Project 3: File System

Due: May 4, 2022

Contents

1 Introduction

Welcome to Project File Systems! In this project, you will add features to the file system of Pintos.

In Project User Programs, you implemented much of the syscall functionality for the file system syscalls. However, much of the internals were abstracted away from you as you called existing file system functions. In this project, you will dive deep into the file system all the way down to the bytes that are stored on disk. In addition, you will also work on adding a buffer cache to speed up accesses to disk.

1.1 Setup

While you may build off of your solution to Project Threads, we **strongly recommend** that you build off of your Project User Programs solution. The tests for Project Threads are not very comprehensive, and as such may leak synchronization bugs into your implementation for this project. Most student implementations will not necessarily have these sorts of issues, but it's generally better to be safe than sorry.

First, log into your VM. Similar to what you did when starting Project Threads, we recommend that you tag your final Project Threads code in case you ever want to refer back to it.

```
> cd ~/code/group
> git tag proj-threads-completed
> git push group master --tags
```

Next, restore your code to your Project User Programs solution.

```
> git fetch --all --tags --jobs=2
> rm -rf src/
> git pull group master
> git checkout proj-userprog-completed -- src/
> git commit -m "Restore project userprog solution"
> git push group master
```

2 Tasks

2.1 Buffer Cache

The functions `inode_read_at` and `inode_write_at` currently access the file system's underlying block device directly each time you call them. Your task is to add a buffer cache for the file system, to improve the performance of reads and writes. Your buffer cache will cache individual disk blocks, so that (1) you can respond to reads with cached data and (2) you can coalesce multiple writes into a single disk operation. The buffer cache should have a maximum capacity of 64 disk blocks. You may choose the block replacement policy, but it should be an approximation of MIN based on locality assumptions. For example, using LRU, NRU (clock), nth chance clock, or second-chance lists would be acceptable, but using FIFO, RANDOM, or MRU would *not* be. Choosing a replacement policy of your own design would require serious justification. The buffer cache must be a **write back cache**, not a write-through cache. You must make sure that all disk operations use your buffer cache, not just the two inode functions mentioned earlier.

2.2 Extensible Files

Pintos currently cannot extend the size of files because the Pintos file system allocates each file as a single contiguous set of blocks. Your task is to modify the Pintos file system to support extending files. Your design should provide fast random accesses to the file, so you should avoid using a design based on File Allocation Tables (FAT). One possibility is to use an indexed inode structure with direct, indirect, and doubly-indirect pointers, similar to Unix FFS. The maximum file size you need to support is 8 MiB (2^{23} bytes). You must also add support for a new system call `inumber(int fd)` which returns the unique inode number of file associated with a particular file descriptor. Make sure that you gracefully handle cases where the operating system runs out of memory or disk space by leaving the file system in a consistent state (especially with regard to inode extension) and without leaking disk space or memory.

2.3 Subdirectories

The current Pintos file system supports directories, but user programs have no way of using them (i.e. files can only be placed in the root directory right now). You must add the following system calls to allow user programs to manipulate directories: `chdir`, `mkdir`, `readdir`, `isdir`. You must also update the following system calls so that they work with directories: `open`, `close`, `exec`, `remove`, `inumber`. You must also add support for **relative paths** for any syscall with a file path argument. For example, if a process calls `chdir("my_files/")` and then `open("notes.txt")`, you should search for `notes.txt` relative to the current directory and open the file `my_files/notes.txt`. You also need to support absolute paths like `open("/my_files/notes.txt")`. You need to support the special `"."` and `".."` names, when they appear in file path arguments, such as `open("../logs/foo.txt")`. Child processes should inherit the parent's current working directory. The first user process should have the root directory as its current working directory.

2.4 Concept Check

1. There are 2 optional buffer cache features that you can implement: write-behind and read-ahead. A buffer cache with write-behind will periodically flush dirty blocks to the file system block device, so that if a power outage occurs, the system will not lose as much data. Without write-behind, a write-back cache only needs to write data to disk when (1) the data is dirty and gets evicted from the cache or (2) the system shuts down. A cache with read-ahead will predict which block the system will need next and fetch it in the background. A read-ahead cache can greatly improve the performance of sequential file reads and other easily-predictable file access patterns.

