NvmLab 实验报告

涂奕腾 2020201018

1.整体思想

如果不考虑持久化，KV 问题可以很方便地使用 std::map 解决，而 map 通过红黑树实现，所以一个比较自然的想法是自己实现一个持久化的二叉树/平衡树即可。但是此方法代码量较大比较难写，考虑是否可以直接利用 map 而不需要自行实现。由于只在 SET 中有中断，我们考虑只持久化 SET 操作的序列，并在 GET/NEXT 之前将操作序列恢复通过 map 实现 KV 映射即可。

2.实现(1)

代码在实现时，整体上参照了官方 github 中 queue 的实现。

在设计节点 node 的结构时，一开始我是想直接设置两个 string 分别表示 key 和 value，但是在拷贝时会出现问题，于是选择和官方 queue 实现中统一使用字符数组，另一方面为了减少 malloc 的较大时间开销，将 key 和 value 一同存储。初始化与结构定义如下：

```
POBJ_LAYOUT_BEGIN (stibiumt);
POBJ_LAYOUT_ROOT (stibiumt, struct root);
POBJ_LAYOUT_TOID (stibiumt, struct node);
POBJ_LAYOUT_TOID (stibiumt, struct queue);
POBJ_LAYOUT_END  (stibiumt);
struct node{
    char ch[K + V];
};
struct queue{
    int pos;
    TOID(struct node) lst[];
};
struct root{
    TOID(struct queue) queue;
};

map<string, string> state;

. . . . . .

if (file_exists(filename) != 0)
        pop = pmemobj_create(filename, POBJ_LAYOUT_NAME (stibiumt), N * 128, 0666);
else{
    pop = pmemobj_open(filename, POBJ_LAYOUT_NAME(stibiumt));
    do_not_dump = true;
}
```

```
TOID(struct root) root = POBJ_ROOT(pop, struct root);
struct root *rootp = D_RW(root);
```

插入序列的操作，我是通过 TX_BEGIN() {...} TX_END()保障原子性，经过测试，TX_ADD_DIRECT 在本题中并不需要：

```
static int push(PMEMobjpool *pop, struct queue *q, const string key, const string &val){
    int ret = 0;
    TX_BEGIN(pop){
        int pos = q->pos;
        TOID(struct node) tmp = TX_NEW(struct node);
        memcpy(D_RW(tmp)->ch, (key + val).c_str(), K + V);
        //TX_ADD_DIRECT(&q->lst[pos]);
        q->lst[pos] = tmp;
        //TX_ADD_DIRECT(&q->pos);
        q->pos++;
    }TX_END
    return ret;
}
```

对于第一次的 SET、GET/NEXT 应当特殊考虑，在第一次 SET 时应当新建序列；第一次 GET/NEXT 时则应该建立 map 映射：

```
//新建队列
static int qconstruct(PMEMobjpool *pop, void *ptr, void *arg){
    struct queue *q = (struct queue *)ptr;
    q->pos = 0;
    pmemobj_persist(pop, q, sizeof(*q));
    return 0;
}
static int qnew(PMEMobjpool *pop, TOID(struct queue) *q, int nentries){
    return POBJ_ALLOC(pop, q, struct queue, sizeof(struct queue)
        + nentries * sizeof(TOID(struct node)), qconstruct, &nentries);
}

qnew(pop, &rootp->queue, N);

//映射
for(int i = 0; i < D_RO(rootp->queue)->pos; ++i){
    string kv(D_RO(D_RO(rootp->queue)->lst[i])->ch);
    state[kv.substr(0, 16)] = kv.substr(16, 128);
}
```

在此方法下对于每一个 SET 操作都要进行一次 TX_NEW 的内存分配，常数大，在第 5、6 个测试点会超时。

3.实现(2)

受到分级页表的启发，考虑在每个 node 中存储多个 KV 对以降低 TX_NEW 的时间消耗，另外根据实际运行情况，将插入序列的函数改为类成员函数，略快于单独实现该函数。

```
struct node{
    char ch[S][K + V];
};

struct queue{
    int pos;
    TOID(struct node) lst[];

    int push(PMEMobjpool *pop, const string &key, const string &val){
        int i = pos / S, j = pos % S;
        TX_BEGIN(pop){
            if(!j){
                TOID(struct node) tmp = TX_NEW(struct node);
                memcpy(D_RW(tmp)->ch[0], (key + val).c_str(), K + V);
                lst[i] = tmp;
            }
            else{
                memcpy(D_RW(lst[i])->ch[j], (key + val).c_str(), K + V);
            }
            ++pos;
        }TX_END
        return 0;
    }
};
```

