KB=-1KB is actual offset for stack ➔ 28KB – 1KB = 27KB is the physical address (within bound, abs(-1KB)<2KB)



Offset in 12bit (4KB) space:
Forward 3KB == Backward 1KB



Positive direction

Negative direction

Address Translation for Stack

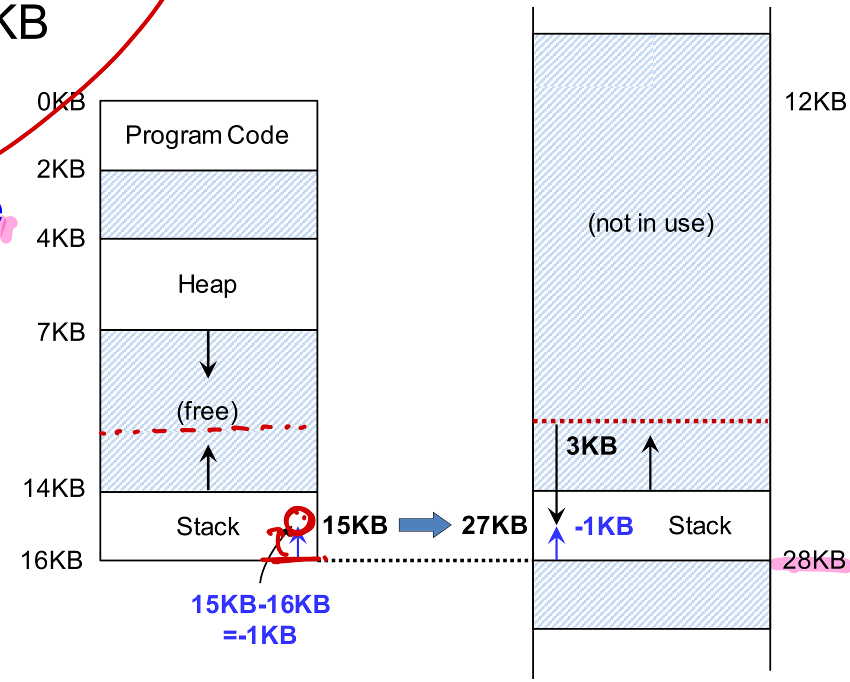
stack segment =12bit, 4KB

3KB

- **Example of virtual address 15KB (11_1100_0000_0000)**
  - To obtain an actual offset for backward growing stack, the virtual address needs to be subtracted by virtual space size (or **stack's virtual address**)
  - Thus, offset = 15KB - 16KB = -1KB

offset (stack) = virtual address – virtual space size

  - Then, physical address = 28KB-1KB=27KB
  - Similarly, this can be obtained by offset (stack) = offset (from address) - max segment size
  - Thus, offset = 3KB - 4KB = -1KB
- **Still difficult?**
  - You take 2's complement of the offset from address and attach minus sign
  - e.g.) 1100_0000_0000 (3KB) ➔ 0100_0000_0000 (1KB) ➔ -1KB



Support for Sharing

→segment를 공유.

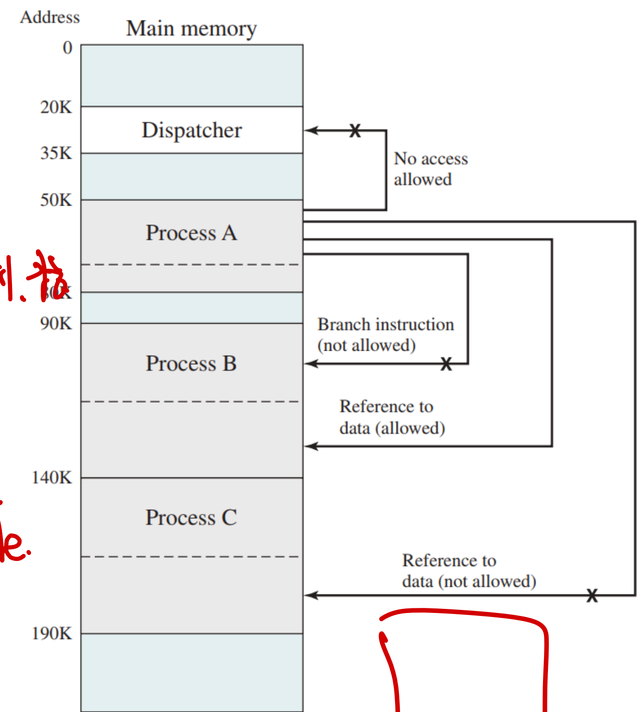
A(1) code ↔ code A(2) 동일. ... code segment를 sharing 하고 싶다.

