IPC Notes

* mkfifo (const char * pathname, mode_t mode)
  0 / -1
  #include <sys/types.h>

→ The moment Gen execute, the FIFO file gets created and makes a entry into linux kernel operating system's file system.
→ FIFO file is a file which have all the attributes any process can open it for reading and writing like any other normal regular file.
→ Ensure that FIFO is opened from both the ends simultaneously. If a process opening FIFO for reading has to wait too otherside process to open FIFO for write operation. and viceversa
→ A FIFO can have multiple readers and writers and supports connection oriented comm protocol.
→ FIFO is used for simple client server comm.

* msgget():-

The moment process call msgget system call, a new message queue is created in kernel space. with a unique message queue id. for newly created message queue a structure called msqid_ds allocated with default initial values.

msgget (key_t,

arg 1) When a process requires for resources such as shared memory, m.q semaphores os needs a key of type key_t. which supposed to be int tancully int.

→ defining #include <sys/stat.h>

arg 2) int flags, IPC_CREAT

returns m.q.ID for a new m.q. created.

returns a m.q ID for existing m.q. of same key.

int
* msgsnd (int m.q.ID, const void *ptr, size_t nbytes, int flags)
IPC_NOWAIT

The moment process executes msgsnd() kernel, pointer pointing to msg sent to m.q given by m.q.ID, send n bytes.
int flags.

EAGAIN :- If m.q is full.
EIDRM :- If m.q is about to send msg but m.q. is removed
EINTR :- If m.q. is about to send msg but High priority interrupt occurs.

Size-t

* msgrcv (id, *ptr, size, type, flags)

msgrcv() receive a msg from m.q of given type and copy it to second argument pointed by ptr.

30/3/2022

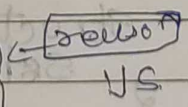
* msgctl (id, int cmd, struct msqid_ds, *void)

IPC_STAT
IPC_SET
IPC_RMID

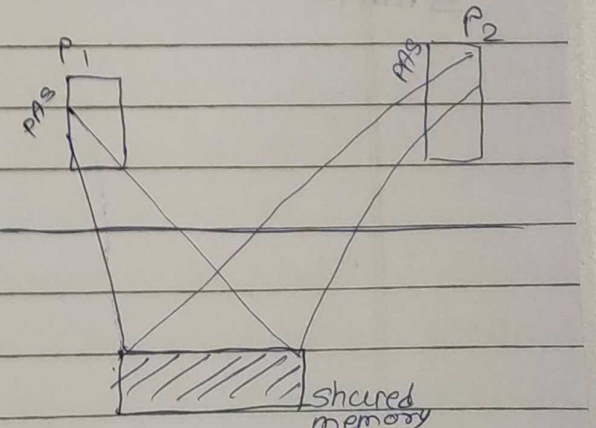
Shared Memory

→ One of the fastest technique of all IPC technique

→ No need to jump for process to K.S., to K.S., shared memory will go there



→ Shared memory, it allows two or more process to access a given region of shared memory.



→ Because need not to jump from one mem location to another memory location for data read and write operation b/w reader and writer process or a client - server process.

→ Shared memory can be used for b/w server and clients only there is synchronizing-the access of given shared memory region with semaphore (lock & unlock) technique

→ Each and every shared memory has a shmid_ds structure

```
#include <sys/types.h>
#include <sys/stat.h>
#include <ipc.h>
#include <shmid.h>
```

* shmget (key_t key, size_t nbytes, int flag);

shmid →

```
struct
shmid_ds
{
    IPC_PERM;
    size,
    pid last shm attached,
    pid last shm detached,
    time last shm attached,
    time last shm detached,
    change time,
};
```

text data, bss, heap, stack

PAS

* shmat (shmid, void address, int flags)

→ When a process executes shmat() call will take a shared memory region and map it to free PAS. (second argu. provided zero.) (recommended).

→ flag shm_RDONLY, if my operation is done you need to call detach operations.

* shmdt (void *addrs).

