Embedded System Software 과제 1

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20181625 남현준

개발기간: 2024. 03. 28. -2024. 04. 16.

최 종 보 고 서

I. 개발 목표

- 각 과제마다 주어지는 주제를 바탕으로 본 과제에서 추구하는 개발 목표를 설정하고 그 내용을 기술할 것.

Cross-compile이 가능한 환경에서 프로그램을 작성하여 임베디드 리눅스에서 실행할 수 있는 바이너리 파일로 변환하여 프로그램의 실행 및 디버깅을 진행한다.

수업에서 배운 지식들을 활용하여 FPGA 보드에서 특정 입력이 switch input을 통해 주어졌을 때, 명세서에 따른 작동이 진행될 수 있는 device control을 구현한다. 입력한 정보를 simple key-value store을 활용해서 관리하며 해당 store의 merge, create가 필요할 시, process간 IPC 통신 기법을 활용하여 한 process에서 다른 process에 message를 보내어 key-value와 관련된 작동이 실행될 수 있는 로직을 구현한다.

해당 과제에서는 simple Key-Value store가 가능한 프로그램의 작성을 진행하게 되며 fork를 통해 생성되는 3개의 프로세스에서 명세서에 맞게 Put, Get, Merge 모드의 구현 및 기능의 작동을 FPGA의 LED, FND, LCD, Motor, Switch 등을 통해 조작하게 될 것이다. 이 때, LED의 경우는 device driver를 사용하는 것이 아닌 mmap() 함수를 이용한 memory mapped IO 방식을 채용할 것이다.

II. 개발 범위 및 내용

- 자신들이 설계한 개발 목표를 달성하기 위하여 어떠한 내용의 개발을 수행할 지 그 범위와 개발 내용을 기술할 것.

가. 개발 범위

I. I/O process

A. Fork를 통해 생성된 3개의 process 중 사용자가 switch의 버튼이나 다른 버튼을 선택하여 input을 주었을 때, 해당 input에 맞는 처리를 진행한다.

II. Main process

A. Fork를 통해 생성된 3개의 process 중, I/O process에서 get, put request가 주어지거나, merge request가 전달되어 merge process에 merge를 수행하라고

메시지를 보내는 등의 처리를 진행한다.

III. Merge process

A. Fork를 통해 생성된 3개의 process 중, background 상태로 실행되어 merge request가 수동으로 들어와 merging을 진행하거나, 생성된 .st 파일의 수가 3개가 넘어가는 경우, merging을 진행할 수 있도록 .st 파일의 상황을 지속적으로 체크한다.

IV. Device control

A. Achroimx 장비에서 명세서에서 요구한 모듈의 input 감지 및 input이 주어졌을 때의 결과를 처리하여 요구에 맞는 결과가 FPGA 보드 상에서 적절하게 보여질 수 있도록 한다.

V. Program operation

A. 작성한 프로그램에서 Put, Get, Merge 모드를 구현하고, 각 모드에서 명세서에서 요구한 기능들이 작동하고 back button이 input으로 주어질 때 정상적으로 종료될 수 있도록 구현을 진행한다. Put과 Get의 경우, memory table에 일정 수 이상의 데이터가 있을 때 flush되어 .st파일의 생성 및 데이터의 저장이 될 수 있도록 하고, 입력이 가장 최근에 일어났던 데이터의 search를 적절히 진행할 수 있도록 구현한다. Merge의 경우, manual하게 수행될 시 .st파일의 수가 2개 이상일 때, background에서 실행되는 merge process에 의해 자동적으로 실행되는 경우 .st파일의 수가 3개 이상일 때 실행될 수 있도록 한다. 종료 시에도 명세서에 맞게 .st파일의 수가 3개 이상일 경우 자동적으로 merge를 수행한 후 종료될 수 있도록 구현을 진행한다.

B. 각 process에서 특정 기능이 수행되는 경우, IPC를 통해 process간 통신을 진행한다. 이 때, Main-IO process 사이에서는 message queue 방식을 사용하고, Main-Merge의 경우 shared memory를 활용해서 process간 통신이 진행될 수 있도록 구현을 진행한다.

나. 개발 내용

I. IPC

A. Fork를 통해 프로그램에서 필요한 3개의 process를 생성하기에 앞서 IPC(Inter-Process Communication)를 사용하기 위해 shared memory와 message queue에서 사용할 key를 정의한다. 이는, fork를 통해 생성되는

process들이 같은 key값을 공유하여 message queue나 shared memory를 참조해 process간 전달되는 message를 확인할 수 있게 하기 위함이다. 이 때, message는 특정 시기에 write되는 것이고 fork된 process의 특성 상 프로그래머는 fork된 process가 어떤 순서로 실행될 지 예측할 수 없다. 이 경우, synchronization이 동반되지 않는 경우, 원하지 않는 결과의 return으로 이어질 수 있기에 semaphore를 선언하여 한 번에 하나의 message 작성 및 전달이 이루어질 수 있도록 한다.

II. READKEY

A. VOL+

버튼 입력 시, program의 모드가 PUT -> GET -> MERGE -> PUT …의 형태로 전환되도록 하고, 해당 모드에 맞는 초기 상태 (LED, FND, LCD 등)로 세팅되도록 한다.

