컴퓨터학부 20162448 김병준

1. **개요**

리눅스에는 다양한 시스템 명령어가 존재한다. 시스템 명령어는 운영체제의 이용에 있어서 복잡한 명령들을 단순화 하여 한 줄로 처리할 수 있는 기능을 제공한다. 이러한 명령어들 중에는 운영체제의 최적한 운영 혹은 변경사항에 대한 기록을 위해서 특정한 작업에 대해서 주기적으로 실행하도록 할 수 있다. 대표적인 예시로 메모리 Garbage collection같은 작업이 있다. 리눅스에서는 사용자가 이러한 주기적으로 실행시킬 작업을 등록하는 시스템 명령어인 crontab과 이를 실행시킬 데몬 프로세스 crond가 존재한다. 이 외에도 두개의 사이트(혹은 작업영역)간의 파일과 디렉토리의 동기화 기능을 제공해주는 rsync가 존재한다. 원격 사이트 미러링 및 데이터 백업등의 용도로 활용을 한다. 이번 설계 과제는 이러한 시스템 명령어인 crontab과 rsync, 그리고 crontab에 등록한 명령어들을 처리할 프로세스 crond를 구현하고자 한다.

1. **설계**

이번 설계 프로젝트의 구성은 사용자가 주기적으로 실행하는 명령어를 목록(ssu\_crontab\_file)에 저장 및 삭제하는 프로그램(ssu\_crontab), 목록을 읽어들여 실행 주기에 맞게 명령어를 실행하는 프로그램(ssu\_crond) 그리고 인자로 주어진 src파일 혹은 디렉토리를 dst디렉토리에 동기화하는 프로그램(ssu\_rsync)로 구성된다. 모든 프로그램에 공통적으로 들어가는 헤더, 함수 및 매크로는 common.\*에 정의되어있다. 마찬가지로 cron관련 공통 사용 헤더, 함수, 매크로는 cron\_support.\*, 그리고 이외의 각 프로그램별 고유 사용 헤더, 함수 및 매크로는 ssu\_crontab.\*, ssu\_crond.\*, ssu\_rsync.\*에 정의되어 있다.

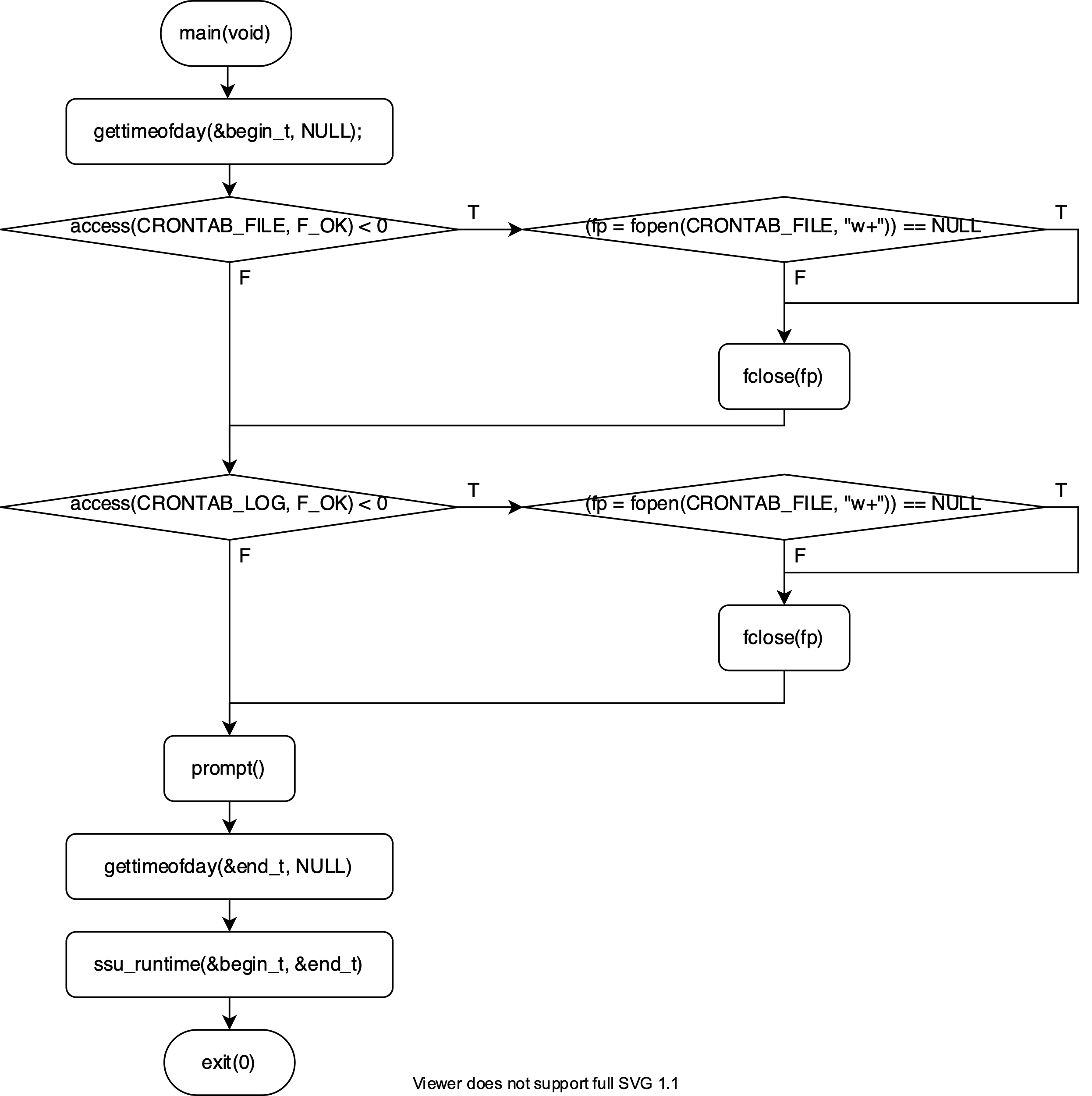
**2-1. Makefile**

Makefile은 기본 컴파일러로 gcc를 사용하며, 각 프로그램에 필요한 오브젝트 파일 생성 시 옵션으로 CFLAGS를 포함한다. 그리고 make명령어를 단독으로 입력했을 때, 모든 프로그램이 컴파일 및 실행 프로그램이 생성되도록 작성하였다. make all을 입력하여도 동일한 결과를 실행한다. 프로그램별 의존성 파일 생성은 make dependency, 재생성은 make new, 생성 파일 삭제는 make clean이다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | .SUFFIXES: .c .o | | .PHONY: dep all new clean | |  | | # Compiler | | CC = gcc | | # Compile option | | # -c: Generate object file | | # -W, -Wall: Print warning about all ambigous grammer | | # -Wextra: Print warning out of -W, -Wall | | # -O2: Optimization | | # -g: Debugging, PLEASE DELETE AFTER PROJECT COMPLETE! | | CFLAGS = -c -W -Wall -Wextra -g $(INC) | |  | | # Execute program file | | CRONTAB = ssu\_crontab | | CROND = ssu\_crond | | RSYNC = ssu\_rsync | | # Source file | | COMMON\_SRCS = common.c | | CRONTAB\_SRCS = ssu\_crontab.c | | CROND\_SRCS = ssu\_crond.c | | RSYNC\_SRCS = ssu\_rsync.c | | CRON\_SUPPORT\_SRCS = cron\_support.c | | SRCS = $(COMMON\_SRCS) $(CRONTAB\_SRCS) $(CROND\_SRCS) $(RSYNC\_SRCS) $(CRON\_SUPPORT\_SRCS) | | # Object file | | COMMON\_OBJS = $(COMMON\_SRCS:.c=.o) | | CRONTAB\_OBJS = $(CRONTAB\_SRCS:.c=.o) | | CROND\_OBJS = $(CROND\_SRCS:.c=.o) | | RSYNC\_OBJS = $(RSYNC\_SRCS:.c=.o) | | CRON\_SUPPORT\_OBJS = $(CRON\_SUPPORT\_SRCS:.c=.o) | | OBJS = $(COMMON\_OBJS) $(CRONTAB\_OBJS) $(CROND\_OBJS) $(RSYNC\_OBJS) $(CRON\_SUPPORT\_OBJS) | | # Library file | | LIBS = | | # Include path | | INC = | |  | | # Execute file grneration | | # $@ = TARGET | | # $^ = DEPENDENCY | | # make all: Make all execute file | | all : $(OBJS) | | $(CC) -o $(CRONTAB) $(COMMON\_OBJS) $(CRONTAB\_OBJS) $(CRON\_SUPPORT\_OBJS) $(LIBS) | | $(CC) -o $(CROND) $(CROND\_OBJS) $(CRON\_SUPPORT\_OBJS) $(LIBS) -lpthread | | $(CC) -o $(RSYNC) $(COMMON\_OBJS) $(RSYNC\_OBJS) $(LIBS) | | $(CRONTAB) : $(COMMON\_OBJS) $(CRONTAB\_OBJS) $(CRON\_SUPPORT\_OBJS) | | $(CC) -o $@ $^ $(LIBS) | | $(CROND) : $(CROND\_OBJS) $(CRON\_SUPPORT\_OBJS) | | $(CC) -o $@ $^ $(LIBS) -lpthread | | $(RSYNC) : $(COMMON\_OBJS) $(RSYNC\_OBJS) | | $(CC) -o $@ $^ $(LIBS) | |  | | # Object file generation | | $(OBJS): | | $(CC) $(CFLAGS) $(SRCS) | |  | | # make dep: Make dependency information file | | dep: | | $(CC) -M $(INC) $(SRCS) > .dependency | |  | | # make new: Re-generation | | new: | | $(MAKE) clean | | $(MAKE) all | |  | | # make clean: Remove all generated file | | clean: | | rm -rf $(OBJS) $(CRONTAB) $(CROND) $(RSYNC) | |

**2-2. SSU\_CRONTAB**

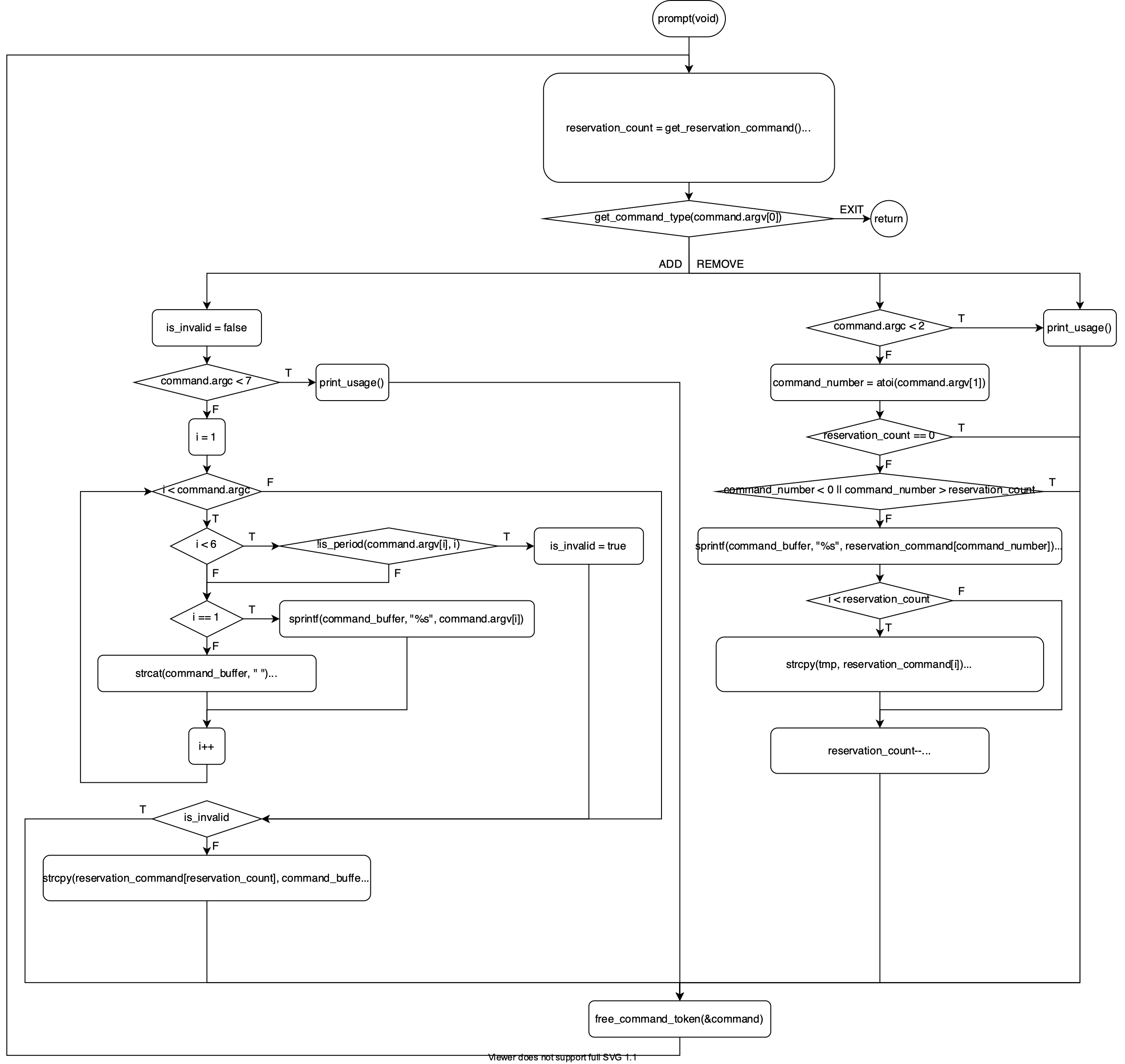
이 프로그램은 사용자가 주기적으로 실행하는 명령어를 목록(ssu\_crontab\_file)을 생성, 저장 및 삭제하는 프로그램이다. 명령어를 저장하기 위해 사용자로부터 명령어를 입력받으며, 이를 위해 프롬프트를 사용자에게 제공한다. 사용자는 프롬프트에서 저장(ADD), 삭제(REMOVE), 종료(EXIT) 명령어를 입력 할 수 있다. 또한 프로그램의 실행 시간을 측정하여 출력한다. ssu\_crontab의 흐름도는 다음과 같다.



