1. 파이프 통신을 구현하고 c type.c 라고 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);

	// 명령어 텍스트를 체크한다.
	if(text[0] != 'c' && text[1] != ' ')
		return NULL;

	//확정자를 체크한다/
	if(text[text_len - 1] != 'c' && text[text_len - 2] != '.')
		return NULL;

	// 텍스트 내부 문자를 체크하여 잘못된 경우 null 을 리턴하고 아닐경우 filename 에 저장한다.
	for(i = 2; i < text_len - 2; i++)
	{
		if(text[i] == ' ' || text[i] == '\t')
			return NULL;
		filename[i - 2] = text[i];
	}
	// 확정자를 붙이고, 해당 문자열의 주소를 리턴한다.
	strcat(filename, ".c");
	return filename;
}

int main(void)
{
	int fo;
	int fd, ret;
	char buf[1024];
	char *string = NULL;

	// fifo 파일인 myfifo 를 읽고 쓰기 형식으로 오픈한다.
	fd = open("myfifo", O_RDWR);

	// nonblock 형식으로 키보드와 myfifo 를 설정한다.
	// block 형식일 경우 진행이 안되기 때문이다.
	fcntl(0, F_SETFL, O_NONBLOCK);
	fcntl(fd, F_SETFL, O_NONBLOCK);
```

