- Main jobs: I/O and processing
- I/O - manage I/O operations and devices
- Lots of code in this level / device drivers here
- Two trends in I/O devices
  - Standardization of devices
  - Many new kinds of devices
- Kinds of devices
  - Storage (disks, tapes, ... block devices)
  - Communication (network, serial, bluetooth, ...)
  - Human interface (keyboard, screen, mouse, audio, ...)
  - All others! (control devices, on/off switches, ...)
- Basic Hardware on a system
  - Main Bus -- processor & memory connected
  - I/O Bus -- e.g. PCI, ISA, SCSI, IDE, ATA, SATA, ...
  - Controller, host adapter -- bus "controllers"

Basic I/O

- Memory Mapped I/O
  - Part of physical address space does not address RAM
  - Controllers/Host Adapters/Devices "listen" on addresses
  - Access to Device Registers
  - May be memory like (write & read back same value)
  - Most not like memory:
    - Write controls device or sends data
    - Read gets data
    - Write followed by read doesn’t get written data
- Special I/O instructions
  - Possibly a separate bus for I/O vs memory
  - Special I/O instructions to read/write device registers
  - Access to Device Registers
- Typical Device Registers
  - Data-in Register, Data-out Register
  - Status Register -- bits that indicate device status
  - Control Register -- writes control device
  - Direct Access Device Memory (few devices have this)

Access methods

Polling

- simple access method
- requires CPU time
- typical operation
  - read status register
  - check "ready" bit
  - if ready, read data-in or write data-out
  - repeat as necessary
- busy waiting vs regular checking
- Issues:
  - busy waiting by OS blocks user processes
  - some devices need data read quickly
  - regular checking too infrequently -> lost data

Access methods (page 2)

Interrupts

- CPU has some interrupt subsystem
- Bus controller interacts with devices and CPU for proper operation
- Often have a priority for interrupts
  - high priority need attention fast
  - low priority can do with slower processing

Operation:

- Device control register turns on interrupts
- Device control register starts I/O operation
- I/O operation completion triggers interrupt
- Bus controller manages and propagates interrupts
- CPU get interrupt, possibly vectored
- OS Code may have to figure out which device interrupted
  - Devices can share interrupt lines

Direct Memory Access Vs Programmed I/O (DMA vs PIO)

- PIO -- All data transferred by instruction access to CR/DR
- Large transfer volume / very fast devices waste CPU in PIO
- DMA allows device to put data directly into memory, cycle stealing
- I/O: tell controller where to get/put data, start operation, interrupt when done

Device Driver Subsystems

Problems:

- Different vendors do things differently
- How to isolate OS from all the specifics of each device
- How do you determine if a device is present?

Solution -- a standard device interface for Device Drivers

- Each OS has a different way of doing device drivers
  - Windows has changed over the years
- Most UNIX systems have a similar system
  - But device drivers still are different
    - Takes work to convert a FreeBSD driver to NetBSD
    - Takes more work to convert a *BSD to Linux or the other way

- Many manufacturers provide drivers for the most popular OSes
  - Windows, Mac and Linux (sometimes)

Device attributes

- Character stream vs Block
- Sequential vs random access
- Synchronous vs asynchronous
- Sharable vs dedicated
- Speed of operation
- R/W vs R only vs W only

UNIX standards:

- block device -- a device that is random access, block based
- character device -- a character stream, may have a way to move "pointer"
  - All block devices under UNIX also have a character device
- Some devices, e.g. clocks don’t appear as a device
- UNIX abstraction of "Top Half" and "Bottom Half"
  - Top half -- kernel core, Shared handling ...
  - Bottom half -- device drivers, normally interrupt driven
- Interface between them is fixed
- Has a general catch all (ioctl)
  - NetBSD calls this structure a "softc"
- At OS startup, all devices attached are located and attached

NetBSD (*BSD) interface

Auto configuration

- old ISA style -- go see if the registers respond
- PCI -- device registers with controller, controller tells OS
- NetBSD builds a tree of devices to find/use
  - config_search()
  - xxx_match(), xxx_attach(), xxx_detach(), xxx_activate()

Normal Operation

- xxx_open() -- open the device for operations
- xxx_close() -- close the device
- xxx_read() -- read via the sequential/character interface
- xxx_write() -- write via the sequential/character interface
- xxx_strategy() -- read or write via the block interface
- xxx_ioctl() -- catch all, any other kind of device operation

All devices implement the auto configuration part (some flavor)

All devices implement open/close

Other calls implemented if they make sense.

