Project File System Report

Group 79

Name	Autograder Login	Email
Jacob Wu	student303	jacobwu@berkeley.edu
Max Ye	student95	maxye@berkeley.edu
Ryan Nader	student277	berkeleybear22ryan@berkeley.edu
Vedansh Malhotra	student39	vedansh@berkeley.edu

Changes

Buffer Cache

One of the most notable alterations was the transition from a global filesystem lock to a more granular locking mechanism. In our original design, we relied heavily on a global lock for managing access to the file system. This was a major bottleneck, limiting concurrency. In our new implementation, we introduced individual locks for each cache entry and a global cache lock. This change drastically improved our system's ability to handle concurrent accesses, allowing multiple threads to interact with different parts of the cache simultaneously without unnecessary blocking.
Another significant change was in the structure and management of cache entries. Our original design proposed a simpler approach to cache entry management, but as we developed the implementation, we realized the need for a more sophisticated system. We introduced the **`struct buffer_cache_entry`** and **`struct buffer_cache`**, which gave us a more robust framework for handling cache entries. This new structure allowed for more efficient implementation of the LRU (Least Recently Used) policy and better management of data integrity, particularly for dirty cache entries.

We also refined our approach to handling cache reads and writes. Initially, we had a more straightforward method for reading from and writing to the cache. However, our final implementation included a more complex, yet efficient, process involving acquiring and releasing locks at different stages to ensure data consistency. This was particularly important for our cache_write function, where we had to ensure that the data integrity was maintained while handling concurrent writes.

The LRU policy for cache eviction also saw significant improvements. In our original design, the LRU mechanism was basic, but as we moved into implementation, we developed a more advanced system for tracking and updating access times. This was facilitated by the introduction of the **`global_tick`** counter and the **`tick_lock`**, which helped in maintaining a precise and synchronized tracking mechanism for cache entry usage.
Finally, the cache initialization and cleanup processes were overhauled. We introduced a comprehensive initialization routine in **`cache_init`**, which systematically set up each cache entry and ensured that all necessary locks were correctly initialized. The cache_cleanup function was also refined to more effectively handle the flushing of dirty cache entries back to the disk, ensuring data integrity during system shutdowns or cache flushes.

Extensible Files

The first change was that the inode lock needs to be in inode disk, not node, for persistence, per OH. We also realized that at the level of the filesystem, what we're operating on is simply a file descriptor and an inode — whether that's a struct dir or a struct file needs to be resolved within the `filesys_` calls. The logic for our inode structure and resizing didn't change significantly from the design, but we ended up using 15 direct pointers since we had room and wanted to minimize the amount of growth necessary.

We also had to ensure to synchronize the free map and the ref count / deny-write count everywhere. For inode_resize, we found it incredibly useful to write multiple helper methods that took care of one single sub-functionality, and distribute work concurrently.

A big change here was that in order to write extensible files, we had to rebase to the skeleton code since our subdirectory / cache integration was doomed, and thus had to glom a lot of the changes described in the subdirectories / buffer cache sections.

Subdirectories

Instead of adding the `is_dir` member to `struct file_descriptor`, we used added the `is_dir` member to `struct inode_disk`. This `is_dir` member is initialized in `inode_create` and is used in the `sys_isdir` call.

We added the `parent` member to `struct inode_disk`, as opposed to adding it to `struct dir`, as it made it easier to work with if both the `is_dir` and `parent` members are in the same struct. `parent` is initialized in `filesys_create` with the helper function `inode_set_parent`.

For the `path_resolve` function, there were a few changes. We added a parameter `char file_part[NAME_MAX + 1]` to have the function store the file name so we could use it later. We also added a parameter called `void** i_ptr` with the similar rationale as with the file name. This pointer would be a pointer to ani node. We didn't a `bool stop_before_last` because the combination of getting both the directory (the return value) and inode of the file (if they respectively exist) was enough. Lastly, we changed `path_resolve` to return `struct dir*` instead of `struct inode*` because the line `dir_lookup(curr_dir, file_part, &curr_inode)` in the function can set `curr_inode` to NULL on the last iteration (right before we return), but it won't set `curr_dir` to NULL on the last iteration.

For the most part, however, the subdirectories design stayed the same as outlined in our original design document.

Reflection

Max: implemented subdirectories, debugged failing tests, added buffer cache testing (and the syscalls to support it.)

