

Embedded System Software 과제 1

(과제 수행 결과 보고서)

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

학번 및 이름: 20171643, 박준혁

개발기간: 2023. 04. 08. -2023. 04. 16.

최종 보고서

I. 개발 목표

본 과제는 ‘임베디드시스템소프트웨어’ 실습 보드 내부에 Key-Value Store System을 구축하여 Device Control과 Embedded System에서의 IPC를 연습해봄으로써 Embedded System 개발의 기초 과정을 익히는 것을 목표로 한다.

따라서 이를 위해 FPGA 보드 상에서 LED, FND, TEXT LCD, MOTOR 등의 Output Device와 SWITCH, KEY와 같은 Input Device들의 Driver 내부 작동 원리, 그리고 그 Driver에서 제공하는 기능을 Application Level에서 활용하는 방법 등을 각각으로 확인하고 연습한다. 이어 Synchronization과 Shared Memory를 위시한 Inter Process Control 기법을 Board Device 내부 Application에서 사용해봄으로써 복수의 Process가 유기적으로 작용하여 Service를 제공하는 Embedded System 개발을 경험한다.

한편, NoSQL Database의 Paradigm인 Key-Value Store를 배열 기반으로 굉장히 간단한 수준에서 구현해봄으로써 Key-Value Interface 개념과 Storage Device의 기본 속성인 Persistency를 이해하는 시간을 가진다.

본인은 약 일주일간의 집중적인 개발을 통해 상기 목표를 달성하는 Key-Value Store 구축을 완료하였으며 그 자세한 내용은 후술한다.

II. 개발 범위 및 내용

본 과제를 수행하기 위해 개발해야 하는 범위와 그 세부 내용에 대해 소개한다(기준 보고서 양식과 다르게 개발 범위와 내용을 각 범위에 대해 하나씩 묶어서 서술한다).

(1) Device Control

우선 본 과제에서 개발을 요구하는 Key-Value Store는 FPGA 보드에 내장된 각종 I/O Device와 유기적인 소통을 할 수 있어야 한다. 따라서 기본적으로 FPGA 내장 Device들의 Device Driver와 그러한 Driver에서 제공하는 I/O 기능의 사용 방법을 체득해야 한다. 이를 위해 본인은 과거 실습2에서 제공되었던 Device Driver들과 각각에 대응되는 Test Program들을 해킹하여 FPGA 보드 Device Control 기법을 이해하였다.

이어 그러한 이해를 토대로 Skeleton Key-Value Store를 설계하여 내부 Key-Value Interface 없이 Device I/O Control을 할 수 있는 System을 만들었다. 이때 Device에 대한 I/O Request 및 그 Result의 통신은 본 과제 명세서의 요구대로 2개의 Process를 띠워 하나는 SWITCH, KEY Device에 대한 Input 및 FND, LED, TEXT LCD, MOTOR Device에 대한 Output을 인식하고 요청하는 I/O(Input/Output) Process로, 다른 하나는 해당 I/O Process와 소통하며 추후 Key-Value Interface를 제공할 예정인 Main Process로 두어 각 Process가 유기적인 소통을 할 수 있도록 만들어주었다. 그 세부 내용은 바로 아래 항목에서 서술한다(Merge Process에 대한 설명은 본 항목에선 생략한다).

이렇게 만들어진 Skeleton 코드를 기반으로 아래 항목들의 개발을 수행하였다.

(2) IPC – Synchronization

상기 항목에서 서술한 ‘Multiple Process 간의 유기적 소통’을 위해선 반드시 동기화 작업이 필요하다. 만약 Input & Output Process와 Main Process가 각자 병렬적으로 아무런 동기화 작업 없이 돌아간다면 Key-Value Interface 제공 간에 Consistency가 깨지는 문제가 발생할 수 있기 때문이다(후술할 Merge Process와의 통신도 마찬가지임).

따라서 본인은 배부된 참고자료를 토대로 <sys/sem.h>의 Semaphore를 도입하여 각 Process 간의 통신을 통제하였다. I/O Flow 통제 간의 가장 핵심적인 동기화 작업은 바로 ‘Input → Main(Key-Value Service) → Output → Input → ...’의 순서를 강제시키는 것이다. Input이 들어오기 이전에 Main이 돌아가거나, 또는 Main이 출력물을 전달하기 이전에 Output을 처리해버리는 상황이 생기는 것을 방지하기 위함이다.

이를 위해 각 Process 통신 관계(‘I/O<->Main’, ‘I/O<->Merge’ 등)에 대해 각각 Binary Semaphore를 마련하여 해당 Semaphore들에 P Operation, V Operation을 적절히 수행해 통신 관계에 Mutual Exclusion을 제공하여 Synchronized I/O Flow를 구축하였다. 자세한 내용은 이하 ‘개발 방법’란에서 Code-Level로 소개한다.

(3) IPC – Shared Memory

한편 상기한 ‘통신 관계’에서 각종 Metadata, 이를테면 I/Oed Data 등의 전달은 어떻게 진행해야 할까? **복수의 Process**가 서로 다른 Memory Space를 점유한 채 돌아가는 상황에선 Process 간에 특정 Data Sharing 방법을 도입해 소통해야 한다. 본 과제에서는 이에 대해 Shared Memory 기법의 사용을 요구한다. 따라서 본인은 배부된 참고자료에 상세히 설명되어 있는 <sys/shm.h>를 이용하여 Shared Memory를 구축하였다.

이때 Shared Memory에는 앞서 언급한 것처럼 각 통신 관계에서 필요로 하는 Data들을 모두 포함시켜 원활한 Process Communication을 도모하였으며, 그러한 ‘Data’에는 Input Device에서 요구하는 Buffer, Output Device에서 요구하는 Buffer뿐만 아니라 Key-Value Interface 제공 간에 필요한 Memory Table(약칭 ‘Memtable’)과 각종 Persistence 관련 Metadata도 포함된다. 자세한 내용은 마찬가지로 이하 ‘개발 방법’란에서 Code-Level로 상세히 소개하겠다.

(4) Key-Value Interface

본 과제에서 최종적으로 구현해야 하는 Key-Value Store 기능인 PUT, GET, MERGE Operation을 구축하고, 이 기능을 제공하는 Interface를 마련하여 상기한 Main Process 또는 I/O Process가 이를 특정 Mode에 따라 호출하여 Service할 수 있도록 환경을 만든다.

일반적으로 Key-Value Store는 LSM(Log-Structured Merge)-Tree 혹은 Hash Data Structure를 기반으로 설계하는데 본 과제에선 그것이 목표가 아니기 때문에 본인은 간단한 Array로 Memtable 및 SSTable(Sorted-String Table)을 구축하였으며, PUT, GET은 명세서에 언급된 조건들을 모두 충족시키며 동작할 수 있도록 설계한다.

Memtable이 꽉 찼을 땐 On-Disk SSTable로의 Flush가 이뤄져야 하며, 이렇게 생성된 SSTable들이 System 내에 여럿 쌓였을 땐 MERGE(Compaction)도 이뤄져야 한다. 이때 명세서 조건에 따라 MERGE의 경우엔 따로 Merge Process를 두어 개별 처리하도록 한다 (이를 위한 Synchronization 및 Shared Memory 적용도 당연히 필요하다). 물론 명세서대로 Main Process 역시 마찬가지로 Mode 변경을 통해 MERGE를 Trigger할 수 있다.

III. 추진 일정 및 개발 방법

상기 항목에서 언급한 개발 범위 및 내용을 수행하기 위해 본인이 추진한 개발 일정 및 그 상세 과정을 소개한다.

가. 추진 일정

본인이 2주 전까지 연구실 논문 작업에 참여했던 관계로 충분한 시간 여유가 나지 않아 본격적인 개발은 과제가 출제되고 시간이 조금 지난 지난주 4월 8일부터 시작하였다. 그렇게 일주일 정도 개발에 몰두하였으며, 그 세부 과정은 다음과 같다.

- 04/08 - 04/09: 과제 명세서 및 참고자료 이해
- 04/09 - 04/10: Device Driver 및 Test Program 코드 이해, Skeleton Codes 작성
- 04/10 - 04/11: Skeleton 기반 IPC(Synchronization & Shared Memory) 구축
- 04/11 - 04/13: Key-Value Interface 구축 및 FPGA 보드 Test & Debugging
- 04/13 - 04/16: Program 안정화 및 보고서 작성

이러한 일정에 맞춰 과제를 수행하였으며 명세서에서 요구하는 조건을 모두 충족하였다. 본 과제의 Code-Level Implementation은 바로 다음 항목에서 소개한다.

나. 개발 방법

상기 개발 일정에 따라 과제 수행 초반에는 명세서 및 참고자료, Device Driver 이해의 시간을 가졌고, 그 이후부터 Code-Implementation에 돌입하였다. 지금부터는 상기 ‘II. 개발 범위 및 내용’ 항목의 세부 항목 구분을 따라서 개발 방법 및 내용을 소개한다.

