컴퓨터학부 20160548 이승현

1. 과제 개요

ssu_mntr 프로그램은 디렉토리를 지정하여 해당 디렉토리를 모니터링하며 변경사항을 로그로 남기고, 지정한 디렉토리의 파일들을 삭제, 복구, 조회하는 기능을 제공하는 프로그램이다.

변경사항 모니터링 기능은 백그라운드에서 디몬 프로세스가 담당하며 지정된 디렉토리 내의 파일들에 대해서 수정, 삭제, 추가가 발생하면 해당 내용을 log.txt에 로그로 남긴다.

파일 삭제, 복구, 조회기능은 DELETE, RECOVER, TREE, SIZE 명령어 등으로 구현된다. 파일 삭제(DELETE 명령어), 복구(RECOVER 명령어) 기능은 trash 디렉토리를 휴지통으로 사용하여 삭제한 파일을 임시 보관, 복구한다. 조회 기능에는 두 가지 명령어가 있다. TREE 명령어로 지정한 디렉토리의 구조를 트리 구조로 표현된 그림으로 보여주며, SIZE 명령어로는 지정한 디렉토리 내의 파일 크기를 확인할 수 있다. 그 외의 명령어로는 EXIT 명령어와 HELP 명령어가 있다. EXIT 명령어는 프롬프트를 종료하는 명령어, HELP 명령어는 프로그램 사용법을 출력하는 명령어이다.

이 과제에서 구현해야 하는 항목은 다음과 같다.

가) ssu_mntr 동작 - 20
나) delete 명령어 - 10
다) delete -i 옵션 - 5
라) delete -r 옵션 - 5
마) size 명령어 - 10
바) size -d 옵션 - 5
사) recover 명령어 - 20
아) recover -l 옵션 - 5
자) tree 명령어 - 10
차) exit 명령어 - 5
카) help 명령어 - 5

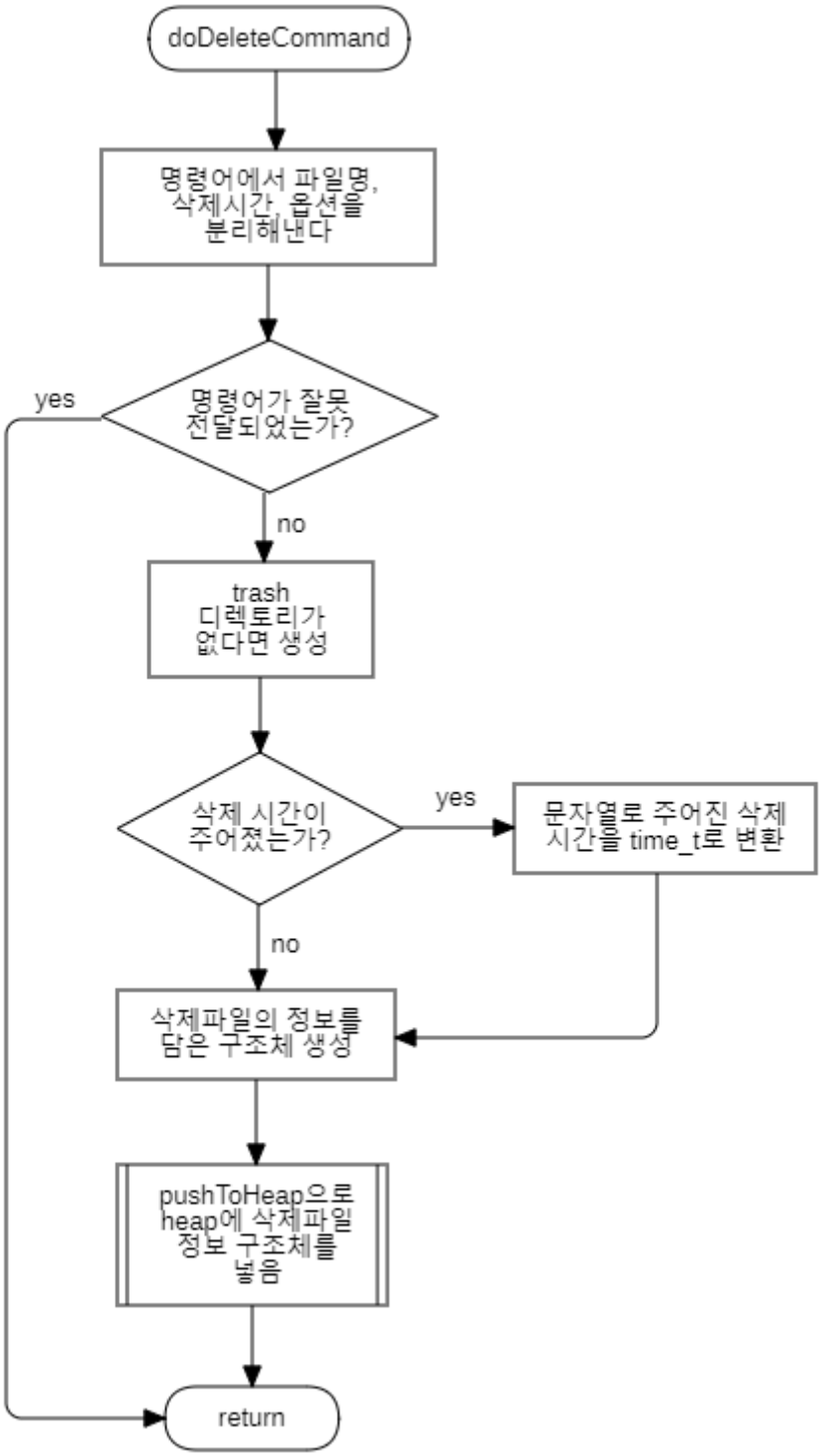
위 기능들을 모두 구현하였다.

코드는 ssu_mntr.c, monitor.c 파일 두개로 나뉘어 있다. Monitor.c 파일에는 지정 파일을 모니터링하는 디몬 프로세스의 코드가 들어있고, ssu_mntr.c 파일에는 지정 디렉토리 모니터링 외의 모든 기능들의 코드가 들어있다. (ssu_mntr.c에는 '나', '다', '라', '마', '바', '사', '아', '자', '차', '카' 기능이, monitor.c에는 '가' 기능이 구현 되어있다.)

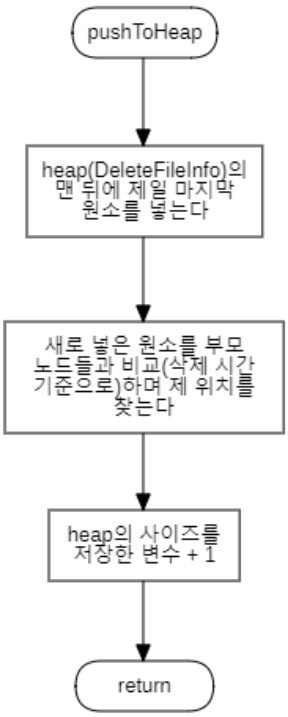
2. 설계

<ssu_mntr.c> - 모니터링 외의 모든 기능들 코드

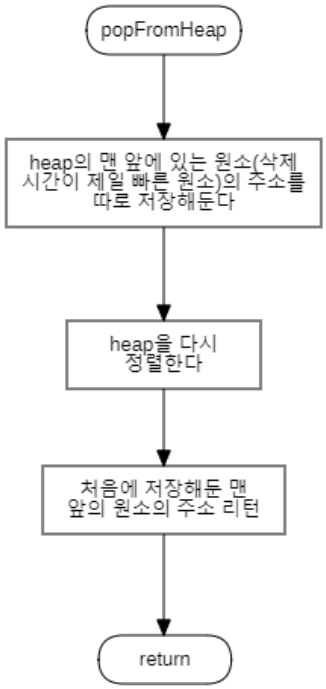
void doDeleteCommand(); // delete 명령어를 수행하는 함수



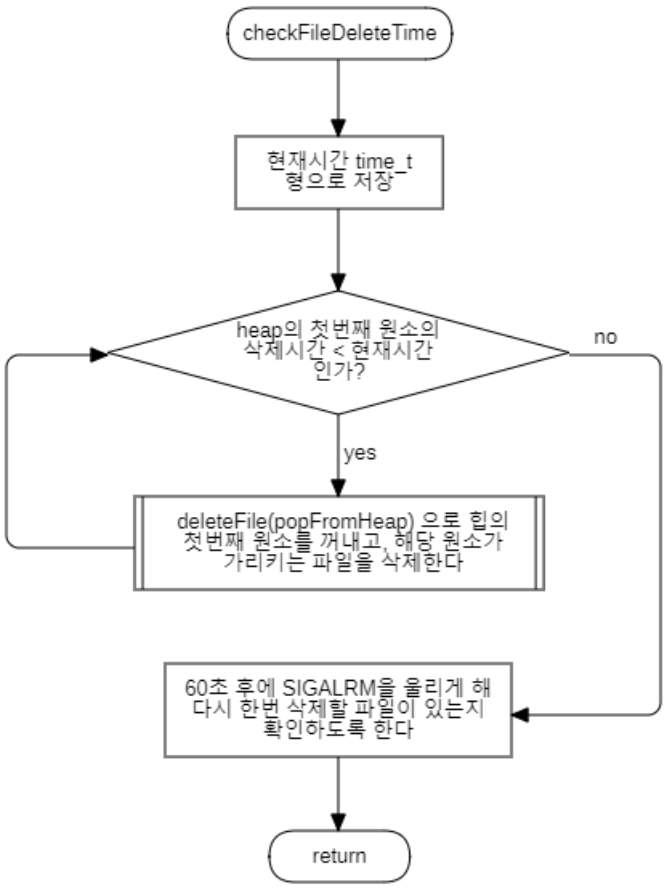
void pushToHeap(struct DeleteFileInfo* newElement); // DeleteFileHeap에 새로운 원소를 추가하는 함수



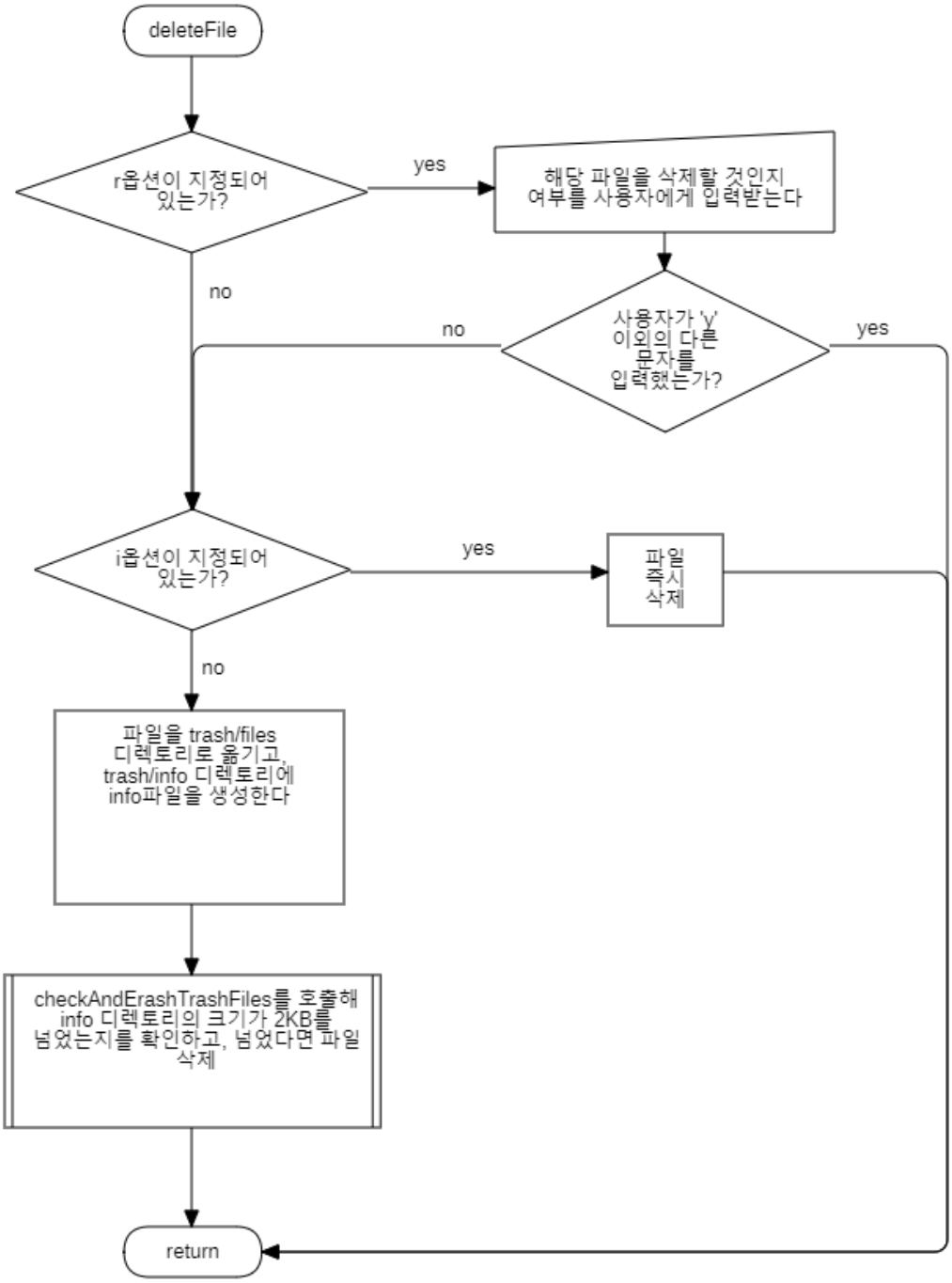
struct DeleteFileInfo *popFromHeap(); // DeleteFileHeap에서 원소를 하나 빼오는 함수



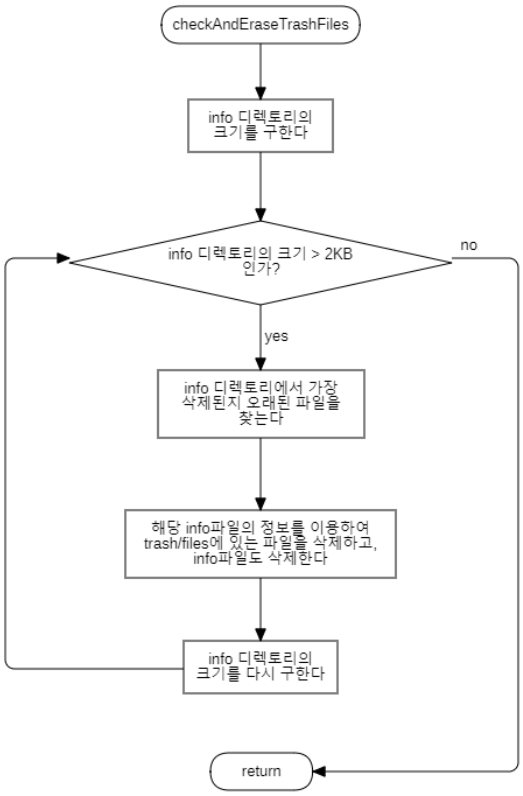
void checkFileDeleteTime(); // 삭제할 파일이 있는지 확인하는 함수



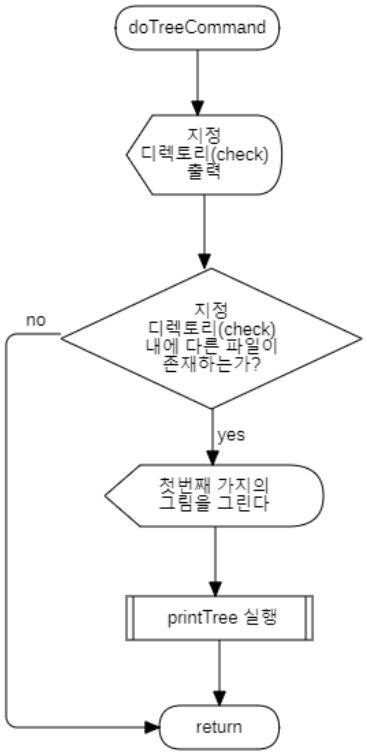
void deleteFile(struct DeleteFileInfo *deleteFileInfo); // 파일을 삭제하는 함수



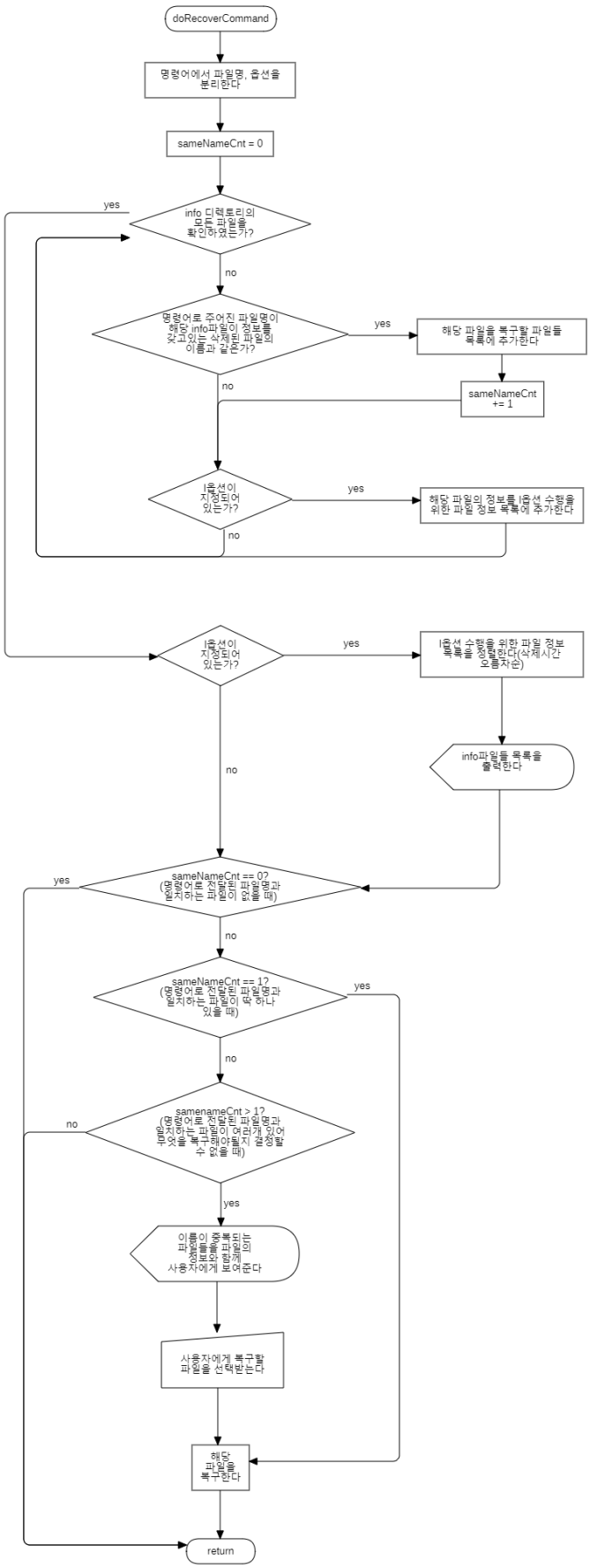
void checkAndEraseTrashFiles(); // trash/info 디렉토리의 크기가 2KB를 넘어가면 오래된 파일부터 삭제하는 함수



