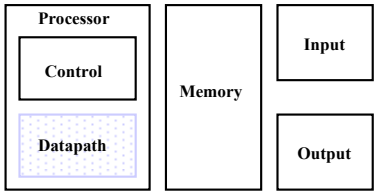
I/O
Devices and Drivers

Tore Larsen, University of Tromsø
based on and including
original material developed by
Kai Li, Princeton University
Including material from:
UC Berkeley (Patterson, Kubiatowicz et.al)
Andy Tanenbaum, VU/Amsterdam
Pål Halvorsen, University of Oslo

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



- Today's Topic: I/O
- *Five separate boxes? Some glue must be missing here!*

Content

- Introduction, overview
- Characteristics of I/O devices
- Interconnection of I/O devices to CPU/Memory
- Principles of I/O software
- I/O Software layers
- A couple of devices

- *Deadlocks handled previously*
- *Disks to be handled later*

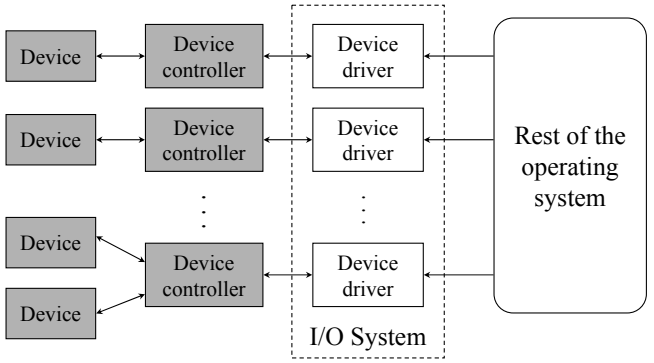
What is I/O?

- Transfer of data to/from I/O device
- Non-overlapped operation
  - Processor "waits" for I/O to complete
- Overlapped operation
  - Processor proceeds with other tasks until I/O complete
- Application Programming Interface (API)
  - Blocking I/O operations
    - Process initiating I/O blocks until I/O completed
  - Non-blocking I/O opeartions
    - Process initiating I/O proceeds without waiting for I/O to complete

Definitions and General Method

- Overhead
  - CPU time to initiate operation (cannot be overlapped)
- Latency
  - Time to perform I/O operation
- Bandwidth
  - Rate of I/O transfer
- General method
  - Abstraction of byte transfers
  - Batch transfers into block I/O for efficiency to prorate overhead and latency over a large unit

I/O—"Bird's Eye View"



I/O Devices

- Keyboard, mouse, microphone, joystick, magnetic-card reader, graphic tablet, scanner, videocam, loudspeaker, printer, display, display-wall, LAN-card, DVD, disk, floppy, wind-sensor, etc. etc.
- Diversity: Many, widely differing device-types
  - Devices within each type also differs.
- Physical motion. Motion of media, mechanism, or information (e.g. delay lines)
- Varying, often slow access & transfer compared to CPU.
  - Some device-types require very fast access & transfer (e.g. graphic display, emerging high-speed networks)
- Malfunctions: Intermittent and complete
- *Expect to see new types of I/O devices, and new application of old types*

I/O Devices

- Block devices: store information in fixed-size blocks, each one with its own address
  - common block sizes: 512 B – 64 KB
  - addressable
  - it is possible to read or write each block independently of all others
  - e.g., disks, floppy, tape, CD, DVD, ...
- Character devices: delivers or accepts a stream of characters, without regard to any block structure
  - it is not addressable and does not have any seek operation
  - e.g., keyboards, mice, terminals, line printers, network interfaces, and most other devices that are not disk-like...
- Does all devices fit in?
  - clocks and timers
  - memory-mapped screens

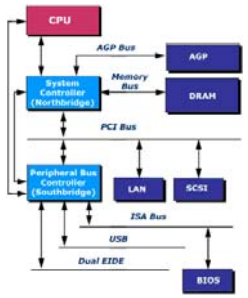
Device controllers

- Piece of HW that
  - Controls one or more devices
- May be located on the host
  - On motherboard (PC keyboard controller
  - PC-card, Host Bus Adaptor (PCI SCSI-card)
- And embedded with device
  - Most disks have additional embedded controller

