Project 4

Andrew Quinn

Due: Friday June 13th, 2024 at 11:59 PM (n.b., you *can* use grace days, but I *must* have your submission by June 16th at 11:59 PM.))

Learning Objectives:
1. Deepen your understanding of file systems.
2. Practice managing complexity in a large system.
3. Practice explaining and justifying computer system designs.

OVERVIEW

The goal of this assignment is to extend file system support in PintOS. This project builds on Project 2—so bugs from that project are likely to crop up again. You can build on your solution to Project 3 if you would like, but we do not test the Project 3 functionality.

REQUIREMENTS

Your submission should include a design document, extensible files support, subdirectory support, and working directory support. You can include a buffer cache to receive extra credit. Our advice: don't attempt the extra credit *before* you have finished the base points.

Like always: this project will be completed in partnerships, but you will each submit individually on canvas. You will each need to have your group's source code in your individual CSE 134 repository. See past projects for guidance on how to setup your repositories.

DESIGN DOCUMENT

Your submission should include a design document that describes key design decisions that you made in your system. The document should be located at `docs/p04.md`. Your design document should include a separate section for each of the other tasks. For each section, you should outline (1) any data structures that you created or extended in your design; (2) any algorithms that you created for your design; (3) any synchronization used in your design; (4) [*new*] any changes that you had to make to your solution from project 2; and (5) a justification of your design (Why is it correct? Why is it fast? etc.). You should aim to have enough detail in your design that a fellow 134 student would be capable of re-implementing your system by following it. Note: specify "N/A" if your design does not require any of these features.

```
create("foo.txt", 5);
int fd = open("foo.txt");
lseek(fd, 10);
int size1 = filesize(fd);
write(fd, "hello", 5);
int size2 = filesize(fd);
printf("%d, %d\n", size1, size2);
```

Figure 1: A simple program.

EXTENSIBLE FILES

The current PintOS file system allocates files when created using extents (i.e., contiguous block ranges). This approach does not allow the files to grow *and* causes external fragmentation. Your job is to fix this.

Your file system should allow files to grow and should limit external fragmentation. The system should initially allocate a file's size based upon the `size` argument passed to `create`. However, if a user program writes past the current end of a file, the file system should extend the file's size for the new data. You can assume that the file system partition will be at most 8 MiB—your solution should support files fill this whole space (i.e., files that are 8 MiB).

Note that extending a file should occur when the file is written to, not when a program seeks past the end of the file. So, fig. 1 prints the line `5, 15`. File data that has been "seeked" over should be 0 bytes. So, the value of `"foo.txt"` after executing fig. 1 is ten null terminator bytes (i.e., ASCII 0) followed by "hello".

SUBDIRECTORIES

The current PintOS file system only supports a single root directory. Add support for a hierarchical directory structure; you should use UNIX's directory separator (i.e., `/`). Each directory should be able to store any number of files or subdirectories. Each directory and file name should include at most `READDIR_MAX_LEN` (currently 14, defined in `lib/user/syscall.h`) bytes, but the full absolute path to a file should support much larger path lengths.

The `open` system call should support opening a directory. The `close` system call should add support for closing a directory. The `remove` system call should support deleting *empty* directories (other than the root) in addition to regular files; `remove` should fail if the directory is not empty (i.e., it contains anything other than `.` and `..`). You can allow a user to delete a directory that is currently in use (e.g., one that is open in another process); if you allow it, then you should prevent future attempts to create or open files in it. None of the other system calls need to support directory file descriptors (e.g., read does not need to read from a directory).

PintOS should add the following system calls to provide subdirectory support:

```
bool mkdir (const char *dir)
```

Creates the directory, `dir`, returning true if successful and false otherwise. Should fail if `dir` already exists or if any subdirectory, besides the last, does not already exist. For example: `mkdir("a/b/c")` works when `a/b` already exists, but `a/b/c` does not.

```
bool readdir (int fd, char *name)
```

Reads a directory entry from `fd`, a file descriptor obtained from calling `open` on a directory. On success, return true and store the null-terminated file name in `name`. Return false if there are no entries are left in the directory. Your solution can allow duplicate (or missing) directory entries in the event that a directory changes while it is open. When there is no concurrency, each directory entry should be read once. Note: `.` and `..` should not be returned by `readdir`.

```
bool isdir (int fd)
```

Returns true if `fd` represents a directory, i.e., was obtained by calling `open` on a directory. Otherwise, returns false.

```
int inumber (int fd)
```

Returns the inode number of the inode associated with `fd`, which may represent a file or a directory. An inode number persistently identifies a file or directory and is unique during the file's existence. In PintOS, the sector number of the inode is suitable for use as an inode number.

WORKING DIRECTORIES

Now that PintOS supports subdirectories, add current working directory support to PintOS. Each process should have a separate current directory for each process. The initial process should have a current directories as root (i.e., `/`). Child processes should inherit their parent's working directory. But, changing the directory in a child *should not* affect the parent's directory. You should extend all system calls that take a path as input to support relative and absolute paths. Your system should provide support for UNIX's interpretation of the special names `.` and `..`.

Finally, your system should provide the following system call to support working directories.

```
bool chdir (const char *dir)
```

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

EXTRA CREDIT: BUFFER CACHE

If you are felling ambitious: cache file blocks. Each read or write to a file block should first check the cache. If it is found, the operation should not interact with the disk. Otherwise, it should fetch the requested block into the cache, evicting an existing block if necessary. The cache should be a write-back cache, as opposed to write-through. It should flush on `filesys_done`; you can decide to flush more frequently if you would like. Your cache should only use 128 sectors and should replace all other caching in the file system (i.e., the "bounce" buffer that `inode_{readat|writeat}` currently use).

RUBRIC

We will use the following rubric for this assignment:

Category	Percentage
Testing	60%
Design	40%
Extra Credit	10%

TESTING. We will use the provided tests for testing you assignment. You should have 120 tests. If you see more than that, then you should update your repository from the course upstream (by clicking "update fork" on gitUCSC). This will account for 60% of your grade. You can run these tests by executing the following from the `src/vm/build` directory (after you have run `make` from the `src/vm` directory):

```
make check
```

To see the tests as they will be weighted for the final score, execute:

```
make grade
```

DESIGN. We will evaluate your design document based upon the following criteria:

1. **Sufficient:** (30%) Does your design document describe the system with sufficient detail as to be re-creatable by an engineer?
2. **Accurate:** (30%) Does your design document accurately describe the design that you implemented?
3. **Correctness:** (30%) Would your proposed design, assuming it were implemented correctly, satisfy the requirements of the assignment?
4. **Simplicity:** (10%) Is your design simple, rather than overly complex?

EXTRA CREDIT. We will grade extra credit by hand—there are no tests that pass with a buffer cache but fail without.

HINTS

There is a fair amount to implement for this project. We suggest implementing the parts of the project *in order*: extensible files, then subdirectories, and then working directories. That said, you would do well to “think ahead” when implementing subdirectories to ensure that adding relative paths does not drastically disrupt your design.

Check in with what tests you pass as you go. You should currently have a solution that passes a number of the tests that we’ll run. As you add new features, make sure that you don’t introduce “regression” bugs—i.e., bugs where you fail tests that you once passed.

If you would like to build your project using the code from project 3, edit `src/filesys/Make.vars` and uncomment the final two lines.