New York University Abu Dhabi
CS-UH 3010 (001) - Spring 2018
Programming Assignment 4
Due: May 8th, 2018

Preamble:
In this assignment, you will write an application termed `mymirror` that initially creates and subsequently, helps maintain a replica of a specific LINUX directory for back-up purposes.

Your program should function as a daemon and should *dynamically follow* changes that occur in the *source* (or *origin*) directory. Such changes should be automatically reflected in the *target* (or *destination*) directory. The latter, also known as back-up copy, should accurately maintain both the entire structure of the source directory and produce consistent copies for files and links found in the portion of the file-system being watched. To accomplish this, you will be using the `inotify` monitoring subsystem *API* [1, 2].

The invocation of `mymirror` is as follows:
`prompt >> mymirror source backup`

Your program should carry out the following two actions:

1. Initially, when the `backup` does not exist, `mymirror` should "replicate" the content of source to that of target so that both have exactly the same information (logical structure and data).
2. Subsequently, your program should start monitoring the source (and all its subdirectories) and when a change occurs, `mymirror` should immediately become "aware" and take the appropriate corrective action to (re-)synchronize by properly updating the file-system structure and/or content of the backup or destination directory.

Procedural Matters:
◇ Your programs are to be written in `C/C++`
◇ You will have to first submit your project to newclasses.nyu.edu and then, demonstrate your work.
◇ Nabil Rahiman (`nr83-AT+nyu.edu`) will be responsible for answering questions as well as reviewing and marking the assignment.

Preliminaries:
The LINUX file-system (FS) consists of nested directory structures and files. Both directories and files maintain symbolic names and are represented in the FS-structure using `i-nodes`. Every directory or file is represented by an `i-node` that maintains all the pertinent information of the respective entity in the FS including number of `i-node`, date of last modification and size among others. Although directory-names and `i-nodes` maintain an 1-to-1 relationship, we might have multiple file-names pointing into the same `i-node`. This is what *hard-links* do. In general, the logical structure of a FS can be depicted through a directed acyclic graph (or tree) as:

- Directories may contain other directories on their own in a recursive manner.
- There are no hard links when it comes to directories. This helps avoid cycles in the FS logical structure.

Figure 1 depicts the structure of one catalog (*drectory1*) that features a nested catalog (*drectory2*) with 4 files. Each of the FS-entities is represented by an `i-node` but *file1* and *file2* are essentially the same file as they point into the same `i-node`.

You can find out the `i-node` of a file (or directory) either through the use of the system program `ls -i <filename>` at the shell-level or you can use the system call `stat()`. LINUX is particularly helpful for it provides a system program *also called* `stat` that provides `i-node` (and other) information. We should note that the creation of a hard link is attained through the use of the command: `ln <existingfile> <newfile>`.

Regarding the use of `inotify` start from reading the manual page (`man inotify`) that provides a good overview of the monitoring subsystem. Also, we provide a short program, `samplenotify.c` so that you can

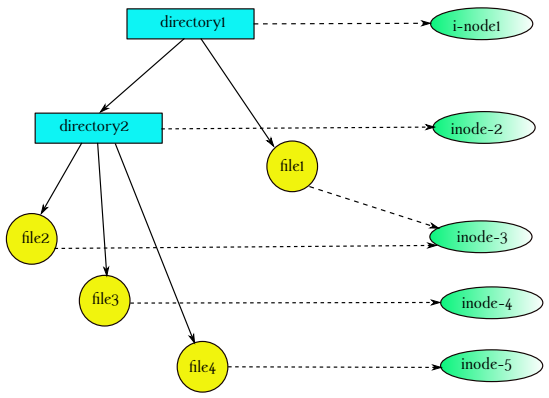

Figure 1: Both logical organization and `i-node` layout of *directory1*

gain some insight as far as the use of `inotify` is concerned. In this program, please note that the structure that describes an event (`struct inotify_event`) is of variable length [1] and so, it some attention is required while reading; `samplenotify.c` addresses this issue. Reference also the material found in [2, 1].

Realizing your Program:
During the initial phase of `mymirror` operation, you will have to create the appropriate structures that will reflect the name-hierarchies and corresponding `i-nodes` in a way very similar to that maintained by the LINUX FS. In particular, you will have to record the `i-nodes` that are part of your structure and the symbolic names that point to these `i-nodes`. At the end of this first phase, `mymirror` will have to maintain *synchronized structures* for both source as backup directories.

Let's assume that for either source or destination, your application maintains an array of `i-nodes` with each element providing the following information:

- last modification date,
- size of the file,
- list with all file names pointing to this entry
- number of files names pointing here (this can be computed on the fly), and
- especially for the source directory, there must be a pointer to the backup structure (i.e., replica).

Similarly, `mymirror` has to maintain a tree-structure for each of the two hierarchies whose consistency has to maintain:

- the symbolic name,
- pointer to `i-node`.

A possible algorithm that would synchronize source with backup hierarchies is as follows:

1. Create the above structures for both hierarchies by fetching information from the FS for each symbolic name; you may sort alphabetically the sibling nodes while working with the tree structure.
2. Simultaneously traverse the two trees in "depth first" fashion and:
   (a) If a name-catalog exists in the source but is missing from the target, create it on the target catalog. If the same name exists at the target directory but pertains to a file (and not a directory), just remove this element from the target before creating a new-entry with the same name for a directory.
   (b) If a name for a directory exists in the destination but not the sources, then just purge the name in question from the destination.

