## Software Systems

The course has two major modules:

- System Software
- Enterprise Software Development

MODULE 1: SYSTEM SOFTWARE

This module starts with the introduction of the computer architecture, operating system and kernel architecture. Different types of kernel design namely monolithic, micro and hybrid architecture are analyzed. File, process, signals and memory management has been explained with the suitable live examples. Signals, inter process communication and synchronization mechanisms are explained in a practical point of view. The implementation of soft real time
systems according to POSIX standard are analyzed. Finally the difference between application program and kernel module are discussed. During Lab, the students will be asked to write code from scratch for more than 30 real time/live exercise. This comprehensive hands-on course provides the knowledge and skills of system programming and most of the concept such as File, Process, Signals and IPC are compatible with the UNIX variants like UNIX, Linux, Solaris, HP-UX and AIX.

MODULE 2: ENTERPRISE SOFTWARE DEVELOPMENT

In this module, student is exposed to the elements of enterprise software development with primary focus on web application development and mobile application development.

**Course Content**

**MODULE 1: SYSTEMS SOFTWARE**

**Topic 1: Computer Architecture**

- Basic structure of computer hardware and software
- Process, Memory and I/O systems: CPU, RAM, Virtual Memory, I/O devices
- Types of System - Server, Desktop, Embedded and Real Time
- Operating System Vs Kernel

**Topic 2: Kernel Architecture**

- Kernel Subsystems (computing resource management)
- Types of Kernel: Monolithic, Micro and Hybrid Architecture
- Monolithic - Server and Desktop
- Microkernel - Embedded and Real Time systems
- Hybrid - Handle both RT and Non-RT tasks

**Topic 3: System Internals**

- Brief implementation of - process, file, memory and signal management
- Communication Mechanisms - pipe, FIFO, message Q, shared memory

**Topic 4: Synchronization Mechanisms - File and process**

- Implementation of Soft Real Time Systems - as per POSIX standard
- Application Program Vs Kernel Module

**MODULE 2: Enterprise Software Development**

**Topic 1: Fundamentals of Object-oriented Analysis and Design**

- OO concepts
- Unified Modeling Language (UML)

**Topic 2: Software Architectures**

- Understanding large scale systems – MVC, Web architecture, REST, mobile architecture, and hybrid systems along with relevant concepts on reference and enterprise architecture.
- Understanding quality attributes
- Introduction to design and architectural patterns

**Topic 3: Web application development**

- MVC for Web - Twitter Bootstrap (rendering view), jQuery, Ajax (from jQuery) and servlets (controller), REST service, back-end model - MySql, Java programming and concepts of key value pair (like mongo DB – implemented using MySql)

**Topic 4: Mobile application development**

Camlin Page
Date / /

सरस्वति नमस्तुभ्यम् वरदे कामरूपिणी।
विद्यारम्भं करिष्यामि सिद्धिर्भवतु मे सदा।

- Connectivity, security, online/offline modes, integration of sensors, location services, responsiveness.
- AngularJS and related frameworks

**Text Book / References**


1. Operating systems - Internals and Design Principles by William Stallings, Prentice Hall. http://dinus.ac.id/repository/docs/ajar/Operating_System.pdf
2. Architect Korner: Challenges in Platform Selection by Gururajan and Thangaraju, Linux for You, June 2010, pp.51-56
3. Software Architecture in Practice by Bass and Clements, Addison Wesley.
4. Ajax - https://www.youtube.com/watch?v=f46WEeM8HTA
5. REST Services - https://www.youtube.com/watch?v=xkKcdK1u95s
6. Jquery Tutorial - https://www.youtube.com/watch?v=8mwKq7_JlS8

# PPT-1

Camlin Page 013
Date 03/08/2018

## SOFTWARE SYSTEMS

(lecture 3)

### GPOS vs EMBEDDED SYSTEMS

**General Purpose Operating Systems**

- built to be customizable in software.
- eg:- desktop PCs, servers

**Embedded/Real Time Operating Systems**

- it is a part of a bigger system (generally).

EMB/RT

- Microcontrollers
  i.e. specific task systems like refrigerators, μWaves etc.
- Hard Real Time OS
- Soft Real Time OS

all RTOS are Embedded Systems (EMB), while all Embedded systems, while all EMB are not RTOS.

Real Time Operating Systems are designed to meet some deadline.

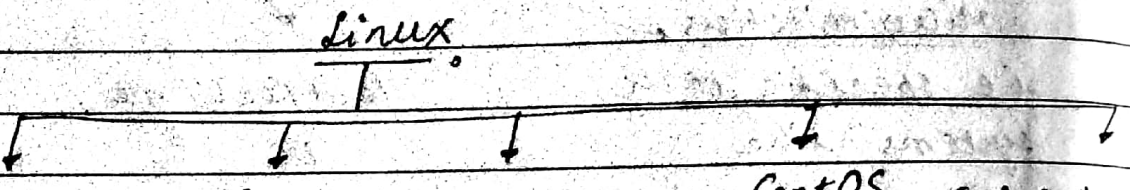
**Soft Realtime OS**
If deadline is not met, throughput is effected.

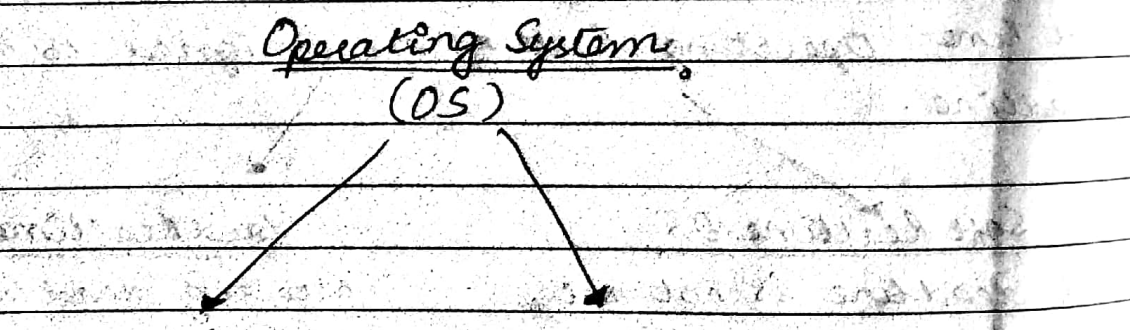
**Hard Realtime OS**
Meeting deadlines becomes necessary, as Human lives are at risk or some disaster will occur.

GPOS
- Windows
- Unix
  - Solaris → outdated
  - Linux
  - HP-Ux
  - AIX → used by IBM m/cs

- Kernel architecture, system call interface and basic commands are almost same in mostly all of the above UNIX based Operating Systems.
  They all have "MONOLITHIC ARCHITECTURE".

- Linux :-
  Linux is a family of free and open-source software operating systems built around the Linux kernel.




Utilities
↓
utilities makes an OS user friendly.

- for all Linux distribution, Kernel is same, but utilities vary based on the target customers of the distributor.

1. WRLinux (Wind River Linux) } Support EMB and RT
2. MK Linux (Monta Vista Linux) } Linux OS.
3. RHEL → Support enterprise applications.
4. CentOS → for Sale
5. Fedora → for server, desktop
6. Ubuntu → for development Systems.

- Course Agenda :-

1. COMPUTER ARCHITECTURE :-
   - ↳ Basic Structure of Computer software & hardware
   - ↳ Process, Memory and I/O systems : CPU, RAM, Virtual memory, I/O devices.
   - ↳ Types of systems - Server, Desktop, Embedded and Real Time.
   - ↳ Operating System Vs Kernel

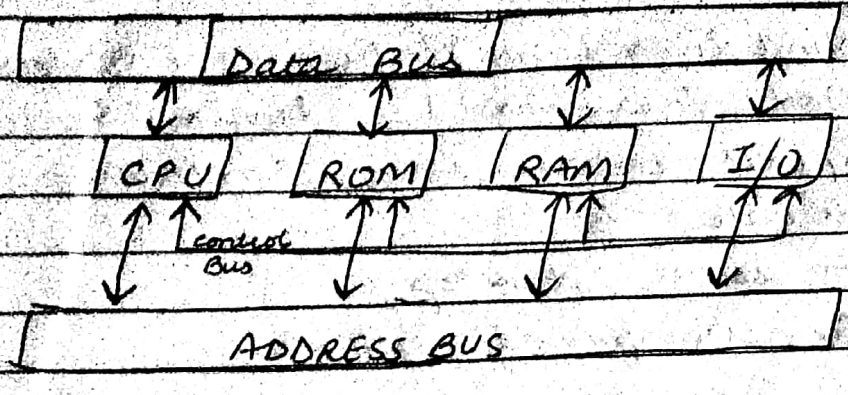
2. KERNEL ARCHITECTURE :-
   - ↳ Kernel Subsystems (Computing and resource mgmt)
   - ↳ Types of Kernel : Monolithic, Micro and Hybrid

3. SYSTEM INERNALS :-
   - ↳ Process, file, memory and signal management.
   - ↳ Communication mechanisms - pipe, fifo, message Q, shared memory.

4. SYNCHRONIZATION MECHANISMS - FILE & PROCESS :-
   - ↳ Implementation of RTOS - as per POSIX standard
   - ↳ Application module Vs Kernel module.

5. COMPUTER ARCHITECTURE :-

→




↳ ROM :- It contains Boot loader (BIOS)
↳ GRUB (Grand Unified Boot Loader)
↳ LILO (outdated. was used in Linux)
↳ U-Boot (Universal Boot loader)
used in embedded systems

has a pointer to where zipped Kernel image is stored.

Load Kernel to RAM

Proceed further.

↳ RAM is a volatile memory.

↳ PROCESS CONTROL BLOCK :-
Contains register values and other informations needed to be accessed/stored due to context switches

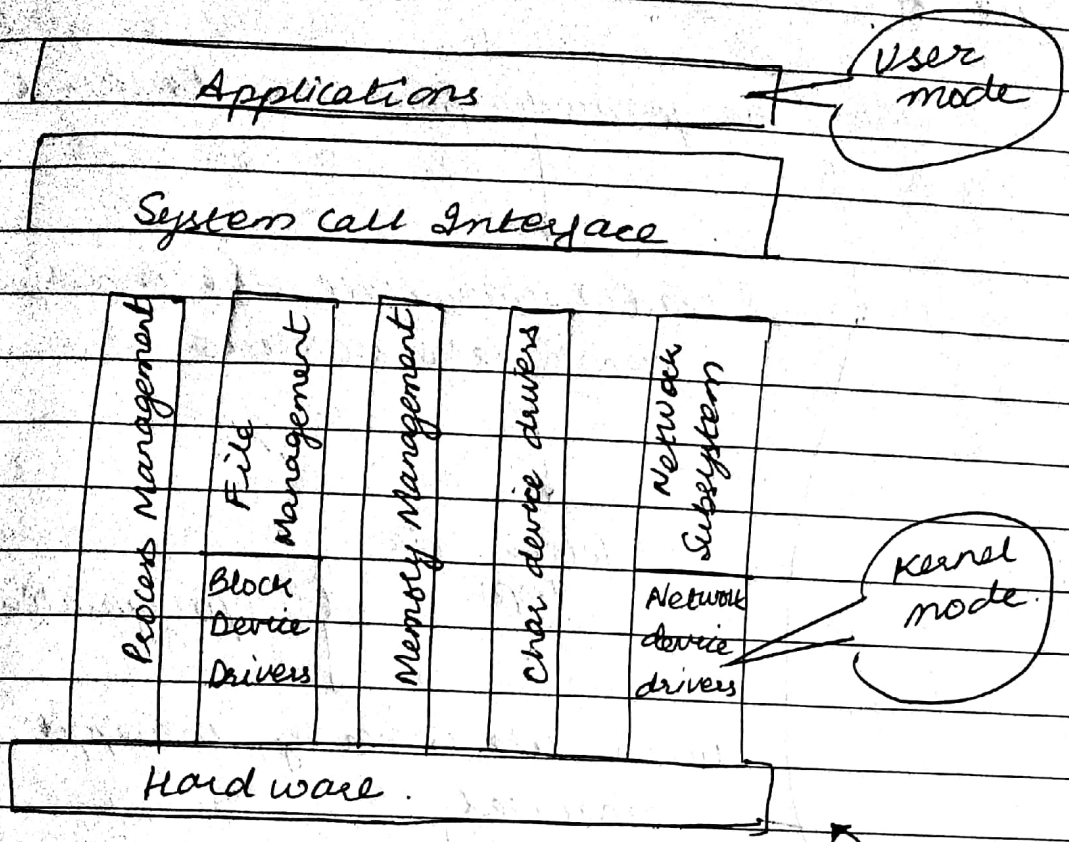
Processor affinity :- pinning process to a particular processor

