20/3/24

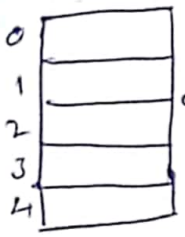
Comp → Proc → pross id

FILE MANAGEMENT SUBSYSTEMS

File I/O operations

File descriptar table

fd table → keeps the track of opened file information



default fd's opened in exectuable file

These are associated with console/ shell prompt

Each index in the fd table have file descriptor associated with opened file information.

Q. Who opens the default fd's?

Terminal opens the default fd's

* linux shell / terminal is the parent process of the a.out
* When a.out is created as child process then the contents of fd table are copied from terminal

File IO operations can be done by two different type of calls

Standard library	Basic IO calls
Standard IO calls	Universal IO calls
library calls	system calls
	↑
	Linux System

- fopen
- fclose
- formatted IO calls: fscanf, fprintf
- unformatted IO calls: fread, fwrite

} internally invokes system calls

- open
- close
- read
- write
- ioctl
- fcntl

* Standard IO calls, will aviable in almost all the plat forms like windows, Linux, Mac, Android etc
* Basic IO calls will specitic to Linux system
* standard IO calls are opplicable to only .

normal files

- text: .txt, .c, .java, .py, .doc
- binary: .exe, .jpg, .png, .mp3

* Basic IO calls are opplicable to normal files and special files

- Irc objects
- device files

Inter process communications

↓

For establishing comm b/w 2 process

* Disoidvantages is it is only applicable to Linux o

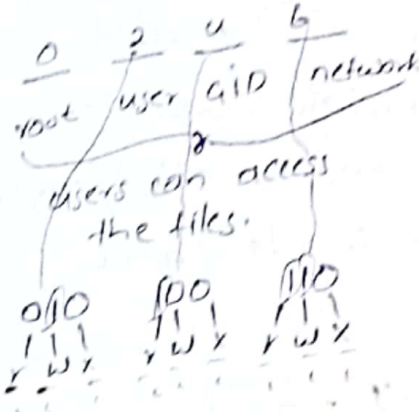
BASIC IO CALLS

open → open(filename, mode, permissions);
↓ It is applicable to file when we create a file.

Read - O_RDONLY
write - O_WRONLY
r/w - O_RDWR
append - O_APPEND
trunk - O_TRUNC

0-7

0 root | 2 user | 0 GID | 6 network

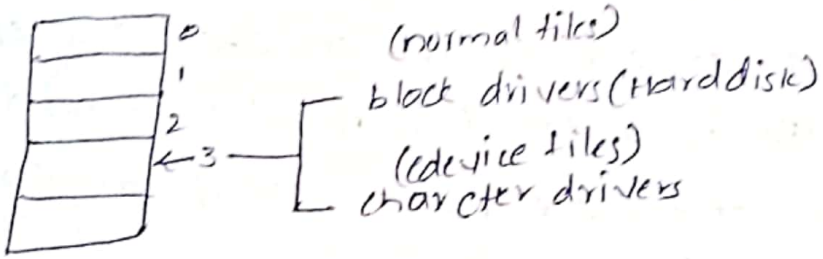


create - O_CREAT
of file
doesn't exis

open ("filename.txt", O_RDONLY);

VFS → sys_open

open creates an entry in fd table in the next freely available index



open returns index to the fd table entry on success
open returns '-1' on failure.

fd = open ("filename.txt", O_RDONLY/O_CREAT, 0700);

root, user, Group, network } rwx
100 - 4 → read
110 - 6 → read & write
111 - 7 → read, write, execute

