**컴퓨터학부 20182647 이형규**

**1. 개요**

**< ssu\_crond >**

**< 기능 >**

- 기본적으로 ssu\_crontab\_file에 작성된 명령어들을 주기에 따라 실행시켜주는 데몬 프로 그램이다.

- 로그파일(ssu\_crontab\_log)에 명령어가 실행된 시각과 run <실행주기> <명령어>를 작성 한다.

- 이 프로그램은 ssu\_crontab\_file의 변화에 반응한다.

**< ssu\_crontab >**

**< 프롬프트 기능 >**

**-** ssu\_crontab\_file에 존재하는 주기적으로 실행될 명령어들의 주기와 명령어를 숫자를 붙혀 보여줌

**- add <실행주기> <명령어>**

- ssu\_crontab\_file에 주기적으로 실행할 명령어 등록시킴

- 로그파일(ssu\_crontab\_log)에 추가되었다고 로그를 작성한다.

**- remove <COMMAND\_NUMBER>**

- 프롬프트로 출력한 ssu\_crontab\_file의 정보를 숫자를 입력받아 제거함

- 로그파일(ssu\_crontab\_log)에 명령어가 삭제되었다고 로그를 작성한다.

**- exit**

- 프로그램을 종료함

**< ssu\_rsync >**

**실행 : ./ssu\_rsync [OPTION : -t -m -r] <src> <dst>**

**< 기능 >**

**- 기본 기능**

- command-line을 통해 입력받은 <src>를 <dst> 디렉토리에 동기화를 시킨다.

- 기본적으로 src의 서브디렉토리는 동기화하지 않는다.

- 파일의 이름, 크기, 최종수정일이 같을 경우 동일한 파일이라고 감지한다.

- <src>에는 디렉토리 및 일반파일 가능, <dst>에는 디렉토리만 가능하다.

- 동기화 중 SIGINT 발생시 동기화 하지 않은 상태로 유지

- 로그파일(ssu\_rsync\_log)파일에 명령어와 동기화한 파일의 이름과 크기를 로그 에 남긴다.

**- r 옵션**

- <src>가 디렉토리일 경우 서브디렉토리도 동기화 시켜준다.

- 로그파일(ssu\_rsync\_log)파일에 명령어와 동기화한 파일의 이름과 크기를 로그 에 남긴다.

**- t 옵션**

- 동기화가 필요한 파일들을 tar파일로 묶어서 한번에 동기화 시킨다.

- 로그파일(ssu\_rsync\_log)에 tar파일의 크기와 동기화된 파일의 이름과 크기를 작성한다.

**- m 옵션**

- <src>에 존재하지 않는 파일이 <dst>에 존재하면 <dst>에서 삭제 시켜준다.

- 옵션을 설정하지 않은 경우 존재할 수 있음.

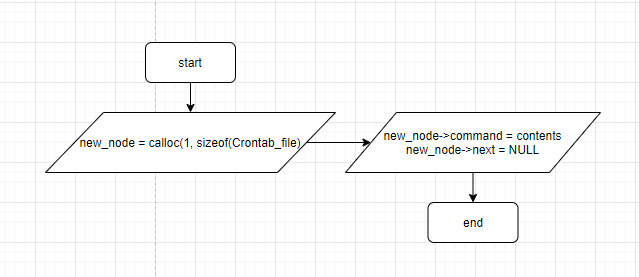
- 로그파일(ssu\_rsync\_log)에 삭제된 파일의 정보를 작성한다.

**2. 설계**

**<ssu\_crontab.c>**

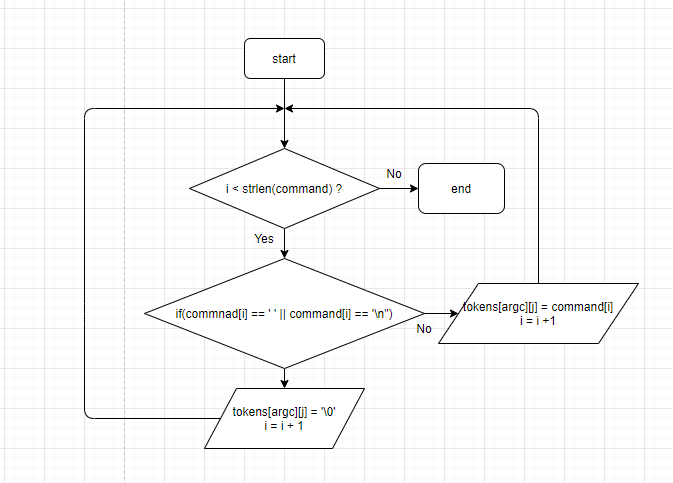
//ssu\_crontab\_file에서 한줄 읽어 노드를 생성하는 함수

Crontab\_file \*create\_node(char \*contents);

****

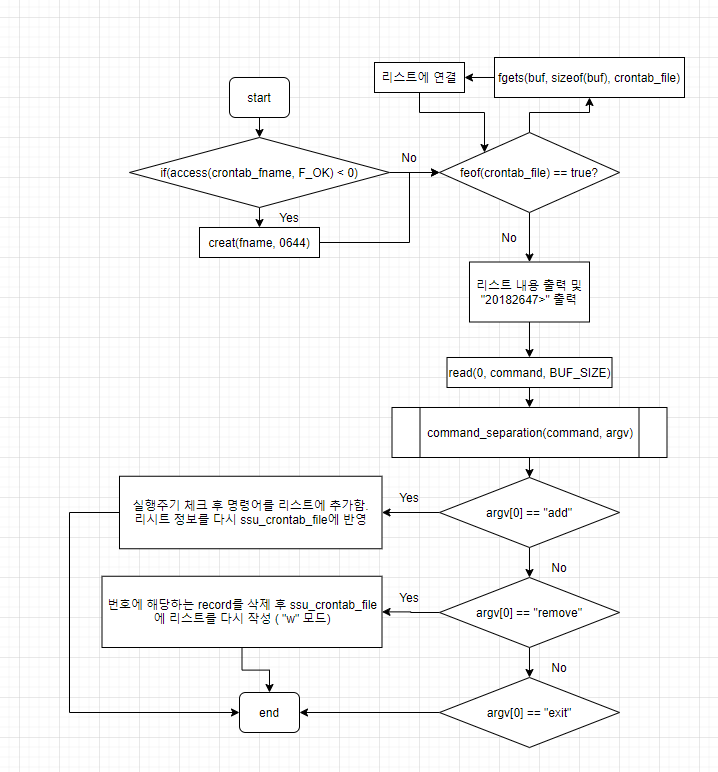
//프롬프트에서 입력받은 명령어들을 분리시켜주는 함수

int command\_separation(char \*command, char (\*argv)[BUF\_SIZE]);

****

//ssu\_crontab.c의 메인함수

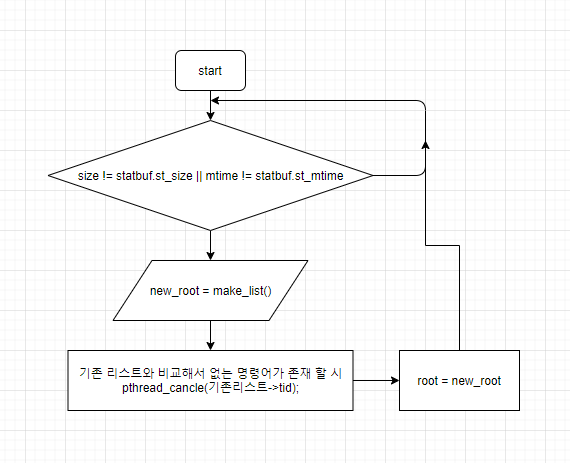
int main(void)

****

**<ssu\_crond.c>**

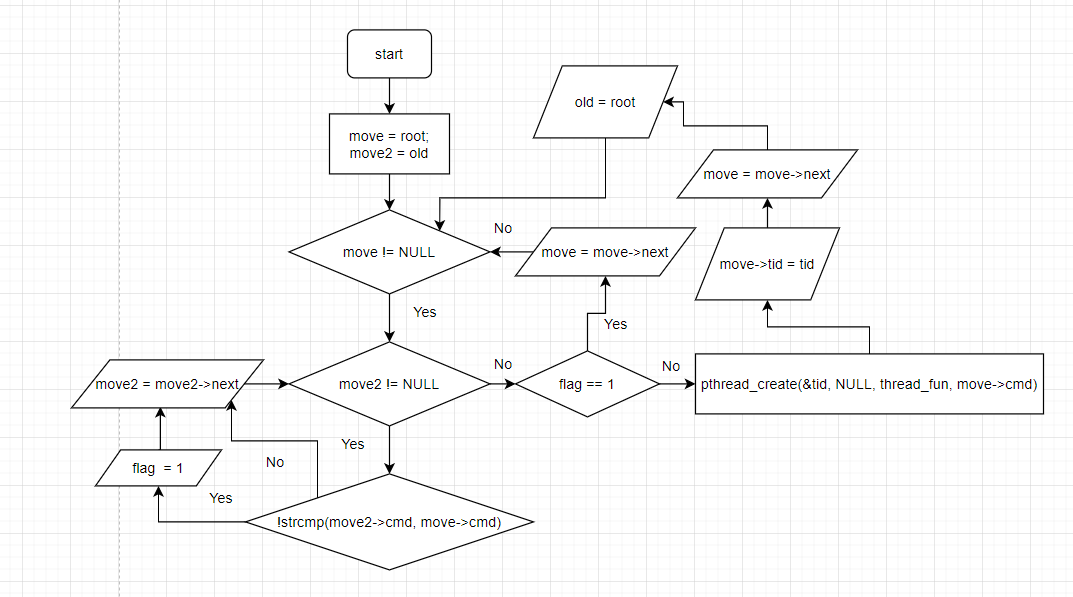
//파일 수정여부를 확인하고 업데이트

void \*check\_file\_modify(void \*arg);

****

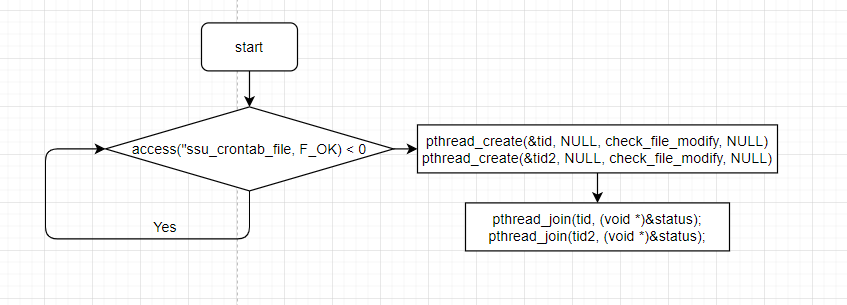
//명령어 별로 thread생성과 id 저장 후 list 교체

void \*check\_cmd(void \*arg);

****

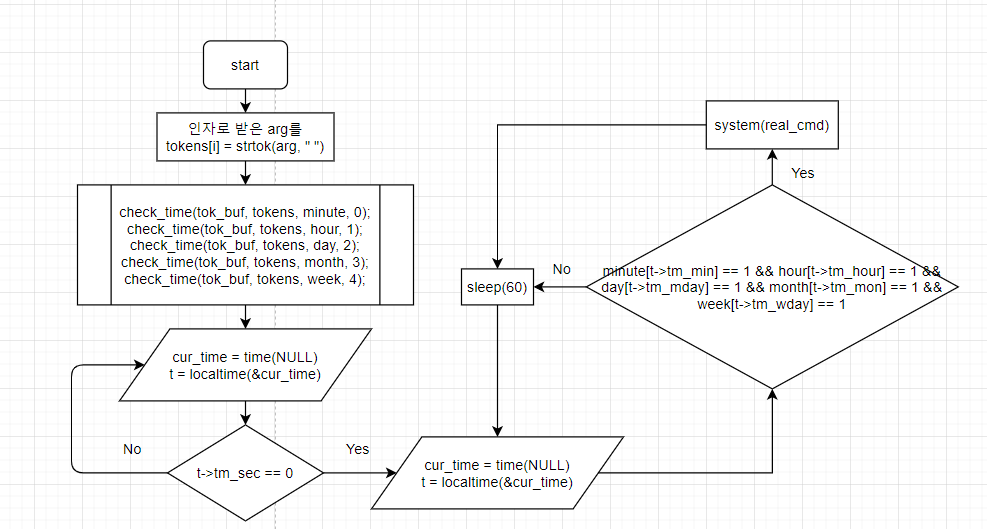
//하나는 ssu\_crontab\_file 수정 여부 체크용, 하나는 명령어별 thread생성

int main()

****

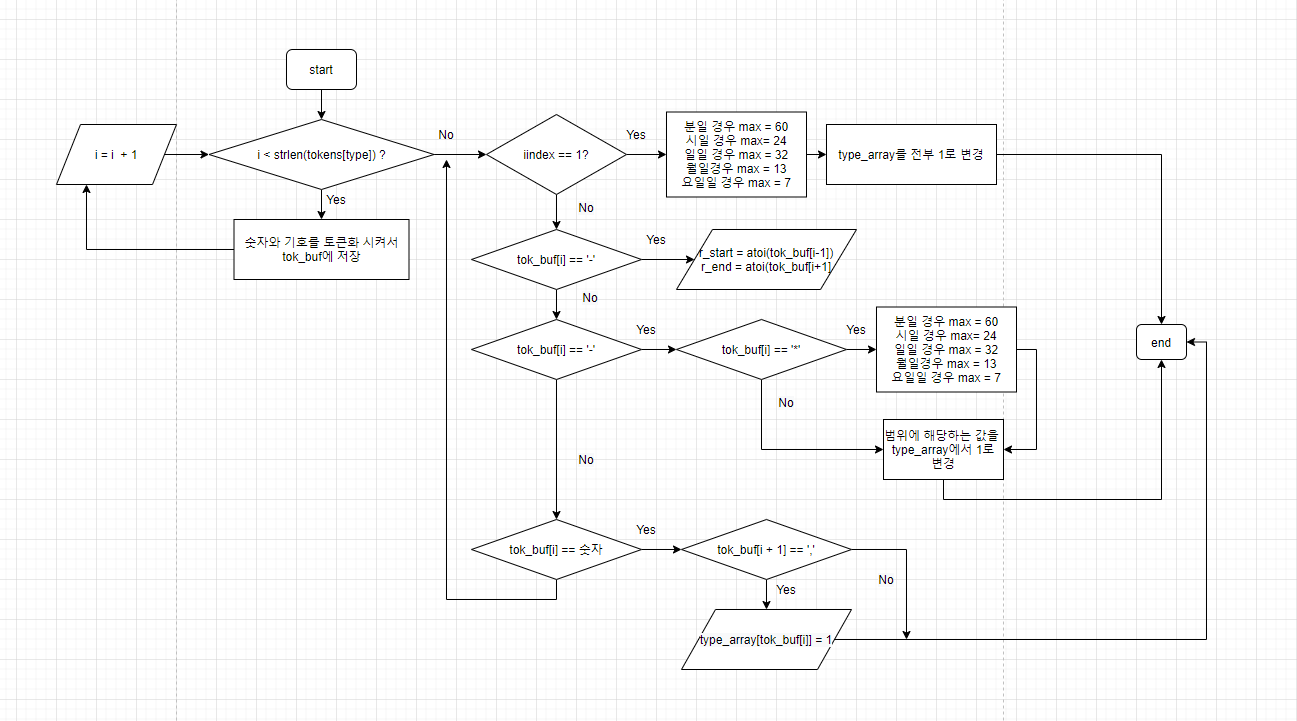
//각 스레드가 할 함수

void \*thread\_fun(void \*arg)

