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.

We can think of a disk as a randomly addressable array of bytes. (In fact, the smallest addressable unit is larger than a byte, and is called a sector. A typical sector size is 512 bytes. We will ignore this distinction at this time). We'll call each such randomly-addressable array 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 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.

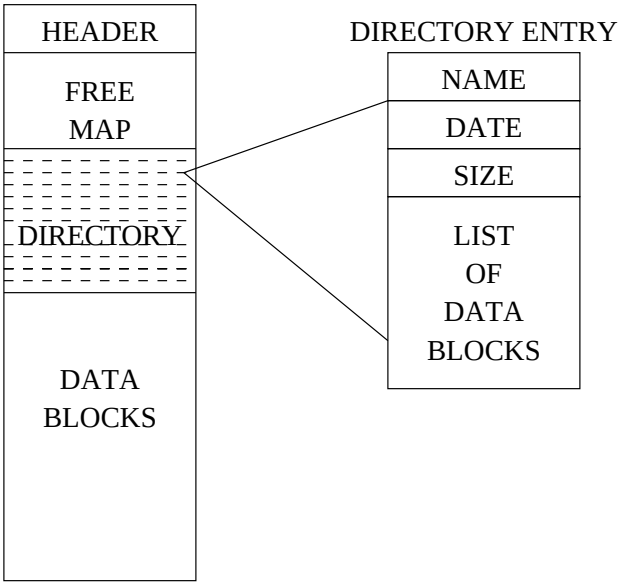
• **Meta-data**: The filesystem tracks other data which are not the actual contents of the files, but are data about the data, or "meta" data. These data may include the size of the file, time of last modification, the creator, etc.

• **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 prohibitely expensive, for example, many embedded devices, such as digital cameras or network switches/routers.



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.

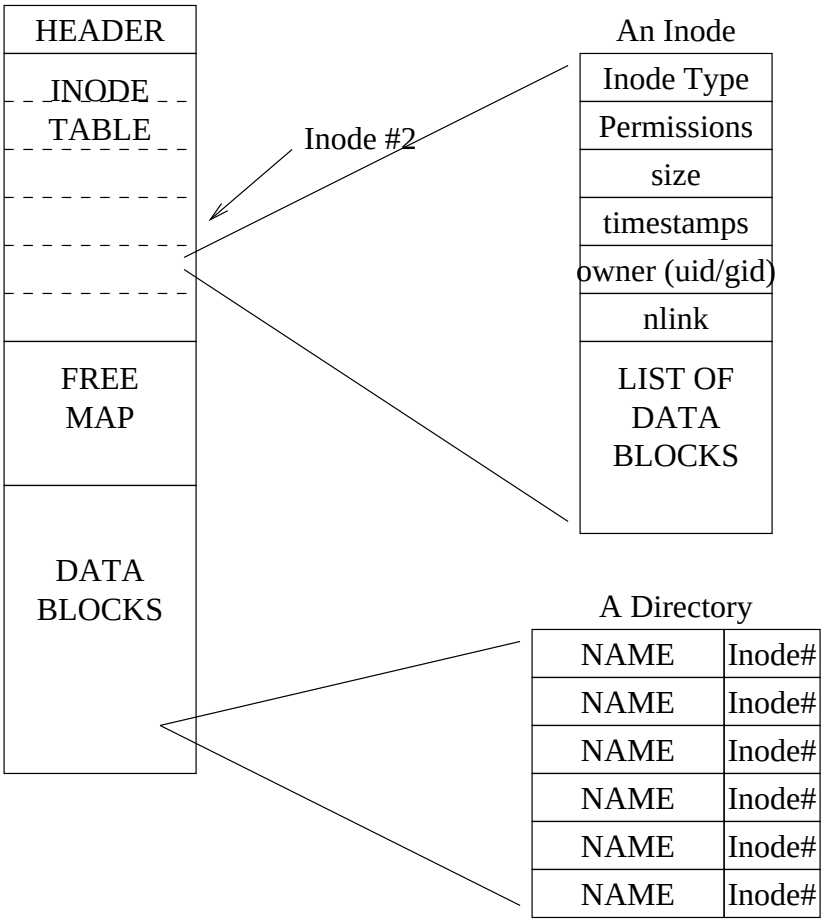
The contents of the files are stored within the data block section. The free map section maintains an overview of which data blocks are currently being used, and which are free to be allocated to files as they are written. Common implementations of the free map are a linked list of free blocks, or a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

In the FAT filesystem (aka the MSDOS filesystem), the free map area 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.

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 disk 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).

We'll see in subsequent units that the block becomes 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. The data block size should not be confused with the physical block (or "sector") size.