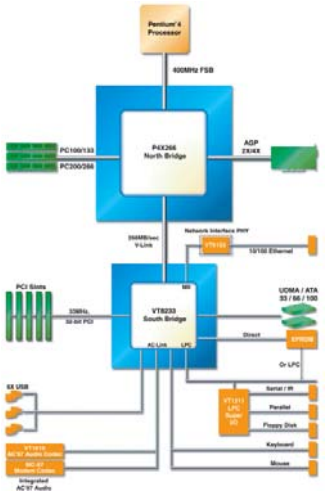
Four Basic Questions

- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

How are devices connected to CPU/Memory Example PC North/South type chipset



- Northbridge: Four-way intersection, handles traffic from towards 1) CPU/caches, 2) Memory, 3) Advanced Graphics Port, and 4) PCI-bus
  - "Concurrent" memory access → Internal buffers
- Southbridge: Legacy ports, USB, IEEE 1394, ++
- *Which buses are faster?*
- *Where are keyboard, mouse, disk, LAN, graphics controller, etc. connected?*



- Newer PC chipset (VIA 8233)
- PCI moved to Southbridge
- Faster Southbridge—Northbridge link
- Integrated 10/100 Ethernet through Southbridge

How is SCSI-disk connected? Where are data cached/buffered?

- SCSI controller on motherboard or PCI-card *(Host bus adaptor)*
  - Provides a SCSI "bus"
  - May or may not cache data
- SCSI disk connected on SCSI bus
  - Embedded disk controller
    - SCSI bus interface
    - Internal caching
    - Internal request ordering
  - Disk mechanism
- There is additional buffering/caching in
  - OS (host memory)
  - CPU caches

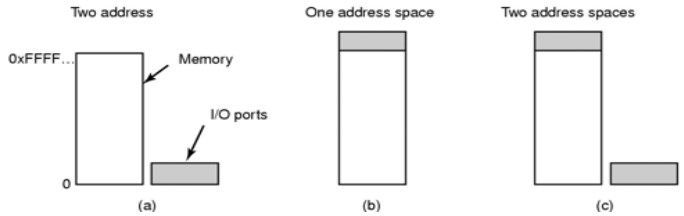
Four Basic Questions

- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

Accessing Device Controller Registers

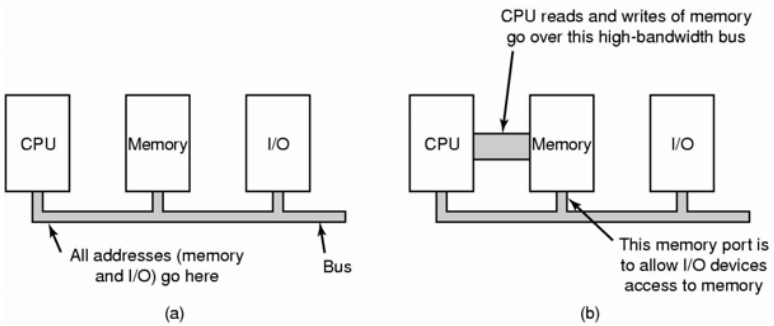
- Memory mapped I/O
  - Device registers mapped into regular address space
  - Use regular move (assignment) instructions to read/write registers
  - Use memory protection mechanism to protect device registers
- Port I/O
  - Devices registers mapped onto "ports;" a separate address space
  - Use special I/O instructions to read/write ports
  - Make I/O instructions available only in kernel/supevisor mode

Memory-Mapped I/O (1)



- Separate I/O and memory space
- Memory-mapped I/O
- Hybrid

Memory-Mapped I/O (2)



(a) A single-bus architecture
(b) A dual-bus memory architecture

Memory Mapped I/O vs. Port I/O

- Ports:
  - special I/O instructions are CPU dependent
- Memory mapped:
  - \+ memory protection mechanism allows greater flexibility than protected instructions
  - \+ may use all memory reference instructions for I/O
  - – cannot cache device registers (must be able to selectively disable caching)
  - – I/O devices do not see the memory address - how to route only the right memory address onto slower peripheral buses (may initiate bridge at setup time to transfer required address areas)
- Intel Pentium use a hybrid
  - – address 640K to 1M is used for memory mapped I/O data buffers
  - – I/O ports 0 to 64K is used for device control registers

Memory Mapped I/O vs. Port I/O Strengths and weaknesses

- Ports: Special I/O instructions won't be available through HLL compiler
- MM: Memory protection mechanism allows greater flexibility than protected instructions
- MM: May use all memory reference instructions for I/O
- MM: Cannot cache device registers
  - Must be able to selectively disable caching
- MM: How to route only the right memory address onto slower peripheral buses
  - May initiate bridge at setup time to transfer required address areas

Four Basic Questions

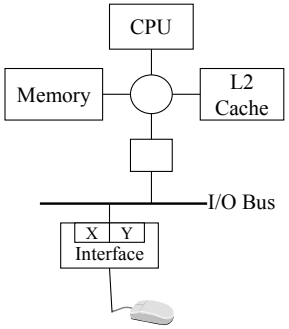
- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

How are data transmissions controlled?

