Virtualizing Memory

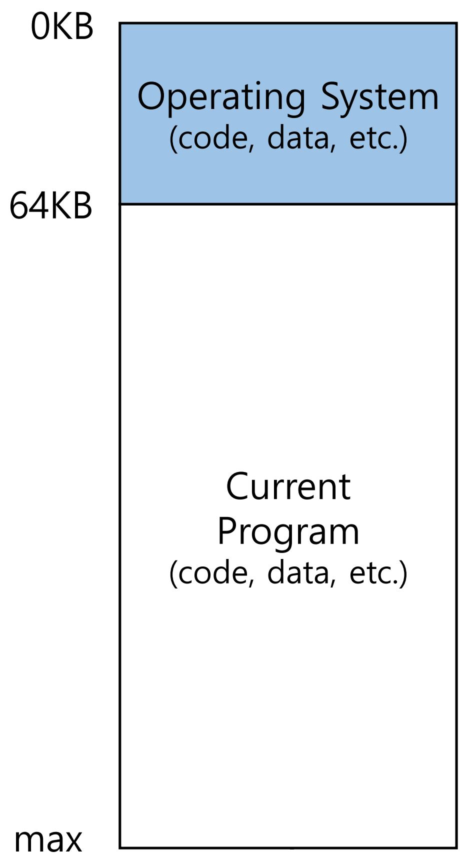
Sridhar Alagar

Virtualization

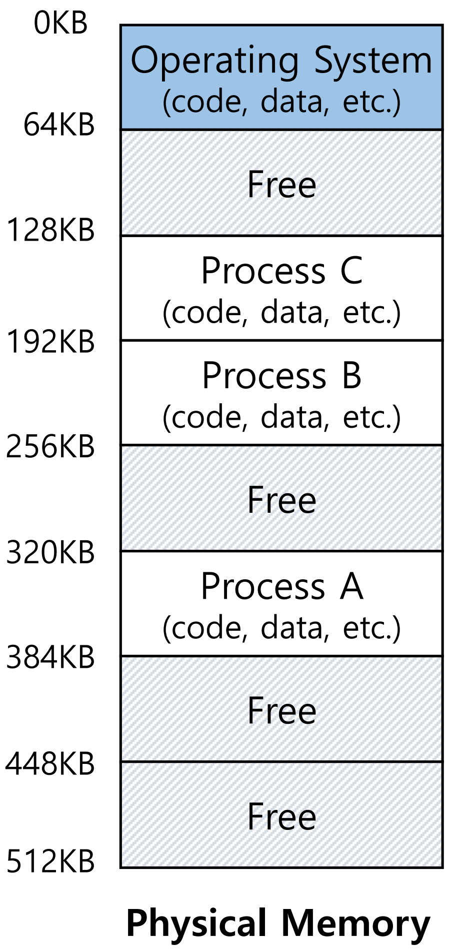
- CPU
  - Abstraction: Process
  - Sharing:
    - Policy
    - Mechanism
  - Isolation
    - Limited direct execution
- Memory
  - Abstraction
  - Sharing
  - Isolation

Early OS

- Uniprogram: load only one program
- Drawback?
  - poor utilization
  - process can destroy OS



Multiprogramming

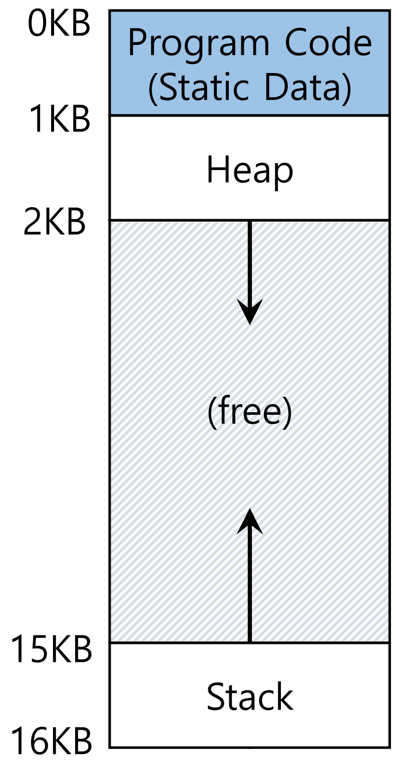


Multiprogramming Goals

- Transparency
  - processes are not aware of sharing
  - works regardless of where it is loaded in memory
- Protection
  - cannot access OS or other process' memory
- Efficient
- Share memory among cooperating process

Abstraction: Address Space

- Each process has a set of addresses that map to bytes
- What is in an address space?
- Static: code and global variables
- Dynamic: heap and stack



Quiz: Find that address location

```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int)););
}
```

- Possible segments: static data, code, stack, heap

Address	Location
x	Static data
main	Code
y	Stack
z	Stack
*z	Heap

Example: Memory Access

```
void func()
      int x;
      ...
      x = x + 3;//this is the line of code we are interested
```

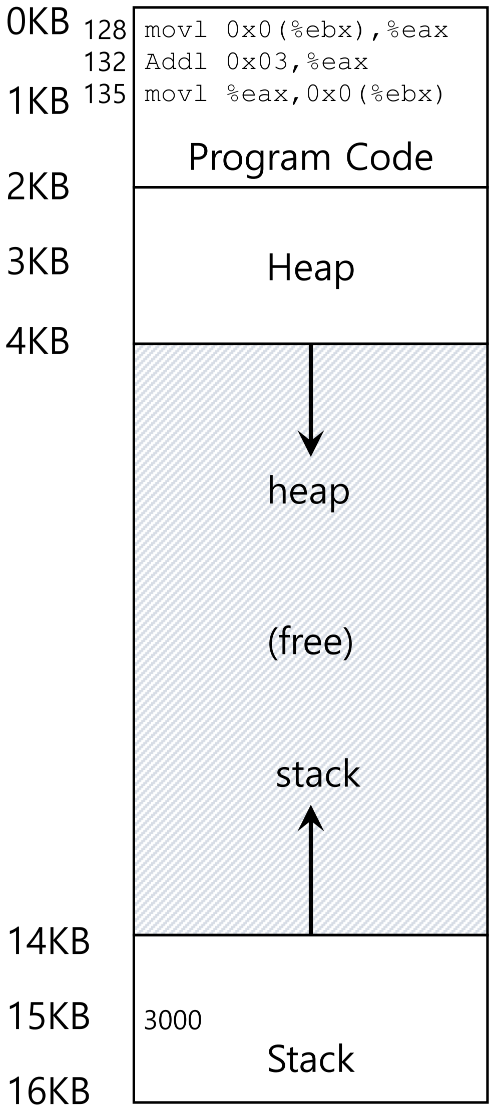
Load a value from memory
Increment it by three
Store the value back into memory