```
	for(;;)
	{
		// 키보드 입력을 받아온다.
		if((ret = read(0, buf, sizeof(buf))) > 0)
		{
			buf[ret - 1] = 0;		// 배열이기 때문에 마지막 위치를 0 으로 초기화한다.
			printf("Keyboard Input : [%s]\n", buf);
			string = check_text(buf);	// 문자열 주소를 받아온다.
			printf("String : %s\n", string);
		}
		// fifo 의 입력을 받아온다.
		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);
		}
		//null 이 아닐 경우, 해당 파일을 생성하고 닫는다.
		fo = open(string, O_WRONLY | O_CREAT | O_TRUNC, 0644);
		close(fo);
	}
	close(fd);
	return 0;
}
```

4. Unix 계열의 모든 OS 는 모든 것을 무엇으로 관리하는가 ?

파일

5. 리눅스에는 여러 장점이 있다.
 아래의 장점들 각각에 대해 기술하라.

* 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.

11. 리눅스 커널 소스에 보면 current 라는 것이 보인다.
이것이 무엇을 의미하는 것인지 적으시오.
커널 소스 코드와 함께 기술하시오.

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.

19. UMA 와 NUMA 에 대해 기술하고
Kernel 에서 이들을 어떠한 방식으로 관리하는지 기술하시오.
커널 내부의 소스 코드와 함께 기술하도록 하시오.

메모리 접근 속도가 같은 것을 Bank 라 한다.
그리고 Kernel 에서 이들은 Node 라 한다.
UMA 는 모든 CPU 가 메모리 접근 속도가 같은 SMP 와 같은것을 의미한다.
NUMA 는 CPU 마다 메모리 접근 속도가 다른 Intel 의 i 계열의 CPU 군을 의미한다.
Linux Kernel 에선 이를 contig_page_data 를 통해 전역변수로 관리한다.
그리고 UMA 의 경우엔 Node 가 1 개인데 pglist_data 구조체로 표현된다.
NUMA 의 경우엔 pglist_data 가 여러개 연결되어 있다.

20. Kernel 의 Scheduling Mechanism 에서 Static Priority 와 Dynamic Priority 번호가 어떻게 되는지 적으시오.

Static Priority : 1 ~ 99
Dynamic Priority : 100 ~ 139

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

물리 메모리의 최소 단위는 Page Frame 이라 하며
이것을 SW 적 개념으로 구현한 구조체의 이름은 page 다.

27. 프로그램을 실행한다고 하면 fork(), execve()의 콤보로 이어지게 된다.
이때 실제 gcc *.c 로 컴파일한 a.out 을 ./a.out 을 통해 실행한다고 가정한다.
실행을 한다고 하면 a.out File 의 Text 영역에 해당하는 부분을 읽어야 한다.
실제 Kernel 은 무엇을 읽고 이 영역들을 적절한 값으로 채워주는가 ?

ELF Header 와 Program Headers 를 읽고 값을 적절하게 채운다.

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

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

32. Character 디바이스 드라이버를 작성할 때 반드시 Wrapping 해야 하는 부분이 어디인가 ?
(Task 구조체에서 부터 연결된 부분까지를 쭉 이어서 작성하라)

task_struct->files_struct->file->file_operations 에 해당함

33. 예로 유저 영역에서 open 시스템 콜을 호출 했다고 가정할 때 커널에서는 어떤 함수가 동작하게 되는가 ?
실제 소스 코드 차원에서 이를 찾아서 기술하도록 한다.

Kernel Source 에서 아래 파일에 위치한다.
fs/open.c 에 위치하며 SYSCALL_DEFINE3(open, ~~~) 형태로 구동된다.
이 부분의 매크로를 분석하면 결국 sys_open() 이 됨을 알 수 있다.

35. VFS(Virtual File System)이 동작하는 Mechanism 에 대해 서술하시오.

VFS 는 Super Block 인 ext2_sb_info 와 같은 것들을 읽어 super_block 에 채워넣는다.
그리고 읽고자 하는 파일에 대한 메타 정보를
ext2_inode 와 같은 것들을 읽어 inode 에 채운다.
이후에 디렉토리 엔트리인 ext2_dir_entry 를 읽어 dentry 구조체에 채우고
현재 위치 정보에 대한 정보를 위해 path 구조체를 채운 이후
실제 File 에 대한 상세한 정보를 기록하기 위해 file 구조체를 채운다.
각각 적절한 값을 채워넣어 실제 필요한 파일을 Task 와 연결시킨다.

40. System Call 호출시 Kernel 에서 실제 System Call 을 처리하기 위해
Indexing 을 수행하여 적절한 함수가 호출되도록 주소값을 저장해놓고 있다.
이 구조체의 이름을 적으시오.

Intel 의 경우에 sys_call_table
ARM 의 경우에는 __vectors_start + 0x1000 에 해당함

41. 38 에서 User Space 에서 System Call 번호를 전달한다.
Intel Machine 에서는 이를 어디에 저장하는가 ?
또한 ARM Machine 에서는 이를 어디에 저장하는가 ?

Intel 의 경우에는 ax 레지스터에
ARM 의 경우에는 r7 레지스터에 해당한다.

43. 또한 Page Directory 를 가르키는 Intel 전용 Register 가 존재한다.
이 Register 의 이름을 적고 ARM 에서 이 역할을 하는 레지스터의 이름을 적으시오.

Intel 의 경우엔 CR3
