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

Carmi Merimovich

Tel-Aviv Academic College

December 11,2016

Interrupt Serivce routines

- In xv6 the C-routine **trap()** is the basis for interrupt servicing.
- Interrupt service routines are necessarily hardware depedent, thus
  - ISRs cannot be written **totally** in C.
  - (as usual) We aim to write whatever is possible in C.
- In the following we look at what the hardware forces us to do.

Infrastructure of interrupt service routines in C

Using **trap0()** issues

- Assume **trap0** is a C routine to handle interrupt request 0.
- Issues we need to deal with:
  - Can `IDT[0]` point directly to **trap0**?
  - What about the interrupted code registers?
  - What about our segment registers?

We solve these issues one by one.

Using **trap0()** issues

- Assume **trap0** is a C routine to handle interrupt request 0.
- Issues we need to deal with:
  - Can `IDT[0]` point directly to **trap0**? Yes, but returing will crash.
  - What about the interrupted code registers?
  - What about our segment registers?

We solve these issues one by one.

Using **trap0()** issues

- Assume **trap0** is a C routine to handle interrupt request 0.
- Issues we need to deal with:
  - Can `IDT[0]` point directly to **trap0**? Yes, but returing will crash.
  - What about the interrupted code registers? Proably will be destroyed.
  - What about our segment registers?

We solve these issues one by one.

Using **trap0()** issues

- Assume **trap0** is a C routine to handle interrupt request 0.
- Issues we need to deal with:
  - Can `IDT[0]` point directly to **trap0**? Yes, but returing will crash.
  - What about the interrupted code registers? Proably will be destroyed.
  - What about our segment registers? Unexpected if was at user mode.

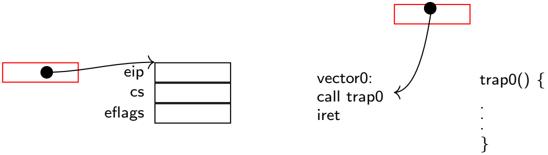
We solve these issues one by one.

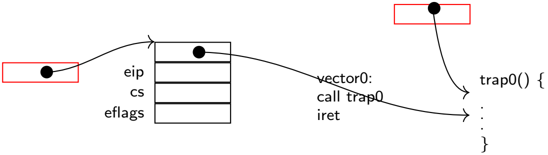
IDT[0] pointing to a good place

- **trap0()** ends with **ret**, no way around it.
- So we wrap **trap0** inside a routine ending with **iret**:

```
vector0:
 call trap0
 iret
```

- Then we let **IDT[0]** point to **vector0**.


eip
cs
eflags
vector0:
call trap0
iret
trap0() {
.
.
.
}


eip
cs
eflags
vector0:
call trap0
iret
trap0() {
.
.
.
}

Saving interrupted code GPRs

- Registers destroyed by the interrupt service routine should be saved.
- **eip**, **eflags**, and possibly **esp** where already saved.
- The registers destroyed by **trap0()** are not known, thus save them all:

```
vector0:
  pushal
  call trap0
  popal
  iret
```

Our segment registers?

- **cs** was saved and then loaded from the IDT.
- **ss**:
  - User mode interrupted: **ss** was saved and then loaded from the TSS.
  - Kernel mode interrupted: **ss** already contains the correct value (SEG_KDATA).
- **ds**, **es**, **fs**, **gs**: Code needed:

```
movw $SEG_KDATA<<3,%ax
movw %ax,%ds
movw %ax,%es
movw $0,%ax
movw %ax,%fs
movw %ax,%gs
```

Infrastructure (1) for trap0()

```
vector0:
 pushal
 movw $SEG_KDATA<<3,%ax
 movw %ax,%ds
 movw %ax,%es
 movw $0,%ax
 movw %ax,%fs
 movw %ax,%gs
```

```
 call trap0
 popal
 iret
```

- We might have just destroyed user mode registers!