(1) Device Control

미리 학습한 Device Control 체계를 그대로 모아 하나의 Source Code에 모아놓는다. 이름은 ‘device_control.h’라고 하자. device_control.h 내에는 먼저 본 Key-Value Store가 필요로 하는 I/O Device Driver들을 Open하고 Close하는 Routine을 마련한다. 아래와 같이 말이다.

```
/* Open all the devices we need here */
void devOpen(void) {
    DEVICES dv;
    for (dv = FND; dv <= SWITCH; dv++) {
        if ((devFD[dv] = open(DEVICE_ADDR[dv], OPEN_FLAGS[dv])) < 0)
            perror("Error occurs in device opens");
    }

    // Memory mapping for LED device
    mappedAddr = (unsigned long*)mmap(NULL, 4096, PROT_READ | PROT_WRITE,
        MAP_SHARED, devFD[LED], FPGA_BASE_ADDR);
    if (mappedAddr == MAP_FAILED)
        perror("Error occurs in memory mapping FPGA");
    ledAddr = (unsigned char*)((void*)mappedAddr + LED_ADDR);
}
```

```

/* Close all the devices we need here */
void devClose(void) {
    DEVICES dv;
    // Turn off all the output devices
    fndPrint(0); ledPrint(0);
    textlcdPrint(""); motorPrint(0);

    for (dv = FND; dv <= SWITCH; dv++) {
        if (close(devFD[dv]) < 0)
            perror("Error occurs in device closes");
    }

    // Unmap the memory mapped area (of LED)
    if (munmap((void*) mappedAddr, 4096) < 0)
        perror("Error occurs in memory unmapping FPGA");
}

```

FND, LED, TEXT_LCD, MOTOR와 같은 Output Device와, SWITCH, KEY와 같은 Input Device에 대해 각각 open System Call을 호출하여 Device Driver를 가져온다. 이 때, LED의 경우엔 I/Oed Data의 송/수신을 위해 Memory Mapping이 필요해 상기 코드와 같이 mmap을 호출하는 부분이 있음을 주목하자(마찬가지로 Close 시에 unmap을 해주어야 한다).

참고로, 이러한 Open, Close Routine 시에 Device Driver 주소를 가져오기 위해 아래와 같은 Data들이 필요하다. 상기 코드에서 ‘dv’라는 변수를 통해 ‘DEVICES’라는 Enum Type 을 순회하며 open, close가 연쇄적으로 일어날 수 있음을 주목하자. 이를 위해 각 Device Open 시 요구되는 Flag도 순회를 고려하여 배열 형태로 선언되어 있다.

```

/* Addresses of device drivers that we use here */
const char DEVICE_ADDR[DEVICE_CNT][25] = {
    "/dev/fpga_fnd",
    "/dev/mem",
    "/dev/fpga_text_lcd",
    "/dev/fpga_step_motor", // Output devices
    "/dev/input/event0",
    "/dev/fpga_push_switch" // Input devices
};

/* Flags for each open function call of devices */
const int OPEN_FLAGS[DEVICE_CNT] = {
    O_RDWR, // FND
    O_RDWR | O_SYNC, // LED
    O_WRONLY, // TEXT_LCD
    O_WRONLY, // MOTOR
    O_RDONLY | O_NONBLOCK, // KEY
    O_RDWR // SWITCH
};

```

이어 각 Device에 대한 Read/Write Routine을 마련한다. 앞선 devOpen 함수를 통해 추출한 Device Driver File에 대해 read/write System Call을 호출하면 대응되는 Device I/O Routine이 호출될 것이다. 아래는 FND, LED에 대한 Output(Write) 함수들을 보여준다.

다. LED의 경우엔 write System Call 대신 Memory Mapping Area에 대한 Assignment로 Write을 수행함을 주목하자.

```
/* Print data to the device (FND) */
void fndPrint(const int data) {
    unsigned char digits[MAX_DIGIT + 1] = { '\0' };
    int i, digit = 1;
    if (data < 0 || data > 9999)
        perror("Error occurs in FND printing (wrong data)");

    // Construct the string
    for (i = MAX_DIGIT - 1; i >= 0; i--) {
        digits[i] = (data / digit) % 10;
        digit = digit * 10;
    }

    // Write to device
    write(devFD[FND], digits, MAX_DIGIT);
}

/* Print data to the device (LED) */
void ledPrint(const int data) {
    int leds = data;
    if (leds < 0 || leds > 255)
        perror("Error occurs in LED printing (wrong data)");

    // Write to device
    *ledAddr = leds;
}
```

아래는 Input Devices인 KEY와 SWITCH에 대한 Read Routine들이다. SWITCH Read 과정에서 RESET Pressed 상황 Handling을 위해 (일반적인 SWITCH (1)~(9) 입력으로는 나올 수 없는) 특수 값을 사용하고 있음을 주목하자.

```
/* Read data from the device (KEY) */
int keyRead(void) {
    struct input_event keyBuf[MAX_KEY_CNT];

    // Read from the device
    if (read(devFD[KEY], keyBuf, sizeof(keyBuf)) >= sizeof(struct input_event)) {
        if (keyBuf[0].value == PRESSED) {
            if (keyBuf[0].code == KEY_VOLUMEDOWN ||
                keyBuf[0].code == KEY_VOLUMEUP ||
                keyBuf[0].code == KEY_BACK) {
                keyBuf[0].value = NOT_PRESSED;
                return keyBuf[0].code;
            }
        }
    }

    // No Key read
    return 0;
}

/* Read data from the device (SWITCH) */
int switchRead(void) {
```

```

unsigned char switchBuf[SWITCH_CNT];
int i, swch = 0;

// Read from the device
read(devFD[SWITCH], &switchBuf, sizeof(switchBuf));

// RESET Pressed
if (switchBuf[0] == 80 && switchBuf[SWITCH_CNT - 1] == 96)
    swch = SWITCH_RESET;
// Otherwise (normal switch)
else {
    for (i = 0; i < SWITCH_CNT; i++) {
        swch = swch * 10;
        if (switchBuf[i] == PRESSED)
            swch += 1;
    }
}

return swch;
}

```

이렇게 Device Controlling 환경이 구축되면 I/O Process에선 아래와 같이 각 Routine들을 호출하여 FPGA 상에서의 I/O를 통제한다. (공간 상의 이유로 Output 부분만 소개한다. Input 부분은 첨부 코드 io_process.c에서 확인할 수 있다.

```

/* 2. Output Routine */
for (dv = FND; dv <= MOTOR; dv++) {
    // print only if the device is in use
    if (shmOut->used[dv]) {
        switch (dv) {
            case FND: fndPrint(shmOut->fndBuf); break;
            case LED: ledPrint(shmOut->ledBuf); break;
            case TEXT_LCD: textlcdPrint(shmOut->textlcdBuf); break;
            case MOTOR: motorPrint(shmOut->motorOn); break;
            default: break;
        }
    }
    // if the device is used in the right before turn, then reset it
    else if (usedBefore[dv]) {
        switch (dv) {
            case FND: fndPrint(0); break;
            case LED: ledPrint(0); break;
            case TEXT_LCD: textlcdPrint(""); break;
            case MOTOR: motorPrint(0); break;
            default: break;
        }
    }
}

```

(2) IPC – Synchronization

본 Key-Value Store는 총 3개의 Process로 구성된다. 상술했던 것과 같이, Key-Value Interface를 제공하는 Main Process, FPGA Device에 대한 I/O를 수행하는 I/O Process, 그리고 SSTable들에 대한 Background MERGE 연산을 수행하는 Merge Process까지.

이들이 유기적으로 작동하면서 정상적인 System Flow를 유지하려면 Synchronization이 필요하다. Input이 먼저 들어오고 나서 Main이 그 Input에 대한 작업을 수행하고, Main의 작업이 끝나면 Output이 일어나야 한다. 이 I/O Flow 순서가 깨지면 Data Consistency 등에 문제가 생길 수 있기 때문이다.

