Crash Consistency

2016311821 한승하

마지막 과제 설명에 들어가기 앞서, 저는 이번 과제 base 코드를 지난 PA2의 db.c 와 db.h 파일을 사용하였음을 알려드립니다. Db_open, db_put, 등 다수의 부분의 코드가 PA2와 동일하며 이번 보고서에서는 Crash Consistency Implementation을 위해 수정한 부분 위주로 기술하였습니다.

<Main.c>

```
posix_fallocate(log_fd, 0, 4096*256*256);  // 256MB
printf("DB log file opened\n");

char* openstr = (char*)malloc(sizeof(char)*6);
strcpy(openstr,"OPEN\n");
memcpy(log_buf,openstr,sizeof(char)*5);
if(write(log_fd,log_buf,BUF_SIZE));
fsync(log_fd);
free(openstr);
recovery(DB);
while(1)
{
        if(end) break;
        wr = 0;
        while(read(0,buf,1))
        {
                if(*buf == '\n') break;
                bf_pars[wr++] = *buf;
        }
        if(!strncmp(bf_pars,"GET",3))
        {
                key_wr = 0;
                for(i=0;i<wr;i++) if(bf_pars[i] == '[') break;
                i++;
                while(i<=wr)
                {
                        if(bf_pars[i] == ']') break;
                        key[key_wr++] = bf_pars[i++];
                }
                key[key_wr] = '\0';
                key_len = strlen(key);
                val = db_get(DB,key,key_len,&val_len);
                if (val == NULL)
                {
                        printf("GETOK [%s] [NULL]\n",key);
                }
                else
                {
                        printf("GETOK [%s] [%d]\n",key,*(int*)val);
                        free(val);
                }
```

```
                else if(!strncmp(bf_pars,"PUT",3))
                {
                        key_wr = 0;
                        val_wr = 0;
                        for(i=0;i<wr;i++) if(bf_pars[i] == '[') break;
                        i++;
                        while(i<=wr)
                        {
                                if(bf_pars[i] == ']') break;
                                key[key_wr++] = bf_pars[i++];
                        }
                        key[key_wr] = '\0';
                        for(;i<wr;i++) if(bf_pars[i] == '[') break;
                        i++;
                        while(i<=wr)
                        {
                                if(bf_pars[i] == ']') break;
                                value[val_wr++] = bf_pars[i++];
                        }
                        value[val_wr] = '\0';
                        key_len = strlen(key);
                        val_len = strlen(value);
                        printf("PUTOK\n");
                        db_put(DB,key,key_len,value,val_len);
                }
                else if(!strncmp(bf_pars,"DB_CLOSE",8)) break;
        }

        db_close(DB);
        close(log_fd);
        printf("DB closed\n");
        return 0;
}
```

Main.c는 위와 같이 구성되어 있습니다. Log_file의 시작지점을 표시하기 위해 OPEN을 표시해 주었고, recovery함수를 호출하여 log_file tracking을 통해 data를 복구할 수 있게 해 주었습니다. (이는 Crash Consistency 2 part 에서 자세히 기술해 놓았습니다.)

이후 while문을 돌며 new_line 단위로 input을 받아, 마치 PA4에서 client의 요청을 처리하던 server와 같이 동작하며, GET, PUT, DB_CLOSE에 대한 동작들을 할 수 있도록 하였습니다.

<Crash Consistency 1>

Crash Consistency 1은 Hash table에 100개의 key-value pair가 존재해 disk로 flush하는 과정에서의 Consistency보장입니다.

