National and Kapodistrian University of Athens
Department of Informatics and Telecommunications

[K22] Operating Systems

Programming Assignment #4

February 2024

Michael-Raphael Kostagiannis sdi2100078
George Sofronas sdi2100180

Contents

1. About

All the requirements of the project have been fulfilled. The **cmpcat** program can be used to both print the differences between two existing directory hierarchies and merge the two hierarchies into a new one. If the program compares two identical hierarchies then nothing is printed in the tty. However for every difference inside the two hierarchies a message is printed whether those differences concern directories, files or links. In case the user wants to merge the catalogs, the new hierarchy created will contain both the common entries found in the two hierarchies and their unique differences without any redundancies. Both comparing and merging are done by utilizing information found in i-nodes. Also, the program completes both operations with zero memory leaks and zero memory errors. We recommend using the memory error detector `valgrind`. You can achieve this by running the following command on the tty: `valgrind ./cmpcat <args>`

2. Compilation and Execution

Along with the source and header files, the .zip file contains a **Makefile** which automatically compiles the necessary source files using separate compilation and produces the **cmpcat** executable. Object files are stored inside a real-time created directory called `build/`. The available commands are listed in the following table:

Compilation command:	`make`
Cleanup command:	`make clean`
File–accounting command:	`make count`

As far as the program's execution is concerned, the assignment's rules have been strictly followed. There are two available flags:

- **-d**: This flag is **mandatory**. It should be followed by two paths, one leading to input-directory A and the other leading to input-directory B.

  The execution command, only using flag `-d`, is: `./cmpcat -d pathTo/dirA pathTo/dirB`

- **-s**: This flag is **optional**. If provided, the program will produce a new directory generated by merging input-directories A and B. It is important to note that if the new directory doesn't already exist, the program will automatically create it. However, if the directory does exist, it can be used as an argument only if it is empty. It is also important to note that if the new directory is found in either hierarchy A or B, the program will not execute properly. This happens due to the fact that finding the differences and merging happen concurrently, meaning that the new directory's entries will be marked as differences and be re-merged, causing an endless loop.

  The execution command is: `./cmpcat -d pathTo/dirA pathTo/dirB -s pathTo/output`

The flags as well as the paths in the -d argument can be given in any order. Also, the program can be called using either relative or absolute paths as arguments and the executable can even be found inside one of the path arguments. Since the program uses paths as its arguments, all of the following paths are valid:

- `data/dirA`
- `./data/dirA`
- `./data/dirA/`

- `/home/User/.../data/dirA`
- `~/Desktop/.../data/dirA`

In order to be consistent, we have reached the following agreement: All paths end with '/', all relative paths start with './' and all absolute paths start with '/'.

3. datatar.tar

This compressed file contains some test cases for our program. The reason we decided to compress the folders onto a single .tar file is to preserve the symlink and hardlink properties of some files for easy and accurate sharing. In order to properly untar it, run the command `tar xf datatar.tar` and the corresponding `data` folder containing all test cases should appear. Let's dive a little bit deeper into the test cases:

3.1. projectDir_A and projectDir_B These two hierarchies represent the hierarchies 'dirA' and 'dirB' found on the assignment's examples. Common files found in both hierarchies, such as `B,A1,C.txt and file.csv` have the same contents. The purpose of these test files is to make sure that the program can correctly distinguish differences between common and unique files and folders as well as correctly merge them.

3.2. dirA and dirB These two hierarchies are essentially 'evolved' versions of the assignment's examples. More files were added (e.g. `dirA/accounts50.bin, dirA/A, dirB/B.txt etc.`), common files now have different contents and both symlinks and hardlinks were added (e.g. `dirA/hardlinkToA`, `dirB/symlinkOutside` etc.). The purpose of these test files is to ensure all types of entries are properly compared as different or not and then merged correctly. Symlinks that point outside their respective hierarchies are named as `symlinkOutside` and should be omitted according to the assignment.

From this point onwards, all remaining test folders have the exact same structure:

- A file called `A.txt` with the contents 'same content'
- An entry called `folder`. This entry can be a file, a directory or a symlink. If `folder` is a directory, it contains a file `B.txt` with 'foo' as its contents.

The purpose of these test cases is to depict the behavior of the program when two entries with the same name and different file type are encountered. According to the assignment, we only keep the entry that was last modified, ignoring file type.

3.3. fileIsNewer_A and fileIsNewer_B Here, B's file called `folder` was created after A's directory called `folder`. Keep B's file.

3.4. fileIsOlder_A and fileIsOlder_B Here, B's directory called `folder` was created after A's file called `folder`. Keep B's directory as well as all of its contents.

