Lecture XI
File Systems

§1. Introduction to File Systems

- Computers have different kinds of memory, as typified by the memory hierarchy (see Lecture 1). Among these, the **secondary memory** is the most familiar to users. That is because secondary memory is logically organized into a file structure, and this file structure is largely determined by users: they can create, modify and delete files in this structure.

  The two characteristic of secondary memory are that it is relatively abundant, and that it is non-volatile. Non-volatility means it maintains its information even after the computer is powered off. Hence it can serves as the permanent repository of data in the computer. In contrast, the main memory (popularly known as RAM) is volatile and must be loaded with information from secondary memory when the computer starts up.

  abundant and non-volatility

  Secondary memory is popularly known as **disk memory**, because the magnetic disk drive is the dominant technology used to provide secondary memory. So we will use these two terms interchangeably.

  secondary memory = disk memory

- We briefly discuss the physical layout of disk drives. Physically, a disk memory is distributed over some geometry that depends on the particular device. Each disk is a stack of platters (like a CD) ranging from 1.8 to 5.25 inches. Each platter has two sides. Each side is called a **cylinder** (admittedly, a somewhat confusing terminology). The cylinders are covered with a magnetic substrate, so that each unit area of the cylinder can be magnetized in one of two orientations, representing a bit of information. They form the basic memory units. The cylinder surface is divided into discrete memory units along independent dimensions: it is divided into concentric circular **tracks**, and also into **sectors** by radial lines. Thus each memory bit has a physical address given by a triple (cylinder, track, sector). There is a reading arm that can slide in or out along a fixed radial line of the disk. The tip of this arm is a reading head that can simultaneously read the memory bits on all the cylinders A given position of the head corresponds to some track number. As the cylinders spins, a given head position at a given time instant will read all the memory bits of a given track of a given bit in a sector in all the cylinders. E.g., if there are 8 cylinders, we can regard the data being read in an instant as a byte. Moreover, the bytes read over a suitable time interval will correspond to the bytes in a given sector.

- We will assume the addressable units of memory are byte-size (=8 bits). Instead of bytes, one could assume other units such as **words** (= 4 bytes).

Conceptually, *we regard physical memory as an linear array of bytes, indexed from* 0 *to some maximum value*. In this linear array, the index of a byte is called its **address**.

Physical memory is a linear array

The view of disk memory as a linear array represents a slight abstraction beyond the geometry of the disk device. There is a corresponding problem of translating linear addresses to the device address space (e.g., cylinders, sectors, tracks) that can be handled by device drivers.

- The physical memory is organized into logical units called **files** where each file is logically an array of bytes. However, the linear array in a file need not correspond to the physical memory array. Therein lies the key issue: *The main problem of secondary memory management is to provide an efficient mapping from the logical view of files to the physical view of memory.*

Main problem of secondary memory

§2. Logical File Organization

- In most operating systems, our logical view of files is further refined into a hierarchical structure. This will be our assumption.

  More precisely, the hierarchy is a labeled tree with a unique root, where each node in the tree has zero or more children. Nodes that have one or more children are called **directories**.

- To implement this hierarchical structure within our file system, we treat directories as special files! Hence we have two types of files: **regular files** or **directory files**.

- **File attributes**

  1. File name: human readable string
  2. Identifier: a number that is the file system
  3. Type: ascii, binary, graphics, etc
  4. Location: physical address
  5. Size
  6. Protection:
  7. Time, Date, Ownership: creation, last modification, last use, etc.

  These file attributes are useful in implementing functions for protection, security, monitoring and search.

  Files can store data of various types: E.g., source, binary, graphics, sound, etc.

  But for the present purposes, these distinctions are irrelevant.

  Remark: file name typically has 2 parts (second part is the extension, indicating type for some systems)

- There is a file directory, also in secondary storage, for this info.
- File Operations:
  1. Create: find space, create directory entry
  2. Write: depends on current location
  3. Read: depends on current location
  4. Seek: change the current location
  5. Delete:
  6. Truncate:
- Open and Close:

  Since read/write/seek, etc, has nontrivial initialization cost, most OS requires that we must first OPEN a file before we can do these operations. Hence we also need to CLOSE a file when done.

