컴퓨터학부 20142314 김승호

1. 과제 개요

ssu_backup 프로그램은 리눅스 시스템 상에서 사용자가 원하는 파일을 백업하고 복구할 수 있는 프로그램이다. 프로그램은 명령 프롬프트를 제공하여 사용자가 원하는 명령어를 입력하여 백업 파일을 추가하고 제거 할 수 있다. ls 명령어, vim 명령어 또한 제공하여 디렉토리를 확인하거나, 파일을 수정할 수 있다. 한 파일의 백업은 한 쓰레드가 담당하게 되는데, 이를 통해 여러 파일이 서로 다른 옵션과 시간 간격으로 백업될 수 있다. 파일이 추가되거나, 삭제, 백업, 복구 될 때 이에 대한 정보가 로그에 기록되는데, mutex를 사용하여 서로 다른 쓰레드에서 로그를 기록하여도 시간 순서대로 로그가 기록되게 만들어졌다.

2. 설계

이 프로그램의 가장 핵심적인 부분은 백업 파일을 관리하는 부분이다. 백업 리스트에는 여러 파일이 존재하며, 각각의 파일들마다 여러 개의 복사본이 존재한다. 이 프로그램에서는 다음과 같이 백업 파일 리스트의 구조를 설계하였다.

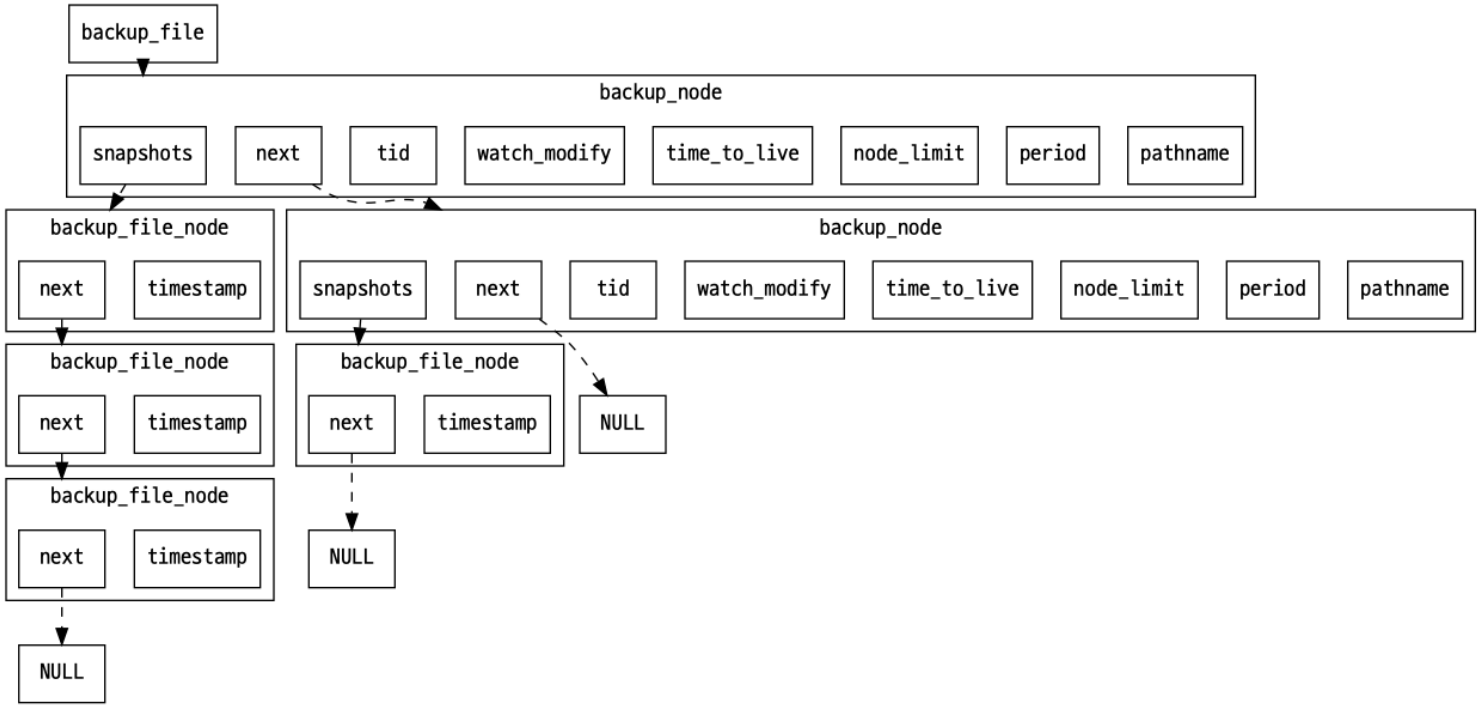

그림 1 백업 파일 리스트의 구조도

백업 파일은 backup_node, 각 노드의 복사본은 backup_file_node를 하나씩 갖는다.
backup_node의 pathname은 백업할 파일의 절대 경로, period는 백업 간격, node_limit은 최대 파일 수 (-n 옵션), time_to_live는 파일의 수명 (-t 옵션), watch_modify는 파일의 수정을 확인할지 여부 (-m 옵션)을 나타낸다.
tid는 실행할 쓰레드의 id이며, 파일 하나당 쓰레드를 하나 씩 갖게 된다.
backup_node의 snapshots는 backup_file_node를 가리키는데, 이 구조체는 파일이 백업된 시간(timestamp)를 갖는다.