B. VOL-

버튼 입력 시, program의 모드가 PUT -> GET -> MERGE -> PUT …의 형태로 전환되도록 하고, 해당 모드에 맞는 초기 상태 (LED, FND, LCD 등)로 세팅되도록 한다.

C. BACK

프로그램 종료 및 종료에 따른 memory table에 남아 있던 data들의 flushing과 .st파일의 생성이 진행될 수 있도록 한다.

III. I/O process

A. PUT

i. Switch를 통해 Key-Value 값을 입력받고 해당 결과를 FND와 LCD panel 상의 결과로 출력하게 된다. Key-Value 입력의 경우 초기에는 key 입력이 진행되며, FPGA의 RESET 버튼을 누르는 것을 통해 key <-> value input으로의 전환이 가능하다.

ii. I/O process의 PUT mode에서는 다음과 같은 module들의 작동이 일어난다.

1. LED

LED는 PUT mode가 실행되고 있는 경우, 초기에 1번 LED만 점등되어 있는 상태이다. 이후, key나 value의 입력이 진행됨에 따라 다른 LED가 점등하게 된다. Key가 입력되는 중인 경우, 3, 4번 LED가 번갈

아가며 점등하고 Value가 입력되는 중인 경우 7, 8번 LED가 번갈아가며 점등된다. 이후 (4) – (6) switch의 입력을 통해 PUT request가 주어진 경우, 1 ~ 8번 LED의 점등 이후 초기 상태인 1번 LED만 점등한 상태로 전환된다.

2. Switch

Switch의 입력은 Key인지 Value인지에 따라 각각 FND, LCD panel 상에 switch 입력에 따른 결과가 출력된다. Key 입력의 경우 (1) ~ (9) switch 입력에 따라 각 switch의 번호에 맞게 1 ~ 9 까지의 숫자가 FND 상에 출력된다. (1) switch를 1초 이상 누르고 있는 경우, FND에 입력되어 있던 값들이 삭제되고 초기 FND panel의 상태인 0000 상태로 되돌아간다.

Value 입력의 경우, (1)번 switch가 value의 숫자 <-> 알파벳 입력의 전환에 해당되는 버튼이기에 LCD에 출력될 수 있는 숫자는 2 ~ 9 까지이다.

알파벳 입력의 경우 다음과 같이 매핑되어 있다.

그림4. Switch key mapping

같은 switch를 연속적으로 누르는 경우 해당 switch에 대응되는 알파벳이 예전에 사용하던 피처폰의 자판처럼 전환된다. 가령 value에서 알파벳을 입력할 때 (2) – (2) – (2) – (5) – (5) – (4)가 연달아서 들어오게 된다면 LCD에는 CKG가 표시된다는 의미이다.

1번 switch를 1초 이상 누르고 있는 경우 현재 입력이 key인지 value인지에 따라 FND나 LCD panel이 초기 상태로 돌아가게 된다. 이 때, 초기 상태는 FND의 경우 0000, LCD의 경우 현재 모드가 PUT mode인지 GET mode인지에 따라 해당 모드의 이름이 LCD에 표현되는 32 string (각 첫 줄 16 characters, 두번째 줄 16 characters)의 첫

16 characters에 출력된다.

(4) – (6) switch의 입력을 통해 PUT request가 주어진 경우, FND와 LCD가 PUT mode의 초기화면으로 전환된다. 그리고 main process에 message queue를 통해 put을 하라는 명령이 전달될 수 있도록 한다.

B. GET

i. LED

LED는 GET mode가 실행되고 있는 경우, 초기에 5번 LED만 점등되어 있는 상태이다. 이후, search key의 입력이 진행됨에 따라 다른 LED가 점등하게 된다. Search key가 입력되는 중인 경우, 3, 4번 LED가 번갈아가며 점등하고 이후 RESET 버튼의 입력을 통해 GET request가 주어진 경우, 1 ~ 8번 LED의 점등 이후 초기 상태인 5번 LED만 점등한 상태로 전환된다.

ii. Switch

GET mode에서 입력되는 search key는 FND 상에 출력된다. PUT mode에서와 마찬가지로 switch 입력이 주어짐에 따라 해당 switch에 대응되는 번호가 FND에 출력된다. GET mode에서와 마찬가지로 1번 switch를 1초 이상 누르는 경우 FND의 화면이 초기 상태인 0000으로 리셋된다.

IV. Main process

A. Message queue

I/O process에서 msgsnd() 함수를 통해 전달된 request를 msgrcv() 함수를 통해 가져오고, I/O process에서 요청한 request에 맞게 put, get 등의 처리를 통해 storage table의 생성, memory table의 갱신, search key에 대응하는 data의 search 결과를 반환할 수 있도록 한다.

B. Shared memory

Shared memory를 통해 main process에서 merge process에 merge 요청을 보내고 merge 기능이 수행될 수 있도록 한다.

