컴퓨터학부 20211407 김지호

1. 개요

본 프로그램은 리눅스 환경에서 사용자가 지정한 디렉토리를 지속적으로 모니터링하여, 새로 생성되거나 변경된 파일을 자동으로 분류·정리하는 백그라운드 데몬 프로세스이다.

2. 기능 및 프로토타입

2-1. main.c

2-1-1. 개요

프로그램 진입점. 초기 설정 수행 후 사용자 명령을 처리할 command_loop()를 실행한다.

2-1-2. 함수 프로토타입 및 설명

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

- command_loop() 실행하여 사용자 입력 대기

2-2. command.c

2-2-1. 개요

표준 입력으로부터 문자열 형태의 명령어를 받아 파싱하고, 각 명령별 처리 함수를 호출한다.

- 지원 함수
  add <DIR> [OPTIONS]: 모니터링 추가
  show <PATH> : 등록된 데몬 목록 및 상세 정보
  modify <DIR> [OPTIONS]: 기존 설정 수정
  remove <DIR>: 모니터링 해제
  help: 도움말
  exit: 프로그램 종료

2-2-2. 함수 프로토타입 및 설명

- void command_loop(void);
  무한 루프를 돌며 fgets()로 입력 문자열 획득
  command_process() 호출 후 프롬프트 재출력
- static void command_process(const char *str);
  입력 문자열(str)을 첫 공백까지 잘라 명령어(cmd) 추출
  나머지 문자열(args)을 분리하여 각 command_* 함수 호출
- int command_add(const char *args);
  파라미터: 등록할 디렉토리 경로와 옵션이 포함된 문자열
  strdup() 후 strtok()로 기본 경로와 옵션 파싱
  경로 유효성 확인(stat, access, 홈 디렉토리 내 포함 여부 등)
  설정 파일(ssu_cleanupd.config) 생성: write_config_file() 호출
  daemon_add() 호출하여 부모 프로세스에 메타데이터 등록 및 자식 데몬 생성
- int command_show(void);
  daemon_list_all()로 등록된 데몬 목록 획득
  사용자 선택에 따라 해당 데몬 설정 파일(ssu_cleanupd.config) 파싱 및 로그(ssu_cleanupd.log) 출력
- int command_modify(const char *args);
  파라미터: 수정 대상 디렉토리 및 옵션 문자열
  대상 디렉토리 확인 및 기존 설정 로드(parse_config)
  -d, -i, -l, -x, -e, -m 옵션 처리 (기존 command_add와 유사)
  설정 파일 덮어쓰기(write_config_file)
  daemon_reload(pid) 호출하여 SIGHUP 전송 (데몬에 재로딩 신호)
- int command_remove(const char *args);
  파라미터: 해제 대상 디렉토리 문자열
  realpath()로 절대 경로 변환
  daemon_list_all()로 일치하는 데몬 검색

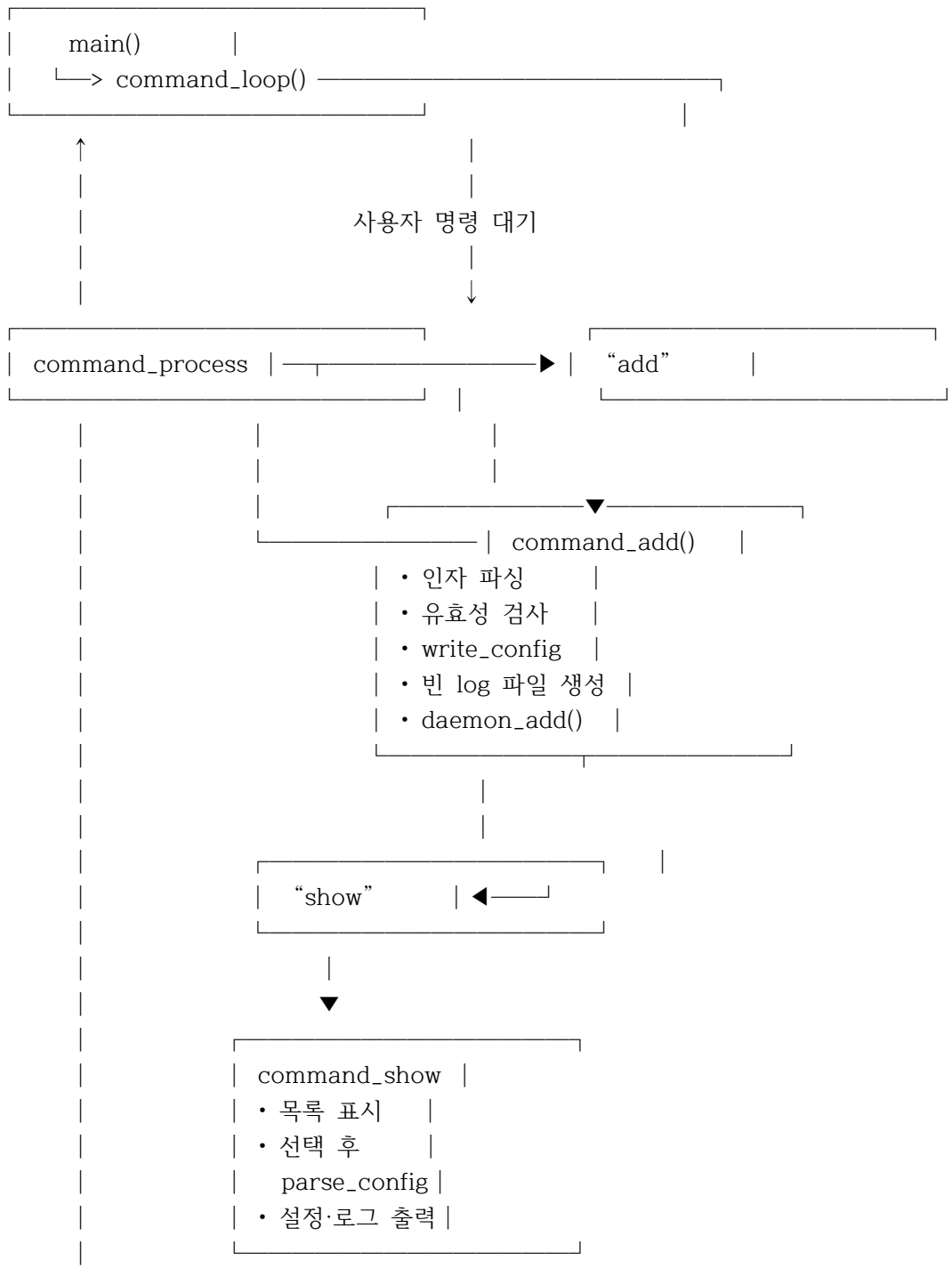
  daemon_remove(pid) 호출 (SIGTERM 전송 후 목록에서 제거)
- int write_config_file(const char *path, const char *monitoring_path, const char *output_path, long time_interval, size_t max_log_lines, char **exclude_paths, int exclude_count, const char *extensions, int mode);
  설정 파일 생성 및 각 필드 기록
2-3. daemon.c
2-3-1. 개요
command_add()로 포크된 자식 프로세스에서 데몬화(daemonize_process) 수행하며, 백그라운드에서 주기적으로 디렉토리 정리 및 로그 기록
2-3-2. 함수 프로토타입 및 설명
2-3-2. 함수 프로토타입 및 상세 설명
- int daemon_add(const char *monitor_path, const char *output_path, const char *config_file, long interval, size_t max_logs, char **exclude_paths, int exclude_count, const char *extensions, int mode);
  부모: daemon_t 구조체에 PID, 경로, 설정 정보 저장
  자식: daemonize_process() → monitor_loop() 진입
- static void daemonize_process(void);
  이중 포크, 세션 분리, 표준 입출력 /dev/null 리다이렉션, 작업 디렉토리 / 변경
- static void monitor_loop(daemon_t *d);
  무한 루프:
   parse_config(config_file, &cfg)로 최신 설정 로드
   arrange_directory(monitoring_path, output_path, exclude_paths, exclude_count, extensions, mode) 호출하여 실제 파일 정리
   log_event(&cfg, src, dst)로 변경된 파일 이동 기록 (최초 실행 제외 옵션)
   nanosleep()으로 대기
- int parse_config(const char *config_file, daemon_config_t *cfg);
  설정 파일 파싱: PID, start_time, time_interval, exclude_path, extension, mode 등 필드 로드
- int log_event(const daemon_config_t *cfg, const char *src_path, const char *dst_path);
  현재 시간(hh:mm:ss), cfg->pid, 소스 및 대상 경로 기록
  로그 최대 줄 수(max_log_lines) 초과 시 오래된 항목 삭제
- 기타 유틸리티 함수:
  int daemon_remove(pid_t pid); (SIGTERM)
  int daemon_reload(pid_t pid); (SIGHUP)
  int daemon_list_all(daemon_t ***out_list); (메타데이터 조회)
2-4. header.h
- daemon_t
```
typedef struct {
    pid_t pid;
    char *monitoring_path;
    char *output_path;
    char *config_file;
    daemon_config_t cfg;
} daemon_t;
```
부모(메인) 프로세스가 관리하는, 등록된 각 데몬 인스턴스의 메타데이터와 마지막 설정 상태를 저장한다.
command show 목록 출력, remove/modify 명령 처리 시 이 배열을 탐색해 대상 프로세스를 찾는다.
```
typedef struct {
    long time_interval;
    size_t max_log_lines;
    char **exclude_paths;
```