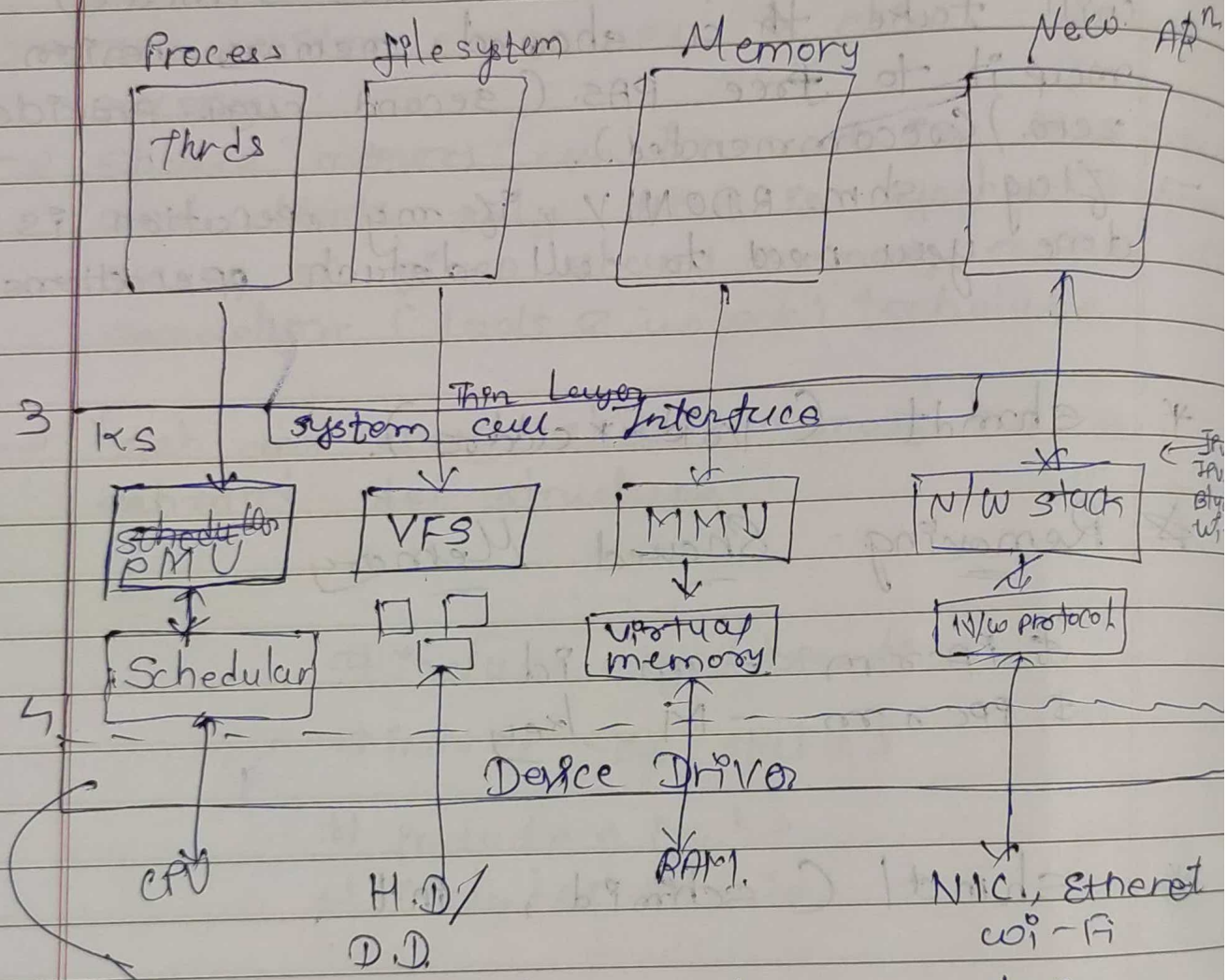
* Removing Shared Memory

$ ipcrm -m id
$ ipcrm -M key

* shmctl (shmid, int

3/3/2022

Linux kernel Architecture:-



→ Every part have its own device drivers

→ keybord, Hard disk, NIC, Printer.

* System Call :-

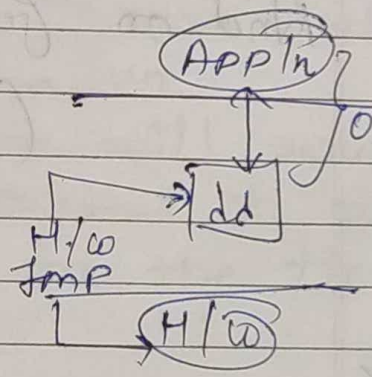
→ System calls are kernel space services not user space service. They are meant to invoke kernel services.

