컴퓨터학부 20160548 이승현

1. 과제 개요

설계과제3은 3가지 프로그램으로 구성 되어있다. ssu_crontab은 사용자가 주기적으로 실행하고자 하는 명령어를 등록하는 프로그램이다. ssu_crond은 ssu_crontab을 이용하여 사용자가 등록한 명령어를 주기적으로 확인하고, 실행하는 디몬 프로세스이다. ssu_rsync는 원하는 파일이나 디렉토리를 동기화하는 프로그램이다.

ssu_crontab에서는 add, remove, exit 세가지 명령어를 사용할 수 있다. add는 새로운 명령어를 사용할 때, remove는 기존에 등록되어있던 명령어를 삭제할 때, exit 명령어는 프로그램을 종료할 때 사용한다. 이 프로그램을 이용해 추가한 명령어는 ssu_crontab_file 파일에 저장되고, 명령어 추가/삭제 내역은 ssu_crontab_log 파일에 기록된다.

ssu_crond 프로그램은 백그라운드에서 실행되며, 주기적으로 ssu_crontab_file에 저장된 명령어들을 확인해, 실행해야 하는 명령어들을 실행시킨다. 실행된 명령어는 ssu_crontab_log 파일에 기록한다.

ssu_rsync 프로그램은 특정 파일이나 디렉토리를 지정하여 동기화 하는 프로그램이다. -r, -m, -t 옵션을 사용할 수 있으며, 한번 실행할 때 하나의 옵션만 사용할 수 있도록 구현하였다. -r 옵션은 동기화 대상으로 지정한 파일이 디렉토리일 때 해당 디렉토리의 하위 디렉토리까지 모두 동기화 한다. -m 옵션은 dst 디렉토리에 src 디렉토리에 없는 파일 및 디렉토리가 존재할 경우 dst 디렉토리에서 해당하는 파일 및 디렉토리를 삭제하는 옵션이다. -t 옵션은 tar을 활용하여 동기화가 필요한 대상들을 묶어 한번에 동기화 작업을 수행하는 옵션이다.

이 과제에서 구현해야 하는 항목은 다음과 같다.

가. ssu_crontab 30

나. ssu_crand 30

다. ssu_rsync 30

라. -r 3

마. -t 4

바. -m 3

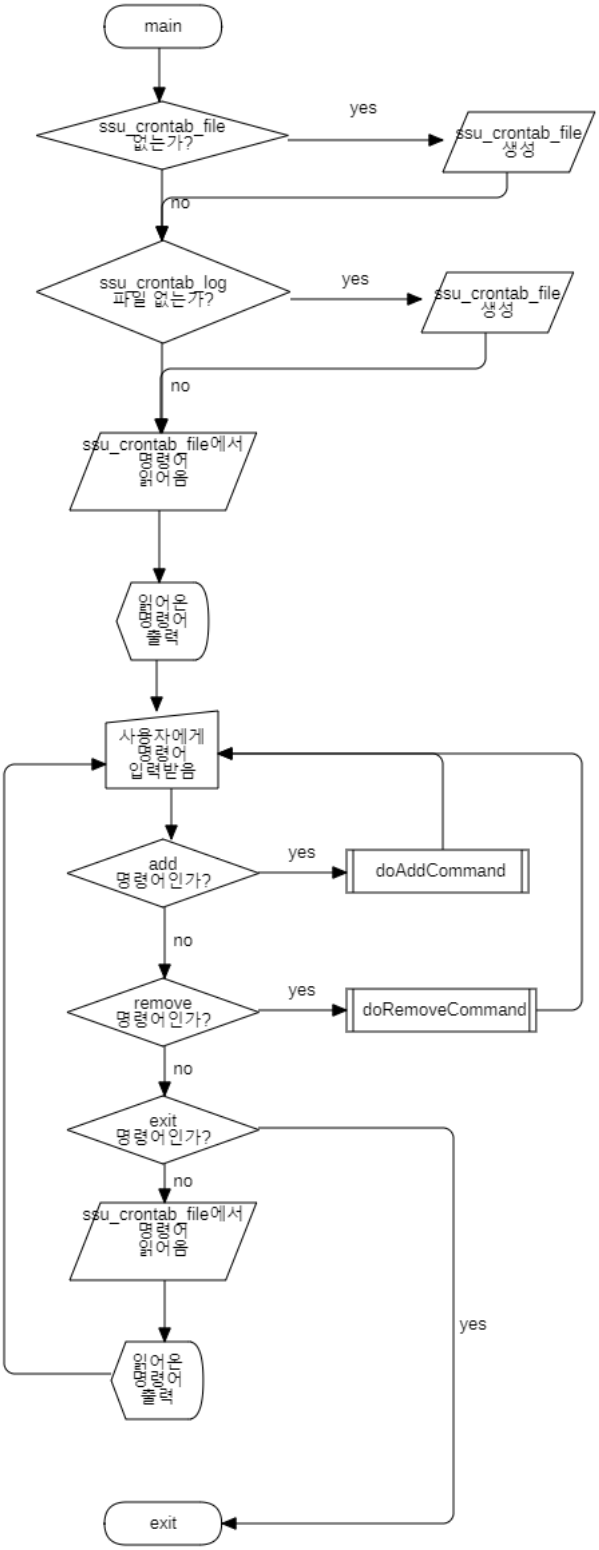
사. gettimeofday()를 사용하여 프로그램의 수행 시간 측정

위 기능들을 모두 구현하였다.

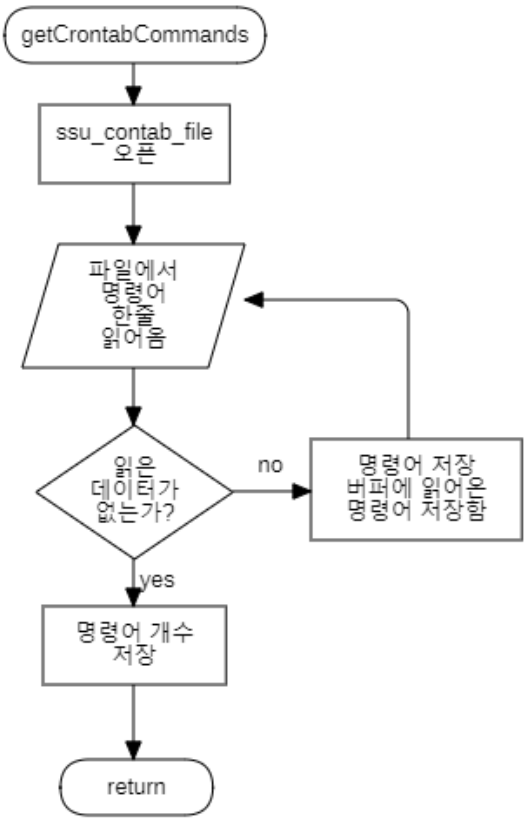
2. 설계

<ssu_crontab.c>

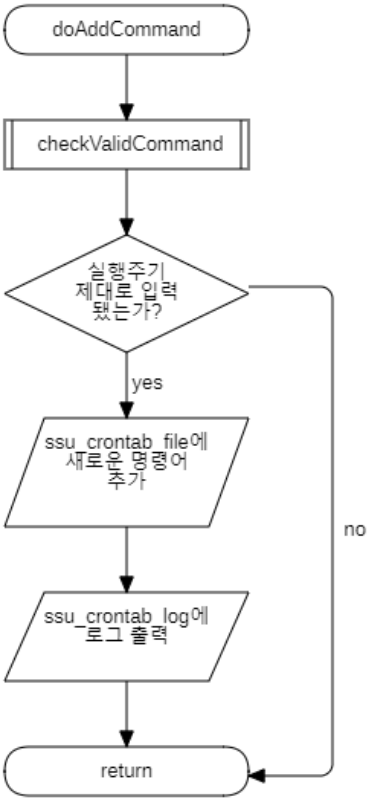
int mian();



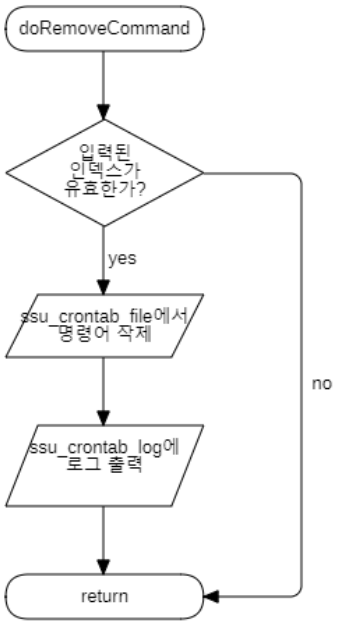
void getCrontabCommands(void); // 파일에 저장된 명령어들을 읽어오는 함수



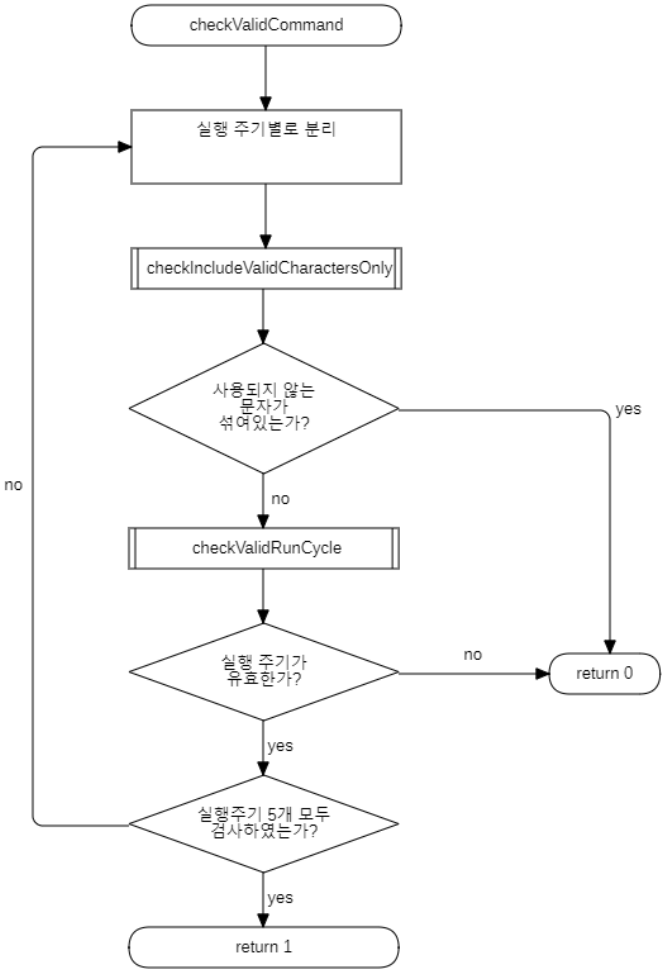
void doAddCommand(char *input_command); // 명령어를 추가하는 함수



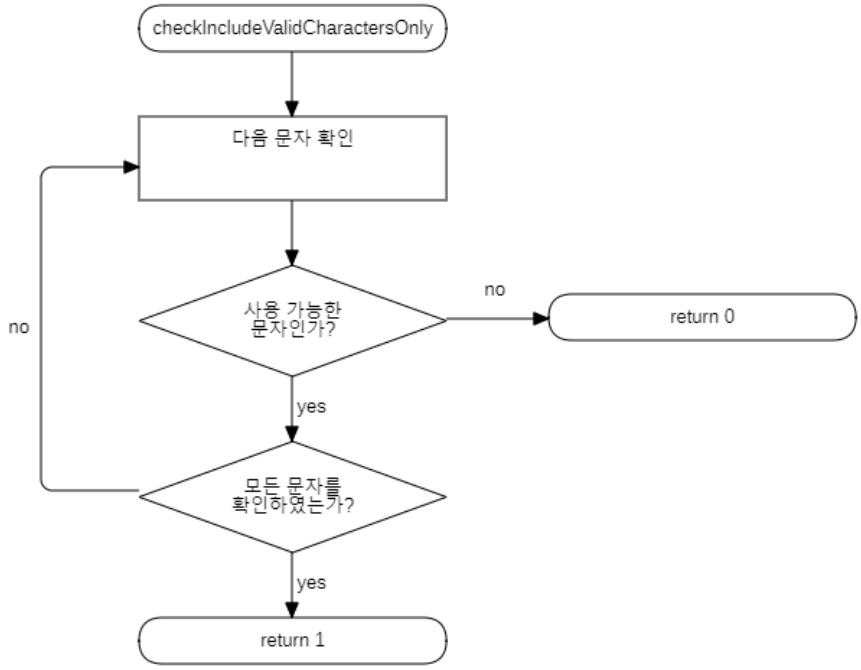
void doRemoveCommand(void); // 명령어를 삭제하는 함수



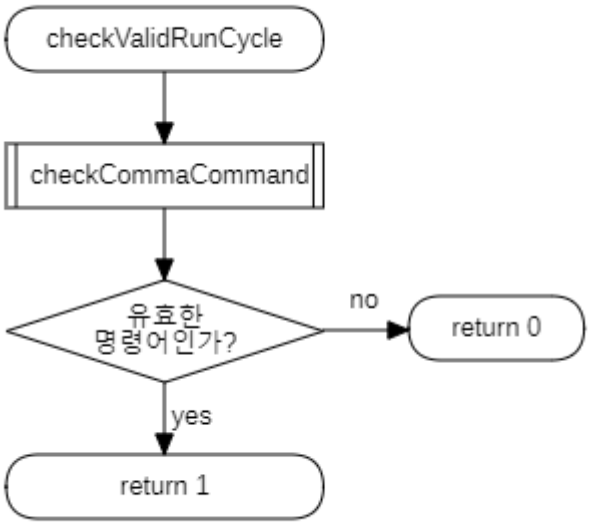
int checkValidCommand(const char *input_command); // 유효한 명령어인지 확인하는 함수



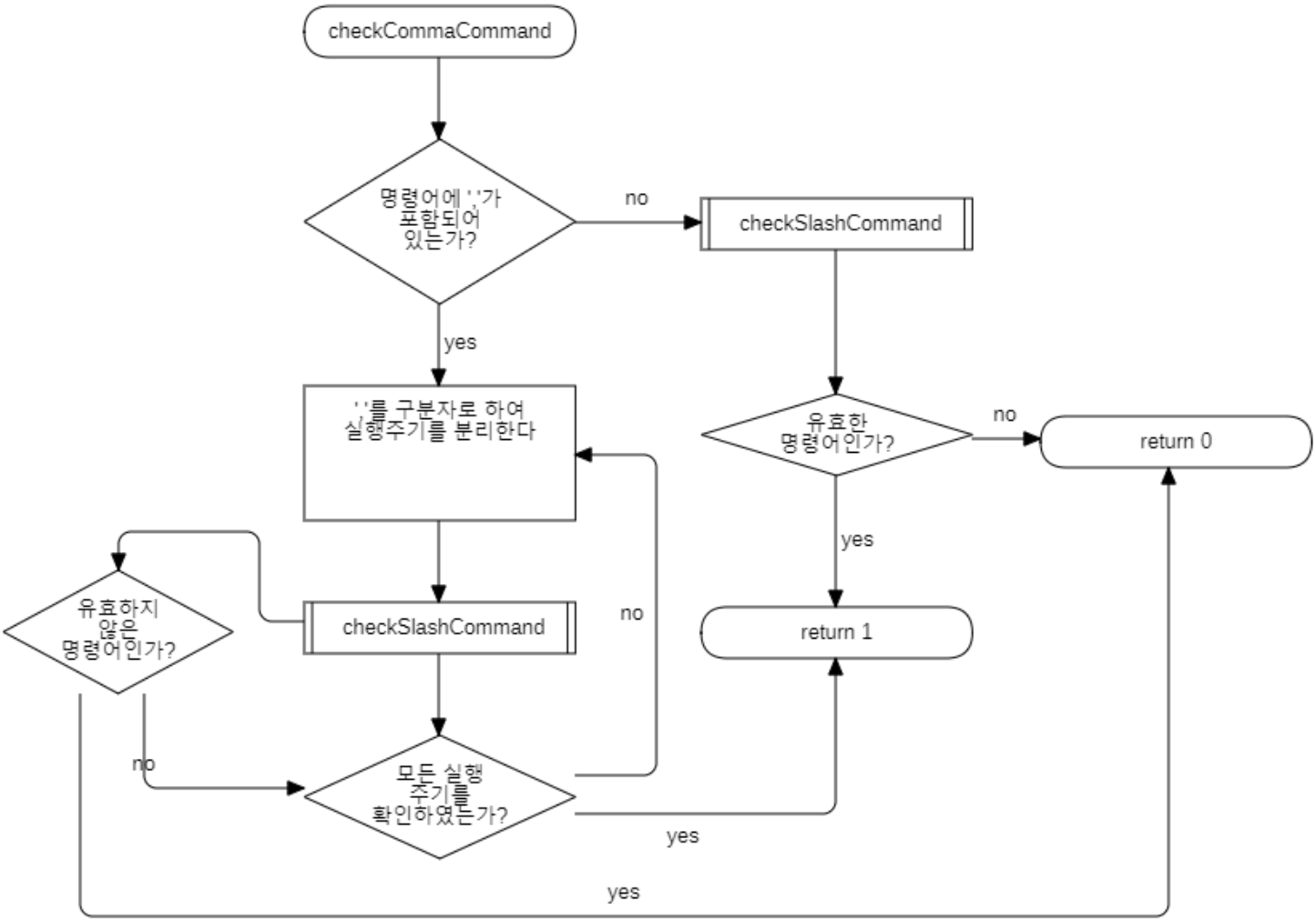
int checkIncludeValidCharactersOnly(const char *lexeme); // 사용할 수 없는 문자가 명령어에 포함되어있는지 확인하는 함수



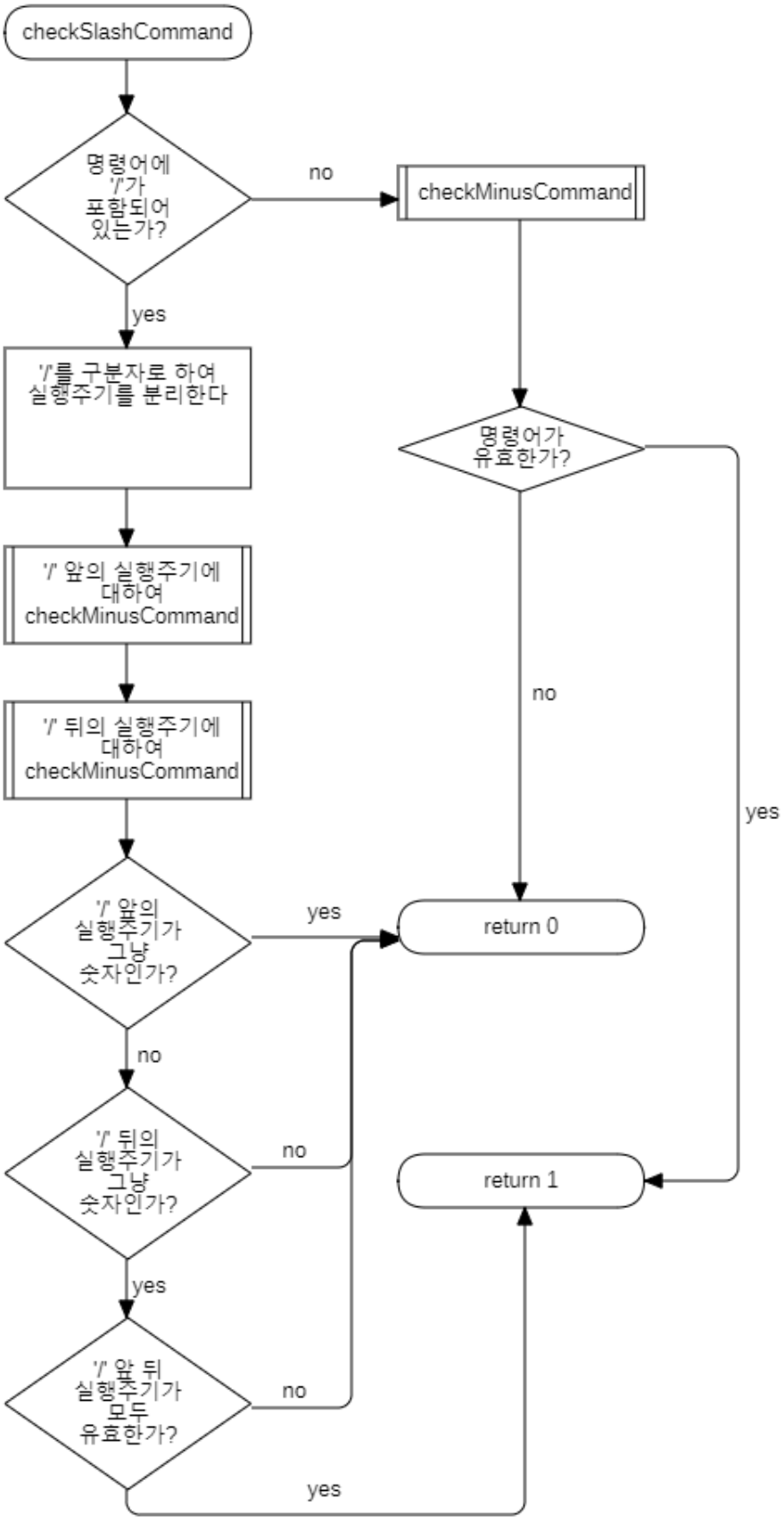
int checkValidRunCycle(char *lexeme, int run_cycle_index); // 실행 주기가 유효한 명령어인지 확인하는 함수



