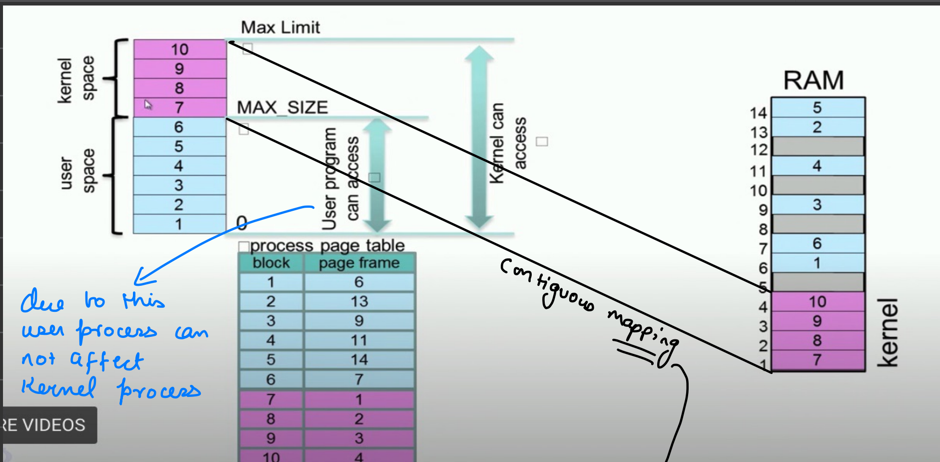
Process Creation

Process → Program under execution.

Kernel And User Space



due to this user process can not affect Kernel process

Contiguous mapping

This makes virtual add to physical add conversion easy & vice-a-versa And same kernel mapping is present for all process leads to same page frames in RAM for all process.

Kernel Data About A Process

Corresponding to each process, the kernel keeps some metadata.

- ↳ PCB (process control block)
- ↳ Kernel stack for each process to store context
- ↳ Page tables for user process.

Process Stacks

Each process has 2 stacks –

- User space stack → normal func^n call stack
  - Used when executing user code
- Kernel space stack → stores context of process
  - Used when kernel code in the context of a process (for eg:- during system calls)

★ Advantages of Kernel Stack –

Kernel execute even if user stack is corrupted. Attacks that target the stack, such as buffer overflow will not affect kernel.

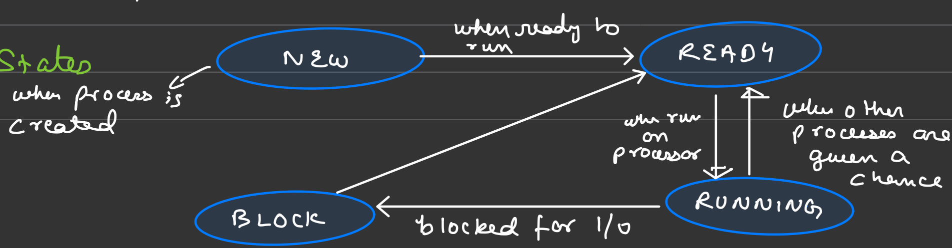
Summary Of Entries In PCB

Size of process memory
List of files opened
current working directory
Kernel stack ptr, process ID
Page directory ptr and executable name

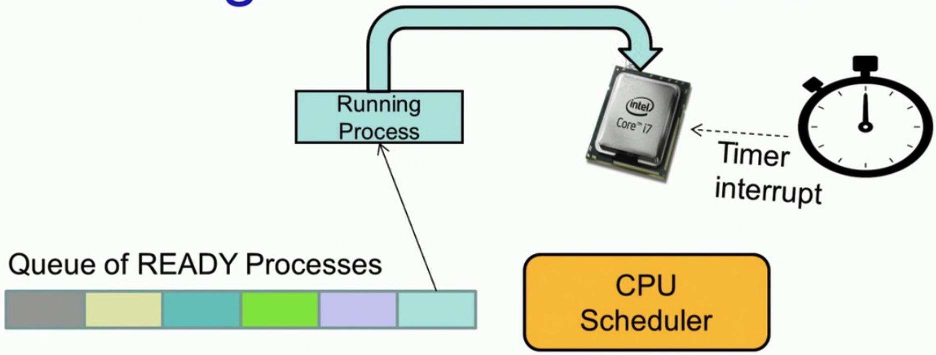
PID

- Process Identifier
- No. incremented sequentially
- Unique PID for all process.