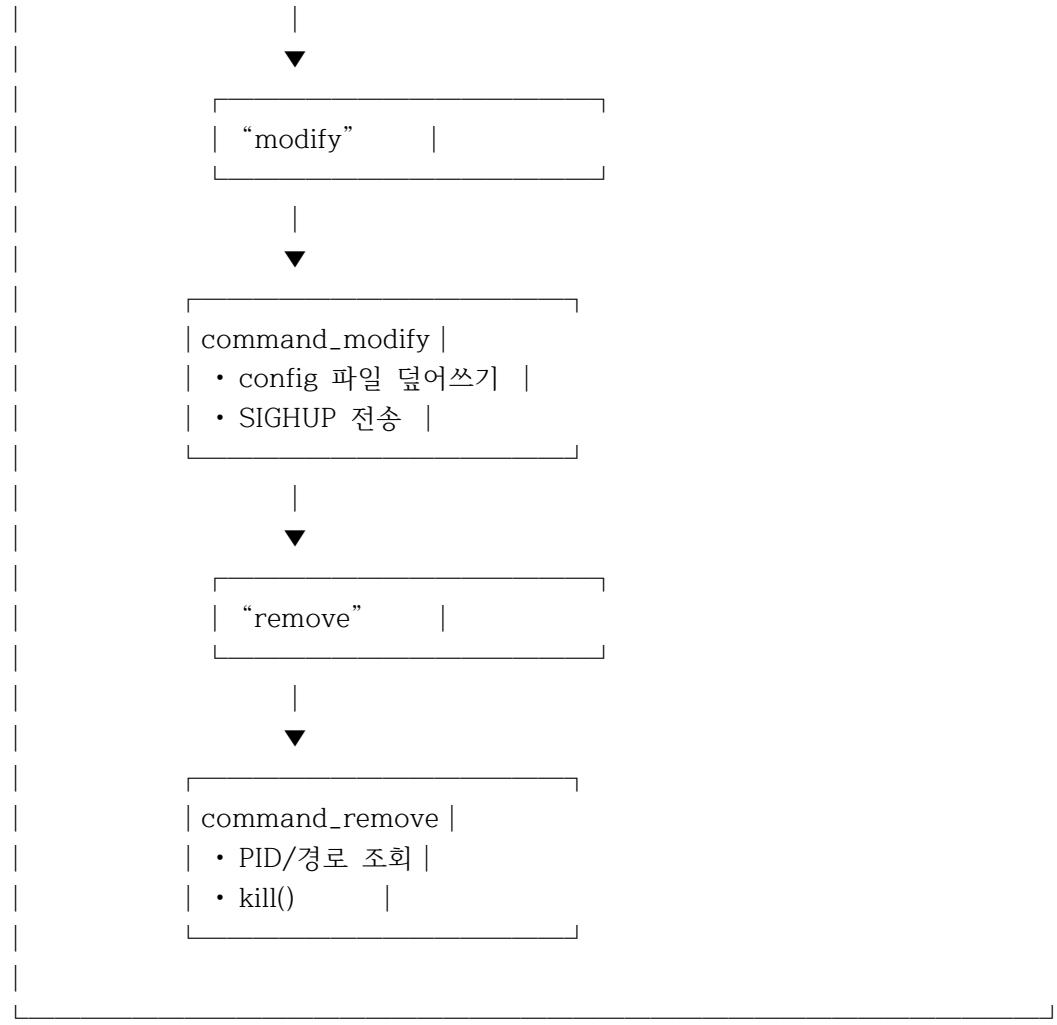
```
    int exclude_count;
    char *extensions;
    int ext_count;
    int mode;
    pid_t pid;
    char *start_time;
} daemon_config_t;
```

개별 데몬 프로세스가 동작하는 데 필요한 모든 설정값을 저장한다.
설정 파일(ssu_cleanupd.config)을 파싱해 이 구조체에 채워넣고, 이후 모니터링·정리·로그 작성 시 참조한다.

3. 상세설계

3-1. flow chart





3-2. 전체 flow

3-2-1. 프로그램 시작 (main.c)

main() 에서 command_loop() 진입.

- 명령 처리 루프 (command_loop())

  프롬프트 출력 후 fgets() 로 사용자 입력 대기.

  입력 문자열을 command_process() 로 전달.

- 명령 파싱 (command_process())

  첫 번째 토큰(cmd)으로 명령어 판별: show, add, modify, remove, exit 등.

  나머지 문자열 전체를 args 로 넘겨주어, 공백·탭 단위가 아닌 "명령어 뒤 전체"를 보존.

3-2-2. add 명령 흐름 (command_add())

args 를 strdup() 한 뒤 strtok(..., " \t") 으로 첫 토큰인 모니터링 경로 추출.

이후 반복문으로 -d, -i, -l, -x, -e, -m 옵션과 인자를 차례로 파싱·검증.

