TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2 차시험 오답노트

(2018-04-30)

1 과목: 시스템 프로그래밍
2 과목: 네트워크 프로그래밍
3 과목: 리눅스 커널

강사: lnnova Lee(이상훈)
gcccompil3r@gmail.com

학생: **정유경**
ucong@naver.com

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);
	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);
	fcntl(0, F_SETFL, O_NONBLOCK);
	fcntl(fd, F_SETFL, O_NONBLOCK);
	for(;;)
	{
		if((ret = read(0, buf, sizeof(buf))) > 0)
		{
			buf[ret - 1] = 0;
			printf("Keyboard Input : [%s]\n", buf);
			string = check_text(buf);
			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;
}
```

2. 369 게임을 작성하시오. 2 초내에 값을 입력하게 하시오. 박수를 쳐야 하는 경우를 Ctrl + C 를 누르도록 한다. 2 초내에 값을 입력하지 못할 경우 게임이 오버되게 한다. Ctrl + C 를 누르면 "Clap!"이라는 문자열이 출력되게 한다.

```
/* gcc 2.c -lm */

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

char buf[1024];
int starting_val = 1;
int needs_clap;

void gameover(int signo)
{
   printf("\nGame Over!!!\n");
   exit(0);
}

void clap(void)
{
   printf("\nClap!\n");
   needs_clap--;
}

int find_parm(void)
{
   int i;
   int parm = 0;
   for(i = 0; ; i++)
      if(starting_val < pow(10, i))
         break;

   return parm = i;
}

void counting_clap(int signo)
{
   int i;
   int flag;
   int buf_len;

   if(needs_clap == 0)
   {
      flag = 1;
      buf_len = find_parm();
      for(i = 0; i < buf_len; i++)
      {
```

```
            if(i == 0)
                buf[i] = starting_val % (int)pow(10, i + 1);
            else
            {
                buf[i] = starting_val - buf[0];
                buf[i] /= pow(10, i);
            }
        }
        starting_val++;
    }

    if(flag == 1)
        for(i = 0; i < buf_len; i++)
            if(buf[i] == 3 || buf[i] == 6 || buf[i] == 9)
                needs_clap++;

    if(needs_clap)
    {
        flag = 0;
        clap();
    }
    else
        gameover(0);
}

int main(void)
{
    int i;
    int ret;
    int cmp;
    int buf_len;

    signal(SIGALRM, gameover);
    signal(SIGINT, counting_clap);

    for(;;)
    {
        alarm(2);
        ret = read(0, buf, sizeof(buf));
        buf[ret - 1] = 0;
        buf_len = find_parm();

        for(i = 0; i < buf_len; i++)
            if(buf[i] == '3' || buf[i] == '6' || buf[i] == '9')
                gameover(0);

        cmp = atoi(buf);

        if(starting_val != cmp)
            gameover(0);
        starting_val++;
    }

    return 0;
}
```

3. 리눅스 커널은 운영체제(OS)다. OS 가 관리해야 하는 제일 중요한 5 가지에 대해 기술하시오.
메모리, 태스크, 디바이스 드라이버, 네트워크, 파일 시스템

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.
* 열약한 환경에서도 HW 자원을 적절히 활용하여 동작한다.
* 커널의 크기가 작다.
* 완벽한 멀티유저, 멀티태스킹 시스템
* 뛰어난 안정성
* 빠른 업그레이드
* 강력한 네트워크 지원
* 풍부한 소프트웨어

* 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 지속적으로 개발되기 때문에 어떠한 열약한 환경에서도 잘 동작한다.

* 커널의 크기가 작다.
커널의 크기가 윈도우 커널보다 작다 즉, 최적화가 잘 되어 있다

* 완벽한 멀티유저, 멀티태스킹 시스템
리눅스는 RT Scheduling 방식을 채택하여 Multi-Tasking 을 효율적으로 잘 해낸다.
* 뛰어난 안정성과 빠른 업그레이드
전 세계의 많은 개발자들이 지속적으로 유지보수하여 안정성이 뛰어나다.
* 강력한 네트워크 지원
TCP/IP Stack 을 Router 및 Switch 등의 장비에서 사용함
* 풍부한 소프트웨어
GNU(GNU is Not Unix) 정신에 입각하여 많은 Tool 들 즉, 풍부한 소프트웨어가 개발되어 있다.

6. 32bit System 에서 User 와 Kernel 의 Space 구간을 적으시오.
가상주소공간은 유저영역과 커널영역으로 나뉜다
0x00000000~0x7FFFFFFF: 유저 메모리 구간
0x80000000~ 0xFFFFFFFF: 커널영역
커널 영역 1GB 를 빼고는 나머지 3GB 는 유저 영역이다.
즉, User 0 ~ 3 GB, Kernel 3 ~ 4 GB

7. Page Fault 가 발생했을때 운영체제가 어떻게 동작하는지 기술하시오.
Page Fault 는 가상 메모리를 물리 메모리로 변환하는 도중 물리 메모리에 접근했더니 할당된 페이지가 없을 경우 발생한다.(즉, 프로그램의 페이지가 물리 메모리에 부재하는 경우) 이것이 발생하면 현재 수행중이던 ip(pc) 레지스터를 저장하고 페이지에 대한 쓰기 권한을 가지고 있다면 Page Fault Handler 를 구동시켜서 페이지를 할당한다. 이후, 저장해 놨던 ip(pc) 를 복원하여 다시 Page Fault 가 발생했던 루틴을 구동시킨다. 만약 쓰기 권한이 없다면 Segmentation Fault 를 발생시킨다.
CPU 는 물리 메모리을 확인하여페이지가 없으면 trap 을 발생하여 운영체제에 알린다.
→ 운영체제는 CPU 의 동작을 잠시 멈춘다. → 운영체제는 페이지 테이블을 확인하여 가상메모리에 페이지가 존재하는지 확인하고, 없으면 프로세스를 중단한다. → 페이지 폴트이면, 현재 물리 메모리에 비어있는 프레임(Free Frame)이 있는지 찾는다. → 비어있는 프레임에 해당 페이지를 로드하고, 페이지 테이블을 갱신한다. → 중단되었던 CPU 를 다시 시작한다

8. 리눅스 실행 파일 포맷이 무엇인지 적으시오.
ELF(Executalbe Linkable Format)

9. 프로세스와 스레드를 구별하는 방법에 대해 기술하시오.
Process 는 Group 의 Leader 에 해당하므로 tgid 와 pid 값이 같다. Thread 의 경우 Process 그룹에 속한 구성원으로 pid 는 고유하지만 tgid 가 모두 서로 같다.

10. Kernel 입장에서 Process 혹은 Thread 를 만들면 무엇을 생성하는가 ?
task_struct 구조체가 메모리에 Load 되어 객체화된다. 즉, task_struct 를 생성한다.

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. (thread_info *task 가 current 이다)

