FUSE OS-assignment 3

krabr21

May 2023

Contents

1 Introduction

In this assignment we are to implement a file system using FUSE. Implementing FUSE lets us implement the logic behind the file system in user-space, avoiding all the trouble around writing C-code in kernel-space, as kernel-space code can be troublesome. The implementation implements all of the regular file system methods, such as the support for making directories, files, updating the information of files and directories, removing files and directories. **The code is written with slaub21 & mikkn21**

2 Design choises

2.1 Folder placement

The backup folder used is created at *~/fusebackup*, i.e ~ has to be pointing to a folder, or the program will not work, if not altering the path to the backup-folder. Also, the FUSE-instance, has to be mounted onto */tmp/fuse*, else the backups will not be done using the correct folder.

2.2 Regular programming

Decided to make helper functions for the flow of the logic, through the implemented methods, to abbreviate the code to chunks of code, where what the code does, is specified via the function names, making it more readable and easier to understand. Futhermore, the implementation uses structs to contain the data, which further improves the readability, as we can read each field in the structs, and get a understanding, of what data is stored, and how the program will progress throughout the code, by filling the allocated structs with data. As the implementation is in C, and all argument-passing to functions are done via pass-by-value, i.e copying the actual arguments, pointers are used, for less copying of data, resulting in speed up of the file-system. The speed is further increased, by the fact that we also then use pointer manipulation, to go through each layer of the directories.

2.3 Errors

The implementation uses error-codes defined in *errno.h*, making the programs error-code easier to understand and debug, for other C-programmers using this standard of error-handling.

2.4 Memory sharing & concurrency

The implementation contains a daemon spawned by the mounter-program, which backups the contents of the file system to a local directory regularly. Using a background daemon to backup the file system, will allow for a more responsive file system, as the process running the actual file system, will use all resources

only for doing operations on the file system. When the FUSE file system is mounted to a directory on the hosts file system, it will also copy all contents from this backup directory, allowing for quick and easy use. To manage the synchronization between the process which runs FUSE instance and the backup-daemon, shared memory is being used, to allow for communication between specific variables. As the daemon does not have to backup every time it can, sleeping is used, such that there will be a specified interval of time, between each backup - which ensures the daemon is not hugging the CPU.

2.5 Backup strategy

The implementation uses an existing utility tool, called via the CLI, from within the code. This is to ensure a fast and reliable backup tool, storing the FUSE instance content, to a semi-huge file, where in this instance, a regular file and directory, seen from the UI perspective, is the same exact thing. I.e files and directories are both files.

2.6 File system implementation

In the implementation, files and directories are disguishable, where directories contains files and other directories, mimicking the UI when traversing through directories. As mentioned, these two structs hold different fields, containing different data.

3 Implementation

3.1 Helper structs and functions

The helper structs and functions used are

```
1  static struct dir_file_info *find_info(const char *path);
2  static void check_is_file(char *token, struct dir_data *dir, struct
       dir_file_info *info, int is_last_token);
3  static int check_is_dir(char *token, struct dir_data **dir, struct
      dir_file_info *info);
4  static char **extract_tokens(char *path, int n_tokens);
5  static int make_dir(struct dir_data *dir, char *name);
6  static int make_nod(struct dir_data *dir, char *name);
7  static int make_content(const char *path, mode_t mode, int is_file)
      ;
8  static struct path_info *get_path_info(const char *path);
9  static void free_path_info(struct path_info *path_info);
10 static int init_root();
11
12 struct dir_file_info
13 {
14   int is_dir;
15   int found;
16   void *item;
17 };
```

```
18
19 struct path_info
20 {
21   char *creation_path;
22   char **tokens;
23   int n_tokens;
24   char *not_const_path;
25 };
```

Where *find_info()* takes in an absolute path, and tries to find the struct dir_data or struct inode. This is done via traversing from the root-directory to the last token in path split upon the *'/'*-character. This uses the *get_path_info()*-method, to get the information about the path. This uses the *extract_tokens()*-helper-function, to get all the tokens. In the method, the entire *struct path_info* is created and filled with information. The *creation_path* is used when finding the directory containing the last token, whether it being a file or directory. Fx. if the path is */tobias/test.txt* 2 tokens are returned by the *extract_tokens()*-function i.e *["tobias", "test.txt"]*. We can then from the root scan the dir-list to see if there is a directory with the name "tobias", if there is, we update the pointer to go into this dir. As "test.txt" is the last token, we first scan if it is a directory, if it is we return a *struct dir_file_info **, with the field *is_dir = 1, found = 1, void *item = (the struct dir_data *)*. If it is a file it returns *is_dir = 0, found = 1, void *item = (the struct inode *)*. And if not found it is obvious. Figuring this out uses the *check_is_file()* and *check_is_dir()* for figuring out this information.