****

//실행 주기를 분석하여 type\_array에 플래그를 설정시켜줌

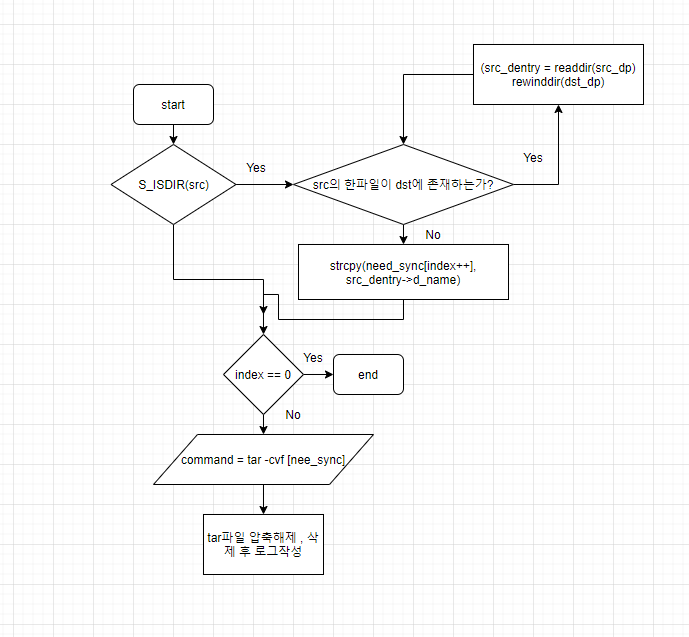
void check\_time(char (\*tok\_buf)[BUF\_SIZE], char (\*tokens)[BUF\_SIZE], int \*type\_array, int type)

****

**<ssu\_rsnyc.c>**

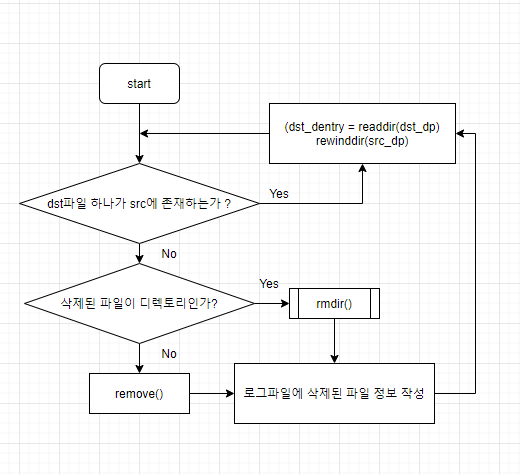
//t 옵션 수행 함수

void do\_tOption(char \*src, char \*dst)

****

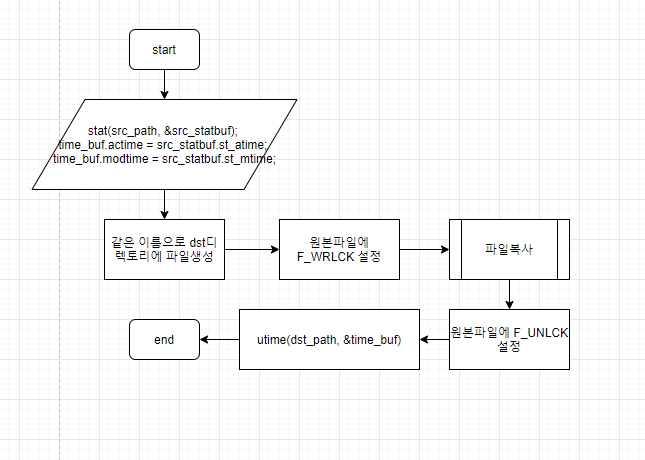
//-m 옵션 수행 함수

void do\_mOption(char \*src, char \*dst)

****

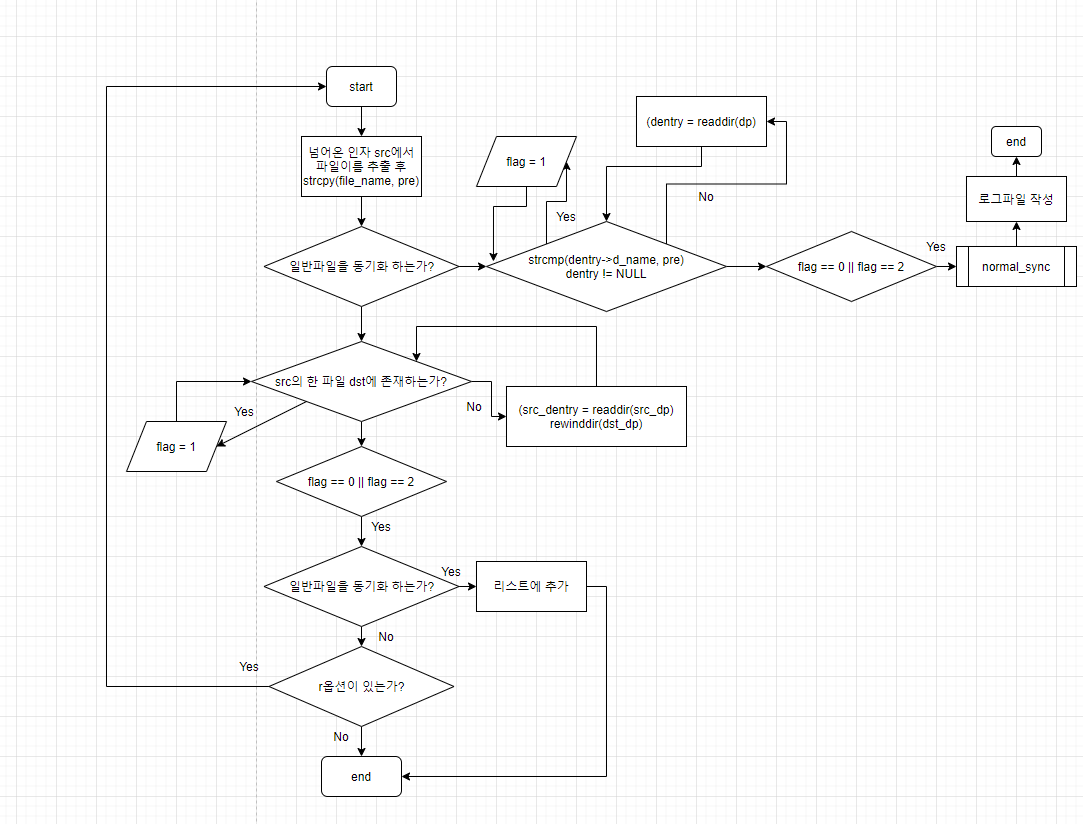
//실질적인 파일 복사가 이루어지는 동기화 함수

void normal\_sync(char \*src, char \*dst, char \*fname)

****

//동기화 시켜주는 함수

void Sync(char \*src, char\* dst, char \*sub\_dir, int argc, char \*\*argv, int depth)

****

**3. 구현**

**<ssu\_crontab.h>**

#define BUF\_SIZE 1024

//프롬프트에서 보여줄 ssu\_crontab\_file의 정보를 링크드 리스트 형태로

//표현하기 위한 구조체

typedef struct crontab\_file

{

char command[BUF\_SIZE];

struct crontab\_file \*next;

}Crontab\_file;

//ssu\_crontab\_file에서 한줄 읽어 노드를 생성하는 함수

Crontab\_file \*create\_node(char \*contents);

//프롬프트에서 입력받은 명령어들을 분리시켜주는 함수

int command\_separation(char \*command, char (\*argv)[BUF\_SIZE]);

//ssu\_crontab.c의 메인함수

//프롬프트 출력 및 add, remove 명령어 수행

//ssu\_crontab\_file에서 한줄 씩 읽어 리스트 형성

//add시 리스트에 추가하고 ssu\_crontab\_file에 반영

//remove시 리스트에서 삭제하고 처음부터 ssu\_crontab\_file에 작성

int main(void)

**<ssu\_crond.c>**

#define BUF\_SIZE 1024

//리스트를 구성할 구조체

typedef struct \_Command

{

char cmd[BUF\_SIZE];

time\_t last\_exec;

pthread\_t tid;

struct \_Command \*next;

}Command;

//실행주기를 토큰화 시켜서 각 요소 별로 배열을 생성해

//실행될 시각에 대해서 flag를 켜서 현재시간과 총 5개의 요소 flag가 1이면 명령어를 수행한다

void check\_time(char (\*tok\_buf)[BUF\_SIZE], char (\*tokens)[BUF\_SIZE], int \*type\_array, int type);

//명령어별로 각 thread가 수행할 명령어를 인자로서 넘겨받고

//주기적으로 명령어를 수행한다.

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

//ssu\_crontab\_file이 수정되는지 확인한다

//파일이 수정되었을 경우 삭제된 명령어를 수행하는 thread를 종료시키고 리스트 정보를 업데이트

void \*check\_file\_modify(void \*arg);

//리스트 정보를 이용해 각 명령어별로 thread를 생성해주고 id를 리스트에 저장한다

void \*check\_cmd(void \*arg);

//리스트에 필요한 노드를 생성하고 리턴시키는 함수

Command \*make\_node(char \*cmd);

//명령어에 대한 리스트와 그 명령어를 수행하는 thread\_id를 저장할 리스트를 제작한다

Command \*make\_list();

**<ssu\_rsync.h>**

//동기화가 필요한 파일에 대한 정보를 담고있음

typedef struct \_List

{

char fname[BUF\_SIZE];

long size;

struct \_List \*child;

struct \_List \*next;

}List;

//사용법을 출력하는 함수

//usage: ./ssu\_rsync [OPTION] <src> <dst>

void print\_usage();

//동기화를 수행하기 위한 작업을 진행함

//일반파일을 동기화하는 경우 normal\_sync호출 하고 끝

//디렉토리의 경우 동기화가 필요한 파일을 리스트화 시키고 r옵션이 있으면 서브디렉토리도 동기화함

void Sync(char \*src, char \*dst, char \*sub\_dir, int argc, char \*\*argv, int depth);

//t옵션을 수행함

//tar로 묶어서 동기화를 진행 일반파일, 디렉토리 모두 가능

void do\_tOption(char \*src, char \*dst);

//m옵션을 수행함

//src에 존재하지 않는파일을 dst에서도 삭제함

void do\_mOption(char \*src, char \*dst);

//실질적으로 동기화를 진행함

//파일을 열고 복사하는 과정

void normal\_sync(char \*src, char \*dst, char \*fname);

//동기화 정보를 리스트화 시키는 함수들

List \*make\_node(char \*fname);

List \*make\_list(char \*path);

List \*print\_list(List \*root);

//디렉토리 삭제함수

void rmdirs(char \*path);

//최종적으로 로그에 시각과 파일정보를 작성해주는 함수

void print\_log(char \*src, char \*dst);

//SIGINT 발생시 동기화 이전으로 돌리는 기능을 가지고 있음

void signal\_handler();

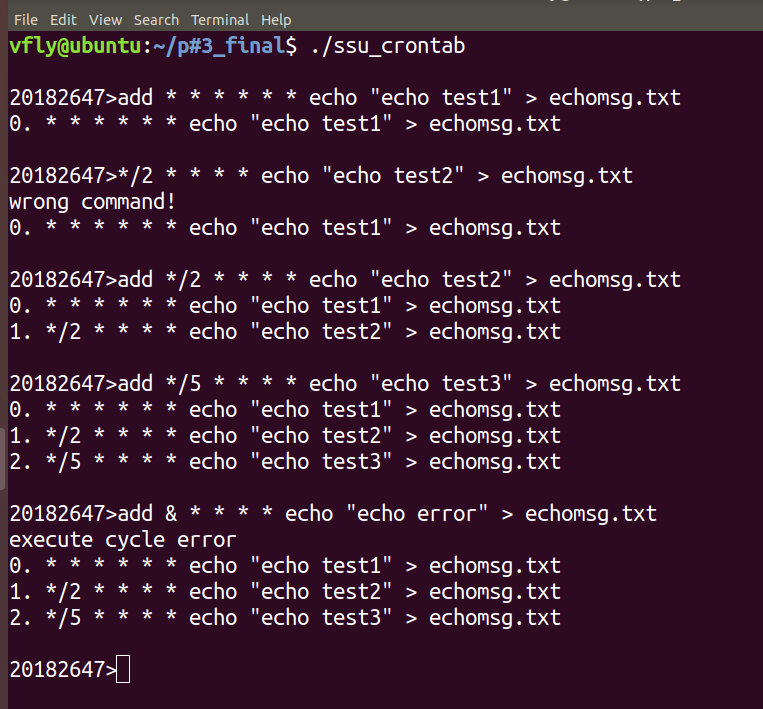
**4. 테스트 및 결과**

**<ssu\_crontab>**

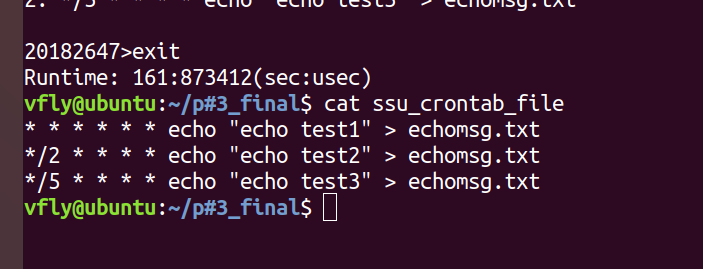
**-** add 명령어를 이용한 주기적으로 실행할 명령어 추가 테스트

- 실행주기에 \* , / - 이외의 문자가 들어갈 경우 execute cycle error출력

- 정상적으로 add가 성공 시 프롬프트에 출력이됨

****

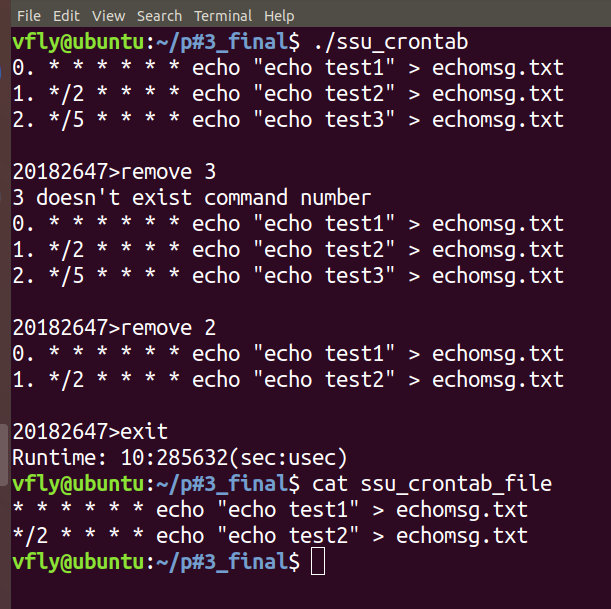
**-** exit 명령어 입력 후 ssu\_crontab\_file에 정상적으로 반영됨

