UNIX File System

File System

- Unix file system is an hierarchical arrangement of directories and files. Everything starts in the directory called *root* whose name is the single character /.
- A directory is a file that contains filenames along with its attributes such as type of file (file or directory), size of file, owner of the file etc.

File System…

- Filename cannot have two characters: / and NULL character.
- Two file names are automatically created when a new directory is created. those are . (current directory name) and .. (parent directory name).
- A filename can have up to 255 characters.

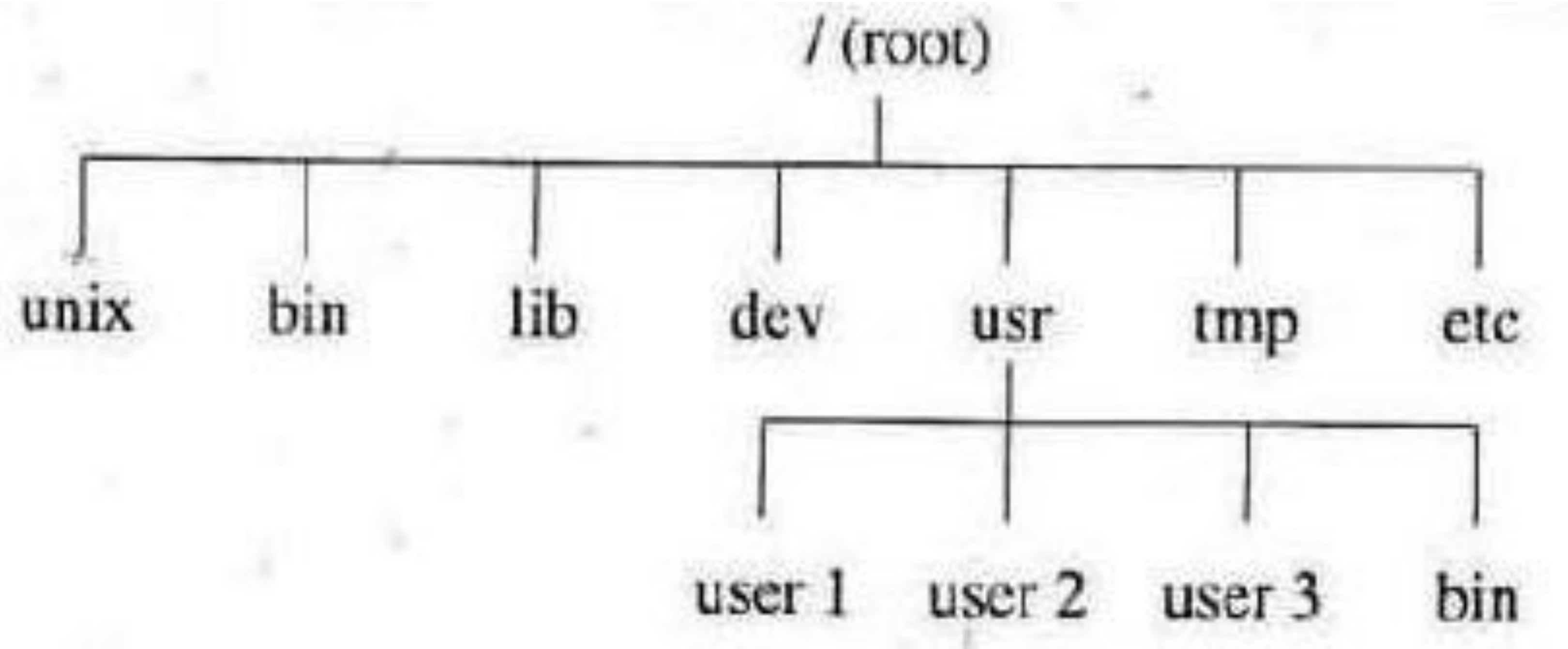
File system

- Everything in UNIX is either a file or a process.
- All utilities, applications, data in unix is stored as file. Even a directory is treated as file which contains several files.
- A process is an executing program identified by a unique PID (process identifier).
- A file is a collection of data. They are created by users using text editors, running compilers etc.

