TI DSP, MCU 및
Xilinx Zynq FPGA
프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 : 황수정

sue100012@naver.com

2 개월 차 시험 (2018. 04. 17)

[임베디드 애플리케이션 구현 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 main (void)
{
  int fd, ret, fdin;
  char buf[1024];
  mkfifo("myfifo");
  fd = open("myfifo", O_RDWR);
 for(;;)
  {
    ret = read(0, buf, sizeof(buf));
    buf[ret-1] = 0;
    printf("Keyboard Input : [%s]₩n", buf);
    read(fd, buf, sizeof(buf));
    if(buf = c type.c)
    fdin = open("type.c", O_RDONLY);

    buf[ret-1] = 0;
    printf("pipe Input : [%s]₩n", buf);
   close(fidn);
  }
     return 0;
}
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
int main (void)
{
	int fd, ret;
	char buf[1024];
	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);
		}
		if ((ret = read(fd, buf, sizeof(buf))) >0)
		{
			buf[ret-1] =0;
			printf("Pipe Input :  [%s]₩n", buf);
		}
	}
	close(fd);
	return 0;
}
```

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

[임베디드 애플리케이션 구현 3] 리눅스 커널은 운영체제(OS)다. OS 가 관리해야 하는 제일 중요한 5 가지에 대해 기술하시오.

파일 시스템, 메모리, 네트워크, 디바이스, 프로세스(태스크)

+system call

[임베디드 애플리케이션 구현 4] Unix 계열의 모든 OS 는 모든 것을 무엇으로 관리하는가 ?

파일

‘모든 것은 파일이다.’ 가 핵심 철학이기 때문이다. 커널에서 관리한다.

[임베디드 애플리케이션 구현 5] 리눅스의 장점에 대한 각각의 상세한 기술
리눅스에는 여러 장점이 있다.(배점 0.2 점) 아래의 장점들 각각에 대해 기술하라.

* 사용자 임의대로 재구성이 가능하다.

> 커스텀 마이징이 가능하다. C 언어, 어셈블리어 공개 되어 있기 때문이다. 그리고 공개한다면 본인 마음대로 재구성할 수 있다.

* 열약한 환경에서도 HW 자원을 적절히 활용하여 동작한다.

>모놀리딕에 필요한 것(디바이스 드라이버)은 마이크로 커널로 하이브리드 방식이기 때문이다. 이는 최적화가 매우 잘 되어 있다는 뜻으로 컴퓨터 사양이 좋지 않아도 리눅스가 깔리는 이유이다.

* 커널의 크기가 작다.

> 윈도우보다 작다.

* 완벽한 멀티유저, 멀티태스킹 시스템

> 리눅스가 멀티태스킹으로 되어 있다. 하이브리드 방식이다.

* 뛰어난 안정성

> 네트워크과 기상기지국 등에서 사용하고 있다. 또한 많은 방산 업체 등에서 사용하고 있다는 것은 이미 이 안정성이 인정 되었다는 것이다.

* 빠른 업그레이드

> 원하는 사람은 누구나 그 소스르 ㄹ수정하여 성능을 향상시킬 수 있다는 장점 덕분에 많은 사람들에 의해 테스트 되고 개선, 개발 되었기 때문이다.

* 강력한 네트워크 지원

>TCP / IP 가 지원된다.

* 풍부한 소프트웨어

> 스톨만의 GNU 덕분이다.

[임베디드 애플리케이션 구현 6] 32 bit System 에서 User 와 Kernel 의 Space 구간을 적으시오.

0xffffffff ~ 0xc0000000 : kernel
0x00000000~ 0xbfffffff : user

32bit CPU 의 경우 운영체제는 각 프로세스에게 총 4GB 크기의 가상공간을 할당한다. 리눅스는 이중에서 0GB~3GB 의 공간을 사용자 공간으로 사용하고, 나머지 3GB~4GB 를 커널 공간으로 사용한다.

[임베디드 애플리케이션 구현 7] Page Fault 가 발생했을 때, 운영체제가 어떻게 동작하는지 기술하시오.