[그림 1] ssu\_crontab 프로그램 흐름도

**2-2-1. void prompt(void)**

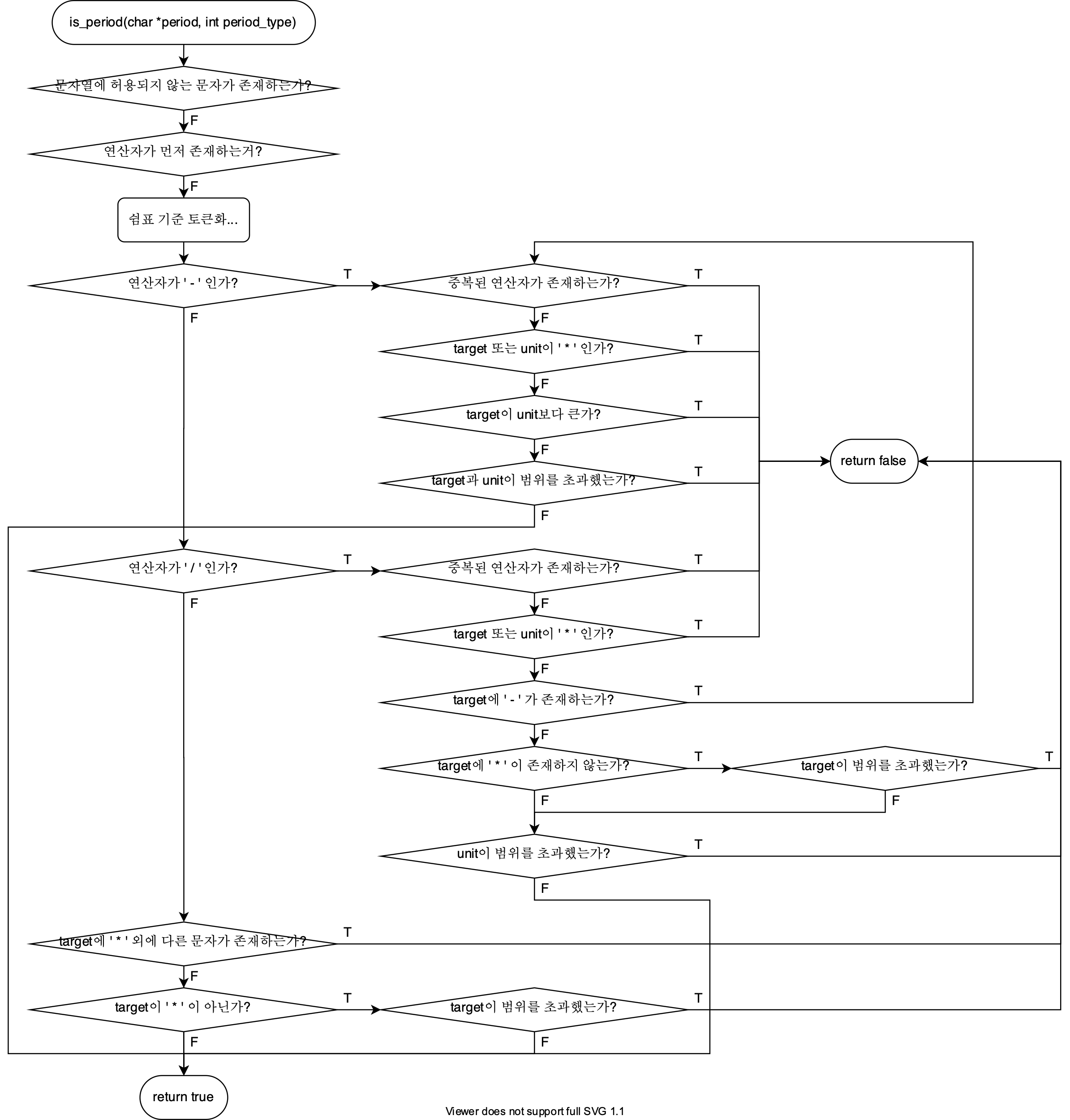
ssu\_crontab에서 제공하는 프롬프트는 사용자가 주기적으로 실행하는 명령어를 저장 혹은 삭제한다. 프롬프트는 시작하면 ssu\_crontab\_file에 저장된 예약 명령어를 읽어들인 후 출력하고, 이후 프롬프트를 출력한다. 그리고 사용자가 입력한 모든 명령행 문자열에 대하여 토큰으로 분리한다. 이후 명령어에 해당하는 작업을 수행한다. 앞의 과정은 프롬프트가 종료되기 전까지 무한적으로 반복한다. prompt의 흐름도는 다음과 같다.



[그림 2] prompt 흐름도

**2-2-2. bool is\_period(char \*period, int period\_type)**

사용자에게 주기적으로 실행할 명령을 입력받으면, 주기가 올바른지 검사하는 함수이다. 정상일 경우 true, 오류를 발견했을 경우 false를 리턴한다. 이 함수를 호출할 때, 각 시간 타입에 해당하는 명령행 문자열 토큰과 시간 타입이 인자로 주어진다. 이후 함수가 호출되면, 기본적으로 문자열 내부에 존재해선 안되는 문자가 존재하는지 판단한다. 이후 쉼표(,) 기준으로 주기를 토큰화 시킨다. 만약 쉼표 앞뒤로 토큰이 비어있다면 오류 처리한다. 쉼표 분리가 완료되었다면, 슬래시(/)와 범위(-)를 기준으로 분리한다. 만약 슬래시와 범위가 동시에 존재한다면 슬래시를 기준으로 분리를 한다. 만약 둘 중에 하나만 존재한다면 존재하는 것을 기준으로 분리한다. 분리가 된 후에는 연산자를 기준으로 앞에 존재하는 문자열은 target, 뒤에 존재하는 문자열은 unit이다. 슬래시를 우선으로 분리하므로, 만약 슬래시가 존재하지 않는다면 operator는 범위가 된다. 이후 unit에 중복된 operator가 존재할 경우, 시작과 끝이 전체(\*)로 끝날 경우, target이 unit보다 클 경우, 시간 타입에 따른 허용 범위를 초과했을 경우를 판별한다. 반대로 만약 슬래시가 존재한다면, 중복된 연산자가 존재할 경우, target이 전체(\*)만 존재하지 않을 경우, unit이 전체(\*)일 경우를 판별한다. 또한 target이 범위일 경우, 범위(-)를 기준으로 다시 한번 분리를 한다. 이후 과정은 앞서 operator가 범위일 경우와 동일하다. 마지막으로 operator가 존재하지 않을 경우 target이 허용 범위를 초과했을 경우를 판별한다. is\_period의 흐름도는 다음과 같다.



[그림 3] is\_period 흐름도

**2-3. SSU\_CROND**

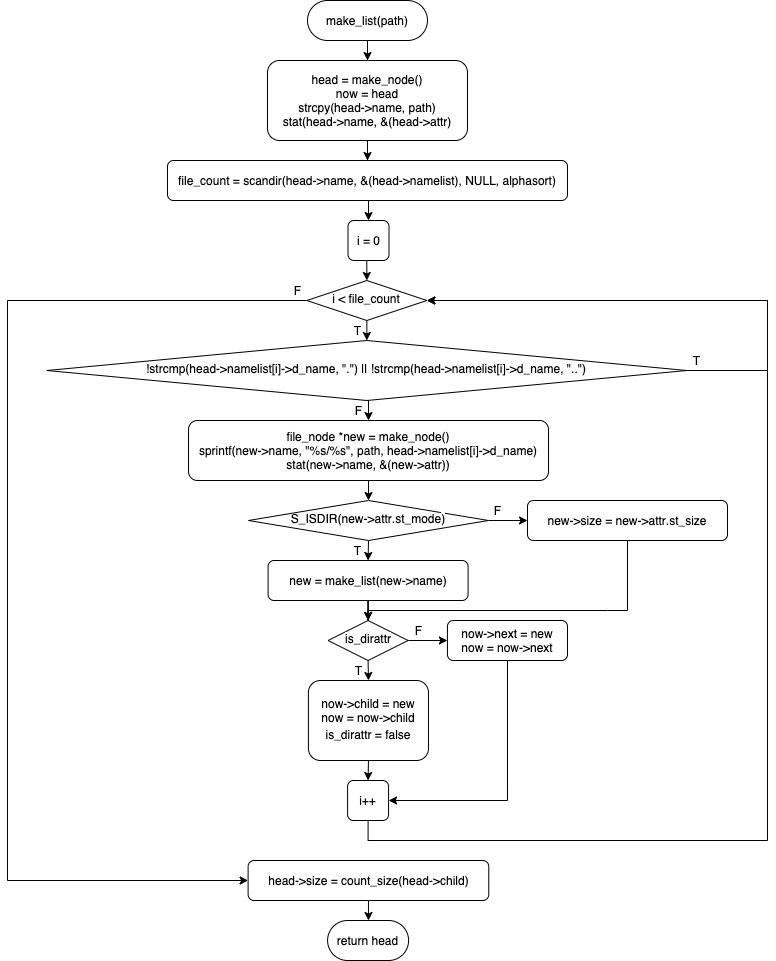
**2-3-2. file\_node 구조체**

하나의 노드가 가져야하는 정보는 경로, 속성, 크기, 변경상태, 디렉토리의 경우 하위 파일 목록과 첫번째 하위 파일을 가리키는 포인터, 다음 파일을 가리키는 포인터이다..

|  |
| --- |
| typedef struct ssu\_fileNode{ // 모니터링 파일 목록 구조체  char name[BUFFER\_SIZE]; // 파일 경로  struct stat attr; // 파일 상태 정보  struct dirent \*\*namelist; // 디렉토리 경우 하위 파일 목록  struct ssu\_fileNode \*next; // 하위 디렉토리 파일 포인터  struct ssu\_fileNode \*child; // 같은 레벨의 다음 파일 포인터  int size; // 파일 크기  int status; // 모니터링 확인 상태  } file\_node; |

**2-3-3. file\_node \*make\_list(char \*path)**

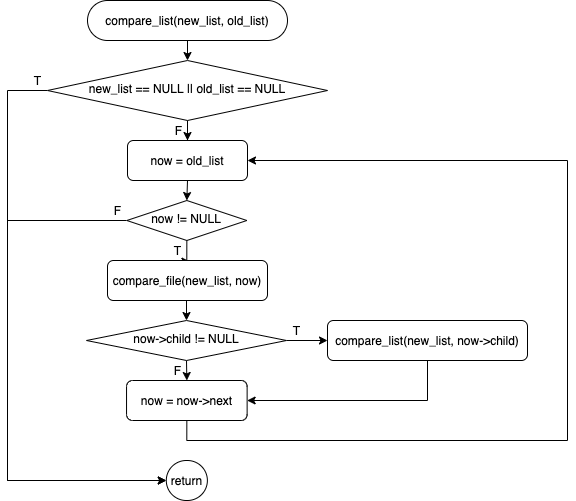
인자로 주어진 path에 해당하는 파일 혹은 디렉토리의 모든 정보를 트리로 만들어서 file\_node 구조체인 루트 노드를 반환한다. 최초 실행 시 노드를 생성하여 루트 노드를 만들어 준다. 생성된 루트 노드에 인자로 주어진 파일 경로와 상태 정보를 저장한다. 이 후 해당 경로 하위에 존재하는 파일들에 대하여 파일일 경우 형제로, 디렉토리일 경우 자식으로 노드를 연결한다. 디렉토리의 경우 재귀적으로 호출하여 반환된 노드를 연결한다. 이 함수의 흐름도는 다음과 같다.

****

[그림 4] make\_list 흐름도

**2-3-4. void compare\_list(file\_node \*new\_list, file\_node \*old\_list)**

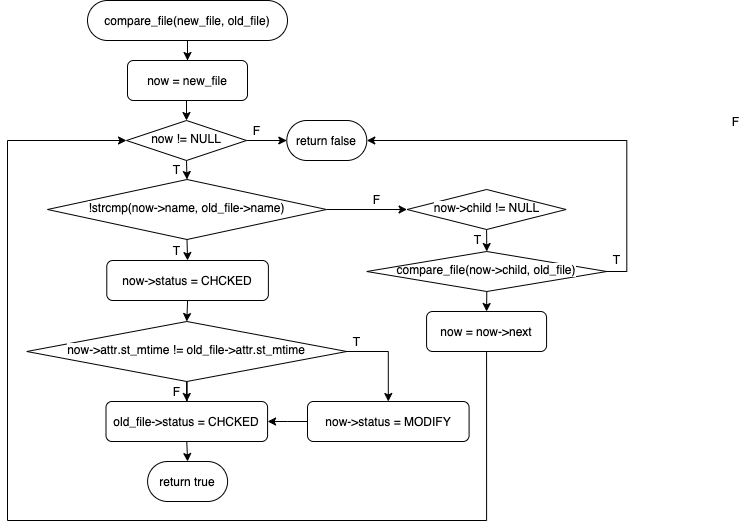
모니터링의 핵심은 이전 상태와 비교했을 때, 무엇이 변경되었는지 파악하는 것이다. 앞서 모니터링 대상 디렉토리에 대하여 우리는 트리를 생성하였다. 이 함수에서는 인자로 주어진 이전 트리와 새로운 트리에 대해서 비교를 한다. 기본적으로 이전 트리의 노드들을 탐색하여 트리를 비교하며, 만약 이전 트리에서 디렉토리 노드를 탐색했을 경우 재귀적으로 하위 파일 노드를 인자로 주어 호출한다. 이 함수의 흐름도는 다음과 같다.



[그림 5] compare\_list 흐름도

**2-3-5. void compare\_file(file\_node \*new\_file, file\_node \*old\_file)**

인자로 주어진 두 파일에 대하여 비교를 해주는 함수이다. 이전 파일 정보를 가지고 새로운 트리를 탐색하여 같은 경로의 파일이 존재하는지 확인한다. 존재하지 않을 경우 이전 트리에 존재하는 파일의 status는 UNCHCK상태이다. 그러나 같은 경로의 파일이 새로운 트리에서도 탐색이 되었을 경우, 새로운 트리에 존재하는 해당 노드는 status가 CHCKED상태가 된다. 이후 최종 수정 시간을 비교하였을 때 변경이 되었으면 이전 트리에 존재하는 파일의 status는 MODIFY가 된다. 아니라면 CHCKED가 된다. 이 함수의 흐름도는 다음과 같다.



[그림 6] compare\_file 흐름도

**2-3-5. change\_file 구조체**

변경 사항을 기록하기 위해 필요한 정보는 변경 시간, 상태, 파일 이름이다.

|  |
| --- |
| typedef struct ssu\_changeItem { // 변경사항 구조체  time\_t time; // 변경 시간  char name[BUFFER\_SIZE]; // 파일 이름  int status; // 변경 상태  } change\_file; |

**2-3-6. int write\_change\_list(file\_node \*head, int idx, int status)**

인자로 주어진 트리에 대하여 변경 사항을 체크하고 수집하여 변경 목록 change\_list에 저장한다. status가 UNCHCK인 경우 인자로 준 상태로 저장하고 현재 시간을 time에 저장한다. MODIFY인 경우 상태 그대로 저장하며 해당 파일의 st\_mtime을 time에 저장한다. 마찬가지로 디렉토리 노드를 탐색했을 경우 하위 파일 노드를 인자로 주어 재귀적으로 호출한다. 이 함수의 흐름도는 다음과 같다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

[그림 7] write\_change\_list 흐름도

**2-3-7. void write\_change\_log(int idx)**

수집 및 저장한 변경 사항을 요구사항으로 주어진 양식에 맞추어 로그파일에 저장한다. 이 함수의 흐름도는 다음과 같다.

