CS537 Fall 2024, Project 6

Administrivia

- **Due Dates:**
  - Deadline for `mkfs.c` is by November 27, 2024 at 11:59 PM, *no slip days possible*
  - Deadline for the rest is by December 6, 2024 at 11:59 PM
- **Questions**: We will be using Piazza for all questions.
- This project is to be done on the lab machines, so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- **Handing it in**:
  - Copy the whole project, including solution and tests folder, to ~cs537-1/handin/login/p6 where login is your CS login.
  - Be sure to `make clean` before handing in your solution.
  - Only one person from the group needs to submit the project.
- **Slip Days**:
  - In case you need extra time on projects, you each will have 2 slip days for the final three projects. After the due date we will make a copy of the handin directory for on time grading.
  - To use a slip days or turn in your assignment late you will submit your files with an additional file that contains only a single digit number, which is the number of days late your assignment is(e.g 1, 2, 3). Each consecutive day we will make a copy of any directories which contain one of these slipdays.txt files. This file must be present when you submit you final submission, or we won't know to grade your code.
  - We will track your slip days and late submissions from project to project and begin to deduct percentages after you have used up your slip days.
  - After using up your slip days you can get up to 90% if turned in 1 day late, 80% for 2 days late, and 70% for 3 days late, but for any single assignment we won't accept submissions after the third days without an exception. This means if you use both of your individual slip days on a single assignment you can only submit that assignment one additional day late for a total of 3 days late with a 10% deduction.
  - Any exception will need to be requested from the instructors.
  - Example slipdays.txt

```
1
```

- **Collaboration**:
  - The assignment may be done by **yourself or with one partner**. Copying code from anyone else is considered cheating. Read this for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
  - When submitting each project, you will submit a `partners.txt` file containing the cslogins of both people in the group. One cslogin per line. Do not add commas or any other additional characters.
  - Only one person from the group needs to submit the project.
  - Partners will receive the same grades for the project.
  - Slip days will be deducted from both members of the group if used. If group members have unequal numbers of slip days, the member with the lower number of days will not

be penalized.

Introduction

In this project, you'll create a filesystem using FUSE (Filesystem in Userspace). FUSE lets regular users build their own file systems without needing special permissions, opening up new possibilities for designing and using file systems. Your filesystem will handle basic tasks like reading, writing, making directories, deleting files, and more.

Objectives

- To understand how filesystem operations are implemented.
- To implement a traditional block-based filesystem.
- To learn to build a user-level filesystem using FUSE.

Background

RAID

RAID (Redundant Array of Independent Disks) is a data storage technology that combines multiple physical disks into a single logical unit to improve performance, reliability, or both. It uses techniques like data striping, mirroring, and parity to distribute and protect data across the disks. Common RAID levels include:

RAID 0: Stripes data across disks for better performance but offers no redundancy.
RAID 1: Mirrors data on multiple disks for redundancy.
RAID 5: Distributes data and parity across disks, balancing performance and fault tolerance.
RAID 10: Combines mirroring and striping for high performance and redundancy.
RAID is widely used in servers and storage systems to enhance data integrity and availability.

In this project, you will implement filesystem with integrated RAID capabilities, which will resemble modern filesystems like BTRFS and ZFS. RAID can be implemented on the filesystem layer (our case) or block device layer (md-raid on Linux, hw disk controllers). We will implement only RAID 0 and RAID 1. For more information about RAID, use OSTEP, favorite search engine, language model or knowledgable human.

FUSE

FUSE (Filesystem in Userspace) is a powerful framework that enables the creation of custom filesystems in user space rather than requiring modifications to the kernel. This approach simplifies filesystem development and allows developers to create filesystems with standard programming languages like C, C++, Python, and others.

To use FUSE in a C-based filesystem, you define callback functions for various filesystem operations such as getattr, read, write, mkdir, and more. These functions are registered as handlers for specific filesystem actions and are invoked by the FUSE library when these actions occur. Callbacks are registered by installing them in the struct `fuse_operations.`

Here's an example demonstrating how to register a getattr function in a basic FUSE-based filesystem:

```
#include <fuse.h>
#include <stdio.h>
#include <string.h>
```

```c
#include <errno.h>

static int my_getattr(const char *path, struct stat *stbuf) {
    // Implementation of getattr function to retrieve file attributes
    // Fill stbuf structure with the attributes of the file/directory indicated
by path
    // ...

    return 0; // Return 0 on success
}

static struct fuse_operations ops = {
    .getattr = my_getattr,
    // Add other functions (read, write, mkdir, etc.) here as needed
};

int main(int argc, char *argv[]) {
    // Initialize FUSE with specified operations
    // Filter argc and argv here and then pass it to fuse_main
    return fuse_main(argc, argv, &ops, NULL);
}
```

This code demonstrates a basic usage of FUSE in C. The `my_getattr` function is an example of a callback function used to retrieve file attributes like permissions, size, and type. Other functions (like read, write, mkdir, etc.) can be similarly defined and added to `ops`.

