PROJECT 6: PINTOS FILE SYSTEM

CS124 – Operating Systems

Winter 2015-2016, Lecture 25

Project 6: Pintos File System

- Last project is to improve the Pintos file system
  - **Note: Please ask before using late tokens on Project 6**
- Initial Pintos file system is somewhat limited:
  - Each file is a single contiguous extent on disk
  - Files cannot grow once they are created
  - There is only a single root directory in the file system
  - The root directory can only hold up to 16 directory entries
  - The filesystem can only be used by one kernel thread at a time
- You must implement:
  - Extendable files that don't require contiguous space on disk
  - Directories that can grow to hold more file entries as needed
  - Directory hierarchies
  - Support for fast, concurrent access from many kernel threads

Project 6: Pintos File System (2)

- Suggested order of implementation:

1. File system buffer cache
   - Will force you to think about concurrent access from the start
2. Extendable files
   - Suggested: use an ext2-like indexed file layout (not required)
   - *Note: files don't need to support shrinking or truncation…*
3. Subdirectories
4. Remaining miscellaneous items
   - Process "current directory"
   - Parsing and navigating directory paths
   - Read-ahead and write-behind facilities

- Assignment suggests that items 2 and 3 can be implemented in parallel

Project 6: Pintos File System (3)

- The file system should support concurrent access
- Files read and written by multiple concurrent processes
  - File-extension needs to be atomic (for correctness)
  - Readers shouldn't block each other
  - Writes to different parts of a file shouldn't block each other (this can be true on a per-sector basis)
  - Readers shouldn't see written data until the write is completed (this can be true on a per-sector basis)
- Directory operations on different directories should also be concurrent
- You should not have a global filesystem lock (or generally, other global locks) in your Project 6 submission
  - -40 point deduction for a global filesystem lock

Filesystem Cache

- Filesystem cache is pretty straightforward to implement
- Limited to 64 cache blocks
  - Both file data and metadata count against this limit
  - (Can exclude the file system's free-space map, if you wish)
  - Blocks need "dirty" bit; write dirty blocks back to disk when evicted
- One idea for an approach:
- Initially build cache but retain the global file-system lock
  - Make all file-system accesses use the cache
  - Make sure all tests still pass…
- Then, replace global lock with per-entry read/write locking
  - Readers don't block readers; writers block everybody
  - Make sure all tests still pass…
- Finally, any more advanced concurrency issues to resolve

Filesystem Cache (2)

- Can implement cache as statically alloc'd array of entries
  - Could use a hash-table to look up entries based on sector # (fast), or do a linear search through the table (much simpler, but slower)
  - Entries will need a lock of some kind (i.e. a read/write lock) to coordinate access to the data they hold
- Will need some kind of eviction policy when cache is full
  - Assignment again says CLOCK is the minimum bar; this is simple, or you could even implement LRU if you want to be fancy
- Many other policies are possible! For example:
  - Metadata is perhaps more useful to keep in cache than file data…
  - Maybe try a "points-based NFU" mechanism – metadata access is worth 2 points, and data access is worth 1 point (see lecture 21)
  - Evict the entry with the lowest score
  - (Recall NFU's limitations – likely need to decay entries' points…)

Filesystem Cache (3)

- Just as with virtual memory, there are many concurrent access scenarios to consider
- Example:
  - Process A wants to read sector 234; cache uses a global hash table to look up which cache entry contains this sector
  - Meanwhile, Process B has determined that sector 216 is not in the cache, and needs to evict an entry to make space…
  - …and, sector 234’s entry was chosen for eviction
- If you use a hash table, how to guard concurrent access and modification of the table itself? (another r/w lock?)
- Can a process get to a filesystem-cache entry, and then discover that the entry is not for the sector it wants?
  - Some solutions for the specified concurrency goals have this characteristic

Cache Concurrency

- Using a global mapping of sector numbers to file system cache entries can introduce concurrency issues
- Example:
  - A global hash-map that maps sector numbers to cache entries
  - A global lock used to guard the hash-map
- Reading or writing a block in the cache:
  - Acquire global lock
  - Find block in cache
  - Acquire the block's lock
  - Release global lock
  - Perform IO on block
  - Release the block's lock
- (A simple crabbing mechanism)

Cache Concurrency (2)

