Data-Intensive Systems
Software Technology
EEMCS
Delft University of Technology

# Operating Systems (CSE2430)
# Lab 1: Introduction to Processes

**Deadline: 21 February 2025 - 12:30**

In this session, you will learn what a process is, how to create new processes, and how to build a command shell using processes.

## Processes in Linux

Early computers allowed only executed one program at a time. This program had complete control of the system and had access to all of the system's resources. In contrast, contemporary computer systems allow multiple programs to be executed concurrently. A process is a program in execution. Each process has a unique ID.

You can use the command `ps` in the console screen to display the current processes. Normally `ps` only shows the processes associated with the current user and terminal session. To show all the system processes you can use `ps -aux`. This often returns a long list so you might want to use `ps -aux | less` to be able to scroll through the text. Another option is to use the `top` command, which displays real-time process information.

To run a process from the terminal in the background, end your call with an `&`. For example try running and notice the difference between `sleep 3 &` and `sleep 3`. To stop a running process, you can use `kill <Process ID>` or `pkill <Process name>`. For example try calling `sleep 1000 &` and then try to stop this process using both methods.

## Assignments

For the assignments all code has to be written in C. Only the code has to be submitted to Weblab (more details on Weblab per assignment).

Some code is provided for assignments to enforce structure or avoid spending time implementing parts not relevant to current study goals. These materials are available on Brightspace and will be provided for each assignment. To ease the building process a `Makefile` is provided. To build an assignment, run `make <assignment>` where `<assignment>` is the name of the source file without '.c'. For example, to compile `2_2.c`, run `make 2_2`.

**Please read the assignments carefully and make sure your code exactly implements the given descriptions.** We will assess both the functionality and the clarity of your code and comments. **Weblab spec tests do not guarantee that your solution is fully correct.**

You have to research how to solve these assignments on your own: you can look for and read the description of the functions to use and study examples that you will find online. For example, type `man 2 fork` in a terminal to learn more about the function that creates a fork. To learn more details about the manual: `man man`.

### 1.1

Write a program that:

1. creates variables with your name and student number
2. prints your name and student number on a single line separated by a white space
3. prints the ID of the process that runs your program using the appropriate Linux system call

You can use the makefile for the compilation step which should make it easier to compile it again. You should observe and understand why the process ID differs between different runs.

### 1.2

Write a program that:

1. creates a sub-directory in the directory where the program is executed (i.e., the equivalent of the `mkdir` command)
2. lists all files and directories in the current directory (i.e., the equivalent of the `ls` command)

Implement each of the two tasks above using two methods:

- invoking a shell command from the C program using the `popen` command or the `system` command
- using calls from the C standard library to interact with the file system and access the contents of the working directory

### 1.3

Write a program that:

1. creates a child process running the same code as the main process using `fork`
2. uses an if/else statement to print which process is active (child or parent)
3. makes the child process show all files and folders using the functionality you built in assignment 1.2
4. makes the parent process wait for the child process to finish and return

### 1.4

This assignment involves designing a C program that implements a command shell, which accepts user commands and then executes each command in a separate process. A command shell interface provides the user with a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command `cat prog.c`, which displays file prog.c in the terminal using the Linux `cat` command:

```
osh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing.

Linux shells typically also allow the child process to run in the background or concurrently. To achieve this, one adds an ampersand (&) at the end of the command. Thus, if we rewrite the above command as `osh> cat prog.c &`, then the parent and child processes will run concurrently. The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the `exec()` family of system calls.

A C program that provides the general operations of a command-line shell is given below. The `main()` function presents the prompt `osh->` and outlines the steps to be taken after the user's input has been read. The `main()` function continuously loops as long as the boolean value `should_run` equals 1; when the user enters exit at the prompt, your program will set `should_run` to 0 and terminate.

This exercise is organized into two parts:

1. creating the child process and executing the command in the child, and
2. enabling the command shell to support several different commands.

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

#define MAX_LINE 80 /* The maximum length command */
int main(void) {
char *args[MAX_LINE/2 +1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */
while (should_run) {
printf("osh>");
fflush(stdout);
/**
* After reading user input, the steps are:
* (1) fork a child process using fork()
* (2) the child process will invoke execvp()
* (3) if command did not include &, parent will invoke wait()
*/
}
return 0;
}
```

**Part 1: Creating a Child Process**

The first task is to modify the `main()` function as mentioned above so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings. For example, if the user enters the command `ps -ael` at the `osh>` prompt, the values stored in the `args` array are:

```
args[0] = "ps"
```

```
args[1] = "-ael"
args[2] = NULL
```

This `args` array will be passed to the `execvp()` function, which has the following prototype:

```
execvp(char *command, char *params[]);
```

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the `execvp` function should be invoked as `execvp(args[0], args)`. Be sure to check whether the user included an & to determine whether or not the parent process is to wait for the child to exit.
Make sure that, like any normal shells, your program terminates gracefully when it receives an EOF as input. This is achieved by pressing `CTRL+D` on your keyboard when testing.

**Part 2: Shell Commands**

Your shell interface program must support the following shell commands:

1. `mkdir`
2. `rmdir`
3. `ls`
4. `cd`
5. `pwd`
6. `exit`

Some of these programs are already implemented by GNU coreutils and are accessible from the path as normal programs. Adding explicit support for those commands is not necessary. For this task it is important that all of these commands work as expected, which will require you to implement some of them as built-in commands.