****

**-** remove 명령어 실행 테스트

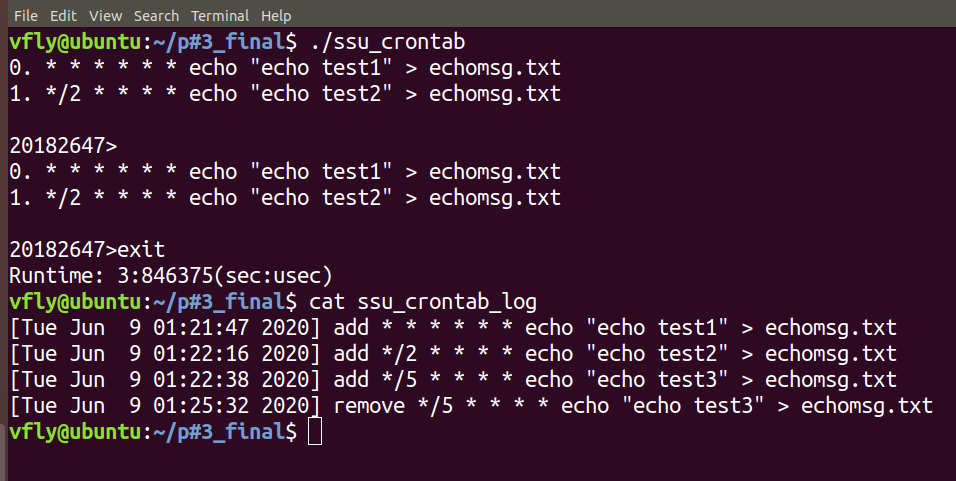
- command-line에 프롬프트에 출력된 최대숫자보다 큰 숫자 입력 시 에러처리

- 이후 2번 명령어 삭제 시 정상적으로 프롬프트와 ssu\_crontab\_file에 반영함

****

**-** 프롬프트에서 엔터만 입력 시 프롬프트 다시 출력

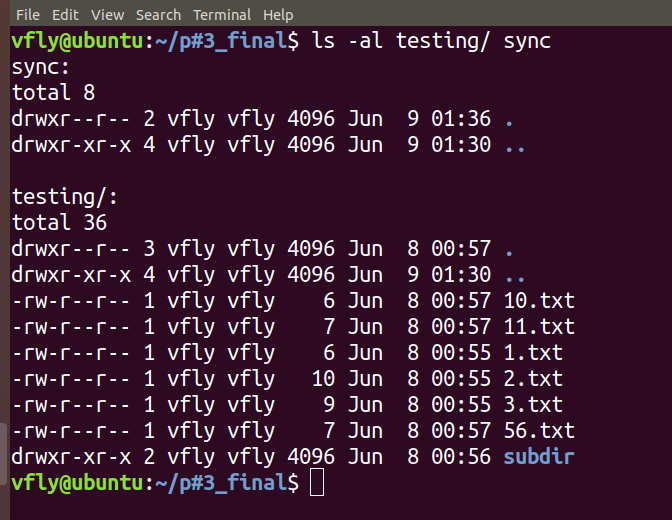
- 위에서 수행했던 add명령어와 remove명령어는 ssu\_crontab\_log에 정상적으로 반영

****

**<ssu\_rsync>**

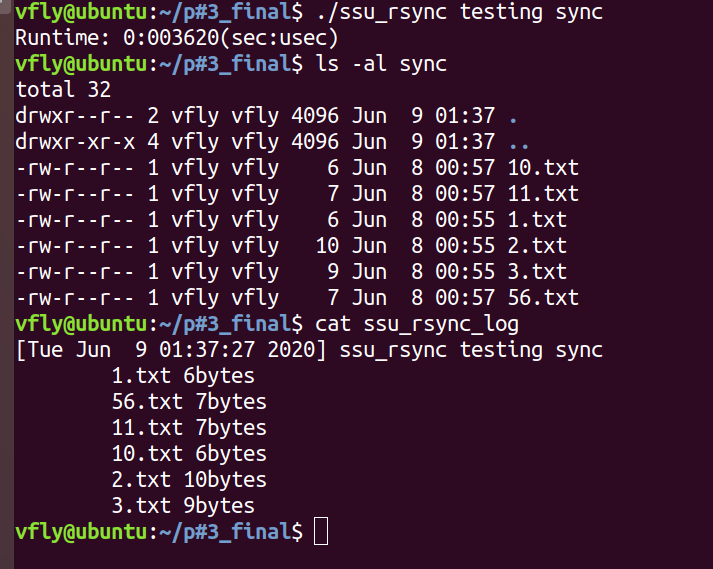
**-** 기본 동기화 테스트

- testing :<src> , sync : <dst>



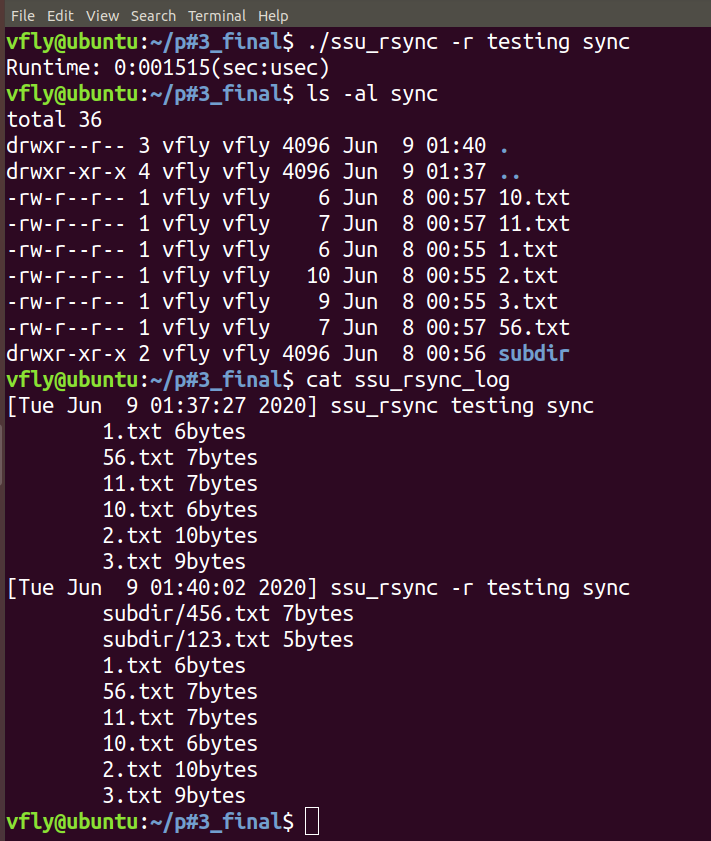
- 동기화 진행

- log파일에 입력한 명령어와 동기화한 파일의 이름과 크기를 작성함 ( 서브디렉토리는 동기화x )

  
- -r 옵션 진행 시

- testing은 동일, sync는 비우고 진행하였습니다.

- r옵션 적용시 일반파일 뿐만 아니라 서브 디렉토리내에 존재하는 파일들도 동기화를 진행하고 로그를 작성함

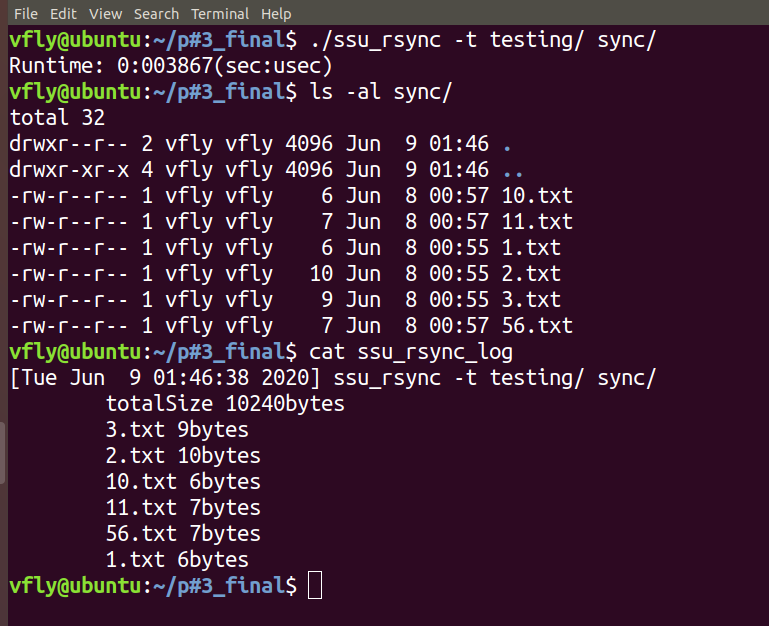


- -t옵션 적용시

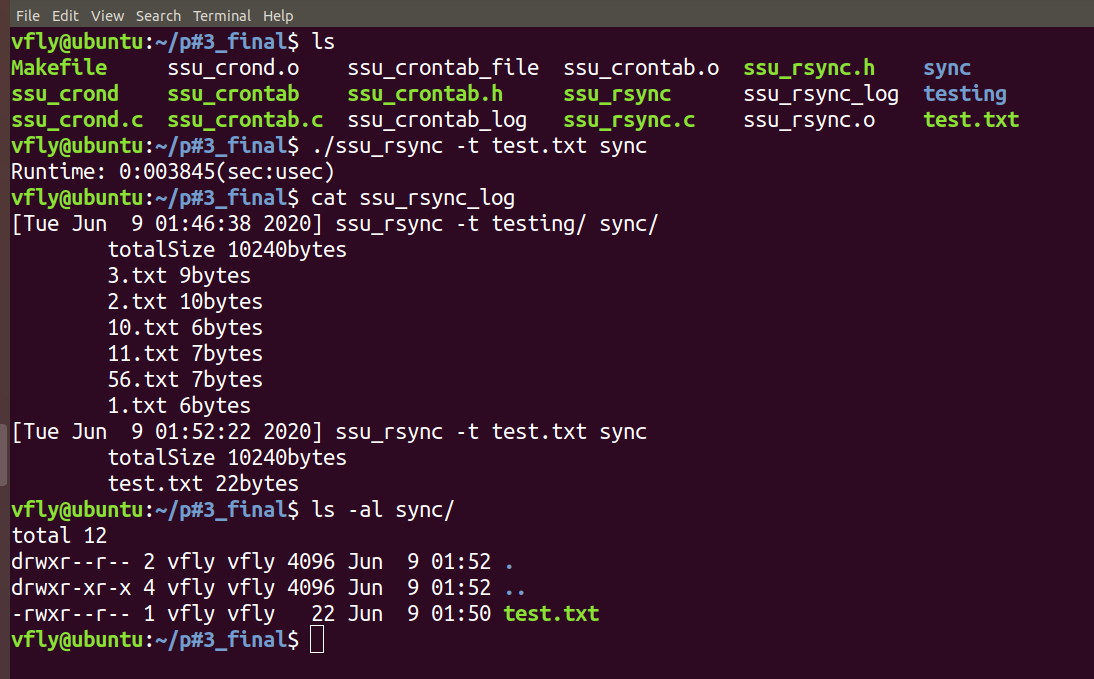
- testing은 동일, sync는 비우고 진행하였습니다. + 로그파일 초기화 진행

- tar파일을 이용하여 동기화를 진행 ( 서브디렉토리는 동기화하지 않음 )

- 로그파일에 tar파일의 사이즈와 동기화한 파일들의 정보 작성



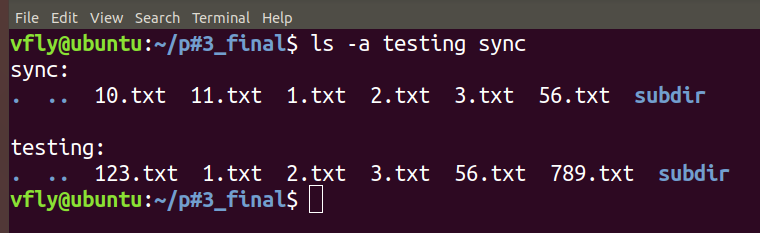
- 일반파일을 -t 옵션 적용시, sync는 비우고 진행하였습니다.



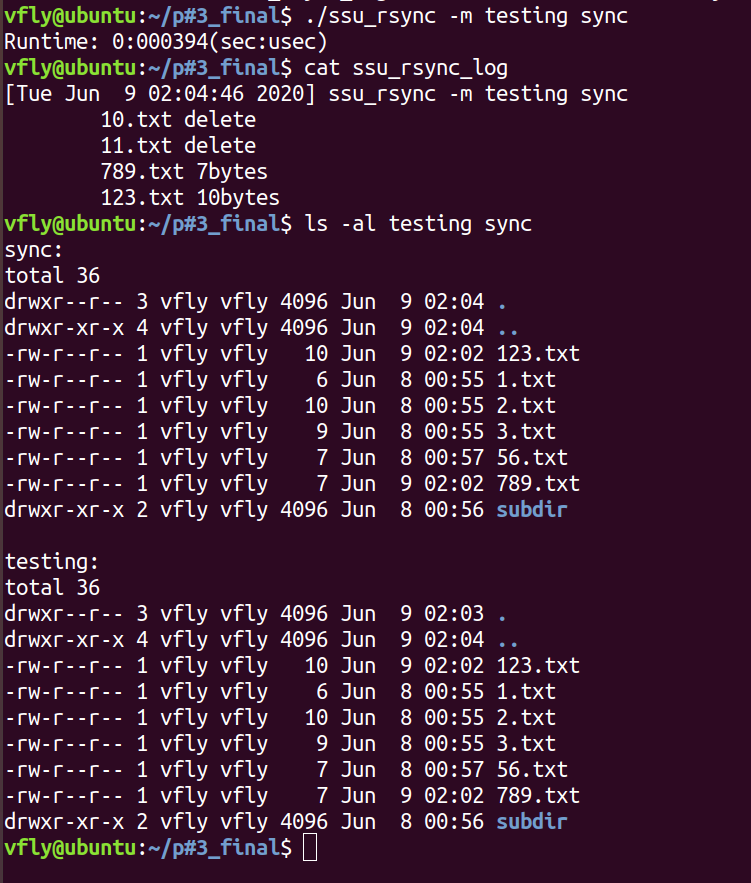
- -m 옵션 적용시

- 로그파일은 초기화 했습니다.