Max: it was hard to find time where we could work together because of conflicting time constraints due to grad school applications and other commitments (some people were free one week while the others were free another week). As such, integrating the three tasks was difficult.

Ryan: Worked on extensible files, debugged failing tests and debug codebase in general with Vedansh.

Ryan: I could have done better on abstracting away other parts of pintos and other parts of the project 3 in general as I spent a lot of time trying to understand everything and think that took a lot of time. Also could have done better managing time around finals week and the midterm for this course with the project 3.

Vedansh: Wrote the entire design, worked on extensible files (and thus, updated subdirectories), debugged failing tests and updated the design from PRJ-1 as needed for convenience, contributed to Max's work in the `testing` section of the report.

Vedansh: In this project, the painful realization was that integrating all three tasks was very challenging. Unfortunately, with grad school apps, finals, and other commitments, our group wasn't able to find times in our schedule wherein everyone was able to work together. This was probably the reason why we couldn't integrate all the changes together.

Jacob: ***[By proxy]*** Jacob wrote `cache.c` and `cache.h` and looked into `inode.c`.

Jacob: ***[By proxy]*** Jacob probably wishes we communicated more often and started earlier.

Testing

Disclosure: though the tests currently do not have the expected output, we implemented the entire buffer cache and wrote test cases that would've functioned had our project integrated everything. You can find parts relevant to our testing with ctrl+shift+F "buffer cache testing". The buffer cache is under filesys/cache.c and filesys/cache.h.
Also, our most latest commit has these test files, but the latest commit isn't the highest scoring one. Please refer to our highest scoring commit for the code quality portion.