ARM 의 경우엔 CP15

44. 커널 내부에서 메모리 할당이 필요한 이유는 무엇인가 ?

커널 스택으로 주어진 메모리 공간은 고작 8K 에 해당한다.
(물론 이 값은 ARM 이라고 가정하였을 때고 Intel 은 16 K 에 해당한다)
문제는 스택 공간이라 특정 작업을 수행하고
이후에 태스크가 종료되면 정보가 사라질 수도 있다.
이 정보가 없어지지 않고 유지될 필요가 있을 수도 있다.

뿐만 아니라 커널 자체도 프로그램이기 때문에 메모리 공간이 필요하다.
운영체제 또한 C 로 만든 프로그램에 불과하다는 것이다.
그러니 프로그램이 동작하기 위해 메모리를 요구하듯 커널도 필요하다.

45. 메모리를 불연속적으로 할당하는 기법은 무엇인가 ?

vmalloc()

46. 메모리를 연속적으로 할당하는 기법은 무엇인가 ?

Kmalloc()

47. Mutex 와 Semaphore 의 차이점을 기술하시오.

Mutex 와 Semaphore 모두 Context Switching 을 유발하게 된다.
차이점이라면 Mutex 는 공유된 자원의 데이터를 여러 쓰레드가 접근하는 것을 막는다.
Semaphore 는 공유된 자원의 데이터를 여러 프로세스가 접근하는 것을 막는 것이다.

50. thread_union 에 대해 기술하시오.

include/linux/sched.h 에 있는 공용체로 내부에는 커널 스택 정보와 thread_info 를 가지고 있다.
이 안에는 현재 구동중인 task 의 정보가 들어 있고 Context Switching 등에 활용하기 위해 cpu_context_save 구조체가 존재한다.

51. Device Driver 는 Major Number 와 Minor Number 를 통해 Device 를 관리한다.
실제 Device 의 Major Number 와 Minor Number 를 저장하는 변수는 어떤 구조체의 어떤 변수인가 ?
(역시 Task 구조체에서부터 쭉 찾아오길 바람)

task_struct 내에 files_struct 내에 file 내에 path 내에 dentry 내에 inode 구조체에 존재하는 i_rdev 변수에 저장한다.

52. 예로 간단한 Character Device Driver 를 작성했다고 가정해본다.
그리고 insmod 를 통해 Driver 를 Kernel 내에 삽입했으며
mknod 를 이용하여 /dev/장치파일을 생성하였다.
그리고 이에 적절한 User 프로그램을 동작시켰다.
이 Driver 가 실제 Kernel 에서 어떻게 관리되고 사용되는지 내부 Mechanism 을 기술하시오.

Device Driver 에서 class_create, device_create 를 이용해서
Character Device 인 /dev/장치파일을 만든다.

그리고 Character Device Driver 를 등록하기 위해서는 register_chrdev()가 동작해야 한다.
동적으로 할당할 경우에는 alloc_chrdev_region() 이 동작한다.

이때 file_operations 를 Wrapping 할 구조체를 전달하고 Major Number 와 장치명을 전달한다.
chrdevs 배열에 Major Number 에 해당하는 Index 에 file_operations 를 Wrapping 한 구조체를 저장해둔다.

그리고 이후에 실제 User 쪽에서 생성된 장치가 open 되면 그때는 Major Number 에 해당하는 장치의 File Descriptor 가 생성되었으므로 이 fd_array 즉, file 구조체에서 i_mode 를 보면 Character Device 임을 알 수 있고 i_rdev 를 보고 Major Number 와 Minor Number 를 파악할 수 있다.

그리고 chrdevs 에 등록한 Major Number 와 같음을 확인하고
이 fd_array 에 한해서는 open, read, write, lseek, close 등의
file_operations 구조체 내에 있는 함수 포인터들을 앞서 Wrapping 한 구조체로 대체한다.

그러면 User 에서 read()등을 호출할 경우 우리가 알고 있는 read 가 아닌
Device Driver 작성시 새로 만든 우리가 Wrapping 한 함수가 동작하게 된다.

53. Kernel 자체에 kmalloc(), vmalloc(), __get_free_pages()를 통해 메모리를 할당할 수 있다.
또한 kfree(), vfree(), free_pages()를 통해 할당한 메모리를 해제할 수 있다.
이러한 Mechanism 이 필요한 이유가 무엇인지 자세히 기술하라.

Device Driver 나 기타 여러 Kernel 내부의 Mechanism 을 수행하는데 있어서
자료를 저장할 공간이 Kernel 역시 필요할 것이다.
그 공간을 확보하기 위해 Memory Allocation Mechanism 이 Kernel 에도 존재한다.

54. Character Device Driver 를 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;
}
```

```
// 변경 후, 사용할 read 이다.
static int mydrv_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
	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];
}

// 변경 후, 사용할 write 이다.
static ssize_t mydrv_write(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
	int i;

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

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

	return write_ret[0];
}

// 기존 file_operations 의 기능을 만들어 둔 함수로 변경한다.
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);
```