The `fuse_main` function initializes FUSE, passing the specified operations (in this case, `ops`) to handle various filesystem operations. This code structure allows you to define and register functions tailored to your filesystem's needs, enabling you to create custom filesystem behaviors using FUSE in C.

`fuse_main` also accepts arguments which are passed to FUSE using `argc` and `argv`. Handle arguments specific to your program separately from those intended for FUSE. That is, you should filter out or process your program's arguments before passing `argc` and `argv` to `fuse_main`. If the first argument to your program is not meant to be passed to FUSE, then `argc` in `main()` would need to be decremented by one, and the elements of `argv` in `main()` would need to be shifted down by one as well.

Mounting

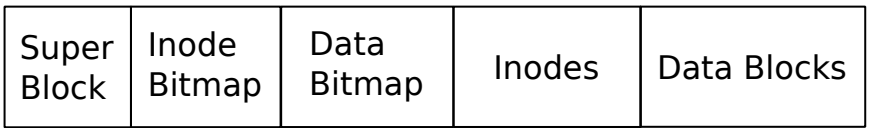
The mountpoint is a directory in the file system where the FUSE-based file system will be attached or "mounted." Once mounted, this directory serves as the entry point to access and interact with the FUSE file system. Any file or data within this mount point is associated with the FUSE file system, allowing users to read, write, and perform file operations as if they were interacting with a traditional disk.

A file can represent a container or virtual representation of a disk image. This file, when mounted to the mountpoint for the FUSE file system, effectively acts as a disk image, presenting a virtual disk within the file system.

When the FUSE file system is mounted on a file (such as an image file), it's as if the contents of that file become accessible as a disk or filesystem.

Filesystem Details

We will create a simple filesystem similar to those you have seen in class such as FFS or ext2. Our filesystem will have a superblock, inode and data block bitmaps, and inodes and data blocks. There are two types of files in our filesystem -- directories and regular files. Data blocks for regular files hold the file data, while data blocks for directories hold directory entries. Each inode contains pointers to a fixed number of direct data blocks and a single indirect block to support larger files. You may presume the block size is always 512 bytes. The layout of a disk is shown below.



Inodes

At this point, we would like to explicitly state that every inode always starts at the location divisible by 512. Do not pack inodes close to each other and "allocate" full 512B for each inode. In other words, superblock and bitmaps are stored continuously in the beginning of the disk without any padding. Inodes and data blocks are always aligned to the block size (512B).

Creating a file

To create a file, allocate a new inode using the inode bitmap. Then add a new directory entry to the parent inode, the inode of the directory which holds the new file. New files, whether they are regular files or directories, are initially empty.

Writes

To write to a file, find the data block corresponding to the offset being written to, and copy data from the write buffer into the data block(s). Note that writes may be split across data blocks, or span multiple data blocks. New data blocks should be allocated using the data block bitmap.

Reads

To read from a file, find the data block corresponding to the offset being read from, and copy data from the data block(s) to the read buffer. As with writes, reads may be split across data blocks, or span multiple data blocks.

Removing files and directories

`unlink` and `rmdir` are responsible for deleting files and directories. To delete files, you should free (unallocate) any data blocks associated with the file, free it's inode, and remove the directory entry pointing to the file from the parent inode.

Your implementation should use the structures provided in `wfs.h`.

Project details

You'll need to create the following C files for this project:

Part 1 --- `mkfs.c` (15% of the grade)

This C program initializes a file to an empty filesystem. I.e. to the state, where the filesystem can be mounted and other files and directories can be created under the root inode. The program receives three arguments: the raid mode, disk image file (multiple times), the number of inodes in the filesystem, and the number of data blocks in the system. The number of blocks should always be rounded up to the nearest multiple of 32 to prevent the data structures on disk from being misaligned. For example:

```
./mkfs -r 1 -d disk1 -d disk2 -i 32 -b 200
```

initializes all disks (disk1 and disk2) to an empty filesystem with 32 inodes and 224 data blocks. The size of the inode and data bitmaps are determined by the number of blocks specified by `mkfs`. If `mkfs` finds that the disk image file is too small to accommodate the number of blocks, it should exit with return code -1. `mkfs` should write the superblock and root inode to the disk image.

