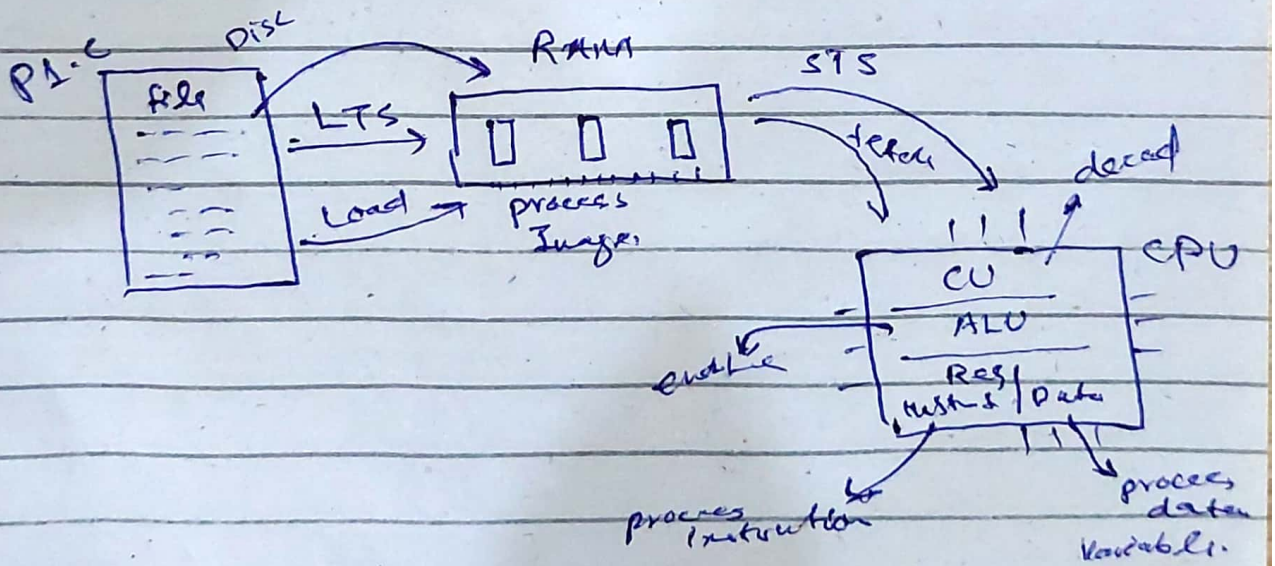
System Programming Lec 28/9/23

Book: Unix Systems Programming: Communications, Concurrency & Threads. (K.A Robins & Steven Robbins)

Chap :2 → Programs, Processes & Threads:

→ Program → Compile (.obj) → Link (.exe) → Load to memory → process.

* gcc f1.c -o f1.o



→ Library function calls.

* int close (int fd) → file discripter.

success = 0
error = -1.

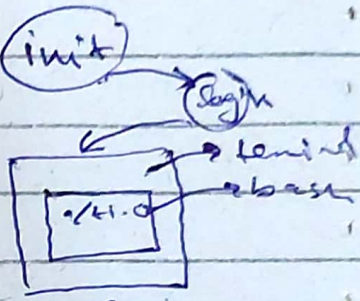
These files are open when a process is being executed.

STDIN_FILENO → Keyboard
STDOUT_FILENO → Monitor
STDERR_FILENO → Monitor

split, splice, slice.

→ Shell take command and execute it.

→ This above function is used to close a file.



→ when executing sys-call. (return value) we check for errors. if error occur we then check for its reason.

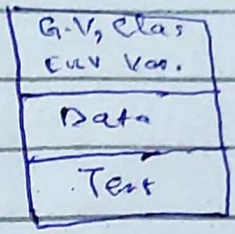
using ① perror(); ② strerror();

(3)

•env
•printenv

System Program Lec. 9/10/23

Environment Variables.



→ $ env.
- display all env variable..

→ $ echo $HOME
- print a variabl in env.

→ $HOME = "value"
- Change value of a var.

→ #include <stdio.h>
char** environ;
- an array of character to environt var.

