Lab 8

Due Nov 3, 2023 by 6:30p.m. **Points** 1

Lab 8: pipe, exec, dup2

Introduction

In an application that requires a user to login, the application must be able to read in a user id and password, and validate it to determine whether the login is successful. One approach to validation is to hand the task off to a separate process.

In this lab, you are given a program that validates user id and passwords, and will apply what you've learned about processes and pipes to create a program that runs this validation program in a child process and reports the result of the validation. To begin, login to MarkUs and go to the **lab8** assignment to obtain the starter files.

1. Understanding the `validate` program

You can find the code for the validation program used for this lab in `validate.c`. The `validate` program reads the user id and password from `stdin`, because if they were given as command-line arguments they would be visible to programs such as `ps` that inspect the state of the system. You are also given a sample file containing user ids and password combinations, which is used by `validate`.

After reading the comments at the top of `validate.c`, you will want to compile it and try running `validate` directly first.

How many bytes does `validate` expect to read in each read call? Does it require the input to be null-terminated? What happens in each case?

Notice that this program doesn't print any output; the only information it provides comes in the exit code of the program.

Use the shell variable `$?` to refer to the exit code of the last process run (e.g., by running `echo $?`).

NOTE: You may not change the validate program.

2. Create the main program

Your task is to complete `checkpasswd.c`, which reads a user id and password from `stdin`, creates a new process to run the `validate` program, sends it the user id and password, and prints a message to `stdout` reporting whether the validation is successful.

Your program should use the exit status of the `validate` program to determine which of the three following messages to print:

- "Password verified" if the user id and password match.
- "Invalid password" if the user id exists, but the password does not match.
- "No such user" if the user id is not recognized

The exact messages are given in the starter code as defined constants.

The only case that should be handled directly in `checkpasswd` is the case where either the `userid` or `password` are too long. This is to prevent sending the wrong number of bytes to validate.

Note that in the given password file `pass.txt`, the "killerwhales:swim" has a user id that is too large, and "monkeys:eatcoconuts" has a password that is too long. The examples are expected to fail, but the other cases should work correctly.

You will find the following system calls useful: `fork`, `exec`, `pipe`, `dup2`, `write`, `wait` (along with `WIFEXITED`, `WEXITSTATUS`). You may **not** use `popen` or `pclose` in your solution.

Important: `execl` arguments

Week 7 Video 6 "Running Different Programs" demonstrates a version of `execl` that takes only two arguments. The signature for `execl` is:

- `int execl(const char *path, const char *arg0, ... /*, (char *)NULL */);`

In the video, the `execl` call only passed two arguments (`execl("./hello", NULL)`), but that shortcut doesn't work on teach.cs. Instead, you need to pass the middle argument (respresenting `argv[0]`) explicitly: `execl("./hello", "hello", NULL)`.

Let's consider two more examples. If you want to call the executable `./giant` with the arguments `fee fi fo`, you would do it like this: `execl("./giant", "giant", "fee", "fi", "fo", NULL);` If you want to call `./giant` with no arguments you would call it like this: `execl("./giant", "giant", NULL);`

Submission

Submit your final `checkpasswd.c` file to MarkUs under the `lab8` folder in your repository.