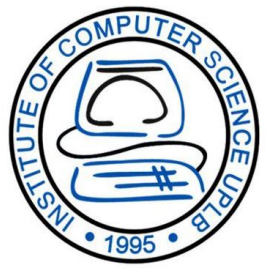
# CMSC 125: Operating Systems


- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org



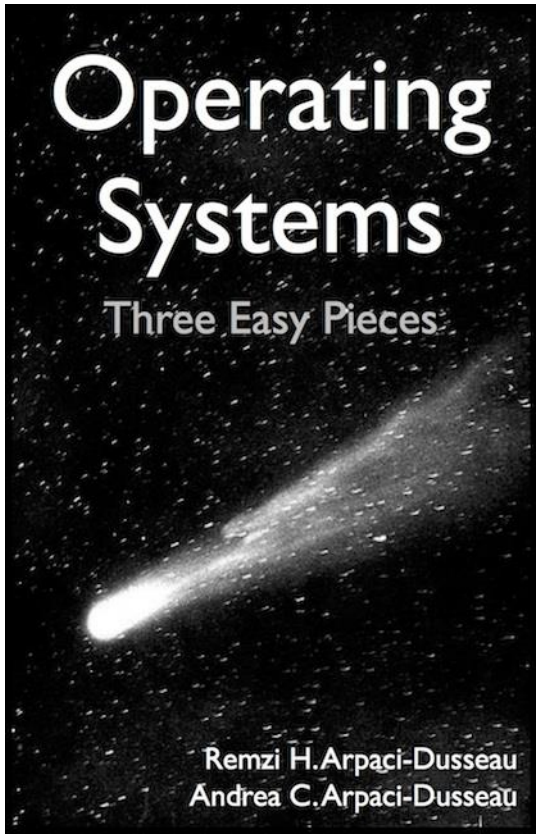




# Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



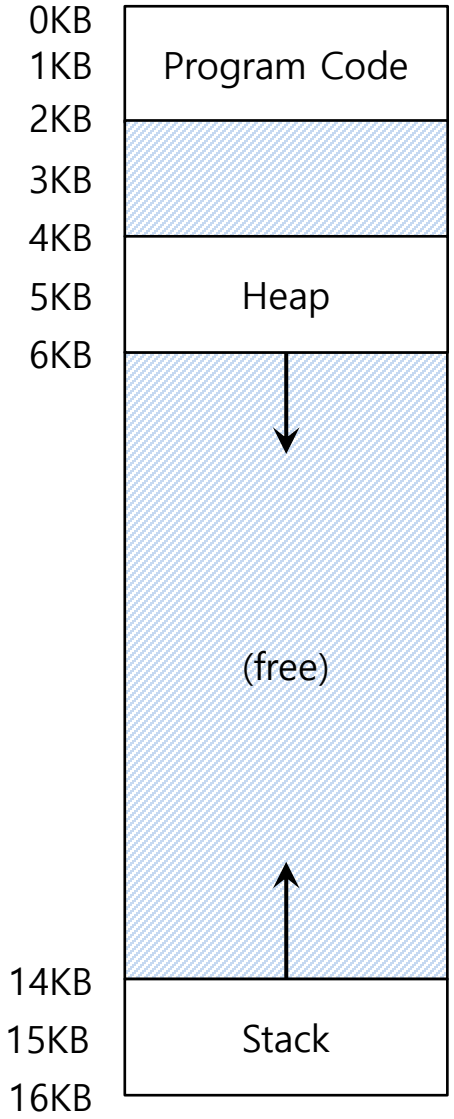
# Acknowledgement


- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

# 16. Segmentation

## Operating System: Three Easy Pieces

# Inefficiency of the Base and Bounds Registers Approach

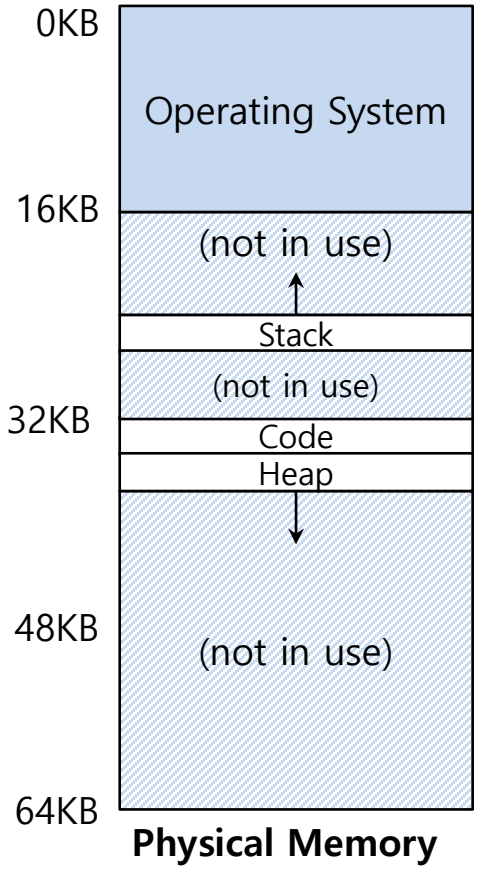



- **Big chunk of "free"** space
