# Chapter 3

# Processes

A system call is an interface between your program and the kernel. The linux kernel's job is to provide a variety of services to application programs and this is done using the provision of system calls.

## 3.1 Understanding Pro cesses

The fundamental building block of each pro gram is the process. A process is the name given to a program when it is loaded for the purpose of execution on the operating system. For a full description of what is comprised in side a process, please refer to your class-notes or slides.

To view a list of current processes in the sys tem for a user, you may use the ps command.

```
ps
```

To view the complete list, you can adapt it

```
to: ps au
```

You will see plenty of columns as output. The columns you should be familiar with at this moment are underlined below:

- User: The owner of that process
- PID: The integer identi er
- CPU: Percent utilization of CPU
- MEM: Percent utilization of Memory space
- VSZ: Virtual Memory Size
- RSS: Non-swapped physical memory size
- TTY: Controlling Terminal
- STAT: The process states (D) Uninter ruptible sleep (R) Running or ready (S) Interruptible sleep (T) Stopped (Z) Zom bie (<) High Priority (N) Low Priority (s) Session leader, i.e., has child processes (l) multi-threaded (+) foreground
- START: When process has

started • TIME: Time running since

- COMMAND: The program/command used in this process.

The basic attribute of a process is it's ID (PID), and it's Parent ID (PPID). The sys tem calls that can nd the process ID, and the Parent Process ID are the getpid() and getp pid() calls respectively. To use both of them, you will be required to include the sys/types.h and unistd.h C libraries.

The mere presence of the PPID means that there is a hierarchy of processes inside our op erating system. Each process has a track of who are it's children, and each child process has a record of who is it's parent. The original process that is the grand-father of all processes is the Init Process. You can view this hierar chy using the pstree command.

```
pstree
```

This command will show a list of all processes currently in the system in the form of a tree.

### 3.1.1 Exercise

With respect to the my rst.c le that you cre ated in last labs, write a code that is able to nd out the following:

- The PID value for my rst.c
- The PPID value for my rst.c
- The process name from the PPID value. (Note: You can use the system() call for this purpose. To nd a process name from PPID, you would normally enter the fol lowing command on the shell, where 12345 is the ID of any process)

```
pstree -p | grep 12345
```

## 3.2 Process Lifecycle

Each process has to go through the following states during it's existence:

1. Creation
2. Running
3. Non-Running
   (a) Ready
   (b) Waiting
4. Termination

### 3.2.1 Creation States

About process creation concepts, understand and run the following code:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
 int main(void)
{
  int i;
  printf("Process PID %6d \t PPID %6d \n",
                                    getpid(), getppid());
  for (i = 0; i<1; ++i)
  {
    if (fork()==0)
    printf("Process PID %6d \t PPID %6d \n",
                                    getpid(), getppid());
  }
  return 0;
}
```

Q1 How many processes are created?

Q2 Increase the value in for loop from i<1 to i<2 (i.e., 2 iterations in the loop). Com pile and run your program. How many processes does it show this time? Draw a tree hierarchy of processes that you just created as given in Figure-3.1.

Q3 Increase the value again to i<3 (i.e., 3 it erations). Compile and run your program. How many processes does it show? Draw a tree again. Why is it that we have called fork() 3 times in our code, yet we are see ing $2^n - 1$ processes listed on screen?

Q4 For fun, increase the value yet again to 100. Compile and run. What is going to happen? Does your OS Crash? Does your Program Crash? Can you modify your code to count the total number of fork()s made?

Try and run the following code:

```
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <unistd.h>
04 int main(void)
05 {
06 fork(); printf("He\n");
07 fork(); printf("Ha\n");
08 fork(); printf("Ho\n");
09 }
```

Note that we are calling each He, Ha, and Ho only once. Yet that does not appear when the program is run.

Q5 Can a Ho be output before a He? Why?

### Explanation

The Fork() system call simply creates a new process which is exact replica of parent process. It requires the unistd.h C library. It's usage is as such:

```
// Fork code 1
#include <unistd.h>
#include <???> // See Question 2 int
 main()
{
   int p;
   p = fork();
   printf("Job Done\n");
}
```

To understand this code, try to answer the fol lowing questions:

Question1 We have used p = fork(). Why not simply fork()? Check man fork for answer.

Question2 Check man page for printf. What library is used for this call?

Question3 Run your program. Why is it that printf() is used only once, yet we see the output Job Done displaying twice on our screen.

Question4 Add the following statement to the end of your code and run it again. What output would you see?

