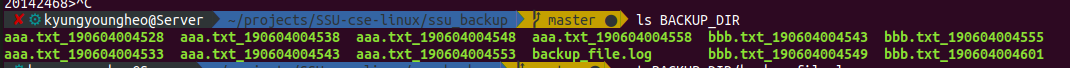
컴퓨터학부 20142468 허경영

1. 과제개요

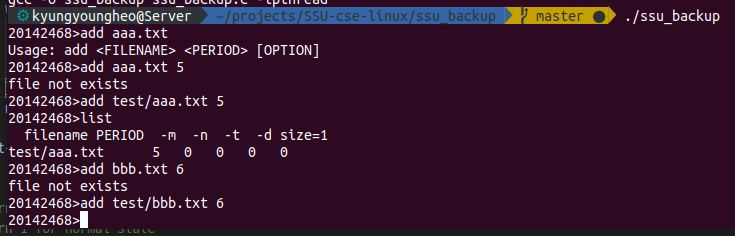
ssu\_backup은 파일경로와 옵션을 입력받아서 파일을 주기적으로 백업하는 프로젝트이다. 백업할 파일 하나마다 한 쓰레드가 백업을 담당하며, PERIOD 를 주기로 쓰레드가 sleep과 copy를 반복한다.

Remove 기능은 파일의 백업을 중지하는 기능으로, 쓰레드의 tid를 이용해 쓰레드를 종료시킨다. Compare 기능은 두 파일의 mtime과 크기를 이용해 파일이 같은지 확인하는 기능이다. Recover 기능은 백업한 파일을 복원하는 기능이다.