```
struct thread_info {
	unsigned long		flags;		/* low level flags */
	int			preempt_count;	/* 0 => preemptable, <0 => bug */
	mm_segment_t		addr_limit;	/* address limit */
	struct task_struct	*task;		/* main task structure */
```

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 방식은 특유의 세그먼트 레지스터를 사용하여 관리하는 것을 알수 있다.

12. Memory Management 입장에서 Process 와 Thread 의 핵심적인 차이점은 무엇인가 ?
프로세스는 자신만의 고유 공간과 자원을 할당 받아 사용(Process 간에는 서로 메모리 영역을 공유하지 않는다)하는데 비해 스레드는 프로세스 내에서 각각의 스택 공간을 제외한 나머지 공간과 시스템 자원을 공유(Thread 는 자신의 그룹에 속한 모든 Task 들과 메모리를 공유)한다.

13. Task 가 관리해야하는 3 가지 Context 가 있다. System Context, Memory Context, HW Context 가 있다. 이중 HW Context 는 무엇을 하기 위한 구조인가 ?
- 스케줄링이 일어나면 문맥교환이 발생하고 문맥교환시엔 현재 수행 중이던 태스크의 문맥을 저장해두어야 한다.
- Task 가 동작하다 Context Switching 등이 발생했을 경우 다른 Task 가 Register 를 변경할 수 있으므로 자신이 현재 어디까지 진행했는지를 기록할 필요가 있다. 이러한 레지스터등의 정보를 저장하기 위한 용도이다.

14. 리눅스 커널의 스케쥴링 정책중 Deadline 방식에 대해 기술하시오.
리눅스 커널의 Deadline Scheduler 는 Dario Faggioli 에 의해 2013 년 3 월 Linux Kernel v3.14 에서 소개되었다.
Deadline Scheduler 는 EDF + CBS 알고리즘 기반으로 동작한다. 임베디드 개발자들이 주로 원하는 기능이지만 토발즈가 별로 탐탁치 않아 한다. (CPU Scheduler 만으로 Real-Time 을 해결할 수 없다고 생각해서 그러함)
Task 마다 주어진 주기를 가지고 실행되는 것을 보장해야 한다. dl Task 의 우선 순위는 종종 변동되는데, 스케줄링 이벤트가 일어날 때마다, 큐에서 마감시간이 가장 가까운 프로세스를 탐색하여 다음에 수행되도록 한다.
즉 가장 급한 태스크를 스케쥴링 대상으로 선정하며, 각 태스크들은 RB 트리에 정렬되어 있다

15. TASK_INTURRUPTIBLE 과 TASK_UNINTERRUPTIBLE 은 왜 필요한지 기술하시오.
TASK_INTERRUPTIBLE: 인터럽트 수신가능한 상태이다
인터럽트 처리를 위해 기다리던 이벤트에 대한 대기를 중지하고 준비 상태로 복귀해야 하는 경우에 필요하다.
TASK_UNINTERRUPTIBLE: 인터럽트를 허용하지 않는 상태, 대기 중에 해당 프로세스에 시그널이 전달되어도 원래의 기다리는 이벤트가 발생할 때 까지 대기를 유지하기 위해 필요하다

ex 우리가 사용하는 프로그램에서는 반드시 순차적으로 동작해야 하는 구간이 존재하고 또한 순차적으로 동작하면 성능에 손해를 보는 구간도 존재한다.
이러한 것 때문에 User Level 에서 thread 를 사용할 경우 Critical Section 이 존재한다면 Lock Mechanism 을 사용한다. Lock Mechanism 을 사용중일 경우에는 Interrupt 를 맞으면 안되기 때문에 TASK_UNINTERRUPTIBLE 이 필요하다. 반대로 그렇지 않은 경우라면 Interrupt 에 무관하므로 TASK_INTERRUPTIBLE 을 사용한다.

16. O(N)과 O(1) Algorithm 에 대해 기술하시오. 그리고 무엇이 어떤 경우에 더 좋은지 기술하시오.
O(N)의 경우 데이터 개수가 많아질 수록 스케줄링에 걸리는 시간이 선형적으로 증가한다 즉, 알고리즘의 성능이 떨어진다. 처리량이 적은 경우 유리하다
O(1)의 경우 데이터 개수와 상관없이 처리시간은 일정하다 즉, 데이터의 양이 많아지더라도 알고리즘의 성능이 언제나 동일하다. 처리량이 많은 경우 유리하다

17. 현재 4 개의 CPU(0, 1, 2, 3)가 있고 각각의 RQ 에는 1, 2 개의 프로세스가 위치한다. 이 경우 2 번 CPU 에 있는 부모가 fork()를 수행하여 Task 를 만들어냈다. 이 Task 는 어디에 위치하는 것이 좋을까 ? 그리고 그 이유를 적으시오.
2 번에 위치하는 것이 좋다. fork()는 같은 Task 를 복사하게 된다. 그리고 다시 그 코드를 사용할 것이라면 CPU 의 Instruction Cache, Data Cache 를 재활용하는 것이 최고다. 결론적으로 Cache Affinity 를 적극 활용하자는 것이다.

18. 15 번 문제에서 이번에는 0, 1, 3 에 매우 많은 프로세스가 존재한다. 이 경우 3 번에서 fork()를 수행하여 Task 를 만들었다. 이 Task 는 어디에 위치하는 것이 좋을까 ? 역시 이유를 적으시오.
2 번에 위치하는 것이 좋다. 이번에는 주어진 시간내에 0, 1, 3 은 Scheduling 을 수행할 수 없기 때문에 오히려 상대적으로 Task 가 적은 2 번에 배치하는 것이 좋다.
(3 번 CPU 에 이미 많은 프로세스가 존재하므로 3 번 CPU 를 사용하면 cache miss 가 많이 발생할 것이다. 따라서 내부적으로 Load Balnacing 을 수행하게 된다.)

Cache Affinity 를 포기할지라도 아예 Task 를 동작시킬 기회가 없는것보다 좋기 때문이다.

19. UMA 와 NUMA 에 대해 기술하고 Kernel 에서 이들을 어떠한 방식으로 관리하는지 기술하시오. 커널 내부의 소스코드와 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.
Kernel 은 NUMA, UMA 모두를 노드라는 일관된 자료구조를 통해서 전체 물리메모리에 접근할 수 있도록 관리한다. Linux Kernel 에선 이를 contig_page_data 를 통해 전역변수로 관리한다. 그리고 UMA 의 경우엔 Node 가 1 개인데 pglist_data 구조체로 표현된다.//??
UMA 구조의 시스템에서는 한개의 노드가 존재하며 이 노드는 전역변수인 contig_page_data 를 통해 접근가능하다. NUMA 구조의 시스템에서는 복수개의 노드가 존재하며 이는 구조체 pglist_data 를 통해 접근 가능하다. 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 : 실시간 프로세스에만 부여되는 값, 스케줄러로 변경할 수 없다 0 ~ 99
Dynamic Priority : 스케줄러가 각각의 프로세스에게 지정한 우선순위 값 100 ~ 139 (0~99 가 리얼타임 프로세스 우선순위이고, 일반 프로세스는 nice()로 우선순위가 정해지는데 (-20 ~ 19)의 범위를 가진다. 이 값은 커널 내부적으로 priority+120 으로 변환된다 즉, 100 ~ 139 가 일반 프로세스 우선순위에 해당하며 항상 실시간 태스크의 우선순위보다 작다.
(-20 ~ 19) + 120 → (100 ~ 139))

21. ZONE_HIGHMEM 에 대해 아는대로 기술하시오.
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 1:1 로 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_NORMAL 을 초과하는 메모리가 이 영역을 사용한다.
64bit 시스템에서는 모든 물리 메모리가 1:1 매핑이 가능하므로 ZONE_HIGHMEM 을 사용하지 않는다.
- 리눅스의 가상주소공간과 물리메모리 공간을 1:1 로 연결한다면 1 GB 이상은 접근이 불가능하다. 따라서 물리메모리가 1GB 이상일때 896MB 까지를 커널의 가상주소 공간과 1:1 로 연결하고 나머지부분은 필요할 때 동적으로 연결하여 사용하는 구조를 가진다. 이때 896MB 이상의 메모리 영역을 ZONE_HIGHMEM 이라고 부른다
즉, 커널은 1G 영역을 896M 직접 매핑 가능한 메모리를 제외하고 High memory 공간으로 128M 를 사용한다.
64 비트 시스템의 경우 직접 매핑 가능한 메모리의 공간이 크기 때문에 high memory 의 크기는 0 이다.

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.

23. Linux Kernel 은 외부 단편화와 메모리 부하를 최소화하기 위해 Buddy 할당자를 사용한다. Buddy 할당자의 Algorithm 을 자세히 기술하시오.
Buddy 할당자가 메모리를 관리하는 방식에서 할당할 경우 Bitmap 을 같이 고려하는 것은 확실히 Overhead 다.
그러나 메모리를 해제하고 어떤 녀석이 어디에 얼마만큼 비어있는지 파악하는 것은 바로 이 Buddy Mechanism 을 이용하면 굉장히 유용하다.
Order(0)일때는 총 8K 를 비교하는데 4K, 4K 의 공간에서 2 개의 상태값이 같은지를 본다. 찾고자하는 것은 지금 이 공간의 Maximum 이 얼마만큼 되는지를 찾아서 4K 를 찾는다고하면 4K 만 정확히 떼주고 16K 라면 16K 를 떼주기 위한 의도다.
Order(1)일 경우에는 16K 를 비교하게 된다. 8K 와 8K 의 상태값이 서로 같으면 역시 Bitmap 을 0 으로 표기하고 둘의 상태값이 다르면 1 이 표기되면서 8K 공간을 활용할 수 있다는 의미가 된다.
둘다 사용하지 않을 경우에도 0 인데 제공할 수 있는 Maximum 에 해당하는 공간을 가지고 있기 위함이다. Lazy Buddy 의 경우 bitmap 을 없애고 nr_free 를 사용한다.
