Embedded System Software 과제 1

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박성용

학번 및 이름: 20161566, 권형준

개발기간: 2021.03.29~2021.04.15

목차

I. 개발 목표

Device Control과 IPC를 이용하여 주어진 Clock, Counter, Text editor, Draw Board를 구현한다. 조건으로는 Input을 받는 process, 정보를 처리하는 process, output을 처리하는 process 총 3개의 process로 구성이 되어야 하며, 이 사이의 IPC는 shared memory로 이루어져야 한다. Input device와 Output device는 각각 2개, 4개로 구성이 되어 있다.

II. 개발 범위 및 내용

가) Input process

'Switch'와 'Key' 두개의 device를 통해 input을 받는다. Switch는 총 1~9까지 있으며, Key는 (실제로는 4개)3개, Back. Vol+, Vol-가 있다. Input process에서는 Main process와 공유하는 shared memory영역에 어떠한 input을 받았는지 넘겨주는 역할을 한다. Input device 하나는 blocking이며 다른 하나는 non-blocking이기 때문에 2개의 thread를 생성하여 각각의 input을 독립적으로 main process로 넘겨주는 형식으로 구현한다. 또한 각 thread의 생성을 위해 또한 main과의 synchronization을 맞추기 위해 sleep을 넣어 오류를 방지한다.

나) Main process

Main process는 Input process와 공유하는 shared memory에서 메시지를 읽어와서 정보를 처리한 이후 output process로 넘겨준다. Output process에서는 단순히 출력만 하기 위하여 모든 데이터 연산은 main process에서 진행이 된다. 각각의 mode별로 각각 output device가 어떠한 출력을 가져야 되는지 결정하고 이를 실시간으로 shared memory에 upload한다.

다) Output process

Main process에서 넘겨준 정보를 가지고 device에 output을 출력한다. Shared memory에서 정보를 읽는 방식을 mode를 기준으로 하지 않고 Device를 기준으로 구현한다. 즉, output process는 device마다 하나의 thread를 생성하여 모든 device는 끊임없이 output shared memory영역에서 output할 내용을 가져오게 구현한다.

III. 추진 일정 및 방법

가) 추진 일정

3/30~4/5	IPC를 위한 shared memory 구현
4/5~4/6	IPC를 통한 shared memory의 synch(semaphore) 구현
4/6~4/12	Input, Output process에서 shared memory에서 읽은 내용 처리 구현
4/12~4/15	보고서 작성

나) 개발 방법

1) Fork를 사용하여 output, input process를 구현한다.

2) Shared memory를 구현하여 input-main, main-output 사이의 IPC를 가능하게 구현한다.

3) 프로그램이 종료될 때까지 반복해서 IPC사이의 정보가 전달됨으로, 이 사이의 shared memory공간에 대한 synchronization을 semaphore를 사용하여 적용한다.

4) Input process와 Output process를 device별로 thread를 할당하여 구현한다.

5) 각각의 mode에 따른 device의 상태를 구현한다.