   Please discuss a possible implementation strategy for each feature. **You must answer this question regardless of whether you actually decide to implement these features.**

2.5 Testing

Pintos already contains a test suite for file system functionalities, but it does not cover the buffer cache. For this project, you must implement **two** of the following test cases:

- Test your buffer cache's effectiveness by measuring its cache hit rate. First, reset the buffer cache. Next, open a file and read it sequentially, to determine the cache hit rate for a cold cache. Then, close it, re-open it, and read it sequentially again, to make sure that the cache hit rate improves.
- Test your buffer cache's ability to coalesce writes to the same sector. Each block device keeps a `read_cnt` counter and a `write_cnt` counter. Write a large large file at least 64 KiB (i.e. twice the maximum allowed buffer cache size) byte-by-byte. Then, read it in byte-by-byte. The total number of device writes should be on the order of 128 since 64 KiB is 128 blocks.
- Test your buffer cache's ability to write full blocks to disk without reading them first. If you are, for example, writing 100 KiB (200 blocks) to a file, your buffer cache should perform 200 calls to `block_write`, but 0 calls to `block_read` since exactly 200 blocks worth of data are being written. Read operations on inode metadata are still acceptable. As mentioned earlier, each block device keeps a `read_cnt` counter and a `write_cnt` counter. You can use this to verify that your buffer cache does not introduce unnecessary block reads. **If your buffer cache does not have this property, then implement the other two options listed above.**

You should focus on writing tests for general buffer cache features, rather than writing tests for your specific implementation of the buffer cache. You should write your test cases with a minimal set of assumptions about the underlying buffer cache implementation, but you are permitted to make as many basic assumptions about the buffer cache as you need to, since it is very difficult to write buffer cache tests without doing so. Use your best judgement and create test cases that could potentially be adapted to a different group's project without rewriting the whole thing. Once you finish writing your test cases, make sure that they get executed when you run `make check` in the `filesys/` directory.

3 Deliverables

3.1 Design

Before you start writing any code for your project, you need to create an design plan for each feature and convince yourself that your design is correct. You must **submit a design document** and **attend a design review** with your TA. This will help you solidify your understanding of the project and have a solid attack plan before tackling a large codebase.

3.1.1 Document

Like any techinical writing, your design document needs to be clean and well formatted. We've provided you with a template linked on the website that you must use. The template can be found on the website. We use Dropbox Paper[1] which supports real-time collaboration like Google Docs with the added benefit of techincal writing support (e.g. code blocks, LaTeX). **Not using this template or failure to use code formatting will result in losing points**. The main goal of this is not to punish you for this but rather make it easy for TAs to read.

To get started, navigate to the template and click the "Create doc" button on the top right hand corner. You can share this doc with other group members to collaborate on it. **Make sure to click the blue "Share" button in *your* document, not the template**.

For each task except for Concept Check, you must explain the following aspects of your proposed design. We suggest you create a section for each task which has subsections for each of the following aspects.

Data Structures and Functions

List any `struct` definitions, global or static variables, `typedef`s, or enumerations that you will be adding or modifying (if it already exists). These should be **written with C not pseudocode**. Include a **brief explanation** (i.e. a one line comment) of the purpose of each modification. A more in depth explanation should be left for the following sections.

Algorithms

Tell us how you plan on writing the necessary code. Your description should be at a level below the high level description of requirements given in the assignment. **Do not repeat anything that is already stated on the spec**.

On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. You may use *small snippets* of pseudocode or C in places you deem appropriate. Instead, you need to convince us that your design satisfies all the requirements, **especially any edge cases**. We expect you to have read through the Pintos source code when preparing your design document, and your design document should refer to the appropriate parts of the Pintos source code when necessary to clarify your implementation.

Synchronization

List all resources that are shared across threads and processes. For each resource, explain how the it is accessed (e.g. from an interrupt context) and describe your strategy to ensure it is shared and modified safely (i.e. no race conditions, deadlocks).

In general, the best synchronization strategies are simple and easily verifiable. If your synchronization strategy is difficult to explain, this is a good indication that you should simplify your strategy. Discuss the time and memory costs of your synchronization approach, and whether your strategy will significantly limit the concurrency of the kernel and/or user processes/threads. When discussing the concurrency allowed by your approach, explain how frequently threads will contend on the shared resources, and any limits on the number of threads that can enter independent critical sections at a single time. You should aim to avoid locking strategies that are overly coarse.

[1] http://paper.dropbox.com/

Interrupt handlers cannot acquire locks. If you need to access a synchronized variable from an interrupt handler, consider disabling interrupts. Locks do not prevent a thread from being preempted. Threads can be interrupted during a critical section. Locks only guarantee that the critical section is only entered by one thread at a time.

Do not forget to consider memory deallocation as a synchronization issue. If you want to use pointers to `struct thread`, then you need to prove those threads can't exit and be deallocated while you're using them.

If you create new functions, you should consider whether the function could be called in 2 threads at the same time. If your function access any global or static variables, you need to show that there are no synchronization issues.

Rationale Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

3.1.2 Review

After you submit your design doc, you will schedule a design review with your TA. A calendar signup link will be posted sometime before the design doc due date. During the design review, your TA will ask you questions about your design for the project. You should be prepared to defend your design and answer any clarifying questions your TA may have about your design document. The design review is also a good opportunity to get to know your TA for participation points.