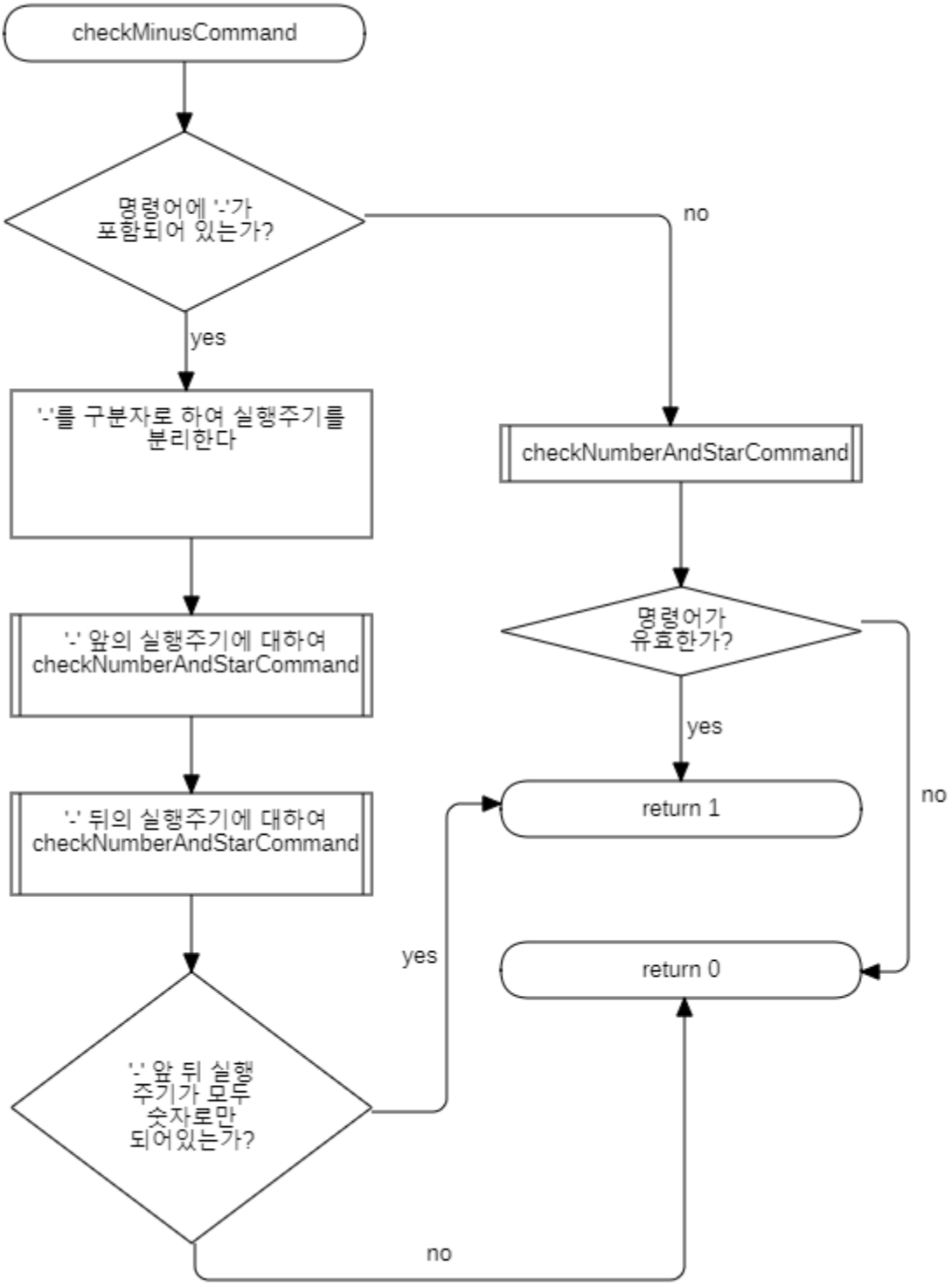
int checkCommaCommand(char *lexeme, int run_cycle_index); // 실행 주기 문자 ','가 제대로 쓰였는지 확인하는 함수



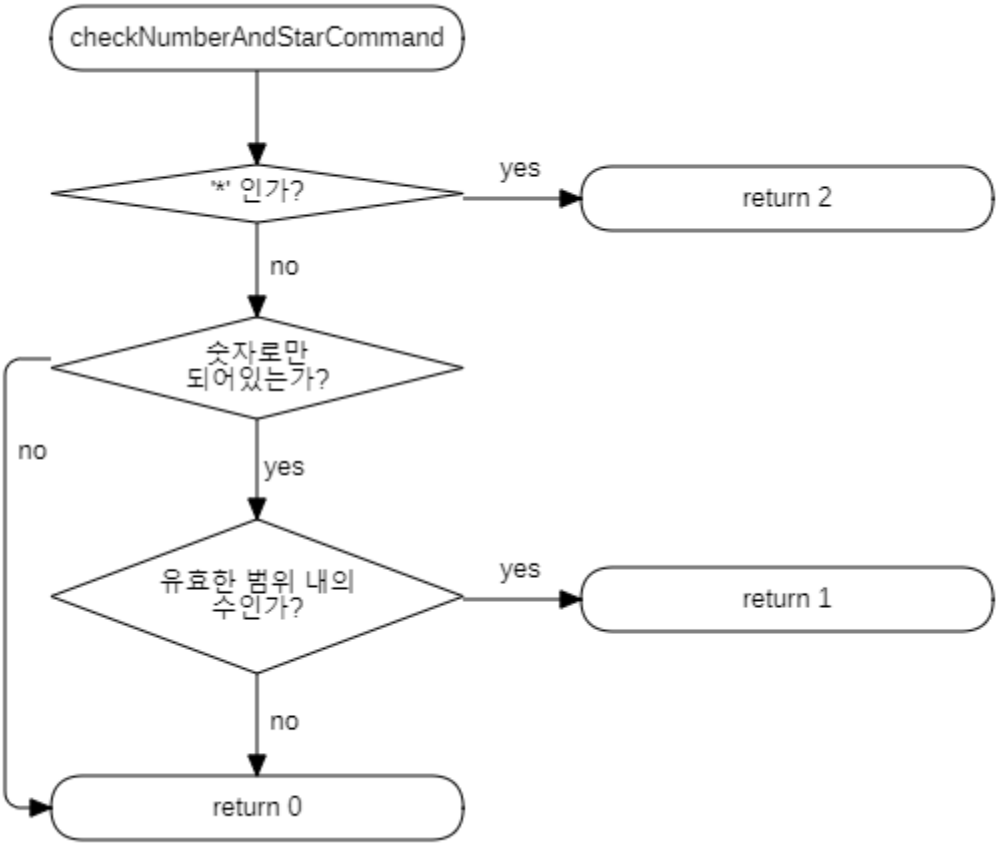
int checkSlashCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '/'가 제대로 쓰였는지 확인하는 함수



int checkMinusCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '-'가 제대로 쓰였는지 확인하는 함수

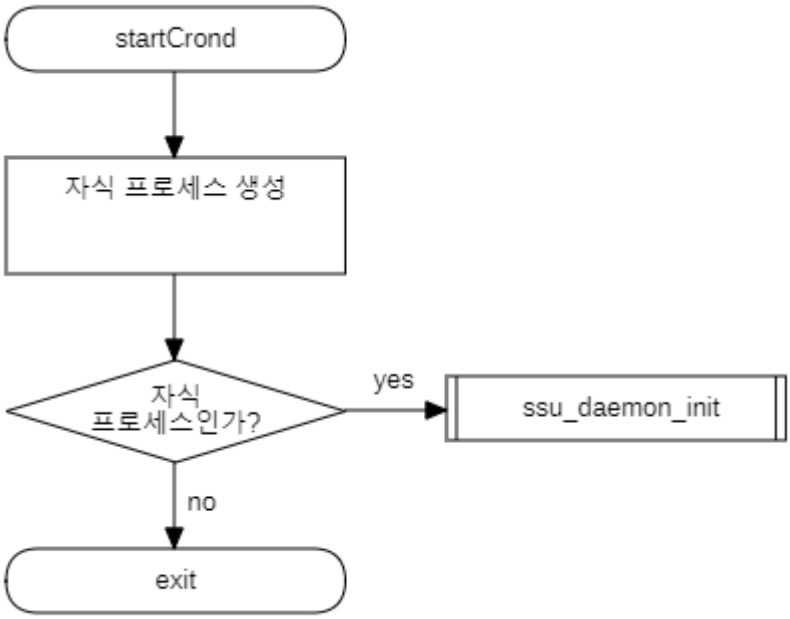


int checkNumberAndStarCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '*'와 숫자가 제대로 쓰였는지 확인하는 함수

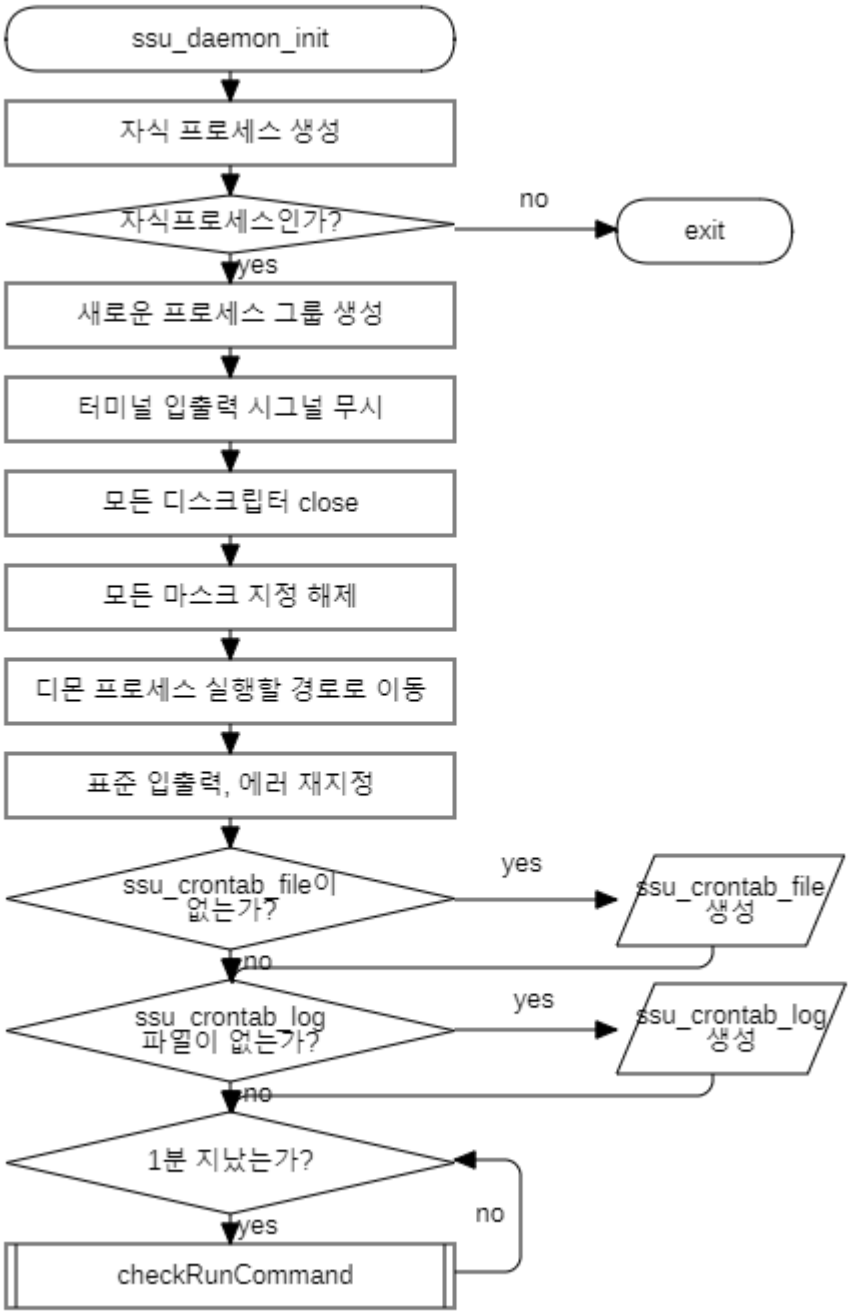


<ssu_crond.c>

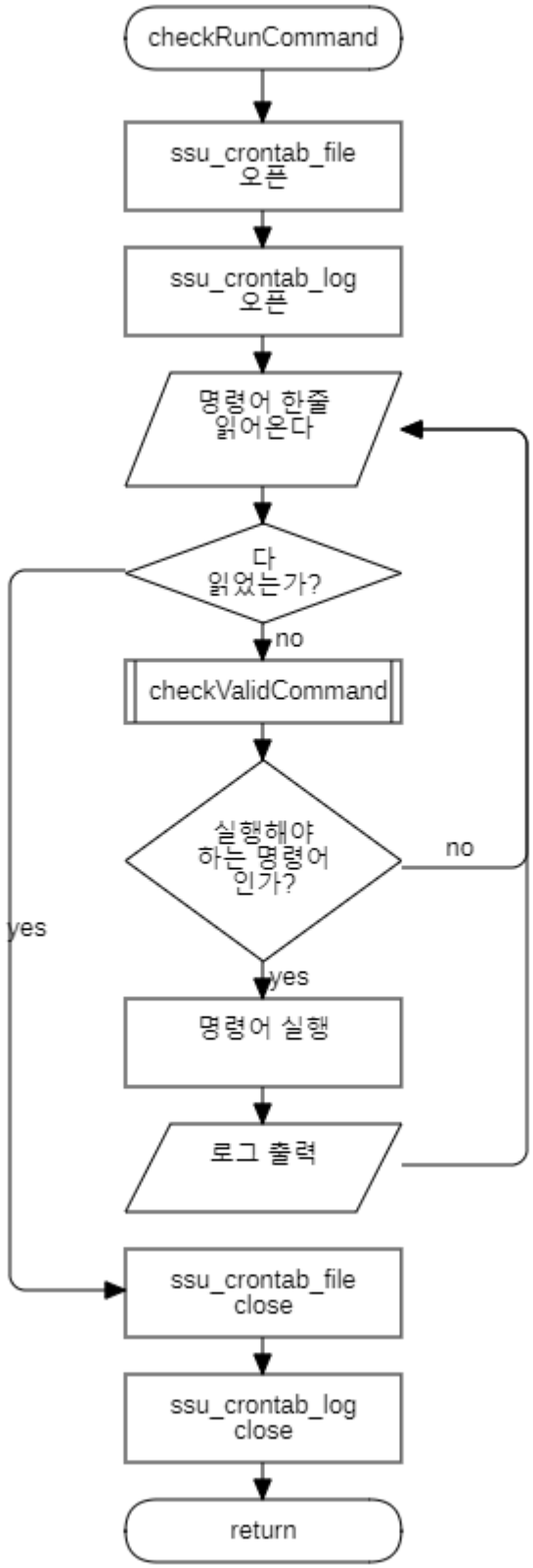
void startCrond(); // ssu_crond 실행시키는 함수



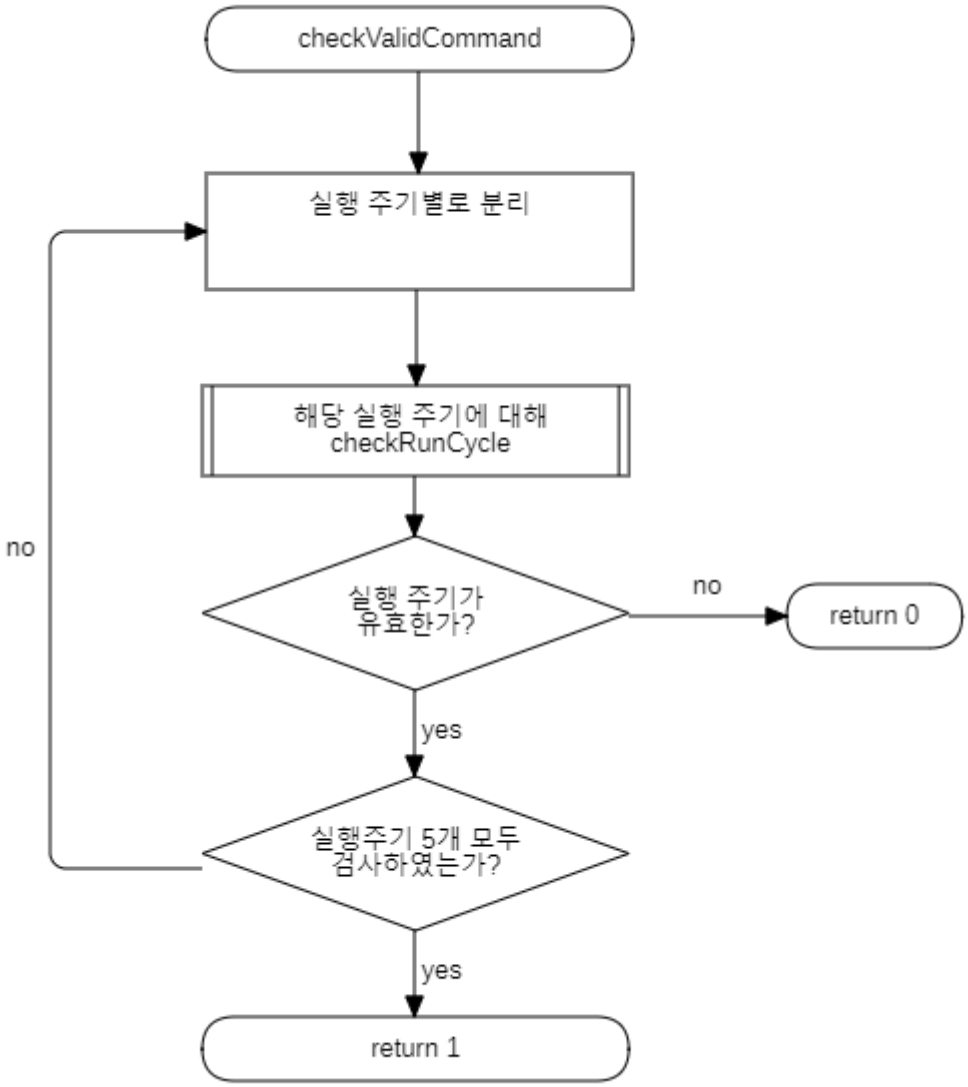
int ssu_daemon_init(const char *path); // 디몬 프로세스 생성하는 함수



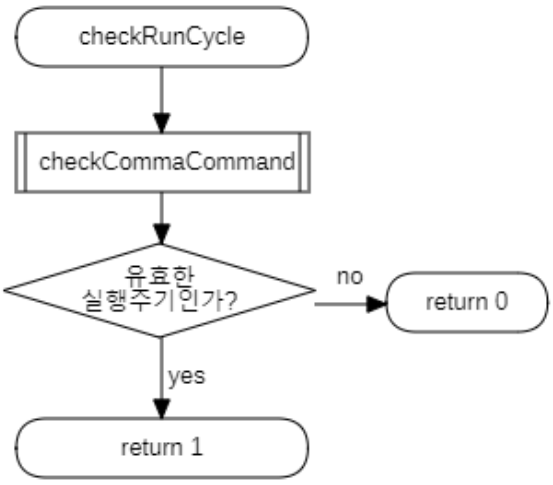
void checkRunCommand(); // 실행시킬 명령어 있는지 확인하는 함수



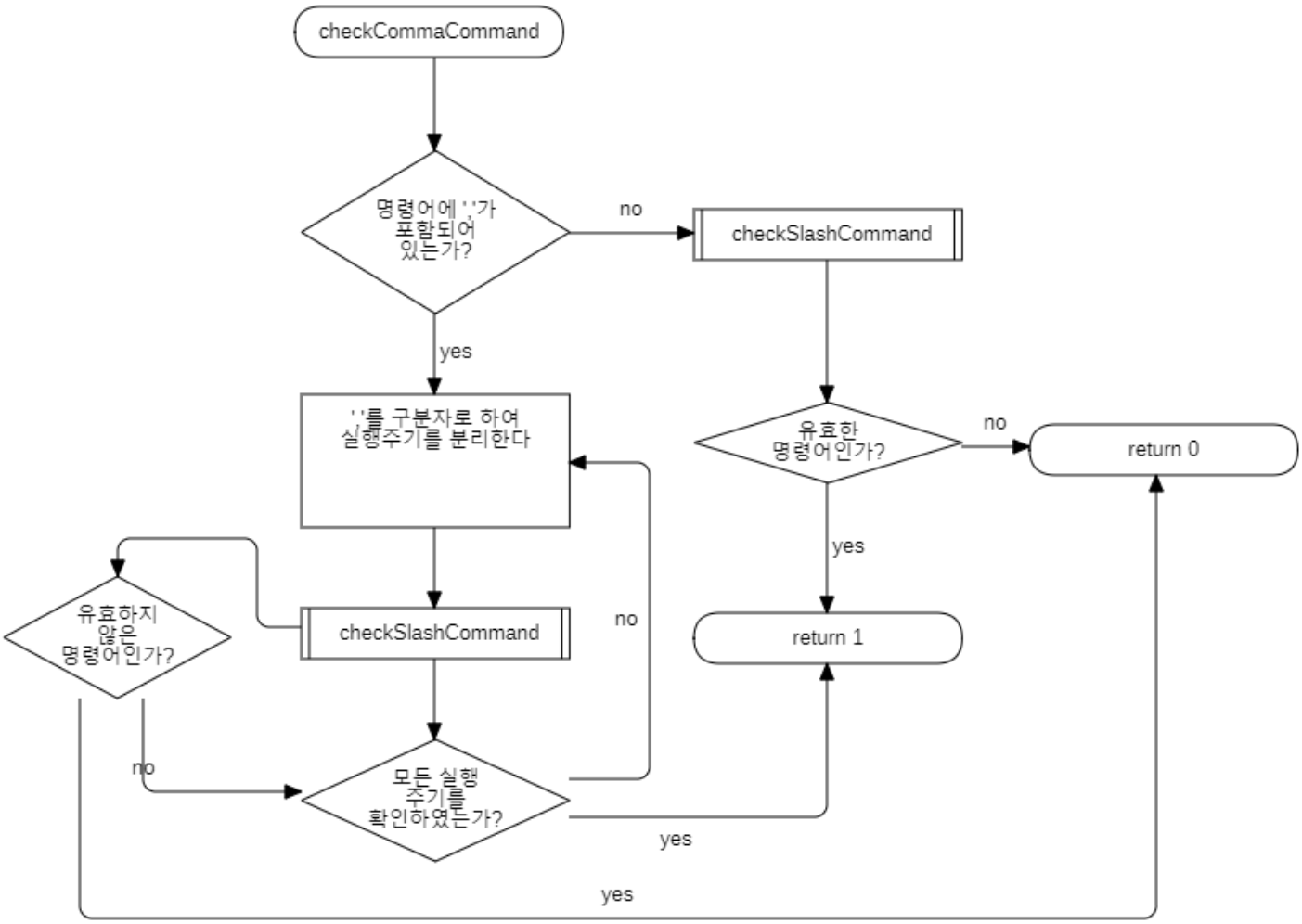
int checkValidCommand(const char *input_command); // 실행 주기를 확인해 실행시킬 시간인지 확인하는 함수



