The University of Melbourne
School of Computing and Information Systems

COMP30023
Computer Systems

Semester 1, 2017

File systems

File systems

The purpose of a file system is the reliable long-term storage of large quantities of data.

Typically, the file system is implemented using disks (mostly magnetic disks but sometimes optical disks such as DVD-RAM). Flash memory could also be used for file systems (often for specific niches such as music players.

Tapes are unsuitable for general purpose file systems because they are too slow.

Main memory is unsuitable in general both because it is volatile and because it is too small (although one can use main memory for temporary files, e.g. /tmp on Unix).

Issues in file systems

- What constitutes a file?
- What operations are available on files?
- How are files named and protected?
- How is free space managed?
- How can the kernel ensure fast access to files?
- How can the file system recover from crashes?

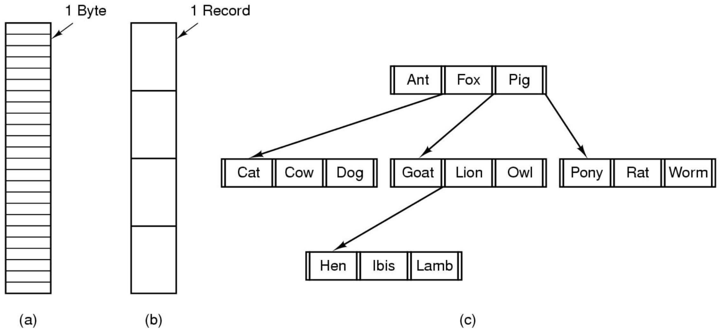
What is a file?

A file is a named "collection of bytes" stored on disk.

From the OS standpoint, the file consists of a "bunch of blocks" stored on the device. This provides an **abstraction** from the physical properties of the storage device, thereby defining a logical storage unit.

A programmer may actually see a different interface (e.g., records), but this doesnt matter to the file system (*just pack bytes into disk blocks, unpack them again on reading*).

File structure – 3 different examples



(a) byte sequence, (b) record sequence, (c) tree

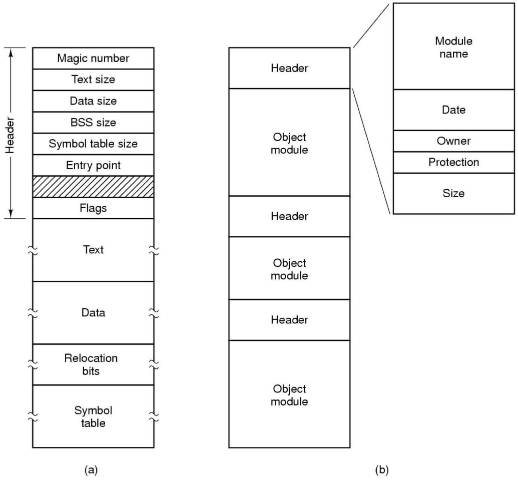
Unix view of files

Unix always views files as *sequence of bytes*. Any structure on top of this is strictly a matter for user programs.

Several libraries exist (e.g. dbm) to provide such structure. User programs can use any of these, none of these, or programmers can write their own.

The Unix view therefore provides much more flexibility than an OS that supports only a fixed set of file structures.

Example: Unix file types



(a) an executable file, (b) an archive

File attributes (representative examples)

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date**, and **user identification** – data for protection, security, and usage monitoring Information about files are kept in the directory structure, which is maintained on the disk

. . . an appropriate data structure encapsulates this information

Filenames and Types

Many OSs support two-part file system names, where parts are separated by a period (.)

eg. `filename.txt`

Extensions typically indicate the "purpose" of the file. However, not all OSs are aware of extensions. Unix/Linux does not rely on filename extensions, but some applications may expect them.

Unix also has:

- character files related to I/O and used to model serial devices (eg. `/dev/tty`, `/dev/lp`)
- block files are used to model disks (eg. `/dev/hd1`)

File system name spaces

In primitive systems, all files were in a single name space accessible to every process. This meant that users had to cooperate in the choice of file names.