- With this approach, processes can become blocked on the global lock ☹
- An example scenario:
  - Kernel thread 1 performs a read on block B
  - Kernel thread 2 chooses to evict block B (or do anything else on B)
  - Kernel thread 3 performs IO on *any other block* in the system

Kernel Thread 1	Kernel Thread 2	Kernel Thread 3
Acquire global lock Find block B in cache Acquire B's lock Release global lock Start reading data into B…		
	Acquire global lock Find block B in cache Block on B's lock…	
		Block on global lock…

Cache Concurrency (3)

- Scenario:

Kernel Thread 1	Kernel Thread 2	Kernel Thread 3
Acquire global lock Find block B in cache Acquire B's lock Release global lock Start reading data into B…		
	Acquire global lock Find block B in cache Block on B's lock…	
		Block on global lock…

- The problem is that we hold the global lock until the entry's lock is acquired…
  - …and sometimes other threads are doing *really slow things* with the entry

Cache Concurrency (4)

- What if we changed the approach to release the global lock *before* acquiring the block's lock?

Kernel Thread 1	Kernel Thread 2	Kernel Thread 3
Acquire global lock Find block B in cache Release global lock Acquire B's lock Start reading data into B…		
	Acquire global lock Find block B in cache Release global lock Block on B's lock…	
		Acquire global lock *(continue on…)*

- Allows kernel-thread 3 to proceed…
- But, also allows new interleavings of kernel threads…

Cache Concurrency (5)

- Now this is also possible:

Kernel Thread 1	Kernel Thread 2
Acquire global lock Find block B in cache Release global lock	
	Acquire global lock Find block B in cache Release global lock Acquire B's lock Evict B Release B's lock
Acquire B's lock Start accessing B's data ??	

- Solutions?
- A simple solution: after the thread acquires B's lock, verify that the cache entry still holds block B
- If cache entry no longer holds block B when the thread gets the cache entry's lock, just go back to the hash table and try again
  - Acquire the global lock, find block B in cache, release the lock, etc.

Read-Write Locks

- Typically, must implement some kind of read-write lock
  - (also known as a shared-exclusive lock)
  - Pintos implementation doesn't include one
- Should try to make your read-write lock fair for both readers and writers
  - If lock is currently held by readers, and another reader requests the lock, only grant the request it if no writers are currently blocked
  - If lock is released by a writer, should probably try to grant the lock to waiting readers next, so that writers also can't starve out readers
  - (Really don't need to order lock-grants based on the order of lock-requests; just make sure nobody can be starved!)
- Many ways to build a read-write lock…
  - Just make sure to encapsulate lock and unlock operations so that it's easy to use them wherever you need to!

Read-Write Locks (2)

- Pintos has locks (i.e. mutexes) and condition variables – perfectly sufficient to build a read-write lock
  - One lock for making operations on the read-write lock atomic
  - Two condition variables that wait on this lock – one for waiting readers, and one for waiting writers
  - Other state variables, e.g. for current mode of the lock (unlocked, read-locked, write-locked), number of waiting readers, number of waiting writers, etc. (many variations to choose from)
- Condition variables maintain a queue of waiting threads
  - Can signal on the condition variable to wake up one waiter (e.g. to allow one waiting writer to proceed)
  - Can also broadcast on the condition variable to wake up all waiters (e.g. to allow all waiting readers to proceed)
- Important: waiters on a condition variable cannot assume that just because they wake up, the desired condition is true!
  - Must wait on condition variable in a loop, until the desired condition is actually true

Read-Ahead and Write-Behind

- Filesystem must support read-ahead and write-behind
- **Read-ahead:** If a process reads sector *n*, the filesystem should prefetch sector *n*+1 in the background
  - Goal is entirely to improve sequential file access
  - (Note that this can easily affect random access very negatively)
- **Write-behind:** Filesystem should periodically traverse all cached entries, and write dirty ones back to disk
  - Goal is to ensure that the operating system is robust in the face of unexpected crashes or other system failures
- Both these operations need to happen in background…
- An easy approach: Implement them with dedicated kernel threads that take care of these responsibilities

Read-Ahead and Write-Behind (2)

