CIS520 – Operating Systems
Handout 12
Introduction to File Systems

- File systems. Important topic - most crucial data stored in file systems, and file system performance is crucial component of overall system performance. In practice, is maybe the most important.
- What are files? Data that is readily available, but stored on non-volatile media. Standard place to store files: on a hard disk or floppy disk. Also, data may be a network away.
- Most systems let you organize files into a tree structure, so have directories and files.
- What is stored in files? Latex source, Nachos source, FrameMaker source, C++ object files, executables, Perl scripts, shell files, databases, PostScript files, etc.
- Meaning of a file depends on the tools that manipulate it. Meaning of a Latex file is different for the Latex executable than for a standard text editor. Executable file format has meaning to OS. Object file format has meaning to linker.
- Some systems support a lot of different file types explicitly. Macintosh, IBM mainframes do this. Knowledge of file types built into OS, and OS handles different kinds of files differently.
- In Unix, meaning of a file is simply a sequence of bytes. How do Unix tools tell file types apart? By looking at contents! For example, how does Unix tell executables apart from shell scripts apart from Perl files when it executes it?
  - Shell Scripts - start with a `#`.
  - Perl Scripts - start with a `#!/usr/bin/perl`. In general, if file starts with `#!tool`, Unix shell interprets file using `tool`.
  - How about executables? Start with Unix executable magic number. Recall Nachos object file format.
  - What about PostScript files? Start with something like `%!PS-Adobe-2.0`, which printing utilities recognize.

  Single exception: directories and symbolic links are explicitly tagged in Unix.
- What about Macintosh? All files have a type (pict, text) and the name of program that created the file. When double click on the file, it automatically starts the program that created file and loads the file. Have to have utilities that twiddle the file metadata (types and program names).
- What about DOS? Have an ad-hoc file typing mechanism built into file naming conventions. So, .com and .exe identify two different kinds of executables. .bat identifies a text batch file. These are enforced by OS (because it is involved with launching executables). Other file extensions are recognized by other programs but not by OS.
- File attributes:
  - Name
  - Type - in Unix, implicit.
  - Location - where file is stored on disk

  - Size
  - Protection
  - Time, date and user identification.

- All file system information is stored in nonvolatile storage in a way that it can be reconstructed on a system crash. Very important for data security.

- How do programs access files? Several general ways:

  - Sequential - open it, then read or write from beginning to end.
  - Direct - specify the starting address of the data.
  - Indexed - index file by identifier (name, for example), then retrieve record associated by name.

  Files may be accessed more than one way. A payroll file, for example, may be accessed sequentially by paycheck program and indexed by personnel office. Nachos executable files are accessed directly.

- File structure can be optimized for a given access mode.

  - For sequential access, can have file just laid out sequentially on disk. What is problem?
  - For direct access, can have a disk block table telling where each disk block is. To access indexed data, first traverse disk block table to find right disk block, then go to the block containing data.
  - For more sophisticated indexed access, may build an index file. Example: IBM ISAM (Indexed Sequential Access Mode). User selects a key, and system builds a two-level index for the key. Uses binary search at each level of index, then linear search within final block. Notice how memory hierarchy considerations drive file implementation.

- Easy to simulate a sequential access file given a direct access file - just keep track of current file position. But simulating direct access file with a sequential access file is a lot harder.

- Fundamental design choice: lots of file formats or few file formats? Unix: few (one) file format. VMS: few (three). IBM lots (I don't know just how many).

- Advantage of lots of file formats: user probably has one that fits the bill.

- Disadvantage: OS becomes larger. System becomes harder to use (must choose file format, if get it wrong it is a big problem).

- Directory structure. To organize files, many systems provide a hierarchical file system arrangement. Can have files, and then directories of files. Common arrangement: tree of files. Naming can be absolute, relative, or both.

- There is sometimes a need to share files between different parts of the tree. So, structure becomes a graph. Can get to same file in multiple ways. Unix supports two kinds of links:

  - Symbolic Links: directory entry is name of another file. If that file is moved, symbolic link still points to (non-existent) file. If another file is copied into that spot, symbolic link all of a sudden points to it.
  - Hard Links: sticks with the file. If file is moved, hard link still points to file. To get rid of file, must delete it from all places that have hard links to it.

  Link command (ln) sets these links up.

- Uses for soft links? Can have two people share files. Can also set up source directories, then link compilation directories to source directories. Typically useful file system structuring tool.

- Graph structure introduces complications. First, must be sure not to delete hard linked files until all pointers to them are gone. Standard solution: reference counts. Second, only want to traverse files once even if have multiple references to same file. Standard solution: marking. cp does not handle this well for soft links; tar handles it well.

