CS3210: User Environments and Interrupts

Dr. Tim Andersen

Administrivia

- Quiz 1 on Tuesday, 10/3.
- Final Project Pre-Proposal due 10/2.
- Team Propsals presented on 10/24. Can change your group membership freely until then.
- Lab 2 due Friday, 9/29.
- Lab 3 Part A due 10/6 (Part B with Part A corrections due the following week).

Quiz 1

- 80 minutes
- Open notes and laptop, NO internet
- Hand Written (no electronic submissions but you may be asked to run code or shell commands)
- Covers:
  - Lab 1-2, Chapter 0-2, Appendix A/B
  - Lectures up to and including Lecture 6
  - Tutorials up to and including Tutorial 4
- Be sure to understand the following:
  - Boot up sequence
  - Segmentation and Isolation
  - Shells and OS organization (syscalls, fork, pipe, FDs, etc.)
  - Virtual memory (including x86 architecture)
  - C coding
  - x86 assembly that has been used in labs, lectures, and tutorials so-far
  - Calling conventions (stack frames, etc.)
- Old MIT 6.828 quizzes (which are the same format) are linked from CS3210 website's references (at the bottom)

User Environments

- Environments or Processes used for executing user processes
- Space for meta-data on processes allocated in kernel at boot:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC]; // Allocate for NPROC processes
} ptable;
```

- In xv6, this is done in the kernel's stack. In JOS, need to allocate pages.

Steps to Running a Process (xv6) or Environment (JOS)

1. Initialize the environment table and set up the GDT.
2. Set up the kernel portion (>KERNBASE) of virtual memory for each process
3. Map physical memory for the process (below KERNBASE) (text + rodata + stack + heap)
4. Load the user binary into the mapped memory for text + rodata by reading the ELF headers
5. Run the environment or hand it off the scheduler.

Initializing User Environments

JOS function:

- env_init()
  - Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).

Xv6 function:

- Partly just definition (no linked list, just an array) and `switchuvm` to set up the process GDT

Setting up Virtual Memory Mapping

JOS function:

- env_setup_vm()
  - Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

Xv6 function:

- setupkvm()
  - Allocates the kernel portion of a process (as well as kpgdir when there is no process)

Mapping physical memory for an environment

JOS function:

- region_alloc()
  - Allocates and maps physical memory for an environment

Xv6 function:

- allocuvm()
  - Grows the process allocation (can start from 0)

Loading ELF code from binary

JOS functions:

- env_create()
  - Allocate an environment with env_alloc and call load_icode load an ELF binary into it.
- load_icode()
  - You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.
  - JOS supports all user programs being linked into the kernel image so no file system needed!

Xv6 function:

- exec() (exec.c)
  - Xv6 assumes you have a file system and that files are being loaded from disk.
  - exec loads the binary from disk (loaduvm() helper function loades each segment in pages)
- Also userinit() and inituvm() for the first process initcode which is linked into the kernel but not ELF.

Running the binary

JOS function:

- env_run()
  - Start a given environment running in user mode.

Xv6 function:

- No real equivalent. Handled by the scheduler() in proc.c and switchuvm().

Interrupt

- An interrupt informs the CPU that a service is needed
- Sources of interrupts
  - Internal faults: divide by zero, overflow
