存储技术基础 期中KV Store项目说明

作业描述

本作业要求在给定 C++ 代码框架下，实现高效的并发 Key-Value 存储引擎，支持 KV 的基本操作：Read (Get)、Write (Put、Delete) 和 Range (Scan)(可选)等。

本作业要求引擎**基于NVM**进行设计与实现。你可以假定一块内存区域被映射到了NVM上（见下文），并使用该内存区域做非易失存储。作业提交后，我们将（可能）把项目运行在实际的NVM中做测试。

框架代码见网络学堂或此链接。

基本要求

1. **每组2-3人**，完成代码实现工作。
2. 基于课程提供的代码框架进行程序开发(修改 engine_race.{cc|h} 文件)，实现 Read、Write 接口。
3. 要求实现**多线程并发正确**的版本，保证线性一致性(Linearizability)：即并发读写时，写请求一旦返回，其更改需要体现在接下来所有的读请求中。
4. 要求 Key-Value 引擎保证**崩溃一致性(Crash Consistency)**：Write操作过程中机器崩溃，需保证Write要么完全没发生，要么完全发生。Write 成功返回之后，保证该操作插入的键值对被持久化，即使机器重启后也不会丢失。
5. 通过框架提供的正确性和性能测试程序(test/ 与 bench/ 目录下)。你可以添加更复杂的测试代码，以保证系统的正确性。
6. 要求**以小组为单位**提交**实验代码**和**实验报告**。报告内容需包括 KV 设计、实现细节，线程并发安全的实现，崩溃一致性的实现，性能结果与分析及思考题等。**报告需注明每位同学所负责的部分。**

作业详细说明

代码的实现部分

框架

KV 框架在压缩代码包 engine.zip 中，可从网络学堂下载。编译运行环境为 Linux，大家自行搭建虚拟机、docker、云环境或 Windows Subsystem for Linux 环境。

目录树介绍

engine.zip 解压后有如下文件: 其中 engine_example 里是一个参考版本的 KV 存储引擎。engine_race 目录中包含 engine_race.{cc|h}, 这是本次作业中需要修改的目录和文件。test 中包含正确性测试代码，bench 中包含性能测试代码。

接口

```
// name 为存储引擎的数据路径, 初始化存储引擎,  返回指针到*eptr
RetCode EngineRace::Open(const std::string& name, Engine** eptr);

// 将<key, value>插入存储引擎,  如果 key 已经存在,  则该操作为更新
RetCode EngineRace::Write(const PolarString& key, const PolarStrin
g& value)

//根据 key 在存储引擎中索引数据,  返回数据到value 变量
RetCode EngineRace::Read(const PolarString& key, std::string value
)
```

其中 PolarString 是一个封装的字符串类，详见 include/polar_string.h。

接口定义在 include/engine.h 和 engine_race/engine_race.{cc|h}。更详细的接口语义可阅读 test/single_thread_test.cc。

EngineRace::Open传入的name是**文件名（而非目录）**。你需要对其 open 再 mmap 得到一段内存来使用。

编译

编译时，执行 make 命令(如果需要编译 engine_example 中的参考代码，执行 make TARGET_ENGINE=engine_example)。编译完成后在 lib 目录下生成静态链接库 libengine.a。

正确性验证

测试正确性时，进入 test 目录，执行./build.sh 来编译测试程序， 执行./run_test.sh 来运行测试程序。现提供的三个测试程序比较简单：

- single_thread_test 测试单线程正确性
- multi_thread_test 测试多线程情况下的正确性
- crash_test 测试进程被 kill 后的系统正确性

你需要为系统添加更全面的验证脚本，并保证系统通过所有验证。

性能测试

bench/ 目录中提供了性能测试程序，执行./build.sh 来编译，执行./bench 来运行测试程序。./bench 程序有三个参数：

- thread_num：并发执行的线程个数，
- read_ratio：Read 操作的比例，
- isSkew：key 的分布 (0 时为均匀分布，1 时为 zipfan 分布)

bench 程序中 key 的大小固定为 8 bytes，value的大小固定为 16 bytes。

备注

对test/和bench/目录下现有文件的任何修改都需要在报告中指明。

你可以为test/和bench/*增加*更多的文件（例如额外的测试，更丰富的性能评测等），并在报告中说明。

* 实验环境的搭建：模拟NVM设备

> 通过编译指令调整宏 MOCK_NVM 的定义，切换真实NVM设备与模拟的NVM设备（内存）。

在完成实验时，大家手头上没有NVM设备，测试脚本会传入位于ramdisk的 /tmp/ramdisk/test-xxx 文件，以模拟NVM用。
在验证代码时，我们（可能应该）会使用真实设备运行提交的代码，测试脚本会传入代指真实NVM的/dev/dax0.0设备。

默认情况下编译，会使用模拟设备，具体编译和运行方法见 README.md 。

为此，大家在运行、测试前需要挂载ramdisk到 /tmp/ramdisk/ 目录上。

挂载ramdisk

也可见 参考链接

```
# 创建 ramdisk
sudo mkdir /tmp/ramdisk
sudo chmod 777 /tmp/ramdisk
sudo mount -t tmpfs -o size=4g myramdisk /tmp/ramdisk
```

最好关闭系统的swap，否则 ramdisk 会发生page swap导致性能不稳定。

编程

```
RetCode EngineRace::Open(const std::string& name, Engine** eptr);
```

name 传入的是文件名：在模拟实验时指向 ramdisk 文件，在真实实验时指向 NVM 设备文件。
不要对该文件名做目录：你无法把它当做目录并创建子文件。

一般地，使用该文件的方法是 open & mmap ，得到一段内存后对其直接读写。

报告的思考题部分（任选1-2问）

1. 如何保证和验证 Key Value 存储引擎的 Crash Consistency? 考虑如下 Crash 情况:
    a. KV 崩溃(进程崩溃)
    b. 操作系统崩溃
    c. 机器掉电
2. 基于SSD和HDD的键值存储系统比基于NVM更需要考虑如何高效地完成IO操作。目前 KV 对外存读写数据的方式有以下几种，他们对KV的整体吞吐、IO利用率和内存使用率有何差异?
    a. 系统调用 read、write、fsync
    b. 系统调用 mmap、msync
    c. 异步 IO 框架 libaio、io_uring 和 SPDK
3. PMDK是intel开发的persistent memory编程工具库。调研PMDK下列功能的接口和内部实现，并说明他们和memory的实现有何区别
    a. memory allocator
    b. transaction
    c. lock

4. Write-ahead log（WAL）和 Copy on Write（CoW） 是NVM中保证崩溃一致性的常用手段。假设你实现了一个*单线程*的NVM上的链表。链表的每个key都是8 Byte的，而value是4KB的。
   a. put操作如果原地更新value，put操作在崩溃时是原子的吗？为什么？
   b. 请分别简述如何保证该链表的崩溃一致性：简述系统处理 put 时的每个操作是什么；说明系统在任一操作前、后崩溃后都可保证一致性。
      i. 使用 WAL 来保证
      ii. 使用 CoW 来保证
   c. 如何将该链表拓展为线程安全的实现？

附加题部分(可选)

附加题的要求：

- 实现该拓展功能，并展示接口和使用上的demo
- 为该拓展添加正确性测试
- 为该拓展添加性能测试

实现范围查找接口 Range

```
RetCode Range(const PolarString& lower, const PolarString& upper,
Visitor &visitor)
```

实现的思考与权衡（供参考）：

1. 你会为Range操作提供怎么样的一致性保证？Linearizability？Snapshot Read？Read Committed？
2. 不同的一致性保证有何区别？他们在性能上有何优劣？
3. 何为多版本并发控制？

实现 Snapshot

思考与权衡（供参考）：

1. snapshot是什么？它与backup有何区别？
2. 应该为snapshot设计怎么样的接口？（可参考LevelDB、RocksDB的接口）
3. 如何实现snapshot？
4. 探讨snapshot的性能与开销：
   a. 产生snapshot的延迟有多大
   b. snapshot是否会对前台的读写造成影响
   c. snapshot会带来多少额外的存储开销

评分标准

评分仅供参考，会根据作业的具体表现而调整。

大类	小类	分值	备注
代码与实现 50%	运行性能	20%	越高越好
	多线程可扩展性	5%	越高越好
	崩溃一致性	10%	通过测试
	并发控制与并发安全	10%	通过测试
	代码风格	5%	
报告 45%	清晰准确	15%	
	崩溃一致性叙述	10%	叙述正确且具体
	并发控制与并发安全叙述	10%	叙述正确且具体
	优化技巧与创新性	5%	
	思考题	5%	
互评 5%	互评性能与正确性	5%	
总体		100% +10%	

对在某一方面做得特别突出的项目（例如附加题或某项优化），有最多+10%的额外加分。

参考资料

基于PM的数据结构、KV引擎研究：

1. Youmin Chen, et al. "FlatStore: An efficient log-structured key-value storage engine for persistent memory". ASPLOS 20（基于NVM的KV系统设计 FlatStore）
2. Deukyeon Hwang, et al. "Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree". FAST 18 （基于NVM的B+树设计 FastFair）
3. Pengfei Zuo, et al. "Write-optimized and high-performance hashing index scheme for persistent memory". OSDI 18（基于NVM的hash table设计 Level Hash）
4. Shimin Chen, et al. "Persistent B+-Trees in Non-Volatile Main Memory". VLDB 15（基于NVM的B+树设计）
5. Se Kwon Lee, et al. "Recipe: Converting concurrent DRAM indexes to persistent-memory indexes". SOSP 19（通用的将并发安全数据结构转为NVM数据结构的方法 Recipe）
6. Moohyeon Nam, et al. "Write-Optimized Dynamic Hashing for Persistent Memory". FAST 19（基于NVM的hash table设计 CCEH）
7. Amirsaman Memaripour, et al. "Pronto: Easy and Fast Persistence for Volatile Data Structures". ASPLOS 20（将内存数据结构转为NVM数据结构的帮助库 Pronto）

*** NVM性能相关的研究报告：（与性能调优相关）**

1. Jian Yang, et al. **An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.** FAST 19（经典的叙述intel NVM性能的文章）
2. Joseph Izraelevitz, et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. ArXiv（十分详细、全面的NVM性能评价。与上一篇文章出自同一团队，相当于拓展版）
3. Björn Daase, et al. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. SIGMOD 21.（虽然标题看似与数据库相关，实则提供了NVM在多线程读写、跨NUMA结点读写时的性能）。

成熟的KV引擎设计：

1. https://github.com/google/leveldb.（基于 LSM Tree 的 KV 存储引擎）
2. https://github.com/facebook/rocksdb.（基于 LSM Tree 的 KV 存储引擎）
3. https://fallabs.com/kyotocabinet/.（基于 B+Tree 或 Hashtable 的 KV存储引擎）

框架、库、OS接口

1. https://spdk.io/（SPDK）
2. http://man7.org/linux/man-pages/man2/io_submit.2.html（Linux libaio）
3. https://lwn.net/Articles/776703/（Linux io_uring）
4. https://pmem.io/pmdk/（PMDK，持久性内存应用开发库）

基于SSD的KV引擎研究：

1. Lu, Lanyue, et al. “WiscKey: separating keys from values in SSD-conscious storage.” Proceedings of the 14th Usenix Conference on File and Storage Technologies. USENIX Association, 2016.
2. Raju, Pandian, et al.“Pebblesdb: Building key-value stores using fragmented log-structured merge trees. ”Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017.
3. Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key- value store. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP ’19).