저는 아래와 같은 방법을 사용하였습니다.

```
if(wordcount == 100) //flush when there is 100 key-value pair
{
        for(i=0;i<dbsize;i++)
        {
                if(db[i].left != NULL) save_and_free(db[i].left,i);
                if(db[i].right != NULL) save_and_free(db[i].right,i);
                db[i].left = NULL;
                db[i].right = NULL;
        }
        new_file_name++;
        int get_newname = open("./db/new_file_name",O_CREAT|O_RDWR,0777);
        char* filenum = (char*)malloc(sizeof(char)*10);
        sprintf(filenum,"%d",new_file_name);
        if(write(get_newname,filenum,strlen(filenum)));
        if(write(get_newname,"$",1));
        fsync(get_newname);
        close(get_newname);
        char* checkpoint = (char*)malloc(sizeof(char)*12);
        strcpy(checkpoint,"CHECKPOINT\n");
        memcpy(log_buf,checkpoint,sizeof(char)*11);
        if(write(log_fd,log_buf,BUF_SIZE));
        fsync(log_fd);
        free(checkpoint);
        wordcount = 0;
}
```

```
void db_close(db_t *db)
{
        int i;
        char* checkpoint = (char*)malloc(sizeof(char)*12);
        for(i=0;i<dbsize;i++)
        {
                if(db[i].left != NULL) save_and_free(db[i].left,i);
                if(db[i].right != NULL) save_and_free(db[i].right,i);
        }
        new_file_name++;
        int get_newname = open("./db/new_file_name",O_CREAT|O_RDWR,0777);
        char* filenum = (char*)malloc(sizeof(char)*10);
        sprintf(filenum,"%d",new_file_name);
        if(write(get_newname,filenum,strlen(filenum)));
        if(write(get_newname,"$",1));
        fsync(get_newname);
        close(get_newname);
        strcpy(checkpoint,"CHECKPOINT\n");
        memcpy(log_buf,checkpoint,sizeof(char)*11);
        if(write(log_fd,log_buf,BUF_SIZE));
        fsync(log_fd);
        free(checkpoint);
        free(db);
}
```

위는 Hash table을 disk로 내려야 하는 두가지 상황, key-value pair가 100개가 존재할 경우와, DB_close로 인해 남아있는 Entry들을 Disk로 내려야 하는 상황입니다. Disk로의 기록은 save_and_free함수로 기록되며, 생성되는 파일의 개수가 많기 때문에, 한번의 flush에 하나의 new_file_name을 정해 flush가 완료된 이후에 이를 기록해 두도록 하였습니다. 만약 파일을 내리

던 중간에 Crash가 발생한다면, 새로운 new_file_name은 기록되지 않을 것이고, 내리던 data들은 다음 open시에 무시될 것입니다. Fsync를 사용하여 기록을 확인해 주었으며,

```
int get_newname = open("./db/new_file_name",O_RDONLY);
if(get_newname != -1)
{
        idx = 0;
        while(read(get_newname,buffer,1))
        {
                if(*buffer == '$') break;
                filenum[idx++] = *buffer;
        }
        filenum[idx] = '\0';
        new_file_name = atoi(filenum);
}
```

open에선 다음과 같은 과정으로 기록된 new_file_name을 가져옵니다. 이는 이후, db_get 혹은 flush상황의 키워드가 됩니다.

추가사항으로는 Crash Consistency 2를 위해 logfile에 CHECKPOINT를 기록해 주었습니다. 이는 new_file_name이 기록된 이후에 기록하게 하여, CHECKPOINT가 존재하는 경우에 그 이전에 기록된 log에 대한 data들은 consistency를 보장할 수 있게 하였습니다.

<Crash Consistency 2>

Crash Consistency 2는 log_file을 사용하여 구현하였습니다.

```
#define SECTOR_SIZE 512
#define BUF_SIZE (SECTOR_SIZE * 8)

typedef struct db {
        struct db *left;
        struct db *right;
        int num;
        char* word;
} db_t;

db_t *db_open(int size);
void db_close(db_t *db);
void db_put(db_t *db, char *key, int keylen, char *val, int vallen);
void db_put_with_no_log(db_t *db, char *key, int keylen, char *val, int vallen);
/* Returns NULL if not found. A malloc()ed array otherwise.
 * Stores the length of the array in *vallen */
char *db_get(db_t *db, char *key, int keylen, int *vallen);
void save_and_free(db_t *db,int idx);
int findw(db_t *db,char* key,int idx);
void recovery(db_t *db);
void* log_buf;
int log_fd, dir_fd;
```

먼저 db.h입니다. O_DIRECT flag로 인해 입출력 align을 맞추기 위하여, BUF_SIZE를 정의해 주었습니다.

