CS 162 Project 3: `File Systems`

Initial Design Document Due:
Friday, November 20 2015
Code Due:
Monday, December 7, 2015
Student Testing Report and Final Report Due:
Wednesday, December 9, 2015

Contents

1 Getting Started

In Project 3, you will be adding features to the Pintos file system. This project is a continuation of the userprog code you implemented in Project 2. You should use your group's Project 2 code as a starting point. The autograder will run the userprog tests of Project 2 in addition to the Project 3 file system tests.

1.1 Source Files

In this project, you'll be working with a large number of files, primarily in the filesys directory. To help you understand all the code, we've selected some key files and described them below:

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

file.c Performs file reads and writes by doing disk sector reads and writes.

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

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

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

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.

All the basic functionality of a file system is already in the skeleton code, so that the file system is usable from the start, as you've seen in Project 2. However, the current file system has some severe limitations which you will remove.

2 Testing File System Persistence

Until now, each test invoked Pintos just once. However, an important purpose of a file system is to ensure that data remains accessible from one boot to another. Thus, the Project 3 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 (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 your project 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 (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`.

3 Requirements

3.1 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.

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 (modify the `filesys_done()` function to do 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 non-busy waiting `timer_sleep()` from the Project 1 working, this would be an excellent use for it. Otherwise, you may implement a less general facility, 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.

We recommend integrating the cache into your design early. In the past, many groups have tried to tack the cache onto a design late in the design process. This is very difficult.

3.2 Indexed and 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 (as does the extent-based file system we provide).

You can assume that the file system partition will not be larger than 8 MB. 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 MB 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 zeros. 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.

3.3 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 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.)

