Buffer cache

The functions `inode_read_at` and `inode_write_at` currently access the file system's underlying block device directly each time you call them. Your task is to add a buffer cache for the file system, to improve the performance of reads and writes. Your buffer cache will cache individual disk blocks, so that (1) you can respond to reads with cached data and (2) you can coalesce multiple writes into a single disk operation. The buffer cache should have a maximum capacity of 64 disk blocks. You may choose the block replacement policy, but it should be an approximation of MIN based on locality assumptions. For example, using LRU, NRU (clock), n-th chance clock, or second-chance lists would be acceptable, but using FIFO, RANDOM, or MRU would *not* be. Choosing a replacement policy of your own design would require serious justification. The buffer cache must be a **write back cache**, not a write-through cache. You must make sure that all disk operations use your buffer cache, not just the two inode functions mentioned earlier.

Implementation details

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. **Your cache must be 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 metadata compared to data. You can experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses. Running Pintos from the `filesys/build` directory will cause a sum total of disk read and write operations to be printed to the console, right before the kernel shuts down.

When one thread is actively writing or reading data to/from a buffer cache block, you must make sure other threads are prevented from evicting that block. Analogously, during the eviction of a block from the cache, other threads should be prevented from accessing the block. If a block is currently being loaded into the cache, other threads need to be prevented from also loading it into a different cache entry. Moreover, other threads must not access a block before it is fully loaded. You can keep a cached copy of the free map permanently in a special place in memory if you would like. It doesn't count against the 64 sector limit.

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.

When data is written to the cache, it does not need to be written to disk immediately. You should keep dirty blocks in the cache and write them to disk when they are evicted and when the system shuts down (modify the `filesys_done()` function to do this).

If you only flush dirty blocks on eviction or shut down, your file system will be more fragile if a crash occurs. As an optional feature, you can also make your buffer cache periodically flush dirty cache blocks to disk. If you have a non-busy-waiting `timer_sleep()` from Project 2 working, this would be an excellent use for it. Otherwise, you may implement a less general facility, but make sure that it does not exhibit busy-waiting.

As an optional feature, you can also implement read-ahead, that is, automatically fetch the next block of a file into the cache when one block of a file is read. Read-ahead is only really useful when done asynchronously. That means, if a process requests disk block 1 from the file, it should block until disk block 1 is read in, but once that read is complete, control should return to the process immediately. The read-ahead request for disk block 2 should be handled asynchronously, in the background.
