컴퓨터학부 20162506 전연선

I. 개요

디몬 프로세스는 사용자가 직접 제어하지 않고, 백그라운드 상태에서 반복적으로 작업 수행 하는 프로그램이다. 모든 세션으로부터 독립 시키고 열려 있는 모든 터미널과 파일 디스크립터를 닫고 파일 생성 마스크를 0으로 설정해 디몬 프로세스가 되기 전 설정을 마치고 자식 프로세스는 생성하고 부모 프로세스를 종료하면 디몬 프로세스를 생성할 수 있다. 그리고 실행 중인 디몬 프로세스(또는 일반 프로세스)를 종료하고 싶을 때는 해당 프로세스에 SIGKILL 시그널을 보내주어 종료 할 수 있다.

II. 설계

가) 목표설정

ssu_backup : Linux 환경에서 디몬 프로세스를 생성하여 주기적으로 파일을 백업하는 기능을 구현하도록 한다. 형식은 ssu_backup [filename] [period] [option] 이며, [filename]은 특별한 옵션이 없는 경우에는 일반파일이고, [period]는 범위가 3에서 10까지 이다. ssu_backup을 실행했을 때 이미 실행 중인 ssu_backup 디몬 프로세스가 있는 경우에는 사용자 정의 시그널을 발생시켜서 종료시키고, 백업을 진행하면서 'backup_log.txt'의 로그 파일에 과정을 기록하도록 한다.

나) 구조 설계

1) main 함수 :

① main 함수의 인자를 받아 백업 대상 파일, 백업 주기, 옵션을 받아 변수에 저장한다. (option( ) 사용)

② 에러가 없는 경우 디몬 프로세스를 생성한다(start_daemon( )). 생성하기 전 이미 실행 중인 ssu_backup 디몬 프로세스가 있는 경우 종료한다(fnd_proc( )).

③ 백업 대상파일의 파일 정보를 저장한 후(get_info( )), 백업 될 파일을 복사하여 새 백업 파일을 만들어낸다(backup_name( ), ssu_copy( )). 백업을 수행한 후 결과를 로그파일에 남긴다(write_log( )). 이 과정을 [period]에 맞춰 반복해서 백업한다.
옵션 '-d'가 적용 된 경우 디렉터리 안의 하위 파일들을 차례대로 백업하여 처리한다(option_d( )).

④ 옵션 '-r' : 백업 대상 파일을 16진수화 한 이름과 백업 된 폴더 파일들과 비교해서 복구할 백업파일을 출력한다 (backup_name( ), option_r( ))

III. 구현 (각 함수별 기능)

가) ssu_backup.c

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

① 매개 변수 : argc, argv[]

i. argc : 명령 줄에서 입력된 명령행 옵션들의 개수를 저장한다.

ii. argv[] : 명령 줄에서 입력된 옵션들과 백업될 파일, 백업 주기를 받아온다.

② 기능

백업될 파일을 디몬 프로세스를 통해 반복적으로 백업한다.

사용자가 프로그램을 처음 시작하면, option()을 호출하여 옵션과 백업파일, 백업 주기 등의 정보를 받는다. 지정한 변수에 값들을 저장한 후, 기존에 실행 중인 ./ssu_backup

디몬 프로세스가 있는지 fnd_process()를 호출하여 확인한다. 그 후, 옵션에 맞게 진행한다.

'-r'이 적용 된 경우, [period]는 받지 않는 것으로 간주하고 백업 파일을 'filename'변수에 저장한다. 그리고 해당 파일의 파일 정보(수정 시간, 크기, 백업한 시간)를 받아와 "ssu_backup.h"에 저장되어 있는 구조체를 이용하여 값들을 저장한다(get_info()). 변수 'backupName'에 백업 대상 파일의 절대경로를 16진수화 하여 백업 파일의 이름을 새로 만들고(backup_name()), option_r()을 호출해 해당 옵션 기능을 작동시킨다.

그 외의 경우 [period], [filename]을 변수 'period', [filename]에 각각 저장한다. '-d' 옵션이 있는 경우 우선 해당 파일이 디렉터리인지 확인하여 에러가 나지 않게끔 한다. '-d' 옵션이 적용 되지 않은 경우에는 반대로 해당 파일이 일반파일이 맞는지 확인하여 에러가 나지 않게끔 한다. 에러가 나지 않는 지 확인한 후, 디몬 프로세스를 생성한다('-r', '-c' 옵션은 제외).

나) kill_process.c

1) fnd_proc();

① 기능 : 이미 실행 중인 ssu_backup 디몬 프로세스가 있는 지 확인한다. SIGUSR1 시그널이 발생한 경우 'kill_handler()' 사용자 핸들러 함수를 호출하여 처리하도록 설정한다. 디렉터리 '/proc'을 살펴보면 진행 중인 프로세스의 pid의 이름으로 된 하위 디렉터리들이 만들어져 있는 것을 알 수 있다. scandir()함수를 통해 '/proc' 디렉터리 값들을 변수 'namelist'에 정리하여 저장하고 차례대로 디렉터리 내부 stat을 살펴봐서 ssu_backup이 써져 있는 경우 이미 실행 중인 프로세스가 있다는 것을 의미하므로 raise()를 통해 자신에게 시그널 SIGUSR1을 발생시킨다. 시그널 SIGUSR1이 발생했으므로 핸들러 함수를 호출하여 종료하도록 한다.

