TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사: Innova 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.

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.

3. 리눅스 커널은 운영체제(OS)다.
OS 가 관리해야 하는 제일 중요한 5 가지에 대해 기술하시오.
프로세스 관리, 메모리 관리, 파일 관리, CPU 관리, I/O 관리

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

→ C 언어, 공개되어 있다. 띠리서 사용자 임의대로 재구성이 가능하며 모놀리식 시스템이지만 모듈의 형태로 쉽게 시스템 기능을 추가할 수 있다

마이크로식으로 탈부착이 가능하기 때문에 최신버전 커널도 오래된 컴퓨터에서 작동이 가능하다

커널의 크기가 윈도우 커널보다 작으며

고성능 네트워크 장비에 사용되는 것으로 그 안정성이 검증되었다

대표적으로 구글 서버가 유닉스이다

TCP/IP, BSP 와 같은 강력한 네트워크를 지원한다

또한 스톨만의 GNU 덕분에 풍부한 소프트웨어를 가지고 있다.

6. 32 bit System 에서 User 와 Kernel 의 Space 구간을 적으시오.

가상주소공간은 유저영역과 커널영역으로 나뉜다

0x00000000~0x7FFFFFFF 까지가 유저 메모리 구간, 0x80000000~ 0xFFFFFFFF 까지가 커널영역으로

커널 영역 1GB 를 빼고는 나머지 3GB 는 유저 영역이다.

7. Page Fault 가 발생했을때 운영체제가 어떻게 동작하는지 기술하시오.

프로그램의 페이지가 물리 메모리에 부재하는 경우 이것을 페이지 폴트(Page Fault)라 한다.

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

9. 프로세스와 스레드를 구별하는 방법에 대해 기술하시오.

프로세스와 쓰레드를 구별하러면 tgid 를 사용한다.

pid 와 tgid 가 일치하면 프로세스이다

tgid 만 같고 pid 가 서로 다르면 쓰레드이다.

10. Kernel 입장에서 Process 혹은 Thread 를 만들면 무엇을 생성하는가 ?

Task_struct 구조체를 만든다

11. 리눅스 커널 소스에 보면 current 라는 것이 보인다. 이것이 무엇을 의미하는 것인지 적으시오.

커널 소스 코드와 함께 기술하시오.

```
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 */
```

```
SYSCALL_DEFINE0(getpid)
{
      return task_tgid_vnr(current);
}
```

→ current 는 현재 실행중인 태스크의 task_struct 를 가리키는 포인터 변수

threaed_union > thread_info > task_struct > thread_info *task 가 current 이다

12. Memory Management 입장에서 Process 와 Thread 의 핵심적인 차이점은 무엇인가 ?

프로세스는 자신만의 고유 공간과 자원을 할당 받아 사용하는데 비해 스레드는 프로세스 내에서 각각의 스택 공간을 제외한 나머지 공간과 시스템 자원을 공유한다.

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.

이때 CPU 레지스터가 바로 HW context 를 뜻한다.

이는 task_struct 밑의 struct thread_struct thread 에 저장된다

14. 리눅스 커널의 스케쥴링 정책중 Deadline 방식에 대해 기술하시오.

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

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

처리량이 적은 경우 유리하다

O(1)의 경우 데이터 개수와 상관없이 처리시간은 일정하다

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.

→ fork()시 기존에 있던 캐시를 활용하기 위해 2 번 CPU 에 위치하는 것이 좋다.

18. 앞선 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.

→ 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. 따라서 Task 는 0 이나 1 에 위치하게 된다.

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

→ UMA 는 모든 프로세서들이 하나의 버스를 공유하여 메모리 접근에 걸리는 시간이 일정하다.

NUMA 는 메모리에 접근하는 시간이 프로세서와 메모리에 따라서 달라진다.

Kernel 은 NUMA, UMA 모두를 노드라는 일관된 자료구조를 통해서 전체 물리메모리에 접근할 수 있도록 관리한다.

UMA 구조의 시스템에서는 한개의 노드가 존재하며 이 노드는 전역변수인 contig_page_data 를 통해 접근가능하다.

