Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – hoseong Lee(이호성)

hslee00001@naver.com


CONTENT

chapter 6. 인터럽트와 트랩 그리고 시스템

인터럽트란 주변장치와 커널이 통신하는 방식중 하나이다. 즉, 주변 장치나 cpu가 자신에게 발생한 사건을 리눅스 커널에게 알리는 매커니즘이다.

외부인터럽트:
내부인터럽트:

cpu는 인터럽트가 발생하면 pc(program counter : arm),ip(instruction pointer: intel) 레지 값을 미리 정해진 특정 번지로 변경하도록 정해져 있다.
리셋 인터럽트가 발생하면 0번지로 가게됨.
ARM CPU 인 경우 인터럽트가 발생하면 0x00000000+offset 번지로 점프한다.

하드웨어 인터럽트든 소프트웨어 인터럽트든 모두 IDT(Interrupt Descriptor Table) or IVT(Interrupt Vector Table) 를 본다.
이제 리눅스가 이들을 관리하는 것을 알아 보자.
리눅스는 인터럽트와 트랩을 처리하기 위한 루틴을 함수로 구현해 놓은 뒤, 각 함수의 시작 주소를 리눅스의 IDT인 idt_table이라는 이름의 배열에 기록해 둔다.
그렇담 코드를 까보자 vi -t idt_table
vi -t arch/arm/common/ kernel/traps.c
→ grep -rn "ivt_table" ./ (안잡혔음)

→ grep -rn "ivt" ./ | grep arm (이것도 잘못짚음)

→ grep -rn "idt_desc"

2 F d __vectors_start arch/arm/kernel/vmlinux.lds.S
2 F l __vectors_start arch/arm/kernel/entry-armv.S

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.
내부 인터럽트에는 0으로 나누는 연산(device by zero), 세그멘테이션 결함, 펭지 결함, 보호결함, 시스템 콜 등이 있다.

인터럽트 디스크립터 테이블을 찾는 경우 → 시스템콜이 호출이되면 제어권이 커널이 되고 커널이 유저한테 다시 제어권을 넘겨주는 과정을 봐야함.

제어권을 넘겨주는 데 초기화를 해야함?? 아직..
idt는 arm에서 __vectors_start로 바뀜
→ init_early
grep -rn "init_early" ./ | grep arm

```
./arch/arm/mach-omap2/board-generic.c:47:       .init_early     = omap2420_init_early,
./arch/arm/mach-omap2/board-generic.c:64:       .init_early     = omap2430_init_early,
./arch/arm/mach-omap2/board-generic.c:82:       .init_early     = omap3430_init_early,
./arch/arm/mach-omap2/board-generic.c:100:      .init_early     = omap3430_init_early,
./arch/arm/mach-omap2/board-generic.c:117:      .init_early     = omap3630_init_early,
./arch/arm/mach-omap2/board-generic.c:134:      .init_early     = omap3430_init_early,
./arch/arm/mach-omap2/board-generic.c:150:      .init_early     = am35xx_init_early,
./arch/arm/mach-omap2/board-generic.c:169:      .init_early     = ti814x_init_early,
./arch/arm/mach-omap2/board-generic.c:186:      .init_early     = ti816x_init_early,
./arch/arm/mach-omap2/board-generic.c:204:      .init_early     = am33xx_init_early,
./arch/arm/mach-omap2/board-generic.c:228:      .init_early     = omap4430_init_early,
./arch/arm/mach-omap2/board-generic.c:253:      .init_early     = omap5_init_early,
./arch/arm/mach-omap2/board-generic.c:275:      .init_early     = am43xx_init_early,
./arch/arm/mach-omap2/board-generic.c:301:      .init_early     = dra7xx_init_early,
./arch/arm/mach-omap2/board-generic.c:323:      .init_early     = dra7xx_init_early,
```

```
__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
```