가상 메모리 시스템에 있는 프로그램 이터럽트의 일종으로 현재 주기억 장치에 존재하지 않기 때문에 디스크로부터 읽어 들여야 할 경우/ 현재 주기억 장치에 없는 페이지가 참조되었을 때 일어나는 인터럽트다.

[임베디드 애플리케이션 구현 8] 리눅스 실행 파일 포맷이 무엇인지 적으시오.

ELF

[임베디드 애플리케이션 구현 9] 프로세스와 스레드를 구별하는 방법에 대해 기술하시오.

pid 와 tgid 를 비교한다.
pid = tgid 면 프로세스 (즉, 쓰레드 리더가 프로세스이다.)
pid 와 tgid 가 같지 않으면 해당 쓰레드이다.
tgid 는 thread Group ID 라는 개념으로 한 프레스 내의 쓰레드는 동일한 pid 를 공유해야 한다는 POSIX 표준에 의하여 도입된 것이다.

[임베디드 애플리케이션 구현 10] Kernel 입장에서 Process 혹은 Thread 를 만들면 무엇을 생성하는가 ?

태스크 (task_struct)

[임베디드 애플리케이션 구현 11] 리눅스 커널 소스에 보면 current 라는 것이 보인다. 이것이 무엇을 의미하는 것인지 적으시오. 커널 소스 코드와 함께 기술하시오.

current 라는 매크로는 커널 내부에 정의되어 있는 매크로로써 현재 태스크의 task_struct 구조체를 가리킬 수 있게 해주며(현재 구동중인 task 의 task_struct 의 포인터), task_tgid_vnr()은 해당 task_struct 구조체의 tgid 필드를 리턴한다. 따라서 이 함수는 task_struct 구조체의 tgid 필드 값을 리턴하는 단순한 함수라고 볼 수 있다.

[임베디드 애플리케이션 구현 12] Memory Management 입장에서 Process 와 Thread 의 핵심적인 차이점은 무엇인가 ?

프로세스는 메모리 위에서 완전히 독립된 공간이고 쓰레드는 스택 영역을 제외한 나머지 메모리 영역을 공유한다.

[임베디드 애플리케이션 구현 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 ?

하드웨어 문맥(Hardware Context)로 context switch(문맥 교환)할 대, 태스크의 현재 실행 위치에 대한 정보를 유지하며, 쓰레드 구조 또는 하드웨어 레지스터 문맥이라고 불린다. 이 부분은 실행 중이던 태스크가 대기 상태나 준비 상태로 전이할 때 이 태스크가 어디까지 실행했는지 기억해 두는 공간으로, 이후 이 태스크가 다시 실행될 때 기억해 두었던 곳부터 다시 시작하게 된다.

[임베디드 애플리케이션 구현 14] 리눅스 커널의 스케쥴링 정책중 Deadline 방식에 대해 기술하시오.

EDF(Earliest Deadline First) 알고리즘을 구현한 것이다. 이는 deadline 이 가장 가까운(즉, 가장 급한) 태스크를 스케줄링 대상으로 선정하다. 정책을 사용하는 태스크들은 deadline 을 이용하여 RBTree 에 정렬되어 있다. DEADLINE 을 사용하는 태스크의 경우 우선순위는 의미가 없고 주기성을 가지는 프로그램(영상,음성,스트리밍)과 제약시간을 가지는 응용들에 효과적으로 적용 가능하다.
 예를 들어 동영상을 재생하는 태스크가 수행중이라고 가정해보자. 끊김 없이 화면을 재생하기 위해 초당 30 프레임을 디코딩하여 화면에 출력해야 한다면 이 태스크는 1 초당 30 번씩 '해야하는 일'을 가지고 있다고 볼 수 있다.
이는 시간 내에 처리해야 하는 deadline 이 있다는 것이다. 모니터 그리기 1/30 초로 약 0.033 초 마다 deadline 이 와서 모니터 그리기를 할 때 실제 걸리는 시간이 0.01 초라 가정했을 때 0.23 초가 남는다. 이 시간 동안 다른 작업을 하여 0.02 초 동안 context switching 이 가능하다는 것이다. 그러다가 deadline 이 오면 화면에 그리는 작업을 해주면 되는 것이다. 즉, deadline 이 올 때, 그 작업을 우선적으로 수행하여 context switching 을 하면 되는 것이다.