```
printf("Value of P is %d\n", p);
```

So, the fork() system call will always return di erent kinds of values (As you should know from Question4). One of the values returned would be 0 , the other will be a positive non zero value.

The process that receives the 0 value is the child process.

The process that receives the positive non zero value is the parent process.

Based on these values, we can then program our code in such a manner that both parent and child can do their own jobs and not appear to be doing the same things.

Why 0? Why Non-Zero?

The reason for this is that the child can al ways nd out who it's parent is via the getp pid() function call. However, it is di cult for the parent to know who it's children are. This can only be done if the value of the child's PID is returned to it at the time of the fork().

Look at the following code structure and im plement it:

```
// Fork code 2
#include <unistd.h>
#include <???> // For printf()
int main()
{
   int p;
   printf("Original Process,
                    pid = %d\n", getpid() );
   p = fork();
 if (p == 0)
 {
        printf("Child PID = %d, PPID = %d\n",
                        getpid(), getppid() );
 }
 else
 {
      printf("Parent PID = %d, Child ID = %d\n",
                        getpid(), p);
 }
}
```

What would happen if we don't use the If/Else conditions and immediately write the two printf() statements?

Make sure you understand the structure of the Code, as well as what are the ways of know ing the following:

1. The PID

2. The Parent PID

3. The Child PID

### 3.2.1.1 Exercise 1

Modify the fork() call code 2 above and add a system call for sleep() after both the printf() statements. Give a time of 120 seconds to the sleep call. While both the parent and child are sleeping, open a new terminal and check out the outcome of pstree command as well as the ps command. Study the output for your parent and child process in both commands, especially noting the STAT column.

### 3.2.1.2 Exercise 2

Model a fork() call in C/C++ so that your pro gram can create a total of EXACTLY 6 pro cesses (including the parent). (Note: You may check the number of processes created using the method from Exercise 1 (Section-3.2.1.1 by using a sleep of 60 seconds or more and enter ing eithe ps, or pstree in another terminal). Your process hierarchy should be as follows:


Figure 3.1: Exercise 2 Model

### 3.2.2 Running States

The job of the exec() call is to replace the current process with a new process/program. Once the exec() call is made, the current pro cess is gone and a new process starts. The Exec() call is actually a family of 6 system calls. Di erence between all 6 can be seen from man 3 exec

- int execl(const char *path, const char *arg, ...);
- int execlp(const char * le, const char *arg, ...);
- int execle(const char *path, const char *arg, char *const envp[]);
- int execv(const char *path, char *const argv[]);
- int execve(const char *path, const char *argv[], char *const envp[]);
- int execvp(const char * le, char *const argv[]);

Following is usage of one avor of Exec, the execv() system call:

```
#include <unistd.h>
#include <stdio.h>
int main()
{
   int p;
   char *arg[] = {"/usr/bin/ls", 0}; p =
   fork();
   if (p == 0)
   {
         printf("Child Process\n");
      execv(arg[0], arg);
```

you understand, try nding answers to the

```
         printf("Child Process\n");
   }
   else
   {
         printf("Parent Process\n");
   }
}
```

Again look at the code. We have referred to Exec() system call but in code we see Execv(). Read man pages for this and nd out. To help fol lowing:

Question1 What is the 1st argument to the execv() call? What is it's contents?

Question2 What is the 2nd argument to it? What is it's contents?

Question3 What is arg?

Question4 Look at the code of the child pro cess (p==0). How many times does the statement Child Process appear? Why?

Diagrammatically, the functioning of the exec() system calls would be represented in Figure 3.2, where the dotted line mark the execution and transfer of control whereas the straight lines mark the waiting time.

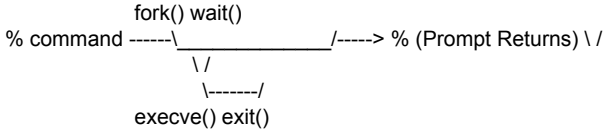


Figure 3.2: Fork() Exec() representation

### 3.2.3 Waiting States

#### 3.2.3.1 Sleep()

The sleep() call can be used to cause delay in execution of a program. The delay can be pro vided as an integer number (representing num ber of seconds). Usage is simply sleep(int), which will delay the process execution for int seconds. To use sleep(), you will require the unistd.h C library.

Exercise

Write a C program that can display a count from 10 to 0 (reverse order) using a for or a while loop. Each number should be displayed after delay of 1 second.

## 3.2.4 Termination States

A process can terminate in one of the following ways:

1. Main function calls return
2. Exit function called
3. Aborted by higher priority process (e.g., parent or kernel)
4. Terminated by a signal

Regardless of any of these reasons, the same kernel code is invoked which performs the fol lowing:

1. Close open les
2. Noti es Parent and Children
3. Release memory resources

### 3.2.4.1 Exit() Call

The exit() system call causes a normal program to terminate and return status to the parent process. Study the behaviour of the following program. You will see that there may be a number of exit points from a program. Can you identify which exit() call is being used each time a program exits??

