File Management

lseek() System call

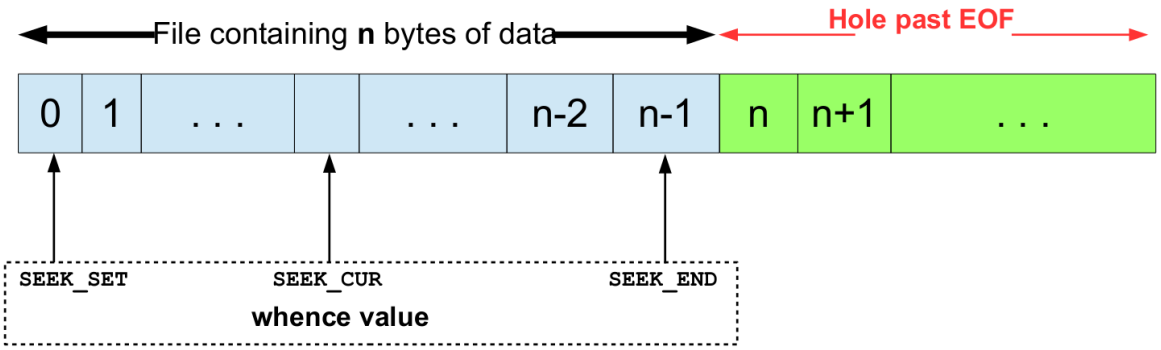
off_t lseek(int fd,off_t offset,int whence);

For each open file, the kernel records a file offset, also called current file offset (cfo), which is there in the SWFT. This is the location in the file at which the next read() or write() will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0

The file offset is set to point to the start of the file when the file is opened (unless the O_APPEND option is specified) and is automatically adjusted by each subsequent call to read() or write() so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive read() and write() calls progress sequentially through a file

The lseek() system call adjusts the file offset of the open file referred to by the file descriptor fd, according to the values specified in offset and whence. On success, returns the resulting offset location and -1 on failure

Interpreting whence argument of lseek()



Examples

```
off_t posn;
posn = lseek(fd, 54, SEEK_SET);
posn = lseek(fd, +/-54, SEEK_CUR);
posn = lseek(fd, +/-54, SEEK_END);
```

The directive “whence” can take following five values:

WHENCE		Description
SEEK_SET	0	The cfo is set `offset` bytes from the beginning of the file
SEEK_CUR	1	The cfo is set `offset` bytes from current value of cfo
SEEK_END	2	The cfo is set `offset` bytes from the end of the file
SEEK_HOLE	3	The cfo is set to start of the next hole greater than or equal to `offset`
SEEK_DATA	4	The cfo is set to start of the next non-hole (i.e., data region) greater than or equal to `offset`

rename() Function

int rename(const char*oldpath,const char* newpath);

A programmer can rename a file or a directory with the rename() library function

A sample code snippet that renames a file named file1.txt to file2.txt in the present working directory is shown below:

```
if(rename("file1","file2") == -1)
perror("rename(1)");
```

remove() and unlink()

int remove(const char *pathname);
int unlink(const char* pathname);

Remove is a library call that deletes a name from file system. It calls unlink() for files and rmdir() for directories

However, if any process has this file open currently, the file won't be actually erased until the last process holding it open closes it. Until then it will be removed from the directory (i.e., ls won't show it), but not from disk

When a file is deleted, the OS Kernel performs following tasks:
i. Frees the inode number associated with that file
ii. Frees all the data blocks associated with that file and add them to the list of free blocks
iii. Delete the entry from the directory containing that file

The metadata of the file is still there in the inode block and the data of the file in its data blocks (U just need to know how to access those blocks)

Symlink and link Function

int symlink(const char* oldpath, const char* newpath);
int link(const char* oldpath, const char* newpath);

The link() and symlink() functions are used to create a hard link and a soft link to a file

chown ,fchown and lchown Function

int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *linkname, uid_t owner,gid_t group);

These system calls change the owner and group of the file specified by path or file descriptor

If owner or group is specified as -1, then that ID is not changed

Only a process with super user privileges can use these functions to change any file user ID and group ID

