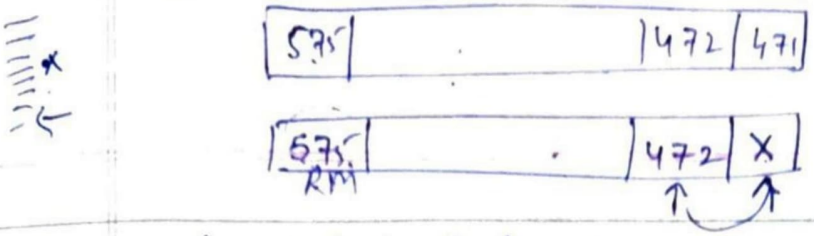
470




Linked List of free blocks vs Array of free Inodes —:
- Free inode determinable by inspection
-
- More frequent consumption of disk blocks than inodes.

Filesystem System Calls —

File System Commands —:

1) mkdir (create directory)
2) touch (create file)
3) cat (read from file)
4) cat / nano (write to file)
5) ls (display dir. contents)
6) ls -li (display dir. contents + perms)
7) cat >> (append to file).
8) f1 > f2 > prg > f3 (merge 2+ files)

System Calls for Files. —:

- Creation : creat, mknod (normal & spl. files)
- Access : open (open for access), read (read data), lseek (move ptr for read pos.), write (write data), close (close & free resources)
- Inode Manipulation : chmod (change perms), ~~chmod~~ chown (change users), fstat, stat (view inode information).
- Advanced : pipe, dup.
- Tree Visibility : ~~link, unlink~~ mount, umount.
- Change Structure : link, unlink

1) open () : <fcntl.h>

int open (const char* path, int oflag, /* mode_t mode */);

Arguments : path : path to file to open

oflag : mode to open file (open flag options)

O_RDONLY : Read only

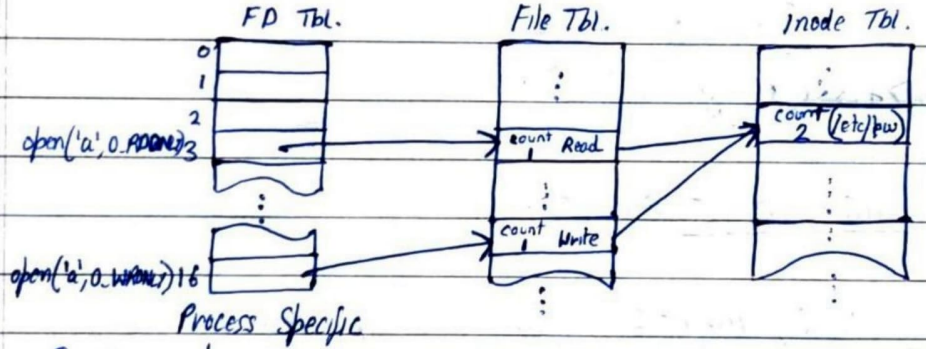
O_WRONLY : Write only

O_RDWR : Read / Write

O_EXEC : Execute

Returns : file descriptor, int. (-1 if failed)

× Kernel maintains a global file ~~descriptor~~ table for files in use by user-level processes, & a file table containing information on open mode, current read & write offset, reference count & link to inode table's inode entry.




× Each process maintains a user-level file descriptor table containing references to entries in file table (global). 0, 1 & 2 FDs are reserved for STDIN, STDOUT & STDERR respectively.

× Opening the same file creates multiple file table entries (if different open mode) but the same inode reference is used, & ref-count is updated.

× Open file : Map path to inode (namei), fail if non-existent or error. If success, create ~~file~~ entry in memory ~~to~~ inode, create user FD entry & set ref. to file table entry.

If truncate mode, free disk blocks (free), followed by unlocking of inode (locked by namei) & return user FD.

x Note : While same inode may be referred by multiple file table entries, each file entry is seperate (even for same mode) & referred by a unique FD entry. : Each global file entry is different corresponding to different FD links.
- Ref. count of file entry in global file table used by system calls like dup().

2) Creat() : <fcntl.h>

int creat ( const char* ~~path~~, mode_t mode);

Arguments :

path : path to create file over.

mode : file access permissions (octal value)

| | | | |
|---|---|---|---|
| S_IREAD | ; read perm, owner | (0000400) | <sys/stat.h> |
| S_IWRITE | ; write perm, owner | (0000200) | |
| S_IRGRP | ; read, group | (0000040) | |
| S_IROTH | ; ~~write~~ read, others | (0000004) | |

Returns : file descriptor, int (-1 on error, failure)

Creation : Fetch inode for file name (namei) & if inode exists & permission not available, fail. If inode does not exist, then allocate a free inode (ialloc), create directory entry in parent's inode & create a new file table entry. If inode existed for file, then remove all disk blocks if requested. Finally unlock inode & create FD over file table entry & return FD.

