Introduction to virtual memory and Mechanism of Address

Why virtualize memory?

- Because real view of memory is messy!
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory of many processes must be in memory
  - Non-contiguous too
- Need to hide this complexity from user

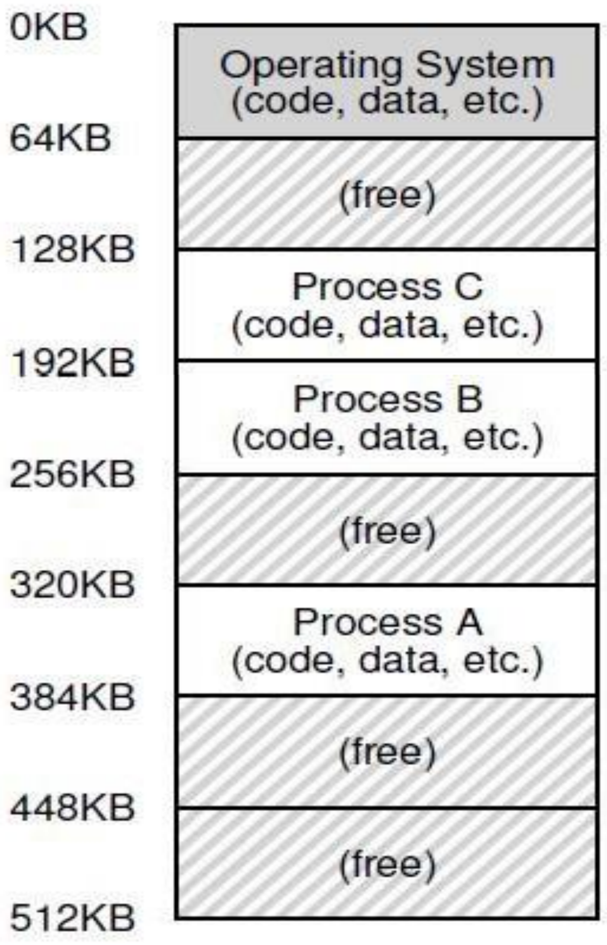


Figure 13.2: **Three Processes: Sharing Memory**

Abstraction: (Virtual) Address Space

- Virtual address space: every process assumes it has access to a large space of memory from address 0 to a MAX
- Contains program code (and static data), heap (dynamic allocations), and stack (used during function calls)
  - Stack and heap grow during runtime
- CPU issues loads and stores to virtual addresses