3.1.3 Grading

The design document and design review will be graded together. Your score will reflect how convincing your design is, based on your explanation in your design document and your answers during the design review. If you cannot make a design review, please contact your TA to work out an arrangement. **An unexcused absence from a design review will result in a 0 for the design portion**.

3.2 Code

Code will be submitted to GitHub via your groupX repo. Pintos comes with a test suite that you can run locally on your VM. We will run the same tests on the autograder, meaning there are **no hidden tests**. As a result, we recommend you test locally as much as possible since the autograder's bandwidth is limited.

3.2.1 Checkpoints

On the autograder, we will provide checkpoints for you to stay on pace with the project. **Checkpoints will not be graded and have no effect on your grade**. However, we still encourage students to keep up with the checkpoints See **??** for more details on the specific checkpoints for this project.

Each checkpoint will have a corresponding autograder. Checkpoint 1 autograder will automatically run until its deadline, then Checkpoint 2 will automatically run until its deadline. The final autograder will not automatically run until after the Checkpoint 2 autograder. If you'd like, you can manually trigger the autograders as they'll be available throughout the entire project duration.

3.2.2 Testing

Your testing code needs to be included in your repo as well under the appropriate folder.

3.2.3 Quality

The score of your code will be mainly determined by your autograder score. However, you will also be graded on the quality of your code on some factors including but not limited to

- Does your code exhibit any major memory safety problems (especially regarding strings), memory leaks, poor error handling, or race conditions?
- Is your code simple and easy to understand? Does it follow a consistent naming convention?
- Did you add sufficient comments to explain complex portions of code?
- Did you leave commented-out code in your final submission?
- Did you copy and paste code instead of creating reusable functions?
- Did you re-implement linked list algorithms instead of using the provided list manipulation functions?
- Is your Git commit history full of binary files?

Note that you don't have to worry about manually enforcing code formatting (e.g. indentation, spacing, wrapping long lines, consistent placement of braces.) as long as you setup your precommit hook correctly in Project User Programs. You may also find it helpful to format code every once in a while by running `make format`.

3.3 Report

After you complete the code for your project, your group will write a report reflecting on the project. While you're not expected to write a massive report, we are asking for the same level of detail as expected for the design document. Your report should include the following sections.

Changes

Discuss any changes you made since your initial design document. Explain why you made those changes. Feel free to reiterate what you discussed with your TA during the design review if necessary.

Reflection

Discuss the contribution of each member. Make sure to be specific about the parts of each task each member worked on. Reflect on the overall working environment and discuss what went well and areas of improvement.

Testing

For each test case you write, provide

- Description of the feature your test case is supposed to test.
- Overview of how the mechanics of your test case work, as well as a *qualitative* description of the expected output.
- Output and results of your own Pintos kernel when you run the test case. These files will have the extensions `.output` and `.result`.
- Two non-trivial potential kernel bugs and how they would have affected the output of this test case. Express these in the form "If my kernel did X instead of Y, then the test case would output Z instead.". You should identify two different bugs per test case, but you can use the same bug for both of your two test cases. These bugs should be related to your test case (e.g. syntax errors don't

In addition, tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? What did you learn from writing test cases?

3.4 Evaluations

After you finish all the components above, you must submit an evaluation of your group members. You are will be given $20(n-1)$ points to distribute amongst your group members *not including yourself*, where n is the number of people in your group. For instance, if you have four team members, you will get 60 points to distribute amongst the rest of your group members. If you believe all group members contributed equally, you would give each member 20 points each.

You will also fill out the details of what each member worked on. While this is similar to Report portion of the report, this will serve as a space to truthfully speak about the contributions since the report is a collaborative document.

To submit evaluations, navigate to your `personal` repo and pull from the staff repo. You should see a folder for evaluations (e.g. `proj-userprog-eval` for Project User Programs) which contains a `evals.txt`. Fill out the `evals.txt` file. Each line should pertain to one group member, so make your comments for each group member remain in that line. The following is an example of what your csv should look like.

```
Name|Autograder ID (XXX)|Score|Comment
Jieun Lee|516|23|Worked on all tasks finished the testing report
Taeyeon Kim|309|20|Worked on tasks 2, 3, 4, helped come up with a lot of design ideas.
Sean Kim|327|17|Helped with task 1 and testing but didn't work on design doc.
```

Your comments should be longer and more descriptive than the ones given above. Ideally, aim to write between 100 and 200 words for each group member.

This evaluation will remain anonymous from the rest of your group members. If we notice some extreme weightings of scores, we will reach out and arrange a meeting to discuss any group issues. **These evaluations are important and hold substantial weight, so fill them out honestly and thoroughly**. See Grading for more information on how evaluations will be used.

