Memory isolation

- Processes have their own reserved quota of RAM
  - also called the address space of the process

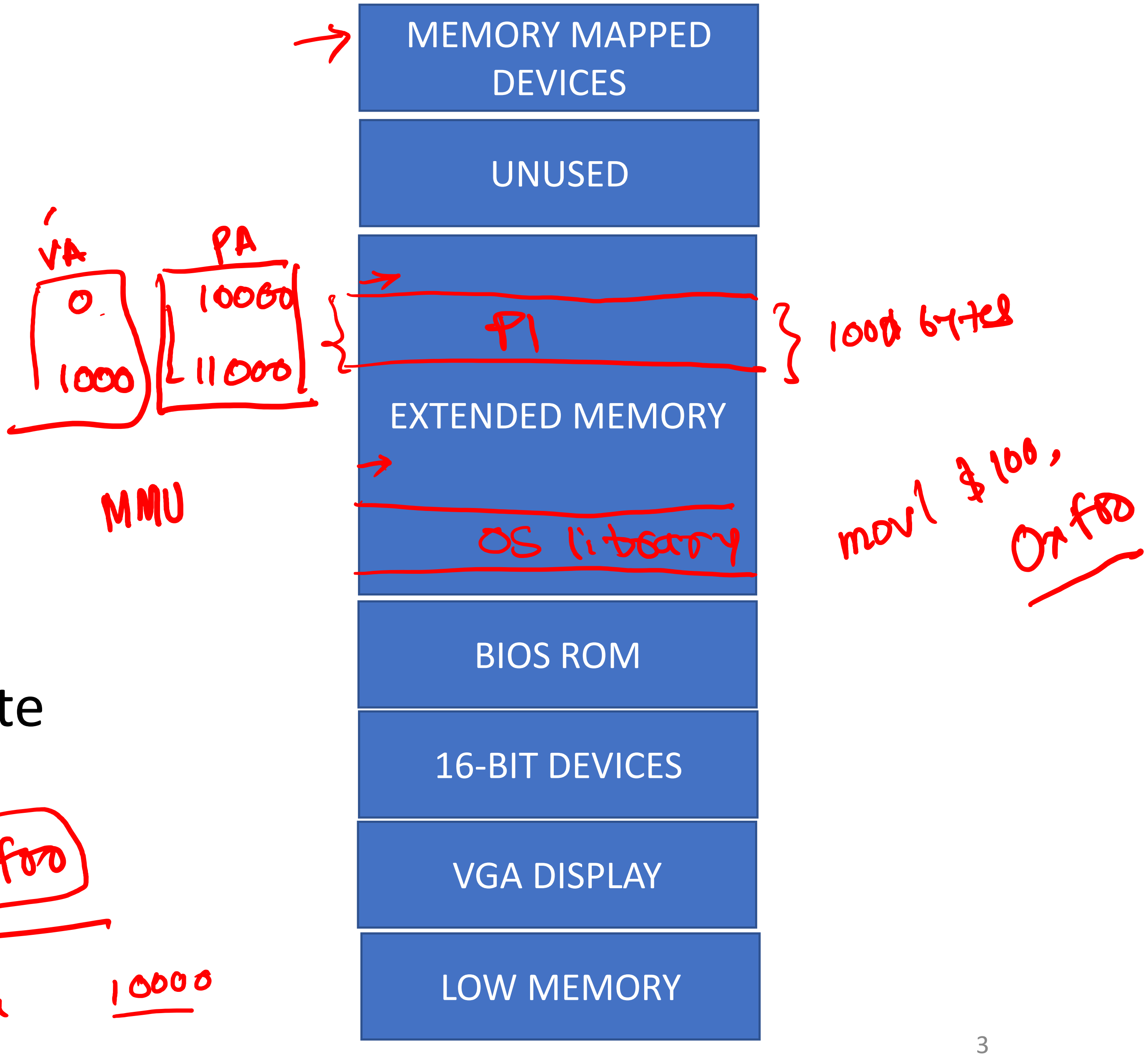
- Decided by OS during fork/exec

Memory map

- The compiler does not know actual RAM addresses in advance
- If we only allow physical addresses, the compiler is unnecessarily forced to generate relocatable code

