

# Project File System Report

Group 36

Name	Autograder Login	Email
Sidd Shashi	student63	sshashi@berkeley.edu
Aman Doshi	student457	aman.doshi@berkeley.edu
Matthew Nguyen	student161	matthew.t.nguyen@berkeley.edu
Pritish Parmar	student251	prishparmar@berkeley.edu

## Changes

### Buffer Cache

To implement the buffer cache, we made 2 major changes to our original design. The first change stems from our discussion with our TA in the design review.

1. In our original design, any thread accessing the buffer cache would hold the global buffer cache lock while reading/writing into a cache block. This approach serializes access to the buffer cache, preventing concurrent access to different cache blocks.  
To resolve this problem, we tweaked our design by introducing a reference count (for cache block access) and condition variable (for sleeping) for each cache block. In the new design, threads no longer hold the global buffer cache lock while reading/writing into a cache block, allowing for concurrent access to different cache blocks. Additionally, threads wait in a condition variable if the associated cache block is in use (reference count > 0). This ensures serial access for individual cache blocks, which allows for synchronization.
2. In our original design, a thread accessed the buffer cache through a `buffer_cache_access()` function, which would read/write into a cache block. This approach was impractical in code for specific read/write operations (e.g., update direct pointer in a struct `inode_disk`).

To resolve this problem, we split the `buffer_cache_access` function into 2 parts: `buffer_cache_acquire()` and `buffer_cache_release()`. These functions act as `lock_acquire()` and `lock_release()`, allowing a thread to acquire mutually exclusive rights to a particular cache block.

The use of these 2 functions enabled greater flexibility of read/write operations into a cache block. For example (shown below), a `block_buffer` can be cast into a `struct inode_disk`, allowing access to specific struct members.

```
void* block_buffer = buffer_cache_acquire(block_index, write=false)
read ((struct inode_disk) block_buffer)->length
buffer_cache_release(block_buffer)
```

## Extensible Files

To implement extensible files, we made 2 minor changes compared to our original design, the first of which was inspired by our design review with our TA.

1. In our original design, we had two locks in the in-memory inode struct (one for inode extensions, one for inode metadata access). To simplify our design, we instead used a single lock, used for inode extension AND inode metadata access. Having a single lock simplified our synchronization and possibly avoided deadlock situations (if both locks are accessed in different orders by different threads).
2. We introduced a macro `INODE_NUM_DP` to track the number of direct block pointers in a `struct inode_disk`. This allowed us to easily change the structure of our `struct inode_disk` (e.g., reduce/increase number of direct block pointers) without messing up computation in our `inode_file_resize` function.

## Subdirectories

To implement subdirectories, we made 3 major changes. The first change was inspired by our design review with our TA.

1. In our original design, we had separate logic to handle `."` and `.."` in our path resolution helper function. Instead, we created entries in the directory itself, where `."` points to itself and `.."` points to the parent directory. This simplified the code for our path resolution helper function, as we don't need special cases for `."` and `.."` anymore.
2. After starting to code, we realized that our root directory was uninitialized, which resulted in failing tests. To resolve this issue, we initialized the root directory in

the `filesystem_init()` function, when `FORMAT` is true (i.e., when the file system needs to be reformatted).

3. We created a new helper function called `dir_split_file_path()`, which separates a path into the file name and directory location. For example, `dir_split_file_path("/a/b/c/d/e")` splits a path into `"a/b/c/d"` and `"e"`. This is useful for the `mkdir` and `create` syscall, as we want to separate a provided path into a valid directory path and the new file/directory name.
- 

## Reflection

### Overall Working Environment

Very similar to the working environment in project 1 and 2, our group has remained active, persistent, and organized. From day 1, we have made steady progress to project deadlines by scheduling frequent group meetings and keeping ourselves accountable. Each meeting, we outline concrete goals and push to achieve them.

We understood this project would be demanding, with the final midterm and RRR week due date. Accordingly, our group invested extra effort into the design document to ensure our ideas were fleshed out and well thought-through. Before the last midterm, we pushed to complete extensible files. After the midterm, we made another push to complete sub-directories, and then buffer cache. Most of our code development was on VSCode Live Share, which helped with collaborative development and shared knowledge. This especially helped with brain storming and debugging in GDB. The overall group dynamic was positive.

### What Went Well

Throughout this project, our group retained a clear and respectful channel of communication. This was key to our steady and active progress.

