Linux Kernel Programming

Project 2024 – Life : Light Insertion File systEm

AUTHORS
Aymeric Agon-Rambosson, Sorbonne University
Julien Sopena, Sorbonne University

In this project, you will have to implement new features on top of an existing file system, **ouiche_fs**, in the form of a kernel module for Linux 6.5.7.

As usual, you are strongly encouraged to use a cross-reference tool to navigate the Linux kernel source code, be it integrated in your development environment or online like Bootlin's Elixir.

You are also strongly encouraged to read the documentation available in `Documentation/filesystem/vfs.rst` or online to better understand how the virtual file system (VFS) layer works and how a file system is implemented.

> ⓘ **Objectives of the project**
>
> In this project, you will implement a **fast insertion file system**, where adding new data in the middle of a file will be faster thanks to a management of partially filled blocks. Indeed, in a traditional filesystem, data blocks are filled contiguously, and only the last block can be partially filled. Therefore, when we want to add data in the middle of a file, we have to shift all the subsequent data to maintain the sequentiality of the blocks. In this project, we will allow *OuicheFS* to fill the blocks in a more flexible manner, which will enable faster data insertion.

1 Project roadmap

The project is divided into several steps, each of which corresponds to a specific feature to be implemented. You will need to complete each step in order, as each step builds on the previous one.

> ⓘ **Important**
>
> **Please read the assignment in its entirety before starting your implementation!!!**
>
> If you know everything you need to do beforehand, you will make better design decisions, and you won't have to change things back and forth often!

1.1 Mounting and testing a vanilla *OuicheFS*

This project is based on the *OuicheFS* file system that you need to download from GitHub at the following address: https://github.com/rgouicem/ouichefs. Be careful to use the branch named `v6.5.7`!

In addition to the source code, you will find documentation explaining the design of this simple file system as well as the relationship between various structures in the kernel. Don't hesitate to read the source code, it is documented, and to play around with the vanilla file system in your virtual machine to better understand how it works, its limitations, etc.

After downloading the source and building the kernel module, you will need to format a partition with *OuicheFS* and mount it. You can then test the file system by creating files, writing data, and reading it back.

1.2 Implementation of a benchmark

Before starting to modify the code of *OuicheFS*, you need to implement a benchmark to verify that the modifications you will make work correctly, as well as to measure the performance of your optimization.

To do this, you need to write a user C program that creates files of variable size and then adds data at different positions. Your program should measure the write times as well as the read times. Additionally, your test should verify that the data is written and read correctly.

1.3 Reimplementation of the read and the write functions

The first step of your project is to reimplement the read and the write functions of *OuicheFS* without changing their behavior. Currently, *OuicheFS* does not implement these functions and therefore uses the default implementation provided by the Linux kernel. Unfortunately, this implementation assumes that the data blocks are filled contiguously, which will no longer be the case at the end of the project.

To simplify this step, you will implement read and write functions that behave like the default implementation, but that will not use the page cache. To do this, you will need to use the `sb_bread` and `brelse` functions provided by the Linux kernel to read data directly from the disk. You will also need to use the `mark_buffer_dirty` and `sync_dirty_buffer` functions to write data to the disk.

The read function, whose prototype you can find in the documentation of the vfs referred to above, has the following semantic :

- You must return the amount of bytes you copied to userspace. There are some cases when that quantity is 0.
- Updating the offset given to you by the application is your responsibility.
- Note that you do not have to return all the bytes that the application asked you for. The returned quantity and the offset update must only be consistent with what you copy. An application that wishes to read until the end of the file (like `cat(1)`) will attempt the read syscall again. However, note that returning 0 can be interpreted by the application that there will be no more bytes left to read, prompting them to give up.

The write function has the following semantic :

- You must return the amount of bytes you have copied from userspace.
- You must still update the offset.
- You do not have to accept all the bytes the application sent you.
- All the updates to the inode are your responsibility.
- Note that writes done after the end of the file are valid. It is your responsibility to fill the possible blanks in the file.

You are strongly advised to read the function `ouichefs_write_begin` and `ouichefs_write_end` to see what checks and modifications are made. Your write function will be called instead of those !

Check that your implementation works correctly using the test bench you wrote in the previous step and measure the performance loss compared to the default implementation.

1.4 Modification of the data structure and implementation of an ioctl command

One of the difficulties of this project lies in the management of the data blocks. Ouichfs does not allow to store the size of the data in the block, which makes it difficult to manage partially filled blocks. To address this issue, you will need to modify the data structure used to store the data blocks.