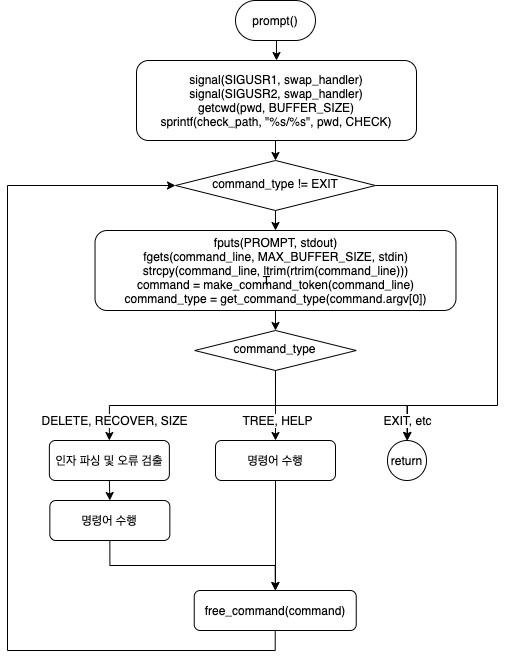
스크린샷이(가) 표시된 사진

자동 생성된 설명

[그림 8] write\_change\_log 흐름도

**2-4. 프롬프트 프로그램(prompt.c)**

프롬프트 프로그램은 사용자에게 기본적인 서비스를 제공할 수 있도록 연결시켜주는 인터페이스로, 사용자는 프롬프트를 통해 모니터링 디렉토리내부의 파일들에 대해 삭제(DELETE), 복원(RECOVER), 정보 출력(SIZE, TREE), 도움말(HELP), 종료(EXIT) 기능을 수행 할 수 있다. 이 외에 명령 혹은 올바르지 않은 입력에 대해서는 도움말을 출력하고 다시 입력을 받는다. 개행만 입력 받았을 경우 프롬프트는 다시 입력을 받는다. 지정된 명령어를 입력 받았을 경우 해당 명령어의 기능들을 수행하기 위해 입력받은 인자들을 확인하고, 오류를 검출한다. 오류가 존재하지 않을 경우 해당 기능을 수행하고 앞에 서술한 내용을 반복문에서 수행한다. 프롬프트 프로그램의 흐름도는 다음과 같다.



[그림 9] 프롬프트 프로그램 흐름도

**2-4-1. commands 구조체**

명령 수행에 필요한 정보는 입력한 명령행을 분리한 토큰들과 토큰의 개수이다.

|  |
| --- |
| typedef struct ssu\_commandToken { // 프롬프트 명령어 구조체  char \*\*argv; // 명령행 토큰  int argc; // 명령행 인자 개수  } commands; |

**2-4-2. commands make\_command\_token(char \*command\_line)**

프롬프트에 입력한 명령 문자열을 공백을 기준으로 토큰들로 구분 및 저장 하고 토큰의 개수를 카운트하는 구조체이다. 이 함수의 흐름도는 다음과 같다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

[그림 10] make\_command\_token 흐름도

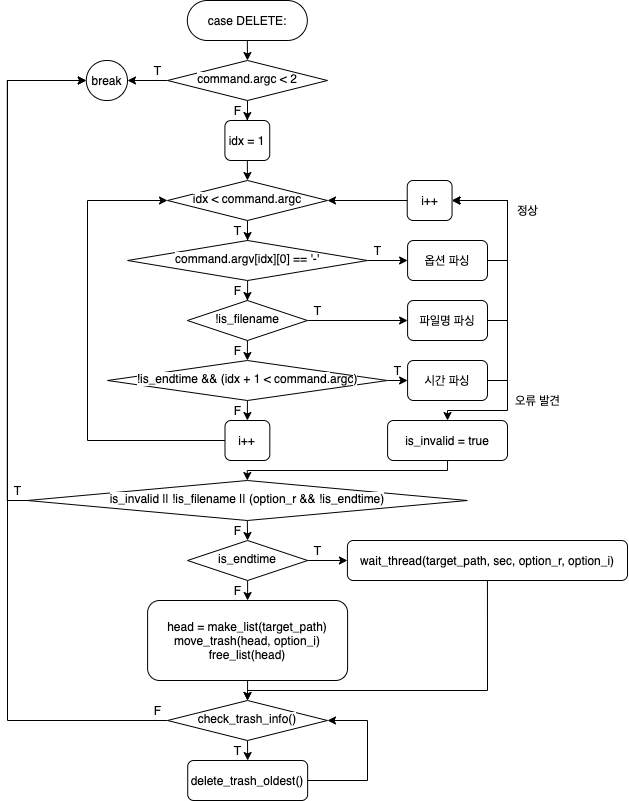
**2-4-3. 삭제(DELETE)**

모니터링 대상 디렉토리에 존재하는 파일들에 대해서 삭제 기능을 수행한다. 옵션을 지정하지 않았을 경우 해당 기능을 수행 하더라도 복원이 가능하다. 기본적으로 명령을 수행하기에 앞서 commands구조체에 저장된 토큰들을 사용하여 인자들을 추출하고, 오류를 검출한다. 인자 및 에러 검출 우선순위로는 인자 개수, 옵션[OPTION], 파일명<FILE\_NAME>, 시간<END\_TIME>순이다. 토큰들에 오류가 존재하지 않을 경우, 사용자가 입력한 파일을 대상으로 삭제 기능을 수행한다. 삭제 대상은 파일과 디렉토리 둘 다 가능하다. 그리고 YYYY-MM-DD HH-MM-SS와 같은 시간 형식을 이용하여 지정한 시간에 삭제가 가능하다. R옵션을 지정할 경우 삭제 예약을 했을 때 지정한 시간이 되면 삭제 확인 응답을 요구한다. 또한 I옵션을 지정할 경우 영구히 삭제되며 복원이 불가능하다. 오류 검출하는 방법은 다음과 같다.

* + - * OPTION
        + command.argv[idx][0] == ‘-‘ 일때 아래의 경우에 포함되지 않을 경우 오류

1. command.argv[idx][1] == ‘r’
2. command.argv[idx][1] == ‘i’
   * + - FILE\_NAME
         * 해당 파일이 존재하지 않는 경우 오류
         * 해당 파일이 존재할 때
3. 모니터링 대상 디렉토리에 존재하지 않는 경우 오류
   * + - END\_TIME
         * 예약한 시간이 현재 시간보다 이전일 경우 오류
         * 시간 형식이 올바르지 않은 경우 오류

이러한 인자 및 에러 검출 과정이 마무리 되고 나면 END\_TIME의유무에 따라 프로그램 대기 함수인 wait\_thread를 호출하거나 바로 트리를 생성하여 삭제 메인 함수인 move\_trash함수를 호출한다. 그리고 마지막으로 휴지통 정보 디렉토리의 크기를 확인하여 2048KB이상일 경우 휴지통에 존재하는 파일들 중 가장 삭제된 지 오래된 파일을 제거한 뒤 반복적으로 확인한다. 삭제 기능의 전체적인 흐름도는 다음과 같다.



[그림 11] 삭제 기능 흐름도

**2-4-4. void move\_trash(file\_node \*head, int option\_i)**

삭제 기능 메인 함수이다. 인자로 삭제할 파일 혹은 디렉토리의 노드가 주어진다. 처음 함수가 호출되고, 휴지통 디렉토리와 하위 원본, 정보 디렉토리에 존재 유무를 확인 한 뒤, 존재하지 않을 경우 해당 디렉토리를 생성한다. 이후 i옵션이 존재한다면, 인자로 주어진 노드를 통해 휴지통 디렉토리로 삭제하지 않고 바로 링크를 끊는다. 반대로 존재하지 않는다면, 휴지통 디렉토리에 존재하는 중복 파일의 개수를 확인한다. 중복 파일이 1개 이상 존재한다면, NUMBER\_FILE\_NAME과 같은 형식의 이름으로 원본 파일을 휴지통에 이동시킨다. 그리고 마찬가지로 원본 파일의 정보를 담은 파일도 중복 여부에 따라 원본 파일과 같은 형식의 이름으로 삭제한다. 이 함수의 흐름도는 다음과 같다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

**2-4-5. void wait\_thread(char \*path, int sec, int option\_r, int option\_i)**

END\_TIME이 존재할 경우 삭제 예약 프로세스를 생성하는 함수이다. 함수가 처음 호출되면 프로세스를 분리하고 인자로 주어진 sec만큼의 시간을 프로세스 대기한다. 시간이 지나간 뒤 인자로 주어진 경로에 파일이 존재하는지 확인하고 존재할 경우 트리를 생성한다. R옵션이 존재할 경우 부모 프로세스에게 SIGUSR1 시그널을 보내고 사용자에게 질의를 한 뒤 move\_trash함수를 호출한다. 만약 존재하지 않는다면 바로 move\_trash함수를 호출한다. 이후 트리를 초기화하고 부모 프로세스에게 SIGUSR2 시그널을 보내고 프로세스를 종료한다. 이 함수의 흐름도는 다음과 같다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

[그림 12] wait\_thread 흐름도

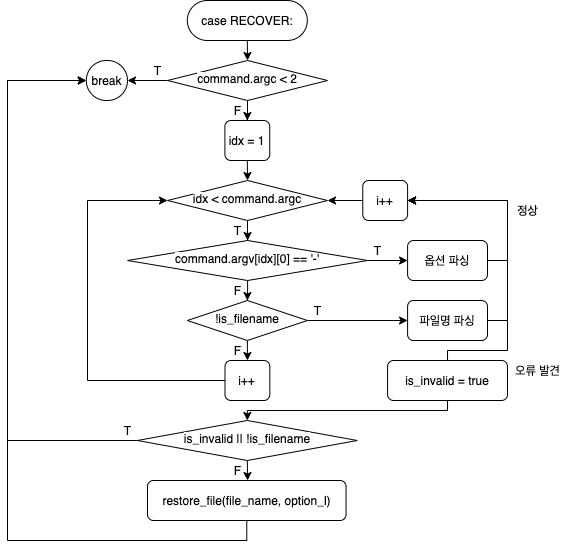
**2-4-6. 복원(RECOVER)**

프롬프트에서 모니터링 대상 디렉토리에서 삭제한 파일들에 대해서 기존 위치로 복원한다. 삭제 기능 사용 시 I 옵션을 사용하였다면 복원이 불가능하다. 삭제 기능과 마찬가지로 commands구조체에 저장된 토큰들을 사용하여 인자들을 추출하고, 오류를 검출한다. 인자 및 에러 검출 우선순위로는 인자 개수, 옵션[OPTION], 파일명<FILE\_NAME>순이다. 토큰들에 오류가 존재하지 않을 경우, 사용자가 입력한 파일을 대상으로 복원 기능을 수행한다. 복원 대상은 파일과 디렉토리 둘 다 가능하다. L옵션을 주었을 경우 휴지통 디렉토리에 존재하는 삭제된 순서대로 화면에 출력 후 명령을 진행한다. 오류 검출하는 방법은 다음과 같다.

* + - * OPTION
        + command.argv[idx][0] == ‘-‘ 일때 아래의 경우에 포함되지 않을 경우 오류

1. command.argv[idx][1] == ‘l’
   * + - FILE\_NAME
         * 해당 파일이 존재하지 않는 경우 오류
         * 해당 파일이 존재할 때
2. 모니터링 대상 디렉토리에 존재하지 않는 경우 오류

이러한 인자 및 에러 검출 과정이 마무리 되고 나면 FILE\_NAME에 해당하는 파일을 휴지통 디렉토리에서 복원한다. 복원 기능의 전체적인 흐름도는 다음과 같다.



[그림 13] 복원 기능 흐름도

**2-4-7. file\_infos 구조체**

파일 복원에 필요한 파일에 정보는 삭제되기 전 경로, 삭제 시간, 수정 시간이다.

|  |
| --- |
| typedef struct ssu\_fileinfo {  int num;  char path[BUFFER\_SIZE];  struct tm d\_tm; // 삭제 시간  struct tm m\_tm; // 수정 시간  } file\_infos; |

**2-4-8. void restore\_file(const char \*file\_name, int option\_l)**

파일 복원 기능 메인 함수이다. 인자로 복원할 파일의 이름과 L옵션 유무가 주어진다. 함수가 호출되면, L옵션이 존재할 경우 휴지통 정보 디렉토리를 탐색한다. 안에 존재하는 파일들을 모두 읽어서 정보들을 file\_info배열에 저장한다. 이후 배열에 존재하는 파일 정보들에 대해 삭제된 시간 순으로 정렬한 뒤 출력한다. 이후 본격적인 복원 작업을 진행한다. 복원 기능 수행 시 복원할 파일의 우선 순위는 다음과 같다.

1. 휴지통 디렉토리에 동일한 이름으로 파일이 존재하는 경우
2. N\_FILE\_NAME형식으로 존재하는 경우(이미 중복되는 경우)

휴지통 정보 디렉토리를 탐색하면서, 정보 파일의 이름의 끝에 존재하는 텍스트 파일 확장자를 제거한다. 이후 인자로 주어진 file\_name과 비교한다. 이름이 같을 경우 해당 파일의 내용을 읽어서 파일을 복원한다. 만약 복원할 지점까지의 경로가 존재하지 않을 경우 에러처리 하였다. 또한 이미 동일한 이름의 파일이 복원할 지점에 존재하는 경우 중복된 파일의 수를 파악하여 OVERLAP\_file\_name형식으로 복원할 지점에 복원한다. 복원 하고 나면 함수를 반환한다. 반대로 이름이 다를 경우 현재 탐색하고 있는 파일이 중복된 파일인지 확인한다. 같은 이름의 중복된 파일일 경우 file\_info배열에 해당 정보를 저장한다.

휴지통 정보 디렉토리를 탐색을 끝낼 때 까지 함수가 반환되지 않았다면, file\_info 배열을 중복파일 카운트 순으로 정렬한다. 이후 사용자에게 질의를 통해 복원할 파일을 선택하여 복원한다. 마찬가지로 복원할 지점에 원본 파일 이름으로 동일하게 존재한다면 이전의 경우와 마찬가지로 진행한다.

