The File System

Any operating system must provide a framework for the storage of **persistent** data, i.e. those data which live beyond the execution of particular programs and which are expected to survive reboots. UNIX-like operating systems place a great deal of emphasis on the file system and present a fairly simple but robust interface.

The file system arose historically as a way of managing bulk storage devices. When data were stored on punched cards or punched tape, the grouping of individual records into files and of files into "folders" was purely physical, as these paper data were kept in specialized file cabinets. With the advent of magnetic media (disk, drum and tape), it was now incumbent upon the operating system to manage these raw bits and organize them so that users could store and retrieve their files.

Today, persistent storage is provided by one of two technologies. The Hard Disk (or sometimes Disc) Drive (HDD) uses one or more spinning platters of magnetic material with tiny read/write heads flying a few thousandths of an inch above the platter surface. The other major technology is persistent solid-state memory, provided by "Flash" electrically erasable programmable read-only memory (EEPROM) chips in various configurations. Solid-state drives (SSDs) are flash devices optimized for use to replace HDDs as the primary storage for a desktop, laptop or server system, while other flash products such as the ubiquitous USB memory "stick" or SD cards used in cameras and phones, are designed and priced with less writing activity in mind.

In this unit, we won't go much further into the construction of HDDs or SSDs but will focus on how the operating system manages their contents. We can think of a disk as a randomly addressable array of bytes. However, since a byte is a very small amount of storage, all mass storage devices have a minimum addressable element size which is known as a **disk block** or **sector**. The latter term is preferred since disk block could mean something else, as we'll see later. Historically most devices used a 512 byte sector size. With hard drives breaking through the 2TB mark, this has required a shift to 2048 byte sectors. (If the sector number is considered as a 32-bit unsigned number, with 2^32 maximum sectors and a sector size of 2^9, the total disk size would be limited to 2^41 or 2 TB).

To give things a clear name, we'll call each such randomly-addressable array of sectors (whatever their size) a **volume**. A volume may be an entire hard disk, a removable storage device, or a partition.

How, then, do we go about organizing those bytes of the disk to form a permanent data structure, much as we organize the bytes of RAM to form in-memory data structures? Each operating system historically had both its own layout scheme of bytes on a disk, and its own interface details, such as how files and directories were named, and what other information was kept.

There were also radical variations in how operating systems presented an interface for users or programs to explore and manipulate its system of files. We have already seen that UNIX takes the simple "just a bunch of bytes" approach. This was not so with many other operating systems, which often had built-in notions of file "types" (e.g. a program source file was a certain type of file for which different operations were possible from, say, an executable file).

What the File System provides

Every type of filesystem, from the simplest to the most robust, must provide certain basic services to the user. These services are delivered via the kernel system call interfaces.

• **Naming**: We must have a way of giving human-readable names to the files (and/or other objects) stored in the filesystem.

• **Data Storage**: Obviously the filesystem must provide a way to read and write the actual data contained in the files.

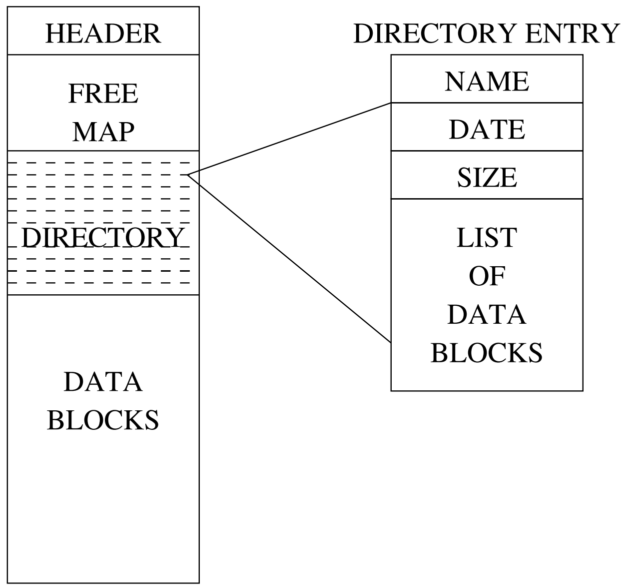
• **Meta-data**: The filesystem tracks other data which are not the actual contents of the files, but are data about the data, or "meta" data. These data may include the size of the file, time of last modification, the creator, etc.

• **Free space management**: The disk on which the files are stored is finite. We must have a way of determining how much empty space is left on the disk, and how much space each file is taking up.

Flat file systems

The simplest file system is a flat one, in which there are no subdirectories, and the number of files is limited to a fixed number. Historically, the last major general-purpose operating system to use a flat filesystem was MSDOS version 1.

However, today flat filesystems may still be found where storage needs are simple and adding the complexity of a directory hierarchy is prohibitively expensive, for example, many embedded devices, such as digital cameras or network switches/routers. As processor power and ROM size grows with embedded processors, flat filesystems are becoming more rare.



This volume is divided into 4 distinct parts. A header block serves to name to volume, give its size, and provide other summary information such as how much free space is available.

The contents of the files are stored within the data block section. The free map section maintains an overview of which data blocks are currently being used, and which are free to be allocated to files as they are written. Common implementations of the free map are a linked list of free blocks, or a bitmap with each bit representing the status of one block.

In the FAT filesystem (aka the MSDOS filesystem), the free map area (or "File Allocation Table") contains many linked lists. One is the list of free blocks, and additional lists chain together the data blocks comprising each file. This strategy does not perform well for random-access to file contents.

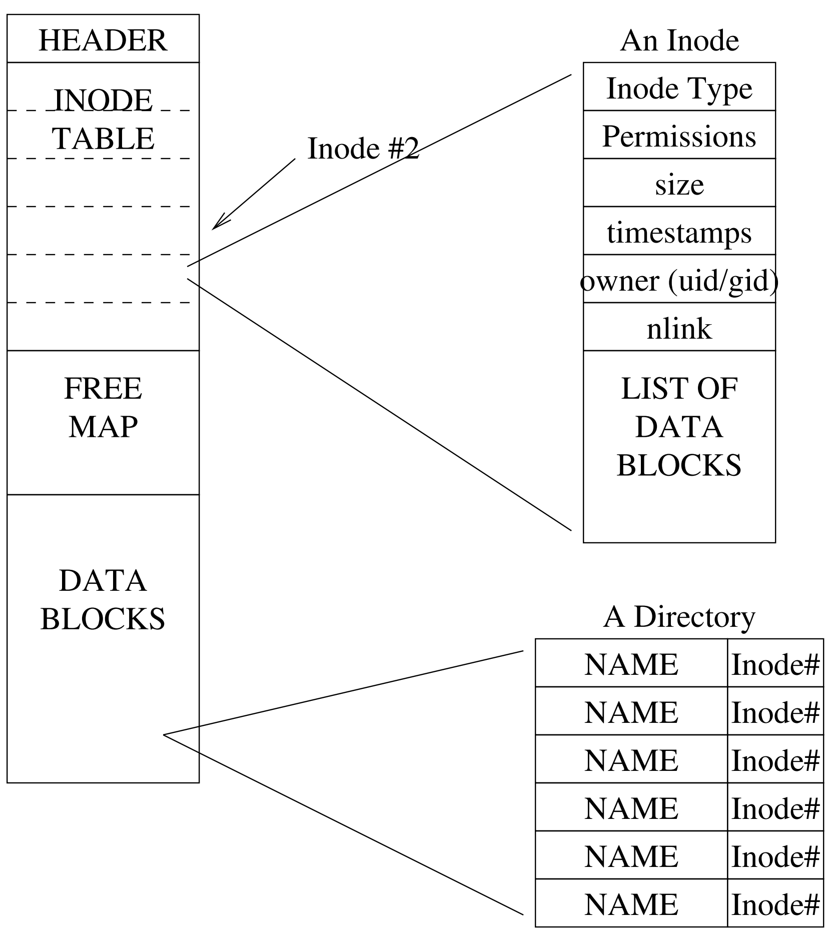
The directory lists the names of the files on the volume. Note that in this flat filesystem, the number of directory slots is fixed, and each directory entry contains everything there is to know about a file: its name, its **metadata** (such as the file size and time of last modification), and the mapping of the file contents to specific data blocks. The metadata are not the actual contents of the file, but are still important to users and programs, and are thus accessible through the system call interfaces (e.g. `stat(2)` in the UNIX kernel). Other data within the directory entry slot, e.g. the list of data blocks, is not meaningful at the user level, and is thus typically not accessible except by viewing the raw underlying on-disk data structure.

The MSDOS filesystem had a fixed file name size of 8 characters plus a 3-character extension, which determined how the file was treated. Common extensions included .COM and .EXE for executable programs, .BAT for MSDOS batch (shell script) language, .SYS for system files, .TXT for plain text files, etc. This 8+3 filename structure still haunts us today. E.g. TIFF files are .TIF, and JPEG files are .JPG

The simple flat filesystem can be trivially extended, e.g. the MSDOS/FAT filesystem, by adding a flag to a directory entry which causes that "file" to be treated as another directory. This allows a fully hierarchical view. We're not going to spend any more time on these primitive filesystems, and the curious reader is invited to consult online resources which describe FAT/VFAT/FAT32 filesystems more completely.

UNIX Filesystems

The UNIX file system model is derived from the actual data structures which were used on disk many years ago in the earliest UNIX operating systems. The approach taken back then was flexible enough that modern UNIX systems can use the same interface to "seamlessly integrate" many volumes of many different file system organizational types into one hierarchy. Let's begin by exploring, abstractly, how UNIX organizes a volume.



Once again, the volume comprises 4 distinct areas, the size of each of which is fixed at volume initialization time. The UNIX command to create a filesystem on a volume is called `mkfs`. This command accesses the disk directly, on a byte-by-byte raw basis, and lays out the filesystem data structure.

The volume header contains miscellaneous information about the entire volume, such as a descriptive name, the number of active files within the volume, the time of last use, and other critical information. For historical reasons, this header data structure is often called the "superblock". The superblock describes the size and layout of the rest of the volume. If the superblock were to be lost or corrupted, then no data could be accessed as it would be impossible, e.g., to discern where the data block section was. For this reason, UNIX operating systems keep redundant copies of the superblock at other well-known locations in the volume.

The data block section can be thought of as a resource pool, divided into **filesystem allocation blocks** of a certain size. Historically, UNIX used 512 byte blocks, and this size still creeps into certain dark corners of the operating system. However, larger block sizes such as 1K, 2K, 4K or 8K are more common. Given the block size, we can think of the data block area as an array of blocks, indexed by a block number. (For reasons which may become clearer after reading some kernel source code, the first block number is not 0, but some larger number). Filesystem blocks are not necessarily the same size as sectors. They are usually larger, although some pathological cases could exist (e.g. creating an old System-V type filesystem with 512 byte blocks on a 2+TB drive with 2K sectors).

The filesystem data block is the smallest unit of storage allocation in the filesystem. If the block size is 1K, then a file which is 518 bytes long still consumes 1024 bytes of disk space. There is a tradeoff between space efficiency and time efficiency and the selection of block size can be tuned accordingly.

What was unique about the UNIX approach was the treatment of directories as just another type of file. Thus the directories are stored in the same data block resource pool as the file contents.

Another unique feature of the UNIX filesystem was the divorcing of metadata information from the directory entries. A directory in a UNIX filesystem is simply a list of filenames. Information about a single file or directory is kept within a data structure known as an **inode**.