Example: Memory access

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

Load a value from memory
Increment it by three
Store the value back into memory

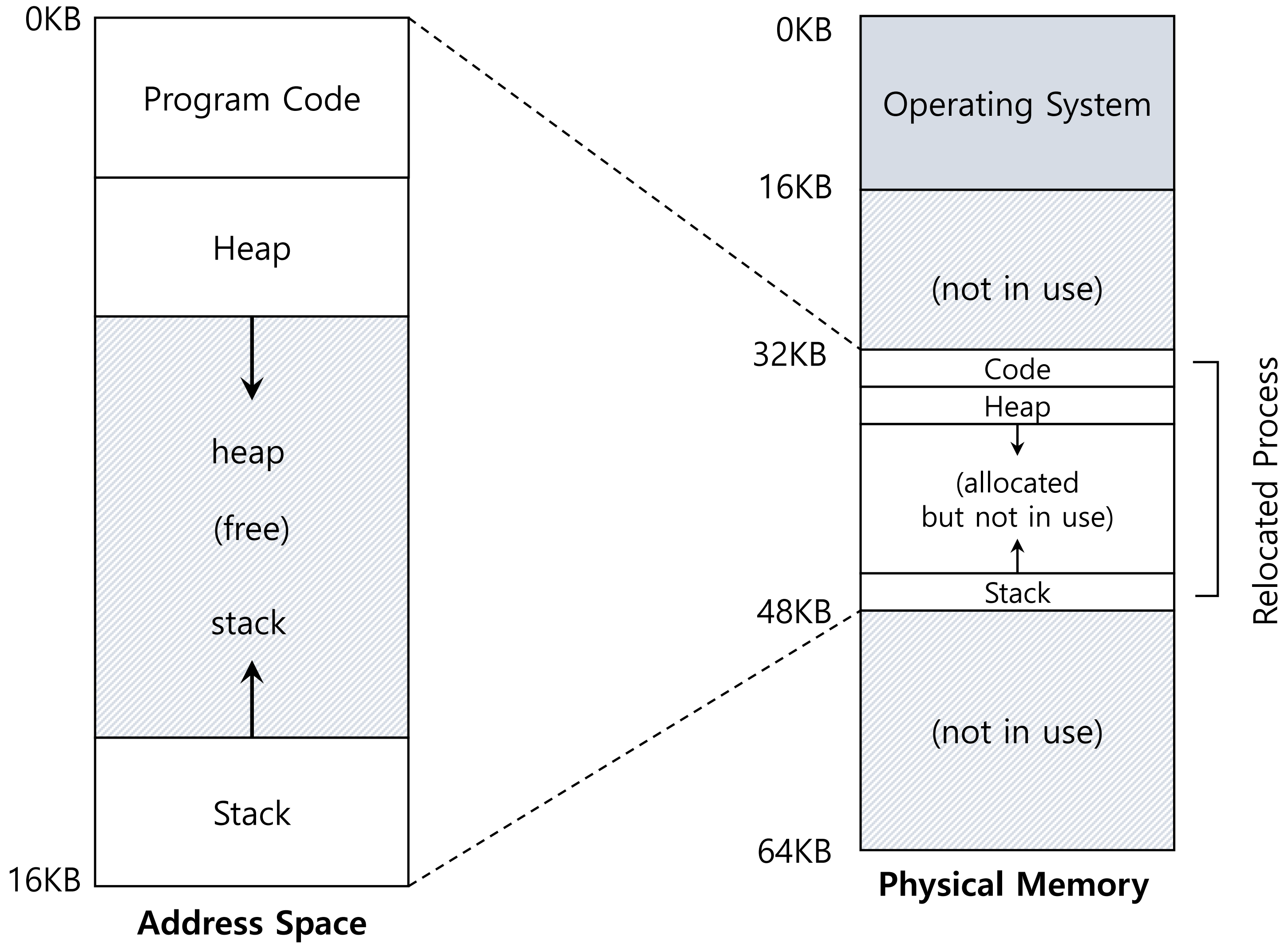
Example: Memory access



- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

Relocation

- OS wants to relocate the process

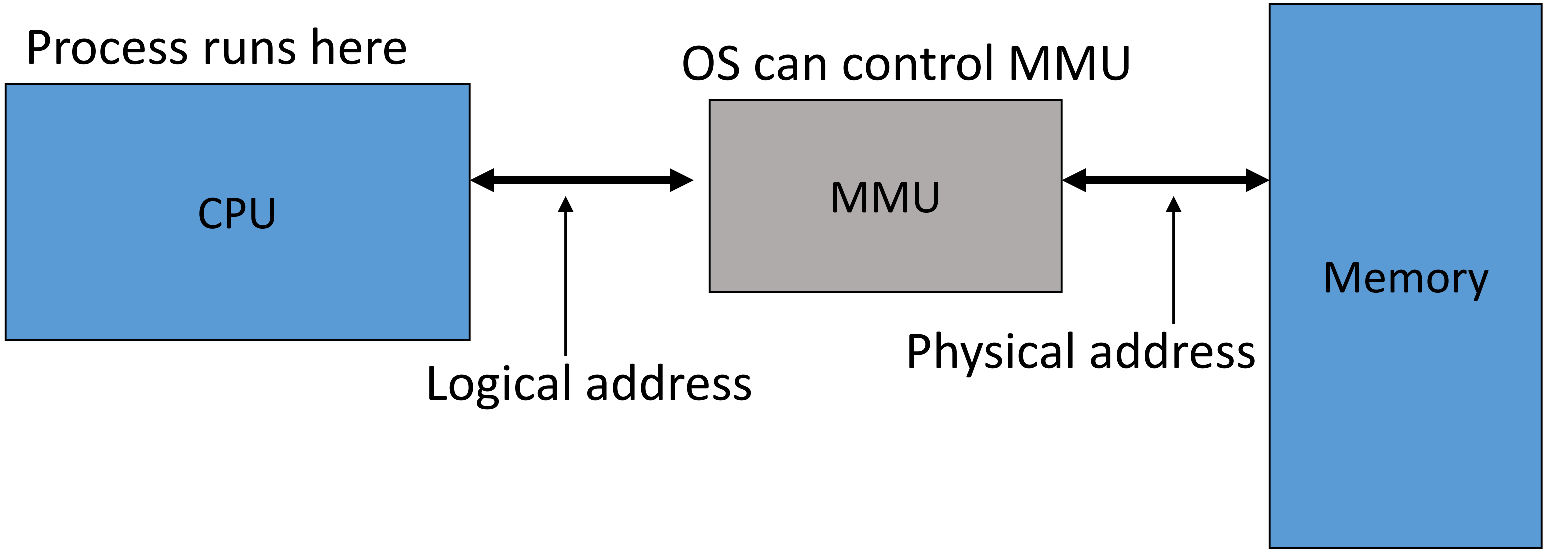


Dynamic relocation

- Process generates virtual address
- Memory access needs real address
- Every virtual address referenced by a process must be dynamically translated to a real address

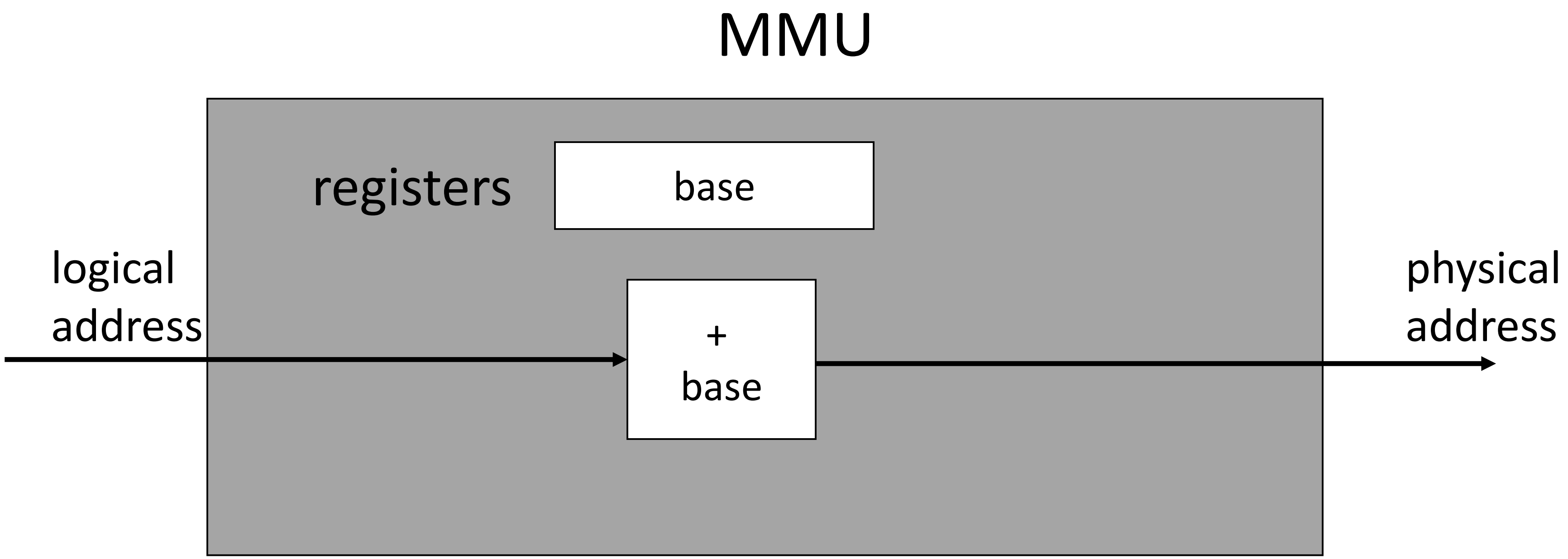
Dynamic relocation

- H/W does the translation: Memory Management Unit (MMU)

