리눅스 시스템 프로그래밍 설계 과제 3: ssu\_crontab, ssu\_rsync

20162481 나반

컴퓨터학부 안승훈

1. 개요

- 리눅스에서 사용자가 동기화를 원하는 파일이나 디렉토리를 동기화하는 프로그램인 ssu\_rsync를 구현한다.

- 리눅스에서 사용자가 주기적으로 실행하고자 하는 명령어를 등록하고 실행시켜주는 ssu\_crontab을 구현한다.

- ssu\_crontab에서 명령어를 실행하기 위한 디몬 프로세스인 ssu\_crond를 구현한다.

- 새로운 명령어의 사용을 통하여 쉘의 원리를 이해하도록 한다.

- 리눅스 시스템에서의 프로그래밍을 통해 각종 시스템 자료구조를 이용하는 능력을 향상할 수 있도록 한다.

2. 설계

**<ssu\_crontab>**

- 사용자가 주기적으로 실행하는 명령어를 “ssu\_crontab\_file”에 저장 및 삭제한다.

- 주기적으로 파일에 저장된 명령어를 수행할 “ssu\_crond” 디몬 프로그램을 포함한다.

- ssu\_crond 프로그램은 운영체제 시작시 함꼐 실행되어 파일에 저장된 명령어를 실행한다.

- 파일에 저장된 명령어가 정상적으로 수행된 경우 ssu\_crontab\_log를 작성한다.

- 명령어의 추가 삭제도 log 파일에 저장할 수 있도록 한다.

- 프롬프트를 통해 add, remove, exit의 명령을 수행 할 수 있도록 한다.

- add의 경우 사용자가 입력한 명령어의 주기가 올바른지 확인하고, 올바른 주기를 가진 명령어만 파일에 추가한다.

- remove의 경우에는 사용자가 입력한 번호의 문장을 삭제한다.

- remove의 경우 번호가 입력되지 않는 경우 에러처리 후 프롬프트로 돌아간다.

- remove 진행 시, 없는 번호가 입력될 경우 오류 출력 후 프롬프트로 돌아간다.

- 프롬프트에 개행 문자 입력시, 프롬프트를 재출력한다.

**<ssu\_crond>**

- ssu\_crontab\_file에 저장된 명령어를 실행할 디몬 프로그램이다.

- ssu\_crontab\_file에 저장된 명령어를 읽어들여 주기를 분석한 후, 주기에 맞추어 프로그램을 수행한다.

- ssu\_crontab\_file이 수정되면 수정사항을 반영하여 실행한다.

- ssu\_crontab\_file의 명령어를 실행하면 ssu\_crontab\_log 로그파일에 로그를 작성한다.

- 운영체제 시작 시 함꼐 실행되어 명령어를 수행한다.

**<ssu\_rsync>**

- 인자로 src 파일 또는 디렉토리, dst 디렉토리를 입력받는다.

- 주어진 src를 dst 디렉토리에 동기화한다.

- src가 파일일 때, src가 dst 디렉토리에 존재하지 않거나, 파일 크기, 수정시간이 다를 경우 같은 파일로 만든다.

- src가 디렉토리일 때, src 디렉토리의 파일을 dst에 동일하게 위치시킨다.

- dst는 디렉토리의 경우만 가능하고, src dst 모두 접근권한이 있을 때에만 프로그램이 작동하도록 한다.

- 동기화중 SIGINT가 발생한 경우 동기화가 취소되고 dst 디렉토리 내 파일들을 동기화하지 않은 상태로 둔다.

- -r 옵션이 설정되면 src의 서브 디렉토리와 그 내부의 파일까지 동기화한다.

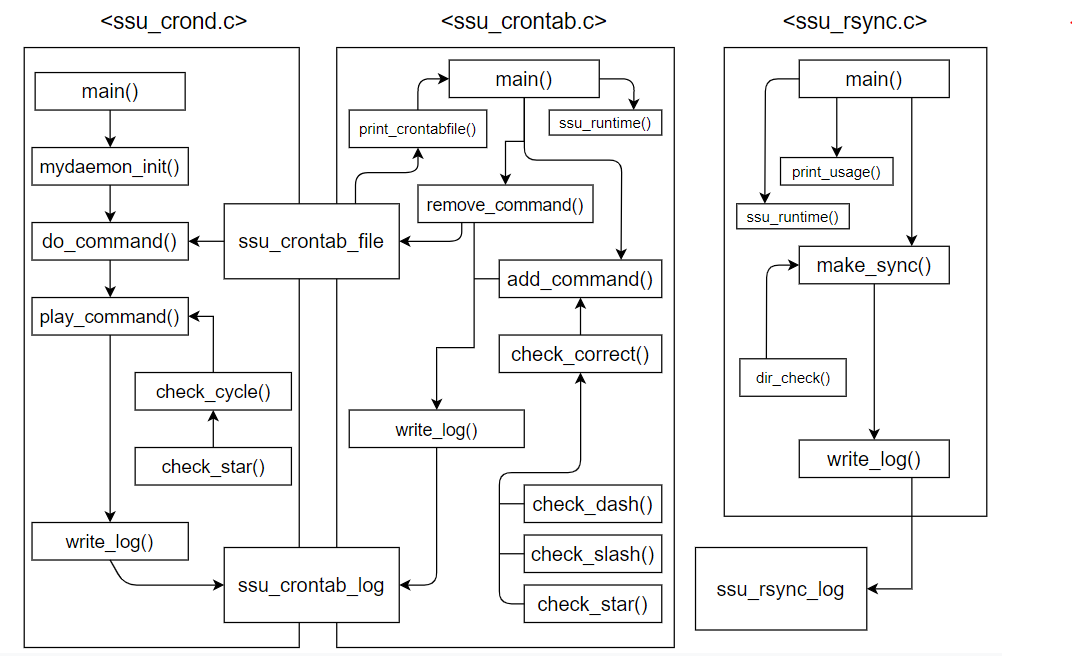
- -t 옵션이 설정되면 tar을 사용하여 src 디렉토리를 묶어서 전송하여 동기화한다.

- -m 옵션이 설정되면 dst 디렉토리에 있는 파일 중, src에 존재하지 않는 경우 삭제한다.

- 동기화가 진행된 명령어, 시간, 파일의 이름과 사이즈를 로그 파일에 작성한다.

- -t, -m 옵션의 경우 별도의 로그 출력 방식을 사용한다.

- 프로그램에서 사용할 함수를 이용하여 순서 구조도를 그려보면 다음과 같다.



3. 구현

- 함수의 프로토타입을 통해, 프로그램에서 사용한 함수를 설명한다.

**<ssu\_crond>**

**int** **mydaemon\_init**(**void**); //디몬 프로세스

**void** **do\_command**(**void**); //정해진 시간에 명령어를 읽는 함수

**void** **play\_command**(**char** buf[]); //읽은 명령어를 실행하는 함수

**void** **check\_cycle**(**char** arr[], **int** num); //주기를 실제시간으로 환산하는 함수

**int** **check\_star**(**char** arr[]); //주기가 \*로만 이루어져있는지 확인하는 함수

**void** **write\_log**(**char** buf[]); //로그를 작성하는 함수

**<ssu\_crontab>**

**void** **ssu\_runtime**(**struct** timeval \*begin\_t, **struct** timeval \*end\_t); //프로그램 수행시간 측정

**void** **print\_crontabfile**(**void**); //ssu\_crontab\_file에 쓰는 함수

**int** **check\_correct**(**char** arr[], **int** index); //주기가 올바른지 확인하는 함수

**void** **add\_command**(**void**); //명령어를 추가하는 함수

**void** **remove\_command**(**void**); //명령어를 삭제하는 함수

**int** **check\_star**(**char** arr[]); //주기가 \*인지 확인하는 함수

**int** **check\_slash**(**char** arr[]); //주기에 /가 포함되어있는지 확인하는 함수

**int** **check\_dash**(**char** arr[]); //주기에 -가 포함되어있는지 확인하는 함수

**void** **write\_log**(**char** arr[], **char** sentence[]); //log를 작성하는 함수

**<ssu\_rsync>**

**void** **print\_usage**(**char** \*str); //사용법 출력

**void** **make\_sync**(**void**); //동기화를 진행하는 함수

**int** **dir\_check**(**const** **struct** dirent \*info); //scandir 필터함수

**void** **write\_log**(**char** \*fname, **struct** stat \*statbuf); //로그 작성 함수

**void** **ssu\_runtime**(**struct** timeval \*begin\_t, **struct** timeval \*end\_t); //프로그램 수행 시간 측정

**<ssu\_crond>**

\* int mydaemon\_init(void);

**int** **mydaemon\_init**(**void**) {

**pid\_t** pid;

**int** fd, maxfd;

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

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

exit(**1**);

}

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

exit(**0**);

pid = getpid();

setsid();

signal(SIGTTIN, SIG\_IGN);

signal(SIGTTOU, SIG\_IGN);

signal(SIGTSTP, SIG\_IGN);

maxfd = getdtablesize();

**for**(fd = **0**; fd < maxfd; fd++)

close(fd);

umask(**0**);

//chdir("/");

fd = open("/dev/null", O\_RDWR);

dup(**0**);

dup(**0**);

**if**((filefp = fopen("ssu\_crontab\_file", "r")) == NULL) {

fprintf(stderr, "fopen error for command file**\n**");

exit(**1**);

} //명령어가 들어있는 파일을 오픈

**if**((logfp = fopen("ssu\_crontab\_log", "a+")) == NULL) {

fprintf(stderr, "fopen error for logfile**\n**");

} //로그 파일을 오픈

setbuf(logfp, NULL);

**time\_t** prevtime, curtime;

**struct** tm t1, t2;

prevtime = time(NULL);

**while**(**1**) { //과거에 저장해둔 시간과 현재 시간을 비교하여 분이 바뀌면 실행

localtime\_r(&prevtime, &t1);

curtime = time(NULL);

localtime\_r(&curtime, &t2);

**if**(t1.tm\_min + **1** == t2.tm\_min) {

prevtime = curtime;

do\_command(); //명령어를 실행 하는 함수 실행

}

sleep(**1**);

}

**return** **0**;

}

- 디몬 프로세스를 생성하는 함수이다.

- fork를 통해 새 프로세스를 생성 후, 부모 프로세스를 중단하여 디몬 프로세스가 된다.

- 전에 측정한 시각과 현재 시각을 비교하여, 분 단위가 변경된 경우 명령문을 분석하는 함수를 호출한다.

- 그 외의 시각에는 대기한다.

\* void do\_command(void);

**void** **do\_command**(**void**) {

**char** sentence[BUFSIZE];

rewind(filefp); //파일 포인터를 파일의 가장 처음으로 돌림

**while**(fgets(sentence, BUFSIZE, filefp) != NULL) { //한문장 읽어옴

play\_command(sentence);

} //파일의 끝까지 한줄씩 읽어서, 한줄씩 실행여부를 판단함

**return**;

}

- ssu\_crontab\_file에서 명령문을 한 줄씩 읽어서 명령문을 실행하는 함수를 호출하는 함수이다.

- 파일의 첫 줄부터 마지막 줄까지 읽어들인다.

\* void play\_command(char buf[]);

**void** **play\_command**(**char** buf[]) {

**char** \*token;

**char** tmpbuf[BUFSIZE];

**char** command[BUFSIZE];

**struct** tm t;

**time\_t** now;

memset(s\_min, **0**, **sizeof**(s\_min));

memset(s\_hour, **0**, **sizeof**(s\_hour));

memset(s\_day, **0**, **sizeof**(s\_day));

memset(s\_month, **0**, **sizeof**(s\_month));

memset(s\_wday, **0**, **sizeof**(s\_wday));

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

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

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

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

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

memset(tmpbuf, **0**, BUFSIZE);

//배열 초기화

strcpy(tmpbuf, buf);

token = strtok(tmpbuf, " ");

strcpy(s\_min, token);

token = strtok(NULL, " ");

strcpy(s\_hour, token);

token = strtok(NULL, " ");

strcpy(s\_day, token);

token = strtok(NULL, " ");

strcpy(s\_month, token);

token = strtok(NULL, " ");

strcpy(s\_wday, token);

token = strtok(NULL, "**\n**");

strcpy(command, token);

//주기를 각각 나누어서 저장함

check\_cycle(s\_min, **1**);

check\_cycle(s\_hour, **2**);

check\_cycle(s\_day, **3**);

check\_cycle(s\_month, **4**);

check\_cycle(s\_wday, **5**);

//나눠진 주기에 맞추어 실행가능한 시간을 정리함

now = time(NULL);

localtime\_r(&now, &t);

**if**(min[t.tm\_min] && hour[t.tm\_hour] && day[t.tm\_mday] && month[t.tm\_mon+**1**] && wday[t.tm\_wday]) {

system(command);

write\_log(buf);

} //현재 시간이 실행 가능 상황에 적합할때, system을 통해 명령어를 실행하고 로그를 작성함

//시간이 모두 1이면 실행

**return**;

}

- 명령어를 분해하여 각 주기를 계산하고 명령문을 실행하는 함수이다.