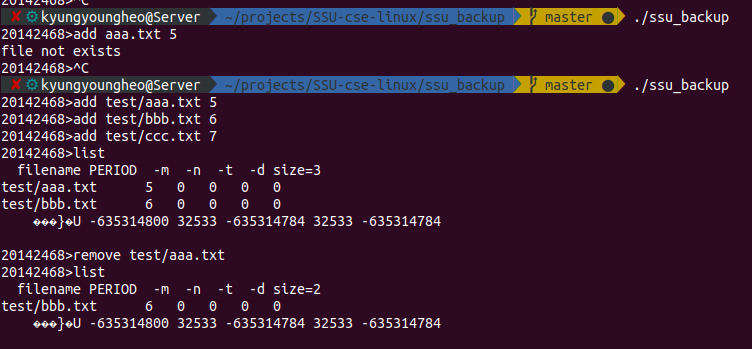
2. 실행결과

- ssu\_backup 프롬프트 동작

- add



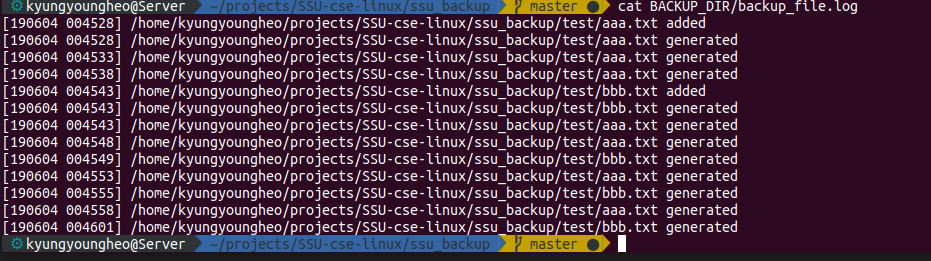
- add -m

- add -n

- add -t

- add -m -n -t

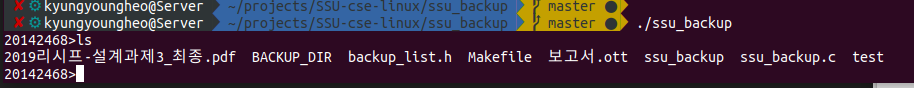
- backup.log



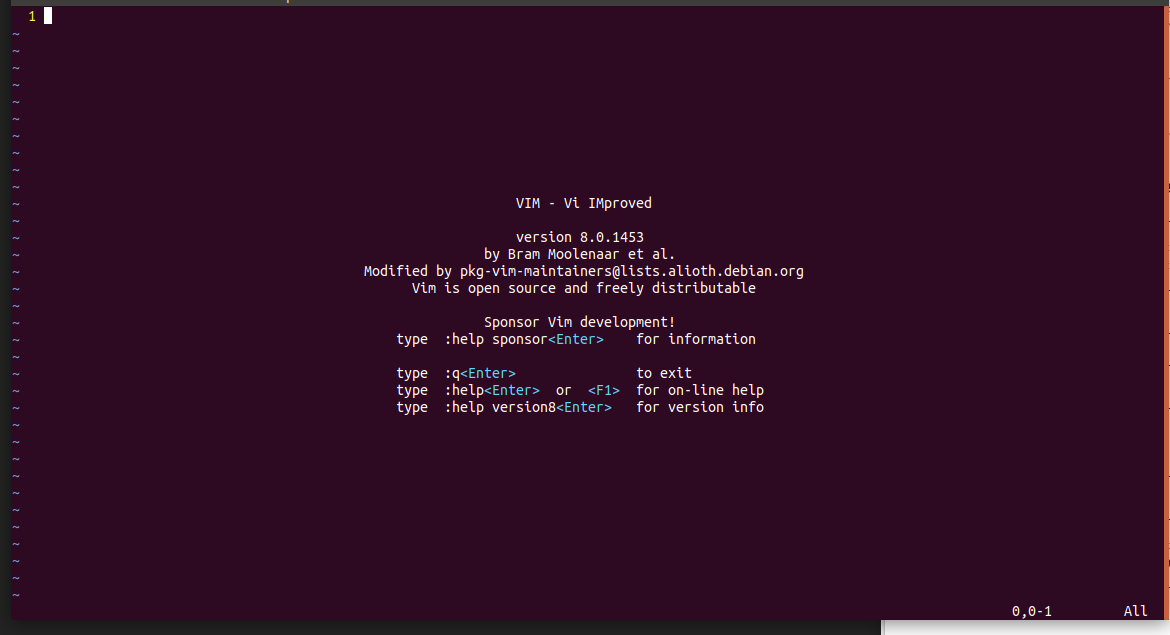
- compare

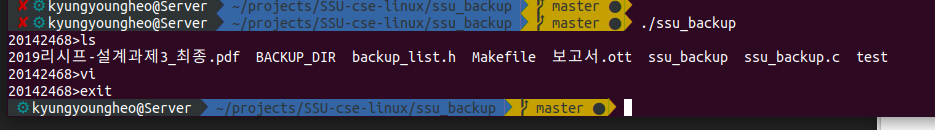
- remove

- remove -a

- list

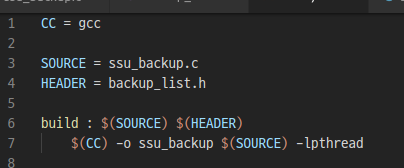
- vi

- ls



- exit

- makefile

3. 소스코드

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <getopt.h>

#include <pthread.h>

#include <errno.h>

#include <dirent.h>

#include <fcntl.h>

#include <ctype.h>

#include <time.h>

#include <sys/stat.h>

#include <sys/stat.h>

#include "backup\_list.h"

#define BACKUP\_DIR "BACKUP\_DIR"

/\*

// add 기능

\*/

void add(int argc, char \*\*argv);

/\*

// remove 기능

\*/

void remove\_list(int argc, char \*\*argv);

/\*

// compare 기능

\*/

void compare(int argc, char \*\*argv);

/\*

// list 기능

\*/

void list(void);

/\*

// 주어진 문자열이 몇개의 단어인지 계산하는 함수

\*/

int count\_words(char \*string);

/\*

// 주어진 문자열의 cmd를 구하는 함수

// cmd:

// add

// remove

// compare

// recover

// list

// ls

// vi

// vim

// exit

\*/

void getcmd(char \*string, char \*cmd);

/\*

// backup\_list를 update하고 쓰레드를 실행/종료하는 함수

\*/

void update\_thread(void);

/\*

// string을 argc, argv로 만드는 함수

\*/

void setup\_argv(char \*str, char \*\*argv);