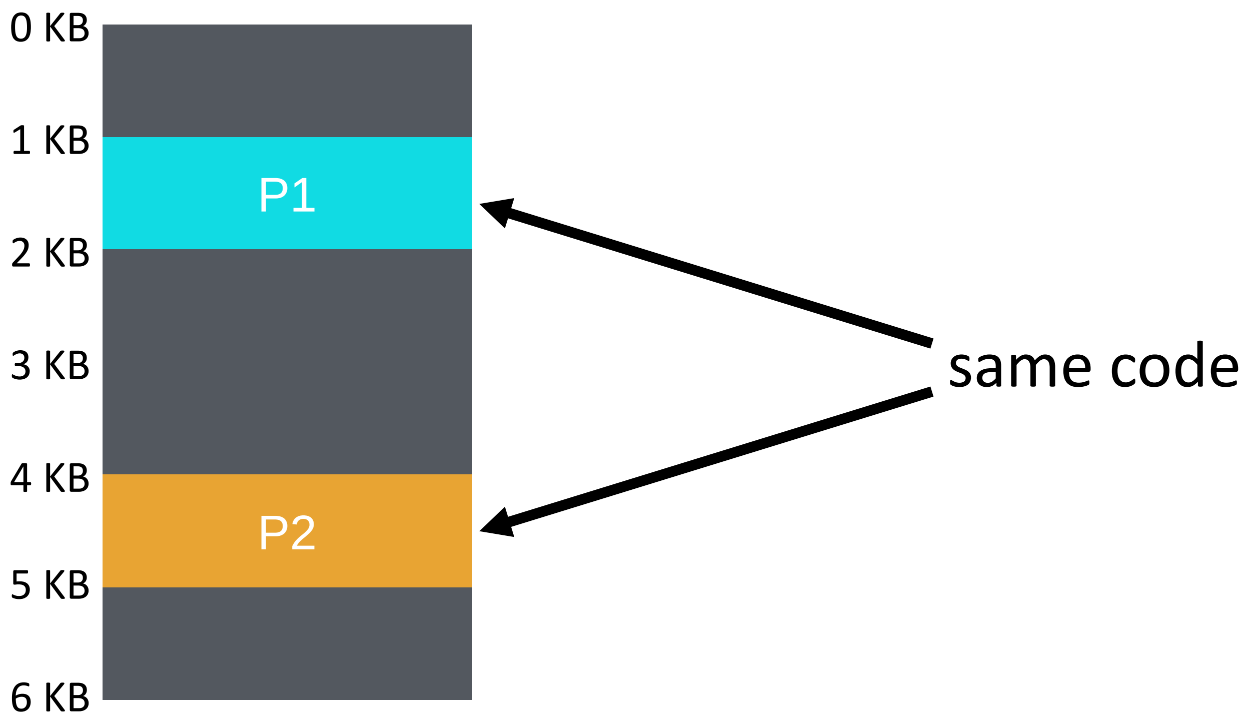


Dynamic relocation: Base register

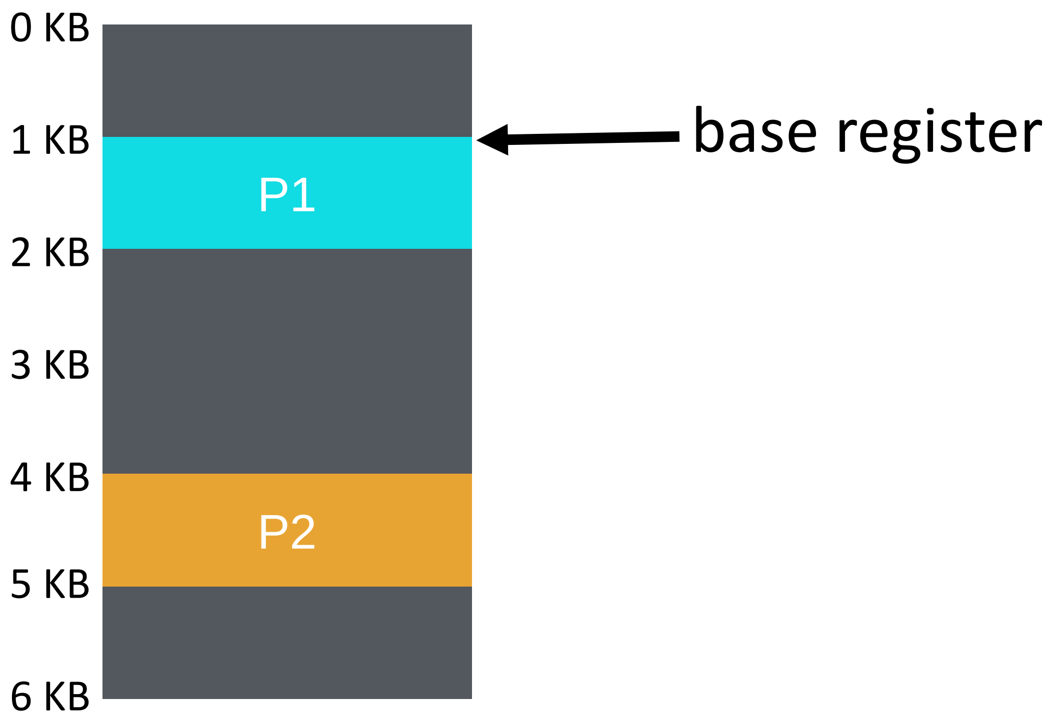
- MMU contains base register and adds it value to virtual address



