Operating System Design and Implementation

Lecture 10: Segmentation

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Virtual memory address
  - Address space
  - Static relocation
  - Dynamic relocation
- Segmentation
  - Base and bounds

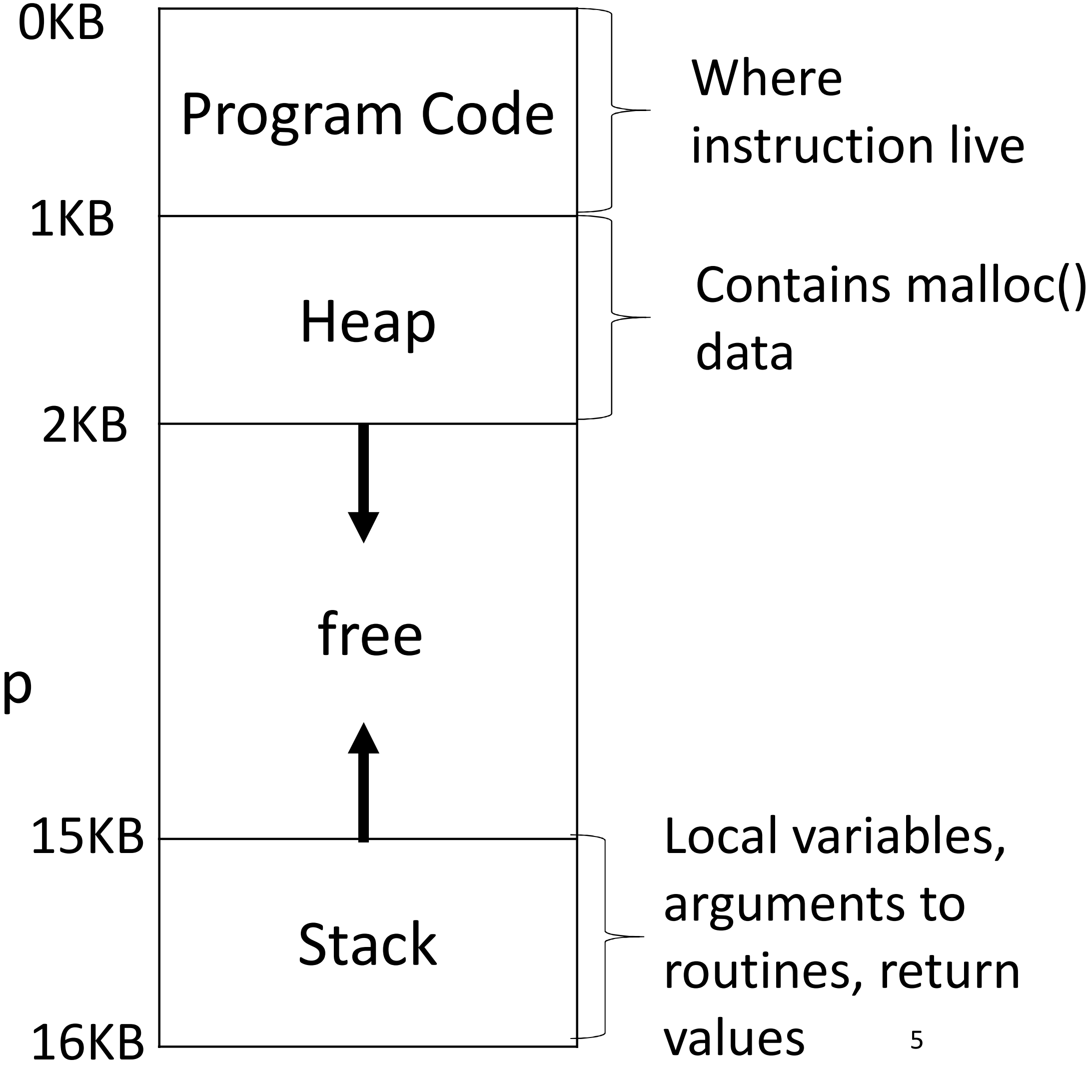
Memory

- Program must be brought into memory and run
- CPU only can direct access **main memory** and **registers**
  - Register access in one CPU clock (or less)
  - Main memory can take hundreds of cycles
  - Cache sits between main memory and CPU registers
- **Protection of memory** is required to ensure correct operation
  - Isolation: kernel/user space, processes
  - We don’t want process to be able to read/write other one’s memory

Address space

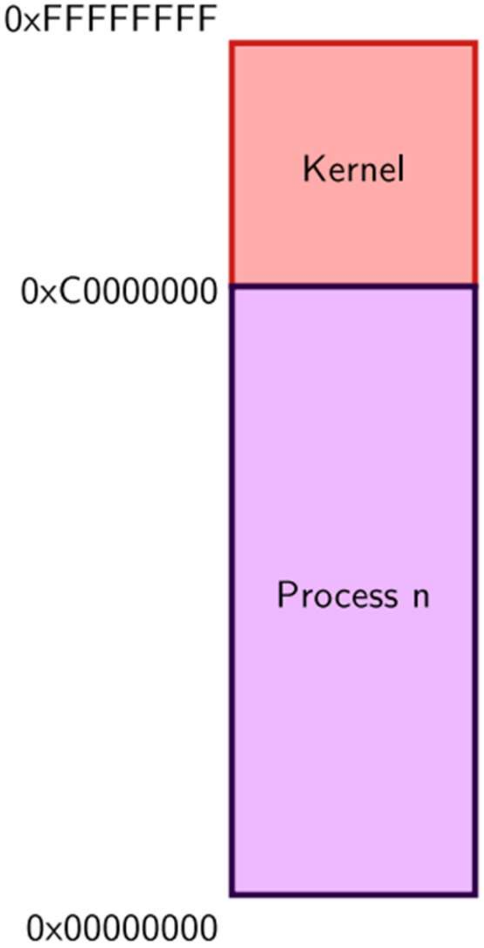
- **Address space**
  - The running program's view of memory in the system
  - For example, the address of a process contains all of memory state of the running program
  - This placement of stack and heap is just a convention
    - Can be arrange in a different way

Process address space



Address space (cont.)

- **In 32-bit virtual address**
  - 1GB reserved for kernel-space
    - Contains kernel code and core data
    - Most memory can be a direct mapping of physical memory at a fixed offset
  - Complete 3GB exclusive mapping available for each user space process
    - Process code and data (program, stack, …)
    - Memory-mapped files
    - Page tables



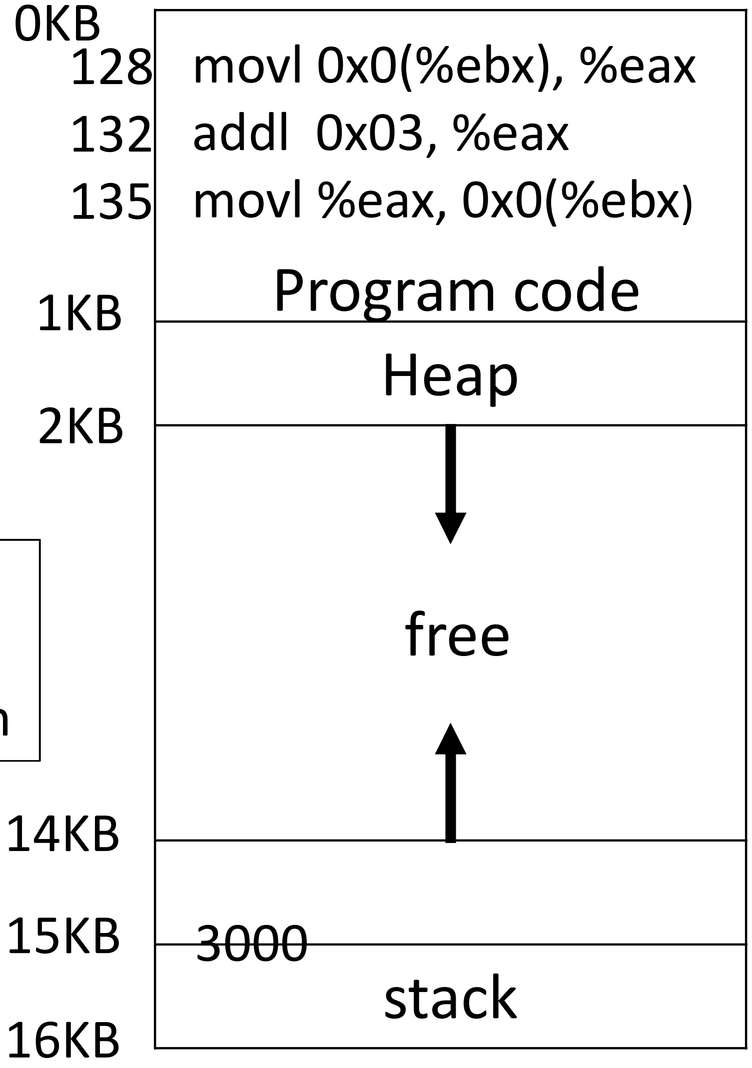
https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

A process address space

```
void func() {
        int x = 3000;
        x = x + 3;
}
```

```
128: movl       0x0(%ebx), %eax   ; load 0+ebx into eax
132: addl       $0x03, %eax        ; add 3 to eax reg
135: movl       %eax, 0x0(%ebx)    ; store eax back to mem
```