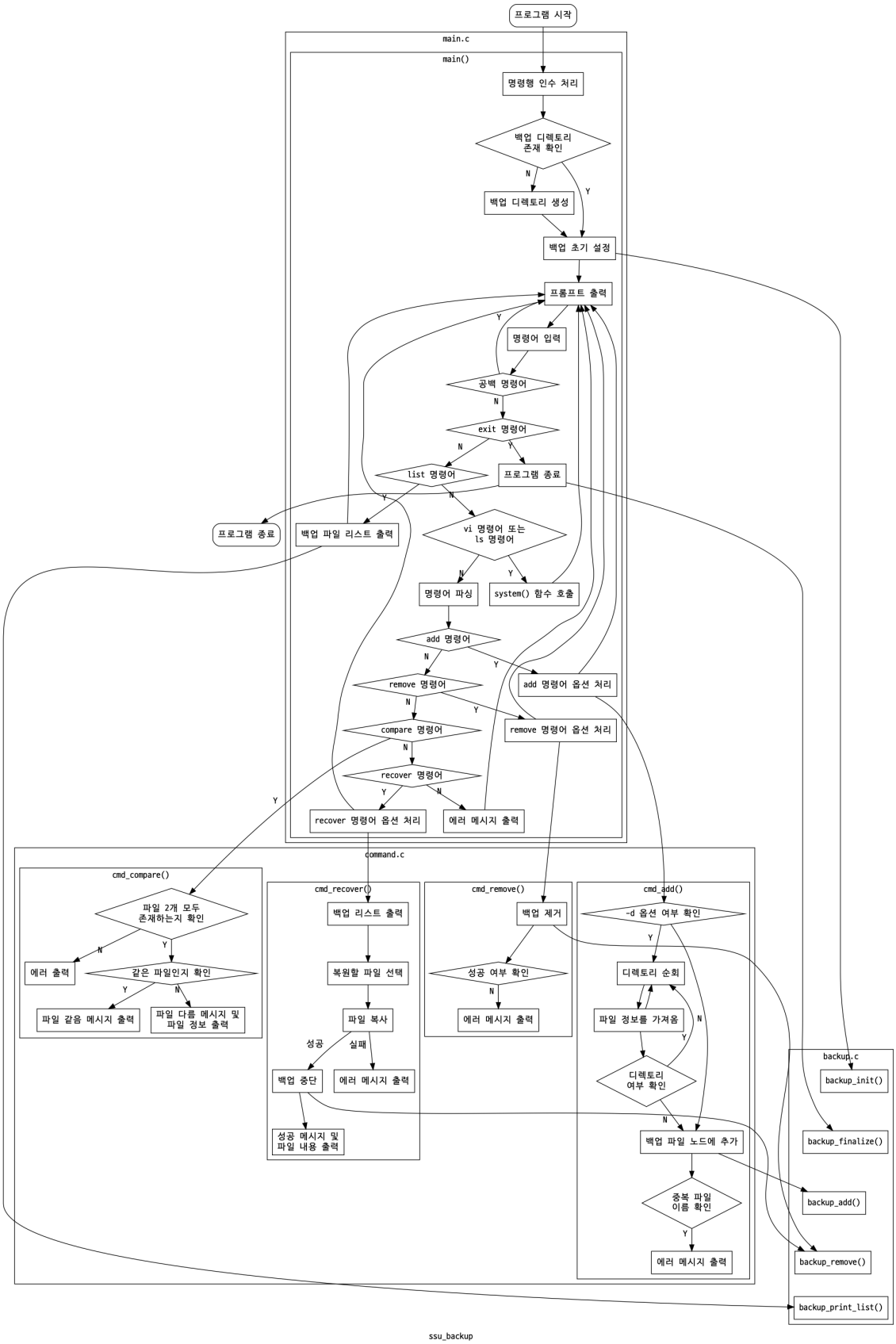

그림 2 프로그램의 전체적인 구조도

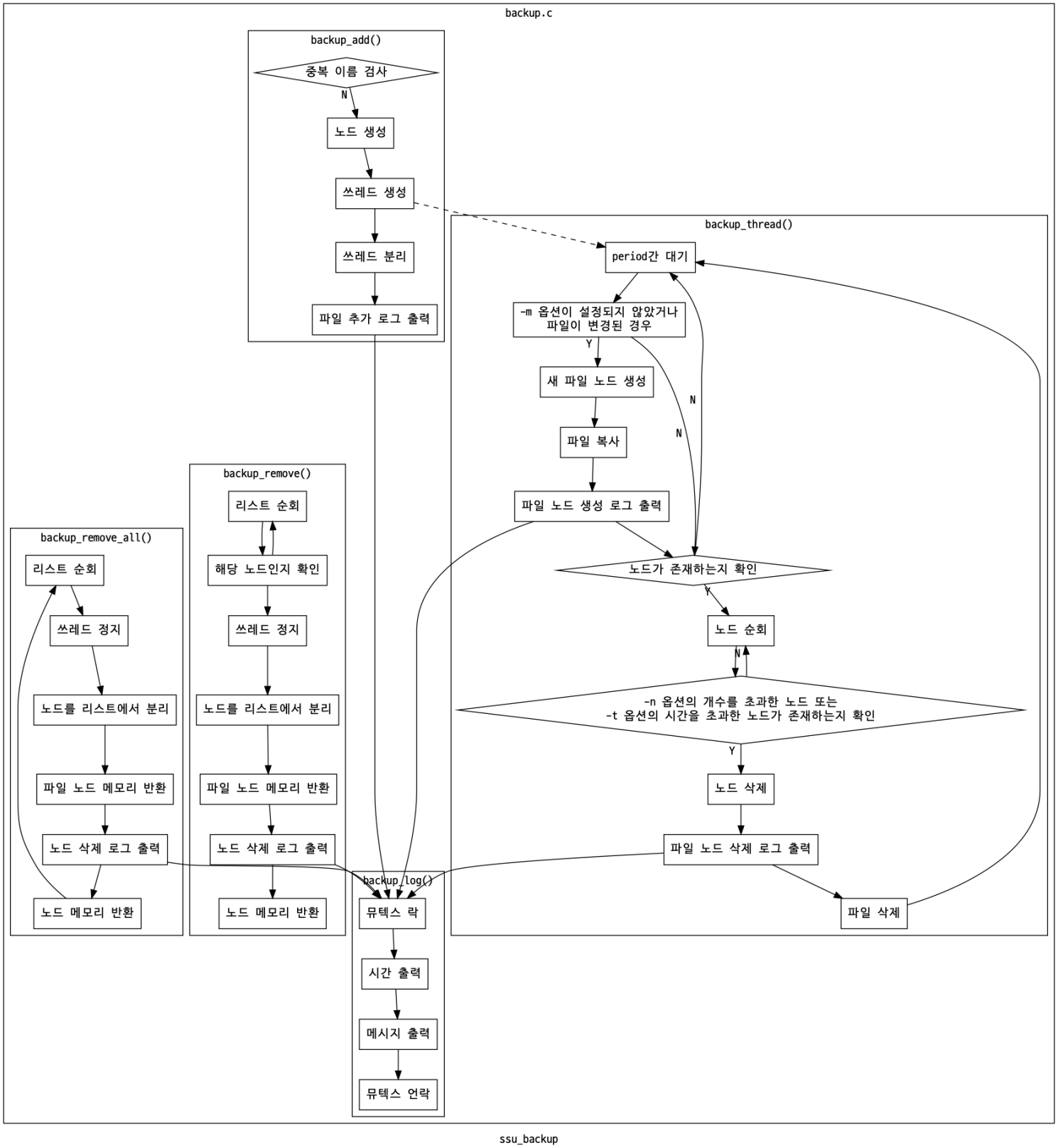

그림 3 백업의 주요 함수가 작동하는 과정

3. 구현

<command.h>

```
void cmd_add(const char *pathname, int period, bool recursive, int node_limit,
             int time_to_live, bool watch_modify);
// 백업리스트 파일 제거
void cmd_remove(const char *pathname);
// 두 파일 비교
void cmd_compare(const char *file1, const char *file2);
// 파일 복구
void cmd_recover(const char *pathname, const char *new_pathname);
```

<backup.h>

```
// 백업 초기 설정
void backup_init(const char *backup_path);
// 백업 마무리
void backup_finalize();
// 백업 로그 함수
void backup_log(const char *format, ...);
// 백업 디렉토리 반환
char *backup_get_backup_path();
// 특정 경로의 백업 파일에 대한 노드 반환
backup_node *backup_get_backup_node(const char *pathname);
// 백업 파일 추가
int backup_add(const char *pathname, int period, int node_limit,
               int time_to_live, bool watch_modify);
// 백업 파일 제거
int backup_remove(const char *pathname);
// 백업 파일 모두 제거
void backup_remove_all();
// 백업 파일 리스트 출력
void backup_print_list();
// 백업 파일의 이름 생성
void backup_get_filename(char *pathname, const char *original,
                         const time_t *time_val);
// 백업 파일이 수정되었는지 확인
bool backup_check_modified(backup_node *node);

// 백업 쓰레드가 수행하는 함수
void *backup_thread(void *arg);
```

<util.h>

```
// 표준 입력 함수
void getln(char *buf);

// 문자열이 특정 문자열로 시작하는지 검사
bool string_starts_with(const char *str, const char *prefix);

// 두 파일 비교
int file_compare(const char *file1, const char *file2);
// 파일 복사
int file_copy(const char *src, const char *dest);

// 파일의 절대 경로를 구함
void path_normalize(const char *path, char *normalized_path);
// 경로에서 파일 이름을 구함
char *path_get_filename(const char *path);
```

