Homework 6

W4118 Fall 2023

UPDATED: Saturday 12/2/2023 at 7:58pm EST
DUE: Monday 12/11/2023 at 11:59pm EST

All homework submissions are to be made via **Git**. You must submit a detailed list of references as part of your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this **file** in the top-level directory of your homework submission in the `main` branch of your team repo. **Be aware that commits pushed after the deadline will not be considered.** Refer to the homework policy section on the **class web site** for further details.

Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository for your group has been setup already on Github. It can be cloned using:

```
git clone git@github.com:W4118/f23-hmwk6-teamN.git
```

(Replace `teamN` with the name of your team, e.g. `team0`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as **git-pull**, **git-merge**, **git-fetch**.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with the checkpatch.pl (or provided run_checkpatch.sh) script on the default path in the provided VM. Errors from the script in your submission will cause a deduction of points.

For students on Arm Mac computers (e.g. with M1 or M2 CPU): if you want your submission to be built/tested for Arm, you must create and submit a file called `.armpls` in the top-level directory of your repo; feel free to use the following one-liner:

```
cd "$(git rev-parse --show-toplevel)" && touch .armpls && git add .armpls
&& git commit -m "Arm pls"
```

You should do this first so that this file is present in any code you submit for grading.

For all programming problems you should submit your source code, a Makefile, and a single README file documenting your files and code for each part. Please do NOT submit kernel images. The README should explain any way in which your solution differs from what was assigned, and any assumptions you made. You are welcome to include a test run in your README showing how your system call works. **It should also state explicitly how each group**

member contributed to the submission and how many hours each member spent on the homework.

For this assignment, the README should explicitly state which parts of the file system assignment were completed successfully and which parts are not functional. It should also state explicitly how each group member contributed to the submission.

Finally, since this is the last assignment of the semester, EACH group member should indicate at the top of the README five important pieces of information: (1) the number of hours spent on this assignment, (2) a rank ordering of the difficulty of the homework assignments, (3) a rank ordering of how much you learned on each homework assignment, (4) the extent to which you agree that this assignment has significantly improved your understanding of file systems (1=strongly disagree, 2=disagree, 3=neutral, 4=agree, 5=strongly agree), and (5) any comments about your how your educational experience doing this assignment. The format should be EXACTLY as shown below for each group member:

```
abc123: 15hrs
difficulty: hmwk1 < hmwk2 < hmwk4 < hmwk6 < hmwk5 < hmwk
learned: hmwk1 < hmwk2 < hmwk4 < hmwk3 < hmwk5 < hmwk6
rating: 5
comments: any comments here
```

would indicate that student with UNI abc123 spent 15 hrs on this assignment, hmwk1 was the easiest and hmwk3 was the hardest, learned the least on hmwk1 and most on hmwk6, and strongly agree that this assignment significantly improved abc123's understanding of file systems. The README should be placed in the top level directory of the `main` branch of your team repo. **5% of the grading points for this assignment will be allocated to grading your README.**

In this assignment, you will write your own disk-based file system, EZFS. You will learn how to use a loop device to turn a regular file into a block storage device, then format that device into an EZFS file system. Then you will use EZFS to access the file system. EZFS will be built as a kernel module that you can load into the **stock Debian 12.1 kernel** in your VM. You do not need to use the 4118 kernel you built for previous homework assignments and there is no need to build the entire Linux kernel tree for this assignment.