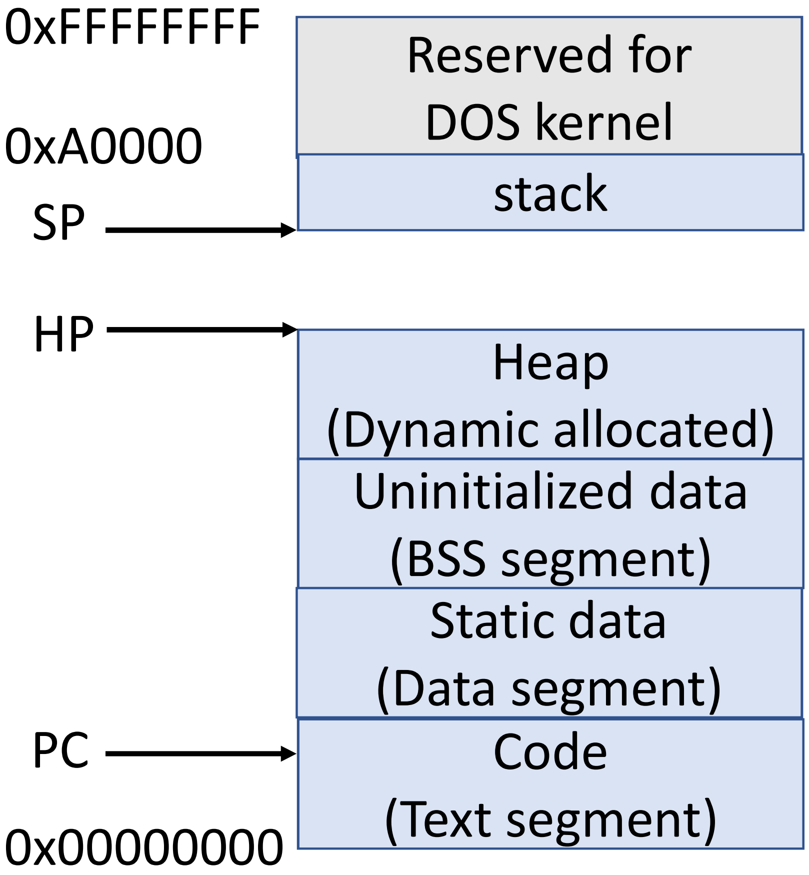
Process address space



0KB
128 movl 0x0(%ebx), %eax
132 addl 0x03, %eax
135 movl %eax, 0x0(%ebx)
Program code
1KB
Heap
2KB
free
14KB
15KB
3000
stack
16KB

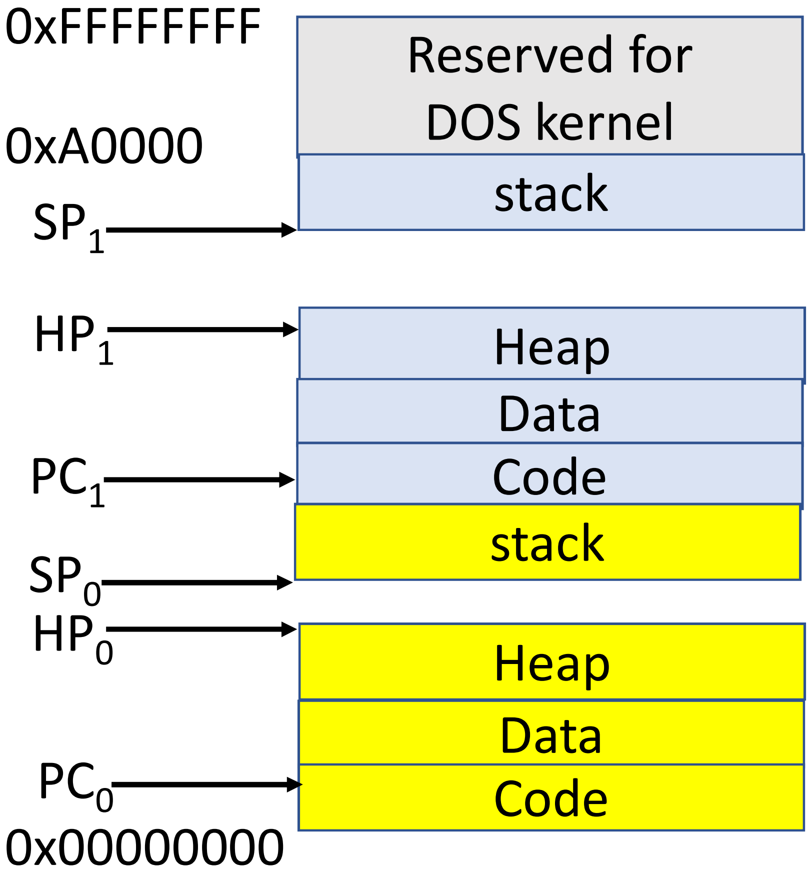
Uni-programming (e.g. DOS)

- **One process at a time**
- User code compiled to sit in fixed range (e.g. [0, 640 KB])
  - No hardware virtualization of addresses
- **OS in separate addresses**
  - E.g. above 640 KB
- **Goals**
  - Safety: None
  - Efficiency: Poor (I/O and compute not overlapped, response time)



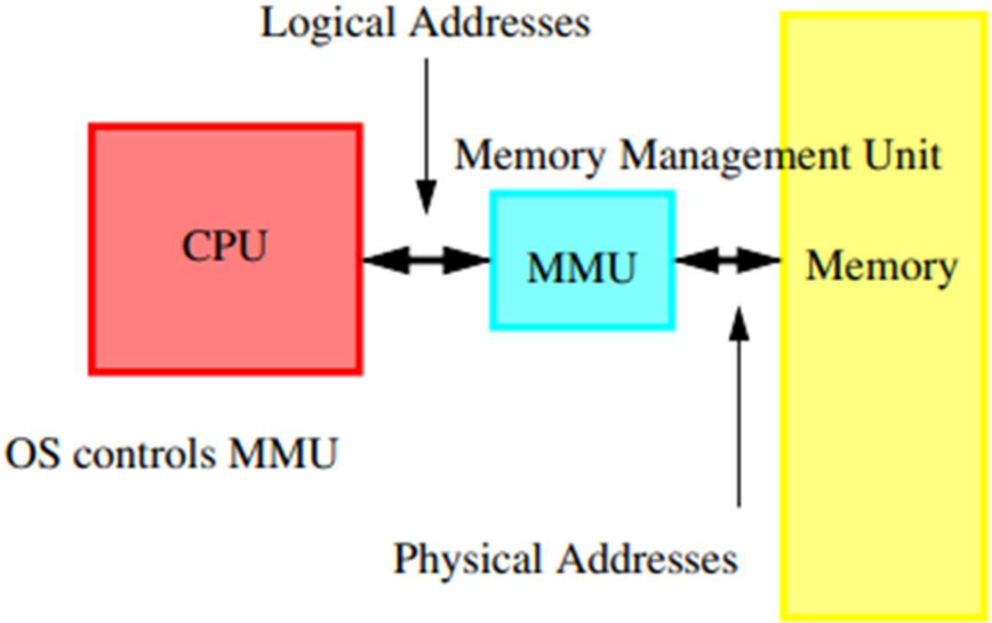
Multi-programming: static relocation

- **Moving data or codes to absolute locations before a program is run**
  - Modify addresses statically (similar to linker)
  - Processes can run anywhere in memory (can't predict in advance)
- **Advantages**
  - Allows multiple processes to run
  - Require no hardware support
- **Problems**
  - Creating contiguous holes
  - Process may not be able to increase address space



