CS 253: Introduction to Systems Programming
Smash Part One

February 25, 2020

Learning Objectives

1. String processing
2. Preventing buffer overflow
3. Functions
4. Structures, malloc and free
5. Functional test cases

Overview

This project begins an incremental implementation of a simplified command shell called, `smash`, akin to the Linux `bash` shell. *Incremental development* is a software development practice that constructs a final product as a series of small, evolutionary mini-projects, each adding incremental, fully tested, functionality. P4 is your first incremental release of `smash` while P5 and P6 implement additional functionality. At the end of the semester, you will have a functional shell albeit without a scripting language. P4 includes the following major features:

- Display a prompt on stderr, read and parse user commands from stdin
- Implement the `cd` command
- Implement the `history` command
- Implement the `exit` command
- Print the arguments array for external commands

Internal and External Shell Commands

While a few commands are implemented by the shell itself, most are implemented by separate *external* executable programs. Examples of *internal* commands implemented within smash include, the `cd` command, the `history` command, and the `exit` command. Examples of *external* commands implemented by other programs include, the `ls` command, the `cp` command, and the `gcc` command. P4 only fully implements *internal* commands; *external* commands are parsed but not executed.

Full support for *external* commands will be completed in a future project. Other shell features not implemented in P4 include I/O redirections and pipes.

Getting Started

1. Create a new directory, p4, for your project. There are no starter files. Be sure to organize your code so all the source files, Makefile, and the build artifacts (e.g. object files, etc) reside in this one directory, not in subdirectories where the grader's tests won't find them. Likewise, don't change the names of required files as doing so may break the tests.

2. Create a source file named, smash.c, containing a main function. At this point in time, all it needs to do is loop, printing a prompt, "$ ", to `stderr`, reading a line of text from `stdin`, and echoing that line on `stdout`. The prompt must be written to `stderr`, not `stdout` as you might expect. Your code will look something like this:

```
  ...
#define MAXLINE 4096
  ...
char bfr[MAXLINE];
fputs("$ ",stderr);  //Output the first prompt

//Loop reading commands until EOF or error
while (fgets(bfr, MAXLINE, stdin) != NULL) {
   bfr[strlen(bfr) - 1] = '\0';    //Replace newline with NUL
   puts(bfr);
   fputs("$ ",stderr);
 }
 return 0;
```

3. Implement and test your `Makefile` with the targets noted below. Be sure to compile with `-std=c99` and `-Wall`.

- all - Must be the default (first) rule. Builds the smash shell.
- smash - Builds the smash shell.
- clean - Cleans the working directory by deleting all binary files and/or other build artifacts.

You might save time by also implementing a debug target using the approach in ExampleGdb.

Your Makefile will eventually need to handle header file (*.h) dependencies correctly, rebuilding only the appropriate object and executable files when a header file has been updated. You may wish to review ExampleMakefile3 and Mecklenberg, `https://www.oreilly.com/openbook/make3/book/ch02.pdf`, page 31.

4. Verify `smash` will read and echo lines of text[1], and exit normally after detecting an End-of-File

[1]Echoing is just temporary functionality to enable you to test your Makefile, prompt and code that reads the user's command. You'll discard the `puts` in a later step.

condition (Ctrl-D on a Linux/MacOS keyboard). What happens if your attacker supplies your program with a line of text containing more than MAXLINE characters? Does `make` rebuild your product if a source file is updated? What happens if nothing has been updated when make is run?

5. Investigate how the real bash `history` command functions with the following bash commands:

```
ls | wc >zot
rm zot
history
history
```

Note that bash records the `history` command itself in the history. Also note that the bash `history` command sequentially numbers each printed entry. With regard to the above, smash functions similar to bash.

While similar, the smash `history` command has some differences:

- Bash saves your history in a file restored when you next login; smash does not save your history in a file. Each time smash starts, it begins with an empty history record, and assigns the sequence number 1 to the first command.
- Smash records the exit status of each command, and the smash `history` command displays each command's exit status in brackets like this:

  ```
  1 [0] cd foo
  2 [127] no-such-command
  3 [1] cd no-such-directory
  4 [0] history
  ```

  In P4, smash records an exit status of 127 for external commands as they are not yet supported and, thus, cannot be found. You can learn more about exit status here, `https://www.tldp.org/LDP/abs/html/exitcodes.html`. In bash (but not smash), you can display the exit status of the previous command with, `echo $?`.
- Internal commands do not cause a program to exit and thus have no true *exit status*; both bash and smash fabricate a fake status for them: 0 if successful, 1 if not. Thus, if the smash command, `cd noSuchDirectory` fails, then smash records the command with an exit status of 1 in the history.
- Smash need not record a history entry for the `exit` command.

6. Create a new source file, commands.c, containing the implementation of a function, `void executeCommand(char *str)`, defined in a new header file, smash.h. Replace your existing smash.c call to `puts` with a call to `executeCommand`. Implement function `executeCommand` in commands.c to parse its single parameter into space-separated tokens using the `strtok` API[2].

Construct an `args` array so that `args[0]` will reference the name of the user's command, `args[1]`

[2]K&R documents `strtok` in appendix section, B3, page 249. You may also wish to consult the tutorial at, `https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm`.

will reference the first argument, `args[2]` will reference the second argument, and so on.