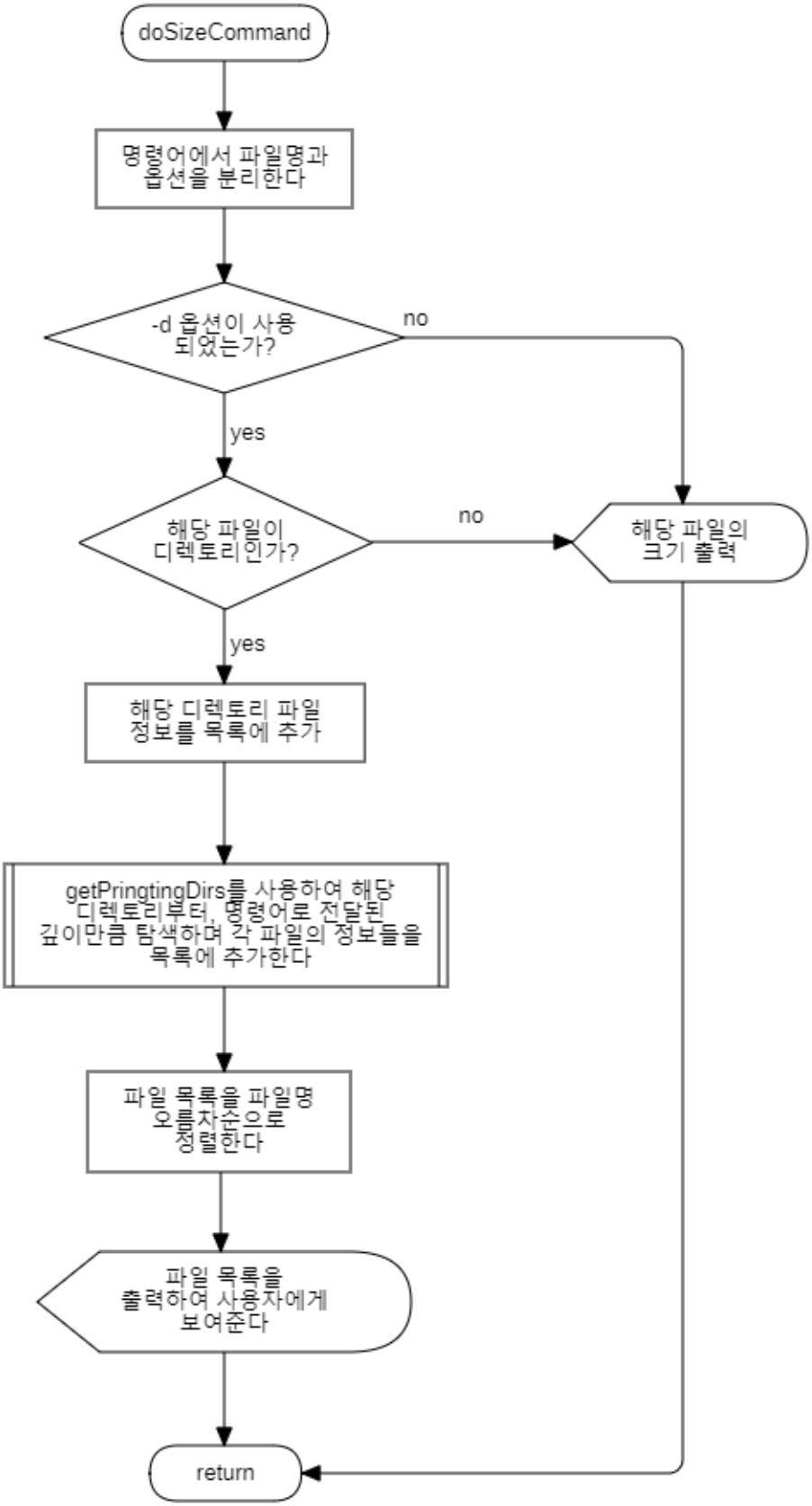
void doTreeCommand(); // tree 명령어를 수행하는 함수



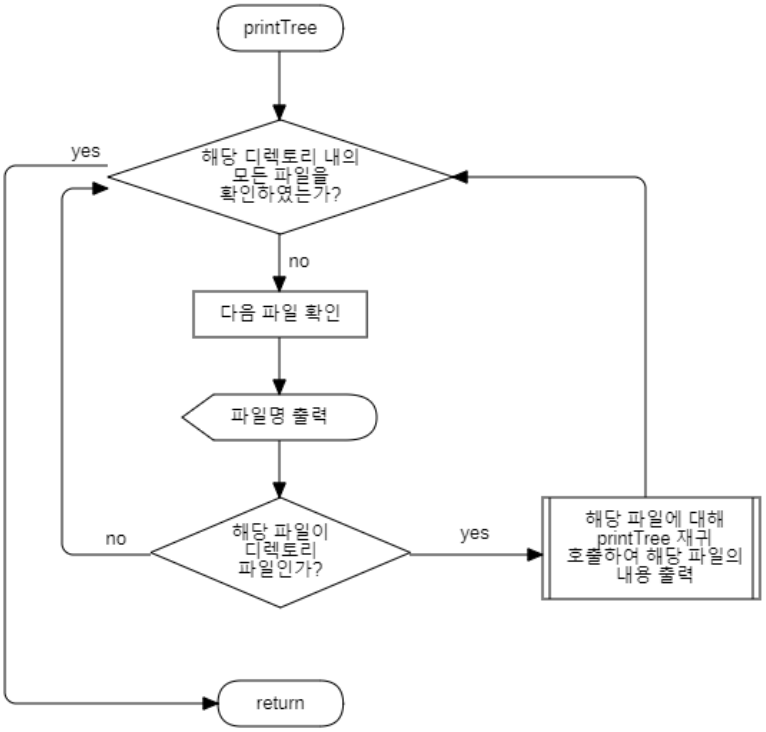
void doRecoverCommand(); // recover 명령어를 수행하는 함수



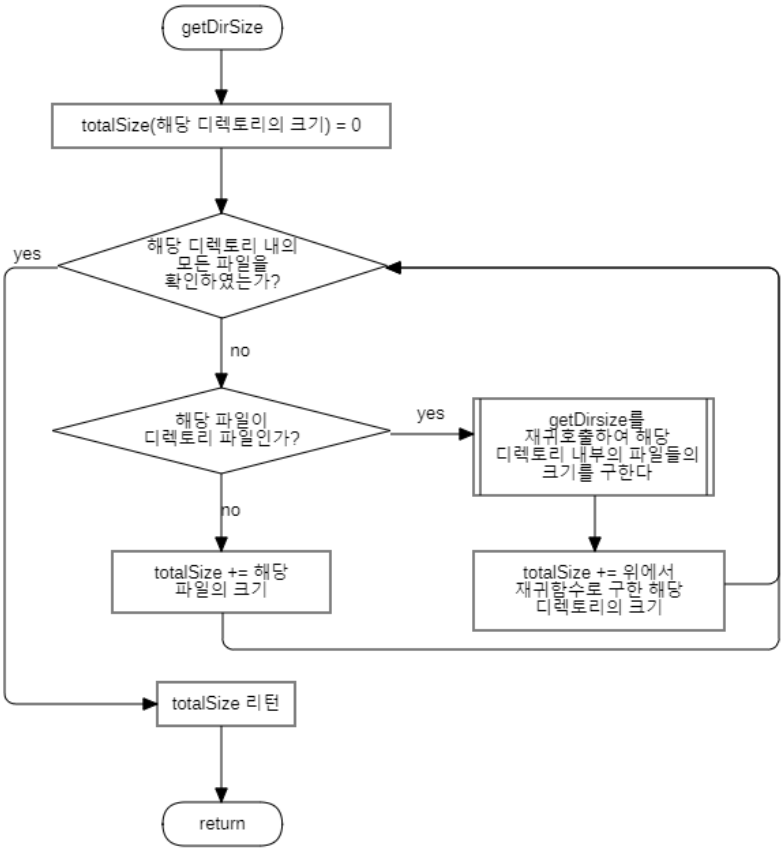
void doSizeCommand(); // size 명령어를 수행하는 함수



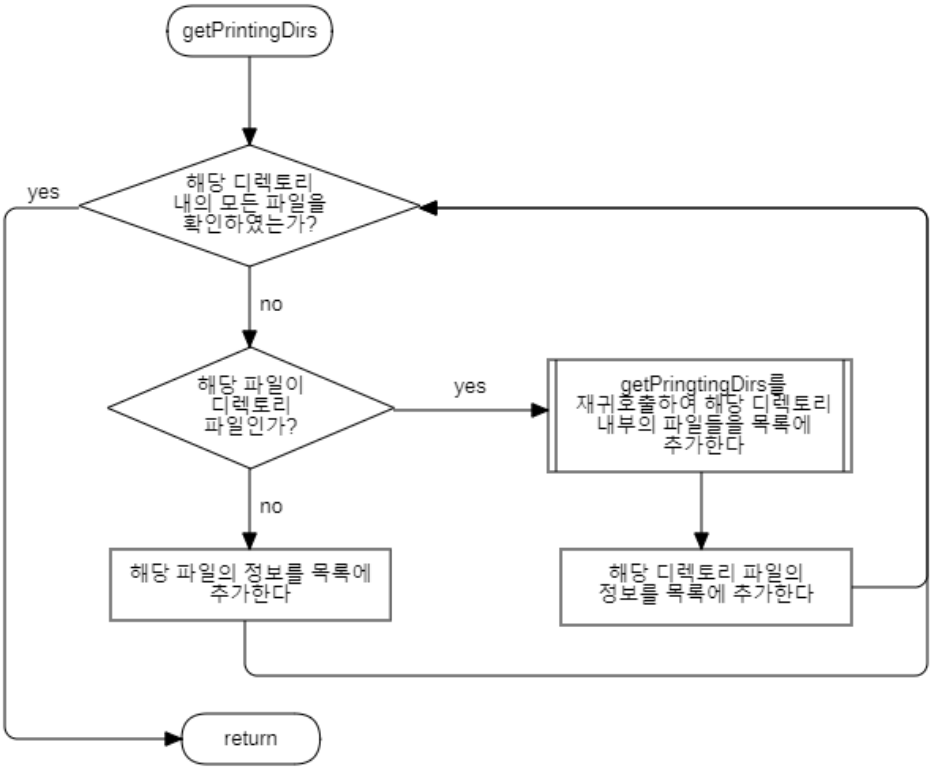
void printTree(const char *dirname, int depth); // tree 명령어에서 tree를 출력하는 재귀호출 함수



off_t getDirSize(const char *dirname); // 해당 디렉토리의 크기를 구하는 함수

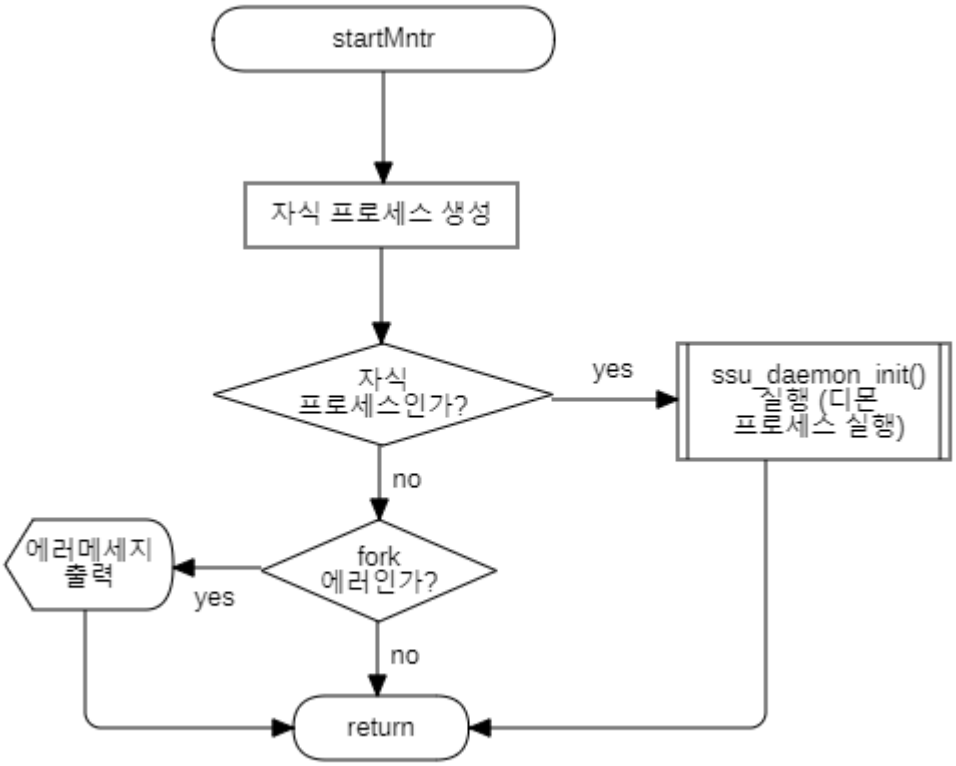


void getPrintingDirs(const char *dirname, const char *dirpath, int depth, int maxDepth); // size 명령어에 의해 출력될 파일들을 확인해 fileSizes 구조체(출력할 파일들 목록)에 추가하는 함수

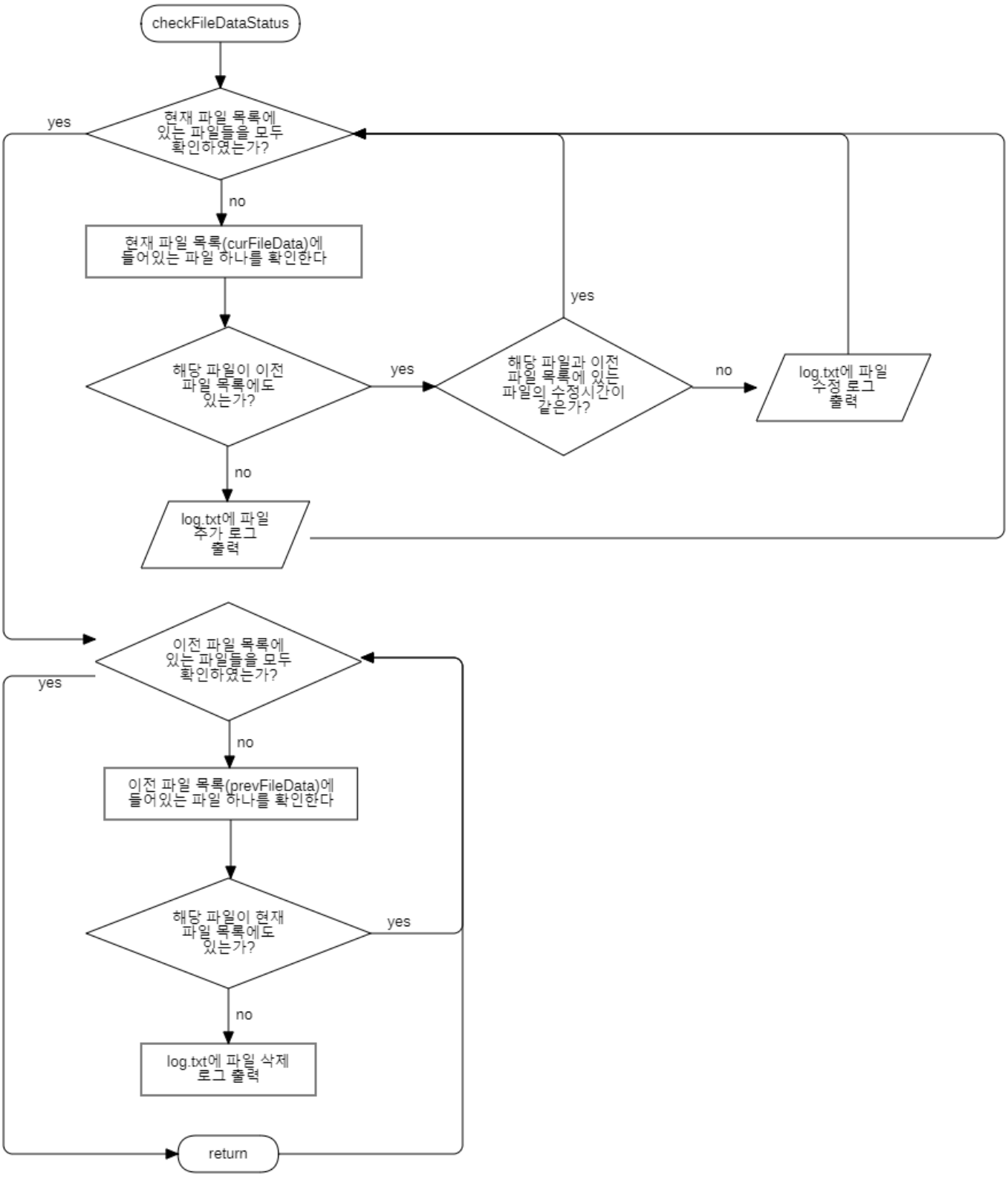


<monitor.c> - 지정 디렉토리를 모니터링하는 디몬 프로세스의 코드

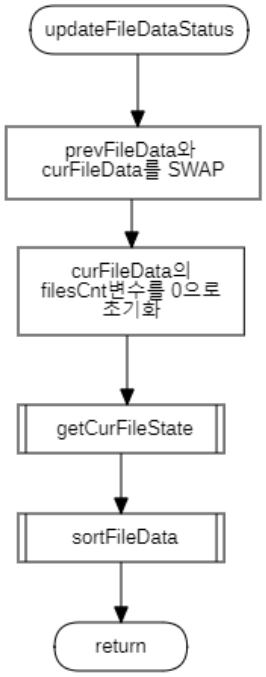
void startMntr(void);



