Project 2
COM S 362
Fall 2021

1. Introduction

Database Management Systems (DBMS) such as PostgreSQL, MySQL or MongoDB are generally constructed with a layered architecture, with the lowest layer being a storage engine. The purpose of the storage engine is to provide a low-level API for reading and writing records to persistent storage (e.g., the disk). High-level database features are provided on top of the low-level storage engine.

In this project you will construct a storage engine with a very simple API, it supports only four functions.

```
int open_storage(char* filename);
int read_storage(int key, char* block);
int write_storage(int key, char* block);
int close_storage();
```

The storage engine reads and writes fixed size blocks of data from and to disk. The blocks have a size of `BLOCK_SIZE` (512-byte) and they are indexed by a key that ranges from 0 to `NUM_BLOCKS-1`. What they database does with these blocks of data is not our concern, the only purpose of the storage engine is to provide the reliable persistence of blocks.

A fully working version of the storage engine is supplied to you in the file simple-se.c. The challenge of this project is to improve the performance and reliability of the storage engine. Specifically, you will modify the engine to allow safe and efficient multi-threaded calls to the API functions. Also, you will add a cache and batching of reads and writes. Importantly, the storage engine must be fault tolerant, meaning it must survive a power outage or system crash at any moment and be able to restart in a consistent state.

2. Project Setup

Your code must be written in C and compile and run on the pyrite Linux servers. The graders will only use pyrite when testing code. How or where you do your development is up to you, but it would be wise to test your code early and often on pyrite.

You are given a simple implementation of the storage engine called simple-se.c. Make a copy of this file called storage-engine.c, this is where you will make your implementation of the storage engine. The file storage-engine.h must be used as is, it cannot be modified in any way. You are

provided with client.c, which is for building tests, modifying it in any way to test the storage engine.

3. Storage Engine API

The storage engine only supports a simple API. You may not modify the API or the file storage-engine.h which defines it in any way.

```
int open_storage(char* filename);
```

Opens or creates a new persistent storage file. The filename is any file path. The storage engine will open an existing file if it exists or create a new one if it does not. On success returns 0, on failure returns -1.

```
int read_storage(int key, char* block);
```

Copies the data of a single block (of size BLOCK_SIZE) into the char* block. The key is an index which indicates which block to read. On success returns 0, on failure returns -1.

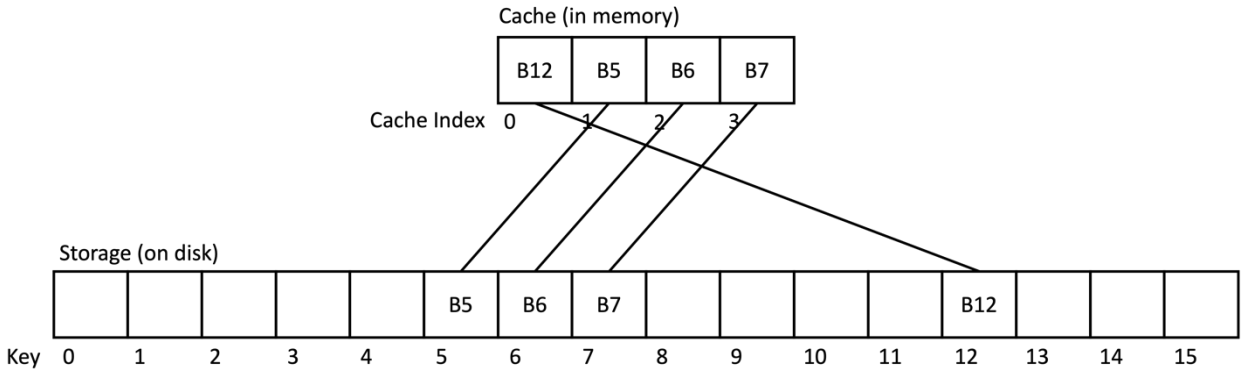
```
int write_storage(int key, char* block);
```

Stores the single block (of size BLOCK_SIZE) pointed to by char* block. The key is an index which indicates which block to write. On success returns 0, on failure returns -1.

```
int close_storage();
```

Closes the persistent storage file. After close, calls to read and write will be ignored. On success returns 0, on failure returns -1.

