The UNIVERSITY of VERMONT

Benjamin Fenster
CS-201 (Arslan)
12 December 2005

Notes – File System

I. File System Interface
   a. Indexed Files
      i. Have <last name, logical record #> in one file
      ii. Then in a relative file store related data (referenced by the logical record #)
   b. Directories
      i. Contain links to physical files
      ii. Device Directory: Store length of data, date of last access / update
   c. Operations
      i. Create File
      ii. List files in directory
      iii. et cetera
   d. Important Features
      i. Efficiency: Locate files quickly
      ii. Naming: Different users can use the same names for their files
      iii. Grouping: Want a logical grouping of files

II. Types
   a. Single-Level Directory
      i. Have just a directory with links to files
      ii. Efficient for sure
      iii. Naming: Can invent some solutions but this becomes problematic. See 2-level directory
      iv. Grouping: Doesn't really exist
      v. In any case it's not very *convenient*
   b. Two-Level Directory
      i. First level is users; then each user has a directory (second level) that maps to files
      ii. This resolves the naming problem.
      iii. Grouping is still problematic.
   c. Tree-Structure Directories
      i. No limit on levels
      ii. Resolves the grouping problem.
      iii. Naming definitely isn't an issue.a file.
      iv. To find a file, need a *path* along the tree (at last that word makes sense!)
   d. Graph Structure
      i. Allows users to share the same physical files
      ii. Brings up many new issues (e.g. dangling pointers)

III. File System Mounting
   a. Must be mounted before it can be accessed
   b. Have a directory tree (subtree); attach it.
   c. If you pick a mounting point that already has a directory tree, it will become inaccessible as long as the new tree is mounted

IV. Protection
   a. Access Types: Read, write, execute, append, delete, list
   b. Access Groups: owner, group, public
   c. Have access bit for rwx for ugo
   d. Alternative: Store list of users that can access a certain file (called the Access List)

V. File System Implementation
   a. File tasks (open, write, …) are all system calls. That means the OS needs to know how to do them.
   b. File Control Block
      i. File Permissions
      ii. Dates (creation, last access)
      iii. Owner

iv. Size
v. File data blocks

c. Implementation
   i. Could represent as a linear list of iles with pointers to data blocks
   ii. Could add a hash table to be more efficient
   iii. Could also use a tree or acyclic graph as discussed in the last class.

d. Contiguous File Allocation
   i. Easy to implement
   ii. Files can grow though. How can this be handled?
   iii. Extent-Based Contiguous Allocation

e. Linked Allocation
   i. More feasible
   ii. Each block of the file contains a pointer to the next block
   iii. A directory lists the file start/end block
   iv. Problem: If one block is corrupt, the whole file is lost

f. File Allocation Table
   i. Each directory entry is: file x, starting block (say, 217)
   ii. Then at entry 217 of the FAT, you'll find the number of the *next* block (or a special EOF flag)
   iii. Now links are stored separately from files. A corrupt file block ruins nothing but the file itself. A corrupt FAT itself would be a very different story

g. Indexed Allocation
   i. The entry at each index points to a block of a file.
   ii. Directory Entry = x, 19 (meaning index Block = 19)
   iii. Inside block 19 is an index table with pointers to each block of the file
   iv. Have an entire block for this table, with unused entries set at block 1.
   v. One-level indexing imposes a limit on the file size.
   vi. Two-level indexing allows much bigger files.
      1. Directory-table points to outer index block.
      2. Outer-Index entry points to an index block.
      3. Entry in the index block points to data blocks.
      4. And a partridge in a pear tree.
   vii. UNIX allows pointers directly to blocks *and* pointers to the index table (sing-level index) *and* double *and* triple indirect. It uses whatever's needed based on the file size.
   viii. Remember: More indirect = More slow.

h. Free-Space Management
   i. Have a bit vector. If bit x is 1, block x is allocated. If zero, block x is free.
   ii. OR: Maintain a list of free blocks.
      1. Have a pointer to the first free block.
      2. Each free block points to the next free block
      3. Obviously you can't have any broken pointers.