In the design stages, we repeatedly communicated our ideas, confusion, and concerns during group meetings. This enabled a fleshed-out design document that we all understood and had contributed towards. In the coding stage, we outlined a timeline of soft goals to make sure the project was completed in a timely manner. We always communicated our availability, to ensure we met as often as possible to meet these soft goals. We debugged code in groups of 2 or 3, allowing for improved brain storming and error spotting. When the midterm approached, we allowed all group

members sufficient time to prepare, then pushed immediately afterwards, allowing us to finish this project on time.

### **Areas of Improvement**

During the design stage, we brainstormed many different ideas in our regular group meetings, but only made progress on the actual design document near the deadline. Inevitably, there was a major rush to finish the final design document, which could have led to some oversights. Moving forward, we should summarize our meetings ideas into the actual design document, to ensure steady progress and fewer oversights.

Additionally, only two group members worked on extensible files, due to lack of availability from other members. Accordingly, only these group members knew the code well, making it difficult for the entire group to debug the extensible files section. To prevent these issues for future sections, we ensured all group members had wholistic understanding of the code and were directly involved in code development.

Due to the last midterm and project due date in RRR week, this project was quite time constraining on all group members. We chose to not work during the week of the last midterm, to allow for sufficient revision time. However, this led to a large push after the midterm, well into RRR week. Instead, we could have made a larger push before the last midterm, allowing for less stress and rigor during RRR week.

### **Sidd Shashi:**

For designing, Sidd designed and wrote each of the sections in some capacity, particularly spending a lot of time on extensible files, subdirectories, and the concept check.

Writing code, Sidd worked closely with Aman on extensible files, creating the data structures and writing the resize function. He then worked with the group on subdirectories, writing the new syscalls and designing the first iteration of the `dir_split_file_path()` function. On the buffer cache, he worked with the rest of the group, creating data structures, figuring out how to best write the functions and find the places where to call the buffer cache access functions.

For testing and the final report, Sidd came up with high-level ideas for both tests and wrote a lot of the code needed for bc-hit-rate. He also wrote much of the

changes and testing sections on this report document. Finally, he helped the rest of the group in the final code clean-up and review process.

#### **Aman Doshi:**

In the design section, Aman worked on all sub-sections of the project. For the buffer cache, he worked on the buffer cache data structure design and access pseudocode. He also worked on all parts of extensible files and sub-directories. He also helped brainstorm ideas for the concept check questions.

During code development, he worked alongside Sidd on the extensible files section, focusing most of his time on the inode resize function. In the sub-directories section, he mainly worked on the functions in `filesys.c` and `directory.c`, so they could integrate with the system calls. Finally, he worked with the entire group on the buffer cache section, heavily involved with buffer cache access and updating block read/write references in `inode.c` to interface with the buffer cache.

For testing, Aman helped writing syscalls and integrating the tests into the Pintos test structure. He worked on parts of the bc-hit-rate test and most of the bc-write test. He also helped in the final code clean-up and review process, prior to project submission.

#### **Matthew Nguyen:**

In the design section, Matthew mostly worked on designing extensible files and subdirectories. Furthermore, he wrote the concept check section and gained help from Sidd and Aman for ideas when writing it.

For the code writing part of the project, Matthew mainly worked on the subdirectories section and the buffer cache section. More specifically, he helped implement/update syscall functions (e.g. `filesys_create` and `filesys_remove`) and buffer cache functions like `buffer_cache_access` and `inode_file_resize`. He also helped identify bugs in the subdirectories section, mainly relating to `dir-vine` and `dir-mk-tree`.

Matthew also helped write the 2 new Pintos tests, bc-hit-rate and bc-write, by helping to create new syscalls for the tests, write the test `.c` file, and the Perl `.ck` file.

#### **Pritish Parmar:**

Prithish mainly worked on the actual implementation of the project, as well as helping brainstorm ideas for the buffer cache. He contributed to the subdirectories section, and buffer cache section alongside all members of the group.

Also helped debug the implementation of subdirectories section, and appropriately writing functions for the said section. As well as working with the rest of group to update the implementation of buffer cache.

Lastly, worked with the group on brainstorming and creating the first test of the two, appropriately calling the syscall with appropriate arguments.

---

## Testing

### Test 1: bc-hit-rate

#### Description:

The Pintos test **bc-hit-rate** tests the buffer cache's effectiveness by measuring its hit rate. The test calculates the hit rate of a cold cache and a hot cache, and checks that the hit rate has improved between the two states.