3. Version 1: Caching



Because of the slowness of reading and writing from and to disk, cache can provide a significant performance boost. For the project, implement a direct-mapped write-through cache. Direct-mapped means that each address can only be found in one possible cache location, the formula

for the cache index is simply `cache_index = key % CACHE_SIZE`. Write-through means that when a block is written, the write updates both the cache and the storage at the same time.

Define the cache size as 64. The cache itself should be kept as an array of a struct. We will add to the structure in the next sections, but for now a cache can be defined as follows.

```
#define CACHE_SIZE 64


struct cache_entry {
    int key;
    char data[BLOCK_SIZE];
};
struct cache_entry cache[CACHE_SIZE];
```

The `read_storage()` function should check if the requested key is the same as the key already loaded in cache. If so, the data can be copied from the cache entry. Otherwise, the cache entry must be updated from the file before copying the data. It is good to know the following way for efficiently copying data.

```
memcpy(data, cache[cache_index].data, BLOCK_SIZE);
```

All calls to `write_storage()` must update both the file and the cache entry with the passed key and data.

4. Version 2: Multi-Threading Requirements

The four API functions must be "thread safe", meaning multiple threads many be calling these functions concurrently. Consider the case of two concurrent calls: `read_storage(10, &buff1)` and `read_storage(74, &buff2)`. Assume the data for key 10 is loaded into cache entry 10. The second call to `read_storage()` needs to load key 74 into cache entry 10. If the second call is coping values into the cache entry's data at the same time as the first call is copying the data out a race condition can occur.

A very simple solution to provide thread safety is to completely lock a function, for example:

```
int read_storage(int key, char* block) {
    pthread_mutex_lock(&lock);
    // ...
    pthread_mutex_unlock(&lock);
}
```

This type of solution results in poor performance, because we are allowing only one thread to read at a time. Suppose two threads are reading the same key that is already loaded into the cache. Then they should be allowed to perform their reads concurrently. However, if one of the threads requires modifying the contents of the cache entry, it must not do so when other threads are reading from the cache entry.

The problem is very similar to the classic readers/writers problem. However, the "writer" in the problem can refer to anything that is modifying the cache entry, it could be either a call to `write_storage()` or for a different key than is currently in the cache `read_storage()`.

The requirement of this step is that synchronization is controlled independently for each cache entry. In other words, you should be adding variables/mutexes/condition variables/semaphores to `cache_entry`. Any concurrency mechanism provided by pthreads is allowed for the project. Spinning loops are not allowed.

```
struct cache_entry {
    int key;
    char data[BLOCK_SIZE];
    // Add additional fields for concurrency control here
};
```

5. Version 3: Write Batching

Suppose that `block[0]` to `block[4]` are blocks that need to be written to consecutive locations on disk, consider the following two scenarios.

```
// Version 1
// assuming O_SYNC is set, so no system buffering of writes
write(fd, cache[0].data, BLOCK_SIZE);
write(fd, cache[1].data, BLOCK_SIZE);
write(fd, cache[2].data, BLOCK_SIZE);
write(fd, cache[3].data, BLOCK_SIZE);
write(fd, cache[4].data, BLOCK_SIZE);

// Version 2
// copy the blocks into a buffer and then write the buffer to disk
char combined_buff[BLOCK_SIZE * 5];
for (int i=0; i<5; i++) {
    memcpy(combined_buff+i*BLOCK_SIZE, cache[i+start_key].data, BLOCK_SIZE);
}
// writes all blocks with a single call to write()
write_blocks(start_key, combined_buff, BLOCK_SIZE*5);
```

Even though version 2 requires the extra step of copying the blocks to a buffer, it is significantly faster (about 5 times when tested on pyrite) than version 1. Combining multiple writes into a single write is known as batching.