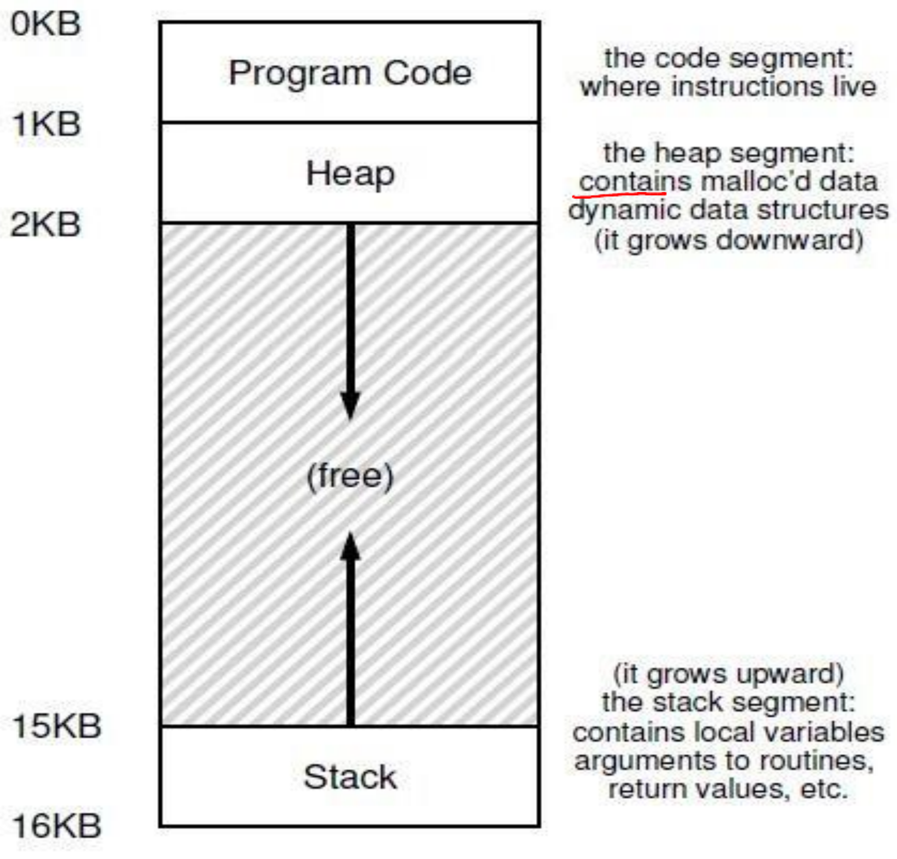
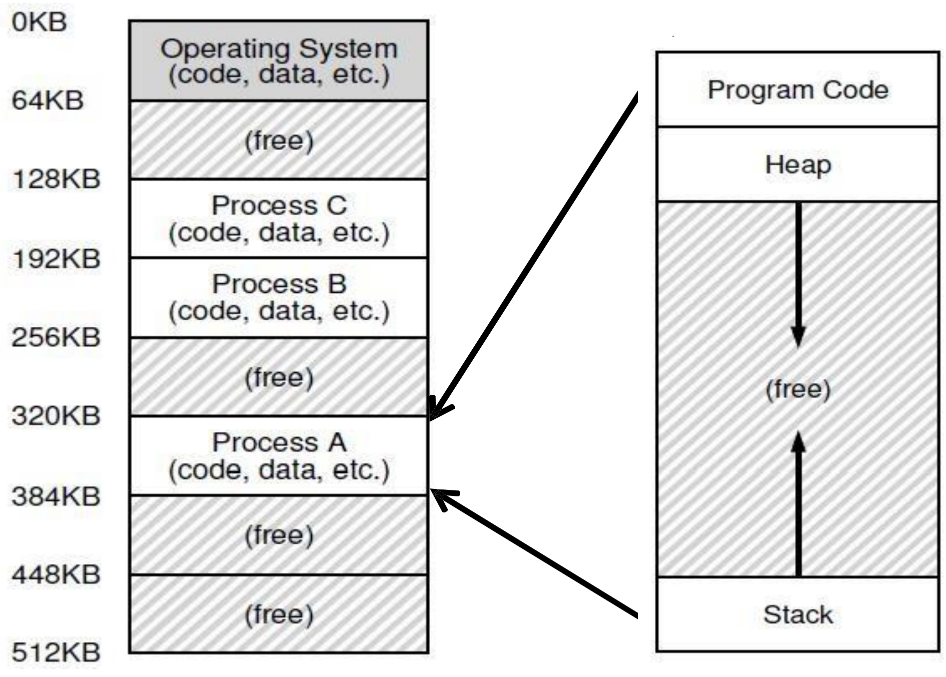


