Process Management:-

nice	chdir	wait	fork
setgid	setpgrp	getpgrp	getppid
getgid	getrgid	getuid	getruid

exec exit signals → alarm
setuid → signal
→ kill
→ pause.

इन सभी methods/fun को use करेंगे.

Process:- is a Program in execution.
Multiple instances of the same program are different processes.

Process memory image contains 3 components (context, data, Program (code)) in 2 different address Spaces:-

1. User Address Space:-
   - → An executable program code.
   - → Associated data needed by program.
2. Kernel Address Space:-
   - → Execution context needed by OS to manage the process (PID, CPU registers, CPU time, stack, open files, signals).

DATE / / PAGE triangel

Process Control Block (PCB):-

→ દરેક Process માટે PCB maintain કરવામાં આવે છે. System માં process table array or linked list માં.
→ Context / Kernel address space માં PCB હોવાથી stack માં જાય છે.
→ PCB contains:-
Process ID, Parent Process ID
Process state
CPU state : CPU register contents, PSW (Process status Word).
priority and other scheduling info.
Pointers to different memory areas.
Open file information.
Signals and signal handler info.
Various accounting info like CPU time etc.
Many other OS specific fields.

Operations on Process:-

1. Process Creation:-
   → Data structures like PCB ને initialize અને set up કરવું.
   → Initial resources allocate કરવા અને જરૂર પડવી તો initialize કરવા.
   → Ready Queue માં add કરવી Process ને.
   ↳ que of processes which are ready to run.
2. Process scheduling:- CPU allot કરવામાં આવે process ને અને process run કરવાનું start કરે.

Process Termination:-
→ Process ને remove કરી દેવો.
→ Resources ને reclaim કરવો.
→ exit status & કોઈ data હોય
તો તે parent process ને pass કરવો.
→ signal હોય તો Parent process ને
interruption: inform કરવો

Events that lead to Process Creation.

1. System boot :- bg process or daemons
start થાય. ↓
eg. init bg મા ચાલે અને
interactive ના હોય.
Sys ને supervise કરે તેવા process
ને functionality provide કરે.

2. User Request :- CLI shell પર cmd run
કરો or App run કરવી dbl
click કરે.

3. A process can spawn another
ie. child process using fork() eg.
server can create new process
for each client req.

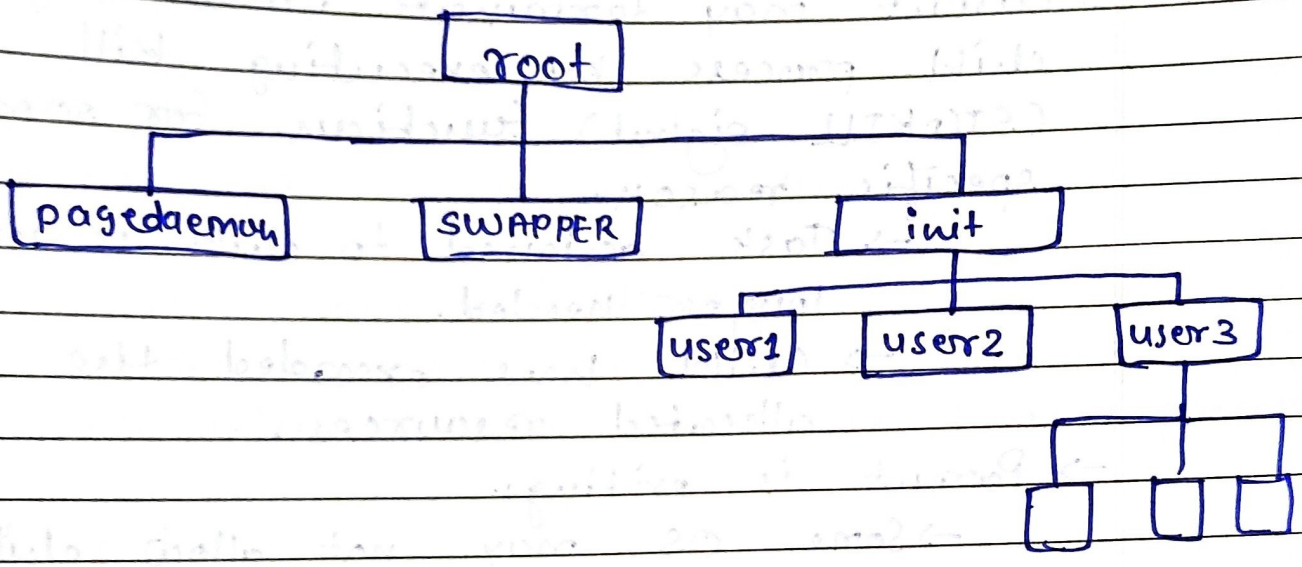
DATE / / PAGE triangel

Stages of Linux Boot Process.

