Review: Want Standard Interfaces to Devices

- **Block Devices**: e.g. disk drives, tape drives, CD rom
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible

- **Character Devices**: e.g. keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing

- **Network Devices**: e.g. Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    » Separates network protocol from network operation
    » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

Review: How Does User Deal with Timing?

- **Blocking Interface**: “Wait”
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data

- **Non-blocking Interface**: “Don’t Wait”
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing

- **Asynchronous Interface**: “Tell Me Later”
  - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user

Goals for Today

- Finish Discussing I/O Systems
  - Hardware Access
  - Device Drivers

- Disk Performance
  - Hardware performance parameters
  - Queuing Theory

- File Systems
  - Structure, Naming, Directories, and Caching

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.
How does the processor talk to the device?

- CPU interacts with a Controller
  - Contains a set of registers that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - Example from the Intel architecture: out 0x21, AL
  - Memory mapped I/O: load/store instructions
    - Registers/memory appear in physical address space
    - I/O accomplished with load and store instructions

Transferring Data To/From Controller

- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- Sample interaction with DMA controller (from book):

Memory-Mapped Display Controller Example

- Memory-Mapped:
  - Hardware maps control registers and display memory to physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - Addr: 0x8000F000–0x8000FFFF
  - Writing graphics description to command-queue area
    - Say enter a set of triangles that describe some scene
    - Addr: 0x80010000–0x8001FFFF
- Can protect with page tables

A Kernel I/O Structure
Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the ioctl() system call

- Device Drivers typically divided into two pieces:
  - **Top half**: accessed in call path from system calls
    - Implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    - This is the kernel's interface to the device driver
      - Top half will start I/O to device, may put thread to sleep until finished
  - **Bottom half**: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error

- **I/O Interrupt**:
  - Device generates an interrupt whenever it needs service
  - Handled in bottom half of device driver
    - Often run on special kernel-level stack
  - **Pro**: handles unpredictable events well
  - **Con**: interrupts relatively high overhead

- **Polling**:
  - OS periodically checks a device-specific status register
    - I/O device puts completion information in status register
    - Could use timer to invoke lower half of drivers occasionally
  - **Pro**: low overhead
  - **Con**: may waste many cycles on polling if infrequent or unpredictable I/O operations

- Actual devices combine both polling and interrupts
  - For instance: High-bandwidth network device:
    - Interrupt for first incoming packet
    - Poll for following packets until hardware empty

Administrivia

- **Group Evaluations (Both Projects 1 and 2)**
  - These MUST be done: you will get a ZERO on your project score if you don't fill them out
  - We will be asking you about them, so make sure you are careful to fill them out honestly

- **Thursday sections**
  - Fill out a survey form to see how class is going
  - Give you an opportunity to give feedback

- **Other things**
  - Group problems? Don't wait.
  - Talk to TA/talk to me
    - Let's get things fixed!
Hard Disk Drives

Properties of a Hard Magnetic Disk

- Independently addressable element: sector
  - OS always transfers groups of sectors together—"blocks"
  - A disk can access directly any given block of information it contains (random access). Can access any file either sequentially or randomly.
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
  - A sector is the smallest unit that can be read or written
- Zoned bit recording
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location

Properties of a Hard Magnetic Disk

- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time
- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track

Disk I/O Performance

- Performance of disk drive/file system
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - Software paths (can be loosely modeled by a queue)
    - Hardware controller
    - Physical disk media
  - Queuing behavior:
    - Can lead to big increases of latency as utilization approaches 100%

Magnetic Disk Characteristic

- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time
- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track
Typical Numbers of a Magnetic Disk

- Average seek time as reported by the industry:
  - Typically in the range of 8 ms to 12 ms
  - Due to locality of disk reference may only be 25% to 33% of the advertised number
- Rotational Latency:
  - Most disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- Transfer Time is a function of:
  - Transfer size (usually a sector): 512B – 1KB per sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
    - Typical values: 2 to 50 MB per second
- Controller time depends on controller hardware
- Cost drops by factor of two per year (since 1991)