1. **Formatting and mounting a disk.** A loop device is a pseudo-device that makes a file accessible as a block device. Files of this kind are often used for CD ISO images. Mounting a file containing a file system via such a loop mount makes the files within that file system accessible. You will do this with EZFS, but to first gain some experience with a loop device, the following gives you a sample session for creating a loop device and building and mounting an ext2 file system on it. This session starts from the home directory of a user zzj. You should read man pages and search the Internet so you can understand what is going on at each step.

```
$ sudo su
```

```
# dd if=/dev/zero of=./ext2.img bs=1024 count=104
104+0 records in
104+0 records out
106496 bytes (106 kB, 104 KiB) copied, 0.000600037 s, 1
# modprobe loop
# losetup --find --show ext2.img
/dev/loop0
# mkfs -t ext2 /dev/loop0
mke2fs 1.47.0 (5-Feb-2023)
Creating filesystem with 104 1k blocks and 16 inodes

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting informati

# mkdir mnt
# mount /dev/loop0 ./mnt
# df -hT
Filesystem     Type      Size  Used Avail Use% Mounted
...
/dev/loop0     ext2       93K   14K   74K  16% /root/mn
# cd mnt
# ls -al
total 17
drwxr-xr-x  3 root root  1024 Apr 21 02:22 .
drwxr-xr-x 37 zzj zzj  4096 Apr 21 02:22 ..
drwx------  2 root root 12288 Apr 21 02:22 lost+found
# mkdir sub2
# ls -al
total 18
drwxr-xr-x  4 root root  1024 Apr 21 02:23 .
drwxr-xr-x 37 zzj zzj  4096 Apr 21 02:22 ..
drwx------  2 root root 12288 Apr 21 02:22 lost+found
drwxr-xr-x  2 root root  1024 Apr 21 02:23 sub2
# cd sub2
# ls -al
total 2
drwxr-xr-x 2 root root 1024 Apr 21 02:23 .
drwxr-xr-x 4 root root 1024 Apr 21 02:23 ..
# mkdir sub2.1
# ls -al
total 3
drwxr-xr-x 3 root root 1024 Apr 21 02:24 .
drwxr-xr-x 4 root root 1024 Apr 21 02:23 ..
drwxr-xr-x 2 root root 1024 Apr 21 02:24 sub2.1
# touch file2.1
# ls -al
total 3
drwxr-xr-x 3 root root 1024 Apr 21 02:24 .
drwxr-xr-x 4 root root 1024 Apr 21 02:23 ..
-rw-r--r-- 1 root root    0 Apr 21 02:24 file2.1
drwxr-xr-x 2 root root 1024 Apr 21 02:24 sub2.1
# cd ../../
# umount mnt/
# losetup --find
/dev/loop1
# losetup --detach /dev/loop0
# losetup --find
/dev/loop0
# ls -al mnt/
total 8
drwxr-xr-x  2 root root 4096 Apr 21 02:22 .
drwxr-xr-x 37 zzj zzj 4096 Apr 21 02:22 ..
```

In the sample session shown above, files and directories are created. Make sure you see the number of links each file or directory has, and make sure you understand why.

2. **Exploring EZFS.** Now that you understand how to use a loop device, mount a loop device and format it as EZFS. To do the latter, we have provided you with source code for an EZFS formatting program. First create a disk image and assign it to a loop device:

```
$ dd bs=4096 count=1000 if=/dev/zero of=~/ez_disk.img
# losetup --find --show ~/ez_disk.img
```

This will create the file `ez_disk.img` and bind it to an available loop device, probably `/dev/loop0`. Now, `/dev/loop0` can be used as if it were a physical disk, and the data backing it will be stored in `ez_disk.img`.

Now format the disk as EZFS. The skeleton code for a formatting utility program is in `format_disk_as_ezfs.c`. Compile it, then run it:

```
# ./format_disk_as_ezfs /dev/loop0 1000
```

We have provided you with reference kernel modules that implement EZFS which are designed to work with your stock Debian 12.1 kernel (`6.1.0-11-amd64` and `6.1.0-11-arm64`). x86 and arm kernel modules are in `ref/ez-x86.ko` and `ref/ez-arm.ko`, respectively. You should familiarize yourself with writing and using **Linux kernel modules**. You can use the reference kernel module to explore your newly created EZFS by mounting the disk and loading the kernel module:

```
# mkdir /mnt/ez
# insmod ez-ARCH.ko
# mount -t ezfs /dev/loop0 /mnt/ez
```

where ARCH is either x86 or arm. Now you can create some new files, edit `hello.txt`, etc. If your kernel name is slightly different (e.g. `6.1.0-12-amd64`), you may get a versioning error when you try to load the kernel module. In that case, you can try forcibly inserting the module with `insmod -f`.

