$Id: asg2-shell-fnptrs-oop.mm,v 1.49 2019-10-08 15:40:49-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cse111-wm/Assignments/asg2-shell-fnptrs-oop
URL: http://www2.ucsc.edu/courses/cse111-wm/:/Assignments/asg2-shell-fnptrs-oop/

1. Overview

You will maintain a tree structure with a simple hierarchy, maintained by a map of functions. The program will involve object-oriented programming using virtual functions, and displatching functions from a table of functions.

Programming will be done C++ style, not C style, as shown in the following table.

Do not use :	Instead, use :
raw **char*** strings	**<string>**
raw C arrays	**<vector>**
<stdio.h>, **<cstdio>**	**<iostream>**, **<iomanip>**
raw pointers	**<shared_ptr>** or **<unique_ptr>**
union	inheritance
<*header***.h>**	**<c***header***>**

Header files : Include only C++11/14/17 header files and facilities where feasable, and use namespace **std**. Include **<c***header***>** files only when C++-style files are unavailable. Include **<***header***.h>** files from C only when an appropriate **<c***header***>** files is unavailable. Use the script **cpplint.py.perl** (a wrapper for **cpplint.py**) to check style, and **checksource** to check some formatting considerations.

2. Program specification

The program specification is given in terms of a Unix **man**(1) page.

NAME

yshell — in memory simulated tree shell

SYNOPSIS

yshell [**-@** *flags*]

DESCRIPTION

This shell reads commands from the standard input and write output to the standard output, with errors being written to the standard error. Each line read by the shell is parsed into words by splitting using space characters, with any number of spaces between words. There may also be leading and trailing spaces. The first word on any line is a command to be simulated, and the rest are operands to that command. If either the standard input or the standard output is not a tty, each line from the standard input is echoed to the standard output.

The commands modify an inode tree, where each inode is either a file or a directory. Files contain data and directories contain inodes. An inode is specified by means of a pathname. A pathname consists of a sequence of characters separated by slash (**/**) characters.

The inode tree has a root, which is a special node, and also a current inode as well. Whenever a pathname is decoded, if the first character is a slash (**/**), decoding begins at the root, otherwise it begins with the current directory. Whenever a pathname component is a dot (**.**), it refers to the current directory.

If a component is a double dot (`..`) it refers to the parent of the current directory. Every directory has both of these entries, with the root being its own parent. Multiple adjacent slashes are treated as a single slash. Trailing slashes are permitted only on directories.

Every inode has three attributes: an inode number, which is uniquely assigned, starting from 1 for the root; contents, which is a map from filenames to inodes for a directory, and text for a file; and a size, which is the byte count for text, and the number of sub-inodes for a directory.

OPERANDS

None. All input comes from the standard input.

OPTIONS

The `-@` option is followed by a sequence of flags to enable debug output, written to the standard error.

COMMANDS

The following commands are interpreted. Error messages are printed and nothing is done in the case of invalid operands.

`#` *string*

If the first non-space character on a line is a hash, the line is a comment and is ignored.

`cat` *pathname*...

The contents of each file is copied to the standard output. An error is reported if no files are specified, a file does not exist, or is a directory.

`cd` [*pathname*]

The current directory is set the the pathname given. If no pathname is specified, the root directory (`/`) is used. It is an error if the pathname does not exist or is a plain file, or if more than one operand is given.

`echo` [*words*...]

The string, which may be empty, is echoed to the standard output on a line by itself.

`exit` [*status*]

Exit the program with the given status. If the status is missing, exit with status 0. If a non-numeric argument is given, exit with status 127.

`ls` [*pathname*...]

A description of the files or directories are printed to the standard output. It is an error if any of the file or directory does not exist. If no pathname is specified, the current working directory is used. If a pathname specified is a directory, then the contents of the directory are listed. A directory listed within a directory is shown by a terminating slash. Elements of a directory are listed lexicographically.

For each file listed, output consists of the inode number, then the size, then the filename. Output is lined up into columns and each column is separated from the next by two spaces. The numeric fields are exactly 6 characters wide and the units position in a column must be aligned.