However, if a process effective user ID matches a file user ID and its effective group ID, the process can change the file group ID only (Will discuss this later)

lchown() is like chown(), but does not dereference symbolic links

chmod and fchmod System Call

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

These two functions allow us to change the file access permissions for an existing file

The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened using its file descriptor

The mode is the same as discussed in the flags argument of open()

Following code snippet will give the owner read and write permissions to the file and deny access to all other users

```
if(chmod("file.txt",S_IRUSR|S_IWUSR) == -1){
perror("chmod"); exit(1);}
```

umask Function

mode_t umask(mode_t mask);

The umask() function sets the file mode creation "mask" for the process and returns the previous value

Remember the mask value of a process is the same as that of its creating shell, i.e. its parent. (mask value is inherited after fork)

The file mode creation mask is used whenever the process creates a new file or a new directory

```
umask(0077);
int fd = open("myfile.txt",O_CREAT|O_RDWR,0633);
```

access() System Call

int access(const char *pathname, int mode);

The access() system call determines whether the calling process has access permission to a file or not and it can also check for file existence as well

The mode argument is a bit mask consisting of one or more of the permission constants shown in the table below:

If a process has all the specified permissions the return value is 0, otherwise the return value is -1 & sets errno to EACCES

The open() system call performs its access tests based on the EUID and EGID, while the access() system call bases its tests on the real UID & GID

Mode	Description
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission
F_OK	Test for existence of file

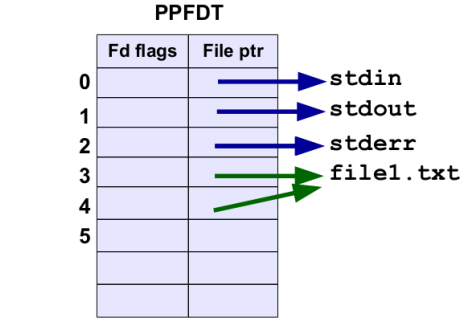
I/O Redirection using dup()

int dup(int oldfd);

The dup() call takes oldfd, an open file descriptor, and returns a new descriptor that refers to the same open file description

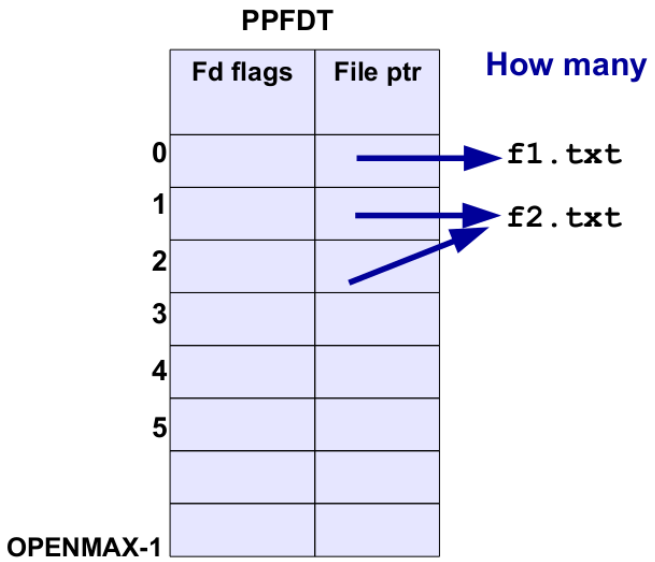
The old and the new descriptor both point to the same entry in the SWFT. After a successful return from these function , old and new fd's can be used interchangeably

The new descriptor is guaranteed to be the lowest unused file descriptor.