- 분, 시, 일, 월, 요일을 각각 주기로 환산한 다음, 현재 시각에 실행해도 되는지를 판단하여, 명령문을 실행한다.

\* void check\_cycle(char arr[], int num);

**void** **check\_cycle**(**char** arr[], **int** num) { //주기를 실제 가능 시간으로 환산하는 함수

**int** slash\_loc = -**1**;

**int** dash\_loc = -**1**;

**int** term;

**char** \*ptr;

**char** buf[**20**][**20**];

**char** tmp[**100**];

**int** i = **0**;

**int** cnt = **0**;

**int** checksum = **0**;

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

strcpy(tmp, arr);

ptr = strtok(tmp, ",");

strcpy(buf[i++], ptr);

cnt++;

**while**((ptr = strtok(NULL, ","))!= NULL) {

strcpy(buf[i++], ptr);

cnt++;

} //쉼표를 기준으로 주기를 분해함

**for**(i = **0**; i < cnt; i++) { //분해한 주기를 하나씩 시간으로 바꿈

dash\_loc = -**1**;

slash\_loc = -**1**;

checksum = **0**;

**if**(num == **1**) { //분

**if**(check\_star(buf[i])) {

memset(min, **1**, **sizeof**(min));

**return**;

} //\*만 있으면 모든 경우에 실행

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

term = atoi(buf[i]+**2**);

**for**(**int** i = **0**; i < **sizeof**(min); i++) {

**if**((i % term) == (term - **1**))

min[i] = **1**;

}

**return**;

} //\*/로 시작하면 뒤에 숫자 간격만큼 건너서 실행

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-' || buf[i][x] == '/')

checksum = **1**;

} //숫자 단독인지 확인

**if**(!checksum) {

**int** tmp = atoi(buf[i]);

min[tmp] = **1**;

**return**;

} //그 숫자 ON

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-') {

dash\_loc = x;

**break**;

}

}//dash가 있는 경우

checksum = **0**;

**if**(dash\_loc != -**1**) { //dash 가 있을때

**int** start = atoi(buf[i]);

**int** end = atoi(buf[i] + dash\_loc + **1**);

**for**(**int** k = **0**; k < strlen(buf[i]); k++) {

**if**(buf[i][k] == '/') {

checksum = **1**;

slash\_loc = k;

**break**;

}

}

**if**(checksum) { // slash가 있을떄

**int** div = atoi(buf[i]+slash\_loc + **1**);

**int** hop = **0**;

**for**(**int** s = start; s <= end; s++) {

hop++;

**if**(hop % div == **0**)

min[s] = **1**;

}

}

**else** { //slash가 뒤에 없을떄

**for**(**int** s = start; s<= end; s++)

min[s] = **1**;

}

}

}

**else** **if** (num == **2**) { //시간

**if**(check\_star(buf[i])) {

memset(hour, **1**, **sizeof**(hour));

**return**;

} //\*만 있을떄

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

term = atoi(buf[i]+**2**);

**for**(**int** i = **0**; i < **sizeof**(hour); i++) {

**if**((i % term) == (term - **1**))

hour[i] = **1**;

}

**return**;

} //\*/일때

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-' || buf[i][x] == '/')

checksum = **1**;

} //쉼표로 뽀갠것이 숫자일떄

**if**(!checksum) {

**int** tmp = atoi(buf[i]);

hour[tmp] = **1**;

**return**;

} //그 숫자 ON

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-') {

dash\_loc = x;

**break**;

}

}

checksum = **0**;

**if**(dash\_loc != -**1**) { //dash 가 있을때

**int** start = atoi(buf[i]);

**int** end = atoi(buf[i] + dash\_loc + **1**);

**for**(**int** k = **0**; k < strlen(buf[i]); k++) {

**if**(buf[i][k] == '/') {

checksum = **1**;

slash\_loc = k;

**break**;

}

}

**if**(checksum) { // slash가 있을떄

**int** div = atoi(buf[i]+slash\_loc + **1**);

**int** hop = **0**;

**for**(**int** s = start; s <= end; s++) {

hop++;

**if**(hop % div == **0**)

hour[s] = **1**;

}

**return**;

}

**else** { //slash가 뒤에 없을떄

**for**(**int** s = start; s<= end; s++)

hour[s] = **1**;

**return**;

}

}

}

**else** **if** (num == **3**) { //날짜

**if**(check\_star(buf[i])) {

memset(day, **1**, **sizeof**(day));

**return**;

} //\*만 있을떄

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

term = atoi(buf[i]+**2**);

**for**(**int** i = **0**; i < **sizeof**(day); i++) {

**if**((i % term) == (term - **1**))

day[i] = **1**;

}

**return**;

} //'\*/'이 있을떄

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-' || buf[i][x] == '/')

checksum = **1**;

} //쉼표로 뽀갠것이 숫자일떄

**if**(!checksum) {

**int** tmp = atoi(buf[i]);

day[tmp] = **1**;

**return**;

} //그 숫자 ON

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-') {

dash\_loc = x;

**break**;

}

}

checksum = **0**;

**if**(dash\_loc != -**1**) { //dash 가 있을때

**int** start = atoi(buf[i]);

**int** end = atoi(buf[i] + dash\_loc + **1**);

**for**(**int** k = **0**; k < strlen(buf[i]); k++) {

**if**(buf[i][k] == '/') {

checksum = **1**;

slash\_loc = k;

**break**;

}

}

**if**(checksum) { // slash가 있을떄

**int** div = atoi(buf[i]+slash\_loc + **1**);

**int** hop = **0**;

**for**(**int** s = start; s <= end; s++) {

hop++;

**if**(hop % div == **0**)

day[s] = **1**;

}

**return**;

}

**else** { //slash가 뒤에 없을떄

**for**(**int** s = start; s<= end; s++)

day[s] = **1**;

**return**;

}

}

}

**else** **if**(num == **4**) { //달

**if**(check\_star(buf[i])) {

memset(month, **1**, **sizeof**(month));

**return**;

}

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

term = atoi(buf[i]+**2**);

**for**(**int** i = **1**; i < **sizeof**(month); i++) {

**if**((i % term) == (term - **1**))

month[i] = **1**;

}

**return**;

}// 문장에 '\*/'이 있을때

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-' || buf[i][x] == '/')

checksum = **1**;

} //쉼표로 뽀갠것이 숫자일떄

**if**(!checksum) {

**int** tmp = atoi(buf[i]);

month[tmp] = **1**;

**return**;

} //그 숫자 ON

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-') {

dash\_loc = x;

**break**;

}

}

checksum = **0**;

**if**(dash\_loc != -**1**) { //dash 가 있을때

**int** start = atoi(buf[i]);

**int** end = atoi(buf[i] + dash\_loc + **1**);

**for**(**int** k = **0**; k < strlen(buf[i]); k++) {

**if**(buf[i][k] == '/') {

checksum = **1**;

slash\_loc = k;

**break**;

}

}

**if**(checksum) { // slash가 있을떄

**int** div = atoi(buf[i]+slash\_loc + **1**);

**int** hop = **0**;

**for**(**int** s = start; s <= end; s++) {

hop++;

**if**(hop % div == **0**)

month[s] = **1**;

}

**return**;

}

**else** { //slash가 뒤에 없을떄

**for**(**int** s = start; s<= end; s++)

month[s] = **1**;

**return**;

}

}

}

**else** **if** (num == **5**) { //요일

**if**(check\_star(buf[i])) {

memset(wday, **1**, **sizeof**(wday));

**return**;

}

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

term = atoi(buf[i]+**2**);

**for**(**int** i = **0**; i < **sizeof**(wday); i++) {

**if**((i % term) == (term - **1**))

wday[i] = **1**;

}

**return**;

}

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-' || buf[i][x] == '/')

checksum = **1**;

} //쉼표로 뽀갠것이 숫자일떄

**if**(!checksum) {

**int** tmp = atoi(buf[i]);

wday[tmp] = **1**;

**return**;

} //그 숫자 ON

**for**(**int** x = **0**; x < strlen(buf[i]); x++) {

**if**(buf[i][x] == '-') {

dash\_loc = x;

**break**;

}

}

checksum = **0**;

**if**(dash\_loc != -**1**) { //dash 가 있을때

**int** start = atoi(buf[i]);

**int** end = atoi(buf[i] + dash\_loc + **1**);

**for**(**int** k = **0**; k < strlen(buf[i]); k++) {

**if**(buf[i][k] == '/') {

checksum = **1**;

slash\_loc = k;

**break**;

}

}

**if**(checksum) { // slash가 있을떄

**int** div = atoi(buf[i]+slash\_loc + **1**);

**int** hop = **0**;

**for**(**int** s = start; s <= end; s++) {

hop++;

**if**(hop % div == **0**)

wday[s] = **1**;

}

**return**;

}

**else** { //slash가 뒤에 없을떄

**for**(**int** s = start; s<= end; s++)

wday[s] = **1**;

**return**;

}

}

}

}

**return**;

}

- 주기를 읽어들여 각각의 시각으로 환산하는 함수이다.

- 분해한 주기를 숫자로 바꾸어, 특수문자를 보고 범위 또는 실행할 시각을 저장한다.

- 배열을 이용하여 매 분, 시, 일, 월, 요일 마다 실행할 시각을 저장한다.

- 저장된 시각과 현재 시각이 일치 하는 경우 명령어를 system함수를 통해 실행한다.

\* int check\_star(char arr[]);

**int** **check\_star**(**char** arr[]) { //주기가 \*뿐인지 확인하는 함수

**if**(!strcmp(arr, "\*"))

**return** **1**;

**else**

**return** **0**;

}

- 주어진 문자열이 \*인지 확인하는 함수이다.

\* void write\_log(char buf[]);

**void** **write\_log**(**char** buf[]) { //로그를 작성하는 함수

**time\_t** curtime;

**char** tmp[BUFSIZE];

memset(tmp, **0**, BUFSIZE);

curtime = time(NULL);

sprintf(tmp, "[%s", ctime(&curtime));

tmp[strlen(tmp) - **1**] = ']';

strcat(tmp, " run ");

strcat(tmp, buf);

fputs(tmp, logfp);

//현재 시간과 명령어를 로그에 출력함

**return**;

}

- 명령문이 작동하였을 때, 그 실행을 로그 파일에 작성하기 위한 함수이다.

**<ssu\_crontab>**

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

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

{ //프로그램의 수행시간 측정

end\_t->tv\_sec -= begin\_t->tv\_sec; //끝난 시간에서 처음시간을 뺌

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

end\_t->tv\_sec--;

end\_t->tv\_usec += **1000000**;

} //나중의 us값이 작을경우 1초를 뺴서 us값이 더해줌

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

printf("Runtime: %ld:%02ld:%06ld(min:sec:usec)**\n**",

end\_t->tv\_sec / **60**, end\_t->tv\_sec % **60**, end\_t->tv\_usec);

} //프로그램 수행시간을 분, 초, 마이크로초로 출력

- 프로그램의 수행 시간을 측정하는 함수이다.

\* void print\_crontabfile(void);

**void** **print\_crontabfile**(**void**) { //입력받은 명령어를 파일에 쓰는 함수

**char** tmp[BUFSIZE];

memset(tmp, **0**, BUFSIZE);

**int** num = **0**;

fseek(filefp, **0**, SEEK\_SET);

**while**(fgets(tmp, BUFSIZE, filefp) != NULL) {

printf("%d. ", num);

fputs(tmp, stdout);

num++;

}

printf("**\n**");

crontabnum = num; //crontabfile에 있는 갯수를 저장

**return**;

}

- ssu\_crontab\_file을 프롬프트 출력 전에 실행하는 함수이다.