Dynamic relocation

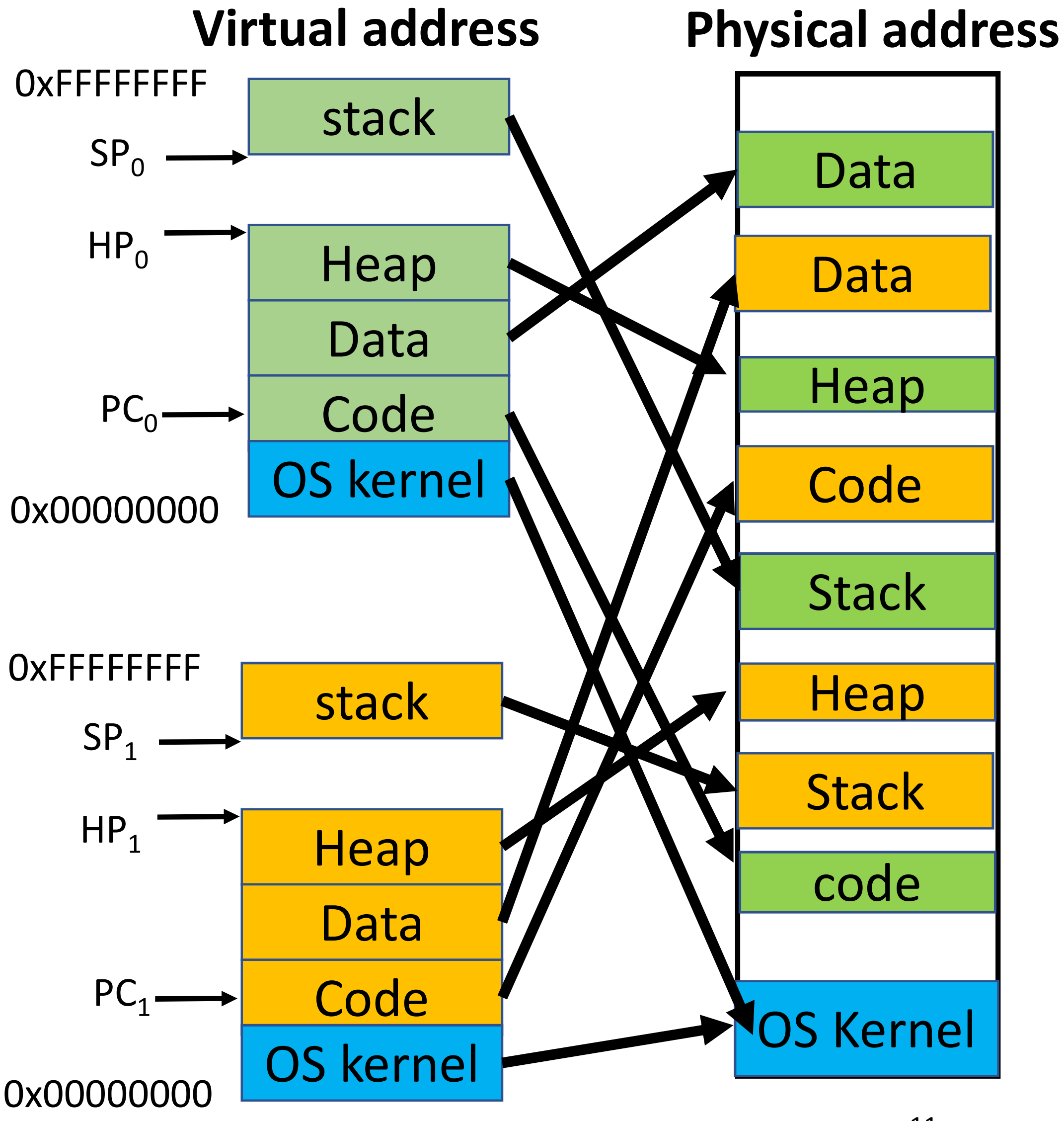
- Change address dynamically at every reference
  - Program-generated address translated to hardware address
  - Program addresses are called **virtual** addresses
  - Hardware addresses are called **physical** addresses
  - Address space: view of memory for each process



https://pages.cs.wisc.edu/~eli/537/lectures/lecture13.2x2.pdf

Dynamic relocation

- **Idea**
  - Programs all laid out the same
  - Relocate addresses when used
  - Requires hardware support
- **Two views of memory**
  - Virtual: Process's view
  - Physical: Machine's view
- **Variants**
  - Base and bounds
  - Segmentation
  - Paging

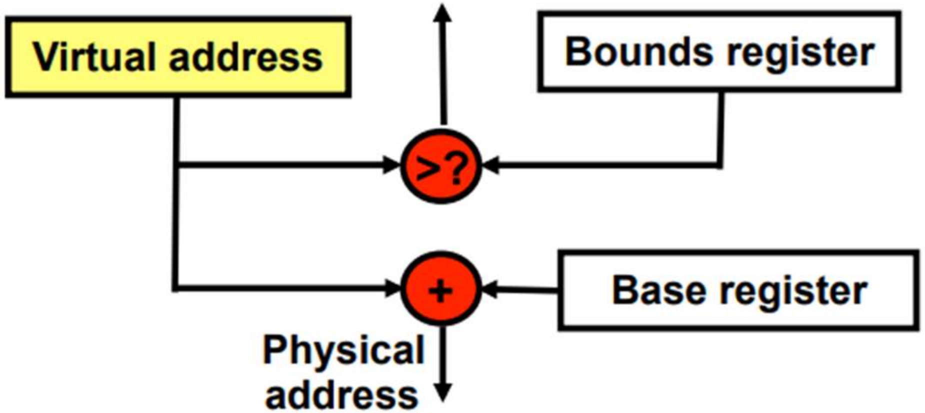


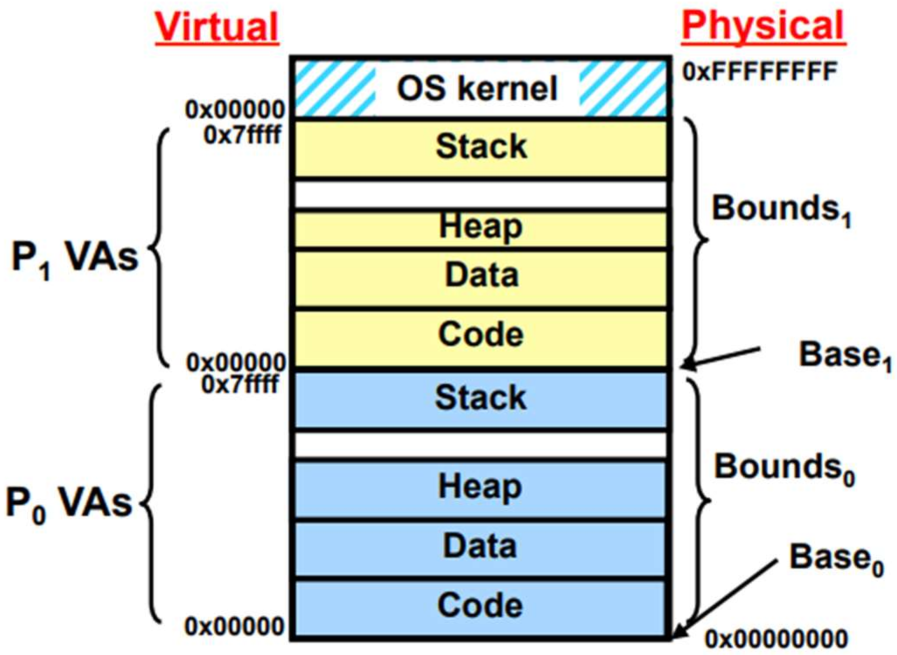
Dynamic relocation

- **Virtual address**
  - Each memory reference generated by the process
- **Base and bound (Dynamic relocation)**
  - **Base register** is use to transform virtual address into physical address
  - **Limit register** ensures such addresses are within the confines of the address space
  - **Efficient:** only a little hardware logic is required
  - **Protection:** no process can generate memory references outside its own address space
- **Memory translation**
  - Transforming a virtual address into a physical address
  - **Physical address = virtual address + base**

Base and bounds

- Each process mapped to contiguous physical region
  - Each process sees a private and uniform address space (0 … max)
  - Everything belonging to a process must fit in that region
- **Two hardware registers**
  - **Base**: starting physical address
  - **Bounds**: Size in bytes
- On each reference
  - Check against bounds
  - Add base to get physical address





https://my.eng.utah.edu/~cs5460/slides/Lecture13.pdf

Base and bounds (cont.)

- **Each process has private address space**
  - No relocation done at load time
- **Operating system handled specially**
  - Only OS can modify base and bound registers

Pros and cons of base and bounds

- **Advantages**
  - Support dynamic relocation of address spaces
  - Support protection across multiple address spaces
  - Cheap: few registers and little logic
  - Fast: Add and compare can be done in parallel
- **Disadvantages**
  - Each process must be allocated contiguously in real memory
    - Fragmentation: Cannot allocate a new process
  - Must allocate memory that may not be used
  - No sharing: Cannot share limited parts of address space
    (e.g. cannot share code with private data)