- "Satlink" driver: no write/strategy

Windows Driver Model (plug-and-play)

- Hardware Abstraction Layer -- HAL
  - Hides registers and so forth from drivers
  - READ_PORT_*, READ_REGISTER_*, WRITE_*
- Registry based information
  - Each device needs an entry
  - Each driver needs an entry
- Multi-level driver structure under the I/O Manager
- Classes of devices get a "Class Driver" object
- Each device gets an object representing it
- Primary method of communication:
  - IRP -- I/O Request Packet
  - IRP may be processed by several objects
- Driver Routines
  - DriverEntry(), Reinitialize(), Unload(), Shutdown()
  - Open(), Close() - required
  - Read(), Write(), DeviceIoControl() -- optional
  - More complicated interaction than UNIX

I/O Processing

- Block devices
  - Use by OS, buffered, FS access, ...
  - Opened directly by user (/dev/rwd0a)
    - "raw", sequential, not buffered
- Clocks and timers
  - Uses interrupts, not often available to users directly
    - e.g. setitimer(2) is closest use in UNIX
- Blocking vs Nonblocking I/O
  - Blocked request: e.g. Read on a character device
    - sys_call Read:
    - calls xxx_read()
    - nothing available, block()
    - Wait for interrupt
    - At interrupt, add data to driver (queue?)
    - unblock a waiting process
    - driver reads data, sends to user
    - returns

I/O Processing (page 2)

- Non-Blocked request: e.g. read on character device
  - calls sys_read()
  - calls xxx_read()
  - nothing available, return error EWOULDBLOCK
  - returns to user
- User could now poll() driver or use select()
  - When I/O happens (interrupt) data collected
  - notification to poll/select of available data
  - User calls sys_read() again, now gets data

- OS may provide vectored I/O
  - e.g. UNIX readv/writev
  - vector (list) of buffers to fill or write
  - May provide for faster I/O, lower syscall overhead

I/O Scheduling

- Ordering I/O requests
  - Disks (e.g. Elevator Algorithm)
  - Network devices
- In General:
  - Shared devices may order requests
  - non-shared devices as received

I/O Buffering

- Buffers for speed mismatch (e.g. serial vs disk)
  - Double buffering -- decouples producer / consumer
- Buffers for different data transfer sizes
- Buffers for copy semantics for I/O
  - e.g. disk write -- copy user buffer
  - future changes to user buffer don’t change written data

I/O Subsystem (page 2)

I/O Caching

- Caching and Buffers are different
  - Buffers in disk write may not delay write to disk
  - Caching a disk write may delay write to disk
- Can use same "pool of memory"

Spooling

- "Extended Buffering" for a device that can’t interleave data
- Often done by a userland daemon
- Some OSes include spooling in the OS
- Prime devices: printers

Error Handling

- Don’t want a device malfunction to kill entire OS
- Error handling must be an integral part of the OS design
- Errors often returned to user process

I/O Subsystem (page 3)

Performance

- I/O performance a major component of OS performance
- Interrupt processing is expensive (similar to a syscall)
- Small amounts of busy waiting may be faster than interrupts
- Network cards could cause high CPU load
  - Some network cards now have DMA

Performance improvements

- reduce number of context switches
  - Windows UI in kernel space
- reduce number of times data is copied
- reduce frequency of interrupts by large transfers
- increase concurrence by using DMA
- where possible move operations do device controllers / co-processors
- balance CPU, memory subsystem, bus, and I/O performance

I/O subsystems is typically the largest area of "churn" for an OS

- Many changes due to new devices

Read sections 13.5 and 13.6 (not covered in class)