Lect. 16: Interrupt Mechanism and Context Switching

Xin Liu

Florida State University

xliu15@fsu.edu

CIS 5370 Computer Security

https://xinliulab.github.io/cis5370.html

Processor and Interrupts

The Ideal Processor

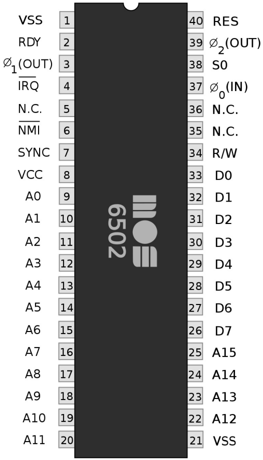
- A machine that executes instructions **unconditionally**.

```
    for (day = TODAY; day != FOREVER; day++) {
        say("I␣love␣you\n");
    }
```

The Real Processor is NOT ”Unconditionally Executing Instructions”

- It **”cares”** and responds to external **interrupts**.
- If you fall into an infinite loop in the library...
  - A friendly security guard will ”interrupt” you.

Interrupt = A Single Wire



- **"Telling the processor: Stop, something has happened!"**
- **"The rest is up to the processor."**

Processor Interrupt Behavior

If the Processor Interrupts are Enabled

- **x86 Family (CISC, a legacy of history; a nightmare for processor designers)**
  - Reads interrupt vector number *n* via the interrupt controller.
  - Saves CS, EIP, EFLAGS, SS, and ESP onto the stack.
  - Jumps to the ”Gate” in `IDT[n]`.
  - **A data structure that describes privilege-level switching and long jumps.**
- **RISC-V (M-Mode, Direct Exception Mode)**
  - Checks whether this interrupt should be masked.
  - Jumps: `PC = (mtvec & ˜0xF)`
  - Updates: `mcause.Interrupt = 1`

Another Way to Understand Interrupts

Forcibly ”Injected” syscalls

- **Interrupt**
  - Saves: `mepc = PC`
  - Jumps: `PC = (mtvec & ˜0xF)`
  - Updates: `mcause.Interrupt = 1`
- **System Call (ecall)**
  - Saves: `mepc = PC`
  - Jumps: `PC = (mtvec & ˜0xF)`
  - Updates: `mcause.Ecall = 1`

”No matter what you are doing right now, go execute the system core code!”

Interrupts Grant OS ”Supremacy”

Operating System Kernel (Code)

- Can enable and disable interrupts at will.

User Applications

- Sorry, no direct control over interrupts.
  - You can inspect the flags register (`FL_IF`) in `gdb`.
  - **CLI - Clear Interrupt Flag**
    - #GP(0) occurs if **CPL** is greater than **IOPL** and less than 3.
    - Try using: `asm volatile ("cli");`
- Regardless of what code you write, it will always be interrupted.

Assume an Interrupt Occurs

What Should the Operating System Code Do?

- `mov (kernel_rsp), %rsp`
  - This can be fatal.
  - The process (state machine) state will be lost forever.

First: Save the State Machine (Registers)

- Preserve control over memory and data.
- Save register states to **physical memory** for later restoration.

Then: Execute the Operating System Code

- C code can freely use registers.
- The OS code selects a state machine for return.
- Restore register states from **physical memory**.
- Execute `sysret` (`iret`).

This is the most elegant pieces of code in operating systems.

Implementing Context Switching

Operating System Implementation Tricks

- Set up a ”current context”.
- Save and restore register states.
  - **AbstractMachine** already helps you obtain registers.

```
Context *on_interrupt(Event ev, Context *ctx) {
   // Save context.
   current->context = *ctx;

   // Thread schedule.
   current = current->next;

   // Restore current thread's context.
   return &current->context;
}
```

Takeaways

System call instructions are a special type of ”long jump”

- The jump target is pre-configured by the OS and **cannot be controlled by applications**.

Processor interrupts also trigger long jumps to the OS kernel

- The OS kernel **preserves** the process state machine:
  - **Memory pages remain unchanged**.
  - Carefully designed code ensures all registers are safely stored in memory.

At this moment, the system is in a state where:

- **All programs are suspended**, and only OS code is executing.
- The OS selectively schedules the next register context onto the CPU to achieve **context switching**.