CMPT 300
Assignment 2
Our new shell – cshell
Marks: 80 marks (plus 20 bonus marks)

Notes:

1. **Failure to follow the instructions and rules may lead to failed test cases and/or a final grade of 0.**
2. You can do this assignment individually or in a team of two. If you do it in a group, only one submission per group is required
3. You may submit multiple times until the deadline. Grade penalties will be imposed for late submissions (see the course outline for details).ls).
4. Always plan before coding.
5. For this assignment, you can use:
    - the standard C libraries
    - POSIX APIs - POSIX is a standardized operating systems interface based on UNIX. You can find a list here.
6. All the codes in this lab must be done using C language only. No other languages should be used.
7. 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.
8. Check FAQ before posting questions on Piazza.
9. We will carefully analyze 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.

Coding Rules

- You have to follow the file name as specified in the instructions.
- **Makefile**: Makefile provides the following functionality:
    - **all**: compiles your program (this is the default behaviour), producing an executable file named the same as the C file.
    - **clean**: deletes the executable file and any intermediate files (.o, specifically)
    - You will receive 0 if your makefile fails.
        - Check your build to ensure that there are no errors.
        - Visit TA's programming office hours to get help.

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 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.

In this assignment, you will develop a **simple shell**.

- The shell accepts user commands and then executes each command separately. The shell provides the user with a prompt to enter the next Command.
- 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 the 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 supporting *the environment variables and history of executed commands.* Let us call it **cshell**. **The cshell** will **s**upport 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.

cshell will also support the following *built-in* commands:

- *exit*, the shell terminates on this Command. This must be implemented for a clean exit of the program.
- *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 the color of and output

When **cshell** takes a *non-built-in command* (like *ls, pwd, whoami*), 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 for the child process to terminate via *wait()* system call.
Child process executes a non-built-in command using *exec()* and its analogues.

Hint: You can create a pipe from the parent process to the child, using *pipe()*. Then you must redirect the 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 or you can use *fork()* and *exec()*.

So **cshell** should be able to parse the command name and its arguments. Clearly, the hint is that the command name always goes first, and arguments are 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, and if the line is non-valid, print an error message
- If the line was valid, do what should be done.

Script mode

For script mode, it should start like ./cshell *<filename>*

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,** the 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.
- The program must exit cleanly after executing in script mode.

You must submit *myscript.txt* showing your input for script mode. The last input must be the log command. For example:

```
$VAR=foo
print $VAR
pwd
whoami
theme red
log
```

Note : For the above myscript.txt, the program must exit cleanly after execution.

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 a 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 does not exist, an error message must be printed.
- A command starting with $var should be immediately followed by =value. (Note: **no spaces** before and after =)

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 would support four built-in commands *exit, print, theme* and *log*.

1. The cshell must start with ***cshell$*** (no other characters or welcome messages are allowed)

2. The ***exit*** command is simple. It just terminates the shell with a message "Bye!"
   Note: The output should be exactly the same as shown below (Bye must have exclamation sign, Bye should be case sensitive, and no other messages or symbols allowed)

```
cshell$ exit
Bye!
```

3. The ***print*** command takes arguments <*arg0*> <*arg1*> … <*argN*> and just prints them. Some examples:

```
cshell$ print "hazra"
"hazra"
cshell$ print 'hazra'
'hazra'
cshell$ print 123
123
cshell$ print true
true
cshell$ print A[10]
A[10]
```

```
cshell$ print "Good" "Morning"
"Good" "Morning"
```

```
cshell$ $var = "foo"
Variable value expected
cshell$ $var=foo
cshell$ print $var
foo
cshell$ $var="foo"
cshell$ print $var
"foo"
cshell$
```

If the input is just `print' with no arguments, then display an error message and show the next prompt.

4. The ***log*** command should display a history of executed commands with time and return code. So, shell should store for each executed command a struct like:

```
typedef struct {
        char *name;
        struct tm time;
        return value;
} Command;
```

Note: struct tm is defined in <time.h>

So the *log* command prints an array of such structs.

```
cshell$ log
Thu Jan 27 09:49:33 2022
 print 0
Thu Jan 27 09:49:37 2022
 print 0
Thu Jan 27 09:49:42 2022
 print 0
Thu Jan 27 09:49:45 2022
 print 0
Thu Jan 27 09:49:55 2022
 print 0
Thu Jan 27 09:50:40 2022
 log 0
Thu Jan 27 09:50:46 2022
 ls 0
Thu Jan 27 09:50:48 2022
 pwd 0
```

5. The ***theme*** command takes one argument: a name of a colour. And then, the shell using ANSI escape codes changes the colour of its output. Cshell should support three colours (red, green and blue). For any other color, cshell must display a message "unsupported theme" (the message is case sensitive).