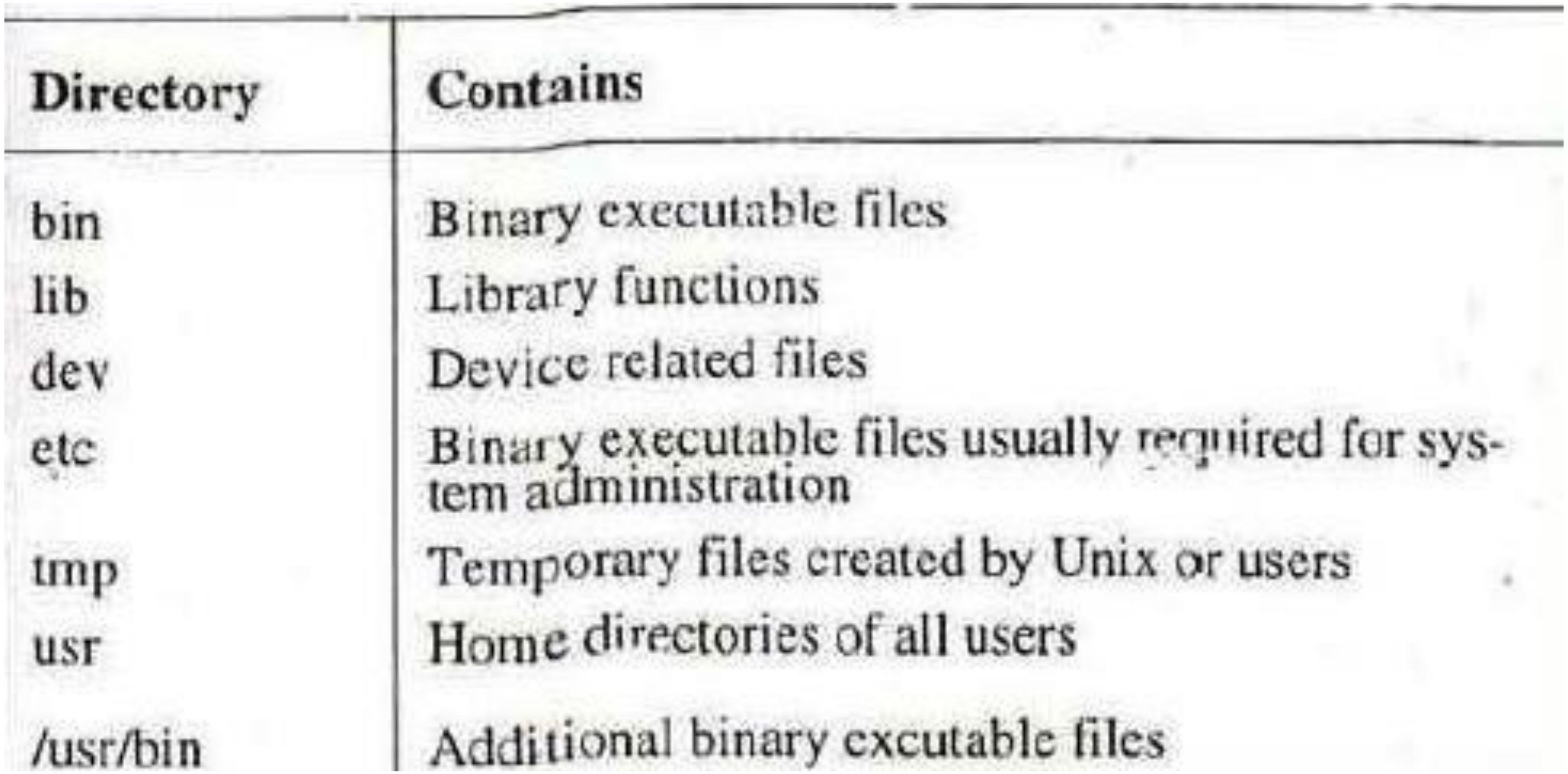
File system

- A sequence of one or more filenames, separated by slashes (/) and optionally starting with a slash forms a *pathname*.
- Pathname that begins with slash is called an *absolute path/*fully qualified pathname.
- Pathname , which are not starting with slash are called as *relative path*.
- The root directory also contains a file called unix which is kernel itself.

File system



File system



Directory	Contains
bin	Binary executable files
lib	Library functions
dev	Device related files
etc	Binary executable files usually required for system administration
tmp	Temporary files created by Unix or users
usr	Home directories of all users
/usr/bin	Additional binary excutable files

File system

- Relative pathname
  - It refers to files relative to current directory.
  - aaa/bbb/ccc refers to the file or directory ***ccc*** in the directory ***bbb***, in the directory ***aaa***, which must be a directory within the current working directory.
- Absolute Pathname
  - /usr/lib/lint refers to the file or directory ***lint*** in the directory ***lib***, in the directory ***usr***, which is in the root directory.

File system

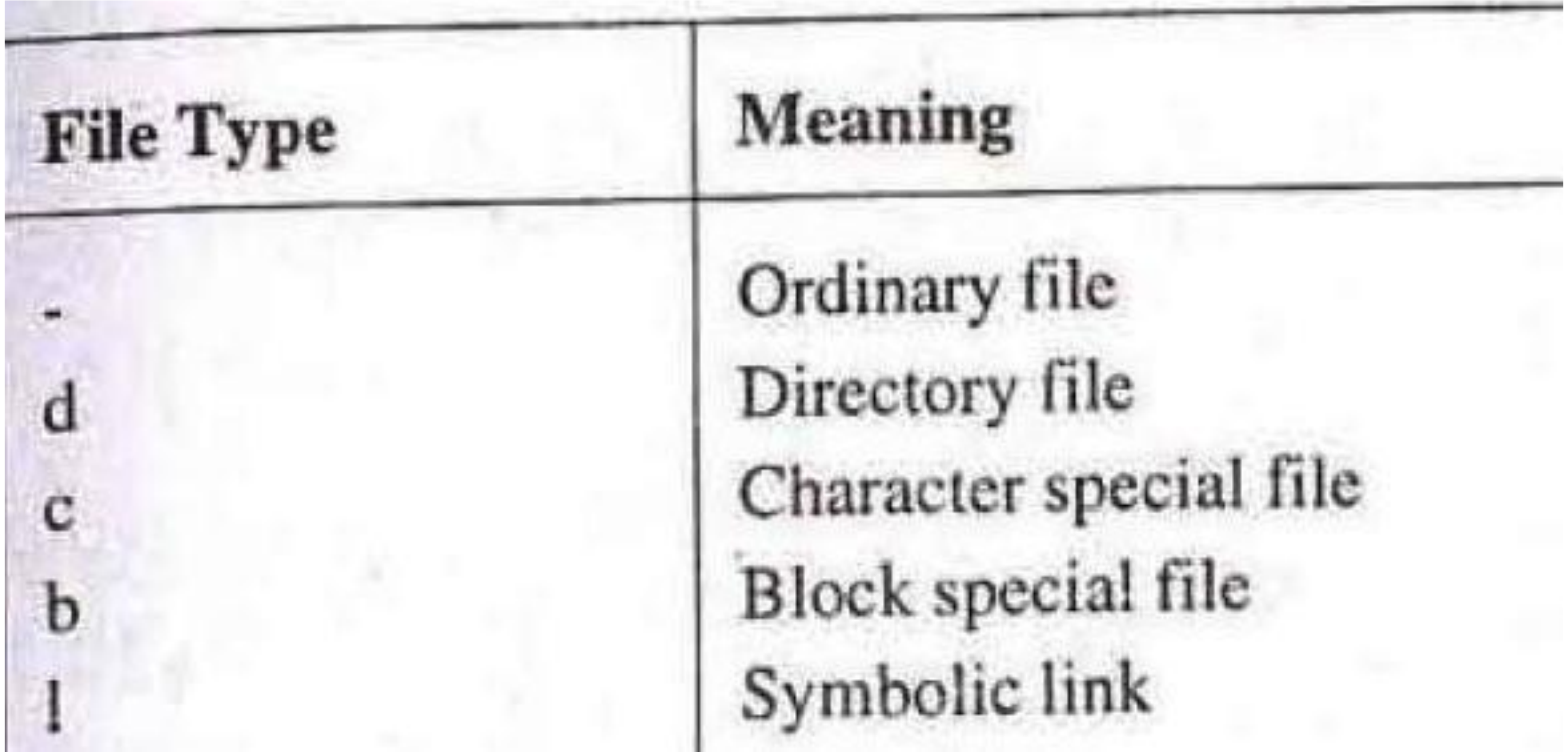
Salient features of UNIX file system:

- It has a hierarchical file structure.
- File can grow dynamically.
- File have access permission.
- All devices are implemented as a file.