이 순서를 지켜주기 위해 본인은 배부된 첨부자료를 토대로 <sys/sem.h>를 Program에 포함시켜 Semaphore를 도입한다. 우선, Semaphore를 사용하기 전에 아래와 같이 2개의 Process를 Fork한다. (2개가 Fork되었으니 총 3개의 Process가 존재하는 상황)

```
/* Fork two processes needed for our project */
PROCESS_TYPE forks(void) {
    pid_t pid;
    // First child -> I/O Process
    if ((pid = fork()) == 0)
        return _IO;
    else {
        ioPID = pid;
        // Second child -> Merge process
        if ((pid = fork()) == 0)
            return _MERGE;
        else mergePID = pid;
    }
    return _MAIN;
}
```

이어 Semaphore를 마련하고 아래와 같이 초기화한다. 좌측은 ipc.h에 Macro 형식으로 정의되어 있는 Semaphore들이고, 이를 토대로 우측과 같은 함수를 사용해 초기화한다.

```
/* Constants for semaphore interface */
#define SEM_CNT 4
#define SEM_KEY (key_t)0x20
#define SEM_MAIN_LOCK 0
#define SEM_IO_LOCK_1 1
#define SEM_IO_LOCK_2 2
#define SEM_MERGE_LOCK 3

/* Data structures for semaphore interface */
typedef union _semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *_buf;
} semun;

/* Semaphores */
struct sembuf p[SEM_CNT];      // For P operation
struct sembuf v[SEM_CNT];      // For V operation
/* Allocate the semaphore for each process */
int allocateSem(void) {
    int semID, i;
    semun op; op.val = 0;
    // Get semaphores
    if ((semID = semget(SEM_KEY, SEM_CNT, IPC_CREAT)) == -1)
        perror("Error occurs while allocating semaphore");
    // Initialize corresponding data structures
    for (i = 0; i < SEM_CNT; i++) {
        if (semctl(semID, i, SETVAL, op) == -1)
            perror("Error occurs while calling semctl system call");
        p[i].sem_num = i;
        p[i].sem_flg = SEM_UNDO;
        p[i].sem_op = -1;
        v[i].sem_num = i;
        v[i].sem_flg = SEM_UNDO;
        v[i].sem_op = 1;
    }
    return semID;
}
```

위 좌측 코드와 같이 Semaphore가 총 4개 존재한다. 먼저 SEM_MAIN_LOCK은 Main Process에 대한 Binary Semaphore이다. Main Process는 while Loop의 시작과 함께 SEM_MAIN_LOCK에 대해 P Operation을 호출하여 잠든다. I/O Process가 Device Input 을 받아들이길 기다리는 것이며, I/O Process는 Device Input 이후 Mutex Lock 역할의 SEM_MAIN_LOCK에 대해 V Operation을 수행하여 Main을 깨워줄 것이다. 한편, 두 번째 Semaphore인 SEM_IO_LOCK_1은 I/O Process가 Input을 받아들인 후 Main을 깨우고 나서 Main의 수행을 기다리기 위해 잠들 때 사용되는 Semaphore이다. SEM_IO_LOCK_2와 SEM_MERGE_LOCK은 Background Merge Process를 위한 Semaphores이다(잠시 뒤 설명). 아래는 ‘Main Process <-> I/O Process’ 간의 Semaphore 호출 관계이다.

```

/* Main Process handler */
const int mainProcess(const int semID) {
    static MODE_TYPE mode = PUT, prevMode = PUT;
    bool exitFlag = 0;
    DEVICES dv;

    // Arguments for Key-Value interface
    putArgs pargs; initPutArgs(&pargs);
    getArgs gargs; initGetArgs(&gargs);
    mergeArgs margs; initMergeArgs(&margs);
#ifndef DEBUG_FLAG
    printf("Mode -> %d (%s)\n");
#endif

    while (1) {
        // Wait for input from I/O process
        semop(semID, &p[SEM_MAIN_LOCK], 1);
        bool modeChanged = 0;

/* 1. Input Routine */
// Load switches input data from devices onto the shared memory
shmIn->key = -1;
for (swch = SWITCH_CNT; swch >= 1; swch--)
    shmIn->switches[swch] = 0;

// Read key input or switches input
if ((key = keyRead()) != 0)
    shmIn->key = key;
else {
    if ((swchs = switchRead()) != 0) {
        // RESET button pressed
        if (swchs == SWITCH_RESET) {
            for (swch = SWITCH_CNT; swch >= 1; swch--)
                shmIn->switches[swch] = 1;
        }
        // Otherwise like SW_1, SW_2, etc
        else {
            for (swch = SWITCH_CNT; swch >= 1; swch--) {
                shmIn->switches[swch] = swchs % 10;
                swchs = swchs / 10;
            }
        }
    }
}
// Sleep little bit for LCD input
usleep(130000);

// Synchronization
semop(semID, &v[SEM_MAIN_LOCK], 1);      // Allow main to proceed
semop(semID, &p[SEM_IO_LOCK_1], 1);      // Wait for main to be stopped

```

SEM_IO_LOCK_2은 Background Merge Process의 정상 동작을 위한 Semaphore이다. 본 Key-Value Store에선 Merge Process가 돌아가며 (아직 Merge되지 않은) On-Device SSTable의 개수가 3개 이상이 되면 자동으로 Background MERGE를 수행한다. 이때, 그러한 Merge가 아무런 Synchronization 없이 수행된다면 MERGE 수행 도중 PUT, GET의 발생을 막지 못한다. 즉, Persistency, Data Consistency가 언제든 깨질 수 있는 것이다.

따라서 Merge Process의 수행 순서도 고정시키자. 본인은 Background Merge Process

가 항상 I/O Process의 Output Routine 수행 직전에 일을 하도록 고정시켰다. 직전 Input에 대한 Main Process의 PUT/GET 작업까진 보장해준 다음, 그 직전 연산의 결과까지 고려한 다음 SSTable의 개수를 확인해 MERGE(Compaction)를 수행할지 말지 결정하는 것이다. 따라서 이를 위해 SEM_IO_LOCK_2와 SEM_MERGE_LOCK이 도입된 것이며, 그 동작 관계는 아래에서 확인할 수 있다. (β : I/O Process, δ : Merge Process)

```

semop(semID, &v[SEM_MERGE_LOCK], 1);      // Allow merge to proceed
semop(semID, &p[SEM_IO_LOCK_2], 1);        // Wait for merge to be stopped

/* 2. Output Routine */
for (dv = FND; dv <= MOTOR; dv++) {

/* Merge Process handler */
const int mergeProcess(const int semID) {
    mergeArgs margs; initMergeArgs(&margs);
    int swch;

    while (1) {
        semop(semID, &p[SEM_MERGE_LOCK], 1);      // wait for I/O

        // Merge process triggers compaction only if there's more than 3 sstables
        if ((shmMemtable->sstableCnt - shmMemtable->sstMergedCnt) >= 3) {
            // Automatically set RESET condition (for triggering)
            for (swch = SWITCH_CNT; swch >= 1; swch--)
                shmIn->switches[swch] = 1;

            // Merge (it must happen)
            Merge(&margs, 0);

            // Run MOTOR for a second
            shmOut->used[MOTOR] = 1;
            shmOut->motorOn = 1;
            shmMerge->bgMergeFlag = 1;           // Let IO process know merge happen (for MOTOR)
        }
        semop(semID, &v[SEM_IO_LOCK_2], 1);        // allow I/O to proceed
    }

    return 0;      // main function control
}
}

```

이렇게 해서 본 KV Store를 구성하는 세 프로세스의 Flow가 Synchronous하게 Fix된다. 이제 해당 프로세스들이 어떤 Method를 통해 Data를 주고 받으며 소통하는지 알아보자.

(3) IPC – Shared Memory

상기 항목처럼 Multiple Processes가 유기적인 소통을 하며 본 과제의 Key-Value Store를 작동시킨다. 이때, Process 간의 Data 소통은 보고서 초반에 언급한 바와 같이 <sys/sem.h>에서 제공하는 Shared Memory 기법을 사용한다. 첨부 자료를 참고하여 아래와 같이 간단한 Shared Memory 할당/해제 함수를 마련한다. (내부 구현 설명 생략)

```

int allocateShm(const key_t, void **, const size_t);
void freeShm(const int);

#endif /* _IPC_H_ */
(ipc.h 파일에 선언된 Shared Memory 할당/해제 함수)

```

이어 이 함수들을 사용해 Program 시작 시 각 Shared Memory들을 초기화한다. 이때, Shared Memory는 다음과 같다. 첫째, I/O Process와 Main Process가 소통할 때 사용하

는 Input Buffer와 Output Buffer이다. Input Buffer는 Input Device로부터 I/O Process가 읽어 들인 Data를 저장해 Main Process에게 전달하는 용도이며, Output Buffer는 반대로 Main Process가 Input을 기반으로 수행한 Key-Value Functionality의 결과물을 Output Device로 전달해 출력하고자 할 때 I/O Process와 Main Process가 소통하는 매개체이다.

상기 항목에서 언급한 것처럼 I/O Process는 Merge Process와 동기화되어 있는데, 이 과정에서도 Shared Memory가 필요하다. Merge Process가 MERGE를 수행하고 나면 MOTOR Device에 대한 출력이 요구되는데, 이 작업을 위해 Background MERGE 여부를 알리는 Flag가 두 프로세스 간에 공유되어야 한다.