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

#define MAX_BUFFER      26

char bin[MAX_BUFFER];
char bout[MAX_BUFFER];

int main(void)
{
        int fd, i, c = 65;
        int write_ret, read_ret;

        if((fd = open("/dev/mydevicefile", O_RDWR)) < 0)
        {
                perror("open()");
                return -1;
        }
           // 변경된 write 와 read 기능을 이용하여, 결과값을 저장한다.
        write_ret = write(fd, bout, 10);
        read_ret = read(fd, bin, 5);

        printf("write_ret = %d, read_ret = %d\n", write_ret, read_ret);

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

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

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

58. Compiler 의 Instruction Scheduling 은 Run-Time 이 아닌 Compile-Time 에 결정된다.
고로 이를 Static Instruction Scheduling 이라 할 수 있다.
Intel 계열의 Machine 에서는 Compiler 의 힘을 빌리지 않고도
어느저도의 Instruction Scheduling 을 HW 의 힘만으로 수행할 수 있다.
이러한 것을 무엇이라 부르는가 ?

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

ISA(Instruction Set Architecture) : 명령어 집합

61. Intel 의 Hyper Threading 기술에 대해 상세히 기술하시오.

Hyper Threading 은 Pentium 4 에서
최초로 구현된 SMT(Simultaneous Multi-Threading) 기술명이다.
Kernel 내에서 살펴봤던 Context 를 HW 차원에서 지원하기 때문에
실제 1 개 있는 CPU 가 논리적으로 2 개가 있는것처럼 보이게 된다.
HW 상에서 회로로 이를 구현하는 것인데
Kernel 에서 Context Switching 에선 Register 들을 저장했다면
이것을 HW 에서는 이러한 HW Context 들을 복사하여
Context Switching 의 비용을 최소화하는데 목적을 두고 있다.
TLP(Thread Level Parallelization) 입장에서보면
Mutex 등의 Lock Mechanism 이 사용되지 않는한
여러 Thread 는 완벽하게 독립적이므로 병렬 실행될 수 있다.
한마디로 Hyper Threading 은 Multi-Core 에서 TLP 를 극대화하기에 좋은 기술이다.

62. 그동안 많은 것을 배웠을 것이다.
최종적으로 Kernel Map 을 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) 등

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 레이아웃
을 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 발생
하면서 Run Queue 와 Wait Queue 를 왔다 갔다 할 것이다.

64. 서버와 클라이언트가 1 초 마다 Hi, Hello 를 주고 받게 만드시오.

- 클라이언트의 기본 형식은 동일하다.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

typedef struct sockaddr_in     si;
typedef struct sockaddr *      sp;

typedef struct __d{
        int data;
        float fdata;
} d;

#define BUF_SIZE                32

void err_handler(char *msg)
{
        fputs(msg, stderr);
        fputc('\n', stderr);
        exit(1);
}
```

```
// 서버를 통해 hello 를 받아오는 함수이다.
void read_proc(int sock, d *buf)
{
	char msg[32] = {0};

	for(;;)
	{
		int len = read(sock, msg, BUF_SIZE);

		if(!len)
			return;

		printf("%s\n", msg);
	}
}

// 서버에 hi 를 보내는 함수이다.
void write_proc(int sock, d *buf)
{
	char msg[32] = "Hi";

	for(;;)
	{
		write(sock, msg, strlen(msg));
		sleep(1);
	}
}

int main(int argc, char **argv)
{
	pid_t pid;
	int i, sock;
	si serv_addr;
	d struct_data;
	char buf[BUF_SIZE] = {0};

	if(argc != 3)
	{
		printf("use: %s <IP> <port>\n", argv[0]);
		exit(1);
	}

	sock = socket(PF_INET, SOCK_STREAM, 0);

	if(sock == -1)
		err_handler("socket() error");

	memset(&serv_addr, 0, sizeof(serv_addr));
	serv_addr.sin_family = AF_INET;
	serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
	serv_addr.sin_port = htons(atoi(argv[2]));
```

```
	if(connect(sock, (sp)&serv_addr, sizeof(serv_addr)) == -1)
		err_handler("connect() error");
	else
		puts("Connected!\n");

	pid = fork();

	if(!pid)
		write_proc(sock, (d *)&struct_data);
	else
		read_proc(sock, (d *)&struct_data);

	close(sock);

