"System Programming"

Book : Unix system Programming
Steve Robbins
And Key Robbins. ①

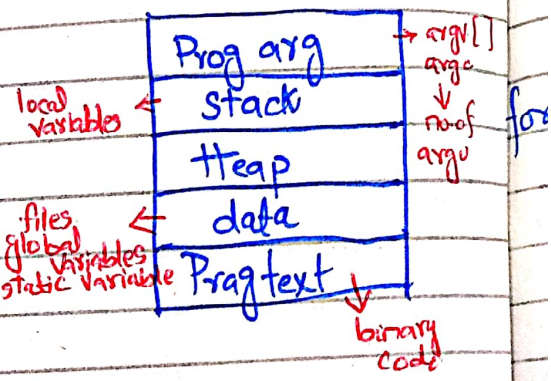
Program – single / multiple file
↳ obj file (code) and exe file. (executable)

Program → instructions → compiler → executive program

launch prog → instance process ← Main thread ← PCB / TCB

↳ Memory allocation.

The text section comprises of the compiled program.



Data stores global & variables that are not declared inside function. they can be accessed throughout the whole life of application.

Stack contain temporary data & is used for local variables & return addresses & Func parameters.

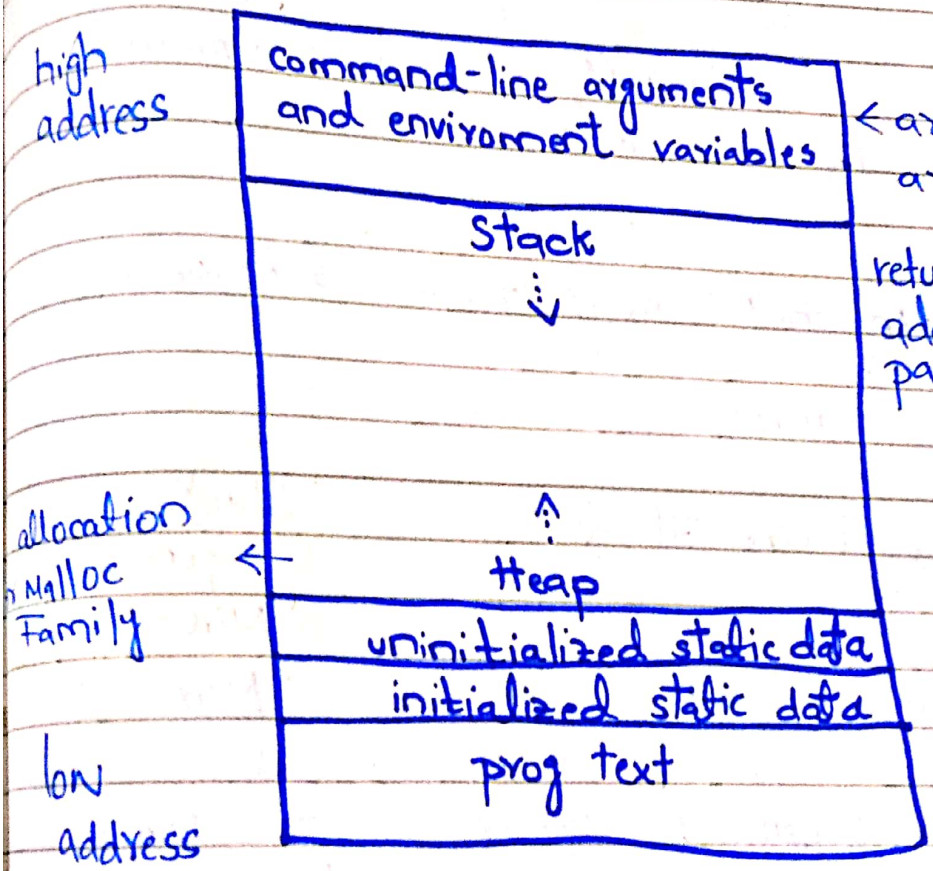
Heap is used for dynamic Memory allocation & is managed through calls malloc, new, free etc

⇒ PCB is data structure used for managing scheduling diff processes.

process has some address space and at least one thread.

TCB has its own
- Stack
- Program counter
- register
- State

(2)



obj file → binary code
exe file → giving starting & ending point connect b/w diff files.
compiler → executable program

argc → no of arguments
argv → lists of argument strings.

envp [] → environment string.

Command line Arguments.

① Main ( )

int argc → argument counter. (3)

char * argv [ ]

↳ argument vector ↳ to store arguments.

argc is always greater or equal to one.

argv[0] → Program is invoked.

argv[1] first command line argument.

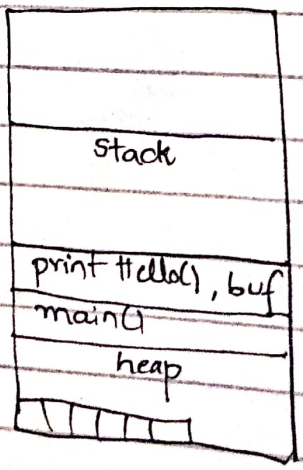
Any thing starting with # is known as Macros.

Lecture # 2

Stack function
Min = 1 frame
Max = 1 frame.

Memory Leak:

When the pointer of some dynamically allocated variable is displaced.



Memory Leaks.

```
printHello() {
    char *buff = (char*) malloc(20);
    strcpy(buf, "Hello World");
    printf("%s", buf); }

int main()
{
    printHello();   1st time
    printHello();
}
```

(4)

Disadvantage

- Memory is wasted.

if variable is allocated at end

*(x+10), x

if memory is not allocalled then the error we have is Segmentation

Memory allocation needs

Environment

```
① #include <unistd.h>
② # include <stdio.h>
③ extern char** environ;  (pointer towards string arr)
④ int main() {
⑤ for(int i=0; environ[i] != NULL; i++)
⑥ printf("%s", environ[i]);
⑦ return 0;
}
```

```
char a;          character v
char *b;         pointer to char v
char arr[];      char array
char *arr[];     pointer to char array
char **c;        pointer to string array.
```

Array k nam NO khud char pointer hota ha

```
b = arr;  -> str
```

array name = string.

```
str*  = string array
```

③ Extern

↓

tells complier this this variable is already defined.