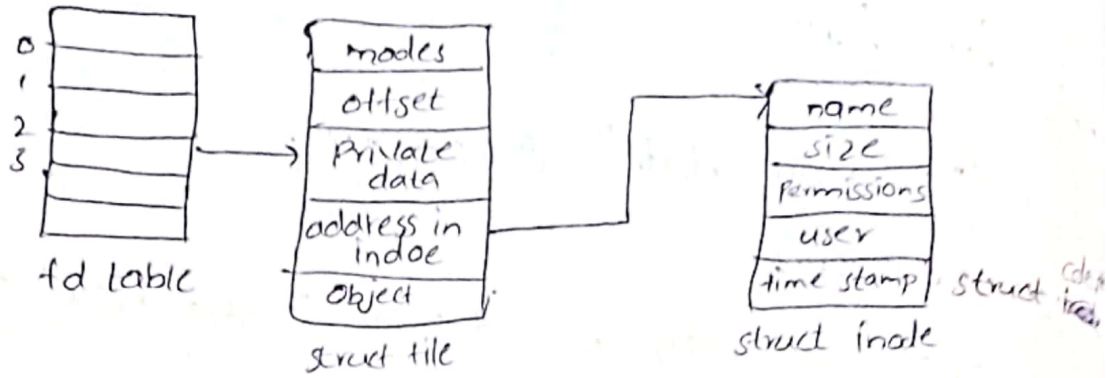
sudo is used for applying the root permissions.
root user have default permissions to access all the permissions.

User	group	network	
rwx	rwx	rwx	→
1 1 1	0 0 0	0 0 0	
7	0	0	

The permissions are ind... by octal
or
0700
user rw

The subsequent read& write system calls will be using fd to read & write

* Through this fd table only file object is created
* file object is also type of a structure struct file
* Members of the file objects are modes, offset/ position /private data, address of inode object



* Inode contains information about the file.
* File object contains information about the open file.
* When a file is created it is stored in Hard disk, The feature of file are name, size, permission, user, time stamps
* Inode object is created with the creation of file. It is the part of file stored in hard disk
* When the file is opened file object is created in kernal space of RAM. Inode object is copied from hard disk to kernal space of RAM.
* Fd table entry points of file object and it in points to inode object.

int write (int fd, void * buf, int len);

↑ index to fdtable entry (returned by open)

↑ Data which needs to be written to file

↑ number of bytes to be written

On success write returns number of bytes written to file.

Write is able to write only 50 bytes that's why it return no. of bytes written to file.

On failure it returns '-1'

int read(int fd, void * buf, int len)

↑ index to fd table entry

↑ buffer where read data from file shall be stored

↑ maxi no. of bytes to be read.

On success read returns no of bytes read.

On failure it returns '-1'

"unistd.h"

26/3/24

Reading the file

Reasons of failing

* file doesn't exists
* file doesnot have relavent opperational permissions.
* File doesn't have relavent group, user, network permissions.

```
#include<stdio.h>
#include<unistd.h>
Void main()
{
  int fd, int ret;
  char buf[64];
  fd = open("file.txt", O_RDONLY);
  if (fd< 0)
  {
    printf("Failed to open the file \n");
    return;
  }
  ret= read (fd, buf, 64);
  if (ret <0)
  {
    printf(" Failed to read the file \n");
    close(fd);
    return;
  }
  buf[ret]= NULL;
  Printf("%s", buf);
  close (fd);
}
```

→ if read returns 0 it means we have reached end of the file.

→ If we don't place this condition then buf returns all the remaining characters present there. If we place NULL then it prints the strings present in the file.

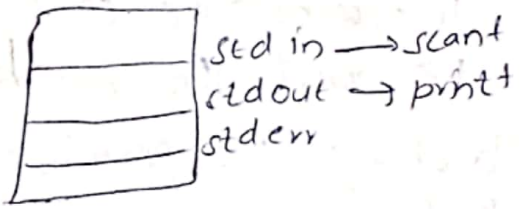
Writing to the file

```
#include <stdio.h>
#include <unistd.h>
void main()
{
  int fd, ret;
  char buf[] = "hello";
  fd = open("file.txt", O_WRONLY);
  if(fd<0)
  {
    printf("Failed to open the file \n");
    return;
  }
  ret = write(fd, buf, strlen(buf));
  if (ret<0)
  {
    printf("Fail to write in to the file \n");
    close(fd);
    return;
  }
  close(fd);
}
```

