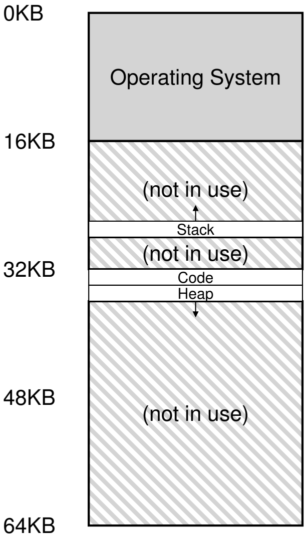


CS 250

OPERATING SYSTEMS

Lecture 7

Address Translation

Base/Bounds

Segmentation

Instructor

Dr. Dhiman Saha

- How can we build an efficient virtualization of memory?
- How do we provide the flexibility needed by applications?
- How do we maintain control over which memory locations an application can access?
- How do we do all of this efficiently?

Assumptions

1. Users address space is placed contiguously in physical memory.
2. Size of the address space is not too big
3. It is less than the size of physical memory
4. Each address space is exactly the same size

Example

```
void func() {
    int x = 3000;
    x = x + 3;    // point of interest
...
```

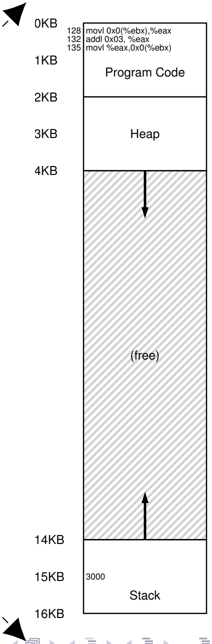
Behind the scenes — Example

```
void func() {
    int x = 3000;
    x = x + 3;     // point of interest
...
```

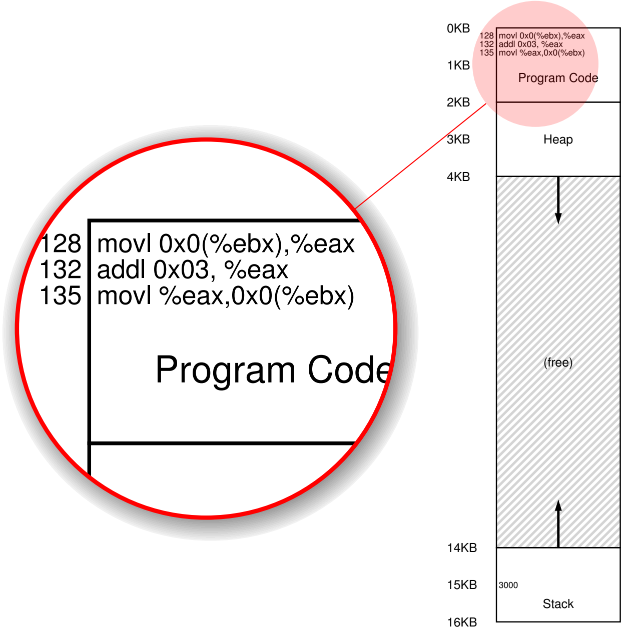
- Using `objdump` (in x86 assembly)

```
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
```