- "free" space **takes up** physical memory.
- Hard to run when an address space **does not fit** into physical memory

# Segmentation: Generalized Base and Bounds

- Motivation: Why not have a base and bounds registers for each **logical section** in a process' address space?
- A **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** registers exist **per segment**

# Placing Segments In Physical Memory



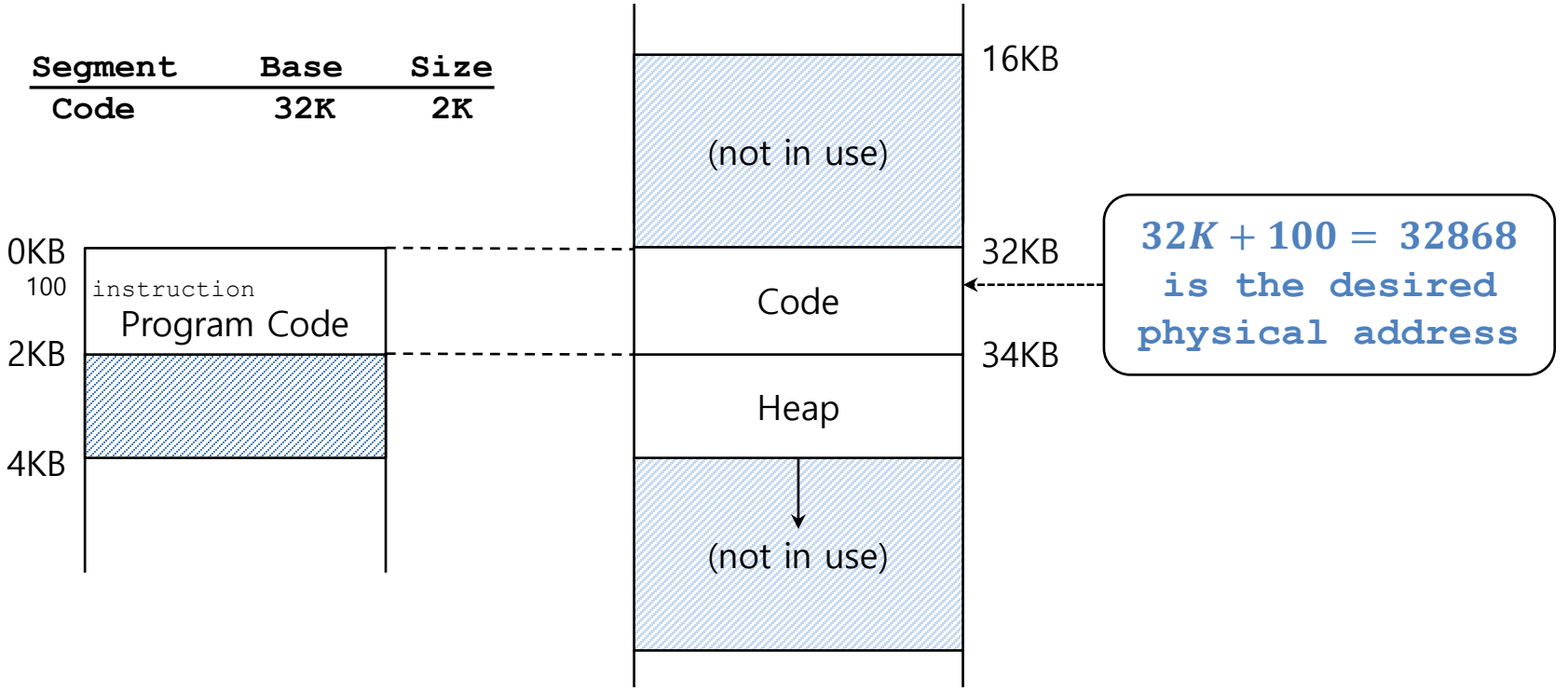

| Segment | Base | Size |
|---|---|---|
| Code | 32K | 2K |
| Heap | 34K | 2K |
| Stack | 28K | 2K |