1. BIOS :- Basic Input Output system
   -> performs system integrity checks, search boot loader on cd-rom or disk, and executes MBR boot loader.
2. MBR :- Master Boot Record :-
   -> Available in 1st sector of bootable disk, executes GRUB boot loader.
3. GRUB :- GRand Unified Bootloader.
   -> multiple Kernel imgs ಇದ್ದಾಗ GRUB ನಲ್ಲಿ use ಮಾಡಿಕೊಂಡು selected Kernel ನ execute ಮಾಡಿ ಕೊಡುತ್ತದೆ.
4. Kernel :- root file system ನ mount ಮಾಡಿ. ನಂತರ init ಅನ್ನು execute ಮಾಡಿ.
   ↳ first prg to run.
5. init :- executes run level progs depending on runlevel.
6. Run level -> Runlevel programs executed from /etc/rc.d/rc*.d
   0 -> System halt; no activity, the system can be safely powered down.
   1 -> Single user, rarely used.
   2 -> Multiple users, no NFS (network filesystem); also used rarely.
   IMP. <- 3 -> Multiple users, command line ~~Interface~~ the std run lvl for most linux based server hardware.
   4 -> user definable.

5 → Multiple users, GUI, std runlvl for most Linux-based desktop systems.
6 → Reboot; used when restarting the system.

Process Tree on UNIX System:-



Pagedaemon → responsible for virtual page management.

Swapper → scheduler, kernel daemon → moves whole processes b/w secondary main memory and virtual memory. (kernel Adel space. as a part of VMS virtual memory system.)

→ first process after the kernel is loaded

init → root of all user processes.
↳ user Adel space.

Events leading to Process termination:-

→ Process executes last statement and asks OS to terminate it using exit() function.
→ Process encounters fatal error like divide by zero, I/O error, memory allocation errors etc.
→ Parent may terminate execution of child process by executing kill (SIGKILL signal) function for some specific reason.
  → Task assigned to child is no longer needed.
  → Child has exceeded the allocated resources.
→ Parent is exiting:-
  → Some OS may not allow child to continue if parent terminates.

Process Relation b/w Parent and child ~~Resources~~ Processes.

→ Resource sharing possibilities
  → Parent and children share all resources.
  → Children share subset of parents resources.
  → Parent and child share no resources.

triangel

→ Execution Possibilities:
  → Parent and children execute concurrently.
  → Parent waits until children terminate.
→ Memory address space possibilities:
  → Address space of child duplicate of parent.
  → Child has a new program loaded into it.

Process Management System Call:

1. pid_t getpid(void);
   → get process ID.
   → Returns process ID of calling process.

2. pid_t getppid(void);
   → get parent's process ID.
   → returns parent process ID of calling process.

IMP

3. pid_t fork(void);
   → Create child process by duplicating memory of parent process.
   → child gets copy of data space etc. Now uses Copy-On-Write (COW)
   → Returns 0 in child, process ID of child in parent, -1 on error.

child process હવે parent process ની જ memory duplicate કરે તો method પહેલા તો વાર return કરાવી દેશે child માં અને ત્યાર બાદ વાર parent માં

[IMP]

4. pid_t vfork (void);
   - → Now Absolete.
   - → Earlier fork() used to copy a complete memory of parent to child so vfork was used as optimized method.
   - → returns 0 in child, processID of child in parent, -1 on error.

5. void exit(int);
   - → terminated the process.
   - → Parent process will receive the success value passed as int.