  There is a list of OPENED files (per process, and system-wide).

  The per-process list of OPENED files points to the system-wide list.

  We need these info for the per-process opened files:
  1. File pointer: current location
  2. File-open count: closes file when it reaches 0
  3. Memory location: location in main memory about file
  4. Buffer: this is sometimes used to speed up I/O.
  5. Access rights:

§3. File Allocation Methods

See section 11.4, p.421, Silbers. (Edn.7)

- We begin with the problem of managing physical memory. Recall that this is conceptually a linear sequence of bytes.
- Disk memory is partitioned into two parts: "free memory pool" and "allocated memory". The latter is presumably used by the file system.

  Each byte of memory are either **free** or **allocated**, depending on which partition they belong. There are two basic functions which move bytes from one partition to the other:
  - Alloc(n) will move n contiguous bytes from the free memory pool to allocated memory. The address of the first of the n bytes is returned.
  - Free(ADDR,n) will move n contiguous bytes, starting at address ADDR, from allocated memory to physical memory pool.

In normal usage, there is an asymmetry in these 2 requests: the Alloc(n) request could fail, and return a NULL pointer if there is no contiguous n bytes in the free memory pool. On the other hand, Free(ADDR,n) will normally succeed. We say "normally" because we can normally assume that ADDR is a valid address for a block of n bytes. Otherwise, Free could also fail.

- The **allocation problem** to implement the Alloc/Free functions.

  We have seen this problem before, in memory allocation. There is an obvious solution to this problem, but it leads to serious problems of fragmentation. The way out of this dilemma was to allocate only fixed size units called pages or frames.

  We use the analogous solution here: we divide the physical memory into fixed size units called **blocks**. Thus blocks is the analogue of pages and/or frames.

  We re-interpret "Alloc(n)" to mean the request for n blocks, or "Free(ADDR,n)" as request to free n blocks. Now, every block is either **free** or **allocated**.

- Most often, block size ranges from 512B to 4KB. But how should we choose block sizes?

  One factor is to correlated with disk sector sizes. Typical hardware disk sectors ("disk blocks") are typically 512 Bytes. So block sizes tend to be small multiples of this. Alternatively, we want to correlate it with page sizes (which in turn is correlated with disk sector sizes).

  For any choice of a block size B, note that (1) for files with size s maller than B, the mandatory waste is $B - s$, and (2) for files with size s larger than B, the average waste is $B/2$. Therefore, to help us choose a file size by looking at typical file systems, we ought to look at the median value for s, not the average s. One study [Mullender and Tannenbaum (1984), in Tanenbaum p.411] indicated that the median file size in UNIX is about 1 KB.

- There is a trade-off between reducing wasted memory (by choosing small block sizes) and increasing disk transfer speed (by choosing larger block sizes).

  Let us give a simple analysis of disk performance as a function of block size B. First, we characterize the disk drive in terms of the following constants: average seek time σ, rotation time ρ, and the number κ of bytes per track.

  To explain κ, note that each track number corresponds to certain radius. With larger radius, we can pack more bits into the track. However, it is convenient to divide the tracks into groups so that, within a single group, their radii do not differ by much, and we may let each track of the group have the same κ.

Let us assume that a disk block of size B resides entirely in a single track. Then the transfer rate τ is given by the formula:

$$\tau = \sigma + (\rho/2) + (B\rho/\kappa).$$

Tanenbaum (p.411) gives these illustrative numbers: $\sigma = 10$ msec, $\rho = 8.33$ msec, $\kappa = 130K$. Then with $B = 1K$, τ is somewhat less than 100 KB/sec. With $B = 4K$, $\tau = ...$

REMARK: figure 6-20 (the solid curve) in Tanenbaum is suspicious!

- Sometimes, there is an additional **fragment size**, which is between 512 Bytes and block size. A file would use the regular block sizes for all but the last one.

