Topic 5
Unix File Systems and Programming

Objectives

- Understand the concept of file in Unix systems
- Understand the internal data structures for open files
- Understand and be able to use low level I/O primitives
- Understand the I/O efficiency issue related to the buffer size
- Understand the standard input, standard output and standard error and their redirections via `dup` and `dup2` system calls
- Be able to use function `fcntl`
- Understand and be able to use C's standard I/O library
- Understand the role and the content of i-nodes
- Be aware of the structure and content of directory files
- Understand the structure of a typical Unix filesystem
- Be aware of the various types of UNIX files
- Understand and be able to change the access permissions of files in programs
- Understand the role of `umask` value
- Understand and be able to use both hard links and symbolic links
- Be able to obtaining information about a file
- Understand and be able to read and write directories in programs.

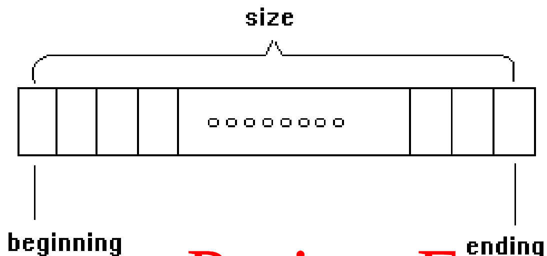
Readings:

- Stevens & Rago: Ch 3 & Ch 4
- Skim Stallings' Chapter 12

1. The Unix File Abstraction

To an application programmer, every Unix file has the following features:

(a) a file name
(b) a sequence of bytes
(c) no further structures (such as lines or records)



Behind the scenes, a file is stored as a collection of blocks of a fixed size on a storage media such as hard disk. These blocks are scattered around different parts of the disk or similar block device.

2. The Kernel Data Structures for File I/O

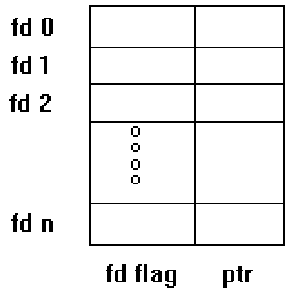
Before reading from or writing to a file, the kernel must "open" the file first. The open file operation establishes the appropriate data structures in the kernel to facilitate the actual I/O between the disk and the RAM in the future:

(a) To locate the file on the disk according to its file name (or path);
(b) To establish data structures for accessing and sharing the file among different processes;
(c) To allocate a file descriptor for the file so that the subsequent read/write requests would use this file descriptor (instead of the file name) to refer to this open file.

The classical Unix system uses three data structures for this purpose: 1) the *Per Process Table of Open Files*, 2) the system-wide *File Table,* and 3) the system-wide *V-Node Table*. Understanding these data structures will help you gain an insight into how the files are accessed and shared and also various efficiency issues.

Please note that not all Unix-like operating systems (e.g., Linux) use exactly the same kernel data structures for their open files. However, they all support the abstraction that the classical Unix kernel data structures implement.

(1) The Per Process Table of Open Files



Note:

(a) Each process has one Per Process Table of Open Files, which contains the information of the files the process opened (and not yet closed), one entry for each file. The index of the entry becomes the file descriptor of the open file.

(b) In the entry of an open file, the file descriptor flag specifies whether this file descriptor should be closed upon executing one of the “exec” functions. POSIX.1 only defined one flag: `FD_CLOEXEC`, which means that the open file is closed when the exec function loads in a new program into the process.

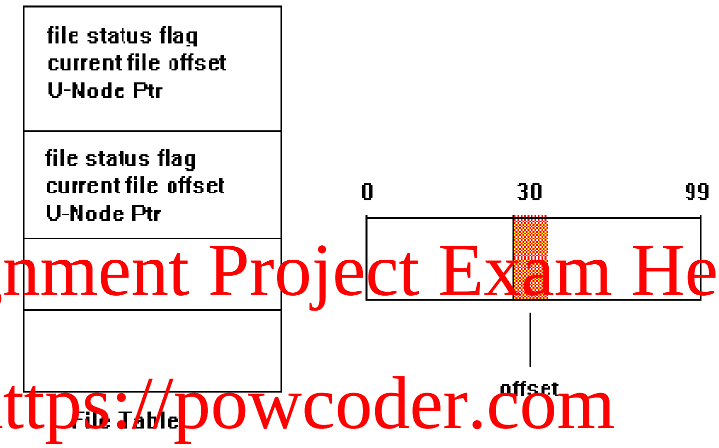
(c) The “ptr” points to the entry in the File Table that describes how the file should be accessed.

(d) When a new process is created, three "files" are open automatically pointing to the terminal input device (usually the keyboard) and the terminal output device (usually the display or terminal window), and their file descriptors are 0, 1 and 2 respectively. These three open files are known as the *standard input (file descriptor 0)*, the *standard output (file descriptor 1)* and the *standard error* (file descriptor 2) for the new process.

(e) Any read/write operation for a file must use its file descriptor (not the file name, why?) to refer to the file.

(2) The File Table

The kernel also keeps one system-wide table for all open files and for all processes. Each open call is given one entry in this table. This means that, if the same file is opened several times (not yet closed) from several different processes, there will be multiple entries in this table for that file.



Note:

(a) The File Status Flags indicate whether the file is opened for

- read only (`O_RDONLY`) or
- write only (`O_WRONLY`) or
- both read and write (`O_RDWR`)

  +

- append on each write (`O_APPEND`)
- non-blocking read (`O_NONBLOCK`)
- wait for write to complete (`O_SYNC`)

(b) Current offset indicates the position where the next read/write begins;

(c) V-node ptr points to the entry in the V-node Table that contains all pertinent information on that file.