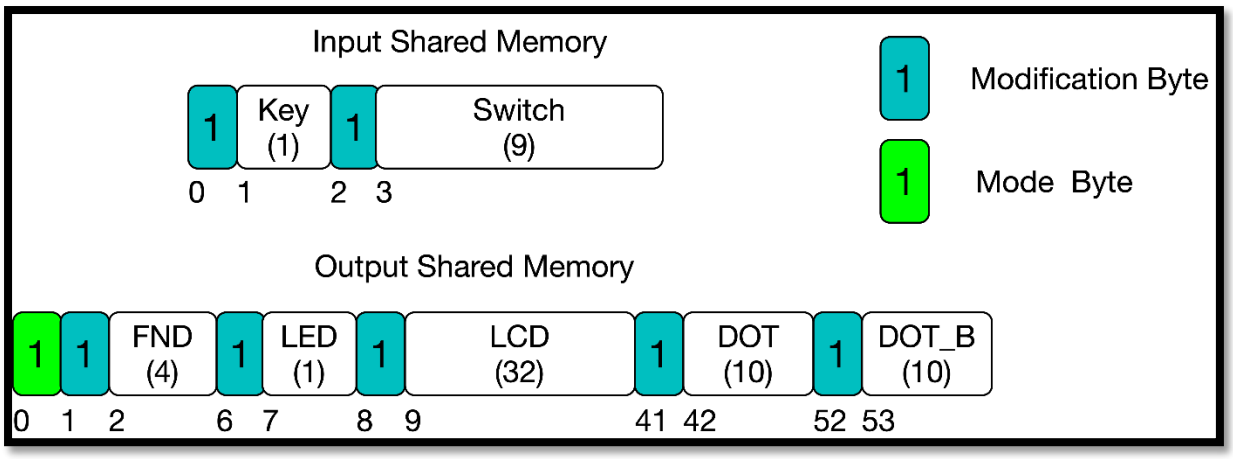

Figure 1. Input & Output Shared Memory

IV. 연구 결과

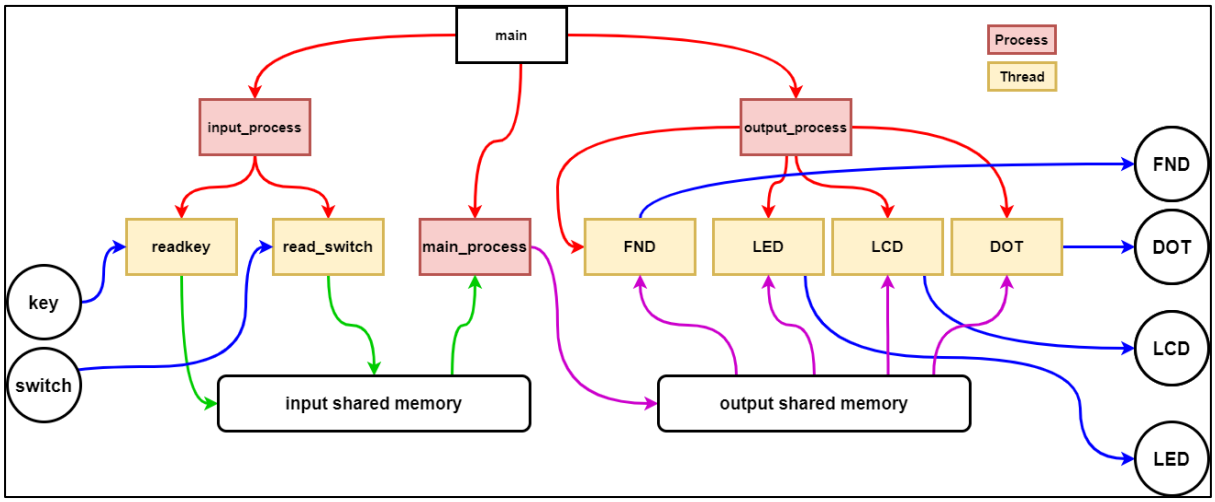

Figure 2. 전체적 Flow

가) 흐름도

<그림 2>를 보면, 우선 main함수에서 fork를 2번 호출하여 input과 output process를 호출한다. 각각의 input과 output process는 key, switch, FND, DOT, LCD, LED라는 thread를 만들고 device와 연결을 한다. Input process에서 만든 thread들은 반복문을 돌면서 key와 switch에서 데이터를 읽어서 input shared memory에 적는 역할을 하고, output process에서 만든 thread들은 반복문을 돌면서 output shared memory에서 데이터를 읽어와서 Device에 출력하는 역할을 한다. Main은 input shared memory와 output share memory사이에 데이터를 가공, 이동 역할을 해준다.

나) Shared Memory

이번 프로젝트에서 가장 중요한 IPC부분이다. Shared Memory를 통해 IPC를 하기 위해 우선 <main.c>에서 shared memory영역을 선언하고, 이 shared memory영역에 접근하기 위해 semaphore도 initialize한다.

```
//make shared memory for IPC
shmid_input = shmget(IPC_PRIVATE,10,IPC_CREAT|0666); // parent-child IPC-> IPC_PRIVATE
if(shmid_input == -1){
        perror("shmget");
        exit(1);
}
shmid_output = shmget(IPC_PRIVATE,10,IPC_CREAT|0666);
if(shmid_output == -1){
        perror("shmget");
        exit(1);
}
//make semaphore for shared memory
sem_init(&m_input,1,1);
sem_init(&m_output,1,1);
```

다음 각각 input, output process에서 Shared memory에 접근하기 위한 과정을 거친다. 여기서 인자로 넘겨준 shmid_input이 shared memory영역을 identify한다.

```
//shared memory
shmaddr = (char*)shmat((shmid_input),(char*)NULL,0);
```

```
//connect with IPC
output_shmaddr= (char*)shmat(shmid_output,(char*)NULL,0);
```