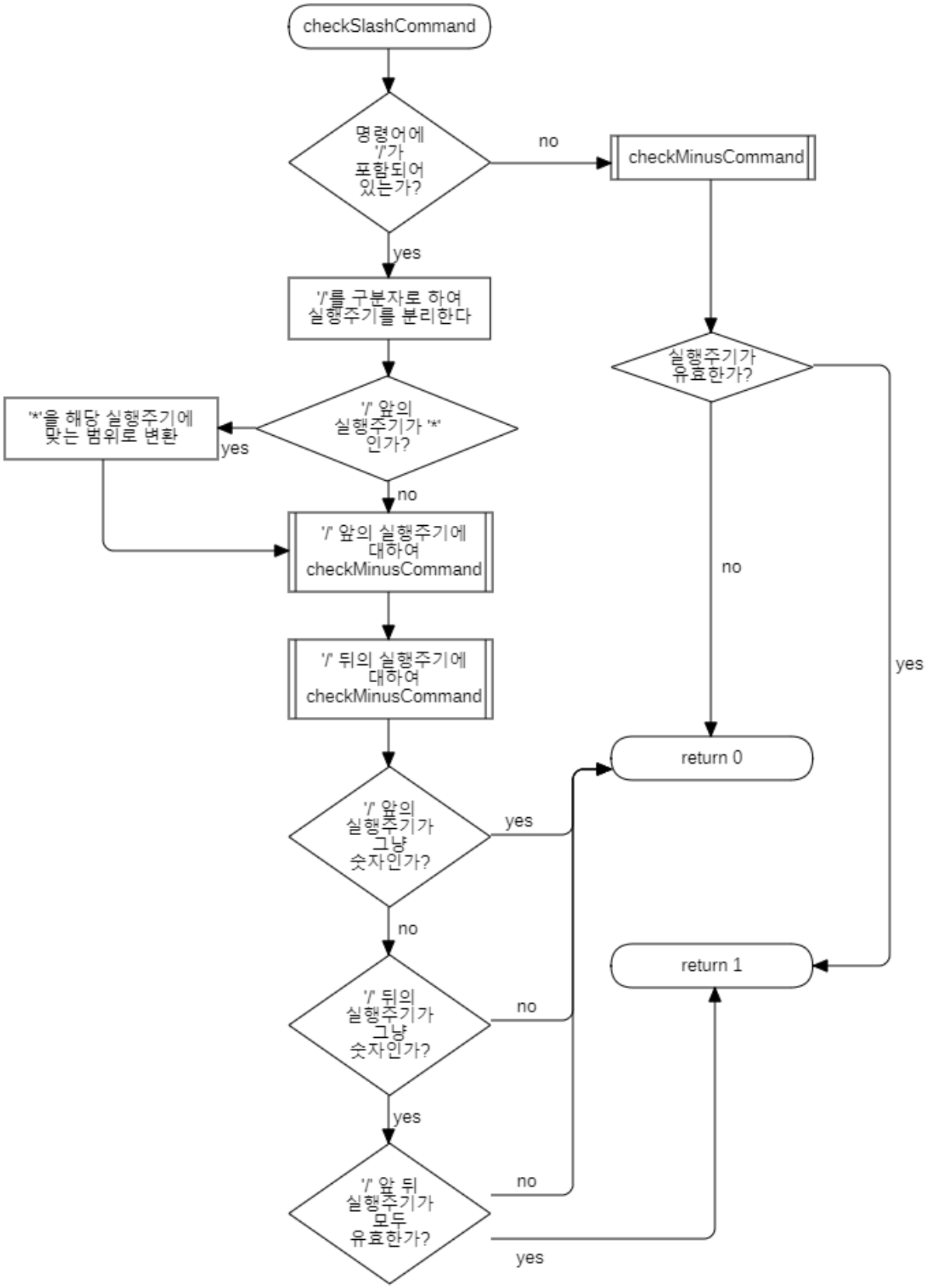
int checkRunCycle(char *run_cycle, int run_cycle_index, int current_time); // 실행주기 확인하는 함수



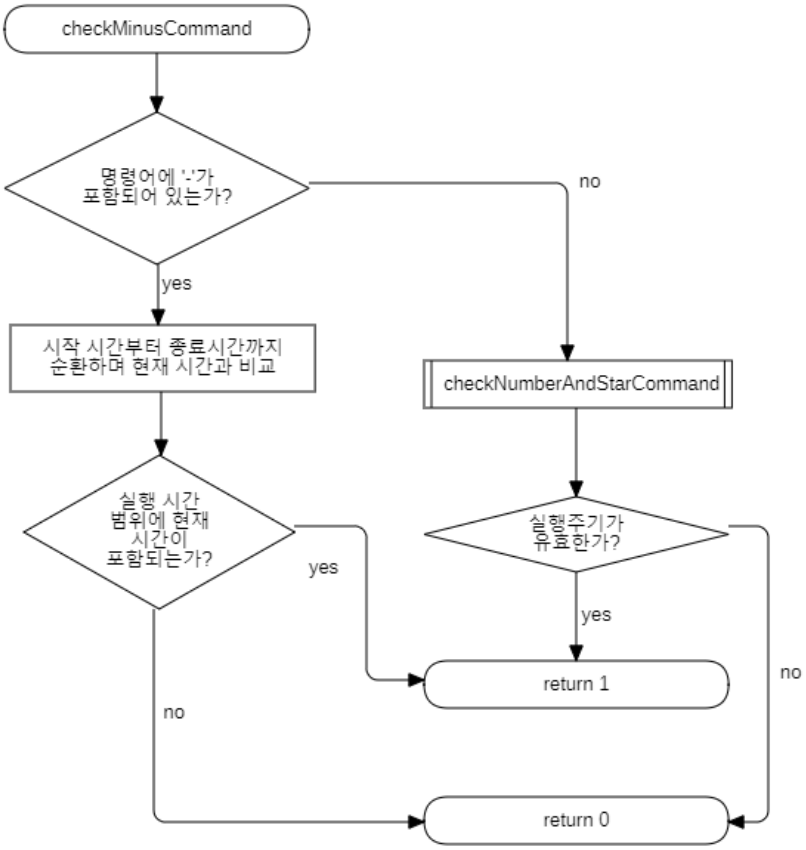
int checkCommaCommand(char *lexeme, int run_cycle_index, int current_time); //',' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수



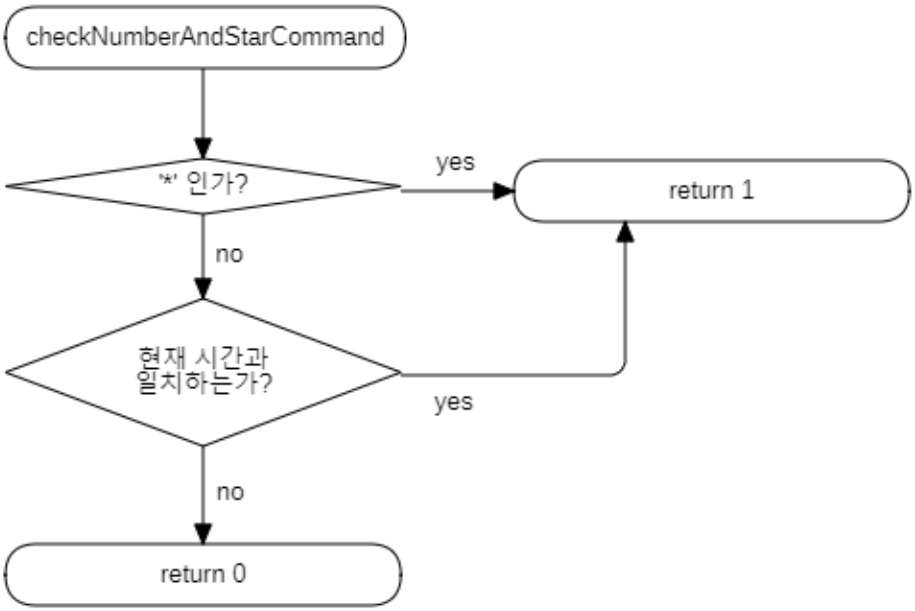
int checkSlashCommand(char *lexeme, int run_cycle_index, int current_time); //'/' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수



int checkMinusCommand(char *lexeme, int run_cycle_index, int current_time, int increase); //'-' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

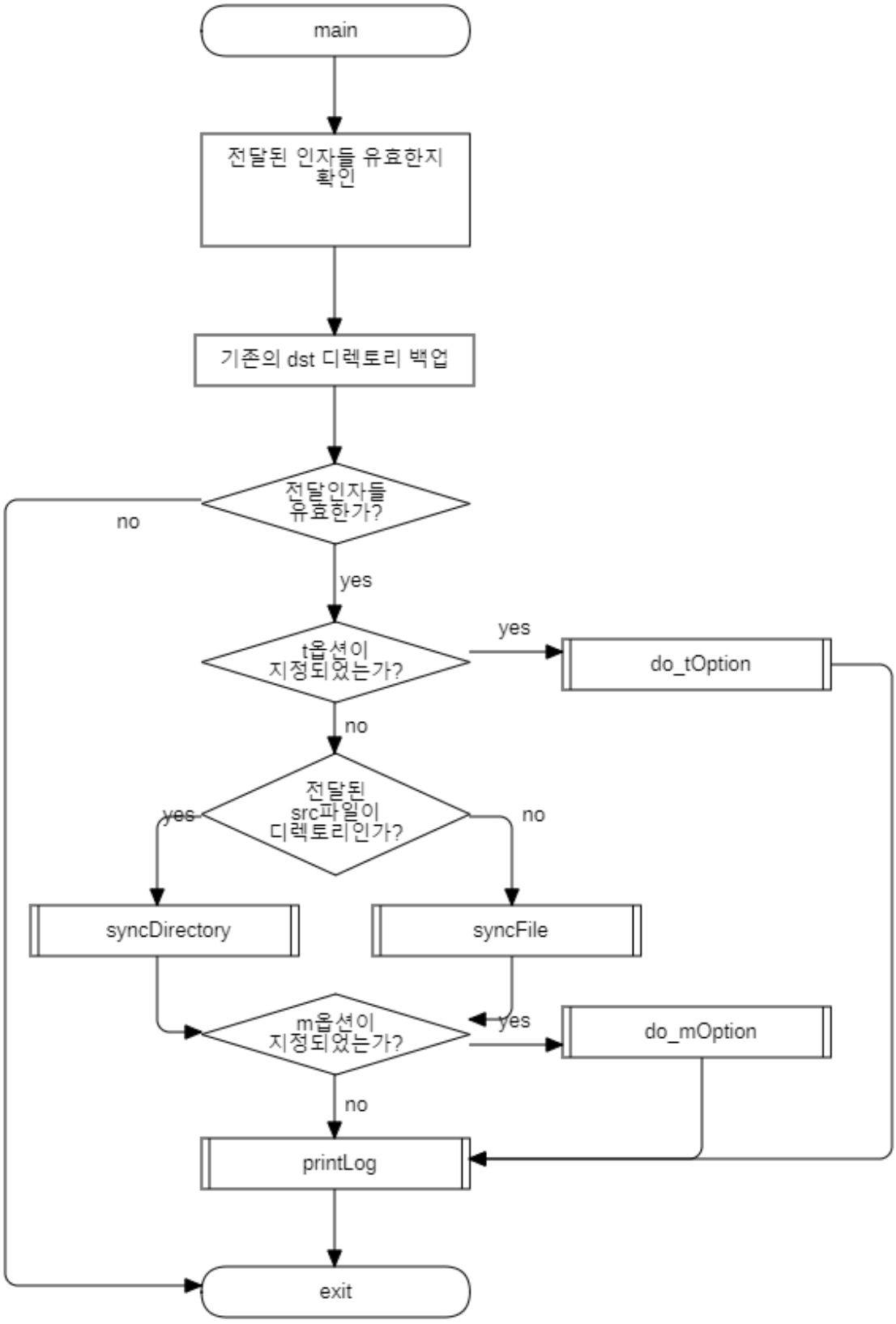


int checkNumberAndStarCommand(char *lexeme, int current_time); //'*', 숫자를 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

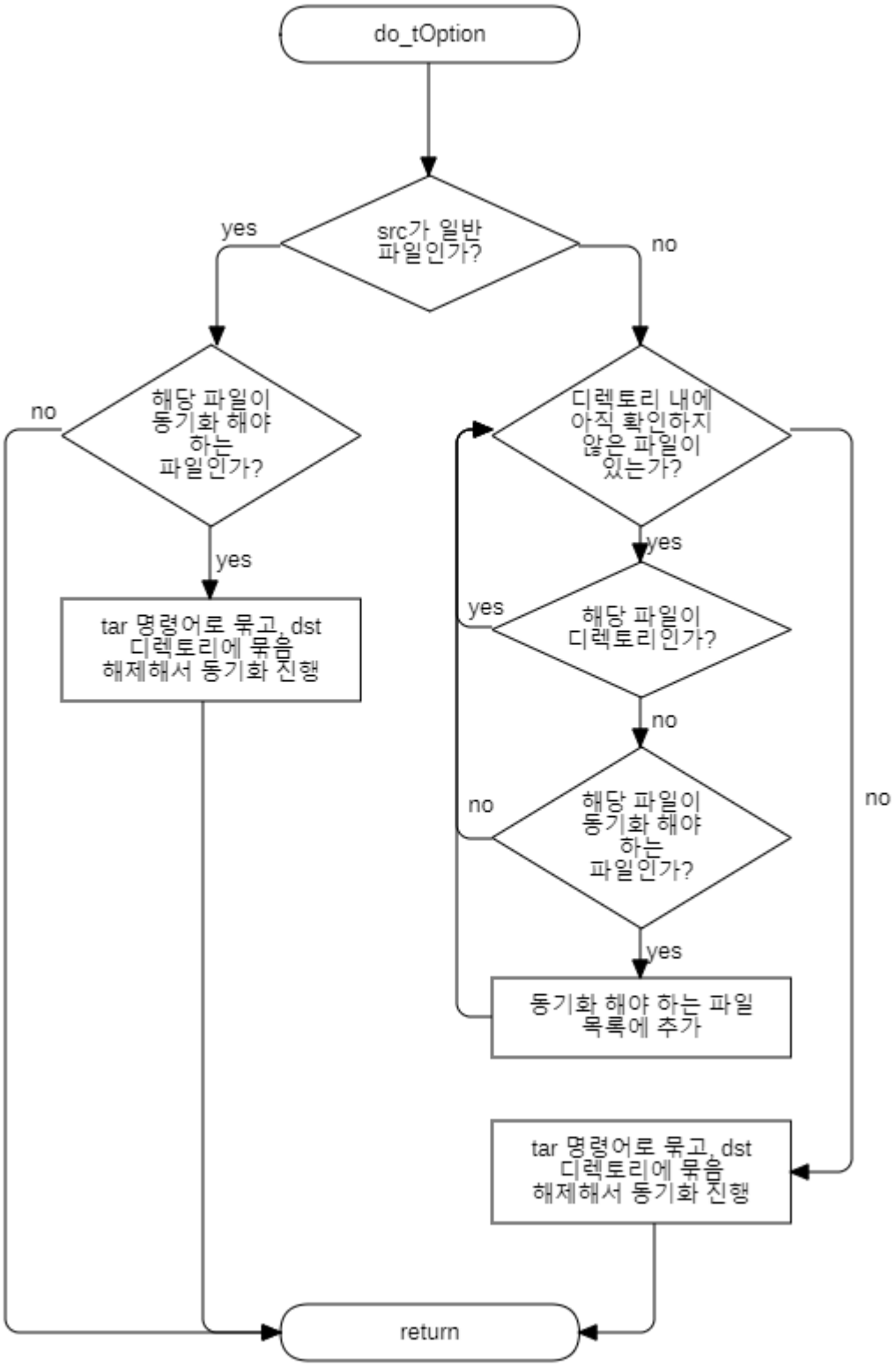


<ssu_rsync.c>

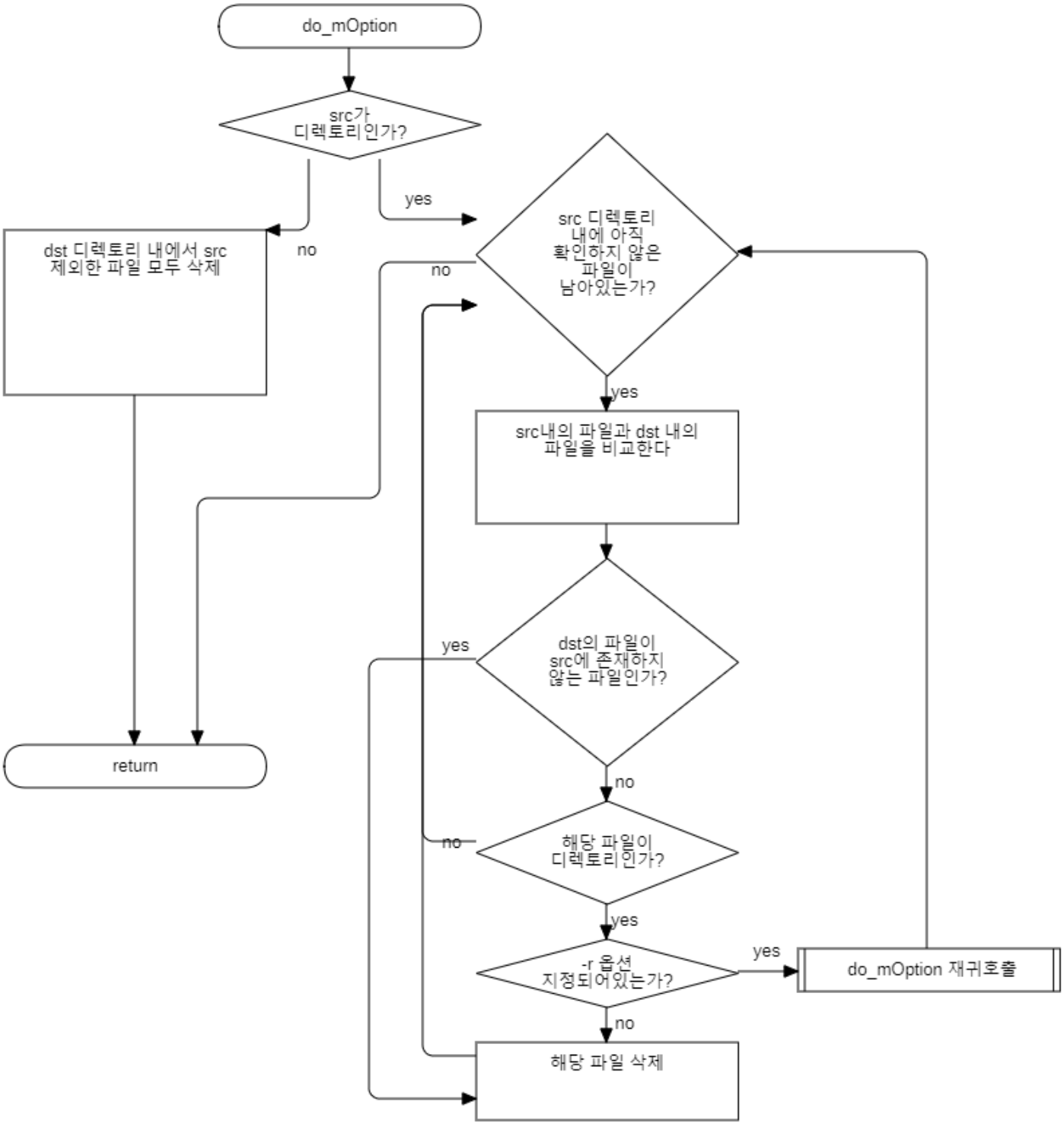
int main(int argc, char *argv[]);



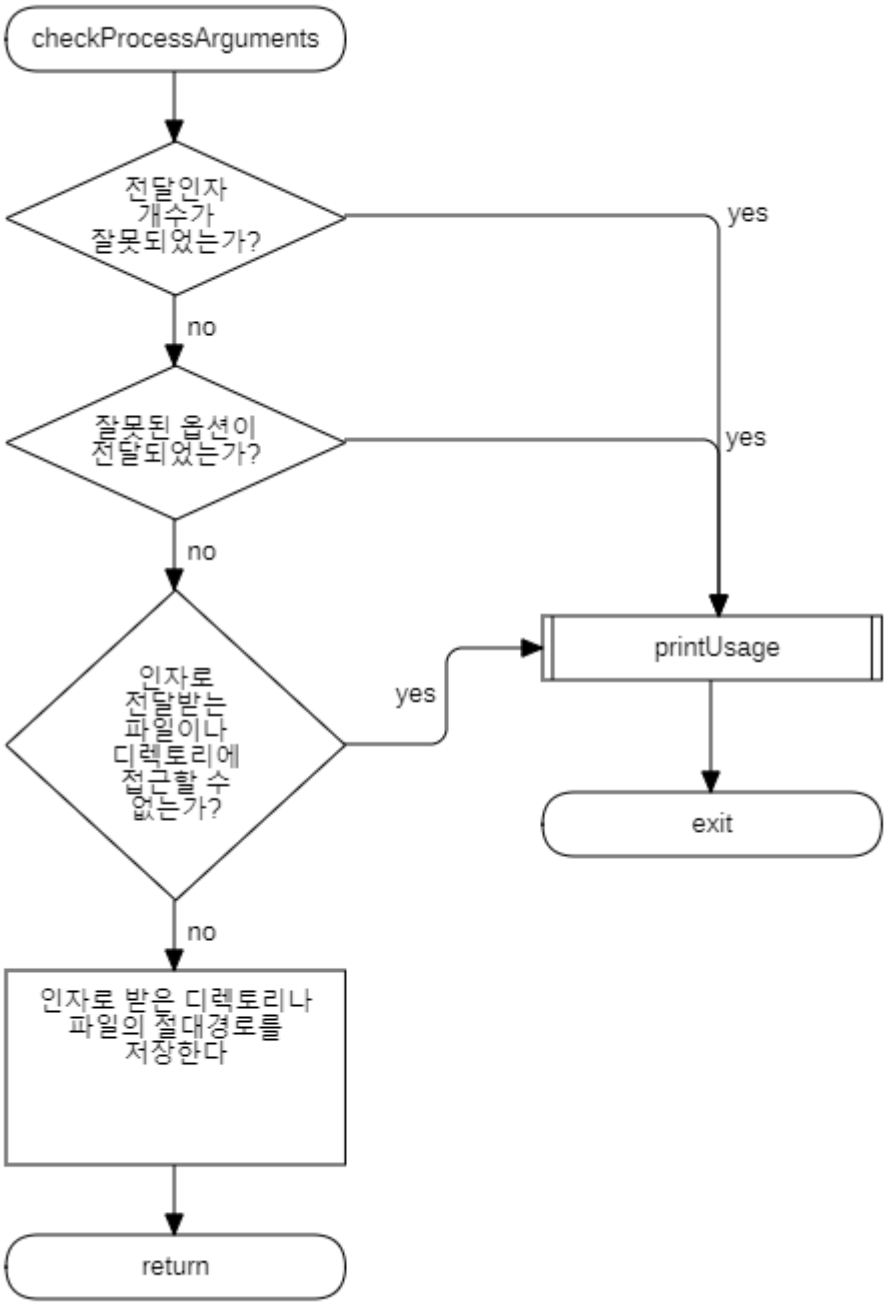
void do_tOption(const char *src_path_name, const char *dst_path_name); // -t 옵션 수행하는 함수