- struct zone > free_area[ ] > struct free_area > free_list, nr_free
free_area 배열은 10 개의 엔트리를 가지는데 0~9 의 숫자는 각각 해당 free_area 가 관리하는 할당의 크기를 나타내며, 버디는 2 의 정수승 개의 페이지 프레임을 할당해준다 (4,8,16,32KB)
free_area 구조체는 free_list 변수를 통해 사용하지 않는 페이지 프레임을 리스트로 관리한다.
(free_area[1]에는 연속된 2 개의 페이지프레임이 free_list 로 연결되어 있다)
만일, 2 개의 페이지 프레임 할당 요청이 발생하면 버디할당자는 free_area[1]의 free_list 를 보고 할당가능한 연속한 페이지프레임을 찾아낸뒤 할당하고 비트맵을 변경해준다.
만일, 4 개의 페이지 프레임이 필요하여 리스트를 살펴보았는데 없을 경우, 8 개의 연속된 페이지 프레임을 찾아서 분할 할당 받고, 나머지 4 개는 다시 free_area[2]의 free_list 에 넣어둔다.
즉. 최대한 큰 연속된 공간을 유지하면서 효율적으로 메모리를 관리할 수 있다.

24. 21 번에 이어 내부 단편화를 최소화 하기 위해 Buddy 에서 Page 를 받아 Slab 할당자를 사용한다. Slab 할당자는 어떤식으로 관리되는지 기술하시오.
사용자가 19byte 를 요청하면 커널은 시스템의 페이지 단위인 4KB 를 할당하는 것이 아니라 32byte 를 할당한다. 만일 물리메모리 최소단위인 4KB 를 할당할 경우 내부단편화로 공간 낭비를 불러온다. 이러한 내부단편화 문제를 해결하기 위한 것이 슬랩할당자이다. 슬랩할당자는 미리 정해진 크기의 cache 를 가지고 있다가 필요할 때 할당한다.
Slab 할당자는 Buddy 로부터 미리 Page 를 할당받아서 32byte 부터 128K 까지 적정한 수준의 크기로 분할을 하여 가지고 있다.
(다양한 크기의 캐시를 효율적으로 관리하기 위해 kmem_cache_t 라는 자료 구조를 만들어 두고, 새로운 캐시를 생성할 때 kmem_cache_t 라는 구조체로부터 할당 받는다. ← 이 때 슬랩할당자로부터 할당받는다)
그리고 우리가 작은 크기의 메모리 할당을 요청하면 미리 받아둔 이 조각들을 우리에게 넘겨준다. (각 cache 들은 슬랩들로 구성되고, 슬랩은 다시 객체들로 구성된다.
64byte cache 이면, 64byte 공간들이 각각의 객체이고, 이 객체들이 모여서 슬랩이 되고, 이 슬랩들이 모여서 cache 가 된다)
내부적으로도 사용할 수 있는 공간이 존재하므로 Free, Full, Partial 라는 상태 정보를 기록하여 새로 Free 한 부분을 사용해야할지 Partial 한 부분을 사용할지 결정할 수도 있다.

25. Kernel 은 Memory Management 를 수행하기 위해 VM(가상 메모리)를 관리한다. 가상 메모리의 구조인 Stack, Heap, Data, Text 는 어디에 기록되는가 ? (Task 구조체의 어떠한 구조체가 이를 표현하는지 기술하시오)
가상메모리의 구조를 기록하는 변수들이 있다.
task_struct 구조체에 있는 mm_struct 내에 start_code, end_code 등으로 기록됨
이들은 unsigned long 으로 선언되어 있으며 총 7 개이다.
(unsigned long start_code, end_code, start_data, end_data, start_brk, brk. start_stack; → 순서대로 텍스트영역, 데이터 영역, 힙영역과 스택영역을 나타낸다)
26. 23 번에서 Stack, Heap, Data, Text 등 서로 같은 속성을 가진 Page 를 모아서 관리하기 위한 구조체 무엇인가 ? (역시 Task 구조체의 어떠한 구조체에서 어떠한 식으로 연결되는지 기술하시오)
- 가상메모리 공간 중 같은 속성을 가지고 연속인 영역을 region 이라고 한다.
- task_struct 내에 mm_struct 포인터를 따라가 보면 vm_area_struct 구조체가 있다. 이 녀석이 서로 같은 Segment 들(즉 region)을 모아서 관리한다.
(task_struct > mm_struct > vm_area_struct) 이때, 같은 태스크에 속한 vm_area_struct 가 모여 하나의 mm_struct 내에서 관리된다

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.

28. User Space 에도 Stack 이 있고 Kernel Space 에도 Stack 이 존재한다. 좀 더 정확히는 각각에 모두 Stack, Heap, Data, Text 의 메모리 기본 구성이 존재한다. 그 이유에 대해 기술하시오.
(커널스택은 유저 스택보다 크기가 훨씬 작으며 ARM 의 경우 8KB, 인텔 CPU 의 경우 16KB 를 고정할당받는다.)
C 언어를 사용하기 위해서는 반드시 Stack 이 필요하다. Kernel 영역에서도 동작하는 코드가 올라가기 위한 Text 영역 전역 변수가 있는 Data 영역, 동적 할당하는 Heap, 지역 변수를 사용하는 Stack 이 존재한다. 이는 역시 User 영역에서도 동일하므로 양쪽에 모두 메모리 공간이 구성된다.

29. VM(가상 메모리)와 PM(물리 메모리)를 관리하는데 있어 VM 을 PM 으로 변환시키는 Paging Mechanism 에 대해 Kernel 에 기반하여 서술하시오.
mm_struct 에 pgd 라는 필드가 있다. Page Directory 를 의미하는 것으로 pgd -> pte -> page 로 3 단계 Paging 을 수행한다. 각각 10bit, 10bit, 12bit 로 VM 의 주소를 쪼개서 Indexing 을 한다. 그러면 물리메모리 상의 실제 접근하게 될 페이지 프레임 주소 얻을 수 있다.

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

31. 하드디스크의 최소 단위를 무엇이라 부르고 그 크기는 얼마인가 ?
Sector, 512 byte

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.
- 실제 리눅스 커널에 선언되어 있는 open() 역할을 하는 함수의 이름은 sys_open() 이다. open() 의 시스템콜 번호가 0x05 이므로 0x05 * 4 하면 해당 시스템콜에 대한 핸들러 함수 주소를 가지고 있는 System Call Vector Table 상의 위치가 나온다 여기서 커널 내부에 존재하는 sys_open() 함수로 가는 분기 명령이 존재 한다. c 라이브러리에서 open 시스템콜의 고유번호 '5'를 eax 레지스터에 저장하고 0x80 인터럽트를 발생시킨다. 인터럽트가 발생하면 사용자 모드에서 커널 모드로 전환된다. 커널은 IDT 에서 0x80 주소에 있는 system_call()을 찾는다. system_call() 함수에서는 ia32_sys_call_table 에서 시스템 콜 번호에 해당하는 함수를 호출한다.그러면 sys_open()를 수행하여 실질적인 open() 작업을 커널이 수행하게 된다.

34. task_struct 에서 super_block 이 하는 역할은 무엇인가 ?
super_block 은 루트 파일 시스템('/') 의 위치 정보를 가지고 있다.
ex. a 라는 디렉토리를 찾는다고 할때 root 디렉토리의 위치를 알아야 하는데 이는 root 디렉토리의 inode 를 알아야하고 inode 를 알가 위해서는 superblock 에 접근하여야 한다.
또한 super_block 은 파일시스템의 메타 데이터를 가지고 있으며, 파일 시스템마다 하나씩 존재한다.

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.

36. Linux Kernel 에서 Interrupt 를 크게 2 가지로 분류한다. 그 2 가지에 대해 각각 기술하고 간략히 설명하시오.
1) 외부 인터럽트: CPU 외부의 실제 Device 가 발생시키는 인터럽트
2) 내부인터럽트 : CPU 내에서 일어나는 인터럽트,‘ 예외처리’라고도 함
ex. 0 으로 나누는 연산, 세그멘테이션 결함, 페이지 결함, 보호 결함, 시스템 호출

37. 내부 인터럽트는 다시 크게 3 분류로 나눌 수 있다. 3 가지를 분류하시오.
trap, fault, abort

38. 35 번에서 분류한 녀석들의 특징에 대해 기술하시오.

1) fault : fault 를 일으킨 명령어 주소를 복귀주소로 eip 에 저장하고 Fault 에 대한 처리를 진행하고 다시 돌아와서 Fault 가 났던 부분을 다시 한 번 더 수행한다. (ex. page fault)
2) trap : trap 을 일으킨 명령어 다음 주소를 복귀주소로 eip 에 저장 하고 System Call 에 대한 수행을 처리한 이후 System Call 바로 아래 주소부터 실행을 시작한다 (함수 호출의 복귀 주소와 비슷한 형태)(ex. 시스템 콜)
3) abort: 심각한 오류에 해당, 저장하지 않음 , 현재 태스크 강제 종료 (ex. divide by zero)

39. 예로 모니터 3 개를 쓰는 경우 양쪽에 모두 인터럽트를 공유해야 한다. Linux Kernel 에서는 어떠한 방법을 통해 이들을 공유하는가 ?
외부 인터럽트의 경우 32 ~ 255 까지의 번호를 사용한다. 여기서 128(0x80)에 해당하는 System Call 만은 제외한다.
idt_table 에서 128 을 제외한 32 ~ 255 사이의 번호가 들어오면 실제 HW Device 다.
여기서 같은 종류의 HW Device 가 들어올 수 있는데 그들에 대해서 Interrupt 를 공유하게 하기 위해 irq_desc 라는 Table 을 추가로 두고 active 라는 것으로 Interrupt 를 공유하게 한다.