(3) The V-Node Table

Apart from the File Table, the kernel also keeps another system-wide table, known as the V-Node Table, for all open files and for all processes. Unlike the File Table, each open file has one and only one entry in this table.

V-node information I-node information (incl Current file size)
V-node information I-node information (incl Current file size)

Note:

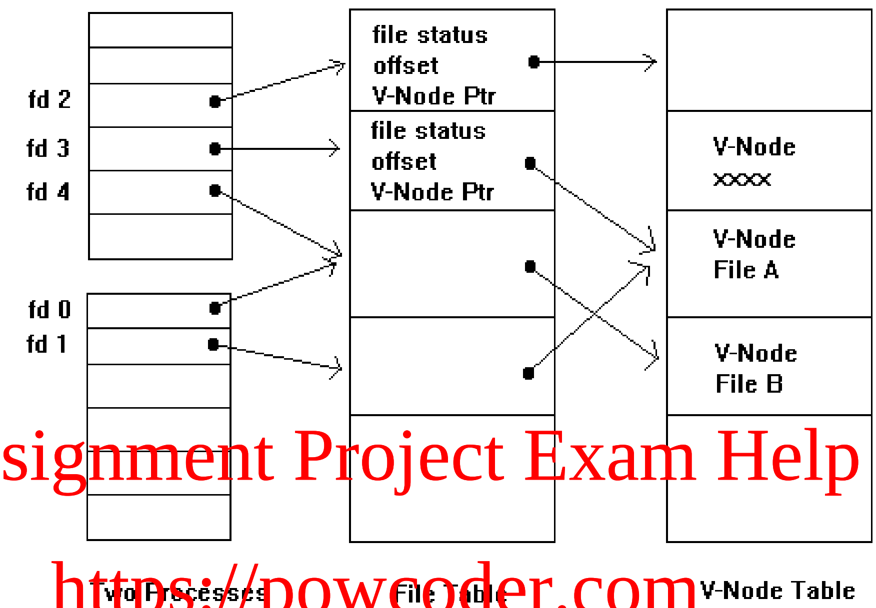
(a) V-node information: the information about the type of file (regular, pipe, directory, etc.) and a pointer to the function that operates on that type of file.

(b) If the file is a regular or directory file, the entry in the V-node Table also contains the i-node of the file, which is copied from the disk. The i-node contains all pertinent information about the file, such as owner, access permission, size and access time, about that file.

(c) A file can only have one entry in the V-Node Table, no matter how many times it was opened and how many processes have opened it. This is contrary to the File Table where a file may have several entries in the File Table, depending on how many times it was opened (and not closed).

(4) Sharing Files Among Processes

The following diagram illustrates how file sharing is achieved among different processes via the three tables.



In the above diagram, there are two processes, each having its own Per Process Table of Open Files, sharing two files: File A and File B.

There are two types of file sharing between the two processes. However, there are differences in the way these two files are shared. For example, File B is shared via the same File Table entry while File A is shared via two separate File Table entries.

What are the consequences due to the differences in the way the files are shared?

3. System calls `open` and `close`

The `open` system call establishes the necessary data structures (allocates an entry in the Per Process File Table and add an entry in File Table and possibly entry in V-Node Table if this is the first time the file is open by any process) for a file and returns a file descriptor that will be used to represent the open file. The file descriptor is the index to the allocated entry in the Per Process Table of Open Files.

The `close` system call removes the data structures for the given file.


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

int open (const char pathname, int oflag,
          /* mode_t mode */ )
```

Return the file descriptor if OK or -1 on error

`oflag`: consists of exactly *one* of the following three flags (mutually exclusive):

- O_RDONLY – read only or
- O_WRONLY – write only or
- O_RDWR -- both read and write

and a *combination* of the following optional flags (incomplete list):

- `O_APPEND` -- to append to end of the file on each write, i.e., set the offset to the end of file before each write
- `O_CREAT` -- to create the file on disk if the named file does not exist. This option requires the third argument, *mode*, which specifies the access permission for the new file.
- `O_EXCL` -- to generate an error if option O_CREAT is specified and the file already exists.
- `O_TRUNC` -- if the file exists and is open for write or for both read and write, truncate its length to 0 (i.e., replace the old content)
- `O_NONBLOCK` – mainly for FIFOs, character special files or block special files. If the I/O cannot be satisfied immediately, return with an error without wait.

The file descriptor returned from the `open` or `creat` system call is guaranteed to be the lowest number available (unused descriptor).

Example 1: Open an existing file for writing

```
fd = open ("foo", O_WRONLY);
```

Example 2: Append any writing to the existing file

```
fd = open("foo", O_WRONLY|O_APPEND);
```

Example 3: If the file "`foo`" exists, open it for writing. Otherwise create a new file on the disk for writing with access mode 0766 (octal number)

```
fd = open("foo", O_WRONLY|O_CREAT, 0766);
```

Example 4: Same as the above except that each write is appended to the end of the file

```
fd = open("foo", O_WRONLY | O_CREAT|
O_APPEND, 0766);
```

Example 5: Open the terminal device `/dev/ttyp0` for non-blocking read

```
fd = open ("/dev/ttyp0", O_RDONLY |
O_NONBLOCK );
```

Close System Calls


```
#include <unistd.h>

int close (int filedescriptor);
```


Return 0 if OK or -1 on error

4. System Calls: `read` and `write`

```
#include <unistd.h>

ssize_t read (int filedes, void *buffer,
              size_t nbytes );
```

Return: number of bytes read in, or 0 if the offset is pointing to the end of file, or –1 if error

Note: it is the responsibility of the caller to allocate space for the buffer!