3.5. linkIsNewer_A and linkIsNewer_B Here, B's symlink called `folder` was created after A's directory called `folder`. Keep B's symlink.

3.6. linkIsOlder_A and linkIsOlder_B Here, B's directory called `folder` was created after A's symlink called `folder`. Keep B's directory as well as all of its contents.

4. utils.h and utils.c

These files contain utility functions based around file reading/writing and string manipulation. `utils.h` also contains the macro command `NULL_CHECK(cond, funcName)` which checks whether `funcName` returned the value `NULL`, in which case it exits.

4.1. char *get_absolute_path(char *from, char *relative); 'from' string represents a starting path in absolute form. 'relative' string represents a relative path starting from the starting path and ending to an entry inside any hierarchy. Concatenate those two paths together in order to get the absolute path of the entry by correctly replacing all `../` and `./` occurrences with each parent and current folder respectively.

4.2. char *fix_path(char *path); Takes a path as an argument and normalizes it according to the agreement mention above.

5. avltree.h and avltree.c

These files contain a basic implementation of AVL trees. The implementation is based on the provided source code of the course **[K08] Data Structures**. The only difference concers the data stored in each tree-node, which instead of `int` or `void *` (as in the provided implementations), a pair of `ino_t` and `char *` is used to store i-node number and path respectively (Check out the explanation of function `manage_hardlinks()` for details).

6. info.h and info.c

These files contain some useful information which will be shared among all files. This information is encapsulated in a struct called `GlobalInfo` which includes the following:

- Absolute paths of hierarchies A, B and C
- Absolute path of the executable (process)
- Lengths of the above strings
- An AVL tree which will be used to efficiently manage hardlinks

The above fields are initialized in `info_init()` using `realpath()` and de-allocated using `info_destroy()`.

7. entry_manager.h and entry_manager.c

These files contain all necessary information about an entry as well as some useful functions regarding entry management. Starting with the macros, all `#defines` are used for readability purposes. Furthermore, the struct `EntryInfo` as well as its constructor(`entry_init()`) and destructor(`entry_destroy()`) are described in detail on both files. Most functions are self-explanatory by reading the corresponding comments, however here are some notable functions that require further explaining:

7.1. int symlink_in_hierachy(char *symlink, char hierarchy); As mentioned in the *Piazza forum*, for simplicity purposes the program only cares about the final destination-file of a symlink-chain. Therefore, the function calls `realpath()` to dereference the symlink chain and get the absolute path of the final destination-file (completely resolved, without `../` or `./` because of `realpath()`). Since we have the absolute path of the symlink's hierarchy in *global* `info`, we can now compare the first `n` bytes of the file with the hierarchy, where `n` is the string length of

the absolute path of the hierarchy. If they match, then the file is somewhere under the hierarchy and the function returns 1 (true), otherwise 0 (false).

7.2. void manage_hardlinks(EntryInfo *entry, char *destination); The function is responsible for creating a mirror of the hardlinks' relationships found in the input-hierarchies A, B in the new hierarchy C. The idea is simple. Since all hardlinks to the same disk-file are equivalent, create the first one found using `copy_file()` and the next ones using the `link()` system call, linking to the file that was initially created (first hardlink).

However, iterating through the whole array to find files with the same i-node can be expensive. Thus, an AVL tree is used. Each tree-node stores a pair of `ino_t` and `char *` to save *i-node* number and *path* of the firstly created file mentioned above. With this structure in play, the hardlinks' creation algorithm becomes very simple. When encountering a hardlink, we search the AVL tree with its *i-node* as key and the mentioned *path* returned in case of successful search.

- If the *i-node* was found (a *path* was returned), the disk-file has been created in the new hierarchy (through `copy_file()`). Having its path, the program simply calls the `link()` system call to create another hardlink to the same file.
- Otherwise (`NULL` was returned), the corresponding disk-file hasn't been created yet in the new hierarchy. Therefore, the program calls `copy_file()` to create the file and inserts its *i-node* and *path* to the AVL tree for future references to it by the next hardlinks to the same file.

Apart from making the algorithm simple to implement, the AVL tree also offers $O(\log n)$ worst-case time complexity for both `search` and `insert` operations. The AVL tree was picked instead of other binary search trees (ex. Red-Black tree) because it has the most strict balance criterion, keeping the tree as short as possible and therefore making the `search` operations as fast as they can be. This is crucial for this program, since the function calls `insert` only once for each "hardlink-group" (for the first encountered hardlink) but `search` for all the hardlinks.