Later systems break up the name space, giving a chunk of it to every user; each user can manage his/her name space without interference.

One can give every user a single name space for his/her files. This scheme works for users with few files, but breaks down for users with many files, because it forces users to choose inconvenient names, names that are either too long or too cryptic (or both). Users may even run out of ideas for names.

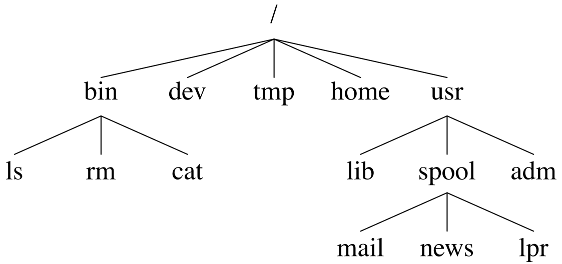
It is best to give each user multiple name spaces. These name spaces are usually known as *directories*.

Directory structure

Early directory systems had a fixed structure: a home directory and a variable number of subdirectories, one for each project the user was involved in.

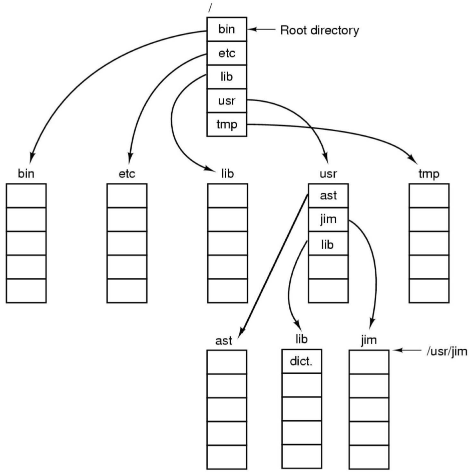
Unix was one of the first OSs to use a more flexible arrangement, where subdirectories may themselves have subdirectories, yielding a tree-structured hierarchy.

Directory tree



In a Unix file system tree, one can refer to files by either absolute pathnames (starting at the root, /) or by relative pathnames (starting at the current directory).

Directory tree (an alternative view)



Inodes

In Unix, a single data structure called an inode (information-node) contains all the information that the kernel has “about” a file, except its name. This information includes:

- type (regular file, directory, etc)
- owner
- group
- permissions
- time of last file access
- time of last file modification
- time of last inode change
- file size
- disk addresses

Directory format

In Unix, a directory is just a file maintained by the kernel.

In the original version of Unix, a directory contained a simple list of filename/inode number pairs.

Searching this unordered list can take a long time if the directory is large, so modern versions of Unix replace it with a search tree structure such as a B-tree (that still maps filenames to inode numbers).

Every directory contains the entries "." and "..", for self and the parent directory respectively. At the root, ".." refers to the same inode as ".".

Unlike many other operating systems, Unix does not require filenames to have an extension or to be in one case.

Pathname translation

A Unix filename is a sequence of names separated by slashes. The kernel translates this to an inode number by looking up each component in sequence.

If the filename begins with a slash, the kernel starts at the inode of the root directory; otherwise, it starts at the inode of the current directory of the process.

At each stage it retrieves the directory referred to by the current inode, searches it for the next component of the pathname, gets the matching inode number, and retrieves that inode from disk.

The kernel caches the results of pathname translations, so later translations of the same pathname can often be done simply by looking up the pathname in this cache.

Directory structure issues

The number of files on a system can be extensive. A procedure to deal with this “complexity” is required. A possible solution:

- break file systems into partitions (treated as a separate storage device)
- hold information about files within partitions.

A **device directory** is a collection of nodes containing information about all files on a partition. Both the directory structure and files reside on disk (backups should be kept).

Links

`ln` command in Unix.

The same inode (file) may be referred to from more than one directory entry; the inode then belongs to each such directory entry equally. Each reference is called a link.

Since two links may appear beside different names, one cannot necessarily find a single name for a given file.

Every inode contains a link count. When this count reaches zero, the file has been removed from its last directory and has therefore become unreachable, so the kernel frees its data blocks and its inode.