/\*

// 사용법 출력하고 종료하는 함수

\*/

void print\_usage\_and\_exit(void);

/\*

// 정규파일, 디렉토리 확인하는 함수

\*/

int is\_reg\_or\_dir(struct stat statbuf, int opt);

/\*

// 쓰레드에서 실행할 함수

\*/

void \* thread\_func(void \*arg);

/\*

//

\*/

void make\_postfix(time\_t timer, char \*postfix1, char \*postfix2);

/\*

//

\*/

void write\_log(char \*pathname, int option);

/\*

//

\*/

void get\_filename\_only(char \*origin, char \*filename);

/\*

// 백업 디렉터리 생성하는 함수

\*/

void create\_backup\_dir(char \*pathname);

/\*

//

\*/

void delete\_timeout\_files(char \*backupname, int backup\_time);

/\*

// 파일 복사하는 함수

\*/

time\_t copy(char \*pathname1, char \*pathname2);

char backup\_pathname[256];

struct Backup\_list list\_head;

FILE \*log\_file\_fp;

int log\_file\_fd;

int main(int argc, char \*\*argv){

int cnt;

char inputbuf[256], cmd[10];

char \*argv\_param[256];

// 인자가 없는 경우

if(argc == 1){

create\_backup\_dir("./");

}

// 인자가 1개인 경우

else if(argc == 2){

create\_backup\_dir(argv[1]);

}

// 인자를 2개 이상 입력한 경우

else if(argc > 2){

// usage 출력 후 종료

print\_usage\_and\_exit();

}

while(1){

printf("20142468>");

fgets(inputbuf, 255, stdin);

// printf("%s\n", inputbuf);

getcmd(inputbuf, cmd);

setup\_argv(inputbuf, argv\_param);

cnt = count\_words(inputbuf);

// printf("cmd = %s, cnt = %d\n",cmd, cnt);

if(strcmp(cmd, "add") == 0){

add(cnt, argv\_param);

}

else if(strcmp(cmd, "remove") == 0){

remove\_list(cnt, argv\_param);

}

else if(strcmp(cmd, "compare") == 0){

compare(cnt, argv\_param);

}

else if(strcmp(cmd, "recover") == 0){

}

else if(strcmp(cmd, "list") == 0){

list();

}

else if(strcmp(cmd, "ls") == 0){

system("ls");

}

else if(strcmp(cmd, "vi") == 0 || strcmp(cmd, "vim") == 0){

system("vim");

}

else if(strcmp(cmd, "exit") == 0){

exit(0);

break;

}

else if(strcmp(cmd, "print") == 0){

print\_backup\_list(&list\_head);

printf("size=%d\n",list\_head.size);

}

// 백업 리스트의 정보를 가지고 쓰레드를 업데이트

// update\_thread\_from\_backup\_list();

}

exit(0);

}

int is\_reg\_or\_dir(struct stat statbuf, int opt){

// opt == 1 정규파일인지 확인

if(opt == 1){

if(!S\_ISREG(statbuf.st\_mode)){

printf("Not regular file\n");

return -1;

}

else

return 1;

}

// opt == 2 디렉토리인지 확인

else if(opt == 2){

if(!S\_ISDIR(statbuf.st\_mode)){

printf("Not directory\n");

return -1;

}

else

return 1;

}

return -1;

}

void add(int argc, char \*\*argv){

char \*pathname = argv[1];

char realpathname[1024];

int period;

int opt, opt\_number, opt\_time;

struct stat statbuf;

struct Node new\_node;

memset(&new\_node, 0, sizeof(struct Node));

// period 가 정수가 아닐 경우.

if(argc < 3){

printf("Usage: add <FILENAME> <PERIOD> [OPTION]\n");

return ;

}

for(int i=0; i<strlen(argv[2]); i++){

if(!isdigit(argv[2][i])){

printf("period should be Integer\n");

return ;

}

}

period = atoi(argv[2]);

realpath(pathname, realpathname);

// printf("pathname = %s %s\n",pathname, realpathname);

// 파일이 존재하지 않는 경우

if(access(pathname, F\_OK) != 0){

printf("file not exists\n");

return ;

}

if(5 > period || period > 10){

printf("period should be greater than or equal to 5 and less than or equal to 10\n");

return ;

}

// lstat

if(lstat(pathname, &statbuf) < 0){

fprintf(stderr, "lstat error\n");

return ;

}

// -d 옵션에서 처리해야함.

