Preliminary

Grading Policy, Course Syllabus

Operating system concepts you should already know:

- process management and scheduling
- memory management
- process synchronization
- interprocess communication
- input/output devices

This course covers file systems and distributed systems. Also might discuss some advanced operating systems topics such as multiprocessor operating systems and real-time systems.

Look at concept map (topics/innovations that have effected changes in distributed systems).

Look at general issues then at specific file systems.

File Management

A *file* is a collection of permanent data that has a *name* assigned to it.

The data is permanent in the sense that it remains in existence after the process that creates it has terminated, and remains in existence until explicitly deleted. Moreover, the data remains in existence across machine shut-downs/crashes.

Note: files are abstract entities provided by the operating system, not physical entities provided by hardware.

The operating system defines:

1. how files are *named*
2. operations used to access them
3. how files are protected for these operations
4. access methods
5. their physical representation on permanent storage

Naming

A name may be divided into *components*. File names often consist of a *basename* and *extension*. The basename names the file, while the extension indicates the *type* of data the file contains.

Organizing files into types is convenient because it allows similar programs to be grouped together.

In addition, it allows applications to reject operations on files of the “wrong type”. Also to guide to correct application to operate on a file when it is opened.

Unix/Linux has the idea of “magic numbers” stored at the beginning of a file to indicates some types—executable, shell script, PostScript, etc.

However, should the operating system *require* and *force* all files into this mold?

Should the operating system restrict the length and format of filenames?

MS-DOS used a 8.3 style for names. Limit on file portion is 255 for Windows NT file system (actually keeps both long and short name). Unix variants have a large or no limit on size.

Directory

A *directory* (*folder*) is a list of filenames, along with information about the file such as:

- a pointer to its physical location
- access permissions
- owner
- size

Often, operating systems store the contents of directories in files, and the file manager treats those files specially.

Namespace

In a *flat namespace* all files are contained in a single, global directory.

Disadvantages:

- no two files can have the same name, forcing users to make up long, silly names
- searching for a file in the directory might be expensive, because the directory contains many file names.
- the cost of providing features such as file name expansion (e.g., “*” in Unix variants) becomes expensive

In a *hierarchical* file system, directories may contain pointers to other directories. All directories can be accessed by starting at the *root* directory, and names consist of a sequence of directories followed by the file name.

Now, file names can be specified as *full path names*, in which case the file is named relative to the *root* of the directory tree. The names of directories are separated by a reserved character, such as “/” (e.g., /usr/include/stdio.h) in Unix, “
” in DOS/Windows, “>” in MULTICS, “:” in MAC OS/VMS.

Relative Names

Alternatively, file names can be *relative names.* That is, they are accessed with respect to the *current* or *working directory.* (Needed in a hierarchical namespace).

The working directory is associated with each process, and a system call is used to change to another working directory.

Use of Namespace

Note: the namespace is a fundamental aspect of the operating system because users access objects through the namespace.

In Unix/Linux, for instance, devices are in the same namespace with files. Each device *foo* is given a corresponding entry of */dev/foo* in the filesystem.

Thus, users can read and write from terminals by opening (say) */dev/tty01*, and issuing reads and writes. Device independence (of course!) translates the high level operations into the device specific ones.

As another example, programs can read and write memory through the devices */dev/mem* and */dev/kmem.*

MS-DOS/Windows variants include the device as part of the name (`A:\MYDOC.DOC`) so the name is not device independent. Floppy drive, CD-ROM, hard drive, zip drive, ...

Aliases and Indirect Files

Some systems allow a file to be referenced by several *aliases*. Aliases allow the use of shorter names for files that do not belong to one directory.

Typically, the operating system supplies a system call of the form:

alias(oldname, newname)

Note: the alias routine does not create a new copy of the file, just another way of naming it.

In Unix/Linux, this alias is called a "hard link" and is created with the *ln* command (invokes *link()* system call). Example:

```
ln curname aliasname
```

Aliasing brings up two issues:

1. *deletion* — if the file is deleted under one alias, should the file be deleted for all aliases?

   Finding all of a file's aliases could be difficult, or might force the operating system to maintain a special table for aliases.