  - User software
  - Hardware
  - Reset

Def: An event external to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU.*

* "Design and Implementation of the FreeBSD Operating System", Glossary

Why Interrupts?

- People can't use a CPU without things attached to it
  - Keyboard, mouse, screen, disk drives, network cards, etc.
- Also want to have the ability to stop a running program when it messes something up and clean up.
- Devices need CPU services at unpredictable times.
- Want the CPU busy doing useful work between events but also stop what it is doing and service those events in a timely manner.

Polling?

- Have the CPU periodically check each device to see if it needs attention

Polling?

- Have the CPU periodically check each device to see if it needs attention
    - Inefficient if events are on a slow timescale
    - If events happen rapidly, can be more efficient.
- Polling is like checking your phone every few seconds to see if you have a message.
- Interrupts is like waiting for your phone to play a sound when you have a message

x86 Exceptions and Interrupts

- Every Exception/Interrupt type is assigned a number
  - its vector
- When an interrupt occurs, the vector determines what code is invoked to handle the interrupt
- JOS example:
  - vector 14 -> page fault handler
  - vector 32 -> clock handler -> scheduler

Hardware Interrupts

- **Non-Maskable Interrupts**
  - Never ignored, e.g., power failure, memory error
  - In x86, vector 2, prevents other interrupts from executing.
- **INTR Maskable**
  - Ignored when [[IF]] in `EFLAGS` is 0
  - Enabling/disabling:

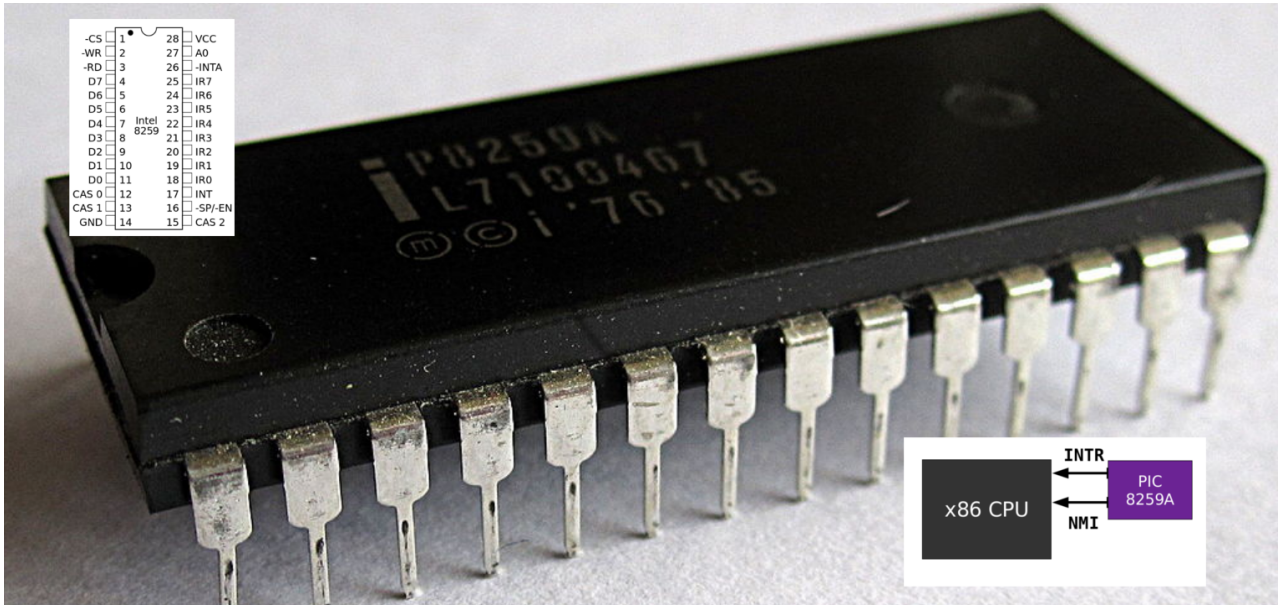
```
- `sti`: set interrupt
- `cli`: clear interrupt
```

- INTA
  - Interrupt acknowledgement

PIC: Programmable Interrupt Controller (8259A)

- Has 16 wires for 8 devices, interrupts and data lines, (IR0-IR7 and D0-D7)
- Can be programmed to map IRQ0-7 -> vector number
- Vector number is signaled over INTR line
- Daisy-Chain up to 8 slave PICs to one master for up to 64 devices.
- Interrupt Mask Register (Port 21H) enbles/disables interrupts at PIC
- In JOS/lab4
  - vector <- (IRQ# + OFFSET)

PIC Diagram


-CS 1
-WR 2
-RD 3
D7 4
D6 5
D5 6
D4 7
D3 8
D2 9
D1 10
D0 11
CAS 0 12
CAS 1 13
GND 14
Intel 8259
28 VCC
27 A0
26 -INTA
25 IR7
24 IR6
23 IR5
22 IR4
21 IR3
20 IR2
19 IR1
18 IR0
17 INT
16 -SP/-EN
15 CAS 2
P8250A
L7100467
i '76 '85
x86 CPU
INTR
NMI
PIC 8259A

"Software" interrupt: INT

- Intentionally interrupts
  - x86 provides the INT instruction
  - Invokes the interrupt handler for the vector (0-255)
  - JOS: `INT 0x30` for system calls
    - xv6: `INT 0x40` for system calls

- Entering: `int N`
- Exiting: `iret`

The INT instruction

- The INT instruction has the following steps:
  - decide the vector number, in this case it's the 0x40 in int 0x40
  - fetch the interrupt descriptor for vector 0x40 from the IDT. The CPU finds it by taking the 0x40'th 8-byte entry starting at the physical address that the IDTR CPU register points to.
  - check that CPL <= DPL in the descriptor (but only if INT instruction).
  - save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).
  - load SS and ESP from TSS ("")
  - push user SS ("")
  - push user ESP ("")
  - push user EFLAGS
  - push user CS
  - push user EIP
  - clear some EFLAGS bits
  - set CS and EIP from IDT descriptor's segment selector and offset

Example: entering (usys.S)

- `vectorN` -> `alltraps` -> `trap()` -> `syscall()`