make_dir(), make_nod() will uses *make_content()* which in turn uses *find_path_info()* to retrieve the creation path, and calling *find_info()* with this creation path, finding the parent directory for which to make the file or directory, by calling *find_info()* with the creation path extracted within the method, passed via the *struct path_info()*.

init_root() will just initialize the root, which is a struct dir_data, and it is called in main().

3.2 Main-driver functions and structs specifications

The essential parts of the code, used to run the file system are the following structs

```
1 int lfs_getattr(const char *, struct stat *);
2 int lfs_readdir(const char *, void *, fuse_fill_dir_t, off_t,
      struct fuse_file_info *);
3 int lfs_open(const char *path, struct fuse_file_info *);
4 int lfs_read(const char *path, char *, size_t, off_t, struct
      fuse_file_info *);
5 int lfs_release(const char *path, struct fuse_file_info *fi);
6 int lfs_mkdir(const char *path, mode_t mode);
7 int lfs_mknod(const char *path, mode_t mode, dev_t dev);
8 int lfs_write(const char *path, const char *buf, size_t size, off_t
       offset, struct fuse_file_info *fi);
9 int lfs_truncate(const char *path, off_t length);
```

```c
int lfs_rmdir(const char *path);
int lfs_unlink(const char *path);
int lfs_utime(const char *path, struct utimbuf *file_times);

static struct fuse_operations lfs_oper = {
  .getattr = lfs_getattr,
  .readdir = lfs_readdir,
  .mknod = lfs_mknod,
  .mkdir = lfs_mkdir,
  .unlink = lfs_unlink,
  .rmdir = lfs_rmdir,
  .truncate = lfs_truncate,
  .open = lfs_open,
  .read = lfs_read,
  .release = lfs_release,
  .write = lfs_write,
  .rename = NULL,
  .utime = lfs_utime
 };

struct inode
{ // used like the Inode of the specific file
  char *name;
  __time_t access_time;
  __time_t modify_time;
  __mode_t mode;
  char *content;
  int size_allocated;
  int content_size;
};

struct dir_data
{
  char *name;
  struct dir_data *dirs;
  struct inode *files;
  int dir_count;
  int current_dir_max_size;
  int file_init_size;
  int file_count;
  int current_file_max_size;
  __mode_t mode;
  __time_t access_time;
  __time_t modify_time;
};
```

The struct inode is representing a file, holding information about the files, and the struct dir_data represents a directory. The struct fuse_operations specifies which functions should be used, when the using the FUSE instance. Fx when using *ls*-command in the CLI, the *readdir* attribute is used, which is a pointer to our implementation, the *lfs_readdir()* method.
Each of the lfs functions does exactly what their name specifies.
The *lfs_getattr()* gets an absolute path, and a *struct stat **, which we have to fill up given the path. Therefore this function will call *find_info()* with the passed path, and find the specific file or directory. If it is found and its a directory we

fill up the values of access- and modify time, the mode/permissions and number of links in this directory. If it is a file we fill up the same again, but also the current content size. If not found we simply return *-ENOENT* inditcating that there is no such file or directory. Furthermore, this method will update a shared variable indicating the file-system is ready, therefore it will be updating this shared variable to be equal to 1. Also introducing a memory barrier, such that it will do it when imediatly called upon. This will be explain the *main()*-explanation

lfs_readdir() will call *find_info()* with the given path, check if the returned *struct dir_file_info()* is a directory. If it is not, again *-ENOENT* is returned, indicating that there is no such directory or file. After the checks, we know we have found a directory, and we will continue to go through each of the *struct dir_data*'s and *struct inode*'s, filling up the given buffer with the names, via the *filler*-function passed in the arguments.

lfs_open() will call *find_info()* with the given patch, check if the returned is a file. If not the same error message is returned again. If the check passes, it will then proceed to fill up the *struct fuse_file_info ** given in the arguments, setting the *fh*-field in the struct to point to the found file. However the pointer to the file is first type casted to a *uint64_t*, for compatability reasons, and it will then update the access time of the file, as it has just been accessed. Returning 0 on success.

