Assignments and Projects

CS3307 Individual Assignment - Returned

Title	CS3307 Individual Assignment
Student	Jakob Franz Wanger
Submitted Date	Oct 3, 2019 10:36 am
Grade	**98.00 (max 100.00)**
History	Tue Oct 01 13:09:03 EDT 2019 Jakob Franz Wanger (jwanger) submitted Tue Oct 01 13:17:29 EDT 2019 Jakob Franz Wanger (jwanger) submitted Wed Oct 02 23:03:20 EDT 2019 Jakob Franz Wanger (jwanger) submitted Thu Oct 03 10:36:22 EDT 2019 Jakob Franz Wanger (jwanger) submitted

Instructions

CS3307 Individual Assignment

Fall Session 2019

Purpose of the Assignment

The general purpose of this assignment is to develop some simple C++ utilities for the Raspberry Pi Desktop, given a number of requirements, making use of the principles and techniques discussed throughout the course. This assignment is designed to give you experience in:

- object-oriented programming using C++, using basic language constructs, classes, and data types
- looking through Linux manual pages and documentation, as you will likely need to do this in your projects later
- getting acquainted with the Linux-based Raspberry Pi Desktop system and services, which will help in your use of an actual Raspberry Pi later in the course

The assignment is intended to give you some freedom in design and programming the explore the subject matter, while still providing a solid foundation and preparation for the type of work you will later be doing in the group project.

Assigned

Friday, September 13, 2019 (please check the main course website regularly for any updates or revisions)

Due

The assignment is due Thursday, October 3, 2019 by 11:55pm (midnight-ish) through an electronic submission through the OWL site. If you require assistance, help is available online through OWL.

Late Penalty

Late assignments will be accepted for up to two days after the due date, with weekends counting as a single day; the late penalty is 20% of the available marks per day. Lateness is based on the time the assignment is submitted.

Individual Effort

Your assignment is expected to be an individual effort. Feel free to discuss ideas with others in the class; however, your assignment submission must be your own work. If it is determined that you are guilty of cheating on the assignment, you could receive a grade of zero with a notice of this offence submitted to the Dean of your home faculty for inclusion in your academic record.

What to Hand in

Your assignment submission, as noted above, will be electronically through OWL. You are to submit all source code files, header files, and build files necessary to compile and execute your code. If any special instructions are required to build or run your submission, be sure to include a README file documenting details. (Keep in mind that if the TA cannot run your assignment, it becomes much harder to assign it a grade.)

Assignment Task

Your assignment task is to familiarize yourself with the Raspberry Pi Desktop and develop some simple C++ utilities that help manage files on the system. In a way, you are building some stripped down replacements to utilities like mv, cp, ls, cat, rm, diff, and stat. Because of the way Linux works, this is surprisingly easy and doesn't take a whole lot of work. (Please note that manual pages linked for your convenience above and below are for Debian Stretch, the same basic Linux distribution that Raspberry Pi Desktop is based upon. That said, things may look or function differently under Raspberry Pi Desktop itself, and so you should consult the man pages on the system itself as your primary source of documentation.)

Familiarizing Yourself with Raspberry Pi Desktop

To complete this assignment, you will need access to Raspberry Pi Desktop, including its C++ compiler and requisite supporting tools, libraries, and packages. It is likely easiest to build yourself a virtual machine running this system; details on how to do so can be found under Useful Links in the OWL site side bar. You should do this as early as possible to make sure you are set up and ready to go for the assignment.

If you have a computer that completely lacks virtualization support, Science Technology Services has a solution for remotely accessing something that is compatible for this work. They have created a cloud based Linux machine running Raspberry Pi Desktop; this can be found at cs3307.gaul.csd.uwo.ca. To access this machine, you should be able ssh to log in from pretty much anywhere, using your Western credentials for access. You can scp/sftp files to and from this machine as necessary.

Modeling for this Assignment

For this assignment, you will be creating a C++ class to help manage files on the Raspberry Pi Desktop system. (You might also be creating other support classes too, depending on how you do things.) This class will nicely encapsulate both information pulled from the file system for the file(s) in question, as well as operations that can be performed on the files, with each instance of the class handling a single file. This will be accomplished using a collection of system calls and file I/O operations. First we will discuss the data that you need to concern yourselves with and how to access it, and then we will discuss operations that should be supported by your class and how to execute them.

The file manager class you are to create will include at least the following information for this assignment:

- Name. The name of the file, given to the class through its constructor.

- Type. Whether the file is a regular file, directory, and so on. Not only is this useful information to have, but this will also allow you to permit certain operators on certain types of files. (When we explore design patterns in more detail, we will discuss a better way of doing this.). This can be retrieved using the stat() function, and can be found in the st_mode field of the structure provided by this function. You can store the type as a string representation or using the same numeric code used in the st_mode field.

- Size. The size of the file. This can be retrieved using the stat() function, and can be found in the st_size field of the structure provided by this function.

