

JAEHO's PORTFOLIO

- OS implementation

프로젝트 주제 선정이유

평소 희망하던 직무가 OS개발, 디바이스 드라이버, 시스템 소프트웨어 분야였습니다. 사용자 입장에서 안보이지만 서비스 제공에 필수적인 역할을 하는 시스템 프로그래밍에 관심이 있었습니다. MDS아카데미 임베디드 S/W 개발자 양성과정 종료후 OS기능을 구현해 본다면 임베디드 S/W개발 역량을 키울수 있고, 제 관심분야에 대해 좀 더 다가갈 수 있다 생각했습니다. 그래서 OS기능 구현을 시작했습니다. 단 부팅과정까지 직접 한다면 목표로 잡은 2개월내 완성할수 없다 판단했고, 해당 보드업체의 부트 코드를 사용했습니다. 주로 kldp사이트와 임베디드OS만들기 라는 책을 참고했습니다.

개발환경

호스트 운영체제 : Ubuntu 12.04.5 LTS 32bit

보드 : gumstix

qemu version 1.0.50 (Debian 1.0.50-2012.03-0ubuntu2.1) – 에뮬레이터로 가상환경구축

칩 : pxa255

크로스컴파일러 : arm-linux-gcc 3.3.2

목차

1. Summary	pg5
1) 파일 구성	
2. 개발환경구축	pg6
1) 크로스 컴파일러 설치	
2) qemu설치	
3) qemu로 돌릴 보드환경 구축	
4) 해결방법	
5) 파일구성 – helloworld	
6) ulmage 변환	
7) 램디스크 이미지 만들기	
8) 플래시 이미지 만들고 부팅하기	
3. Exception Handling	pg12
1) Exception발생시 작업 순서	
2) 코드수정	
3) u-boot수정	
4. Software Interrupt Handler	pg14
1) 컨텍스트 스위칭 - ISR과 태스크	
2) 스택을 이용한 컨텍스트 스위칭 구현	

3) 시스템 콜 번호 추출

4) 메인함수 수정

5. IRQ Handler – using OS timer

pg17

1) 인터럽트 과정

2) OS타이머 관련 함수 구현

3) 컨텍스트 스위칭 구현

4) 파이프라인과 pc, lr

6. Memory map

pg20

1) 스택 주소 초기화

7. Memory Manager

pg21

1) 메모리 관리자 정의

2) 메모리 관리자 구현

8. Process Manager

pg23

1) 프로세스 관리자 정의

2) 프로세스 관리자 구현

3) 사용자 프로세스 추가

4) 메인함수 구현

9. Context Switching

pg28

1) 컨텍스트 스위칭 구현

2) 스케줄러 구현

3) 그 외 함수 수정

10. System Call

pg31

1) 시스템 콜 계층 추가

2) 시스템 콜 초기화 함수 호출

3) 시스템 콜 관련 헤더 파일 작성

4) 사용자 프로세스 함수 수정

5) 시스템 콜 래퍼 함수 구현

6) 소프트웨어 인터럽트 ISR수정

7) 정리 – 시스템 콜 추가 절차

11. Communication with Process

pg35

1) 컨텍스트 스위칭 시스템 콜 구현

2) 메시지 관리자 정의 – 메시지큐

12. Synchronization

pg41

- 1) 세마포어 정의
- 2) 세마포어 구현

13. Device Driver

pg49

- 1) 디바이스 드라이버 관리자 정의
- 2) 디바이스 드라이버 관리자 구현
- 3) 시스템 콜에 등록
- 4) 디바이스 드라이버 추가

1. Summary

1) 파일 구성

부트코드를 제외한 소스코드

testOS_lib.S testOS_user.c entry.S testOS.c testOS_msg.c
testOS_clib.c testOS_sys.c testOS_drv.c testOS_process.c testOS_memory.c

헤더파일

testOS_lib.h testOS_process.h testOS_memory.h testOS_user.h testOS.h testOS_msg.h
testOS_drv.h testOS_sys.h syscalltbl.h

entry.S, testOS.c, testOS.h

testOS.c에는 커널 메인함수가 있습니다. 커널 초기화 및 소프트웨어 인터럽트, IRQ핸들러가 구현되었습니다. entry.S 에는 커널구동시 필요한 여러 어셈블리가 구현되었습니다.(ex)컨텍스트스위칭)

testOS_memory.c, testOS_memory.h

메모리 관리자가 구현된 파일입니다.

testOS_process.c, testOS_process.h

프로세스 관리자가 구현된 파일입니다. 사용자 프로세스를 커널에 등록하는 과정과 Process Control Block이 구현되어 있습니다.

testOS_user.c, testOS_user.h

사용자 프로세스 함수가 구현되어있는 파일입니다.

testOS_sys.c, testOS_sys.h, syscalltbl.h, testOS_lib.S, testOS_clib.c, testOS_lib.h

시스템 콜 구현이 되어 있는 파일 목록입니다. testOS_sys.c와 testOS_sys.h에는 시스템 콜과 관련된 자료구조가 정의되어 있습니다. Syscalltbl.h는 시스템 콜 번호를 등록하는 헤더파일입니다. 나머지는 래퍼 함수와 관련되어 있어 사용자 레벨에서 시스템 콜을 사용하기위해 중계하는 함수 입니다.

testOS_msg.c, testOS_msg.h

메시지 관리자가 구현된 파일입니다. IPC, 세마포어 모두 동일한 메시지 자료구조를 사용합니다.

testOS_drv.c, testOS_drv.h

디바이스 드라이버 계층을 구현한 파일입니다.

2. 개발환경구축

1) 크로스 컴파일러 설치

```
$ tar xvf arm-linux-gcc-3.3.2.tar.bz2
$ vi .bashrc
PATH=$PATH: /home/jaeho/usr/local/arm/3.3.2/bin    추가
$ source .bashrc
jaeho@ubuntu:~$ echo $PATH    //수정내용이 변경됐는지 확인
```

2) qemu설치

```
jaeho@ubuntu:~$ sudo apt-get install qemu
jaeho@ubuntu:~$ sudo apt-get install qemu-system
```

3) qemu로 돌릴 보드환경 구축

Gumstix는 u-boot를 사용하므로 u-boot를 설치한다.

```
$tar xvf gumstix_uboot.tgz
폴더로 이동
$ make distclean
$ make gumstix_config
$ make all    // 최종적으로 u-boot.bin파일 생성
이 파일을 qemu를 통해 실행
$ qemu-system-arm -M connex -pflash u-boot.bin -nographic
```

◆ 에러발생 - qemu: Error registering flash memory.

4)해결방법

```
$ dd if=/dev/zero of=flash.bin bs=1k count=16k    // 1kbytes씩 16K번 /dev/zero에서 읽어와서
                                                    flash.bin에 기록
```

```
$ dd if=u-boot.bin of=flash.bin bs=1k conv=notrunc
```

(conv=notrunc 는 출력 파일이 잘리지(truncated) 않을 것이라는 옵션입니다)

```
$ qemu-system-arm -M connex -pflash flash.bin -nographic
```

인터넷 자료들을 보면 그냥 실행되는 경우도 있는 것 같습니다. 새로 만든 flash.bin파일은 정확히 16MB입니다. 아마 qemu에서 메모리 설정을 위해 파일이 16MB를 만족해야하는 것 같습니다.

```

jaehc@ubuntu:~/gunstix_uboot$ qemu-system-arm -M connex -pflash flash.bin -nographic
pxa2xx_clkpwr_write: CPU frequency change attempt

U-Boot 1.1.4 (Sep 23 2016 - 04:55:39) - 200 MHz -

*** Welcome to Gunstix ***

U-Boot code: A3F00000 -> A3F26C70  BSS: -> A3F5BD8C
RAM Configuration:
Bank #0: a0000000 64 MB
Flash: 16 MB
Using default environment

SMC91C1111-0
Net: SMC91C1111-0
Hit any key to stop autoboot: 0
Instruction Cache is ON
## JFFS2 loading 'boot/uImage' to 0xa2000000
Scanning JFFS2 FS: done,
find_inode failed for name=boot
load: Failed to find inode
## JFFS2 LOAD ERROR<0> for boot/uImage!
GUM> █

```

5) 파일구성 – helloworld

보드사에서 제공한 소스를 이용한다.

entry.S lib1funcs.S testOS.c(직접작성) serial.c time.c gpio.c main-ld-script
printf.c string.c vsprintf.c

include폴더

gpio.h pxa255.h serial.h stdarg.h stdio.h string.h time.h

testOS.c

```
#include <testOS.h>
```

```
int main(void)
```

```
{
```

```
    while(1){
```

```
        printf("hello world\n");
```

```
        msleep(1000);
```

```
    }
```

```
    return 0;
```

```
}
```

Makefile

CC = arm-linux-gcc

LD = arm-linux-ld

OC = arm-linux-objcopy

CFLAGS = -nostdinc -I. -I./include -I fs/include

CFLAGS += -Wall -Wstrict-prototypes -Wno-trigraphs -O2

CFLAGS += -fno-strict-aliasing -fno-common -pipe -mapcs-32

CFLAGS += -mcpu=xscale -mshort-load-bytes -msoft-float -fno-builtin

LDFLAGS = -static -nostdlib -nostartfiles -nodefaultlibs -p -X -T ./main-ld-script

OCFLAGS = -O binary -R .note -R .comment -S

all: testOS.c

\$(CC) -c \$(CFLAGS) -o entry.o entry.S

\$(CC) -c \$(CFLAGS) -o gpio.o gpio.c

\$(CC) -c \$(CFLAGS) -o time.o time.c

\$(CC) -c \$(CFLAGS) -o vsprintf.o vsprintf.c

\$(CC) -c \$(CFLAGS) -o printf.o printf.c

\$(CC) -c \$(CFLAGS) -o string.o string.c

\$(CC) -c \$(CFLAGS) -o serial.o serial.c

\$(CC) -c \$(CFLAGS) -o lib1funcs.o lib1funcs.S

\$(CC) -c \$(CFLAGS) -o testOS.o testOS.c

\$(LD) \$(LDFLAGS) -o testOS_elf entry.o gpio.o time.o vsprintf.o printf.o string.o serial.o lib1funcs.o
testOS.o

\$(OC) \$(OCFLAGS) testOS_elf testOS_img

\$(CC) -c \$(CFLAGS) -o serial.o serial.c -D IN_GUMSTIX

\$(LD) \$(LDFLAGS) -o testOS_gum_elf entry.o gpio.o time.o vsprintf.o printf.o string.o serial.o
lib1funcs.o testOS.o

\$(OC) \$(OCFLAGS) testOS_gum_elf testOS_gum_img

clean:

rm *.o

rm testOS_elf

rm testOS_img

rm testOS_gum_elf

rm testOS_gum_img

testOS_img는 보드용, testOS_gum_img는 에뮬레이터용 입니다.

pxa255칩을 사용한 보드의 회사(falinux)에서 제공하는 부트코드에서 시리얼포트를 통해 UART 메시지를 출력하는 부분을 수정해 사용합니다.