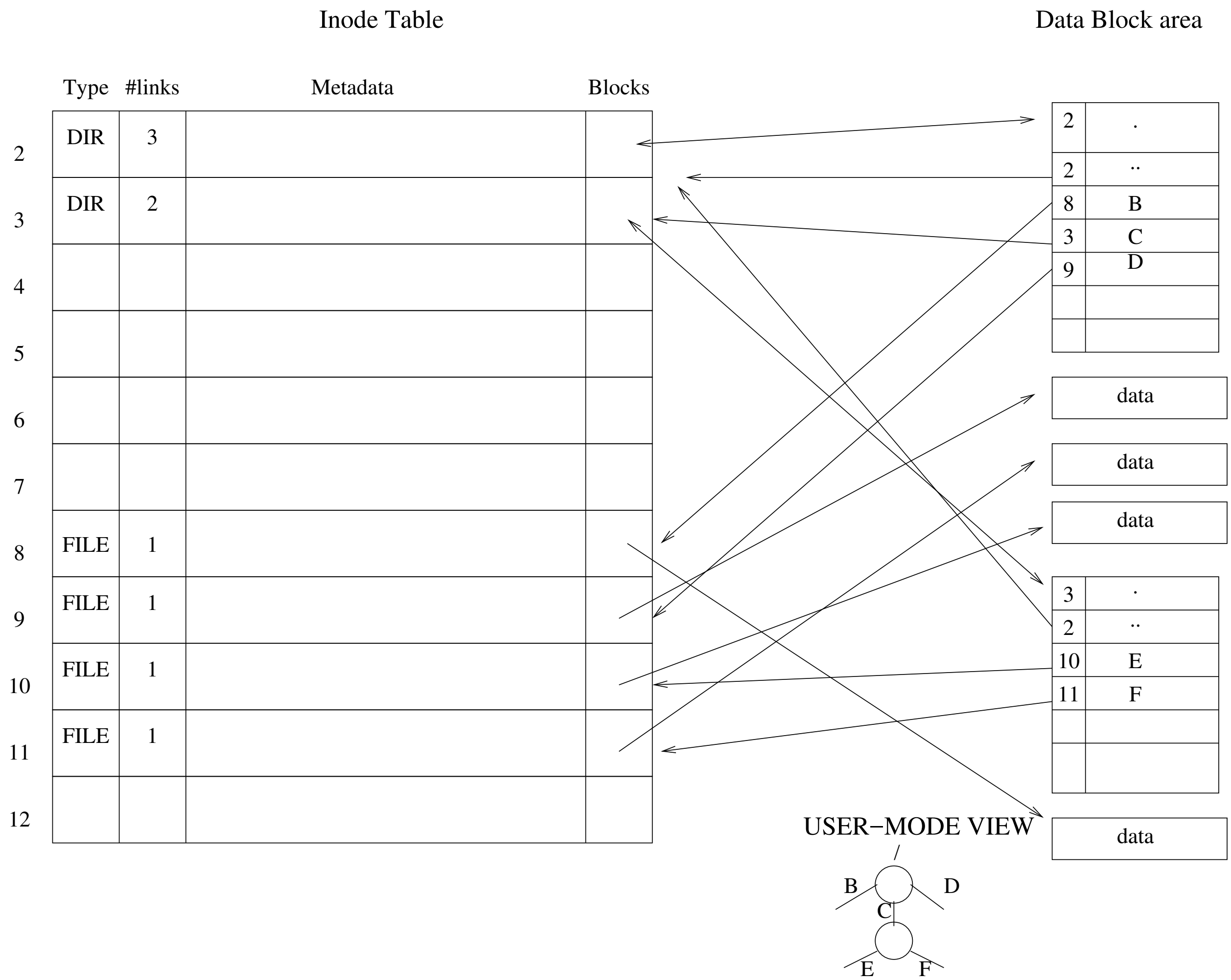
The inode table is conceptually an array of these inodes (a typical on-disk inode size is 128 bytes), indexed by **inode number**. Again, for historical and kernel programming reasons, the first inode is usually numbered 2 (because 0 was used to indicate an empty directory slot, and 1 was reserved for the boot block).

The inode contains all of the **metadata**, such as the size, owner, permissions and times of last access. It also contains (conceptually) the list of block numbers which comprise the contents of the file or directory. Another important bit of metadata is the **inode type**, e.g. is this a regular file or a directory.

The free map provides an overview of which data blocks are in use, and which are free to be allocated to files. Most modern UNIX systems use a bitmap representation of the free map.

Directories and inodes

A directory is simply another instance of a file, specially identified as such via the Type field in its inode. Logically, a directory is an unordered list of names and inode numbers. In actual implementation, directory entries may be of fixed or of variable length, and in some cases more sophisticated data structures such as hashes or trees are introduced to make searching them faster. In the original UNIX filesystem, the name field of the directory entry was fixed at 14 characters, and the inode number was 2 bytes long, yielding a fixed 16-byte directory entry size, and also imposing an annoying 14-character limit on file names. Most modern UNIX filesystems allow at least 255 characters per path component name, and at least 1024 characters for a complete path name.



Note that the directory entry does not contain any other metadata information. In order to retrieve that, or to figure out where the associated file's contents are in the data section, the inode must be consulted. Effectively, the inode number in the directory entry is a **pointer** to the inode. Note that the inode, in turn, does not have a "Name" field. A file is only given a name in the sense that there exists a path that refers to it. Given a particular inode, there is no way to find the path or paths associated with it except through exhaustive search of the filesystem.

Inode numbers can be considered "cookies". They are integers with definite bounds that can be compared for equality, but no other semantics can be inferred. In particular, the fact that one inode has a lower number than the other does not necessarily imply that it was created first. As a filesystem ages and files are created and removed, inode numbers will get recycled. Similarly, even though, e.g., inodes 69 and 71 exist, there is no basis to assume that inode 70 exists.

Pathnames and Wildcards

There must be a starting point, a root of the tree. By convention, the first inode of the filesystem (which has inode #2) is a directory and forms the "root directory" of the filesystem. UNIX pathnames that begin with "/" are evaluated starting from the root. These are known as **absolute**, or **fully-qualified** paths. Otherwise, they are **relative** paths and are evaluated starting at the **current working directory** (`cwd`, which is a state variable associated with each process).

A pathname under UNIX is a series of component names delimited by the pathname separator character, forward slash (/) . Each component is a string of **any** characters, **except for the / character or the NUL terminator (\0) character**. The length of each component name has a finite limit, typically 255 characters. Each component of a pathname, except for the last, must refer to a directory. While there is no limit on the number of components in a pathname, there may be limits as to the total length of the pathname string, such as 1024 characters. This is half a screen of text so one would not want to have to type such a long pathname too often.

Doubtless the reader is familiar with UNIX wildcard syntax, such as `rm *.c`. Wildcard expansion is performed by the **shell**, which is the UNIX command-line interpreter. In a later unit, we will see how the shell functions. From the standpoint of the operating system kernel and system calls, there are no wildcards. The * or ? characters have no significance and are valid path component name characters. So are spaces, control characters, hyphens and a host of other characters which often cause confusion to novices.

Note also that UNIX does not have a notion of a file "extension" as does the DOS/WINDOWS family of operating systems. There is a naming convention which some programs follow. E.g. the C compiler expects that C source code files end in `.c`. This is strictly an application-level issue, and is not the concern of the kernel in any way.

The component names `.` and `..` are reserved. They are always found in any directory, even an freshly-created empty one, and refer to the directory itself and to the directory's parent directory, respectively.

Because empty component names do not make any sense, any sequence of forward slashes in a pathname is equivalent to just one. E.g. "/C//////E" is the same as "/C/E". As a result of this, and the . and .. names, **there is an unbounded number of possible pathnames which refer to the same node**. The existence of hard links makes this statement even more important.

Hard Links

To draw the analogy to the world of paper records, the UNIX file system is a file cabinet where the individual files are given arbitrary serial numbers (the inode number), and there is a separate card catalog index which allows us to determine the number associated with a particular naming. It is often useful to retrieve a given physical file under multiple names or subjects, e.g. the file for product "X1000" might be filed under Products/ Widgets/ X1000; also under Product Recall Notices/Safety Critical; and Patents/Infringement Claims.

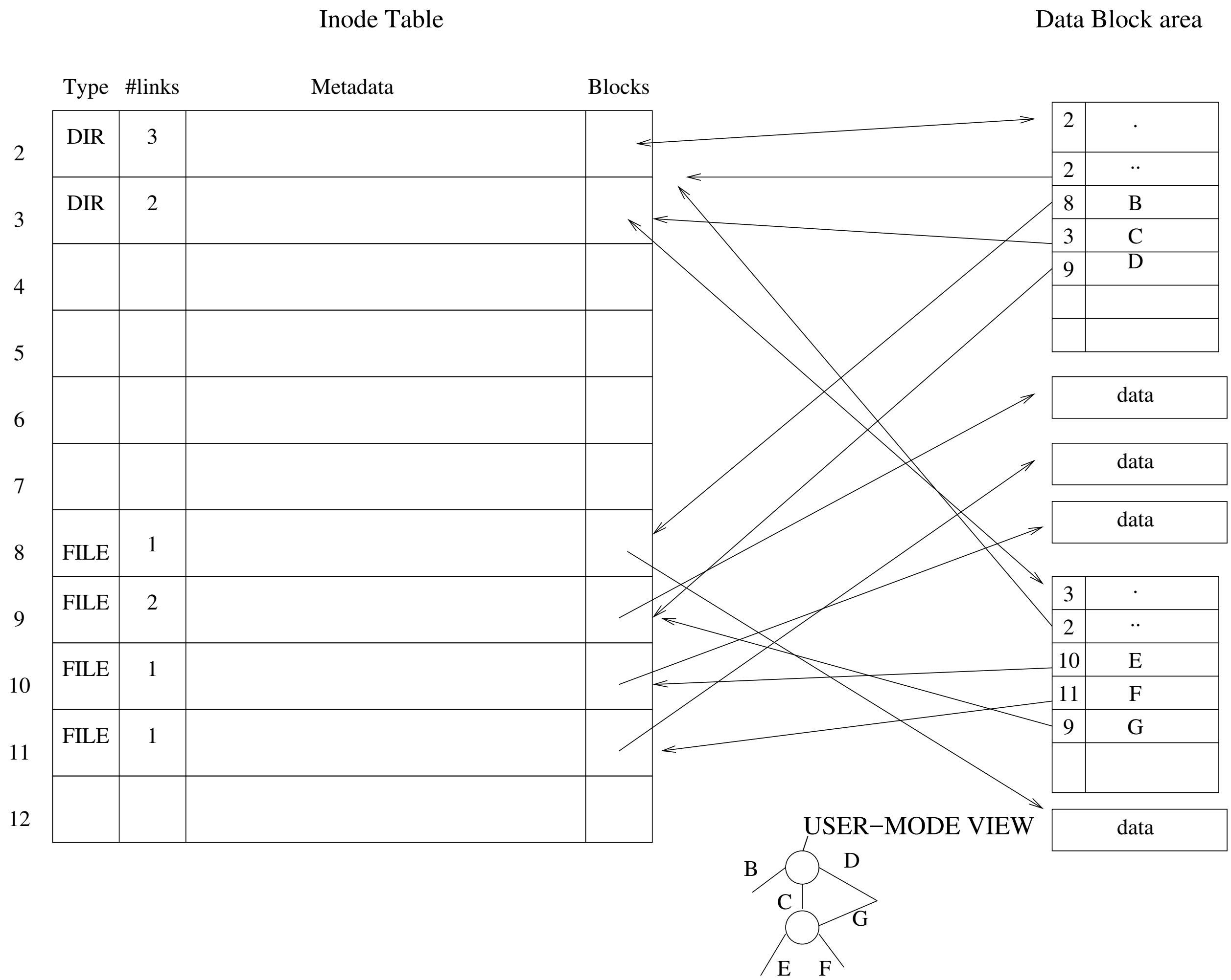
There are two ways to look at this, which are analogous to Hard Links vs Symbolic Links in UNIX. In the latter case, we think of one of the names of the file as being canonical, while all the others are "aliases." In the former case, all of the names are equal in stature.

In the UNIX filesystem, for a given inode number, there can be multiple directory entries throughout the volume which refer to that inode number. In other words, there can be multiple paths (above and beyond those created syntactically by the use of . .. and multiple slashes) that resolve to the same inode.

The system call `link` creates a hard link. Consider:

```
link("/D","/C/G");
```

This creates another name ("/C/G") for an existing file ("/D"). The existing file must actually exist and the new name must not already exist prior to the `link` system call. Here is the situation after executing this call:



Once the link has been completed, the "old" and the "new" names are indistinguishable. There is no way to learn which was in fact the old name and which the new. This is a by-product of the UNIX philosophy which de-couples the name of the file from the actual file.