- config 파일 생성

  write_config_file() 호출 : ssu_cleanupd.config 에 설정 값 기록

- ssu_cleanupd.log 파일 생성

- 데몬 생성

  daemon_add(): fork()

  부모: 전역 리스트에 daemon_t 등록

  자식: daemonize_process() 후 monitor_loop() 진입

3-2-3. show 명령 흐름 (command_show())

  등록된 데몬 리스트 출력 → 사용자 선택

해당 daemon_t 의 config_file 을 parse_config() 로 읽어와
설정 내용과 ssu_cleanupd.log 파일 내용 출력

3-2-4. modify 명령 흐름 (command_modify())

첫 토큰으로 모니터링 경로 파악 → realpath() 검증

등록 리스트에서 일치하는 데몬 찾음 → d->cfg 초기화

-d, -i, -l, -x, -e, -m 옵션 재파싱

write_config_file() 로 덮어쓰기 → kill(d->pid, SIGHUP)

3-2-5. remove 명령 흐름 (command_remove())

args 경로를 realpath() 로 절대경로 변환

등록 리스트에서 일치하는 daemon_t 찾아 daemon_remove(pid)

kill(pid, SIGTERM) → waitpid() → 리스트에서 해제

4. 실행결과

4-1. ssu_cleanupd

4-2. show

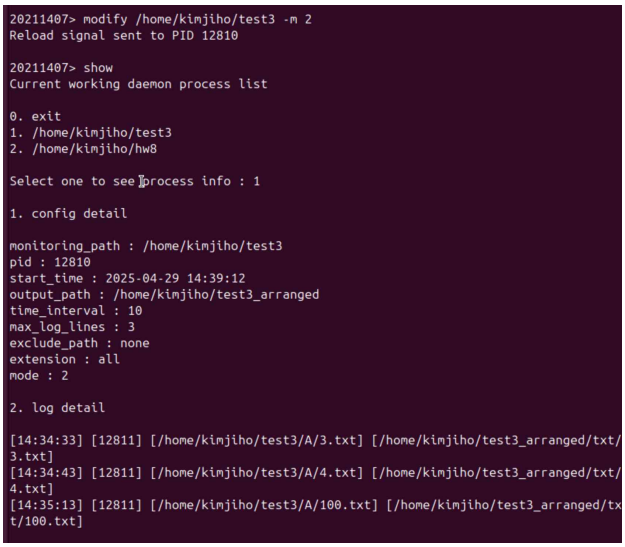
```
20211407> show
Current working daemon process list

0. exit
1. /home/kimjiho/test3
2. /home/kimjiho/hw8

Select one to see process info : remove /home/kimjiho/hw8

20211407> show
Current working daemon process list

0. exit
1. /home/kimjiho/test3
2. /home/kimjiho/hw8

Select one to see process info : 0

20211407> remove /home/kimjiho/hw8
Daemon watching /home/kimjiho/hw8 removed (PID 12853)
```

4-3. add

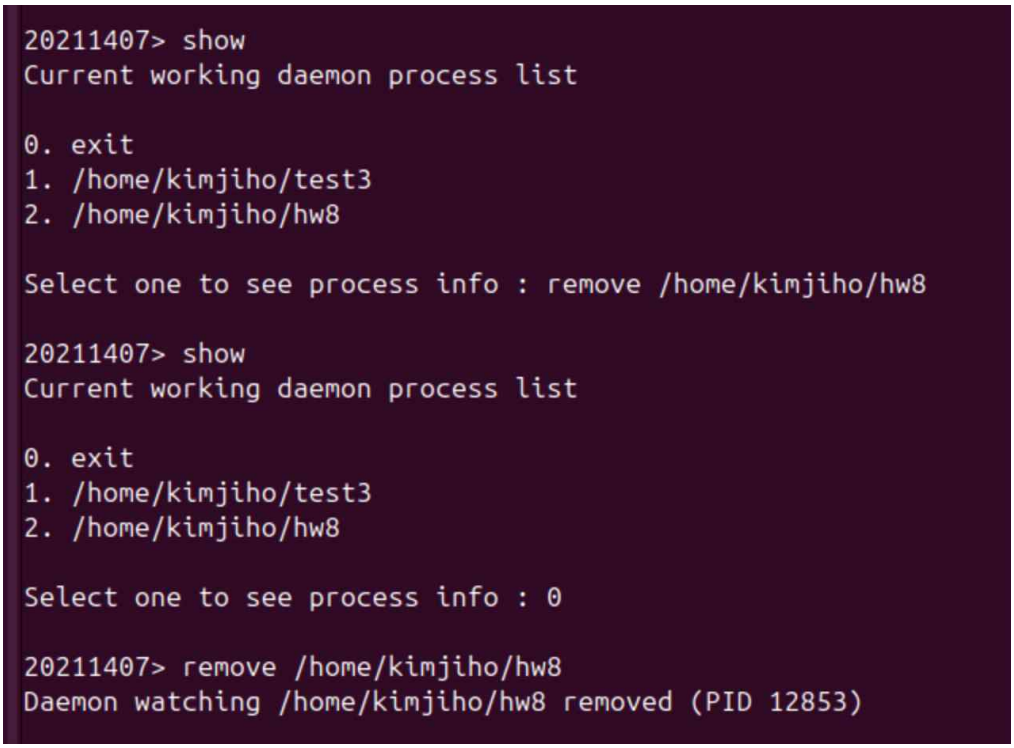
```
kimjiho@kimjiho:~/hw_p2$ ./ssu_cleanupd
20211407> add /home/kimjiho/test3 -i 10 -l 3

20211407> show
Current working daemon process list

0. exit
1. /home/kimjiho/test3

Select one to see process info : 1

1. config detail

monitoring_path : /home/kimjiho/test3
pid : 12810
start_time : 2025-04-29 14:33:03
output_path : /home/kimjiho/test3_arranged
time_interval : 10
max_log_lines : 3
exclude_path : none
extension : all
mode : 1

2. log detail
```

```
kimjiho@kimjiho:~$ tree /home/kimjiho/test3
/home/kimjiho/test3
├── A
└── A_ouput

3 directories, 0 files
kimjiho@kimjiho:~$ cd /home/kimjiho/test3/A
kimjiho@kimjiho:~/test3/A$ touch 1.txt
kimjiho@kimjiho:~/test3/A$ touch 2.txt
kimjiho@kimjiho:~/test3/A$ touch 3.txt
kimjiho@kimjiho:~/test3/A$ touch 4.txt
kimjiho@kimjiho:~/test3/A$ touch 100.txt
kimjiho@kimjiho:~/test3/A$
```

4-4. modify



4-5. remove



4-6. help

```
kimjiho@kimjiho:~/hw_p2$ ./ssu_cleanupd
20211407> help
Usage:
  > show
    <none> : show monitoring daemon process info

  > add <DIR_PATH> [OPTION]...
    <none> : add daemon process monitoring the <DIR_PATH> directory
    -d <OUTPUT_PATH> : Specify the output directory where <DIR_PATH> will be arrang
ed
    -i <TIME_INTERVAL> : Set the time interval (in seconds) for the daemon process
monitor
    -l <MAX_LOG_LINES> : Set the maximum number of log lines
    -x <EXCLUDE_PATH1,EXCLUDE_PATH2,...> : Exclude directories
    -e <EXTENSION1,EXTENSION2,...> : Specify file extensions for organization
    -m <M> : Specify the duplicate file handling mode (1~3)

  > modify <DIR_PATH> [OPTION]...
    <none> : modify daemon process config

  > remove <DIR_PATH>
    <none> : remove daemon process monitoring the <DIR_PATH> directory

  > help
    Display this help information

  > exit
    Exit the program
```

4-7. exit

```
kimjiho@kimjiho:~/hw_p2$ ./ssu_cleanupd
20211407> exit
```

5. 소스코드

5-1. main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "header.h"

int main(void)
{
    // start prompt
    command_loop();

    // end of program
    return 0;
}
```

5-2. arrange.c

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

```
#include <utime.h>
#include <limits.h>
#include "header.h"  // prototypes for arrange_directory, log_event

extern daemon_config_t *current_cfg;
extern bool current_first_run;

// Helper: get file extension (without dot), or empty string if none
static const char* get_extension(const char *filename) {
    const char *dot = strrchr(filename, '.');
    if (!dot || dot == filename) return "";
    return dot + 1;
}

