File System Part I: Disk Cache

Yunhao Zhang Robbert van Renesse

1 Motivation for having a Disk Cache

1.1 Hardware: Capacity vs. Access Latency

A common trade-off for many storage devices is between *capacity* and *access latency*. Usually, storage devices with smaller capacity have lower access latency, while those with larger capacity have higher access latency.

In this project, you will focus on two storage devices: main memory and a hard disk. In modern personal computers, the capacity of memory is typically between 16 GB and 128 GB, while that of a hard disk is typically 1TB or larger. As for access latency, main memory is accessed in the unit of a page (typically 4KB), costing about 200 CPU cycles. In contrast, a hard disk is accessed in the unit of a block (512 bytes usually), costing about 100,000 CPU cycles.

Such differences suggest that an operating system should try to keep data in main memory rather than on disk whenever possible. Since the capacity of memory is limited, this leads to the following question: which part of the disk should be loaded into memory to reduce disk accesses? To answer this question, we need to look into a common software trait.

1.2 Software: Locality

Locality is one of the most important observations in the field of operating systems (Denning, 1968). In the context of this project, locality means: a page that is accessed recently is likely to be accessed again and often in the near future.

To take advantage of locality, many operating systems maintain a region in memory called the *disk cache*. If a process reads the same disk block twice without modification, the operating system can put its content in the disk cache during the first read, and directly reply with the content for the second read operation. In this way, the disk cache leverages the locality of the process and improves its performance by converting a disk access into a memory access, transparently.

When the disk cache is full and a new block needs to be cached, the operating system needs to choose a block and *evict* it from the disk cache. You have learned several eviction algorithms in CS4410. In this project, you will implement the CLOCK algorithm for both a *write-through* disk cache and a *write-back* (aka *write-behind*) disk cache. You will also learn about the general metrics for evaluating an eviction algorithm.

2 Understanding a Hard Disk

A hard disk is a typical *block device*, so called because it is organized as a sequence of blocks. For example, a 200GB hard disk with block size of 512 bytes would have 419,430,400 blocks, numbered from 0 to 419,430,399. The operating system can issue `read` and `write` operations to the hard disk by providing three arguments: a *inode number*, a *block number*, and a memory address. For read operations, the hard disk finds the block specified by the inode number and block number in its internal physical structure and copies its content to the given memory address. Write operations, inversely, read the data at the memory address and write it to the physical structure in the hard disk.

3 Write-Through Cache & Write-Back Cache

3.1 Write-Through Cache Control Flow

Figure 1 shows the structure of a write-through cache. In general, a cache contains a fixed number of *cache slots*, each containing some metadata and some data. In the case of disk cache, each slot is used for caching a disk block. If a slot is in use, it contains the inode number and the block number in its metadata and the block content in its data region. If a slot is not in use, it will have arbitrary content (whatever was left in it during its last use).

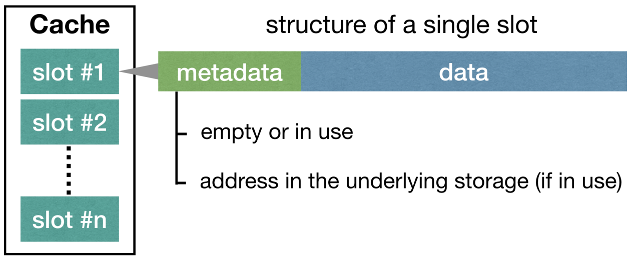
Cache
slot #1
slot #2
slot #n
structure of a single slot
metadata
data
empty or in use
address in the underlying storage (if in use)

Figure 1: The structure of a write-through cache

For a `read` operation, the operating system scans the cache slots and checks whether there is a slot in use for the target inode number and block number. If such a slot exists, the operating system replies with the data in that slot. Otherwise, it reads the block from the hard disk, adds this block to the cache, and replies with the block data it just read.

For a `write` operation, similarly, the operating system scans the cache slots and checks whether there is a slot in use for the target inode number and block number. If such a slot exists, the operating system updates its content and **immediately writes the block data to the disk.** This is why it's called a **write-through cache**. If such a slot doesn't exist, the operating system first allocates a slot in the cache, then writes the data to both the cache and the disk.

3.2 Write-Back Cache Control Flow

Figure 2 shows the common structure of a write-back cache. Compared to a write-through cache, there is one more field in the metadata, indicating whether the cache slot is *dirty* or *clean*.

The `read` operation for write-back cache is the same as for a write-through cache. For a `write` operation, the operating system writes the block data to a cache slot and sets its dirty bit to 1, **without modifying the data on disk**. By doing so, the write operation can complete much faster than using a write-through cache, but the trade-off is that data in the cache and disk are now inconsistent (i.e., the disk contains stale data). Later, the operating system can issue a `sync` operation to the cache that writes all the dirty blocks in the cache to disk, making the two consistent once again.