When you execute the shell command `ls -l'` the number between the permissions and the owner is the link count.

Symbolic links

`ln -s` command in Unix.

In Unix, hard links can only point to regular files (not directories) in the same filesystem as the link.

The reason is that each partition has its own set of inodes, and thus of inode numbers.

A symbolic link (or symlink) is a special kind of file that contains a filename. Whenever a symbolic link is opened, the kernel sees the symbolic link kind in the inode and opens the named file instead. The named file itself may be a symbolic link.

For most purposes, symbolic links are much preferable to hard links.

Shortcuts in Windows are similar to symbolic links, but have significant limitations.

The access matrix

The OS must know which *subjects* are authorized to perform which *operations* on which *objects*. Typical subjects are users or processes executing on their behalf; typical objects are files, areas of memory, interprocess communication mechanisms, etc.

A full access matrix that lists every subject and every object is inconvenient, because of its large size and its dynamic nature. Most systems therefore store authorization information with either subjects or with objects.

They also tend to use tricks to both compress this information and to avoid the requirement for frequent updates.

Access control lists

The logical place to store authorization information about a file is with the file itself or with its inode.

Each element of an ACL contains a set of user names and a set of operation names. To check whether a given user is allowed to perform a given operation, the kernel searches the list until it finds an entry that applies to the user, and allows the access if the desired operation is named in that entry. The ACL usually ends with a default entry.

Systems that use ACLs stop searching when they find an ACL entry that applies to the subject. If this entry denies access to the subject, it does not matter if later entries would also match and grant access; the subject will be denied access.

This behavior is useful for setting up exceptions, such as everyone can access this file *except* people in this group.

Revision: Unix file access permissions

Mode of access: read, write, execute

Three classes of users: user (owner), group and other (public)

In UNIX, 3 fields (ugo) with length of 3 bits (rwx) are used

eg. `chmod go+rw file` or `chmod 761 game`

Revision: Unix groups

Each user is a member of one or more groups. If there is more than one group for a user, one of the groups is considered primary and the others secondary.

Groups also have a name and a numeric group-id (`gid`). They should also be unique.

There is no relationship between group names and user names or between uids and gids.

Information about users and groups is stored in two files which are almost always readable by all users. The files are `/etc/passwd` and `/etc/group`.

Unix file security

The security of Unix files is protected by an ACL of a restricted form, with three entries.

- The first applies to the owner of the file.
- The second applies one to members of the group associated with the file.
- The third applies to everyone else.

This ACL can be represented in nine bits that are often represented as an octal number.

```
751 rwxr-x--x u1 g1
705 rwx---r-x u2 g2
```

Understanding Unix file access permissions

```
1. File: ‘demo.txt’
2. Size: 58   Blocks: 2  IO Block: 8192  regular file
3. Device:30c0627h/51119655d Inode: 320966   Links: 2
4. Access:(0760/-rwxrw----)  Uid:(341/ mkirley)  Gid:(10/ sta
5. Access:2010-05-14 17:46:46.562688596 +1000
6. Modify:2010-03-29 13:19:05.582539168 +1100
7. Change:2010-05-14 17:53:31.271288540 +1000
```

You should be able to explain what each entry means.

Permissions for directories

The Unix kernel uses the execute bits of directories to determine whether a process has a right to look for a file in that directory.

The kernel uses the write bits of directories to determine whether a process has a right to create new files or delete old files in that directory. This makes sense if you consider that a Unix directory is a table of filename/inode number pairs, and creating new files and deleting old files both update this table.

As a special case, when the *sticky* bit is set on a directory, the write bit allows a process to delete a file from the directory only if the user id of the process matches the file's owner, which usually means that the process was invoked by the file's owner (but see the slide on "setuid" below).

Umask

When a process creates new files, the default is to create them with public access, i.e. with mode 666, or mode 777 for executable files.

Users may specify a mask that is applied to the default permission to clear some of its bits.

The default umask is usually 022, so most files have 644 or 755 permissions. Paranoid users can set their umask to 077.