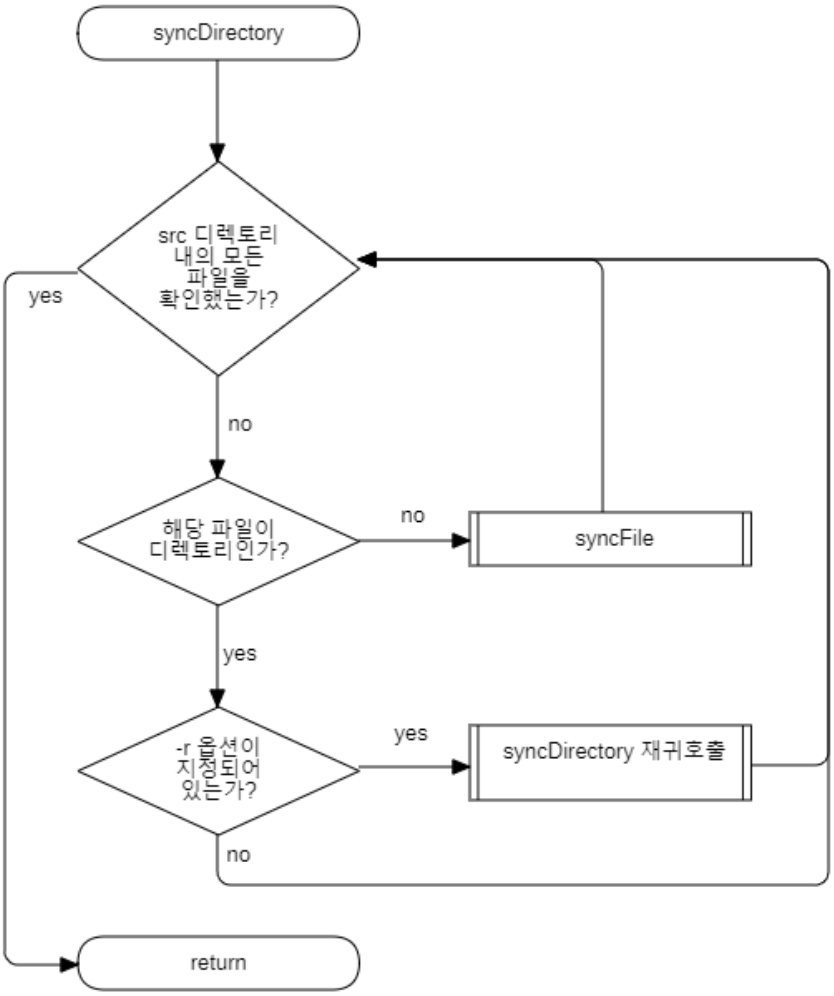
void do_mOption(FILE *tmp_log_fp, const char *path_name, const char *src_path_name, const char *dst_path_name, int sync_dir_flag); // -m 옵션 수행하는 함수



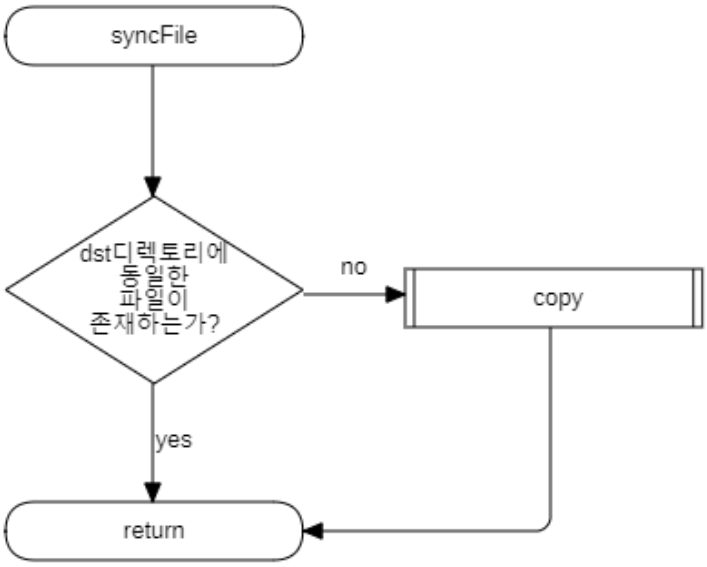
void checkProcessArguments(int argc, char *argv[]); // 프로세스에 전달된 인자들이 유효한지 확인하는 함수



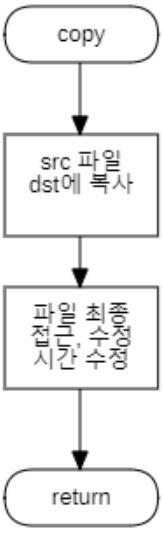
void syncDirectory(const char *src_path_name, const char *dst_path_name, int sync_dir_flag); // 디렉토리를 동기화하는 함수



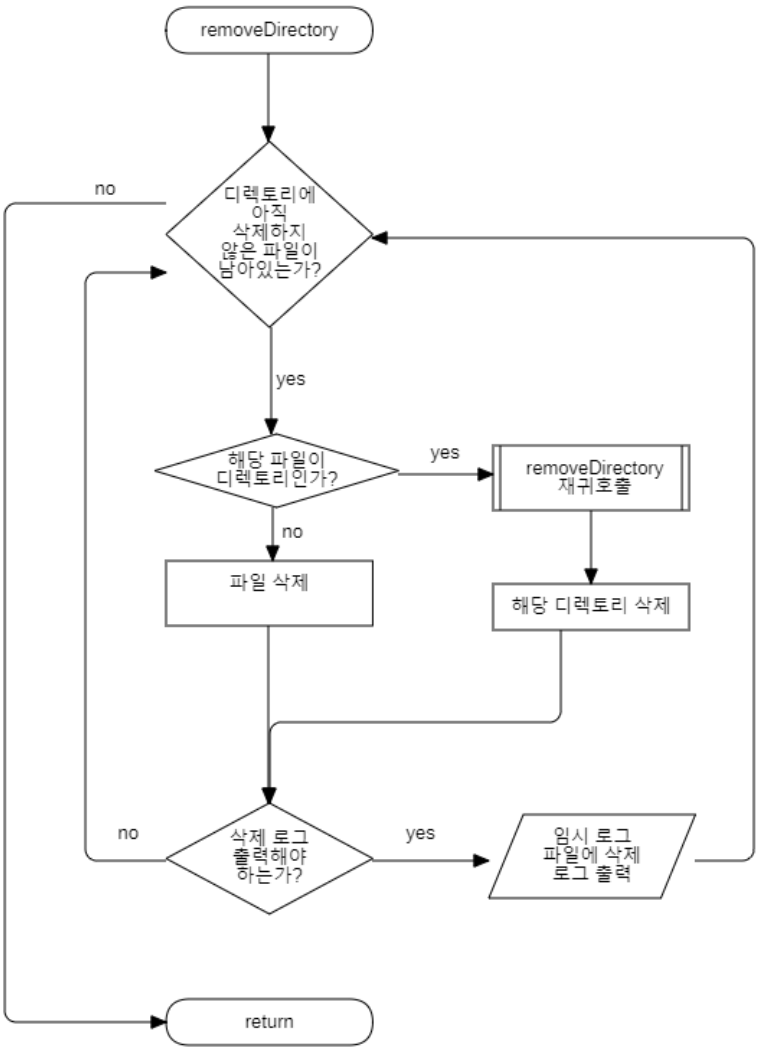
void syncFile(const char *src_file_name, const char *dst_path_name); // 파일을 동기화하는 함수



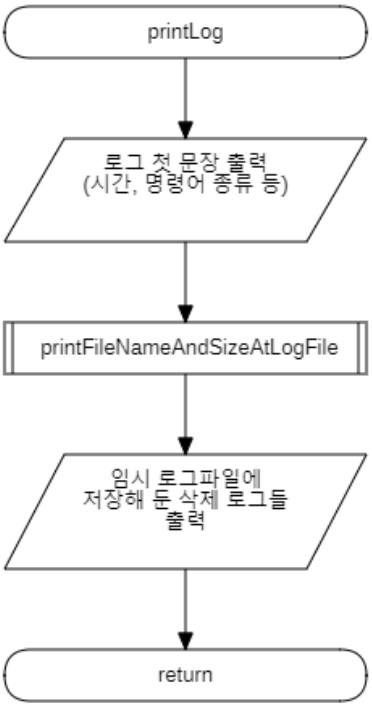
void copy(const char *src, const char *dst); // 파일을 복사하는 함수



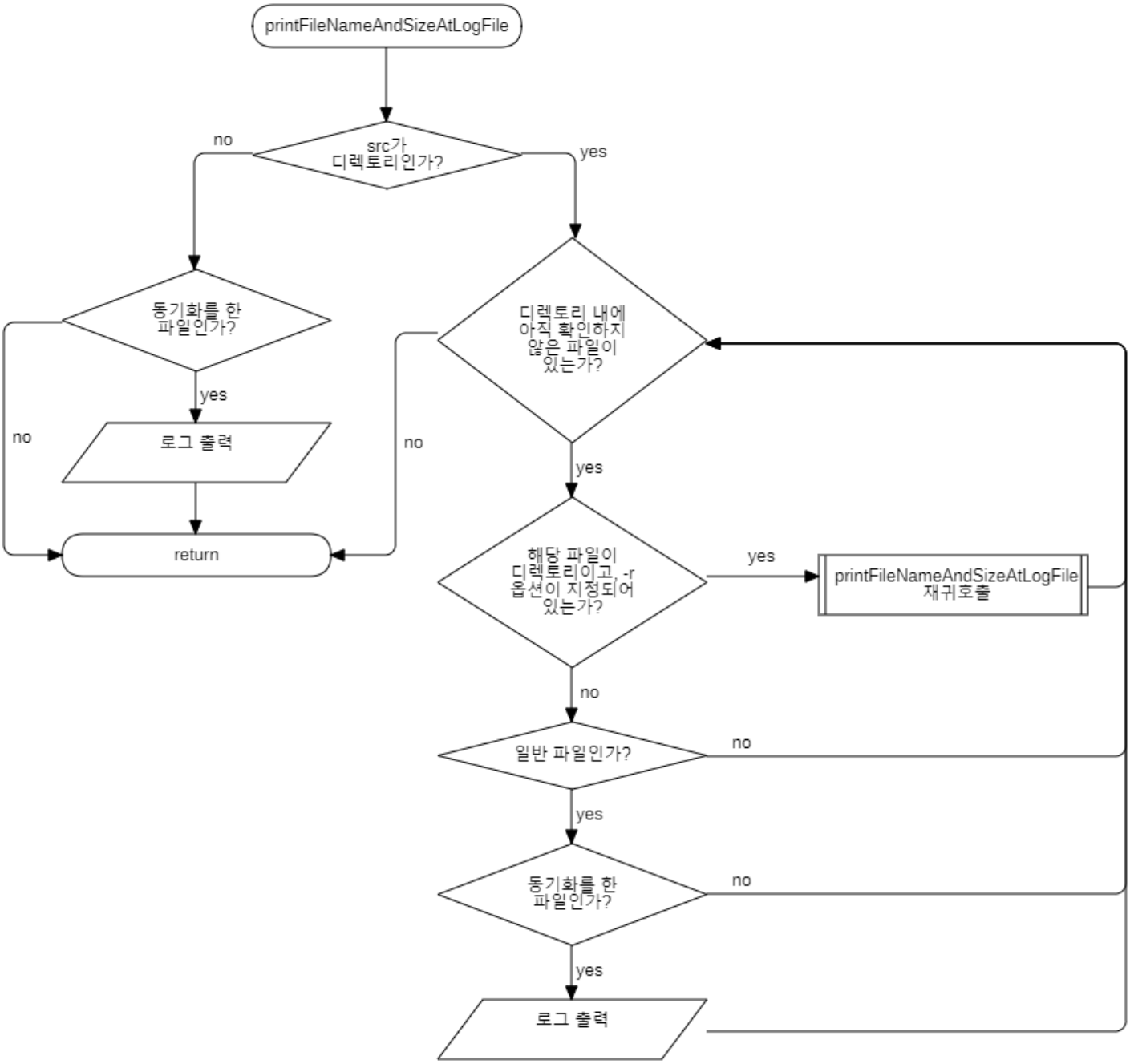
void removeDirectory(FILE *tmp_log_fp, const char *path_name, const char *target); // 디렉토리와 그 디렉토리 하위의 모든 파일을 삭제하는 함수



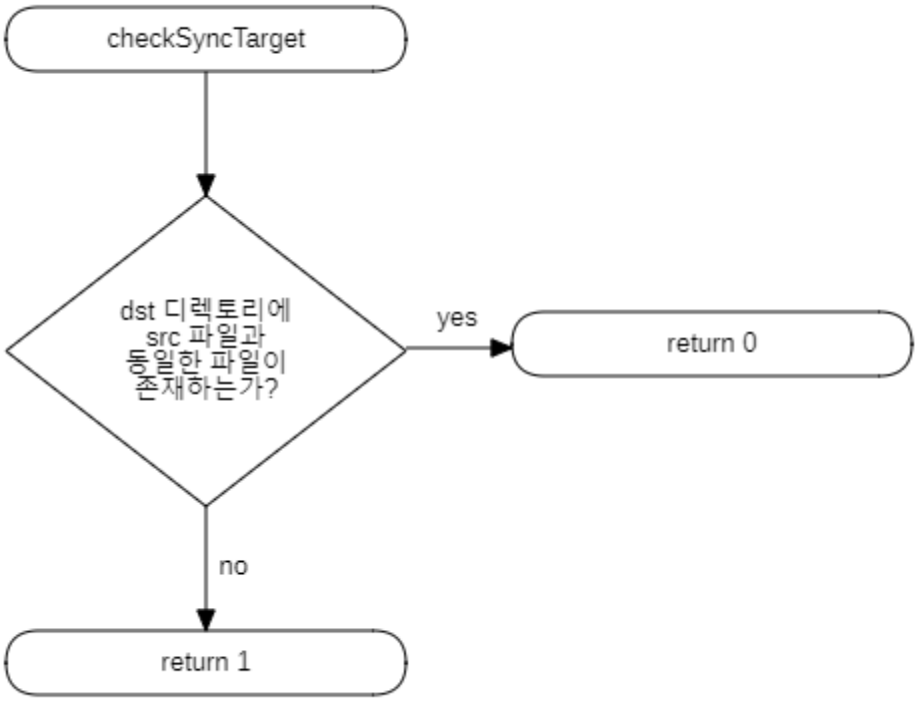
void printLog(FILE *tmp_log_fp, const char *src_path_name, int sync_dir_flag); // 로그를 출력하는 함수



void printFileNameAndSizeAtLogFile(FILE * fp, const char *src_path_name, const char *path_name, int sync_dir_flag); // 동기화된 파일의 이름과 사이즈를 로그파일에 출력하는 함수



int checkSyncTarget(const char *src_file_name, const char *dst_path_name); // 동기화 대상 파일인지 확인하는 함수



static void sigint_during_sync_handler(int signo);

3. 구현

<ssu_crontab.c>

void printCrontabCommands(void); // 저장된 명령어들을 읽어와 프롬프트에 출력하는 함수

전달인자 : 없음

리턴 값 : 없음

void getCrontabCommands(void); // 파일에 저장된 명령어들을 읽어오는 함수

전달인자 : 없음

리턴 값 : 없음

void doAddCommand(char *input_command); // 명령어를 추가하는 함수

전달인자 : char *input_command // 사용자가 입력한 명령어

리턴 값 : 없음

void doRemoveCommand(void); // 명령어를 삭제하는 함수

전달인자 : 없음

리턴 값 : 없음

int checkValidCommand(const char *input_command); // 유효한 명령어인지 확인하는 함수

전달인자 : const char *input_command // 사용자가 입력한 명령어

리턴 값 : int // 유효하면 1, 유효하지 않으면 0

int checkIncludeValidCharactersOnly(const char *lexeme); // 사용할 수 없는 문자가 명령어에 포함되어있는지 확인하는 함수

전달인자 : const char *lexeme // 유효한지 확인할 명령어

리턴 값 : int // 유효하면 1, 유효하지 않으면 0

int checkValidRunCycle(char *lexeme, int run_cycle_index); // 실행 주기가 유효한 명령어인지 확인하는 함수

전달인자 :

- char *lexeme // 실행 주기가 유효한지 확인할 명령어

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

리턴 값 : int // 유효하면 1, 유효하지 않으면 0

int checkCommaCommand(char *lexeme, int run_cycle_index); // 실행 주기 문자 ','가 제대로 쓰였는지 확인하는 함수

전달인자 :

- char *lexeme // 유효한지 확인할 명령어

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

리턴 값 : int // 유효하면 1, 유효하지 않으면 0

int checkSlashCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '/'가 제대로 쓰였는지 확인하는 함수

전달인자 :

- char *lexeme // 유효한지 확인할 명령어

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

리턴 값 : int // 유효하면 1, 유효하지 않으면 0

int checkMinusCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '-'가 제대로 쓰였는지 확인하는 함수

전달인자 :

- char *lexeme // 유효한지 확인할 명령어

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

리턴 값 : int // '-'가 포함 되어있고 유효하면 3, '-'가 없고 유효하면 1, 유효하지 않으면 0

int checkNumberAndStarCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '*'와 숫자가 제대로 쓰였는지 확인하는 함수

전달인자 :

- char *lexeme // 유효한지 확인할 명령어

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

리턴 값 : int // '*'이면 2, 숫자이면 1, 유효하지 않으면 0

char *commaStrtok(char *start); // ',' 문자를 검사할 때 사용하기 위해 만든 수행되는 strtok 함수, ','만 구분자로 인식한다는 것 외에는 strtok함수와 동일하게 작동함

전달인자 : char *start // strtok 할 문자열의 시작 주소

리턴 값 : char * // 찾아낸 토큰

<ssu_crond.c>

void startCrond(); // ssu_crond 실행시키는 함수

전달인자 : 없음

리턴 값 : 없음

int ssu_daemon_init(const char *path); // 디몬 프로세스 생성하는 함수

전달인자 : const char *path // 디몬 프로세스 실행할 경로

리턴 값 : int // 정상 종료 시 0 리턴

void checkRunCommand(); // 실행시킬 명령어 있는지 확인하는 함수

전달인자 : 없음

리턴 값 : 없음

int checkValidCommand(const char *input_command); // 실행 주기를 확인해 실행시킬 시간인지 확인하는 함수

전달인자 : const char *input_command // ssu_crontab_file에서 읽어온 명령어

리턴 값 : int // 유효하면 (실행해야 한다면) 1, 유효하지 않으면 0

int checkRunCycle(char *run_cycle, int run_cycle_index, int current_time); // 실행주기 확인하는 함수

전달인자 :

- char *run_cycle // 확인 할 실행 주기

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

- int current_time // 실행 주기에 맞는 현재 시간 (분,시간,일,월,요일 중 확인할 시간을 전달)

리턴 값 : int // 유효하면 (실행해야 한다면) 1, 유효하지 않으면 0

int checkCommaCommand(char *lexeme, int run_cycle_index, int current_time); //',' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

전달인자 :

- char * lexeme // 확인 할 실행 주기

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

- int current_time // 실행 주기에 맞는 현재 시간 (분,시간,일,월,요일 중 확인할 시간을 전달)

리턴 값 : int // 유효하면 (실행해야 한다면) 1, 유효하지 않으면 0

int checkSlashCommand(char *lexeme, int run_cycle_index, int current_time); //'/' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

전달인자 : 전달인자 :

- char * lexeme // 확인 할 실행 주기

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

- int current_time // 실행 주기에 맞는 현재 시간 (분,시간,일,월,요일 중 확인할 시간을 전달)

리턴 값 : int // 유효하면 (실행해야 한다면) 1, 유효하지 않으면 0

int checkMinusCommand(char *lexeme, int run_cycle_index, int current_time, int increase); //'-' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

전달인자 : 전달인자 :

- char * lexeme // 확인 할 실행 주기

- int run_cycle_index // 무엇을 뜻하는 실행주기인지 (0:분, 1:시간, 2:일, 3:월, 4:요일)

- int current_time // 실행 주기에 맞는 현재 시간 (분,시간,일,월,요일 중 확인할 시간을 전달)

리턴 값 : int // 유효하면 (실행해야 한다면) 1, 유효하지 않으면 0

int checkNumberAndStarCommand(char *lexeme, int current_time); //'*', 숫자를 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

전달인자 : 전달인자 :

- char * lexeme // 확인 할 실행 주기

- int current_time // 실행 주기에 맞는 현재 시간 (분,시간,일,월,요일 중 확인할 시간을 전달)

리턴 값 : int // 유효하면 (실행해야 한다면) 1, 유효하지 않으면 0

char *commaStrtok(char *start); // ',' 이용한 실행주기에서 각각의 실행주기 분리해낼 때 사용하는 함수

전달인자 : char *start // strtok 할 문자열의 시작 주소

리턴 값 : char * // 찾아낸 토큰

<ssu_rsync.c>

void do_tOption(const char *src_path_name, const char *dst_path_name); // -t 옵션 수행하는 함수

전달인자 :

- const char *src_path_name // 프로그램 실행 시 전달된 src 인자의 절대경로

- const char *dst_path_name // 프로그램 실행 시 전달된 dst 인자의 절대경로

리턴 값 : 없음

void do_mOption(FILE *tmp_log_fp, const char *path_name, const char *src_path_name, const char *dst_path_name, int sync_dir_flag); // -m 옵션 수행하는 함수

전달인자 :

- FILE *tmp_log_fp // 임시로 삭제 로그를 저장해 둘 임시파일의 파일 포인터

- const char *path_name // 로그에 출력할 상대경로

- const char *src_path_name // src 파일의 절대경로

- const char *dst_path_name // dst 파일의 절대경로

- int sync_dir_flag // -r 옵션으로 실행되었다면 1을, 그렇지 않다면 0을 전달

리턴 값 : 없음

void printUsage(const char *process_name); // 프로세스 사용법 출력하는 함수

전달인자 : const char *process_name // 프로그램 이름 (main의 전달인자 argv[0])

리턴 값 : 없음

void checkProcessArguments(int argc, char *argv[]); // 프로세스에 전달된 인자들이 유효한지 확인하는 함수

전달인자 :

- int argc // 프로그램 실행 시 전달된 인자의 개수

- char *argv[] // 프로그램 실행시 전달된 인자들

리턴 값 : 없음

void syncDirectory(const char *src_path_name, const char *dst_path_name, int sync_dir_flag); // 디렉토리를 동기화하는 함수

전달인자 :

- const char *src_path_name // src 파일의 절대경로

- const char *dst_path_name // dst 파일의 절대경로

- int sync_dir_flag // -r 옵션으로 실행되었다면 1을, 그렇지 않다면 0을 전달

리턴 값 : 없음

void syncFile(const char *src_file_name, const char *dst_path_name); // 파일을 동기화하는 함수

전달인자 :

- const char *src_file_name // src 파일의 파일명

- const char *dst_path_name // dst 디렉토리의 절대경로

리턴 값 : 없음

void copy(const char *src, const char *dst); // 파일을 복사하는 함수

전달인자 :

- const char *src // src 파일의 절대경로

- const char *dst // dst 파일의 절대경로

리턴 값 : 없음

void removeDirectory(FILE *tmp_log_fp, const char *path_name, const char *target); // 디렉토리와 그 디렉토리 하위의 모든 파일을 삭제하는 함수

전달인자 :

- FILE *tmp_log_fp // 임시로 삭제 로그를 저장해 둘 임시파일의 파일 포인터, 로그 출력할 필요 없으면 NULL

- const char *path_name // 로그에 출력할 상대경로

- const char *target // 삭제 대상 디렉토리 경로

리턴 값 : 없음

void printLog(FILE *tmp_log_fp, const char *src_path_name, int sync_dir_flag); // 로그를 출력하는 함수

전달인자 :

- FILE *tmp_log_fp // 임시로 삭제 로그를 저장해 둘 임시파일의 파일 포인터, 로그 출력할 필요 없으면 NULL

- const char *src_path_name // src파일 경로

- int sync_dir_flag // -r 옵션 지정되었으면 1, 그렇지 않으면 0

리턴 값 : 없음

void printFileNameAndSizeAtLogFile(FILE * fp, const char *src_path_name, const char *path_name, int sync_dir_flag); // 동기화된 파일의 이름과 사이즈를 로그파일에 출력하는 함수

