
G
O
D
O
O
O
O
D


G
O
D
O
O
O
O
D

operating systems :→

Programs are of two types
- → System Programs
- → Application Programs

- ⊙ System programs manages the operation of the computer
- ⊙ Application programs does what user wants. It is a kind of user facing program with which user interacts.

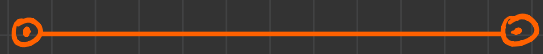
→ Application programs exsist because otherwise programmers will have to do all things. For ex :→ Reading a

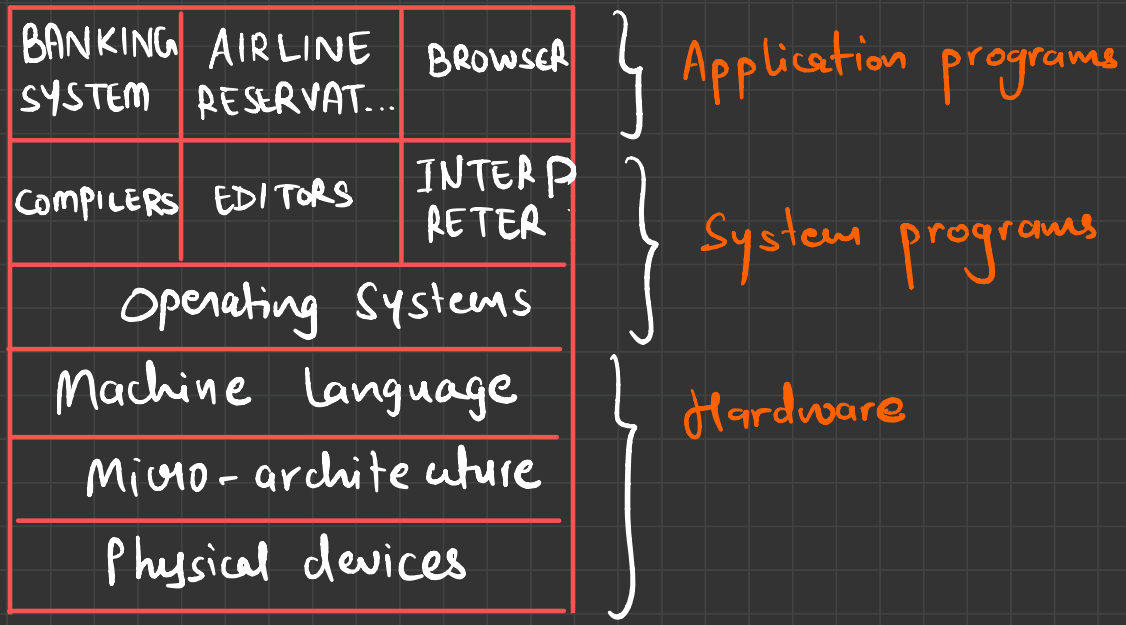
file from the disk, and also considering what can go wrong while reading a disk block.

That's why an interface was needed with which user can interact and do his work

So, the most fundamental system program introduced is OPERATING SYSTEM.

Application programs are built on top of the OS.





Kernel mode OS is that portion of sys program which runs in Kernel mode or supervised mode; which means user programs can't temper the hardware becz user prog. runs in user mode.

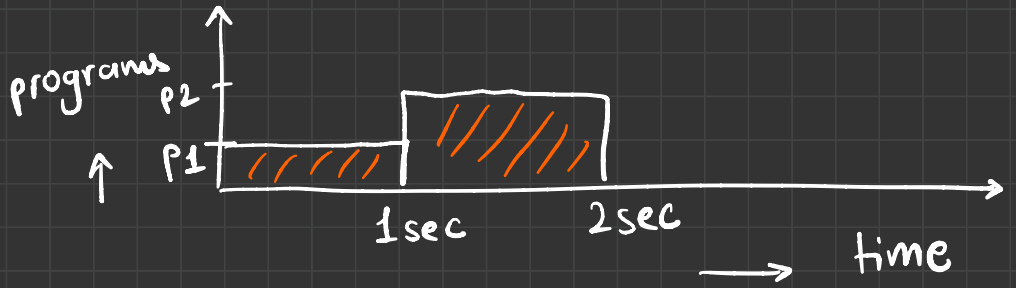
User mode generally runs appl^{n} programs in itself & interact with system programs running in Kernel mode, which means user mode's programs can't change or alter anything in hardware directly bcz that only Kernel's mode programs can do.

Operating System :→ is a system program which runs in Kernel mode and talks to appl^{n} programs and manages the resources. It makes the life of a programmer easy, for ex: instead of programmer adjusting disk heads, blocks to read data, we have file system as an interface using which user can access files & folders

Operating system handles so many complex operations.

And resource managment includes multiplexing

- Time multiplexing
- Space mult.



* scheduling processes on CPU

[Time multiplexing]

* scheduling / providing mem. to programs

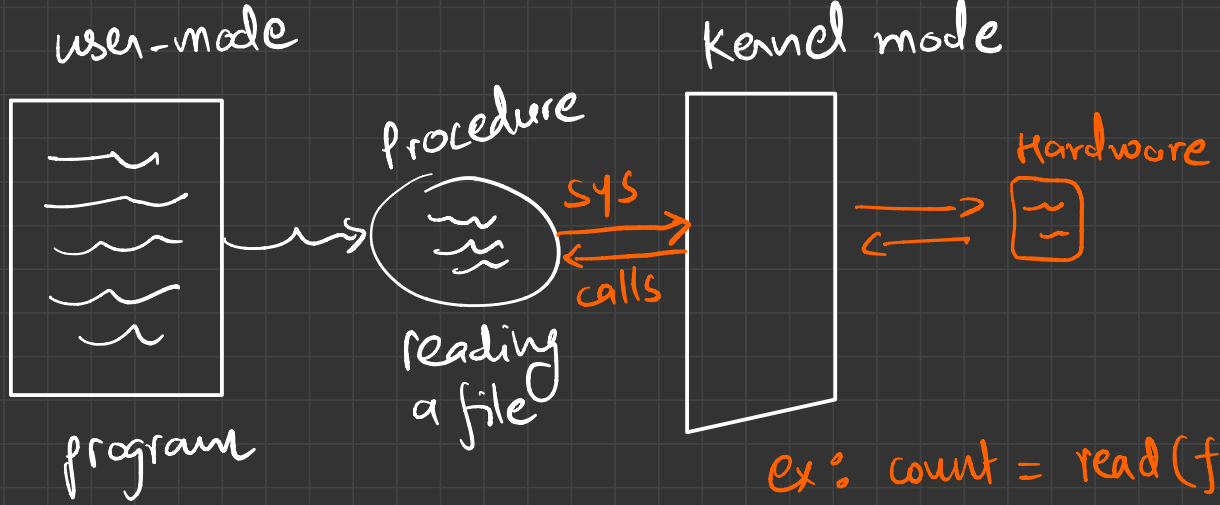
[space multiplexing]

MINIX we will study MINIX OS and will code also if possible so that everything we'll understand in a great detail and can benefit from it lifelong

MINIX is inspired from UNIX OS because UNIX - did change license to not use it for any - University course OR study purpose.

