Project 1 (20 points)
Assigned: Monday, March 12, 2018
Due: Friday, March 23, 2018, 11:59 PM

CS-4513, Distributed Systems
D-Term 2018

CS4513 Project 1

Dumpster Diving

You've been working on your MQP for the last 2 terms are well on your way to writing the final report. At 4am, fuzzy-brained after pulling 2 all-nighters, you accidentally type **`rm mqp.tex`**. *Argh, weeks of effort wasted! You vow not to be bitten the same way at the end of your MQP. But how to avoid this? Then, inspiration strikes you … you'll use your new-found file system knowledge to build a way of doing automatic file backups instead of just deleting them!*

Description | Hints | Experiments | Turn In | Grading

Description

Your dumpster diving facility will consist of several utilities that can take the place of existing system utilities. Basically, you will have a version of **`rm`** that puts deleted files into a separate "dumpster" (a hidden directory) rather than actually deleting them. Users can then recover these "deleted" files with another utility, named **`dv`** (Unix-speak for "dive"). A **`dump`** program completely removes all files in the dumpster. All utilities should work silently if successful but should report appropriate error messages when unsuccessful.

It is expected that each user has a dumpster directory specified with the **`DUMPSTER`** environment variable. Your programs can expect this directory to be created ahead of time. If the dumpster directory is not created, your utilities should display and appropriate error message and exit.

You must do your coding in C/C++ in Linux, using your virtual machine from the operating system course (CS-3013). If you do not have a functioning virtual machine, two virtual machines can be found at the following link:–

http://poppy.ind.wpi.edu/~ciaraldi/VMs

See the Professor if you need additional assistance on setting up this or any other virtual machine.

Three utilities

The three file utilities that you must develop are described in the next three sections:–

RM

You must write a program named **rm** (intended to transparently replace **/bin/rm**) that moves files to a directory specified by the user using the **rename()** system call. (Note, you do not (and should not) actually replace the **/bin/rm** call. Rather, a user can just be sure your new **rm** appears first in the *path*.) If the file to be removed is on the same partition as the dumpster directory, the file should not be copied but instead should be renamed (or hard-linked). If the file being removed is not on the same partition as the dumpster directory, your **rm** must copy the file and then delete it (via the **unlink()** or **remove()** system call).

If a file with the same name already exists in the dumpster, the new file should receive a **.num** extension, where **num** is the next consecutive number (e.g., 1, 2, 3 ...) up to a maximum of 9.

The file permissions, including access times, should be preserved. You can use the **stat()** system call to gather these values when a new entry must be made, and **chmod()** and **touch()** to set them.

Your version of **rm** should support the following command line options:

- **-f** : force a complete remove, do not move to dumpster
- **-h** : display a basic help and usage message
- **-r** : copy directories recursively
- **file [file ...]** – one or more file(s) to be removed

Feel free to support other command line options that you think are useful. For example, you might want a **-d** option that allows removal to a different dumpster that is specified by name.

If the file to be removed is a directory (and the **-r** flag is given), **rm** should move the directory and its contents to the dumpster. If the file to be removed is a directory but the **-r** flag has *not* been given, **rm** should return an appropriate error message. As for files, permissions and access times should be preserved when removing a directory.

Note, paths are relative to the running **rm** process, with the exception of that a path that begins with a "**/**" is absolute. For example, "**/tmp/a**" is absolute, but "**tmp/a**" is relative to the directory **tmp**.

DV

As part of this assignment, you must also write a program named **dv** to (potentially) restore any file that has been "deleted" by your **rm** program. Your **dv** looks in the dumpster directory and, if the indicated file is found, moves it to the current working directory — *not* to the original directory from which it was deleted. You can use **getcwd()** to obtain the current working directory. If **dv** is called on a directory, the directory and all of the files inside should be restored.

If a file or directory with the same name already exists in the current directory, `dv` should report an appropriate error message and exit.

The file permissions, including access times, should be preserved. You can use the **`stat()`** system call to obtain this information when a new entry must be made.

Your `dv` command should support the following command line options:

- **`-h`** : display basic usage message
- **`file [file ...]`** : file(s) to be restored

Feel free to support other command line options that you think are useful. For example, you may want to support a **`-a`** option that lists (or restores) all files with the same name, but also those with a **`.1`**, **`.2`**, ... extension.

DUMP

For the third part of this assignment, you must write a program named **`dump`** to permanently remove files from the dumpster directory. To actually remove files, `dump` should use **`unlink()`** and for directories, **`rmdir()`**. Alternatively, you may use the **`remove()`** Linux function. Your **`dump`** command should recursively remove files and directories, as needed, until the dumpster is empty.

Your **`dump`** command should support the following command line options:

- **`-h`** : display basic usage message

Feel free to support other command line options that you think are useful.

