Operating Systems Engineering

Recitation 3: Interrupts, Exceptions

Based on MIT 6.828 (2014, lec8)
Stacks in the System

• Where can our code run?
  – User mode
  – Kernel Mode

• Same stack for user and kernel?
  – How many user stacks? 1 or many?
  – 1 for each process? More?
  – How many kernel stacks?

• Why would you need more than 1?
Mode Transitions (1)

• What mode transitions do we have?
  • User – Kernel
    – System calls
    – Program faults (div by zero, page fault)
    – External device interrupts
  • Kernel - User
  • Kernel - Kernel
    – Interrupts
    – Context switch
Mode Transitions (2)

- Why is user - kernel transition delicate?
- What do we need to maintain?
  - Need to maintain isolation/security
    - only kernel can touch devices, MMU, FS, other process's state
    - think of user program as a potential malicious adversary
  - Need to maintain transparency
Reminder: x86 Privilege Levels

- CPL in low 2 bits of CS
  - cs: [ ] CPL

- CPL=0 --> can modify cr<n>, devices, can use any PTE
- CPL=3 --> can't modify cr<n>, or use devs, and PTE_U enforced
System Calls

• What needs to happen in a system call?
  1. Save user state
  2. Transition to kernel (stack, CPL=0)
  3. Choose kernel entry point
  4. Get system call arguments

• Is it secure? What if user could interfere between 2 and 3?
Interrupt vectors

- Where does “int $0x30” jump to?
  - $0x30 is an interrupt vector
  - A vector is an allowed kernel entry point
- x86 has 256 vectors
- Different uses (devices, exceptions, syscalls…)
- Each vector in an index in the IDT (Interrupt Descriptor Table)
- IDTR register holds the (virtual) base address of the IDT
- Each descriptor contains a segment selector, an offset in that segment, and a DPL
Interrupt Descriptor Table

- Table of all interrupt descriptors
- Pointer to by the **IDTR** register
- Contains interrupt / trap gates

<table>
<thead>
<tr>
<th>Offset 31..16</th>
<th>P</th>
<th>DPL</th>
<th>0</th>
<th>D</th>
<th>1</th>
<th>1</th>
<th>Type 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment selector</td>
<td>Offset 15..0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Type**
- 0 = Interrupt gate
- 1 = Trap gate

**Selector**
- Destination CS

**Offset**
- Destination IP or EIP

**P** Present
- Descriptor privilege level
- (CPL required to invoke gate)

**D** Size of gate (0=16-bits, 1=32-bits)
CPU Interrupt Handling

1. fetch vector's descriptor from IDT
2. If segment selector's PL < CPL (need switch to kernel)
   - load SS and ESP from TSS
   - push user SS
   - push user ESP
3. push user EFLAGS
4. push user CS
5. push user EIP
   - optionally push error word
6. clear some EFLAGS bits
7. load CS and EIP from IDT
Kernel Stack After INT

- `sp from task segment`
- `ss`
- `esp`
- `eflags`
- `cs`
- `eip`
- `error code`
- `(empty)`

Only present on privilege change
xv6 Interrupt Handling (1)

- Kernel SS and ESP setup
  - `switchuvm()` in `vm.c` (sheet 17)
- IDT setup
  - `tvinit()` in `trap.c` (sheet 30)
- Entry point calls `alltraps()`, which calls `trap()`
- Who initialized and passed `trapframe` to `trap()`?
  - `trapframe` (sheet 06)
Entry points – vectors.pl (sheet 29)

Each entry pushes error code + INT number
- Jump to alltraps (3004)

alltraps() proceeds with
- Push %DS, %ES, %FS, %GS
- All gen. purpose regs with PUSHA
- Setup segments registers
- Push %ESP (argument for trap() func, sheet 31)
- Call trap(%ESP)
Kernel Stack After xv6 alltrap()
struct trapframe {
    uint edi;
    uint esi;
    uint ebp;
    uint oesp;
    uint ebx;
    uint edx;
    uint ecx;
    uint eax;
    ushort gs; ushort padding1;
    ushort fs; ushort padding2;
    ushort es; ushort padding3;
    ushort ds; ushort padding4;
    uint trapno;
    uint err;
    uint eip;
    ushort cs;
    ushort padding5;
    uint eflags;
    uint esp;
    ushort ss;
    ushort padding6;
};
System Call Handling

• If tf->trapno == T_SYSCALL (0x40) – trap() calls syscall() (sheet 33)
• Syscall number determined by tf->eax
• Where is this value set?
• No value returned to trap – where is system call return value?
• How are parameters passed?
Handling Other Interrupts

- Other interrupts are handled the same
- Can be handled internally by kernel
- User can subscribe to some POSIX Signals
- Can happen in kernel
- Trapframe is a bit different, how?
Device Interrupts

• Hardware generated
• Different vector for different devices
  – Timer
  – Console
  – Disk
  – Network
Per-CPU data segments

- alltraps() sets %FS and %GS
  - point at the SEG_KCPU per-CPU data segment (3016-3018)
- %GS:0 ← &cpus[cpunum()]
- %GS:4 ← cpus[cpunum()].proc
  - proc.h (2079-2080) extern structs
  - loadgs() called in seginit() (1631,1634)
Fork

fork() – proc.c (sheet 23)
allocproc() – proc.c (sheet 22)