Dynamic relocation: Example

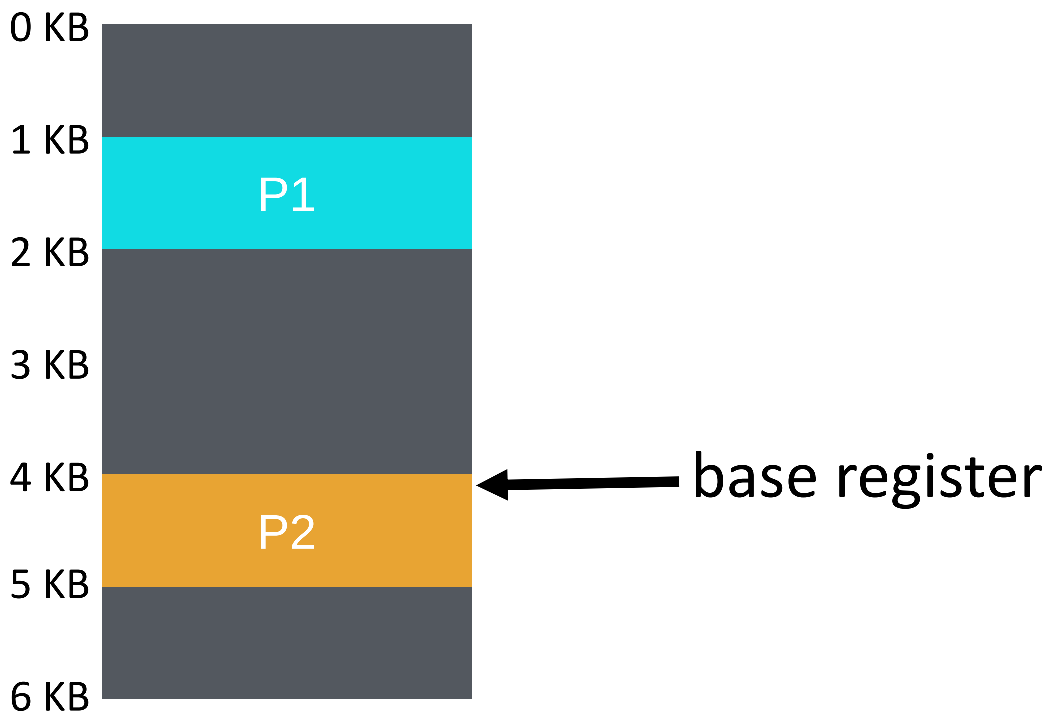


Dynamic relocation: Example



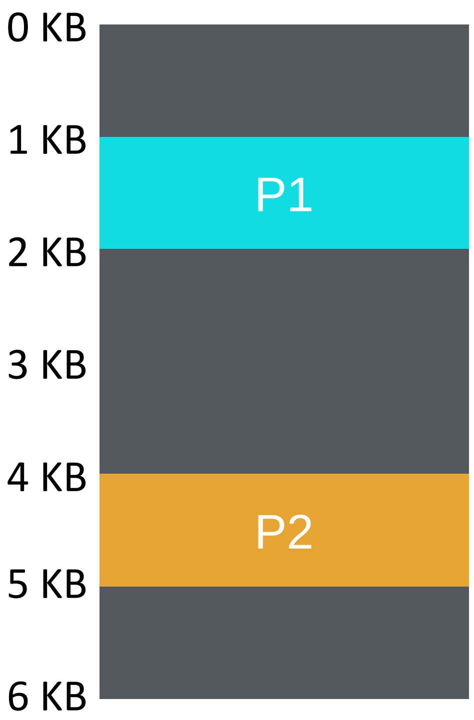
P1 is running

Dynamic relocation: Example



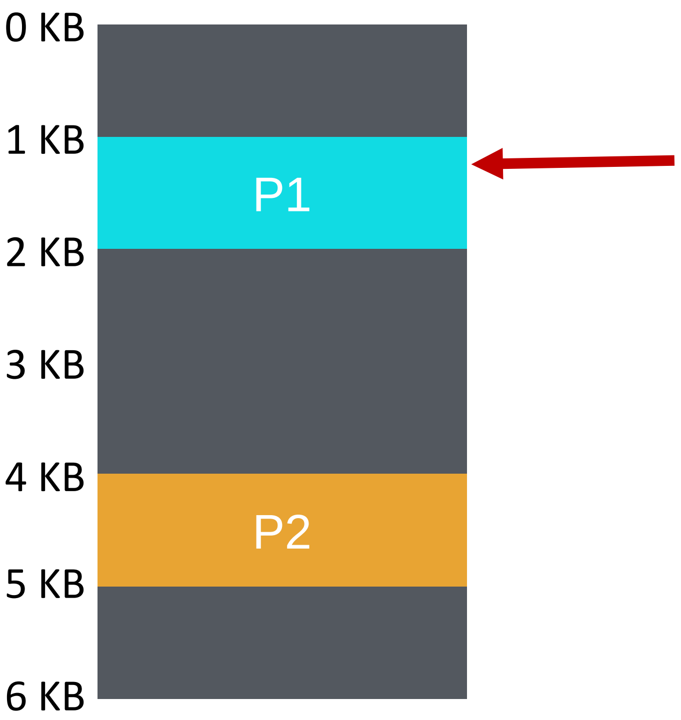
P2 is running

Dynamic relocation: Example



Virtual	Physical
P1: load 100, R1	

Dynamic relocation: Example

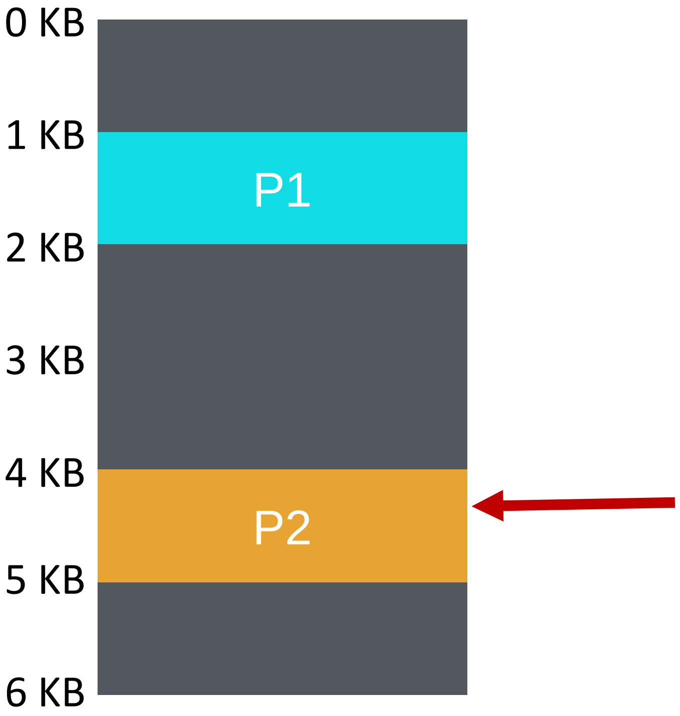


Virtual	Physical
P1: load 100, R1	load 1124, R1

Dynamic relocation: Example

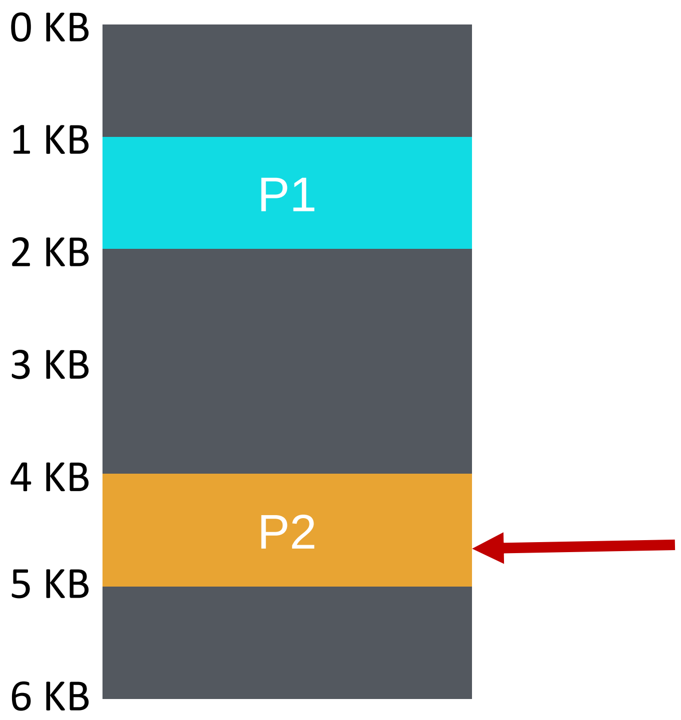
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	