The umask facility is a *mechanism*. The default umask is a *policy* (see later).

Chmod, chown and chgrp

Unix users can change the ACLs of files using the commands “chmod”, “chown” and “chgrp”, but there are restrictions.

Only the owner of a file (and the superuser) can change the protection attributes of that file.

Versions of Unix with quotas allow only the superuser to execute chown; otherwise it could be used to cause other people to be billed for the storage consumed by a file.

Only the superuser can change the group of a file to a group of which he/she is not a member.

Temporary rights amplification

Processes often need to perform operations that could lead to a security breach or a crash if used irresponsibly but that can be done safely in a controlled manner.

One example is writing to someone's email inbox.

For these situations, the OS should provide mechanisms that allow processes to gain privileges as long as the code they execute with those privileges can be trusted to behave responsibly.

Set user-id programs

When a Unix process invokes `exec` on an executable file whose set-user-id bit is on (which makes the file a *setuid program*), the kernel executes that program with the effective user id of the owner of the file.

Similarly, a setgid program is executed with the effective group id given by the group field of the inode of the executable file.

Permissions are checked against the effective ids, so a setuid program can do things that the owner of the file can do, and a setgid program can do things that the group of the file can do.

Such a program can also do things that its invoker can do, because it can ask the kernel to use its invoker's user or group id as its effective id for a time.

The superuser

Every Unix system has a user called *root* (the superuser). The kernel allows root to perform any action without any permission checks. Many setuid programs are setuid to root.

An example is the mail delivery program. Users should have access only to their own mailbox files, but they need to be able to append messages to the end of other people's mailboxes. The mail program is setuid to root, so it can modify any file, but it can be trusted to only append to mailbox files.

The Windows equivalent to the superuser is an account with administrator privileges. On most versions of Windows, every user gets these privileges by default.

The su and sudo commands

The su (substitute user) command is implemented as a program that is setuid to root. It takes a user name as argument (root is the default), requests the password for that user, and attempts to verify the user's answer. If the password checks out, it creates a shell that has the privileges of the named user.

Sysadmins use their own accounts most of the time and run as superuser only when they need to, to minimize the consequences of accidents.

Recent Unix systems have a program called "sudo" that allows sysadmins to execute a single command with superuser privileges.

"sudo" can be configured so that certain users (e.g. help desk personnel) can execute some low-risk programs (e.g. restarting a line printer daemon) without supplying their password. This can considerably enhance convenience at a small cost in security.

File operations

A file is an abstract data type, where the following operations are defined:

- create
- write
- read
- reposition within file
- delete
- truncate
- open(*fi*) search the directory structure on disk for entry *fi*, and move the content of entry to memory
- close (*fi*) move the content of entry *fi* in memory to directory structure on disk

File descriptors

A Unix program refers to a given input or output stream by a **file descriptor** (so named because most streams read from or write to a file).

A file descriptor is a small integer.

Most system calls that manipulate files (read from ,or write to files or other streams) take a file descriptor as one of their arguments.

The kernel uses the file descriptor to index into a table representing the process's open files. Only the kernel can access this table.

It is impossible for a user program to access a file unless the kernel says it is OK for it to do so (based on entries in the table).

Unix system calls

open	Checks whether the requested operation is permitted on the named file. If yes, returns a file descriptor for that file, making it available for reading or writing or both.
read	Read a given number of bytes from the given file into a given buffer; returns the number of bytes actually read.
write	Writes a given number of bytes to the given file from a given buffer.
close	Makes the given file descriptor unavailable for further operations.

Creating a file

In Unix and many other OSs,

- files are created at zero size through a variant of the "open" system call, and
- a write beyond the current size of the file extends the file to the necessary size.

Historical note: In some operating systems from the 1960s, the size of a file had to be specified in advance, as part of the system call that created the file. This is inconvenient, so OSs designed since the 1970s drop this requirement.

