COMP 242 Class Notes
Section 6: File Management

1. FILE MANAGEMENT

We shall now examine how an operating system provides file management. We shall define a **file** to be a collection of permanent data with a global name. The data are permanent in the sense that they remain in existence until they are destroyed explicitly by a process or a user (through some process). The name is global in the sense that it does not belong to the address space of a single process and can be used by any process with appropriate access rights.

In many respects files are similar to variables in programs:
they have a name
the name refers to some data
the data may be examined or modified
access to the name and the data is controlled

As a result, file management, done by an operating system, is somewhat similar to symbol table management, done by a compiler. However, there are several differences in the two management schemes, which stem from the fact that files contain permanent data.

It is important to note that files are *abstract* entities provided by an operating system and not physical entities provided by the hardware. They do require hardware that supports permanent storage (disks or tapes). However, the operations defined on them, how they are named and protected, their physical representation on permanent storage are all defined by an operating system. An operating system may, of course, use hardware characteristics in deciding some of these issues (such as physical representation) for efficiency reasons.

1.1 Naming

There are several issues in naming files. We discuss some of these below.

Flat vs Hierarchical

An operating system may provide a *flat* or a *hierarchical* name space. A **flat** name space is similar to a global scope in FORTRAN. Thus, just as two variables with the same name 'i' cannot be created in FORTRAN, two files with the same name 'foo' cannot be created in a flat name space. The Exec-8 system for the Univac provides such a name space. (It alleviates some of the problems of a flat name space by dividing a name into three components, and defaults the first component to the user's id.)

A **hierarchical** name space is similar to nested scopes in Pascal. A file system calls these nested scopes **directories**, which may themselves be named. (Can programming language scopes be named?) Now each file (or directory) has a **local** name, and a **full name**, which is the full name of the directory in which it is contained followed by the local name of the file. No two files in the same directory may have the same local name. However, files in different directories may have the same local name.

A process may name a file by its full name or a name **relative** to a special directory called the **working directory** of the process. For instance in Unix, if the full name of the working directory is '/u2/joe', then the names '/u2/joe/foo' and 'foo' refer to the same file. The symbol '/' distinguishes full names from relative names. A special operation is provided to change the current working directory. In Unix the working directory has the special relative name "." and its ancestors have the names "..", "../.." and so on.

A directory is more than a naming tool. An operating system defines special operations on a directory that give the names and other **attributes** of the files contained in the directory. These attributes include information such as creation date, owner, size, and so on.

The decision to use directories as parts of hierarchical names and as collections of attributed files is a convenient one. However, the two functions of providing hierarchical names and collections of files are not inseparable. Assume that files of remote computers can be accessed. Then the name of the form "jeeves.cs.unc.edu:/foo" is hierarchical even though "jeeves" is a machine name and not a directory name. Similarly, systems that provide flat name spaces also provide a mechanism to examine the name and other attributes of the files in the system.

Devices in Name Space

Should device names be parts of the name space of files? If files and devices share the same operations, then yes. In this situation certain file names are reserved, such as (in Unix):

```
/dev/console
/dev/floppy
/dev/ttyia
```

Aliases

Operating systems such as Unix allow a file to be referred by several **aliases**. Thus the names "/paper1/references" and "/paper2/references" may refer to the same file. Aliasing allows use of shorter relative names for files that do not belong to one single directory. For instance, while in "paper2" a user may use the shorter name "references", rather than the longer name "/paper1/references" To make a new alias of a file, a special call of the form

```
alias (old name, new name)
```

is provided. It is important to note that the call does not create a new copy of the file but only provides a new name for it.

Aliasing raises several two interesting issues related to deletion and accounting:

Deletion: if the file is deleted under one alias should it disappear under all aliases? If so, a search is required to find all aliases of a file. To make this search efficient, all aliases of the file could be stored in a central place, perhaps with the contents of the file. If not, then a reference count is stored with each file, and deleting a file decrements the count and removes the file only if the count goes to zero.

Accounting: if two different users have access to the same file, who should be charged for the file space occupied by the file? We could charge the owner of the file, that is, the user who first created it. However, this situation is not fair, specially if the owner deletes it. A better alternative is to divide the space used among all directories in the system.