// Helper: copy file from src to dst, return 0 on success
static int copy_file(const char *src, const char *dst, mode_t mode) {
    int infd = open(src, O_RDONLY);
    if (infd < 0) return -1;
    int outfd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, mode);
    if (outfd < 0) { close(infd); return -1; }
    char buf[8192];
    ssize_t n;
    while ((n = read(infd, buf, sizeof(buf))) > 0) {
        char *p = buf;
        while (n > 0) {
            ssize_t written = write(outfd, p, n);
            if (written < 0) { close(infd); close(outfd); return -1; }
            n -= written;
            p += written;
        }
    }
    close(infd);
    close(outfd);
    return 0;
}

// Recursive scan and arrange
static void scan_dir(const char *base_src,
                     const char *curr_src,
                     const char *dst,
                     char **exclude_paths,
                     int exclude_count,
                     char **ext_list,
                     int ext_count,
                     int all_ext,
                     int mode) {
    DIR *dir = opendir(curr_src);
    if (!dir) return;
```

```
    struct dirent *entry;
    while ((entry = readdir(dir))) {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0 || strcmp(entry->d_name,
"ssu_cleanupd.config") == 0 || strcmp(entry->d_name, "ssu_cleanud.log") == 0)
            continue;
        // Build full source path
        char src_path[PATH_MAX];
        snprintf(src_path, sizeof(src_path), "%s/%s", curr_src, entry->d_name);
        struct stat st;
        if (lstat(src_path, &st) < 0) continue;
        if (S_ISDIR(st.st_mode)) {
            // Check exclude paths
            int excluded = 0;
            for (int i = 0; i < exclude_count; i++) {
                if (strncmp(src_path, exclude_paths[i], strlen(exclude_paths[i])) == 0) {
                    excluded = 1;
                    break;
                }
            }
            if (!excluded) {
                scan_dir(base_src, src_path, dst,
                         exclude_paths, exclude_count,
                         ext_list, ext_count, all_ext, mode);
            }
        }
        else if (S_ISREG(st.st_mode)) {
            // Extension filter
            const char *ext = get_extension(entry->d_name);
            int match = all_ext;

            const char *ext0 = get_extension(entry->d_name);
            if(strcmp(ext0,"log") == 0) continue;

            if (!all_ext) {
                for (int i = 0; i < ext_count; i++) {
                    if (strcmp(ext_list[i], ext) == 0) {
                        match = 1; break;
                    }
                }
            }
            if (!match) continue;
            // Prepare dest directory for this extension
            char ext_dirname[NAME_MAX + 1];
            if (ext[0] != '\0') snprintf(ext_dirname, sizeof(ext_dirname), "%s", ext);
            else strncpy(ext_dirname, "others", sizeof(ext_dirname));
            char dest_dir[PATH_MAX];
            snprintf(dest_dir, sizeof(dest_dir), "%s/%s", dst, ext_dirname);
            if (mkdir(dest_dir, 0755) < 0 && errno != EEXIST) continue;
```

```
            // Prepare dest file path
            char dest_path[PATH_MAX];
            snprintf(dest_path, sizeof(dest_path), "%s/%s", dest_dir, entry->d_name);
            // Duplicate handling
            int do_copy = 1;
            struct stat dst_st;
            if (stat(dest_path, &dst_st) == 0) {
                if (mode == 1) {
                    // Keep newest
                    if (st.st_mtime <= dst_st.st_mtime) do_copy = 0;
                } else if (mode == 2) {
                    // Keep oldest
                    if (st.st_mtime >= dst_st.st_mtime) do_copy = 0;
                } else if (mode == 3) {
                    // Skip duplicates
                    do_copy = 0;
                }
            }
            if (!do_copy) continue;
            // Copy file
            if (copy_file(src_path, dest_path, st.st_mode & 0777) < 0) continue;
            // Preserve modification time
            struct utimbuf times = { st.st_atime, st.st_mtime };
            utime(dest_path, &times);
            if(!current_first_run) log_event(current_cfg, src_path, dest_path);
        }
    }
    closedir(dir);
}

/**
 * Arrange files in 'src' into subdirs under 'dst' by extension
 */
void arrange_directory(const char *src,
                       const char *dst,
                       char **exclude_paths,
                       int exclude_count,
                       const char *extensions_str,
                       int mode) {
    // Parse extensions list
    char **ext_list = NULL;
    int ext_count = 0;
    int all_ext = 0;
    if (!extensions_str || strlen(extensions_str) == 0) {
        all_ext = 1;
    } else {
        char *exts = strdup(extensions_str);
        char *token = strtok(exts, ",");
```

