QUIDET Victor
TARBE Théophile
ING4- GR01 (Inter)

LAB 1

II Creating and Running a Process (1) - *fork*

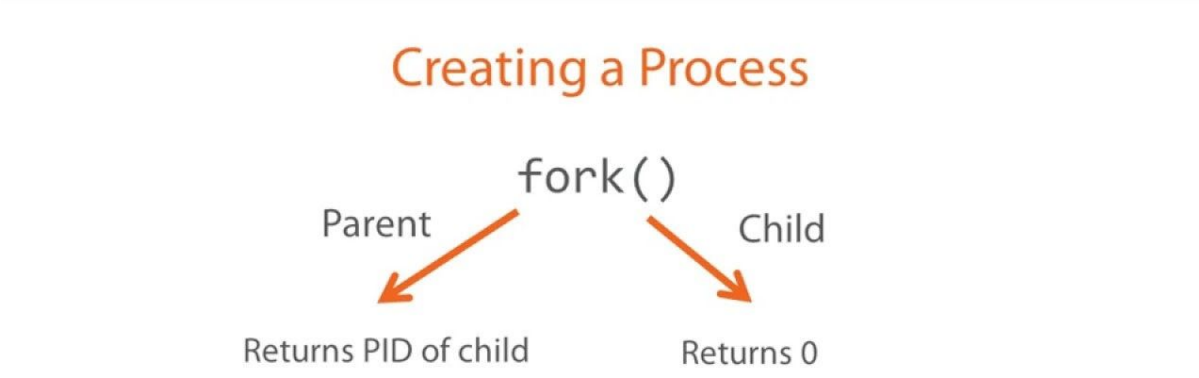
1)

→ fork manual
https://man7.org/linux/man-pages/man2/fork.2.html
→ getpid / getppid manuals
https://man7.org/linux/man-pages/man2/getpid.2.html

2)

What happens after a `fork()` call ?

→ It creates a new process by duplicating the calling process. Then both the parent and the child process start execution from the next instruction.
→ The function `fork()` returns the process Id of the child to the parent process :



How are parent and child differentiated ?

→ The child process and the parent process run in separate memory spaces. Their PIDs are different.

3)

Write a small C program in which the parent process creates a child process and each displays a different message : I'm the parent vs I'm the child. Display the process id and the parent process id for every running process.

Users > theophiletarbe > Desktop > cours > ING4 > SystemeExpl > lab1 > C new.c > main(void)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {

  pid_t pid = fork();

  if(pid == 0) { //Child process
    printf("I'm the Child ! => PPID: %d PID: %d\n", getppid(), getpid());
    exit(EXIT_SUCCESS);
  }
  else if(pid > 0) { //Parent process
    printf("I'm the Parent ! => PID: %d\n", getpid());
    wait(NULL); //waiting for the end of child process
  }
  else {
    printf("Unable to create child process.\n");
  }

  return EXIT_SUCCESS;
}
```

```
macbook-air-de-theophile-2:lab1 theophiletarbe$ gcc -g -o exe new.c
macbook-air-de-theophile-2:lab1 theophiletarbe$ ./exe
I'm the Parent ! => PID: 39568
I'm the Child ! => PPID: 39568 PID: 39569
macbook-air-de-theophile-2:lab1 theophiletarbe$
```

→ We can observe the process IDs. As we can see, the PID of the parent process is the PPID of the child process. So, the child process **39569** belongs to the parent process **39568**.

→ It's interesting to have process IDs because it allows us to get a process tree, and understand which process belongs to who. It gives a precise vision of how the code is running/working.

4) Is data shared between parent and child ?

The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. But the parent and the child don’t share these portions of memory.