It is written in C-language. MINIX inspired, LINUX originated from MINIX only after a person named "Linus Torvalds" made few changes in MINIX and Linux 1.0 born on March 13, 1994.

① **SYSTEM CALLS** :-> whenever user/appl" programs wants to run a program that needs to interact with hardware, for ex :- if our program wants to read a file from our Hard disk, it has to go to kernel mode to accen the resources.



ex: count = read (fd, buffer, nbytes).

SYSTEM CALLS FOR PROCESS MANAGEMENT :→

pid = fork() → creates a child process identical to parent

pid = waitpid (pid , &staloc , opts) → wait for child to terminate

s = execve (name , argv, envp) → replace a process core image

exit (status) → terminate process execution and return status

size = brk(addr) → set the size of the data segment

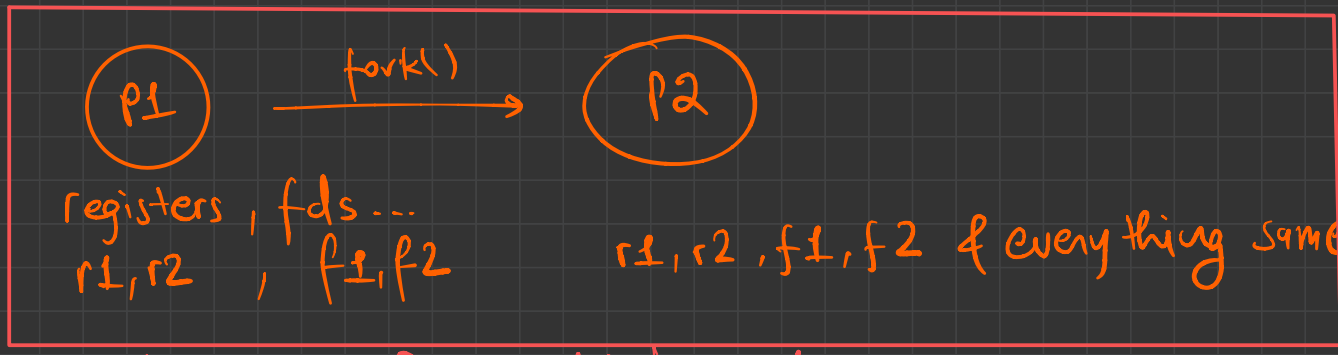
pid = getpid() → return the caller's process id

pid = getgrp() → returns caller's process group id

pid = setsid() → creates a new session & returns its proc. group id

l = ptrace (req, pid, addr, data) → used to get trace & debugging

⊙ fork()

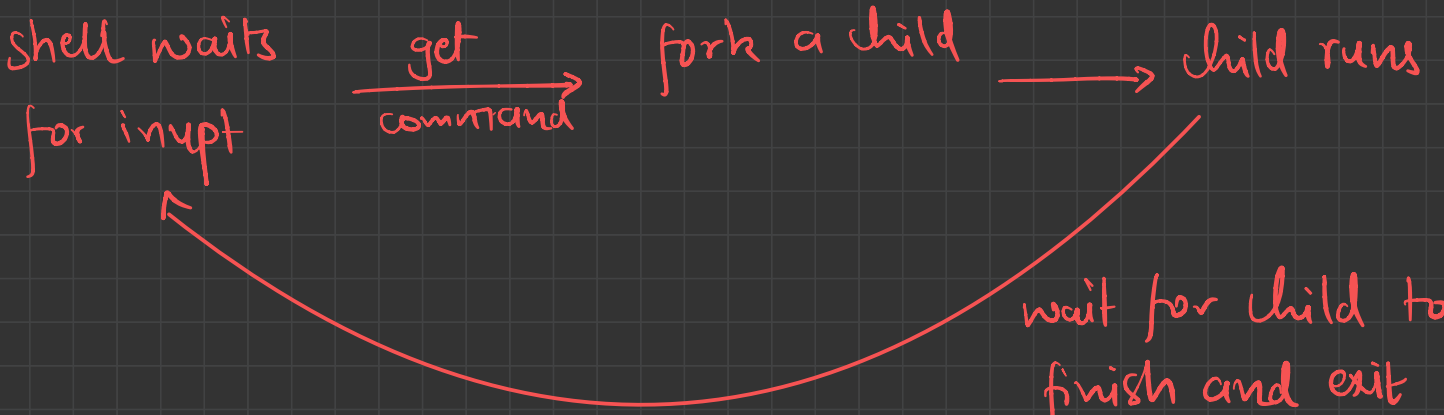


→ at the time of forking everything is same including registers, fds & variable's values.

→ when both runs, they run separately and state change of one doesn't effect another.

pid = fork() returns 0 for child & child's PID for the parent.

running a command in shell :→



{ to wait, parent executes a waitpid system call }

waitpid() waits for any, child to terminate
specific

→ waitpid (pid, &Statloc, opts)

→ after child process's completion the address pointed out by "statloc" will have child's termination - status. { normal, abnormal or exit value }

Now, when child executes the user command → It does so by calling execve system call

High Level working :→

[C-program]

parent waiting ←

child executing the command ←

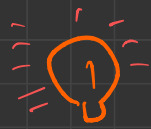
```
#define TRUE 1
while (TRUE) {
    typeprompt();
    readCommand();
    if (fork() != 0) {
        waitpid (-1, &status, 0); }
    else { execve (commd, params, 0); }
}
```

execve(name , argv , envp)

name of the file to be executed

pointer to the arg array

pointer to the env array



Note :-

we will see this later on too.