```
#include <unistd.h>
```



```
ssize_t write (int filedes, void *buf,
               size_t  nbytes);
```


Return: number of bytes written if OK or –1 on error

Example: Copy file `fd1` **to file** `fd2`

```
void copy (int fd1, int fd2)
{
    ssize_t nread;
    char buf[100];

    while ((nread=read(fd1,buf,100))>0)
        write (fd2, buf, nread );
}
```

Is the above program correct? If not, what could be the problem? (Hints: think about the write call in the program).

5. I/O Efficiency

Since disk files (i.e., regular and directory files) are stored block by block in the disk, and disk I/O is performed block by block, the size of the temporary buffer used in read and write operations can affect I/O performance significantly. In order to reduce the number of actual disk I/O, hence increasing I/O performance for large files, the buffer size in our programs should either be the same or multiple of the block size.

The following example illustrates the performance variation for different buffer sizes. The program tries to copy a file of size 1,468,802 bytes. The block size of the disk is 8192 (Stevens P.56).

BUFSIZE	USER CPU (seconds)	SYS CPU (seconds)	Clock Time (seconds)	#loops
1	23.8	397.9	423.4	1468802
64	0.3	6.6	7.0	22950
1024	0.0	0.6	0.6	1435
8192	0.0	0.3	0.3	180
32768	0.0	0.3	0.3	45
131072	0.0	0.3	0.3	12

6. Change Offset – `lseek`

When a file is opened, its offset is set to zero (0), i.e., it points to the beginning of the file. Each successive read and write would advance the offset by the number of bytes read or written. The offset can be changed to any value (>=0) by the `lseek` system call.

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

off_t lseek (int filedes, off_t offset,
             int WHENCE );
```


Returns: the new file offset if OK, or -1 on error


```
WHENCE:
   SEEK_SET:
        new_offset = 0 + offset
```



```
   SEEK_CUR:
        new_offset = current_offset + offset

   SEEK_END:
        new_offset = file_size + offset
```

Note:

(a) Argument `offset` can be non-negative as well as negative

(b) The new offset should normally be non-negative (at least for regular files)

(c) `lseek` does not cause any file I/O

Example Program: test `lseek`

```
/* file name:     test_lseek.c
*/

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

#define MODE 0666  /* access permission */

int main(void)
{
    char buf1[]="abcdefghijklmnopqrstuvwxyz";
    char buf2[]="*************************";
    int fd;
    off_t newpos;
    ssize_t n;

    if ((fd=open("testfile", O_RDWR
          | O_CREAT | O_TRUNC, MODE))== -1){
        printf("Error: unable to open or create file\n");
        exit(1);
    }

    if (write(fd, buf1, 26) != 26) {
        printf("Error:  cannot write 26 bytes\n");
        exit(1);
    }

    newpos = lseek(fd, 10, SEEK_SET);  // newpos=10
    printf("Current offset = %d\n", newpos);

    newpos = lseek(fd, -10, SEEK_END); // newpos=26-10=16
    printf("Current offset = %d\n", newpos);

    newpos = lseek(fd, -10, SEEK_CUR); // newpos=16-10=6
    printf("Current offset = %d\n", newpos);

    n = read(fd, buf2, 10); // read in "ghijklmnop"
    printf("buf2 = %s\n", buf2);

    buf2[n] = '\0';
    printf("buf2 = %s\n", buf2);

    close(fd);
    exit(0);
}
```

7. The `dup` and `dup2` system call for Standard Input/Output/Error Redirection

The normal output of a command such as `ls` usually goes to the terminal output (the screen or the terminal window). We can redirect this output to a file from the shell prompt:

```
% ls > foo
```

In the above example, the normal output from the `ls` process would not go to the terminal output device. Instead, it is written to file `foo`.

Since no one has changed the program `ls` in anyway, we must assume that the process `ls` knows nothing about the redirection. How then is this redirection achieved without the knowledge of the process?

Earlier, we mentioned that when a process is created, three "files" are opened automatically, with the following file descriptors:

Descriptor	*Constant symbol*	*Common name*
0	`STDIN_FILENO`	standard input
1	`STDOUT_FILENO`	standard output
2	`STDERR_FILENO`	standard error

These three file descriptors are initially associated with the control terminal of the process:

- the standard input with the terminal input device (eg, keyboard)
- the standard output with the terminal output device (eg, the monitor screen or the terminal window)
- the standard error with the terminal output device

Most Unix programs are written as “filters”. A filter is a program that takes its input from the standard input and then sends its normal output to the standard output and its error messages to the standard error.

Function `printf` sends a string to the standard output. This is why statements such as

```
printf("hi, there");
```

would usually send their arguments to the monitor screen or the terminal window, because the standard output is linked to the terminal output device. Similarly, statements such as

```
scanf("%d", &n);
```


would usually read their inputs from the keyboard.

However, we can change the content in the entry 0, 1, or 2 in the Per Process Table for Open Files so that it points to the File Table entry of another file. This would cause the standard input, the standard output or the standard error to be directed away from the terminal device to a regular file.

To achieve this, `dup` or `dup2` system call is used to duplicate the entry of a given file descriptor on another entry of the Per Process Table of Open File.

```
#include <unistd.h>

int dup (int filedes);
int dup2 (int filedes, int filedes2);
```

Both return: the new file descriptor if OK, or -1 on error

Note:

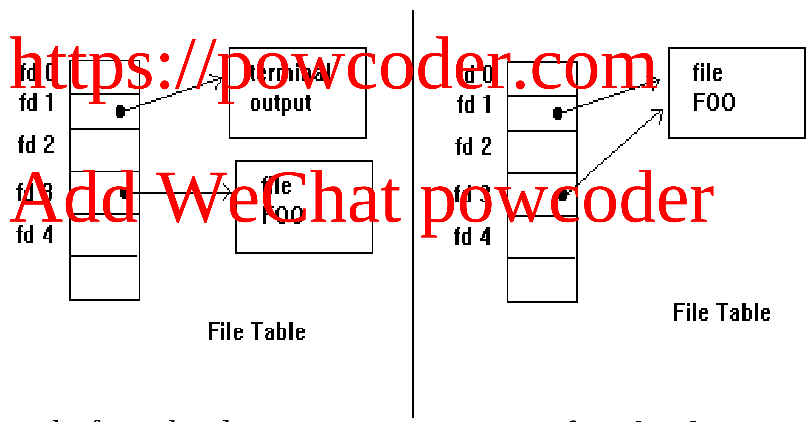
(a) With `dup`, the new file descriptor is guaranteed to be the lowest numbered file descriptor that was available in the Per Process Table of Open Files.