Part 2 --- `wfs.c` (85% of the grade)

This file contains the implementation for the FUSE filesystem. The bulk of your code will go in here. Running this program will mount the filesystem to a mount point, which are specifed by the arguments. The usage is

```
./wfs disk1 disk2 [FUSE options] mount_point
```

You need to pass `[FUSE options]` along with the `mount_point` to `fuse_main` as `argv`. You may assume `-s` is always passed to `wfs` as a FUSE option to disable multi-threading. We recommend testing your program using the `-f` option, which runs FUSE in the foreground. With FUSE running in the foreground, you will need to open a second terminal to test your filesystem. In the terminal running FUSE, `printf` messages will be printed to the screen. You might want to have a `printf` at the beginning of every FUSE callback so you can see which callbacks are being run.

Features

Your filesystem needs to implement the following features:

- Create empty files and directories
- Read and write to files, up to the maximum size supported by the indirect data block.
- Read a directory (e.g. `ls` should work)
- Remove an existing file or directory (presume directories are empty)
- Get attributes of an existing file/directory\
  Fill the following fields of struct stat
  - st_uid
  - st_gid
  - st_atime
  - st_mtime
  - st_mode
  - st_size
- Support RAID0 and RAID1 modes (More details in RAID section)

Therefore, you need to fill the following fields of `struct fuse_operations`:

```
static struct fuse_operations ops = {
  .getattr = wfs_getattr,
  .mknod   = wfs_mknod,
  .mkdir   = wfs_mkdir,
  .unlink  = wfs_unlink,
  .rmdir   = wfs_rmdir,
  .read    = wfs_read,
  .write   = wfs_write,
  .readdir = wfs_readdir,
};
```

See https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/homework/fuse/fuse_doc.html to learn more about each registered function.

Structures

In `wfs.h`, we provide the structures used in this filesystem. It has been commented with details about each of them. In order for your filesystem to pass our tests, you should not modify the structures in this file. You must not modify this file except the Superblock modifications mentioned below.

RAID

`wfs` is raid-only filesystem. It means, it always uses RAID 0 or RAID 1 mode.

RAID 0 (Striping)

To create the filesystem in RAID 0 mode, `-r 0` parameter is passed to `mkfs`. To make this project easier, we apply the RAID 0 only to data blocks. Metadata always use RAID 1 (Mirroring) hence are just duplicated across multiple disks.

One data stripe is 512B. It means that first 512B of the file are written to disk 1, the second 512B to disk 2 etc. Write down formulas for finding disk and location based on the file offset and number of disks in the array.

RAID 1 (Mirroring)

To create the filesystem in RAID 1 mode, `-r 1` parameter is passed to `mkfs`. In this mode, all data and metadata are mirrored across all disks, hence all images will look identically.

We also implement an extensions of RAID 1:

RAID 1v (Verified Mirroring)

To create the filesystem in RAID 1v mode, `-r 1v` parameter is passed to `mkfs`. This mode has identical on-disk structure as plain RAID 1 but every read operation will compare all copies of data blocks on different drives and return the data block present on majority of the drives. if there is a tie, then data block on disk with lower index is returned. By index we mean its position during mount.

Superblock Modification

You will need to extend the superblock (`struct wfs_sb`, you must add field(s) to the end of the structure) to remember in what mode the filesystem was created and also information about disks order (needed for RAID 0). Remember that all metadata including superblock are mirrored across disks. Just for clarification, you must not change definitions of other metadata structures or other code in this file.

Mounting Restrictions

If the filesystem was created with n drives, it has to be always mounted with n drives. Otherwise you should report an error and exit with a non-zero exit code. The order of drives during mount in `wfs` command does not matter and the mount should always be succeed if correct drives are used. Note that filenames representing disks cannot be used as a filesystem identifier. Mount still has to work when disk images are renamed.

