임베디드 시스템 실습 PA1

2016310936 우승민

PA1 의 목표는 key-value ssd 를 구현하는 것입니다. 우선 제가 사용하는 data management 방법을 간단하게 설명하겠습니다. metadata 에서는 총 3 가지 array 를 통해 data 의 정보를 확인합니다.

1. key_pos array : array 에 순서대로 사용되는 key 값을 입력 받아 key 의 existence 를 판단합니다.

2. key array : key_pos 와 동일한 index 를 사용함으로써 key 와 block 을 연결하는 역할을 합니다.

3. block array : block 의 state 를 unused/valid/invalid 값을 각각 0/1/2 로 저장하여 block 의 사용 유무를 판단합니다.

이 metadata 를 바탕으로 QEMU 에서 매 command 실행할 때마다 metadata 의 정보를 읽고 key_pos array 를 통해 key 의 existence 를 판단한 후 key array 를 통해 해당 key 와 연결된 block 의 위치를 찾고 kvssd.data 에서 해당 위치의 data 를 읽고 쓰는 역할을 수행합니다.

저는 key 의 값을 입력 받는 과정은 key 16 바이트를 모두 더하고 1 을 추가로 더해주었습니다.

➔ key(real) = key[0]+key[1]+key[2]+key[3]+1

자세한 내용은 코드와 함께 설명하겠습니다.

kvssd 의 init 함수입니다. fd 에는 kvssd.data 를 meta 에너느 kvssd.meta 를 사용하도록 하였습니다.

```
void goldfish_kvssd_init(void)
{
        struct goldfish_kvssd_state *s;

        s = (struct goldfish_kvssd_state *)g_malloc0(sizeof(*s));
        s->dev.name = "goldfish_kvssd";
        s->dev.base = 0;
        s->dev.size = 0x1000;
        s->dev.irq_count = 1;
        s->dev.irq = 15;
        s->kvssd_fd = open("/media/seungmin/Windows/emu-2.2-release/external/qemu/kvssd.data", O_RDWR | O_CREAT, 0777);
        s->kvssd_meta = open("/media/seungmin/Windows/emu-2.2-release/external/qemu/kvssd.meta", O_RDWR | O_CREAT, 0777);

        goldfish_device_add(&s->dev, goldfish_kvssd_readfn, goldfish_kvssd_writefn, s);


        register_savevm(NULL,
                        "goldfish_kvssd",
                        0,
                        KVSSD_STATE_SAVE_VERSION,
                        goldfish_kvssd_save,
                        goldfish_kvssd_load,
                        s);
}
```

QEMU 에서 사용하는 kvssd_state structure 입니다.

```
struct goldfish_kvssd_state {
        struct goldfish_device dev;
        int kvssd_fd;
        int kvssd_meta;
        uint32_t status; //Ready: 0, Busy: 1
        uint32_t cmd;   //Read: 0, Write: 1
        uint32_t key[4];                        //current
        uint32_t value_pos;
        uint32_t key_pos;
        uint32_t pbn;
        uint32_t wait;
        uint32_t kvssd_value[1024];             //current

        uint32_t kvssd_block[MAX_BLOCK];        //meta
        uint32_t kvssd_key[MAX_BLOCK];  //meta
        uint32_t kvssd_keypos[MAX_BLOCK];       //mtea
};
```

여기서 key[4], kvssd_value[1024], value_pos, key_pos 는 현재 QEMU 에서 사용하는 key 와 block 의 value 값을 읽고 쓰는 역할을 수행합니다. pbn 은 kvssd.data 에서 읽을 data 위치를 나타내고, wait 은 GET, EXIST 를 수행하였을 때 state 가 WAIT 상태로 된 것을 따로 표기하기 위해 사용하였습니다. 아래 3 가지 array 는 metadata 에 저장된 data 를 관리하는 용도로 사용됩니다.

```
static void goldfish_kvssd_meta_write(struct goldfish_kvssd_state *s) {
        pwrite(s->kvssd_meta, s->kvssd_block, sizeof(s->kvssd_block), 0);
        pwrite(s->kvssd_meta, s->kvssd_key, sizeof(s->kvssd_key), sizeof(s->kvssd_block));
        pwrite(s->kvssd_meta, s->kvssd_keypos, sizeof(s->kvssd_keypos), sizeof(s->kvssd_block)+sizeof(s->kvssd_key));
        fsync(s->kvssd_meta);
        return;
}

static void goldfish_kvssd_meta_read(struct goldfish_kvssd_state *s) {
        pread(s->kvssd_meta, s->kvssd_block, sizeof(s->kvssd_block), 0);
        pread(s->kvssd_meta, s->kvssd_key, sizeof(s->kvssd_key), sizeof(s->kvssd_block));
        pread(s->kvssd_meta, s->kvssd_keypos, sizeof(s->kvssd_keypos), sizeof(s->kvssd_block)+sizeof(s->kvssd_key));
        return;
}
```

metadata 의 data 를 읽거나 수정해야할 때 위의 함수들을 수행하여 갱신해줍니다.