\* int check\_correct(char arr[], int index);

**int** **check\_correct**(**char** arr[], **int** index) { //주기가 올바른지 확인하는 함수

**int** checksum = **0**;

**int** slash\_loc = -**1**;

**int** dash\_loc = -**1**;

**int** length;

**char** buf[**20**][**20**];

**char** tmp[**100**];

**char** \*ptr, \*sav;

**int** i = **0**;

**int** cnt = **0**;

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

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

strcpy(tmp, arr);

ptr = strtok\_r(tmp, ",", &sav);

strcpy(buf[i++], ptr);

cnt++;

//쉼표 단위로 주기를 분해함

**while**((ptr = strtok\_r(NULL, ",", &sav)) != NULL) {

strcpy(buf[i++], ptr);

cnt++;

} //쉼표를 기준으로 토큰으로 분해

**for**(i = **0**; i < cnt; i++) { //분해한 것 각각을 분석함

dash\_loc = -**1**;

slash\_loc = -**1**;

checksum = **0**;

**for**(**int** f = **0**; f < strlen(buf[i]) - **1**; f++) {

**if**((buf[i][f] == '/' || buf[i][f] == '-') && (buf[i][f+**1**] == '-' || buf[i][f+**1**] == '/'))

**return** **1**;

} //'-''/'기호가 2개이상 나올 경우 무조건 불가능한 주기

**if**(index == **1**) { // 분 확인

**if**(check\_star(buf[i])) **return** **0**; // \*만 있으면 ok

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

checksum = check\_slash(buf[i] + **2**);

checksum = check\_dash(buf[i] + **2**);

**if**(checksum) **return** **1**;

**int** temp = atoi(buf[i]+**2**);

**if**(temp < **0** || temp > **60**) **return** **1**;

} // '\*/'로 시작하는 경우 뒤에 숫자만 올 수 있음

**if**(check\_dash(buf[i])) { //문장에 '-'가 있을때

**int** loc = check\_dash(buf[i]);

**int** tmp1 = atoi(buf[i]);

**int** tmp2 = atoi(buf[i] + loc + **1**);

**if**(tmp1 >= tmp2) **return** **1**;

**if**(tmp1 < **0** || tmp2 > **60**) **return** **1**;

loc = check\_slash(buf[i]+ loc + **1**);

**if**(loc) { //문장에 '/'도 포함되어있는 경우

**int** tmp3 = atoi(buf[i] + loc + **1**);

**if**(tmp3 > tmp2 - tmp1) **return** **1**;

}

}

**else** **if**(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

**int** tmp1 = atoi(buf[i]);

**if**(tmp1 < **0** || tmp1 > **60**) **return** **1**;

} //문장에 - / 가 없는 경우 숫자만 판단함

**return** **0**;

}

**else** **if** (index == **2**) { //시간 확인

**if**(check\_star(buf[i])) **return** **0**; // \*만 있으면 ok

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

checksum = check\_slash(buf[i] + **2**);

checksum = check\_dash(buf[i] + **2**);

**if**(checksum) **return** **1**;

**int** temp = atoi(buf[i]+**2**);

**if**(temp < **0** || temp > **23**) **return** **1**;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올수 있음

**if**(check\_dash(buf[i])) { //문장에 -가 있을때

**int** loc = check\_dash(buf[i]);

**int** tmp1 = atoi(buf[i]);

**int** tmp2 = atoi(buf[i] + loc + **1**);

**if**(tmp1 >= tmp2) **return** **1**;

**if**(tmp1 < **0** || tmp2 > **23**) **return** **1**;

loc = check\_slash(buf[i]+ loc + **1**);

**if**(loc) { //같은 문장에 /도 포함되어있을때

**int** tmp3 = atoi(buf[i] + loc + **1**);

**if**(tmp3 > tmp2 - tmp1) **return** **1**;

}

}

**else** **if**(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

**int** tmp1 = atoi(buf[i]);

**if**(tmp1 < **0** || tmp1 > **23**) **return** **1**;

} //문장에 -또는 /가 없을 때

**return** **0**;

}

**else** **if**(index == **3**) { //날짜 확인

**if**(check\_star(buf[i])) **return** **0**; // \*만 있으면 ok

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

checksum = check\_slash(buf[i] + **2**);

checksum = check\_dash(buf[i] + **2**);

**if**(checksum) **return** **1**;

**int** temp = atoi(buf[i]+**2**);

**if**(temp < **1** || temp > **31**) **return** **1**;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올 수 있음

**if**(check\_dash(buf[i])) { //문장에 -가 있는 경우

**int** loc = check\_dash(buf[i]);

**int** tmp1 = atoi(buf[i]);

**int** tmp2 = atoi(buf[i] + loc + **1**);

**if**(tmp1 >= tmp2) **return** **1**;

**if**(tmp1 < **1** || tmp2 > **31**) **return** **1**;

loc = check\_slash(buf[i]+ loc + **1**);

**if**(loc) { //같은 문장에 /도 포함되어있는 경우

**int** tmp3 = atoi(buf[i] + loc + **1**);

**if**(tmp3 > tmp2 - tmp1) **return** **1**;

}

}

**else** **if**(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

**int** tmp1 = atoi(buf[i]);

**if**(tmp1 < **1** || tmp1 > **31**) **return** **1**;

} //문장에 -또는 /기호가 없을 경우

**return** **0**;

}

**else** **if**(index == **4**) { //월 확인

**if**(check\_star(buf[i])) **return** **0**; // \*만 있으면 ok

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

checksum = check\_slash(buf[i] + **2**);

checksum = check\_dash(buf[i] + **2**);

**if**(checksum) **return** **1**;

**int** temp = atoi(buf[i]+**2**);

**if**(temp < **1** || temp > **12**) **return** **1**;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올 수 있음

**if**(check\_dash(buf[i])) { //문장에 -가 포함되어있는 경우

**int** loc = check\_dash(buf[i]);

**int** tmp1 = atoi(buf[i]);

**int** tmp2 = atoi(buf[i] + loc + **1**);

**if**(tmp1 >= tmp2) **return** **1**;

**if**(tmp1 < **1** || tmp2 > **12**) **return** **1**;

loc = check\_slash(buf[i]+ loc + **1**);

**if**(loc) { //문장에 /도 포함되어 있는 경우

**int** tmp3 = atoi(buf[i] + loc + **1**);

**if**(tmp3 > tmp2 - tmp1) **return** **1**;

}

}

**else** **if**(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

**int** tmp1 = atoi(buf[i]);

**if**(tmp1 < **1** || tmp1 > **12**) **return** **1**;

} //문장에 -또는 /가 포함되어 있지 않은 경우 숫자만 판단함

**return** **0**;

}

**else** **if**(index == **5**) { //요일 확인

**if**(check\_star(buf[i])) **return** **0**; // \*만 있으면 ok

**if**(buf[i][**0**] == '\*' && buf[i][**1**] == '/') {

checksum = check\_slash(buf[i] + **2**);

checksum = check\_dash(buf[i] + **2**);

**if**(checksum) **return** **1**;

**int** temp = atoi(buf[i]+**2**);

**if**(temp < **0** || temp > **6**) **return** **1**;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올 수 있음

**if**(check\_dash(buf[i])) { //문장에 -가 포함되어 있을때

**int** loc = check\_dash(buf[i]);

**int** tmp1 = atoi(buf[i]);

**int** tmp2 = atoi(buf[i] + loc + **1**);

**if**(tmp1 >= tmp2) **return** **1**;

**if**(tmp1 < **0** || tmp2 > **6**) **return** **1**;

loc = check\_slash(buf[i]+ loc + **1**);

**if**(loc) { //문장에 /가 포함되어 있는 경우

**int** tmp3 = atoi(buf[i] + loc + **1**);

**if**(tmp3 > tmp2 - tmp1) **return** **1**;

}

}

**else** **if**(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

**int** tmp1 = atoi(buf[i]);

**if**(tmp1 < **0** || tmp1 > **6**) **return** **1**;

} //문장에 - 또는 /가 포함되지 않은 경우

**return** **0**;

}

}

}

- add 명령 수행 시, 입력된 명령문의 주기가 올바른지 확인하는 함수이다.

- 문자열을 , 를 토큰으로 하여 분해한 후, 각각의 문장이 적절한지 확인하는 함수이다.

\* void add\_command(void);

**void** **add\_command**(**void**) { //add를 실행하는 함수

**char** \*token;

**char** \*tmptmp;

**char** min[**20**];

**char** hour[**20**];

**char** day[**20**];

**char** month[**20**];

**char** wday[**20**];

**int** checksum;

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

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

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

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

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

//배열 초기화

checksum = **0**;

**for**(**int** i = **0**; i < strlen(input); i++)

**if**(input[i] == ' ' || input[i] == '\n')

checksum++;

**if**(checksum < **7**) {

printf("Input error!**\n**");

**return**;

}

token = strtok\_r(input, " ", &tmptmp); //명령어 컷

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(min, token);

**if**(check\_correct(min, **1**)) {

printf("minute input error!**\n**");

**return**;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(hour, token);

**if**(check\_correct(hour, **2**)) {

printf("hour input error!**\n**");

**return**;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(day, token);

**if**(check\_correct(day, **3**)) {

printf("day input error!**\n**");

**return**;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(month, token);

**if**(check\_correct(month, **4**)) {

printf("month input error!**\n**");

**return**;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(wday, token);

**if**(check\_correct(wday, **5**)) {

printf("weekday input error!**\n**");

**return**;

}

//입력된 문장을 각각의 주기에 맞게 분해하여 각 주기가 올바른지 확인함

fseek(filefp, **0**, SEEK\_END);

fwrite(command, strlen(command), **1**, filefp); //file에 새로 명령어를 씀

write\_log("add", command); //로그 작성

**return**;

}

- add 명령 시 호출되는 함수이다.

- 주기가 올바른지 확인하는 함수인 check\_correct를 호출하여, 각 분, 시, 일, 월, 요일 주기를 확인한다.

\* void remove\_command(void);

**void** **remove\_command**(**void**) { //remove를 수행하는 함수

**int** i = **0**;

**int** num = atoi(command);

**int** del\_loc = **0**; //삭제할 문장의 시작 위치

**int** next\_loc = **0**; //삭제할 문장 다음줄의 시작 위치

**char** tmp[BUFSIZE];

**char** save[BUFSIZE];

**char** del\_command[BUFSIZE];

**int** filesize;

**if**(command[**0**] == '\n' || command[**0**] == '\0') {

printf("No number!!**\n**");

**return**;

}

**if**(num >= crontabnum || num < **0**) {

printf("number error!!!**\n**");

**return**;

}

fseek(filefp, **0**, SEEK\_END);

filesize = ftell(filefp); //원래의 파일 크기 구하기

fseek(filefp, **0**, SEEK\_SET);

**for**(i = **0**; i < num; i++) {

memset(tmp, **0**, BUFSIZE);

fgets(tmp, BUFSIZE, filefp);

del\_loc += strlen(tmp);

}

memset(tmp, **0**, BUFSIZE);

memset(del\_command, **0**, BUFSIZE);

fgets(del\_command, BUFSIZE, filefp);

next\_loc = ftell(filefp);

//삭제할 문장의 시작점과 그 다음 문장의 시작점을 저장함

memset(save, **0**, BUFSIZE);

**if**(fread(save, BUFSIZE, **1**,filefp) < **0**) {

fprintf(stderr, "fread error!**\n**");

**return**;

} //삭제할 문장의 시작점부터 그 다음문장을 덮어씀

fseek(filefp, del\_loc, SEEK\_SET);

fwrite(save, BUFSIZE, **1**, filefp);

truncate("ssu\_crontab\_file", filesize - strlen(del\_command));

//파일에서 삭제한만큼 파일의 크기를 줄임

write\_log("remove", del\_command); //로그 작성

**return**;

}

- remove 명령시 호출되는 함수이다.

- 파일에서 삭제할 위치를 찾아 삭제할 다음 문장부터 읽어 덮어쓴다.

- 이후, 로그를 작성하는 함수를 호출하여, 로그에 변동사항을 저장한다.

\* int check\_star(char arr[]);

**int** **check\_star**(**char** arr[]) { //문장에 \*만 포함되어 있는지 확인하는 함수

**if**(!strcmp(arr, "\*"))

**return** **1**;

**else**

**return** **0**;

}

- 주어진 문자열이 \* 인지 확인하는 함수이다.

\* int check\_slash(char arr[]);

**int** **check\_slash**(**char** arr[]) { //문장에 '/' 가 포함되었는지 확인하는 함수

**int** i;

**for**(i = **0**; i <strlen(arr); i++) {

**if**(\*(arr + i) == '/')

**return** i;

}

**return** **0**;

}

- 주어진 문자열에 ‘/’ 가 포함되어 있는지 확인하는 함수이다.

\* int check\_dash(char arr[]);

**int** **check\_dash**(**char** arr[]) { //문장에 '-' 기 포함되었는지 확인하는 함수

**int** i;

**for**(i = **0**; i < strlen(arr); i++) {

**if**(\*(arr + i) == '-')

**return** i;

}

**return** **0**;

}

- 주어진 문자열에 ‘-‘ 가 포함되어 있는지 확인하는 함수이다.