→ Each and every system call has to pass through system call interface and then invoke a appropriate kernel service.

* Device Drivers-

→ Driver from kernel perspectives-
Device Driver is a software that communicates with the hardware manages the HW and brings the functionality of the end device to the User.

- In Linux OS device drivers resides in the kernel space because only the kernel space driver has privilged to communicate with hardware one is
- driver has two interfaces, b/w Appln and driver which is OS specific, and one more interface driver & hardware, which is based on hardware spp.



* Semaphores-
(IPC comm. technique)

Semaphore(s) ⟶ array of semaphore
Complex

→ Semaphore function deals with array of semaphores, array of semaphores is a bit complex issue but in large app software process needs to work on lots of resources need more protection then having array of semaphore is big advantage.

ipcs -q → Message Queue
ipcs -m → Shared Memory
ipcs -s → Semaphore

→ semget(), semop(), semctl() are designed to work on arrays of semaphores.

★

(int) semget (key, int no of semaphores, int sem flag);

[header files
<sys/types.h>, <sys/sem.h>, <sys/ipc.h>]

→ On successful creation of semget(), create semaphore and returns sem_id. And on failure returns -1.

arg
(int sem flags)
|
IPC_CREAT

★

(int) semop (int sem_id, struct sembuf *ptr, size_t no. of semphores)

struct sembuf
{
sem_no 0
sem_op → -1 (waiting)
sem_flg → +1 (available)
}
SEM_UNDO

→ Used to change the value of semaphore

→ Array of structure used in (Semaphores) Technique of IPC comm.
→ Function Pointer we had used in threads.

* (int) semctl (int sem_id, int sem_no, int command, Union semun);

→ To control the semaphore operations.

int command
SETVAL

- used to initialize semaphore to a well known no.
- The value is required to pass value member of Union sem_un
- Only then the semaphore will setup.
- This process is performed, if we are using semaphore for the first time

SEM_UNDO flag:
→ kernel will read flag called SEM_UNDO and will know what changes have been done to the semaphore by the process.
→ If process terminated without releasing semaphore semo UNDO will allow to automatically release semaphore.

1/4/2022

* Signals :-
  Signals are software interrupts they notify process about an event occured
→ Signals are asynchronous in nature they can happen at any time
→ CPU kernel and any software that running on CPU can trigger signals to process.
→ A process with enough permissions can send a signal to other process
→ A process can send signal to itself.

* Terminal generated Signals :-

The signal generated by certain terminal keys,

Ex: ctrl + c
↳ Generating Interrupt signal
SIGINT → delivered by kernel to process

Ex: ctrl + z
↳ Generating Interrupt signal
↳ SIGTSTP → delivered by kernel to process → Process will be on wait queue.
↳ fg → SIGCONT → STDOUT → Continue with execution

→ Linux system have naming convention, they start with three chars. SIG. Each and every signal is defined by a number provided in a headerfile called <signal.h>.

→ Linux has got 2 classification of signals
1) Standard (Traditional) → Numbered from 1-31
2) Real time.

* Hardware Exception Signals :-

→ Invalid Memory Refrence } detected by hardware & notified by Kernel.
→ divide / 0

→ Kernel will send a signal to process
SIGSEGV
└ signal → Process terminate

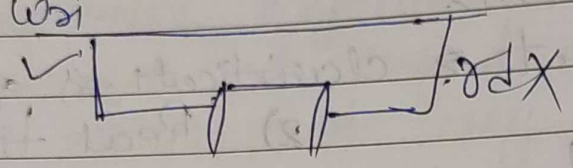
* kill() function :-

kill() will kill process - but you can kill process if you are (parent) owner of that process.

→ Interrupt tell CPU.
→ CPU generate signals → tell process.

* Software Signals:-

1) SIGURG1
2) SIGURG2
3) wsi



SIGPIPE

4) SIGALRM

* Pending signals:-
→ Signal between signal generation and signal delevering called pending signal
→ In response to the signal process can run some default action. Proces can ignore a signal or a process can run user defined function.
→ Instead of running default action on delivery of signal a user can program user defined fun. can and register coi th the kernel.
→ On delivery of signal kernel should invoke usd instead of default action.
→ This Proces called as signal handler

(void*) signal (int sig no., void(*ptr)(int))