본격적으로 read 와 write 함수를 설명하겠습니다.

```
static uint32_t goldfish_kvssd_read(void* opaque, hwaddr offset, uint32_t value)
{
        struct goldfish_kvssd_state* s = (struct goldfish_kvssd_state*)opaque;
        uint32_t temp;

        if ( offset < 0 ) {
                cpu_abort(cpu_single_env, "kvssd_dev_read: Bad offset %" HWADDR_PRIx "\n", offset);
                return 0;
        }

        switch (offset) {
                case KVSSD_STATUS_REG:
                        return s->status;
                case KVSSD_KEY_REG:
                        temp = 1;
                        int i=0;
                        for(i=0; i<4; i++) temp += s->key[i];
                        goldfish_kvssd_meta_read(s);
                        i = 0;
                        while(i<MAX_BLOCK){
                                if(s->kvssd_keypos[i] == temp){
                                        return 1;
                                }
                                i++;
                        }
                        return 0;
                case KVSSD_VALUE_REG:
                        temp = s->value_pos;
                        s->value_pos = (s->value_pos+1)%1024;
                        return s->kvssd_value[temp];
        };

        return 0;
}
```

read 함수에서는 command 가 STATUS_REG, KEY_REG, VALUE_REG 일 경우에 수행합니다. STATUS_REG 일 경우에는 현재 kvssd 의 status 를 return 하였습니다.

KEY_REG 를 read 할 경우에는 metadata 를 읽은 후에 입력 받은 key 값과 비교하여 존재유무를 판단합니다. 만약 keypos array, 입력 받았던 key 의 값을 저장하는 array 에 동일한 key 가 있으면 1 을 return 하고, 아니면 0 을 return 합니다.

VALUE_REG 을 read 할 경우에는 입력 받은 key 에 해당하는 value 값을 차례로 return 합니다.

다음으로 write 함수에 대해 설명하겠습니다. 우선 command 가 GET 일 경우입니다.

```
static void goldfish_kvssd_write(void* opaque, hwaddr offset, uint32_t value)
{
        struct goldfish_kvssd_state* s = (struct goldfish_kvssd_state*)opaque;
        int status;

        if ( offset < 0 ) {
                cpu_abort(cpu_single_env, "kvssd_dev_read: Bad offset %" HWADDR_PRIx "\n", offset);
                return;
        }

        uint32_t i=0;
        uint32_t j=0;
        uint32_t temp = 0;

        switch (offset) {
                case KVSSD_CMD_REG:
                        s->status = KVSSD_BUSY;
                        s->cmd = value;
                        switch(s->cmd) {
                                case CMD_GET:
                                        goldfish_kvssd_meta_read(s);
                                        temp = 1;
                                        for(i=0; i<4; i++) temp += s->key[i];
                                        i = 0;
                                        while(i< MAX_BLOCK){
                                                if(s->kvssd_keypos[i] == temp) break;
                                                i++;
                                        }
                                        if(i==MAX_BLOCK){
                                                s->pbn = 0;
                                                status = goldfish_kvssd_data_read(s);
                                                break;
                                        }
                                        else{
                                                s->pbn = s->kvssd_key[i];;
                                                status = goldfish_kvssd_data_read(s);
                                        }
                                        break;
```

kvssd 의 state 를 BUSY 로 바꾼 후 metadata 를 읽고 입력 받은 key 의 존재를 확인합니다.

만약 없으면(key_pos array 의 MAX_BLOCK 까지 비교) pbn 에 0 을 넣고 data read 를 수행하여 data 의 첫번째 값을 읽도록 수행하였습니다. (특별한 행동 지침이 없음)

key 가 존재하면, pbn 에 해당 key 와 연결된 block 위치 kvssd_key[i]값을 넣고 data read 를 수행하도록 하였습니다.

아래는 data_read, data_write 함수입니다. 저는 s->pbn 을 기준으로 data 의 위치를 정해주었습니다.

```
#define PAGE_SHIFT 12

static int goldfish_kvssd_data_read(struct goldfish_kvssd_state *s) {
        pread(s->kvssd_fd, s->kvssd_value, sizeof(s->kvssd_value), s->pbn << PAGE_SHIFT);

        return 1;
}

static int goldfish_kvssd_data_write(struct goldfish_kvssd_state *s) {
        pwrite(s->kvssd_fd, s->kvssd_value, sizeof(s->kvssd_value), s->pbn << PAGE_SHIFT)
        fsync(s->kvssd_fd);

        return 0;
}
```

