title: CS 537 Project 3 layout: default

CS537 Fall 2024, Project 3

Updates

- TBD

Administrivia

- **Due Date** by October 8, 2024 at 11:59 PM
- Questions: We will be using Piazza for all questions.
- Collaboration: The assignment has to be done by yourself. Copying code (from others) is considered cheating. Read this for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- This project is to be done on the lab machines, so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- A few sample tests are provided in the project repository. To run them, execute `run-tests.sh` in the `tests/` directory. Try `run-tests.sh -h` to learn more about the testing script. Note these test cases are not complete, and you are encouraged to create more on your own.
- Handing it in: Copy the whole project, including solution and tests folder, to `~cs537-1/handin/$login/p3` where `$login` is your CS login.
- **Slip Days**:
  - In case you need extra time on projects, you each will have 2 slip days for the first 3 projects and 2 more for the final three. After the due date we will make a copy of the handin directory for on time grading.
  - To use a slip days or turn in your assignment late you will submit your files with an additional file that contains only a single digit number, which is the number of days late your assi s which contain one of these `slipdays.txt` files. This file e your code.
  - We will track your slip days ar you have used up your slip days.
  - After using up your slip days y s late, but for any single assignment we won't accept submissions aft l slip days on a single assignment you can only submit that assignm
  - Any exception will need to be
  - Example of `slipdays.txt` :

```
$ cat slipdays.txt
1
```

⚠Warnings⚠

- This project will be checked for mem
- This project will be examined during i
- This project is time consuming, start

Before you start - Makefil

The first task in this project is to create a `Ma` n your project and execute them using `make` in a simple way. You can read more about `make` and `Makefile` s in GNU's Make Manual, Makefile Tutorial and Lab Tutorial.

Your `Makefile` must include at least following variables:

- `CC` specifying the compiler. Please use `gcc` or `clang` .
- `CFLAGS` specifying the arguments for compilation. Use at least the following: `-Wall -Wextra -Werror -pedantic -std=gnu18`
- `LOGIN` specifying you login.
- `SUBMITPATH` specifying the path where you handin your files.
- `$@` has to be used at least once.
- `$<` or `$^` has to be used at least once.

Your `Makefile` must include at least following targets:

- `all` is the first target in your `Makefile` and runs `wsh` and `wsh-dbg` targets. `all` is a `.PHONY` target. Creating `all` target as a first target is a common convention, since the first target is executed when `make` is called without a target.
- `wsh` is a target which depends on `wsh.c` and `wsh.h` and builds the `wsh` binary with the compiler specified in `CC` and compiler flags in `CFLAGS` . The compilation must produce `-O2` optimized binary. Hence `make wsh` will compile your code and create `wsh` binary.
- `wsh-dbg` is a target which depends on `wsh.c` and `wsh.h` and builds the `wsh-dbg` binary with the compiler specified in `CC` and compiler flags in `CFLAGS` . This binary is not optimized and is to be used for debugging with `gdb` . I.e. use `-Og -ggdb` flags. `make wsh-dbg` will compile your code and create `wsh-dbg` binary.
- `clean` removes binaries from the current directory. I.e. it just keeps source files. Must be called before submission.
- `submit` target automatically submits your solution according to the submission instructions above. This means, that you should submit your project simply by typing `make submit` .

We encourage you to create your own simple tests while developing your shell. It is **very helpful** to create a `test` target in your `Makefile`, which will compile your code and run all your tests. Like this, you can speed up your development and make sure, that every change in your source code still passes your tests (i.e. after every change of you source code, you can just type `make test` and the shell will be compiled and tested).

Before beginning: Read `man 3p fork` and `man 3p exec`.

Unix Shell

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Overview

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shell you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably `bash` or `zsh` (try `echo $SHELL`). One thing you should do on your own time is to learn more about your shell, by reading the man pages or other online materials. Also, when you are in doubt about some behavior in this assignment, try the behavior in `bash` before you ask. Maybe it makes things clear. Or not, and you will come to office hours (preferably) or ask on Piazza.

Program Specifications

Basic Shell: `wsh`

Your basic shell, called `wsh` (short for Wisco `wsh>` (note the space after the greater-than sign), parses the input, executes This is repeated until the user types `exit`. The name of your final executable sh

The shell can be invoked with either no argu

```
prompt> ./wsh
wsh>
```

At this point, `wsh` is running, and ready to ac

The mode above is called *interactive* mode, a de, which instead reads input from a batch file and executes commands from the

```
prompt> ./wsh script.wsh
```

One difference between batch and interactiv mpt should be printed.