→ exit (status)

↳ It has value 0-255

→ It is used by the processes to exit/terminate and status returned will go/store to the adds pointed out by &statloc. & will be returned to the parent-

→ lower value like 0, means normal exit/termination and higher values represents some errors

SYSTEM CALLS FOR SIGNALING :→

A process can use sigaction() syscall to announce that it is ready to listen signals

Ctrl+C

process state stored

signal state pushed to a signal handler that process gives the address of while calling sigaction()

- Signals can be blocked in MINIX3
- Blocked signal are held pending until unblocked
- sigpending() call returns a set of bitmap {0, 1, 1, 0} to represent the signals which are pending and blocked
- sigprocmask() call allows a process to define a set of bitmap for kernel to tell the blocked signals
- Insted using a function by a procus, the processes can define a constant SIG_IGN to tell that signals of specific type are blocked/ignored.
- SIG_DFL is to restore the default action of the signal when it occurs.

⊙ the default action of the signal is either - kill the process or ignore the signal which depends on the signal.

* while running a command in the background

"command &" ⟶ shell forks a child process (using fork)

① ↓ runs the command (using exec)

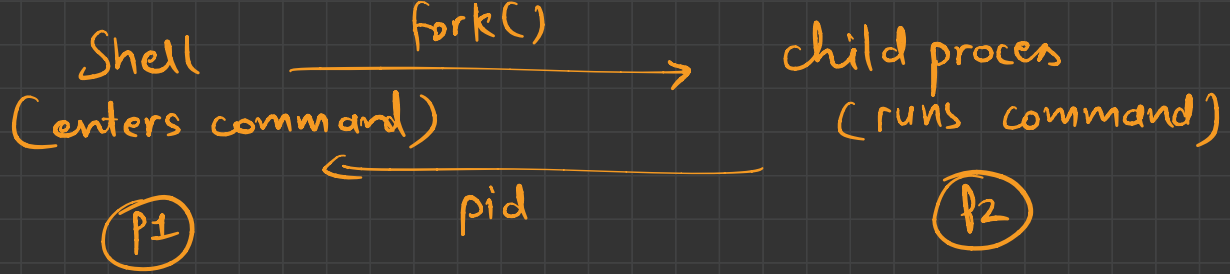
① After the fork() but before running the command - using exec(), the shell does

sigaction(SIGINT, SIG_IGN, NULL) | sigaction(SIGQUIT, SIG_IGN, NULL)

→ It blocks SIGINT and SIGQUIT signals
[because background processes shouldn't be stopped using SIGINT (ctrl+C) and SIGQUIT (ctrl+\)]

* So, for the foreground these signals will be ignored.

Q Then how a background process can be stopped ?



→ Kill system call can be used to kill a process but, to kill P2 from P1 they should be related.

→ unrelated processes can not signal each other.

→ By sending signal 9 (SIGKILL) [kill -9 <pid>]

→ In real life, many a time a process is required to do something after an interval of time. for this SIGALARM is used

SYSTEM CALLS FOR FILE MANAGEMENT :→

```
fd = creat("abc", 0751)
```

↳ creates a file with "abc" as name and with -

rights represented by 0751.

- 0 → 0 in C represents that constant is in octal
- 751 → represents rwx bits

$7_8 = (111)_2$
$5_8 = (101)_2$
$1_8 = (001)_2$

$7_8 = 7 \times 8^0 = 7 = 111$

$74_{10} = 7 \times 10^1 + 4 \times 10^0$

$112_8 = 1 \times 8^2 + 1 \times 8^1 + 2 \times 8^0$

- rwx → read, write, executable permission
- owner (7) → rwx permission
- his group (5) → rx permission
- others (1) → x permission

↓

- fd = creat(---) not only creates file but also opens it for writing through the same fd (file descrip - tor)
- apart from this, to write a file it should be open, write OR read, then close.

open() → read() / write() → close()

- whenever reading OR writing a file a pointer needs to be maintained. lseek() call can be used to move the pointer anywhere
- lskeek() → has three params where first one is the file descriptor (fd), second one tells the position

and third tells the relativity
↳ relative to start, end or current

⊙ MINIX3 stores the info like file mode [regular, special or directory, size and time of last modification. and other information.

* programs can use stat() or fstat() syscalls to get this info.

* OPTIONAL :→ In UNIX like systems like LINUX there are special files too which exists in /dev

These files can be created using a special syscall called mknod().

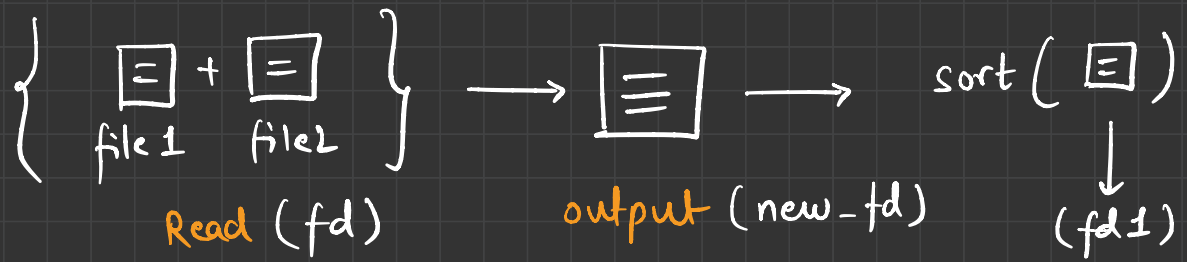
* mknod ("/dev/abc", 020744, 0x0402);

major device (sata disk)

minor device (disk 3)

