CMPT 300 D200
Assignment 2
Our new shell - cshell
Marks: 80 marks (plus 30 bonus marks)

Notes:

1. You can do this assignment individually or in a team of two. If you are doing it in a group, then only one submission per group is required.
2. You may submit multiple times until the deadline. Grade penalties will be imposed for late submissions (see the course outline for the details).
3. Always plan before coding.
4. All the codes in this lab must be done using C language only. No other languages should not be used.
5. Use function-level and inline comments throughout your code. We will not be specifically grading documentation. However, remember that you will not be able to comment on your code unless sufficiently documented. Take the time to document your code as you develop it properly.
6. We will be carefully analyzing the code submitted to look for plagiarism signs, so please do not do it! If you are unsure about what is allowed, please talk to an instructor or a TA.

Assignment Goals

- To understand the relationship between OS command interpreters (shells), system calls, and the kernel.
- To design and implement an extremely simple shell and system call (Bonus).

For this assignment, you should understand the concepts of environment variables, system calls, standard input and output, I/O redirection, parent and child processes, current directory, pipes, jobs, foreground and background, signals, and end-of-file.

Background

OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is usually called the **shell**, it is a user-level program that gives people a command-line interface to launch, suspend, or kill other programs. sh, ksh, csh, tcsh, bash, ... are all examples of UNIX shells. (It might be useful to look at the manual pages of these shells, for example, type: "man csh"). Although most of the commands people type on the prompt are the name of other UNIX programs (such as ls or more), shells recognize some special commands (called internal commands) which are not program names. For example, the exit command terminates the shell, and the cd command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

Created by Dr. Hazra Imran

In this assignment , you will develop a simple shell. The shell accepts user commands and then executes each command in a separate process. The shell provides the user a prompt at which the next command is entered. One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background - or concurrently - as well by specifying the ampersand (&) at the end of the command. The separate child process is created using the fork() system call and the user's command is executed by using one of the systems calls in the exec() family.

We can summarize the structure of shell as follows

1. print out a prompt
2. read a line of input from the user
3. parse the line into the program name, and an array of parameters
4. use the fork() system call to spawn a new child process
    - the child process then uses the exec() system call to launch the specified program
    - the parent process (the shell) uses the wait() system call to wait for the child to terminate
5. when the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

Task 1: [80 marks] Build a new shell (cshell)

You will develop a command-line interpreter or shell with *support of the environment variables and history of executed commands.* Let's call it **cshell**. **The cshell** will support basic shell functionalities.

cshell recognizes the following lines:

- It recognizes lines of the form $<VAR>=<value>
- It recognizes lines of the form <command> <arg0> <arg1> … <argN>, where <command> is a name of built-in command or some executable.

cshell supports the following built-in commands:

- *exit,* the shell terminates on this command
- *log,* the shell prints history of executed commands with time and return code
- *print,* the shell prints argument given to this command
- *theme,* the shell changes color of and output

When **cshell** takes a non-built-in command, it is executed in the child process and command's output is printed. **cshell** creates a child process using *fork()* system call, then **cshell** waits child process to terminate via *wait()* system call. Child process executes a non-built-in command using exec() and its analogues.

You have to create a pipe from the parent process to the child, using *pipe()*. Then you must redirect standard output (STDOUT_FILENO) and error output (STDERR_FILENO) using *dup* or *dup2* to the pipe, and in the parent process, read from the pipe. This is needed to control the output of commands.

So **cshell** should be able to parse the command name and its arguments. Clearly, the hint is that command name always goes first, and arguments separated by space.

Two modes

Our cshell will work in two modes: *interactive mode and script mode*.

The **interactive mode** is activated when **cshell** starts without command line arguments.

Interactive mode is essentially the following loop:

- Print out a prompt
- Read line
- Parse line, if the line is non-valid print an error message
- If the line was valid, do what should be done.

If the shell starts like *./cshell* <filename> it works in script mode. The script is a file containing a set of lines e.g.
<command1> <arg0> <arg1> … <argN>
$VAR1=<value1>
…..
<commandK> <arg0> <arg1> … <argN>
…..
In **script mode** shell does the following for each line of the file:

- Read the line of the file
- Parse line, if the line is non-valid print an error message
- If the line was valid, do what should be done.

Environment variables
The shell will support the inner environment variables. Each environment variable could be stored in a struct like

```
typedef struct {
    char *name;
    char *value;
} EnvVar;
```

When **cshell** takes a line of the form $<VAR>=<value> it should allocate new EnvVar struct with name=<VAR> and value=<value>. All environment variables could be stored in some array. EnvVar's name should be unique. If a variable already exists its value should be updated.