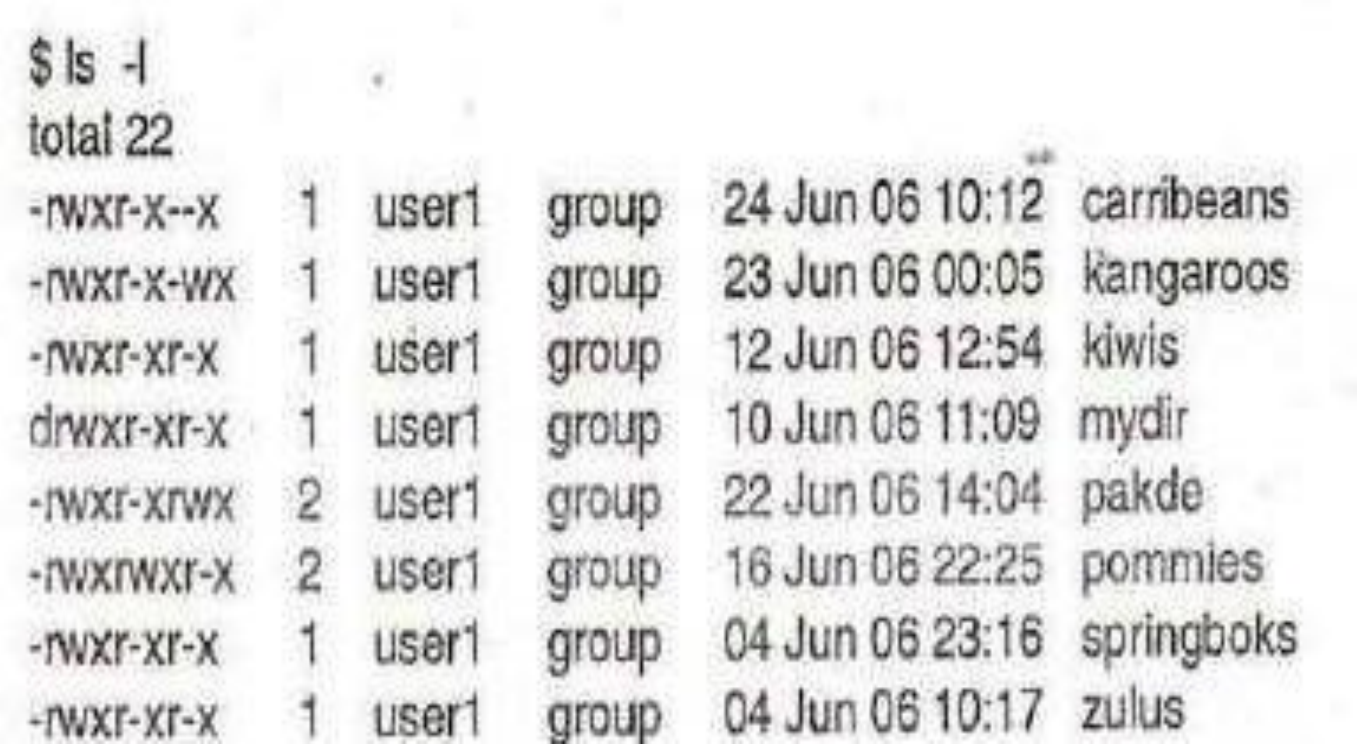
File system

- At any time, each process has an associated directory, called the ***current working directory***, that it uses for pathname resolution.
- A ***home directory*** is a file system directory on a multi-user OS containing files for a given user of the system.
- When you first login, your current working directory is your home directory.
- Your home directory has the same name as your user-name.

File system



File Type	Meaning
-	Ordinary file
d	Directory file
c	Character special file
b	Block special file
l	Symbolic link



```
$ ls  -l
total 22
-rwxr-x--x    1   user1    group    24 Jun 06 10:12   carribeans
-rwxr-x-wx    1   user1    group    23 Jun 06 00:05   kangaroos
-rwxr-xr-x    1   user1    group    12 Jun 06 12:54   kiwis
drwxr-xr-x    1   user1    group    10 Jun 06 11:09   mydir
-rwxr-xrwx    2   user1    group    22 Jun 06 14:04   pakde
-rwxrwxr-x    2   user1    group    16 Jun 06 22:25   pommies
-rwxr-xr-x    1   user1    group    04 Jun 06 23:16   springboks
-rwxr-xr-x    1   user1    group    04 Jun 06 10:17   zulus
```

File system

- The disk space allotted to a UNIX file system is made up of “blocks”, each of which are typically of 512 bytes.

- All the blocks belonging to file system are logically divided into four parts:
  - Boot Block
  - Super Block
  - Data Block
  - Inode Table

Boot Block

- The boot block represents the beginning of the file systems.
- The boot block located in the first few sectors of a file system.
- The boot block contains the initial bootstrap program used to load the OS

Super Block

- The super block describes the state of the file system:
  - ➢The total size of the partition.
  - ➢The block size.
  - ➢Pointers to a list of free blocks.
  - ➢The inode number of the root directory, etc.

Data Block

- It contains the actual file contents.
- All allocated block can belong to only one file in the file system.
- This block cannot be used for storing any other file’s contents unless the file to which it originally belonged is deleted.

Inode Table

- The information related to all the files is stored in an Inode Table on the disk.
- For each file, there is an Inode entry in the table.