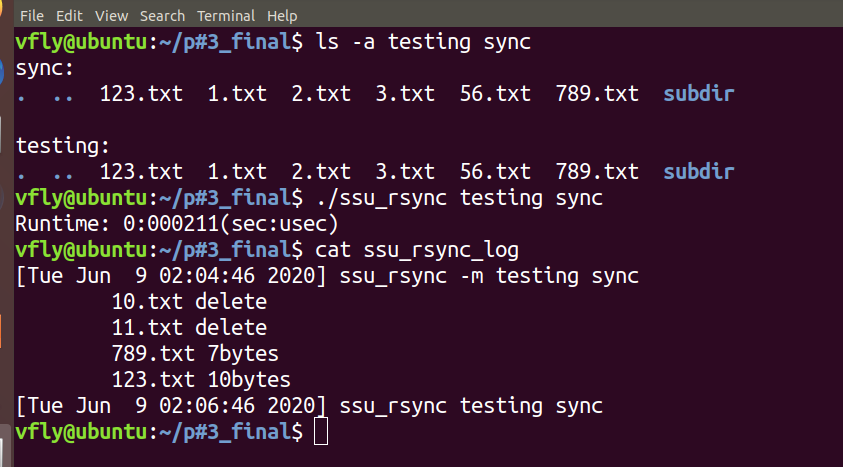
- 초기 testing과 sync 상태



- 동기화 진행후 10.txt. 11.txt 삭제 후 123.txt 789.txt 동기화

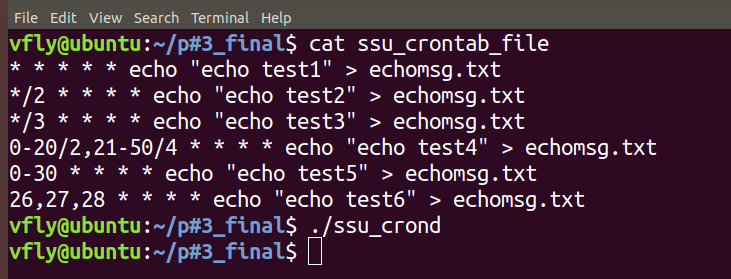


- 동기화할 파일이 없을 경우 로그파일에 명령어만 작성함



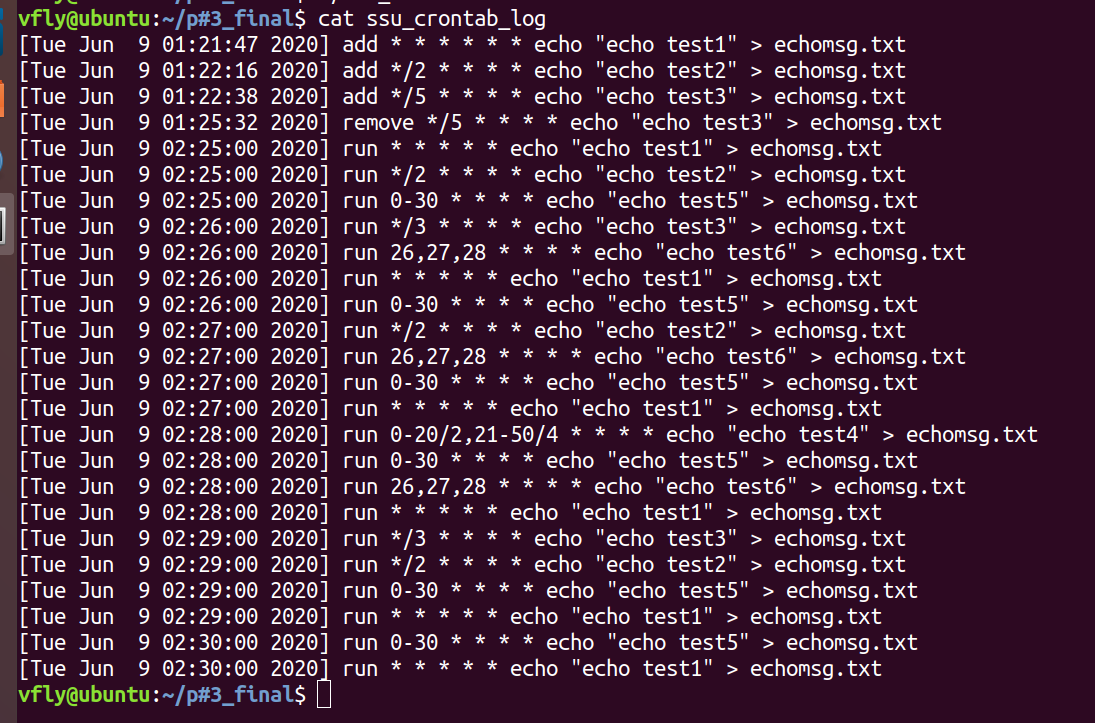
**<ssu\_crond>**

- ssu\_crontab\_file의 상태는 아래와 같이 테스트를 진행하였습니다(분을 제외한 나머지는 테스트하기 너무 오래걸리는 관계로 분만 진행했습니다)

  
- 실행 시작 시각 : 02:25분 화요일

- 로그 확인 시각 : 02:30분 화요일

- 각 주기에 맞게 실행되서 로그에 출력된 화면



**5. 소스코드 및 주석**

**<ssu\_crontab.h>**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

#include <string.h>

#include <sys/stat.h>

#include <signal.h>

#define BUF\_SIZE 1024

//프롬프트에서 보여줄 ssu\_crontab\_file의 정보를 링크드 리스트 형태로

//표현하기 위한 구조체

typedef struct crontab\_file

{

char command[BUF\_SIZE];

struct crontab\_file \*next;

}Crontab\_file;

//ssu\_crontab\_file에서 한줄 읽어 노드를 생성하는 함수

Crontab\_file \*create\_node(char \*contents);

//프롬프트에서 입력받은 명령어들을 분리시켜주는 함수

int command\_separation(char \*command, char (\*argv)[BUF\_SIZE]);

//수행시간 측정함수

void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t);

**<ssu\_crontab.c>**

#include <stdio.h>

#include <sys/time.h>

#include <time.h>

#include <ctype.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include "ssu\_crontab.h"

#define SECOND\_TO\_MICRO 1000000

//ssu\_crontab.c의 메인함수

//프롬프트 출력 및 add, remove 명령어 수행

//ssu\_crontab\_file에서 한줄 씩 읽어 리스트 형성

//add시 리스트에 추가하고 ssu\_crontab\_file에 반영

//remove시 리스트에서 삭제하고 처음부터 ssu\_crontab\_file에 작성

int main(void)

{

struct timeval begin\_t, end\_t;

FILE \*crontab\_file;

FILE \*crond\_log;

int ssu\_crontab\_file\_command\_cnt = 0;

char \*crontab\_fname = "ssu\_crontab\_file";

char \*crond\_log\_fname = "ssu\_crontab\_log";

char buf[BUF\_SIZE];

memset(buf, 0, BUF\_SIZE);

Crontab\_file \*head = NULL;

Crontab\_file \*tail = NULL;

gettimeofday(&begin\_t, NULL);

//ssu\_crontab\_file이 존재하지 않을 경우 생성함

if(access(crontab\_fname, F\_OK) < 0)

{

int fd;

creat(crontab\_fname, 0644);

}

//get ssu\_crontab\_file info

if((crontab\_file = fopen(crontab\_fname, "r+")) == NULL)

{

fprintf(stderr, "open error for %s\n", crontab\_fname);

exit(1);

}

//ssu\_crontab\_file에서 한줄 씩 읽어서 리스트화 시킴

while(!feof(crontab\_file))

{

memset(buf, 0, BUF\_SIZE);

fgets(buf, sizeof(buf), crontab\_file);

//읽은 길이가 0일 경우 파일의 끝임

if(strlen(buf) == 0) break;

//ssu\_crontab\_file에 존재하는 명령어의 갯수를 나타내는 변수

//명령어의 갯수를 증가시켜줌

ssu\_crontab\_file\_command\_cnt++;

//새로운 노드를 생성하고

Crontab\_file \*new\_node = create\_node(buf);

//항상 리스트 맨뒤에 연결 시켜준다

//순서를 맞추기 위함

if(head == NULL)

{

head = new\_node;

tail = new\_node;

}

else if(tail != NULL)

{

tail->next = new\_node;

tail = new\_node;

}

}

fclose(crontab\_file);

while(1)

{

char command[BUF\_SIZE];

char argv[200][BUF\_SIZE];

for(int i=0; i<200; i++) memset(argv[i], 0, BUF\_SIZE);

memset(command, 0, BUF\_SIZE);

int count = 0;

//프롬프트 출력

//리스트 정보와 학번을 출력함

Crontab\_file \*move = head;

while(move != NULL)

{

printf("%d. %s", count++, move->command);

move = move->next;

}

//prompt

write(1, "\n20182647>", strlen("\n20182647>"));

//stdin으로 부터 명령어를 입력받음

read(0, command, BUF\_SIZE);

//입력받은 명령어를 공백을 기준으로 분리하여 argv에 저장함

int argc = command\_separation(command, argv);

//단순 엔터만 입력했을 경우 프롬프트 재출력

if(command[0] == '\n')

continue;

//명령어 시작이 add일경우

if(!strcmp(argv[0], "add"))

{

//check cycles

//실행 주기 분석

int flag = 0;

//총 5개를 분석함

for(int i=1; i<6; i++)

{

//각자의 길이 만큼 검사를 진행

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

{

if(j == 0)

{

//맨처음이 숫자나 \* 이 아닐 경우 무조건 에러처리

if(argv[i][j] != '\*' && !isdigit(argv[i][j]))

{

flag = 1;

break;

}

}

else

{

// / , - 숫자 이외의 다른 기호가 들어올 경우 flag를 세워 에러처리

if(argv[i][j] != '/' && argv[i][j] != ',' && argv[i][j] != '-'

&& !isdigit(argv[i][j]))

{

flag = 1;

break;

}

}

}

//플래그 확인

if(flag == 1)

break;

}

//플래기가 존재할 경우 에러처리

if(flag == 1)

{

fprintf(stderr, "execute cycle error\n");

continue;

}

char exec\_cmd[BUF\_SIZE];

char exec\_cycle[BUF\_SIZE];

memset(exec\_cycle, 0, BUF\_SIZE);

memset(exec\_cmd, 0 ,BUF\_SIZE);

//실행주기를 하나로 합쳐줌

for(int i=1; i<6; i++)

{

strcat(exec\_cycle, argv[i]);

strncat(exec\_cycle, " ", 1);

}

//공백으로 분리되었던 실행명령어를 다시 하나로 합쳐줌

for(int i=6; i<argc; i++)

{

strcat(exec\_cmd, argv[i]);

strncat(exec\_cmd , " ", 1);

}

//crontab\_file 작성에 필요한 문자열 제작

memset(buf, 0, BUF\_SIZE);

sprintf(buf, "%s%s\n", exec\_cycle, exec\_cmd);

//ssu\_crontab\_file 오픈

if((crontab\_file = fopen(crontab\_fname, "a+")) == NULL)

{

fprintf(stderr, "open error for %s\n", crontab\_fname);

continue;

}

//위에서 제작한 <실행주기> <실행할 명령어> 를 파일에 작성하고

fwrite(buf, strlen(buf), 1, crontab\_file);

//그 string을 리스트로 관리하기 위해 노드를 생성후 리스트에 연결시켜줌

Crontab\_file \*new\_node = create\_node(buf);

//리스트 맨끝에 연결

if(head == NULL)

{

head = new\_node;

tail = new\_node;

}

else if(tail != NULL)

{

tail->next = new\_node;

tail = new\_node;

}

fclose(crontab\_file);

//로그작성을 위한 로그파일 오픈

if((crond\_log = fopen(crond\_log\_fname, "a+")) == NULL)

{

fprintf(stderr, "open error for %s\n", crond\_log\_fname);

continue;

}

//현재시각 추출

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

struct tm \*t = localtime(&cur\_time);

char added\_cmd[BUF\_SIZE];

char buf2[BUF\_SIZE];

memset(buf2, 0, BUF\_SIZE);

memset(added\_cmd, 0, BUF\_SIZE);

strcpy(added\_cmd, buf);

memset(buf, 0, BUF\_SIZE);

//현재시각을 문자열로 변경하고 맨뒤에 개행문자 삭제

strcpy(buf2, ctime(&cur\_time));

buf2[strlen(buf2) - 1] = 0;

//형식에 맞게 문자열 제작 [현재시각] add [추가한 명령어]

sprintf(buf, "[%s] %s %s", buf2, "add", added\_cmd);

//로그파일에 작성함

fwrite(buf, strlen(buf), 1, crond\_log);

fclose(crond\_log);

//ssu\_crontab\_file에 존재하는 명령어 갯수를 나타내는 변수 1 증가

//-> 나중에 삭제할때 사용함

ssu\_crontab\_file\_command\_cnt++;

}

//명령어 시작이 remove일 경우

else if(!strcmp(argv[0], "remove"))

{

//명령어 인자의 갯수를 체크하여 에러처리

if(argc > 2)

{

fprintf(stderr, "too many arguments\n");

continue;

}

if(argc == 1)

{

fprintf(stderr, "input command\_number\n");

continue;

}

//만약 2번째인자로 숫자가 아닌 다른정보가 있다면 에러처리함

int flag = 0;

for(int i=0; i<strlen(argv[1]); i++)

{

//숫자 인지 판별

if(!isdigit(argv[1][i]))

{

flag = 1;

break;

}

}

//에러처리

if(flag == 1)

{

fprintf(stderr, "command number error\n");

continue;

}

//에러가 없을 경우

else

{

//두번째 인자를 정수로 변환

int command\_number = atoi(argv[1]);

//만약 입력한 숫자가 리스트에 존재하는 정보의 갯수보다 많을 경우 에러

if(command\_number >= ssu\_crontab\_file\_command\_cnt)

{

fprintf(stderr, "%d doesn't exist command number\n", command\_number);

continue;

}

//범위내의 숫자를 입력했을 경우

else

{

Crontab\_file \*move = head;

Crontab\_file \*pre = NULL;

//로그 작성에 필요한 삭제된 명령어 정보를 저장하는 배열

char will\_deleted\_command[BUF\_SIZE];

memset(will\_deleted\_command, 0, BUF\_SIZE);

//입력한 숫자만큼 리스트에서 이동

for(int i=0; i<command\_number; i++)

{

pre = move;

move = move->next;

}

//지울 데이터에 대한 정보를 저장

strcpy(will\_deleted\_command, move->command);

//start

if(move == head)

{

head = move->next;

free(move);

}

//end

else if(move == tail)

{

pre->next = NULL;

tail = pre;

free(move);

}

//middle

else

{

pre->next = move->next;

free(move);