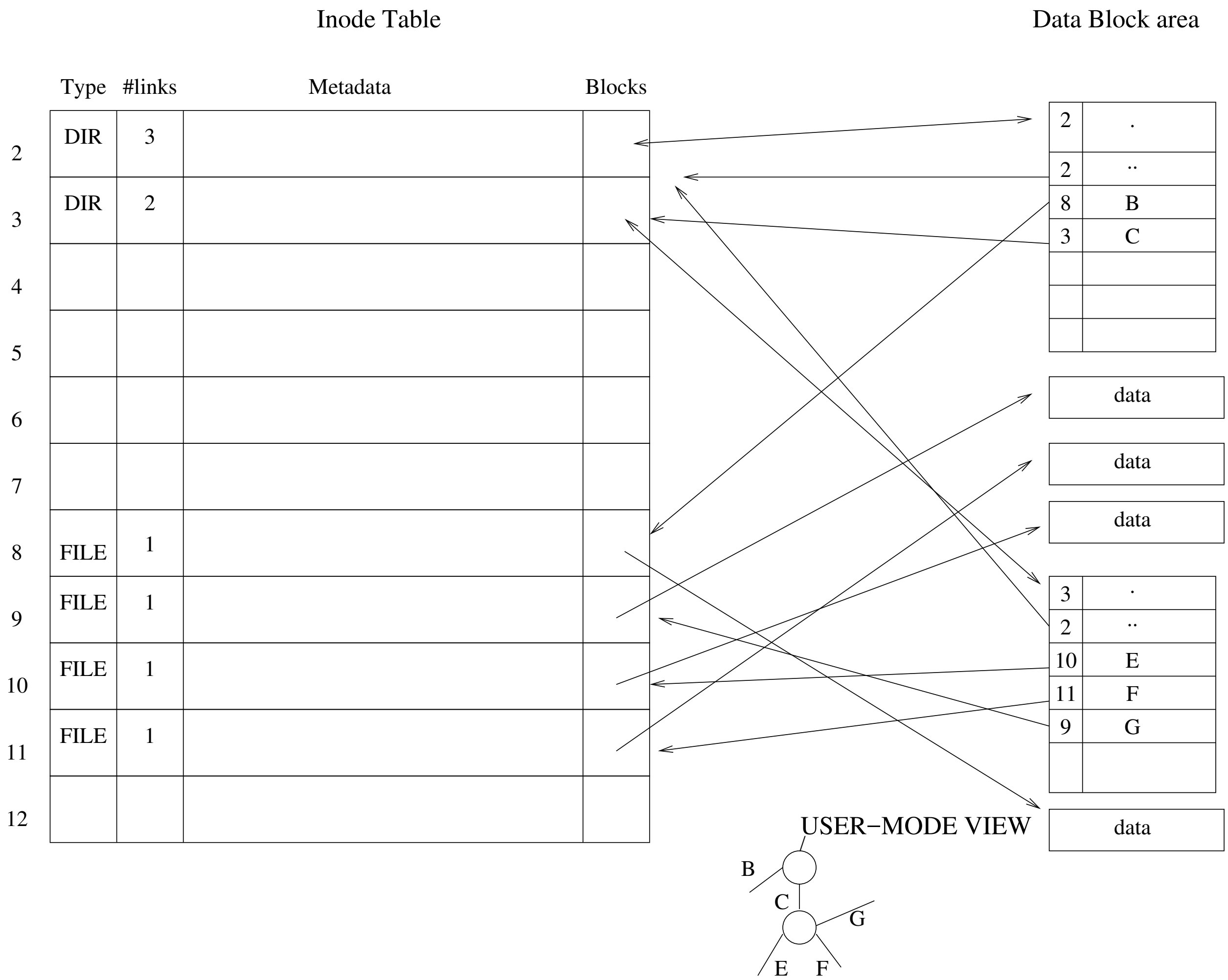
Notice that the inode #9 field *links* has gone up. The operating system must maintain this counter in order to be aware of how many paths exist in the filesystem which point to this inode. We could now remove the pathname "/D", by executing the UNIX command:

```
rm /D
```

This command will, in turn, execute the underlying `unlink` system call:

```
unlink("/D");
```

After which, the filesystem looks like this:



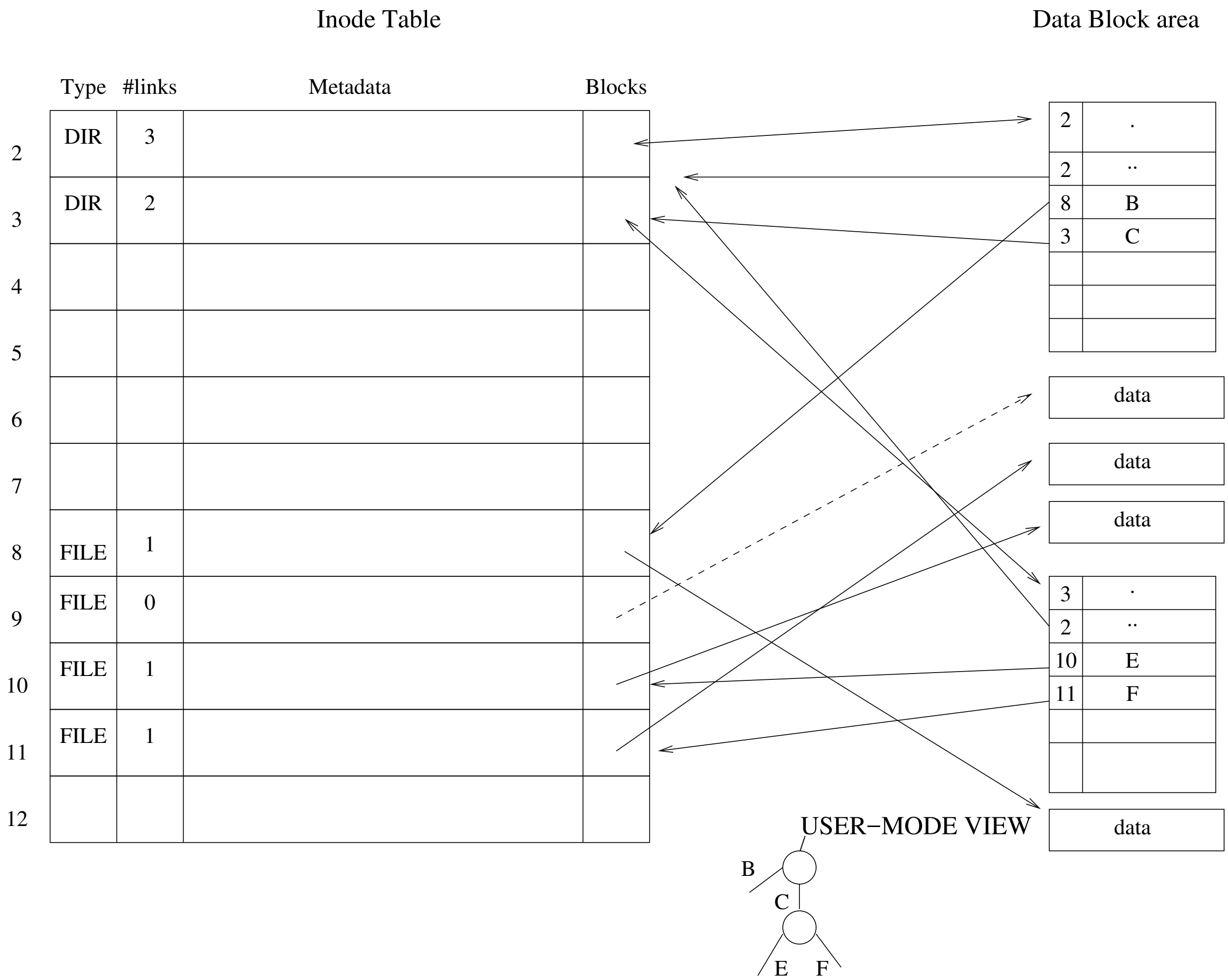
Even though we have issued the **rem**ove command, the actual file represented by inode #9 has not been removed! There is still another pathname which refers to this file ("/C/G"). The file will not actually be deleted until its link count falls to 0, i.e. there are no more pathnames that point to the file. This is why there is an `unlink` system call in UNIX, but no `remove`, `delete`, `erase` or similarly named system call.

File Deletion

Let us execute:

```
unlink("/C/G");
```

The file system then looks like:



Now there are no pathnames pointing to inode #9. There is no way to access the file ever again. Its link count falls to 0 and it is considered deleted. [Linux calls these "orphan" inodes] Its data blocks are marked as free in the free map and the inode structure itself is marked as free and allowed to be re-used when needed. Note that the deletion of the file does not destroy the actual data, but merely marks those data blocks as free for subsequent use. This can be a security exposure if the file contains sensitive data. [Because the data blocks pointers in the inode are not erased when the link count reaches 0, an unlinked file could be recovered intact if one knows the inode number (or can find it by searching through all of the deleted files for a given pattern). There is no system call to do this, but using a byte-level filesystem editor such as `debugfs` is is fairly easy. However, if the volume has had significant write activity since the accidental deletion, it becomes increasingly likely that either the inode has been re-used or the data blocks have been partially or entirely re-used. This would depend on how much writing has taken place and how tight the volume is on free inodes and free data blocks.]

BUT, if any process happened to have the file controlled by inode #9 open at the time, the resources would not be freed, and the file would continue to be active, until that process exits or closes the file. This leads to the interesting phenomenon of "ghost files" under UNIX, in which files continue to exist and consume disk space, but can not be opened new. There are some systems programming tricks which exploit this, e.g. the ability to have a temporary working file which is guaranteed to disappear when the process exits.

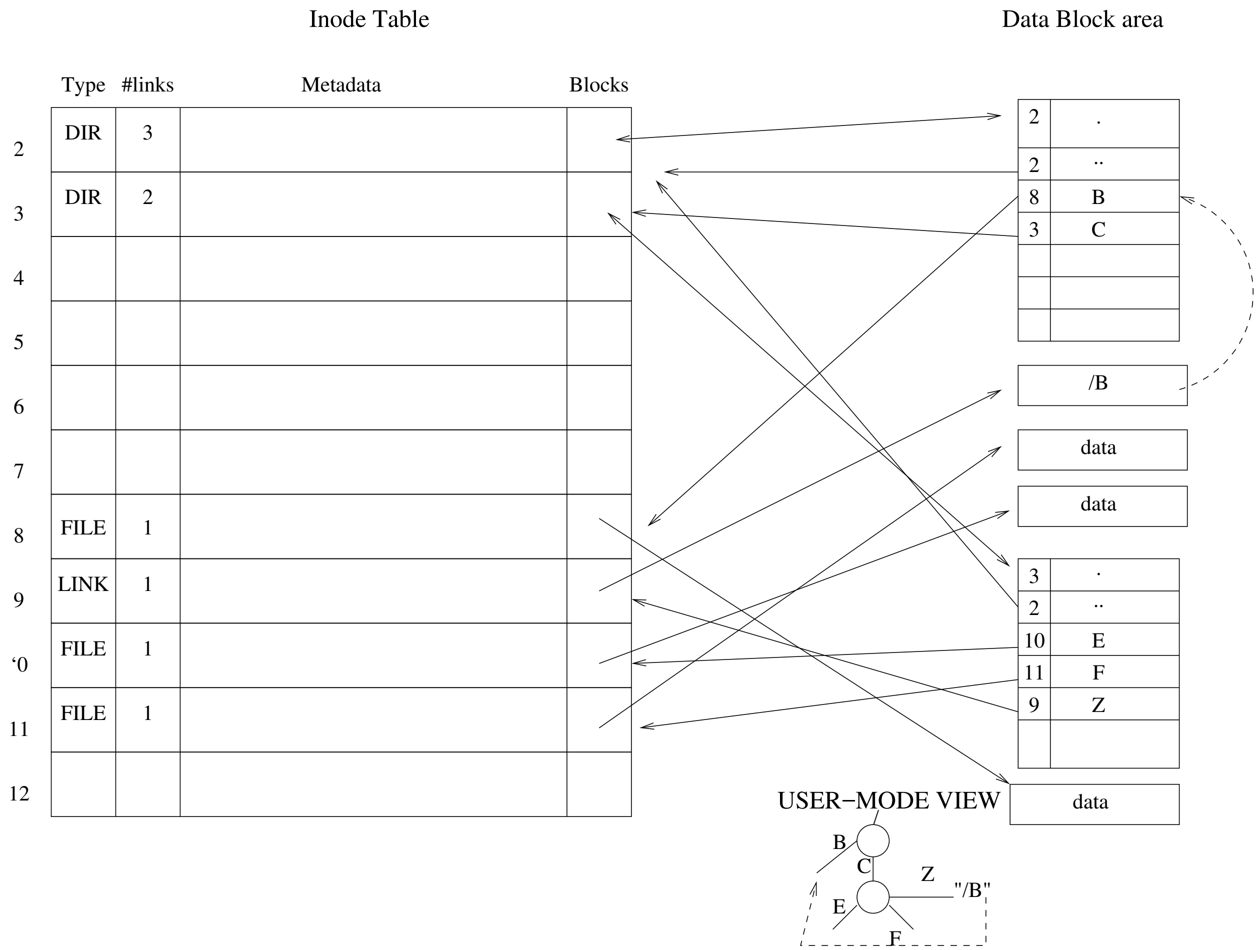
Symbolic Links

Symbolic links, aka soft links, aka symlinks, was a kluge-on feature added to the UNIX filesystem. All modern UNIX variants support it. Unlike a hard link, a symlink is asymmetrical. There is a definite a link and a definite target. The symlink is merely another inode type (S_IFLNK). The data associated with that inode form a string that is substituted into the pathname when the link is traversed. A symlink is created with the `symlink` system call:

```
symlink("/B","/C/G");
```

The first argument is the target of the link (the existing node), the second is a path name which will be created as the symlink. Note that the "existing" name need not presently exist; it is permissible and even useful to make a symbolic link to a non-existent target. The symlink can start with a leading / in which case it is absolute, otherwise it is relative to the symlink inode's place in the filesystem tree. It is a common idiom to have symlinks with relative pathnames, such as "../../bin/foo"

After executing the system call above, our filesystem looks like this:



Most system calls "follow" symlinks, i.e. they transparently substitute the contents of the link into the pathname and continue traversal. (`open(2)` follows symlinks, unless the flag `O_NOFOLLOW` is given.) `unlink` does not follow symlinks. An attempt to `unlink` a node which is a symlink will cause that symlink to be deleted, but will not have any effect on the target. Also note that no count is kept on the number of symlinks pointing to a particular target. That's why it's a soft link. It is possible to create a circularity of symlinks. This will not be detected until an attempt is made to traverse this loop, at which point the operating system will give an error ELOOP. Most UNIX-like kernels use a fairly dumb algorithm for symlink loop detection which places a static limit on the number of symlink expansions allowed at path evaluation time.