전달인자 :

- FILE * fp

- const char *src_path_name // src 파일 경로

- const char *path_name // 로그에 출력할 상대경로

- int sync_dir_flag // -r 옵션 지정되었으면 1, 그렇지 않으면 0

리턴 값 : 없음

int checkSyncTarget(const char *src_file_name, const char *dst_path_name); // 동기화 대상 파일인지 확인하는 함수

전달인자 :

- const char *src_file_name // src 파일의 이름

- const char *dst_path_name // dst 디렉토리의 절대경로

리턴 값 : int // 동기화 대상 파일이면 1, 그렇지 않으면 0 리턴

static void sigint_during_sync_handler(int signo); // 동기화 중 발생한 SIGINT의 핸들러, 동기화를 취소하고, 기존의 내용으로 복원한다

전달인자 : int signo // 전달된 시그널

리턴 값 : 없음

4. 테스트 및 결과

<ssu_crontab, ssu_crond>

- **ssu_crontab 이용한 명령어 추가**

```
shlee@shlee-virtual-machine:~/project3$ ./ssu_crontab

20160548>add * * * * * echo "echo every minute" > echomsg.txt
0. * * * * * echo "echo every minute" > echomsg.txt

20160548>add 5 * * * * echo "echo every 5 minute" > echomsg.txt
0. * * * * * echo "echo every minute" > echomsg.txt
1. 5 * * * * echo "echo every 5 minute" > echomsg.txt

20160548>add 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
0. * * * * * echo "echo every minute" > echomsg.txt
1. 5 * * * * echo "echo every 5 minute" > echomsg.txt
2. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt

20160548>echo 5-100 * * * * echo "err test" > echomsg.txt
20160548>add echo 5-100 * * * * echo "err test" > echomsg.txt
0. * * * * * echo "echo every minute" > echomsg.txt
1. 5 * * * * echo "echo every 5 minute" > echomsg.txt
2. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt

20160548>
20160548>
20160548>
20160548>add * * 10/* * * echo "invalid run cycle" > echomsg.txt
0. * * * * * echo "echo every minute" > echomsg.txt
1. 5 * * * * echo "echo every 5 minute" > echomsg.txt
2. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt

20160548>add 23,30,33 * * * * echo "comma test" > echomsg.txt
0. * * * * * echo "echo every minute" > echomsg.txt
1. 5 * * * * echo "echo every 5 minute" > echomsg.txt
2. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
3. 23,30,33 * * * * echo "comma test" > echomsg.txt

20160548>exit
Runtime: 385:426034(sec:usec)
shlee@shlee-virtual-machine:~/project3$ vim ssu_crontab_log
shlee@shlee-virtual-machine:~/project3$ ./ssu_crond
```

->명령어 추가

->잘못된 명령어 추가 시 프롬프트 재출력

->엔터만 입력 시 프롬프트 재출력

->잘못된 실행주기 입력 시 추가 안됨 ('/' 뒤에는 '*'이 올 수 없음)

- **ssu_crontab 이용한 명령어 삭제**

```
shlee@shlee-virtual-machine:~/project3$ ./ssu_crontab
0. * * * * * echo "echo every minute" > echomsg.txt
1. 5 * * * * echo "echo every 5 minute" > echomsg.txt
2. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
3. 23,30,33 * * * * echo "comma test" > echomsg.txt
4. * * 10 6 2 echo "date test" > echomsg.txt
5. * * * 5 * echo "month test" > echomsg.txt
6. * * * * 1 echo "weekday test" > echomsg.txt

20160548>remove 0
0. 5 * * * * echo "echo every 5 minute" > echomsg.txt
1. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
2. 23,30,33 * * * * echo "comma test" > echomsg.txt
3. * * 10 6 2 echo "date test" > echomsg.txt
4. * * * 5 * echo "month test" > echomsg.txt
5. * * * * 1 echo "weekday test" > echomsg.txt

20160548>remove 3
0. 5 * * * * echo "echo every 5 minute" > echomsg.txt
1. 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
2. 23,30,33 * * * * echo "comma test" > echomsg.txt
3. * * * 5 * echo "month test" > echomsg.txt
4. * * * * 1 echo "weekday test" > echomsg.txt

20160548>remove 1
0. 5 * * * * echo "echo every 5 minute" > echomsg.txt
1. 23,30,33 * * * * echo "comma test" > echomsg.txt
2. * * * 5 * echo "month test" > echomsg.txt
3. * * * * 1 echo "weekday test" > echomsg.txt

20160548>exit
Runtime: 25:018428(sec:usec)
shlee@shlee-virtual-machine:~/project3$
```

- **위의 명령어 추가/삭제 및 ssu_crond 작동 로그들**

```
[Tue Jun  9 00:15:32 2020] add * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:16:10 2020] add 5 * * * * echo "echo every 5 minute" > echomsg.txt
[Tue Jun  9 00:18:36 2020] add 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
[Tue Jun  9 00:21:01 2020] add 23,30,33 * * * * echo "comma test" > echomsg.txt
[Tue Jun  9 00:29:49 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:30:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:30:09 2020] run 23,30,33 * * * * echo "comma test" > echomsg.txt
[Tue Jun  9 00:31:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:32:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:33:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:33:09 2020] run 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
[Tue Jun  9 00:33:09 2020] run 23,30,33 * * * * echo "comma test" > echomsg.txt
[Tue Jun  9 00:34:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:35:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:36:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:37:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:38:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:39:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:40:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:40:09 2020] run 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
[Tue Jun  9 00:41:09 2020] add * * 10 6 2 echo "date test" > echomsg.txt
[Tue Jun  9 00:41:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:41:35 2020] add * * * 5 * echo "month test" > echomsg.txt
[Tue Jun  9 00:42:04 2020] add * * * * 1 echo "weekday test" > echomsg.txt
[Tue Jun  9 00:42:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:43:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:44:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:45:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:46:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:47:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:48:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:49:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:50:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:51:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:52:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:53:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:54:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:55:09 2020] run * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:55:22 2020] remove * * * * * echo "echo every minute" > echomsg.txt
[Tue Jun  9 00:55:27 2020] remove * * 10 6 2 echo "date test" > echomsg.txt
[Tue Jun  9 00:55:37 2020] remove 20-40/7 0 9 6 2 echo "hello" > echomsg.txt
~
```

->명령어 추가 로그

->실행 주기별로 실행되는 명령어들

->실행 주기들 제대로 인식되는지 테스트하기 위해 추가. (실행 안돼야 함)

```
[Tue Jun  9 00:59:53 2020] remove 5 * * * * echo "echo every 5 minute" > echomsg.txt
[Tue Jun  9 01:00:54 2020] add */5 * * * * echo "echo every 5 minutes" > echomsg.txt
[Tue Jun  9 01:04:09 2020] run */5 * * * * echo "echo every 5 minutes" > echomsg.txt
[Tue Jun  9 01:09:09 2020] run */5 * * * * echo "echo every 5 minutes" > echomsg.txt
[Tue Jun  9 01:14:09 2020] run */5 * * * * echo "echo every 5 minutes" > echomsg.txt
~
~
~
~
"ssu_crontab_log" 48L, 3833C
```

->5분마다 실행되는 명령어 추가하고, 실행 확인

<ssu_rsync>

```
shlee@shlee-virtual-machine:~/project3$ mkdir dst
shlee@shlee-virtual-machine:~/project3$ ls
dst  echomsg.txt  makefile  ssu_crond  ssu_crond.c  ssu_crontab  ssu_crontab.c  ssu_crontab_file  ssu_crontab_log  ssu_rsync  ssu_rsync.c
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync echomsg.txt dst
Runtime: 0:000585(sec:usec)
shlee@shlee-virtual-machine:~/project3$ ls dst/
echomsg.txt
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync ssu_crontab_file dst
Runtime: 0:000319(sec:usec)
shlee@shlee-virtual-machine:~/project3$ ls dst/
echomsg.txt  ssu_crontab_file
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync -m echomsg.txt dst
Runtime: 0:000446(sec:usec)
shlee@shlee-virtual-machine:~/project3$ ls dst/
echomsg.txt
```

->일반 파일 동기화

->동기화 확인

->-m 옵션 확인을 위해 추가로 일반 파일 동기화

->-m 옵션 이용해 파일 동기

->-m 옵션 이용했으므로 해당 파일 이외의 파일은 사라진 것 확인

```
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync ../lsptest/ dst
Runtime: 0:058924(sec:usec)
shlee@shlee-virtual-machine:~/project3$ tree ../lsptest/
../lsptest/
├── 12.c
├── a.out
├── ssu_access_2
├── ssu_access_2.c
├── ssu_chown
├── ssu_chown.c
├── ssu_link.c
├── ssu_remove
├── ssu_remove.c
├── ssu_rename
├── ssu_rename.c
├── ssu_symlink.c
├── ssu_unlink_1
├── ssu_unlink_1.c
├── ssu_unlink_2
├── ssu_unlink_2.c
├── ssu_utime.c
├── testdir
│   ├── eoslab
│   ├── oslab -> eoslab
│   ├── ssu_oslab.c
│   ├── ssu_utime
│   └── testdir1-1
│       ├── ssu_symlink
│       └── ssu_test.txt
└── testdir2
    ├── ssu_link
    └── ssu_myfile

3 directories, 25 files
shlee@shlee-virtual-machine:~/project3$ tree dst
dst
├── 12.c
├── a.out
├── echomsg.txt
├── ssu_access_2
├── ssu_access_2.c
├── ssu_chown
├── ssu_chown.c
├── ssu_link.c
├── ssu_remove
├── ssu_remove.c
├── ssu_rename
├── ssu_rename.c
├── ssu_symlink.c
├── ssu_unlink_1
├── ssu_unlink_1.c
├── ssu_unlink_2
├── ssu_unlink_2.c
└── ssu_utime.c

0 directories, 18 files
```

->디렉토리 동기화

->디렉토리 동기화가 제대로 작동했는지 확인하기 위해 src 디렉토리 내부 구조 확인

->-r 옵션 없이 동기화 했으므로 하위 디렉토리 제외한 파일들 동기화 된 것 확인

```
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync -r ../lsptest/ dst
Runtime: 0:022268(sec:usec)
shlee@shlee-virtual-machine:~/project3$ tree dst/
dst/
├── 12.c
├── a.out
├── echomsg.txt
├── ssu_access_2
├── ssu_access_2.c
├── ssu_chown
├── ssu_chown.c
├── ssu_link.c
├── ssu_remove
├── ssu_remove.c
├── ssu_rename
├── ssu_rename.c
├── ssu_symlink.c
├── ssu_unlink_1
├── ssu_unlink_1.c
├── ssu_unlink_2
├── ssu_unlink_2.c
├── ssu_utime.c
├── testdir
│   ├── eoslab
│   ├── oslab
│   ├── ssu_oslab.c
│   ├── ssu_utime
│   └── testdir1-1
│       ├── ssu_symlink
│       └── ssu_test.txt
└── testdir2
    ├── ssu_link
    └── ssu_myfile

3 directories, 26 files
```

->-r 옵션 이용한 디렉토리 동기화

->-r 옵션 이용해 동기화 했으므로 하위 디렉토리까지 모두 동기화 된 것 확인

```
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync -m echomsg.txt dst
Runtime: 0:002360(sec:usec)
shlee@shlee-virtual-machine:~/project3$ tree dst/
dst/
└── echomsg.txt

0 directories, 1 file
shlee@shlee-virtual-machine:~/project3$ ./ssu_rsync -t ../lsptest/ dst
Runtime: 0:005342(sec:usec)
shlee@shlee-virtual-machine:~/project3$ tree dst/
dst/
├── 12.c
├── a.out
├── echomsg.txt
├── ssu_access_2
├── ssu_access_2.c
├── ssu_chown
├── ssu_chown.c
├── ssu_link.c
├── ssu_remove
├── ssu_remove.c
├── ssu_rename
├── ssu_rename.c
├── ssu_symlink.c
├── ssu_unlink_1
├── ssu_unlink_1.c
├── ssu_unlink_2
├── ssu_unlink_2.c
└── ssu_utime.c

0 directories, 18 files
```

->-m 옵션 사용해 나머지 파일들 제대로 삭제되는지 확인

->-m 옵션 사용했으므로 해당 파일 이외의 모든 파일 사라짐

->-t 옵션 사용해 동기화

->-t 옵션 사용해 제대로 동기화 된 것 확인

- 위의 내용들 로그

```
shlee@shlee-virtual-machine:~/project3$ cat ssu_rsync_log
[Tue Jun  9 00:30:38 2020] ssu_rsync  echomsg.txt dst
        echomsg.txt 11bytes
[Tue Jun  9 00:31:10 2020] ssu_rsync  ssu_crontab_file dst
        ssu_crontab_file 192bytes
[Tue Jun  9 00:31:43 2020] ssu_rsync -m echomsg.txt dst
        echomsg.txt 18bytes
        ssu_crontab_file delete
[Tue Jun  9 00:32:07 2020] ssu_rsync  ../lsptest/ dst
        ssu_access_2 8504bytes
        ssu_rename.c 780bytes
        12.c 755bytes
        ssu_chown.c 651bytes
        ssu_link.c 295bytes
        ssu_remove.c 479bytes
        ssu_unlink_2.c 848bytes
        a.out 8576bytes
        ssu_unlink_1 8568bytes
        ssu_unlink_2 8696bytes
        ssu_access_2.c 538bytes
        ssu_unlink_1.c 388bytes
        ssu_utime.c 696bytes
        ssu_remove 8568bytes
        ssu_symlink.c 356bytes
        ssu_chown 8648bytes
        ssu_rename 8632bytes
[Tue Jun  9 00:32:48 2020] ssu_rsync -r ../lsptest/ dst
        testdir2/ssu_link 8480bytes
        testdir2/ssu_myfile 0bytes
        testdir/ssu_oslab.c 102bytes
        testdir/testdir1-1/ssu_test.txt 74bytes
        testdir/testdir1-1/ssu_symlink 8512bytes
        testdir/oslab 0bytes
        testdir/ssu_utime 8624bytes
        testdir/eoslab 0bytes
[Tue Jun  9 00:33:16 2020] ssu_rsync -m echomsg.txt dst
        echomsg.txt 11bytes
        testdir2/ssu_link delete
        testdir2/ssu_myfile delete
        ssu_access_2 delete
        ssu_rename.c delete
        12.c delete
        ssu_chown.c delete
        ssu_link.c delete
        ssu_remove.c delete
        ssu_unlink_2.c delete
        a.out delete
        ssu_unlink_1 delete
        ssu_unlink_2 delete
        ssu_access_2.c delete
        ssu_unlink_1.c delete
        ssu_utime.c delete
        ssu_remove delete
        testdir/ssu_oslab.c delete
        testdir/testdir1-1/ssu_test.txt delete
        testdir/testdir1-1/ssu_symlink delete
        testdir/oslab delete
        testdir/ssu_utime delete
        testdir/eoslab delete
        ssu_symlink.c delete
        ssu_chown delete
        ssu_rename delete
[Tue Jun  9 00:33:39 2020] ssu_rsync -t ../lsptest/ dst
        ssu_access_2 8504bytes
        ssu_rename.c 780bytes
        12.c 755bytes
        ssu_chown.c 651bytes
        ssu_link.c 295bytes
        ssu_remove.c 479bytes
        ssu_unlink_2.c 848bytes
        a.out 8576bytes
        ssu_unlink_1 8568bytes
        ssu_unlink_2 8696bytes
        ssu_access_2.c 538bytes
        ssu_unlink_1.c 388bytes
        ssu_utime.c 696bytes
        ssu_remove 8568bytes
        ssu_symlink.c 356bytes
        ssu_chown 8648bytes
        ssu_rename 8632bytes
shlee@shlee-virtual-machine:~/project3$
```

- **동기화 중 SIGINT 발생 시 TEST**

```
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ mkdir dst
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ tree dst
dst

0 directories, 0 files
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ ./ssu_rsync -r ../project2 dst
^CSIGINT raised during sync
Runtime: 4:414300(sec:usec)
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ ls dst/
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ ./ssu_rsync ../project2 dst
Runtime: 24:137533(sec:usec)
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ tree dst
dst
├── Makefile
├── log.txt
├── monitor.c
├── monitor.o
├── ssu_mntr
├── ssu_mntr.c
├── ssu_mntr.o
└── tmp

0 directories, 8 files
```

->dst 디렉토리는 비어있는 상태

->디렉토리 동기화 중 SIGINT 발생

동기화 중단되어 dst 디렉토리는 비어있던 -> 그대로

->추가 테스트를 위해 디렉토리를 동기화 함 (SIGINT 테스트를 위해 프로그램 실행 속도를 느리게 만듦)

```
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ ./ssu_rsync -m ssu_rsync_log dst
^CSIGINT raised during sync
Runtime: 24:759770(sec:usec)
shlee@shlee-virtual-machine:~/workspace/lsp/project3$ tree dst
dst
├── Makefile
├── log.txt
├── monitor.c
├── monitor.o
├── ssu_mntr
├── ssu_mntr.c
├── ssu_mntr.o
└── tmp
```

->-m 옵션으로 동기화 중 SIGINT 발생

->아무 파일도 삭제/추가되지 않고 원래 상태 유지

5. 소스코드

<ssu_crontab.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <fcntl.h>
#include <sys/time.h>

#define SECOND_TO_MICRO 1000000
void ssu_runtime(struct timeval *begin_t, struct timeval *end_t); // 수행시간 계산 함수

enum {MIN, HOUR, DAY, MON, WDAY};

#define BUFFER_SIZE 1024
#define MAX_COMMAND_COUNT 100

const char *PROMPT_STR = "20160548>"; // 프롬프트에 출력할 문자열
const char *CRONTAB_FILE_NAME = "ssu_crontab_file"; // crontab 명령어 저장할 파일 명
const char *LOG_FILE_NAME = "ssu_crontab_log"; // log 저장할 파일 명

char crontab_commands[MAX_COMMAND_COUNT][BUFFER_SIZE]; // crontab 명령어를 불러와 저장할 버퍼
int crontab_commands_count; // crontab 명령어 개수를 저장할 변수

void printCrontabCommands(void); // 저장된 명령어들을 읽어와 프롬프트에 출력하는 함수
```

```
void getCrontabCommands(void); // 파일에 저장된 명령어들을 읽어오는 함수

void doAddCommand(char *input_command); // 명령어를 추가하는 함수

void doRemoveCommand(void); // 명령어를 삭제하는 함수

int checkValidCommand(const char *input_command); // 유효한 명령어인지 확인하는 함수

int checkIncludeValidCharactersOnly(const char *lexeme); // 사용할 수 없는 문자가 명령어에 포함되어있는지 확인하는 함수

int checkValidRunCycle(char *lexeme, int run_cycle_index); // 실행 주기가 유효한 명령어인지 확인하는 함수

int checkCommaCommand(char *lexeme, int run_cycle_index); // 실행 주기 문자 ','가 제대로 쓰였는지 확인하는 함수

int checkSlashCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '/'가 제대로 쓰였는지 확인하는 함수

int checkMinusCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '-'가 제대로 쓰였는지 확인하는 함수

int checkNumberAndStarCommand(char *lexeme, int run_cycle_index);// 실행 주기 문자 '*'와 숫자가 제대로 쓰였는지 확인하는 함수

char *commaStrtok(char *start); // ',' 문자를 검사할 때 사용하기 위해 만든 수행되는 strtok 함수, ','만 토큰으로 인식한다는 것 외에는 strtok함수와 동일하게 작동함


int main(void)

{

	char input_str[BUFFER_SIZE]; // 사용자가 프롬프트에 입력한 명령어를 저장할 배열

	char *input_command; // 사용자가 입력한 명령어를 strtok으로 나누어 확인 할 때 사용하는 포인터

	struct timeval begin_t, end_t; // 프로세스 수행시간 측정을 위해 사용

	gettimeofday(&begin_t, NULL); // 시작 시간 기록


	if (access(CRONTAB_FILE_NAME, F_OK) < 0) { // ssu_crontab_file이 존재하지 않으면 새로 생성

		close(open(CRONTAB_FILE_NAME, O_WRONLY | O_CREAT | O_TRUNC, 0666));

	}

	if (access(LOG_FILE_NAME, F_OK) < 0) {   // ssu_crontab_log 파일이 존재하지 않으면 새로 생성

		close(open(LOG_FILE_NAME, O_WRONLY | O_CREAT | O_TRUNC, 0666));

	}
```

```
getCrontabCommands(); // 파일에서 기존에 저장되어 있던 명령어들을 읽어옴
printCrontabCommands(); // 저장되어 있던 명령어들을 프롬프트에 출력함
while (1) { // 사용자가 exit 입력해 프로세스 종료시킬 때 까지 반복

    printf("%s", PROMPT_STR); // 프롬프트 문자열 출력
    fgets(input_str, sizeof(input_str), stdin); // 사용자에게 명령어 입력받음
    if (input_str[0] == '₩n') continue; // 사용자가 엔터 입력했을 시에는 다시 프롬프트 문자열 출력
    if (strlen(input_str) > 0) input_str[strlen(input_str) - 1] = '₩0'; // fgets로 문자열 읽어오면 맨 끝에 개행문자가 남아있으므로 개행을 없애고 널문자로 바꿈
    else continue;

    input_command = strtok(input_str, " "); // 사용자가 어떤 명령어를 입력했는지 확인하기 위해 strtok 사용
    if (!strcmp(input_command, "add") || !strcmp(input_command, "ADD")) { // 사용자가 add 명령어를 사용했다면
        doAddCommand(input_command); // 명령어 추가
    } else if (!strcmp(input_command, "remove") || !strcmp(input_command, "REMOVE")) { // 사용자가 remove 명령어를 사용했다면
        doRemoveCommand(); // 명령어 삭제
    } else if (!strcmp(input_command, "exit") || !strcmp(input_command, "EXIT")) { // 사용자가 exit 명령어를 사용했다면
        break; // 프로세스 종료를 위해 반복문 빠져나감
    } else {
        continue; // 인식할 수 없는 명령어라면 다시 프롬프트 문자열 출력
    }

    getCrontabCommands(); // 파일에 저장된 명령어들 읽어옴
    printCrontabCommands(); // 파일에 저장된 명령어들을 프롬프트에 출력함
```

```
	}

	gettimeofday(&end_t, NULL); // 종료 시간 기록
	ssu_runtime(&begin_t, &end_t); // 프로그램 실행 시간 계산, 출력

	exit(0); // 프로세스 종료
}

void printCrontabCommands(void) {
	int i;

	for (i = 0; i < crontab_commands_count; ++i) { // 명령어 개수만큼 반복
		printf("%d. %s₩n", i, crontab_commands[i]); // 명령어들 저장해 놓은 배열을 이용해 명령어를 화면에 출력한다
	}

	printf("₩n");
}

void getCrontabCommands(void) {
	FILE *fp;
	int i;

	if ((fp = fopen(CRONTAB_FILE_NAME, "r")) == NULL){ // 명령어 저장된 파일 오픈
		fprintf(stderr, "fopen error for %s₩n", CRONTAB_FILE_NAME);
		exit(1);
	}
```