RAM :- has buffer, cache etc

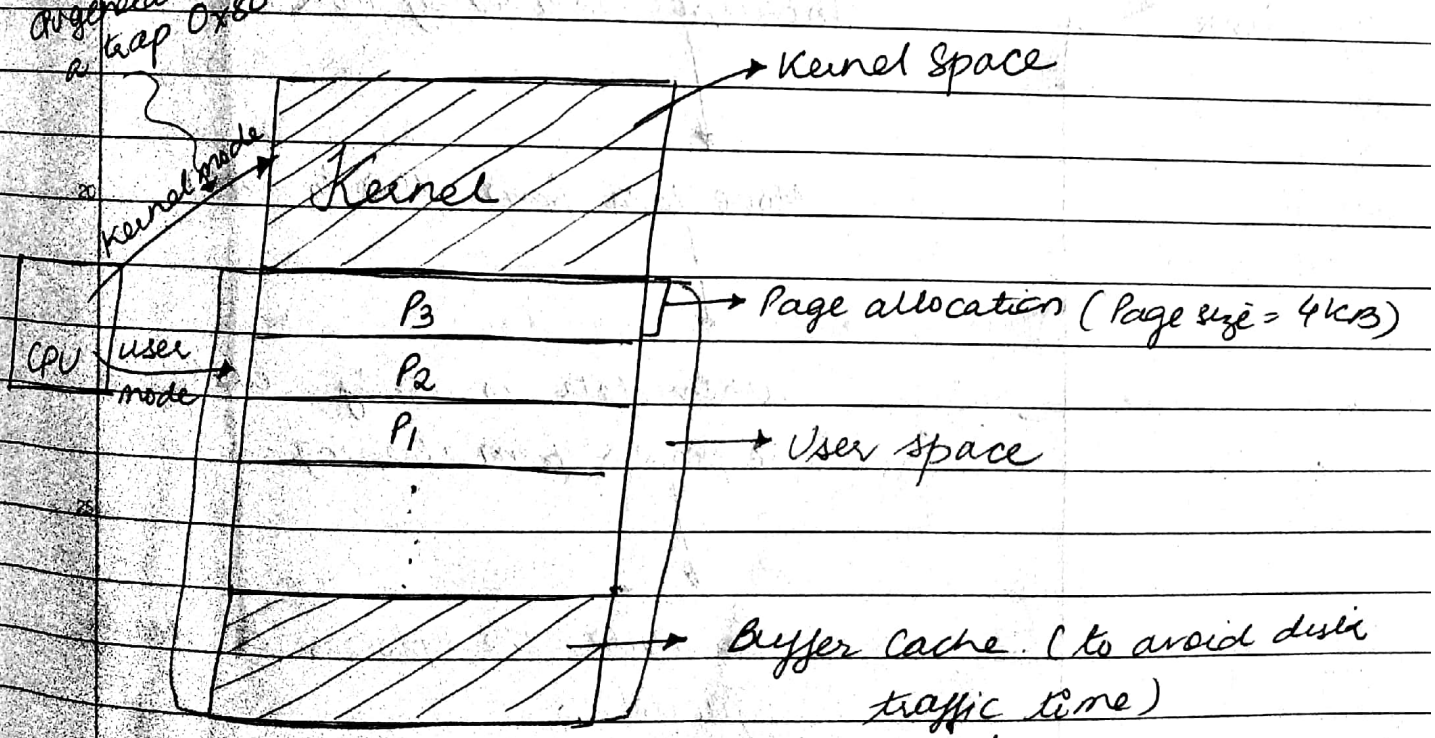
CPU :- process management, processing, computations etc

* formatting a disk :- creating a file system on the device

- General Purpose OS :-




Block diagram of a GPOS.

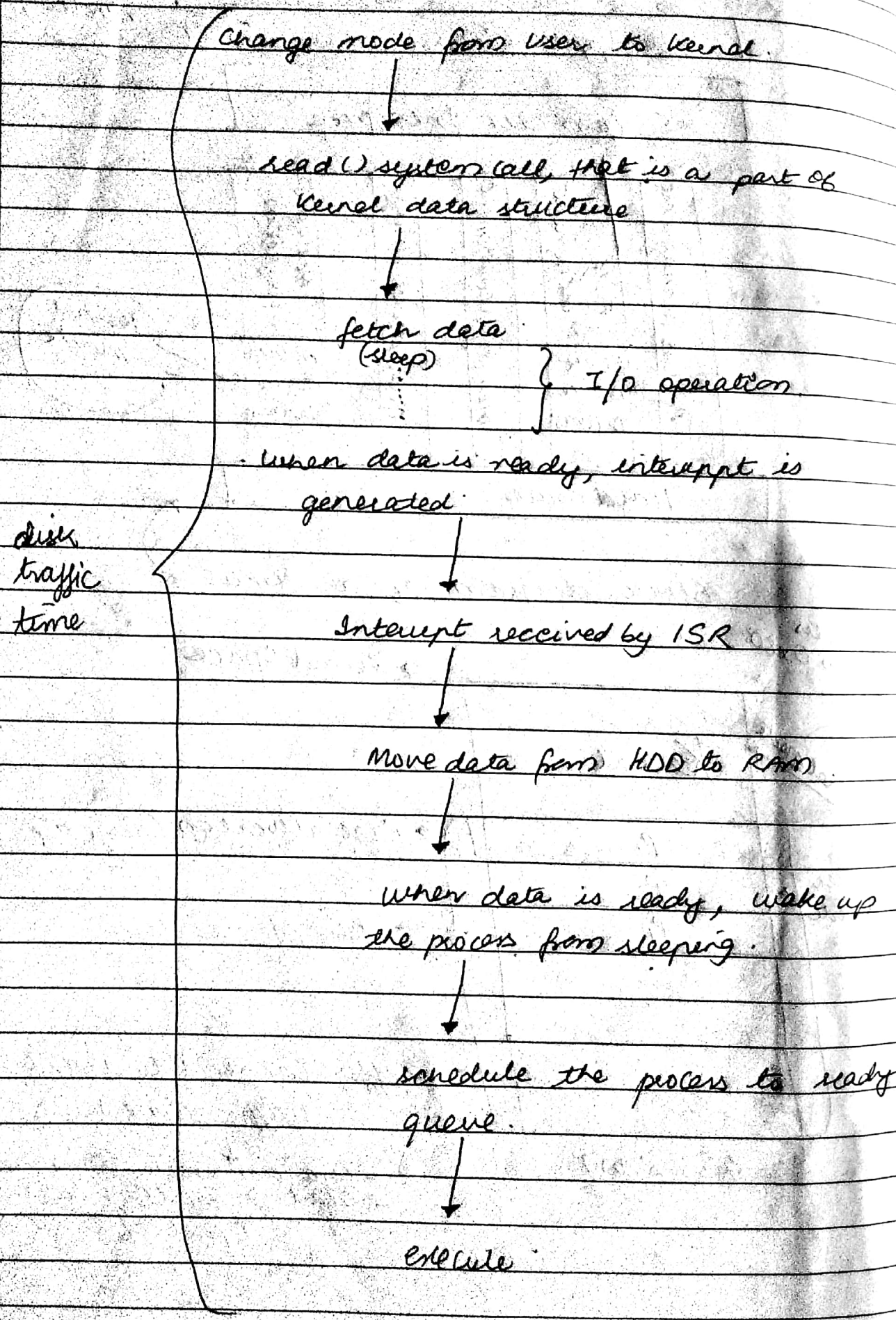



- changes mode from user to Kernel
- read() system call
(part of Kernel Data Structure)

↓

fetch data.

- Disk traffic time :-



- disadvantage of buffer cache:-
  - ↳ during huge project execution, if power fails in between, all changes are lost as, a file is updated only when it is closed.
  - ↳ system call to explicitly save data in HDD, even during execution is sync().

- Monolithic Kernel:- all Kernel utilities are clubbed together, so it is called monolithic Kernel.
  eg:- to see Kernel version/Kernel itself in Linux,
  cd /boot/
  ls -hl vmlinuz*
  eg: In my PC, there was a 7.6MB executable file, it contains all utilities, so, it is monolithic.
  Z here is meant for Zipped format.

- To see the Kernel version you use:-
  uname -r

- Microkernel architecture:-
  - ↳ it includes small #% functionalities as compared to monolithic kernel;
    - interrupt handler, scheduler, FIFO etc.
    - i.e. minimum possible functionalities for smooth execution; its size is in KBs.

↳ It is generally used for Embedded System as it is a dedicated system to a specific task.
↳ due to its very small size, only a limited number of functionalities can be implemented.

↳ We can load it as a user module / in user mode kernel in flash drive → bootup → drive contains all utilities. (also for RTOS)

* eCOS is embedded and μ-controller based.

- Monolithic Vs μ Kernel :-

↳ less performance load.
↳ no space constraint
↳ no real time behaviour

↳ more performance load
↳ space constraints

* htop shows background processes without work load.

↳ Thread creation :-

pthread_create (thread_id, attributes, function, farg)

→ POSIX_THREADS
thread is a function in execution.
→ Thread creation inside a process.

- HYBRID KERNEL APPROACH:-
↳ They have RTAI (Real time application interface) for both RT and non-RT processes.
↳ Ideally,

95% of the processes in execution are non-Realtime.

5% of the processes are Realtime (Hard RT)

↳ To update our Application, we can download the patch file and apply it.

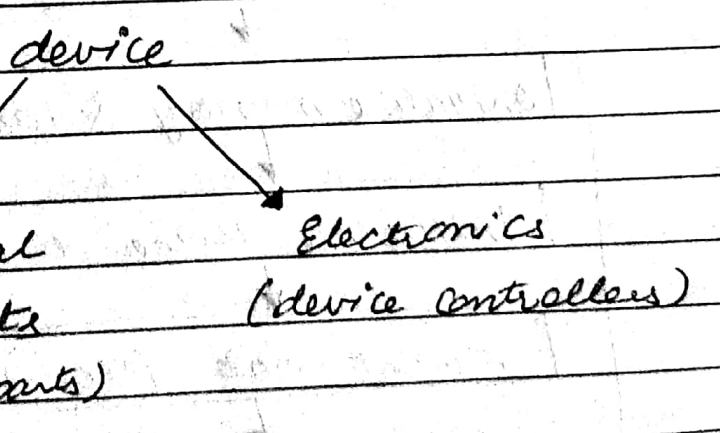
↳ contains only the differences between two updates.

eg: If changes done = 10MB, but update is of 1.2GB, then downloading and applying 10MB is efficient.

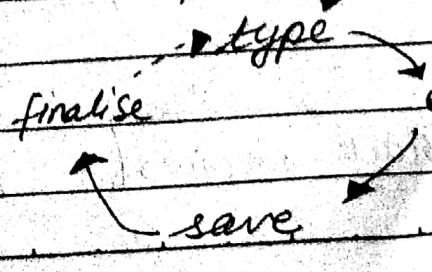
↳ rt_sometask → real time processes will be created directly to μkernel, with highest priority.

↳ WR Linux is optimal Linux Support (proprietary).

↳ Linux kernel source code can be downloaded from "www.kernel.org".



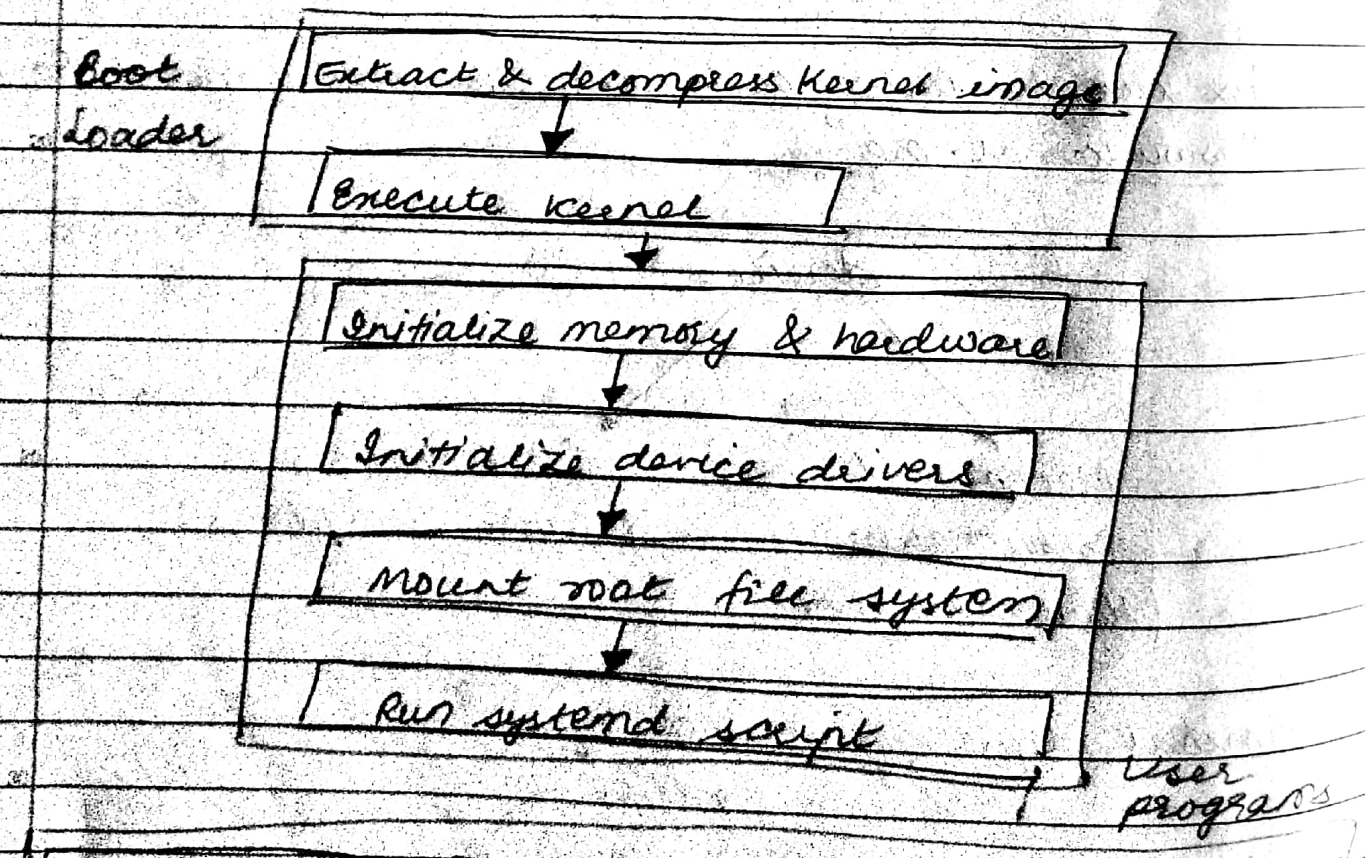

↳ terminal type interface:-



- KERNEL :-

| Architecture-dependent code | Architecture independent code |
|---|---|
| ↳ Low level system startup functions | ↳ System call handling |
| ↳ Trap & fault handling | ↳ File system |
| ↳ Low level manipulation of runtime context of process. | ↳ Terminal handling support |
| ↳ Configuration & initialization of hardware devices. | ↳ IPC facilities. |
| ↳ Runtime support for I/O devices. | ↳ Network communication support. |

- BOOTING PROCEDURE

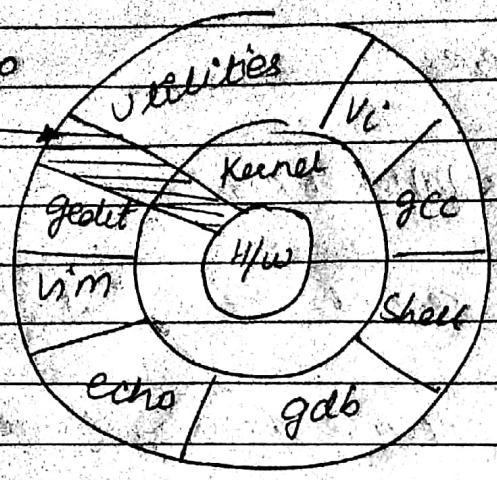


Power on
↓
Bootloader / MBR
↓
Use RAM Disk Image (RDI)
↓
decompress kernel
↓
RDI will execute during booting. → it activates essential utilities like keyboard, mouse, monitor etc.
↓
You can see messages on the screen.
↓
Mount file systems (/etc/fstab)
↓
choose run level (/etc/inittab)
↓
spawn systemd script → d - here is daemon process. eg: backup of data @ friday night 12:00AM.
↓
shell / login prompt
↓
RDI is removed.