// // 정규파일이 아닌 경우

// if(!S\_ISREG(statbuf.st\_mode)){

// fprintf(stderr, "file is not regular file\n");

// return ;

// }

printf("before search func\n");

// 백업리스트에 이미 존재하는 경우

if(search\_backup\_list(pathname, &list\_head) != -1){

fprintf(stderr, "file already exist in backup list\n");

return ;

}

// argv[1] filename

// argv[2]가 5이상 10이하이면 period

// 아니면 option

// getopt

// update\_thread

// printf("add func\n");

// 옵션이 있는 경우

if(argc > 3){

while((opt = getopt(argc, argv, "mn:t:d")) != -1){

switch(opt){

case 'm':

if(is\_reg\_or\_dir(statbuf, 1) == -1) return;

// printf("add -m\n");

// mtime이 수정된 경우에만 백업 실행

strcpy(new\_node.pathname, pathname);

new\_node.interval = period;

new\_node.mtime = statbuf.st\_mtime;

new\_node.options[0] = 1;

// append\_backup\_list(new\_node, &list\_head);

break;

case 'n':

if(is\_reg\_or\_dir(statbuf, 1) == -1) return;

opt\_number = atoi(optarg);

// printf("add -n\n");

// 백업한 파일의 최대 갯수: opt\_number

strcpy(new\_node.pathname, pathname);

new\_node.interval = period;

new\_node.mtime = statbuf.st\_mtime;

new\_node.options[1] = 1;

new\_node.number = opt\_number;

// append\_backup\_list(new\_node, &list\_head);

break;

case 't':

if(is\_reg\_or\_dir(statbuf, 1) == -1) return;

for(int i=0; i<strlen(optarg); i++){

if(!isdigit(optarg[i])){

printf("period should be Integer\n");

return ;

}

}

opt\_time = atoi(optarg);

// printf("add -t\n");

strcpy(new\_node.pathname, pathname);

new\_node.interval = period;

new\_node.mtime = statbuf.st\_mtime;

new\_node.options[2] = 1;

new\_node.time = opt\_time;

// append\_backup\_list(new\_node, &list\_head);

break;

case 'd':

if(is\_reg\_or\_dir(statbuf, 2) == -1) return;

printf("add -d\n");

// append\_dir\_backup();

break;

case '?':

break;

}

}

}

// 옵션 없이 add 실행하는 경우

else {

strcpy(new\_node.pathname, pathname);

new\_node.interval = period;

new\_node.mtime = statbuf.st\_mtime;

// append\_backup\_list(new\_node, &list\_head);

}

append\_backup\_list(new\_node, &list\_head);

// print\_backup\_list(&list\_head);

write\_log(realpathname, 0);

// printf("before update\_thread\n");

update\_thread();

// printf("after update\_thread\n");

}

void remove\_list(int argc, char \*\*argv){

struct Node \*node;

char pathname[256];

int size = list\_head.size;

if(argc != 2){

printf("Usage: remove <FILENAME> [OPTION]\n");

return ;

}

if(strcmp(argv[1], "-a") == 0){

for(int i=0; i<size; i++){

node = get(i, &list\_head);

write\_log(node->pathname, 2);

pthread\_cancel(node->tid);

}

remove\_all(&list\_head);

print\_backup\_list(&list\_head);

// else{

// printf("Usage: remove <FILENAME> [OPTION]\n%s\n",argv[1]);

// return ;

// }

}

else{

strcpy(pathname, argv[1]);

int search\_res = search\_backup\_list(pathname, &list\_head);

if(search\_res == -1){

printf("File is not exist in backup list\n");

return ;

}

node = get(search\_res, &list\_head);

write\_log(node->pathname, 2);

pthread\_cancel(node->tid);

remove\_from\_list(pathname, &list\_head);