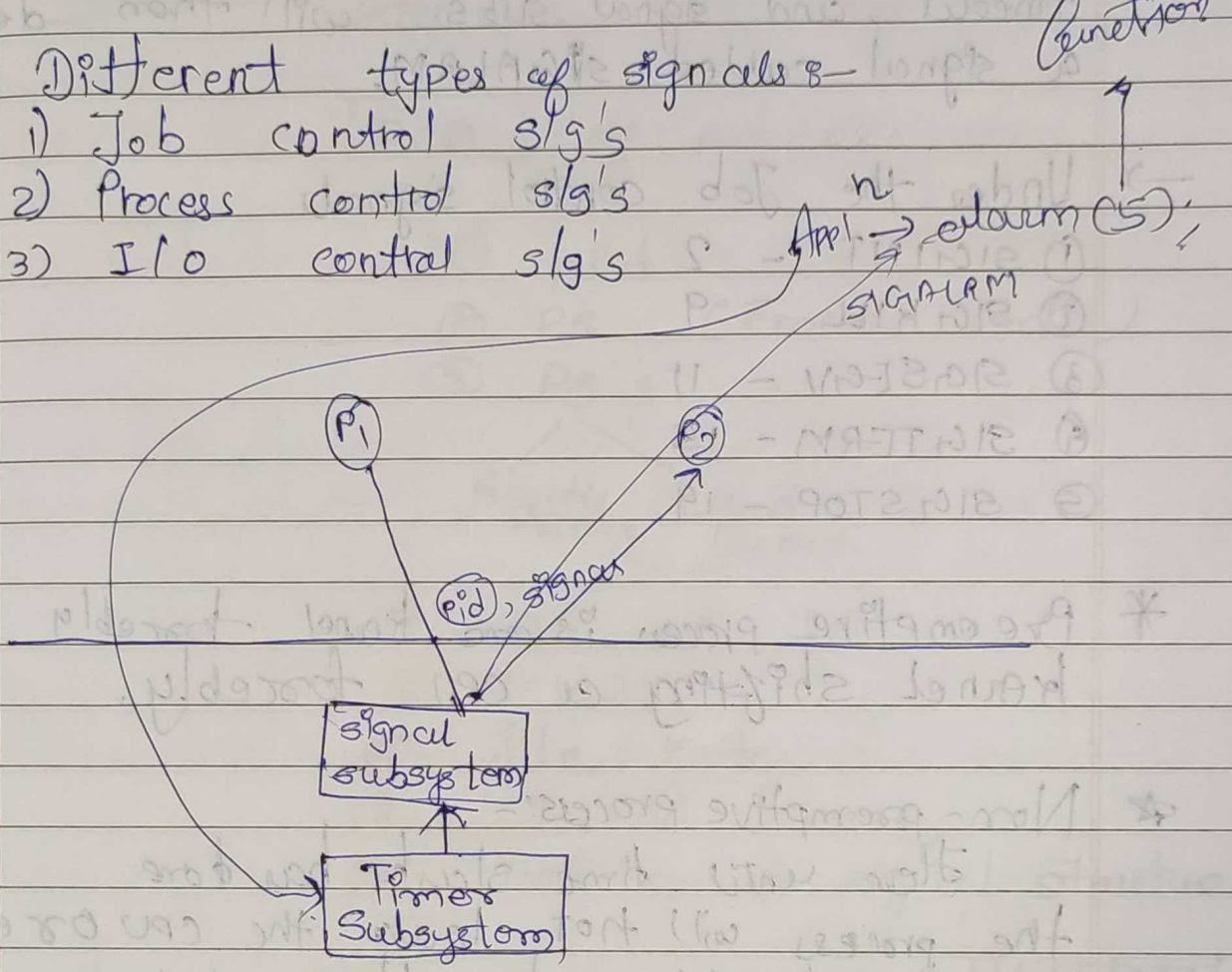
void(* signal (int sig. no ; void (*ptr) (int))) (int);

→ This is called installing handling and signal handler.

4/4/2022

* Signal handler :-

→ Different types of signals :-
1) Job control s/g's
2) Process contrd s/g's
3) I/O contral s/g's



Case:1

→ In the kernel there is a signal subsystem which is responsible of delivering a each process in a user space.
→ If appl wants to send a signal to app2, it has to register the signal subsystem and convey the appn id and signal no. to which appl wants to send a signal
→ Signal subsystem on behalf of appl delivered the signal to app2.