`lsr` [*pathname*...]
: As for **`ls`**, but a recursive depth-first preorder traversal is done for subdirectories.

`make` *pathname* [*words*...]
: The file specified is created and the rest of the words are put in that file. If the file already exists, a new one is not created, but its contents are replaced. It is an error to specify a directory. If there are no words, the file is empty.

`mkdir` *pathname*
: A new directory is created. It is an error if a file or directory of the same name already exists, or if the complete pathname to the parent of this new directory does not already exist. Two entries are added to the directory, namely dot (`.`) and dotdot (`..`). Directory entries are always kept in sorted lexicographic order.

`prompt` *string*
: Set the prompt to the words specified on the command line. Each word is separated from the next by one space and the prompt itself is terminated by an extra space. The default prompt is a single percent sign and a space (`%␣`).

`pwd`
: Prints the current working directory.

`rm` *pathname*
: The specified file or directory is deleted (removed from its parent's list of files and subdirectories). It is an error for the pathname not to exist. If the pathname is a directory, it must be empty.

`rmr` *pathname*
: A recursive removal is done, using a depth-first postorder traversal.

EXIT STATUS

0 No errors were detected.

1 Error messages were printed to the standard error.

3. A sample run

The following table shows a sample run. Each interaction with the shell is listed in a separate box with shell output in `Courier Roman` and user input in **`Courier Bold`** typeface. A commentary about what is happening is opposite in the right column.

`%` **`pwd`** `/`	Initially the cwd is the root directory.

`% ls`
`/:`
`1 2 .`
`1 2 ..`
	The absence of an operand to **`ls`** means that dot is used as its operand, which is currently the root. Directories always contain at least two items, namely dot and dotdot. The inode number of the root is always inode #1. The parent of dotdot is itself.

`% make foo this is a test`	Make a file called **`foo`** which contains the string "**`this is a test`**", which is 14 characters. An inode is allocated, namely inode #2.
`% make bar test a is this`	Another file, similarly created, with inode #3.

`% ls`
`/:`
`1 4 .`
`1 4 ..`
`3 14 bar`
`2 14 foo`
	Same as the previous output of **`ls`**, except with two more files. Note that files are kept in lexicographic order, so **`bar`** is listed before **`foo`**.

`% cat food` `cat: food: No such file or directory`	An error message is printed, causing the return code from the shell eventually to be 1 rather than 0. Note the error format: command followed by object causing the problem followed by the reason for the failure.
`% cat foo` `this is a test`	Files can consist of only one line, namely a string.
`% echo 0 for a muse of fire` `0 for a muse of fire`	Arguments to **`echo`** are simply written to the standard output.
`% prompt =>`	The prompt is changed to the characters "**`=>`**" followed by a space. Multiple words would have been permitted.
`=> rm bar`	The file **`bar`** is deleted and the size of the root directory is reduced by 1.
`=> make baz foo bar baz`	A new file is created with inode #4.
`=> mkdir test`	Inode #5 is created as a directory called **`test`**. This directory is a child of the root and contains the two usual entries, dot and dotdot.
`=> prompt %`	The prompt is changed back to a `%` followed by a space.

% **ls /** /: 1 5 . 1 5 .. 4 11 baz 2 14 foo 5 2 test/	Just checking the contents of the root.
% **cd test**	The cwd is now **test**.
% **pwd** /test	Yes, it is.
% **cd**	Without arguments **cd** goes back to the root directory.
% **pwd** /	OK.
% **cd test**	Go to a directory called **test** which is a subdirectory of the cwd, whose alias name is always dot.
% **pwd** /test	
% **cd ..**	Dotdot is always an alias for the parent of the cwd.
% **pwd** /	
% **cd test** % **make me me me me**	This would have errored out if **test** were not a directory or did not exist. The next available inode is #6.
% **cat me** me me me	
% **cd ..** % **cd test**	
% **cat me** me me me	
% **cd**	
% **lsr /** /: 1 5 . 1 5 .. 4 11 baz 2 14 foo 5 3 test/ /test: 5 3 . 1 5 .. 6 8 me	Recursive directory listing. This is done using a preorder traversal. Within a given level, lexicographic ordering is used. Recursion will go through all subdirectories at all levels.
% **cd test**	

`% mkdir foo`
`% cd foo`
`% mkdir bar`
`% cd bar`
`% mkdir baz`
`% cd baz`
	Note that **foo** uses inode #7, **bar** uses inode #8, and **baz** uses inode #9.