```
	for (i = 0; i < MAX_COMMAND_COUNT && !feof(fp); ++i) { // 명령어 읽어올 수 있는 최대 개수만큼 읽는다
		if (fgets(crontab_commands[i], sizeof(crontab_commands[i]), fp) == NULL) { // 파일에서 한줄(명령어 하나) 읽어옴
			break; // 아무것도 읽어온게 없다면 명령어를 다 읽어왔다는 뜻이므로 반복 종료
		} else {
			if (strlen(crontab_commands[i]) > 0)
				crontab_commands[i][strlen(crontab_commands[i]) - 1] = '₩0'; // 명령어 맨 끝의 개행문자 없앤다
		}
	}
	crontab_commands_count = i; // 읽어온 명령어 개수 저장

	fclose(fp);

	return;
}

void doAddCommand(char *input_command){
	FILE *log_fp;
	FILE *command_fp;
	char new_command[BUFFER_SIZE];
	time_t current_time;
	char *time_str;
	struct flock fl;

	// 입력된 명령어의 실행 주기가 유효한지 검사
	while(*input_command != '₩0') ++input_command; // 명령어 앞쪽의 불필요한 문자, 널문자 pass
```

```
while(*input_command == '₩0') ++input_command;
strcpy(new_command, input_command);

if (!checkValidCommand(new_command)) return; // 실행 주기의 입력이 잘못 된 경우 에러 처리 후 프롬프트 제어가 넘어감

// 유효한 명령어인 경우
if ((command_fp = fopen(CRONTAB_FILE_NAME, "a")) == NULL) { // 명령어 저장하는 파일 오픈
        fprintf(stderr, "fopen error for %s₩n", CRONTAB_FILE_NAME);
        exit(1);
}

// 해당 파일은 ssu_crond와 함께 쓰기 때문에 오류 발생하지 않기 위해 쓰기 락을 걸어준다
fl.l_type = F_WRLCK;
fl.l_start = 0;
fl.l_whence = SEEK_SET;
fl.l_len = 0;

if (fcntl(fileno(command_fp), F_SETLKW, &fl) < 0) {
        fprintf(stderr, "fcntl error for %s₩n", CRONTAB_FILE_NAME);
        exit(1);
}

fprintf(command_fp, "%s₩n", new_command); // 파일에 새로운 명령어를 추가한다

// 쓰기 작업 끝났으므로 락 해제한다
fl.l_type = F_UNLCK;
```

```
	fl.l_start = 0;
	fl.l_whence = SEEK_SET;
	fl.l_len = 0;

	if (fcntl(fileno(command_fp), F_SETLKW, &fl) < 0) {
		fprintf(stderr, "fcntl error for %s₩n", CRONTAB_FILE_NAME);
		exit(1);
	}

	fclose(command_fp);

	// 명령어 추가 됐으면 로그 남긴다
	if ((log_fp = fopen(LOG_FILE_NAME, "a")) == NULL) { // 로그 파일 오픈
		fprintf(stderr, "fopen error for %s₩n", LOG_FILE_NAME);
		exit(1);
	}

	current_time = time(NULL);
	time_str = ctime(&current_time);
	time_str[strlen(time_str) -1] = '₩0'; // ctime으로 리턴된 시간 문자열 맨 끝의 개행문제 지운다
	fprintf(log_fp, "[%s] add %s₩n", time_str, new_command); // 로그 메시지 출력
	fclose(log_fp);
}

void doRemoveCommand(void){
	int i;
	int selected_index;
```

```
char *next_lexeme;
FILE *command_fp;
FILE *log_fp;
time_t current_time;
char *time_str;
struct flock fl;

next_lexeme = strtok(NULL, " "); // 입력된 문자열에서 삭제할 명령어 번호 추출
if (next_lexeme == NULL) return; // 입력된 번호가 없다면 프롬프트로 제어가 넘어감
for (i = 0; i < strlen(next_lexeme); ++i) // 숫자가 아닌 문자가 입력됐다면 프롬프트로 제어가 넘어감
        if (!isdigit(next_lexeme[i])) return;

selected_index = atoi(next_lexeme); // 입력된 번호를 정수형으로 변환
if (selected_index < 0 || crontab_commands_count <= selected_index) return; // (인덱스 번호를 벗어난) 잘못된 번호가 입력되었다면 프롬프트로 제어가 넘어감

if ((command_fp = fopen(CRONTAB_FILE_NAME, "w")) == NULL) { // 명령어 저장된 파일 오픈
        fprintf(stderr, "fopen error for %s₩n", CRONTAB_FILE_NAME);
        exit(1);
}

// 쓰기 락을 건다
fl.l_type = F_WRLCK;
fl.l_start = 0;
fl.l_whence = SEEK_SET;
fl.l_len = 0;
```

```
if (fcntl(fileno(command_fp), F_SETLKW, &fl) < 0) {
        fprintf(stderr, "fcntl error for %s₩n", CRONTAB_FILE_NAME);
        exit(1);
}

for (i = 0; i < crontab_commands_count; ++i) { // 삭제할 명령어를 뺀 모든 명령어들을 파일에 다시 쓴다
        if (i == selected_index) continue; // 삭제할 명령어 차례이면 skip 한다

        fprintf(command_fp, "%s₩n", crontab_commands[i]); // 파일에 명령어 출력
}

// 락 해제
fl.l_type = F_UNLCK;
fl.l_start = 0;
fl.l_whence = SEEK_SET;
fl.l_len = 0;

if (fcntl(fileno(command_fp), F_SETLKW, &fl) < 0) {
        fprintf(stderr, "fcntl error for %s₩n", CRONTAB_FILE_NAME);
        exit(1);
}

fclose(command_fp);

// 삭제 됐으면 로그 남긴다
if ((log_fp = fopen(LOG_FILE_NAME, "a")) == NULL) {
        fprintf(stderr, "open error for %s₩n", LOG_FILE_NAME);
```

```
            exit(1);
        }

        current_time = time(NULL);
        time_str = ctime(&current_time);
        time_str[strlen(time_str) -1] = '₩0';
        fprintf(log_fp, "[%s] remove %s₩n", time_str, crontab_commands[selected_index]); // 로그 메시지 출력
        fclose(log_fp);
}

int checkValidCommand(const char *input_command) {
        char *next_lexeme;
        char copied_input_command[BUFFER_SIZE];
        int lexeme_count = 0;
        strcpy(copied_input_command, input_command);

        next_lexeme = strtok(copied_input_command, " "); // 맨 앞의 실행 주기 가져온다
        do {
            if (lexeme_count == 5) break; // 실행 주기 5개 모두 검사했으면 반복 종료
            if (!checkIncludeValidCharactersOnly(next_lexeme)) return 0; // 실행 주기에 유효한 문자만 포함되어 있는지 검사, 유효하지 않은 명령어이면 0 리턴
            if (!checkValidRunCycle(next_lexeme, lexeme_count)) return 0; // 유효한 실행 주기인지 검사, 유효하지 않은 명령어이면 0 리턴

            ++lexeme_count; // 검사한 실행 주기 개수 ++
            //printf("lexeme complete₩n");
        } while ((next_lexeme = strtok(NULL, " ")) != NULL); // 다음 실행주기 가져온다
```

```
	return 1; // 유효한 명령어이면 1 리턴
}

int checkIncludeValidCharactersOnly(const char *lexeme) {
	int i;
	for (i = 0; i < strlen(lexeme); ++i) { // 인자로 전달된 문자열의 모든 문자를 확인
		if (lexeme[i] == '*' || lexeme[i] == '-'|| lexeme[i] == ','|| lexeme[i] == '/' || (0 <= lexeme[i] && lexeme[i] <= '9')) continue; // 유효한 문자라면 다음 문자 확인
		else return 0; // 유효하지 않은 문자가 포함되어있다면 0 리턴
	}

	return 1; // 유효한 문자만 들어있다면 1 리턴
}

int checkValidRunCycle(char *lexeme, int run_cycle_index) { // 없어도 되는 함수...
	//printf("check valid runcycle %s₩n", lexeme); //////////////////////
	return checkCommaCommand(lexeme, run_cycle_index);
}

int checkCommaCommand(char *lexeme, int run_cycle_index){
	char *ptr;

	//printf("check comma command %s₩n", lexeme); ////////////////////////

	if (strstr(lexeme, ",") == NULL) { // 실행 주기에 ','가 들어있는지 확인
		return checkSlashCommand(lexeme, run_cycle_index); // 들어있지 않다면 바로 '/' 문자가 유효한지 검사하러 간다
	} else {
```

```
		if (lexeme[0] == ',') return 0; // 맨 앞 문자가 콤마이면 잘못 된 실행 주기
		if (lexeme[strlen(lexeme) - 1] == ',') return 0; // 맨 마지막 문자가 콤마이면 잘못 된 실행 주기

		ptr = commaStrtok(lexeme); // ','를 토큰으로 해서 실행주기를 분리한다
		do {
			if (!checkSlashCommand(ptr, run_cycle_index)) return 0; // '/'문자가 유효한지 확인한다, 유효하지 않으면 0을 리턴한다
		} while ((ptr = commaStrtok(NULL)) != NULL); // ','를 토큰으로 해서 실행주기를 분리한다

		return 1; // 유효하면 1을 리턴한다
	}
}

int checkSlashCommand(char *lexeme, int run_cycle_index){
	char *ptr1;
	char *ptr2;
	int checkMinusCommandResult1;
	int checkMinusCommandResult2;

	//printf("check slash command %s₩n", lexeme); ///////////////////////

	if ((ptr2 = strstr(lexeme, "/")) == NULL) { // 실행 주기에 '/'문자가 들어있지 않다면
		return checkMinusCommand(lexeme, run_cycle_index); // '-' 바로 문자가 유효한지 확인하러 간다
	} else {
		if (lexeme[0] == '/') return 0; // 맨 앞 문자가 슬래쉬이면 잘못 된 실행 주기
		if (lexeme[strlen(lexeme) - 1] == '/') return 0; // 맨 마지막 문자가 슬래쉬이면 잘못 된 실행 주기
```

```
            ptr1 = lexeme; // '/' 앞의 실행 주기는 ptr1에
            *ptr2 = '₩0';
            ++ptr2; // '/' 뒤의 실행 주기는 ptr2에
            if ((strstr(ptr2, "/")) != NULL) return 0; // '/'문자가 여러개 있다면 잘못 된 실행 주기

            // '/' 뒤에는 숫자만 올 수 있다. checkMinusCommandResult2가 1이 아니라면 숫자가 아니란 뜻이므로 유효하지 않은 실행 주기이다.
            checkMinusCommandResult1 = checkMinusCommand(ptr1, run_cycle_index); // '/' 앞의 실행 주기가 유효한지 확인해 그 결과를 저장
            checkMinusCommandResult2 = checkMinusCommand(ptr2, run_cycle_index); // '/' 뒤의 실행 주기가 유효한지 확인해 그 결과를 저장
            if (checkMinusCommandResult1 == 1) return 0; // '/'문자 앞에는 '*' 또는 '/'를 이용한 범위가 와야 한다.
            if (checkMinusCommandResult2 != 1) return 0; // '/' 뒤에는 반드시 숫자가 와야 한다

            if (checkMinusCommandResult1 * checkMinusCommandResult2) return 1;   // '/' 앞뒤 실행주기가 모두 유효하면 1 리턴
            else return 0;
        }
}

int checkMinusCommand(char *lexeme, int run_cycle_index){
        char *ptr1;
        char *ptr2;
        int checkNumAndStarResult;

        //printf("check minue command %s₩n", lexeme); ///////////////////////

        if ((ptr2 = strstr(lexeme, "-")) == NULL) { // 실행주기에 '-'가 들어있지 않다면 바로 다음 단계 확인하러 감
```

```
		return checkNumberAndStarCommand(lexeme, run_cycle_index);
	} else {
		if (lexeme[0] == '-') return 0; // 맨 앞 문자가 '-'이면 잘못 된 실행 주기
		if (lexeme[strlen(lexeme) - 1] == '-') return 0; // 맨 마지막 문자가 '-'이면 잘못 된 실행 주기

		ptr1 = lexeme; // '-'앞의 실행주기는 ptr1에
		*ptr2 = '₩0';
		++ptr2; // '-'뒤의 실행주기는 ptr1에
		if ((strstr(ptr2, "-")) != NULL) return 0; // '-'문자가 여러개 있다면 잘못 된 실행 주기

		// '-' 앞 뒤가 모두 숫자여야 유효한 실행 주기이다. 둘중 하나라도 '*'이면 안됨
		checkNumAndStarResult = checkNumberAndStarCommand(ptr1, run_cycle_index) * checkNumberAndStarCommand(ptr2, run_cycle_index);
		if (checkNumAndStarResult == 1) return 3; // '/' 뒤에는 숫자만 올 수 있다. 이를 구분하기 위해 3을 리턴한다
		else return 0;
	}
}

int checkNumberAndStarCommand(char *lexeme, int run_cycle_index){
	int i;
	int num;

	//printf("check number and star command %s₩n", lexeme); ////////////////////////

	if (!strcmp(lexeme, "*")) return 2; // '-' 앞뒤로 '*'가 오면 안되기 때문에, '-'에서 '*'가 있는지 판단하기 위해 '*'인 경우에는 2를 리턴한다.
```

```
//printf("%s₩n", lexeme);
for (i = 0; i < strlen(lexeme); ++i) {
        if (!isdigit(lexeme[i])) return 0; // '*'도 아니고, 숫자도 아니면 잘못된 실행주기이므로 0리턴
}

num = atoi(lexeme);
switch(run_cycle_index) { // 각 실행 주기 별 유효한 범위 내에 있는지 검사
        case MIN: // 분 (0~59)
                if (num < 0 || num > 59) return 0;
                break;
        case HOUR: // 시 (0~23)
                if (num < 0 || num > 23) return 0;
                break;
        case DAY: // 일 (0~31)
                if (num < 0 || num > 31) return 0;
                break;
        case MON: // 월 (1~12)
                if (num < 1 || num > 12) return 0;
                break;
        case WDAY: // 요일 (0~6) (일요일부터 시작)
                if (num < 0 || num > 6) return 0;
                break;
        default:
                return 0;
}
//printf("check number end₩n");//////////////////////////
return 1; // 유효하다면 1 리턴
```

```
}

char *commaStrtok(char *start) {
	static char *next_start;
	char *prev_start;
	int i;
	int length;

	//printf("commastrtok start₩n");//////////////////

	if (start != NULL) { // 전달인자가 null이 아니라면 새로운 문자열에 대한 함수 호출
		next_start = start;
		prev_start = start;
	} else { // 기존의 문자열에 대한 함수 호출
		prev_start = next_start;
	}

	if (next_start == NULL) return NULL; // 문자열 끝까지 확인 마쳤다면 NULL 리턴

	length = strlen(next_start);
	for(i = 0; i < length; ++i) {
		if (next_start[i] == ',') break; // ',' 토큰 찾는다
	}

	if (i < length) {
		next_start[i] = '₩0'; // 토큰을 널문자로 바꾼다
		if (i + 1 < length)
```

```
                next_start = next_start + i + 1; // 다음 시작위치 저장한다
            else next_start = NULL; // 문자열 끝까지 확인했으면 NULL 저장한다
        } else {
            next_start = NULL;
        }

        //printf("commastrtok end₩n"); ////////////////////

        return prev_start;
}

void ssu_runtime(struct timeval *begin_t, struct timeval *end_t)
{
        // 시작시간과 종료시간의 차이 계산
        end_t->tv_sec -= begin_t->tv_sec;

        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); // 프로그램 실행에 걸린 시간 출력
}
```

<ssu_crond.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <signal.h>
#include <dirent.h>
#include <fcntl.h>
#include <sys/stat.h>

#define BUFFER_SIZE 1024
#define MAX_COMMAND_COUNT 100
#define NUMBER_OF_RUN_CYCLE_COMMANDS 5

const char *CRONTAB_FILE_NAME = "ssu_crontab_file"; // crontab 명령어 저장할 파일 명
const char *LOG_FILE_NAME = "ssu_crontab_log"; // log 저장할 파일 명

enum {MIN, HOUR, DAY, MON, WDAY};

void startCrond(); // ssu_crond 실행시키는 함수
int ssu_daemon_init(const char *path); // 디몬 프로세스 생성하는 함수

void checkRunCommand(); // 실행시킬 명령어 있는지 확인하는 함수
int checkValidCommand(const char *input_command); // 실행 주기를 확인해 실행시킬 시간인지 확인하는 함수
int checkRunCycle(char *run_cycle, int run_cycle_index, int current_time); // 실행주기 확인하는 함수
```

int checkCommaCommand(char *lexeme, int run_cycle_index, int current_time); //',' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

int checkSlashCommand(char *lexeme, int run_cycle_index, int current_time); //'/' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

int checkMinusCommand(char *lexeme, int run_cycle_index, int current_time, int increase); //'-' 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

int checkNumberAndStarCommand(char *lexeme, int current_time); //'*', 숫자를 이용한 실행주기를 확인해 실행해야 할 명령어인지 확인하는 함수

