Today we will start writing our own Unix Shell. The features we plan to implement today are:

- built-in commands: cd, echo, exit;
- calling an external program in foreground;

Notice that we will not deal with environment variables yet. So, no access to $PATH or $HOME. To execute an external program, the user needs to inform the full path of the command. Also, the command “cd” without parameters would usually return to the HOME directory. For now, we will use the “DOS” (a.k.a., Windows Command Line) style: executing “cd” without parameters will display the current directory (similar to “pwd” in Unix).

1. Preliminaries

Other than the obvious “main” function, we will need some auxiliary variables:

- command: stores the command typed by the user
- argument_list: an external command must be parsed and turned into an “array of strings” to be executed. We will parse everything entered by the user and simply deal with them differently.
- argument_count: number of arguments in the array “argument_list”.

Let's start the program. Create the following program: 457shell.c

```c
static char command[MAX_COMMAND_LENGTH];
static char *argument_list[MAX_ARGUMENT_COUNT];
static int argument_count = 0;

int main(int argc, char *argv[])
{
    return 0;
}
```

NOTE: When typing your program, include comments as you feel is necessary.

2. Also, we will need some constants/macros. Add the following lines to the beginning of your program.

```c
#define MAX_COMMAND_LENGTH 256
#define MAX_ARGUMENT_COUNT 16
#define PROMPT "cpsc457 shell > "
```

Where, MAX_COMMAND_LENGTH is the maximum size of the command a user can type. MAX_ARGUMENT_COUNT defines the maximum size of the argument list. PROMPT is the message that is going to be displayed to the user.
3. To compile our code, we will need a “Makefile”:

```
all: 457shell

457shell: 457shell.o  
gcc -o 457shell 457shell.o

457shell.o: 457shell.c  
gcc -c -o 457shell.o 457shell.c

clean:  
    rm -rf 457shell.o 457shell
```

4. Our code still does not do anything. Let us start by implementing main function. A normal shell, in a very basic description, consists on showing a prompt, allowing the user to type some command and executing that command.

Our main function then becomes:

```c
int main(int argc, char *argv[])  
{
    while (1) {
        // prompt
        // parse_command
        // execute_command;
    }  
    return 0;
}
```

5. Next, we will implement the prompt. We will check for what letters are typed by the user and fill our “command” variable. Notice that we have to take care not to overfill the variable. When the user press “ENTER”, then we will finish the entry.

```
int prompt()  
{
    int ch=0;  
    unsigned char finish = 0;  
    unsigned int size = 0;  

    // Initialize variable “command”  
    memset(command,0x00, MAX_COMMAND_LENGTH);

    // Initialize variables “argument_list” and “argument_count”  
    while (argument_count > 0)  
        argument_list[--argument_count] = NULL;  

    if (write(1, PROMPT, sizeof(PROMPT)) != sizeof(PROMPT) )  
        return -EIO;

    while (!finish) {  
        if (read(0,&ch,1) < 0)  
            return -EIO;
```
switch (ch) {
    case '\n': // new line
        command[size+1]=0; // End string;
        finish = 1;
        break;
    case '\b': // backspace (erasing last character)
        if (size>0)
            command[size--]=0;
        break;
    default:
        if (size < (MAX_COMMAND_LENGTH-1))
            command[size++]=ch;

    }
    return 0; // no error

NOTES:
– Notice that for each function we call (except memset), we will check the return value to verify if it was executed correctly or if we had an error.
– Why not “memset”? (Hint: Check manual.)

6. Now, we need to change our “main” function to call this. Substitute the comment “// prompt” for a call to this function “prompt()”. Remember that you should check if the function did not return any error. (If an error happen, you could, for now, simply exit “main”.)

7. Now, try to compile this program. (Running “make” should work.)

You will receive some errors and warnings. The reason is that we are using functions (e.g., memset) and constants (e.g., NULL, EIO) there were not defined in our code. Add the following lines at the begginging our your program. These lines will be including some headers from the C library. Notice that some of these headers are not used yet.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
```

Now the program should compile and execute fine. To exit, press CTRL-C.

```
[user@term dir] $ ./cpsc457
cpsc457 shell > ls
cpsc457 shell > exit
cpsc457 shell > ^C
[user@term dir]$ _
```
8. Next step is parsing the user's entry:

```c
int parse_command()
{
    char* argument;
    argument = strtok(command, " ");
    while (argument != NULL) {
        if (argument_count == MAX_ARGUMENT_COUNT) {
            printf("Too many arguments\n");
            break;
        }
        argument_list[argument_count++] = argument;
        argument = strtok(NULL, " ");
    }
    return argument_count;
}
```

We are using the function for tokens `strtok`. A “manual” version of the same function will be also posted for reference.

Now you should modify the “main” function to call this function after the user entry. You might also want to compile again to check for errors. (Execution won't be different yet, though.)

9. Finally, we need to execute the command typed by the user. It can be a built-in command or an external executable.

```c
int execute_command()
{
    if (!strcmp(argument_list[0], "cd")) {
        // execute_cd();
    } else if (!strcmp(argument_list[0], "echo")) {
        // execute_echo();
    } else if (!strcmp(argument_list[0], "exit")) {
        exit(0); // finishes the execution
    } else {
        // execute_external();
    }
}
```

Again, you need to modify your “main” function to call “execute_command”. Notice that you only need to do this when there is a command to be executed. I.e., if “argument_count” is zero, you should give a prompt again, instead.

After you’re done, compile and test your program. The command “exit” should work now.

```
[user@term dir]$ ./457shell
cpsc457  shell > ls
cpsc457 shell > exit
[user@term dir]$ _
```
10. Finally, let us execute an external command.

```c
int execute_external()
{
    int child = fork();
    if (child < 0) {
        return -ECHILD;
    } else if (child == 0) {
        if (execve(argument_list[0], argument_list, environ) == -1) {
            printf("Error executing %s: %s\n", argument_list[0],
                    strerror(errno));
            printf("TODO: Read $PATH?\n");
            exit(-1);
        }
    } else {
        int status;
        wait(&status);
    }
}
```

Now, you will need to change the “execute_command” function to call this function.

The instruction “fork” creates a new process as a copy of its parent (our shell) and both “processes” continue the execution from the instruction that follows “fork”.

When we test the value of “child”, we are essentially testing 3 conditions:

1) An error happened: the value returned by fork, i.e., in our variable “child” is a negative number.

2) We are executing the child process: the return value is “0”. In this case, we will execute the function “execve”. Notice that the third parameter required is the environment variables. You will need to declare that global variable “environ” at the beginning of the code. For instance, right after the last “#define”.

```c
extern char **environ;
```

Also, notice that the function “execve” returns -1 when an error happens. Otherwise it will never return. This child process will be substituted by the one being executed. Also, notice the use of “strerror” to obtain the error message.

3) We are the “parent” process (our shell): in this implementation, we are waiting for the child to finish, i.e., we are executing it in “foreground”. We will eventually modify our code to deal with processes in background.

Now compile and you should have a very basic shell running. To execute an external command, say “ls”, you will need (for now) to give the entire path (e.g., “/bin/ls”).

NOTE: This code will be posted in the wiki. The functions for the other built-in commands will also be included. Next tutorial will be based on the code posted in the wiki.