Notes along:

* cmd is normally a name of file that contains an executable program “ls” or “g++” (files stored in “/bin”
  + shell will do:
    - find file
    - prepare list of parameters
    - execute
* Other is implemented in the shell

**1 How you are going to get user input in a loop and tokenize it**

**2 How you are going to “parse” the tokens into commands**

**3 Functions for all your built-in commands.**

How to create an executable

**cis-linux1> g++ -c –I. –o main.o main.cpp**

**cis-linux1> g++ -o main.o main**

**cis-linux1> ./main //** The shell finds the **main** file in the current directory (specified by ./) and then executes it

Logistics:

Shell size: 25 lines & 80 spaces each line

1. Shell initialize variable & perform endless loop till reach EOF condition (Ctrl + C or “exit”)

waits for the user to type a command.

1. Print the prompt

When the shell is started, it can look up the name of the machine on which it is running

and prepend this string to the standard prompt character

**cislinux1>**

the shell also print the current directory as part of the prompt, meaning that

each time that the user types **cd** to change to a different directory, the prompt string is redefined. Once the prompt string is determined, the shell prints it to **stdout** whenever it is ready to accept a command

line.

1. Get the cmd:

Once the user types the command line (and terminates it with a NEWLINE ('\n') character), the command line string is returned to the shell program.

process that executes the shell will be asleep until the user types a command line in response to the prompt

std::string cmd\_string = “ ”;

while (cmd\_string == “ ”) {

std::cout << "prompt>";

std::getline (std::cin, cmd\_string);

std::cout << "The entered command is: " << cmd\_string << "\n";

}

1. I/O processing

#define ARG\_NUM 100

string arg[ARG\_NUM];

int i = 0;

stringstream ssin(cmd\_string);

while (ssin.good() && i < ARG\_NUM){

ssin >> arr[i];

++i;

}

for(i = 0; i < ARG\_NUM; i++){

cout << arr[i] << endl;

}

Parse the command.

The parser begins at the left side of the command line and scans until it sees a whitespace character (such as space, tab, or NEWLINE). **The first word is the command name**, and subsequent words are the parameters.

Take input in as a C-string

Store number of strings in the cmd in argc

Store the C-Strings in an array of character pointers char\* argv[100], set end of C-string to NULL

tokenize user input: **command [arg1] [arg2] ... [argN] (end by Enter/Return key/newline character)**

find next token --> check name

A command to be executed

Arguments to the command

BASH commands < > >> | &

Input file Output file

if “-“ or “flag”

which flag can go with this “command” or can be grouped

is position important

if argument “a” “l”

if pipe

if “>” redirect stdout ...  
 if “&” run process in background & return prompt  
  
Input redirection using <   
Output redirection using > (truncate file)   
Output redirection using >> (append to file)   
Piping between 2 processes using |   
Background execution using &

1. Find the cmd file:

The shell provides a set of *environment variables* for each user

for the bash shell this is /home/<username>/.bashrc

The PATH environment variable (whose value can be viewed by typing

“echo $PATH” at the bash shell) is an ordered list of absolute pathnames specifying where the shell should search for command files

-----------------

Your shell will have to be able to execute two different types of commands:

Built-in Commands you will program inside of the shell code

External commands (i.e. any exernal program on your computer)

**Built in cmds: 🡪 functions**

***cd change the current directory***

#include to get access to chdir

void cd (String path\_to\_change){

char s[100];

// printing current working directory

printf("%s\n", getcwd(s, 100));

chdir(path\_to\_change);

printf("%s\n", getcwd(s, 100));

}  
***clr clears the screen clear dir***

***ls print the contents of the current directory or a specified directory***

***environ list the environment string***

***echo print the user’s input -🡪 or interpret $PATH***

***help display the shell’s user manual help or man***

***pause pause the shell until the user presses enter, then continue execution***

***quit exit the shell***

if not built-in

using the fork() system call to create a child process (originally

a copy of the shell program)

handle any input/output redirection by changing the file

descriptors of the child process using the **dup2** system call.

handle possible piping between programs by using the **pipe()**

system call.

using the **exec()** system call to overwrite the the child

process with the program to be executed.

use the **wait()** system call to wait for the child process to exit

before continuing to the shell’s main loop (unless & was used)

1. Prepare the arguments

use argv[0] to find the executable file

find cmd files in current directory

find cmd files based on the absolute pathname

find cmd files in the $PATH environment variable

* check if file exist?

Try to open the file. If the open fails, then the file does not exist.

* Check if arguments valid (hmmm not really 🡪 throw straight to cmd execution)

interprets what the list of tokens is telling the shell to do. It will also throw an error here if it encounters something which is not valid.

use:  
 #include <unistd.h>

extern char \*environ[]; // The last item in the array is a NULL C-String.

1. Cmd execution

UNIX shells have always been designed **to protect the original process from crashing when it executes a program**. That is, since a **command can be *any* executable file, then the process that is executing the shell must protect itself in case the executable file contains a fatal error**. *Somehow, the shell wants to launch the executable so that even if the executable contains a fatal error (which destroys the process executing it), then the shell will remain unharmed*.(by fork())

Create a child process to execute the command

The final step in executing the command is to fork a child process to execute the specified command and then to cause the child to execute that command. The following code skeleton will accomplish this.

// Child

if (fork() == 0)

{

execvp(<fullpathnameofcommand>, argv);

}

// Parent

else

{

int status=0;

wait(&status);

cout << "Child exited with status of " << status << endl;

}

Use file descriptor for I/O redirection

Watch out for file permission problems with the redirection operators. You need to bone up on the man pages for

file() to get these flags correct. The default permissions for the file open for write meant subsequent redirects to

the same file didn't work of because permission problems.  
int newstdin = open(inputFileName.c\_str(),O\_RDONLY);

int newstdout = open(outputFileName.c\_str(),O\_WRONLY|O\_CREAT,S\_IRWXU|S\_IRWXG|S\_IRWXO);

Your shell will have to support output redirection for both

external commands and for the following built-ins:

dir

environ

echo

help

Your shell only has to support input redirection for external commands.

It should support both input AND output redirection on the same command!

Your shell only needs to support piping and background execution for external commands as well.

To get command line pipes to work, re-read the Shell Pipes section carefully. Since a pipe is just like a file... there’s

no reason why you couldn’t use I/O Redirection and pipes together!

1. Have the parent create a pipe.

2. Spawn a child.

3. Have the parent redirect stdout to the write portion of the pipe.

4. Have the parent execute the first command with excecvp().

5. Have the child redirect stdin to the read portion of the pipe.

6. Have the child execute the second command with execvp().

7. The output from the parent command should flow to the child command via the pipe and i/o

redirection you performed on the pipe.

Parent (create “pipe” file)

Pipe(parent)