Case:2 appn or wants to perform periodic task fun, then register with the timer subsystem for delivery of a signal at a particular timeout. (Timer subsystem collects the time slices.)

→ Timer s.s invoke the signal s.s after some timeout, and signal subs. will then deliver a signal called SIGALRM.

→ Under the Job control signals
① SIGINT - 2
② SIGKILL - 9
③ SIGSEGV - 11
④ SIGTERM - 15
⑤ SIGSTOP - 19

* Preemptive process is one kernel forcebly kernel shifting a CPU forcebly.

* Non-preemptive process:-
Here until time slash has done the process will not leave the CPU or else process should go to block state

→ SIGKILL and SIGSTOP signal can not be handled or can't caught.
→ Except you can handle any signal (proce
→ USR1, US2.

*→ SIGUSR1, SIGUSER2.
*→ I/O signals
→ The signals are delivered to the ready to perform input output operation.

Too [Two types]
1) Real time - 3/
2) traditional - 3/

* SIG_IGN :-
Interrupts will ignored

PCB of process
1. pid
2. PS (Process state)
3. PQ's
   - Ready queue
   - wait queue
4. PC, SP, IP
5. MMU
6. file info.
7. Thread library
8. pointer to signal structure

→ struct signal_struct *signal
struct sighand signal_struct *signal

sigset_s block

struct sigpending

→ In
* Execution of default handler

→ In signal subsystem there is a fun called as send_signal(), send this fun. is executed before delivering of a signal.

→ steps followed by step-signal:

1) send-signal(); will get reference of PCB coher signal can be delivered

2) will get point to signal structure, ref by PCB

3) There is a signal pointer + signal structure coher is ponting to a vector of 64 elements.

4) send-signal(num) will manipulate signal structure.

5) for signal hndlr; Second element of the vector.

6) 0 disable
1 enable

7) Send will receive CPU exception after setting CPU stop current task and CPU create exception handler.