https://pages.cs.wisc.edu/~eli/537/lectures/lecture13.2x2.pdf

Memory translation

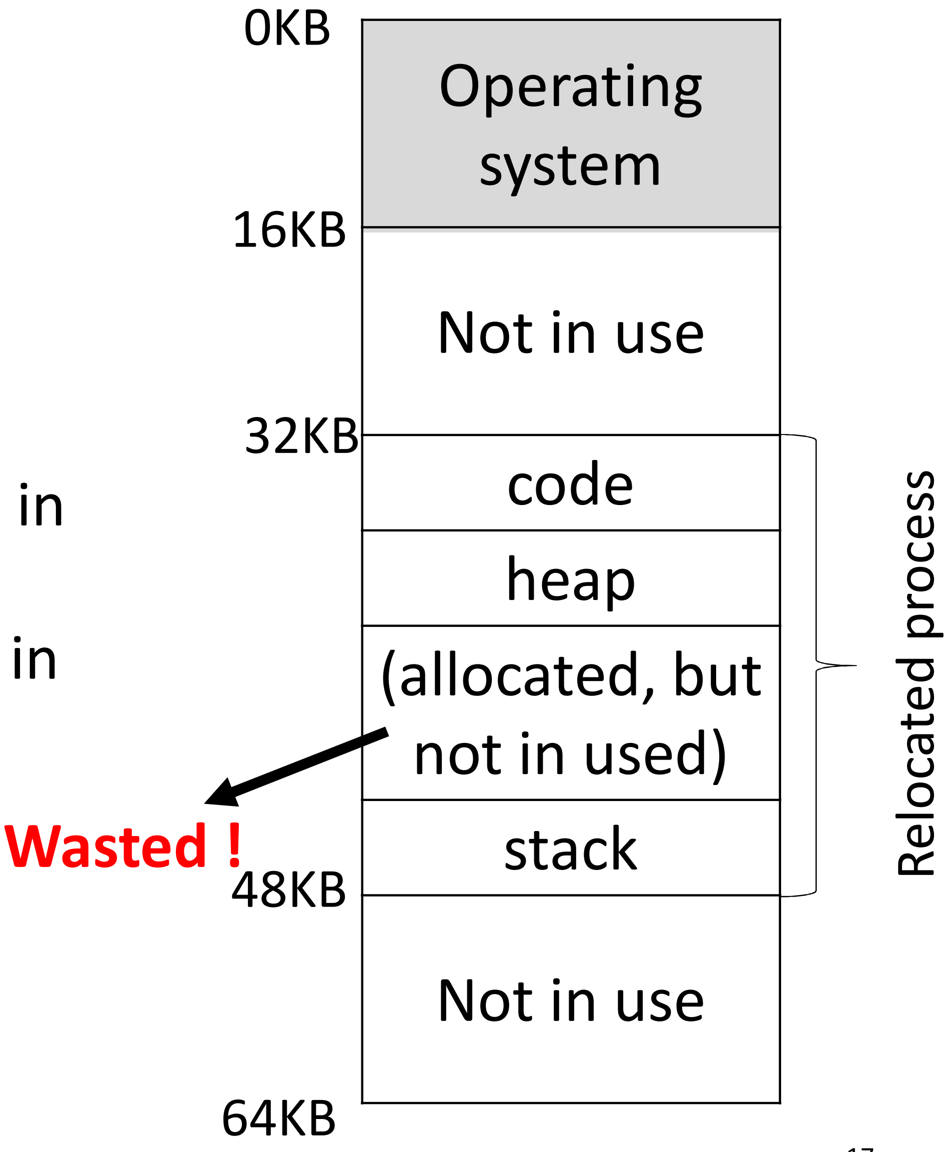
```
128: movl 0x0(%ebx), %eax
```

- How does dynamic relocation work?
    - The program counter (PC) is set to 128
    - Adds the value of PC to the base register value of 32KB to get a physical address 32896 (=32768 + 128)
- **Bound register**
    - Help with protection
    - Check the memory reference is within bounds to make sure it’s legal
    - CPU raise an exception when a process generates a virtual address that is great than the bounds

Virtual address		Physical address
0	→	16KB
1KB	→	17KB
3000	→	19384
4400	→	Fault (out of bounds)

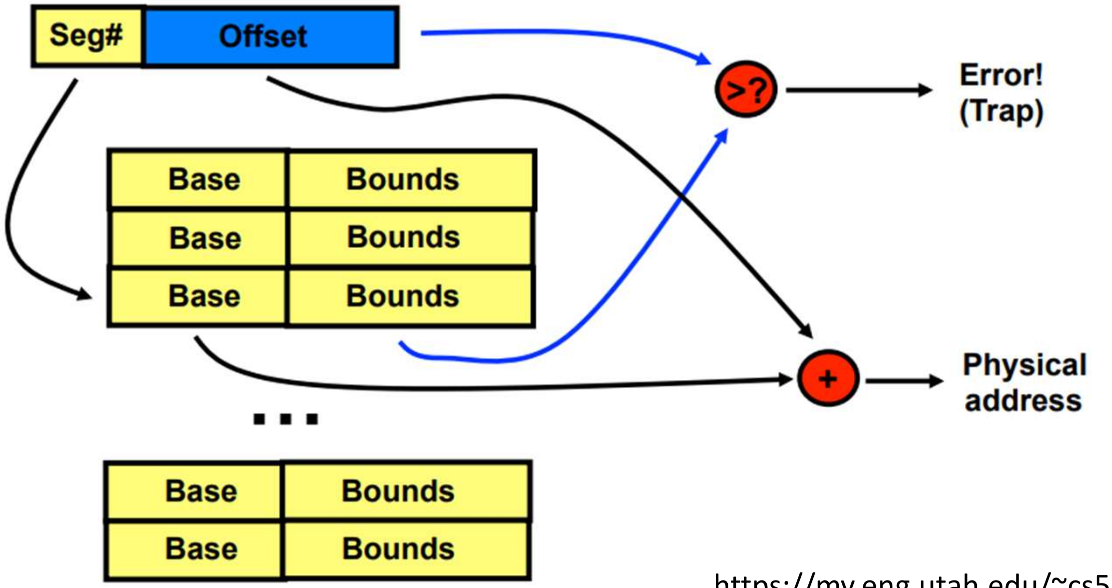
Fragmentation

- Problems of dynamic relocation
  - **Fragmentation**
  - E.g. internal fragmentation shown in the relocated process
  - Restrict to place an address space in a fixed-sized slot
- How to avoid internal fragmentation ?
  - Segmentation



Segmentation

- **Idea: Create N separate segments**
  - Each segment has separate base and bounds register
  - Segment number is fixed portion of virtual address



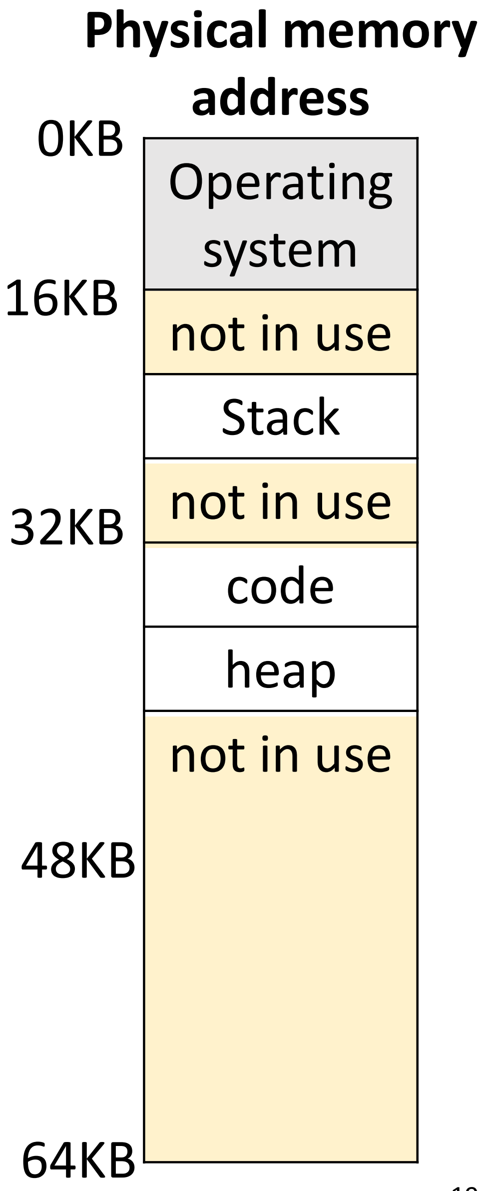
https://my.eng.utah.edu/~cs5460/slides/Lecture13.pdf

Segmentation (cont.)

- Why not have a base and bounds pair per logical segment of the address space ?
- A segment
    - A contiguous portion of the address space of a particular length