3. **Changing the formatting program.** The formatting utility creates the new file system's root directory and places `hello.txt` in that directory. You can think of the formatting utility as statically creating the file system on a disk. You will first create directories and files by modifying the formatting utility, as this will help you later to figure out what EZFS must do to perform these file system operations. Start by reviewing the EZFS **specification**, then read the formatting utility source code `format_disk_as_ezfs.c`. Make sure you understand the on-disk format and what each line contributes toward creating the file system. **A key simplifying concept in EZFS is how file data is stored, specifically directories are limited to one block in size and regular files may use multiple blocks but the blocks used for storing the data for a given file are managed using a simple index allocation scheme with a**

single direct block and a single indirect block. Since EZFS blocks are 4KB, this means that your maximum file size is slightly more than 2MB.

The entries in your indirect block should simply be `uint64_t` stored block numbers. A 4KB indirect block should therefore contain 512 `uint64_t` entries. Entries that are not in use should be zeroed. You should also use zero to denote entries that are not in use for your inodes. In other words, if either the direct or indirect block entry in the inode are not used, the respective entry should be zeroed.

Now extend the formatting utility program to create a subdirectory called `subdir`. The directory should contain `names.txt` that lists the names of your team members, `big_img.jpeg`, and `big_txt.txt`. The latter two files are in your repo subdirectory `big_files`. `names.txt` should be stored in disk block number 5, `big_img.jpeg` in disk block numbers 6-13, and `big_txt.txt` in disk block numbers 14-15. Be sure to set the directories' link counts correctly. Any required indirect blocks can be placed in any free disk block and are not included in the disk block numbers listed above. For example, `big_txt.txt` will require an additional disk block for its indirect block in addition to disk block numbers 14-15 for its data.

Create and format a new disk using your modified program. Use the reference EZFS kernel module provided to verify that the new files and directory were created correctly. You can use the `stat` command to see the size, inode number, and other properties. Note that the primary purpose of the reference EZFS kernel module is to provide a way to check that your formatting utility program operates correctly. It does not necessarily implement all of the functionality that you will provide in your own EZFS implementation.

4. **Initializing and mounting the file system.** Now that you understand how to manually add files to your file system via your formatting utility, you will now write a file system to allow you to use standard file commands mount the file system, list directories, read files, modify existing files, create new files, delete files, and even create and remove directories. The rest of this assignment is structured to guide you toward incrementally implementing your file system functionality, which you will do by implementing `ez_ops.h` and `myez.c` in your team repo. In some cases, you may find that what you implemented is correct enough to get some piece of functionality working, but may not be completely correct such that some later functionality that depends on it ends up not working. Keep that in mind during your debugging. Here are some resources that might be useful, though keep in mind that some of the information contained therein may be out of date:

   - LKD chapter 13
   - LKD chapter 14: pages 289 - 294
   - LKD chapter 16: pages 326 - 331
   - **Page cache overview slides**
   - **Linux Kernel Internals chapter 3 (Virtual Filesystem)**
   - **Documentation/filesystems/vfs.rst**
   - **Linux VFS tutorial at lwn.net**

Note that the VFS has evolved over the years and some functions exist primarily for backwards compatibility with older file system implementations. **In your implementation, you should make sure to use the newer VFS interface functions discussed in class whenever possible.** As always, the best source of correct information is the source code, especially other file system implementations, some of which were described in class, including **ramfs**. Other file system implementations are also good references to see what functions you have to implement and which ones you do not have to implement, or can implement by leveraging functions already provided by the VFS.

This part of the assignment focuses on writing the code that initializes the file system and enables mounting disks. Create the basic functionality for your file system to work as a kernel module so that it can be loaded and unloaded from the kernel. Then make the `mount` and `umount` commands work cleanly. We won't be reading any files or directories at this time.

The name attribute of your `struct file_system_type` MUST BE **myezfs**. *Failure to provide the correct naming of your file system will result in an automatic zero on your grade*.

Some Hints:

- Use `sb_set_blocksize()` to ensure that the block layer reads blocks of the correct size.
- Read the EZFS superblock and inodes. Assign them to an instance of `struct ezfs_sb_buffer_heads`. Store this struct in the `s_fs_info` member of the VFS superblock. This way, we can always find the EZFS superblock and inodes by following the trail of pointers from the VFS superblock. **EZFS fill_super** shows the relationship between these structs after the the superblock is read from disk and its in-memory representation is initialized.
- You will have to fill out some additional members of the VFS superblock structure, such as the magic number and pointer to the ops struct.
- Use `iget_locked()` to create a new VFS inode for the root directory. Read the kernel source to learn what this function does for you and get some hints on how you're supposed to use it. The only metadata you need to set is the mode. Make the root directory `drwxrwxrwx` for now.
- After creating an inode for the root directory, you need to create a dentry associated with it. Make the VFS superblock point to the dentry.
- Make sure to handle errors by returning an appropriate error code. For example, what if someone asks you to mount a filesystem that isn't EZFS?
- Remember to take care of any buffer_heads and dynamically allocated pointers.

