CS 302

OPERATING SYSTEM

SPRING 2019

Project 3: File System

Code Due:	May 31, 2019
Final Report Due:	May 31, 2019

DBGroup@SUSTech

April 29, 2019

Contents

1 Your task 2
1.1 Task 1: Buffer cache . . . 2
1.2 Task 2: Extensible files . . . 2
1.3 Task 3: Subdirectories . . . 2

2 Grading Policy 3
2.1 Code (Due 5/31) . . . 3
2.2 Final Report (Due 5/31) and Code Quality . . . 3

3 Submission 4
3.1 What to submit . . . 4
3.2 Where to submit and ddl . . . 4
3.3 Remarks . . . 4

4 Reference 5
4.1 Getting Started . . . 5
4.1.1 Source Files . . . 5
4.1.2 Testing File System Persistence . . . 6
4.1.3 Pintos user program tests . . . 6
4.1.4 How to add tests to Pintos . . . 7
4.1.5 Suggested Order of Implementation . . . 8
4.2 Requirements . . . 8
4.2.1 Buffer Cache . . . 8
4.2.2 Indexed and Extensible Files . . . 9
4.2.3 Subdirectories . . . 9
4.2.4 System Calls . . . 11
4.3 FAQ . . . 12

5 Acknowledgment 13

1 Your task

Now that you've worked with Pintos and are becoming familiar with its infrastructure and thread package, it's time to start working on the parts of the system that allow running user programs. The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this project, you will enable programs to interact with the OS via system calls.

You will be working out of the userprog directory for this assignment, but you will also be interacting with almost every other part of Pintos. We will describe the relevant parts below.

1.1 Task 1: Buffer cache

The functions "`inode_read_at()`" and "`inode_write_at()`" currently access the filesystem's underlying block device directly, each time you call them. Your task is to add a buffer cache for the filesystem, 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 a block replacement policy, but it must be at least as good as the clock algorithm. 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 filesystem allocates each file as a single contiguous set of blocks. Your task is to modify the Pintos filesystem to support extending files. You may want to use an indexed inode structure with direct, indirect, and doubly-indirect pointers, like the Unix file system does. The maximum file size you need to support is 8MiB (223 bytes). You must also add support for a new syscall, **"inumber(int fd)"**, which returns the unique inode number of file associated with a particular file descriptor.

1.3 Task 3: Subdirectories

The current Pintos filesystem 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 syscalls to allow user programs to manipulate directories: `chdir, mkdir, readdir, and isdir/`. You must also update the following syscalls 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.

2 Grading Policy

Your project grade will be made up of 2 components, with 100% in total:

- **60%** — Code
- **40%** — Final Report and Code Quality

Project 3 is team project, at most two students per group, but you must have different team members in project 2 and 3.

2.1 Code (Due 5/31)

The code section of your grade will be determined by your score in our final test. Pintos comes with a test suite that you can run locally on your VM and we run the same tests in the final test. Your marks will be a weighted one, after considering your rank in your class.

2.2 Final Report (Due 5/31) and Code Quality

After you complete the code for your project, you will submit a final report. Please include the following in your final report:

- 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 filesystem take a relative path like **../my_files/notes.txt** and locate the corre- sponding 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 syscall 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 filesystem is unable to allocate new disk blocks, you must have a strategy to abort the current operation and rollback to a previous good state.

3 Submission

3.1 What to submit

A team submits only one zip file, which can be submitted by any member of the team.

The zip file should:

- Create a folder named as **{student1number-student2number}-project3**.
  Example: 1234-2345-project3
  - A folder named **{student1number-student2number}-pintos** for your pintos source code.
    Example: 1234-2345-pintos
  - A pdf file named as **{student1number-student2number}-report.pdf** for your final report.
    Example: 1234-2345-report.pdf
- Zip the folder and name your zip: The zip should be named as **{student1number-student2number}-project3.zip**.
  Example: 1234-2345-project3.zip

If your files does not fit the naming schema, no points will be given!

3.2 Where to submit and ddl

Upload your pdf and zip file via Blackboard.
The pdf file should be submitted no later than **23:00 (Beijing time) 31 May, 2019**.
The zip file should be submitted no later than **23:00 (Beijing time) 31 May, 2019**.

3.3 Remarks

You should read carefully all requirements of this project.

Prohibition
You will get 0 as score for this project if you break any of the prohibitions:

- You should not Plagiarism.

- You should give clear citation when you use others ideas.
- You should follow the naming policy.
- You should give a clear report for others to read.
- Your code should be well documented for others to understand.

Contact
For any question regarding this project, please email to *11510237@mail.sustc.edu.cn (Boris)* or *11510213@mail.sustc.edu.cn (Yue Leng)*. The subject of the email should respect the format: **[OS] Project 3 (LastName FirstName-StudentNumber)**.
Example: [OS] Project 3 (Boris Chen-12345678)

4 Reference

4.1 Getting Started

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 some of the userprog tests of Project 2 in addition to the Project 3 file system tests.

4.1.1 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.
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.

4.1.2 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**.

4.1.3 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).

4.1.4 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.

4.1.5 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.

4.2 Requirements

4.2.1 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.

4.2.2 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.

4.2.3 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;
}
```

4.2.4 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.

4.3 FAQ

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 512byte 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.

You can still refer to previous document (Project 1, 2) to find out other details about pintos.

5 Acknowledgment

This project is originally designed by Stanford for their operating System course. Our project materials are based on the released guide for UC Berkeley CS 162 Operating System. We check the project materials from both schools and combine useful information together to form this one. You can find the original version on the Internet.