Processes

Samuele Germiniani
samuele.germiniani@univr.it

UNIVERSITÀ
di VERONA
Department
of ENGINEERING FOR INNOVATION
MEDICINE

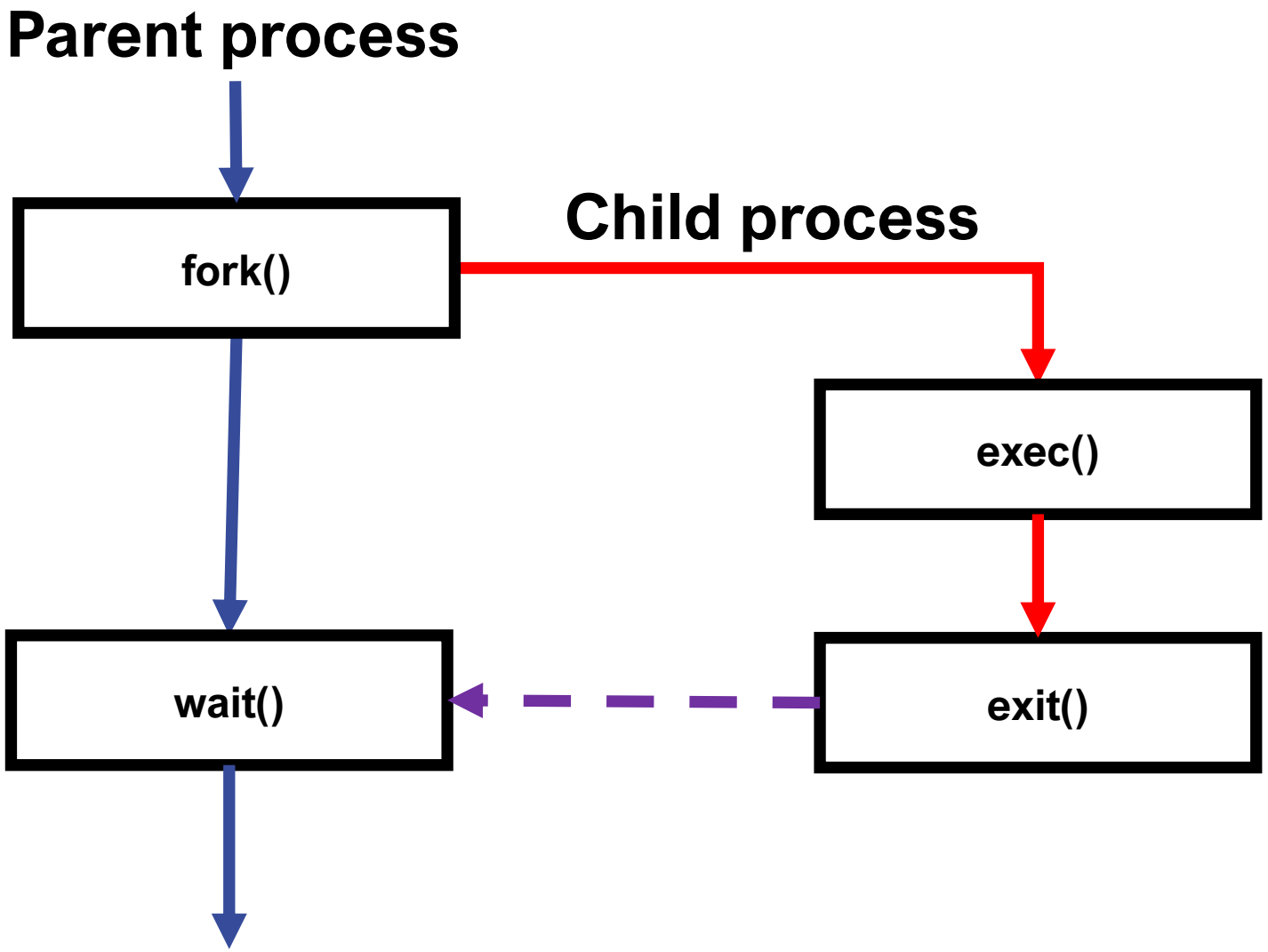
Introduction

A process is an instance of an executing program

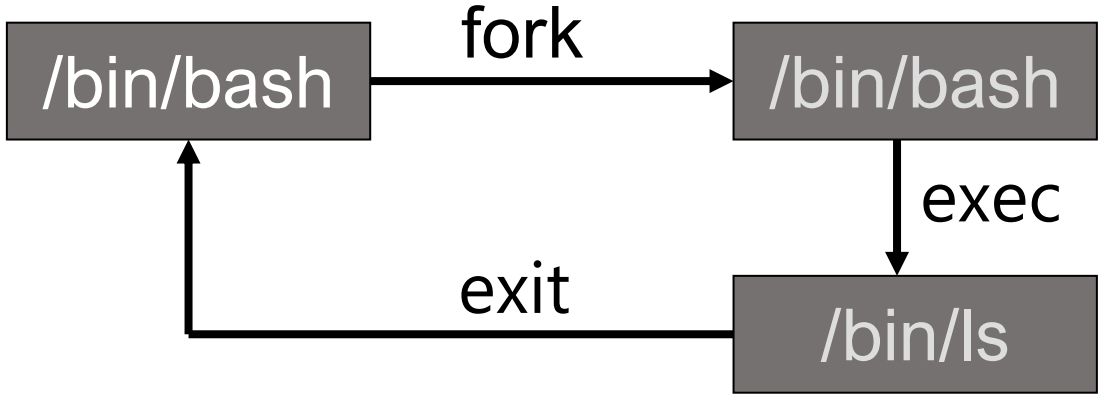
From the Kernel's point of view, a process consists of:

- User-space memory containing program code, the variables used by that code, and a set of kernel data structures that maintain information about the process's state

Process Life Cycle



Bash command lifecycle



Exercise - Implement a simple bash

- Compile and execute this code
- Give "**ls**" or "**ps**" as input

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

int main() {
    char command[100];
    // Continuously read commands from the user
    while (fgets(command, sizeof(command), stdin) != NULL) {
        // Remove newline character
        command[strlen(command) - 1] = '\0';
        // Fork a child process
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            // Execute the command using execlp
            execlp(command, command, NULL);
            // If execlp fails, print an error message and exit
            perror("execlp");
            exit(EXIT_FAILURE);
        } else {
            // Parent process
            int status;
            // Wait for the child process to complete and get its exit status
            wait(&status);
            // Print the exit status of the child process
            printf("Child process %d exited with status %d\n", pid,
                   WEXITSTATUS(status));
        }
    }
    return 0;
}
```

Exercise templates

- Download the templates of the exercises

```
git clone https://github.com/SamueleGerminiani/ex_processes_templates.git
```

Exercise 1

Specifications

- Write a program that reads a positive integer N from the command line.
- The program then creates N child processes. Each child process prints its PID, the PID of its parent process, and then terminates with an exit code (a random number between 0 and 255).
- After creating N child processes, the program waits for the termination of each child process.
- Whenever a child process terminates, the program prints the termination code of the child.

Hints