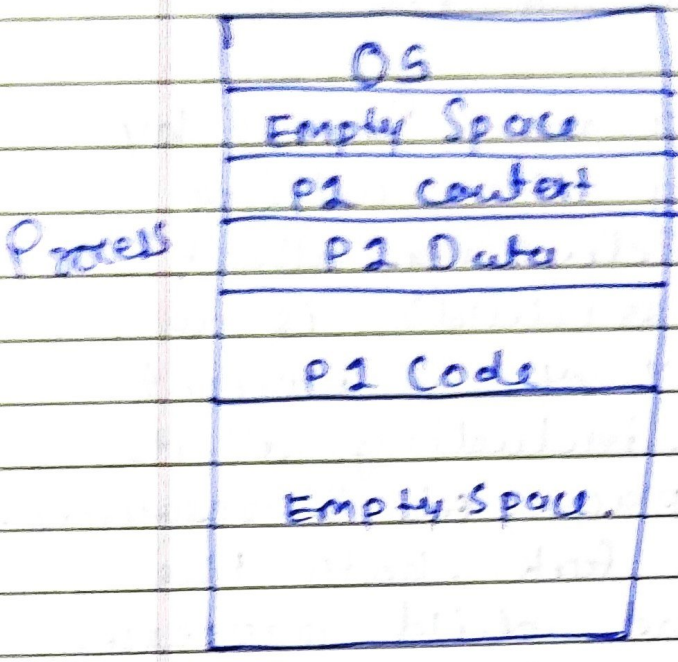
6. pid_t wait (int *statloc);
   - → wait for child process to terminate.
   - → Returns PID of child that terminated and statloc indicates returned status.

7. int exec (const char pathname, const char *arg0, ...).
   - → replaces current process memory with new process to be executed in pathname.
   - → return -1 on error and no return on success.

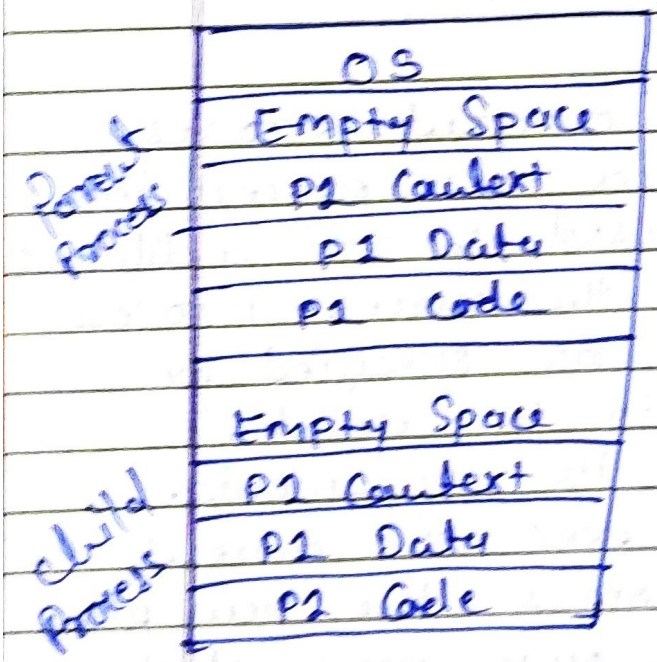
→ Difference b/w fork() & exec();

1. fork() → create a new Process by duplicating the calling Process. Executing
→ new Process (child) is an exact copy of the parent process, including all its memory space, fds, & resources.
→ After the fork, both the parent and child processes run independently of eachother, with their own unique process IDs.

2. exec() → replace the current process img with a new process img.
→ typically used to run a different program in the current process, which can be specified by the path to the executable file and its arguements.
→ After the exec, the current process becomes the new prg, with its own new set of memory space, fds, and other system resources.

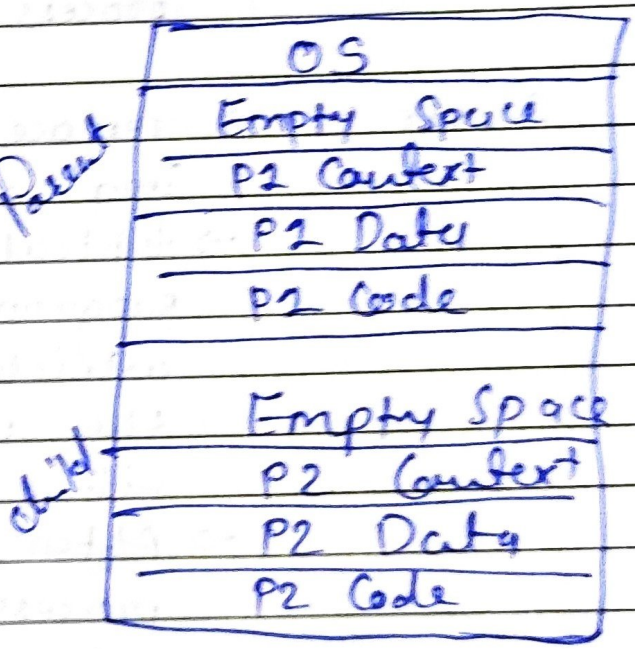


Pid = fork()



clone the child Process.

exec(prog-name)



Replace child img.

triangel

Process States: changes as it executes.