↳ This is declared in unistd.

points towards environment variables.

↳ data : P.A

& pointer : Data section

System Call

```
char name[] = "PWD";
char output[20];
char * getenv (char* name);
```

↳ Request OS to check the name if this env variable is present or not.

↳ if this is present returns "PWD" otherwise "NULL"

```
output=
if (getenv (name != NULL) {
    printf ("%s", output);
}
else
    printf ("Environment Variable is not present");
return 0;
}
```

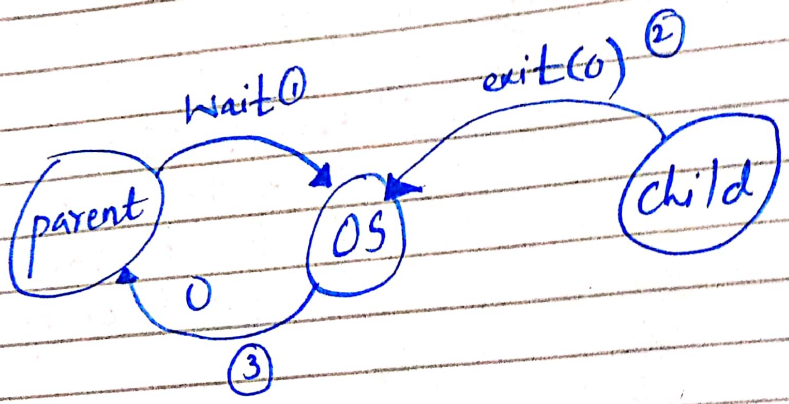
Process Termination:

return 0
↳ successfully ended
↓
O.S

Void exit (int status); (7)

status = number
0 → successful
1 → some other state.

void - Exit (int status);
void - exit (int status);



possibility :
=
parent child se pehle mar jaye.
tho child "init" ka child process
ban jata hai.
Kisi process ko ye nahi pata hoga ki uska child kon ha.
After every few secs, wait call issue karta hai

Abolish PCB
Reclaim Memory (garbage Collector)
delete signal.

⇒ K if koi wait par hai I am available.

➡ That phase in which no parent process is waiting for child process that process is known as zombie process.

(8)

⇒ Ultimately init calls 4 wo process ka wait khatam hojata hai.

➡ Orphan process jiska original parent process mar jaye and later on zombie process bann jataa hai.

↳ we don't do these 3 thing.

=

=

=

PCB - Status - terminated.

Function Call:

void atexit ((void)(*exithandler)(void))

when prg exit, wo is func ko call karegga.

```
exit handler ( ) {
printf ("It was nice ... ")
                    }
```

and message will be generated.

↗ It occurs because of the bytes that are used by not released

(9)

End of Chap2

Memory leaks occurs dynamically when you allocate memory and never give it back.

```
int *node;
node = (int *) malloc (8);
char * login = new char (50);
```

↳ to avoid Memory leak.

do this

```
free (node);
delete login.
```

A memory leak is caused when you allocated memory, haven't yet deallocated it and you will never be able to deallocate it bcz you can't access it anymore This creates memory leak.

Function without Memory Leak.

```
#include <stdlib.h>;
void f()
{
    int *ptr = (int *) malloc sizeof(int);
    free (ptr);
    return ;
}
```

10

-> void Exit (int status);
void exit (int status);

void at exit ((void) (*exit handler)(void));
Function call.

Lecture # 3

Unix processes

getpid (); No argument.

return : parent id pid_t.

↳ PCB
↓
OS.

↳ system call that returns the process id of calling processing.

pid-t get ppid (); - return parent process id.

System Calls

uid_t getuid(); returns user id
↳ unsigned integer.

gid_t getgid(); returns group id

uid_t geteuid(); returns effective user id

gid_t getegid(); returns effective grp id.

pid_t fork(); system call for creating child process.
↳ unistd.

```
pid_t pid;
    if (pid = fork() == -1)
        perror ("Error");
else
    if (pid == 0)
        printf ("child process");
else
    printf ("parent process");
```

if interrupt is failed the process will again start from next instruction but if SC fails then sara prg hi fail hojata hai.

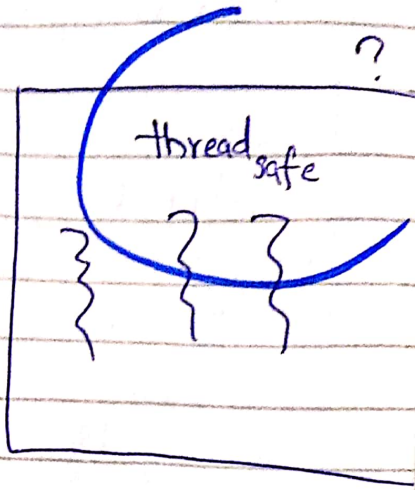
perror → C library call

↳ read error no

↳ Print message that corresponds to that number.

Error: Got interupted.

perror is not thread safe.



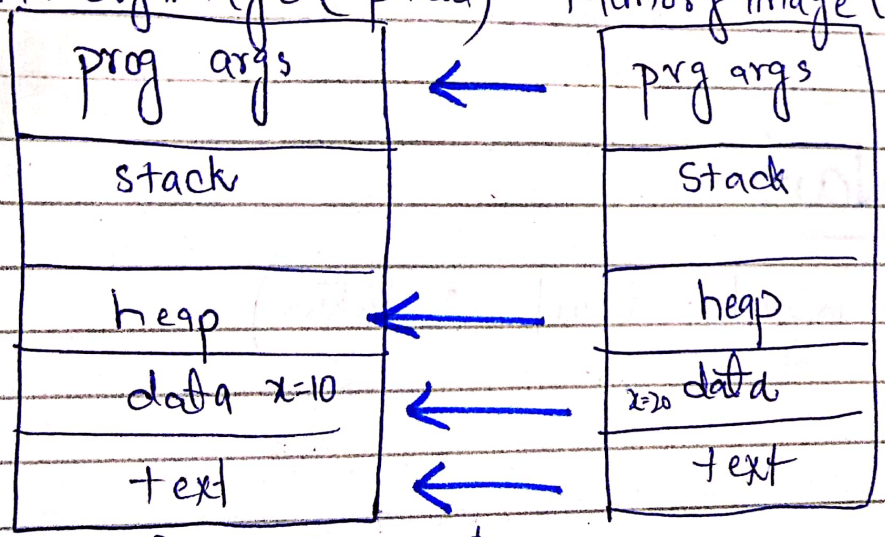
12

Call by value:

Strerror (errno);

↳ original copy & one other.