프로그램이 시작되면 가장 먼저 명령행 인자에 대한 처리를 한다. 만약 명령행 인자를 하나 입력받은 경우, 경로는 백업 디렉토리는 현재 디렉토리 아래에 생성된다. 만약 명령행 인자를 두 개 입력받은 경우 두 번째 인자가 만약 존재하며, 권한을 가진 디렉토리인 경우 해당 디랙토리의 아래에 백업 디렉토리를 생성한다. 만약 올바른 디렉토리를 입력받지 않았거나 입력받은 명령행 인자의 수가 2개를 초과하는 경우에는 에러 메시지를 띄우고 종료한다.
이후 프롬프트가 실행되어 명령어를 입력받게 되는데, 입력 받은 명령어에 따라 프로그램은 다르게 동작한다.
add 명령어를 입력 받은 경우 메인 함수에서는 백업할 파일의 주소, 파일을 백업할 시간 간격, 그리고 파일의 옵션을 처리하고 파일의 존재 여부, 형식이 올바른지 여부를 검사하여 cmd_add()를 호출한다. cmd_add()에서는 디렉토리를 -d 옵션과 인자로 받은 경우를 검사하여 디렉토리를 순회하고 재귀호출하여 디렉토리 내의 모든 파일이 백업될 수 있도록 한다. 일반 파일이 cmd_add()로 호출된 경우에는 backup_add() 함수를 호출하는데, 이 함수에서는 파일에 대한 노드를 생성하고 backup_thread()를 실행하는 쓰레드를 생성하여 주기적으로 파일이 백업될 수 있도록 한다.
파일이 백업될 때에는 파일을 복사하기 위해 file_copy() 명령어를 사용하는데, 이 명령어는 파일의 권한, 소유자, 수정 시간 등을 보존하여 복사하도록 구현되었다.
remove 명령어를 입력 받은 경우 프로그램은 옵션에 따라 backup_remove() 함수나 backup_remove_all() 함수를 호출하여 해당 파일이나 모든 파일에 대해 쓰레드를 중지 한 뒤, 링크드 리스트에서 파일에 대한 노드를 제거하여 백업을 중지한다.
compare 명령어는 두 파일을 비교하는데, 두 파일의 수정 시간, 크기를 비교하여 해당 파일이 같거나 다른지 메시지를 표시하며, 만약 다른 파일인 경우 두 파일의 수정 시간과 크기 정보를 출력한다.
recover 명령어는 파일을 복구하는 옵션으로, 옵션에 따라 파일을 기존 파일로 복원하거나, 새 파일로 복원할 수 있다. 복원에 성공하면 복원된 파일 정보를 출력하고, 만약 도중에 백업 파일이 -t 옵션이나 -n 옵션을 통해 제거되어 복원에 실패하는 경우 에러 메시지를 출력한다.
list 명령어는 백업 파일 리스트를 순회하며 파일의 경로, 백업 간격, 옵션을 출력한다.
vi / vim / ls 명령어는 system() 함수를 통해 실행되는데, 이를 통해 프로그램을 종료하지 않고 디렉토리를 확인하거나 파일을 수정할 수 있도록 하였다.
exit 명령어는 mutex를 제거하고, 로그 파일을 flush한 뒤, 프로그램을 종료한다.
백업이 진행될 때 프로그램은 로그 파일을 생성하는데, 로그 생성은 mutex를 이용하여 여러 쓰레드에서 동시에 쓰기를 진행하더라도 문제 없이 순차적으로 기록할 수 있도록 구현되었으며, 라인 버퍼를 사용하도록 설정하여 도중에 로그 파일을 오픈하였을 때 끊어진 라인이 없도록 하였다.

4. 테스트 및 결과

테스트는 Docker의 ubuntu:16.04 이미지를 통해 리눅스 환경에서 실행되었습니다.

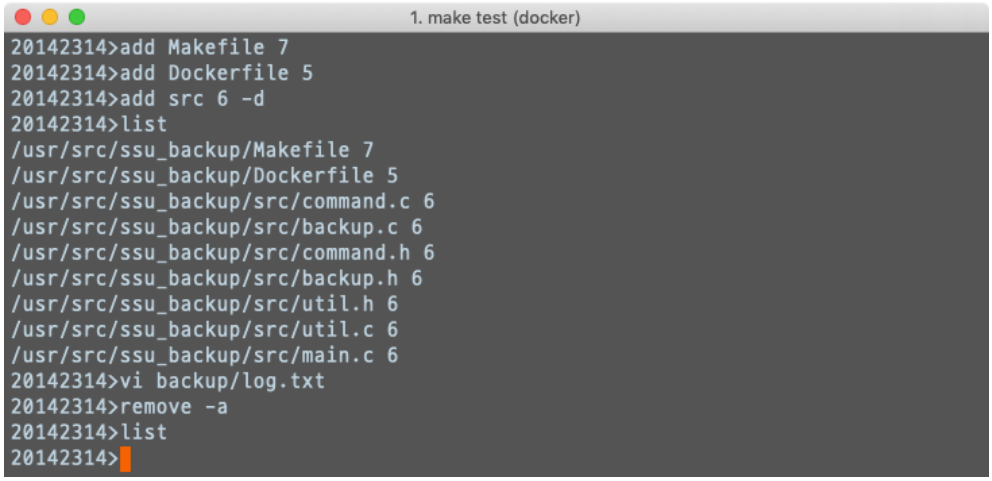

그림 4 여러 파일 추가 후 remove -a를 사용하는 예.

