Assignment 2

Due Date and Time: Refer to CSD2180 page. All submissions (programming) must be uploaded to the assignment page link.

Topics: uShell

> If you copy, you are less prepared for the exams, you aren't learning, and you violate the principles of academic integrity.
>
> For this Assignment, you should avoid reading/copying anyone's (or Internet) code that implements a shell or pipe. However, it is OK to see how to use APIs (such as execve, fork etc.).

Background

Shell is a computer program that exposes OS services to users, it uses either a command-line interface (CLI) or graphical user interface (GUI). In this assignment, you are going to implement a simple shell with CLI.

A shell with CLI gives the user a prompt, after which the next command is entered. The example below illustrates the prompt $ and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the Linux cat command.)

$cat prog.c

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c) 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.

Programming Section

- **Objective**: Implement a simple CLI shell program (**uShell.c**) to achieve the following features:
  1. Command prompt:
     a. **uShell>**
     b. newline
  2. Internal Commands
     a. **echo**: print out the rest of the arguments.
     b. **exit**: terminate the shell program with exit value 0.
     c. **setvar**: set environment variables and **${…}**: access environment variables

3. External commend
    a. Executing an external command with arguments
    b. **&**: Executing the external command in background
    c. **finish**: Terminate a process
4. Historical commend
    a. **!!**: execute the most recent command
5. Perform input/output redirection
    a. >
    b. <
6. Allowing the parent and child processes to communicate via a pipe
    a. |

Details:

You can start with something like this:

```
#include <stdio.h>
#include <unistd.h>
#define MAX_LINE 80 /* The maximum length command */
char command[MAX_LINE]; // the command that was entered

int main(void)
{
        /* command line (of 80) has max of 40 arguments */
        char *args[MAX_LINE/2 + 1];
        /* flag to determine when to exit program */
        int should_run = 1;
        while (should_run) {
                printf("uShell>"); /* prompt */
                fflush(stdout); /*note this*/
                /* read a command from the keyboard */
                /* parse the command */

                 /*
                 * After parsing, the steps are:
```

```
            * For internal comments:
            * (1) invoke corresponding functions
            * For external comments:
             * (1) fork a child process using fork()
             * (2) the child process will invoke execve()
             * (3) parent will invoke wait() unless command included &
             */
        }
        return 0;
}
```

1. Command prompt and input:

   When standard input is used as input, there should be a prompt printed before accepting input. For example:

   ```
   uShell>echo hello
   hello
   ```

   The program reads a command line one at a time from the standard input until the end of file or when an exit command is encountered. Every command line is terminated upon a newline character. Every command line is either empty or a sequence of word, where every word is a character sequence of printable characters but not whitespace characters. The length of the command line is usually limited by the parameter ARG_MAX, however we assume there will be a maximum of 80 lines of commands. It is possible for a command line to be empty it is simply a newline character or it is made up of only whitespaces, this will simply reprint the prompt in a newline.

   ```
   uShell>
   uShell>
   ```

   Note that the input may contain variables to be replaced (indicated using **${…}**).

   ```
   uShell>setvar HA hello
   uShell>echo ${HA}
   hello
   ```

2. An internal command is directly executed by the shell itself without executing without executing another program. The **first word** in the command determines which internal is to be executed.
   a. **echo** command should be able to re-print the rest of the command line properly. Multiple whitespaces may be ignored and treated as if it is a single whitespace.
   b. **exit** commend terminates the shell program with 0.

c. **setvar, ${…}**

Any word in a command line that begins with the character **$** and **enclosed with curly braces** refers to a uShell variable (**${…}**). The default value of a variable is an empty string i.e., if a uShell variable is used without any initialization, it is an empty string. The value can be changed via command line. The command to **define** and **change** the value of a variable is **setvar**. The usual syntax of the **setvar** command is as follows:

setvar <varname> <value>

The names of the uShell variable are case-sensitive and contain no spaces. To access a uShell variable, the **$** character and **curly brace {…}** are used. The following examples demonstrate the usage of the **setvar** and **${…}** commands.

```
uShell>setvar HAHA hoohoo  # assign the value hoohoo to HAHA
uShell>echo ${HAHA}        # calling out the value of HAHA
hoohoo
uShell>                    # hoohoo is printed on the screen.
uShell>setvar haha         # variable haha is defined and given a default value.
uShell>echo ${Haha}123     # Attempting to call out the value of an undefined variable.
Error: Haha123 is not a defined variable.
uShell>echo ${HAHA}123     # disambiguate the beginning and end of the var name
hoohoo123
uShell>echo ${HAHA }123    # wrong use of curly braces, Would be read as 2 words.
${HAHA }123
uShell>echo $${HAHA}       # $ sign can be used together with variables
$hoohoo
uShell>echo ${${HAHA}}     # nested use of curly braces are not supported
${hoohoo}
uShell>                    # So the replacement only happens once.
```

Please ensure that your program prints out the same behavior as given above. In particular, should the user attempt to

- Call out an existing variable properly e.g., $fHAHAg123, abc$fHahagdef, the part of the command line must be replaced by the value of the existing variable.

- Call out a non-existing variable properly e.g., $fHoHog123, abc$fHohogdef, but these variables have not been previously set with a setvar command.
- uShell should print out the error message if the variable is not defined.
- Use whitespace within the curly braces e.g., ${HAHA }, ${HA HA}, your uShell must recognize them as two separate words.
- Attempts of nested use of calling out Shell variables are not supported. Hence only 1 replacement should happen for each of these occurrences.

3. External commend

The example below illustrates user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the Linux cat command.)

```
uShell>cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c) and then **create a separate child process that performs the command (fork()+exec() family)**. Unless otherwise specified (with &), the parent process waits for the child to exit before continuing.

This is similar in functionality to the new process creation illustrated in Figure 3.9. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an **ampersand (&)** at the end of the command. Thus, if we rewrite the above command as

```
uShell>cat prog.c &
```

The parent and child processes will run concurrently. The separate child process is created using the **fork()** system call, and the user's command is executed using one of the system calls in the **exec()** family (as described in Section 3.3.1). When a background job is launched, the shell program responds by printing the process index (internal to the shell) and the process id (via the function getpid()). For example:

```
uShell>gedit &        # A window for gedit program should pop up after this
[1] 1496              # gedit is the first background job to launch, hence 1.
                      # gedit's process id from getpid is 1496.
```

Proper error message should be displayed, if the external comment can be found in the via system PATH environment variables.