\* void write\_log(char arr[], char sentence[]);

**void** **write\_log**(**char** arr[], **char** sentence[]) { //명령어를 로그에 작성하는 함수

**time\_t** curtime;

**char** timebuf[BUFSIZE];

memset(timebuf, **0**, BUFSIZE);

curtime = time(NULL);

sprintf(timebuf, "[");

sprintf(timebuf+**1**, "%s", ctime(&curtime));

timebuf[strlen(timebuf)-**1**] = ']';

strcat(timebuf, " ");

strcat(timebuf, arr);

strcat(timebuf, " ");

strcat(timebuf, sentence);

fwrite(timebuf, strlen(timebuf), **1**, logfp);

**return**;

}

- add remove 가 발생할 때, 로그 파일에 발생한 내용을 기록하는 함수이다.

**<ssu\_rsync>**

\* void print\_usage(char \*str);

**void** **print\_usage**(**char** \*str) {

printf("Usage : %s <option> <src file or directory> <dst directory>**\n**", str);

printf(" <option>**\n**");

printf(" -r : Make sync <src>'s sub directory**\n**");

printf(" -t : Make sync with .tar file**\n**");

printf(" -m : Delete <dst>'s files that not exist in <src>**\n**");

**return**;

} //사용법 출력 함수

- 프로그램의 사용법을 출력하는 함수이다.

\* void make\_sync(void);

**void** **make\_sync**(**void**) { //동기화 하는 함수

**struct** dirent \*\*namelist\_src; //src가 디렉토리일때 내부 파일 목록

**struct** dirent \*\*namelist\_dst; //dst의 파일 목록

**int** src\_count, dst\_count; //src dst의 파일 갯수

**struct** stat statbuf;

**struct** stat src\_statbuf;

**struct** utimbuf timebuf;

**int** index\_src;

**char** src\_fname[FILEMAX]; //src가 파일일 때 파일 이름을 저장함

**char** src\_check[PATHMAX]; //src의 파일 이름을 절대 경로로 바꿔서 저장함

**char** dst\_check[PATHMAX]; //dst의 파일 이름을 절대 경로로 바꿔서 저장함

**char** orifile\_rename[PATHMAX]; //dst에 있는 이름이 같은 파일을 이름을 바꿔서 저장

**int** loc, readcnt;

**int** fd\_sync, fd\_src;

**char** buf[PATHMAX]; //read\_write용 버퍼

**int** checksum = **0**;

memset(src\_fname, **0**, FILEMAX);

**for**(**int** i = **0**; i < strlen(src); i++)

**if**(src[i] == '/')

loc = i;

strcpy(src\_fname, src+loc+**1**); //src가 파일 이름일때 파일 이름만 추출

**if**(lstat(src, &statbuf) < **0**) {

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

exit(**1**);

}

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

index\_src = **1**;

**else**

index\_src = **0**;

//index\_src 가 1이면 src가 디렉토리, 0이면 파일

**if**(index\_src) { //src가 디렉토리 일때

**if**((src\_count = scandir(src, &namelist\_src, dir\_check, alphasort)) == -**1**) {

**return**;

}

**if**((dst\_count = scandir(dst, &namelist\_dst, dir\_check, alphasort)) == -**1**) {

**return**;

}

**for**(**int** i = **0**; i < src\_count; i++) { //src 디렉토리 내의 파일 갯수만큼

checksum = **0**;

memset(src\_check, **0**, PATHMAX);

strcpy(src\_check, src);

strcat(src\_check, "/");

strcat(src\_check, namelist\_src[i]->d\_name);

**if**(lstat(src\_check, &src\_statbuf) < **0**) {

fprintf(stderr, "lstat error2**\n**");

**return**;

}

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

**continue**;

//디렉토리면 패스

**for**(**int** j = **0**; j < dst\_count; j++) { //dst 디렉토리 순회

memset(dst\_check, **0**, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, namelist\_dst[j]->d\_name);

//dst의 파일을 절대경로로 만듬

//printf("%s\n%s\n\n", src\_check, dst\_check);

**if**(lstat(dst\_check, &statbuf) < **0**) {

fprintf(stderr, "lstat error3**\n**");

exit(**1**);

}

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

**continue**;

//디렉토리면 패스

**if**(!strcmp(namelist\_src[i]->d\_name, namelist\_dst[j]->d\_name)){

checksum = **1**;

**if**(src\_statbuf.st\_size == statbuf.st\_size && src\_statbuf.st\_mtime == statbuf.st\_mtime)

**continue**;

//파일 크기가 같고 최종 수정시간이 같으면 패스

memset(orifile\_rename, **0**, PATHMAX);

strcpy(orifile\_rename, dst);

strcat(orifile\_rename, "/\_");

strcat(orifile\_rename, namelist\_dst[j]->d\_name);

//rename할꺼 만듬

rename(dst\_check, orifile\_rename);

**if**((fd\_sync = creat(dst\_check, **0644**)) < **0**) {

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

**return**;

}

**if**((fd\_src = open(src\_check, O\_RDONLY)) < **0**) {

fprintf(stderr, "open error for src");

**return**;

}

memset(buf, **0**, PATHMAX);

**while**((readcnt = read(fd\_src, buf, PATHMAX)) > **0**) {

write(fd\_sync, buf, readcnt);

memset(buf, **0**, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

**if**(utime(dst\_check, &timebuf) < **0**) {

fprintf(stderr, "utime error for dst\_check**\n**");

**return**;

}

unlink(orifile\_rename);

write\_log(namelist\_src[i]->d\_name, &src\_statbuf);

**break**;

}

}

**if**(checksum == **0**) { //dst에 파일이 없는 경우

memset(dst\_check, **0**, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, namelist\_src[i]->d\_name);

//dst 디렉토리에 만들 파일 경로

**if**((fd\_sync = creat(dst\_check, **0644**)) < **0**) {

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

**return**;

}

**if**((fd\_src = open(src\_check, O\_RDONLY)) < **0**) {

fprintf(stderr, "open error for src");

**return**;

}

memset(buf, **0**, PATHMAX);

**while**((readcnt = read(fd\_src, buf, PATHMAX)) > **0**) {

write(fd\_sync, buf, readcnt);

memset(buf, **0**, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

**if**(utime(dst\_check, &timebuf) < **0**) {

fprintf(stderr, "utime error for dst\_check**\n**");

**return**;

}

write\_log(namelist\_src[i]->d\_name, &src\_statbuf);

}

}

}

**else** { //src가 파일일때

**if**(lstat(src, &src\_statbuf) < **0**) {

fprintf(stderr, "lstat error for src");

**return**;

}

**if**((dst\_count = scandir(dst, &namelist\_dst, dir\_check, alphasort)) == -**1**) {

**return**;

}

**for**(**int** i = **0**; i < dst\_count; i++) { //dst 디렉토리 순회

memset(dst\_check, **0**, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, namelist\_dst[i]->d\_name);

//dst 파일의 절대경로 생성

**if**(lstat(dst\_check, &statbuf) < **0**) {

fprintf(stderr, "lstat error1**\n**");

**return**;

}

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

**continue**;

//디렉토리일 경우 패스함

**if**(!strcmp(src\_fname, namelist\_dst[i]->d\_name)) { //파일 이름이 같은게 있는 경우

checksum = **1**;

**if**(src\_statbuf.st\_size == statbuf.st\_size && src\_statbuf.st\_mtime == statbuf.st\_mtime)

**continue**;

//파일 크기가 같고 최종 수정시간이 같으면 패스

memset(orifile\_rename, **0**, PATHMAX);

strcpy(orifile\_rename, dst);

strcat(orifile\_rename, "/\_");

strcat(orifile\_rename, namelist\_dst[i]->d\_name);

rename(dst\_check, orifile\_rename); //dst에 원래 있던 파일을 \_(파일) 로 만듦

**if**((fd\_sync = creat(dst\_check, **0644**)) < **0**) {

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

**return**;

}

**if**((fd\_src = open(src, O\_RDONLY)) < **0**) {

fprintf(stderr, "open error for src");

**return**;

}

memset(buf, **0**, PATHMAX);

**while**((readcnt = read(fd\_src, buf, PATHMAX)) > **0**) {

write(fd\_sync, buf, readcnt);

memset(buf, **0**, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

**if**(utime(dst\_check, &timebuf) < **0**) {

fprintf(stderr, "utime error for dst\_check**\n**");

**return**;

}

unlink(orifile\_rename);

write\_log(src\_fname, &src\_statbuf);

**return**;

}

}

**if**(checksum == **0**) { //이름이 일치하는 파일이 없는 경우

memset(dst\_check, **0**, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, src\_fname);

//dst 디렉토리에 만들 파일 경로

**if**((fd\_sync = creat(dst\_check, **0644**)) < **0**) {

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

**return**;

}

**if**((fd\_src = open(src, O\_RDONLY)) < **0**) {

fprintf(stderr, "open error for src");

**return**;

}

memset(buf, **0**, PATHMAX);

**while**((readcnt = read(fd\_src, buf, PATHMAX)) > **0**) {

write(fd\_sync, buf, readcnt);

memset(buf, **0**, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

**if**(utime(dst\_check, &timebuf) < **0**) {

fprintf(stderr, "utime error for dst\_check**\n**");

**return**;

}

write\_log(src\_fname, &src\_statbuf);

}

**return**;

}

}

- 동기화를 진행하는 함수이다.

- src가 파일일 때와 디렉토리일 때를 나누어서 생각한다.

- dst에 같은 이름의 파일이 존재하면 파일의 이름을 바꾼 후, 새 파일을 생성하여 src의 파일과 동일하게 한다.

- 이후, 이름을 바꿔둔 파일을 삭제하여 동기화를 완료한다.

- 동기화를 파일 단위로 진행한다.

- 동기화에 성공한 파일의 이름과 사이즈를 로그 파일에 작성한다.

\* int dir\_check(const struct dirent \*info);

**int** **dir\_check**(**const** **struct** dirent \*info) {

**if**(!strncmp(info->d\_name, ".", **1**))

**return** **0**;

**else**

**return** **1**;

} //scandir 필터함수

- scandir 함수를 사용할 때, 현재 디렉토리와 상위 디렉토리를 제외하기 위한 필터 함수이다.

\* void write\_log(char \*fname, struct stat \*statbuf);

**void** **write\_log**(**char** \*fname, **struct** stat \*statbuf) {

fprintf(fp, " %s %ldbytes**\n**", fname, statbuf->st\_size);

**return**;

} //로그를 작성하는 함수

- ssu\_rsync\_log 로그파일에 로그를 작성하는 함수이다.

- 파일의 이름과 파일의 사이즈를 로그에 작성한다.

- 동기화가 진행된 파일만 로그에 작성한다.

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

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

{ //프로그램의 수행시간 측정

end\_t->tv\_sec -= begin\_t->tv\_sec; //끝난 시간에서 처음시간을 뺌

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

end\_t->tv\_sec--;

end\_t->tv\_usec += **1000000**;

} //나중의 us값이 작을경우 1초를 뺴서 us값이 더해줌

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

printf("Runtime: %ld:%02ld:%06ld(min:sec:usec)**\n**",

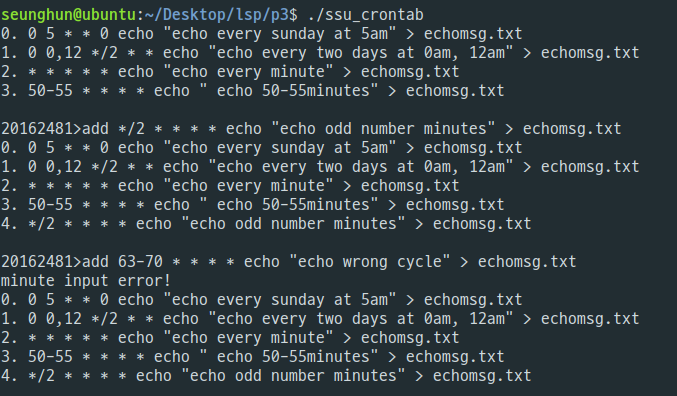
end\_t->tv\_sec / **60**, end\_t->tv\_sec % **60**, end\_t->tv\_usec);

} //프로그램 수행시간을 분, 초, 마이크로초로 출력

- 프로그램의 수행 시간을 측정하는 함수이다.

