IT308: Operating Systems

Virtual memory: Address Translation

Address Space

- Address space: Each process' view of its own memory range
  - Set of addresses that map to bytes
- Problem: how can OS provide illusion of private address space to each process?
- Address space has static and dynamic components
  - Static: Code and some global variables
  - Dynamic: Stack and Heap

Address Translation

- Hardware does [virtual address (provided by instruction) $\rightarrow$ physical address] translation
  - translate each and every memory reference
- OS manages memory by setting up hardware
  - to create illusion that each program has its own private memory, even though many programs are sharing memory at the same time
- Create useful, powerful, and easy to use abstraction

Example

- Assumptions:
  - process' address space must be placed contiguously in physical memory
  - its size is less than the size of physical memory
  - each address space is exactly the same size

Question

```
void func() {
    int *x = (int *) malloc(sizeof(int));
}
```

Where are we allocating memory?

1. Heap
2. Stack
3. Stack and Heap
4. None of the above

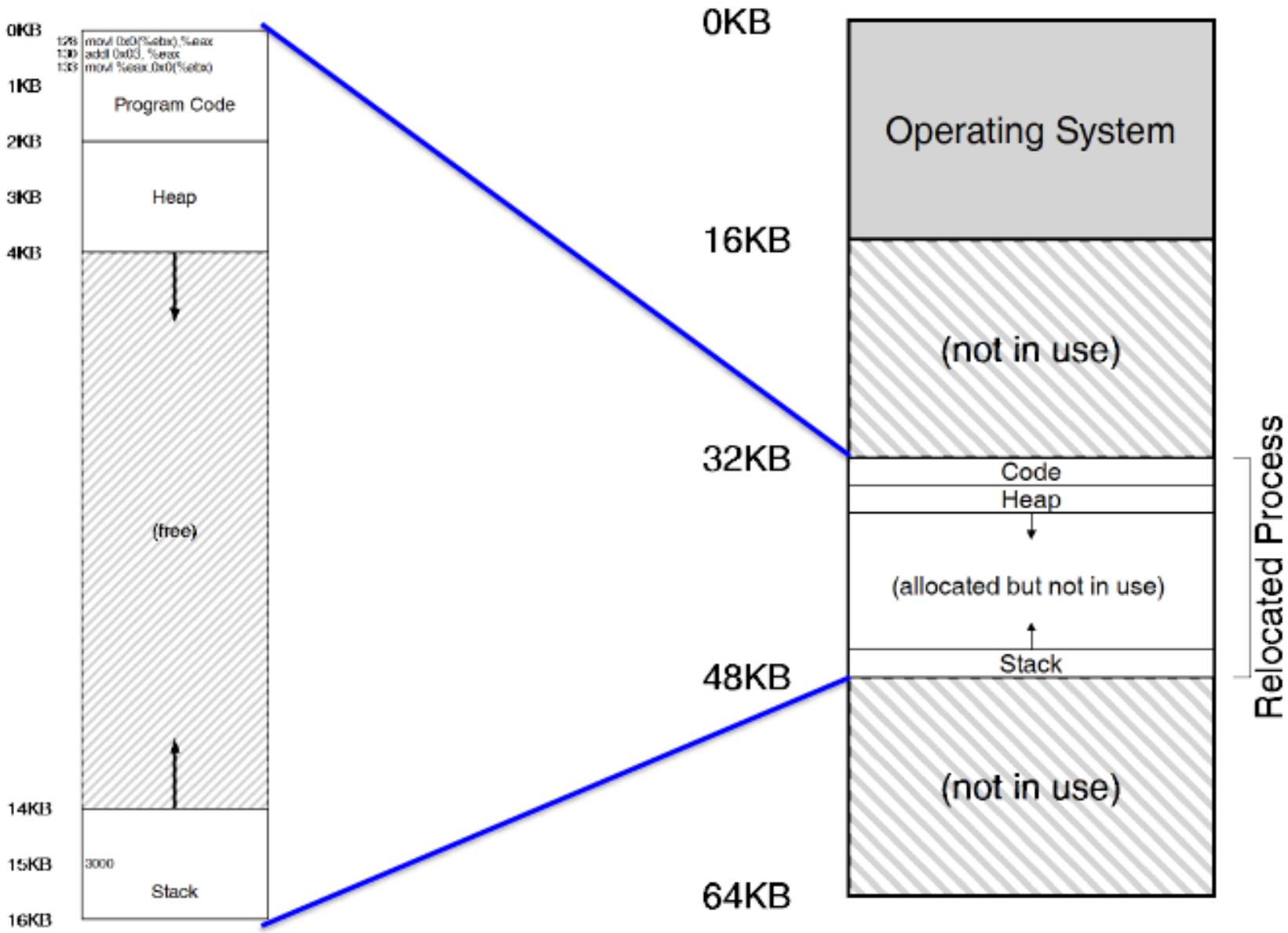
Interposition for Transparency

```
int x = 3000;
x = x + 3;

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