Using file system calls (from Tannenbaum)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                    /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);         /* ANSI prototype */

#define BUF_SIZE 4096                     /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                  /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);              /* syntax error if argc is not 3 */
```

Using file system calls (from Tannenbaum)

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);    /* open the source file */
if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
     rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;           /* if end of file or error, exit loop */
     wt_count = write(out_fd, buffer, rd_count); /* write data */
     if (wt_count <= 0) exit(4);    /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0)                  /* no error on last read */
     exit(0);
else
     exit(5);                       /* error on last read */
}
```

Current offset

The Unix kernel maintains the notion of a *current offset* into a file: the value of this variable is the distance in bytes between the beginning of the file and the next byte to be read or written.

Sequential access is the default: read and write requests advance the current offset by the number of bytes read or written.

Programs can perform random accesses by manipulating the current offset via the lseek system call.

Tables for file operations

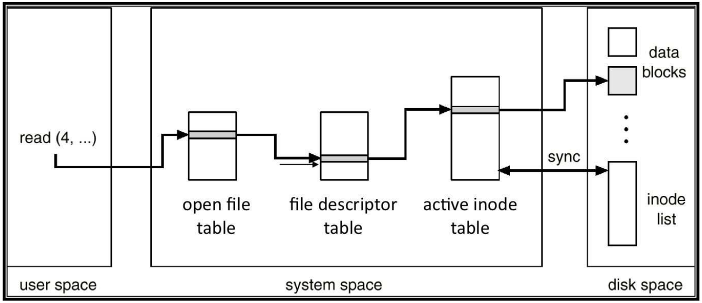
The Unix kernel keeps three main kinds of tables related to open files:

the open file table a global table containing one entry for each open without a close, with each entry containing a current offset; and

file descriptor tables small per-process tables, each of which maps the file descriptors of that process to open file table entries.

the active inode table a global table containing information about every active file (an inode contains information about a file; more about them later);

Tables for file operations (from Silberschatz et al)



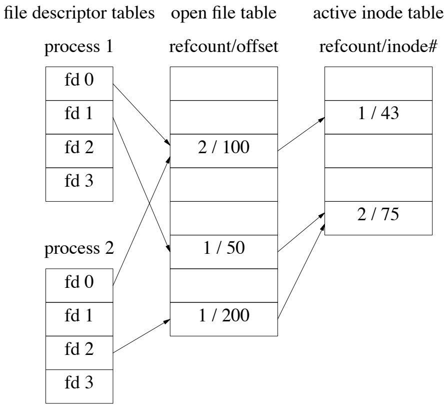
The open file table

The per-process file descriptor table is copied when a new process is created via a `fork()`. The new process can close some files, open others, and leave the rest alone.

If two processes inherit the same open file from a common ancestor, they share orderly access to that file without any special arrangements.

Each operation by either process advances the current offset stored in the open file table.

Example: file tables



The two processes inherit **fd0** (pointing to the file with inode #43) from a common ancestor and both open the file with inode #75 themselves.

Standard file descriptors

Unix has a convention for the use of three file descriptors:

> standard input (**stdin**) – program can read what the user types
>
> standard output (**stdout**) – program can send output to user's screen
>
> standard error output (**stderr**) – error output

Every program invoked by the shell can count on having these file descriptors open.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process (I/O redirection). Most programs can also accept a file (rather than a terminal) for standard input and standard output.

Setting up standard file descriptors

```
if ((pid = fork()) == 0) {
    /* code for the child */
    /* redirect stdin */
    close(0);
    open(stdinfile, O_RDONLY, 0);
    /* redirect stdout */
    close(1);
    open(stdoutfile, O_WRONLY, 0644);
    /* redirect stderr */
    close(2);
    dup2(1, 2);

    execv(cmd, argv);
}
```

Another example

File “script”:

```
#!/bin/sh
prog1
prog2
```

Command:

```
script < infile > outfile
```

prog1 and prog2 inherit the same open file table entries as each of stdin, stdout and stderr from the shell.

The shared open file table entry for fd0 tells prog2 how much of `infile` was consumed by prog1, while the shared entry for fd1 says how much of `outfile` it has written.