(do_IRQ()함수가 호출되면 외부인터럽트 번호를 가지고 irq_table 을 인덱싱하여 해당 외부인터럽트 번호와 관련된 irq_desc_t 자료구조를 찾는다. 이 자료구조 안에는 하나의 인터럽트를 공유할 수 있는 action 이라는 자료구조의 리스트를 유지하고 있다, 이 리스트를 통하여 단일 인터럽트 라인을 공유할 수 있다)

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 //ia32_syscall_table 명칭변경??
ARM 의 경우에는 __vectors_start + 0x1000 에 해당

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 은 r7 레지스터를 사용한다.

42. Paging Mechanism 에서 핵심이 되는 Page Directory 는 mm_struct 의 어떤 변수가 가지고 있는가 ?
pgd (mm_struct > pgd 에 페이지 디렉터리의 시작점 주소를 저장하고 있다)

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

(인텔 CPU 는 가상주소를 물리주소로 변환하기 위해 페이지테이블의 시작점, 즉 page directory 를 CR3 레지스터에 담아둔다)
Intel 의 경우엔 CR3, ARM 의 경우엔 CP15

44. 커널 내부에서 메모리 할당이 필요한 이유는 무엇인가 ?
1) 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.
2) 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.
ex. 커널 스택 8KB(thread_union > kernel_stack)

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.

48. module_init() 함수 호출은 언제 이루어지는가 ?
module_init() 함수는 insmod 명령어로 Device Driver Module 을 부착시킬때 동작한다.

49. module_exit() 함수 호출은 언제 이루어지는가 ?
module_exit() 함수는 rmmod 명령어로 Driver Module 을 탈착시킬때 동작한다.

50. thread_union 에 대해 기술하시오.
- 태스크가 생성이 되면 각각의 태스크마다 task_struct 구조체와 8KB 의 스택(thread_union) 이 할당된다. 즉, thread_union 은 태스크당 할당되는 8KB 의 스택을 나타낸다.
- 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.
또는 task_struct > files_struct > file > inode > i_rdev 에 주번호와 부번호 저장

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.

1) 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.
2) 동적으로 할당할 경우에는 alloc_chrdev_region() 이 동작한다. 이때 file_operations 를 Wrapping 할 구조체를 전달하고 Major Number 와 장치명을 전달한다.
3) chrdevs 배열에 Major Number 에 해당하는 Index 에 file_operations 를 Wrapping 한 구조체를 저장해둔다.
4) 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.
5) 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.

6) 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!

[driver.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 <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)
{
#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)
{
#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);
	write_ret[0] = 0;

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

	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);

[user.c]
#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;
    }

    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.
- CPU 의 실행 속도를 빠르게 하기 위해 여러 기술들이 등장했고, 비순차 실행은 그 중 하나이다. 과거에는 CPU 는 명령어가 들어오는 순서대로 그 명령어를 처리했는데, 이를 순차 실행(In-order) 이라고 한다.
하지만, 명령어의 실행 순서를 바꿈으로써 실행 시간을 단축시킬수 있고 이를 OoO(Out-of-Order)라 한다.
- 데이터 의존성이 존재하게되면 어쩔 수 없이 지연(Stall)이 발생하게 된다. 이러한 Stall 을 최소화하기 위해 앞서 실행했던 코드와 의존성이 없는 코드를 찾아서 아래의 의존성이 있는 코드 위로 끌어올려서 실행하는 방식이다.

56. Compiler 의 Instruction Scheduling 에 대해 기술하라. // ??
위의 OoO 개념으로 Compile-Time 에 Compiler Level 에서 직접 수행한다. Compiler 가 직접 Compile-Time 에 분석하므로 최적화의 수준이 높으면 높을수록 Compile Timing 이 느려질 것이다.
그러나 성능만큼은 확실하게 보장할 수 있을 것이다.

57. CISC Architecture 와 RISC Architecture 에 대한 차이점을 기술하라.
- CISC 와 RISC 의 가장 큰 차이점은 다이 사이즈다. CISC 는 다이 사이즈가 크다보니 여러가지 기능 유닛들을 포함할 수 있다. 그러다보니 상대적으로 전력을 많이 소비하게 된다. RISC 는 다이 사이즈가 작다보니 CISC 처럼 여러 기능 유닛들을 집어넣을 수 없다. 그러한 이유로 전력 소모량도 상대적으로 적다.
(CISC 는 명령어의 길이가 가변적으로 구성된 것이다. 한 명령어의 길이를 줄여 디코딩 속도를 높이고 최소크기의 메모리 구조를 가진다. RISC 는 CPU 에서 수행하는 동작 대부분이 몇개의 명령어 만으로 가능하다는 사실에 기반하여 구현된것이다. 적은수의 명령어로 명령어 집합을 구성하며 기존의 복잡한 명령은 보유한 명령어를 조합해서 사용한다)
- 또한 Dynamic 하게 서포팅을 해주는 것이 없으므로 Compiler 의 최적화 알고리즘은 RISC 쪽이 더 신경을 많이 써야 한다. (CISC 쪽은 HW 가 어느정도 커버 해주므로) //??

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 어느 정도의 Instruction Scheduling 을 HW 의 힘만으로 수행할 수 있다. 이러한 것을 무엇이라 부르는가 ?
Dynamic Instruction Scheduling // ??
/*싱글 사이클 오퍼레이션일때는 기본 파이프라인이 순서적 동작(In-order issue)로 부터 시작 되었음, 기본 파이프라인이 멀티 사이클 오퍼레이션으로 확장되고 멀티플 이슈가 가능 해짐에 따라 다이나믹 스케줄링이 필요해짐.
다이니믹 스케줄링은 Out-of-order 로 동작을 가능하게 함.
*/

59. Pipeline 이 깨지는 경우에 대해 자세히 기술하시오.
분기(jmp or branch)가 발생했을 경우다. 분기가 발생하게되면 바로 아래에서 Fetch 및 Decode 해오던 값들이 무의미해지기 때문이다.

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. //??

하이퍼 스레딩은 한개의 물리 코어에 스레드를 추가해 가상의 논리 코어를 추가해주며 기술 분류는 SMT 로 나뉜다.
폴락의 법칙과 관련되어 있다.
하이퍼스레딩은 명령어 파이프라인 대역폭 개선으로 명령어쪽 병목을 완하하여 다중작업을 개선하는 방법이다. 즉 CPU 자원 중 남는 요소들을 하이퍼스레딩이라는 기술을 넣어줌으로써 CPU 에 스레드(가상 논리코어)가 추가되고 그로 인해 CPU 의 자원을 효율적으로 분배해 CPU 멀티 성능을 올려주는 기술이라 할 수 있다.

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.
참고: http://www.makelinux.net/kernel_map/
(kernel_map.jpg)

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

ex. 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.

63. 파일의 종류를 확인하는 프로그램을 작성하시도록 하시오.

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

int main(int argc, char **argv)
{
        struct stat s;
        char ch;

        stat(argv[1], &s);

        if(S_ISDIR(s.st_mode))
```

```
		ch = 'd';
	if(S_ISREG(s.st_mode))
		ch = '-';
	if(S_ISFIFO(s.st_mode))
		ch = 'p';
	if(S_ISLNK(s.st_mode))
		ch = 'l';
	if(S_ISSOCK(s.st_mode))
		ch = 's';
	if(S_ISCHR(s.st_mode))
		ch = 'c';
	if(S_ISBLK(s.st_mode))
		ch = 'b';

	printf("%c\n", ch);

	return 0;
}
```