Inode Table

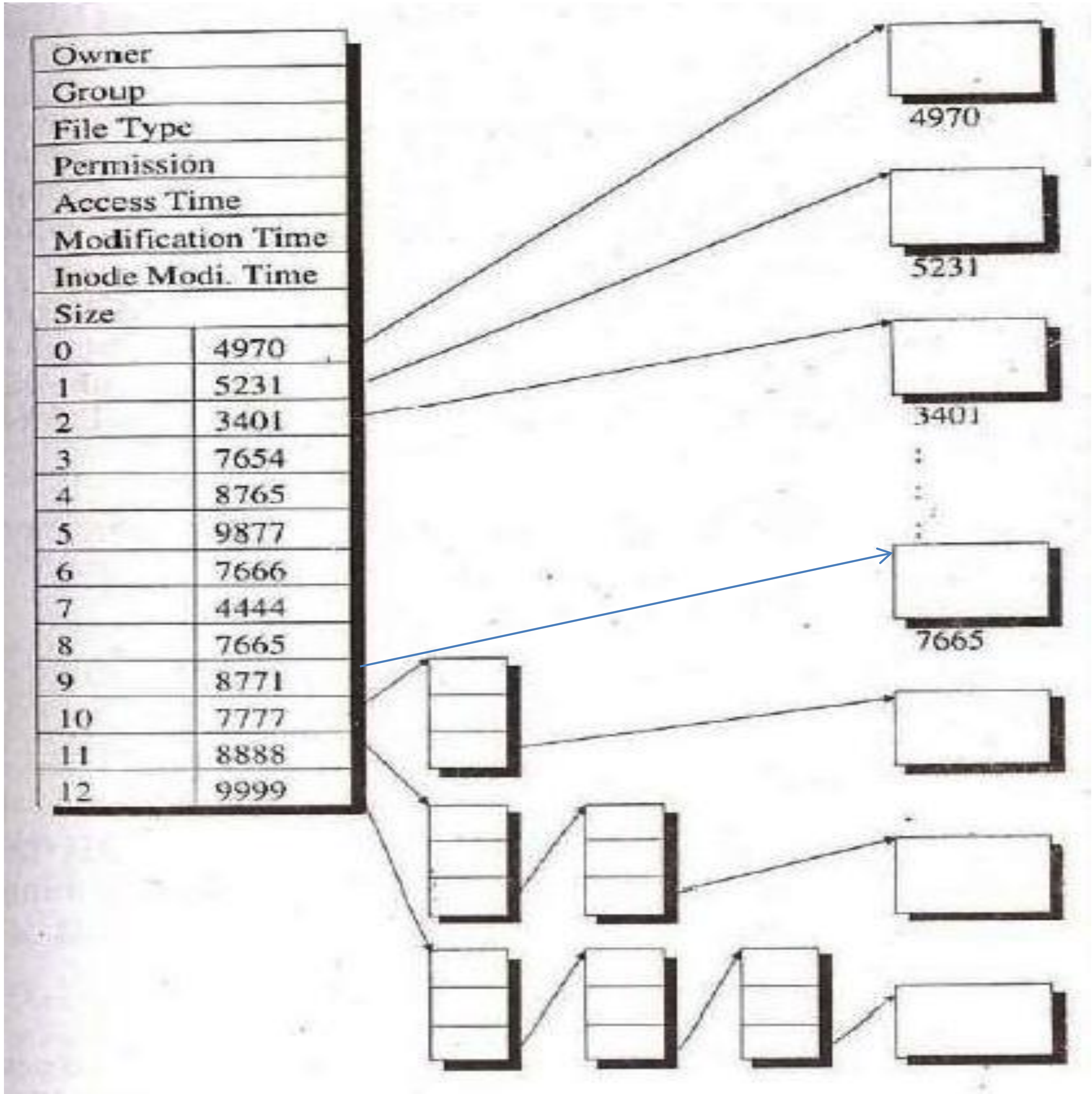
- Each entry is made up of 64 bytes and contains the following information of the files:
  - file ownership
  - file type
  - file access permissions
  - time of last access, and modification
  - number of links (aliases) to the file
  - pointers to the data blocks for the file
  - size of the file in bytes
- Information the Inode does not contain:
  - Path name of file

Inode Table

- Each Inode entry in the Inode Table consists of 13 addresses each, which specify completely where the contents of the file are stored on the disk.
  - These addresses are numbered 0 to 12
  - First ten addresses (0 to 9) points 1KB blocks on disk
  - For large file size, 10, 11 and 12 entries are used

Inode Table

- Block 0-9--> 10X1=10KB (direct)
- 10th block--> 256X1=256KB (single indirection)
- 11th block--> 256X256=64MB(double indirection)
- 12th block-->256X256X256=16GB (triple indirection)



