COEN 346 OPERATING SYSTEMS – FALL 2024

Programming Assignment 1: Creating a SHELL for UNIX

Context

A shell in an operating system is a user interface that provides access to the services of the operating system. It allows users to interact with the system in various ways. We will develop a Command line shell, where users type commands using a keyboard.
The objective is to familiarize yourself with process management.

Learning objectives

This programming assignment evaluates the following learning objectives and graduate attributes:

1. Design and implement simple operating system components, such as a scheduler or a memory management system, using appropriate data structures and algorithms. (ECE-DE-1 and ECE-DE-3)

In this case, we will design and implement a command line shell.

Skeleton code description

You are provided with a skeleton code that reads user input from standard input (readLine in main.c) and parses it into commands (parser.c). It will print an error if a command is incorrectly written. Commands should follow the grammar rules of unix shell language.
To start:

1. Download the starting code from moodle.
2. Compile the code and run it. Test various commands and see the output. The code contains the cmake list files required to compile.
3. Understand the structure cmdline defined in parser.h file. It holds the parsed command.

Tasks

Simple command execution with background execution control (15%)

Start by enabling the execution of one process. For example,

```
Echo coen346
```

Should print coen346 to the terminal.
Calling a command that takes some time to execute, for example: (assuming sleeptest is a compiled c program that sleeps for 1 minute or more)

```
./sleeptest &
```

should run in the background, that is the shell should continue to accept input from the user immediately, but

```
./sleeptest
```

should wait until the completion of the execution before the next command can be issued.

You will use fork(), execvp( ) and wait() system calls for this.

Listing background processes (20%)

Add to your shell a built-in command *jobs* which lists the processes running in the background of your shell with their pid and the command launched. How can you tell if they have finished?

Input and output redirection (15%)

Implement the input (cmd < in) and output (cmd > out) redirection to a file. File redirection either sends output to a file instead of the screen or pulls input from a file instead of a file.
For example:

```
Echo coen346 >out.txt
```

should write coen346 into the file out.txt.
You will need the following system calls : dup() or dup2() close() open()
Test also with commands that take input from a file.

Existing files

Existing files are not automatically truncated by opening for writing. By default, writings overlap what is already written. When redirecting output to an existing file, it will be necessary, just after opening it, to reduce its size to 0 to erase its previous content with the ftruncate function.

Simple pipe (15%)

Enable pipes (|) between two processes. A pipe takes output from one command and uses it as input for another.

```
Cat records.txt | sort
```

Should print the contents of records.txt sorted alphabetically.
You will need the following system calls: pipe(), dup() or dup2() close()

Multiple pipe (20%)

Enable the execution of multiple sequences of pipes, such as:

```
Cat records.txt | sort | uniq
```

The pipes should support file redirection and running in background. If file redirection is used, the input file is used as input for the first command in the pipe and the output file will print the output of the whole pipe.
You will need the following system calls: pipe(), dup() or dup2() close()

Code style guidelines (5%)

1. Your code should compile without errors. It should run correctly on the test data and your own test data.
2. Code clearly, your code should be easy to read for another *human*. Use descriptive variable names. Follow indentation practices. Comment your code using . If you need comments for each line, your code is not well-written. We reserve the right to deduct points if we cannot easily understand your code.

Submission

To submit your assignment:

1. Clean all compiled code from the project.

2. Create a file AUTHORS.txt in the root folder of your project containing your names, last name and student id.
3. Name the folder <S1_id>_<S2_id>_pa1. Where S1_id and S2_id are the student id's of each member of the group.
4. Zip your folder and submit it via Moodle. **ALL members of the group should submit the assignment.**

Grading criteria

1. Code and tests during demo (85%) divided in each question as indicated in the task description.
2. Coding style (5%)
3. Answers in demo - individual (10%)

Academic Honesty

You commit to following the University's Academic Integrity policy. You are permitted and encouraged to discuss ideas with other students. However, you may not share answers or your code with others. **Copying code/solutions from other sources, including online files, or solutions from other educational institutions, but not limited to these, is prohibited**. We will monitor source code for individuality. The Academic Code of Conduct outlines the consequences of cheating, plagiarism or any other dishonest behavior related to academic pursuits.

Use of AI tools for coding

AI is changing programming. Programmers use AI to improve their productivity. You may use Copilot as a collaborator rather than as a replacement, at your own risk. If you do, report the prompts used, and the modifications you made to the code, if any. You are still expected to understand all parts of the code and be able to replicate it. You cannot use AI to write your report.

Suggested Readings: https://spectrum.ieee.org/ai-software

This programming assignment is based on the assignment by prof. Yves Denneulin in Université Grenoble Alpes and adapted with his permission.