3) read() : <unistd.h>

ssize_t read (int fd, void* buf, size_t nbytes);

Arguments - fd : FD to opened file. (returned by open())
buf : Buffer to read content.
nbytes : # bytes to read.

Returns : { >0 : # bytes read
0 : EOF
-1 : error

Reading : Get file table entry from FD & check file accessibility (read mode for user available). Using file table entry, get inode entry & lock it.
Set parameters in u-area for byte count, I/O, & user address, followed by setting byte offset in u-area from file table offset.
Read file block by block till number of bytes required are read or EOF reached. In each read, convert file offset to disk block (bmap) & compute in-block offset & # bytes to read.
Read block (bread / breada) & copy buffer data to user buffer address & update u-area's fields.
Once read complete, unlock inode & update file table offset for next read.

× Read / Access Caveats :
1) Regular File : EOF reached reads less data than requested.
2) Terminal : Line by line buffering
3) Network : Buffering impacts performance (small size)

× File treated as byte stream, read/write is sequential.

× UNIX allows for random-access, via lseek.

4) lseek() <unistd.h> :

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

Arguments :

fd : FD entry for file

offset : Position to move pointer

whence : relative location (beginning, end, current)

+ SEEK_SET : from beginning (New pos : offset)

+/- SEEK_CUR : from current. (New pos : current pos + offset)

- SEEK_END : from end. (New pos : end pos. + offset)

Returns : New file position, -1 on error.

5) write() <unistd.h> :

ssize_t write (int fd, void const * buf, size_t nbytes);

6) close() <unistd.h>

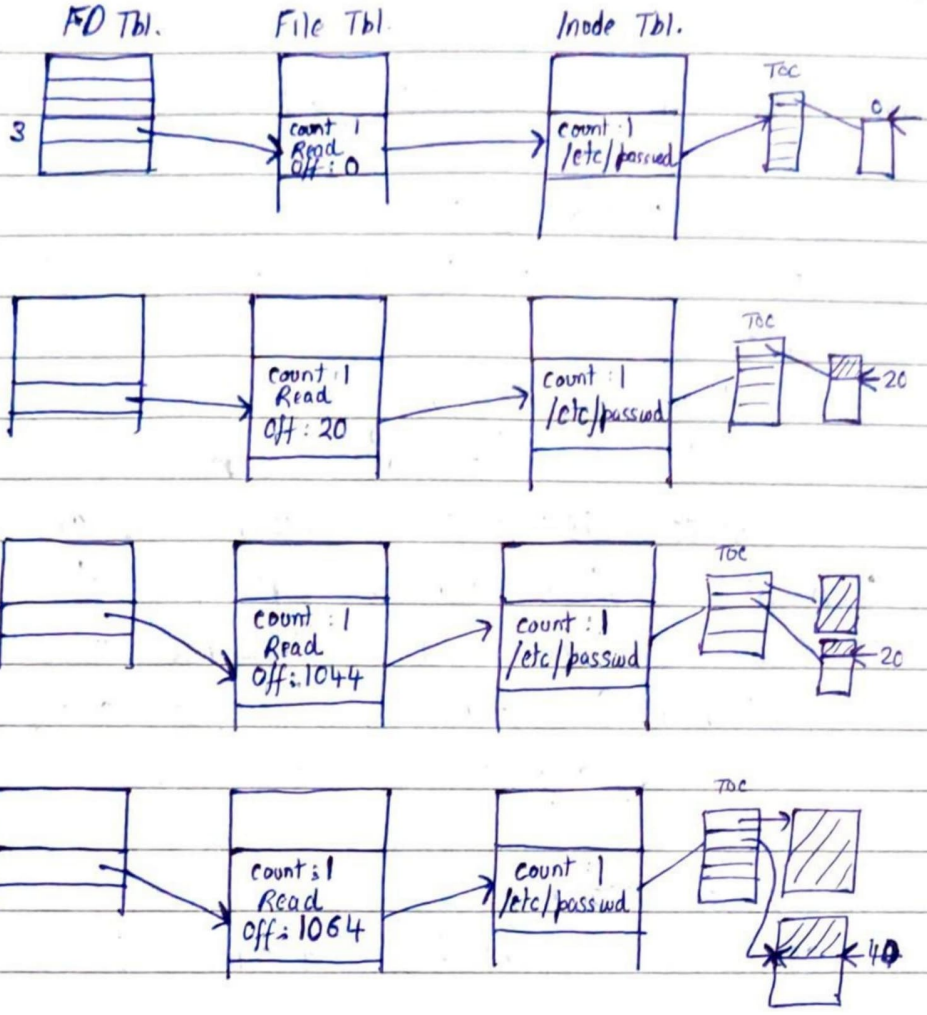
int close (int fd);

× Close performs decrement of inode ref count, followed by removal of file table & FD entry. (ref count of file table entry is also decremented).

