Introduction

This application is a file hosting service, offering storage and synchronization among clients.

mirror runs a client's code. First off, it creates a file in the common directory named after the client's id, like a sign-up. After that, it copies all data of other users under his mirror directory creating a sub-directory for each user containing his files and folders. Finally, it monitors the system every now and then, inspecting possible changes, like new registrations or deletions of users and makes sure the client's mirror directory is always up to date and in sync with the directories of other users . More details about the functional requirements of this application can be found in the project's description/analysis.

How it works

- ### Syncing

  There are two cases where syncing is required. The first is when a client enters the system and the second is when a user insertion or deletion is detected while monitoring. In order to update the mirror directories both for the new user and the one that detected his arrival, each mirror process forks twice. The first process is supposed to send files and the second to receive files via named-pipes using low-level i/o. They both accomplish that sending/receiving folders recursively. So, before sending a file, a **header** defining whether it is a directory or a file is required. In case of an error, which would either be an open or read block on a named-pipe for at least 30 seconds[1] or a directory/file creation type error, a SIGUSR signal is sent to the parent process.

- ### Signals

  Three sets of signals might have to get handled during mirror's operation.

  1. **SIGALRM**

     In this case, the invoking process, a child, either a sender or a receiver, follows the pattern mentioned a couple of lines above and also de-allocates any heap memory it might have been using at the time.

[1] This is implemented using the **alarm** system call, setting the alarm before each open/read and resetting when it's done

2. **SIGINT/SIGQUIT**

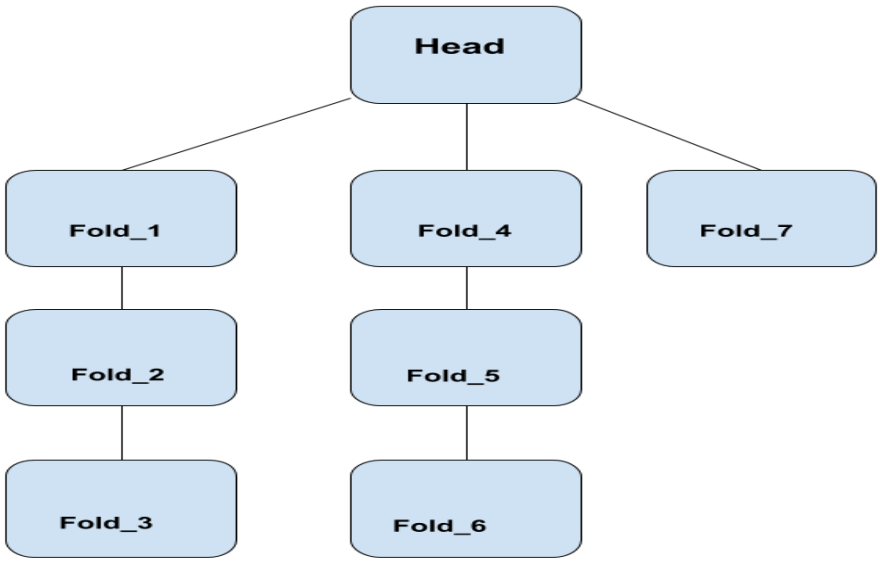
   mirror is a daemon process, meaning it runs forever, monitoring every now and then and inspecting new arrivals/deletions. To stop it while managing to set all resources and heap-allocated memory free, a SIGINT/SIGQUIT handler is defined doing just that.

3. **SIGUSR**

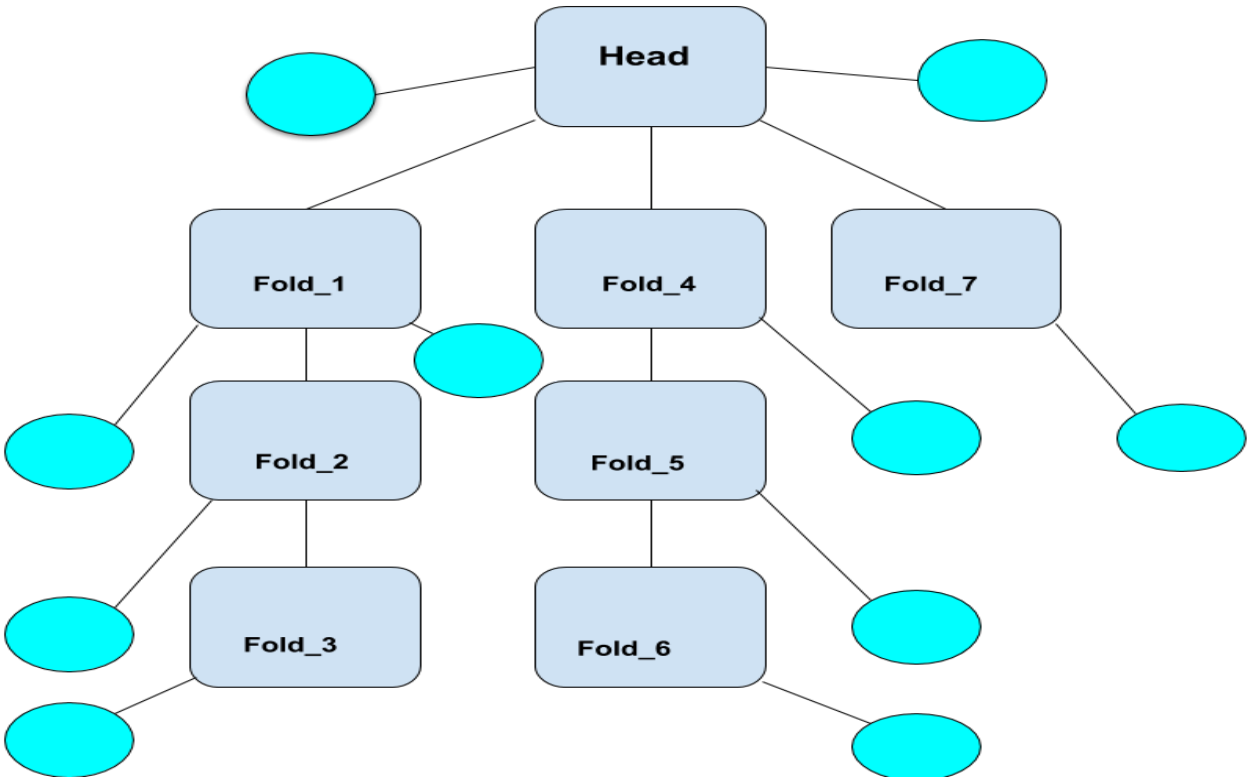
   In case of a sending error, a SIGUSR1 signal is sent by the sending process to its parent. Similarly, the receiving process sends a SIGUSR2. A handler is defined in the parent process, incrementing a counter indicating the number of times synchronization was attempted. After three unsuccessful attempts, synchronization is cancelled.