V. Merge process

Fork를 통해 생성된 3개의 process 중, background 상태로 실행되어 merge request가 수동으로 들어와 merging을 진행하거나, 생성된 .st 파일의 수가 3개가 넘어가는 경우, merging을 진행할 수 있도록 .st 파일의 상황을 지속적으로 체크한다.

Merge가 수행될 수 있는 상황인 경우, 가장 오래 된 .st파일 2개를 선택해 중복 Key가 있는 경우, 가장 최근에 입력된 Key-value pair를 선택해서 살린다. 중복 데이터의 제거가 완료되었다면, 해당 데이터에 Key에 대해 오름차순 정렬을 진행해서 중복 pair들이 제거 및 정렬됨에 따라 입력 순서의 재조정을 진행한다. 이후에 저장되는 data들은 merge가 진행된 파일에서의 순서 + 1부터 저장되고 생성되는 .st파일은 merge를 통해 생성된 파일의 번호 + 1을 이름으로 가지게 된다.

.st파일들은 storage_list 파일을 생성하여 .st파일과 관련된 데이터를 관리하게 된다. Storage_list 파일에는 새롭게 생성될 .st파일의 이름이 될 max_num, 다음에 put되게 되는 key_value의 순서 번호가 될 key_value_idx, 이전에 생성된 .st파일을 BACK 버튼을 통한 종료 및 실행 파일의 재실행 시 생성 및 로드될 수 있도록 storage_file의 이름을 가지고 있게 된다. 해당 내용들은 프로그램이 실행됨에 따라 지속적으로 값을 갱신해나가게 된다.

Merge의 경우 I/O process에서 요청이 들어와 merge가 수행되거나, background 상태에서 storage_list의 갱신을 체크하며 현재 생성된 .st파일의 수가 3개 이상이 되었을 때 자동적으로 merge가 수행될 수 있도록 한다. 프로그램 종료 시(ctrl + C와 같은 강제 종료가 아닌 BACK key를 통한 종료) memory table에 남아있던 data를 우선 .st 형식으로 저장한 뒤, 그렇게 했을 때 .st 파일이 3개 이상이라면 merge를 수행한 후 해당 내용을 storage_list 파일에 반영한 후 프로그램이 최종적으로 종료되도록 한다.

III. 추진 일정 및 개발 방법

- 자신들이 설정한 개발 목표를 달성하기 위한 개발 일정을 설정하고, 각 요소 문제를 해결하기 위해서 어떤 방법을 사용할 지 기술할 것.

가. 추진 일정

03.28 ~ 03.31 : 명세서 정독 및 프로그램 구현 방안 생각

04.01 ~ 04.15 : I/O, Main, Merge process의 구현, IPC (message queue, shared memory) 적용

04.15 ~ 04.16 : 최종 테스트

04.16 ~ 04.18 : 보고서 작성

나. 개발 방법

프로그램을 작성하기 앞서, 해당 프로그램에서 사용할 모듈들이 적절하게 open되어 사용될 수 있어야 한다. 해당 프로그램의 제약조건에서 LED는 memory addressed mapping을 통해 사용하고, 다른 모듈들은 device driver를 통해 접근 및 사용되어야 한다고 명시되어 있다.

이를 위해 다음과 같이 macro를 선언하였다.

```
#define FPGA_BASE_ADDRESS 0x08000000 // FPGA Base Address
#define LED_ADDR 0x16


#define FPGA_BACK_KEY 158
#define FPGA_VOLUP_KEY 115
#define FPGA_VOLDOWN_KEY 114

#define BUFF_SIZE 64

//for LCD
#define MAX_BUFF 32
#define LINE_BUFF 16

#define KEY_RELEASE 0
#define KEY_PRESS 1

#define MESSAGE_QUEUE_KEY 53201657
#define SHARED_MEMORY_KEY 53201658

#define READ_KEY "/dev/input/event0"
#define PUSH_SWITCH "/dev/fpga_push_switch"
#define FPGA_FND "/dev/fpga_fnd"
#define FPGA_LED "/dev/mem"
#define FPGA_LCD "/dev/fpga_text_lcd"
#define FPGA_MOTOR "/dev/fpga_step_motor"
```

LED의 경우, mmap을 통해 사용할 수 있도록 다음과 같이 정의했다.

```
//SetLED : set LED with given number to boolean turned status. if turned is 1 LED is ON, if 0 LED is OFF.
void SetLED(int led_num, bool turned){
    int dev_LED = open(FPGA_LED, O_RDWR | O_SYNC);

    if(dev_LED < 0){
        printf("FPGA_LED open error₩n");
        close(dev_LED);
        exit(-1);
    }

    unsigned long *fpga_addr = 0;
    fpga_addr = (unsigned long *)mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, dev_LED, FPGA_BASE_ADDRESS);

    if(fpga_addr == MAP_FAILED){
        printf("mmap error!₩n");
        close(dev_LED);
        exit(-1);
    }

    //calculate physical address of LED
    unsigned char *led_addr = 0;
    led_addr = (unsigned char*)((void*)fpga_addr+LED_ADDR);

    //00000000 <- LED is represented in binary. Each bit represents for whether LED is on or not. LSB for D8 and MSB for D1
    int led_value = 1;
    int i = 0;
```

```c
//data structure for ipc which will be used by message queue and shared memory
typedef struct _ipc_message{
    int process_type;
    int message_type;
    char data[50];
} IPC_MESSAGE;
```

