Announcements:

- Papers posted online: not required reading, just for reference
- File posted online called "introstuff": material relevant to first lectures in course; includes obituaries for people from 1960s, etc.
- IMPORTANT: Midterm is 2 weeks from today (4/15)

I/O devices (continued):

Digital Video Disk (DVD)

- same size as CD, but dots are closer together; higher dot densities readable with blue or green (versus red) lasers
- capacity: 4.5GB, 9GB, 18GB (approx)
- can be one- or two-sided, and have 1 or 2 layers
- thickness 0.6mmx2, beam spot size 1.32um, light 650nm
- working distance 1.7mm, max transfer rate 10.08Mb/s

Blue Ray and HD Disks

- Sony led Blue Ray; Toshiba led HD
- HD more compatible with CD, but has less capacity
- Blue Ray: 25 GB/layer
- HD Disk: 15 GB/layer
- thickness: 0.1mm+1.1mm, beam spot size 0.58um, light 405nm; violet light
- access efficiency: on disk typically takes 5-15 ms overhead before transfer begins
- CPU overhead: ~3K-25K instructions
- seek time: avg 2-12ms, range 0ms-30ms (random seek avgs only 1/3 of disk surface; typical seek much shorter because of contiguous data)
- rotational latency: avg 2-8.33ms

- Most software that deals with disks and other I/O devices attempt to process information in large blocks (usually sequentially)

Device Interconnection:

Design from the 60s:

- nothing in system very smart; everything in system very expensive
- CPU had channel program, channel program had I/O operation that talked to storage controller, etc. (see immediately below)

  |CPU| => |Channel| => |Storage Controller| => |String Controller| => |Device|
- storage controller is relatively smart
- string controller is relatively stupid, mostly does analog-to-digital signal processing
- logic is expensive and CPU can't talk to lots of devices, so want to share as many levels as possible
- each device is both big and expensive
- (*Problem*) One data path: lots of inefficiency; i.e. if string controller is busy, device must wait
- (*Solution?*) Create multiple data paths: make more channels that can talk to multiple storage controllers, and more storage controllers that can talk to multiple string controllers, etc.

Storage (NAS and SAN):

- NAS (Network Attached Storage): storage attached to local area network (i.e. ethernet); provides "file" interface; low-to-midrange product

- SAN (Storage Area Network): possibly separate network containing storage; "block level" interface; mid-to-highrange end product
- Storage Networking Industry Association (SNIA): works on standards for NAS and SAN so they can interoperate and attach to multiple types of systems
- storage service providers: storage provided by third party; often connects over internet or via dedicated cable to provider; this is expensive

Diagrams and pictures: most diagrams are posted online
[diagram of DVD burner]
- multiple layers
- DVD-R and DVD-RW have slight differences between layers

[diagrams for 2-layer disks]
- laser beam focuses on different depth
- dips in DVD much smaller and denser than in CD
- recordable 2-layer disk is tricky, since focus needs to be very precise on one layer
- different focus for CD, DVD, and DVR/BlueRay
- not all DVD recorders are the same: can be benchmarked by the time it takes to burn an image, back-up, etc.

[diagrams for magneto-optic recording]
- recording: heat up magnetic field to switch bits
- reading: laser polarizes light and causes Farraday rotation of light; this can be detected by a polarized filter
- people used to believe that MO (magneto-optic) products would become popular, but then HDD devices developed to have greater density

[diagram of mainframe system]
- CPU connected to channels; channels talk to storage controllers which talk to string controllers/switches
- mainframe different from Unix/Windows servers
- software reliable and powerful; been around for a while and used throughout the world
- banks and airlines all run mainframe systems for reliability and redundancy; lots of back-ups

[diagrams of network connections between storage devices]
- SANs are despited as "clouds" with invisible interconnections between servers and disks
- cloud computing: means everything is very fuzzy and indistinct
- SCSI over IP: instead of talking to device, server talks to network; storage at far-end of network, with packets sent between the storage device and server

[diagrams/pictures of storage devices and mainframes]
[diagrams/pictures of printers]
[diagrams/pictures of tape drives]

File Structure, I/O Operation:

- from OS point of view, the file consists of a bunch of blocks stored on device
- from programmer point of view, the programmer may see a different interface; the file system doesn't care, since it just moves bytes around
- file properties (depending on OS): name(s), file protection, time of creation, time of last use, time of last modification, length count, number of pointers to it, owner, etc.

Modern file and I/O systems must address four general problems:

(1) Disk management:
- efficient use of disk space
- fast access to files: file structures and device optimization
- user has hardware independent view of disk (OS generally does too)

(2) Naming: how do users refer to files?
- concerns directories, links, etc.