- Programmed I/O (PIO)—The processor handles the transfers
  - Transfers data between registers and device
- DMA—The adaptor accesses main memory
  - Transfers blocks of data between memory and device
- Channel—Simple specialized peripheral processor dedicated to I/O.
  - Handles most transmission, but less control. Shared memory. No private memory.
- Peripheral Processor (PPU)—General processor dedicated to I/O control and transmission.
  - Shared and private memory. (CDC 6600, 1964)

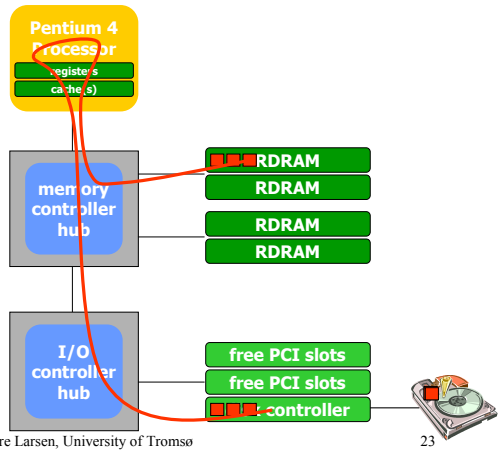
Programmed I/O "Slow" Input Device

- Device
  - Data registers
  - Status register (ready, busy, interrupt, …)
- A simple mouse design
  - Put (X, Y) in data registers on a move
  - Interrupt
- Perform an input
  - On an interrupt
    - reads values in X, Y registers
    - sets ready bit
    - wakes up a process/thread or execute a piece of code



PIO

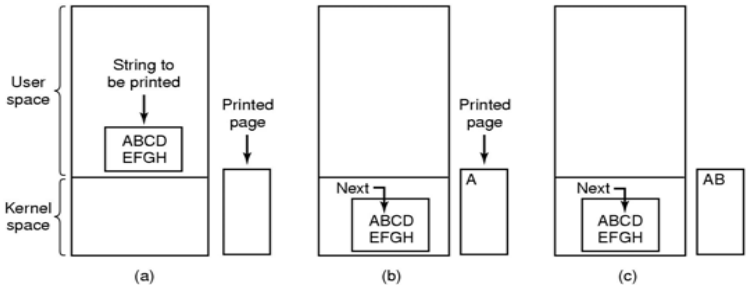
- Device delivers data to controller
- PIO:
  - CPU reads data from controller buffer to register
  - CPU writes register to memory location
- CPU is busy moving data



Programmed I/O Output Device

- Device
  - Data registers
  - Status registers (ready, busy, …)
- Perform an output
  - Polls the busy bit
  - Writes the data to data register(s)
  - Sets ready bit
  - Controller sets busy bit and transfers data
  - Controller clears the ready bit and busy bit

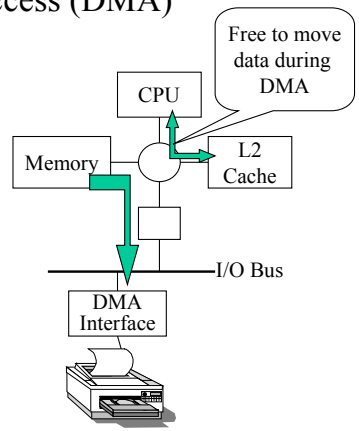
Programmed I/O



Steps in printing a string

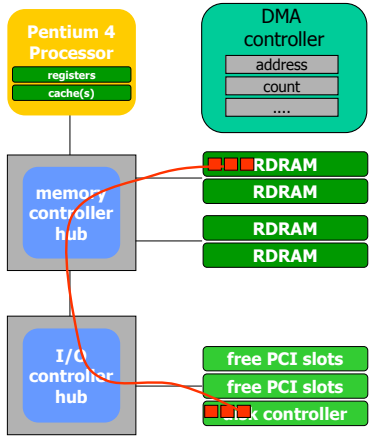
Direct Memory Access (DMA)

