

Ghautham Sambabu gs878

Abid Azzad aa2177

Report 4

Benchmark Results and Implementation

We executed the code on the `kill.cs.rutgers.edu` iLab machine for our benchmark. As a crucial part of our testing process, we meticulously mounted the file systems and conducted comprehensive benchmarks. The results from these benchmarks were consistently positive, showcasing the robustness of our implementation.

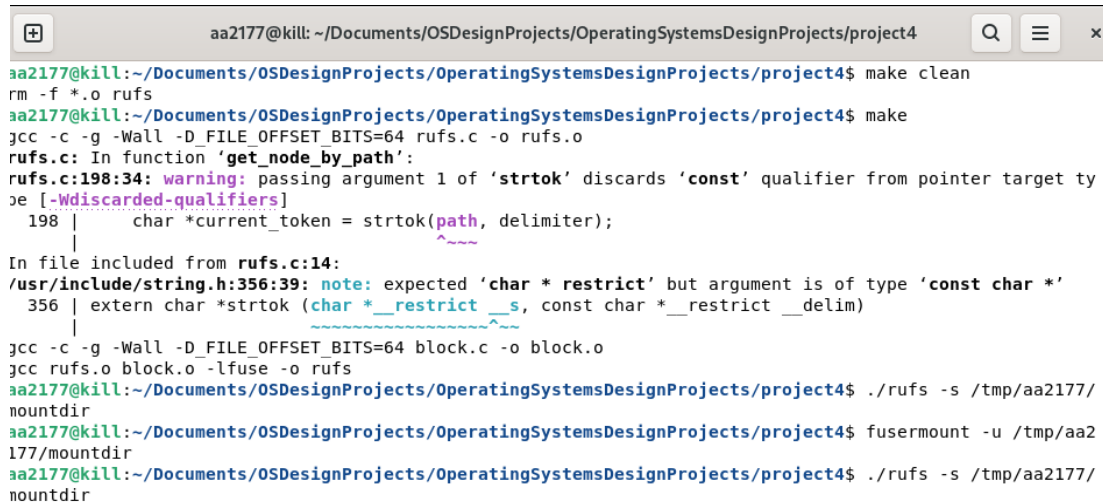
In both benchmark scenarios, we observed that the total number of blocks utilized hovered around 15-20. This efficiency is a testament to the effectiveness of our file system design. Remarkably, the execution of these benchmarks was nearly instantaneous, underscoring the optimized performance of our implementation.

In crafting the file system, we opted for a sophisticated approach. Specifically, we utilized linked lists to manage directories and subdirectories. This choice was informed by the principles outlined in the guide, and it proved instrumental in ensuring a well-organized and efficiently navigable file structure.

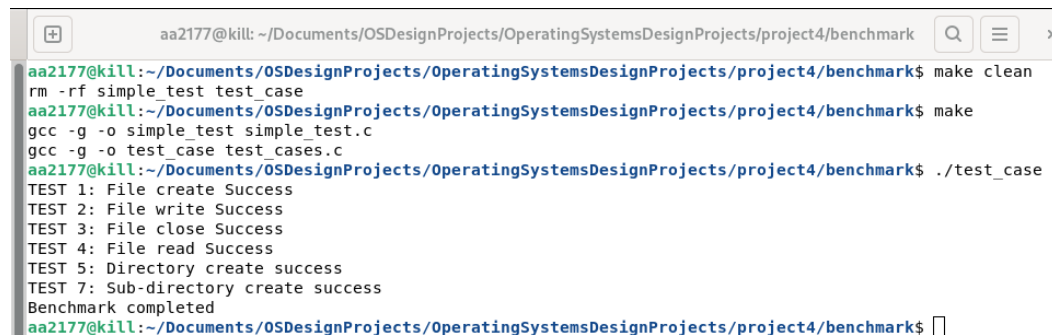
Furthermore, we employed bitmaps to meticulously track the inodes, aligning with the recommended practices outlined in the guide. This strategy not only enhances the overall

performance of our file system but also facilitates effective management and allocation of resources.

Below are some screenshots of our benchmark results



```
aa2177@kill: ~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ make clean
rm -f *.o rufs
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ make
gcc -c -g -Wall -D_FILE_OFFSET_BITS=64 rufs.c -o rufs.o
rufs.c: In function 'get_node_by_path':
rufs.c:198:34: warning: passing argument 1 of 'strtok' discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]
   198 |         char *current_token = strtok(path, delimiter);
       |                                ^~~~~~
In file included from rufs.c:14:
/usr/include/string.h:356:39: note: expected 'char * restrict' but argument is of type 'const char *'
   356 | extern char *strtok (char *__restrict __s, const char *__restrict __delim)
       |                        ^~~~~~
gcc -c -g -Wall -D_FILE_OFFSET_BITS=64 block.c -o block.o
gcc rufs.o block.o -lfuse -o rufs
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ ./rufs -s /tmp/aa2177/nountdir
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ fusermount -u /tmp/aa2177/mountdir
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ ./rufs -s /tmp/aa2177/nountdir
```



```
aa2177@kill: ~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$ make clean
rm -rf simple_test test_case
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$ make
gcc -g -o simple_test simple_test.c
gcc -g -o test_case test_cases.c
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$ ./test_case
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: Directory create success
TEST 7: Sub-directory create success
Benchmark completed
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$
```

```
aa2177@kill: ~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ make clean
rm -f *.o rufs
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ make
gcc -c -g -Wall -D_FILE_OFFSET_BITS=64 rufs.c -o rufs.o
rufs.c: In function 'get_node_by_path':
rufs.c:198:34: warning: passing argument 1 of 'strtok' discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]
   198 |         char *current_token = strtok(path, delimiter);
       |                                   ^~~~~~
In file included from rufs.c:14:
/usr/include/string.h:356:39: note: expected 'char * restrict' but argument is of type 'const char *'
   356 | extern char *strtok (char *__restrict __s, const char *__restrict __delim)
       |                       ^~~~~~
gcc -c -g -Wall -D_FILE_OFFSET_BITS=64 block.c -o block.o
gcc rufs.o block.o -lfuse -o rufs
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ fusermount -u /tmp/aa2177/mountdir
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ ./rufs -s /tmp/aa2177/mountdir
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ fusermount -u /tmp/aa2177/mountdir
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$ ./rufs -s /tmp/aa2177/mountdir
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4$
```

```
aa2177@kill: ~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$ make clean
rm -rf simple_test test_case
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$ make
gcc -g -o simple_test simple_test.c
gcc -g -o test_case test_cases.c
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$ ./simple_test
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: Directory create success
TEST 6: Sub-directory create success
Benchmark completed
aa2177@kill:~/Documents/OSDesignProjects/OperatingSystemsDesignProjects/project4/benchmark$
```