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**.

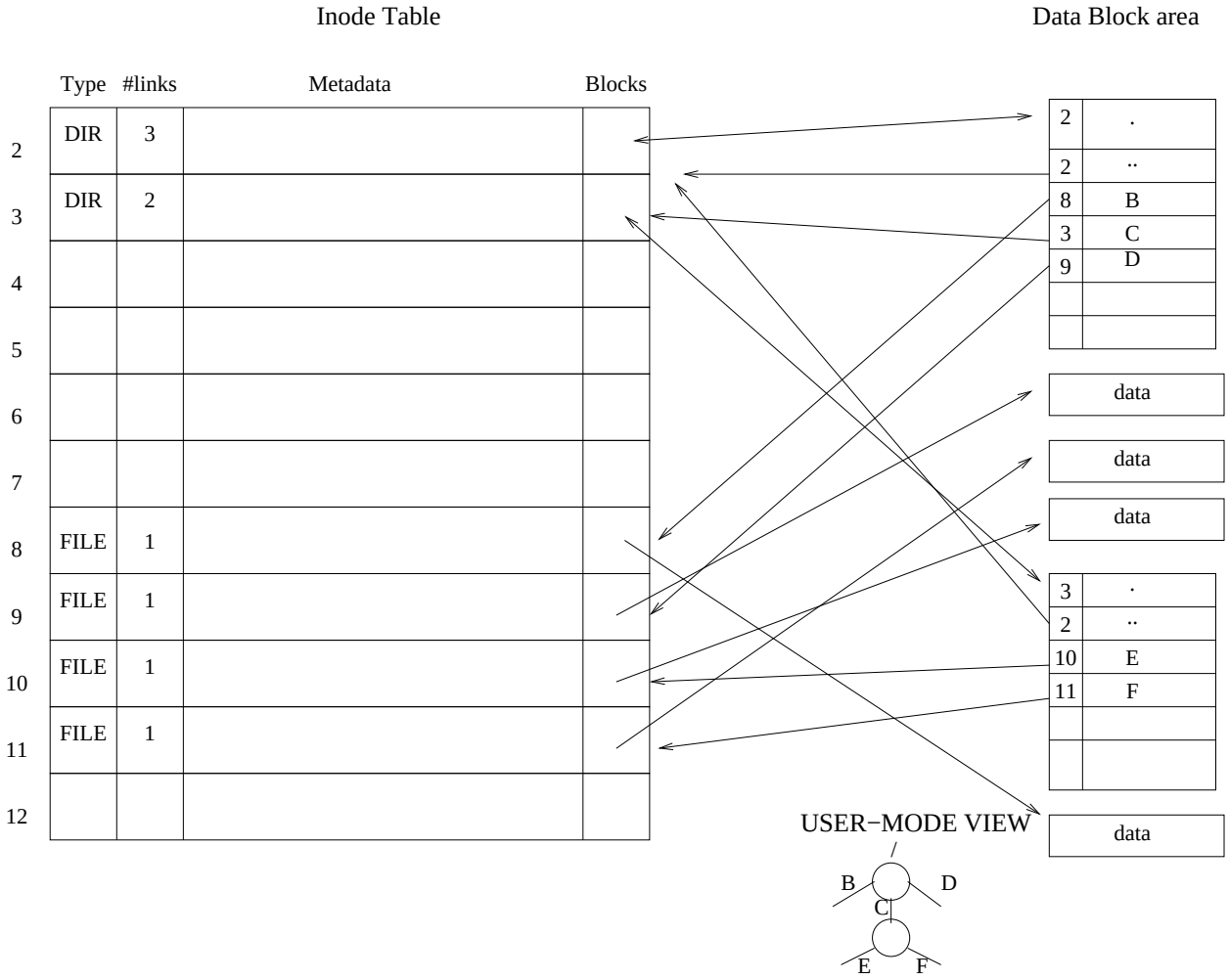
The inode table is conceptually an array of these inodes (a typical on-disk inode size is 128 bytes), indexed by **inode number**. Again, for historical and kernel programming reasons, the first inode is usually numbered 2 (because 0 was used to indicate an empty directory slot, and 1 was reserved for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

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 trees are introduced. 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.



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.