4. 테스트 및 결과

**<ssu\_crontab & ssu\_crond>**

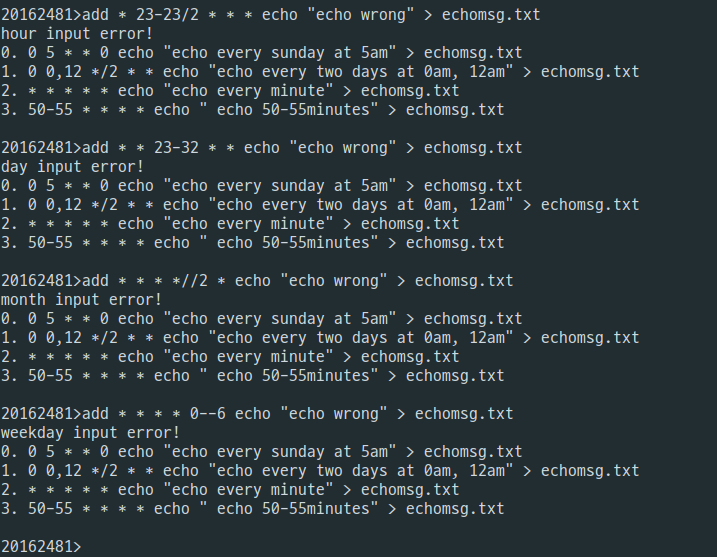


- ssu\_crontab을 실행한 화면이다

- 초기에 프롬프트를 출력하기 전에 ssu\_crontab\_file을 출력한다.

- 주기에 문제가 없는 문장일 경우,

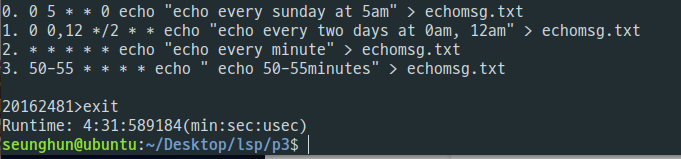
- 에러가 포함된 문장을 입력하는 경우, 에러를 출력한다.



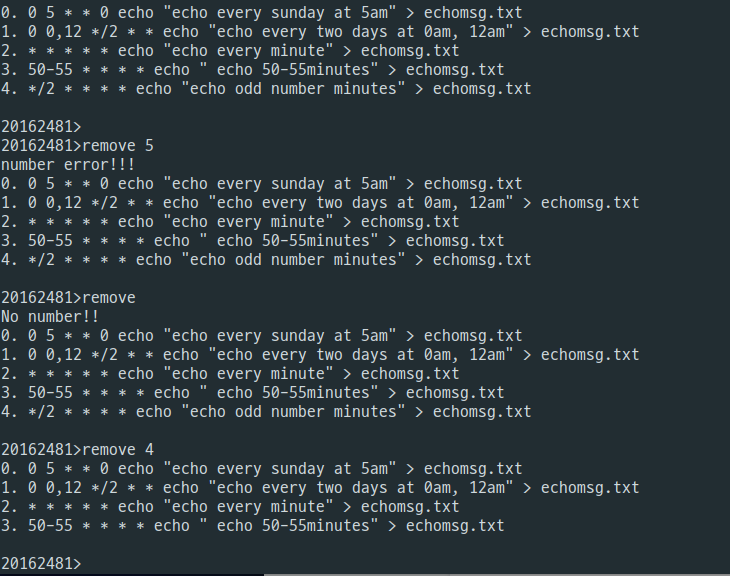
- 문장에 오류가 있는 경우 그에 맞는 에러를 출력한다.

- 각각 오류가 있는 부분을 알려준다.

- 오류가 있는 문장을 입력하는 경우 ssu\_crontab\_file에 추가되지 않고, 로그 파일에도 추가되지 않는다.



- exit를 입력하는 경우, 프로그램을 종료하고, 총 프로그램의 수행시간을 출력한다.

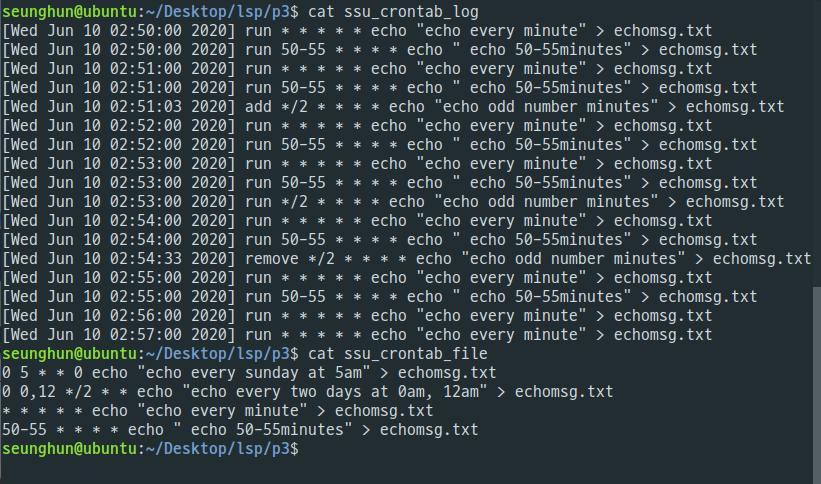


- 프롬프트에 개행 문자를 입력하는 경우 프롬프트를 재출력한다.

- remove 뒤에 범위 바깥의 숫자를 입력하는 경우 오류를 출력한다.

- remove 뒤에 숫자가 없는 경우 오류를 출력한다.

- remove 4를 입력했을 때, 4번 문장이 삭제되었음을 확인할 수 있다.

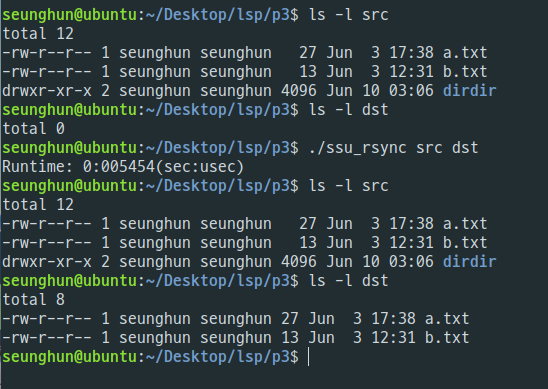


- 위의 실행결과를 작성하는 동안, ssu\_crond가 작동하면서 작성한 로그이다.

- 명령어가 실행되고 로그가 작성되었음을 알 수 있다.

- crontab에서 수행한 add, remove 의 명령도 작성되어있는 것을 볼 수 있다.

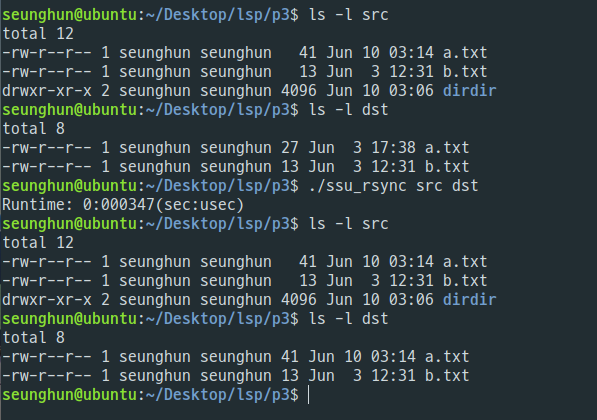
**<ssu\_rsync>**



- src 디렉토리와 dst 디렉토리를 동기화 한 실행화면이다.

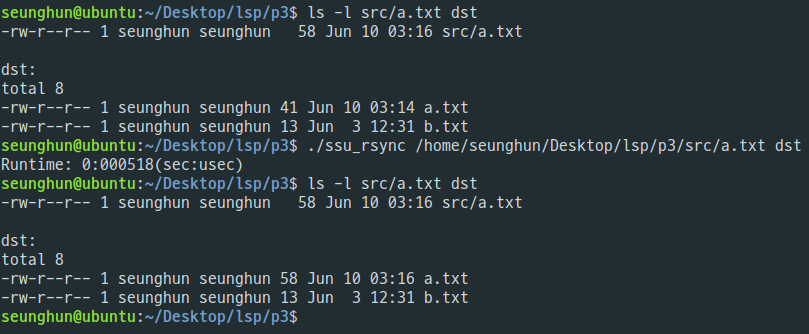
- dst가 비어있을 때, src와 동일한 디렉토리가 아닌 파일을 새 파일을 생성하여 동기화 하였다.

- 두 파일의 사이즈, 이름, 최종 수정시간이 동일한 것을 확인할 수 있다.



- a.txt의 최종 수정시간만 다를 때, 동기화를 진행한 실행화면이다.

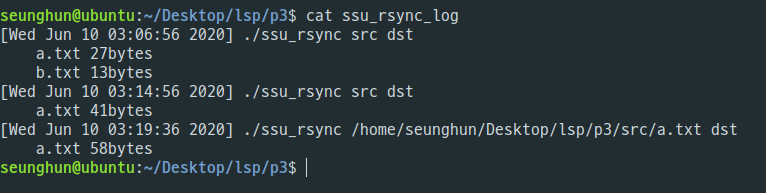
- b.txt의 이름, 사이즈, 최종수정시간이 동일하므로, a.txt만 동기화된 것을 볼 수 있다.



- src가 파일일 때 실행화면이다.

- src가 파일일 때도 동일하게, dst 디렉토리 내의 같은 이름의 파일과 사이즈, 최종수정시간을 비교한다.

- 최종 수정시간과 사이즈가 다르므로, 파일을 동일한 파일로 동기화한 것을 볼 수 있다.



- 위의 실행결과 화면을 생성하면서 작성된 ssu\_rsync\_log 로그 파일이다.

- 가장 초기의 명령, 2번쨰 명령, 3번째 명령 모두 작성되어있는 것을 볼 수 있다.

5. 소스코드

**<ssu\_crond.c>**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <time.h>

#include <sys/time.h>

#include <dirent.h>

#include <syslog.h>

#include <signal.h>

#include <limits.h>

#include <utime.h>

#include <stdbool.h>

#define BUFSIZE 1024

int mydaemon\_init(void); //디몬 프로세스

void do\_command(void); //정해진 시간에 명령어를 읽는 함수

void play\_command(char buf[]); //읽은 명령어를 실행하는 함수

void check\_cycle(char arr[], int num); //주기를 실제시간으로 환산하는 함수

int check\_star(char arr[]); //주기가 \*로만 이루어져있는지 확인하는 함수

void write\_log(char buf[]); //로그를 작성하는 함수

FILE \*filefp;

FILE \*logfp;

char s\_min[20];

char s\_hour[20];

char s\_day[20];

char s\_month[20];

char s\_wday[20];

//주기 받아옴

bool min[60];

bool hour[24];

bool day[32];

bool month[13];

bool wday[7];

//주기를 각각의 단위로 환산함

int main(void) {

if(mydaemon\_init() < 0) {

fprintf(stderr, "daemon failed\n");

exit(1);

}

return 0;

}

int mydaemon\_init(void) {

pid\_t pid;

int fd, maxfd;

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

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

exit(1);

}

else if (pid != 0)

exit(0);

pid = getpid();

setsid();

signal(SIGTTIN, SIG\_IGN);

signal(SIGTTOU, SIG\_IGN);

signal(SIGTSTP, SIG\_IGN);

maxfd = getdtablesize();

for(fd = 0; fd < maxfd; fd++)

close(fd);

umask(0);

//chdir("/");

fd = open("/dev/null", O\_RDWR);

dup(0);

dup(0);

if((filefp = fopen("ssu\_crontab\_file", "r")) == NULL) {

fprintf(stderr, "fopen error for command file\n");

exit(1);

} //명령어가 들어있는 파일을 오픈

if((logfp = fopen("ssu\_crontab\_log", "a+")) == NULL) {

fprintf(stderr, "fopen error for logfile\n");

} //로그 파일을 오픈

setbuf(logfp, NULL);

time\_t prevtime, curtime;

struct tm t1, t2;

prevtime = time(NULL);

while(1) { //과거에 저장해둔 시간과 현재 시간을 비교하여 분이 바뀌면 실행

localtime\_r(&prevtime, &t1);

curtime = time(NULL);

localtime\_r(&curtime, &t2);

if(t1.tm\_min + 1 == t2.tm\_min) {

prevtime = curtime;

do\_command(); //명령어를 실행 하는 함수 실행

}

sleep(1);

}

return 0;

}

void do\_command(void) {

char sentence[BUFSIZE];

rewind(filefp); //파일 포인터를 파일의 가장 처음으로 돌림

while(fgets(sentence, BUFSIZE, filefp) != NULL) { //한문장 읽어옴

play\_command(sentence);

} //파일의 끝까지 한줄씩 읽어서, 한줄씩 실행여부를 판단함

return;