- Perform DMA from host CPU
  - Device driver call (kernel mode)
  - Wait until DMA device is free
  - Initiate a DMA transaction (command, memory address, size)
  - Block
- Interrupt handler (on completion)
  - Wakeup the blocked process
- DMA interface
  - DMA data to device (size--; address++)
  - Interrupt on completion (size == 0)



DMA

- Device delivers data to controller
- DMA:
  1. set up DMA controller
  2. DMA controller initiates transfer
  3. data is moved (increasing address, reducing count)
  4. disk controller notifies DMA controller when finished (count = 0)
  5. DMA controller interrupts
- CPU is free
- Cycle stealing



PIO vs. DMA

- DMA:
  - \+ supports large transfers, latency of requiring bus is amortized over hundreds/thousands of bytes
  - – may be expensive for small transfers
  - – overhead to handle virtual memory and cache consistence
  - o is common practice
- PIO:
  - – uses the CPU
  - – loads data into registers and cache *(May well be a "plus")*
  - \+ potentially faster for small transfers with carefully designed software

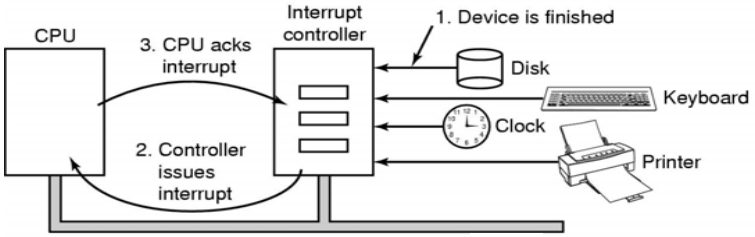
Four Basic Questions

- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

Synchronization: Interrupts versus polling

- Polling: Processor polls the device while waiting for I/O to complete.
  - Wastes cycles. Inefficient.
- Interrupt: Device asserts interrupt when I/O completed.
  - Frees processor to move on to other tasks.
  - Interrupt processing is costly and introduces latency penalty.
- Possible strategy
  - Apply interrupts, but reduce interrupts frequency through careful driver/controller interaction

Interrupts Revisited



Interrupts

- Interrupt controller (IC) receives interrupt from device
  - Checks that interrupts are enabled
  - Checks that no other interrupt is being processed, no interrupt pending, and no higher priority simultanous interrupt
- IC puts a index number identifying the device on the address lines & asserts CPUs interrupt signal
- CPU
  - Interrupt checking is part of fetch-decode-execute loop
  - HW actions: Save PC, PSW; load new PC from interrupt vector table; set PSW for disabled interrupts and supervisor/kernel mode
- Interrupt service routine (interrupt handler)
  - Acknowledge interrupt
  - Copy saved state to process table
  - Run interrupt service routine
  - Unblock device driver
  - Select and set up next process to run

OS goals

- Provide higher level, device independent I/O operation
  - Application program should be able to access any device without modification
- Uniform naming
  - UNIX style path-name addressing
- Error handling
  - Errors should be handled close to hardware,
  - In concert with employed embedded error handling
- Sharing/Protection of devices

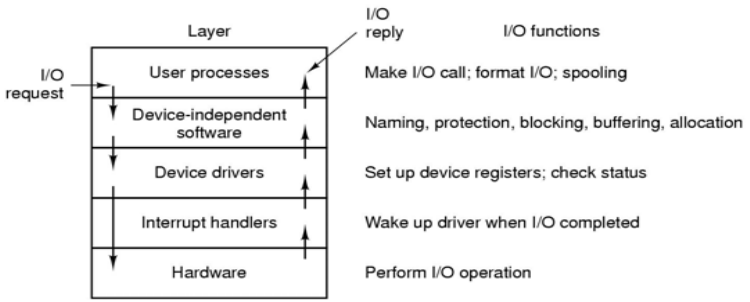
Higher level functionality

- Device independent API
  - CreateFile
  - ReadFile
  - CloseHandle
- Device-type-dependent API for
  - Disk
  - Tape
  - Sound cards
  - For Win32, more than 100 device-type-dependant APIs

I/O Software Layers Organization

User-level I/O software
Device independent operating system software
Device drivers
Interrupt handlers
Hardware