```
#include <stdio.h>
int main()
{
   int num;
  {
   void anotherExit(); // Function
   Prototype printf("Enter a Number: ");
   scanf("%i", num);
   if (num>25)
   {
   printf("exit 1\n");
   exit(1);

}
else
anotherExit(); }
void anotherExit()
   printf("Exit 2\n");
   exit(1);
}
```

### 3.2.4.2 Atexit() Call

The code below provides two functions; atexit() and exit(). Note the structure of code and the behaviour of execution and then at tempt the questions in the end.

```
01 #include<stdio.h>
02 #include<stdlib.h>
03 int main(void) {
04 void f1(void), f2(void), f3(void);
05 atexit(f1);
06 atexit(f2);
07 atexit(f3);
08 printf("Getting ready to exit\n");
09 exit(0);
10 }
11 void f1(void) {
12 printf("In f1\n");
13 }
14 void f2(void) {
15 printf("In f2\n");
16 }
17 void f3(void) {
18 printf( In f3\n );
19 }
```

Q1 What is the di erence between exit() and atexit()? What do they do? (Check man atexit and man 3 exit).

Q2 What does the 0 provided in the exit() call mean? What will happen if we change it to 1? (Check manual page for exit)

Q3 If we add an exit call to function f1, f2, or f3. What will happen to execution of our program?

Q4 Why do you think we are getting reverse order of execution of atexit calls?

### 3.2.4.3 Abort Call

Type, run and execute the code below. Then answer the questions in the end.

```
01 #include <stdio.h> 02 #include <stdlib.h> 03
int main() {
04 abort();
05 exit(0);
06 }
```

Q1 Check the man pages for abort. How does the abort call terminate the pro gram? What is the name of the particular signal?

Q2 Execute your program. What is the out put of our program?

Q3 Include the abort call in function f3 in our code provided for Atexit() call. How does our program terminate using this?

#### 3.2.4.4 Kill Call

Full description of this call will appear in Section-5.1.2. But as a demonstration, you may run the following code to see how the kill() call works.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
 int main()
{
  printf("Hello");
  kill(getpid(), 9);
  printf("Goodbye");
}
```

You are already familiar with getpid() system call. To nd out what 9 is, rst look at the output of the command:

```
kill -l
```

Find out the word mentioned next to 9. Search on the internet for this.

## 3.3 Death of Parent or Child

We covered the process termination using the exit() system call and that a program may have more than one exit points. In a relationwhip where a process has spawned children, what would be the e ect if either the parent dies or the child dies on the other processes linked to it? This section will look and study such a development.

### 3.3.1 Parent Dies Before Child

If a parent process dies before it's children, the children will be orphaned. They will be as signed to another process in the system and as such the children should be informed about who their new parent is going to be. This new parent process is the parent of all processes in the system, i.e., the init process.

```
#include <unistd.h>
#include <stdio.h>
 int main()
{
   int i, pid;
   pid = fork();
   if (pid > 0) // Parent
   {
      sleep(2);
      exit(0);
   }
   else if (pid == 0) // Child
   {
      for (i=0; i < 5; i++)
      {
         printf("My parent is %d\n", getppid());
         sleep(1);
      }
   }
}
```

Run the code.

1. What are the PPID values you are receiv ing from the for loop?
2. What has happened when the numbers of the PPID change?
3. What is now PID of the init process?

### 3.3.2 Child Dies Before Parent

Following code is complete opposite of what we have seen so far.

```
#include <unistd.h>
#include <stdio.h>
 int main()
{
   int i, pid;
   pid = fork();
   if (pid > 0) // Parent
   {
      sleep(120);
   }
   else if (pid == 0) // Child
   {
      exit(0);
   }
}
```

Run the code. In another window (terminal), check the status of your parent and child pro cesses using the ps command. You should be able to see a Z, or <defunct> next to the child process which has been created. Such a process is usually termed a Zombie process.

Wait for 60 seconds, then check the status of your process again. You will note that both the parent, as well as the Zombie process are now gone.