Interprocess - comm" :-

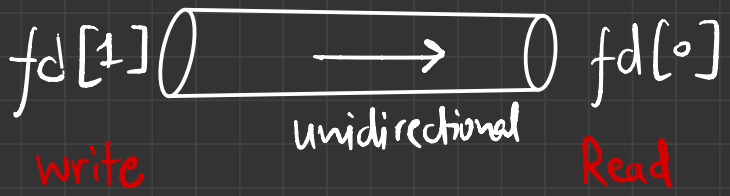
```
cat file1 file2 | sort
```

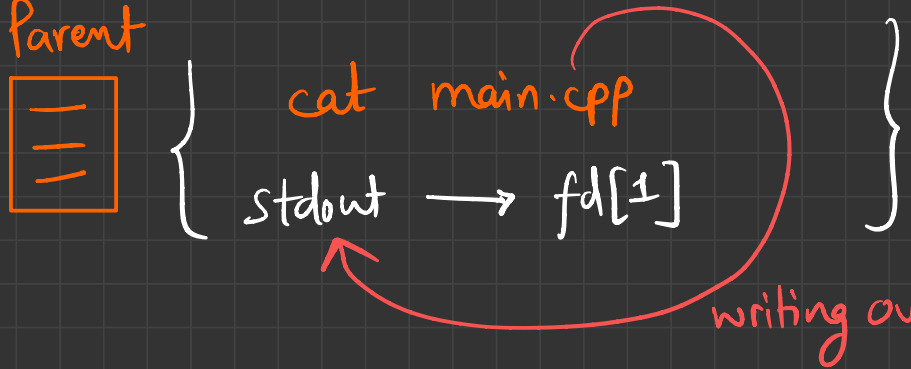


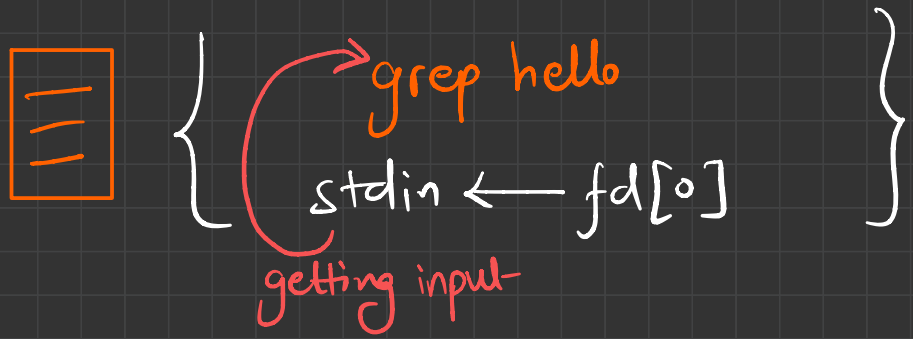
How pipes work :- [dup() sys call]

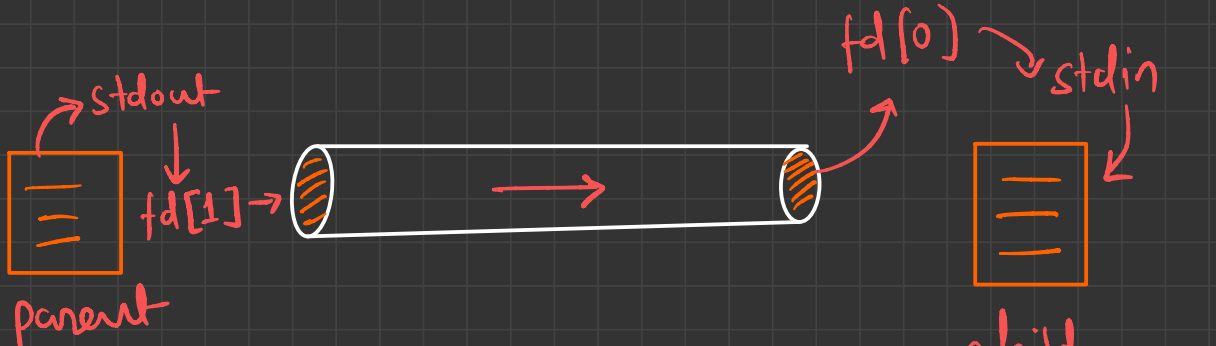
→ " cat main.cpp | grep hello "

```
int fd[2];
pipe(fd);
```









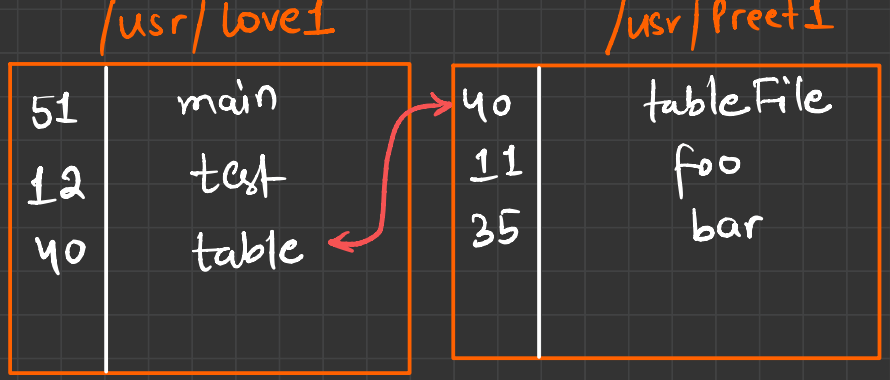
{ cat main.cpp }

{ grep hello }

{ IPC USING PIPE }

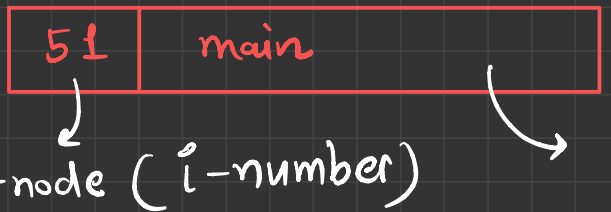
* [SYSTEM CALLS FOR DIRECTORY MANAGEMENT] *

- mkdir() → make an empty directory
- rmdir() → remove an empty directory
- link() → It creates link btw files or we can say It allows same file to appear in two different directories. (shared file)



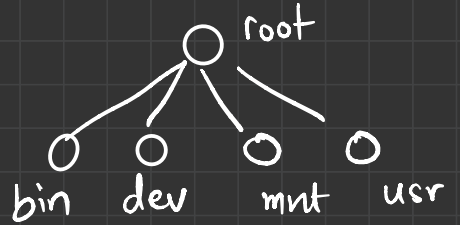
→ link("/usr/love1", "/usr/love2")

→

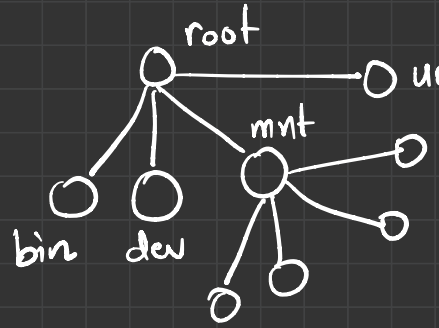


⊙ mount() → mount system call allows two file system to be merged into one.