To retrieve the metadata associated with the symlink itself, without following it, use the `lstat` system call, which is identical to `stat` except it does not follow symlinks. To retrieve the text value of the symlink (e.g. `/B`), use the `readlink` system call.

Symlinks are useful when it is desired to preserve the distinction between the "real" file and its "alias". Most other operating systems provide an equivalent mechanism. In fact, the hard link is fairly unique to UNIX. A restriction of the hard link, which a symlink overcomes, is that it is not possible to make a hard link across **volumes**. The reason for this will become clear very shortly.

Directories and link counts

A new directory is created with the mkdir system call:

```
mkdir("/foo/bar",0755);
```

The second parameter is the **mode**, or permissions mask, which we will explore below. `mkdir` creates an empty directory with but two entries, "." and ".." In traversing a UNIX pathname, the component "." refers to the same directory, and ".." refers to the parent directory. For historical reasons having to do with simplifying the path name evaluation routine in the kernel, every directory contains an entry for "." and an entry for ".."

Therefore, an empty directory has a link count of 2: one link is the "." entry of the directory itself, the other is the entry in the parent directory pointing to child directory. Whenever a subdirectory is created, the ".." entry of the subdirectory effects a link back to the parent directory and increments the parent directory's link count by 1.

Probably one of the first problems discovered with hierarchical filesystems is that inadvertent removal of a directory will leave dangling and stranded all subdirectories and files beneath the removed directory, thus having the supplementary effect of recursively removing all of these nodes! Therefore, the `unlink` system call is not valid for directory inodes, and will fail returning the error `EISDIR`. A separate system call is provided:

```
rmdir("/foo/bar");
```

In order for `rmdir` to succeed, the target must be an empty directory, i.e. it must only contain the entries "." and ".." If not, the error `EEXIST` will be returned. To remove a populated directory, one must first explicitly unlink all of the children of that directory (which may involve recursion), then remove the directory itself, thus forestalling the cries of "oops, I didn't mean to do that!"

Under some versions of UNIX, it was possible to create a hard-link to a directory, but because of the potential confusion, such behavior is strongly discouraged and is disallowed by most modern UNIX kernels. Exploring the amusing consequences of hard-linked directories is left as an exercise to the reader.

Reading directories

Directories, as we have seen, are special files that equate path component names to inode numbers. Directories can be read using a set of standard C library functions:

```
#include <dirent.h>


r(char *fn)
{
 DIR *dirp;
 struct dirent *de;
        if (!(dirp=opendir(fn)))
        {
                fprintf("Can not open directory %s:%s\\n",fn,strerror(errno));
                return;
        }
        while (de=readdir(dirp))
        {
                printf("%s\n",de->d_name);
        }
        closedir(dirp);
}
```

More information about these calls can be gleaned from the man pages. These functions are in section 3 of the man pages as they are technically not system calls. Just as `fopen(3)` is a stdio library layer on top of the `open(2)` system call, `readdir(3)` is an abstraction of the `getdents(2)` system call. Since the behavior of `getdents(2)` is awkward and non-portable, the use of readdir is preferred in all directory scanning applications. The use of the `readdir(3)` family of calls isolates the application from implementation-specific details of directory structure.

The stat system call

Metadata are informational data about a file, directory or other object in the filesystem, distinct from the data, i.e. contents of that object. UNIX provides the `stat` and `fstat` system calls to retrieve the metadata of a node:

```
#include <sys/types.h>
#include <sys/stat.h>

struct stat st;
int fd;

stat("/path/name",&st);                // This does NOT open /path/name
fd=open("/foo/bar",O_RDONLY);
fstat(fd,&st);
```

The stat structure provides the following information:

```
struct stat {
          dev_t              st_dev;
          ino_t              st_ino;
          umode_t            st_mode;
          nlink_t            st_nlink;
          uid_t              st_uid;
          gid_t              st_gid;
          dev_t              st_rdev;
          off_t              st_size;
          blksize_t st_blksize;
          blkcnt_t  st_blocks;
          time_t             st_atime;
          time_t             st_mtime;
          time_t             st_ctime;
}
```

Most of these fields are stored in the metadata section of the on-disk inode data structure.

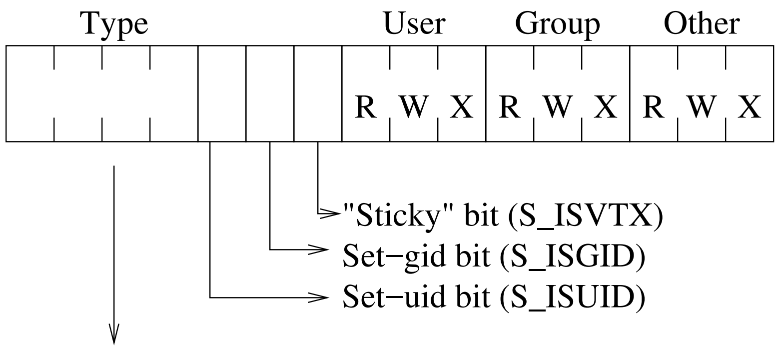
• `st_mode`: The inode type and permissions (see below)
• `st_nlink`: Number of pathnames linking to inode
• `st_uid`: The user id which owns the inode
• `st_gid`: The group owner of the inode
• `st_rdev`: "Raw" Device number (character and block device special inodes only)
• `st_size`: The size of the data, in bytes, if applicable (some inode types do not have a size, such as device inodes). The `st_size` is one greater than the byte position of the last byte in the file. However, it is possible in UNIX to have sparse files, e.g. bytes 0 and 65536 have been written, but all contents in between are undefined. Undefined areas do not occupy storage space and return 0 when read.
• `st_blocks`: The number of blocks of storage space occupied by the file, measured in units of 512 byte sectors. It will generally be a little larger than `st_size` because of the granularity of disk block allocation. However, for sparse files, disk space consumption may be less than `st_size`.
• `st_blksize`: The "best" buffer size to use with operations on this inode. This is

generally the filesystem allocation disk block size.

• `st_atime, st_mtime, st_ctime`: There are 3 timestamps contained within the inode. Each is a UNIX Time, i.e. the number of seconds since midnight January 1, 1970 UTC. [On some filesystems, higher-resolution timestamps are available] The `st_mtime` is the last time a write operation was performed to the *contents* of the file or directory. `st_atime` is the time of the last read operation. `st_ctime` gets touched whenever one of the *metadata* are modified. The `utime` system call can be used to directly modify the atime and mtime stamps, but the ctime field can not be changed. The `touch` command uses the `utime` system call to update timestamps.

`st_blksize`, `st_dev` and `st_ino` do not appear anywhere in the on-disk inode. They are are added by the operating system. The blocksize is a property of the particular volume in which this inode appears. `st_ino` is the inode number, which is inferred by the inode's position in the inode table. `st_dev` identifies the volume on which this inode resides. This will be discussed later in this unit.

The `st_mode` field is a 16 bit bitmask, as follows:



0000	Deleted/Free inode	
0001	Named pipe (FIFO)	(S_IFIFO)
0010	Character Device	(S_IFCHR)
0100	Directory	(S_IFDIR)
0110	Block Device	(S_IFBLK)
1000	Regular File	(S_IFREG)
1010	Symlink	(S_IFLNK)
1100	Network/IPC socket	(S_IFSOCK)

The top nybble of the mode field identifies the type of node. Macros are provided in <sys/stat.h> to give symbolic names to these types:

```
if ((st.st_mode&S_IFMT)==S_IFDIR)
{
        printf("Directory\n");
}
```

Inode Types

There are 15 possible inode types. Inode type 0 is generally reserved to mark a free or deleted inode. The seven inode types depicted in the figure above represent the major, universal types.

• S_IFREG (type==8): A regular file.
• S_IFDIR (type==4): A directory.
• S_IFLNK (type==10): A symbolic link.
• S_IFCHR (type==2) and
 S_IFBLK (type==6): UNIX gives names to devices and provides access to them through the filesystem. E.g. `/dev/sda1` is a file-like node in the filesystem which provides direct access to the first partition of the first hard drive. Within the kernel, devices are identified by an integer device number. The Character Special and Block Special inode types provide a mapping from a pathname to a device number, using the `st_rdev` field. This will all be discussed in a subsequent unit.
S_IFIFO (type==1): A FIFO or "pipe". To be discussed in a later unit.

• S_IFSOCK (type==12): A networking socket. To be discussed in a later unit.

In addition to these, some inode types are/were used only in certain variants of UNIX, and are considered non-portable:
• S_IFMPC (type=3): Obsolete multiplexed character special device
• S_IFNAM (type=5): Obsolete XENIX named file
• S_IFMPB (type=7): Obsolete multiplexed block special device.
• S_IFCMP (type=9): Compressed file, proprietary to Veritas, or "network special" file on proprietary HP-UX operating system.
• S_IFSHAD (type=B): Shadow inode for ACL extensions under some versions of Solaris. Never seen by user-mode programs.
• S_IFDOOR (type=D): Proprietary IPC mechanism under Solaris.
• S_IFWHT (type=E): "Whiteout". An obscure topic which falls outside of the traditional filesystem model, and comes into play with "union mounts".
• S_IFPORT (type=E): Solaris (version 10 and higher) uses this type for an "event port", which provides a way for a process to place watchpoints on various system events or filesystem accesses.

The "sticky bit" S_ISVTX was historically used as a hint to the virtual memory system but now has a different meaning associated with directories (see below). The set-uid and set-gid bits, when applied to executable files, cause the effective user or group id to be changed. This allows a non-superuser to gain controlled access to superuser powers through specific commands, and will be covered in a later unit on security models. The set-gid bit is also used, on non-executable files, to indicate that file and record locking should be strictly enforced. This subject is beyond the scope of this introduction. Additionally, when the set-gid bit is set for a directory, nodes created in that directory take the group ownership associated with that directory, rather than the gid of the running process.

The remaining 9 bits determine the permissions associated with the node.

The UNIX file permissions model

Every user of the UNIX operating system has an integer **user id**. For the purposes of group collaboration, users may also be lumped into groups identified by an integer **group id**. Historically, uids and gids are 16 bit numbers. Each running program, or **process**, has associated with it the user id of the invoking user, the group id of the user's primary group, and a list of groups (including the primary group) to which the user belongs.

Every inode has an individual owner, `st_uid` and a group owner `st_gid`. This ownership is established when the node is first created. The uid of the node when created is the (effective) uid of the process, and the gid is the (effective) primary gid of the process (but see below about the set-gid bit and directories).

When a system call operation requires checking of filesystem permissions, the first step is to determine which of the 3 sets of 3-bit permissions bit masks to extract from the `st_mode` field:
• If the user attempting an operation matches the owner of the file, the user portion of the permissions mask is checked.
• Otherwise, if the group ownership of the file is among the list of groups to which the current process belongs, the group portion of the mask is checked.
• Otherwise, the "other" portion is used.
• Once the appropriate mask is selected, the read, write or execute bit is consulted based on the operation being attempted.
• For files, permissions are checked once, when the file is opened or execution of the file is attempted. If the file is being opened O_RDONLY, read permission must be present. If the file is being opened O_WRONLY, write permission must be present, and in the case of O_RDWR, both permissions are needed. Once the file is opened successfully, changing the permissions on the file has no effect on programs that already have the file open. Execute permission is checked when one attempts to use a file as an executable program (e.g. a.out). This will be covered in a subsequent unit.
• What should write permission for a directory mean? Being able to write to a directory implies the ability to create new directory entries, or to modify or remove existing ones. I.e. directory write permission allows creation, renaming, or unlinking of the nodes within. This original interpretation was found to be problematic in shared directories, (such as /tmp which is generally 777 mode) in that another user might be able to delete a file which s/he did not own. The presence of the "sticky bit" in the permissions mode modifies the semantics of writable directories such that only the owner of a file can rename or unlink it.
• Read permission on a directory implies the ability to search the directory and learn the contents.
• Execute permission is the ability to traverse the directory, that is, to reference an element of the directory. One can have execute permission but not read permission on a directory, allowing one to access files or subdirectories as long as their name is known.

