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

#오답노트

강사 : Innova Lee(이 상훈)

학생 : 김 시윤

1.배운내용 복습.

QUIZ 2번

1.파이프 통신을 구현하고 c type.c라고 입력할 경우 현재 위치의 디렉토리에 type.c 파일을 생성하도록 프로그래밍하시오.

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

int flag;

char *check_text(char *text)
{
	int i;
	static char filename[1024];
	int text_len = strlen(text);
	if(text[0] != 'c' && text[1] != ' ')
		return NULL;
	if(text[text_len - 1] != 'c' && text[text_len - 2] != '.')
		return NULL;
	for(i = 2; i < text_len - 2; i++)
	{
		if(text[i] == ' ' || text[i] == '\t')
			return NULL;
		filename[i - 2] = text[i];
	}
	strcat(filename, ".c");
	return filename;
}
```

```
int main(void)
{
	int fo;
	int fd, ret;
	char buf[1024];
	char *string = NULL;
	fd = open("myfifo", O_RDWR);		//피포통신의 매개체역할
	fcntl(0, F_SETFL, O_NONBLOCK);	//플래그를 난블럭으로 설정?
	fcntl(fd, F_SETFL, O_NONBLOCK);
	for(;;)
	{
		if((ret = read(0, buf, sizeof(buf))) > 0)	//입력데이터를 버프만큼 읽
는다
		{
			buf[ret - 1] = 0;  //엔터지우기
			printf("Keyboard Input : [%s]\n", buf);
			string = check_text(buf);   //함수에서 받아온 text를 저장
			printf("String : %s\n", string);
		}
		if((ret = read(fd, buf, sizeof(buf))) > 0)
		{
			buf[ret - 1] = 0;
			printf("Pipe Input : [%s]\n", buf);
			string = check_text(buf);
			printf("String : %s\n", string);
		}
		fo = open(string, O_WRONLY | O_CREAT | O_TRUNC, 0644);
		close(fo);
	}
	close(fd);
	return 0;
}
//이렇게 하면 .c 를 안붙여도 통신이 되지 않는지 ..
실행방법을 까먹어 실행해보지 못하였다..
```

4.Unix 계열의 모든 OS는 모든 것을 무엇으로 관리하는가 ?
파일
(책에 OS는 커널로 관리한다 라고 써있는데 커널도 프로그램이니 파일 개념이라고 볼수 있을까여?)

5.

* 사용자 임의대로 재구성이 가능하다.

리눅스 커널이 F/OSS 에 속하여 있으므로 소스 코드를 원하는 대로 수정할 수 있다.
그러나 License 부분을 조심해야 한다.

* 열약한 환경에서도 HW 자원을 적절히 활용하여 동작한다.

리눅스 커널이 가볍고 좋을뿐만 아니라 소스가 공개되어 있어
다양한 분야의 사람들이 지속적으로 개발하여 어떠한 열약한 환경에서도 잘 동작한다.

* 커널의 크기가 작다.

최적화가 잘 되어 있다는 뜻

* 완벽한 멀티유저, 멀티태스킹 시스템

리눅스는 RT Scheduling 방식을 채택하여 Multi-Tasking 을 효율적으로 잘 해낸다.

* 뛰어난 안정성

전 세계의 많은 개발자들이 지속적으로 유지보수하여 안정성이 뛰어남

* 빠른 업그레이드

위와 같음

* 강력한 네트워크 지원

TCP/IP Stack 이 원채 잘 되어 있다보니 Router 및 Switch 등의 장비에서 사용함

* 풍부한 소프트웨어

GNU(GNU is Not Unix) 정신에 입각하여 많은 Tool 들이 개발되어 있다.

11번
리눅스 커널 소스에 보면 current라는 것이 보인다. 이것이 무엇을 의미하는 것인지 적으시오.

#if 0

먼저 vi -t current 로 검색하면
아래 헤더 파일에 x86 에 한하여 관련 정보를 확인할 수 있다.

arch/x86/include/asm/current.h

여기서 get_current() 매크로를 살펴보면 ARM 의 경우에는 아래 파일에