→ mount ("/dev/cdrom0", "/mnt", 0);



→ Before mount() syscall



→ After mounting CDROM

→ chdir() → changes the working directory

→ chroot() → changes the root, all absolute paths starting with "/" will be replaced with new root directory.

* use case :→ if we want to show user1 only few files then superuser can change the root and remote users can only see the portion of file system

under this new file root.

SYSTEM CALLS FOR PROTECTION :→

→ chmod ("file", 0644)

- 100 (others)
- 100 (group members)
- 110 (owner)

→ SETUID & SETGID bits can also be set using fourth bit as :→

chmod ("file", 02755) — SETGID

chmod ("file", 04755) — SETUID

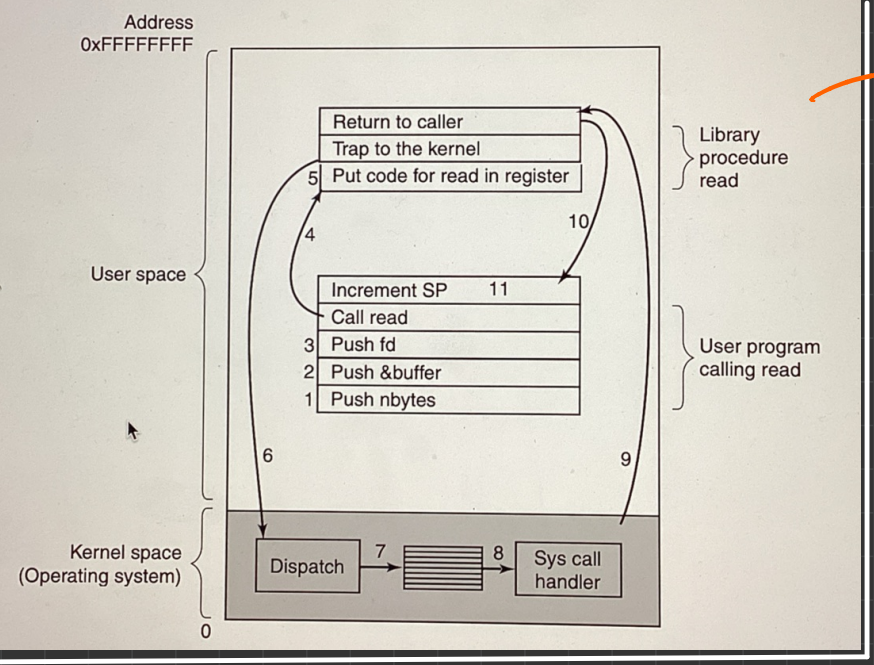
- if user runs any program with "UID" bit on then user's effective UID is changed to that of file owner. Similarly for GID.
- to get user's real UID / GID system calls present are → getuid, getgid.
- to get effective uid / gid → geteuid, getegid.
- chown() → change the owner of the file.
- umask (022); mask the bits
  like creat ("file", 0777) → creat("file", 0755)

Operating Systems :→

* **Monolythic-OS** :→ when everything runs in a single Address space. While it can have a little structure but still not very decoupled.

Sys' calls are called by using a special call i.e. Kernel call.

This call switches user mode → Kernel mode.



→ Required 11 steps to read

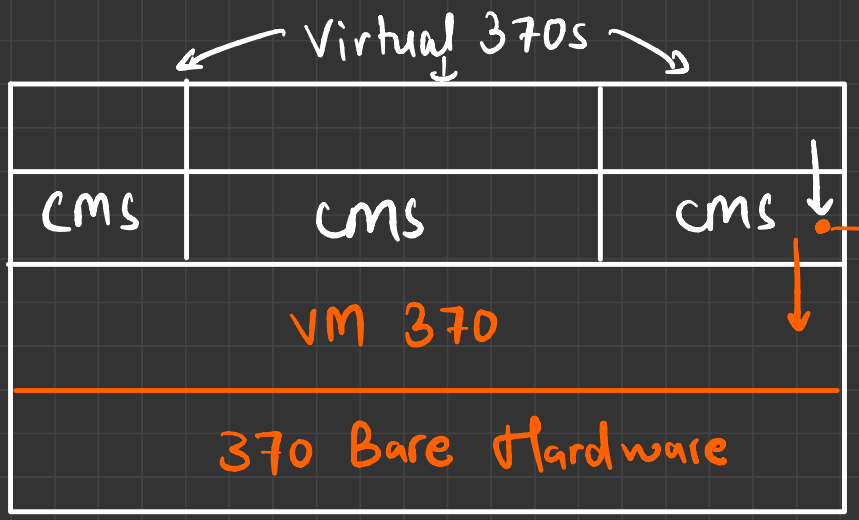
- read (fd, buffer, nbytes)

* LAYERED - OS :→

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- abstraction
- Modular & clean
- well defined structure & clear communication

* Virtual machines



CMS → Conversational Monitor system. (Single user interactive OS)

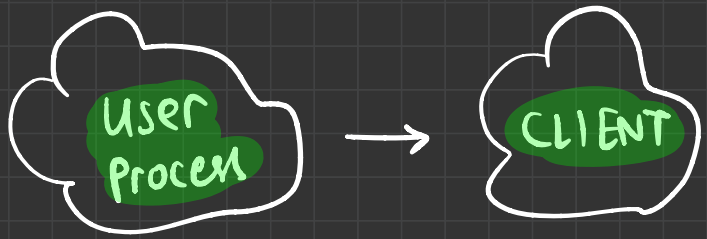
- different machines running layerwise
- This architecture helped running OLD MS DOS programs on Pentium.

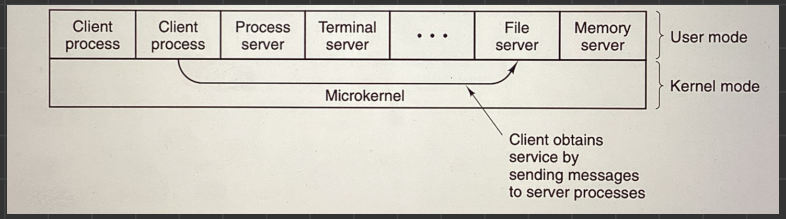
* EXOKERNELS :-> It is similar to VM architecture but at the bottom instead of kernel, exokernel runs. which allocates the resources to different VM but also isolating.

