PA2 Report

2016311821 한승하

PA2에 대한 Code Report입니다. 이번 Report는 지난번 과제에서 수정, 추가한 점 위주로 기술하였음을 말씀드립니다.

<Global Veriables>

```
#include "db.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
int dbsize;
int wordcount;
int new_file_name;
```

이번 pa2를 위해 선언해준 Global veriables입니다.

지난번 pa1때에 추가하여, system call들을 위한 header files, hash table에 존재하는 쌍 개수를 체크 하기 위한 wordcount 변수와 새로 만들어야 하는 file name을 나타내는 new_file_name 변수를 추가로 선언해 주었습니다.

<DB open>

(Code Screan shot은 다음 페이지에 첨부되어 있습니다.)

DB Open에서는 DB 폴더의 생성과, parsing 중간에 dump되어 프로그램이 꺼진 경우 최신 file_name을 가져오기 위한 작업을 추가해 주었습니다. DB 폴더의 생성은 fork를 통해 자식 process를 execl으로 mkdir syscall을 호출하여 생성할 수 있게 해주었습니다.

최신 file 탐색은 db folder에 대한 ls로 구현해 주었습니다. Fork를 통한 자식 process의 stdout을 dup2로 "lslist.txt"라는 file로 변환시켜 주었고, 이후 execl을 통해 ls한 file 목록을 lslist.txt에 저장하여 부모 process가 읽어올 수 있게 해 주었습니다.

부모 process는 list를 탐색하여 가장 큰 file name을 가져오고 (file name은 0,1,2 ... 순입니다) 저는 hash index마다 하나의 file을 생성하였기에, 이를 제가 만든 hash fuction의 역을 통해 새로 생

성될 file name의 기준 값을 알아내도록 하였습니다. 알아낸 값은 new_file_name에 저장되어 이전 file을 최신 file 부터 탐색과, 새로 생성될 file이 이어져 생성될 수 있게 해 주었습니다.

기준 값이란 hash size가 10일 경우 기준 값 1에는 file 0~9, 기준 값 2로는 file 10~19가 생성/반환될 수 있도록 설정해 놓은 기준치입니다.

```
db_t *db_open(int size)
{
        int i,tmp,idx = 0;
        char* buffer = (char*)malloc(sizeof(char));
        char* filenum = (char*)malloc(sizeof(char)*10);
        dbsize = size;
        wordcount = 0;
        new_file_name = 0;
        if(fork() == 0) execl("/bin/mkdir","mkdir","-p","db",NULL);
        else wait(NULL);
        if(fork() == 0)
        {
                int lslist = open("lslist.txt",O_CREAT|O_RDWR,0777);
                dup2(lslist,1);
                execl("/bin/ls","ls","db",NULL);
        }
        else wait(NULL);
        int lslist = open("lslist.txt",O_CREAT|O_RDWR,0777);
        while(read(lslist,buffer,1))
        {
                if(*buffer != '\n') filenum[idx++] = *buffer;
                else
                {
                        tmp = atoi(filenum);
                        if(tmp>new_file_name) new_file_name = tmp;
                        for(i=0;i<10;i++) filenum[i] = 0;
                        idx = 0;
                }
        }
        new_file_name = (new_file_name/dbsize) + 1;
        free(buffer);
        free(filenum);
        close(lslist);
        db_t *db = (db_t*)malloc(sizeof(db_t)*size);
        for(i=0;i<size;i++)
        {
                db[i].word = (char*)malloc(sizeof(char));
                *db[i].word = (char)94;
                db[i].left = NULL;
                db[i].right = NULL;


        }
        return db;
}
```