The uid and gid of a node can be changed (at the same time) with the `chown` system call. The permissions of a file can be changed with the `chmod` system call. In both cases, the user attempting the operation must already be the owner of the file (uids match). Furthermore, on most UNIX systems, to avoid problematic interactions with quotas, file "giveaways" are not permitted for ordinary users, i.e. an ordinary user can change the group id of their files but can't change the individual ownership to another user.

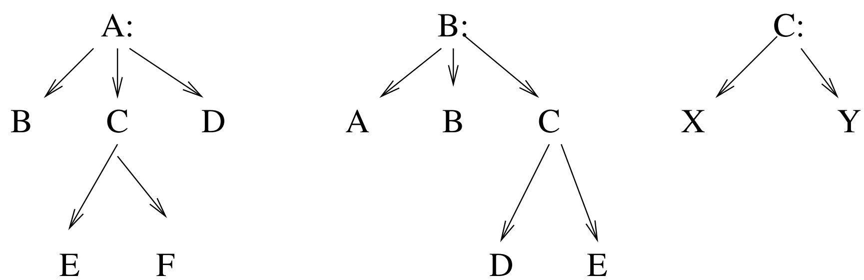
The user with uid 0 is the **superuser**, or **root** account, aka the system administrator. When the process uid is 0, all of these permissions checks are bypassed.

Most UNIX systems support a more elaborate way of expressing filesystem permissions known as **Access Control Lists**. Their application is not widespread because the traditional 3-tiered UNIX permissions model is sufficient for most applications.

Mounted Volumes

Of course a system that supports just a single random-access storage device is not very useful. We have defined a **volume** to be one instance of such a device. Each volume is an independent filesystem data structure which can be detached from the system and attached to another system. Some types of volumes are designed to be removable (e.g. a flash drive) while others require more effort to relocate (e.g. a hard disk).

When a volume is attached to a system and available to users as a file store, it is said to be **mounted**. Many operating systems take the "forest of trees" approach to multiple volumes. For example, Microsoft operating systems such as DOS and Windows assign drive letters starting with A: to each volume:

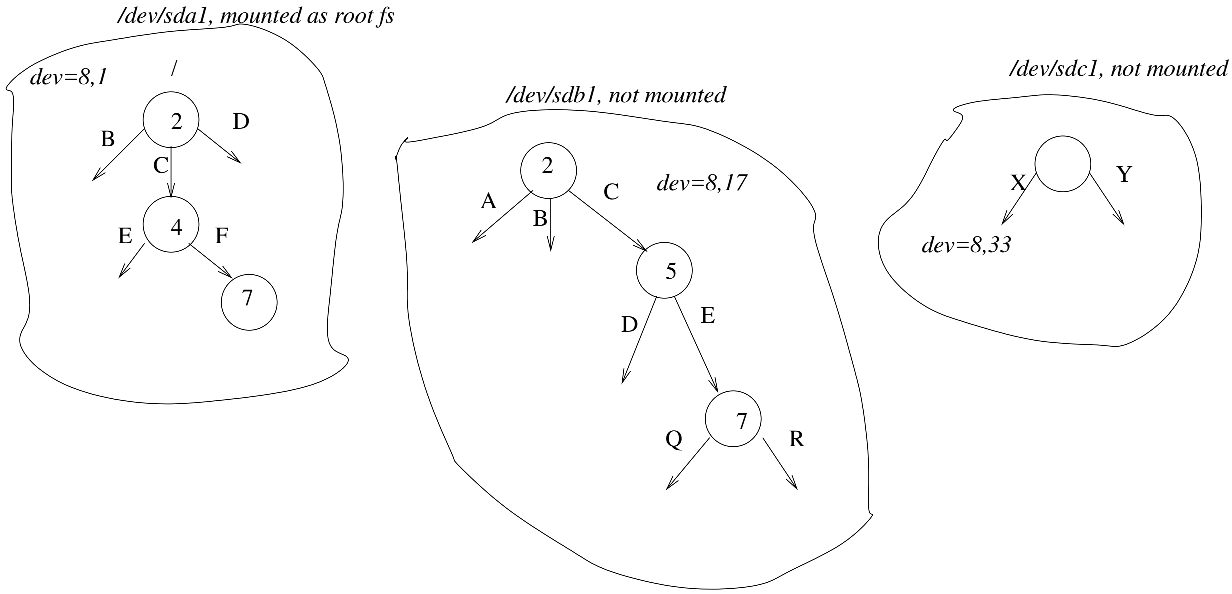


Each volume is an independent tree, and the collection of all such trees forms a flat namespace.

UNIX takes a "big tree" approach:

When a UNIX system first comes up, there is only one volume mounted. This is known as the **root filesystem**. The root of this volume is "/", the root of the entire namespace. Additional volumes get mounted over empty place-holder directories in the root filesystems (or recursively: a volume can be mounted on another volume which is in turn mounted on the root volume, etc.)

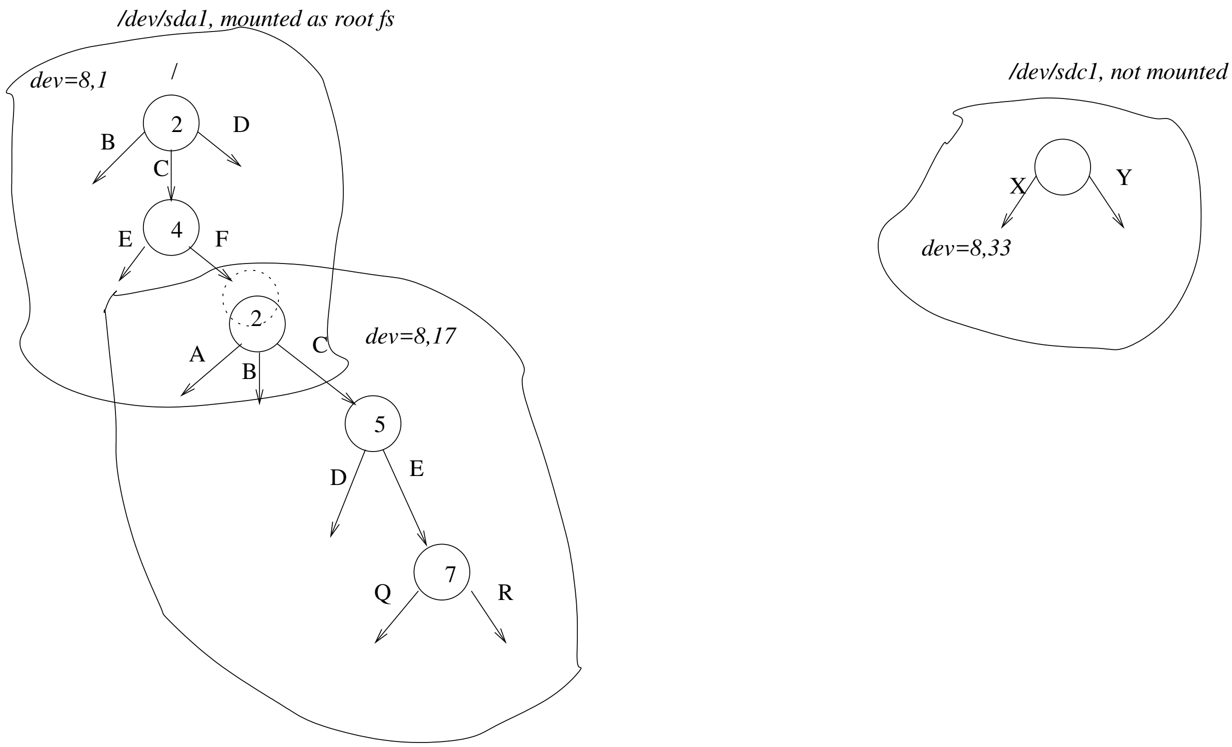
Below we see a system where the root filesystem resides on disk partition `/dev/sda1`. Two other partitions on other drives, `/dev/sdb1` and `/dev/sdc1` are present but are not yet mounted and thus not visible. (For clarity, inode #s for non-directory nodes are omitted)



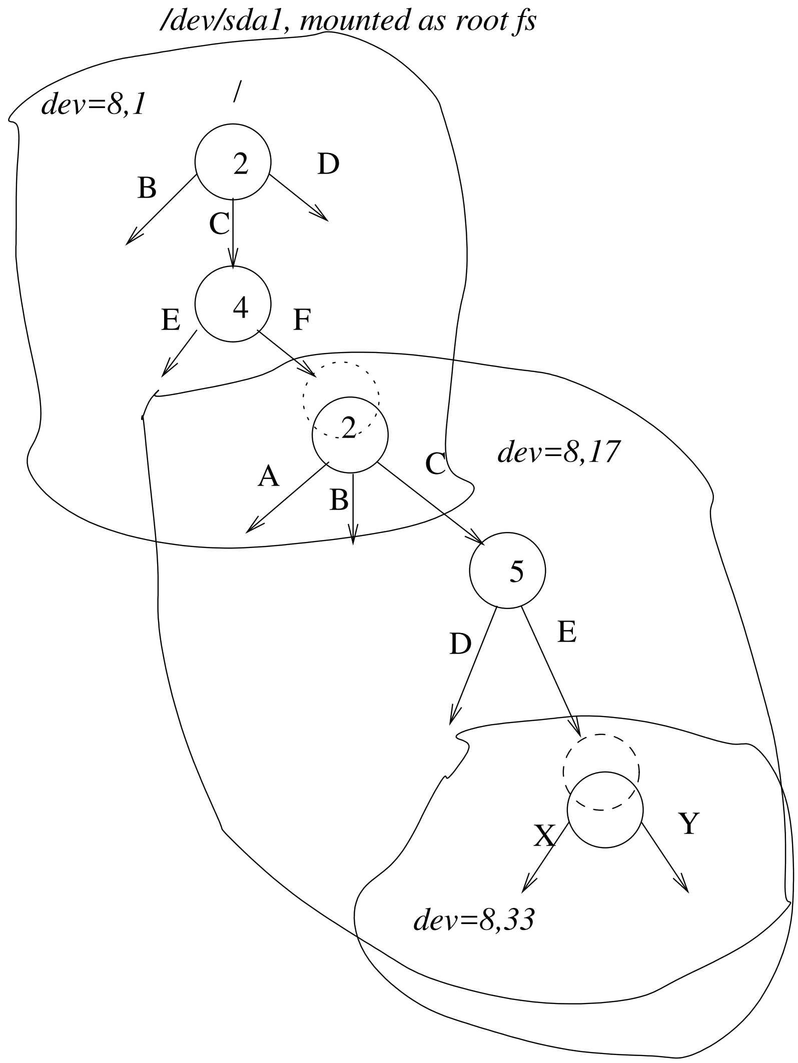
Now we execute the command:
`mount /dev/sdb1 /C/F`

The pathname `/dev/sdb1` is in a special part of the filesystem in which the nodes are device special inodes. Disks are of type S_IFBLK. When the kernel translates this pathname, it produces a major,minor device number pair, e.g. 8,17. This uniquely identifies the first partition of the second SCSI/SATA hard disk on the system.

Pathname /C/F becomes the **mount point**. It was inode #7 in the root filesystem. That inode now becomes obscured, and is replaced with the root inode of the *mounted* volume, which is inode #2:



A mount point must be an existing directory in an already-mounted part of the path name space. Normally, it is an empty directory. But if the directory had contents, they become obscured by the mount:



Here we have performed `mount /dev/sdc1 /C/F/C/E`. The files Q and R, formerly visible through /C/F/C/E (inode #7 in device 8,17) are no longer accessible. If we later `umount /dev/sdc1`, these files will once again be visible.

Note that many UNIX kernels support the concept of a "union" or "overlay" mount in which both the newly mounted and the mounted-over volumes are visible. This feature is useful, e.g., when working off a large read-only volume such as a DVD-ROM while needing to make changes on the fly. We will not be considering this further in this course.

The kernel keeps track of which inodes are being used as mount points, and whenever the path name resolution algorithm within the kernel traverses a mount point inode, that inode is not considered further, but instead is replaced in the traversal by the root inode of the mounted volume. This applies in both downward and upward traversals. Consider what happens if we had entered the directory /C/F/C and then accessed the path "../../E".

