CS2002 Practical 5: Parallel Program Running

Report

Details

- **Assignment**: Practical 5: Parallel Program Running
- **Course**: CS2002
- **Date**: April 21, 2017

1 Warning

This submission runs code for creating processes, removing and creating files and semaphores. Please, make sure to be careful with running it.

2 Overview

The purpose of this practical was to implement a selection of unix shell, which would parse, execute and parallelize several programs. As an extension, the submission code deals with a larger selection of grammar for parsing arguments and streams, including single quotes, backslashes, flexible command separators, redirections to streams and empty arguments in empty quotes. A few examples the program can deal with could be:

```
>&2
>&2 ./program 2>&1<file
./program <file>"f>i\&le"
```

3 Design

The design of a UNIX shell is a very careful process, which requires a standard parser, that could be but was not generated with *yacc* and *flex*, executor, which needs to execute a program, and a shell environment, or subshell system [1].

3.1 Shell splitting

The first part of the practical only required a parser and a layout of executor.

First thing to notice is that shell grammar is more complicated than it seems. The grammar for parsing command-line arguments is, if possible, hard to describe as context-free, as we can have something like:

```
command -> command word
word -> space argpart stream |
          space argpart stream argpart |
          space stream argpart
stream_rightarrow space argparg -> stream
...
```

There are many different ways to implement a parser, ranging from a big while loop with a linearly increasing number of flags to a very high level CSG parsing including tokenization, prediction branching and evaluating results. Both ways seemed inappropriate for shell grammar, as the first option will not allow many extensions, and too complicated approach will set too many restrictions on the grammar and require too much code to start with.

Using FSAs seemed to be the best solution: it is very easy to implement yet it is much more flexible for parsing than simpler solutions. Moreover,

it is very useful when working with context-sensitive grammars like the one for describing shell arguments.

3.2 Running commands

In this part, the executor had to use the parsed command, set the command accordingy and execute it cleanly, so that the next command will not fail.

3.3 Running commands in parallel

The design of this part only required parallelizing previous part, with no significant changes other than telling the executor to change redirections.

4 Implementation

4.1 Parser

The parsing would use 3 different FSAs to parse the arguments. The first FSA would be used to distinguish the arguments and streams from each other, and put them into different kinds of storage. The second FSA would parse streams and preset the command structure accordingly. The third FSA seeks the command operation after the command, which is in case of specification a newline. Each character processing is dealt with a special reader structure relevant to the context, which holds the fsa, some buffers and some flags.

The two other tools are used for parsing: buffer and filestream. The buffer is a fixed size array which provides interface for adding a character to a string it contans, popping and so on, and itself takes care of string being terminated and remembering its length. The filestream is a FILE, a buffer of size 8 for storing returned characters (as *ungetc* does not guarantee a return of more than 1 character) and an indicator whether EOF was or not seen.

4.2 Executor

The executor receives an already parsed command, opens all input and output files/streams, forks and, in the child process, duplicates and executes them. After this, both processes close open files and free memory.

4.3 Parallelization

The parallelization required two things: synchronized output and limit on the number of processes/threads it needs to run. As the simplest solution, it seemed reasonable to fork each execution and give it its own id and filename,

retreivable after when the output is synchronized, rather than multithreading, since my code was not thread safe af the time, and a failure of any thread would mean an unclosed semaphore and undeleted files. The synchronization is done by using an integer, which is incremented in a parent process after each forking and named semaphores, which help to keep track of the number of processes currently executing and are shared in the operating system. The synchronization is done after all processes are finished, for convenience, since synchronizing while executing would require a list implementation (since the number of commands is unbounded by design) and shared memory mapping, or something very similar. For debugging purposes each filename has a prefix random to a running program instance.

5 Testing

The submission passes all stacscheck tests without any errors and finishes the parallelization test in 2 seconds.

5.1 Logging

Each parsing error that occured while running the code was tested using logging, which is disabled with *-DNDEBUG*, where the most important steps of the program are thoroughly recorded. For example, while parsing the command, for each symbol the argument FSA will tell which state it is in, and the parser will tell which buffers contain which symbols.

5.2 Argvp with bash/zsh

The program argvp is used for detecting which arguments the executed program sees passed to it, while testing the program or comparing the output with the shell. Zsh and bash were used to parse input into the program.

5.3 FSA dumps

The program fsa_dump gives a sample output for each automaton, showing what kind of input it would expect. In the log file, one can see which states it would end up in after each symbol.

5.4 *_test files

Each program was tested separately using a test file, written to test some features of each program. Note, that since all three programs use almost the same code, one program passing the test means all three are [2].

6 Evaluation

The code achieves all required functionality and deals with the majority of unix shell argument grammar, in several places extending beyond POSIX shell grammar.

7 Conclusion

While doing this practical, I understood how the parsing could be done in C, and how probably the major implemtations of a POSIX shell parse commands (zsh, bash). I have learnt how to use a larger selection of POSIX library, how to debug a multiprocessed program and how to dirtily parse a grammar in C, with returning characters back to file streams; storing things in a buffer without having to implement a dynamic array and using this buffer for almost every temporary storage; structuring a program which does not fail when it has to, and itself the model of interpreting the code rather than compiling was not something I have done before.

7.1 Difficulties

The major difficulties with this practical were testing, parsing and executing. The parsing was difficult because the chosen approach, though it does not lead to too many hacks or code duplication, is hard to test especially because there is too many things to test. The execution required usage of POSIX, which I have not used much before, and I found that difficult to debug, especially when I was waiting for children which have terminated already and thus resetting standard input (I am still not sure why), which is why I switched to named semaphores. Another difficulty was to ignore all errors and make sure the program does not crash further just because the practical specification has explicitly defined the behaviour, so I was setting some values to make sure I am not doing anything impossible and it does not print error messages from everywhere. If I had more time, I would extend it to predictive parser since FSAs don't handle branching very well in terms of tokenization, and implement loops, pipes, and-or, () and {} clauses, variables and functions.

References

[1] Gustavo A. Junipero RodriguezRivera and Justin Ennen, *Introduction to Systems Programming: a Hands-on Approach.* `https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/`.

[2] *Zsh source code.* `https://github.com/zsh-users/zsh`, 2017.