Falinux의 부트코드는 Standard UART를 사용했지만 u-boot는 Full Function UART를 사용했습니다. 그래서 Full Function UART의 기준 레지스터 주소를 설정해야 합니다.

```
serial.c //볼드체를 추가
```

```
#ifdef IN_GUMSTIX
```

```
static volatile Word *UART = (volatile Word *) FFUART;
```

```
#else
```

```
static volatile Word *UART = (volatile Word *) STUART;
```

```
#endif
```

컴파일시 gcc옵션으로 IN_GUMSTIX을 줄경우(gcc -D IN_GUMSTIX) FFUART가 세팅됩니다.

에뮬레이터에서는 시리얼 포트를 통해 부트로더에 이미지를 다운로드하는 것이 불가능합니다. 터미널 프로그램을 통해 출력하지 않고 에뮬레이터 자체에서 출력을 하기 때문입니다. 그래서 에뮬레이터로 보내는 이미지 파일을 생성할 때 커널이미지를 묶어야 합니다. 또한 u-boot는 ulmage포맷을 사용합니다. 그래서 gcc로 생성한 바이너리 이미지를 mkimage툴을 이용해 ulmage로 변환해야합니다. 다시말해 커널 이미지를 ulmage로 변환후, 나머지 유틸리티를 하나의 램 디스크 파일로 만들어 보드의 플래시 메모리에 올려야 합니다.

6) ulmage 변환

이미지 전용 디렉토리를 생성후 u-boot소스에 있는 mkimage툴을 사용해 ulmage로 변환합니다..

```
$ mkdir img
```

```
$ cp ../gumstix_uboot/tools/mkimage ./ //mkimage프로그램을 img폴더로 복사
```

u_boot.bin파일과 testOS_gum_img파일을 복사합니다.

```
$ cp testOS_gum_img ../img/
```

```
$ cp u-boot.bin ../img
```

testOS 커널 이미지(testOS_gum_img)를 ulmage로 변환합니다.

```
$ ./mkimage -A arm -O linux -T kernel -C none -a a0008000 -e a0008000 -n 'testOS'
```

```
-d testOS_gum_img ulmage
```

* -A : 아키텍처 지정

* -O : 운영체제 지정 (리눅스인것처럼 지정)

- * -T : 이미지 종류 지정
- * -C : 압축여부 지정
- * -a : 로딩될 메모리 주소
- * -e : 엔트리 포인트 지정
- * -n : 메시지 입력
- * -d : 변환할 대상 파일 지정, 한칸뒤에는 output파일의 이름

7) 램디스크 이미지 만들기

여태까지 커널 이미지를 만들었으니 램디스크를 만듭니다. jffs2파일시스템 이미지를 생성하는 유틸리티를 이용해 램디스크 이미지를 생성할 것입니다.

```
$ sudo apt-get install mtd-utils
$ mkdir kernel
$ mkdir kernel/boot
$ mv ulmage boot/
$ mkfs.jffs2 -e 0x20000 -d kernel -p -o testOS.jffs2
→testOS.jffs2파일 생성
```

8) 플래시 이미지 만들고 부팅하기

이제 u-boot.bin과 testOS.jffs2 를 하나의 파일(testOSimg)로 만듭니다.

```
$ dd if=u-boot.bin of=testOSimg bs=1k conv=notrunc
$ dd if=testOS.jffs2 of=testOSimg bs=1k conv=notrunc seek=180
```

편의를 위해 쉘 스크립트를 작성하겠습니다.

```
$ vi start.sh
#!/bin/sh
rm testOSimg -f
rm testOS.jffs2 -f
./mkimage -A arm -O linux -T kernel -C none -a a0008000 -e a0008000 -n 'testOS' -d testOS_gum_img
ulmage
mv ulmage kernel/boot/
mkfs.jffs2 -e 0x20000 -d kernel -p -o testOS.jffs2
dd if=u-boot.bin of=testOSimg bs=1k conv=notrunc
dd if=testOS.jffs2 of=testOSimg bs=1k conv=notrunc seek=180
#qemu error.. So I add this way
```

```
dd if=/dev/zero of=final_testOSimg bs=1k count=16k
```

```
dd if=testOSimg of=final_testOSimg bs=1k conv=notrunc
```

```
qemu-system-arm -M connex -pflash final_testOSimg -nographic
```

```
$ chmod +x start.sh
```

```
$ ./start.sh
```

```
SMC91C1111-0
Net: SMC91C1111-0
Hlt any key to stop autoboot: 0
Instruction Cache is ON
## JFFS2 loading 'boot/uImage' to 0xa2000000
Scanning JFFS2 FS: . done.
## JFFS2 load complete: 7268 bytes loaded to 0xa2000000
## Booting image at a2000000 ...
Image Name: testOS
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 7204 Bytes = 7 kB
Load Address: a0008000
Entry Point: a0008000
OK

Starting kernel ...

hello world
hello world
..
```

3. Exception Handling

1) Exception발생과 작업 순서

ARM프로세서는 예외처리를 위해 Exception vector table로 예외상황을 7가지로 구분합니다. Exception vector table이란 ARM프로세서에서 Exception이 발생했을 때 ARM프로세서가 실행을 분기할 주소를 모아 놓은 테이블 입니다. 또한 최소한의 구현을 할 것이기에 exception은 Software Interrupt와 IRQ를 구현할 것입니다. Software Interrupt는 SWI명령을 이용해 시스템콜, 컨텍스트 스위칭을 구현할 것이고, IRQ는 외부 인터럽트, 타이머 인터럽트 처리에 사용할 것 입니다.

Exception발생시 작업 순서

- 1) exception발생
- 2) exception 모드의 spsr에 cpsr을 저장
- 3) exception 모드의 lr에 pc값을 저장
- 4) cpsr의 모드 비트를 변경해 해당 exception에 대응하는 동작 모드로 진입
- 5) pc에 exception handler의 주소를 저장하여 해당 exception에서 해야할 작업을 수행

2) 코드 수정

entry.S파일을 수정합니다. 커널의 엔트리 포인트 위치는 bl main 명령입니다. 이 명령이 수행되면 C로 작성된 main함수로 점프하게 됩니다. 그리고 main이 종료됐음에도 커널이 종료되지 않게하기위해 재 분기를 사용했습니다. testOS_swiHandler는 SWI exception vector table로부터 진입할 위치 입니다.

entry.S

```
.globl _ram_entry
```

```
_ram_entry:
```

```
    bl    main                // 0xA0008000
    b     _ram_entry          // 0xA0008004
    b     testOS_swiHandler    // 0xA0008008
    b     testOS_irqHandler    // 0xA000800C
```

```
.global testOS_swiHandler
```

```
testOS_swiHandler:
```

```
    bl    swiHandler
```

```
.global testOS_irqHandler
```

```
testOS_irqHandler:
```

```
    bl    irqHandler
```

SWI와 IRQ를 간단히 테스트 해보기 위해 testOS.c 코드를 수정한다.

testOS.c

```
#include <testOS.h>

void swiHandler(unsigned int syscallnum)
{
    printf("system call %d\n", syscallnum);
}

void irqHandler(void)
{
}

int main(void)
{
    __asm__("swi 77");
    return 0;
}
```

3) u-boot 수정

gumstix_uboot/cpu/pxa/start.S 파일 수정

software_interrupt:

ldr pc, =0xa0008008

(생략)

irq:

ldr pc, =0xa000800c


(생략)

irq:

ldr pc, =0xa000800c

irq레이블이 두번 나오는 이유는 u-boot 소스에서 #ifdef CONFIG_USE_IRQ를 사용하기 때문이다.

다시 u-boot를 빌드하고 2장에서 했던 과정을 반복합니다.



```
Starting kernel ...
system call 0
```

4. Software Interrupt Handler

1) 컨텍스트 스위칭 - ISR과 태스크

인터럽트가 발생하면 인터럽트 핸들러로 진입해 인터럽트를 처리하고 다시 인터럽트발생전 작업으로 돌아갑니다. 즉 ISR이 실행됩니다. ISR수행후 인터럽트 발생 전 상황으로 돌아가기 위해 컨텍스트를 저장해 놓아야 합니다. 그리고 ISR이 끝나면 저장된 컨텍스트를 다시 불러옵니다. 이 과정을 컨텍스트 스위칭이라 합니다.

컨텍스트 스위칭 구현을 위해서는 별도의 자료구조를 만들어 그곳에 저장합니다. 지금은 Software Interrupt자체에 중점을 두기 위해 임시로 컨텍스트 스위칭을 구현할 것 입니다.

ISR과 태스크간의 컨텍스트 스위칭 과정

1) ARM코어의 data 레지스터를 스택에 저장합니다.

= r0 ~ r14 까지의 레지스터를 스택에 저장합니다.

2) 현재 모드의 spsr을 스택에 저장합니다.

3) ISR을 수행합니다.