Idle → fork() will create and go to runnable state

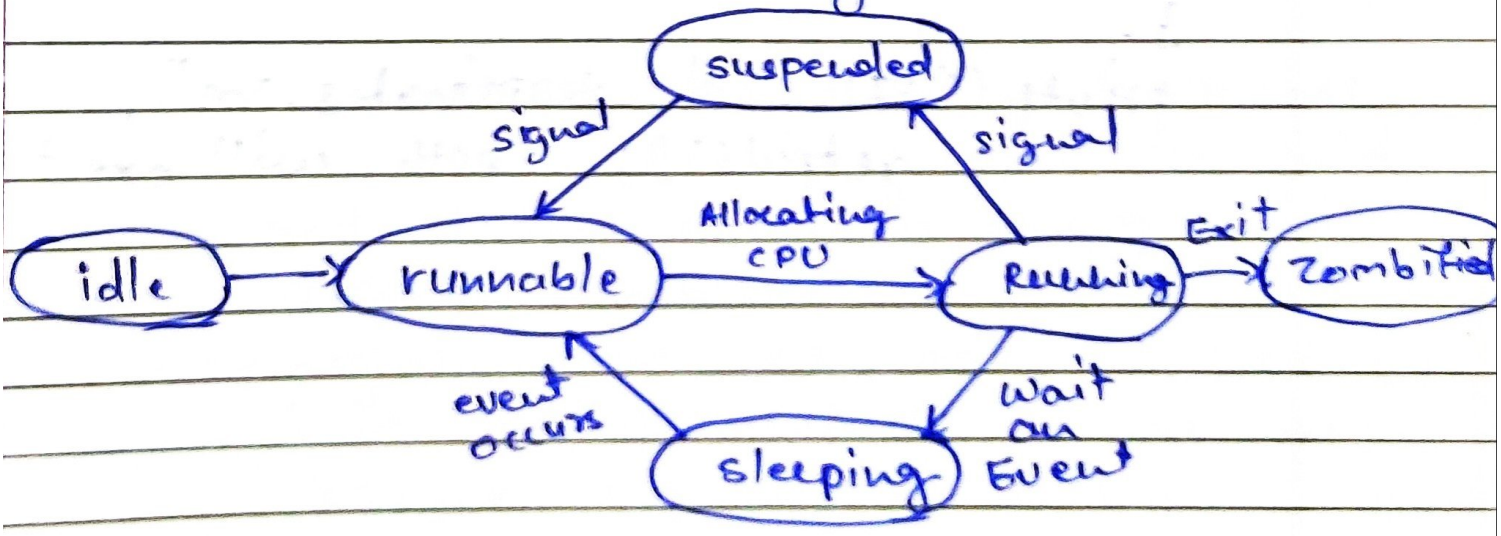
runnable → waiting for CPU to start running.

running → process is current running executing instructions.

sleeping → process is waiting for an event to occur. eg. If process has executed read() system calls, it will go in sleep until I/O req is complete.

suspended → process has been "frozen" by signal such as SIGSTOP, it will resume when SIG CONT signal received. eg. Ctrl+z suspend all the processes of foreground

zombified → process has terminated but job. has not yet returned its exit code to its parent. The process remains in zombie state until parent accepts return code using wait() sys call.



→ System Calls fork(), getpid(), getppid().

```
main()
{
    int pid;
printf("I'm the org process with PID %d
        and PPID %d", getpid(), getppid());
pid = fork(); // Duplicate. child and parent
                Continue from here.
if(pid!=0) // pid !=0 must be parent
{
    printf("I'm the org process with
        PID %d and PPID %d", getpid(),
        getppid());
    printf(" my child's pid is %d", pid);
}
else // child.
{
    printf("I'm the child process with
        PID %d and PPID %d.\n",
        getpid(), getppid());
}.
printf("PID %d terminates.\n",
        getpid()); // both will exe this.
```

Parent and Child Process Variable Scope.

-> before fork there is only one process so only one copy of variable.

-> Immediately after fork, two copies of the variables are created. (one for parent and other for child). After fork any changes to the variable is done in the local copy of the respective process, hence the other process variable value will be different.

How many total processes will be created?

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

-> Total 8 process. 2^n

-> Orphan Process:- If parent process does not wait for child and it first terminates leaving child process orphan.

-> orphan processes are adopted by init process which started the parent (ie. parent of parent).