⇒ When child process is created parent inheret some features to child.

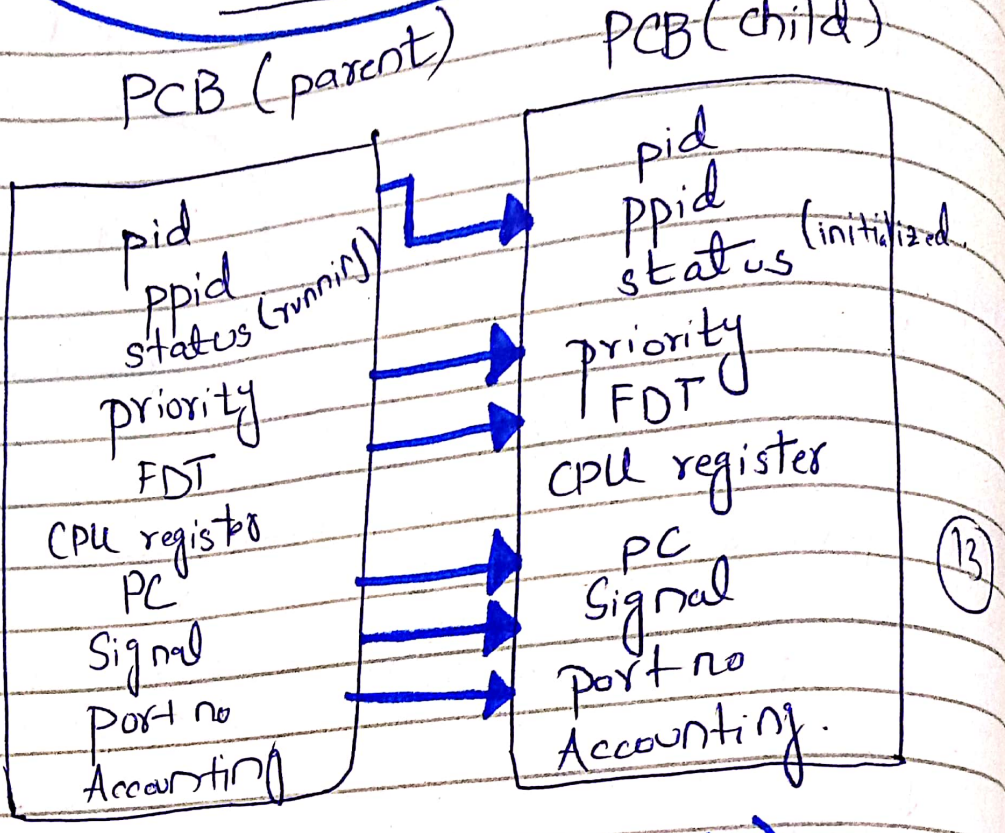


if child wants x = 20

So parent tells child to do changes in ur own memory space.

process

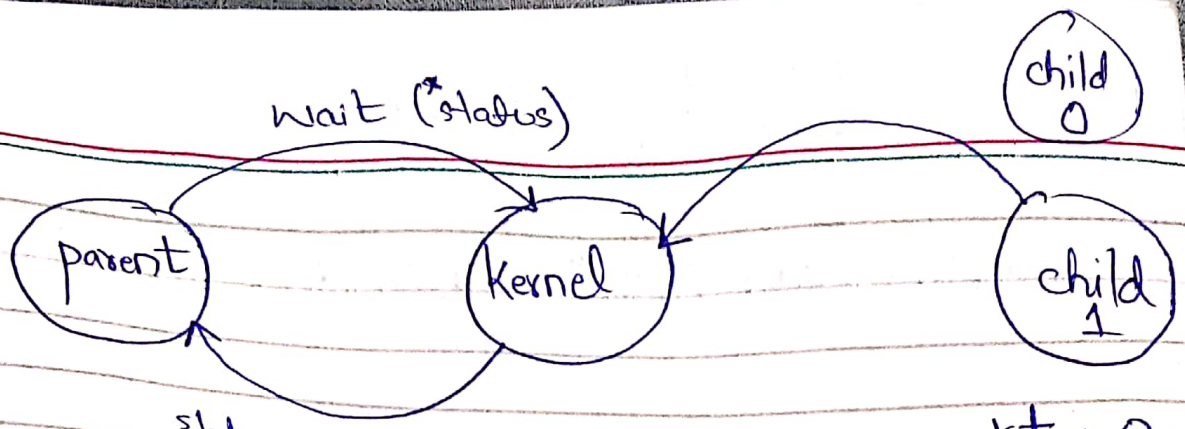
Copy on write (inheritance)



pid	initializing (state)
running (same priority)	(same priority)
FDT	FDT (child can do read/write)

Wait:

pid_t wait (int *status)



return 0;
exit(0);
_exit(0)
_Exit(0);

Wait:
parent wait for child to terminate.

→ blocking call

coz jab tak child process terminate nahi hoga wo block rahyga.

(14)

Through pid_t we can identify specific childs.

pid_t waitpid (pid_t pid, int *status, int options);

can be blocking or non-blocking call.

WNOHANG
non blocking
check currently
if terminated terminal
if not return status.

⇒ when system call fails errno ko set kia jata hai.

Lecture#4

pid_t wait(int* status);

How we will know how a process is terminated if terminated

&

1. int WIFEXITED (int status);
2. int WEXITSTATUS (int status);
3. int → WIFSIGNALED (int status);
4. int WTERMSIG (int status);
5. int WIFSTOPPED (int status);
6. int WSTOPSIG (int status);

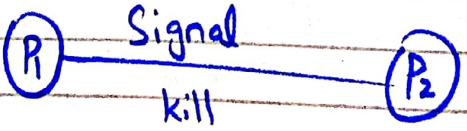
(1) // return 0-normal termination, else abnormal termination.

