TI DSP,MCU 및 Xilinux Zynq FPGA

프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/4/12
수업일수	36 일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

Chapter 6 인터럽트와 트랩 그리고 시스템 호출

(1) 인터럽트 처리 과정

: 인터럽트(Interrupt) - 주변 장치와 커널이 통신하는 방식 중의 하나로 주변 장치나 CPU 가 자신에게 발생한 사건을 리눅스 커널에게 알리는 매커니즘.
: 인터럽트가 발생되면 발생한 이유를 찾고 처리해야하는데, 작업을 처리하는 함수를 인터럽트 핸들러(interrupt handler)라고 부른다.
: 모든 CPU 는 인터럽트가 발생하면 program counter(or instruction pointer)레지스터의 값을 미리 정해진 특정 번지로 변경하도록 정해져있다.
ARM CPU 인 경우 인터럽트가 발생하면 0x00000000+offset 번지로 점프한다.

> offset 은 인터럽트 종류에 따라 결정되는데,
> reset interrupt → offset 0
> undefined instruction(정의되지 않은 인터럽트) → offset 4
> software interrupt → offset 8

원인에 따른 인터럽트 구분
: CPU 내부에서 감지하는지 외부에서 감지하는지를 기준으로 한다.
1. 외부 인터럽트
CPU 의 수 많은 핀에 연결된 주변 장치(키보드, 마우스 …)에서 발생된 비동기적 하드웨어적인 사건
2. 내부 인터럽트
현재 수행중인 태스크와 관련있고 동기적으로 발생하는 사건.
트랩 = 예외 처리 = 내부 인터럽트
내부 인터럽트에는 0으로 나누는 연산(device by zero), 세그멘테이션 결함, 펭지 결함, 보호결함, 시스템 콜 등이 있다.

ARM CPU 의 인터럽트 디스크립터 테이블(IDT)
(사진첨부)

: IDT 인 id_table 이라는 이름의 배열에 외부 인터럽트와 내부 인터럽트를 처리하기 위한 루틴을 함수로 구현한 것의 시작주소를 기록한다.
: idt_table 의 0~31까지의 32개의 엔트리를 CPU 의 트랩 핸들러를 위해 할당하고 그 외의 엔트리는 외부 인터럽트 핸들러를 위해 사용한다.
: 하지만 idt_table 은 이제 바뀌어 벡터테이블에서 +0x1000한 위치가 idt_table 로 활용된다.

인터럽트 동작과정
: CPU 에 내부적으로 내부 인터럽트가 발생되면 (devide by zero error, debug, nmi, int3 …) IDT 의 0~31엔트리까지 사용한다.
: PIC 라는 핀에는 각 하드웨어 장치들이 연결되어 있는데 이 PIC 의 한 핀과 CPU 가 연결되어있어 IDT 의 32번 엔트리부터 사용가능하다. System call 은 128엔트리에 저장 되어있다. 하드웨어 인터럽트 처리는 do_IRQ 가 실제로 처리하게된다.
: open()에 해당하는 시스템콜을 전달 할 때 해당하는 시스템 콜 번호를 5라고 가정.
1. ax 레지스터에 시스템 콜 번호인 5번이 저장된다.
2. 시스템 콜이어서 제어권이 kernel 로 넘어간다.
3. IDT 에 0x80(128)에 해당하는 엔트리에 가면 system_call 이 존재함.
4. ax 레지스터에서 저장되어 있던 5번을 보고, sys_call_table 의 5번을 찾으면 sys_~~을 구동하게 되고 open 이 동작하게 된다.

: 커널이 인터럽트를 받으면 즉시 인터럽트 핸들러를 호출할 수 있는 것은 아니고, context switching 을 해야한다.

인터럽트의 분류
: 인터럽트는 외부 인터럽트와 트랩(내부 인터럽트)로 나뉜다.
: 내부 인터럽트는 fault, trap, abort 로 나뉘게 된다.

fault - page, fault
fault 를 일으킨 명령어 주소를 eip 에 넣어 두었다가 해당 핸들러가 종료되고 나면 eip 에 저장되어 있는 주소부터 다시 수행을 시작한다.

trap - int, system call

trap 을 일으킨 명령어의 다음 주소를 eip 에 넣어두었다가 그 다음부터 다시 수행한다.

abort - devide by zero

이는 심각한 에러인 경우이므로 eip 값을 저장해야할 필요가 없으며 현재 태스크를 강제종료시키면 된다.