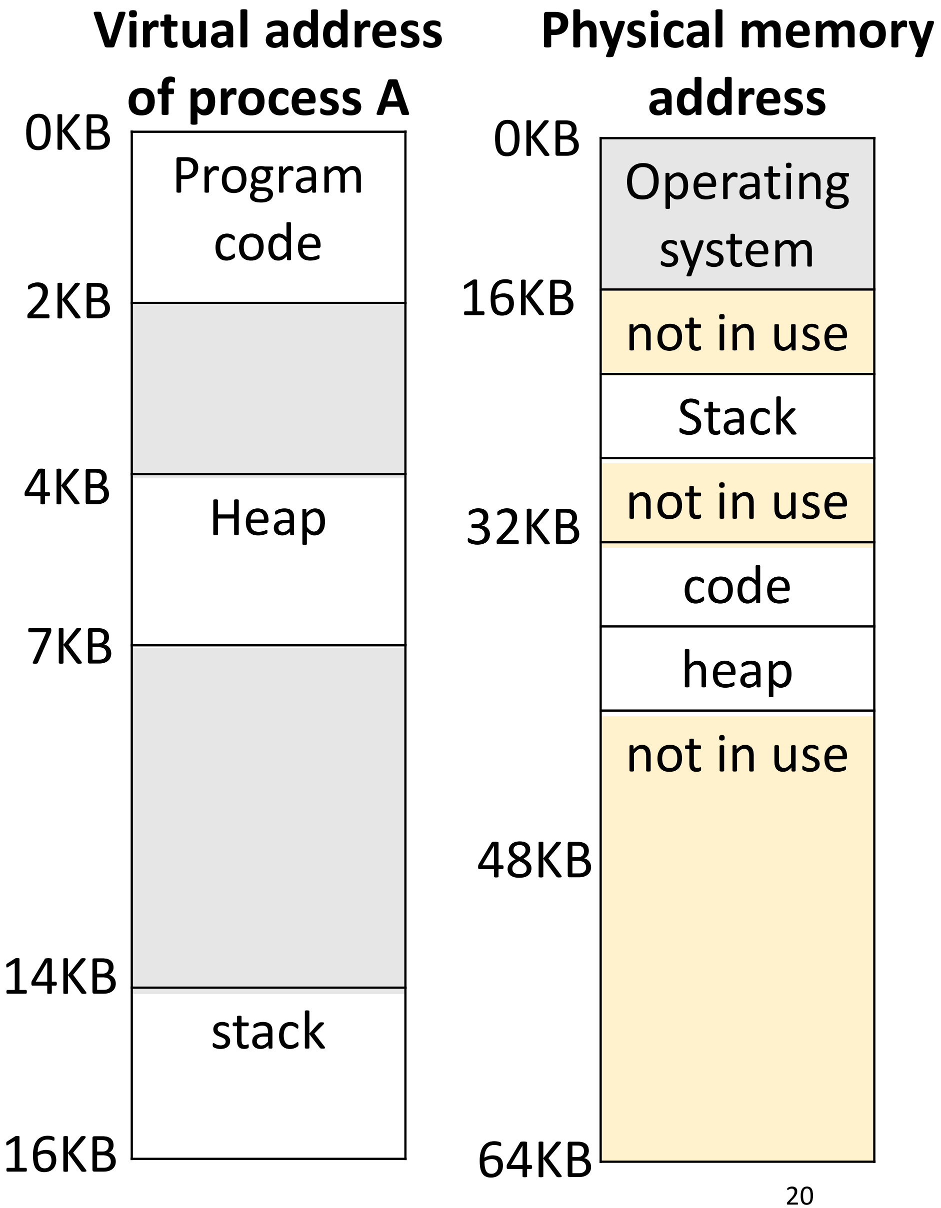
Segment register value

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


Physical memory address

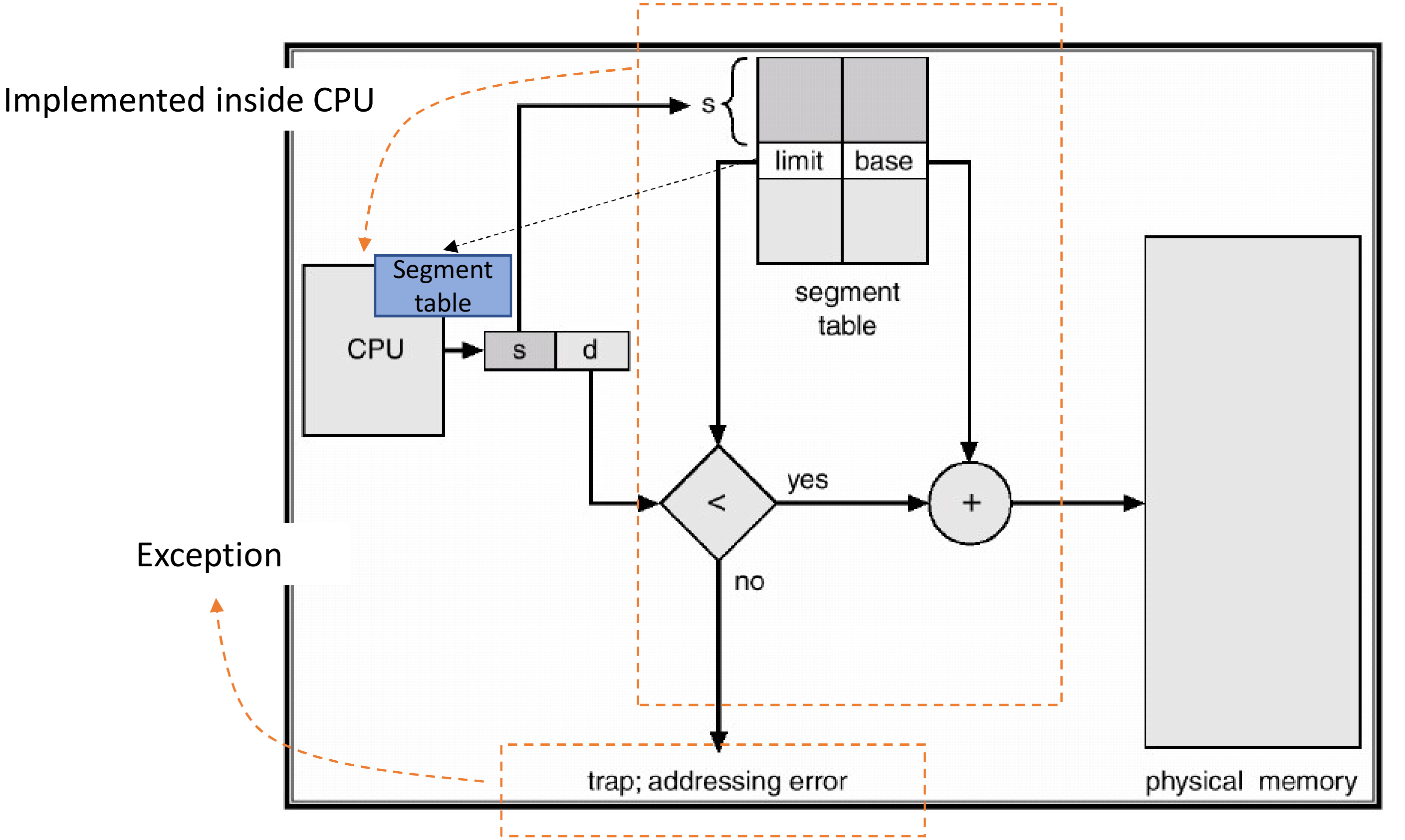
Segmentation (cont.)

- Assume a memory reference is made by an instruction to virtual address 100
  - Desired physical: 100 + 32 KB
  - Check the address is within bounds (100 less than 2KB)
- Assume a heap virtual address 4200
  - Get physical address 4200 + 34KB = 39016 that is a incorrect physical address
  - The heap start at virtual address 4KB
  - The **offset** 4200 is 104 (4200 - 4096)
  - Add offset with base register physical address (34K) to get 34920



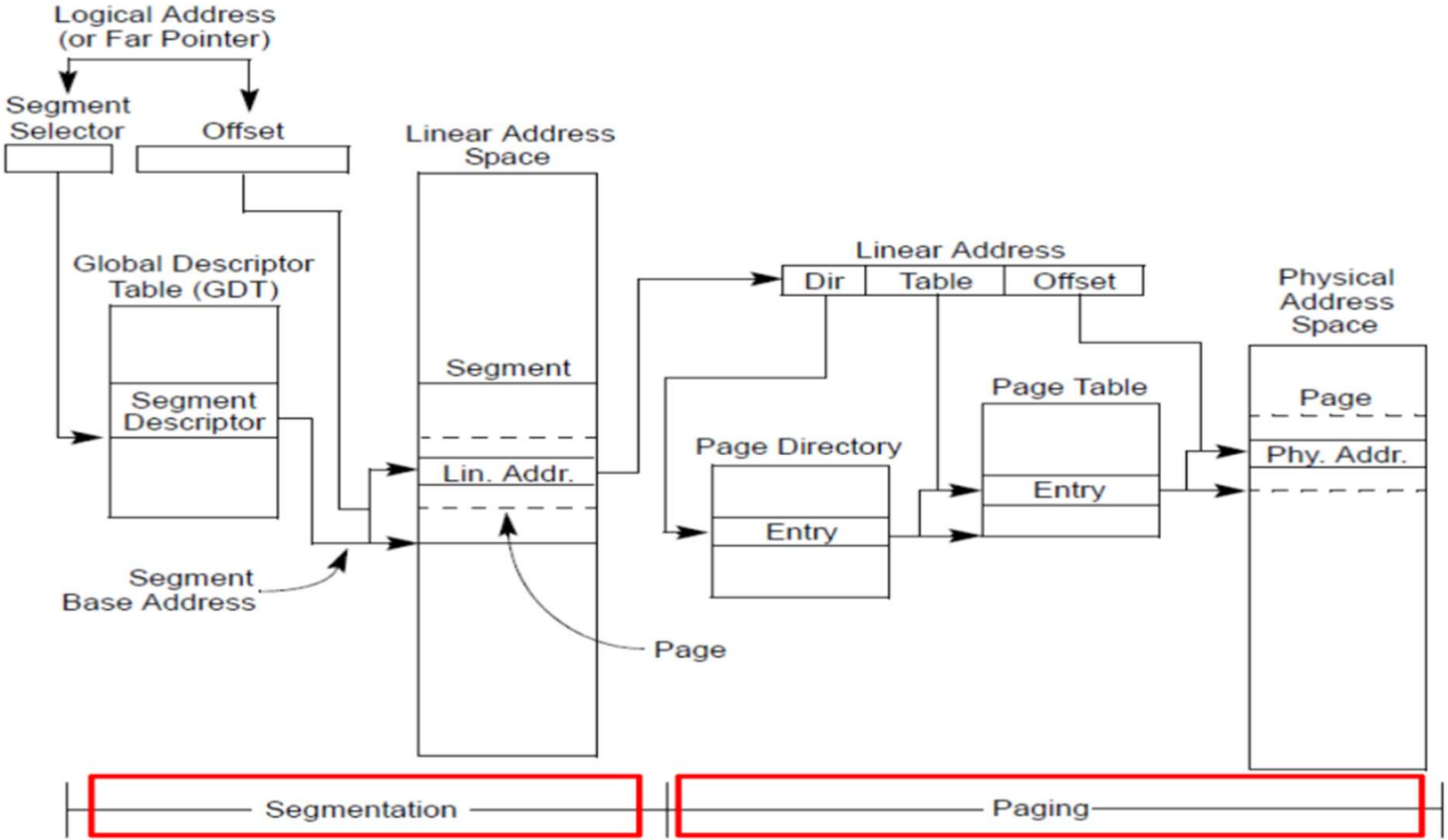
Segmentation Memory Management

Memory management unit (MMU) or memory protection unit (MPU) which is hardware does the check