(2) // return 0-normal termination, otherwise returns last 8 bits of status.

⇒ For last 8 bits we make another call. we find cause of error.

int → 4 byte
last byte → error

(3) when P1 sends short msg to P2.



1 process can kill another

if we want to know child ko kill kia ha ya khad mara

We get this inf from here.

// if 0 - not killed by a signal otherwise killed by signal.

④ There are multiple signal types jo batata ha kis signal ne kill kia ha.

// return the signal number used to kill process.

⑤ // returns positive number if process is stopped/suspended.

⇒ We have multiple signal which stops the process

⑥ // return the signal used to stop the child process.

Sleep(30); jis process ne execute kia wo 30 sec sleep me chala jata ha

* Exec

```
#include <unistd.h>
int main() {
    int pid = fork();
    if (pid == -1)
        perror("Error");
    else
        if (pid == 0)
            execvp("childProg", "5 7 Hello");
        else
            wait(NULL);
    return 0;
}
```

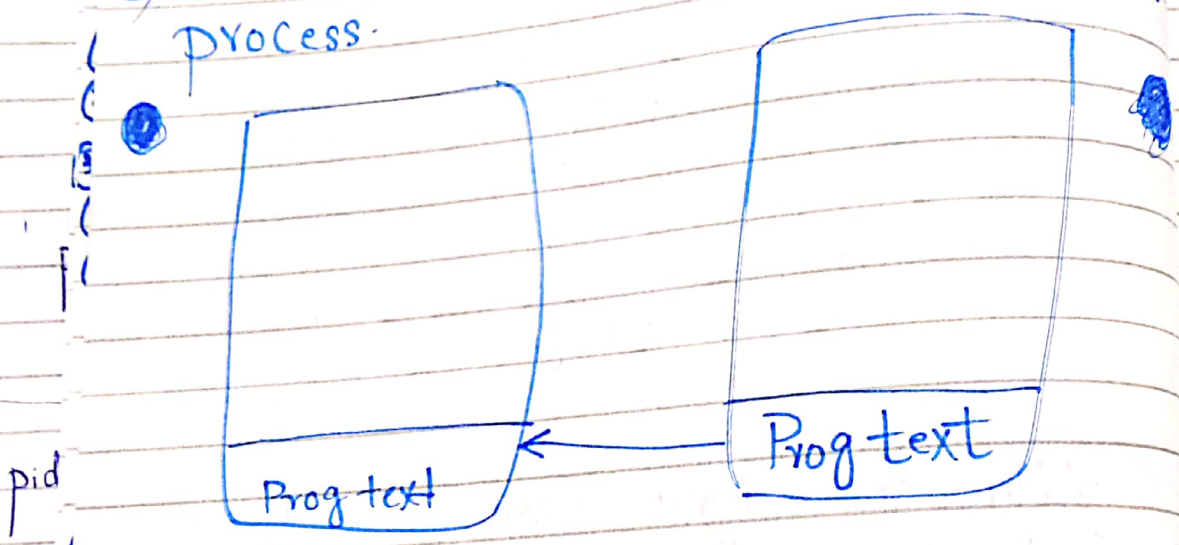
// child prog.c

```
int main()
{
    printf("Child prog has been called
    using exec );
    return 0;
}
```

(17)

exec (system call)

⇒ It starts new program on calling process.



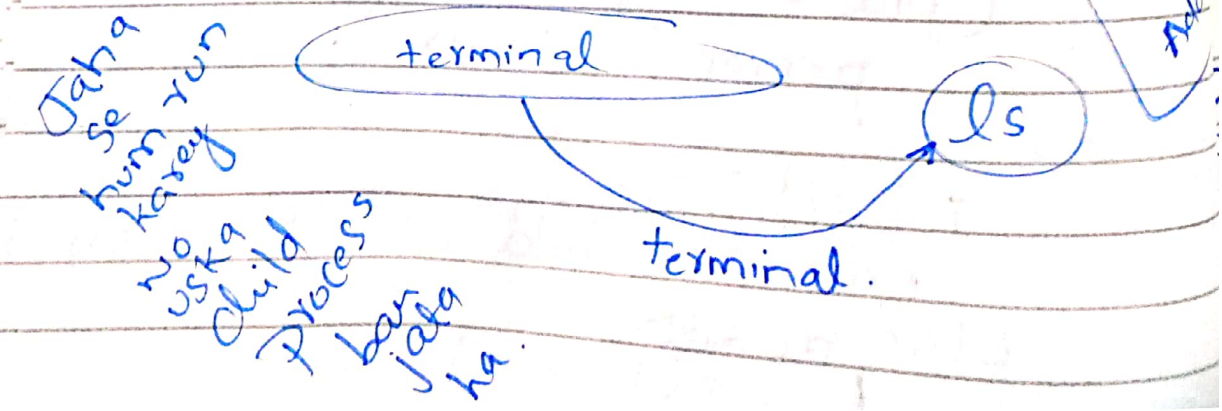
⇒ Flushes out the memory image.

basically bash program.

e.g ls -l

↓

call from terminal.



Scanned by CamScanner

Memory Flush out :

Inorder to avoid Memory Leak.

① ⑯ execl ✓

int execl(char * path, char * arg0, ... , char * argv)

②

int excle(char * path, char * arg0, ... char * argv, char * envp);

③

int execlp (char * File, char * arg0, ...

④

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

⑤

int execve (char * path, char * argv[], char * envp[]);

⑥

int execvp (char * File, char * argv[]);

① • path
• argument list

// execl("/usr/bin/ls", "ls", -1)

② path, list of arguments, environment v

// excle("/user/bin/ls", "ls", "-l", envp);

③ // execlp("ls", "ls", "-l");

④ execv("/user/bin/ls", argv);

⑤ execve("/ver/bin/ls", argv, envp);

⑥ execvp ("ls", argv);

Lab #3.

Process:-

Process
↓
hierarchal Structure

init
↳ initializes system.

(19)

first process with id (1).