한편, Memtable과 Key-Value Store Metadata는 Main Process와 Merge Process가 공유해야 하기 때문에 ‘SHM_MEMTABLE’이라는 Shared Memory로 필요하다. 단순히 Memtable 정보뿐만 아니라 SSTable 개수, Merged SSTable 개수, Total Flushed Pair Count와 같은 KV Store 전반의 정보가 담겨 있음을 주목하자.

```
/* Shared memory between Main & IO processes (for input) */
typedef struct _shmMainIn {
    bool switches[SWITCH_CNT + 1]; // Switch 1 to 9
    int key;                      // VOL+, VOL-, BACK
} SHM_MAIN_IN;

// There's a max limit for the size of shared memory, so I divide into two
/* Shared memory between Main & IO processes (for output) */
typedef struct _shmMainOut {
    int fndBuf;                  // Buffer for FND device
    int ledBuf;                  // Buffer for LED device
    char textLCDBuf[MAX_TEXT_LEN]; // Buffer for TEXT_LCD device
    bool motorOn;                // Whether the motor is on/off
    bool used[O_DEVICE_CNT];     // Which device is in use
    bool waitFlag;               // Wait for a second (PUT, GET, MERGE success)
} SHM_MAIN_OUT;

/* Shared memory between Merge & IO processes */
typedef struct _shmMergeIO {
    bool bgMergeFlag;            // Background MERGE occur or not
} SHM_MERGE_IO;

/* Shared memory between Main & Merge processes */
typedef struct _shmMemtable {
    int keys[MAX_PAIR];          // Keys
    char values[MAX_PAIR][MAX_TEXT_LEN]; // Values
    int pairCnt;                 // # of pairs in memtable
    int totalPutCnt;              // # of flushed pairs
    int sstableCnt;               // # of sstables
    int sstMergedCnt;             // # of merged sstables
} SHM_MEMTABLE;

/* Share Memories */
SHM_MAIN_IN *shmIn;           // For IPC of Input-Main
SHM_MAIN_OUT *shmOut;          // For IPC of Output-Main
SHM_MERGE_IO *shmMerge;        // For IPC of Merge-IO
SHM_MEMTABLE *shmMemtable;     // Globally shared memtable & KV store metadata
```

이러한 총 4개의 Shared Memory가 필요하며, 더 자세한 설명은 주석으로 대체한다. 이 4개의 공유 영역은 본 프로그램 시작 시 initProgram이라는 프로그램 초기화/설정 함수에서 아래 코드와 같이 초기화된다. Shared Memory에 대한 초기화뿐만 아니라 Semaphore Allocation, Device Open, Metadata Open 등의 행위가 함께 이뤄짐을 주목하자.

```

/* Initializing routines of this application */
const void initProgram(int *semID, int *shmInID, int *shmOutID, int *shmMergeID, int *shmMemID) {
    // Allocate a semaphore
    *semID = allocateSem();

    // Allocate shared memories
    *shmInID = allocateShm(SHM_KEY_1, (void**) &shmIn, sizeof(SHM_MAIN_IN));
    *shmOutID = allocateShm(SHM_KEY_2, (void**) &shmOut, sizeof(SHM_MAIN_OUT));
    *shmMergeID = allocateShm(SHM_KEY_3, (void**) &shmMerge, sizeof(SHM_MERGE_IO));
    *shmMemID = allocateShm(SHM_KEY_4, (void**) &shmMemtable, sizeof(SHM_MEMTABLE));

    // Initialize shared memories
    memset(shmIn->switches, 0, sizeof(shmIn->switches));
    shmIn->key = -1;
    shmOut->fdnBuf = shmOut->ledBuf = 0; shmOut->motorOn = shmOut->waitFlag = 0;
    memset(shmOut->textlcdBuf, '\0', sizeof(shmOut->textlcdBuf));
    memset(shmOut->used, 0, sizeof(shmOut->used));

    shmMerge->bgMergeFlag = 0;

    memset(shmMemtable->keys, 0, sizeof(shmMemtable->keys));
    memset(shmMemtable->values, '\0', sizeof(shmMemtable->values));
    shmMemtable->pairCnt = shmMemtable->totalPutCnt
        = shmMemtable->sstMergedCnt = shmMemtable->sstableCnt = 0;

    // Open all the devices
    devOpen();

    // Load existing metadata of key-value store
    loadMetadata();
}

```

참고로 Program 종료 시에는 아래와 같은 함수가 호출된다. I/O Process와 Merge Process를 죽이는 것뿐만 아니라 Device Close, Semaphore/Shared Memory 해제, Metadata Persist 과정까지 수행되고 있음을 주목하자.

```

/* Clearing routines when termination */
const void exitProgram(const int semID, const int shmInID, const int shmOutID, const int shmMergeID, const int shmMemID) {
    // Kill all the child processes
    killChildren();

    // Close all the devices
    devClose();

    // Free all the IPCs
    freeSem(semID);
    freeShm(shmInID);
    freeShm(shmOutID);
    freeShm(shmMergeID);
    freeShm(shmMemID);

    // Persist metadata of key-value store
    persistMetadata();
}

```

이렇게 해서 Multiple Process의 유기적 동작 관계를 구축할 수 있다. 이제 본 Program의 핵심 기능은 Key-Value Interface에 대해 알아볼 것인데, 그에 앞서 잠시 Persistency에 대해 짧게 소개하겠다. 위 initProgram, exitProgram 함수에서 알 수 있듯 본 Program은 Key-Value Store, 즉, Database / Storage이기 때문에 Program 종료 시에도 Data를 정상적으로 유지할 수 있어야 한다. Program 종료 후에도 여전히 Data가 유지되어야 한다.

따라서 Persistency를 위한 작업이 필요한데, 기본적으로 KV Pairs 자체는 SSTable File의 형태로 Disk에 기록한다. 이때 Program은 Program 시작 시에 현재 Disk 상에 있는

모든 SSTable에 대한 정보를 Load할 수 있어야 한다. 그래야 과거의 Data를 정상적으로 해석할 수 있기 때문이다. 반대로 Program 종료 시엔 SSTable의 현 상태를 기록할 수 있어야겠다. 이러한 기능을 담당하는 함수들이 바로 아래의 두 함수이다.

```
/* Load metadata of Key-Value Store */
const void loadMetadata(void) {
    char fileName[MAX_TEXT_LEN];
    bool loadedFlag = 0;
    FILE *sst;

    sprintf(fileName, "kvstore_meta.txt");
    if ((sst = fopen(fileName, "r")) == NULL)
        loadedFlag = 0;
    if (loadedFlag) {
        fscanf(sst, "%d %d %d", &(shmMemtable->sstableCnt), &(shmMemtable->sstMergedCnt),
               &(shmMemtable->totalPutCnt));
        fclose(sst);
    }
#endif DEBUG_FLAG
    if (loadedFlag)
        printf("[PERSIST] Metadata %d, %d, %d is loaded!\n", shmMemtable->sstableCnt,
               shmMemtable->sstMergedCnt, shmMemtable->totalPutCnt);
    else printf("[PERSIST] There's no previous metadata in our system.\n");
#endif
}
```

loadMetadata 함수는 함수명대로 Metadata를 Load하는 역할이다. Metadata는 ‘kvstore_meta.txt’라는 Disk File에 기록되는데, 만약 그러한 File이 기존에 존재하지 않는다면 그것은 첫 번째 KV Store 동작 상황을 의미한다. Metadata에는 SSTable 개수, MERGE된 SSTable 개수, Flush된 총 Pair 개수가 있음을 알 수 있다.

```
/* Persist metadata of Key-Value Store */
const void persistMetadata(void) {
    char fileName[MAX_TEXT_LEN];
    FILE *sst;
    int i;

    // If there's more than one entry in the memtable, then persist it!
    if (shmMemtable->pairCnt) {
        sprintf(fileName, "%d.sst", ++(shmMemtable->sstableCnt));
        if ((sst = fopen(fileName, "w")) == NULL)
            perror("Error occurs while persisting metadata (memtable)\n");
        for (i = 0; i < shmMemtable->pairCnt; i++)
            fprintf(sst, "%d %d %s\n", (++shmMemtable->totalPutCnt),
                    shmMemtable->keys[i], shmMemtable->values[i]);
        fclose(sst);
#endif DEBUG_FLAG
        printf("[PERSIST] Pairs in memtable ");
        for (i = 0; i < shmMemtable->pairCnt; i++)
            printf("- %d, %s\n", shmMemtable->keys[i], shmMemtable->values[i]);
        printf("are(is) persisted to File %s\n", fileName);
#endif
    }
#endif DEBUG_FLAG
    else printf("[PERSIST] There's no entries to persist in Memtable\n");
#endif

    sprintf(fileName, "kvstore_meta.txt");
    if ((sst = fopen(fileName, "w")) == NULL)
        perror("Error occurs while persisting metadata\n");
    fprintf(sst, "%d %d %d\n", shmMemtable->sstableCnt,
            shmMemtable->sstMergedCnt, shmMemtable->totalPutCnt);
    fclose(sst);

#endif DEBUG_FLAG
    printf("[PERSIST] Metadata %d, %d, %d is persisted!\n", shmMemtable->sstableCnt,
           shmMemtable->sstMergedCnt, shmMemtable->totalPutCnt);
#endif
}
```

`persistMetadata` 함수는 당연하게도 `loadMetadata` 함수의 반대 기능을 수행하며, 위 코드에서 쉽게 이해할 수 있듯 SSTable 개수 등의 정보를 ‘kvstore_meta.txt’ 파일에 업데이트하는 기능을 수행한다. Program 종료 시 Memtable에 Entry가 최소 하나라도 있으면 Memtable을 그대로 Freshest SSTable로 Persist하고 있음도 주목하자.