`atoi(<string>)`

`fork()`
`getpid()`
`getppid()`
`rand()`

`exit(<code>)`

`wait(&<status>)`
`WEXITSTATUS(<status>)`

Exercise 2

Specifications

- Modify the program from "Exercise 1" so that the parent process only waits for the termination of the last child created. The waiting must occur in polling mode!

Hints

Same as before +

waitpid(<pid>,<state>,<option>)

WNOHANG

Exercise 3

Specifications

- Using the environment variables, write a program that reads the username (USER variable) and home directory (HOME variable) of the current user.
- The program compares the path of its current working directory with the user's home directory.
- If the working directory is not a subdirectory of the user's home directory, the program sets the user's home directory as its working directory, creates an empty text file, and prints to the screen: "Dear USER, I am inside your home!", where USER is the user's username.
- If the working directory is a subdirectory of the user's home directory, the program prints to the screen: "Dear USER, I am already in the right place!".

Hints

getenv(<string>)
getcwd(<buffer>, sizeof(<buffer>))

strncmp(<buffer>, <homeDir>, strlen(<homeDir))

chdir(homeDir)
open(<fileName>)
close(<fileDescripto>)

Exercise 4

Specifications

- Write a program (moltiplicatore.c) that reads two integers, n and m, from the command line. The program prints to the screen: "The product of n and m is: x", where x is the result of the operation n * m.
- Write a second program (generator.c) that generates two random numbers. Using the exec system call, the program uses the program described in the first part to calculate their product.

Hints

`atoi(<string>)`
`rand()`

`sprintf(<string>, "%d", <number>)`
`execl(…)`

Exercise 5

Specifications

- Write a program that reads a system command X and its arguments from the command line. The program creates a child process that, using the exec system call, executes X and redirects its standard output and error to a text file.
- The parent process waits for the termination of the child process, prints the termination code to the screen, and then terminates.

Hints

`fork()`
close(<fileDescriptor>)
`STDOUT_FILENO`
open(<fileName>)

dup(<fileDescriptor>)
`execvp(…)`

`wait(&<status>)`