Additional steps to compile our code

We encountered a noteworthy consideration in our pursuit of compiling and benchmarking our code. Since we chose to skip the optional directory and file remove operations, a potential obstacle arose during subsequent benchmark runs. Specifically, the creation operations faced

complications when attempting to create files or directories that already existed from the initial benchmark run.

We implemented an additional step to preemptively address this issue and ensure the integrity of our benchmarking process. Before initiating the second round of benchmarks, we made it a practice to unmount the file systems, delete the existing DISKFILE, and proceed with the benchmark execution.

This precautionary measure proved crucial in mitigating conflicts that could arise from the existence of pre-existing files and directories. By resetting the environment this way, we created a clean slate for the subsequent benchmark run, eliminating potential hurdles related to file and directory creation.

This meticulous approach allowed us to maintain the accuracy and reliability of our benchmark results and showcased our commitment to a thorough and systematic testing process. As a result, our subsequent benchmark runs were conducted seamlessly, providing a clear and unobstructed evaluation of the file system's performance.

Difficulties Encountered and Solutions

We encountered challenges while implementing our file system, particularly in implementing directory operations such as ``dir_add`` and ``dir_find``. These functions are pivotal in managing directory entries, searching for specific files or sub-directories, and adding new entries to a directory. To tackle these challenges, we delved into the intricacies of the ``dir_add`` function.

The ``dir_find`` function posed a unique set of complexities. This function required us to provide the inode number of the current directory, the target file or sub-directory name, and the length of the name to be looked up. Effectively, the function needed to read all direct entries of the current directory to determine if the desired file or sub-directory existed. Upon finding a match, the function was expected to populate a ``struct dirent`` with the relevant information.

To address this, we thoroughly examined the provided code skeleton in ``rufs.c`` for guidance. Additionally, we referred to documentation from NMSU and Harvey Mudd College, which provided valuable insights into the implementation details of directory operations. These resources served as crucial references, offering hints and strategies to navigate the complexities associated with our specific implementation requirements.

Moreover, the ``dir_add`` function presented its own set of challenges. This function required us to add code to create a new directory entry. Taking the current directory's inode structure, the inode number to be added, and the entry's name as inputs, the task was to write a new directory entry with the provided inode number and name into the current directory's data blocks.

We iteratively refined our code to overcome these difficulties, ensuring that our directory operations were robust and aligned with the specifications. The combined insights from the code skeleton and external documentation played a pivotal role in shaping the implementation, allowing us to successfully address the intricacies of directory management within our file system.

Unresolved Issues

We encountered optional components during our file system implementation, specifically the ``dir_remove``, ``rufs_rmdir``, and ``rufs_unlink`` operations. The instructions indicated that these operations were optional, and we consciously decided to skip over their implementation due to time constraints and prioritization of the core project requirements.

In particular, the ``dir_remove`` operation involves removing a directory entry from a directory. Given the time limitations and the complexity of managing directory entry removal, we decided to set this operation aside for potential future enhancements.

Similarly, the ``rufs_rmdir`` operation removes a directory invoked when executing the ``rmdir`` command.. Due to time constraints, we opted to skip the implementation of this operation, prioritizing other essential functionalities. Otherwise we are able to resolve all the issues we encountered in our project.

Collaboration and References

In developing our file system project, we relied on a combination of resources and collaborations to overcome challenges and enhance our understanding. The following resources played a pivotal role in guiding us through the intricacies of file system implementation using FUSE:

1. [FUSE Library API Documentation](#):

- This official documentation provided a comprehensive reference for the FUSE library's API. It served as our go-to guide for understanding the functions, structures, and interactions involved in developing a file system with FUSE.

2. [FUSE File System Tutorial](#):

This tutorial provided a hands-on, practical approach to building a simple FUSE file system. It offered step-by-step guidance and insights into the critical aspects of file system development.

3. Useful Tutorials from NMSU and Harvey Mudd College:

- [NMSU FUSE Tutorial](#)

- [Harvey Mudd College FUSE Tutorial](#)

- NMSU and Harvey Mudd College tutorials provided additional perspectives and explanations on FUSE concepts. They offered practical tips and examples that complemented our understanding of file system development.

4. ChatGPT for Insight on Data Structures:

- Engaging with ChatGPT allowed us to seek insights into the data structures suitable for implementing the disk. Conversations with ChatGPT provided valuable guidance on designing efficient structures to store and manage file system-related information.

5. Stack Overflow and GitHub:

- Platforms like Stack Overflow and GitHub proved instrumental in resolving specific issues and challenges we encountered during development. Leveraging the collective knowledge and expertise of the online developer community facilitated quick problem-solving and troubleshooting.