↳ cron jobs:- cron allows users to run commands or scripts at a given date & time. It is usually used for sysadmins jobs. All cron jobs are daemon process.

↳ Conditions for a job to be a daemon process:-
1. Orphan process (not associated with shell/parent)
2. Session Leader (generally shell is the leader, once created, it should be session leader.
3. Privelages.
4.

- LINUX KERNEL ARCHITECTURE :-




OS = kernel + OS specific
Kernel - core part of OS

Processor architecture
↓
Assembly language code.

- My kernel :-

permissions only for root

ls -l vmlinuz (in /boot)
sizi vmlinu* .

- Some basic commands :-

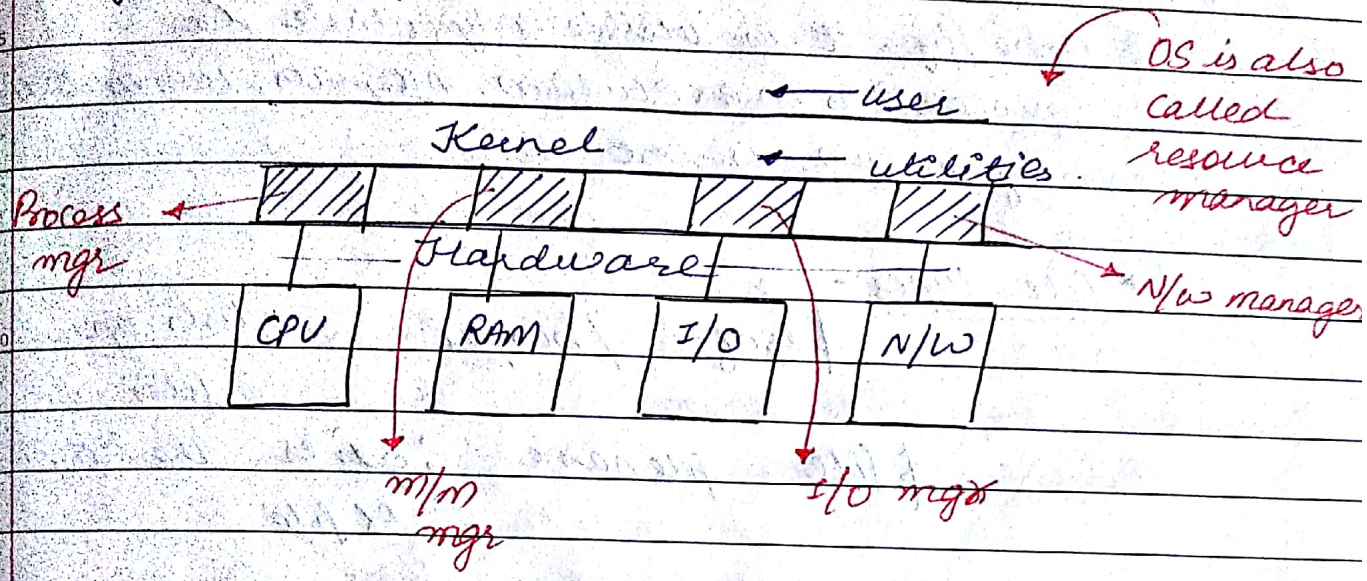
w :- terminal type info.
ptty :- pseudo terminal type.
tty :- terminal type.

→ drivers / kernel modules
↓
schedule mechanism as kernel modules? (X)

OR.

file system as kernel module?
(✓)

Kernel modules should be executable files, but should be object files as there are dynamically loaded modules.
eg: $ lsmod ↵

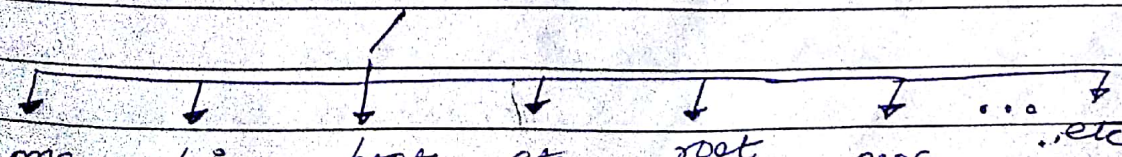



monolithic kernel {
Process manager;
Memory manager;
I/O manager;
Network manager;
} subsystems

If we have no m/m driver (m/m manager) then our memory directory should be a kernel module (m/m subsystem).

* Embedded systems have no concept of virtual memory.

- Inverted tree structure :- (easy to maintain)



Camlin Page 026
Date 03/08/18

↳ /proc contains → process info
→ system info
→ system limitations

this information is on the fly. It is visible only while the process is in execution. After it's termination information is lost.

↳ file types:-

$lsof /bin/vi : lists open files.

$file filename : gives the type of file

# LECTURE-4

- Recap :-

- all basic commands are in "/bin" directory
- command → can be used directly
- system call → write a system level program.
- library fn → nothing but wrap around system call.

- System calls can be monitored through "strace".

eg: strace -c mkfifo myfifo.

strace → traces system calls done for execution of this command.

-c → gives the count

- read() and write() system calls use file descriptors.

so, if I do "int fd = open ("some file");

The first file opened would usually have its value = 3.

because, 0 ← STDIN
1 ← STDOUT
2 ← STDERR

- whenever any process is created, default 0, 1, 2 descriptors are assigned.

This can be checked at: /proc/<process id>/fd

/proc contains → CPU info. → /proc/cpuinfo
→ m/m info → /proc/meminfo

H/W. • Interview Question :-
I have a huge log of msgs (say $10^6$ lines).
Some error came & my system crashed / user login failed etc.
I want to get 10 lines above & 10 lines below that position of log, to judge what had actually happened.
How will I do this?

Sol.
Use grep -c to find crash's log
then use head & tail to get o/p

- jobs :- this command shows the background processes.
eg:- ls -l /proc/<pid>/status.

- df -h :- gives disk fragmentation.
(sda ← hard disk partition)

- System Call Interface :-
CPU
→ System call Interface → access to Kernel data structure
↳ Usually system calls are executed in Kernel mode
↳ User ← ~~special~~ → Kernel
fifo

eg- signaling mechanism is a part of System call interface.
we can see signal.c file in Kernal

- Mount means how to get that fs from.
it can be checked by $df -h
↳ mounted on

OR

$mount

- Embedded/RTOS :- (Kernel + Root file System).
Keep in flash memory
↳ NAND flash
↳ NOR flash → BIOS
↳ application execute here only
↳ Kernel + root fs
↓
firmware
(embedded s/w)

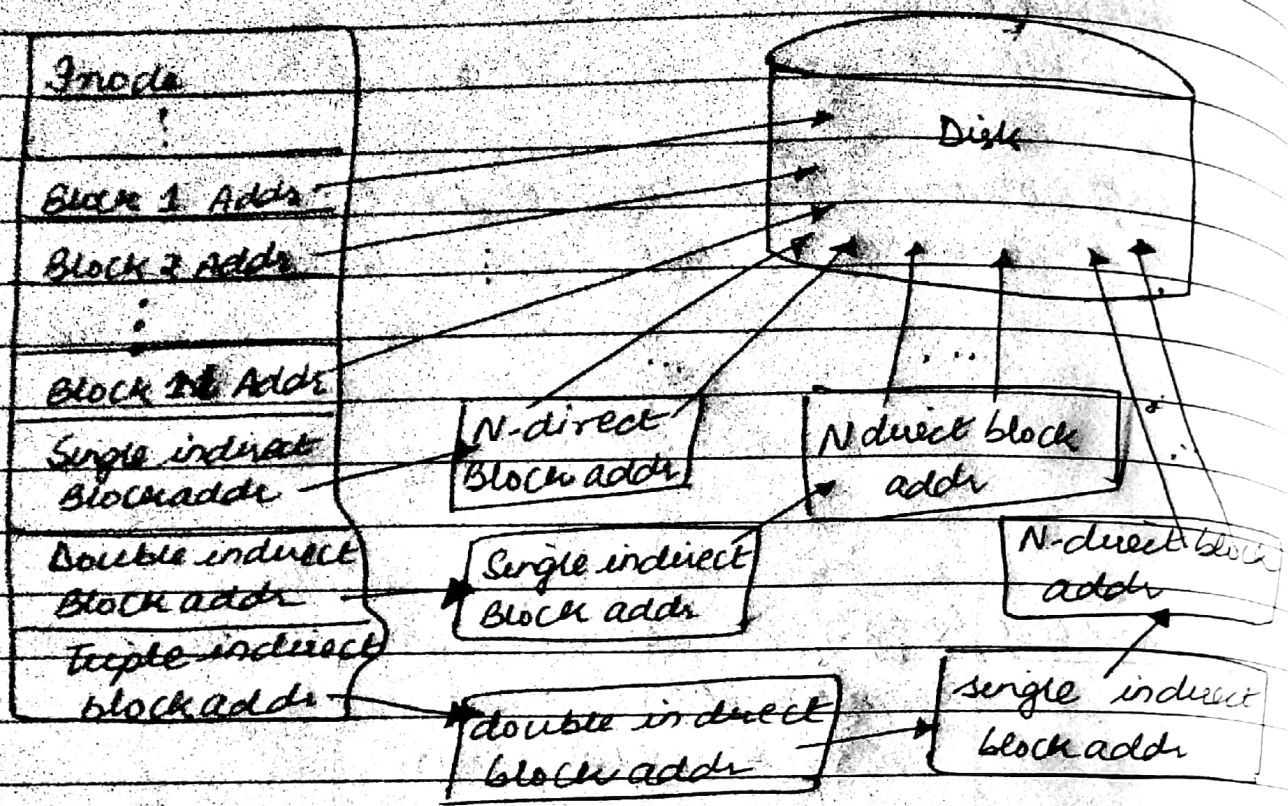
application → performance monitoring
↳ GUI

- File System :- Contains some data & information
↳ Kernel uses inode # to locate a file in FS.
↳ $mount also shows file system used.

- default FS is different for different Linux distributions → RedHat → XFS
↳ Ubuntu → ext4

- Virtual FS is essential, as we can use same command for all FS. eg ls., otherwise, we should have api's for all different FS

- ext File System.




- Each inode entry can track very large file
- address block = 4Bytes
- size of 1 Block = 4KBytes → to avoid fragmentation (easy to allocate)
- single indirect pointer contains 4KB of block addresses i.e. 1024 addresses.
- So, for block size = 4KB and N=12, a single inode entry can track file that has max size of:-

$$12*4KB + (1024*4KB) + (1024^2*4KB) + (1024^3*4KB)$$

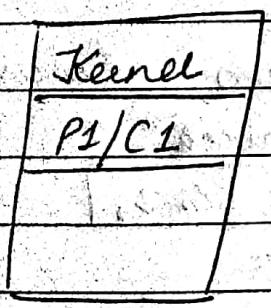
$$\simeq 16TB.$$

- Excercise :- Type following commands and see usage of block size.

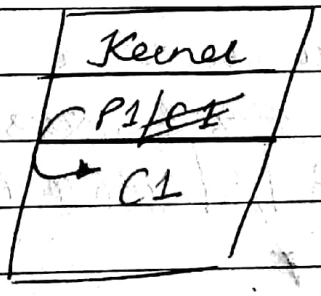
```
touch file
ls -l file
echo > a file
ls -l file
du -h
```

- to see linux supported file systems, type
  $man fs.

- Copy on write (COW) :-
  If you create exact image of a process (child) and only perform read, then no need to write child; otherwise only write.




Copy on write
(C1 read only)




Copy on write
(C1 writes)

# LECTURE-5 :-

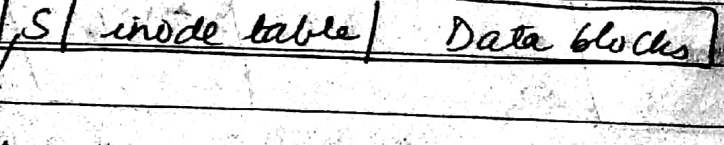
↳ The International Electrotechnical Commission is a non-profit Non-Governmental international standards organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

* File System Comparison :-

| Feature | EXT4 | XFS | BTRFS |
|---|---|---|---|
| Architecture | Hashed B-Tree | B+ Tree | Extent based |
| Max Vol. Size | 1EBytes | 8EBytes | 16 EBytes |
| max file size | 16TB | 8EB | 16 EB |
| max # of file | $4 \times 10^9 / 2^{32}$ | $2^{64}$ | $2^{64}$ |
| max file name | 255 Bytes | 255 Bytes | 255 Bytes |

- XFS :-

When you create a file system, Linux creates a # of blocks on that device




↳ Linux creates the entry for "/" directory in the inode table and allocates data block to store the contents of "/"

↳ Super block contains information like :-

(i) a bitmap of blocks on the device, each bit specifies whether a block is free or in use?

(ii) size of data block

(iii) The count of entries in the Inode table.
(iv) date & time when the file system was last checked.

- each device also contains more than one copy of the super block
- Linux maintains multiple copies of super-block as it contains information that must be available to use the device.
- If the original super block is corrupted, an alternate superblock is used to mount the file system.