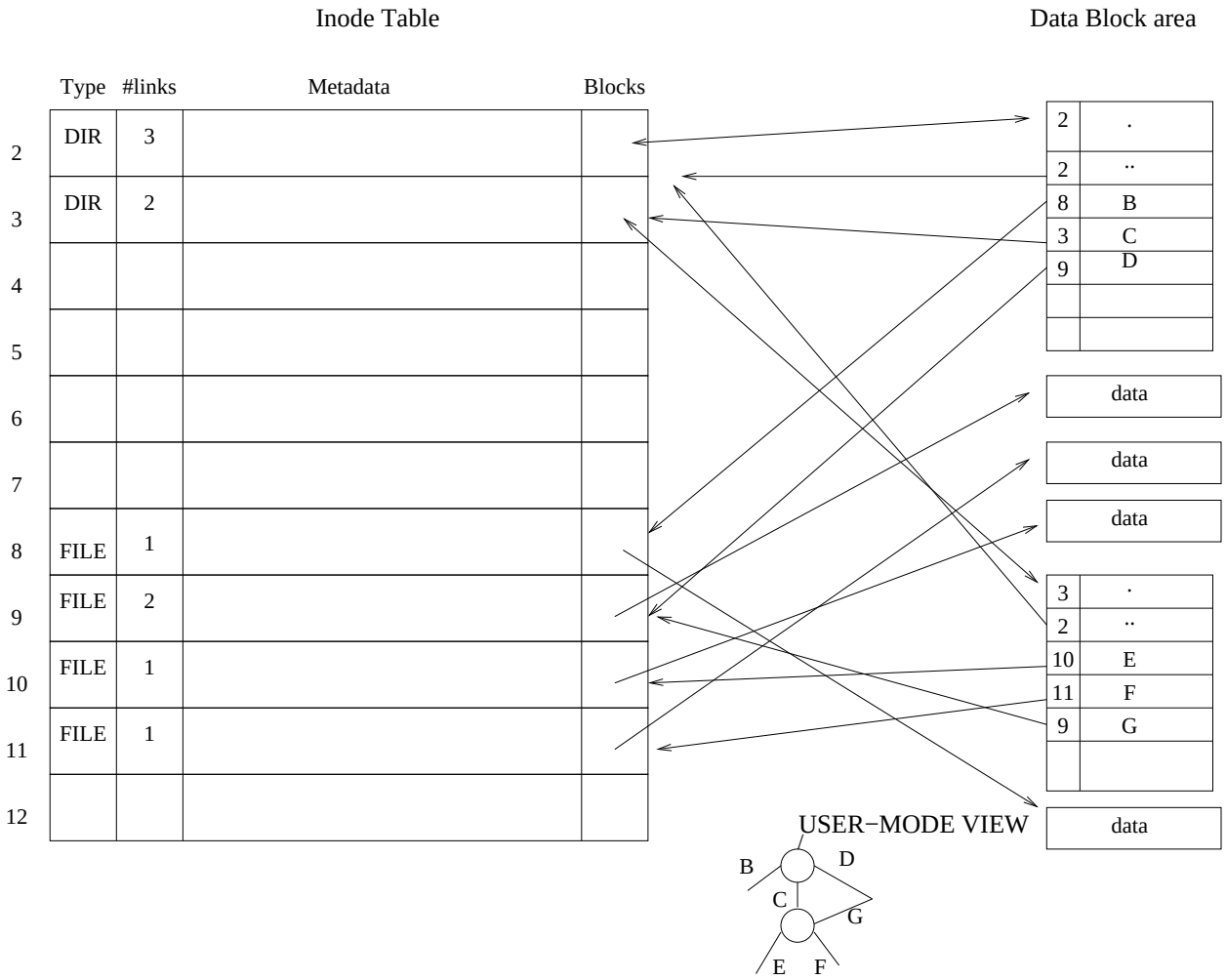
There are two ways to look at this, which are analogous to Hard Links vs Symbolic Links in UNIX. In the latter case, we think of one of the names of the file as being canonical, while all the others are "aliases." In the former case, all of the names are equal in stature.

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.