Dynamic relocation: Example



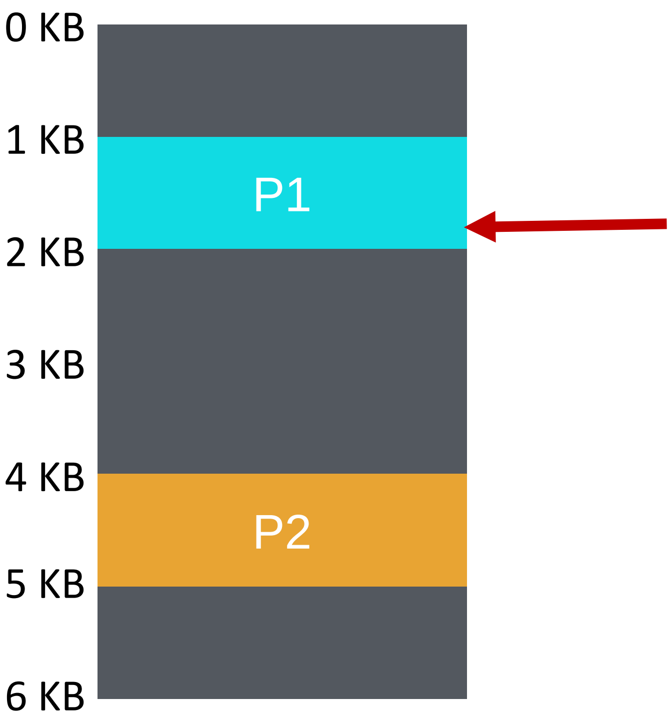
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1

Dynamic relocation: Example



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1

Dynamic relocation: Example



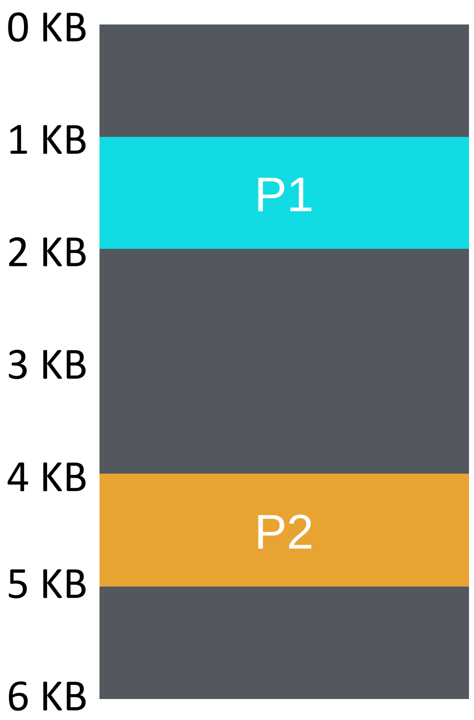
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

Who controls base register?

- Who should translate the address with base register?
  (a) process
  (b) OS
  (c) h/w

- Who should modify the base register?
  (a) process
  (b) OS
  (c) h/w

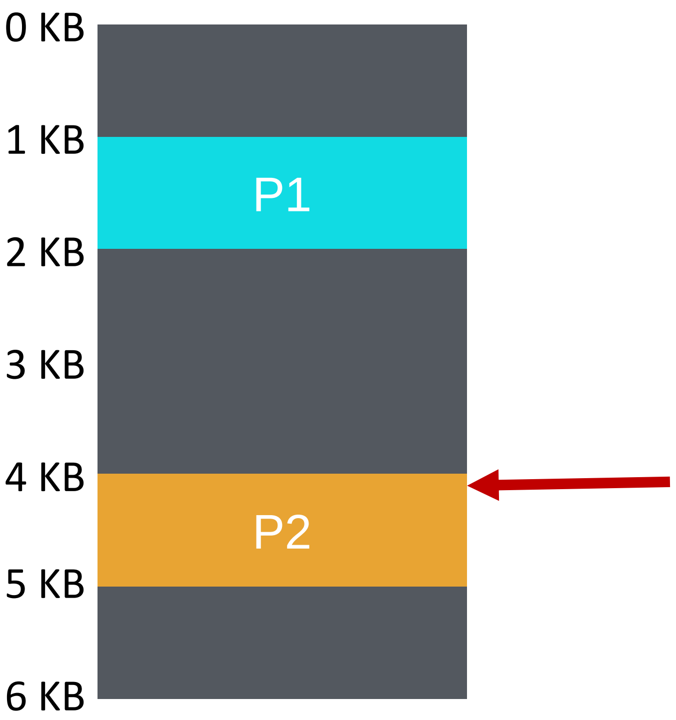
Dynamic relocation: Protection



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

Can P1 corrupt P2?

Dynamic relocation: Protection



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	store 4096, R1 (3072 + 1024)

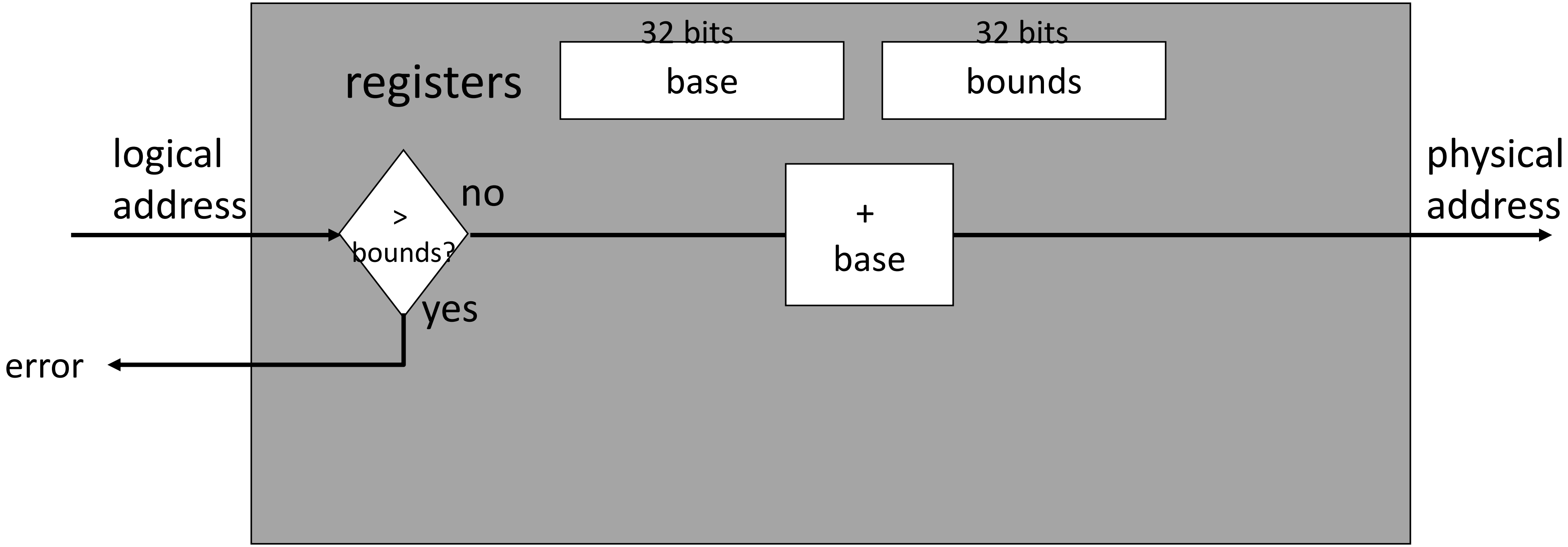
Can P1 corrupt P2?

Dynamic relocation with base register is not adequate for protection.