27/5/24

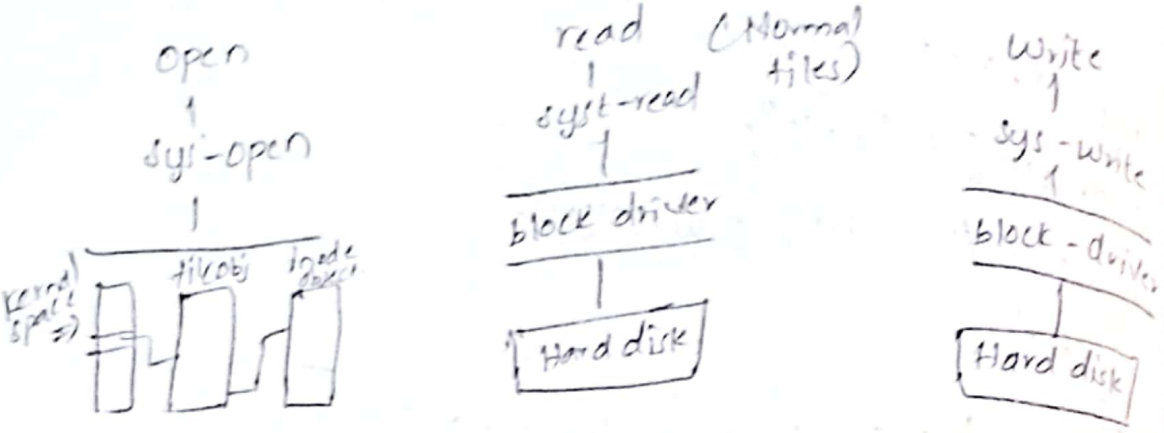
printf —invokes→ write

scanf —invokes→ read



* printf invokes write system call on fd1 similarly scanf invokes read system call on fd0

* Read system call is a blocking call, scant block because of read()
* Read blocks only when operating on (pipe obje, device, special file)
  it never blocks on normal files.



* In device files read goes to character driver.
* user space can fetch the contains of file object and inode object.
* To fetch inode object we use fstat() call.
* To fetch file object we use fcntl() call.

22/3/20

struct stat buf;

fstat (fd, struct stat *);
fstat (fd, &buf);

stat ("filename.txt", struct stat*);
stat ("filename.txt", &buf);

It fetaches the contents of inode object and fills the struct stat buf.

To change the permissions of file we use command line/system call.
command line interface is directly executed from kernal shell.

chmod() & fchmod() for system calls

chmod() is for command line interface

in command line interface
- chmod +0660 filename.txt → to change the permissions
- chmod +x filename → To apply the individual permissions

```
chmod("filename", 0660);
fchmod(fd, 0660);
```

Context Switching

* Switching of CPU from one process to another process is called as context switching

Q. What is reason of context switching?

1. In Round-Robin scheduling
   1. when the process is completed
   2. CPU time has expired.
2. In preemptive priority based scheduling
   1. When the process is completed then CPU is given to next high priority task execute
   2. Any Higher priority task come to execution.

30/3/21

ARM Processors:

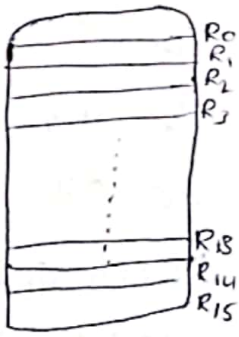
R_0-R_2 General Purpose registers

R_3-R_{15} special purpose registers

R_{13} - stack pointer (SP)

R_{14} - link registers (LR)

R_{15} - Program Counter (PC)



* Stack pointer points to the top of the stack.
* Link register points to the location of the next instruction that it keeps the address of the instruction which is to be executed by the function call.
* Program counter points to the address of next instruction
* CPU uses program counter to get the next instruction to be executed

PCB
context area { SPSR → saved program status register
CPSR → current program status register

If P_1 process is going to switch to P_2 Process then the P_1 related registers should be stored in SPSR and it is switched from P_1 to P_2 then the contents of spsr into CPSR and reload it in to the user space.

* Just before switching from one process to another process the current status register will be stored in saved program status register.
* Just before executing the next program spsr information will be fetched to cpsr to reload R_0-R_{15} related to next process.
* The information stored in registers R_0-R_{15} is called as 'context info'.

Push & POP are assembly level instruction to fetch & store the data form stack pointer.

branch & link assembly instruction to stored next instruction address before jump into functions address space in link register.