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
struct pglist_data __refdata contig_page_data = {
    .bdata = &bootmem_node_data[0]
};
EXPORT_SYMBOL(contig_page_data);
#endif
```

NUMA 구조의 시스템에서는 복수개의 노드가 존재하며 이는 구조체 pglist_data 를 통해 접근 가능하다.

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;
```

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

Dynamic Priority : 스케줄러가 각각의 프로세스에게 지정한 우선순위 값

→ 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.

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 ?

물리메모리의 최소단위는 페이지프레임이고 이를 SW 적으로 구현한 구조체는 struct page 이다.

23. Linux Kernel 은 외부 단편화와 메모리 부하를 최소화하기 위해 Buddy 할당자를 사용한다.

Buddy 할당자의 Algorithm 을 자세히 기술하시오.

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.

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. 다양한 크기의 캐시를 효율적으로 관리하기 위해 kmem_cache_t 라는 자료 구조를 만들어 두고, 새로운 캐시를 생성할때 kmem_cache_t 라는 구조체로부터 할당 받는다. ← 이 때 슬랩할당자로부터 할당받는다.

kmem_cache_t 구조체 크기의 객체를 담고 있는 cache 의 이름이 cache_cache 이며, cache_cache 를 이용해서 kmem_cache_t 를 위한 공간을 할당받아 다양한 캐시를 생성한다.

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 구조와 관련하여 이를 기록하는 변수들이 있다. task_struct > mm_struct 를 보면 unsigned long 으로 7 개의 변수가 선언되어 있다. Unsigned long start_code, end_code, start_data, end_data, start_brk, brk. start_stack;

→ 순서대로 텍스트영역, 데이터영역, 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)

task_struct > mm_struct > vm_area_struct 구조체에서 가상메모리 공간 중 같은 속성을 가지고 연속인 영역 (region)을 관리한다. ← 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 ?

task_struct > mm_struct 에서 current-> mm 은 현재 실행되는 프로세스의 메모리구조를 가리킨다.

fork()시 copy_mm()에 의해 프로세스 주소 공간이 생성되고 이때, 새로운 프로세스의 모든 페이지 테이블과 메모리 디스크립터를 설정하게 된다.

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.

커널 스택이 존재하는 이유는 다음과 같다

1) Kernel Mode 로 전환된 프로세스는 언젠간 다시 User Mode 로 되돌아가야 하므로, User Mode 로 전환하기 위해 필요한 정보 중 일부를 저장한다.

2) Kernel Mode 에서 함수를 호출하게 되면 그 함수의 지역변수는 Kernel Mode 스택에서 할당된다.

커널스택은 유저 스택보다 크기가 훨씬 작으며 ARM 의 경우 8KB, 인텔 CPU 의 경우 16KB 를 고정할당받는다.

29. VM(가상 메모리)와 PM(물리 메모리)를 관리하는데 있어 VM 을 PM 으로 변환시키는 Paging Mechanism 에 대해 Kernel 에 기반하여 서술하시오.

가상주소를 물리주소로 변환하는 방법: 페이지 테이블(page table) 이용한다. 페이지 테이블은 페이지 프레임에 적재할 때 페이지 테이블 함께 생성된다.

가상 주소 접근할 때, 가상 주소를 가상 메모리의 최소 단위인 페이지(4KB)로 나눈다. 이때 몫은 페이지 테이블의 엔트리를 탐색하는 인덱스이고 나머지는 페이지 프레임 내에 옵셋(offset)이다.

리눅스는 주소 변환을 위한 페이징을 여러 단계로 구분한다.

pgd: 가상 주소를 물리 주소로 변환하기 위한 시작점으로 페이지 프레임 번호 → pgd 를 통해 가상 주소 중 일부를 이용하여 pmd 를 얻는다 → pmd 를 통해 가상 주소 중 일부를 이용하여 인덱싱하여 pte 를 얻는다 →: PTE 테이블에서 가상 주소 일부를 이용하여 인덱싱하면 실제 접근하게 될 페이지 프레임 주소 얻을 수 있다.

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

대부분의 CPU 는 가상 주소로부터 물리 주소로의 변환을 담당하는 별도 하드웨어가 있다. 이것이 MMU 이다.(MMU 에게 페이지 테이블의 시작점(pgd)과 변환하고자 하는 가상 주소를 입력으로 넣어주면 된다)
MMU 의 주요 핵심 기능은 가상 메모리 주소를 실제 메모리 주소로 변환하며, 메모리 보호, 캐시 관리, 버스 중재이다.