```
01    #define SYSCALL(name)          \
02      .globl name;                 \
03      name:                        \
04        movl $SYS_ ## name, %eax; \
05        int $T_SYSCALL;            \
06        ret
07
08    SYSCALL(fork)
09    SYSCALL(exit)
10    ...
```

Example: exiting (trapasm.S)

- `syscall()` -> `trapret()` -> `iret`

```
01    .globl trapret
02    trapret:
03      popal
04      popl %gs
05      popl %fs
06      popl %es
07      popl %ds
08      addl $0x8, %esp  # trapno and errcode
09      iret
```

Interrupt Vector (vector.S)

- `int 0` -> `vector0`

```
01   # handlers
02   vector0:
03     pushl $0
04     pushl $0
05     jmp alltraps
06   ...
07
08   # vector table
09   vectors:
10     .long vector0
11     .long vector1
12   ...
```

Interrupt Vector

- Some vectors need to push 0 for their error code and others do not

```
01   vector0:
02     pushl $0      ; error code
03     pushl $0      ; #vector
04     jmp alltraps
05
06   ...
07   vector8:
08     pushl $8      ; #vector
09     jmp alltraps
10   ...
```

Trap Handling DEMO

- `int 0x40` entered the kernel at vector64, generated by vectors.pl. `b vector64`
- What is the current CPL? How was it set?
  - Could the user abuse the INT instruction to exercise privilege or break the kernel?
- `x/6x $esp` in order to see what int put on the stack.
  - What stack is being used?
- `x/3i vector64`
  - vector64 pushes a few items on the stack and then jumps to alltraps.
  - Why not have vector 64 in the IDT point directly to alltraps?
- Single-step alltraps until pushl %esp, then x/19x $esp.
  - Compare with struct trapframe (x86.h)
- At the start of trap(), what is tf->trapno?
  - How was it set?

Trap Return

- syscall() returns to trap(), and trap() returns to alltraps
- b trap.c:44 (instruction after call syscall).
  - print *tf
  - What is different and why?
  - si until popal.
  - x/19x $esp to see the trap frame again.
- single-step until iret, x/5x $esp, single-step
  - into user space. Print the registers and stack.

Fault Handling Traps

- What would happen if a user program divided by zero? - What if kernel code divided by zero?
- In Unix, traps often get translated into signals to the process.
  - Some traps, though, are (partially) handled internally by the kernel -- which ones?
- Some traps push an extra error code onto the stack (typically containing the segment descriptor that caused a fault).
  - But this error code isn't pushed by the INT instruction.
  - Can the user confuse the kernel by invoking INT 0xc (or any other vector that usually pushes an error code)? Why not?

JOS Trap Frame