복원을 마치면 휴지통 디렉토리에 파일 이름에 해당하는 중복된 이름이 여전히 존재하는지 파악하고, 존재한다면 중복 카운트 번호를 재정립해준다. 이 함수의 흐름도는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

[그림 14] restore\_file 흐름도

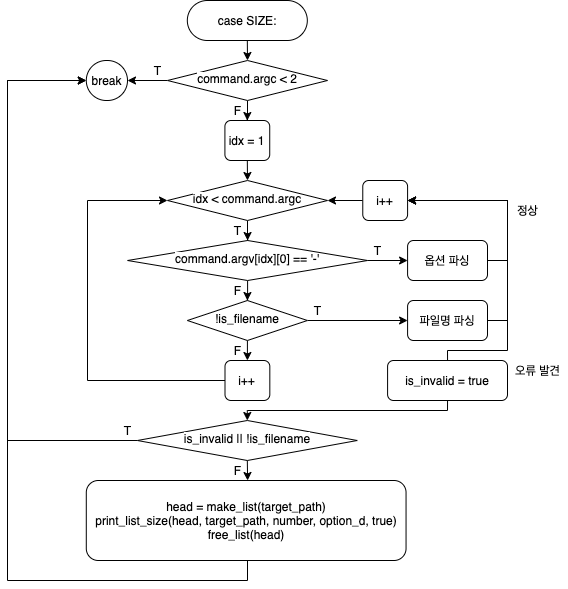
**2-4-9. 크기 정보(SIZE)**

모니터링 대상 디렉토리에 존재하는 파일 중 입력한 FILE\_NAME에 대한 크기 정보를 출력한다. 삭제와 복원 기능과 마찬가지로 명령을 수행하기에 앞서 commands구조체에 저장된 토큰들을 사용하여 인자들을 추출하고, 오류를 검출한다. 인자 및 에러 검출 우선순위로는 인자 개수, 옵션[OPTION], 파일명<FILE\_NAME>순이다. 토큰들에 오류가 존재하지 않을 경우, 사용자가 입력한 파일을 대상으로 크기 정보를 출력한다. D옵션을 지정해주었을 경우, FILE\_NAME에 해당하는 파일이 디렉토리일 경우, 같이 주어진 NUMBER의 깊이만큼 해당 디렉토리의 하위 파일 크기 정보를 모두 출력한다. 오류 검출하는 방법은 다음과 같다.

* + - * OPTION
        + command.argv[idx][0] == ‘-‘ 일때 아래의 경우에 포함되지 않을 경우 오류

1. command.argv[idx][1] == ‘d’
   * + - FILE\_NAME
         * 해당 파일이 존재하지 않는 경우 오류
         * 해당 파일이 존재할 때
2. 모니터링 대상 디렉토리에 존재하지 않는 경우 오류

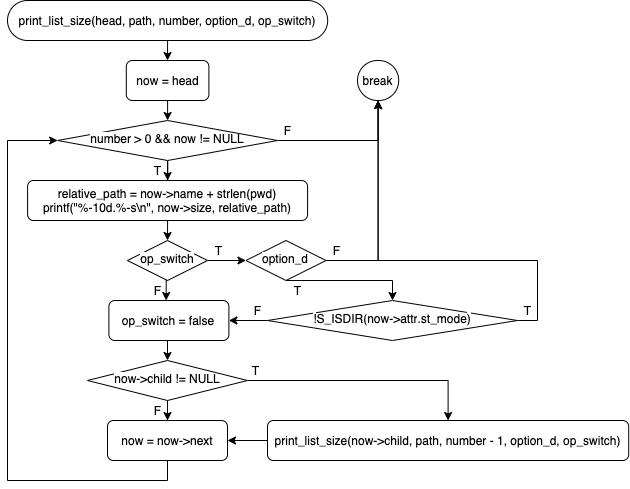
이러한 인자 및 에러 검출 과정이 마무리 되고 나면. FILE\_NAME을 대상으로 하는 트리를 생성한 뒤, 화면에 크기 정보를 출력한다. 크기 정보 출력 기능의 전체적인 흐름도는 다음과 같다.



[그림 15] 크기 정보 출력 기능 흐름도

**2-4-10. void print\_list\_size(file\_node \*head, char \*path, int number, int option\_d, int op\_switch)**

주어진 트리에 대해서 number만큼의 깊이만큼 크기 정보를 출력한다. 이 함수의 흐름도는 다음과 같다.



[그림 16] print\_list\_size 흐름도

**2-4-511 구조 정보(TREE)**

모니터링 대상 디렉토리에 존재하는 파일들 전체를 트리 구조로 출력한다. 트리 구조 출력 기능의 전체적인 흐름도는 다음과 같다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

[그림 17] 트리 구조 출력 기능 흐름도

**2-4-12. void print\_list\_tree(file\_node \*head, int level, int level\_check[], int is\_root)**

주어진 트리에 대해서 현재 level 깊이에 위치한 파일들을 출력한다. 이 함수의 흐름도는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

[그림 18] print\_list\_tree 흐름도

**3. 구현**

**3-1. 메인 프로그램(ssu\_mntr.c)**

**3-1-1. file\_node\* make\_list(char \*path);**

디렉토리 파일 목록 트리화, 반환: 루트 노드

**3-1-2. file\_node \*make\_node(void);**

노드 생성, 반환: 새 노드

**3-1-3. void free\_list(file\_node \*head);**

파일 목록 구조체 메모리 할당 해제

**3-1-4. char \*make\_time\_format(struct tm time);**

시간 형식 문자열 생성, 반환: 시간 형식 문자열

**3-1-5. void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t);**

실행시간 출력

**3-2. 모니터링 프로그램(monitoring.c)**

**3-2-1. void monitoring(void);**

모니터링 메인

**3-2-2. void init\_list\_status(file\_node \*head, int status);**

모니터링 파일 상태 초기화

**3-2-3. void compare\_list(file\_node \*new\_list, file\_node \*old\_list);**

파일 목록 트리 비교

**3-2-4. int compare\_file(file\_node \*new\_file, file\_node \*old\_file);**

파일 정보 비교, 반환: 발견(1), 미발견(0)

**3-2-5. int write\_change\_list(file\_node \*head, int idx, int status);**

변경 사항 목록 작성, 반환: 변경 사항 개수

**3-2-6. void sort\_change\_list(int idx);**

변경사항 목록 시간순 정렬

**3-2-7. void write\_change\_log(int idx);**

변경사항 파일 기록

**3-2-8. void set\_daemon\_process(void);**

데몬프로세스 설정

**3-3. 프롬프트 프로그램(prompt.c)**

**3-3-1. void prompt(void);**

프롬프트 메인

**3-3-2. commands make\_command\_token(char \*command\_line);**

명령어 전체 문장 토큰화, 반환: 토큰으로 분리된 commands 구조체

**3-3-3. int get\_command\_type(char \*command);**

COMMAND 타입 확인 및 반환, 반환: 타입별 정수

**3-3-4. char \*get\_file\_name(char \*path);**

파일명 추출, 반환: 파일명 문자열

**3-3-5. void move\_trash(file\_node \*head, int option\_i);**

파일 휴지통 이동

**3-3-6. void wait\_thread(char \*path, int sec, int option\_r, int option\_i);**

삭제 대기 스레드

**3-3-7. void swap\_handler(int signo);**

시그널로 표준입출력 전환

**3-3-8. struct tm get\_tm(char \*date, char \*time);**

문자열을 시간 구조체로 변환, 반환: 시간 구조체

**3-3-9. void remove\_directory(const char \*path);**

디렉토리 삭제

**3-3-10. int check\_trash\_info(void);**

휴지통 파일 정보 디렉토리 크기 확인, 반환: 초과(1), 여유(0)

**3-3-11. void delete\_trash\_oldest(void);**

휴지통에서 가장 오래 삭제된 파일 제거

**3-3-12. int find\_trash\_overlap(const char \*file\_name);**

휴지통 중복 파일 탐색, 반환: 중복 파일 개수

**3-3-13. void print\_list\_size(file\_node \*head, char \*path, int number, int option\_d, int op\_switch);**

지정 파일 상대 경로 및 크기 출력

**3-3-14. void restore\_file(const char \*file\_name, int option\_l);**

휴지통 파일 복원

**3-3-15. char \*get\_file\_path(char \*path, const char \*file\_name);**

파일 경로 추출, 반환: 파일 경로 문자열

**3-3-16. void refresh\_trash(const char \*file\_name, int idx, int delete\_idx);**

삭제 후 중복 파일 번호 재정렬

**3-3-17. void sort\_info\_oldest(file\_infos \*file\_info, int idx);**

삭제 시간 오름차순 정렬

**3-3-18. void sort\_info\_order(file\_infos \*file\_info, int idx);**

중복 파일 오름차순 정렬

**3-3-19. void print\_list\_tree(file\_node \*head, int level, int level\_check[], int is\_root);**

모니터링 파일 목록 트리 출력

**3-3-20. void print\_tree\_indent(int level, int level\_check[]);**

트리 출력 보조 함수

**3-3-21. char \*rtrim(char \*\_str);**

문자열 오른쪽 공백 제거, 반환: 공백 제거 된 문자열

**3-3-22. char \*ltrim(char \*\_str);**

문자열 왼쪽 공백 제거, 반환: 공백 제거 된 문자열

**3-3-23. void to\_lower\_case(char \*str);**

문자열 소문자 변환

**3-3-24. void free\_command(commands command);**

명령행 구조체 초기화

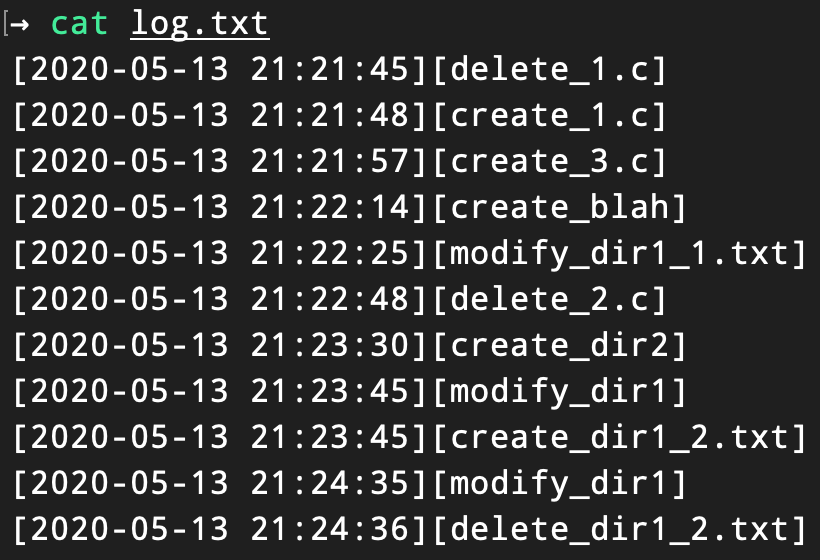
**3-3-25. void print\_usage(void);**

사용법 출력

**4. 테스트 및 결과**

**4-1. 모니터링 프로그램**

**4-1-1. 로그 생성 및 기록**



**4-1-2. 데몬프로세스 생성**

**병, 검은색, 표지판, 빨간색이(가) 표시된 사진

자동 생성된 설명**

**4-2. 프롬프트 프로그램**

**4-2-1. DELETE 명령어**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**4-2-2. DELETE 명령어 휴지통 삭제 파일 정보**

**그리기이(가) 표시된 사진

자동 생성된 설명**

**4-2-3. DELETE 명령어 -r 옵션**

**스크린샷이(가) 표시된 사진

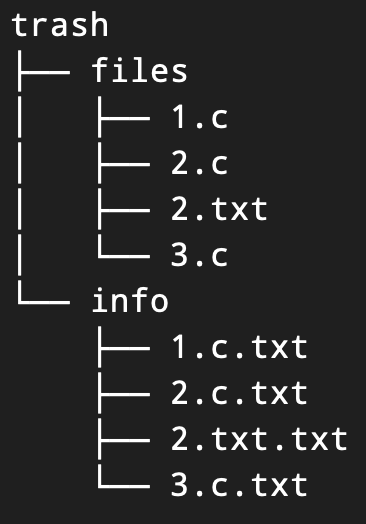
자동 생성된 설명**

**4-2-4. DELETE 명령어 -i 옵션**

**사진, 검은색, 남자, 그리기이(가) 표시된 사진

자동 생성된 설명**

**4-2-5. DELETE 명령어 휴지통 디렉토리 구조**

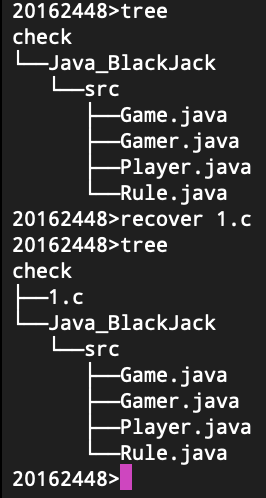
****

**4-2-6. DELETE 명령어 휴지통 정보 디렉토리 크기가 2048KB를 초과한 경우**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**4-2-7. RECOVER 명령어**

****

**4-2-8. RECOVER 명령어 -l 옵션**

**텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

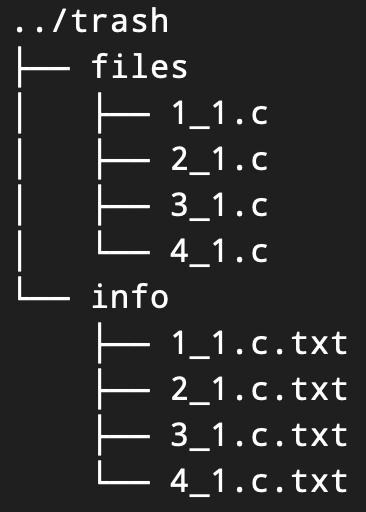
**4-2-9. RECOVER 명령어 휴지통 디렉토리에 중복된 파일이 존재하는 경우**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**4-2-10. RECOVER 명령어 휴지통 디렉토리 중복된 파일 복구 전/후 구조**

**검은색이(가) 표시된 사진

자동 생성된 설명**

**4-2-11. SIZE 명령어**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**4-2-12. TREE 명령어**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**4-2-13. HELP 명령어 / 잘못된 명령어 및 엔터 입력**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**4-2-14. EXIT 명령어**

**그리기이(가) 표시된 사진

자동 생성된 설명**

**5. 소스코드와 주석**

**5-1. Makefile**

ssu\_mntr: support.o monitoring.o prompt.o ssu\_mntr.o common.h monitoring.h prompt.h

gcc -o ssu\_mntr ssu\_mntr.o monitoring.o prompt.o support.o

support.o: support.c common.h

gcc -c support.c

monitoring.o: monitoring.c common.h monitoring.h

gcc -c monitoring.c

prompt.o : prompt.c common.h prompt.h

gcc -c prompt.c

ssu\_mntr.o: ssu\_mntr.c common.h

gcc -c ssu\_mntr.c

clean:

rm \*.o

rm ssu\_mntr

**5-2. common.h**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <ctype.h>

#include <signal.h>

#include <dirent.h>

#include <time.h>

#include <pthread.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/prctl.h>

#include <sys/time.h>

// 불 타입

#ifndef true

#define true 1

#endif

#ifndef false

#define false 0

#endif

// 표준 입출력 타입

#ifndef STDOUT

#define STDOUT 1

#endif

#ifndef STDERR

#define STDERR 2

#endif

// 명령어 타입

#ifndef DELETE

#define DELETE 1

#endif

#ifndef SIZE

#define SIZE 2

#endif

#ifndef RECOVER

#define RECOVER 3

#endif

#ifndef TREE

#define TREE 4

#endif

#ifndef EXIT

#define EXIT 5

#endif

#ifndef HELP

#define HELP 6

#endif

#ifndef UNKNOWN

#define UNKNOWN 7

#endif

// 크기

#define DATE\_SIZE 10

#define TIME\_SIZE 8

#define BUFFER\_SIZE 256

#define MAX\_BUFFER\_SIZE 1024

#define MAX\_INFO\_SIZE 2048

// 디렉토리

#define CHECK "check"