31. 하드디스크의 최소 단위를 무엇이라 부르고 그 크기는 얼마인가 ?

섹터, 512Byte

32. Character 디바이스 드라이버를 작성할 때 반드시 Wrapping 해야 하는 부분이 어디인가 ?

(Task 구조체에서 부터 연결된 부분까지를 쭉 이어서 작성하라)

33. 예로 유저 영역에서 open 시스템 콜을 호출 했다고 가정할 때 커널에서는 어떤 함수가 동작하게 되는가 ?

실제 소스 코드 차원에서 이를 찾아서 기술하도록 한다.

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 시스템 콜의 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 ?

수퍼 블록(super block)은 파일시스템의 전체적인 정보와 '/'의 위치를 기억한다. 파일 시스템마다 하나씩 존재한다.

superblock 은 다음과 같이 사용할 수 있다.

a 라는 디렉토리를 찾는다고 할때 root 디렉토리의 위치를 알아야 하는데 이는 root 디렉토리의 inode 를 알아야하고 inode 를 알가 위해서는 superblock 에 접근하여야 한다.

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

파일 시스템과 사용자 태스크 사이에 가상적인(Virtual)층 도입으로 리눅스에서 다양한 파일 시스템 지원이 가능하다. 즉. 상위 계층에서 open, read()와 같은 단일한 함수를 통해 파일 시스템에 접근하는 것을 가능하게 해준다.

만일, 사용자 태스크가 b.txt 라는 파일을 인자로 open()시스템 콜을 호출했다면, VFS 는 어떤 파일 시스템에 속해있는지를 판단하고 b.txt 의 정보를 담을 구조체를 생성한다.

이 구조체를 인자로 하여 파일시스템 내부에 구현되어 있는 고유한 open()함수를 호출한다. 그러면 해당 파일 시스템은 자신의 inode 테이블을 뒤져서 b.txt 의 inode 를 찾아내고 inode 의 정보를 VFS 가 넘긴 구조체에 저장한 후 리턴하게 된다. VFS 는 이 구조체의 내용을 바창으로 사용자 태스크에게 필요한 내용을 전달한다.

36. Linux Kernel 에서 Interrupt 를 크게 2 가지로 분류한다. 그 2 가지에 대해 각각 기술하고 간략히 설명하시오.

인터럽트는 원인에 따라 2 가지로 구분한다

1) 외부 인터럽트: 현재 수행 중인 태스크와 관련없는 주변 장치에서 발생, 비동기적, 하드웨어적

2) 트랩 : 현재 수행 중인 태스크와 관련, 동기적, 소프트웨어적 → '예외처리'라고도 함

ex. 0 으로 나누는 연산, 세그멘테이션 결함, 페이지 결함, 보호 결함, 시스템 호출

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 eip 에 저장 (ex. page fault)

2) trap : trap 을 일으킨 명령어 다음 주소를 eip 에 저장 (ex. 시스템 콜)

3) abort: 저장하지 않음 , 현재 태스크 강제 종료 (ex. divide by zero)

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

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.

ia32_syscall_table

41. 38 에서 User Space 에서 System Call 번호를 전달한다.

Intel Machine 에서는 이를 어디에 저장하는가 ?

또한 ARM Machine 에서는 이를 어디에 저장하는가 ?

ARM 에서는 시스템콜을 전달할 때 r7 레지스터를 사용한다.

x86 에서는 eax 레지스터에 시스템콜번호를 넣는다.

42. Paging Mechanism 에서 핵심이 되는 Page Directory 는 mm_struct 의 어떤 변수가 가지고 있는가 ?

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. ARM 에서 이 역할을 하는 레지스터는 TTBR 이다.

44. 커널 내부에서 메모리 할당이 필요한 이유는 무엇인가 ?

커널도 프로그램이므로 스택이 필요하다

커널 스택 8KB(thread_union > kernel_stack)

45. 메모리를 불연속적으로 할당하는 기법은 무엇인가 ?