Figure 13.3: **An Example Address Space**

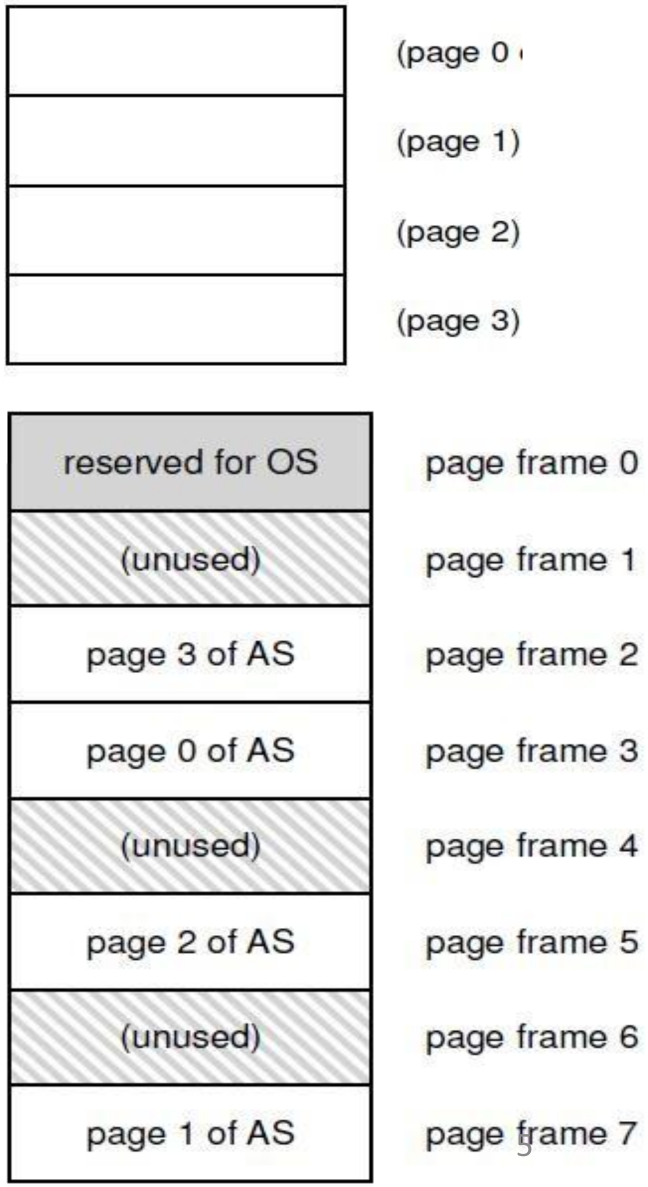
How is actual memory reached?

- Address translation from virtual addresses (VA) to physical addresses (PA)
  - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes
- Translation done by memory hardware called Memory Management Unit (MMU)
  - OS makes the necessary information available


0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)
Program Code
Heap
(free)
Stack

Example: Paging

- OS divides virtual address space into fixed size pages, physical memory into frames
- To allocate memory, a page is mapped to a free physical frame
- Page table stores mappings from virtual page number to physical frame number for a process (e.g, page 0 to frame 3)
- MMU has access to page tables, and uses it to translate VA to PA



(page 0

(page 1)

(page 2)

(page 3)

reserved for OS	page frame 0
(unused)	page frame 1
page 3 of AS	page frame 2
page 0 of AS	page frame 3
(unused)	page frame 4
page 2 of AS	page frame 5
(unused)	page frame 6
page 1 of AS	page frame 7

Goals of memory virtualization

- Transparency: user programs should not be aware of the messy details
- Efficiency: minimize overhead and wastage in terms of memory space and access time
- Isolation and protection: a user process should not be able to access anything outside its address space

How can a user allocate memory?

- OS allocates a set of pages to the memory image of the process
- Within this image
  - Static/global variables are allocated in the executable
  - Local variables of a function on stack
  - Dynamic allocation with `malloc` on the heap

