Supporting an Operating System
Part 2: IO, System Calls, and Boot

CSE 378 Spring 2009

Overview of IO

- All external interactions are IO
  - Keyboard, display, disk, network, scanner, camera, etc.
- It would be a mistake to tightly couple IO devices to CPU architecture
  - E.g., don't want to have a "write to disk" instruction
  - Why?
- Similarly, it would be a mistake to tightly couple IO devices to an operating system's implementation
- Finally, we'd even like to decouple the application code from the OS (i.e., portable languages/applications)

Review

- The exception architecture provides a protected way to cause a jump into OS code:
  - PC is set to trap handler entry point
  - This is the only way to set Status to privileged mode
- Memory mapping provides a general facility for introducing components into the datapath without introducing new instructions

A General View of IO

For engineering reasons, there is actually more than one bus on a typical system.
Controller ↔ Device Interface

- Presents a standard interface that isolates upstream components (bus, CPU, OS) from specific devices
  - Hitachi vs. Seagate drive
  - VGA vs. DVI vs. HDMI display

CPU / OS ↔ Controller

- Controller offloads work from CPU
  - CPU tells video card “Draw a (width=300, height=200) blue filled rectangle at pixel location (400, 525)”
- Don’t want to impede use of controllers that are invented after the CPU and/or OS are created
  - CPU
    - No special instructions to talk with controllers
    - Instead, controllers are memory mapped
  - OS
    - Encapsulate code that understands how to talk with controller in a driver
    - Allow introduction of driver to OS after OS has been installed
      - E.g., put new driver in special directory that OS looks at during boot

CPU / OS ↔ Controller

- Command issued using a store word (sw) on CPU
- Response read using a load word (lw) on CPU

Application ↔ OS

The library code knows how to make the appropriate system call for the OS on which its running.
Cebollita / SMOK

- Cebollita includes all of these concepts
  - An iolib.s library
  - An os with (built-in) drivers that talk with controllers
  - Hardware controller components that talk with IO devices
    - SMOK implements these controllers as well
- Character controller
  - Output side is the display (character display, not pixels)
  - Input side is the keyboard
- Disk controller
  - Input and output sides are a disk

Cebollita/SMOK Character IO

- Operates asynchronously
- Output side:
  - Bit 15: If one, indicates the device is ready to print a character
  - Bit 14: If on, indicates that a keyboard character is available
  - Bits 0-7: The keyboard character
- Input side:
  - Bit 15: Start a character write.
  - Bit 14: Clear the "keyboard character ready" status bit.
  - Bits 0-7: The character to write if bit 15 is on.

Example Use: Write a Character

- $t0 has the memory mapped address of the character controller
- $a0 has the character to write
  ```
  lw  $t1, 0($t0)
 andi $t1, $t1, 0x8000
  beq  $t1, $0, wait
  ori  $t1, $a0, 0x8000
  sw  $t1, 0($t0)
  ```
  - This is a busy wait loop
  - This is also polled I/O

How Do We Read a Character

- Using busy wait and polled I/O
- Code:
Disk I/O

Note: We're talking about the physical devices, not files. “File” is an operating system abstraction.

- The logical organization of a disk is as an array of blocks
  - Blocks are typically in the range 512B to 8KB
  - The disk's unit of addressing is the block
- Disks are direct memory access (DMA) devices
  - They read/write direction from/into memory

Reading a Disk Block

- Busy wait to start:
  - Read the device status until bits 15 and 14 are 0
  - Write the address of a buffer in memory to the memory address port
  - Write the disk block address to the disk address port
  - Write the block count to the count port
  - Write 0x0000 to the control port

- Busy wait to recognize completion:
  - Loop reading status until bit 15 is 1
  - Bit 13 should now be on, and bits 0-7 should have value 0
  - Write 0x2000 to the control input to clear the exception

Cebollita/SMOK Block Controller

- Four inputs:
  - Control (commands)
    - Bit 15: start a read
    - Bit 14: start a write
    - Bit 13: clear exceptions
    - Memory address (for DMA)
    - Disk Address (block number)
    - Count (blocks to transfer)
- Outputs:
  - Status: indicates state of device
    - Bit 15: busy reading
    - Bit 14: busy writing
    - Bit 13: exception has occurred
    - Bits 0-7: Exception cause
      - Bit 0: I/O completion
      - Bit 1: bad memory address
      - ...

Integrating the Block Controller

- Each of the four inputs is memory mapped
  - 0x40000010: control
  - 0x40000014: memory (buffer) address
  - 0x40000018: disk (block) address
  - 0x4000001C: block count
- The status output is also memory mapped:
  - 0x40000010
- The exception output could be used to provide interrupt driven IO, rather than polled
Interrupt Driven I/O

- How can the OS tell when the transfer is done?
  - Could sit in a busy loop reading the controller status
    - Busy waiting
  - Could check every once in a while
    - Polling
  - Or... the controller could raise an I/O completion interrupt
- Interrupts are “asynchronous exceptions”
  - They cause a transfer of control to the trap handler
  - When they occur has nothing to do with the instruction currently being executed by the CPU

Polling vs. Completion Interrupts

- It should be obvious what the advantages of completion interrupts are
- Cebollita uses polling and busy waits...

Next Topic: System Calls

- System calls are “protected procedure calls” of methods implemented in the OS
  - These methods have root privilege (so can do IO, for instance)
- Invoked using a special instruction, syscall
  - syscall causes an exception (i.e., jump to trap handler)
  - The cause register indicates a syscall happened
- jal vs. syscall
  - jal: caller decides what next PC will be
  - syscall: callee decides what next PC will be (trap handler)

syscall convention

- Just like procedure call, we need a convention to define how to pass arguments and how to get return values
  - We could also have a convention about saving registers, but...
  - Because OS must handle interrupts, it must be prepared to save everything itself
    - Why?
  - Since the caller isn't specifying a next PC address, we also need to communicate which OS method to invoke
    - Passed as a system call number (an int)
- Cebollita conventions:
  - $v0: syscall number (which method to invoke)
  - $a0 ...: argument(s)
  - $v0: return value
Cebollita iolib

```
.text
.global printInt
printInt:
    ori  $v0, 50, 1
    lw   $a0, 0($esp)
    syscall
    jr   $ra

.global readInt
readInt:
    ori  $v0, 50, 5
    syscall
    jr   $ra
...
```

Next Topic: Boot

- When machine is powered on, need to load the OS
- But:
  - Registers are nonsense
  - Memory is nonsense
- Need some "initial program" that isn't nonsense
  - BIOS, stored in NVRAM (non-volatile RAM)
- BIOS contains a very small, OS independent program
  - Loads block 0 of boot device into memory at location 0
  - Branches to location 0

Boot

- Block 0 of the boot device contains the boot loader
- The boot loader has enough smarts to load the remainder of the OS into memory
- It then branches to the entry point of the OS, which initializes itself
- Once initialized (booted), the OS launches some initial process(es)
  - The login process, or...
  - A shell

Cebollita Boot In Pictures

1. BIOS loads boot loader
2. Bootloader copies itself to high memory and jumps
3. Bootloader loads OS into low memory and jumps
4. OS initializes itself and starts shell
Wrap Up

- That's pretty much everything
  - As always, Cebollita favors simple above all else
    - Real systems make some other decisions
- HW5 is about implementing a machine capable of supporting all these mechanisms
  - New hardware components are introduced into datapath
  - New control is required
  - (Possibly some modification of software, e.g., the OS)