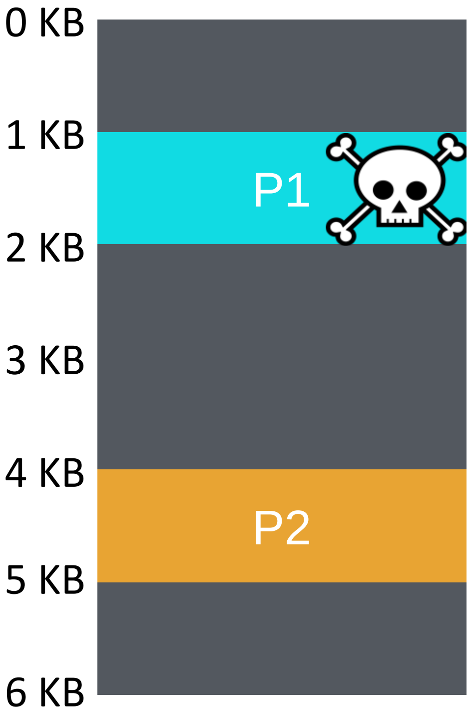
Dynamic relocation: Bounds register

- Limit the address space with bounds register
- Base register: starting location (smallest physical addr)
- Bounds register: size of the process virtual space
  - could be largest physical address (base + size)
- What should OS do if a process load/stores beyond bounds?

MMU with base and bounds register



Dynamic relocation: Protection with bounds



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	interrupt OS (3072 > 1024)

Can P1 corrupt P2?

Context Switching

- What to do with base and bounds registers during context switch?
  - base and bound part of process' context
  - save them in PCB of old process
  - restore them from PCB of new process

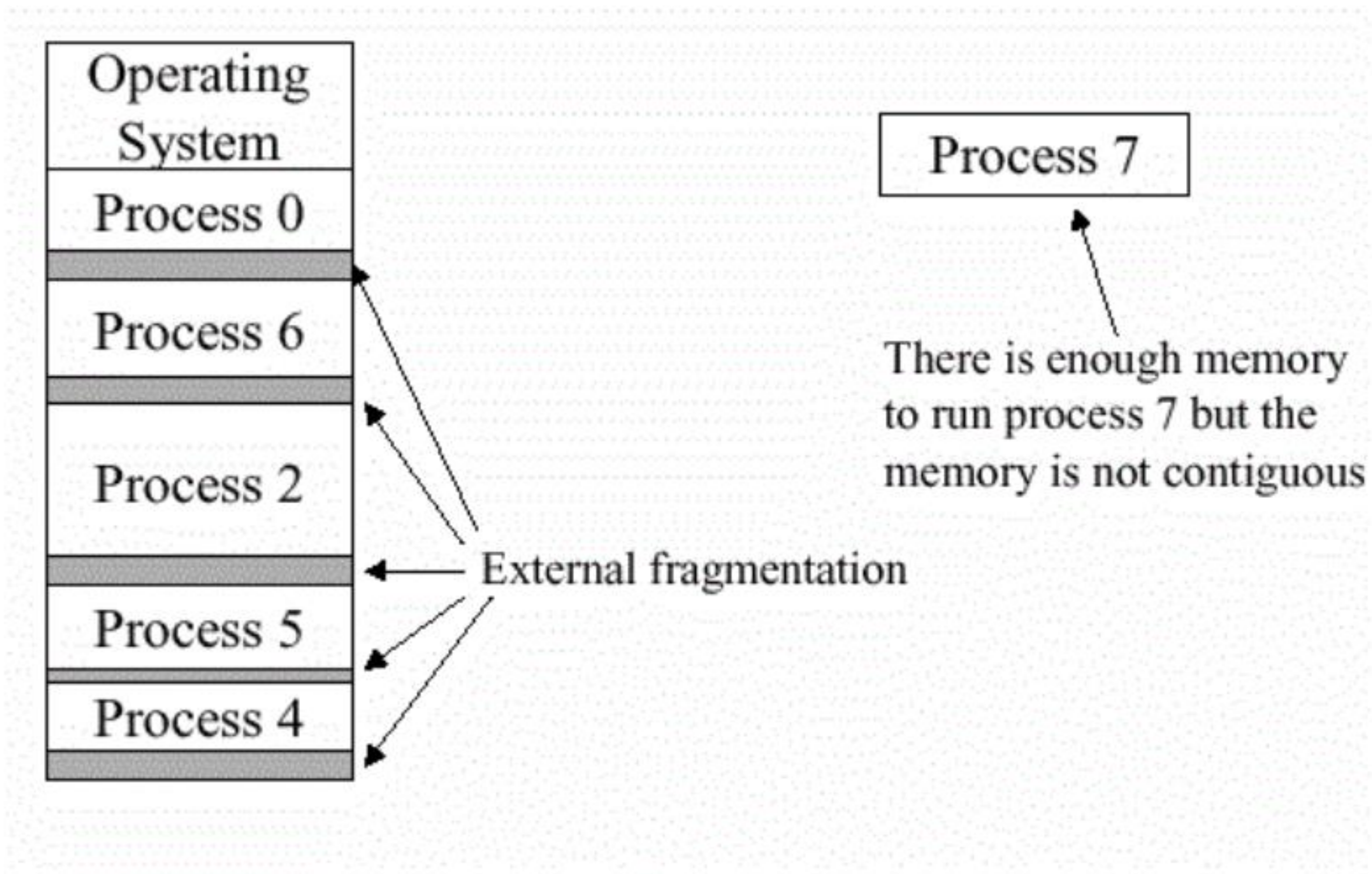
Base and Bounds: Advantages

- Dynamic relocation
- Protection
- Simple to implement in h/w

Base and Bounds: Disadvantages

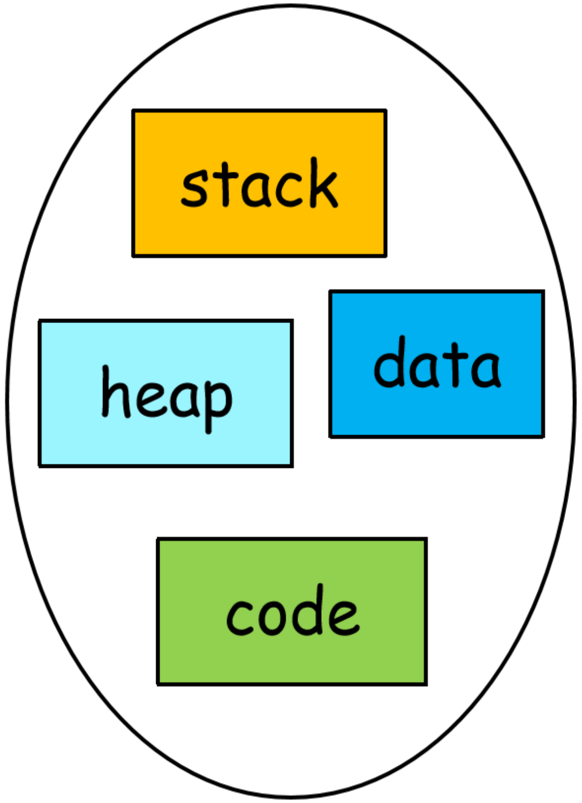
- Process requires contiguous memory space
- A process may not use all the space allocated to it
  - internal fragmentation
- There may not be enough memory that are contiguous, but total free memory available is greater that what a process needs
  - external fragmentation