Disk Performance

- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms, avg rotational delay of 4ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- Random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 10ms to fetch/put data: 100 KByte/sec
- Random place in same cylinder:
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 5ms to fetch/put data: 200 KByte/sec
- Next sector on same track:
  - Transfer (0.25ms): 4 MByte/sec
- Key to using disk effectively (esp. for filesystems) is to minimize seek and rotational delays

Disk Tradeoffs

- How do manufacturers choose disk sector sizes?
  - Need 100-1000 bits between each sector to allow system to measure how fast disk is spinning and to tolerate small (thermal) changes in track length
- What if sector was 1 byte?
  - Space efficiency - only 1% of disk has useful space
  - Time efficiency - each seek takes 10 ms, transfer rate of 50 – 100 Bytes/sec
- What if sector was 1 KByte?
  - Space efficiency - only 90% of disk has useful space
  - Time efficiency - transfer rate of 100 KByte/sec
- What if sector was 1 MByte?
  - Space efficiency - almost all of disk has useful space
  - Time efficiency - transfer rate of 4 MByte/sec

Introduction to Queuing Theory

- What about queuing time??
  - Let’s apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Little’s Law:
  - Mean # tasks in system = arrival rate x mean response time
  - Observed by many, Little was first to prove
  - Simple interpretation: you should see the same number of tasks in queue when entering as when leaving.
- Applies to any system in equilibrium, as long as nothing in black box is creating or destroying tasks
  - Typical queuing theory doesn’t deal with transient behavior, only steady-state behavior
Background: Use of random distributions

- Server spends variable time with customers
  - Mean (Average) \( m_1 = \sum p(T) \times T \)
  - Variance \( \sigma^2 = \sum p(T) \times (T - m_1)^2 = \sum p(T) \times T^2 - m_1 \)
  - Squared coefficient of variance: \( C = \sigma^2 / m_1^2 \)

  Aggregate description of the distribution.

- Important values of \( C \):
  - No variance or deterministic \( \Rightarrow C=0 \)
  - “memoryless” or exponential \( \Rightarrow C=1 \)
    - Past tells nothing about future
    - Many complex systems (or aggregates)
      well described as memoryless
  - Disk response times \( C = 1.5 \) (majority seeks < avg)

A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium: No limit to the queue
  - Time between successive arrivals is random and memoryless

- Parameters that describe our system:
  - \( \lambda \): mean number of arriving customers/second
  - \( T_{\text{ser}} \): mean time to service a customer ("m1")
  - \( C \): squared coefficient of variance = \( \sigma^2 / m_1^2 \)
  - \( \mu \): service rate = \( 1 / T_{\text{ser}} \)
  - \( u \): server utilization \( (\lambda/\mu) = \lambda \times T_{\text{ser}} \)

- Parameters we wish to compute:
  - \( T_q \): Time spent in queue
  - \( L_q \): Length of queue = \( \lambda \times T_q \) (by Little’s law)

- Results:
  - Memoryless service distribution (\( C = 1 \)):
    - Called M/M/1 queue: \( T_q = T_{\text{ser}} \times u / (1 - u) \)
  - General service distribution (no restrictions), 1 server:
    - Called M/G/1 queue: \( T_q = T_{\text{ser}} \times \frac{1}{2}(1+C) \times u / (1 - u) \)

A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed (\( C=1.0 \))
  - Avg. service = 20 ms (From controller+seek+rot+trans)

- Questions:
  - How utilized is the disk?
    - Ans: server utilization, \( u = \lambda T_{\text{ser}} \)
  - What is the average time spent in the queue?
    - Ans: \( T_q \)
  - What is the number of requests in the queue?
    - Ans: \( L_q \)
  - What is the avg response time for disk request?
    - Ans: \( T_{\text{sys}} = T_q + T_{\text{ser}} \)

- Computation:
  - \( \lambda \) (avg # arriving customers/s) = 10/s
  - \( T_{\text{ser}} \) (avg time to service customer) = 20 ms (0.02s)
  - \( u \) (server utilization) = \( \lambda \times T_{\text{ser}} = 10/s \times 0.02s = 0.2 \)
  - \( T_q \) (avg time/customer in queue) = \( T_{\text{ser}} \times u / (1 - u) = 20 \times 0.2 / (1 - 0.2) = 20 \times 0.25 = 5 \) ms (0.005s)
  - \( L_q \) (avg length of queue) = \( \lambda \times T_q = 10/s \times 0.05s = 0.05 \)
  - \( T_{\text{sys}} \) (avg time/customer in system) = \( T_q + T_{\text{ser}} = 25 \) ms

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?