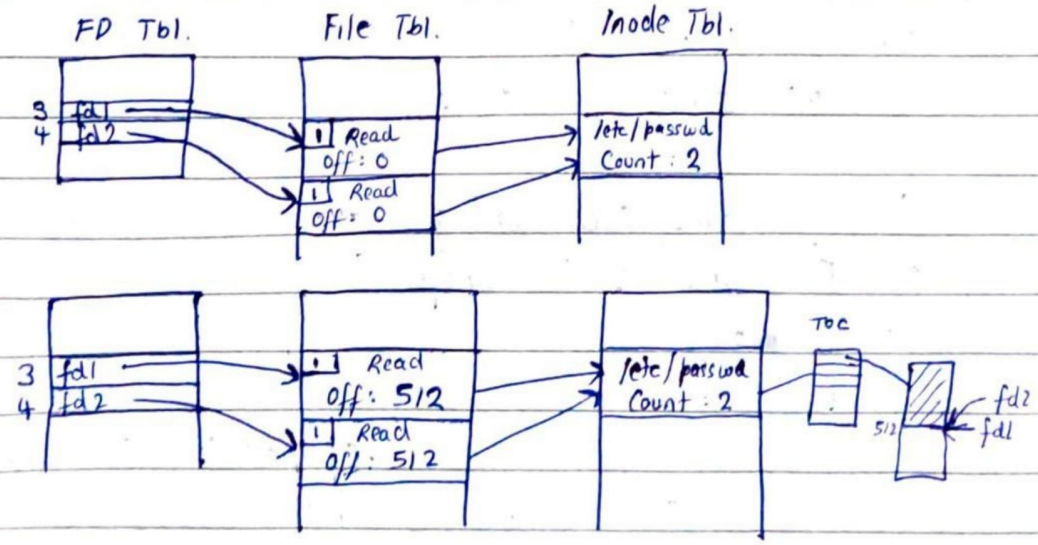
Eg:

```
int fd = open ("/etc/passwd", O_RDONLY);   // 1
char lilbuf[20], bigbuf[1024];
read (fd, lilbuf, 20);                     // 2
read (fd, bigbuf, 1024);                   // 3
read (fd, lilbuf, 20);                     // 4
```

Eg Q1




Eg Q3




```
fd1 = open("/etc/passwd", O_RDONLY);          // 1
fd2 = open("/etc/passwd", O_RDONLY);          // 2
read (fd1, buf1, sizeof(buf1) /* 512 */);     // 3
read (fd2, buf2, sizeof(buf2));               // 4
```

```
// Write data in file at position 10240 (byte offset)
char buf[512]; int fd;
fd = open("/etc/passwd", O_CREAT|O_WRONLY);
lseek(fd, 10240, 0);
write(fd, buf, sizeof(buf));
close(fd);
// TODO: Add error handling & signal traps.
// TODO: Add buffer population block
```

7) chdir() : <unistd.h>

```
int chdir(const char* path);
int fchdir(int fd);          // Lib fn, invokes other.
```

Returns : New directory's fd or void or status code.

cd : If path given, resolve to fd using namei.
Fail when inode type is non-directory or permissions not allowed for action. otherwise, unlock inode & place inode in process u-area.

8) chroot() :

```
chroot(const char* path);
```

× Change root of filesystem for single process.
× Changes root fs entry in u-area in a way like chdir.

9) chown() : <unistd.h>

```
int chown(const char* path, uid_t owner, gid_t group);
int fchown(int fd,          uid_t owner, gid_t group);
```

Returns : status code (0 = success, -1 = error).

× Changes file owner & group information.

10) chmod () : <unistd.h>

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

× Changes permissions (owner, group & other level) (r, w, x).

11) stat () : <unistd.h>

```
int stat (const char* path, struct stat* statbuf);
int fstat (int fd, struct stat* statbuf)
```

Arguments : path / fd : File reference
statbuf : Buffer to hold inode data.

