TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-04-12 (36 회차)

강사: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생: 정유경
ucong@naver.com

Ch6. 인터럽트와 트랩 그리고 시스템 호출

Ch8.디바이스 드라이버

Ch6. 인터럽트와 트랩 그리고 시스템 호출

인터럽트(Interrupt)
주변 장치와 커널이 통신하는 방식 중 하나
→ 주변 장치나 CPU 가 자신에게 발생한 사건을 리눅스 커널에게 알리는 매커니즘
인터럽트가 발생되면 운영체제는 인터럽트 핸들러 함수를 통해 처리

인터럽트는 원인에 따라 2 가지로 구분
1) 외부 인터럽트: 현재 수행 중인 태스크와 관련없는 주변 장치에서 발생, 비동기적, 하드웨어적
2) 트랩 : 현재 수행 중인 태스크와 관련, 동기적, 소프트웨어적 → ‘예외처리’ 라고도 함
ex. 0 으로 나누는 연산, 세그멘테이션 결함, 페이지 결함, 보호 결함, 시스템 호출

모든 CPU 는 인터럽트 발생시 PC(program counter, ARM), IP(instruction pointer, 인텔)레지스터의 값을 미리 정해진 특정 번지로 변경하도록 정해짐
각각의 특정 번지에는 해당 인터럽트를 처리하는 인터럽트 핸들러 가리킴

ex. ARM 계열 CPU 는 인터럽트 발생시 0x00000000+offset 번지로 점프함
offset 은 인터럽트의 종류에 따라 결정
(reset: 0, undefined instruction: 4 , Software interrupt: 8)

0x00000000 번지에는 인터럽트 핸들러로 점프하는 명령어만 기록
다른 위치에 인터럽트 핸들러 기록(4byte 보다는 클것이므로)

ARM CPU 의 인터럽트 벡터 테이블

```
__vectors_start:
        W(b)    vector_rst
        W(b)    vector_und
        W(ldr)  pc, __vectors_start + 0x1000
        W(b)    vector_pabt
        W(b)    vector_dabt
        W(b)    vector_addrexcptn
        W(b)    vector_irq
        W(b)    vector_fiq
```

*. 인텔 시스템 콜은 0x80 (숫자로 접근) , ARM 은 vectors_start+0x1000(주소번지로 접근)
vectors + 0x1000 에 해당하는 부분이 System Call Table

인터럽트 관리 기법
- 외부인터럽트, 트랩을 동일한 방식으로 처리
- 처리 루틴을 함수로 구현해 놓고 함수의 시작 주소를 ARM: verctors_start+0x1000 (인텔:idt_table)에 기록
IDT 의 0~31 은 트랩에 할당, 나머지는 (32 번 부터는)외부 인터럽트 핸들러를 위해 사용
- 외부 인터럽트 발생시키는 주변 장치는 하드웨어적으로 PIC(Programmable Interrupt Controller) 칩 각 핀에 연결,
 PIC 은 CPU 의 한 핀과 연결 (PIC 은 idt_table 의 32 번부터 사용 가능)
- 외부 인터럽트를 위한 번호는 별도로 관리→ irq_desc 테이블
- idt_table(128 번 제외) 32~255 번은 같은 인터럽트 핸들러 함수(common interrupt)→ do_IRQ()

시스템 콜 외의 인터럽트를 처리(common interrupt)하는 어셈블리 루틴

```
common_interrupt:
        ASM_CLAC
        addl    $-0x80, (%esp)
        SAVE_ALL
        TRACE_IRQS_OFF
        movl    %esp, %eax
        call    do_IRQ
        jmp     ret_from_intr
ENDPROC(common_interrupt)
```

- do_IRQ 는 외부 인터럽트 번호를 가지고 irq_desc 테이블을 인덱싱해서 외부 인터럽트 번호와 관련된 irq_desc_t 자료구조를 찾음 (irq_desc_t 에는 단일 인터럽트를 공유하는 action 리스트)
- 인터럽트 → 문맥 저장 → 인터럽트 핸들러 → 문맥 복원
(SAVE_ALL → do_IRQ() → ret_from_intr → RESTORE_ALL)