Indirect Files

Berkeley 4.2 Unix provides another way to have multiple (full) names for the same file through **indirect files**. An indirect file is a file that contains nothing but the name (full name or relative) of another file. A special call of the form

```
Indirect(F, I)
```

may be used to create a new indirect file for F. Any attempt to open the indirect file will open F instead.

Indirect files also raise several issues including deletion and accounting:

Deletion: Deleting an indirect file I does not delete F. Deleting F makes I useless. It is then illegal to open I, though it remains in the directory. If a file named F is created again, I is again usable. (Why is this feature useful?)

Accounting: If an indirect file I refers to F, then the owner of I pays for the minuscule amount of storage store the name F. Thus the owner of F pays for all of it.

Multiple Indirection: Can Indirect files name other indirect files? If so, then there is a danger of recursion, as shown below:

```
I1 -> I2 -> I1
```

(Does such a danger exist with aliasing?) To detect this recursion, an indirect file chain may be kept to a small number such as 5.

Interpretation of Relative Names: With respect to what directory should relative names in indirect files be understood? Two alternatives are: 1) the working directory of the process, and 2) the directory in which the indirect file is stored. (Going back to the analogy between variables and files, the alternatives essentially signify 'dynamic' binding and 'static' binding) Unix chooses the second alternative.

Naming by Users vs Processes

The naming scheme discussed here is the one used by processes when they make service calls that specify file names. A user names a file via some process such as a command interpreter, which may embellish or replace the naming scheme provided to it by the operating system. For instance, a command interpreter might allow a '*' in the file name to match any string.

A danger with allowing embellishment of the naming scheme by different processes is that a user may not see a single naming scheme. It would be useful if an operating system provides a naming scheme that does not need any embellishment for users and can be provided unaltered by all processes. However, this situation is not always possible. Some systems provide a special program that allows a user to point at the name of a file in a directory listing instead of typing it. These processes may allow two different files in a directory to have the same name, as shown below:

```
bar
foo
foo
```

There is no ambiguity since a user points at the desired file name. Such a naming scheme cannot be supported by the operating system since a service call cannot 'point' at a file entry.

1.2 Access Methods

An **access method** defines the way processes read and write files. We study some of these below.

Sequential Access

Under this access method, the entire file is read or written from the beginning to the end sequentially. Files in popular programming languages such as Pascal and Ada provide such access. The file is associated with a **read/write mark**, which is advanced on each access. If several processes are reading or writing from the same file, then the system may define one read/write mark or several. In the former case, the read/write mark is kept at a central place, while in the latter case it is kept with the process table entry. In Unix, a combination of the two schemes is provided, as we shall see later.

Direct Access

This access allows a user to position the read/write mark before reading or writing. This feature is useful for applications such as editors that need to randomly access the contents of the file.

Mapped Access

The Multics operating systems provide a novel form of access which we shall call **mapped access**. When a process opens a file, it is mapped to a segment. The open call returns the number of this segment. The process can thus access the file as part of its virtual store. The CloseSegment call may be used to close the file.

Structured Files

So far, we have treated files as byte streams. Database applications often wish to treat them as records, that may be accessed by some key. To accommodate these applications, some systems support **typed** or **structured** files that are considered streams of records. If a file is structured, the owner of the file describes the records of the file and the fields to be used as keys. OS/360 for IBM computers and DEC VMS provide such files.

Binding of Access Methods

An access method may be specified at various times:

When the operating system is designed. In this case, all files use the same method.

When the file is created. Thus, every time the file is opened, the same access method will be used.

When the file is opened. Several processes can have the same file open and access it differently.

1.3 Physical Representation

We now study the physical representation of files on disks. We first consider the hardware and then consider some general concepts.

1.3.1 *Disks.* Typically a **disk drive** consists of a **disk pack** consisting of several **platters** stacked like phonograph records. Information is recorded magnetically usually on both sides. All the platters move as one unit. Each platter is divided into concentric rings called **tracks**, and each track is divided into **sectors**. Data transferred to and from a sector is called a **block**. The number of sectors per track and number of bytes per block is determined by how the disk is **formatted**. Some disk units allow the formatting to be done under software control.