이제 각각의 shared memory공간이 생겼는데, 양쪽에서 어떻게 shared memory를 사용할지에 대한 약속(protocol)이 없으면 제대로 된 정보 교환이 이루어질 수 없다. 이를 위해서 Input과 Output shared memory에 대한 형식을 지정할 필요가 있는데, <figure 1>와 같이 형식을 설정하였다. 단위는 byte로 하였으며, Modification byte란 바로 뒤에 오는 영역에 수정된 것이 있는지, 즉, 입력한 내용이 있는지 또는 출력할 내용이 있는지를 나타내는 byte이다. 이를 bit단위로 하였으면 조금 더 효율적이었을 수 있지만, 주고받는 내용도 적고 또 input, output device에서 인자로 사용하는 것들이 byte단위로 되어 있어서 byte를 그대로 사용하였다. 예를 들어 Output shared memory의 index 6에 '1'이 들어와 있다면, 이는 LED에 출력할 값이 있으며, 그 값은 index 7에 저장되어 있다는 뜻이다. 추가적으로 Mode byte는 Output process에서 읽어서 현재 mode를 알 수 있게 해주며, DOT_B는 Dot Blink의 약자로 mode 4에서 Dot matrix가 blink하기 위해 현재 cursor가 있는 위치만 실제 Dot matrix와 반전된 결과를 저장한다. 이는 Mode 4일 때 1초 간격으로 읽어서 화면에 DOT와 돌아가면서 출력을 한다.

이러한 구조로 인해 각각의 thread(device에 할당된)들은 정해진 영역만을 읽게 되고 독립적으로 돌아가게 된다. Main process만이 중간에서 다리 역할을 하게 된다.

다) FND

FND에서 읽어오는 영역은 output shared memory index**<1~5>**이다. while문을 돌면서 계속 이영역에 (semaphore를 걸고)접근을 하여 index[1]이 '1'이라면 index[2]~index[5]까지의 값을 읽어와서 그대로 FND로 출력한다. FND는 출력만을 담당하는 함수(thread)이다. 모든 결과는 main process에서 계산을 해서 내려온다. Clock mode에서는 switch에 따른 시간을, Counter에서는 증가된, 변형된 진수에 따른 수를, Text editor, Draw board에서는 몇 번 switch를 눌렀는지에 대한 정보 등등을 4개의 character로 넘겨주고 이를 그대로 system call을 통하여 device에 출력한다.

```
while(!Output_Terminated){
        memset(fnd_data,0,sizeof(fnd_data));
        output_mode=output_shmaddr[0];
        if(output_shmaddr[1]==1){
                for(i=0;i<FND_MAX_DIGIT;i++){
                    fnd_data[i]=output_shmaddr[i+2];
                }
                output_shmaddr[1]=0;
                retval=write(dev,&fnd_data,FND_MAX_DIGIT);
                if(retval<0){
                        printf("FND error\n");
                        exit(1);
                }
        }
        sem_post(&m_output);
```

라) LED

FND와 크게 다르지 않다. 똑같이 LED에 주어진 영역의 값 **index<6~7>**을 읽어와서 출력한다. 다만 LED는 mode에 따라 수행해야 될 일이 다를 수 있다. 즉, main에서 그대로 출력할 내용을 보내주는 것이 아닌, 특정 mode를 보내주면 간단한 조건문을 통해서 출력할 내용을 LED로 보낸다. 이는 LED에서 출력되는 값의 내용이 매우 한정적이어서 이렇게 구현하였다.

```
void LED_MODE1(int led_mode, unsigned char*led_addr){
        unsigned char data;
        if(led_mode==1){
                data=128;
                *led_addr=data;
        }
        else if(led_mode==2){
                data=32;
                *led_addr=data;
                usleep(1000000);
                if(output_mode==0){
                    data=16;
                    *led_addr=data;
                    usleep(1000000);
                }
        }
}
void LED_MODE2(int led_mode, unsigned char* led_addr){
        unsigned char data;
        if(led_mode==1){
                data=64;
        }
        else if(led_mode==2){
                data=32;
        }
        else if(led_mode==3){
                data=16;
        }
        else{
                data=128;
        }
        *led_addr=data;
}
```

위의 코드를 부면 "led_mode"가 output shared memory에서 읽어온 값으로 경우가 1~4이기 때문에 간단하게 구현할 수 있다. Mode1의 경우 LED3, LED4를 번갈아 가면서 출력하기 위해 usleep을 추가하였다. 이때, 중간에 mode가 바뀌어도 1초 더 쉬는 것을 방지하기 위해 usleep사이에 if문을 추가하였다.

