

# CprE 308 Project 1: UNIX Shell

Department of Electrical and Computer Engineering  
Iowa State University

This is a two-week programming project assignment.

## 1 Submission

**Submit the following items to Canvas:**

- (15 pts) A cohesive summary of what you learned in the project. This should be no more than two paragraphs.
- (85 pts) A .zip or .tar file containing all source code files and a Makefile for your program. The grading criteria is as follows:
  - (10pts) program compiles by executing the `make` command
  - (10pts) source code is neatly formatted and contains comments that describe the important parts of your program. Use good programming practices (create methods to abstract large bodies of code and code that is common to multiple methods, only use global variables when you must, and check `syscall` or system dependent `libcall` return values for errors – don’t assume the OS will always succeed in fulfilling your requests.)
  - (65 pts) required functionality works

## 2 Project Description

In this project you will be creating your own version of a UNIX shell. It will not be as sophisticated as *bash*, the shell you have been using in lab, but it will perform similar functions. A UNIX shell is simply an interactive interface between the OS and the user. It repeatedly accepts input from the user and initiates processes on the OS based on that input.

### 2.1 Requirements

1. (5 pts) Your shell should accept a `-p <prompt>` option on the command line when it is initiated. The `<prompt>` option should become the user prompt. If this option is not specified, the default prompt of “308sh> ” should be used. Read section 3.1 on command line options for more information.
2. (5 pts) Your shell must run an infinite loop accepting input from the user and running commands until the user requests to exit.
3. Your shell should support two types of user commands: (1) built-in commands, and (2) program commands.
  - (a) (5 pts) Built-in commands are commands to interact with your shell program, such as `cd`, `cd <dir>`, etc. See section 2.2 for a list of the builtin commands your shell should support.

- (b) (15 pts) Program commands require your shell to spawn a child process to execute the user input (exactly as typed) using the `execvp()` call. This means a user command will be entered either using an absolute path, a path relative to the current working directory, or a path relative to a directory listed in the `PATH` environment variable. (Read manual page of `execvp`, especially the “Special semantics for `execlp()` and `execvp()`” section. Note that `execvp` will search `PATH` for you.)
4. (5 pts) The shell should notify the user if the requested command is not found and cannot be run.
5. When executing a program command (not a built-in command), print out the creation and exit status information for child process (see section 4 for examples):
- (a) (5 pts) When a child process is spawned, print the process ID (PID) before executing the specified command. It should only be printed once and it must be printed before any output from the command. You can include other information (name of program, command number, etc) if you find it useful.
- (b) By default, the shell should block (wait for the child to exit) for each command. Thus the prompt will not be available for additional user input until the command has completed. Your shell should only wake up when the *most recently executed* child process completes. See `waitpid`.
- (c) (5 pts) When a child process is finished, print out its PID and exit status. Revisit Lab 2 for a refresher on how to handle exit status.
6. (10 pts) Your shell should support background commands using the suffix ampersand (&), i.e. the child should run in the background, meaning the shell will not wait for the child to exit before prompting the user for further input. You still need to print the creation and exist status of these background processes.

An example is, if the user entered “`/usr/bin/emacs`” then the shell would wait for the process to complete before returning to the prompt. However, if the user entered “`/usr/bin/emacs &`” then the prompt would return as soon as the process was initiated.

**HINT:** to evaluate and print the exit status of a background child process call `waitpid` with -1 for the pid and `WNOHANG` set in the options. This checks all children and doesn’t block – see the man page. To do this periodically, a check can be done every time the user enters a command.

## 2.2 Builtin Commands

The following commands are special commands (also called built-in commands) that are not to be treated as programs to be launched. No child process should be spawned when these are entered.