5. **Listing the contents of the root directory.** In the previous part, you may have created a VFS inode without associating it with the corresponding EZFS inode from disk. Although this may be sufficient for `mount` to work, it will not be enough to properly list the contents of the root directory. You need to update your code to associate the root VFS inode with the root EZFS inode. Use the `i_private` member of the VFS inode to store a pointer to the EZFS

inode. All of the EZFS inodes live in the inode store that we read from disk in the previous section. Consult the diagram in the EZFS Specification section.

Now you can add support for listing the root directory. You should be able to run `ls` and `ls -a`. Note that we do not support listing the contents of a subdirectory yet. Here's sample session:

```
# ls /mnt/ez
hello.txt  subdir
# ls -a /mnt/ez
.  ..  hello.txt  subdir
# ls /mnt/ez/subdir
ls: cannot access '/mnt/ez/subdir': No such file or dire
```

The VFS framework will call the `iterate_shared` member of the `struct file_operations`. Inside your `iterate_shared` implementation, use `dir_emit()` to provide VFS with the contents of the requested directory. VFS will continue to call `iterate_shared` until your implementation returns without calling `dir_emit()`. Make sure you implement `iterate_shared`, not `iterate`, as the latter is an older interface. For now, you can pass in `DT_UNKNOWN` as the `type` argument for `dir_emit()`. We will revisit this in the next part. You can use the `ctx->pos` variable as a cursor to the directory entry that you are about to emit. Note that iterating through a directory using `dir_emit()` will list each directory entry contained in the directory, but what should be done to cause the `.` and `..` to appear in the listing? Some file systems accomplish this by actually storing separate entries for `.` and `..` so that they will appear just like any other entry, but other file systems do not, such as the proc file system. Look at how the proc file system achieves this **behavior**, and use a similar approach for your EZFS.

The following is an excerpt from the output of `strace ls /usr/bin > /dev/null`:

```
[...]
openat(AT_FDCWD, "/usr/bin", O_RDONLY|O_NONBLOCK|...) =
[...]
getdents64(3, /* 1003 entries */, 32768) = 32744
[...]
getdents64(3, /* 270 entries */, 32768) = 8888
[...]
getdents64(3, /* 0 entries */, 32768)   = 0
close(3)                                = 0
```

The `ls` program first opens the `/usr/bin` directory file. Then, it calls `getdents64()` three times to retrieve the list of 1,273 files in `/usr/bin`. Finally, `ls` closes the directory file. Each call to `getdents64()` will result in one call to `iterate_dir()`, which in turn will call your `iterate_shared` implementation. Consequently, your `iterate_shared` implementation should call `dir_emit()` until the given buffer is full.

Running `ls -l` might print error messages because the `ls` program is unable to `stat` the files. This is the expected behavior for this part.

6. **Accessing subdirectories.** Add support for looking up filepaths. You should be able to `cd` into directories and `ls` the contents of directories that aren't the root. As a side effect, the `-l` flag and `stat` command should work on both files and directories now. Here's a sample session:

```
# ls /mnt/ez/subdir
    names.txt
# cd /mnt/ez/subdir
# stat names.txt
  File: names.txt
  Size: 0           Blocks: 0      IO Block: 4096   regu
Device: 700h/1792d  Inode: 4       Links: 1
Access: (0000/----------)  Uid: (0 /    root)   Gid: (0
Access: 2017-03-30 02:42:27.629345430 -0400
Modify: 2017-03-30 02:42:27.629345430 -0400
Change: 2017-03-30 02:42:27.629345430 -0400
  Birth: -
# stat does_not_exist.txt
stat: cannot stat 'does_not_exist.txt': No such file or
# ls -l ..
total 0
---------- 1 root root 0 Apr  3 23:31 hello.txt
d--------- 1 root root 0 Dec 31  1969 subdir
```