}

//ssu\_crontab\_file을 쓰기 모드로 open

if((crontab\_file = fopen(crontab\_fname, "w")) == NULL)

{

fprintf(stderr, "open error for %s\n", crontab\_fname);

continue;

}

//리스트 정보를 ssu\_crontab\_file에 갱신시켜줌

move = head;

while(move != NULL)

{

fwrite(move->command, strlen(move->command), 1, crontab\_file);

move = move->next;

}

fclose(crontab\_file);

//log파일을 append 모드로 open 하여 작성 준비를 함

if((crond\_log = fopen(crond\_log\_fname, "a+")) == NULL)

{

fprintf(stderr, "open error for %s\n", crond\_log\_fname);

continue;

}

//로그파일에 현재시각과 사라진 명령어 정보를 작성

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

struct tm \*t = localtime(&cur\_time);

char buf2[BUF\_SIZE];

memset(buf2, 0, BUF\_SIZE);

memset(buf, 0, BUF\_SIZE);

//현재시각을 문자열화 시킴

strcpy(buf2, ctime(&cur\_time));

buf2[strlen(buf2) - 1] = 0;

sprintf(buf, "[%s] %s %s", buf2, "remove", will\_deleted\_command);

fwrite(buf, strlen(buf), 1, crond\_log);

fclose(crond\_log);

}

}

}

//exit명령어를 입력받았을 시

else if(!strcmp(argv[0], "exit"))

{

gettimeofday(&end\_t, NULL);

ssu\_runtime(&begin\_t, &end\_t);

exit(0);

}

//이외의 모든 명령어는 에러처리

else

{

fprintf(stderr, "wrong command!\n");

continue;

}

}

}

//프롬프트에서 입력받은 문자열을 공백단위로 분리시켜서 tokens배열에 하나씩 저장

int command\_separation(char \*command, char (\*tokens)[BUF\_SIZE])

{

int argc = 0;

for(int i=0; i< strlen(command); i++)

{

for(int j=0; ; j++,i++)

{

//공백이나 엔터를 만날 경우 count를 하나 증가시키고 배열에 널문자를 저장

if(command[i] == ' ' || command[i] == '\n')

{

argc++;

tokens[argc][j] = '\0';

break;

}

//엔터를 입력받았을 경우 중지

if(command[i] == '\n')

break;

//일반문자들일 경우 하나씩 저장

tokens[argc][j] = command[i];

}

}

return argc;

}

//ssu\_crontab\_file의 리스트를 조직할 노드 생성함수

Crontab\_file \*create\_node(char \*contents)

{

Crontab\_file \*new\_node = (Crontab\_file \*)calloc(1, sizeof(Crontab\_file));

strcpy(new\_node->command, contents);

new\_node->next = NULL;

return new\_node;

}

//수행시간 측정함수

void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t)

{

//끝난 시각에서 시작한 시각을 뺌

end\_t->tv\_sec -= begin\_t->tv\_sec;

//만약 끝난시각의 밀리초가 더 작으면

//끝난시각의 초를 하나 줄이고 마이크로초를 증가시킴

if(end\_t->tv\_usec < begin\_t->tv\_usec)

{

end\_t->tv\_sec--;

end\_t->tv\_usec += SECOND\_TO\_MICRO;

}

end\_t->tv\_usec -= begin\_t->tv\_usec;

printf("Runtime: %ld:%06ld(sec:usec)\n", end\_t->tv\_sec, end\_t->tv\_usec);

}

**<ssu\_rsync.h>**

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <dirent.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <utime.h>

#include <ctype.h>

#include <time.h>

#define BUF\_SIZE 1024

//동기화가 필요한 파일에 대한 정보를 담고있음

typedef struct \_List

{

char fname[BUF\_SIZE];

long size;

struct \_List \*child;

struct \_List \*next;

}List;

//사용법을 출력하는 함수

//usage: ./ssu\_rsync [OPTION] <src> <dst>

void print\_usage();

//동기화를 수행하기 위한 작업을 진행함

//일반파일을 동기화하는 경우 normal\_sync호출 하고 끝

//디렉토리의 경우 동기화가 필요한 파일을 리스트화 시키고 r옵션이 있으면 서브디렉토리도 동기화함

void Sync(char \*src, char \*dst, char \*sub\_dir, int argc, char \*\*argv, int depth);

//t옵션을 수행함

//tar로 묶어서 동기화를 진행 일반파일, 디렉토리 모두 가능

void do\_tOption(char \*src, char \*dst);

//m옵션을 수행함

//src에 존재하지 않는파일을 dst에서도 삭제함

void do\_mOption(char \*src, char \*dst);

//실질적으로 동기화를 진행함

//파일을 열고 복사하는 과정

void normal\_sync(char \*src, char \*dst, char \*fname);

//동기화 정보를 리스트화 시키는 함수들

List \*make\_node(char \*fname);

List \*make\_list(char \*path);

List \*print\_list(List \*root);

//디렉토리 삭제함수

void rmdirs(char \*path);

//최종적으로 로그에 시각과 파일정보를 작성해주는 함수

void print\_log(char \*src, char \*dst);

//SIGINT 발생시 동기화 이전으로 돌리는 기능을 가지고 있음

void signal\_handler();

//수행시간측정함수

void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t);

**<ssu\_rsync.c>**

**#include <stdio.h>**

**#include <ctype.h>**

**#include <unistd.h>**

**#include <stdlib.h>**

**#include <dirent.h>**

**#include <fcntl.h>**

**#include <string.h>**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include "ssu\_rsync.h"**

**#include <sys/time.h>**

**#define SECOND\_TO\_MICRO 1000000**

**#define TRUE 1**

**#define FALSE 0**

**//옵션 체크용 변수**

**int tOption = FALSE;**

**int rOption = FALSE;**

**int mOption = FALSE;**

**int nonOption = FALSE;**

**//-t옵션 적용시 tar파일 생성을 위한 문자열**

**char \*tar\_command = "tar -cf";**

**char \*tar\_fname = "temp.tar";**

**char cur\_file\_name[BUF\_SIZE];**

**//로그파일 작성시 로그 작성시각과 명령어 문장의 출력을 판단하기 위한 변수**

**int first\_flag = 0;**

**//SIGINT 발생시 이전 dst로 되돌리기 위한 리스트**

**List \*old\_dst = NULL;**

**//동기화할 파일들의 리스트**

**List \*sync\_list = NULL;**

**//임시변수**

**char glob\_dst[BUF\_SIZE];**

**char glob\_src[BUF\_SIZE];**

**struct timeval begin\_t, end\_t;**

**int main(int argc, char \*argv[])**

**{**

**//SIGINT 발생시 시그널 핸들러 등록**

**signal(SIGINT, signal\_handler);**

**int log\_file;**

**char \*log\_fname = "ssu\_rsync\_log";**

**char option[BUF\_SIZE];**

**char src[BUF\_SIZE];**

**char dst[BUF\_SIZE];**

**memset(src, 0, BUF\_SIZE);**

**memset(dst, 0, BUF\_SIZE);**

**memset(option, 0, BUF\_SIZE);**

**gettimeofday(&begin\_t, NULL);**

**//command-line으로 들어온 인자의 갯수 체크**

**if(argc > 4 || argc < 3)**

**{**

**fprintf(stderr, "usage: ./ssu\_rsync [option] <src> <dst>\n");**

**gettimeofday(&end\_t, NULL);**

**ssu\_runtime(&begin\_t, &end\_t);**

**exit(1);**

**}**

**//각 옵션별로 flag 설정**

**//option check**

**if(!strcmp(argv[1], "-t")) tOption = TRUE;**

**else if(!strcmp(argv[1], "-r")) rOption = TRUE;**

**else if(!strcmp(argv[1], "-m")) mOption = TRUE;**

**else nonOption = TRUE;**

**//옵션이 없을 경우 두번째 세번째 인자가 src와 dst**

**if(nonOption == TRUE)**

**{**

**strcpy(src, argv[1]);**

**strcpy(dst, argv[2]);**

**}**

**//옵션이 무엇이든 존재하면 세번째 네번째 인자가 src와 dst**

**else**

**{**

**strcpy(option, argv[1]);**

**strcpy(src, argv[2]);**

**strcpy(dst, argv[3]);**

**}**

**//signal handler에서 src와dst 정보를 얻기 위해 데이터 복사**

**strcpy(glob\_dst, dst);**

**strcpy(glob\_src, src);**

**//src가 존재하는지, src에 접근권한 rwx가 모두 존재하는지 확인**

**if(access(src, F\_OK) < 0)**

**print\_usage();**

**if(access(src, R\_OK) < 0 || access(src, W\_OK) < 0 || access(src, X\_OK) < 0)**

**print\_usage();**

**//dst가 존재하는지 확인**

**if(access(dst, F\_OK) < 0)**

**print\_usage();**

**struct stat statbuf;**

**if(stat(dst, &statbuf) < 0)**

**{**

**fprintf(stderr, "stat error\n");**

**exit(1);**

**}**

**//dst가 디렉토리가 아니거나, rwx접근권한이 없을 경우 사용법출력**

**if(S\_ISDIR(statbuf.st\_mode) == 0)**

**print\_usage();**

**if(access(dst, R\_OK) < 0 || access(dst, W\_OK) < 0 || access(dst, X\_OK) < 0)**

**print\_usage();**

**//로그파일이 없으면 새로생성됨**

**FILE \*log\_file\_fp;**

**if((log\_file\_fp = fopen(log\_fname, "a+")) == NULL)**

**{**

**fprintf(stderr ,"fopen error for %s\n", log\_fname);**

**exit(1);**

**}**

**fclose(log\_file\_fp);**

**//SIGINT 발생시 되돌리기 위해 동기화전 데이터 저장**

**old\_dst = make\_list(dst);**

**//m옵션이 있을 경우 src에 없는 파일은 dst에서도 삭제를 수행 하고 동기화진행**

**if(mOption == TRUE) do\_mOption(src,dst);**

**//t옵션이 있을 경우 따로 tar를 만들어서 동기화를 시켜줌**

**if(tOption == TRUE)**

**{**

**do\_tOption(src,dst);**

**gettimeofday(&end\_t, NULL);**

**ssu\_runtime(&begin\_t, &end\_t);**

**exit(1);**

**}**

**//그냥 동기화거나 r옵션, m옵션 수행 후 동기화 진행**

**//아무작업을 진행하지 않더라도 로그파일에 command-line에서 넘어온 명령어를 출력하라는**

**//게시판글을 보고 추가한 구문이라 이 뒤에 존재하는 if(first\_flag == 0) {} 은 실행되지 않음**

**Sync(src, dst, NULL, argc, argv, 0);**

**if(first\_flag == 0)**

**{**

**first\_flag = 1;**

**//로그파일 오픈**

**if((log\_file\_fp = fopen(log\_fname, "a+")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", log\_fname);**

**exit(1);**

**}**

**//현재시각 추출**

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

**char tmp\_buf[BUF\_SIZE];**

**memset(tmp\_buf, 0, BUF\_SIZE);**

**//현재시각을 문자열로 변경하고 맨뒤에 개행문자를 널문자로 변경함**

**strcpy(tmp\_buf, ctime(&cur\_time));**

**tmp\_buf[strlen(tmp\_buf) - 1] = 0;**

**//로그파일에 command-line에서 입력받은 명령어를 출력시켜줌**

**fprintf(log\_file\_fp, "[%s] ssu\_rsync", tmp\_buf);**

**for(int i=1; i<argc; i++)**

**{**

**fprintf(log\_file\_fp, " %s", argv[i]);**

**}**

**fprintf(log\_file\_fp, "\n");**

**fclose(log\_file\_fp);**

**}**

**//로그 작성**

**print\_log(src,dst);**

**gettimeofday(&end\_t, NULL);**

**ssu\_runtime(&begin\_t, &end\_t);**

**}**

**//SIGINT 발생시 처리함수**

**void signal\_handler()**

**{**

**//현재 dst에 존재하는 파일들을 리스트화**

**List \*new\_list = make\_list(glob\_dst);**

**List \*move1 = old\_dst, \*move2 = new\_list;**

**struct stat statbuf;**

**while(move2 != NULL)**

**{**

**int flag = 0;**

**//리스트 맨 처음으로 되돌아감**

**move1 = old\_dst;**

**while(move1 != NULL)**

**{**

**//같은 이름의 파일이 존재할 경우 pass**

**if(!strcmp(move1->fname, move2->fname))**

**{**

**flag = 1;**

**break;**

**}**

**move1 = move1->next;**

**}**

**//현재 dst디렉토리에 처음에 저장했던 dst 정보에 파일이 존재하지않으면 삭제를 진행함으로써 원래대로 복구시킴**

**if(flag == 0)**

**{**

**char \*tmp = (char \*)calloc(BUF\_SIZE, sizeof(char));**

**memset(tmp, 0, BUF\_SIZE);**

**sprintf(tmp, "%s/%s", glob\_dst, move2->fname);**

**stat(tmp, &statbuf);**

**//디렉토리일경우 따로 삭제함수를 호출**

**if(S\_ISDIR(statbuf.st\_mode))**

**rmdirs(tmp);**

**else**

**remove(tmp);**

**}**

**move2 = move2->next;**

**}**

**exit(1);**

**}**

**//t 옵션 수행 함수**

**void do\_tOption(char \*src, char \*dst)**

**{**

**struct stat src\_statbuf, dst\_statbuf;**

**struct stat statbuf;**

**struct dirent \*src\_dentry, \*dst\_dentry;**

**DIR \*src\_dp, \*dst\_dp;**

**int index = 0;**

**char buf[BUF\_SIZE];**

**char src\_buf[BUF\_SIZE];**

**char dst\_buf[BUF\_SIZE];**

**char need\_sync[1000][BUF\_SIZE];**

**stat(src, &statbuf);**

**//일반파일의 경우**

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

**{**

**strcpy(need\_sync[index++], src);**

**}**

**//디렉토리를 동기화 하는경우**

**else**

**{**

**//src와 dst 디렉토리 각각 opendir 해줌**

**if((src\_dp = opendir(src)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", src);**

**return;**

**}**

**if((dst\_dp = opendir(dst)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", dst);**

**return;**

**}**

**//src의 한파일과 dst 전부를 비교하여 동기화가 필요한지 판단함**

**while((src\_dentry = readdir(src\_dp)) != NULL)**

**{**

**//"." 와 ".."은 무시함**

**if(!strcmp(src\_dentry->d\_name, ".") || !strcmp(src\_dentry->d\_name, "..")) continue;**

**memset(src\_buf, 0, BUF\_SIZE);**

**sprintf(src\_buf, "%s/%s", src, src\_dentry->d\_name);**

**stat(src\_buf, &src\_statbuf);**

**//디렉토리는 t옵션에서 동기화 하지 않음**

**if(S\_ISDIR(src\_statbuf.st\_mode)) continue;**

**//dst 디렉토리 포인터 처음으로 되돌림**

**rewinddir(dst\_dp);**

**int flag = 0;**

**while((dst\_dentry = readdir(dst\_dp)) != NULL)**

**{**

**//"." 와 ".."은 무시함**

**if(!strcmp(dst\_dentry->d\_name, ".") || !strcmp(dst\_dentry->d\_name, "..")) continue;**

**memset(dst\_buf, 0, BUF\_SIZE);**

**sprintf(dst\_buf, "%s/%s", dst, dst\_dentry->d\_name);**

**stat(dst\_buf, &dst\_statbuf);**

**//파일이름, 크기 , 최종수정일이 같을 경우 같은 파일 동기화x**

**if(!strcmp(src\_dentry->d\_name, dst\_dentry->d\_name) && src\_statbuf.st\_size == dst\_statbuf.st\_size && src\_statbuf.st\_mtime == dst\_statbuf.st\_mtime)**