根据调参可得，一个节点包含的 KV 数 S 在一定范围内越小得分越高，最终将其设置为 32。完整代码如下：

```
#include "mian.h"

#include <fstream>
#include <iostream>
#include <libpmemobj.h>
#include <map>
#include <stdio.h>
#include <string.h>
#include <string>
#include <unistd.h>
#include <vector>
using namespace std;

const int K = 16;
```

```
const int V = 128;
const int S = 32#include "mian.h"

#include <fstream>
#include <iostream>
#include <libpmemobj.h>
#include <map>
#include <stdio.h>
#include <string.h>
#include <string>
#include <unistd.h>
#include <vector>
using namespace std;

const int K = 16;
const int V = 128;
const int S = 64;
const int N = 1E6 + 10;

POBJ_LAYOUT_BEGIN (stibiumt);
POBJ_LAYOUT_ROOT (stibiumt, struct root);
POBJ_LAYOUT_TOID (stibiumt, struct node);
POBJ_LAYOUT_TOID (stibiumt, struct queue);
POBJ_LAYOUT_END  (stibiumt);

bool do_not_dump = false;

struct node{
    char ch[S][K + V];
};

struct queue{
    int pos;
    TOID(struct node) lst[];

    int push(PMEMobjpool *pop, const string &key, const string &val){
        int i = pos / S, j = pos % S;
        TX_BEGIN(pop){
            if(!j){
                TOID(struct node) tmp = TX_NEW(struct node);
                memcpy(D_RW(tmp)->ch[0], (key + val).c_str(), K + V);
                lst[i] = tmp;
            }
            else{
                memcpy(D_RW(lst[i])->ch[j], (key + val).c_str(), K + V);
            }
```

```
            ++pos;
        }TX_END
        return 0;
    }

};

struct root{
    TOID(struct queue) queue;
};

map<string, string> state;

static int qconstruct(PMEMobjpool *pop, void *ptr, void *arg){
    struct queue *q = (struct queue *)ptr;
    q->pos = 0;
    pmemobj_persist(pop, q, sizeof(*q));
    return 0;
}

static int qnew(PMEMobjpool *pop, TOID(struct queue) *q, int
nentries){
        return POBJ_ALLOC(pop, q, struct queue, sizeof(struct queue)
        + nentries * sizeof(TOID(struct node)), qconstruct, &nentries);
}

static inline int file_exists(char const *file) { return access(file,
F_OK); }

void mian(std::vector<std::string> args){
    auto filename = args[0].c_str();
    PMEMobjpool *pop;
    if (file_exists(filename) != 0)
        pop = pmemobj_create(filename, POBJ_LAYOUT_NAME (stibiumt), N *
256 , 0666);
    else{
        pop = pmemobj_open(filename, POBJ_LAYOUT_NAME(stibiumt));
        do_not_dump = true;
    }
    if (pop == NULL){
        std::cout << filename << std::endl;
        perror("pmemobj_create");
        return;
    }

    TOID(struct root) root = POBJ_ROOT(pop, struct root);
```

```
    struct root *rootp = D_RW(root);

    Query q = nextQuery();
    switch (q.type) {
        case Query::SET:
            qnew(pop, &rootp->queue, N);
            D_RW(rootp->queue)->push(pop, q.key, q.value);
            break;
        case Query::GET:
            for(int k = 0; k < D_RO(rootp->queue)->pos; ++k){
                int i = k / S, j = k % S;
                    string kv(D_RO(D_RO(rootp->queue)->lst[i])->ch[j]);
                    state[kv.substr(0, 16)] = kv.substr(16, 128);
            }
            if(state.count(q.key))
                q.callback(state[q.key]);
            else
                q.callback("-");
            break;
        case Query::NEXT:
            for(int k = 0; k < D_RO(rootp->queue)->pos; ++k){
                int i = k / S, j = k % S;
                    string kv(D_RO(D_RO(rootp->queue)->lst[i])->ch[j]);
                    state[kv.substr(0, 16)] = kv.substr(16, 128);
            }
            if(auto it = state.upper_bound(q.key); it!= state.end())
                q.callback(it->first);
            else
                q.callback("-");
            break;

        default:
            throw std::invalid_argument(std::to_string(q.type));
    }


    while (1) {
        q = nextQuery();
        switch (q.type) {
        case Query::SET:
            D_RW(rootp->queue)->push(pop, q.key, q.value);
            break;
        case Query::GET:
            if(state.count(q.key))
                q.callback(state[q.key]);
            else
```

```
                q.callback("-");
            break;
        case Query::NEXT:
            if(auto it = state.upper_bound(q.key); it!= state.end())
                q.callback(it->first);
            else
                q.callback("-");
            break;

        default:
            throw std::invalid_argument(std::to_string(q.type));
        }
    }

    pmemobj_close(pop);
}
;
const int N = 1E6 + 10;

POBJ_LAYOUT_BEGIN (stibiumt);
POBJ_LAYOUT_ROOT (stibiumt, struct root);
POBJ_LAYOUT_TOID (stibiumt, struct node);
POBJ_LAYOUT_TOID (stibiumt, struct queue);
POBJ_LAYOUT_END  (stibiumt);

bool do_not_dump = false;

struct node{
    char ch[S][K + V];
};

struct queue{
    int pos;
    TOID(struct node) lst[];

    int push(PMEMobjpool *pop, const string &key, const string &val){
        int i = pos / S, j = pos % S;
        TX_BEGIN(pop){
            if(!j){
                TOID(struct node) tmp = TX_NEW(struct node);
                memcpy(D_RW(tmp)->ch[0], (key + val).c_str(), K + V);
                lst[i] = tmp;
            }
            else{
                memcpy(D_RW(lst[i])->ch[j], (key + val).c_str(), K + V);
            }
```

```
            ++pos;
        }TX_END
        return 0;
    }
};

struct root{
    TOID(struct queue) queue;
};

map<string, string> state;

static int qconstruct(PMEMobjpool *pop, void *ptr, void *arg){
    struct queue *q = (struct queue *)ptr;
    q->pos = 0;
    pmemobj_persist(pop, q, sizeof(*q));
    return 0;
}

static int qnew(PMEMobjpool *pop, TOID(struct queue) *q, int nentries){
    return POBJ_ALLOC(pop, q, struct queue, sizeof(struct queue)
        + nentries * sizeof(TOID(struct node)), qconstruct, &nentries);
}

static inline int file_exists(char const *file) { return access(file, F_OK); }

void mian(std::vector<std::string> args){
    auto filename = args[0].c_str();
    PMEMobjpool *pop;
    if (file_exists(filename) != 0)
        pop = pmemobj_create(filename, POBJ_LAYOUT_NAME (stibiumt), N * 256 , 0666);
    else{
        pop = pmemobj_open(filename, POBJ_LAYOUT_NAME(stibiumt));
        do_not_dump = true;
    }
    if (pop == NULL){
        std::cout << filename << std::endl;
        perror("pmemobj_create");
        return;
    }

    TOID(struct root) root = POBJ_ROOT(pop, struct root);
    struct root *rootp = D_RW(root);

    Query q = nextQuery();
    switch (q.type) {
```

```
        case Query::SET:
            qnew(pop, &rootp->queue, N);
            D_RW(rootp->queue)->push(pop, q.key, q.value);
            break;
        case Query::GET:
            for(int k = 0; k < D_RO(rootp->queue)->pos; ++k){
                int i = k / S, j = k % S;
                    string kv(D_RO(D_RO(rootp->queue)->lst[i])->ch[j]);
                    state[kv.substr(0, 16)] = kv.substr(16, 128);
            }
            if(state.count(q.key))
                q.callback(state[q.key]);
            else
                q.callback("-");
            break;
        case Query::NEXT:
            for(int k = 0; k < D_RO(rootp->queue)->pos; ++k){
                int i = k / S, j = k % S;
                    string kv(D_RO(D_RO(rootp->queue)->lst[i])->ch[j]);
                    state[kv.substr(0, 16)] = kv.substr(16, 128);
            }
            if(auto it = state.upper_bound(q.key); it!= state.end())
                q.callback(it->first);
            else
                q.callback("-");
            break;

        default:
            throw std::invalid_argument(std::to_string(q.type));
    }


    while (1) {
        q = nextQuery();
        switch (q.type) {
        case Query::SET:
            D_RW(rootp->queue)->push(pop, q.key, q.value);
            break;
        case Query::GET:
            if(state.count(q.key))
                q.callback(state[q.key]);
            else
                q.callback("-");
            break;
        case Query::NEXT:
            if(auto it = state.upper_bound(q.key); it!= state.end())
```

```
                q.callback(it->first);
            else
                q.callback("-");
            break;

        default:
            throw std::invalid_argument(std::to_string(q.type));
        }
    }

    pmemobj_close(pop);
}
```