3.5 Submission

Design documents and final reports should be submitted to Gradescope to their respective locations. You can export your document from Dropbox Paper by clicking on the three dots in the top right hand corner and clicking "Export".

Make sure you've pushed your code and have an autograder build. This build **must include the testing code** for you to receive credit on testing.

Make sure to push your evals to your individual repo to trigger the autograder. If you get errors from the autograder, make sure to check that you've done the following.

- Included exactly $n - 1$ members (i.e. no evals of yourself).
- Autograder IDs are correct and *not prefixed with student* (i.e. 162 instead of student162).
- The points sum to exactly $20(n - 1)$.
- No negative points for any member.
- Header row is present.
- Each line contains one evaluation. Your comment section should not consist of multiple lines.
- No extra lines at the end of the file.

Make sure to not change the header row.

3.6 Grading

The components above will be weighed as follows: 15% Design, 70% Code, 15% Report. While evaluations are not explicitly part of your grade, your project score will be impacted based on them. To keep evaluations fair and anonymous, we will not be releasing how evaluations are factored in nor the final scores calculated after factoring in evaluations.

Slip days or extensions cannot be applied on design documents. TAs need enough time to read these design documents to be able to hold design reviews in a timely manner. Slip days will be applied as a maximum across all the other parts. **Evaluation submission will be factored into slip day usage**. For instance, if code is submitted on time, report is submitted one hour late, three members submit their evaluations on time, and one member submits their evaluation two days late, the number of slip days used will be 2 days.

4 FAQ

How much code will I need to write?

Here's a summary of our reference solution. Note that the diff was generated relative to the staff solution for Project User Programs. The reference solution represents just one possible solution. Many other solutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution.

```
filesys/directory.c |   43 ++-
filesys/directory.h |    2
filesys/filesys.c   |  182 ++++++++++-
filesys/filesys.h   |    5
filesys/free-map.c  |   32 +-
filesys/fsutil.c    |    2
filesys/inode.c     |  435 ++++++++++++++++++++++++++------
filesys/inode.h     |    6
threads/thread.c    |    1
userprog/process.c  |    4
userprog/process.h  |   10
userprog/syscall.c  |  110 +++++++
userprog/syscall.h  |    5
13 files changed, 717 insertions(+), 120 deletions(-)
```

Can `BLOCK_SECTOR_SIZE` change?

No. `BLOCK_SECTOR_SIZE` is fixed at 512. For integrated drive electronic (IDE) disks, this value is a fixed property of the hardware. Other disks do not necessarily have a 512-byte sector, but for simplicity Pintos only supports those that do.

What is the largest file size that we are supposed to support?

The file system partition we create will be 8 MiB or smaller. However, individual files will have to be smaller than the partition to accommodate the metadata. You'll need to consider this when deciding your inode organization.

How should a file name like `a//b` be interpreted?

Multiple consecutive slashes are equivalent to a single slash, so this file name is the same as `a/b`.

How about a file name like `/../x`?

The root directory is its own parent, so it is equivalent to `/x/`.

How should a file name that ends in `/` be treated?

Most Unix systems allow a slash at the end of the name for a directory, and reject other names that end in slashes. We will allow this behavior, but you may also choose to simply reject a name that ends in a slash.

Can we keep a `struct inode_disk` inside `struct inode`?

The goal of the 64-block limit is to bound the amount of cached file system data. If you keep a block of disk data, whether file data or metadata, anywhere in kernel memory then you have to count it against the 64-block limit. The same rule applies to anything that's similar to a block of disk data, such as a `struct inode_disk` without the `length` or `sector_cnt` members.

That means you'll have to change the way the inode implementation accesses its corresponding on-disk inode right now since it currently just embeds a `struct inode_disk` in `struct inode` and reads the corresponding sector from disk when it's created. Keeping extra copies of inodes would subvert the 64-block limitation that we place on your cache.

You can store a pointer to inode data in `struct inode`, but if you do so you should carefully make sure that this does not limit your operating system to 64 simultaneously open files. You can also store

other information to help you find the inode when you need it. Similarly, you may store some metadata along with each of your 64 cache entries.

You can keep a cached copy of the free map permanently in memory if you like. It doesn't have to count against the cache size.

`byte_to_sector` in `filesys/inode.c` uses the `struct inode_disk` directly without first reading that sector from wherever it was in the storage hierarchy. This will no longer work. You will need to change `inode_byte_to_sector` to obtain the `struct inode_disk` from the cache before using it.

A File Systems

A.1 Source Files

`filesys/directory.c`
Manages the directory structure. In Pintos, directories are stored as files.

`filesys/file.c`
Performs file reads and writes by doing disk sector reads and writes.

`filesys/filesys.c`
Top-level interface to the file system.