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

[clnt.c]

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

typedef struct sockaddr_insi;
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_proc(int sock, d *buf)
{
	char msg[32] = {0};

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

		if(!len)
			return;
```

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

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;
```

```
}

[serv.c]
#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_insi;
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);

		while((len = read(clnt_sock, buf, BUF_SIZE)) != 0)
		{
			printf("%s\n", buf);
			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.

[recv.c]

```
#include "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;
}
```

[send.c]

```
#include "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.h]

```
#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);
```

[shmlib.c]

```
#include "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);
}
```

66. 자신이 사용하는 리눅스 커널의 버전을 확인해보고
https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/$(자신의 버전).tar.gz 를 다운받는다.
이 압축된 파일을 압축 해제하고 task_struct 를 찾아보도록 한다. 일련의 과정을 기술하면 된다.

```
cd ~
mkdir kernel
cd kernel
wget https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/linux-4.4.tar.gz
tar zxvf linux-4.4.tar.gz
cd linux-4.4
ctags -R; mkcscsope.sh
vi -t task_struct
```

67. Multi-Tasking 의 원리에 대해 서술하시오. (Run Queue, Wait Queue, CPU 에 기초하여 서술하시오)
CPU 마다 RQ, WQ 가 1 개씩 존재한다.
active 배열과 expired 배열이 존재해서 어떤 우선순위의 Process 가 현재 올라가 있는지 bitmap 을 체크하게되며 queue 에는 만들어진 task_struct 가 들어가 있게 된다.
즉, bitmap 을 보고 우선순위에 해당하는 것이 존재하면 빠르게 queue[번호]로 접근해서 해당 task_struct 를 RQ 에 넣거나 주어진 Time Slice 가 다했는데 수행할 작업이 남아 있다면 RQ 에서 WQ 로 집어넣는등의 작업을 수행한다.
결국 Scheduling 이란 작업이 Multi-Tasking 을 지원하는데 있어 핵심인 기술이다.
여러 Task 들을 동시다발적으로 동작하는것처럼 보이게하는 트릭이라 할 수 있겠다.

68. 현재 삽입된 디바이스 드라이버의 리스트를 보는 명령어는 무엇인가 ?
lsmod

69. System Call Mechanism 에 대해 기술하시오.
- System Call 은 User 가 Kernel 에 요청하여 작업을 처리할 수 있도록 지원한다.
내부적으로 굉장히 복잡한 과정을 거치지만 User 가 read()등만 호출하면 실질적인 복잡한 모든 과정은 Kernel 이 수행하게 되어 편의성을 제공해준다.

70. Process 와 VM 과의 관계에 대해 기술하시오.
모든 Process 는 자신의 고유한 공간으로 가상의 4GB(32bit 프로세스의 경우 가상메모리 4GB 즉 4GB 의 주소공간을 가짐)를 가지고 있다. 실제 이 공간을 모두 사용하지 않으며 Demand on Paging 에 따라 필요한 경우에만 실제 물리 메모리 공간을 Paging Mechanism 을 통해 할당받아 사용한다. 이는, 한정된 물리 메모리의 한계를 극복하고자 디스크와 같은 느린 저장장치를 활용하여 애플리케이션들이 더 많은 메모리를 활용할 수 있게 해 주는 것이다.
(Page 란, 가상 메모리를 사용하는 최소 크기 단위이다)

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];
   fd = open(argv[1], O_RDONLY);
   ret = read(fd, buf, sizeof(buf));
   for(i = 0, index = 1; buf[i]; i++)
   {
      if(buf[i] == '\n')
      {
         save_area[index] = i;
         index++;
      }
   }
   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;
```

```
}
```

72. 디렉토리 내에 들어 있는 모든 File 들을 출력하는 Program 을 작성하시오.

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

void recursive_dir(char *dname);

int main(int argc, char **argv)
{
   recursive_dir(".");
   return 0;
}

void recursive_dir(char *dname)
{
   struct dirent *p;
   struct stat buf;
   DIR *dp;
   chdir(dname);
   dp = opendir(".");
   printf("\t%s : \n", dname);
   while(p = readdir(dp))
      printf("%s\n", p -> d_name);
   rewinddir(dp);
   while(p = readdir(dp))
   {
      stat(p -> d_name, &buf);
      if(S_ISDIR(buf.st_mode))
         if(strcmp(p -> d_name, ".") && strcmp(p -> d_name, ".."))
            recursive_dir(p -> d_name);
   }
   chdir("..");
   closedir(dp);
}
```

73. Linux 에서 fork()를 수행하면 Process 를 생성한다. 이때 부모 프로세스를 gdb 에서 디버깅하고자하면 어떤 명령어를 입력해야 하는가 ?

set follow-fork-mode child
// (gdb) set follow-fork-mode parent 가 맞는것 같은데??
자식을 디버깅 하고자 하는 경우에는 set follow-fork-mode child 지만
부모의 경우는 디폴트 설정 값이므로 특별한 명령어를 줄 필요가 없다.

74. C.O.W Architecture 에 대해 기술하시오.

(Copy On Write) 실질적인 쓰기가 발생하는 시점에 복사를 하겠다는 것이다.
fork 시에 부모가 갖고 있는 모든 것을 복제하기에는 시스템 자원의 소모가 크다. 따라서 fork 시에 프로세스가 사용하는 모든 메모리 공간을 일일히 모두 복사하지 않고, 실제 쓰기가 발생하는 시점에 실질적인 복사(새로운 영역을 할당)를 수행한다. 그렇게 함으로써 시스템 전체의 부하를 상대적으로 많이 줄일 수 있다.