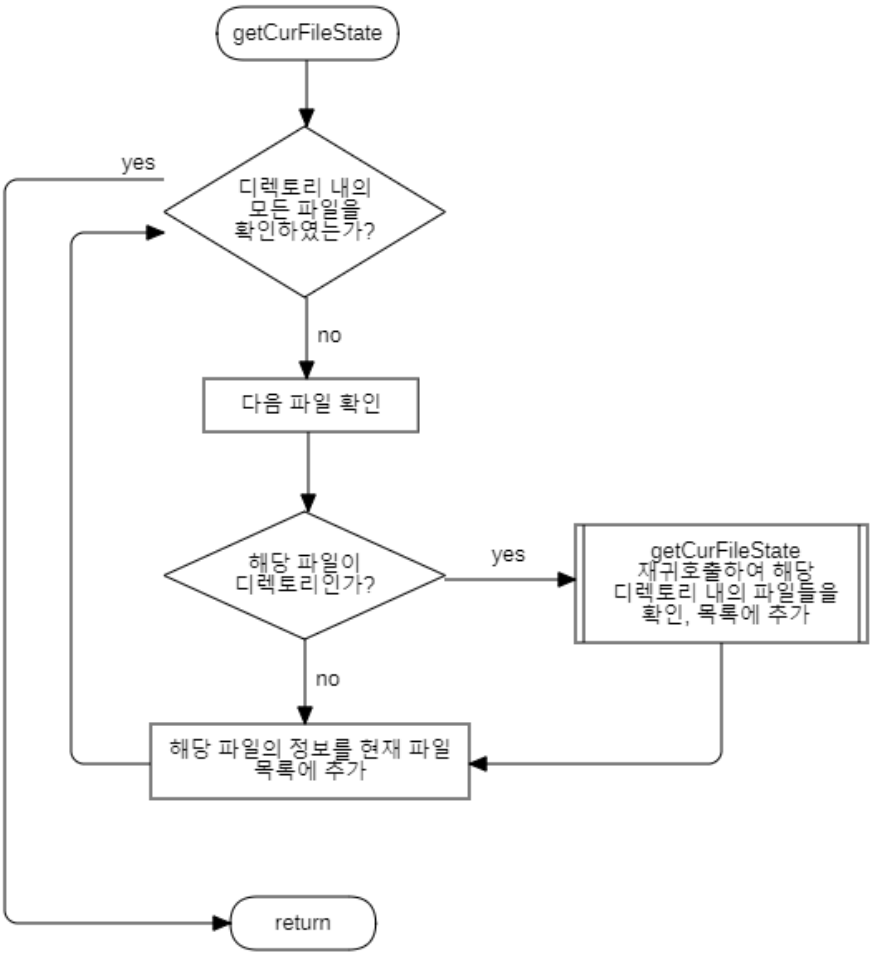
void checkFileDataStatus(const struct FileData *prevFileData, const struct FileData *curFileData); // 이전 파일들 목록과 갱신된 파일들 목록을 확인하여 수정, 추가, 삭제된 파일이 있는지 확인하는 함수



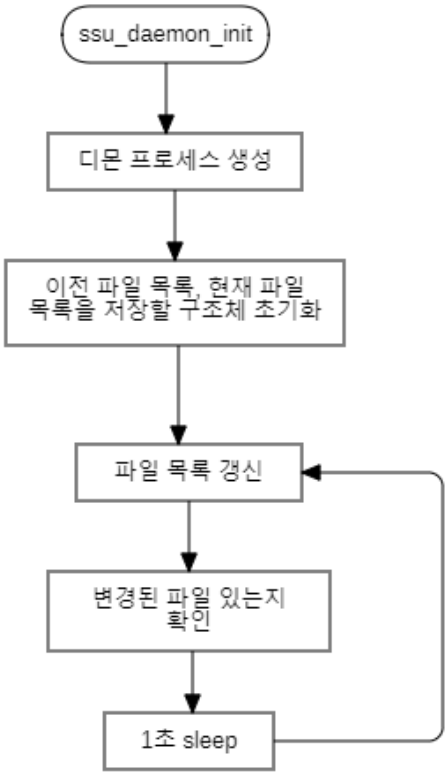
void updateFileDataStatus(struct FileData **prevFileData, struct FileData **curFileData); // 파일 목록을 갱신하는 함수



void getCurFileState(struct FileData *fileData, const char *dirname); // 파일들 상태를 가져오는 재귀호출 함수



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



3. 구현

<ssu_mntr.c>

void doDeleteCommand(); // delete 명령어를 수행하는 함수

전달인자: 없음

리턴 값: 없음

void doExitCommand(); // exit 명령어를 수행하는 함수

전달인자: 없음

리턴 값: 없음

void doHelpCommand(); // help 명령어를 수행하는 함수

전달인자: 없음

리턴 값: 없음

void doTreeCommand(); // tree 명령어를 수행하는 함수

전달인자: 없음

리턴 값: 없음

void doRecoverCommand(); // recover 명령어를 수행하는 함수

전달인자: 없음

리턴 값: 없음

void doSizeCommand(); // size 명령어를 수행하는 함수

전달인자: 없음

리턴 값: 없음

void printTree(const char *dirname, int depth); // tree 명령어에서 tree를 출력하는 재귀호출 함수

전달인자:

const char *dirname – 탐색할 디렉토리의 이름

int depth – 지정한 디렉토리(check) 부터 현재 디렉토리까지의 깊이

리턴 값: 없음

void pushToHeap(struct DeleteFileInfo* newElement); // DeleteFileHeap에 새로운 원소를 추가하는 함수

전달인자: struct DeleteFileInfo* newElement – 삭제할 파일의 정보가 담겨있는 구조체 struct DeleteFileInfo의 포인터

리턴 값: 없음

struct DeleteFileInfo *popFromHeap(); // DeleteFileHeap에서 원소를 하나 빼오는 함수

전달인자: 없음

리턴 값: 삭제할 파일의 정보가 담겨있는 구조체 struct DeleteFileInfo의 포인터

void alarmHandlercheckFileDeleteTime(); // SIGALRM이 전달됐을 때 삭제할 파일이 있는지 확인하는 함수

전달인자: 없음

리턴 값: 없음

void checkFileDeleteTime(); // 삭제할 파일이 있는지 확인하는 함수

전달인자: 없음

리턴 값: 없음

void deleteFile(struct DeleteFileInfo *deleteFileInfo); // 파일을 삭제하는 함수

전달인자: 삭제할 파일의 정보가 담겨있는 구조체 struct DeleteFileInfo의 포인터

리턴 값: 없음

void checkAndEraseTrashFiles(); // trash/info 디렉토리의 크기가 2KB를 넘어가면 오래된 파일부터 삭제하는 함수

전달인자: 없음

리턴 값: 없음

int compareInfoFile(const void *a, const void *b); // infoFile구조체 배열을 qsort 함수로 sort할 때 사용하는 비교 함수

전달인자:

const void *a – void*로 형 변환된 struct DeleteFileInfo의 포인터

const void *b - void*로 형 변환된 struct DeleteFileInfo의 포인터

리턴 값: 비교 결과 값

off_t getDirSize(const char *dirname); // 해당 디렉토리의 크기를 구하는 함수

전달인자: const char *dirname - 크기를 구할 디렉토리의 이름

리턴 값: 디렉토리의 크기를 off_t 형으로 리턴

void getPrintingDirs(const char *dirname, const char *dirpath, int depth, int maxDepth); // size 명령어에 의해 출력될 파일들을 확인해 fileSizes 구조체에 추가하는 함수

전달인자:

const char *dirname – 현재 디렉토리의 이름

const char *dirpath – 모니터링을 위해 지정한 디렉토리(check)에서부터 현재 디렉토리까지의 경로

int depth – 현재 탐색중인 디렉토리의 깊이

int maxDepth – 탐색할 최대 깊이

리턴 값: 없음

int compareFileSize(const void *a, const void *b); // FileSize 구조체 배열을 qsort 함수로 sort할 때 사용하는 비교 함수

전달인자:

const void *a - void*로 형 변환된 struct FileSize(파일명과 크기가 저장된 구조체)의 포인터

const void *b - void*로 형 변환된 struct FileSize(파일명과 크기가 저장된 구조체)의 포인터

리턴 값: 비교 결과 값

<monitor.c>

void startMntr(void); // 지정 디렉토리를 모니터링하는 디몬 프로세스를 실행하는 함수

전달인자: 없음

리턴 값: 없음

void checkFileDataStatus(const struct FileData *prevFileData, const struct FileData *curFileData); // 이전 파일들 목록과 갱신된 파일들 목록을 확인하여 수정, 추가, 삭제된 파일이 있는지 확인하는 함수

전달인자:

const struct FileData *prevFileData – 이전 파일 목록을 담아 놓는 struct FileData 구조체의 포인터

const struct FileData *curFileData - 현재 파일 목록을 담아 놓는 struct FileData 구조체의 포인터

리턴 값: 없음

void updateFileDataStatus(struct FileData **prevFileData, struct FileData **curFileData); // 파일 목록을 갱신하는 함수

전달인자:

struct FileData **prevFileData – 이전 파일 목록을 담아 놓는 struct FileData 구조체의 이중 포인터

struct FileData **curFileData - 현재 파일 목록을 담아 놓는 struct FileData 구조체의 이중 포인터

리턴 값: 없음

void getCurFileState(struct FileData *fileData, const char *dirname); // 파일들 상태를 가져오는 재귀호출 함수

전달인자:

struct FileData *fileData – 새롭게 갱신된 파일들의 목록을 담아 놓을 struct FileData 구조체의 포인터

const char *dirname – 확인할 디렉토리의 이름

리턴 값: 없음

void printFileData(const struct FileData *fileData); // 파일 데이터를 출력하는 함수, 디버그용으로 사용

전달인자: const struct FileData *fileData – 출력할 파일 목록을 담아 놓은 struct FileData 구조체의 포인터

리턴 값: 없음

void sortFileData(struct FileData *fileData); // 파일 데이터를 파일명 기준으로 정렬하는 함수

전달인자: struct FileData *fileData – 정렬할 파일 목록을 담아 놓은 struct FileData 구조체의 포인터

리턴 값: 없음

int compareFileInfo(const void *a, const void *b); // 파일 데이터를 정렬할 때 사용하는 비교함수

전달인자:

const void *a – void*형으로 변환된 Struct FileInfo의 포인터

const void *b – void*형으로 변환된 Struct FileInfo의 포인터

리턴 값: 비교 결과 값

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

전달인자: const char *path – 디몬 프로세스의 작업 시작 경로

리턴 값: 정상 종료 시 0 리턴

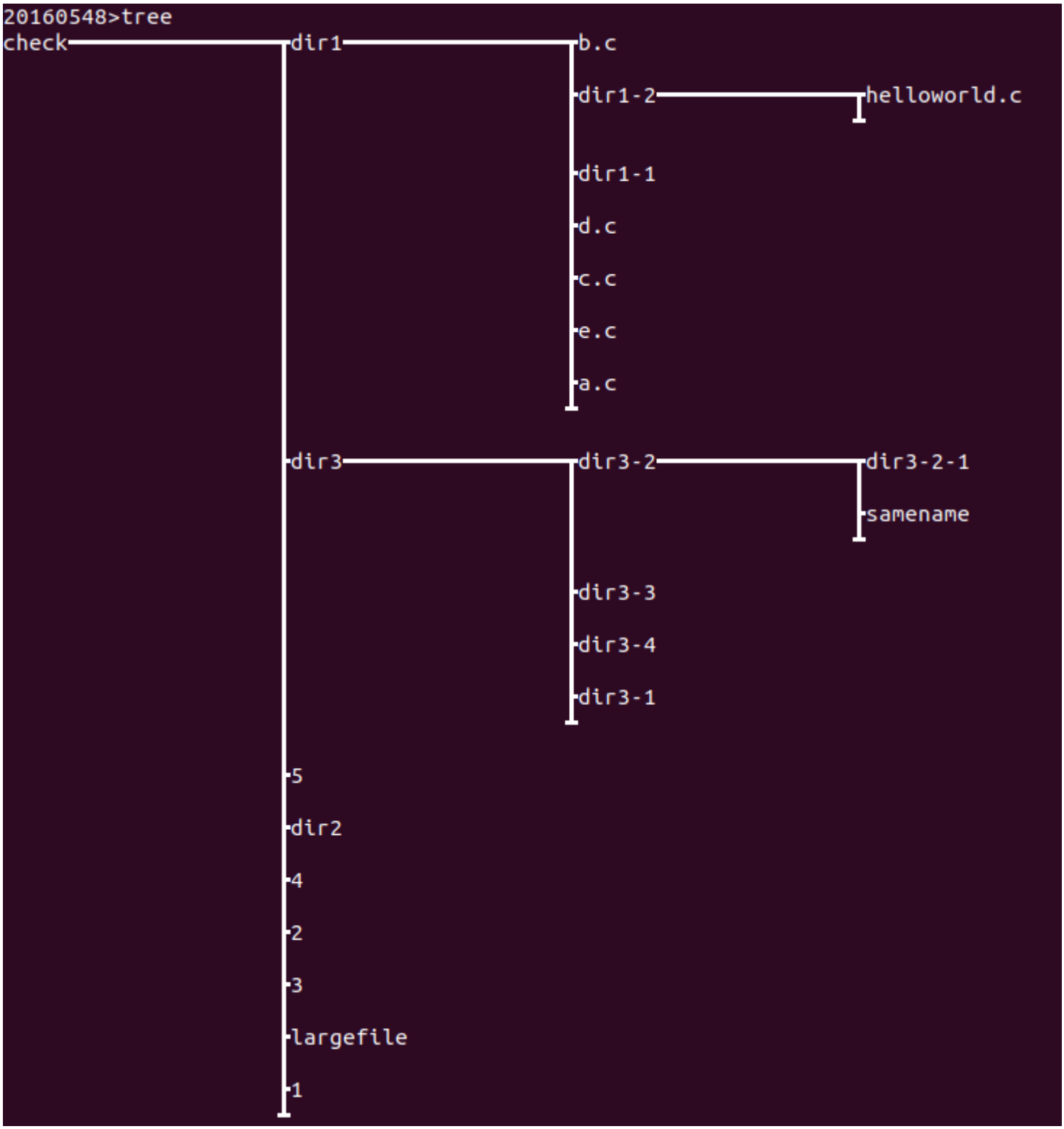
const char *getRelativePath(const char *absolutePath); // 절대경로에서 상대경로 구하는 함수