```
        while (token) {
            ext_list = realloc(ext_list, sizeof(char*) * (ext_count + 1));
            ext_list[ext_count++] = strdup(token);
            token = strtok(NULL, ",");
        }
        free(exts);
        if (ext_count == 0) all_ext = 1;
    }
    // Ensure base output exists
    mkdir(dst, 0755);
    // Recursive scan and arrange
    scan_dir(src, src, dst,
             exclude_paths, exclude_count,
             ext_list, ext_count,
             all_ext, mode);
    // Clean up ext_list
    for (int i = 0; i < ext_count; i++) free(ext_list[i]);
    free(ext_list);
}
```

5-3. command.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "header.h"
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <ctype.h>

/* Internal function */
static void command_process(const char *str)
{
    char buffer[MAX_COMMAND];
    strncpy(buffer, str, sizeof(buffer));
    buffer[sizeof(buffer)-1] = '\0';

    char *cmd = strtok(buffer, " \t\n");
    if (!cmd) return;

    const char *p = str + strlen(cmd);
    while (*p == ' ' || *p == '\t') p++;
    const char *args = *p ? p : NULL;

    if (strcmp(cmd, "show") == 0) {
        command_show();
    }
    else if (strcmp(cmd, "add") == 0) {
```

```
        command_add(args);
    }
    else if (strcmp(cmd, "modify") == 0) {
        command_modify(args);
    }
    else if (strcmp(cmd, "remove") == 0) {
        command_remove(args);
    }
    else if (strcmp(cmd, "help") == 0) {
        command_help();
    }
    else if (strcmp(cmd, "exit") == 0) {
        exit(0);
    }
    else {
        command_help();
    }
}

/* Command Loop */
void command_loop(void)
{
    char input[MAX_COMMAND];

    printf("20211407> ");
    while(fgets(input, MAX_COMMAND, stdin)){
        input[strcspn(input, "\n")] = '\0';
        if(strlen(input) > 0) command_process(input);

        printf("\n20211407> ");
    }

    return;
}

/* Command */

// 1. help
// print usage
void command_help() {
    printf("Usage:\n");
    printf("  > show\n");
    printf("    <none> : show monitoring daemon process info\n\n");
    printf("  > add <DIR_PATH> [OPTION]...\n");
    printf("    <none> : add daemon process monitoring the <DIR_PATH> directory\n");
    printf("    -d <OUTPUT_PATH> : Specify the output directory where <DIR_PATH> will be arranged\n");
    printf("    -i <TIME_INTERVAL> : Set the time interval (in seconds) for the daemon process monitor\n");
    printf("    -l <MAX_LOG_LINES> : Set the maximum number of log lines\n");
```

```
    printf("    -x <EXCLUDE_PATH1,EXCLUDE_PATH2,...> : Exclude directories\n");
    printf("    -e <EXTENSION1,EXTENSION2,...> : Specify file extensions for organization\n");
    printf("    -m <M> : Specify the duplicate file handling mode (1~3)\n\n");
    printf("  > modify <DIR_PATH> [OPTION]...\n");
    printf("    <none> : modify daemon process config\n\n");
    printf("  > remove <DIR_PATH>\n");
    printf("    <none> : remove daemon process monitoring the <DIR_PATH> directory\n\n");
    printf("  > help\n");
    printf("    Display this help information\n\n");
    printf("  > exit\n");
    printf("    Exit the program\n");
}

// 2. add
// --- Helper functions for validation ---
static int path_exists_and_dir(const char *path) {
    struct stat st;
    return stat(path, &st) == 0 && S_ISDIR(st.st_mode);
}

static int has_read_access(const char *path) {
    return access(path, R_OK) == 0;
}

static const char* get_home_dir(void) {
    struct passwd *pw = getpwuid(getuid());
    return pw ? pw->pw_dir : NULL;
}

static int is_within_home(const char *path) {
    char resolved[PATH_MAX];
    if (!realpath(path, resolved)) return 0;
    const char *home = get_home_dir();
    size_t len = strlen(home);
    return strncmp(resolved, home, len) == 0;
}

static int is_subpath(const char *parent, const char *child) {
    char rp[PATH_MAX], rc[PATH_MAX];
    if (!realpath(parent, rp) || !realpath(child, rc)) return 0;
    size_t lp = strlen(rp);
    return strncmp(rp, rc, lp) == 0 && (rc[lp] == '/' || rc[lp] == '\0');
}

int command_add(const char *args) {
    // settind
    char *buf = strdup(args);
    if (!buf) return -1;
```

```
char *tok = strtok(buf, " \t");
if (!tok) {
    fprintf(stderr, "Usage: add <DIR_PATH> [OPTIONS]\n");
    free(buf);
    return -1;
}

// raw inputs
char src_raw[PATH_MAX];
char output_raw[PATH_MAX] = "";
long interval = 10;                 // time_interval
size_t max_logs = 10;                 // max_log_lines
int mode = 1;                         // mode
// exclude paths: collect multiple
char **excl_list = NULL;
int excl_count = 0;
// extensions comma-separated
char *extensions = NULL;

strncpy(src_raw, tok, PATH_MAX);

// options
while ((tok = strtok(NULL, " \t")) != NULL) {
    if (tok[0] != '-' || strlen(tok) != 2) {
        fprintf(stderr, "Unknown option or token: %s\n", tok);
        free(buf);
        return -1;
    }
    char opt = tok[1];
    if (opt == 'd') {
        char *arg = strtok(NULL, " \t");
        if (!arg){
            fprintf(stderr, "Option -d requires a path\n");
            free(buf);
            return -1;
        }
        // cannot be subpath of src (will set after src realpath)
        strncpy(output_raw, arg, PATH_MAX);
    }
    else if (opt == 'i') {
        char *arg = strtok(NULL, " \t");
        if (!arg || !isdigit(arg[0])) {
            fprintf(stderr, "Option -i requires a natural number\n");
            free(buf);
            return -1;
        }
        interval = atol(arg);
        if (interval <= 0) {
```

```
            fprintf(stderr, "Invalid interval: %s\n", arg);
            free(buf);
            return -1;
        }
    }
    else if (opt == 'l') {
        char *arg = strtok(NULL, " \t");
        if (!arg || !isdigit(arg[0])) {
            fprintf(stderr, "Option -l requires a natural number\n");
            free(buf);
            return -1;
        }
        max_logs = (size_t)atoi(arg);
    }
    else if (opt == 'm') {
        char *arg = strtok(NULL, " \t");
        if (!arg || !isdigit(arg[0])) {
            fprintf(stderr, "Option -m requires a number 1-3\n");
            free(buf);
            return -1;
        }
        mode = atoi(arg);
        if (mode < 1 || mode > 3) {
            fprintf(stderr, "Mode must be 1,2 or 3\n");
            free(buf);
            return -1;
        }
    }
     else if (opt == 'x') {
        char *arg;
        while ((arg = strtok(NULL, " \t")) != NULL && !(arg[0] == '-' && strlen(arg) >= 2)) {
            if (!path_exists_and_dir(arg) || !has_read_access(arg)) {
                fprintf(stderr, "Error: %s is not accessible directory\n", arg);
                free(buf);
                return -1;
            }
            excl_list = realloc(excl_list, sizeof(char*) * (excl_count + 1));
            excl_list[excl_count++] = strdup(arg);
        }
        if (arg && arg[0] == '-' && strlen(arg) >= 2) {
            size_t len = strlen(arg);
            memmove(buf, arg, len + 1);
        }
    }
    else if (opt == 'e') {
        char *arg = strtok(NULL, " \t");
        if (!arg) { fprintf(stderr, "Option -e requires extensions\n"); free(buf); return -1; }
        extensions = strdup(arg);
```

```
        }
        else {
            fprintf(stderr, "Unknown option -%c\n", opt);
            free(buf); return -1;
        }
    }
    free(buf);

    // source path process
    char real_src[PATH_MAX];
    if (!realpath(src_raw, real_src)) {
        fprintf(stderr, "Error: %s is not valid path\n", src_raw);
        return -1;
    }
    if (!path_exists_and_dir(real_src) || !has_read_access(real_src)) {
        fprintf(stderr, "Error: %s is not accessible directory\n", real_src);
        return -1;
    }
    if (!is_within_home(real_src)) {
        printf("%s is outside the home directory\n", real_src);
        return -1;
    }
    // already monitored or parent/subdir of monitored
    if (daemon_is_monitored(real_src)) {
        fprintf(stderr, "Error: %s is already monitored\n", real_src);
        return -1;
    }
    char **mon_paths; int mon_count;
    mon_paths = daemon_list_all_paths(&mon_count);
    for (int i = 0; i < mon_count; i++) {
        if (is_subpath(mon_paths[i], real_src) || is_subpath(real_src, mon_paths[i])) {
            fprintf(stderr, "Error: %s conflicts with monitored path %s\n", real_src, mon_paths[i]);
            return -1;
        }
    }

    // output path
char parent[PATH_MAX], resolved_parent[PATH_MAX], real_out[PATH_MAX];
strncpy(parent, output_raw[0] ? output_raw : real_src, sizeof(parent));
dirname(parent);
if (!realpath(parent, resolved_parent)) {
    perror("realpath(parent)");
    return -1;
}
if (!is_within_home(resolved_parent) || is_subpath(real_src, resolved_parent)) {
    fprintf(stderr, "Error: output path %s is invalid\n", resolved_parent);
    return -1;
}
```

```
if (output_raw[0]) {
    const char *base = strrchr(output_raw, '/');
    base = base ? base + 1 : output_raw;
    snprintf(real_out, sizeof(real_out), "%s/%s", resolved_parent, base);
} else {
    snprintf(real_out, sizeof(real_out), "%s_arranged", real_src);
}
    if (mkdir(real_out, 0755) < 0 && errno != EEXIST) {
        perror("mkdir(real_out)");
        return -1;
    }

    // exclusive path
    for (int i = 0; i < excl_count; i++) {
        char exrp[PATH_MAX]; realpath(excl_list[i], exrp);
        if (!is_subpath(real_src, exrp)) {
            fprintf(stderr, "Error: exclude path %s is not under %s\n", excl_list[i], real_src);
            return -1;
        }
        for (int j = i+1; j < excl_count; j++) {
            char er2[PATH_MAX]; realpath(excl_list[j], er2);
            if (is_subpath(exrp, er2) || is_subpath(er2, exrp)) {
                fprintf(stderr, "Error: exclude paths %s and %s overlap\n", excl_list[i], excl_list[j]);