- The inode table contains an entry for each file stored in the file system. The total # of inodes in a FS determines the # of files it can contain.
- when a FS is created, the Inode for the root directory of the FS is automatically created.
- each inode entry describes one file.
- Each Inode contains:-
  (i) Owner's GID, UID
  (ii) file type and access permissions
  (iii) date & time, the file was created, last accessed, last modified
  (iv) size of file
  (v) # of hardlinks to the file

Ordinary file creation:
- The kernel allocates space in the HDD. The text in the file is stored one character per byte of memory.

↳ A file holds these characters and nothing more. Nor does it contain any information about its beginning & ending.
↳ An inode entry is created on a section of the hard disk.

- Device Special file :- describes the following characteristics of a file
  ↳ device name
  ↳ device type (block/character)
  ↳ major device number. (eg: '2'-floppy, '3'-HDD)
  ↳ minor device number. (eg: '1' for HDA1)

Switch table :- UNIX Kernel maintains a set of tables using the major device numbers. It is used to find out which device driver should be invoked.

eg: fd → file table → inode table → switch table → device drivers ←

Mount command :-
↳ each file is located on a file system.
↳ each FS is created on a device associated with a device special file.
↳ ∴ when you use a file, UNIX can find out which device special file is associated with that file & send your request to corresponding device driver.
↳ Normally, all FS are mounted during system startup depending on /etc/fstab
↳ root FS is mounted by default.
↳ it is possible to mount a file system at any time using "mount" command

↳ a file system is mounted typically under an empty directory called mount point for the file system.
↳ when a FS is mounted, the system reads the Inode table and the super block into mem.

- Buffer Cache :-
↳ The FS also maintains a buffer cache, stored in physical, i.e. non-paged m/m.
↳ Used to store any data that is read from or written to a block device such as HDD, CDROM.
↳ It reduces disk traffic & access time.
↳ If data not present in Buffer Cache :-
  - system allocates a free buffer in buffer cache
  - reads data from disk
  - write/store data in buffer cache.
↳ If data is present but no free buffer,
  - system selects a used buffer
  - writes it to the disk.
  - marks buffer as free.
  - allocates it to requesting process.
↳ while this is going on, requesting process is in wait state.
↳ A process can use sync() system call to tell the system that any changes made by itself in Buffer cache must be written to the disk.

- I/O Handling :-
↳ The basic model of I/O system is a sequence of bytes that can be accessed either randomly or sequentially.
↳ There are no file formats & control blocks in a typical

↳ The I/O system is visible to a user process as a stream of bytes.
↳ A UNIX process uses descriptors to refer to I/O streams.
↳ Each process has a separate File Descriptor (FD) table.
↳ Valid descriptor ranges from '0' to a maximum descriptor number that is configureable.

0 → STDIN
1 → STDOUT
2 → STDERR

↳ If file is not opened, FD returns -1.

## System Calls :-

↳ System call code is physically stored / located in the kernel.
↳ Therefore, the first thing that is required to execute a system call is to change to kernel mode - so that kernel memory can be accessed.

File descriptor table → fd, process specific
File table → offset, mode, permission, pointer to inode
Inode table → Inode #, pointer to data blk
Switch table → only for device special files
data block → where file is stored

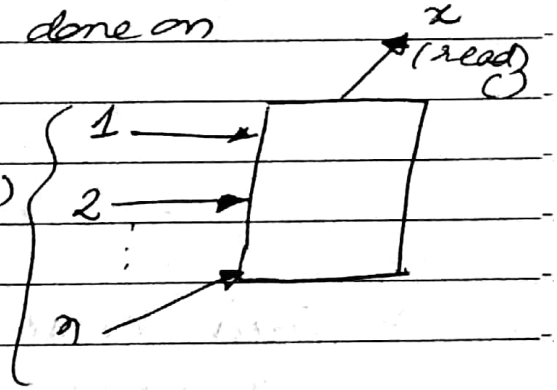
* Select() is a system call used to handle more than one file descriptor in an efficient manner.

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

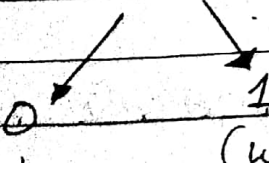
- Linux Uses internal routines for accessing a file.
  eg: namei() → Convert filename to an inode
  iget() → reads an Inode
  iput() → writes an Inode
  bread() → reads a block from HDD/Buffer cache
  bwrite() → writes a block from HDD/Buffer cache
  getblk() → get a free block in buffer cache.

## FILE LOCKING :-

- We use shared memory for increasing concurrency.
- The operations that can be done on a shared m/m are :-
  (i) read
  (ii) write
  (iii) append
  (iv) update
  (v) delete




- Hazards :- Read After Read (RAR) not a Hazard
  Read After Write (RAW) } Hazards.
  Write After Read (WAR) }
  Write After Write (WAW) }
  i.e. we can allow all processes to read a resource parallely.
- However, write, update, append & delete will create synchronization problem (race condition)
- We can deal with such situations using MUTEX (mutual exclusion)
  ↳ binary locking mechanism



* Lock and unlock are implemented as atomic operations.

— How to use a lock?

(i) declare a lock.
(ii) Initialize a lock
(iii) Lock()
critical section
(iv) unlock()

eg: declare → struct semaphore sem;
Initialize → sem.value = 1; //MUTEX
lock(sem); //sem--. (atomic)
:
unlock(sem); //sem++

* Linux kernel has 10000+ critical sections, but at process level we have 3-CS :-

1. File → file lock
2. Independent (w/o file) → Semaphore
   - → POSIX semaphore
   - → System V Semaphore
3. Thread level
   - → pthread_mutex() (POSIX)
   - → POSIX - unnamed semaphore.

## SEMAPHORES

* SVR4 (System V Release 4)
  - → is a UNIX release, which has implemented of a semaphore called SVR4 designed for servers.
  - → It uses complex systems like semaphore set

i.e. a semaphore array [n].

| | |
|---|---|
| We can get the max. value of 'n' using ipcs -ls command | where each semaphore out of these 'n' may be counting or binary (as per requirement) |

* POSIX Semaphores :- is very light weight.
  - It is binary or counting
    - in case of shared memory

- Types of locks :-
1. Mandatory locking :- lock entire file.
   - Only one user/file is allowed
2. Advisory locking :- also called record locking.
   - can lock specific byte range (records)
   - Granularity (can lock 1 Byte)
   - Improve performance.

- read lock :- allows many readers but not a single writer
- write lock :- allows only one writer but not a single reader
- locks and unlocks are performed by fcntl.

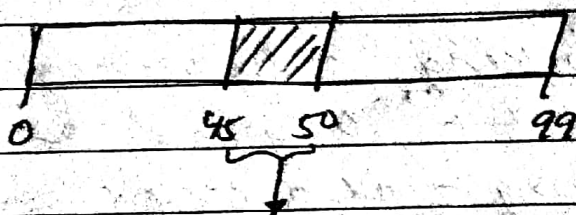
int fcntl(int fd, int cmd, struct flock &);

where, cmd may be :-
- F_SETLKW :- if lock not available, wait.
- F_SETLK :- if lock not available, returns -1.
- F_GETLK :- gets status of a lock file

and structure flock is defined as:-

```
struct flock {
    short l_type;   // type: read, write, unlock
    short l_whence; // from where you want to lock
    off_t l_start;  // when to start
    off_t l_len;    // lock length
    pid_t l_pid;    // your pid
};
```

↳ a writer has to wait until all readers exits. (starvation) so you can also implement ageing)




```
l_whence = SEEK_SET; (look at start)
{ l_start = 45; (move to byte 45)
  l_len   = 5;  (lock 5 bytes)
  l_pid   = getpid(); (get your own pid)
```

This example was for advisory locking
For mandatory locking,

```
l_start = 0;   } entire file will be
l_len   = 0;   } locked
```

CORE FILE:- (Core dump file)
default action of certain signals is to cause a process to terminate and produce a core dump file, a disk file containing

an image of processes's memory at the time of termination.
the image can be used in a debugger.

eg: create a situation:-

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int i;
    scanf("%d",&i);
}
```

- default size of core file is zero, so it is not created itself.

segmentation fault, coredumped.

- we can create it using ulimit command.
  ulimit -c 1024.
  then we can see & access it.

- we can see all system limitations at /proc/sys/kernel $
  ulimit -a

# LECTURE-6

- (Recap) :-

Q. Why is write lock not working if we are opening files in different processes?

Sol: Cooperating Processes :- If we execute same program in different incidences / instances. Practically, mobile app is an executable program. But, when we execute them as two different processes, there is no relation between one and the other window/process.

- Locks :-

1. read() ✓ (allowed) } then
2. write() X blocked

1. write() ✓ (allowed) } then
2. read() X blocked

1. write() ✓ (allowed) : when it exits
2. write() X wait
3. read() X allow this

(default), or when there are many readers but less writers.

↳ File locks does not have any counter like semaphore.

↳ super block :- contains important information about FS; like
- where File Descriptor (FD) is located
/proc/process_id (process space)
- total number of inodes

- files recently deleted

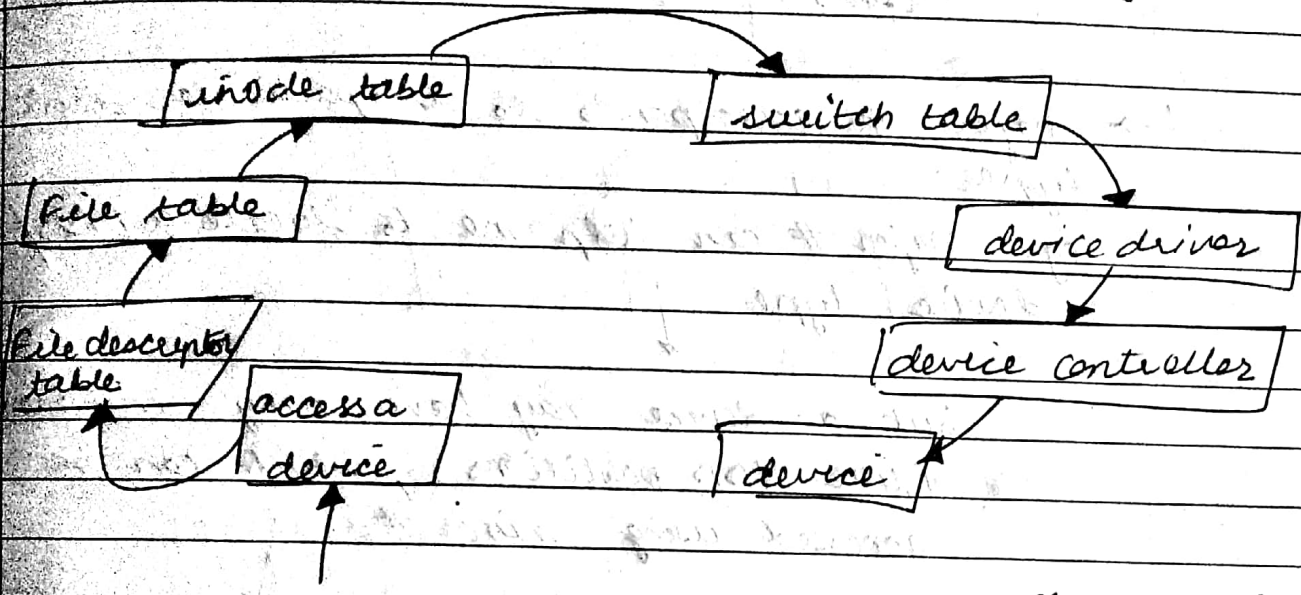
-:. Super block is a structure that contains info. about file system.

↳ Commond Used :-

f2chk (file system check)

usage :- $f2chk /dev/sda 4

- User Process accessing device special file :-




device special file
- character device special file
- block device special file.

switch table
- character device
- block device

| major # | entry points |
|---|---|
| 253 | fops structure (read, write, open, close, lseek, fcntl). |

Camlin Page
Date 24/08

↳ to see device files on your system
$ ls -l /dev/sda*

output format:

brwxr--r--  1  root  device  8,  0  Aug 20 ...

- b → type (block device)
- rwxr--r-- → permissions
- 1 → # of links
- root → owner
- device → group
- 8 → major #
- 0 → minor #

↳ major # corresponds to a particular device type

eg: major # can help me to locate a HDD/ device type

but a device may have many instances (like HDD partitions), which can be accessed using minor #.

↳ device file is a pointer to access a hardware device.

↳ System call definition is in Kernel (space), so it has to be executed in Kernel mode only

↳ Normal application program has a single entry point (main()), while, device driver has many entry points.

object code, not exe; so that whenever there is requirement, program executes then waits for future use.

↳ for changing mode from user to kernel, kernel generates s/w interrupt.

eg:
```
fun1(){ ... }
main(){
    fun1();
    read();
}
```

fun1() → user fn, called directly.

read() → push to stack, trap occurs.

↳ 0x80 is assembly instruction that changes the kernel from user mode to system mode.
— you can see the signal (s/w interrupt) values using $kill -l

↳ system call generates only traps unlike µP

↳ watch -n 0.1 cat /proc/interrupts

watch → live show

cat /proc/interrupts → display Hardware interrupts

↳ IRQ# :- Interrupt Request Number (for external devices only)
↳ Hardware interrupts.

↳ Trap is a Hardware interrupt, but called software interrupt because no external devices are involved.

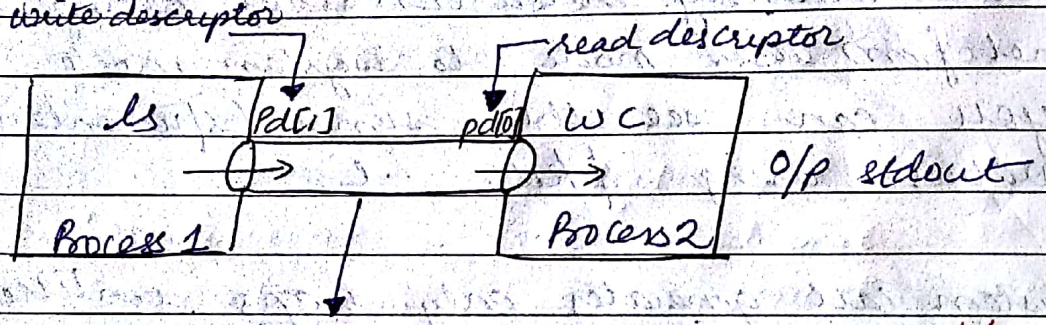
↳ dispatcher :-
There are many interfaces, not only system calls. So, dispatchers task is to decide whether to forward to system call or any other interface?

interrupt → interrupt table → interrupt service routine

* unlink is used to reduce link count by 1
  eg: like link --
  if a file has link count = 1,
  then, unlink file will delete it.
  ⇒ unlink("filename"); ≡ $rm filename

* dup :- used to duplicate a file descriptor
  ↳ used

  eg: ls | wc




usage :- for process 1 → close(1)
dup(pd[1])

stdout

write descriptor of pipe

for process 2 :- close(0)
dup(pd[0])

stdin

read descriptor of pipe

- This can be done together for using dup2 system call:

  dup2(file descriptor, desired #)

closes that file & opens/allocates it to file whose descriptor

↳ dup & dup2 internally uses fcntl()
(function control system call)

↳ fcntl(fd, dup-fd)

↳ fcntl() can be used for
- duplicating file descriptor
- file status GET_STATUS
- file locking

* stat() :- this system call gives information about file and file type.

* sync() :- synchronise the file
↳ updates the contents of Buffer Cache to HDD.

* select() :- this system call is used to monitor file descriptors and socket descriptors.
↳ If there is any change in any of the files, it returns true.
↳ also specifies how long you can wait.

eg: ls -l > myfifo ------> cat myfifo

I can specify how long can I wait
(socket descriptor)

- Homework :-
man 2 select.

- PROCESS MANAGEMENT
Process is a program in execution.
eg :- fork(), shell command, exe.

eg- ls|wc