	return 0;
}
```

- 서버의 기본 형식도 동일하다.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <signal.h>
#include <sys/wait.h>

typedef struct sockaddr_in	si;
typedef struct sockaddr *	sp;

typedef struct __d{
	int data;
	float fdata;
} d;

#define BUF_SIZE			32

void err_handler(char *msg)
{
	fputs(msg, stderr);
	fputc('\n', stderr);
	exit(1);
}


void read_cproc(int sig)
{
	pid_t pid;
	int status;
	pid = waitpid(-1, &status, WNOHANG);
	printf("Removed proc id: %d\n", pid);
}
```

```
int main(int argc, char **argv)
{
	int serv_sock, clnt_sock, len, state;
	char buf[BUF_SIZE] = {0};
	si serv_addr, clnt_addr;
	struct sigaction act;
	socklen_t addr_size;
	d struct_data;
	pid_t pid;

	if(argc != 2)
	{
		printf("use: %s <port>\n", argv[0]);
		exit(1);
	}

	act.sa_handler = read_cproc;
	sigemptyset(&act.sa_mask);
	act.sa_flags = 0;
	state = sigaction(SIGCHLD, &act, 0);

	serv_sock = socket(PF_INET, SOCK_STREAM, 0);

	if(serv_sock == -1)
		err_handler("socket() error");

	memset(&serv_addr, 0, sizeof(serv_addr));
	serv_addr.sin_family = AF_INET;
	serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
	serv_addr.sin_port = htons(atoi(argv[1]));

	if(bind(serv_sock, (sp)&serv_addr, sizeof(serv_addr)) == -1)
		err_handler("bind() error");

	if(listen(serv_sock, 5) == -1)
		err_handler("listen() error");

	for(;;){
		addr_size = sizeof(clnt_addr);
		clnt_sock = accept(serv_sock, (sp)&clnt_addr, &addr_size);

		if(clnt_sock == -1)
			continue;
		else
			puts("New Client Connected!\n");

		pid = fork();

		if(pid == -1){
			close(clnt_sock);
			continue;
		}
```

```
                if(!pid)
                {
                        close(serv_sock);

                        // 연결된 소켓을 통해 읽어온다.
                        while((len = read(clnt_sock, buf, BUF_SIZE)) != 0)
                        {
                                // hi 를 출력한다.
                                printf("%s\n", buf);
                                // hello 를 보낸다.
                                write(clnt_sock, "Hello", strlen("Hello"));
                                sleep(1);
                        }

                        close(clnt_sock);
                        puts("Client Disconnected!\n");
                        return 0;
                }
                else
                        close(clnt_sock);
        }
        close(serv_sock);

        return 0;
}
```

65. Shared Memory 를 통해 임의의 파일을 읽고 그 내용을 공유하도록 프로그래밍하시오.

- 헤더를 통해 프로토타입과 공유되는 것들을 선언한다.

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

typedef struct
{
      char name[20];
      char buf[1024];
} SHM_t;

int CreateSHM(long key);
int OpenSHM(long key);
SHM_t *GetPtrSHM(int shmid);
int FreePtrSHM(SHM_t *shmptr);
```

- 프로토 타입으로 선언된 것들의 내용물이다.

```
#include "65_shm.h"

int CreateSHM(long key)
{
	return shmget(key, sizeof(SHM_t), IPC_CREAT | 0777);
}

int OpenSHM(long key)
{
	return shmget(key, sizeof(SHM_t), 0);
}

SHM_t *GetPtrSHM(int shmid)
{
	return (SHM_t *)shmat(shmid, (char *)0, 0);
}

int FreePtrSHM(SHM_t *shmptr)
{
	return shmdt((char *)shmptr);
}
```

- 파일을 열고, 데이터를 SHM 으로 옮기는 프로세스다.

```
#include "65_shm.h"
#include <fcntl.h>

int main(int argc, char **argv)
{
	int fd;
	int ret;
	int mid;
	char buf[1024];
	SHM_t *p;

	mid = OpenSHM(0x888);

	p = GetPtrSHM(mid);

	getchar();
	fd = open(argv[1], O_RDONLY);
	ret = read(fd, buf, sizeof(buf));
	buf[ret - 1] = 0;
	strcpy(p -> name, argv[1]);
	strcpy(p -> buf, buf);

	FreePtrSHM(p);

	return 0;
}
```

- SHM 을 통하여, 데이터를 받아서 출력하는 프로세스이다.

```
#include "65_shm.h"

int main(void)
{
	int mid;
	SHM_t *p;

	mid = CreateSHM(0x888);

	p = GetPtrSHM(mid);

	getchar();
	printf("name : %s\nbuf : %s\n", p -> name, p -> buf);