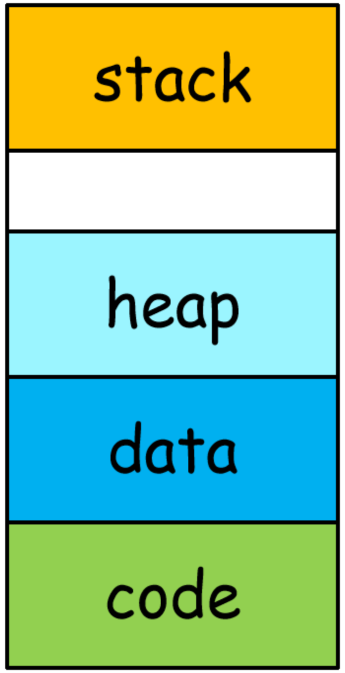
External Fragmentation



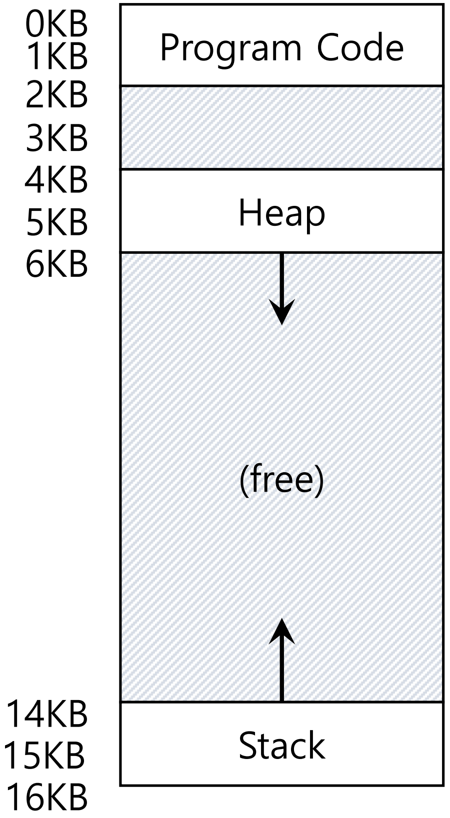
Segmentation

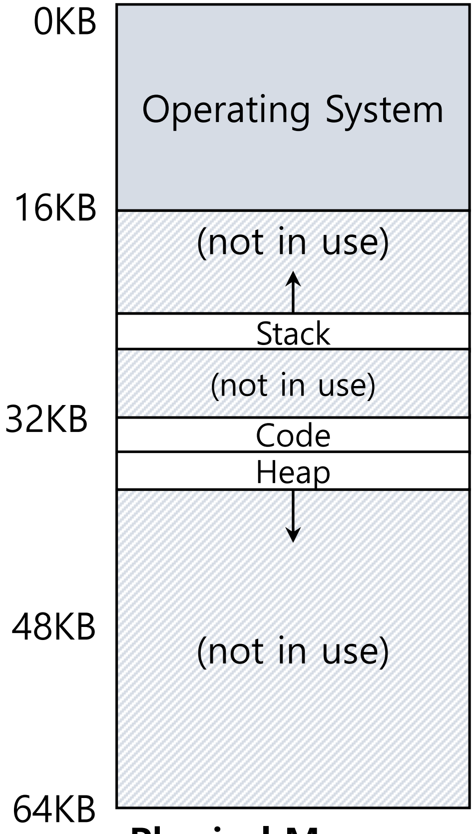
- Divide a process' address space into logical segments
  - each segment corresponds to a logical entity
- Each segment can be
  - independently placed separately in memory
  - grow and shrink (varied size)
  - be protected





Placing Segment in Memory





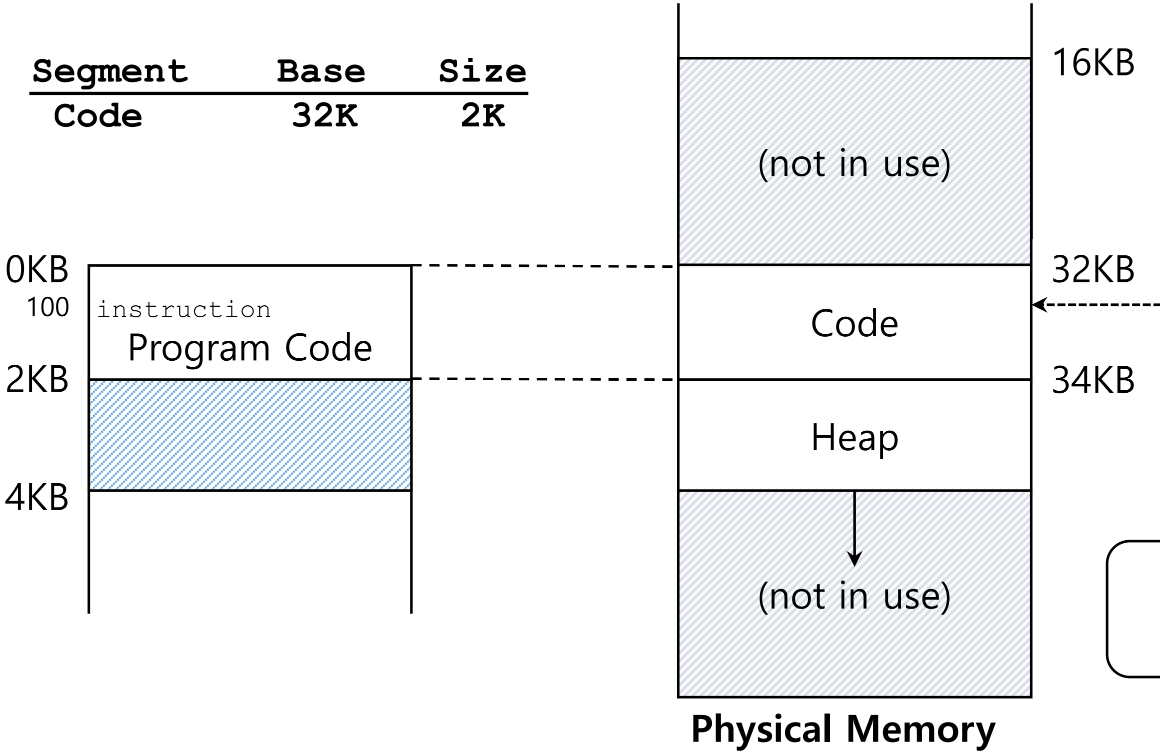
Physical Memory

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

Address translation in Segmentation

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

Segment	Base	Size
Code	32K	2K



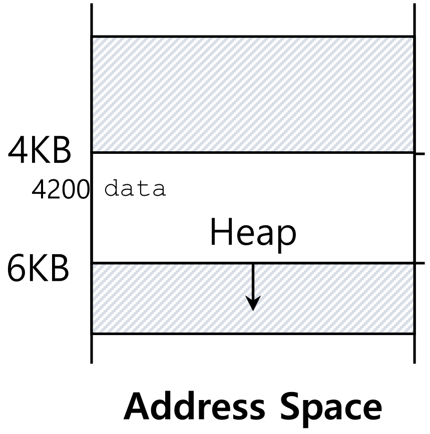
$100 + 32K$ or 32868 **is the desired physical address**