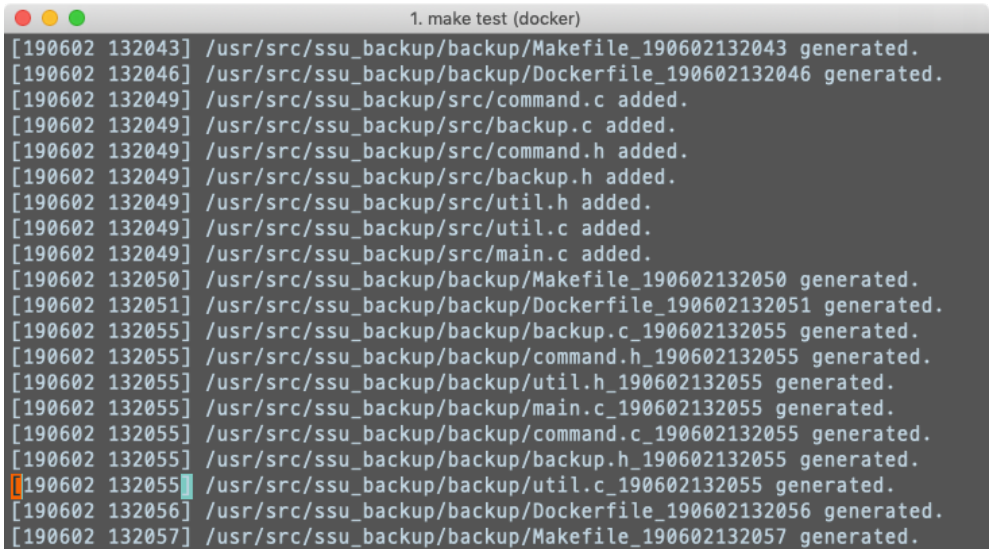

그림 5 위에서 생성된 로그 파일을 vi 명령어를 통해 오픈한 모습

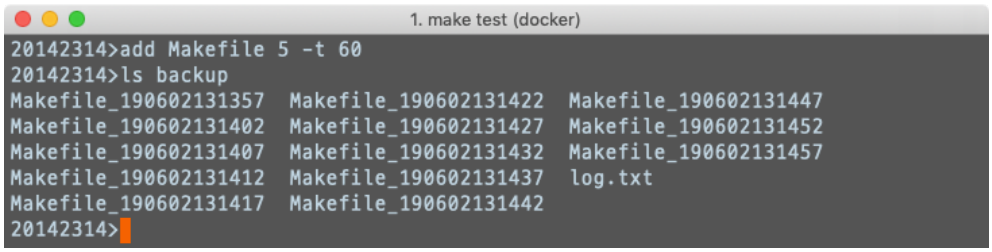

그림 6 -t 옵션을 사용한 예

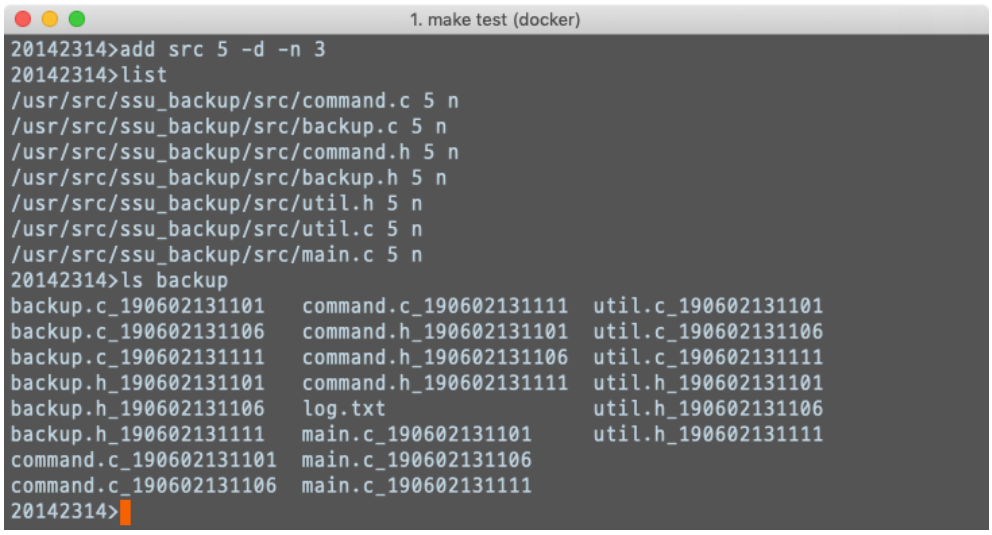

그림 7 -d 옵션과 -n 옵션을 동시에 사용한 예

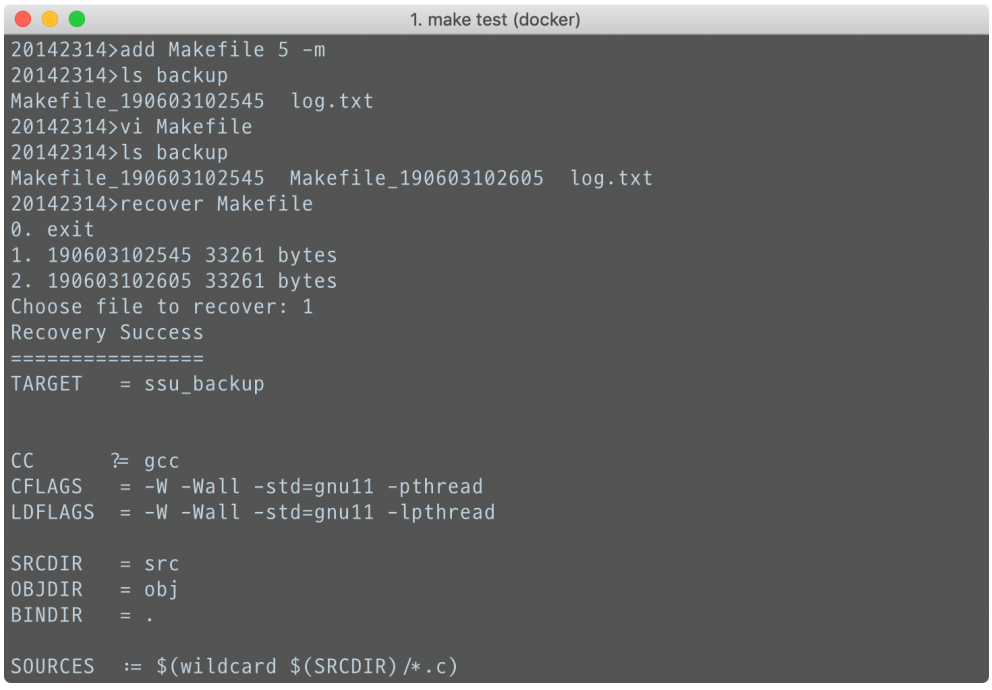

그림 8 -m 옵션 사용 후 vi 명령어를 사용해 파일을 변경한 후, recover 명령어로 파일을 복구하는 예

5. 소스코드

<main.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <sys/stat.h>

#include "backup.h"
#include "command.h"
#include "util.h"

void exit_with_print_usage(char *);

int main(int argc, char *argv[])
{
  char backup_path[PATH_MAX];
  struct stat statbuf;

  // 인자가 1개만 주어진 경우 현재 경로 밑에 디렉토리 생성
  if (argc == 1) {
    getcwd(backup_path, PATH_MAX);
  }
  // 인자가 2개 주어진 경우
  else if (argc == 2) {
    // 절대 경로 확인
    path_normalize(argv[1], backup_path);
    // 디렉토리 또는 디렉토리에 대한 권한이 없는 경우 프로그램 종료
    if (access(backup_path, F_OK | R_OK | W_OK) != 0) {
      exit_with_print_usage(argv[0]);
    }
    lstat(backup_path, &statbuf);
    // 주어진 경로가 디렉토리가 아닌 경우 프로그램 종료
    if (!S_ISDIR(statbuf.st_mode)) {
      exit_with_print_usage(argv[0]);
    }
  }
  // 3개 이상의 인자가 주어진 경우
  else {
```

```
    exit_with_print_usage(argv[0]);
}

// 하위 디렉토리 "backup"이 존재하지 않으면 생성
strcat(backup_path, "/backup");
if (access(backup_path, F_OK) != 0) {
    mkdir(backup_path, 0755);
}

system("clear");

// 백업 초기 설정
backup_init(backup_path);

while (true) {
    char buf[LINE_MAX];
    memset(buf, 0, sizeof(0));

    // 프롬프트 출력
    printf("20142314>");
    // 입력을 받는다.
    getln(buf);
    // 빈 입력 시 프롬프트로 넘어간다.
    if (strcmp(buf, "") == 0)
        continue;
    // "exit" 입력 시 종료
    else if (strcmp(buf, "exit") == 0) {
        backup_finalize();
        break;
    }
    // "list" 입력 시 리스트 출력
    else if (strcmp(buf, "list") == 0) {
        backup_print_list();
    }
    // "vi" / "vim" / "ls" 입력 시 system()을 통해 명령어 실행
    else if (strcmp(buf, "vi") == 0 || strcmp(buf, "vim") == 0 ||
                strcmp(buf, "ls") == 0) {
        system(buf);
    }
    // 인자 받는 경우 포함
    else if (string_starts_with(buf, "vi ") ||
                string_starts_with(buf, "vim ") ||
                string_starts_with(buf, "ls ")) {
        system(buf);
```

```
}
else {
  // 공백에 따라 파라미터 분리
  char *ret_ptr, *save_ptr;
  ret_ptr = strtok_r(buf, " ", &save_ptr);
  int _argc = 0;
  char _argv[16][512];
  memset(&_argv[0][0], 0, 16 * 512);
  while (ret_ptr != NULL) {
    if (argc >= 16)
      break;
    strcpy(_argv[_argc++], ret_ptr);
    ret_ptr = strtok_r(NULL, " ", &save_ptr);
  }
  // add 명령어
  if (strcmp(_argv[0], "add") == 0) {
    // 인자가 3개 미만인 경우 에러
    if (_argc < 3) {
      fprintf(stderr, "Usage:\n"
                      "  add <FILENAME> <PERIOD> [OPTION]\n");
      continue;
    }