You should structure your shell such that it c s, discussed below). Your basic shell should be able to parse a command and run `/tmp /tmp2`, your shell should run the program `/bin/cp` with the given arguments `-r`, `/tmp` and `/tmp2` (how does the shell know to run `/bin/cp`? It's something called the shell **path**; more on this below).

Structure

Basic Shell

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits. That's it!

For reading lines of input, you should use `strtok()` and we guarantee that each token is delimited by a single space. Generally, the shell will be run in *interactive mode*, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support *batch mode*, in which the shell is given an input file of commands; in this case, the shell should not read user input (from `stdin`) but rather from this file to get the commands to execute.

In either mode, if you hit the end-of-file marker (EOF), you should call `exit(0)` and exit gracefully. EOF can be generated by pressing `Ctrl-D`.

To execute commands, look into `fork()`, `exec()`, and `wait()/waitpid()`. See the man pages for these functions, and also read the relevant book chapter for a brief overview.

You will note that there are a variety of commands in the `exec` family; for this project, you must use `execv`. You should **not** use the `system()` library function call to run a command. Remember that if `execv()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

Comments and executable scripts

In your shell, you should ignore all lines starting with `#`. Note that there can be spaces (` `) in front of `#`. These lines serve as comments in most shells you will work with (`bash, zsh`).

Furthermore, once you implement comments, you should be able to create `wsh` script, which can be directly executed. For example, if you put following script (let's call it `script.wsh`) into a directory with your compiled `wsh` binary, you must be able to run the script by typing `./script.wsh`.

```
 $ cat > script.wsh <<EOF
#!./wsh

echo hello
EOF

$ chmod +x script.wsh
$ ./script.wsh
hello
```

If you are curious, the first line in the script (`#!./wsh`) is called shebang and it tells OS how to deal with this executable. There is a wiki page about it.

Redirections

Our shell will also support redirections as for example `bash` does. Please check Redirections in Bash Manual to learn about this powerful feature. To simplify the assignment, `wsh` supports only following and we guarantee, that the redirection token is always the last one on the command line, i.e. after all the command parameters. Also there can be a at most one redirection per command.

- Redirecting Input. The token always l
- Redirecting Output. The token always
- Appending Redirected Output. The to
- Redirecting Standard Output and Star
- Appending Standard Output and Stan `>`.

Examples of redirection:

```
wsh> echo hello &>>log.txt
```

```
wsh> cat <input.txt
```

Environment variables and shell v

Every Linux process has its set of environme d use `man environ` to learn more about this.

Some important things about environment ar

1. When `fork` is called, the child proce
2. When a system call from the `exec` f environment or a user specified environment depending on the exact
3. There are functions such as `getenv` ocess. See the `man environ` for more details.

Shell variables are different from environment variables. They can only be used within shell commands, are only active for the current session of the shell, and are not inherited by any child processes created by the shell.

We use the built-in `local` command to define and set a shell variable:

```
local MYSHELLVARNAME=somevalue
```

The variable never contains space (` `) hence there is no need nor special meaning of quotes (`""`).

This variable can then be used in a command like so:

```
cd $MYSHELLVARNAME
```

which will translate to

```
cd somevalue
```

In our implementation of shell, a variable that does not exist should be replaced by an empty string. An assignment to an empty string will clear the variable.

Environment variables may be added or modified by using the built-in `export` command like so:

```
export MYENVVARNAME=somevalue
```

After this command is executed, the `MYENVVARNAME` variable will be present in the environment of any child processes spawned by the shell.

Variable substitution: Whenever the `$` sign is used in a command, it is always followed by a variable name. Variable values should be directly substituted for their names when the shell interprets the command. Tokens in our shell are always separated by white space, and variable names and values are guaranteed to each be a single token. For example, given the command `mv $ab $cd,` , you would need to replace variables `ab` and `cd` . If a variable exists as both the environment variable and a shell variable, the environment variable takes precedence.