75. Blocking 연산과 Non-Blocking 연산의 차이점에 대해 기술하시오.

Blocking/NonBlocking 은 호출되는 함수가 바로 리턴하느냐 마느냐에 따라 구분된다.
- Blocking 연산의 경우 호출된 함수가 자신의 작업을 모두 마칠 때까지 호출한 함수에게 제어권을 넘겨주지 않고 대기하게 만듬
즉, CPU 를 계속 잡고 있으므로 다른 일을 수행할 수 없다.
ex. 예를 들어, 소켓이 blocking 일 경우 Server 가 Client 의 메시지 요청을 받기 위해 read 에서 기다리게 되는 경우를 들 수 있다. Client 가 write 하기 전에는 read 에서 빠져나오지 못한다
- Non-Blocking 연산의 경우 호출된 함수가 바로 리턴해서 호출한 함수에게 제어권을 넘겨주고, 호출한 함수가 다른 일을 할 수 있는 기회를 준다.
즉, CPU 를 잡고 있지 않고 이런 일을 할꺼니까 준비되면 알아서 해줘라는 식으로 처리를 하기 때문에 Blocking 연산에 비해 성능면에서 뛰어난다.
- 그러나 반드시 Blocking 연산이 필요한 경우 또한 존재한다.

76. 자식이 정상 종료되었는지 비정상 종료되었는지 파악하는 프로그램을 작성하시오.

```
/*
1) 자식 프로세스 정상 종료 : WIFEXITED(status) 매크로가 true 를 반환
- 자식 프로세스가 exit 에 넘겨준 인자값: WEXITSTATUS(status)
2) 자식 프로세스 비정상 종료 : WIFSIGNALED(status) 매크로가 true 를 반환
- 비정상 종료 이유를 WTERMSIG(status) 매크로를 사용하여 구할 수 있음

ex.
if(WIFEXITED(status))
{
   printf("wait : 자식 프로세스 정상 종료 %d\n",WEXITSTATUS(status));
}
else if(WIFSIGNALED(status))
{
   printf("wait : 자식 프로세스 비정상 종료 %d\n",WTERMSIG(status));
}
*/
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
   pid_t pid;
   int status;
   if((pid = fork()) > 0)
   {
      wait(&status);
      if((status & 0xff) == 0)
         printf("(정상 종료)status : 0x%x\n", WEXITSTATUS(status));
      else if(((status >> 8) & 0xff) == 0)
         printf("(비정상 종료)status : 0x%x\n", status & 0x7f); //??
   }
   else if(pid == 0)
      abort();
   else
   {
      perror("fork() ");
      exit(-1);
   }
```

```
    return 0;
}
```

77. 데몬 프로세스를 작성하시오. 잠시 동안 데몬이 아니고 영구히 데몬이 되게 하시오.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int daemon_init(void)
{
    int i;
    if(fork() > 0)
        exit(0);
    setsid();
    chdir("/");
    umask(0);
    for(i = 0; i < 64; i++)
        close(i);
    signal(SIGCHLD, SIG_IGN);
    return 0;
}

int main(void)
{
    daemon_init();
    /* 데몬에서 동작시키고자 하는 작업을 이곳에 작성한다.*/
    for(;;)
        ;
    return 0;
}
```

78. SIGINT 는 무시하고 SIGQUIT 을 맞으면 죽는 프로그램을 작성하시오.

```
#include <signal.h>
#include <unistd.h>

int main(void)
{
    signal(SIGINT, SIG_IGN);
    for(;;)
        pause();
    return 0;
}
```

79. goto 는 굉장히 유용한 C 언어 문법이다. 그러나 어떤 경우에는 goto 를 쓰기가 힘든 경우가 존재한다. 이 경우가 언제인지 기술하고 해당하는 경우 문제를 어떤식으로 해결 해야 하는지 프로그래밍 해보시오.

[1] goto 는 스택을 해제할 수 없기 때문에 main()의 err 로 갈 수 없다.

[2] Setjmp(), longjmp() 시스템 콜로 해결 가능하다. //???

[1]

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

```
#include <stdlib.h>
#include <fcntl.h>
void func()
{
	goto err;
}int main()
{
	func();
	return 0;
err:
	perror("doesn't work!\n");
	exit(-1);
}
```

[2-1]

```
#include <fcntl.h>
#include <stdlib.h>
#include <setjmp.h>
#include <stdio.h>
jmp_buf env;
void test(void)
{
	longjmp(env,1);
}
int main(void)
{
	int ret;
	if((ret=setjmp(env))==0)
		test();
	else if( ret > 0)
		printf("error\n");
	return 0;
}
```

[2-2]

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

jmp_buf env;

void error_label(int flag)
{
   if(flag == 1)
      longjmp(env, 1);
}

int main(void)
{
   int ret;
   int error_flag = 1;
   if(ret = setjmp(env) == 0)
      error_label(error_flag);
   else
      printf("Error\n");
   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.

81. stat(argv[2], &buf)일때 stat System Call 을 통해 채운 buf.st_mode 의 값에 대해 기술하시오.
Stat System Call 실행시 argv[2] 인자로 받은 파일의 상태를 struct stat buf 에 저장한다. buf 구조체에는 여러가지 정보가 저장되게 되는데 그중 st_mode 필드즉, buf.st_mode 에는 파일의 종류와 파일에 대한 접근권한 정보가 저장된다(파일의 종류 4 비트와 setuid, setgid, sticky bit, 그리고 rwx 가 3 개씩 존재한다)

82. 프로세스들은 최소 메모리를 관리하기 위한 mm, 파일 디스크립터인 fd_array, 그리고 signal 을 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 signal 을 주고 받을 필요가 있기 때문에 signal 이 필요하다.

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)
	{
		printf("Usage: exe dir_name\n");
		exit(-1);
	}

	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)
{
	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());
        //dname[i] = rand_name();
    }
}

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

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

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

    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());
	}
}

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("..");
	}
}

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("..");
        }
}

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

        dp = opendir(".");

        while(p = readdir(dp))
        {
                stat(p->d_name, &buf);
                len = strlen(p->d_name);

                for(i = 0; i < len; i++)
                {
                        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.
[clnt.c]

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

#define BUF_SIZE		128
#define NAME_SIZE		32

typedef struct sockaddr_in	si;
typedef struct sockaddr *	sp;

char name[NAME_SIZE] = "[DEFAULT]";
char msg[BUF_SIZE];

void err_handler(char *msg)
{
```

```
	fputs(msg, stderr);
	fputc('\n', stderr);
	exit(1);
}

void *send_msg(void *arg)
{
	int sock = *((int *)arg);
	char name_msg[NAME_SIZE + BUF_SIZE];

	for(;;)
	{
		fgets(msg, BUF_SIZE, stdin);

		if(!strcmp(msg, "q\n") || !strcmp(msg, "Q\n"))
		{
			close(sock);
			exit(0);
		}

		sprintf(name_msg, "%s %s", name, msg);
		write(sock, name_msg, strlen(name_msg));
	}

	return NULL;
}

void *recv_msg(void *arg)
{
	int sock = *((int *)arg);
	char name_msg[NAME_SIZE + BUF_SIZE];
	int str_len;

	for(;;)
	{
		str_len = read(sock, name_msg, NAME_SIZE + BUF_SIZE - 1);

		if(str_len == -1)
			return (void *)-1;

		name_msg[str_len] = 0;
		fputs(name_msg, stdout);
	}

	return NULL;
}

int main(int argc, char **argv)
{
	int sock;
	si serv_addr;
	pthread_t snd_thread, rcv_thread;
	void *thread_ret;

	if(argc != 4)
```

```
	{
		printf("Usage: %s <IP> <port> <name>\n", argv[0]);
		exit(1);
	}

	sprintf(name, "[%s]", argv[3]);
	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!");

	pthread_create(&snd_thread, NULL, send_msg, (void *)&sock);
	pthread_create(&rcv_thread, NULL, recv_msg, (void *)&sock);

	pthread_join(snd_thread, &thread_ret);
	pthread_join(rcv_thread, &thread_ret);

	close(sock);
	return 0;
}

[serv.c]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/epoll.h>

#define BUF_SIZE		128
#define MAX_CLNT		256

typedef struct sockaddr_in	si;
typedef struct sockaddr *	sp;

int clnt_cnt = 0;
int clnt_socks[MAX_CLNT];
pthread_mutex_t mtx;

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

