# Assignment 2

Implement a simple shell.

Implement the basic version and then as many features as you can. All features are required for full marks. There are many examples of shell implementations on the internet, you are free to read them of course, but ensure what you submit is entirely your own work. You may be asked to explain parts of your code.

## BASIC VERSION

Implement a shell that behaves the same as Bash. When your is executed it should:

- Print a prompt (# character).
- Read in a line from stdin.
- This line will be a command followed by space separated arguments.
- Execute the command with the given arguments.
- Wait until command completes.
- Print the prompt again and accept more commands.
- Exit when EOF is received (Ctrl+D).

Example:

```
$ ./comp20200shell
# ls
file1 file2 my_shell
# pwd
/home/user/assign2/
# echo hello world
hello world
# sleep 5
# ^D
$
```

*Note: In this document "$" is the normal bash shell, "#" is your new shell and you won't see "^D" printed on screen.*

Notes:

- Read from stdin with any function you like, (but suggest `getline()`).
- You may need to trim trailing '\n'
- Use system() for initial testing. Later replace with Feature 1.
- Use functions, put each well defined task into a separate function.
- Add error checks where appropriate.
  - Exit if your shell does something it should not.
  - Print error and stay alive if the user make a error (command not found etc).
- Mimic the behaviour of Bash in requested features, but you are not expected to write a full featured shell!
- All needed functions are listed here, you can use others but ensure they are portable and run on a standard Linux instillation.
- Use multiple c source files. Your submission must have at least two .c source files and one header file.
- Use GNU autotools to package your assignment.
- Name your project "assign2_xxxx" and version 1.0, where xxxx is your student number.
- Submit the .tar.gz output by `make dist` (Ensure everything is in tar, like header files etc).
- If you are unable to use auto tools, write your own Makefile and create a .tar.gz manually.
- Put your name, student number and email at the head of all code files submitted.

## FEATURE 1. EXEC AND FORK

Replace `system()` which relies on the shell `/bin/sh`. You can use the `execve` system call or any of the `exec()` family of wrappers found in `man 3 exec` (suggest you use `execvp`).

Because these functions replace the current process with the new process you need to use `fork()` to continue to receive more commands. The parent process should wait for the command to finish before prompting for next command.

You will need to split the single string (`char*`) into an array of strings (`char**`) (just like how argv is), this array should be NULL terminated. `strtok` can help with this.

Example:

```
str_in = "one two three "
str_out[0] = "one"
str_out[1] = "two"
str_out[2] = " three "
str_out[3] = NULL
```

Useful functions: fork() , wait() , execvp and strtok()

## FEATURE 2. DATE IN PROMPT

Improve the prompt so that it displays the date and local time as well as a hash # in the following format:

```
[dd/mm hh:mm]#
```

*(put a single space after # to separate the typed command)*

Useful functions: time() , localtime() and strftime() .

## FEATURE 3. CATCH SIGINT SIGNAL

Improve your shell by catching the SIGINT signal so that when the user types Ctrl+C your shell does not exit. Instead it should behave like the bash shell, printing the prompt on a fresh line.

```
# somecommand^C
# ^C
#
```

Your shell should still exit with EOF Ctrl+D. While debugging you may find Ctrl+\useful (sends SIGQUIT).

Useful functions: signal(), fflush(stdout) .

## FEATURE 4. CHANGE DIRECTORY BUILTIN

Change directory cd is not a user command like ls and cat. On completion of this assignment you should see why. If we were to use execve to execute a cd command, the forked child process would be changed and then would exit, but the parent process would remain unchanged. You can test this with:

```
$ (cd /)
$
```

Instead, cd is a shell builtin. For more info run commands:

```
$ help
$ help cd
```

Improve your shell by implementing a `cd` builtin. Before executing a command, test if first word is `cd`, if it is then use `chdir()` system call, otherwise execute the command as normal. Change to the path given in first argument. Just ignore any subsequent arguments. If no arguments are given change to the path given by the environment variable `HOME`. Relative and absolute paths should work, eg: `../` and `/bin` (this shouldn't require you to do anything).

If the path does not exist or you don't have permission print an error just like bash (read the manual for `chdir` and `perror`).

```
# cd /root
cd: /root: Permission denied
# cd nosuch
cd: nosuch: No such file or directory
```

Useful functions: `chdir()`, `strtok()`, `strcmp()`, `getenv("HOME")` and `perror()`

## FEATURE 5. REDIRECT BUILTIN

Improve your shell by adding a redirect for `stdout` to a file. Only attempt after completing Feature 1. Parse the line for `>`, take everything before as command, and the first word after as the filename (ignore `<`, `>>`, `|` etc).

Standard out is written out to file descriptor 1 (stdin is 0, stderr is 2). So this task can be achieved by opening the file, and copying it's file descriptor over to 1 with `dup2` system call.

```
int f = open(filename, O_WRONLY|O_CREAT|O_TRUNC, 0666);
dup2(f, 1);
```

Note: Using system call `open` not library wrapper `fopen` here.