}

}

void compare(int argc, char \*\*argv){

char filename1[256], filename2[256];

struct stat statbuf1, statbuf2;

if(argc != 3){

printf("Usage: compare <FILENAME1> <FILENAME2>\n");

return ;

}

// 파일이 존재하지 않는 경우

if(access(filename1, F\_OK) != 0 || access(filename2, F\_OK) != 0 ){

printf("file not exists\n");

return ;

}

// lstat

if(lstat(filename1, &statbuf1) < 0){

fprintf(stderr, "lstat error\n");

return ;

}

// lstat

if(lstat(filename2, &statbuf2) < 0){

fprintf(stderr, "lstat error\n");

return ;

}

if(is\_reg\_or\_dir(statbuf1, 1) == -1 || is\_reg\_or\_dir(statbuf2, 1) == -1){

return;

}

if(statbuf1.st\_mtime == statbuf2.st\_mtime && statbuf1.st\_size == statbuf2.st\_size){

printf("%s and %s are same file\n", filename1, filename2);

}

else{

printf("filename=%s, mtime=%ld, filesize=%ld\n", filename1, statbuf1.st\_mtime, statbuf1.st\_size);

printf("filename=%s, mtime=%ld, filesize=%ld\n", filename2, statbuf2.st\_mtime, statbuf2.st\_size);

}

}

void list(void){

print\_backup\_list(&list\_head);

}

int is\_mtime\_changed(char \*pathname, struct stat originstat){

struct stat statbuf;

if(lstat(pathname, &statbuf) < 0){

fprintf(stderr, "lstat error\n");

return 0;

}

if(statbuf.st\_mtime != originstat.st\_mtime){

return 1;

}

else

return 0;

}

void update\_thread(void){

//backup\_pathname

struct Node \*np;

int size = list\_head.size;

pthread\_t tid;

for(int i=0; i<size; i++){

np = get(i, &list\_head);

// printf("np=%s, count = %d\n", np->pathname, np->saved\_count);

// To add Node from thread

if(np->saved\_count == 0){

pthread\_create(&tid, NULL, thread\_func, (void \*)np);

}

// To remove Node from thread

else if(np->saved\_count == -1){

// 쓰레드 종료.

pthread\_cancel(np->tid);

}

}

}

void \*thread\_func(void \*arg){

struct Node \*np = (struct Node \*)arg;

struct stat statbuf;

time\_t timer;

char yypostfix[10], hhpostfix[10];

char backup\_filename[256], filename\_only[512], writebuf2[512];

char realpath1[256];

int interval = np->interval, will\_copy = 0;

// 처음 생성될 때.

// 로그파일에 added 기록

// printf("thread func\n");

// printf("backup\_filename=%s\n",backup\_filename);

// save before mtime

if(lstat(np->pathname, &statbuf) < 0){

fprintf(stderr, "lstat error\n");

pthread\_exit(NULL);

}

while(1){

make\_postfix(timer, yypostfix, hhpostfix);

get\_filename\_only(np->pathname, filename\_only);

strcpy(backup\_filename, backup\_pathname);

strcat(backup\_filename, "/");

strcat(backup\_filename, filename\_only);

strcat(backup\_filename, "\_");

strcat(backup\_filename, yypostfix);

strcat(backup\_filename, hhpostfix);

// -m option

if(np->options[0] == 1){

int mtime\_flag = is\_mtime\_changed(np->pathname, statbuf);

// 파일이 수정되거나 처음 추가된 파일인 경우

if(mtime\_flag == 1 || np->saved\_count == 0){

will\_copy = 1;

// copy(np->pathname, backup\_filename);

// update stat

if(lstat(np->pathname, &statbuf) < 0){

fprintf(stderr, "lstat error\n");

pthread\_exit(NULL);

}

}

}

// -n option

if(np->options[1] == 1){

int max\_count = np->number;

if(np->saved\_count < max\_count){

will\_copy = 1;

// copy(np->pathname, backup\_filename);

}

}

// -t option

if(np->options[2] == 1){

// pathname로 시작하는 기존 백업 파일 찾기

// 백업 파일의 생성시간과 현재 시간을 비교

// np->time 이상이면 삭제

will\_copy = 1;