- Owner. The user who owns the file. This can be retrieved using the stat() function, and can be found in the st_uid field of the structure provided by this function. You must keep the numeric user ID from this field, as well as the string user name obtained using the getpwuid() function.

- Group. The group of the file. This can be retrieved using the stat() function, and can be found in the st_gid field of the structure provided by this function. You must keep the numeric group ID from this field, as well as the string group name obtained using the getgrgid() function.

- Permissions. The read, write, and execute permissions on the file. This can be retrieved using the stat() function, and can also be found in the st_mode field of the structure provided by this function. You can store the type as a string representation or using the numeric code used in the st_mode field. (When you go to print this later, it should always be printed in the familiar rwx notation, whether you store it that way or not.)

- Access time. The time of last access. This can be retrieved using the stat() function, and can be found in the st_atim field of the structure provided by this function.

- Modification time. The time of last modification. This can be retrieved using the stat() function, and can be found in the st_mtim field of the structure provided by this function.

- Status change time. The time of last status change. This can be retrieved using the stat() function, and can be found in the st_ctim field of the structure provided by this function.

- Block size. The block size for the file, to determine the optimal chunk size for I/O operations for the file. This can be retrieved using the stat() function, and can be found in the st_blksize field of the structure provided by this function.

- Children. If the file is a directory, this field may contain a collection of file objects for the various files under the directory. (This field is only populated by the expand member function below; otherwise, this contains no child objects.). For storing these objects, you should look into using a vector. While this could be done with an array, a vector is likely easier and better in the long run.

- Error number. Many of the file operations performed using your class will set the errno variable in your program if something goes wrong. You are to cache the value from the most recent operation executed on this particular file

object; it should be initialized to 0 and set to 0 on each successful operation. If the operation failed, however, it should get the value of errno at that point, so the user can see what exactly went wrong later.

Your class will also need at least the following member functions to permit operations on the file associated with the class, and to support the class as well:

- Constructor. This creates an instance of the class and initiatives it. It will take the name of a file as a parameter and use the stat() function to initialize everything else. It will also initialize its error number, as noted above. Note that constructing an instance of this does not actually create the file in the filesystem if it does not exist. (If the stat() function fails, because the file does not exist for example, the attributes should be initialized to indicate that the file object is invalid and should be destroyed.)

- Destructor. This destroys and frees up any resources attached to the object the destructor was called on. Even if you think you have nothing to do, you are still required to have a destructor. Note that destroying an object does not delete the corresponding file that the object is associated with. There is a separate method for doing that.

- Dump. This function will take a file stream as a parameter and dump the contents of the named file to that file stream. This can be used to copy the file, display its contents to the terminal, and so on. For performance reasons, you should use the block size of the file (one of the attributes from above) to determine the amount of data to be read from the file and written to the file stream at a time. This function should return some kind of error code if the operation could not be completed, and it should store the error number generated in the process in the appropriate attribute. This function can only be used on regular files; attempts to do this on other types of files should have an error returned, with the object's internal error number set to something appropriate like ENOTSUP.

- Rename. This changes the name of the file from its existing name to the new name provided as a parameter to this function. In addition to changing the corresponding attribute of the file object in question, this will change the name of the file on disk. This can be done using the rename() system function. This function should return some kind of error code if the operation could not be completed, and it should store the error number generated in the process in the appropriate attribute. Note that this function may allow files to be moved from one directory to another, but will not likely allow files to be moved from one filesystem to another.

- Remove. This removes the file from the file system. Once completed, this function should clear out or reset the attributes of the file object in question as this object no longer refers to a file that exists any more. (In theory, the caller of this function should destroy this object after the successful removal of the file too.) This can be done using the unlink() system function. This function should return some kind of error code if the operation could not be completed, and it should store the error number generated in the process in the appropriate attribute. (Note that you do not need to ensure that this works on directories as well as other files ... whatever unlink() works on is fine for us.)

- Compare. This function takes another one of the file objects as a parameter and will compare the contents of the file object this function was invoked upon to this other file object. For performance reasons, you should use the block size of the file (one of the attributes from above) to determine the amount of data to be compared at a time. (While the other file could have a different block size, don't worry about that for now.) The function should return some kind of indication of whether the files were the same or if the files were different. It does not need to note what the differences were. This function should also return some kind of error code if the operation could not be

completed, and it should store the error number generated in the process in the appropriate attribute.

- Expand. This function operates on directories only and is used to fill in the children of the file object this function was invoked upon. It does so by obtaining a listing of the directory in question and creating a new file object for each file found in the directory, adding these new objects to the collection of children as it goes. For assistance on working with directories, you might want to consult the man pages for opendir(), readdir(), and closedir(), although there are certainly other ways of doing this. This function should return some kind of error code if the operation could not be completed, and it should store the error number generated in the process in the appropriate attribute. This function can only be used on directories; attempts to do this on other types of files should have an error returned, with the object's internal error number set to something appropriate like ENOTSUP.

When creating your class, keep the following in mind:

- Your class will need its own header and source code file so that it is a reusable entity on its own.

