CSCI 444/544 Operating Systems
Project 1: What the Shell, Jim?

This project has never before been assigned. It likely contains errors. Ask questions if anything is unclear.

In this project, you will implement an interactive, text-based terminal shell that executes user-specified programs. The shell program is basically an infinite loop that prompts the user for a command, parses and executes that command and its arguments, and waits for the command to terminate before prompting for the next one. Depending on how many points you target in your submission, you may implement additional features.

We have provided several small examples of using the fork/exec paradigm in the lecture materials. You should begin with `loop_exec.c`, which provided the most basic functionality for the project: it reads a command from the user, assumes that the command does not have arguments, executes that command with the default environment, and waits for it to finish before prompting again. Using this example code as your starting shell, begin by writing accessory programs that your shell can execute. At least four such programs are required from the following list of options, which vary greatly in complexity. Your submission must include at least one starred program from the list.

- cat
- **cp ***
- hexdump
- **ls ***
- mkdir
- mv
- **ps ***
- rm
- rmdir
- whoami

We provide edited (to reduce complexity) man pages for each of these programs that outline the features that you should consider implementing. Refer to the point ladder/rubric for details about how much of each program you should implement. The man pages for the programs follow in alphabetical order.

Note that implementing these programs will require you to explore various system and library calls beyond what our example code has explicitly covered. You will need to research which system calls are appropriate for each program and how to use those system and library calls in your programs. **Reviewing the source code for the GNU implementations of these programs before you submit will be treated as an Honor Code violation.** We absolutely will check for any similarities between your submission and GNU's implementations. As you have never seen GNU's implementations, the similarities should be quite minimal.

CAT(1) User Commands

NAME

cat - concatenate files and print on the standard output

SYNOPSIS

cat [OPTION]... [FILE]...

DESCRIPTION

Concatenate FILE(s), or standard input, to standard output.

-b, --number-nonblank

number nonempty output lines, overrides -n

-E, --show-ends

display $ at end of each line

-n, --number

number all output lines

-s, --squeeze-blank

suppress repeated empty output lines

With no FILE, or when FILE is -, read standard input.

EXAMPLES

cat f - g

Output f's contents, then standard input, then g's contents.

cat Copy standard input to standard output.

CP(1) User Commands

NAME

cp - copy files and directories

SYNOPSIS

cp [OPTION]... SOURCE... DIRECTORY

DESCRIPTION

Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

-f, --force

if an existing destination file cannot be opened, remove it and try again (this option is ignored when the -n option is also used)

-i, --interactive

prompt before overwrite (overrides a previous -n option)

-n, --no-clobber

do not overwrite an existing file (overrides a previous -i option)

-R, -r, --recursive

copy directories recursively

-s, --symbolic-link

make symbolic links instead of copying

-u, --update

copy only when the SOURCE file is newer than the destination file or when the destination file is missing

HEXDUMP(1) User Commands

NAME

hexdump - display file contents in hexadecimal, decimal, octal, or ascii

SYNOPSIS

hexdump [options] file...

DESCRIPTION

The hexdump utility is a filter which displays the specified files, or standard input if no files are specified, in the following format:

Display the input offset in hexadecimal, followed by sixteen space-separated, two-column, hexadecimal bytes, followed by the same sixteen bytes in %_p format enclosed in '|' characters.

LS(1) User Commands

NAME

ls - list directory contents

SYNOPSIS

ls [OPTION]... [FILE]...

DESCRIPTION

List information about the FILEs (the current directory by default). Sort entries alphabetically if neither -S or -t is specified.

-a, --all

do not ignore entries starting with .

-d, --directory

list directories themselves, not their contents

--group-directories-first

group directories before files;

-l use a long listing format

-r, --reverse

reverse order while sorting

-R, --recursive

list subdirectories recursively

-S sort by file size

-t sort by modification time, newest first

Exit status:

0 if OK,

1 if minor problems (e.g., cannot access subdirectory),

2 if serious trouble (e.g., cannot access command-line argument).

MKDIR(1) User Commands

NAME

mkdir - make directories

SYNOPSIS

mkdir [OPTION]... DIRECTORY...

DESCRIPTION

Create the DIRECTORY(ies), if they do not already exist.

-p, --parents

no error if existing, make parent directories as needed

-v, --verbose

print a message for each created directory

MV(1) User Commands

NAME

mv - move (rename) files

SYNOPSIS

mv [OPTION]... SOURCE... DIRECTORY

DESCRIPTION

Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

-f, --force

do not prompt before overwriting

-i, --interactive

prompt before overwrite

-n, --no-clobber

do not overwrite an existing file

If you specify more than one of -i, -f, -n, only the final one takes effect.

-u, --update

move only when the SOURCE file is newer than the destination file or when the destination file is missing

PS(1) User Commands

NAME

ps - report a snapshot of the current processes.