다음으로 PUT Command 의 경우입니다. 가장 먼저 기존에 ket 를 입력 받은 적이 있을 경우입니다.

```
case CMD_PUT:
        goldfish_kvssd_meta_read(s);
        temp = 1;
        for(i=0; i<4; i++) temp += s->key[i];
        i = 0;
        j = 0;
        int check = 0;
        while(i< MAX_BLOCK){
                j = i;
                if(s->kvssd_keypos[i] == temp){
                        check = 1;
                        s->kvssd_block[s->kvssd_key[i]] = 2;
                        while(s->kvssd_block[j]) j++;
                        s->kvssd_block[j] = 1;
                        s->kvssd_key[i] = j;
                        break;
                }
                i++;
        }
        if(check == 1){
                goldfish_kvssd_meta_write(s);
                s->pbn = j;
                status = goldfish_kvssd_data_write(s);
        }
```

만약 입력 받은 key 와 동일한 key 가 이전에 있을 경우에는 해당 block 을 2(invalid) 시키고, 이후로 빈 block 을 찾은 후 새로 연결시켜줍니다. 그리고 metadata 를 update 해주고, 변경된 block 에 data 를 write 해줍니다.

동일한 key 가 없을 경우입니다. keypos array 의 빈 곳을 찾아보고, 만약 block 이 (MAX_BLOCK-4)만큼 (GC 기준점) 사용되지 않았으면, keypos 와 key, block 을 새로 연결해주고, metadat 를 업데이트한 후 data 를 입력해줍니다.

```
else{
        i = 0;
        j = 0;
        while(s->kvssd_keypos[i] && i<MAX_BLOCK) i++;
        while(j<MAX_BLOCK-4){
                if(s->kvssd_block[j] == 0){
                        s->kvssd_block[j] = 1;
                        s->kvssd_key[i] = j;
                        s->kvssd_keypos[i] = temp;
                        goldfish_kvssd_meta_write(s);
                        s->pbn = j;
                        status = goldfish_kvssd_data_write(s);

                        break;
                }
                j++;
        }
```

```
        if(j==MAX_BLOCK-4){
                i = 0;
                while(i<MAX_BLOCK){
                        if(s->kvssd_block[s->key[i]] == 2){
                                s->kvssd_block[s->key[i]] = 0;
                                s->kvssd_key[i] = 0;
                                s->kvssd_keypos[i] = 0;
                        }
                        i++;
                }
                i = 0;
                j = 0;
                while(s->kvssd_keypos[i] && i<MAX_BLOCK) i++;
                while(s->kvssd_block[j]) j++;
                if(j==MAX_BLOCK){
                        cpu_abort(cpu_single_env, "kvssd_block: exceed block size an

                }
                s->kvssd_block[j] = 1;
                s->kvssd_key[i] = j;
                s->kvssd_keypos[i] = temp;
                goldfish_kvssd_meta_write(s);
                s->pbn = j;
                status = goldfish_kvssd_data_write(s);
        }
}

break;
```

만약 GC 기준점인 (MAX_BLOCK-4) 까지 빈 block 이 없으면 전체 block 을 다시 돌면서 invalid 된 block 을 unused 로 바꾸어주고, 연결된 key 와 keypos 또한 0 으로 초기화 해줍니다.

GC 를 한 이후 unused 된 block 을 새로 할당하고, key 와 keypos 와 연결해주고, metadata 를 update 한 후 data 를 입력해줍니다. 만약 GC 를 한 이후에도 할당할 block 이 없다는 것은 모든 block 이 valid 하다는 것이므로 cpu_abort 를 일으키게 해주었습니다.

ERASE Command 입니다. 입력 받은 key 의 위치에 해당하는 block 을 찾은 후 invalid 시켜줍니다. 저는 추가적으로 keypos 또한 0 으로 초기화해주었습니다. (PUT 할 때 구분하기 위해)

```
        case CMD_ERASE:
                goldfish_kvssd_meta_read(s);
                temp = 1;
                i=0;
                for(i=0; i<4; i++) temp += s->key[i];
                i = 0;
                while(i<MAX_BLOCK){
                        if(s->kvssd_keypos[i] == temp) break;
                        i++;
                }
                s->kvssd_block[s->kvssd_key[i]] = 2;
                s->kvssd_keypos[i] = 0;
                goldfish_kvssd_meta_write(s);
                status = 0;



                break;
        case CMD_EXIST:
                goldfish_kvssd_meta_read(s);
                temp = 1;
                i =0;
                for(i=0; i<4; i++) temp += s->key[i];
                i = 0;
                while(i< MAX_BLOCK){
                        if(s->kvssd_keypos[i] == temp) break;
                        i++;
                }
                if(i==MAX_BLOCK) s->pbn = 0;
                else s->pbn = s->kvssd_key[i];
                status = 1;

                break;
        default:
                cpu_abort(cpu_single_env, "kvssd_cmd: unsupported command %d\n", s->cmd);
                return;
}
```

EXIST Command 입니다. 입력 받은 key 가 존재할 경우에는 pbn 에 현재 block 의 위치를 저장하였고 없으면 0 을 저장하였습니다. status 는 나중에 WAIT 로 바꾸기 위해 1 을 넣어주었습니다.

