Lab 1. Simpleton shell

[111 home > syllabus]

Introduction

You are a programmer for Big Data Systems, Inc., a company that specializes in large backend systems that analyze big data. Much of BDSI'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 improving the infrastructure for these scripts.

Many of the shell scripts have command sequences that look like the following (though the actual commands tend to be more proprietary):

```
cd data
(sort < a | cat b - | tr A-Z a-z > c) 2>> d
```

This command changes the working directory to `data` and then invokes three subcommands. The first runs the command `sort` with standard input being the file `a` and standard output being a unnamed pipe 1. The second runs the command `cat b -` with standard input being pipe 1 and standard output being pipe 2. The third runs the command `tr A-Z a-z` with standard input being pipe 2 and standard output being the file `c`. All three commands have standard error sent, via the same file descriptor, to file `d` in append-only mode. Since all commands are run in the `data` directory, the actual file names are `data/a`, `data/b`, `data/c` and `data/d`.

BDSI's developers have several complaints about these shell scripts:

- The shell script syntax does not let them open files with special flags available at the system call level. For example, there is no way to open a file with the `O_DSYNC` flag of the `open` system call.
- The developers want to be able to create arbitrary directed graphs of processes connected via pipes, but the shell syntax forces them into pipelines.
- The developers want a simpler way to invoke the shell, one that is more easily generated from their programs and scripts, one that does not require that they write a shell script. They do not mind if the simpler shell is harder for humans to use, because it's not intended to be used directly by programmers.

Basic idea

To address these issues, your boss proposes a new program `simpsh`, short for "SIMPleton SHell", a very simple, stripped down shell. `simpsh` does not use a scripting language at all, and you do not interact with it at a terminal or give it a script to run. Instead, developers invoke the `simpsh` command by passing it arguments telling it which files to access, which pipes to create, and which subcommands to invoke. It then creates or accesses all the files and creates all the pipes processes needed to run the subcommands, and reports the processes's exit statuses as they exit.

For example, the abovementioned command in the standard shell could be run using the following `simpsh` command. This invocation uses standard shell syntax, because it is invoking `simpsh` from the standard shell; the command itself, though, is just an array of strings and `simpsh` interprets this array and executes the same three subcommands that the abovementioned shell command does.

```
simpsh \
  --chdir data \
  --rdonly a \
  --pipe \
  --pipe \
  --creat --trunc --wronly c \
  --creat --append --wronly d \
  --command 3 5 6 tr A-Z a-z \
  --command 0 2 6 sort \
  --command 1 4 6 cat b - \
  --close 2 \
  --close 4 \
  --wait
```

This example invocation first changes the working directory to `data`, and then creates seven file descriptors:

0. A read only descriptor for the file `data/a`, created by the `--rdonly` option as modified by the preceding `--chdir`.
1. The read end of the first pipe, created by the first `--pipe` option.
2. The write end of the first pipe, also created by the first `--pipe` option.
3. The read end of the second pipe, created by the second `--pipe` option.
4. The write end of the second pipe, also created by the second `--pipe` option.
5. A write only descriptor for the file `data/c`, created by the first `--wronly` option as modified by the preceding `--chdir`, `--creat` and `--trunc` options.
6. A write only, append only descriptor for the file `data/d`, created by the `--wronly` option as modified by the preceding `--chdir`, `--creat` and `--append` options.

It then creates three subprocesses, each with working directory `data`:

- A subprocess with standard input, output, and error being the file descriptors numbered 3, 5, and 6 above. This subprocess runs the command `tr` with the two arguments `A-Z` and `a-z`.
- A subprocess with standard input, standard output, and standard error being the file descriptors numbered 0, 2, and 6 above, respectively. This subprocess runs the command `sort` with no arguments
- A subprocess with standard input, output, and error being the file descriptors numbered 1, 4, and 6 above. This subprocess runs the command `cat` with the two arguments `b` and `-`.

It then closes the write ends of the pipes, and waits for all three subprocesses to finish. As each finishes, it outputs "`exit` *`N`*" if it exited with status *N* or "`signal` *`S`*" if it terminated with signal number *S*, followed by the command and arguments. The output might look like this:

```
exit 0 sort
exit 0 cat b -
exit 0 tr A-Z a-z
```

although not necessarily in that order, depending on which order the subprocesses finished.

`simpsh` options

Here is a detailed list of the command-line options that `simpsh` should support. Each option should be executed in sequence, left to right.

First are the file flags. These flags affect the next file that is opened. They are ignored if no later file is opened. Each file flag corresponds to an *oflag* value of [`open`]; the corresponding *oflag* value is listed after the option. Also see [Opening and Closing Files] and [Open-time Flags].

`--append`
: `O_APPEND`

`--cloexec`
: `O_CLOEXEC`

`--creat`
: `O_CREAT`

`--directory`
: `O_DIRECTORY`

`--dsync`
: `O_DSYNC`

`--excl`
: `O_EXCL`

`--nofollow`
: `O_NOFOLLOW`

`--nonblock`
: `O_NONBLOCK`

`--rsync`
: `O_RSYNC`

`--sync`
: `O_SYNC`

`--trunc`
: `O_TRUNC`

Second are the file-opening options. These flags open files. Each file-opening option also corresponds to an *oflag* value, listed after the option. Each opened file is given a file number; file numbers start at 0 and increment after each file-opening option regardless of whether file-opening action is successful. Normally file numbers increment by 1, but the `--pipe` option causes them to increment by 2.