`filesys/free-map.c`
Utilities for modifying the file system's free block map.

`filesys/fsutil.c`
Simple utilities for the file system that are accessible from the kernel command line.

`filesys/inode.c`
Manages the data structure representing the layout of a file's data on disk.

`lib/kernel/bitmap.c`
A bitmap data structure along with routines for reading and writing the bitmap to disk files.

A.2 Persistence Tests

Until now, each test was contained within one boot of Pintos. However, an important purpose of a file system is to ensure that data remains accessible from one boot to another. Thus, the file system tests invoke Pintos twice. During the second invocation, all the files and directories in the Pintos file system are combined into a single file (known as a tarball), which is then copied from the Pintos file system to the host (i.e. your development VM) file system.

The grading scripts check the file system's correctness based on the contents of the file copied out in the second run. This means that **you will not pass any of the extended file system tests labeled `*-persistence` until the file system is implemented well enough to support `tar`, the Pintos user program that produces the file that is copied out.** The `tar` program is fairly demanding as it requires both extensible file and subdirectory support), so this will take some work. Until then, you can ignore errors from `make check` regarding the extracted file system.

Incidentally, as you may have surmised, the file format used for copying out the file system contents is the standard Unix `tar` format. You can use the Unix `tar` program to examine them. The tar file for test `T` is named `T.tar`.

Step by Step Walkthrough

To debug persistence tests, you must first understand what is exactly going on. Let's use the `dir-mkdir` test as an example to walk through how a persistence tests works.

When you run `pintos-test dir-mkdir` from `filesys/` directory, you'll notice several commands that are run. **Keep in mind while you run the `pintos-test` commands from the `filesys/` directory, the following commands (if run individually) should be run in the `filesys/build/` directory.**

1. `> rm -f tmp.dsk`

   We make sure to delete any remaining disk files if by some off chance it was left behind by another test because all tests use the `tmp.dsk` filename.

2. `> pintos-mkdisk tmp.dsk --filesys-size=2`

This creates a new disk image called `tmp.dsk` initialized with room for two files initially. Any persistence tests will contain at least two files.

3. ```
   > pintos -v -k -T 60 --qemu  --disk=tmp.dsk \
     -p tests/filesys/extended/dir-mkdir -a dir-mkdir \
     -p tests/filesys/extended/tar -a tar \
     -- -q -f run dir-mkdir < /dev/null 2> \
     tests/filesys/extended/dir-mkdir.errors > tests/filesys/extended/dir-mkdir.output
   ```

   This boots up a simulator to run Pintos on. Other flags can be understood using pintos –help, but they're omitted for brevity.

   `--disk=tmp.dsk` uses the existing tmp.dsk we created in step 2 as the disk image for the simulator. `-p tests/filesys/extended/dir-mkdir -a dir-mkdir` and `-p tests/filesys/extended/tar -a tar` copy over `tests/filesys/extended/dir-mkdir` and `tests/filesys/extended/tar` from your machine to the simulator and names them `dir-mkdir` and `tar` respectively. `dir-mkdir` is the executable of the test (i.e. the code that will run), and `tar` is the executable for the tar program. These are the two files which space was allocated for back when creating the new disk image.

4. ```
   > pintos -v -k -T 60 --qemu --disk=tmp.dsk \
     -g fs.tar -a tests/filesys/extended/dir-mkdir.tar \
     -- -q  run 'tar fs.tar /' < /dev/null 2> \
     tests/filesys/extended/dir-mkdir-persistence.errors \
     > tests/filesys/extended/dir-mkdir-persistence.output
   ```

   This tests the persistence tests. We use the same `tmp.dsk` as the disk image.

   `-g fs.tar -a tests/filesys/extended/dir-mkdir.tar` will copy over `fs.tar` from the simulator to your machine and names it `dir-mkdir.tar` within the `filesys/build/tests/filesys/extended` directory. The `fs.tar` is created within the `fsutil.c` methods you may have encountered; it is the tar ball of the root directory after all the functionality of your test has run.

5. ```
   > rm -f tmp.dsk
   ```

   This deletes tmp.dsk so other tests will not use the same file.

6. ```
   > perl -I../.. ../../tests/filesys/extended/dir-mkdir.ck \
     tests/filesys/extended/dir-mkdir tests/filesys/extended/dir-mkdir.result
   ```

   This provides you with some test output of the entire test.

As a result, you can run each of these commands individually instead of running pintos-test all at once. In between, you may benefit from examining files after running each step, mainly the `tmp.dsk` and `dir-mkdir.tar`. While you could `tar dir-mkdir.tar` to examine this, you may run into issues if you have functionality issues in inode. To examine these files as binary or ASCII, you can use `hexedit` (you may need to install using `sudo apt install hexedit`) or the Hex Editor extension on VS Code[2].