Hints

You *may not* use the **`system()`** Linux function at all, nor **`fork()`** nor any flavor of **`exec()`**! Likewise, no third-party libraries are allowed (e.g., you *may not* use the **`boost`** libraries.)

For development questions, consider the cs4513 question-answer forum. Both the professor and TA will look to answer all questions there, but students can also answer each other's questions. You may even find your question has been answered already!

For help in parsing command line parameters, you might see the sample program `get-opt.c`, which uses the user-level library call **`getopt()`**. See the manual pages on **`getopt`** for more information.

See `stat.c` for sample code on how to get status information (using **`stat()`**) about a file, including permissions, access time, and directory information.

See the sample program env.c for an example on how to (more easily) access environment variables (using **getenv()**), such as **DUMPSTER**.

See **ls.c** for sample code on how to do a directory listing (say, for recursive removal of a directory).

See **touch.c** for sample code on how to set the modification times (using **utime()**) on a file.

Other system calls that may be useful:

- **chmod()** – change (user, group, other) permissions
- **chown()** – change ownership
- **link()** – add a hard link to a file
- **unlink()** – delete a file
- **rmdir()** – remove a directory
- **remove()** – delete a file or directory
- **rename()** – change the name or location of a file
- **getcwd()** – get current working directory for a process
- **access()** – can be used to check for existence of a file
- **open()** – to open a file
- **creat()** – to create a file with a specific mode
- **umask()** – to set a file mode creation mask (e.g., umask(0) before calling creat()).
- **basename()** and **dirname()** – split full path to **dir** and **filename** (warning! can modify incoming string so copy first)

You can use **perror()** to print appropriate text-based strings for system call errors, or **strerror()** to more generally get the same error string. The include files **<errno.h>** and **<linux/errno.h>** may be useful for using error codes.

Lastly, although not required for this project, you could easily put an entry into your **crontab** to execute **dump** periodically (say, once per week).

If you are having trouble getting started, you might start with **rm** and consider the following notes:

In general:

- Proceed incrementally, write SMALL pieces of code and then test.
- When appropriate, place code in separate a function to aid readability and testing (e.g., **int getExtension(char *name)** – could return the extension number (**.1**, **.2**, ...) for a file that is already in the dumpster, using **.0** if there is no needed extension).
- Do not worry about efficiency nor elegance in initially getting functions to work (e.g., using static arrays with a reasonable maximum size to avoid cumbersome memory management with **malloc()** is fine).
- Refactor code, as appropriate to aid readability and functionality as development proceeds.
- In most cases, there is more than one way to do something. For example, checking if a file to be moved is on another partition can be done using the **stat()** call, looking at **st_dev** or by doing the **rename()** call, when an **errno** of **EXDEV** could be set.

Suggested development steps for **rm**:

- Setup program, ensuring at least one command line argument (**argv[1]**) and **DUMPSTER** set using **getenv()**.
- Move file in current working directory to **DUMPSTER** on same partition using **rename()**.
- Add check for same name using **access()**, adding extension **.num**.
- Add support for single file in another directory (e.g., **/tmp**), using **dirname()** and **basename()**.
- Check if file to be moved is on another partition using **stat()** with **st_dev** or **EXDEV** error set by **rename()** call.
- For files on another partition, write your own "copy" function to move across partitions, removing the previous file with **unlink()** once done.
- Modify copy to preserve permissions using **stat()** and **utime()**.
- Check if file to be moved is a directory using **stat()** and **S_ISDIR()**.
- For directories on another partition, write function to iterate through directory using **opendir()** and **readdir()**, providing each file name for moving.
- Parse additional command line arguments (e.g., **-r**) using **getopt()**.
- Modify code to recurse through sub-directories, as appropriate, moving all to **DUMPSTER**.
- Enable iteration through all filenames passed in on command line.
- Error check many test cases thoroughly.

The final overall **rm** flow design may look something like the following:–

```
Parse command line arguments
Check environment setup
For each file to be removed
  If file does not exist
    Report error
  If file is directory
    Create directory in dumpster
    For each file to be removed ... (recursive)
  If file is on another partition
    Copy file to dumpster
    Change permissions & access times on file
  If file is on same partition
    Link file to dumpster
  Unlink old file
end For
```

Diving and dumping should be relatively easy once **rm** is complete.

Experiments

After you have implemented and debugged your utilities, you will then design experiments to measure: 1) the amount of time required (the latency, in milliseconds) to perform either a `link()+unlink()` or a `rename()` system call; 2) measure the throughput (in bytes per second) in copying a large file across separate partitions; 3) use data from 1 and 2 to estimate the time for doing an `rm -r` (using your `rm`) on a large directory (10s of directories with 10s+ of files) in a separate partition, then measure this time. Note, you should look into using `sync()` to ensure your disk operations are actually flushed to the disk and not cached. For both sets of measurements, you will need to do multiple runs in order to account for any variance in the data between runs.