IPC에 사용될 메시지의 정보를 저장하는 구조체를 선언하여 관리할 수 있도록 했다. Process_type의 경우 어떤 process에서 message를 보냈는지를 저장하고, message_type은 해당 request가 PUT, GET, MERGE request 중 어떤 것인지를 저장하는 변수이다. Data 변수는 PUT과 GET request 였을 경우, Key-value pair나 key의 값을 저장하기 위해 선언되었다.

```c
sem_t message_q_sema, merge_sema, shared_memory_sema, exit_sema;
```

또한, message의 경우 한 process로부터 다른 process에 message를 전달하게 되는 시점이 다르고, 특정 정보를 write하는 것이기 때문에 race condition이 발생하여 정보의 전달이 꼬이게 될 수 있다. 따라서, 이를 방지하기 위해 다음과 같은 semaphore들을 정의하여 initialize를 진행했다.

```
//For IPC, IO - Main uses message queue,  Main - Merge uses shared memory
key_t messageQId = ftok("keyfile", 1);
    messageQId = (key_t)MESSAGE_QUEUE_KEY;

if(messageQId < 0){
    printf("message Queue key creation failed\n");
    exit(-1);
}

//getting id of message queue #messageQId
int msqid = msgget(messageQId, IPC_CREAT|0644);

//initializing semaphores for synchronization
sem_init(&message_q_sema, 1, 1);
sem_init(&merge_sema, 1, 1);
sem_init(&shared_memory_sema, 1, 1);
sem_init(&exit_sema, 1, 1);

key_t sharedM = ftok("keyfile", 1);
sharedM = (key_t)SHARED_MEMORY_KEY;
if(sharedM < 0){
    printf("shared memory key creation failed\n");
    exit(-1);
}


int shid = shmget(sharedM, sizeof(IPC_MESSAGE), IPC_CREAT | 0644);
```

Message queue와 shared memory를 통한 IPC의 경우, fork를 통해 생성될 3개의 process에서 공유하여 사용할 수 있어야 하기에 fork하기에 앞서 정의를 진행하여 공통된 key를 가지는 message queue, shared memory에 message가 전달될 수 있도록 한다. 그리고 동기화를 위해 선언한 semaphore들의 초기화를 진행한다.

```
pid_t pid = fork();

int p_type;

if(pid < 0){
    printf("fork error₩n");
    exit(-1);
}

if(pid == 0){ //child process, I/O
    p_type = PROCESS_IO;
}
else{    //parent process

    pid = fork();

    if(pid < 0){
        printf("fork error₩n");
        exit(-1);
    }

    if(pid == 0){    //if it was child process after fork, name it as merge process
        p_type = PROCESS_MERGE;
    }
    else{        //if it was child process after fork, name it as main process
        p_type = PROCESS_MAIN;
    }
}
```

이후, fork를 진행하며 I/O, main, merge process가 생성되도록 한다.

```
if(p_type == PROCESS_IO){
    //SetMotor(0);
    int PROCESS_MODE_NAME = PROCESS_PUT;  //if first launched, mode is set to PUT
    SetLED(1, true);
    int input_event = open(READ_KEY, O_RDONLY | O_NONBLOCK);

    if(input_event < 0){
        printf("failed to open read key device₩n");
        close(input_event);
        exit(-1);
    }

    int push_switch = open(PUSH_SWITCH, O_RDONLY | O_NONBLOCK);

    //printf("works till here₩n₩n");

    if(push_switch < 0 ){
        printf("failed to open push switch device₩n");
        close(push_switch);
        printf("closing dev_push₩n");
        exit(-1);
    }
```

I/O process에서는 program에서 요구한 작동이 수행되기에 앞서 open()을 통해

READ_KEY와 PUSH_SWITCH가 ACHROIMX 기기에 설치되어 있는지 확인한다. 해당 모듈들이 설치되어 있지 않다면 사용자의 입력을 감지할 수 없기 때문에 프로그램을 종료한다.