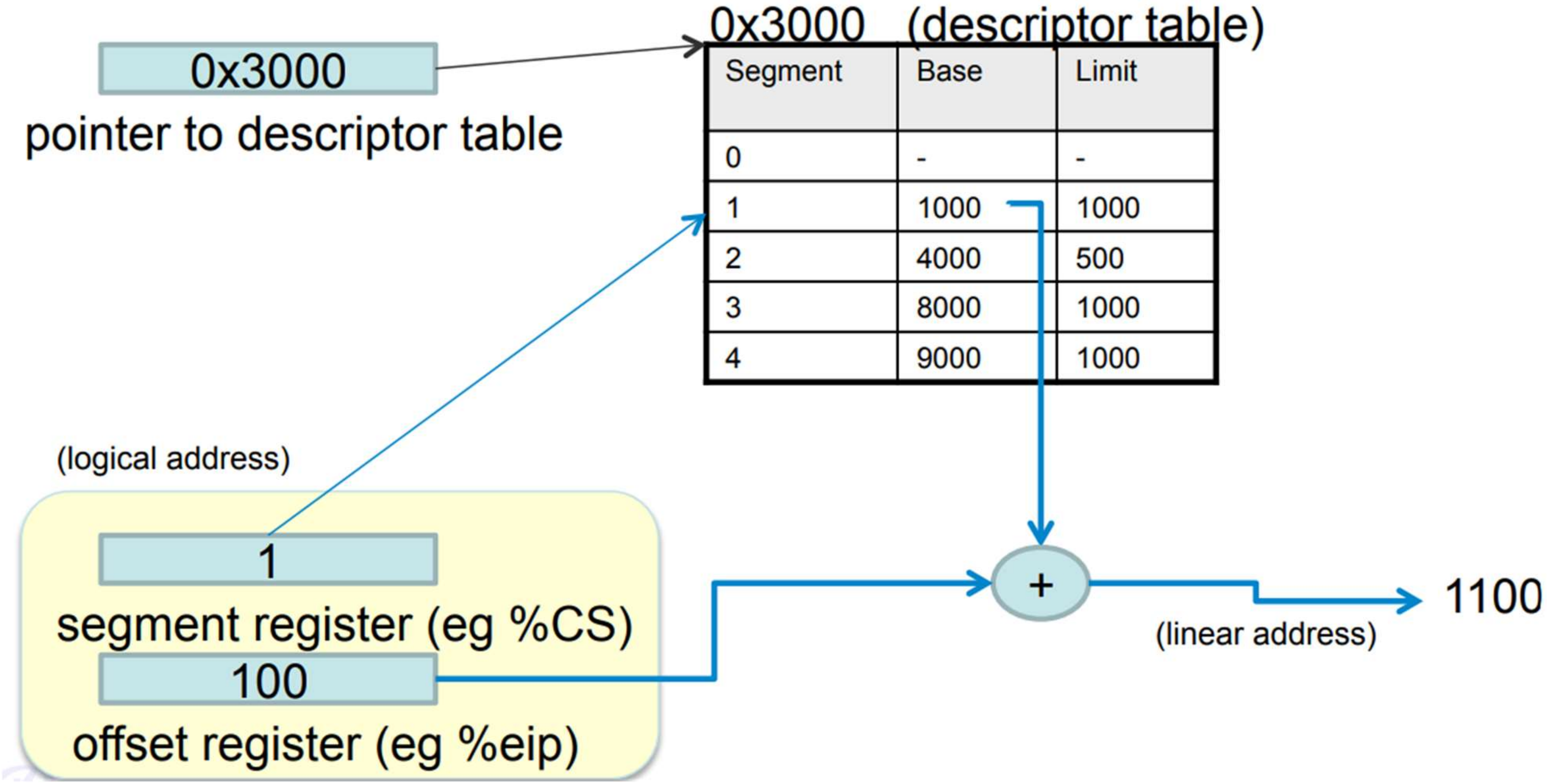
Source: Operating System Concepts by Abraham Silberschatz, Greg Gagne, Peter B. Galvin

x86 memory management



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/4_Memory.pdf

Example of segmentation



0x3000

pointer to descriptor table

0x3000 (descriptor table)

Segment	Base	Limit
0	-	-
1	1000	1000
2	4000	500
3	8000	1000
4	9000	1000

(logical address)

1

segment register (eg %CS)

100

offset register (eg %eip)

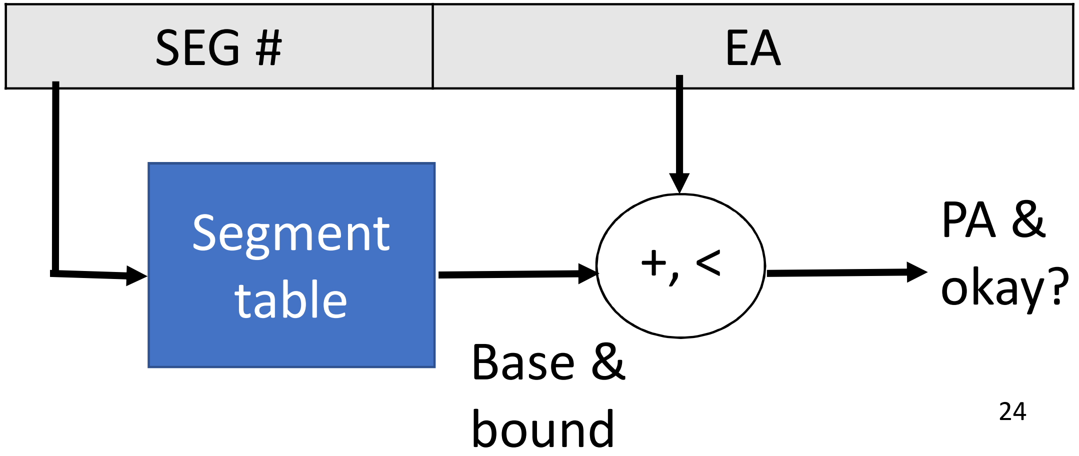
1100

(linear address)

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/4_Memory.pdf

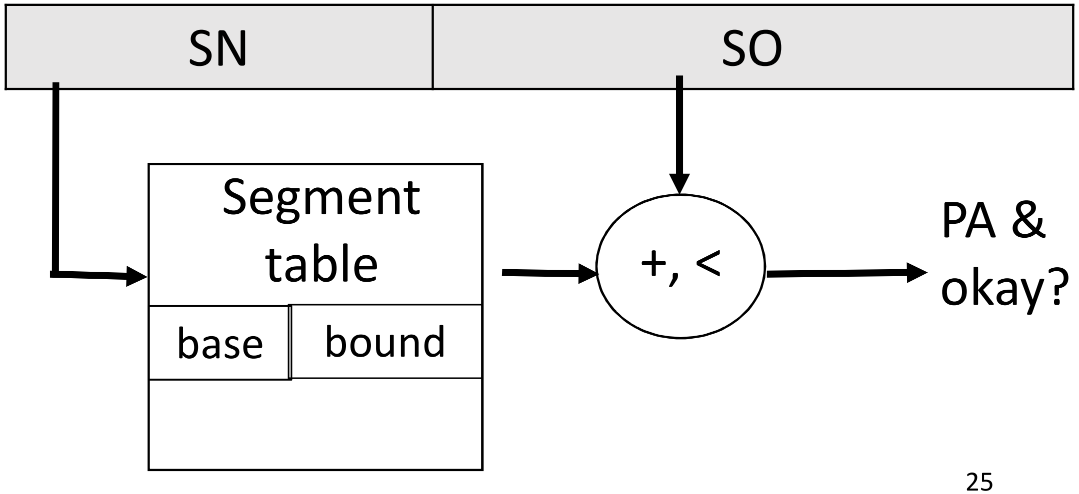
Segmented address space

- **Segment == a base and bound pair**
- **Each process has multiple segments**
  - Separate code and data segments
  - 2 sets of base-and-bound register’s for instruction and data fetch
  - Allowed sharing code segments
- **Segment table**
  - Privileged data structures
  - Private/unique to each process