페이징과 세그멘테이션을 이용하여 프로그램을 불연속적으로 할당한다.

cf. valloc()은 커널에 불연속적인 메모리를 할당한다

46. 메모리를 연속적으로 할당하는 기법은 무엇인가 ?

3 가시 방식이 존재한다. 최초적합. 최적적합, 최악적합

cf. kalloc()은 커널에 연속적인 메모리를 할당한다.

47. Mutex 와 Semaphore 의 차이점을 기술하시오.

세마포어(Semaphore): 공유된 자원의 데이터를 여러 프로세스가 접근하는 것을 막는 것

뮤텍스(Mutex): 공유된 자원의 데이터를 여러 쓰레드가 접근하는 것을 막는 것

48. module_init() 함수 호출은 언제 이루어지는가 ?

커널에 모듈 insert 시 호출된다

49. module_exit() 함수 호출은 언제 이루어지는가 ?

커널에서 모듈 remove 시 호출된다

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.

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 > 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.

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.

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!

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

비순차적 명령어 처리(Out-of-order execution, 줄여서 OoOE) 또는 비순차적 실행은 고성능 마이크로프로세서가 특정한 종류의 지연으로 인해 낭비될 수 있는 명령 사이클을 이용하는 패러다임이다.

56. Compiler 의 Instruction Scheduling 에 대해 기술하라.

파이프라인이 깨지는 것을 최소화 하는 방향으로 컴파일러는 instruction scheduling 을 수행한다

57. CISC Architecture 와 RISC Architecture 에 대한 차이점을 기술하라.

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 구현된 것이다. 적은수의 명령어로 명령어 집합을 구성하며 기존의 복잡한 명령은 보유한 명령어를 조합해서 사용한다.

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 ?

59. Pipeline 이 깨지는 경우에 대해 자세히 기술하시오.

분기시 파이프라인이 깨진다.

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 ?

61. Intel 의 Hyper Threading 기술에 대해 상세히 기술하시오.

- fork()가 회로로 구현되어 있다 , 실제 CPU 는 4 개인데 8 개처럼 동작하게 만든것
- 코어에서 스레드를 실행하고 있는 중에, 다른 스레드가 실행되기를 원하는데 그 기능이 코어에서 놀고있는 기능이면 동시에 실행해 준다. 두 스레드가 같은 기능을 요청하면 코어에서는 하나의 스레드만 실행 된다.

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.

63. 파일의 종류를 확인하는 프로그램을 작성하시도록 하시오.

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

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

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

67. Multi-Tasking 의 원리에 대해 서술하시오.
    (Run Queue, Wait Queue, CPU 에 기초하여 서술하시오)

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

69. System Call Mechanism 에 대해 기술하시오.

사용자 프로세서가 시스템콜을 요청하면 제어가 커널로 넘어간다. (사용자모드 -> 커널모드)

커널은 내부적으로 각각의 시스템콜을 구분하기 위해 기능별로 고유번호를 할당해 놓는다.

해당번호는 커널내부에 제어루틴을 정의한다.

커널은 요청받은 싯템 콜에 대응하는 기능번호를 확인한다.

커널은 그 번호에 맞는 서비스 루틴을 호출하게 된다.

서비스 루틴을 모두 처리하고나면 커널 모드에서 사용자 모드로 다시 넘어온다

70. Process 와 VM 과의 관계에 대해 기술하시오.

모든 프로세스는 자신만의 가상 주소 공간을 가지고 있다.

32 비트/64 비트 프로세스는 각 비트수에 맞게 최대 4GB/16EB 의 주소 공간을 가진다.

한정된 물리 메모리의 한계를 극복하고자 디스크와 같은 느린 저장장치를 활용해,

애플리케이션들이 더 많은 메모리를 활용할 수 있게 해 주는 것이 가상 메모리이다.

Page 란, 가상 메모리를 사용하는 최소 크기 단위이다.

최근의 윈도우 운영체제는 모두 4096 (4KB)의 페이지 크기를 사용한다.

71. 인자로 파일을 입력 받아 해당 파일의 앞 부분 5 줄을 출력하고

추가적으로 뒷 부분의 5 줄을 출력하는 프로그램을 작성하시오.