* Har ek process kisi ha kisa process ka child hoga

init
↙ ↘
login email.
↓
parallel execution starts.

t1.0

terminal → input
↓
output

ps → list of currently executing process.

Desktop/SP/t1.0
1217 ← ye isko execute karegga.

shell hamare process ko execute karta hq

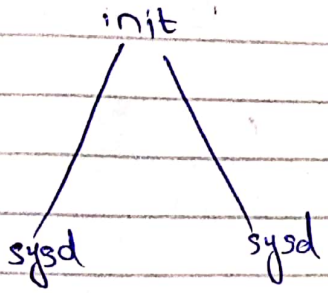
Jis process ka parent un k execution se pehle die kar jaye tho that process orphan.

⇒ When parent is not waiting for child then → Zombie

parent wait hdki kar raha hota & its lifetime is from when we system till switching off it.
switch on the
↳ Demon.

init systemd ①

why ① it should be init.



ps -e
- returns status of all alive processes

ps -e | grep systemd.

process tree | pstree.

wid id | pstree -p

Process creation is responsibity of OS.

fork();

int fork();

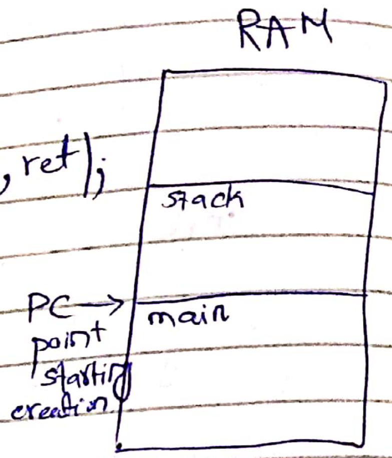
fork();

error : -1
success :
fail :

program starts from main.

error.
creation reason

```
void main()
{   int ret= fork();
    printf ("My ret= %d", ret);
if (x==0) // child
{
} else if (x>0) // parent
{
}
```



:) jab ftn call hota ha yaha se jump kar k function code par chalay jatay.

PC→ fork().

PI

When ftn ends tho, wo wapas apni jaga pay address ki waja se aye jo stack me store hai.

Parent and Child k darmian kuch share nahi hota by Default.

Child is exact replica of parent.

child jb create hojaye wo child Memory image me return karta.

Success :

1) > 0

2) 0

22

Ek dafa parent
↓
gives childID
and dosri child.
0 returns

Process Termination.

↳ Normal → exit → impliait exit
expliit exit

↳ Abnormal.

↳ signal

kill -l list of signal.

return 0 → process terminate nahi hota
↓
only returns fun.

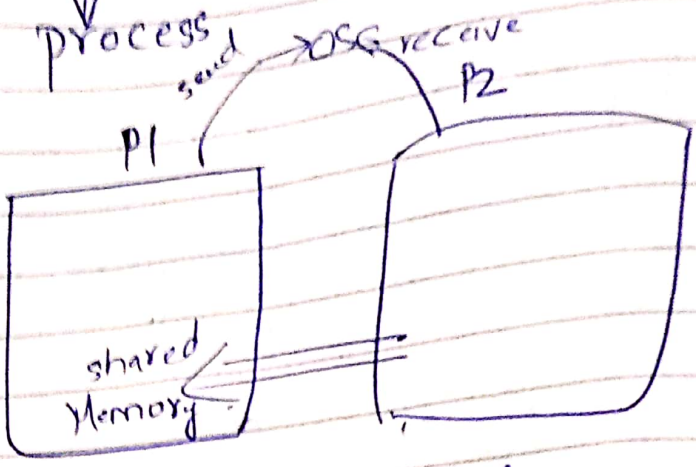
main explicitly add exit to prg

int exit (int a);

error : -1

success : other than -1.

Intercommunication



:) Jis process ne communicate kia hota ha wo attach hoja ha.

Communication.

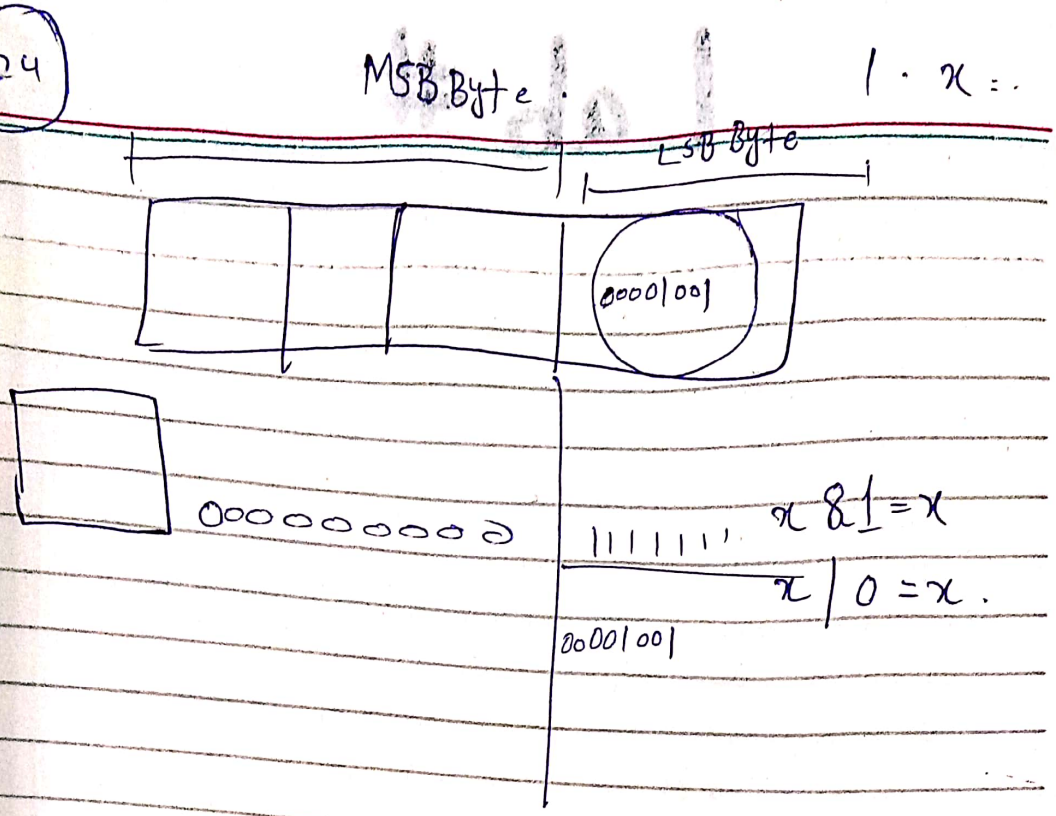
- Shared Memory
- Message.

exit (9) ke through we informed the parent.

how parent will receive it through wait (&y);

```
{ int y.
printf("my rid = %d", rid)
```

24



WIFEXITED → normal.