i. Caching
   i. Disk controller may have its own cache (store one complete track)
   ii. Main memory may be allocated for caching frequently used / accessed disk data
   iii. For cache management, see the book

VI. I/O Systems

a. Steps
   i. Device driver initiates I/O
   ii. I/O driver starts I/O
   iii. I/O completes; generates an interrupt
   iv. Interrupt handler processes data and returns
   v. Then, finally, the CPU resumes the process

b. System Structure
   i. Have several I/O devices
   ii. Hardware has device controller for each device

iii. In the system have a device driver
iv. So there are three key layers for each device: the device itself, its controller, and its driver
v. This is all on top of the kernel I/O subsystem

c. Blocking/Non-Blocking? Blocking I/O means the process is suspended until the I/O completes.

VII. Performance

a. It's a good idea to look at scheduled requests and arrange them to minimize the time spent moving the disk head (a huge component of the total access time)
b. Buffering
   i. Move data from a buffer on a slow device to a faster device
   ii. Double Buffering: The fast device also keeps a buffer of its own (data goes from buffer to buffer)
   iii. Until the transfer is complete, don't write anything
c. Caching: Been there, done that
d. Spooling
e. Et Cetera
f. Other Considerations
   i. There are many context switches in a typical I/O cycle because of interrupts
   ii. If using a common system bus, all other I/O on that bus also must wait.
   iii. So reduce interrupts (using large transfers), don't copy unnecessary data, and use DMA.
   iv. To maximize throughput, memory/devices/et cetera need to be used in a balanced manner.

VIII. Mass Storage Systems

a. Disk Scheduling
   i. Disk access requires a track number and sector number within that track
   ii. A cylinder is the same track across all cylinders
   iii. So we'll examine cylinder numbers that are requested (since we can access the same track on a different platter easily)
   iv. Example:
      1. Requested: 98, 183, 37, 122, 14, 124, 65, 67
      2. Currently At: 53
      3. Ordered: 0, 14, 37, 53, 65, 67, 98, 122, 124, 183, 199
b. Algorithms
   i. First Come-First Served
      1. No thought to scheduling
      2. Very long total distance for a random of cylinders
   ii. Shortest Seek Time First (SSTF)
      1. 53, 65, 7, 37, 14, 98, 122, 124, 183, 199
      2. Better, but still based on local data
   iii. Elevator
      1. Choose a direction. Service all requests in that direction, then go the other way.
      2. Allows for new requests on the way.
      3. The idea: Changing directions too often is bad
   iv. Variation: Circular Scan
      1. Service requests on the way up
      2. Then reset to the lowest cylinder required and service requests going in the same direction as before.
      3. This is more fair to processes that arrive sooner.
      4. Could move all the way to either end and hope more requests arrive on the way, or could just move to the most extreme requests that have already been made.
c. Formatting
   i. There's physical and logical formatting.

ii. In some virtual memory implementations, swap space is treated as separate partitions.

d. RAID
    i. To use secondary storage as permanent storage, need a mechanism to guarantee reliability
    ii. Use several disks in cooperation
    iii. Mirror Operations
        1. Everything you do to one disk, do to the others
        2. So every disk should end up with exactly the same data.
    iv. Striping
        1. Store data across several disks in a stripe
        2. Could be done at the bit level or block level
        3. Faster: Read one bit from each disk simultaneously
    v. Error Correcting
        1. Additionally, store some error correcting data
        2. Then you can even recover from errors!
e. Storage Area Network: Connect disks over the network instead of I/O port
f. Tertiary Storage
    i. Removable storage: floppy, tape, optical
    ii. WORM: CD-ROM, DVD-ROM
    iii. Tapes: Sequential. Usually “open” a whole tape, not just one file on the tape. The application knows the logical format of the data (not the OS)
g. Speed
    i. Two components
    ii. Bandwidth: Transfer rate in bytes per second
    iii. Excess Latency: Time to locate data
h. Cost: Price per MB tends downward (though sometimes increases again from one year to the next)