`--rdonly` *`f`*
: `O_RDONLY`. Open the file *f* for reading only.

`--rdwr` *`f`*
: `O_RDWR`. Open the file *f* for reading and writing.

`--wronly` *`f`*
: `O_WRONLY`. Open the file *f* for writing only.

`--pipe`
: Open a pipe. Unlike the other file options, this option does not take an argument. Also, it consumes two file numbers, not just one.

Third are the subcommand options:

`--command` *`i o e cmd args`*
: Execute a command with standard input *i*, standard output *o* and standard error *e*; these values should correspond to earlier file or pipe options. The executable for the command is *cmd* and it has zero or more arguments *args*. None of the *cmd* and *args* operands begin with the two characters "`--`".

`--wait`
: Wait for all commands to finish. As each finishes, output its exit status or signal number as described above, and a copy of the command (with spaces separating arguments) to standard output.

Finally, there are some miscellaneous options:

`--chdir` *`dir`*
: Change the working directory to *dir*. This affects the interpretation of all later options, including later `--chdir` options. For example, if `foo` is a directory (not a symbolic link), the two successive options "`--`

`chdir foo --chdir ..`" taken together are a no-op if successful.

`--close` *N*
: Close the *N*th file that was opened by a file-opening option. For a pipe, this closes just one end of the pipe. Once file *N* is closed, it is an error to access it, just as it is an error to access any file number that has never been opened. File numbers are not reused by later file-opening options.

`--verbose`
: Just before executing an option, output a line to standard output containing the option. If the option has operands, list them separated by spaces. Ensure that the line is actually output, and is not merely sitting in a buffer somewhere.

`--profile`
: Just after executing an option, output a line to standard output containing the resources used. Use `getrusage` and output a line containing as much useful information as you can glean from it.

`--abort`
: Crash the shell. The shell itself should immediately dump core, via a segmentation violation.

`--catch` *N*
: Arrange for signal *N*, where *N* is a decimal integer, to be caught by a handler that outputs the diagnostic `N caught` to stderr, and exits with status *N* (i.e., exiting the entire shell). *N* uses the same numbering as your system; for example, on GNU/Linux, a segmentation violation is signal 11.

`--ignore` *N*
: Arrange for signal *N* to be ignored.

`--default` *N*
: Arrange for the default behavior to occur if signal *N* arrives.

`--pause`
: Pause, waiting for a signal to arrive.

When there is a syntax error in an option (e.g., a missing operand), or where a file cannot be opened, or where is some other error in a system call, `simpsh` should report a diagnostic to standard error and should continue to the next option. However, `simpsh` should ignore any write errors to standard error, so that it does not get into an infinite loop outputting write-error diagnostics.

When `simpsh` finishes other than in response to a signal, it should finish with the worst result of all the subcommands that it ran and successfull waited for. By "worst" is meant the following: if any subcommand terminated with a signal, `simpsh` should terminate with the highest-numbered signal that a subcommand terminated with; otherwise `simpsh` should exit with status equal to the maximum of all the exit statuses of all the subcommands. If there are no such subcommands, or if no subcommands terminated with a signal and the maximum subcommand exit status is zero, `simpsh` should exit with status 0 if all options succeeded, and with status 1 one of them failed. For example, if a file could not be opened, `simpsh` must exit with nonzero status.

Implementation

Your implementation will take three phases:

- In Lab 1A, you'll warm up by implementing just the options `--rdonly`, `--wronly`, `--command`, and `--verbose`.
- In Lab 1B, you'll implement the rest of the options, except for `--profile`.
- In Lab 1C, you'll implement `--profile` and compare the performance of your implementation to that of `bash` and that of `dash`. Design three nontrivial benchmarks of your own, and translate them from `simpsh` to POSIX shell form so that they can be run on `bash` and `dash`, and time all three benchmarks on all three implementations. Run each benchmark three times and take the average of the user + system CPU times, counting all overhead of starting and finishing the shell. Each `bash` and `dash` benchmark should use the

`times` command at the end to output CPU times for the shell and its children, and you should compare that to the `--profile` output of the `simpsh` benchmark.

Before charging ahead and implementing, you should be familiar with the man pages for `close`, `dup2`, `execvp`, `fork`, `getopt_long`, `open`, `pipe`, and `sigaction`.

Your program should come with a file named `Makefile` that supports the following actions.

- '`make`' builds the `simpsh` program.
- '`make clean`' removes the program and all other temporary files and object files that can be regenerated with '`make`'.
- '`make check`' tests the `simpsh` program on test cases that you design. You should have at least three test cases.
- '`make dist`' makes a software distribution compressed tarball `lab1-`*`YourID`*`.tar.gz` and does some simple testing on it. *`YourID`* should be your nine-digit tudent ID, without any hyphens. This tarball is what you should submit via CCLE. All the files in the tarball should have names of the form `lab1-`*`YourID`*`/...` and one of the files should be `lab1-`*`YourID`*`/Makefile`.

Your solution should be written in the C programming language. Your code should be robust, for example, it should not impose an arbitrary limit like 2^{16} bytes on the length of a string. You may use the features of C18 as implemented on the SEASnet GNU/Linux servers running RHEL 7. 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 test your program by running it directly. Eventually, you should put your own test cases into a file `test.sh` and run it automatically as part of '`make check`'.

Submit

After you implement Lab 1A, submit via CCLE 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.