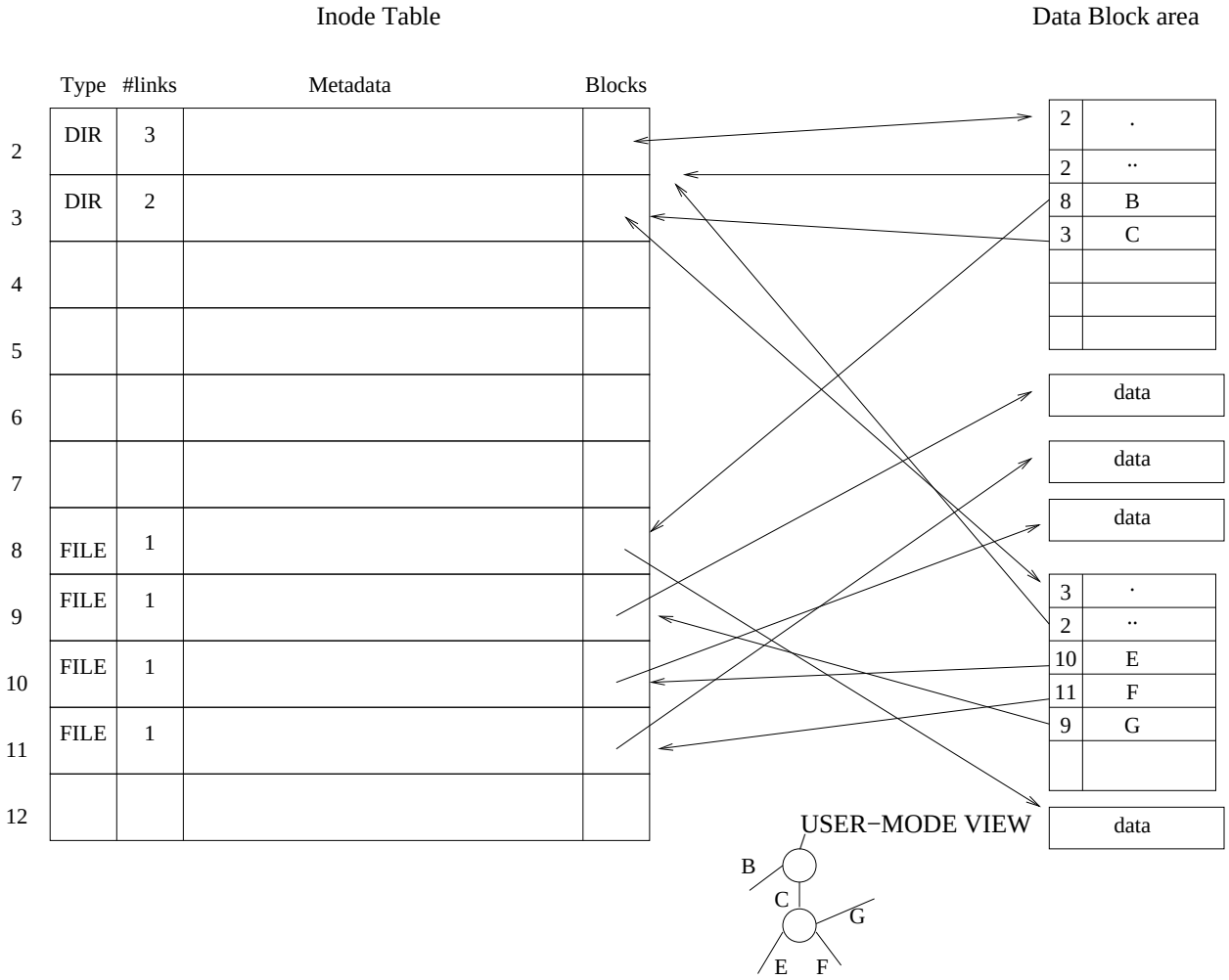
Notice that the inode #9 field *links* has gone up. The operating system must maintain this counter in order to be aware of how many paths exist in the filesystem which point to this inode. We could now remove the pathname "/D", by executing the UNIX command:

```
rm /D
```

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

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

After which, the filesystem looks like this:

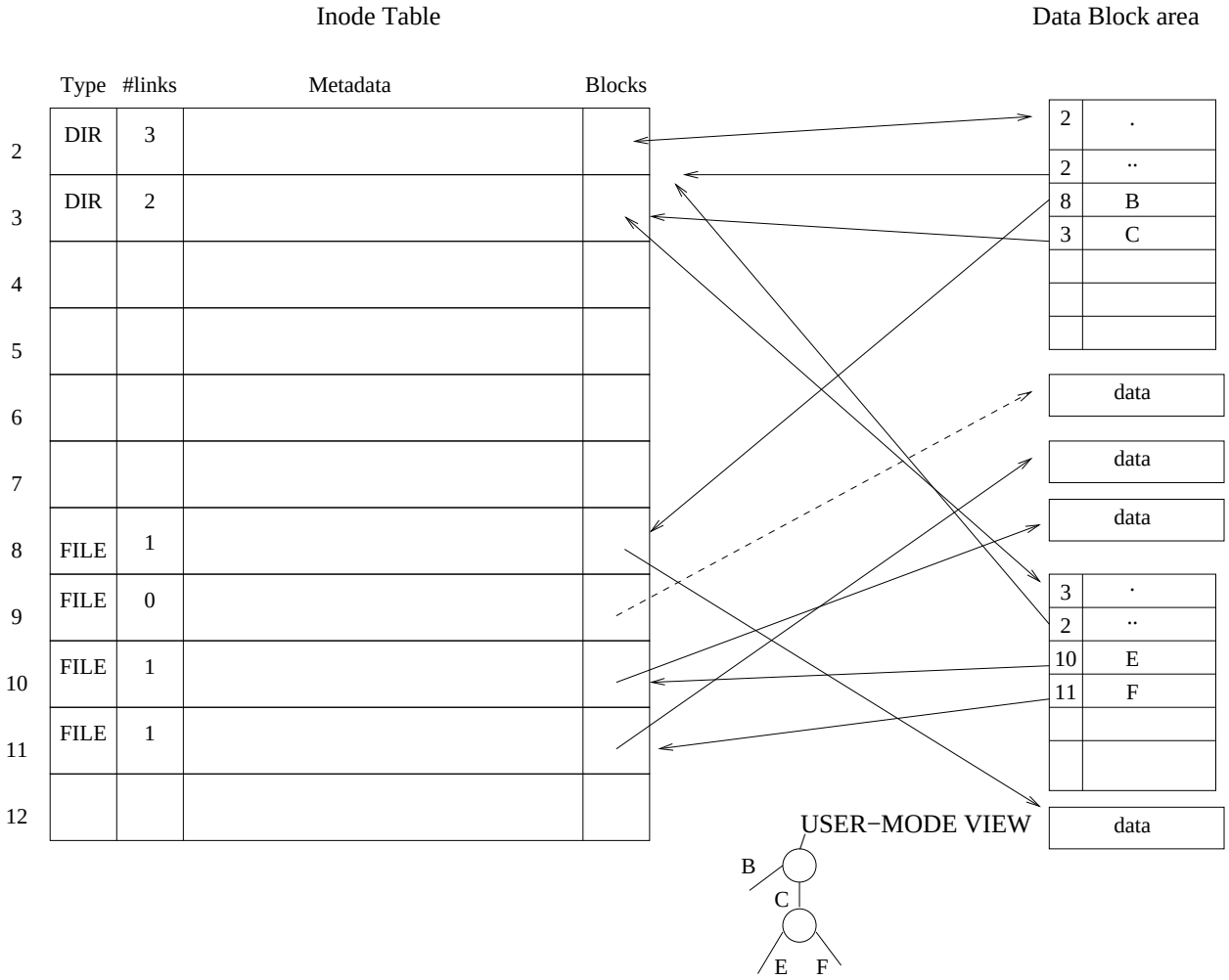


Even though we have issued the **rem**ove command, the actual file represented by inode #9 has not been removed! There is still another pathname which refers to this file ("/C/G"). The file will not actually be deleted until its link count falls to 0, i.e. there are no more pathnames that point to the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

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. 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.

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.