Infrastructure (2) for trap0()

```
vector0:
 pushl  %ds
 pushl  %es
 pushl  %fs
 pushl  %gs
 pushal
 movw $SEG_KDATA<<3,%ax
 movw %ax,%ds
 movw %ax,%es
 movw $0,%ax
 movw %ax,%fs
 movw %ax,%gs
```

```
 call  trap0
 popal
 popl %gs
 popl %fs
 popl %es
 popl %ds
 iret
```

Interrupted code registers content

- ISRs might be interested in accessing the interrupted code registers.
- (This is reasonable only for internal interrupt requests!)
- The register values where pushed onto the stack.
- A slight addition allows easy access to them from C.
- (elective) The values are accessible also without the change.

Infrastructure (3) for `trap0()`

```
vector0:
 pushl  %ds
 pushl  %es
 pushl  %fs
 pushl  %gs
 pushal
 movw $SEG_KDATA<<3,%ax
 movw %ax,%ds
 movw %ax,%es
 movw $0,%ax
 movw %ax,%fs
 movw %ax,%gs
```

```
 pushl %esp
 call trap0
 addl $4,%esp
 popal
 popl %gs
 popl %fs
 popl %es
 popl %ds
 iret
```

frame struct

```
struct frame {
 long edi;
 long esi;
 long ebp;
 long esp_irrelevant;
 long ebx;
 long edx;
 long ecx;
 long eax;

 short gs;
 short padding1;
 short fs;
 short padding2;
 short es;
 short padding3;
```

```
 short ds;
 short padding4;

 long  eip;
 short cs;
 short padding5;
 long eflags;

 long esp;
 short ss;
 short padding6;
}
```

Accessing interrupted code registers

- Possible declaration of `trap0()`:

```
void trap0(struct frame *f) {
 f->eax = someValue(f);
 ⋮
}
```

- Note: The **esp** and **ss** fields exist only if user mode was interrupted.

Infrastructure for **trap8()** (serving IRQ 8)

- IRQ8 leaves error code below **eip**.
- Hence:
  - The **frame** struct does not give the correct picture.
  - There is a superfulous word before the **iret** instruction.
- So:
  - A new structure replacing **frame** struct is needed.
  - Revmoing a word from the stack is needed.

Infrastructure (1) for **trap8()**

```
vector8:
 pushl   %ds
 pushl   %es
 pushl   %fs
 pushl   %gs
 pushal
 movw  $SEG_KDATA<<3,%ax
 movw  %ax,%ds
 movw  %ax,%es
 movw  $0,%ax
 movw  %ax,%fs
 movw  %ax,%gs
```

```
 pushl   %esp
 call  trap8
 addl  $4,%esp
 popal
 popl  %gs
 popl  %fs
 popl  %es
 popl  %ds
 addl  $4,%esp
 iret
```

eframe struct

```
struct eframe {
 long edi;
 long esi;
 long ebp;
 long esp_irrelevant;
 long ebx;
 long edx;
 long ecx;
 long eax;

 short gs;
 short padding1;
 short fs;
 short padding2;
 short es;
 short padding3;
```

```
 short ds;
 short padding4;

 long err;
 long eip;
 short cs;
 short padding5;
 long eflags;

 long esp;
 short ss;
 short padding6;
}
```

Accessing error code

- Possible declaration of `trap8()`:

```
void trap8(struct eframe *f) {
 if (f->err)
  f->eax = someValue(f);
 else
  f->eax = someOtherValue(f);
 ⋮
}
```

Uniform behaviour would be useful

- The above solution is feasible.
- However, if one uniform routine `trap()` is asked for, we tinker.
- How?

Infrastructure (4) for **trap0()**

```
vector0:
  pushl   $0
  pushl   %ds
  pushl   %es
  pushl   %fs
  pushl   %gs
  pushal
  movw  $SEG_KDATA<<3,%ax
  movw  %ax,%ds
  movw  %ax,%es
  movw  $0,%ax
  movw  %ax,%fs
  movw  %ax,%gs
```

```
  pushl   %esp
  call  trap0
  addl  $4,%esp
  popal
  popl  %gs
  popl  %fs
  popl  %es
  popl  %ds
  addl  $4,%esp
  iret
```

The **trapframe** struct (1)

```
struct trapframe {
 long edi;
 long esi;
 long ebp;
 long esp_irrelevant;
 long ebx;
 long edx;
 long ecx;
 long eax;

 short gs;
 short padding1;
 short fs;
 short padding2;
 short es;
 short padding3;
```

```
 short ds;
 short padding4;

 long  err;
 long  eip;
 short cs;
 short padding5;
 long  eflags;

 long  esp;
 short ss;
 short padding6;
}
```

All isr's can use the **trapframe** struct

```
void trap0(struct trapframe *f) {
  ⋮
}

void trap8(struct trapframe *f) {
  ⋮
}
```

- Let us reorganize the entry code.