전달인자: const char *absolutePath – 상대경로를 알아내고 싶은 절대경로

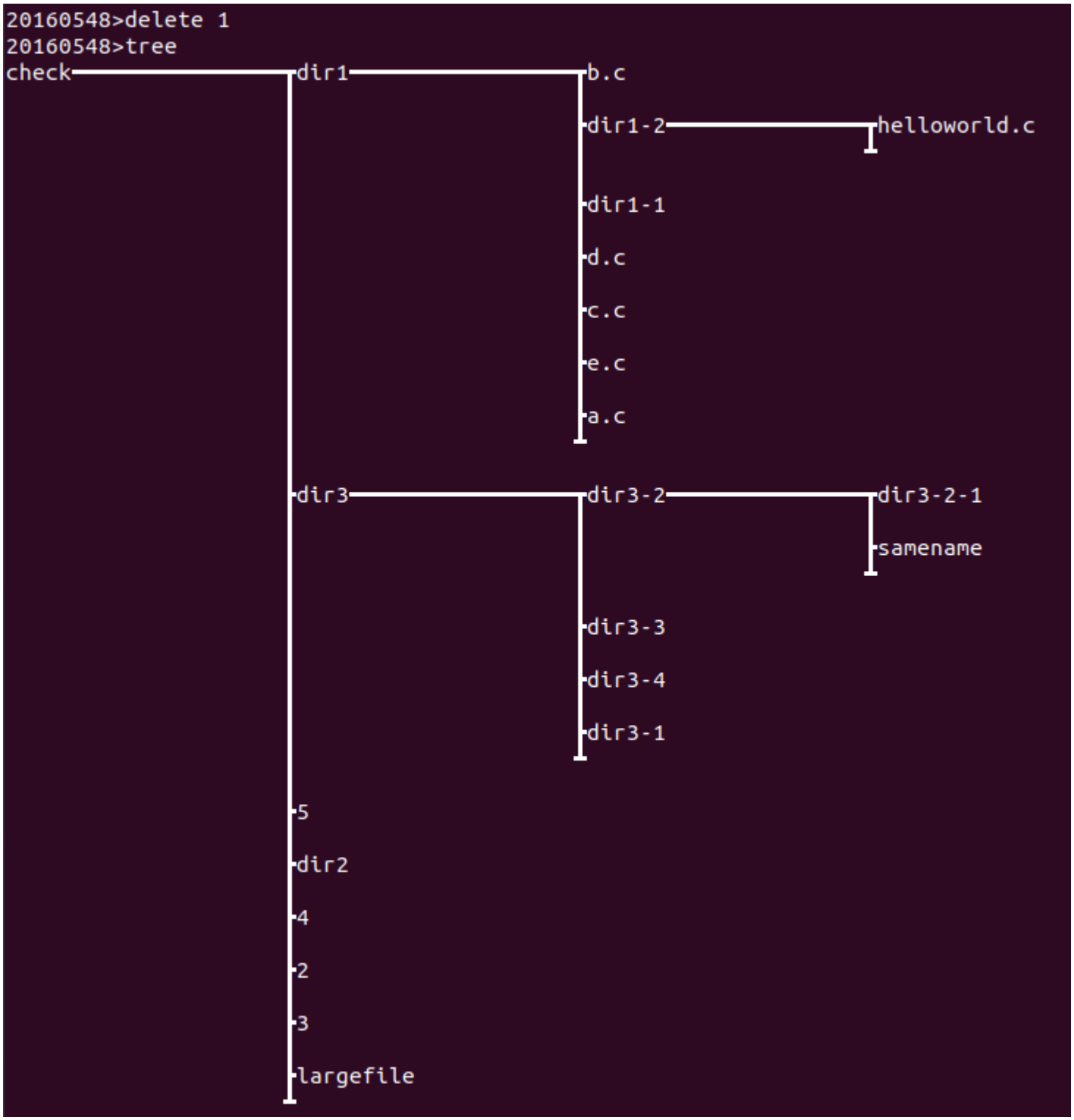
리턴 값: 절대경로 문자열 내에서 상대경로가 시작되는 위치를 리턴

4. 테스트 및 결과

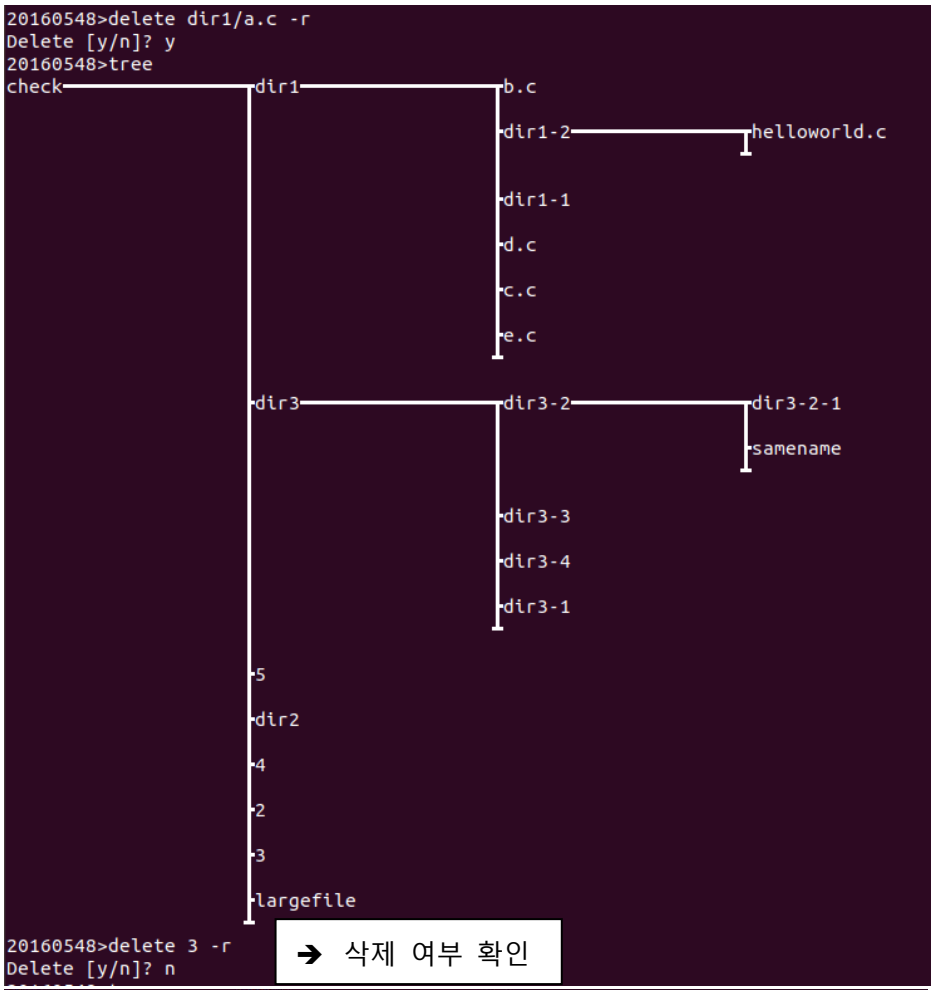
Tree 명령어

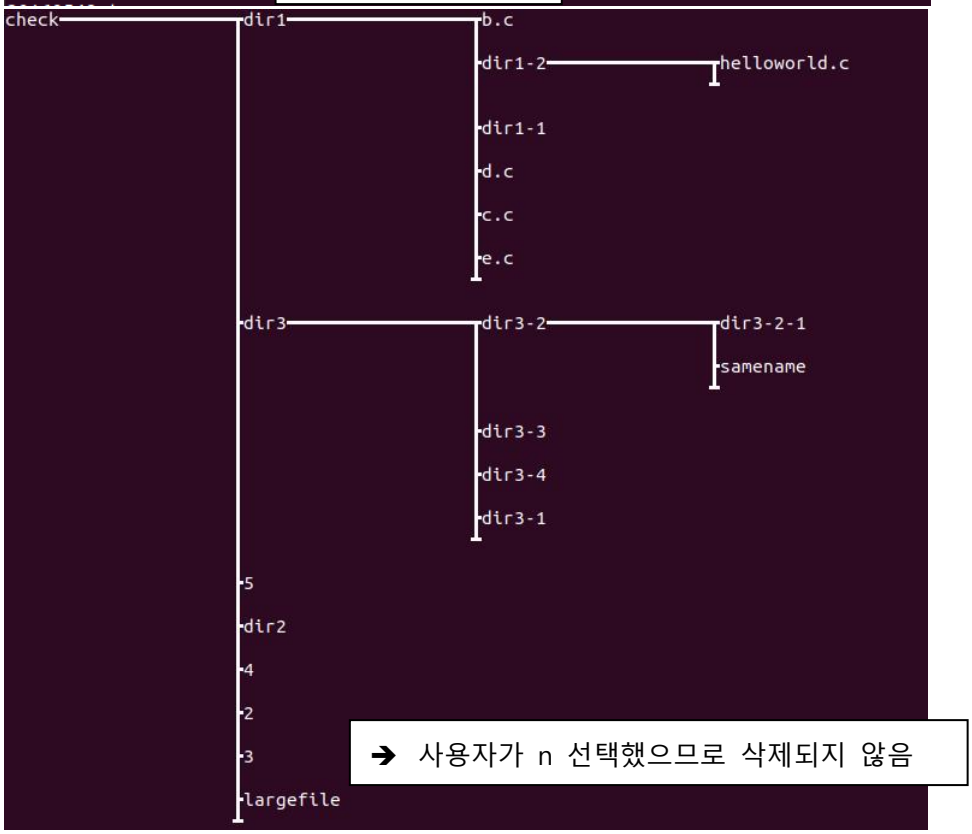


Delete 명령어

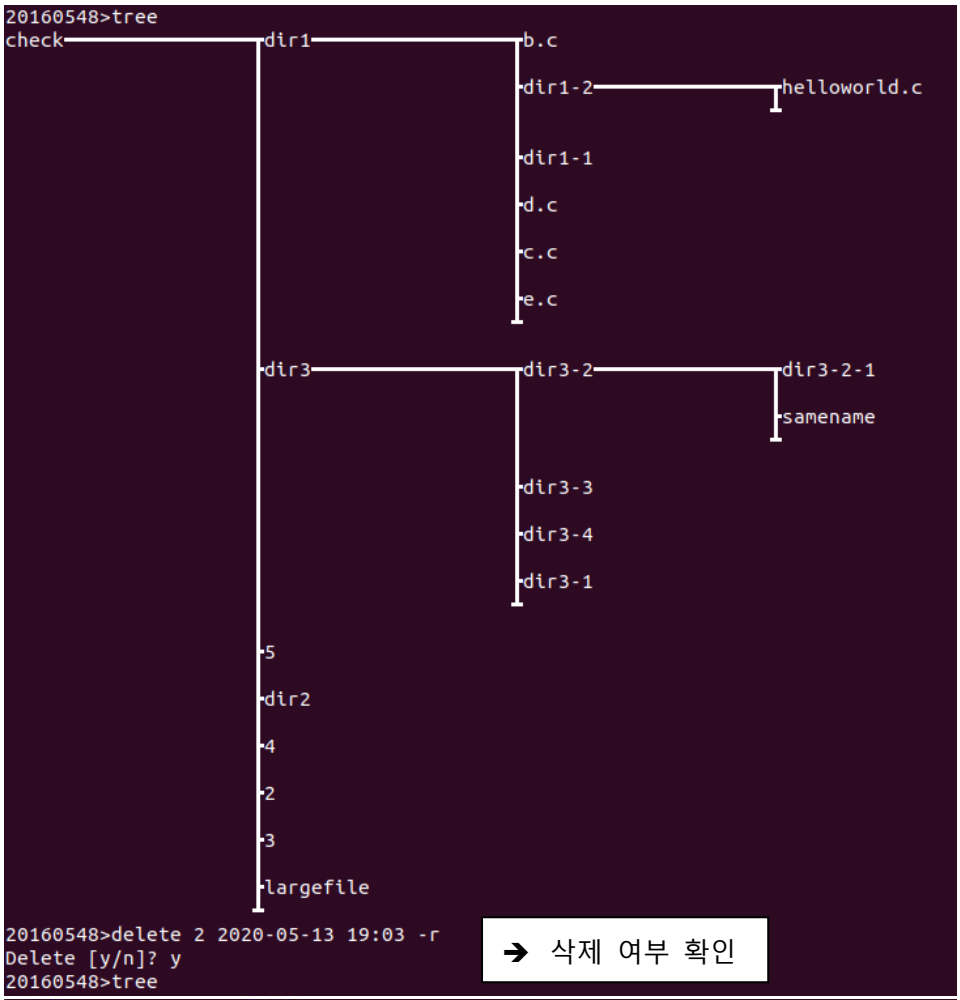

20160548>delete 1
20160548>tree
check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
a.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
2
3
largefile

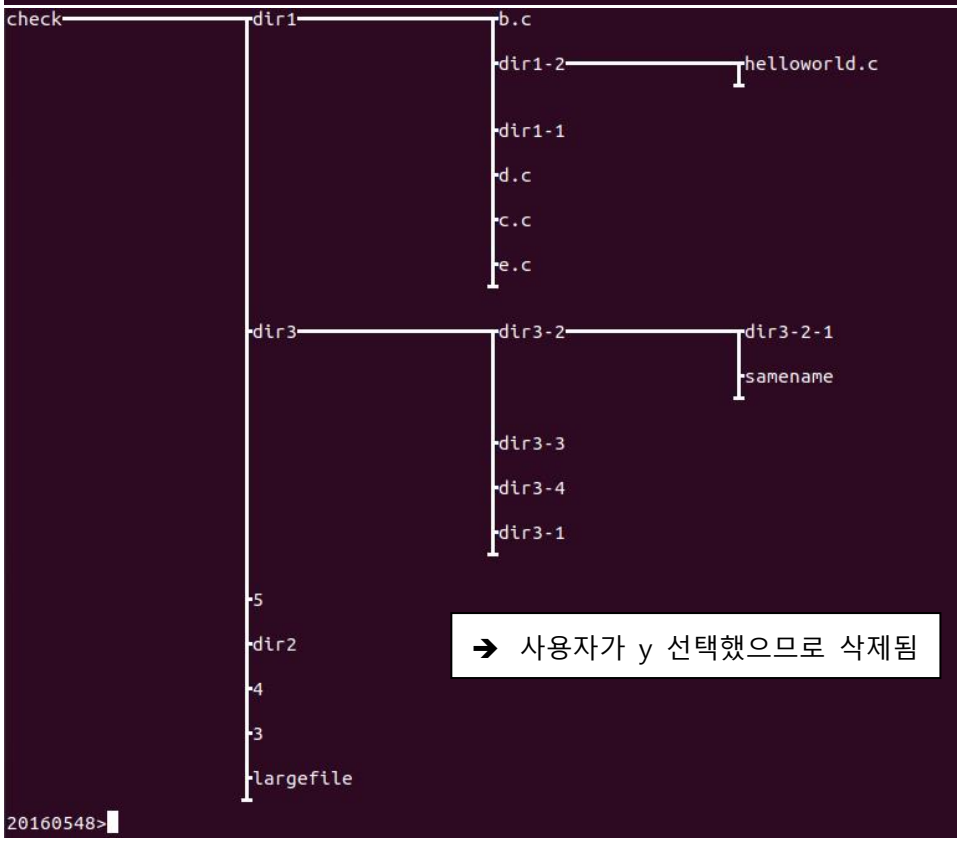
Delete -r 옵션


20160548>delete dir1/a.c -r
Delete [y/n]? y
20160548>tree
check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
2
3
largefile
20160548>delete 3 -r
Delete [y/n]? n
➔ 삭제 여부 확인

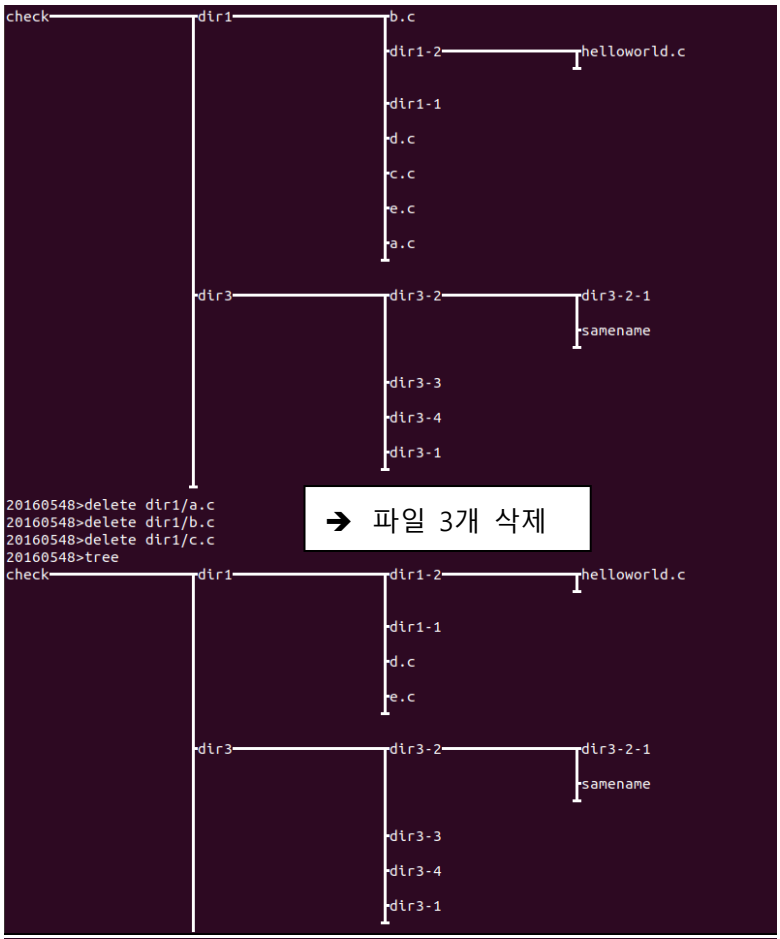

check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
2
3
largefile
➔ 사용자가 n 선택했으므로 삭제되지 않음

Delete 시간지정


20160548>tree
check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
2
3
largefile
20160548>delete 2 2020-05-13 19:03 -r
Delete [y/n]? y
20160548>tree
➔ 삭제 여부 확인


check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
3
largefile
20160548>
➔ 사용자가 y 선택했으므로 삭제됨

Delete 명령어 info 디렉토리 사이즈가 2KB보다 커졌을 때




```
20160548>delete dir1/a.c
20160548>delete dir1/b.c
20160548>delete dir1/c.c
20160548>tree
```

➔ 파일 3개 삭제


```
shlee@shlee-virtual-machine:~/workspace/lsp/project2/trash/info$ ls -al
합계 84
drwxr-xr-x 2 shlee shlee 4096  5월 12 19:35 .
drwxr-xr-x 4 shlee shlee 4096  5월 11 17:49 ..
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 1
-rw-r--r-- 1 shlee shlee  106  5월 12 19:34 1_1
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 2
-rw-r--r-- 1 shlee shlee  106  5월 12 19:34 2_1
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 3
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 4
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 5
-rw-r--r-- 1 shlee shlee  104  5월 12 19:34 6
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d1
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d2
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d4
-rw-r--r-- 1 shlee shlee  105  5월 12 19:31 d5
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d6
-rw-r--r-- 1 shlee shlee  105  5월 12 19:31 d7
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d8
-rw-r--r-- 1 shlee shlee  107  5월 12 19:35 dir2
-rw-r--r-- 1 shlee shlee  112  5월 12 19:34 largefile
-rw-r--r-- 1 shlee shlee  111  5월 12 19:34 samename
-rw-r--r-- 1 shlee shlee  116  5월 12 19:34 test.txt
```

➔ 파일 삭제 전

➔ 파일 삭제 전에는 d2, d6, d8이 있었음

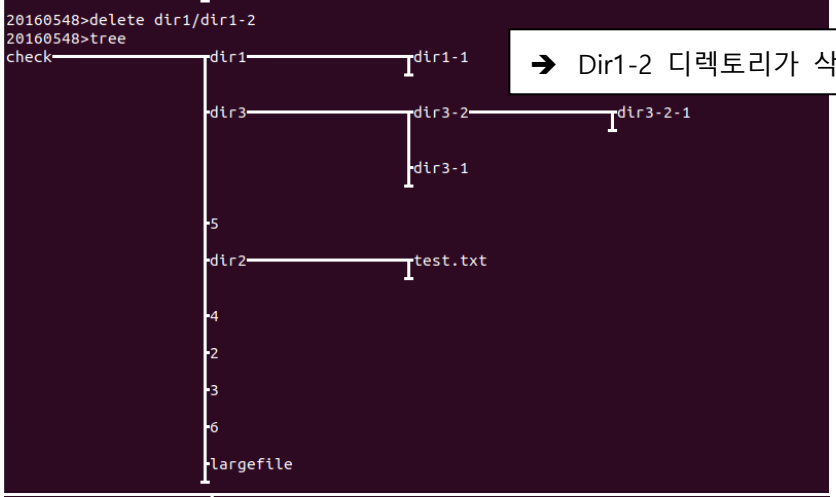
```
shlee@shlee-virtual-machine:~/workspace/lsp/project2/trash/info$ ls -al
합계 84
drwxr-xr-x 2 shlee shlee 4096  5월 12 19:36 .
drwxr-xr-x 4 shlee shlee 4096  5월 11 17:49 ..
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 1
-rw-r--r-- 1 shlee shlee  106  5월 12 19:34 1_1
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 2
-rw-r--r-- 1 shlee shlee  106  5월 12 19:34 2_1
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 3
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 4
-rw-r--r-- 1 shlee shlee  104  5월 12 19:33 5
-rw-r--r-- 1 shlee shlee  104  5월 12 19:34 6
-rw-r--r-- 1 shlee shlee  111  5월 12 19:36 a.c
-rw-r--r-- 1 shlee shlee  111  5월 12 19:36 b.c
-rw-r--r-- 1 shlee shlee  111  5월 12 19:36 c.c
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d2
-rw-r--r-- 1 shlee shlee  105  5월 12 19:30 d4
-rw-r--r-- 1 shlee shlee  105  5월 12 19:31 d5
-rw-r--r-- 1 shlee shlee  105  5월 12 19:31 d7
-rw-r--r-- 1 shlee shlee  107  5월 12 19:35 dir2
-rw-r--r-- 1 shlee shlee  112  5월 12 19:34 largefile
-rw-r--r-- 1 shlee shlee  111  5월 12 19:34 samename
-rw-r--r-- 1 shlee shlee  116  5월 12 19:34 test.txt
shlee@shlee-virtual-machine:~/workspace/lsp/project2/trash/info$
```

