HW3.design

October 6, 2023

1 HW3 Design

1.1 *LINK()*: Hard Link

To acomplish this function should be able to -

1.1.1 *shell.py*

- add `lnh` command definition with two input - file source and target link name. Since we are creating the hardlink in context of the current working directory, the value for the target directory will be managed by the variable `cwd`

1.1.2 *absolutepath.py*

0. Performing validations => first priority is to perform validations and implement error handling as required in the porblem.
1. After validation, all we need is -
    a. inode number of the file (target) we want to create a link for
    b. cwd inode number and then call FileNameObject.InsertFilenameInodeNumber() to insert the new (filename-inode) entry into the directory inode table
2. Finally, we need to increase the reference count for the target file and the cwd Inode by one.

1.2 *SYMLINK()*: Soft Link

Symbolic links unlike hard links, creates a pointer to the original data directory/file. (they do not even have the same permissions as the original directory/file)

We need a type for the symlink file as 'INODE_TYPE_SYM' as given in the file system.

1. find and create a new inode to hold information about the symlink. - FindAvailableInode()
2. validate whether the total available block_size is sufficient to store the original file path. - replicate checks on the bounds like Write operation. (inode offset + len(original_file_path) < fsconfig.MAX_FILE_SIZE)
3. allocate an available and required number of blocks to store the original file path. and append the allocated blocks to the inode.block_numbers[] array.
4. write the original file path to the allocated blocks. and increment the reference count for the symlink inode.
5. save the symlink inode information into blocks - StoreInode()

6. save the inode <-> symlink_name mapping into the directory table - InsertFilenameInodeNumber()
7. we now increament the directory inode reference count with 1 and save this information. - StoreInode()

Note: we don't need to increment the value of reference count for the original file since the symlink is not referencing but rather pointing towards this.

1.3 *PathNameToInodeNumber()*

The method design has been implemented via textbook and it consists of the following steps -

1. get the inode number for the input path
2. if the inode.type says that the path provided is for a symlink -> fetch the block_numbers and retrieve the file_path
3. For the retrieval, we need to read the file block data. Important consideration is that we read only equal to the size of the file. (provided by the inode)
4. if there are multiple blocks - we concatenate all of the data to create one single file_path.
5. Finally, the 'GeneralPathToInodeNumber" method is called with the input of retrieved file_path.

1.4 Shell Commands' Modifications

1.4.1 *shell.py* > ls

Modify ls command to indicate symlinks.

Check if the inode.type == SYM, if true - retrieve data from the blocks for the file_path and create an array to log into the terminal -

@linkName -> /path/to/target/file

path_to_target_file is resolved by getting the block_numbers from the symlink inode and then retrieving the file_path from them.

1.4.2 *shell.py* > mirror, slice, cat, cd

update the default `FileNameOperation.FileNameObject.Lookup` with `AbsolutePathNameObject.PathNameToInodeNumber` function to be able to resolve pathnames.

1.5 Testing:

Along with manually putting checkpoints, logs and verifying the information in inodes and how that's changing with each operation; another source of checking the correctness was the hw3 diff file which was run after the implementation and highlighted 1 major and 2 minor bugs in the implementation.

Trailing spaces in the file_path while reading in PathNameToInodeNumber method - This was critical error as it affected multiple shell commands including - cat, append, mirror and slice. The output indicated that the path being referred is not found because inodenumber returned as -1.

Adding logging.debug messages, reading the raw inode information with `showinode` and block information helped identifying that there were unwanted characters (trailing spaces) coming into the retrieved file_path.

First I tried to fix that by triming the string but that didn't work. (still confused on this one)

This was fixed by reading only the required number of bytes from the block; equal to file size.

I tested for multiple variations too -

- creating nested directories.
- creating a file deep into nested directories.
- reading (cat) content from these files and writing back on the original one through a symlink. and much more.