Some examples:

```
$ ./mkfs -r 1 -d disk1 -d disk2 -i 32 -b 200
$ ./wfs disk1 disk2 -f -s mnt # valid
$ echo $?
0
```

```
$ ./mkfs -r 1 -d disk1 -d disk2 -d disk3 -i 32 -b 200
$ ./wfs disk1 disk2 -f -s mnt # invalid
Error: not enough disks.
$ echo $?
-1
```

```
$ ./mkfs -r 0 -d disk1 -d disk2 -d disk3 -i 32 -b 200
$ ./wfs disk3 disk1 disk2 -f -s mnt # valid
$ echo $?
0
```

```
$ ./mkfs -d disk1 -i 32 -b 200 # invalid
Error: No raid mode specified.
$ echo $?
-1
```

Utilities

To help you run your filesystem, we provided several scripts:

- `create_disk.sh` creates a file named `disk` with size 1MB whose content is zeroed. You can use this file as on of your disk images. We may test your filesystem with images of different sizes, so please do not assume the image is always 1MB. You will need to modify this file, to create more disks, because your filesystem will always need at least 2 disks. This script is not graded.
- `umount.sh` unmounts a mount point whose path is specified in the first argument.
- `Makefile` is a template makefile used to compile your code. It will also be used for grading. Please make sure your code can be compiled using the commands in this makefile.

A typical way to compile and launch your filesystem with 2 disks in RAID 1 (`-r 1`) is:

```
$ make
$ # somehow create disk1.img and disk2.img files
$ ./mkfs -r 1 -d disk1 -d disk2 -i 32 -b 200
$ mkdir mnt
$ ./wfs disk1 disk2 -f -s mnt
```

You should be able to interact with your filesystem once you mount it. Open new terminal and try following commands:

```
$ stat mnt
$ mkdir mnt/a
$ stat mnt/a
$ mkdir mnt/a/b
$ ls mnt
$ echo asdf > mnt/x
$ cat mnt/x
```

Error handling

If any of the following issues occur during the execution of a registered function, it's essential to return the respective error code. These error code macros are accessible by including header `<errno.h>`.

- File/directory does not exist while trying to read/write a file/directory\
  return `-ENOENT`
- A file or directory with the same name already exists while trying to create a file/directory\
  return `-EEXIST`
- There is insufficient disk space while trying to create or write to a file\
  return `-ENOSPC`

Debugging

Inspect superblock

You can see the exact contents present in the disk image, before mounting. For a new disk image, the disk should contain the superblock after running `mkfs`.

```
$ ./create_disk.sh
$ xxd -e -g 4 disk | less
$ ./mkfs disk
$ xxd -e -g 4 disk | less
```

Printing

Your filesystem will print to `stdout` if it is running in the foreground (`-f`). We recommend adding a print statement to the beginning of each callback function so that you know which functions are being called (e.g. watch how many times `getattr` is called). You may, of course, add more detailed print statements to debug further.

Debugger

To run the filesystem in `gdb`, use `gdb --args ./wfs disk.img -f -s mnt`, and type `run` once inside `gdb`.

Important Notes

1. Manually inspecting your filesystem (see Debugging section above), before running the tests, is highly encouraged. You should also experiment with your filesystem using simple utilities such as `mkdir`, `ls`, `touch`, `echo`, `rm`, etc.
2. Directories will not use the indirect block. That is, directories entries will be limited to the number that can fit in the direct data blocks.
3. Directories may contain blank entries up to the size as marked in the directory inode. That is, a directory with size 512 bytes will use one data block, both of the following entry layouts would be legal -- 15 blank entries followed by a valid directory entry, or a valid directory entry followed by 15 blank entries. You should free all directory data blocks with `rmdir`, but you do not need to free directory data blocks when unlinking files in a directory.
4. A valid file/directory name consists of letters (both uppercase and lowercase), numbers, and underscores (_). Path names are always separated by forward-slash. You do not need to worry about escape sequences for other characters.
5. The maximum file name length is 28
6. Please make sure your code can be compiled using the commands in the provided Makefile.
7. We recommend using `mmap` to map the entire disk image into memory when the filesystem is mounted. Mapping the image into memory simplifies reading and writing the on-disk structures a great deal, compared to `read` and `write` system calls.
8. Think carefully about the interfaces you will need to build to manipulate on-disk data structures. For example, you might have an `allocate_inode()` function which allocates an inode using the bitmap and returns a pointer to a new inode, or returns an error if there are no more inodes available in the system.
9. You must use the superblock and inode structs as they are defined in the header file, but the actual allocation and free strategies are up to you. Our tests will evaluate whether or not you have the correct number of blocks allocated, but we do not assume they are in a particular location on-disk.

Further Reading and References

- https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/homework/fuse/fuse_doc.html
- https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/html/index.html
- http://libfuse.github.io/doxygen/index.html
- https://github.com/fuse4x/fuse/tree/master/example
- `/usr/include/asm-generic/errno-base.h`
- https://pages.cs.wisc.edu/~remzi/OSTEP/file-raid.pdf