```
if(!fork())
    { ls }
else
    { wc }
```

→ we can check process related information using size command.

eg: size a.out

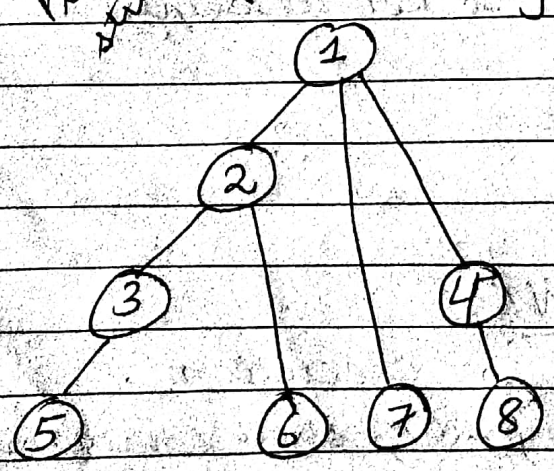
| text | data | bss | dec | hex | filename |
|---|---|---|---|---|---|
| 1577 | 568 | 8 | 2153 | 869 | a.out |

- text: It however does not gives info about stack, as it is a runtime entity.
- bss: uninitialized data
- dec: decimal format size
- hex: hexadecimal format size.

Creating child processes :-

```
main(){
    fork();
    fork();
    fork();
}
```

Process tree structure




---- fork() 'n' time (parent)

--- fork() once

--- fork() twice

--- fork() thrice

- we can see process tree for our system using pstree command.

- If I don't use "\n" here
  printf("Hello \n");
  fork()
  fork()
  fork()
  then, printf's output is still there in the buffer, that is being outputted by each child simultaneously this should though, not happen,
- In such a situation we insert a '\n', that helps clearing the buffer.

PS:- we donot have system call to flush the buffer contents.

# LECTURE-7

## - PROCESS MANAGEMENT :-

↳ A process includes program counter, CPU registers and process stack, which contains temporary data.

↳ **mode & space :-**

```
eg: int i;
    i = fork();
    if (i == 0) {
        child process  → returns 0
    }
    else {
        parent process → returns child pid
    }
```

However, the kernel source code has a convention of

```
if (!fork())
{ ...
}
else { . . }
```

## - modes

**user mode**
↳ can execute any non priviledged instr.

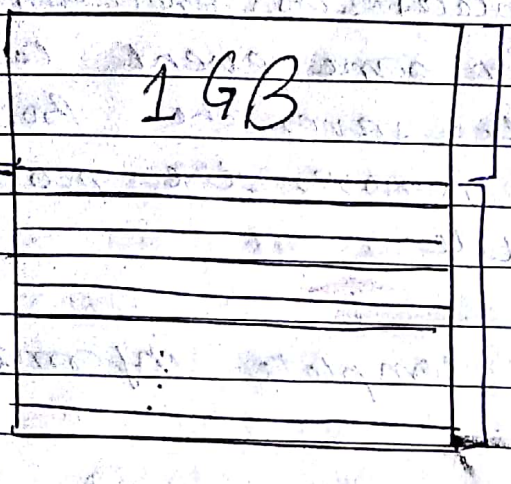
**Kernel/system mode**
↳ can execute any priviledged instr.

- priviledged instr is an instruction that access hardware, system calls etc.

- system calls can access kernel data structure and hardware.
- If there are not two modes, then, kernel may be crashed by user program.

↳ Space in memory / RAM :-

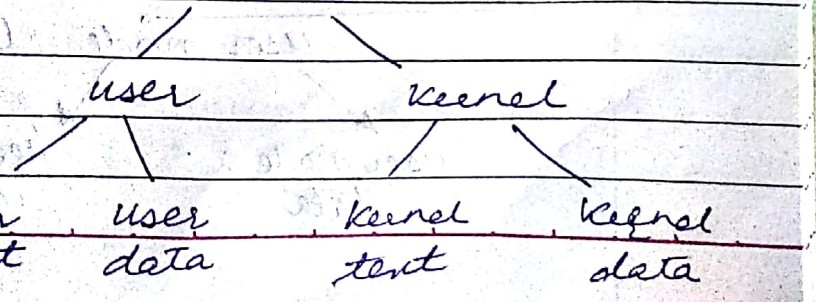
Kernel space → RAM (initial portion)
User space → RAM (remaining).




since kernel space is not in paged allocated format, so it's not swapped in swap space.

↳ swap space has swap file System -
↳ is a hardware partition
↳ only memory management can use it
↳ theoratically, it is extended n/m (RAM)

↳ Context Switch :-
In UNIX there are two modes of execution of a program : User & Kernel,
However, Intel has 4 modes



- each process has a virtual address space, references to virtual m/m are translated to m/m locations using address translation maps.

- execution context is changing from one process to another called context switch.
- when my currently executing process either completes its execution or is waiting for some event to happen, the kernel saves the Process Control Block (PCB) and removes the process from running state.

- PCB contains complete information about the process.

- kernel then loads next process' registers & pointers for execution.

- This whole process is called context switch.

→ perprocess object :-
each process can access kernel D.S. It looks as if each process has its own kernel D.S, though, they're accessing it parallely.

- Execution Context :-
  - user mode (less privilege)
    - user mode's data
    - ~~kernel mode~~ user space.

Kernel mode (high privilege)

user space  kernel space

eg: rescheduling process privileges needs Kernel mode + Kernel DS.

- User mode can access only process space.
- system calls and signals are handled in Kernel mode but in process context, may access process & system space.
- interrupts and system wide tasks are handled in Kernel mode & system context, and must only access system space.

- Process Structure :-
  - Every process in Linux is represented by a task_struct data structure, which is very big. (atleast 5 register pages).
  - whenever a new process is created, a new task_struct structure is created by the kernel & complete process information is maintained by it.
  - when it is terminated, then corresponding structure is removed.
  - it uses doubly linked list for faster access

the structure task_struct is as follows :-

```
struct task_struct{                                    1-stopped
    volatile long state; // -1 unrunnable, 0 runnable, k
    void *stack;
    atomic_t usage;
```

- Process States:-

[R] Running
[S] sleep interruptable → process waiting for some signal
[D] sleep uninterruptable → signal waits until sleep is over eg: file lock
[T] stopped
[Z] Zombie

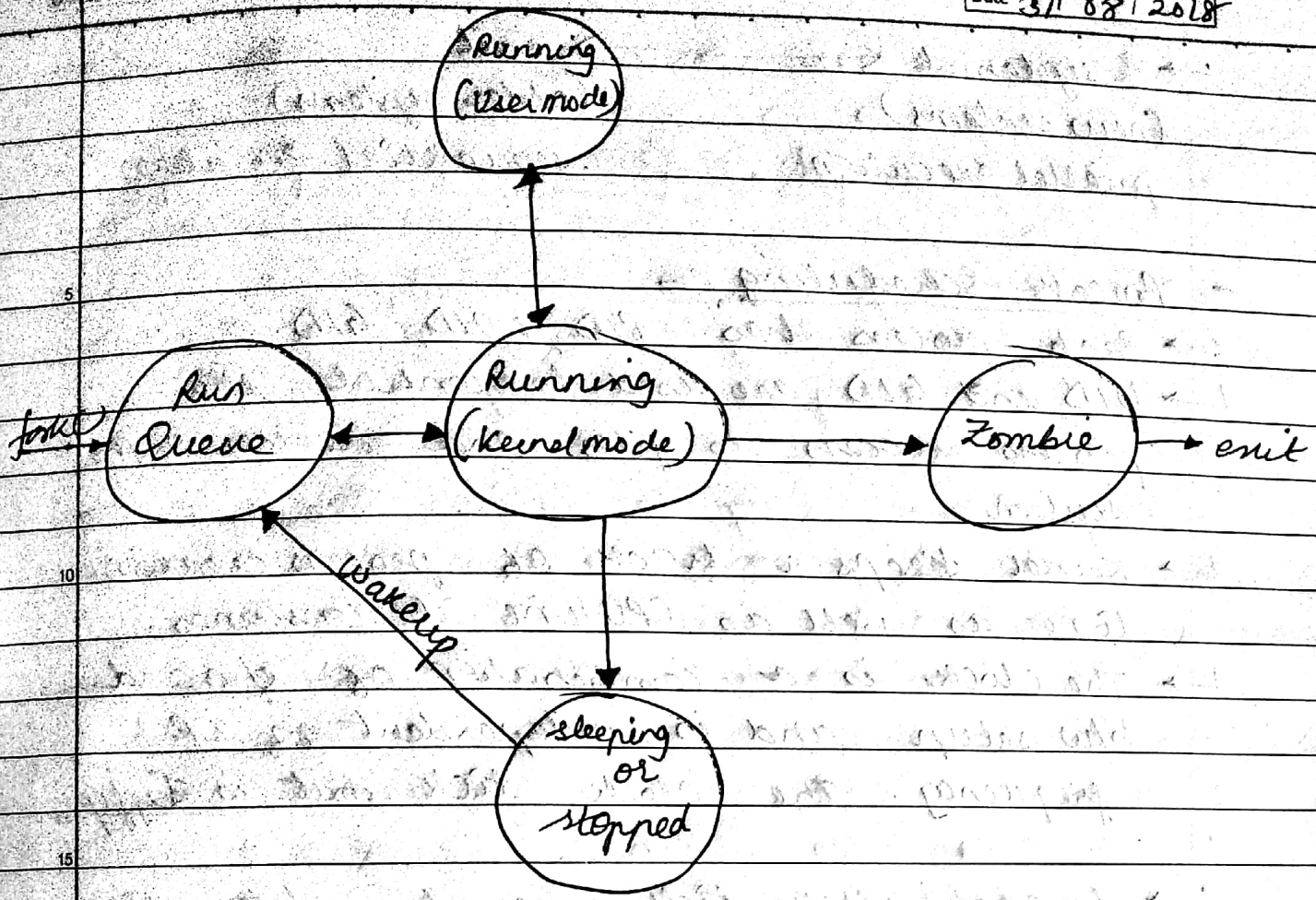
↳ However, child, parent, orphan, daemon are types of process.

↳ sleep state → whether interrupted (killed) or not?

↳ in Kernel level program I have (semaphore)
down_interruptable(){..} up(){}
↖ for interruptable ↗
and for uninterruptable sleep
down(){..}

↳ stopped: this state is temporarily suspended
↳ we can restart/resume the process
↳ eg: ctrl+Z or ^Z or SIGSTOP (19)

↳ Zombie:- parent's execution completes, but child still exists/executes and has not yet given it's exit status to parent yet
A zombie process is unkillable,
i.e. kill -9 PID will not work




↳ executing a shell command:-

$ls

fork()

child process

parent (shell)
waits for child's
execution to
complete

uses exec("ls")

execute ls

returns status
to parent

↳ systemd ⟷ init
(new systems) (older systems)
parallel execution. sequential execution

- Process Scheduling :-
  - ↳ Each process has PID, UID, GID.
  - ↳ UID and GID are used to control the process access to files and devices of the system.
  - ↳ Kernel keeps a track of process creation time as well as CPU time it consumes.
  - ↳ The clock is a combination of s/w and h/w setup and is independent of CPU frequency. The clock tick unit is Jiffy.
  - ↳ for each clock tick, Kernel updates the amount of time that the current process has spent in system / user mode. Linux also supports process specific interval timers.
  - ↳ job of a scheduler is to select the most deserving process to run out of all of the runnable processes in the run queue; implement fair scheduling to avoid starvation, suitable scheduling policy & update state of process in every jiffy
  - ↳ Policy/Algorithms :-
    - FIFO, RR, Priority } linux uses them
    - used in embedded systems → SRTF, SJF

↳ In linux, priority can be provided explicitly.
↳ priority - many linux variants support Real Time scheduling priority.

↳ Linux Priority

Nice value
↳ -20 to 19
↳ with default zero
↳ time slices :-
min - 10ms
default - 150ms
max - 300ms

real time priority.
↳ 0 - 99 range.
↳ all RT priority have higher priority than normal process
↳ implemented in accordance with POSIX.

| Real Time | | Non-Real Time |
|---|---|---|
| 0 SCHED_FIFO (fixed priority) | SCHED_RR 99 | SCHED_OTHER (dynamic priority) 139 |
| highest | | lowest |
| no time slice (it can run until it blocks or explicitly yields the processor) | | (identical to SCHED_FIFO but has a predetermined time slice). |

| Non-Real time | | |
|---|---|---|
| -20 | 0 | +19 |
| Time slice: 300ms (highest priority) | 150ms | 10ms (lowest priority) |

(Nice levels)

$ top OUTPUTS

priority form

RT 98 89 55 -5 0 38 39

→ rescheduling process priorities:-

- aging → waiting for CPU, priority increases
  → using CPU, priority decreases.

- Real time priority is static.
  You can assign your own priorities, as a root user, then it's completely your responsibility.

- Non-real time priorities are dynamic (based on clock) (system takes care).
  → small task → FIFO
  → Complex task → RR.

- eg:- assigning priority to Non Real time Process
  $ nice -19 ./a.out

- multilevel process scheduling is done for load balancing.

exec :- execute 'ls' inside a program.

exec("ls");
exec("./a.out");

text
data
stack

text data stack

Changes process image completely.

- reentrant means :- same instances execute in multiple instances/places parallely.

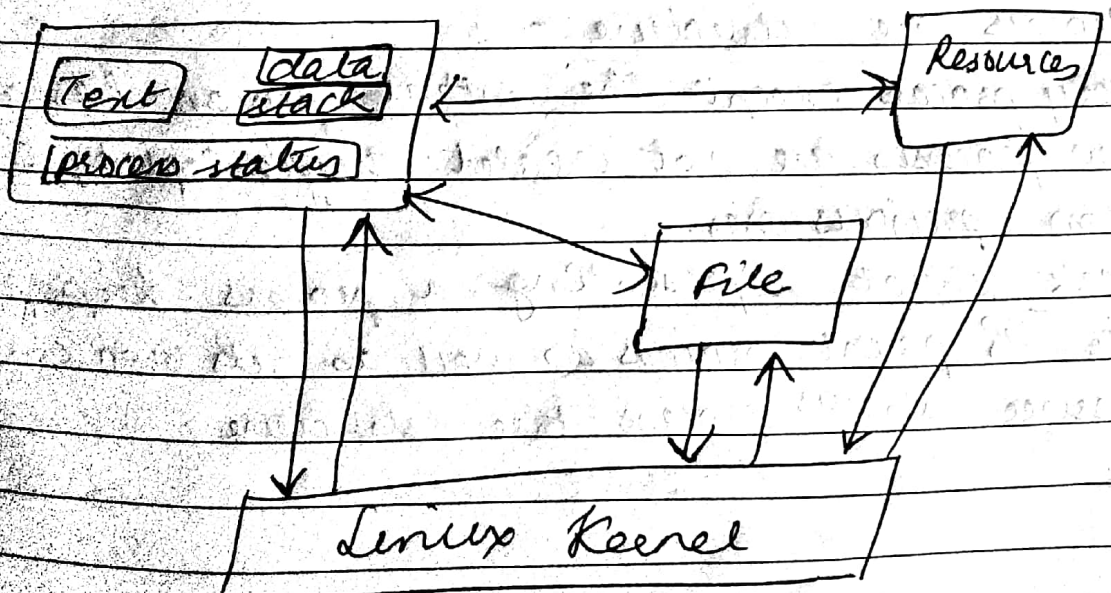
- to check CPU frequency :-
  cd /proc
  cat cpuinfo | head

- timer frequency :-
  100 Hz → server & desktop (10ms)
  250 Hz
  500 Hz
  1000 Hz → embedded/RTOS (1ms)

overhead; every timer interrupt, rescheduling occurs.
eg: 1ms sleep in desktop (100Hz), will wake after 20ms !

- timers → high resolution
  → interval : eg: wait(10), sleep(10)

- Process creation :- each process has its own text, data and stack.



# LECTURE - 8

- **PROCESS SCHEDULING :-**
this can be done using System calls :-
  - ↳ nice() :- to set a processes scheduling policy.
  - ↳ sched_setscheduler() :- set policy
  - ↳ sched_getscheduler() :- get priority & algorithm
  - ↳ sched_setparam() } real time
  - ↳ sched_getparam() } priority.
  - ↳ sched_rr_get_interval() → time slice for round robin.

↳ **process creation :-**
- Command executes in a shell.
- When we execute a command, the shell searches the corresponding command's exe image in the 'PATH', loads it & executes.
- for execution, shell uses fork() and creates a new child process & child process's image is replaced with command's exe.
- after execution completes, child gives its exit status to shell (parent)

↳ **Process tree structure :-**
- every process except the initial process has a parent
- new processes are not created. They're copied, cloned from previous ones.
- task_struct representing a process keeps pointer to it's parent process as well as its own children
- $pstree shows process tree structure.

execution → parent & children executes concurrently
→ parent waits for child to terminate.

address space → child duplicates that of parent
→ child has a program loaded into it.

↳ creating child processes:-

pid_t fork(void); → creates child process

↳ all statements after fork() system call in a program are executed by two processes — original one & new process (child) created by it.

