In this exercise we familiarize ourselves to process creating and control in Linux OS.

Exercise 6a (Chain of processes, 1p)

Part A (Avoiding zombies in the server)

Phase 1

The following program simulates a server. The function `fgets` reads a command (that represents connection request from a client). If the command is "n" like next client, the server creates a child process to serve the client. The server cannot serve the client itself, because it may take a long time and the server needs to respond immediately to the possible new connection requests. For the same reason, the server cannot wait for the child (and block) either. That's why zombies are left in the system always when a client has been served.

Compile and run the program below. Enter n for example 5 times. Then use another terminal and enter the command `ps -u "your username" -a` to see that all children are still there as zombies, even if they have been terminated.

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

int main (void) {
  pid_t pid;
  char buf[20];
  printf("Enter input (n = next, other input terminates):");
  fgets(buf, 20, stdin);
  while(buf[0] == 'n') {
    pid = fork();
    if (pid < 0) {
      perror("Fork:");
      exit(1);
    }
    if (pid == 0) {
      sleep(5);  // This represents something real work
      exit(0);   // that is done for the client
    }
    printf("Enter input (n = next, other input terminates):");
    fgets(buf, 20, stdin);
  }
}
```

Phase 2 (Double forking)

Use double forking to solve the zombie-problem. Test the program and see, that the problem of the increasing number of zombies is really solved.

Extra exercise 6b (Process handling, 0.25p)

Part A (Chain of processes)

Write a program that contains 6 processes. The process that is started from the command line (the main process of the system) first creates one child process. This child process also creates a child. The recently created child again creates a child and so on. This means that each process in the system has one child process except the process that was created last (this is the 5th child in the chain).

Each process that has created a child starts to wait for its child immediately after creating it. When a child has terminated, the parent process takes the exit value of it's child, sleeps one second, displays the exit value, adds one to the exit value and returns this value as its own exit value to its parent. The child that has been created lastly only sleeps for a second and returns zero as an exit value to its parent. The whole program terminates of course when all processes have terminated.

Hint. You need again the macro `WEXITSTATUS`.

Part B (The child process runs it's own program file)

Write a program that creates one child. The child process starts to run the program file of its own. The parent process continues its working after it has created the child. The parent is working in the loop as long as its child is working. In this program we simulate the parent's work by displaying the text "Parent is working" once in the second (parent goes to sleep for one second after displaying the text). When the child terminates, the parent first displays the message "Child terminated" and then the contents of the file, which the child has written to the file. The parent itself terminates after that.

The child can write for example 5 times character set AAAAA to the file (25 characters altogether). Put the child to sleep for one second after each write operation so that it is easy to follow, how the whole system works.

Parent process opens the file before the fork, so that the child inherits the opened file descriptor and can use it without opening again. It is necessary to pass the file descriptor number to the new program. This is done using command line parameter. Because file descriptor is integer and command line parameters are strings, the conversion needs to be done.

Hint. Use non-blocking wait in the parent, because then it is possible to do the work at the same time.