#define TRASH "trash"

#define TRASH\_FILES TRASH "/files"

#define TRASH\_INFO TRASH "/info"

#define LOG "log.txt"

// 모니터링 상태

#define UNCHCK -1

#define CHCKED 0

#define CREATE 2

#define MODIFY 3

// 형식

#define YYMMDD\_HHMMSS "%.4d-%.2d-%.2d %.2d:%.2d:%.2d"

#define YYMMDD "%.4d-%.2d-%.2d"

#define HHSS "%.2d:%.2d"

#define PROMPT "20162448>"

#define SECOND\_TO\_MICRO 1000000

// 권한

#define DIR\_MODE 0755

typedef struct ssu\_fileNode{ // 모니터링 파일 목록 구조체

char name[BUFFER\_SIZE]; // 파일 이름

struct stat attr; // 파일 상태 정보

struct dirent \*\*namelist; // 디렉토리 경우 하위 파일 목록

struct ssu\_fileNode \*next; // 하위 디렉토리 파일 포인터

struct ssu\_fileNode \*child; // 같은 레벨의 다음 파일 포인터

int size; // 파일 크기

int status; // 모니터링 확인 상태

} file\_node;

void prompt(void); // 프롬프트

void monitoring(void); // 모니터링

file\_node\* make\_list(char \*path); // 디렉토리 파일 목록 트리화

file\_node \*make\_node(void); // 노드 생성

int count\_size(file\_node \*head); // 디렉토리 크기 반환

void free\_list(file\_node \*head); // 파일 목록 구조체 메모리 할당 해제

char \*make\_time\_format(struct tm time); // 시간 형식 문자열 생성

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

**5-3. ssu\_mntr.c**

#include "common.h"

int main(void) // 프로세스 운영 메인 함수

{

pid\_t daemon\_pid;

struct timeval begin\_t, end\_t;

gettimeofday(&begin\_t, NULL); // 측정 시작

// 프롬프트, 모니터링 프로세스 시작

if((daemon\_pid = fork()) < 0) {

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

exit(1);

}

// 자식 프로세스(데몬): 모니터링

if(daemon\_pid == 0)

monitoring();

// 부모 프로세스: 프롬프트

else

prompt();

// 모니터링 프로세스 종료 (삭제됨)

//kill(daemon\_pid + 1, SIGKILL); // SIGKILL(LINUX), KILL(MAC) == 9

gettimeofday(&end\_t, NULL); // 측정 종료

ssu\_runtime(&begin\_t, &end\_t); // 런타임 출력

exit(0);

}

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

{

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);

}

**5-4. support.c**

#include "common.h"

file\_node \*make\_node(void) // 노드 생성

{

file\_node \*tmp = calloc(true, sizeof(file\_node));

memset(tmp->name, 0, BUFFER\_SIZE);

tmp->next = NULL;

tmp->child = NULL;

tmp->namelist = NULL;

tmp->size = 0;

tmp->status = UNCHCK;

return tmp;

}

file\_node \*make\_list(char \*path) // 디렉토리 파일 목록 트리화

{

file\_node \*head, \*now;

// 파일 : 노드 생성(절대경로/이름, 상태정보)

// 디렉토리: 트리 생성

int file\_count;

int is\_dirattr = true;

int i;

// 부모: 현재 경로, 디렉토리 상태정보, 파일 목록 정보

// 자식: 절대경로 파일이름, 파일 상태정보, 다음 파일 포인터 및 디렉토리 하위 부모노드 포인터

head = make\_node();

now = head;

strcpy(head->name, path); // 현재 경로 저장

stat(head->name, &(head->attr)); // 상태 정보 저장

file\_count = scandir(head->name, &(head->namelist), NULL, alphasort); // 현재 경로의 모든 파일 탐색 및 개수 저장

for(i = 0; i < file\_count; i++) {

if(!strcmp(head->namelist[i]->d\_name, ".") || !strcmp(head->namelist[i]->d\_name, "..")) // 현재, 상위 디렉토리 접근 지정자 생략

continue;

file\_node \*new = make\_node(); // 새로운 노드 생성

sprintf(new->name, "%s/%s", path, head->namelist[i]->d\_name); // 파일 이름 저장

stat(new->name, &(new->attr));

if(S\_ISDIR(new->attr.st\_mode)) // 현재 경로의 파일 목록 중 탐색한 파일이 디렉토리일 경우

new = make\_list(new->name); // 해당 디렉토리 파일 목록 트리화

else

new->size = new->attr.st\_size;

if(is\_dirattr) { // 현재 노드가 현재 경로의 부모노드일 경우

now->child = new;

now = now->child;

is\_dirattr = false;

} else { // 아닐 경우 형제로 연결

now->next = new;

now = now->next;

}

}

head->size = count\_size(head);

return head;

}

int count\_size(file\_node \*head) // 디렉토리 크기 반환

{

file\_node \*now;

int size;

size = false;

if(S\_ISDIR(head->attr.st\_mode))

now = head->child;

else

return head->attr.st\_size;

while(now != NULL) {

size += now->size;

now = now->next;

}

return size;

}

void free\_list(file\_node \*head) // 모니터링 파일 목록 메모리 할당 해제

{

// 모든 노드들을 찾아서 메모리 할당을 해제한다.

if(head->child != NULL) // 자식 탐색

free\_list(head->child);

if(head->next != NULL) // 형제 탐색

free\_list(head->next);

free(head->namelist);

free(head); // 메모리 엑세스 허용

}

char \*make\_time\_format(struct tm time) // 시간 형식 문자열 생성

{

static char time\_format[BUFFER\_SIZE];

sprintf(time\_format, YYMMDD\_HHMMSS,

time.tm\_year + 1900,

time.tm\_mon + 1,

time.tm\_mday,

time.tm\_hour,

time.tm\_min,

time.tm\_sec);

return (char\*)time\_format;

}

**5-5. monitoring.h**

#include "common.h"

typedef struct ssu\_changeItem { // 변경사항 구조체

time\_t time; // 변경 시간

char name[BUFFER\_SIZE]; // 파일 이름

int status; // 변경 상태

} change\_file;

void monitoring(void); // 모니터링

void init\_list\_status(file\_node \*head, int status); // 모니터링 파일 상태 초기화

void compare\_list(file\_node \*new\_list, file\_node \*old\_list); // 파일 목록 트리 비교

int compare\_file(file\_node \*new\_file, file\_node \*old\_file); // 파일 정보 비교

int write\_change\_list(file\_node \*head, int idx, int status); // 변경 사항 목록 작성

void sort\_change\_list(int idx); // 변경사항 목록 시간순 정렬

void write\_change\_log(int idx); // 변경사항 파일 기록

void set\_daemon\_process(void); // 데몬프로세스 설정

**5-6. monitoring.c**

#include "monitoring.h"

char pwd[BUFFER\_SIZE];

char check\_path[MAX\_BUFFER\_SIZE]; // $(PWD)/check 절대경로

char log\_path[MAX\_BUFFER\_SIZE];

change\_file change\_list[MAX\_BUFFER\_SIZE]; // 변경 목록

void monitoring(void) // 모니터링

{

FILE \*fp; // log.txt 파일 구조체

int old\_list\_cnt, new\_list\_cnt; // 모니터링 디렉토리 파일 개수(기존, 신규)

file\_node \*old\_list, \*new\_list; // 모니터링 디렉토리 트리(기존, 신규)

int change\_list\_cnt;

getcwd(pwd, BUFFER\_SIZE);

sprintf(check\_path, "%s/%s", pwd, CHECK);

sprintf(log\_path, "%s/%s", pwd, LOG);

if(access(check\_path, F\_OK) < 0) // 모니터링 디렉토리 확인

mkdir(check\_path, DIR\_MODE); // 존재하지 않을 경우 생성

if((fp = fopen(log\_path, "w+")) == NULL) { // 로그 파일 열기 및 생성, 기존 로그 덮어쓰기

fprintf(stderr, "fopen error for log.txt\n");

exit(1);

}

fclose(fp);

set\_daemon\_process(); // 데몬프로세스화

// 프로세스 이름 변경

prctl(PR\_SET\_NAME, "ssu\_mntr-daemon\0", NULL, NULL, NULL);

old\_list = make\_list(check\_path);

while(true) {

change\_list\_cnt = 0;

new\_list = make\_list(check\_path); // 현재 파일 목록 및 상태 저장

compare\_list(new\_list->child, old\_list->child); // 파일 목록 트리 비교

change\_list\_cnt = write\_change\_list(new\_list->child, change\_list\_cnt, CREATE); // 생성된 파일 확인

change\_list\_cnt = write\_change\_list(old\_list->child, change\_list\_cnt, DELETE); // 삭제된 파일 확인

sort\_change\_list(change\_list\_cnt);

write\_change\_log(change\_list\_cnt);

free\_list(old\_list);

old\_list = new\_list;

init\_list\_status(old\_list->child, UNCHCK);

sleep(1);

}

}

void init\_list\_status(file\_node \*head, int status) // 모니터링 파일 상태 초기화

{

file\_node \*now;

now = head;

while(now != NULL) {

now->status = status;

if(now->child != NULL)

init\_list\_status(now->child, status); // 디렉토리 하위 파일 모니터링 상태 초기화

now = now->next;

}

}

void compare\_list(file\_node \*new\_list, file\_node \*old\_list) // 파일 목록 트리 비교

{

file\_node \*now;

if(new\_list == NULL || old\_list == NULL) // 둘중 하나라도 비교 대상이 존재하지 않을 경우

return;

now = old\_list;

while(now != NULL) {

compare\_file(new\_list, now);

if(now->child != NULL)

compare\_list(new\_list, now->child);

now = now->next;

}

}

int compare\_file(file\_node \*new\_file, file\_node \*old\_file) // 파일 정보 비교

{

file\_node \*now;

now = new\_file;

while(now != NULL) {

if(!strcmp(now->name, old\_file->name)) { // 해당 이름을 가진 파일이 기존에 이미 존재할 경우

now->status = CHCKED;

if(now->attr.st\_mtime != old\_file->attr.st\_mtime) // 해당 파일이 수정되었을 경우

now->status = MODIFY;

old\_file->status = CHCKED;

return true;

}

if(now->child != NULL) // 디렉토리 안에 파일이 존재할 경우

if(compare\_file(now->child, old\_file))

break;

now = now->next;

}

return false;

}

int write\_change\_list(file\_node \*head, int idx, int status) // 변경사항 목록 작성

{

file\_node \*now;

now = head;

while(now != NULL) {

switch(now->status) {

case UNCHCK:

change\_list[idx].time = time(NULL);

strcpy(change\_list[idx].name, now->name);

change\_list[idx++].status = status;

break;

case MODIFY:

change\_list[idx].time = now->attr.st\_mtime;

strcpy(change\_list[idx].name, now->name);

change\_list[idx++].status = MODIFY;

break;

}

if(now->child != NULL)

idx = write\_change\_list(now->child, idx, status);

now = now->next;

}

return idx;

}

void sort\_change\_list(int idx) // 변경사항 목록 시간순 정렬

{

int i, j;

change\_file tmp;

for(i = 0; i < idx; i++)

for(j = i+1; j < idx; j++)

if(change\_list[i].time > change\_list[j].time) {

tmp = change\_list[i];

change\_list[i] = change\_list[j];

change\_list[j] = tmp;

}

}

void write\_change\_log(int idx) // 변경사항 파일 기록

{

char file\_name[BUFFER\_SIZE];

char \*time\_format;

struct tm time;

FILE \*fp;

int i;

char \*tmp;

if((fp = fopen(log\_path, "r+")) < 0) {

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

exit(1);

}

fseek(fp, 0, SEEK\_END); // 오프셋을 로그파일의 끝으로 지정

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

tmp = strstr(change\_list[i].name, CHECK);

tmp += strlen(CHECK) + 1; // strlen("check") + 1(/)

strcpy(file\_name, tmp);

while((tmp = strchr(file\_name, '/')) != NULL) // file\_name = "파일명\_"

\*tmp = '\_';

time = \*localtime(&change\_list[i].time);

time\_format = make\_time\_format(time);

switch(change\_list[i].status) {

case CREATE:

fprintf(fp, "[%s][%s\_%s]\n", time\_format, "create", file\_name);

break;

case DELETE:

fprintf(fp, "[%s][%s\_%s]\n", time\_format, "delete", file\_name);

break;

case MODIFY:

fprintf(fp, "[%s][%s\_%s]\n", time\_format, "modify", file\_name);

break;

}

}

fclose(fp);

}

void set\_daemon\_process(void) // 데몬 프로세스 설정

{

pid\_t pid;

int fd, maxfd;

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

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

exit(1);

} else if(pid != 0) // #1 백그라운드 수행

exit(0);

// #2 새로운 프로세스 그룹 생성

setsid();

// #3 터미널 입출력 시그널 무시

signal(SIGTTIN, SIG\_IGN); // STDIN 무시

signal(SIGTTOU, SIG\_IGN); // STDOUT 무시

signal(SIGTSTP, SIG\_IGN); // STDERR 무시

// #4 파일 모드 생성 마스크 해제

umask(false);

// #5 루트 디렉토리 이동

chdir("/");

// #6 모든 파일 디스크럽터 연결 종료

maxfd = getdtablesize(); // 모든 파일 디스크럽터 개수 획득

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

close(fd);

// #7 표준 입출력 및 에러 재지정

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

dup(0);

dup(0);

