# Process Creation

Fork
- Creates a new process
- Child gets a copy of the address space of the parent ( not the name)
- copy of the variables, and the code
- Exact replica of the parent

Exec
- To ensure that the child and the parent are doing different things
- can Destroy the prev process image ,and load a new image
- If no error occurs, you DON'T return from the call

Wait
- the parent waits for the child to finish execution
- if there are multiple child processes
    - the parent waits for atleast one child to terminate

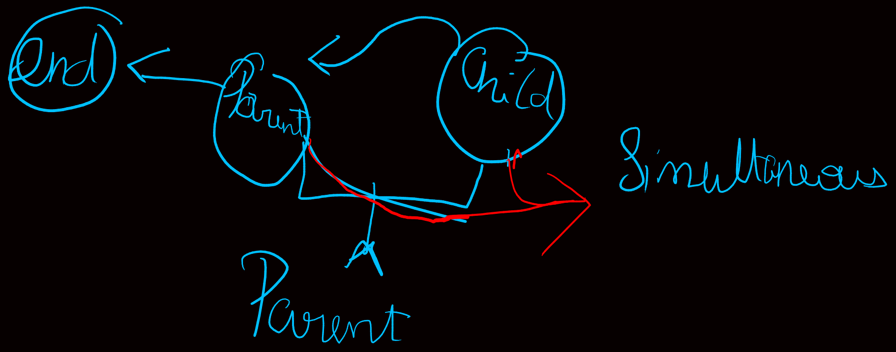
wait(&status);

pid_ t (signed integer) for the PIDs
fork() -> returns 0 for the child and the Child PID for the parent

wait(NULL); -> wait for all children, NULL if you're not bothered about the status code

the child gets the code of the parent from the fork system call, not including the fork ofcourse :P, otherwise infinite loop

Hello



Exec:
Lets say you want to exec ls in the child process

call ls, who's path is /bin/ls, and then last arg is null

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

./a.out in /home
execlp("/home/ex.out", "./","hi",NULL); -> ./ itself works as it can infer
other variants: execv etc.

Modification of variables

Let both the parents and the child use a variable x, and let the child modify a variable which the parent uses
if both processes are sharing the vars, then the changes should reflect too
It doesn't however -- the variable isn't shared, they are independently copied.

```
int x = 20;
pid_t pid = fork();
if(pid == 0){
        x = 10;
        printf("Child Process\n");
        printf("%d\n", x);
}
else{
        wait(NULL);
        printf("%d" ,x);
        return 0;
}
```

parent - 20
child - 10

```
int x = 20;
pid_t pid = fork();
if(pid == 0){
        for(int i = 0; i < INT_MAX ;i++);
        printf("Child Process\n");
        printf("%d\n", x);
}
else{
       x = 10;
        wait(NULL);
        printf("%d" ,x);
        return 0;
}
```

child - 20
parent - 10

Both have independent copies

getpid() get process id of the calling process
getppid() get process id of the parent of the calling process

exit - asks OS to delete the process
may return status data to parent, via wait()
parent may terminate the execution of child process because
- child has exceeded allocated resources limit
- task assigned to child is no longer required

Parent is exiting and OS doesn't let child to exist without parent

This can lead to Cascading termination

All children, grandchildren terminated

If child terminated, but the parent didn't call wait
-the entry for the child is still in the process table
- the child is now a Zombie process
- all processes will be zombie process for sometime, till the wait function is executed

```
ps -el | grep "Z"
```

If the child is executing and then the parent terminates - orphan