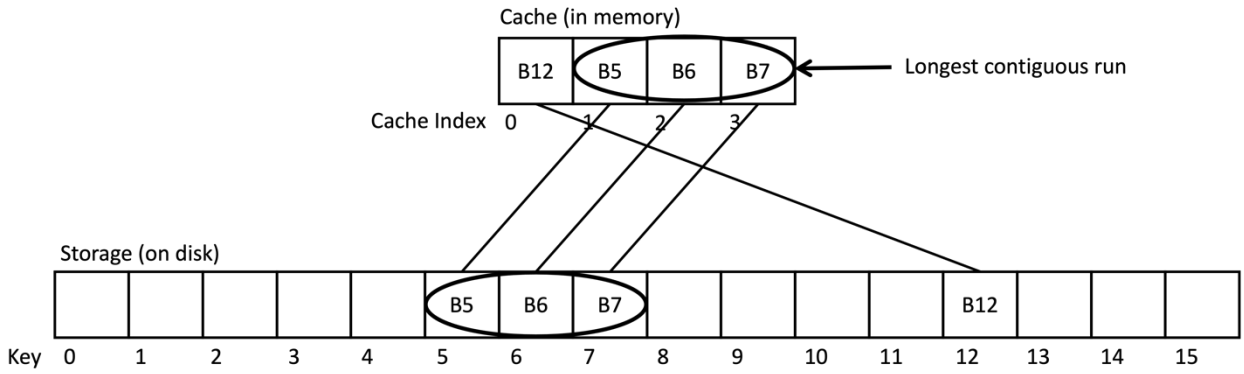
To implement batching, `write_storage()` no longer writes to the file, it instead writes pending information to the cache entry and signals to a batcher thread (described below) that a write is pending. Here is the modified `cache_entry`.

```
struct cache_entry {
    int key;
    char data[BLOCK_SIZE];

    bool write_pending;
    int pending_key;
    char* pending_data;

    // Add additional fields for concurrency control here
};
```

The batching policy is to always write the longest contiguous run of pending writes first. Searching for contiguous runs is simple when using direct caching, because the cache entries will also be contiguous. For example, see the diagram below.



In `open_storage()` create a thread that executes a method called `batcher()`. The thread is what `write_storage()` signals when a write is pending.

```
void* batcher(void* arg) {
    while(1) {
        // wait for at least one pending write

        // loop through the cache and find the largest contiguous set of keys

        // copy the pending data from that cache into a single buffer
        // see the code for Version 2 above

        // call write_blocks() to write the data

        // update all the cache entries with the data that was written

        // for each cache entry signal that the write is complete
    }
}
```

A call to `write_storage()` must not return until the batcher thread has completed the write. Returning early would result in inconsistency, for example, if the same thread called `read_storage()` with the same key it might be provided with the data from before the write.

If there are multiple writes to the same cache entries, only one is able to put its data as pending, the others must wait.

5. Testing

You must test your solution. Modify client.c to create tests. Think of different scenarios of the ordering of events. Adding calls to `sleep()` in `read_block()` and `write_blocks()` can cause these scenarios to occur.

5. Submission

You will submit your project on Canvas. Your program must compile and run without errors on `pyrite.cs.iastate.edu`.

Put all your source files (including the Makefile and the README file) in a folder. Then use command zip -r <your ISU Net-ID> <src_folder> to create a .zip file. For example, if your Net-ID is ksmith and project2 is the name of the folder that contains all your source files, then you will type zip -r ksmith project2 to create a file named ksmith.zip and then submit this file.

4.1. Documentation and Makefile (20 points)

Submit a **README** file containing a brief explanation of the functionality of each source file. Also describe the current state of the project.

Each source file must also be well-documented. There should be a paragraph or so at the beginning of each source file describing the code; functions and data structures should also be explained. Basically, another programmer should be able to understand your code from the comments.

Update the Makefile to compile your test code and include it in the submission. Be sure that make successfully builds the executable on pyrite.

4.1. Version 1 (40 points)

Submit version1.c that implements caching. If you are confident that all parts of the project are working you can submit only storage-engine.c.

4.2. Version 2 (20 points)

Submit version2.c that implements multi-threading. If you are confident that all parts of the project are working you can submit only storage-engine.c.

4.3. Version 3 (10 points)

Submit storage-engine.c that implements all requirements including batching.

4.3. Testing (10 points)

Submit client.c that demonstrates you have tested the code.