```
void send_msg(char *msg, int len)
{
	int i;

	pthread_mutex_lock(&mtx);

	for(i = 0; i < clnt_cnt; i++)
		write(clnt_socks[i], msg, len);

	pthread_mutex_unlock(&mtx);
}

void *clnt_handler(void *arg)
{
	int clnt_sock = *((int *)arg);
	int str_len = 0, i;
	char msg[BUF_SIZE];

	while((str_len = read(clnt_sock, msg, sizeof(msg))) != 0)
		send_msg(msg, str_len);

	pthread_mutex_lock(&mtx);

	for(i = 0; i < clnt_cnt; i++)
	{
		if(clnt_sock == clnt_socks[i])
		{
			while(i++ < clnt_cnt - 1)
				clnt_socks[i] = clnt_socks[i + 1];
			break;
		}
	}

	clnt_cnt--;

	pthread_mutex_unlock(&mtx);
	close(clnt_sock);

	return NULL;
}

int main(int argc, char **argv)
{
	int serv_sock, clnt_sock;
	si serv_addr, clnt_addr;
	socklen_t addr_size;
	pthread_t t_id;

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

```
	pthread_mutex_init(&mtx, NULL);

	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, 10) == -1)
		err_handler("listen() error!");

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

		pthread_mutex_lock(&mtx);
		clnt_socks[clnt_cnt++] = clnt_sock;
		pthread_mutex_unlock(&mtx);

		pthread_create(&t_id, NULL, clnt_handler, (void *)&clnt_sock);
		pthread_detach(t_id);
		printf("Connected Client IP: %s\n", inet_ntoa(clnt_addr.sin_addr));
	}

	close(serv_sock);

	return 0;
}
```

89. 88 번 답에 도배를 방지 기능을 추가하시오.

[load_test.h]

```
#ifndef __LOAD_TEST_H__

#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

typedef struct timeval	tv;

double get_runtime(tv, tv);

#endif
```

[load_test.c]

```
#include "89_load_test.h"

double get_runtime(tv start, tv end)
```

```
{
	end.tv_usec = end.tv_usec - start.tv_usec;
	end.tv_sec = end.tv_sec - start.tv_sec;
	end.tv_usec += end.tv_sec * 1000000;

#if DEBUG
	printf("runtime = %lf sec\n", end.tv_usec / 1000000.0);
#endif

	return end.tv_usec / 1000000.0;
}
```

[serv.c]

```
#include "89_load_test.h"

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

#define  BUF_SIZE	128
#define MAX_CLNT	256

typedef struct sockaddr_in	si;
typedef struct sockaddr *	sp;

int clnt_cnt = 0;
int clnt_socks[MAX_CLNT];
int cnt[MAX_CLNT];
pthread_mutex_t mtx;

// Black List
int black_cnt;
char black_list[MAX_CLNT][16];

// Information of Thread
typedef struct __iot{
	int sock;
	char ip[16];
	int cnt;
} iot;

iot info[BUF_SIZE];

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

```
void proc_msg(char *msg, int len, int sock)
{
	int i;

	pthread_mutex_lock(&mtx);

	for(i = 0; i < clnt_cnt; i++)
	{
		if(info[i].sock == sock)
			continue;
		write(info[i].sock, msg, len);
	}

	pthread_mutex_unlock(&mtx);
}

void add_black_list(char *ip)
{
	pthread_mutex_lock(&mtx);
	strcpy(black_list[black_cnt++], ip);
	printf("black_list = %s\n", black_list[black_cnt - 1]);
	pthread_mutex_unlock(&mtx);
}

bool check_black_list(char *ip)
{
	int i;

	pthread_mutex_lock(&mtx);

	printf("Here\n");

	for(i = 0; i < black_cnt; i++)
	{
		if(!strcmp(black_list[i], ip))
		{
			pthread_mutex_unlock(&mtx);
			return true;
		}
	}

	pthread_mutex_unlock(&mtx);

	return false;
}

void *clnt_handler(void *arg)
{
	iot thread_info = *((iot *)arg);
	int len = 0, i;
	char msg[BUF_SIZE] = {0};

	tv start, end;
	double runtime = 0.0;
```

```
	double load_ratio;

	for(;;)
	{
		gettimeofday(&start, NULL);
		//len = read(clnt_sock, msg, sizeof(msg));
		len = read(thread_info.sock, msg, sizeof(msg));
		proc_msg(msg, len, thread_info.sock);
		gettimeofday(&end, NULL);

		runtime = get_runtime(start, end);

		load_ratio = 1.0 / runtime;
		printf("load_ratio = %lf\n", load_ratio);

		if(load_ratio > 1.5)
			thread_info.cnt++;

		if(thread_info.cnt > 10)
		{
			write(thread_info.sock, "You're Fired!!!\n", 16);
			add_black_list(thread_info.ip);
			goto end;
		}
	}

#if 0
	while((str_len = read(clnt_sock, msg, sizeof(msg))) != 0)
		proc_msg(msg, str_len, i);
#endif

end:
	pthread_mutex_lock(&mtx);

	for(i = 0; i < clnt_cnt; i++)
	{
		if(thread_info.sock == info[i].sock)
		{
			while(i++ < clnt_cnt - 1)
				info[i].sock = info[i + 1].sock;
			break;
		}
	}

#if 0
	for(i = 0; i < clnt_cnt; i++)
	{
		if(clnt_sock == clnt_socks[i])
		{
			while(i++ < clnt_cnt - 1)
				clnt_socks[i] = clnt_socks[i + 1];
			break;
		}
	}
#endif
```

```
	clnt_cnt--;
	pthread_mutex_unlock(&mtx);
	close(thread_info.sock);

	return NULL;
}

int main(int argc, char **argv)
{
	int serv_sock, clnt_sock;
	si serv_addr, clnt_addr;
	socklen_t addr_size;
	pthread_t t_id;
	int idx = 0;

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

	srand(time(NULL));

	pthread_mutex_init(&mtx, NULL);

	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, MAX_CLNT) == -1)
		err_handler("listen() error");

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

		printf("Check Black List\n");

		if(check_black_list(inet_ntoa(clnt_addr.sin_addr)))
		{
			write(clnt_sock, "Get out of my server!!!\n", 23);
			close(clnt_sock);
			continue;
		}
```

```
                pthread_mutex_lock(&mtx);

                info[clnt_cnt].sock = clnt_sock;
                strcpy(info[clnt_cnt].ip, inet_ntoa(clnt_addr.sin_addr));
                info[clnt_cnt++].cnt = 0;

                pthread_mutex_unlock(&mtx);

                //pthread_create(&t_id, NULL, clnt_handler, (void *)&clnt_sock);
                pthread_create(&t_id, NULL, clnt_handler, (void *)&info[clnt_cnt - 1]);
                pthread_detach(t_id);
                printf("Connected Client IP: %s\n", inet_ntoa(clnt_addr.sin_addr));
        }

        close(serv_sock);

        return 0;
}
```

90. 89 번 조차도 공격할 수 있는 프로그램을 작성하시오.

[clnt.c]

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

#define BUF_SIZE                128
#define NAME_SIZE               32

typedef struct sockaddr_insi;
typedef struct sockaddr *  sp;

char name[NAME_SIZE] = "[내가이긴다]";
char msg[2048];

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

void make_rand_str(char *tmp)
{
        int i, end = rand() % 7 + 3;

        for(i = 0; i < end; i++)
                tmp[i] = rand() % 26 + 65;
}
```

```
void *send_msg(void *arg)
{
	int sock = *((int *)arg);
	char msg2[] = "https://kr.battle.net/heroes/ko/  <<== 지금 당장 접속하세요!!\n";
	srand(time(NULL));

	char tmp1[32] = {0};

	for(;;)
	{
#if PASSIVE
		fgets(msg, BUF_SIZE, stdin);

		write(sock, msg, strlen(msg));
#endif
#if ATTACK
		make_rand_str(tmp1);

		printf("%s\n", msg);
		sprintf(msg, "%s %s %s", name, tmp1, msg2);
		printf("tmp1 = %s\n", tmp1);
		write(sock, msg, strlen(msg));
		sleep(5);
#endif
	}

	return NULL;
}

void *recv_msg(void *arg)
{
	int sock = *((int *)arg);
	char msg[NAME_SIZE + 2048];
	int str_len;

	for(;;)
	{
		str_len = read(sock, msg, NAME_SIZE + 2047);

		msg[str_len] = 0;
		fputs(msg, stdout);
	}

	return NULL;
}

int main(int argc, char **argv)
{
	int sock;
	si serv_addr;
	pthread_t snd_thread, rcv_thread;
	void *thread_ret;

	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");

	pthread_create(&snd_thread, NULL, send_msg, (void *)&sock);
	pthread_create(&rcv_thread, NULL, recv_msg, (void *)&sock);
	pthread_join(snd_thread, &thread_ret);
	pthread_join(rcv_thread, &thread_ret);

	close(sock);

	return 0;
}
```