➔ 파일 3개 삭제 후

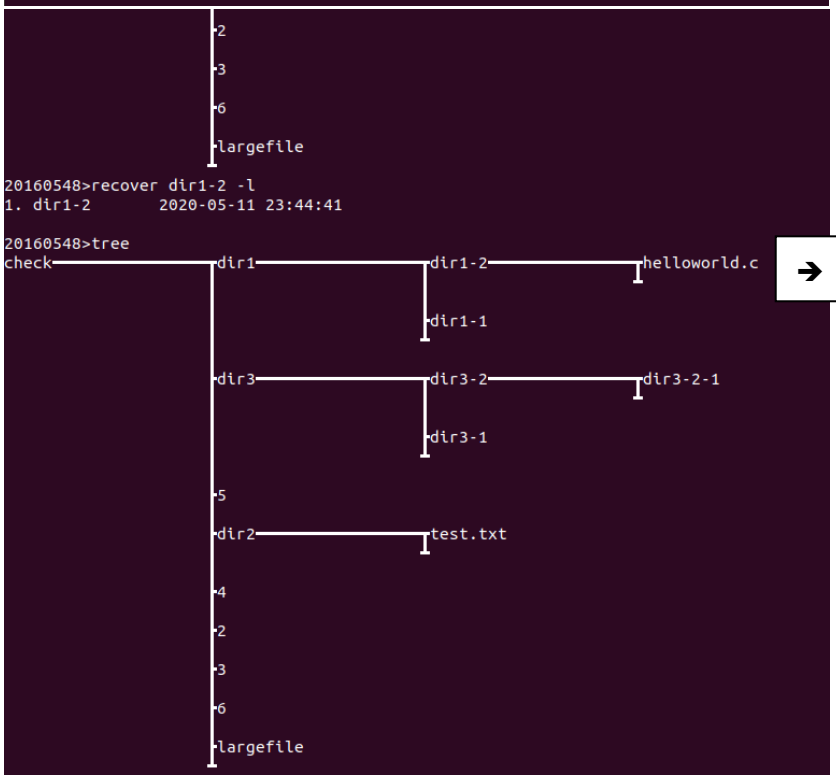
➔ 삭제 한 파일 3개에 대한 info파일 생성됨

➔ 파일 삭제 후에는 info 디렉토리의 크기가 2KB를 초과했기 때문에 d2, d6, d8이 사라짐

디렉토리 delete, recover

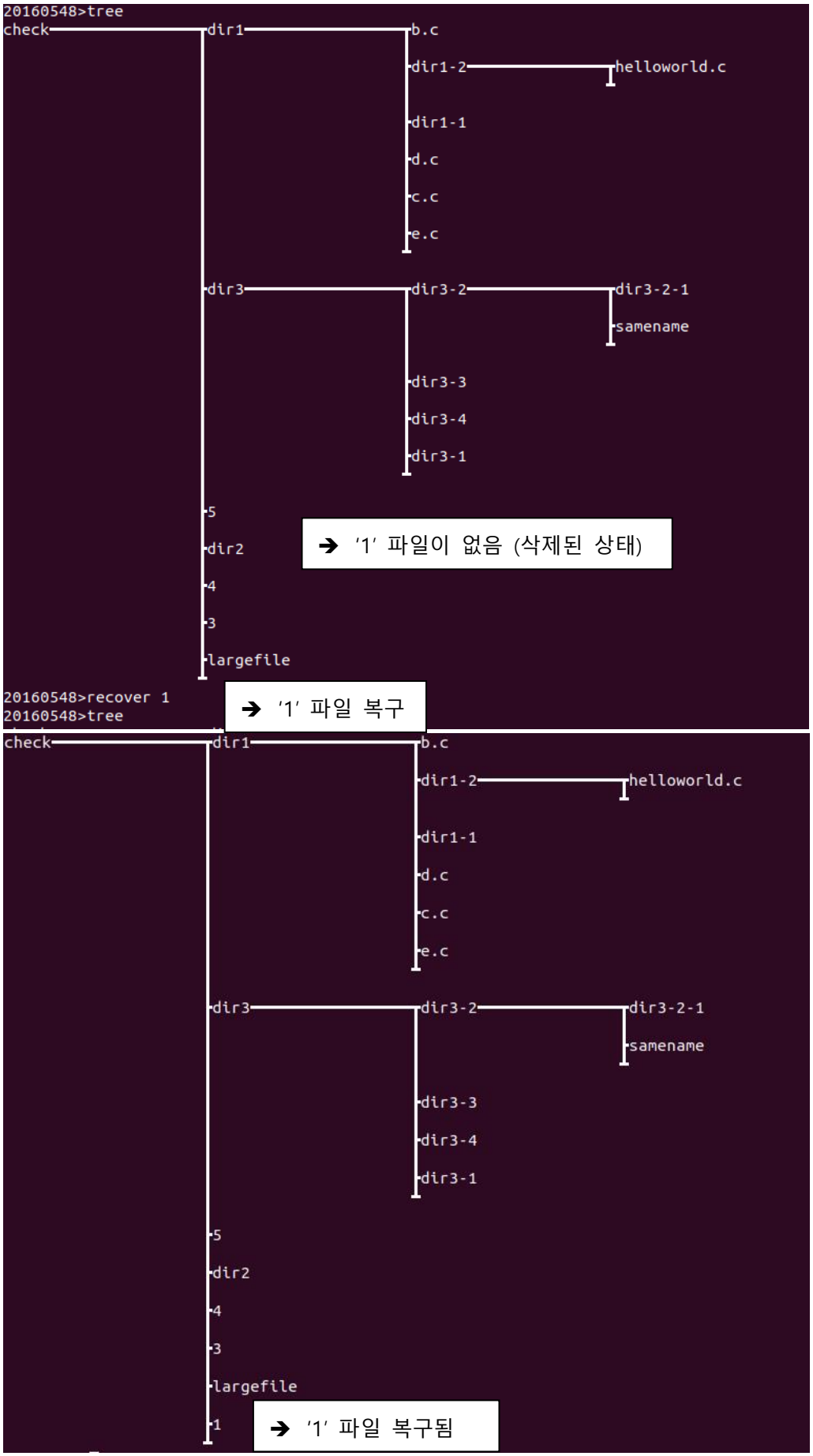


➔ Dir1-2 디렉토리가 삭제됨



➔ Dir1-2 디렉토리가 복구됨

Recover 명령어


20160548>tree
check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
3
largefile
➔ '1' 파일이 없음 (삭제된 상태)
20160548>recover 1
20160548>tree
➔ '1' 파일 복구
check
dir1
b.c
dir1-2
helloworld.c
dir1-1
d.c
c.c
e.c
dir3
dir3-2
dir3-2-1
samename
dir3-3
dir3-4
dir3-1
5
dir2
4
3
largefile
1
➔ '1' 파일 복구됨

Recover 중복된 이름

➔ 중복된 이름의 파일들 복구

➔ 이름이 같은 파일이 있으면 숫자_이름 형태로 복구됨

➔ 이름이 같은 파일이 있으면 숫자_이름 형태로 복구됨

➔ 중복된 이름의 파일

Recover -l 옵션

```
20160548>recover 2 -l
1. a.c  2020-05-13 18:54:24
2. 2    2020-05-13 19:03:11

20160548>
```

Recover 존재하지 않는 파일 에러 메시지

```
20160548>recover 23 -l
1. d5   2020-05-12 19:31:01
2. d7   2020-05-12 19:31:05
3. 6    2020-05-12 19:34:02
4. 1_1  2020-05-12 19:34:06
5. 2_1  2020-05-12 19:34:09
6. d2   2020-05-12 19:40:55
7. d4   2020-05-12 19:40:58

There is no '23' in the 'trash' directory
20160548>recover dfsfa
There is no 'dfsfa' in the 'trash' directory
20160548>recover 6 -l
1. d5   2020-05-12 19:31:01
2. d7   2020-05-12 19:31:05
3. 6    2020-05-12 19:34:02
4. 1_1  2020-05-12 19:34:06
5. 2_1  2020-05-12 19:34:09
6. d2   2020-05-12 19:40:55
7. d4   2020-05-12 19:40:58
```

Size 명령어

```
20160548>size check
20835   ./check
20160548>size check/dir1
1058    ./check/dir1
20160548>
```

Size -d 옵션

```
20160548>size check -d 4
20989    ./check
0        ./check/1
0        ./check/2
0        ./check/3
0        ./check/4
0        ./check/5
1212     ./check/dir1
154      ./check/dir1/a.c
58       ./check/dir1/b.c
960      ./check/dir1/c.c
15       ./check/dir1/d.c
0        ./check/dir1/dir1-1
0        ./check/dir1/dir1-2
0        ./check/dir1/dir1-2/helloworld.c
25       ./check/dir1/e.c
0        ./check/dir2
17       ./check/dir3
0        ./check/dir3/dir3-1
17       ./check/dir3/dir3-2
0        ./check/dir3/dir3-2/dir3-2-1
17       ./check/dir3/dir3-2/samename
0        ./check/dir3/dir3-3
0        ./check/dir3/dir3-4
19760    ./check/largefile
20160548>size check -d 2
20989    ./check
0        ./check/1
0        ./check/2
0        ./check/3
0        ./check/4
0        ./check/5
1212     ./check/dir1
0        ./check/dir2
17       ./check/dir3
19760    ./check/largefile
20160548>size check/dir1 -d 2
1212     ./check/dir1
154      ./check/dir1/a.c
58       ./check/dir1/b.c
960      ./check/dir1/c.c
15       ./check/dir1/d.c
0        ./check/dir1/dir1-1
0        ./check/dir1/dir1-2
25       ./check/dir1/e.c
20160548>
```

Help 명령어

```
20160548>help
- DELETE 지정한 삭제 시간에 자동으로 파일을 삭제해주는 명령어
        usage : DELETE [FILENAME] [END_TIME] [OPTION]
        -i              삭제 시 'trash' 디렉토리로 삭제 파일과 정보를 이동시키지 않고 파일 삭제
        -r              지정한 시간에 삭제 시 삭제 여부 재확인
- SIZE 파일경로(상대경로), 파일 크기 출력하는 명령어
        usage : SIZE [FILENAME] [OPTION]
        -d NUMBER               NUMBER 단계만큼의 하위 디렉토리까지 출력
- RECOVER "trash" 디렉토리 안에 있는 파일을 원래 경로로 복구하는 명령어
        usage : RECOVER [FILENAME] [OPTION]
        -l              'trash'디렉토리 밑에 있는 파일과 삭제 시간들을 삭제 시간이 오래된 순으로 출력 후, 명령어 진행
- TREE "check" 디렉토리의 구조를 tree 형태로 보여주는 명령어
        usage : TREE
- EXIT 프로그램 종료시키는 명령어
        usage : EXIT
- HELP 명령어 사용법을 출력하는 명령어
        usage : HELP
```

Exit 명령어

```
20160548>exit
ssu_mntr 종료...
shlee@shlee-virtual-machine:~/workspace/lsp/project2$
```

Log.txt

```
[2020-05-13 18:53:50][delete_1]
[2020-05-13 18:54:25][modify_dir1]
[2020-05-13 18:54:25][delete_a.c]
[2020-05-13 19:03:11][delete_2]
[2020-05-13 19:03:12][delete_2]
[2020-05-13 19:05:16][create_1]
[2020-05-13 19:06:11][create_2]
~
~
~
~
~
~
~
```

5. 소스코드

<ssu_mntr.c>

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

#define BUFFER_SIZE 1024

void startMntr(void); // monitor.c의 함수



struct DeleteFileInfo{ // 삭제할 파일의 정보를 담는 구조체
        char *absolutePathName; // 삭제할 파일 절대경로
        int checkBeforeDelete; // -r 옵션이 지정되어 삭제 이전에 사용자에게 확인을 해야 하는지를 저장하는 flag
        int deleteImmediately; // -i 옵션이 지정되어 바로 삭제해야하는지를 저장하는 flag
        time_t deleteTime; // 사용자가 지정한 삭제 시간
};
```

```
struct DeleteFileHeap{ // 삭제할 파일 목록을 저장하는 heap. 삭제할 시간 오름차순으로 정렬
	struct DeleteFileInfo *deleteFiles[BUFFER_SIZE];
	int size; // heap의 원소의 개수
};

struct infoFile{ // recover 명령어를 수행할 때 info디렉토리에 있는 파일의 정보를 읽어들여 저장할 때 사용하는 구조체
	char filename[MAXNAMLEN]; // 파일명
	char deleteTime[100]; // 삭제시간 ("'년-월-일 시간-분-초" 형식)
};

struct FileSize{ // 파일명과 파일 크기를 저장하는 구조체
	char pathname[512]; // 파일명
	off_t filesize; // 파일 크기
};

struct FileSizes{ // 파일 파일명과 파일크기를 저장하는 구조체
	struct FileSize fileSize[512];
	int size; // 위의 배열에 들어있는 유효한 원소의 개수
};

void doDeleteCommand(); // delete 명령어를 수행하는 함수
void doExitCommand(); // exit 명령어를 수행하는 함수
void doHelpCommand(); // help 명령어를 수행하는 함수
void doTreeCommand(); // tree 명령어를 수행하는 함수
void doRecoverCommand(); // recover 명령어를 수행하는 함수
void doSizeCommand(); // size 명령어를 수행하는 함수
void printTree(const char *dirname, int depth); // tree 명령어에서 tree를 출력하는 재귀호출 함수
```

```
void pushToHeap(struct DeleteFileInfo* newElement); // DeleteFileHeap에 새로운 원소를 추가하는 함수

struct DeleteFileInfo *popFromHeap(); // DeleteFileHeap에서 원소를 하나 빼오는 함수

void alarmHandlercheckFileDeleteTime(); // SIGALRM이 전달됐을 때 삭제할 파일이 있는지 확인하는 함수

void checkFileDeleteTime(); // 삭제할 파일이 있는지 확인하는 함수

void deleteFile(struct DeleteFileInfo *deleteFileInfo); // 파일을 삭제하는 함수

void checkAndEraseTrashFiles(); // trash/info 디렉토리의 크기가 2KB를 넘어가면 오래된 파일부터 삭제하는 함수

int compareInfoFile(const void *a, const void *b); // infoFile구조체 배열을 qsort 함수로 sort할 때 사용하는 비교 함수

off_t getDirSize(const char *dirname); // 해당 디렉토리의 크기를 구하는 함수

void getPrintingDirs(const char *dirname, const char *dirpath, int depth, int maxDepth); // size 명령어에 의해 출력될 파일들을 확인해 fileSizes 구조체에 추가하는 함수

int compareFileSize(const void *a, const void *b); // FileSize 구조체 배열을 qsort 함수로 sort할 때 사용하는 비교 함수


struct FileSizes fileSizes; // size 명령어 수행에 사용할 구조체

struct DeleteFileHeap deleteFileHeap; // 파일 삭제에 사용할 heap

char exec_path[PATH_MAX]; // 프로세스 실행 위치

int deleteUsingRFlag = 0; // 예약된 시간에 -r 옵션에 의해서 삭제된 파일이 있는지 확인하는데 사용할 flag


int main(void)

{

        char command_buf[BUFFER_SIZE]; // 프롬프트 창에서 입력될 명령어를 저장할 배열

        char *command; // 명령어를 가리킬 포인터

        sigset_t sig_set; // SIGALRM을 블록하는데 사용할 시그널 집합


        if (access("check", F_OK) != 0) { // 이 프로세스가 감시할 check 디렉토리가 존재하는지 확인

                mkdir("check", 0777); // 존재하지 않는다면 새로 생성한다

        }
```

```
sigemptyset(&sig_set);
sigaddset(&sig_set, SIGALRM); // 시그널 집합에 SIGALRM을 추가한다

getcwd(exec_path, PATH_MAX); // 프로세스 실행 위치를 저장한다
startMntr(); // 타겟 디렉토리를 감시하는 디몬 프로세스를 실행한다
setbuf(stdout, NULL); // 프롬프트 메세지 출력을 위해 표준출력을 non buffer로 바꾼다

signal(SIGALRM, alarmHandlercheckFileDeleteTime); // 삭제할 파일이 있는지 지속적인 체크를 위해 SIGALARM에 핸들러를 등록한다
alarm(60); // 60초마다 SIGALRM을 발생시켜 삭제할 파일이 있는지 확인한다

while(1) {
	printf("20160548>");
	fgets(command_buf, sizeof(command_buf), stdin); // 명령어를 입력받는다
	sigprocmask(SIG_BLOCK, &sig_set, NULL); // 명령어 수행 도중에는 SIGALRM이 전달돼도 핸들러가 실행되지 않도록 블록한다
	if(command_buf[0] == '₩n') {
		sigprocmask(SIG_UNBLOCK, &sig_set, NULL);
		continue;
	}
	command_buf[strlen(command_buf) - 1] = '₩0'; // fgets로 읽어온 문자열 맨 끝의 개행을 지운다
	command = strtok(command_buf, " "); // 명령어 부분만 분리

	if (!strcmp(command, "DELETE") || !strcmp(command, "delete")) {
		doDeleteCommand(); // delete 명령어 수행
	} else if (!strcmp(command, "EXIT") || !strcmp(command, "exit")) {
		doExitCommand(); // exit 명령어 수행
		break; // 반복(프롬프트) 종료
```