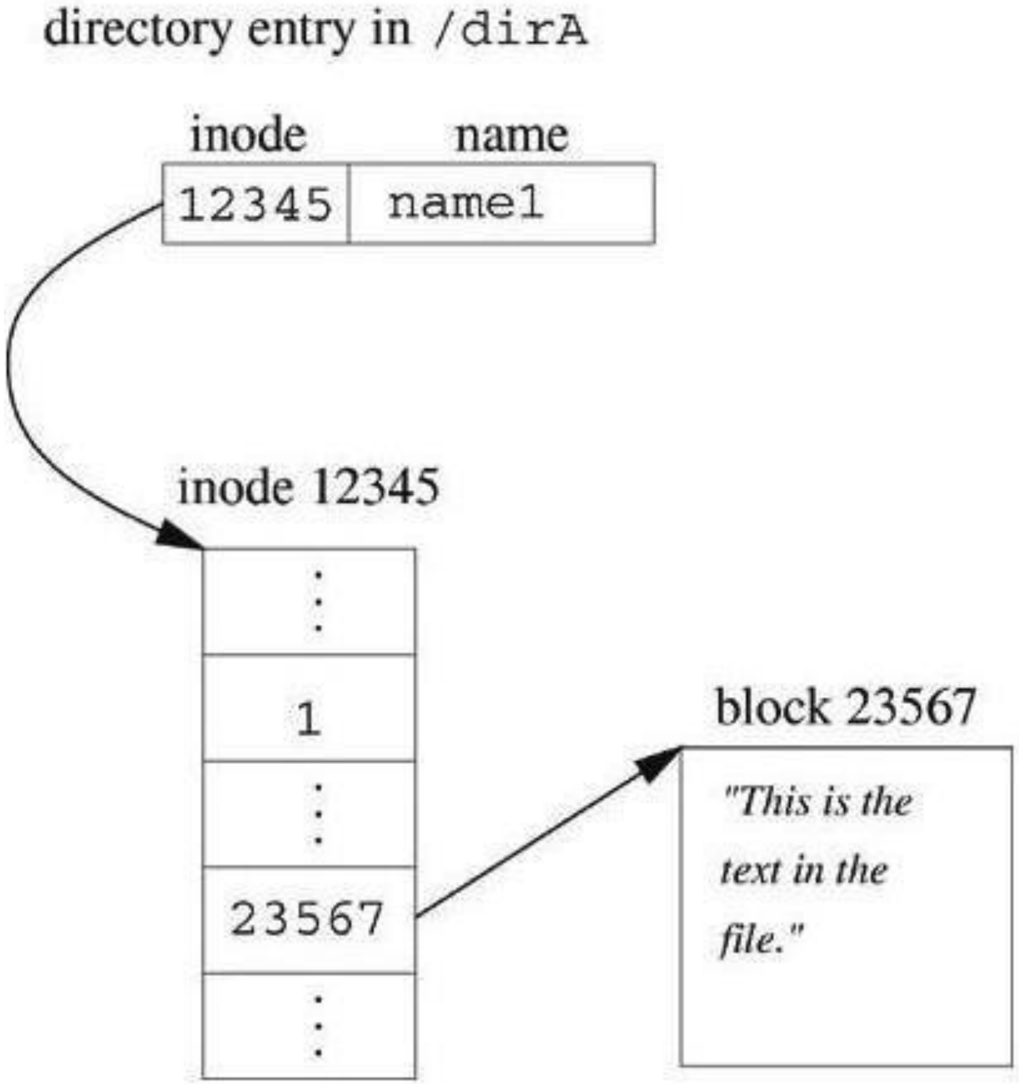
Owner	
Group	
File Type	
Permission	
Access Time	
Modification Time	
Inode Modi. Time	
Size	
0	4970
1	5231
2	3401
3	7654
4	8765
5	9877
6	7666
7	4444
8	7665
9	8771
10	7777
11	8888
12	9999

File Link

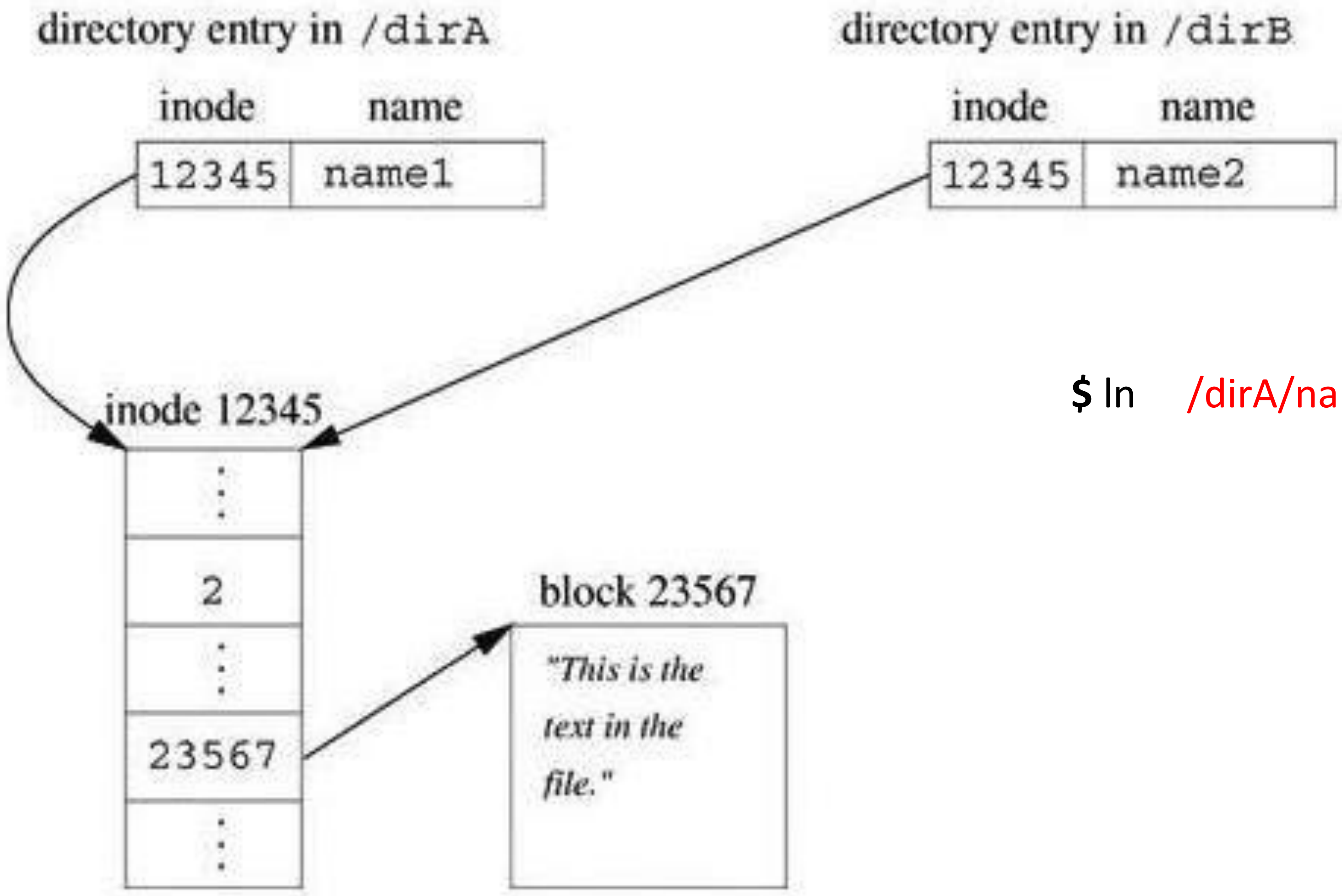
- A Hard link is a directory entry, which associates a filename with a file location.

