Lab 1. Time travel shell

Introduction

You are a programmer for Big Data Systems, Inc., a company that specializes in large backend systems that analyze big data. Much of BDS's computation occurs in a cloud or a grid. Computational nodes are cheap SMP hosts with a relatively small number of processors. Nodes typically run simple shell scripts as part of the larger computation, and you've been assigned the job of speeding up these scripts.

Many of the shell scripts have command sequences that look like the following (though the actual commands are proprietary):

```
sort < a | cat b - | tr A-Z a-z > c
sort -k2 d - < a | uniq -c > e
diff a c > f
```

In this example, the standard POSIX shell executes the code serially: it waits for the command in the first line to finish before starting the command in the second line. Your goal is to speed this up by running commands in parallel when it is safe to do so. In this example, it is safe to start running the second command before the first has finished, because the second command does not read any file that the first command writes. However, the third command cannot be started until the first command finishes, because the first command writes a file `c` that the third command reads.

Your goal is to write a prototype for a shell that runs code like the above considerably faster than standard shells do, by exploiting the abovementioned parallelism. If this prototype works well, the idea is that you'll later (i.e., after CS 111 is over...) improve the prototype until it is production quality.

To simplify things, your prototype needs to exploit parallelism only at the top level, such as in the examples shown above. It need not parallelize subcommands. For example, it is OK if your prototype executes the following commands in sequence without parallelizing them, because all the commands are subsidiary to the parentheses at the top level:

```
(sort < a | cat b - | tr A-Z a-z > c
 sort -k2 d - < a | uniq -c > e
 diff a c > f)
```

Your company's shell scripts all follow some simple rules which should make the prototype easier to write:

- They limit themselves to a small subset of the shell syntax, described in "Shell syntax subset" below.
- Simple commands in scripts have limited behavior, described in "Time travel limitations on computations" below.
- They don't care about some properties of your implementation, and will work regardless of how your implementation behaves in these areas, as described in "Don't care behaviors" below.

Your implementation will take three phases:

- In Lab 1a, you'll warm up by implementing just the shell's command reader. This shell will support a `-p` option, so that the command "`timetrash -p script.sh`" will read the shell commands in the file `script.sh` and output them in a standard format that is already supplied by a code skeleton available on CourseWeb; sample output is in the skeleton's test script`test-p-ok.sh`.
- In Lab 1b, you'll implement the standard execution model for your shell subset. Once this is done, the command "`timetrash script.sh`" should behave like the standard command "`sh script.sh`", assuming `script.sh` is in the shell subset described in this assignment.
- In Lab 1c, you'll implement the time-traveling execution model, which allows extra parallelism. Once this is done, the command "`timetrash -t script.sh`" should execute the script faster than the standard shell does, if the script has enough inherent parallelism.

Implementation

A skeleton implementation will be given to you on CourseWeb. It comes with a makefile that supports the following actions. Your solution should have similar actions in its makefile.

- "`make`" builds the `timetrash` program.
- "`make clean`" removes the program and all other temporary files and object files that can be regenerated with "`make`".
- "`make check`" tests the `timetrash` program on the available test cases. The initial test cases are just for Lab 1a, and they fail on the skeleton code because the skeleton code doesn't do anything useful. Your program should succeed on them. For Lab 1b and 1c, you should add two test cases each, in the same style as the existing cases for 1a.

- "`make dist`" makes a software distribution tarball `lab1-`*`yourname`*`.tar.gz` and does some simple testing on it. This tarball is what you should submit via CourseWeb.

Your solution should be written in the C programming language. Stick with the programming style used in the skeleton, which uses standard GNU style for C. Your code should be robust, for example, it should not impose an arbitrary limit like 2^{16} bytes on the length of a token. You may use the features of C11 as implemented on the SEASnet GNU/Linux servers. Please prepend the directory `/usr/local/cs/bin` to your PATH, to get the versions of the tools that we will use to test your solution. Your solution should stick to the standard GNU C library that is installed on SEASnet, and should not rely on other libraries.

You can run your program directly by invoking, for example, `./timetrash -p foo`. Eventually, you should put your own test cases into a file `test`*`something`*`.sh` so that it is automatically run as part of "`make check`".

More details on syntax and semantics of the shell subset

The "time travel" feature of your shell is feasible partly because of the restricted subset of the shell that you need to implement. Also, for this assignment, the shell has been simplified further so as to avoid some work that can be deferred until a production version.

Shell syntax subset

Your implementation of the shell needs to support only the following small subset of the standard POSIX shell grammar:

- Words, consisting of a maximal sequence of one or more adjacent characters that are ASCII letters (either upper or lower case), digits, or any of: `! % + , - . / : @ ^ _`
- The following eight special tokens: `; | && || ( ) < >`
- Simple commands, which are sequences of one or more words. The first word is the file to be executed.
- Subshells, which are complete commands surrounded by `( )`.
- Commands, which are simple commands or subshells followed by I/O redirections. An I/O redirection is possibly empty, or < followed by a word, or > followed by a word, or <followed by a word followed by > followed by a word.
- Pipelines, which are one or more commands separated by `|`.

