CSC 412 – Operating Systems

Programming Assignment 04, Fall 2023

Friday, October 12th, 2023

Due date: Sunday, October 22nd, 11:59pm

1 What This Assignment Is About

1.1 Objectives

The objectives of this assignment are for you to get some multiprogramming experience with `fork` and `exec`.

This is an individual assignment.

1.2 Handout

The only handout for this assignment is this document as a `.pdf` file.

2 Part I: Executing a List of Bash Commands

2.1 File organization

All versions of the program, including the ones for extra credit should come in a separate subfolder of the `Programs` folder. These folders should be named `Version1`, `Version2`, `Version3`, `EC1`, etc.

2.2 Format of the data

Your program will get as arguments a list of paths to data files containing a number of single-line bash terminal commands to execute. For example:

```
ls -la
rm *.txt
ps -a
gcc -Wall prog.c -o prog
ls -la /usr/bin
man gcc | grep compiler
du -H -l 2 ..
```

You can expect that

- Each command is written on a single line of the input file (i.e. long commands are not broken into multiple lines using the character at the end of a line);
- The commands don't redirect their output to a file: If they produce an output, it goes to the terminal (standar output).
- The commands don't expect input from a user (and therefore once launched will terminate without user intervention;
- The data files will all have a `.dat` file extension.

2.3 A note on the tasks that we perform

Our focus is on system programming. I want you to get familiar and comfortable with process creation through `fork`, multithreading, interprocess communication, etc. However, we don't `fork()` just for the sake of performing these calls, but to implement a `system` that accomplishes a *complex task*. This leaves me with three options:

- Make the implementation of a meaningful/interesting application a part of the assignment. This means that most of your development time will be spent on "regular programming" rather than system programming.
- Provide a complete application/framework and let you concentrate on system programming aspects. This approach has its virtues, and we will go this route a few times this semester. Its drawback is that students often get lost understanding just what they need in order to complete the task (which is actually an important skill for a programer).
- Use deliberately a "silly task" (or one that could be performed just as well another way, and hope that you will see how this could work if we had a real/interesting task to perform instead.

This assignments belongs clearly to the latter category (we will do some of the second alternative in future assignments). The task we are performing is not particularly interesting or useful. We are just "working the muscle" on the mechanics of `fork` and `exec`, but hopefully you will see how the concepts explored here can be applied to other more relevant contexts soon.

In this case, we are trying to build a complex system by using existing executables. In this case, we use basic Unix executables, but it could be more complex programs, like the small image processing utilities that we did in lab last week.

2.4 Input format for all versions

For all versions of the program, the arguments will be:

- The path to a folder where to redirect output and possibly record a log of operation
- A list of paths to `.dat` data files containing single-line bash commands

2.5 Version 1: No `fork`

The single source file for this version should be named `prog04_v1.c`
The purpose of this version is simply to get a reference output and develop the required utility functions without dealing with the additional complexity (and associated points of failure) of process creation through `fork` and `exec`.

2.5.1 The `system()` function

For this purpose, we are going to use the `system()` function call (defined in `stdlib.h`), which lets one execute system commands within a C/C++ program. For example, the call

```
system("ls -l");
```

will print out to the standard output in "long form" the list of all files in the current working directory.

2.5.2 Output for this version

For each command executed your program should redirect the output to a file in the output folder. That file should be named after the command executed, followed by a unique indicator (of your own design), and the `.txt` extension. For example, if your program was executed with the command

```
./prog03 ./Output ./CommandFiles/file1.dat ./CommandFiles/file2.dat
```

and the command files `file1.dat` and `file2.dat` respectively contain the lists of commands

```
ls -la
rm ../*.txt
ps -a
gcc -Wall prog.c -o prog
du -H -l 2 ..
```

and

```
ls -la ..
rm *.txt
ls ./Output
```

then your program should produce in the output folder files named `ls_<idx>.txt`, `rm_<idx>.txt`, `ps_<idx>.txt`, `gcc_<idx>.txt`, `du_<idx>.txt`, `ls_<idx>.txt`, `rm_<idx>.txt`, and `ls_<idx>.txt`.
where `<idx>` represents a unique index of your own design.

2.6 Version 2: `fork` but no `exec`

The single source file for this version should be named `prog04_v2.c`

In this version, for each command read from an input file, your program should create a child process that will make the appropriate `system()` call.
The output for this version is identical as that of Version 1.

2.7 Version 3: `fork` and `exec`

The single source file for this version should be named `prog04_v3.c`

2.7.1 What to do

In this version, instead of simply making a `system()` call, the child process should execute the command by performing an `exec` call. the parent process should wait for all child processes to complete before terminating.