2) void kill_handler(int signo);

① 기능 : 시그널 SIGUSR1이 발생했으므로 기존 디몬 프로세스를 종료한다. backup_log.txt 로그 파일에 현재 시간을 받아와 종료시점을 남기고 getpid()로 죽은 프로세스의 pid를 구해 기록한다. 자신에게 SIGKILL 시그널을 보내 실행중인 ssu_backup 디몬 프로세스를 죽인다.

다) get_info.c

1) int get_info(char *filename, backupInfo *info);

① 매개 변수 : filename, info

i. filename : 백업 대상 파일의 이름

ii. info : 백업 대상 파일의 파일 정보를 저장할 구조체

② 리턴 값 : 해당 파일이 수정된 경우 1, 삭제 된 경우 -1, 어떠한 변화도 없는 경우 0을 리턴 한다.

③ 기능 : 백업 대상 파일의 파일 정보를 받아온다. 파일의 수정시간과 크기를 stat()함수를 통해 받아와 info->mtime과 info->size에 저장한다. 그리고 time()함수를 통해 현재 백업 중인 시간을 구해 info.->backupt에 저장하고, "MMDDHHMMSS" 형식에 맞게 문자열에 저장한 후 info->btime에 저장한다.

라) backup_name.c

1) char* backup_name(char *filename, char *btime);

① 매개 변수 : filename, btime

i. filename : 백업 대상의 파일 이름

ii. btime : 백업되는 시간을 저장한 문자열(MMDDHHMMSS)

② 리턴 값 : 백업될 파일의 이름 ("백업 대상의 절대경로를 16진수화 한 값_백업시간")

③ 기능 : 새로 백업 될 파일의 이름을 만든다.
사용자가 입력한 파일이 절대 경로로 되어있는지, 상대 경로로 되어 있는 지 파악한 후 모두 절대 경로로 바꿔서 변수 'pathname'에 저장한다. 그리고 'pathname'의 문자를 한글자씩 읽어서 각각 16진수 아스키 값으로 나타내어 변수 'new_fname'에 저장한다. 저장한 'new_fname'뒤에는 "_"와 함께 인자로 받아온 btime을 붙여서 백업될 파일의 이름을 만든다.

마) ssu_cp.c

1) void ssu_cp(char *source, char *backupfile, backupInfo info);

① 매개 변수 : source, backupfile, info

i. source : 백업 될 파일

ii. backupfile : 새로 백업되어 만들어질 파일의 이름

iii. info : 백업 대상 파일의 정보가 담긴 구조체

② 기능 : open()을 통해 백업 될 파일을 열어 파일을 읽어오고, 읽어 온 내용은 write()를 통해 'backupfile'이름의 새로운 백업 파일에 복사를 한다.

바) write_log.c

1) void write_log(backupInfo info, int state);

① 매개 변수 : info, state

i. info : 백업 할 파일의 정보를 담고 있는 구조체

ii. state : get_info에서 리턴 한 값. (0:수정/삭제가 일어나지 않은 경우, 1:수정, -1:삭제)

② 기능 : 백업 파일이 백업됨을 알리는 로그 메시지를 'backup_log.txt'에 기록한다.
'info.backupt'를 이용하여 백업 시간과 파일 이름을 출력하고, state이 1인 경우 "is modified", -1인 경우 "is deleted", 0인 경우 파일의 사이즈와 수정 시간을 형식에 맞게 출력한다.

사) option_d.c

1) void option_d(char *_dirname, char *_dir_hex, char *btime, time_t backup) {

① 매개 변수 : _dirname, _dir_hex, btime, backup

i. _dirname : 입력받은 디렉터리의 절대경로

ii. _dir_hex : 디렉터리의 절대경로를 ASCII 값으로 16진수화 한 값

iii. btime : 백업 시간(MMDDHHMMSS 형식)

iv. backup : 백업 시간

② 기능 : '-d' 옵션에 맞게 백업 될 디렉터리의 내부 파일들의 스레드를 각각 생성하여 백업한다.
구조체 thread_data tid[]에 _dirname, _dir_hex, btime, backup 값을 저장해서 쓰레드가

생성되면 자원을 공유하면서 잘못된 정보로 백업처리되지 않게끔 한다. 입력받은 디렉터리내부 파일들을 scandir()을 통해 namelist에 저장하고, 각각 파일들이 디렉터리인지 일반파일인지 확인한다. 디렉터리인 경우, 해당 디렉터리의 이름으로 다시 option_d()를 호출해서 서브디렉토리에 있는 파일까지 백업처리를 할 수 있도록 한다. 일반 파일임이 확인 되면 pthread_create()를 통해 스레드를 생성하고 backup_thread()함수를 실행한다. 해당 함수로 넘겨지는 인자는 thread_data tid 이다.