Running Individual Persistence Tests

Here is a faster way of running persistence tests manually without running `make check` every time. Again, we'll use `dir-mkdir` test as an example.

1. Run `pintos-test` on the base test (i.e. without the `*-persistence` suffix) and make sure you pass it.

   ```
   > pintos-test dir-mkdir
   ```

2. Then run the following.

   ```
   > cd build
   > perl -I../.. ../../tests/filesys/extended/dir-mkdir-persistence.ck \
   ```

[2] https://marketplace.visualstudio.com/items?itemName=ms-vscode.hexeditor

```
 tests/filesys/extended/dir-mkdir-persistence \
tests/filesys/extended/dir-mkdir-persistence.result
```

Using GDB with Persistence Tests

Again, we'll use `dir-mkdir` test as an example.

1. Run the usual command to debug the base test (i.e. without the `*-persistence` suffix) associated with the persistence test.

   ```
   > PINTOS_DEBUG=1 pintos-test testname
   ```

2. Enter the following commands to run the base test to completion.

   ```
   (gdb) debugpintos
   (gdb) continue
   ```

   Once you have run the above, after a short while you should see the message

   ```
   Remote connection closed
   ```

   This signifies that the base test has finished running, and the persistence test is now ready to be debugged. To debug the persistence test, run `debugpintos` and proceed debugging as usual.

A.3 Buffer Cache

Modify the file system to keep a cache of file blocks. When a request is made to read or write a block, check to see if it is in the cache, and if so, use the cached data without going to disk. Otherwise, fetch the block from disk into the cache, evicting an older entry if necessary. **Your cache must be no greater than 64 sectors in size.**

You must implement a cache replacement algorithm that is at least as good as the clock algorithm. We encourage you to account for the generally greater value of metadata compared to data. You can experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses. Running Pintos from the `filesys/build` directory will cause a sum total of disk read and write operations to be printed to the console, right before the kernel shuts down.

When one thread is actively writing or reading data to/from a buffer cache block, you must make sure other threads are prevented from evicting that block. Analogously, during the eviction of a block from the cache, other threads should be prevented from accessing the block. If a block is currently being loaded into the cache, other threads need to be prevented from also loading it into a different cache entry. Moreover, other threads must not access a block before it is fully loaded.

You can keep a cached copy of the free map permanently in a special place in memory if you would like. It doesn't count against the 64 sector limit.

The provided inode code uses a "bounce buffer" allocated with `malloc` to translate the disk's sector-by-sector interface into the system call interface's byte-by-byte interface. You should get rid of these bounce buffers. Instead, copy data into and out of sectors in the buffer cache directly.

When data is written to the cache, it does not need to be written to disk immediately. You should keep dirty blocks in the cache and write them to disk when they are evicted and when the system shuts down. You will need to modify the `filesys_done` method for this.

If you only flush dirty blocks on eviction or shut down, your file system will be more fragile if a crash occurs. As an optional feature, you can also make your buffer cache periodically flush dirty cache blocks to disk. If you have a non-busy-waiting `timer_sleep` from Project threads working, this would be an excellent use for it. Otherwise, you may implement a less general functionality, but make sure that it does not exhibit busy-waiting.

As an optional feature, you can also implement read-ahead, that is, automatically fetch the next block of a file into the cache when one block of a file is read. Read-ahead is only really useful when done asynchronously. That means, if a process requests disk block 1 from the file, it should block until disk block 1 is read in, but once that read is complete, control should return to the process immediately. The read-ahead request for disk block 2 should be handled asynchronously, in the background.

A.4 Extensible Files

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation. It is possible that an n-block file cannot be allocated even though n blocks are free. **Eliminate this problem by modifying the on-disk inode structure.** In practice, this probably means using an index structure with direct, indirect, and doubly indirect blocks. You are welcome to choose a different scheme as long as you explain the rationale for it in your design documentation, and as long as it does not suffer from external fragmentation.

You can assume that the file system partition will not be larger than 8 MiB. You must support files as large as the partition (minus metadata). Each inode is stored in one disk sector, limiting the number of block pointers that it can contain. Supporting 8 MiB files will require you to implement doubly-indirect blocks.

An extent-based file can only grow if it is followed by empty space, but indexed inodes make file growth possible whenever free space is available. **Implement file growth.** In the basic file system, the file size is specified when the file is created. In most modern file systems, a file is initially created with size 0 and is then expanded every time a write is made off the end of the file. Your file system must allow this.

There should be no predetermined limit on the size of a file, except that a file cannot exceed the size of the file system (minus metadata). This also applies to the root directory file, which should now be allowed to expand beyond its initial limit of 16 files.