include/asm-generic/current.h

thread_info->task 를 확인할 수 있다.

x86 의 경우에는 동일한 파일 위치에서
this_cpu_read_stable() 함수에 의해 동작한다.

이 부분을 살펴보면 아래 파일

arch/x86/include/asm/percpu.h 에서

percpu_stable_op("mov", var) 매크로를 통해 관리됨을 볼 수 있다.
Intel 방식의 특유의 세그먼트 레지스터를 사용하여
관리하는 것을 볼 수 있는 부분이다.

#endif

즉 태스크를 관리한다.

21번
ZONE_HIGHMEM 에 대해 아는대로 기술하시오.

#if 0

커널 공간 1 GB 의 가상 메모리가 모든 물리 메모리를 커버해야 한다.
32 비트 시스템의 경우에는 1 : 3 이라 커널 공간이 1 GB 밖에 없다.
메모리는 8 GB, 64 GB 가 달려 있으니 이것을 커버하기 위한 기법이 필요하다.

기본적으로 ZONE_NORMAL 이 특정 구간까지는 1 : 1 로 맵핑한다.
이후 ZONE_NORMAL 이 처리하지 못하는
모든 메모리 영역을 ZONE_HIGHMEM 이 처리하게 된다.

user 에서는 어떠한 공간이든 각 task 별로 매핑을 하여 사용하지만
kernel 에서는 최대한 빠른 속도를 얻기 위해 ZONE_DMA(DMA32) 및
ZONE_NORMAL 에서는 물리 메모리와 가상 메모리를 미리 1:1로 매핑하여 사용을 한다.

그러나 물리 메모리가 커널로의 1:1 매핑을 허용하는 영역 크기를 초과하는 경우
이 영역을 CONFIG_ZONE_HIGHMEM 영역으로 구성하여
커널에서 사용할 때에는 필요할 때마다 매핑하여 사용한다.

32bit 시스템에서는 1:1 매핑이 일부만 가능하기 때문에
ZONE_NORMAL을 초과하는 메모리가 이 영역을 사용한다.

64bit 시스템에서는 모든 물리 메모리가 1:1 매핑이 가능하므로
ZONE_HIGHMEM을 사용하지 않는다.

#endif

문제를 보지 못했다...

ZONE_HIGHMEN은 메모리 영역을 최대한 활용하기 위한 방식.
1:1 맵핑시 프로세스의 용량이 매우 광대해 1:1 맵핑이 불가능 할 경우
사용한다. 커널에서 어떤 데이터가 필요하게 되면 ZONE_HIGHMEN에서 가져와 사용한다.

22번 물리 메모리의 최소 단위를 무엇이라고 하며 크기가 얼마인가 ? 그리고 이러한 개념을 SW 적으로 구현한 구조체의 이름은 무엇인가 ?

물리메모리의 최소단위 페이지프레임 4KB이다 . task_struct → mm_struct안에 있다.
이렇게 적었었는데
mm_struct -> page 구조체에 들어있다.

25번
Kernel은 Memory Management를 수행하기 위해 VM(가상 메모리)를 관리한다. 가상 메모리의 구조인 Stack, Heap, Data, Text는 어디에 기록되는가 ? (Task 구조체의 어떠한 구조체가 이를 표현하는지 기술하시오)

task_struct -> mm_struct -> vm_struct 인줄 알았는데
task_struct -> mm_struct -> start_code ,end_code 에 의해 관리된다..

27번 프로그램을 실행한다고 하면 fork(), execve()의 콤보로 이어지게 된다. 이 때 실제 gcc *.c로 컴파일한 a.out을 ./a.out을 통해 실행한다고 가정한다. 실행을 한다고 하면 a.out File의 Text 영역에 해당하는 부분을 읽어야 한다. 실제 Kernel은 무엇을 읽고 이 영역들을 적절한 값으로 채워주는가 ?

ELF Header 와 **Program Headers** 를 읽고 값을 적절하게 채운다.
-> 커널 내부 소스코드 인줄 알았는데 아니였다 ..

29. VM(가상 메모리)와 PM(물리 메모리)를 관리하는데 있어 VM을 PM으로 변환시키는 Paging Mechanism에 대해 Kernel에 기반하여 서술하시오.

mm_struct에 pgd라는 필드가 있다.
Page Directory를 의미하는 것으로 pgd -> pte -> page로 3단계 Paging을 수행한다.
각각 10bit, 10bit, 12bit로 VM의 주소를 쪼개서 Indexing을 한다.

page_table을 통해 접근하는 방법도 페이징이라 볼수 있는건지?..

30 MMU(Memory Management Unit)의 주요 핵심 기능을 모두 적고 간략히 설명하시오.

#if 0

1. HAT(HW Address Translation) 는
   가상 메모리 공간을 실제 물리 메모리 공간으로 변환한다.
2. TLB(Translation Lookaside Buffer) 는