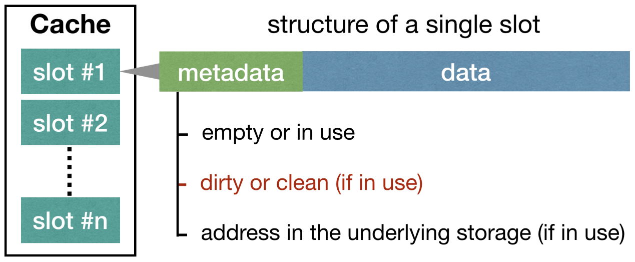
Cache
slot #1
slot #2
slot #n
structure of a single slot
metadata
data
empty or in use
dirty or clean (if in use)
address in the underlying storage (if in use)

Figure 2: The structure of a write-back cache

3.3 Eviction Algorithm

For both `read` and `write` operations, adding a block to the cache involves finding an empty cache slot. This is easy enough if one exists. Otherwise, we need an eviction algorithm to select a cache slot to free up.

In this project, you are required to implement the CLOCK algorithm for both a write-through and a write-back cache. The CLOCK algorithm is an approximation for the LRU (Least-Recently-Used) algorithm. Please study the related CS4410 slides for the details of this algorithm.

4 Implementation

You are to implement the write-through cache in `block/wtclockdisk.c` and the write-back cache in the file `block/clockdisk.c`, both using the CLOCK algorithm. By default, EGOS will run with your write-back cache. If you want to run it with your write-through cache instead, you'll need to edit one line in `apps/blocksvr.c` to call `wtclockdisk_init` instead of `clockdisk_init` (this will be explained shortly). The implementation of each **should be less than 100 lines of code.**

4.1 Preparation

Before you start, you need to understand the block store abstraction in EGOS. Reading the following files will help you.

- `src/h/egos/block_store.h` contains the definition of the block store interface (`struct block_store`) and the initialization functions for all currently-available block stores.
- `src/apps/blocksvr.c` is the "block server," a kernel process for disk access. The `block_init` function is where `clockdisk` is initialized as a layer between the "bottom" layer (a physical storage device) and the filesystem layer (currently `treedisk`). This is also where you can change whether EGOS uses the write-back or write-through cache: If you change `clockdisk_init` to `wtclockdisk_init`, the cache layer will be your write-through cache.

Next, you can start to read `src/block/wtclockdisk.c`. It contains the definitions of `block_info` and `wtclockdisk_state`. `block_info` contains the metadata for a single cache slot, as described in the previous section. `wtclockdisk_state` is the "state structure" for wtclockdisk's implementation of the block store interface, and should contain all the data structures needed for implementing the write-through disk cache with the CLOCK algorithm. The function `wtclockdisk_init` creates a "`wtclockdisk`" instance of the block store interface, and initializes all of `wtclockdisk_state`'s data structures.

Similarly, you should read `src/block/clockdisk.c`, which is where you will implement the write-back cache; it has an analogous `clockdisk_state` that should contain all the data structures needed for implementing the write-back cache.

4.2 Let's Start Coding!

You will need to implement 4 functions in `src/block/wtclockdisk.c`:

```
wtclockdisk_read(bi, ino, offset, *block)
wtclockdisk_write(bi, ino, offset, *block)
wtclockdisk_setsize(bi, ino, nblocks)
cache_update(*cs, ino, offset, *block)
```

and 5 functions in `block/clockdisk.c`:

```
clockdisk_read(bi, ino, offset, *block)
clockdisk_write(bi, ino, offset, *block)
clockdisk_setsize(bi, ino, nblocks)
clockdisk_sync(bi, ino)
cache_update(*cs, ino, offset, *block, dirty)
```

The `cache_update` function is where you will implement the CLOCK algorithm. It should be called every time there is a cache miss, and its arguments identify the block that should go in the cache and provide its contents. This function is not defined in the released version of `wtclockdisk.c` and `clockdisk.c`, but we highly recommend you create it in order to centralize your cache eviction logic.

5 Hints and Submission

- There are a total of `cs->nblocks` slots in the disk cache.
- The memory for slot data is allocated at `cs->blocks`. This does not include any space for slot metadata; you should add a separate data structure for slot metadata to the `clockdisk_state` struct.
- Other than the memory for slot metadata, your implementation should not need to allocate any new heap memory.
- You should update `read_hit`, `read_miss`, `write_hit` and `write_miss` properly in your implementation. These are common metrics for evaluating an eviction algorithm. In your clockdisk, call `clockdisk_dump_stats` every 20 disk operations.
- Grading is based on `wtclockdisk.c` and `clockdisk.c`. You will get full credit if you implement CLOCK correctly and your disk cache doesn't trigger an assertion in checkdisk, which is a block store layer that checks the integrity of the underlying block store.