M6: Key-Value数据库(libkvdb)

截止日期

6月16日(周日)晚23:59:59。

更新：key的长度不超过128字节；value的长度不超过16MiB。

收到的作业

引子

今天数据库(https://en.wikipedia.org/wiki/Database)无处不在。你一定听说过各大公司在后台都用数据库存储各种数据(例如你的剁手购物车，或者用户名和登陆信息)，也听说过“脱裤(http://www.freebuf.com/articles/database/29267.html)”——当你随手拿起你的手机的时候，你的联系人、应用数据……全部都存储在应用的内嵌数据库中。

数据库作为操作系统上的一个应用程序是如何实现的？

背景

如何为应用程序管理持久数据是个相当困难的问题，大家花了很多年才找到(部分的)答案，并且仍然在探索数据存储走向分布时的解决方案。一方面，我们可以让应用程序直接使用read, write, sync这些系统调用，但这么做会使高效可靠地保存结构化数据(例如为每个用户保存购物车)就会异常繁琐。

数据库就是这样的应用程序，例如应用非常广泛的SQLite(https://sqlite.org)，它甚至已经是Android, iOS, MacOS的一部分！它可以把数据库存储在一个文件中，通过SQL语言的接口访问它，例如可以使用一个指令：

```
SELECT salted_passwd_digest FROM users WHERE name = 'jyy'
```

获得jyy用户用于验证的密码信息。在这个实验里，我们仿照SQLite，实现一个最简单的“单文件数据库”，并且具有一个“库”应该有的稳定性。类似于M2 (libco) (OS2019_M2)，应用程序通过动态链接 `libkvdb.so` 调用其中的API进行数据的持久访问。在这个实验里，你使用文件API：read, write, lseek, sync实现实现磁

盘上的一个key-value数据库，类似 `std::map<std::string,std::string>` 。更重要的是，这个实验中，你要真真正正试图动态把一个程序“写对”——即便在各种并发、崩溃的情况下，你的数据库依然能够经受住考验。

编写经得起考验的程序

如果你的数据库系统部署在每一台Android/iOS设备上，你晚上还能好好睡觉么？在实际系统中丢失数据可不是闹着玩的。

造一个轮子很难，造一个好的轮子更难。软件的质量到今天仍然是一个open problem，软件中还有很多功能性的bug、安全漏洞、性能问题……

实验描述

获取实验代码与提交

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

在你原先的os-workbench基础上，执行

```
git pull origin M6
```

将本实验的框架代码下载到本地。在 `libkvdb/` 下编译能得到动态链接库 `libkvdb-32.so` 和 `libkvdb-64.so` 。

在这个实验中，我们实现持久、崩溃一致的key-value数据库API。`libkvdb.h` 声明了库中包含的函数：

```
struct kvdb {
  // 定义kvdb中保存的数据，例如一个文件描述符或一个FILE *。
  // 你可以自由修改这个结构体中的内容。
};
typedef struct kvdb kvdb_t;
int kvdb_open(kvdb_t *db, const char *filename);
int kvdb_close(kvdb_t *db);
int kvdb_put(kvdb_t *db, const char *key, const char *value);
char *kvdb_get(kvdb_t *db, const char *key);
```

在这个简化的实验中，我们假设 **key** 和 **value** 都只包含可打印非空白的**ASCII字符(例如字母、数字、下划线、标点符号)**。所有函数都应当具有容错功能：在正确时返回0，在出错时返回非0，其中：

- `kvdb_open` 打开 `filename` 数据库文件(例如 `filename` 指向 `"a.db"`)，并将信息保存到 `db` 中。如果文件不存在，则创建，如果文件存在，则在已有数据库的基础上进行操作。

- `kvdb_close` 关闭数据库并释放相关资源。关闭后的 `kvdb_t` 将不再能执行put/get操作；但不影响其他打开的 `kvdb_t` 。
- `kvdb_put` 建立 `key` 到 `value` 的映射，如果把 `db` 看成是一个 `std::map<std::string,std::string>` ，则相当于执行 `db[key] = value;` 。因此如果在 `kvdb_put` 执行之前 `db[key]` 已经有一个对应的字符串，它将被 `value` 覆盖。
- `kvdb_get` 获取 `key` 对应的value，相当于返回 `db[key]` 。返回的value是通过动态内存分配实现的(例如 `malloc` 或 `strdup` 分配的空间)，因此在使用完毕后需要调用 `free` 释放。

错误处理：

- 如果出现任何错误(文件无权限创建、文件无权限写入、已经关闭的 `db` 等)， `kvdb_open` ，`kvdb_close` ，`kvdb_put` 均返回非零值，返回0意味着操作成功执行。尽可能使你的程序在不非法修改 `kvdb_t` 结构体中数据的前提下不会有undefined behavior。
- 如果 `db` 不合法、内存分配失败或 `key` 不存在，则 `kvdb_get` 返回空指针。

为什么不要crash?

在平时我们写程序(例如OJ题)时，如果输入**不合法**，你程序的行为是 `undefined` ：它可能crash，可能输出不对的结果，也可能把你的机器炸了。但对库函数来说，这样的行为会给使用库的人造成麻烦，也可能留下安全隐患——If it crashes, it might be exploitable！看看黑客是怎样利用代码中的漏洞的(http://www.opensecuritytraining.info/Exploits1_files/SoftwareExploits_public.pdf)。

如果你的库被广泛地使用——嘿，那就要当心了——你的用户并不100%保证他调用库的方式是满足库的规约的。

在这个实验中，你面临的挑战是把事情做对：

- **线程安全**。同一个进程多个线程可以打开同一个数据库并发读写；
- **进程安全**。多个进程可以打开同一个数据库并发读写；
- **崩溃一致性**。访问数据库的进程可能随时被杀死；文件写入操作可能被缓存；系统可能崩溃……无论发生什么，当下次启动后再次打开数据库，总是能回到一个过去的一致状态，而且你应该竭尽所能使这个状态越接近崩溃的时间点越好(空的数据库总是一致的状态)。关于文件系统为应用程序提供的崩溃一致性，“All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications (https://www.usenix.org/sys-

tem/files/conference/osdi14/osdi14-paper-pillai.pdf)”是一篇很棒的论文。

对实现的约束

请大家在实现时只使用 `read`，`write`，`lseek`，`sync` 四个系统调用实现对文件的修改(其他如打开、关闭、上锁等对文件数据没有影响的操作我们并不记录)。这允许你使用libc中的诸如 `fread` 等函数。

我们的正确性测试将会拦截这些系统调用，并据此测试你数据库的正确性。具体来说，我们会模拟虚拟机/容器的崩溃，然后在重新启动后再次运行程序，使用 `kvdb_open` 打开数据库操作，因此你的数据库应该在打开时带有一致性数据恢复功能。不建议使用其他系统调用修改文件(例如 `mmap`)。

好奇我们怎么测试大家的代码?

我们会用类似我们研究工作 (http://moon.nju.edu.cn/spar/publication/jiang_crash_2016.pdf)中的技术测试你的代码。没错，如果你想到该怎么测试你的代码，你就具备了做计算机软件/系统研究的素质啦!

实验指南

只有一个 `libkvdb-xx.so`？

在 `make` 以后，你会得到一个 `libkvdb-xx.so` ，用 `file` 命令查看:

```
$ file libkvdb-64.so
libkvdb.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dy
```

直接运行的话(它居然可以运行)，你应该会得到一个Segmentation Fault。为了运行它，你需要编写另一个测试程序，链接libkvdb.so，例如 `test.c` :

```c
#include "kvdb.h"
#include <stdlib.h>

int main() {
  kvdb_t db;
  const char *key = "operating-systems";
  char *value;

  kvdb_open(&db, "a.db"); // BUG: should check for errors
  kvdb_put(&db, key, "three-easy-pieces");
  value = kvdb_get(&db, key);
  kvdb_close(&db);
  printf("[%s]: [%s]\n", key, value);
  free(value);
  return 0;
}
```

建议大家把测试代码的编译运行放进Makefile中，事半功倍。

实现线程安全

线程安全相当简单——只需要互斥锁就行了。互斥锁可以保存在 `kvdb_t` 的结构体中。

> **Tricky Cases**
>
> 注意进程是独立的“虚拟计算机”，每个进程都有自己的地址空间、文件描述符等。如果两个线程用 `kvdb_open(&db, "a.db")` 打开了同一个文件，它们就会被分配两把不同的锁！这个问题需要在“进程安全”中解决。

实现进程安全

我们需要在进程间上锁，防止两个进程同时访问某个资源。这时候，就需要进程间同步/通信的API了。理想情况下，我们可以以文件为单位进行进程间的并发控制，这样直接对数据库文件进行控制就好了。请阅读Wikipedia上的文档(https://en.wikipedia.org/wiki/File_locking)或搜索互联网。里面有链接通往The F**king Manual。

实现崩溃一致性

考虑一种数据结构的两种表示方法：

- 可以在存储系统中直接存储数据结构；
- 可以存储数据结构上所有的操作，这样可以保证在内存中重建数据结构。

如果采用第二种方案，崩溃一致性的实现就相对容易了——我们只需要保证append only write的正确性即可。这可以通过课堂上介绍过的journaling的方式实现。此外，我们可以在磁盘上存储key-value pairs，并且用journal更新，从而在实现崩溃一致性的同时，加快库的性能。

测试崩溃一致性

你可能做了一些很酷的崩溃一致性实现——例如在文件中的一部分作为journaling的区域。但是你怎么知道你的程序是否写对了呢？你可以在你的程序里插入一些“可能崩溃”的数据点，例如：

```
int kvdb_put(kvdb_t *db, const char *key, const char *value) {
  if (!check_db_open(db)) return -1;
  journal_write();
  __may_crash();
  data_write();
  __may_crash();
  sync();
}
```

然后这样实现：

```
void __may_crash() {
  int p = flip_a_coin();
  if (p == 0) {
    dump_db_file();
    exit(0); // crash
  }
}
```

反复运行程序，就能得到很多crash模拟得到的文件系统镜像。很快，你就能想到，`__may_crash` 还可以做得更彻底一点，例如你可以让malloc每次随机返回出错：

```
void *malloc() {
  if (flip_a_coin() == 0) return NULL;
  return real_malloc();
}
```

这样很可能能暴露出系统里的严重缺陷——这些都是你没有想过的dark corners，大家可以阅读OSDI2006的论文EXPLODE: A Lightweight, general system for finding serious storage system errors (https://web.stanford.edu/~engler/explode-osdi06.pdf)。

Key-Value数据库

Key-Value数据库看起来非常简单，你可能会怀疑它是否在实际中非常有用。毋庸置疑，复杂的查询语言SQL (关系型数据库，RDBMS)在实现复杂的业务逻辑方面带来的方便是无可比拟的——它是整个现代软件工业最重要的基础实施之一。然而，当我们的数据越来越大、必须跨越多台机器甚至多个数据中心存储、必须考虑机器故障时，更简单的数据模型提供了一个不错的折中：我们可以得到不错的性能、很好的容错和分布性，并且在大部分时候也能满足应用的需求。目前广泛在工业界里应用的一些Key-Value Store (及其变种)：Cassandra, HBase, Redis, Memcached, MongoDB等等。