   VFS does most of the heavy lifting when looking up a filepath. To avoid repeated work when looking up similar paths, the kernel maintains a cache called the dentry cache. Learn how the dentry cache works by reading the materials given earlier. A given path is split up into parts and each part is looked up in the dentry cache. If a part isn't in the dentry cache, the VFS will call the file system-specific `lookup` function of `inode_operations` to ask the file system to add it. For example, given a filepath such as `/a/b/c/d/e/f.txt`, once the kernel knows the inode of `c`, it will ask for the inode associated with the name `d` in the directory `c`. If there is no matching dentry in the cache, the `lookup` function will be called to retrieve the inode for `d` from the filesystem. Before you add things to the dentry cache, you are responsible for determining whether the given parent directory contains an entry with the given name.

   Make sure your code returns correct metadata for all files and directories. These include size, link count, timestamps, permissions, owner, and group.

   - Test by using `ls -l` and `stat`.
   - You should also pass the correct type to `dir_emit()` in `ezfs_iterate()`. Check out this **StackOverflow post** for why it matters. Hint: you should use `S_DT()`.

7. **Reading the contents of regular files.** Add support for reading the contents of files. There are a number of ways to do this, but you should take advantage of generic functions that are already available as part of the VFS to implement `read_iter`, not `read`. For example, `generic_file_read_iter` handles complex logic to read ahead so that file blocks can be cached in memory by the time they are actually needed to avoid blocking on slow I/O devices. However, generic file system functions are unaware of file system-specific functionality for deciding what data blocks are actually associated with each file, so the job

of the file system is to provide that information through appropriate functions that will be called by the generic functions. You should read `generic_file_read_iter` to understand how it interacts with `address_space_operations` to see what functions need to be implemented. Hint: what is `read_folio` and how is it used? You may find it particularly helpful to refer to the **BFS file system**, specifically **file.c**. What is the functionality or magic of `map_bh`? Once you have read support, you should be able to do the following:

```
# cat /mnt/ez/hello.txt
Hello world!
# cat /mnt/ez/subdir/names.txt
Emma Nieh
Haruki Gonai
Zijian Zhang
# dd if=/mnt/ez/hello.txt
Hello world!
0+1 records in
0+1 records out
13 bytes copied, 4.5167e-05 s, 266 kB/s
# dd if=/mnt/ez/hello.txt bs=1 skip=6
world!
7+0 records in
7+0 records out
7 bytes copied, 5.1431e-05 s, 117 kB/s
```

If you try using other programs to read files, you may encounter some errors. For example, `vim` by default places swap files in the current directory and seeks through them upon opening a file using `llseek`. You may have noticed an error when trying to open files using `vim` because your EZFS has no support for `llseek` yet. Fix it. Hint: there's already a generic implementation in the kernel for `llseek`.

At this point, you should stress test your EZFS implementation. The rest of this assignment will be easier if you can depend on the reading functionality to report things correctly. Some of the things you should make sure work include:

- Try copying all the files out of your ez using `cp` or `rsync`.
- Extend the formatting program again to create additional files and a more complex directory structure. Be sure to include different file types and larger files. For example, add a small team photo to the `subdir` directory.
- Overwrite the disk with random garbage from `/dev/urandom` (instead of `/dev/zero`). Format it. After formatting, the random data should not affect the normal operation of the filesystem.
- Write a program that requests invalid offsets when reading files or iterating through directories.

8. **Writing to existing files.** So far, we've only been reading what's already on the filesystem. Implement functions for modifying the filesystem contents. Again, you should implement `write_iter` instead of `write`.

Read `generic_file_write_iter`, try to understand how it helps us to write iteratively, and find out how it interacts with `address_space_operations`. Do we need to worry about changing the length of the file ourselves? How about time accounting and `inode->i_blocks`? It seems that only `write_begin` and `write_end` are called in `generic_file_write_iter`. When is `writepage` called? What's the benefit of doing so? Referring to BFS's **file.c**, implement `ezfs_writepage` and `ezfs_write_begin`. We recommend you first make sure your write functionality works for a file that requires no more than one data block for its contents. Test for writing the contents of files:

```
$ cd /mnt/ez
$ echo -ne "4118" | dd of=hello.txt bs=1 seek=7 conv=no
[...]
$ cat hello.txt
Hello w4118!
$ echo "Greetings and salutations, w4118!" | dd of=hell
[...]
$ cat hello.txt
Greetings and salutations, w4118!
```

Once you have the one block case working, then you should consider what if the file requires more than one block. EZFS only supports index allocation of blocks to a file. As indicated above, the entries in your indirect block should simply be `uint64_t` stored block numbers. Indirect block entries that are not in use should be zeroed. If either the direct or indirect block entry in the inode are not used, the respective entry should also be zeroed.