- A soft link is a special type of file that contains a reference to another file or directory.

Directory entry, inode and data block for a simple file



Two hard links on the same file



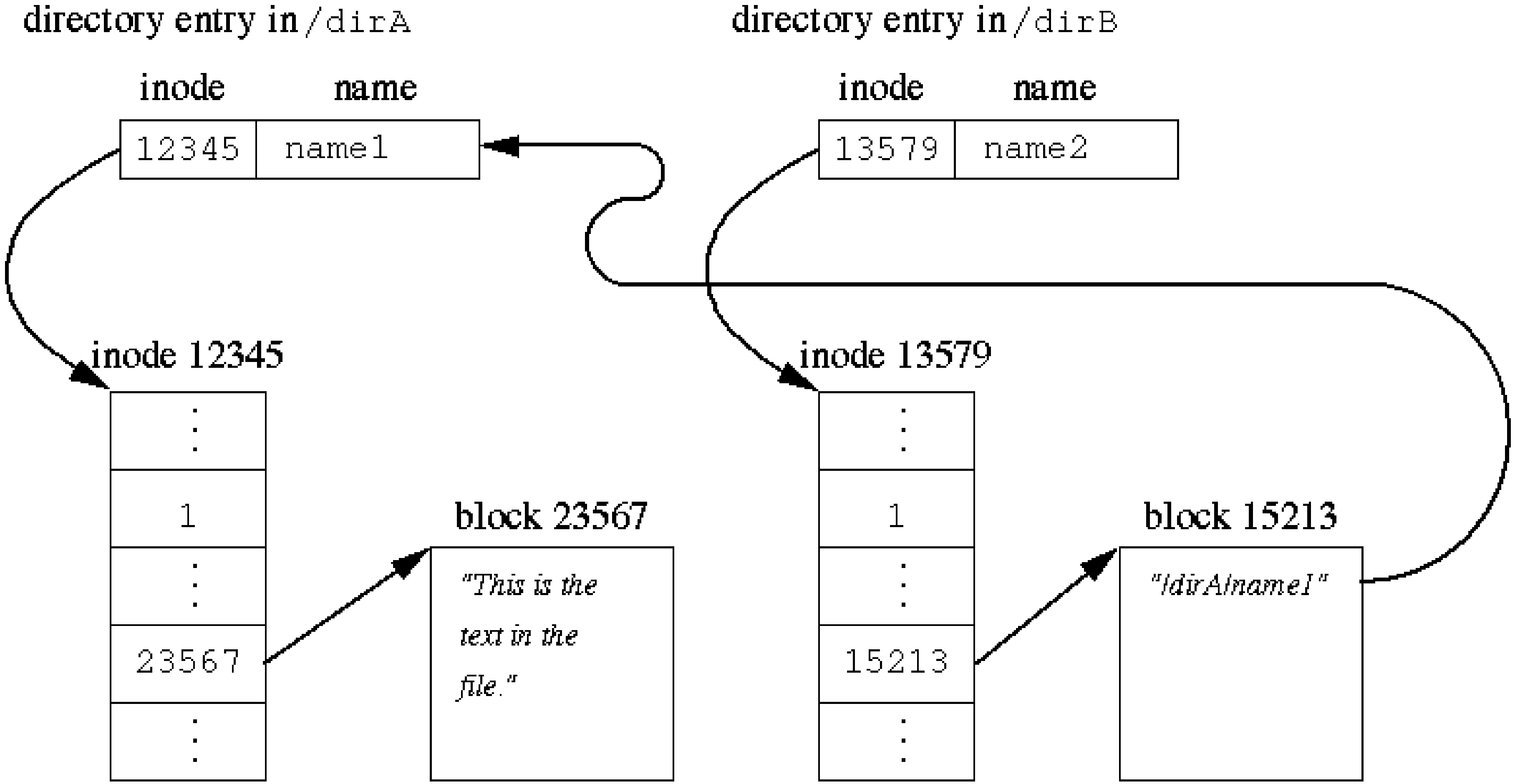
```
$ ln    /dirA/name1   /dirB/name2
```

Symbolic link

- A symbolic link is a special type of file that contains the name of another file.
- Symbolic lines are created with the command:

```
$ ln   -s  /dirA/name1   /dirB/name2
```

Symbolic link



creat() system call

Prototype for the creat() system call

```
int creat(file_name, mode)
char *file_name;
int mode;
```

- Where file_name is pointer to a null terminated character string that names the file.
- mode defines the file's access permissions.
- If the file named by file_name does not exist, the UNIX system creates it with the specified mode permissions.
- If the file does exist, its contents are discarded and the mode value is ignored. The permissions of the existing file are retained.

mode

Mode argument as defined in /usr/include/sys/stat.h:

- #define S_IRWXU 0000700 /* -rwx------ */
- #define S_IREAD 0000400 /* read permission, owner */
- #define S_IWRITE 0000200 /* write permission, owner */
- #define S_IEXEC 0000100 /* execute/search permission, owner */
- #define S_IRGRP 0000040 /* read permission, group */
- #define S_IROTH 0000004 /* read permission, other */

open()

open a file for reading, writing, or reading and writing

- **Prototype**

```
#include <fcntl.h>
int open(file_name, option_flags [ , mode])
char *file_name;
int option_flags, mode;
```

- where file_name is a pointer to the character string that names the file.
- option_flags represent the type of channel
- mode defines the file's access permissions if the file is being created.

open()

