FileSystem(Buffer Cache)

Introduction

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

Background

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

fileName	function
‘fsutil.c’	Simple utilities for the file system that are accessible from the kernel command line.
‘filesys.h/c’	Top-level interface to the file system.
‘directory.h/c’	Translates file names to inodes. The directory data structure is stored as a file.
‘inode.h/c’	Manages the data structure representing the layout of a file’s data on disk.
‘file.h/c’	Translates file reads and writes to disk sector reads and writes.
‘lib/kernel/bitmap.h/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.

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

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 meta- data 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 think about synchronization throughout.

Requirement

1. Implement a cache replacement algorithm for buffer cache.
2. Implement write-behind.
3. You should download and **fill** the **DESIGNDOC** and archive your code(**the whole pintos folder**) and DESIGNDOC into a **zip** file **named sid*nameInPinYin*vid.zip(e.g 12330441*zhangsan*v0.zip)** and upload to the ftp.