                return -1;
            }
        }
    }

    // make config
    char conf_path[PATH_MAX];
    snprintf(conf_path, sizeof(conf_path), "%s/ssu_cleanupd.config", real_src);
    if (access(conf_path, F_OK) != 0) {
        if (write_config_file(conf_path,
                              real_src,
                              real_out,
                              interval,
                              max_logs,
                              excl_list,
                              excl_count,
                              extensions,
                              mode) < 0) {
            fprintf(stderr, "Failed to write config\n");
            return -1;
        }
    }

    // create log
    char log_path[PATH_MAX];
```

```
    snprintf (log_path,sizeof(log_path),"%s/ssu_cleanupd.log", real_src);
    FILE *fp = fopen(log_path, "a");
    if(fp) fclose(fp);

    // create daemon
    if (daemon_add(real_src,
                   real_out,
                   conf_path,
                   interval,
                   max_logs,
                   excl_list,
                   excl_count,
                   extensions,
                   mode) < 0) {
        fprintf(stderr, "Failed to start daemon\n");
        return -1;
    }

    return 0;
}

int command_show(void) {
    daemon_t **list;
    int count = daemon_list_all(&list);
    while (1) {
        printf("Current working daemon process list\n\n");
        printf("0. exit\n");
        for (int i = 0; i < count; i++) {
            printf("%d. %s\n", i + 1, list[i]->monitoring_path);
        }

        printf("\nSelect one to see process info : ");
        char buf[32];
        if (!fgets(buf, sizeof(buf), stdin)) return 0;
        int sel = atoi(buf);
        if (sel == 0) {
            return 0;
        }
        if (sel < 0 || sel > count) {
            printf("Please check your input is valid\n\n");
            continue;
        }

        // valid selection
        daemon_t *d = list[sel - 1];
        parse_config(d->config_file, &d->cfg);

        // Config detail
```

```
        printf("\n1. config detail\n\n");
        printf("monitoring_path : %s\n", d->monitoring_path);
        printf("pid : %d\n", d->pid);
        printf("start_time : %s\n", d->cfg.start_time);
        printf("output_path : %s\n", d->output_path);
        printf("time_interval : %ld\n", d->cfg.time_interval);
        if (d->cfg.max_log_lines > 0)
            printf("max_log_lines : %zu\n", d->cfg.max_log_lines);
        else
            printf("max_log_lines : none\n");
        if (d->cfg.exclude_count > 0) {
            printf("exclude_path : ");
            for (int j = 0; j < d->cfg.exclude_count; j++) {
                printf("%s%s", d->cfg.exclude_paths[j], j < d->cfg.exclude_count - 1 ? "," : "");
            }
            printf("\n");
        } else {
            printf("exclude_path : none\n");
        }
        printf("extension : %s\n", *d->cfg.extensions ? d->cfg.extensions : "all");
        printf("mode : %d\n", d->cfg.mode);
        // Log detail
        printf("\n2. log detail\n\n");
        char log_path[MAX_PATH];
        snprintf(log_path, sizeof(log_path), "%s/ssu_cleanupd.log", d->monitoring_path);
        FILE *lf = fopen(log_path, "r");
        if (lf) {
            char line[MAX_PATH * 2];
            while (fgets(line, sizeof(line), lf)) {
                printf("%s", line);
            }
            fclose(lf);
        } else {
            printf("\n");
        }
        return 0;
    }
}


// Modify (reload) daemon
int command_modify(const char *args) {
    if (!args) {
        fprintf(stderr, "Usage: modify <DIR_PATH> [OPTION]...\n");
        return -1;
    }

    // 복사 버퍼 확보
```

```
char *buf = strdup(args);
if (!buf) return -1;

// 첫 번째 토큰: 모니터링할 디렉토리
char *tok = strtok(buf, " \t");
if (!tok) {
    fprintf(stderr, "Usage: modify <DIR_PATH> [OPTION]...\n");
    free(buf);
    return -1;
}
char src_raw[PATH_MAX];
strncpy(src_raw, tok, PATH_MAX);

// 절대경로 변환 및 검증
char real_src[PATH_MAX];
if (!realpath(src_raw, real_src)) {
    fprintf(stderr, "Error: %s is not valid path\n", src_raw);
    free(buf);
    return -1;
}

// 대상 데몬 찾기
daemon_t **list;
int count = daemon_list_all(&list);
int idx = -1;
for (int i = 0; i < count; i++) {
    if (strcmp(list[i]->monitoring_path, real_src) == 0) {
        idx = i;
        break;
    }
}
if (idx < 0) {
    fprintf(stderr, "No daemon watching %s\n", real_src);
    free(buf);
    return -1;
}

// 기존 config 로드
daemon_t *d = list[idx];
parse_config(d->config_file, &d->cfg);

// 옵션 파싱: add와 동일하게 구현
char output_raw[PATH_MAX] = "";
long interval        = d->cfg.time_interval;
size_t max_logs      = d->cfg.max_log_lines;
int mode             = d->cfg.mode;
char **excl_list     = d->cfg.exclude_paths;
int excl_count       = d->cfg.exclude_count;
```

```
char *extensions     = strdup(d->cfg.extensions);

while ((tok = strtok(NULL, " \t")) != NULL) {
    char opt = tok[1];
    char *arg = strtok(NULL, " \t");
    switch (opt) {
        case 'd':
            if (!arg) {
                fprintf(stderr,"-d requires a path\n");
                free(buf);
                return -1;
            }
            strncpy(output_raw, arg, PATH_MAX);
            break;
        case 'i':
            if (!arg || !isdigit(arg[0])) {
                fprintf(stderr,"-i requires a number\n");
                free(buf);
                return -1;
            }
            interval = atol(arg);
            break;
        case 'l':
            if (!arg || !isdigit(arg[0])) {
                fprintf(stderr,"-l requires a number\n");
                free(buf);
                return -1;
            }
            max_logs = atoi(arg);
            break;
        case 'x':
            // 기존 리스트 해제 후 재생성
            for (int j=0; j<excl_count; j++) free(excl_list[j]);
            free(excl_list); excl_list = NULL; excl_count = 0;
            while (arg && arg[0] != '-') {
                excl_list = realloc(excl_list, sizeof(char*)*(excl_count+1));
                excl_list[excl_count++] = strdup(arg);
                arg = strtok(NULL, " \t");
            }
            break;
        case 'e':
            free(extensions);
            extensions = strdup(arg ? arg : "");
            break;
        case 'm':
            if (!arg || !isdigit(arg[0])) {
                fprintf(stderr,"-m requires 1-3\n");
                free(buf);
```

```
                return -1;
            }
            mode = atoi(arg);
            break;
        default:
            fprintf(stderr, "Unknown option -%c\n", opt);
            free(buf);
            return -1;
    }
}

// config 파일에 덮어쓰기
if (write_config_file(d->config_file,
                      real_src,
                      output_raw[0] ? output_raw : d->output_path,
                      interval,
                      max_logs,
                      excl_list,
                      excl_count,
                      extensions,
                      mode) < 0) {
    fprintf(stderr, "Failed to write config\n");
    free(buf);
    return -1;
}

// SIGHUP 전송
if (daemon_reload(d->pid) == 0) {
    printf("Reload signal sent to PID %d\n", d->pid);
} else {
    fprintf(stderr, "Failed to reload PID %d\n", d->pid);
}

free(buf);
return 0;
}

// Remove daemon
int command_remove(const char *args) {
if (!args) {
    fprintf(stderr, "Usage: remove <DIR_PATH>\n");
    return -1;
}

char real_src[PATH_MAX];
if (!realpath(args, real_src)) {
    fprintf(stderr, "Error: %s is not a valid path\n", args);
    return -1;
```

```
    }

    daemon_t **list;
    int count = daemon_list_all(&list);
    pid_t target_pid = 0;
    for (int i = 0; i < count; i++) {
        if (strcmp(list[i]->monitoring_path, real_src) == 0) {
            target_pid = list[i]->pid;
            break;
        }
    }
    if (target_pid == 0) {
        fprintf(stderr, "No daemon watching %s\n", real_src);
        return -1;
    }

    if (daemon_remove(target_pid) == 0) {
        printf("Daemon watching %s removed (PID %d)\n", real_src, target_pid);
        return 0;
    } else {
        fprintf(stderr, "Failed to remove daemon PID %d\n", target_pid);
        return -1;
    }
}



// Write configuration file
int write_config_file(const char *path,
                      const char *monitoring_path,
                      const char *output_path,
                      long time_interval,
                      size_t max_log_lines,
                      char **exclude_paths,
                      int exclude_count,
                      const char *extensions,
                      int mode)
{
    FILE *fp = fopen(path, "w");
    if (!fp) return -1;
    time_t now = time(NULL);
    struct tm *tm = localtime(&now);
    char tbuf[20];
    strftime(tbuf, sizeof(tbuf), "%Y-%m-%d %H:%M:%S", tm);
    fprintf(fp, "monitoring_path : %s\n", monitoring_path);
    fprintf(fp, "pid             : %d\n", getpid());
    fprintf(fp, "start_time      : %s\n", tbuf);
    fprintf(fp, "output_path     : %s\n", output_path);
```

```
    fprintf(fp, "time_interval    : %ld\n", time_interval);
    if (max_log_lines > 0)
        fprintf(fp, "max_log_lines    : %zu\n", max_log_lines);
    else
        fprintf(fp, "max_log_lines    : none\n");
    if (exclude_paths && *exclude_paths) {
        fprintf(fp, "exclude_path     : ");
        for (int i = 0; exclude_paths[i]; i++)
            fprintf(fp, "%s%s", exclude_paths[i], exclude_paths[i+1] ? "," : "\n");
    } else {
        fprintf(fp, "exclude_path     : none\n");
    }
    fprintf(fp, "extension         : %s\n", extensions && *extensions ? extensions : "all");
    fprintf(fp, "mode               : %d\n", mode);
    fclose(fp);
    return 0;
}
```