마) LCD

LCD는 FND와 마찬가지로 output shared memory에 있는 내용을 그대로 출력한다. **Index<8~40>**까지를 사용하며, 처음 16byte는 위에 줄을, 다음 16byte는 밑의 줄을 출력한다. 한가지 주의할 것은 기본적으로 0으로 초기화 되어 있는데, 이를 그대로 출력하면 쓰레기 값이 찍히게 되어, 0을 빈칸(공백)으로 출력하는 예외 처리가 필요하다.

```
while(1){
        output_mode=output_shmaddr[0];
        if(output_mode!=2){
                memset(string,0,sizeof(string));
                LCD_DEFAULT(lcd_dev);
                continue;
        }
        sem_wait(&m_output);
        if(output_shmaddr[8]==1){
                for(i=0;i<LCD_BUFF_SIZE;i++)
                        string[i]=output_shmaddr[i+9];
                output_shmaddr[8]=0;
                for(i=0;i<LCD_BUFF_SIZE;i++)
                        if(string[i]==0)
                                string[i]=' ';
                write(lcd_dev,string,LCD_BUFF_SIZE);
        }
        sem_post(&m_output);
```

바) DOT

DOT은 Mode3와 Mode4에서만 사용이 된다. 사용하는 범위는 **index<41~51>**이다. Mode3같은 경우에는 단순히 'A'또는 '1'어느 것인지 알면 되기 때문에 간단하게 main에서 모드를 설정하여 넘겨 받은 것을 출력하였다. DOT도 FND와 같이 Main에서 모두 처리하여 데이터를 넘겨줄 수 있기 때문에 입력 받은 10byte를 그대로 출력하는 system call로 넘겨주었다. 까다로웠던 것은 Mode4이다. 계속 깜빡이는 Cursor에 대한 구현을 위해 새로운 thread를 만들어야 되나 생각을 했지만, 하나의 device를 사용해야 되는 점은 변하지 않고 또한 한 좌표 단위로 출력이 안되는 Dot Device 때문에 포기하였다. 결국 1초 간격으로 출력할 2개의 판이 필요했고 따라서 DOT_B를 shared memory에 추가하였다. Mode4인경우 추가적으로 **index<52~62>**까지 읽어서 저장하였다.

```
void DOT_MODE4(int dot_dev, unsigned char mode4_output[10], unsigned char mode4_output_cursor[10]
){
        write(dot_dev,mode4_output,MAX_FPGA_SIZE);
        usleep(1000000);
        if(output_mode==3){
            write(dot_dev,mode4_output_cursor,MAX_FPGA_SIZE);
            usleep(1000000);
        }
}
```

그리고 그렇게 저장한 mode4_output_cursor와 mode4_output을 1초 간격으로 출력하였는데, 2초의 시간은 크기 때문에 중간에 mode가 변경될 수 있는 점을 고려하여 중간에 mode를 확인하는 것을 넣었다. 이는 Mode change시에 DOT에서 발생하는 약간의 딜레이를 막아주었다.

사) Input Process

Input process같은 경우에는 mode에 대한 정보, 어떤 데이터를 받았는지에 대한 정보, 현재 켜져 있는 device에 대한 정보 모두 필요가 없다. 오로지 key, switch device로부터 읽어온 값이 무엇인지만 처리하여 input shared memory로 넘겨준다. Key device같은 경우에는 blocking read(?)를 한다. 즉, key를 누르기 전까지는 key를 입력받는 코드, 함수를 넘어가지 않는다. 하지만 switch같은 경우에는 non-blocking으로 계속 입력을 기다린다. 따라서 잠깐 누르는 경우 기계는 그 짧은 순간을 길게 받아들여 여러 번의 입력을 전달할 수 있다. 이를 방지하기 위해 2가지 방법을 사용하였다. 첫번째는 usleep을 사용하여 입력을 받는 interval을 두었다. 아무리 빨리 누르더라도 0.2초보다는 오래 걸릴 것이라고 생각하여 usleep(200000)을 하여 0.2초마다 입력을 받게 하였고, 또한 단순히 '0'과 '1'로 구분하지 않고 어느정도 interval을 두어 '0'→ '1', rising edge를 감별하여 버튼의 누름 여부를 판단하였다. 이를 통해 애매하게 누른 '0'→'1'→'1'→'0'→'1'과 같은 sequence를 하나의 입력으로 처리할 수 있었다. 이렇게 구현했음에도 불구하고 usleep과 interval이 잘못 겹쳐서 한번 눌렀을 때 2번씩 입력되는 경우가 종종 발생하였지만, 대부분의 경우 빠르고 정확하게 누를 경우 오류는 발생하지 않았다.