(2) 시스템 호출 처리 과정

시스템 호출 : 사용자 수준 응용 프로그램들에게 커널이 자신의 서비스를 제공하는 인터페이스.

ex) sys_fork(), sys_read(), sys_nice(), sys_~~~

1. 사용자가 fork() system call 을 요청
2. fork()라는 이름의 라이브러리가 호출
3. fork()에 할당된 고유 번호 2를 eax 레지스터에 넣고 0x80을 인자로 트랩을 건다.
4. 커널은 context swhiching 을 하며, 트랩 번호(0x80)에 대응되는 엔트리에 등록된 함수를 호출
5. eax 의 값을 인덱스로 sys_call_table 을 탐색하여 sys_fork()의 함수포인터를 얻음

Chapter 7 리눅스 모듈 프로그래밍

(1) 마이크로 커널

: 커널 모듈이란 디바이스 드라이버이다. 모듈은 탈 부착이 가능해야하고, 탈부착이 가능하므로 경령화에 유리하다.
: 리눅스는 대부분 모놀리식 방식을 사용하지만 디바이스 드라이버만은 마이크로 방식을 사용하게 된다. 디바이스 드라이버 만큼은 탈부착이 가능해야하기 때문이다. 이 두 방식을 사용하기에 리눅스는 하이브리드 방식이다.

모놀리식 커널과 마이크로 커널
모놀리식 커널 : 커널이 제공해야하는 태스크 관리, 메모리 관리, 파일시스템, 디바이스 드라이버, 통신 프로토콜 등의 기능이 단일한 커널 공간에 구현된 구조.
마이크로 커널 : 모놀리식 커널과 반대되는 개념으로 커널이 제공해야하는 기능이 분할되어 있다. 일반적으로 context switch 나 주소변환, system call, 디바이스 드라이버 등 하드웨어와 밀접하게 관련된 기능을 커널공간에 구현하고 나머지를 사용자 공간에 구현한다. 이리하여 커널의 크기를 작게해 휴대용 시스템의 운영체제로 사용가능하다. 많은 기능이 사용자 공간에서 서버 형태로 구현되기 때문에 클라이언트-서버 모델 같은 분산 환경에 잘 적용할 수 있다.

(2) 모듈 프로그래밍 무작정 따라 하기

~hello_module.c

```
#include <linux/kernel.h>
#include <linux/module.h>

int hello_module_init(void)
{
        printk(KERN_EMERG "Hello Module~! I'm in kernel₩n");
        //printk:커널에서 프린트, dmesg(디바이스메세지)를 쳐야 보인다.
            return 0;
}

void hello_module_cleanup(void)
{
```

```
        printk("<0>Bye Module~!₩n");
}

module_init(hello_module_init);
//module_init : insmod 할 때 동작하고,()안의 함수가 동작하는 시스템 콜
module_exit(hello_module_cleanup);
//module_exit : rmmod 할 때 동작하고, ()안의 함수가 동작하는 시스템 콜

MODULE_LICENSE("GPL"); //오픈소스 라이센스
```

~makefile

```
bj-m    := hello_module.o //커널이 모듈로 만드려는 것

KERNEL_DIR      := /lib/modules/$(shell uname -r)/build
//uname -r 을 치면 리눅스 버전이 뜬다, 위의 명령어는 KERNEL_DIR 에 리눅스 커널 디렉토리 위치를 저장.
PWD     := $(shell pwd)
//pwd 를 치면 현재 위치가 뜬다. PWD 에 현재 위치를 저장

default :
        $(MAKE) -C      $(KERNEL_DIR)   SUBDIRS=$(PWD)  modules
// C 옵션을 주면 해당 위치에서 작업하라는 의미.
// KERNEL_DR 에서 서브 디렉토리는 현재위치로 하고 모듈을 만들어라
clean :
        $(MAKE) -C      $(KERNEL_DIR)   SUBDIRS=$(PWD)  clean
```

~결과

make 라는 명령어를 입력하면 ko 파일 등이 생성되고, ismod hello_module.ko 를 입력한 뒤 dmesg 를 입력하면 Hello Module~! I'm in Kernel 이 출력된다.
Rmmod hello_module 을 입력한 뒤 dmesg 를 입력하게 되면 Bye Module~!이 모니터에 출력하게 된다.

확장자 ko 는 컴파일이 잘 되면 커널에서 구동할 수 있도록 커널 오브젝트로 바꿔주는 확장자.

리눅스버전

xeno@xeno-NH:~/kernel/linux-4.4$ uname -r
4.13.0-38-generic

리눅스 커널 디렉토리

```
xeno@xeno-NH:~/kernel/linux-4.4$ ls /lib/modules/4.13.0-38-generic/build
arch    crypto          firmware  init     Kconfig  Makefile         net       security  tools  virt
block   Documentation   fs        ipc      kernel   mm               samples   sound     ubuntu
zfs
certs   drivers         include   Kbuild   lib      Module.symvers   scripts   spl       usr
```

Chapter 8 디바이스 드라이버

(1) 디바이스 드라이버 일반

: 모니터, 키보드 마우스 같은 디바이스도 유닉스 시스템에서는 파일로 관리한다. file_operations 내에 함수 포인터들이 정의되어 open(), read(), write()등의 함수를 이용할 수 있다.

