Processes and System Calls

Objectives:

- Understand processes and `fork()` in Linux.
- Learn to use `wait()` and `waitpid()` to manage child processes.
- Explore `exec()` system call to run new programs.

Process

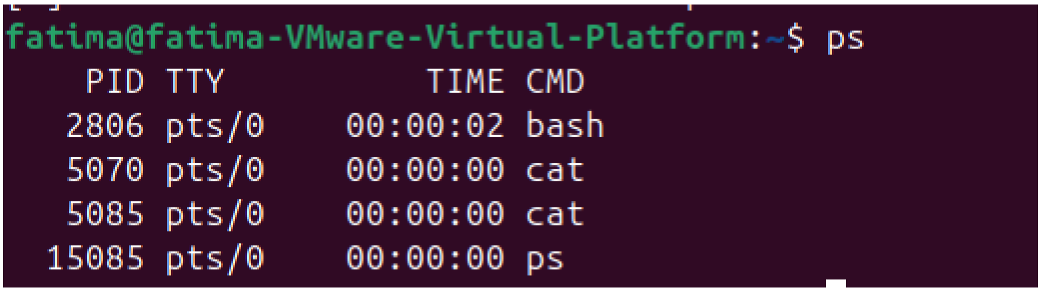
A process is a computer program that is currently in execution. In other words, an instance of a program is called a process. A process is a dynamic entity, as it changes state during execution.
A process requires system resources such as:

- CPU time
- Memory
- Input/Output devices

These resources are allocated by the operating system when the process is created.In simple terms, every command you execute in Linux starts a process**.**

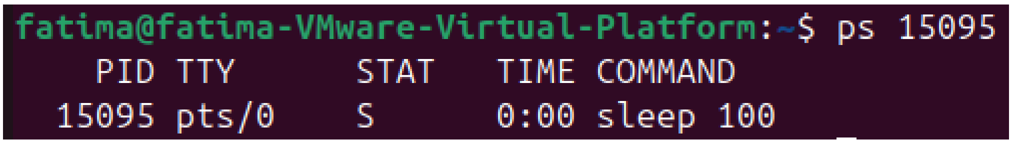
Viewing Processes

- The *ps* command displays information about processes associated with the current terminal (shell)**.**



```
fatima@fatima-VMware-Virtual-Platform:~$ ps
    PID TTY          TIME CMD
   2806 pts/0    00:00:02 bash
   5070 pts/0    00:00:00 cat
   5085 pts/0    00:00:00 cat
  15085 pts/0    00:00:00 ps
```

- The *ps aux* command displays all running processes in the system, including processes owned by other users and background/system processes.
- You can also check the process status of a single process, using the command - *ps PID*

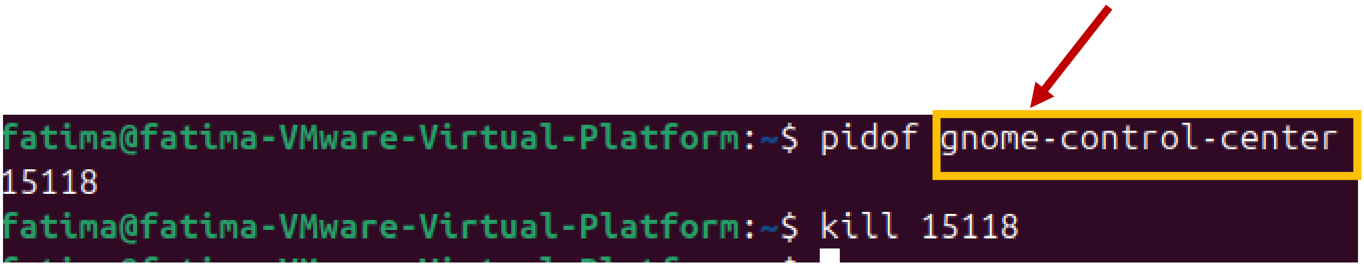


```
fatima@fatima-VMware-Virtual-Platform:~$ ps 15095
    PID TTY      STAT   TIME COMMAND
  15095 pts/0    S      0:00 sleep 100
```

Killing the process:

To kill aprocess, you need to know the PID (process id) of the process you want to kill. To find the PID of a process simply type: *pidof process_name* .

process name for the Settings application in Linux.



```
fatima@fatima-VMware-Virtual-Platform:~$ pidof gnome-control-center
15118
fatima@fatima-VMware-Virtual-Platform:~$ kill 15118
```

Creating Processes:

In Linux, process are created by first duplicating the parent process. This is called forking.

fork():

It is a system call that creates a new process under the Linux operating system. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the calling process. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

Return value of fork():

a. fork() returns a positive value to the parent process. This value is the PID (Process ID) of the newly created child.
b. fork() returns zero (0) to the child process.
c. If fork() fails (due to system limits), it returns -1 to both the parent and child.

