LECTURE 3.1

INPUT AND OUTPUT

COP4600

Dr. Matthew Gerber

4/4/2016

INPUT AND OUTPUT: HARDWARE

Ports and Buses (13.2)

- To do anything, I/O devices must be connected to the computer
- A *port* is a set of wires that connects an I/O device and the computer
- A device that allows other devices to hook up to it in turn is said to support *daisy-chained* devices
- A *bus* is a common set of wires used by more than one I/O device
- Often, devices connect to ports, which connect **in turn** to buses
- This arrangement can have several levels to it

Controllers

- We call the electronics that manage a port, bus or I/O device that device's *controller*
- The controller is what the operating system actually talks to
- This arrangement has several levels as well
- **Example:**
  - The operating system communicates with the PCI bus controller…
  - …which communicates across the PCI bus, establishing a channel to the ATA controller…
  - …which communicates across the ATA port, establishing a channel to the disk controller…
  - …which finally actually controls the disk
- It is the chain of *controllers* that the OS actually works with

Ports and Their Registers

- From the operating system's point of view, a port consists of four registers:
  - A **data in register** that allows the computer to get input from the device
  - A **data out register** that allows the computer to send output to the device
    - These can theoretically be combined
  - A **status** register that indicates the current state of the device, including (at least) whether the current command has completed, whether the data registers are ready, and whether there is an error
  - A **control** register used to send commands to the device

INPUT AND OUTPUT: METHODS

Ports and Polling (13.2.1)

- What actually happens when we write to a port?
  1. The OS repeatedly reads the **status** register until the port is ready
  2. The OS sets the **write** bit in the **command** register and writes a word into the **data out** register
  3. The OS sets the **ready** bit in the **command** register
  4. The device controller notices the **ready** bit in the **command** register and sets the **busy** bit in the **status** register
  5. The device controller reads the **command** register and sees the **write** bit
  6. The device controller reads the **data out** register to get the word being written
  7. The device controller performs the actual output to the device
  8. The device controller clears the **error** bit in the **status** register to indicate that the operation succeeded
  9. The device controller clears the **ready** bit in the **command** register so that the command will not immediately repeat itself
  10. The device controller clears the **busy** bit n the **status** register to indicate that it has completed its work
- All of this is for the simplest possible version of polled, busy-waiting I/O!

Interrupts in hardware (13.2.2)

- Back in Unit 1 we discussed the superiority of *interrupt-driven* I/O
- The processes required are even more complex than the process for polled I/O
- The interrupt/trap mechanism ends up handling several purposes:
  - User-process calls for I/O, and notifications for those calls
  - Notifications from hardware about I/O completion
  - Memory exceptions
  - Arithmetic exceptions (divide by zero)
  - Privilege exceptions
- The interrupt handler is typically complex, and must implement a priority scheme all its own

Direct Memory Access (13.2.3)

- In Unit 1 we also discussed *direct memory access* and its application to I/O
- For DMA to work properly, device controllers must be given their own channel to system memory, with its own control registers
- Depending on the implementation, this may slow down memory access by the CPU when it is in use
- Note that this creates its own set of complications with virtual memory!
  - When do you use virtual addresses?
  - When do you use physical addresses?

INPUT AND OUTPUT: INTERFACES

Device Drivers (13.3)

- A *device driver* is, in its original form, a piece of software intended to handle interfacing with a particular device's controller
- Device drivers quickly evolved to provide common interfaces to the kernel for classes of devices
- We now often (mis)use the term to refer to kernel modules that interface with things other than hardware devices
- Device drivers handle the particulars of interacting with a **specific device's** controller, providing the kernel with an interface that the kernel already understands as how it interacts with that **type** of device
- The devices handled by drivers can be as "remote" as printers on the other side of a network, or as fundamental as the memory and system bus controllers

Application I/O Interface (13.3)

- We don’t want applications to think individually about each I/O device any more than we want the kernel to
- We noted that the kernel interacts with device drivers to support various instances of device classes
- In turn, the kernel provides user processes with abstract programming interfaces to manage those same classes…
- …or, preferably, larger and more simplified classes
- The vast majority of these can simply be presented as files, possibly with a limited number of particular operations attached
- **Devices can become entries in the virtual file system**

I/O Device Characteristics

- **Stream versus block**
  - Transmit byte by byte, versus in groups
- **Sequential versus random**
  - Seeking impossible or irrelevant, versus able to seek
- **Synchronous versus asynchronous**
  - Predictable coordinated timing, versus arbitrary timing
- **Sharable versus dedicated**
  - Usable by multiple processes, versus a need to lock
- **Read/write versus read-only versus write-only**
- **Speed of operation**
  - Technically just a parameter but can rise to the level of a characteristic

Block and Stream Devices (13.3.1)

- Block devices act like traditional files
  - Used as a metaphor for devices that provide a notion of a space, most obviously disks and other types of memory
  - Fundamental functions are **read**, **write** and **seek**
  - Block devices can also be memory-mapped
- Character-stream devices have no memory-like space
  - Used as a metaphor for devices that accept or generate a linear stream of output or input, most obviously human input devices and printers
  - Fundamental functions are **get** and **put**
  - Often still mapped to **read** and **write** with buffering to allow multiple bytes to be read and written; that way they can still appear as files
  - **Seek** makes no sense
  - Stream devices cannot be usefully memory-mapped

Network Devices (13.3.2)

- Theoretically very different from files
- Need a unique API for addressing, opening, and maintaining connections
- Once a connection is ready, a network connection can act like a stream device
- The *socket* interface provides a file-like metaphor for reading and writing to a network connection

INPUT AND OUTPUT: SCHEDULING AND MANAGEMENT

Clocks (13.3.3)

- Hardware clocks provide three key functions:
  - Report the current time
  - Report the elapsed time
  - Set an *interval timer* to report in the future
- Simple and deceptively important
- Used by the OS to, at a *minimum*…
  - …preempt processes
  - …flush cache buffers
  - …report the failure of timed-out network operations
- How many more can you think of?
- Generally fundamental enough that it has its own interface
- Unfortunately, this interface has little standardization
  - POSIX timers exist but often offer poor performance

I/O Scheduling (13.4.1)

- We discussed I/O scheduling in the context of magnetic disks; that's only one case
- The OS needs to perform broader scheduling and prioritization of all I/O
- The OS keeps track of each device and related pending request queues in the *device status table*
- The OS then does its best to order requests for priority and efficiency
- Caching is one obvious way to improve I/O efficiency
- Let's look at two more

Buffering (13.4.2)

- Buffering has obvious uses, but also less obvious ones
- We've talked about the producer/consumer problem before, but this is a little less theoretical
- Consider a slow producer *p* and fast consumer *c*
  - *p* fills a buffer that *c* then **quickly** uses…
  - …but not **instantly**, and we want *p* to be able to keep producing
  - The solution is to maintain two buffers *a* and *b*
  - Whenever *p* fills *a*, it is switched to *b*, and *c* is given the now-full *a*
  - *c* will finish with *a* long before *p* is finished writing into *b*
  - When *p* finishes writing into *b*, we give *b* to *c* and switch *p* back to *a*
  - The cycle continues
  - This process is called *double buffering*

Spooling (13.4.4)

- A *spool* is a special-purpose buffer that serves entire coherent jobs
- Classic case is the print spool
  - A printer can only usefully handle one job at once
  - Printers are **really** slow
  - We don't want user processes to block waiting on the printer
  - The OS accepts jobs quickly on the printer's behalf, and presents them to the printer in order
  - Note that this can go through multiple levels of spooling…

NEXT TIME: LAB SESSION