아) Mode 1: Clock

Clock에서 switch를 통해 입력 받는 내용은 (key 제외) '시간'을 증가해라, '분'을 증가해라, 모드를 바꿔라, 보드의 시간을 불러와라. 4가지이다. 프로그램 내부에 '저장된 시간'과 '임시 시간'을 두어 시간 변경 모드에서 임의로 변경한 시간을 잠시 저장해 두었고 이후에 다시 시간 변경 switch (SW1)을 누르면 '임시 시간'이 '저장된 시간'에 저장되게 하였다. 또한 보드의 시간을 불러오는 함수를 설정하여 처음에 호출하고 이후에 SW2가 눌러질 때마다 호출하였다. Input과 output과정은 앞에서 설명한 흐름으로 진행이 되었고, FND에는 "시간/분"을 LED에는 mode(시간 변경 중/시간 변경 중 아님)을 넘겨주었다.

자) Mode 2: Counter

Counter mode에서는 정수형 변수를 하나 설정하여 switch로부터 받는 입력을 통해 해당 정수를 변형하였다. 또한 mode(2/4/8/10진수)를 설정하여 LED에 넘겨주어 진수에 맞는 LED를 출력하게 하였고, 해당 mode에 따라 FND의 인자로 넘어갈 값을 알맞게 변형하여 넘겨주었다.

차) Mode 3: Text editor

Counter와 비슷하게 LCD에 출력할 string을 char[32]의 형태로 저장하였고, switch로부터 받은 입력을 통해 이 string을 update하였다. 같은 버튼이 연속적으로 들어오면 string의 마지막 char을 변경해주어야 되기 때문에 string의 마지막 인자를 지정하고 있는 lcd_last_point와 같은 변수도 사용하였고 내장된 mode를 DOT device에도 보내주었다. Mode를 보내주었다고 하기보다는 mode에 맞는 predefined된 char array를 보내주어 그대로 출력하게 하였다.

카) Mode 4: Draw board

Mode 3의 연장선으로 현재 DOT의 상태를 알아야 하고 또한 cursor의 위치도 알아야 한다. 하나의 작은 Dot light는 1bit를 통해 '켜짐'과 '꺼짐'을 구분하기 때문에 bit 단위의 연산을 사용하였다. 하지만 cursor의 경우 0~10이 아닌 실제 dot device의 index처럼 0~70으로 계산한 다음, 마지막에 출력에 추가해주거나 dot을 추가할 때 bit단위로 변형하였다.

Cursor의 깜빡임을 구현하기 위해 현재 board에서 cursor가 있는 위치를 반전시킨 임의의 다른 boar를 생성하고 output process로 같이 넘겨주었다. Output process는 mode를 구분하여 만약 mode4라면 두개의 board를 1초 간격으로 출력하여 마치 하나의 점이 깜빡이는 것처럼 보이게 구현하였다. 반전의 경우 모든 값을 NOT연산한 뒤 overflow를 방지하기 위해 0x7f와 and연산을 해주었다.

타) Inter Mode

Mode와 Mode사이를 넘어갈 때, 각각 mode의 초기 상태로 보드가 변형되어야 한다. 또한 프로그램이 종료될 때 모든 device는 꺼져야 된다. 이는 각각의 output device에 대한 thread를 설정한 환경에서 구현하기 간단하였다. Mode가 바뀌는 key(vol+, vol-)가 눌리게 되면, 프로그램(main)내에 저장된 mode변수의 값을 변경해 줌과 동시에 output shared memory의 첫 byte를 mode값으로 설정한다. 이렇게 한 이유는, output process내에 따로 변수를 두게 될 경우 output process내의 4개의 thread중 하나만이 update하게 하는 경우 output process내부에서 inconsistency가 생길 수 있기 때문이다. 즉, output shared memory에 접근을 함과 동시에 항상 output process의 thread들은 mode에 alter가 있었는지 확인함으로써 더욱더 정확도를 높인 것이다.