// 라이브러리 루틴 무효화

}

**5-7. prompt.h**

#include "common.h"

typedef struct ssu\_commandToken { // 프롬프트 명령어 구조체

char \*\*argv; // 명령행 토큰

int argc; // 명령행 인자 개수

} commands;

typedef struct ssu\_fileinfo {

int num;

char path[BUFFER\_SIZE];

struct tm d\_tm; // 삭제 시간

struct tm m\_tm; // 수정 시간

} file\_infos;

commands make\_command\_token(char \*command\_line); // 명령어 전체 문장 토큰화

int get\_command\_type(char \*command); // COMMAND 타입 확인 및 반환

char \*get\_file\_name(char \*path); // 파일명 추출

void move\_trash(file\_node \*head, int option\_i); // 파일 휴지통 이동

void wait\_thread(char \*path, int sec, int option\_r, int option\_i); // 삭제 대기 스레드

void swap\_handler(int signo); // 시그널로 표준입출력 전환

struct tm get\_tm(char \*date, char \*time); // 시간 구조체 획득

void remove\_directory(const char \*path); // 디렉토리 삭제

int check\_trash\_info(void); // 휴지통 파일 정보 디렉토리 크기 확인

void delete\_trash\_oldest(void); // 휴지통에서 가장 오래 삭제된 파일 제거

int find\_trash\_overlap(const char \*file\_name); // 휴지통 중복 파일 탐색

void print\_list\_size(file\_node \*head, char \*path, int number, int option\_d, int op\_switch); // 지정 파일 상대 경로 및 크기 출력

void restore\_file(const char \*file\_name, int option\_l); // 휴지통 파일 복원

char \*get\_file\_path(char \*path, const char \*file\_name); // 파일 경로 추출

void refresh\_trash(const char \*file\_name, int idx, int delete\_idx); // 삭제 후 중복 파일 번호 재정렬

void sort\_info\_oldest(file\_infos \*file\_info, int idx); // 삭제 시간 오름차순 정렬

void sort\_info\_order(file\_infos \*file\_info, int idx); // 중복 파일 오름차순 정렬

void print\_list\_tree(file\_node \*head, int level, int level\_check[], int is\_root); // 모니터링 파일 목록 트리 출력

void print\_tree\_indent(int level, int level\_check[]); // 트리 출력 보조 함수

char \*rtrim(char \*\_str); // 문자열 오른쪽 공백 제거

char \*ltrim(char \*\_str); // 문자열 왼쪽 공백 제거

void to\_lower\_case(char \*str); // 문자열 소문자 변환

void free\_command(commands command); // 명령행 구조체 초기화

void print\_usage(void); // 사용법 출력

**5-8. prompt.c**

#include "prompt.h"

char pwd[BUFFER\_SIZE];

int in\_fd;

int out\_fd;

int err\_fd;

void prompt(void) // 프롬프트 메인 함수

{

// 프롬프트

char check\_path[MAX\_BUFFER\_SIZE]; // $(PWD)/check 절대경로

char command\_line[MAX\_BUFFER\_SIZE]; // 입력받은 실행 명령 버퍼

// 명령어

commands command;

int command\_type = false; // 실행 명령 타입

char target\_path[BUFFER\_SIZE]; // FINE\_NAME 경로

// 공통

file\_node \*head;

int is\_invalid;

int is\_filename;

int idx;

// DELETE

struct tm reserv\_tm;

time\_t current\_t;

time\_t reserv\_t;

int is\_endtime;

int option\_i;

int option\_r;

int sec;

// SIZE

int number;

int option\_d;

// RECOVER

int option\_l;

char trash\_files\_path[MAX\_BUFFER\_SIZE];

char file\_name[BUFFER\_SIZE];

// TREE

int level\_check[BUFFER\_SIZE];

// 사용자 정의 시그널

// SIGUSR1: 표준입출력 닫기

// SIGUSR2: 표준입출력 열기

signal(SIGUSR1, swap\_handler);

signal(SIGUSR2, swap\_handler);

getcwd(pwd, BUFFER\_SIZE);

sprintf(check\_path, "%s/%s", pwd, CHECK); // 모니터링 디렉토리 경로

while (command\_type != EXIT) {

fputs(PROMPT, stdout);

fgets(command\_line, MAX\_BUFFER\_SIZE, stdin); // 실행 명령 입력

strcpy(command\_line, ltrim(rtrim(command\_line))); // 실행 명령 좌우 공백 지우기

command = make\_command\_token(command\_line);

command\_type = get\_command\_type(command.argv[0]); // 명령 타입 구분

// COMMANDS

// DELETE(1) : DELETE <FILENAME> <END\_TIME> [OPTION]

// SIZE(2) : SIZE <FILENAME> [OPTION]

// RECOVER(3) : RECOVER <FILENAME> [OPTION]

// TREE(4) : TREE

// EXIT(5) : EXKT

// HELP(6) : HELP

switch(command\_type) {

case DELETE:

is\_invalid = false;

is\_endtime = false;

is\_filename = false;

option\_i = false;

option\_r = false;

chdir(CHECK);

if(command.argc < 2) { // 인자 개수가 부족할 경우

fprintf(stderr, "%s1: FILE\_NAME doesn't exist\n", command.argv[0]);

continue;

}

// 명령행 인자 파싱 및 에러 검출 순위

// 1. 옵션

// 2. 파일명

// 3. 시간

for(idx = 1; idx < command.argc; idx++) {

// 옵션 파싱

if(command.argv[idx][0] == '-') {

if(command.argv[idx][1] == 'r') // -r 옵션

option\_r = true;

else if(command.argv[idx][1] == 'i') // -i 옵션

option\_i = true;

else {

fprintf(stderr, "%s: invalid option %c\n", command.argv[0], command.argv[idx][1]);

is\_invalid = true;

break;

}

continue;

}

// FILE\_NAME 파싱

if(!is\_filename) {

if(access(command.argv[idx], F\_OK) < 0) { // 파일이 존재하지 않을 경우

fprintf(stderr, "%s: access error for %s\n", command.argv[0], command.argv[idx]);

is\_invalid = true;

break;

} else { // 파일이 존재하는 경우

realpath(command.argv[idx], target\_path); // 절대 경로로 변경

if(strstr(target\_path, check\_path) == NULL) { // FILE\_NAME이 모니터링 디렉토리에 존재하지 않을 경우

fprintf(stderr, "%s: %s doesn't exist in %s\n", command.argv[0], command.argv[idx], check\_path);

is\_invalid = true;

break;

}

is\_filename = true;

}

continue;

}

// END\_TIME 파싱

if(!is\_endtime && (idx + 1 < command.argc)) {

current\_t = time(NULL);

reserv\_tm = get\_tm(command.argv[idx], command.argv[idx + 1]);

reserv\_t = mktime(&reserv\_tm);

if((sec = (reserv\_t - current\_t)) < 0) {

fprintf(stderr, "%s: invalid END\_TIME\n", command.argv[0]);

is\_invalid = true;

break;

}

is\_endtime = true;

}

}

chdir(pwd);

if(is\_invalid) // 파싱 중 에러 발견 시

break;

else if(!is\_filename) { // FILE\_NAME이 존재하지 않을 경우

fprintf(stderr, "%s: invalid input\n", command.argv[0]);

break;

} else if(option\_r && !is\_endtime) { // -r 옵션이 주어지고, END\_TIME이 존재하지 않는 경우

fprintf(stderr, "%s -r: END\_TIME doesn't exist\n", command.argv[0]);

break;

}

if(is\_endtime) // END\_TIME이 존재할 경우

wait\_thread(target\_path, sec, option\_r, option\_i);

else { // END\_TIME이 존재하지 않을 경우

head = make\_list(target\_path); // 파일 목록 구조체 생성

move\_trash(head, option\_i);

free\_list(head);

}

while(check\_trash\_info())

delete\_trash\_oldest();

break;

case SIZE:

number = true;

is\_invalid = false;

is\_filename = false;

option\_d = false;

if(command.argc < 2) { // 인자 개수가 부족할 경우

fprintf(stderr, "%s2: FILE\_NAME doesn't exist\n", command.argv[0]);

continue;

}

// 명령행 인자 파싱 및 에러 검출 순위

// 1. 옵션

// 2. 파일명

for(idx = 1; idx < command.argc; idx++) {

//옵션 파싱

if(command.argv[idx][0] == '-') {

if(command.argv[idx][1] == 'd') { // -d 옵션

option\_d = true;

if(idx + 1 == command.argc) {

fprintf(stderr, "%s: NUMBER doesn't exist\n", command.argv[0]);

is\_invalid = true;

break;

} else if((number = atoi(command.argv[++idx])) <= 0) { // NUMBER

fprintf(stderr, "%s: invalid input NUMBER\n", command.argv[0]);

is\_invalid = true;

break;

}

} else {

fprintf(stderr, "%s: invalid option %c\n", command.argv[0], command.argv[idx][1]);

is\_invalid = true;

break;

}

continue;

}

// FILE\_NAME 파싱

if(!is\_filename) {

if(access(command.argv[idx], F\_OK) < 0) { // 파일이 존재하지 않을 경우

fprintf(stderr, "%s: access error for %s\n", command.argv[0], command.argv[idx]);

is\_invalid = true;

break;

} else { // 파일이 존재하는 경우

realpath(command.argv[idx], target\_path); // 절대 경로로 변경

if(strstr(target\_path, check\_path) == NULL) { // FILE\_NAME이 모니터링 디렉토리에 존재하지 않을 경우

fprintf(stderr, "%s: %s doesn't exist in %s\n", command.argv[0], command.argv[idx], check\_path);

is\_invalid = true;

break;

}

is\_filename = true;

}

continue;

}

}

if(is\_invalid) // 파싱 중 에러 발견 시

break;

else if(!is\_filename) {

fprintf(stderr, "%s: invalid input\n", command.argv[0]);

break;

}

head = make\_list(target\_path); // 해당 경로의 파일 목록 구조체 생성

print\_list\_size(head, target\_path, number, option\_d, true); // 출력

free\_list(head);

break;

case RECOVER:

is\_filename = false;

option\_l = false;

is\_invalid = false;

if(command.argc < 2) { // 인자 개수가 부족할 경우

fprintf(stderr, "%s3: FILE\_NAME doesn't exist\n", command.argv[0]);

continue;

}

// 명령행 인자 파싱 및 에러 검출 순위

// 1. 옵션

// 2. 파일명

for(idx = 1; idx < command.argc; idx++) {

//옵션 파싱

if(command.argv[idx][0] == '-') {

if(command.argv[idx][1] == 'l') // -l 옵션

option\_l = true;

else {

fprintf(stderr, "%s: invalid option %c\n", command.argv[0], command.argv[idx][1]);

is\_invalid = true;

break;

}

continue;

}

// FILE\_NAME 파싱

if(!is\_filename) {

sprintf(trash\_files\_path, "%s/%s/%s", pwd, TRASH\_FILES, command.argv[idx]);

if(access(trash\_files\_path, F\_OK) < 0) { // 파일이 존재하지 않을 경우

sprintf(trash\_files\_path, "%s/%s/1\_%s", pwd, TRASH\_FILES, command.argv[idx]);

if(access(trash\_files\_path, F\_OK) < 0) { // 중복 파일 존재 유무 확인

fprintf(stderr, "%s: access error for %s\n", command.argv[0], command.argv[idx]);

is\_invalid = true;

break;

}

}

strcpy(file\_name, command.argv[idx]);

is\_filename = true;

continue;

}

}

if(is\_invalid) // 파싱 중 에러 발견 시

break;

else if(!is\_filename) {

fprintf(stderr, "%s: invalid input\n", command.argv[0]);

break;

}

restore\_file(file\_name, option\_l);

break;

case TREE:

memset(level\_check, 0, BUFFER\_SIZE);

head = make\_list(check\_path); // 모니터링 디렉토리 파일 목록 구조체 생성

print\_list\_tree(head, 0, level\_check, true); // 출력

free\_list(head); // 메모리 할당 해제

break;

case HELP:

case UNKNOWN:

print\_usage();

break;

case EXIT:

default:

break;

}

free\_command(command);

fflush(stdin);

fflush(stdout);

}

printf("Successful prompt termination.\n");

}

commands make\_command\_token(char \*command\_line) // 명령어 전체 문장 토큰화

{

commands result;

char \*tmp;

char \*command;

result.argv = (char \*\*)calloc(BUFFER\_SIZE, sizeof(char\*));

result.argc = 0;

if((command = strtok(command\_line, " ")) == NULL) { // 엔터만 쳤을 경우

result.argv[result.argc] = (char \*)calloc(true, sizeof(char));

strcpy(result.argv[result.argc], "");

return result;

}

to\_lower\_case(command); // 명령어 소문자화

result.argv[result.argc] = (char \*)calloc(BUFFER\_SIZE, sizeof(char)); // 메모리 공간 할당

strcpy(result.argv[result.argc++], command); // 토큰 배열에 복사

while((tmp = strtok(NULL, " ")) != NULL) { // 나머지 인자 복사

result.argv[result.argc] = (char \*)calloc(BUFFER\_SIZE, sizeof(char)); // 메모리 공간 할당

strcpy(result.argv[result.argc++], tmp); // 토큰 배열에 복사

}

return result;

}

int get\_command\_type(char \*command) // COMMAND 타입 확인 및 반환

{

// 명령어 타입 확인

if(!strcmp(command,""))

return false;

else if(!strcmp(command, "delete"))

return DELETE;

else if(!strcmp(command, "size"))

return SIZE;

else if(!strcmp(command, "recover"))

return RECOVER;

else if(!strcmp(command, "tree"))

return TREE;

else if(!strcmp(command, "exit"))

return EXIT;

else if(!strcmp(command, "help"))

return HELP;

else

return UNKNOWN;

}

char \*get\_file\_name(char \*path) // 파일명 추출

{

static char file\_name[BUFFER\_SIZE];

char \*tmp;

strcpy(file\_name, strtok(path, "/"));

while((tmp = strtok(NULL, "/")) != NULL) {

strcpy(file\_name, tmp);

}

return (char\*)file\_name;

}

void move\_trash(file\_node \*head, int option\_i) // 파일을 휴지통 이동

{

FILE \*fp;

char \*file\_name;

char target\_path[MAX\_BUFFER\_SIZE];

char trash\_files\_path[MAX\_BUFFER\_SIZE];

char trash\_info\_path[MAX\_BUFFER\_SIZE];

char \*time\_format;

struct tm time\_info;

time\_t cur\_time;

int overlap;

sprintf(trash\_files\_path, "%s/%s", pwd, TRASH\_FILES);

sprintf(trash\_info\_path, "%s/%s", pwd, TRASH\_INFO);

strcpy(target\_path, head->name);

if(access(TRASH, F\_OK) < 0) // 휴지통 디렉토리가 존재하지 않을 경우

mkdir(TRASH, DIR\_MODE);

if(access(TRASH\_FILES, F\_OK) < 0) // 삭제 파일 저장 디렉토리가 존재하지 않을 경우

mkdir(TRASH\_FILES, DIR\_MODE);

if(access(TRASH\_INFO, F\_OK) < 0) // 삭제 파일 정보 디렉토리가 존재하지 않을 경우

mkdir(TRASH\_INFO, DIR\_MODE);

file\_name = get\_file\_name(target\_path); // 파일 이름 추출

if(option\_i) {

if(S\_ISDIR(head->attr.st\_mode))

remove\_directory(head->name);

else

unlink(head->name);

} else {

// 파일 정보 생성

if((overlap = find\_trash\_overlap(file\_name)) > 0)

sprintf(target\_path, "%s/%s/%d\_%s.txt", pwd, TRASH\_INFO, overlap, file\_name);

else

sprintf(target\_path, "%s/%s/%s.txt", pwd, TRASH\_INFO, file\_name);

if((fp = fopen(target\_path, "w+")) < 0) {

fprintf(stderr, "fopen error for %s\n", target\_path);

return;

}

time(&cur\_time);

time\_info = \*localtime(&cur\_time);

time\_format = make\_time\_format(time\_info);

fprintf(fp, "[Trash info]\n%s\nD : %s\n", head->name, time\_format);

time\_info = \*localtime(&(head->attr.st\_mtime));

time\_format = make\_time\_format(time\_info);

fprintf(fp, "M : %s\n", time\_format);

fclose(fp);