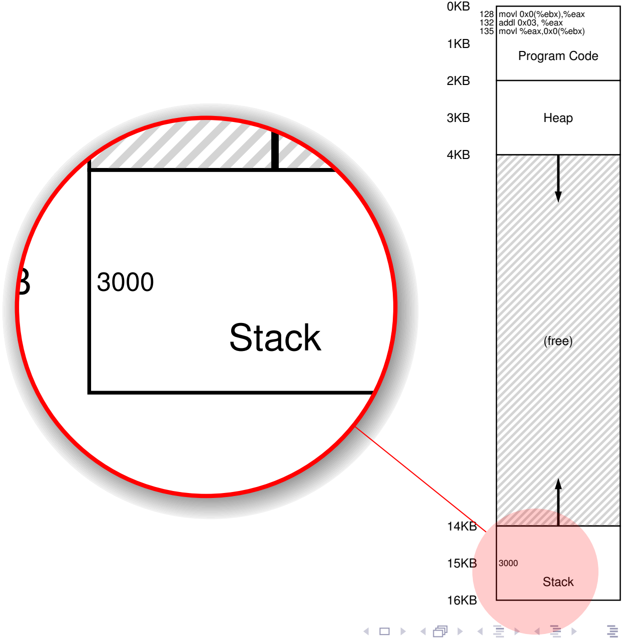
The Address Space



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



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



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

As seen by the process

```
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
```

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- 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)

As seen by the process

- `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`

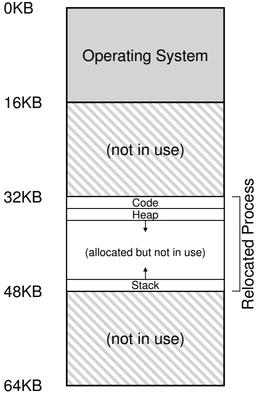
- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction **(no memory reference)**
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

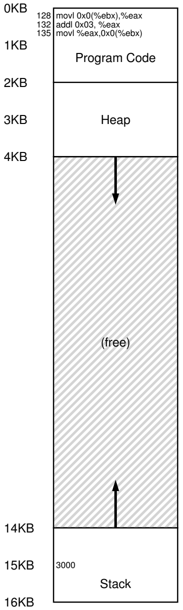
As seen by the process

```
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
```

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- 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)

Physical Memory with a Single Relocated Process





Attempt #1: Software-Based Relocation

Static Relocation

Static Relocation

Idea

Rewrite the program itself before **loading** it as a process

- Using software support: **loader**
- The **loader** takes an executable that is about to be run
- **Rewrites** its **addresses**
- To the desired **offset** in physical memory

Classwork

- How would it effect our example program?
- What if there are multiple processes?

Issues?

Why?

- Protection
- Re-Relocation

Attempt #2:
Hardware-Based Relocation
The Base Register

Dynamic Relocation

Dynamic Relocation

Idea

- Address translation by adding a fixed offset.
- Offset stored in *Base* Register
- Base register has different value for each process

- OS tells the hardware the base (starting address)
- Memory hardware calculates PA from VA
- “dynamic relocation”

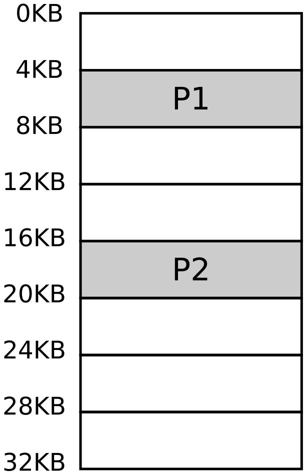
```
physical address = virtual address + base
```

Note

Each program is written and compiled as if it is loaded at address zero.

Let us try an example

- Two Process Scenario
- To Do: Address Translation

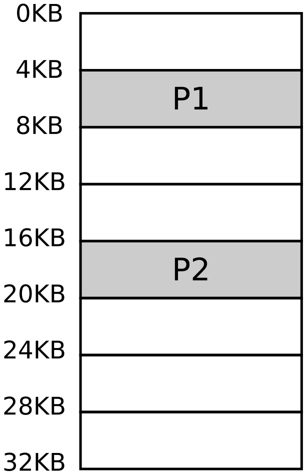


Issues?

Base Register

How?

- Protection



Attempt #3:
Hardware-Based Relocation
Base + Bounds

Dynamic Relocation

Base + Bounds

Idea

- "Bound" the address space
- The largest addressable physical address for a process
- Stored in *Bounds (limit)* register

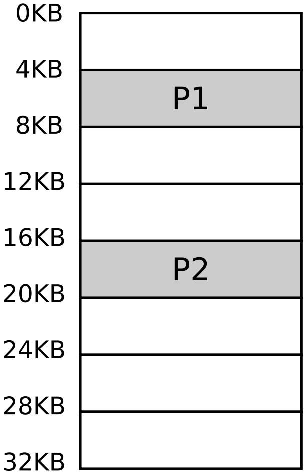
- Base → translate the address
- Bounds → ensure physical address lies within address space

Classwork

- What can bounds-register actually store?
- Where are these registers located?