Layered I/O Software



Interrupt handlers (1)

- Should be hidden
  - Driver starting I/O blocks until I/O has occurred (correct interrupt received)
- Handler started on receipt of correct interrupt
  - Corresponding driver is made runnable (unblocks)
- Details of interrupt handling varies among different processors/computers

Interrupt Handlers (2)
Detailed Example SW steps

1. Save registers not already saved by HW
2. Set up context for service procedure
3. Set up stack for service procedure
4. Ack interrupt controller, reenable interrupts
5. Copy saved registers to proc. table
6. Run service procedure
7. Set up MMU context for next process to run
8. Load new process' registers
9. Start running new process

Device Drivers

- Contains the device dependent code
- One device driver per device type or class of closely related devices
- Accepts abstract requests from device independent I/O software
- Translates to concrete device actions
- Senses and controls the device (actually the device controller) accordingly
- Typically blocks if it has to wait long for device to complete
- May also proceed without blocking, depending on device/delay

Device Driver Design Issues

- Operating system and driver communication
  - Commands and data between OS and device drivers
- Driver and hardware communication
  - Commands and data between driver and hardware
- Driver operations
  - Initialize devices
  - Interpreting commands from OS
  - Schedule multiple outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Device Driver Interface

- Open(deviceNumber)
  - Initialization and allocate resources (buffers)
- Close(deviceNumber)
  - Cleanup, de-allocate, and possibly turnoff
- Device driver types
  - Block: fixed sized block data transfer
  - Character: variable sized data transfer
  - Terminal: character driver with terminal control
  - Network: streams for networking

Block Device Interface

- read(deviceNumber, deviceAddr, bufferAddr)
  - transfer a block of data from “deviceAddr” to “bufferAddr”
- write(deviceNumber, deviceAddr, bufferAddr)
  - transfer a block of data from “bufferAddr” to “deviceAddr”
- seek(deviceNumber, deviceAddress)
  - move the head to the correct position
  - usually not necessary

Character Device Interface

- read(deviceNumber, bufferAddr, size)
  - reads “size” bytes from a byte stream device to “bufferAddr”
- write(deviceNumber, bufferAddr, size)
  - write “size” bytes from “bufferSize” to a byte stream device

Example Unix Device Driver Interface Entry Points

- init(): Initialize hardware
- start(): Boot time initialization (require system services)
- open(dev, flag, id): initialization for read or write
- close(dev, flag, id): release resources after read and write
- halt(): call before the system is shutdown
- intr(vector): called by the kernel on a hardware interrupt
- read/write calls: data transfer
- poll(pri): called by the kernel 25 to 100 times a second
- ioctl(dev, cmd, arg, mode): special request processing

Device-Independent I/O Software

Uniform interfacing towards device drivers
Device naming
Device protection
Providing device-independent block size
Buffering
Storage allocation on block devices
Allocating and releasing dedicated devices
Error reporting

Why Buffering

- Speed mismatch between the producer and consumer
  - Character device and block device, for example
- Adapt different data transfer sizes
  - Packets vs. streams
- Support copy semantics
- Deal with address translation
  - I/O devices see physical memory, but programs use virtual memory
- Spooling
  - Avoid deadlock problems
- Caching
  - Avoid I/O operations

Buffering

- No buffering (unbuffered I/O)
  - One interrupt per char/block
- User space buffering
  - User process blocks until buffer full or I/O completed
  - How about paging
- Kernel space buffering
  - Transfer kernel space buffer → user space buffer in one go
- Double buffering in kernel
  - Two kernel space buffers, second buffer available while first is copied

Detailed Steps of Blocked Read

1. A process issues a read call which executes a system call
2. System call code checks for correctness and cache
3. If it needs to perform I/O, it will issues a device driver call
4. Device driver allocates a buffer for read and schedules I/O
5. Controller performs DMA data transfer, blocks the process
6. Device generates an interrupt on completion
7. Interrupt handler stores any data and notifies completion
8. Move data from kernel buffer to user buffer and wakeup blocked process
9. User process continues

Asynchronous I/O

- Why do we want asynchronous I/O?
  - Life is simple if all I/O is synchronous
- How to implement asynchronous I/O?
  - On a read
    - copy data from a system buffer if the data is there Otherwise, block the current process
  - On a write
    - copy to a system buffer, initiate the write and return

Synchronous or Asynchronous Coordination in time at different levels

Among processes using message passing API	Blocking- or non-blocking send / receive system call semantics
Between device driver and device	Polling or interrupt driven I/O
On the transmission medium (wire)	Synchronous (with respect to a shared timing signal) or asynchronous (e.g. handshake protocol).