4) 백업해 두었던 spsr값, data 레지스터 값을 복구합니다.

= 스택의 첫 4바이트의 값을 spsr에 복구하고, 이후 4bytes씩 읽어가면서 r0~r14에 복구합니다.

2) 스택을 이용한 컨텍스트 스위칭 구현

entry.S

(생략)

testOS_swiHandler:

```
stmfd    sp!,{r0-r12,r14}
mrs      r1,spsr
stmfd    sp!,{r1}
ldr      r10,[lr,#-4]
bic      r10,r10,#0xff000000
mov      r0,r10
bl       swiHandler
ldmfd    sp!,{r1}
msr      spsr_cxsf,r1
ldmfd    sp!,{r0-r12,pc}^
```

stmfd sp!,{r0-r12,r14}

높은 메모리 주소부터 낮은 메모리 주소로 스택을 증가시키면서 r0 ~ r12의 값을 저장하고, r14값을 저장합니다. r13은 sp이기 때문에 자연스럽게 복구됩니다.

mrs r1,spsr

stmfd sp!,{r1}

mrs = Move to Register from Status register입니다. 상태 레지스터의 값을 data레지스터로 복사하는 것입니다. ARM에서는 상태레지스터를 직접 사용할 수 없습니다. 그 후 스택에 r1값(spsr)을 저장합니다.

ldr r10,[lr,#-4]

bic r10,r10,#0xff000000

mov r0,r10

시스템 콜 번호 추출을 위한 코드입니다. swiHandler()는 syscallnum이라는 값을 매개변수도 받습니다. ARM에서 r0, r1, r2, r3이런 순서로 매개변수를 전달하기 때문에 swiHandler로 분기하기전 레지스터에 시스템콜 번호를 저장하면 됩니다.

3) 시스템 콜 번호 추출

ARM은 명령어(1word)를 읽고 pc값을 1워드(보통 4bytes)를 증가시킵니다. 즉 pc값은 다음에 실행할 명령(fetch 할 명령)을 가리키고 있습니다. swi77로 소프트웨어 인터럽트를 발생시키면 pc값은 SVC모드의 lr에 저장됩니다. 그러명 lr에는 swi77 바로 다음 명령을 가리키고 있습니다. 여기서 lr -4를 하면 swi 77 명령에 해당하는 주소가 나옵니다. 여기서 77을 추출합니다.

swi명령은 4bytes중 최상위 1bytes만 명령어고, 나머지는 인자라고 합니다. 그래서 하위 3bytes를 마스킹해서 레지스터에 저장하면 됩니다.

ldr r10,[lr,#-4]

bic r10,r10,#0xff000000 //bic R0, R2, R3 → R3의 1이 있는 bit만 0으로 set

mov r0,r10

bl swiHandler

testOS.c에 있는 swiHandler로 분기합니다.

ldmfd sp!,{r1}

msr spsr_cxsf,r1

ldmfd sp!,{r0-r12,pc}^

가장 나중에 들어간 값은 spsr값이었다. 스택에서 spsr값을 r1 데이터 레지스터에 저장후, spsr로

복구시킨다. 그 후 r0~12, pc값을 복구시킨다. ^기호는 메모리에 있는 데이터를 레지스터에 복구하고, pc값에 있는 곳으로 자동분기하고, 동시에 spsr값을 cpsr로 복사한다. 이로써 인터럽트 발생 전 상황으로 복구된다.

4) 메인함수 수정


testOS.c

```
#include <testOS.h>

void swiHandler(unsigned int syscallnum)
{
    printf("system call %d\n", syscallnum);
}

void irqHandler(void)
{}

int main(void)
{
    while(1){
        __asm__("swi 77");
        msleep(1000);
    }
    return 0;
}
```

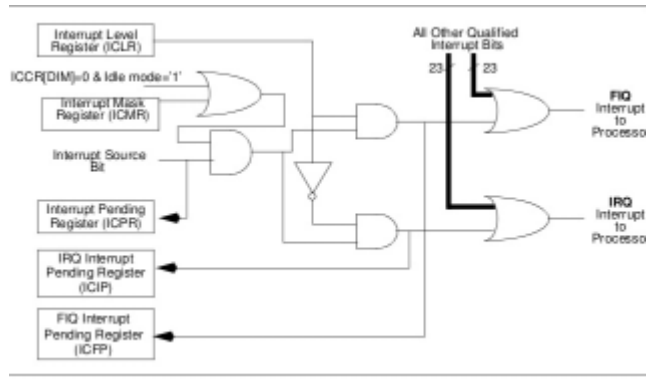


```
OK
Starting kernel ...
system call 77
system call 77
```


5. IRQ Handler – using OS timer

타이머는 외부 인터럽트의 일종이다. 그래서 IRQ나 FIQ로 처리된다.

1) 인터럽트 과정



(PXA255 developers manual)

인터럽트가 감지되면 무조건 ICPR에 기록된다.

ICCR의 0번 비트가 0이고, 프로세서가 Idle모드라면 ICMR에 상관없이 모든 인터럽트를 받는다.

ICMR에서 허용하는 인터럽트만 받으려면 ICCR의 0번 비트를 1로 설정한다.

ICLR의 해당비트가 0이면 IRQ, 1이면 FIQ로 인식한다.

IRQ로 인식하면 ICIP에 기록되고, FIQ로 인식되면 ICFP에 기록된다.

그 후 프로세서에 인터럽트가 전달된다.

2) OS타이머 관련 함수 구현

testOS.c에 다음 함수를 추가한다.

```
void os_timer_init(void)
{
    ICCR = 0x01;
    ICMR |= (1 << 27);           //OS timer1을 사용할 것이기 때문에 27번 비트를 설정한다.
    ICLR &= ~(1 << 27);
    OSCR = 0;                    //카운터 초기화
    OSMR1 = OSCR + 3686400;      //1초간격으로 타이머가 발생하게 설정
    OSSR = OSSR_M1;
}

void os_timer_start(void)
{
    OIER |= (1 << 1);           //1번 OS타이머에 대해 인터럽트 사용
    OSSR = OSSR_M1;
```

```

}

void irq_enable(void)
{
    __asm__("msr    cpsr_c,#0x40|0x13"); //cpsr의 I 비트를 0, 0~5번 비트를 SVC모드로 설정
}

void irq_disable(void)
{
    __asm__("msr    cpsr_c,#0xc0|0x13"); //cpsr의 I 비트를 1, 0~5번 비트를 SVC모드로 설정
}

void irqHandler(void)
{
    if( (ICIP&(1<<27)) != 0 ){                //OS timer1 인지 확인한다
        OSSR = OSSR_M1;
        OSMR1 = OSCR + 3686400;                //값을 더하고
        printf("Timer Interrupt!!!\n");        //출력
    }
}

```

3) 컨텍스트 스위칭 구현

소프트웨어 인터럽트 구현시 사용했던 코드와 비슷하지만 IRQ는 IRQ모드로 동작하기 때문에 커널 시작전 IRQ모드의 스택을 설정하는 코드가 필요합니다. 소프트웨어 인터럽트는 기본모드가 SVC모드이기 때문에 스택설정을 안해도 되지만 IRQ는 모드가 바뀌기 때문에 sp설정을 해야한다.

entry.S

```
.globl _ram_entry
```

```
_ram_entry:
```

```
    b    kernel_init
```

```
    b    _ram_entry
```

```
    b    testOS_swiHandler
```

```
    b    testOS_irqHandler
```

```
#define irq_stack    0xa0380000
```

```
.global kernel_init
```

```
kernel_init:
```

```

msr    cpsr_c,#0xc0|0x12    //IRQ mode
ldr    r0,=irq_stack
sub    sp,r0,#4              //왜 4를 뺄까?-> 4)에서 설명

msr    cpsr_c,#0xc0|0x13

bl     main
b      _ram_entry

```

testOS.c의 main함수에 os_timer_init() , os_timer_start(), irq_enable() 함수 추가

```

Starting kernel ...

system call 77
Timer Interrupt!!!
system call 77
Timer Interrupt!!!

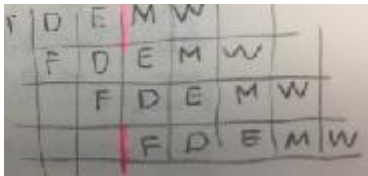
```

4) 파이프라인과 pc, lr

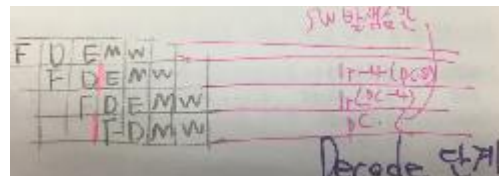
PX255는 ARM9기반이다. 그래서 Fetch - Decode - Execute - memory - WriteBack 5단계의 파이프라인 구조입니다. ARM에서는 Exception이 발생해 프로세서의 동작모드가 바뀔 때, 하드웨어적으로 pc-4값을 변환될 동작모드의 lr에 저장합니다.

코드를 보면 소프트웨어 인터럽트를 할 때 lr은 그대로 사용하고, IRQ일때는 lr-4를 사용합니다.

결정적 차이는 소프트웨어 인터럽트는 Decode가 끝나는 단계에서 발생하고, IRQ는 Execute가 끝나는시점에 발생합니다.



(IRQ)



(소프트웨어 인터럽트)

6. Memory map

에뮬레이터로 사용하는 gumstix의 SDRAM은 64MB입니다. 0xA0000000 ~ 0xA4000000 까지 입니다. 커널의 로딩위치는 0xA0008000입니다. 그리고 2MB는 SVC스택, 0.5MB는 IRQ스택, 0.5MB는 예비로 SYSTEM스택으로 정했습니다. 그위로 40MB는 사용자 영역, 20MB는 힙으로 정했습니다.

