Embedded System Software 과제 1

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어
담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20181664 이동건
개발기간: 2023. 04. 11. -2023. 04. 19

최 종 보 고 서

I. 개발 목표

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

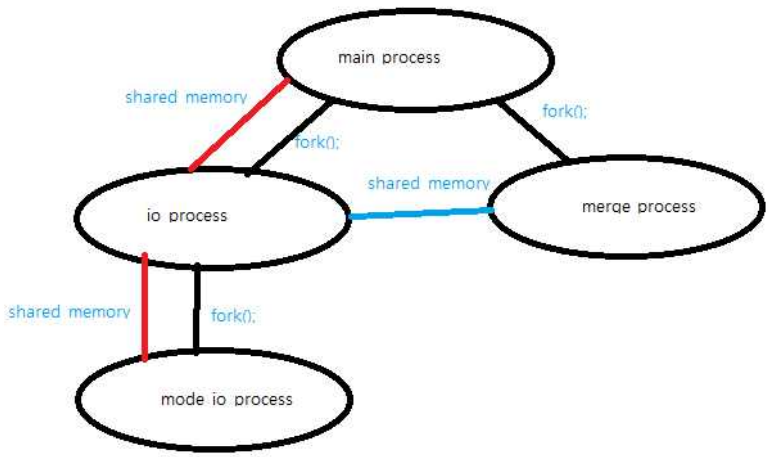
-Linux 환경에서 커널 환경에 대한 이해를 통한 개발, 또한 DeviceDriver, Memory Mapped I/O 등의 보드 디바이스 조작에 대한 내용을 이해를 통한 개발을 목표로 한다.
-디바이스에서 실행될 프로그램을 크로스 컴파일하는 개념을 이해하고 직접 수행해볼 수 있도록 한다.
-fork를 통해 여러 process를 다뤄보고, shared memory를 통한 IPC 방식과 semaphore를 이용하여 shared memory를 다루는 것에 대한 이해와 구현을 목표로 한다.
-위 개념의 이해를 바탕으로 명세서에서 요구하는 기능들인 PUT GET MERGE 기능 구현을 목표로 한다.

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

Process와 shared memory(IPC)



먼저 main process에서 프로세스를 두 개를 fork()를 통해 생성 하였다. 하나는 device의 input output을 담당하는 io process 이고 나머지 하나는 merge를 담당하는 merge process이다. 또한 추가로 io process에서 fork()를 통해 process를 하나 더 생성해 주었

다.
기존 과제 명세서에는 설명되어 있지 않았으나, mode 변경을 위한 key를 입력 받는 구현을 초기에 main process 혹은 io process에서 구현을 해보았는데 readkey 함수 특성상 키가 입력 받기 전까지 process 가 멈추는 것을 확인할 수 있었다. 따라서 main process 와 io process에서 mode 변경하는 readkey 기능을 구현하기 힘들다고 판단 하여, 따로 io process에서 fork를 통해 mode 변경 readkey만 입력 받는 process를 따로 생성하여 주었다.
또한 shared memory는 두 가지 생성하여 주었는데 하나는 main process, io process, mode io process 세 프로세스가 공유하는 것으로 input 받는 key, value 작동 모드 등의 정보를 공유하는 메모리이고 나머지 하나는 io process 와 merge process 두 프로세스가 공유하는 것으로 io process에서 저장된 key value 값들의 정보를 공유하는 메모리이다.

Device Control

디바이스를 제어하기 위해서는 device driver를 사용하거나 혹은 직접 mmap 함수를 이용하여 직접 접근하는 방법이 필요하다.
device driver를 사용하기 위해 device driver를 직접 보드에 insmod를 통해 삽입하였다. device driver 사용 방법에 관해서는 예시 코드를 참조하였다.

과제 기능 구현(PUT GET MERGE)

1. 모드변경 기능 : 프로그램 모드인 PUT GET MERGE 모드를 변경할 수 있는 기능을 구현한다.
2. put 기능 : key 와 value를 입력 받고, 이를 저장할 수 있는 기능을 구현한다.
3. get 기능 : key를 입력받고, 해당 key와 일치하는 key-value 값을 찾는 기능을 구현한다.
4. merge 기능 : 일정 개수 이상의 file 이나 data가 저장될 시 merge 하는 기능을 구현한다.

