CS 162 Project 3: `File Systems`

Design Document Due:	Sunday, April 12, 2020
Code Checkpoint #1:	Monday, April 20, 2020
Code Checkpoint #2:	Monday, April 27, 2020
Code Due:	Monday, May 4, 2020
Final Report Due:	Wednesday, May 6, 2020

Contents

1 Your task

In this project, you will add 3 new features to the Pintos file system. A brief summary of the tasks is provided here. A more detailed explanation can be found in section 3.4.

Important: This project requires a working implementation of Project 1. If you have not correctly implemented all of the functionality tests for Project 1, you should fix that first. Talk to your TA if you have trouble with this.

1.1 Task 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 acceptable. You are welcome to use one of the replacement policies discussed in lecture and/or section; 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.

1.2 Task 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 the Unix FFS file system. 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 out of disk space by leaving the file system in a consistent state (especially with regard to inode extension) and without leaking disk space or memory.

1.3 Task 3: Subdirectories

The current Pintos file system supports directories, but user programs have no way of using them (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`, and `isdir`. You must also update the following system calls so that they work with directories: `open`, `close`, `exec`, `remove`, and `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.

1.4 Synchronization Requirement

Your project code should always be thread-safe, but for Project 3, you may not use a single global lock around the entire file system. It's fine for your solution to have a global lock around I/O operations, but **you may *not* perform read and write operations across multiple sectors with the global lock enabled!** The key point is operations that are independent (e.g., operating on different sectors) should be able to issue disk I/O operations concurrently, without one waiting for the other to complete.

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 already executed `int notes = open("/my_files/notes.txt");` and `int test = open("/my_files/test.c");`:

1. `read(notes)` and `write(test)` should be allowed to run concurrently, since they operate on two different files that are stored on different sectors.
2. `read(notes)` and `write(notes)` aren't 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).
3. `read(notes)` and `read(notes)` may or may not be allowed to run concurrently, since they read from the same sector.

This requirement applies to all 3 tasks: the buffer cache, extensible files, and subdirectories.
Note: If you added a global file system lock in Project 1, remember to remove it!

2 Deliverables

Your project grade will be made up of 4 components:

- 15% Design Document and Design Review
- 60% Code
- 15% Student Testing
- 10% Final Report and Code Quality

2.1 Design Document (Due 04/12) and Design Review

Before you start writing any code for your project, you should create an implementation plan for each feature and convince yourself that your design is correct. For this project, you must **submit a design document** and **attend a design review** with your project TA.

2.1.1 Design Document Guidelines

Write your design document inside the `doc/project3.md` file, which has already been placed in your group's GitHub repository. You must use GitHub Flavored Markdown[1] to format your design doc. You can preview your design document on GitHub's web interface by going to the following address: (replace group0 with your group number)

`https://github.com/Berkeley-CS162/group0/blob/master/doc/project3.md`

For each of the 3 tasks of this project, you must explain the following 4 aspects of your proposed design. We suggest you create a section for each of the 3 project parts. Then, create subsections for each of these 4 aspects.

1. **Data structures and functions** – Write down any struct definitions, global (or static) variables, typedefs, or enumerations that you will be adding or modifying (if it already exists). These definitions should be written with the **C programming language**, not with pseudocode. Include a **brief explanation** the purpose of each modification. Your explanations should be as concise as possible. Leave the full explanation to the following sections.

2. **Algorithms** – This is where you tell us how your code will work. Your description should be at a level below the high level description of requirements given in the assignment. We have read the project spec too, so it is unnecessary to repeat or rephrase what is stated here. 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. Instead, you should try to convince us that your design satisfies all the requirements, **including any uncommon edge cases**. This section should be similar in style and format to the design document you submitted for Projects 1 and 2. We expect you to read through the Pintos source code when preparing your design document, and your design document should refer to the Pintos source when necessary to clarify your implementation.

3. **Synchronization** – This section should list all resources that are shared across threads. For each case, enumerate how the resources are accessed (e.g., from inside of the scheduler, in an interrupt context, etc), and describe the strategy you plan to use to ensure that these resources are shared and modified safely. For each resource, demonstrate that your design ensures correct behavior and avoids deadlock. 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

[1] https://help.github.com/articles/basic-writing-and-formatting-syntax/

simplify your strategy. Please discuss the time/memory costs of your synchronization approach, and whether your strategy will significantly limit the parallelism of the kernel. When discussing the parallelism 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.

4. **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.

2.1.2 Topics for your Design Document

Make sure to address each of these issues in the **Algorithms** and **Synchronization** sections of your design document. You do not need to answer these questions directly, but your design document should clearly demonstrate that your design will not exhibit any of these problems.