WEXITSTATUS → return

```
void main (int argc, char argv[])
{
    int r;
    for(i=0; i<atoi(argv[1]); i++)
    {
        r = fork();
        // error check.
        if(rd>0)
            break;
    }
    printf("My ID: %d \t My Parent ID = %d
    printf("My i : %d \t My ID: %d \t My parent ID =
        %d \n", getpid(), getppid());

    return 0;
```

Lecture 5

14/10/2019

↳ system (char* str); // system call.

→ It start a child with new memory image
→ creates a child process
→ Child process does not inherit data from parent's memory

↳ It is used for running script commands.

system ("ls -l");

Chapter 4

Unix I/O

→ Device Handling
→ File Handling

→ open, read, write, close, ioctrl.

include <fcntl.h>
include <stat.h>

⇒ Open a file :-

Open (char* path, int oflags);

// int fd;
// fd = open("file1.txt", O_RDONLY);

int oflags:

O_RDONLY
O_WRONLY
O_RDWR
O_CREAT
O_CONCAT
O_NON_BLOCK ↳ O_BLOCK

fd = open("file1.txt", O_RDWR | O_CREAT);

fd → file descripter.
fd → file1.txt

O_CONCAT → Waha se writing start karu jaha data khatam ho rahi ho.

O_flags → it is a request to open a file for reading, writing etc.

↳ READ A FILE | DEVICE:

int read(int fd, char* buf, int nBytes);

file → buff (Data).

buf → holds the data returned by read (it is a buffer)

nBytes → how much data you want to read.

data size that we intend to read.

// read (fd, buff, 1000);

→ -1
→ 1000
→ 570
→ 0
→

except -1, all of them are Successful.

i-e system call does not fails except -1.

Program:

```
int main()
{
    int count = 0
    int fd;
    char buf[10];
    int bytesread = 0;

if (fd = open ("file1.txt", O_RDONLY == -1))
        perror ("Error");

    else
    {
      do {
if ( bytesread = read fd, buf, 10) != -1)
        count += bytes read ; }

while( bytesread != 0) {

printf ("Total no. of char in file1.txt : d"
} return 0;      , count );
```

```
if (bytesread = read (fd, buf, 10) == -1 && errno == EINTR)
{
    bytesread = 0;
}
else
{
    break;
}
```

Lecture # 6

10/16/2019

Write (int Fd, char* buff, int size);

- Fd → File description jo writing keliye open kia ha.
- buff → stores data.

```
int open("File1.txt", O_WRONLY);
fd=
```

Program:

buff → file (Data)

```
int main() {
    int Fd;
    int bw1 = 0;   // in bw = 0
    char buff[] = "Hello World!";

    Fd = open("File1.txt", O_WRONLY);
    int bw = write(Fd, buf, 10);
                   [bw1-1] 13-bw1.
    write(Fd, buf[bw1-1], 13-bw1];
```

⇒ 12 se kam means save write nahi howe.

27

```
if (bw == -1)
    perror ("Error");
```

// if 12 characters na write howe hu
// then

```
if (bw == -1)
    perror ("Error");
else
    bw1 += bw;
} while (bw1 != 12)
```

① ② → -1 ③ → 12 → 0 > < 12

This is blocking call (thabtak wapas nahi ayega jab tak kuch write na karle tabhi 0 nahi deta

Linux Command

Copy data:

argv[1] argv[2]

$ cp File1.txt File2.txt

argv[0]

→ copy File1.txt to File2.txt

→ File1.txt & File2.txt are passed as arguments.

2 arguments

~~1st argument~~

bw = write(Fd, buf, 12-bw1);

for control buffer

→ wo hamesha starting address pe na for this we need to update buff buff[bw1-1], 12-bw1);

-> create file if it doesn't exist.
-> give error if less than two arguments passed.

Program * mycp.c

return 0 : coz it is not an error.

mycp file1.txt file2.txt

```
int main(int argc, char* argv[]){
if (argc != 3) {
    printf (".\usage : arguments mismatch");
        return 0;
```

Kernel pointer gets updated by itself.

```
    int Fd1 = open ("file1.txt", O_RDONLY);
int Fd2 = open("File2.txt", O_WRONLY, O_CREAT);
    int br = 0, char c;
do {
if ( br = read (Fd1, &c, 1), != -1)
    write(Fd2, &c, br);
while (br != 0)
```

File1 for reading

store 1 character.

[we want to copy character by char.

-> if not ampers.. then call by value hogi & wana address nahi hota.

br me 0 thab aye gg jab read/write khatam hoga.

When file empty then 0

char c[10];

if(br=read(Fd1,c,10)!=-1)

write(Fd2,c,br);

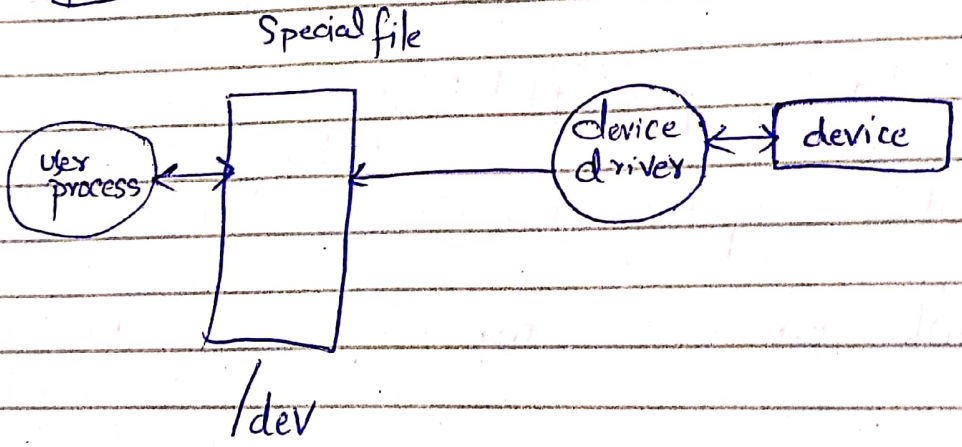
} while(br!=0).

if we dont want to read char by char.