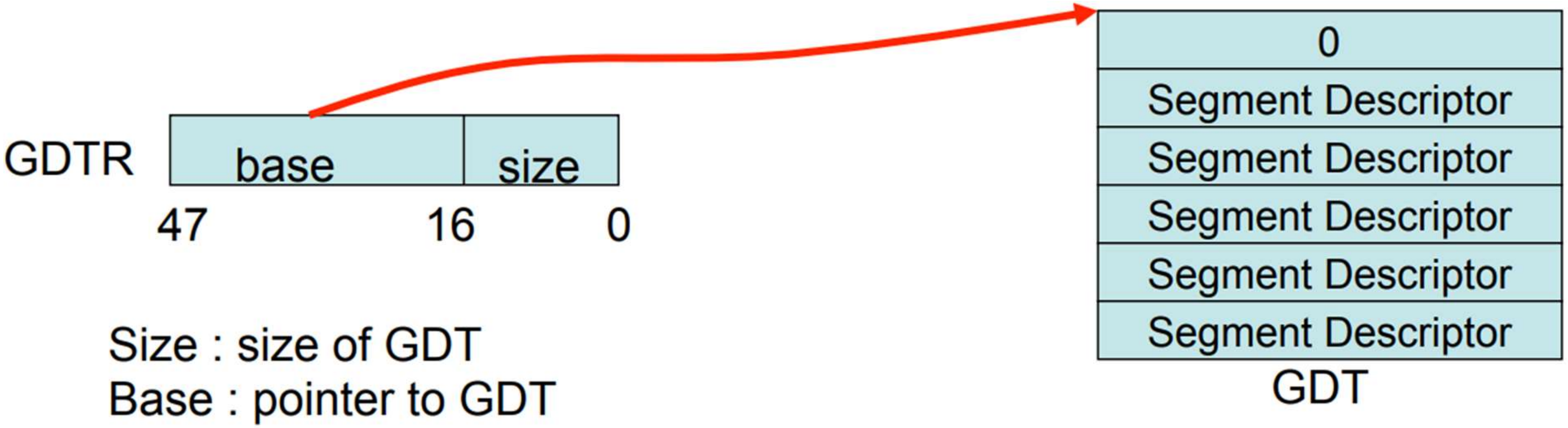
Segmented address translation

- EA: segment number (SN) and a segment offset (SO)
  - SN was used to indicate specified segments
  - Segment size limited by the range of SO
  - Segments can have different sizes
- Segment translation
  - Maps SN to corresponding base and bound
  - Separate mapping for each process



Pointer to descriptor table

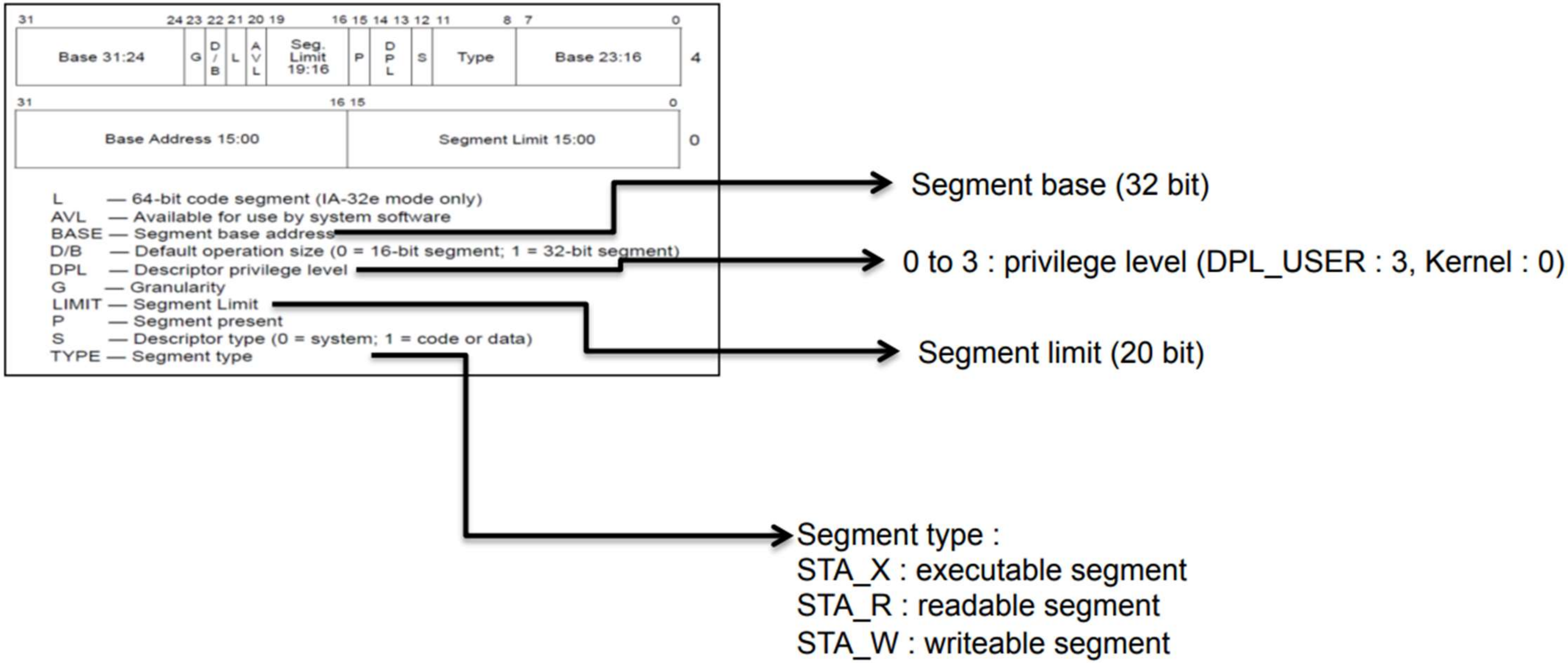
- Global descriptor table (GDT)
    - Stored in memory
- Pointed to by GDTR (GDT Register)
    - lgdt (instruction used to load the GDT register)



Size : size of GDT
Base : pointer to GDT

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/4_Memory.pdf

Segment descriptor



31–24	23	22	21	20	19–16	15	14–13	12	11–8	7–0	
Base 31:24	G	D / B	L	A V L	Seg. Limit 19:16	P	D P L	S	Type	Base 23:16	4

31–16	15–0	
Base Address 15:00	Segment Limit 15:00	0

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Segment base (32 bit)

0 to 3 : privilege level (DPL_USER : 3, Kernel : 0)

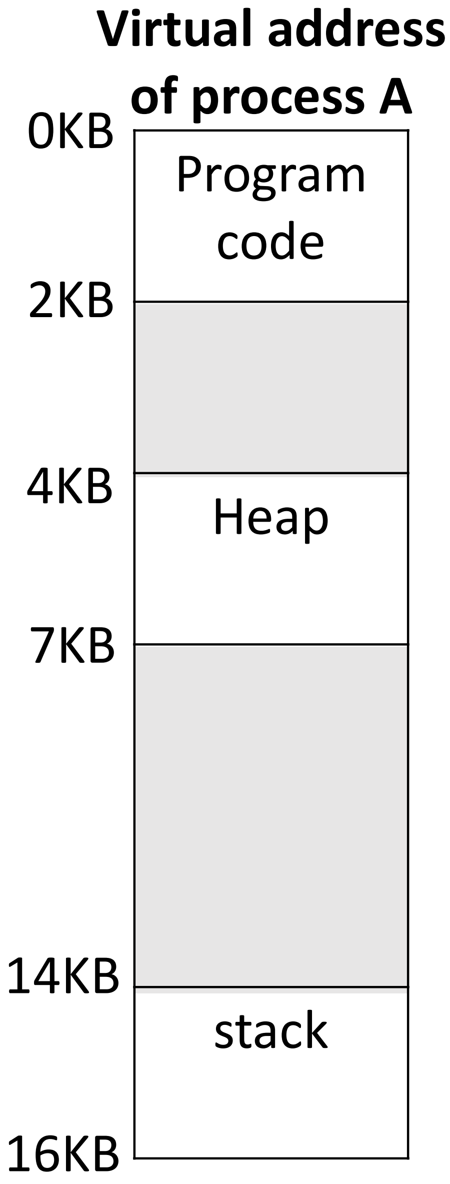
Segment limit (20 bit)

Segment type :
STA_X : executable segment
STA_R : readable segment
STA_W : writeable segment

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/4_Memory.pdf

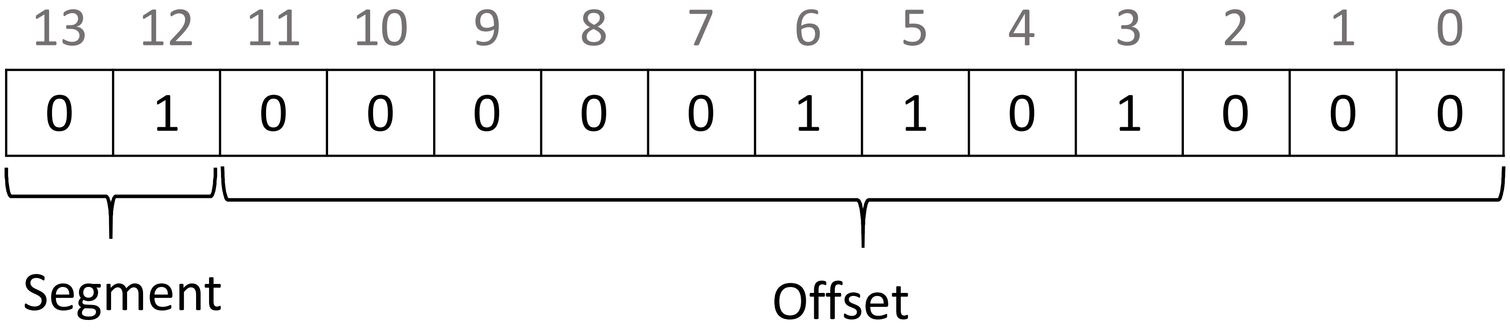
Segmentation fault

- What if we tried to refer to an illegal address ?
  - Beyond the end of heap (a virtual address 7 KB or greater)
  - The hardware detects that address is out of bounds
  - Traps into the OS and terminates the offending process



Which segment are we referring to ?

