Operating System

Name: Niraj Kumar Tiwari
22MCA10026

fork ():

The fork () is one of the system calls that is very special and useful in Linux/Unix systems. It is used by processes to create the processes that are copies of themselves. With the help of such system calls, the child process can be created by the parent process. Until the child process is executed completely, the parent process is suspended.

Some of the important points on fork () are as follows.

- The parent will get the child process ID with a non-zero value.
- Zero Value is returned to the child.
- If there will be any system or hardware errors while creating the child, -1 is returned to the fork ().
- With the unique process ID obtained by the child process, it does not match the ID of any existing process group.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
int main(int argc, char **argv)
{
pid_t pid;
pid = fork();
if(pid==0)
{
printf("It is the child process and pid is %d\n",getpid());
exit(0);
}
else if(pid > 0)
{
printf("It is the parent process and pid is %d\n",getpid());
}
else
{
printf("Error while forking\n");
exit(EXIT_FAILURE);
}
return 0;
}
```

Output:

```
It is the parent process and pid is 2091
It is the child process and pid is 2095
```

exec ():

The exec () is such a system call that runs by replacing the current process image with the new process image. However, the original process remains as a new process but the new process replaces the head data, stack data, etc. It runs the program from the entry point by loading the program into the current process space.

Code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
main(void) {
  pid_t pid = 0;
  int status;
  pid = fork();
if (pid == 0) {
  printf("I am the child.");
  execl("/bin/ls", "ls", "-l", "/home/ubuntu/", (char *) 0);
  perror("In exec(): ");
}
if (pid > 0) {
  printf("I am the parent, and the child is %d.\n", pid);
  pid = wait(&status);
  printf("End of process %d: ", pid);
  if (WIFEXITED(status)) {
    printf("The process ended with exit(%d).\n", WEXITSTATUS(status));
  }
  if (WIFSIGNALED(status)) {
    printf("The process ended with kill -%d.\n", WTERMSIG(status));
  }
}
if (pid < 0) {
  perror("In fork():");
}
exit(0);
}
```

Output:

```
ubuntu@linuxways:~$ ./exec
I am the parent, and the child is 18264.
total 18
drwxrwxr-x 2 ubuntu ubuntu     3 Dec 10 13:58 __pycache__
-rwxrwxr-x 1 ubuntu ubuntu 16912 Dec 14 09:23 exec
-rw-r--r-- 1 root   root     962 Dec 14 09:23 exec.c
-rwxrwxr-x 1 ubuntu ubuntu 16864 Dec 14 07:43 fork
-rw-r--r-- 1 root   root     555 Dec 14 07:41 fork.c
-rw-r--r-- 1 ubuntu ubuntu   341 Dec 10 13:58 sample.py~
-rw-r--r-- 1 root   root     119 Dec 10 14:25 sample1.py
-rw-r--r-- 1 ubuntu ubuntu   185 Dec 10 15:10 sample2.py
-rw-r--r-- 1 root   root     349 Dec 10 15:48 sample3.py
-rw-r--r-- 1 root   root     264 Dec 10 16:22 sample4.py
-rw-r--r-- 1 root   root      94 Dec 10 16:32 sample5.py
End of process 18264: The process ended with exit(0).
```

wait ():

As in the case of a fork, child processes are created and get executed but the parent process is suspended until the child process executes. In this case, a wait () system call is activated automatically due to the suspension of the parent process. After the child process ends the execution, the parent process gains control again.

Code:

```
#include<stdio.h> // printf()
#include<stdlib.h> // exit()
#include<sys/types.h> // pid_t
#include<sys/wait.h> // wait()
#include<unistd.h> // fork
int main(int argc, char **argv)
{
pid_t pid;
pid = fork();
if(pid==0)
{
printf("It is the child process and pid is %d\n",getpid());
int i=0;
for(i=0;i<8;i++)
{
printf("%d\n",i);
}
exit(0);
}
else if(pid > 0)
{
printf("It is the parent process and pid is %d\n",getpid());
int status;
wait(&status);
```

```
printf("Child is reaped\n");
}
else
{
printf("Error in forking..\n");
exit(EXIT_FAILURE);
}
return 0;
}
```

OUTPUT:

```
It is the parent process and pid is 209
It is the child process and pid is 213
0
1
2
3
4
5
6
7
Child is reaped
```

exit ():

The exit () is such a function or one of the system calls that is used to terminate the process. This system call defines that the thread execution is completed especially in the case of a multi-threaded environment. For future reference, the status of the process is captured.

After the use of exit () system call, all the resources used in the process are retrieved by the operating system and then terminate the process. The system call Exit () is equivalent to exit ().

Synopsis:

```
#include <unistd.h>
void _exit (int status);
#include <stdlib.h>
void _Exit (int status);
```

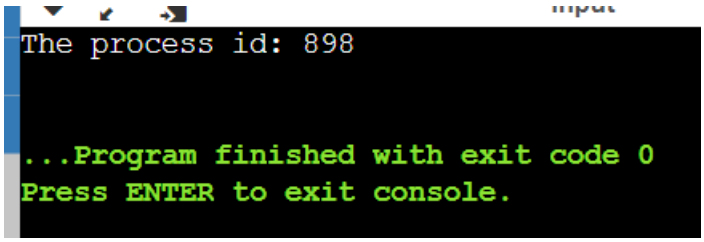
getpid ():

When any process is created, it has a unique id which is called its process id. This function returns the process id of the calling function.

CODE:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
	//variable to store calling function's process id
	pid_t process_id;
	//variable to store parent function's process id
	pid_t p_process_id;
	//getpid() - will return process id of calling function
	process_id = getpid();
	//printing the process ids
	printf("The process id: %d\n",process_id);
	return 0;
}
```

OUTPUT:



close()

To close a channel, use the close() system call. The prototype for the close() system call is:

Synopsis:

int close(file_descriptor)
int file_descriptor;

where file_descriptor identifies a currently open channel. close() fails if file_descriptor does not identify a currently open channel.