5-4. daemon.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <limits.h>
#include <utime.h>
#include <pwd.h>
#include <ctype.h>
#include "header.h"  // daemon_t, daemon_config_t, prototypes for arrange_directory, daemon_list_all, etc.

#define MAX_DAEMONS 128
static daemon_t *daemon_list[MAX_DAEMONS];
static int daemon_count = 0;
daemon_config_t *current_cfg = NULL;
bool current_first_run = true;

// Forward declarations
static void daemonize_process(void);
static void monitor_loop(daemon_t *d);
static int is_subpath(const char *parent, const char *child);
static char *trim(char *s);
static int count_ext(const char *extensions_str);
```

```
// Add a new daemon: fork, daemonize child, and register in parent
int daemon_add(const char *monitor_path,
               const char *output_path,
               const char *config_file,
               long interval,
               size_t max_logs,
               char **exclude_paths,
               int exclude_count,
               const char *extensions,
               int mode)
{
    if (daemon_count >= MAX_DAEMONS) return -1;
    pid_t pid = fork();
    if (pid < 0) {
        return -1;
    }
    if (pid > 0) {
        // Parent: register daemon metadata
        daemon_t *d = malloc(sizeof(*d));
        if (!d) return -1;
        memset(d, 0, sizeof(*d));
        d->pid = pid;
        d->monitoring_path = strdup(monitor_path);
        d->output_path     = strdup(output_path);
        d->config_file     = strdup(config_file);
        // Initialize config
        d->cfg.time_interval = interval;
        d->cfg.max_log_lines = max_logs;
        d->cfg.exclude_paths = exclude_paths;
        d->cfg.exclude_count = exclude_count;
        d->cfg.extensions    = strdup(extensions ? extensions : "");
        d->cfg.ext_count     = count_ext(extensions);
        d->cfg.mode          = mode;
        daemon_list[daemon_count++] = d;
        return 0;
    }
    // Child: become daemon
    daemonize_process();
    // Initialize own struct
    daemon_t self;
    memset(&self, 0, sizeof(self));
    self.pid = getpid();
    self.monitoring_path = strdup(monitor_path);
    self.output_path     = strdup(output_path);
    self.config_file     = strdup(config_file);
    // Initial config load
    parse_config(self.config_file, &self.cfg);
    // Enter monitoring loop
```

```
    monitor_loop(&self);
    exit(0);
}

// Remove daemon by PID: send SIGTERM and unregister
int daemon_remove(pid_t pid) {
    for (int i = 0; i < daemon_count; i++) {
        if (daemon_list[i]->pid == pid) {
            kill(pid, SIGTERM);
            waitpid(pid, NULL, 0);
            // Free and shift list
            free(daemon_list[i]->monitoring_path);
            free(daemon_list[i]->output_path);
            free(daemon_list[i]->config_file);
            free(daemon_list[i]->cfg.extensions);
            free(daemon_list[i]);
            for (int j = i; j < daemon_count - 1; j++)
                daemon_list[j] = daemon_list[j+1];
            daemon_count--;
            return 0;
        }
    }
    return -1;
}

// Reload daemon config by PID: send SIGHUP
int daemon_reload(pid_t pid) {
    for (int i = 0; i < daemon_count; i++) {
        if (daemon_list[i]->pid == pid) {
            kill(pid, SIGHUP);
            return 0;
        }
    }
    return -1;
}

// List all registered daemon metadata
int daemon_list_all(daemon_t ***out_list) {
    *out_list = daemon_list;
    return daemon_count;
}

// List all monitored paths for conflict detection
char **daemon_list_all_paths(int *out_count) {
    if (out_count) *out_count = daemon_count;
    char **paths = malloc(sizeof(char*) * daemon_count);
    for (int i = 0; i < daemon_count; i++) {
        paths[i] = daemon_list[i]->monitoring_path;
```

```
    }
    return paths;
}

// Check if a path is already monitored (exact match)
int daemon_is_monitored(const char *path) {
    for (int i = 0; i < daemon_count; i++) {
        if (strcmp(daemon_list[i]->monitoring_path, path) == 0)
            return 1;
    }
    return 0;
}

// Core monitoring loop: parse config, arrange, log, and sleep
static void monitor_loop(daemon_t *d) {
    struct timespec interval;
    while (1) {
        parse_config(d->config_file, &d->cfg);
        d->cfg.pid = d->pid;
        d->cfg.monitoring_path = d->monitoring_path;
        d->cfg.output_path = d->output_path;
        interval.tv_sec  = d->cfg.time_interval;
        interval.tv_nsec = 0;
        current_cfg = &d->cfg;
        arrange_directory(d->monitoring_path,
                          d->output_path,
                          d->cfg.exclude_paths,
                          d->cfg.exclude_count,
                          d->cfg.extensions,
                          d->cfg.mode);
        current_first_run = false;
        nanosleep(&interval, NULL);
    }
}

// Daemonize process using double-fork method
static void daemonize_process(void) {
    pid_t sid;
    if ((sid = setsid()) < 0) exit(1);
    // restore default SIGHUP handler
    signal(SIGHUP, SIG_DFL);
    pid_t pid = fork();
    if (pid < 0) exit(1);
    if (pid > 0) exit(0);
    umask(0);
    chdir("/");
    for (int fd = 0; fd < 3; fd++) close(fd);
    open("/dev/null", O_RDONLY);
```

```
    open("/dev/null", O_RDWR);
    open("/dev/null", O_RDWR);
}

// Check if 'child' is subpath of 'parent'
static int is_subpath(const char *parent, const char *child) {
    char rp[PATH_MAX], rc[PATH_MAX];
    if (!realpath(parent, rp) || !realpath(child, rc)) return 0;
    size_t lp = strlen(rp);
    return strncmp(rp, rc, lp) == 0 && (rc[lp] == '/' || rc[lp] == '\0');
}