  User Requests: 1,2,3,4,5,6,7,8,9,10

- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk \( \Rightarrow \) Very long seeks

- SSTF: Shortest seek time first
  - Pick the request that’s closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation

- SCAN: Circular-Scan: only goes in one direction
  - No starvation, but retains flavor of SSTF
  - S-SCAN: Circular-Scan: only goes in one direction
    - Skips any requests on the way back
    - Fairer than SCAN, not biased towards pages in middle
Building a File System

- **File System**: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

- **File System Components**
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc

- **User vs. System View of a File**
  - User’s view:
    - Durable Data Structures
  - System’s view (system call interface): Collection of Bytes (UNIX)
  - System’s view (inside OS): Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)

- **Block size ≥ sector size; in UNIX, block size is 4KB**

Translating from User to System View

- **What happens if user says: give me bytes 2—12?**
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block

- **What about: write bytes 2—12?**
  - Fetch block
  - Modify portion
  - Write out Block

- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time

- From now on, file is a collection of blocks

Disk Management Policies

- **Basic entities on a disk**
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files (next lecture)

- **Access disk as linear array of sectors. Two Options:**
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - First case: OS/BIOS must deal with bad sectors
    - Second case: hardware shields OS from structure of disk

- **Need way to track free disk blocks**
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk

- **Need way to structure files: File Header**
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files’ disk blocks to match access and usage patterns

Designing the File System: Access Patterns

- **How do users access files?**
  - Need to know type of access patterns user is likely to throw at system

- **Sequential Access: bytes read in order (“give me the next X bytes, then give me next, etc”)**
  - Almost all file access are of this flavor

- **Random Access: read/write element out of middle of array (“give me bytes i—j”)**
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don’t want to have to read all bytes to get to the middle of the file

- **Content-based Access: (“find me 100 bytes starting with JOSEPH”)**
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)
Designing the File System: Usage Patterns

- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: Beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?
- Digression, danger of predicting future:
  - In 1950's, marketing study by IBM said total worldwide need for computers was 7!
  - Company (that you haven't heard of) called "GenRad" invented oscilloscope; thought there was no market, so sold patent to Tektronix (bet you have heard of them!)

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    » Analogous to base+bounds in virtual memory
  - User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    » What if not enough contiguous space for new file?
  - File Header Contains:
    » First block/LBA in file
    » File size (# of blocks)
  - Pros: Fast Sequential Access, Easy Random Access
    - Cons: External Fragmentation/Hard to grow files
      » Free holes get smaller and smaller
      » Could compact space, but that would be really expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

How to organize files on disk (continued)

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk
  - File Header
  - Null
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    » No attempt to allocate contiguous blocks
- MSDOS used a similar linked approach
  - Links not in pages, but in the File Allocation Table (FAT)
    » FAT contains an entry for each block on the disk
    » FAT Entries corresponding to blocks of file linked together
- Compare with Linked List Approach:
  » Sequential access costs more unless FAT cached in memory
  » Random access is better if FAT cached in memory

How to Organize Files on Disk (continued)

- Third Technique: Indexed Files (Nachos, VMS)
  - System allocates file header block to hold array of pointers big enough to point to all blocks
    » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index
    Random access is fast
  - Cons: Clumsy to grow file bigger than table size
    Still lots of seeks: blocks may be spread over disk
Where do we still have to go?

- Still don't have good internal file structure
  - Want to minimize seeks, maximize sequential access
  - Want to be able to handle small and large files efficiently
- Don't yet know how to name/locate files
  - What is a directory?
  - How do we look up files?
- Don't yet know how to make file system fast
  - Must figure out how to use caching
- Will address these issues next time....

Summary

- I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)
- Disk Performance:
  - Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Transfer time: spec of disk depends on rotation speed and bit storage density
- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency \( \to \infty \)
  \[ T_q = T_{ser} \times \frac{1}{2}(1+C) \times \frac{u}{1-u} \]
- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access