Process Management:-

Process:-

- program loaded in memory for execution
- active entity
- can consume system resources -- CPU, memory, i/o
- resource usage with multiplexing
- for eg:-
  - cpu -- time multiplexing
  - memory -- space multiplexing

program/binary:-

- passive entity, stored in disk

program on disk:-

- code + idata

process in memory:-

- code + idata + udata + stack + rodata
- some heap association -- heap blocks
- address space of a process -- user space

Kernel support:-

- process id(pid)
- parent process id(ppid)

process table/process list -- an entry for each process(slot)
process control block(PCB)/process descriptor(pd)
        -- data structure holding attributes of a process
process -- user process
----
attributes of a process:-

process id(pid)
parent process id(ppid)
process name(cmd)

state
policy
priority
time left

address space descriptors (memory description)
fs description,file descriptors
i/o description

accounting info -- ownership
termination status
reg save area?? for context -- context area

Process state, process life cycle:-

ready -- waiting for CPU, eligible for execution by scheduler

running -- consuming CPU cycles
-- instructions getting executed by CPU
blocked -- waiting for a resource other than CPU
terminated -- assigned code got completed
return/exit got executed
state transitions:-
newly created process -- ready
scheduled -- ready to running
blocking -- running to blocked
i/o req, sleep API, resource locking
unblocking -- blocked to ready
i/o done, sleep over, resource unlocked
preemption -- running to ready
h/w interrupt, high prio process, timeout
termination -- running to terminated

every process maintains unique address space
i.e independent stack for each process
context of process is typically saved on top of stack

terminated:-

I.normal -- by exit or return
a) success
b) failure -- some conditions not met
II.abnormal termination
==> due to exceptions,signals

Ready queue/Run queue

Context switching:-

Context saving -- saving the register snap shot in reg save area when a process is leaving the CPU

Context loading/restore -- loading the context of a process from reg save area to CPU regs when scheduled again

saving + loading ==> context switching

cpu cycles spent on context switching are not accounted on behalf of any process,but context switching is essential to acheive multitasking..minimize no.of context switchings

init is the origin of unix/linux process hierarchy
pid of init is 1
created at end of booting process based on init scripts

Commands:-

- ps
- ps -el
- ps aux
- ps -e -o pid,ppid,stat,cmd
- pstree

ps states:-

- S -- blocked
- R -- ready/running
- S+/R+ -- foreground process, terminal focus
- S/R -- background process, detached from stdin

system calls, lib APIs:-

- fork
- waitpid
- exit
- execl, execlp etc.
- getpid
- getppid
- sleep -- lib call

fork:-

- creates a new process known as child
- current process is known as parent
- assign new pid,PCB to child process by locating
  - free slot in process table,throw error(return -ve)
  - if process table is full
- address space is duplicated from parent to child
- returns zero to child, +ve value to parent
- child resumes from next stmt of fork(context)
- parent & child can execute concurrently in
  - their own independent address space

reparenting/adoption by init:-

- if parent terminated before child, existing child will be reparented to init
- ppid of child becomes 1

waitpid:-

- block the parent process till the completion of child
- collect the exit status of child

fork returns a +ve value to parent which is pid of created child waitpid params:-

1st param:- pid of target child, -1 means any one child
2nd param:- address of a variable to collect the status
3rd param:- flags, 0 means default

collecting status from child:-

```
        waitpid(-1,&status,0); //wait(&status);
                printf("parent--child exit status=%d\n",
                                        WEXITSTATUS(s1));  // sys/wait.h
```

```
ret=fork();
if(ret==0)
{
        printf("child--welcome,pid=%d,ppid=%d\n",getpid(),getppid());
        k=execl("/bin/ls", "ls", NULL);
        if(k<0)
        {
                perror("execl");
                exit(1);
        }
        printf("child--thank you\n");    //redundant if execl is sucessful
}
else
        //parent code
```

execl overwrite child address space with code and data
of new program

on success of execl child discards duplicated address
space and attach to new address space

shell of your own:-

1.read command name as string
2.create a child process using fork
3.launch the requested command in child using execl/execlp
4.parent blocks till command execution using waitpid
parent may print status of child exit
5.read one more command ... in a loop

```
execl("/bin/ls","ls",NULL);
execlp("ls","ls",NULL);
```

```
cal 10 2018
        execl("/usr/bin/cal","cal","10","2018",NULL);
        execlp("cal","cal","10","2018",NULL);
```

```
        ./t.out abcd 10 xyz
        execl("./t.out","t.out","abcd","10","xyz",NULL);
```