트랩
1) fault : fault 를 일으킨 명령어 주소를 eip 에 저장 (ex. page fault)
2) trap : trap 을 일으킨 명령어 다음 주소를 eip 에 저장 (ex. 시스템 콜)
3) abort: 저장하지 않음 , 현재 태스크 강제 종료 (ex. divide by zero)

핸들러 수행 후 리턴
1) trap(시스템 콜 제외) : ret_from_exception()
2) 외부 인터럽트 : ret_from_intr()
3) 시스템 콜: ret_from_sys_call()
4) fork, vfork, clone System Call : ret_from_fork()

시스템 호출
사용자 수준 응용에게 커널이 자신의 서비스를 제공하는 인터페이스, 커널로의 진입점
ex. sys_fork, sys_read, sys_nice - 현재 실행 태스크의 실행 우선순위 제어

시스템 호출 처리과정 (인텔 cpu 의 경우)
1) 사용자 수준 응용이 **fork()시스템 콜 요청** - 모든 시스템 호출은 각각 고유한 번호를 가지고 있음
2) 표준 C 라이브러리 함수인 __fork() 호출
→ 사용자 대신 트랩 요청(eax 에 fork 함수 고유번호 저장, 트랩번호 0x80 으로 트랩요청)
3) 트랩이 걸리면 제어가 커널로 넘어간다.
→ 문맥저장, **IDT 테이블에서 0x80 에 해당하는 함수(system call()) 호출**
4) 이 함수는 eax 의 값을 인덱스로 **ia32_sys_call_table** 을 탐색하여 sys_fork()함수의 포인터를 얻어옴
5) 커널에서 구현된 **sys_fork() 함수 호출**

시스템 콜에 할당된 고유번호

1. ARM

```
 *  This file is included thrice in entry-common.S
 */
/* 0 */         CALL(sys_restart_syscall)
                CALL(sys_exit)
                CALL(sys_fork)
                CALL(sys_read)
                CALL(sys_write)
/* 5 */         CALL(sys_open)
                CALL(sys_close)
                CALL(sys_ni_syscall)            /* was sys_waitpid */
                CALL(sys_creat)
                CALL(sys_link)
/* 10 */        CALL(sys_unlink)
                CALL(sys_execve)
                CALL(sys_chdir)
                CALL(OBSOLETE(sys_time))        /* used by libc4 */
                CALL(sys_mknod)
/* 15 */        CALL(sys_chmod)
                CALL(sys_lchown16)
                CALL(sys_ni_syscall)            /* was sys_break */
                CALL(sys_ni_syscall)            /* was sys_stat */
                CALL(sys_lseek)
/* 20 */        CALL(sys_getpid)
                CALL(sys_mount)
                CALL(OBSOLETE(sys_oldumount))   /* used by libc4 */
                CALL(sys_setuid16)
                CALL(sys_getuid16)
arch/arm/kernel/calls.S" 409L, 11144C
```

2. 인텔 x86 (64bit)

```
yukyoungch@jamiech-lenovo-ideapad-320s-13ikb: ~/kernel/linux-4.4/arch/x86
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
11      common  munmap                  sys_munmap
12      common  brk                     sys_brk
13      64      rt_sigaction            sys_rt_sigaction
14      common  rt_sigprocmask          sys_rt_sigprocmask
15      64      rt_sigreturn            stub_rt_sigreturn
16      64      ioctl                   sys_ioctl
17      common  pread64                 sys_pread64
18      common  pwrite64                sys_pwrite64
19      64      readv                   sys_readv
20      64      writev                  sys_writev
21      common  access                  sys_access
22      common  pipe                    sys_pipe
23      common  select                  sys_select
24      common  sched_yield             sys_sched_yield
25      common  mremap                  sys_mremap
"syscall_64.tbl" 373L, 12862C                 1,1           Top
```

Ch8.디바이스 드라이버

8.1 디바이스 드라이버 일반

- 정규파일 vs 장치파일

사용자 태스크 관점에서의 디바이스 드라이버

- 사용자 태스크가 접근하는 장치 파일은 VFS 가 제공하는 파일 객체
- 파일 객체에 행할 수 있는 연산 files_operations

-
```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
```

file_operations 구조체에 정의된 함수를 통해 장치파일에 접근할 때 호출할 함수를 정의하고 구현하는 것이 바로 **디바이스 드라이버**