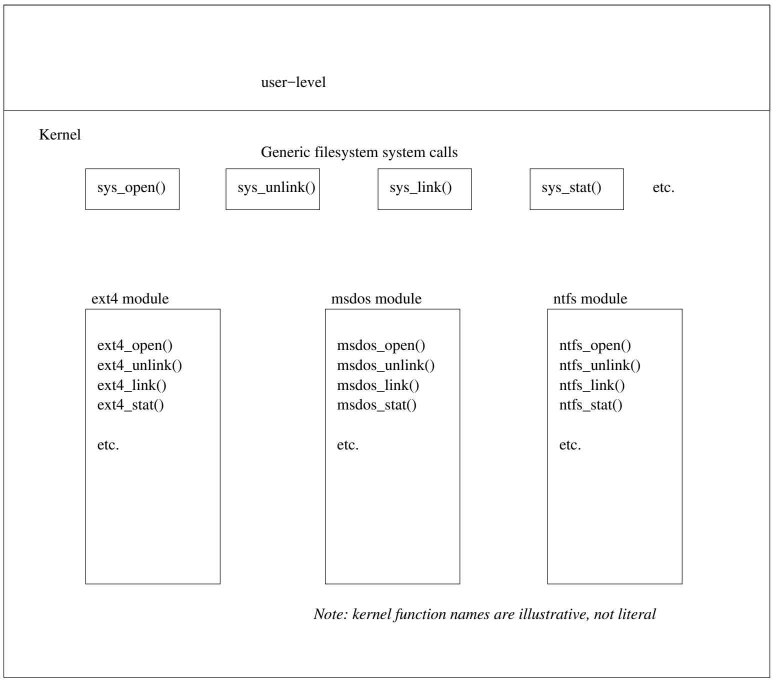
This mounting of a volume onto the existing filesystem hierarchy is not permanent. There is a corresponding `umount` operation, and a corresponding `umount` command, which removes the volume from the filesystem and unveils the original mount point again. In order to unmount a volume, there must not be any open dependencies on it (e.g. a process with an open file, executable or current working directory that is within that volume).

Alien Filesystems / Virtual Filesystem Layer

A given volume need not always be mounted at the same place in the filesystem. When the media containing the volume is moved to a different machine, the volume may even be mounted by a different operating system. Issues can arise, e.g. with byte order, mapping of user identifiers, differences in path name conventions, and other semantics. The UNIX kernel creates an "inode" interface to these alien filesystems. Some features which may be present in other operating systems, such as "resource forks" in traditional Macintosh OS, may not map cleanly to UNIX semantics, and new system calls were added to the traditional UNIX calls to provide access to these "alien" semantics.

The filesystem in UNIX is heterogeneous, i.e. the different volumes need not follow the same data structure. This allows a running system to mount volumes which were created under older versions of the operating systems, or entirely different operating systems. For example, while the current default filesystem for Linux is of the EXT3 type, it is just as easy to use the older EXT2 filesystem type, or the very latest EXT4 type. There are other filesystem types which attempt to optimize for certain situations, e.g. ReiserFS. Linux can also deal with the native filesystem layouts of other UNIX variants, such as UFS (BSD and Solaris), HPFS (HP-UX), and non-UNIX systems such as MSDOS, NTFS, HFS (older Macs), etc. There is support for adding new filesystem types to the kernel dynamically (after the system is already booted), and even to allow a user-level process to implement a filesystem ("FUSE").

This integration of different filesystem types into the overall hierarchy is known as the **Virtual Filesystem** layer of the kernel. Because all file system interface access flows through the kernel, it can keep track of exactly what parts of the naming hierarchy correspond to what type of filesystem, and delegate to the various filesystem modules which are loaded as part of the kernel. Traversal of a mount point is thus transparent to the user. The UNIX pathname space is a flexible one which is independent of the constraints of physical disks.



Pseudo- and Network- filesystems

A mounted filesystem is not necessarily located on a physical disk device on the local machine. This is known as a **Pseudo-filesystem**. **Network filesystems** are pseudo-filesystems which make parts of a filesystem on a remote machine appear local. Two major examples of this are the NFS (the most popular network filesystem for UNIX-like systems) and the SMBFS (the network filesystem used by Windows).

Other pseudo-filesystems provide a filesystem-like interface to things which have nothing to do with files, hard disks and the like. For example, the `procfs` filesystem found under `/proc` in Linux allows one to enter a directory whose name is equal to a process id, and once in there see the processes's memory map, command line invocation, resource usage, and many other interesting things.

Traditionally, mounting of filesystems is a global operation which is performed by a process running as the super-user (uid==0), and all processes see identical namespaces. In some modern variants of UNIX, there are capabilities to associate a different namespace with different processes or users. This can be used to some advantage in securing an application and restricting its access to the filesystem.

Loopback and RAM filesystems

Most UNIX kernels support a "loopback" mount where a regular file that currently exists someplace in the filesystem tree can be treated as a raw disk image. E.g. one can download a `.iso` file which is an image of a CD or DVD-ROM and then mount that file directly, instead of "burning" the CD/DVD.

Most UNIX kernels support a "ramdisk" where an area of RAM is treated as if it were a volume. This is especially prevalent in "live" media, e.g. a CD/DVD or USB stick that contains a "live", bootable Linux system. We need a place for the root filesystem, including places to write things, but we don't necessarily want to rely on existing hard disks. The union or overlay mount technique can be used here to merge an initial filesystem read-only image from the boot medium with the ramdisk. Any changes are simply discarded when the system reboots.

Device Numbers & Inode Numbers

As we have seen, the UNIX kernel associates an integer **device number** with each volume on the system. The device number is not something which would ever be found on the volume itself, rather it is a tracking number maintained by the kernel. The stat field `st_dev` is filled in with the device number of the volume on which the inode in question resides. Like inode numbers, device numbers should be treated as cookies: they can be compared for equality, but no other assumptions should be made about their properties. By examining the `st_dev` field, the user can determine if two paths reside on the same volume. In the example above, stat'ing `/C/F` would yield the `st_dev` device number of the second volume, not the root volume, because the original mount point in the root volume is inaccessible.

Because each volume is a self-contained independent data structure, the inode numbers (`st_ino`) are unique only within the same volume. A consequence of this is that hard links can not be made across volumes, because the inode number in the first volume would have no validity in the second volume. The combination of `st_ino` with `st_dev` uniquely identifies any node within the pathname space at the time that comparison is made. Of course, if volumes are later mounted or unmounted by the system administrator, that might invalidate such a test.

Move/Rename

The `rename` system call:

```
int rename (char *oldpath, char *newpath)
```

is used either to rename or move a file within the same volume(filesystem). The same result could be accomplished by using link(oldpath,newpath) following by unlink(oldpath), and indeed very early versions of UNIX did not have an explicit rename syscall. However, consider what would happen if the process dies (e.g. the user hits Control-C -- we'll talk about signals and process termination in a few more weeks) between the link and unlink. The rename system call is atomic. The mv(1) command uses rename(2). Note: it is not possible to move a file using rename(2) from one volume to another, because it is not possible to make a hard link from one volume to another. The mv(1) command hides this annoyance from the user. If oldpath and newpath are on different volumes, the command instead does the equivalent of `cp oldpath newpath ; rm oldpath`. This is not atomic, instead the mv command also uses the utime, chown, and chmod system calls to make newpath as much as possible have the same metadata as oldpath. Note that since the ctime can not be changed via system call, that will be at least one imperfection.

Is that filesystem, or file system, or filesystem?

Unfortunately the terminology pertaining to the file system is often inconsistent, ambiguous and confusing. In operating system literature, "filesystem" can mean:

• The overall file system, its semantics and interfaces. e.g. "The UNIX filesystem provides a simple, clean interface."

• A particular schema for organizing data within a volume, e.g. "The Reiser filesystem performs better than the EXT2 filesystem when there are many, small files."

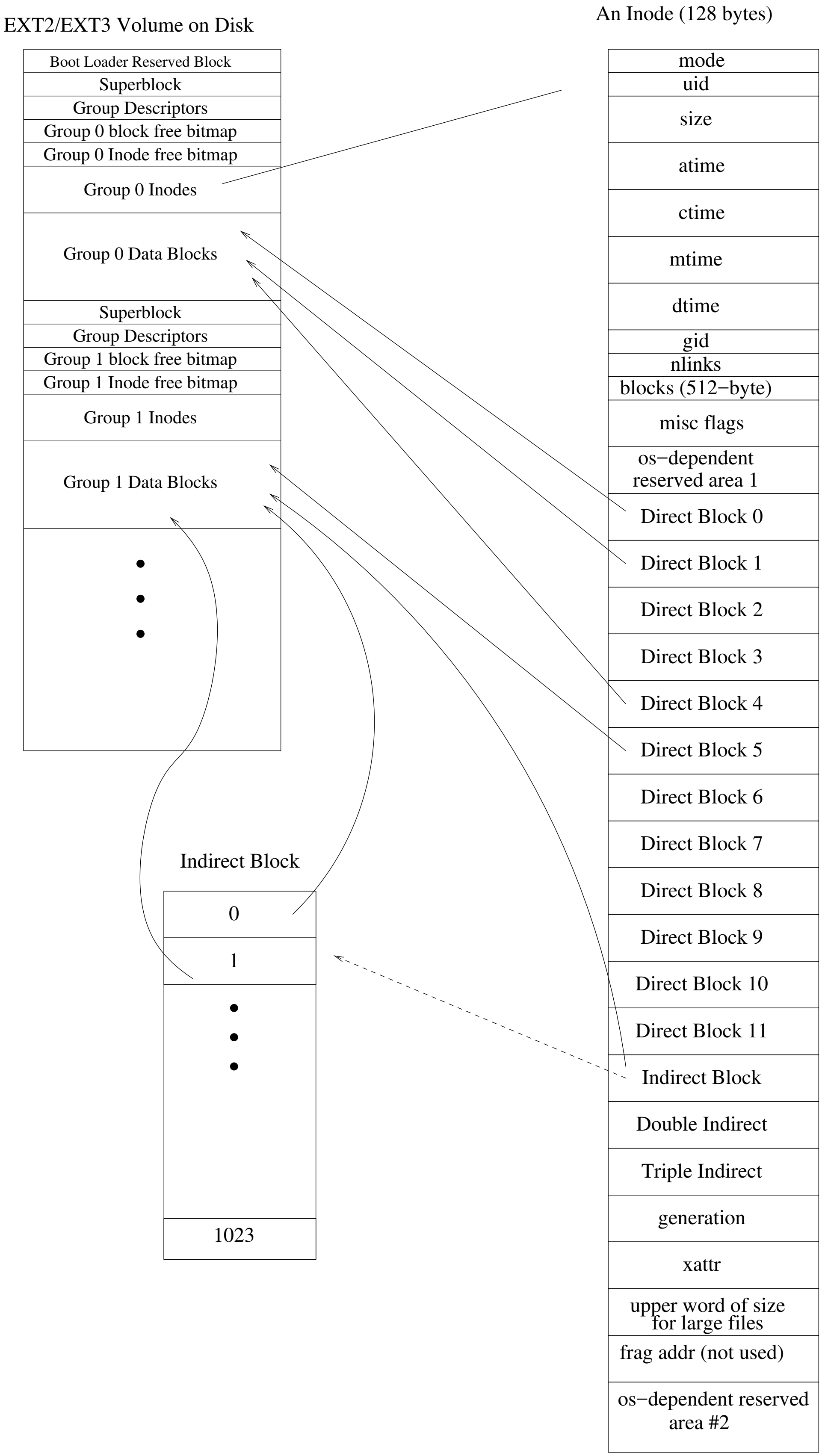
• A code module within the kernel for implementing a filesystem, in the sense of "filesystem" given in the last item.

• A particular instantiation of such an organization of data. We called this a "volume" in these notes, which term is also used in the literature.

Locality and fragmentation

With traditional hard disks that have moving heads, it takes a longer time to access two sectors in a row that are far apart from each other on the disk than if they are close together, and the time grows with distance. Consider the reference model of a UNIX filesystem presented earlier with Header, Inodes, Free Map and Data Blocks. In order to perform an operation on a file, one needs to access the inode, possibly the free map (if the file is being written to) and the data blocks. It would be nice to keep these things close to each other.

All modern UNIX filesystems use **cylinder groups** or **block groups** (these are essentially equivalent terms):


EXT2/EXT3 Volume on Disk
Boot Loader Reserved Block
Superblock
Group Descriptors
Group 0 block free bitmap
Group 0 Inode free bitmap
Group 0 Inodes
Group 0 Data Blocks
Superblock
Group Descriptors
Group 1 block free bitmap
Group 1 Inode free bitmap
Group 1 Inodes
Group 1 Data Blocks
Indirect Block
0
1
1023
An Inode (128 bytes)
mode
uid
size
atime
ctime
mtime
dtime
gid
nlinks
blocks (512−byte)
misc flags
os−dependent reserved area 1
Direct Block 0
Direct Block 1
Direct Block 2
Direct Block 3
Direct Block 4
Direct Block 5
Direct Block 6
Direct Block 7
Direct Block 8
Direct Block 9
Direct Block 10
Direct Block 11
Indirect Block
Double Indirect
Triple Indirect
generation
xattr
upper word of size for large files
frag addr (not used)
os−dependent reserved area #2

Each block or cylinder group is like a mini-filesystem, containing a portion of the overall inode table and data blocks, and a free block bitmap covering just the data blocks in that cylinder group. The kernel will attempt to keep a file allocated all within the same cylinder group. This improves locality and filesystem performance. Inode numbers and block numbers continue to have global meaning, but the inodes and blocks are now spread out among the groups instead of being concentrated together.