For ex. instead of providing whole disk to the VM, it might give 0 → 1023 disk-blocks to one VM and 1024 to 2047 to another

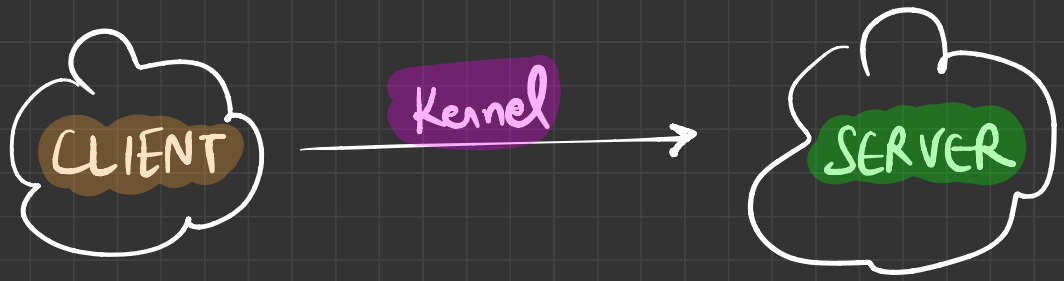
* CLIENT SERVER OS :->

- This approach was to focus on moving most of the functionality out of Kernel and make it simple.

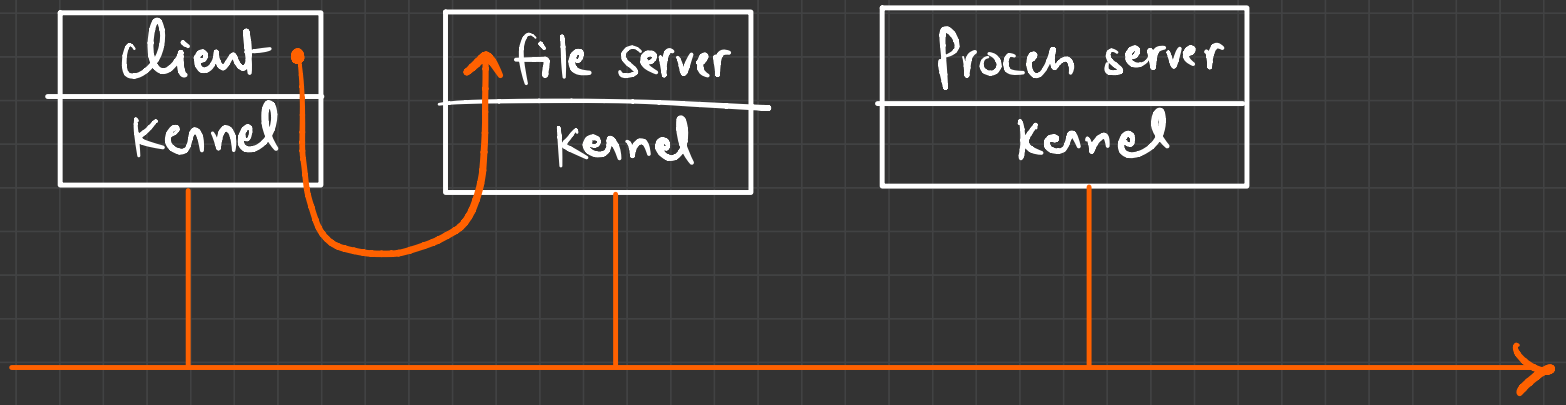




→ Kernel here now handles communication



CLIENT-SERVER IN DISTRIBUTED SYSTEMS !->



* clients are not aware if requests are being served from local or any remote program.