It's important to mention that the assignment's instructions and rules dictate the criterea that should be met in order to copy over a hardlink to the new hierarchy. For example, for two input-hierarchies `dirA/`, `dirB/` suppose we have these files:

- `dirA/TEST.txt` and `dirA/hardlinkTEST.txt` are hardlinks to the same disk-file with *i-node* x
- `dirB/TEST.txt` and `dirB/hardlinkTEST.txt` are hardlinks to the same disk-file with *i-node* y

Since the files are on the same hierarchy-level and have same names, the program will decide on which to keep based on the most recent modification time. There is a possibility that for instance `dirA/TEST.txt` and `dirB/hardlinkTEST.txt` are chosen, which in the new `dirC/` hierarchy are called: `dirC/TEST.txt` and `dirC/hardlinkTEST.txt`. Despite their naming, **these are (and should be) different disk-files** with different *i-nodes*. This edge-case is being discussed so as to clarify that the mentioned output has nothing to do with this function's implementation, but with the program's **required guidelines**.

8. wrapper.h and wrapper.c

These files contain an entries-array wrapper, which stores all the necessary information needed in order to optimize the program's throughput. The `EntryInfo **array` groups in a 1D array all the entries found in the hierarchy.

In order to minimize the array iterations and maximize performance, the `EntryInfo *` pointers are stored in the array **by hierarchy level**. Hierarchy level is defined by the depth of directories needed to be entered until the entry (regfile, directory, ...) can be reached. For example, all entries under the initial hierarchy `dirA/` are on level 0. The entry `dirA/d1/myfile.txt` is on level 1, the directory `dirA/d1/d11/d111/` is on level 2 and the symlink `dirA/d1/d11/d111/mySymlink` is on level 3. Details on why and how this approach vastly increases performance are described in the `cat_manager` section. Inside the `ArrayWrapper` struct, an `int` array called `levels[]` is also being stored. In each position `levels[i]`, this array stores the index of the first element of **level i** in the entries' array. Conceptually the start-index of each level is being stored.

8.1. static void array_ init(char *path, ArrayWrapper *wp); This internal (static) function achieves the array initialization described above. Two dynamic 1D-string-array buffers are used to store directory-paths of current and next/new level respectively. Let's go through the algorithm. Suppose we are currently on level `n` of the hierarchy. The goal is to traverse all its directories (currently stored in `currDirs` buffer) in order to:

1. Appropriately store valid entries (all except for dangling symlinks) in the `wp->array[]`
2. Store all the subdirectories in the `newDirs` buffer, to traverse later on the next level

The outer `while` loop simply executes one time for each level. The index of the first element (start) of each level is stored in the `wp->levels[]` array.

The middle `while` loop traverses over the directories of the current level. For example, if the only directories on level 0 are `dirA/d1` and `dirA/d2`, then when expanding level 1, both directories `dirA/d1` and `dirA/d2` will be traversed.

The inner `while` loop traverses all entries of current directory of current level, assuring that the above mentioned two goals are being met. In the above example, all entries of directories `dirA/d1` and `dirA/d2` will be accessed.

After all the directories of the current level have been iterated, moving on to the next level, `currDirs` is now the previous's level `newDirs`. This process is repeated until there are no more levels found (= no more subdirectories).

This algorithm assures that the hierarchy's entries as stored in the `wp->array[]` **by level**.

9. cat_manager.h and cat_manager.c

As already mentioned, the user has two options when executing the `cmpcat` program. (1) Only use `-d` flag and print the catalogs' differences or (2) also use `-s` flag and not only print the catalogs' differences but also merge them into a newly created catalog. For choice (1), the function `find_differences()` is being called. For choice (2), the function `find_and_merge()` is being called.

The functions' source code alongside their comments explain very efficiently their functionality. Nevertheless, it is vital to mention that the fact that the arrays have all their entries stored by level massively improves the functions' speed and efficiency. Suppose that the entries weren't stored this way. Then, the inner loop would have to iterate over the whole `wrapperB`'s array and also check that the currently examined entries `wrapperA->array[i]` and `wrapperB->array[j]` are on the same **hierarchy level**! This would hinder performance, costing

1. Complete iterations of `wrapperB`'s array
2. Many unnecessary extra calls to `strcmp()` function

Fortunately, the current implementation assures that each entry on level n is only compared to entries of the same level. In addition to that, every extra level of a hierarchy (one hierarchy has more levels than the other), is immediately marked as a difference (and copied to the new hierarchy) without any condition-checks.

10. cmpcat.c

This file is the glue that sticks it all together. It is responsible for all of the following actions:

- Correctly parsing user arguments and checking if execution command is valid using `parse_args()`
- Allocating and de-allocating all necessary memory for auxiliary structs to avoid memory leaks and errors
- Executing the two different modes of the program by either calling `find_differences()` or `find_and_merge()` depending on the number of arguments the user passed