Process States



Scheduling Runnable Processes



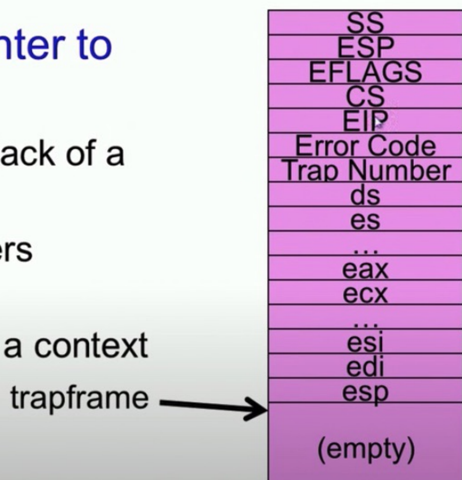
Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O

Scheduler picks another process from the ready queue

Performs a context switch

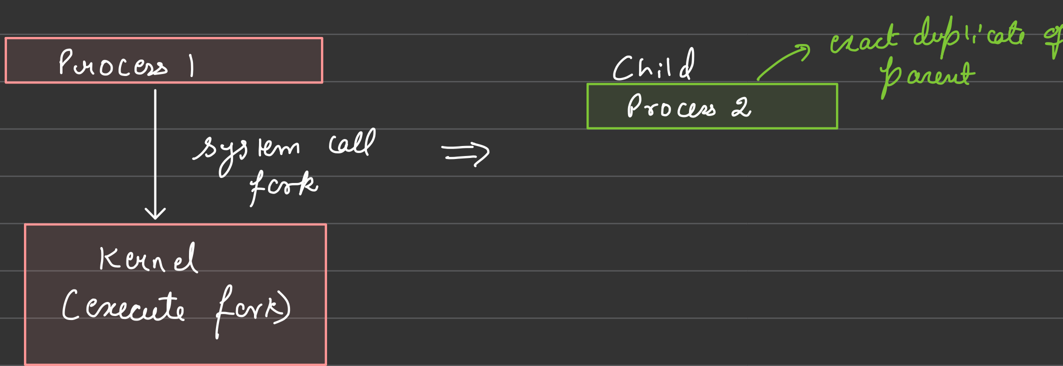
Entries in PCB

- Pointer to trapframe and pointer to context
  - Present as part of the kernel stack of a process.
  - Contains the state of all registers corresponding to the process
  - Used to restart a process after a context switch

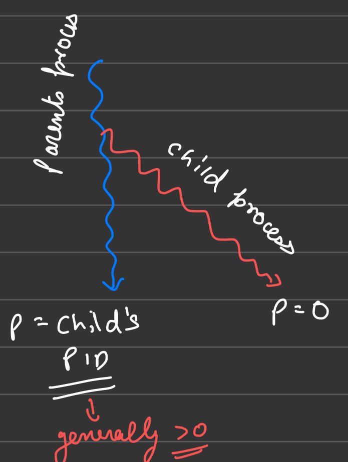


Creating A process By Cloning

- Cloning
  - Child process is an exact replica of the parent
  - fork system call is used.