Infrastructure (5) for **trap0()**

```
vector0:
 pushl  $0
 jmp alltraps

alltraps:
 pushl  %ds
 pushl  %es
 pushl  %fs
 pushl  %gs
 pushal
 movw $SEG_KDATA<<3,%ax
 movw %ax,%ds
 movw %ax,%es
 movw $0,%ax
 movw %ax,%fs
 movw %ax,%gs
```

```
 pushl  %esp
 call trap0
 addl $4,%esp
 popal
 popl %gs
 popl %fs
 popl %es
 popl %ds
 addl $4,%esp
 iret
```

Infrastructure (5) for **trap8()**

```
vector8:
 jmp alltraps


alltraps:
  pushl  %ds
  pushl  %es
  pushl  %fs
  pushl  %gs
  pushal
 movw  $SEG_KDATA<<3,%ax
 movw  %ax,%ds
 movw  %ax,%es
 movw  $0,%ax
 movw  %ax,%fs
 movw  %ax,%gs
```

```
  pushl  %esp
  call  trap8
  addl  $4,%esp
  popal
  popl  %gs
  popl  %fs
  popl  %es
  popl  %ds
  addl  $4,%esp
  iret
```

Infrastructure (1) for **trap()**, IRQ0

```
vector0:
  pushl  $0
  jmp  alltraps
vector8:
  jmp  alltraps
alltraps:
  pushl  %ds
  pushl  %es
  pushl  %fs
  pushl  %gs
  pushal
  movw  $SEG_KDATA<<3,%ax
```

```
  movw  %ax,%ds
  movw  %ax,%es
  movw  $0,%ax
  movw  %ax,%fs
  movw  %ax,%gs
  pushl  %esp
  call  trap
  addl  $4,%esp
  popal
  popl  %gs
  popl  %fs
  popl  %es
  popl  %ds
  addl  $4,%esp
  iret
```

Infrastructure (1) for **trap()**, IRQ0

```
vector0:
 pushl   $0
 jmp  alltraps
vector8:
 jmp  alltraps
alltraps:
 pushl   %ds
 pushl   %es
 pushl   %fs
 pushl   %gs
 pushal
 movw  $SEG_KDATA<<3,%ax
```

What is lost??

```
 movw  %ax,%ds
 movw  %ax,%es
 movw  $0,%ax
 movw  %ax,%fs
 movw  %ax,%gs
 pushl   %esp
 call  trap
 addl  $4,%esp
 popal
 popl  %gs
 popl  %fs
 popl  %es
 popl  %ds
 addl  $4,%esp
 iret
```

Infrastructure (2) for **trap()**

```
vector0:
 pushl  $0
 pushl  $0
 jmp alltraps
vector8:
 pushl  $8
 jmp alltraps
alltraps:
 pushl  %ds
 pushl  %es
 pushl  %fs
 pushl  %gs
 pushal
 movw $SEG_KDATA<<3,%ax
```

```
 movw %ax,%ds
 movw %ax,%es
 movw $0,%ax
 movw %ax,%fs
 movw %ax,%gs
 pushl  %esp
 call trap
 addl $4,%esp
 popal
 popl %gs
 popl %fs
 popl %es
 popl %ds
 addl $8,%esp
 iret
```