// Trim leading/trailing whitespace
static char *trim(char *s) {
    char *end;
    while (isspace((unsigned char)*s)) s++;
    if (*s == 0) return s;
    end = s + strlen(s) - 1;
    while (end > s && isspace((unsigned char)*end)) end--;
    end[1] = '\0';
    return s;
}

// Count comma-separated extensions
static int count_ext(const char *extensions_str) {
    if (!extensions_str || *extensions_str == '\0') return 0;
    int count = 0;
    char *copy = strdup(extensions_str);
    char *tok = strtok(copy, ",");
    while (tok) {
        count++;
        tok = strtok(NULL, ",");
    }
    free(copy);
    return count;
}

// Parse config file into daemon_config_t
int parse_config(const char *config_file, daemon_config_t *cfg) {
    FILE *fp = fopen(config_file, "r");
    if (!fp) return -1;
    // set defaults
    cfg->time_interval = 10;
    cfg->max_log_lines = 0;
    cfg->exclude_paths = NULL;
    cfg->exclude_count = 0;
    free(cfg->extensions);
    cfg->extensions = strdup("");
```

```
    cfg->ext_count = 0;
    cfg->mode = 1;
    char line[1024];
    while (fgets(line, sizeof(line), fp)) {
        char *p = strchr(line, ':');
        if (!p) continue;
        *p = '\0';
        char *key = trim(line);
        char *val = trim(p + 1);
        if (strcmp(key, "monitoring_path") == 0) {
            // ignored in reload
        } else if (strcmp(key, "pid") == 0) {
            cfg->pid = (pid_t)atoi(val);
        } else if (strcmp(key, "start_time") == 0) {
            free(cfg->start_time);
            cfg->start_time = strdup(val);
        } else if (strcmp(key, "output_path") == 0) {
            free(cfg->output_path);
            cfg->output_path = strdup(val);
        } else if (strcmp(key, "time_interval") == 0) {
            cfg->time_interval = atol(val);
        } else if (strcmp(key, "max_log_lines") == 0) {
            if (strcmp(val, "none") != 0) cfg->max_log_lines = (size_t)atoi(val);
        } else if (strcmp(key, "exclude_path") == 0) {
            if (strcmp(val, "none") != 0) {
                char *tok = strtok(val, ",");
                while (tok) {
                    cfg->exclude_paths = realloc(cfg->exclude_paths,
                        sizeof(char*) * (cfg->exclude_count + 1));
                    cfg->exclude_paths[cfg->exclude_count++] = strdup(trim(tok));
                    tok = strtok(NULL, ",");
                }
            }
        } else if (strcmp(key, "extension") == 0) {
            if (strcmp(val, "all") != 0) {
                free(cfg->extensions);
                cfg->extensions = strdup(val);
                cfg->ext_count = count_ext(val);
            }
        } else if (strcmp(key, "mode") == 0) {
            cfg->mode = atoi(val);
        }
    }
    fclose(fp);
    return 0;
}

// Log an event and enforce max_log_lines
```

```
int log_event(const daemon_config_t *cfg,
              const char *src_path,
              const char *dst_path)
{
    // timestamp
    time_t now = time(NULL);
    struct tm lt;
    localtime_r(&now, &lt);
    char tbuf[9];
    snprintf(tbuf,sizeof(tbuf),"%02d:%02d:%02d",lt.tm_hour,lt.tm_min,lt.tm_sec);

    // new entry
    char entry[PATH_MAX*2];
    snprintf(entry, sizeof(entry), "[%s] [%d] [%s] [%s]\n",
             tbuf, (int)cfg->pid, src_path, dst_path);
    // read existing
    char log_path[PATH_MAX];
    snprintf(log_path, sizeof(log_path), "%s/ssu_cleanupd.log", cfg->monitoring_path);
    FILE *fp = fopen(log_path, "r");
    char **lines = NULL; size_t count = 0;
    if (fp) {
        char buf[PATH_MAX*2];
        while (fgets(buf, sizeof(buf), fp)) {
            lines = realloc(lines, sizeof(char*)*(count+1));
            lines[count++] = strdup(buf);
        }
        fclose(fp);
    }
    lines = realloc(lines, sizeof(char*)*(count+1));
    lines[count++] = strdup(entry);
    // enforce max
    if (cfg->max_log_lines > 0 && count > cfg->max_log_lines) {
        size_t to_remove = count - cfg->max_log_lines;
        for (size_t i = 0; i < to_remove; i++) free(lines[i]);
        memmove(lines, lines + to_remove, sizeof(char*) * (count - to_remove));
        count = cfg->max_log_lines;
    }
    fp = fopen(log_path, "w");
    if (!fp) { for (size_t i=0;i<count;i++) free(lines[i]); free(lines); return -1; }
    for (size_t i=0;i<count;i++) { fputs(lines[i], fp); free(lines[i]); }
    free(lines);
    fclose(fp);
    return 0;
}
```

5-5. header.h

```
#ifndef HEADER_H
#define HEADER_H
```

```
#include <sys/types.h>
#include <stdbool.h>
#include <stddef.h>
#include <limits.h>
#include <time.h>
#include <sys/wait.h>
#include <errno.h>
#include <libgen.h>

// Maximum path length
#define MAX_PATH 4096
#define MAX_COMMAND 1000

// Daemon configuration structure
typedef struct {
    char    *monitoring_path;    // Monitored directory path
    pid_t    pid;                // Daemon PID
    char    *start_time;         // Daemon start time string
    char    *output_path;        // Output directory path
    long     time_interval;      // Monitoring interval (seconds)
    size_t   max_log_lines;      // Maximum log lines (0 = unlimited)
    char   **exclude_paths;      // List of paths to exclude
    int      exclude_count;      // Number of exclude paths
    char    *extensions;         // Comma-separated extensions to include
    int      ext_count;          // Number of extensions
    int      mode;               // Duplicate handling mode
} daemon_config_t;

// Daemon metadata
typedef struct {
    pid_t           pid;                // Daemon PID
    char            *monitoring_path;    // Monitored directory
    char            *output_path;        // Output directory
    char            *config_file;        // Path to config file
    daemon_config_t  cfg;                // Current configuration
} daemon_t;

// command.c
int command_add(const char *args);
int command_show(void);
int command_modify(const char *args);
int command_remove(const char *args);
void command_help(void);
void command_loop(void);

// daemon.c
int daemon_add(const char *monitor_path,
```

```
                const char *output_path,
                const char *config_file,
                long interval,
                size_t max_logs,
                char **exclude_paths,
                int exclude_count,
                const char *extensions,
                int mode);
int daemon_remove(pid_t pid);
int daemon_reload(pid_t pid);
int daemon_list_all(daemon_t ***out_list);
char **daemon_list_all_paths(int *out_count);
int daemon_is_monitored(const char *path);
int parse_config(const char *config_file, daemon_config_t *cfg);
int log_event(const daemon_config_t *cfg,
              const char *src_path,
              const char *dst_path);
int write_config_file(const char *path,
                      const char *monitoring_path,
                      const char *output_path,
                      long time_interval,
                      size_t max_log_lines,
                      char **exclude_paths,
                      int exclude_count,
                      const char *extensions,
                      int mode);

// arrange.c
void arrange_directory(const char *src,
                       const char *dst,
                       char **exclude_paths,
                       int exclude_count,
                       const char *extensions,
                       int mode);

#endif // HEADER_H
```