	FreePtrSHM(p);

	return 0;
}
```

68. 현재 삽입된 디바이스 드라이버의 리스트를 보는 명령어는 무엇인가 ?

lsmod

70. Process 와 VM 과의 관계에 대해 기술하시오.

Process 는 자신의 고유한 공간으로 가상의 4GB 를 가지고 있다.
실제 이 공간을 모두 사용하지 않으며 Demand Paging 에 따라 필요한 경우에만
실제 물리 메모리 공간을 Paging Mechanism 을 통해 할당받아 사용한다.

71. 인자로 파일을 입력 받아 해당 파일의 앞 부분 5 줄을 출력하고
추가적으로 뒷 부분의 5 줄을 출력하는 프로그램을 작성하시오.

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

int save_area[1024];

int main(int argc, char **argv)
{
   int i;
   int ret;
   int index;
   int fd;
   char buf[1024];

// 파일을 열고, 데이터를 읽어온다.
   fd = open(argv[1], O_RDONLY);
   ret = read(fd, buf, sizeof(buf));
```

```
    for(i = 0, index = 1; buf[i]; i++)
    {
        if(buf[i] == '\n')
        {
            // 행이 나뉘는 위치를 저장한다.
            save_area[index] = i;
            index++;
        }
    }
            // 저장한 위치를 기반으로 행을 출력한다.
    printf("Front 5 Lines\n");
    write(0, buf, save_area[5] + 1);
    printf("Back 5 Lines\n");
    printf("%s", &buf[save_area[index - 6] + 1]);
    return 0;
}
```

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

리눅스 커널에 존재하는 Task 를 나타내는 구조체인
task_struct 내에 files_struct 내에 file 구조체에 대한 포인터가 fd_array 다.
거기서 우리가 System Programming 에서 얻는 fd 는 바로 이 fd_array 에 대한 index 다.

83. 디렉토리를 만드는 명령어는 mkdir 명령어다.
man -s2 mkdir 을 활용하여 mkdir System Call 을 볼 수 있다.
이를 참고하여 디렉토리를 만드는 프로그램을 작성해보자!

```
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        if(argc != 2) // 인자 체크를 한다.
        {
                printf("Usage: exe dir_name\n");
                exit(-1);
        }

           // 2 번째 인자를 통해 파일을 생성한다.
        mkdir(argv[1], 0755);

        return 0;
}
```

84. 이번에는 랜덤한 이름(길이도 랜덤)을 가지도록 디렉토리를 3 개 만들어보자!
(너무 길면 힘드니까 적당한 크기로 잡도록함)

```
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

char *rand_name(void)          // 파일 이름을 랜덤하게 설정한다.
{
        int i, len;
        static char buf[8] = "\0";

        memset(buf, 0, sizeof(buf));
        len = rand() % 4 + 4;      // 파일명 길이를 랜덤으로 정한다.

        for(i = 0; i < len; i++)     // 파일 명을 저장한다.
                buf[i] = rand() % 26 + 97;

        printf("buf = %s\n", buf);

        return buf;
}

void make_rand_dname(char **dname){
        int i;

        for(i = 0; i < 3; i++){
                dname[i] = (char *)malloc(sizeof(char) * 8); // 파일명을 저장할 메모리 공간을 확보한다.
                strcpy(dname[i], rand_name()); // 함수를 통해 랜덤한 이름을 받고, 저장한다.
                //dname[i] = rand_name();
        }
}

int main(void){
        int i;
        char *dname[3] = {0};

        srand(time(NULL));       // 렌덤을 사용하기 위해 사용한다.
        make_rand_dname(dname);       // 렌덤하게 파일명을 설정한다.

        for(i = 0; i < 3; i++){
                printf("dname[%d] = %s\n", i, dname[i]);
                mkdir(dname[i], 0755);      // 각각의 파일을 생성한다.
        }

        return 0;
}
```