7. Implement smash's `exit` command: Enhance the `executeCommand` function to recognize[3] and execute the `exit` command. When the user enters an `exit` command, smash should immediately exit normally (i.e. status=0) with the `exit` API. Note: smash, like bash, has two normal exits, End-of-File (e.g. Ctrl-D on Linux/MacOS) and its internal `exit` command.

8. Implement the `cd` command: When the user enters a command in the form,

```
cd dirname
```

smash will change the shell's current working directory[4] to the specified directory, and print its name on stdout. If the user specifies a non-existent directory, smash outputs an error message[5] to stderr:

```
dirname: No such file or directory
```

where *dirname* is the name of the non-existent directory.

Unlike bash, the P4 `cd` command without arguments will not change the current working directory to the user's home directory. P4 should ignore the `cd` command issued without an argument.

9. If the user enters an *external* command (any command not recognized as *internal*), smash should print the tokens in exactly the format illustrated below[6] to stdout. The $ is the smash input prompt. The following example illustrates smash output for an *external* then two *internal* commands:

```
$ ls -l -a
[0] ls
[1] -l
[2] -a
$ cd /home
/home
$ cd asdfasdf
asdfasdf: No such file or directory
$
```

You'll implement functionality to actually execute external commands in a later assignment.

10. Create a new header file, `history.h`, containing the definitions and function prototypes for the history feature:

```
//Define the layout of a single entry in the history array
struct Cmd {
   char* cmd;       //A saved copy of the user's command string
   int exitStatus;  //The exit status from this command
};
```

[3]Use the `strcmp` API.
[4]See, `man 2 chdir`.
[5]See, `man perror`
[6]The smash tests analyze this output so don't change it

```
//Define the maximum number of entries in the history array
#define MAXHISTORY 10    //Smash will save the history of the last 10 commands

//Function prototypes for the command history feature
void init_history(void);  //Builds data structures for recording cmd history
void add_history(char *cmd, int exitStatus);  //Adds an entry to the history
void clear_history(void); //Frees all malloc'd memory in the history
void print_history(int firstSequenceNumber); //Prints the history to stdout
```

Note: you may modify but do not omit the function prototypes.

11. Implement the `history` command in, `history.c`, with a capacity of storing exactly 10 commands. Consider how history behaves after the user enters more than 10 commands:

- Smash always retains the history of the last 10 commands. When the user enters the 11th command, smash discards the entry for the command labeled 1, and records the 11th command in the last entry of the history array. Smash prints the history for the most recent 10 commands; don't change this as the grader's tests depend upon exactly 10 entries are printed.
- Smash always displays the correct sequence number for all commands. After the user enters an 11th command, smash will only be able to print the history for the commands labeled 2 through 11 inclusive.
- Note that the firstSequenceNumber parameter to print_history specifies the sequence number to be printed for the first element in the history array.

Smash must always release all malloc'd memory before exiting in response to an `exit` command or end-of-file (EOF).

Code Quality

Your code must be well written and use functions where appropriate. As usual, the Makefile must compile the code without warnings or errors using `-Wall` and `-std=c99`. Function prototypes appearing in this specification must be implemented as specified; failure to do so may break test code. Likewise, the source and header filenames cited herein are also required and exercised by the tests.

Extra Credit (10 Points Possible)

Create a list of numbered test cases in a file called, `testcases.txt`. Example:

1. Verify that a blank line merely prints a prompt to stderr for the next command
2. Verify that EOF (e.g. Ctrl-D) exits normally
3. Verify that `exit` exits normally

4. Verify that `cd` to an existing directory actually changes the smash working directory
5. Verify that `cd` to a non-existent directory prints an error message to stderr
6. Verify that smash will not crash in response to an extremely long command

You should be able to identify many, many test cases, but there is a trade-off between implementing features vs. removing obscure defects. Industry practice sometimes creates the list of test cases and even the tests (a la *Test Driven Development)* before implementing the product as it's easier to avoid the defects than to implement and later repair them.

Submitting

At a minimum, you must submit every file required to build smash with `make`. Never submit binary artifacts (e.g. *.o) of your build. Be sure to include `testcases.txt` if you did the extra credit. The onyx commands to submit P4 is:

```
cd p4
submit instructor cs253 p4
```

...where *instructor* is your instructor's onyx username.

Hints

This assignment is much more difficult than P1..P3, but you have lots of time to implement it. However, if you wait till the last days to begin, you'll likely be unable to complete all of its features.

The recommended organization of the source files in P4 facilitate later, more functionally complete versions of smash. Here's how those files are organized in P4:

commands.c Contains the `executeCommand` function with implementations of most internal commands except history.

history.c Contains the implementation of the history command. It's complicated enough to deserve its very own module.

history.h Contains the data types, constants and function prototypes for the history command.

smash.c Contains the `main` function with its loop for reading and executing commands.

smash.h Defines prototypes for most functions except for those implemented in history.c.

You may add members to `struct Cmd` as required by your implementation.

When testing smash, it may be helpful to examine the exit status of a bash command. In bash, you can retrieve the exit status of the previous command with the bash variable, `$?`. Examples:

```
cd foo
echo $?
cd non-existent-directory
```

```
echo $?
history
echo $?
```