SYNOPSIS

ps [options]

DESCRIPTION

ps displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information, use top(1) instead.

By default, ps selects all processes with the same effective user ID (euid=EUID) as the current user.

-A Select all processes.

NOTES

This ps works by reading the virtual files in /proc. This ps does not need to be setuid kmem or have any privileges to run. Do not give this ps any special permissions.

RM(1) User Commands

NAME

rm - remove files or directories

SYNOPSIS

rm [OPTION]... FILE...

DESCRIPTION

This manual page documents the GNU version of rm. rm removes each specified file. By default, it does not remove directories.

If the -I or --interactive=once option is given, and there are more than three files or the -r, -R, or --recursive are given, then rm prompts the user for whether to proceed with the entire operation. If the response is not affirmative, the entire command is aborted.

Otherwise, if a file is unwritable, standard input is a terminal, and the -f or --force option is not given, or the -i or --interactive=always option is given, rm prompts the user for whether to remove the file. If the response is not affirmative, the file is skipped.

OPTIONS

Remove (unlink) the FILE(s).

-f, --force

ignore nonexistent files and arguments, never prompt

-i prompt before every removal

-r, -R, --recursive

remove directories and their contents recursively

RMDIR(1) User Commands

NAME

rmdir - remove empty directories

SYNOPSIS

rmdir [OPTION]... DIRECTORY...

DESCRIPTION

Remove the DIRECTORY(ies), if they are empty.

-p, --parents

remove DIRECTORY and its ancestors; e.g., 'rmdir -p a/b/c' is similar to 'rmdir a/b/c a/b a'

WHOAMI(1) User Commands

NAME

whoami - print effective userid

SYNOPSIS

whoami

DESCRIPTION

Print the user name associated with the current effective user ID.

As a minimum (D-level) requirement, each program should provide basic functionality without any arguments (using reasonable default values such as `stdout` or . for the current working directory. Implementing more features, such as command-line arguments and the associated parsing and population of the argument vector for `execv/execve` will generally result in higher scores. In addition to fleshing out the implementation details of the application programs, modifications to the core shell code counts, as well. Consider the following points ladder:

D-range. Basic functionality in four programs is present. Arguments aren't supported. The core shell code is largely the same as what I provided.

C-range. Some arguments are supported. execv/execve is being called with a populated arguments list parsed from the command line, even if not all features are present. More than four programs may be implemented.

B-range. All suggested arguments are supported for the implemented programs, and those arguments trigger features that are correctly implemented. The shell supports setting a `PATH` variable in the environment and applying that environment to launched programs via the environment vector for `execve`. Environment variables should not survive across invocations of your shell. More than four applications programs are present in the submission, and they are all of good quality.

A-range. It is not possible to achieve an A-level score on this project by following only these specifications. The difference between B-level submissions and A-level submissions is based solely on *dazzle points*. You must implement additional features of reasonable utility that are not specified here. Without revealing the difficulty in these features, some ideas might include colorizing ls output by file type, supporting background, non-blocking execution of programs with the `&` modifier on the command line, and others. We seek creativity in your solutions. Show us what you can do, then argue that what you did is worth an A over a B.

Suggested Structure

Though I do not require it for this project, I strongly suggest that you create a directory structure to store your work. A standard approach is to create a work folder, and in that folder create subfolders called `src` and `bin`. Source code files should live in `src`. When they are compiled (hopefully using a `Makefile`), the compiled and linked binaries should be placed directly in bin. A good strategy is to add your bin folder to your shell's `PATH` environment variable and execute your compiled applications from within your shell without providing the full path or executing from the same folder.

Writeup

Your writeup for this project should be significant. It must contain an explanation of how to build and run your submission and its components. It should explain in English prose how your implementations work by describing their user/kernel interactions through system and library calls. It should also explain what features you have

implemented for dazzle points, how those features should be used and tested, and why they justify a full letter grade improvement of your project score. Justify these points well, as they are not guaranteed.

Submission Expectations

1. **`{cp,ls,ps}.c`**: At least one of these source code files must be present. How complete the implementation is depends on your target level in the points ladder.
2. **`{cat,cp,hexdump,ls,mkdir,mv,ps,rm,rmdir,whoami}.c`**: At least four of these source code files must be present. Like the at-least-one subset above, how complete the implementations are depends on your target level in the points ladder.
3. **`<shell-name>.c`**: The source code implementation of your interactive shell, likely based on `loop_exec.c`. You can choose whatever name you like, just provide instructions for how to build everything. Pats on the back to teams that use `Makefile`s to build everything at once instead of `gcc` for every file manually.
4. **`Writeup.pdf`**: A well-structured, thoughtful, team-created prose document that outlines
5. **`<team-name>.zip`**: This is the only file you will submit on blackboard. It is a zip file that contains all of the requirements above in a single archive.