```
		} else if (!strcmp(command, "TREE") || !strcmp(command, "tree")) {
			doTreeCommand(); // tree 명령어 수행
		} else if (!strcmp(command, "RECOVER") || !strcmp(command, "recover")) {
			doRecoverCommand(); // recover 명령어 수행
		} else if (!strcmp(command, "SIZE") || !strcmp(command, "size")) {
			doSizeCommand(); // size 명령어 수행
		} else {
			doHelpCommand(); // help 명령어 수행
		}
		chdir(exec_path); // 처음 프로세스가 실행된 경로로 돌아간다
		checkFileDeleteTime(); // 삭제할 파일이 있는지 확인한다
		deleteUsingRFlag = 0;
		sigprocmask(SIG_UNBLOCK, &sig_set, NULL); // 명령어 수행이 끝났으므로 SIGALRM을 언블록한다
	}

	exit(0);
}

void doDeleteCommand(){ // 삭제 명령어를 수행하는 함수
	struct DeleteFileInfo *newDeleteFileInfo; // 삭제할 파일의 정보를 저장하는 함수
	char *filename; // 파일명
	char *absolutePathName; // 파일의 절대경로
	char *nextToken;
	const char *filesDirPath = "trash/files/";
	const char *infoDirPath = "trash/info/";
	char *END_DATE = NULL; // 삭제할 날짜 문자열
	char *END_TIME = NULL; // 삭제할 시간 문자열
```

```
int i_option = 0; // -i옵션으로 실행됐는지
int r_option = 0; // -r옵션으로 실행됐는지
int i;
time_t deleteTime = 0; // 삭제할 시간

nextToken = strtok(NULL, " "); // 명령어에서 다음 토큰 가져온다
filename = nextToken;

nextToken = strtok(NULL, " "); // 명령어에서 다음 토큰 가져온다
if (nextToken == NULL) { // 삭제 시간 주어지지 않고 바로 삭제하는 경우
        ;
}else if (nextToken[0] == '-') { // END_TIME 주어지지 않고 옵션이 전달된 경우
        // 옵션 확인
        if (nextToken[1] == 'i') i_option = 1;
        else if (nextToken[1] == 'r') r_option = 1;
        else;

        // 다음 옵션 확인
        nextToken = strtok(NULL, " ");
        if (nextToken == NULL) {
                ;
        }else if (nextToken[0] == '-' && nextToken[1] == 'i') {
                i_option = 1;
        } else if (nextToken[0] == '-' && nextToken[1] == 'r') {
                r_option = 1;
        } else {
                return;
```

```
		}
	} else if ('0' <= nextToken[0] && nextToken[0] <= '9') { // END_TIME 옵션 전달된 경우
		END_DATE = nextToken;
		nextToken = strtok(NULL, " ");
		END_TIME = nextToken;
		if (END_TIME == NULL) return;
		// 옵션들 확인
		for (i = 0; i < 2; ++i) {
			nextToken = strtok(NULL, " ");
			if (nextToken == NULL) {
				break;
			}else if (nextToken[0] == '-' && nextToken[1] == 'i') {
				i_option = 1;
			} else if (nextToken[0] == '-' && nextToken[1] == 'r') {
				r_option = 1;
			} else {
				return;
			}

		}
	} else { // 잘못된 옵션 전달
		return;
	}

	//printf("i_option : %d   r_option : %d   END_TIME : %s₩n", i_option, r_option, END_TIME);
```

```
if (chdir("trash") < 0) { // trash 디렉토리로 이동
        if (mkdir("trash", 0777) < 0) { // trash 디렉토리가 없다면 새로 생성
                fprintf(stderr, "mkdir error₩n");
        }
        chdir("trash");
}

if (chdir("files") < 0) { // files 디렉토리가 없다면 새로 생성
        if (mkdir("files", 0777) < 0) {
                fprintf(stderr, "mkdir error₩n");
        }
        chdir("files");
}
chdir("..");

if (chdir("info") < 0) { // info 디렉토리가 없다면 새로 생성
        if (mkdir("info", 0777) < 0) {
                fprintf(stderr, "mkdir error₩n");
        }
        chdir("info");
}
chdir(".."); // 원래 위치로 복귀
chdir("..");
chdir("check"); // check 디렉토리로 이동

if (END_DATE != NULL && END_TIME != NULL) { // 삭제 예정 시간이 주어졌다면
        struct tm deletetm;
```

```
		char *dateToken;
		char *timeToken;

		// 삭제 시간을 time_t로 변환
		deletetm.tm_year = atoi(strtok(END_DATE, "-")) - 1900;
		deletetm.tm_mon = atoi(strtok(NULL, "-")) - 1;
		deletetm.tm_mday = atoi(strtok(NULL, "-"));

		deletetm.tm_hour = atoi(strtok(END_TIME, ":"));
		deletetm.tm_min = atoi(strtok(NULL, ":"));
		deletetm.tm_sec = 0;

		deleteTime = mktime(&deletetm); // 삭제 시간을 time_t로 변환해 저장
		//printf("%d-%d-%d %d:%d₩n", deletetm.tm_year, deletetm.tm_mon, deletetm.tm_mday, deletetm.tm_hour,
deletetm.tm_min);
	}

	newDeleteFileInfo = (struct DeleteFileInfo *) malloc(sizeof(newDeleteFileInfo));
	newDeleteFileInfo->absolutePathName = realpath(filename, NULL); // 삭제할 파일의 절대 경로를 저장
	if (i_option) { // i옵션으로 실행됐다면
		newDeleteFileInfo->deleteImmediately = 1;
	} else {
		newDeleteFileInfo->deleteImmediately = 0;
	}
	if (r_option) { // r옵션으로 실행됐다면
		newDeleteFileInfo->checkBeforeDelete = 1;
	} else {
```

```
            newDeleteFileInfo->checkBeforeDelete = 0;
        }
        newDeleteFileInfo->deleteTime = deleteTime; // 삭제할 시간 저장

        // 우선순위 큐에 넣기
        pushToHeap(newDeleteFileInfo);

        //printf("deleteTime : %ld, curTime : %ld, i_option : %d, r_option : %d, absolutePathName = %s₩n", newDeleteFileInfo->deleteTime, time(NULL), newDeleteFileInfo->deleteImmediately, newDeleteFileInfo->checkBeforeDelete, newDeleteFileInfo->absolutePathName);

        chdir("..");

        return;
}

void doExitCommand(){ // exit 명령어 수행하는 함수
        printf("ssu_mntr 종료...₩n"); // 종료 메세지 출력
}

void doHelpCommand(){ // help 명령어 수행하는 함수
        printf("- DELETE 지정한 삭제 시간에 자동으로 파일을 삭제해주는 명령어₩n");
        printf("₩tusage : DELETE [FILENAME] [END_TIME] [OPTION]₩n");
        printf("₩t-i₩t₩t삭제 시 'trash' 디렉토리로 삭제 파일과 정보를 이동시키지 않고 파일 삭제₩n");
        printf("₩t-r₩t₩t지정한 시간에 삭제 시 삭제 여부 재확인₩n");
        printf("- SIZE 파일경로(상대경로), 파일 크기 출력하는 명령어₩n");
        printf("₩tusage : SIZE [FILENAME] [OPTION]₩n");
        printf("₩t-d NUMBER₩t₩tNUMBER 단계만큼의 하위 디렉토리까지 출력₩n");
```

```
	printf("- RECOVER ₩"trash₩" 디렉토리 안에 있는 파일을 원래 경로로 복구하는 명령어₩n");
	printf("₩tusage : RECOVER [FILENAME] [OPTION]₩n");
	printf("₩t-l₩t₩t'trash'디렉토리 밑에 있는 파일과 삭제 시간들을 삭제 시간이 오래된 순으로 출력 후, 명령어 진행
₩n");
	printf("- TREE ₩"check₩" 디렉토리의 구조를 tree 형태로 보여주는 명령어₩n");
	printf("₩tusage : TREE₩n");
	printf("- EXIT 프로그램 종료시키는 명령어₩n");
	printf("₩tusage : EXIT₩n");
	printf("- HELP 명령어 사용법을 출력하는 명령어₩n");
	printf("₩tusage : HELP₩n");

}

void doTreeCommand(){ // tree 명령어 수행하는 함수
	int i;
	int cnt;
	struct dirent **nextdirlist;
	printf("check");
	if ((cnt = scandir("check", &nextdirlist, NULL, NULL)) >= 2) { // 해당 디렉토리에 파일이 들어있다면
		for(i = 0; i < 21 - strlen("check"); ++i) { // 트리의 가지 그림 그린다
			printf("—");
		}
		printTree("check", 1); // 재귀하며 모든 디렉토리 출력
	}
	printf("₩n");
	chdir("..");
}
```

```
void printTree(const char *dirname, int depth){ // tree 명령어에서 tree 출력하는 함수
        // | └┬├─ - 트리 가지 그릴 때 사용할 문자들
        int i;
        int firstFileFlag = 1; // 해당 디렉토리 내에서 첫번째로 출력하는 파일인지 확인하는 플래그
        char filename[MAXNAMLEN]; // 파일명 저장할 배열
        struct dirent *dentry;
        struct stat statbuf;
        struct dirent **nextdirlist;
        DIR *dir = opendir(dirname);
        if (dir == NULL) {
                fprintf(stderr, "opendir error, %s₩n", strerror(errno));
                exit(1);
        }
        chdir(dirname); // 해당 디렉토리로 이동


        while((dentry = readdir(dir)) != NULL) {
                if (dentry->d_ino == 0) continue;

                memcpy(filename, dentry->d_name, MAXNAMLEN); // 출력하면 안되는 것들 pass
                if (!strcmp(filename, ".") || !strcmp(filename, "..")) continue;


                if (stat(filename, &statbuf) == -1) { // 해당 파일의 stat 구조체 가져옴
                        fprintf(stderr, "stat error for %s₩n", filename);
                        printf("ERROR:%s₩n", strerror(errno));
```

```
            exit(1);
}

// 트리의 가지와 파일명 출력
if (!firstFileFlag) { // 첫번째 파일이 아니라면
            for (i = 0; i < depth; ++i) {
                        printf("%24s", " ₩ ");
            }
            printf("₩n");
            for (i = 0; i < depth; ++i) {
                        if (i + 1 == depth)
                                    printf("%24s", " ├");
                        else
                                    printf("%24s", " ₩ ");
            }
            printf("%s", filename);

} else { // 첫번째 파일이라면
            printf("┬%s", filename);
            firstFileFlag = 0; // 플래그 0으로 바꿈

}

if (S_ISDIR(statbuf.st_mode)) { // 디렉토리라면
            int cnt;
            if ((cnt = scandir(filename, &nextdirlist, NULL, NULL)) > 2) {
                        for(i = 0; i < 21 - strlen(filename); ++i) {
```

```
                        printf("—");
                    }
                    // 재귀호출
                    printTree(filename, depth + 1);
                    // 작업 후 원래 디렉토리로 복귀하기
                    chdir("..");

                }

                if(cnt != -1)
                    free(nextdirlist);
            }
            printf("₩n");

}

// 트리의 가지 끝부분 출력
for (i = 0; i < depth; ++i) {
            if (i + 1 == depth)
                printf("%24s", "ㄴ");
            else
                printf("%24s", " | ");
}

if (closedir(dir) < 0) {
            fprintf(stderr, "closedir error₩n");
            exit(1);
```

```
	}
}

void pushToHeap(struct DeleteFileInfo* newElement){ // DeleteFileInfo heap에 새로운 원소 추가하는 함수
	int i;

	deleteFileHeap.deleteFiles[deleteFileHeap.size] = newElement; // heap의 맨 뒤에 새로운 파일 추가
	i = deleteFileHeap.size;

	while(deleteFileHeap.size != 0 && i >= 0) { // 원소 크기 비교해가며 새로운 원소의 위치를 찾아간다
		if (deleteFileHeap.deleteFiles[i]->deleteTime < deleteFileHeap.deleteFiles[(i - 1) / 2]->deleteTime) {
			//printf("swap₩n");
			struct DeleteFileInfo *tmp = deleteFileHeap.deleteFiles[i];
			deleteFileHeap.deleteFiles[i] = deleteFileHeap.deleteFiles[(i - 1) / 2];
			deleteFileHeap.deleteFiles[(i - 1) / 2] = tmp;
		} else {
			break;
		}
		i = (i - 1) / 2;
	}
	++(deleteFileHeap.size); // heap size 늘린다
}

struct DeleteFileInfo *popFromHeap(){ // DeleteFileInfo heap에서 원소를 하나 빼내는 함수
	int i;
	struct DeleteFileInfo *popedData = deleteFileHeap.deleteFiles[0]; // 빼낼 데이터의 주소 저장
	deleteFileHeap.deleteFiles[0] = deleteFileHeap.deleteFiles[--(deleteFileHeap.size)]; // heap 맨 뒤의 원소를 맨 앞으
```

로 옮긴다

```
        deleteFileHeap.deleteFiles[deleteFileHeap.size] = NULL; // 맨 뒤의 원소가 있던 자리에 NULL 넣는다


        // heap을 다시 정렬한다
        i = 0;
        while (i * 2 + 1 < deleteFileHeap.size) {
                int nextIndex = 0;
                if (i * 2 + 2 < deleteFileHeap.size){
                        nextIndex = deleteFileHeap.deleteFiles[2 * i + 1]->deleteTime < deleteFileHeap.deleteFiles[2 * i + 2]->deleteTime ? 2 * i + 1 : 2 * i + 2;
                } else {
                        nextIndex = 2 * i + 1;
                }

                if (deleteFileHeap.deleteFiles[i]->deleteTime > deleteFileHeap.deleteFiles[nextIndex]->deleteTime) {
                        struct DeleteFileInfo *tmp = deleteFileHeap.deleteFiles[i];
                        deleteFileHeap.deleteFiles[i] = deleteFileHeap.deleteFiles[nextIndex];
                        deleteFileHeap.deleteFiles[nextIndex] = tmp;
                } else {
                        break;
                }

                i = nextIndex;
        }

        return popedData; // 빼낸 원소의 주소 리턴
}
```

```
void checkFileDeleteTime(){ // 삭제할 파일이 있는지 확인하는 함수
	time_t tmptime;
	time_t currentTime;
	char tmpPath[PATH_MAX];
	getcwd(tmpPath, PATH_MAX); // 다른 작업 도중에 SIGALRM 전달됐을 때 실행될 수 있는 함수이기 때문에 수행 후에 원위치로 돌아가기 위해 현재 프로세스의 위치를 저장해 놓는다
	chdir(exec_path); // 처음 프로세스가 실행됐던 위치로 이동

	tmptime = time(NULL);
	currentTime = mktime(localtime(&tmptime)); // 현재시간 저장
	//printf("curtime : %ld₩n", currentTime);
	// 삭제해야 할 파일들을 모두 삭제할 때까지 반복
	while(1) {
		// heap의 맨 앞의 원소의 삭제시간 < 현재시간 이라면 해당 원소를 삭제해야 한다
		if (deleteFileHeap.size != 0 && deleteFileHeap.deleteFiles[0]->deleteTime < currentTime) {
			//printf("delete time : %ld, current time : %ld₩n", deleteFileHeap.deleteFiles[0]->deleteTime, currentTime);
			deleteFile(popFromHeap()); // heap에서 원소 빼내서 해당 원소가 가리키는 파일을 삭제한다
		} else {
			break;
		}
	}
	alarm(60); // 60초 후에 다시 SIGALRM을 전달한다

	chdir(tmpPath); // 이 함수가 호출되기 이전에 프로세스가 있던 위치로 돌아간다
}
```

```
void deleteFile(struct DeleteFileInfo *deleteFileInfo){ // 파일 삭제하는 함수
	if (access(deleteFileInfo->absolutePathName, F_OK) != 0) { // 삭제할 파일에 접근할 수 없다면 바로 함수 종료
		return;
	}


	if(deleteFileInfo->checkBeforeDelete) { // r옵션 수행
		char yesOrNo;
		printf("Delete [y/n]? "); // 삭제할지 물어보는 메세지 출력
		scanf("%c", &yesOrNo);
		while(getchar()!='₩n');
		deleteUsingRFlag = 1;

		// 사용자의 대답이 y가 아니라면 함수를 종료한다
		if (yesOrNo != 'y') return;
	}

	if(deleteFileInfo->deleteImmediately) { // i옵션 수행
		if (remove(deleteFileInfo->absolutePathName) < 0) { // 해당 파일을 'trash'로 옮기지 않고 바로 삭제한다
			fprintf(stderr, "remove error for %s₩n", deleteFileInfo->absolutePathName);
			fprintf(stderr, "%s₩n", strerror(errno));
		}
	} else { // i옵션 아닐 때 (trash 디렉토리 이용)
		struct stat statbuf;
		struct tm *curtm; // 현재시간
		struct tm *modtm; // st_mtime
		time_t curtime;
```

```
char trashFileName[MAXNAMLEN]; // trash 로 이동될 파일 이름
char trashFilePath[PATH_MAX]; // trash 로 이동될 경로 이름
char infoFileName[MAXNAMLEN]; // info 파일의 이름
char curTimeStr[BUFFER_SIZE]; // 문자열로 변환한 현재시간
char modTimeStr[BUFFER_SIZE]; // 문자열로 변환한 st_mtime
char *relativePathName; // 상대경로 저장할 변수
int i;
FILE *fp;

// 삭제할 파일의 상대 경로명 찾아서 relativePathName이 가리키도록 한다
for(i = strlen(deleteFileInfo->absolutePathName) - 1; i >= 0; --i) {
        if (deleteFileInfo->absolutePathName[i] == '/') {
                relativePathName = deleteFileInfo->absolutePathName + i + 1;
                break;
        }
}

if (stat(deleteFileInfo->absolutePathName, &statbuf) < 0) { // 삭제할 파일의 stat 구조체 가져온다
        fprintf(stderr, "stat error for %s₩n", deleteFileInfo->absolutePathName);
        return;
}

// 현재 시간 구해서 문자열로 변환
curtime = time(NULL);
curtm = localtime(&curtime);
sprintf(curTimeStr, "%d-%02d-%02d %02d:%02d:%02d", curtm->tm_year + 1900, curtm->tm_mon + 1,
curtm->tm_mday, curtm->tm_hour, curtm->tm_min, curtm->tm_sec);
```

