CS140 HW4: File Systems

March 2, 2018

Main Tasks

- Buffer Cache.
- Indexed and Extensible Files.
- De-Coarsening File System Synchronization.
- New syscalls.

Getting Started

- You may choose to base this project off of either project 2 or project 3.
- If you base off of project 3, you can opt for some extra credit by enabling vm testing in `filesys/Make.vars`:

```
#Uncomment the lines below to enable VM.
#kernel.bin: DEFINES += -DVM
#KERNEL_SUBDIRS += vm
#TEST_SUBDIRS += tests/vm
#GRADING_FILE = $(SRCDIR)/tests/filesys/Grading.with-vm
```

  To build off project 3, but without enabling extra-credit grading, just uncomment the first two lines.

Disk Abstraction

- Can think of disk as a sequence of numbered sectors, each of which is `BLOCK_SECTOR_SIZE` bytes long.
- Read/write individual sectors via `block_read` and `block_write` in `devices/block.h`.

Buffer Cache

Buffer Cache

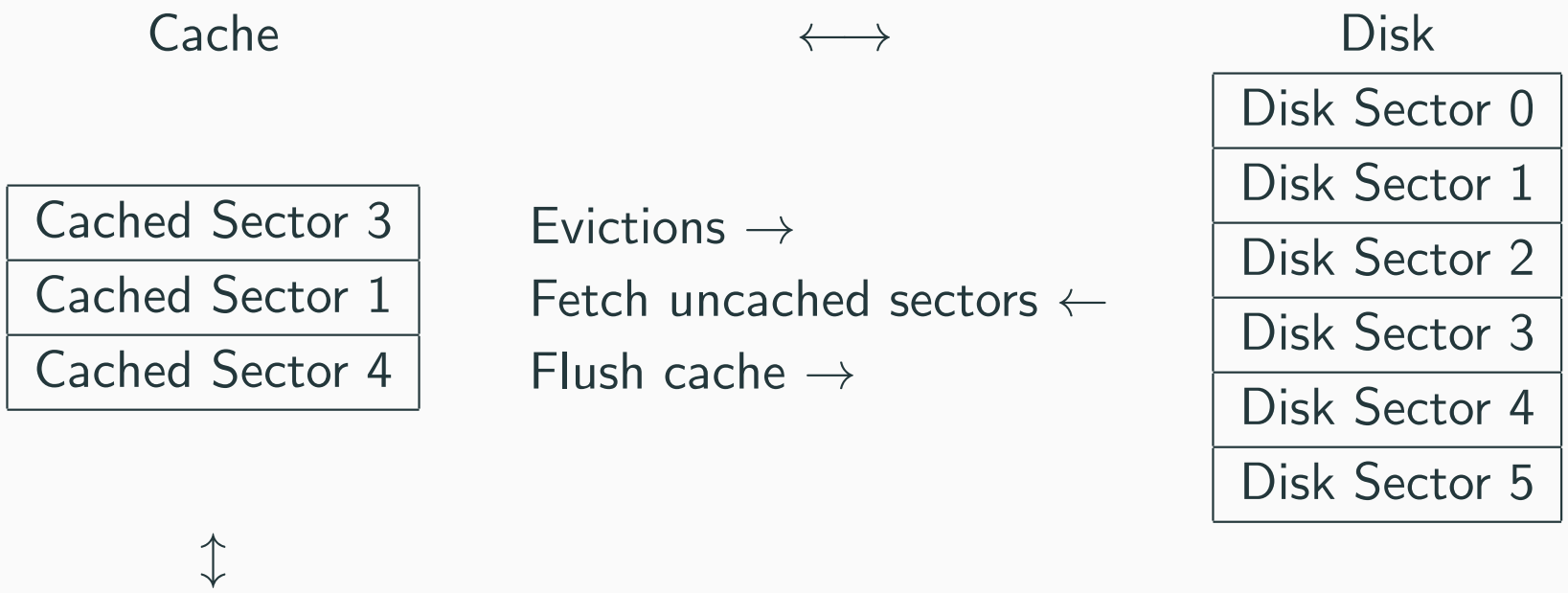
Current Setup

`file_write` calls `block_write` (by way of `inode_write_at`).