1) 스택 주소 초기화

testOS.c의 메인함수(커널의 메인함수)로 들어가기전에 모드를 바꿔가며 sp의 값을 정해줍니다.

entry.S

(생략)

```
#define svc_stack 0xa0300000
```

```
#define irq_stack 0xa0380000
```

```
#define sys_stack 0xa0400000
```

.global kernel_init

kernel_init:

```
msr    cpsr_c,#0xc0|0x13    //SVC mode
```

```
ldr    r0,=svc_stack
```

```
sub    sp,r0,#4
```

```
msr    cpsr_c,#0xc0|0x12    //IRQ mode
```

```
ldr    r0,=irq_stack
```

```
sub    sp,r0,#4
```

```
msr    cpsr_c,#0xc0|0x1f    //SYSTEM mode
```

```
ldr    r0,=sys_stack
```

```
sub    sp,r0,#4
```

7. Memory Manager

1) 메모리 관리자 정의

메모리맵을 구성할 때 SDRAM의 4MB ~ 44MB를 사용자 태스크 공간으로 정했습니다. 각 태스크를 1MB로 구성할 것입니다. 메모리 관리자는 총 40개의 태스크를 각각 블록으로 구분하고, 이를 추상화하는 자료구조를 구성할 것입니다.

블록의 시작주소의 끝주소를 이용해 태스크를 관리할 것입니다.

```
include/testOS_memory.h
```

```
#ifndef _testOS_MEM
```

```
#define _testOS_MEM
```

```
#define MAXMEMBLK 40
```

```
typedef struct _testOS_mem_block {           //메모리 블록 구조체 정의
```

```
    unsigned int block_start_addr;
```

```
    unsigned int block_end_addr;
```

```
    int is_used;
```

```
} TestOS_mem_block;
```

```
typedef struct _testOS_mem_mng {           //메모리 관리자 정의
```

```
    TestOS_mem_block free_mem_pool[MAXMEMBLK];
```

```
    void (*init)(void);
```

```
    unsigned int (*alloc)(void);
```

```
} TestOS_mem_mng;
```

```
void mem_init(void);
```

```
unsigned int mem_alloc(void);
```

```
#endif
```

TestOS_mem_block 구조체는 선형 메모리 공간을 추상화한 자료구조입니다. 변수로는 블록의 시작주소, 끝주소, 사용여부를 나타내는 플래그입니다.

TestOS_mem_mng는 메모리 관리자입니다. Mem_init()은 메모리 관리자를 초기화하고, mem_alloc()은 사용자의 태스크를 메모리에 할당합니다. 또한 포인터를 전달할 함수의 프로토타입이 선언되었습니다.

2) 메모리 관리자 구현

새로 추가한 testOS_memory.h를 testOS.h에 추가해주고, makefile또한 새로 추가된 파일에 맞게 수정합니다.

mem_init()은 for loop에서 정의한 개수 만큼 메모리 블록을 초기화 합니다. 그 후 메모리 블록 함수 포인터를 실제 함수와 연결합니다.

mem_alloc()은 태스크에게 스택의 시작위치를 반환합니다. 메모리 블록 리스트를 보며 사용중인지 확인하고, 비어있는 메모리 블록의 끝 주소를 반환합니다. 스택이 아래로 자라기 때문에 끝주소를 넘겨줘야 합니다.

testOS_memory.c

```
#include <testOS.h>

TestOS_mem_mng memmng;

#define STARTUSRSTACKADDR 0xA0400000 // 4MB
#define USRSTACKSIZE      0x00100000 // 1MB

unsigned int mem_alloc(void)
{
    int i;
    for(i = 0 ; i < MAXMEMBLK ; i++){
        if(memmng.free_mem_pool[i].is_used == 0){
            memmng.free_mem_pool[i].is_used = 1;
            return memmng.free_mem_pool[i].block_end_addr;
        }
    }
    return 0;
}

void mem_init(void)
{
    unsigned int pt = STARTUSRSTACKADDR;
    int i;
    for(i = 0 ; i < MAXMEMBLK ; i++){
        memmng.free_mem_pool[i].block_start_addr = pt;
        memmng.free_mem_pool[i].block_end_addr = pt + USRSTACKSIZE -4;
        memmng.free_mem_pool[i].is_used = 0;
        pt += USRSTACKSIZE;
    }

    memmng.init = mem_init;
    memmng.alloc = mem_alloc;
}
```

8. Process Manager

프로세스는 OS가 제어하는 프로그램의 기본단위입니다. 프로세스를 관리하기 위해서는 태스크를 추상화한 자료구조가 있어야 합니다. 리눅스에서는 PCB라고 합니다.

1) 프로세스 관리자 정의

간단한 구현을 위해 Process ID를 넣지는 않았습니다.

구현 형식은 7장에서 구현한 메모리 관리자와 흡사합니다.

include/testOS_process.h

```
#ifndef _testOS_PROCESS
```

```
#define _testOS_PROCESS
```

```
#define MAXTASKNUM 40
```

```
#define CONTEXTNUM 13
```

```
typedef struct _testOS_process {
```

```
    unsigned int context_spsr;
```

```
    unsigned int context[CONTEXTNUM];
```

```
    unsigned int context_sp;
```

```
    unsigned int context_lr;
```

```
    unsigned int context_pc;
```

```
} TestOS_process;
```

```
typedef struct _testOS_process_mng {
```

```
    TestOS_process free_task_pool[MAXTASKNUM];
```

```
    int max_task_id;
```

```
    void (*init)(void);
```

```
    int (*create)(void(*startFunc)(void));
```

```
} TestOS_process_mng;
```

```
void process_init(void);
```

```
int process_create(void(*startFunc)(void));
```

```
#endif
```

2) 프로세스 관리자 구현

프로세스 관리자 초기화 함수는 프로세스 블록의 변수값을 초기화하고, init과 create변수에 함수포인터를 연결합니다. testOS_memory에 선언되었는 memmng전역변수를 extern으로 선언했습니다. 프로세스를 생성할 때 스택의 주소를 할당받기 위해 메모리 관리자의 alloc()을 사용해야 하기 때문입니다. STARTUSRCPSR이 0x68000050인 이유는 하위비트 50은 이진수로 01010000 = USER모드를 나타낸다. 또한 FIQ와 Thumb를 사용하지 않고, IRQ를 사용하지 않습니다.

testOS_process.c

```
#include <testOS.h>
```

```
extern TestOS_mem_mng memmng;
```

```
TestOS_process_mng processmng;
```

```
#define STARTUSRCPSR    0x68000050
```

```
int process_create(void(*startFunc)(void))
```

```
{
```

```
    int process_idx = 0;
```

```
    unsigned int stack_top = 0;
```

```
    processmng.max_process_id++;
```

```
    process_idx = processmng.max_process_id;
```

```
    if(process_idx > MAXPROCESSNUM){
```

```
        return -1;
```

```
    }
```

```
    stack_top = memmng.alloc();
```

```
    if(stack_top == 0){
```

```
        return -2;
```

```
    }
```

```
    processmng.free_process_pool[process_idx].context_spsr = STARTUSRCPSR;
```

```
    processmng.free_process_pool[process_idx].context_sp = stack_top;
```

```
    processmng.free_process_pool[process_idx].context_pc = (unsigned int)startFunc;
```

```
    return process_idx;
```



```

}

void process_init(void)
{
    int i;
    for(i = 0 ; i < MAXPROCESSNUM ; i++){
        processmng.free_process_pool[i].context_spsr = 0x00;
        memset(processmng.free_process_pool[i].context, 0, sizeof(unsigned int) * CONTEXTNUM);
        processmng.free_process_pool[i].context_sp = 0x00;
        processmng.free_process_pool[i].context_lr = 0x00;
        processmng.free_process_pool[i].context_pc = 0x00;
    }

    processmng.max_process_id = -1;

    processmng.init = process_init;
    processmng.create = process_create;
}

```

3) 사용자 프로세스 추가

함수별로 구분이 되기 때문에 태스크개념이 더 와닿을수도 있지만 리눅스의 PCB를 본따기 위해 프로세스라고 이름지었습니다.

```

testOS_user.c

#include <testOS.h>

extern TestOS_process_mng processmng;

void user_process_1(void)
{
    int a, b, c;

    a = 0;
    b = 1;
    c = a + b;

    printf("process1 - a:%p\tb:%p\tc:%p\n", &a, &b, &c);
}

```

```

void user_process_2(void)
{
    int a, b, c;

    a = 0;
    b = 2;
    c = a + b;

    printf("process2 - a:%p\tb:%p\tc:%p\n", &a, &b, &c);
}

```

```

void user_process_3(void)
{
    int a, b, c;

    a = 0;
    b = 3;
    c = a + b;

    printf("process3 - a:%p\tb:%p\tc:%p\n", &a, &b, &c);
}

```

```

void testOS_user(void)
{
    processmng.create(user_process_1);
    processmng.create(user_process_2);
    processmng.create(user_process_3);
}

```

4) 메인함수 구현

```

extern TestOS_process_mng processmng;
(생략)
void testOS_init(void)
{
    mem_init();
    process_init();
}

```

```

    os_timer_init();
    os_timer_start();
}

int main(void)
{
    testOS_init();
    testOS_user();

    irq_enable();

    int i;
    for(i = 0 ; i <= processmng.max_process_id ; i++){
        printf("PCB : process%d - init PC(%p) \t\t init SP(%p)\n", i+1,
            processmng.free_process_pool[i].context_pc, processmng.free_process_pool[i].context_sp);
    }

    printf("REAL func process1 : %p\n", user_process_1);
    printf("REAL func process2 : %p\n", user_process_2);
    printf("REAL func process3 : %p\n", user_process_3);

    while(1){
        msleep(1000);
    }

    return 0;
}

```

```

Starting kernel ...

PCB : process1 - init PC(a000ba40)          init SP(a04ffffc)
PCB : process2 - init PC(a000ba94)          init SP(a05ffffc)
PCB : process3 - init PC(a000bae8)          init SP(a06ffffc)
REAL func process1 : a000ba40
REAL func process2 : a000ba94
REAL func process3 : a000bae8
Timer Interrupt!!!