}

void play\_command(char buf[]) {

char \*token;

char tmpbuf[BUFSIZE];

char command[BUFSIZE];

struct tm t;

time\_t now;

memset(s\_min, 0, sizeof(s\_min));

memset(s\_hour, 0, sizeof(s\_hour));

memset(s\_day, 0, sizeof(s\_day));

memset(s\_month, 0, sizeof(s\_month));

memset(s\_wday, 0, sizeof(s\_wday));

memset(min, 0, sizeof(min));

memset(hour, 0, sizeof(hour));

memset(day, 0, sizeof(day));

memset(month, 0, sizeof(month));

memset(wday, 0, sizeof(wday));

memset(tmpbuf, 0, BUFSIZE);

//배열 초기화

strcpy(tmpbuf, buf);

token = strtok(tmpbuf, " ");

strcpy(s\_min, token);

token = strtok(NULL, " ");

strcpy(s\_hour, token);

token = strtok(NULL, " ");

strcpy(s\_day, token);

token = strtok(NULL, " ");

strcpy(s\_month, token);

token = strtok(NULL, " ");

strcpy(s\_wday, token);

token = strtok(NULL, "\n");

strcpy(command, token);

//주기를 각각 나누어서 저장함

check\_cycle(s\_min, 1);

check\_cycle(s\_hour, 2);

check\_cycle(s\_day, 3);

check\_cycle(s\_month, 4);

check\_cycle(s\_wday, 5);

//나눠진 주기에 맞추어 실행가능한 시간을 정리함

now = time(NULL);

localtime\_r(&now, &t);

if(min[t.tm\_min] && hour[t.tm\_hour] && day[t.tm\_mday] && month[t.tm\_mon+1] && wday[t.tm\_wday]) {

system(command);

write\_log(buf);

} //현재 시간이 실행 가능 상황에 적합할때, system을 통해 명령어를 실행하고 로그를 작성함

//시간이 모두 1이면 실행

return;

}

void write\_log(char buf[]) { //로그를 작성하는 함수

time\_t curtime;

char tmp[BUFSIZE];

memset(tmp, 0, BUFSIZE);

curtime = time(NULL);

sprintf(tmp, "[%s", ctime(&curtime));

tmp[strlen(tmp) - 1] = ']';

strcat(tmp, " run ");

strcat(tmp, buf);

fputs(tmp, logfp);

//현재 시간과 명령어를 로그에 출력함

return;

}

void check\_cycle(char arr[], int num) { //주기를 실제 가능 시간으로 환산하는 함수

int slash\_loc = -1;

int dash\_loc = -1;

int term;

char \*ptr;

char buf[20][20];

char tmp[100];

int i = 0;

int cnt = 0;

int checksum = 0;

memset(buf, 0, sizeof(buf));

strcpy(tmp, arr);

ptr = strtok(tmp, ",");

strcpy(buf[i++], ptr);

cnt++;

while((ptr = strtok(NULL, ","))!= NULL) {

strcpy(buf[i++], ptr);

cnt++;

} //쉼표를 기준으로 주기를 분해함

for(i = 0; i < cnt; i++) { //분해한 주기를 하나씩 시간으로 바꿈

dash\_loc = -1;

slash\_loc = -1;

checksum = 0;

if(num == 1) { //분

if(check\_star(buf[i])) {

memset(min, 1, sizeof(min));

return;

} //\*만 있으면 모든 경우에 실행

if(buf[i][0] == '\*' && buf[i][1] == '/') {

term = atoi(buf[i]+2);

for(int i = 0; i < sizeof(min); i++) {

if((i % term) == (term - 1))

min[i] = 1;

}

return;

} //\*/로 시작하면 뒤에 숫자 간격만큼 건너서 실행

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-' || buf[i][x] == '/')

checksum = 1;

} //숫자 단독인지 확인

if(!checksum) {

int tmp = atoi(buf[i]);

min[tmp] = 1;

return;

} //그 숫자 ON

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-') {

dash\_loc = x;

break;

}

}//dash가 있는 경우

checksum = 0;

if(dash\_loc != -1) { //dash 가 있을때

int start = atoi(buf[i]);

int end = atoi(buf[i] + dash\_loc + 1);

for(int k = 0; k < strlen(buf[i]); k++) {

if(buf[i][k] == '/') {

checksum = 1;

slash\_loc = k;

break;

}

}

if(checksum) { // slash가 있을떄

int div = atoi(buf[i]+slash\_loc + 1);

int hop = 0;

for(int s = start; s <= end; s++) {

hop++;

if(hop % div == 0)

min[s] = 1;

}

}

else { //slash가 뒤에 없을떄

for(int s = start; s<= end; s++)

min[s] = 1;

}

}

}

else if (num == 2) { //시간

if(check\_star(buf[i])) {

memset(hour, 1, sizeof(hour));

return;

} //\*만 있을떄

if(buf[i][0] == '\*' && buf[i][1] == '/') {

term = atoi(buf[i]+2);

for(int i = 0; i < sizeof(hour); i++) {

if((i % term) == (term - 1))

hour[i] = 1;

}

return;

} //\*/일때

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-' || buf[i][x] == '/')

checksum = 1;

} //쉼표로 뽀갠것이 숫자일떄

if(!checksum) {

int tmp = atoi(buf[i]);

hour[tmp] = 1;

return;

} //그 숫자 ON

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-') {

dash\_loc = x;

break;

}

}

checksum = 0;

if(dash\_loc != -1) { //dash 가 있을때

int start = atoi(buf[i]);

int end = atoi(buf[i] + dash\_loc + 1);

for(int k = 0; k < strlen(buf[i]); k++) {

if(buf[i][k] == '/') {

checksum = 1;

slash\_loc = k;

break;

}

}

if(checksum) { // slash가 있을떄

int div = atoi(buf[i]+slash\_loc + 1);

int hop = 0;

for(int s = start; s <= end; s++) {

hop++;

if(hop % div == 0)

hour[s] = 1;

}

return;

}

else { //slash가 뒤에 없을떄

for(int s = start; s<= end; s++)

hour[s] = 1;

return;

}

}

}

else if (num == 3) { //날짜

if(check\_star(buf[i])) {

memset(day, 1, sizeof(day));

return;

} //\*만 있을떄

if(buf[i][0] == '\*' && buf[i][1] == '/') {

term = atoi(buf[i]+2);

for(int i = 0; i < sizeof(day); i++) {

if((i % term) == (term - 1))

day[i] = 1;

}

return;

} //'\*/'이 있을떄

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-' || buf[i][x] == '/')

checksum = 1;

} //쉼표로 뽀갠것이 숫자일떄

if(!checksum) {

int tmp = atoi(buf[i]);

day[tmp] = 1;

return;

} //그 숫자 ON

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-') {

dash\_loc = x;

break;

}

}

checksum = 0;

if(dash\_loc != -1) { //dash 가 있을때

int start = atoi(buf[i]);

int end = atoi(buf[i] + dash\_loc + 1);

for(int k = 0; k < strlen(buf[i]); k++) {

if(buf[i][k] == '/') {

checksum = 1;

slash\_loc = k;

break;

}

}

if(checksum) { // slash가 있을떄

int div = atoi(buf[i]+slash\_loc + 1);

int hop = 0;

for(int s = start; s <= end; s++) {

hop++;

if(hop % div == 0)

day[s] = 1;

}

return;

}

else { //slash가 뒤에 없을떄

for(int s = start; s<= end; s++)

day[s] = 1;

return;

}

}

}

else if(num == 4) { //달

if(check\_star(buf[i])) {

memset(month, 1, sizeof(month));

return;

}

if(buf[i][0] == '\*' && buf[i][1] == '/') {

term = atoi(buf[i]+2);

for(int i = 1; i < sizeof(month); i++) {

if((i % term) == (term - 1))

month[i] = 1;

}

return;

}// 문장에 '\*/'이 있을때

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-' || buf[i][x] == '/')

checksum = 1;

} //쉼표로 뽀갠것이 숫자일떄

if(!checksum) {

int tmp = atoi(buf[i]);

month[tmp] = 1;

return;

} //그 숫자 ON

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-') {

dash\_loc = x;

break;

}

}

checksum = 0;

if(dash\_loc != -1) { //dash 가 있을때

int start = atoi(buf[i]);

int end = atoi(buf[i] + dash\_loc + 1);

for(int k = 0; k < strlen(buf[i]); k++) {

if(buf[i][k] == '/') {

checksum = 1;

slash\_loc = k;

break;

}

}

if(checksum) { // slash가 있을떄

int div = atoi(buf[i]+slash\_loc + 1);

int hop = 0;

for(int s = start; s <= end; s++) {

hop++;

if(hop % div == 0)

month[s] = 1;

}

return;

}

else { //slash가 뒤에 없을떄

for(int s = start; s<= end; s++)

month[s] = 1;

return;

}

}

}

else if (num == 5) { //요일

if(check\_star(buf[i])) {

memset(wday, 1, sizeof(wday));

return;

}

if(buf[i][0] == '\*' && buf[i][1] == '/') {

term = atoi(buf[i]+2);

for(int i = 0; i < sizeof(wday); i++) {

if((i % term) == (term - 1))

wday[i] = 1;

}

return;

}

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-' || buf[i][x] == '/')

checksum = 1;

} //쉼표로 뽀갠것이 숫자일떄

if(!checksum) {

int tmp = atoi(buf[i]);

wday[tmp] = 1;

return;

} //그 숫자 ON

for(int x = 0; x < strlen(buf[i]); x++) {

if(buf[i][x] == '-') {

dash\_loc = x;

break;

}

}

checksum = 0;

if(dash\_loc != -1) { //dash 가 있을때

int start = atoi(buf[i]);

int end = atoi(buf[i] + dash\_loc + 1);

for(int k = 0; k < strlen(buf[i]); k++) {

if(buf[i][k] == '/') {

checksum = 1;

slash\_loc = k;

break;

}

}

if(checksum) { // slash가 있을떄

int div = atoi(buf[i]+slash\_loc + 1);

int hop = 0;

for(int s = start; s <= end; s++) {

hop++;

if(hop % div == 0)

wday[s] = 1;

}

return;

}

else { //slash가 뒤에 없을떄

for(int s = start; s<= end; s++)

wday[s] = 1;

return;

}

}

}

}

return;

}

int check\_star(char arr[]) { //주기가 \*뿐인지 확인하는 함수

if(!strcmp(arr, "\*"))

return 1;

else

return 0;

}

**<ssu\_crontab.c>**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <time.h>

#include <sys/time.h>

#include <dirent.h>

#include <syslog.h>

#include <signal.h>

#include <limits.h>

#include <utime.h>

#define BUFSIZE 1024

void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t); //프로그램 수행시간 측정

void print\_crontabfile(void); //ssu\_crontab\_file에 쓰는 함수

int check\_correct(char arr[], int index); //주기가 올바른지 확인하는 함수

void add\_command(void); //명령어를 추가하는 함수

void remove\_command(void); //명령어를 삭제하는 함수

int check\_star(char arr[]); //주기가 \*인지 확인하는 함수

int check\_slash(char arr[]); //주기에 /가 포함되어있는지 확인하는 함수

int check\_dash(char arr[]); //주기에 -가 포함되어있는지 확인하는 함수

void write\_log(char arr[], char sentence[]); //log를 작성하는 함수

char const\_input[BUFSIZE];

char input[BUFSIZE];

char command[BUFSIZE];

int crontabnum;

FILE \*filefp; //ssu\_crontab\_file 파일 포인터

FILE \*logfp; //ssu\_crontab\_log 파일 포인터

char \*reserve[3] = {"add", "remove", "exit"};

int main(void)

{

struct timeval begin\_t, end\_t;

char todo[10]; //가장 앞의 명령어를 저장

int i = 0, checksum = 0;

int enter = 1;

gettimeofday(&begin\_t, NULL); //프로그램 시작 시간 저장

if((filefp = fopen("ssu\_crontab\_file", "r+")) < 0) {

filefp = fopen("ssu\_crontab\_file", "w+");

fclose(filefp);

filefp = fopen("ssu\_crontab\_file", "r+");

} //file 파일을 오픈함

if((logfp = fopen("ssu\_crontab\_log", "a+")) < 0) {

fprintf(stderr, "log file fopen error\n");

exit(1);

} //로그 파일을 오픈함

setbuf(filefp, NULL);

setbuf(stdout, NULL);

setbuf(logfp, NULL);

//버퍼를 NULL로 설정

while(1) {

memset(const\_input, 0, BUFSIZE);

memset(input, 0, BUFSIZE);

memset(todo, 0, sizeof(todo));

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

if(enter)

print\_crontabfile(); //file의 내용을 출력함

enter = 1;

printf("20162481>"); //프롬프트를 구성함

fgets(input, BUFSIZE, stdin);

strcpy(const\_input, input);

i = 0;

if(input[0] == '\n') {

enter = 0;

continue;