    // 경로를 절대 경로로 변환
    char path[PATH_MAX];
    path_normalize(_argv[1], path);

    // period 숫자 변환
    int backup_period;
    char *end_ptr = NULL;
    backup_period = strtol(_argv[2], &end_ptr, 10);
    // 정수를 입력받지 않은 경우 에러
    if (end_ptr == _argv[2] || end_ptr[0] != '\0') {
      fprintf(stderr, "Error: period must be an integer\n");
      continue;
    }
    // 올바른 범위가 아닌 경우 에러
    if (backup_period < 5 || backup_period > 10) {
      fprintf(stderr, "Error: period range error. "
                      "period must be between 5 and 10.\n");
      continue;
    }

    // 옵션 처리
```

```
int backup_limit = -1, backup_time_to_live = -1;
bool backup_watch_modify = false, backup_dir = false,
     error_occurred = false;

for (int i = 3; i < _argc; ++i) {
  // -m 옵션 처리
  if (strcmp(_argv[i], "-m") == 0) {
    backup_watch_modify = true;
  }
  // -n 옵션 처리
  else if (strcmp(_argv[i], "-n") == 0) {
    // 파라미터를 입력받지 않은 경우 에러
    if (i == _argc - 1) {
      fprintf(stderr, "Error: -n option has no NUMBER parameter\n");
      error_occurred = true;
      break;
    }
    // 파라미터 숫자 변환
    end_ptr = NULL;
    backup_limit = strtol(_argv[i + 1], &end_ptr, 10);
    // 파라미터가 정수가 아닌 경우 에러
    if (end_ptr == _argv[i + 1] || end_ptr[0] != '\0') {
      fprintf(
          stderr,
          "Error: -n option's NUMBER parameter must be an integer\n");
      error_occurred = true;
      break;
    }
    // 파라미터가 올바른 범위가 아닌 경우 에러
    if (backup_limit < 1 || backup_limit > 100) {
      fprintf(
          stderr,
          "Error: -n option's NUMBER parameter range error. "
          "-n option's NUMBER parameter must be between 1 and 100.\n");
      error_occurred = true;
      break;
    }
    ++i;
  }
  // -t 옵션 처리
  else if (strcmp(_argv[i], "-t") == 0) {
    // 파라미터를 입력받지 않은 경우 에러
    if (i == _argc - 1) {
      fprintf(stderr, "Error: -t has no TIME parameter\n");
```

```
      error_occurred = true;
      break;
    }
    // 파라미터 숫자 변환
    end_ptr = NULL;
    backup_time_to_live = strtol(_argv[i + 1], &end_ptr, 10);
    // 파라미터가 정수가 아닌 경우 에러
    if (end_ptr == _argv[i + 1] || end_ptr[0] != '\0') {
      fprintf(stderr,
              "Error: -t option's TIME parameter must be an integer\n");
      error_occurred = true;
      break;
    }
    // 파라미터가 올바른 범위가 아닌 경우 에러
    if (backup_time_to_live < 60 || backup_time_to_live > 1200) {
      fprintf(
          stderr,
          "Error: -t option's TIME parameter range error. "
          "-t option's TIME parameter must be between 60 and 1200.\n");
      error_occurred = true;
      break;
    }
    ++i;
  }
  // -d 옵션 처리
  else if (strcmp(_argv[i], "-d") == 0) {
    backup_dir = true;
  }
  // 이외 올바르지 않은 옵션은 에러
  else {
    fprintf(stderr, "Error: unknown option \"%s\"\n", _argv[i]);
    error_occurred = true;
    break;
  }
}

// 에러가 발생하면 더 진행하지 않음
if (error_occurred)
  continue;

// 경로 길이 확인 후 초과하면 에러
if (strlen(path) > 255) {
  fprintf(stderr, "Error: path length limit exceeded.\n");
  continue;
```

```
    }

    // 파일이 존재하지 않으면 에러
    if (access(path, F_OK) != 0) {
      fprintf(stderr, "Error: file not exists.\n");
      continue;
    }

    // 일반 파일 또는 -d 옵션을 사용한 디렉토리만 백업에 추가
    lstat(path, &statbuf);
    if ((S_ISREG(statbuf.st_mode) && !backup_dir) ||
        (S_ISDIR(statbuf.st_mode) && backup_dir)) {
      cmd_add(path, backup_period, backup_dir, backup_limit,
              backup_time_to_live, backup_watch_modify);
    }
    // 이외 파일은 백업에 추가되지 않는다.
    else {
      fprintf(stderr, "Error: file type is not valid.\n");
      continue;
    }
  }
  // remove 명령어
  else if (strcmp(_argv[0], "remove") == 0) {
    // 인자가 2개 아닌 경우 에러
    if (_argc != 2) {
      fprintf(stderr, "Usage:\n"
                      "  remove <FILENAME>\n"
                      "  remove -a\n");
      continue;
    }
    // -a 옵션 입력받은 경우 백업 파일 모두 제거
    if (strcmp(_argv[1], "-a") == 0) {
      backup_remove_all();
    }
    // 이외의 경우 특정 파일만 제거
    else {
      char path[PATH_MAX];
      path_normalize(_argv[1], path);
      cmd_remove(path);
    }
  }
  // compare 명령어
  else if (strcmp(_argv[0], "compare") == 0) {
    // 인자가 3개가 아닌 경우 에러
```

```
  if (_argc != 3) {
    fprintf(stderr, "Usage:\n"
                    "  compare <FILENAME1> <FILENAME2>\n");
    continue;
  }
  // 파일 비교
  cmd_compare(_argv[1], _argv[2]);
}
// recover 명령어
else if (strcmp(_argv[0], "recover") == 0) {
  char path[PATH_MAX];
  // 인자의 개수가 올바르지 않은 경우 에러
  if (_argc < 2 || _argc > 4) {
    fprintf(stderr, "Usage:\n"
                    "  recover <FILENAME> [OPTION]\n");
    continue;
  }
  path_normalize(_argv[1], path);
  if (_argc > 2) {
    // 올바른 옵션이 아니면 에러
    if (strcmp(_argv[2], "-n") != 0) {
      fprintf(stderr, "Error: unknown option \"%s\"\n", _argv[2]);
      continue;
    }
    // 파라미터가 주어지지 않은 경우 에러
    if (_argc == 3) {
      fprintf(stderr, "Error: -n has no NEWFILE parameter\n");
      continue;
    }
    char new_path[PATH_MAX];
    path_normalize(_argv[3], new_path);
    // 파일이 이미 존재하는 경우 에러
    if (access(new_path, F_OK) == 0) {
      fprintf(stderr, "Error: file \"%s\" already exists.\n", new_path);
      continue;
    }
    // 새 파일을 생성하여 복구
    cmd_recover(path, new_path);
  }
  else {
    // 원래 파일로 복구
    cmd_recover(path, path);
  }
}
```

```
      // 알 수 없는 명령어는 에러 출력
      else {
        fprintf(stderr, "Error: command is not valid.\n");
      }
    }
  }

  return 0;
}

// Usage 출력 후 프로그램 종료
void exit_with_print_usage(char *executable)
{
  fprintf(stderr, "Usage: %s [backup directory]\n", executable);
  exit(1);
}
```

<backup.h>