A process can use function getpid() to retrieve the process ID assigned to this process. Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that the child process gets a copy of the parent process's memory, file descriptors, and other resources.

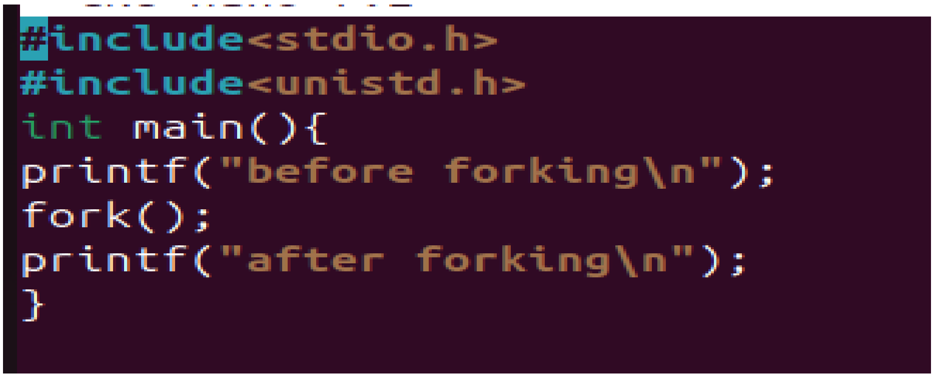
Working:

a. The parent and child processes run concurrently after the fork().
b. Both processes will execute the code that follows the fork() call, but they will have different return values for fork().
c. Memory and resources are copied, but modifications in one process (parent or child) do not affect the other, as memory is duplicated.

Waiting for child process:

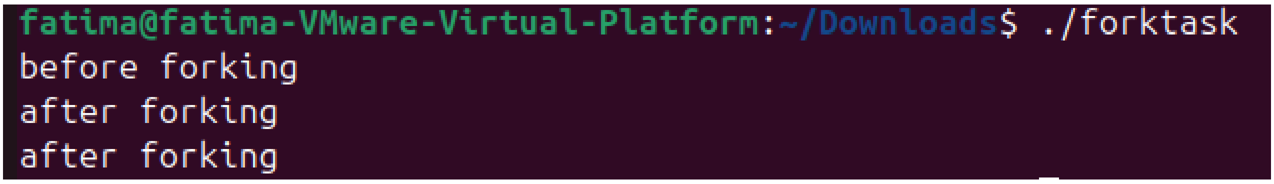
a. The parent process can wait for the child process to finish using wait() or waitpid(). This helps avoid zombie processes (terminated children that still occupy process table slots).
b. The child process can exit with an exit status using the exit() system call. The parent can collect this status.

Let’s do an example:



```
#include<stdio.h>
#include<unistd.h>
int main(){
printf("before forking\n");
fork();
printf("after forking\n");
}
```

The output is:



```
fatima@fatima-VMware-Virtual-Platform:~/Downloads$ ./forktask
before forking
after forking
after forking
```

Example to distinguish parent and child process:

Consider one simpler example, which distinguishes the parent from the child.

Example#01

```
#include<stdio.h>

#include<unistd.h>

void main(){

int p=fork();  //call fork();

if(p==0){

printf("\n Hi I am child process\n");

} else {

printf("\nHi I am Parent Process\n");

}
```

Let’s see an example of how waitpid() works:

Example:02

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>  // defines pid_t

#include <sys/wait.h>  // used for wait(), waitpid()

int main() {

    pid_t pid;

    pid = fork();   // create child process
    if (pid == -1) {

        perror("fork failed")

        exit( EXIT_FAILURE );

    }

    if (pid == 0) {

        printf("Child process running (PID = %d)\n", getpid());

        sleep(2);

        exit(10);   // child exits with status 10

    }

    if (pid > 0){

int status;

if (waitpid(pid, &status, 0) == -1){

            perror("waitpid() failed");

            exit(EXIT_FAILURE);}

 if (WIFEXITED(status)) {

            int es = WEXITSTATUS(status);

            printf("Exit status was %d\n", es);