}

//개행 입력하면 프롬프트 재출력

while(input[i] != ' ' && input[i] != '\n') {

todo[i] = input[i];

i++;

}

strcpy(command, input + i + 1);

//사용자가 입력한것을 add remove 부분과 뒷부분을 분리

checksum = 0;

for(i = 0; i < 3; i++) {

if(!strcmp(reserve[i], todo))

checksum += (i + 1);

}

if(!checksum) {

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

continue;

} //add remove exit 이외의 명령일 경우 무시

if(!strcmp(reserve[0], todo)) //add일떄

add\_command();

else if(!strcmp(reserve[1], todo)) //remove 일떄

remove\_command();

else if(!strcmp(reserve[2], todo)) //exit일떄

break;

}

gettimeofday(&end\_t, NULL);

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

exit(0);

}

void add\_command(void) { //add를 실행하는 함수

char \*token;

char \*tmptmp;

char min[20];

char hour[20];

char day[20];

char month[20];

char wday[20];

int checksum;

memset(min, 0, sizeof(min));

memset(hour, 0, sizeof(hour));

memset(day, 0, sizeof(day));

memset(month, 0, sizeof(month));

memset(wday, 0, sizeof(wday));

//배열 초기화

checksum = 0;

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

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

checksum++;

if(checksum < 7) {

printf("Input error!\n");

return;

}

token = strtok\_r(input, " ", &tmptmp); //명령어 컷

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(min, token);

if(check\_correct(min, 1)) {

printf("minute input error!\n");

return;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(hour, token);

if(check\_correct(hour, 2)) {

printf("hour input error!\n");

return;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(day, token);

if(check\_correct(day, 3)) {

printf("day input error!\n");

return;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(month, token);

if(check\_correct(month, 4)) {

printf("month input error!\n");

return;

}

token = strtok\_r(NULL, " ", &tmptmp);

strcpy(wday, token);

if(check\_correct(wday, 5)) {

printf("weekday input error!\n");

return;

}

//입력된 문장을 각각의 주기에 맞게 분해하여 각 주기가 올바른지 확인함

fseek(filefp, 0, SEEK\_END);

fwrite(command, strlen(command), 1, filefp); //file에 새로 명령어를 씀

write\_log("add", command); //로그 작성

return;

}

void remove\_command(void) { //remove를 수행하는 함수

int i = 0;

int num = atoi(command);

int del\_loc = 0; //삭제할 문장의 시작 위치

int next\_loc = 0; //삭제할 문장 다음줄의 시작 위치

char tmp[BUFSIZE];

char save[BUFSIZE];

char del\_command[BUFSIZE];

int filesize;

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

printf("No number!!\n");

return;

}

if(num >= crontabnum || num < 0) {

printf("number error!!!\n");

return;

}

fseek(filefp, 0, SEEK\_END);

filesize = ftell(filefp); //원래의 파일 크기 구하기

fseek(filefp, 0, SEEK\_SET);

for(i = 0; i < num; i++) {

memset(tmp, 0, BUFSIZE);

fgets(tmp, BUFSIZE, filefp);

del\_loc += strlen(tmp);

}

memset(tmp, 0, BUFSIZE);

memset(del\_command, 0, BUFSIZE);

fgets(del\_command, BUFSIZE, filefp);

next\_loc = ftell(filefp);

//삭제할 문장의 시작점과 그 다음 문장의 시작점을 저장함

memset(save, 0, BUFSIZE);

if(fread(save, BUFSIZE, 1,filefp) < 0) {

fprintf(stderr, "fread error!\n");

return;

} //삭제할 문장의 시작점부터 그 다음문장을 덮어씀

fseek(filefp, del\_loc, SEEK\_SET);

fwrite(save, BUFSIZE, 1, filefp);

truncate("ssu\_crontab\_file", filesize - strlen(del\_command));

//파일에서 삭제한만큼 파일의 크기를 줄임

write\_log("remove", del\_command); //로그 작성

return;

}

int check\_correct(char arr[], int index) { //주기가 올바른지 확인하는 함수

int checksum = 0;

int slash\_loc = -1;

int dash\_loc = -1;

int length;

char buf[20][20];

char tmp[100];

char \*ptr, \*sav;

int i = 0;

int cnt = 0;

memset(buf, 0, sizeof(buf));

memset(tmp, 0, sizeof(tmp));

strcpy(tmp, arr);

ptr = strtok\_r(tmp, ",", &sav);

strcpy(buf[i++], ptr);

cnt++;

//쉼표 단위로 주기를 분해함

while((ptr = strtok\_r(NULL, ",", &sav)) != NULL) {

strcpy(buf[i++], ptr);

cnt++;

} //쉼표를 기준으로 토큰으로 분해

for(i = 0; i < cnt; i++) { //분해한 것 각각을 분석함

dash\_loc = -1;

slash\_loc = -1;

checksum = 0;

for(int f = 0; f < strlen(buf[i]) - 1; f++) {

if((buf[i][f] == '/' || buf[i][f] == '-') && (buf[i][f+1] == '-' || buf[i][f+1] == '/'))

return 1;

} //'-''/'기호가 2개이상 나올 경우 무조건 불가능한 주기

if(index == 1) { // 분 확인

if(check\_star(buf[i])) return 0; // \*만 있으면 ok

if(buf[i][0] == '\*' && buf[i][1] == '/') {

checksum = check\_slash(buf[i] + 2);

checksum = check\_dash(buf[i] + 2);

if(checksum) return 1;

int temp = atoi(buf[i]+2);

if(temp < 0 || temp > 60) return 1;

} // '\*/'로 시작하는 경우 뒤에 숫자만 올 수 있음

if(check\_dash(buf[i])) { //문장에 '-'가 있을때

int loc = check\_dash(buf[i]);

int tmp1 = atoi(buf[i]);

int tmp2 = atoi(buf[i] + loc + 1);

if(tmp1 >= tmp2) return 1;

if(tmp1 < 0 || tmp2 > 60) return 1;

loc = check\_slash(buf[i]+ loc + 1);

if(loc) { //문장에 '/'도 포함되어있는 경우

int tmp3 = atoi(buf[i] + loc + 1);

if(tmp3 > tmp2 - tmp1) return 1;

}

}

else if(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

int tmp1 = atoi(buf[i]);

if(tmp1 < 0 || tmp1 > 60) return 1;

} //문장에 - / 가 없는 경우 숫자만 판단함

return 0;

}

else if (index == 2) { //시간 확인

if(check\_star(buf[i])) return 0; // \*만 있으면 ok

if(buf[i][0] == '\*' && buf[i][1] == '/') {

checksum = check\_slash(buf[i] + 2);

checksum = check\_dash(buf[i] + 2);

if(checksum) return 1;

int temp = atoi(buf[i]+2);

if(temp < 0 || temp > 23) return 1;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올수 있음

if(check\_dash(buf[i])) { //문장에 -가 있을때

int loc = check\_dash(buf[i]);

int tmp1 = atoi(buf[i]);

int tmp2 = atoi(buf[i] + loc + 1);

if(tmp1 >= tmp2) return 1;

if(tmp1 < 0 || tmp2 > 23) return 1;

loc = check\_slash(buf[i]+ loc + 1);

if(loc) { //같은 문장에 /도 포함되어있을때

int tmp3 = atoi(buf[i] + loc + 1);

if(tmp3 > tmp2 - tmp1) return 1;

}

}

else if(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

int tmp1 = atoi(buf[i]);

if(tmp1 < 0 || tmp1 > 23) return 1;

} //문장에 -또는 /가 없을 때

return 0;

}

else if(index == 3) { //날짜 확인

if(check\_star(buf[i])) return 0; // \*만 있으면 ok

if(buf[i][0] == '\*' && buf[i][1] == '/') {

checksum = check\_slash(buf[i] + 2);

checksum = check\_dash(buf[i] + 2);

if(checksum) return 1;

int temp = atoi(buf[i]+2);

if(temp < 1 || temp > 31) return 1;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올 수 있음

if(check\_dash(buf[i])) { //문장에 -가 있는 경우

int loc = check\_dash(buf[i]);

int tmp1 = atoi(buf[i]);

int tmp2 = atoi(buf[i] + loc + 1);

if(tmp1 >= tmp2) return 1;

if(tmp1 < 1 || tmp2 > 31) return 1;

loc = check\_slash(buf[i]+ loc + 1);

if(loc) { //같은 문장에 /도 포함되어있는 경우

int tmp3 = atoi(buf[i] + loc + 1);

if(tmp3 > tmp2 - tmp1) return 1;

}

}

else if(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

int tmp1 = atoi(buf[i]);

if(tmp1 < 1 || tmp1 > 31) return 1;

} //문장에 -또는 /기호가 없을 경우

return 0;

}

else if(index == 4) { //월 확인

if(check\_star(buf[i])) return 0; // \*만 있으면 ok

if(buf[i][0] == '\*' && buf[i][1] == '/') {

checksum = check\_slash(buf[i] + 2);

checksum = check\_dash(buf[i] + 2);

if(checksum) return 1;

int temp = atoi(buf[i]+2);

if(temp < 1 || temp > 12) return 1;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올 수 있음

if(check\_dash(buf[i])) { //문장에 -가 포함되어있는 경우

int loc = check\_dash(buf[i]);

int tmp1 = atoi(buf[i]);

int tmp2 = atoi(buf[i] + loc + 1);

if(tmp1 >= tmp2) return 1;

if(tmp1 < 1 || tmp2 > 12) return 1;

loc = check\_slash(buf[i]+ loc + 1);

if(loc) { //문장에 /도 포함되어 있는 경우

int tmp3 = atoi(buf[i] + loc + 1);

if(tmp3 > tmp2 - tmp1) return 1;

}

}

else if(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

int tmp1 = atoi(buf[i]);

if(tmp1 < 1 || tmp1 > 12) return 1;

} //문장에 -또는 /가 포함되어 있지 않은 경우 숫자만 판단함

return 0;

}

else if(index == 5) { //요일 확인

if(check\_star(buf[i])) return 0; // \*만 있으면 ok

if(buf[i][0] == '\*' && buf[i][1] == '/') {

checksum = check\_slash(buf[i] + 2);

checksum = check\_dash(buf[i] + 2);

if(checksum) return 1;

int temp = atoi(buf[i]+2);

if(temp < 0 || temp > 6) return 1;

} //문장이 \*/로 시작하는 경우 뒤에 숫자만 올 수 있음

if(check\_dash(buf[i])) { //문장에 -가 포함되어 있을때

int loc = check\_dash(buf[i]);

int tmp1 = atoi(buf[i]);

int tmp2 = atoi(buf[i] + loc + 1);

if(tmp1 >= tmp2) return 1;

if(tmp1 < 0 || tmp2 > 6) return 1;

loc = check\_slash(buf[i]+ loc + 1);

if(loc) { //문장에 /가 포함되어 있는 경우

int tmp3 = atoi(buf[i] + loc + 1);

if(tmp3 > tmp2 - tmp1) return 1;

}

}

else if(!check\_dash(buf[i]) && !check\_slash(buf[i])) {

int tmp1 = atoi(buf[i]);

if(tmp1 < 0 || tmp1 > 6) return 1;

} //문장에 - 또는 /가 포함되지 않은 경우

return 0;

}

}

}

int check\_dash(char arr[]) { //문장에 '-' 기 포함되었는지 확인하는 함수

int i;

for(i = 0; i < strlen(arr); i++) {

if(\*(arr + i) == '-')

return i;

}

return 0;

}

int check\_slash(char arr[]) { //문장에 '/' 가 포함되었는지 확인하는 함수

int i;

for(i = 0; i <strlen(arr); i++) {

if(\*(arr + i) == '/')

return i;

}

return 0;

}

int check\_star(char arr[]) { //문장에 \*만 포함되어 있는지 확인하는 함수

if(!strcmp(arr, "\*"))

return 1;

else

return 0;

}

void write\_log(char arr[], char sentence[]) { //명령어를 로그에 작성하는 함수

time\_t curtime;

char timebuf[BUFSIZE];

memset(timebuf, 0, BUFSIZE);

curtime = time(NULL);

sprintf(timebuf, "[");

sprintf(timebuf+1, "%s", ctime(&curtime));

timebuf[strlen(timebuf)-1] = ']';

strcat(timebuf, " ");

strcat(timebuf, arr);

strcat(timebuf, " ");

strcat(timebuf, sentence);

fwrite(timebuf, strlen(timebuf), 1, logfp);

return;