Data are transferred from a platter side through a **read/write head**. These can move in and out to read different tracks; this motion is called **seeking**. The read/write heads of different disks are usually physically tied together so they all access the same track at the same time. The set of all track positions for a given head position is called a **cylinder**.

Three types of delay, or **latency**, affect how long it takes to read or write disk data. First, the **seek latency** is the amount of time needed to seek to the desired cylinder, and depends on how far the head has to seek. **Rotational latency** is the amount of time needed for the desired sector to arrive at the read/write head. Last, the **transfer latency** is the amount of time needed for the sector to be completely scanned.

As a concrete example, consider a Seagate 1.7 Gbyte disk drive. It has 512 bytes per sector, 63 sectors per track, 3305 cylinders, 2 disk platters, and 4 R/W heads, average seek time of 12.0 msec, track to track seek of 2.0 msec, spindle speed of 4500 RPM (13 msec per rotation).

An another example, consider the Seagate 4.3 GB drive. It has 512 bytes per sector, 63 sectors per track, 8894 cylinders, 4 disks, and 8 R/W heads, average seek of 12.0 msec, track to track seek of 2 msec, and spindle speed of 4500 RPM.

Thus, the difference seems to be in number of disks and cylinders.

One or more disk drives are connected to a **disk controller**, which positions the disk arms, controls transfer of data, and senses any errors that occur. A disk controller talks to the cpu through a **host interface**, which is connected to the system bus. The host interface passes requests from the cpu to the controller to start an I/O operation, and interrupts the CPU when the controller signals completion. Like the SLU unit, it is associated with device registers. The main difference between the two is that the disk hardware can transfer large amounts of data to and from memory, interrupting only when the transfer is complete.

1.3.2 *Some General Concepts. File Sizes*

An important property of files, used in the physical representation of files, is that most files are small while some are very large. A study done at Los Almos shows that half the files were within 40Kbytes, whereas many reached 10M bytes or so. Another study at UW-Madison showed that 42 percent of all files fit within one disk block, and 73 percent fit within 5 disk blocks.

File System Blocks and Fragments

We shall distinguish between **file system blocks** and disk blocks discussed earlier. A disk block is the data in one sector while a file block is data in one or more consecutive sectors. Thus a file system block size is a multiple of a disk block size.

Making a file system block different from a disk block gives the operating system some control on the unit of data transfer. A large (file system) block size allows greater utilization of the disk bandwidth, while a small block size allows lesser internal fragmentation. In view of the fact that there are both some very large files that may be read sequentially, and some very small files of 5 or more block sizes, it is important to optimize both.

To achieve the above purpose, some systems support 'large' and small 'blocks'. In particular, 4.2BSD divides a file system block into 2, 4, or 8 **fragments**, each of which is addressable. Each fragment in turn is made up of one or more disk blocks. The last part of a file that does not fit into a file block is stored in one or more fragments. This scheme reduces the internal fragmentation while allowing large throughput.

One problem with the above scheme is that it may involve excessive copying of data. As a file grows, data in fragments may need to be copied to blocks to ensure that only the last parts of the file are in fragments. This problem may be particularly severe when a process sequentially writes large amounts of data into a file. To reduce this problem, an operating system may make the file system block size available to applications, so that they can make units of data transfer multiples of these sizes.

In the rest of the discussion we shall use the term *block* to refer to a file system block. Moreover, unless necessary, we shall not distinguish between a block and a fragment.

Reducing Latency

It is important to allocate blocks in such a way that both rotational and seek latency are reduced. Rotational latency is reduced by keeping consecutive blocks of a file in either adjacent in a cylinder or a few sectors apart. The former alternative is chosen if a large number of blocks are transferred without processor intervention. The latter alternative is chosen if each transfer generates an interrupt. It allows the R/W head to be positioned on a block when that block is needed. The number of sectors between adjacent file blocks is based on the expected time between transfer requests and is a function of processor speed.

Seek latency is reduced through **clustering**, which keeps the data in a file confined to nearby cylinders. We shall study below techniques to do such allocation.

Logical Disks

A physical disk may be divided into several **logical disks**. These define independent file systems in the sense that a file cannot span different logical disks. Logical disks can be structured to promote clustering. They also allow administrators to segregate files pertaining to different projects and to limit the amount of disk space each project uses. Finally, they support different block sizes, thus allowing logical disks to be associated with applications with similar characteristics.

Cylinder Groups

A logical disk may be further divided into **cylinder groups**, which are adjacent cylinders. Unlike logical disks, cylinder groups are not visible to a user. Thus the files of a directory can span different cylinder groups.

The operating system tries to keep the blocks of a file in the same cylinder group. However, it is dangerous to do so consistently, Files that grow may have their parts scattered in different cylinder groups. (why?) Thus in Unix 4.2BSD a file is assigned a new cylinder group when a file exceeds 48 kilobytes, and every 1 megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Also, the system keeps a certain percentage of a logical disk free to reduce the scattering of data.

Managing Free Space

The free space may be managed by keeping either a linked list of free blocks or a bit map containing a bit for each block which indicates if the bit is allocated or not. A bit map wastes space, but allows clustering.

Static Allocation vs Dynamic Allocation

So far we have assumed that files can grow dynamically. Some systems force a user to specify the size of the file when it is specified. Thus a file may be allocated a contiguous chunk. The disadvantages of static allocation is that it forces the user to provide an estimate and may cause external fragmentation/compaction. The advantage is that it is easy to implement.

Caching

Some of the information on disk is kept in memory through **caching**. Thus, when a process writes data to a file, the operating system may transfer the data to a buffer without writing to the disk. This reduces disk transfers if a process attempts to read or write the buffer before it is flushed to the disk. Caching is also useful when applied to directories that are looked-up often.

Unfortunately, caching causes data on disk to be out of step with data in memory. Thus it leads to loss of data if a system crash occurs.

Disk-Head Scheduling

The CPU can generate disk I/O requests much faster than the disk hardware can service them. Therefore, at any time there are several requests pending. We can impose an order on the outstanding disk requests so that they can be serviced efficiently. We consider below different **disk-head scheduling** policies.

The simplest policy is to order disk requests first come, first served (FCFS). This policy is fair; however, every request is likely to result in a seek. This method works well for light loads, but saturates quickly.

An alternative is to apply the shortest seek-latency first (SSF) policy, which next serves the request whose track is closest to the current one. This is the best policy for minimizing the total seek latency. However, it is less fair than the previous policy in that it makes some requests wait much longer than others. Thus the variance of wait time is large.

A compromise is to adopt the ELEVATOR policy, which uses the strategy adopted by an elevator in servicing requests. Under this policy, the disk head is either in an inward-seeking phase or an outward-seeking phase at any time. While moving inwards or outwards, the head picks up any requests that it passes. It changes direction when there are no more requests ahead.

Under ELEVATOR, a request may have to wait as long as two sweeps until it is serviced. Therefore a circular variant of it is used under high loads to improve the variation in wait times. Under this scheme the head always moves inwards. When the last seek has been performed in that direction, a single seek is performed to the outermost track with a pending request, without picking any requests on the way out. Thus a request never waits for more than one sweep.

1.4 Unix File Management

We now look at the main properties and data structures of the Berkeley 4.2BSD and 4.3BSD Unix file system. Note that it has several differences from previous Unix systems, which we shall not discuss.

1.5 Properties

The name space is hierarchical and supports aliases and indirect files.

Supports dynamic allocation.

Files and directories are stored in the same way. Thus, directories can grow and can store long file names.

Supports access lists.

Supports logical disks. Some of these are in the same physical disk, while others are in different disks.

Device names are included in the naming space.

Supports cylinder groups, blocks and fragments.

Free blocks are indicated by bitmaps.

Supports constant time seek.

File and directory contents are cached.

A file may be accessed by several processes at the same time. Some of them share the R/W mark, while others have their own copy.

1.6 Data Structures on Disk

A physical disk is divided into several logical disks. The components of a logical disk are discussed below.