디바이스 드라이버 번호

1) 주번호

- 시스템에 존재하는 여러개 디바이스 드라이버를 구분하기 위하여 디바이스 드라이버마다 고유한 번호를 정의
- 4096 개의 주 번호 지원
- inode > i_rdev 에 주번호와 부번호 저장
- 사용자 태스크가 특정 장치 파일에 접근하면, 장치파일과 연관된 디바이스 드라이버의 주 번호를 알 수 있음

즉 inode 구조체에 저장되어 있는 i_mode(파일의 종류와 권한), i_rdev 의 값을 이용하여 적절한 files_operations 구조체를 찾아서 호출 → module 적재시 files_operations 셋팅

2) 부번호

- 같은 장치가 여러 개 있을 때, 이를 구분하기 위한 번호
- 같은 장치이기 때문에 같은 디바이스 드라이버 사용하고 같은 주 번호를 가짐
- 동일한 드라이버에 의해 관리될 수 있는 장치를 구분하기 위해 사용

*. 리눅스는 /dev 디렉터리 밑에 mydrv 파일을 생성
→ 파일의 아이노드의 i_name 에 mydrv 를, i_rdev 에 주번호/부번호를 저장

개발자 입장에서 디바이스 드라이버

어떤 OS 에서도 구동되는 디바이스 드라이버를 만들어야 한다 (OS 가 없는 환경에서도 구동 가능해야)
→ 디바이스 드라이버 작성 시 '하드웨어와 밀접한 코드'와 '운영체제와 관련된 코드' 분리
→ 리눅스의 디바이스 드라이버는 특정 하드웨어 접근을 위한 디바이스 드라이버 코어 + 코어를 리눅스에서 사용할 수 있도록 하는 래퍼(wrapper)로 이루어짐

1) 디바이스 드라이버 코어 : 하드웨어 특성에 맞도록 작성
2) 래퍼 : 사용자 태스크 호출 함수와 코어 함수 연결 ←files_operations 를 모듈 init 할때 대체하게 된다.

디바이스 드라이버를 커널이 관리하는 방법

1) 문자형과 블록형 디바이스 드라이버를 위한 자료구조(cdev_map, bdev_map)
→ 각각 cdev 구조체, gendisk 구조체를 255 개씩 저장할 수 있는 배열 형태로 구현되어 있음
2) cdev > ops 필드에는 문자형 디바이스 드라이버가 제공하는 files_operations 구조체 저장,
gendisk > fops 필드에는 블록 디바이스 드라이버가 제공하는 block_device_operations 구조체 저장
→ 사용자 태스크가 장치파일에 접근하는 경우, 장치파일의 inode 구조체에 저장된 i_mode, i_rdev 필드의 값을 이용하여 적절한 파일 오퍼레이션 구조체를 찾아서 호출함으로써 디바이스 드라이버가 제공하는 함수를 사용