Let us try an example

- Two Process Scenario
- To Do: Illegal Memory Access



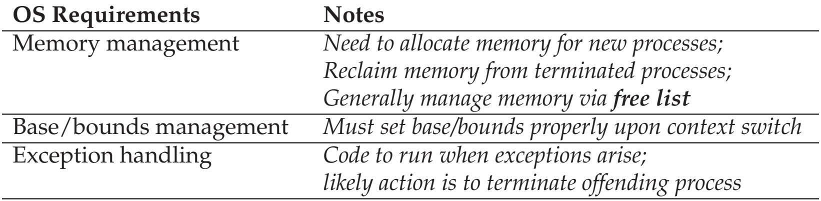
The Free-List

- A special data-structure used by OS
- To track which parts of free memory are not in use
- Simply is a list of the ranges of the physical memory which are not currently in use

Hardware Support

Hardware Requirements	**Notes**
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

OS Responsibilities



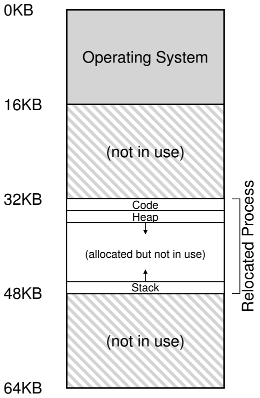
OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via **free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

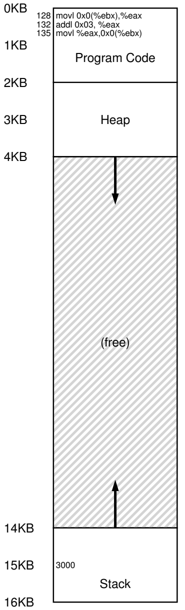
Classwork

Can the OS perform **address space relocation** when a process not running? How?

- Look at LDE protocol with dynamic relocation in OSTEP book.

Issues?





- 1. Internal Fragmentation
- 2. What if address space does not fit into memory?

Attempt #4: Hardware-Based Relocation Segmentation

Generalized Base/Bounds

Segmentation

Generalized Base/Bounds — Idea

Instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical **segment** of the address space?

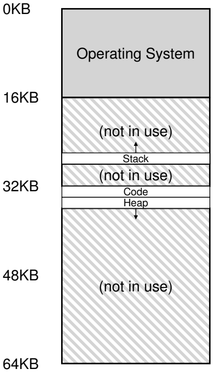
Segment

A segment is just a **contiguous** portion of the address space of a particular length.

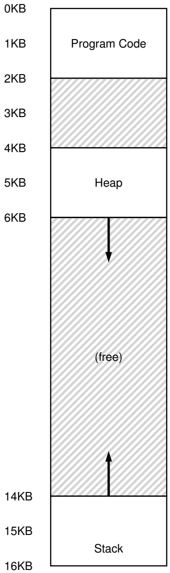
- Code
- Stack
- Heap

Three logically-different segments. How to utilize this setting?