→ >which ls
- shows/print path of a command. It only seerch for this in path var.

process termination:
→ process termination are of two types:
1. Normal termination
→ exit().
→ _exit() / _Exit().
→ return status of main();

(6)

~~tokenization~~

→ with normal termination they return a status too.

calls atexits.

→ Their prototypes are

int exit(int status);

They are same

int _exit(int status);

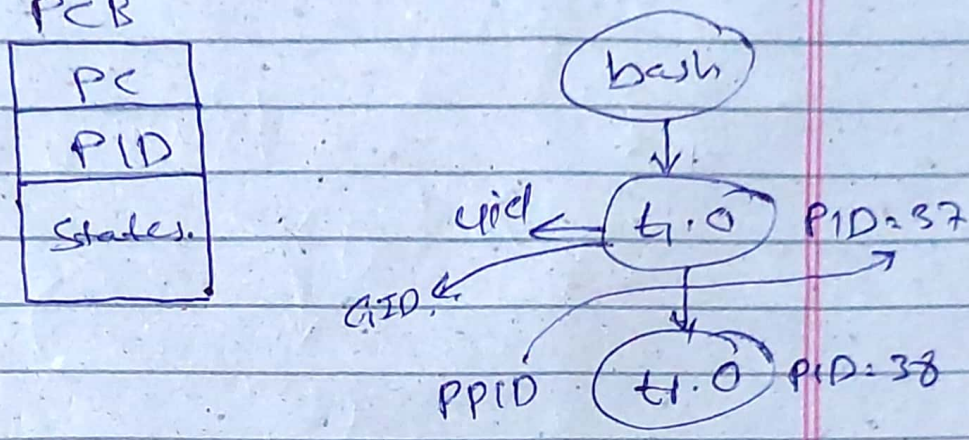
int _Exit(int status);

int atexit(void *(func));

- This is a function which take function name as a parameter and returns.

Sp - Lecture 16-10-23/MO

Ch # 3: Processes.



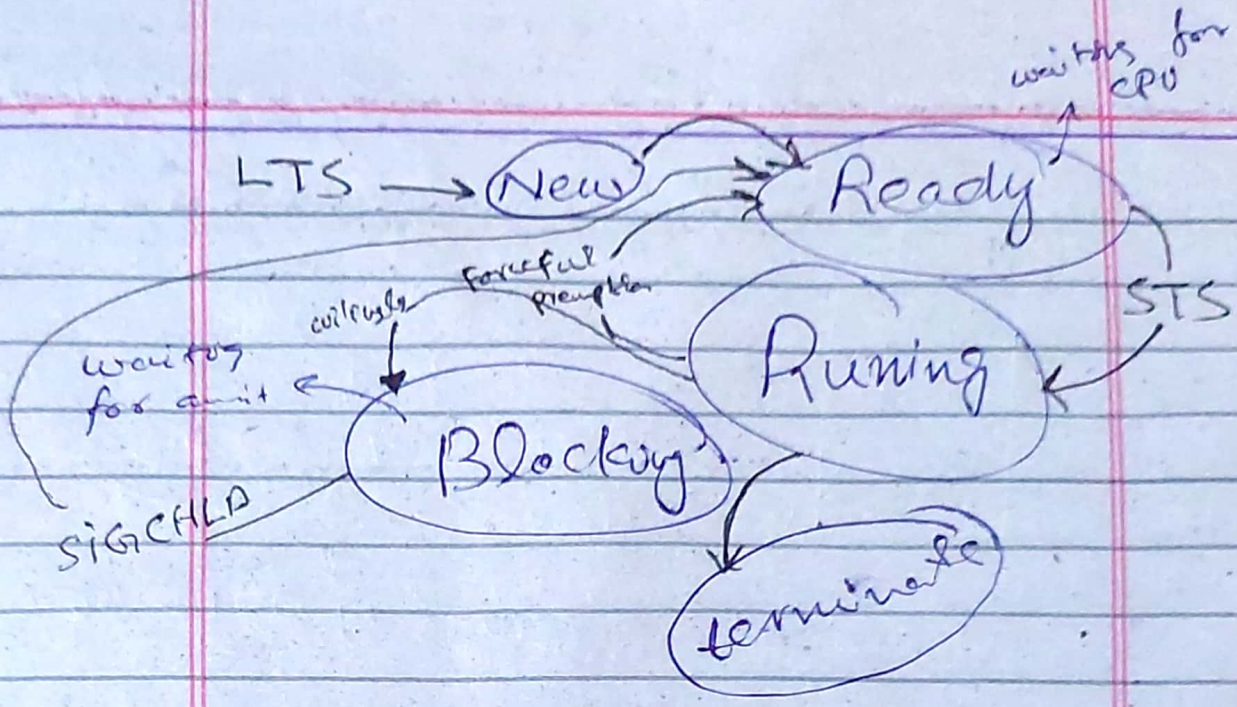
→ $ sudo addgroup SECTION-A
→ $ sudo usermod -aG SECTION-A user*