8.2 문자 디바이스 드라이버 구조

```
/*디바이스드라이버를 위한 Makefile*/
obj-m              := chr_test.o

KERNEL_DIR    :=/lib/modules/$(shell uname -r)/build
```

```
PWD                 := $(shell pwd)

default :
        $(MAKE)         -C      $(KERNEL_DIR)           SUBDIRS=$(PWD)      modules
clean :
        $(MAKE) -C      $(KERNEL_DIR)           SUBDIRS=$(PWD)      clean
```

```
/*chr_test.c*/
#include <linux/kernel.h> // 커널공간에서
#include <linux/module.h> // 모듈로써 동작
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "mydrv" // 디바이스 드라이버의 이름
#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;

/*mydrv_open, release 주번호를 확인한다*/
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) // → kmalloc 을 해주어야 한다.
	{	/*driver space is too small*/
		return 0;
	}
	if(copy_from_user(mydrv_data+mydrv_write_offset, buf, count)) // 유저영역 정보를 커널영역으로
복사
		return -EFAULT;
	mydrv_write_offset +=count; // offset +26 (26 개 썼다)
	return count; // 실제 write 한  byte

}

struct file_operations mydrv_fops={ //files_operations 자료구조
	.owner = THIS_MODULE, // 지금 동작하는 이 모듈에 제어권이 있다
	.read = mydrv_read,
	.write = mydrv_write,
	.open = mydrv_open, // files_operations 의 opend 을 대체할 mydrv_open
	.release = mydrv_release,
};

int mydrv_init(void) // 모듈의 초기화 함수
{ /* alloc_chrdev_region 함수를 호출하여  주번호를 동적할당받음 → 캐릭터디바이스 등록
class_create,  device_create 는 캐릭터 디바이스 등록시 형식적으로 따라오는 루틴, 디바이스 드라이버의 종류가
많을때 그룹화 하는데에 사용한다.*/
	if(alloc_chrdev_region(&mydev,0,1,DEVICE_NAME)<0){
		return -EBUSY;
	}
	myclass=class_create(THIS_MODULE, "mycharclass"); // 대분류
	 if(IS_ERR(myclass)){
		unregister_chrdev_region(mydev,1);
		return PTR_ERR(myclass);
	}
	mydevice=device_create(myclass, NULL, mydev, NULL, “mydevicefile”); // 소분류
	if(IS_ERR(mydevice)){
		class_destroy(myclass);
```

```
                unregister_chrdev_region(mydev,1);
                return PTR_ERR(mydevice);
        }

//cdev_alloc :217p
        mycdev = cdev_alloc(); // cdev 구조체를 할당받음
        mycdev→ops=&mydrv_fops; // 파일 연산구조를 등록 struct files_operations → mydrv_fops
        mycdev->owner=THIS_MODULE;
//cdev_add:217p
        if(cdev_add(mycdev,mydev,1)<0){ // 초기화된  cdev 구조체를  cdev_add()를 통해 커널(cdev_map)
에 등록
                device_destroy(myclass, mydev);
                class_destroy(myclass);
                unregister_chrdev_region(mydev,1);
                return -EBUSY;
        }

/*등록성공하면, 드라이버가 사용할 mydrv_data 라는 커널공간을 kmalloc 으로 할당
커널도 프로그램이므로 스택이 필요하다
커널 스택 8KB(thread_union > kernel_stack) , 현재 구동중인 task 찾기 : threaed_union >  thread_info >
task_struct > current: thread_info 의 current 가 가리키는 task*/
        mydrv_data = (char*)kmalloc(MYDRV_MAX_LENGTH* sizeof(char), GFP_KERNEL);
/* GFP_KERNEL : 할당될때까지 블록킹 (수행완료후 다음작업으로 간다고 명시)*/
        mydrv_read_offset = mydrv_write_offset = 0; // 공간의 첫번째 위치를 가리키도록
        return 0;

} // 드라이버 insert 완료

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); // 모듈 초기화 함수
module_exit(mydrv_cleanup); // 모듈 제거시 호출될 함수
MODULE_LICENSE("GPL");
```

```
/*user 영역의 프로그램 : mydevfile.c */
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define MAX_BUFFER  26
```

```
char buf_in[MAX_BUFFER];
char buf_out[MAX_BUFFER];

int main(void)
{
	int fd,i,c=65; // A
	if((fd=open("/dev/mydevicfile", O_RDWR))<0){ // fd=3 //dev/mydevicefile 이 장치파일로 생성된 상
태
		perror("open error");
		return -1;
	}

	for(i=0; i<MAX_BUFFER;i++){
		buf_out[i] = c++; // A~Z
		buf_in[i] = 65; //A~A
	}

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

	write(fd, buf_out, MAX_BUFFER); //218p mydrv_write()동작
	read(fd, buf_in, MAX_BUFFER); // mydrv_read()


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

*. open 할때 무슨일이 일어날까
task_struct > files_struct > file > path , files_operations
path > dentry > inode > superblock 에서 루트파일 위치 찾는다 (2 번)
dev/mydevicefile 검색, inode>imode 보고 일반파일이 아닌 캐릭터 디바이스임을 알아낸다
cdev 등록되어 있는지 살펴보고 insmod 하면서 files_operations 를 대체시킨다.
task_struct > … > files_operations 바뀐다.