사용자 입력의 경우, VOL+, VOL−, BACK key와 같은 키, switch input, reset이 입력될 때를 기점으로 크게 분류를 나누어 진행했다.

```
if(key_type == FPGA_BACK_KEY){
    printf("BACK KEY pressed\n");
    close(input_event);
    close(push_switch);
    //sending message to main process via message queue
    IPC_MESSAGE *ipcm = (IPC_MESSAGE *)malloc(sizeof(IPC_MESSAGE));
    ipcm->process_type = PROCESS_MAIN;
    ipcm->message_type = MESSAGE_IO_EXIT;
    sprintf(ipcm->data, "");

    sem_wait(&message_q_sema);
    msgsnd(msqid, (void *)&(*ipcm), sizeof(IPC_MESSAGE), 0);
    sem_post(&message_q_sema);

    free(ipcm);
    return 0;
}
```

```
else if(key_type == FPGA_VOLUP_KEY){
    //MODE changed in PUT -> GET -> MERGE -> PUT ... order
    mode_change_flag = true;
    printf("VOL UP pressed\n");
    PROCESS_MODE_NAME = PROCESS_MODE_NAME + 1 > 3 ? 1 : PROCESS_MODE_NAME + 1;
    printf("Mode changed to %s\n", PROCESS_MODE_NAME_STR[PROCESS_MODE_NAME]);

    process_cur = PROCESS_NONE;
    //Reset FPGA and other variables to initial values
    initDevice(PROCESS_MODE_NAME);
    SetLCDFirstLine(PROCESS_MODE_NAME);
    perform_action = false;

    if(PROCESS_MODE_NAME == PROCESS_PUT)
        SetLED(1, true);
    else if(PROCESS_MODE_NAME == PROCESS_GET){
        SetLED(5, true);
    }

    for (i = 0; i < 4; i++) {
        fnd_string[i] = '\0';
    }
    fnd_strlen = 0;

    for (i = 0; i < 6; i++) {
        lcd_string[i] = '\0';
    }
    lcd_strlen = 0;


}
```

```
else if(key_type == FPGA_VOLDOWN_KEY){
    //MODE changed in PUT -> MERGE -> GET -> PUT ... order
    mode_change_flag = true;
    printf("VOL DOWN pressed\n");
    PROCESS_MODE_NAME = PROCESS_MODE_NAME - 1 < 1 ? 3 : PROCESS_MODE_NAME - 1;
    printf("Mode changed to %s\n", PROCESS_MODE_NAME_STR[PROCESS_MODE_NAME]);
    process_cur = PROCESS_NONE;

    //Reset FPGA and other variables to initial values
    initDevice(PROCESS_MODE_NAME);
    SetLCDFirstLine(PROCESS_MODE_NAME);

    perform_action = false;

    for(i = 0; i < 8; i++){
        SetLED(i+1, false);
    }

    if(PROCESS_MODE_NAME == PROCESS_PUT)
        SetLED(1, true);
    else if(PROCESS_MODE_NAME == PROCESS_GET){
        SetLED(5, true);
    }

    for (i = 0; i < 4; i++) {
        fnd_string[i] = '\0';
    }
    fnd_strlen = 0;

    for (i = 0; i < 6; i++) {
        lcd_string[i] = '\0';
    }
    lcd_strlen = 0;

}
```

VOL+, VOL-, BACK key의 경우, 각 버튼의 작동에 맞게 LCD 상에 현재 모드의 이름을 출력하도록 했으며 back key가 눌러지는 경우 input 입력을 중단하고 main process에 종료 message를 보낸 후 I/O process가 종료되는 형식으로 진행되었다.

VOL+와 VOL-가 선택됨에 따라 다음과 같은 함수들이 호출되어 FPGA의 모듈이 초기 상태로 돌아갈 수 있도록 한다.

```c
//SetLCDFirstLine : Set first 16 characters of LCD panel to string of process_type.
void SetLCDFirstLine(int process_type){
    int dev = open(FPGA_LCD, O_WRONLY);

    if(dev < 0){
        printf("LCD open error\n");
        close(dev);
        exit(-1);
    }

    unsigned char lcd_str[32];
    int i = 0;
    for(i = 0; i < 32; i++){
        lcd_str[i] = ' ';
    }
    if(process_type == PROCESS_PUT){
        char first_string[] = "PUT MODE";
        int str_size = strlen(first_string);

        for(i = 0; i < str_size; i++){
            lcd_str[i] = first_string[i];
        }
    }
    else if(process_type == PROCESS_GET){
        char first_string[] = "GET MODE";
                int str_size = strlen(first_string);

        for(i = 0; i < str_size; i++){
                        lcd_str[i] = first_string[i];
                }

    }
    else if(process_type == PROCESS_MERGE){
        char first_string[] = "MERGE MODE";
                int str_size = strlen(first_string);

            for(i = 0; i < str_size; i++){
                        lcd_str[i] = first_string[i];
                }

    }

    //initialize second line of LCD to blank
    //memset(lcd_str+LINE_BUFF, ' ', LINE_BUFF);
    write(dev, lcd_str, MAX_BUFF);
    close(dev);
}
```

```c
// TextFnd : Function that sets shown text of FND in respect to given input
void TextFnd(unsigned char FndString[4]){
    int open_dev = open(FPGA_FND, O_RDWR | O_SYNC);

    if(open_dev < 0){
        printf("Failed to open FPGA_FND\n");
        close(open_dev);
        exit(-1);
    }

    unsigned char text[4] = {'0', };
    int i = 0;

    for(i = 0; i < 4; i++){
        if(FndString[i] < 0x30 || FndString[i] > 0x39){
            text[i] = FndString[i];
        }
        else
            text[i] = FndString[i] - 0x30;
    }

    unsigned char retval = write(open_dev, text, 4);

    if(retval < 0 ){
        printf("Failed to copy text to FND\n");
        close(open_dev);
        exit(-1);
    }

    close(open_dev);
}
```

Switch 입력의 경우 1번, 4번, 6번 switch가 선택되었을 때 일정 시간 입력이 멈춰질 수 있도록 했다.

```c
//if input given is (1), wait for 1s to check whether it was pressed for more than 1 second
if(buttons_pushed[0] == 1){
    usleep(1000000);
}
//if input given is (4) or (6), wait for 0.2s for another input. if (6) is given, put request should be initiated
else if(buttons_pushed[3] == 1 || buttons_pushed[5]){
    usleep(200000);
}
```

이렇게 진행한 이유는, 1번 스위치의 경우 1초 이상 누르고 있을 때, FND나 LCD의 초기화가 진행되어야 하고, 4번과 6번의 경우 동시 입력을 통해 특정 mode에서 request가 수행되어야 하기에 동시 입력이 명확하게 처리될 수 있기 위함이다.

