By: Archer Heffern
To: Tech Education

Programming Assignment 1: Basic Shell in C

Description

A shell is a special program that is usually integrated in an OS, and which offers humans an interface with the computer. The way it appears to users may vary from system to system (command line, file explorer, etc), but the concept is always the same:

- Allow the user to select a program to be started, and optionally give it arguments.
- Allow trivial operation on the file system like listing the content of directories, moving and copying files across the system.

In order to complete those actions, the shell program may have to issue numerous system calls, like "execute program 'x'; open file 'y' and create it if it doesn't exist; read content from X, write into Y, close both files, write 'done' to standard output".

There are many different versions of shells (sh, bash, zsh, ksh, fish, etc), Linux by default has users use bash, MacOS uses zsh, and Windows uses powershell. On Linux the default shell for each user is determined by the /etc/passwd configuration file. See passwd(5) for more info.

The most important takeaway is that the Shell is just a program used to launch other programs, so we can launch our own shell from the default shell.

Project

We are going to create our own simplified shell. Then expand it to include some common facilities included by most programs.

Our shell will be a read-eval-print-loop (REPL). It should…

1. Read a line of input
2. Evaluate the expression by running the program specified in the input
3. Print the output of the program to the console
4. Loop

See the sample usage at the end of this document, and play around with the demo program sent with this document. Make sure to run it on a Linux environment!

There are 3 parts to the project:

1. Create the basic shell interpreter
2. Make the shell configurable (argument variables, environment variables, configuration files
3. Other frequently seen features (logging, documentation, accessible anywhere on computer.

By: Archer Heffern
To: Tech Education

Prerequisites

If you are on Windows, install the Windows Subsystem for Linux (WSL). Once it's installed, you will have access to a new terminal that gives you access to WSL. WSL is pretty much a second computer running on top of Windows, that acts like Linux, complete with its own file system, binaries, and more.

To run your programs on this second machine, you will need your files to be on the Linux machine, and run via wsl, not powershell, git bash, or any other shell you have.

We will be using C with the GNU C Library - The standard for writing Linux programs in C since it gives us convenience functions for executing system calls.

What is a Shell

To get a feel for how your shell works, open up your terminal (a GUI frontend for Shells - Historically a physical device). Then execute any of the following commands.

- man(1)
- ls(1)
- pwd(1)
- cat(1)

For a great description of what the shell does: https://wiki.osdev.org/Introduction#What_is_a_shell.3F

Programming in C

I will host a C intro session on Sunday. C however, is a pretty small language, has great man page documentation, similar syntax to java, and a mature community, so self learning should not be too hard. Most C functions are covered in section 3 manpages.

I have also linked some helpful resources along with this document.

Here is an excellent video of how you should be using man pages while programming. Don't worry if you don't understand what's actually going on, we will get to that later.

For the split terminal he is using a program called tmux, and the tiling window manager is i3.

Mentality: Don't focus on learning the language, instead you are learning concepts. You already know programming languages (java, python, etc), and at the end of the day, their core (control flow, conditional statements, variables, etc) are conceptually the same. Here are some concepts you should instead be focusing on:

- Memory management (stack, heap, pointers, malloc, free)
- Data modeling with Structs
- Compiling / Linking / Header Files

By: Archer Heffern
To: Tech Education

And here are some OS level concepts to focus on:

- Processes
- File Descriptors

Configuration of Programs

Programs typically are configurable. See the manpages of a program for its specifics.

Config Files

Wiki

Files a program checks for initial settings. Can be formatted however you decide best. Some common formats are INI, TOML, YAML, JSON, and XML.

Argument Variables

An array of string values passed to a process on invocation. We typically interpret them according to POSIX Standards. In C and most programming languages, it is made accessible by the main function.

Eg in java and C:
Java: public static void main(String args[]){} # <- String args are the argument variables
C: int main(int argc, char** argv){} # <- argc is the number of variables, and argv is the array of Strings

Environment Variables

Keeps track of information that is shared by many programs, changes infrequently, and that is less frequently used. Are inherited from the parent process. See environ(5) and execve(2) for more information.