User programs are allowed to seek beyond the current end-of-file (EOF). The seek itself does not extend the file. Writing at a position past EOF extends the file to the position being written, and any gap between the previous EOF and the start of the `write` must be filled with literal byte zeros (not the value 0). A `read` starting from a position past EOF returns no bytes.

Writing far beyond EOF can cause many blocks to be entirely zero. Some file systems allocate and write real data blocks for these implicitly zeroed blocks. Other file systems do not allocate these blocks at all until they are explicitly written. The latter file systems are said to support sparse files. You may adopt either allocation strategy in your file system.

You are already familiar with handling memory exhaustion in C, by checking for a NULL return value from malloc. In this project, you will also need to handle disk space exhaustion. When your file system is unable to allocate new disk blocks, you must have a strategy to abort the current operation and rollback to a previous good state.

A.5 Subdirectories

Implement support for hierarchical directory trees. In the basic file system, all files live in a single directory. Modify this to allow directory entries to point to files or to other directories. Make sure that directories can expand beyond their original size just as any other file can.

The basic file system has a 14-character limit on file names. You may retain this limit for individual file name components, or may extend it. **You must allow full path names to be much longer than 14 characters.**

Maintain a separate current working directory for each process. At startup, set the file system root as the initial process's current directory. When one process starts another with the `exec` system call, the child process inherits its parent's current directory. After that, the two processes' current directories are independent, so that either changing its own current directory has no effect on the other. This is why, under Unix, the `cd` command is a shell built-in, not an external program. You must decide if a process is

allowed to delete a directory if it is the current working directory of a running process. The test suite will accept both yes and no, but in either case, you must make sure that new files cannot be created in deleted directories.

You will need to update your existing file descriptor table and its lookup strategies to support directories as well.

Update the existing system calls so that anywhere a file name is provided by the caller, an absolute or relative path name may be used. The directory separator character is forward slash (`/`). You must also support special file names `.` and `..`, which have the same meanings as they do in Unix.

Update the `open` system call so that it can also open directories. You **should not** support `read` or `write` on a file descriptor that corresponds to a directory. You will implement the `readdir` and `mkdir` syscalls for directories instead. You **should** support `close` on a directory, which just closes the directory.

Update the `remove` system call so that it can delete empty directories (other than the root) in addition to regular files. Directories may only be deleted if they do not contain any files or subdirectories (other than `.` and `..`). You may decide whether to allow deletion of a directory that is open by a process or in use as a process's current working directory. If it is allowed, then attempts to open files (including `.` and `..`) or create new files in a deleted directory must be disallowed.

Here is some code that will help you split a file system path into its components. It supports all of the features that are required by the tests. It is up to you to decide if and where and how to use it.

```
/* Extracts a file name part from *SRCP into PART, and updates *SRCP so that the
   next call will return the next file name part. Returns 1 if successful, 0 at
   end of string, -1 for a too-long file name part. */
static int get_next_part(char part[NAME_MAX + 1], const char** srcp) {
  const char* src = *srcp;
  char* dst = part;

  /* Skip leading slashes.  If it's all slashes, we're done. */
  while (*src == '/')
    src++;
  if (*src == '\0')
    return 0;

  /* Copy up to NAME_MAX character from SRC to DST.  Add null terminator. */
  while (*src != '/' && *src != '\0') {
    if (dst < part + NAME_MAX)
      *dst++ = *src;
    else
      return -1;
    src++;
  }
  *dst = '\0';

  /* Advance source pointer. */
  *srcp = src;
  return 1;
}
```

A.6 System Calls

`bool chdir(const char* dir)`

Changes the current working directory of the process to `dir`, which may be relative or absolute. Returns true if successful, false on failure.

`bool mkdir(const char* dir)`
Creates the directory named `dir`, which may be relative or absolute. Returns true if successful, false on failure.

Fails if `dir` already exists or if any directory name in `dir`, besides the last, does not already exist. That is, `mkdir("/a/b/c")` succeeds only if `/a/b` already exists and `/a/b/c` does not.

`bool readdir(int fd, char* name)`
Reads a directory entry from file descriptor `fd`, which must represent a directory. If successful, stores the null-terminated file name in `name`, which must have room for `READDIR_MAX_LEN + 1` bytes, and returns true. If no entries are left in the directory, returns false.

`"."` and `".."` entries should not be returned by `readdir`.

If the directory changes while it is open, then it is acceptable for some entries not to be read at all or to be read multiple times. Otherwise, each directory entry should be read once, in any order.

`READDIR\_MAX\_LEN` is defined in `lib/user/syscall.h`. If your file system supports longer file names than the basic file system, you should increase this value from the default of 14.

`bool isdir(int fd)`
Returns true if `fd` represents a directory, false if it represents an ordinary file.

`int inumber(int fd)`
Returns the inode number of the inode associated with `fd`, which may represent an ordinary file or a directory.