- `exit` – the shell should terminate and accept no further input from the user
- `pid` – the shell should print its process ID
- `ppid` – the shell should print the process ID of its parent
- `cd <dir>` – change the working directory. With no arguments, change to the user’s home directory (which is stored in the environment variable `HOME`)
- `pwd` – print the current working directory

## 2.3 Extra Credit (5 pts)

- Add a built-in command `jobs` that outputs the names and PID of your child processes (i.e. “process pid (procnam)”). For this, you need a data structure which can maintain a list of currently executing background tasks – we suggest using a list.
- Make sure you clearly describe your implementation in your lab report. Your summary may be longer than two paragraphs in this case.

## 3 Useful Information

### 3.1 Command line options

Command line options are options that are passed to a program when it is initiated. For example, to get `ls` to display the size of files as well as their names the `-l` option is passed to it. (e.g. `ls -l /home/me`). In this example, `/home/me` is also a command line option which specifies the desired directory for which the contents should be listed. From within a C program, command line options are handled using the parameters `argc` and `argv` which are passed into `main`.

```
int main(int argc, char **argv)
```

The parameter `argc` is a value that contains the number of command line arguments. This value is always at least 1, as the name of the executable is always the first parameter. The parameter `argv` is an array of string values that contain the command line options. The first option, as mentioned above, is the name of the executed program. The program below simply prints each command line option passed in on its own line.

```
int main(int argc, char **argv) {
    int i;
    for(i=0; i<argc; i++)
        printf("Option %d is \"%s\"\n", i, argv[i]);
    return 0;
}
```

If you are unfamiliar with how command line options work, enter the above program and try it with different values. (e.g. `./myprog I guess these are options`)

### 3.2 Useful system and library calls

The following system calls will be useful in this project. Read the corresponding man pages for those you are unfamiliar with.

- `fork` – create a child process
- `execvp` – replace the current process with that of the specified program
- `waitpid` – wait for a child to exit (or get exit status)
- `exit` – force the current process to exit, with the given return value
- `chdir` – change working directory
- `getcwd` – get current working directory
- `getenv/setenv` – retrieve and set environment variables.

- perror – display error messages based on the value of errno
- strcmp, strcpy, strcat, strtok – string manipulation.

## 4 Example and Test Cases

This is an example test case and output for your program. Your output does not have to look exactly like this; it is simply meant to give you some idea of how the shell should work and how output looks. A `$` prompt indicates a bash prompt, used to execute your shell, and `308sh>` indicates your shell prompt. Also, you may want to prefix all your output messages with some sort of identifier, such as “>>>”, to more easily determine which lines are yours and which came from the executed program.

You should test all of the builtin commands in your shell, in addition to several commands that are not builtin. You should test any error handling you use. You do not have to turn in any test code that you use, but don’t expect to find errors without testing!

Note that you do not have to output the process name on the “exit” line unless you are doing the extra credit; just the PID and status is sufficient.

Shell command line:

```
$ ./shell -p "hello> "
hello> exit
$
```

Simple execution:

```
308sh> /bin/ls
[977801] ls
shell.c  shell.o  shell
[977801] ls Exit 0
308sh> pwd
/home/danield/308/
308sh> cd
308sh> pwd
/home/danield/
308sh> cd 308
308sh> pwd
/home/danield/308/
308sh> ps
[977819] ps
      PID TTY          TIME CMD
    3590 pts/2        00:00:01 bash
   977797 pts/2        00:00:00 shell
   977819 pts/2        00:00:00 ps
[977819] ps Exit 0
308sh>
308sh> nonexistent
[977850] nonexistent
Cannot exec nonexistent: No such file or directory
[977850] nonexistent Exit 255
308sh> test 2 = 3
[978229] test
[978229] test Exit 1
```

308sh>

Background processes:

308sh> sleep 1 &

[977999] sleep

308sh>

[977999] sleep Exit 0

308sh>

308sh> sleep 1 &

[977864] sleep

308sh> sleep 2

[977865] sleep

[977865] sleep Exit 0

[977864] sleep Exit 0

308sh>

308sh> sleep 10 &

[977943] sleep

308sh> kill 977943

[977945] kill

[977945] kill Exit 0

[977943] sleep Killed (15)

308sh>