#### Test Mechanics and Expected Output:

In `inode.c`, we have created functions `buffer_cache_hit_rate()` and `buffer_cache_reset()` to respectively get the buffer cache's hit rate and reset the buffer cache. New syscalls interface with these functions, and are called from our test (outlined below).

Here is a chronological overview of the test mechanics.

1. Create a large file with `create()` syscall that can fit into the buffer cache (e.g., 10KiB).
2. Reset buffer cache with `bc_reset()` syscall. The buffer cache is now cold.
3. Open the file with `open()` syscall.
4. Read the file in a loop with `read()` syscall, 256B at a time until EOF.
5. Get the cold cache hit rate with `bc_stat()` syscall.
6. Close the file with `close()` syscall, then reopen with `open()` syscall.
7. Read the file again in a loop with `read()` syscall, 256B at a time until EOF.
8. Get the hot cache hit rate with `bc_stat()` syscall.

9. Compare cache hit rates, checking that final hit rate (hot cache) > initial hit rate (cold cache).

At each step, we output a message describing what action has occurred. If a step does not occur or fails, then the test will fail and abort. Critically, if the cache hit rate does not improve between a cold and hot cache, the test will fail.

Output and Results:

Initial (cold) cache hit rate: 87%

Final (hot) cache hit rate: 94%

```
C bc-hit-rate.c  bc-hit-rate.output X
filesys > build > tests > filesystems > extended > bc-hit-rate.output
1 Copying tests/filesystems/extended/bc-hit-rate to scratch partition...
2 Copying tests/filesystems/extended/tar to scratch partition...
3 qemu-system-i386 -device isa-debug-exit -hda /tmp/CqRkrfXZu0.dsk -hdb tmp.dsk -m 4 -net none -nographic -monitor null
4 qemu-c[?7L]mSeaBIOS (version 1.15.0-1)
5 Booting from Hard Disk...
6 PPiLLoo hddaa1
7 1
8 LLoaaaddiinnngg.....
9 Kernel command line: -q -f extract run bc-hit-rate
10 Pintos booting with 3,968 kB RAM...
11 367 pages available in kernel pool.
12 367 pages available in user pool.
13 Calibrating timer... 190,668,800 loops/s.
14 ide0: unexpected interrupt
15 hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
16 hda1: 244 sectors (122 kB), Pintos OS kernel (20)
17 hda2: 297 sectors (148 kB), Pintos scratch (22)
18 hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
19 hdb1: 4,096 sectors (2 MB), Pintos file system (21)
20 ide1: unexpected interrupt
21 filesystem: using hdb1
22 scratch: using hda2
23 Formatting file system...done.
24 Boot complete.
25 Extracting ustar archive from scratch device into file system...
26 Putting 'bc-hit-rate' into the file system...
27 Putting 'tar' into the file system...
28 Erasing ustar archive...
29 Executing 'bc-hit-rate':
30 (bc-hit-rate) begin
31 (bc-hit-rate) Create file "test".
32 (bc-hit-rate) Open file "test".
33 (bc-hit-rate) Total bytes read 10240.
34 (bc-hit-rate) Get hit rate for cold cache.
35 (bc-hit-rate) Reopen file "test".
36 (bc-hit-rate) Total bytes read 10240.
37 (bc-hit-rate) Get hit rate for hot cache.
38 (bc-hit-rate) Improved hit rate for hot cache.
39 (bc-hit-rate) end
40 bc-hit-rate: exit(0)
41 Execution of 'bc-hit-rate' complete.
42 Timer: 88 ticks
43 Thread: 26 idle ticks, 26 kernel ticks, 37 user ticks
44 hdb1 (filesystem): 684 reads, 621 writes
45 hda2 (scratch): 296 reads, 2 writes
46 Console: 1373 characters output
47 Keyboard: 0 keys pressed
48 Exception: 0 page faults
49 Powering off...
50
```

*bc-hit-rate.output*

```
C bc-hit-rate.c  bc-hit-rate.result X
filesys > build > tests > filesystems > extended > bc-hit-rate.result
1 PASS
2
```

