# Lab 7

**Due** Oct 27, 2023 by 6:30p.m. **Points** 1

## Lab 7: fork

## Introduction

The purpose of this exercise is to play with `fork`, and get a feeling for how it works. It should be a fairly short lab. As usual, to begin login to MarkUs and add the starter files for this lab to your repository.

## 1. Run the simplefork program

Open `simplefork.c`. Read it through to figure out what it is doing. Compile and run it a few times. Recall from lecture that it is up to the operating system to decide whether the parent or the child runs first after the `fork` call, and it may change from run to run. Try running it on your own machine and on teach.cs.  Do you get output in the same order?

*Question 1*: Which lines of output are printed more than once?

*Question 2*: Write down all the different possible orders for the output. Note that this includes output orders that you may not be able to reproduce.

## 2. Fork in a loop

The program in `forkloop.c` takes one command-line argument, which is the number of iterations of the loop that calls `fork`. Try running the program first with 1, 2, or 3 iterations.

Here are a few things to notice:

- The original process still has a parent process id.  The parent of the original process is the shell.
- The shell prompt may appear in the middle of the output. This happens because the parent process exits before some of the children get a chance to print their output.
- Some of the parent process ids are 1. This happens when the parent process terminates before the child calls `getppid`. (What do we call the child and the parent process when it is in this state?) If you want avoid this situation you can add a `sleep(1);` just before the return call in main. Note that this is really just a hack and if we really want to ensure that a parent does not terminate before its child, we need to use `wait` correctly.

Question 3: How many processes are created, including the original parent, when `forkloop` is called with 2, 3, and 4 as arguments? *n* arguments?

Question 4: If we run `forkloop 1`, two processes are created, the original parent process and its child. Assuming the process id of the parent is 414 and the process id of the child is 416, we can represent the relationship between these processes using the following ASCII diagram:

```
414 -> 416
```

Use a similar ASCII diagram to show the processes created and their relationships when you run `forkloop 3`. In other words, how many processes does the parent create? How many do each of the child processes create?

## 3. Make the parent create all the new processes

Create a copy of `forkloop.c` called `parentcreates.c`. In the new file, modify the program so that the new children do not create additional processes. Only the original parent calls `fork`. Keep the `printf` call for all processes. The resulting diagram will look something like the following when `parentcreates 3` is run. In this case, the parent process creates 3 child processes. Note that the child process ids will not necessarily be in sequence.

```
414 -> 416
414 -> 417
414 -> 420
```

Note that your program does not need to produce the ASCII diagram, it only needs to ensure that only the original parent process is creating new children.

## 4. Make each child create a new process

Create a copy of `forkloop.c` called `childcreates.c`. In the new file, modify the program so that each process creates exactly one a new process. Keep the `printf` call for all processes. The resulting diagram will look something like the following when `childcreates 3` is called:

```
414 -> 416 -> 417 -> 420
```

## 5. Add wait (optional)

The information provided by the `getppid` system call may not always give you the information you expect. If a process's parent has terminated, the process gets "adopted" by the init process, which has a process id of 1, which is returned by `getppid`.

A process can *wait* for its children to terminate. If a process wants to wait until all its children have terminated, it needs to call `wait` once for each child. Add the appropriate `wait` calls to both `parentcreates` and `childcreates` to ensure that each parent does not terminate before its children.

Your programs should delay calling `wait` as long as possible. In other words, if the process has other work to do like creating more children, it should create the children first and then call `wait`.

## Submission

Submit your final `parentcreates.c` and `childcreates.c` files to MarkUs under the `lab7` folder in your repository.

You do not need to submit answers to the Questions 1-4.