**{**

**flag = 1;**

**break;**

**}**

**//파일이름은 같지만 크기 , 최종수정일이 다를 경우 다른파일로 인식**

**else if(!strcmp(src\_dentry->d\_name, dst\_dentry->d\_name) && (src\_statbuf.st\_size != dst\_statbuf.st\_size || src\_statbuf.st\_mtime != dst\_statbuf.st\_mtime))**

**{**

**flag = 2;**

**break;**

**}**

**}**

**//동기화가 필요하면 배열에 저장하여 이름을 저장해둠**

**if(flag == 0 || flag == 2)**

**{**

**strcpy(need\_sync[index++], src\_dentry->d\_name);**

**}**

**}**

**}**

**if(index == 0)**

**exit(1);**

**//doing tar prcoess**

**struct stat tar\_statbuf;**

**char command[BUF\_SIZE \* 2];**

**char process\_dir[BUF\_SIZE];**

**//현재 작업디렉토리 경로를 미리저장**

**getcwd(process\_dir, BUF\_SIZE);**

**//src 디렉토리로 이동함**

**chdir(src);**

**memset(command, 0 , sizeof(command));**

**strcpy(command, tar\_command);**

**strcat(command, " ");**

**memset(buf, 0, BUF\_SIZE);**

**//tar파일이 어디 생성될지 결정하는 문자열**

**//현재디렉토리/dst디렉토리/tar파일 이름**

**sprintf(buf, "%s/%s/%s", process\_dir, dst, tar\_fname);**

**strcat(command, buf);**

**strcat(command, " ");**

**//명령어 압축할 파일들을 나열해줌**

**//ex tar -cf ./1.txt ./2.txt ./3.txt .....**

**for(int i=0; i<index; i++)**

**{**

**memset(buf, 0, BUF\_SIZE);**

**sprintf(buf, "./%s", need\_sync[i]);**

**strcat(command, buf);**

**strcat(command, " ");**

**}**

**//압축 진행**

**system(command);**

**chdir(process\_dir);**

**chdir(dst);**

**//tar파일의 크기를 얻기위함**

**stat("temp.tar", &tar\_statbuf);**

**memset(command, 0, sizeof(command));**

**//압축해제 명령어 제작**

**sprintf(command, "tar -xf %s", tar\_fname);**

**//압축 해제시킴**

**system(command);**

**//tar파일은 삭제함**

**remove("temp.tar");**

**//다시 현재 작업디렉토리로 돌아와서 로그 작성을 진행함**

**chdir(process\_dir);**

**FILE \*log\_file;**

**char \*log\_fname = "ssu\_rsync\_log";**

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

**memset(buf, 0, BUF\_SIZE);**

**//현재시각을 문자열화 시킴**

**strcpy(buf, ctime(&cur\_time));**

**buf[strlen(buf) - 1] = 0;**

**//로그파일을 append모드로 열기**

**if((log\_file = fopen(log\_fname , "a")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", log\_fname);**

**exit(1);**

**}**

**//현재시각과 명령어 정보 출력**

**fprintf(log\_file, "[%s] ssu\_rsync -t %s %s\n", buf, src, dst);**

**//tar파일의 크기도 출력**

**fprintf(log\_file, "\ttotalSize %ldbytes\n", tar\_statbuf.st\_size);**

**//동기화한 파일들의 이름과 크기도 로그에 작성**

**for(int i=0; i<index; i++)**

**{**

**//디렉토리를 t옵션으로 동기화하는경우**

**//파일 크기를 얻기 위한 작업**

**if(S\_ISDIR(statbuf.st\_mode))**

**{**

**struct stat statbuf2;**

**memset(buf, 0, BUF\_SIZE);**

**sprintf(buf, "%s/%s", src, need\_sync[i]);**

**stat(buf, &statbuf2);**

**fprintf(log\_file, "\t%s %ldbytes\n", need\_sync[i], statbuf2.st\_size);**

**}**

**//일반파일을 동기화하는경우 그냥 로그 출력**

**else**

**{**

**fprintf(log\_file, "\t%s %ldbytes\n", need\_sync[i], statbuf.st\_size);**

**}**

**}**

**fclose(log\_file);**

**}**

**//-m 옵션 수행 함수**

**void do\_mOption(char \*src, char \*dst)**

**{**

**struct stat src\_statbuf, dst\_statbuf;**

**struct dirent \*src\_dentry, \*dst\_dentry;**

**DIR \*src\_dp, \*dst\_dp;**

**char buf[BUF\_SIZE];**

**char src\_buf[BUF\_SIZE];**

**char dst\_buf[BUF\_SIZE];**

**memset(src\_buf, 0, BUF\_SIZE);**

**memset(dst\_buf, 0, BUF\_SIZE);**

**memset(buf, 0, BUF\_SIZE);**

**//src , dst 디렉토리를 open**

**if((src\_dp = opendir(src)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", src);**

**exit(1);**

**}**

**if((dst\_dp = opendir(dst)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", dst);**

**exit(1);**

**}**

**//dst파일 하나와 src디렉토리를 전부 순회하면서 비교**

**while((dst\_dentry = readdir(dst\_dp)) != NULL)**

**{**

**//"." 과 ".."은 무시**

**if(!strcmp(dst\_dentry->d\_name, "..") || !strcmp(dst\_dentry->d\_name, ".")) continue;**

**//파일의 존재여부 판단 flag**

**int flag = 0;**

**memset(dst\_buf, 0, BUF\_SIZE);**

**sprintf(dst\_buf, "%s/%s", dst, dst\_dentry->d\_name);**

**stat(dst\_buf, &dst\_statbuf);**

**//디렉토리 포인터 맨 처음으로 되돌려서 각 파일 별로 src디렉토리를 전부 순회하게 만듬**

**rewinddir(src\_dp);**

**while((src\_dentry = readdir(src\_dp)) != NULL)**

**{**

**//"." 과 ".."은 무시**

**if(!strcmp(src\_dentry->d\_name, "..") || !strcmp(src\_dentry->d\_name, ".")) continue;**

**memset(src\_buf, 0, BUF\_SIZE);**

**sprintf(src\_buf, "%s/%s", src, src\_dentry->d\_name);**

**stat(src\_buf, &src\_statbuf);**

**//파일의 이름, 크기, 최종수정일이 같으면 같은 파일이라고 판단**

**if(!strcmp(dst\_dentry->d\_name, src\_dentry->d\_name) && src\_statbuf.st\_size == dst\_statbuf.st\_size && src\_statbuf.st\_mtime == dst\_statbuf.st\_mtime)**

**{**

**flag = 1;**

**break;**

**}**

**}**

**//dst의 하나의 파일이 src디렉토리에 존재하지 않을 경우**

**if(flag == 0)**

**{**

**//로그파일을 open하여 로그 작성**

**FILE \*log\_file;**

**char \*log\_fname = "ssu\_rsync\_log";**

**if((log\_file = fopen(log\_fname, "a+")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", log\_fname);**

**exit(1);**

**}**

**//현재시각과 명령어 정보를 처음 출력하는 경우에만 출력해줌**

**if(first\_flag == 0)**

**{**

**//현재 시간 정보를 얻어옴**

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

**char buf2[BUF\_SIZE];**

**memset(buf2, 0, BUF\_SIZE);**

**//문자열 형태로 바꾸어서 buf2에 저장**

**strcpy(buf2, ctime(&cur\_time));**

**//맨 뒤에 개행 문자 삭제**

**buf2[strlen(buf2) - 1] = 0;**

**//로그 작성시각 및 명령어 정보 로그파일에 작성**

**fprintf(log\_file, "[%s] ssu\_rsync -m %s %s\n", buf2, src, dst);**

**first\_flag = 1;**

**}**

**//삭제된 파일 이름과 크기를 출력**

**fprintf(log\_file, "\t%s delete\n", dst\_dentry->d\_name);**

**fclose(log\_file);**

**//디렉토리일경우 따로 작성한 디렉토리 삭제함수 호출**

**if(S\_ISDIR(dst\_statbuf.st\_mode))**

**rmdirs(dst\_buf);**

**else**

**remove(dst\_buf);**

**}**

**}**

**}**

**//디렉토리 삭제함수**

**void rmdirs(char \*path)**

**{**

**struct dirent \*dirp;**

**struct stat statbuf;**

**DIR \*dp;**

**char \*tmp = (char \*)calloc(BUF\_SIZE, sizeof(char));**

**//인자로 받은 디렉토리 오픈**

**if((dp = opendir(path)) == NULL)**

**return;**

**while((dirp = readdir(dp)) != NULL)**

**{**

**//"." 과 ".."은 무시**

**if(!strcmp(dirp->d\_name, ".") || !strcmp(dirp->d\_name, "..")) continue;**

**sprintf(tmp, "%s/%s", path, dirp->d\_name);**

**//파일 정보를 불러움. 디렉토리인지 아닌지 판별할때 사용**

**if(lstat(tmp, &statbuf) == -1)**

**continue;**

**//서브 디렉토리가 존재한다면 서브디렉토리로 들어가 내부 파일 삭제**

**if(S\_ISDIR(statbuf.st\_mode))**

**rmdirs(tmp);**

**//일반파일의 경우 그냥 unlink**

**else**

**unlink(tmp);**

**}**

**//내부 파일 삭제가 끝나면 지금 디렉토리도 삭제시킴**

**closedir(dp);**

**rmdir(path);**

**}**

**//사용법 출력**

**void print\_usage()**

**{**

**fprintf(stderr, "usage: ./ssu\_rsync [option] <src> <dst>\n");**

**gettimeofday(&end\_t, NULL);**

**ssu\_runtime(&begin\_t, &end\_t);**

**exit(1);**

**}**

**//동기화 시켜주는 함수**

**void Sync(char \*src, char\* dst, char \*sub\_dir, int argc, char \*\*argv, int depth)**

**{**

**DIR \*dp;**

**struct stat statbuf;**

**struct stat statbuf2;**

**struct flock lock;**

**struct dirent \*dentry;**

**char buf[BUF\_SIZE];**

**char file\_name[BUF\_SIZE];**

**memset(buf, 0, BUF\_SIZE);**

**memset(file\_name, 0, BUF\_SIZE);**

**strcpy(buf, src);**

**//src에서 파일이름만을 추출함**

**char \*p = strtok(buf, "/");**

**char \*pre;**

**while(p != NULL)**

**{**

**pre = p;**

**p = strtok(NULL, "/");**

**}**

**strcpy(file\_name, pre);**

**//src파일의 정보를 저장함 -> 디렉토리인지 파일인지 판별하기 위함**

**if(stat(src, &statbuf) < 0)**

**{**

**fprintf(stderr, "stat error\n");**

**exit(1);**

**}**

**//dst 디렉토리를 오픈함**

**if((dp = opendir(dst)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", dst);**

**exit(1);**

**}**

**//일반파일을 동기화 하는경우**

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

**{**

**//파일이 존재하는지 여부 판단용 flag**

**int flag = 0;**

**while((dentry = readdir(dp)) != NULL)**

**{**

**//"." 와 ".."은 무시**

**if(!strcmp(dentry->d\_name, "..") || !strcmp(dentry->d\_name, ".")) continue;**

**sprintf(buf, "%s/%s", dst, dentry->d\_name);**

**stat(buf, &statbuf2);**

**//파일의 이름, 크기, 최종수정시각이 같으면 같은 파일임으로 동기화할 필요없음**

**if(!strcmp(dentry->d\_name, file\_name) && statbuf.st\_size == statbuf2.st\_size && statbuf.st\_mtime == statbuf2.st\_mtime)**

**{**

**flag = 1;**

**break;**

**}**

**//이름은같은데 크기, 최종수정시각이 다를경우 동기화 필요**

**else if (!strcmp(dentry->d\_name, file\_name) && (statbuf.st\_size != statbuf2.st\_size || statbuf.st\_mtime != statbuf2.st\_mtime))**

**{**

**flag = 2;**

**break;**

**}**

**}**

**//0 의경우 파일이 아예 존재하지 않는 경우**

**//2 의경우 파일이 존재하지만 크기나 수정시각이 다를 경우**

**if(flag == 0 || flag == 2)**

**{**

**//동기화 진행**

**normal\_sync(NULL, dst, file\_name);**

**//로그파일을 오픈하여 작성하는 작업 진행**

**FILE \*log\_file;**

**char \*log\_fname = "ssu\_rsync\_log";**

**if((log\_file = fopen(log\_fname, "a+")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", log\_fname);**

**exit(1);**

**}**

**//로그의 시작부분을 아직 작성안했다면 작성하고 flag를 설정하여 다른곳에서 첫줄을 다시 작성하지 않도록 유도함**

**if(first\_flag == 0)**

**{**

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

**char buf2[BUF\_SIZE];**

**memset(buf2, 0, BUF\_SIZE);**

**strcpy(buf2, ctime(&cur\_time));**

**buf2[strlen(buf2) - 1] = 0;**

**fprintf(log\_file, "[%s] ssu\_rsync %s %s\n", buf2, src, dst);**

**first\_flag = 1;**

**}**

**//파일의 이름과 크기 작성**

**fprintf(log\_file, "\t%s %ldbytes\n", file\_name, statbuf.st\_size);**

**fclose(log\_file);**

**}**

**}**

**//디렉토리를 동기화 하는경우**

**else**

**{**

**DIR \*src\_dp, \*dst\_dp;**

**struct dirent \*src\_dentry, \*dst\_dentry;**

**struct stat src\_statbuf, dst\_statbuf;**

**char src\_buf[BUF\_SIZE], dst\_buf[BUF\_SIZE];**

**//src와 dst 모두 디렉토리임으로 open 함**

**if((src\_dp = opendir(src)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", src);**

**exit(1);**

**}**

**if((dst\_dp = opendir(dst)) == NULL)**

**{**

**fprintf(stderr, "opendir error for %s\n", dst);**

**exit(1);**

**}**

**//src의 한 파일과 dst디렉토리 내부 모든 파일과 비교하는 과정을 거침**

**while((src\_dentry = readdir(src\_dp)) != NULL)**

**{**

**//"."와 ".."은 무시**

**if(!strcmp(src\_dentry->d\_name, ".") || !strcmp(src\_dentry->d\_name, "..")) continue;**

**//파일의 존재여부를 판단하는 flag**

**int flag=0;**

**memset(src\_buf, 0, BUF\_SIZE);**

**sprintf(src\_buf, "%s/%s", src, src\_dentry->d\_name);**

**stat(src\_buf, &src\_statbuf);**

**//디렉토리 포인터 처음으로 돌려서 src의 각 파일이 dst의 모든 파일과 비교하도록 설정**

**rewinddir(dst\_dp);**

**while((dst\_dentry = readdir(dst\_dp)) != NULL)**

**{**

**//"."와 ".."은 무시**

**if(!strcmp(dst\_dentry->d\_name, ".") || !strcmp(dst\_dentry->d\_name, "..")) continue;**

**memset(dst\_buf, 0, BUF\_SIZE);**

**sprintf(dst\_buf, "%s/%s", dst, dst\_dentry->d\_name);**

**stat(dst\_buf, &dst\_statbuf);**

**//파일의 이름과 크기,최종수정일이 모두 같을때**

**if(!strcmp(src\_dentry->d\_name, dst\_dentry->d\_name) && src\_statbuf.st\_size == dst\_statbuf.st\_size && src\_statbuf.st\_mtime == dst\_statbuf.st\_mtime)**