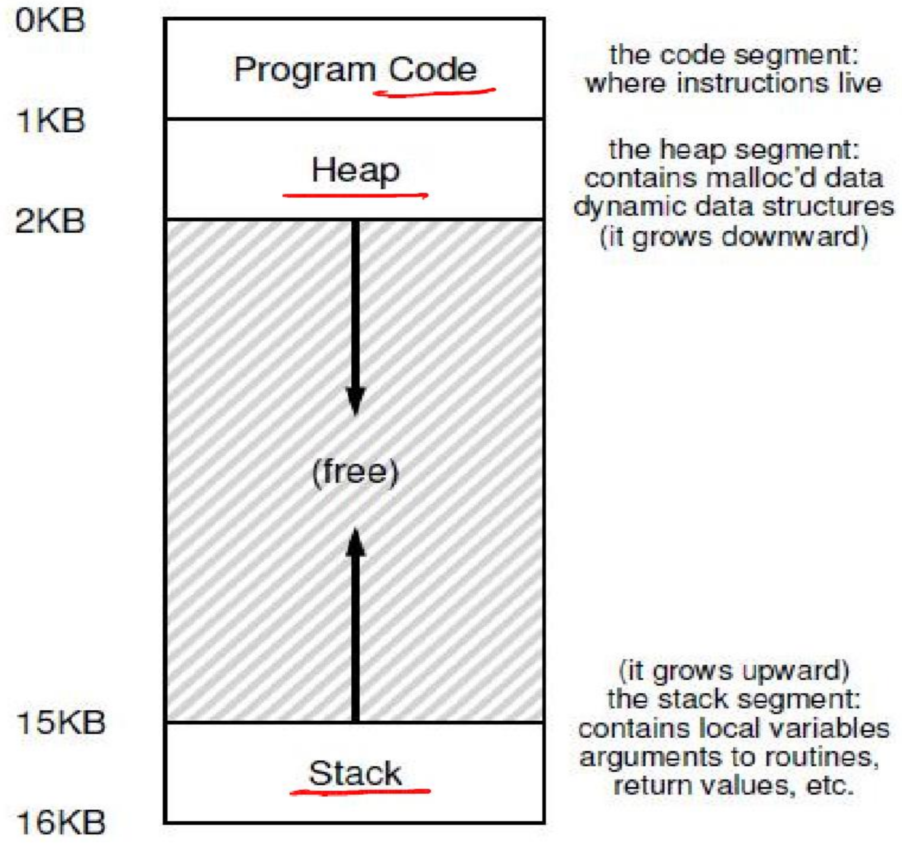
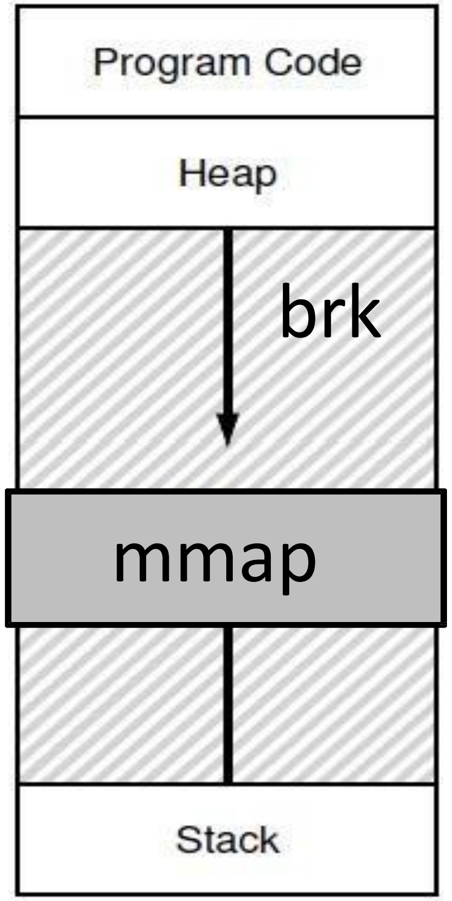

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.

