Embedded System Software 과제 1

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20211584, 장준영

개발기간: 2024. 4. 9. -2024. 4. 16.

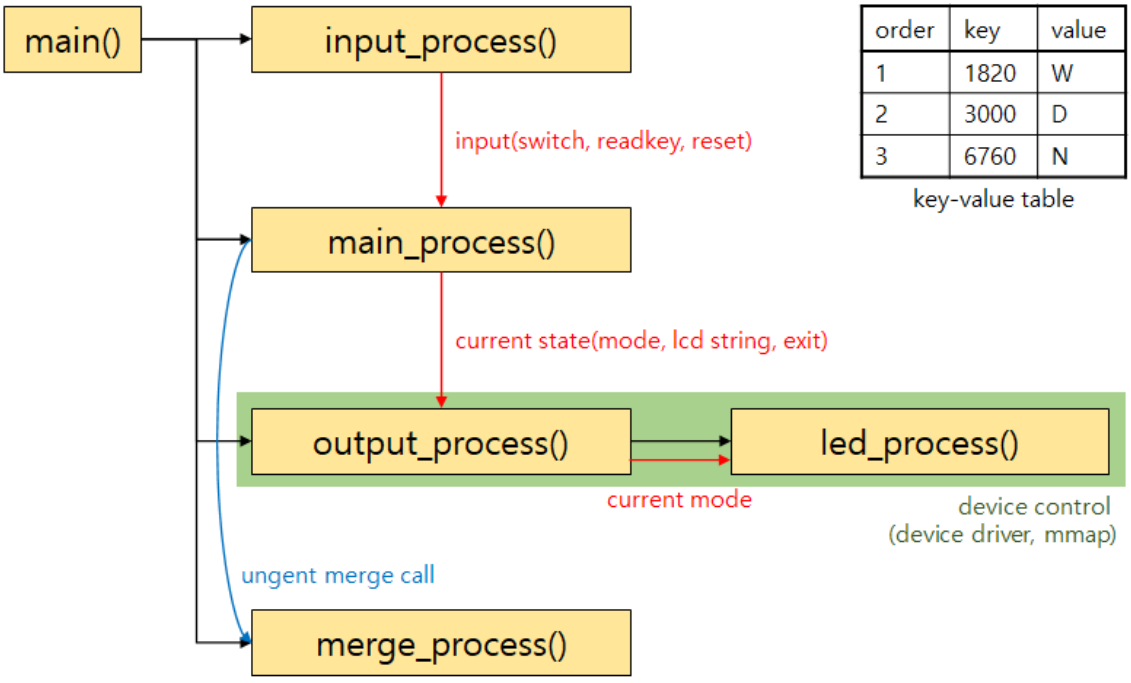
최 종 보 고 서

I. 개발 목표

이번 프로젝트에서는 (1)device driver 및 mmap을 이용해 device를 제어하고, (2)message queue와 shared memory를 이용한 IPC를 통해 여러 프로세스가 소통하는 (3)간단한 NoSQL Key-Value Storage를 구현한다.

II. 개발 범위 및 내용

가. 개발 범위



order	key	value
1	1820	W
2	3000	D
3	6760	N

key-value table

프로그램의 설계도는 위와 같다. main에서 총 네 개의 프로세스를 fork 하고 각각 다음과 같은 루틴을 수행한다.

[1] input process : device에서 들어오는 input을 관리해 main process로 전송한다. switch, reset, readkey input을 device driver를 통해 입력받고, 이를 가공해 message queue로 전송

한다.

[2] main process : device input을 통해 들어온 사용자의 요청을 본격적으로 처리하는 프로세스이다. 모드 변환, key 및 value 입력, 모드 동작 요청 등을 처리하고 memory table과 storage table에 적용한다. 이후 그 결과를 사용자가 확인할 수 있도록, output process로 device에 띄워야 할 정보를 전송한다. 만약 memory table이 한계 크기를 넘어서려 한다면, 자동으로 memory table 정보와 함께 storage table로 flush 하고, 파일이 최대 크기인 세 개가 된다면 merge process로 merge call을 보낸다.

[3] output process : main process에서 보낸 정보를 단순히 output device에 출력해준다. 이 과정에서 LED는 동시에 작동하려면 다른 execution flow를 가져야 하기 때문에, 이 프로세스 내에서 led process를 fork 하여 사용한다. 우선 led process로 main process로부터 전달받은 현재의 모드 정보를 보내고, device driver를 통해 LCD와 FND에 알맞은 출력을 띄운다. led process는 모드 정보에 따라 mmap을 통해 알맞은 LED 출력을 띄운다.

[4] merge process : 상시 돌아가면서 사용자가 merge mode에서 요청을 했거나, 파일 개수가 세 개가 되었는지 체크한다. 만약 조건이 만족된다면 merge 루틴을 수행한다.

나. 개발 내용

- fork

한 프로세스에서 다른 프로세스를 생성한다. 기존 프로세스는 parent process, 생성된 프로세스는 child process로 fork가 수행될 때 까지의 프로세스 정보를 그대로 복사한다. 이번 프로젝트에서 개별적인 execution flow를 가져야 하는 여러 루틴에 대해, 각각 fork로 프로세스를 생성해 운영할 예정이다.