**{**

**flag = 1;**

**break;**

**}**

**//파일의 이름은 같지만 크기, 최종수정일이 다를 경우 동기화가 필요함**

**else if(!strcmp(src\_dentry->d\_name, dst\_dentry->d\_name) && (src\_statbuf.st\_size != dst\_statbuf.st\_size || src\_statbuf.st\_mtime != dst\_statbuf.st\_mtime))**

**{**

**flag = 2;**

**break;**

**}**

**}**

**//0 의경우 파일이 아예 존재하지 않는 경우**

**//2 의경우 파일이 존재하지만 크기나 수정시각이 다를 경우**

**if(flag == 0 || flag == 2)**

**{**

**//src의 한 파일이 디렉토리가 아닐경우**

**if(!S\_ISDIR(src\_statbuf.st\_mode))**

**{**

**//동기화 진행**

**normal\_sync(src, dst, src\_dentry->d\_name);**

**char \*temp = (char \*)calloc(BUF\_SIZE, sizeof(char));**

**//src 바로 밑에 있을 경우**

**if(depth == 0)**

**{**

**strcpy(temp, src\_dentry->d\_name);**

**}**

**//src디렉토리안의 서브디렉토리에 있을 경우 경로를 추가**

**else**

**{**

**sprintf(temp, "%s/%s", sub\_dir, src\_dentry->d\_name);**

**}**

**//새로운 노드를 생성하여 동기화가 필요한 파일의 정보를 담고 있는 sync\_list에 추가해줌**

**List \*new\_node = make\_node(temp);**

**new\_node->size = src\_statbuf.st\_size;**

**new\_node->next = NULL;**

**new\_node->child = NULL;**

**if(sync\_list == NULL)**

**sync\_list = new\_node;**

**else**

**{**

**new\_node->next = sync\_list;**

**sync\_list = new\_node;**

**}**

**}**

**//디렉토리의 경우**

**else**

**{**

**//r 옵션이 존재하지않을 경우 동기화 x**

**if(rOption == FALSE)**

**continue;**

**//r 옵션이 존재할 경우**

**else**

**{**

**struct utimbuf time\_buf;**

**time\_buf.actime = src\_statbuf.st\_atime;**

**time\_buf.modtime = src\_statbuf.st\_mtime;**

**memset(buf, 0, BUF\_SIZE);**

**sprintf(buf, "%s/%s", dst, src\_dentry->d\_name);**

**//dst내에 서브디렉토리를 생성, 접근권한은 원본파일과 동일하게 설정**

**mkdir(buf, src\_statbuf.st\_mode);**

**char new\_src[BUF\_SIZE];**

**char new\_dst[BUF\_SIZE];**

**char new\_sub\_dir[BUF\_SIZE];**

**memset(new\_src, 0, BUF\_SIZE);**

**memset(new\_dst, 0, BUF\_SIZE);**

**memset(new\_sub\_dir, 0, BUF\_SIZE);**

**//src내의 서브디렉토리안의 디렉토리일 경우 경로 추가**

**if(sub\_dir != NULL)**

**{**

**sprintf(new\_sub\_dir, "%s/%s", sub\_dir, src\_dentry->d\_name);**

**}**

**else**

**{**

**sprintf(new\_sub\_dir, "%s", src\_dentry->d\_name);**

**}**

**sprintf(new\_src, "%s/%s", src, src\_dentry->d\_name);**

**sprintf(new\_dst, "%s/%s", dst, src\_dentry->d\_name);**

**//서브 디렉토리의 동기화 진행을 위해 Sync함수 재호출**

**Sync(new\_src, new\_dst, new\_sub\_dir, argc, argv, depth+1);**

**//atime과 mtime을 원본파일과 동일 시 시켜줌**

**utime(buf, &time\_buf);**

**}**

**}**

**}**

**}**

**}**

**}**

**//실질적인 파일 복사가 이루어지는 동기화 함수**

**void normal\_sync(char \*src, char \*dst, char \*fname)**

**{**

**struct flock lock;**

**struct stat src\_statbuf;**

**struct utimbuf time\_buf;**

**int src\_fd, dst\_fd;**

**char dst\_path[BUF\_SIZE];**

**char src\_path[BUF\_SIZE];**

**memset(dst\_path, 0, BUF\_SIZE);**

**memset(src\_path, 0, BUF\_SIZE);**

**if (src == NULL) strcpy(src\_path, fname);**

**else sprintf(src\_path, "%s/%s", src, fname);**

**sprintf(dst\_path, "%s/%s", dst, fname);**

**//원본 파일을 읽기 전용으로 오픈함**

**if((src\_fd = open(src\_path, O\_RDONLY)) < 0)**

**{**

**fprintf(stderr, "open error for %s\n", src);**

**exit(1);**

**}**

**//원본 파일의 atime과 mtime을 얻어옴**

**stat(src\_path, &src\_statbuf);**

**time\_buf.actime = src\_statbuf.st\_atime;**

**time\_buf.modtime = src\_statbuf.st\_mtime;**

**//dst에 같은 이름의 파일을 새로 오픈함, 접근권한은 원본파일과 동일하게 설정**

**if((dst\_fd = open(dst\_path, O\_RDWR|O\_CREAT|O\_TRUNC, src\_statbuf.st\_mode)) < 0)**

**{**

**fprintf(stderr, "open error for %s\n", dst\_path);**

**exit(1);**

**}**

**//원본파일에 쓰기 lock을 걸어서 수정이 안되도록 설정**

**lock.l\_type = F\_WRLCK;**

**lock.l\_whence = 0;**

**lock.l\_start = 0l;**

**lock.l\_len = 0l;**

**fcntl(src\_fd, F\_SETLK, &lock);**

**//src의 원본파일 내용을 -> dst의 새로생성한 파일에 복사**

**char buf[BUF\_SIZE];**

**memset(buf, 0, BUF\_SIZE);**

**while(read(src\_fd, buf, BUF\_SIZE) > 0)**

**{**

**write(dst\_fd, buf, strlen(buf));**

**memset(buf, 0, BUF\_SIZE);**

**}**

**//락을 해제 함**

**lock.l\_type = F\_UNLCK;**

**fcntl(src\_fd, F\_SETLK, &lock);**

**close(src\_fd);**

**close(dst\_fd);**

**//새로생성한 파일의 atime과 mtime을 원본파일의것으로 변경함**

**utime(dst\_path, &time\_buf);**

**}**

**//넘겨 받은 인자로 노드를 생성하여 리턴시켜주는 함수**

**List \*make\_node(char \*fname)**

**{**

**List \*new\_node = (List \*)calloc(1, sizeof(List));**

**strcpy(new\_node->fname, fname);**

**new\_node->next = NULL;**

**new\_node->child = NULL;**

**return new\_node;**

**}**

**//인자로 넘겨받은 디렉토리의 파일정보를 리스트화 시킴**

**List \*make\_list(char \*path)**

**{**

**List \*root = NULL;**

**DIR \*dp;**

**struct dirent \*dentry;**

**struct stat statbuf;**

**char buf[BUF\_SIZE];**

**memset(buf, 0, BUF\_SIZE);**

**//디렉토리를 오픈**

**if((dp = opendir(path)) == NULL)**

**{**

**fprintf(stderr, "open error for %s\n", path);**

**return NULL;**

**}**

**while((dentry = readdir(dp)) != NULL)**

**{**

**//"." 와 ".."은 무시**

**if(!strcmp(dentry->d\_name , ".") || !strcmp(dentry->d\_name, "..")) continue;**

**//파일의 이름으로 새로운 노드를 생성함**

**List \*new\_node = make\_node(dentry->d\_name);**

**char \*temp = (char \*)calloc(BUF\_SIZE, sizeof(char));**

**sprintf(temp, "%s/%s", path, dentry->d\_name);**

**stat(temp, &statbuf);**

**//만약 디렉토리일 경우 child를 루트로 하여 서브디렉토리의 리스트를 생성하여 연결**

**if(S\_ISDIR(statbuf.st\_mode))**

**{**

**new\_node->child = make\_list(temp);**

**}**

**//항상 리스트 맨 앞에 연결**

**if(root == NULL)**

**root = new\_node;**

**else**

**{**

**new\_node->next = root;**

**root = new\_node;**

**}**

**}**

**return root;**

**}**

**/\***

**List \*print\_list(List \*root)**

**{**

**while(root != NULL)**

**{**

**printf("%s\n", root->fname);**

**if(root->child != NULL)**

**print\_list(root->child);**

**root = root->next;**

**}**

**}**

**\*/**

**//로그 출력 함수**

**void print\_log(char \*src, char \*dst)**

**{**

**//동기화 시킬 파일이 하나도 없을 경우 로그작성 x**

**if(sync\_list == NULL) return;**

**FILE \*fp;**

**//현재시각을 불러와 저장**

**time\_t cur\_time;**

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

**char \*fname = "ssu\_rsync\_log";**

**char buf[BUF\_SIZE];**

**memset(buf, 0, BUF\_SIZE);**

**//현재시각을 문자열로 변경해수 맨 뒤에 개행문자 삭제**

**strcpy(buf, ctime(&cur\_time));**

**buf[strlen(buf) - 1] = 0;**

**//append모드로 로그 파일 오픈**

**if((fp = fopen(fname, "a+")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", fname);**

**return;**

**}**

**//현재시각과 명령어 정보를 처음 출력하는 경우**

**if(first\_flag == 0)**

**fprintf(fp, "[%s] ssu\_rsync %s %s\n", buf, src, dst);**

**//동기화가 필요한 파일의 이름과 크기 정보를 담고 있는 리스트를 순회하면서 로그파일에 작성**

**List \*move = sync\_list;**

**while(move != NULL)**

**{**

**fprintf(fp, "\t%s %ldbytes\n", move->fname, move->size);**

**move = move->next;**

**}**

**fclose(fp);**

**return;**

**}**

**//수행시간 측정함수**

**void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t)**

**{**

**//끝난 시각에서 시작한 시각을 뺌**

**end\_t->tv\_sec -= begin\_t->tv\_sec;**

**//만약 끝난시각의 밀리초가 더 작으면**

**//끝난시각의 초를 하나 줄이고 마이크로초를 증가시킴**

**if(end\_t->tv\_usec < begin\_t->tv\_usec)**

**{**

**end\_t->tv\_sec--;**

**end\_t->tv\_usec += SECOND\_TO\_MICRO;**

**}**

**end\_t->tv\_usec -= begin\_t->tv\_usec;**

**printf("Runtime: %ld:%06ld(sec:usec)\n", end\_t->tv\_sec, end\_t->tv\_usec);**

**}**

**<ssu\_crond.c>**

**#include <stdio.h>**

**#include <ctype.h>**

**#include <signal.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include <time.h>**

**#include <pthread.h>**

**#include <string.h>**

**#define BUF\_SIZE 1024**

**//리스트를 구성할 구조체**

**typedef struct \_Command**

**{**

**char cmd[BUF\_SIZE];**

**time\_t last\_exec;**

**pthread\_t tid;**

**struct \_Command \*next;**

**}Command;**

**//파일 수정여부를 확인하고 업데이트**

**//실행주기를 토큰화 시켜서 각 요소 별로 배열을 생성해**

**//실행될 시각에 대해서 flag를 켜서 현재시간과 총 5개의 요소 flag가 1이면 명령어를 수행한다**

**void check\_time(char (\*tok\_buf)[BUF\_SIZE], char (\*tokens)[BUF\_SIZE], int \*type\_array, int type);**

**//명령어별로 각 thread가 수행할 명령어를 인자로서 넘겨받고**

**//주기적으로 명령어를 수행한다.**

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

**//ssu\_crontab\_file이 수정되는지 확인한다**

**//파일이 수정되었을 경우 삭제된 명령어를 수행하는 thread를 종료시키고 리스트 정보를 업데이트**

**void \*check\_file\_modify(void \*arg);**

**//리스트 정보를 이용해 각 명령어별로 thread를 생성해주고 id를 리스트에 저장한다**

**void \*check\_cmd(void \*arg);**

**//리스트에 필요한 노드를 생성하고 리턴시키는 함수**

**Command \*make\_node(char \*cmd);**

**//명령어에 대한 리스트와 그 명령어를 수행하는 thread\_id를 저장할 리스트를 제작한다**

**Command \*make\_list();**

**Command \*root = NULL;**

**pthread\_mutex\_t mutex\_lock = PTHREAD\_MUTEX\_INITIALIZER;**

**pthread\_mutex\_t mutex\_lock1 = PTHREAD\_MUTEX\_INITIALIZER;**

**int main()**

**{**

**FILE \*fp;**

**char \*fname = "ssu\_crontab\_file";**

**pid\_t pid;**

**if((pid = fork()) < 0)**

**exit(0);**

**else if(pid != 0)**

**exit(0);**

**signal(SIGHUP, SIG\_IGN);**

**close(0); close(1); close(2);**

**setsid();**

**while(1)**

**{**

**//ssu\_crontab\_file이 존재할때 까지 대기**

**if(access(fname, F\_OK) < 0)**

**{**

**continue;**

**}**

**else**

**{**

**root = make\_list();**

**break;**

**}**

**}**

**//2개의 스레드를 생성하여 하나는 파일 수정체크용 스레드**

**//하나는 명령어별 스레드 생성 여부 판단을 위한 스레드 생성**

**pthread\_t tid, tid2;**

**if(pthread\_create(&tid, NULL, check\_file\_modify, NULL) != 0)**

**{**

**fprintf(stderr, "pthread\_create error\n");**

**exit(1);**

**}**

**//일정간격으로 리스트에서 아직 출발하지 않은 명령어에 대한 스레드를 생성해주는 스레드**

**if(pthread\_create(&tid2, NULL, check\_cmd, NULL) != 0)**

**{**

**fprintf(stderr, "pthread\_create error\n");**

**exit(1);**

**}**

**//각 스레드가 끝날때 까지 대기**

**int status;**

**pthread\_join(tid, (void \*)&status);**

**pthread\_join(tid2, (void \*)&status);**

**}**

**//ssu\_crontab\_file이 수정되는지 감시하는 스레드가 할 함수**

**void \*check\_file\_modify(void \*arg)**