```

9. Context Switching

컨텍스트 스위칭은 일정 시간 간격으로 발생해야 합니다. 그래서 OS타이머를 이용해야 합니다. OS타이머는 IRQ로 동작하기 때문에 컨텍스트 스위칭도 IRQ 핸들러로 구현합니다.

컨텍스트 스위칭 과정

- 1) 현재 프로세서에서 동작중인 프로세스의 컨텍스트를 프로세스 컨트롤 블록에 백업합니다.
- 2) 스케줄러를 호출해서 다음에 동작할 프로세스를 선택합니다.
- 3) 선택된 프로세스의 컨텍스트 블록에서 컨텍스트를 가져와서 프로세서의 레지스터에 넣습니다.

1) 컨텍스트 스위칭 구현

entry.s

testOS_irqHandler:

```
//프로세스 context 백업시작
msr    cpsr_c,#0xc0|0x12    //IRQ 비활성화 - 인터럽트 중첩을 허용하면 구현하기 너무 어려울
                             //것 같아 비활성화 했습니다.

ldr    sp,=testOS_current    //PCB를 가리키는 커널 전역 포인터 변수
ldr    sp,[sp]                //testOS_current에서 가져온 주소값은 현재 동작중인 프로세스의
                             //PCB의 주소이다. 즉 TestOS_free_process 구조체의 메모리 주소이다.

sub     lr,lr,#4
add     sp,sp,#4
stmia   sp!, {r0-r12}^        //r0 ~ r12까지 레지스터 값을 백업
stmia   sp!, {sp,lr}^         //user모드의 sp, lr을 백업
stmia   sp!, {lr}              //IRQ모드의 lr을 백업
sub     sp,sp,#68              //TestOS_free_process의 context_spsr을 가리키기 위해
mrs     r1,spsr
stmia   sp!, {r1}
//프로세스 context 백업 끝
//IRQ 핸들러 진입
ldr     sp,=irq_stack

bl      irqHandler
//프로세스 context복구시작
ldr     sp,=testOS_next
ldr     sp,[sp]

ldmia   sp!, {r1}
msr     spsr_cxsf, r1
```

```
ldmia    sp!, {r0-r12}^
ldmia    sp!, {r13,r14}^
ldmia    sp!, {pc}^
```

2) 스케줄러 구현

가장 기본적인 라운드 로빈을 활용한 스케줄러를 구현하겠습니다. 소스는 testOS.c에 구현하겠습니다.

testOS.c

```
TestOS_free_process *testOS_current;    //현재 동작하는 프로세스의 PCB
TestOS_free_process *testOS_next;       //다음에 실행될 프로세스의 PCB
TestOS_free_process dummyTCB;           //커널 부팅후 쓰레기 값 처리를 위한 변수
int testOS_current_index;                //현재 실행중인 프로세스의 인덱스
```

```
void scheduler(void)
{
    testOS_current_index++;
    testOS_current_index %= (processmng.max_process_id + 1);

    testOS_next = &processmng.free_process_pool[testOS_current_index];
    testOS_current = testOS_next;
}
```

스케줄러 초기화 코드 작성

```
int sched_init(void)
{
    if(processmng.max_process_id < 0){
        return -1;
    }

    testOS_current = &dummyTCB;           //쓰레기값 처리
    testOS_next = &processmng.free_process_pool[0];    //프로세스블록 리스트의 첫번째 인덱스
    testOS_current_index = -1;
    return 0;
}
```

3) 그 외 함수 수정

testOS.c

메인 함수 수정

```
main(void)
```

(생략)

```
if(sched_init() < 0){
    printf("Kernel Pannic!\n");
    return -1;
}
```

OS타이머 핸들러 수정

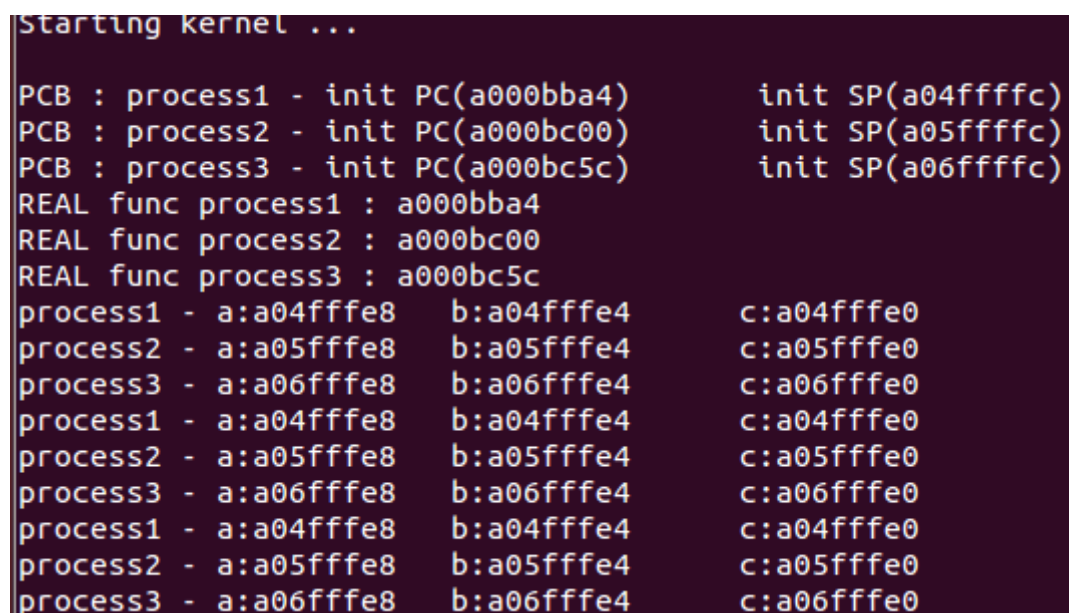
```
void irqHandler(void)
{
    if( (ICIP&(1<<27)) != 0 ){
        OSSR = OSSR_M1;
        OSMR1 = OSCR + 3686400;
        scheduler();
    }
}
```

스케줄러가 호출되기 전까진 testOS_current와 testOS_next의 값이 같다. 그러므로 OS타이머가 아닌 다른 IRQ가 발생하면 프로세스 컨텍스트 스위칭은 발생하지 않는다.

사용자 프로세스 수정 - 무한루프 추가

testOS_user.c

```
while(1){
    printf("process1 - a:%pWtb:%pWtc:%pWn", &a, &b, &c);
    msleep(1000);
}
```



A terminal window with a dark background and light-colored text. The output shows the kernel starting and initializing three processes. Each process has a PCB entry with its PC and SP values, followed by its REAL func address. Then, for each process, three memory addresses (a, b, c) are printed in hexadecimal, showing they are all initialized to the same value (e.g., a04fffe8 for process1).

```
Starting kernel ...
PCB : process1 - init PC(a000bba4)      init SP(a04ffffc)
PCB : process2 - init PC(a000bc00)      init SP(a05ffffc)
PCB : process3 - init PC(a000bc5c)      init SP(a06ffffc)
REAL func process1 : a000bba4
REAL func process2 : a000bc00
REAL func process3 : a000bc5c
process1 - a:a04fffe8    b:a04fffe4    c:a04fffe0
process2 - a:a05fffe8    b:a05fffe4    c:a05fffe0
process3 - a:a06fffe8    b:a06fffe4    c:a06fffe0
process1 - a:a04fffe8    b:a04fffe4    c:a04fffe0
process2 - a:a05fffe8    b:a05fffe4    c:a05fffe0
process3 - a:a06fffe8    b:a06fffe4    c:a06fffe0
process1 - a:a04fffe8    b:a04fffe4    c:a04fffe0
process2 - a:a05fffe8    b:a05fffe4    c:a05fffe0
process3 - a:a06fffe8    b:a06fffe4    c:a06fffe0
```

10. System Call

커널 스레드나 디바이스 드라이버 계층에서만 커널 메모리 주소 영역에 접근할 수 있습니다. 즉 특권을 가진 프로세스만 커널의 메모리에 접근 가능합니다. 그래서 사용자 프로세스가 시스템 자원을 이용하려면 커널에게 서비스 요청을 해야 합니다. 이때 사용하는것이 시스템 콜 입니다.

1) 시스템 콜 커널 함수 구현

순서

사용자 프로세스에서는 리눅스의 glibc처럼 testOS 자체 라이브러리에 정의된 래퍼함수를 호출합니다.

래퍼 함수는 내부에서 swi명령으로 소프트웨어 인터럽트를 발생시킵니다.

그 후 소프트웨어 인터럽트에 대한 ISR를 수행합니다.

시스템 콜 벡터 테이블에 정의된 실제 시스템 콜 함수로 분기해 명령을 실행합니다.

testOS_sys.c 라는 파일을 만들고, 시스템콜 벡터 테이블로 사용할 전역변수를 선언합니다. 이 배열은 unsigned int형 값을 가집니다. 함수 포인터도 주소이기 때문에 1워드입니다. 그래서 unsigned int를 사용해도 문제가 없습니다.

testOS_sys.c

```
#include <testOS.h>
```

```
unsigned int testOS_syscallvec[SYSCALLNUM];
```

```
int sys_mysyscall(int a, int b, int c)
```

```
{
```

```
    printf("My Systemcall - %d , %d , %d\n", a, b, c);
```

```
    return 333;
```

```
}
```

```
int sys_mysyscall4(int a, int b, int c, int d)
```

```
{
```

```
    printf("My Systemcall4 - %d , %d , %d , %d\n", a, b, c, d);
```

```
    return 3413;
```

```
}
```

```
void syscall_init(void)
```

```
{
```

```
    testOS_syscallvec[SYS_MYSYSCALL] = (unsigned int)sys_mysyscall;
```

```
    testOS_syscallvec[SYS_MYSYSCALL4] = (unsigned int)sys_mysyscall4;
```

```
}
```

2) 시스템 콜 초기화 함수 호출

testOS.c에 있는 testOS_init()함수 내부에 syscall_init()함수를 추가합니다.