   가상 메모리 주소와 대응되는 물리 메모리 주소를 Caching한다.

#endif

38. 35번에서 분류한 녀석들의 특징에 대해 기술하시오.

문제를 이해하지 못해 적지 못했다.

Fault의 경우 Page Fault가 대표적이므로 발생시
현재 진행중인 주소를 복귀주소로 저장하고 Fault에 대한 처리를 진행하고
다시 돌아와서 Fault가 났던 부분을 다시 한 번 더 수행한다.

Trap의 경우 System Call이 대표적이므로 발생시
현재 진행중인 바로 아래 주소를 복귀주소로 저장하고
System Call에 대한 수행을 처리한 이후
System Call 바로 아래 주소부터 실행을 시작한다
(함수 호출의 복귀 주소와 비슷한 형태)

Abort의 경우 심각한 오류에 해당하므로 그냥 종료한다.

39. 예로 모니터 3 개를 쓰는 경우 양쪽에 모두 인터럽트를 공유해야 한다. Linux Kernel에서는 어떠한 방법을 통해 이들을 공유하는가 ?

외부 인터럽트의 경우 32 ~ 255까지의 번호를 사용한다.
여기서 128(0x80)에 해당하는 System Call만은 제외한다.
idt_table에서 128을 제외한 32 ~ 255 사이의 번호가 들어오면 실제 HW Device다.
여기서 같은 종류의 HW Device가 들어올 수 있는데
그들에 대해서 Interrupt를 공유하게 하기 위해
irq_desc라는 Table을 추가로 두고 active라는 것으로 Interrupt를 공유하게끔한다.

주번후 부번호에 의해 관리된다.

40. System Call 호출시 Kernel에서 실제 System Call을 처리하기 위해 Indexing을 수행하여 적절한 함수가 호출되도록 주소값을 저장해놓고 있다. 이 구조체의 이름을 적으시오.

ARM만 적어놨었다..

Intel 의 경우에 sys_call_table 에 의해 관리된다

41. 38에서 User Space에서 System Call 번호를 전달한다. Intel Machine에서는 이를 어디에 저장하는가 ? 또한 ARM Machine에서는 이를 어디에 저장하는가 ?

이건 intel만 적어놨었다.

ARM의 경우 ARM 의 경우에는 r7 레지스터에 해당한다

52 예로 간단한 Character Device Driver를 작성했다고 가정해본다. 그리고 insmod를 통해 Driver를 Kernel내에 삽입했으며 mknod를 이용하여 /dev/장치파일을 생성하였다. 그리고 이에 적절한 User 프로그램을 동작시켰다. 이 Driver가 실제 Kernel에서 어떻게 관리되고 사용되는지 내부 Mechanism을 기술하시오.

Device Driver 에서 class_create, device_create 를 이용해서
Character Device 인 /dev/장치파일을 만든다.
그리고 Character Device Driver를 등록하기 위해서는
register_chrdev()가 동작해야 한다.

동적으로 할당할 경우에는 alloc_chrdev_region() 이 동작한다.
이때 file_operations를 Wrapping할 구조체를 전달하고
Major Number와 장치명을 전달한다.

chrdevs 배열에 Major Number에 해당하는 Index에
file_operations를 Wrapping한 구조체를 저장해둔다.

그리고 이후에 실제 User 쪽에서 생성된 장치가 open 되면
그때는 Major Number에 해당하는 장치의 File Descriptor가 생성되었으므로
이 fd_array 즉, file 구조체에서 i_mode를 보면
Character Device임을 알 수 있고
i_rdev를 보고 Major Number와 Minor Number를 파악할 수 있다.

그리고 chrdevs에 등록한 Major Number와 같음을 확인하고
이 fd_array에 한해서는 open, read, write, lseek, close등의
file_operations 구조체 내에 있는 함수 포인터들을
앞서 Wrapping한 구조체로 대체한다.

그러면 User에서 read()등을 호출할 경우 우리가 알고 있는 read가 아닌
Device Driver 작성시 새로 만든 우리가 Wrapping한 함수가 동작하게 된다.

54. Character Device Driver를 아래와 같이 동작하게 만드시오. read(fd, buf, 10)을 동작시킬 경우 1 ~ 10 까지의 덧셈을 반환하도록 한다. write(fd, buf, 5)를 동작시킬 경우 1 ~ 5 곱셈을 반환하도록 한다. close(fd)를 수행하면 Kernel 내에서 "Finalize Device Driver"가 출력되게 하라!