You can assume the following when handling variable assignment:

- There will be at most one variable assignment per line.
- Lines containing variable assignments will not include pipes or any other commands.
- The entire value of the variable will be present on the same line, following the `=` operator. There will not be multi-line values; you do not need to worry about quotation marks surrounding the value.
- Variable names and values will not contain spaces or `=` characters.
- There is no limit on the number of variables you should be able to assign.

Displaying Variables: The `env` utility program (not a shell built-in) can be used to print the environment variables. For local variables, we use a built-in command in our shell called `vars` . Vars will print all of the local variables and their values in the format `<var>=<value>` , one variable per line. Variables should be printed in insertion order, with the most recently created variables printing last. Updates to existing variables will modify them in-place in the variable list, without moving them around in the list. Here's an example:

```
 wsh> local a=b
wsh> local c=d
wsh> vars
a=b
c=d
wsh>
```

Paths

In our original example in the beginning, the u es your shell know this?

It turns out that the user must specify a **path** ctories that comprise the path are sometimes called the search path of the she user types a command.

Important: Note that the shell itself does not utables in one of the directories specified by path and create a new process t

To check if a particular file exists in a directo user types `cp` , and path is set to include both `/usr/bin` and `/bin` (assumin `/usr/bin/cp", X_OK)` . If that fails, try `/bin/cp` . If that fails too, it is an error.

Your initial shell path should contain one dire

Of course, your shell still can execute progra `n/ls` or `cd /usr; bin/ls` .

In general, the priority what to execute is foll

1. Built-in.
2. Full or relative path.
3. Paths specified by `$PATH` .

History

Your shell will also keep track of the last five commands executed by the user. Use the `history` builtin command to show the history list as shown here. If the same command is executed more than once consecutively, it should only be stored in the history list once. **The most recent command is number one.** Builtin commands should not be stored in the history.

```
 wsh> history
1)  man sleep
2)  man exec
3)  rm -rf a
4)  mkdir a
5)  ps
```

By default, history should store up to five commands. The length of the history should be configurable, using `history set <n>` , where `n` is an integer. If there are fewer commands in the history than its capacity, simply print the commands that are stored (do not print blank lines for empty slots). If a larger history is shrunk using `history set` , drop the commands which no longer fit into the history.

To execute a command from the history, use `history <n>` , where `n` is the nth command in the history. For example, running `history 1` in the above example should execute `man sleep` again. Commands in the history list should not be recorded in the history when executed this way. This means that successive runs of `history n` should run the same command repeatedly.

If history is called with an integer greater than the capacity of the history, or if history is called with a number that does not have a corresponding command yet, it will do nothing, and the shell should print the next prompt.

Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your wsh source code, which then will exit the shell.

Here is the list of built-in commands for `wsh` :

- `exit` : When the user types exit, your shell should simply call the `exit` system call with 0 as a parameter. It is an error to pass any arguments to `exit` .
- `cd` : `cd` always take one argument (0 or >1 args should be signaled as an error). To change directories, use the `chdir()` system call with the argument supplied by the user; if `chdir` fails, that is also an error.
- `export` : Used as `export VAR=<value>` to create or assign variable `VAR` as an environment variable.
- `local` : Used as `local VAR=<value>` to create or assign variable `VAR` as a shell variable.
- `vars` : Described earlier in the "environment variables and shell variables" section.
- `history` : Described earlier in the history section.
- `ls` : Produces the same output as `LANG=C ls -1` , however you cannot spawn `ls` program because this is a built-in. This built-in does not implement any parameters.

Miscellaneous Hints

Remember to get the basic functionality of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as `ps`).

Next, add built-in commands. Then, try working on command history, redirections, and variables. Each of these requires a little more effort on parsing, but each should not be too hard to implement. It is recommended that you separate the process of parsing and execution - parse first, look for syntax errors (if any), and then finally execute the commands.

We simplify the parsing by having a single sp ll be delimited by a single space in our tests.

Check the return codes of all system calls fro king these new system calls. It's also just good programming sense.

Beat up your own code! You are the best (an ake sure the shell behaves well. Good code comes through testing; you must run m ers certainly won't be.

Finally, keep versions of your code. More adv d to push the repository to the Internet, but you can still do commits and benefit fron piece of functionality working, make a copy of your `.c` file (perhaps a subdirectory around, you can comfortably work on adding new functionality, safe in the knowled

Error conditions should result in `wsh` termin .