        }

    }

    return 0;}
```

It represents unsuccessful program termination. Defined by the C standard in <stdlib.h>

`waitpid(pid, &status, 0)` waits for the child process with the given pid to finish and stores its exit information in `status`. If it returns `-1`, an error occurred; otherwise, it returns the PID of the child.

exec() System Call:

The **exec** system call is a Unix/Linux system call that is used to replace the current process image with a new process image. When a process executes the **exec** system call, it loads a new program into its address space, replacing the previous program that was running.

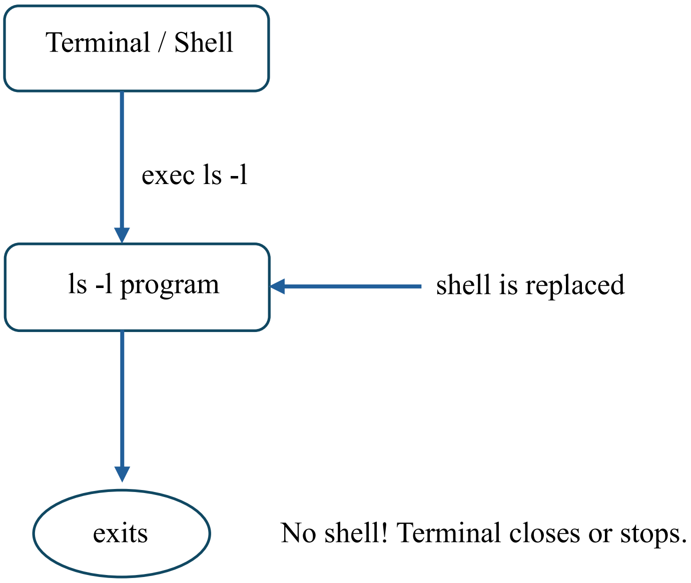
The **exec** system call is typically used by shell programs to execute other programs. When a user enters a command at the shell prompt, the shell program first forks a new process, and then uses the **exec** system call to replace the new process with the program specified by the user's command.

The **exec** system call comes in several variants, such as **execl**, **execv**, **execle**, **execve**, etc., each with slightly different behaviour and usage.

To understand exec command open terminal and write exec command,

exec ls -l

once the command completes your terminal will be closed.



1. **execl**: Executes a program, passing the arguments as a list of parameters.

   ```
   int execl(const char *path, const char *arg, ...);
   ```

   Example:

   ```
   execl("/bin/ls", "ls", "-l", NULL);
   ```

   This call executes the ls -l command.

2. **execlp**: Similar to execl, but searches for the program in the directories listed in the PATH environment variable.

   ```
   int execlp(const char *file, const char *arg, ...);
   ```

   Example:

   ```
   execlp("ls", "ls", "-l", NULL);
   ```

   Here, ls is found in one of the directories listed in PATH.

3. **execle**: Similar to execl, but also allows specifying the environment for the new program.

   ```
   int execle(const char *path, const char *arg, ..., char *const envp[]);
   ```

   Example:

   ```
   char *env[] = {"HOME=/usr/home", "PATH=/bin:/usr/bin", NULL};
   execle("/bin/ls", "ls", "-l", NULL, env);
   ```

4. **execv**: Executes a program, passing the arguments as an array of strings. Takes the path of the executable and an array of arguments (where the last element is NULL).

   ```
   int execv(const char *path, char *const argv[]);
   ```

   Example:

   ```
   char *args[] = {"ls", "-l", NULL};
   execv("/bin/ls", args);
   ```

5. **execve**: Similar to execv, but it allows specifying both the argument array and the environment array.

   ```
   int execve(const char *path, char *const argv[], char *const envp[]);
   ```

Example:

```
char *args[] = {"ls", "-l", NULL};
char *env[] = {"HOME=/usr/home", "PATH=/bin:/usr/bin", NULL};
execve("/bin/ls", args, env);
```

6. **execvp:** Similar to execv, but searches for the program in the directories listed in the PATH environment variable.

```
int execvp(const char *file, char *const argv[]);
```

Example:

```
char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
```

Example:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main() {
pid_t pid;
pid = fork();
if (pid < 0) {
perror("fork failed");
exit(EXIT_FAILURE);
}
if (pid == 0) {
// Child process
printf("Child will execute 'ls -l'\n");
// Replace child process with ls -l
execlp("ls", "ls", "-l", NULL);
// Only runs if exec fails
perror("exec failed");
exit(EXIT_FAILURE);
} else {
// Parent process
int status;
printf("Parent waiting for child to finish...\n");
wait(&status);
if (WIFEXITED(status)) {
printf("Child exited with status = %d\n", WEXITSTATUS(status));
}
printf("Parent continues execution.\n");
}
return 0;
}
```