2.8 Extra credit

2.8.1 Extra Credit 1: Handle comments (3 pts)

This EC can we applied to any of Versions 1, 2, or 3. Some data files may contain lines of bash comments, e.g. the first and third lines in

```
#!/bin/bash
ls -la
# some comments
gcc -Wall someProg.c -o prog
```

You should skip these lines.

Note: There won't be any bash comments at the end of a line of command, e.g.

```
ls -la  # some comments
```

On the other hand, there may be one or more blank characters before the # character indicating the beginning of the comment.

2.8.2 Extra Credit 2: Handle long commands (5 pts)

This EC can we applied to any of Versions 1, 2, or 3. Long bash commands may be broken into multiple lines using the `'\'` character, as in

```
gcc -Wall \
someProg.c \
-o prog
```

Your program should be able to handle such multiple-line commands.

2.8.3 Extra credit 3: Combined log file (5 pts)

This EC can we applied to any of Versions 1, 2, or 3. While the child processes are doing their work, your program should produce a log file named `log_<idx>.txt` (where, again, `<idx>` represents a unique index) that lists all the calls performed by all the child processes. For example, in the case of the command files listed earlier, the log file produced should contain:

```
ls
    ls -la
    ls -la ..
    ls ./Output
rm
    rm *.txt
    rm ../*.txt
ps
    ps -a
gcc
    gcc -Wall prog.c -o prog
du
    du -H -l 2 ..
```

The order in which commands are listed is not important.

3 Part II: `bash` Scripts

As usual, all scripts are expected to be located in the `Scripts` folder located at the root of your project folder, and will be executed from the root of the project folder.

3.1 Command recorder

This bash script named `recorder.sh` is independent from the C part of the assignment. The script will ask the user to enter bash terminal commands. It will first verify that the command doesn't redirect the output to a file and reject commands that do so. If the command is validated, then your script will try to execute it, then ask the user the option to record the command (R), discard it (D), or stop recording (STOP), resulting in the end of the script: the script should ask for a path to a file under which to record the list of commands, write that list to the file, and end execution.

3.2 Watch folder

This script, named `watch.sh`, should first build executables for all the versions of the assignment that you completed. After that it will behave as described below.

The idea for this part of the assignment is that your script should keep watch on a specific folder and process any file or folder that the user drops in that folder:

- If the user dropped a data file (with `.dat` extension), then the script should launch an executable of your program to take care of that file;
- If the user dropped a folder, then your script should fine all the data files (with `.dat` extension) in that folder, and launch an executable of your program will the paths to these data files as list of arguments;
- If the user dropped any other kind of file, then the script should simply ignore that file.

The arguments for this script are, **in this specific order**):

1. The path to a "watch folder." If the folder doesn't already exist, your script should create it;
2. The index of the version of the program to run (1, 2, or 3);
3. The path to an output folder for the results of process.

Your script must be able to handle any proper path. You cannot use hard-coded paths; you cannot assume that the path is relative to the local folder (i.e. the path doesn't necessarily start with `./`; it could very well be a full, absolute path starting from the root of the file system `/`); the path may end with a `/`,. . . or not.

Important note:
This is not a "once and done" type of script. Your script should keep looking for new files or folders added by the user in the drop folder, until the user calls for the end of execution. You must also be careful not to keep processing the same file(s) over and over. It's not simply: "Every n seconds, get a list of all files in the drop folder and launch an executable to process them. You must only process newly added files.

Special note for Mac users:
If you search online, you will see references to a command `inotifywait`. This command does *not exist* on mac OS. In order for your script to work on a Mac, you will need to code your own "busy waiting" solution (sleep a bit and check the folder again).

4 What to Submit

4.1 The pieces

All your work should be organized inside a folder named `Prog04`. Inside this folder you should place:

- Your report;
- A folder named `Programs` containing all the source and header files required to build the program, as well as your bash script;
- A folder named `Scripts` containing the recorder and folder watch scripts.

Please note that we may test your program and script with data other than the ones that you provide, but at least we want to be able to test your code in the same conditions that you did.

4.2 Grading

4.3 Grading

- The C program performs the task specified (execution): 45 points:
  - Version 1: 10 points
  - Version 2: 15 points
  - Version 3: 20 points
- the `bash` script performs the task specified (execution): 25 points
  - Command recorder: 10 points
  - Watch folder: 15 points
- C and `bash` code quality (readability, indentation, identifiers, etc.): 15 points
- Report: 15 points