2) void *backup_thread(void *arg)

① 기능 : 생성된 쓰레드에 해당하는 파일을 백업한다. 주어진 옵션이 없을 때 백업하는 순서대로 get_info()를 통해 파일 정보를 받아오고, backup_name()을 사용하여 만들어질 백업파일의 이름을 만들어낸다. 그리고 write_log()와 ssu_cp()를 호출하여 백업 및 백업 관련 내용을 로그 파일에 기록한다.

아) option_r.c

1) void option_r(char *filename)

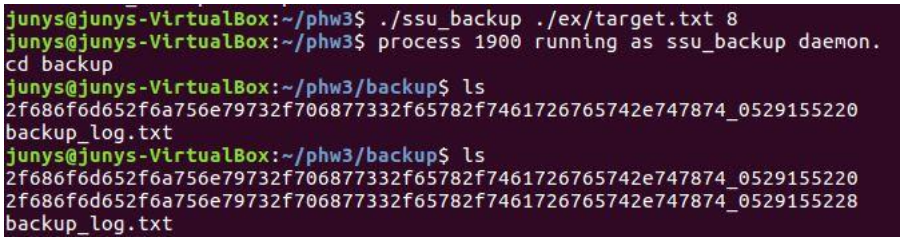
① 매개 변수 filename : 복구할 파일의 이름

② 기능 : '-r' 기능을 수행한다. 입력한 파일을 백업한 백업 파일들을 차례대로 출력해 사용자가 선택한 백업 파일로 복구한다.
입력 받은 filename의 절대경로를 구해 ASCII 값의 16진수화하여 '_filename'에 저장한다. 그리고 './backup' 디렉터리의 파일들과 차례대로 비교하여 같은 이름이 있는 지 확인한다. 백업 파일들은 16진수화한 값과 함께 백업 시간도 저장되어 있으므로 백업 시간이 써져 있는 부분 전까지만을 비교한다. 같은 이름의 파일들은 '1'번부터 "이름_백업시간", 백업 파일 크기를 차례대로 출력한다. 사용자로부터 복구하고 싶은 파일의 번호를 받아온다. 복구할 파일을 읽어와 문자열에 저장한 후, 원본 파일은 덮어쓰는 방식으로 write()를 사용하여 복구하고, 복구된 파일의 내용을 출력한다.

IV. 테스트 및 결과

가) ssu_backup [FILENAME] [PERIOD] : filename을 period 주기에 따라 백업

1) 백업파일 이름은 백업대상의 절대경로를 16진수 아스키값으로 변환하고 "_백업시간"을 붙인다.



2) ssu_backup을 디몬 프로세스로 수행하고, 이미 실행중인 ssu_backup이 있는 경우 종료 후 새로 디몬 프로세스를 생성한다. 종료된 프로세스에 관해서 로그메시지를 남긴다.

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt 7
junys@junys-VirtualBox:~/phw3$ process 3408 running as ssu_backup daemon.
ps -eaf | grep ./ssu_backup
junys      3408  1293  0 14:08 ?        00:00:00 ./ssu_backup ./ex/target.txt 7
junys      3411  2789  0 14:08 pts/4    00:00:00 grep --color=auto ./ssu_backup
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt 7
junys@junys-VirtualBox:~/phw3$ process 3413 running as ssu_backup daemon.
ps -eaf | grep ./ssu_backup
junys      3413  1293  0 14:08 ?        00:00:00 ./ssu_backup ./ex/target.txt 7
junys      3415  2789  0 14:08 pts/4    00:00:00 grep --color=auto ./ssu_backup
```

① 이미 실행중인 ssu_backup 종료 후 로그에 종료 된 pid 기록

```
[0529 14:08:20] target.txt  [size:11/mtime:0529 00:17:39]
[0529 14:08:27] target.txt  [size:11/mtime:0529 00:17:39]
[0529 14:08:28] ssu_backup<pid:3408> exit
[0529 14:08:28] target.txt  [size:11/mtime:0529 00:17:39]
[0529 14:08:35] target.txt  [size:11/mtime:0529 00:17:39]
```

3) 백업대상 파일이 수정되었거나 삭제되었을 경우 로그메시지를 남기고 디몬 프로세스 종료

① 백업대상 파일이 수정된 경우

```
[0529 16:12:49] target.txt  [size:11/mtime:0529 00:17:39]
[0529 16:12:57] target.txt is modified [size:21/mtime:0529 16:12:51]
```

② 백업대상 파일이 삭제된 경우

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target_rm.txt 5
junys@junys-VirtualBox:~/phw3$ process 1961 running as ssu_backup daemon.
rm ./ex/target_rm.txt
junys@junys-VirtualBox:~/phw3$ ps -eaf | grep ./ssu_backup
junys      1964  1805  0 16:00 pts/18   00:00:00 grep --color=auto ./ssu_backup
[0529 16:00:39] target_rm.txt  [size:11/mtime:0529 16:00:26]
[0529 16:00:44] target_rm.txt  [size:11/mtime:0529 16:00:26]
[0529 16:00:49] target_rm.txt  [size:11/mtime:0529 16:00:26]
[0529 16:00:54] target_rm.txt is deleted
```

나) ssu_backup [filename] [period] [option]

1) [filename]

① 파일이 존재하지 않는 경우 에러 처리

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup file.txt 8
"file.txt" backup file does not exist
```

② 일반파일이 아닌 경우 에러 처리

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ~/phw3 8
ssu_backup error
file "/home/junys/phw3" must be regular file
```

2) [period]

① 백업 주기 범위는 3초~10초 (그 외의 경우 에러 처리)

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt 11
ssu_backup error:
[PERIOD] : 3~10
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt 2
ssu_backup error:
[PERIOD] : 3~10
```

3) '-r' 옵션

① '-r' 옵션은 단독적으로 쓰인다

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt -r -d
ssu_backup error:
option r cannot be used with other options
```

② 디몬 프로세스를 생성하지 않고, 이미 실행 중인 경우 종료

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt 3
junys@junys-VirtualBox:~/phw3$ process 2053 running as ssu_backup daemon.
./ssu_backup ./ex/target.txt -r
[0529 19:18:00] ssu_backup<pid:2053> exit
```

③ 백업파일이 존재하지 않는 경우 메시지 출력

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/targ.txt -r
file "./ex/targ.txt" doesn't exist
junys@junys-VirtualBox:~/phw3$
```

④ 해당 백업파일을 선택하면 원본 파일을 백업파일로 바꾼다.

```
junys@junys-VirtualBox:~/phw3$ cat ./ex/target.txt
target.txt
Modified!-00:16
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt -r
FILENAMEONLY : target.txt
        BACKUP FILE name :  2f686f6d652f6a756e79732f706877332f7461726765742e7478
74_0529001737
16_filename : 2f686f6d652f6a756e79732f706877332f7461726765742e747874, size : 54
Original name : target.txt
<target.txt backup list>
0. exit
1. target.txt__0529001614 [size:11]
2. target.txt__0529001622 [size:11]
3. target.txt__0529001630 [size:27]
4. target.txt__0529001638 [size:27]
5. target.txt__0529001646 [size:27]
6. target.txt__0529001654 [size:27]
input : 2
[target.txt]
target.txt
junys@junys-VirtualBox:~/phw3$ cat ./ex/target.txt
target.txt
```

① 백업 될 파일이 디렉터리가 아닌 경우 에러 처리

```
junys@junys-VirtualBox:~/phw3$ ./ssu_backup ./ex/target.txt -d 8
ssu_backup error
file "./ex/target.txt" must be directory with option d
```

② 디렉터리 내의 각 파일마다 스레드를 생성하여 백업한다

**백업 도중

4) '-d' 옵션

소스 코드

<ssu_backup.h>

```
typedef int OPTION;
typedef int MOD;

typedef struct _backupInfo {
        size_t size;
        time_t mtime;
        time_t backupt;
        char btime[10];
}backupInfo;
```

<ssu_backup.c>

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

```c
#include <syslog.h>
#include <string.h>
#include "ssu_backup.h"

#define FNAME 255

OPTION on  = 0;
OPTION c = 0;
OPTION r = 0;
OPTION m = 0;
OPTION n = 0;
OPTION d = 0;
int period;
char *backupName;
extern char *Filenameonly; //used to change its name(10->16 form)

void fnd_proc(); //find out if there is running ssu_backup daemon process
void kill_handler(int signo);
int start_daemon(); //start daemon
void option(int argc, char *argv[], char *filename); //handle option
int get_info(char *filename, backupInfo* info); //get backupfile information
void write_log(backupInfo info, int); //write backup file info. on log file
char* backup_name(char *filename, char *btime); //make backupfile's name
void ssu_copy(char *source, char *target, backupInfo info);
void option_r(char *filename);
void option_d(char *_dirname, char *_dir_hex, char *btime, time_t backupt);
int main(int argc, char *argv[])
{
        char *filename = malloc(123);
        int fd;
        backupInfo info;
        int mod;
        int i;
        char *dir_abs, *dir_hex, *curpath;
        int size;
        if (argc < 3) {
                fprintf(stderr, "argument is not enoughWn");
                exit(1);
        }
        backupName = malloc(FNAME);

        mkdir("backup", 0755);
        option(argc, argv, filename);
        if (!on) {
                mod = get_info(filename, &info);
                while(1) {
                        openlog("lpd", LOG_PID, LOG_LPR);
                        syslog(LOG_ERR, "open failed lpd %m");
                        closelog();
                        backupName = backup_name(filename, info.btime); //rename backup file
                        mod = get_info(filename, &info); //save backup file information
                        write_log(info, mod); //write on the log file
                        ssu_copy(filename, backupName, info); //backup file as 'backupName'
                        sleep(period);
                }
        }

        if(d) { //option d
                dir_abs = malloc(123);
                if (filename[0] == '/' || filename[0] == '~') { //filename is in absolute form
```

```
                        strcpy(dir_abs, filename);
                }
                else if (filename[0] == '.') { //filename is in relative form
                        curpath = malloc(100);
                        getcwd(curpath, 100);
                        chdir(filename);
                        getcwd(dir_abs, 123);
                        chdir(curpath);
                }
                else { //just the directory name
                        getcwd(curpath, 100);
                        sprintf(dir_abs, "%s/%s", curpath, filename);
                }
                size = strlen(dir_abs);
                dir_abs[size] = '₩0';
                strcat(dir_abs, "/");
                printf("dir : %s₩n", dir_abs);

                dir_hex = malloc(255);
                size = strlen(dir_abs);
                for (i = 0; i < size; i++)
                        sprintf(dir_hex+i*2, "%0x", dir_abs[i]);
                dir_hex[i*2] = '₩0';

                while(1) {
                        openlog("lpd", LOG_PID, LOG_LPR);
                        syslog(LOG_ERR, "open failed for %s lpd %m", dir_abs);
                        closelog();
                        get_info(dir_abs, &info);
                        option_d(dir_abs, dir_hex, info.btime, info.backupt);
                        sleep(period);
                }
        }
        else {
        }
        exit(0);
}

void option(int argc, char *argv[], char *filename) {
        int opt;
        backupInfo info;
        int pid;
        int mod;
        int fd;
        struct stat statbuf;

        while((opt = getopt(argc, argv, "mn:dcr")) != -1) {
                switch(opt) {
                        case 'm' : //back up only when backup file is modified
                                if (r) printf("ssu_backup error:₩noption r cannot be used with other
options₩n");
                                if (c) printf("ssu_backup error:₩noption c cannot be used with other
options₩n");
                                if (r||c) exit(1);
                                on++;
                                break;
                        case 'n' : //back up leaving latest N backupfiles
                                if (r) printf("ssu_backup error:₩noption r cannot be used with other
options₩n");
                                if (c) printf("ssu_backup error:₩noption c cannot be used with other
```

```
options₩n");
                                if (r||c) exit(1);
                                on++;
                                break;
                        case 'd' : //back up all files in the directory
                                if (r) printf("ssu_backup error:₩noption r cannot be used with other
options₩n");
                                if (c) printf("ssu_backup error:₩noption c cannot be used with other
options₩n");
                                if (r||c) exit(1);
                                d++;
                                on++;
                                break;
                        case 'c' : //print backup file and compare with original backup file
                                if(on) {
                                        fprintf(stderr, "ssu_backup error :₩n");
                                        fprintf(stderr, "option c cannot be used with another option₩n");
                                        exit(1);
                                }
                                on++;
                                c++;
                                break;
                        case 'r' : //print backup time, file size and let user choose backup files
                                if(on) {
                                        fprintf(stderr, "ssu_backup error :₩n");
                                        fprintf(stderr, "option c cannot be used with another option₩n");
                                        exit(1);
                                }
                                on++;
                                r++;

                                break;
                }
        }
        pid = getpid();
        fnd_proc();
        if (c) {
                strcpy(filename, argv[argc-1]);
                mod = get_info(filename, &info); //get file's information
                backupName = backup_name(filename, info.btime); //get backup file Name
                exit(0);
        }
        else if (r) {
                // print backuped-files and select the file user want to backup
                strcpy(filename, argv[argc-1]);
                mod = get_info(filename, &info);
                if (mod == -1) {
                        printf("file ₩"%s₩" doesn't exist₩n", filename);
                        exit(0);
                }
                backupName = malloc(FNAME);
                backupName = backup_name(filename, info.btime);
                option_r(filename);
                exit(0);
        }
        else {
                period = atoi(argv[argc-1]);
                if (period < 3 || period > 10) {
                        fprintf(stderr, "ssu_backup error:₩n");
                        fprintf(stderr, "[PERIOD] : 3~10₩n");
```

```
                        exit(1);
                }
                strcpy(filename, argv[argc-2]);
                if (d) {
                        stat(filename, &statbuf);
                        if (S_ISDIR(statbuf.st_mode) == 0) {
                                fprintf(stderr, "ssu_backup error₩n");
                                fprintf(stderr, "file ₩"%s₩" must be directory with option d₩n",
filename);
                                exit(1);
                        }
                }
                else if ((fd = open(filename, O_RDONLY)) < 0) {
                        //error if file does not exist
                        fprintf(stderr, "₩"%s₩" backup file does not exist₩n", filename);
                        exit(1);
                }
                else {
                        stat(filename, &statbuf);
                        if (S_ISREG(statbuf.st_mode) == 0) {
                                fprintf(stderr, "ssu_backup error₩n");
                                fprintf(stderr, "file ₩"%s₩" must be regular file₩n", filename);
                                exit(1);
                        }
                }
                if (start_daemon() < 0) {
                        fprintf(stderr, "ssu_daemon error₩n");
                        exit(1);
                }
        }
}
int start_daemon() {
        pid_t pid;
        int fd, maxfd;

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

        else if (pid != 0) {
                exit(0); //terminate parent process
        }

        pid = getpid();
        printf("process %d running as ssu_backup 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("/");
        fd = open("/dev/null", O_RDWR);
        dup(0);
        dup(0);
```

```
        return 0;
}
```

<kill_process.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include "ssu_backup.h"
int pid;

void kill_handler(int signo) { //handler function on SIGUSR
        int fd;
        time_t t;
        struct tm *lt;
        char *message = malloc(50);

        t = time(NULL);
        lt = localtime(&t);
        sprintf(message, "[%02d%02d %02d:%02d:%02d] ssu_backup<pid:%d> exit₩n", lt->tm_mon+1, lt->tm_mday,
lt->tm_hour, lt->tm_min, lt->tm_sec, pid+1);
        fd = open("./backup/backup_log.txt", O_RDWR| O_CREAT | O_APPEND, 0644);
        write(fd, message, strlen(message));
        close(fd);
        raise(SIGKILL);
}
void fnd_proc() { //see if ssu_backup daemon process is running
        DIR *dir;
        struct dirent **namelist;
        struct stat statbuf;
        int count, i;
        int fd, offset;
        int proc, temp;
        char *pid_str = malloc(8);
        char proc_path[20];
        char *message = malloc(20);

        if (signal(SIGUSR1, kill_handler) == SIG_ERR) { //call kill_handler() if signal SIGUSR1 is
received
                fprintf(stderr, "cannot handle SIGUSR1₩n");
                exit(EXIT_FAILURE);
        }
        if ((count = scandir("/proc", &namelist, NULL, alphasort)) == -1) { //scan directory proc if there
is existing ssu_backup daemon process
                fprintf(stderr, "/proc directory scan error ₩n");
                exit(1);
        }
        for (i = 0, proc = 0; i < count; i++) {
                if(strcmp("self", namelist[i]->d_name) == 0) {
                        continue;
                }
                sprintf(proc_path, "/proc/%s", namelist[i]->d_name);
                stat(proc_path, &statbuf);
                if (S_ISDIR(statbuf.st_mode) == 0) {
```

```
                        continue;
                }
                sprintf(proc_path, "/proc/%s/stat", namelist[i]->d_name);
                //debug)
                //printf("//%d/: %s₩n", getpid(), proc_path);
                offset = strlen(namelist[i]->d_name) + 2;

                if ((fd = open(proc_path, O_RDONLY)) < 0) {
                        continue;
                }
                else {
                        lseek(fd, offset, SEEK_SET);

                        read(fd, message, 20);
                        if (strncmp("ssu_backup", message, 10) == 0) {
                                pid = atoi(namelist[i]->d_name);
                                if (pid < getpid()) {
                                        kill(pid, SIGUSR1);
                                        return;
                                }
                        }
                        close(fd);
                }
        }
        for(i = 0; i < count; i++)
                free(namelist[i]);
        free(namelist);
}
```

<get_info.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <time.h>
#include <string.h>
#include "ssu_backup.h"

int get_info(char *filename, backupInfo* info) {
        int mod = 0;
        struct stat statbuf;
        time_t t;
        struct tm *lt;
        char temp[10];

        if (stat(filename, &statbuf) < 0) { //file has been deleted or doesn't exist
                mod = -1;
        }

        else if(statbuf.st_mtime != info->mtime) { //file has been modified
                info->mtime = statbuf.st_mtime;
                info->size = statbuf.st_size;
                mod = 1;
        }
        if ((t = time(NULL)) < 0) {
                fprintf(stderr, "time() call error₩n");
                exit(1);
        }
        if ((lt = localtime(&t)) == NULL) {
                fprintf(stderr, "localtime() call error₩n");
```

```
                exit(1);
        }

        info->backupt = t;
        sprintf(temp, "%02d%02d%02d%02d%02d", lt->tm_mon+1, lt->tm_mday, lt->tm_hour, lt->tm_min, lt-
>tm_sec);
        strcpy(info->btime, temp);

        return mod;
}
```

<backup_name.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

char* Filenameonly;
char* backup_name(char *filename, char* btime) {
        char *curpath = malloc(200);
        char *pathname = malloc(200);
        char temp[50];
        char *new_fname;
        int size; //size of file's name
        int offset = 0, i;

        Filenameonly = malloc(200);
        if(filename[0] == '/' || filename[0] == '~') { //absolute path
                strcpy(pathname, filename);
                for(i = 0; i < strlen(filename); i++) {
                        if(filename[i] == '/') offset = i;
                }
                strcpy(Filenameonly, pathname+offset+1);
        }
        else if (filename[0] == '.') { //relative path
                for(i = 0; i < strlen(filename); i++) {
                        if(filename[i] == '/') {
                                offset = i;
                                strncpy(temp, filename, offset);
                        }
                }
                strcpy(Filenameonly, filename+offset+1);
                getcwd(curpath, 200);
                chdir(temp);
                getcwd(pathname, 200);
                chdir(curpath);
                size = strlen(pathname);
                sprintf(pathname+size, "/%s", filename+offset+1);
        }
        else { //just the file name
                Filenameonly = filename;
                getcwd(curpath, 1024);
                sprintf(pathname, "%s/%s", curpath, filename);
        }

        if ((size = strlen(pathname)) > 122) {
                fprintf(stderr, "ssu_backup error:₩nfilename is too long!₩n");
                exit(1);
        }
        new_fname = (char *)malloc(255);
```

```
        for (i = 0; i < size; i++) {
                sprintf(new_fname+i*2, "%0x", pathname[i]);
        }
        i = i*2;
        sprintf(new_fname+i, "_%s", btime);

        return new_fname;
}
```

<write_log.c>

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

extern char* Filenameonly;
char* logfile = "./backup/backup_log.txt";
void write_log(backupInfo info, int state) {
        char message[1024];
        int fd, size;

        struct tm *lt;
        char b_date[4];
        char b_h[2];
        char b_m[2];
        char b_s[2];

        if((fd = open(logfile, O_RDWR | O_CREAT | O_APPEND, 0644)) < 0) {
                fprintf(stderr, "open %s error₩n", logfile);
                exit(1);
        }
        //lseek(fd, 0, SEEK_END);

        if ((info.backupt = time(NULL)) < 0) {
                fprintf(stderr, "time() call error₩n");
                exit(1);
        }

        if ((lt = localtime(&info.backupt)) == NULL) {
                fprintf(stderr, "localtime() call error₩n");
                exit(1);
        }
        sprintf(message, "[%02d%02d %02d:%02d:%02d] %s ", lt->tm_mon+1, lt->tm_mday, lt->tm_hour, lt-
>tm_min, lt->tm_sec, Filenameonly);

        if (state == 1) { //file is modified
                message[strlen(message)] = '₩0';
                strcat(message, "is modified");
        }
        else if (state == -1) { //file is deleted
                message[strlen(message)] = '₩0';
                strcat(message, "is deleted₩n");
                size = strlen(message);
                write(fd, message, size);
```

```
                kill(getpid(), SIGKILL);
                exit(1);
        }

        else;
        if((lt = localtime(&info.mtime)) == NULL) {
                fprintf(stderr, "localtime() call error₩n");
                exit(1);
        }

        sprintf(message+strlen(message), " [size:%ld/mtime:%02d%02d %02d:%02d:%02d]₩n", info.size, lt-
>tm_mon+1, lt->tm_mday, lt->tm_hour, lt->tm_min, lt->tm_sec);

        size = strlen(message);
        write(fd, message, size);
}
```

<ssu_cp.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <utime.h>
#include <string.h>
#include "ssu_backup.h"

void ssu_copy(char *source, char *backupfile, backupInfo info) {
        int fd_src, fd_tar;
        struct stat stat_src, stat_tar;
        struct utimbuf time_buf;
        char *target;
        char c;

        target = malloc(260);
        strcpy(target, "./backup/₩0");
        strcat(target, backupfile);

        if ((fd_src = open(source, O_RDONLY)) < 0) {
                fprintf(stderr, "open error for %s₩n", source);
                exit(1);
        }

        if ((fd_tar = open(target, O_WRONLY | O_CREAT, 0644)) < 0) { //create backup file
                fprintf(stderr, "open error for %s₩n", target);
                exit(1);
        }

        if (stat(target, &stat_tar) < 0) {
                fprintf(stderr, "stat error for %s₩n", target);
                exit(1);
        }
        while(read(fd_src, &c, 1) > 0) { //copy backup file
                write(fd_tar, &c, 1);
        }

        time_buf.actime = stat_tar.st_atime;
        time_buf.modtime = info.mtime;
        utime(target, &time_buf);
```

}

<option_r.c>

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

extern char* backupName;
extern char* Filenameonly;

void option_r(char *filename) {
        struct dirent **namelist;
        struct stat statbuf;
        int fd, b_fd;
        int count, i, k=0, num, start = 0;
        char *_filename; //filename in hexadecimal
        char *backupf;
        char _btime[15][12]; //backup time
        char *line; //get line by line from backup_log.txt
        char c;
        int size;

        line = malloc(165);
        _filename = malloc(122);
        backupf = malloc(263);
        i = 0;
        memset(_filename, '₩0', 122);
        while(backupName[i] != '_') { //filename_16
                _filename[i] = backupName[i];
                i++;
        }
        size = strlen(_filename);
        if ((count = scandir("./backup", &namelist, NULL, alphasort)) == -1) {
                fprintf(stderr, "/backup directory scan error₩n");
                exit(1);
        }
        for (i = 0; i < count; i++) {
                if (strncmp(_filename, namelist[i]->d_name, size) == 0) { //backup file exists
                        if (k == 0) {
                                printf("0. exit₩n");
                                k++;
                        }
                        strcpy(_btime[k], (namelist[i]->d_name)+strlen(_filename));
                        _btime[k][11] = '₩0';
                        sprintf(backupf, "./backup/%s%s", _filename, _btime[k]);
                        stat(backupf, &statbuf);
                        sprintf(line, "%d. %s_%s [size:%ld]", k++, Filenameonly, _btime[k],
statbuf.st_size);
                        printf("%s₩n", line);
                }
                else continue;
        }
        printf("input : ");
        scanf("%d", &num);
        sprintf(backupf, "./backup/%s%s", _filename, _btime[num]);
```

```
        if (num == 0) exit(0);
        else {
                printf("[%s]₩n", Filenameonly);
                fd = open(filename, O_RDWR | O_TRUNC, 0644);
                b_fd = open(backupf, O_RDONLY);
                while(read(b_fd, &c, 1) > 0) {
                        write(fd, &c, 1);
                        printf("%c", c);
                }
        }

        for (i = 0; i < count; i++)
                free(namelist[i]);
        free(namelist);
        free(line);
        free(_filename);
        free(backupf);
        exit(0);
}
```

<option_d.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <time.h>
#include <pthread.h>
#include <string.h>
#include "ssu_backup.h"

extern char* logfile;
extern char* Filenameonly;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *backup_thread(void *arg);
void write_log(backupInfo info, int state);
void ssu_copy(char *source, char *backupfile, backupInfo info);
int get_info(char *filename, backupInfo* info);

typedef struct _thread_data {
        char dirname[122];
        char dir_hex[244];
        char *btime;
        time_t backupt;
} thread_data;

        thread_data data;
void option_d(char *_dirname, char *_dir_hex, char *btime, time_t backupt) {
        pthread_t tid[500];
        int cnt_dir, cnt, size, size_hex;
        struct dirent **namelist;
        struct stat statbuf;
        char dirname[122];
        char dir_hex[244];
        char temp[122];
        int status;
        int i;
```

```
        strcpy(dirname, _dirname);
        strcpy(dir_hex, _dir_hex);
        if ((cnt_dir = scandir(dirname, &namelist, NULL, alphasort)) == -1) { //scan directory
                fprintf(stderr, "%s directory scan error₩n", dirname);
                exit(1);
        }
        size = strlen(dirname);
        size_hex = strlen(dir_hex);
        dirname[size] = '₩0';
        dir_hex[size_hex] = '₩0';
        for (cnt = 0; cnt < cnt_dir; cnt++) { //scan files in directory user entered
                memset(dirname+size, '₩0', 122-size);
                memset(dir_hex+size_hex, '₩0', 244-size);
                if (strcmp(namelist[cnt]->d_name, ".") == 0 || strcmp(namelist[cnt]->d_name, "..") == 0)
                        continue;
                strcat(dirname, namelist[cnt]->d_name); //"[directory] + filename"


                //make backupfile name in 16 form
                memset(temp, '₩0', 122);
                for (i = 0; i < strlen(namelist[cnt]->d_name); i++) {
                        sprintf(temp+i*2, "%0x", namelist[cnt]->d_name[i]);
                }
                temp[strlen(temp)] = '₩0';
                strcat(dir_hex, temp); //dir_hex : directory/file in hexadecimal form
                stat(dirname, &statbuf);
                if (S_ISDIR(statbuf.st_mode)) { //if there is sub directory
                        dirname[strlen(dirname)] = '/';
                        dir_hex[strlen(dir_hex)] = '₩0';
                        strcat(dir_hex, "2f");
                        dir_hex[strlen(dir_hex)] = '₩0';
                        option_d(dirname, dir_hex, btime, backupt);

                        continue;
                }

                dir_hex[strlen(dir_hex)] = '_';
                strcat(dir_hex, btime);
                if(strlen(dir_hex) > 255) {
                        fprintf(stderr, "ssu_backup error₩n");
                        fprintf(stderr, "backup file [%s] name cannot be over 255 bytes₩n", namelist[cnt]-
>d_name);
                        continue;
                }
                        //copy data to use them in new thread
                memset(data.dirname, '₩0', 122);
                strcpy(data.dirname, dirname);
                memset(data.dir_hex, '₩0', 244);
                strcpy(data.dir_hex, dir_hex);
                data.btime = btime;
                Filenameonly = namelist[cnt]->d_name;
                data.backupt = backupt;

                if(pthread_create(&tid[cnt], NULL, backup_thread, (void *)&data) != 0) {
                        fprintf(stderr, "pthread_create error₩n");
                        continue;
                }
                pthread_join(tid[cnt], NULL);
        }
//      status = pthread_mutex_destroy(&mutex);
```

```
        return;
}

void *backup_thread(void *arg) {
        thread_data data = *(thread_data *)arg;
        backupInfo info;
        int mod;
        pthread_mutex_lock(&mutex);
        mod = get_info(data.dirname, &info);

        if (mod == -1) mod = 1; //file has been deleted -> write on log file "modified"
        else if (mod == 1) mod = 0;
        strcpy(info.btime, data.btime);
        info.backupt = data.backupt;
        write_log(info, mod);
        if (mod == 1) { // deleted file
        }
        else {
                ssu_copy(data.dirname, data.dir_hex, info);
        }
        pthread_mutex_unlock(&mutex);
//      pthread_exit(NULL);
}
```