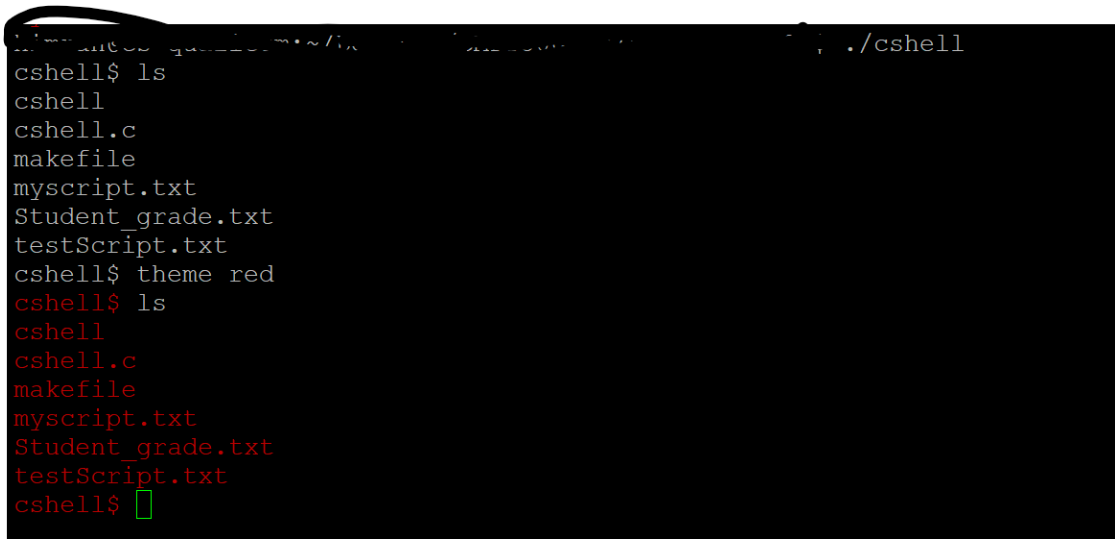
Hint on the steps to get started

- 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 into 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.
- You **cannot** use any third-party libraries.

Sample output:

1. Using the ./cshell in an interactive mode:

```
cshell$ ls
Makefile
cshell
cshell.c
cshell.h
cshell.o
script.txt
syscall_implementation
test.txt
test_script.txt
cshell$ theme red
cshell$ print "ABC"
"ABC"
cshell$ theme blue
cshell$ print 123
123
cshell$ log
Thu Jan 27 11:04:30 2022
 ls 0
Thu Jan 27 11:04:35 2022
 theme 0
Thu Jan 27 11:04:42 2022
 print 0
Thu Jan 27 11:04:51 2022
 theme 0
Thu Jan 27 11:04:55 2022
 print 0
cshell$
```

- 

2. Using the ./cshell script.txt in script mode:

```
1    $VAR=foo
2    print $VAR
3    pwd
4    whoami
5    theme red
6    log
```

```
himran@TABLET-FCBQI4FM:/mnt/c/Users/hazra/CMPT300demo/A2/cshell_A2/cshell$ ./cshell script.txt
foo
/mnt/c/Users/hazra/CMPT300demo/A2/cshell_A2/cshell
himran
Thu Jan 27 11:08:11 2022
 $VAR=foo 0
Thu Jan 27 11:08:11 2022
 print 0
Thu Jan 27 11:08:11 2022
 pwd 0
Thu Jan 27 11:08:11 2022
 whoami 0
Thu Jan 27 11:08:11 2022
 theme 0
Bye!
himran@TABLET-FCBQI4FM:/mnt/c/Users/hazra/CMPT300demo/A2/cshell_A2/cshell$
```

Testcases:

1. Run the ./cshell in an **interactive mode**:
   - Run the built-in commands and verify the output. The commands must include *print, theme, log* and *exit*.
   - Assign values to any variables in your C program and print the updated values.
   - Run non-built-in (*pwd, ls*) commands and verify the output.

2. 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 commands in the file.
   - Place an error case in your file. .

Error outputs

```
cshell$ prin "test"
Missing keyword or command, or permission problem
cshell$ prtint2 "test"
Missing keyword or command, or permission problem
```

```
himran@TABLET-FCBQI4FM:/mnt/c/Users/hazra/CMPT300demo/A2/cshell_A2/cshell$ ./cshell wrongscript.txt
Unable to read script file: wrongscript.txt
```

BONUS QUESTION - This is optional.