```
struct Trapframe {
  struct PushRegs tf_regs;
  uint16_t tf_es;
  uint16_t tf_padding1;
  uint16_t tf_ds;
  uint16_t tf_padding2;
  uint32_t tf_trapno;
  /* below here defined by x86 hardware */
  uint32_t tf_err;
  uintptr_t tf_eip;
  uint16_t tf_cs;
  uint16_t tf_padding3;
  uint32_t tf_eflags;
  /* below here only when crossing rings, such as from user to kernel */
  uintptr_t tf_esp;
  uint16_t tf_ss;
  uint16_t tf_padding4;
} __attribute__((packed));
```

Real-mode

- For any INT n, n is multiplied by 4
  - In the address “4n” the offset address the handler is found
- Example:Intel has set aside INT 2 for the NMI interrupt
  - Whenever the NMI pin is activated, the CPU jumps to physical memory location 00008 to fetch the CS:IP of the interrupt service routine associated with the NMI.
- In protected mode, this scheme is replaced by the Interrupt Descriptor Table

Interrupt Descriptor Table

- **IDT**
  - Table of 256 8-byte entries (similar to GDT)
  - In JOS: Each specifies a protected entry-point into the kernel
  - Located anywhere in memory
- **IDTR register**
  - Stores current IDT
- **lidt instruction**
  - Loads IDTR with address and size of the IDT
  - Takes in a linear address

Interrupt Descriptor Table Diagram

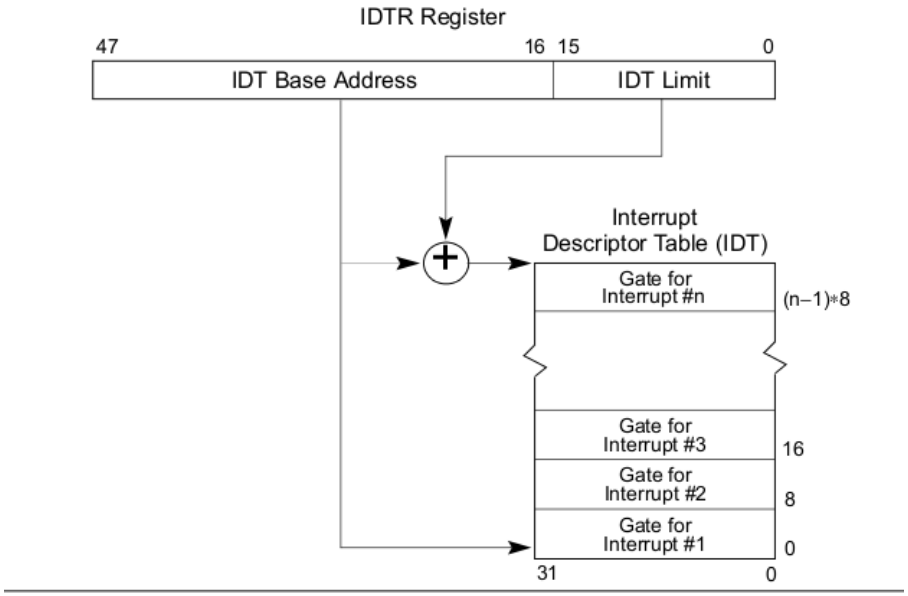


Figure 6-1. Relationship of the IDTR and IDT

Initializing IDT in xv6 (trap.c)

- `main()` -> `tvinit()`

```
01  void tvinit(void)
02  {
03    int i;
04    for (i = 0; i < 256; i++)
05      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
06
07    // Q?
08    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, \
09              vectors[T_SYSCALL], DPL_USER);
10  }
```

Initializing IDT in xv6 (trap.c)

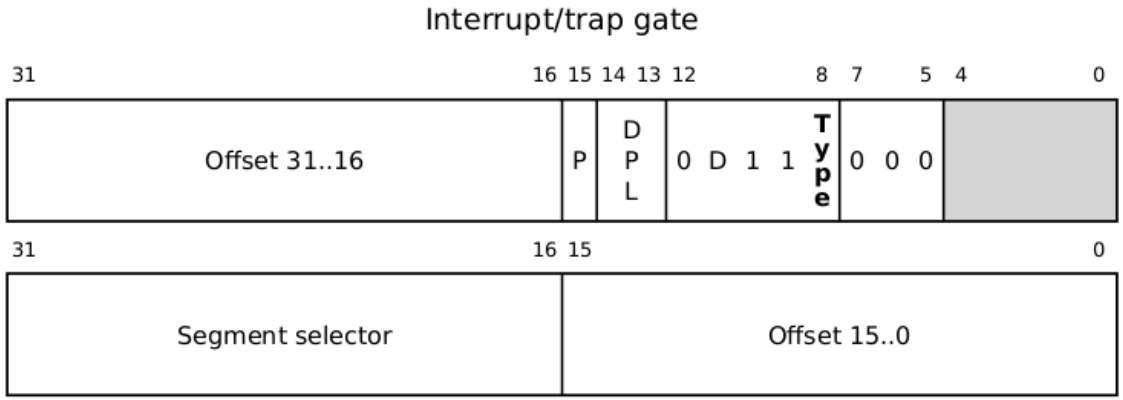
- `main()` -> `idtinit()`