  Yet another variation is to allow any power of 2 of disk block sizes. This means we can have the sizes $512B, 1KB, 2KB, 4KB$, etc.

- We next give some standard solutions to the Free/Alloc problem.

  SOLUTION ONE: FreeList.

  The simplest solution to the Free/Alloc problem is this: keep all the free blocks is a singly linked list called the **FreeList**. See figure.

  FIGURE: free-list

  Alloc and Free are easily solved: for instance, we can assume that Alloc() returns a linked list of n blocks. Free() simply appends the freed blocks (assumed to be a linked list) to the FreeList.

- Issues: to get n blocks, we need to read n blocks from disk. This is inefficient, since it is often sufficient to return the addresses of n blocks.

- SOLUTION TWO: Organize the free blocks into a B-tree. Thus, all the internal nodes of the B-tree are "directory blocks", meaning that they hold a number of addresses of blocks.

  This is essentially the approach taken by the NTFS (New Technology File System) of Microsoft, replacing the FAT system of previous Window systems. Technically, they use a B+ tree, meaning that the leaves are linearly linked. But the implementation details of NTFS are trade secrets.

- SOLUTION THREE: Suppose a block can store A addresses. So we can use a free block as an **index block**, meaning that it will store some number $a \in \{0, 1, \ldots, A\}$ of addresses of other free blocks.

  E.g., block size is 4KB and block addresses are 4 bytes, then $A = 1024$. In the B-Tree solution, a is either 0 or some number between $A/2$ and A.

  In Unix, we use the last address to point to another index block (or to NULL).

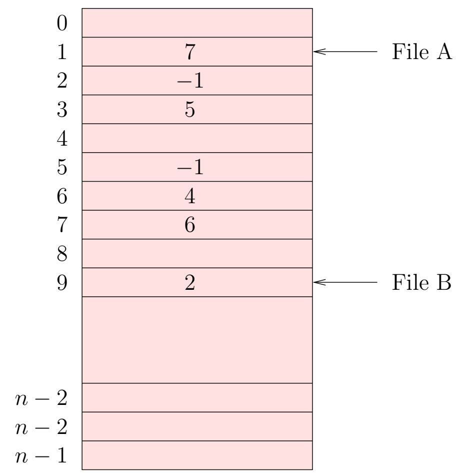

Figure 1: FAT method

- SOLUTION FOUR: The FAT (File Allocation Table) solution. This table has as many entries as there are blocks in the system. The ith entry is either 0 (meaning it is in-use) or it points to another free block (or NULL).

  The FAT solution was used by early Microsoft OS. It gives much faster access to files, but it introduces the problem of large file tables. For instance, a 120 GB disk with 4KB blocks will have 30M entries.

  DETAILS: There are several variants of FAT (FAT12, FAT16, FAT32). The physical disc is divided into 4 areas: boot record, file allocation tables (FAT), root directory, data area. The boot record tells us the locations of the other 3 areas.

  The FAT is the main focus of must technical description of this file system. Incidentally, "blocks" are called "cluster" in FAT terminology. A given file is viewed as a "cluster chain". If we think of files as a linked list of blocks (a.k.a. clusters), then all the "linking information" are collected in the FAT.

  The root directory is fixed in length in FAT16, but in FAT32, the root directory is just another cluster chain.

  The data area is divided into blocks (a.k.a. clusters) and occupies most of the disk. The files are either classified as "regular files" or "directories", so the hierarchical structure is distributed in the entire system. Each directory entry in FAT32 is 32 bytes long, so a 512 byte sector can hold 16 directory entries. Each entry is one of four types: (1) file pointers, (2) subdirectory pointers (3) volume labels (4) long file name components.

  There is some issues with long file names, since the original 8.3 length had to be extended.

§4. Unix File Systems

We will go into the details of the Unix file system.