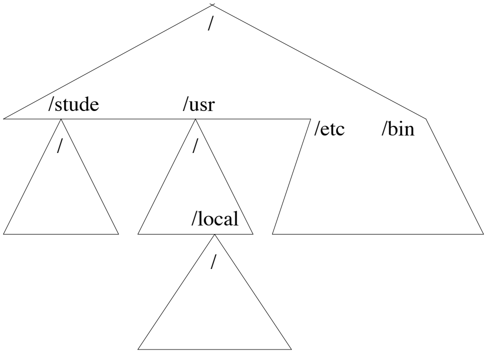
File systems

Under Unix, a disk drive is divided into one or more **partitions**. The kernel uses each partition as a virtual disk. A partition may serve as (a component of) swap space for the system, or it may hold a *file system*.

A file system is a tree-structured hierarchy of directories. Unix allows one file system to be mounted on top of a (preferably empty) directory in some other file system. The root directory of the mounted filesystem conceptually replaces the mounted-on directory.

Mounted-on directories and root directories are flagged as such in the copy of their inodes held in memory.

File systems



When pathname translation arrives at a mounted-on inode, it should continue from the root inode of the filesystem mounted there. Exception: the lookup of the name ".." should be done from the mounted-on inode.

Mounting filesystems

Filesystems may be mounted and unmounted at any time when they are not used, although usually only filesystems on removable media (and network filesystems imported from other hosts) are mounted after system startup. The root filesystem must stay mounted while the system is up.

Every machine has a system file (usually /etc/fstab or /etc/vfstab) that describes the configuration of its filesystems.

Entries in this file give a partition name, a filesystem name, and some options (e.g. is the filesystem read-only, are there quotas on that filesystem, etc).

Users home directories

Users home directories (and sub-directories) are likely to be found under one of the following directories:

```
/home

/export/home

/home/group

/usr/home
```

Note that the disk partition where the home directories are kept must be big enough to cope with all the files the user's will create!

Structure of a file system

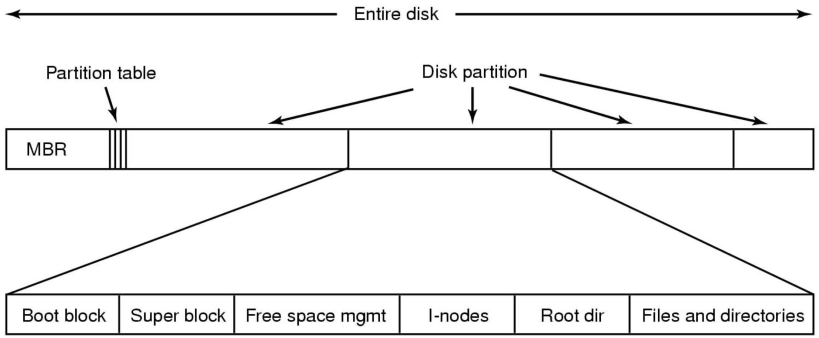
A traditional Unix file system contained the following:

- the boot block (block 0): contains code to bootstrap the system
- the super block (block 1): contains summary information for the file system (where to find everything else)
- blocks for a fixed number of inodes
- blocks for file data

Each disk block consists of a fixed number of disk sectors.

If a file system uses e.g. 4 Kb blocks, then each block consists of eight consecutive sectors, each of 512 bytes.

Structure of a file system (possible layout)



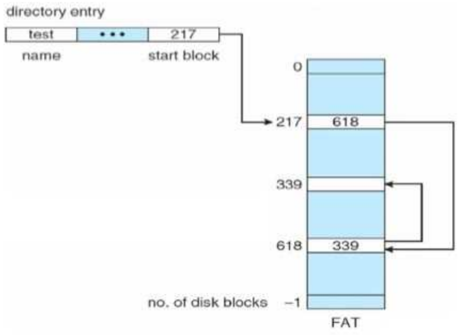
File allocation methods

An allocation method refers to how disk blocks are allocated for files:

- Contiguous allocation
- Linked allocation
- File allocation table
- Indexed allocation
- Multi-level indexed allocation

File allocation table (FAT)

A File Allocation Table (FAT) is similar to linked allocation, but the linked list portion of the blocks is stored in a separate table called the DFAT. This speeds up direct access (only access the FAT); the FAT can be cached.



