xv6©-rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)
x86 Interrupt Dispatching

Carmi Merimovich

Tel-Aviv Academic College

November 30, 2017

Interrupt system

- **Before** fetching an instruction, the processor can make a state change.
- The state change should cause instruction execution stream change.
- The steam change must be **invisible** to the original stream.
- Hence, the original state should be restorable.

Interrupt request classes

- External.
    - Usually some I/O controller.
    - Usually can be masked.
- Internal.
    - E.g., Illegal address reference, illegal opcode, ...
    - Cannot be masked.

Controlling interrupt delivery

On x86:

- The **sti** instruction enables delivery of maskable interrupts.
- The **cli** instruction disables delivery of maskable (i.e., blockable) interrupts.

|			IEF	TF	SF	ZF		AF		PF		CF
31	22		9	8	7	6		4		2		0
Reserved							0		0		1	

Figure: The eflags register

Interrupt id

- Usually interrupts are identified by a number.
- Interrupts on the x86 are identified by numbers in the range 0–255.
- The range 0–31 is reserved by Intel.

x86 interrupts delivery (processor in kernel mode)

Instruction stream change

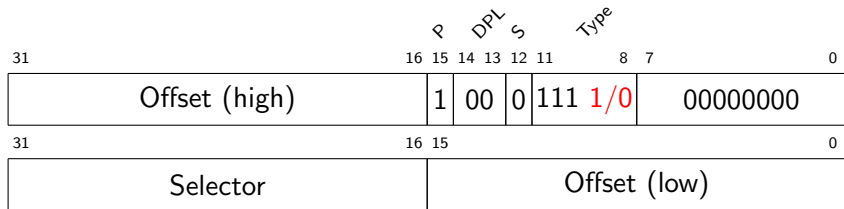If interrupt request is to be serviced:

- Where is the new instruction address to be found?
    - There is a 256 entries vectors, containing the appropriate address.
    - The name of the vector is **IDT**.
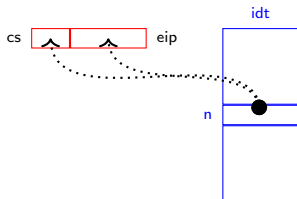
- Thus, instruction stream change entails change in which registers?
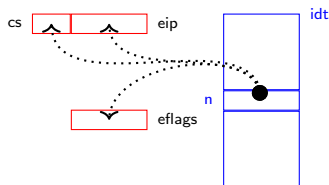   - **cs** and **eip**.

Gates

- IDT entries are called **gates**.
- We are interested in the interrupt and trap gates:

	P	DPL	S	Type			
31	16 15	14 13	12	11	8 7		0
Offset (high)	1	00	0	111 1/0	00000000		

31	16 15	0
Selector	Offset (low)	

- Interrupt gate: (Bit 8 is clear.) An implied CLI is executed.
- trap gate: (Bit 8 is set.) **eflags.if** is without change.


eflags also might change!

Gate in C

```
901  struct gatedesc {
       uint off_15_0 : 16; // low 16 bits of offset in segment
       uint cs : 16; // code segment selector
       uint args : 5; // # args, 0 for interrupt/trap gates
       uint rsv1 : 3; // reserved(should be zero I guess)
       uint type : 4; // type(STS_{TG, IG32, TG32})
       uint s : 1; // must be 0 (system)
       uint dpl : 2; // descriptor(meaning new) privilege level
       uint p : 1; // Present
       uint off_31_16 : 16; // high bits of offset in segment
     };

     #define SETGATE(gate, istrap, sel, off, d) { \
       (gate).off_15_0 = (uint)(off) & 0xffff; \
       (gate).cs = (sel); \
       (gate).args = 0; \
       (gate).rsv1 = 0; \
       (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
       (gate).s = 0; \
       (gate).dpl = (d); \
       (gate).p = 1; \
       (gate).off_31_16 = (uint)(off) >> 16; \
     }
```

Where is the IDT located?
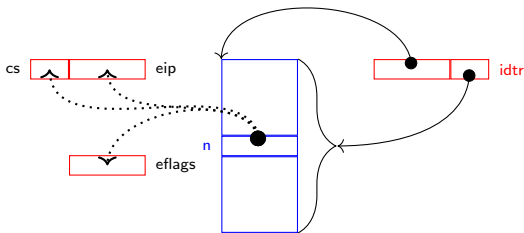
The **idtr** register.

| 47	1615	0
Address		Limit

IDT location setting in C

```
3361    struct gatedesc idt[256];
        //
        // initialize idt
        //
        lidt(idt, sizeof(idt));
```
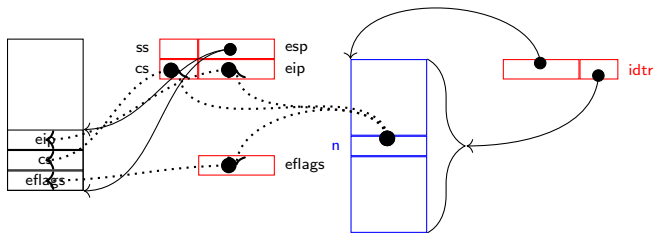
Destroyed registers

- So far, what were the registers destroyed?
  - **eip**, **cs**, **eflags**.
- Invisibe interrupt delivery (to the original stream) entails what?
  - **eip**, **cs**, and **eflags** should be saved.
- Where is a reasonable place to save the registers?
  - The stack.