4번, 6번 switch input을 통해 I/O process에서 put request가 main process에 전달된다면 다음과 같이 처리가 진행된다.

```
if(msg_received->message_type == MESSAGE_IO_PUT_REQ){

    //if there are 3 key-value pairs stored in memory table, save those pairs to storage table
    if(key_value_cnt >= 3){
        printf("Save data to storage table\n");
        SaveDataToStorageTable(storage_list, key_value_tmp, key_value_cnt);
        //after saving key-value pairs to newly created storage table, increase the count.
        storage_list->max_num++;
        //printf("cur num file after put %d\n", storage_list->cur_num_file);
        key_value_cnt = 0;
    }
```

현재 memory table에 저장된 key_value pair의 수가 3개 이상인 경우, 다음 put request가 진행되기 위해서는 우선 memory table에 있던 값들이 storage table로 이동되어 저장되어야 한다. 이를 위해 SaveDataToStorageTable() 함수를 호출하여 현재 memory table에 있는 값들을 저장한 후 다음과 같은 처리가 진행된다.

```
//if message was sent with keys with less than 1000 (or if key was 0000), invalid key
if(atoi(key_value_tmp[key_value_cnt].key) < 1000){
    printf("invalid key was put\n");
    TextFnd(fnd_reset);
    TextLCD(PROCESS_PUT, lcd_tmp);
    for(i = 1; i < 8; i++)
        SetLED(i+1, false);
    SetLED(1, true);
    usleep(300000);
    continue;
}
//if message was sent with values with empty string or more than 5 characters, invalid value
if(strlen(key_value_tmp[key_value_cnt].value) < 1 || strlen(key_value_tmp[key_value_cnt].value) > 5){
    printf("invalid value was put\n");
    TextFnd(fnd_reset);
    TextLCD(PROCESS_PUT, lcd_tmp);
    for(i = 1; i < 8; i++)
        SetLED(i+1, false);
    SetLED(1, true);
    usleep(300000);
    continue;
}
```

Message로 전달된 key-value pair에서 invalid한 값이 전달되는 경우, 해당 값들은 memory table에 저장될 수 없기에 예외 처리를 진행해주었다.

```
printf("Key-Value (%d, %s, %s) stored\n\n", key_value_tmp[key_value_cnt].order, key_value_tmp[key_value_cnt].key, key_value_tmp[key_value_cnt].value);

if(key_value_tmp[key_value_cnt].order < 10){
    sprintf(lcd_tmp, "(%d, %s,%s)", key_value_tmp[key_value_cnt].order, key_value_tmp[key_value_cnt].key, key_value_tmp[key_value_cnt].value);
}
else{
    sprintf(lcd_tmp, "(%d, %s,%s)", key_value_tmp[key_value_cnt].order, key_value_tmp[key_value_cnt].key, key_value_tmp[key_value_cnt].value);
}

storage_list->key_value_idx++;
key_value_cnt++;

//End of PUT request pushed from IO process

//Turn on LED (1) ~ (8) for 1s
for(i = 0; i < 8; i++)
    SetLED(i+1, true);
//Turn LCD to PUT Mode, GET Mode, MERGE Mode based on its current status
//SetLCDFirstLine(PROCESS_PUT);
//char fnd_reset[4] = {'0', };

usleep(300000);

TextFnd(fnd_reset);
TextLCD(PROCESS_PUT, lcd_tmp);

//Turn off LED except for (1)
for(i = 1; i < 8; i++)
    SetLED(i+1, false);
```

문제가 없는 값들이었다면 해당 pair를 memory table에 저장하고 그 결과를 LCD 상에 TextLCD() 함수를 통해 출력할 수 있도록 했다.

GET mode에서 search key에 대한 search request가 주어지는 경우, 가장 최근에 입력된 값이 우선순위가 높다. 이를 위해 우선 현재의 memory table에 저장된 값과 같은 key값을 가지는 pair가 있는지 확인한다.

```
//data which was given most recently should be target
for(i = key_value_cnt-1; i >= 0; i--){
    if(!strcmp(key_value_tmp[i].key, msg_received->data)){
        find_flag = true;
        if(key_value_tmp[i].order < 10)
            sprintf(lcd_tmp, "(%d, %s,%s)", key_value_tmp[i].order, key_value_tmp[i].key, key_value_tmp[i].value);
        else
            sprintf(lcd_tmp, "(%d, %s,%s)", key_value_tmp[i].order, key_value_tmp[i].key, key_value_tmp[i].value);
    }
}
```