Update the existing system calls so that, anywhere a file name is provided by the caller, an absolute or relative path name may 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 fd that corresponds to a directory. (You will implement `readdir` and `mkdir` 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;
}
```

3.4 System Calls

Implement the following new system calls:

System Call: 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.

System Call: 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.

System Call: 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 .. 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.

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

System Call: 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 `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.

3.4.1 File System Synchronization

The provided file system requires external synchronization, that is, callers must ensure that only one thread can be running in the file system code at once. **Your submission must adopt a finer-grained synchronization strategy that does not require external synchronization.** To the extent possible, operations on independent entities should be independent, so that they do not need to wait on each other.

Operations on different cache blocks must be independent. In particular, when I/O is required on a particular block, operations on other blocks that do not require I/O should proceed without having to wait for the I/O to complete. **You may add fairness to your synchronization design if you want, but it is not required.**

Multiple processes must be able to access a single file at once. Multiple reads of a single file must be able to complete without waiting for one another. When writing to a file does not extend the file, multiple processes should also be able to write a single file at once. A read of a file by one process when the file is being written by another process is allowed to show that none, all, or part of the write has completed. (However, after the `write` system call returns to its caller, all subsequent readers must see the change.) Similarly, when two processes simultaneously write to the same part of a file, their data may be interleaved.

On the other hand, extending a file and writing data into the new section must be atomic. Suppose processes A and B both have a given file open and both are positioned at end-of-file. If A reads and B writes the file at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B's data is all nonzero bytes, A is not allowed to see any zeros.

Operations on different directories should take place concurrently. Operations on the same directory may wait for one another.

Keep in mind that only data shared by multiple threads needs to be synchronized. In the base file system, each thread has its own private copy of all of its `struct file` and `struct dir`, so access to those structs would not need to be synchronized.

3.5 Student Testing

This is a new task for Project 3. Your grade for this component will be assigned by a reader or a TA, instead of the autograder.

Pintos does not come with any tests for your buffer cache. Your task will be to create 2 tests for your buffer cache and to turn in a Student Testing Report along with your Final Report.

3.5.1 Testing the buffer cache

The buffer cache is difficult to test without knowing the underlying implementation of the buffer cache itself (known as "black box testing"). However, the buffer cache is a complex system and deserves good test coverage to make sure that all of its components work correctly. **You need to add two new test cases to the Pintos testing framework, which test non-trivial functionality in your buffer cache implementation.** Some examples of ideas you can choose for a test case are:

- Test your buffer cache's effectiveness by measuring its cache hit rate. First, reset the buffer cache. 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 file byte-by-byte (make the total file size at least 64KB, which is twice the maximum allowed buffer cache size). Then, read it in byte-by-byte. The total number of device writes should be on the order of 128 (because 64KB 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 100KB (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.
- Test your buffer cache's locking and synchronization. Create a large pool of threads, which all open the same file and write data to it. Since simultaneous writes do not need to be synchronized, the written data can appear interleaved in any form. However, you should check other cache properties: Does any disk sector appear in the cache twice? Are the control bits (valid, dirty, reference) correct?
- (If your buffer cache uses dynamic memory allocation) Test your buffer cache for potential memory leaks. Write a subroutine that reads and writes large files (bigger than the capacity of your buffer cache). Then, call it a thousand times, to see if the system runs out of memory.
- Come up with your own idea! If you do not choose your two test cases from this list, your own test idea should be similar in depth and scope.

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 good judgement, and create test cases that could potentially be adapted to a different group's project without rewriting the whole thing.

3.5.2 Pintos user program tests

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 test should use `msg()` instead of `printf()` (they have the same function signature).

3.5.3 How to add tests to Pintos

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/student-test-1.c This is the test code for your test (you are free to use whatever name you wish, "student-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/student-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/student-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.

3.5.4 Student Testing Report

After implementing your two test cases, you will prepare a Student Testing Report, which should be submitted in the `pintos/src` directory, alongside your Final Report.

Your student testing report should contain:

- For each of the 2 test cases you write:
  - Provide a description of the feature of the buffer cache your test case is supposed to test.
  - Provide an overview of how the mechanics of your test case work, as well as a qualitative description of the expected output. If you added any new syscalls to support your test case, you should tell us about those as well.
  - Provide the output of your own Pintos kernel when you run the test case. Please copy the full raw output file from `filesys/build/tests/filesys/extended/student-test-1.output` as well as the raw results from `filesys/build/tests/filesys/extended/student-test-1.result`.