Input Directories

To make sure mirror is working properly some input directories had to be generated for each user. The bash script *create_infiles.sh* does just that. The input directory of a client is passed as an argument. The number of directories, files and levels to distribute those under the input directory are also passed as parameters. A random name is assigned to each folder/file created. Files are initialized with random-content using a system program called *fortune*. First off, the script creates a chain of directories iteratively under the input directory. The length of the chain is at most $levels. If fewer directories than $levels are left to create the last chain contains that many directories. For example, creating 7 directories using 3 levels under an input folder named “Head” looks like this.



After that, directories need to be filled with files. So until we create as many files as requested we assign one file to each folder, starting from “Head”, traversing the folder-tree using DFS. So, creating 10 files in the system pictured above would result in a full cycle of 8 assignments, plus 1 more to the “Head” and “fold_2”.



create_infiles.sh uses /dev/urandom to generate random directory names and the program *fortune* to initialize files with random content. If those are not supported by your system use *create_infilesII.sh* instead. This script uses another one, called *names.sh*, which generates random strings of any length requested using an environmental variable called RANDOM.

Statistics

Apart from the create_infiles.sh script, another one, **get_stats.sh** is used to get statistics from the users’ log files[2]. This script reads from stdin and supports the following format:

id code (bytes) ...[3] where:

[2] After a successful transfer, both the sender and the receiver write a message in the log file of the client. Each client has a unique log file so these processes need to be synchronized, using flock syscall to make sure they are not both writing into the file at the same time.
[3] Any number of arguments might follow

- **id** is a positive integer, a client's identifier
- **code** can be assigned s(end),r(eceive) or a(bort)[4]
- **bytes** is the number of bytes send or received, depending on the code

After iterating on the data it prints in stdout a sort id list, containing unique client ids. After that, the maximum and minimum value of those follow. In addition, the script prints out how many bytes and files were sent/received. Finally, it displays the number of clients that left the system(code = a).

Testing

In this project, there are three main points we need to test.

1. What happens when a client requests from another to get synchronized but he does not respond. Run src/*fake_user.sh* to find out.
2. What happens when the other client responds and
3. What happens when a client leaves the system.

To test the last two, run *src/new_user.sh* in a terminal window and then run src/*run_user41.sh* in another. The new user will be detected and get synchronized with the other user. When user 41 leaves the system, the sub-directory containing his input that was created under the other client's mirror directory will be deleted. Commands like 'tree' or 'diff -sr' are run towards the end of the first script to verify just that.

Putting it all together

A script that connects all components of this project can be found under the *scripts* directory and is called *testing.* Yeah, that was obvious. So, this script gets one argument which is the number of users involved in the simulation. It creates a folder named *dbox* and under this folder, it creates the following directories:

- **common**: This is where each user is registered, creating a file name after his id, ending with '.id'.
- **inputs**: Containing an input directory for each user, containing 5 files distributed in 3 folders, 2 levels deep using the *create_infiles.sh* script.
- **mirrors:** Containing a mirror directory for each user
- **logs**: Containing all log files. One for each user**.**
- **trash**: This is where mirror's output is redirected. There will be one file for each user, containing mirror's messages generated during the simulation.

[4] in case of an abort the 3rd field is omitted

After that, it runs a mirror process for each client and sleeps for a while. Then, it sends a SIGINT signal to all mirror processes in the system, so that they terminate normally. Finally, it concatenates all log files and uses *get_stats.sh* to print the simulation's statistics in a file called 'stats' under dbox directory.

Epilogue

All source files are very well commented and each critical code section is explained in detail. So here, I just tried to emphasize on the main idea, so that we can get a generic, high-level look on the project. Hopefully, I made things clear.

Thank you.