You will also notice that Linux uses a second bitmap to keep track of free vs in-use inodes. This improves performance..the inode bitmap often fits within a single disk block, which remains cached in kernel memory anyway. Finding a free inode is a quick in-memory bitmap search, rather than a series of disk accesses to examine each inode to see if it is free.

You may also notice that a space for the volume Superblock appears in each block group. In practice, several replicas of the superblock are stashed in block groups in addition to group #0. Whenever the superblock is flushed to disk, these additional disk writes are also needed, so we don't want to actually replicate the superblock in *every* group. Having replicas means it is still possible to recover the volume if the main superblock gets corrupted or if that sector of the disk goes bad.

Fragmentation is defined as the case where a file's contents are not stored in contiguous data blocks. Fragmentation is impossible to avoid unless one wants to sacrifice a lot of disk space, as with a contiguous filesystem. However, excessive fragmentation will cause a lot of disk seek activity to access a file and is undesirable.

To combat fragmentation, the kernel can avoid allocating space for a new or growing file which is right up against the space of another file. But as the total space used on the volume approaches the total available space, it becomes harder to avoid that. Many system administrators tune filesystems with a **reserve factor** of 5-10%. On a volume with 1GB of data block space and a 10% reserve factor, it would appear from the `df` command that there is only 900MB free. Ordinary users will start to get "disk full" errors when the space in use exceeds 900MB, but in reality 100MB is being held in reserve for system processes (which usually run as "superuser"). With 10% of the volume still free, the kernel is usually able to avoid excessive fragmentation. Volumes that are running below 5% free space typically start to have performance issues from fragmentation.

A filesystem with a lot of fragmentation can be "de-fragmented" if there is sufficient free space. This involves first moving smaller files around on the volume so as to open up larger contiguous regions of free space, to which larger files are moved. This process can take many hours and could potentially cause filesystem corruption if it crashes in the middle. It is best done of volumes that are not currently mounted.

Keeping track of disk blocks in the inode

Inodes which are Directory, Regular File or Symlink store data in the data blocks part of the filesystem. We will now take a peek at how UNIX operating systems translate offsets within a file (or directory or symlink) into disk data block numbers so that the file (or directory or symlink) can be read or written. Traditionally, many UNIX filesystems, including the original (Version 7 UNIX), System V, UFS (BSD) and EXT2/EXT3 have used the following model (which is illustrated several pages back):

There are 12 **direct block** slots which give the block numbers of the first 12 block-sized chunks of the file. E.g. when the filesystem block size is set to 4K (the default on Linux), then slot[0] gives the block number which holds bytes 0..4095 of the file. slot[1] gives bytes 4096..8191, etc.

Now, if the file size exceeds 12*block_size, the **single indirect** block comes into play, which is found at slot[12]. This gives the block number of a block in the data block section, which is marked as allocated in the allocation bitmap, but doesn't actually store any file data. Instead, this block contains an array of block numbers. Let us say that block numbers are represented as 32-bit unsigned integers and the block size is 4K. The first indirect block then maps 1K * 4K = 4M worth of the file, from offset 48K to 48K+4M.

OK, what happens if the file is bigger than 4,243,456 bytes? Now we go on to the **double indirect** block at slot[13]. It contains the block number of a block which contains an array of block numbers, each of which is a single indirect block. Again, if block numbers are 32 bit and blocks are 4K, the double indirect block covers 1K*4M or 4GB of the file.

Twenty years ago, the idea of a file larger than 4GB+4M+48K was ludicrous when a 9GB hard drive cost a thousand dollars. But of course today such large files are common. So we get to slot[14] which is the **triple indirect** block, which is an array of double indirect block numbers, each of them being an array of single indirect block #s, ad nauseam. Keeping in line with our example numbers, the triple indirect block covers 1K*4G=4TB. 4TB+4GB+4M+48K would then seem to be the largest file that could be kept under this representation with 4K blocks.

*Aside: If one searches for "maximum file size" for the older EXT2/EXT3 Linux filesystems, the result is 2TB. This is because there is a field in the on-disk inode called i_blocks which is 32 bits long and which is in units of sectors (512 bytes), not disk blocks. This field describes the number of 512 byte sectors that the file consumes. 2^32 * 2^9 = 2^41 or 2TB. Because this field would overflow beyond 2TB, the kernel prevents a file from exceeding that point. EXT2/EXT3 use 32-bit block numbers, thus with a 4K disk block size, the largest volume (filesystem) which can be created is 16TB. EXT4 addresses some of these limitations.*

Inline file storage

In the Linux EXT2/EXT3/EXT4 implementations, block numbers are 32 bits (4 bytes) and the block map portion of the inode is therefore 60 bytes long. As a further optimization, files which are 60 bytes or less could be stored directly in that 60-byte area and not require any additional disk accesses, beyond the one to retrieve the inode. This optimization is particularly helpful for symlinks which are often very short.

Sparse Allocation

Consider the following code:

```
fd=open("file",O_CREAT|O_TRUNC|O_WRONLY,0666);
write(fd,"X",1);
lseek(fd,16384,SEEK_SET);
write(fd,"Y",1);
```

The file contains an X at offset 0 and a Y at offset 16384. What is in between? Nothing has ever been written there. The philosophy that UNIX takes is that this is a "sparse" file, i.e. a file with a "hole" in it. The data from 1..16383 are not defined, but for the sake of consistency and security, they read back as all-0 bytes. Now, since nothing is really there, do we need to allocate disk data blocks? UNIX does not. In the inode, the [0] slot and the [4] slot would have valid block numbers, but the [1] [2] and [3] slots would have 0 as the block number, indicating that the corresponding 4K region of the file is not allocated.

The field `st_blocks` in the stat structure will report 16 (remember that this field is in units of 512-byte blocks), i.e. only 8K is being allocated to the file. On the other hand, **st_size** will be 16385, which at first glance would seem to require five 4K blocks. Try it at home, folks.

Extent-based allocation

The block map (direct/indirect/double/triple) method has worked very well for many, many years. It is particularly efficient for small files (e.g. under 48K in our example above). However, for larger files, it requires additional disk accesses to get at the indirect blocks. Indirect blocks are cached (see below under "buffer/block cache") but for random-access to a large file in the worst case, it could require several disk accesses per data block access.

In the Linux world, the EXT4 filesystem has moved to extent-based allocation. Other filesystems also use this. An **extent** is defined as a *contiguous* group of disk blocks. Therefore it can be described by (start_file_block_offset, start_disk_block_num, block_count). In the Linux EXT4 implementation, start_file_block_offset is a 32 bit unsigned number, start_disk_block_num is a 48 bit unsigned number, and block_count is effectively a 15 bit number (32768 max blocks). If the block size is 4K, this limits a single file to 16TB and the overall filesystem to approximately 2ˆ60 or 1EB. A single extent descriptor can cover a maximum of 128MB of file space with 4K block sizes.

In this implementation, each extent descriptor is 12 bytes long. EXT4 also uses a 12-byte header for the entire extent data structure. Therefore, within the existing 60-byte area used for the block map, EXT4 can store 4 extent descriptors (4*12=48 + 12 for the header = 60). Now, whereas the block map approach always required a fixed number of 4-byte block numbers to hold the map for a file of a given size, with extent-based allocation, the number of 12-byte extent descriptors is variable depending on the fragmentation of the file. If the file is allocated contiguously, these 4 extent descriptors could map the first 512MB of the file. Extent descriptors are always maintained in sorted order by start_file_block_offset, so a simple binary search will find the descriptor that maps the area of the file that we are interested in.

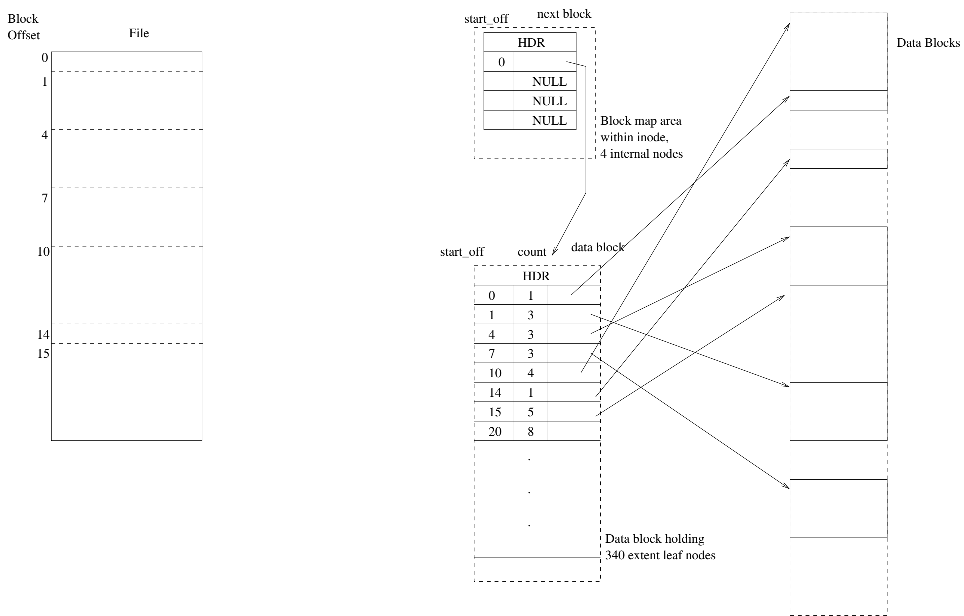
To map larger files and/or if fragmentation requires a greater number of extent descriptors, the extent data structure becomes an odd form of tree which the special property that it is of uniform depth (i.e. at any given moment, the number of internal nodes traversed to reach a leaf node is constant for all leaf nodes). The leaf nodes of this tree are the extent descriptors. A tree of depth 0 is the case where 4 extent descriptors are stored inside the inode. At depth 1, the inode contains 4 internal nodes which contain (start_file_block_offset,next_disk_block). Again, these internal nodes are sorted by start_file_block_offset, but now they contain "pointers" to disk blocks which contain one 12-byte header followed by an array of leaf nodes (extent descriptors) which are sorted by start_file_block_offset. A 4K disk block holds 340 leaf nodes with 4 bytes wasted at the end (these 4 bytes can be used for a checksum to improve filesystem integrity checking).

If the file requires more than 1360 extent descriptors, now we have to move to a 2-level tree. The 4 entries in the inode now each point to disk blocks which contain 340 internal nodes which point to additional disk blocks which finally contain 340 leaf nodes (extent descriptors). Where does it end?

The theoretical worst case is a file that is completely fragmented, such that each extent descriptor has a block_count of just 1. If disk block numbers are 32 bits, we would need 2^32 leaf nodes (of course the filesystem would be useless at this level of fragmentation). A 4-level tree would hold 4*340^3 or 157,216,000 block numbers which is too small, so a 5-level tree is needed which could hold 53,453,440,000 block numbers. This is the maximum tree depth that would be needed, but of course if the file can be mapped in fewer levels, it is.

This data structure would be a poor choice in the general case but for file allocation, once an area of the file has been allocated, the block numbers assigned to that area will generally not change. If the file grows, new extent descriptors can simply be appended, since the previously allocated ones are already in file offset order. When we run out of room to represent the extent tree with the current number of levels, the entire existing tree can remain and a new higher level is created with a pointer to the existing tree.

Extent-based allocation is also able to support sparse allocation. In this case, a flag in the extent descriptor indicates that the corresponding number of blocks represent a "hole", and the start_disk_block_num field is not valid.



Caching

In general, the CPU is faster than RAM and RAM is faster than mass storage. **Caching** is a concept which will arise repeatedly in this course. The basic premise is that if you do something once, you are likely to do it again, or something very much like it or near it, in the near future. Therefore, it usually pays to keep a copy of something which is kept in slower, plentiful, cheaper storage temporarily in faster, scarcer, pricier storage.

When talking about filesystems, caching is going on at multiple levels.
• Individual sectors (or disk blocks) are cached in the **buffer cache**
• The superblock (volume header) is cached by keeping the "live" copy in kernel memory whenever a volume is mounted.
• Individual inodes are cached by keeping a live copy in kernel memory whenever the inode is needed, e.g. when the file is open, during directory traversal, during a `stat` system call, and many other cases.
• Directory entry lookups are cached in the **dentry cache** (Linux terminology, other non-Linux UNIX kernels have called this the namei cache)
• Reads and writes of regular files are cached by keeping copies in kernel memory. We'll also learn in Unit 5 about file/memory equivalencies.