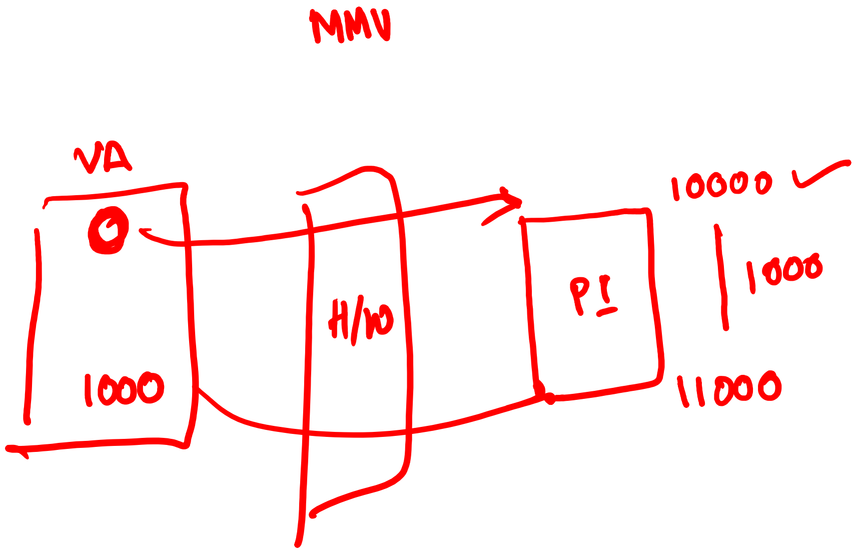
movl $100, 0xf00

a 10000



Virtual addresses

- Uniform among all processes
- Converted into physical address (PA) by hardware during memory access
- Virtual address (VA) range is 0 to size of process reserved quota in RAM
- PA is computed using adding the base of the process address space to VA



Segmentation

- To facilitate address translation CPU has an additional unit called MMU

- In addition to general purpose registers, x86 also have six segment registers namely, %cs, %ds, %ss, %es, %fs, and %gs

Segmentation

- Every memory operand is prefixed by some segment register

- Let us assume for now that %cs segment register is used for all memory operands

%eax

movl $100, %ds:(%eax)

mov $100, %fs:(%eax)