이곳이 바로 소프트웨어 시스템콜 테이블 위치이다
나머지는 51분
여기까지 167p 내용

p168, -1 ,start

만약 open() 함수를 사용한다. 동작 과정은

```
asm volatile("pushfl\n\t"                /* save    flags */     \
             "pushl %%ebp\n\t"           /* save    EBP   */     \
             "movl %%esp,%[prev_sp]\n\t"         /* save    ESP   */ \
             "movl %[next_sp],%%esp\n\t"         /* restore ESP   */ \
             "movl $1f,%[prev_ip]\n\t"   /* save    EIP   */     \
             "pushl %[next_ip]\n\t"      /* restore EIP   */     \
             __switch_canary                                     \
             "jmp __switch_to\n"         /* regparm call  */     \
             "1:\t"                                              \
             "popl %%ebp\n\t"            /* restore EBP   */     \
             "popfl\n"                   /* restore flags */     \
                                                                 \
```

swich_to
30
11분

21번 12분 30초

switch_to

thread_struct → 트랩 넘버가 저장되어있다.

시스템콜도 트랩이기 때문이다

smp_processor 메모리를 공유한다.
vi -t switch_to

grep -rn "SAVE_ALL" ./ |grep x86

시스템 호출과정 : 사용자 수준 응용 프로그램들에게 커널이 자신의 서비스를 제공하는 인터페이스.
(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 2 를 eax 레지스터에 넣고 0x80 을 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

3번째, 21분
커널 밑에 fork.c
__fork → __libc_fork

“movl %1 ,%%eax\n=t”→ %1은 맨처음 온 변수라는 의미

**********fork한테 일어나는 일을 기술하세요

ls /lib/modules/4.13.0-38-generic/build

sudo insmod hello_module.ko
dmesg
sudo rmmod hello_module.ko
dmesg

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
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>
#include <linux/moduleparam.h>

int hello_module_init(void){
    printk(KERN_EMERG "Hello Module~! I'm in
Kernel ₩n");
    return 0;
}

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

hello_module_mode.c

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);
MODULE_INFO(name, KBUILD_MODNAME);

__visible struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
        .name = KBUILD_MODNAME,
        .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
        .exit = cleanup_module,
#endif
        .arch = MODULE_ARCH_INIT,
};
```

module_init(hello_module_init);
module_exit(hello_module_cleanup);
 	
MODULE_LICENSE("GPL");
	static const char __module_depends[]
	__used
	__attribute__((section(".modinfo"))) =
	"depends=";
	
	MODULE_INFO(srcversion, "FCDCA0BFD3E910266B18A22");

(3) 시스템 호출 hooking

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

int hello_module_init(void){
   printk(KERN_EMERG "Hello Module~! I'm in Kernel \n");
   return 0;
}

void hello_module_cleanup(void){
   printk("<0>Bye Module~! \n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);

MODULE_LICENSE("GPL");


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

```
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);
MODULE_INFO(name, KBUILD_MODNAME);

__visible struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
        .name = KBUILD_MODNAME,
        .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
        .exit = cleanup_module,
#endif
        .arch = MODULE_ARCH_INIT,
};

static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";


MODULE_INFO(srcversion, "FCDCA0BFD3E910266B18A22");
```

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 시에 필요한 작업 단계
디바이스 드라이버의 이름과 주번호를 결정하고, 디바이스가 제공하는 인터페이스를 위한 함수들을 구현한다. 그리고 새로운 디바이스 드라이버를 커널에 등록하고 /dev 디렉터리에 디바이스 드라이버를 접근할 수 있는 장치파일을 생성해 주어야 한다.

- mydrv_test.c

```
#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;
   }
```

-chr_test.c