- As noted above, each instance of the class handles or manages a single file. In use cases where you need to work with multiple files, each file has its own separate instance of your class doing the work for it.

- Each piece of information noted above should be a separate attribute in your class. Your class must not simply keep the stat structure returned by the stat() function; instead you must extract the relevant information from the structure and pack it into the various attributes in the constructor for the class.

- You do not need to use the above names for your attributed and member functions, but you should pick appropriate and intuitive names for each.

- For each attribute, you should provide a getter method to allow the attribute to be retrieved from outside of the class. (Direct access to the attributes should not be permitted.) For error processing, you should have a getter method to retrieve the error number as a number and a second getter method to retrieve the error as a string. (This can be accomplished using the strerror() function.) Setter methods are optional for this class, except for the name of the file for renaming purposes; this setter method should simply call your rename function as noted above. The other information generally cannot be manipulated by the class.

Writing / Packaging the Utilities

Of course, just having this new class on its own is just a start. We need some utilities that actually, well, utilize them to do useful things for the user. For this, you're going to create simpler versions of the standard system commands mv, cp, ls, cat, rm, diff, and stat called mymv, mycp, myls, mycat, myrm, mydiff, and mystat, Here are a few notes on each.

- mymv: This is used to move and rename files around, and you will use the rename method of your class from above to do this. Note that if you receive an EXDEV error or other error indicating that the source and destination are on different file systems, you should try to use the dump and remove methods to copy the file and then remove the original instead. Note that you only have to do this copy-delete alternative for regular files; in theory, you could do something similar for directories too, but then you would need to copy-delete things recursively. (Not impossible, but not a lot of fun either. You can try doing so for fun if you like, but you don't have to do this.)

- mycp. This is used to make a copy of a file from a source to a destination named as parameters to the command, and the dump method from above should work nicely for this on its own.

- myls. For a directory, this will list the contents of the directory and for other types of files, it will show the file's name, just as ls would. (And if no file is specified as a parameter to myls, it simply lists the current directory.) Directory listing will use the expand function from your class, as described above. You should also implement a "-l" option for this command (the only time you have to have an option to one of your new commands) so you can do a long form listing of things as the original ls would. (For fun, you could look at doing a recursive version of ls like you would get through the "-R" option. You don't have to do this though.)

- mycat. Like the original command, this will display the contents of all of the files given as parameters to the command to the terminal. (You will need to support more than one file as the original command does.) Your dump function from above should help you do this as well. This should work fine for text files, but may produce weird results for binary files. The original cat does that too, so no worries there.

- myrm. This removes the given files listed as parameters to this command. It does so using the remove method you have implemented above.

- mydiff. This compares two files named as parameters to this command with one another and reports if they are the same or different. This can be done using your compare function in your class above. Note that unlike the original diff, you do not need to list or identify the differences between the files, even if they are plain text. Simply reporting if they are the same or different will suffice.

- mystat. This outputs all of the information on the file named as a parameter to this command. You only need to concern yourself with the attributes that are part of your object, and not the ones you ignored when calling the stat() function. You can format this output how you like, but you can try the stat command to see what it does. You should try to ensure that the data is as readable as you can to a person, giving type, owner, and group names as text, as well as the set of permissions. Times should also be reported in a reasonable fashion.

If any of your new commands encounters an error, you should retrieve the error string from your class and report it so that the user knows what went wrong. Except where noted above, you do not need to implement all the various options from the original commands that your new commands are based off of. You can experiment with doing so as you like, but it is not required for this assignment.

Note that each of these commands will require its own C++ file with its own main() function and will need to be linked with your class from above in order to provide a working program. In some cases, the code for the command will be incredibly simple as your class above will do most of the heavy lifting, and that's okay. So, by the time you are done you will at least have mymv.cpp, mycp.cpp, myls.cpp, mycat.cpp, myrm.cpp, mydiff.cpp, and mystat.cpp, along with another .cpp file and .h file for your file manager class. (You may also have other files, depending on how you implemented things.)

Coding Guidelines

Your code should be written in adherence to the Coding Guidelines available here. Deviations will result in deductions from your assignment grade up to 10% of its overall value.

Submitted Attachments

- FileManager.cpp (9 KB; Oct 2, 2019 11:01 pm)
- mycat.cpp (1 KB; Oct 2, 2019 11:01 pm)
- mycp.cpp (1 KB; Oct 2, 2019 11:01 pm)
- mydiff.cpp (1 KB; Oct 2, 2019 11:01 pm)
- myls.cpp (8 KB; Oct 2, 2019 11:01 pm)
- mymv.cpp (1 KB; Oct 2, 2019 11:01 pm)
- myrm.cpp (1 KB; Oct 2, 2019 11:01 pm)
- mystat.cpp (2 KB; Oct 2, 2019 11:02 pm)
- FileManager.h (2 KB; Oct 3, 2019 10:35 am)

Instructor's attachments to this submission

- CS3307 Individual Assignment Grading Sheet.xlsx (13 KB; Oct 9, 2019 2:51 pm)

Back to list