29

if read fail hojai ya read successful howa 1
read hogaya but write fail howa so it will be read next char.

How to Communicate with Devices in UNIX:



if we want to print something (write) system call is used.

ssty01 ↓ usb port

Microphone echo → Speaker.

1st time Read
4/5 times (control echo).

(30)

int Fd1 = open (/dev/printer , O_RDONLY).

Code a device Driver.

For read and write.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <error.h>
int main()
  {

int fd = open("/dev/scanner"
              O_RDONLY);

int fd = open("/dev/printer",
              O_WRONLY, O_CREATE);
                        ,(S_IRUSR);

  char buf[] = "Hello world";
  int byteswritten = write(fd, buff, 12);

if (byteswritten = write(fd, buff, 12)) == -1)

      perror("Error");

   or

      strerr(errno);
  else
      close(fd);

  return 0;
}
```

Every device in a computer is represented by special file.
Special files → on the other side there is program device driver.

/dev contains those special files.

int fd1 = open ("/dev/printer", O_RDONLY);

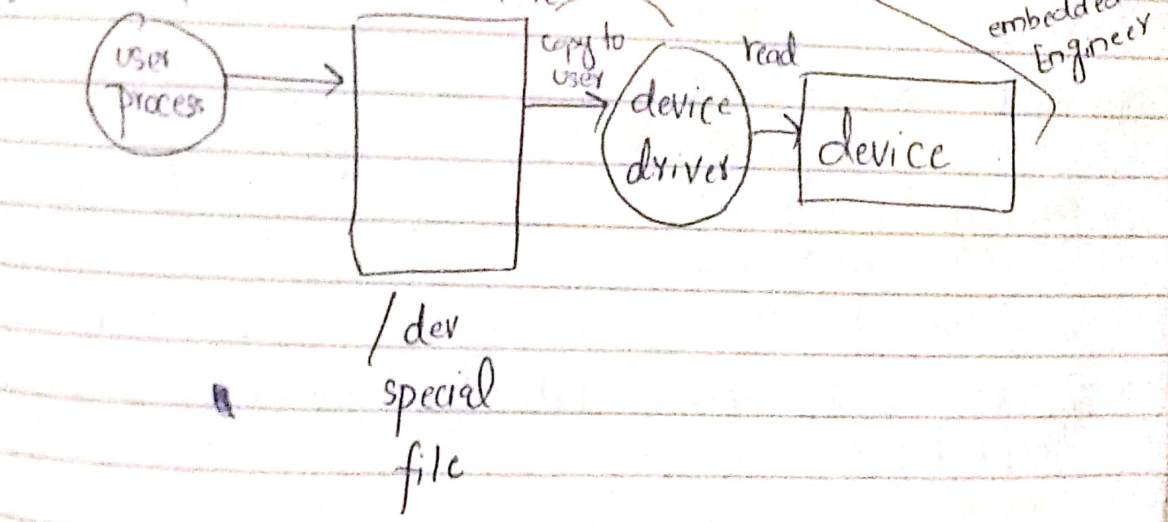
Lecture

register_chdev (int major, int *name, struct File_operation *Fop);

Application developer

System Engineer

Embedded Engineer



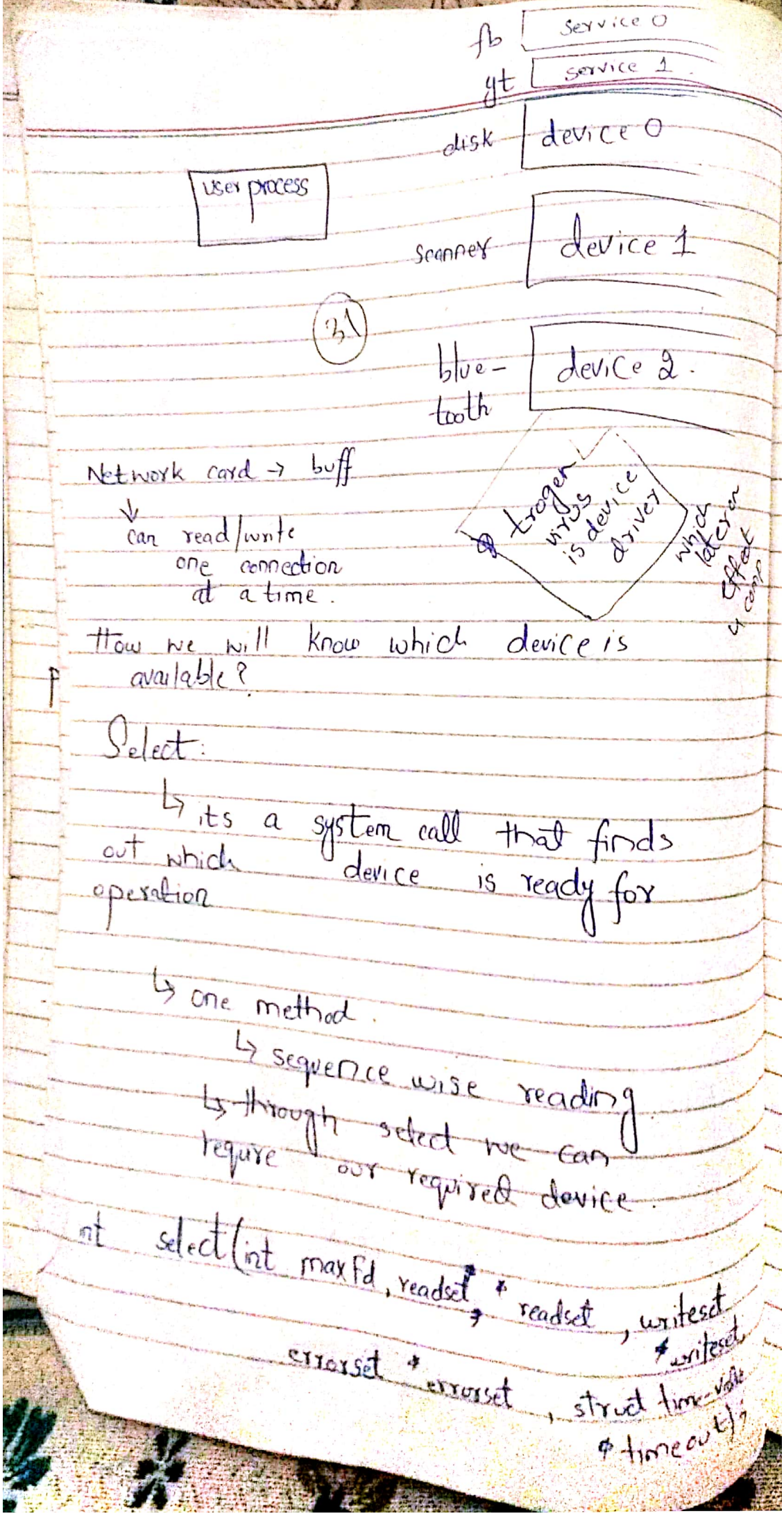
/dev special file

⇒ open
⇒ read
⇒ write
⇒ close

file_read (file *f, char* buff, size, ———)

When we call this, buff reads data from device driver

buff
↓
memory
↓
kernel device memory.



⇒ select (system call)

Take 5 perimeters

maxFd → Maximum Fd value. + 1. → Kernel uses this for optimization

readset *readset → Fds for reading (contains)

32

writeset *writeset → contains Fds for writing

error-set → contains Fds for error.

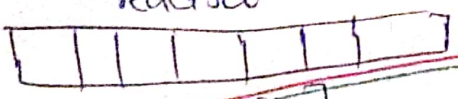
struct tim_val → seconds & milli

Wait for certain time for a device if it is ready otherwise return back.

⇒ -1 → system call fail.
⇒ 0 → time out
⇒ 1 → 1 device ready
⇒ 2 → 2 devices ready.

Program :