참고로 본인은 아래와 같은 SIGINT Signal Handler를 본 Program(Main Process)에 설계하여 Crash(여기선 CTRL+C 입력을 상정) 시에도 여전히 KV Store가 Persistency를 유지할 수 있도록 설계하였다.

```
/* SIGINT handler for persistency in case of crash */
static void sig_handler(int signo) {
    persistMetadata();
    exit(0);
}

/* main Process */
int main(void) {
    int semID, shmInID, shmOutID, shmMergeID, shmMemID, ret;
    initProgram(&semID, &shmInID, &shmOutID, &shmMergeID, &shmMemID)

    // Forks four processes
    switch (forks()) {
        case _MAIN:           // Main process
            // Signal Handling of CTRL+C pressed situation
            signal(SIGINT, (void*)sig_handler);
            ret = mainProcess(semID); break;
        case _IO:              // I/O process
            ret = ioProcess(semID); break;
        case _MERGE:           // Merge process
            ret = mergeProcess(semID); break;
        default: perror("Error occurs during the process creation\n");
    }
    if (ret == 1)
        exitProgram(semID, shmInID, shmOutID, shmMergeID, shmMemID);

    return 0;
}
```

(4) Key-Value Interface

본격적인 Key-Value Interface에 대한 설명에 앞서, 먼저 Key-Value Interface를 Input에 따라 분기 수행하는 Main Process Handler부터 확인해보자. Main Process Handler는 아래와 같은 무한 Loop를 수행한다.

```
while (1) {
    // Wait for input from I/O process
    semop(semID, &p[SEM_MAIN_LOCK], 1);
    bool modeChanged = 0;

    // Verify key input
    switch (shmIn->key) {
        case KEY_VOLUMEDOWN: // Prev mode
            if (mode == PUT)
                mode = MERGE;
            else mode--;
            modeChanged = 1;
            break;
    }
}
```

```

case KEY_VOLUMEUP:           // Next mode
    mode++;
    mode %= MODE_CNT;
    modeChanged = 1;
    break;
case KEY_BACK:               // Exit
    exitFlag = 1;
    break;
default: break;
}
// Clear the output when the mode changed
if (modeChanged) {
    shmOut->fnBuf = 0;
    memcpy(shmOut->textLCDBuf, initialValueBuf, MAX_TEXT_LEN);
}
shmIn->key = -1;
DEBUG_FLAG
if (modeChanged)
    printf("\nMode Changed -> %d\n(PUT:0, GET:1, MERGE:2)\n", mode);

if (exitFlag)
    break;

switch (mode) {
case PUT: Put(&pargs, modeChanged); break;           // PUT
case GET: Get(&gargs, modeChanged); break;           // GET
case MERGE: Merge(&margs, modeChanged); break;        // MERGE
default: break;
}

// Allow input process to work
semop(semID, &v[SEM_IO_LOCK_1], 1);

```

그렇다. KEY Device의 입력을 확인하여 Mode를 변경하고, 해당 Mode가 의미하는 Key-Value API Function을 호출한다. 지금부터 kv_api.h, kv_api.c에 정의된 Key-Value API 구현을 확인하자. 먼저 PUT이다. ‘Put’ 함수는 Wrapper이다. (Get, Merge 동일)

```

/* API wrapper function for PUT operation */
void Put(putArgs *args, const bool modeChanged) {
    bool noResetFlag = 0;
    int i;

    for (i = 0; i <= SWITCH_CNT; i++)
        args->keypad[i] = 0;

    // First time for current mode
    if (modeChanged)
        initPutArgs(args);

    // Check if RESET key is pressed
    for (i = 1; i <= SWITCH_CNT; i++) {
        if (!shmIn->switches[i]) {
            noResetFlag = 1;
            break;
        }
    }
}

```

```

    if (!noResetFlag)
        args->resetFlag = 1;
    else {
        // Verify switches input
        if (shmIn->switches[2] && shmIn->switches[3])
            args->clearFlag = 1;
        else if (shmIn->switches[5] && shmIn->switches[6])
            args->numAlphaMode = 1; // Number input or alphabet input
        else if (shmIn->switches[8] && shmIn->switches[9])
            args->memtableInsert = 1; // Insert to the memtable
    }

    // Set keypad inputs
    for (i = 0; i <= SWITCH_CNT; i++)
        args->keypad[i] = shmIn->switches[i];

    _Put(args);
}

```

Input Device로부터의 KEY, SWITCH Input을 확인하며 상응하는 Action을 실제 PUT 수행 함수 '_Put'에게 전달하고 있다. 각 Action은 명세서에 명시된 조건을 따르며, RESET Button 입력 여부를 확인하는 부분을 주목하자.

아래는 '_Put' 함수이다. PUT 연산 수행을 위한 Key, Value 입력을 통제하며, 각 입력 시의 Board Output Device 출력물 제어, Memtable Insertion 등을 수행한다.

```

/* PUT operation handler */
const void _Put(putArgs *args) {
    static char key[MAX_KEY_LEN + 1];
    static char value[MAX_TEXT_LEN + 1];
    static char inserted[MAX_TEXT_LEN + 1];
    static int index = 0, prev = -1, option = 0, leds = LED_1;
    static bool numMode = 1, inputStart = 0;
    char temp[MAX_KEY_LEN + 1], c = -1;
    bool changed = 0, insertSuccess;
    int order, fnd, i;

    // Turn on devices we need here
    shmOut->used[LED] = 1;
    if (args->keyInputModule) {
        shmOut->used[FND] = 1;
        shmOut->used[TEXT_LCD] = 0;
    }
    else {
        shmOut->used[FND] = 0;
        shmOut->used[TEXT_LCD] = 1;
    }
    shmOut->used[MOTOR] = 1; shmOut->motorOn = 0;

    // Get the device time
    time_t devTime; time(&devTime);
    const struct tm *timeInfo = localtime(&devTime);
    const int devSecond = timeInfo->tm_sec;

    // Initialize if it's first time
    if (args->firstFlag) {
        strncpy(key, "0000", MAX_KEY_LEN);
        strncpy(value, initialValueBuf, MAX_TEXT_LEN);
        numMode = 1; // Default is number input mode
        index = 0; // Start filling the text string from idx 0
        prev = -1; // Previous keypad info (for alphabet mode)
        option = 0; // Sub-option of specific key (for alphabet mode)
    }
}

```

```

    changed = 0;           // Is it changed?
    inputStart = 0;         // Initial LED is LED_1
    leds = LED_1;          // Turn on the LED_1
    args->keyInputMode = 1; // Default is key input mode
    args->firstFlag = 0;   // Reset the flag
}

```

최초로 PUT Mode가 실행될 때의 처리가 나타나 있다. 먼저 PUT에서 필요한 Output Device들을 지정한다. 이어, ‘1초마다 깜빡이는 처리’를 위한 시간 초 설정을 진행하고, 그 다음 PUT 연산에서 사용되는 각종 Flag를 초기화한다. 각 Flag에 대한 설명은 주석으로 대체한다. 매 Mode 전환 시 항상 상기한 것과 같은 Flag 초기화 루틴이 동작한다.

```

// Case1: Change the mode (key to value)
else if (args->resetFlag) {
    args->keyInputModule = 0;
    index = 0; prev = -1; option = 0;
    args->resetFlag = 0;
}
// Case2: Clear the string
else if (args->clearFlag) {
    if (args->keyInputModule)
        strncpy(key, "0000", MAX_KEY_LEN);
    else strncpy(value, initialValueBuf, MAX_TEXT_LEN);
    index = 0; prev = -1; option = 0; changed = 1;
    args->clearFlag = 0;
}
// Case3: Change the alphabet <-> number (only in value input mode)
else if (args->numAlphaMode && !args->keyInputModule) {
    numMode = !numMode;      // Reverse num/alpha mode
    prev = -1; option = 0;
    args->numAlphaMode = 0;
}

```

이어 각 입력에 대한 처리를 수행한다. RESET Button이 눌렸을 땐 Key 입력에서 Value 입력으로의 전환이 이뤄지고, SWITCH 2, 3번이 동시에 눌렸을 땐 (앞선 Wrapper에서 이를 확인해 clearFlag를 Set) Key, Value String들을 초기화하고 있다. SWITCH 5, 6번이 동시에 눌렸을 땐 입력할 Character를 Number에서 Alphabet으로, 또는 Alphabet에서 Number로 전환하고 있음을 주목하자.

```

// Case4: Memtable insertion
else if (args->memtableInsert) {
    #if DEBUG_PROJ
        printf("Memtable before insert\n");
        for (i = 0; i < shmMemtable->pairCnt; i++)
            printf("%d %d %s\n", shmMemtable->totalPutCnt + i + 1,
                   shmMemtable->keys[i], shmMemtable->values[i]);
    #if
        key[4] = '\0';      // Set null character (for preventing any error)
        insertSuccess = insertMemtable(key, value, &order); // Insert
    #if DEBUG_PROJ
        printf("Memtable after insert\n");
        for (i = 0; i < shmMemtable->pairCnt; i++)
            printf("%d %d %s\n", shmMemtable->totalPutCnt + i + 1,
                   shmMemtable->keys[i], shmMemtable->values[i]);
    #if