- message queue

위에서 설명한 fork를 통해 새로운 프로세스를 생성하면, 서로 다른 프로세스끼리 정보를 주고 받아야 하는 일이 생긴다. 이를 IPC(Inter-Process Communication)을 통해 할 수 있는

데, 그 중 message를 queue에 삽입하는 방식으로 송수신 하는 방법이 message queue이다. FIFO 방식으로 동작하는 queue가 커널 내부에서 관리되고, 유저는 코드에서 message queue API를 통해 연결, 삽입, 추출 등을 할 수 있다.

- shared memory

IPC의 다른 방법으로, 공유된 메모리 영역을 선언해 해당 영역에 쓰고 읽는 방식이다. 커널은 보호되는 커널 영역 메모리를 갖고 있고, 이 영역에는 유저가 관여할 수 없다. 그 커널 영역의 일부를 유저가 shared memory로 사용하겠다고 커널에 요청하면, 커널은 그 영역을 shared memory 영역으로 할당해 명령어를 통해 접근할 수 있도록 한다. 추가적인 할당이나 복사가 없기 때문에 가장 빠르지만 무결성을 위해 동기화에 큰 신경을 써야 한다.

- device driver

device driver는 커널에서 사용되는 모듈의 일종으로, 모듈을 추가하고 나선 device file에 단순히 read/write 하는 것 만으로 device를 관리할 수 있다. 우선 device driver(.ko)를 insmod 하여 커널에 추가하고, mknod를 통해 device file을 생성한다. 이후 유저가 해당 파일을 통해 device를 관리할 수 있다. (device driver / module에 관한 설명은 해당 프로젝트와 문맥이 맞지 않아 작성하지 않는다.)

- mmap

device driver를 사용하지 않고 device control 하는 방법으로, 유저가 mmap 명령어를 통해 device register를 매핑하여 직접 읽고 쓸 수 있다. 이번 프로젝트에선 LED를 mmap을 통해 관리한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

4/9 ~ 4/11	프로그램을 구성하는 프로세스, IPC에 이용할 메시지 필드 등 기초적

	인 부분을 설계하였다.
4/12 ~ 4/14	**프로세스별로 본격적인 코드를 작성하였다.**
4/15 ~ 4/16	**merge call이 전송되지 않는 부분, 파일 생성이 규칙적으로 되지 않는 부분, LCD에 가비지가 출력되는 부분, 다른 프로세스가 동작을 마무리하기 이전에 main process가 먼저 exit 되는 부분 등을 디버깅하였다.**

나. 개발 방법

[1] main

우선 main에서 IPC를 위해 초기화한다(ipc_init). 이후 필요한 네 개의 프로세스를 fork 하고 wait 한다. 모든 프로세스의 reaping이 마무리되면 ipc_init에서 초기화한 것을 연결 해제한다 (ipc_clt).

[2] input process

input_process()가 실행되면, 우선 input message queue와 연결하고 사용할 device file을 모두 open 한다. 이후 readkey에서 BACK 입력이 들어오기 전까지 루프를 돌며, input device의 입력을 가공하여 input message queue로 전송한다. BACK 입력이 들어오면 input message queue에 해당 입력을 알리고 종료한다. 이후 이 입력은 모든 프로세스로 다시 전송되어 프로그램이 종료되도록 한다.

[3] main process

가장 먼저 storage file이 저장될 디렉토리를 생성하고, 이후 input/output message queue와 연결한다. memory table과 이를 관리할 준비를 마치면 input message queue에 BACK 요청이 오기 전까지 루프를 돌며 실질적인 사용자의 요청(input)을 처리하고 device에 출력(output)한다. 이를 위해 모드 정보 및 요청을 처리하기 위한 루틴 함수(main_put,

main_flush, main_get 및 편의성 루틴)을 사용한다. BACK 요청이 오면 output process와 merge process로 이를 전달하고, memory table의 모든 값을 storage file로 flush하고 종료한다.

[4] output process

output device를 편리하게 관리하기 위해, device.(c/h)를 작성하였다. 이 파일에는 사용하는 LCD, FND, LED, motor를 관리하는 함수가 작성되어 있다. LED를 제외한 모든 device는 device file에 인자를 작성하는 방식이고, LED는 mmap을 통해 직접 값을 작성하는 방식이다. output process에선 output message queue와 led message queue에 연결하고 led process를 fork 한다. 이후 main process로부터 전달받은 정보를 토대로 device.h에 정의된 함수를 사용해 output device에 값을 띄워준다. BACK 요청이 오면 led process에도 전달해 reap 하고 종료한다.

[5] merge process

main process로부터 merge call을 받을 shared memory에 연결하고, BACK 요청 전까지 루프를 돌며 처리한다. merge가 수행되는 경우는 (1)main process에서 merge call이 올 때, (2)storage file이 세 개가 되었을 때로, 상황에 맞게 동작을 수행한다. 좀 더 중점을 둔 것은 storage file의 개수로, 하나일 때, 두 개일 때, 세 개일 때를 나눠서 동작을 처리했다. 이는 storage file의 이름을 항상 1.st와 2.st로 하고, 2.st에 더 최신 정보를 담아 put 요청을 용이하게 하기 위함이다. BACK 요청이 오면 main에서 memory table flush를 진행하고 있을 수 있기 때문에 3초 가량 sleep하고, 이후 storage file이 세 개가 된다면 merge 하고 종료한다.

IV. 연구 결과

모든 동작을 성공적으로 구현하였다. 편의를 위해 GET 모드와 PUT 모드는 결과가 LCD에 출력되고 나서, switch의 아무 버튼을 누르면 입력 받았던 것들이 모두 사라지고 다시 처음부터 입력을 받는 초기 모드로 돌아간다. MERGE 모드 역시 결과가 출력되고 나서 reset 버튼을

다시 누르면 merge call을 받는 초기 모드로 돌아간다.