```
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdarg.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <pthread.h>
#include <time.h>

#include "util.h"

// 백업된 파일의 정보를 담은 링크드 리스트 노드의 구조체
typedef struct _backup_file_node {
  time_t timestamp;
  struct _backup_file_node *next;
} backup_file_node;

// 백업한 파일의 정보를 담은 링크드 리스트 노드의 구조체
typedef struct _backup_node {
  char pathname[PATH_MAX];
  int period;
```

```
  int node_limit;
  int time_to_live;
  bool watch_modify;
  pthread_t tid;
  backup_file_node *snapshots;
  struct _backup_node *next;
} backup_node;

// 백업 초기 설정
void backup_init(const char *backup_path);
// 백업 마무리
void backup_finalize();
// 백업 로그 함수
void backup_log(const char *format, ...);
// 백업 디렉토리 반환
char *backup_get_backup_path();
// 특정 경로의 백업 파일에 대한 노드 반환
backup_node *backup_get_backup_node(const char *pathname);
// 백업 파일 추가
int backup_add(const char *pathname, int period, int node_limit,
               int time_to_live, bool watch_modify);
// 백업 파일 제거
int backup_remove(const char *pathname);
// 백업 파일 모두 제거
void backup_remove_all();
// 백업 파일 리스트 출력
void backup_print_list();
// 백업 파일의 이름 생성
void backup_get_filename(char *pathname, const char *original,
                         const time_t *time_val);
// 백업 파일이 수정되었는지 확인
bool backup_check_modified(backup_node *node);

// 백업 쓰레드가 수행하는 함수
void *backup_thread(void *arg);
```

<backup.c>

```
#include "backup.h"

// 백업 파일 정보를 저장하는 링크드 리스트
backup_node *backup_files = NULL;
// 파일이 백업될 경로
```

```
char backup_directory[PATH_MAX];
// 로그 파일의 포인터
FILE *fp_log = NULL;
// 로그 기록에 사용될 뮤텍스
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// 백업 초기 설정
void backup_init(const char *backup_path)
{
  // 백업 파일 링크드 리스트에 더미 노드를 만든다.
  backup_files = calloc(1, sizeof(backup_node));
  backup_files->next = NULL;
  // 백업 디렉토리 설정
  strcpy(backup_directory, backup_path);
  // 로그 파일 경로 설정
  char log_path[PATH_MAX];
  sprintf(log_path, "%s/log.txt", backup_directory);
  fp_log = fopen(log_path, "a");
  // 로그 버퍼 설정
  setvbuf(fp_log, NULL, _IOLBF, BUFSIZ);
}

// 백업 마무리
void backup_finalize()
{
  // 로그 플러시
  fflush(fp_log);
  // 뮤텍스 해제
  pthread_mutex_destroy(&mutex);
}

// 백업 로그 함수
void backup_log(const char *format, ...)
{
  // 순차 기록을 위해 뮤텍스 사용
  pthread_mutex_lock(&mutex);
  // 현재 시간을 형식에 맞게 출력한다.
  time_t now = time(NULL);
  char suffix[14];
  struct tm local_time;
  localtime_r(&now, &local_time);
  strftime(suffix, 14, "%y%m%d %H%M%S", &local_time);
  fprintf(fp_log, "[%s] ", suffix);
  // 인자로 받은 내용 출력
```

```
  va_list args;
  va_start(args, format);
  vfprintf(fp_log, format, args);
  va_end(args);
  fprintf(fp_log, "\n");
  //줄바꿈 해제
  pthread_mutex_unlock(&mutex);
}

// 백업 디렉토리 반환
char *backup_get_backup_path()
{
  return backup_directory;
}

// 특정 경로의 백업 파일에 대한 노드 반환
backup_node *backup_get_backup_node(const char *pathname)
{
  // 노드를 탐색하여 값을 반환한다.
  backup_node *cur = backup_files->next;
  while (cur != NULL) {
    if (strcmp(pathname, cur->pathname) == 0)
      return cur;
    cur = cur->next;
  }
  // 해당 노드가 존재하지 않으면 NULL 반환
  return NULL;
}

// 백업 파일 추가
int backup_add(const char *pathname, int period, int node_limit,
               int time_to_live, bool watch_modify)
{
  backup_node *cur = backup_files;
  // 이름이 중복되는 파일은 추가하지 않는다.
  while (cur->next != NULL) {
    char *fn1 = path_get_filename(pathname);
    char *fn2 = path_get_filename(cur->next->pathname);
    if (strcmp(fn1, fn2) == 0)
      return -1;
    cur = cur->next;
  }

  // 노드 생성
```

```
  cur->next = calloc(1, sizeof(backup_node));
  cur = cur->next;
  // 노드 값 할당
  cur->next = NULL;
  strcpy(cur->pathname, pathname);
  cur->period = period;
  cur->node_limit = node_limit;
  cur->time_to_live = time_to_live;
  cur->watch_modify = watch_modify;
  cur->snapshots = calloc(1, sizeof(backup_file_node));
  cur->snapshots->next = NULL;
  // 쓰레드 생성
  if (pthread_create(&cur->tid, NULL, backup_thread, (void *)cur) != 0) {
    fprintf(stderr, "Error: pthread_create error\n");
  }
  // 쓰레드 분리
  pthread_detach(cur->tid);
  // 로그 출력
  backup_log("%s added.", cur->pathname);
  return 0;
}

// 백업 파일 제거
int backup_remove(const char *pathname)
{
  backup_node *cur = backup_files;
  // 리스트를 순회하며 해당 노드를 찾아 제거한다.
  while (cur->next != NULL) {
    if (strcmp(pathname, cur->next->pathname) == 0) {
      // 쓰레드 중지
      pthread_cancel(cur->next->tid);
      // 현재 노드 분리
      backup_node *temp = cur->next;
      cur->next = temp->next;
      // 파일 노드 메모리 반환
      backup_file_node *temp_file = temp->snapshots;
      while (temp_file != NULL) {
        backup_file_node *next = temp_file->next;
        free(temp_file);
        temp_file = next;
      }
      // 로그 출력
      backup_log("%s removed.", temp->pathname);
      // 노드 메모리 반환
```

```
      free(temp);
      return 0;
    }
    cur = cur->next;
  }
  // 노드가 존재하지 않으면 종료한다.
  return -1;
}

// 백업 파일 모두 제거
void backup_remove_all()
{
  backup_node *cur = backup_files->next;
  // 노드 분리
  backup_files->next = NULL;
  // 모든 노드를 순회하며 제거한다.
  while (cur != NULL) {
    // 쓰레드 종료
    pthread_cancel(cur->tid);
    // 현재 노드 분리
    backup_node *temp = cur->next;
    // 파일 노드 메모리 반환
    backup_file_node *temp_file = cur->snapshots;
    while (temp_file != NULL) {
      backup_file_node *next = temp_file->next;
      free(temp_file);
      temp_file = next;
    }
    // 로그 출력
    backup_log("%s removed.", cur->pathname);
    // 노드 메모리 반환
    free(cur);
    cur = temp;
  }
}

// 백업 파일 리스트 출력
void backup_print_list()
{
  backup_node *cur = backup_files->next;
  while (cur != NULL) {
    // 경로 및 period 출력
    printf("%s %d", cur->pathname, cur->period);
    // 옵션 출력
```

```
    if (cur->watch_modify) {
      printf(" m");
    }
    if (cur->node_limit != -1) {
      printf(" n");
    }
    if (cur->time_to_live != -1) {
      printf(" t");
    }
    printf("\n");
    cur = cur->next;
  }
}