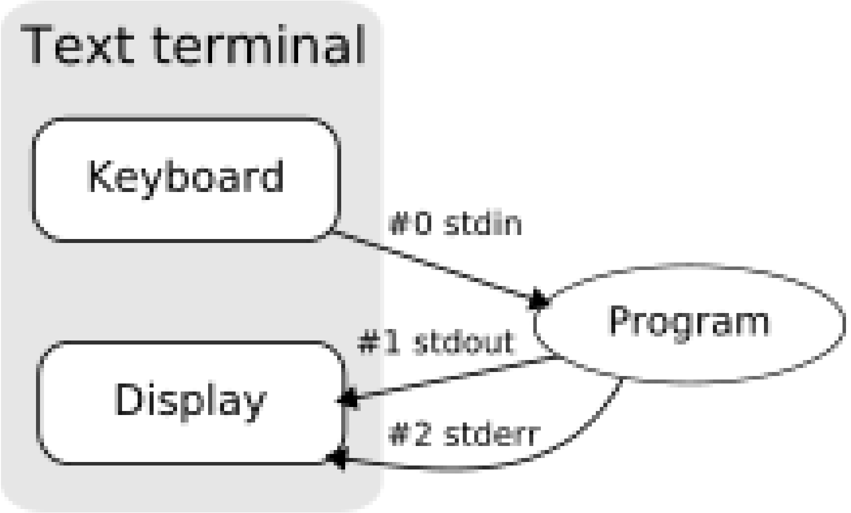
- If the open() system call succeeds, it returns a small non-negative integer called a file descriptor that is used in subsequent I/O operations on that file.
- If open ( ) fails, it returns –1 (**#include <errno.h>)**.

Value	Meaning
0	Standard Input
1	Standard Output
2	Standard Error

Code	Meaning
EBADF	Bad file descriptor
EACCES	Permission denied
EBUSY	Device or resource busy

open()

- The printf () library function always sends its output using file descriptor 1,
- scanf () always reads its input using file descriptor 0, which is the display screen.



option_flags for open()

The allowable option_flags as defined in "/usr/include/fcntl.h"

- #define O_RDONLY /* Open the file for reading only */
- #define O_WRONLY /* Open the file for writing only */
- #define O_RDWR /* Open the file for both reading and writing*/
- #define O_TRUNC /* Truncate file size to zero if it already exists */
- #define O_CREAT /*Create the file if it doesn't already exist */

- #define O_RDONLY 0   /* Open the file for reading only */
- #define O_WRONLY 1   /* Open the file for writing only */
- #define O_RDWR   2   /* Open the file for both reading and writing*/
- #define O_APPEND 010   /* append (writes guaranteed at the end) */
- #define O_CREAT 00400 /*open with file create (uses third open arg) */
- #define O_TRUNC 01000   /* open with truncation */
- #define O_EXCL   02000   /* exclusive open */

Bitwise Oring

EXCLUSIVE

close()

To close a file, use the close() system call.

PROTOTYPE

- int close(file_descriptor)
- int file_descriptor;
- file_descriptor identifies a currently open channel.
- close() fails if file_descriptor does not identify a currently open channel.
- If successful close ( ) returns zero, otherwise it returns –1.

read()

- int read (int fddd, char *buff, int count)
- read ( ) copies count bytes from the file referenced by the file descriptor fddd into the buffer buff.
- The bytes are read from the current file position, which is then updated accordingly.

read()

- read ( ) copies as many bytes from the file as it can, up to the number specified by count, and returns the number of bytes actually copied.
- If a read ( ) is attempted after the last byte has already been read, it returns 0, which indicates end-of-file.
- If successful, read ( ) returns the number of bytes that it read; otherwise, it returns –1.

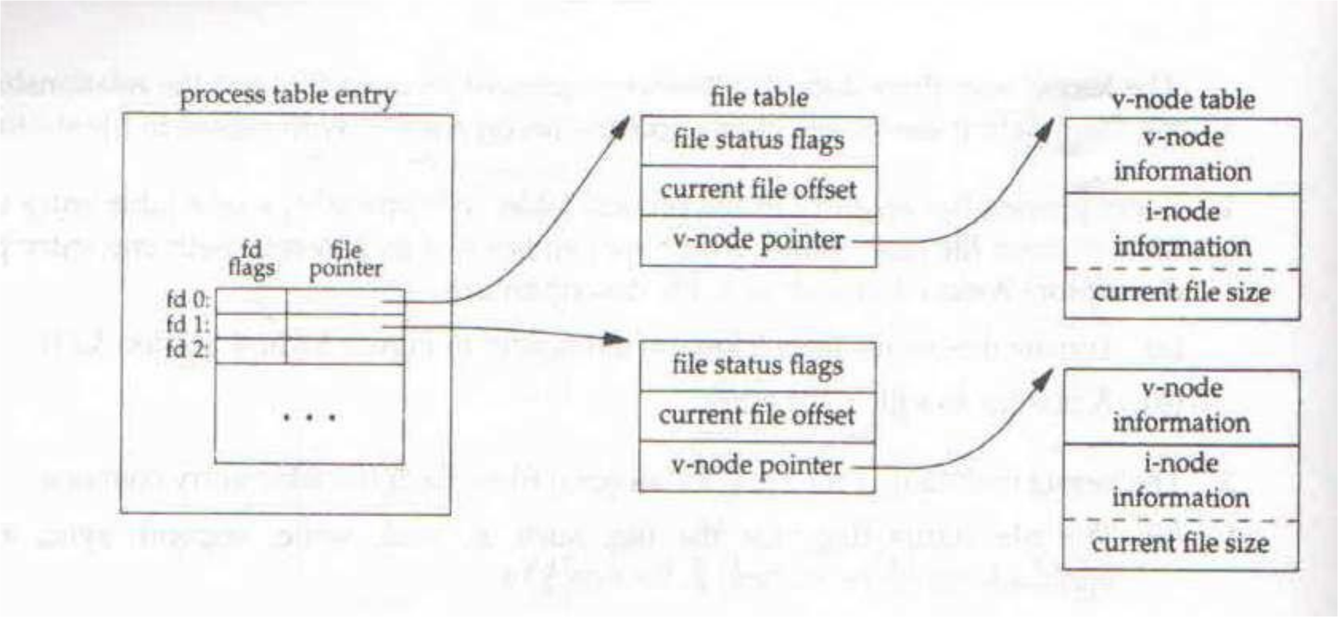
write()

- int write (int fddd, char *buff, int count)
- write ( ) copies count bytes from a buffer buff to the file referenced by the file descriptor fddd.
- The bytes are written at the current file position, which is then updated accordingly.

write()

- If the O_APPEND flag was set for fddd, the file position is set to the end of the file before each write.
- write ( ) copies as many bytes from the buffer as it can, up to the number specified by count, and returns the number of bytes actually copied.
- If the returned value is not count, then the disk probably filled up and no space was left.