// 파일 원본 이동

if(overlap > 0)

sprintf(target\_path, "%s/%s/%d\_%s", pwd, TRASH\_FILES, overlap, file\_name);

else

sprintf(target\_path, "%s/%s/%s", pwd, TRASH\_FILES, file\_name); // 이동할 경로 생성

rename(head->name, target\_path); // 이동

}

}

void wait\_thread(char \*path, int sec, int option\_r, int option\_i) // 삭제 대기 스레드

{

file\_node\* head;

pid\_t pid, ppid;

char input;

ppid = getpid();

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

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

else if(pid == 0) {

sleep(sec); // 시간 대기

if(access(path, F\_OK) < 0) { // 파일이 존재하지 않을 경우

fprintf(stderr, "delete: access error for %s\n", path);

return;

}

head = make\_list(path); // 파일 목록 구조체 생성

if(option\_r) { // -r옵션이 존재할 경우

kill(ppid, SIGUSR1); // 부모 프로세스의 표준 입출력 닫기

printf("\nDelete [y/n]? ");

input = fgetc(stdin);

switch(input) {

case 'y':

move\_trash(head, option\_i);

printf("please input ENTER to continue...");

break;

case 'n':

break;

default:

fputs("delete: invalid input, thread was being aborted\n", stdout);

break;

}

} else

move\_trash(head, option\_i);

free\_list(head);

kill(ppid, SIGUSR2); // 부모 프로세스의 표준 입출력 열기

exit(0); // 자식 프로세스 종료

}

return; // 부모 프로세스 반환

}

void swap\_handler(int signo){ // 시그널로 표준입출력 전환

switch(signo) {

case SIGUSR1:

in\_fd = dup(0);

out\_fd = dup(1);

err\_fd = dup(2);

close(0);

close(1);

close(2);

pause();

break;

case SIGUSR2:

dup2(in\_fd, 0);

dup2(out\_fd, 1);

dup2(err\_fd, 2);

break;

}

}

struct tm get\_tm(char \*date, char \*time) // 시간 구조체 획득

{

static struct tm tmp;

int year, month, day;

int hour, min, sec;

year = month = day = hour = min = sec = 0;

sscanf(date, "%d-%d-%d", &year, &month, &day);

sscanf(time, "%d:%d:%d", &hour, &min, &sec);

// 시간 구조체 파싱 예외

if(year < 0 || month < 0 || month > 12 || day < 0 || day > 31 || hour < 0 || hour > 24 || min < 0 || min > 60 || sec < 0 || sec > 60)

return tmp;

else {

tmp.tm\_year = year - 1900;

tmp.tm\_mon = month - 1;

tmp.tm\_mday = day;

tmp.tm\_hour = hour;

tmp.tm\_min = min;

tmp.tm\_sec = sec;

}

return tmp;

}

void remove\_directory(const char \*path) // 디렉토리 삭제

{

// trash와 info에 저장하는 코드 작성 필요!

file\_node \*now;

struct dirent \*dirp;

struct stat statbuf;

DIR \*dp;

char tmp[MAX\_BUFFER\_SIZE];

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

return;

while((dirp = readdir(dp)) != NULL) { // path에 존재하는 디렉토리 안에 파일들 전부 삭제

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

continue;

sprintf(tmp, "%s/%s", path, dirp->d\_name); // tmp = 디렉토리 내부 파일

if(lstat(tmp, &statbuf) == -1) // 파일 상태 정보 추출

continue;

if(S\_ISDIR(statbuf.st\_mode)) // 디렉토리일 경우 재귀적으로 제거

remove\_directory(tmp);

else

unlink(tmp);

}

closedir(dp);

rmdir(path);

}

int check\_trash\_info(void) // 휴지통 파일 정보 디렉토리 크기 확인

{

char trash\_info\_path[MAX\_BUFFER\_SIZE];

file\_node \*head;

int size;

sprintf(trash\_info\_path, "%s/%s", pwd, TRASH\_INFO);

head = make\_list(trash\_info\_path);

size = head->size;

free\_list(head);

if(size > MAX\_INFO\_SIZE)

return true;

return false;

}

void delete\_trash\_oldest(void) // 휴지통에서 가장 오래 삭제된 파일 제거

{

char trash\_info\_path[MAX\_BUFFER\_SIZE];

char old\_info[BUFFER\_SIZE];

char old\_path[MAX\_BUFFER\_SIZE]; // 가장 오래된 파일 경로

char old\_name[BUFFER\_SIZE];

char date[BUFFER\_SIZE];

char time[BUFFER\_SIZE];

char path[MAX\_BUFFER\_SIZE];

char file\_name[BUFFER\_SIZE];

char tmp[MAX\_BUFFER\_SIZE];

struct tm tmp\_tm;

FILE \*fp;

time\_t old\_sec; // 가장 오래된 파일 삭제 시간

time\_t now\_sec;

int file\_count;

struct stat statbuf;

struct dirent \*\*namelist;

int is\_first;

int i;

is\_first = true;

sprintf(trash\_info\_path, "%s/%s", pwd, TRASH\_INFO);

chdir(trash\_info\_path);

file\_count = scandir(trash\_info\_path, &namelist, NULL, alphasort);

for(i = 0; i < file\_count; i++) { // 정보 파일이 저장된 디렉토리 탐색

if(!strcmp(namelist[i]->d\_name, ".") || !strcmp(namelist[i]->d\_name, ".."))

continue;

sprintf(tmp, "%s/%s/%s", pwd, TRASH\_INFO, namelist[i]->d\_name); // 정보 파일 경로

// 정보 파일에서 데이터 추출

if((fp = fopen(tmp, "r+")) < 0) { // 파일 읽기 모드로 열기

fprintf(stderr, "fopen error for %s\n", namelist[i]->d\_name);

continue;

}

fseek(fp, 13, SEEK\_SET); // 헤더 생략

fscanf(fp, "%s\n", path); // 파일 경로

fscanf(fp, "D : %s %s\n", date, time); // 삭제 시간

fclose(fp); // 파일 닫기

// 시간 변환

tmp\_tm = get\_tm(date, time);

now\_sec = mktime(&tmp\_tm);

// 경로 변환

sprintf(path, "%s/%s/%s", pwd, TRASH\_FILES, get\_file\_name(path));

if(is\_first) { // 최초 탐색일 경우

old\_sec = now\_sec;

strcpy(old\_info, tmp);

strcpy(old\_name, namelist[i]->d\_name);

is\_first = false;

} else {

if(now\_sec < old\_sec) {

old\_sec = now\_sec;

strcpy(old\_info, tmp);

strcpy(old\_name, namelist[i]->d\_name);

}

}

}

sprintf(old\_path, "%s/%s/", pwd, TRASH\_FILES);

strncat(old\_path, old\_name, strlen(old\_name) - 4);

stat(old\_path, &statbuf); // 가장 오래된 파일 정보 획득

if(S\_ISDIR(statbuf.st\_mode)) // 디렉토리일 경우

remove\_directory(old\_path);

else // 파일일 경우

unlink(old\_path);

unlink(old\_info);

free(namelist);

// 중복 파일 정리

is\_first = false;

sscanf(old\_name, "%d\_%s", &i, old\_name); // 중복 번호, 파일 이름 추출

strncpy(file\_name, old\_name, strlen(old\_name) - 4);

// 중복 파일이 두개만 존재했었을 경우

if(i == 1) {

sprintf(tmp, "%d\_%s", i + 2, old\_name);

if(access(tmp, F\_OK) < 0) { // 존재하지 않을 경우

// 정보 파일 정리

sprintf(tmp, "%d\_%s", i + 1, old\_name);

sprintf(path, "%s", old\_name);

rename(tmp, path);

// 원본 파일 정리

sprintf(tmp, "%s/%s/%d\_%s", pwd, TRASH\_FILES, i + 1, file\_name);

sprintf(path, "%s/%s/%s", pwd, TRASH\_FILES, file\_name);

rename(tmp, path);

is\_first = true;

}

} else if(i == 2) { // 중복 파일이 두개만 존재했었을 경우

sprintf(tmp, "%d\_%s", i + 1, old\_name);

if(access(tmp, F\_OK) < 0) { // 존재하지 않을 경우

// 정보 파일 정리

sprintf(tmp, "%d\_%s", i - 1, old\_name);

sprintf(path, "%s", old\_name);

rename(tmp, path);

// 원본 파일 정리

sprintf(tmp, "%s/%s/%d\_%s", pwd, TRASH\_FILES, i - 1, file\_name);

sprintf(path, "%s/%s/%s", pwd, TRASH\_FILES, file\_name);

rename(tmp, path);

is\_first = true;

}

}

while(!is\_first) {

sprintf(tmp, "%d\_%s", i + 1, old\_name);

if(access(tmp, F\_OK) < 0) // 중복 파일 일 때 이후 번호의 파일이 존재할 경우

break;

// 정보 파일 정리

sprintf(tmp, "%d\_%s", i + 1, old\_name);

sprintf(path, "%d\_%s", i, old\_name);

rename(tmp, path);

// 원본 파일 정리

sprintf(tmp, "%s/%s/%d\_%s", pwd, TRASH\_FILES, i + 1, file\_name);

sprintf(path, "%s/%s/%d\_%s", pwd, TRASH\_FILES, i, file\_name);

rename(tmp, path);

i++;

}

chdir(pwd);

}

int find\_trash\_overlap(const char \*file\_name) // 휴지통 중복 파일 탐색

{

char trash\_info\_path[MAX\_BUFFER\_SIZE];

char trash\_files\_path[MAX\_BUFFER\_SIZE];

char target\_path[MAX\_BUFFER\_SIZE];

struct dirent \*\*namelist;

int overlap\_count;

int file\_count;

int tmp;

int i;

overlap\_count = 0;

sprintf(trash\_files\_path, "%s/%s", pwd, TRASH\_FILES);

sprintf(trash\_info\_path, "%s/%s", pwd, TRASH\_INFO);

chdir(trash\_files\_path);

file\_count = scandir(trash\_files\_path, &namelist, NULL, alphasort);

for(i = 0; i < file\_count; i++) {

if(!strcmp(namelist[i]->d\_name, ".") || !strcmp(namelist[i]->d\_name, ".."))

continue;

if(!strcmp(file\_name, namelist[i]->d\_name)) { // 최초 중복되는 경우

sprintf(target\_path, "%s/%s/1\_%s", pwd, TRASH\_FILES, file\_name);

sprintf(trash\_files\_path, "%s/%s/%s", pwd, TRASH\_FILES, file\_name);

rename(trash\_files\_path, target\_path); // 원본 파일 이름 변경

sprintf(target\_path, "%s/%s/1\_%s.txt", pwd, TRASH\_INFO, file\_name);

sprintf(trash\_info\_path, "%s/%s/%s.txt", pwd, TRASH\_INFO, file\_name);

rename(trash\_info\_path, target\_path); // 정보 파일 이름 변경

chdir(pwd);

free(namelist);

return true + 1;

}

sscanf(namelist[i]->d\_name, "%d\_%s", &tmp, target\_path);

if(tmp > false && !strcmp(target\_path, file\_name)) // 다수 중복되는 경우

overlap\_count++;

}

free(namelist);

if(overlap\_count > 0) {

chdir(pwd);

return overlap\_count;

}

chdir(pwd);

return false;

}

void print\_list\_size(file\_node \*head, char \*path, int number, int option\_d, int op\_switch) // 지정 파일 상대 경로 및 크기 출력

{

char \*relative\_path;

file\_node \*now;

now = head;

while(number > 0 && now != NULL) {

relative\_path = now->name + strlen(pwd); // 상대 경로 추출

printf("%-10d.%-s\n", now->size, relative\_path); // 출력

if(op\_switch) {

if(option\_d) { // 옵션이 주어졌을 경우

if(!S\_ISDIR(now->attr.st\_mode)) // 파일인 경우

break; // 탐색 종료

} else break; // 옵션이 주어지지 않은 경우

}

op\_switch = false;

if(now->child != NULL) // 하위 디렉토리 파일들이 존재하면

print\_list\_size(now->child, path, number - 1, option\_d, op\_switch); // 하위 디렉토리 파일 출력

now = now->next; // 같은 레벨의 파일들 이어서 탐색

}

}

void restore\_file(const char \*file\_name, int option\_l) // 휴지통 파일 복원

{

char trash\_info\_path[MAX\_BUFFER\_SIZE];

char trash\_files\_path[MAX\_BUFFER\_SIZE];

char tmp[MAX\_BUFFER\_SIZE];

char date[BUFFER\_SIZE];

char time[BUFFER\_SIZE];

char \*temp;

struct dirent \*\*namelist;

int file\_count;

FILE \*fp;

file\_infos file\_info[50];

int idx;

int overlap;

int i, j;

sprintf(trash\_info\_path, "%s/%s", pwd, TRASH\_INFO);

sprintf(trash\_files\_path, "%s/%s", pwd, TRASH\_FILES);

if(option\_l) { // -l 옵션

idx = 0;

file\_count = scandir(trash\_info\_path, &namelist, NULL, alphasort); // 파일 정보 디렉토리 탐색

for(i = 0; i < file\_count; i++) {

if(!strcmp(namelist[i]->d\_name, ".") || !strcmp(namelist[i]->d\_name, "..")) // 상위 디렉토리 접근자 생략

continue;

sprintf(tmp, "%s/%s/%s", pwd, TRASH\_INFO, namelist[i]->d\_name); // 정보 파일 경로 생성

if((fp = fopen(tmp, "r+")) < 0) { // 정보 파일 읽기 모드로 열기

fprintf(stderr, "fopen error for %s\n", namelist[i]->d\_name);

continue;

}

fseek(fp, 13, SEEK\_SET); // 정보 파일 제목 생략

// 파일 경로 추출

fscanf(fp, "%s\n", file\_info[idx].path);

// 파일 삭제 시간 추출

fscanf(fp, "D : %s %s\n", date, time); // 삭제 시간

file\_info[idx++].d\_tm = get\_tm(date, time);

fclose(fp); // 파일 닫기

}

free(namelist); // 메모리 할당 해제

sort\_info\_oldest(file\_info, idx); // 구조체 배열 삭제 시간 순 오름차순 정렬

for(i = 0; i < idx; i++) // 출력

printf("%d. %-10s %s\n", i + 1, get\_file\_name(file\_info[i].path), make\_time\_format(file\_info[i].d\_tm));

}

// 파일 복원 우선 순위

// 1. 휴지통 디렉토리에 존재하는 경우

// 2. 휴지통 디렉토리에 중복으로 존재하는 경우

idx = 0;

file\_count = scandir(trash\_info\_path, &namelist, NULL, alphasort); // 파일 정보 디렉토리 탐색

for(i = 0; i < file\_count; i++) {

if(!strcmp(namelist[i]->d\_name, ".") || !strcmp(namelist[i]->d\_name, "..")) // 상위 디렉토리 접근자 생략

continue;

memset(tmp, 0, MAX\_BUFFER\_SIZE);

strncpy(tmp, namelist[i]->d\_name, strlen(namelist[i]->d\_name) - 4); // 정보 파일명에서 원본 파일명 추출