  - Identify two non-trivial potential kernel bugs, and explain how they would have affected your output of this test case. You should express these in this form: "If your 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 2 test cases. These bugs should be related to your test case (e.g. "If your kernel had a syntax error, then this test case would not run." does not count).
- Tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? (There's a lot of room for improvement.) What did you learn from writing test cases?

4 Suggested Order of Implementation

To make your job easier, we suggest implementing the parts of this project in the following order. You should think about synchronization throughout all steps.

- Implement the buffer cache and integrate it into the existing file system. At this point all the tests from Project 2 should still pass.
- Extensible files. After this step, your project should pass the file growth tests.
- Subdirectories. Afterward, your project should pass the directory tests.
- Remaining miscellaneous items.
- You can implement extensible files and subdirectories in parallel if you temporarily make the number of entries in new directories fixed.

5 Schedule and Grading

The design you provide during your first week is considered to be a work-in-progress. We're expecting that you'll make some changes to your design during implementation and that you'll need to change your tests to accommodate interesting cases you discover while trying to make things work. That said, your design document needs to be a good-faith effort and reasonably resemble what you build, otherwise you haven't really done a design document.

5.1 Initial Design Document (10%)

Due: Friday, November 20 2015 at 11:59 pm A template will be posted on Piazza. Submit your initial design document by adding a text document in the pintos/src folder to your group Github repo on the master branch. Schedule a design review with your group's TA. This document will be graded on correctness.

5.2 Design Review (10%)

The sign-up sheet will be posted on Piazza. 5 points will be based entirely on whether or not you know what is in your own design document, and can have an intelligent conversation on why you made the design decisions you did. We will NOT be docking points in this section for inefficient or bad design. Another 5 points will be attributed to your ability to answer questions about how you might test certain features of the file system and cover any cases not already covered by the current set of tests. This will be based on correctness. If the explanation is dissatisfactory, we will be taking points away.

5.3 Final Code (55%)

Due: Monday, December 7 2015 at 11:59 pm

You will receive a grade from the autograder, which is based on the provided test suite. In addition to all the pintos tests in the filesys directory, all of the Pintos tests from Project 2 should pass for your final submission. If you do not pass any of the Project 2 tests, your submission will receive zero points on the autograder. Not all of the Project 2 tests are included (notably, multi-oom is absent).

5.4 Student Testing (15%)

Due: Wednesday, December 9 2015 at 11:59 pm

The autograder will not run your Student Tests. The readers and TA's will grade your student testing report manually. Both the test code and the Student Testing Report are due by this deadline. You should push both of these things to your master branch on GitHub.

- (15 points) - Completed test cases and satisfactory report
- (10 points) - Missing some components, but good effort
- (0 points) - No submission

5.5 Code Style (5%)

The following rubric will be used to grade your code style. The minimum score is 0 points and the maximum score is 10 points.

- (-2 points) Inconsistent or bad coding style: no indentation, cramming many statements into one line, other issues at TA's or reader's discretion
- (-2 points) Function(s) should be decomposed for clarity. Please don't exceed 75 lines for a single function.
- (-2 points) Commented-out code makes real code hard to read
- (-2 points) Cut-and-pasted code should be made into functions.
- (-2 points) Git - committing binaries
- (-1 points) Many missing comments on critical structs, struct members, global or static variables, or function definitions
- (-1 points) Numerous very long source code lines (> 100 characters)
- (-1 points) Using nested loops instead of provided list functions.
- (-1 points) Git - committing useless stuff

5.6 Final Report (5%)

Due: Wednesday, December 9 2015 at 11:59 pm

The final report is NOT a rehash of the initial design doc. You must include the following in your final report:

- the changes you made since your initial design doc and why you made them (feel free to re-iterate what you discussed with your TA in the design review)
- a reflection on the project - what went well, what could be improved?

Please do note that we are no longer requiring you to tell us who did what. That is what the anonymous, post-project, peer evaluations will be for.

This section will follow a ternary grading rubric:

- (10 points) - accurately documented the changes made since the initial design doc, provided reasoning behind design changes, and put effort into the reflection (i.e. answered the above questions).
- (5 points) - any of the sections are missing, didn't answer crucial questions, overall minimal effort.
- (0 points) - no submission.

In total, the final report should be around 1 page.

6 FAQ

The following questions have been frequently asked by students in the past.

How much code will I need to write? Here's a summary of our reference solution (including our code for Project 2), produced by the diffstat program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

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, and some may modify files not modified by the reference solution.

```
devices/timer.c      |   42 ++
filesys/cache.c      |  473 +++++++++++++++++++++++++
filesys/cache.h      |   23 +
filesys/directory.c  |   99 ++++-
filesys/directory.h  |    3
filesys/file.c       |    4
filesys/filesys.c    |  194 +++++++++-
filesys/filesys.h    |    5
filesys/free-map.c   |   45 +-
filesys/free-map.h   |    4
filesys/fsutil.c     |    8
filesys/inode.c      |  444 +++++++++++++++++++-----
filesys/inode.h      |   11
threads/init.c       |    5
threads/interrupt.c  |    2
threads/thread.c     |   32 +
threads/thread.h     |   38 +-
userprog/exception.c |   12
userprog/pagedir.c   |   10
userprog/process.c   |  332 ++++++++++++++----
userprog/syscall.c   |  582 +++++++++++++++++++++++++++++-
userprog/syscall.h   |    1
24 files changed, 2094 insertions(+), 286 deletions(-)
```

Can BLOCK_SECTOR_SIZE change? No, **BLOCK_SECTOR_SIZE** is fixed at 512. For 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 MB 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, as well as simply rejecting 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 OS 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 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.