// copy(np->pathname, backup\_filename);

delete\_timeout\_files(np->pathname, np->time);

}

// -d option

if(np->options[3] == 1){

// 인자로 받은 디렉터리 탐색

// 디렉터리의 모든 파일을 append

// copy

}

else{

if(np->saved\_count == -1){

// timer = delete(np->pathname, backup\_pathname);

}

// 삭제할 때

// 로그파일에 deleted 기록

else{

// printf("path1=%s path2=%s\n", np->pathname, backup\_filename);

will\_copy = 1;

// copy(np->pathname, backup\_filename);

}

}

if(will\_copy == 1){

np->tid = pthread\_self();

copy(np->pathname, backup\_filename);

}

np->saved\_count++;

sleep(interval);

}

// 로그파일에 로그 남기기.

// fwrite()

realpath(np->pathname, realpath1);

pthread\_exit(NULL);

}

void delete\_timeout\_files(char \*backupname, int backup\_time){

struct dirent \*\*dentry;

struct stat statbuf;

int dircnt=0;

time\_t cur\_time = time(NULL);

char filename[256];

if((dircnt = scandir(backup\_pathname, &dentry, NULL, NULL)) == -1){

fprintf(stderr, "opendir: chdir error for %s\n",backup\_pathname);

exit(1);

}

chdir(backup\_pathname);

for(int i=0; i<dircnt; i++){

strcpy(filename, dentry[i]->d\_name);

if(strcmp(filename, "..") == 0 || strcmp(filename, ".") == 0)

continue;

// printf("file %d=%s\n", i, filename);

if(lstat(filename, &statbuf) < 0){

fprintf(stderr, "lstat error\n");

printf("%s\n",strerror(errno));

print\_usage\_and\_exit();

}

if(strstr(filename, backupname) != NULL){

if(statbuf.st\_ctime + backup\_time < cur\_time ){

int res = remove(filename);

if(res == 0){

// remove logging

// write\_log(filename, 1);

printf("file removed successfully\n");

}

else

printf("file not removed\n");

}

}

// printf("file %d=%s, ctime=%d, cur\_time=%d, backup\_time=%d\n", i, filename, statbuf.st\_ctime, cur\_time, backup\_time);

}

// chdir(backup\_pathname);

chdir("../");

}

time\_t copy(char \*pathname1, char \*pathname2){

int len;

char buf[513];

int fd1, fd2;

time\_t timer;

// char testpathname2[256] = "./BACKUP\_DIR/aaa.txt\_700101090000";

fd1 = open(pathname1, O\_RDONLY);

fd2 = open(pathname2, O\_RDWR | O\_CREAT | O\_TRUNC, 0777);

// printf("fd2 = %d\n",fd2);

write\_log(pathname1, 1);

while((len = read(fd1, buf, 512)) > 0){

write(fd2, buf, len);

// printf("len = %d buf = %s\n",len, buf);

}

close(fd1);

close(fd2);

return timer;

}

void write\_log(char \*pathname, int option){

char yymmdd[10], hhmmss[10];

char writebuf1[512], writebuf2[256];

time\_t timer = time(NULL);

make\_postfix(timer, yymmdd, hhmmss);

realpath(pathname, writebuf2);

sprintf(writebuf1, "[%s %s] %s ", yymmdd, hhmmss, writebuf2);

// add

if(option == 0){

strcat(writebuf1, "added\n");

}

// copy

else if(option == 1){

strcat(writebuf1, "generated\n");

}

// delete

else if(option == 2){

strcat(writebuf1, "deleted\n");

}

// recover

else if(option == 3){

strcat(writebuf1, "recovered\n");

}

// fwrite(writebuf1, )

// write()

write(log\_file\_fd, writebuf1, strlen(writebuf1));

}

void make\_postfix(time\_t timer, char \*postfix1, char \*postfix2){

struct tm \*cur\_time;

int year, mon, mday, hour, min, sec;