Returns : (0 = success, -1 = error) status code.

× Returns file inode information in a given stat object. (Call by pointer). (includes device #, atime, ctime, mtime, nlink, etc)

12) mknod () :

```
int mknod (const char* path, <type & perms>, <dev>);
```

× Creates pipes, device files & directories.
× Similar to creat, but allows creation of special files.

mknod : First check to see if user is superuser (UID 0) or not, or if user is attempting to create a named pipe & fail if conditions unmet. Then retrieve parent's inode (namei) & if file exists raise error, then otherwise allocate new inode (ialloc) & release parent inode (iput). Then if file type is a block or character file, write device info (major & minor mode #) & return inode's fd. (map to fd).

× Pipes : Allow FIFO flow of data between processes.
: Aids in I/O synchronization.
: Types : named & unnamed.
named : accessible by all processes., persistent in filesystem.
unnamed : accessible to process & children & descendants.

13) pipe() : <unistd.h>
int pipe (int fd[2]);
Arguments : fd : fd's of reader & writer sources. (fd[0] = reader src.)
Returns : status code (0 = success, -1 = error).
× Creates a pipe for FIFO I/O.
× Pipe : Creates a new inode (ialloc) for pipe, followed by allocation of file table entries for reading & writing & adding references to the file inode. Then FD table entries are added & set in the given argument array, which is then returned (call by pointer) at callee site.
× Unnamed pipes only consume direct blocks from inode TOC (avoids redirection complexity).
× I/O performed in FIFO (queue) order. Writer waits until read is complete if writer has written all data to all direct blocks of TOC.
× Byte offset of read & write maintained in inode entry instead of global file table entry.
× Read / Write count maintained in inode entry.

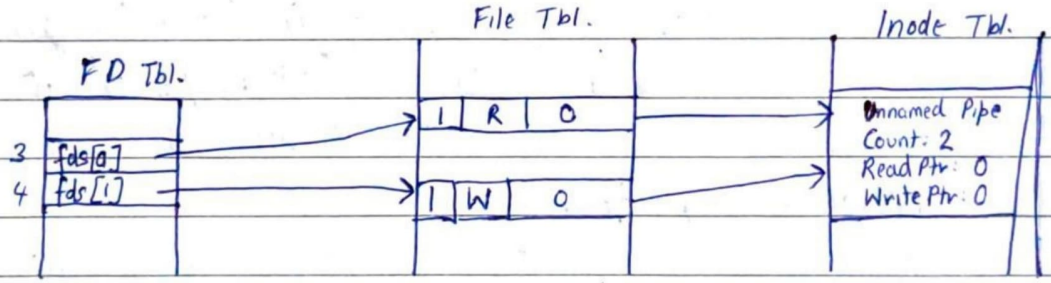
× Named Pipes :
- Initiated in a way similar to regular files.
- Every processes accessing pipe leads to increment in inode's read / write count.
- Accesses opening named pipes will sleep until another process has opened the pipe for the opposite operation (reader waits for writer & vice-versa).

× Pipe I/O Cases :

1) Write to pipe w/ avl. spc : Write data, update write ptr.
2) Redd from pipe w/ enough data for read : Read data, update read ptr.
3) Read from pipe w/ less data than req. : Read enough data as avl., sleep & wakeup when a writer finishes write.
4) Write to pipe w/o avl. spc. : Write as much spc. avl., sleep, wakeup when read completed & spc. avl.

E.g.

```
pipe (fds);
for (;;) {
    write (fds[1], buf, 6);
    read (fds[1], buf, 6);
}
```