// 1. 휴지통에 해당 파일이 존재하는 경우

if(!strcmp(file\_name, tmp)) {

sprintf(tmp, "%s/%s/%s", pwd, TRASH\_INFO, namelist[i]->d\_name); // 정보 파일 경로 생성

if((fp = fopen(tmp, "r+")) < 0) { // 정보 파일 읽기 모드로 열기

fprintf(stderr, "fopen error for %s\n", namelist[i]->d\_name);

continue;

}

fseek(fp, 13, SEEK\_SET); // 정보 파일 제목 생략

// 파일 경로 추출

fscanf(fp, "%s\n", file\_info[idx].path);

fclose(fp); // 파일 닫기

temp = get\_file\_path(file\_info[idx].path, file\_name);

if(access(temp, F\_OK) < 0) { // 복원 지점까지의 경로가 존재하지 않을 경우

fprintf(stderr, "recover: %s path doesn't exist\n", temp);

return;

}

if(access(file\_info[idx].path, F\_OK) < 0) { // a. 복원 지점에 똑같은 이름의 파일이 존재하지 않을 경우

// 정보 파일 삭제

unlink(tmp);

// 원본 파일 복원

sprintf(tmp, "%s/%s/%s", pwd, TRASH\_FILES, file\_name); // 휴지통 원본 파일 경로 생성

rename(tmp, file\_info[idx].path); // 휴지통 원본 파일을 복원 지점으로 이동

} else { // b. 복원 지점에 똑같은 이름의 파일이 존재하는 경우

j = 1; // 중복 파일 카운트

temp = get\_file\_path(file\_info[idx].path, file\_name); // 복원 지점 경로 추출, path/

while(true) {

sprintf(tmp, "%s%d\_%s", temp, j, file\_name); // 복원 파일 이름 생성

if(access(tmp, F\_OK) < 0) { // 복원 지점에 해당 이름의 파일이 존재하지 않는 경우

// 정보 파일 삭제

sprintf(trash\_info\_path, "%s/%s/%s.txt", pwd, TRASH\_INFO, file\_name); // 휴지통 정보 파일 경로 생성

unlink(trash\_info\_path);

// 원본 파일 복원

sprintf(trash\_files\_path, "%s/%s/%s", pwd, TRASH\_FILES, file\_name); // 휴지통 원본 파일 경로 생성

rename(trash\_files\_path, tmp); // 휴지통 원본 파일을 복원지점으로 이동

break;

}

j++; // 중복 파일 카운트 증가

}

}

free(namelist); // 메모리 할당 해제

return;

}

memset(tmp, 0, MAX\_BUFFER\_SIZE);

strncpy(tmp, namelist[i]->d\_name, strlen(namelist[i]->d\_name) - 4);

sscanf(tmp, "%d\_%s", &overlap, tmp); // 중복 파일 카운트, 파일 이름 추출

// 2. 휴지통에 해당 파일이 중복으로 존재하는 경우

if(!strcmp(file\_name, tmp)) {

sprintf(tmp, "%s/%s/%s", pwd, TRASH\_INFO, namelist[i]->d\_name); // 정보 파일 경로 생성

if((fp = fopen(tmp, "r+")) < 0) { // 정보 파일 읽기 모드로 열기

fprintf(stderr, "fopen error for %s\n", namelist[i]->d\_name);

continue;

}

fseek(fp, 13, SEEK\_SET); // 정보 파일 제목 생략

file\_info[idx].num = overlap; // 중복 파일 카운트 저장

// 파일 경로 추출

fscanf(fp, "%s\n", file\_info[idx].path);

// 파일 시간 정보 추출

fscanf(fp, "D : %s %s\n", date, time); // 삭제 시간

file\_info[idx].d\_tm = get\_tm(date, time);

fscanf(fp, "M : %s %s\n", date, time); // 수정 시간

file\_info[idx].m\_tm = get\_tm(date, time);

fclose(fp); // 파일 닫기

idx++; // 구조체 카운트 증가

}

} // 휴지통 탐색 완료

free(namelist); // 메모리 할당 해제

sort\_info\_order(file\_info, idx); // 구조체 배열 중복 파일 카운트순 오름차순 정렬

// 중복된 파일 중 선택 복원

for(i = 0; i < idx; i++)

printf("%d. %-10s D : %s M : %s\n", i + 1, file\_name, make\_time\_format(file\_info[i].d\_tm), make\_time\_format(file\_info[i].m\_tm));

printf("Choose : ");

scanf("%d", &i);

getchar();

if(i < 1 || i > idx) {

fprintf(stderr, "recover: invalid input NUMBER\n");

return;

}

temp = get\_file\_path(file\_info[i - 1].path, file\_name);

if(access(temp, F\_OK) < 0) { // 복원 지점까지의 경로가 존재하지 않을 경우

fprintf(stderr, "recover: %s path doesn't exist\n", temp);

return;

}

if(access(file\_info[i - 1].path, F\_OK) < 0) { // c. 복원 지점에 똑같은 이름의 파일이 존재하지 않는 경우

// 정보 파일 삭제

sprintf(tmp, "%s/%s/%d\_%s.txt", pwd, TRASH\_INFO, i, file\_name); // 휴지통 정보 파일 경로 생성

unlink(tmp);

// 원본 파일 복구

sprintf(tmp, "%s/%s/%d\_%s", pwd, TRASH\_FILES, i, file\_name); // 휴지통 원본 파일 경로 생성

rename(tmp, file\_info[i - 1].path); // 휴지통 원본 파일을 복원 지점으로 이동

} else { // d. 복원 지점에 똑같은 이름의 파일이 존재하는 경우

j = 1; // 중복 파일 카운트

temp = get\_file\_path(file\_info[i - 1].path, file\_name); // 복원 지점 경로 생성, path/

while(true) {

sprintf(tmp, "%s%d\_%s", temp, j, file\_name); // 복원 파일 이름 생성

if(access(tmp, F\_OK) < 0) { // 복원 지점에 해당 이름의 파일이 존재하지 않는 경우

// 정보 파일 삭제

sprintf(trash\_info\_path, "%s/%s/%d\_%s.txt", pwd, TRASH\_INFO, i, file\_name);

unlink(trash\_info\_path);

// 원본 파일 복원

sprintf(trash\_files\_path, "%s/%s/%d\_%s", pwd, TRASH\_FILES, i, file\_name);

rename(trash\_files\_path, tmp);

break;

}

j++; // 중복 파일 카운트 증가

}

}

refresh\_trash(file\_name, idx - 1, i); // 휴지통 중복 파일 재정리

}

char \*get\_file\_path(char \*path, const char \*file\_name) // 파일 경로 추출

{

char \*ptr;

static char file\_path[BUFFER\_SIZE];

char \*tmp;

strcpy(file\_path, path);

ptr = strstr(file\_path, file\_name);

while(ptr != NULL) {

tmp = ptr;

ptr = strstr(ptr+1, file\_name);

}

memset(file\_path, 0, BUFFER\_SIZE);

strncpy(file\_path, path, tmp - file\_path);

return (char\*)file\_path;

}

void sort\_info\_oldest(file\_infos \*file\_info, int idx) // 중복 파일 구조체 삭제 시간 오름차순 정렬

{

int i, j;

char tmp\_path[BUFFER\_SIZE];

struct tm tmp\_tm;

for(i = 0; i < idx; i++)

for(j = i + 1; j < idx; j++)

if(mktime(&file\_info[i].d\_tm) > mktime(&file\_info[j].d\_tm)) {

strcpy(tmp\_path, file\_info[i].path);

strcpy(file\_info[i].path, file\_info[j].path);

strcpy(file\_info[j].path, tmp\_path);

tmp\_tm = file\_info[i].d\_tm;

file\_info[i].d\_tm = file\_info[j].d\_tm;

file\_info[j].d\_tm = tmp\_tm;

}

}

void sort\_info\_order(file\_infos \*file\_info, int idx) // 파일정보 구조체 파일 번호 오름차순 정렬

{

int i, j;

int tmp\_num;

char tmp\_path[BUFFER\_SIZE];

struct tm tmp\_tm;

for(i = 0; i < idx; i++)

for(j = i + 1; j < idx; j++)

if(file\_info[i].num > file\_info[j].num) {

strcpy(tmp\_path, file\_info[i].path);

strcpy(file\_info[i].path, file\_info[j].path);

strcpy(file\_info[j].path, tmp\_path);

tmp\_tm = file\_info[i].d\_tm;

file\_info[i].d\_tm = file\_info[j].d\_tm;

file\_info[j].d\_tm = tmp\_tm;

tmp\_num = file\_info[i].num;

file\_info[i].num = file\_info[j].num;

file\_info[j].num = tmp\_num;

}

}

void refresh\_trash(const char \*file\_name, int idx, int delete\_idx) // 삭제 후 중복 파일 번호 재정렬

{

char trash\_info\_path[MAX\_BUFFER\_SIZE];

char trash\_files\_path[MAX\_BUFFER\_SIZE];

char tmp1[BUFFER\_SIZE];

char tmp2[BUFFER\_SIZE];

struct dirent \*\*namelist;

int file\_count;

int overlap;

int i;

sprintf(trash\_info\_path, "%s/%s", pwd, TRASH\_INFO);

sprintf(trash\_files\_path, "%s/%s", pwd, TRASH\_FILES);

// 중복 파일이 더이상 존재하지 않는 경우

if(idx == 1 && delete\_idx == 1) { // 중복 파일 2개중 1번을 삭제한 경우

chdir(TRASH\_INFO);

sprintf(tmp1, "2\_%s.txt", file\_name);

sprintf(tmp2, "%s.txt", file\_name);

rename(tmp1, tmp2);

chdir(pwd);

chdir(TRASH\_FILES);

sprintf(tmp1, "2\_%s", file\_name);

rename(tmp1, file\_name);

chdir(pwd);

} else if(idx == 1 && delete\_idx == 2) { // 중복 파일 2개중 2번을 삭제한 경우

chdir(TRASH\_INFO);

sprintf(tmp1, "1\_%s.txt", file\_name);

sprintf(tmp2, "%s.txt", file\_name);

rename(tmp1, tmp2);

chdir(pwd);

chdir(TRASH\_FILES);

sprintf(tmp1, "1\_%s", file\_name);

rename(tmp1, file\_name);

chdir(pwd);

} else { // 중복 파일 3개 이상

file\_count = scandir(trash\_info\_path, &namelist, NULL, alphasort);

for(i = 0; i < file\_count; i++) {

if(!strcmp(namelist[i]->d\_name, ".") || !strcmp(namelist[i]->d\_name, ".."))

continue;

memset(tmp1, 0, BUFFER\_SIZE);

strncpy(tmp1, namelist[i]->d\_name, strlen(namelist[i]->d\_name) - 4);

sscanf(tmp1, "%d\_%s", &overlap, tmp1); // 중복 파일 카운트, 파일 이름 추출

if(overlap >= delete\_idx && !strcmp(tmp1, file\_name)) {

chdir(TRASH\_INFO);

sprintf(tmp1, "%d\_%s.txt", overlap, file\_name);

sprintf(tmp2, "%d\_%s.txt", overlap - 1, file\_name);

rename(tmp1, tmp2);

chdir(pwd);

chdir(TRASH\_FILES);

sprintf(tmp1, "%d\_%s", overlap, file\_name);

sprintf(tmp2, "%d\_%s", overlap - 1, file\_name);

rename(tmp1, tmp2);

chdir(pwd);

}

}

free(namelist);

}

}

void print\_list\_tree(file\_node \*head, int level, int level\_check[], int is\_root) // 모니터링 파일 목록 트리 출력

{

file\_node \*now;

char \*file\_name;

now = head;

while(now != NULL) {

file\_name = get\_file\_name(now->name);

if(is\_root) { // 루트 디렉토리 노드일 경우 디렉토리 이름만 출력 후 하위 파일 노드로 이동

printf("%s\n", file\_name); // 파일명 출력

if(now->child != NULL) // 모니터링 디렉토리 하위에 파일이 존재하는 경우

now = now->child; // 하위 파일 노드로 이동

else // 모니터링 디렉토리 하위에 파일이 존재하지 않는 경우

break;

is\_root = false; // 루트 확인 해제

level\_check[level++] = true; // 레벨 증가

continue;

}

print\_tree\_indent(level, level\_check);

if(now->next != NULL)

printf("├──%s\n", file\_name);

else

printf("└──%s\n", file\_name);

if(S\_ISDIR(now->attr.st\_mode)) { // 같은 레벨의 파일이 디렉토리일 경우

if(now->next != NULL) { // 같은 레벨의 파일들이 더 존재할 경우

if(now->child != NULL) {

level\_check[level] = true;

print\_list\_tree(now->child, level + 1, level\_check, is\_root);

level\_check[level] = false;

}

} else // 같은 레벨의 파일들이 더 존재하지 않을 경우

if(now->child != NULL)

print\_list\_tree(now->child, level + 1, level\_check, is\_root);

}

now = now->next;

}

}

void print\_tree\_indent(int level, int level\_check[]) // 트리 출력 보조 함수

{

int i;

for(i = 1; i < level; i++) {

if(level\_check[i])

printf("│ ");

else

printf(" ");

}

}

char \*rtrim(char \*\_str) // 문자열 오른쪽 공백 제거

{

char tmp[MAX\_BUFFER\_SIZE];

char \*end;

strcpy(tmp, \_str);

end = tmp + strlen(tmp) - 1;

while(end != \_str && isspace(\*end))

--end;

\*(end + 1) = '\0';

\_str = tmp;

return \_str;

}

char \*ltrim(char \*\_str) // 문자열 왼쪽 공백 제거

{

char \*start = \_str;

while(\*start != '\0' && isspace(\*start))

++start;

\_str = start;

return \_str;

}

void to\_lower\_case(char \*str) // 문자열 소문자 변환

{

int i = 0;

while(str[i]) {

if(str[i] >= 'A' && str[i] <= 'Z'){

str[i] = str[i]+32;

}

i++;

}

}

void free\_command(commands command) // 명령행 구조체 초기화

{

int i;

for(i = 0; i < command.argc; i++)

free(command.argv[i]);

free(command.argv);

}

void print\_usage(void) // 사용법 출력

{

printf("Usage : [COMMAND]\n");

printf("Command : \n");

printf(" DELETE <FILENAME> <END\_TIME> [OPTION] delete the file at the END\_TIME\n");

printf(" Option : -i delete file without moving 'trash' directory\n");

printf(" -r re-confirm when the specified time, delete or not to delete\n\n");

printf(" SIZE <FILENAME> [OPTION] print relative file path and size\n");

printf(" Option : -d <NUMBER> print as much as NUMBER level\n\n");

printf(" RECOVER <FILENAME> [OPTION] restore deleted files to their original path\n");

printf(" Option : -l print file list in 'trash' directory sort by oldest deletion time before command execute\n\n");

printf(" TREE print a list of monitoring files in a tree format.\n");

printf(" EXIT program exit\n");

printf(" HELP print usage\n");

}