Figure 13.3: **An Example Address Space**

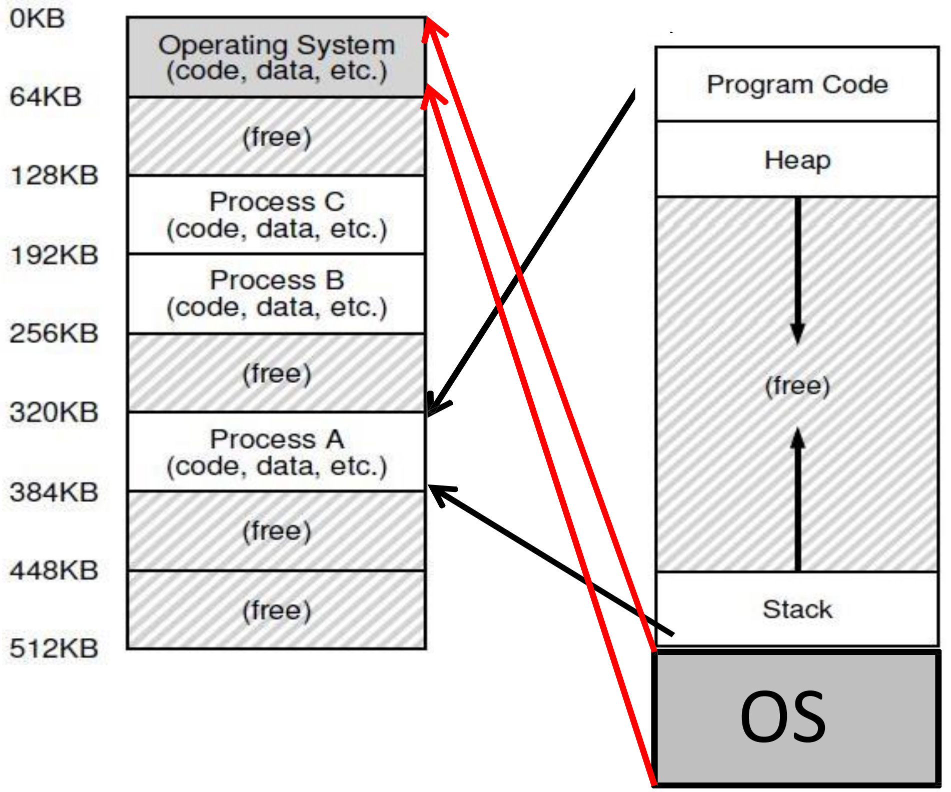
Memory allocation system calls

- `malloc` implemented by C library
  - Algorithms for efficient memory allocation and free space management
- To grow heap, libc uses the `brk/sbrk` system call
- A program can also allocate a page sized memory using the `mmap()` system call
  - Gets “anonymous” page from OS



A subtle point: what is the address space of the OS?

- OS is not a separate process with its own address space
- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., library)
- Page tables map the OS addresses to OS code


0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)
Program Code
Heap
(free)
Stack
OS

How does the OS allocate memory?

- OS needs memory for its data structures
- For large allocations, OS allocates a page
- For smaller allocations, OS uses various memory allocation algorithms (more later)
    - Cannot use libc and `malloc` in kernel!