```
int i = 5;

if (fork() == 0) {
  // I’m the ...
  i++;
} else {
  // I’m the  ...
  sleep(3); // sleep for 3 seconds
  printf("%d\n", i); // what happens here ?? Explain
}
```

In this portion of code, the condition `if(fork() == 0)` creates a new process :

- The child process will verify the condition `fork() == 0`, and then the value of i will be incremented from 5 to 6 (`i++`).
- The parent process won’t verify the condition. Then it will print the value of i which didn’t change : `i==5`.

We can verify it if we compile the code, it gives us the value `5` :

```
macbook-air-de-theophile-2:lab1 theophiletarbe$ gcc -g -o exe test2.c
macbook-air-de-theophile-2:lab1 theophiletarbe$ ./exe
5
macbook-air-de-theophile-2:lab1 theophiletarbe$
```

5)

Is it possible to create more than one child process ?

→ Yes one parent can create multiple child processes. With the function `fork()` the total process created is equal to : 2^number of fork().

Show how using a simple program creates 2 children for the 1st-level process (main parent) and a child for one of the 2nd-level processes (children).

Users > theophiletarbe > Desktop > cours > ING4 > SystemeExpl > lab1 > C test3.c > main()

```c
#include <stdio.h>
#include <unistd.h>

int main(){
  int pi_d ;
  int pid ;

  pi_d = fork();

  if(pi_d == 0){
    printf("Child Process B (1st-level):\npid :%d\nppid:%d\n",getpid(),getppid());
    pid = fork();
    if(pid == 0){
      printf("Child Process C (2nd-level):\npid :%d\nppid:%d\n",getpid(),getppid());
    }
  }
  if(pi_d > 0){
    pid = fork();
    if(pid > 0){
      printf("\nParent Process:\npid:%d\n",getpid());
    }
    else if(pid == 0){
      printf("Child Process A (1st-level):\npid :%d\nppid:%d\n",getpid(),getppid());
    }
  }
}
```

```
[macbook-air-de-theophile-2:lab1 theophiletarbe$ gcc -g -o exe test3.c
[macbook-air-de-theophile-2:lab1 theophiletarbe$ ./exe

Parent Process:
pid:40197
Child Process B (1st-level):
pid :40198
ppid:40197
Child Process A (1st-level):
pid :40199
ppid:40197
Child Process C (2nd-level):
pid :40200
ppid:40198
macbook-air-de-theophile-2:lab1 theophiletarbe$
```

Here we can observe thanks to the PIDs and PPIDs that :

- The Parent process creates 2 child processes : A and B (1st-level process)
- The Child process B creates another process C (2nd-level process)

III Creating and Running a Process (2) - *exec*

2)

home > victor > Desktop > C lab1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(){

    char *args[]={"firefox", NULL};
    printf("PID  : %d\n",getpid());
    execvp(args[0], args);


}
```

```
victor@ubuntu:~/Desktop$ ./lab1
PID  : 33698
```

```
victor@ubuntu:~/Desktop$ pidof firefox
33879 33823 33759 33698
victor@ubuntu:~/Desktop$
```

Using the command *pidof* in a new terminal window we can verify that the PID of firefox is the same as the one of the original process.

3) Data regarding the process is shared, like the PID. But because the new program overlays the calling program, the machine code, data, heap and stack of the process are replaced by those of the new program.

4)

home > victor > Desktop > C lab1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void mySystem(char *cmd){
    system(cmd);
}

int main(){

    char *args[]={"firefox", NULL};
    int i = 5;

    if(fork() == 0){

        printf("PID of the child : %d\n",getpid());

        execvp(args[0], args);
        i++;
        printf("%d\n", i);
    }
    else {
        printf("PID : %d\n",getpid());
        mySystem("pidof firefox");
    }


}
```

```
victor@ubuntu:~/Desktop$ gcc -g -o lab1 lab1.c
victor@ubuntu:~/Desktop$ ./lab1
PID : 15037
PID of the child : 15038
15038 6854 2395 2344 2290 2199
```

The difference resides in the fact that this time, we create a new process by using *fork* before executing the program. By doing this, the “main” program continues its execution because it is a different process that is affected by the launching of firefox. The line printf(“%d\n”, i); is not executed because the process has been taken for the execution of firefox right before.

Sources :

https://linuxtrainers.wordpress.com/2014/12/31/how-fork-system-call-works-what-is-shared-between-parent-and-child-process/

https://linuxhint.com/fork_linux_system_call_c/

https://www.geeksforgeeks.org/creating-multiple-process-using-fork/

https://stackoverflow.com/questions/6542491/how-to-create-two-processes-from-a-single-parent

https://en.wikipedia.org/wiki/Exec_(system_call)#Effects

https://itsfoss.com/how-to-find-the-process-id-of-a-program-and-kill-it-quick-tip/

https://www.geeksforgeeks.org/exec-family-of-functions-in-c/