Creating a process by cloning

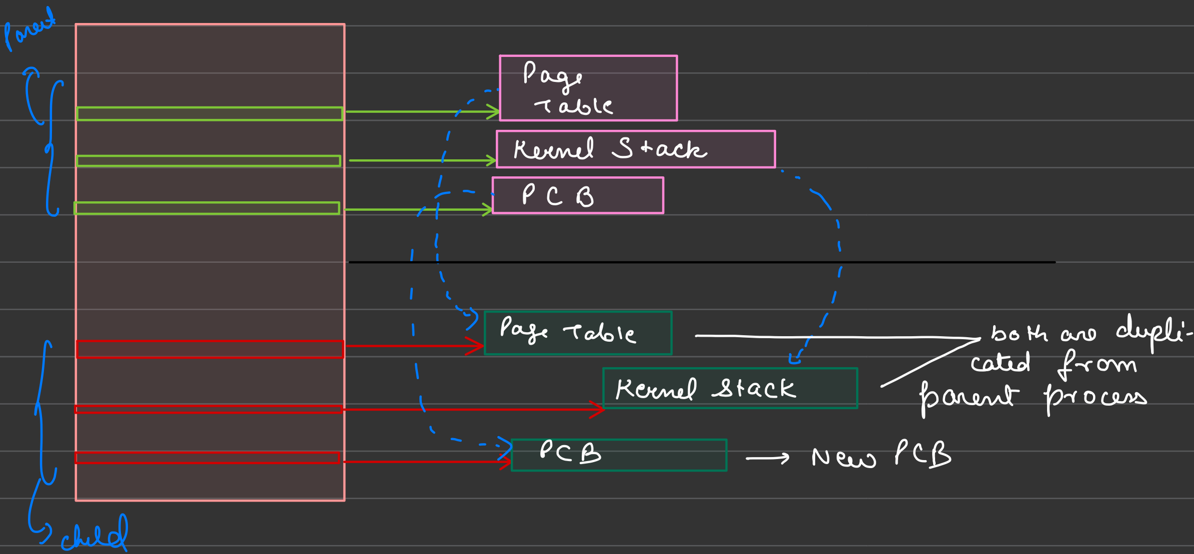


can have -1, 0, or >1 value

```
int p;

p = fork();
if (p > 0){
    printf("Parent : child PID = %d", p);
    p = wait();
    printf("Parent : child %d exited\n", p);
} else if (p == 0){
   printf("In child process");
   exit(0);
} else{
    printf("Error\n ");
}
```

fork: from an OS perspective



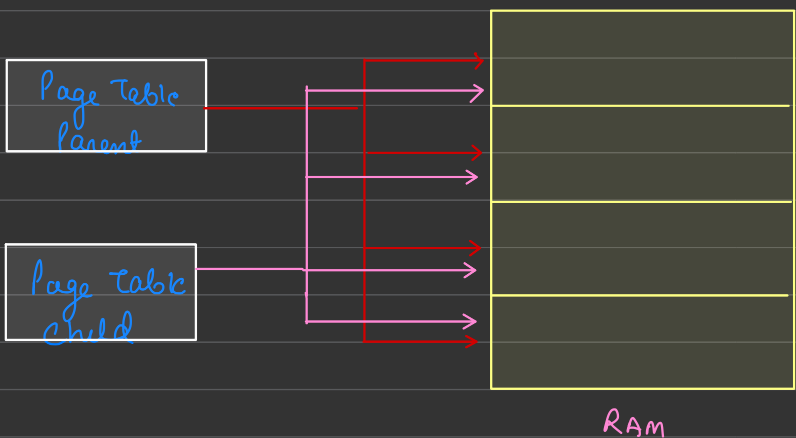
→ find an unused PID
→ Set state of child process to NEW → process being created not ready to run
→ Set pointers to newly formed
 – pagetable
 – Kernel Stack
 – Trap frame & context within new kernel Stack
→ Copy info like size, files opened, cwd from parent
→ Set state to READY before returning

NOTE → Child process will be handled as any other process by ready queue i.e. a new space will be given to child process.