- What about cyclic graph structures? Problem is that cycles may make reference counts not work - can have a section of graph that is disconnected from rest, but all entries have positive reference counts. Only solution: garbage collect. Not done very often because it takes so long.
- Unix prevents users from making hard links create cycles by only allowing hard links to point to files, not directories. But, with .. still have some cycles in structure.
- Memory-mapped files. Standard view of system: have data stored in address space of a process, but data goes away when process dies. If want to preserve data, must write it to disk, then read it back in again when need it.
- Writing IO routines to dump data to disk and back again is a real hassle. What is worse, if programs share data using files, must maintain consistency between file and data read in via some other mechanism.
- Solution: memory-mapped files. Can map part of file into process's address space and read and write the file like a normal piece of memory. Sort of like memory-mapped IO, generalized to user level. So, processes can share persistent data directly with no hassles. Programs can dump data structures to disk without having to write routines to linearize, output and read in data structures.
- Used for stuff like snapshot files in interactive systems.
- In Unix, the system call that sets this up is the `mmap` system call. How is sharing set up for processes on the same machine? What about processes on different machines?
- Next issue: protection. Why is protection necessary? Because people want to share files, but not share all aspects of all files. Want protection on individual file and operation basis.
  - Professor wants students to read but not write assignments.
  - Professor wants to keep exam in same directory as assignments, but students should not be able to read exam.
  - Can execute but not write commands like cp, cat, etc.

  For convenience, want to create coarser grain concepts.
  - All people in research group should be able to read and write source files. Others should not be able to access them.
  - Everybody should be able to read files in a given directory.
- Conceptually, have operations (open, read, write, execute), resources (files) and principals (users or processes). Can describe desired protection using access matrix. Have list of principals across top and resources on the side. Each entry of matrix lists operations that the principal can perform on the resource.
- Two standard mechanisms for access control: access lists and capabilities.
  - Access lists: for each resource (like a file), give a list of principals allowed to access that resource and the access they are allowed to perform. So, each row of access matrix is an access list.
  - Capabilities: for each resource and access operation, give out capabilities that give the holder the right to perform the operation on that resource. Capabilities must be unforgeable. Each column of access matrix is a capability list.

  Instead of organizing access lists on a principal by principal basis, can organize on a group basis.
- Who controls access lists and capabilities? Done under OS control. Will talk more about security later.
- What is the Unix security model? Have three operations - read, write and execute. Each file has an owner and a group. Protections are given for each operation on basis of everybody, group and owner. Like everything else in Unix, is a fairly simple and primitive protection strategy.
- Unix file listing:

```
 4 drwxr-xr--   2 martin   faculty     2048 May 15 21:03 ./
 2 drwxr-xr-x   7 martin   faculty      512 May  3 17:46 ../
 2 -rw-r-----   1 martin   faculty      213 Apr 19 22:27 a0.aux
 8 -rw-r-----   1 martin   faculty     3488 Apr 19 22:27 a0.dvi
 4 -rw-r-----   1 martin   faculty     1218 Apr 19 22:27 a0.log
72 -rw-r--r--   1 martin   faculty    36617 Apr 19 22:27 a0.ps
 6 -rwxr-xr-x   1 martin   faculty     2599 Apr  5 18:07 a0.tex*
```

- How are files implemented on a standard hard-disk based system? It is up to OS to implement it. Why must OS do this? Protection.
- What does a disk look like? It is a stack of platters. Each platter may have two surfaces (one per side). There is one disk head per surface. The surfaces revolve beneath the heads, with the heads riding on a cushion of air. The heads move back and forth between the platters as a unit. The area beneath a stationary head is a track. The set of tracks that can be accessed without moving the heads is a cylinder. Each track is broken up into sectors. A sector is the unit of disk transfer.
- To read a given sector we first move the heads to that sector's cylinder (seek time), then wait for the sector to rotate under the head (latency time), then copy data off of disk into memory (transfer time).
- Typical hard disk statistics: (Sequel 5400 from August 1993, 5.25 inch 4.0Gbyte).
  - Platters: 13
  - Read/Write heads: 26
  - Tracks/Surface: 3,058
  - Track Capacity (bytes): 40,448 - 60,928
  - Bytes/Sector: 512 - 520
  - Sectors/Track: 79-119
  - Media Transfer Rate (MB/s): 3.6-5.5
  - Track-to-track Seek: 1.3 ms
  - Max Seek: 25 ms
  - Average Seek: 12 ms
  - Rotational Speed: 5,400 rpm
  - Average Latency: 5.6 ms
- How does this compare to timings for a standard workstation? DECStation 5000 is a standard workstation available in 1993. Had a 33 MHz MIPS R3000, 60 ns memory. How many instructions can execute in 30 ms (about time for average seek plus average latency)? $33 * 30 * 1000 = 990,000$. Plus, many operations require multiple disk accesses.
- What does disk look like to OS? Is just a sequence of sectors. All sectors in a track are in sequence; all tracks in a cylinder are in sequence. Adjacent cylinders are in sequence. OS may logically link several disk sectors together to increase effective disk block size.
- How does OS access disk? There is a piece of hardware on the disk called a disk controller. OS issues instructions to disk controller. Can either use IO instructions or memory-mapped IO operations.
- In effect, disk is just a big array of fixed-size chunks. Job of the OS is to implement file system abstractions on top of these chunks.