→ int getpid(void); get process id
→ int getppid(void); get parent pid
→ int getuid(void); get user id
→ int getgid(void); get user group id.

→ Effective user is a user which temperorily switch from one user to another.

→ PS -aux → long
→ all
→ show list of all process in bash.

PID Stat
D → uninterruptible step
S → interrupted step
R → running
Z → Zombie
T → Suspended.



Multiprocessing:

int fork(");

Resource exist
limit exceed.

-1 : error → set errno

0 :

>0 :

SP Lec 19/10/23 – Thu

Store reason of child terminate process.

```
int wait(int *status);
```

- failure: -1.
- success: >0 → Child id.

ECHILD — no child exists

EINTR → fail due to interrupt.

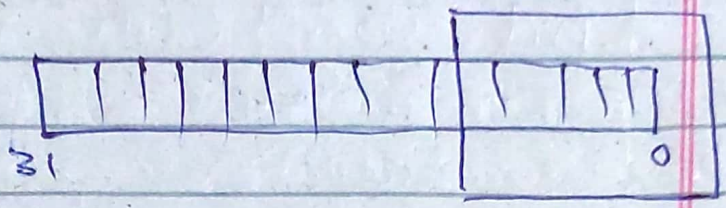
EINVALID → pass an invalid option in option

```
int waitpid(int cid, int *status,
            int option);
```

process termination

- Normal → exit status of child process.
- Abnormal (signal) → Completely terminated → terminating signal?
- Suspended/Stopped → stopping signal?

Status



Stat is exit status of child process.

- WIFEXITED
  - return true if a process is terminated normally.

- WEXITSTATUS:
  - exit status of child process.

- WIFSIGNALED:
  - process is terminated abnormaley completely.

- WTERMSIG:
  - complete abnormal termination signal/reason.

- WIFSTOPED
  - show if process is suspended or stoped

- WSTOPSIG
  - Reason of suspension/ stopping a process.

SP - Lec 26-Oct-23/ Th.

Execute System Call:

→ These system call replace the current process image with another process image.

→ They donot return any int in success case so will not execute the below programe once it is called.

→ In failure they return -1.
- Convention is to use fork and then exec so that it doesnt replace the image.

→ In failure case the set errno also: 'E2BIG' (have command line errors too many), 'EINVAL', 'EACCESS'

MIBSD

Prototypes:

i) int execl(const char* path, const char* filename, const char* arg[0], const char* arg[1], ..., NULL);

ii) int execlp(const char* filename, const char* arg[0], ..., NULL);

iii) int execv(const char* path, const char* argv[]);

iv) int execvp(const char* filename, const char* argv[]);

v) int execle

vi) int execve

lec # 7

DATE: Oct 30. DAY:

Unix I/O

- File Handling:- Arguments.