Are represented as Key Value pairs. Eg.
SHELL=/bin/bash
HOME=/home/archer

Precedence

Conventional precedence of configuration methods are as follows (Low to high):

- /etc/<program_name> # File or directory
- $XDG_CONFIG_HOME/<program_name> # $XDG_CONFIG_HOME is an environment variable
- $HOME/<program_name> # $HOME is an environment variable
- environment variables
- argument variables

etc is global, in that every user on the machine can see this config file, as opposed to the others which are user or invocation specific

https://stackoverflow.com/questions/11077223/what-order-of-reading-configuration-values

By: Archer Heffern
To: Tech Education

PATH

We just learned about environment variables, and how they impact the behavior of programs. Let's take a look at one very important variable - The PATH.

All programs can be invoked as ./<program_name> using `execve(2)` as long as you are in the same directory as them. But what about programs such as `man` or `python` which aren't referenced this way?

The PATH environment Variable is where the computer searches for executable files when we try to execute a file as PROGRAM instead of ./PROGRAM. It is a colon separated list of file paths that the computer will linearly search through to find the binary.

Something to note is execve(2) does not search the path, the helper library functions in the exec(3) suite do.

My Mac's PATH variable is:
/opt/homebrew/anaconda3/bin:/opt/homebrew/anaconda3/condabin:/Users/archerheffern/.cargo/bin:/opt/homebrew/bin:/opt/homebrew/sbin:/usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/local/bin:/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/bin:/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/appleinternal/bin:/Library/Apple/usr/bin

So when I run execvp("python", …) in my program or using the shell as "> python", my computer first searches /opt/homebrew/anaconda3/bin, and if it doesn't find it there, will search /opt/homebrew/anaconda3/condabin, and so on.

Part 1: Creating the MVP

First and foremost, you can create this program however you like. I provide a stub as a nice starting point, but if you think there is a better way to do things, by all means, abstract how you like! Ignore the config section for now.

To run the program, compile it into assembly with `gcc -o tamidsh tamidsh.c` and run it with ./tamidsh.

You will need 3 system calls to complete this part of the program, `fork(2)`, `exec(2)`, and `wait(2)`.

The concept of a shell is as follows.

1. The main process waits for user input, which should be the name of a program, and its arguments.
2. Upon user input, the process should fork, and the child will execute the program.
3. The parent process will wait for the child process to finish, before prompting again.

Error Handling

Recall how a process returns a status code - which is a number. Lets assign a meaning to our numbers. (See Therac-25 Malfunction 54). We will then use these status codes when creating our documentation.

By: Archer Heffern
To: Tech Education

For your status codes, create a C preprocessor macro
#DEFINE EXT_ERR_<error_name>

and return that upon failure

```
if (argc != 2) {
        fprintf(stderr, “Usage: sample_program PATH”);
        return EXT_ERR_TOO_FEW_ARGS;
}
```

Part 2: Pipelines and Builtins

This will be released after break!

Part 3: Miscellaneous Features

This will be released the week after break!

Features such as configurability, startup scripts, manual pages, logging, and more.

Submissions

Create a google drive folder called firstname_lastname_techeducation and share it with me (hefferna@brandeis.edu). This will be used as a bucket for all Projects.

For each part, upload a folder to the google drive named firstname_lastname_<project number>_<part number>. The contents are as such:

1. Source code
2. A README file containing instructions for compiling, and a description of any known bugs
    a. Try using markdown syntax!

Sample Usage of Shell

Make sure to run this in a Linux Environment

```
archerheffern $ ./tamidsh_part1
> whoami
archerheffern
> ls
example        helper        sample        tamidsh_part1.c        tamidsh_stub.c
> pwd
Users/archerheffern/Desktop/code/tmp
> echo hello
hello
> quit
Exiting…
```

By: Archer Heffern
To: Tech Education

archerheffern $