Recovery 과정에서 db_put 사용으로 인해 log_file이 덧 쓰여 지는 일을 방지하기위해 동일한 동작을 하지만 log_file기록은 하지 않는 db_put_with_no_log 함수를 정의하여 사용하였으며, recovery함수를 사용하여 log_file을 통한 data 복구를 진행하였습니다.

Log_file 입출력시 사용할 log_buf와 log_fd, dir_fd를 선언해 주었습니다.

```
void db_put(db_t *db, char *key, int keylen, char *val, int vallen)
{
        int i;
        int idx =0;
        int value = atoi(val);
        char* key_buf = (char*)malloc(sizeof(char)*1024);
        strcpy(key_buf,"PUT$");
        strcat(key_buf,key);
        strcat(key_buf,"$");
        strcat(key_buf,val);
        strcat(key_buf,"$\n");
        memcpy(log_buf,key_buf,sizeof(char)*strlen(key_buf));
        if(write(log_fd,log_buf,BUF_SIZE));
        fsync(log_fd);
        free(key_buf);
```

log_file기록은 db_put opperation마다 다음과 같이 기록됩니다. 위와 같은 과정으로 log_file에는 한 BUF_SIZE당 PUTKEYVALUE$₩n 식의 기록이 진행되게 됩니다.

또한 위에서 설명한 것과 같이 시작시에 시작지점 탐색을 위한 OPEN기록과, file을 내린 후에 CHECKPOINT 기록이 있습니다.

```
void recovery(db_t *db)
{
        int i = 1;
        int rd,wr = 0;
        char buf[BUF_SIZE];
        char key[1024];
        char val[10];
        while(1)
        {
                lseek(log_fd,-i*BUF_SIZE,SEEK_END);
                if(read(log_fd,log_buf,BUF_SIZE));
                if(!strncmp((char*)log_buf,"OPEN",4)) break;
                if(!strncmp((char*)log_buf,"CHECKPOINT",10)) break;
                i++;
        }
```

```
        while(i>0)
        {
                lseek(log_fd,-i*BUF_SIZE,SEEK_END);
                if(read(log_fd,log_buf,BUF_SIZE));
                if(!strncmp((char*)log_buf,"PUT$",4))
                {
                        rd = 0;
                        wr = 0;
                        memcpy(buf,log_buf,sizeof(char)*BUF_SIZE);
                        while(rd<BUF_SIZE) if(buf[rd++] == '$') break;
                        while(rd<BUF_SIZE)
                        {
                                if(buf[rd] == '$') break;
                                key[wr++] = buf[rd++];
                        }
                        rd++;
                        key[wr] = '\0';
                        wr = 0;
                        while(rd<BUF_SIZE)
                        {
                                if(buf[rd] == '$') break;
                                val[wr++] = buf[rd++];
                        }
                        val[wr] = '\0';
                        db_put_with_no_log(db,key,strlen(key),val,strlen(val));
                }
                i--;
        }
}
```

Recovery는 위와 같이 진행됩니다. File을 역방향으로 탐색하여 종료지점에서 가장 가까운 CHECKPOINT 혹은 OPEN을 탐색합니다. CHECKPOINT를 찾았다면, 그 이전의 data들은 consistency가 보장되므로 고려하지 않습니다.

이후 다시 순방향으로 탐색하며 key, value값을 parsing합니다.

Parsing한 data를 db_put_with_no_log를 이용하여 기록해 줌으로써, data복구가 완료됩니다.

이 Recovery함수는 main문의 입력을 받기 직전에 사용되므로, 입력을 받는 Stage에서는 복구된 Database를 사용할 수 있습니다.

감사합니다.