Placing Segments In Physical Memory



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

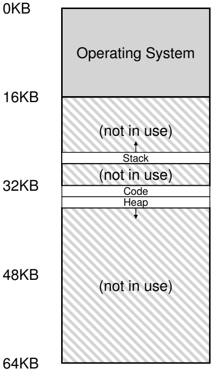


Hardware Support in MMU

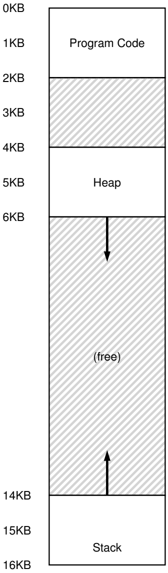
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

- A set of three base and bounds register pairs

Let us try an example

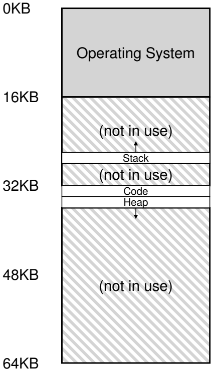


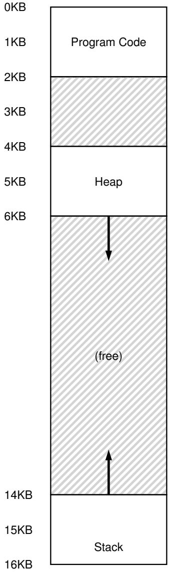
- VA: 135 PA:____________?



Address Translation

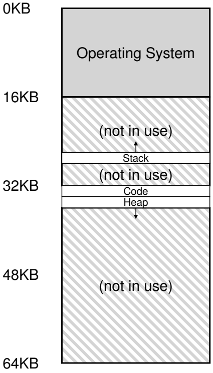
Let us try an example

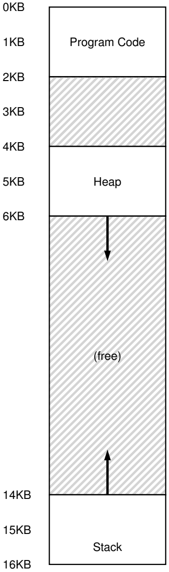




- VA: 4400 PA:____________?

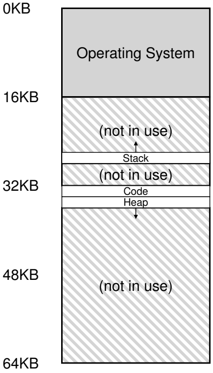
Address Translation Let us try an example

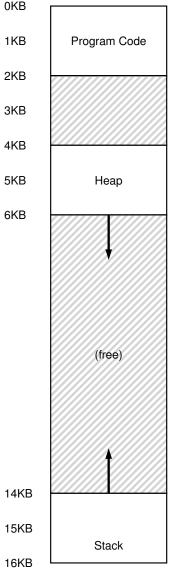




- VA: 7KB PA:____________?

Let us try an example



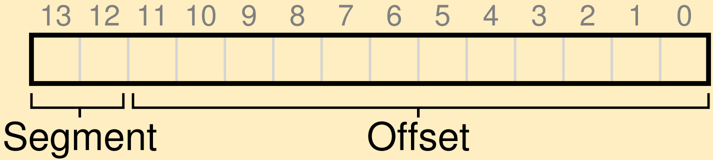


- VA: 7KB PA:____________?
- Segmentation Fault

Which Segment Are We Referring To?

Which Segment Are We Referring To?

Explicit Approach



Classwork

Calculate this for VA:4400

Hardware Support

```
// 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)
```

Classwork

Calculate

- `SEG_MASK`
- `SEG_SHIFT`
- `OFFSET_MASK`

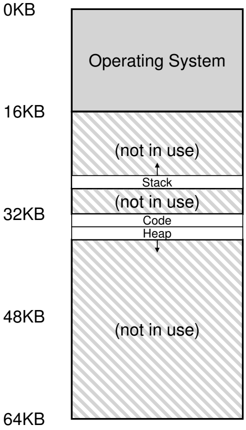
Which Segment Are We Referring To?

Implicit Approach

The hardware determines the segment by noticing **how the address was formed**

- Address generated from PC $\Longrightarrow$ code segment
- Address based of stack pointer $\Longrightarrow$ stack segment
- Otherwise $\Longrightarrow$ heap segment

Did we forget the stack?



Recall

Stack grows backwards

Negative-Growth Support

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

Segment Registers

HW-4

How will address translation take place now?