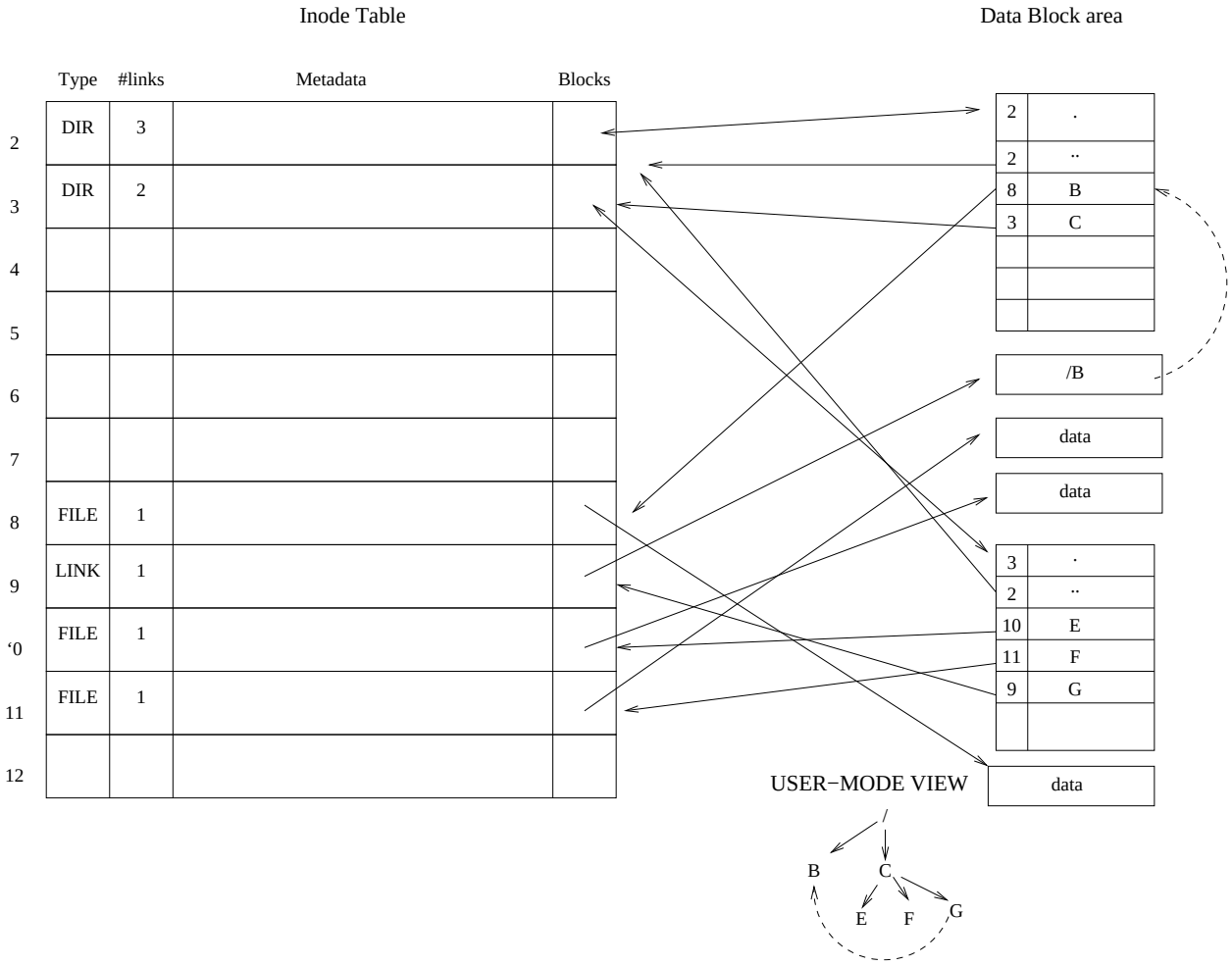
Symbolic Links

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 (we'll see the exact rules for this when we explore the path name evaluation routine in the kernel). 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 value of the symlink, 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.

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);
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.

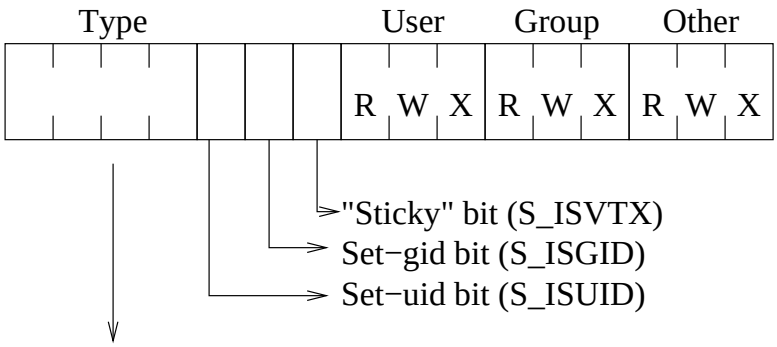
- `st_mode`: The inode type and permissions (see below)
- `st_nlink:` Number of pathnames linking to inode
- `st_uid`: The user id which owns the inode
- `st_gid`: The group owner of the inode
- `st_rdev`: Device number (character and block device special inodes only)
- `st_size`: The size of the data, 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_blksize`: The size, in bytes, to be used in conjunction with:
- `st_blocks`: The number of blocks of storage space occupied by the file. `st_blocks*st_blksize` is the total disk space taken up by the file, which 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_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 GMT. 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.

`st_blksize`, `st_blocks`, `st_dev` and `st_ino` do not appear anywhere in the on-disk inode. They are are added by the operating system. `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 below.

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 16 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):
S_IFBLK (type==6): UNIX gives names to devices and provides access to them through the filesystem. E.g. `/dev/hda1` 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, propertiary 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 and can later be modified by the `chown` system call. 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.