```
#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 <asm/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 int *write_ret;
static int *read_ret;
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 int mydrv_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{       //실질적 로직
#if 0
        if((buf == NULL) || (count < 0))
                return -EINVAL;
        if(mydrv_write_offset - mydrv_read_offset <= 0)
                return 0;
        count = MIN((mydrv_write_offset - mydrv_write_offset), count);
        if(copy_to_user(buf, mydrv_data + mydrv_read_offset, count))
                return -EFAULT;
        mydrv_read_offset += count;
#endif
        int i;

        read_ret = (int *)kmalloc(sizeof(int), GFP_KERNEL);
        read_ret[0] = 1;

        for(i = 1; i <= count; i++)
                read_ret[0] *= i;

        return read_ret[0];
}

static ssize_t mydrv_write(struct file *file, const char *buf, size_t count,
loff_t *ppos) //실질적 로직
{
#if 0
        if((buf == NULL) || (count < 0))
                return -EINVAL;
        if(count + mydrv_write_offset >= MYDRV_MAX_LENGTH)
                return 0;
        if(copy_from_user(mydrv_data + mydrv_write_offset, buf, count))
                return -EFAULT;
        mydrv_write_offset += count;
#endif
        int i;

        write_ret = (int *)kmalloc(sizeof(int), GFP_KERNEL);      //데이터 공
간할당
        write_ret[0] = 0;         //초기값 (쓰레기값 방지)
```

```
	for(i = 1; i <= count; i++)
		write_ret[0] += i;		//write(x,x,count)