To measure the `rename()` or `(un)link()` system calls, since the time scale for a single test is very small, you will measure the time for many operations and then divide by the number of operations performed. You will want to build a harness (a program, shell, perl or python script, or something similar) to make repeated requests.

In order to record the time on your computer (instead of, say, looking at the clock on the wall) you should use the `gettimeofday()` system call from a program, `localtime()` from a perl script or `/usr/bin/time` from a shell (note, some shells have a built-in `time` command, too).

The exact means you use to gather timing is up to you. You could put in timing hooks in your program, perhaps that can be turned on or off via compile options or command line programs, that you use for measurements. Another recommendation is to leave as much of your programs intact as you can. In this case, you might initiate timing from a shell script that calls your program, or from a separate timing program you write program that calls your program. Whatever method you use should be specified in your writeup.

When your experiments are complete, you must turn in a brief (1-2 page) writeup with the following sections:

1. *Design* - describe your experiments, including: a) what programs/scripts you ran and what they did (use pseudo-code); b) how many runs you performed; c) how you recorded your data; d) what the system conditions were like; e) and any other details you think are relevant.
2. *Results* - depict your results *clearly* using a series of tables or graphs. Provide statistical analysis including at least mean and standard deviation.
3. *Analysis* - interpret the results. Briefly describe what the results mean and what you think is happening and any subjective opinions you may have.

Submission

You must use a **`Makefile`** for this project (it is good for you and good for us since it makes grading easier). The program **`make`** is a useful program for maintaining large programs that have been broken into many software modules. The program uses a file, usually named **`Makefile`** that resides in the same directory as the source code. This file describes how to compile a number of targets specified. To use, simply type "**`make`**" at the command line. WARNING: The operation line(s) for a target MUST begin with a TAB (do not use spaces). See the man page for **`make`** and **`gcc`** for additional information.

You must submit the following:–

- A source code package:
  - Your **rm.c**, **dv.c**, and **dump.c**. Note! Make sure your code is well-structured and commented if you expect to receive partial credit.
  - Any other support files, including **.h** files.
  - A **README.txt** file explaining: files, code structure, how to run and build, and anything else needed to understand (and grade) your project.
  - A **Makefile** for building your utilities.
- A document in pdf format describing your code, your tests, and your submission.

Before submitting, "clean" your code (i.e., do a "**make clean**") removing the binaries (executables and **.o** files).

Use **zip** to archive your files. For example:

```
mkdir lastname-proj1
cp * lastname-proj1 /* copy all the files you want to submit */
zip -r proj1-lastname.zip lastname-proj1 /* package & compress */
```

To submit your assignment (**proj1-lastname.zip**), log into the Instruct Assist website:

https://ia.wpi.edu/cs4513/

Use your WPI username and password for access. Visit:

Tools → File Submission

Select "**Project 1**" from the dropdown and then "Browse" and select your assignment (**proj1-lastname.zip**).

Make sure to hit "Upload File" after selecting it!

If successful, you should see a line similar to:

```
Creator     Upload Time            File Name        Size     Status    Removal
Claypool 2016-01-22 21:40:07  proj1-claypool.zip   3208 KB  On Time   Delete
```

Grading

A grading guide shows the point breakdown for the individual project components. A more general rubric follows:

100-90. All three utilities meet all specified requirements. Programs are robust in the face of possible errors, such as insufficient file permissions or user errors. Code builds and runs cleanly without errors or warnings. Experiments effectively test all required measurements. Experimental writeup has the three required sections, with each clearly written and the results clearly depicted.

89-80. The three utilities meet most of the specified requirements, but a few features may be missing. Programs are robust in the face of most errors. Code builds cleanly and runs mostly without errors or warnings. Experiments mostly test all required measurements. Experimental writeup has the three required sections, with details on the methods used and informative results.

79-70. The three utilities are in place, but with only the minimal functionality and without all required features. Code compiles, but may exhibit warnings. Programs may fail ungracefully under some conditions. Experiments are incomplete and/or the writeup does not provide clarity on the methods or results.

69-60. Not all of the utilities are in place and/or significant parts are not functional. Code compiles, but may exhibit warnings. Programs may fail ungracefully under many conditions. Experiments are incomplete and the writeup does not provide clarity on the methods or results.

59-0. Not all of the utilities are in place and for the code that is there, significant parts are not functional. Code may not compiles without fixes. Programs may not run under even slightly more advance conditions. Experiments are incomplete with a minimal writeup.

Description | Hints | Experiments | Turn In | Grading

Return to 4513 Home Page

Send all questions to the staff mailing list (**`cs4513-staff`** at **`cs.wpi.edu`**).