   Alternatively, a *reference count* could be associated with every file, with a count of the number of aliases. Deleting an alias decrements the reference count, deleting the file only if the count goes to zero.

2. *accounting* — who should be charged for space allocated to the file?

   We could charge the original creator of the file, but this is unfair, especially if the original creator deletes file alias.

   Best alternative is to charge each alias owner equally.

Special aliases

One important use for aliasing: how can a process read the directory corresponding to the current working directory?

In Unix(Windows), the *mkdir* (*CreateDirectory()*) system-call/command creates directories. It also creates two aliases in the new directory:

1. "." — an alias for the newly created directory itself

2. ".." — an alias for the directory in which the new directory was created.

Note: no special processing is needed to handle "." and "..". Also in Windows file systems.

Indirect Files

Unix also provides *indirect files*, files that contain (nothing but) the name of another file. The operating system provides a system call of the form:

Indirect(iname, file)

In Unix/Linux, this alias is called a "symbolic link" (or soft link) and is created with the *ln -s* command Example:
`ln -s curname symlink`

In Windows OS is called a *shortcut*.

Use of indirect files raise the following issues:

1. *deletion* — deleting indirect file *iname* does not delete *file*, but deleting *file* renders *iname* useless. Creating *name* again makes *iname* usable again.
2. *accounting* — the owner of indirect file *iname* pays a miniscule amount compared to the owner of *file*
3. *multiple indirection* — if indirect files can name other indirect files, can we have recursive names?

   To prevent infinite loops, systems typically limit the number of indirect links to a small number such as 5.
4. *interpretation of indirect names* — with respect to which directory should relative names be interpreted?

   Unix/Linux interprets the name relative to the directory containing indirect file.

Unix/Linux Link Example

```
% ls -l
total 1
-rw-rw-r--   1 cew      system         9 Mar 15 11:08 origfile

% ln origfile linkfile  # create hard link
% ls -l
total 2
-rw-rw-r--   2 cew      system         9 Mar 15 11:08 linkfile
-rw-rw-r--   2 cew      system         9 Mar 15 11:08 origfile

% ln -s origfile symlink  # create symbolic link
% ls -l
total 2
-rw-rw-r--   2 cew      system         9 Mar 15 11:08 linkfile
-rw-rw-r--   2 cew      system         9 Mar 15 11:08 origfile
lrwxr-xr-x   1 cew      system         8 Mar 15 11:10 symlink@ -> origfile
```

Types of Access

When considering objects of the type files, devices, and directories, the following access rights provide a good base:

1. Read — read the object
2. Write — write the object
3. Append — add to the end of the object
4. Execute — execute the file (but can't read it!)
5. Delete — remove the object
6. Modify Rights — change the access list of the object (owner only)
7. Set Owner — specify who owns the file (administrator)

One aspect of access control concerns *when* access rights are verified. Possibilities:

1. check each time the object is accessed; however, the cost may be quite high
2. for objects "opened" or "closed" before and after access, only perform check at open time.

   While more efficient, it cannot change the access rights for a process that has already opened the object.

Access Control

Instead of focusing solely on file access, we will consider the more general case of controlling access to arbitrary *objects*.

An object provides *operations* through which a service is invoked. A process can invoke an operation only if it has an appropriate *access right* or *privilege* to do so.

We can represent access rights by an *access matrix*.

1. Rows in the matrix represent the rights of a user (process)
2. columns denote protections associated with objects
3. a matrix entry represents a process's privileges regarding a specific resource

The following matrix gives sample protections:

	/dev/console	˜fred/prog.c	˜fred/ltr.txt	/usr/ucb/vi
Fred's P	RW	RW	RW	X
Fred's Q	RW	RW	RW	X
Jane's P	RW	R		X

All three processes can execute */usr/ucb/vi*, but none can read or write it.

Fred's processes P & Q have the same permissions. Usually, a process inherits its access rights from its parent.

Jane can read (but not write) *˜fred/prog.c*.

The access matrix is a conceptual framework and is rarely implemented as an actual matrix. (Why?) Must have an entry for each object and user. Wasteful.

Capability Lists

One way to partition the matrix is by rows, storing the access rights of a user together in a data structure called a *capability list*.

Using our previous example, Fred's list would be:

Fred—>/dev/console(RW)—>˜fred/prog.c(RW)
    —>˜fred/letter(RW)—>/usr/ucb/vi(X)

Jane—>/dev/console(RW)—>˜fred/prog.c(R)
    —>˜fred/letter()—>/usr/ucb/vi(X)

This arrangement has several drawbacks:

1. If the list has an entry for each object, many entries will indicate no access. Better to list just those objects to which the user has some access.
2. The set of objects in a list may be large, especially for privileged users. Searching long lists is expensive.
3. An initial list must be created for new users. What should the list contain?
4. What about new objects?

Access Lists

The dual of capability lists is *access lists*, which divide the access matrix by columns. The access list is associated with objects rather than with users.

1. /dev/console—>fred(RW)—>jane(RW)
2. ˜fred/letter—>fred(RW)
3. ˜fred/prog.c—>fred(RW)—>jane(R)
4. /usr/ucb/vi—>fred(X)—>jane(X)

Disadvantages:

1. set of users likely to be large
2. many processes will have identical rights

Solution: group users into *classes*. Each member in a class has the same access privileges for the object.

Systems

Unix approach partitions into three classes:

1. owner of the file
2. users in the same group as the owner
3. other users

For instance, the file ˜*fred/prog.c* might be given access rights:

self	RW
group	R
others	no access

In addition to groups, Windows NTFS allows individual users to be added to the access list of an object. Thus, ˜*fred/prog.c* might be given the following access list:

self	RW
group	R
others	no access
bob	R

Approach was used in Multics.

Access Control for Directories

The semantics of access control are fairly straightforward in the context of files. What about directories?

Here is one possibility:

- *read* — determine the names in the directory
- *write* — modify the directory contents (add, delete names)
- *append* — add new files to the directory
- *delete* — remove the directory itself
- *modify rights* — modify the access rights
- *set owner* — change the owner of the directory
- *execute* — open files in the directory. A file can only be opened if the user has execute permission in all directories of the file name

Note: should execute permission be required for the full path name? What if relative names are given?

Access Control for Aliases and Indirect Files

If a file has several aliases, should the file have different permissions for each name?

If permissions are associated with the directory entry, the answer is yes. However, this raises security concerns. because it becomes more difficult to verify all possible permissions.

If permissions are associated with the file itself, only one set of permissions can exist. This approach is taken by Unix/Linux.

How should the permissions of indirect files (symbolic links) be interpreted?

1. with respect to the indirect name?
2. with respect to the file it points to?
3. with respect to both

The former approach becomes awkward because the interpretation of the modes also depends on whether the indirect link points to a file or a directory.

Unix/Linux adopts the middle approach, ignoring protections of the indirect file (symbolic link).

Access Methods

An *access method* defines the way processes read and write files.

- *Sequential access.* The entire file is read (or written) from start to end sequentially. The system associates a *read/write mark* with each file that is advanced on each access.

  Where should the mark be stored?

  - *with each process* — only the process can modify the file mark
  - *with the file descriptor* — multiple processes concurrently accessing the file share the same descriptor—Unix/Linux approach.

    Disadvantage:

    - mark may change between accesses (non-determinism)

    Advantages:

    - useful when writing output to a common file or device (e.g., terminal)
    - replicated servers might fetch requests from a common port

- *Direct access.* Allow the user to position the read/write mark before issuing reads and writes to a file. Arbitrary data can be written to the middle of the file without destroying data before or after the new data. The system provides a system call of the form:

  position(fdescriptor, offset)

  seek() call in Unix/Linux.

- *Memory mapped access.* Multics, Digital Unix. Linux. The contents of a file are mapped into the address space of the process. Processes then access file contents through normal memory operations. See *mmap()* and *munmap()* system calls.

  Then access to the file contents is managed by the virtual memory system where the contents of the memory are backed by the mapped file.

- *Structured* or *typed files.* Consist of streams of *records*. The owner of the file describes the records of the files and the keys used to access individual records.

  Different than treating files as unstructured, byte streams. Some applications, such as databases, prefer viewing files as repositories of *records* that can be accessed with a *key*.