```

자, 이제 Memtable Insertion Case이다. 앞서 확인한 Wrapper에서 확인할 수 있듯, SWITCH 8, 9번이 동시에 눌렸을 때 Trigger되는 상황이다. ‘insertMemtable’이라는 함수를 호출해 작업하고 있으며, 그 앞뒤로 Debugging 목적의 Print문들이 설치되어 있다. ‘진정한 PUT’은 ‘insertMemtable’에서 수행한다. 이 함수는 잠시 뒤 확인하자.

```
// Turn on all the LEDs for a second
leds = LED_1 | LED_2 | LED_3 | LED_4 |
       LED_5 | LED_6 | LED_7 | LED_8;
prev = -1; option = 0; changed = 1;
inputStart = 0; // To turn on only LED_1
args->memtableInsert = 0;
args->firstFlag = 1;

// Print inserted information for a second
if (insertSuccess) {
    for (i = 0; ; i++) {
        if (i == MAX_TEXT_LEN - 1 || value[i] == ' ') {
            value[i] = '\0';
            break;
        }
    }
    sprintf(inserted, "(%d, %s, %s)", shmMemtable->totalPutCnt + (order + 1), key, value);
    memcpy(shmOut->textLCDBuf, inserted, MAX_TEXT_LEN);
}

// Initialize for the next input
strncpy(key, "0000", MAX_KEY_LEN);
shmOut->fnBuf = 0;
```

위 코드 블록은 ‘insertMemtable’ 함수 호출 이후의 Flow를 보여준다. Insertion이 마무리되면 LED를 몇 초간 전부 On하기 위해 LED Buffer를 설정한다. 이어 Flag 값 정비를 진행하고, Insertion이 성공적으로 이뤄졌다면(Key, Value String이 비정상(ex. 0000)일 때를 제외하곤 항상 성공함) PDF에서 요구하는 출력 양식을 구성하고 있음을 알 수 있다.

그 밖의 단순 Character(Number / Alphabet) 삽입 Case는 아래와 같이 처리한다.

```
// Case4: Just insert character
else {
    for (i = 1; i <= SWITCH_CNT; i++) {
        if (args->keypad[i]) {
            // Number mode
            if (numMode) {
                c = '0' + i;
                prev = -1; option = 0;
            }
            // Alphabet mode (only in value input mode)
            else if (!args->keyInputMode) {
                if (i == prev)
                    option = (option + 1) % 3;
                else {
                    if (prev != -1)
                        index++;
                    prev = i; option = 0;
                }
                c = KEYPAD[i][option];
            }
            inputStart = changed = 1;
            break;
        }
    }
}
```

Alphabet 처리 시 직전 Alphabet Group에 대한 정보를 토대로 Index 진행을 할지 말지 결정하고 있음을 주목하자. PDF에서 요하는 Key 입력 Pattern을 따르기 위한 처리이다.

아래를 보자. Key Input 또는 Value Input이 시작되었을 때의 ‘LED 깜빡임’ 처리가 이뤄지고 있음을 알 수 있다. 현재 Device Time을 확인해 간단한 MOD 연산을 진행해 처리한다. 이어 Key 또는 Value String에 대해 Character를 Append하는 루틴이 나타나 있으며, Key Input Mode일 때는 알파벳 문자가 허용되고 있지 않음을 주목하자.

```
// If input started, then keep turning on corresponding LEDs
if (inputStart) {
    if (args->keyInputMode) {
        if ((devSecond % 2) == 0)
            leds = LED_3;
        else leds = LED_4;
    }
    else {
        if ((devSecond % 2) == 0)
            leds = LED_7;
        else leds = LED_8;
    }
}
shmOut->ledBuf = leds;
if (!changed)
    return;

// Append char to the string
if ((c >= '0' && c <= '9') || (c >= 'A' && c <= 'Z')) {
    // Insert in a circular manner
    if (args->keyInputMode) {
        if (index >= 40)      // FND (key input)
            index = 0;
    }
    else {
        if (index >= MAX_TEXT_LEN)
            index = 0;          // LCD (value input)
    }
    // Alphabet is banned to be inserted when key input mode
    if (!(args->keyInputMode && !numMode)) {
        if (args->keyInputMode)
            key[index++] = c;
        else {
            value[index] = c;
            if (numMode)
                index++;
        }
    }
}
```

이어지는 부분에선 아래와 같이 Output Buffer에 대한 조정이 이뤄지고 있다. Memtable Insertion (성공) 시에는 waitFlag라는 것을 활성화시키고 있는데, 이 Flag는 I/O Process 가 약 2-3초간 대기하게 하는 Flag로, PUT, GET, MERGE 성공 시 해당 성공 여부를 보드 사용자 입장에서 명확히 확인할 수 있도록 잠시 Output 상태를 유지하려는 기능이다.

```
// Print to FND/LCD/LED via shared memory (while input)
if (linsertSuccess) {
    if (args->keyInputMode) {
        strncpy(temp, key, MAX_KEY_LEN);
        fnd = atoi(temp);
        shmOut->fndBuf = fnd;
    }
    else memcpy(shmOut->textLCDBuf, value, MAX_TEXT_LEN);
}
else shmOut->waitFlag = 1;
```

다시 돌아와서 ‘insertMemtable’ 함수를 확인하자.

```
/* Insert key-value pair into the memtable */
const bool insertMemtable(const char *keyStr, const char *valueStr, int *order) {
    char fileName[MAX_SSTABLE + 5];      // ex) 1234.sst
    char key[MAX_KEY_LEN + 1];
    strncpy(key, keyStr, MAX_KEY_LEN);
    FILE *sst; int i, keyNum;

    // If key or value is as same as default, then decline the request
    if (!strcmp(keyStr, "0000", MAX_KEY_LEN + 1) ||
        !strcmp(valueStr, initialValueBuf, MAX_TEXT_LEN))
        return 0;

    // Flush if memtable is full
    if (shmMemtable->pairCnt == 3) {
        shmMemtable->sstableCnt++;
        sprintf(fileName, "%d.sst", shmMemtable->sstableCnt);
        if ((sst = fopen(fileName, "w")) == NULL)
            perror("Error occurs while opening SSTable (PUT)\n");
        for (i = 0; i < MAX_PAIR; i++)
            fprintf(sst, "%d %d %s\n", (++shmMemtable->totalPutCnt),
                    shmMemtable->keys[i], shmMemtable->values[i]);

        shmMemtable->pairCnt = 0;
        fclose(sst);
    #ifdef DEBUG_PROJ
        printf("Flush occurs!\n");
        int tempNo, tempKey; char tempVal[MAX_TEXT_LEN];
        if ((sst = fopen(fileName, "r")) == NULL)
            perror("Error occurs while opening SSTable (PUT_DEBUG)\n");
        printf("FileName: %s\n", fileName);
        for (i = 0; i < MAX_PAIR; i++) {
            fscanf(sst, "%d %d %s", &tempNo, &tempKey, &tempVal);
            printf("%d %d %s\n", tempNo, tempKey, tempVal);
        }
    #endif
    }

    // Memtable insertion
    keyNum = atoi(key);
    if (keyNum > 9999) keyNum /= 10;      // Just for avoiding any error
    shmMemtable->keys[shmMemtable->pairCnt] = keyNum;      // Inserted as integer (for comparison when MERGE)
    strncpy(shmMemtable->values[shmMemtable->pairCnt], valueStr, strlen(valueStr));
    *order = shmMemtable->pairCnt;      // For printing
    shmMemtable->pairCnt++;

    return 1;
}
```

위와 같다. Shared Memory 영역을 확인해 현재 Memtable의 Pair 개수가 이미 3개(Max)라면 PUT Request 발생 시 Flush가 이뤄져야 한다. 상기 코드의 두 번째 if문을 통해 해당 처리를 확인할 수 있으며, Flush 시엔 sstableCnt라는 Metadata를 통해 가장 최근에 생성된 SSTable Number에서 1을 더한 Number의 SSTable이 생성되고 있음을 확인할 수 있다(동시에 totalPutCnt라는 이름의 Flushed Pair 개수 Counting Variable에 1을 더하고 있음도 주목하자. (보고서 작성 시 추가하는 이에 관한 Comment: 사실 ‘totalPutCnt’ 변수의 이름은 ‘totalFlushedPairCnt’라는 이름이 더 알맞은 것 같다)).

이어 Flush 여부와 상관없이 Memtable 삽입이 이뤄진다.