```
#include <select.h>
int main() {
    int fd1 = open("/dev/scanner", O_RDONLY);
    int fd = open("/dev/bluetooth", O_RDONLY);
    readset  readset;
```




```
char buf[20];
FD_ZERO (& readset);
FD_SET (Fd1, &readset);
FD_SET (Fd2, &readset);

int max = fd1 > fd2 ? Fd1 : Fd2;

int nFds = select (max+1, &readset, NULL, NULL,
                   NULL);
        ↳ ready file will be
          returned.

if (nfds == -1)
        perror ("Error");
else if (nfds > 0)
    if (FD_ISSET (fd1, &readset))
        read (Fd1, buf, 20);
    if (FD_ISET (fd2, &readset))
        read (fd2, buf, 20);
else { }

FD_RESET (& readset);
return 0;
}
```

Poll

31st / Oct / 2019.

↳ system call and can be used to monitor devices / Files for reading, writing and error.

```
int poll(struct pollfd[], int nfds, int timeout);
```

time out: ↳ time in millisecond

↳ not compulsory if device available ha y nahi

↳ So it waits for sometime

↳ if available hojai so operation perform hota ha.

↳ No device available returns 0.

nfds:

no. of fds to be monitored.

(34)

int poll:

-1 call failed

0 → none of the devices ready.

>0 → devices ready.

diff pollfd for diff files

Poll fd:

```
struct pollfd {
    int Fd;      → file descriptor
    int events;  ← events to be monitored
    int revents;
};
```

(POLLIN, POLLOUT, POLLERR) → Flags

read, write.

010
100
110

Here you can do R/W at same time in one variable if we take ; its bitwise OR. events.

How much flags in int?

32 bits

Each bit can act as flag.

bool Flag

↓

bit.

#define POLLIN 2

#define POLLOUT 4

↓

#include <select.h>

if kernel got 2 & 4 both then reading & writing both actions are performed.

Revents:

↳ event for which the device is actually ready.

000 — nothing ← timeout / system call failed.

001 — POLLIN

100 — POLLOUT

110 — Both Ready

Diff Poll & Select

timeout int milliseconds	time out struct microseconds.

Program

part of select → struct pollfd
{
int fd;
int events;
int revents;
}

(30)

```
int main(){
    char buff[100];
    int fd1 = open("/dev/printer", O_WRONLY);
    int fd2 = open("/dev/bluetooth", O_RDONLY);
    struct pollfd pd[2];
    pd[0].fd = fd1;
    pd[0].events = POLLOUT; // writing
    pd[1].fd = fd2;
    pd[1].events = POLLIN; // reading
    int nfds = poll(pd, 2, 100);
                    ↓     ↓
                  files  timeout
if(nfds == -1)
    perror("Error");
else
    if(nfds == 0)
        perror("No device ready
                for operation");
    else {
        if(pd[0].revents == POLLOUT)
            write(Fd1, "c", 1);
        if(pd[1].revents == POLLIN)
            read(fd2, buff, 100);
```

100 millisec
↓
100 million instructions

reading

```
int fd2 = open("/dev/bluetooth", O_RDWR);
struct pollfd pd[2];
pd[0].fd = fd1;
pd[0].events = POLLOUT;
pd[1].fd = fd2;
pd[1].events = (POLLIN | POLLOUT);
int nfds = poll(pd, 2, 100);
// error checking
else
{
if (pd[0].revents == POLLOUT)
    write(          );
if (pd[1].revents & POLLIN) == POLLIN)
    read(Fd2, buf, 100);
if((pd[1].revents & POLLOUT == POLLOUT)
    write(fd2, "Hello", 5);
    return 0;
}
```

37

POLLIN - 10
POLLOUT - 100
revents - 110

```
 110        110
 110 &    100 &
 ---      ---
 010      100
```

110