


CS 3204

Operating Systems

Project 4 Help Session


Godmar Back



Project 4

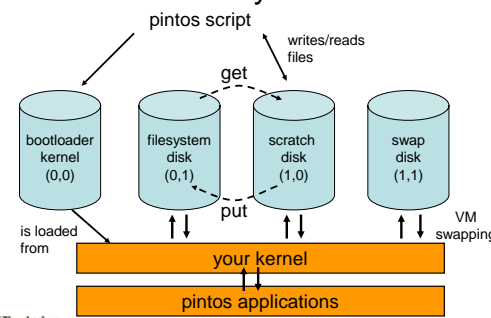
- Final Task: Build a simple file system!
 - “Easier than Project 3”
 - But: more lines of code for complete solution
- Subtasks:
 - Extensible Files
 - Subdirectories
 - Buffer Cache
- Open-ended design problem


Synchronization



CS 3204 Spring 2006
5/3/2006
2

How Pintos's Filesystem Is Used






CS 3204 Spring 2006
5/3/2006
3

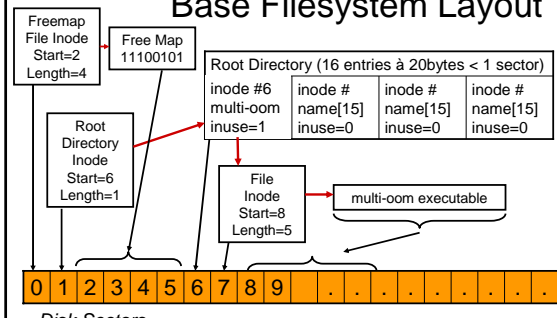
Filesystem Project Overview


- Your kernel must
 - Be able to format the disk when asked (write structures for an initial, empty filesystem on it)
 - Be able to copy files onto it when called from fsutil_put() (which happens before process_execute is called for the first time) – and copy files off of it
 - Be able to support required system calls
 - Be able to write data back to persistent storage
- Only your kernel writes to your disk, you don't have to follow any prescribed layout
 - Can pick any layout strategy that doesn't suffer from external fragmentation and can grow files (simplest strategy is a Unix-style direct, single indirect, double indirect inode layout)
 - Can pick any on-disk inode layout (you have to pick one, the existing one does not work)
 - Can pick any directory layout (although existing directory layout suffices)



CS 3204 Spring 2006
5/3/2006
4

Base Filesystem Layout






CS 3204 Spring 2006
5/3/2006
5

Recommended Order

- Buffer Cache – implement & pass all regression tests
- Extensible Files – implement & pass file growth tests
- Subdirectories
- Miscellaneous: cache readahead, reader/writer fairness, deletion etc.

Synchronization
 (at some point)
 drop global fslock

You should think about synchronization throughout



CS 3204 Spring 2006
5/3/2006
6

The diagram illustrates the flow of data structures in a file system. It starts with a **Per-process file descriptor table** (a vertical list of indices 5, 4, 3, 2, 1, 0) which points to an **Open file table**. The **Open file table** contains entries for **struct file** (inode + position + ...), **struct dir** (inode + ...), and **struct inode** (with a dashed line and a question mark). This table points to the **Buffer Cache**, which contains two blocks. The **Buffer Cache** points to the **On-Disk Data Structures**, which include **files (including directories)**, **nodes**, **index blocks**, **Root Dir Inode**, and **Free Map**. A **PCB** (Process Control Block) is also shown, connected to the **Per-process file descriptor table**.

Virginia Tech CS 3204 Spring 2006 5/3/2006 7

```
graph TD
    A[system calls, fs utils] --> B[file_*( )]
    A --> C[dir_*( )]
    B --> D[inode_*( )]
    C --> D
    D --> E[cache_*( )]
    E --> F[disk_*( )]
```

- Should cache accessed disk blocks in memory
- Should be only interface to disk: all disk accesses should go through it

Cache Block Descriptor

- disk_sector_id, if in use
- dirty bit
- valid bit
- # of readers
- # of writers
- # of pending read/write requests
- lock to protect above variables
- signaling variables to signal availability changes
- usage information for eviction policy
- data (pointer or embedded)

desc → 512 bytes

desc → 512 bytes

desc → 512 bytes

desc → 512 bytes

...

desc → 512 bytes

desc → 512 bytes

desc → 512 bytes

64

Virginia Tech

CS 3204 Spring 2006

5/3/2006

9

```
// cache.h
struct cache_block; // opaque type
// reserve a block in buffer cache dedicated to hold this sector
// possibly evicting some other unused buffer
// either grant exclusive or shared access
struct cache_block * cache_get_block(disk_sector_t sector, bool exclusive);
// release access to cache block
void cache_put_block(struct cache_block *b);
// read cache block from disk, returns pointer to data
void *cache_read_block(struct cache_block *b);
// fill cache block with zeros, returns pointer to data
void *cache_zero_block(struct cache_block *b);
// mark cache block dirty (must be written back)
void cache_mark_block_dirty(struct cache_block *b);
// not shown: initialization, readahead, shutdown
```

- Interface is just a suggestion
- Definition as static array of 64 blocks ok
- Can use Pintos list_elem to implement eviction policy
- Use structure hiding (don't export cache_block struct outside cache.c)
- Must have explicit per-block locking (can't use Pintos's lock since they do not allow for multiple readers)
- (Final version should) provide solution to multiple reader, single writer synchronization problem that starves neither readers nor writers:
 - Use condition variables!
- Eviction: use LRU (or better)

- Would like to bring next block to be accessed into cache before it's accessed
- Must be done in parallel
 - use daemon thread and producer/consumer pattern
- Note: next(n) not always equal to n+1
- Don't initiate read_ahead if next(n) is unknown or would require another disk access to find out

```
b = cache_get_block(n, _);
cache_read_block(b);
cache_readahead(next(n));
```

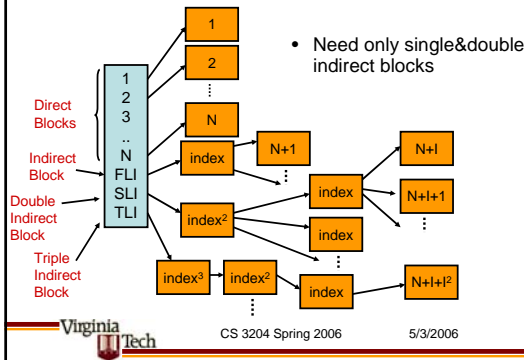
```

queue q;
cache_readahead(sectors) {
    q.lock();
    q.add(request(s));
    qcond.signal();
    q.unlock();
}

cache_readahead_daemon()
while (true) {
    q.lock();
    while (q.empty())
        qcond.wait();
    s = q.pop();
    q.unlock();
    read sector(s);
}

```

Multi-Level Indices



Multi-Level Indices

- How many levels do we need?
- Max Disk size: 8MB = 16,384 Sectors
- Assume sector number takes 2 or 4 bytes, can store 256/128 in one sector
- Filesize (using only direct blocks) < 256
- Filesize (direct + single indirect block) < 2×256
- File (direct + single indirect + double indirect) < $2 \times 256 + 256^2$

Files vs. Inode vs. Directories

- Offset management in struct file etc. should not need any changes
 - Assuming single user of each struct file, so no concurrency issues
- You have to completely redesign struct inode_disk to fit your layout
- You will have to change struct inode & struct dir
 - struct inode can no longer embed struct inode_disk (inode_disk should be stored in buffer cache)

struct inode vs struct inode_disk

```
struct inode {
    disk_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. */
struct inode {
    struct list_elem elem; /* Element in inode list. */
    disk_sector_t sector; /* Sector number of disk location. */
    int open_cnt; /* Number of openers. */
    bool removed; /* True if deleted, false otherwise. */
    int deny_writes; /* True if deny writes, >0: deny writes. */
    struct inode_disk data; /* Inode content. */
};
```

Extending a file

- Seek past end of file & write extends a file
- Space in between is filled with zeros
 - Can extend sparsely (use “nothing here” marker in index blocks)
- Consistency guarantee on extension:
 - If A extends & B reads, B may read all, some, or none of what A wrote
 - But never something else!
 - Implication: do not update & unlock metadata structures (e.g., inode length) until data is in buffer cache

Subdirectories

- Support nested directories (work as usual)
- Requires:
 - Keeping track of type of file in on-disk inode
- Should not require changes to how individual directories are implemented (e.g., as a linear list)
 - should be able to reuse existing code
 - Specifically, path components remain ≤ 14 in length
 - Once file growth works, directory growth should work “automatically”
- Implement system calls: mkdir, rmdir
 - Need a way to test whether directory is empty

Subdirectories: Lookup

- Implement absolute & relative paths
- Use strtok_r to split path
 - Recall that strtok_r() destroys its argument - make sure you create copy if necessary
 - Make sure you operate on copied-in string
- Walk hierarchy, starting from root directory (for absolute paths); current directory (for relative paths)
- All components except last must exist & be directories



CS 3204 Spring 2006

5/3/2006

19

Current Directory

- Need to keep track of current directory (in struct thread)
 - Warning: before first task starts, get/put must work but process_execute hasn't been called
- Current directory needs to be kept open
 - Requires a table (e.g., list) of open directories & reference counting for directories
 - Can be implemented for struct dir analogous to struct inode using a list



CS 3204 Spring 2006

5/3/2006

20

Synchronization – General Hints

- Always consider: what lock (or other protection mechanism) protects which field:
 - If lock L protects data D, then all accesses to D must be within lock_acquire(&L); ... Update D ...; lock_release(&L);
- Should be fine-grained: independent operations should proceed in parallel
 - Example: don't lock entire path when looking up file
 - Files should support multiple readers & writers
 - Removing a file in directory A should not wait for removing file in directory B
- May use embedded locks directly (struct inode, struct dir, free map)
 - Or be built upon locks (struct cache_block)
- For full credit, must have dropped global fs lock
 - Can't see whether any of this works until you have done so



CS 3204 Spring 2006

5/3/2006

21

Free Map Management

- Can leave almost unchanged
- Read from disk on startup, flush on shutdown
- Instead of allocating n sectors at file creation time, now allocate 1 sector at a time when file is growing
 - Do clustering for extra credit
- If file_create("...", m) is called with m > 0, simulate write_at(offset=m, 1 byte of data); to expand to appropriate length
- Don't forget to protect free_map() with lock



CS 3204 Spring 2006

5/3/2006

22

Grading Hints

- Tests are rather incomplete
- Possible to pass all tests without having synchronization (if you keep global fslock), persistence, deletion, or buffer cache implemented
 - TAs will grade those aspects by inspection/reading your design document
- Core parts (majority of credit) of assignment are
 - Buffer cache
 - Indexed & extensible files
 - Subdirectories



CS 3204 Spring 2006

5/3/2006

23