OS Architecture:-

OS components:-
- system utils
- libraries
- kernel
- drivers

kernel:- core of the OS
resides in the memory all the time
provides various services to apps in form of system calls
interface to minimal h/w only,eg:- CPU, memory

drivers:- can access additional hardware
plugins to kernel

libraries:- friendly access to system calls
enables platform independent code

Dual mode operations of modern OS:-

user mode -- normal mode execution
-- Apps, libs, utils

kernel mode -- supervisor mode
-- kernel and drivers

system calls are services from kernel, defined by kernel mode components and invoked by user mode components

memory partitioning (address space):-

reserved space for kernel mode components -- kernel space
unreserved space for user mode components -- user space

user mode execution can access user space only
kernel mode execution can access entire memory,i.e both

Types of kernel:-
A.Monolithic kernel
    all services are deployed as single freezed component running in same execution mode
    (-) not flexible for upgradations
    (+) no overhead of communication among subsystems
B.Micro kernel
    minimal services are in core kernel, additional services are in different layers,may run in other exec modes
    (+) flexible, particular subsystem can be replaced
    (-) overhead of communication among subsystems
    eg:- QNX,some RTOS

C.Modular kernel

most of the services are offered by deployed kernel
any service or feature can be added or removed
dynamically in the form of modules
core image,all loaded modules execute in same mode
eg:- Linux, modern OS

Unix is monolithic

modules -- static and dynamic modules

System calls:-
==> services offered by kernel to users
==> Defined in kernel space, execute in kernel mode
==> Invoked from user space
==> System calls can't be invoked by name(address)
unlike app functions or library calls
==> Identified and invoked by unique no.

Invoking system call from user space:-
0.save user mode context
1.identify the unique no.for desired system call and
store in suitable register,typically general purpose..accumulator
2.store arguments to system calls in available general purpose
registers..if an argument is not compatible with regs,pass base
address only
if no.of arguments exceeds available regs encapsulate
them and pass the base address
3.execute trap instruction to enter kernel mode

printf("abcdxyz");
==> write(1, str, 7);

eg:- in x86
system call --> EAX
args --> EBX, ECX, EDX, ESI, EDI, EBP
retval --> EAX

Application Binary Interface(ABI) -- conventions of register
usage for system call invocation, trap instruction

write(fd,str,len) ==> system call wrappers
                       (or) system call API
                   ==> prototype in unistd.h
                   ==> defined in std C library
                       (libc.a or libc.so provided glibc)

POSIX standard compliance -- Unix/Linux interoperability
Portable Operating System Interface eXchange
commands, lib calls, system calls

```
printf("hello world");
        char str[]="hello world";
        len=11;
        fd=1;
        write(fd,str,len);
```

library calls -- some processing logic, may invoke system calls
system call wrappers -- quickly invoke system calls as per ABI

library calls vs system calls:-

==> user friendly

==> portable across OS

==> efficiency

Interrupts:-

Asynchronous events

Generated by i/o devices, timers, failures, s/w interrupt(syscall)

Interrupt Request -- IRQ lines via interrupt controller to CPU

Interrupt Service Routine(ISR)/Interrupt Handler

ISR Table -- Interrupt Vector

Ignoring interrupting or delaying is not desirable, may lead to inconsistency,unhealthy environment

Interrupts must be serviced with utmost priority

ISRs must execute for short duration without blocking call

Maskable vs non maskable interrupts

Terminology:-

terminal (gnome-terminal,konsole,mate-terminal,xterm)

shell -- command line interpreter

-- take command names,arguments,options as input

-- execute them

-- give the output to user

prompt