```
if(!fork())
{ /* child code */ }
else
{ /* parent code */ }
```

wait(0) OR waitpid(pid, ...)

↳ untraced memory leak can lead to segmentation fault.

for(;;) fork(); → called fork() bomb.

Creates ∞ processes a user can create, It'll terminate this process then.

- Copy On Write (COW) :-
  - Instead of copying the address space of parent Linux uses the COW technique for economical memory use.
  - Parent space is not copied, it can be shared by both child & parent, but memory pages are marked as write protected.
  - If a parent/child wants to modify the pages, Kernel copies the parent pages to the child process.
    - Advantage :- Kernel can defer or prevent copying of parent process address space.

- Zombie Process :-
  - When a child exits, it has to give the exit status to the parent process.
  - If the parent process is busy or suspended then child will not be able to terminate.
  - Such a process state is called Zombie state.

eg:-
```
if (!fork())
{
    getpid();
    exit(0);
}
else sleep(60);
```

eg:
```
if (!fork())
    printf("Child pid: %d\n", getpid());
else
    getchar();
```

usually child pid > parent pid
so, top -p pid can be used to see process state.

↳ defunct another word used to identify zombie state of a process.

If my process creates n- zombies, then I can't clear the memory.
even, kill -9 pid-of- zombie would not work.

- Orphan process :-
If parent exists before child, the child will become orphan process & the init(grand parent) will take care of child process.

eg. if (!fork())
{ print(ppid before orphan)
sleep(1)
print(ppid after orphan)
}
else {
print pid.
sleep(3)
}

- exec Family :-
↳ to run a new program, we use one of the "exec" family function calls and specify :-
• pathname of the program to run, name of program
• each parameter to program.
• (char *)0 or NULL as the last parameter to

specify the end of parameter list.

↳ The family includes!-

(i) int execl( path, arg..)
(ii) int execlp( file, arg)
(iii) int execle( path, arg, ..... const envp[]);
(iv) int execv( path, const argv[])
(v) int execvp( file, const argv[])

↳ all above f^s internally call execve system call.

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

eg: if (!fork()) //executing ls
execl("/bin/ls", "ls", (char*) 0);
else
wait(0)

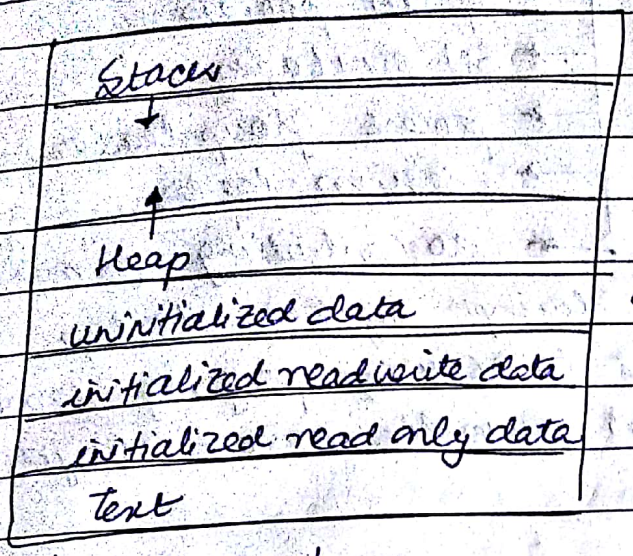
can accept 256 arguments vargs[]

↳ However, we can use:
system("ls"); to do the same thing as above, but it is discouraged

system ("ls")
↓
create child shell
↓
fork()
↓
execl("/bin/ls", "ls", 0);

very costly!
use "exec" family its less costly.

- Executable image :-




eg: $size a.out

↳ text portion contains actual m/c instructions that are executed by hardware.

↳ when a program is executed by OS, the text portion is read into m/m from disk file, unless OS supports shared text & a copy of program is already executing.

↳ data :-

(i) initialized read only :- elements initialized by the program, read only while process is executing.

(ii) initialized read-write :- data elements initialized by the program & may have their values modified during the execution of the program.

(iii) uninitialized ~~read~~ not initialized by program, set to zero before execution starts.

↳ heap :- dynamically allocated memory.

↳ stack :- used dynamically while the process is running to contain the stack frames that are used by many programming languages.

→ stack frames contain return addresses' linkage for each fn call & also data required for it.

↳ Kernel context of a process is maintained & accessible only to the Kernel. This area contains info that the kernel needs to keep track of the process & to stop/restart the process while other processes are allowed to execute.

→ Daemon process:-
- executes in background
- orphan process
- SU privileges
- process group leader
- session leader
- no controlling terminal

↳ execution starts during system startup.
↳ wait for an error to occur, perform some specific task on periodic basis (cron job)

↳ creation:-

```
int init_daemon(void)
{
    if(!fork()){
        setsid();  → change to session leader
        chdir("/"); → root privileges
        umask(0); → all permissions
        /* specify job */
        return 0;
    }
    else
        exit(0);
}
```

↳ eg:-

```
main(){
    if(!fork()){
        setsid();
        chdir("/");
        umask(0);
        while(1){ sleep(2);
            print(pid); }
}
```

else exit(0);
}

↳ you cannot stop execution of a daemon process, nor can you kill it without being root user.

↳ if you are not a root user,
sudo su
kill -9 pid_of_daemon

↳ terminal type for daemon is "?"
↓
so not to associate with any particular terminal/process.

↳ process types :-
Parent, child, Orphan, Daemon

↳ Process states :- → pid & state of process can be checked using ps, top, /proc
Running (R), ~~Zom~~ Stopped (T)
sleep → uninterrupted (D)
↳ interrupted (S)
Zombie (Z).

# PPT-2

## - INTER PROCESS COMMUNICATION :-

↳ shared resources need to be synchronized from the concurrent access by many processes.

↳ IPC mechanisms :-
- Data Transfer,
- Sharing data
- event notification
- resource sharing
- process control.

- Primitive :
  - unnamed pipes
  - named pipe (fifos)
- System V IPC :
  - message queues
  - shared m/m
  - Semaphores
- Socket programming.

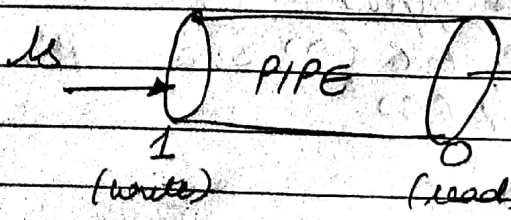
↳ PIPE :- (half duplex)

↳ eg: related process :-

ls | wc

child → ls

parent → wc




- pipe ends are represented by two descriptors and is used to communicate between related processes.
- Half duplex, data passed in order.
- Uses circular buffer & has zero buffering capacity
- read & write are blocking calls.

eg:- int fd[2];

(P) = pipe(fd);

↳ 0 on success, on error -1

so, write (fd[1], ..... ); // writing to pipe

read (fd[0], ..... ); // reading from pipe.

↳ parent can send data & child can read, and vice versa.

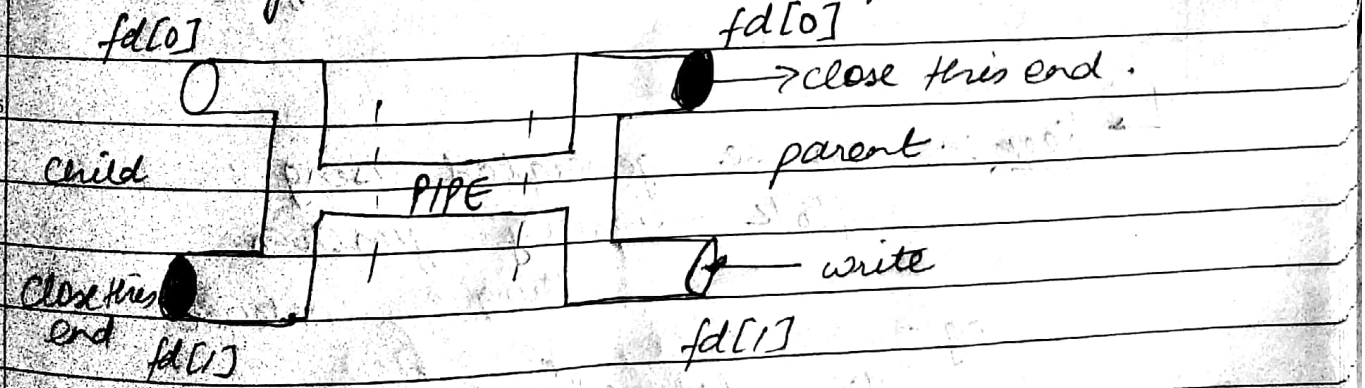
↳ unused ends should be closed.

eg:- ls -Rl | grep ^d | wc -l ] → # of directories in PWD.

ls -Rl / | grep ^d | wc -l ] → # of directories in system.

↳ for un-related processes, use fifo.

↳ One way communication from parent to child.



Date 09/09/18

# LECTURE- 9 (LAB)

↳ pipe & fifo :-

only symbol, communication b/w related processes only

created in HDD, so communication b/w unrelated processes also work.

↳ Zero Buffering Capacity
↳ One way (half duplex)
↳ data transferred as stream of bytes, doesnot have any message boundaries

↳ Message Queue :-

messages stored in Kernel space

- create a queue 'A' (msg Q id)
- post n-messages.
- create a connection to 'A'
- Retrieve the message

↳ overcomes FIFO limitation
↳ any process with permission to access the queue can retrieve messages.

↳ Key :- can be generated using :-
ftok library function.

eg:-