// char postfix[256];

char yymmdd[10], hhmmss[10];

timer = time(NULL);

cur\_time = localtime(&timer);

year = cur\_time->tm\_year;

mon = cur\_time->tm\_mon;

mday = cur\_time->tm\_mday;

hour = cur\_time->tm\_hour;

min = cur\_time->tm\_min;

sec = cur\_time->tm\_sec;

year += 1900;

year %= 100;

mon += 1;

sprintf(yymmdd, "%02d%02d%02d", year, mon, mday);

sprintf(hhmmss, "%02d%02d%02d", hour, min, sec);

strcpy(postfix1, yymmdd);

strcpy(postfix2, hhmmss);

// sprintf(postfix, "%s%s", yymmdd, hhmmss);

}

void get\_filename\_only(char \*origin, char \*filename){

int len = strlen(origin);

char buf[256], tmp;

int j=0;

// printf("%s\n",origin);

for(int i=len-1; i>=0; i--){

if(origin[i] == '/'){

printf("buf in for = %s\n",buf);

break;

}

buf[j++] = origin[i];

}

for(int i=0; i<j/2; i++){

tmp = buf[i];

buf[i] = buf[j-i-1];

buf[j-i-1] = tmp;

}

// printf("buf = %s\n",buf);

strcpy(filename, buf);

}

void setup\_argv(char \*str, char \*\*argv){

int len = strlen(str);

int j = 0, n = 0;

char buf[256];

for(int i=0; i<len; i++){

if(str[i] == ' ' || i == len-1){

buf[j] = '\0';

j=0;

// malloc

// printf("strlen %d\n",strlen(buf)+1);

argv[n] = (char \*)malloc(strlen(buf) +1);

strcpy(argv[n], buf);

// printf("buf = %s argv[%d] = %s %d\n", buf, n, argv[n], strlen(buf));

n++;

memset(buf, 0, 256);

}

else{

// printf("%c\n", str[i]);

buf[j++] = str[i];

}

}

// for(int i=0; i<n; i++){

// printf("argv[%d] = %s\n", i, argv[i]);

// }

return ;

}

void getcmd(char \*string, char \*cmd){

int len = strlen(string);

for(int i=0; i<len; i++){

if(string[i] == ' ' || string[i] =='\n'){

cmd[i] = '\0';

return;

}

cmd[i] = string[i];

}

}

int count\_words(char \*string){

int len = strlen(string);

int cnt = 0;

for(int i=0; i<len; i++){

if(string[i] == ' ')

cnt++;

}

return cnt + 1;

}

void create\_backup\_dir(char \*pathname){

struct dirent \*\*dentry;

struct stat statbuf;

// 디렉토리가 존재하지 않는 경우

if(access(pathname, F\_OK) != 0){

fprintf(stderr, "Directory not exist\n");

print\_usage\_and\_exit();

}

// lstat

if(lstat(pathname, &statbuf) < 0){

fprintf(stderr, "lstat error\n");

print\_usage\_and\_exit();

}

// 디렉토리가 아닌 경우

if(!S\_ISDIR(statbuf.st\_mode)){

fprintf(stderr, "pathname is not directory\n");

print\_usage\_and\_exit();

}

// 권한이 없는 경우

if(access(pathname, R\_OK) != 0 || access(pathname, W\_OK) != 0 || access(pathname, X\_OK) != 0){

fprintf(stderr, "Permission denied\n");

print\_usage\_and\_exit();

}

chdir(pathname);

mkdir(BACKUP\_DIR, 0777);

chdir(BACKUP\_DIR);

// log\_file\_fp = fopen("backup\_file.log", "w+");

log\_file\_fd = open("backup\_file.log", O\_WRONLY | O\_CREAT | O\_TRUNC, 0777);

// system("pwd");

strcpy(backup\_pathname, pathname);

strcat(backup\_pathname, BACKUP\_DIR);

chdir("..");

}

void print\_usage\_and\_exit(void){

// ./ssu\_backup 잘못된 인자로 실행시 사용법 출력 후 종료

printf("Usage: ./ssu\_backup [PATHNAME]\n");

exit(0);

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

struct Backup\_list{

struct Node \*head;

int size;