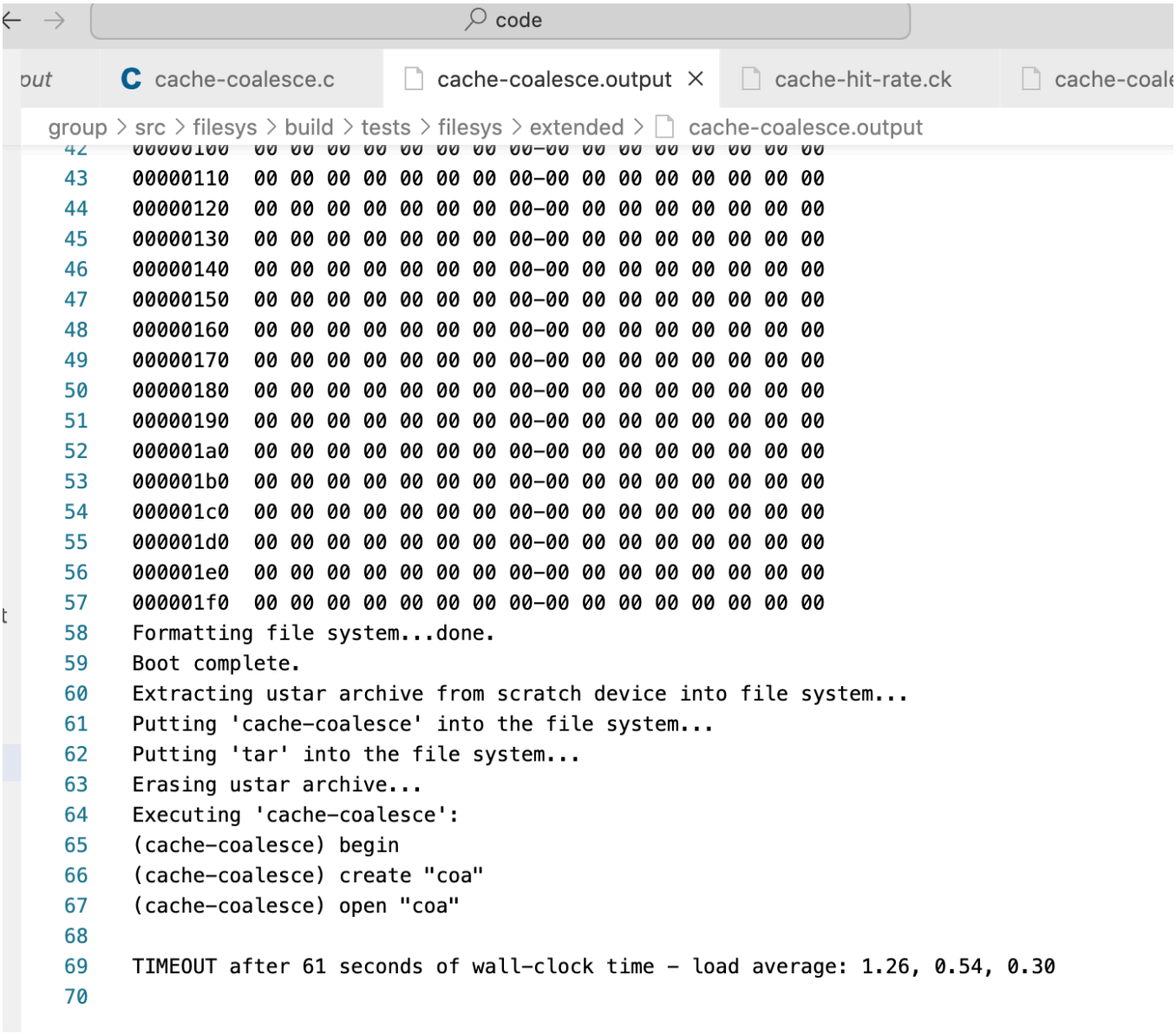
cache-hit-rate

- Description: tests our buffer cache's effectiveness by measuring its cache hit rate
  - First, we write a 4096 byte-sized buffer into a file
  - Next, we reset the buffer cache
  - Then, we open a file and read it sequentially to determine the cache hit rate for a cold cache
  - Then, we close the file, re-open it, and read it sequentially again, to ensure the cache hit rate improves
  - Output

- Result
  - Ideally, it would've been a file with "PASS" in it
- Two kernel bugs
  - If the kernel (in syscall.c) didn't validate the file descriptor, we would get a kernel panic when we ran `open` on the file descriptor instead of exiting with code -1
  - If the kernel (in `setup_stack` of process.c) didn't set the esp correctly (for instance, setting the esp above PHYS_BASE), we would get a page fault in our test output because we tried to read from memory that we shouldn't be reading from (when accessing the arguments for our syscall)

cache-coalesce

- Description: tests our buffer cache's ability to coalesce writes to the same sector
  - First, write a large file (64 KiB) byte-by-byte
  - Next, read that large file byte-by-byte
  - The total number of device writes should be on the order of 128 (because 64 KiB is 128 blocks)
- Output



- Result
  - Ideally, it would've been a file with "PASS" in it
- Two kernel bugs
  - If the kernel (in syscall.c) didn't validate the file descriptor, we would get a kernel panic when we ran `open` on the file descriptor instead of exiting with code -1
  - If the kernel (in `setup_stack` of process.c) didn't set the esp correctly (for instance, setting the esp above PHYS_BASE), we would get a page fault in our test output because we tried to read from memory that we shouldn't be reading from (when accessing the arguments for our syscall)

What can be improved about the Pintos testing system? If there was a way Pintos could automatically identify the new tests added, that would be helpful. This would reduce the need of adding the name of the new test to the `tests/filesys/extended_TESTS` variable in the corresponding `Make.tests` file.

Further, I find it really strange that the test suite doesn't accommodate incremental addition of functionality — why is there no test that allows you to see if you can create a small file that's just one block long, without also extensively relying on subdirectories? I believe at least one such test should be added, even if it's worth 0 points, so that students can learn how to create their own tests from it.

What did you learn from writing test cases? We learned how to add test cases to the Pintos testing system and to think of the edges cases that could happen for our functions. Though it's hard to write tests that cover every single edge case, it's definitely useful as it helps reduce bugs and clarifies your thinking. We learned that writing tests helps the coding process and is not simply an add-on.

Also, we learned some PERL scripting.. We also understood many of the helper methods that staff uses to test, for example in the syn tests there was another c file that actually runs most of the tests. This organization was somewhat illuminating.

In case we needed tests beyond just the 2 for buffer caches:

Custom Test (1): project3_ct1.c

```
/* Test every level of pointers and alerts tester at which level the
pointers no longer work and does not complicate as it keep writing 512 bytes at a time and confirms
that 512 bytes are written and allows tester to see which data corresponds to which pointers as all
dp's are writing 1's, all ip's are writing 2's and the first dip is writing 3's
If my kernel did not correctly handle double indirect pointers, then the test case would output "failed: filling up  double indirect pointe
rs ..." instead
*/
#include "tests/lib.h"
#include "tests/main.h"
#include <stdio.h>
void test_main(void) {
    create("file.txt", 0);
    int fd = open("file.txt");
    int number_of_direct_pointers = 15;
    int number_of_indirect_pointers = 128;
    // due to 2MiB disk max size ... just decided to make 1 but could fill to 128^2
    int number_of_double_indirect_pointers =  1;
    msg("starting up test case 1 ...");
    int retval;
    char buff[512];
    for (int i = 0; i < 512; i++) {
        buff[i] = '1';
    }
    msg("filling up direct pointers ...");
    for (int i = 0; i < number_of_direct_pointers; i++) {
        retval = write(fd, buff, 512);
        if (retval != 512) {
            msg("failed: filling up direct pointers ...");
        }
        if (i == 0) {
            msg("added first direct pointer ...");
        }
    }
    for (int i = 0; i < 512; i++) {
        buff[i] = '2';
    }
    msg("filling up indirect pointers ...");
    for (int i = 0; i < number_of_indirect_pointers; i++) {
        retval = write(fd, buff, 512);
        if (retval != 512) {
            msg("failed: filling up indirect pointers ...");
        }
        if (i == 0) {
            msg("added first indirect pointer ...");
        }
    }
    for (int i = 0; i < 512; i++) {
        buff[i] = '3';
    }
    msg("starting up double indirect pointers ...");
    for (int i = 0; i < number_of_double_indirect_pointers; i++) {
        retval = write(fd, buff, 512);
        if (retval != 512) {
            msg("failed: filling up  double indirect pointers ...");
        }
        msg("added first & only double indirect pointer ... (due to max_disk_size=2MiB)");
    }
}
```

Custom Test (2): project3_ct1.ck

```
# -*- perl -*-
```

```
use strict;
use warnings;
use tests::tests;
check_expected (IGNORE_EXIT_CODES => 1, [<<'EOF']);
(project3_ct1) begin
(project3_ct1) starting up test case 1 ...
(project3_ct1) filling up direct pointers ...
(project3_ct1) added first direct pointer ...
(project3_ct1) filling up indirect pointers ...
(project3_ct1) added first indirect pointer ...
(project3_ct1) starting up double indirect pointers ...
(project3_ct1) added first & only double indirect pointer ... (due to max_disk_size=2MiB)
(project3_ct1) end
EOF
pass;
```

Custom Test (2): project3_ct2.c

```
/* Tests writing 3 bytes, seeking to byte 5 (0 indexed), telling and getting byte 5 b/c that should be where you are, then
writing 3 bytes and telling and getting byte 8 b/c that is now where you are and then seeking to 0 and making sure you can
read 8 bytes
If my kernel did not correctly handle giving 0's, then the test case would error instead
*/
#include "tests/lib.h"
#include "tests/main.h"
#include <stdio.h>
void test_main(void) {
    msg("starting up test case 2 ...");
    create("file.txt", 0);
    int fd = open("file.txt");
    int retval;
    // bytes 0, 1, 2 should be 1
    char buf1[3] = {'1', '1', '1'};
    retval = write(fd, buf1, 3);
    if (retval != 3) {
        msg("failed");
    }
    // bytes 3, 4 == 0 after write b/c should start write at file_ptr=5
    seek(fd, 5);
    retval = tell(fd);
    if (retval != 5) {
        msg("failed");
    }

    char buf2[3] = {'2', '2', '2'};
    write(fd, buf2, 3);

    // now bytes 5, 6, 7 should be 2
    retval = tell(fd);
    if (retval != 8) {
        msg("failed");
    }
    // set back to 0
    seek(fd, 0);
    char buf_res[8];
    retval = read(fd, buf_res, 8);
    if (retval != 8) {
        msg("failed");
        printf("failed_result: \n 0: %c\n 1: %c\n 2: %c\n 3: %c\n 4: %c\n 5: %c\n 6: %c\n 7: %c\n", buf_res[0], buf_res[1], buf_res[2], buf
_res[3], buf_res[4], buf_res[5], buf_res[6], buf_res[7]);
    }
```

```
    printf("correct_result: \n 0: %c\n 1: %c\n 2: %c\n 5: %c\n 6: %c\n 7: %c\n", buf_res[0], buf_res[1], buf_res[2], buf_res[5], buf_res[6], buf_res[7]);
}
```

project3_ct2.ck

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected (IGNORE_EXIT_CODES => 1, [<<'EOF']);
(project3_ct2) begin
(project3_ct2) starting up test case 2 ...
correct_result:
 0: 1
 1: 1
 2: 1
 5: 2
 6: 2
 7: 2
(project3_ct2) end
EOF
pass;
```