```
		// 삭제할 파일의 st_mtime 구해서 문자열로 변환
		modtm = localtime(&(statbuf.st_mtime));
		sprintf(modTimeStr, "%d-%02d-%02d %02d:%02d:%02d", modtm->tm_year + 1900, modtm->tm_mon + 1, modtm->tm_mday, modtm->tm_hour, modtm->tm_min, modtm->tm_sec);

		// info, trash 파일의 파일명, 경로 만든다
		sprintf(trashFileName, "%s_%s", curTimeStr, relativePathName);
		sprintf(trashFilePath, "trash/files/%s", trashFileName);
		rename(deleteFileInfo->absolutePathName, trashFilePath);
		sprintf(infoFileName, "trash/info/%s", relativePathName);
		if (access(infoFileName, F_OK) == 0) { // info파일이 중복된다면 파일명 뒤에 (숫자)를 붙여 덮어쓰기되지 않도록 한다
			i = 1;
			do{
				sprintf(infoFileName, "trash/info/%s(%d)", relativePathName, i);
				++i;
			} while (access(infoFileName, F_OK) == 0);
		}

		if ((fp = fopen(infoFileName, "w")) < 0) {
			fprintf(stderr, "fopen error for %s₩n", infoFileName);
			return;
		}

		// info 파일의 내용을 쓴다
		fprintf(fp, "[Trash info]₩n");
		fprintf(fp, "%s₩n", deleteFileInfo->absolutePathName);
```

```
            fprintf(fp, "D : %s₩n", curTimeStr);
            fprintf(fp, "M : %s₩n", modTimeStr);

            fclose(fp);

            //info 디렉토리의 크기가 2KB가 넘으면 오래된 파일부터 삭제
            checkAndEraseTrashFiles();
        }

        free(deleteFileInfo);
}

void checkAndEraseTrashFiles(){ // info 디렉토리의 크기가 2KB를 넘어가면 오래된 파일부터 삭제하는 함수
        struct stat statbuf;
        struct dirent **namelist;
        size_t infoFilesSize;
        int cnt;
        int i, j;
        chdir("trash/info"); // info 디렉토리로 이동
        cnt = scandir(".", &namelist, NULL, NULL); // 디렉토리에 있는 파일 목록을 구한다
        // info 디렉토리의 크기를 구한다
        infoFilesSize = 0;
        for (i = 0; i < cnt; ++i) {
            if (!strcmp(namelist[i]->d_name, ".") || !strcmp(namelist[i]->d_name, "..")) continue;
            stat(namelist[i]->d_name, &statbuf);
            infoFilesSize += statbuf.st_size;
        }
```

```
//printf("dirsize:%ld₩n", infoFilesSize);
// info 디렉토리의 크기가 2KB를 넘어간다면
if (infoFilesSize > 2048){
        // info 디렉토리의 크기가 2KB 이하가 될 때까지
        while(infoFilesSize > 2048){
                struct stat dirstatbuf;
                time_t oldest = time(NULL) + 100000000; // 가장 오래된 파일을 구하기 위한 변수. 삭제된 시간은 현재시간보다는 무조건 이를 것이기 때문에 초기값을 이렇게 설정했다
                char oldestFileName[MAXNAMLEN]; // 가장 오래된 파일의 파일명을 저장할 배열
                FILE *fp;
                char deleteTime[BUFFER_SIZE]; // 삭제된 시간
                char *relativeFileName; // 상대경로명
                char absolutePathName[PATH_MAX]; // 절대경로명
                char trashFilePath[PATH_MAX]; // trash파일의 경로명
                //cnt = scandir(".", &namelist, NULL, NULL);

                // 가장 삭제된지 오래된 파일을 구한다
                for (i = 0; i < cnt; ++i) {
                        if (!strcmp(namelist[i]->d_name, ".") || !strcmp(namelist[i]->d_name, "..")) continue;
                        stat(namelist[i]->d_name, &dirstatbuf);
                        if (dirstatbuf.st_mtime < oldest){
                                oldest = dirstatbuf.st_mtime;
                                strcpy(oldestFileName, namelist[i]->d_name);
                        }

                }
```

```
// 가장 오래된 파일 삭제
if ((fp = fopen(oldestFileName, "r")) == NULL) {
        fprintf(stderr, "fopen error for %s, %s₩n", oldestFileName, strerror(errno));
        chdir("..");
        chdir("..");
        return;
}
fgets(deleteTime, BUFFER_SIZE, fp); // 의미없이 첫줄 읽어옴
fgets(absolutePathName, PATH_MAX, fp); // 절대경로 읽어옴
fgets(deleteTime, BUFFER_SIZE, fp); // 삭제된 시간 읽어옴
deleteTime[strlen(deleteTime) - 1] = '₩0';

// 상대경로 구한다
for(j = strlen(absolutePathName) - 1; j >= 0; --j) {
        if (absolutePathName[j] == '/') {
                relativeFileName = absolutePathName + j + 1;
                break;
        }
}
relativeFileName[strlen(relativeFileName) - 1] = '₩0';

// trash 파일의 경로를 구한다
sprintf(trashFilePath, "../files/%s_%s", deleteTime + 4, relativeFileName);
// trash 파일을 삭제한다
if (remove(trashFilePath) < 0) {
        fprintf(stderr, "remove error for %s, %s₩n", trashFilePath, strerror(errno));
        chdir("..");
```

```
                    chdir("..");
                    return;
                }

                fclose(fp);
                //info 파일을 삭제한다
                if (remove(oldestFileName) < 0) {
                    fprintf(stderr, "remove error for %s, %s₩n", oldestFileName, strerror(errno));
                    chdir("..");
                    chdir("..");
                    return;
                }

                free(namelist);

                // 가장 오래전에 삭제된 파일이 삭제된 info파일의 크기를 구한다
                cnt = scandir(".", &namelist, NULL, NULL);
                infoFilesSize = 0;
                for (i = 0; i < cnt; ++i) {
                    if (!strcmp(namelist[i]->d_name, ".") || !strcmp(namelist[i]->d_name, "..")) continue;
                    stat(namelist[i]->d_name, &statbuf);
                    infoFilesSize += statbuf.st_size;
                }
            }
        }
        chdir("..");
        chdir("..");
```

```
}

void doRecoverCommand(){ // recover 명령어를 수행하는 함수
	struct dirent **namelist;
	int cnt;
	char *nextToken;
	char *filename; // 명령어로 전달된 파일명
	char realFileName[MAXNAMLEN]; // 상대경로명
	int l_option; // l옵션으로 실행되었는지
	int sameNameCnt = 0; // 같은 이름의 파일 개수
	int sameNameIndex[BUFFER_SIZE]; // 같은 이름인 파일들의 인덱스를 저장할 배열
	int selectedIndex; // 같은 이름의 파일들 중 사용자가 선택한 파일의 인덱스
	int i;
	int printedFileCnt; // 출력할 파일의 개수
	FILE *fp;
	char deleteTime[BUFFER_SIZE]; // 삭제시간
	char modifyTime[BUFFER_SIZE]; // 최종 수정시간
	char originFilePath[PATH_MAX]; // 삭제전 파일이 위치했던 경로
	char tmpOriginFilePath[PATH_MAX];
	char *deleteTimePtr; // 삭제시간
	char *modifyTimePtr; // 최종수정시간
	char trashPathName[PATH_MAX]; // trash 파일의 경로
	time_t mtime; // 수정시간 time_t
	struct tm mtm; // 수정시간 struct tm
	struct utimbuf mutimbuf; // 수정시간 struct utimbuf
	struct infoFile infoFiles[BUFFER_SIZE]; // info파일들의 정보를 담은 구조체 배열
```

```
	nextToken = strtok(NULL, " ");
	filename = nextToken;

	nextToken = strtok(NULL, " ");

	if (filename == NULL) return; // 파일명이 주어지지 않은경우
	else if (nextToken == NULL){ // 옵션 없이 파일명만 주어진 경우
		l_option = 0;
	} else if (!strcmp(nextToken, "-l")) { // -l 옵션이 주어진 경우
		l_option = 1;
	} else {
		return;
	}

	if (chdir("trash/info") < 0){ // info 디렉토리로 이동
		return;
	}

	// info 디렉토리의 파일들 확인
	cnt = scandir(".", &namelist, NULL, NULL);
	sameNameCnt = 0;
	printedFileCnt = 0;
	for (i = 0; i < cnt; ++i) {
		if (!strcmp(namelist[i]->d_name, ".") || !strcmp(namelist[i]->d_name, "..")) continue; // 확인할 필요 없는 파일들 pass
```

```
		if (namelist[i]->d_name[strlen(namelist[i]->d_name) - 1] == ')') { // 파일명 끝이 ')'인 info파일은 이름이 중복되어 파일명 뒤에 (숫자)가 붙어있는 것
			strncpy(realFileName, namelist[i]->d_name, strlen(namelist[i]->d_name) - 3); // 원래 파일명을 저장한다
		} else {
			strcpy(realFileName, namelist[i]->d_name);
		}

		if (!strcmp(realFileName, filename)) { // 이름이 일치하는 파일이 있다면
			sameNameIndex[sameNameCnt] = i; // 같은이름 파일들 인덱스 저장하는 배열에 인덱스를 저장한다
			++sameNameCnt; // 같은이름 파일들 인덱스 배열의 원소의 개수를 하나 늘린다
		}

		// l_option 수행
		if (l_option) { // 삭제 시간이 오래된 순으로 출력해야함
			FILE *fp;
			char deleteTime[BUFFER_SIZE];
			char *deleteTimeForPrint;
			if ((fp = fopen(namelist[i]->d_name, "r")) == NULL) {
				fprintf(stderr, "fopen error for %s₩n", namelist[i]->d_name);
				continue;
			}

			// info파일에서 필요한 정보들을 읽어온다
			fgets(deleteTime, BUFFER_SIZE, fp); // 첫줄 의미없이 읽어들임
			fgets(deleteTime, BUFFER_SIZE, fp); // 둘째줄 의미없이 읽어들임
			fgets(deleteTime, BUFFER_SIZE, fp); // 삭제 시간 읽어옴
```

```
                deleteTime[strlen(deleteTime) - 1] = '₩0';
                deleteTimeForPrint = deleteTime + 4;
                // 정렬 후 출력해야 하기 때문에 출력할 정보들을 저장해둔다
                strcpy(infoFiles[printedFileCnt].filename, realFileName);
                strcpy(infoFiles[printedFileCnt].deleteTime, deleteTimeForPrint);

                ++printedFileCnt;
                //printf("%d. %s ₩t%s₩n", ++printedFileCnt, realFileName, deleteTimeForPrint);
        }
}
if(l_option) {
        // -l 옵션으로 출력할 파일들의 목록을 삭제 시간 오름차순으로 정렬한다
        qsort(infoFiles, cnt - 2, sizeof(struct infoFile),   compareInfoFile);

        // 출력한다
        for(i = 0; i < cnt - 2; ++i) {
                printf("%d. %s ₩t%s₩n", i + 1, infoFiles[i].filename, infoFiles[i].deleteTime);
        }
        printf("₩n");
}

if (sameNameCnt == 0) { // 복구할 파일명과 일치하는 파일을 찾지 못했다면
        chdir("..");
        chdir("..");
        printf("There is no ₩'%s₩' in the 'trash' directory₩n", filename);
        return;
} else if (sameNameCnt == 1) { // 복구할 파일명과 일치하는 파일이 한개라면
```

```
            selectedIndex = 0; // 복구할 파일의 인덱스는 0
        } else if (sameNameCnt > 1) { // 복구할 파일명과 일치하는 파일명이 여러개라면
            for (i = 0; i < sameNameCnt; ++i) {
                if ((fp = fopen(namelist[sameNameIndex[i]]->d_name, "r")) == NULL) {
                    fprintf(stderr, "foopen error for %s₩n", namelist[sameNameIndex[i]]->d_name);
                    continue;
                }

                fgets(deleteTime, BUFFER_SIZE, fp); // 첫줄 의미없이 읽어들임
                fgets(deleteTime, BUFFER_SIZE, fp); // 둘째줄 의미없이 읽어들임
                fgets(deleteTime, BUFFER_SIZE, fp); // 삭제시간 읽어옴
                fgets(modifyTime, BUFFER_SIZE, fp); // 수정시간 읽어옴
                deleteTime[strlen(deleteTime) - 1] = '₩0';
                modifyTime[strlen(modifyTime) - 1] = '₩0';
                printf("%d. %s %s %s₩n", i + 1, filename, deleteTime, modifyTime); // 중복되는 파일들의 정보 출
력

                fclose(fp);
            }
            // 중복되는 파일들 중 어떤 파일을 복구할 것인지 사용자가 선택하도록 한다
            printf("Choose : ");
            scanf("%d", &selectedIndex);
            while(getchar()!='₩n');
            --selectedIndex; // 사용자가 선택한 인덱스보다 실제 인덱스는 1 작으므로 1을 뺀다
            if(selectedIndex < 0 || sameNameCnt <= selectedIndex) { // 사용자가 선택한 인덱스가 잘못되었다면
                chdir("..");
                chdir("..");
```

```
                return;
        }
} else {
        chdir("..");
        chdir("..");
        printf("There is no ₩'%s₩' in the 'trash' directory₩n", filename);
        return;
}

if ((fp = fopen(namelist[sameNameIndex[selectedIndex]]->d_name, "r")) == NULL) {
        fprintf(stderr, "foopen error for %s₩n", namelist[sameNameIndex[i]]->d_name);
        chdir("..");
        chdir("..");
        return;
}

fgets(deleteTime, BUFFER_SIZE, fp); // 첫줄 의미없이 읽어들임
fgets(originFilePath, PATH_MAX, fp); // 원본 파일 경로 읽어옴
fgets(deleteTime, BUFFER_SIZE, fp); // 삭제 시간 읽어옴
fgets(modifyTime, BUFFER_SIZE, fp); // 수정 시간 읽어옴
originFilePath[strlen(originFilePath) - 1] = '₩0';
deleteTime[strlen(deleteTime) - 1] = '₩0';
modifyTime[strlen(deleteTime) - 1] = '₩0';
deleteTimePtr = deleteTime + 4;
modifyTimePtr = modifyTime + 4;
sprintf(trashPathName, "../files/%s_%s", deleteTimePtr, filename); // trash 파일의 경로 만든다
```

```
if (access(originFilePath, F_OK) == 0) { // 복구할 파일명이 중복되는 경우
        // 복구할 파일명의 앞에 "숫자_" 를 붙여 덮어써지지 않도록 한다
        strcpy(tmpOriginFilePath, originFilePath);
        for (i = strlen(tmpOriginFilePath) - 1; i >= 0; --i) {
                if (tmpOriginFilePath[i] == '/') {
                        tmpOriginFilePath[i] = 0;
                        break;
                }
        }

        i = 1;
        do {
                sprintf(originFilePath, "%s/%d_%s", tmpOriginFilePath, i, filename);
                ++i;
        } while (access(originFilePath, F_OK) == 0);
}
// trash 파일 복구
rename(trashPathName, originFilePath);

//복구된 st_mtime 도 원래대로 복구한다
mtime;
mtm.tm_year = atoi(strtok(modifyTimePtr, "-")) - 1900;
mtm.tm_mon = atoi(strtok(NULL, "-")) - 1;
mtm.tm_mday = atoi(strtok(NULL, " "));

mtm.tm_hour = atoi(strtok(NULL, ":"));
mtm.tm_min = atoi(strtok(NULL, ":"));
```

```
		mtm.tm_sec = atoi(strtok(NULL, " "));

		//printf("recover modtime : %d-%d-%d %d:%d₩n", mtm.tm_year, mtm.tm_mon, mtm.tm_mday, mtm.tm_hour, mtm.tm_min);
		mtime = mktime(&mtm); // 수정 시간을 time_t로 변환해 저장
		mutimbuf.actime = mtime;
		mutimbuf.modtime = mtime;
		utime(originFilePath, &mutimbuf);

		//info 파일 삭제
		remove(namelist[sameNameIndex[selectedIndex]]->d_name);

		chdir("..");
		chdir("..");
		return;
}

void alarmHandlercheckFileDeleteTime(){// SIGALRM이 전달됐을 때 삭제할 파일이 있는지 확인하는 함수
		checkFileDeleteTime(); // 삭제할 파일이 있는지 확인
		if(deleteUsingRFlag) { // -r 옵션으로 삭제된 파일이 있다면
			printf("20160548>"); // 프롬프트 메세지를 다시 출력해야함
			deleteUsingRFlag = 0; // 플래그는 다시 0으로 바꿈
		}
}

void doSizeCommand(){ // size 명령어를 수행하는 함수
		char *nextToken;
```

```
char *filename;
struct stat statbuf;
off_t filesize;
int i;

fileSizes.size = 0;

nextToken = strtok(NULL, " ");
filename = nextToken;
if (filename == NULL) return; // 파일명 전달되지 않았다면 바로 함수 ㅈ ㅗㅇ료

// 해당 파일의 stat 구조체 가져온다
if (stat(filename, &statbuf) == -1) {
        fprintf(stderr, "stat error for %s₩n", filename);
        printf("ERROR:%s₩n", strerror(errno));
        return;
}

nextToken = strtok(NULL, " ");
if (nextToken == NULL) { // 인자 전달되지 않음, 해당 파일 정보만 출력
        if (S_ISDIR(statbuf.st_mode)) { // 해당 파일이 디렉토리라면
                filesize = getDirSize(filename); // 디렉토리 크기 구한다
        } else { // 일반파일이라면
                filesize = statbuf.st_size; // stat 구조체 이용하여 파일 크기 구한다
        }
        printf("%ld₩t./%s₩n", filesize, filename); // 해당 파일의 크기 출력
        chdir("..");
```

```
}else if (!strcmp(nextToken, "-d")) { // -d 옵션 전달됐으면
		char dirpath[PATH_MAX];
		int maxDepth; // 탐색 최대 깊이 저장할 변수
		nextToken = strtok(NULL, " ");
		if(nextToken == NULL) return;
		maxDepth = atoi(nextToken); // 명령어에 전달된 탐색 최대 깊이 저장