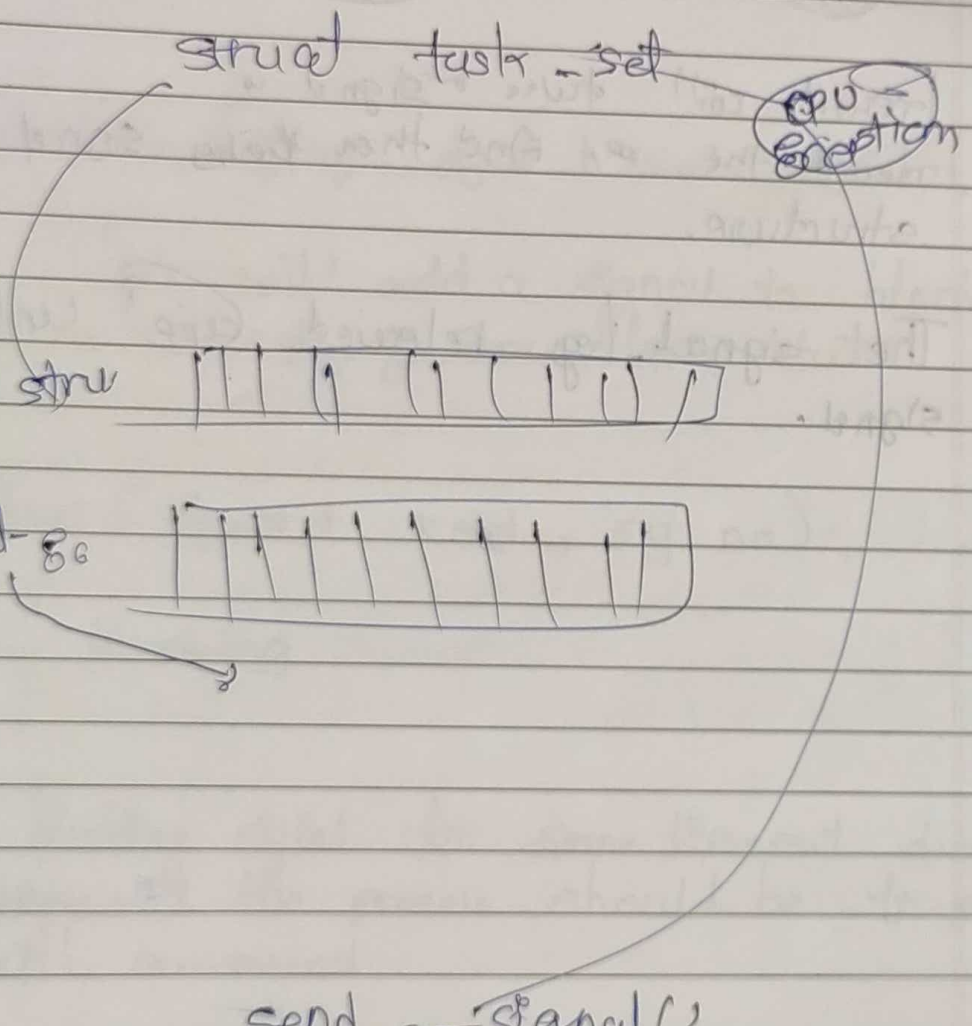
8) Structure is a signal structure

9) Exception number is more than dest. PCB size

10) Pointer of signal handler structure pointing to structure of integers.

11) Exception handler table for the signal structure and load for the enabled domain and maps to respective function in handler.

12)



→ When default context switching is not possible.

* sigemptyset (sigset_t *set);
  ↳ It will initialize the sigset and clear it

* sigfillset (sigset_t, *set);
  ↳ It will set the all sigsets.

* sigprocmask (flag, sigset_t *set, sigset_t *oset
  old
  ↳ We will add a signal to block & remove a signal from block state.

* sigaddset (sigset_t *set, sig. no);

* After blocking

* Appn blocking sigint for some timeout. during the timeout the process should be terminated by kill command

5/4/2022

* Error
A mistake in programming code is error.

* Defects
If the error ditected by tester called as defect.

* Bugs
If defect is accepted by respective developer then it is called bug

* Debug
Process of fixing the bug called debug

* Failure: When system software fails to perform particular run execution leads to failure.

* Fault
A fault is a condition because of which system ~~get~~ software fails to perform an action

* Static code analyzer
Static code analysis is the process of identifying programming errors, bugs on the source code before program is being done.
→ It is done on set of instruction, by using some

coding standards. These kind of a analysis will help to identify the loop hole and witness weakness in source code that might be harmful

→ Analysis is done on stationary piece of software. Thiscohy is knowing static code analysis.

sudo apt install splint

→ Splint is a static code analyzer which used to identify suspicious code, programming code and stylistic errors.

→ splint is

* Programming errors are showed by Compiler and identified by compiler at compile time. Eg: Undeclared variables, syntax, undeclared functions, Typechecking, lib. incompatibility

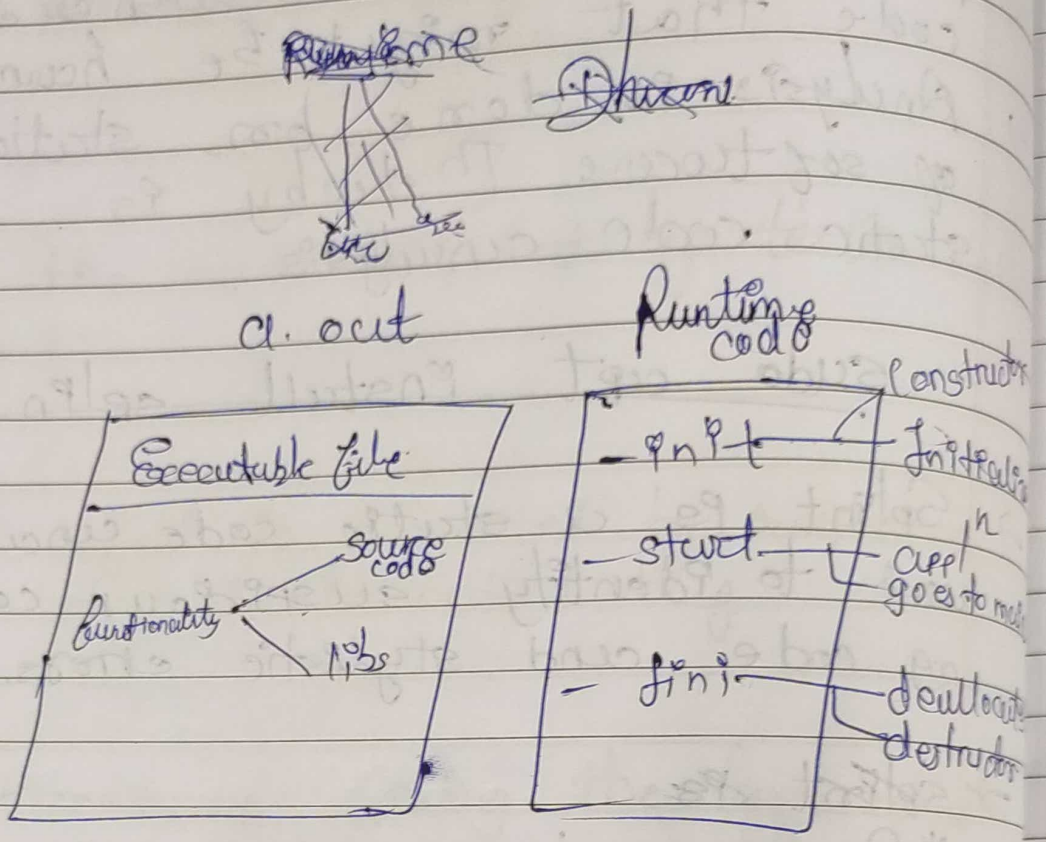
* Stylistic errors : If code is deviating from prescribed coding standard.