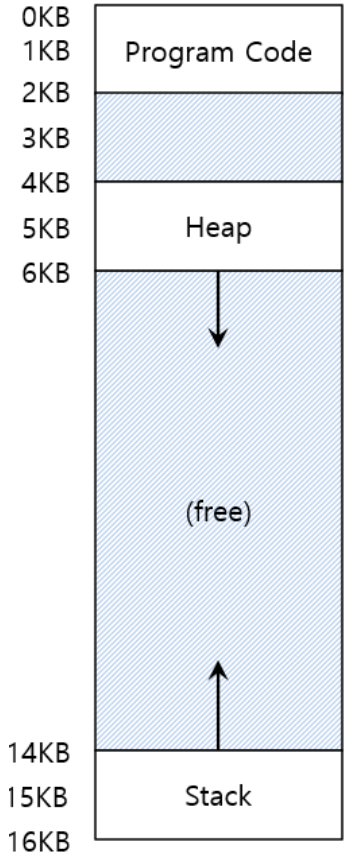
# Segmentation: Address Translation Example

- Assume a reference to virtual address 100 (in code section)
  - Since code section starts at 0 in address space, offset is 100

$$physical\ address = base + offset$$

| Segment | Base | Size |
|---|---|---|
| Code | 32K | 2K |

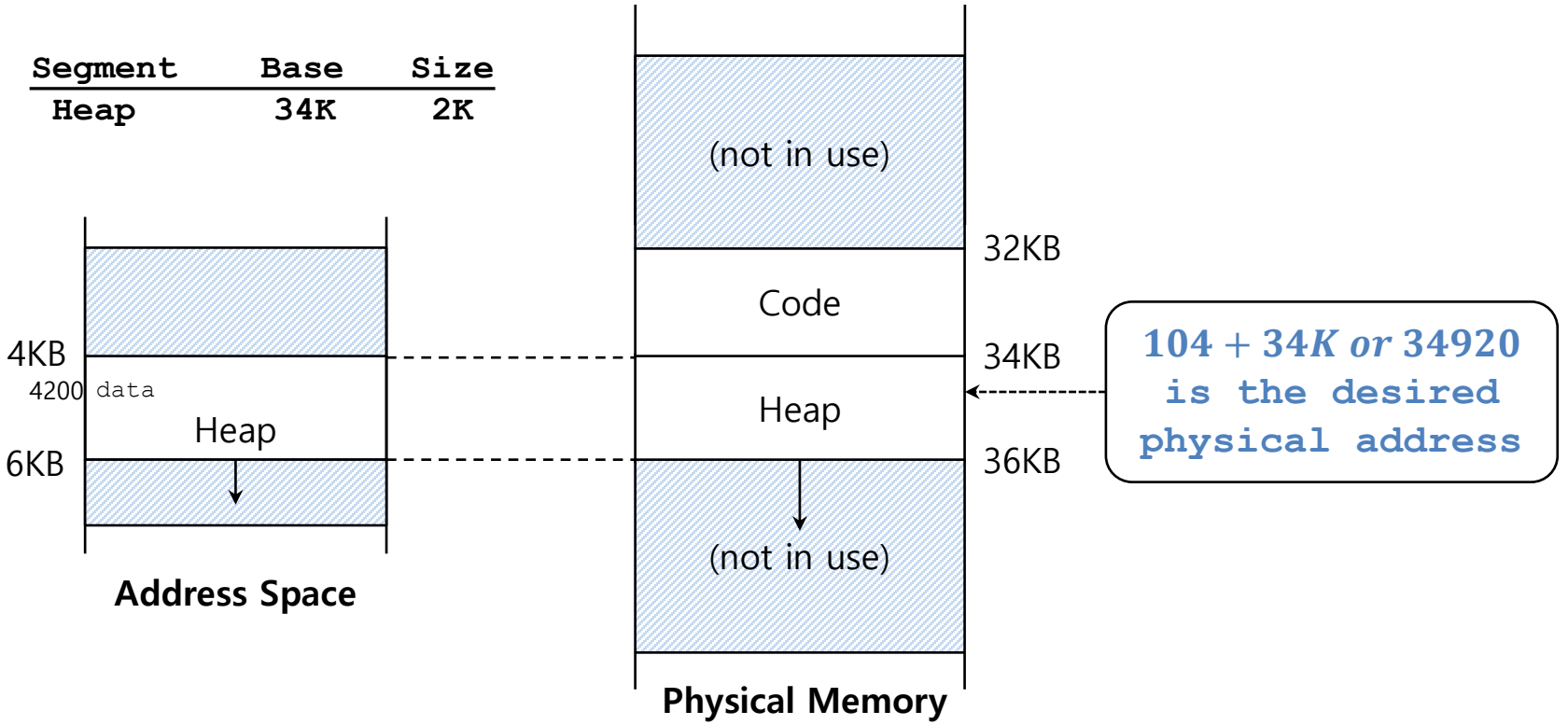


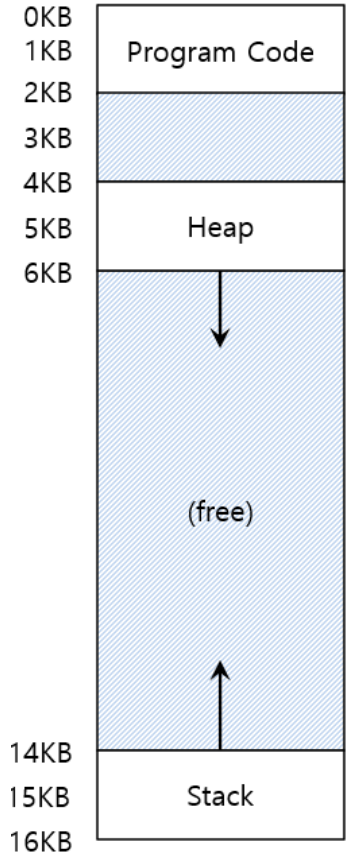



# Segmentation: Address Translation Example (Cont.)

*Virtual address + base* is not the correct physical address.

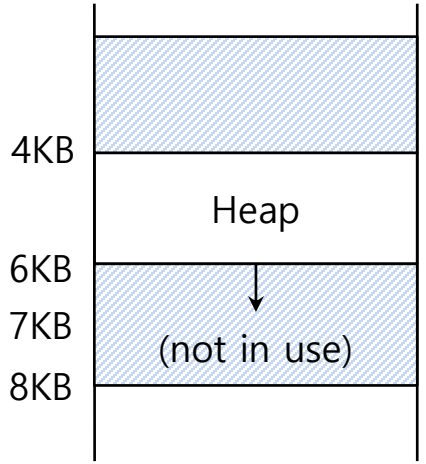