Actually, *OuicheFS* uses an `index_block` to store at most 1024 block numbers encoded on 4 bytes each. In this project, we propose to limit the partition size to 4GB and use the 12 bits of the block number to store the effectively used size of the block.

To facilitate debugging in the project, you will implement an ioctl command that displays, for a given file descriptor:

- the number of used blocks
- the number of partially filled blocks
- the number of bytes wasted due to internal fragmentation
- the list of all used blocks with their number and effective size

1.5 Implementation of the write function

Now you can get to the heart of the matter by implementing the write function of *OuicheFS*. To do this, you will need to modify the write function that you have implemented in step 2, so that it can add data to a file without having to shift the data that follows.

Check that your implementation works correctly with the ioctl command you implemented in Section 1.4. You should observe that the number of partially filled blocks increases. Then, verify that your modification improves the performance of the write function by running the benchmark you implemented in Section 1.2.

Note that there are many possible optimizations to fragment data blocks in order to minimize the number of lost bytes. However, the main objective is to complete this project entirely. In the first step, you will implement a version of the write function that systematically fragments data blocks. Once step 5 is completed, you can try to improve performance by minimizing fragmentation during writing: filling partially filled blocks, utilizing space in adjacent blocks, etc.

1.6 Modification of the read function

At this stage, you will need to modify the read function you implemented in Section 1.3 so that it can correctly read data when the data blocks are not filled contiguously.

Verify that your implementation works correctly. You should also observe a degradation in read performance when the degree of block filling is low.

1.7 Implementation of a defragmentation function

If your optimization speeds up data writing, it will lead to a degradation in read performance. This is because the data will no longer be contiguous, and therefore more blocks will have to be read to retrieve the same amount of data. To address this issue, you will implement an ioctl command that triggers a defragmentation function that reorganizes the data to make it contiguous again.

Verify that your implementation works correctly and that it restores the read performance measured in Section 1.3.

1.8 Bonus: Use of the page cache

One limitation of this implementation is that it does not take advantage of the Linux kernel's page cache, but only the buffer cache. Therefore, if you have time and only if you have completed the previous steps, you can modify your implementation to use the page cache.

2 Bug bounty

If you find any bug in ouiche_fs, do not hesitate to report it and fix it! If you manage to fix a bug, provide a patch and an explanation in your report. **After the submission deadline**, you can submit your fix through a pull request on GitHub (in exchange for a few bonus points of course…).

3 Submission

3.1 Process

You will submit your work on the Moodle platform as a `tar.gz` archive. This archive needs to contain the modified sources of ouiche_fs with no residual files from builds (object files, binaries, etc.), as well as a short `pdf` report (2–4 pages). This report should contain explanations about your design choices (algorithms and data structures) and a status report with the following three sections:

- List of features implemented and functional
- List of features implemented but not fully functional. In this case, explain the problems, why they occur, and potential fixes if you have any in mind.
- List of features not implemented.
- Short conclusion based on your experimental results.

The project must be done in **groups of three students**. You should already have registered your groups by now.

The submission deadline is monday 27 mai at 8:00 am. You will have also to present your work, design and implementations choices, and make a short demonstration.

3.2 Grading

You will be evaluated with functional tests (does your code work?), as well as on the quality and readability of your code (don't forget to use the `scripts/checkpatch.pl` script from the kernel sources, respect the kernel coding style rules, and comment your code). Similarly for the report, please make it easy to understand and run a spell checker. The report itself will not be graded, but if it is hard to understand, it will make it harder for us to give you more points.

4 Appendix

4.1 Buffer cache (include/linux/buffer_head.h)

```
/*
 * Returns a pointer to the cached buffer of block bno on partition sb.
 * The size of the read buffer is sb->s_blocksize, and the data in the buffer is accessible
 * through the b_data member of the struct buffer_head returned.
 */
struct buffer_head *sb_bread(struct super_block *sb, sector_t bno);
```

```
/*
 * Marks the buffer pointed at by bh as dirty, so that the page cache writes it back to disk.
 */
void mark_buffer_dirty(struct buffer_head *bh);
```

```
/*
 * Forces the write back of the buffer pointed at by bh on disk.
 * Returns 0 on success.
 */
void sync_dirty_buffer(struct buffer_head *bh);
```

```
/*
 * Releases a reference to the buffer bh.
 */
void brelse(struct buffer_head *bh);
```

4.2 Inode (include/linux/fs.h)

```
/*
 * Marks the inode as dirty (after a modification).
 * When the inode will be freed, the destroy_inode() function will be called on it.
 */
void mark_inode_dirty(struct inode *inode);
```