gcc -E one.c -o one.i

gcc -S one.i -o one.s.

gcc -C one.s -o one.o

→ Linker job is to provide runtime code to



→ Job of linker is to add runtime code to build a executable

→ Runtime is not a lib, it's a routines added by linker during program build time

→ Runtime talk about three diff^n macros _START, _INIT, _FINI

→ The moment crt^n start execute, starts with __INIT MACRO. also called initializer

→ Initializer will allocate key resources to a object file and also configure the object file (that is providing addres to objectfile load and execute.

→ Once addresses are configured, cntl goes to _start (MACRO).. start (MACRO) is preemption to main fun. and cntl jumps to main fun. execution

→ Here the appl^n functionality executes. When main terminates functionality also terminated then function goes to start macro and jumps to _FINI (MACRO)

→ Now _FINI (MACRO) will deallocated all the resources provided by JNIT (MACRO).

* Clang:

clang -stdlib=libc file.c

objdump --full-contents -section=.comment a.out

sudo update-alternatives --config cc

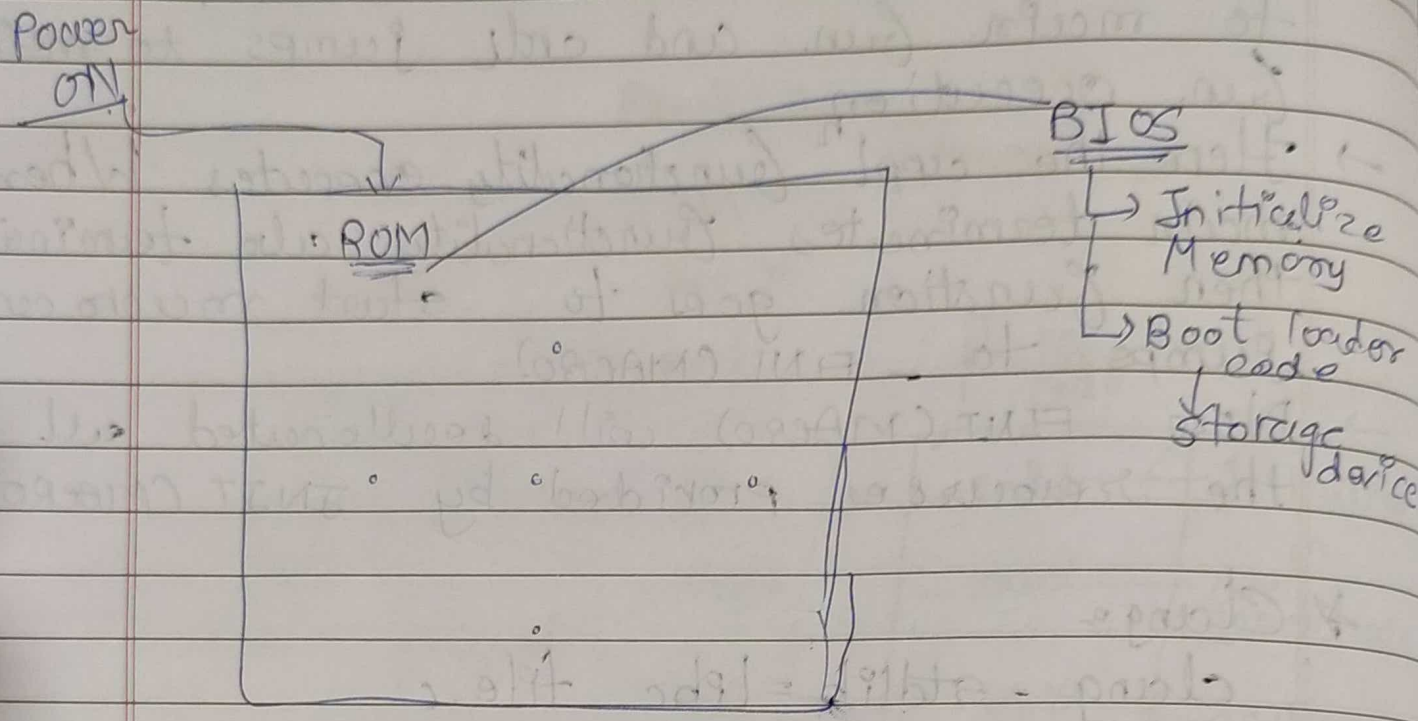
→ clang is a compiler using c and c++ and built using it. and released under apache 2.0 license.