	return write_ret[0];
}

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);
//어디부분에 로직을 추가해야할지 몰랐는데 이제 알거같다.
```

55. OoO(Out-of-Order)인 비순차 실행에 대해 기술하라.

데이터 의존성이 존재하게되면 어쩔 수 없이 Stall이 발생하게 된다.
이러한 Stall을 최소화하기 위해 앞서 실행했던 코드와
의존성이 없는 코드를 찾아서
아래의 의존성이 있는 코드 위로 끌어올려서 실행하는 방식이다.

58 Compiler의 Instruction Scheduling은 Run-Time이 아닌 Compile-Time에 결정된다. 고로 이를 Static Instruction Scheduling이라 할 수 있다. Intel 계열의 Machine에서는 Compiler의 힘을 빌리지 않고도 어느저도의 Instruction Scheduling을 HW의 힘만으로 수행할 수 있다. 이러한 것을 무엇이라 부르는가 ?

Dynamic Instruction Scheduling

60. CPU 들은 각각 저마다 이것을 가지고 있다. Compiler 개발자들은 이것을 고려해서 Compiler를 만들어야 한다. 또한 HW 입장에서도 이것을 고려해서 설계를 해야 한다. 여기서 말하는 이것이란 무엇인가 ?

ISA(Instruction Set Architecture) : 명령어 집합

62. 그동안 많은 것을 배웠을 것이다. 최종적으로 Kernel Map을 그려보도록 한다. (Networking 부분은 생략해도 좋다) 예로는 다음을 생각해보도록 한다. 여러분이 좋아하는 게임을 더블 클릭하여 실행한다고 할 때 그 과정 자체를 Linux Kernel 에 입각하여 기술하도록 하시오. (그림과 설명을 같이 넣어서 해석하도록 한다) 소스 코드도 함께 추가하여 설명해야 한다.

task_struct
mm_struct
vm_area_struct
signal_struct
sigpending
sighand_struct
rt_rq
dl_rq
cfs_rq
files_struct
file
file_operations
path
dentry
inode
super_block
sys_call_table
idt_table
do_IRQ()
Buddy, Slab 할당자
SYSCALL_DEFINE0(fork) 등등이 다루어지면 됨

먼저 마우스를 움직여서 더블 클릭한다는 것은 HW 신호에 해당하므로
인터럽트가 발생해서 마우스 인터럽트를 처리하게 된다.
처리된 인터럽트가 게임을 실행하는 것이라면
fork() 를 수행하고 자식 프로세스를 execve() 하여
게임에 해당하는 메모리 레이아웃으로 변형한다.
이때 사용되는 것이 sys_fork() 와 sys_execve() 로
sys_fork() 를 통해 새로운 task_struct 를 생성하고
sys_execve() 를 통해 ELF Header 와 Program Headers 에서
읽은 내용들을 기반으로 가상 메모리 레이아웃을 만들게 된다.

물론 이 때 먼저 게임이라는 실행 파일을 디스크에서 찾아야 하므로
super_block 을 통해서 파일 시스템의
메타 정보와 '/' 파일 시스템의 위치를 찾아온다.
이를 기반으로 파일이 실제 디스크 블록 어디에 있는지 찾고
그 다음에 앞서 기술했던 내용들을 진행한다.

그러면서 물리 메모리도 할당 해야 하는데
Demand On Paging 에 의해서 Page Fault 가 발생하고
이를 처리하기 위해 Page Fault Handler 도 동작할 것이다.

게임에 접속하기 위해 Networking 도 발생할 것이고
다른 프로세스들과 Context Switching 도 빈번하게 발생하면서
Run Queue 와 Wait Queue 를 왔다 갔다 할 것이다.

68. 현재 삽입된 디바이스 드라이버의 리스트를 보는 명령어는 무엇인가 ?

lsmod

70. Process와 VM과의 관계에 대해 기술하시오.

Process는 자신의 고유한 공간으로 가상의 4GB를 가지고 있다.
실제 이 공간을 모두 사용하지 않으며 Demand Paging에 따라 필요한 경우에만
실제 물리 메모리 공간을 Paging Mechanism을 통해 할당받아 사용한다.

80. 리눅스에서 말하는 File Descriptor(fd)란 무엇인가 ?

리눅스 커널에 존재하는 Task를 나타내는 구조체인
task_struct내에 files_struct내에 file 구조체에 대한 포인터가 fd_array다.
거기서 우리가 System Programming에서 얻는 fd는 바로 이 fd_array에 대한 index다.

82. 프로세스들은 최소 메모리를 관리하기 위한 mm, 파일 디스크립터인 fd_array, 그리고 signal을 포함하고 있는데 그 이유에 대해 기술하시오.

자신이 실제 메모리 어디에 위치하는지에 대한 정보가 필요하고
또 자신이 하드 디스크의 어떤 파일이 메모리에 로드되어
프로세스가 되었는지의 정보가 필요하며
마지막으로 프로세스들 간에
signal을 주고 받을 필요가 있기 때문에 signal이 필요하다.

87. 클라우드 기술의 핵심인 OS 가상화 기술에 대한 질문이다. OS 가상화에서 핵심에 해당하는 3 가지를 기술하시오

CPU 가상화, 메모리 가상화, I/O 가상화

94. 유저에서 fork() 를 수행할때 벌어지는 일들 전부를 실제 소스 코드 차원에서 해석하도록 하시오.

우리가 만든 프로그램에서 fork() 가 호출되면
C Library 에 해당하는 glibc 의 __libc_fork() 가 호출됨
이 안에서 ax 레지스터에 시스템 콜 번호가 기록된다.
즉 sys_fork() 에 해당하는 시스템 콜 테이블의 번호가 들어가고
이후에 int 0x80 을 통해서 128 번 시스템 콜을 후출하게 된다.
그러면 제어권이 커널로 넘어가서 idt_table(Interrupt Descriptor Table)로 가고
여기서 시스템 콜은 128 번으로 sys_call_table 로 가서
ax 레지스터에 들어간 sys_call_table[번호] 의 위치에 있는
함수 포인터를 동작시키면 sys_fork() 가 구동이 된다.
sys_fork() 는 SYSCALL_DEFINE0(fork) 와 같이 구성되어 있다.

95. 리눅스 커널의 arch 디렉토리에 대해서 설명하시오.

CPU(HW) 의존적인 코드가 위치한 영역이다.

96. 95 번 문제에서 arm 디렉토리 내부에 대해 설명하도록 하시오.

ARM 은 하위 호환이 안되고 다양한 반도체 벤더들이 개발을 하고 있기 때문에
해당 디렉토리에 들어가면 회사별 주요 제품들의 이름이 보이는 것을 확인할 수 있다.

97. 리눅스 커널 arch 디렉토리에서 c6x 가 무엇인지 기술하시오.

TI DSP 에 해당하는 Architecture 임

98. Intel 아키텍처에서 실제 HW 인터럽트를 어떤 함수를 가지고 처리하게 되는지 코드와 함께 설명하시오.

일반적인 HW 인터럽트는 어셈블리 루틴 common_interrupt 레이블에서 처리한다.
이 안에서는 do_IRQ() 라는 함수가 같이 함께
일반적인 HW 인터럽트를 처리하기 위해 분발한다.

99. ARM 에서 System Call 을 사용할 때 사용하는 레지스터를 적으시오.

r7