Write-Up for Project 2: File Caching Proxy

Adam Li *zli3@andrew.cmu.edu*

1 Caching Protocol between Server and Proxy

In this project, "Check-on-use" is used where clients chat to server upon `open()` to check if the file copy in the cache is up-to-date. If it is a cache miss or it is a cache hit but the version is not up-to-date, the proxy will download the file from proxy. Otherwise, if cache hit and the version is up-to-date, the client will directly interact with the file item in the local cache.

1.1 Reader Consistency Model

1.1.1 The Slow Reader Situation

Suppose we have two readers, Alice and Bob. Alice is a slow reader, after Alice opened file v1.0 in the cache for 2 minutes, Bob is also going to open the same file. On Bob's check on use, he found out the version for file on the server has already been updated, and he need to fetch the new version of this file, say file v2.0 from the server. This new file should not directly overwrite the one Alice is currently reading because according to open-close semantics, Alice should see the same file v1.0 from open to close.

1.1.2 Solution

To achieve the open-close semantic, and not overwriting the file of the previous version that is now being open, my design is to save the file with a version number suffix in the cache. This way, a single version would only have at most one read copy in this cache. For example, if the filename is A and version is 9, the filename would be `A_9`.
For garbage collection, i.e. collecting and cleaning up the stale read copy upon close or a new check-on-use, I used a reference counter `refCnt` for every file copy. If someone has opened this file, the counter will increment by 1, the count will be decremented by 1 upon close. So if the counter is 0, we know the file is not being referenced.
I also introduced a `isValid` flag for every file copy (cache block). The old version will be marked invalid immediately after a new version has been fetch to cache. Upon garbage collection, the cache will check if the cache block is not valid `AND` is not being open, if that is the case, the file will be deleted from cache.

1.2 Writer Consistency Model

1.2.1 The Multiple Writer Situation

When there are multiple concurrent writers, we should achieve the following effect to align with the open-close semantics:

1. When a writer is writing and not yet closed, the other concurrent reader should not see the real time modification made on the file.

2. When multiple writers are writing, their written file versions should not interfere with each other, and that the file content enventually should be the same with last writer to close.

1.2.2 Solution

The solution for this situation is to create a write copy for every writer, naming the file with the unique file descriptor for that open-close session. The naming format for write copy of file A, session file descriptor 13 is `A_write_13`.
Upon `close()`, the write copy will be write back to the local original file and the server file, the local file version suffix will be updated with the version number distributed by the server. A garbage collection function will take care of the above function and deleting the file on disk and from the LRU double linked list.

2 LRU Cache Implementation

The LRU cache in this project is implemented by

1. a concurrent hash map `cacheBlockMap`, mapping between the file path in cache (e.g. `A_9`) and the corresponding cache block.

2. a doubly linked list for maintaining the order of the MRU to LRU.

3. a concurrent hash map `pathVersion`, mapping between the file name without suffix and the most current version of file in cache.

Upon `close()` of the file, it will be moved to the head of the cache.

3 Chuncking

To ensure fetching of large files does not overflow the heap memory limit, I implemented chunking to fetch large files chunk by chunk, chunk size is setted to be 64000 bytes.

4 Concurrency

At server side, the concurrency is handled by the synchronized method with a master copy lock for every file on server. This is realized by:
`private final Map<String, Object> masterCopysMap;`
On the cache side, only the action of putting new things into the cache is synchronized, realized by adding synchronized keywords before methods.