A simple example

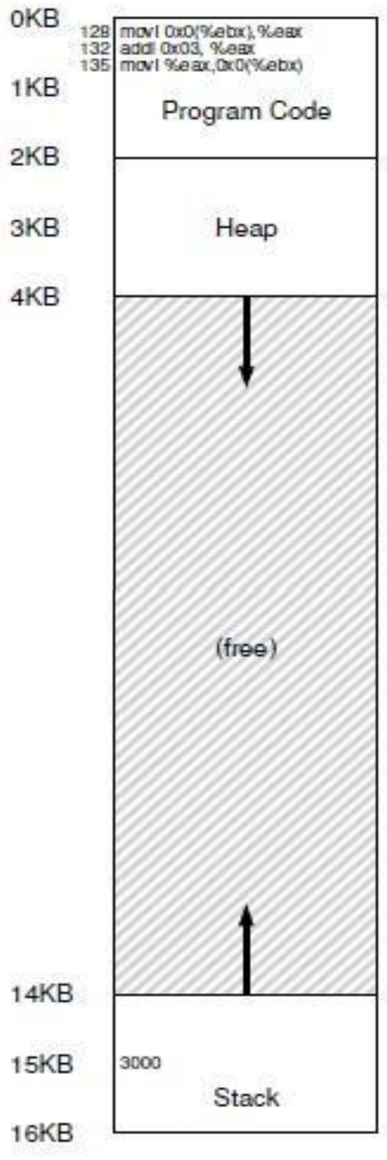
- Consider a simple C function

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

- It is compiled as follows

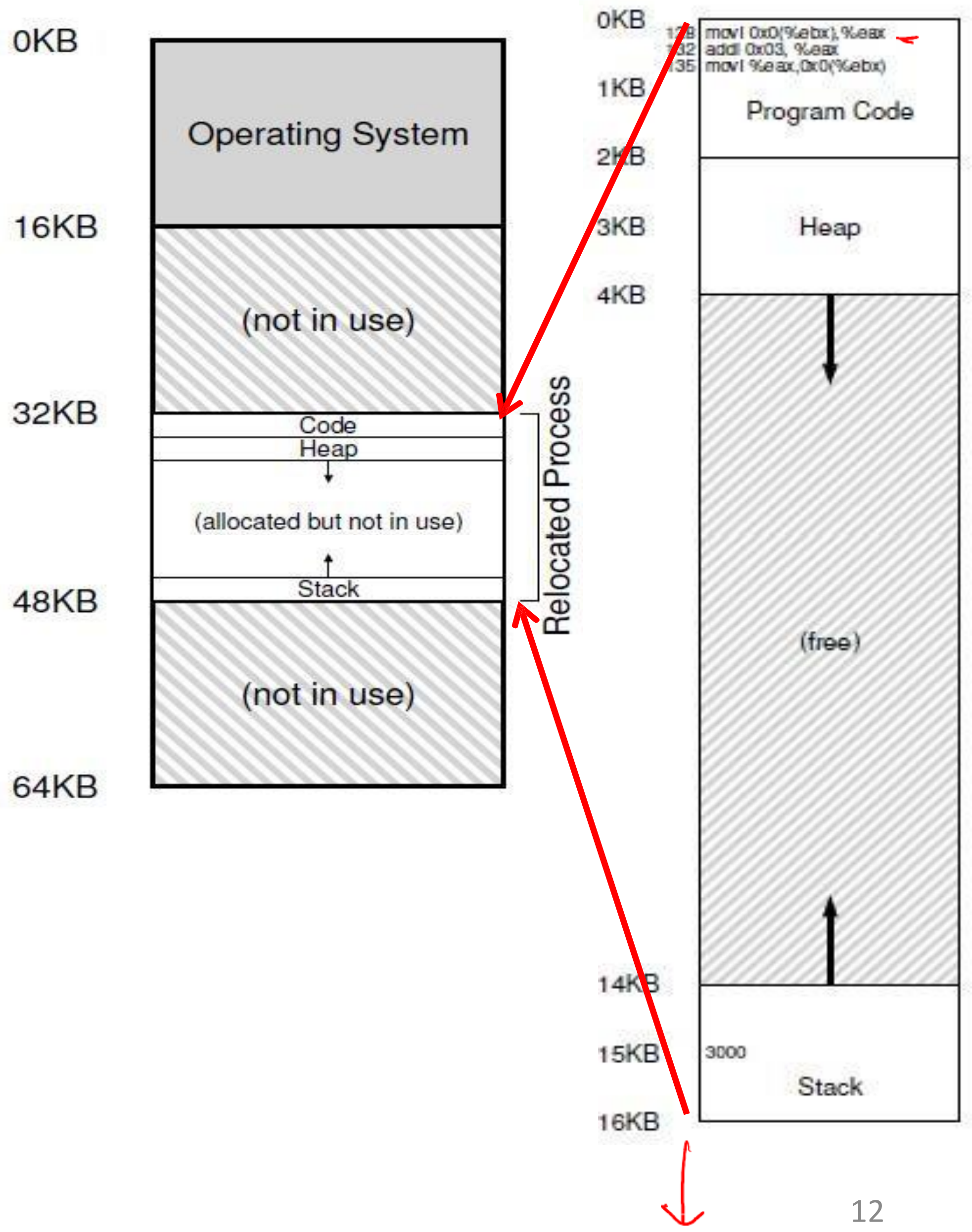
```
128: movl 0x0(%ebx), %eax      ;load 0+ebx into eax
132: addl $0x03, %eax          ;add 3 to eax register
135: movl %eax, 0x0(%ebx)      ;store eax back to mem
```

- Virtual address space is setup by OS during process creation



Address Translation

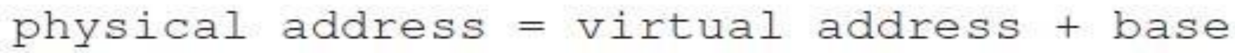
- Simplified OS: places entire memory image in one chunk
- Need the following translation from VA to PA
  - 128 to 32896 (32KB + 128)
  - 1KB to 33 KB
  - 20KB? Error!



```
128 movl 0x0(%ebx),%eax
132 addl 0x03, %eax
135 movl %eax,0x0(%ebx)
```

Who performs address translation?