Multi-level indexed allocation

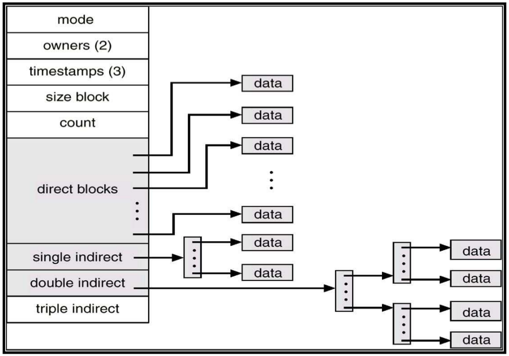
The inode consists of pointers to index blocks.

For a two-level table:

- the inode points to a “second level” index block
- each “second level” index block points to data blocks

High levels of indirection are possible – intermediate index blocks are allocated when needed.

Unix inode map (from Silberschatz et al)



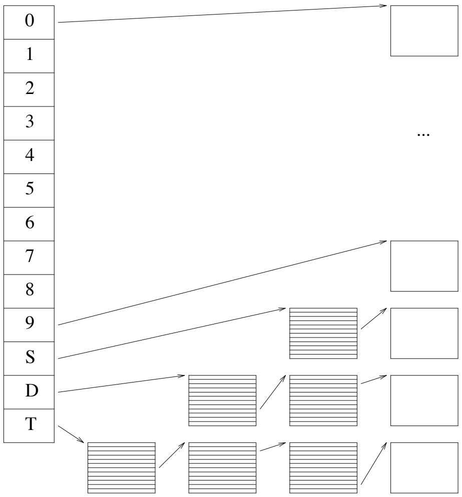
The traditional structure of Unix files

The inode contains pointers to the first ten blocks of a file. Blocks beyond the tenth are accessed via indirection blocks:

- a single indirect block contains addresses of data blocks;
- a double indirect block contains addresses of single indirect blocks;
- a triple indirect block contains addresses of double indirect blocks.

Most files fit in the ten direct blocks; very few need a double indirect block. If blocks are big enough, none need a triple indirect block.

Unix inode map



Exercise: Unix inode map

. . . to be completed in the lecture

The traditional structure of Unix files (cont'd)

The previous diagram illustrates the structure of files in the first versions of Unix in the 1970s, and many Unix variants have kept using this. These include the ext2 and ext3 file systems on Linux.

However, some variants of Unix, and some other filesystems on Linux, use other structures.

Linux file systems

To the user, Linux's file system appears as a hierarchical directory tree obeying Unix semantics.

Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS).

The Linux VFS is designed around object-oriented principles and is composed of two components:

- A set of definitions that define what a file object is allowed to look like. The inode-object and the file-object structures represent individual files. The file system object represents an entire file system
- A layer of software to manipulate those objects.

Linux device-oriented file system

The Linux device-oriented file system accesses disk storage through two caches:

- data is cached in the page cache, which is unified with the virtual memory system
- metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.

Linux splits all devices into three classes:

- block devices allow random access to completely independent, fixed size blocks of data
- character devices include most other devices; they dont need to support the functionality of regular files.
- network devices are interfaced via the kernels networking subsystem

Virtual file system switch

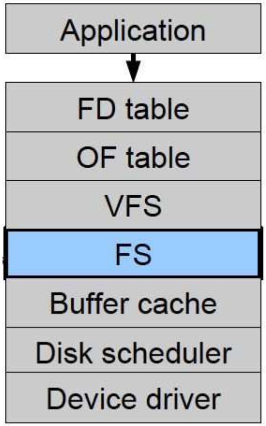
One way to implement alternative file system types is via a virtual file system (VFS) switch.