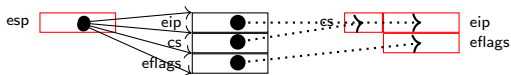# Registers saving
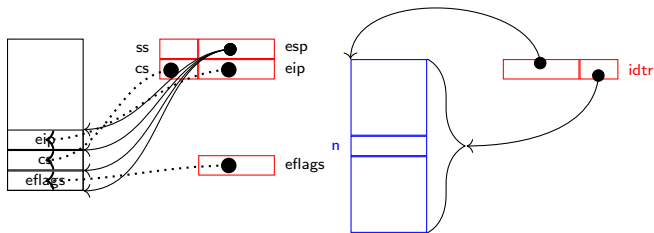
Interrupt Delivery Sequence (kernel mode)

- Note, **esp** also changes!
- This change is easily restorable.

Interrupt Service Routine

- The real work is done in the interrupt service routine.

- The ISR should restore processor state to where it was before delivery done.

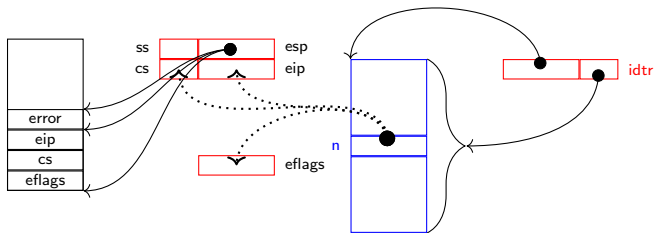- The **iret** instruction it the last instruction to be executed.

iret (kernel to kernel)

- Three words popped since $(\text{saved\_cs}\&3) == 0$!

Slight stipulation
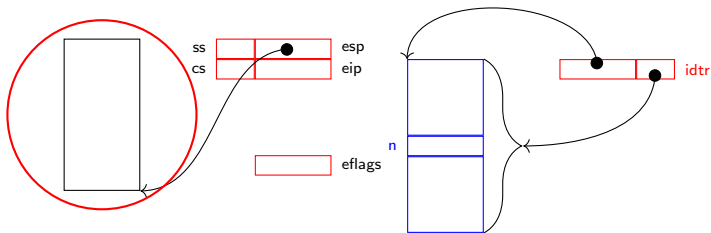

- Interrupts 8, 10-14, and 17, leave an error code on the stack.
- So, an ISR should terminate with:

```
addl $4,%esp
iret
```

Interrupt delivery, processor in user mode

Registers saving

- What is the problem?
- We have user mode stack! Not trustable!!!
- So we must have safe stack to work with.

Hypothetically

- Assume the hypothetical registers **safeesp** and **safess** exist.
- Moreover, assume **safeesp** and **safess** are privileged registers.
- Interrupt delivery in user mode, **switches** to this safe stack.
- Thus, the kernel, **before** switching to user mode, loads these registers.

Hypothetical registers saving (begining in user mode)
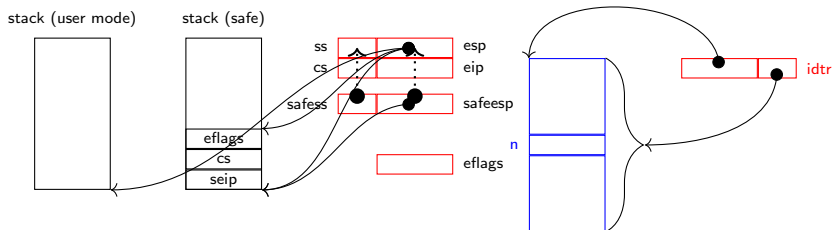
- Where is the problem?
- We **lost** the pointers to the user mode stack!!!!

Fixed hypothetical interrupt dispatch (begining in user mode)
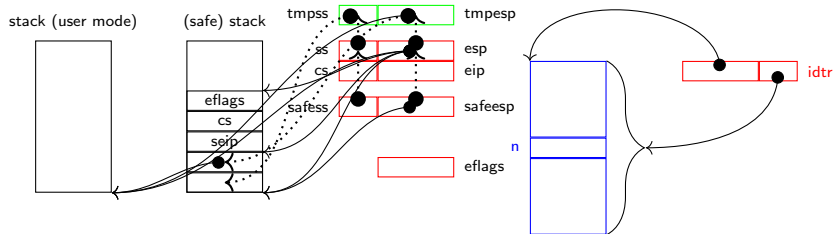
- The x86 has no **safess** and **safeesp** registers.
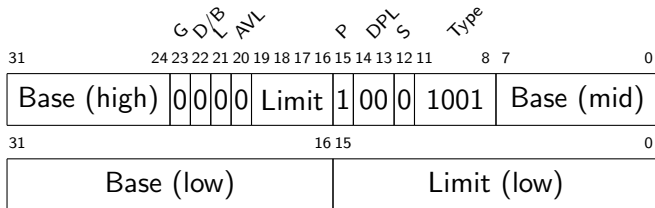- The location of the safe stack in a data structure named the TSS.