You should also be able to edit files with the `nano` editor, although it will complain about `fsync()` not being implemented. Fix this problem.

Ensure that changes to the VFS inode are written back to disk. You should do this by implementing `ezfs_write_inode()`. Of course, VFS needs to be informed that the VFS inode is out of sync with the EZFS inode. Test this by unmounting and remounting. Writing to the buffer head only changes the contents in memory. It does not cause those changes to be written back to disk. Be sure to take the appropriate measures so that your modifications are written to disk.

If there is not enough space in your file system to write what you need to write, you should return an appropriate error, specifically **`ENOSPC`**. Keep in mind that there may be multiple reasons why there is not enough space.

Until you introduced writing files, you were not really modifying your file system. Now that the file system is being modified, you should take care to make sure that concurrent file operations are being handled properly, if you have not done so already. For example, if two files are being modified at the same time, you want to make sure that you do not accidentally assign the same free data block to both files, which would obviously be an error. Make sure that your EZFS operations work properly when multiple processes or threads are performing those operations at any given time. Keep in mind that buffer head operations such as `sb_bread` may block if they need to go to disk. You may find it helpful to review how synchronization is handled in **BFS**.

9. **Creating new files.** Implement creating new files. That is, user programs should be able to call `open()` with a mode that includes `O_CREAT`. **Note that an empty file should have 0 data blocks.** Here's a sample session:

```
$ cd /mnt/ez
$ ls
hello.txt  subdir
$ touch world.txt
$ ls
hello.txt  subdir  world.txt
$ stat world.txt
  File: world.txt
  Size: 0           Blocks: 0      IO Block: 4096   regu
Device: 700h/1792d  Inode: 7       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/     zzj)   Gid:
Access: 2022-11-16 16:51:03.287875291 -0500
Modify: 2022-11-16 16:51:03.287875291 -0500
Change: 2022-11-16 16:51:03.287875291 -0500
  Birth: -
$ cat > subdir/favorite_memes.txt
doge
chad
BigTime Tommie
https://youtu.be/TiC8pig6PGE  # Ctrl+D to denote EOF
$ cat subdir/favorite_memes.txt
doge
chad
BigTime Tommie
https://youtu.be/TiC8pig6PGE
```

10. **Deleting files.** While testing the previous part, you probably created lots of files that are now cluttering your disk. Let's implement a way to delete those files.

    Review how the VFS dentry and inode caches interact with each other using the resources given earlier in this assignment. Implement the `unlink` and `evict_inode` ops so that you can delete files.

    You are not required to implement directory removal in this part, that will happen in the next part. Ensure that you are reclaiming data blocks and EZFS inodes when appropriate. To test this, see if you can repeatedly create and remove files.

```
for i in {1..10}; do touch {1..14}; rm {1..14}; done
```

    In a Unix-like operating system, what is the correct behavior if one process unlinks a file while another process has the same file open? Here's an experiment you can run on ext4 or the EZFS reference implementation to find out:

    - Create a file named `foo`.
    - In terminal window A, run `tail -f foo`. This command will open foo, print out all the contents, and wait for more lines to be written.

- In terminal B, run `cat > foo`. This reads from stdin and outputs the result to `foo`.
- In terminal C, delete `foo`.
- Back in terminal B, type some text and press return.
- The text should appear in terminal A.

11. **Making and removing directories.** Implement creating new directories. That is, user programs should be able to call `mkdir()`. This should be very similar to what you did to support creating regular files. You need to make sure that you're setting a size and link count appropriate for a directory, rather than a regular file. Hint: consider the link count of the parent directory of the newly created directory as well. In this part as well as the preceding ones, you should make sure that whatever robustness tests you did earlier continue to pass.