*bc-hit-rate.result*



```
bc-hit-rate.c  bc-hit-rate-persistence.output x
fileys > build > tests > fileys > extended > bc-hit-rate-persistence.output
1  qemu-system-i386 -device isa-debug-exit -hda /tmp/Sf8yxDG8l_.dsk -hdb tmp.dsk -m 4 -net none -nographic -monitor null
2  c (77l 2J 0mSeaBIOS (version 1.15.0-1)
3  Booting from Hard Disk...
4  PPiLLoo hhddaa1
5  1
6  LLoaaaddiinnngg.....
7  Kernel command line: -q run 'tar fs.tar /' append fs.tar
8  Pintos booting with 3,968 kB RAM...
9  367 pages available in kernel pool.
10 367 pages available in user pool.
11 Calibrating timer... 157,081,600 loops/s.
12 ide0: unexpected interrupt
13 hda: 3,024 sectors (1 MB), model "QM00001", serial "QEMU HARDDISK"
14 hda1: 244 sectors (122 kB), Pintos OS kernel (20)
15 hda2: 2,048 sectors (1 MB), Pintos scratch (22)
16 hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
17 hdb1: 4,096 sectors (2 MB), Pintos file system (21)
18 ide1: unexpected interrupt
19 fileys: using hdb1
20 scratch: using hda2
21 Boot complete.
22 Executing 'tar fs.tar /':
23 tar: exit(0)
24 Execution of 'tar' complete.
25 Appending 'fs.tar' to ustar archive on scratch device...
26 Timer: 87 ticks
27 Thread: 27 idle ticks, 22 kernel ticks, 38 user ticks
28 hdb1 (fileys): 1010 reads, 325 writes
29 hda2 (scratch): 0 reads, 322 writes
30 Console: 869 characters output
31 Keyboard: 0 keys pressed
32 Exception: 0 page faults
33 Powering off...
34 Copying tests/fileys/extended/bc-hit-rate.tar out of /tmp/Sf8yxDG8l_.dsk...
35
```

*bc-hit-rate-persistence.output*

```
bc-hit-rate.c  bc-hit-rate-persistence.result x
fileys > build > tests > fileys > extended > bc-hit-rate-persistence.result
1  PASS
2
```

*bc-hit-rate-persistence.result*

### Potential Kernel Bugs:

1. Ideal: The kernel's buffer cache allows for multiple reads from the same disk block. This will lead to an improved hit rate as the cache becomes hot.  
Bug: The kernel's buffer cache only reads from a disk block once, even if multiple sequential reads target the same disk block. This means disk blocks do not persist inside the buffer cache. This results in a block read (miss) for every buffer

cache read operation. This will result in no improvement for the buffer cache hit rate. The test will fail and output an error message.

2. Ideal: The kernel's buffer cache stores the file, even after the file has been closed. Accordingly, when the file is re-opened, most/all of the file is in the buffer cache, allowing for improved buffer cache hit rate.

Bug: When the file is closed, the file is flushed out of the buffer cache.

Accordingly, the final hit rate will be the same as the initial hit rate, since the buffer cache starts in the same state for the first file read and the second file read. There will be no improvement in the buffer cache hit rate, causing the test to fail and output an error message.

## Test 2: bc-write

### Description:

The Pintos test **bc-write** tests the buffer cache's ability to coalesce writes to the same sector. A large 64KiB file is written to disk byte-by-byte. The total number of device writes should be on the order of 128 (64KiB = 128 blocks).

### Mechanics and Expected Output:

In block.c, we have written a block\_write\_cnt() function, which gets the total number of block writes for a block device. The bc\_stat() syscall interfaces with this function, getting the number of device writes for fs\_device. The syscall is called from our test (outlined below).

Here is a chronological overview of the test mechanics.

1. Get the initial device write count with bc\_stat() syscall. Store result in initial\_write\_cnt.
2. Create a file with create() syscall, with initial size 0. Open the file with open() syscall.
3. Write 64KiB to the file, **byte-by-byte**, in a loop with the write() syscall.
4. Read 64KiB from the file, **byte-by-byte**, in a loop with the read() syscall.
5. Get the final device write count with bc\_stat() syscall. Store result in final\_write\_cnt.
6. Calculate the net device write count (write\_cnt = final\_write\_cnt - initial\_write\_cnt). Check write\_cnt is on the order of 128 (e.g., write\_cnt <= 128 \* 1.25).

At each step, we output a message describing what action has occurred. If a step does not occur or fails, then the test will fail and abort. Critically, if the net write\_cnt

is not on the order of 128 (e.g., `write_cnt > 128 * 1.25`), the test will fail.