```
Key = ftok(".", 'a');
```

(string, integer)

unique key (integer)

↳ syntax of functions :-

(1) int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);

↳ msqid :- message queue ID, ~~address of~~
↳ msgp :- address of structure
↳ msgsz :- size of message text.
↳ msgflg :- 0 or IPC_NOWAIT,
wait nowait

(2) size_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg);

↳ msgtype argument is used to retrieve a particular message :
0 → retrieve in FIFO order.
+ve → retrieve exact value of message
-ve → first message or <= absolute value

↳ on success returns # of bytes actually copied into the message text.

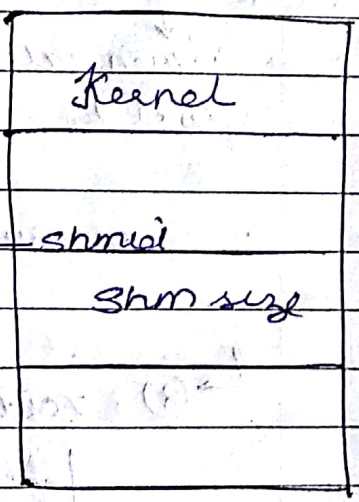
(3) to remove a message queue, use msg control
msgctl(msqid, IPC_RMID, NULL)
= $ipcrm msg msqid

Limitations of message queues:-

- effective if small amount of data is copied/transferred.
- very expensive for large transfers.
- message in queue cannot be reused, i.e. "NO MULTICAST".
- Very slow;
  during sending & receiving, the message is copied from user buffer to kernel buffer & vice versa, so it involves two data copy operations, which results in poor performance.

SHARED MEMORY

- has read & write but no append options because we create shared m/m of static size.
- fastest IPC mechanism.
- used to provide access to:
  - global variable,
  - shared libraries
  - word processors
  - multiplayer gaming environment
  - Http daemons.




- does not require an intermediate kernel buffer.
- easy to use. After shared m/m segment is set up, it is manipulated exactly like other memory area.

↳ steps involved:-

- Create shared m/m using shmget()
- connect to m/m & obtain a pointer to
- read/write & change access mode to m/m
- detach from physical m/m
- delete shared segment.

↳ If not, then m/m leak

↳ system calls:-

→ (1) int shmget(key_t key, int size, int shmflg);

return value of ftok()

size of shared memory

shmflg= IPC_CREAT/0

on success, returns m/m ID; else -1

↳ (2) void *shmat(int shmid, void *shmaddr, int shmflg);

↳ used to attach a created shared m/m segment onto a process address space

eg: data = shmat(shmid, (void *)0, 0);

↳ pass SHM_RDONLY

↳ a pointer is returned on successful execution of the system call & the process can read & write to the segment using the pointer.

→ (3). The detachment of attached shared m/m segment is done by smdt() to pass the address of the pointer as an argument.

↳ int shmdt (void *shmaddr);

→ (4) to remove shared memory,

int shmctl(shmid, IPC_RMID, NULL);

↳ return '0' on success, else -1.

↳ **Race Condition** :-

- since many processes can access the shared memory, any modification done by one process in the address space is visible to all other processes.
- since the address space is a shared resource, the developer should implement a proper locking mechanism to prevent race condition in shared m/m

shared m/m → race condition →

synchronization mechanism

locking mechanism

semaphore (for mutual exclusion)

→ SEMAPHORE :- → synchronization tool
→ avoid busy waiting
→ used in :-
- shared m/m segment
- message queue
- file

↳ operations :-

red light P() → decrement operation
↳ lock

green light V() → increment operation
↳ unlock

} atomic operations

• incrementing operation :-

```
int V(int i)
{ i=i+1; (unlock)
  return i;
}
```

• decrementing operation :-

```
int p(int i)
{ if(i>0)
    then i--; (lock)
  else
    wait till i>0 → locking mechanism for multiprocessor
  return i;
}
```

SPIN-LOCK
(if lock not available, execute tight infinite loop)

↳ if a procas shares shared m/m, it will lock() it by asking semaphore to decrement the counter.

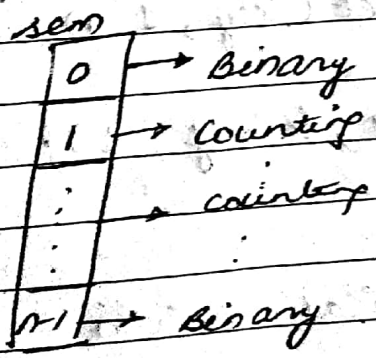
↳ depending upon the current value of counter) sem will either be able to carry this operation or will wait until the operation becomes possible.

↳ if the current value of C > 0, the decrement is possible, else, procas waits.

↳ **SystemV semaphores :-**

- provides a semaphore set - that includes a # of semaphores.
- User can decide the # of semaphores in the set.
- each semaphore can be binary or counting.
- each semaphore can control access to one resource.

eg: semaphore set sem[n]; → # of semaphores




↳ **POSIX Semaphore :-** → unnamed → within a process (multithreading)

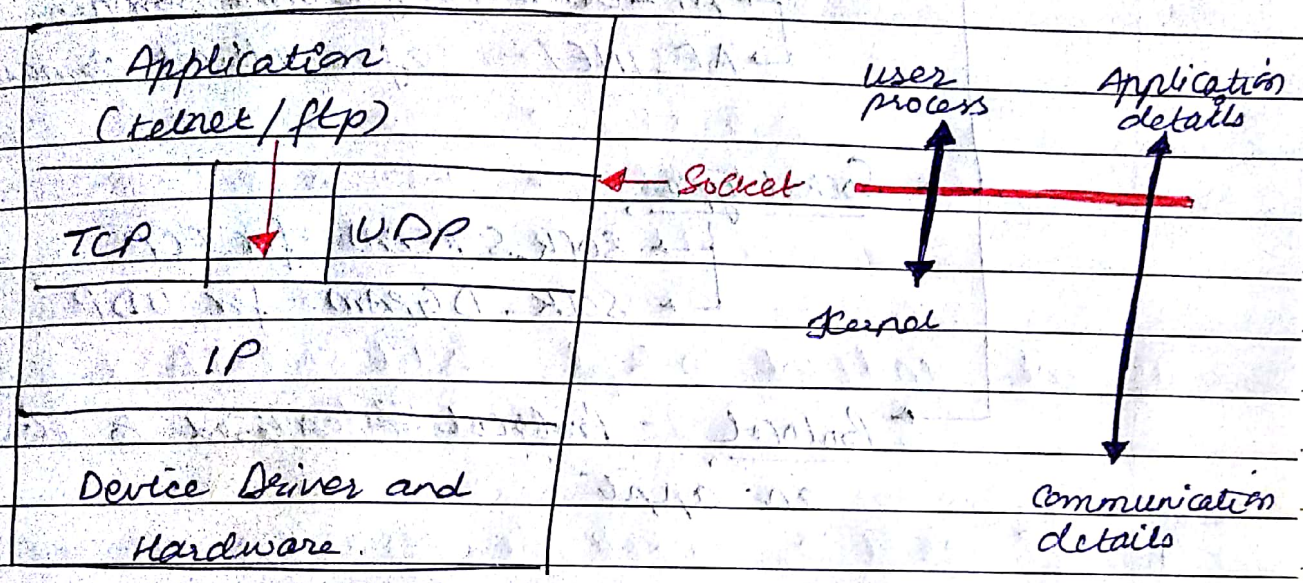
↳ named → inter process.

↳ implementation :-

```
• union semun {
        int val;    // SETVAL
        struct semid_ds *buf;    //for IPC_STAT, IPC_S
        unsigned short int *array;    // GETALL, SETAL
    };

• union semun arg;
  semid = semget(key, 1, IPC_CREAT|0644);
  arg.val = 1;    //binary semaphore

• semctl(semid, 0, SETVAL, arg);

•  struct sembuf {
        short sem_num;    // sem_num; 0 mean
        short sem_op;    //operation:- lock, unlo
        short sem_flg;    // (0, SEM_UNDO, IPC_NO
    };

• sembuf buf = {0, -1, 0}    // -1 previous val
• semid = semget(key, 1, 0);

  semop(semid, &buf, 1);    //locked

        Critical section

  sembuf buf.sem_op = 1;
  semop(semid, &buf, 1);    //unlocked
```

## SOCKET :-

- a socket is used to communicate between different machines (different IP addresses).
- Socket of type SOCK_STREAM is a full duplex byte stream.




- a socket is a communication end point and represents abstract object that a process may use to send or receive messages.
- Two most relevant communication API's for UNIX systems are Berkley Sockets & System V transport Layer Interface (TLI)
- Connection
  - TCP → connection oriented
  - UDP → Connection less.
- more parameters must be specified for n/w connection than for file I/O.
- UNIX file I/O is stream oriented.
- N/w interface should support multiple communication protocols.

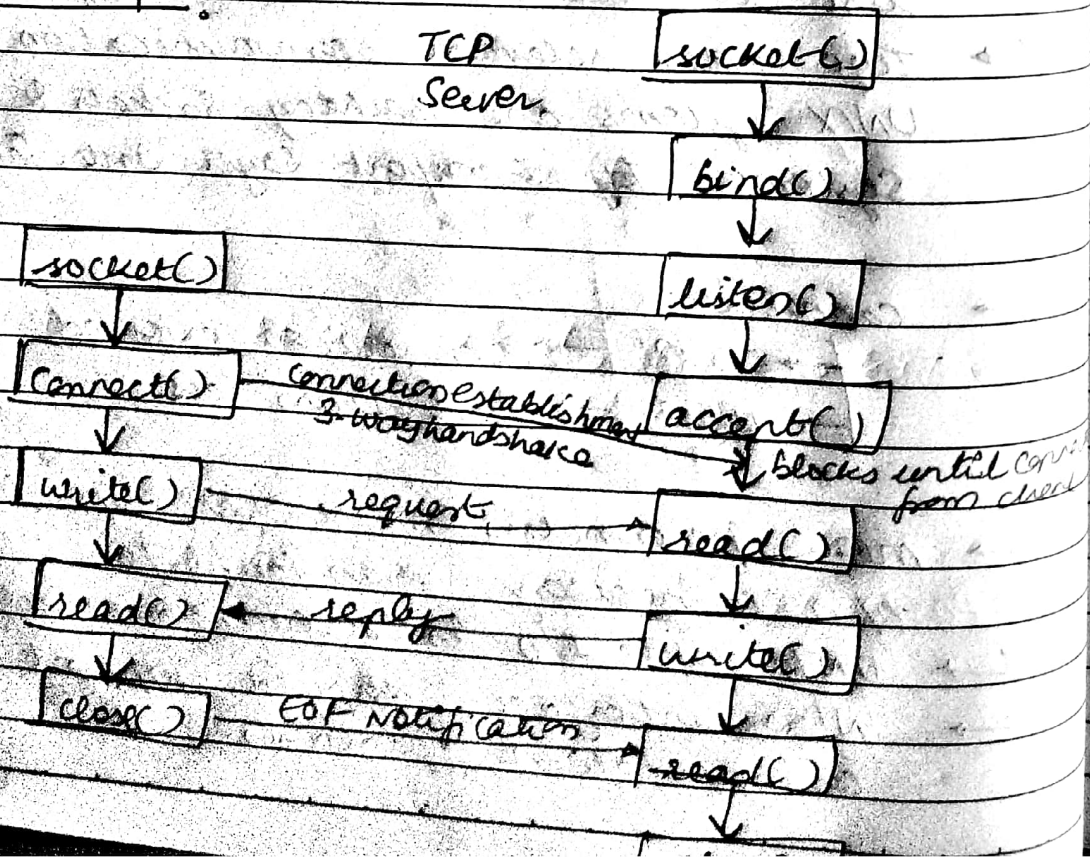
→ system call:-

int socket(int domain, int type, int protocol);

- domain (Address family (AF))
  - AF_UNIX for UNIX domain
  - AF_INET for Internet domain
- Socket type:
  - SOCK_STREAM for TCP
  - SOCK_DGRAM for UDP
- Protocol :- Protocol # is used to identify an appl'

→ it returns a socket descriptor on success, -1 on failure.

→ Socket f"s :-



Q: 3way handshake in sockets?

Camlin Page 081
Date 09/09/18

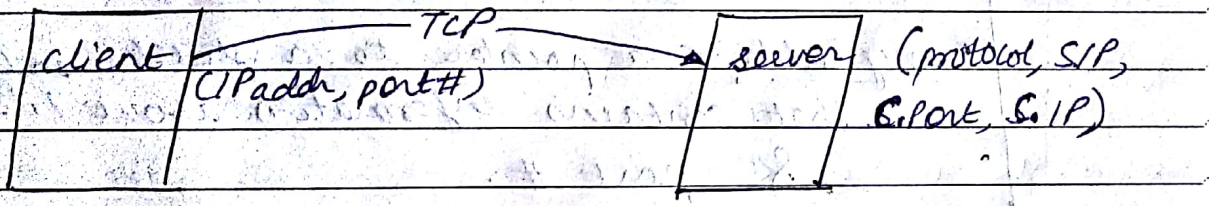
→ sock structure :-

```
struct sockaddr_in {
    short int sin_family; //address family
    unsigned short int sin_port; //port #
    struct in_addr sin_addr; // IPaddress.
};
```

→ the in_addr structure used to declare sin_addr is :-

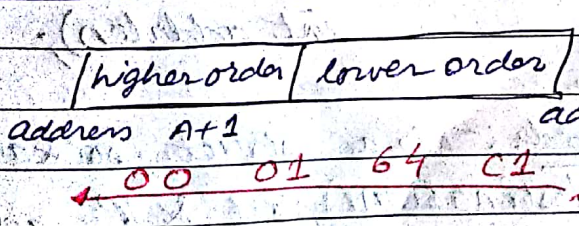
```
struct in_addr {
    unsigned long s_addr; //4B IPaddr
};
```

→ backlog :- How many client can wait in a queue?
max = 5.

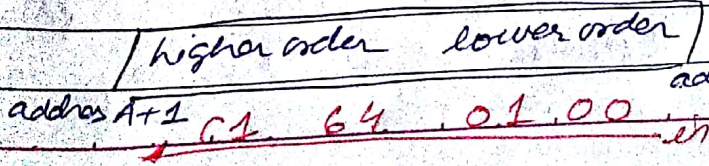



→ Byte order :- (Byte ordering for : 91,329. Hex: 00 01 64 C1)

→ Little Endian byte order. eg: Intel series




→ Big Endian byte order :- eg: IBM370, motorola, Internet



↳ Socket system calls:-

- IP uses big endian byte ordering called Network byte ordering.
- following f"s allow conversions b/w formats:-