- And-ors, which are one or more pipelines separated by `&&` or `||`. The `&&` and `||` operators are left-associative and have the same operator precedence.
- Complete commands, which are one or more and-ors each separated by a semicolon or newline, and which are optionally followed by a semicolon. An entire shell script is a complete command.
- Comments, each consisting of a `#` that is not immediately preceded by an ordinary token, followed by characters up to (but not including) the next newline.
- White space consisting of space, tab, and newline. Newline is special: as described above, it can substitute for semicolon. Also, although white space can ordinarily appear before and after any token, the only tokens that newlines can appear before are `(`, `)`, and the first words of simple commands. Newlines may follow any special token other than < and >.

If your shell's input does not fall within the above subset, your implementation should output to stderr a syntax error message that starts with the line number and a colon, and should then exit.

Your implementation can have undefined behavior if any of the following features are used. In other words, our test cases won't use these features and your program need not diagnose an error if these features are used.

- Shell reserved words such as `!`, `{`, `if`, and `function`, when used as the first word of a command.
- Commands that invoke special built-in utilities such as `break`, `.`, and `exit`. Exception: your implementation should support the special-builtin utility `exec` with a command and optional arguments (you need not support `exec` without a command).
- A token consisting entirely of digits, immediately before < or > (for example, as in the command "`cat 2>/dev/null`").
- Two adjacent left parentheses `((` – see Token Recognition for why.

Time travel limitations on computations

Your implementation of the shell, when it is run in time-travel mode, can have undefined behavior if any of the following limitations are violated. When your shell is run in normal mode, it should not impose these limitations.

- Simple commands read only from standard input or from files whose names are one of the words of the command. For example, the simple command

"`./doit -f x nobody@gmail.com`" reads at most from standard input and from the files `./doit`, `-f`, `x`, and `nobody@gmail.com`, if these files exist.
- The computation never accesses the same file via two different names. For example, if the computation uses the file name `./doit` and the file name `doit` without the leading "`./`", the behavior is undefined.
- Simple commands write only to standard output, to standard error, or to files that are never otherwise accessed by the computation. For example, the `mv` and `rm` commands cannot be used to mess up dependency checking, because they can cause dependency problems only by modifying directories that are later searched.

Don't care behaviors

Similarly, in some cases, your company's scripts don't care how your implementation behaves, and it's OK for it to depart from established semantics when it is run in time-travel mode.

- It is OK if commands behave as if the time of day jumps around at random. For example, it is OK if "`(date >A; date >B)`" puts an earlier time stamp into `B` than into `A`. It is this property of your implementation that prompts the nickname "time travel shell".
- If a set of commands all read from standard input, or write to standard output or standard error, and have no other dependencies that interfere with each other, your implementation can run them in parallel and interleave their reads and writes arbitrarily. For example, "`tr A B; tr A C`" can run two instances of `tr` in parallel, both reading from standard input and writing to standard output; the combination somewhat-randomly transforms some As to Bs and other As to Cs as it copies input to output and it does not necessarily output blocks in the same order that they were input.

You can simplify your shell in one other way, regardless of whether it is run in time-travel mode:

- It is OK if your shell attempts to execute the following commands as regular commands, finding them via the PATH environment variable and running them as executables in a separate child process, even if the commands do not exist in the PATH, and even though POSIX does not allow this behavior: `false` `fc` `fg` `getopts` `jobs` `kill` `newgrp` `pwd` `read` `trueumask` `unalias` `wait`

Submit

After you implement Lab 1a, submit via CourseWeb the `.tar.gz` file that is built by "`make dist`". Similarly for 1b and 1c. Your submission should contain a README file that briefly describes known limitations of your code and any extra features you'd like to call our attention to.

We will check your work on each lab part by running it on the SEASnet GNU/Linux servers, so make sure they work on there. Lab 1 parts are due at different times, but we will not grade each part separately; the lab grade is determined by your overall work on all three parts.

Design problem ideas

Here are some suggestions for design problems, if you have been assigned a design problem for Lab 1. You may implement one of them, or design your own. If you design your own, get approval from us before committing significant work to it. Your implementations should include test cases.

For Lab 1a:

- The redirection operators `>>`, `<&`, `>&`, `<>`, `>|`. The last operator requires that you also implement the `-C` option. Improve on these in at least one way that is not already present in Bash.
- The control structures `if`, `for`, `while`, `until`, and `!`. Improve on these in at least one way that is not already present in Bash.
- Make your shell interactive, with a prompt, and command-line editing, and tab completion, and have it do something reasonable when you type control-C. You can use the GNU readline library for this.

For Lab 1b:

- Debugging in the style of Bash's `-v` and `-x` options. Improve on them in at least one way.
- Add a way to measure resource consumption. The usual `time` builtin works only on simple commands; fix it so that it works arbitrary commands, such as parenthesized commands. Also, fix it so that it counts the number of subprocesses created as well as CPU time and memory consumption.

For Lab 1c:

- Remove one of the significant limitations on shell scripts that are imposed above; that way, we can support more shell scripts in time-travel mode. For example, can you support commands like `sort -o foo`, which output to `foo` instead of inputting from `foo`?
- Limit the amount of parallelism to at most *N* subprocesses, where *N* is a parameter that you can set by an argument to the shell.