The **trapframe** struct (2)

```
struct trapframe {
  long edi;
  long esi;
  long ebp;
  long esp_irrelevant;
  long ebx;
  long edx;
  long ecx;
  long eax;

  short gs;
  short padding1;
  short fs;
  short padding2;
  short es;
  short padding3;
```

```
  short ds;
  short padding4;

  long  trapno;
  long  err;

  long  eip;
  short cs;
  short padding5;
  long  eflags;

  long esp;
  short ss;
  short padding6;
}
```

trap()

```
void trap(struct trapframe *tf) {
 switch (tf->trapno) {
 case 0:
  trap0(tf);
  break;

  ⋮

 case 8:
  trap8(tf);
  break;

 ⋮
 }
}
```

Retro groking on process creation

- Note the **trapframe** structure and recall **userinit()**.
- Now we can understand why things are as they are:
  - There are always interrupts.
  - User mode code is always going to be interrupted.
  - So there must be a framework for:
    - switching from user mode to kernel mode.
    - Return safely from kernel mode to user mode.
  - Running new user code builds on this framework.
  - New user code has no history (It is new!).
    - So a fake history is build for it.
    - And then the kenel 'returns' to the code entry point.

- The xv6 code.

Kernel entry points

We combine the previous techniques to get:

- Kernel entry points **vector0** upto **vector255** follows on the next slide.
- Each entry will contains one or two **push** instructions followed by

```
jmp alltraps
```

- Use some scripting language to generate the `.S` file with the above code.
- (elective) (or by a smart.s and some fancy assembly directives).

all vectors

```
vector0:
  pushl   $0
  pushl   $0
  jmp     alltraps
    ⋮
vector7:
  pushl   $0
  pushl   $7
  jmp     alltraps
vector8:
  pushl   $8
  jmp     alltraps
vector9:
  pushl   $0
  pushl   $9
  jmp     alltraps
```

```
vector10:
  pushl   $10
  jmp     alltraps
  ⋮
vector14:
  pushl   $14
  jmp     alltraps
vector15:
  pushl   $0
  pushl   $15
  jmp     alltraps
vector16:
  pushl   $0
  pushl   $16
  jmp     alltraps
```

```
vector17:
  pushl   $17
  jmp     alltraps
vector18:
  pushl   $0
  pushl   $18
  jmp     alltraps
    ⋮
vector255:
  pushl   $0
  pushl   $255
  jmp     alltraps
```

all vectors

```
vector0:
  pushl  $0
  pushl  $0
  jmp    alltraps
   ⋮
vector7:
  pushl  $0
  pushl  $7
  jmp    alltraps
vector8:
  pushl  $8
  jmp    alltraps
vector9:
  pushl  $0
  pushl  $9
  jmp    alltraps
```

```
vector10:
  pushl  $10
  jmp    alltraps
  ⋮
vector14:
  pushl  $14
  jmp    alltraps
vector15:
  pushl  $0
  pushl  $15
  jmp    alltraps
vector16:
  pushl  $0
  pushl  $16
  jmp    alltraps
```

```
vector17:
  pushl  $17
  jmp    alltraps
vector18:
  pushl  $0
  pushl  $18
  jmp    alltraps
   ⋮
vector255:
  pushl  $0
  pushl  $255
  jmp    alltraps
```

Tedious!

alltraps

```
3304 alltraps::
  pushl   %ds
  pushl   %es
  pushl   %fs
  pushl   %gs
  pushal
  movw  $SEG_KDATA<<3,%ax
  movw  %ax,%ds
  movw  %ax,%es
  movw  $0,%ax
  movw  %ax,%fs
  movw  %ax,%gs
```

```
  pushl  %esp
  call  trap
  addl  $4,%esp
trapret:
  popal
  popl  %gs
  popl  %fs
  popl  %es
  popl  %ds
  addl  $8,%esp
  iret
```

vectors[]

- How do we loop on **vector0** through **vector255** to initialize IDT?

vectors[]

- How do we loop on **vector0** through **vector255** to initialize IDT?
  - We cannot.

vectors[]

- How do we loop on **vector0** through **vector255** to initialize IDT?
  - We cannot.
- So, we build a vector of addresses:

```
vectors::
 .long vector0
 .long vector1
 ⋮
 .long vector255
```

- Writing this is also TEDIOUS.

