Project report

Authors: David Ginten (422984), Christian Wichmann (366924)

Info: The test breaks on the remote machine in the fio test. There's no error message/hint other than fio: open state file: File name too long. As other folks also have this problem we think its the file name length causing this problem.

1. Design choices (algorithms and data structures)

Mounting:

When we mount a filesystem partition, we also create a sysfs directory for that partition, as we need multiple partitions at the same time. After a successful mount, we extract the device name. We only want the name of the partition, e.g. "/dev/vda" → "vda". We then copy the string to a 32-byte buffer for Kernel string size conventions (We only copy the first 20 Bytes, but this is safe and flexible. With that, it also stays NULL-terminated for sure). We then create the sysfs for this specific partition with the corresponding superblock.

Sysfs:

In general, we have 10 attributes for all the relevant data. We add all the show functions with __ATTR_RO(). We also create an attribute_group for all the attributes. Every attribute (i.e. each file, for example, efficiency) has a show function where it collects its superblock (depends on the partition), then it collects the stats and prints its relevant data. As we need to handle multiple partitions at the same time, we created a hash table (for fast lookup) that maintains the mounted filesystems (the corresponding superblocks).

- ouichefs_sysfs_remove_partition() removes the partition-specific sysfs directory on filesystem unmount (i.e. it removes sysfs entries for ONE specific mounted partition and only removes its superblock from the hash table)
- ouichefs_sysfs_exit() cleans up **all** sysfs entries when the kernel module is unloaded.
- ouichefs_sysfs_init(): We use /sys/fs as the parent directory as our kobj represents a filesystem.

How does sysfs get the stats?

When we collect the stats, we basically call two functions to get the data that is not directly stored in the sb info. In ouichefs_count_sliced_stats(), we walk through the list of partially filled sliced blocks and count the sliced_blocks and the free_slices. Further, we need to count fully occupied sliced blocks. For that, we walk through all data blocks, check if they are allocated with the sbi→bfree_bitmap and if it is a sliced block. If so, check that they don't have free slices anymore. Then we can count them to the total_sliced_blocks.

In ouichefs_count_file_stats(), we collect stats about (small) files. To do that, we scan all inodes and first check if they are deleted/free or uninitialized. Then we check if it's a regular file. We also check if the file is a small file. With those checks, we can collect the data.

We maintain a hash table with 16 buckets (i.e. an array with 16 entries, each entry is a linked list. We probably won't have more than 16 live partitions). Each entry is a struct that basically holds the super block. To get the stats of a specific partition, we can now quickly get the super block of a corresponding kobject.

The sliced storage:

In **ouichefs_sliced.h**, we first define some macros for some constants. We also define a Magic number for sliced blocks to better recognize those blocks throughout the whole file system implementation.

To accommodate the metadata in the first slice of a sliced block, we define a sliced_block_metadata struct. Then we have some small helper functions which don't rely on any state of the file system and hence can be inlined.

sliced.c: This file accommodates all functions needed to create, modify and free sliced blocks.

1. If a block should be a sliced one, you need to initialize it first to set the magic number for sliced blocks and mark all data slices as free.
2. As we want to store files smaller than a block in slices, we need to allocate those slices for that file. For files larger than 128 Bytes, we need to allocate multiple, contiguous slices. To find such a memory chunk, we look at existing partially filled blocks or create a new sliced block.
3. If slices are freed, we need to modify the bitmap accordingly and potentially put the block back to the list of partially filled blocks (if it was full before) or put the block back to the general block pool (if it is empty now).
4. To read data from multiple contiguous slices, we first need to determine the amount of data that we want to read and then we read by copying the data to the given buffer. The write functionality works similarly.
5. If we need to relocate slices, e.g. the file size has increased, we potentially need to find a new place to fit the new amount of data.

ioctl:

We have a user space file that triggers the ioctl system call. Given a file, the ioctl handler in kernel space basically reads the whole block where the file is in (Must be a sliced block). We save the data in a two-dimensional array, and copy this data structure to user space. The user space file then prints the data. 32 slices (32 rows) each with 128 characters (128 bytes). The 32-bit bitmap and the reference to the next partially filled block (if existing) are also printed.

Inode:

Differentiate between directories and large and small files in the inode operations. Also, sync manually.

1. When creating new inodes, consider that we don't want to create an index block directly for files, as the file might use the sliced storage and with that, it doesn't need/have an index block anymore. Also, we don't allocate blocks before writing, as we don't know the size before. This pattern is also followed in the inode create operation.
2. When unlinking an inode, we need to handle small files using slices and large files using index blocks. Large files clean all the blocks they pointed to (and scrub the index block) and small blocks call the free_slices function.
3. As mark_inode_dirty() schedules the write to disk of inodes later, we need to manually write the inode to disk. This is necessary as sysfs might read data immediately after the unlink inode operation.