위에 경우에서 status 상황에 따라 s->status 의 값을 정해주었습니다. PUT Command 와 ERASE Command 일 때는 status 가 0 으로 되어 READY 상태가 되고, GET Command 와 EXIST Command 일 때는 status 가 1 으로 되어 WAIT 상태가 됩니다.

```
if ( status == 0 ) {
        s->status = KVSSD_READY;
        goldfish_device_set_irq(&s->dev, 0, 1);
}
else if ( status == 1 ) {
        s->status = KVSSD_WAIT;
        goldfish_device_set_irq(&s->dev, 0, 1);
}
break;
```

write 함수에서 Register 가 Command 가 아닐 경우입니다.

```
case KVSSD_KEY_REG:
        s->key[s->key_pos] = value;
        s->key_pos = (s->key_pos+1) % 4;
        break;
case KVSSD_VALUE_REG:
        s->kvssd_value[s->value_pos] = value;
        s->value_pos = (s->value_pos+1) % 1024;
        break;
case KVSSD_STATUS_REG:
        if(s->key[s->key_pos]==value){
                s->wait = s->wait +1;
                s->key_pos = (s->key_pos+1) %4;
        }
        if(s->wait==4){
                s->wait = 0;
                s->status = KVSSD_READY;
        }
        break;
```

KEY_REG 일 경우에는 s->key 와 s-> key_pos 를 통해 user 로부터 입력받은 key 를 s->key 로 옮겨주었습니다.

VALUE_REG 일 경우에는 s->kvssd_value 와 value_pos 를 통해 user 로부터 입력받은 data 를 s->kvssd_value 로 옮겨주었습니다.

STATUS_REG 일 경우에는 현재 key 가 일치하는지 key[0], key[1], key[2], key[3] 4 회 확인하면서 wait 값을 변화시켜 wait 가 4 가 되었을 때 READY STATE 로 변하게 해주었습니다.

이상으로 QEMU code 설명은 마치겠습니다.

다음은 제가 test 하기 위해 사용한 user code 입니다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>

#define CMD_PUT                 3
#define CMD_GET                 4
#define CMD_ERASE               5
#define CMD_EXIST               6

struct ioctl_env {
        uint32_t key[4];
        uint32_t value[1024];
};

int main(int argc, char *argv[])
{
        int fd;
        int ret;
        struct ioctl_env env;

        fd = open("/dev/kvssd", O_RDWR | O_NDELAY);
        int i;

        for(i=0; i<1024; i++){
                env.key[0] = i;
                env.key[1] = 0;
                env.key[2] = 0;
                env.key[3] = 0;

                for(int j=0; j<1024; j++) env.value[j] = i+j;
                ioctl(fd, CMD_PUT, &env);
        }

        for(int j=0; j<1024; j++) env.value[j]=0;

        for(i=0; i<1024; i++){
                env.key[0] = i;
                env.key[1] = 0;
                env.key[2] = 0;
                env.key[3] = 0;

                ioctl(fd, CMD_GET, &env);
                printf("%d ", env.value[0]);
        }
        printf("\n");
```

```c
        for(int j=0; j<1024; j++) env.value[j]=0;

        for(i=0; i<1024; i++){
                env.key[0] = i;
                env.key[1] = 0;
                env.key[2] = 0;
                env.key[3] = 0;

                ioctl(fd, CMD_GET, &env);
                printf("%d ", env.value[0]);
        }
        printf("\n");

        for(i=0; i<1024; i++){
                env.key[0] = i;
                env.key[1] = 0;
                env.key[2] = 0;
                env.key[3] = 0;

                ioctl(fd, CMD_EXIST, &env);
        }

        for(i=0; i<1024; i++){
                env.key[0] = i;
                env.key[1] = 0;
                env.key[2] = 0;
                env.key[3] = 0;

                ioctl(fd, CMD_ERASE, &env);
        }

        for(i=0; i<1024; i++){
                env.key[0] = i;
                env.key[1] = 0;
                env.key[2] = 0;
                env.key[3] = 0;

                ioctl(fd, CMD_GET, &env);
                printf("%d ", env.value[0]);
        }
        printf("\n");


        return 0;
}
```

실행 결과입니다. 현재 상황은 temp 는 GET 하려는 key 값이고, ERASE 한 key 위치를 읽으려고 하기 때문에 pbn 이 0 으로 출력 되었습니다. 오른쪽 화면은 GET 이 실패하였음으로, 마지막으로 env.value[0]에 저장되었던 1023 이 출력되는 것입니다.