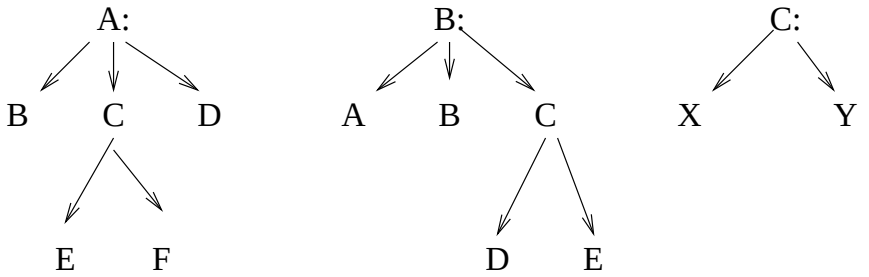
Most UNIX systems support a more elaborate way of expressing filesystem permissions known as **Access Control Lists**. These will be discussed in a future unit. Their application is not widespread because the traditional 3-tiered UNIX permissions model is

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:

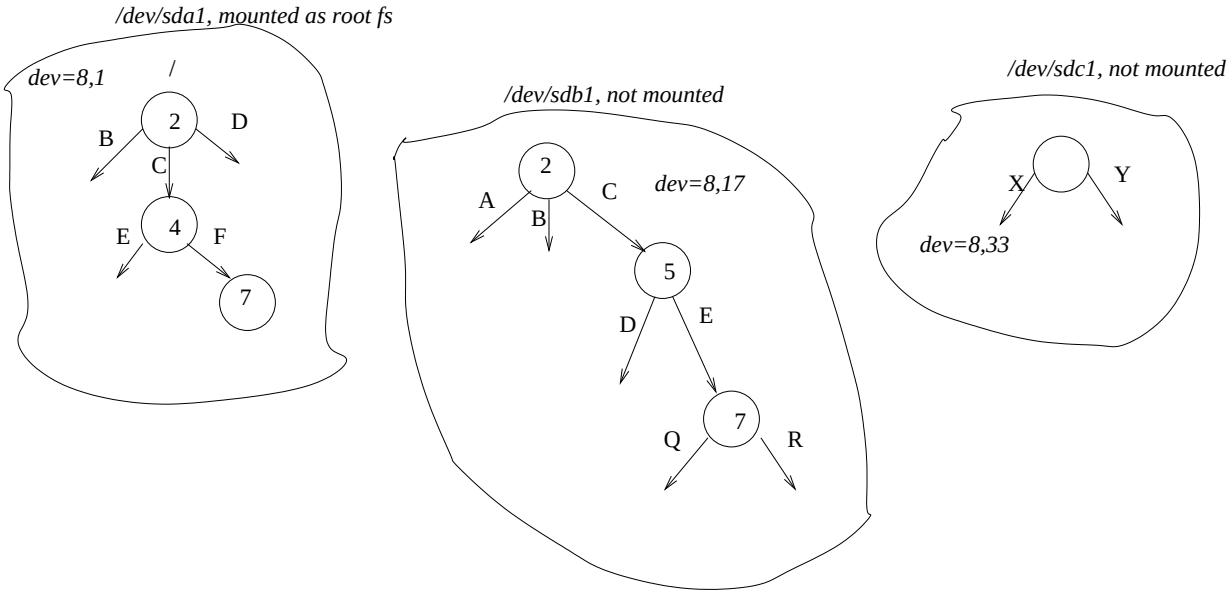


Each volume is an independent tree, and the collection of all such trees forms a flat namespace.

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.