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

```
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
// 메인 앞에 함수의 프로토타입을 기술한다
void resursive_dir(char* dname);
int main(int argc, char* argv[])
{
	recursive_dir("."); // 현재 디렉토리에 대해 재귀함수 호출
	return 0;
}
void recursive_dir(char* dname) // 재귀함수 인자로 현재 디렉토리 "." 전달
{
	struct dirent* p;
	struct stat buf;
	DIR* dp; // 디렉토리포인터 dp
	chdir(dname); // 경로의 위치를 바꿔준다 // dname 을 출력하면 . 뜬다
	dp=opendir(".");
```

```
    printf("\t %s : \n",dname);
    while(p=readdir(dp)) // 디렉토리 내의 파일 리스트 없을때까지 출력
        printf("%s\n", p->d_name);
    rewinddir(dp); // 파일포인터를 다시 처음으로 이동하여 dp 를 다시 사용할 수 있도록 한다
    while(p=readdir(dp)) // 디렉토리 내의 파일 리스트 없을때까
    {
        stat(p->d_name,&buf);// 시스템 콜 stat: int stat(const char *path, struct stat *buf);
        if(S_ISDIR(buf.st_mode))
            if(strcmp(p->d_name,".") && strcmp(p->d_name, "..")) //이름이 "." ".."이 아니>면
                recursive_dir(p->d_name); // 그 (하위) 폴더에 대해 재귀함수 호출
        //if 부분 shortcut!! strcmp 는 같으면 0 반환
    }
    chdir(".."); // 하위 폴더내의 목록을 모두 출력한 후 상위로 디렉토리를 바꾼다
    closedir(dp); // recursive 호출한 곳으로 돌아간다
}
```

73. Linux 에서 fork()를 수행하면 Process 를 생성한다.

이때 부모 프로세스를 gdb 에서 디버깅하고자하면 어떤 명령어를 입력해야 하는가 ?

(gdb) set follow-fork-mode parent

74. C.O.W Architecture 에 대해 기술하시오.

시스템에서 fork 시에 모든 것을 생성하지 않고, 실제로 쓰기 동작이 일어날때 새로운 영역을 할당받는다.

fork 시에 부모가 갖고 있는 모든 것을 복제하기에는 시스템 자원의 소모가 크고 참조만 이루어지는 공간의 경우 복제할 필요가 없기 때문에, 최초 fork 시에 부모로부터 상속받은 data, stack, heap 의 주소를 포함한 자료만을 복사한다. 이후 실제 쓰기 동작이 일어날때 read-only 로 표시된 영역에 접근하기 때문에 mmu 에서 에러가 발생하고 그때 새로운 영역을 할당해주는 방식이다.

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

Blocking/NonBlocking 은 호출되는 함수가 바로 리턴하느냐 마느냐에 따라 구분된다.

호출된 함수가 바로 리턴해서 호출한 함수에게 제어권을 넘겨주고, 호출한 함수가 다른 일을 할 수 있는 기회를 줄 수 있으면 NonBlocking 이다. 예를 들어, 소켓이 blocking 일 경우 Server 가 Client 의 메시지 요청을 받기 위해 read 에서 기다리게 되는 경우를 들수 있다.(Client 가 write 하기 전에는 read 에서 빠져나오지 못한다)

그렇지 않고 호출된 함수가 자신의 작업을 모두 마칠 때까지 호출한 함수에게 제어권을 넘겨주지 않고 대기하게 만든다면 Blocking 이다.

76. 자식이 정상 종료되었는지 비정상 종료되었는지 파악하는 프로그램을 작성하시오.

부모 프로세스가 자식 프로세스의 종료 상태를 얻기 위해서는 wait() 함수를 사용한다.

이때 wait() 함수의 반환값과 status 값에 따라 자식이 정상 종료되었는지 비정상 종료되었는지가 구분된다.
WIFEXITED(status)가 1 이면 자식 프로세스 정상 종료

WIFSIGNALED(status)가 1 이면 자식 프로세스 비정상 종료

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();
    for(;;);
    return 0;
```

```
}
```

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

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

int main()
{
    signal(SIGINT, SIG_IGN); // ^c
    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.

setjmp/longjmp 는 프로그램의 흐름을 비정상적으로 제어하는 것으로 이른바 nonlocal goto 라고 불린다.
nonlocal 이라고 하는 것은 일반 goto 와 달리 함수 밖의 위치로 이동할 수 있기 때문이다.
하지만 아무런 제한없이 아무 함수로나 움직일 수 있는 것은 아니며
이동할 위치의 stack frame 이 보존된 상태여야 한다.