Boot Block

The first sector on the logical disk is the *boot block*, containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5 kbytes.

Cylinder Group

The rest of the logical disk is composed of several cylinder groups. Each of these cylinder groups occupies one or more consecutive cylinders. The components of each cylinder group are discussed below.

Superblock

The **superblock** contains parameters of the logical disk, such as

total size of the logical disk
the block and fragment sizes
the inode number of the root directory

The superblock is identical in each cylinder group, so that it may be recovered from any one of them in the event of disk corruption. A superblock is not kept at the beginning of a cylinder group, since a single head crash would destroy all copies on the top platter. Thus a super block is stored at varying offset from the beginning of the cylinder group so that it is on a different platter from another superblock.

Cylinder Block

A **cylinder block** contains parameters of the cylinder group, such as:
bit map for free blocks and fragments,
bit map of free inodes
statistics of recent progress of allocation strategies

Inodes

There is an array of **inodes** in each cylinder group, which keeps header information about the files in the system. Each file name is associated with an **inode**, which contains the following information about the named object:
user and group identifiers of the object
time of last modification and access
number of aliases
type of the object (plain file, directory, indirect file, socket, or device)
fifteen pointers to data in the file, as discussed below.

The first twelve pointers directly point to data blocks. Thus, blocks of small files may be referenced with few disk accesses,

The next three data block pointers in the inode point to **indirect blocks**. The first of these is a **single indirect block**; that is, it points to a block that contains addresses of data blocks. The next is a **double indirect block**, which contains addresses of single indirect blocks. The third is a **triple indirect block**; it contains pointers to double indirect blocks. A minimum block size of 4096 bytes is supported so that files less than 2**32 bytes can be addressed without triple indirection. (if *b* is the minimum block size, then b/4*b/4*b = 2**32, since a block address is 4 bytes long). Files larger than this size cannot exist since file pointers are 4 bytes.

Data Blocks

The **data blocks** hold the data of both files and directories. Thus, at this level, there is no difference between a file and a directory. A directory contains a sequence of (filename, inode number) pairs. Thus to translate a file name, the system brings the inode of the root directory to memory, accesses its data blocks to find the inode of the next directory in the path, and so on, until it finds the inode of the named file.

Logical disks in Unix can be hierarchically arranged, in the sense that the root directory of one logical disk can be a subdirectory of a directory in another disk. Thus a path name "/u2/joe/foo" can span two disks, a disk that holds "/" and another that holds the tree rooted by "u2". In this situation the translation is more complex. When "u2" is encountered, the system discovers, through a bit in the inode for "u2" (the inode is stored in "/"), that another logical disk is *mounted* on this node. The system then refers to a **mount table** to find the device number of the new disk and the in-core copy of its superblock. The superblock is accessed to find the inode for the root directory of the disk, and this inode is used for "u2".

1.7 Data Structures in Memory

Each process is associated with a per-user **open file table**. This table has an entry for each object opened by the process, and is indexed by file descriptors.

Each entry in the open file table points to an entry in the **R/W mark table**, which stores the R/W marks of the open files in the system. Entries in different open file tables may point to the same R/W mark, thus allowing processes created as a result of *fork* or *exec* to share R/W marks.

Each entry in the R/W mark table points to an entry in the **active inode table**, which stores the inode of the file. Different entries in the R/W mark table can point to the same entry in the same active inode table, thus allowing different processes to have separate I/O marks for the same file.

Caching

Unix relies heavily on caching for efficiency. As we saw earlier, it keeps the inodes of open files in memory. In addition, it keeps the list of disk blocks that have been accessed recently in memory and obtains data from them whenever possible.

The path name to inode translation can be expensive, in particular, when it involves indirect files. Keeping the data blocks of directories recently accessed in memory speeds up this translation. The system takes other steps, in addition, to make this translation efficient.

It maintains a cache of (file name, inode) pairs. Also, it maintains a directory offset cache used in the following manner. If a process requests a file name in the same directory as its previous request, the search through the directory is started where the previous name was found. This strategy allows efficient handling of requests made by processes such as *ls* that scan all files in a directory one by one.