		// 해당 파일이 일반파일이라면 하위 디렉토리가 없으므로
		if (!S_ISDIR(statbuf.st_mode)) {
			filesize = statbuf.st_size;
			printf("%ld₩t./%s₩n", filesize, filename); // 해당 파일의 정보만 출력하고
			return; // 함수종료
		}

		sprintf(dirpath, "./%s/", filename);
		//printf("%ld₩t./%s₩n", getDirSize(filename), filename);
		sprintf(fileSizes.fileSize[fileSizes.size].pathname, "./%s", filename); // 파일 목록 정렬 후 출력하기 위해 출력할 파일의 파일명을 저장해둔다
		fileSizes.fileSize[fileSizes.size].filesize = getDirSize(filename);  // 파일 목록 정렬 후 출력하기 위해 출력할 파일의 크기를 저장해둔다
		++(fileSizes.size);  // 출력할 파일 목록의 크기 + 1
		chdir(exec_path);
		if (maxDepth > 1) {
			getPrintingDirs(filename, dirpath, 2, maxDepth); // 재귀호출하며 출력할 파일들을 찾아 목록에 추가한다
		}
```

```
                qsort(fileSizes.fileSize, fileSizes.size, sizeof(struct FileSize),  compareFileSize); // 출력할 파일들 목록을 파일명 오름차순으로 정렬한다

                // 출력할 파일들 목록을 출력한다
                for (i = 0; i < fileSizes.size; ++i) {
                        printf("%ld₩t%s₩n", fileSizes.fileSize[i].filesize, fileSizes.fileSize[i].pathname);
                }
        } else { // 그 외의 경우(명령어 잘못된 경우)
                return; // 함수 종료
        }

}

void getPrintingDirs(const char *dirname, const char *dirpath, int depth, int maxDepth){ // size 명령어에 의해 출력될 파일들을 확인해 fileSizes 구조체에 추가하는 함수
        int i;
        char filename[MAXNAMLEN]; // 파일명 저장할 배열
        struct dirent *dentry;
        struct stat statbuf;
        struct dirent **nextdirlist;
        DIR *dir = opendir(dirname);
        char nextDirPath[PATH_MAX]; // 다음 디렉토리에서 사용할 dirpath
        if (dir == NULL) {
                fprintf(stderr, "opendir error, %s₩n", strerror(errno));
                exit(1);
        }
        chdir(dirname);
```

```
// 디렉토리 내의 모든 파일들을 확인
while((dentry = readdir(dir)) != NULL) {
        if (dentry->d_ino == 0) continue;

        memcpy(filename, dentry->d_name, MAXNAMLEN); // 파일명 복사
        if (!strcmp(filename, ".") || !strcmp(filename, "..")) continue;

        // 해당 파일의 stat 구조체 가져옴
        if (stat(filename, &statbuf) == -1) {
                fprintf(stderr, "stat error for %s₩n", filename);
                printf("ERROR:%s₩n", strerror(errno));
                exit(1);
        }

        if (S_ISDIR(statbuf.st_mode)) { // 디렉토리 파일이라면
                int cnt;
                if ((cnt = scandir(filename, &nextdirlist, NULL, NULL)) >= 2) {
                        // 재귀호출
                        if(depth < maxDepth) {
                                sprintf(nextDirPath, "%s%s/", dirpath, filename); // 다음 디렉토리에서 사용할 dirpath를 만든다
                                getPrintingDirs(filename, nextDirPath, depth + 1, maxDepth); // 재귀호출
                                chdir("..");
                        }
                        sprintf(fileSizes.fileSize[fileSizes.size].pathname, "%s%s", dirpath, filename);// 파일 목록 정렬 후 출력하기 위해 출력할 파일의 파일명을 저장해둔다
                        fileSizes.fileSize[fileSizes.size].filesize = getDirSize(filename);// 파일 목록 정렬 후 출력하
```

```
기 위해 출력할 파일의 크기를 저장해둔다
                                ++(fileSizes.size);   // 파일 목록의 유효한 원소 개수 + 1
                                // 작업 후 원래 디렉토리로 복귀하기
                                chdir("..");
                        }

                        if(cnt != -1)
                                free(nextdirlist);
                } else {
                        sprintf(fileSizes.fileSize[fileSizes.size].pathname, "%s%s", dirpath, filename);// 파일 목록 정렬 후 출력하기 위해 출력할 파일의 파일명을 저장해둔다
                        fileSizes.fileSize[fileSizes.size].filesize = statbuf.st_size;// 파일 목록 정렬 후 출력하기 위해 출력할 파일의 크기를 저장해둔다
                        ++(fileSizes.size); // 파일 목록의 유효한 원소 개수 + 1
                }
        }

        if (closedir(dir) < 0) {
                fprintf(stderr, "closedir error₩n");
                exit(1);
        }
}

off_t getDirSize(const char *dirname){ // 디렉토리의 크기를 구하는 함수
        int i;
        char filename[MAXNAMLEN]; // 파일명 저장할 배열
        struct dirent *dentry;
        struct stat statbuf;
```

```
struct dirent **nextdirlist;
DIR *dir = opendir(dirname);
off_t totalSize = 0;
if (dir == NULL) {
        fprintf(stderr, "opendir error₩n");
        exit(1);
}
chdir(dirname);


while((dentry = readdir(dir)) != NULL) { // 디렉토리 내의 모든 파일들을 확인
        if (dentry->d_ino == 0) continue;

        memcpy(filename, dentry->d_name, MAXNAMLEN);
        if (!strcmp(filename, ".") || !strcmp(filename, "..")) continue; // 확인할 필요 없는 파일은 패스

        // 파일의 stat 구조체를 가져온다
        if (stat(filename, &statbuf) == -1) {
                fprintf(stderr, "stat error for %s₩n", filename);
                printf("ERROR:%s₩n", strerror(errno));
                exit(1);
        }

        if (S_ISDIR(statbuf.st_mode)) { // 디렉토리 파일이라면
                int cnt;
                if ((cnt = scandir(filename, &nextdirlist, NULL, NULL)) >= 2) { // 해당 디렉토리에 파일이 들어있으
면
```

```
                                        // 재귀호출
                                        totalSize += getDirSize(filename); // 해당 디렉토리의 크기를 재귀호출로 구해 totalSize
에 더한다
                                        // 작업 후 원래 디렉토리로 복귀하기
                                        chdir("..");

                                }

                                if(cnt != -1)
                                        free(nextdirlist);
                        } else { // 일반 파일 이라면
                                totalSize += statbuf.st_size; // 해당 파일의 크기를 totalSize에 더한다
                        }
                }

                if (closedir(dir) < 0) {
                        fprintf(stderr, "closedir error₩n");
                        exit(1);
                }

                return totalSize;
}

int compareInfoFile(const void *a, const void *b){ // infoFile을 정렬하는데 사용하는 비교함수
                const struct infoFile *infoFileA = (const struct infoFile *)a;
                const struct infoFile *infoFileB = (const struct infoFile *)b;
```

```
	return strcmp(infoFileA->deleteTime, infoFileB->deleteTime);
}

int compareFileSize(const void *a, const void *b){ // FileSize를 정렬하는데 사용하는 비교함수
	const struct FileSize *fileSizeA = (const struct FileSize *)a;
	const struct FileSize *fileSizeB = (const struct FileSize *)b;

	return strcmp(fileSizeA->pathname, fileSizeB->pathname);
}
```

<monitor.c>

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

#define MAX_FILE_CNT 1024

#define TARGET_DIRECTORY_NAME "check"

struct FileInfo { // 파일 정보를 담는 구조체
        char absolutePathName[PATH_MAX]; // 파일의 절대경로명
        time_t lastModifyTime; // 파일의 최종 수정시간
};

struct FileData { // 파일 정보들의 배열 관리 구조체
        struct FileInfo files[MAX_FILE_CNT]; // 파일 정보 배열
```

```
	int filesCnt; // 배열의 유효한 원소의 개수
};

void checkFileDataStatus(const struct FileData *prevFileData, const struct FileData *curFileData); // 이전 파일들 목록과 갱신된 파일들 목록을 확인하여 수정, 추가, 삭제된 파일이 있는지 확인하는 함수
void updateFileDataStatus(struct FileData **prevFileData, struct FileData **curFileData); // 파일 목록을 갱신하는 함수
void getCurFileState(struct FileData *fileData, const char *dirname); // 파일들 상태를 가져오는 재귀호출 함수
void printFileData(const struct FileData *fileData); // 파일 데이터를 출력하는 함수, 디버그용
void sortFileData(struct FileData *fileData); // 파일 데이터를 파일명 기준으로 정렬하는 함수
int compareFileInfo(const void *a, const void *b); // 파일 데이터를 정렬할 때 사용하는 비교함수
int ssu_daemon_init(const char *path); // 디몬 프로세스 실행 함수
const char *getRelativePath(const char *absolutePath); // 절대경로에서 상대경로 구하는 함수

void startMntr(void) // 모니터링 디몬 프로세스를 실행하는 함수. 이 함수는 ssu_mntr.c의 메인함수에서 호출함
{
	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;
	}
}
```

```
void checkFileDataStatus(const struct FileData *prevFileData, const struct FileData *curFileData) {// 이전 파일들 목록과 갱신된 파일들 목록을 확인하여 수정, 추가, 삭제된 파일이 있는지 확인하는 함수
	int i, j;
	FILE *logfp;
	time_t curtime;
	struct tm *tmpointer;
	char timebuf[64];
	time(&curtime);
	tmpointer = localtime(&curtime);
	sprintf(timebuf, "%d-%02d-%02d %02d:%02d:%02d", 1900 + tmpointer->tm_year, tmpointer->tm_mon + 1, tmpointer->tm_mday, tmpointer->tm_hour, tmpointer->tm_min, tmpointer->tm_sec);

	if ((logfp = fopen("log.txt", "a")) == NULL) {
		fprintf(stderr, "fopen error for log.txt₩n");
		exit(1);
	}

	// 이진 탐색으로 현재 파일 목록의 파일과 과거 파일 목록의 데이터를 비교한다
	for (i = 0; i < curFileData->filesCnt; ++i) {
		int fileNameCmpResult;
		int start, end, mid;
		int sameNameFindFlag = 0;
		// 이진 탐색으로 해당 파일이 이전 파일 목록에도 존재하는지 확인한다
		start = 0;
		end = prevFileData->filesCnt - 1;
		while(start <= end) {
			mid = (start + end) / 2;
```

```
                fileNameCmpResult = strcmp(prevFileData->files[mid].absolutePathName, curFileData->files[i].absolutePathName);
                if (fileNameCmpResult == 0) {
                        sameNameFindFlag = 1;
                        break;
                } else if (fileNameCmpResult > 0) {
                        end = mid - 1;
                } else {
                        start = mid + 1;
                }
        }

        if (sameNameFindFlag) { // 이름이 일치하는 파일을 찾았다면
                if (prevFileData->files[mid].lastModifyTime == curFileData->files[i].lastModifyTime) { // 파일 수정되지 않은 경우
                        ;
                } else { // 파일 수정된 경우
                        // 파일 수정 메세지 출력
                        fprintf(logfp, "[%s][modify_%s]₩n", timebuf, getRelativePath(curFileData->files[i].absolutePathName));
                        //printf("%s is modified₩n", curFileData->files[i].absolutePathName);
                }
        } else { // 파일 새로 추가된 경우
                // 파일 추가 메세지 출력
                fprintf(logfp, "[%s][create_%s]₩n", timebuf, getRelativePath(curFileData->files[i].absolutePathName));
                //printf("%s is created₩n", curFileData->files[i].absolutePathName);
        }
```

```
	}

	// 이진 탐색으로 과거 파일 목록의 파일과 현재 파일 목록의 데이터를 비교한다
	for (i = 0; i < prevFileData->filesCnt; ++i) {
		int start, end, mid;
		int fileNameCmpResult;
		int sameNameFindFlag = 0;
		// 이진 탐색으로 해당 파일이 현재 파일 목록에도 존재하는지 확인한다
		start = 0;
		end = curFileData->filesCnt - 1;
		mid = (start + end) / 2;
		while(start <= end) {
			mid = (start + end) / 2;
			fileNameCmpResult = strcmp(curFileData->files[mid].absolutePathName, prevFileData->files[i].absolutePathName);

			if (fileNameCmpResult == 0) {
				sameNameFindFlag = 1;
				break;
			} else if (fileNameCmpResult > 0) {
				end = mid - 1;
			} else {
				start = mid + 1;
			}
		}

		if (sameNameFindFlag) { // 위에서 이미 검사 함
```

```
                continue;
            } else { // 파일 삭제된 경우
                // 파일 삭제 메세지 출력
                fprintf(logfp, "[%s][delete_%s]₩n", timebuf, getRelativePath(prevFileData->files[i].absolutePathName));
                //printf("%s is deleted₩n", prevFileData->files[i].absolutePathName);
            }
        }

        fclose(logfp);
}

void updateFileDataStatus(struct FileData **prevFileData, struct FileData **curFileData) { // 파일 목록을 갱신하는 함수
        char path[PATH_MAX];
        struct FileData *tmp;
        getcwd(path, PATH_MAX);

        // 새롭게 데이터를 갱신해야 하므로 curFileData를 prevFileData로 옮김, prevFileData는 새로운 파일 정보를 넣어 재활용
        tmp = *prevFileData;
        *prevFileData = *curFileData;
        *curFileData = tmp;

        // 새로운 파일 데이터 생성(getCurFileState 호출) 전에 반드시 filesCnt를 0으로 만들어 줘야 한다
        (*curFileData)->filesCnt = 0;
        getCurFileState(*curFileData, TARGET_DIRECTORY_NAME);
        sortFileData(*curFileData); // 새로 가져온 파일 목록을 정렬한다
```

```
        chdir(path);

        return;
}

void getCurFileState(struct FileData *fileData, const char *dirname){ // 파일들 상태를 가져오는 재귀호출 함수
        char filename[MAXNAMLEN];
        char absolutePathName[PATH_MAX];
        struct dirent *dentry;
        struct stat statbuf;
        DIR *dir = opendir(dirname);
        if (dir == NULL) {
                fprintf(stderr, "opendir error₩n");
                exit(1);
        }
        chdir(dirname);

        while((dentry = readdir(dir)) != NULL) { // 모든 파일들 확인
                if (dentry->d_ino == 0) continue;

                memcpy(filename, dentry->d_name, MAXNAMLEN);
                if (!strcmp(filename, ".") || !strcmp(filename, "..")) continue; // 확인할 필요 없는 파일들 패스

                realpath(filename, absolutePathName); // 파일의 상대경로 가져옴

                // 해당 파일의 stat 구조체 가져온다
                if (stat(absolutePathName, &statbuf) == -1) {
```

```
                fprintf(stderr, "stat error for %s₩n", absolutePathName);
                printf("ERROR:%s₩n", strerror(errno));
                exit(1);
            }

            if (S_ISDIR(statbuf.st_mode)) { // 해당 파일이 디렉토리 파일이라면
                // 재귀호출
                getCurFileState(fileData, absolutePathName);
                // 작업 후 원래 디렉토리로 복귀하기
                chdir("..");
            }
            // 해당 파일의 정보를 파일 목록에 추가한다
            strcpy(fileData->files[fileData->filesCnt].absolutePathName, absolutePathName);
            fileData->files[fileData->filesCnt].lastModifyTime = statbuf.st_mtime;
            ++(fileData->filesCnt);
        }

        if (closedir(dir) < 0) {
            fprintf(stderr, "closedir error₩n");
            exit(1);
        }
}

void printFileData(const struct FileData *fileData) { // 파일 목록 출력하는 함수 - 디버그용
        int i;
        printf("print file data start₩n");
        // 파일 목록 전체 순회하며 출력
```

```
	for (i = 0; i < fileData->filesCnt; ++i) {
		printf("name : %s, st_mtime : %ld₩n", fileData->files[i].absolutePathName, fileData->files[i].lastModifyTime);
	}
	printf("print file data end₩n");
	printf("₩n");
}

void sortFileData(struct FileData *fileData){ // 파일 목록 파일명 기준으로 정렬하는 함수
	qsort(fileData->files, fileData->filesCnt, sizeof(struct FileInfo), compareFileInfo);
}

int compareFileInfo(const void *a, const void *b) { // 파일 목록 정렬하는데 사용하는 비교함수
	return strcmp(((const struct FileInfo *)a)->absolutePathName, ((const struct FileInfo *)b)->absolutePathName);
}

int ssu_daemon_init(const char *path) { // 디몬 프로세스 시작하는 함수
	pid_t pid;
	int fd, maxfd;

	struct FileData *prevFileData;
	struct FileData *curFileData;

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

```
        exit(0);

pid = getpid();
//printf("process %d running as daemon₩n", pid);
setsid();
signal(SIGTTIN, SIG_IGN);
signal(SIGTTOU, SIG_IGN);
signal(SIGTSTP, SIG_IGN);
maxfd = getdtablesize();

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

umask(0);
chdir(path);
fd = open("/dev/null", O_RDWR);
dup(0);
dup(0);

prevFileData = (struct FileData *) malloc(sizeof(struct FileData)); // 이전 파일 목록 저장할 구조체 동적할당
curFileData = (struct FileData *) malloc(sizeof(struct FileData)); // 현재 파일 목록 저장할 구조체 동적할당
prevFileData->filesCnt = 0; // 이전 파일 목록 구조체 초기화
curFileData->filesCnt = 0; // 현재 파일 목록 구조체 초기화

updateFileDataStatus(&prevFileData, &curFileData); // 파일 목록 갱신
while(1) {
        updateFileDataStatus(&prevFileData, &curFileData); // 파일 목록 갱신
```

```
            checkFileDataStatus(prevFileData, curFileData); // 변경된 파일 있는지 확인
            sleep(1); // 1초 sleep
        }

        return 0;
}

const char *getRelativePath(const char *absolutePath) { // 절대경로에서 상대경로를 구해오는 함수
        int i;
        for (i = strlen(absolutePath) - 1; i >= 0; --i) { // 절대경로 문자열의 맨 뒤부터 앞으로 오면서 확인
            if (absolutePath[i] == '/') break; // '/' 문자열을 만났다면 반복 중지
        }

        return absolutePath + i + 1; // '/' 문자열의 바로 뒤 문자의 포인터 리턴
}
```