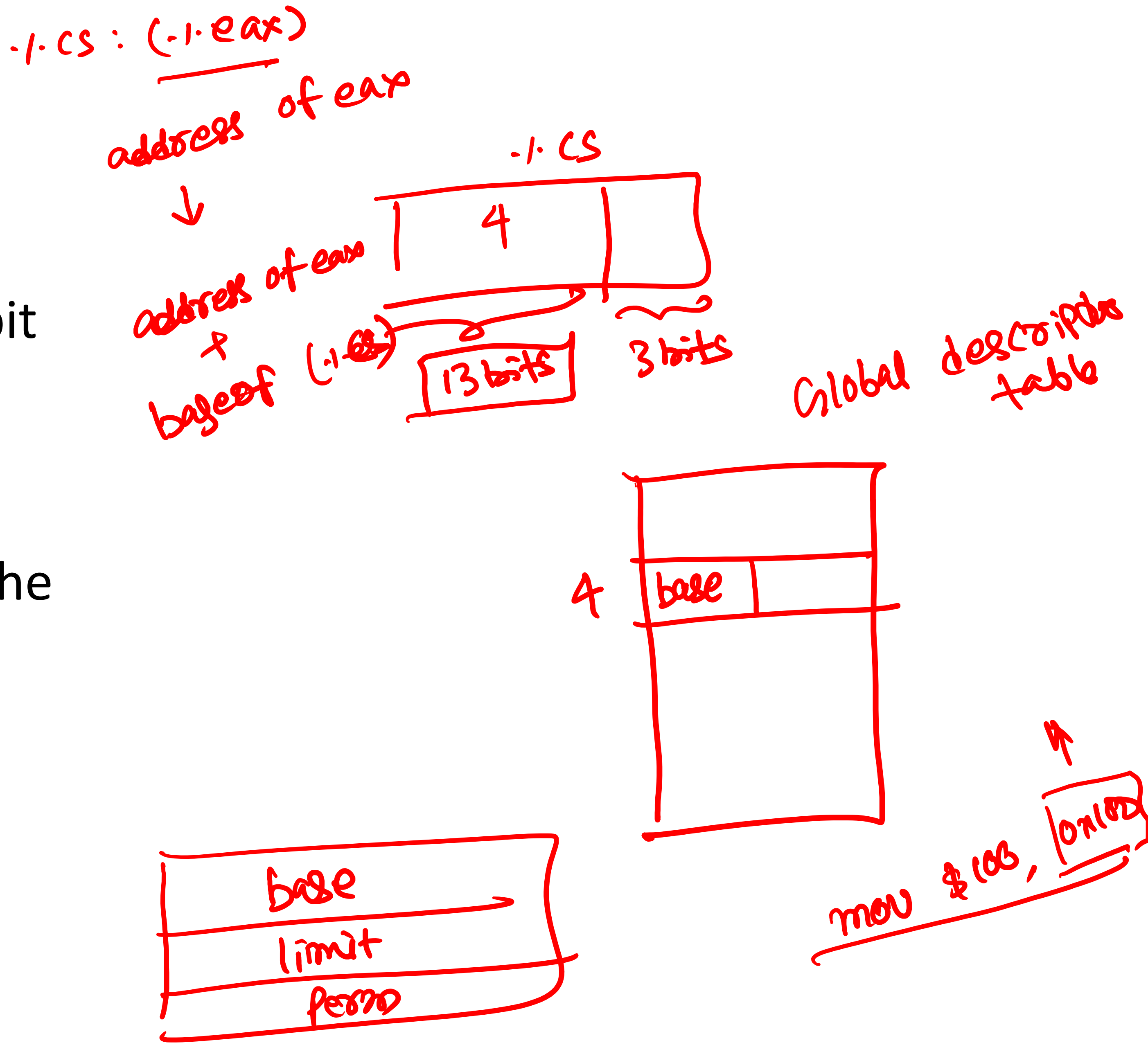
%cs:(%eax)

Segmentation

- Segment registers contain a 16-bit value that is used to index in a global descriptor table (GDT)

- GDT is an array of structures of the following type.

```
struct {
    unsigned base, limit, perm;
};
```



Segmentation

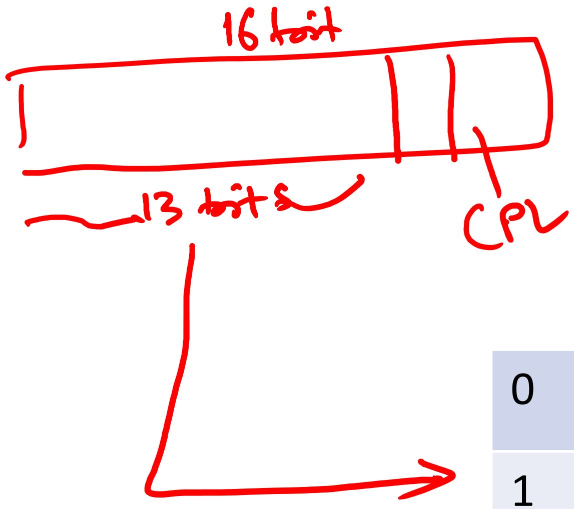
- The MMU unit adds the base of the descriptor to the virtual address to compute the physical address during a memory access
- If the virtual address exceeds a limit, the hardware generates an error

```
movl $100, %cs:100
---------------------------
index = cs >> 3; // cs is 16-bit
base = gdt[index].base;
unsigned *phys = base + 100;
assert (100 < gdt[index].limit);
*phys = 100
```