- **It is useful to share certain segments between address spaces**
  - Code sharing is common and still in use in systems today
  - We need a little extra hardware support: Protection bits per segment to indicate permission of read, write, and execute →플래그 표시. 가
  - Code sharing is common and still in use in systems today

Segment	Base	Size (max 4K)	Grows Positive?	Protection
$Code_{00}$	32K	2K	1	Read-Execute
$Heap_{01}$	34K	3K	1	Read-Write
$Stack_{11}$	28K	2K	0	Read-Write

=segment table.

  - The hardware also has to check whether a particular access is permissible or not



- **Fine-grained and coarse-grained segmentation**
  - Coarse-grained segmentation: a few relatively large, coarse segments
  - Fine-grained segmentation has a large number of smaller segments
  - Supporting many segments requires further hardware support: segment table

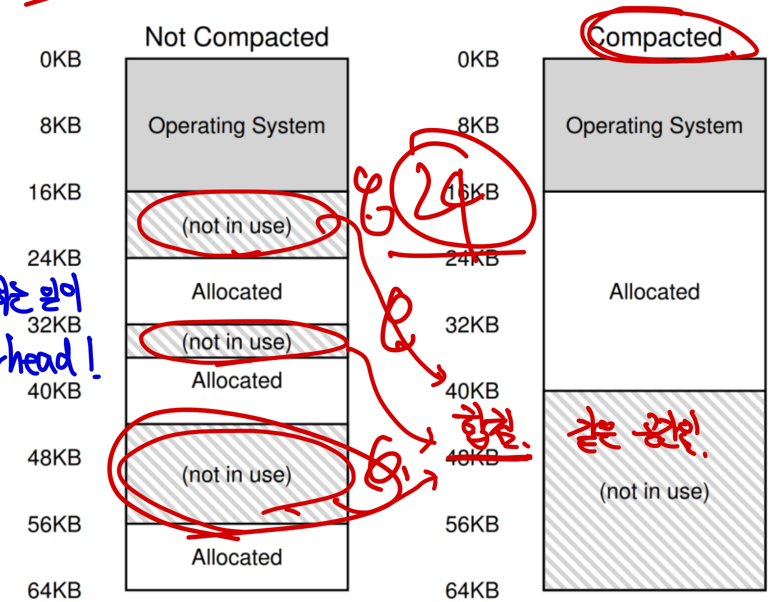
OS Support

coalescing

segment register.

- **Segmentation raises a number of new issues for OS**
  - What should the OS do on a context switch?
    ➔ The segment registers must be saved and restored
  - OS interaction when segments grows (or perhaps shrink)
    e.g.) 1) the heap segment need to grow → 2) the memory-allocation library will perform a system call to grow the heap (e.g. `sbrk()`) → 3) OS provides more space, updating segment size register to the new bigger size
- **The critical issue is managing free space in physical memory** (단편화)
  - External Fragmentation (외부 단편화.): little holes of free space in physical memory that is too small to allocate new segment or grow existing
  - Solution is compaction (압축): rearranging the exiting segments in physical memory:
    1) stop running process
    2) copy data to somewhere in memory
    3) change segment register value

    해야 하는 일이 많음. overhead!
  - New efficient allocation algorithms: best-fit, worst-fit, first-fit, buddy algorithm, etc



Summary

- **Segmentation utilizes a base and bounds register per segment**
  - MMU maintains multiple base and bound register pairs, which will be placed in the memory when the segment table is large