Facts about I/O Redirection on the Shell

$ cat 0< f1.txt 1> f2.txt 2>&1



We know that dup() call guarantees that the new descriptor returned is the lowest unused file descriptor

If we run the following LOCs, the open call will return 3, the dup call will return the lowest unused descriptor which will be zero. So finally descriptor zero points to the opened file instead of stdin

```
fd = open(...);
close(0);
newfd = dup(fd);
```

To make the above code simpler, and to ensure we always get the file descriptor we want, we can use dup2()

dup2() System call

int dup2(int oldfd, int newfd);

The dup2() system call makes a duplicate of the file descriptor given in oldfd using the descriptor number supplied in newfd

If the file descriptor specified in newfd is already open, dup2() closes it first

We can simplify the preceding calls to close(0) and dup(fd) on previous slide to the following:
dup2(fd, 0);

A successful dup2() call returns the number of the duplicate descriptor (i.e., the value passed in newfd)

If oldfd is a valid file descriptor, and oldfd and newfd have the same value, then dup2() does nothing—newfd is not closed, and dup2() returns the newfd

dup3() System call

int dup3(int oldfd, int newfd, int flags);

The dup3() system call performs the same task as dup2(), but adds an additional argument, flags, that is a bit mask that modifies the behavior of the system call

At the time of this writing, dup3() supports one flag, O_CLOEXEC, which causes the kernel to enable the close-on-exec flag (FD_CLOEXEC) for the new file descriptor

The dup3() system call is new in Linux 2.6.27, and is Linux-specific

Examples: Input Redirection

Method 1: close-open (stdinredir1.c)

```
close(0);
fd = open("/etc/passwd", O_RDONLY);
```

Method 2: open-close-dup-close (stdinredir2.c)

```
fd = open("/etc/passwd", O_RDONLY);
close(0);
newfd = dup(fd);
close(fd);
```

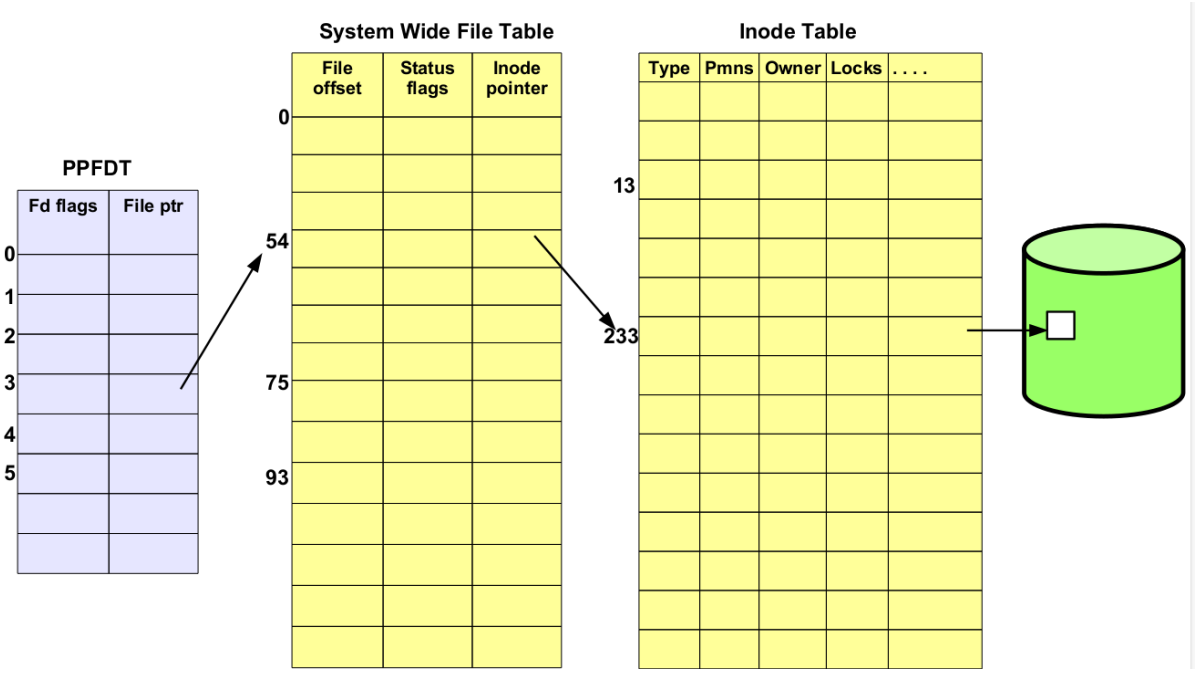
Method 3: open-dup2-close (stdinredir3.c)

```
fd = open("/etc/passwd", O_RDONLY);
newfd = dup2(fd, 0);
close(fd);
```

fcntl() System Call

What fcntl() can do?

