Project 3: Final Report

Group Members

Group#: 83
Xinyu Fu (fuxy@berkeley.edu)
James Tang (jamestang@berkeley.edu)
Lanxiang Hu (hlxde1@berkeley.edu)

Changes Made Since Design Doc

Task 1: Buffer Cache

The majority of the design of buffer cache remains the same. However, we need to translate the block interface used by the underlying block device (disk) to the byte-by-byte interface used by the user program. We decided to handle this translation in `buffer_read()` and `buffer_write`, which would make the two functions easier to use (instead of always reading block by block and copy in and out of the blocks).

Task 2 Extensible Files

We decided to remove the `struct inode_disk data` field of `struct inode`, because we realized that we could use `sector` to keep track of `inode_disk`. In order to get `inode_disk` from `inode`, we read from cache with the sector number. This allowed us to save a lot of memory and make full advantage of our buffer cache instead.
Due to the complexity of the file extension task, our design doc did not fully flush out what we had to implement for resizing, but we used it as a foundation for our final design of file extension. We made the following changes to our original design for growing files.
For indirect and doubly indirect pointers, check if it has already been allocated. If not, call `free_map_allocate` to allocate a sector block to it before going into the allocation of the block_sector_t array it points to.
We did the same for the indirect pointer array (the first layer of the doubly pointer structure) and allocated it dynamically. Specifically, we only allocated a new sector for the indirect pointer when our allocation gets to that layer. This was to prevent having unused sectors taking up the memory of the free map.
In `inode_release`, we also made sure to deallocate any sector that is no longer required after rewinding the file extension.
We also fixed `byte_to_sector`, which returns the sector of some offset in an inode to account for our new tree structure of the file. This required a similar logic as file extension, but now we calculate which layer to do the lookup.
In the original design doc, we don't have a lock to support inode growth and resizing. Therefore, we added the `inode__lock` field to `struct inode`, locking the inode whenever we make changes to its contents such as `open_cnt` and `sector` to ensure synchronization among inode accesses.

Task 3 Subdirectories

We didn't add a file `block_sector_t parent_dir` in the `inode_disk` to keep track of the parent of a directory. We decided to use `dir_add` to create `..` and `.` of a directory directly. In addition, we also let the child process inherit CWD from its parent process, which was missed in our original design. The rest of the task 3 doesn't have some significant changes.
In our design doc, we did not address the issue for supporting "." and ".." in a path. To implement lookup for "." and "..", when a directory is created, we grow its inode_disk to the size `2 * sizeof(struct dir_entry)` instead of 0. We then used `dir_add()` to explicitly add two special directory entries in the initialized directory. In `dir_readdir()`, we specified that we skip these two directories when we are looping over entries, so they wouldn't be returned from the `readdir` syscall.
In project 1, `fd_entry` for each thread contains the field `open_file_object` to store `struct file`. This was problematic for our implementation for subdirectories because a thread could contain both directories and files in its `fd_entry`. Our solution for this was to define a new field variable called `file_or_dir`, containing a `struct dir` and a `struct file`, in accordance with the `is_dir` indicator. For the filesystem calls that needed to support both directory and file operations, at the `filesys.c` level, for example `filesys_open`, we would return `struct dir` instead of `struct file`, and in `syscall.c`, we perform the operations according to the `is_dir` indicator returned from the `filesys` function.

Reflection on Group Work

What each of us did:

- Xinyu Fu: Task 3 and help with debugging.
- James Tang: Task 1, buffer cache test cases and debugging.
- Lanxiang Hu: Task 2 and help with debugging.

Virtual/Asynchronous collaboration strategies:

Since project 3 is a huge project with a lot of details, to ensure the quality of our code, we chose to write code while sharing screens, so that other group members could help check the code while one member writes the code. This turned out to be a very effective method for us, as everyone can follow the thought process well and stay on the same page with others. This prevented us from making too many mistakes, especially in the math heavy part.
To avoid git merge conflict, we only let one person organize and manage the master branch. This goes well over the course of our project development since all of us were therefore able to develop the project and debug both individually and collaboratively via zoom with everybody informed about latest changes.

What went well:

Our design doc, especially for the first two tasks was very detailed with pseudo code, so implementing the main structure off of it did not take too long besides debugging.
We were able to work both individually and collectively this time toward the completion of this project. Each time we made progress, group members were notified and thus able to keep track of what bugs had been resolved and what part of the implementation had been done.
We were able to regularly reconvene for debugging, since it might not be as obvious for one person to locate the bug directly.

What could be improved:

We could make better use of office hours to ask questions. Since the pandemic has changed the office hours to online format, we never developed the habit to use online OH, and just tried to cluelessly debug ourselves. In the future, we should definitely make changes to this habit and learn to seek help from others.
Since this project involved some large-number iterations and loops, we found that it's generally inefficient to loop through code using the basic commands of gdb. If we could get ourselves more familiarized with other useful gdb commands (e.g.: break if condition, skip to the end of a function, deleting breakpoints, etc.), it would save us a lot more time debugging.
We did not mention enough about how to handle deleting current working directories in the design doc, so it took a while to get it figured out when writing the code.