```
uShell> lsss haha
Error: lsss cannot be found
```

uShell>ls&

Error: ls& cannot be found

The internal command that can be used in connection with this is the internal command finish. finish forces the shell program to wait for particular background processes to complete before continuing. The format for using finish is **finish <process index>**. The given process ID must be a valid one. For example:

uShell>gedit &

[1] 10064

uShell>finish 1

process 10064 exited with exit status 0.

uShell>gedit &

[2] 12456

uShell>finish 1

Process Index 1 Process ID 10064 is no longer a child process.

uShell>finish 2

process 12456 exited with exit status 0.

4. Historical commend

The next task is to modify the shell interface program so that it provides a history feature to allow a user to **execute the most recent command by entering !!**. For example, if a user enters the command ls -l, he/she can then execute that command again by entering !! at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the **history buffer** as the next command. (You just save the last one command as the history command.)

Your program should also manage basic error handling. If there is no recent command in the history, entering !! should result in a message "No commands in history buffer."

uShell>!!

No commands in history buffer.

5. Perform input/output redirection

Your shell should then be modified to support the **'>' and '<'** redirection operators, where **'>'** redirects the output of a command to a file and **'<'** redirects the input to a command from a file. For example, if a user enters

uShell>ls > out.txt

The output from the **ls** command will be redirected to the file **out.txt**.

Similarly, input can be redirected as well. For example, if the user enters：

uShell>sort < in.txt

The file **in.txt** will serve as input to the **sort** command.

Managing the redirection of both input and output will involve using the **dup2()** function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call **dup2(fd, STDOUT_FILENO);** duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as:

uShell>sort < in.txt > out.txt.

Display the same error message “Error: xxx cannot be found” if the command or the file xxx cannot be found.

6. Allowing the parent and child processes to communicate via a pipe

   The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

   uShell>ls -l | less

   has the output of the command **ls -l** serve as the input to the **less** command.

   Both the ls and less commands will run as separate processes and will communicate using the UNIX **pipe()** function described in Section 3.7.4. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute **ls -l**). This child will also create another child process (which will execute **less**) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the **dup2()** function as described in the previous section. Display the same error message “Error: xxx cannot be found” if the command xxx cannot be found.

   Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

(Bonus 1) Evaluation on-the-fly (uShellb.c)

Input from stdin and process line by line:

- If the line of input defines a function, the function is compiled and loaded into the process address space.
- If the input is an expression, output its value.

uShell> int fib(int n) { if (n <= 1) return 1; return fib(n - 1) + fib(n - 2); }
OK
uShell>fib(5) * fib(4) + 1
16

Interpret and execute the C "single-line" code in each line of standard input (assuming we only use the int type functions, that is, all input arguments are integers and the return value is also an integer), in the following two cases:

1. Functions always start with an int, e.g.,:

   int fib(int n) { if (n <= 1) return 1; return fib(n - 1) + fib(n - 2); }

   The function accepts several parameters of type int and returns an int value. If a line is a function, we want it to be compiled by gcc and loaded into the current process' address space. Functions can reference previously defined functions.
2. If a line does not start with an int, we consider the line to be a C-language expression with type int, e.g.

   1 + 2 || (fib(3) * fib(4))

   Both functions and expressions can call previously defined functions only within this bonus section. They do not access global environment values (variables) or call functions in standard C.

Details:

1. Handle and parse the inputs as the non-bonus section.

2. Compile the input function to a shared lib (shared object, .so). Save the code into /tmp or working folder with write right, call gcc with correct parameter to compile it into .so.

3. Compile the input expression to a shared lib, i.e., using a wrapper:

   int wrapper_5() {return **1 + 2 || (fib(3) * fib(4))}**

Note the numbers in the function names - we generate different names for the expressions by adding numbers. Our expression becomes a function, and we can compile it into a shared library. Load the dynamic library into the address space and get the address of wrapper_5, and directly call the function to get the value of the expression.

4. Use dlopen to load the shared lib. Please man and read dlopen, elf, and etc..

5. Do not use system and popen in libc.

(Bonus 2) Report is **NOT** required. However, if you have some interesting implementation, test cases, findings, and extensions that you want to report, you can submit them with a report (limit: 2 pages (A4)).

Notes:

- This assignment is in **Linux**, please start early.
- You can try to write Makefile to manage your development, for example, **make** for compile and **make test** for testing.

Deliverables:

- Zip you folder and name the resulting file using the following convention:
  <class>_<student login name>_<assignment#>.zip
  For example, if your sitid is foo and you are submitting assignment 2, your zipped file would be named as: csd2180_foo_2.zip
- Your folder should contain:
  uShell.c
  The following declaration form, **tick what you have done**.
- Optional:
  uShellb.c
  makeFile
  report

Objectives: CSD 2180 A2

Name: _____

SIT Student ID: _____

1. Student program compiles, links, and executes. If not, penalty is -100 points. ____
2. Command prompt. (5)____
3. Internal Commands
    a. **echo**: print out the rest of the arguments. (5)____
    b. **exit**: terminate the shell program with exit value 0. (5)____
    c. **setvar**: set environment variables and **${…}**: access environment variables. (20)____
4. External commend
    b. Executing an external command with arguments. (10)____
    c. **&**: Executing the external command in background. (10)____
    d. **finish**: Terminate a process. (5)____
5. Historical commend. (10)____
6. Perform input/output redirection. (10)____
7. Allowing the parent and child processes to communicate via a pipe (10)____
8. Proper coding style, naming, and comments. (10)____
9. Bonus 1. (20)____
10. Bonus 2. (5)____

DECLARATION:

I have read the statements regarding cheating in both the CSD2180 course handout and DigiPen student handbook. I affirm with my signature that this is my own solution to A2 and the submitted source code is of my creation and represents my own work.

Signature: