CS 3204
Operating Systems

Project 4 Help Session

Godmar Back
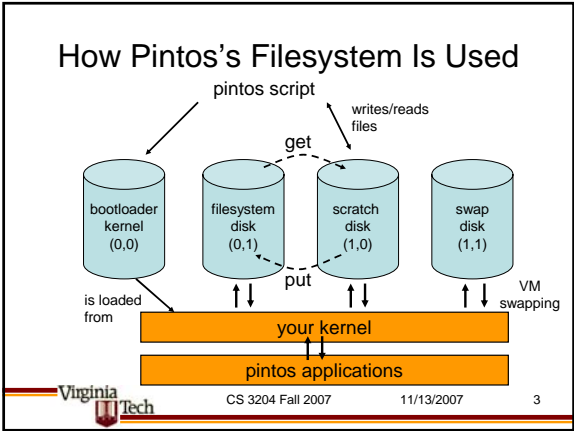
Virginia Tech

Project 4

- Final Task: Build a simple file system!
  - "Easier than Project 3" – maybe
  - But: definitely more lines of code for complete solution
    - And no room for errors – it's a filesystem, after all!
- Subtasks:
  - Buffer Cache
  - Extensible Files   } Synchronization
  - Subdirectories
- Again open-ended design problem

How Pintos's Filesystem Is Used

Project Requirements

- 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
    - New calls: mkdir, readdir, inumber, isdir, chdir
  - Be able to write data back to persistent storage
  - Be able to copy files from it when called from fsutil_get()

Project Requirements (cont'd)

- Only your kernel writes to and reads from your disk
- Don't have to follow any prescribed layout
- Can pick any layout strategy that doesn't suffer from external fragmentation and can grow files
  - recommend Unix-style direct, single indirect, double indirect inode layout – but feel free to be creative
- Can pick any on-disk inode layout (you must design your own, the existing one does not work)
- Can pick any directory layout (although existing directory layout suffices)

Base Filesystem Layout

Disk Sectors

1
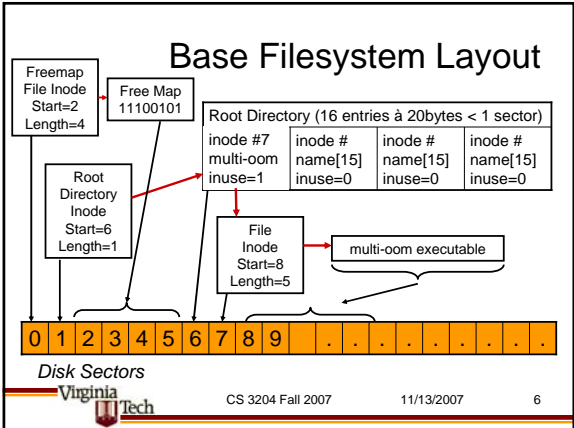
Recommended Order

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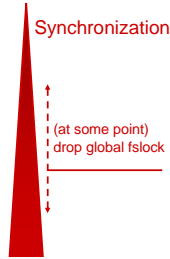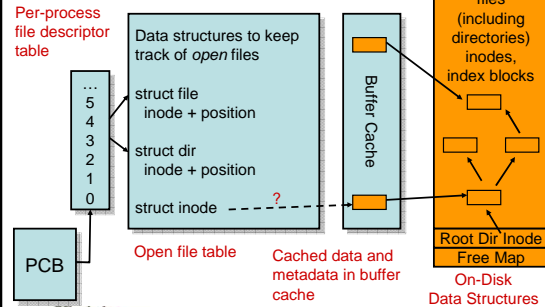2. Extensible Files – implement & pass file growth tests
3. Subdirectories
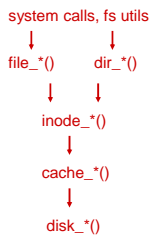4. Miscellaneous: cache readahead, reader/writer fairness, deletion etc.

You should think about synchronization throughout

Synchronization
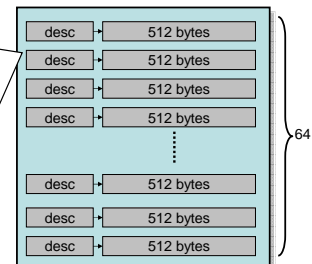
(at some point) drop global fslock

Virginia Tech

CS 3204 Fall 2007   11/13/2007   7

The Big Picture

Per-process file descriptor table

Data structures to keep track of *open* files

...
5
4
3
2
1
0

struct file
  inode + position

struct dir
  inode + position

struct inode - - - - - ?

PCB

Open file table

Buffer Cache

files (including directories) inodes, index blocks

Cached data and metadata in buffer cache

Root Dir Inode
Free Map

On-Disk Data Structures

Virginia Tech

CS 3204 Fall 2007   11/13/2007   8

Buffer Cache (1): Overview

system calls, fs utils
↓
file_*()   dir_*()
↓   ↓
inode_*()
↓
cache_*()
↓
disk_*()

- Should cache accessed disk blocks in memory
- Buffer cache should be only interface to disk: all disk accesses should go through it
  – Ensures consistency!

Virginia Tech

CS 3204 Fall 2007   11/13/2007   9

Buffer Cache (2): Design

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 Fall 2007   11/13/2007   10

Buffer Cache (3): Interface

```
// 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
```

Virginia Tech

CS 3204 Fall 2007   11/13/2007   11

Buffer Cache (4): Notes

- Interface is just a suggestion
- Definition as static array of 64 blocks ok
- 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)
- Should provide solution to multiple reader, single writer synchronization problem that starves neither readers nor writers:
  – Use condition variables!
- Eviction: use LRU (or better)
  – Can use Pintos list_elem to implement eviction policy, such as LRU via stack implementation

Virginia Tech
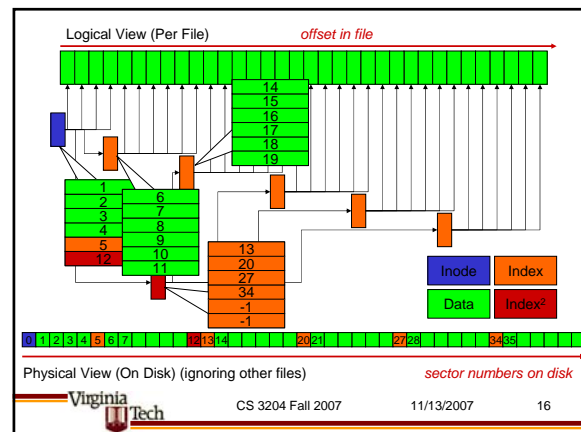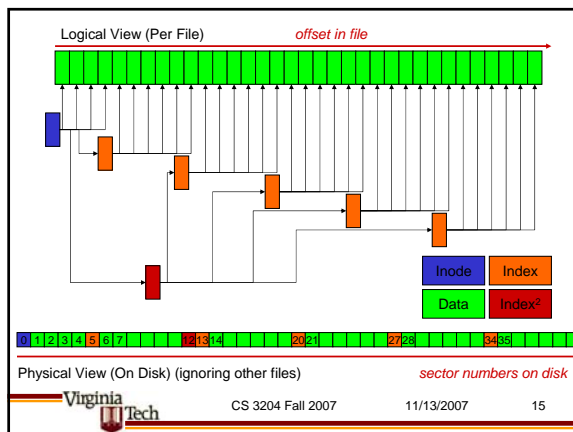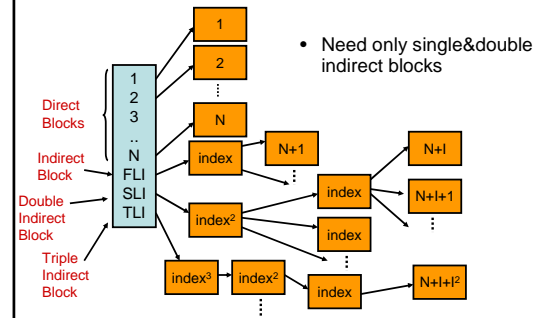
CS 3204 Fall 2007   11/13/2007   12