3) 시스템 콜 관련 헤더파일 작성

시스템 콜 함수를 선언하는 헤더파일(include/testOS_sys.h)과 시스템 콜 번호를 정의하는 헤더파일(syscalltbl.h)를 정의 할 것입니다.

```
include/testOS_sys.h    //앞으로 시스템 콜을 추가할 때 마다 프로토 타입을 이 파일에 추가합니다.
#ifndef _testOS_SYS
#define _testOS_SYS
#include <syscalltbl.h>
#define SYSCALLNUM 255
void syscall_init(void);
int sys_mysyscall(int, int, int);
int sys_mysyscall4(int, int, int, int);
#endif
```

```
include/syscalltbl.h    //나중에 어셈블리 소스에서도 include할 것이기 때문에 오로지 define문장
#ifndef _testOS_SYS_TBL    으로만 작성한다.
#define _testOS_SYS_TBL
#define SYS_MYSYSCALL    0
#define SYS_MYSYSCALL4    1
#endif
```

4) 사용자 프로세스 함수 수정

```
testOS_user.c          //user_process_3() 함수 수정
void user_process_3(void)
{
    int a, b, c;
    a = 0;
    b = 3;
    c = a + b;
    while(1){
        printf("process3 - a:%pWtb:%pWtc:%pWn", &a, &b, &c);
        c = mysyscall(1,2,3);
        printf("Syacall return value is %dWn",c);
        msleep(1000);
    }
}
```

Mysyscall()함수는 미리 구현한 sys_mysyscall() 시스템 콜 함수에 대한 래퍼 함수를 호출 한 것 입니다.
mysyscall()함수는 래퍼 함수를 모아놓은 testOS_lib.S에 구현할 것 입니다.

5) 시스템 콜 래퍼 함수 구현

래퍼함수는 특별한 기능 없이 SWI명령으로 소프트웨어 인터럽트를 호출합니다.

testOS_lib.S

```
#include <syscalltbl.h>
```

```
.global mysyscall
```

```
mysyscall:
```

```
    swi SYS_MYSYSCALL //swi명령을 수행해 ISR로 진입합니다. IRQ에서 복귀한 후에는  
    mov pc, lr        //복귀==lr에 저장된 곳으로 복귀합니다.
```

```
.global mysyscall4
```

```
mysyscall4:
```

```
    swi SYS_MYSYSCALL4  
    mov pc, lr
```

어셈블리로 작성된 mysyscall을 C에서 호출하려면 프로토타입이 선언되어있어야 합니다. 파일명은 testOS_lib.h입니다.

include/testOS_lib.h

```
#ifndef _testOS_LIB
```

```
#define _testOS_LIB
```

```
extern int mysyscall(int, int, int);
```

```
extern int mysyscall4(int, int, int, int);
```

```
#endif
```

6) 소프트웨어 인터럽트 ISR 수정

시스템 콜 번호를 추출하고, 번호에 따라 연결되어 있는 시스템 콜 벡터 테이블의 함수로 분기하는 코드를 작성해야 합니다.

```
entry.S
```

```
testOS_swiHandler:
```

```
    //USER모드의 context를 SVC stack에 백업
```

```
    msr    cpsr_c, #0xc0|0x13 //IRQ 비활성화
```

```
    ldr    sp, =svc_stack
```

```
    stmfd  sp!, {lr}
```

```
    stmfd  sp!, {r1-r14}^
```

```
    mrs    r10, spsr
```

```
    stmfd  sp!, {r10}
```

```
    //USER모드의 context를 SVC stack에 백업 끝
```

```

//시스템 콜 번호 추출
ldr    r10, [lr,#-4]
bic    r10, r10, #0xff000000
mov    r11, #4
mul    r10, r10, r11           //testOS_syscallvec변수는 요소가 4bytes이기 때문에 시스템 콜
                               //번호로 인덱싱 하기위해 4를 곱한다.

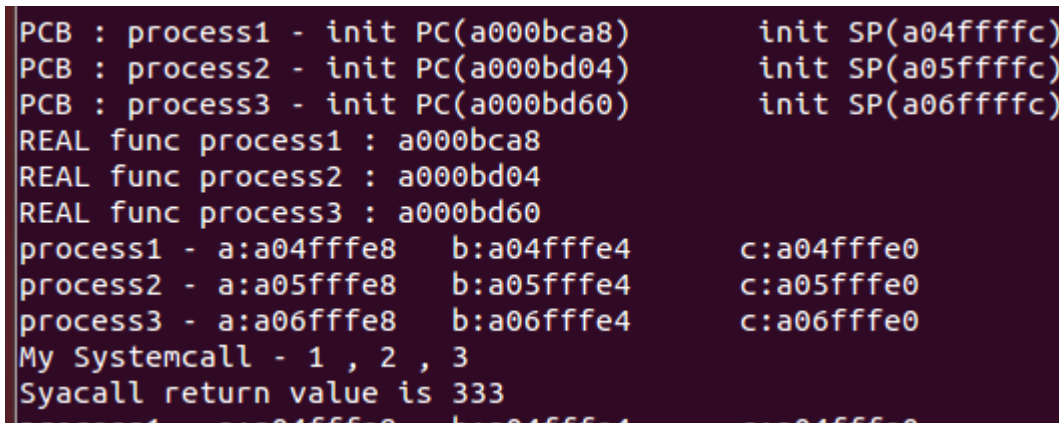
//시스템 콜 번호 추출 끝
//pc값 백업 및 시스템 콜 함수로 분기
ldr    r11, =testOS_syscallvec //커널 전역 배열 변수의 시작주소를 받아옵니다.
add    r11, r11, r10           //시스템 콜 함수의 시작주소가 있는 곳으로 분기
ldr    r11, [r11]              //해당 메모리 값 읽는다.=r11에는 시스템 콜 함수의 주소가 들어감.
mov    lr, pc                  //시스템 콜 진입 전 lr에 pc값을 백업합니다. 그래서 시스템 콜 종료
                               //후 복귀 가능

mov    pc, r11                 //pc로 r11복사해서 시스템 콜 함수로 진입

//context복구
ldmfd  sp!, {r1}
msr    spsr_cxsf, r1
ldmfd  sp!, {r1-r14}^
ldmfd  sp!, {pc}^

```

소스가 수정됐으므로 Makefile을 수정합니다.



```

PCB : process1 - init PC(a000bca8)      init SP(a04ffffc)
PCB : process2 - init PC(a000bd04)      init SP(a05ffffc)
PCB : process3 - init PC(a000bd60)      init SP(a06ffffc)
REAL func process1 : a000bca8
REAL func process2 : a000bd04
REAL func process3 : a000bd60
process1 - a:a04fffe8  b:a04fffe4      c:a04fffe0
process2 - a:a05fffe8  b:a05fffe4      c:a05fffe0
process3 - a:a06fffe8  b:a06fffe4      c:a06fffe0
My Systemcall - 1 , 2 , 3
Syacall return value is 333

```

7) 정리 - 시스템 콜 추가 절차

1. sysscalltbl.h에 시스템 콜 번호를 추가합니다.
2. testOS_sys.c에 시스템 콜 함수를 작성합니다.
3. syacall_init()함수에서 testOS_syscallvec에 새로 추가한 시스템 콜의 함수 포인터를 등록합니다.
4. testOS_sys.h에 시스템 콜의 프로토 타입을 선언합니다.
5. testOS_lib.h에 시스템 콜 래퍼 함수의 프로토타입을 선언합니다.
6. testOS_lib.S에 시스템 콜 래퍼 함수를 작성합니다.
7. 사용자 프로세스에서 시스템 콜을 사용합니다.

11. Communication with Process

IPC는 USER모드 프로세스간에 통신을 할 수 있게 해주는 시스템 콜의 집합입니다. 파이프, FIFO, 메시지큐, 공유메모리 등 여러 방법이 있습니다.

IPC로 통신을 할 때, IPC에 데이터가 없다면 프로세스는 데이터가 들어올 때까지 블로킹 되어야 합니다. 블로킹을 구현하는 방법으로 블로킹이 걸려야 하는 상황에 컨텍스트 스위칭을 발생시켜 다음 프로세스로 컨텍스트를 넘기겠습니다. 사용자가 프로세스가 커널에 요청하는 것 이므로 시스템 콜 계층에 구현해야 합니다.

1) 컨텍스트 스위칭 시스템 콜 구현

entry.S에 컨텍스트 스위칭과 스케줄러만 호출하는 어셈블리 레이블을 만들겠습니다.

entry.S

.global sys_scheduler:

sys_scheduler:

ldr sp, =testOS_current

ldr sp, [sp]

add sp, sp, #4

stmia sp!, {r0-r12}^

stmia sp!, {sp,lr}^

stmia sp!, {lr}

sub sp, sp, #68

mrs r1, spsr

stmia sp!, {r1}

ldr sp, =svc_stack

bl scheduler

ldr sp, =testOS_next

ldr sp, [sp]

ldmia sp!, {r1}

msr spsr_cxsf, r1

ldmia sp!, {r0-r12}^

ldmia sp!, {r13,r14}^

ldmia sp!, {pc}^

sys_scheduler는 시스템 콜 함수 입니다. 이 함수를 전 장에서 했던 것처럼 시스템 콜을 추가 하면 됩니다.(시스템 콜 번호 추가, 시스템 콜 래퍼함수 프로토 타입 선언, testOS_lib.S 래퍼함수 구현)

entry.S파일을 수정할 때 sys_scheduler는 컨텍스트 스위칭과 스케줄러가 동작해야하기 때문에 다른 시스템콜과는 별도의 로직이 필요합니다.

entry.S

#include "syscalltbl.h"

...