(3) Protection: all users are not equal
- want to protect users from each other
- want to have files from various users on the same disk
- want to permit controlled sharing

(4) Reliability
- information must last safely for long periods of time

Disk Management:
- How should the blocks of hte file be placed on the disk?
- How should the map to find and access the blocks look?
- File descriptor: data structure that gives file attributes and contains the map which indicates where the blocks of a file are
- every file is associated with a file descriptor; these are stored on disk along with the files (when the files are not open)

Some system, user, and file characteristics:
- most files are small: in Unix, most files are very small; lots of files with only a few commands in them
- however, some files are huge: the majority of the space on a disk consists of large files
- many of the I/O operations are made to large files
- most I/Os are reads
- most I/Os are sequential
- *Conclusion*: per-file cost must be low, but large files must have good performance

File Block Layout and Access:
- Three ways to allocation files: contiguous, linked, or indexed/tree-structured
- Note: this is standard data structures stuff, but on the disk

Contiguous Allocation:
- allocate file in a contiguous set of blocks or tracks
- keep a free list of unused parts of the disk
- when creating a file, make the user specify its length and allocate all the space at once
- file descriptor contains location and size
- Advantages:
    - easy access for both sequential and random accesses
    - low overhead
    - simple
    - few seeks
    - very good performance for sequential access
- Disadvantages:
    - horrible fragmentation will make large files impossible
    - hard to predict needs at file creation time
    - may over-allocate

- can improve the scheme by permitting files to be allocated in extents: i.e. ask for contiguous blocks, but if it's not enough then get another contiguous block; extra space on last extent can be released after file is written
- IBM OS/360 permits up to 16 extents

Linked files:

- link the blocks of a file together as a linked list
- file descriptor contains first pointer to first block; each block keeps pointer to next block
- Advantages:
    - files can be extended
    - no external fragmentation problem
    - sequential access is easy: just chase links
- Disadvantages:
    - random access requires sequential access through list
    - lots of seeking even in sequential access
    - some overhead in blocks for link
- i.e. TOPS-10 and Alto (sort of)

(Simple) Indexed files:

- simplest approach: keep an array of block pointers for each file
- file maximum length must be declared at creation
- allocate an array to hold pointers to all the blocks, but don't allocate the blocks
- fill in pointers dynamically in a free list
- Advantages:
    - not as much space wasted by overpredicting
    - both sequential and random accesses are easy
    - only waste space in index
- Disadvantages:
    - lots of seeks
    - index array may be large and may require large file descriptor

Multi-level indexed files (Unix solution, version 4.4):

- in general, this refers to any sort of multi-level structure; the following describes Berkeley 4.3BSD Unix
- file descriptor contains 15 block points: first 12 pointers point to data blocks; next 3 to indirect, doubly-indirect, then triply-indirect blocks (256 pointers in each indirect block)
- maximum file length is fixed, but large
- descriptor space isn't allocated until needed
- Advantages:
    - simple; easy to implement
    - incremental expansion
    - easy access to smale files
    - good random access to to blocks
    - easy to read and write blocks in middle of file
    - easy to append to file
    - good for small files; just add indirect blocks for large files
- Disadvantages:
    - indirect mechanism doesn't provide very efficient access to large files: 3 descriptor operations for each real operation (when "opening" a file, can keep the first level or two of the file descriptor around, so it doesn't need to read each time)
    - file generally isn't allocated contiguously; have to seek between blocks

Block allocation:

- if all blocks are the same size, and use a bit-map solution:
  - one bit per disk block
  - cache parts of bit map in memory; select block at random (or not) from bitmap
- if blocks are variable size, can use free list:
  - requires free storage area management; fragmentation and compaction
  - in Unix, blocks are grouped for efficiency: each block on the free list contains pointers to many free blocks, plus a pointer to the next list block, so not many references are involved in allocation or deallocation
  - block-by-block organization of free list means that file data gets spread around on the disk
- more efficient solution for variable-size block allocation:
  - used in DEMOS system built at Los Alamos
  - allocate groups of sequential blocks; use multi-level index scheme, but each pointer is to a sequence of blocks
  - allocation tries to pick the next block, or at least another block on the same track or cylinder; this minimizes rotational seek time
  - if a pattern of sequential writing is detected, then a bunch of blocks are grabbed at a time (then released if unused)
  - part of the disk is always kept unallocated (Unix does this now), so there's a high probability a sequential block can be found to allocate

I/O Optimization:

Block size optimization:

- Small blocks:
  - small I/O buffers (used for reads and writes)
  - quickly transferred
  - require more transfers for a fixed amount of data
  - high overhead on disk; wasted bytes for every disk block (inter-record gaps, header bytes, etc)
  - more entries in the file descriptor to point to blocks
  - less internal fragmentation
  - if random allocation, more seeks
- optimal block sizes tend to range from 2K to 8K bytes; this size is increasing with improvements to technology