- When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?
- During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?
- If a block is currently being loaded into the cache, how are other processes prevented from also loading it into a different cache entry? How are other processes prevented from accessing the block before it is fully loaded?
- How will your file system take a relative path like `../my_files/notes.txt` and locate the corresponding directory? Also, how will you locate absolute paths like `/cs162/solutions.md`?
- Will a user process be allowed to delete a directory if it is the cwd 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.
- How will your system call handlers take a file descriptor, like `3`, and locate the corresponding file or directory struct?
- 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.

2.1.3 Design Document Additional Questions

You must also answer these additional questions in your design document:

1. For this project, 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 write-behind and a strategy for read-ahead. **You must answer this question regardless of whether you actually decide to implement these features.**

2.1.4 Design Review

You will schedule a 20 minute design review with your project TA. 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 those participation points.

2.1.5 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. You **must** attend a design review in order to get these points. We will try to accommodate any time conflicts, but you should let your TA know as soon as possible.

2.2 Code (Due 04/20, 04/27, 05/04)

The code section of your grade will be determined by your autograder score. Pintos comes with a test suite that you can run locally on your VM. We run the same tests on the autograder. The results of these tests will determine your code score.

You can check your current grade for the code portion at any time by logging in to the course autograder. Autograder results will also be emailed to you.

We will also manually look at your code to make sure your implementation includes a buffer cache and meets the synchronization requirements. If we find that you did not implement a buffer cache, for example, we will adjust your score accordingly.

2.3 Checkpoint #1 (Due 04/20)

For the first checkpoint, implement Task 1: Buffer Cache and integrate the buffer cache into the existing file system. At this point all the tests from Project 1 should still pass, but no additional tests for Project 3 will pass.

2.4 Checkpoint #2 (Due 04/27)

For the second checkpoint, implement Task 2: Extensible Files, using your design for extensible and large files. Don't forget to make use of your buffer cache, and don't forget to handle synchronization. Both of these are best handled from the start, as they can be difficult to add later on. Keep in mind that many latent bugs in Task 1: Buffer Cache and Task 2: Extensible Files, such as those relating to persistence and handling allocation failures, may not surface until you implement Task 3: Subdirectories. Be sure to budget time to debug those issues should they arise.

2.5 Final Code (Due 05/04)

By the final code due date, you must have implemented all three tasks: Task 1: Buffer Cache, Task 2: Extensible Files, and Task 3: Subdirectories. Additionally, complete Student Testing Code (see below).

2.5.1 Student Testing Code

Pintos already contains a test suite for Project 3, 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. 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.

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.

Once you finish writing your test cases, make sure that they get executed when you run "`make check`" in the `pintos/src/filesys/` directory.

2.6 Final Report (Due 05/06) and Code Quality

After you complete the code for your project, you will submit a final report. Write your final report in the `reports/project3.md` file, which has already been placed in your group's GitHub repository. Please include the following in your final report:

- The changes you made since your initial design document 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 exactly did each member do? What went well, and what could be improved?
- Your Student Testing Report (see the previous section for more details)

You will also be graded on the quality of your code. This will be based on many factors:

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
- Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.
- Is your code simple and easy to understand?
- If you have very complex sections of code in your solution, did you add enough comments to explain them?
- Did you leave commented-out code in your final submission?
- Did you copy-paste code instead of creating reusable functions?
- Did you re-implement linked list algorithms instead of using the provided list manipulation
- Are your lines of source code excessively long? (more than 100 characters)
- Is your Git commit history full of binary files? (don't commit object files or log files, unless you actually intend to)

2.6.1 Student Testing Report (Due 05/06)

You will need to prepare a Student Testing Report, which will help us grade your test cases. Place your Student Testing Report in your final report, in the `reports/project3.md` file which is already placed in your group directory.

Make sure your Student Testing Report contains the following:

- For each of the 2 test cases you write:
  - Provide a description of the feature 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.
  - 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/your-test-1.output` as well as the raw results from `filesys/build/tests/filesys/extended/your-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 two 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?

We will grade your test cases based on effort. If all of the above components are present in your Student Testing Report and your test cases are satisfactory, you will get full credit on this part of the project.

3 Reference

3.1 Getting Started

This project is a continuation of the userprog code you implemented in Project 1. You should use your group's Project 1 code as a starting point. You must implement Project 3 in a way that does not break your Project 1. The autograder will run some of the tests for Project 1 in addition to the Project 3 file system tests.

3.2 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 1. However, the current file system has some severe limitations which you will removein this project.

3.3 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`.

3.4 Requirements

3.4.1 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 Project 2 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.

3.4.2 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 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 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.

3.4.3 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;
}
```

3.4.4 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.

3.5 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).

3.6 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/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.

3.7 FAQ

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

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 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, 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 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 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.