Buffer Cache (5): Prefetching

- 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(sector s) {
   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

- Need only single&double indirect blocks

Logical View (Per File)   *offset in file*

Inode | Index
Data | Index²

Physical View (On Disk) (ignoring other files)   *sector numbers on disk*

Logical View (Per File)   *offset in file*

Inode | Index
Data | Index²

Physical View (On Disk) (ignoring other files)   *sector numbers on disk*

Multi-Level Indices (cont'd)

- How many levels do we need?
- Max Disk size: 8MB = 16,384 Sectors
- Assume sector number takes 2 or 4 bytes, can store 256 or 128 in one sector
- Filesize(using only direct blocks) < 256
- Filesize(direct + single indirect block) < 2*256
- File (direct + single indirect + double indirect) < $2*256 + 256^2 = 66,048$

Files vs. Inode vs. Directories

- Offset management in struct file etc. should not need any changes
  - If there's no sharing of struct file/dir instances between processes, then there are no concurrency issues since Pintos's processes are single-threaded!
- You have to completely redesign struct inode_disk to fit your layout
- You will have to change struct inode
  - struct inode are necessarily shared between processes – since they represent files on disk!
  - struct inode can no longer embed struct inode_disk (inode_disk should be stored in buffer cache)

struct inode vs struct inode_disk

```
struct in redesign for indexed approach
{
    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_wr store in buffer cache writes ok, >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 file 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
  - Distinction between file descriptors in syscall layer – e.g., must reject write() to open directory
- Should only require minor changes to how individual directories are implemented (e.g., as a linear list – should be able to reuse existing code)
  - Must implement "." and ".." – simple solution is to create the two entries on disk when a directory is created.
  - Must support path names such as ///a/b/../c/./d
  - Path components can remain <= 14 in length
  - Once file growth works, directory growth should work "automatically"
- Implement system calls: readdir, mkdir, rmdir
  - Need a way to test whether directory is empty
  - readdir() should not return . and ..

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
- Make sure you don't leak memory, or you fail dir-vine.

Current Directory

- Need to keep track of current directory
  - in struct thread
  - be aware of possible initialization order issues: before first task starts, get/put must work but process_execute hasn't been called
- When an attempt is made to delete the current directory, or any open directory, either
  - Reject
  - Allow, but don't allow further use

Synchronization Issues (1)

- 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);
- Embed locks in objects or define as static variables where appropriate (struct inode)
- For buffer cache entries, must build new synchronization structure (Single Writer/Multiple Reader lock without starvation) on top of existing ones (locks + condition variables)
- For directories, can use lock on underlying inode directly to guarantee exclusive access

Synchronization Issues (2)

- Should be fine-grained: independent operations should proceed in parallel, for example
  - Don't lock entire buffer cache when waiting for read/write access of individual buffer cache entry
  - Example: don't lock entire path resolution component when looking up file along /a/b/c/d
  - Files should support multiple readers & writers
    - Data writes do not require exclusive access to buffer cache block holding the data!
  - Process removing a file in directory A should not wait for removing file in directory B
- For full credit, must have dropped global fs lock
  - Can't see whether any of this works until you have done so

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, and only when file is growing
  - Implement extents for extra performance + credit
- But: must still support creating files that have an initial size greater than 0; easy to do:
  - If **file_create("…", m)** is called with m > 0, simulate **write_at(offset=m, 1byte of data);** to expand to appropriate length
- Don't forget to protect free_map() with lock

Grading Hints

- Extended tests won't fully pass until file growth + subdirectories are sufficiently implemented such that 'tar' works.
- Core parts (majority of credit) of assignment are
  - Buffer cache
  - Extensible files
  - Subdirectories
- For this assignment, credit for regression tests will depend on how many parts (n = 0, 1, 2) of the assignment you've implemented
  - Credit for regression tests = Reported TestScore * n/3
  - Don't get credit for resubmitting P2.
- Tests will not detect
  - If you keep global fslock or not
  - If you have a buffer cache
  - TAs will grade those aspects by inspection/reading your design document
- Good Luck!