lfs_release() will set the given file handle to 0, i.e it does not have a pointer to a *struct inode* anymore. No checks are done to see if the given path is actually file, because it is just faster to write

```
1     int lfs_release(const char *path, struct fuse_file_info *fi)
2     {
3   fi->fh = 0;
4   return 0;
5     }
```

Since this function be executed in constant time.

lfs_read() grabs the file handle from the *struct fuse_file_info **, check if it is NULL or 0, since we in release set the file handle to 0, and in open set it to something else than NULL. After this check we type cast it to the correct type, a struct inode, granting access to its fields. We then figure out how many bytes we can copy, given the requested size of bytes to read, and the given offset. This is calculated such that we dont overflow into the buffer. lastly, we memcpy() into the buffer from the files content, update the access time of the file, and return how many bytes were transferred.

lfs_write() will with the received buffer, which is "size"-long, append to the files content, where the file is retrieved from the *struct fuse_file_info **. First

we realloc to the size of the content, to be able to store the extra characters, update the content size, then use memcpy() to append the content of the buffer to the content of the file, and return how many bytes appended.

lfs_mkdir() and *lfs_mknod()* both will just call *make_content()* with a boolean, whether its a file or a directory, and return whatever this function returns.

lfs_truncate) will from the given path find the file, if it doesnt exist, we return -ENOENT. If it does, we realloc the files content to only be of the argument "length" big, and update the *size_allocated* in the file, to match how big the file is after the truncation.

lfs_rmdir() and *lfs_unlink()* will both receive the absolute path, and find the actual file or directory. They both do their checks if the given path is a file or a directory, and fail if wrong information were given, fx if the path in rmdir is to a file. If the directory that is trying to be removed is nonempty, then -ENOTEMPTY is returned, since a directory that is non-empty, needs to be removed recursively. This means they both will try to find the parent-directory which contains this file or directory. We then find the correct index of the file or the directory in the parent-dir array of files or directories, and swap the last directory or file in the array, with the current index of the file or directory we try to remove. This is to keep a sequential array with no internal fragmentation in the array. After the removal of the file or directory, the parent-directory has its modification-time updated, and the methods return 0 on success.

lfs_utime() is called to update the modification time and access time of a given file or directory, found via the absolute path. This means that we call *find_info()* with the given path, if we dont find a file or directory then -ENOENT is returned, indicating no such file or directory. Else we update the access time and modify time to the given times in the *struct utimebuf ** provided. I.e setting our struct dir_data or struct inode to the *actime* and *modtime*. Returning 0 on success.

main()

```
1     int main(int argc, char *argv[])
2 {
3   //int cp_error_code;
4   int id, shmid;
5   int err = 0;
6   if (!root) err = init_root();
7
8   // Create a shared memory segment
9   shmid = shmget(IPC_PRIVATE, sizeof(int), 0600);
```

```
  if (shmid < 0) {
    err = 1;
  }

  // Attach to the shared memory segment
  shared_variable = (int *) shmat(shmid, NULL, 0);
  if (shared_variable == (int *) -1) {
    err = 1;
  }

  // init shared variable
  *shared_variable = 0;

  id = fork();
  if (id == 0 && !err) {
    // stall to startup fuse before loading backup
    while(!(*shared_variable))
    {
    __sync_synchronize(); // memory barrie
    }


    system("mkdir ~/fusebackup");
    system("cp -R ~/fusebackup/ /tmp/fuse/ ");

    while(*shared_variable) {
    __sync_synchronize(); // memory barrier, for no compiler
    opmitimization
    system("rsync -a --delete /tmp/fuse/ ~/fusebackup/");
    sleep(1);
    }
    exit(0);
    } else {
    fuse_main(argc, argv, &lfs_oper);
    *shared_variable = 0;
  }

  // Detach from the shared memory segment
  shmdt(shared_variable);

  // Destroy the shared memory segment
  shmctl(shmid, IPC_RMID, NULL);


  return err;
}
```

