CSC458 - Operating Systems

Operating systems with Dr. Karlsson

Exams - A Diagnostic Shell

This is a shell called `dash` that is used for system process diagnostics and experimentation. The shell is executed using the `./dash` command after which you can do your usual day to day shell operations at the `dash>` prompt. Pipes and file redirect are supported as well as basic memory management and process management. Certain system process info is printed before and after issuing commands for your understanding benefit.

Algorithms/Libraries

I decided to use C++ so I could have access to more modern lanuage features.

For command parsing, I used the C++ string class and a string stream for tokenization.

I used standard system calls for all other operations to keep dependencies to a minimum.

Functions/Program Structure

main.cpp

This file contains the main loop, user input, various helpful constants, and a function useful for splitting a string into its tokens.

dash.cpp

The main bread and butter of the program, this contains a class `Dash` that contains one public function:

`run(vector<string> tokens)` - this commmand accepts tokenized input from `main.cpp`, and runs the command. It verifies if the command is unique to `dash`, in which case it runs the desired commands for desired output, or if it is a general system command. Each dash specific call should call a function that is private to the Dash class.

The private functions:

`getInfo(string value)` - this is a private function that runs informational commands how all my commands should have been run but I wasn't smart enough/was too lazy until the end of writing the program. Essentially takes in a destination and cats it to the output stream. This is used for getting cpuinfo, meminfo, kernel version, and uptime.

`printCommandName(vector<string> tokens)` - this is borrowed from exam 1, and prints the name of the command entered.

`printPID(vector<string> tokens)` - thie is borrowed from exam 1 and prints the process id of the command entered

`changeDir(vector<string> tokens)` - this is borrowed from exam 1 and allows Dash to change directories, allowing for absolute and relative paths.

`printStats()` - this is borrowed from exam 1 and allows the user to print basic stats about the system being used including linux kernel version, uptime, memory and CPU info.

`getInfo(string value)` - borrowed from exam 1, helper that gets the info from the /proc folder for the info above. Accepts a value in `/proc` folder to print.

`workingDirectory()` - borrowed from exam 1, returns a string of the current working directory, used for relative cd in the `changeDir` command.

Exam 2

`memman(vector<string> tokens)` - A useful function that assumes a system with 32-bit virtual adressing with a 4kb page size. It prints out the page number and offset for the inputted memory address. An example of usage is `memman 19986`.

`handleCommand(vector<string> tokens)` - this function accepts tokens from the command line and analyzes them, determining how they are to be run. If there is a pipe or a file redirect, the appropriate method to handle that will be called, otherwise it will be handled as a standalone command.

`handleRedirectCommand()` - this function will take care of commands of the form `command -args > file.extension`. Any output that the original command generates will be redirected into the file instead of standard out.

`handlePipedCommand()` - this function takes care of piped commands of the form `cmd1 -args | cmd2 -args`. It uses a pipe and redirected stdout to take the output from command one and redirect it as the input to command two. It essentially forks twice and runs the first command in the grandchild, reroutes the output to the pipe, waits and backs out to just the child, and then accepts the input from the pipe for the second command. HOWEVER, this one isn't working quite right. It seems to hang somewhere in the transition betweent the grandchild and the child and I am really not quite sure why. So I guess count it as a bug 😦

`handleSingleCommand(vector<string> tokens)` - this is the fall back for an ordinary command that doesn't do any fancy pipes, redirects, or whatnot. This is largely also borrowed from exam 1, just here because it was rearranged (for the better).

PID Manager

There is a PID manager that has associated required functions from your API you laid out. It also has a testPID() function that runs a bunch of threads and tests the process management capabilities of the functions I wrote. It is thread safe, locking and unlocking when necessary.

makefile

Just your good old, classic makefile that got made, scaled to really cool modular structure, didn't work, messed with it for a day or two, didn't get anywhere, and converted back to just your regular good ol' classic makefile again. Tradition!. It does have a couple of targets now though. Here they are:

- clean - classic cleanup

- dash - builds the dash shell, described above
- manage - builds the PID manager test program, which can be run with `./manage`

PID Manager

Compilation/Usage

Build

As stated above, the good ol' regular make file should build the Dash thing just fine with:

```
make
```

and should clean up the whole thing just fine with:

```
make clean
```

And builds the PID manager with:

```
make manage
```

I'm not a C++/make wizard on purpose cause I have better things to do with my time so that's as complicated as that build system gets.

Usage

To start the program, build and then run `./dash`.

There are a couple of custom commands for your own personal benefit/enjoyment. Here are their details:

`cmdnm <process_id>` - prints the command used to initiate the process_id from the arguments

`pid <command>` - prints the process id or ids associated with a command name, supplied from the arguments

`systat` - prints various interesting system stat stuff

`memman <memory address>` - see function docs above for meaning but this is the usage.

Pipes and file redirects should work the same way that they do in BASH and other shells except for the bug in the pipe that breaks it sadly.

Everything else about the shell should work similar to what you're familiar to, minus all the good parts that make shells like ZSH such a pleasure to use.

Testing & Bugs

This was tested by running a bunch of commands against it during development as well as general 'poking around' on the following systems:

- Raspbian Linux on Pi 2
- Ubuntu 20.10 on my home server
- MacOS Big Sur
- Lubuntu 20.04 in VM

- Ubuntu 20.04 on Windows Subsystem for Linux

Various levels of functionality were supported in each environment. Here's a summary:

- getrusage() was suspect in all of the environments but worked best in MacOS (other systems struggled to give me system time)
- Windows Subsystem for Linux did all the normal shell stuff fine but the custom commands didn't work as you don't have access to the same file structure at root
- Raspbian and native ubuntus did well, supporting most things pretty well. Did notice that the arm processor reports `cpuinfo` in a much more verbose manner than the Intel x86 procs do
- (RESOLVED) There is a bug (?) that happens sometimes where you enter in a command and it pretends like it runs but then locks the whole shell from executing any more commands, even though input is still accepted. I wasn't able to trace down a definite cause so it's just gonna be a known thing that happens. A quick `ctrl-c` and `./dash` will fix the problem. Not ideal, but what can you do. 🤷

Bugs:

- Occasionally backspace remaps itself to `^?` and I'm very unsure why, it doesn't seem to happen reliably. (WAIT NOW IT'S DOING IT IN NATIVE ZSH TOO WTF IS GOING ON)
- Pipes have the issue described above in the Program Structure Section
- There are probably other bugs so this is my blanket catchall for those 😛