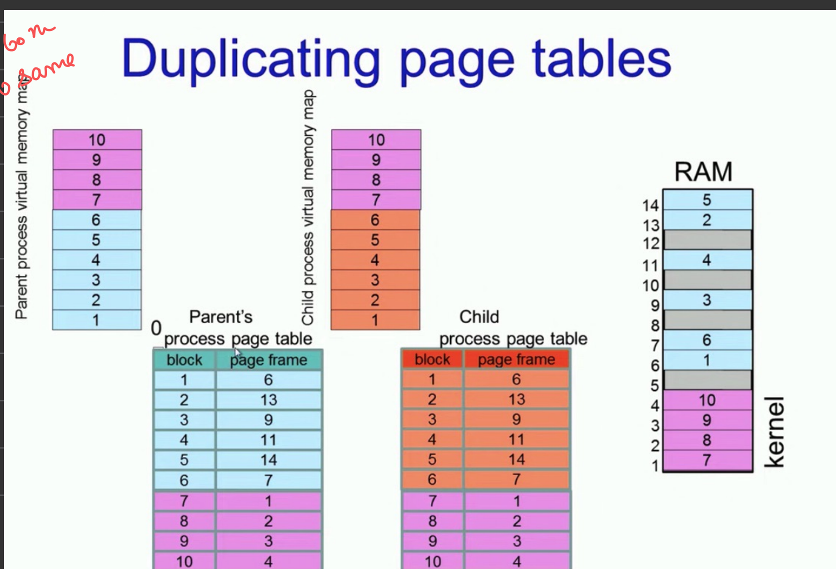
Copying Page tables

when a child process is created it makes a duplicate page table of parent's page table.

Parent and child page tables points to same physical memory



Blocks from 6 on are pointing to same page frame



Example

Output
child : 23
parent : 23

```
int i=23, pid;
pid = fork();
if (pid > 0){
        sleep(1);
        printf("parent : %d\n", i);
        wait();
} else{

        printf("child : %d\n", i);
}
```

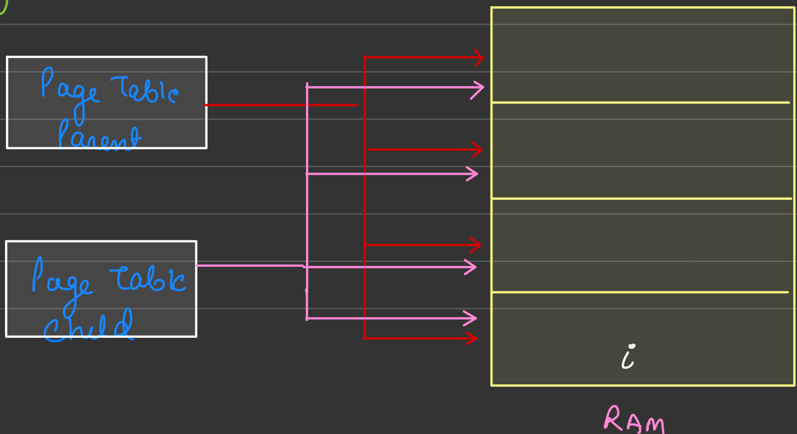
Now let's take example given below. In parent process we have added a sleep so child has sufficient time to update i to i+1. If the page frames for both parent and child are same then it seems that both the parent & child will give output as 24 because they will point to same memory location. But

```
int i=23, pid;
pid = fork();
if (pid > 0){
        sleep(1);
        printf("parent : %d\n", i);
        wait();
} else{
        i = i + 1;
        printf("child : %d\n", i);
}
```

Output is → child : 24
parent : 23 → How????

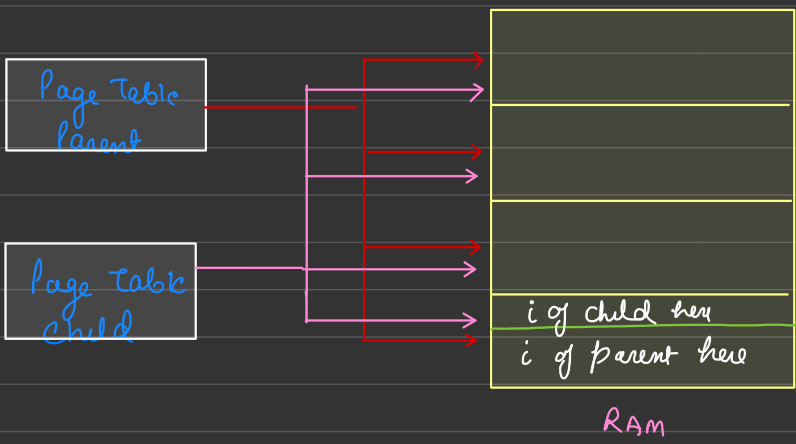
This happens due to a process called COW.

Copy On Write (COW)



initially both pts to same page but when child update then →

- All parents pages are initially marked as shared
- When data in any of the shared pages change, OS intercepts and make copy of page.
- Thus, parent and child will have different copies of this page (all other pages remain same.