An inode number persistently identifies a file or directory. It is unique during the file's existence. In Pintos, the sector number of the inode is suitable for use as an inode number.

We have provided the `ls` and `mkdir` user programs, which are straightforward once the above syscalls are implemented. We have also provided `pwd`, which is not so straightforward. The `shell` program implements `cd` internally.

The `pintos extract` and `pintos append` commands should now accept full path names, assuming that the directories used in the paths have already been created. This should not require any significant extra effort on your part.

A.7 Synchronization

Your project code should always be thread-safe, but for the current project, you may not use a single global lock around the entire file system. It is fine to have a global lock around the buffer cache, but **you may *not* perform blocking I/O with the global lock held!** The key point is operations that are independent (e.g. operating on different files, or different parts of the same file) should be able to issue disk I/O operations concurrently, without one waiting for the other to complete. If Thread A and Thread B are performing independent operations, and Thread B is not blocked on I/O, then it's fine for Thread A to block waiting for Thread B. [3]

What does it mean for two operations to be independent? For this project, **operations are considered independent if they are acting on different disk sectors, and such operations should be allowed to execute concurrently.** If two operations are writing to the same sector or extending the same file, then they are *not* considered independent and you may serialize those operations to maintain data consistency. Concurrent reads are not required.

Here are some examples. Assume that we have have the following file descriptors.

[3] On a multicore system, it may be desirable to allow Thread A and Thread B to execute concurrently, even if neither is blocked on I/O, to better take advantage of multiple cores. For example, one may prefer to not have a global lock around the buffer cache so that multiple cores can scan the buffer cache at the same time. But, given that Pintos does not support multicore systems, we are not requiring this.

```
int notes = open("/my_files/notes.txt");
int test = open("/my_files/test.c");
```

`read(notes)` and `write(test)` should be allowed to run concurrently since they operate on two different files that are stored on different sectors. `read(notes)` and `write(notes)` need not be allowed to run concurrently since they operate on the same sector. Note that `open` starts at the beginning of the file, so they operate on sector 0. `read(notes)` and `read(notes)` need not be allowed to run concurrently since they read from the same sector.

This requirement applies to all tasks. If you added a global file system lock in Project User Programs, remember to remove it!

A.8 Buffer Cache Testing

You should add your two test cases to the `filesys/extended` test suite, which is included when you run `make check` from the filesys directory. All of the filesys and userprog tests are "user program" tests, which means that they are only allowed to interact with the kernel via system calls. **Since buffer cache information and block device statistics are NOT currently exposed to user programs, you must create new system calls to support your two new buffer cache tests.** You can create new system calls by modifying these files (and their associated header files):

lib/syscall-nr.h Defines the syscall numbers and symbolic constants. This file is used by both user programs and the kernel.

lib/user/syscall.c Syscall functions for user programs

userprog/syscall.c Syscall handler implementations

Some things to keep in mind while writing your test cases:

- User programs have access to a limited subset of the C standard library. You can find the user library in `lib/`.
- User programs cannot directly access variables in the kernel.
- User programs do not have access to `malloc`, since `brk` and `sbrk` are not implemented. User programs also have a limited stack size. If you need a large buffer, make it a static global variable.
- Pintos starts with 4MB of memory and the file system block device is 2MB by default. Don't use data structures or files that exceed these sizes.
- Your tests should use `msg()` instead of `printf()` (they have the same function signature).

You can add new test cases to the `filesys/extended` suite by modifying these files:

tests/filesys/extended/Make.tests Entry point for the `filesys/extended` test suite. You need to add the name of your test to the `raw_tests` variable, in order for the test suite to find it.

tests/filesys/extended/my-test-1.c This is the test code for your test (you are free to use whatever name you wish, "my-test-1" is just an example). Your test should define a function called `test_main`, which contains a user-level program. This is the main body of your test case, which should make syscalls and print output. Use the `msg()` function instead of `printf`.

tests/filesys/extended/my-test-1.ck Every test needs a .ck file, which is a Perl script that checks the output of the test program. If you are not familiar with Perl, don't worry! You can probably get through this part with some educated guessing. Your check script should use the subroutines that are defined in `tests/tests.pm`. At the end, call `pass` to print out the "PASS" message, which tells the Pintos test driver that your test passed.

tests/filesys/extended/my-test-1-persistence.ck Pintos expects a second .ck file for every `filesys/extended` test case. After each test case is run, the kernel is rebooted using the same file system disk image, then Pintos saves the entire file system to a tarball and exports it to the host machine. The

`*-persistence.ck` script checks that the tarball of the file system contains the correct structure and contents. **You do not need to do any checking in this file, if your test case does not require it.** However, you should call `pass` in this file anyway, to satisfy the Pintos testing framework.