```
[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]

```
#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;
}
```

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

리눅스 시스템에서 모든 것은 파일이다. 일반적인 정규파일, 디렉토리, 소켓, 파이프, 블록 디바이스, 캐릭터 디바이스 등등 모든 객체들은 파일로써 관리된다. 프로세스가 이 파일들을 접근할 때에 파일 디스크립터(File Descriptor)라는 개념을 이용한다. 파일디스크립터는 단순한 숫자로 표현된다. 파일 디스크립터가 단순히 숫자인 이유는 프로세스가 유지하고 있는 FD 테이블의 인덱스이기 때문이다. FD 3 번이라는 의미는 FD 테이블의 3 번 항목이 가리키는 파일이라는 의미이다. FD 테이블의 각 항목은 FD 플래그와 파일 테이블로의 포인터를 가지고 있다. 이 포인터를 이용하여 FD 를 통해 시스템의 파일을 참조 할 수 있는 것이다.

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 필드에는 파일의 종류와 파일에 대한 접근권한 정보가 저장된다.

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

83. 디렉토리를 만드는 명령어는 mkdir 명령어다. man -s2 mkdir 을 활용하여 mkdir System Call 을 볼 수 있다.

이를 참고하여 디렉토리를 만드는 프로그램을 작성해보자!

84. 이번에는 랜덤한 이름(길이도 랜덤)을 가지도록 디렉토리를 3 개 만들어보자!

(너무 길면 힘드니까 적당한 크기로 잡도록함)

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)

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

```
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);
}

void kind_of_file(buf)
{
}

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

87. 클라우드 기술의 핵심인 OS 가상화 기술에 대한 질문이다.

OS 가상화에서 핵심에 해당하는 3 가지를 기술하시오.

CPU 가상화, I/O 가상화, 메모리 가상화

88. 반 인원이 모두 참여할 수 있는 채팅 프로그램을 구현하시오.

```
#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)
```

```
#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 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];
```

```
{
	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);
```

```
                        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");
```

```
                        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));
```

```
        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;
```

```
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(atoi(argv[2]));
        serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
        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;
}
```

```
}
```

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.

```
#ifndef __COMMON_H__
#define __COMMON_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

#endif
```

```
#include "common.h"
#include <signal.h>
#include <sys/wait.h>

typedef struct sockaddr_insi;
typedef struct sockaddr *  sp;

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;
```

```
#include "common.h"

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 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
```