있을 경우 find_flag를 true로 처리하고 없는 경우 다음 로직을 수행한다.

```c
if(!find_flag){
    int num;
    char key_tmp[5], value_tmp[33];
    printf("printing cur num file in find_flag if : %d\n", storage_list->cur_num_file);

    //there can be at least two files with same key value. To get most recent input, search order should be reversed
    //since merged result is saved after key-value pairs in memory (if there were any) are saved to storage table.
    for(i = storage_list->cur_num_file - 1; i >= 0; i--){
        printf("%s\n", storage_list->storage_file[i]);
        FILE *fp = fopen(storage_list->storage_file[i], "r");
        while(fscanf(fp, "%d %s %s\n", &num, key_tmp, value_tmp) != EOF){
            printf("%s\n", key_tmp);
            if(!strcmp(key_tmp, msg_received->data)){
                if(num < 10)
                    sprintf(lcd_tmp, "(%d, %s,%s)", num, key_tmp, value_tmp);
                else
                    sprintf(lcd_tmp, "(%d, %s,%s)", num, key_tmp, value_tmp);
                find_flag = true;
                break;
            }
        }
        fclose(fp);
    }
}
```

Storage table에서 일치하는 key값을 가지는 pair가 있는지 확인한다.

이후, 해당 결과가 있었다면 key-value pair를, 없다면 ERROR를 LCD 상에 출력하도록 한다.

```c
if(!find_flag){
    strcpy(lcd_tmp, "ERROR");
}
//printf("lcd_tmp : %s\n", lcd_tmp);
//Turn of LED except for (5)
//Turn on LED (1) ~ (8) for 1s
for(i = 0; i < 8; i++){
    //printf("%d ", i);
    SetLED(i+1, true);
    usleep(100000);
}
//printf("\n");
usleep(300000);

printf("hi after textlcd process get\n");
TextFnd(fnd_reset);
TextLCD(PROCESS_GET, lcd_tmp);
for(i = 0; i < 8; i++)
    SetLED(i+1, false);
SetLED(5, true);
```

Merge의 경우 background와 원할 시 수행될 수 있어야 한다. 이를 위해, Merge process에서 지속적으로 현재의 storage_list 파일의 상태를 점검하도록 한다.

```
while(1){
    initStorageList(tmp);
    //SetMotor(1);
    if(tmp->cur_num_file >= 3){
        sem_wait(&merge_sema);
        Merge(3);
        initStorageList(tmp);
        sem_post(&merge_sema);
        continue;
    }
```

이 때, merge의 경우에도 storage_list 파일에 갱신이 일어날 수 있기에 semaphore를 통해 한 번에 하나의 merge만 수행되도록 한다. Background에서 실행되는 merge의 경우 파일이 3개 이상 시에 수행되는 것이기에 initStorageList() 함수를 통해 관찰하여 현재 상태가 3개 이상의 .st 파일이 있을 경우 수행될 수 있도록 if() 문을 통해 관리한다.