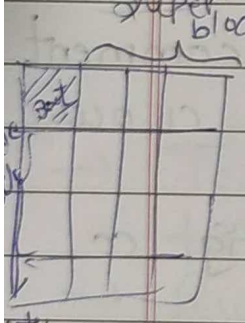
→ clang is faster and take less memory when compared to gcc.
Ex google chrome browser on linux, window now built using clang?

→ clang supports very few development environments.

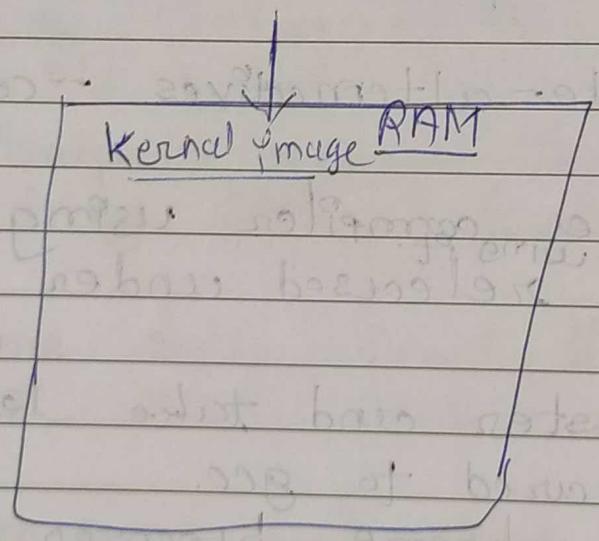
6/4/2022

Linux Boot Sequence





* In Linux there is only one code
GRUB = Grand Unified Bootloader code
↳ will read root file system and map os image and load



① CPU
Real mode
protected mode

→ While executing BIOS mode, CPU will in Real mode and looking in CPU in series.

→ Vendor of the motherboard is providing address of BIOS mode, and Boot loader code

→ Kernel Image divided into two parts;
1) Boot strap code → first executes and shift CPU from real to protected mode process called as processor initialization
2) Vmlinez

→ Kernel is first piece of software, loaded into RAM, executes until shutdown initiated.

→ Kernel roles :-
1) Setup memory
2) Setup Interrupts D.S
3) Setup device d.s
4) Setup Process tables
5) Setup file Management
6) Setup CPU Scheduler
7) Initialize kernel threads

→ There are 3 boot pins it has possibility $2^3 = 8$.

→ If 5 boot pins $2^5 = 32$ to different booting sequence.

* GCOV tool :-
→ dynamic tool analyze the source code. GCOV - GCC coverage tool.
→ Opensource tool.
→ gcov works with only gcc.
→ It will check for untested part of your source code Identifies unexecuted instruction.
→ Can also be used as Profiling tool for browsing and navigating the source code

→ allows you to modifies and enhance the source code

→ Compiling source code with two flags
1) -fprofile-arcs
2) -ftest-coverage

→ -ftest-coverage will do analysis the source code and generates a hidden code in binary. and also records how many times code executed.

→ -fprofile-arcs creating a profiling output file. When we are using -fprofile-arcs with source code generating profiling output file with .gcda and .gcno files.