Initial_mode라는 함수를 통해 mode가 바뀌게 되면 일단 모든 device를 초기상태로 reset시킨다. (FND=0000, LED=꺼짐, LCD=꺼짐, DOT=꺼짐). 다음 mode에 따른 초기 설정을 다시 해주는 형식으로 구현하였다. 이렇게 abstraction(?)을 통해서 프로그램이 시작할 때, 각각의 모드가 변할 때, 프로그램이 끝날 때 몇번의 함수 호출만으로 초기화가 가능하였고 수정도 간편하게 할 수 있었다. 아래의 간단한 예시를 보면, 우선 output shared memory를 0으로 초기화한 뒤 필요한 mode에 대한 initializer함수를 부르고 마지막에 각각의 device에 modification이 발생하였다는 것을 알려줌으로써 초기화+초기 setting이 진행된다. 프로그램 종료 시 case 4를 실행하여 아무런 초기 setting 없이 초기화만 이루어지어 모든 device가 꺼지게 된다.

```
int initial_mode(){
        int i=0;
        initial=false;
        sem_wait(&m_output);
        for(i=1;i<62;i++)
                shmaddr_output[i]=0;

        switch(mode){
                case 0:
                        initial_mode1();
                        break;
                case 1:
                        initial_mode2();
                        break;
                case 2:
                        initial_mode3();
                        break;
                case 3:
                        initial_mode4();
                        break;
                case 4:
                        break;
        }
        shmaddr_output[0]=mode;
        shmaddr_output[1]=1;
        shmaddr_output[6]=1;
        shmaddr_output[8]=1;
        shmaddr_output[41]=1;

        sem_post(&m_output);
}
```

```
void initial_mode4(){
        int i;
        memset(mode4_fpga_number,0,sizeof(mode4_fpga_number));
        cursor=0;
        mode4_mode=1;
        mode4_count=0;

        shmaddr_output[52]=0x40;
}
```

V. 기타

가) 느낀 점

pintos에서 여러 개의 process를 생성하는 kernel에 대한 코드는 만들어봤지만, 실제로 fork와 pthread를 사용하여 코딩을 한 것이 처음이었다. 기존에는 하나의 flow만을 고려하면 되었기 때문에 debugging도 비교적 쉬웠고 논리적이었지만, thread programming의 경우 thread들 사이의 synchronization, scheduling등으로 인해 이 과정이 쉽지 않았다. 특히 if else문으로 거른 오류, 예외 사항을 통과하여 진행된 경우가 있었는데, 이는 한쪽 thread에서 if문을 통과함과 동시에 다른 thread에서 값이 변경되어 else문에도 걸리지 않는 상황이었다. Fork를 처음 한만큼 IPC도 처음 사용해보았는데 단순히 파일에 쓰고 읽는 것 같은 과정이어서 단순하게 생각했지만, semaphore설정 등 과정에서 어려움을 겪었다.

나) 구현의 한계점

정상적으로 동작은 하지만, semaphore의 동작에 대한 의구심이 든다. Output, input shared memory의 modification byte를 통해 사실상 semaphore는 없어도 synchronization이 된다. 즉, 쓰는 쪽에서 우선 내용을 다 쓴 이후 modification byte를 쓰게 되며, modification byte가 이미 쓰여 있는 경우(reader가 읽지 않은 경우)에는 쓰지 않는다. 읽는 쪽에서는 modification byte가 1인 경우에 먼저 내용을 다 읽고 그 다음 modification byte를 0으로 바꾼다. 1byte를 쓰고 읽는 것은 atomic하기 때문에 이러한 과정이 자체적으로 lock이 되어 진행이 된다.

따라서 semaphore가 추가적으로 필요 없지 않을까 생각이 들었으며, modification byte을 사용하기 이전의 semaphore만 사용한 구현에서 약간의 딜레이, 오류가 발생하였던 것으로 보아 더 확신이 생겼다. <semaphore.h>의 library를 사용할 수 없지 않을까 생각이 들었다.

결론: 자체적 구현 때문에 semaphore가 추가적으로 필요 없지 않을까?

다) 문제점

없음.

감사합니다.