[1] main

```
#define INPUT_KEY (key_t)0x1A1A
#define OUTPUT_KEY (key_t)0x2B2B
#define MERGE_KEY (key_t)0x3C3C
#define LED_KEY (key_t)0x4D4D
```

```
/* key-value structure */
struct table_elem
{
    int order;
    int key;
    char val[6];
};
```

```
struct output_msg // msgq
{
    long mtype;
    bool _BACK_;
    bool _RESET_;

    enum mode cur_mode;
    int fnd;
    char lcd2[16];
    bool motor;
};
#define OUTPUT_MSG_SIZE sizeof(struct output_msg) - sizeof(long)
```

main.h는 모든 코드가 포함하므로, IPC에서 사용할 고유 키 값, 메시지 타입, key-value table 타입 등을 선언하였다.

```
/* main : modes */
enum mode
{
    PUT_INIT = 0,
    PUT_KEY,
    PUT_VAL,
    PUT_REQ,
    GET_INIT,
    GET_KEY,
    GET_REQ,
    MERGE_INIT,
    MERGE_REQ
};
```

특히 모드를 정의하는 enum mode 타입은 위와 같다. 초기 상태인 INIT과 값을 입력받는

KEY/VAL, 요청을 보낸 상태인 REQ이 있다.

```c
int main()
{
    ipc_init();

    int input_pid, output_pid, merge_pid, main_pid;
    /* INPUT PROCESS */
    input_pid = fork();
    if (input_pid == 0)
    {
        input_process();
        _exit(0);
    }

    // ...

    waitpid(input_pid, NULL, 0);
    waitpid(output_pid, NULL, 0);
    waitpid(merge_pid, NULL, 0);
    waitpid(main_pid, NULL, 0);
    ipc_ctl();

    printf("Thank you!\n");

    return 0;
}
```

ipc_init에는 msgget, shmget의 IPC를 초기화하는 루틴이 들어있어, 프로그램 내에서 사용할 전체 IPC queue 및 memory를 요청한다. 이후 네 개의 프로세스를 fork/wait 하고, IPC를 연결 해제하는 msgctl, shmclt 루틴이 포함된 ipc_ctl을 호출한다.

[2] input process

```c
void input_process()
{
    printf("BEGIN : input process\n");
    int input_q = msgget(INPUT_KEY, 0666 | IPC_CREAT);
    if (input_q == -1)
    {
        perror("ERROR(input_process.c) : msgget failed.\n");
        _exit(-1);
    }

    int readkey_fd = open("/dev/input/event0", O_RDONLY | O_NONBLOCK),
        switch_fd = open("/dev/fpga_push_switch", O_RDWR),
        reset_fd = open("/dev/fpga_dip_switch", O_RDWR);
    if (readkey_fd == -1 || switch_fd == -1 || reset_fd == -1)
    {
        perror("ERROR(input_process.c) : open failed.\n");
        _exit(-1);
    }

    struct input_msg *input = (struct input_msg *)malloc(sizeof(struct input_msg));
    input->mtype = 1;
    input->_BACK_ = false;
```

우선 사용할 message queue에 연결하고 device file을 open 한다. 이후 메시지를 받을 변수를 미리 선언하고 초기화한다.

```
    while (!input->_BACK_)
    {
        input_readkey(readkey_fd, input);
        input_reset(reset_fd, input);
        input_switch(switch_fd, input);
        if (msgsnd(input_q, (void *)input, INPUT_MSG_SIZE, 0) == -1)
        {
            perror("ERROR(input_process.c) : msgsnd failed.\n");
            _exit(-1);
        }
        usleep(50000);
    }

    close(readkey_fd);
    close(switch_fd);
    close(reset_fd);

    free(input);

    printf("END : input process\n");
}
```

BACK 요청이 올 때까지 동작하는 루프 내에선 세 종류의 입력을 받고, input에 가공된 메시지를 삽입해 message queue로 전송한다. BACK 요청이 와 input->_BACK_이 true가 되면 루프를 빠져나와 device file을 close 하고 종료한다.

```
void input_reset(int fd, struct input_msg *msg)
{
    unsigned char buf = 0;
    if (read(fd, (void *)&buf, 1) < 1)
    {
        perror("ERROR(input_process.c) : reset read failed.\n");
        _exit(-1);
    }
    if (buf == 0)
        msg->reset_input = true;
    else
        msg->reset_input = false;
}
```

input 신호를 받는 함수는 모두 비슷하기 때문에 input_reset만 작성한다. device file로부터 정해진 크기만큼 read 하여 값을 확인하고, 그 값에 따라 input 메시지의 필드에 값을 채운다.

[3] main process

```c
void main_process()
{
    printf("BEGIN : main process\n");

    /* STORAGE FILE DIRECTORY */
    struct stat st = {0};
    if (stat(STORAGE_DIR, &st) == -1)
    {
        if (mkdir(STORAGE_DIR, 0755) == -1)
        {
            perror("ERROR(main_process.c) : mkdir failed.\n");
            _exit(-1);
        }
    }

    int input_q = msgget(INPUT_KEY, 0666 | IPC_CREAT);
    int output_q = msgget(OUTPUT_KEY, 0666 | IPC_CREAT);
    if (input_q == -1 || output_q == -1)
    {
        perror("ERROR(main_process.c) : msgget/shmget failed.\n");
        _exit(-1);
    }
    struct input_msg *input = (struct input_msg *)malloc(sizeof(struct input_msg));
    struct output_msg *output = (struct output_msg *)malloc(sizeof(struct output_msg));
    output->mtype = 1;
    output->_BACK_ = false;
    output->_RESET_ = false;
```

storage file을 저장할 디렉토리를 우선 생성한다(/data/local/tmp/storage_files/). 이후 입력을 받을 input message queue, 출력을 보낼 output message queue에 연결하고 메시지 변수를 초기화한다.

```c
enum mode cur_mode = PUT_INIT;
char key_buf[4], val_buf[16], val_input_buf[1000], merge_lcd[17];
int val_input_buf_top = 0;
int prev_switch_input = 255, prev_reset_input = 0, term_counter = 0;
init_buf(key_buf, val_buf, val_input_buf, &val_input_buf_top);

struct table_elem mem_table[3];
int put_order = 1;
int mem_table_cnt = 0;
```

이후 main process 운영에 필요한 정보(현재 모드, 입력받은 key/value, 이전 switch/reset input, 1이 얼마나 오래 눌렸는지 체크하는 term counter 등)를 선언하고, memory table을 선언한다. put_order는 record가 삽입된 순서, mem_table_cnt는 메모리 테이블에 현재 몇 개의 record가 있는지 저장한다.

```c
/* URGENT REQUESTS */
if (input->_BACK_)
{
    output->_BACK_ = true;
    if (msgsnd(output_q, output, OUTPUT_MSG_SIZE, 0) == -1)
    {
        perror("ERROR(main_process.c) : msgsnd failed\n");
        _exit(-1);
    }
    main_flush(mem_table, mem_table_cnt, true, false);
    usleep(5000000);
    break;
}
if (input->readkey_input == READKEY_VOL_UP)
{
    init_buf(key_buf, val_buf, val_input_buf, &val_input_buf_top);
    memset(merge_lcd, ' ', sizeof(merge_lcd));
    mode_up(&cur_mode);
    output->_RESET_ = true;
    if (msgsnd(output_q, output, OUTPUT_MSG_SIZE, 0) == -1)
    {
        perror("ERROR(main_process.c) : msgsnd failed\n");
        _exit(-1);
    }
    continue;
}
if (input->readkey_input == READKEY_VOL_DOWN)
{
    // ...
}
```

루프 내부에선 input 메시지를 받고 처리한다. 먼저 readkey에 의해 BACK, VOL+, VOL- 등의 긴급한 요청이 왔을 때의 경우를 가장 먼저 처리한다. BACK 요청이 들어온 경우 output process로 해당 요청을 전달하고, memory table을 flush 한 후 루프를 빠져나온다.

```c
void main_flush(struct table_elem *mem_table, int mem_table_cnt, bool _BACK_, bool _CALL_)
{
    // ...

    int merge_q = shmget(MERGE_KEY, MERGE_MSG_SIZE, 0666 | IPC_CREAT);
    struct merge_msg *merge = (struct merge_msg *)shmat(merge_q, NULL, 0);
    merge->_BACK_ = _BACK_;
    merge->_CALL_ = _CALL_;
    usleep(100000);
    shmdt(merge);
}
```

이때 main_flush에는 merge process와 IPC 하는 루틴이 있기 때문에, _BACK_ 인자를 true로 설정하면 merge process로도 종료 요청이 전달된다. VOL+, VOL-를 통해 모드 변경 요청이 온다면 초기화해야 할 값들을 초기화하고 모드를 변경한다. output process에게 모든 device를 초기화하라는 명령도 전송한다.

```
switch (cur_mode)
{
// ...
case PUT_VAL:
{
    if (switch_input == 46)
    {
        main_put(key_buf, val_buf, mem_table, &mem_table_cnt, &put_order);
        cur_mode = PUT_REQ;
    }
    if (term_counter > 10) /* 1~ */
        init_buf(NULL, val_buf, val_input_buf, &val_input_buf_top);
    if (switch_input != 255 && prev_switch_input == 255) /* (#)123456789 */
    {
        val_input_buf[val_input_buf_top++] = (char)switch_input;
    }
    if (switch_input == 255 && prev_switch_input != 255)
    {
        term_counter = 0;
        if (val_input_buf_top != 0)
            val_input_buf[val_input_buf_top++] = '#';
    }
    else if (switch_input == 1)
        term_counter++;
    val_interpret(val_buf, val_input_buf, val_input_buf_top);

    output->cur_mode = cur_mode;
    output->fnd = my_atoi(key_buf);
    memcpy(output->lcd2, val_buf, 16);

    if (reset_input && prev_reset_input == 0) /* reset */
        cur_mode--;
    break;
}
```

이후 현재 모드에 따라 입력을 처리하는 루틴이 있다. input을 통해 key/value 값을 정하는 부분이 많은데, 모두 과정이 일정하기 때문에 PUT 모드에서 value를 입력하는 과정만 작성하겠다. switch_input이 46인 경우 4와 6이 동시에 눌렸으므로 main_put 함수를 호출하고 모드를 PUT_REQ로 변경한다. term_counter가 10 이상이 된 경우 1이 꾹 눌렸으므로 버퍼들을 초기화한다. switch_input이 255가 아니고 prev_switch_input이 255인 경우 switch가 눌리지 않고 있다가 새로 눌린 경우이므로 val_input_buf에 값을 저장한다. switch_input이 255이고 prev_switch_input이 255가 아닌 경우 더 이상 switch가 눌리지 않고 있는 것이므로 term_counter를 초기화하고 아무것도 눌리지 않았다는 뜻의 '#'를 val_input_buf에 저장한다. switch_input이 1인 경우 1이 눌리고 있으므로 term_counter를 1 증가시킨다. 지금까지

val_input_buf에 저장된 값을 해석해서 value string으로 바꾸기 위해 val_interpret을 호출한다. 이 과정에서 2#6#9#9#2#2#1#2#는 AMXB2로 변환되어 val_buf에 저장된다. 모든 처리 과정을 마치면 output 메시지에 필요한 값을 넣어 전송할 준비를 마친다. reset_input이 true이고 prev_reset_input이 false인 경우 reset이 눌린 것이므로 현재 모드를 1 낮춘다(PUT_VAL->PUT_KEY).

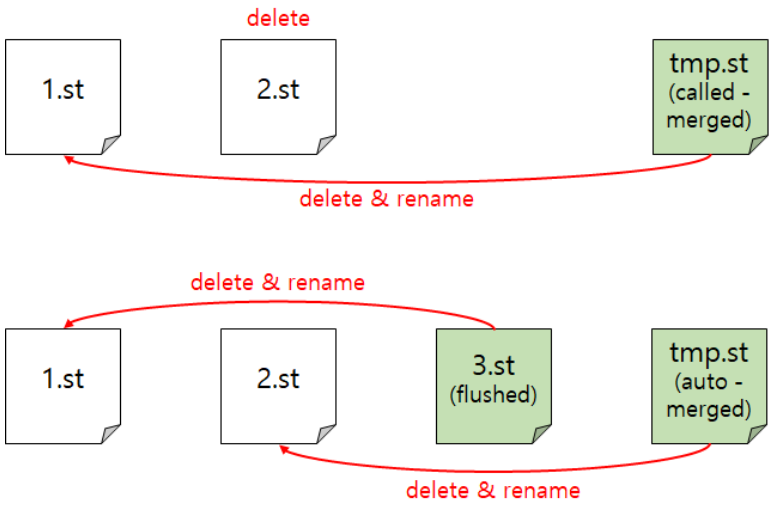
```
case MERGE_REQ:
{
    output->cur_mode = cur_mode;
    output->fnd = 0;

    DIR *dir = opendir("storage_files");
    struct dirent *entry;

    if (dir == NULL)
        _exit(-1);

    int fileCount = 0;
    while ((entry = readdir(dir)) != NULL)
    {
        if (entry->d_name[0] != '.')
            fileCount++;
    }
    closedir(dir);

    if (fileCount == 1)
    {
        memcpy(merge_lcd, "ONLY ONE FILE    ", 16);
    }
```



다른 특수한 루틴인 MERGE_REQ 루틴이다. 우선 내가 설계한 storage file 관리 과정은 아래 이미지와 같다. 파일이 두 개일 때 유저가 merge call을 통해 merge를 요청하면 tmp.st 파일

에 1.st와 2.st를 merge 한 결과를 저장한 후 두 파일을 모두 삭제하여 tmp.st를 1.st로 rename 한다. memory table이 꽉 차 flush 되어 3.st 파일이 생성되면 자동으로 merge를 진행하는데, 이땐 tmp.st에 1.st와 2.st를 merge 한 결과를 저장한 후 3.st는 1.st로, tmp.st는 2.st로 rename 한다. 이렇게 하면 항상 2.st에 최신 정보를 담을 수 있다. 유저가 reset 버튼을 통해 merge call을 하면 MERGE_REQ 모드가 되는데, 이때 우선 storage_files 디렉토리에 파일이 몇 개 있는지 체크한다. 파일이 하나밖에 없으면 merge 할 수 없으므로 에러 메시지를 LCD에 띄운다.

```
else if (fileCount == 2)
{
    main_flush(mem_table, mem_table_cnt, false, true);
    int merge_q = shmget(MERGE_KEY, MERGE_MSG_SIZE, 0666 | IPC_CREAT);
    struct merge_msg *merge = (struct merge_msg *)shmat(merge_q, NULL, 0);
    while (!merge->merge_end)
        usleep(250000);
    merge->merge_end = false;
    shmdt(merge);
    char str[25];
    memset(str, ' ', 25);
    sprintf(str, "1 st %d                ", countLinesInFile("storage_files/1.st"));
    memcpy(merge_lcd, str, 16);
}

cur_mode = MERGE_INIT;
```

만약 파일이 두 개라면 _CALL_ 인자를 true로 한 main_flush를 호출하고 merge process에서 merge가 끝날 때까지 기다린다. merge가 끝나면 1.st 파일에 그 결과가 저장되므로, 이 파일의 라인 수를 LCD에 띄워준다. 이후 자동으로 MERGE_INIT 모드로 돌아간다.

```
        if (msgsnd(output_q, output, OUTPUT_MSG_SIZE, 0) == -1)
        {
            perror("ERROR(main_process) : msgsnd failed.\n");
            _exit(-1);
        }

        prev_switch_input = switch_input;
        prev_reset_input = reset_input;
        free(output);
    }

    free(input);
    free(output);

    printf("END : main process\n");
}
```

루프 마지막엔 기록된 정보를 바탕으로 output message queue에 메시지를 삽입하면서 main_process가 종료된다.

```c
void main_put(char *key_buf, char *val_buf, struct table_elem *mem_table,
              int *mem_table_cnt, int *put_order)
{
    if (*mem_table_cnt == 3)
    {
        main_flush(mem_table, *mem_table_cnt, false, false);
        *mem_table_cnt = 0;
    }

    struct table_elem tmp;
    tmp.order = (*put_order)++;
    tmp.key = my_atoi(key_buf);
    int i;
    for (i = 0; i < 5; i++)
    {
        tmp.val[i] = val_buf[i];
    }
    tmp.val[5] = '\0';

    mem_table[*mem_table_cnt] = tmp;

    *mem_table_cnt = *mem_table_cnt + 1;
}
```

main process의 핵심 함수는 main_put, main_get, main_flush가 있다. 우선 main_put에선 먼저 memory table이 꽉 찼는지 확인한다. 만약 꽉 찼다면 memory table을 flush 한다. 이후 전달받은 값을 토대로 필드를 채워 할당해준다.

```c
dir = opendir(dirPath);
if (dir == NULL)
{
    perror("opendir");
    return;
}

while ((entry = readdir(dir)) != NULL)
{
    if (!strcmp(entry->d_name, "1.st"))
        file1Exists = 1;
    else if (!strcmp(entry->d_name, "2.st"))
        file2Exists = 1;
    else if (!strcmp(entry->d_name, "3.st"))
        file3Exists = 1;
}

closedir(dir);

if (!file1Exists)
    snprintf(filePath, sizeof(filePath), "%s/1.st", dirPath);
else if (!file2Exists)
    snprintf(filePath, sizeof(filePath), "%s/2.st", dirPath);
else if (!file3Exists)
    snprintf(filePath, sizeof(filePath), "%s/3.st", dirPath);

file = fopen(filePath, "w");
if (file == NULL)
{
    perror("fopen");
    return;
}
for (i = 0; i < mem_table_cnt; i++)
    fprintf(file, "%d %d %s\n", mem_table[i].order, mem_table[i].key, mem_table[i].val);
fclose(file);
```

main_flush는 앞서 과정을 간략하게 설명했으니 새로운 storage file을 생성하는 부분만 설명

하겠다. storage_files 디렉토리를 열어 1.st, 2.st, 3.st 중 생성되지 않은 가장 앞의 파일을 만들어준다. 해당 파일에 전달받은 memory table 값을 작성한다.

```c
void main_get(char *key_buf, char *val_buf, struct table_elem *mem_table)
{
    int key = 1000 * key_buf[0] + 100 * key_buf[1] + 10 * key_buf[2] + key_buf[3];
    int i;
    for (i = 2; i >= 0; i--)
    {
        if (mem_table[i].key == key)
        {
            memcpy(val_buf, mem_table[i].val, 16);
            return;
        }
    }

    if (findValueByKey("storage_files/2.st", key, val_buf))
        return;
    findValueByKey("storage_files/1.st", key, val_buf);
}
```

main_get에선 전달받은 key 값을 토대로 해당하는 value를 찾는다. 최신 데이터를 찾기 위해선 역순으로 탐색해야 하므로, (1)memory table을 역순으로 탐색하고, (2)값을 찾지 못한 경우 2.st와 1.st 순으로 storage file을 탐색한다.

[4] output process

```c
#include "device.h"

void fnd_dd(int fnd_data)
{
    int dev = open("/dev/fpga_fnd", O_RDWR);
    if (dev < 0)
    {
        perror("ERROR(device.c) : fnd open failed.\n");
        _exit(-1);
    }

    unsigned char data[4];
    int i;
    for (i = 3; i >= 0; i--)
    {
        data[i] = fnd_data % 10;
        fnd_data /= 10;
    }

    write(dev, &data, 4);

    close(dev);
}
```

output_process를 설명하기 전에, device control 루틴을 함수로 작성한 device.c를 먼저 알

아야 한다. fnd_dd는 device driver를 통해 FND를 제어하는데, 앞서 설명한 것처럼 device file을 열어 알맞은 값을 작성하는 방식으로 쉽게 제어할 수 있다.

```
void led_mm(unsigned char led_data)
{
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0)
    {
        perror("ERROR(device.c) : led mem open failed.\n");
        _exit(-1);
    }

    unsigned long *fpga_addr = (unsigned long *)mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                                                     MAP_SHARED, fd, FPGA_BASE_ADDRESS);
    if (fpga_addr == MAP_FAILED)
    {
        perror("ERROR(device.c) : led mmap error. fd closed.\n");
        close(fd);
    }
    unsigned char *led_addr = (unsigned char *)((void *)fpga_addr + LED_ADDR);
    *led_addr = led_data;

    munmap(led_addr, 4096);
    close(fd);

    usleep(1000);
    return;
}
```

유일하게 LED는 mmap을 통해 제어한다. FPGA 내 LED의 주소를 직접 연결해서, 전달받은 값을 할당하는 방식으로 작동한다.

```
int led_pid = fork();
if (led_pid == 0) // led process
{
    printf("BEGIN : led process\n");

    struct led_msg *led = (struct led_msg *)malloc(sizeof(struct led_msg));
    led->cur_mode = MERGE_INIT;
    while (1)
    {
        msgrcv(led_q, led, LED_MSG_SIZE, 1, IPC_NOWAIT);
        if (led->_BACK_)
            break;

        switch (led->cur_mode)
        {
        case PUT_INIT:
        {
            led_mm(0x80);
            break;
        }
        // ...
```

output process는 간단하게 작동한다. 먼저 fork 된 led process인데, 단순히 입력받은 값에

대한 동작을 수행하기만 한다. 이때 IPC_NOWAIT 플래그를 설정한 msgrcv를 통해, 메시지가 오지 않아도 block 되지 않도록 해야 LED의 동작이 멈추지 않고 계속된다.

```
enum mode prev_mode = MERGE_REQ;
while (1)
{
    if (msgrcv(output_q, output, OUTPUT_MSG_SIZE, 1, 0) == -1)
    {
        perror("ERROR(output_process.c) : msgrcv failed.\n");
        _exit(-1);
    }

    if (output->_BACK_)
    {
        usleep(5000000);
        break;
    }
    if (output->_RESET_)
    {
        output_reset();
        continue;
    }
    if (prev_mode != output->cur_mode)
    {
        led->cur_mode = output->cur_mode;
        if (msgsnd(led_q, led, LED_MSG_SIZE, 0) == -1)
        {
            perror("ERROR(output_process.c) : msgsnd failed\n");
            _exit(-1);
        }
    }
    fnd_dd(output->fnd);
    switch (output->cur_mode)
    {
    case PUT_INIT:
    {
        lcd_dd("PUT MODE        ", output->lcd2);
        break;
    }
```

기존 output process의 루틴 역시 led process와 다를 것 없이, output 메시지에 해당하는 출력을 device.h에 정의된 함수를 통해 띄워주는 동작 뿐이다.

[5] merge process

```
void merge_process()
{
    printf("BEGIN : merge process\n");

    int merge_q = shmget(MERGE_KEY, MERGE_MSG_SIZE, 0666 | IPC_CREAT);
    struct merge_msg *merge = (struct merge_msg *)shmat(merge_q, NULL, 0);
    merge->_BACK_ = false;
    merge->_CALL_ = false;
    merge->merge_end = false;
```

merge_process가 시작되면 우선 shared memory에 연결하고 초기화한다.

```
while (1)
{
    int file_cnt = file_count("storage_files");
    if (merge->_BACK_)
    {
        usleep(4000000);
        if (file_cnt == 3)
            merge_files("storage_files");
        break;
    }
    else if (merge->_CALL_)
    {
        merge_files("storage_files");
        merge->merge_end = true;
        merge->_CALL_ = false;
    }
    else if (file_cnt == 3)
    {
        merge_files("storage_files");
    }
    // usleep(100000);
}
```

이후 루프 내에서 merge가 일어나야 하는 상황에 따라 적절한 행동을 취한다. 위에서부터 (1)_BACK_ 요청이 들어온 경우, (2)MERGE 모드에서 유저가 요청한 경우, (3)storage file이 세 개가 된 경우이다.

```
void merge_files(const char *dirPath)
{
    motor_dd(MOTOR_ON);

    char file1Path[256], file2Path[256], file3Path[256], tmpFilePath[256];
    struct stat st;
    int file1Exists = 0, file2Exists = 0, file3Exists = 0;
    int fileCount = 0;

    snprintf(file1Path, sizeof(file1Path), "%s/1.st", dirPath);
    snprintf(file2Path, sizeof(file2Path), "%s/2.st", dirPath);
    snprintf(file3Path, sizeof(file3Path), "%s/3.st", dirPath);
    snprintf(tmpFilePath, sizeof(tmpFilePath), "%s/tmp.st", dirPath);

    file1Exists = (stat(file1Path, &st) == 0);
    file2Exists = (stat(file2Path, &st) == 0);
    file3Exists = (stat(file3Path, &st) == 0);

    fileCount += file1Exists ? 1 : 0;
    fileCount += file2Exists ? 1 : 0;
    fileCount += file3Exists ? 1 : 0;

    if (fileCount == 1)
        return;
```

merge_files에선 앞서 설명한 파일 구조를 만족하기 위해, 디렉토리 내에 어떤 1.st, 2.st, 3.st 중 어떤 파일이 존재하는지 확인한다. 만약 파일이 하나 뿐이라면 merge를 할 수 없으므로 종료한다.

```
    FILE *tmpFile = fopen(tmpFilePath, "w");
    if (!tmpFile)
    {
        perror("fopen tmpFile");
        exit(EXIT_FAILURE);
    }

    mergeAndSaveRecords("storage_files/1.st",
                        "storage_files/2.st",
                        "storage_files/tmp.st");
    fclose(tmpFile);

    if (file1Exists && file2Exists && !file3Exists)
    {
        remove(file1Path);
        remove(file2Path);
        rename(tmpFilePath, file1Path);
    }
    else if (file1Exists && file2Exists && file3Exists)
    {
        remove(file1Path);
        remove(file2Path);
        rename(file3Path, file2Path);
        rename(tmpFilePath, file1Path);
    }

    usleep(2000000);
    motor_dd(MOTOR_OFF);
}
```

이후 tmp.st를 생성하여 1.st와 2.st를 merge한 결과를 저장한다. 파일이 어떻게 존재하냐에 따라, remove & rename 과정을 다르게 하여 일관성을 유지한다.

```
void mergeAndSaveRecords(const char *inputFile1, const char *inputFile2, const char *outputFile)
{
    Record *records = malloc(sizeof(Record) * 2000);
    int count = 0;

    readRecordsFromFile(inputFile1, &records, &count);
    readRecordsFromFile(inputFile2, &records, &count);

    qsort(records, count, sizeof(Record), compareRecords);

    FILE *outFile = fopen(outputFile, "w");
    if (!outFile)
    {
        free(records);
        return;
    }
    int i;
    int new_order = 1;
    for (i = 0; i < count; i++)
    {
        if (i < count - 1 && records[i].key == records[i + 1].key)
            continue;
        fprintf(outFile, "%d %d %s\n", new_order++, records[i].key, records[i].value);
    }

    fclose(outFile);
    free(records);
}
```

실질적인 merge 루틴에선, 파일에서 record를 가져와 정렬한 후 중복된 값을 제거한다. 이

결과는 1부터 시작하는 새로운 order와 함께 outFile에 저장된다.

V. 기타

프로젝트를 조금 늦게 시작해서 초기에 설계한 대로 끝까지 진행할 수밖에 없었는데, 그것 때문에 코드에 아쉬운 부분이 많다. 다시 시간이 주어진다면 아래와 같은 내용을 개선할 것이다.

[1] IPC는 message queue와 shared memory만 사용하고 semaphore는 사용하지 않았는데, 다시 작성한다면 무조건 사용할 것 같다. 여러 개의 프로세스가 IPC 내용을 기다리면서 spin하는데, 이 과정에서 부하가 생기고 프린트 디버깅도 어려워졌다(msgrcv는 IPC_NOWAIT 플래그 없이 사용하면 block이 되지만, shared memory를 사용하는 프로세스나 message를 보내기만 하는 프로세스는 계속 spin했다). 또, 특정 프로세스가 작업을 하면서 잠시 공백이 생기면 함께 협업하는 다른 프로세스도 잠시 멈춰주어야 시스템이 안정적이었다(BACK 키 신호를 받고 main_process가 정지하는데 merge_process는 merge를 진행하느라 mem_table의 값이 유실되는 등). semaphore는 요구 사항에 있지 않았고 사용하면 구현이 어려워질 것이라고 생각해 이렇게 구현하였는데, 오히려 코드 크기가 커지면서 semaphore를 사용하지 않으니 안정성이 매우 떨어진다고 느꼈다.

[2] LED는 다른 output device와 다르게 현재 상태에 대한 긴 텀의 동작이 있기 때문에, LED 프로세스의 호출 방법이 달라야 했다. 나는 output_process 내에 led_process를 생성하고 output_process의 while loop에서 수신하는 output_msg를 조금 변형하여 led_process로 보내주었는데, 이 방법으로 하니 LED의 동작이 제대로 구현되지 않은 것 같다. 모드가 갱신되면 즉각적으로 LED가 바뀌는 것이 아니라, 그 전 상태의 usleep이 포함된 동작을 마친 이후에 변경된다. output_process의 시작 부분에 fork를 하는 것이 아니라, output_process가 메시지를 받고 모드의 변화를 인식했으면, 이후 led 동작 자체만을 지정해 루프 내에서 fork 했어야 한다.

[3] 급하게 쓰다 보니 LCD에 띄울 문자열을 준비하는 과정이 상당히 가독성이 떨어지고 비일관적이다. LCD에 가비지 없이 원하는 내용을 띄우려면 문자열을 입력하고 그 뒤를 모두 공백으로 채워주어야 한다. memset, memcpy, strncpy, snprintf 등 너무 다양한 함수를 긴 줄에 걸쳐 사용했더니 가독성이 매우 떨어지는 코드가 되었다. 16바이트 이하의 문자열을 공백으로 마무리한 문자열로 바꿔주는 함수를 작성했으면 좋았을 것이다.

[3] merge 알고리즘을 최적화하고 싶다. 시간이 부족해 모든 내용을 우선 합친 이후에 정렬 및 중복 제거 과정을 거치는 알고리즘을 작성하였다($\Omega(2N \log 2N + 2N)$). 문제 크기를 작게 하기 위해 두 파일을 우선 정렬하고($\Omega(2N \log N)$) 합쳤다면($\Omega(N)$) 더 나은 알고리즘이 될 것 같다.

[4] BACK 요청으로 main이 종료될 때 merge process가 3초 sleep 하는 것이 아니라, main process flush를 마친 후 merge call을 보내면 동기화에 있어 더 안정적일 것 같다. 만에 하나 3초 동안에 flush가 마무리되지 않는다면 1.st와 3.st가 디렉토리에 남는 등 의도한 storage file의 일관성이 깨지고, flush가 너무 빨리 끝난다면 기다리는 시간이 낭비된다.

[5] main_get에서 memory table에 값이 존재하지 않고, 1.st와 2.st 파일이 모두 존재하지 않는 경우를 고려하지 않았다. 맨 마지막까지 찾지 못한 경우 ERROR를 반환하는 루틴을 추가해야 했다.

```
void main_get(char *key_buf, char *val_buf, struct table_elem *mem_table)
{
    int key = 1000 * key_buf[0] + 100 * key_buf[1] + 10 * key_buf[2] + key_buf[3];
    int i;
    for (i = 2; i >= 0; i--)
    {
        if (mem_table[i].key == key)
        {
            memcpy(val_buf, mem_table[i].val, 16);
            return;
        }
    }

    if (findValueByKey("storage_files/2.st", key, val_buf))
        return;
    if(findValueByKey("storage_files/1.st", key, val_buf))
        memcpy(val_buf, "ERROR            ", 16);
}
```

전체적으로 코드의 완성도가 높지 않기 때문에 시간이 된다면 꼭 개선해보고 싶다.