char *commaStrtok(char *start); // ',' 이용한 실행주기에서 각각의 실행주기 분리해낼 때 사용하는 함수

```
int main(void)
{
	//checkRunCommand(); // 디버깅용
	startCrond(); // crond 프로세스 시작

	exit(0);
}

void startCrond(){
	pid_t pid;
	if ((pid = fork()) == 0) { // 자식 프로세스 생성, 자식 프로세스라면
		char path[PATH_MAX];
		getcwd(path, PATH_MAX); // 현재 경로 구한다
		ssu_daemon_init(path); // 디몬 프로세스 실행
	} else if (pid < 0) { // fork 에러라면
		fprintf(stderr, "mntr starting error₩n");
		exit(1);
	} else { // 부모 프로세스라면
```

```
                return;
        }
}

int ssu_daemon_init(const char *path) { // 디몬 프로세스 시작하는 함수
        pid_t pid;
        int fd, maxfd;
        time_t current_time;
        struct tm *current_tm;
        int prev_minute;

        if ((pid = fork()) < 0) { // 백그라운드 프로세스가 된다
                fprintf(stderr, "fork error₩n");
                exit(1);
        }
        else if (pid != 0)
                exit(0);

        setsid(); // 새로운 프로세스 그룹 생성
        signal(SIGTTIN, SIG_IGN); // 터미널 입출력 시그널 무시
        signal(SIGTTOU, SIG_IGN); // 터미널 입출력 시그널 무시
        signal(SIGTSTP, SIG_IGN); // 터미널 입출력 시그널 무시
        maxfd = getdtablesize(); // 최대 디스크립터 개수 구함

        for (fd = 0; fd < maxfd; fd++) // 모든 디스크립터 close
                close(fd);
```

```
umask(0); // 파일 모드 생성 마스크 해제
chdir(path); // 디몬 프로세스 실행할 경로로 이동
fd = open("/dev/null", O_RDWR); // 표준 입출력, 에러 재지정
dup(0); // 표준 입출력, 에러 재지정
dup(0); // 표준 입출력, 에러 재지정

if (access(CRONTAB_FILE_NAME, F_OK) < 0) { // crontab 명령어 저장된 파일 없다면 생성
        close(open(CRONTAB_FILE_NAME, O_WRONLY | O_CREAT | O_TRUNC, 0666));
}

if (access(LOG_FILE_NAME, F_OK) < 0) { // crontab 로그 저장된 파일 없다면 생성
        close(open(LOG_FILE_NAME, O_WRONLY | O_CREAT | O_TRUNC, 0666));
}

prev_minute = -1;
while(1) {
        // 현재 시간 구한다
        current_time = time(NULL);
        current_tm = localtime(&current_time);

        if (prev_minute != current_tm->tm_min) { // 1분마다 실행할 명령어 있는지 확인
                prev_minute = current_tm->tm_min;
                checkRunCommand();
        }

        sleep(10); // 10초씩 sleep하면서 시간 확인
}
```

```
	return 0;
}

void checkRunCommand() {
	FILE *command_fp;
	FILE *log_fp;
	char buf[BUFFER_SIZE];
	char run_cycle[BUFFER_SIZE];
	char command[BUFFER_SIZE];
	int i, j;
	time_t current_time;
	char *time_str;
	struct flock fl;

	if ((command_fp = fopen(CRONTAB_FILE_NAME, "r")) == NULL) { // 실행할 명령어 저장된 파일 오픈
		fprintf(stderr, "fopen error for %s₩n", CRONTAB_FILE_NAME);
		exit(1);
	}

	if ((log_fp = fopen(LOG_FILE_NAME , "a")) == NULL) { // 로그 저장할 파일 오픈
		fprintf(stderr, "fopen error for %s₩n", LOG_FILE_NAME );
		exit(1);
	}

	// ssu_crontab 프로세스에서 파일을 수정할 수 있으므로 명령어 저장된 파일에 읽기 락을 건다
	fl.l_type = F_RDLCK;
```

```
fl.l_start = 0;
fl.l_whence = SEEK_SET;
fl.l_len = 0;

if (fcntl(fileno(command_fp), F_SETLKW, &fl) < 0) {
        fprintf(stderr, "fcntl error for %s₩n", CRONTAB_FILE_NAME);
        exit(1);
}

while (fgets(buf, BUFFER_SIZE, command_fp) != NULL) { // 파일에서 명령어 하나씩 읽어온다
        if (feof(command_fp)) break;

        if (strlen(buf) - 1 >= 0) buf[strlen(buf) - 1] = '₩0';

        // 실행 주기 명령어들 pass
        for (i = 0, j = 0; i < NUMBER_OF_RUN_CYCLE_COMMANDS && j < strlen(buf); ++i) {
                while(buf[j] == ' ') ++j;
                while(buf[j] != ' ') ++j;
        }
        strncpy(run_cycle, buf, j); // 실행주기 명령어들 따로 저장
        run_cycle[j] = '₩0';
        while(buf[j] == ' ') ++j; // 실행할 명령어로 인덱스 이동
        strcpy(command, buf + j); // 실행할 명령어 따로 저장

        if (checkValidCommand(run_cycle)) { // 현재 실행해야 할 실행주기인지 확인
                system(command); // 현재 실행할 실행주기라면 명령어 실행
```

```
                        current_time = time(NULL);
                        time_str = ctime(&current_time);
                        time_str[strlen(time_str) -1] = '₩0';
                        fprintf(log_fp, "[%s] run %s₩n", time_str, buf); // 실행 했으면 로그 출력
                }
        }

        // 명령어 확인 끝났으면 락 해제
        fl.l_type = F_UNLCK;
        fl.l_start = 0;
        fl.l_whence = SEEK_SET;
        fl.l_len = 0;

        if (fcntl(fileno(command_fp), F_SETLKW, &fl) < 0) {
                fprintf(stderr, "fcntl error for %s₩n", CRONTAB_FILE_NAME);
                exit(1);
        }

        fclose(command_fp);
        fclose(log_fp);
}

int checkValidCommand(const char *input_command) {
        char *next_lexeme;
        char copied_input_command[BUFFER_SIZE];
        int lexeme_count = 0;
        time_t current_time;
```

```
	struct tm *current_tm;

	current_time = time(NULL); // 현재 시간 가져옴
	current_tm = localtime(&current_time);

	//printf("min:%d, hour:%d, mday:%d, mon:%d, wday:%d₩n", current_tm->tm_min, current_tm->tm_hour, current_tm->tm_mday, current_tm->tm_mon, current_tm->tm_wday);

	strcpy(copied_input_command, input_command);

	next_lexeme = strtok(copied_input_command, " "); // 첫번째 실행 주기 가져옴
	do {
		if (lexeme_count == 5) break; // 실행주기 5개 모두 확인했다면 반복 종료

		switch(lexeme_count) { // 실행 주기별 확인
			case MIN: // 분 (0~59)
				if (!checkRunCycle(next_lexeme, MIN, current_tm->tm_min)) return 0; // 실행 x 인 경우 바로 리턴
				break;
			case HOUR: // 시 (0~23)
				if (!checkRunCycle(next_lexeme, HOUR, current_tm->tm_hour)) return 0; // 실행 x 인 경우 바로 리턴
				break;
			case DAY: // 일 (0~31)
				if (!checkRunCycle(next_lexeme, DAY, current_tm->tm_mday)) return 0; // 실행 x 인 경우 바로 리턴
				break;
			case MON: // 월 (1~12)
```

```
                        if (!checkRunCycle(next_lexeme, MON, current_tm->tm_mon + 1)) return 0; // 실행 x 인 경우 바로 리턴
                        break;
                case WDAY: // 요일 (0~6) (일요일부터 시작)
                        if (!checkRunCycle(next_lexeme, WDAY, current_tm->tm_wday)) return 0; // 실행 x 인 경우 바로 리턴
                        break;
                default:
                        return 0;
                }

                ++lexeme_count; // 확인한 실행 주기 개수 ++
        } while ((next_lexeme = strtok(NULL, " ")) != NULL); // 다음 실행 주기 가져옴

        return 1;
}

int checkRunCycle(char *run_cycle, int run_cycle_index, int current_time) { // 없어도 되는 함수...
        return checkCommaCommand(run_cycle, run_cycle_index, current_time);
}

int checkCommaCommand(char *lexeme, int run_cycle_index, int current_time){
        char *ptr;

        //printf("check comma command %s₩n", lexeme); ////////////////////////

        if (strstr(lexeme, ",") == NULL) { // 실행 주기에 ','가 없다면
                return checkSlashCommand(lexeme, run_cycle_index, current_time); // '/' 확인하러 간다
```

```
	} else {
		ptr = commaStrtok(lexeme); // ',' 기준으로 실행 주기 분리
		do {
			if (checkSlashCommand(ptr, run_cycle_index, current_time)) return 1; // ','은 or 이므로 실행주기 중 하나라도 유효하면 바로 1 리턴
		} while ((ptr = commaStrtok(NULL)) != NULL); // 다음 실행주기 확인

		return 0; // 여기까지 왔다면 유효하지 않은 실행주기임
	}
}

int checkSlashCommand(char *lexeme, int run_cycle_index, int current_time){
	char *ptr1;
	char *ptr2;
	int increase;
	int i;
	int malloc_flag = 0;
	int result;

	//printf("check slash command %s₩n", lexeme); ////////////////////////

	if ((ptr2 = strstr(lexeme, "/")) == NULL) { // 실행 주기에 '/'가 없다면
		return checkMinusCommand(lexeme, run_cycle_index, current_time, 1); // '-' 확인하러 간다
	} else {
		ptr1 = lexeme; // '/' 앞쪽의 실행 주기는 ptr1에
		*ptr2 = '₩0';
		++ptr2; // '/' 뒤의 실행 주기는 ptr2에
```

```
increase = atoi(ptr2); // '/' 뒤의 실행주기(무조건 숫자)를 정수형으로 변환하여 저장, 이 숫자를 이용해 '-'를 확인하면서 유효한 실행 주기인지 확인한다


if (!strcmp(ptr1, "*")) { // '/'앞의 실행주기가 '*' 이라면
    malloc_flag = 1;
    ptr1 = (char *) malloc(10);
    // '*'을 각 실행주기에 맞는 범위로 변환한다
    switch(run_cycle_index) {
        case MIN: // 분 (0~59)
            strcpy(ptr1, "0-59");
            break;
        case HOUR: // 시 (0~23)
            strcpy(ptr1, "0-23");
            break;
        case DAY: // 일 (0~31)
            strcpy(ptr1, "0-31");
            break;
        case MON: // 월 (1~12)
            strcpy(ptr1, "1-12");
            break;
        case WDAY: // 요일 (0~6) (일요일부터 시작)
            strcpy(ptr1, "0-6");
            break;
        default:
            return 0;
    }
}
```

```
                result = checkMinusCommand(ptr1, run_cycle_index, current_time, increase); // '-' 확인한다
                if (malloc_flag) free(ptr1);

                if (result) return 1; // 유효하면 1 리턴
                else return 0; // 유효하지 않으면 0 리턴
        }
}

int checkMinusCommand(char *lexeme, int run_cycle_index, int current_time, int increase){
        char *ptr1;
        char *ptr2;
        int start;
        int end;
        int max_value;
        int i;
        int count;

        //printf("check minue command %s₩n", lexeme); ///////////////////////

        if ((ptr2 = strstr(lexeme, "-")) == NULL) { // '-'가 실행주기에 들어있지 않다면
                return checkNumberAndStarCommand(lexeme, current_time); // 다음 단계로 이동
        } else {
                ptr1 = lexeme; // '-' 앞쪽의 실행 주기 ptr1에
                *ptr2 = '₩0';
                ++ptr2; // '-' 뒤의 실행 주기 ptr2에
```

```
start = atoi(ptr1); // 앞쪽 실행 주기 정수형으로 변환
end = atoi(ptr2); // 뒤쪽 실행 주기 정수형으로 변환

switch(run_cycle_index) { // 실행 주기 확인을 위해 각 실행 주기 별 최대값 저장
        case MIN: // 분 (0~59)
                max_value = 59;
                break;
        case HOUR: // 시 (0~23)
                max_value = 23;
                break;
        case DAY: // 일 (0~31)
                max_value = 31;
                break;
        case MON: // 월 (1~12)
                max_value = 12;
                break;
        case WDAY: // 요일 (0~6) (일요일부터 시작)
                max_value = 6;
                break;
        default:
                return 0;
}

i = start - 1;
count = 0;
do {
        ++i; // '-'앞쪽의 시간부터 뒤쪽의 시간까지 이동하며 확인
```

```
                ++count; // '/' 뒤쪽의 실행주기를 이용해 명령어 실행할 순서인지 확인하는데 사용
                if (i > max_value) { // i 가 실행주기 최댓값을 넘어섰다면
                        // 최솟값으로 바꿔준다
                        if (run_cycle_index == MON) i = 1;
                        else i = 0;
                }

                if (count % increase == 0 && i == current_time) return 1;
        } while (i != end);

        return 0;
    }
}

int checkNumberAndStarCommand(char *lexeme, int current_time){
    int time;

    //printf("check number and star command %s₩n", lexeme); ////////////////////////
    if (!strcmp(lexeme, "*")) return 1; // '*'이면 항상 유효하므로 1 리턴

    time = atoi(lexeme);
    if (time == current_time) return 1; // 숫자가 현재 시간과 일치하면 유효하므로 1 리턴

    //printf("check number end₩n");//////////////////////////

    return 0;
}
```

```
char *commaStrtok(char *start) { // ssu_crontab의 commaStrtok함수와 동일
	static char *next_start;
	char *prev_start;
	int i;
	int length;

	//printf("commastrtok start₩n");//////////////////

	if (start != NULL) {
		next_start = start;
		prev_start = start;
	} else {
		prev_start = next_start;
	}

	if (next_start == NULL) return NULL;

	length = strlen(next_start);
	for(i = 0; i < length; ++i) {
		if (next_start[i] == ',') break;
	}

	if (i < length) {
		next_start[i] = '₩0';
		if (i + 1 < length)
			next_start = next_start + i + 1;
```

```
            else next_start = NULL;
        } else {
            next_start = NULL;
        }

        //printf("commastrtok end₩n"); ////////////////////

        return prev_start;
}
```

<ssu_rsync.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <utime.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/time.h>

#define SECOND_TO_MICRO 1000000
void ssu_runtime(struct timeval *begin_t, struct timeval *end_t);
struct timeval begin_t, end_t;
time_t exctime; // 프로세스 시작 시간 저장할 변수 (log 출력에 이용)

#define BUFFER_SIZE 1024
#define COMMAND_BUFFER_SIZE 4096 // tar 명령어 저장할 버퍼 사이즈

const char *LOG_FILE_NAME = "ssu_rsync_log";
const char *TEMP_PATH_NAME = "tmp_for_sigint_20160548"; // SIGINT 발생 시 원상 복구를 위해 기존의 파일들을 백업해 둘 디렉토리 이름
const char *TEMP_LOG_FILE = "tmp_for_delete_log_20160548"; // -m 옵션 사용 시 삭제된 파일들의 로그들을 임시로 기록해 둘 임시 파일 이름
const char *TAR_FILE_NAME = "tmptar20160548.tar"; // -t 옵션 사용 시 생성될 tar 파일의 이름
```

char *execute_path; // 프로세스 실행 위치의 절대경로 저장할 포인터

char *src_path; // src 디렉토리 절대 경로 저장할 포인터

char *dst_path; // dst 디렉토리 절대 경로 저장할 포인터

char *src; // 프로세스 실행 시 전달된 src 디렉토리 혹은 파일 이름 저장할 포인터

char *dst; // 프로세스 실행 시 전달된 dst 디렉토리 혹은 파일 이름 저장할 포인터

char *temp_dir_absolute_path; // 임시로 사용할 디렉토리의 절대경로 저장할 포인터

void do_tOption(const char *src_path_name, const char *dst_path_name); // -t 옵션 수행하는 함수

void do_mOption(FILE *tmp_log_fp, const char *path_name, const char *src_path_name, const char *dst_path_name, int sync_dir_flag); // -m 옵션 수행하는 함수

void printUsage(const char *process_name); // 프로세스 사용법 출력하는 함수

void checkProcessArguments(int argc, char *argv[]); // 프로세스에 전달된 인자들이 유효한지 확인하는 함수

void syncDirectory(const char *src_path_name, const char *dst_path_name, int sync_dir_flag); // 디렉토리를 동기화하는 함수

void syncFile(const char *src_file_name, const char *dst_path_name); // 파일을 동기화하는 함수

void copy(const char *src, const char *dst); // 파일을 복사하는 함수

void removeDirectory(FILE *tmp_log_fp, const char *path_name, const char *target); // 디렉토리와 그 디렉토리 하위의 모든 파일을 삭제하는 함수

void printLog(FILE *tmp_log_fp, const char *src_path_name, int sync_dir_flag); // 로그를 출력하는 함수

void printFileNameAndSizeAtLogFile(FILE * fp, const char *src_path_name, const char *path_name, int sync_dir_flag); // 동기화된 파일의 이름과 사이즈를 로그파일에 출력하는 함수

int checkSyncTarget(const char *src_file_name, const char *dst_path_name); // 동기화 대상 파일인지 확인하는 함수

int r_option;

int t_option;

int m_option;

static void sigint_during_sync_handler(int signo); // 동기화 중 발생한 SIGINT의 핸들러, 동기화를 취소하고, 기존의 내용으로 복원한다

```
int main(int argc, char *argv[]) { ///////// exit(1)들 sigint로 바꾸기
	FILE *tmp_log_fp; // 임시 로그 파일의 디스크립터
	struct sigaction sig_act;
	sigset_t sig_set;
	gettimeofday(&begin_t, NULL); // 시작 시간 기록
	exctime = time(NULL);

	execute_path = getcwd(NULL, 0); // 프로세스 실행 위치의 절대경로 구해서 저장
	umask(0);
	checkProcessArguments(argc, argv); // 프로세스에 전달된 인자들이 유효한지 확인

	// 파일 동기화 중 SIGINT 전달 될 때를 대비한 임시 파일 저장 디렉토리 초기화
	if (chdir(TEMP_PATH_NAME) == 0) {  // 임시 파일 저장 디렉토리가 이미 존재한다면 삭제하고 다시 생성한다
		chdir(execute_path);
		removeDirectory(NULL, NULL, TEMP_PATH_NAME);
		rmdir(TEMP_PATH_NAME);
	}
	mkdir(TEMP_PATH_NAME, 0777);
	temp_dir_absolute_path = realpath(TEMP_PATH_NAME, NULL); // 임시 파일 저장 디렉토리의 절대 경로를 구해놓는다

	syncDirectory(dst_path,temp_dir_absolute_path, 1); // 기존의 동기화 디렉토리의 모든 파일들을 임시 파일 저장 디렉토리로 백업한다
	// SIGINT 핸들러 등록한다
	sigemptyset(&sig_act.sa_mask);
	sig_act.sa_flags = 0;
	sig_act.sa_handler = sigint_during_sync_handler;
```

```
	if (sigaction(SIGINT, &sig_act, NULL) != 0) {
		fprintf(stderr, "sigaction error₩n");
		exit(1);
	}

	if ((tmp_log_fp = fopen(TEMP_LOG_FILE, "w+")) == NULL) { // 임시 로그 파일 생성한다
		fprintf(stderr, "fopen error for %s₩n", TEMP_LOG_FILE);
		raise(SIGINT);
	}


	// 동기화
	if (t_option){ // -t 옵션이 지정된 경우
		if (src_path) { // src 인자가 디렉토리인 경우
			do_tOption(src_path, dst_path); // 디렉토리에 대해 toption 수행
		} else { // src 인자가 일반 파일인 경우
			do_tOption(src, dst_path); // 파일에 대해 toption 수행
		}
	} else {
		if (src_path) { // src 인자가 디렉토리인 경우
			syncDirectory(src_path, dst_path, r_option); // 디렉토리에 대해 동기화 수행
			if (m_option) do_mOption(tmp_log_fp, "", src_path, dst_path, r_option); // -m 옵션 수행 (동기화 되지 않은 파일들 삭제)
		} else { // src 인자가 일반 파일인 경우
			syncFile(src, dst_path); // 파일에 대해 동기화 수행
			if (m_option) do_mOption(tmp_log_fp, "", src, dst_path, r_option); // -m 옵션 수행 (동기화 되지 않은 파일들 삭제)
		}
```

```
}

// 이제 동기화 끝났으므로 기본 핸들러로 바꾼다
sig_act.sa_handler = SIG_DFL;
if (sigaction(SIGINT, &sig_act, NULL) != 0) {
        fprintf(stderr, "sigaction error₩n");
        raise(SIGINT);
}

chdir(execute_path);
if (src_path) { // src 인자가 디렉토리인 경우
        printLog(tmp_log_fp, src_path, r_option); // 로그 출력
} else { // src 인자가 일반 파일인 경우
        printLog(tmp_log_fp, src, r_option); // 로그 출력
}

chdir(execute_path);
fclose(tmp_log_fp); // 임시 로그 파일 닫는다
unlink(TEMP_LOG_FILE); // 임시 로그 파일 삭제한다
removeDirectory(NULL, NULL, TEMP_PATH_NAME); // 임시 백업 디렉토리 내부 모든 파일들을 삭제한다
rmdir(TEMP_PATH_NAME); // 임시 백업 디렉토리 삭제한다

// 기타 동적할당 했던 문자열들 해제한다
if (src_path) free(src_path);
free(dst_path);
free(execute_path);
```

```
	gettimeofday(&end_t, NULL); // 종료 시간 기록
	ssu_runtime(&begin_t, &end_t); // 프로그램 실행 시간 계산, 출력

	exit(0);
}

void do_tOption(const char *src_path_name, const char *dst_path_name){
	DIR *dp;
	struct dirent *dirp;
	struct stat statbuf;
	char *current_path;
	char tar_command[COMMAND_BUFFER_SIZE];
	const char *src_relative_path;
	int i;
	int do_sync_flag = 0;

	if (access(TAR_FILE_NAME, F_OK) == 0) unlink(TAR_FILE_NAME); // 이전에 생성했던 tar파일이 남아있다면 삭제한다

	sprintf(tar_command, "tar cvf %s ", TAR_FILE_NAME); // tar 명령어를 만든다

	if (stat(src_path_name, &statbuf) < 0) { // 파일의 종류를 알아내기 위해 stat함수 사용
		fprintf(stderr, "stat error for %s₩n", src_path_name);
		raise(SIGINT);
	}

	// src가 일반 파일인 경우
	if (!S_ISDIR(statbuf.st_mode)) {
```

```
		// src파일의 상대경로 명 구한다
		src_relative_path = src_path_name;
		for (i = strlen(src_path_name) - 1; i >= 0; --i) {
			if (src_path_name[i] == '/') {
				src_relative_path = src_path_name + i + 1;
				break;
			}
		}

		if(checkSyncTarget(src_relative_path, dst_path_name)){ // 해당 파일이 동기화 대상인지 (동일한 파일이 동기화 디렉토리 내에 없는지) 확인한다
			// tar 명령어 생성
			strcat(tar_command, src_relative_path);
			strcat(tar_command, "> /dev/null"); ////////////////////////////////////////////////
			system(tar_command);
			// 묶음 해제 명령어 생성
			sprintf(tar_command, "tar xvf %s -C %s", TAR_FILE_NAME, dst_path_name);
			strcat(tar_command, "> /dev/null"); ////////////////////////////////////////////////
			system(tar_command);
			unlink(TAR_FILE_NAME); // 사용한 tar 파일은 삭제한다
			return;
		} else {
			return;
		}

	}
```

```
// src가 디렉토리인 경우
if ((dp = opendir(src_path_name)) == NULL) {
        fprintf(stderr, "opendir error for %s₩n", src_path_name);
        raise(SIGINT);
}

current_path = getcwd(NULL, 0);
chdir(src_path_name);

if (access(TAR_FILE_NAME, F_OK) == 0) unlink(TAR_FILE_NAME); // 해당 디렉토리 내부에 이전에 생성했던 tar파일이 남아있다면 삭제한다

while ((dirp = readdir(dp)) != NULL) { // 디렉토리 내의 모든 파일들을 확인한다
        if (dirp->d_ino == 0) continue;
        if (!strcmp(dirp->d_name, ".") || !strcmp(dirp->d_name, "..")) continue;

        if (stat(dirp->d_name, &statbuf) < 0) {
                fprintf(stderr, "stat error for %s₩n", dirp->d_name);
                continue;
        }

        if (S_ISDIR(statbuf.st_mode)) { // 해당 파일이 디렉토리라면
                ; // 스킵
        } else { // 디렉토리가 아닌 파일이라면
                if(checkSyncTarget(dirp->d_name, dst_path_name)) {// 동기화 해야 하는 파일이라면
                        do_sync_flag = 1;
                        // 명령어 뒤에 해당 파일 이름 덧붙임
```

```
                                strcat(tar_command, dirp->d_name);
                                strcat(tar_command, " ");
                        }
                }
        }

        if (do_sync_flag) { // 동기화 해야 하는 파일이 존재한다면
                // tar 명령어 생성
                strcat(tar_command, "> /dev/null"); ////////////////////////////////////////////////
                system(tar_command);
                // 묶음 해제 명령어 생성
                sprintf(tar_command, "tar xvf %s -C %s", TAR_FILE_NAME, dst_path_name);
                strcat(tar_command, "> /dev/null"); ////////////////////////////////////////////////
                system(tar_command);
                unlink(TAR_FILE_NAME);
        }

        chdir(current_path);
        free(current_path);
}

void printLog(FILE *tmp_log_fp, const char *src_path_name, int sync_dir_flag){
        FILE *log_fp;
        time_t current_time;
        char *time_str;
        char *option;
        char tmp_log_buffer[BUFFER_SIZE];
```

```
if ((log_fp = fopen(LOG_FILE_NAME, "a")) == NULL) { // 로그파일 오픈
        fprintf(stderr, "fopen error for %s₩n", LOG_FILE_NAME);
        exit(1);
}

// 어떤 옵션이 지정되었는지 저장
if (m_option) option = "-m";
else if (r_option) option = "-r";
else if (t_option) option = "-t";
else option = "₩0";

// 현재 시간 구한다
current_time = time(NULL);
time_str = ctime(&current_time);
time_str[strlen(time_str) -1] = '₩0';

fprintf(log_fp, "[%s] ssu_rsync %s %s %s₩n", time_str, option, src, dst); // 로그 첫 부분에 출력할 문자열 생성
chdir(src_path_name);
printFileNameAndSizeAtLogFile(log_fp, src_path_name, "", sync_dir_flag); // 동기화 된 파일들의 이름과 사이즈를 출력한다
chdir(execute_path);

fseek(tmp_log_fp, 0, SEEK_SET); // 임시 로그파일을 확인한다
while (fgets(tmp_log_buffer, sizeof(tmp_log_buffer), tmp_log_fp) != NULL) {
        fprintf(log_fp, "%s", tmp_log_buffer); // 임시 로그파일에 기록되어있던 삭제 로그들을 로그파일에 옮겨 출력한다
}
```

```
	fclose(log_fp);
}

void printFileNameAndSizeAtLogFile(FILE * fp, const char *src_path_name, const char *path_name, int sync_dir_flag){
	DIR *dp;
	struct dirent *dirp;
	struct stat statbuf;
	struct stat sync_statbuf;
	char cur_path_buf[PATH_MAX];
	char next_src_path[PATH_MAX];
	char dst_path_name[PATH_MAX];
	const char *src_relative_path;
	int i;

	// src가 디렉토리가 아닌 파일이었다면 해당 파일만 출력
	if (stat(src_path_name, &statbuf) < 0) { // 파일의 종류를 알아내기 위해 stat() 호출
		fprintf(stderr, "stat error for %s₩n", src_path_name);
		exit(1);
	}

	if (!S_ISDIR(statbuf.st_mode)) { // 일반 파일이라면
		// src파일의 파일명을 구한다
		src_relative_path = src_path_name;
		for (i = strlen(src_path_name) - 1; i >= 0; --i) {
			if (src_path_name[i] == '/') {
				src_relative_path = src_path_name + i + 1;
```

```
                        break;
                }
        }

        // 동기화 된 경우에만 출력//////////////////////////////////////////////////////////
        //  path_name 을 이용하여 dst 내의 동기화된 파일의 경로를 구하고, 해당 파일의 st_atime을 확인
        // 동기화 된 파일의 경로 만든다
        if (dst_path[strlen(dst_path) - 1] == '/') {
                sprintf(dst_path_name, "%s%s", dst_path, src_relative_path);
        } else {
                sprintf(dst_path_name, "%s/%s", dst_path, src_relative_path);
        }

        if (stat(dst_path_name, &sync_statbuf) < 0) { // 동기화된 파일의 정보를 가져오기 위해 stat() 호출
                fprintf(stderr, "stat error for %s₩n", dst_path_name);
                exit(1);
        }