.global testOS_swiHandler

testOS_swiHandler:

msr cpsr_c, #0xc0|0x13

ldr sp, =svc_stack

stmfd sp!, {lr}

stmfd sp!, {r1-r14}^

mrs r10, spsr

stmfd sp!, {r10}

ldr r10, [lr, #-4]

bic r10, r10, #0xff000000

cmp r10, #SYS_CALLSCHED

beq sys_scheduler

mov r11, #4

mul r10, r10, r11

사용자 프로세스에서 스케줄러 호출

while(1){

...

printf("before call Scheduler\n");

call_scheduler();

printf("after Scheduler\n");

...

}

```

process1 - a:a04fffe8    b:a04fffe4    c:a04fffe0
before call Scheduler
process2 - a:a05fffe8    b:a05fffe4    c:a05fffe0
process3 - a:a06fffe8    b:a06fffe4    c:a06fffe0
My Systemcall - 1 , 2 , 3
Syacall return value is 333
after Scheduler

```

2) 메시지 관리자 정의 – 메세지큐

메시지 관리자도 메시지 블록과 메시지 관리자 자료구조를 구성할 것 입니다.

include/testOS_msg.h

```

typedef struct _testOS_msg_mng {
    testOS_free_msg free_msg_pool[MAXMSG];
    void (*init)(void);
    int (*itc_send)(int, int);
    int (*itc_get)(int, int*);
} testOS_msg_mng;

void msg_init(void);
int msg_itc_send(int, int);
int msg_itc_get(int, int*);
#endif

```

testOS_msg.c

```

#include <testOS.h>

TestOS_msg_mng msgmng;

int msg_itc_send(int itcnum, int data)
{
    if(itcnum >= MAXMSG || itcnum < 0){
        return -1;
    }
    msgmng.free_msg_pool[itcnum].data = data;
    msgmng.free_msg_pool[itcnum].flag = 1;
    return itcnum;
}

int msg_itc_get(int itcnum, int *data)
{
    if(itcnum >= MAXMSG || itcnum < 0){
        return -1;
    }

```

```

    }
    if(msgmng.free_msg_pool[itcnum].flag == 0){
        return -2;
    }
    *data = msgmng.free_msg_pool[itcnum].data;
    msgmng.free_msg_pool[itcnum].flag = 0;
    msgmng.free_msg_pool[itcnum].data = 0;
    return 0;
}

void msg_init(void)
{
    int i;
    for (i = 0 ; i < MAXMSG ; i++){
        msgmng.free_msg_pool[i].data = 0;
        msgmng.free_msg_pool[i].flag = 0;
    }
    msgmng.init = msg_init;
    msgmng.itc_send = msg_itc_send;
    msgmng.itc_get = msg_itc_get;
}

```

이제 시스템 콜이 추가됐을 때 해야할 과정을 수행합니다.

...

사용자 프로세스 영역에서 블로킹이 걸리게 하려면 시스템 콜 래퍼 함수가 두단계에 걸쳐서 호출되어야 한다.

testOS_lib.h

...

```
int testOS_ipc_send(int,int);
```

```
int testOS_ipc_get(int,int);
```

시스템 콜 C래퍼 함수 작성

testOS_clib.c – 새로 작성

```
#include <testOS.h>
```

```
int testOS_ipc_send(int ipcnum, int data)
```

```
{
```

```
    return ipc_send(ipcnum, data);
```

```
}
```

```
int testOS_ipc_get(int ipcnum)
```

```

{
    int ret_value = 0;
    int data = 0;
    while(1){
        ret_value = ipc_get(ipcnum, &data);
        if(ret_value == 0){
            return data;
        }else if(ret_value == -1){
            return ret_value;
        }else{
            call_scheduler();          //블로킹
        }
    }
}

```

먼저 ipc_get()함수를 호출하여 시스템 콜 계층으로부터 넘겨받은 리턴 값을 검사합니다. 0이 리턴된다면 정상이기 때문에 두번째 매개변수인 data에 있는 값을 리턴합니다. 이 외의 경우에는 블로킹이 걸립니다.

사용자 프로세스 수정

프로세스2에서 메시지를 보내고 프로세스3에서 메시지를 받습니다.

```

void user_process_2(void)
{
    int a, b, c;
    a = 0;
    b = 2;
    c = a + b;
    while(1){
        printf("process2 - a:%pWtb:%pWtc:%pWn", &a, &b, &c);
        printf("IPC Count is %dWn", a);
        if (a == 3){
            navilnux_itc_send(2, 342);
            a = 1;
            printf("IPC send!!!Wn");
        }
        a++;
        msleep(1000);
    }
}

```

```

void user_process_3(void)
{
    int a, b, c;
    a = 0;
    b = 3;
    c = a + b;
    while(1){
        c = testOS_ipc_get(2);
        printf("PROCESS3 - a:%pWtb:%pWtc:%pWn", &a, &b, &c);
        printf("IPC get!!!! ---> %dWn", c);
        msleep(1000);
    }
}

```

```

process2 - a:a05fffe8    b:a05fffe4    c:a05fffe0
IPC Count is 3
IPC send!!!
PROCESS3 - a:a06fffe8    b:a06fffe4    c:a06fffe0
IPC get!!!! ---> 342

```

12. Synchronization

1)세마포어 정의

S는 세마포어 변수, P는 세마포어를 잠그는 역할, V는 세마포어를 푸는 역할입니다. 크리티컬 섹션의 앞부분에서 P함수를 실행하고, 나올때 v함수를 실행합니다.

```
include/testOS_mgs.h
```

```
...
```

```
#define ITCSTART 0
```

```
#define ITCEND 99
```

```
#define SEMSTART 100
```

```
#define SEMEND 199
```

```
...
```

```
typedef struct _testOS_msg_mng {
```

```
    TestOS_free_msg free_msg_pool[MAXMSG];
```

```
    void (*init)(void);
```

```
    int (*ipc_send)(int, int);
```

```
    int (*ipc_get)(int, int*);
```

```
    int (*sem_init)(int, int);
```

```
    int (*sem_p)(int);
```

```
    int (*sem_v)(int);
```

```
} TestOS_msg_mng;
```

```
...
```

```
int msg_sem_init(int,int);
```

```
int msg_sem_p(int);
```

```
int msg_sem_v(int);
```

255개의 메시지 블록중 0~99까지를 IPC, 100~199까지를 세마포어가 사용하게 했습니다.

2) 세마포어 구현

```
testOS_msg.c
```

```
int msg_sem_init(int semnum, int s)
```

```
{
```

```
    semnum += SEMSTART;
```

```
    if(semnum > SEMEND || semnum < SEMSTART){
```

```
        return -1;
```

```
    }
```

```
    msgmng.free_msg_pool[semnum].flag = s;
```

```

        return 0;
    }

int msg_sem_p(int semnum)
{
    semnum += SEMSTART;
    if(semnum > SEMEND || semnum < SEMSTART){
        return -1;
    }
    if(msgmng.free_msg_pool[semnum].flag <= 0){
        return -2;
    }
    msgmng.free_msg_pool[semnum].flag--;
    return 0;
}

int msg_sem_v(int semnum)
{
    semnum += SEMSTART;
    if(semnum > SEMEND || semnum < SEMSTART){
        return -1;
    }
    msgmng.free_msg_pool[semnum].flag++;
    return 0;
}

```

```

process4 enter critical section SEMAPHORE
process4 - a:a07fffe8  b:a07fffe4  c:a07fffe0
process5 enter critical section SEMAPHORE
process5 - a:a08fffe8  b:a08fffe4  c:a08fffe0
after Scheduler
process2 - a:a05fffe8  b:a05fffe4  c:a05fffe0
IPC Count is 1
process4 out critical section SEMAPHORE
process6 enter critical section SEMAPHORE
process6 - a:a09fffe8  b:a09fffe4  c:a09fffe0
process2 - a:a05fffe8  b:a05fffe4  c:a05fffe0

```

13. Device Driver

1) 디바이스 드라이버 관리자 정의

testOS_drv.h – 디바이스 드라이버관리자의 자료구조 정의

```
#ifndef _testOS_DRV
#define _testOS_DRV
#define DRVLIMIT 100
#define O_RDONLY    0
#define O_WRONLY    1
#define O_RDWR      2
typedef struct _fops{
    int (*open)(int drvnum, int mode);
    int (*read)(int drvnum, void *buf, int size);
    int (*write)(int drvnum, void *buf, int size);
    int (*close)(int drvnum);
} fops;
typedef struct _testOS_free_drv {
    fops *testOS_fops;
    int usecount;
    const char *drvname;
} TestOS_free_drv;
typedef struct _testOS_drv_mng {
    TestOS_free_drv free_drv_pool[DRVLIMIT];
    void (*init)(void);
    int (*register_drv)(int, const char*, fops*);
} TestOS_drv_mng;
void drv_init(void);
int drv_register_drv(int, const char*, fops*);

#endif
```

fops구조체

가장 기본적인 open, read, write, close만 정의했습니다.

TestOS_free_drv구조체가 하나의 디바이스 드라이버를 추상화 할 것입니다.

2) 디바이스 드라이버 관리자 구현

디바이스 드라이버 관리자에는 디바이스 드라이버 관리자를 초기화하는 init함수포인터와 디바이스 드라이버의 fops를 커널에 등록하는 register_drv 함수 포인터가 있습니다.

testOS_drv.c

```
#include <testOS.h>

TestOS_drv_mng drvmng;

int drv_register_drv(int drvnum, const char *name, fops *drvfops)
{
    if(drvnum > DRVLIMIT || drvnum < 0){
        return -1;
    }
    if(drvmng.free_drv_pool[drvnum].usecount >= 0){
        return -1;
    }
    drvmng.free_drv_pool[drvnum].testOS_fops = drvfops;
    drvmng.free_drv_pool[drvnum].drvname = name;
    drvmng.free_drv_pool[drvnum].usecount = 0;
    return 0;
}

void drv_init()
{
    int i;
    for(i = 0 ; i < DRVLIMIT ; i++){
        drvmng.free_drv_pool[i].testOS_fops = (fops *)0;
        drvmng.free_drv_pool[i].usecount = -1;
        drvmng.free_drv_pool[i].drvname = (const char *)0;
    }
    drvmng.init = drv_init;
    drvmng.register_drv = drv_register_drv;
}
```

아직 이상 상태에서 디바이스 드라이버를 사용자 프로세스에서 사용할 수 없습니다. 디바이스 드라이버계층을 시스템 콜 계층에 올려야 하기 때문입니다.