(b) With `dup2`, the new file descriptor value is `filedes2`. If `filedes2` already exists, it is closed first.

(c) The duplicated file descriptor shares the same entry in the File Table as the original file descriptor.

Example:

```
fd = open ("FOO", O_WRONLY|O_CREAT, 0766);
close (STDOUT_FILENO);
dup (fd);
```

before the dup call

after the dup call

The above `close` and `dup` calls can be combined with a single `dup2` call:

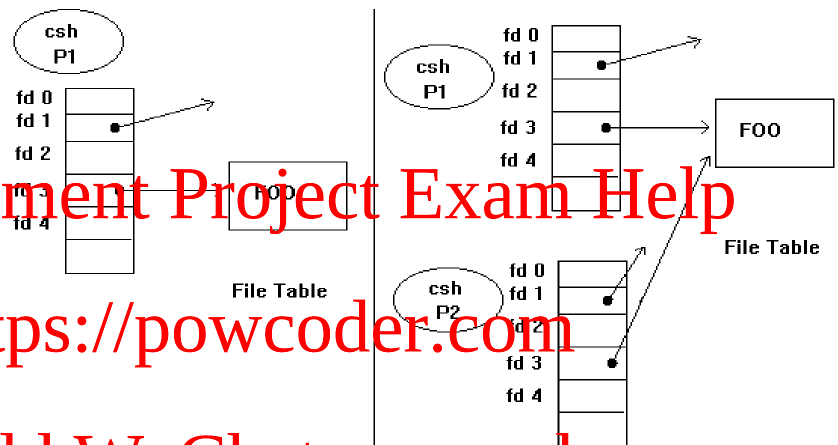
```
fd = open("FOO", O_WRONLY|O_CREAT, 0766);
dup2 (fd, STDOUT_FILENO)
```

How does a shell achieve standard I/O redirection in the command below?

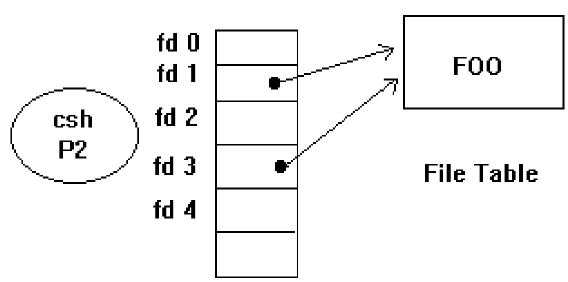
```
% ls > FOO
```

Step 1: The shell opens or creates the file `FOO` and obtains the file descriptor `fd` allocated to the file.

Step 2: The shell creates a copy of itself using `fork` system call. The newly created child process inherits all file descriptors from the parent process, including `fd`. In the diagram below, the shell is C shell (csh). Other shells achieve this in the same way.



Step 3: In the child process P2, it closes file descriptor 1 and duplicates `fd` on 1:



Step 4: Child process P2 loads the program `ls` using one of the “`exec`” functions, and then passes the control to the `ls` program. The `ls` program would run and send its output to the standard output, which is now the file `FOO`!

8. The `fcntl` Function

This function is used to change the properties of a file that is already open.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl (int filedes, int cmd,
         /* int arg */ );
```

Returns: depends on cmd if OK, -1 on Error

Depending on `cmd`, `fcntl` can be used to perform various operations including the following:

(1) **duplicate existing File Descriptor** (`cmd = F_DUPFD`)

Example:

```
newfd = fcntl (fd, F_DUPFD, 3);
```

`newfd` is guaranteed to be the lowest numbered file descriptor available that is greater than or equal to 3, the third argument.

Note: both descriptors, `fd` and `newfd`, point to the same entry in the File Table.

(2) **Get or set File Descriptor Flags** (`cmd = F_GETFD, F_SETFD`)

Example: check the file descriptor flag

```
fd_flag = fcntl (fd, F_GETFD, 0);

if (fd_flag & FD_CLOEXEC)
     printf("fd flag: close-on-exec \n");
```

Example: set `close-on-exec` **flag:**

```
fcntl (fd, F_SETFD, FD_CLOEXEC );
```

Example: clear file descriptor flag:


```
fcntl (fd, F_SETFD, 0 );
```


(3) get or set File Status flags (`cmd = F_GETFL, F_SETFL`**)**

Example: check file status flag (Stevens P.65):

```
int  accmode, val;

val = fcntl (fd, F_GETFL, 0 );

accmode = val & O_ACCMODE;
if (accmode == O_RDONLY)
      printf("read only");
else if (accmode == O_WRONLY)
      printf("write only");
else if (accmode == O_RDWR)
      printf("Read and Write");
else
      printf("error in access mode");
```

```
If (val & O_APPEND )
      printf (", Append");
If (val & O_NONBLOCK)
      printf (", NON-Blocking");
```

Example: Set non-blocking mode:

```
val = fcntl (fd, F_GETFL, 0);
val = val | O_NONBLOCK;
fcntl (fd, F_SETFL, val);
```

9. i-nodes and directory files

Internally in the disk, each file has one i-node, which contains all information about that file except:

- the i-node number
- the file name

“i-node” stands for “information node”. An i-node contains following information about a file:

- file type
- access permissions
- owner and group owner
- size of the file
- number of links
- device number of the device where the file is stored
- time of last access
- time of last modification
- time of last change to the i-node
- block numbers of the blocks in which file is stored

Inside the Unix kernel, a file is identified by its i-node number, not by its file name.

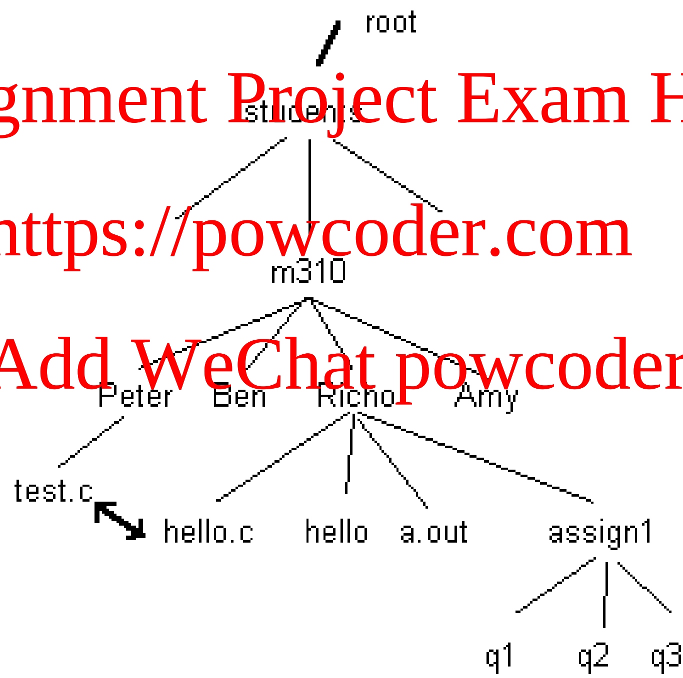
A Unix directory file contains the names of the files it "contains" and the i-node numbers of these files. It provides a mapping between the file name and its i-node number:

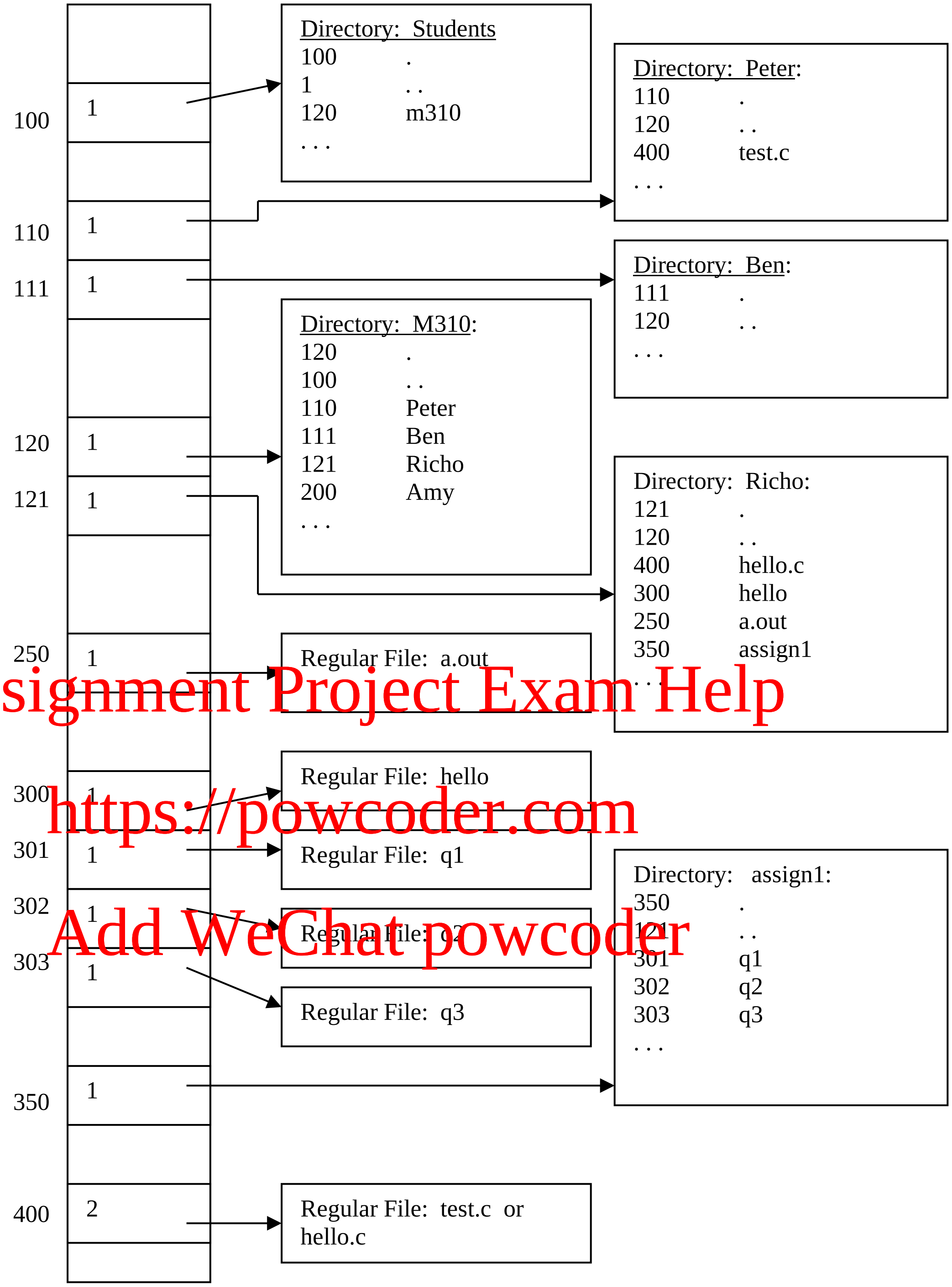
Example: In the following diagram, the directory “`assign1`” contains three files: `q1`, `q2`, `q3` with the corresponding i-node numbers `200`, `280`,and `90`. The contents of the directory file would be something like:

```
100         .
140         ..
200         q1
280         q2
90          q3
```

Note: each directory contains the i-node number of itself (the dot "`.`") as well as the i-node number of its parent directory (dot-dot "`..`"), even for "empty" directories.

Example. A sample directory tree





In the above diagram, the list on the left is the *i-list* (a list of i-nodes) of the filesystem (think of it as a disk partition in Windows). Each block in the i-list contains the information about one file. The number shown in each block is the link count of the file. The diagram illustrates how the files may be organised internally in a filesystem.

10. The Types of Files

	the character used in "`ls -l`" listing
(1) a regular file	-
(2) a directory files	d
(3) a character Special Files	c
(4) a block Special Files	b
(5) a FIFO (named pipe)	p
(6) a socket	s
(7) a symbolic Links	l

Example: Character Special File

```
% ls -l /dev/ttyp1
```



```
crwx-w----  1  hong  20,1  Mar 12  10:24  /dev/ttyp
```


Example: Block special file

```
% ls  -l  /dev/rz3e
```



```
brw-------  1  root  21,28   Jun 9  1995   /dev/rz3e
```

Example: Symbolic Link

```
% ln  -s     m310/tut01.txt       tut1.txt
% ls  -l     tut1.txt

lrwx--- ---   1  hong  14  Mar 12  10:15   tut1.txt -
-> m310/tu01.txt
```

11. Access Permissions of Files

Each file belongs to one user (known as the owner) and one user group (known as group owner). A user can change the access permissions of the files s/he owns. This is the only extra privilege the owner has over other users.

The access permissions consist of three sets of permissions for read, write and execute (or search for directory): the first set of permissions for its owner, the second set of permissions for any member in the group (excluding the owner), and the last set of permissions for anyone else (excluding the owner and any members of the group).

However if the file is a directory, the read permission means that the user is permitted to read the file names in the directory file (because the directory file contains the file names), the write permission means that the user is allowed to add or remove files under the directory (because to add or to remove file under the directory would change the content of the directory file), and search permission means that the user is allowed to access files under that directory, eg, making a reference to a file name under that directory, or "enter" that directory using command `cd`.

When a process makes a request to access a file, the kernel decides whether to grant the access to the process by checking the user of the process using the following rule:

if the user is the owner of the file then
    check the owner permissions only
else if the user belongs to the group that owns the file then
    check the group's permissions only
else (the user is not the owner, nor in the group)
    check other's permissions only

Note: in the above algorithm, the term "user" and "group" means the *effective user* and *effective group* of the process that attempts to access the file. The Effective User ID (`euid`) and the Effective Group ID (`egid`) are the properties of a process. They are usually the same as the Real User ID (`uid`) and Real Group ID (`gid`) of the process (which are the `uid` and `gid` of the user that invokes the process). However, in some circumstances, `euid` and `egid` may be different from the real `uid` and real `gid`.

Example: Suppose that directory file named `testdir` has the following access permissions (in the form of `ls -l` output):

```
drwxr-xr--
```


Assuming the files `a.out` ***and*** `test.c` ***exist under directory*** `testdir`***, determine whether or not the following types of users are allowed to perform the operations listed below (explain why):***

owner:


```
% rm testdir/a.out
```

group:

```
% vi testdir/test.c
% rm testdir/a.out
```

others:

```
% ls testdir
% ls -l testdir
% vi testdir/test.c
```

The owner and the superuser may use the following systems calls to change the access permissions:

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

int chmod(const char *pathname, mode_t mode);
int fchmod (int filedes, mode_t mode);
```

Returns 0 if ok, or -1 on error

Examples:

```
if ((chmod("calendar", 0700) == -1)
     printf("Unable to change mode\n");

if ((fchmod(1, 0700) == -1)
     printf("Unable to change mode\n");
```

12. The `umask`

The `umask` value is a property of a process. It is used to automatically turn *off* certain file access permissions when the process creates a new file.

The `umask` value is often set in the shell initialization file such as `.login` or `.cshrc` for C shell, or `.bash_profile` or `.bashrc`, or `.profile` for Bash shell. The following shell command would set the `umask` value to `007` (octal number, which is equivalent to binary `000 000 111`):

```
umask   007.
```

This means turn off read, write and execute permissions for others regardless of the mode value given in `open` or `creat` system calls.

The umask value can be changed within a program using system call `umask`:


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

mode_t umask (mode_t  cmask);
```

Return the previous `umask` in the process

Example: create a new file `foo` with the exact permissions `0766`:

```
oldmask = umask(0);
fd = open ("foo", O_RDWR|O_CREAT, 0766);
umask (oldmask);   // restore old umask
```

13. Adding and Removing Filenames

A Unix file may have several different filenames. Some of these are *hard links* (the different file names in the same filesystem mapped to the same i-node) and others may be just symbolic links (a separate file containg the path to another file, allowing cross filesystem reference). At the shell prompt we use the command `ln` to add a new name to an existing file:

```
ln original_name new_name     (hard link)
ln -s orginal_name new_name   (symbolic)
```

We may remove a file name using the command `rm`. How do we add or remove a file name in a Unix program?

Creating hard links

We use the following system call to add a new name to an existing file (hard link):


```
#include  <unistd.h>
int link(const char *original_path,
            const char *new_path);
```
Return 0 if OK, or -1 on error

Example:

```
if (link("foo", "foo1")<0) {
      perror ("can't link to foo");
}
```

Note: the use of `perror` to report error is preferred over `printf` function. `perror` not only outputs a message of your choice but also a system error message to standard error. For instance if no file named `foo` existed in the current directory, the above program would output the following message:

```
cann't link to foo: No such file or
directory
```

The message following colon is the system error message.

Creating symbolic links

To create a symbolic link, we use the following call:

```
#include <unistd.h>
int symlink(const char *real_name,
                   const char *sym_name);
```

Return 0 if OK, or -1 on error

Read the content of a symbolic link file

When opening a file with `open` call on a symbolic link, the file opened will be the original file, not the symbolic link file. To read the content of the symbolic link file, you should use `readlink` call:


```
#include <unistd.h>
int readlink(const char *sympath,
       char *buffer, size_t bufsize);
```

Return the number of bytes in the link file, or -1 on error

Example:

```
#include <unistd.h>

main()
{
```

```
    char buf[100];
    int n;

    n = readlink("foo1", buf, 100);
    if (n<0) {
       perror ("cann't read link");
       exit(1);
    }

    buf[n]='\0';
    printf("foo1:%s\n", buf);
}
```

Remove file names

The following functions are used to remove file names:


```
#include <unistd.h>
int unlink(const char *pathname);
```

Return 0 if ok, or -1 on error

Change file names

The following functions are used to change file names:

```
#include <stdio.h>
int rename(const char *oldname,
                   const char *newname);
```

Return 0 if ok, or -1 on error

There is another standard C function `remove` that can be used to remove the filename. This function can also remove empty directory. For details consult the manual page of the function.

14. Obtaining information about a file

Most information stored in the i-nodes is obtained using `stat`, `fstat` or `lstat` system calls:

```
#include  <sys/types.h>
#include  <sys/stat.h>
int stat (const char *pathname,
                       struct stat *buf);
int fstat (int filedes,
                       struct stat  *buf);
int lstat (const char *pathname,
                       struct stat *buf)
```

Return 0 if ok, or -1 on error

Note: `lstat` is similar to `stat` except when the named file is a symbolic link. In that case, `stat` returns the information about the file referenced by the symbolic link, whilst `lstat` returns the information about the symbolic link file itself.

All three system calls return the information in a structure variable of `struct stat` type defined in `<sys/stat>` header file.

```
Struct stat
{
    mode_t    st_mode;     /* file type & mode */
    ino_t     st_ino       /* I-node number */
    dev_t     st_dev;      /* device number */
    *dev_t    st_rdev;     /* device number for special
                           file */
    nlink_t   st_nlink;    /* number of links */
    uid_t     st_uid;      /* user ID for owner */
    gid_t     st_gid;      /* group ID of group owner */
    off_t     st_size;     /* size in bytes, for regular
                           file */
    time_t    st_atime;    /* time of last access */
```

```
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last file status
                               change */
    blksize_   st_blksize;  /* best I/O blocks size */
    t
    Blkcnt_t   st_blocks    /* number of disk blocks
                               allocated */
};
```

*not specified in POSIX.1. Note: `st_rdev` is only meaningful if the file is a special device file.

Example 1. Print the types of files (adapted from Stevens p.76)

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

int main(int argc, char *argv[])
{
   int i;
   struct stat buf;
   char *p;

   for (i = 1; i < argc; i++) {
      printf("%s:  ", argv[i]);
      if  (lstat(argv[i], &buf) < 0) {
         perror("lstat error");
         continue;
      }

      if(S_ISREG(buf.st_mode))
           p = "regular file";
      else if (S_ISDIR(buf.st_mode))
           p = "directory";
      else if (S_ISCHR(buf.st_mode))
           p = "character special";
      else if (S_ISBLK(buf.st_mode))
           p = "block special";
```

```
      else if (S_ISFIFO(buf.st_mode))
           p = "fifo";
      else if (S_ISLNK(buf.st_mode))
           p = "symbolic link";
      else if (S_ISSOCK(buf.st_mode))
           p = "socket";
      else
           p = "**unknown file type";

      printf("%s\n", p);
   }

   exit(0);
}
```

Example 2: print access permissions

Use the predefined `st_mode` mask


```
if (buf.st_mode & S_IRUSR)
  printf(“user read”);

if (buf.st_mode & S_IWUSR)
  printf(“user write”);

if (buf.st_mode & S_IXUSR)
  printf(“usr execute”);

if (buf.st_mode & S_IRGRP);
  printf(“group read”);

. . . . . . . .

if (buf.st_mode & S_IXOTH)
  printf(“other execute”);
```

For details of these masks, see header file

```
/usr/include/sys/stat.h.
```

eg:

```
#define  S_IRUSR  00400
```

Bit-wise operation - an example:

```
st_mode  = 034755
```

st_mode:	011	100	111	101	101
S_IRUSR	000	000	100	000	000
st_mode & S_IRUSE:	000	000	100	000	000

15. File Systems

A file system (or filesystem, as is often spelled in many literatures) is a "tree" of files stored in a disk or disk partition. A file system is divided into many blocks of equal size. In a typical Unix file system, there are four distinct sections:

- **boot block** (block 0): bootstrap code (only one file system, the root file system, requires this code).

- **super block**: (block 1): information such as size of file system, time and date the file system was created, number of blocks reserved for i-nodes, and number of blocks reserved for data and directories, etc.

- **i-list**: (block 2 to n): a list of i-nodes. The number of I-nodes equals to the number of files that can be stored in the file system. Each file has one i-node.

- **directory and data blocks** (block n+1 onwards): blocks that store actual contents of regular files and directory files.