[1] the field `len` indicates how many characters follow in order to complete the name

(c) If a name-file appears in the source but not in the destination look up its `i-node` and find out if there is a replica of this `i-node` at the backup:
   i. If yes, link the name with the `i-node` found in the backup.
   ii. Otherwise, create a new name-file at the destination and link the `i-node` of the source with that of the destination.

(d) If a name-file does not exists in the source but appears in the destination, just unlink it.

(e) If a name-file appears in both source and destination, compare the information provided by their respective `i-nodes`. If they are the same (you can use date of last modification and size for this comparison), you forfeit any further action. Otherwise, unlink the name-file from the destination and proceed as in step (2c).

In the above algorithm, for each of the used phrases, the following system calls can be deployed:

- fetch information ⇒ `stat` system call
- create a file ⇒ `creat` or `open` system call
- delete/unlink a file ⇒ `unlink` system call
- create a catalog ⇒ `mkdir` system call
- delete a catalog ⇒ `rmdir` system call
- link a name with an existing `i-node`⇒ `link` system call

Note that the field `st_nlink` yielded by `stat` (see `man 2 stat`) is not helpful in the context of `mymirror` as it may entail names from outside of the source hierarchy. For this reason, the "number of file names pointing here" has to be explicitly maintained by your program.

In the second phase of `mymirror` operation, the monitoring of the directories in the source will commence. In brief, this is realized as follows:

1. create a queue of events using `inotify()` → `fd`
2. add the objects under monitoring with `inotify_add_watch()` → `wd` (watch descriptor). In our case, this pertains to all catalogs in the source hierarchy.
3. Continuously do:
   (a) read the next-event from the queue
   (b) depending on its type, carry out updates on the backup directory.

The events that are of interest to `mymirror` and the suggested way to address them are as Table **??** suggests.

Working Assumptions:

- The backup directory is not included in any way in the source catalog.
- The target hierarchy is not modified by any other process.
- The source hierarchy is not modified during the first step of `mymirror` synchronization work.
- If you have time, provide support for *symbolic links*. However, this is not a hard requirement.

What you Need to Submit:

1. A directory that contains all your work including source, header, make, a `README` file, etc.
2. A short write-up about the design choices you have taken in order to design your program(s); 1-2 pages in `ASCII`-text would be more than enough.
3. All the above should be submitted in the form of `tar` or `zip` file bearing your name (for instance `AlexDelis-Proj4.tar`).
4. Submit the above tar/zip-ball using *NYUclasses*.

Table for Events and Actions	
EVENT	Action
IN_CREATE	if it is a file find the i-node of the source If there exists already a copy link it otherwise, create a new one otherwise, if it is a catalog create a catalog at the destination add the new catalog in the objects under monitoring
IN_ATTRIB	If it is a file and the last date of modification has changed update the replica at destination
IN_MODIFY	(this event is received with *every* modification) If it is a file, mark it as modified
IN_CLOSE_WRITE	If it has been marked as modified, copy it
IN_DELETE	If it is a file, unlink it from the target
IN_DELETE_SELF	(This concerns only a catalog as we monitor only directories) Delete it from the target Take it off the list of monitored objects using inotify_rm_watch()
IN_MOVED_FROM	(A name has been moved outside of the monitored catalog) Make a note of the field cookie of the event. No action is taken until the next event. If the next event is IN_MOVED_TO follow up what the next down description says Otherwise, unlink the name (as it has moved outside the source hierarchy)
IN_MOVED_TO	(A name has been moved/introduced within the monitored hierarchy) If the field cookie is the same with that recorded just above (the movement is within the monitored hierarchy of catalogs) move the name respectively to the target catalog Otherwise, act as in IN_CREATE moreover, copy the data.

Table 1: Events and Respective Actions to be Taken

Aspect of Programming Assignment Marked	Percentage of Grade (0–100)
Quality in Code Organization & Modularity	15%
Correct Execution for FS Changes/Insertions/Deletions	25%
Addressing All Requirements	20%
Handling Hard-Links	15%
Use of Makefile & Separate Compilation	7%
Well Commented Code	8%
Division of Labor Justification/Log and Use of LINUX	10%

Grading Scheme:

Noteworthy Points:

1. The project is to be done either ***individually*** or in groups of at most ***two***. If you chose to work in pairs you will *have to document* the division of labor.
2. You **have to use *separate compilation*** in the development of your program.
3. Although it is understood that you may exchange ideas on how to make things work and seek advice from fellow students, **sharing of code is *not allowed***.
4. If you use code that is not your own, you will have to provide ***appropriate citation*** (i.e., explicitly state where you found the code). Otherwise, plagiarism questions may ensue. Regardless, you have to fully understand what and how such pieces of code do.

References

[1] Michael Kerrisk, *The Linux Programming Interface*, No Starch Paper, San Fracisco, CA June 2010, portion of the book can be found on *newclasses*.

[2] Ian Shields, *Monitor Linux file system events with inotify*, `https://www.ibm.com/developerworks/library/l-inotify/` April 2006.