- Read-ahead service:
  - A kernel thread that is spun up on initialization of the filesystem
  - A shared queue that the kernel thread listens to – sectors to read-ahead are added to this queue
  - When a process reads a sector *n*, the kernel thread adds the value *n*+1 to the read-ahead thread's queue
  - The read-ahead thread receives these requests and attempts to read the specified sector (causes it to be loaded into the cache)
- Write-behind service:
  - Another kernel thread spun up on initialization of the filesystem
  - Sits in a loop, sleeping for a specific period of time, then waking up and writing back any dirty sectors in the filesystem cache
  - When a sector is written back, its dirty bit must be cleared
  - (Probably should lock entry so it can't change while writing it back)

Extendable Files

- Need to update file system to support file-extension, and files that are not contiguous extents
- Requires two things:
  - A free-space map (`free-map.c` / `free-map.h`)
  - A more sophisticated inode structure for representing where a file's data is (`inode.c` / `inode.h`)
- Both of these structures are initially focused on simple single-extent contiguous files
  - Feel free to change both the data representations, and the APIs that these C files expose!
- Example: the free map can continue being a bitmap...
  - But, change `free_map_allocate()` to allocate non-contiguous sectors from the disk (or, change it to allocate one sector at a time)

Extendable Files (2)

- Given: file system partition will never be larger than 8MiB (and also, files can only be a maximum of 8MiB)
- 8MiB = 16384 512-byte sectors
- Implications:
  - Free-space map can be up to 2KiB (4 sectors) in size
  - File inodes must be able to index up to 16384 sectors using some kind of multilevel indexing structure
- The free-space map is a specific file on the file system
  - It doesn't appear in the directory structure
  - Rather, its inode is hard-coded to reside at sector 0 (see **`FREE_MAP_SECTOR`** in **`filesys.h`**)
  - The bitmap data structure implementation includes functions to read and write bitmaps to a file

Inodes

- An “inode” is the root of the indexing and metadata information for a given file on disk
  - The inode may (usually will) occupy multiple non-contiguous sectors
- The file **`inode.c`** has two inode data structures
  - **`inode_disk`** is the on-disk representation; must be exactly 512 bytes (specified as **`BLOCK_SECTOR_SIZE`** bytes)
  - **`inode`** is the in-memory representation
  - Both of these will likely change significantly as you do Project 6
- Note: **`inode`** structure initially has an **`inode_disk`** member…
  - This is before you have added the file system cache!
  - Once you create the file system cache, the **`inode_disk`** data will actually be stored in a cache entry, not in the **`inode`** struct
  - (Can cast the cache entry’s array of data into **`inode_disk*`** to access the inode’s contents)

Inodes (2)

- The assignment suggests using an ext2-like approach
- Inode will reference (besides metadata):
  - Some number of direct nodes
  - Some number of single-indirect nodes (> 0)
  - Some number of double-indirect nodes (> 0)
  - Not triple-indirect nodes (thankfully!)
- Direct nodes cannot represent 8MiB files by themselves…
  - 512 bytes / 4-byte index entries = 128 sectors = 64KiB files
  - (Ignoring space occupied by metadata in the inode!)
- It's up to you to decide how many single-indirect and double-indirect entries to use in your implementation
  - Hint: Probably want to use as few as possible, for simplicity ☺

Inodes (3)

- Example: constants for number of inode entries

```
#define NUM_DIRECT ...
#define NUM_INDIRECT ...
#define NUM_DOUBLE_INDIRECT ...
#define NUM_ENTRIES (NUM_DIRECT + NUM_INDIRECT + NUM_DOUBLE_INDIRECT)
```

- Can either put all entries into a single array…

```
struct inode_disk {
    block_sector_t sectors[NUM_ENTRIES];
    ...
};
```

- Or, can have multiple arrays

```
struct inode_disk {
    block_sector_t direct[NUM_DIRECT];
    block_sector_t indirect[NUM_INDIRECT];
    block_sector_t double_indirect[NUM_DOUBLE_INDIRECT];
    ...
};
```

- Both approaches have identical data layout within the sector
- Second approach is probably easier to understand

File Extension

- Need to support file extension, by multiple processes
- **File extension must be an atomic operation:**
  - Data and metadata must be kept in sync, or else file will be corrupted
- Implication: in-memory **`inode`** structure needs some kind of lock for governing file extension
- Still need to make sure that other reads/writes within the file are allowed to proceed concurrently…
- Must check the file's size to see if a given write is going to extend the file or not
  - If extending, make sure the file-extension lock is acquired first
  - If not extending, rely on the locks in the file system cache to govern concurrent access
- Recall: in Project 6, files can only grow, not shrink
  - A write within a file will remain a write within the file, regardless of what other processes might be doing to the file