GET, MERGE의 경우도 PUT과 마찬가지로 Wrapper(ex. Get, Merge), Handler(_Get, _Merge)의 관계가 정립되어 있다. 따라서 이들에 대한 설명은 생략하며, 지금부터는 각 Operation이 호출하는 ‘진정한 Handler’인 ‘searchTables’와 ‘compactTables’ 함수들을 확인하자. 두 함수 모두 이름 그대로, 전자는 Key-Value Store의 Memtable, SSTables에서 Target Key에 대한 Value를 Retrieve하는 함수이고, 후자는 가장 Oldest SSTable 2개에

대해 Victim으로 선정하여 Compaction을 수행하는 함수이다. 우선, 전자부터 확인하자. 아래를 보자.

```
/* Retrieve the latest value of passed key */
const int searchTables(const char *keyStr, char *valueStr, int *order) {
    char fileName[MAX_SSTABLE + 5];      // ex) 1234.sst
    char temp[MAX_KEY_LEN + 1];
    strncpy(temp, keyStr, MAX_KEY_LEN);
    int key = atoi(temp), i, tempNo, tempKey, valueFound = 0;
    char tempValue[MAX_TEXT_LEN];
    FILE *sst;

    // First search key in the memtable
    if (shmMemtable->pairCnt > 0) {
        for (i = (shmMemtable->pairCnt - 1); i >= 0; i--) {
            if (key == shmMemtable->keys[i]) {
                strncpy(valueStr, shmMemtable->values[i], strlen(shmMemtable->values[i]));
                *order = i;
                valueFound = 1;
                break; // The first found value is the latest (cuz it searches from end)
            }
        }
    }
#endif DEBUG_PROJ
    if (valueFound)
        printf("Target value is retrieved from the memtable!\n");
    else {
        printf("Memtable doesn't have the target value, so let's search sstables\n");
        printf("* Current # of sstables: %d * Current # of merged ssts: %d\n",
               shmMemtable->sstableCnt, shmMemtable->sstMergedCnt);
    }
#endif
    // If search failed in the memtable, then let's try same thing in sstables
    if (!valueFound && shmMemtable->sstableCnt > 0) {
        // Start searching from the latest sstable
        for (i = shmMemtable->sstableCnt; i >= (shmMemtable->sstMergedCnt + 1); i--) {
            sprintf(fileName, "%d.sst", i);
            if ((sst = fopen(fileName, "r")) == NULL)
                perror("Error occurs while opening SSTable (GET)\n");
            while (fscanf(sst, "%d %d %s", &tempNo, &tempKey, tempValue) != EOF) {
                if (tempKey == key) {
                    *order = tempNo;
                    strncpy(valueStr, tempValue, strlen(tempValue));
                    valueFound = 2; // The lastly found value is the latest
                }
            }
            fclose(sst);
            if (valueFound) { // If found, then stop
#ifndef DEBUG_PROJ
                printf("Target value is retrieved from sstable %s!\n", fileName);
#endif
                break;
            }
        }
    }
#endif DEBUG_PROJ
    if (!valueFound)
        printf("Target value cannot be found, so, it's an error!\n");
#endif
    // If program reach here without strncpy, then GET finally fails
    return valueFound;
}
```

Target Key에 대해 먼저 Memtable을 Search한다. Memtable에는 Key-Value Pair가 Ascending Order로 Number를 부여받아 저장되어 있는데, 이때 Order가 높을수록 Fresh한 Data임은 자명하다. 따라서, 위 코드를 보면, 저장 순서, 정확히는 Memtable을 형성하는 Array를 뒤에서부터 읽으면 Target Key가 발견될 시 그대로 Search를 멈추고 Value를 반환한다. First-Fit 논리로 발견된 Key가 곧 Freshest일 것이기 때문이다.

만약 Memtable에서 Search가 실패하면 연이어 On-Disk SSTables에 대한 Search를 진행한다. 복수의 On-Disk SSTables 중 가장 Fresh한 SSTable은 어떤 Table일까? 그렇다. File Name Number가 가장 큰 SSTable, 즉, (당연하게도) 가장 최근에 Create된 SSTable일 것이다. 따라서 상기 코드를 확인해보면 sstableCnt라는 Metadata를 활용하여 Freshest SSTable File부터 Open하여 Search를 진행함을 알 수 있다.

SSTable File을 Open하면 해당 File에 대한 Scanning 작업이 필요하다. 해당 File에는 몇 개의 KV Pair가 존재할지 알 수 없기 때문에 fscanf 함수가 실패할 때까지 Scan을 지속한다. 이때 Desired Key에 대한 (Multiple하게 존재할 수 있는) Corresponding Value 중 어떤 Value가 가장 Freshest일까? 그렇다. 가장 마지막에 발견된 Value일 것이다. Memtable의 Pair 삽입 순서를 생각하면 굉장히 자명한 서술이다. 따라서 본 코드에서도 그러한 논리를 따라 SSTable Search 시엔 중간에 Target Key가 발견되더라도 끝까지 File을 훑으며 Search를 진행한다. 가장 늦게 발견된 Value가 Target Value이다.

아래는 MERGE Operation을 담당하는 ‘compactTables’이다. 우선 코드부터 확인하자.

```
/* Trigger compaction: merge two sstables with sorting pairs and create new table */
const int compactTables(char **sstableName) {
    char fileName[MAX_SSTABLE + 5]; // ex) 1234.sst
    char tempName[MAX_SSTABLE + 5];
    char tempValue[MAX_PAIR_NUM][MAX_TEXT_LEN], c;
    int valuePointer[MAX_PAIR_NUM], tempKey[MAX_PAIR_NUM];
    int mergeStartFileNo = shmMemtable->sstMergedCnt + 1;
    int valueStables = shmMemtable->sstableCnt - shmMemtable->sstMergedCnt;
    int i, j, totalPairCnt = 0, pairCnt = 0, temp;
    bool passFlag, putNoReadFlag = 0;
    FILE *sst, *tempSst;

    // If there's more than 2 sstables in the disk, then perform merge!
    if (shmMemtable->sstableCnt && valueStables >= 2) {
        // Get all the pairs in two victim sstables
        for (i = mergeStartFileNo; i <= mergeStartFileNo + 1; i++) {
            sprintf(fileName, "%d.sst", i);
        #ifdef DEBUG_PROJ
            printf("Selected victim of compaction: %s\n", fileName);
        #endif
            if ((sst = fopen(fileName, "r")) == NULL)
                perror("Error occurs while opening SSTable (MERGE)\n");
            while (fscanf(sst, "%d %d %s", &temp, &tempKey[pairCnt], tempValue[pairCnt]) != EOF) {
                // Record the PUTed number of first found entry
                if (!putNoReadFlag) { // for proper counting of new sstable
                    putNoReadFlag = 1;
                    totalPairCnt = temp - 1;
                }
                valuePointer[pairCnt] = pairCnt; // Value pointer for avoiding costly value copying
                passFlag = 0;

                // If current key is already found before, then only get value and copy to prev entry
                for (j = 0; j < pairCnt; j++) {
                    if (tempKey[pairCnt] == tempkey[j]) {
                        strncpy(tempValue[j], tempValue[pairCnt], MAX_TEXT_LEN);
                        passFlag = 1; // And don't increment the index (for omitting this key)
                        break;
                    }
                }
            }
        }
    }
}
```

```

        if (!passFlag) // Increment the index only if the key is found for the first time
            pairCnt++;
        totalPairCnt++; // Total pair count for counting
    }
    fclose(sst);
}
#endif DEBUG_PROJ
printf("Temporary Storage for pairs\n");
for (i = 0; i < pairCnt; i++)
    printf("%d %s\n", tempKey[i], tempValue[valuePointer[i]]);
#endif
// Bubble sort for key sorting (ascending order)
for (i = 0; i < pairCnt - 1; i++) {
    for (j = 0; j < pairCnt - i - 1; j++) {
        if (tempKey[j] > tempKey[j + 1]) {
            temp = tempKey[j];
            tempKey[j] = tempKey[j + 1];
            tempKey[j + 1] = temp;

            // Value pointer literally points where real value is
            temp = valuePointer[j]; // for speedy merge operation
            valuePointer[j] = valuePointer[j + 1];
            valuePointer[j + 1] = temp;
        }
    }
    // Actual values are never moved, only pointers are moved
}
#endif DEBUG_PROJ
printf("Temporary Storage for pairs (after sorting)\n");
for (i = 0; i < pairCnt; i++)
    printf("%d %s\n", tempKey[i], tempValue[valuePointer[i]]);
#endif
// Make a room for new sstable, by pushing all the leftsstables
for (i = shmMemtable->sstableCnt; i >= mergeStartFileNo + 2; i--) {
    sprintf(fileName, "%d.sst", i);
    sprintf(tempName, "%d.sst", i + 1); // Change the name of file
    if ((sst = fopen(fileName, "rb")) == NULL)
        perror("Error occurs while opening STable (MERGE)\n");
    if ((tempSst = fopen(tempName, "wb")) == NULL)
        perror("Error occurs while opening STable (MERGE)\n");

    // Just copy the contents of file
    while (!feof(sst)) {
        fread(&c, sizeof(char), 1, sst);
        fwrite(&c, sizeof(char), 1, tempSst);
    }
    fclose(sst);
    fclose(tempSst);
}
// Create a new sstable!
sprintf(fileName, "%d.sst", mergeStartFileNo + 2);
if ((sst = fopen(fileName, "w")) == NULL)
    perror("Error occurs while opening STable (MERGE)\n");
for (i = 0; i < pairCnt; i++)
    fprintf(sst, "%d %d %s\n", (totalPairCnt - pairCnt + i + 1),
            tempKey[i], tempValue[valuePointer[i]]);
fclose(sst);
#endif DEBUG_PROJ
printf("Created new sstable %s\n", fileName);
#endif
shmMemtable->sstMergedCnt += 2;
shmMemtable->sstableCnt += 1;
strncpy(sstableName, fileName, strlen(fileName));
}
#endif DEBUG_PROJ
else printf("There's no MERGE needed here!\n");
#endif
// If there's no merge, then pairCnt would be zero!
return pairCnt;
}