<DB GET>

(DB get과 DB put은 코드 길이가 길어 분할하여 기술하였습니다)

```
char *db_get(db_t *db, char *key, int key_len,
                        int *val_len)
{
        char *value = NULL;
        int idx = 0;
        char* rword = (char*)malloc(sizeof(char)*(key_len+1));
        char* num = (char*)malloc(sizeof(char)*8);
        char buffer[1];
        int i,j,tmp = 0;
        int found;
        db_t *find;
        for(i=0;i<key_len;i++) idx = idx + key[i];
        idx = idx%dbsize;
        if(key[0]>db[idx].word[0]) find = db[idx].right;
        else find = db[idx].left;
        while(find != NULL)
        {
                if(strcmp(key,find->word) == 0)
                {
                        value = (char*)malloc(sizeof(int));
                        memcpy((void*)value,(void*)(&find->num),sizeof(int));
                }
                if(key[0]>find->word[0]) find = find->right;
                else find = find->left;
        }
```

db_get의 앞부분입니다. Pa1에 이어 file에서의 read를 통한 탐색을 하기 위해 rword, num그리고 buffer를 선언해주었습니다. Hash function이 값들을 좀더 고르게 분산시키게 하기 위해 hash index (idx)를 구하는 과정을 바꿔주었습니다. 이후 hash table 내의 탐색과정은 pa1과 동일합니다.

(아래 설명에 대한 스크린샷은 다음페이지에 첨부되어 있습니다)

이후 현재 db폴더에 존재하는 file들을 역순으로 탐색합니다. 이는 new_file_name이 다음번 file로의 저장 시 만들어 질 file name의 기준값이며 현재 상태에서 가장 최근에 갱신된 file은 new_file_name -1 을 기준으로 하고 있는 file들 이기 때문입니다.

기준 값으로 부터 file_name을 이끌어 내는 방법은 다음과 같습니다. (기준값)*dbsize + hash_index 따라서 각 file로의 저장 시 hash index당 하나의 파일에 저장되도록 하였습니다.

각 file 안에는 (단어 숫자₩n)식으로 저장되어 있습니다. 따라서 각 단어를 띄어쓰기를 기준으로 받아오며, 받아온 단어가 key와 같을 경우 뒤에 있는 숫자를 읽어 그 값을 value에 기록해 줍니다. 값을 찾았을 경우 found를 1로 만들어 loop를 벗어날 수 있도록 하였습니다.

읽은 단어가 key가 아닐 경우 ₩n이 읽힐 때까지 읽어주어 다음 줄로 넘어갈 수 있도록 하였습니다. 모든 file을 다 읽을 때 까지 단어를 찾지 못하면, value값은 초기에 설정한 null로 반환됩니다. 이후 사용한 file들을 close하고 변수들을 free해 주었습니다.

```
        for(i=new_file_name-1;i>0;i--)
        {
                if(found) break;
                j = 0;
                if(value != NULL) break;
                char* filename = (char*)malloc(sizeof(char)*20);
                char* filenum = (char*)malloc(sizeof(char)*10);
                int searchf;
                strcpy(filename,"./db/");
                sprintf(filenum,"%d",(i*dbsize)+idx);
                strcat(filename,filenum);
                searchf = open(filename,O_RDONLY);
                if(searchf == -1){
                        free(filename);
                        free(filenum);
                        continue;
                }
                while(read(searchf,buffer,1))
                {
                        for(j=0;j<key_len+1;j++)
                        {
                                if(buffer[0] == ' '){
                                        break;
                                }
                                rword[j] = buffer[0];
                                if(read(searchf,buffer,1));
                        }
                        rword[j] = '\0';
                        if(strcmp(rword,key) == 0 && j != key_len+1)
                        {
                                for(j = 0;j<100;j++)
                                {
                                        if(read(searchf,buffer,1));
                                        if(buffer[0] == '\n') break;
                                        num[j] = buffer[0];
                                }
                                num[j] = '\0';
                                tmp = atoi(num);
                                value = (char*)malloc(sizeof(int));
                                memcpy((void*)value,(void*)(&tmp),sizeof(int));
                                found = 1;
                                break;
                        }
```

```
                        else while(buffer[0] != '\n') if(read(searchf,buffer,1));
                }
                for(j=0;j<key_len;j++)
                {
                        rword[j] = '\0';
                }
                free(filename);
                free(filenum);
                close(searchf);
        }
        free(num);
        free(rword);
        return value;
}
```

<DB PUT>

```
void db_put(db_t *db, char *key, int key_len,
                        char *val, int val_len)
{
        int i;
        int idx =0;
        for(i=0;i<key_len;i++) idx = idx + key[i];
        idx = idx%dbsize;
        if(wordcount == dbsize)
        {
                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++;
                wordcount = 0;
        }
```

DB Put의 앞부분입니다. DB Get과 마찬가지로 더 넓은 분포를 위해 hash function을 수정해 주었습니다.