int fcntl(int fd,int cmd, long arg);



fcntl() (Duplicate file descriptor)

The fcntl() system call can be used instead of dup() to return a duplicate file descriptor of an already opened file. The second argument passed to fcntl() for this purpose is F_DUPFD. It will return the lowest-numbered available file descriptor greater than or equal to the third argument

```
int fd = open("/etc/passwd", O_RDONLY);
printf("The first file descriptor is %d\n",fd);
int rv = fcntl(fd, F_DUPFD, 54);
printf("Duplicate file descriptor is %d\n",rv);
```

fcntl() (Get file status flags)

The fcntl() system call can be used to get the file status flags of an already opened file from SWFT. For example suppose you have opened a file and want to check the file access mode flags (O_RDONLY, O_WRONLY, O_RDWR). The second argument passed to fcntl() for this purpose is F_GETFL and the third argument is ignored. It will return all the file status flags in an integer variable which when bitwise anded with the O_ACCMODE constant will tell you about the permissions

```
int fd = open(“file”, O_RDONLY);
int flags = fcntl(fd, F_GETFL, 0);
flags = flags & O_ACCMODE;
if (flags == O_RDONLY) printf("read only\n");
```

fcntl() (Set file operating mode flags)

O_APPEND flag is used to ensure that each call to write() implicitly includes an lseek to the end of the file. Moreover, the kernel combines lseek() and write() into an atomic operation. Suppose you forgot to set this flag while making the open() call. Now if you have already opened a file and want to set O_APPEND flag, you can do that with fcntl() system call with a simple three-step procedure:

```
int flags = fcntl(fd, F_GETFL, 0);          //get settings
flags = flags | O_APPEND;                   //modify settings
fcntl(fd, F_SETFL, flags);                  //set them back
```

fcntl() (Set file operating mode flags)

O_SYNC flag is used to turn off disk buffering. It tells the kernel that call to write() should return only when the bytes are written to the actual hardware rather than the default action of returning when the bytes are copied to a kernel buffer. However, setting O_SYNC eliminates all the efficiency kernel buffering provides. Suppose you want to set this flag, but forgot to set it while making the open() call. Now if you have already opened a file and want to turn off Kernel disk buffering, you can do that with fcntl() system call with a simple three-step procedure:

```
int flags = fcntl(fd, F_GETFL, 0);          //get settings
flags = flags | O_SYNC;                     //modify settings
fcntl(fd, F_SETFL, flags);                  //set them back
```

File / Record Locking

Types of Locking Mechanisms:

Advisory locks:
Kernel maintains knowledge of all files that have been locked by a process. But it does not prevent a process from modifying that file. The other process can, however, check before modifying that the file is locked by some other process. Thus advisory locks require proper coordination between the processes

Mandatory Locks:
are strict implications, in which the kernel checks every read and write request to verify that the operation does not interfere with a lock held by a process. Locking in most UNIX machines is by default advisory. Mandatory locks are also supported but it needs special configuration

Types of Advisory Locks:

Read Locks/Shared Locks:
Locks in which you can read, but if you want to write you'll have to wait for everyone to finish reading. Multiple read locks can co-exist

Write Locks/Exclusive Locks:
Locks in which there is a single writer. Everyone else has to wait for doing anything else (reading or writing). Only one write lock can exist at a time.

fcntl() (File/Record Locking)

int fcntl(int fd,int cmd, struct flock* lock);

The fcntl() system call can be used for achieving read/write locks on a complete file or part of a file

To lock a file the second argument to fcntl() should be F_SETLK for a non-blocking call, or F_SETLKW for a blocking call

The third argument to fcntl() is a pointer to a variable of type struct flock (See its details in man page)

Locks acquired using fcntl() are not inherited across fork(). But are preserved across execve()