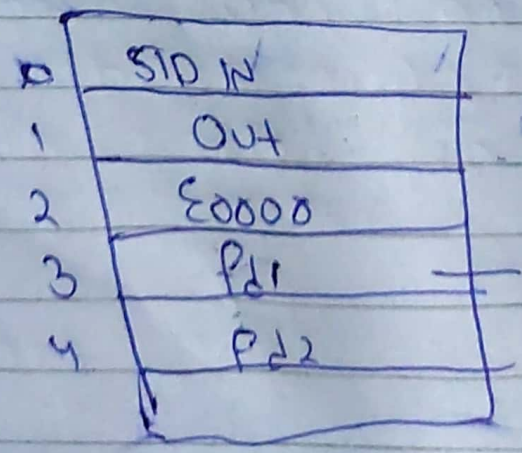
int open (const close *path, int flag)
int read(int fd , char *Buff , int Size of buffer)

int write (int fd, const char *buffer, int length of buff)

Copy src to dest.
cp src dest (STEPS)

1) OPEN Src file (for Reading)
2) open destination file (if don't create it. (for writing)
3) Read Src file
4) write dest file
5) Close Both file.

- Open System call, add otha file in file Descriptor table.



• in Reading of file we dont use the third Argument. (ORD-ONLY)

Data will come in Buffer.

DATE: DAY:

| = or gate

```
int main ()
{
    int fd1 = open ("f1.text", ORD-ONLY);
    if (fd1 == -1) {
    Perror ("failed to open f1 for reading \n");
    return -1; } # Error handling
    int fd2 = open ("f2.text", O-WRONLY | O-CREAT | O-TRUNC, S-IRWXU | S-IRGRP | S-IROTH);
) int fd2 = open ("f2.text", O-WRONLY | O-CREAT | O-TRUNC, S-IRWXU );
    // for Step 3 make Buffer
    char Buff[100];
3)  int br = Read (fd1, buff, 100);
4)  int bw = write (fd2, buff, br or 18);
                    strlen (buff)+1;
// 18 because if there is 18 bit data we read from file.

int close (fd1);
    close (fd2);
}
```

when you create a file then you have to give/tell the Permission in the last argument of it.

DATE: DAY:

O-Flags	O-Permissions.
	user, Read, write, execute.
O-RDONLY } m-exclusive	S-IRUSR
O-WRONLY } m-exclusive	S-IWUSR
O-RDWR } m-exclusive	S-IXUSR
	S-IRWXU
O-CREAT.	GROUP.
	S-IRGRP
O-EXCL } m-exclusive	S-IWGRP
O-TRUNC } m-exclusive	S-IXGRP
O-APPEND } m-exclusive	S-IRWXG
	S-IROTH
	S-IWOTH
	S-IXOTH
	S-IRWXO

Permission of Read, write, exec for others.

. Those who ar not part of the group associated with file can access it.

• flags are used to tell what we want to Do with file • Read, write, Read write.

DATE: DAY:

• if file already exist and we want to open same file. So instead of Replacing it give us the error. this is done by excl.

• if file does not exist we create the file by O-CREAT.

• if we want that error should not come, just make another file available for ove write. we wont use excl.

• O-TRUNC finish the existing content & ove write it.

O-APPEND Dont finish the existing content, But will start working ~~of~~ from below of old content. Instead of Removing OLD,

• for cp command we will use O-TRANS

DATE: DAY:

- Buffa will Return 0, when it has Success fully Read all from buffa.

- if one Buffa is not enough So the Remaining will go to next Buffa, but the Return will be the no. which are not Succes fully left.
  bl2= read (fd1, buffa, 100);
- or dont make new buffa but write the buffa & make it empty and come back, Read the Remaining.

if more buffa

while(1) ← char buff[10];
  int br= read (fd, buff, buffa);
  =10);

  //error handling
  if (br==0)
    break;

int unlink (const char* Path)
  for move command.

SP-Lec #9

DATE: DAY:

// when you make buffer so buffer(stack, heap) will overflow.