3) 시스템 콜에 등록

include/syscalltbl.h

```
#define SYS_OPEN      10
#define SYS_CLOSE     11
#define SYS_READ      12
#define SYS_WRITE     13
```

시스템 콜 함수 선언 - fops구조체에 선언된 함수포인터의 매개변수 타입과 일치하게 각 함수의 프로토타입을 잡습니다.

include/testOS_sys.h

```
int sys_open(int, int);
int sys_close(int);
int sys_read(int, void*, int);
int sys_write(int, void*, int);
```

시스템 콜 함수 작성

testOS_sys.c

```
...
extern TestOS_msg_mng msgmng;
extern TestOS_mem_mng memmng;
extern TestOS_drv_mng drvmng;
...
int sys_open(int drvnum, int mode)
{
    if(drvnum > DRVLIMIT || drvnum < 0){
        return -1;
    }

    if(drvmng.free_drv_pool[drvnum].usecount < 0){
        return -1;
    }

    drvmng.free_drv_pool[drvnum].usecount++;
    return drvmng.free_drv_pool[drvnum].testOS_fops->open(drvnum, mode);
}

int sys_close(int drvnum)
{

```

```

    if(drvnum > DRVLIMIT || drvnum < 0){
        return -1;
    }

    drvmng.free_drv_pool[drvnum].usecount--;
    return drvmng.free_drv_pool[drvnum].testOS_fops->close(drvnum);
}

int sys_read(int drvnum, void *buf, int size)
{
    if(drvnum > DRVLIMIT || drvnum < 0){
        return -1;
    }

    return drvmng.free_drv_pool[drvnum].testOS_fops->read(drvnum, buf, size);
}

int sys_write(int drvnum, void *buf, int size)
{
    if(drvnum > DRVLIMIT || drvnum < 0){
        return -1;
    }

    return drvmng.free_drv_pool[drvnum].testOS_fops->write(drvnum, buf, size);
}

void syscall_init(void)
{
    ...

    testOS_syscallvec[SYS_OPEN] = (unsigned int)sys_open;
    testOS_syscallvec[SYS_CLOSE] = (unsigned int)sys_close;
    testOS_syscallvec[SYS_READ] = (unsigned int)sys_read;
    testOS_syscallvec[SYS_WRITE] = (unsigned int)sys_write;
}

```

이제 시스템 콜 래퍼 함수를 구현해야 합니다. 시스템콜 래퍼 함수의 프로토 타입은 testOS_lib.h에 모두 선언되어 있습니다. 또한 Read()함수는 읽을값이 없을때 블로킹되어야 하므로 IPC구현처럼 한번더 감싸는 형식으로 testOS_clib.c에 구현해야합니다.

include/testOS_lib.h

...

```
extern int open(int, int);
extern int close(int);
extern int read(int, void*, int);
extern int write(int, void*, int);
...
```

```
int testOS_open(int, int);
```

//아래 4 함수는 사용자 프로세스에게 제공되는 래퍼함수의 프로토타입 입니다.

```
int testOS_close(int);
int testOS_read(int, void*, int);
int testOS_write(int, void*, int);
,
```

testOS_lib.S

...

```
.global open
```

```
open:
```

```
    swi SYS_OPEN
```

```
    mov pc, lr
```

```
.global close
```

```
close:
```

```
    swi SYS_CLOSE
```

```
    mov pc, lr
```

```
.global read
```

```
read:
```

```
    swi SYS_READ
```

```
    mov pc, lr
```

```
.global write
```

```
write:
```

```
    swi SYS_WRITE
```

```
    mov pc, lr
```

testOS_clib.c에서 testOS_open(), testOS_close(), testOS_write()함수는 계층 유지를 할뿐 다른 기능은 없습니다. testOS_read()함수만 브로킹을 위한 로직이 구현되었습니다.

testOS_clib.c

...

```

int testOS_open(int drvnum, int mode)
{
    return open(drvnum, mode);
}

int testOS_close(int drvnum)
{
    return close(drvnum);
}

int testOS_read(int drvnum, void *buf, int size)
{
    int ret_value = 0;
    while(1){
        ret_value = read(drvnum, buf, size);
        if(ret_value >= 0){
            return ret_value;
        }else if(ret_value == -1){
            return -1;
        }else{
            call_scheduler();
        }
    }
}

int testOS_write(int drvnum, void *buf, int size)
{
    return write(drvnum, buf, size);
}

```


4) 디바이스 드라이버 추가

LED스위치를 제어하는 디바이스 드라이버 - 에뮬레이터 환경이다 보니 동작을 확인할 수는 없습니다.

스위치에 대한 인터럽트 핸들러는 IRQ핸들러를 이용해 구현 할 수 있습니다. 이때 IRQ핸들러 벡터 테이블을 만들고, IRQ핸들러 함수를 등록, IRQ핸들러에서는 IRQ핸들러 벡터 테이블에 등록되었는 함수를 호출하는 식으로 작성해보겠습니다.

testOS.c - IRQ핸들러 벡터 테이블을 전역 변수로 추가합니다.

```
int (*testOS_irq_vector[IRQNUM])(void);
```

testOS_irq_vector라는 함수포인터 배열을 선언했습니다. IRQNUM만큼의 배열입니다. 관련 define은 testOS.h에 선언했습니다.

testOS.h

```
#define IRQNUM 64  
  
#define IRQ0    0  
#define IRQ1    1  
#define IRQ2    2  
#define IRQ3    3  
#define IRQ4    4  
#define IRQ5    5  
#define IRQ6    6  
#define IRQ7    7  
#define IRQ8    8  
#define IRQ63   63
```

irq_Handler의 내용을 수정합니다.

```
if ( (ICIP&(1<<8)) != 0 ){  
    GEDR0 = 1;  
    if(testOS_irq_vector[IRQ8] != NULL) testOS_irq_vector[IRQ8]();  
}
```

GPIO<0>만 처리합니다. ICIP의 8번 비트에 IRQ8 벡터에 실제 핸들러 함수를 할당했습니다. 핸들러가 할당되지 않은 인터럽트가 발생할 수도 있기 때문에 NULL인지 먼저 확인합니다. 그 후 핸들러 함수를 실행합니다.

read(), write()구현

read()함수는 스위치 입력을 기다리고, write()함수는 LED를 제어하는 디바이스 드라이버 입니다. 새로운 파일을 생성합니다. 그리고 코드내부에서 디바이스 드라이버 이름을 mydrv로 정했습니다.

```
#include <testOS.h>  
  
extern TestOS_drv_mng drvmng;  
  
extern int (*testOS_irq_vector[64])(void);  
  
int switch_pushed;
```

```

int gpio0_irq_handler(void)
{
    if(switch_pushed) return -1;

    printf("Switch Push!! in Device Driver Layer\n");
    switch_pushed = 1;
    return 0;
}

int mydrv_open(int drvnum, int mode)
{
    testOS_irq_vector[IRQ8] = gpio0_irq_handler;
    switch_pushed = 0;
    return 0;
}

int mydrv_close(int drvnum)
{
    return 0;
}

int mydrv_read(int drvnum, void *buf, int size)
{
    int *b = (int *)buf;
    if(switch_pushed == 1){
        *b = switch_pushed;
        switch_pushed = 0;
        return 4;
    }else{
        return -2;
    }
}

int mydrv_write(int drvnum, void *buf, int size)
{
    int *b = (int *)buf;
    int n = (int)b[0];
    int s = (int)b[1];
    GPIO_SetLED(n,s);
    return 0;
}

```

```

}
fops mydrv_fops =
{
    open : mydrv_open,
    read : mydrv_read,
    write : mydrv_write,
    close : mydrv_close,
};
int mydrv_init()
{
    return drvmng.register_drv(MYDRV, "testOS first drv", &mydrv_fops);
}

```

드바이스 드라이버 구조체인 drvmng와 testOS_irq_vector 전역변수를 extern으로 가져왔습니다. gpio0_irq_handler()함수는 testOS_irq_vector에 등록될 핸들러 함수 입니다. 즉 GPIO<0>에 들어오는 IRQ에 대한 최종 수행코드입니다.

mydrv_read()함수

스위치에서 들어오는 입력을 프로세스로 전달합니다. 내부 포인터 변수 b에 buf로 넘어온 주소값을 저장합니다. void*타입으로 넘어오기 때문에 한번더 사용하는 것 입니다. 그리고 switch_pushed변수의 값을 검사합니다. 1이면 스위치가 눌린것이므로 그 값을 b로 전달합니다. -2가 반환되면 블로킹됩니다.

함수 구현후 fops구조체에 함수들을 등록해야합니다. mydrv_fops라는 fops구조체 변수를 선언하고, 연결시킵니다. 그리고 지금까지 구현한 mydrv모듈을 초기화 시키는 mydrv_init()를 구현합니다. 이 mydrv_init()은 testOS_drv.c의 drv_init()에서 호출되어야 합니다.

testOS_drv.c의 drv_init()함수 맨 마지막에 mydrv_init();을 추가합니다. 또한 서로 다른파일에서 함수를 호출했으므로 프로토타입을 testOS_drv.h에 선언합니다.

```

#define MYDRV 0          //mydrv를 구분하게 될 이름인 MYDRV
Int mydrv_init(void);

```

사용자 디바이스 드라이버 테스트

->

```

testOS_open(MYDRV, O_RDWR);  //장치를 open하고
                               //
testOS_read(MYDRV,&a,4)        // 블로킹 되었다가 입력이 들어오면 디바이스에서 읽어온 값을
                               a에 저장합니다.
testOS_write(MYDRV,led,8);     //led제어

```