Error reporting

- Bulk of error handling is done by device drivers and device controllers
- After controller and driver gives in, device independent I/O SW reports the error

User-Space I/O Software

- Library procedures that are linked with user-level programs
  - Puts parameters at appropriate places
  - Execute a system call, typically trapping to invoke O/S
- Spooling
  - Provides sharing of devices that may be shared over time, but not concurrently. E.g. printers
  - Deamon process
  - Spooling directory
  - Print file generated and put in spooling directory
  - Deamon has exclusive access to printer, and prints at it's convenience
  - Spoolers are also used for network traffic

Detailed Steps of Blocked Read

- A process issues a read call which executes a system call
- System call code checks for correctness and cache
- If it needs to perform I/O, it will issues a device driver call
- Device driver allocates a buffer for read and schedules I/O
- Controller performs DMA data transfer, blocks the process
- Device generates an interrupt on completion
- Interrupt handler stores any data and notifies completion
- Move data from kernel buffer to user buffer and wakeup blocked process
- User process continues

Think About Performance

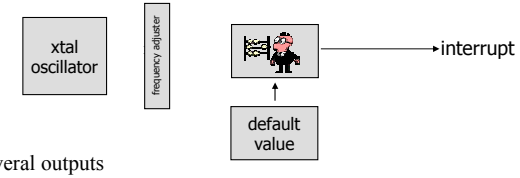
- A terminal connects to computer via a serial line
  - Type character and get characters back to display
  - RS-232 is bit serial: start bit, character code, stop bit (9600 baud)
- Do we have any cycles left?
  - 10 users or 10 modems
  - 900 interrupts/sec per user
  - Overhead of handing an interrupt = 100 μsec

Other Design Issues

- Build device drivers
  - statically
  - dynamically
- How to download device driver dynamically?
  - load drivers into kernel memory
  - install entry points and maintain related data structures
  - initialize the device drivers

Example: Clocks

- Old, simple clocks used power lines and caused an interrupt at every voltage pulse (50 - 60 Hz)
- New clocks use
  - quartz crystal oscillators generating periodic signals at a very high frequency
  - counter which is decremented each pulse - if zero, it causes an interrupt
  - register to load the counter



- May have several outputs
- Different modes
  - one-shot - counter is restored only by software
  - square-wave - counter is reset immediately (e.g., for clock ticks)


Examples: Clocks

- HW only generates clock interrupts
- It is up to the clock software (driver) to make use of this
  - Maintaining time-of-day
  - Preventing processes from running longer than allowed
  - Accounting for CPU usage
  - Handling ALARM system call
  - Providing watchdog timers
  - Doing profiling, monitoring, and statistics gathering

Terminals

- Memory mapped terminals
  - Character or bit oriented
- RS-232 connected terminals
  - Glass TTY (“dumb”) or intelligent terminals
- Network connected terminals
  - X Terminals

OS, Network Interfaces
Ref.: Druschel & Peterson

- Network
  - Multiplexed among multiple applications
  - Implements NW protocols
  - Provides a standardized, abstract communication interface
    - Portability
    - Interoperability

NW-OS Problems

- High per-byte processing overhead
  - Causes loss of performance & QoS
  - Memory hierarchies depend on locality
  - NW subsystems fails to isolate & optimize common case
  - Improper resource scheduling

Principles

- Coordinated design
- Early demultiplexing
- Path-oriented structures
- Integrated layer processing
- User application-level framing

Network Interface Design

- I/O versus memory bus
  - Shorter path to memory (latency)
  - Higher bandwidth on memory bus
  - Integration/interaction with caches & cache coherency protocols (DSM)
- DMA versus PIO
  - Small or large transfers
  - Memory vs. Registers
  - Virtual memory - cache consistency
- Interrupts versus polling
  - Removing unnecessary interrupts
  - Poll first, then interrupt

Pointers

- III @ Intel http://developer.intel.com/design/iio/
- Paralell I/O http://www.cs.dartmouth.edu/pario/
- For a pointed quiz-like syllabus, you may check Mark Smothermans “Topical Schedule” for Clemson’s CPSC 422
  - http://www.cs.clemson.edu/~mark/422.html