GDT

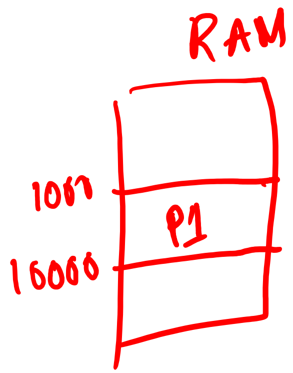
```
mov 0, %ax
mov %ax, %ds
movl $1, %ds:100
%ds:100 =     1000 + 100
assert (100 < 10000)
```

```
mov 16, %ax
mov %ax, %ds
movl $1, %ds:100
%ds:100 =     200 + 100
assert (100 < 256)
```



	base	limit
0	1000	10000
1	30000	31000
2	~~200~~ 0	~~256~~ 0xffffffff
3	..	..
4	..	..
5	..	..
6	..	..
7	..	..

system/supervisor



Who creates GDT?

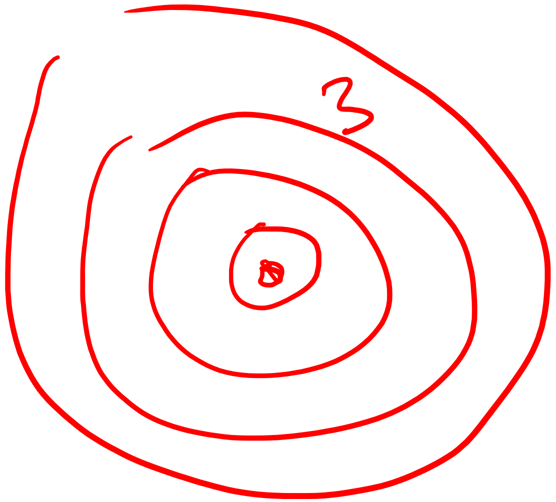
- OS creates GDT

- OS creates GDT in RAM and executes "lgdt" instruction to load the address of GDT in GDTR register

- Why can't user programs do the same thing?

Protection rings

- X86 has 4 protection rings (0,1,2, and 3)
- 0 is most privileged and, 3 is the least privileged mode
- OS executes in most privileged mode
- User programs execute in the least privileged mode



Protection rings

- Some instructions are only allowed in most privileged mode

- Some instructions have different semantics in different privileged mode

- “lgdt” is only allowed in most privileged mode

How does hardware know the current privilege level (CPL)?

- The last two bits of cs cs segment register contains the CPL
  - For OS CPL == 0
  - For user programs CPL == 3

What prevent applications from modifying the GDT entries?

- GDT lives in OS memory

How does OS access the entire memory?

- OS segment selectors point to a GDT entry with base and limit set to 0 and 2^{32} respectively
- Why user programs cannot set their segment registers to OS GDT entries
  - Supervisor (system) flag in the GDT entry

ds - 0

ds - (2<<3)

1000	2000	0
0	0xffffffff	1 ✓

ljmp $cs : eip

Context switch

- A dedicated GDT for every process
- Load the target process' GDT on every context switch
  - execute lgdt

P2 lgdt Y

P1 lgdt X

X P1

0	1000	2000	0
8	~~3000~~	~~4000~~	~~0~~
✓	0	0xffffffff	1

~~3000~~

P1 :
$ds ← 0

P2 :
$ds ← 8

Y P2

	3000	4000	0
✓			
✓	0	0xffffffff	1

Context switch

- Only one GDT
- Overwrite the GDT entries with target process entries on every context switch
  - modify GDT table itself

2000 ~~1000~~	3500 ~~2000~~	0
0	0xffffffff	1

Default segment registers

- %cs is default for EIP

- %ss is default of stack
  - e.g., push, pop

- %ds is default for most of memory operands
  - e.g., movl $100, (%eax)   /* default ds */

- %es is default for string instructions   %es
  - e.g., movsb

- %ds can be overridden by other segment registers
  - e.g., movl $100, %fs:(%eax) /* use fs instead of ds */

Exceptions

- Divide by zero
  - CPU executes divide by zero

- General protection fault
  - Process tries to access a VA outside its limit

- and many more …

- 0 – 32 interrupt vectors are reserved for exceptions

Interrupt handler

- IDT is an array of structures of 256 entries

- IDT contains cs:eip pairs of interrupt handlers

```
struct {
    unsigned interrupt_handler;
    unsigned short cs;
    …
};
```

Who creates IDT?