TSS

31	16	15	0
I/O Map Base Address		Reserved	T
Reserved		LDT segment selector	
Reserved		GS	
Reserved		FS	
Reserved		DS	
Reserved		SS	
Reserved		CS	
Reserved		ES	
EDI			
ESI			
EBP			
ESP			
EBX			
EDX			
ECX			
EAX			
EFLAGS			
EIP			
CR3			
Reserved		SS2	
ESP2			
Reserved		SS1	
ESP1			
Reserved		SS0	
ESP0			
Reserved		Previous Task Link	

Used } (brace spanning SS0 and ESP0 rows)

The TSS location

- TSS stands for Task State **Segment**.
- Hence, like all segments, there is a descriptor for it (in the GDT),

G D/B AVL   P DPL S Type

31	24 23 22 21 20 19 18 17 16 15 14 13 12 11	8 7	0
Base (high)	0 0 0 0 Limit 1 0 0 0 1001	Base (mid)	

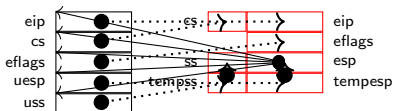| 31	16 15	0
Base (low)	Limit (low)	

How do we know the GDT index of the TSS?

- Indices to the GDT are always in a segment registers..
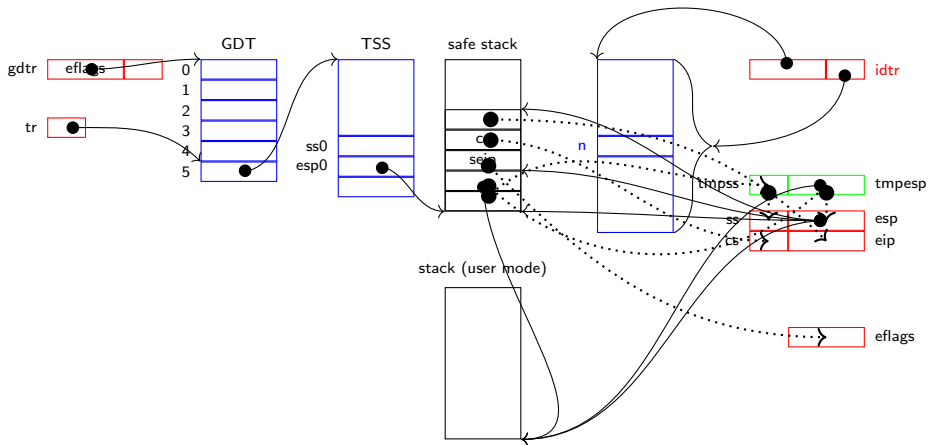- There is a privileged segment register **tr** which does exactly this.

x86 Interrupt Delivery Sequence (user mode)
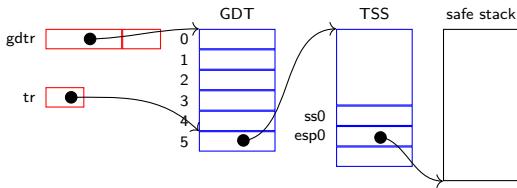
iret (kernel to user)

- Five words are popped since $(saved\_cs\&3) == 3$!
- The stipulation holds here, error is pushed for IRQs 8, 10-14, 17.

iret full context

Figuring out completely **switchuvm**

```
1860   void switchuvm(struct proc *p) {
        pushcli();
        mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A,
                                      &mycpu()->ts,
                                      sizeof(mycpu()->ts)-1, 0)
        mycpu()->gdt[SEG_TSS].s = 0;
        mycpu()->ts.ss0 = SEG_KDATA<<3;
        mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
        mycpu()->ts.iomb = (ushort) 0xFFFF;
        ltr(SEG_TSS << 3);
        if (p->pgdir == 0)
         panic("switchuvm: no pgdir");
        lcr3(v2p(p->pgdir)); // switch to new address space
        popcli();
       }
```

Figuring out **struct cpu**

```
2301  struct cpu {
       uchar apicid; // Local APIC ID
       struct context *scheduler; // swtch() here to enter
       struct taskstate ts; // Used by x86 to find stack f
       struct segdesc gdt[NSEGS]; // x86 global descriptor
       volatile uint started; // Has the CPU started?
       int ncli; // Depth of pushcli nesting.
       int intena; // Were interrupts enabled before pushc
       struct proc *proc; // The currently running proce
     };

     extern struct cpu cpus[NCPU];
```