File Extension (2)

- Two main choices for file extension

1. Always lock file-extension lock before checking length
   - Benefit: easy to get right!
   - Drawback: must acquire this lock for all reads and writes, even if it is immediately released

- If several writers are trying to extend the file, a writer to the interior will be blocked until it can acquire this lock
- Has the potential to significantly reduce the concurrency of the file system ☹

File Extension (3)

2. Use **double-checked locking**
   - Pseudocode:

```
if (write_position >= inode->length) {
    /* We might be extending the file... */
    /* Get the lock and then check again to be sure! */
    acquire_lock(inode->extension_lock);
    if (write_position >= inode->length) {
        /* Yep, definitely extending the file. */
        ...  /* extend the file */
        release_lock(inode->extension_lock);
        return;
    }
    release_lock(inode->extension_lock);
}
/* If we get here, we are writing the file’s interior. */
... /* write to interior */
```

File Extension (4)

- Double-checked locking can be very tricky to get right
  - Pseudocode:

```
if (write_position >= inode->length) {
    /* We might be extending the file.  Get the lock
     * and then check again to be sure! */
    acquire_lock(inode->extension_lock);
    if (write_position >= inode->length) {
```

- Compilers can be too clever, and cache `inode->length` into a register
  - Second comparison doesn't actually give us a correct answer
  - (Or, second comparison is simply optimized away, yay!)
- Solutions are the same as always
  - Cast `inode->length` to be volatile (easiest), or use a barrier

File Extension (5)

- Finally, must think about order of operations when extending files
  - When a process extends a file by writing data past EOF, other processes shouldn't see the new data until the write is completed
- Need to do these things:
  - Update the file's length metadata stored in the inode
  - Write new data into the file itself (including intervening data if you implement a dense file representation, not a sparse one)
  - Update the file's inode structure to point to the new sectors
  - Release the file-extension lock
- Think carefully about what order to do these things, so that other processes reading from the file won't see the write until it is fully completed

Directories

- In UNIX systems, directories are simply special files
  - Instead of exposing the directory-file's raw contents to applications, the OS parses and manages the file's contents itself
- File-system inodes can record whether a file is a "normal file" or a "directory" in the metadata
- Directory entries associate string names with where files or subdirectories reside on disk…
  - Easy approach is to have the directory store the sector of the inode of each file or subdirectory
- As with the free-space map, the location of the root directory's inode is hard-coded to sector 1 in the OS
  - (See **`ROOT_DIR_SECTOR`** in **`filesys.h`**)

Directories (2)

- As usual, Project 6 specifies strong concurrency requirements on directory operations
- Concurrent operations on different directories should proceed in parallel
  - Suggests that synchronization should be performed using locks on directory-inodes (or the corresponding `dir` struct)
  - (Can probably come up with a unified abstraction for both files and directories to use the same basic inode type)
- Try to avoid global collections of open directories, if possible
  - Any global collection like this will require synchronization that reduces the concurrency of your overall system
- Typically, the things that need to keep track of directories can simply store the directory’s inode
  - e.g. the process’ current directory
  - Also allows the process to record that it is using the directory, so that e.g. it can’t be deleted while the process is running

In-Use Files and Directories

- Pintos follows UNIX-like semantics for file deletion (Project 4)
  - A file may be deleted even if a process has the file open
  - When the process terminates, the file is reclaimed
- In Project 6, processes will keep track of their current directory
  - Can prevent a directory from being deleted if any process is using the directory
- Easy enough to implement by adding an "in-use" count to the inode corresponding to the file or directory
  - Increment "in-use" value when file or directory is opened (or when a process changes into the directory)
  - Decrement "in-use" value when file or directory is closed (or when a process changes away from the directory)
  - When "in-use" value hits 0, carry out appropriate actions.
    Or, if "in-use" value > 0, prevent certain actions.

Project 6: Pintos File System

- That should cover most of the key points for Project 6
  - As always, the devil is in the details…
- Ask for help if you get stuck on anything!
- **Please ask before using late tokens on Project 6!**
  - Donnie has to grade all of the submissions in about 2 days. ☹