### Output and Results:

`write_cnt = 132`

Our `write_cnt` is on the order of 128 ( $132 < 128 \cdot 1.25 = 160$ ).

```
C bc-write.c  bc-write.output X
fileSYS > build > tests > fileSYS > extended > bc-write.output
1  Copying tests/fileSYS/extended/bc-write to scratch partition...
2  Copying tests/fileSYS/extended/tar to scratch partition...
3  qemu-system-i386 -device isa-debug-exit -hda /tmp/glzvIpPA6w.dsk -hdb tmp.dsk -m 4 -net none -nographic -monitor null
4  SeaBIOS (version 1.15.0-1)
5  Booting from Hard Disk...
6  PPiLLoo  hhddaa1
7  1
8  LLoaaaddiinnngg.....
9  Kernel command line: -q -f extract run bc-write
10 Pintos booting with 3,968 kB RAM...
11 367 pages available in kernel pool.
12 367 pages available in user pool.
13 Calibrating timer... 211,353,600 loops/s.
14 ide0: unexpected interrupt
15 hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
16 hda1: 244 sectors (122 kB), Pintos OS kernel (20)
17 hda2: 298 sectors (149 kB), Pintos scratch (22)
18 hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
19 hdb1: 4,096 sectors (2 MB), Pintos file system (21)
20 ide1: unexpected interrupt
21 fileSYS: using hdb1
22 scratch: using hda2
23 Formatting file system...done.
24 Boot complete.
25 Extracting ustar archive from scratch device into file system...
26 Putting 'bc-write' into the file system...
27 Putting 'tar' into the file system...
28 Erasing ustar archive...
29 Executing 'bc-write':
30 (bc-write) begin
31 (bc-write) Create file "test".
32 (bc-write) Open file "test".
33 (bc-write) Write >= 64KiB to file.
34 (bc-write) File has size >= 64KiB.
35 (bc-write) Read >= 64KiB from file.
36 (bc-write) The total number of device writes is on order of 128.
37 (bc-write) end
38 bc-write: exit(0)
39 Execution of 'bc-write' complete.
40 Timer: 501 ticks
41 Thread: 48 idle ticks, 31 kernel ticks, 422 user ticks
42 hdb1 (fileSYS): 776 reads, 732 writes
43 hda2 (scratch): 297 reads, 2 writes
44 Console: 1277 characters output
45 Keyboard: 0 keys pressed
46 Exception: 0 page faults
47 Powering off...
48
```

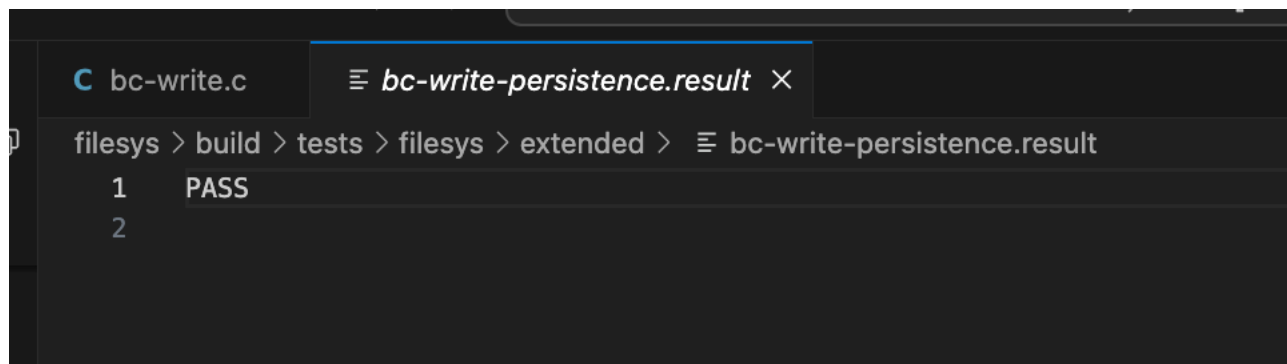
*bc-write.output*

```
bc-write.c  bc-write.result X
fileSYS > build > tests > fileSYS > extended > bc-write.result
1 PASS
2
```