```
#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;
```

```
    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;
}
```

```
static int mydrv_open(struct inode *inode, struct file *file)
{
    printk("%s\n",__FUNCTION__); // 하나일까?
    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 spase 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(myclass)){
        unregister_chrdev_region(mydev,1);
        return PTR_ERR(myclass);
    }

    mydevice = device_create(myclass, NULL, mydev, NULL, "mydevicefile");
    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");
```

- chr_test.mod.c

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);
MODULE_INFO(name, KBUILD_MODNAME);
```

```
__visible struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
        .name = KBUILD_MODNAME,
        .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
        .exit = cleanup_module,
#endif
        .arch = MODULE_ARCH_INIT,
};

static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";


MODULE_INFO(srcversion, "9781C05F29A3379E15F89F4");
```

chapter9 . 네트워킹

(1) 계층구조

: NIC(Network Interface Card)마다 지정되어 있는 IP 주소를 통해 누구와 통신을 할 지 지정한다. 한 컴퓨터는 복수개의 NIC 가 장착 가능하고 IP 주소를 복수 개 가질 수 있다. 여러 태스크가 한 개의 NIC 를 통해 통신할 수 있으니 각각의 사용자 태스크를 구분하기 위한 포트(port)번호가 필요하다.
프로토콜
→ 표준 프로토콜인 OSI 프로토콜은 7 층을 가지지만 실제로는 4 계층만 구현 가능하다.
→ 가장 상위층은 사용자에게 소켓 인터페이스를 제공하는 BSD 소켓 층이다. 사용자 태스크에게 소켓이라는 객체를 제공한다. 대표적인 소켓 인터페이스는 socket(), bind(), connect(), accept(), listen(), send(), recv()등이 있다.
→ IP 계층 아래에는 PPP, SLIP 또는 이더넷과 같은 데이터 링크 층이 존재한다. 디바이스는 net_device 라는 자료구조에서 자신의 정보 및 기능을 저장.
→ IP 계층 아래에는 PPP, SLIP 또는 이더넷과 같은 데이터 링크 층이 존재한다. 디바이스는 net_device 라는 자료구조에서 자신의 정보 및기능을 저장하여 IP 층에 제공하게 된다.

→ IP 주소를 이더넷주소(MAC 주소)로 변환하기 위해 ARP(Address Resolutiojnj Protocol)을 사용하고, 이더넷 주소를 IP 주소로 변환하기 위해 RARP(Reverse Address Resolution protocol)를 사용한다.
→ ftp 는 21, telnet 은 23, www 는 80 의 포트를 사용한다.

(2) 주요 커널 내부 구조

: 통신 프로토콜은 계층구조를 갖는다. 상위 층은 여러 하위 층과 연동할 수 있다. 통신 프로토콜 층을 내려오면서 다양한 곳으로 제어 흐름이 분기할 수 있다.

→ 리눅스는 소켓을 생성하면 fd 가 생성되고 네트워크도 결국 파일로 전달하게 된다. 즉, 리눅스는 제어 흐름이 다양한 곳으로 분기할 수 있는 상황을 위해 층 사이에서 제어가 전달될 때 자료구조를 이용한 간접 호출 방법으로 통신 프로토콜을 구현한다. 파일연산(file_operations) 자료구조를 이용해 디바이스 드라이버 함수를 호출한 것과 유사하다. : sk_buff 에 ethernet_frame 에 대한 정보들을 채워 넣는다. 위의 데이터 encapsulation.

부록 A 리눅스와 가상화 그리고 XEN

가상화 기법 : OS 도 결국 프로세스이고, 중첩페이징을 이용하여 윈도우 커널에서 페이지를 받아와 3 번(10/10/12bit)를 페이지하고 리눅스 내에서도 또 다시 3 번 페이지를 하여 총 9 번하게 되면 윈도우에서 리눅스를 띄울 수 있음. 하지만 속도가 느리다는 단점이 있다.

가상화 기술의 장점
1. 가상화는 서버의 이용률(utilization)을 높이고 관리 부하를 줄일 수 있다.
2. 각 사용자의 수행 환경을 다른 환경들로부터 고립시킬 수 있어 보안수준을 높일 수 있다.
3. 여러 물리자원들을 단일한 가상자원으로 집합할 수 있다.
4. 시스템의 이동성을 증가시킨다.
5. 새로운 시스템이나 아직 개발되지 않는 하드웨어를 모의실험(emulation)하는 기능을 제공한다.