If some argument takes the form $<VAR> **cshell** should look up stored environment variables, find the one with name=<VAR> and substitute argument with environment variable's value. If the corresponding variable doesn't exist, error message must be printed.

Parsing of lines of the form $<VAR>=<value> should be simple given that it starts with $ symbol and variable name and value separated by = sign. Parsing of lines of the form <command> <arg0> <arg1> … <argN> gets a little more complicated, shell should check if <arg> starts with $ symbol.

Built-in commands

We mentioned earlier that the shell will support 3 built-in commands *exit, print and log*. The *exit* is simple, it just terminates the shell.

- The *print* command takes arguments <arg0> <arg1> … <argN> and just prints them. Mind that environment variables could be passed.
- The *log* command history of executed commands with time and return code. So shell should store for each executed command struct like

```
typedef struct {
        char *name;
        struct tm time;
    int code;
} Command;
```

Note: struct tm defined in <time.h>

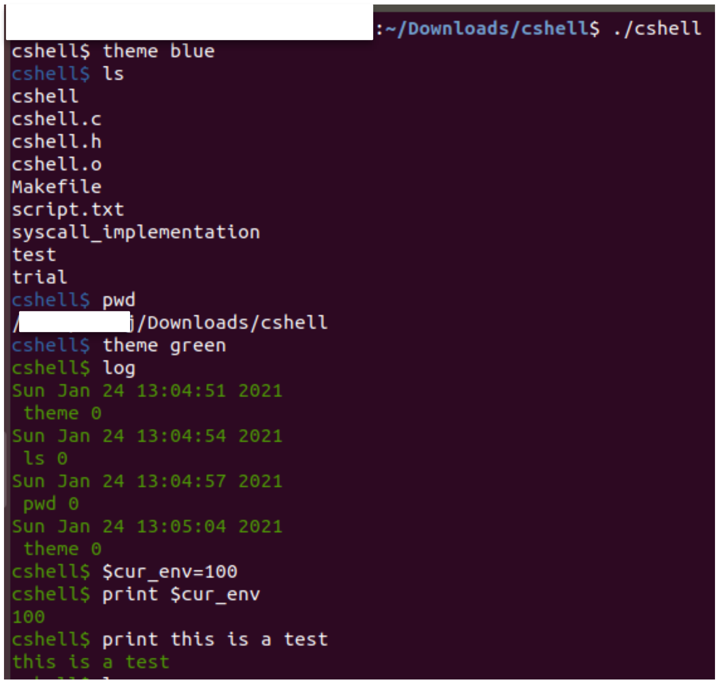
- So the *log* command prints an array of such structs.
- The *theme* command takes one argument which is a name of a color. And then the shell using ANSI escape codes changes the color of its output.

<u>*Hint on the steps to get started*</u>

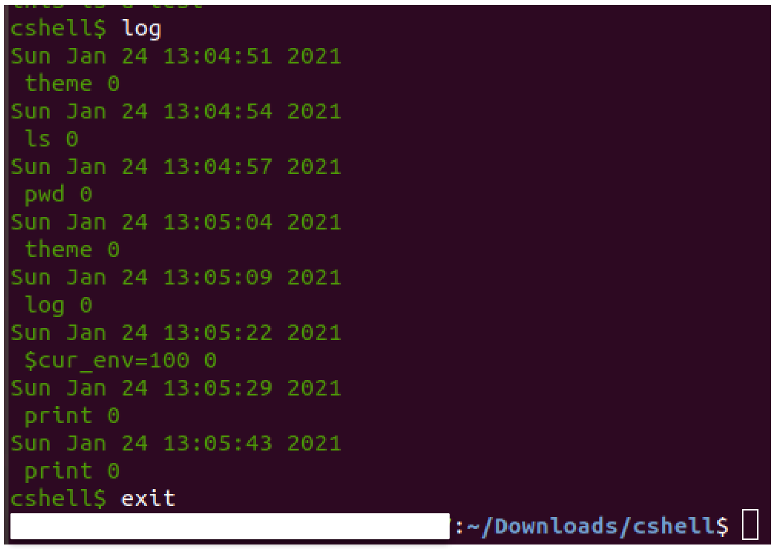
- Read the man pages for fork(), execvp(), wait (), dup2(), open(), read(), fgets(), write(), exit(), malloc(), realloc(), strtok(), strdup() system calls
- Write a loop to read a line and split it to tokens
- Write a function to parse tokens
- Come up with how to store environment variables and log information
- Write a function to handle built-in commands and to execute external files (execvp)
- Handle reading from file
- Be aware of memory leaks. You can manage time with <time.h> library

Sample output:

1. Using the ./cshell in interactive mode:

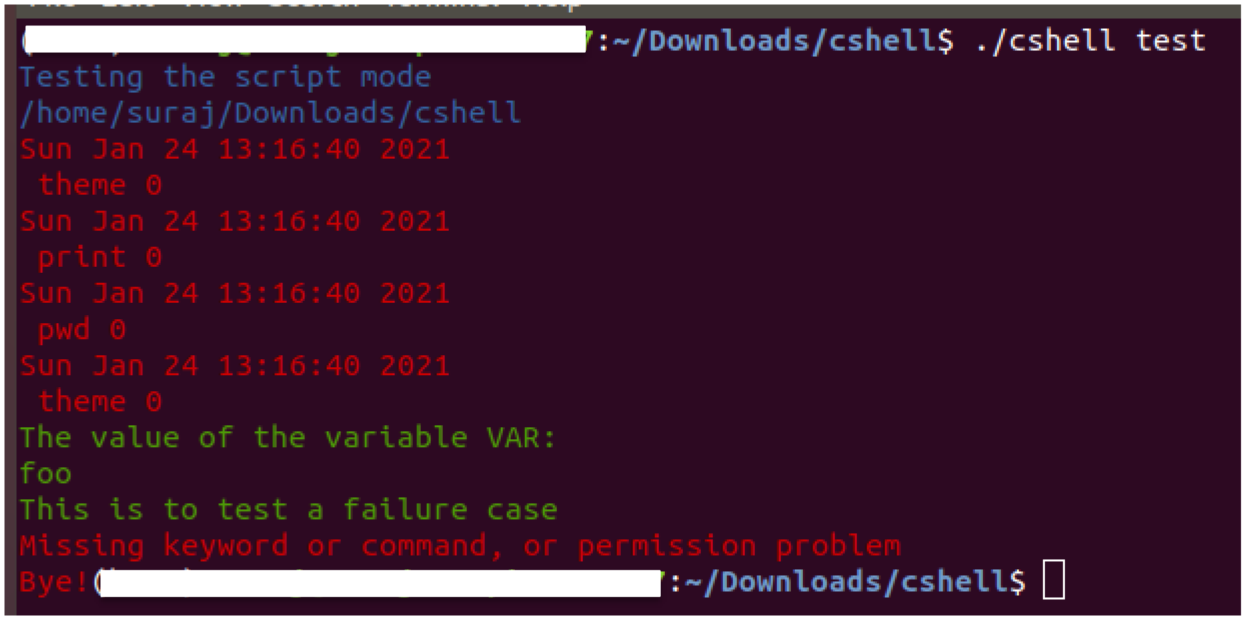


```
[redacted]:~/Downloads/cshell$ ./cshell
cshell$ theme blue
cshell$ ls
cshell
cshell.c
cshell.h
cshell.o
Makefile
script.txt
syscall_implementation
test
trial
cshell$ pwd
/[redacted]/Downloads/cshell
cshell$ theme green
cshell$ log
Sun Jan 24 13:04:51 2021
 theme 0
Sun Jan 24 13:04:54 2021
 ls 0
Sun Jan 24 13:04:57 2021
 pwd 0
Sun Jan 24 13:05:04 2021
 theme 0
cshell$ $cur_env=100
cshell$ print $cur_env
100
cshell$ print this is a test
this is a test
```



```
cshell$ log
Sun Jan 24 13:04:51 2021
 theme 0
Sun Jan 24 13:04:54 2021
 ls 0
Sun Jan 24 13:04:57 2021
 pwd 0
Sun Jan 24 13:05:04 2021
 theme 0
Sun Jan 24 13:05:09 2021
 log 0
Sun Jan 24 13:05:22 2021
 $cur_env=100 0
Sun Jan 24 13:05:29 2021
 print 0
Sun Jan 24 13:05:43 2021
 print 0
cshell$ exit
[redacted]:~/Downloads/cshell$
```

2. Using the ./cshell in script mode:



```
[redacted]:~/Downloads/cshell$ ./cshell test
Testing the script mode
/home/suraj/Downloads/cshell
Sun Jan 24 13:16:40 2021
 theme 0
Sun Jan 24 13:16:40 2021
 print 0
Sun Jan 24 13:16:40 2021
 pwd 0
Sun Jan 24 13:16:40 2021
 theme 0
The value of the variable VAR:
foo
This is to test a failure case
Missing keyword or command, or permission problem
Bye![redacted]:~/Downloads/cshell$
```

Testcases:

Run the ./cshell in interactive mode:

- Run the built-in commands and verify the output. The commands must include ls, theme, log and exit.
- Assign values to any variables in your C program and should print the updated values.