- (virtual) address space starts from address 0
- To virtualize memory, OS needs to place the process somewhere else in physical memory
- Problem: how can OS relocate process in memory that is transparent to the process?
- Solution: hardware interposes on each memory access, and translates each virtual address issued to a physical address

Transparent Address Relocation



Transparent Address Relocation

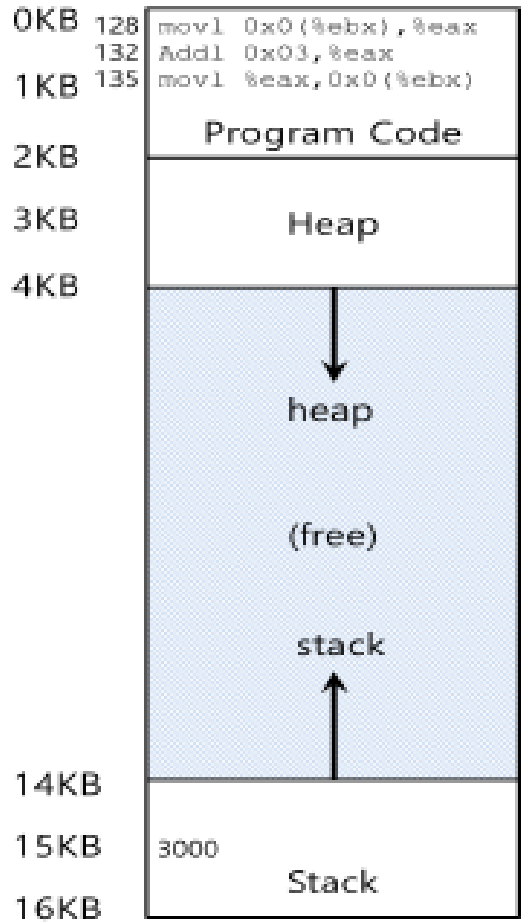
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



Dynamic (H/W-based) Relocation

- Base and bounds registers (parts of Memory management unit (MMU) inside CPU)
- Each program is written and compiled as if it is loaded at address 0
- When a program starts running, OS decides where in physical memory it should be loaded and sets the base register to that value

Dynamic (H/W-based) Relocation

- Base and bounds registers (parts of Memory management unit (MMU) inside CPU)
- Each program is written and compiled as if it is loaded at address 0
- When a program starts running, OS decides where in physical memory it should be loaded and sets the base register to that value
- Address translation (happens at runtime):

  physical address = virtual address + base

Dynamic (H/W-based) Relocation

- Base and bounds registers (parts of Memory management unit (MMU) inside CPU)