// 백업 파일의 이름 생성
void backup_get_filename(char *pathname, const char *original,
                         const time_t *time_val)
{
  // 시간을 형식에 맞게 문자열로 만든다.
  char suffix[13];
  struct tm local_time;
  localtime_r(time_val, &local_time);
  strftime(suffix, 13, "%y%m%d%H%M%S", &local_time);
  // 배열에 복사
  sprintf(pathname, "%s/%s_%s", backup_directory, path_get_filename(original),
          suffix);
}

// 백업 파일이 수정되었는지 확인
bool backup_check_modified(backup_node *node)
{
  backup_file_node *file_node = node->snapshots->next;
  // 파일이 존재하지 않는 경우 백업
  if (file_node == NULL)
    return true;
  char backup_pathname[PATH_MAX];
  backup_get_filename(backup_pathname, node->pathname, &file_node->timestamp);
  // 파일이 수정 시간이나 크기가 다른경우 백업
  return file_compare(node->pathname, backup_pathname) == 0 ? false : true;
}

// 백업 쓰레드가 수행하는 함수
void *backup_thread(void *arg)
{
```

```
backup_node *node = (backup_node *)arg;
while (true) {
  // period 만큼 대기
  sleep(node->period);

  char path[PATH_MAX];
  backup_file_node *snapshot = node->snapshots->next;

  // -m 옵션이 적용되지 않았거나, 적용되었는데 파일이 변경된 경우 백업
  if (!node->watch_modify || backup_check_modified(node)) {
    // 새 파일 노드 생성
    snapshot = calloc(1, sizeof(backup_file_node));
    snapshot->timestamp = time(NULL);
    snapshot->next = node->snapshots->next;
    node->snapshots->next = snapshot;
    // 파일 경로 생성
    backup_get_filename(path, node->pathname, &snapshot->timestamp);
    // 파일 복사
    file_copy(node->pathname, path);
    // 로그 출력
    backup_log("%s generated.", path);
  }

  backup_file_node *cur = node->snapshots;
  int n = 1;
  while (cur->next != NULL) {
    // -t 옵션이 적용 되고 특정 시간이 지난 경우
    // 또는 -n 옵션이 적용되고 파일의 개수가 초과된 경우 파일을 삭제한다.
    if ((node->time_to_live > 0 &&
         difftime(snapshot->timestamp, cur->next->timestamp) >
             node->time_to_live) ||
        (node->node_limit > 0 && n > node->node_limit)) {

      // 파일 노드 분리
      backup_file_node *tmp = cur->next;
      cur->next = tmp->next;
      // 삭제 파일 경로 생성
      char del_path[PATH_MAX];
      backup_get_filename(del_path, node->pathname, &tmp->timestamp);
      // 로그 출력
      backup_log("%s deleted.", del_path);
      // 파일 삭제
      remove(del_path);
      // 파일 노드 메모리 반환
```

```
        free(tmp);
      }
      cur = cur->next;
      ++n;
      if (cur == NULL)
        break;
    }
  }
}
```

<command.h>

```
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
#include <limits.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "backup.h"

// 백업리스트 파일 추가
void cmd_add(const char *pathname, int period, bool recursive, int node_limit,
             int time_to_live, bool watch_modify);
// 백업리스트 파일 제거
void cmd_remove(const char *pathname);
// 두 파일 비교
void cmd_compare(const char *file1, const char *file2);
// 파일 복구
void cmd_recover(const char *pathname, const char *new_pathname);
```

<command.c>

```
#include "command.h"

// 백업리스트 파일 추가
```

```
void cmd_add(const char *pathname, int period, bool recursive, int node_limit,
             int time_to_live, bool watch_modify)
{
  // -d 옵션을 통해 디렉토리가 추가된 경우
  if (recursive) {
    // 백업 경로는 백업에 포함 될 수 없다.
    if (strcmp(pathname, backup_get_backup_path()) == 0) {
      fprintf(stderr, "Error: backup path cannot be added.\n");
      return;
    }
    // 디렉토리 순회
    DIR *dirp = opendir(pathname);
    struct dirent *dirent;
    while ((dirent = readdir(dirp)) != NULL) {
      if (strcmp(dirent->d_name, ".") == 0 || strcmp(dirent->d_name, "..") == 0)
        continue;
      // 절대 경로를 가져온다.
      char abs_path[PATH_MAX];
      sprintf(abs_path, "%s/%s", pathname, dirent->d_name);
      struct stat statbuf;
      // 파일 정보를 가져온다.
      lstat(abs_path, &statbuf);
      // 일반 파일인 경우 -d 옵션을 제외하고 재귀호출
      if (S_ISREG(statbuf.st_mode)) {
        cmd_add(abs_path, period, false, node_limit, time_to_live,
                watch_modify);
      }
      // 디렉토리 파일인 경우 -d 옵션을 포함하고 재귀호출
      if (S_ISDIR(statbuf.st_mode)) {
        cmd_add(abs_path, period, true, node_limit, time_to_live, watch_modify);
      }
    }
  }
  // 일반 파일이 추가 된 경우
  else {
    if (backup_add(pathname, period, node_limit, time_to_live, watch_modify) <
        0) {
      // 중복 파일인 경우 에러 메시지 출력
      char filename[PATH_MAX];
      strcpy(filename, pathname);
      fprintf(stderr, "Error: duplicated filename \"%s\". skipping.\n",
              path_get_filename(filename));
    }
  }
```

```
}

// 백업리스트 파일 제거
void cmd_remove(const char *pathname)
{
  if (backup_remove(pathname) < 0) {
    fprintf(stderr, "Error: file \"%s\" is not exist.\n", pathname);
  }
}

// 두 파일 비교
void cmd_compare(const char *file1, const char *file2)
{
  // 파일이 존재하지 않으면 에러 메시지 출력
  if (access(file1, F_OK) != 0) {
    fprintf(stderr, "Error: file \"%s\" is not exist.\n", file1);
    return;
  }
  if (access(file2, F_OK) != 0) {
    fprintf(stderr, "Error: file \"%s\" is not exist.\n", file2);
    return;
  }
  // 두 파일 비교
  int comp = file_compare(file1, file2);
  if (comp == -1) {
    // 파일 비교 과정에서 에러 발생 시 메시지 출력
    fprintf(stderr, "Error: stat error\n");
  }
  else if (comp == 0) {
    // 파일이 같은 경우 메시지를 출력한다.
    printf("Files are equal.\n");
  }
  else {
    // 파일이 다른 경우 파일 크기와 수정 시간을 출력한다.
    printf("Files are different.\n");
    struct stat statbuf1, statbuf2;
    char mod_time1[26], mod_time2[26];
    // 파일 정보를 가져온다.
    if (lstat(file1, &statbuf1) != 0 || lstat(file2, &statbuf2) != 0) {
      fprintf(stderr, "Error: stat error\n");
      return;
    }
    // 시간을 문자열로 변환한다.
    ctime_r(&statbuf1.st_mtime, mod_time1);
```

```
    ctime_r(&statbuf2.st_mtime, mod_time2);
    // 파일 정보 출력
    printf("   %s\n"
#ifdef __APPLE__
           "     size: %lld bytes\n"
#else
           "     size: %ld bytes\n"
#endif
           "     last modified time: %s",
           file1, statbuf1.st_size, mod_time1);
    printf("   %s\n"
#ifdef __APPLE__
           "     size: %lld bytes\n"
#else
           "     size: %ld bytes\n"
#endif
           "     last modified time: %s",
           file2, statbuf2.st_size, mod_time2);
  }
}