- As noted above, there are two views of files: the physical and the logical (or user) view. Physically, the disk is a sequence of blocks.
- Consider the physical blocks.
  1. Block 0 is the **boot block** (or boot control or boot partition block). If we want to boot an OS from this partition, this block is used. Otherwise it can be empty.
  2. Block 1 called the **superblock**. Contains number of blocks, size of blocks, free-block count, free-block pointers, i-node count, i-node pointers, etc. Note: i-node is also called File Control Block (FCB).
  3. Block 2 to some Max are the blocks for i-nodes (see below).
  4. The rest are data-blocks.
- Logically, a file is a sequence of blocks. There are two main kinds of files: **regular files** and **directory files**. There is a third, **non-disk files**. See below.
- There is an **i-node** (short for **index node** for each file. It contains the following information about a (regular) file:
  1. user ID and group ID of owner of file,
  2. time of last modification and access,
  3. number of hard links to file,
  4. mode (read/write/execute permissions),
  5. type of file (regular, directory, symbolic link, char/block/socket dev),
  6. 12 pointers to data blocks on disk. Thus, access to the first 48KB ($= 12 \times 4$ K) of data is very fast.
  7. 3 additional indirect block pointers: single indirect, double indirect, and triple indirect.

     Since each block holds 4K bytes, and each pointer is 4 bytes, the first single indirect can access 4MB of data. The second can access 4GB. The triple indirect is not used, since current systems address space of 32-bit cannot go beyond 4GB.

  Normal users refer to files by their names and path. The System uses i-nodes to refer to files.

  Q: As user, how do you get the i-node number of a file?

  A: Type "ls -i file-name"

- A directory file is just a sequence of variable length triples: (length, inode-ptr, file-name).

  Each triple corresponds to a file in the directory. The inode-ptr refers to the i-node of the file, and file-name can be up to 255 chars long. The length refers to this file-name length.

  The first two entries in the file refers to "." and "..".

- Given a path, there is the obvious sequential algorithm to search directories (starting from / or from current directory).

  To avoid infinite loops, we cound the number of symbolic liks encountered and stop when a limit (8) is reached. REMARK: can't a hard link cause infinite loop too? If so, hard links should be counted as well.

- LINKS:

  Hard link are directory entries, like ordinary entries. So each hard link has a corresponding i-node.

  Symbolic links: such links do not creat a new i-node, only an entry in some directory.

  Hard links has no effect on the search algorithm, but symbolic links affect the search algorithm as follows. When our search reaches a symbolic link, we start the search all over, starting from the head of a path name that is associated with the symbolic link!

  Q: How to you verify that on your unix that hard links creates a new i-node but symbolic links does not?

  A: Create a symbolic and hard link to some file "foo" and look at their inodes:
  @ ln -s foo symfoo
  @ ln foo hardfoo
  @ ls -i foo symfoo hardfoo

- HOW TO HANDLE NON-DISK FILES.

  There are 3 main types: block device, char device, or sockets. We call appropriate drivers to handle them.

- Opening and Closing files.

  After we found the i-node of the file, we allocate a **open-file structure** for this i-node. An index into the **open-file (structure) table** is what we call a **file descriptor**. This open-file table is PER PROCESS.

- A directory name cache can be used to hold recent directory-to-inode translations, to speed up file access.

  There is a in-core list of i-nodes. All currently open files have a copy of their i-node here.

- It turns out, we want an intermediate structure between open-file structure and the i-node. The *current position* of an open file is one information that does not belong to either i-node or the per-process open-file structure. Hence we store this information in an entry of the **system-wide open-file (structure) table**. This entry, in turn, points to the i-node. It can also keep track of the number of processes that has opened this file. (See Fig A.7 of Appendix A)

  Let us illustrate a situation where we need this intermediate structure. Suppose a process P0 opens a file F1, and then fork two processes: P1 to write some head info into F1, followed by P2 to write additional info into F2. Note that P0's open-file structure for F1 will be inherited by P1 and P2 through the fork.

  Now, after P1 has done its work, the updated location should be accessible to P2. But this information, if stored in P1's open-file table, would not be accessible to P2. But if both of them points to the "system-wide open-file structure", then P2 can continue where P1 left off!