The main function will do a call to initialize the globally scoped variable *root*, which is a *struct dir_data*, and if the initialization goes wrong, it will return the error code provided by the initialization code. This can only fail if the malloc-function fails. Next, since we use a variable, *int *shared_variable* to indicate whether the fuse-filesystem has been initialized, we create a shared memory segment, which can only be accessed by the processes that the parent spawns. I.e we use the key *IPC_PRIVATE*, and use the perimission bits 0666, specifying its not a directory, can be written and read from the parent process, by root and by everyone (such that the spawned process can also read and write). We also

only allocate the size of an int, since we only want to share a signle int. Next we initialize the shared variable, by getting a pointer to the shared variable, we initialize using *schmat()*, where we specify the id of the shared segment, specify we dont care where the variable is placed and also specify that no special flags should be used. Lastly we set it to the initial value, that the program has not yet been initialized, i.e shared_variable = 0. We then create the child thread, which will be used for backuping the system, with some time interval specified, and also copying the existing backup, if there is any. We setup a busy wait, using a while-loop. In this while loop we setup a memory barrier, such that the compiler will not update the structure of the program, such that this while loop may not be where we expect it to be. This is needed, since we have to be sure of when the file-system is ready. This variable is updated to be equal to 1 in *lfs_getattr()*, when it has received the first call, meaning it is ready. When exiting the while loop, we know that FUSE has been initialized and is ready to handle incoming call to creation of directories and files. Therefore we call the CLI, via using the *system()*-function, calling to make the backup-directory, located at */fusebackup*, if it is already there it will throw an error, which is piped into */dev/null*. Next it will call to copy the contents of this backup into the folder */tmp/fuse*, where the FUSE-instance has to be located. Then, while running the FUSE-instance, it wil backup the directory using *rsync -a –delete /tmp/fuse/ /fusebackup*, syncing all the files that has been changed, and deleting the files/directories that has been removed in the FUSE-instance. Then it sleeps for a interval, and backups again. When the program is exitted the child process will call *exit()* to exit.
The parent thread will call *fuse_main()* with the *struct fuse_operations*, which holds pointers to our method. When unmounting the program, it will continue to the lines of code after this call, where it will set the shared variable to 0, indicating the child process should exit the while loop, stop syncing and exit itself. The shared variable is the detached from the shared memory segment, and it is the closed. If there were any errors, the errors will be returned, or else it returns 0.

4 Recover if power failure

If the system encounters a power failure, the disc will only have the state of the FUSE-instance at max 10 seconds ago. Therefore when powering up the machine again, you are ensured to have almost all of your saved changes, that is, if you didn't do some major updates in the last 10 seconds before the power failure. If the synchronization has failed due to another power failure, the disc will still be intact, and you can try again when it boots backup to still synchronize with the full backup. I.e it does not matter if the synchronization with the disc and the FUSE-instance encounters a power failure.

5 Tests

5.1 1. create dirs

create directories, and show directories via *tree-command*. This shows we can create directories and subdirectories with the correct permissions and flags.

5.2 2. remove dirs and subdirs recursively

using *rm -r* on the created dir which contains all the directories. using tree to show that all directories are removed.

5.3 3. test resizing

Since we initialize the directory list and file list in the directory structure to allocate place for 10 files and 10 directories, and when we get passed this limit we use realloc, we show that if we make 11 of each, that there in fact are 11 of each, and not 10 or some error. After they are made, we use tree to show the result.

5.4 3. touch and remove

Shows we can create files and directories, also showing you cannot create files and directories with duplicate names.

5.5 4. duplicates

Showing you cannot create a file with the name of an already existing directory and vice versa.

5.6 5. stat

Showing the implementation updates the access and modification time correctly of files and directories.

5.7 6. backup management

Shows that when we make directories and files in the FUSE-instance they will be backed up to the backup directory stored on the disc, in the correct format. To show this, we write to a file in the FUSE-instance, cat it to show its content, and next we cat the file on the back-up to see it has the same content. We then close the FUSE-instance and spawn a new one, showing that it has synchronized with the backup, and it has all the content of the backup.

5.8 7. text editor

Showing we can open the files with a text editor, modify them and save them.

6 Conclusion

In this project we have developed a file system in user space using FUSE. Allowing for our own logic in the implementation of how files, directories, permissions etc. are altered and created. The file-system has the basic functions of listing, creating, removing write and read files and directories, and being able to change path into directories. The file-system lives in main memory, therefore it is synchronized with the disc, such that work done in the FUSE-instance is not lost when it is unmounted again.