```
#include<netinet/in.h>
htons() → Host to Network Short
htonl() → Host to Network long
ntohs() → Network to host Short
ntohl() → Network to host long.
```

(i) int bind(int sockfd, struct sockaddr *my_addr, int addrlen);

- sockfd :- socket file descriptor returned by socket()
- my_addr :- a pointer to a struct sockaddr that contains information about IP address & port #.
- addrlen :- set to sizeof(struct sockaddr)

(ii) int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

- serv_addr :- is a struct sockaddr containing destination port & IP address.

(iii) int listen(int sockfd, int backlog)

→ backlog :- # of connections allowed on the incoming queue. Should be >0 as servers always expect connection from client

→ this system call converts an unconnected socket into passive socket.

→ on successful execution, it indicates that the kernel should accept incoming connection requests directed to the this socket

(iv) int accept(int sockfd, void *addr, int *addrlen)

→ addr : ptr to a struct sockaddr_in. the information about the incoming connection like IP address & port # is stored

→ addrlen :- local integer variable that should be set to sizeof(struct sockaddr_in) before its address is passed to accept().

(v) close(sockfd);

→ socket descriptor can be closed like a file descriptor

(vi) int shutdown(int sockfd, int how);

→ sockfd :- socket file descriptor to be shutdown

→ how :- if it is :- 0 → further receives are disallowed
1 → further sends are disallowed
2 → further sends & receives disallowed

X1, P: 10

↳ shutdown() call gives more control than close(sockfd) over how the socket descriptor can be closed.

↳ server program :-

```
struct sockaddr_in serv, cli;

sd = socket(AF_INET, SOCK_STREAM, 0);

serv.sin_family = AF_INET;
serv.sin_addr.s_addr = INADDR_ANY;
serv.sin_port = htons(portno);

bind(sd, &serv, sizeof(serv));
listen(sd, 5);

nsd = accept(sd, &cli, &sizeof(client));
read/write(nsd, ....);
```

↳ Client program :-

```
struct sockaddr_in serv;
sd = socket(AF_INET, SOCK_STREAM, 0);

servaddr.sin_family = AF_INET;
serv.sin_addr.s_addr = inet_addr("ser ip");
servaddr.sin_port = htons(portno);

connect(sd, &server, sizeof(server));
read/write(sd, ....);
```

## → Iterative vs Concurrent Server :-

→ **iterative server** :- services only one client request at a time.

```
nsd = accept(sd, &cli, ....);
while(1){
        read/write(nsd, ....);
        :
}
```

→ **concurrent server** :- services many client requests concurrently.

```
while(1){
    nsd = (accept(sd, &client, ....));
    if(!fork()){
            close(sd);
            read/write(nsd, ....);
            exit();
    }
    else
            close(nsd);
}
```

Camlin Page 086
Date 14/09/18

# LECTURE - 10

## ALARM & TIMERS

unsigned int alarm(unsigned int seco

- used to set an alarm for delivering SIGALARM signal.
- on success, returns zero.

- there are three interval timers :-

- ITIMER_REAL :-
  - timer counts down real/wall clock ti
  - at each expiration, SIGALARM is generate

- ITIMER_VIRTUAL :-
  - counts down against the user mode CP time consumed by the process
  - measurement includes CPU time consume by all threads in a process
  - At each expiration, a SIGVTALRM is generated.

- ITIMER_PROF :-
  - This timer counts down against the to (both user & system) CPU time consum by the process.
  - At each expiration, SIGPROF signal is generated.

- get & set timer :-
  - get value of an interval timer.

int getitimer(int which, struct itimerval

on success returns zero, & timer value is

stored in the itimerval structure
eg: ret = getitimer(ITIMER_REAL, val);

↳ set value of an interval timer -
int setitimer(int interval_timers, const
struct itimerval *val, struct itimerval
*oldval);
↳ on success returns zero.
eg: setitimer(ITIMER_PROF, &value, 0);

↳ we can check this from /proc/cpuinfo.

- Time stamp Counter :- (TSC)
system provides high resolution time measurement through the time-stamp counter which counts # of instructions since boot.

```
#include<sys/time.h>
unsigned long long rdtsc()
{
    unsigned long long dst;
    __asm__ __volatile__ ("rdtsc":"A"(dst));
    return dst;
}

main(){
    long long int start, end;
    start = rdtsc();
    // job;
    end = rdtsc();
    printf("Diff: %llu\n", end-start);
}
```

it has to change everytime I execute

in seconds = (end - start) / clock speed

14/09/18

## → RESOURCE LIMITS :-

- OS enforces limits for certain system resources it can use.
- "ulimit" shell built-in can be used to set or get the limits.
- Applicable to a specific process.
- ulimit - a returns the user limit values

$ulimit -a → all

-c : max size of core file created.
-f : max size of files created
-l : max amount of m/m that can be locked using mlock() system call
-n : max # of open file descriptor
-s : max stack size
-u : max # of processes available to single user.

## → Hard & soft limits :-

Each user/resource has two limits -

(i) Hard limit :-

- absolute limit for a particular resource. It can be a fixed value or "unlimited"
- only super user can set hard limit
- ulimit command has "-H" or "-S" option to set hard/soft limit.
- Default is soft limit.
- Hard limit cannot be increased once it is set.

(ii) Soft limit :-
- user definable parameter for a resource.
- value from 0 to <hard limit>
- any user can set soft limit.
- limits are inherited (new values are applicable to descendent processes).

- Resource limitation :-

getrlimit() / setrlimit() → system calls.

syntax :-

getrlimit (<resource>, &r)
setrlimit (<resource>, &r)

where, 'r' is of type struct rlimit.

- to get system configuration at runtime :-
long sysconf(int name);
eg: ret = sysconf(_SC_CLK_TCK);
- on success returns value given in system limits

- Multithreading :- {Light Weight Program (LWP)}
- Thread is a sequential flow of control through a program.
- If process = program in execution, then thread = function in execution.
- Types :- → Single Threading
→ Multi Threading.

- Kernel creates LWP for each process (if it requires). We can set priority for each
- Threads within a process share
  - instructions of a process
  - process address space & data
  - open file descriptors
  - signal Handlers
  - pwd, uid & gid
- Created threads have (its own):-
  - Thread identification no. (tid)
  - pc, sp, set of registers
  - stack
  - priority of threads
  - scheduling policy
- advantages:-
  - less time for creation of new thread is requi
  - termination of a thread & communication b/w threads is easy.
  - improves responsiveness.
  - use multiprocessors efficiently.
  - improve program structure
  - use fewer system resources.
  - specific application in uniprocessor m/cs.
- applications:-
  - file server on Local Area N/w
  - Graphical User Interface
  - web applications

2. thread creation :- → POSIX

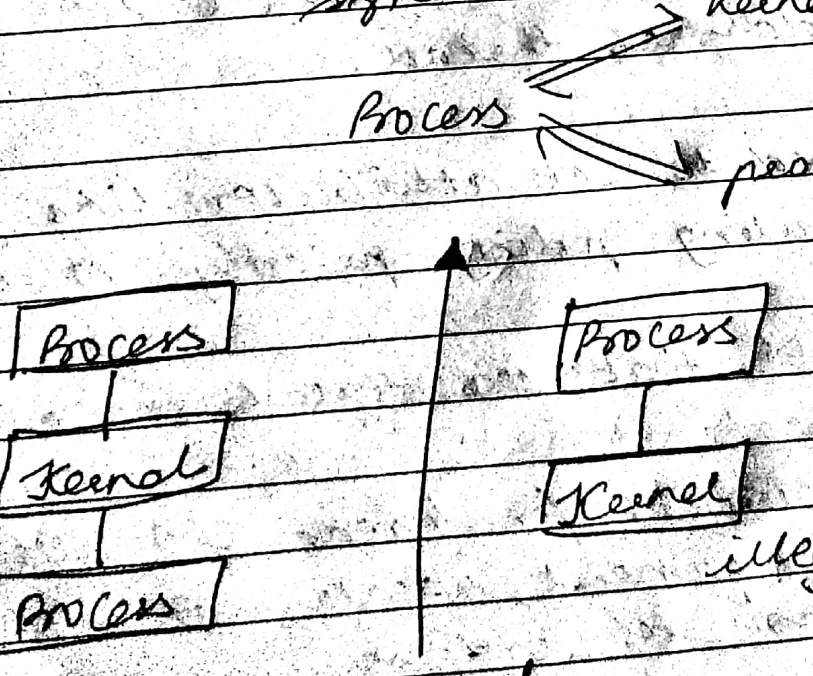
```
#include <pthread.h>
void thread_func(void)
{ printf("Thread id: %d\n", pthread_self());
}
main() {
  pthread_t mythread;
  pthread_create(&mythread, NULL, (void*)thread_func, NULL);
}
```

↳ needs to be linked at compile time (explicitly)
$cc pthread.c -lpthread.

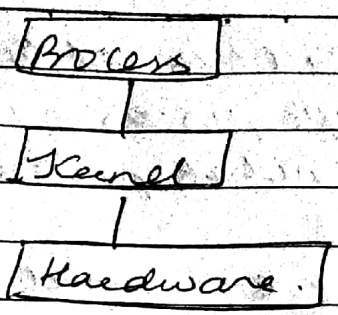
↳ pthread_t ≡ unsigned long int.

↳ arguments of pthread_create()
- ↳ arg1 :- addr of thread
- ↳ arg2 :- used to set attributes like stack-size, scheduling policy, priority. NULL if default
- ↳ arg3 :- f^n that the thread needs to execute.
- ↳ arg4 :- argument for thread f^n. If more than one argument, declare a structure & pass its address.

- SIGNALS : $ kill -l
  - ↓ method for IPC & used for everything in n/w.
  - → 64 signals.
  - → 0-31 NRT signals
  - → 33-63 Real time
  - → each signal starts with macro SIGXXX
  - → each signal may also specify its integer # eg: kill -8 or man 7 signal
  - → when a signal is sent to a process, kernel stops the execution & forces it to call signal handler.
  - → when a process executes signal handler if some other signal arrives, the new signal is blocked until the handler returns
  - → signal → generally refers software generated signal.



↳ interrupt.




↳ signal() system call :-

- when a signal occurs, a process should
  - catch the signal
  - ignore the signal (SIG_IGN)
  - set a default action SIG_DFL

- Two signals that cannot be caught or ignored
  - SIGSTOP
  - SIGKILL

(signals are processor architecture dependant)

- signal system call is used to catch, ignore or set the default action of a specified signal.

```
int signal(int signum, (void*)handler);
```

- signum :- signal #
- handler :- user defined signal handler.

↳ sigaction() has a lot of control over a given signal.

```
int sigaction(int signum, const struct
        sigaction *act, struct sigaction
        *oldact)
```

- signum = specified signal
- act :- set new action of signum
- oldact :- used to store previous action usually NULL

- kill system call is used to send a given sig... to a specific process.
  int kill(pid_t process_id, int signal_num...
  - "pid" is +ve, signal is sent to a partic... process.
  - if negative, signal is sent to the process whose groupid matches to abso... value of pid.

- MEMORY MANAGEMENT:-
  - Virtual m/m :- Programmer need not worry about size of RAM (large address spac...
    - virtual m/m is much larger than physical memory in a system
    - process supports shared virtual m/m i.e. more than one process can share a shared page.
    - uses paging.
  - swap space :-
    if (RAM Size > 2GB)
      swapspace = Ramsize + 2GB;
    else
      swapspace = Ramspace * 2;

To see virtual memory stats :-
$ vmstat 1
↑ for every one second

or vmstat 1 1
↳ only one such instance

- to see CPU stat :-
$ mpstat 1
↳ all CPU states for every one second.
$ mpstat -P 1
↳ 1st CPU states.

static allocation → internal fragmentation.
dynamic allocation → external fragmentation.
avoid fragmentation → Thrashing - overhead.

- Page Table (PT) - Translate a process virtual address to physical address since processor use only virtual address space.
↳ size of PT is normally size of a page.
↳ holds information about
↳ page valid or not
↳ Page Frame Number (PFN)
↳ access control information.

↳ swapping :-
↳ swap space in HDD partition.
↳ page waiting for certain event to occur, (stop & swap it
↳ use physical m/m space efficiently.
↳ if no space for physical m/m, swap LRU pages into swap space.

↳ Demand paging :- don't load all the pages of a process into m/m. Load necessary pages initially & if required page is

not found, generate page fault. Then the page fault handler brings the corresponding page into m/m

↳ page fault :-

- major (not loaded from HDD to memory)
- minor (loaded in m/m, swaps in virtual m/m)

→ handled by page fault service routine. (PFSR)

↳ KERNEL DATA STRUCTURE :-

↳ Virtual m/m represented by mm_struct data structure.

↳ It has ~~two~~ pointers to vm_area_struct data structure.

- created when an exe image is mapped with process virtual address
- has starting & end points of virtual memory
- represents process image like text, data & stack portion.
- has control access info.

Camlin Page 97
Date 14/09/18

#include <mm_types.h>

struct mm_struct {
    struct vm_area_struct *mmap;
    .
    .
    .
};

mmap → lists of virtual m/m addresses

struct vm_area_struct {
/* first cache line has the info for VMA tree walking */
    unsigned long vm_start;
    unsigned long vm_end;
/* linked lists of VM areas per task, sorted by address */
    struct vm_area_struct *vm_prev, *vm_next;
    .
    .
    .
};

↳ to get any information about Kernel module, use modinfo command.
eg: $modinfo bluetooth.

where, module name can be given by lsmod.

↳ Kernel level functions have 'k' prefixed.
eg. printk() } Kernel level
kmalloc() } functions.

↳ to add a Kernel module (ours), we need to create a Kernel object file ".ko" extension file, using

some make file ("·mk")

- insmod → insert module
- rmod → remove module

↳ to see execution status of a kernel module, use command
$ dmesg .

↳ we can see all kernel symbols (currently linked) using
$ cat /proc/kallsyms.

↳ free command :-
$ free → displays RAM size in byte
$ free -m → displays RAM size in MB
$ free -g → displays RAM size in GB

↳ Walk through program execution