This is a two-dimensional table mapping system calls (one dimension) and file system types (the other dimension) into a pointer to a procedure that implements the appropriate variant of the system call for files on that type of filesystem.

If the system call does not apply to that file system type (e.g. seeking on a terminal or a pipe), then the procedure will always return an error indication.

This technique allows programs to access many different types of file systems through the same interface. This is how e.g. a single program on a Linux machine can simultaneously use e.g. the ext3, efs, CD-ROM and /proc filesystems without being aware of the fact.

An alternative perspective:



The buffer cache

A disk block that has been accessed recently will probably also be referenced in the near future. Unix therefore caches disk blocks, including blocks holding inodes and indirection blocks, in main memory. The memory used for this is called the buffer cache.

All I/O transfers in Unix move data between disk and the buffer cache. The replacement strategy of the buffer cache is usually LRU: every reference to a buffer moves that block to the end of the queue.

However, some versions of Unix keep indirection blocks in the buffer cache regardless of age until the inode to which they belong becomes inactive.

Buffer cache use

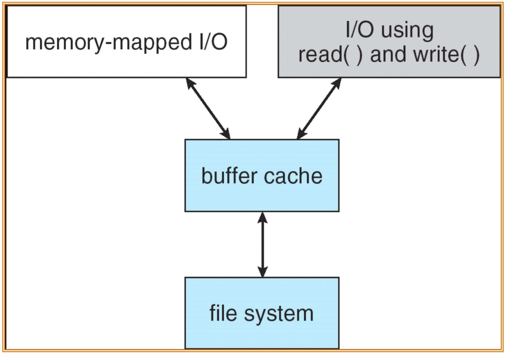
A read system call can succeed without a disk access if the necessary blocks are in the buffer cache; the kernel just copies the data from there to the user address space.

If recent accesses to this file were sequential, Unix will prefetch the next block, i.e. start reading it into the buffer cache ahead of time. This is *read-ahead*.

Write system calls copy data from the user address space to the buffer cache. The blocks affected are marked "delayed write". A daemon process that wakes up every 30 seconds schedules all such blocks to be written out to disk; the mark is deleted when the write is completed. This is *write-behind*.

Blocks are deleted (*evicted*) from the buffer cache only when their space is needed to cache some other block. If the evicted block is marked delayed write, it is written to disk first.

Unified buffer cache



A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O

Advantages and disadvantages

- + Reduce number of disk reads required if references show locality.
- + No need to require read/write requests to be aligned.
- + Files with short lifetimes ($<$ 30s, e.g. compiler temporaries) usually require no disk accesses.
- + If a file is updated several times within a short time, only the final version usually needs to be written to disk.
- + Scheduling the writing of many blocks at the same time gives more scope for the disk scheduler.
- + No need to lock user pages in memory during I/O transfers.
- - Disk accesses require an extra pass over the data, e.g. disk to buffer cache and then to user memory, versus just disk to user memory.
- - Delayed writes may be lost in a crash.

Interprocess communication

Suppose a program wants to send a large amount of information to another process. One way to do it is to save the information in a file, like this:

```
prog1 > tmpfile
prog2 < tmpfile
```

However, this requires prog2 to wait until prog1 has completed, and one must remember to delete tmpfile afterwards.

. . . see code example

Pipes

A Unix pipe is an interprocess communication channel that can be accessed with read and write operations.

```
    prog1 | prog2
```

Each pipe has an associated buffer (a typical size is one page). A read on an empty pipe and a write on a full pipe both cause the process concerned to suspend. Read and write fail on broken pipes.

Pipes provide a reliable unidirectional byte stream between two processes.

A benefit of the pipes' small size is that pipe data are seldom written to disk; they usually are kept in memory by the normal block buffer cache.

Pipe creation

The `pipe` system call creates a pipe and returns two file descriptors: one for reading and one for writing on the pipe.

When processing a command line like "`prog1 | prog2`", the shell causes the write end of the pipe to become the standard output of prog1, and the read end of the pipe to become the standard input of prog2.

Each subprocess has access to only one end of the pipe: the shell closes the other before the exec.