- Each program is written and compiled as if it is loaded at address 0
- When a program starts running, OS decides where in physical memory it should be loaded and sets the base register to that value
- Address translation (happens at runtime):

  $$\text{physical address} = \text{virtual address} + \text{base}$$

- What is Bounds for?

Dynamic (H/W-based) Relocation

- Base and bounds registers (parts of Memory management unit (MMU) inside CPU)
- Each program is written and compiled as if it is loaded at address 0
- When a program starts running, OS decides where in physical memory it should be loaded and sets the base register to that value
- Address translation (happens at runtime):

  $$\text{physical address} = \text{virtual address} + \text{base}$$

- What is Bounds for?
  - protection

Static (S/W-based) Relocation

- take an executable that is about to be run and rewrites its addresses to the desired offset in physical memory
- e.g., movl 1000,%eax, and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (movl 4000,%eax)
- does NOT provide protection
- happens at load time

Question

If the process is not running, can the OS change the base and bounds and move the address space in physical memory?

1. Yes
2. No, then all of the pointer values in the program would be wrong

Involvement of OS

OS must take action:

- when a process is created, finding space for its address space in memory
- when a process is terminated, reclaiming all of its memory for use in other processes
- when a context switch occurs (save/restore base-bounds register values to/from PCB)

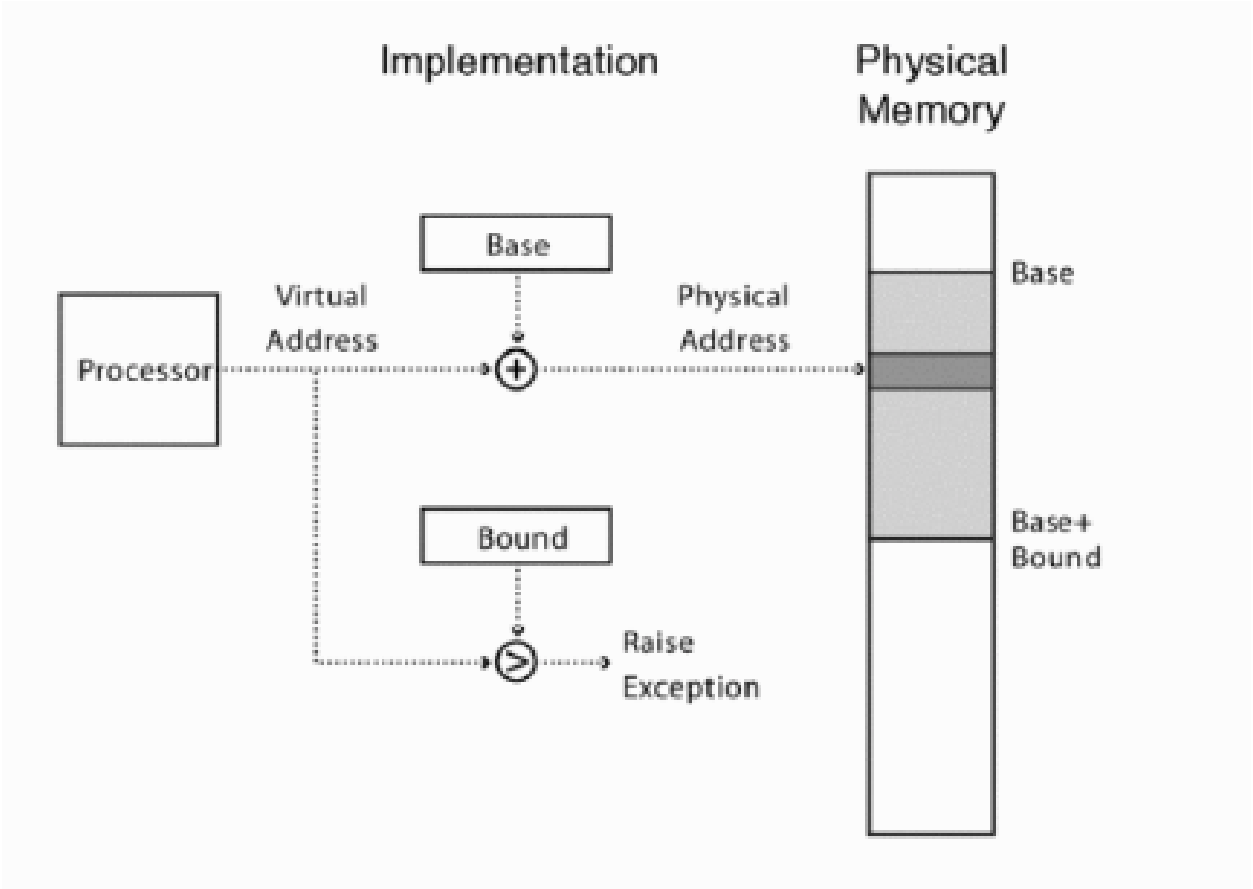
Question

What would happen if the process could modify its base and bounds values?

1. The process could read other program's memory
2. The process could issue reads for memory that doesn't exist
3. The process could corrupt the OS
4. All of the above

Base and Bound

- Access to base and bounds registers is privileged (only OS can change them)



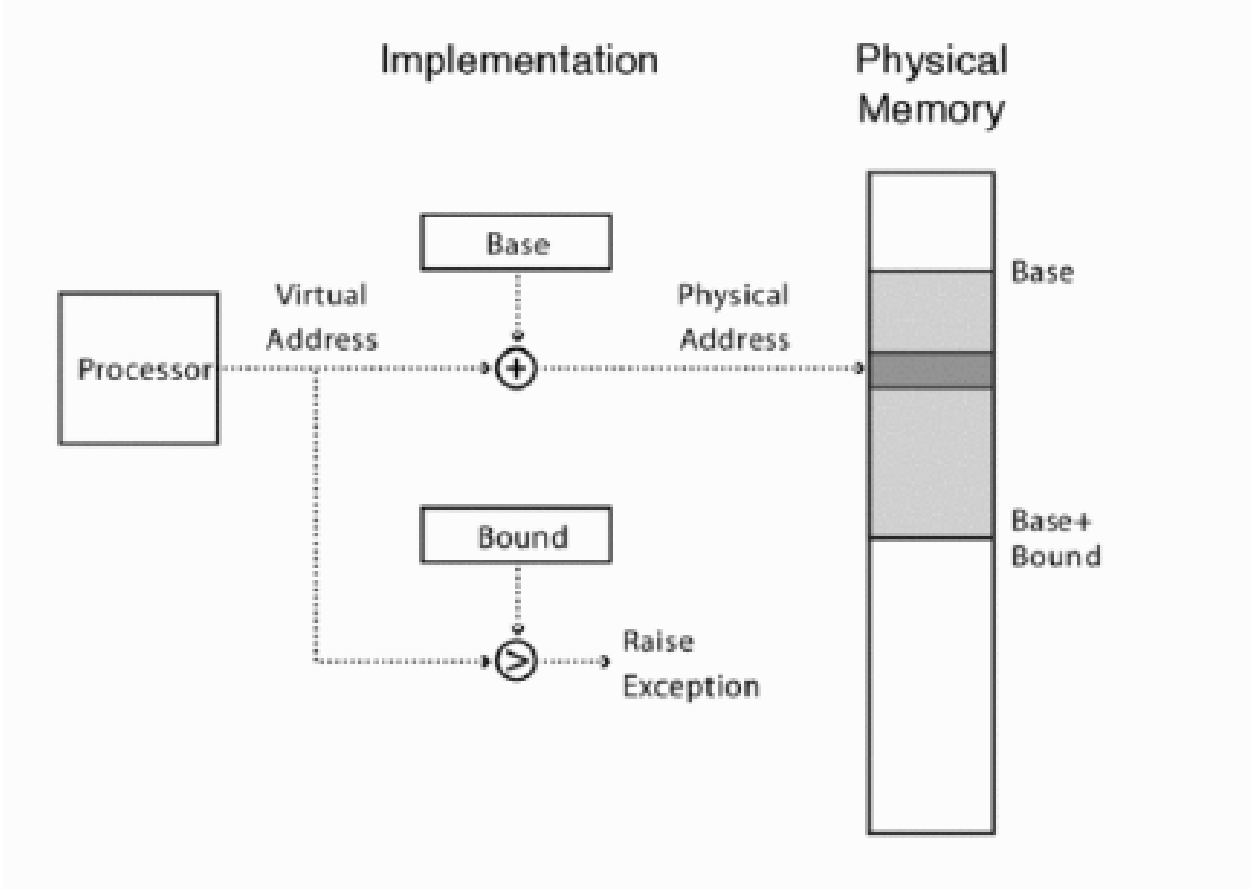
Question

What happens if the process attempts to read memory beyond the bound?

1. It will get a null value
2. The MMU will issue an exception to the OS
3. The CPU will crash the program itself
4. Your computer will halt and catch fire

Base and Bound

- Exception raised for out-of-bound reference



Involvement of OS

- when a process is created, finding space for its address space in memory
- when a process is terminated, reclaiming all of its memory for use in other processes or the OS
- when a context switch occurs (save/restore base-bounds register values to/from PCB)