- **Clustering Schemes.** It is best to allocate an i-node and its data blocks from the smake locality in a disk. FreeBSD has a concept called **cylinder group** (cylinder refers to a locality on the disk). Each cylinder block has a superblock, array of inodes and data blocks, just as in a partition. All cylinder blocks have the same superblock. Block allocation can use such information to allocate free blocks to preserve locality (and other considerations). See p.908 of Appendix A.

- **Free Space Management.** There are three basic methods:

  (1) The simplest method is to keep a linked list of all the free blocks.

  (2) We could also use bitvectors to track the free blocks.

  (3) Finally, we could let each free block store a sequence of pairs (ADDR, NUM) where ADDR is the address of NUM free contiguous blocks.

  FIGURE illustrating (3).

  Let us discuss scheme (3): Which of these free blocks should recursively be used to store similar information? If ALL the free blocks are used in this way, then we cannot easily give away the current free block to requests for blocks. An intermediate solution is to just use the last two free blocks for recursively storing such free-block information.

  Advantage: we have a branching factor of two, but all but two of the free blocks in the current direct can be directly given away.

  If a free block is released (by file delete), this free block can often be inserted directly at the root. This is most efficient.

- Solution to Thrashing in (3): keep the first two blocks in memory, writing out a full block only after we have 1.5 blocks of memory!

- **File Consistency Problem.** Suppose the computer crashes. The open-file table is generally lost. We need to check the consistency of the file system. The loss of an i-node, a directory block, or a free-list block is most serious. To deal with this, most OS has a utility program to check consistency of a file system. In Unix, it is called "fsck" and Windows it is called "scandisk". We normally run this utility on system reboot, especially after a crash. In Unix, we need to check consistency of two things: block structure and file structure.

  Let us first consider block consistency: we will consider the problem under the assumption that free blocks are kept in a free-list. Basically, every block must exactly once, either in the free list or referenced by some i-node. Suppose b is the block number and $free[b]$ and $inode[b]$ indicates how many times b appears in either list. Then we have the following states:

  - $free[b] + inode[b] = 1$: b is consistent
  - $free[b] + inode[b] = 0$: b is missing
  - $free[b] > 1$: b is duplicated in free list
  - $inode[b] > 1$: b is duplicated in file system list

  So our algorithm begins by initializing $free[b] = inode[b] = 0$ for all b. Then we go through the free list, and increment $free[b]$ for each b that we find in the free list. We also go through the file hierararchy, and for each i-node, we look for all the blocks pointed to via this i-node.

  PRESUMABLY, there is enough main memory for storing the $free[b]$ and $inode[b]$ arrays.

  In case of inconsistency, we need to take some action: if a block is missing, we add it to the free list. If it is duplicated in the inode list, we duplicate the data in the block (by getting blocks from the free list), and modify the inodes that point to these blocks. If it is duplicated in the free list (either because $free[b] > 1$ or ($free[b] = 1$ and $inode[b] > 0$), we can fix the free list.

  Next consider checking consistency of the file system: This is basically a check that the number of hard links to an inode is correct. We simply recompute this number for each inode.

  Finally, it is possible to combine both block consistency and file consistency check into one single algorithm.

§5. Virtual File Systems

- File systems can become very general when it has to deal with
  - Raw disk (for database applications, etc)
  - Dual boots of different OS

  - Different file systems mounted together
  - Network file systems
  - etc
- This calls for a Virtual File System organization.
  - There are 3 layers.
  - Layer 1 has the basic interface of open(), read(), write(), close().
  - Concept of vnode: unique identifer and kernel object for each directory or file.

§6. Network File Systems

- Sect 11.9 of Silbers.

EXERCISES

Exercise 6.1: What are the trade-off issues in choosing block sizes? ◇

Exercise 6.2: Suppose you have a file named `f0` and you do:

```
ln f0 f1
ln -s f0 f2
ls -i f0 f1 f2
```

What do you expect to see from the last command? ◇

END EXERCISES