*bc-write.result*

```
bc-write.c  bc-write-persistence.output X
fileSYS > build > tests > fileSYS > extended > bc-write-persistence.output
1 qemu-system-i386 -device isa-debug-exit -hda /tmp/8xM1vePgYv.dsk -hdb tmp.dsk -m 4 -net none -nographic -monitor null
2 qemu: [77L][2J] 0mSeaBIOS (version 1.15.0-1)
3 Booting from Hard Disk...
4 PiiLoo hddaa1
5 1
6 LLoaaaddiinnngg.....
7 Kernel command line: -q run 'tar fs.tar /' append fs.tar
8 Pintos booting with 3,968 kB RAM...
9 367 pages available in kernel pool.
10 367 pages available in user pool.
11 Calibrating timer... 189,644,800 loops/s.
12 ide0: unexpected interrupt
13 hda: 3,024 sectors (1 MB), model "QM00001", serial "QEMU HARDDISK"
14 hda1: 244 sectors (122 kB), Pintos OS kernel (20)
15 hda2: 2,048 sectors (1 MB), Pintos scratch (22)
16 hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
17 hdb1: 4,096 sectors (2 MB), Pintos file system (21)
18 ide1: unexpected interrupt
19 fileSYS: using hdb1
20 scratch: using hda2
21 Boot complete.
22 Executing 'tar fs.tar /':
23 tar: exit(0)
24 Execution of 'tar' complete.
25 Appending 'fs.tar' to ustar archive on scratch device...
26 Timer: 104 ticks
27 Thread: 29 idle ticks, 24 kernel ticks, 51 user ticks
28 hdb1 (fileSYS): 1342 reads, 435 writes
29 hda2 (scratch): 0 reads, 431 writes
30 Console: 870 characters output
31 Keyboard: 0 keys pressed
32 Exception: 0 page faults
33 Powering off...
34 Copying tests/fileSYS/extended/bc-write.tar out of /tmp/8xM1vePgYv.dsk...
35
```

*bc-write-persistence.output*



```
bc-write.c  bc-write-persistence.result X
fileys > build > tests > fileys > extended > bc-write-persistence.result
1 PASS
2
```

*bc-write-persistence.result*

### Potential Kernel Bugs:

1. Ideal: The Kernel's buffer cache is write-back. This allows 1B writes to coalesce in a buffer cache block, and be written back to disk together on eviction.  
Bug: The Kernel's buffer cache is write-through. This will result in a device write on every 1B write operation to the buffer cache. Accordingly, the device writes will be on the order of 64K instead of 128. Hence, the test will either time out (from too many device writes) or fail and output an error message, since device writes is not on the order of 128.
2. Ideal: Buffer cache blocks that are only read (not written) are not dirty blocks. Hence, they are not written back to disk on eviction.  
Bug: The buffer cache marks cache blocks that are only read (not written) as dirty blocks. Accordingly, these blocks will be written back to disk on eviction. When the entire 64KiB file is read, many of these blocks will be evicted and unnecessarily written back to disk. Accordingly, the test will fail and output an error message due to high device writes (not on the order of 128).

### **Overall Testing Experience:**

#### Writing Pintos Tests:

Overall, we had a positive experience writing tests for this project. After writing tests in projects 1 and 2, they seem more simple and intuitive to implement. Furthermore, the Pintos specification provides a clear outline on how to write Pintos tests, which helped us seamlessly integrate our tests into the provided testing framework.

#### Suggested Improvements:

As mentioned in our Project 2 report, students can only test in two ways.

1. Run tests individually

2. Run “make check” to run all tests at the same time

This testing framework takes a lot of time to test certain sections of the project (e.g., extensible files). As a result, we suggest that the Pintos testing framework implement a system to easily group specific tests together, so students can run them in one go. In the long run, this would decrease the testing time for individual sections (e.g., extensible files, subdirectories).

Moreover, we reiterate the suggested improvements outlined in Project 1. Writing tests in Pintos is more effort compared to other testing suites our group has used in other classes (e.g., JUnit in CS 61B, GO in CS 161). For an improved testing framework, the Pintos testing suite should allow the programmer to integrate the test and expected result into one C file, instead of splitting it across a C file and a CK file. This change would make it easier to write tests and would declutter the testing folders.

#### What We Learned:

Like in Projects 1 and 2, we have learned a considerable amount from writing tests.

1. Multi-threaded code can be difficult to debug, and requires carefully thought-out tests to exploit potential errors. This realization has taught us the importance of designing a test before actual implementation.
2. There are numerous edge cases for large code systems like Pintos. Accordingly, a good testing suite should thoroughly test these edge cases to ensure the Kernel handles them appropriately and does not panic. This has taught us the importance of having a variety of tests that think outside the box.
3. After debugging for countless hours in Pintos, we have learned the importance of descriptive outputs from failed test cases. Detailed descriptions make it faster to identify bugs in Pintos and resolve the test. Accordingly, tests should be written with detailed comments and outputs to aid the programmer as much as possible.