};

struct Node{

struct Node \*next;

char pathname[256];

int interval;

int options[4]; // 옵션 설정되면 1 아니면 0

int time; // -t 옵션

int mtime;

int number; // -n 옵션

int saved\_count;

pthread\_t tid;

};

void init(struct Backup\_list \*list){

list->head = NULL;

list->size = 0;

}

int search\_backup\_list(char \*pathname, struct Backup\_list \*list){

struct Node \*node = list->head;

int cnt=0;

if(node == NULL) return -1;

while(1){

// printf("search %s to %s\n",pathname, node->pathname);

if(strcmp(pathname, node->pathname) == 0)

return cnt;

if(node->next == NULL) break;

node = node->next;

cnt++;

}

return -1;

}

void append\_backup\_list(struct Node data, struct Backup\_list \*list){

struct Node \*node = list->head;

struct Node \*new\_node = (struct Node \*)malloc(sizeof(struct Node));

strcpy(new\_node->pathname, data.pathname);

new\_node->interval = data.interval;

for(int i=0; i<4; i++){

new\_node->options[i] = data.options[i];

}

new\_node->time = data.time;

new\_node->mtime = data.mtime;

new\_node->number = data.number;

new\_node->saved\_count = 0;

// printf("new\_node::pathname=%s append\nlist->size = %d\n", new\_node->pathname, list->size);

if(node == NULL){

list->head = new\_node;

// printf("first append\n\n");

}

else{

// printf("list->head = %d\n",list->head->interval);

while(1){

if(node->next == NULL) break;

node = node->next;

// printf("node::pathname=%s\n",node->pathname);

}

node->next = new\_node;

}

list->size++;

}

// return -1 when error occurs;

// return 1 for normal state

int save\_recent\_list(int n, struct Backup\_list \*list){

struct Node \*node = list->head;

for(int i=0; i<n-1; i++){

if(node->next == NULL){

return -1;

}

node = node->next;

}

node->next = NULL;

return 1;

}

// search before remove from backup\_list

void remove\_from\_list(char \*pathname, struct Backup\_list \*list){

struct Node \*node = list->head;

struct Node \*tmp;

int res;

res = search\_backup\_list(pathname, list);

if(res != -1){

if(res == 0){

list->head = list->head->next;

// free(list->head);

}

else{

tmp = node;

for(int i=0; i<res; i++){

if(i!=0)

tmp = node;

node = node->next;

}

tmp->next = node->next;

free(node);

node = NULL;

}

}

list->size--;

}

void remove\_all(struct Backup\_list \*list){

struct Node \*node = list->head;

struct Node \*prev;

int size = list->size;

if(node == NULL)

return;

if(node->next == NULL){

free(node);

node = NULL;

}

for(int i=0; i<size; i++){

node = list->head;

prev = list->head;

for(int j=0; j<size-i-1; j++){

if(j!=0)

prev = node;

// printf("node=%s, prev=%s, i=%d, j=%d\n", node->pathname, prev->pathname, i, j);

node = node->next;

}

free(node);

node = NULL;

prev->next = NULL;

}

list->head = NULL;

list->size = 0;

}

struct Node\* get(int n, struct Backup\_list \*list){

struct Node \*node = NULL;

if(n > list->size || list->head == NULL)

return NULL;

node = list->head;

for(int i=0; i<n; i++){

node = node->next;

}

return node;

}

void print\_backup\_list(struct Backup\_list \*list){

struct Node \*node = NULL;

node = list->head;

if(node == NULL) return;

printf("%10s %6s %3s %3s %3s %3s size=%d\n", "filename", "PERIOD", "-m", "-n", "-t", "-d", list->size);

for(int i=0; i < list->size; i++){

printf("%10s %6d %3d %3d %3d %3d\n", node->pathname, node->interval, node->options[0], node->options[1], node->options[2], node->options[3]);

if(node->next == NULL) return;

node = node->next;

}

printf("\n");

}

CC = gcc

SOURCE = ssu\_backup.c

HEADER = backup\_list.h

build : $(SOURCE) $(HEADER)

$(CC) -o ssu\_backup $(SOURCE) -lpthread