```
01 void
02 idtinit(void)
03 {
04   lidt(idt, sizeof(idt));
05 }
```

Interrupt Descriptor Entry

- Offset is a 32-bit value split into two parts pointing to the destination IP or EIP
- Segment selector points to the destination CS in the kernel
- Present flag indicates that this is a valid entry
- Descriptor Privilege Level indicates the minimum privilege level of the caller to prevent users from calling hardware interrupts directly
- Size of gate can be 32 bits or 16 bits
- Gate can be interrupt (`int` instruction) or trap gate

Interrupt Descriptor Entry



Type	0=Interrupt gate 1=Trap gate	P	Present
Selector	Destination CS	DPL	Descriptor privilege level (CPL required to invoke gate)
Offset	Destination IP or EIP	D	Size of gate (0=16-bits, 1=32-bits)

Interrupt Descriptor Table

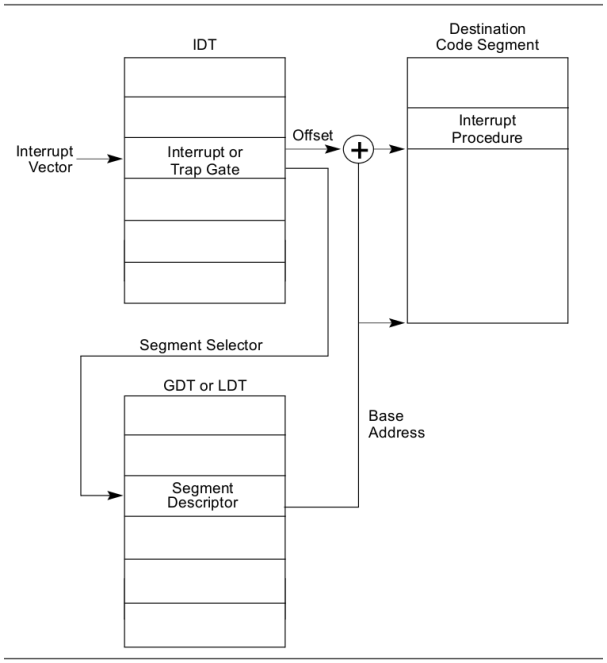


Figure 6-3. Interrupt Procedure Call

Predefined Interrupt Vectors

- `0`: Divide Error
- `1`: Debug Exception
- `2`: Non-Maskable Interrupt
- `3`: Breakpoint Exception (e.g., `int3`)
- `4`: Invalid Opcode
- `13`: General Protection Fault
- `14`: Page Fault
- `18`: Machine Check (abort)
- `32-255`: User Defined Interrupts

Software Exceptions

- Processor detects an error condition while executing
- E.g., divl %eax, %eax
  - Divide by zero if eax = 0
- E.g., movl %ebx, (%eax)
  - Page fault or seg violation if eax is unmapped
- E.g., jmp $BAD_JMP
  - General Protection Fault (jmpd out of CS)

Example: Divide Error

```
01  int main(int argc, char **argv)
02  {
03    int x, y, z;
04    if (argc < 3)
05      exit();
06
07    x = atoi(argv[1]);
08    y = atoi(argv[2]);
09
09    // Q?
10    z  = x / y;
11    printf(1, "%d / %d = %d\n", x, y, z);
12    exit();
13  }
```

Example: 0/0 = ?


0:00 / 0:20
Messages
Music
Google Maps
Camera
Facebook
Twitter
Safari
Reminders
Clock
Yelp
Passbook
Instagram
Slack
Travel
Shopping
News
Health&Fitness
Food
Photography
Notes
Settings
Calendar