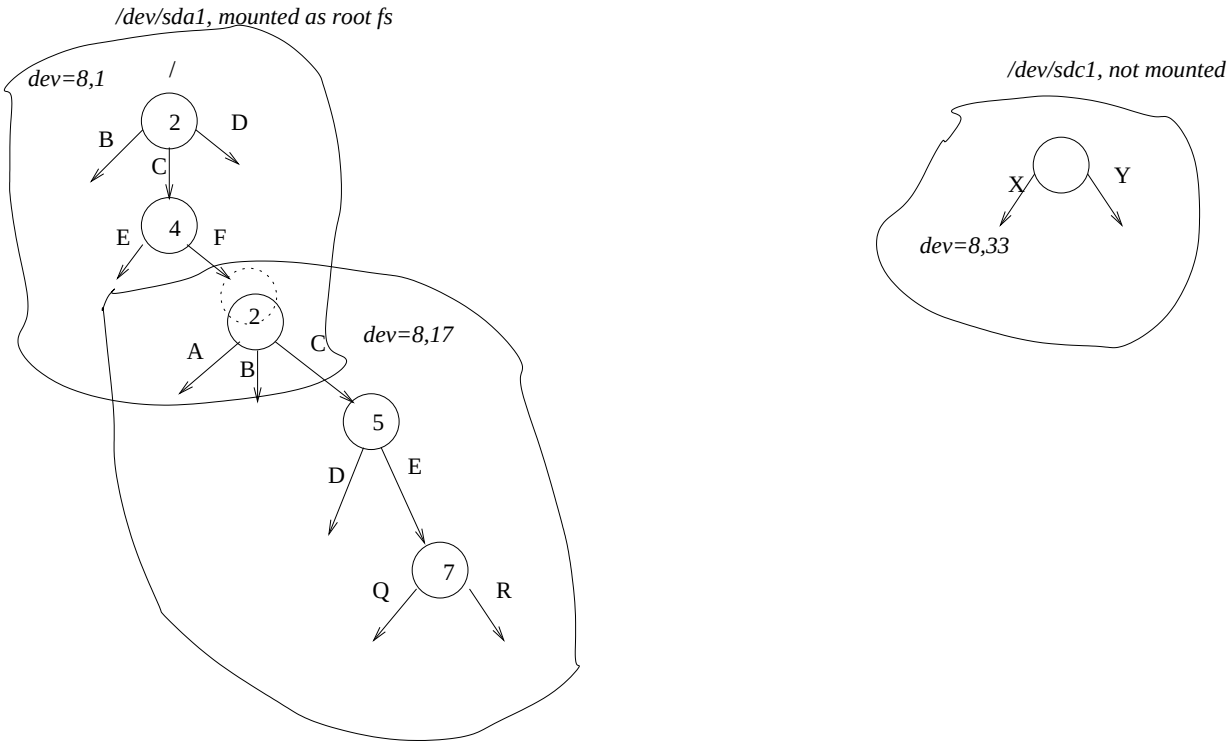


Now we execute the command:

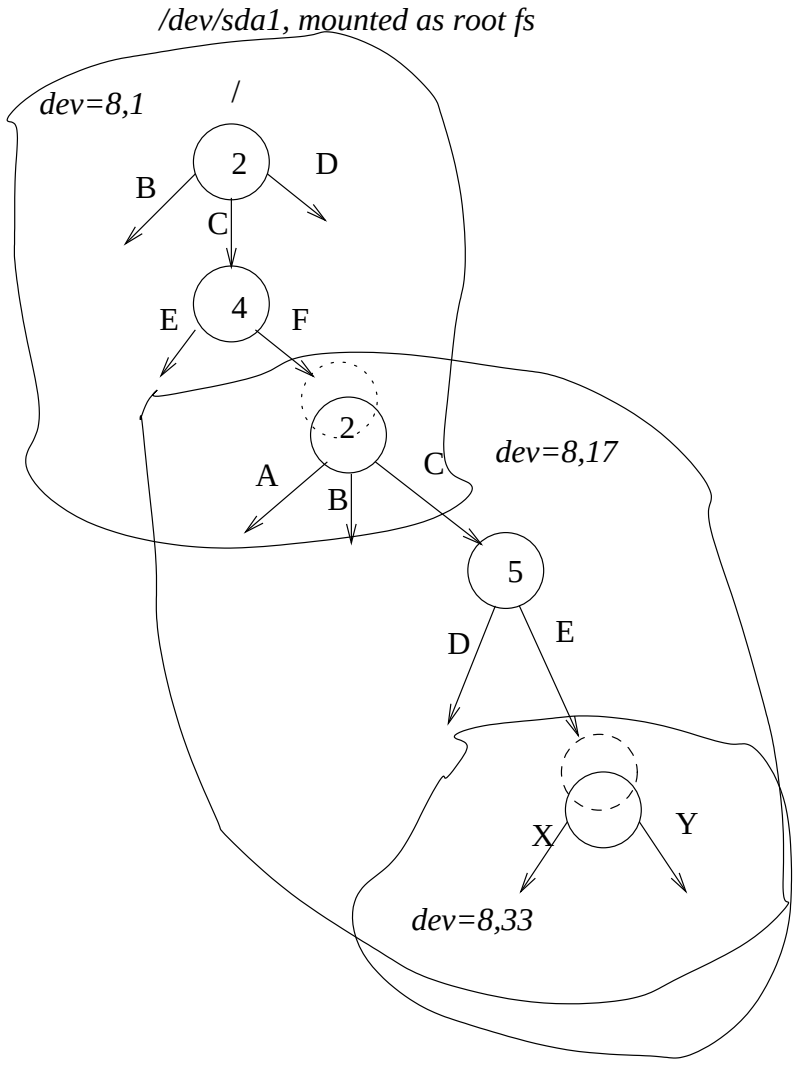
```
mount /dev/sdb1 /C/F
```

The pathname `/dev/sdb1` is in a special part of the filesystem in which the nodes are device special inodes (S_IFBLK or S_IFCHR). When the kernel translates this pathname, it produces a major,minor device number pair, in this case, 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 obscurred, 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 obscurred 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 some 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).

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 `unmount` 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).

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. In future units, we will explore some of the implications of doing so.

The filesystem in UNIX is heterogenous, 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.

However, note that 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 "foreign" semantics.

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.

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.

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.

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
```