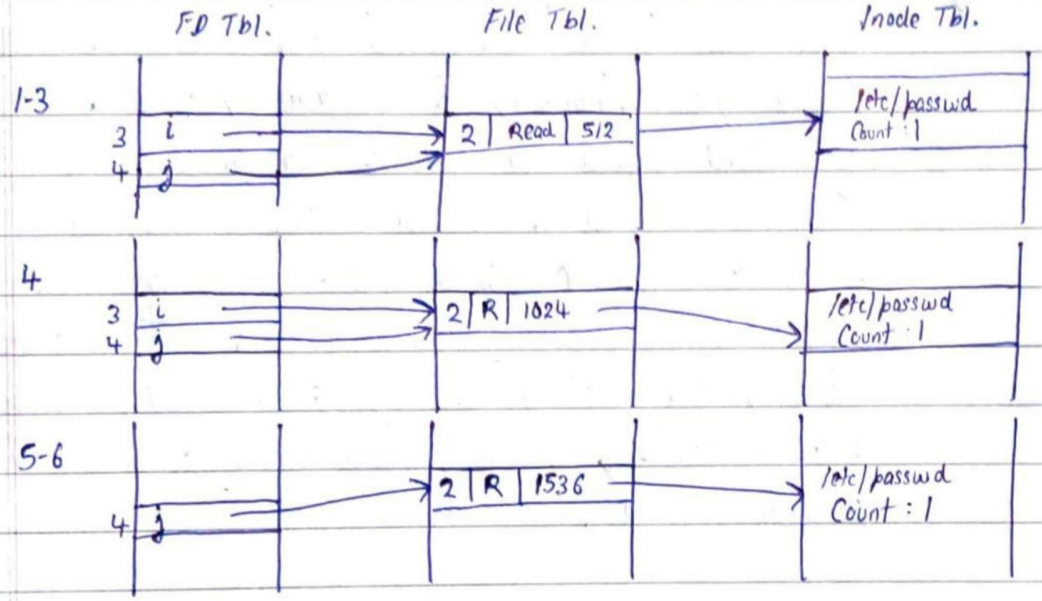



14) dup() : <unistd.h>

```
int dup (int fd);
int dup2 (int fd, int fd2);
```

× Duplicates file descriptor to a free slot in user file descriptor table. (Leads to increment of count in global file table, referred by multiple FD entries). (**No** updates to reference count in inode table).

Eg:
```
1   i = open ("/etc/passwd", O_RDONLY);
2   j = dup (i);
3   read (i, buf1, 512);    // offset 0 → 512
4   read (j, buf2, 512);    // offset 512 → 1024
5   close (i);
6   read (j, buf2, 512);    // offset 1024 → 1536
```




15) mount() : <unistd.h>

× Connects filesystem in a specified disk section to existing filesystem hierarchy. (Sections : Logical partitions in disk)

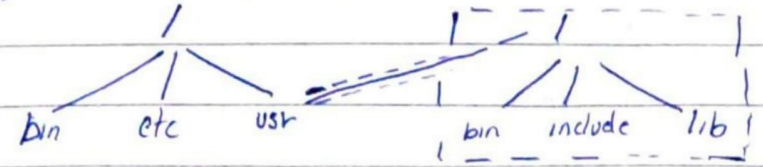
× mount (<spl path>, <directory path>, <opts>)

Arguments : <spl_path> : name of device spl. file of disk containing filesystem to be mounted.

<directory_path> : path to mount filesys. at.

<opts> : perms, access restrictions, etc.

Eg: mount ("/dev/dsk1", "/usr", 0).




× Non-persistent across reboots.

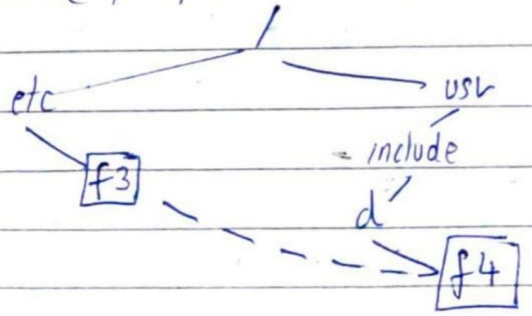
× Kernel maintains a mount table with entries for mounted filesystems
Fields : Device # to identify mounted fs
: Ptr. to buffer containing superblock
: Ptr to root inode of mounted fs.
: Ptr. to inode of dir that is the mount pt.

Mounting : First check for superuser & fail if not (only su can mount). Then get inode for block file (name) & inode for mount point (namei) & validate perms & access. If mount point is not of directory or ref. count > 1 (in use), fail & release inodes. Else, find a free slot in mount table, open block device (driver routine), read super block in free buffer, intt super block fields, get root of inode) of mounted fs & store in mount table, mark parent inode as mount point & unlock inodes.

Linking :

× Create multiple paths over the same file / directory in the filesystem.
× Changes required when creating links :
1) Update link count of source file.
2) Update inode of parent of symbolic link with name as name of link referring inode of source.

link ("/etc/f3", "/usr/include/d/f4");



Unlinking :

- Decrements link count by 1, & removes directory entry from parent.
  unlink (<source>)
- May deallocate disk blocks when link count hits 0, along w/ release of inode memory.

Unmount : umount ("<directory_path>)

- Removes mount point, after waiting & checking if inode for mounted point is in use.
- Race condition in multiple link calls : May exist when inodes retrieved for resolution during namei are not released, & can lead to deadlocks.
- Recovery from filesystem inconsistencies : Using fsck.