Buffer Cache Behavior

`file_write` calls a `cached_block_write` (or equivalent design).
Should *decouple* timings of file system operations from disk operations as much as possible. Hence the name "buffer".

Buffer Cache Diagram



Buffer Cache Requirements

- Write-behind: don't immediately write dirty cached sectors to disk (remember we want to decouple disk operations from file system operations).
  - Write to disk on cache-eviction.
  - Periodically flush cache (e.g. write all dirty cached sectors to disk every 30 seconds - perhaps a good time for a new kernel thread and `timer_sleep`!).
- Read-ahead: when you read sector N into the cache, also read in sector $N + 1$.
  - Return control to calling thread as soon as sector N has been read in - read in sector $N + 1$ asynchronously (need a different thread!)
- Eviction algorithm should approximate LRU at least as well as the clock algorithm.

Indexed and Extensible Files

Indexed and Extensible Files

- Indexed: avoid fragmentation by allowing file data to be scattered over the disk rather than limited to a contiguous range.
- Extensible: allow file sizes to change after file creation.

Indexed and Extensible Files: Inodes

- An inode records which sectors on disk store the data for a file.
- Inodes are stored on disk themselves, so how do we find them?
- The inode for the root directory file is in a hard-coded sector.
- A directory is just a special file whose contents are an array of filename-to-inode location mappings.

```
/* A single directory entry. */
struct dir_entry
 {
  block_sector_t inode_sector; /* Sector number of inode.*/
  char name[NAME_MAX + 1];     /* Null terminated file name.*/
  ...
 };
```

Inodes

Current Inode Implementation

File data uses a contiguous block of sectors:

```
/* On-disk inode. */
struct inode_disk
 {
  block_sector_t start; /* First data sector. */
  off_t length;         /* File size in bytes. */
  unsigned magic;       /* Magic number. */
  uint32_t unused[125]; /* Not used. */
 };
```

Inodes

Inode
Metadata (e.g. file size)
Data Sector 1
...
Data Sector N
Indirect Sector
Doubly Indirect Sector
Triply Indirect Sector

Indirect Block
Data Sector $N+1$
Data Sector $N+2$
...
...
...
...
Data Sector $N+M$

Doubly Indirect Block
Indirect Sector
Indirect Sector
...
...
...
...
Indirect Sector

- Keep `sizeof(inode_disk) = BLOCK_SECTOR_SIZE`, `sizeof(indirect_block) = BLOCK_SECTOR_SIZE` etc. This keeps life simpler.
- N will depend how you structure your inodes. $M > N$ is just the number of sector numbers you can fit in a sector.

Example Inode Block

Inode [Sector 10]
Metadata
Data Sector [4]
Data Sector [12]
...
Indirect [23]
Doubly Indirect [33]
Triply Indirect [87]

[Sector 23]
Data [66]
Data [123]
...
...
...
...
Data [17]

[Sector 33]
Indirect [5]
Indirect [NULL]
...
...
...
...
Indirect [34]

[Sector 5]
Data [40]
Data [NULL]
...
...
...
...
Data [91]

- File data can be found on sectors 4, 12, 66, 40, etc.
- NULL indicates that no data has been written to that part of the file.
- Reads from a NULL part of the file should return all 0s.
- You may choose to actually put zeroed sectors in on disk at those locations or not as you like.

Example: How big of a file can I make?

- Suppose your inodes have $N = 10$ direct blocks, 1 indirect block and 1 doubly indirect block. Suppose indirect blocks point to $M = 12$ sectors. Suppose `BLOCK_SECTOR_SIZE=512`. Then the maximum file size is:

$$(10 + 12 + 12 \times 12) \times 512 \text{ bytes} = 84992 \text{ bytes}$$

Other Structures

- You need some way to allocate new sectors to a file as it grows.
- Starter code keeps bitmap of free sectors (similar to the bitmap of free pages in the VM). This bitmap is kept at a hard-coded sector. See `filesys/free-map.c`.

inode_disk vs inode

```
/* On-disk inode. */
struct inode_disk {
  block_sector_t start; /* First data sector. */
  off_t length;         /* File size in bytes. */
  unsigned magic;       /* Magic number. */
  uint32_t unused[125]; /* Not used. */
 };
/* In-memory inode: keep track of transient state, and sector no
struct inode {
  struct list_elem elem; /* Element in inode list. */
  block_sector_t sector; /* Sector number of disk location. */
  int open_cnt;          /* Number of openers. */
  bool removed;          /* True if deleted, false otherwise. */
  int deny_write_cnt;    /* 0: writes ok, >0: deny writes. */
  struct inode_disk data;/* Inode content.*/
  /*^^ YOU SHOULD REMOVE THIS FIELD; RELY ON CACHE*/
 };
```

`inode_disk` vs `inode`

- `inode_disk` is the structure that dictates how inodes actually look on disk.
- `inode` is an in-memory structure that records where to find an inode on disk, as well as temporary information about the corresponding file.
- The `inode` struct will vanish when the computer halts, but `inode_disk` should still be safe on disk.

Synchronization

Synchronization

- Currently you probably have a global "filesystem lock" that serializes all file operations. Now it's time to fix that.
- Operations on different sectors should not impede each other, just like IO in your VM shouldn't block unrelated VM operations.
  - Process B should be able to write to sector 4 in the cache while process A is reading sector 8 into the cache from disk.
  - You'll probably need some kind of fine-grained locking on your cache structure.

Synchronization

- You should allow writing or reading a file from multiple processes at once.
  - You *don't* need to make any guarantees about what happens with simultaneous writes/reads to the same part of a file - writes can interleave, reads can see all or part or none of the writes (synchronizing these accesses is the job of the user application).
  - One exception: a write that *extends* the length of the file should be atomic.
- Be careful: you still need to synchronize finding and evicting items in the buffer cache.

New Syscalls

Syscalls

- `bool chdir(const char *dir)`
- `bool mkdir(const char *dir)`
- `bool readdir(int fd, char *name)`
- `bool isdir(int fd)`
- `bool inumber(int fd)` (It's fine to just have this function return the sector number of the inode for the specified file).

Current Directory, Subdirectories

Current Directory

- Each user process has an associated directory called the current (or working) directory. You need to keep track of this somewhere.
- Child processes inherit the parent's current directory at the time of `exec`.
- Need to handle both *relative* and *absolute* paths in file names. Absolute paths start with "/". Relative paths don't.
- A relative path should traverse the directory tree starting at the current directory, while absolute paths start at the root directory.

Subdirectories

- It is not allowed to use `write` to edit a directory.
- Although simultaneous writes to files may be interleaved, operations on directories must be atomic (otherwise the file system would get corrupted).
- Take care when designing your locking scheme around directories: when deleting a subdirectory you may need to aquire two directory locks at the same time. How can you avoid deadlock when doing this?

Things from lecture that you don't need to do

- You *don't* need to be robust to sudden power failures, so no need to do soft updates or journaling (unless you really want to of course).
- You don't need to implement hard or soft links, so no need for reference counting.
- You don't need to think about the type of disk when scheduling your writes and reads (e.g. no need for CSCAN).

Miscellany

- Although directories are files, you are only allowed to delete a directory via the `remove` syscall if it is empty.
- It is highly recommended to implement the buffer cache first. When done correctly it should be totally transparent to other code and still allow you to pass tests from previous homeworks.
- File names and paths are currently capped at 14 characters. You may allow longer file names if you wish. You MUST allow longer paths.
- You must support file sizes such that you can fill up the entire disk with just one file and its accompanying metadata. The disk is 8MB in size.