File Data Structure



Courtesy: Stevens and Rago. “advance programming in unix environment”, 2nd ed., Addison- Wesley

lseek()

- long lseek (int fddd, long offset, int where)
- lseek ( ) allows you to change a descriptor’s current file position.
- fddd is the file descriptor.
- ***offset*** is a long integer, and ***where*** describes how offset should be interpreted.

lseek()

- If successful, lseek ( ) returns the current file position; otherwise, it returns –1.
- The three possible values of ***where*** are defined in "/usr/include/sys/file.h", and have the following meaning:

Value	Meaning
SEEK_SET	offset is relative to the start of the file.
SEEK_CUR	offset is relative to the current file position.
SEEK_END	offset is relative to the end of the file.

```
#include <stdio.h>
#include <sys/types.h> /*defines types used by sys/stat.h*/
#include <sys/stat.h>   /* defines S_IREAD & S_IWRITE*/
int main()
{
int fd;
fd = creat("datafile.txt", S_IREAD | S_IWRITE);
if (fd == -1)
    printf("Error in opening datafile.txt\n");
else
    {
 printf("datafile.txt opened for read/write access\n");
 printf("datafile.txt is currently empty\n");
     }
close(fd);
exit (0);
}
```

Output

```
datafile.txt opened for read/write access
datafile.txt is currently empty
```

```
#include <fcntl.h>              /* defines options flags */
#include <sys/types.h>     /* defines types used by sys/stat.h */
#include <sys/stat.h>      /* defines S_IREAD & S_IWRITE  */
char message[] = "Hello world";
void main()
{
int fd;
char buffer[80];
fd = open("datafil3.txt", O_RDWR | O_CREAT ,  S_IREAD | S_IWRITE);
if (fd != -1)
{
printf(“\n datafil3.txt opened for read/write access\n");
write(fd, message, sizeof(message));
lseek(fd, 0L, 0);     /* go back to the beginning of the file */
if (read(fd, buffer, sizeof(message)) == sizeof(message))
printf(“\n\"%s\" was written to datafile3.txt\n", buffer);
else
printf(“\n*** error reading datafile3.txt ***\n");
close (fd);
}
else
printf(“\n*** datafile3.txt already exists ***\n");
}
```

- **datafil3.txt opened for read/write access**
- **"Hello world" was written to datafile3.txt**

dup()

- The dup() system call duplicates an open file descriptor and returns the new file descriptor.
- The prototype:

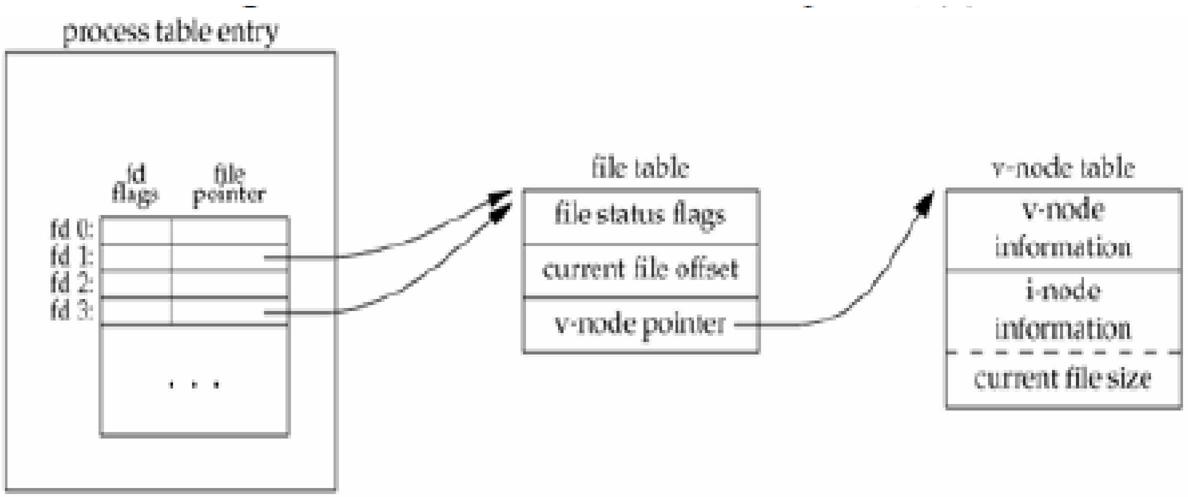
  ```
  int dup(file_descriptor)
  int file_descriptor;
  ```
- file_descriptor is the file descriptor describing the original I/O channel returned by creat(), open() system calls.

dup()

```
#include <stdio.h> #include <fcntl.h> #include <sys/types.h>
#include <sys/stat.h>
void main()
{   int fd, fd1;
fd = open("dup.txt", O_WRONLY | O_CREAT, S_IREAD | S_IWRITE );
printf("\n\n original fd=%d\n\n", fd);
if (fd == -1)
{  printf(“\n\n ERROR\n\n");
exit (1);   }
 close(1);      /* close standard output  */
 fd1=dup(fd);  /* fd will be duplicated  */
printf("\n\n fd after dup()=%d\n\n", fd1);
close(fd);       /* close the extra slot */
printf("Hello, world!\n");    /* should go to file dup.txt */
exit (0);         /* exit() will close the files */
}
```

```
original fd=3
Contents of dup.txt
fd after dup()=2
Hello, world!
```

Kernel data structure after dup(1)



Courtesy: Stevens and Rago. “advance programming in unix environment”, 2nd ed., Addison- Wesley

dup2()

int dup2(int oldfd,   int newfd)

- dup2(): an existing file descriptor *oldfd* is duplicated as file descriptor *newfd*.
- If the file corresponding to descriptor newfd is open, then it is closed.
- the original and copied file descriptors share the same file pointer and access mode just like in dup()

dup/dup2 system call

- They both return the index of the new file descriptor if successful, and –1 otherwise.
- dup/dup2 duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe.
- The call fails if the argument is bad (not open) or if 20 file descriptors are already open.