- Which segment an address refers ?
  - The following shows the heap virtual address **4200** in binary form
  - The top two bits (01) tells the hardware which segment we are referring to
  - The bottom 12 bits are offset into the segment
    0000 0110 1000, or hex 0x068, or 104 in decimal



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

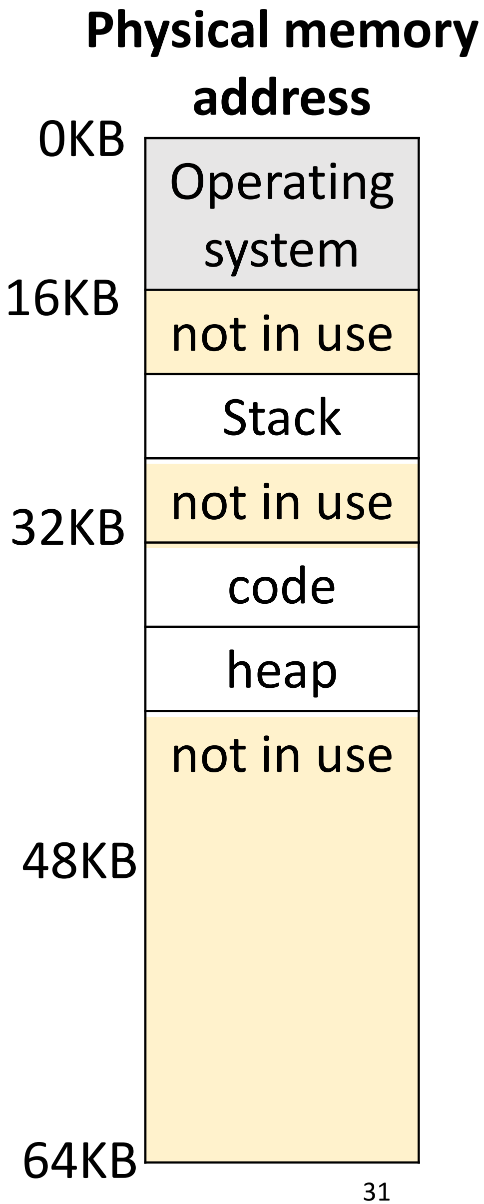
To fully utilize the virtual address space

- One segment of the address space goes unused
  - If we use the top two bits, and we only have three segments (code, heap, and stack)
  - Some systems put code in the same segment as the heap, and use only one bit to select which segment to use
- Using many bits to select a segment that limits the use of the virtual address space
  - Each segment is limited to a maximum size

What about the stack ?

- The stack grows backwards
  - The stack starts at 28 KB, grows back in physical memory
  - The hardware needs to know which way the segment grows

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



Physical memory address

0KB

Operating system

16KB

not in use

Stack

not in use

32KB

code

heap

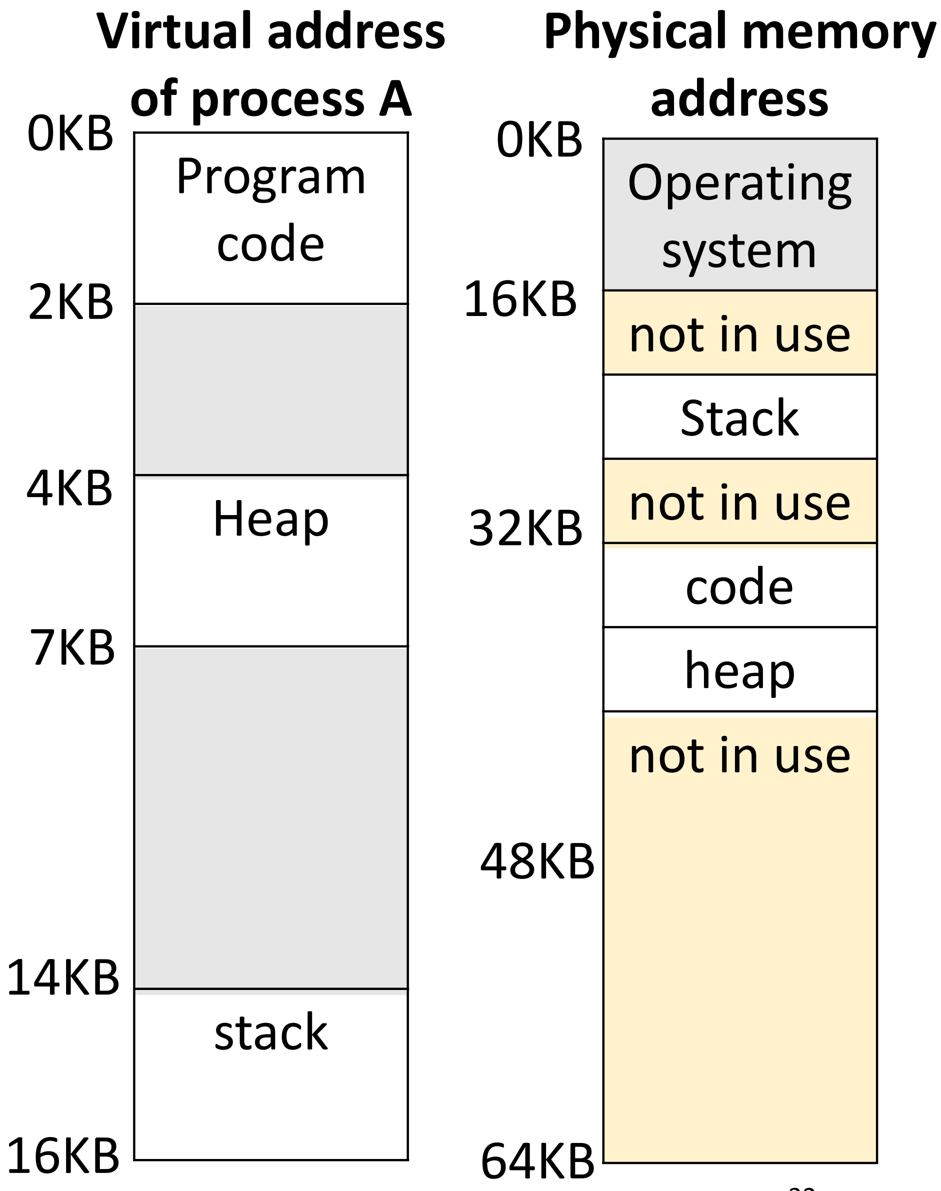
not in use

48KB

64KB

Case study: mapping stack

- Assume we wish to access virtual address 15 KB
  - Virtual address: 11 1100 0000 0000 (hex 0x3C00)
  - We are left with an offset of 3KB
  - A segment is 4KB
  - The correct negative offset is -1KB (=3KB – 4KB)
  - The correct physical address is 27 KB (=-1 KB + 28 KB)



Virtual address of process A

0KB

Program code

2KB

4KB

Heap

7KB

14KB

stack

16KB

Physical memory address

0KB

Operating system

16KB

not in use

Stack

32KB

not in use

code

heap

not in use

48KB

64KB

Support for sharing

- To save memory,
  - Share certain memory segments between address space
  - To support sharing, we need extra **protection bits** per segment
  - The read-only segment can be shared across multiple processes

Segment	Base	Size (max 4K)	Grows Positive ?	Protection
Code	32K	2K	1	Read-execute
Heap	34K	3K	1	Read-write
Stack	28K	2K	0	Read-write

Fine-grained vs. coarse-grained segmentation

- **Coarse-grained segmentation**
  - A system just has a few segments (i.e., code, stack , heap)
- **Fine-grained segmentation**
  - Consists of a large number of smaller segments
  - Using a segment table stored in memory to manage segments
- Why fine-grained segmentation ?
  - The OS could better learn about which segments are in use
  - Use the main memory more effectively

Context switch with base and bounds

- **Context switch**
  - Add base and bounds registers to PCB
  - Steps during context-switch
    - Change to privileged mode
    - Save base and bounds registers of old process
    - Load base and bounds registers of new process
    - Change to user mode and jump to new process

https://pages.cs.wisc.edu/~eli/537/lectures/lecture13.2x2.pdf

Summary

- **Good features of segmentation**
  - More flexible than base and bounds -> enable sharing (How ?)
  - Reduces severity of fragmentation (How ?)
- **Problems**
  - Still have fragmentation -> How ? What kind ?
  - Non-contiguous virtual address space -> Real problem ?
- **Possible solutions**
  - Fragmentation: Copy and compact
  - Paging