Overview of Hardware
Overview

- Hardware
  - Processor
  - Memory (DRAM)
  - I/O devices
  - Interrupts
Processor (CPU)

- CPU executes a set of instructions
  - Different for different CPU architectures
  - Various memory and register-based instructions

- Anatomy of a CPU
  - Program Counter (PC): holds address of next instruction
  - Instruction Register (IR): holds instruction being executed
  - General Registers (Reg. 0..n): hold variables
  - Stack Pointer (SP): holds address for accessing stack
  - Status Register (SR): holds control bits that affect program execution, also called processor status word
CPU Execution

PC = <start address>;
// fetch-decode-execute loop
while (halt flag is not set) {
    IR = memory[PC]; // fetch next instrn. from mem
    PC = PC + 1;
    execute(IR); // decode & execute instruction
        // uses registers, stack pointer,
        // status register, etc.
}

- All a CPU does is Fetch/Decode/Execute
Memory

- Memory (DRAM) provides storage
  - Think of it as an array of bytes
  - Each byte has unique address
- Nr. of bits that represent address is called the address width
  - Example: 32-bit, 64-bit CPUs
- Simple abstraction
  - Write(address, value), writes a byte
    - Also called store instruction
  - value = Read(address), returns last value written
    - Also called load instruction
I/O Devices

- Computers are connected to IO devices via device-specific controllers
- One or more buses connect the CPU to memory and to device controllers
How does CPU Communicate With Devices?

- Each controller owns a range of "bus" addresses
  - CPU communicates with controller by sending message to address owned by the controller
  - Controllers listen for messages addressed to them

- How does the CPU allow the OS to access devices?
  - Special I/O instructions
    - Read/write to device registers
  - Memory-mapped I/O
    - Certain memory locations are mapped to device registers
    - CPU reads/writes these memory locations
      - H/W routes these loads/stores to device controller
    - I/O instructions are not needed
Communicating with Devices

- Say OS needs to read file from hard drive controller

  Communication model
  
  - Write(address, value)
    - CPU writes value (e.g., sector location, r/w) to controller address
  
  - value = Read(address)
    - CPU will poll (continuously read) address for a value to determine whether data is available from drive controller
    - Then read sector data

- How often should the OS poll the device?
  - Hard drive, keyboard device, high-speed n/w device
Interrupts

- Polling requiring tuning polling period, also not efficient
- CPU and devices can communicate more efficiently using hardware interrupts
  - CPU has an “interrupt request” flag that can be set by devices
  - Device sends an interrupt signal to the CPU when it is done

CPU and device can operate concurrently
Processor Execution with Interrupts

- Result: interrupt handler function can run at any time, and program is unaware it occurred!

Save value of PC in Step 1

1. Interrupt
2. Dispatch to handler
3. Return

Current instruction
Next instruction

Interrupt handler
PC = <start address>;
while (halt flag is not set) {
    IR = memory[PC]; // fetch next instrn. from mem
    PC = PC + 1;
    execute(IR);
    if (InterruptRequest) {
        hardware saves PC, SP, SR;
        PC = 0xIH; // h/w sets PC to address 0xIH,
                    // contains code of intr. handler
    }
}

Interrupt_handler() {
    save_processor_state(); // saves most CPU registers
    handle_interrupt();
    restore_processor_state(); // restores registers
    return from interrupt; // restores saved PC, SP, SR
}
Summary

- **CPU**: executes instructions
- **Memory**: array of bytes, used to store code and data
- **I/O devices**: run **concurrently** with CPU
  - CPU requests service from device
  - CPU can poll to check when device has finished serving request
- **Interrupts**
  - Allow CPU to do useful work until device is done with request
  - Interrupt handling requires support from h/w and software
Think Time

- What program variables are stored on the stack?
- What is memory-mapped IO?
- Is the device communication model similar to the memory abstraction?
- Should an OS ever use polling instead of interrupts?
- Can you think of a real world analogy when human use polling versus interrupts?
- In the processor execution code with interrupts, why are some registers saved by hardware, while others are saved by software?
What program variables are stored on the stack?

- Programs store their local variables, function arguments and return values on the stack. The other types of program variables are global variables and heap variables. These are not stored on the stack.

What is memory-mapped IO?

- Reading and writing certain regions of memory (using load/store instructions) allow communicating with a device.
Think Time Answers

- Is the device communication model similar to the memory abstraction?
  - The device communication model involves read and writing to the device and appears to be similar to reading and writing memory (see the read/write API shown in the slides). This is by design because it allows using memory-mapped IO to access devices. However, accessing a device is generally much more complicated because data may never arrive from a device, or it can be corrupted, or it may arrive out of order. In fact, it's well known that many of the bugs in operating systems are caused by bugs in device controllers or device drivers (the code in the operating system that handles interaction with devices and their controllers). Generally, these problems don’t occur with memory, which provides the guarantee that a load at an address will return the last value stored at that address.
Should an OS ever use polling instead of interrupts?

- We have seen in the slides that interrupts enables the CPU and the device to run concurrently. So is there ever a reason to use polling? Polling is beneficial with very high speed devices. The reason is that interrupt handling is expensive, and with high speed devices, lots of interrupts are generated, and so the OS may spend a lot of time running interrupt handler code. Polling a device tends to be cheaper in this case, and so it may make sense for the OS to poll the device whenever convenient. For example, the OS may poll the device at a fixed frequency. The tradeoff is that higher frequency polling is more expensive, but reduces the latency for handling the device event and is less likely to lose data.
Think Time Answers

- Can you think of a real world analogy when human use polling versus interrupts?
  - Normally, people don’t poll their phone to check whether a phone call is arriving. They just wait for a phone ring, which is an interrupt (it interrupts their work, after which they go back to doing what they were doing). Interrupts work well in this case because phone calls do not occur frequently. On the other hand, may people poll their email account periodically while doing their work, because emails arrive more frequently. Having a popup on each email (interrupt) can be highly disruptive to their work.
In the processor execution code with interrupts, why are some registers saved by hardware, while others are saved by software?

- The code shows that the PC, SP and the SR are saved by hardware, while the rest of the registers are saved by the interrupt handler in `save_processor_state`. The reason is that the hardware needs to save the PC, SP, SR to ensure that the interrupt handler code in software can start running correctly and it can restore these values when it returns. The rest of the registers do not need to be saved by hardware. For example, the hardware can run a null (empty) interrupt handler function in the OS by saving these three register values. The reason that the interrupt handler saves the rest of the registers is that it may overwrite them when it runs the `handle_interrupt` function. However, if the OS knows that the `handle_interrupt` function doesn’t use of these registers, then it doesn’t need to save and restore them, which makes the interrupt handling more efficient. For example, the Linux OS does not use floating point code, and so it does not save and restore the floating point registers in its interrupt handling code.