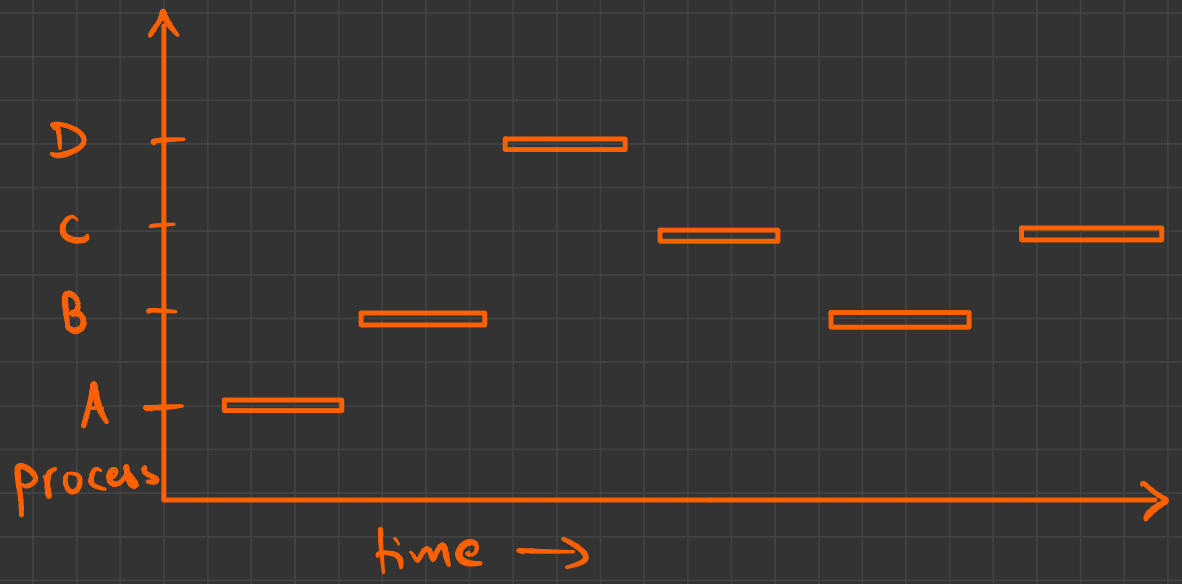
* PROCESSES o->

* Points to remember [imp] :→

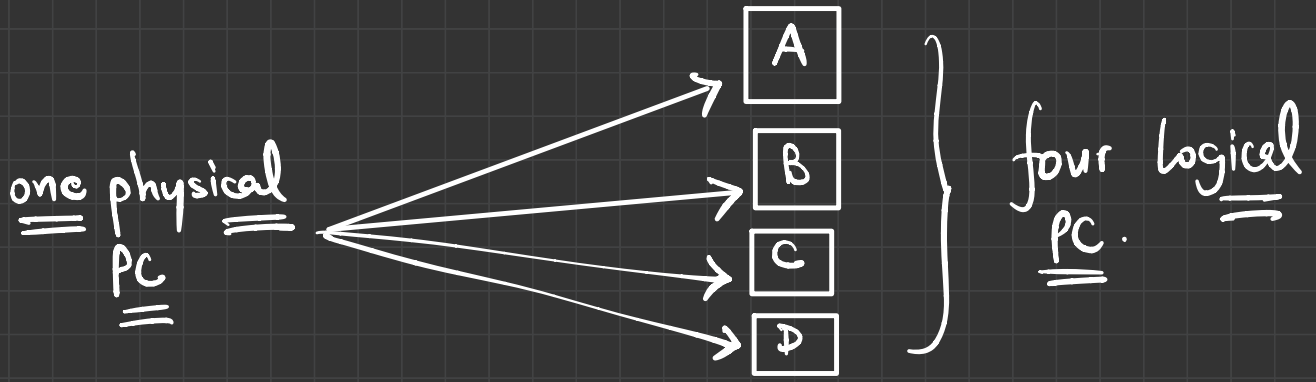
- Running Program { set of instructions } is called a - process.
- One CPU can run only one process at a time
- Using time multiplexing, different scheduling - diff. processes can run on a single CPU but it doesn't mean that single CPU can run multiple processes at a given instant.

  It is called pseduparallelism.

⊙ CPU switches back and forth btw the processes



*



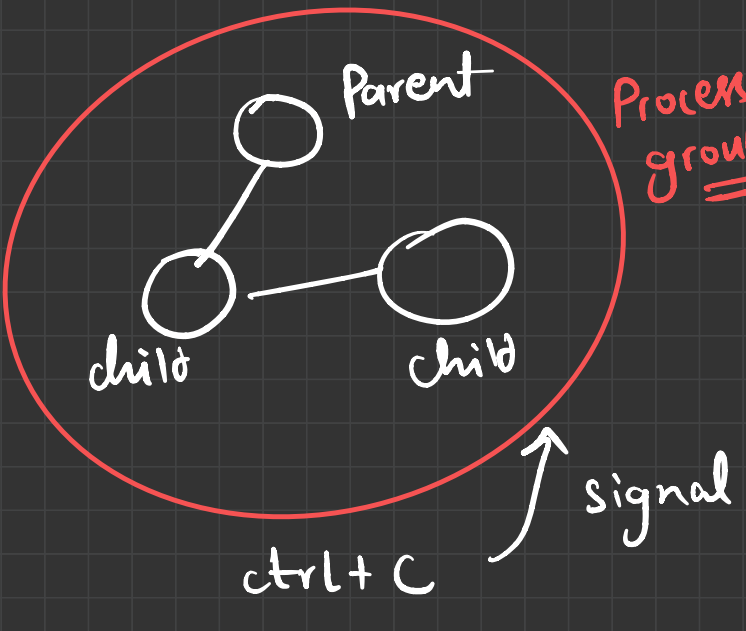
* **Process creation** :-> Processes are created at various times. for ex. when System starts few background processes might run & these are called demon processes.

for ex: fork() can create a child processes & after this call both parent & child will have same memory image. Everything will be copied but both will operate in the different address space.

Note :-> However, It is possible that child can share the open files or fds.

Process Termination :->

a) Normal exit
b) Error exit
c) fatal error
d) Killed by another process



This signal will sent to every process in the process group and it can be blocked, skipped etc.

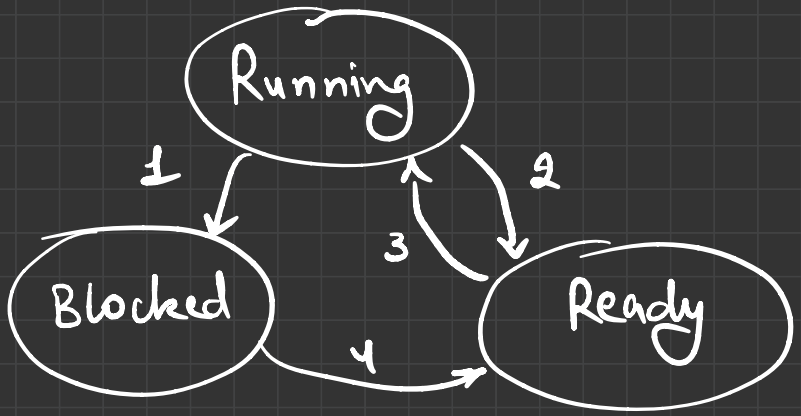
Process States :->

Each process is an independent entity having its own PC, Registers, stack, open files etc.

→ cat file1 file2 | grep tree

* It might happen that "grep" is ready but input is not available so It can not continue logically

States :-



(Running → Blocked) when input is not available process goes into waiting and wait for input

→ In some OS, process might call a sys call to go into blocked state

→ while some listen through pipe/file and goes into blocked if nothing is available

---

(Running ⟷ Ready) this happens due to - process schedular.

* if input is ready process goes (Blocked → Ready) and gets scheduled.

*** Process Implementation** :→ To implement processes, OS maintains a process table OR process control blocks. These tables OR data stru. Contains one process per row having stored process's info like :→

a) process state
b) program counter
c) stack pointer
d) memory allocation
e) status of open files
f) scheduling information

for ex :→ MINIX 3 - OS process table :→

Kernel	Process management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Real id
Current scheduling priority	Process ID	Effective UID
Maximum scheduling priority	Parent process	Real GID
Scheduling ticks left	Process group	Effective GID
Quantum size	Children's CPU time	Controlling tty
CPU time used	Real UID	Save area for read/write
Message queue pointers	Effective UID	System call parameters
Pending signal bits	Real GID	Various flag bits
Various flag bits	Effective GID	
Process name	File info for sharing text	
	Bitmaps for signals	
	Various flag bits	
	Process name	

Figure 2-4. Some of the fields of the MINIX 3 process table. The fields are distributed over the kernel, the process manager, and the file system.

