CIS 520 - Fall 2015 Pintos Project #4 – File Systems

(Adapted from the Pintos Project by Ben Pfaff)

Now that you've enabled virtual memory, it's time to provide a file system. The base version of the code for this project solves the problems in the first three projects and provides the groundwork for this project. It is available in the public directory as /pub/CIS520/pintos4.tar.gz. You can start with your own fully-working copy of a solution to the first three projects and just use the newly posted version for inspiration, or you can just start with pintos4.tar.gz.

DUE: Upload via Canvas no later than 11:59 pm on Sunday, Dec. 6th, 2015

TO DO: Upload a gzipped tape archive file called Proj4.tar.gz that includes:

- A design document (/pub/CIS520/Design4.txt) in the pintos4/src/filesys directory.
- All of your modified Pintos source code; e.g., use the commands:
  - **`tar cvf Proj4.tar pintos4`** and
  - **`gzip Proj4.tar`** to gzip the tar file.

In the previous assignments, you made extensive use of a file system without actually worrying about how it was implemented underneath. For this last assignment, you will improve the implementation of the file system. You will be working primarily in the **filesys** directory.

You may build project 4 on top of project 2 or project 3. In either case, all of the functionality needed for project 2 must work in your filesys submission. If you build on project 3, then all of the project 3 functionality must work also. You will need to edit filesys/Make.vars to disable VM functionality. You can receive up to 5% extra credit if you do enable VM (the default).

5.1 Background

5.1.1 New Code

Here are some files that are probably new to you. These are in the filesys directory except where indicated:

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

filesys.h
filesys.c
: Top-level interface to the file system. For additional information, see Section 3.1.2 Using the File System in pintos.pdf, for an introduction.

directory.h
directory.c
: Translates file names to inodes. The directory data structure is stored as a file.

inode.h
inode.c
: Manages the data structure representing the layout of a file's data on disk.

file.h
file.c
: Translates file reads and writes to disk sector reads and writes.

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

Our file system has a Unix-like interface, so you may also wish to read the Unix man pages for creat, open, close, read, write, lseek, and unlink. Our file system has calls that are similar, but not identical, to these. The file system translates these calls into disk operations.

All the basic functionality is there in the code above, so that the file system is usable from the start, as you've seen in the previous two projects. However, it has severe limitations which you will remove.

While most of your work will be in filesys, you should be prepared for interactions with all previous parts.

5.1.2 Testing File System Persistence

By now, you should be familiar with the basic process of running the Pintos tests. See Section 1.2.1 Testing in pintos.pdf, for review, if necessary.

Until now, each test invoked Pintos just once. However, an important purpose of a file system is to ensure that **data persists**; that is, it remains accessible from one boot to another. Thus, the tests that are part of the file system project invoke Pintos a second time. The second run combines all the files and directories in the file system into a single file, then copies that file out of the Pintos file system into the host (Unix) 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 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 and time consuming (it requires both extensible file and subdirectory support), so, due to our limited time, this has been completed.

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.

5.2 Suggested Order of Implementation

To make your job easier, we suggest implementing the parts of this project in the following order (some of these may have been completed in your Proj2/Proj3 tar files):

1. Buffer cache (see Section 5.3.4 Buffer Cache). Implement the buffer cache and integrate it into the existing file system. At this point all the tests from project 2 (and project 3, if you're building on it) should still pass.

2. Extensible files (see Section 5.3.2 Indexed and Extensible Files). After this step, your project should pass the file growth tests.
3. Subdirectories (see Section 5.3.3 Subdirectories). Afterward, your project should pass the directory tests.
4. Remaining miscellaneous items.

You can implement extensible files and subdirectories in parallel if you temporarily make the number of entries in new directories fixed.

You should think about synchronization throughout.

5.3 Requirements

5.3.1 Design Document

Before you turn in your project, you must copy the project 4 design document template into your source tree under the name pintos/src/filesys/Design4.txt and fill it in. We recommend that you read the design document template before you start working on the project. See Section D - Project Documentation, for a sample design document that goes along with a fictitious project.

5.3.2 Indexed and Extensible Files

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation, that is, 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.

5.3.3 Subdirectories

Implement a hierarchical name space. 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, at your option. 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 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.

The open system call has been updated so that it can also open directories. Of the existing system calls, only close needs to accept a file descriptor for a 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 should not allow deletion of a directory that is open by a process or in use as a process's current working directory; that is, don't allow the current working directory '.' or parent '..' to be deleted.**

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*) [DONE]

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*) [DONE]

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. [DONE]

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. [Extend to also work for directories]. Hint: the function file_get_inode() returns the inode number associated with a given file, and the function dir_get_inode() returns the inode number associated with 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 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.

5.3.4 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. You are limited to a cache 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. Experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses.

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.

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.

Your cache should be *write-behind*, that is, keep dirty blocks in the cache, instead of immediately writing modified data to disk. Write dirty blocks to disk whenever they are evicted. Because write-behind makes your file system more fragile in the face of crashes, in addition you should periodically write all dirty,

cached blocks back to disk. The cache should also be written back to disk in filesys_done(), so that halting Pintos flushes the cache.

If you have timer_sleep() from the first project working, write-behind is an excellent application. Otherwise, you may implement a less general facility, but make sure that it does not exhibit busy-waiting.

You should 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, in case that block is about to be 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. These groups have often turned in projects that failed most or all of the tests.

5.3.5 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.

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, struct file and struct dir are accessed only by a single thread.

5.4 FAQ

How much code will I need to write?

The only files that need to be modified are userprog/syscall.c and filesys/directory.c. See references to // ADD CODE HERE

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.

5.4.1 Indexed Files FAQ

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.

5.4.2 Subdirectories FAQ

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.

5.4.3 Buffer Cache FAQ

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.