Student Testing Report

Test 1: Buffer Cache Hit Rate (buffer-hit)

Description:

This tests the hit rate of buffer cache and see if it improves from being a cold cache initially to a hot cache.

Overview

In order to obtain the hit rate, we keep track of two global variables: `int num_hit` and `num_access`, and add corresponding syscalls. We then perform two rounds of read from the same file. In the first round, the cache is cold, so hit rate should be low. We use `num_buffer_hit()` and `num_buffer_access()` to get the number of hits and total accesses. In the second round however, the cache is hot and the hit rate should improve. Again we use the two syscalls to get hits and accesses. We then subtract the previous number of hits and accesses to obtain hits and accesses in the second round of read. Since the OS does not support floating point operation, if `hot_access * cold_hit` > `cold_access * hot_hit`, then the cache rate improves.

Output

```
Copying tests/filesys/extended/buffer-hit to scratch partition...
Copying tests/filesys/extended/tar to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/9sgypqNqOD.dsk -hdb tmp.dsk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading..............
Kernel command line: -q -f extract run buffer-hit
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  108,339,200 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 230 sectors (115 kB), Pintos OS kernel (20)
hda2: 247 sectors (123 kB), Pintos scratch (22)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
scratch: using hda2
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'buffer-hit' into the file system...
Putting 'tar' into the file system...
Erasing ustar archive...
Executing 'buffer-hit':
(buffer-hit) begin
(buffer-hit) create "newfoo"
(buffer-hit) open "newfoo"
(buffer-hit) reading "newfoo"
(buffer-hit) setting fd position to 0
(buffer-hit) reading "newfoo"
(buffer-hit) close "newfoo"
(buffer-hit) hit rate improves
(buffer-hit) end
buffer-hit: exit(0)
Execution of 'buffer-hit' complete.
Timer: 80 ticks
Thread: 41 idle ticks, 38 kernel ticks, 2 user ticks
hdb1 (filesys): 39 reads, 503 writes
hda2 (scratch): 246 reads, 2 writes
Console: 1215 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Result

PASS

Potential Kernel Bugs

1. When the kernel somehow fails to find the correct cache block in the second round of reading, it would always read into a new buffer cache block which means the hit rate would not improve. The test would output “hit rate does not improve”.

2. If the kernel fails to set `int num_hit` and `int num_access` to 0 when initializing, then the test might output “hit rate does not improve”.

Test 2: Buffer Cache Coalesce Writes (buffer-coal)

Description

Tests if buffer cache is able to coalesce writes.

Overview

We added a syscall `disk_write_cnt()` in order to get the number of writes to fs_device. First, we write a 64KB file byte-by-byte. Then read in the same file byte-by-byte. We keep track of the write count using `disk_write_cnt()`. The test passes if write count is on the order of 128.

Output

```
Copying tests/filesys/extended/buffer-coal to scratch partition...
Copying tests/filesys/extended/tar to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/p8s0uOou4R.dsk -hdb tmp.dsk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading..............
Kernel command line: -q -f extract run buffer-coal
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  111,411,200 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 230 sectors (115 kB), Pintos OS kernel (20)
hda2: 246 sectors (123 kB), Pintos scratch (22)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
scratch: using hda2
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'buffer-coal' into the file system...
Putting 'tar' into the file system...
Erasing ustar archive...
Executing 'buffer-coal':
(buffer-coal) begin
(buffer-coal) create "foonew"
(buffer-coal) open "foonew"
(buffer-coal) writing "foonew"
(buffer-coal) setting fd position to 0
(buffer-coal) reading "foonew"
(buffer-coal) reasonable write count (about 128)
(buffer-coal) close "foonew"
(buffer-coal) end
buffer-coal: exit(0)
Execution of 'buffer-coal' complete.
Timer: 1039 ticks
Thread: 90 idle ticks, 40 kernel ticks, 909 user ticks
hdb1 (filesys): 168 reads, 627 writes
hda2 (scratch): 245 reads, 2 writes
Console: 1251 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Result

PASS

Potential Kernel Bugs

1. If the kernel fails to reset the fd position back to 0 after the first write, then the test would output “unreasonable write count” since the program will read from the place where it finishes writing.

2. If the kernel flushes the buffer cache to disk too frequently, we would have multiple writes for the same block sector which would make the write count a lot larger than 128, which means the test would output “unreasonable write count”.

Experience Writing Tests

We added two syscalls to support getting two variables from the kernel. We think this should be improved. Since pintos tests test the entire operating system, at least some tests (like this one) should be able to access data in the kernel. Adding a kernel

mode or a kernel test module would be helpful. We definitely learned a lot writing pintos tests, it really helps understand from top to bottom how our OS should behave and what potential problems it has. We also learned that we should be careful and mindful of writing these tests since unexpected behavior could occur if we are not.