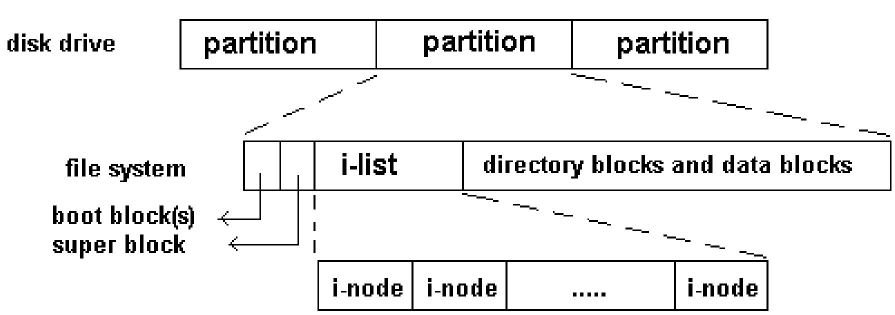


Figure 4.7 Disk drive, partitions and a filesystem

At the shell prompt, we use command `df` to list the file systems that are currently mounted. These separate file systems are grouped together to form a single file tree. The command `df` also shows the *mount point* of each file system. The root (superuser) can mount or unmount a non-root file system from such a file tree. If configured appropriately an ordinary user may also be able to mount and unmount an individual file system.

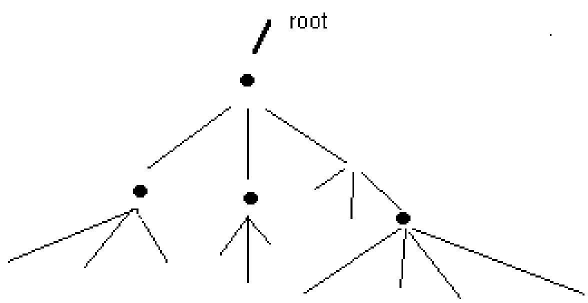
For example, on Linux system, if you have added the following line

```
/dev/fd0 /mnt/floppy auto noauto,users  0 0
```


in the file `/etc/fstab`, and have also added the required filesystem types in the file `/etc/filesystems`, eg:


```
vfat
msdos
ext2
```


any user may mount and unmount a vfat, msdos or ext2 filetsystem on the floppy drive. The system automatically probes the super block of the file system to find out its type.



16. Caching in File Systems

To improve the I/O efficiency the superblock of each mounted file system is cached in the kernel memory, so that updating the super block can be performed quickly. Similarly data transfer (read/write) between a process and a file is buffered in the kernel's memory so that the number of actual disk I/O can be significantly reduced.

Due to this reason, at any given moment the contents in a mounted file system may be inconsistent with what they are supposed to be. This is usually not a problem. However occasionally it may be necessary to flush out the data cached in the kernel's memory to the disk to force it to be consistent. For instance, when you shut down the machine using the shutdown command, it will perform this operation before it actually brings down the machine.

Two functions can be used to flush the buffer to file systems. The function `sync` flushes the buffers of all mounted file systems. The function `fsync` flushes out all data associated with a particular file.


```
#include <unistd.h>
void sync(void);
```

- `sync` returns when the flush operations are scheduled (but not necessarily completed)

```
int fsync(int filedes);
```

- `fsync` returns 0 after the operation has completed or -1 on error

There is a user command of the same name: `sync` as well as the command `update` which call the above two functions.

17. Special Files

(1) Peripheral devices, eg:
floppy disk drives
hard disk drives
keyboard
printers
serial ports
CD/DVD-ROMS
magnetic tapes
USB drives/SSDs

(2) On Unix, each device is represented by a file name under directory `/dev`

Examples:

```
/dev/ttyp0
/dev/rz0g
/dev/lp
```


(3) Each device is also represented by two device numbers – the major device number and the minor device number
- **major number**: the type of device. The major device number determines how the device should be accessed (which device driver)
- **minor number**: which device in the device type. This number identifies the individual device from that device type.

(4) Read/write operations can be performed on these device names as if they are regular files (permission allowed). For example,

```
ls -l > /dev/tty3
```

18. Reading Directories

Directories can be read by the following functions:

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

DIR *opendir  (const char *pathname);
```

Returns: a pointer if OK, or NULL on error

```
struct dirent *readdir (DIR *dp);
```

Returns: a pointer if ok, or NULL on error or when reaching the end of the directory stream


```
void rewinddir (DIR *dp);
```



```
int closedir (DIR *dp);
```


Returns: 0 if OK, or -1 on error

Note:

(1) **`opendir`**
Opens a directory stream, and set the current position to the first directory entry

(2) **`readdir`**
Reads the current directory entry and set the position to the next entry

(3) **`readdir`**
Returns NULL when it reaches the end of the stream

(4) **`rewinddir`**
Reset the current position to the first directory entry

(5) **`closedir`**
Must be called after directory reading is complete

The above functions are centred around the `dirent` structure. The data type usually contains the i-node number and the filename of a directory entry:

```
struct dirent
{
   ino_t  d_ino;      /* I-node number */
   char   d_name[];   /* null-terminated filename */
}
```


Other useful system calls or functions:

(1) create and remove directories:


```
mkdir
rmdir
```

(2) get and change current working directory:

```
getcwd
chdir
```

For details see the relevant manual pages of the functions.

Example: read directory entries

```
/*
 * t3ex1.c     - read directory
*/

#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char *argv[])
{
    char dirname[256];
    DIR *dp;
    struct dirent *direntp;

    if (argc == 1) /* list the current directory */
        strcpy(dirname, ".");
    else
        strcpy(dirname, argv[1]);

    if ((dp = opendir(dirname)) == NULL) {
        printf("Error in opening directory %s\n",
dirname);
        exit(1);
    }

    while (( direntp=readdir(dp)) != NULL)
        printf(" file name = %s, i-node number=%d\n",
                  direntp->d_name, direntp->d_ino);

   closedir(dp);
   exit(0);
}
```