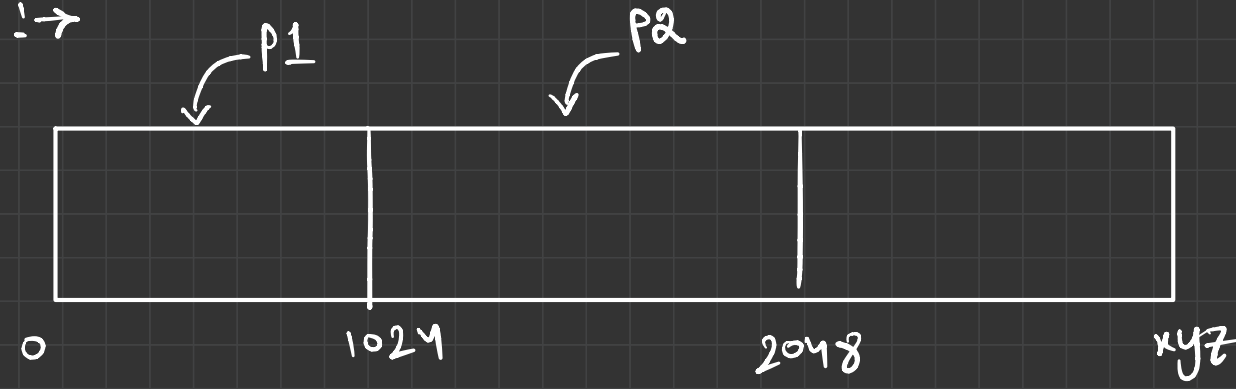
* One entry in the process table have these many field associated

* whenever an interrupt - comes and current process goes to block / ready state then all the info gets stored, so that to retrieve next time when this process will be scheduled again.

* Hardware interrupts are not handled by our High level programming languages.

* instead assembler will run a routine to handle interrupt and run other process.

THREADS :→



⊙ P1 and P2 have their own address space and we can say that they are running a single thread

of control.

- Now, a Process can spawn multiple threads to do things in parallel.

for ex.

P1

```
fd = read()
// operation on fd

int x = 1+2;
print(x)
```

} Thread1

} Thread2

* Now, these threads can do things in parallel
* as Thread 1 reads and do something on file
* Thread 2 doesn't depend on Thread 1.

Optimisations :->

a) Run T1, shift it to ready/block state till file is being read

b) meanwhile T2 can take CPU and do its calculations.

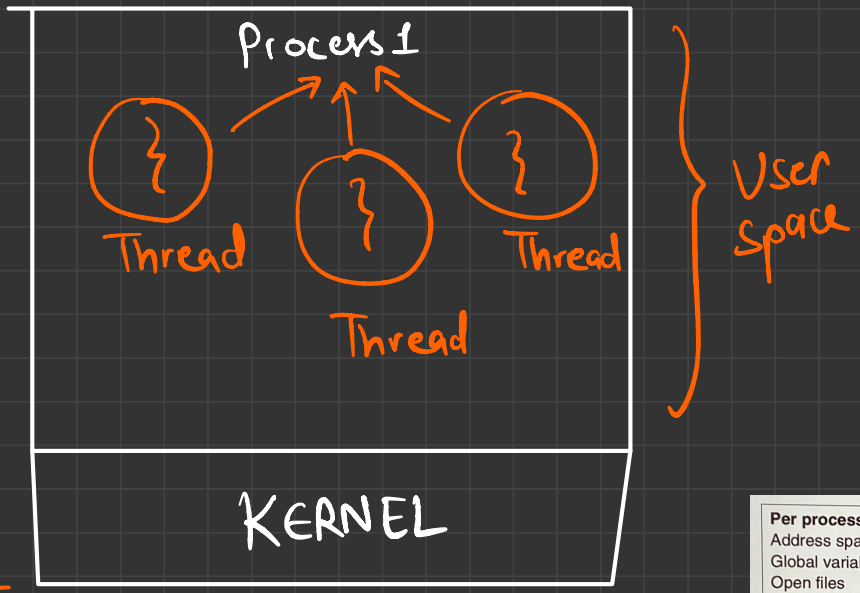
c) when file is ready, T1 can run again.

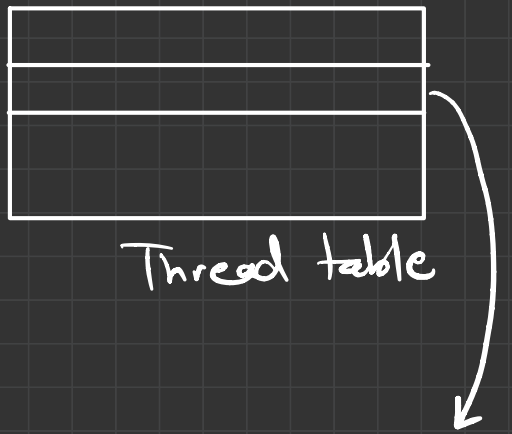
* Threads are lightweight small processes that shares the address space of the process.

* THREAD :→ It has its
  * own PC → to keep track of which instruct^{n} to run next.
  * own registers → which hold current working - variables.
  * own stack → To keep track of its execution History.

Note :→ There are two types of threads → a) Hardware
b) Software

* Virtual threads, GoRoutines uses software threads





Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-7. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

* SOME PROBLEMS TO CONSIDER

* doing thread management in User space is much faster than doing it in Kernel space because sys calls are expensive.
* But doing everything in User space has downside
  - → if one thread blocks for some I/O
  - → Kernel would block the entire process
  - → because Kernel is not aware if process has other threads waiting and they could be scheduled meanwhile
* So, Hybrid approach of both above is used mostly

* if P1 has 3 threads, and after fork() call
  - → should child have all of them too
  - → if yes, what will happen if any of them gets blocked on I/O [input from keyboard]
  - → if user types, will both threads of child and parent will get the input copy?
  - → if one thread is writing a file and other is reading. How to handle this?
  - → if threads are running at user space level,

how signals will travel from sys call to the specific thread ?

→ because Kernel is not aware about signal to send to which thread if they are running in the user space.

→ All of these things gets complicated after the introduction of threading model.

→ Some kind of backward campatibity needs to be present.

* INTERPROCESS - COMMUNICATION :->