        // 동기화 디렉토리에 파일을 동기화 할 때 동기화 했다는 사실을 표시하기 위해 atime을 프로세스 실행 시 구했던 exctime으로 수정했음
        if (sync_statbuf.st_atime == exctime) { // 동기화된 파일과 exctime이 일치하면 해당 파일은 이번 실행에서 동기화된 파일임
                fprintf(fp, "₩t%s %ldbytes₩n", src_relative_path, statbuf.st_size); // 로그 출력
        }

        return;
}
```

```
// 디렉토리였다면 하위 모든 파일들 출력
if ((dp = opendir(src_path_name)) == NULL) {
        fprintf(stderr, "opendir error for %s₩n", src_path_name);
        exit(1);
}

while ((dirp = readdir(dp)) != NULL) {
        if (dirp->d_ino == 0) continue;
        if (!strcmp(dirp->d_name, ".") || !strcmp(dirp->d_name, "..")) continue;

        if (stat(dirp->d_name, &statbuf) < 0) {
                fprintf(stderr, "stat error for %s₩n", dirp->d_name);
                continue;
        }

        // 로그에 상대경로명을 출력하기 위해 상대경로 만드는 부분
        if (!strcmp(path_name, "")) sprintf(cur_path_buf, "%s", dirp->d_name); // src 디렉토리 바로 아래의 파일이
라면
        else sprintf(cur_path_buf, "%s/%s", path_name, dirp->d_name); // 하위 디렉토리 내의 파일이라면

        if (S_ISDIR(statbuf.st_mode) && sync_dir_flag) { // -r 옵션이 지정되었고, 디렉토리라면
                // 다음 탐색할 디렉토리의 절대경로 생성
                if (src_path_name[strlen(src_path_name) - 1] == '/') {
                        sprintf(next_src_path, "%s%s", src_path_name, dirp->d_name);
                } else {
                        sprintf(next_src_path, "%s/%s", src_path_name, dirp->d_name);
```

```
                }

                chdir(dirp->d_name);
                printFileNameAndSizeAtLogFile(fp, next_src_path, cur_path_buf, sync_dir_flag); // 다음 디렉토리에 대해서 재귀호출
                chdir("..");
            } else if (!S_ISDIR(statbuf.st_mode)) { // 일반 파일이라면
                // 동기화 된 경우에만 출력//////////////////////////////////////////////////////////////
                // 동기화 된 파일의 절대 경로 생성
                if (dst_path[strlen(dst_path) - 1] == '/') {
                    sprintf(dst_path_name, "%s%s", dst_path, cur_path_buf);
                } else {
                    sprintf(dst_path_name, "%s/%s", dst_path, cur_path_buf);
                }

                if (stat(dst_path_name, &sync_statbuf) < 0) { // 동기화된 파일의 정보를 얻기 위해 stat() 호출
                    fprintf(stderr, "stat error for %s₩n", dst_path_name);
                    exit(1);
                }

                if (sync_statbuf.st_ctime == exctime) { // 이번 프로세스 실행에서 동기화된 파일이라면
                    fprintf(fp, "₩t%s %ldbytes₩n", cur_path_buf, statbuf.st_size); // 로그 메시지 출력
                }

            }
        }
}
```

```
void do_mOption(FILE *tmp_log_fp, const char *path_name,  const char *src_path_name, const char *dst_path_name, int sync_dir_flag){
	DIR *dp;
	struct dirent *dirp;
	struct stat statbuf;
	char *current_path;
	const char *src_relative_path;
	char next_src_path[PATH_MAX];
	char next_dst_path[PATH_MAX];
	char *absolute_path_name_of_target_file;
	int i;
	char cur_path_buf[BUFFER_SIZE];

	if ((dp = opendir(dst_path_name)) == NULL) { // dst 디렉터리 오픈
		fprintf(stderr, "opendir error for %s₩n", dst_path_name);
		raise(SIGINT);
	}

	current_path = getcwd(NULL, 0);
	chdir(dst_path_name);

	// src가 디렉토리가 아닌 파일이었다면 나머지 모든 파일 삭제해야함
	if (stat(src_path_name, &statbuf) < 0) {
		fprintf(stderr, "stat error for %s₩n", src_path_name);
		raise(SIGINT);
	}
```

```
// src가 아닌 일반 파일인 경우
if (!S_ISDIR(statbuf.st_mode)) {
	// src 파일의 상대경로명 구함
	src_relative_path = src_path_name;
	for (i = strlen(src_path_name) - 1; i >= 0; --i) {
		if (src_path_name[i] == '/') {
			src_relative_path = src_path_name + i + 1;
			break;
		}
	}

	while ((dirp = readdir(dp)) != NULL) { // dst 디렉토리 내의 모든 파일에 대해서
		if (dirp->d_ino == 0) continue;
		if (!strcmp(dirp->d_name, ".") || !strcmp(dirp->d_name, "..")) continue;

		if (stat(dirp->d_name, &statbuf) < 0) {
			fprintf(stderr, "stat error for %s₩n", dirp->d_name);
			continue;
		}

		if (tmp_log_fp) { // 삭제 로그를 출력해야 한다면(삭제 로그를 출력해야 할 때 이외에는 tmp_log_fp에 NULL 넣어서 호출)
			// 로그에 상대경로명을 출력하기 위해 상대경로 만드는 부분
			if (!strcmp(path_name, "")) sprintf(cur_path_buf, "%s", dirp->d_name); // src 디렉토리 바로 아래의 파일이라면
			else sprintf(cur_path_buf, "%s/%s", path_name, dirp->d_name); // 하위 디렉토리 내의 파일이라면
```

```
                }

                if (strcmp(src_relative_path, dirp->d_name)) { // 파일명과 일치하지 않는다면
                    if (S_ISDIR(statbuf.st_mode)) { // 디렉토리라면
                        absolute_path_name_of_target_file = realpath(dirp->d_name, NULL); // 삭제할 디렉토리의 절대경로 구함
                        removeDirectory(tmp_log_fp, cur_path_buf, absolute_path_name_of_target_file); // 해당 디렉토리 내의 모든 파일 삭제
                        rmdir(absolute_path_name_of_target_file); // 해당 디렉토리 삭제
                        free(absolute_path_name_of_target_file);
                    } else { // 디렉토리가 아닌 일반 파일이라면
                        unlink(dirp->d_name); // 파일 unlink
                        fprintf(tmp_log_fp, "₩t%s delete₩n", cur_path_buf); // 파일 삭제 로그 출력
                    }
                }
            }

            return;
        }

        // src가 디렉토리인 경우
        while ((dirp = readdir(dp)) != NULL) { // dst 디렉토리 내의 모든 파일에 대해서
            if (dirp->d_ino == 0) continue;
            if (!strcmp(dirp->d_name, ".") || !strcmp(dirp->d_name, "..")) continue;

            if (stat(dirp->d_name, &statbuf) < 0) {
```

```
                fprintf(stderr, "stat error for %s₩n", dirp->d_name);
                continue;
        }

        // src 내부의 파일의 절대경로 생성
        if (src_path_name[strlen(src_path_name) - 1] == '/') {
                sprintf(next_src_path, "%s%s", src_path_name, dirp->d_name);
        } else {
                sprintf(next_src_path, "%s/%s", src_path_name, dirp->d_name);
        }

        // 삭제 로그를 찍어야 한다면
        if (tmp_log_fp) {
                // 로그에 출력할 상대경로 생성
                if (!strcmp(path_name, "")) sprintf(cur_path_buf, "%s", dirp->d_name);
                else sprintf(cur_path_buf, "%s/%s", path_name, dirp->d_name);
        }

        if (access(next_src_path, F_OK) < 0) { // src에 존재하지 않는 파일이었다면 (동기화로 생성된 것이 아닌 파일)
                if (S_ISDIR(statbuf.st_mode)) { // 디렉토리라면
                        absolute_path_name_of_target_file = realpath(dirp->d_name, NULL); // 해당 디렉토리의 절대경로 생성
                        removeDirectory(tmp_log_fp, cur_path_buf, absolute_path_name_of_target_file); // 해당 디렉토리 내의 모든 파일 삭제
                        rmdir(absolute_path_name_of_target_file); // 해당 디렉토리 삭제
                        free(absolute_path_name_of_target_file);
```

```
                } else { // 일반 파일이라면
                        unlink(dirp->d_name); // 해당 파일 unlink
                        fprintf(tmp_log_fp, "₩t%s delete₩n", cur_path_buf); // 로그 출력
                }

                continue;
        }


        if (S_ISDIR(statbuf.st_mode)) { // 동기화 된 해당 파일이 디렉토리라면 내부의 파일들을 검사해봐야 함
                // 옵션에 따라 달라져야 할 부분
                if (sync_dir_flag) { // 하위 디렉토리도 동기화 한 경우엔 해당 디렉토리 내부의 파일들도 확인해봐야 함
                        do_mOption(tmp_log_fp, cur_path_buf, next_src_path, dirp->d_name, sync_dir_flag); // next_src_path는 위에서 생성함

                } else { // 하위 디렉토리는 동기화 하지 않은 경우 - 하위 디렉토리는 동기화 되지 않기 때문에 무조건 지워야 하는 파일임
                        absolute_path_name_of_target_file = realpath(dirp->d_name, NULL); // 해당 디렉토리의 절대경로 생성
                        removeDirectory(tmp_log_fp, cur_path_buf, absolute_path_name_of_target_file); // 해당 디렉토리 내의 모든 파일 삭제
                        rmdir(absolute_path_name_of_target_file); // 해당 디렉토리 삭제
                        free(absolute_path_name_of_target_file);

                        continue;
                }
        } else { // 디렉토리가 아닌 파일이라면
```

```
                continue;
            }
        }

        chdir(current_path);
        free(current_path);

}

static void sigint_during_sync_handler(int signo){
        printf("SIGINT raised during sync₩n"); // SIGINT 전달됐다는 메시지 출력
        chdir(execute_path); // 프로세스 실행된 위치로 chdir
        removeDirectory(NULL, NULL, dst_path); // 원래 상태로 복원하기 위해 동기화 하고 있던 디렉토리 내의 모든 파일들을 삭제함
        rmdir(dst_path); // 동기화 디렉토리 삭제
        rename(temp_dir_absolute_path, dst_path); // 백업해뒀던 기존 동기화 디렉토리를 이용하여 복원함
        //removeDirectory(NULL, NULL, temp_dir_absolute_path);
        //rmdir(temp_dir_absolute_path);

        gettimeofday(&end_t, NULL); // 종료 시간 기록
        ssu_runtime(&begin_t, &end_t); // 프로그램 실행 시간 계산, 출력
        exit(0); // 프로세스 종료
}

void syncDirectory(const char *src_path_name, const char *dst_path_name, int sync_dir_flag){
        DIR *dp;
        struct dirent *dirp;
```

```
struct stat statbuf;
char *current_path;
char next_dst_path[PATH_MAX];

if ((dp = opendir(src_path_name)) == NULL) { // 동기화 해야 할 src 디렉토리 오픈
        fprintf(stderr, "opendir error for %s₩n", src_path_name);
        raise(SIGINT);
}

current_path = getcwd(NULL, 0);
chdir(src_path_name);

while ((dirp = readdir(dp)) != NULL) { // src 디렉토리 내의 모든 파일에 대해서
        if (dirp->d_ino == 0) continue;
        if (!strcmp(dirp->d_name, ".") || !strcmp(dirp->d_name, "..")) continue;

        if (stat(dirp->d_name, &statbuf) < 0) {
                fprintf(stderr, "stat error for %s₩n", dirp->d_name);
                continue;
        }

        if (S_ISDIR(statbuf.st_mode)) { // 해당 파일이 디렉토리라면
                // 옵션에 따라 달라져야 할 부분
                if (sync_dir_flag) { // -r 옵션이 지정되어 있다면
                        // 동기화할 디렉토리의 절대경로명을 생성한다
                        if (dst_path_name[strlen(dst_path_name) - 1] == '/') {
                                sprintf(next_dst_path, "%s%s", dst_path_name, dirp->d_name);
```

```
                } else {
                        sprintf(next_dst_path, "%s/%s", dst_path_name, dirp->d_name);
                }

                if (access(next_dst_path, F_OK) < 0) mkdir(next_dst_path, statbuf.st_mode); // 해당 디렉토리가 동기화 디렉토리 내에 존재하지 않는다면 새로 생성
                syncDirectory(dirp->d_name, next_dst_path, sync_dir_flag); // 해당 디렉토리에 대해 재귀호출
            } else continue;

        } else { // 디렉토리가 아닌 파일이라면
            syncFile(dirp->d_name, dst_path_name); // 해당 파일에 대해 동기화 수행
        }
    }

    chdir(current_path);
    free(current_path);
}

int checkSyncTarget(const char *src_file_name, const char *dst_path_name){
    int i;
    const char *src_relative_path;
    struct stat statbuf;
    struct utimbuf utimebuf;
    char dst_file_name[PATH_MAX];
    off_t src_file_size;

    if (stat(src_file_name, &statbuf) < 0) {
```

```
		fprintf(stderr, "stat error for %s₩n", src_file_name);
		exit(1);
}

// 동기화 할 파일의 정보들을 저장해둔다
src_file_size = statbuf.st_size;
utimebuf.actime = statbuf.st_atime;
utimebuf.modtime = statbuf.st_mtime;

// src 파일의 파일명 구함
src_relative_path = src_file_name;
for (i = strlen(src_file_name) - 1; i >= 0; --i) {
		if (src_file_name[i] == '/') {
			src_relative_path = src_file_name + i + 1;
			break;
		}
}

// dst 디렉토리로 옮겨질 파일의 절대경로 생성
if (dst_path_name[strlen(dst_path_name) - 1] == '/') {
		sprintf(dst_file_name, "%s%s", dst_path_name, src_relative_path);
} else {
		sprintf(dst_file_name, "%s/%s", dst_path_name, src_relative_path);
}

if(access(dst_file_name, F_OK) < 0) { // 동기화 디렉토리에 해당 파일이 존재하지 않는다면
		return 1;
```

```
	} else { // dst 디렉토리에 이미 이름이 동일한 파일이 존재하는 경우
		if (stat(dst_file_name, &statbuf) < 0) { // 해당 파일의 정보를 가져온다
			fprintf(stderr, "stat error for %s₩n", dst_file_name);
			exit(1);
		}
		if (statbuf.st_mtime == utimebuf.modtime && statbuf.st_size == src_file_size) return 0; // 동기화 디렉토리 내의 파일과 동기화할 파일의 최종 수정시간, 파일 크기가 동일하면 같은 파일이므로 새롭게 동기화 할 필요가 없다. 따라서 0리턴
	}

	return 1; // 여기까지 왔으면 동기화 해야 하는 파일이다. 1 리턴
}


void syncFile(const char *src_file_name, const char *dst_path_name){
	int i;
	const char *src_relative_path;
	struct stat statbuf;
	struct utimbuf utimebuf;
	char dst_file_name[PATH_MAX];
	off_t src_file_size;

	if (stat(src_file_name, &statbuf) < 0) {
		fprintf(stderr, "stat error for %s₩n", src_file_name);
		raise(SIGINT);
	}

	// 해당 파일에 대한 정보들을 저장해 둔다
```

```
src_file_size = statbuf.st_size;
utimebuf.actime = statbuf.st_atime;
utimebuf.modtime = statbuf.st_mtime;

// src 파일의 상대경로명 구함
src_relative_path = src_file_name;
for (i = strlen(src_file_name) - 1; i >= 0; --i) {
        if (src_file_name[i] == '/') {
                src_relative_path = src_file_name + i + 1;
                break;
        }
}

// dst 디렉토리로 옮겨질 파일의 절대경로 생성
if (dst_path_name[strlen(dst_path_name) - 1] == '/') {
        sprintf(dst_file_name, "%s%s", dst_path_name, src_relative_path);
} else {
        sprintf(dst_file_name, "%s/%s", dst_path_name, src_relative_path);
}

if(access(dst_file_name, F_OK) < 0) { // 동기화 디렉토리에 해당 파일이 존재하지 않는다면
        //printf("copy %s to %s₩n", src_file_name, dst_file_name); ////////////////////////////////////////////////////
        copy(src_file_name, dst_file_name); // 파일 복사하여 동기화 수행
} else { // dst 디렉토리에 이미 이름이 동일한 파일이 존재하는 경우
        if (stat(dst_file_name, &statbuf) < 0) {
                fprintf(stderr, "stat error for %s₩n", dst_file_name);
                raise(SIGINT);
```

```
                }
                if (statbuf.st_mtime == utimebuf.modtime && statbuf.st_size == src_file_size) return; // 동일한 파일이 이미 존재한다면 동기화가 필요 없으므로 리턴

                // 이름만 같고 서로 다른 파일이라면
                //printf("copy %s to %s₩n", src_file_name, dst_file_name); ////////////////////////////////////////////////////
                remove(dst_file_name); // 동일한 이름의 기존 파일 삭제
                copy(src_file_name, dst_file_name); // 복사하여 동기화 수행
        }
}

void printUsage(const char *process_name){
        fprintf(stderr, "usage : %s <option> <src> <dst>₩n", process_name); // 사용법 출력
}

void checkProcessArguments(int argc, char *argv[]) { // 옵션 주어졌을 경우 제대로 동작 안함
        struct stat statbuf;

        // 전달인자 수 잘못 됐을 때
        if (argc < 3) {
                printUsage(argv[0]);
                exit(1);
        }

        if (argv[1][0] == '-') { // 옵션이 전달됐을 때
                switch(argv[1][1]) { // 어떤 옵션인지 판별
                        case 'r':
```

```
                r_option = 1;
                break;
            case 't':
                t_option = 1;
                break;
            case 'm':
                m_option = 1;
                break;
            default:
                printUsage(argv[0]);// 잘못된 옵션 전달됐으면 사용법 출력하고
                exit(1); // 프로세스 종료
        }
        // 전달인자로 전달된 src와 dst 문자열 저장
        src = argv[2];
        dst = argv[3];
} else {
        // 전달인자로 전달된 src와 dst 문자열 저장
        src = argv[1];
        dst = argv[2];
}

// src 인자
// 인자로 입력받은 파일 혹은 디렉토리를 찾을 수 없으면 usage 출력 후 프로그램 종료
if (access(src, F_OK) < 0) {
        printUsage(argv[0]);
        exit(1);
}
```

```
// 인자로 입력받은 파일 혹은 디렉토리의 접근권한이 없는 경우 usage 출력 후 프로그램 종료
if (access(src, R_OK) < 0) {
        printUsage(argv[0]);
        exit(1);
}



// dst 인자
// 인자로 받은 디렉토리를 찾을 수 없으면 usage 출력 후 프로그램 종료
if (access(dst, F_OK) < 0) {
        printUsage(argv[0]);
        exit(1);
}
// 인자로 입력받은 디렉토리가 디렉토리 파일이 아니라면 usage 출력 후 프로그램 종료
if (stat(dst, &statbuf) < 0) {
        fprintf(stderr, "stat error for %s₩n", argv[2]);
        exit(1);
}
if (!S_ISDIR(statbuf.st_mode)) {
        printUsage(argv[0]);
        exit(1);
}
// 인자로 입력받은 디렉토리의 접근권한이 없는 경우 usage 출력 후 프로그램 종료
if (access(dst, R_OK) < 0) {
        printUsage(argv[0]);
        exit(1);
}
```

```
	if (stat(src, &statbuf) < 0) {
		fprintf(stderr, "stat error for %s₩n", src);
		exit(1);
	}

	if (S_ISDIR(statbuf.st_mode)) { // src 인자가 디렉토리라면
		chdir(src);
		src_path = getcwd(NULL, 0); // 해당 디렉토리의 절대 경로 저장
		chdir(execute_path);
	}

	chdir(dst); // dst 인자는 무조건 디렉토리
	dst_path = getcwd(NULL, 0); // dst 디렉토리의 절대경로 저장
	chdir(execute_path);
}

void copy(const char *src, const char *dst) {
	int src_fd;
	int dst_fd;
	char buf[BUFFER_SIZE];
	size_t length;
	struct stat statbuf;
	struct utimbuf utimebuf;

	if ((src_fd = open(src, O_RDONLY)) < 0) { // src파일 오픈
```

```
            fprintf(stderr, "open error for %s₩n", src);
            raise(SIGINT);
        }

        if (stat(src, &statbuf) < 0) {
            fprintf(stderr, "stat error for %s₩n", src);
            raise(SIGINT);
        }

        // 동기화 파일에 저장할 actime 지정, 동기화 됐음을 표시하기 위하여 프로세스 시작 부분에서 저장했던 exctime 을 동기화된 파일의 actime으로 지정한다
        utimebuf.actime = exctime;
        // src 파일의 최종 수정시간 가져와 저장
        utimebuf.modtime = statbuf.st_mtime;

        if ((dst_fd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, statbuf.st_mode)) < 0) { // 동기화 파일 생성
            fprintf(stderr, "open error for %s %s₩n", dst, strerror(errno));
            raise(SIGINT);
        }

        while((length = read(src_fd, buf, BUFFER_SIZE)) > 0) { // 기존의 파일을 새로 생성한 동기화 파일에 복사한다
            write(dst_fd, buf, length);
        }

        utime(dst, &utimebuf); // 동기화 파일의 시간 정보를 수정한다

        close(src_fd);
```

```
        close(dst_fd);
}

void removeDirectory(FILE *tmp_log_fp, const char *path_name, const char *target){
        DIR *dp;
        struct dirent *dirp;
        struct stat statbuf;
        char *current_path;
        char cur_path_buf[PATH_MAX];

        if ((dp = opendir(target)) == NULL) { // 삭제할 디렉토리 open
                fprintf(stderr, "opendir error for %s₩n", target);
                raise(SIGINT);
        }

        current_path = getcwd(NULL, 0);
        chdir(target);

        while ((dirp = readdir(dp)) != NULL) { // 해당 디렉토리 내의 모든 파일에 대해서
                if (dirp->d_ino == 0) continue;
                if (!strcmp(dirp->d_name, ".") || !strcmp(dirp->d_name, "..")) continue;

                if (stat(dirp->d_name, &statbuf) < 0) {
                        fprintf(stderr, "stat error for %s₩n", dirp->d_name);
                        continue;
                }
```

```
		if (tmp_log_fp) { // 삭제 로그를 찍어야 한다면
			// 삭제 로그에 출력할 상대 경로를 생성한다
			if (!strcmp(path_name, "")) sprintf(cur_path_buf, "%s", dirp->d_name);
			else sprintf(cur_path_buf, "%s/%s", path_name, dirp->d_name);
		}

		if (S_ISDIR(statbuf.st_mode)) { // 해당 파일이 디렉토리라면
			removeDirectory(tmp_log_fp, cur_path_buf, dirp->d_name); // 재귀호출하여 해당 디렉토리 내의 모든 파일을 삭제
			rmdir(dirp->d_name); // 해당 디렉토리 삭제
		} else { // 디렉토리가 아닌 파일이라면
			unlink(dirp->d_name); // 해당 파일 unlink
			if (tmp_log_fp) fprintf(tmp_log_fp, "₩t%s delete₩n", cur_path_buf); // 삭제 로그를 출력해야 한다면, 임시 로그 파일에 삭제 로그를 출력
		}
	}

	chdir(current_path);
	free(current_path);
}

void ssu_runtime(struct timeval *begin_t, struct timeval *end_t)
{
	// 시작시간과 종료시간의 차이 계산
	end_t->tv_sec -= begin_t->tv_sec;

	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); // 프로그램 실행에 걸린 시간 출력
}
```