**{**

**FILE \*fp;**

**char \*fname = "ssu\_crontab\_file";**

**struct stat statbuf;**

**long size;**

**time\_t mtime;**

**Command \*new\_root = NULL;**

**//초기 파일의 크기와 수정시각을 저장**

**stat(fname, &statbuf);**

**mtime = statbuf.st\_mtime;**

**size = statbuf.st\_size;**

**//혹시 모르는 상황을 대비해 mutex\_lock을 설정**

**pthread\_mutex\_lock(&mutex\_lock1);**

**while(1)**

**{**

**stat(fname, &statbuf);**

**//만약 크기나 최종수정일이 변경되었을 경우**

**if(size != statbuf.st\_size || mtime != statbuf.st\_mtime)**

**{**

**//최신 정보를 저장한뒤**

**size = statbuf.st\_size;**

**mtime = statbuf.st\_mtime;**

**//ssu\_crontab\_file의 명령어 정보를 리스트화 시킴**

**new\_root = make\_list();**

**//기존의 리스트와 비교를 진행**

**//move1 -> 기존 리스트**

**//move2 -> 변경된파일내용을 담은 리스트**

**Command \*move1 = root, \*move2 = new\_root;**

**while(move1 != NULL)**

**{**

**int flag = 0;**

**move2 = new\_root;**

**while(move2 != NULL)**

**{**

**//같은 내용이 존재할경우 패스**

**if(!strcmp(move1->cmd, move2->cmd))**

**{**

**move2->tid = move1->tid;**

**flag = 1;**

**break;**

**}**

**move2 = move2->next;**

**}**

**//같은 내용의 명령어를 찾지 못 했을 경우**

**//기존 리스트에 저장된 스레드 id를 통해 thread를 강제 종료시킴**

**if(flag == 0)**

**{**

**pthread\_cancel(move1->tid);**

**}**

**move1 = move1->next;**

**}**

**//리스트 갱신**

**root = new\_root;**

**}**

**//1초단위로 검사**

**sleep(1);**

**}**

**pthread\_mutex\_unlock(&mutex\_lock1);**

**}**

**//각 스레드가 수행할 함수**

**void \*thread\_fun(void \*arg)**

**{**

**time\_t cur\_time;**

**struct tm \*t;**

**//실행할 주기를 설정할 배열들**

**int minute[60];**

**int hour[24];**

**int day[32];**

**int month[13];**

**int week[7];**

**//배열 초기화**

**for(int i=0; i<60; i++) minute[i] = 0;**

**for(int i=0; i<24; i++) hour[i] = 0;**

**for(int i=0; i<32; i++) day[i] = 0;**

**for(int i=0; i<13; i++) month[i] = 0;**

**for(int i=0; i<7; i++) week[i] = 0;**

**//문자열 정보 배열 초기화**

**char tok\_buf[500][BUF\_SIZE];**

**char arg\_data[BUF\_SIZE];**

**char buf[BUF\_SIZE];**

**char real\_cmd[BUF\_SIZE];**

**char tokens[5][BUF\_SIZE];**

**memset(arg\_data, 0, BUF\_SIZE);**

**memset(buf, 0, BUF\_SIZE);**

**memset(real\_cmd, 0, BUF\_SIZE);**

**for(int i=0; i<5; i++) memset(tokens[i], 0, BUF\_SIZE);**

**//인자로 넘겨받은 데이터를 따로 보관**

**strcpy(arg\_data, (char \*)arg);**

**strcpy(buf, arg\_data);**

**//실행주기를 공백 기준으로 토큰화 시킴**

**char \*p;**

**p = strtok(buf, " ");**

**strcpy(tokens[0], p);**

**for(int i=1; i<5; i++)**

**{**

**p = strtok(NULL, " ");**

**strcpy(tokens[i], p);**

**}**

**strcpy(real\_cmd, strtok(NULL, ""));**

**//토큰화 시킨 주기를 숫자와 기호에 맞게 다시 토큰화 시켜서 3번째인자 배열에다가**

**//해당하는 주기를 1로 변화시킴**

**check\_time(tok\_buf, tokens, minute, 0);**

**check\_time(tok\_buf, tokens, hour, 1);**

**check\_time(tok\_buf, tokens, day, 2);**

**check\_time(tok\_buf, tokens, month, 3);**

**check\_time(tok\_buf, tokens, week, 4);**

**//??분 00초를 기준으로 명령어를 실행하기위해 00초까지 대기**

**while(1)**

**{**

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

**t = localtime(&cur\_time);**

**if(t->tm\_sec == 0)**

**break;**

**sleep(1);**

**}**

**while(1)**

**{**

**//현재시간을 가져와서 저장**

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

**t = localtime(&cur\_time);**

**char time\_buf[BUF\_SIZE];**

**strcpy(time\_buf, ctime(&cur\_time));**

**time\_buf[strlen(time\_buf) - 1] = 0;**

**//현재시간과 각 배열의 값이 모두 1일 경우에만 명령어를 실행**

**//하나라도 1이 아닐경우 실행할 주기가 아님을 판단**

**if(minute[t->tm\_min] == 1 && hour[t->tm\_hour] == 1 && day[t->tm\_mday] == 1 &&**

**month[t->tm\_mon] == 1 && week[t->tm\_wday] == 1)**

**{**

**FILE \*fp;**

**char \*fname = "ssu\_crontab\_log";**

**char new\_buf[BUF\_SIZE];**

**memset(new\_buf, 0, BUF\_SIZE);**

**//로그파일을 오픈하여 실행했다는 로그를 작성함**

**if((fp = fopen(fname, "a")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", fname);**

**exit(1);**

**}**

**sprintf(new\_buf, "[%s] run %s\n", time\_buf, (char \*)arg);**

**fwrite(new\_buf, strlen(new\_buf), 1, fp);**

**fclose(fp);**

**//실제 명령어 실행**

**system(real\_cmd);**

**}**

**//60초 간격으로 주기를 확인**

**sleep(60);**

**}**

**}**

**//실행 주기를 분석하여 type\_array에 플래그를 설정시켜줌**

**void check\_time(char (\*tok\_buf)[BUF\_SIZE], char (\*tokens)[BUF\_SIZE], int \*type\_array, int type)**

**{**

**//버퍼 초기화**

**for(int i=0; i<500; i++) memset(tok\_buf[i], 0, BUF\_SIZE);**

**int index = 0;**

**//실행주기들을 -,/,\*,숫자에 따라 토큰화 시킴**

**for(int i=0; i<strlen(tokens[type]); i++)**

**{**

**//주기와 범위 그리고 모든값을 의미하는 기호의 경우 혼자서 토큰이 됨**

**if(tokens[type][i] == '/' || tokens[type][i] == '-' || tokens[type][i] == '\*')**

**strncat(tok\_buf[index++], &tokens[type][i], 1);**

**//앞에 토큰이 숫자고 현재 문자도 숫자면 앞에다가 연결시켜줌**

**else if(i>0 && isdigit(tokens[type][i]) && isdigit(tok\_buf[index-1][strlen(tok\_buf[index-1]) - 1]))**

**strncat(tok\_buf[index-1], &tokens[type][i] , 1);**

**//이외의 모든경우 그냥 토큰화시켜줌**

**else**

**strncat(tok\_buf[index++], &tokens[type][i], 1);**

**}**

**//단 한개의 정보만 있을 경우**

**if(index == 1)**

**{**

**//정보가 하나밖에 없는데 \*일경우**

**//모든 주기에 대해 플레그를 설정**

**if(tok\_buf[0][0] == '\*')**

**{**

**int max;**

**//0 -> 분, 1 -> 시, 2 -> 일, 3 -> 월, 4-> 요일**

**if(type == 0) max = 60;**

**else if(type == 1) max = 24;**

**else if(type == 2) max = 32;**

**else if(type == 3) max = 13;**

**else if(type == 4) max = 7;**

**//플래그 설정**

**for(int i=0; i<max; i++) type\_array[i] = 1;**

**}**

**else**

**type\_array[atoi(tok\_buf[0])] = 1;**

**}**

**else**

**{**

**//범위값 초기화**

**int r\_start = -1, r\_end = -1;**

**for(int i=0; i<index; i++)**

**{**

**// '-' 기호를 만났을 경우 앞 뒤 숫자가 범위가 됨**

**if(!strcmp(tok\_buf[i] , "-"))**

**{**

**//범위값을 정수화 시켜서 저장**

**r\_start = atoi(tok\_buf[i-1]);**

**r\_end = atoi(tok\_buf[i+1]);**

**i++;**

**int tmp = i + 1;**

**if(tmp>=index || (tmp < i && !strcmp(tok\_buf[tmp], ",")))**

**{**

**for(int z=r\_start; z<=r\_end; z++)**

**type\_array[z] = 1;**

**}**

**}**

**// "/"의 경우 앞에 주기를 뒤에 숫자만큼 점프시킴**

**else if(!strcmp(tok\_buf[i], "/"))**

**{**

**//만약 앞에 주기가 \*일 경우 type에 따라 최대범위 설정**

**if(!strcmp(tok\_buf[i-1] , "\*"))**

**{**

**if(type == 0)**

**{**

**r\_start = 0, r\_end = 59;**

**}**

**else if(type == 1)**

**{**

**r\_start = 0, r\_end = 23;**

**}**

**else if(type == 2)**

**{**

**r\_start = 1, r\_end = 31;**

**}**

**else if(type == 3)**

**{**

**r\_start = 1, r\_end = 12;**

**}**

**else if(type == 4)**

**{**

**r\_start = 0, r\_end = 6;**

**}**

**}**

**//점프할 주기는 "/" 뒤에 숫자**

**int jump = atoi(tok\_buf[i+1]);**

**//만약 범위가 있다면 범위에 해당하는 숫자와 점프를 생각하여 플래그 설정**

**if(r\_start != -1 && r\_end != -1)**

**{**

**int starting\_point = r\_start +jump - 1;**

**for(int j=starting\_point; j<=r\_end; j=j+jump)**

**type\_array[j] = 1;**

**}**

**i++;**

**}**

**//숫자일 경우**

**else if(isdigit(tok\_buf[i][0]))**

**{**

**//다음 토큰이 ","의 경우 현재 숫자는 단순 그 숫자 주기에 실행하라는 의미**

**//ex) 0,12일경우 0,12의 플래그만 설정**

**if(!strcmp(tok\_buf[i+1], ",") || i+1 == index)**

**{**

**type\_array[atoi(tok\_buf[i])] = 1;**

**}**

**}**

**else**

**continue;**

**}**

**}**

**}**

**//명령어에 대해 스레드를 생성할지 말지 결정함**

**void \*check\_cmd(void \*arg)**

**{**

**Command \*old = NULL;**

**time\_t cur\_time;**

**struct tm \*t;**

**char buf[BUF\_SIZE];**

**char real\_cmd[BUF\_SIZE];**

**memset(real\_cmd, 0, BUF\_SIZE);**

**memset(buf, 0, BUF\_SIZE);**

**//혹시 모르는 상황을 대비해 mutex\_lock을 설정**

**pthread\_mutex\_lock(&mutex\_lock);**

**while(1)**

**{**

**//2초전 리스트와 최신 리스트를 비교하기 위해 변수선언**

**Command \*move = root, \*move2 = old;**

**move = root; move2 = old;**

**while(move != NULL)**

**{**

**int flag = 0;**

**move2 = old;**

**//만약 이전리스트와 현재리스트에 동일한 명령어가 존재할시**

**//그 명령어에 대한 스레드가 존재함으로 추가적으로 생성할 필요없음**

**while(move2 != NULL)**

**{**

**//같은 명령어인지 아닌지 비교**

**if(!strcmp(move2->cmd, move->cmd))**

**{**

**flag = 1;**

**break;**

**}**

**move2 = move2 -> next;**

**}**

**//존재하는 명령어 대한 필터**

**if(flag == 1)**

**{**

**move = move->next;**

**continue;**

**}**

**//명령어에 대해 스레드가 없으면 스레드를 새로 생성해주고 새로 생성한 스레드의 id를 저장**

**char \*new\_buf = (char \*)calloc(BUF\_SIZE, sizeof(char));**

**strcpy(new\_buf, move->cmd);**

**new\_buf[strlen(new\_buf) - 1] = 0;**

**pthread\_t tid;**

**//스레드 생성부분**

**if(pthread\_create(&tid, NULL, thread\_fun, new\_buf) != 0)**

**{**

**fprintf(stderr, "pthread\_create error\n");**

**exit(1);**

**}**

**//id 저장부분**

**move->tid = tid;**

**move = move->next;**

**}**

**//현재리스트를 이전 리스트로 설정**

**old = root;**

**sleep(2);**

**}**

**pthread\_mutex\_unlock(&mutex\_lock);**

**}**

**//넘겨 받은 인자로 노드를 생성해서 리턴**

**Command \*make\_node(char \*cmd)**

**{**

**Command \*new\_node = (Command \*)calloc(1, sizeof(Command));**

**memset(new\_node->cmd, 0, BUF\_SIZE);**

**//인자 저장**

**strcpy(new\_node->cmd, cmd);**

**new\_node->last\_exec = 0;**

**new\_node->next = NULL;**

**return new\_node;**

**}**

**//리스트 생성함수**

**Command \*make\_list()**

**{**

**Command \*root = NULL;**

**FILE \*fp;**

**char \*fname = "ssu\_crontab\_file";**

**char buf[BUF\_SIZE];**

**memset(buf, 0, BUF\_SIZE);**

**//ssu\_crontab\_file을 읽기 모드로 오픈**

**if((fp = fopen(fname, "r")) == NULL)**

**{**

**fprintf(stderr, "fopen error for %s\n", fname);**

**exit(1);**

**}**

**//ssu\_crontab\_file에서 읽을 정보가 없을때까지**

**while(!feof(fp))**

**{**

**memset(buf, 0, BUF\_SIZE);**

**//한줄 씩 읽기**

**fgets(buf, BUF\_SIZE, fp);**

**//읽은 길이가 0일 경우 종료**

**if(strlen(buf) == 0) break;**

**//읽은 내용을 인자로 노드를 생성**

**Command \*new\_node = make\_node(buf);**

**//노드를 항상 맨 앞에 연결**

**if(root == NULL)**

**{**

**root = new\_node;**

**}**

**else**

**{**

**new\_node->next = root;**

**root = new\_node;**

**}**

**}**

**fclose(fp);**

**return root;**

**}**