Executing A New program

Its a 2 step process.

first fork and then exec

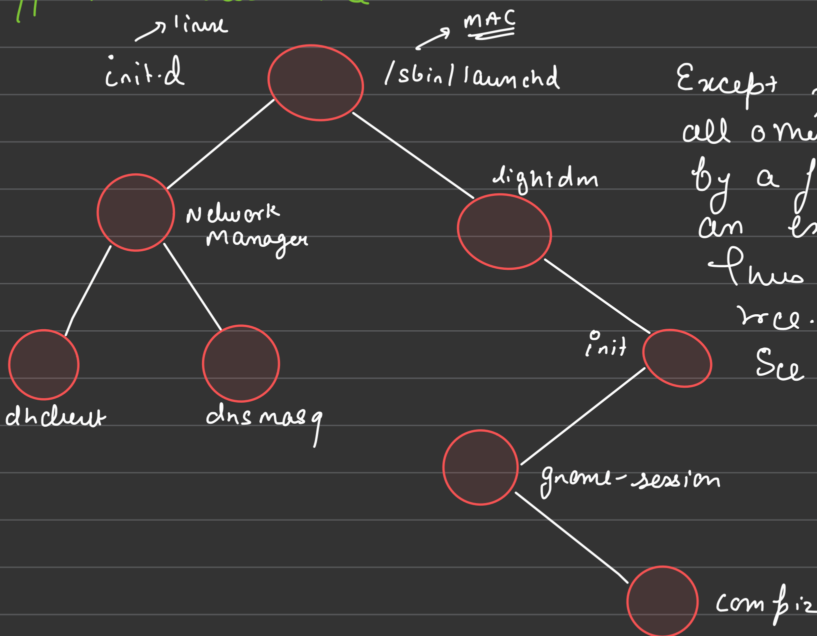
- Exec system call
  - find on hdd the location of the 'a.out' executable
  - Load on demand the pages reqd to execute a.out.

```
int pid;

pid = fork();
if (pid > 0){
    pid = wait();
} else{
   execlp("./a.out", "", NULL);
   exit(0);
```

Advantage of COW → Big advantage for exec. Common code (for ex shared libraries) would continue to be shared. Ex printf

The Process Tree



Except for the first process all other processes are created by a fork followed by an exec.
Thus forming an process tree.
See command PSTREE

The first Process

- UNIX: /sbin/init

Unlike the others, this is created by the kernel during boot.
Called as Super Parent
Responsible for forking all other processes.
Typically starts several scripts present in /etc/init.d in Linux.

Exit System Call → called as voluntary termination

Called in child process
Results in the process terminating
The return status (0 here), is passed on the parent

Involuntary Termination.

kill (pid, signal)
- Signal can be sent by another process or by OS.
- pid is for the process to be killed.
- signal is a sign that the process needs to be killed.
  Ex SIGQUIT (ctrl + \), SIGINT (ctrl + C)

Wait System Call -
- Called in the parent process
- Parent goes to block state
  - Until one of it's children exists
  - If no children executing then returned -1
- Return status of child can be collected by wait (& status)
  → pointer when OS puts exit status of child process.

Zombies
- When a process terminates it becomes a zombie (or defunct process)
  - PCB in OS still exists even though the program is no longer executing so that the parent process can read the child's exit status (through wait call)

- When parent read status
  - zombie entries removed from OS .... process reaped
- Suppose parent doesn't read status
  - Zombie will continue to exist infinitely ..... a resource leak.
  These are later removed by reaper process.

Orphans

When parent process terminates before it's child
Adopted by first process (/sbin/init)

There are 2 types of orphans –

1. Unintentional
   - When parent crashes

2. Intentional
   - Process becomes detached from user session & runs in the background.
   - Called as daemons, used to run background service
   - See "nohup"

Exit() internals –

- init, the first process can never exit
- for all other process on exit –
  - Decrement the usage count of all open files. Close file if usage count is 0.
  - wakeup parent
    → if parent state is sleeping, make it runnable cause parent may be sleeping due to wait.
  - Make init adop the children
  - Set process as zombie

NOTE → page directory, kernel stack are not de-allocated here. They are deallocated by parent, allowing parent to debug the crashed child.

Wait Internals