vectors.pl

```perl
for (my $i = 0; $i < 256; $i++) {
 print "vector$i:\n";
 if (!($i == 8 ||
     ($i >= 10 && $i <= 14) || $i == 17)) {
  print " pushl \$0\n";
 }
 print " pushl \$$i\n";
 print " jmp alltraps\n";
}
print ".data\n";
print "vectors:\n";
for (my $i = 0; $i < 256; $i++) {
 print " .long vector$i\n";
}
```

(elective) a direct vectors.S

```
  .globl  alltraps, vectors
  .data
vectors:
 v = 0
.rept 256
  .text
0:
  .if !(v == 8 || (10 <= v && v <= 14) || v == 17)
    pushl $0
  .endif
  pushl $v
 jmp alltraps
  .data
  .long 0b
 v = v + 1
.endr
 .end
```

IDT initializing

Gate descriptor

Trap/Interrupt gate:

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

31 16	15 0
Selector	Offset (low)

```c
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
};
```

SETGATE macro

```c
#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; \
}
```

IDT construction

```c
#define T_SYSCALL 64

struct gatedesc idt[256];
extern uint vectors[]; // in vectors.S: array of 256

tvinit(void) {
 for (int i = 0; i < 256; i++)
  SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

 SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
}

void idtinit(void) {
 lidt(idt, sizeof(idt));
}
```

Finaly, we can investigate **trap()**.

trap()

The code of **trap()** is composed of three parts:

1. Handle **killed** and dispatch to **syscall()** if invoked by **int $64**.
2. Dispatch to code handling other IRQ's.
3. Code handling **killed** and **yield()**.

Note about **killed**:

- **killed** set means the process should terminate.
- The **killed** field should be checked at the following locations:
  - Entering the kernel.
  - Leaving the kernel.
  - Whenever the process might enter **SLEEPING** for a long time.

trap() part 1

```
void trap(struct trapframe *tf) {
 if (tf->trapno == T_SYSCALL) {
  if (myproc()->killed)
   exit();
  myproc()->tf = tf;
  syscall();
  if (myproc()->killed)
   exit();
  return;
 }
```

- User mode might be loooong, hence the check on **myproc()**->**killed** before returing.

trap() part 2, controller interrupts

```
switch (tf->trapno) {
case T_IRQ0+IRQ_TIMER:
 if (cpuid() == 0) {
  acquire(&tickslock);
  ticks++;
  wakeup(&ticks);
  release(&tickslock);
 }
 lapiceoi();
 break;
case T_IRQ0+IRQ_IDE:
 ideintr();
 lapiceoi();
 break;
case T_IRQ0+IRQ_IDE+1:
 break;
```

```
case T_IRQ0+IRQ_KBD:
 kbdintr();
 lapiceoi();
 break;
case T_IRQ0+IRQ_COM1:
 uartintr();
 lapiceoi();
 break;
case T_IRQ0+7:
case T_IRQ0+IRQ_SPURIOUS:
 cprintf("cpu%d: spurious \
    interrupt at %x:%x\n",
   cpuid(), tf->cs, tf->eip);
 lapiceoi();
 break;
```

trap() part 2, unexpected interrupt

```
  default:
    if (myproc() == 0 || (tf->cs&3) == 0) {
      cprintf("unexpected trap %d from cpu %d \
          eip %x (cr2=0x%x)\n",
              tf->trapno, mycpu()->id, tf->eip, rcr2());
      panic("trap");
    }
    cprintf("pid %d %s: trap %d err %d on cpu %d "
        "eip 0x%x addr 0x%x     kill proc\n",
    myproc()->pid, myproc()->name, tf->trapno, tf->err
            cpuid(), tf->eip,
    rcr2());
    myproc()->killed = 1;
  }
```

trap() part 3

```
    if (myproc() && myproc()->killed &&
            (tf->cs&3) == DPL_USER)
        exit();

    if (myproc() && myproc()->state == RUNNING &&
          tf->trapno == T_IRQ0+IRQ_TIMER)
        yield();

    if (myproc() && myproc()->killed &&
            (tf->cs&3) == DPL_USER)
        exit();
}
```