}

void print\_crontabfile(void) { //입력받은 명령어를 파일에 쓰는 함수

char tmp[BUFSIZE];

memset(tmp, 0, BUFSIZE);

int num = 0;

fseek(filefp, 0, SEEK\_SET);

while(fgets(tmp, BUFSIZE, filefp) != NULL) {

printf("%d. ", num);

fputs(tmp, stdout);

num++;

}

printf("\n");

crontabnum = num; //crontabfile에 있는 갯수를 저장

return;

}

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

{ //프로그램의 수행시간 측정

end\_t->tv\_sec -= begin\_t->tv\_sec; //끝난 시간에서 처음시간을 뺌

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

end\_t->tv\_sec--;

end\_t->tv\_usec += 1000000;

} //나중의 us값이 작을경우 1초를 뺴서 us값이 더해줌

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

printf("Runtime: %ld:%02ld:%06ld(min:sec:usec)\n",

end\_t->tv\_sec / 60, end\_t->tv\_sec % 60, end\_t->tv\_usec);

} //프로그램 수행시간을 분, 초, 마이크로초로 출력

**<ssu\_rsync.c>**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <time.h>

#include <sys/time.h>

#include <dirent.h>

#include <syslog.h>

#include <signal.h>

#include <limits.h>

#include <utime.h>

#define PATHMAX 1024

#define FILEMAX 256

void print\_usage(char \*str); //사용법 출력

void make\_sync(void); //동기화를 진행하는 함수

int dir\_check(const struct dirent \*info); //scandir 필터함수

void write\_log(char \*fname, struct stat \*statbuf); //로그 작성 함수

void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t); //프로그램 수행 시간 측정

char src[PATHMAX];

char dst[PATHMAX];

char command[PATHMAX];

FILE \*fp;

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

{

struct timeval begin\_t, end\_t;

gettimeofday(&begin\_t, NULL); //시작시간 저장

struct stat statbuf;

setbuf(stdout, NULL);

time\_t now;

//struct tm \*t;

char timestring[FILEMAX];

memset(src, 0, PATHMAX);

memset(dst, 0, PATHMAX);

memset(command, 0, PATHMAX);

memset(timestring, 0, FILEMAX);

if(!(argc == 3 || argc == 4)) {

print\_usage(argv[0]);

exit(1);

}

//인자가 잘못들어온 경우 예외처리

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

strcat(command, argv[i]);

if(i < argc-1)

strcat(command, " ");

}

//main 함수 인자로 들어온 명령어를 저장해둠

time(&now);

sprintf(timestring, "[%s", ctime(&now));

timestring[strlen(timestring) - 1] = '\0';

strcat(timestring, "]");

if((fp = fopen("ssu\_rsync\_log", "a+")) < 0) {

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

exit(1);

}

fprintf(fp, "%s %s\n", timestring, command); //로그 첫줄 작성

if(argc == 3) {

realpath(argv[1], src);

realpath(argv[2], dst);

}

else if (argc == 4) {

realpath(argv[2], src);

realpath(argv[3], dst);

}

//인자로 입력받은 src dst를 절대경로로 저장

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

print\_usage(argv[0]);

exit(1);

} //dst가 없는 경우 에러처리 후 종료

if(!S\_ISDIR(statbuf.st\_mode) || access(dst, W\_OK) || access(dst, R\_OK) || access(dst, X\_OK)) {

print\_usage(argv[0]);

exit(1);

} //dst가 디렉토리가 아니고, 쓸수 없고, 읽을수 없고 실행할 수 없으면 에러처리 후 종료

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

print\_usage(argv[0]);

exit(1);

} //src를 찾을수 없는 경우 에러처리 후 종료

if(access(src, W\_OK) || access(src, R\_OK) || (S\_ISDIR(statbuf.st\_mode) && access(src, X\_OK))) {

print\_usage(argv[0]);

exit(1);

} //src가 쓸수 없고, 읽을수 없고, 실행할 수 없으면 에러처리 후 종료

make\_sync(); //동기화 함수 호출

fclose(fp);

gettimeofday(&end\_t, NULL); //끝시간 저장

ssu\_runtime(&begin\_t, &end\_t); //시간계산, 끝시간 - 처음 시간

return 0;

}

void make\_sync(void) { //동기화 하는 함수

struct dirent \*\*namelist\_src; //src가 디렉토리일때 내부 파일 목록

struct dirent \*\*namelist\_dst; //dst의 파일 목록

int src\_count, dst\_count; //src dst의 파일 갯수

struct stat statbuf;

struct stat src\_statbuf;

struct utimbuf timebuf;

int index\_src;

char src\_fname[FILEMAX]; //src가 파일일 때 파일 이름을 저장함

char src\_check[PATHMAX]; //src의 파일 이름을 절대 경로로 바꿔서 저장함

char dst\_check[PATHMAX]; //dst의 파일 이름을 절대 경로로 바꿔서 저장함

char orifile\_rename[PATHMAX]; //dst에 있는 이름이 같은 파일을 이름을 바꿔서 저장

int loc, readcnt;

int fd\_sync, fd\_src;

char buf[PATHMAX]; //read\_write용 버퍼

int checksum = 0;

memset(src\_fname, 0, FILEMAX);

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

if(src[i] == '/')

loc = i;

strcpy(src\_fname, src+loc+1); //src가 파일 이름일때 파일 이름만 추출

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

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

exit(1);

}

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

index\_src = 1;

else

index\_src = 0;

//index\_src 가 1이면 src가 디렉토리, 0이면 파일

if(index\_src) { //src가 디렉토리 일때

if((src\_count = scandir(src, &namelist\_src, dir\_check, alphasort)) == -1) {

return;

}

if((dst\_count = scandir(dst, &namelist\_dst, dir\_check, alphasort)) == -1) {

return;

}

for(int i = 0; i < src\_count; i++) { //src 디렉토리 내의 파일 갯수만큼

checksum = 0;

memset(src\_check, 0, PATHMAX);

strcpy(src\_check, src);

strcat(src\_check, "/");

strcat(src\_check, namelist\_src[i]->d\_name);

if(lstat(src\_check, &src\_statbuf) < 0) {

fprintf(stderr, "lstat error2\n");

return;

}

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

continue;

//디렉토리면 패스

for(int j = 0; j < dst\_count; j++) { //dst 디렉토리 순회

memset(dst\_check, 0, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, namelist\_dst[j]->d\_name);

//dst의 파일을 절대경로로 만듬

//printf("%s\n%s\n\n", src\_check, dst\_check);

if(lstat(dst\_check, &statbuf) < 0) {

fprintf(stderr, "lstat error3\n");

exit(1);

}

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

continue;

//디렉토리면 패스

if(!strcmp(namelist\_src[i]->d\_name, namelist\_dst[j]->d\_name)){

checksum = 1;

if(src\_statbuf.st\_size == statbuf.st\_size && src\_statbuf.st\_mtime == statbuf.st\_mtime)

continue;

//파일 크기가 같고 최종 수정시간이 같으면 패스

memset(orifile\_rename, 0, PATHMAX);

strcpy(orifile\_rename, dst);

strcat(orifile\_rename, "/\_");

strcat(orifile\_rename, namelist\_dst[j]->d\_name);

//rename할꺼 만듬

rename(dst\_check, orifile\_rename);

if((fd\_sync = creat(dst\_check, 0644)) < 0) {

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

return;

}

if((fd\_src = open(src\_check, O\_RDONLY)) < 0) {

fprintf(stderr, "open error for src");

return;

}

memset(buf, 0, PATHMAX);

while((readcnt = read(fd\_src, buf, PATHMAX)) > 0) {

write(fd\_sync, buf, readcnt);

memset(buf, 0, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

if(utime(dst\_check, &timebuf) < 0) {

fprintf(stderr, "utime error for dst\_check\n");

return;

}

unlink(orifile\_rename);

write\_log(namelist\_src[i]->d\_name, &src\_statbuf);

break;

}

}

if(checksum == 0) { //dst에 파일이 없는 경우

memset(dst\_check, 0, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, namelist\_src[i]->d\_name);

//dst 디렉토리에 만들 파일 경로

if((fd\_sync = creat(dst\_check, 0644)) < 0) {

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

return;

}

if((fd\_src = open(src\_check, O\_RDONLY)) < 0) {

fprintf(stderr, "open error for src");

return;

}

memset(buf, 0, PATHMAX);

while((readcnt = read(fd\_src, buf, PATHMAX)) > 0) {

write(fd\_sync, buf, readcnt);

memset(buf, 0, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

if(utime(dst\_check, &timebuf) < 0) {

fprintf(stderr, "utime error for dst\_check\n");

return;

}

write\_log(namelist\_src[i]->d\_name, &src\_statbuf);

}

}

}

else { //src가 파일일때

if(lstat(src, &src\_statbuf) < 0) {

fprintf(stderr, "lstat error for src");

return;

}

if((dst\_count = scandir(dst, &namelist\_dst, dir\_check, alphasort)) == -1) {

return;

}

for(int i = 0; i < dst\_count; i++) { //dst 디렉토리 순회

memset(dst\_check, 0, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, namelist\_dst[i]->d\_name);

//dst 파일의 절대경로 생성

if(lstat(dst\_check, &statbuf) < 0) {

fprintf(stderr, "lstat error1\n");

return;

}

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

continue;

//디렉토리일 경우 패스함

if(!strcmp(src\_fname, namelist\_dst[i]->d\_name)) { //파일 이름이 같은게 있는 경우

checksum = 1;

if(src\_statbuf.st\_size == statbuf.st\_size && src\_statbuf.st\_mtime == statbuf.st\_mtime)

continue;

//파일 크기가 같고 최종 수정시간이 같으면 패스

memset(orifile\_rename, 0, PATHMAX);

strcpy(orifile\_rename, dst);

strcat(orifile\_rename, "/\_");

strcat(orifile\_rename, namelist\_dst[i]->d\_name);

rename(dst\_check, orifile\_rename); //dst에 원래 있던 파일을 \_(파일) 로 만듦

if((fd\_sync = creat(dst\_check, 0644)) < 0) {

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

return;

}

if((fd\_src = open(src, O\_RDONLY)) < 0) {

fprintf(stderr, "open error for src");

return;

}

memset(buf, 0, PATHMAX);

while((readcnt = read(fd\_src, buf, PATHMAX)) > 0) {

write(fd\_sync, buf, readcnt);

memset(buf, 0, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

if(utime(dst\_check, &timebuf) < 0) {

fprintf(stderr, "utime error for dst\_check\n");

return;

}

unlink(orifile\_rename);

write\_log(src\_fname, &src\_statbuf);

return;

}

}

if(checksum == 0) { //이름이 일치하는 파일이 없는 경우

memset(dst\_check, 0, PATHMAX);

strcpy(dst\_check, dst);

strcat(dst\_check, "/");

strcat(dst\_check, src\_fname);

//dst 디렉토리에 만들 파일 경로

if((fd\_sync = creat(dst\_check, 0644)) < 0) {

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

return;

}

if((fd\_src = open(src, O\_RDONLY)) < 0) {

fprintf(stderr, "open error for src");

return;

}

memset(buf, 0, PATHMAX);

while((readcnt = read(fd\_src, buf, PATHMAX)) > 0) {

write(fd\_sync, buf, readcnt);

memset(buf, 0, PATHMAX);

} //동기화 대상 파일을 읽어서 새로 만든 파일에 씀

close(fd\_sync);

close(fd\_src);

timebuf.actime = src\_statbuf.st\_atime;

timebuf.modtime = src\_statbuf.st\_mtime;

if(utime(dst\_check, &timebuf) < 0) {

fprintf(stderr, "utime error for dst\_check\n");

return;

}

write\_log(src\_fname, &src\_statbuf);

}

return;

}

}

void write\_log(char \*fname, struct stat \*statbuf) {

fprintf(fp, " %s %ldbytes\n", fname, statbuf->st\_size);

return;

} //로그를 작성하는 함수

int dir\_check(const struct dirent \*info) {

if(!strncmp(info->d\_name, ".", 1))

return 0;

else

return 1;

} //scandir 필터함수

void print\_usage(char \*str) {

printf("Usage : %s <option> <src file or directory> <dst directory>\n", str);

printf(" <option>\n");

printf(" -r : Make sync <src>'s sub directory\n");

printf(" -t : Make sync with .tar file\n");

printf(" -m : Delete <dst>'s files that not exist in <src>\n");

return;

} //사용법 출력 함수

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

{ //프로그램 수행시간 측정

end\_t->tv\_sec -= begin\_t->tv\_sec; //끝난 시간에서 처음시간을 뺌

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

end\_t->tv\_sec--;

end\_t->tv\_usec += 1000000;

} //나중의 us값이 작을경우 1초를 뺴서 us값이 더해줌

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

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

} //프로그램 총 소요시간 계산 후 출력하는 함수