Bonus : [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;
- converts string characters in the range 0x61-0x7A (a-z) **to upper case** and returns the string

Steps on 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 associated with each system call
   - In *arch/x86/include/asm/unistd_32.h*:
     - add the definition for our new system call
     - increment 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/

- Add your own system call to the Linux kernel

For Ubuntu 20.04 users: https://dev.to/jasper/adding-a-system-call-to-the-linux-kernel-5-8-1-in-ubuntu-20-04-lts-2ga8

Hint on the steps to get started

- You should read how to recompile the 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 print the value in an uppercase on the terminal.

Submission Instructions:

For A2, the concrete required deliverables are:

- myscript.txt
- cshell.c (with bonus question code if any)
- .h files (if any)
- Makefile
- Bonus question - Also submit a snapshot showing your terminal's output displaying the result of the implemented system call.

Grading Criteria

Rubric (Out of 80)

Incorrect makefile: -80 (0 grade)

Correct makefile: +10 marks
Interactive mode – exit +10
Interactive mode – log + 10
Interactive mode – print +10
Interactive mode – theme +10
Interactive mode – non-built-in command +10
Script mode – +20
Bonus (optional) – uppercase +20

FAQ

1. I am just about to start assignment 2. Can I still work in csil?

   Working in csil is fine for Task 1. But for the bonus question, you will need to setup new syscall on your own virtual environment.

2. In the assignment, are we allowed to change members of a struct?

   Yes

3. Can we assume that the input for commands will be in lower case? For example, typing 'print' will run the print command but typing 'Print' will cause an error.

   Yes, only lower-case commands are valid commands.

4. "Hint on the steps to get started" mentions using execvp for non-built in commands, but is it alright to use execlp?

   Yes

5. I was wondering what the correct output should be when a user types in a command that requires no additional arguments. E.g. "log hi" or "exit 2"

   You can give some meaningful error message like "Error: Too many Arguments detected"

6. What would be the output of "*print hello $VAR*"? (For each whitespace interval there are 5 spaces) - Should the number of spaces be preserved?

   Number of spaces need not be preserved. Some examples:

```
cshell$ print 1234 123    123
1234 123 123
cshell$ print 123 "1231         123"
123 "1231 123"
cshell$ print 123 '11231         123'
123 "1231 123"
```

7. What would be the output of "*print hello $INVALID_VAR*"? (For each whitespace interval there are 5 spaces) - Should the number of spaces be preserved? And should $INVALID_VAR be changed to '' (empty char) or should an error message be printed, like Missing keyword or command, or permission problem.

   White spaces are not preserved. Using a $INVALID_VAR should result in an **error message**.

```
cshell$ $a=4
cshell$ print hello $a
hello 4
cshell$ print hello $b
Error: No Environment Variable $b found.
```

8. For non-built-in commands, In addition to the ls, pwd, whoami given in the example, do we need to complete other commands?

   Your cshell program only identifies the following as built-in-commands:

   1. exit
   2. log
   3. print
   4. theme

This means that you define the behaviour of these commands in your code.

For all other commands (**not limited to pwd, whoami**), your goal is to create a child process, run that command, get the child process' output, and wait for it to terminate.

In general, if the command is not one of the built-in-commands or an assignment operation, the default behaviour is to treat it as a non-built-in-command.

9. Does the '*log*' only store successful commands that run? should it also stores invalid commands with value 1? so if command given was "printt 123", should the 'printt' and '1' be stored into command struct as well?

   log command stores both successful and unsuccessful commands. For unsuccessful built-in-commands, you can use a return value of -1.

10. Does $Var support more than one equal sign? Do we count the first equal sign valid or the latest one valid or just return invalid?

    "$Var==abc" store "=abc" into Var? or store "abc" into "Var="? or return Invalid input? Same question with separated equal sign

    "$Var=ab=c" store "ab=c" into Var?

    Only the exact syntax specified in the assignment instructions is valid.

11. I just want to make sure, the command *uppercase* should be integrated into our cshell or the linux bash? Also, do we need to submit the system call code as well?

    It should be integrated into your cshell.

    but you must implement the system call in your kernel source code first to be able to use your defined systemCall in the cshell.

12. Can we use pipe for this assignment?

    For non-built-in commands, you can use pipe.

13. What should we do if the input is just `theme' with no arguments?

    Print "unsupported theme" as described in the assignment instructions.

14. For the bonus, I used some code from one of the references. Is that ok? If it needs to be cited, where should we include this?

Add a comment in your code and explain what you have used from the source alongside the link to the source