Each of these caches (with the exception of the superblock which is only caching one thing) is, like any cache, an associative array. The cache is addressed by a "key" which results in either a hit, in which case the requested item is in memory, or a "miss" which requires loading it from disk. In some cases, the key is simply the direct "address" of the object (such as a sector number) but in other cases the key is part of the object's data itself (such as directory entry names).

All caches have overflow issues: if there is no room to cache a new object, some older object must be released from the cache, possibly writing it back to the disk. All of these filesystem caches will release all of their in-memory resources and synchronize the data back to disk when the volume is unmounted. We'll now look at each cache briefly.

Buffer Cache

One view of the volume is as just a collection of sectors. The buffer cache (or "block cache") is keyed by (device_num,sector_num) and objects in the cache are the sectors (or "disk blocks" which may be multiple sectors). Since most of the sectors of a filesystem represent more high-level objects, such as inodes, directories or files, the buffer cache is often superceded by the other caches below. The buffer cache does come into play with caching indirect blocks and inode/data block allocation bitmaps.

Inode Cache

Whenever any system call is performed on a node in the filesystem, the kernel needs read (and potentially write) access to the corresponding inode fields. The inode cache is keyed on (device_num,inode_num) and contains "in-core inodes". This data structure, kept in kernel memory, has all of the fields of an on-disk inode, plus numerous other fields that the kernel needs, such as pointers to the correct virtual filesystem modules for the filesystem which contains that inode. Once an inode is brought into kernel memory, it generally is locked there until no longer needed. In particular, this means that any open file has an in-core inode.

Dentry Cache

In the Linux kernel, the dentry cache is keyed on (inode,component_name) where inode is a pointer to the in-core inode which is a directory. Another way of viewing this is that the key is (device_num,inode_num,component_name). Whenever a directory must be searched, such as during path name evaluation, the (device_num,inode_num) of that directory is presented to the dentry cache along with the component_name being searched. Unlike the other caches we have seen, the dentry cache is also a **negative cache**. If there is no dentry object corresponding to the component_name, this results in the filesystem module's directory lookup function being called on the directory inode. How this is implemented is obviously filesystem-dependent (a directory lookup on an

MSDOS/FAT filesystem will be very different from ext4fs). The result of this function call is either the inode number (if the component was found) OR it is nil (not found). In either case, the result must be cached (negative caching).

The dentry cache must be notified whenever a filesystem operation happens that affects a directory entry. I.e. an unlink or rename system call must cause the corresponding cached dentry to be either changed to a nil, or renamed. Therefore the dentry cache is a complex and confusing part of the kernel source code!

File Data Cache

We'll see in Unit #5 that areas of memory and areas of regular files have an equivalency. Prior to that revelation, let us just say that when read and write system calls are made, that causes the corresponding areas of the file to be cached as areas of memory (it turns out these areas are always in 4K chunks, for reasons that will become apparent in Unit 5). Generally speaking, this improves the efficiency of small reads and writes. Rather than having to make say four disk requests to read 4K of data with 1024 buffers in the read system call, just one disk request of 4K is needed. The first read system call takes a little while, but the next three are satisfied from the cached copy in memory, and are much faster.

When writing, it is important to realize that the write system call does not generally result in an immediate write to the disk. Instead, the bytes that are supplied to the write system call are copied into the in-memory image of that area of the file. The write back to the disk happens at a later time.

Cache modes & sync

In general, there are three modes to caching: **uncached** (or "raw") in which all object accesses result in direct and immediate disk accesses, **write-through** where reads are satisfied from cache but writes cause immediate write-through to disk, and **write-back** where the write to disk takes place at some indeterminate later time. Raw mode would generally not be used except for system utility programs. Write-through mode has the advantage of keeping the disk in a consistent state and minimizes the risk of data loss or filesystem corruption. However it causes a large amount of disk write activity. Therefore, most systems are run in write-back mode.

In write-back mode, a system call such as `write` will complete before the data are written to the disk. In fact, the `close` will complete too. The precious data are sitting someplace in kernel memory, awaiting write-back to the disk. If the system crashes (sudden power loss, kernel bug, hardware issue) at this point, the data will be lost, but more importantly, the program (and the end-user) will be under the false impression that the data were saved, because no system call errors were raised.

If this is not acceptable, there are a few solutions (better than turning to write-through caching). The `fsync` system call takes a file descriptor and commands the kernel to flush all data and metadata associated with that file descriptor to the disk. The system call blocks until the disk writes have completed. The file could also be opened with the `O_SYNC` flag, in which case all write system calls will block until the corresponding disk writes have completed. Finally, there is a `sync` system call and a corresponding user-level command which causes ALL cached data to be written back to disk.

Outside of these explicit flushes, the Linux kernel periodically scans memory for cached objects and flushes them to disk. The rate at which this is done depends on system load, but generally on an idle system, flushes will be done at least once a minute. A system that is quiet and crashes, therefore, is not likely to lose any data. In addition, when a volume is unmounted, all cached objects are flushed, and the `umount` system call blocks until these disk writes are complete. This is particularly important for removable storage devices -- it ensures that once you give the `umount` command and get your shell prompt back, it is safe to remove the device!

Filesystem corruption, recovery & journaling

The filesystem is a data structure built out of disk blocks. As with any complex data structure, there are transient moments when it can be in an inconsistent state. Consider how the kernel conceptually handles the creation of a new file:

Step 1) Search the inode allocation bitmap for a free inode, say we pick #9999
Step 2) Mark the inode as non-free by changing its type field to S_IFREG (and also set the uid, gid and ctime fields, and set nlink to 1)
Step 3) Mark the inode non-free in the inode bitmap
Step 4) Search the directory containing the new file for a free slot
Step 5) Write the path component name and inode number 9999 to the directory slot

Let us say that somewhere between steps 2 and 5, the system loses power and therefore the directory is not updated. When the system comes back up and the volume is mounted again, we now have a "phantom" inode which is not free but is also not linked anywhere in the filesystem. This is just one example of many kinds of corruption which can arise from the underlying problem that filesystem operations reduce to a **non-atomic** series of writes to different disk blocks. The filesystem corruption problem is different from the cache consistency problem : filesystem corruption could occur even if there were no caching and all filesystem blocks were written through immediately.

When the system comes back up after a crash, there must be a way of determining if a filesystem (volume) is corrupt, and if so, to correct before mounting. Otherwise, the corruption could lead to further corruption and data loss. The first step is that all UNIX filesystems maintain a flag in their superblock known as the "clean" status. When the volume is dismounted properly, using the `umount` system call/command, or automatically as part of a controlled system shutdown, the superblock on disk is written to and the CLEAN status is set. While the volume is mounted, the status is set to DIRTY.

If the system crashes, the volumes will not be explicitly unmounted, and the status of the on-disk superblock will be DIRTY. The traditional way of correcting the corruption is via a user-level program known as `fsck` which examines the on-disk filesystem on a block-by-block basis. It can do this since the raw contents of the volume are available via a special file name, e.g. `/dev/sda1`. fsck is a complex, multi-pass program. It does a recursive descent exploration of the pathname tree, visiting each node, and cross-checking that view with the view represented by the inode table and free block map. If a consistency error is detected, fsck prompts before taking action to correct it. I.e. fsck is a read-only operation unless the user invoking it gives permission for writing. In batch mode, e.g. when the system is being booted, fsck is normally run in a mode where it will always attempt to fix problems, unless certain serious problems are detected, or the number of problems is abnormally high, at which point the booting process will stop pending system administrator interaction.

Often fsck will find inodes that might represent valid files, or might be invalid or deleted files. Because the filesystem is inconsistent, it has no way of knowing for sure. These inodes are landed in a pre-existing subdirectory of the root directory of the volume, called `lost+found`.

Filesystem Journal

fsck can be a very time-consuming process, since it needs to visit every part of the filesystem data structure, including parts that correspond to unallocated space, e.g. unused inodes. As hard disk capacities have increased, this has caused fsck times to be unacceptably long. Most modern UNIX systems have gone to journaling to address this.

A **journal** is a circular array of blocks somewhere in the filesystem, generally at a fixed location and generally contiguous. The journal contains **journal entries** which are ordered from oldest to newest. Since the journal is of finite size, eventually newer entries overwrite older ones. (Linux by default uses a journal size of 128MB) That may cause operations to hang until disk write flushes complete. While there are many different ways of implementing a journal, they all accomplish a similar goal: they make it possible to quickly recover a corrupted filesystem without having to use the very slow `fsck`. They do this at the expense of additional disk write traffic. Blocks that are critical enough to deserve journaling are written twice: once to the journal, and once again at a later time when the applicable cache is flushed.

Here is an example of hypothetical entries in a journal for our previous example of creating a new file:

```
--BEGIN TRANSACTION ID#1234--
Copy of disk block containing inode #9999
Copy of disk block containing entry 9999 in free inode bitmap
Copy of disk block containing new directory entry referring to ino9999
***** potentially other, unrelated transactions ******
--COMMIT TRANSACTION ID #1234--
```

The kernel, when journaling is in use, first assembles all of the elements of the atomic transaction and writes them to the journal. Then, only after the COMMIT TRANSACTION record has been written to the journal, the kernel schedules the various disk blocks to be flushed (sync'd) to disk. This might happen a "long time" later (several seconds) during which time additional journal entries might be written for other, unrelated transactions. Depending on kernel settings, the kernel might cause the system call (e.g. `creat`) to block until the COMMIT TRANSACTION record has been written. This ensures that when the system call returns, the operation will actually happen, even if the system crashes.

When the volume is mounted after a system crash and is found to be DIRTY, rather than invoking `fsck`, the journal is examined in order from oldest to newest entry. For each transaction ID number mentioned, there are two possible outcomes:

1) The BEGIN entry is there but no COMMIT. This means that the system crashed while the transaction record was being written to the journal. Since there is no COMMIT entry, we don't know if we have the entire transaction. No valid action can be taken. Since disk flushes don't happen until the COMMIT entry is written, no partial transaction was written to the disk. In other words, this transaction never happened.

2) The BEGIN and COMMIT entries are both there. This transaction is then "replayed", i.e. the copies of blocks for the transaction that are in the journal are written to their places in the filesystem. It does not matter that they may have already been written. There is no harm in writing the same data again, other than a slight delay.

Recovery via the journal is very, very fast, typically less than one second!

Journal modes

In Linux, a particular mounted volume which supports journaling can be set to one of three modes (this is configured at each mount and is not a permanent attribute of the volume) which controls how the data itself (as opposed to metadata) are handled in the journal:

• `data=journal`: In this, the most paranoid mode, all data writes are written into the journal first, i.e. the data have the same level of protection as the metadata. This can have a very heavy toll on performance since all writes effectively require two disk accesses (one to the journal, the other to the actual data block) and because data write traffic is

high-volume and may overflow the small journal area causing a stall until disk writes are completed. This mode guarantees that the order in which data are written to a file is consistent with the order in which files are created, renamed or deleted.
• `data=ordered`: Data are written back before corresponding metadata are written into the journal. The data blocks themselves are not written to the journal. This is the default mode. Consider a file that is being written with new data. As it grows, its size field in the inode has to increase, the block maps have to be updated, and the free map bits have to be marked as in-use. If these metadata are put into the journal first and then the system crashes, upon journal recovery the file would *appear* to contain the new data, but in fact the contents of those data blocks as described in the block map will be garbage. By forcing data write-back prior to metadata journal commit, this situation is avoided.
• `data=writeback`: This mode will have the best performance but the worst integrity. The write-back of data blocks is not synchronized at all with metadata writes to the journal. The situation described above can happen.

This discussion of journaling is necessarily a brief one. There are many complex issues which affect performance and reliability. Journaling is still an active area of research and development.

A review of filesystem-related system calls seen thus far

```
open:     Create an ordinary file, and/or open a file for I/O.

read:     Read data from an open file

write:    Write data to an open file

lseek:    Change the current position within an open file

close:    Close an open file

unlink:   Destroy a path to a file (or symlink)

mkdir:    Create a directory node

rmdir:    Destroy a directory node (must be empty)

link:     Clone a file, producing a hard link

symlink:  Create a soft (symbolic) link

readlink: Query the value of a symlink

stat:     Retrieve the metadata information about a path
```

```
fstat:    Retrieve the metadata information about an open file

lstat:    Retrieve the metadata information without following symlinks

chown:    Change the uid and gid ownership of a node

chmod:    Change the permissions mask of a node

utime:    Change the timestamps

opendir:  Pseudo-system call to open a directory

readdir:  Pseudo-system call to read the next directory entry

closedir: Close a directory opened by opendir
```