    Implement deleting directories. User programs should be able to call `rmdir()` successfully on empty directories. This should be very similar to what you did in the previous part. Take a look at `simple_rmdir()` for some additional directory-specific steps. Note that `simple_empty()` is not sufficient to check if a directory is empty for our purposes, because the function simply checks the dentry cache to see if a directory has children. Can you think of a case where this would lead to incorrect behavior?

    Here's a sample session:

```
$ ls -alF
total 16
drwxrwxrwx 3 zzj  zzj  4096 Nov 16 17:22 ./
drwxr-xr-x 3 root root 4096 Nov 16 17:23 ../
-rw-rw-rw- 1 zzj  zzj    13 Nov 16 17:22 hello.txt
drwxrwxrwx 2 zzj  zzj  4096 Nov 16 17:22 subdir/
$ mkdir bigtime
$ ls -alF
total 20
drwxrwxrwx 4 zzj  zzj  4096 Nov 16 17:23 ./
drwxr-xr-x 3 root root 4096 Nov 16 17:23 ../
drwxr-xr-x 2 zzj  zzj  4096 Nov 16 17:23 bigtime/
-rw-rw-rw- 1 zzj  zzj    13 Nov 16 17:22 hello.txt
drwxrwxrwx 2 zzj  zzj  4096 Nov 16 17:22 subdir/
$ cd bigtime
$ touch tommie
$ ls -alF
total 8
drwxr-xr-x 2 zzj zzj 4096 Nov 16 17:24 ./
drwxrwxrwx 4 zzj zzj 4096 Nov 16 17:23 ../
-rw-r--r-- 1 zzj zzj    0 Nov 16 17:24 tommie
$ cd ..
$ rmdir bigtime
rmdir: failed to remove 'bigtime': Directory not empty
$ ls -alF
total 20
drwxrwxrwx 4 zzj  zzj  4096 Nov 16 17:23 ./
drwxr-xr-x 3 root root 4096 Nov 16 17:23 ../
drwxr-xr-x 2 zzj  zzj  4096 Nov 16 17:24 bigtime/
-rw-rw-rw- 1 zzj  zzj    13 Nov 16 17:22 hello.txt
drwxrwxrwx 2 zzj  zzj  4096 Nov 16 17:22 subdir/
```

```
$ rm bigtime/tommie
$ rmdir bigtime
$ ls -alF
total 16
drwxrwxrwx 3 zzj  zzj  4096 Nov 16 17:25 ./
drwxr-xr-x 3 root root 4096 Nov 16 17:23 ../
-rw-rw-rw- 1 zzj  zzj    13 Nov 16 17:22 hello.txt
drwxrwxrwx 2 zzj  zzj  4096 Nov 16 17:22 subdir/
```

12. **Compile and run executable files.** Compiling and running executable files requires some additional functionality beyond what you have already implemented, specifically support for `mmap`. Given the approach you should have taken thus far, implementing `mmap` support should be trivial. Do it. At this point, you should now be able to compile and execute programs. This part will also double verify that you implemented the functionality of "read/write/fsync", "create/delete" correctly.

    Here's a sample session:

```
$ cd /mnt/ez
$ vim test.c
$ ls
hello.txt  subdir  test.c
$ cat test.c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
$ gcc test.c
$ ls
a.out  hello.txt  subdir  test.c
$ ./a.out
Hello, World!
```

At this point, you should make sure that whatever robustness tests you did earlier continue to pass with your completed file system, and your tests should include having multiple processes or threads perform various file system operations concurrently. In addition, you should try running various programs manipulating the files in your file system. You should also make sure you test by unmounting and remounting to make sure all your programs manipulating files work correctly with the file data actually written to disk and not just file data in the page cache. In your README, note which applications you have used, which ones worked, and which ones do not. What are some file operations supported on your default Linux file system that are not supported by EZFS? Which of these affect the functionality of the programs you ran?

Submission Checklist

Include the following in your main branch. Only include source code (ie *.c,*.h) and text files, do **not** include compiled objects.

- `.armpls` file for teams with M1/M2 CPUs

- `README` file
- `references.txt` file
- Implementation of `format_disk_as_ezfs.c`
- Implementation of `myez.c` and `ezfs_ops.h`