: inode 객체에는 i_rdev 에 주번호(장치의 종류를 나타내는 번호)와 부번호(징치의 개수)를 저장한다.

: inode 객체에 파일의 종류와 권한을 나타내는 i_mode 의 앞의 4bit 를 보고 module_init 할 때 files_operations 를 채워넣게 된다.

(2) 문자 디바이스 드라이버 구조

새로운 디바이스 드라이버를 구현할 때 필요한 작업 단계

1. 디바이스 드라이버의 이름과 주번호를 결정

2. 디바이스 드라이버가 제공하는 인터페이스를 위한 함수들을 구현해야한다.

3. 새로운 디바이스 드라이버를 커널에 등록

4. /dev 디렉터리에 디바이스 드라이버를 접근할 수 있는 장치파일을 생성해 주어야 한다.

```
#include <stdio.h>
#include <fcntl.h>

#define MAX_BUFFER     26
char buf_in[MAX_BUFFER];
char buf_out[MAX_BUFFER];

int main(void)
{
    int fd, i, c = 65;
    if((fd =open("/dev/mydevicefile", O_RDWR)) < 0){
        perror("open error");
        return -1;
    }
    for(i=0; i<MAX_BUFFER; i++){
        buf_out[i] = c++;
        buf_in[i] = 65;
    }

    for(i=0; i<MAX_BUFFER; i++){
        fprintf(stderr, "%c", buf_in[i]);
    }
```

```
    fprintf(stderr, "₩n");

    write(fd, buf_out, MAX_BUFFER);
    read(fd, buf_in, MAX_BUFFER);

    for(i=0; i<MAX_BUFFER; i++){
        fprintf(stderr, "%c", buf_in[i]);
    }
    fprintf(stderr, "₩n");

    close(fd);
    return 0;
}
```

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/uaccess.h>

#define DEVICE_NAME     "mydrv"
#define MYDRV_MAX_LENGTH    4096
#define MIN(a,b) (((a) < (b)) ? (a):(b))

struct class *myclass;
struct cdev *mycdev;
struct device *mydevice;
dev_t mydev;

static char* mydrv_data;
```

```
static int mydrv_read_offset, mydrv_write_offset;

static int mydrv_open(struct inode *inode, struct file *file)
{
    printk("%s₩n", __FUNCTION__);
    return 0;
}

static int mydrv_release(struct inode *inode, struct file *file)
{
    printk("%s₩n", __FUNCTION__);
    return 0;
}



static ssize_t mydrv_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{

    if((buf==NULL)||(count<0))
        return -EINVAL;
    if((mydrv_write_offset - mydrv_read_offset) <=0)
        return 0;
    count = MIN((mydrv_write_offset-mydrv_read_offset),count);
    if(copy_to_user(buf,mydrv_data+mydrv_read_offset, count))
        return -EFAULT;

    mydrv_read_offset +=count;
    return count;
}

static ssize_t mydrv_write(struct file* file, const char *buf, size_t count, loff_t *ppos)
{
```

```
    if((buf==NULL)||(count<0))
        return -EINVAL;
    if(count+mydrv_write_offset >= MYDRV_MAX_LENGTH)
    {    /*driver space is too small*/
        return 0;
    }
    if(copy_from_user(mydrv_data+mydrv_write_offset, buf, count))
        return -EFAULT;
    mydrv_write_offset +=count;
    return count;

}

struct file_operations mydrv_fops={
    .owner = THIS_MODULE,
    .read = mydrv_read,
    .write = mydrv_write,
    .open = mydrv_open,
    .release = mydrv_release,
};

int mydrv_init(void)
{
    if(alloc_chrdev_region(&mydev,0,1,DEVICE_NAME)<0){
        return -EBUSY;
    }

    myclass=class_create(THIS_MODULE, "mycharclass");
    if(IS_ERR(mydevice)){
        class_destroy(myclass);
        unregister_chrdev_region(mydev,1);
```

```
        return PTR_ERR(mydevice);
    }


    mycdev = cdev_alloc();
    mycdev->ops=&mydrv_fops;
    mycdev->owner=THIS_MODULE;

    if(cdev_add(mycdev,mydev,1)<0){
        device_destroy(myclass, mydev);
        class_destroy(myclass);
        unregister_chrdev_region(mydev,1);
        return -EBUSY;
    }


    mydrv_data = (char*)kmalloc(MYDRV_MAX_LENGTH* sizeof(char), GFP_KERNEL);
    mydrv_read_offset = mydrv_write_offset = 0;
    return 0;

}

void mydrv_cleanup(void)
{

    kfree(mydrv_data);
    cdev_del(mycdev);
    device_destroy(myclass, mydev);
    class_destroy(myclass);
    unregister_chrdev_region(mydev,1);
}
```

```
module_init(mydrv_init);
module_exit(mydrv_cleanup);
MODULE_LICENSE("GPL");
```