- Assume a reference to virtual address 4200 (in head section)
  - The heap section **starts at virtual address 4096(4KB)** in address space, offset=4200-4096=104



# Segmentation Fault or Segmentation Violation

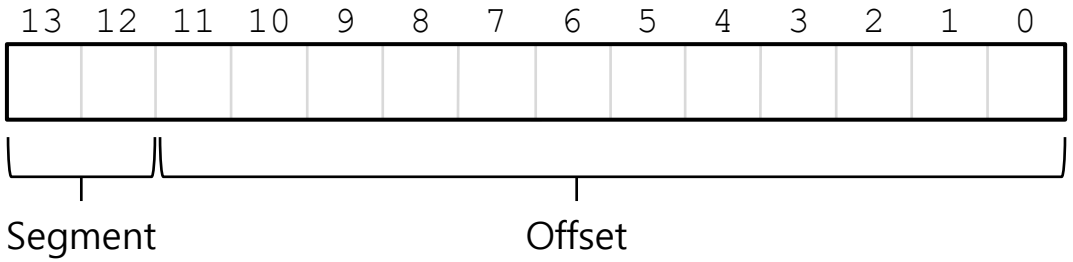
- If an **illegal address,** such as 7KB, which is beyond the end of heap is referenced, the OS generates **segmentation fault**.
  - The hardware detects that address is **out of bounds**.




**Address Space**

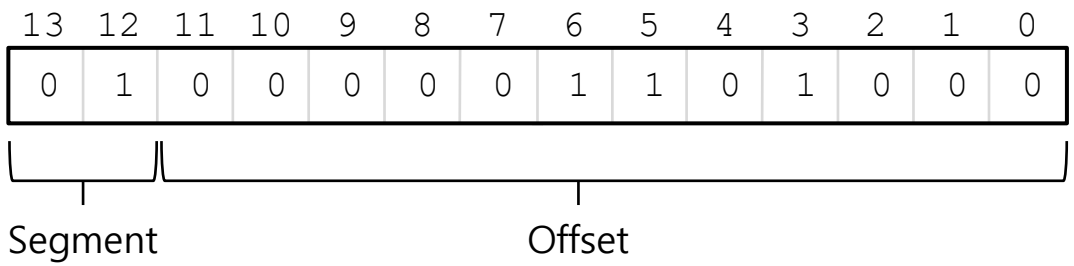
# Referring to a Segment(Cont.)