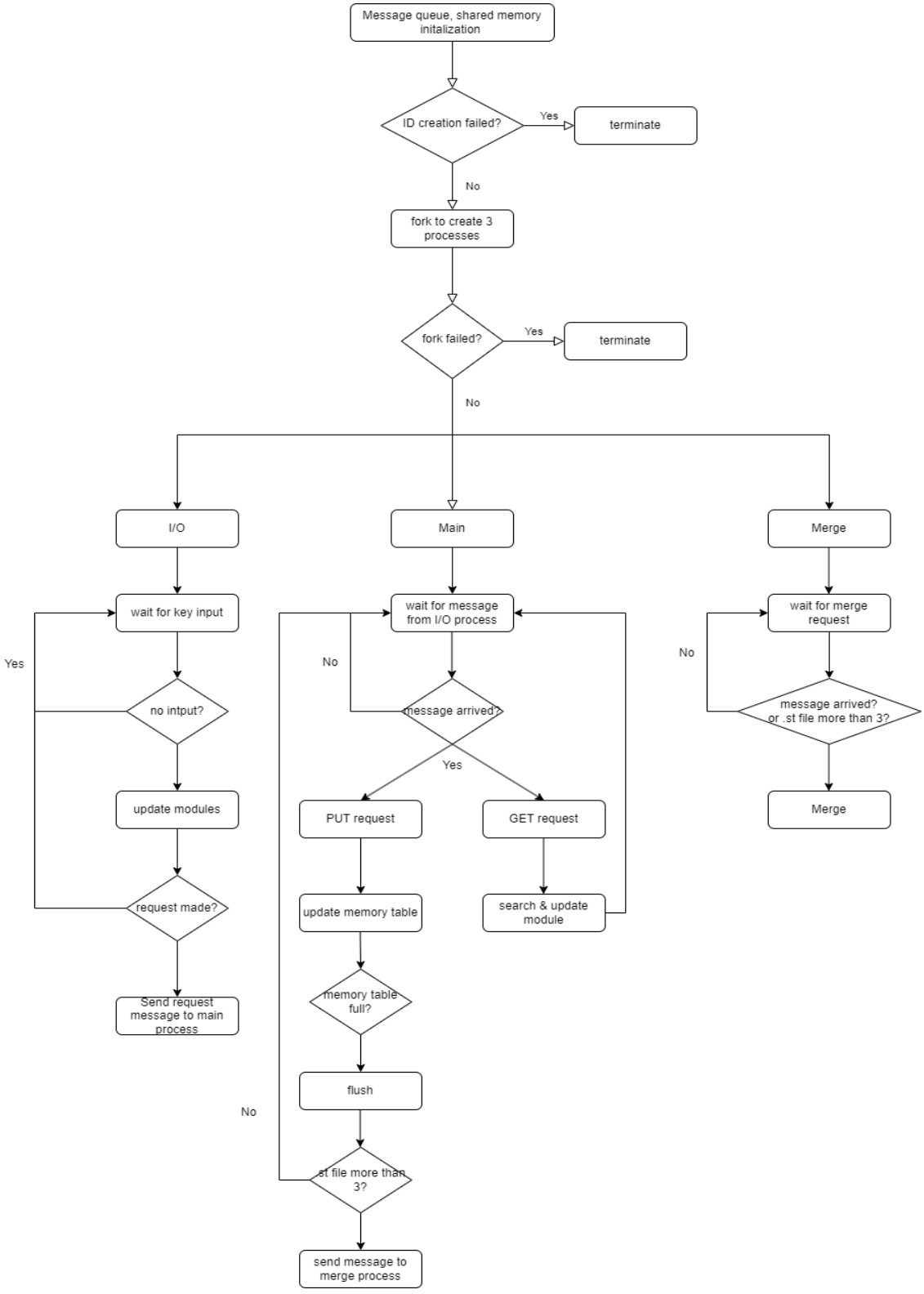
```
if(ipcm->message_type == MESSAGE_IO_MER_REQ){
    SetMotor(1);
    printf("motor set\n");

    clock_t start = clock(), time_spent;

    // while(1){
    //  time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    //  if(time_spent >= 3.0)
    //      break;
    // }

    //usleep(30000000);
    printf("HI MERGE 2\n");
    sem_wait(&merge_sema);
    Merge(2);
    initStorageList(tmp);
    sem_post(&merge_sema);
}
else if(ipcm->message_type == MESSAGE_IO_EXIT){
    sem_wait(&merge_sema);
    Merge(3);
    initStorageList(tmp);
    sem_post(&merge_sema);
    sem_wait(&shared_memory_sema);
    ipcm->process_type = PROCESS_NONE;
    ipcm->message_type = -1;
    strcpy(ipcm->data, "");
    sem_post(&shared_memory_sema);

    return 0;
}
```

직접 merge를 요청한 경우와 back key를 통해 program의 종료가 실행될 때 또한 merge가 수행될 수 있어야 한다. 이를 위해, shared memory에 merge request가 main process를 통해 전달된 경우, merge를 수행하도록 한다.

Back key의 경우, 프로그램이 종료되며 memory table에서 storage table로 이동된 data들을 저장한 후, .st 파일의 수가 3개 이상이 된 경우라면 merge가 실행되도록 하였다.

Ⅳ. 연구 결과

- 최종 연구 개발 결과를 자유롭게 기술할 것.



해당 프로젝트의 구현을 flow chart로 나타낼 시 다음과 같은 flow chart가 형성된다. 시스템프로그래밍, 운영체제와 같은 수업에서 습득한 내용들과 IPC(Inter-Process Communication)을 어떻게 구현하는지에 대한 보충자료의 내용으로 실제 process간에 일어날법한 communication 과정을 구현해보고 해당 결과를 확인해볼 수 있었다.

V. 기타

- 본 설계 프로젝트를 수행하면서 느낀 점을 요약하여 기술할 것. 내용은 어떤 것이든 상관이 없으며, 본 프로젝트에 대한 문제점 제시 및 제안을 포함하여 자유롭게 기술할 것.

해당 과제에서는 기존의 환경이 아닌, cross-compile을 통해 target machine에서 실행될 수 있는 프로그램을 작성하였고 이로 인해 생기는 제약 조건들이 몇 가지 있었다.

기존의 source machine에서의 코딩과 달리, embedded linux 상에서 cross-compile을 하게 될 때에는 기존의 문법과 약간의 차이가 존재했고 이러한 방식에 빠르게 적응할 수 있어야 했다. 가령 for() 반복문의 경우, for(int i = 0; i < n; i++)과 같은 방식으로 작성된 코드는 허용되지 않았기에 이렇게 작성된 부분들을 모두 고친 기억이 있다.

수업에서 언급된 내용과 같이 embedded Linux의 경우 source machine과는 달리 memory, 처리 속도, 용량 등에 limit가 존재하고 이에 맞는 프로그램의 작성이 동반되어야 최대 효율을 낼 수 있다. 하지만 과제 수행 간 프로그램의 속도에 직접적으로 영향을 주게 되는 branch나 loop를 무의식적으로 사용했고 이로 인해 실제 프로그램의 성능 지표가 그리 뛰어나지 않을 수도 있겠다는 생각이 들었다.