```

(Code-Level Detail은 다음 장에서 서술)

우선 첫 번째 if문을 보자. Disk에 (Merge되지 않은) SSTable의 개수가 2개 이상부터 MERGE가 가능하다. Background Merge Process에선 SSTable의 개수가 3개일 때 연산을 Request할 수 있는 반면, Main Process의 MERGE Mode에선 2개의 SSTable에 대해서도 MERGE할 수 있음이 큰 차이이다. (Merge Process에선 자체 Condition문을 통해 이를 제어하고 있음)

그렇게 MERGE가 시작되면 다음의 순서로 일이 진행된다.

1) Select victim SSTables

-> 먼저 Oldest SSTable 두 개를 선정한다. 'sstMergedCnt'라는, 'Merged SSTable의 개수'를 추적하는 Metadata를 활용한다. 가장 최근에 Merge된 SSTable의 File Name Number에서 1, 2를 더하면 현재 MERGE의 Victim SSTables를 쉽게 지정할 수 있다.

2) Scan all the pairs of victims

-> 그 다음, 두 Victim SSTables를 fscanf로 Scan하며 Pair들을 찾아 Memory에 임시 보관한다. 이때, SSTable의 의미를 고려했을 때 Key는 항상 Unique하게 가져야 하므로 앞서 확인된 Key가 다시 등장하면 Value만 기존 Value Entry of previously found key로 Copy하고 그냥 Pass(Index를 늘리지 않음)한다.

3) Perform bubble sort on temporary array of scanned KV pairs

-> 1), 2)로 인해 구축된 In-Memory 임시 배열에 대해 Key값을 기준으로 Bubble Sort를 수행한다. 이때, Value 그 자체를 (Copy가 불가피할) Swap하지 않고 Value Pointer라는 것을 따로 두어 Value Array는 변화시키지 않은 채 Pointer만 Swap해가며 Sort를 진행한다. Bubble Sort를 차용하는 대신 String Copy Overhead를 최소화하기 위한 목적이다.

4) Make a room for new SSTable, by pulling all the left (non-victim) SSTables

-> 새롭게 생성될 SSTable은 어떤 File Number를 부여 받아야 하는가? 만약 Victim SSTable이 3.sst, 4.sst이고 현재 System엔 7.sst까지 SSTable이 생성되어 있으면(사실 Background MERGE 때문에 이런 상황은 발생하지 않음), 5.sst, 6.sst, 7.sst를 각각 한 칸씩 뒤로 밀어 6.sst, 7.sst, 8.sst로 만들어주어야 한다. 그 다음에 Newly created SSTable에 5.sst라는 파일명을 부여해야 한다. 4)에선 바로 이러한 'Pulling' 작업을 수행한다.

5) Create new SSTable

-> 간단하다. 정렬된 In-Memory Pairs Array를 그대로 Persist하면 된다.

MERGE는 이러한 순서로 일어난다. Main Process에서 MERGE Mode로 수행하든, Merge Process에서 Background MERGE를 수행하든 마찬가지이다. 단지 두 Case의 유일한 차이는 System 내 Un-Merged SSTable의 개수가 2개일 때도 Trigger할 수 있느냐 없느냐일 뿐이다.

이렇게 본인은 과제1의 Key-Value Store를 완성하였다. 더 자세한 Code-Level Detail은 제출 Source Code의 주석으로 확인할 수 있다. 주석을 많이 달아놓았다.

IV. 연구 결과

지금까지 설명한 Key-Value Store는 정상적으로 동작하며 PDF 명세서의 요구 조건을 모두 충족하고 있다. FND, LED, TEXT LCD, MOTOR 등이 모두 요구사항대로 적시에 정확히 동작하고 있으며, Key 입력도 예외없이 모두 처리되고 있다. 또한, 본인은 현재 제출 Source Code에서 일부 File에 Logging Flag를 활성화(C Macro로 코드 최상단에 정의, 주석 처리 시 Disable)해 Minicom Prompt에 Key-Value Store 내부 동작 상황을 자세하게 Logging하고 있다(과제 평가 시 조교님께서 이를 참조하시면 조금 더 편리한 평가를 하실 수 있습니다). 자세한 결과물은 직접 FPGA 보드 동작 및 Minicom을 통해 확인할 수 있다.

결론적으로 과제 명세서에서 요구하는 Key-Value Store 구현을 완성하였다. 모두 잘 동작하고 있다. 현재 Key-Value Store가 지원하는, 만족시키는 주요 평가 포인트는 다음과 같이 정리할 수 있다.

- Merge 시 Merge할 Storage Table을 선택하여 그 중 중복된 Key를 갖는 Value를 제거하고 Key를 정렬하여 새로운 Storage Table 생성 ([충족](#))

~> Enable된 Logging Function으로 Application 자체적으로 실시간 검증 가능

- 모든 Device Input & Output, 구체적으로 말하면 Device에 대한 (직접적) Read/Write는 오로지 I/O Process에서만 수행 ([충족](#))

- Merge Process는 I/O Process로부터 IPC를 통해 전달받은 Merge 요청을 처리한다. 또는 Merge Mode에서 Merge를 수동으로 수행한다. ([충족](#))

~> I/O Process의 Output Routine 수행 직전에 Merge Process 동작을 Synchronize시켜 매번 Background MERGE를 확인하게 만들었고, 그 MERGE 결과를 Merge Process에서 I/O Process로 Shared Memory를 통해 전달함. 한편, Main Process의 Merge Mode 역시 정상적으로 동작함.

- LED Device는 mmap() 함수를 사용하고 나머지 Device들은 Device Driver를 사용하여 프로그램을 구현한다. ([충족](#))

- 프로그램 시작 시 기본 모드는 모드 1, PUT Mode이다. ([충족](#))

- 프로그램 종료 시 모든 Device는 각자의 초기 상태로 세팅 ([충족](#))

- PUT, GET, MERGE의 TEXT LCD Device 출력 양식 ex. (1, 1000, A) ([충족](#))

- PUT 시의 LED 처리 (Key 입력 시엔 3, 4번이 번갈아 가며 On, Value 입력 시엔 7, 8번이 번갈아 가며 On, PUT 완료 시 전체 LED On 후 다시 1번 LED에만 불이 들어오도록 조정) ([충족](#))

~> GET, MERGE에 대해서도 모든 LED 조건을 충족하고 있음.

- SWITCH Alphabet 영문 입력 처리 ([충족](#))

- PUT, GET 시 SWITCH 2&3, 5&6, 8&9와 같은 동시 입력 처리 (충족)
- Background MERGE 발생 시에도 MOTOR 회전 (충족)
- 프로그램 재실행 시 KV Store의 Storage Table의 Database는 유지 (충족)

=> 이밖에도 명세서에 명시된 모든 조건 및 요구사항을 모두 충족

(참고: 본 보고서에 캡처 형식으로 첨부된 코드의 *Key-Value Interface* 관련 부분에서 보고서 작성 이후 약간의 수정이 있었습니다. PUT, GET, MERGE 시 Output을 몇 초간 유지하는 과정 처리를 약간 변경하였는데, 해당 부분 관련 몇 줄을 제외하곤 동일하기 때문에 캡쳐 이미지는 그대로 유지하였습니다)

V. 기타

위와 같은 과정을 통해 본 과제1을 성공적으로 수행할 수 있었다. FPGA 보드의 각 I/O Device를 제어하는 부분이 상당히 인상적이었으며, 보드 내부 DRAM에 Application을 설치하여 동작시킴으로써 Embedded System을 구축할 수 있다는 점 역시 마찬가지로 인상적이다. 추후 과제 수행자 본인의 진로와도 깊은 관계가 있는 부분이라 느껴지며, 본 과제에서 학습한 여러 Skill을 잘 기억해야 할 것이다.

한편, 본 과제에 대한 특별한 코멘트는 없으며, 향후 출제될 과제들도 기다려진다. 남은 학기 계속해서 실습과 과제에 열심히 임해 Embedded System 설계 능력을 더 키울 수 있도록 노력할 것이다.