- file copy-

```
int main()
{
 int fd1 = open("f1.text", O_RDONLY);
        // Error checking.
int fd2 = open("f2.text", O_WRONLY
| O_CREAT | O_TRUNC, S_IRWXU | S_IRGRP
| S_IWGRP | S_IROTH);
        // Error checking.

  int br = Read (fd1, buff, 10);
   if (br == 0) break;
  int bw = write (fd2, buff, br);
 }
```

- Read first line of src file.

```
  int main()
{       char buff[100];
  int fd1 = open("f1.text", O_RDONLY);
 // while (1)
{//
```

5 Bytes

10	Hello
12	World
13	123

```
int br = myreadline (fd1, buff);
  // if (br == 0) //break ; (100)
Printf ("First line = %.5s\n", buff);
  }     // line#
// Shows code for line.
   whole line.
```

. we will apply (\n) on the last of hello. So we will read every untill we reach \n.

DATE: DAY:

```
int myreadline(int fd, char* buff,
                        int size)
{  int i
   int numbytes = 0

   while (1) // until we Reach
   {
      br = read(fd, &buff[0], 1)
      if (buff[i] == '\n' || br == 0)
         break;
      i++;
   }
   buff[i] = '\0';
   numbytes = i;
   Return numbytes;
}
```

* for Sentence Reading the Condition will be (-) instead of In.

DATE: DAY:

if to check Hello is Repeated How many times.

```
if(!strcmp(buff,"Hello"))
        itt;
}
Printf("Freq of Hello in
f1 = %d\n",i);
```

Freq of Hello world.

Assigment 02.

DATE: DAY:

- File monitoring:

Target:
- read from STDIN & WRITE to f1.txt
- Read from f2.txt & write to F3.txt

```
main()
{
open f1->w, f2-R, f3->w
// we know the code

br = read (STDIN-FILENO, buff, 100);

bw = write (fd1, buff, br);

br = read (fd2, buff, 100);

bw = write (fd3, buff, br);
```

- Lets say do not enter any data So program will block.

DATE: DAY:

- but this Rd2 we can also be block
  . we dont know which will be blocked.

- So Solution run them on diffent Process.
- make an child Process.
- but thae will be many problem (another Process image) another memory

- So best Solution is file monitioring.

Syst) Function call :- Select.

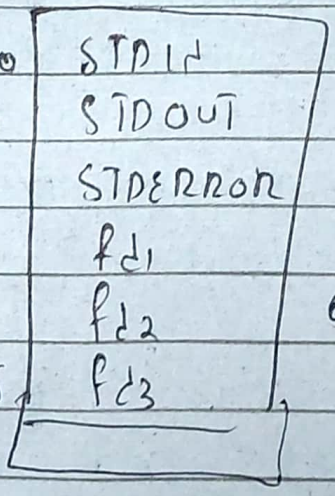
Select = Success 0
>0
-1 error.

In case of Select System call check no of Ready files.

>0 Represent how may files are Ready.
. if no file is ready it is blocked by default.

DATE: DAY:

- if we restrict Select Sys call ~~with~~ time ~~Restrict~~ So after the time is up.
- It will come out from blocking state and will Return 0 (if no file ready) (only return 0 in timing case)
- So if its not time Restricted so it will never Return 0.



max = 4
5 no.
max+1 = 5

- first argument show) the no of max file.

DATE:_____ DAY:_____

So we can find it by max + 1.
if fd2.

- So 2nd argument we will tell Select func which file to monitor.
- So we will make (1) the index which we want file to monitor
- we will put them in read Set.
- Same for write, & execute Set.

- So the file which is Ready will be (1) otha zero in Rset.

int Select(int nfd, fd-set * ~~Rset~~ readset
, fd-set * writeset
maxfd+1 , fd-set * exset
,

DATE:_____ DAY:_____

Program:

~~[illegible]~~

~~[illegible]~~

1) Declare Read Set
2) Initialize Read Set with Zero.
3) Set indexes with (1) used to be monitor.
4) max fd.
5) Select
6) Condition to find which file is Ready.

main

{ open $f_1 \rightarrow w$, $f_2 - R$, $f_3 \rightarrow w$

1. fd-set my read set;
2. FD-zero (&my Read set);
3. FD-SET (STDIN-FILE NO, &my Read set);
   FD-SET (FD2, &my Read set);

// STDIN & FD2 to be monitored

DATE: DAY:

```
int max fd = (fd2 > STDIN_FILENO)
              ? FD2 : STDIN_FILENO;

int Select ( max fd+1, &myreadSet
nrf=         , NULL, NULL, NULL);

if ( FD_ISSET ( STDIN_FILENO, &myRead
     Set ))
   // Read from Stdin & write to f1)

if (FD_ISSET(fd2, &myreadSet))
  { Read f2
    write f3
  }
}
        Solution.
    Loop if more then 2 files.
```

DATE: DAY:

- file Representation:-

```
int main()
{
 int fd1= open("f1.text", O_RDONLY
  int  x = fork();
int fd2= open("f2.text", O_RDONLY);
  int bo = read(fd, buff, 6);
 Printf("%s",buff);
 }
```

- if we read from fd:1 So Hello, world will be read. whole because the offset is shared.
- when offset is changed So other will Str Read from updated offset.

- if int bo= read(fd2, buff, 6); So This is will be Read because offset is not Share.

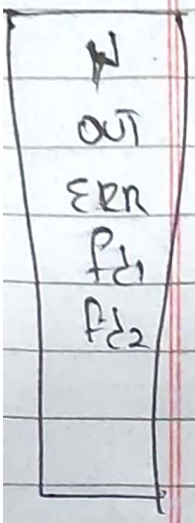
DATE: _____ DAY: _____

when close (fd1)
So no of links in SFT is 1, & Ps will be 1 too
when close (fd2) So no of links in SFT will be zero & Ps will be zero So
For write.

```
int main()
{
   int fd1 = open("f1.text", O_WRONLY);
   int x = fork();
int fd2 = open("f2.text", O_WRONLY);
  if (x==0) //child
  {
      char buff[] = "file0";
write (fd1, buff, strlen(buff)+1);
  }
else
{   char buff[] = "world";

   write(fd2, buff, strlen(buff)+1);
}
}
```

DATE:_____ DAY:_____



• offset points at start for Reading.
• offset update after every Read.

DATE: DAY:

User Space.

- File descriptor table is in user space.
- Each process has its own file discriptor.

Kernl Space.

- The System file table is in Kernl.
- when open System call execute the entry is added in system file table.
- info in system call table.
- e.g:- offset (location about Reading and writing).

- At the end the offset Return NULL. (when every file is Read).

- System file table tell us about the mode (Read, write, Exec) which we want to execute.

- no of time open func called. No of entry are made at SFT:

DATE: DAY:

- . tell us about the no of links pointing toward SFT.
- if O-RD ONLY So 1 Link.
- if O-RDONLY, O-WRONLY 2 links.

- System file table points links to other table known as ino table.

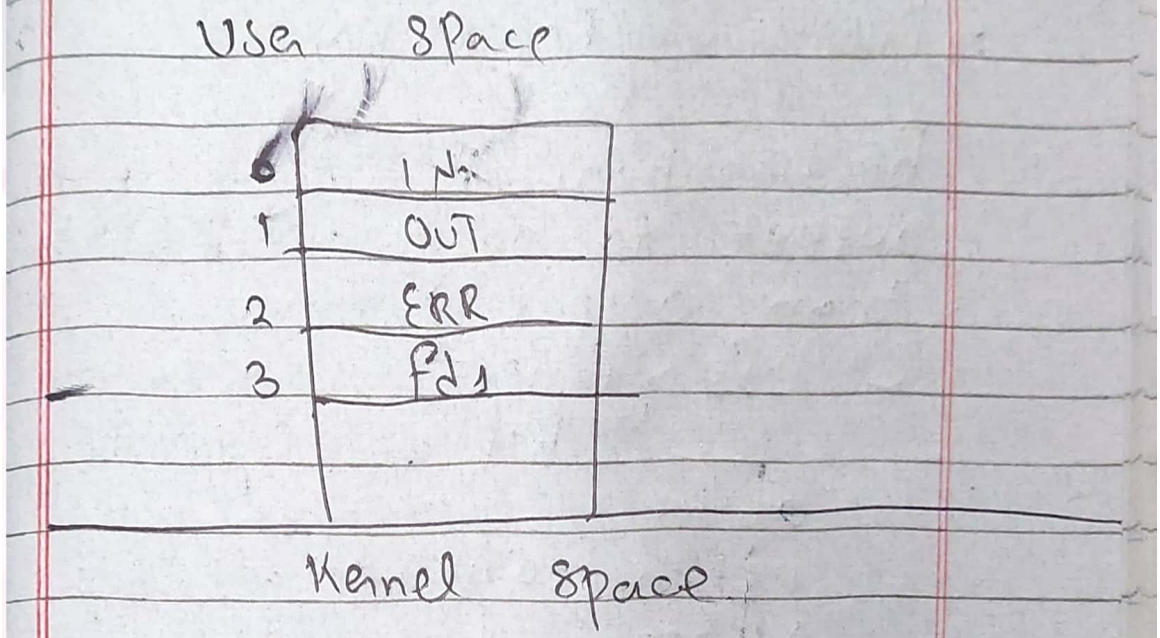
- iono table is also in Kernel process.

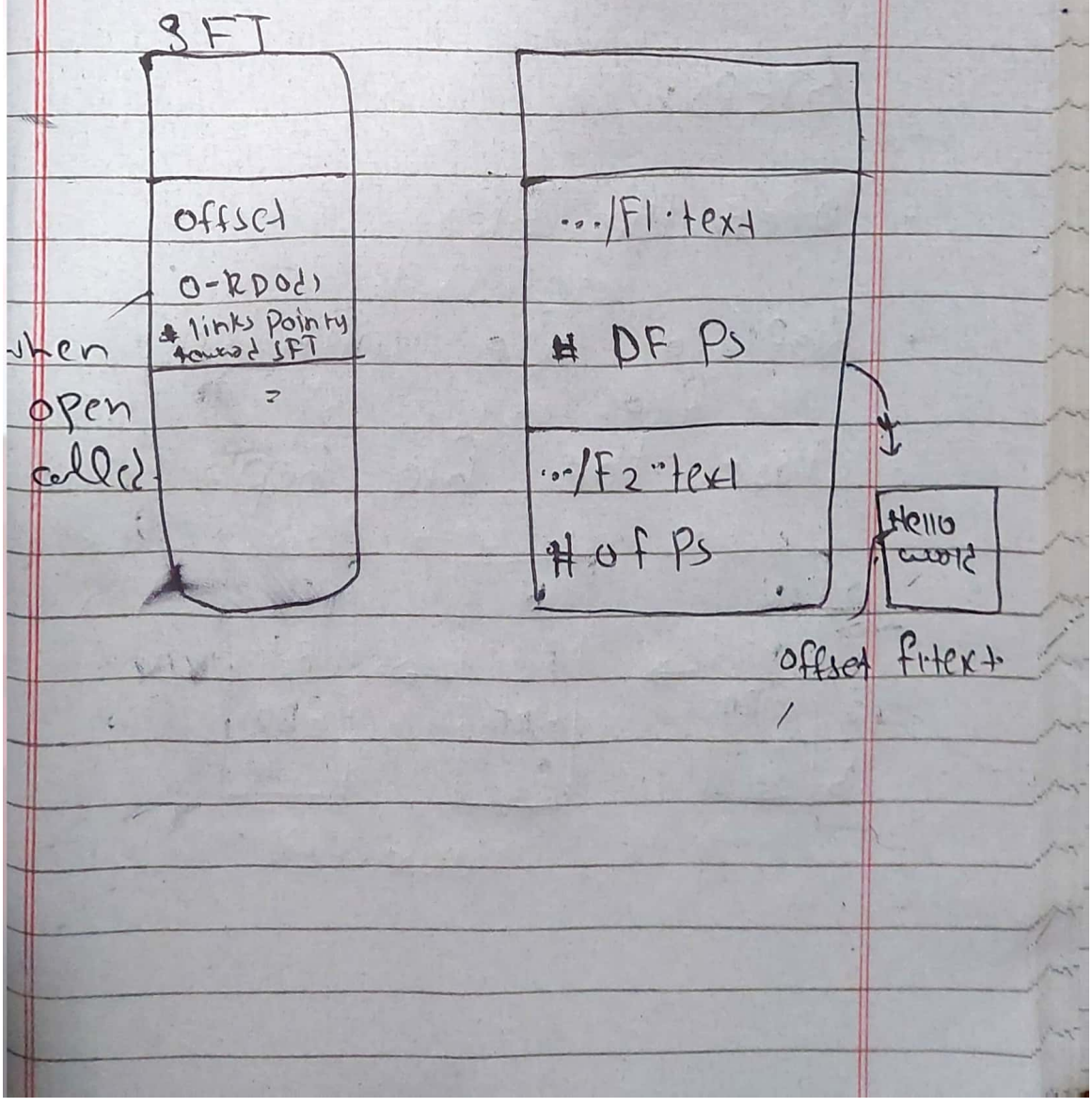
- iono table tell us about the no of opened files.
- if 10 files are opened in /fi-text So entry is 1. but the no of process opened file is 10. (PS)

- ino table display on hard isk.

DATE: DAY:

· DFT table is shared only in case of inheritance. entry





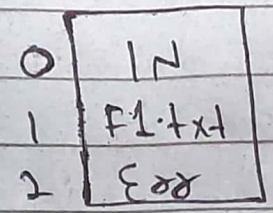
DATE:_____ DAY:_____

Filters & Redirection:-

Redirection:- Redirect our output to our file Rather than output Screen.

ls > fi.text

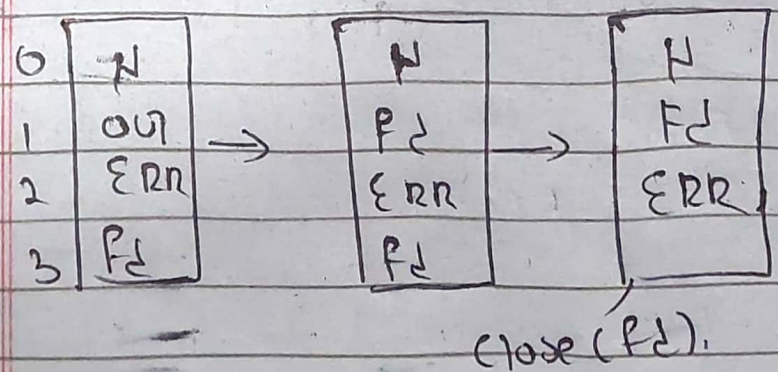
↳ Replace Symbol.



- output / changed with file.

STEPS:-

- Open the file.
- Replace file with other file.
- Close the file Replaced.



- Cat command is used to check every Redirection

DATE: DAY:

- The System call we use is dup2.

- int dup2(int fd1, int fd2); (fd1: want to Replace, fd2: Destination on which we want to)

- int dup2(fd1, STDOUT);

- Cat command is used to Redirect

- if cat we write with no file So no Redirection will occur.

- Read from STDIN, write to STDOUT.

(2) Cat h.text

Read from h.text, write to STD OUT.

(3) Cat > h.text

Read from STD IN, write to h.text

Cat > f1.txt

DATE: (1) (2) (3) DAY: anyone.

(4) Cat f1.text > f2.text

Read from fd1 write to fd2.

CODE:-

```
int main (int argc, char *argv[])
{
  if (argc == 2)  // input Redirection
  {
    int src = open (argv[1], O_RDO
    int = dup2 (src, STD_IN_fileNo);
        close (src);
  }
  else if (argc == 3) // output
                      redirected.
  {
    int dst = open (argv[2], O_WRO
    int x = dupt (dst, STDOUT_file);
        close (dst);
  }
  Else if ( argc == 4) // I/P/OUT red
  {
```

DATE: DAY:

```
int src = open(argv[1], O_RDONLY);
int = dup2(src, STDIN_FILENO);
    close(src);
}
int dest = open(argv[2], O_RWRONLY);
int Dupt(dst, STDOUT_FILENO);
    close(SRC), close(DST)
}

int br = read(STDIN_FILENO, buff, 100);
int bw = write(STDOUT_FILENO, buff, b);
```