```
	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;
}
```

```
*)&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;
}
```

92. 91 번을 응용하여 Queue 와 Network 프로그래밍을 연동하시오.

93. Critical Section 이 무엇인지 기술하시오.

배타적 접근(한 순간에 하나의 쓰레드만 접근)이 요구되는 공유 리소스(ie. 전역변수)에 접근하는 코드 블록이다.

여러 task 들이 동시에 접근하여 정보가 꼬일 수 있는 공간을 말한다.

스레드 하나면 lock 을 걸 필요가 없지만 스레드 여러개면 꼭 lock 을 걸어주어야 한다.

(전역변수이면 critical section 일까? 아니다. 여러 스레드에 의해서 공유될 때에만 임계영역이고

이때는 lock 을 걸어주어야 한다.)

fork 는 부모자식간에 독립적인 가상메모리를 갖는다.

종속적인 가상 메모리를 갖는 경우도 있다. pthread_create()로 스레드를 만들면 메모리(힙, 데이터,

텍스트)를 공유한다. 스택의 경우 지역변수를 사용하기 위한 독립적인 공간이므로 가상메모리를

공유하지 않는다.

이때 공유하는 공간에

lock 을 걸어서 A 가 사용중일때 B 가 이 영역을 침범하지 못하도록 해야한다.

94. 유저에서 fork() 를 수행할때 벌어지는 일들 전부를

실제 소스 코드 차원에서 해석하도록 하시오.

1) 사용자 수준 응용이 fork()시스템 콜 요청 - 모든 시스템 호출은 각각 고유한 번호를 가지고 있음

2) 표준 C 라이브러리 함수인 __fork() 호출

→ 사용자 대신 트랩 요청(eax 에 fork 함수 고유번호 저장, 트랩번호 0x80 으로 트랩요청)

3) 트랩이 걸리면 제어가 커널로 넘어간다.

→ 문맥저장, IDT 테이블에서 0x80 에 해당하는 함수(system call()) 호출

4) 이 함수는 eax 의 값을 인덱스로 ia32_sys_call_table 을 탐색하여 sys_fork()함수의 포인터를 얻어옴

5) 커널에서 구현된 sys_fork() 함수 호출

95. 리눅스 커널의 arch 디렉토리에 대해서 설명하시오.

arch : 하드웨어 종속적인 부분 구현, CPU 종류에 따라 하위 디렉토리로 구분 즉, 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 하위호환을 하지 않기 때문에 디렉토리가 많고 복잡하다

exynos: 삼성, 안드로이드 폰에 들어간다

omap: TI

zynq: 자일링스

s3c24xx , s3c64xx: 삼성

keystone: TI DSP , DSP 만으로 커널이 올라감(키스톤 2 는

레이더에 사용되며 아직 오픈소스는 아니다)

lpc: NXP, 차량용

bcm: 라즈베리파이

davinchi: TI→ 블랙박스, 스마트 tv, CCTV 등 비디오 시장

을 장악

stm32: 차량용 processor 로 coretex R 이 사용될 것, 그러면

서 커널에 등록됨

coretex R 전용 리눅스가 나올것

tegra: NVIDIA 에서 사용하는 리눅스 플랫폼

97. 리눅스 커널 arch 디렉토리에서 c6x 가 무엇인지 기술하시오.

C6x 는 TI 사의 DSP 이다

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 루틴으로 처리된다.

인터럽트 → 문맥 저장 → 인터럽트 핸들러 → 문맥 복원

(SAVE_ALL → do_IRQ() → ret_from_intr → RESTORE_ALL)

```
common_interrupt:
    ASM_CLAC
    addl    $-0x80, (%esp)              /* Adjust vector into the [-256, -1] range */
    SAVE_ALL
    TRACE_IRQS_OFF
    movl    %esp, %eax
    call    do_IRQ
    jmp     ret_from_intr
ENDPROC(common_interrupt)
```

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

ARM 에서는 시스템콜을 전달할 때 r7 레지스터를 사용한다.

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.

이번달에는 개인적인 일들로 수업에 잘 집중하지 못한 것 같다. 시스템 프로그래밍, 네트워크. 커널 수업을 쭉 거치면서 막히는 부분들이 있었는데 시험을 보니 빈 구멍들이 아주 잘 드러났다. 이번에 수~금의 자유시간동안 과제위주로 복습을 할 계획이다.

이번 시험을 준비하면서 복습 과정에서 많은 부족함을 느꼈다. 얕은 이해만 한 상태로 복습을 하지 않고 수업을 계속 들었기 때문에 시험날짜가 다가와서 몰아서 복습을 하려니 마음만 다급해지고 제대로 복습을 할 수 없었다. 당시에는 나름 열심히 했다고 생각했지만, 복습하면서 제출했던 과제와 공부했던 내용을 보니 이해 수준이 매우 미흡했다고 생각되었다. 그 당시에는 녹음파일을 여러번 들어도 어려웠지만, 다시 보니 쉽게 이해가 되는 부분도 있었고 당시에 '?'를 남겼던 부분이 미해결인채로 남은것도 많았다. 어떻게 복습할 것인가가 이번 시험을 통해서 꼭 해결해야할 부분이라고 생각한다.

얼마나 노력하였는가에 대하여 당당히 대답할 수 없는 시험결과를 예상하고 있어서, 나 자신에게도 수업해주신 선생님께도 죄송하다. 수업이 정말 재밌고 잘 하고 싶은데 따라가지 못해서 아쉽다. 앞으로 약 4 개월 뒤에 팀 프로젝트를 할때 주어진 역할을 잘 수행해내고 싶은 마음은 큰데 그러기 위해서는 노력을 많이 해야할 것 같다.