나. 개발 내용

Process와 shared memory(IPC)

위의 언급한 것과 같이 main process에서 먼저 io 처리를 위한 process 와 merge 처리를 위한 process 두 개를 fork를 이용하여 생성하여 준다. 또한 mode 변경 처리를 위한 process 도 io process에서 추가로 fork를 이용하여 생성하여 주었다. 이러한 process 별로 수행해야 되는 함수와 기능이 다르므로 각각을 pid를 지정하여 주어 pid 값을 구분하여 각 process 수행 기능을 구분하였다.
또한 각 process 는 서로 data를 공유하고 통신을 해야 하는데 이는 shared memory를 이용하여 구현하였다. 커널 영역에 메모리를 할당하고 각 process 마다 해당 메모리에 접근할 수 있도록 주소를 받아오는 형태로 진행하였다.
shared memory 는 main process, io process, mode io process 간 통신, io process, merge process 간 통신 이렇게 두가지의 shared memory로 구성하였다.

하지만 이렇게 메모리를 공유하게 되다 보면 메모리에 동시 접근하게 되는 문제가 생길 수

있다. 이러한 부분의 문제는 semaphore를 각 memory 마다 하나씩 사용하여 해결하였다.

Device Control

device control은 두가지 방법으로 수행할 수 있는데 바로 device driver 과 mmap 함수를 이용하는 것이다. 과제 명세서에 나와 있듯이, led control은 mmap함수를 이용하여 control 할 수 있도록 구현하였고, 이외의 control은 모두 device control를 이용하였다.

mmap함수를 사용하지 않고 device driver를 이용하려면 해당 모듈을 모두 설치해야하고 디바이스에 설치 후 아래에 있는 디바이스 드라이버 파일을 통해 접근, 이용하였다.

mmap 함수를 이용한 led device는 해당 주소에 직접 접근하여 해당 주소에 직접 data를 입력하는 식으로 구현하였다.

총 8개의 led가 있고 이는 이진수로 표현이 되므로 해당 led가 켜지게 하기 위해서는 1 값을 갖도록 하는 데이터를 입력하였다.

과제 기능 구현(PUT GET MERGE)

1. 모드변경 기능 : 프로그램 모드인 PUT GET MERGE 모드를 변경할 수 있는 기능을 구현한다.

모드 변경 기능은 readkey를 이용하여 작동하는데, readkey의 device driver 특성이 키를 입력 받기 전까지 process 가 멈추는 현상을 발견하였다. 따라서 해당 모드 변경만을 담당하는 process를 추가로 구현하였고, 공유 메모리에 모드를 저장하는 형태로 구현하였다.

2. put 기능 : key 와 value를 입력 받고, 이를 저장할 수 있는 기능을 구현한다.

해당 기능은 io process에서 진행하였고, 계속 키 입력을 읽으며 키 입력시 해당 키에 맞는 기능을 수행하도록 했다.

reset 버튼을 눌렀을 시에 key-value 모드가 변경되는데, 이를 공유메모리에 key 입력인지 value 입력인지 구분하는 값을 저장하여 주는 형태로 구현하였다.

이어 key-value 모두 switch라는 키로 값을 입력 받는데 같은 switch 기능을 이용하다 보니 key 입력 모드인지, value 입력 모드인지 구분하고, 해당 모드에 따라 값을 입력할 수 있도록 구현하였다.

value 모드의 경우 같은 키를 여러번 입력할 시 알파벳이 바뀌도록 해야 해서 2차원 배열로 알파벳을 저장해 놓고, 키 누른 횟수에 따라 알맞은 알파벳이 저장될 수 있도록 하였다. 또한 switch 키를 계속 입력 받으며 값 초기화, 알파벳-숫자 입력 변경, 값 저장 등을 처리하는 기능도 구현하였다.

key 와 value를 디바이스에 표현하는 것은 key가 입력될 때마다 디바이스에 표시될 수 있도록 하였다.

key- value 저장의 경우는 merge process 와 공유하고 있는 메모리에 값을 저장하도록 하였고, 3개가 입력될 시 file로 저장할 수 있도록 하는 신호를 공유메모리에 저장하도록 하였다.

3. get 기능 : key를 입력받고, 해당 key와 일치하는 key-value 값을 찾는 기능을 구현한다.

이 기능 또한 io process에서 진행될 수 있도록 하였고 put 에서와 마찬가지로 키를 입

력받고 이를 공유메모리에 저장한다. 또한 이 값을 지금 까지 저장된 메모리와 파일에 순차적으로 검색해보며 키 값이 일치하는 쌍을 찾는 기능을 구현하였다.