- In this simple example, OS tells the hardware the base (starting address) and bound (total size of process) values
- Memory hardware Memory Management Unit (MMU) calculates PA from VA



```
physical address = virtual address + base
```

- MMU also checks if address is beyond bound
- OS is not involved in every translation

Role of hardware in translation

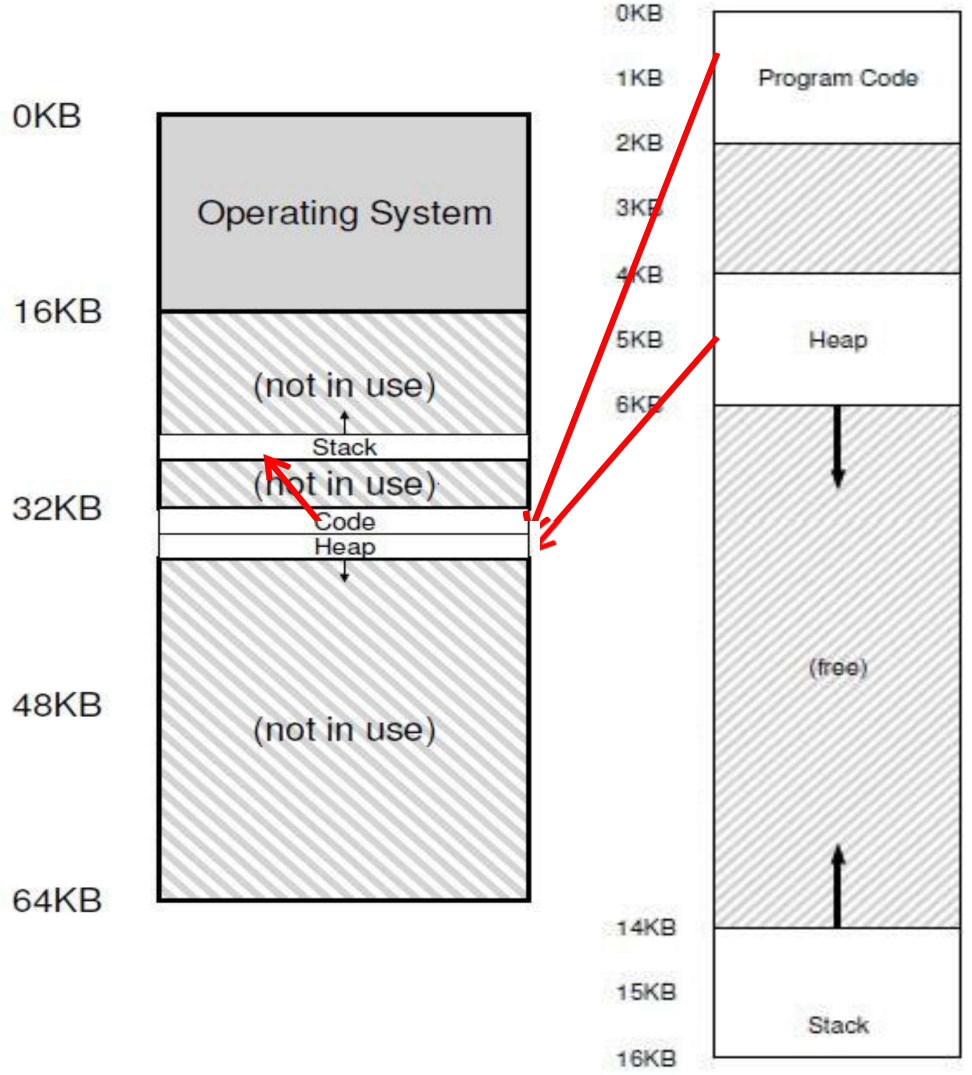
- CPU provides privileged mode of execution
- Instruction set has privileged instructions to set translation information (e.g., base, bound)
- Hardware (MMU) uses this information to perform translation on every memory access
- MMU generates faults and traps to OS when access is illegal (e.g., VA is out of bound)

Role of OS in translation

- OS maintains free list of memory
- Allocates space to process during creation (and when asked) and cleans up when done
- Maintains information of where space is allocated to each process (in PCB)
- Sets address translation information (e.g., base & bound) in hardware
- Updates this information upon context switch
- Handles traps due to illegal memory access

Segmentation

- Generalized base and bounds
- Each segment of memory image placed separately
- Multiple (base, bound) values stored in MMU
- Good for sparse address spaces
- But variable sized allocation leads to external fragmentation
  - Small holes in memory left between segments



Thank you!!!!