// 파일 복구
void cmd_recover(const char *pathname, const char *new_pathname)
{
  // 주어진 경로에 대한 노드를 가져온다.
  backup_node *node = backup_get_backup_node(pathname);
  // 노드가 존재하지 않는 경우 에러 메시지 출력 후 종료
  if (node == NULL) {
    fprintf(stderr, "Error: filename \"%s\" not found.\n", pathname);
    return;
  }
  // 리스트를 만들어 오름차순 출력
  backup_file_node *list = NULL, *cur = node->snapshots->next;
  while (cur != NULL) {
    backup_file_node *tmp = calloc(1, sizeof(backup_file_node));
    tmp->timestamp = cur->timestamp;
    tmp->next = list;
    list = tmp;
    cur = cur->next;
  }
  printf("0. exit\n");
  int index = 0;
  cur = list;
  while (cur != NULL) {
```

```
  char suffix[13];
  struct tm local_time;
  // 시간 포맷팅
  localtime_r(&cur->timestamp, &local_time);
  strftime(suffix, 13, "%y%m%d%H%M%S", &local_time);
  // 파일 크기를 가져온다.
  char backup_pathname[PATH_MAX];
  backup_get_filename(backup_pathname, pathname, &cur->timestamp);
  struct stat statbuf;
  lstat(backup_pathname, &statbuf);
  printf("%d. %s\t%d bytes\n", ++index, suffix, statbuf.st_mode);
  cur = cur->next;
}
// 복구할 파일에 대한 입력을 받는다.
char buf[LINE_MAX];
memset(buf, 0, sizeof(0));
printf("Choose file to recover: ");
getln(buf);

// 받은 입력을 숫자로 전환
int file_id;
char *end_ptr = NULL;
file_id = strtol(buf, &end_ptr, 10);
// 입력받은 숫자가 올바른 숫자가 아닌경우
if (end_ptr == buf || end_ptr[0] != '\0') {
  fprintf(stderr, "Error: file id must be an integer.\n");
}
// 입력 받은 숫자가 범위를 초과한 경우
else if (file_id < 0 || file_id > index) {
  fprintf(stderr, "Error: file id is not valid.\n");
}
// 입력 받은 숫자가 exit이 아닌경우
else if (file_id != 0) {
  // 해당 노드를 가져온다.
  cur = list;
  index = 1;
  while (index < file_id) {
    ++index;
    cur = cur->next;
  }
  if (cur != NULL) {
    // 파일 이름을 가져온다.
    char backup_pathname[PATH_MAX];
    backup_get_filename(backup_pathname, pathname, &cur->timestamp);
```

```
      // 파일 복사에 성공한 경우
      if (file_copy(backup_pathname, new_pathname) != -1) {
        // 로그 출력
        backup_log("%s recovered.", pathname);
        // 백업 중단
        backup_remove(pathname);
        // 성공 메시지 출력
        printf("Recovery Success\n");
        printf("===============\n");
        // 파일 내용 출력
        FILE *fp = fopen(new_pathname, "r");
        char file_buf[4096];
        while (fgets(file_buf, 4096, fp)) {
          fputs(file_buf, stdout);
        }
      }
      // 도중에 -n, -t 옵션에 의해 파일이 삭제된 경우
      else {
        // 에러 메시지 출력
        fprintf(stderr, "Error: backup file not exist. file copy failed.\n");
      }
    }
    else {
      // 해당 id에 대한 노드가 없는 경우 에러 메시지 출력
      fprintf(stderr, "Error: file id is not valid.\n");
    }
  }
  cur = list;
  // 사용한 리스트 노드 제거
  while (cur != NULL) {
    backup_file_node *tmp = cur;
    cur = cur->next;
    free(tmp);
  }
}
```

<util.h>

```
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <utime.h>
#include <pwd.h>
#include <sys/stat.h>
#include <sys/types.h>

// 표준 입력 함수
void getln(char *buf);

// 문자열이 특정 문자열로 시작하는지 검사
bool string_starts_with(const char *str, const char *prefix);

// 두 파일 비교
int file_compare(const char *file1, const char *file2);
// 파일 복사
int file_copy(const char *src, const char *dest);

// 파일의 절대 경로를 구함
void path_normalize(const char *path, char *normalized_path);
// 경로에서 파일 이름을 구함
char *path_get_filename(const char *path);
```

<util.c>

```
#include "util.h"

// 표준 입력 함수
void getln(char *buf)
{
  // fgets 사용 후, 마지막 줄바꿈 문자를 제거한다.
  fgets(buf, LINE_MAX, stdin);
  buf[strlen(buf) - 1] = '\0';
}

// 문자열이 특정 문자열로 시작하는지 검사
bool string_starts_with(const char *str, const char *prefix)
{
  // str이 prefix보다 길이가 더 짧은 경우 항상 거짓
  if (strlen(str) < strlen(prefix))
    return false;
```

```
  // 이외의 경우 prefix길이만큼 비교
  return strncmp(str, prefix, strlen(prefix)) == 0;
}

// 두 파일 비교
int file_compare(const char *file1, const char *file2)
{
  // 두 파일의 정보를 가져온다.
  struct stat statbuf1, statbuf2;
  // stat 에러 발생 시 -1 반환
  if (lstat(file1, &statbuf1) != 0 || lstat(file2, &statbuf2) != 0) {
    return -1;
  }
  // 두 파일의 크기 및 수정 시간 비교
  if (statbuf1.st_size == statbuf2.st_size &&
      statbuf1.st_mtime == statbuf2.st_mtime) {
    return 0;
  }
  else {
    return 1;
  }
}

// 파일 복사
int file_copy(const char *src, const char *dest)
{
  // 파일이 존재하지 않거나 권한이 없는 경우 -1 반환
  if (access(src, F_OK | R_OK) != 0)
    return -1;

  // 파일 정보를 가져온다.
  struct stat statbuf;
  lstat(src, &statbuf);

  int fd_src, fd_dest;
  char buf[4096];

  // 파일 오픈 후 에러 발생시 -1 반환
  if ((fd_src = open(src, O_RDONLY)) < 0) {
    return -1;
  }
  if ((fd_dest = open(dest, O_WRONLY | O_CREAT | O_TRUNC, statbuf.st_mode)) <
      0) {
    return -1;
```

```
  }

  // 내용을 버퍼 크기만큼 읽어서 복사한다.
  ssize_t len;
  while ((len = read(fd_src, buf, sizeof(buf))) > 0) {
    write(fd_dest, buf, len);
  }

  // 파일을 닫는다.
  close(fd_src);
  close(fd_dest);

  // 시간 정보 변경
  struct utimbuf time_buf;
  time_buf.actime = statbuf.st_atime;
  time_buf.modtime = statbuf.st_mtime;
  utime(dest, &time_buf);

  // 소유권자 변경
  chown(dest, statbuf.st_uid, statbuf.st_gid);

  return 0;
}

// 파일의 절대 경로를 구함
void path_normalize(const char *path, char *normalized_path)
{
  // "~"를 경로에 포함하는 경우
  if (path[0] == '~') {
    strcpy(normalized_path, getpwuid(getuid())->pw_dir);
    strcat(normalized_path, path + 1);
  }
  // 이외의 경우 realpath를 사용하여 절대 경로를 구한다.
  else {
    realpath(path, normalized_path);
  }
}

// 경로에서 파일 이름을 구함
char *path_get_filename(const char *path)
{
  // 마지막 '/'위치를 구한다.
  char *ptr = strrchr(path, '/');
  if (ptr)
```

```
    // '/' 문자가 존재하는 경우 마지막 '/' 문자의 다음 위치 반환
    return ptr + 1;
  else
    // '/' 문자가 존재하지 않는 경우 경로 전체 반환
    return (char *)path;
}
```