Open/Read/Write file operations:

Complete write functionality for a file that can be opened with truncate and append mode.
First, for small files, we cannot use the page cache functionality anymore, but rather the slices.
Hence the checks for the address space operations.

The steps to write to a file are as follows:

1. When opening a file, we can now either append or truncate. When truncating the file, we first free the content.
2. In the write function, we need to check how many Bytes we want to write. Based on that and the current file, we need to decide if we have a file that uses slices and will keep doing so, or if this file needs to be converted to a larger file (i.e. it exceeds 31*128 Bytes). Also, once a file is large, it will stay large even if the size drops below the threshold.
3. Depending on the particular case, we either write small files (using the slice functionality), transform a small file to a large file, or write a large file.

The Read functionality:

For the read functionality, we differentiate between small files using the slice storage and large files using the traditional blocks. When reading small files, we can use the read_slice functionality we have implemented before. When reading from large files, we directly access the block data where we want to read from (given by *pos). We need to make sure that we can read over multiple pages and that we cannot read from two blocks in one go, as the blocks are potentially not contiguous in memory. We also need to make sure that we don't read beyond the file's size.

2. List of features implemented and functional

- Sysfs: To check the state of the file system. You can check all the required values in the corresponding files.

- ioctl command: From user space, you can call the ioctl syscall with a file and then the block, where the file is stored, is printed with its 32 slices (Block needs to be sliced). The bitmap and the next block reference are also printed.

- You can write small files that will take the new sliced storage (for files with the following size range: 0 < size(file) < 4096 -128 bytes). That means files using multiple slices are allowed.

- You can write large files using the traditional storage and if a small file exceeds the upper size bound, then it will convert to a large file (i.e., using the traditional storage).

- You can read small and large files

- You can delete small and large files. With that, it frees the slices and blocks, respectively.

- Small and large files can exist at the same time.

3. List of features not implemented

- If large files shrink under the upper bound for small files, they do not convert back to small files (i.e. they will keep using the traditional storage).

- No defragmentation algorithm

4. Conclusion and experimental results

We did the following benchmarks:

- Write multiple files of different sizes (tiny, small, medium, large, mixed (tiny + small))
- Read files (tiny, small, medium, large)
- Efficiency: The ratio of the total size of the data in the file system to the total size of the used data blocks (in %)

Writes	Original	After 1.2	After 1.9	After 1.10
80 files á 64 Byte	0.006036 sec.	0.015440 sec	0.006770 sec	0.016374 sec
20 files á 800 Byte	0.004918 sec.	0.005316 sec	0.004027 sec	0.006464 sec
8 files á 8192 Byte	0.001660 sec.	0.003249 sec	0.002448 sec	0.004311 sec
2 files á 1 MB	0,006493 sec.	0.066389 sec	0.068967 sec	0.051293 sec
20 x 64 & 20 x 800	0.001094 sec	0.003758 sec	0.003164 sec	0.002762 sec

Reads	Original	After 1.2	After 1.9	After 1.10
80 files á 64 Byte	0.000216 sec	0.000394 sec	0.000233 sec	0.000264 sec
20 files á 800 Byte	0.000047 sec	0.000066 sec	0.000085 sec	0.000052 sec
8 files á 8192 Byte	0.000030 sec	0.000035 sec	0.000046 sec	0.000038 sec
2 files á 1 MB	0.000254 sec	0.000358 sec	0.000521 sec	0.000360 sec

Efficiency	Original	After 1.2	After 1.9	After 1.10
80 files á 64 Byte	0,78%	Same as orig.	31 %	31 %
20 files á 800 Byte	9,76%	Same as orig.	9 %	65 %
8 files á 8192 Byte	66,67%	Same as orig.	64 %	64 %
2 files á 1 MB	99,61%	Same as orig.	99 %	99 %
20 x 64 & 20 x 800	5,27%	Same as orig.	10 %	60 %

We have performed the benchmarks for the original version (with page cache) multiple times sequentially in order to get the relevant pages into the cache. We can see that when we don't use the page cache anymore (task 1.2), our reads and writes get slower. This doesn't really change after tasks 1.9 and 1.10, as the benchmarks are roughly the same (same magnitude). So with page cache, we can write and read faster. The performance loss differs based on what files we write.

However, if we look at the efficiency for tiny files (< 128 bytes), we can see that we get a 30x better efficiency when using the new sliced storage. For small files (files that can use multiple contiguous slices), we get 65 % efficiency with task 1.10, which is about 6x better than with the original version (~10 %). With files that use traditional blocks, we reach the same efficiency as expected. Allocating tiny and small files together, we can reach 10x better efficiency with the new functionality that handles files spanning multiple slices.

All in all, we can conclude that this new sliced storage system is definitely worth it when allocating small files (i.e. files smaller than 4096 – 128 bytes). It also does not result in any loss of efficiency for larger files. Task 1.11 would make it even more memory efficient. Currently, it is a bit slower than the original version, but with a page cache (that could be used in the future), we might get better in this regard as well.