- Given just a virtual address, how does the hardware determine the segment and offset?
- **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 a 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)
```

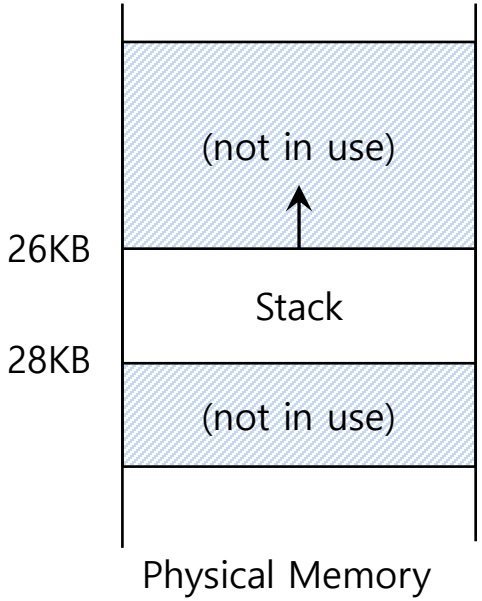
- SEG_MASK = 0x3000(110000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (00111111111111)

- Disadvantages of explicit approach using top 2 bits
  - If only 3 sections, then 2 bits to store segment is wasteful
  - Limits a segment size, ex. max segment size is 4KB

- **Implicit approach** – determines segment based on how the virtual address was generated, ex. if from program counter, then code segment

# Referring to Stack Segment

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

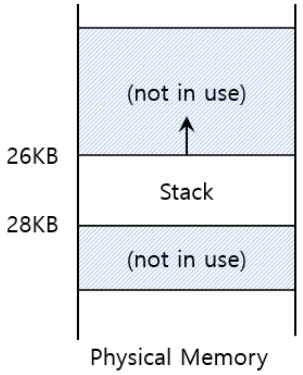



Segment Register(with Negative-Growth Support)

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

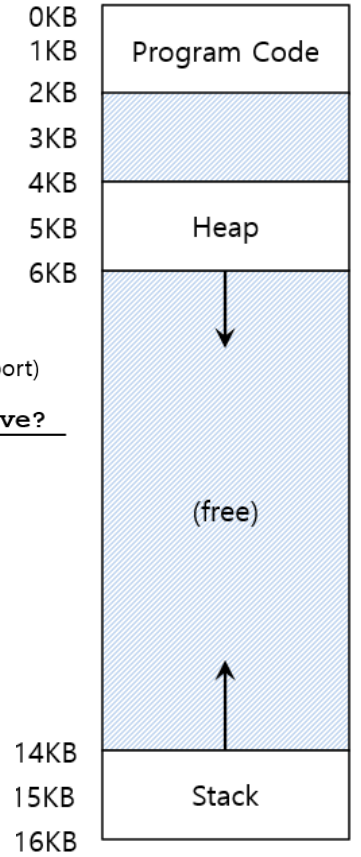
# Referring to Stack Segment(Cont.)

- Example: reference to virtual address 15KB = 11 1100 0000 0000 = 0x3C00
- Segment=0x3(3), Offset=0xC00 (3KB)
- Negative Offset = Offset - Max Segment Size = 3KB - 4KB = -1KB
- Physical Address = Negative Offset + Stack Base
- Physical Address = -1KB + 28KB = 27KB




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: **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

- **Coarse-Grained** means segmentation in a small number
  - e.g., code, heap, stack
- **Fine-Grained** segmentation allows **more flexibility** for address space in some early system
  - To support many segments, hardware support with a **segment table** is required

# OS Support for Segmentation

- What should the OS do during context switch?
- What should the OS do when a segment grows?
- How should the free space in memory be managed?
  - There are variable segment sizes

# OS Support for Segmentation: Fragmentation

- **External Fragmentation**: little holes of **free space** in physical memory that make it difficulty to allocate new segments
  - There is **24KB free**, but **not in one contiguous** segment
  - The OS **cannot** satisfy a **20KB request**
- **Compaction**: **rearranging** the existing segments in physical memory
  - Compaction is **costly**
    - **Stop** running process.
    - **Copy** data to somewhere
    - **Change** segment register value
- Better to use a **free-list management algorithm**
  - Keep large extends of memory available for allocation
  - Approaches: best-fit, worst-fit, buddy-algorithm

# Memory Compaction