85. 랜덤한 이름을 가지도록 디렉토리 3 개를 만들고
각각의 디렉토리에 5 ~ 10 개 사이의 랜덤한 이름(길이도 랜덤)을 가지도록 파일을 만들어보자!
(너무 길면 힘드니까 적당한 크기로 잡도록함)

```
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>

char *rand_name(void){
	int i, len;
	static char buf[8] = "\0";

	memset(buf, 0, sizeof(buf));
	len = rand() % 4 + 4;

	for(i = 0; i < len; i++)
		buf[i] = rand() % 26 + 97;

	printf("buf = %s\n", buf);

	return buf;
}

void make_rand_dname(char **dname){
	int i;

	for(i = 0; i < 3; i++)
	{
		dname[i] = (char *)malloc(sizeof(char) * 8);
		strcpy(dname[i], rand_name());
	}
}

// 생성 방법은 동일하다.
void make_rand_file(void){
	int i, j, fd, len, cnt;
	char buf[11] = "\0";

	len = rand() % 6 + 5;
	cnt = rand() % 4 + 2;

	for(i = 0; i < cnt; i++)
	{
		for(j = 0; j < len; j++)
			buf[j] = rand() % 26 + 97;

		fd = open(buf, O_CREAT, 0644);
```

```
            close(fd);

            memset(buf, 0, sizeof(buf));
        }
}

// 추가적으로 내부로 들어가 파일을 생성한다.
void lookup_dname(char **dname)
{
        int i;

        for(i = 0; i < 3; i++)
        {
            chdir(dname[i]);
            make_rand_file();
            chdir("..");
        }
}

int main(void)
{
        int i;
        char *dname[3] = {0};

        srand(time(NULL));
        make_rand_dname(dname);

        for(i = 0; i < 3; i++)
            mkdir(dname[i], 0755);

        lookup_dname(dname);

        return 0;
}
```

86. 85 번까지 진행된 상태에서 모든 디렉토리를 순회하며
3 개의 디렉토리와 그 안의 모든 파일들의 이름 중 a, b, c 가 1 개라도 들어있다면 이들을 출력하라!
출력할 때 디렉토리인지 파일인지 여부를 판별하도록 하시오.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>
```

```
char *rand_name(void)
{
	int i, len;
	static char buf[8] = "\0";

	memset(buf, 0, sizeof(buf));
	len = rand() % 4 + 4;

	for(i = 0; i < len; i++)
		buf[i] = rand() % 26 + 97;

	printf("buf = %s\n", buf);

	return buf;
}

void make_rand_dname(char **dname)
{
	int i;

	for(i = 0; i < 3; i++)
	{
		dname[i] = (char *)malloc(sizeof(char) * 8);
		strcpy(dname[i], rand_name());
	}
}

void make_rand_file(void)
{
	int i, j, fd, len, cnt;
	char buf[11] = "\0";

	len = rand() % 6 + 5;
	cnt = rand() % 4 + 2;

	for(i = 0; i < cnt; i++)
	{
		for(j = 0; j < len; j++)
			buf[j] = rand() % 26 + 97;

		fd = open(buf, O_CREAT, 0644);
		close(fd);

		memset(buf, 0, sizeof(buf));
	}
}

void lookup_dname(char **dname)
{
	int i;

	for(i = 0; i < 3; i++)
```

```
	{
		chdir(dname[i]);
		make_rand_file();
		chdir("..");
	}
}

// 찾는 함수이다.
void find_abc(void)
{
	struct dirent *p;
	struct stat buf;
	DIR *dp;
	int i, len;

	dp = opendir(".");

	while(p = readdir(dp))	// 내부에 위치한 파일들을 p 에 저장한다.
	{
		stat(p->d_name, &buf);	// 정보와 길이를 저장한다.
		len = strlen(p->d_name);

		for(i = 0; i < len; i++)	// 해당 단어가 들어있을 경우 출력한다.
		{
			if(!strncmp(&p->d_name[i], "a", 1) ||
			   !strncmp(&p->d_name[i], "b", 1) ||
			   !strncmp(&p->d_name[i], "c", 1))
			{
				printf("name = %s\n", p->d_name);
			}
		}
	}

	closedir(dp);
}

void recur_find(char **dn)
{
	int i;

	for(i = 0; i < 3; i++)
	{
		chdir(dn[i]);
		find_abc();
		chdir("..");
	}
}
```

```
int main(void)
{
	int i;
	char *dname[3] = {0};

	srand(time(NULL));
	make_rand_dname(dname);

	for(i = 0; i < 3; i++)
		mkdir(dname[i], 0755);

	lookup_dname(dname);
	recur_find(dname);

	return 0;
}
```

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

CPU 가상화, 메모리 가상화, I/O 가상화

88. 반 인원이 모두 참여할 수 있는 채팅 프로그램을 구현하시오.

89. 88 번 답에 도배를 방지 기능을 추가하시오.

90. 89 번 조차도 공격할 수 있는 프로그램을 작성하시오.

91. 네트워크 상에서 구조체를 전달할 수 있게 프로그래밍 하시오.

92. 91 번을 응용하여 Queue 와 Network 프로그래밍을 연동하시오.

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.

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.