- OS creates IDT

- OS creates IDT in its own address space and executes “lidt” instruction to load the address of IDT in IDTR register

- Why can’t user programs do the same thing?
    - “lidt” can only be executed in ring 0

Task state segment (TSS)

- On interrupt in user programs, the hardware automatically switch to the kernel stack

- A special structure TSS contains the address of kernel stack and kernel stack segment

Interrupt/exception user mode

```
-----
switch to kernel stack (ss:esp)
push old_ss ✓
push old_esp ✓
push old_eflags ✓
push old_cs ✓
push old_eip ✓
-----
```

Interrupt/exception in kernel mode

```
-----
push old_eflags
push old_cs
push old_eip
-----
```

Returning to user mode

```
iret
-----
pop old_eip
pop old_cs
pop old_eflags
pop old_esp
pop old_ss
-----
```

Returning to kernel mode

```
iret
-----
pop old_eip
pop old_cs
pop old_eflags
-----
```

Fork and exec

iret

-----

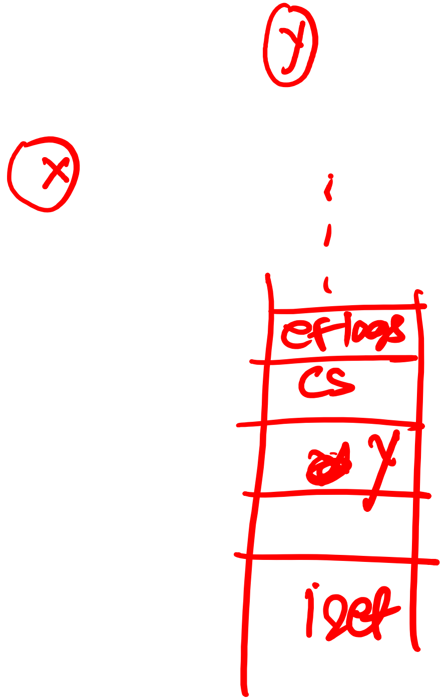
pop old_eip

pop old_cs

pop old_eflags

pop old_esp

pop old_ss



Software interrupts

GDB

main()

mov %eax, %ecx

int3

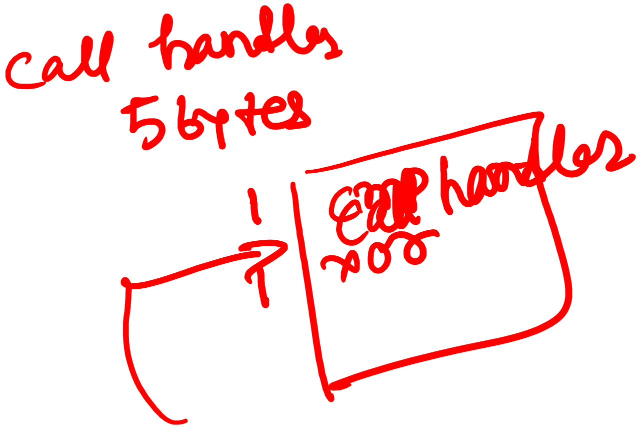
- int3
  - Generates interrupt of vector number 3
- into
  - Generates interrupt of vector number 4, if the overflow condition is set

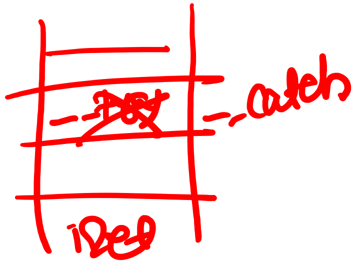
-- try {

}

catch {

}





divide_by_zero

{