4. merge 기능 : 일정 개수 이상의 file 이나 data가 저장될 시 merge 하는 기능을 구현한다.
io process에서 key -value 값을 입력 받고 공유 메모리에 저장을 하도록 구현하였는데 이것이 일정 개수를 넘어가게 되면 file 로 저장할 수 있는 기능을 merge process에서 작동하도록 구현하였다.
또한 file 개수가 3개가 되면 merge 작업을 자동으로 수행하도록 구현하였으며, 3개가 아니어도 merge mode에서 reset 버튼을 누르면 merge 가 가능하도록 구현하였다.
하지만 명세서에 프로그램 실행 전에 기존에 파일이 있는 상태인지 아닌지 정확히 명세가 되어있지 않아 그 부분에 대한 예외 사항은 처리하지 못하였다. 오직 기존에 파일이 없는 상태만을 가정하고 구현하였다.

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

2023-04-11 : 과제 이해 및 fork와 readkey 기능 구현
2023-04-12 : IPC 기능 구현 및 PUT 기능 구현
2023-04-13~15 : PUT 기능 구현
2023-04-16~17 : MERGE 기능 구현
2023-04-18 : Merge 기능 구현 마무리 및 GET 기능 구현
2023-04-19 : test 및 보고서 작성

나. 개발 방법

-semaphore 사용

```c
int initsem(key_t semkey) {
    union semun semunarg;
    int semid;



    semid = semget(semkey, SEM_CNT, IPC_CREAT);
    if(semid == -1) {
            printf("sema error!\n");
    }
    else {
        semunarg.val = 1;

         int i;
        for (i = 0; i < SEM_CNT; i++) {
            if (semctl(semid, i, SETVAL, semunarg) == -1) {
                perror("initsem");
                return -1;
            }

            p[i].sem_num = v[i].sem_num = i;
            //semaphore index store
            p[i].sem_flg = v[i].sem_flg = SEM_UNDO;

            p[i].sem_op = -1;  // The semaphore p() operation
            v[i].sem_op = 1;   // The semaphore v() operation
        }
    }

    return semid;
}
void erase_sema(int semid)
{
    if (semctl(semid, 0, IPC_RMID, 0) == -1) {
        printf("fail to erase sema\n");
    }
    return;

}
```

해당 부분은 semaphore를 생성하는 것과 삭제하는 함수 이다. semaphore를 통해 process 간 동기화 문제를 해결하도록 하였다.

Fork() , shared memory 생성

```c
int make_fork(){
    int i,j;
    pid_t pid;
    io_shmid = shmget(SHM_KEY_1, sizeof(struct input_buff), IPC_CREAT|0644);
    merge_shmid = shmget(SHM_KEY_2, 3*sizeof(struct merge_buff), IPC_CREAT|0644);
    if(io_shmid == -1) {
        perror("io_shmget");
        printf("io error");
    }
    if(merge_shmid == -1) {
        perror("merge_shmget");
        printf("merge error");
    }


    switch(io_pid = fork()) {
        case -1:
            perror("fork");
            exit(1);
            break;
        case 0: //IO process
            io_shmaddr = (struct input_buff *)shmat(io_shmid, NULL, 0);
            merge_shmaddr= (struct merge_buff *)shmat(merge_shmid, NULL, 0);
            io_shmaddr->mode = PUT;
            io_shmaddr->buf_size = 0;
            io_shmaddr->is_changed = 1;
            io_shmaddr->key_idx = 0;

            merge_shmaddr->input_cnt = 0;
            merge_shmaddr->storage_cnt = 0;
            merge_shmaddr->need_merge = false;
            merge_shmaddr->need_store = false;

            io_shmaddr->get_request = false;
            //IO process shared memory attach

            return IO_PID;

        default: //Main Process
           // io_pid = pid;

            switch(merge_pid = fork()){
                case -1:
                            perror("fork");
                            exit(1);
                            break;
                case 0: // Merge Process
                    merge_shmaddr = (struct merge_buff *)shmat(merge_shmid, NULL, 0);
                    return MERGE_PID;
                default:
                    break;
            }
            io_shmaddr = (struct input_buff *)shmat(io_shmid, NULL, 0);
            io_shmaddr->mode = PUT;
```

해당 코드는 main process에서 fork 함수를 이용하여 io process와 merge process를 생성하는 모습이고, 또한 shared memory 사용을 위해 shmat 함수와 shmget 함수를 사용한 것을 확인할 수 있다.