Run the ./cshell in script mode:

- Place all the commands in a txt file and run cshell by passing this file
- The result should be the output of all the commads in the file.
- Place an error case in your file for example have a line as "failure" and the output should print missing command.

Question 2. [2 marks]

Please specify how long you spend on it and what you learned. (This question is worth marks, so please do it!)

BONUS QUESTIONS - These are not mandatory.

Bonus 1: [20 marks] Add a new system call

The task is to add a new system call to the kernel, which will perform the following actions:

- accepts a pointer from the user's application to a string of ASCII characters with codes 32-127 and the length of this string in bytes;
- converts string characters in the range 0x61-0x7A (a-z) to upper case and returns the string.

Steps how to add the system call:

1. Make sure you have the source code of the Linux Kernel
2. Create a folder with your C file and a Makefile in the root directory of the kernel sources
3. Add the system call to the system call table
4. Define the macros assoicated to each system call
    - In arch/x86/include/asm/unistd_32.h:
        - add the definition for our new system call
        - incremented the value of the macro NR_SYSCALLS
    - In arch/x86/include/asm/unistd_64.h add the macro definition
    - In include/linux/syscalls.h add the prototype of the system call
5. The kernel maintains a Makefile as well. Add your directory in the core-y field
6. Compile the kernel

References:

- https://arvindsraj.wordpress.com/2012/10/05/adding-hello-world-system-call-to-linux/
- compiling the kernel: https://www.howopensource.com/2011/08/how-to-compile-and-install-linux-kernel-3-0-in-ubuntu-11-04-10-10-and-10-04/
- https://www.youtube.com/watch?v=6NI1w3DcL7Q

Hint on the steps to get started

- You should read how to recompile kernel and try to do this
- Write system call login in regular function and test it
- Add a new system call

Now, integrate the system call into the shell: Add a new command *uppercase* <*string*> which prints a string modified by your new system call.

Testcase:
Pass any random string to your program and the result should be in an upper case.For example:
Input: "Hello this is a test:
Output: "HELLO THIS IS A TEST"

Bonus 2: **[10 marks]** Implement one more useful system call. Explain why you think it's useful and how it's different from any other system call?

Note: For bonus questions, you have to submit the snapshot of the output

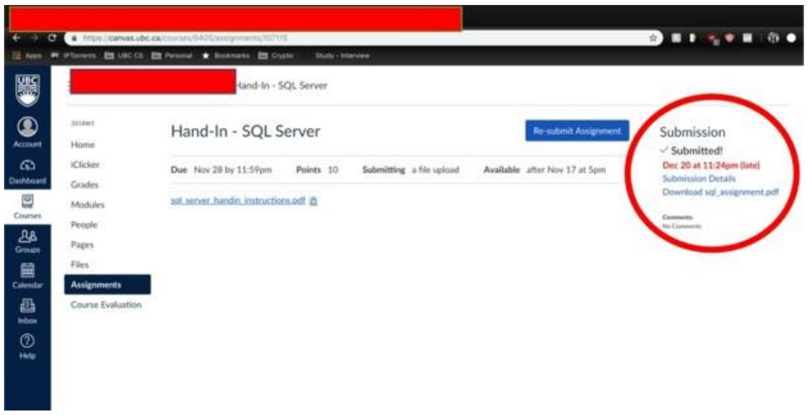
Submission Instructions:

Submit a ZIP file of your assignment folder. Your submission is composed of the following files:

- **Source code (Task 1 + (optional) bonus questions)**
- **Makefile**: the Makefile provides the following functionality:
    - **all**: compiles your program (this is the default behaviour), producing an executable file named same as the C file.
    - **clean**: deletes the executable file and any intermediate files (.o, specifically)
- **README.pdf**: documentation
    - Include your name (and team member name) and assignment number at the top of the file
    - Write down any resources that you consulted (URLs).
    - Write down the name of the class members with whom you have discussed your solution.
    - Mention the claimed late days for the assignment.

    - The answer to Question 2.
    - Bonus question + snapshots+ answer to bonus 2

After submitting the file on canvas, you should be able to see the below message:


[redacted]
https://canvas.ubc.ca/courses/9405/assignments/107115

[redacted] Hand-In - SQL Server

Hand-In - SQL Server

Due Nov 28 by 11:59pm   Points 10   Submitting a file upload   Available after Nov 17 at 5pm

sql_server_handin_instructions.pdf

Submission
✓ Submitted!
Dec 20 at 11:24pm (late)
Submission Details
Download sql_assignment.pdf

Comments
No Comments

Grading Criteria

- Correctness : passes all of the (hidden) unit tests
    - [40 marks] Interactive mode
    - [35 marks] Script mode
- README.pdf [5 marks]: contains the required information