`% ls .`
`.:`
`9 2 .`
`8 3 ..`
	At this point dot is **baz** and dotdot is **bar**.

`% cd /`
`% lsr test`
`/test:`
`5 4 .`
`1 5 ..`
`7 3 foo/`
`6 8 me`
`/test/foo:`
`7 3 .`
`5 4 ..`
`8 3 bar/`
`/test/foo/bar:`
`8 3 .`
`7 3 ..`
`9 2 baz/`
`/test/foo/bar/baz:`
`9 2 .`
`8 3 ..`
	A rather large test showing inode numbers, file and directory sizes, and filenames. Note that directory names are indicated in the listing with a trailing slash. Again, the size of a file is the number of characters in it and the size of a directory is the sum of the number of files The subdirectory count is not recursive.

`% ^D`	End of file or Ctrl/D exits the shell.

4. A tour of the code

Begin by studying the code provided in the **code/** subdirectory. There are four modules arranged into a header (**.h**) file and an implementation (**.cpp**) file, the main program in **main.cpp**, and, of course, a **Makefile**. Notice that comments are in the header for when specifying general functionality, and only in the implementation as a way of explaining how something works.

Makefile Study the various targets **all**, **${EXECBIN}**, **%.o**, **ci**, **lis**, **clean**, **spotless**, **submit**, **verify**, and **deps**, which perform their usual functions.

debug.{h,cpp} The **debug** module is already written for you. It is useful in tracing through your code. Other parts of the code may want to have more **DEBUGF** and **DEBUGS** calls added. Note that you should also use **gdb** to track down bugs. Use **valgrind** to check for invalid memory references and memory leak.

util.{h,cpp}	The **util** module is just a collection of independent functions which are herded together. It is a module without any cohesion, but a useful place to park various random functions.
main.cpp	The main program is mostly complete. It scans options, then loops reading and executing commands.
commands.{h,cpp}	Note that the functions are provided but do not do more than print a trace. Each function as a **inode_state** argument, passed by reference, which it might update, and a **wordvec** arlgument. **words[0]** is the name of the command itself, so the first argument is **words[1]**. This will take the most work, but commands can be added one at a time, by addition to the organization that is already there. You may add private functions is you need to. This is the major execution engine.
file_sys.{h,cpp}	The **inode**, or file system, module is the main data structure that you are working on. As you implement commands, also implement the functions of this module as well.

5. What to submit

Submit the files **Makefile**, **README**, and all C++ header and source files. All header files must end with **.h**, and source files must end with **.cpp** as the suffix.

Run **gmake** to verify that the build is possible. And when you run **submit**, do so from the **Makefile** target of that name. That way you won't forget to submit a file. If you forget to submit a file, or submit the wrong version, you will lose at least 50% of your program's value. Make sure that you do not get any warnings from **g++** and that **checksource** and **cpplint.py.perl** dot not complain about anything. Be sure that **valgrind** shows no uninitialized variables and no memory leak. Do not submit any file that is built by the **Makefile**.

If you are doing pair programming, follow the additional instructions in :
/afs/cats.ucsc.edu/courses/cse111-wm/Syllabus/pair-programming
Note that points will be deducted for an improperly formatted **PARTNER** file.

The code must compile and run using **g++** on **unix.ucsc.edu**, regardless of whether it runs elsewhere. When this document was formatted (October 8, 2019) that was :

```
bash-1$ which g++
/opt/rh/devtoolset-8/root/usr/bin/g++
bash-2$ g++ --version | head -1
g++ (GCC) 8.3.1 20190311 (Red Hat 8.3.1-3)
bash-3$ uname -srp
Linux 3.10.0-1062.1.1.el7.x86_64 x86_64
bash-4$ hostname
unix1.lt.ucsc.edu
```

6. Instructions to graders

Look in the **.score/** subdirectory for instructions to graders.