또한 각 process 가 종료 될 때는 shared memory 와 semaphore를 삭제해 주어야 하는데 그렇지 않으면 shared memory 와 semaphore 가 남아있게 되어 추후 프로그램 작동에 문제를 일으킬 수 있다.

```
struct input_buff{
    int mode;
    unsigned char key[MAX_DIGIT+1];
    unsigned char key_value_mode;
    unsigned char value[LCD_MAX_BUFF];
    int buf_size;
    int key_size;
    int key_idx;
    char is_changed;

    bool get_request;
};

struct merge_buff{
    struct input_buff io_merge_buf[3];
    int input_cnt;
    bool need_store;
    bool need_merge;
    int storage_cnt;
};

struct input_buff *io_shmaddr;
struct merge_buff *merge_shmaddr;
```

io.h에 저장되어 있는 구조체인데 해당 구조체로 shared memory를 생성하였다.
input_buff에 해당하는 구조체에는 현재 작동 모드를 저장하는 변수, 키 값 저장 변수, value 값 저장 변수, put 모드에서 key-value 모드를 인식하는 모드, value size, key size, 모드 변경이 있었는지 여부 에대한 정보를 담고 있다.
merge_buff에 해당하는 구조체에는 input_buff를 멤버로 가지고 있고 3개를 배열로 저장할 수 있도록 구현하였고, 메모리 테이블에 저장된 값이 몇 개인지 저장하는 input_cnt, 또한 파일로 저장된 storage table이 몇 개인지 나타내는 storage_cnt 등의 정보를 담고 있다. need_store, need_merge flag로 store 할지 merge 할지 나타내어 준다.

```
#define DOT_DEVICE "/dev/fpga_dot"
#define FND_DEVICE "/dev/fpga_fnd"
#define LED_DEVICE "/dev/mem"
#define TEXT_LCD_DEVICE "/dev/fpga_text_lcd"
#define BUZZER_DEVICE "/dev/fpga_buzzer"
#define PUSH_SWITCH_DEVICE "/dev/fpga_push_switch"
#define DIP_SWITCH_DEVICE "/dev/fpga_dip_switch"
#define STEP_MOTOR_DEVICE "/dev/fpga_step_motor"
#define KEY_DEVICE "/dev/input/event0"

#define FPGA_BASE_ADDRESS 0x08000000 //fpga_base address
#define LED_ADDR 0x16

#define DEVICES_CNT 9
#define BUFF_SIZE 64
```

디바이스 파일이 저장되어 있는 경로를 매크로로 저장하여 디바이스 control 할 때 보다 수월하게 작성할 수 있도록 하였다. 또한 FPGA_BASE_ADDRESS와 LED_ADDR 매크로를 통해 led 디바이스의 mmap 함수에 사용할 주소를 쉽게 구할 수 있도록 하였다.

IV. 연구 결과

- 최종 연구 개발 결과를 자유롭게 기술할 것.
이번 과제를 수행하며 보드 디바이스에서 실행될 프로그램을 cross compile을 진행해보고 생성된 실행 파일을 usb port로 adb push 명령어를 사용하여 보드에 전송 후 실행하여 보았다.
- 디바이스 드라이버와 mmap 함수를 이용하여 디바이스의 기능들을 직접 수행해볼 수 있었다.

-Shared memory, device driver, semaphore, mmap() 등 수업시간에 배운 내용을 직접 활용하여 과제 명세에 맞는 프로그램을 최대한 구현하도록 노력했다.

-해당 과제 수행을 통해 process의 수행 과정, semaphore의 사용법, 디바이스 드라이버 사용법 등을 배울 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.
이번 과제를 통해 평소에 local pc에서 local compile 하는 프로그래밍과 달리 외부 디바이스를 연결하여 해당 디바이스에 돌아갈 프로그램을 cross compile 해보는 경험을 해보게 되어 새로우면서 흥미로웠다.
낯선 프로그래밍이었기에 처음에는 많이 헤멨던 것이 사실인데 적응하다 보니 나름의 결과를 낼 수 있었던 것 같다.
조금 더 깔끔한 코드를 짜고 싶었지만, 그렇지 못한 것이 조금 아쉽고 응답이 조금 느리다는 생각이 들었다. 다음 과제에는 더욱 응답이 빠르고 정확한 코드를 만들어 보고 싶다는 생각이 들었다.