Wordcount는 현재 DB에 저장되어 있는 word개수를 의미합니다. Word가 dbsize가 될 경우 db에 대해서 저장 및 free를 할 수 있는 Save_and_free 함수를 호출해 주었습니다. Free와 save가 모두 끝나면 new_file_name을 1 증가시켜, 다음 번 저장 때 기준값이 될 수 있게 해 주었습니다.

```
void save_and_free(db_t *db,int idx)
{
        if(db->left != NULL) save_and_free(db->left,idx);
        if(db->right != NULL) save_and_free(db->right,idx);
        char* filename = (char*)malloc(sizeof(char)*20);
        char* filenum = (char*)malloc(sizeof(char)*10);
        char* num = (char*)malloc(sizeof(char)*10);
        sprintf(num,"%d",db->num);
        strcat(num,"\n");
        strcpy(filename,"./db/");
        sprintf(filenum,"%d",(new_file_name*dbsize)+idx);
        strcat(filename,filenum);
        free(filenum);
        int file = open(filename,O_CREAT|O_WRONLY,0777);
        lseek(file,0,SEEK_END);
        if(write(file,db->word,strlen(db->word)));
        if(write(file," ",1));
        if(write(file,num,strlen(num)));
        free(filename);
        free(num);
        free(db->word);
        free(db);
        close(file);
}
```

위 함수에서 사용한 save_and_free 함수입니다. db_get때와 마찬가지로 new_file_name을 이용해

각 hash index당 file을 만들어주고, lseek(, ,SEEK_END)를 이용해 맨 뒤부터 이어 쓸 수 있도록 해 주었습니다. Write이 모두 끝나면 free를 통해 각 db node를 해지해 주었습니다.

```
int isthere = findw(db,key,idx);
if(isthere == 0)
{
        wordcount++;
        db_t *insert = (db_t*)malloc(sizeof(db_t));
        insert->word = (char*)malloc(sizeof(char)*(key_len+1));
```

이후 findw를 사용해 db에 해당 단어가 존재하는지 탐색합니다.

```
int findw(db_t *db,char* key,int idx)
{
        db_t *find;
        if(key[0]>db[idx].word[0]) find = db[idx].right;
        else find = db[idx].left;
        while(find != NULL)
        {
                if(strcmp(key,find->word) == 0)
                {
                        return 1;
                }
                if(key[0]>find->word[0]) find = find->right;
                else find = find->left;
        }
        return 0;
}
```

Find 함수는 다음과 같이 생겼으며, db의 해당 index에 해당 단어가 존재하는지 탐색합니다.

이후 db에 단어가 존재하지 않을 경우 wordcount를 증가시켜 저장된 단어 개수를 인식할 수 있게 해 주었습니다. 이후 과정은 pa1과 동일합니다.

<DB_Close>

```
void db_close(db_t *db)
{
        int i;
        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);
        }
        free(db);
        if(fork() == 0) execl("/bin/rm","rm","lslist.txt",NULL);
}
```

Db_close입니다. Pa1과 전체적으로 비슷하나, free이전에 남은 단어들을 file로 저장하기 위해 put 에서 사용한 save_and_free함수로 저장 밑 해지 시켜 주었습니다.

이후 open때 사용한 lslist는 저장할 필요성이 없으므로 fork및 execl을 이용해 삭제해 주었습니다.

<RESULT>

아래는 위에서부터 순서 대로 stdout으로, output.txt으로 출력했을 때의 결과 및 execution time 입니다.