91. 네트워크 상에서 구조체를 전달할 수 있게 프로그래밍 하시오.

[serv.c]

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

typedef struct sockaddr_insi;
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);

                        while((len = read(clnt_sock, (d *)&struct_data, BUF_SIZE)) != 0)
                        {
                                printf("struct.data = %d, struct.fdata = %f\n", struct_data.data, struct_data.fdata);
                                write(clnt_sock, (d *)&struct_data, len);
                        }

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

        return 0;
}

[clnt.c]
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

typedef struct sockaddr_insi;
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_proc(int sock, d *buf)
{
        for(;;)
        {
                int len = read(sock, buf, BUF_SIZE);
```

```
                if(!len)
                        return;

                printf("msg from serv: %d, %f\n", buf->data, buf->fdata);
        }
}

void write_proc(int sock, d *buf)
{
        char msg[32] = {0};

        for(;;)
        {
                fgets(msg, BUF_SIZE, stdin);

                if(!strcmp(msg, "q\n") || !strcmp(msg, "Q\n"))
                {
                        shutdown(sock, SHUT_WR);
                        return;
                }

                buf->data = 3;
                buf->fdata = 7.7;

                write(sock, buf, sizeof(d));
        }
}

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;
}
```

92. 91 번을 응용하여 Queue 와 Network 프로그래밍을 연동하시오.

[serv.c]

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

typedef struct sockaddr_insi;
typedef struct sockaddr *  sp;

typedef struct __q{
	int data;
	struct __q *link;
} q;

#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);
}

q *get_node(void)
{
	q *tmp;
```

```
	tmp = (q *)malloc(sizeof(q));
	tmp->link = NULL;
	return tmp;
}

bool enqueue(q **head, int data)
{
	if(!(*head))
	{
		*head = get_node();
		(*head)->data = data;
		return true;
	}

	return enqueue(&(*head)->link, data);
}

void print_queue(q *head)
{
	q *tmp = head;

	while(tmp)
	{
		printf("data = %d\n", tmp->data);
		tmp = tmp->link;
	}
}

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;
	q *head = NULL;
	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)
	{
		int data;
		bool test;
		close(serv_sock);

		while((len = read(clnt_sock, buf, BUF_SIZE)) != 0)
		{
			data = atoi(buf);
			test = enqueue(&head, data);
			printf("Now printing Queue\n");
			print_queue(head);

			if(test)
			{
				char msg[32] = "success\n";
				write(clnt_sock, msg, strlen(msg));
			}
			else
			{
				char msg[32] = "failure\n";
				write(clnt_sock, msg, strlen(msg));
			}
		}

		close(clnt_sock);
```

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

        return 0;
}

[clnt.c]
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

typedef struct sockaddr_insi;
typedef struct sockaddr *  sp;

#define BUF_SIZE                        32

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

void read_proc(int sock)
{
        char buf[32] = {0};

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

                if(!len)
                        return;

                printf("msg from serv: %s", buf);
        }
}

void write_proc(int sock)
{
        char msg[32] = {0};

        for(;;)
        {
                fgets(msg, BUF_SIZE, stdin);

                if(!strcmp(msg, "q\n") || !strcmp(msg, "Q\n"))
```

```
                {
                        shutdown(sock, SHUT_WR);
                        return;
                }

                write(sock, msg, strlen(msg));
        }
}

int main(int argc, char **argv)
{
        pid_t pid;
        int i, sock;
        si serv_addr;
        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);
        else
                read_proc(sock);

        close(sock);

        return 0;
}
```

93. Critical Section 이 무엇인지 기술하시오.
프로세스나 스레드가 동시 다발적으로 접근하여 정보를 공유할 수 있는 영역을 의미함 즉, 여러 task 들이 동시에 접근하여 정보가 꼬일 수 있는 공간을 말한다.
배타적 접근(한 순간에 하나의 쓰레드만 접근)이 요구되는 공유 리소스(ie. 전역변수)에 접근하는 코드 블록이다.

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

1) 우리가 만든 프로그램에서 fork() 가 호출되면
2) C Library 에 해당하는 glibc 의 __libc_fork() 가 호출됨
- 이 안에서 ax 레지스터에 시스템 콜 번호(fork 함수 고유번호)가 기록된다.
즉 sys_fork() 에 해당하는 시스템 콜 테이블의 번호가 들어가고
- 이후에 'int 0x80' 을 통해서 128 번 시스템 콜을 호출(트랩번호 0x80 으로 트랩요청)하게 된다.
3) 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.
4) sys_fork() 는 SYSCALL_DEFINE0(fork) 와 같이 구성되어 있다.

95. 리눅스 커널의 arch 디렉토리에 대해서 설명하시오.
CPU(HW) 의존적인 코드가 위치한 영역이다.
CPU 모델에 따라 바뀌어야 하는 코드들이 들어있다 → Context switching, Bootloader 등등

96. 95 번 문제에서 arm 디렉토리 내부에 대해 설명하도록 하시오.
인텔은 단순하지만, ARM 은 하위 호환이 안되고 다양한 반도체 벤더들이 개발을 하고 있기 때문에 디렉토리가 많고 복잡하다.
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

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

인텔 아키텍쳐에서 HW 인터럽트는 외부 인터럽트를 위한 공통 핸들러인 common interrupt 라는 어셈블리 루틴 레이블에서 처리한다.
이 안에서는 do_IRQ() 라는 함수가 같이 함께 일반적인 HW 인터럽트를 처리하기 위해 분발한다.
인터럽트 → 문맥 저장 → 인터럽트 핸들러 → 문맥 복원
(SAVE_ALL → do_IRQ() → ret_from_intr → RESTORE_ALL)

99. ARM 에서 System Call 을 사용할 때 사용하는 레지스터를 적으시오.
r7

100. 벌써 2 개월째에 접어들었다. 그동안 굉장히 많은 것들을 배웠을 것이다. 상당수가 새벽 3 ~ 5 시에 자서 2 ~ 4 시간 자면서 다녔다. 또한 수업 이후 저녁 시간에 남아서 9 시 ~ 10 시까지 공부를 한 사람들도 있다. 하루 하루에 대한 자기 자신의 반성과 그 날 해야할 일을 미루지는 않았는지 성찰할 필요가 있다. 그 날 해야할 일들이 쌓이고 쌓여서 결국에는 수습하지 못할정도로 많은 양이 쌓였을 수도 있다. 사람이란 것이 서 있으면 앉고 싶고 앉으면 눕고 싶고 누우면 자고 싶고 자면 일어나기 싫은 법이다. 내가 정말 죽을둥 살둥 이것을 이해하기 위해 열심히 했는지 고찰해보자! 2 개월간 자기 자신에 대한 반성과 성찰을 수행해보도록 한다. 또한 앞으로는 어떠한 자세로 임할 것인지 작성하시오.