OS Assignment 2

Derek Mao mm2180, Kevin Pei ksp98, Quzhi Li ql88
Tested on top.cs.rutgers.edu

!!! IMPORTANT !!!

You need to go into sfs.h and change the following line:

#define FS_FILE "/tmp/ksp98/testfsfile"

You need to change it to whichever testfsfile you're using to represent your memory. Otherwise the code won't work.

!!! IMPORTANT !!!

Introduction: This is a program meant to implement FUSE in order to create a simple file system. We implemented init, getattr, readdir, create, unlink, open, release, write, and read. In order to do so, we use blocks of memory to represent data.

Filesystem Layout: In order to represent data, we used the i-node p-node structure that Linux uses. I-nodes store information about a given file, such as the file size, when it was last modified, and the user that owns it. The i-node in turn contains several direct-mapped data nodes. If the file is too large to be stored in the direct-mapped segments, then a single-indirect mapping is used, where the i-node points to a p-node which itself stores direct-mapped data. If there still isn't enough room, then the i-node uses double-indirect mapping, which points to p-nodes which point to more p-nodes which each have direct-mapping. If the file is still too large to be stored using double indirect mapping, then the file is too large and no more can be written to the file.

Each block is 512 bytes large and are referred to by number. Our memory is laid out so that the first 30,000 blocks of our memory are devoted to i-nodes and p-nodes. This means that there is a max of 30,000 files in our filesystem. Our i-nodes also have 256 bytes dedicated to storing the path, meaning that the maximum path length of a file is 256 characters, including null characters. Our p-nodes don't need to store anything the i-nodes do, other than a mode variable to distinguish them from i-nodes. This means that p-nodes can store more direct-mapped data nodes than i-nodes can. Data nodes themselves are actually 511 bytes data, 1 byte variable to tell whether the data is in use. This is so that data that's already in use isn't overwritten.

Functions: The functions we implemented are as follows:

init: Initializes the filesystem with a root directory and flushes the i-node area. All of them are initialized to have 0 in all fields, meaning that the first 512 * 30,000 = 15 KB of the testfsfile are initialized to 0. This is so that files from previous mounting sessions won't interfere.

destroy: Removes the root directory.

create: Initializes a new i-node with all of the appropriate attributes. This says that setting time "Function isn't implemented", but you can ignore that. It does not affect the performance of this code.

unlink: Removes the i-node associated with the given path, as well as all p-nodes and data blocks associated with the i-node.

getattr: Returns information about the given file based on information stored in its i-node. Information includes user, permissions, date created, etc.

open: Gets a file handle for the file.

release: Removes the file handle for the given file and makes the file handle invalid.

read: Reads the given number of bytes into the buffer given, offset the given number of bytes from the start of the file.

write: Writes the given number of bytes from the buffer into the data of the file, offset the given number of bytes from the start of the file. Whenever you use echo, it assumes there is a new line character at the end of whatever you wrote, hence why when you use cat, it will show a new line character after each echo use.

readdir: Returns information about all the files in the current directory.

We also had more helper functions, like a function to get the first free i node or the first free data block.