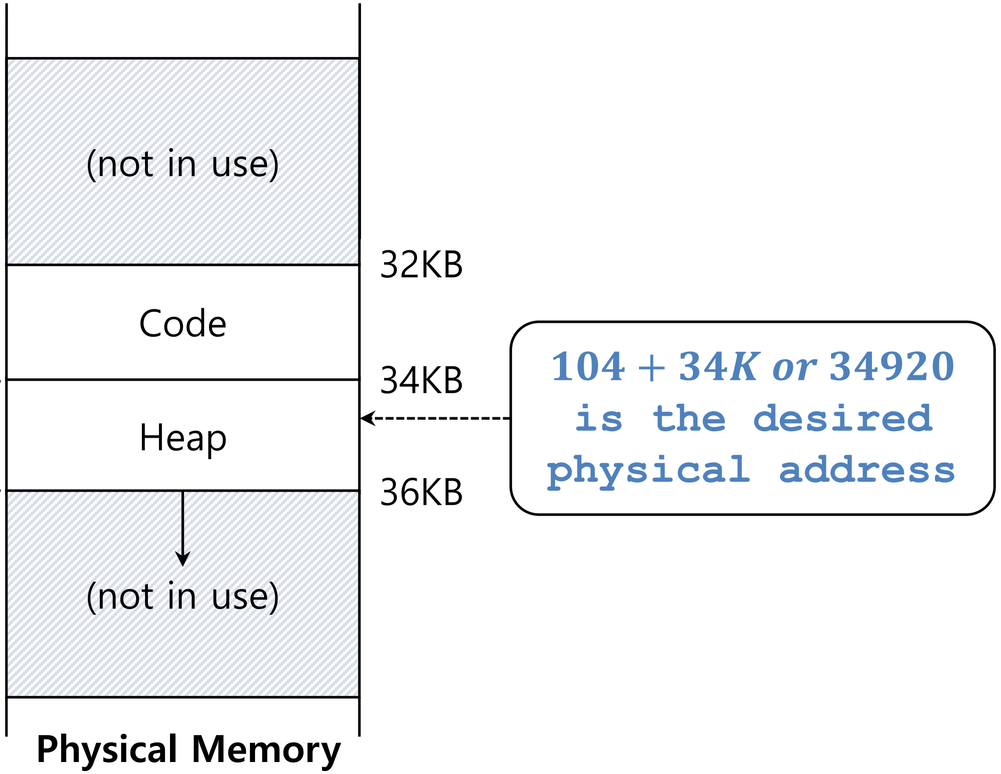
$$physical\ address = offset + base$$

Address translation in Segmentation

- The offset of virtual address 4200 is 104
  - The heap segment **starts at virtual address 4096**

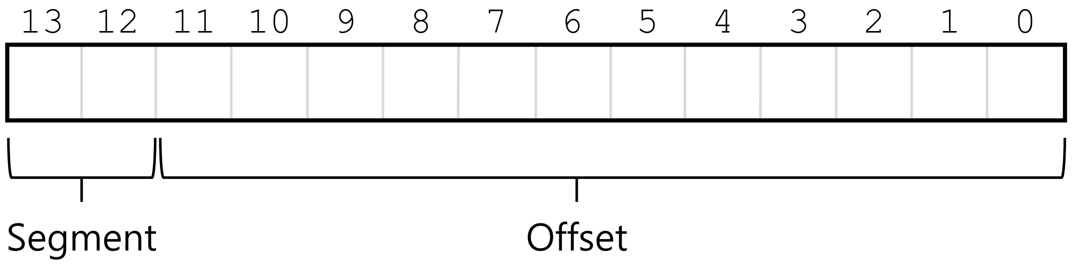
Segment	Base	Size
Heap	34K	2K





Which segment are we referring to?

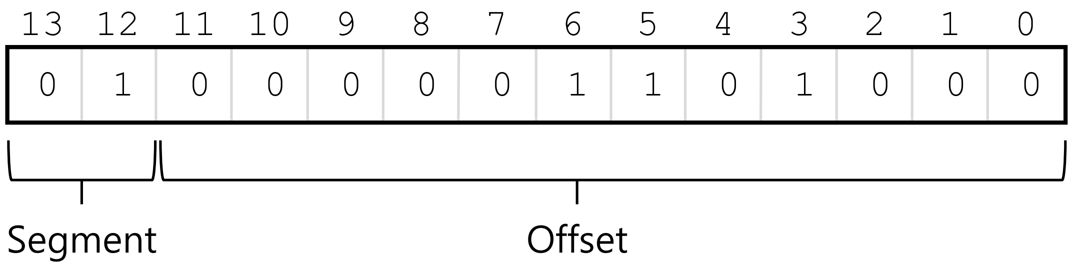
- Explicit approach: use part of logical address
  - The top bits specify the segment
  - The remaining bits specify the offset with in segment



Which segment are we referring to?

- Example: virtual address 4200 (01000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11

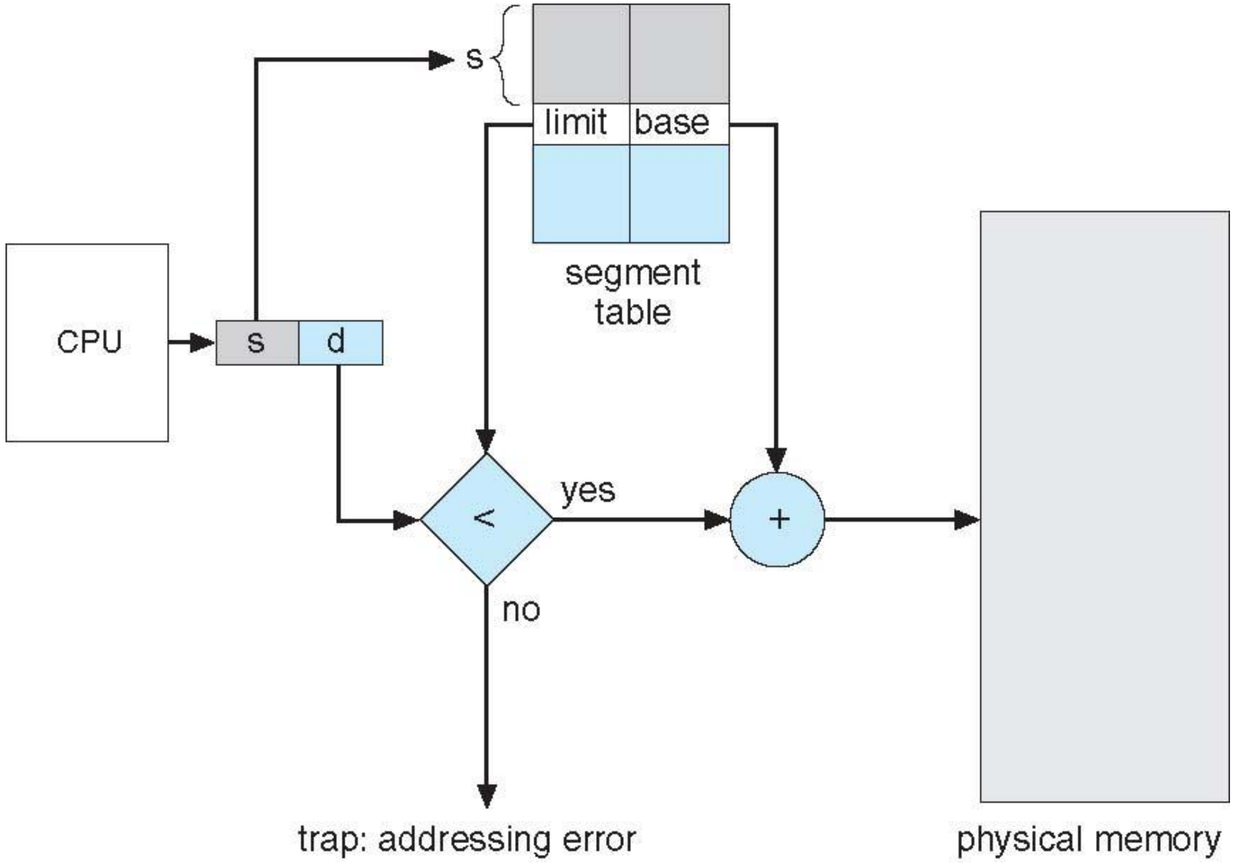


How to find segment and offset?

```
// 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(11000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (00111111111111)

Segmentation Hardware



Sharing segment

- Add permission/protection bits in segment table

Segment	Base	Size	Protection
Code	32K	2K	Read-Execute
Heap	34K	2K	Read-Write
Stack	28K	2K	Read-Write

Segmentation

- Advantages
  - Segment sharing
  - Easier to relocate segment than entire program
- Disadvantages
  - Segments have variable length
  - Segments can be large
  - Fragmentation not solved

Disclaimer

- Some of the materials in this lecture slides are from the lecture slides by Prof. Arpaci, Prof. Youjip, and other educators. Thanks to all of them.