Codes Chatgpt [illegible]

DATE / / PAGE

Call wait() to avoid orphans.

zombie process:- process that finished running, but it's parent process has not yet read its exit status.
-> If the parent process does not call wait() and waitpid() to read exit status, child become zombie and if parent dies it become orphan.

Race Condition:- a type of programming error that can occur when multiple processes access a shared resource or data structure con-currently, and the outcome of the program depends on the order in which the processes execute.
eg. If two processes simultaneously try to update the same var or file, the result may depend on the order in which the update occur.
if the processes or threads does not synchronize their access to the shared resource, result may lead to bugs.

triangel

Additional status info from wait()

childpid = wait(&wstatus);
→ returns the exit status from child which can further be inspected using these macros.

WIFEXITED(wstatus) → returns true if child terminated normally

WEXITSTATUS(wstatus) → returns exit status (least significant 8 bits)

WIFSIGNALED(wstatus) → returns true if child process was terminated by a signal.

WTERMSIG(wstatus) → returns the no. of signal.

WCOREDUMP(wstatus) → returns true if child produced a core dump.

WIFSTOPPED(wstatus) → returns true if child was stopped by a signal.

WSTOPSIG(wstatus) → returns the signal no. which caused child to stop.

WIFCONTINUED(wstatus) → returns true if child was resumed with SIGCONT signal

DATE / / PAGE triangel

exec() family of system calls:-

When a process executes an "exec()" system call, its PID and PPID no.s stay the same - only the code that the process is executing changes.

```
int execl(const char * path, const char*
          arg0, const char* argn, NULL);
int execv(const char* path, const char*
          argv[]);
int execlp(const char* path, const char*
          arg0, const char* argn, NULL);
int execvp(const char* path, const
          char * argv[]);
```

These calls replaces the calling process's code, data and stack with those of the executable whose pathname is stored in path.

execlp() & execvp() :-
use the $PATH environment var to find path.
-> If exe not found -> returns -1.
otherwise replace code and starts to execute new code.

execl() & execlp() :-
→ invoke the exe with the st
arguments pointed to by arg1
through argn.
→ arg0 must be the name of
the executable file itself.
and ~~execlp~~ the list of
arguments must be null
terminated.

execv() & execvp() :-
→ invoke the executable with
the string arguments pointed
to by argv[1] to argv[n],
where argv[n+1] is null.
→ argument be ~~point~~ the name
of the executable file
itself.

→ Master-Slave Implementation :-

Master accepts n values from CLI,
Set of 2 values to be passed to each
child process i.e. argv[1] & argv[2]
pass to child 1, 3-4 to 2 so on.
Forks as many child processes as
required i.e. n/2.
Reads the value returned using exit()
call using WEXITSTATUS macro.

execl() & execlp() :-

→ invoke the exe with the string arguements pointed to by arg1 through argn.

→ arg0 must be the name of the executable file itself and the list of arguements must be null terminated.

execv() & execvp() :-

→ invoke the executable with the string arguements pointed to by argv[1] to argv[n], where argv[n+1] is null.

→ argv[0] must be the name of the executable file itself.

> Master-Slave Implementation :-

Master accepts n values from CLI.
Set of 2 values to be passed to each child process i.e. argv[1] & argv[2] pass to child 1, 3-4 to 2 so on.
Forks as many child processes as required ie. n/2.
Reads the value returned using exit() call using WEXITSTATUS macro.

child process accepts the 2 CLI args performs required processing. Returns the result back as exit status using exit() system call.

system()
→ int system (const char* command);
→ implemented using fork(), exec() and waitpid()
→ used to execute the command passed as parameter.
eg. system ("ls -ltr");
runs ls -ltr command.

→ How to get additional information about process which is running.

when a process starts it creates a directory with process ID under /proc for per process info.

1. check PID using ps command.
proc $ ps -aux

2. Now change the dir to /proc/pid
and list the content ls -l. 2125 etc.

Environment var
gir all lecq ried

most files with its content is described here.

cmdline → commandline Arguments.
cpu → current and last cpu in which it was executed.
cwd → Link to the current working directory.
environ → values of environment variables.
exe → Link to the executable of this process.
fd → Directory which contains all the file descriptors.
mem → memory held by this process.
stat → Process status.
mountinfo → info about mounts.
comm & /task/<tid>/comm → Common content info about the parent task and each of the child task eg. on web browser there may be multiple windows each of which is child task.
/fd info/<fd> → info about opened file