[임베디드 애플리케이션 구현 15] TASK_INTURRUPTIBLE 과 TASK_UNINTERRUPTIBLE 은 왜 필요한지 기술하시오.

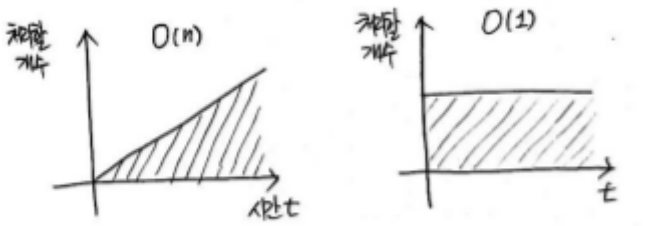
TASK_INTERRUPTIBLE: 프로세스는 잠든 상태이며, 사건이 일어나기를 기다리고 있다. 프로세스는 시그널이 인터럽트하도록 열려 있다. 일단 시그널을 받거나 명시적인 깨어나기 호출로 깨어나면, 프로세스 상태가 TASK_RUNNING 으로 전이한다.

TASK_UNINTERRUPTIBLE: 프로세스 상태는 TASK_INTERRUPTIBLE 과 비슷하다. 하지만 이 상태에서 프로세스는 시그널을 처리하지 않는다. 몇몇 중요한 작업을 완료하는 중간에 있기에 이 상태에 있을 경우 프로세스를 인터럽트하는 상황이 그리 바람직하지 않을 가능성도 있다. 기다리고 있던 사건이 발생할 때, 프로세스는 명시적인 깨어나기 호출로 깨어난다.

- 태스크가 특정한 사건을 기다려야 할 필요가 있을 때 필요하다. 태스크가 이미 lock 이 걸려있는 자원에 대해 lock 을 획득하려고 시도한다. 디스크같은 주변 장치에 요청을 보내고 그 요청이 완료되기까지 기다리는 것이다. 대기하는 동안 기다리는 그 사전 외에는 일체 방해받아선 안되는 경우가 TASK_UNINTERRUPTIBLE, 그렇지 않으면 TASK_INTERRUPTIBLE 이다. 대기 상태로 전이한 태스크는 기다리는 사건(event)에 따라 특정 큐(queue)에서 대기한다. 태스크가 기다리고 있던 사건(event)이 발생하면 대기 상태 태스크는 TASK_RUNNING(ready)상태로 전이한다.

[임베디드 애플리케이션 구현 16] O(N)과 O(1) Algorithm 에 대해 기술하시오. 그리고 무엇이 어떤 경우에 더 좋은지 기술하시오.

이는 SCHED_FIFO, SCHED_RR 사용할 때의 문제점으로 태스크의 개수가 늘어나면 그만큼 스케줄링에 걸리는 시간도 선형적으로 증가하게 되며 (O(n)의 시간복잡도), 따라서 스케줄링에 소모되는 시간을 예측할 수 없다.



옆의 그림에서 색칠한 면적은 총 처리해야 할 CPU 명령어 개수이다.
처리해야 할 task 가 적으면 O(n)방식이 좋고 처리해야 할 task 가 많으면 O(1)방식이 좋다.
그 이유는 O(n)방식은 데이터가 많을 수록 걸리는 시간이 늘어나고 O(1)은 데이터가 얼마던 걸리는 시간이 일정하기 때문이다.

O(1)로 바꾸는 방법은 Hash, Map 이라는 자료구조를 만들면 O(1)으로 만들 수 있다. 태스크들이 가질 수 있는 모든 우선순위 레벨(0~99)을 표현할 수 있는 비트맵을 준비한다. 태스크가 생성되면 그 태스크의 우선순위에 해당하는 비트를 1 로 set 한 뒤, 태스크의 우선순위에 해당되는 큐에 삽입된다. 태스크를 생성하고 1 로 세팅하는 이유는 &연산하면 1 이 있는 것만 확인이 되고 없으면 0 이 나오게 된다. queue 를 사용하는 이유는 같은 우선순위를 가진 task 들이 존재할 수 있기 때문에 우선순위에 따른 task 들을 queue 로 넣어 리스트로 관리한다. 이렇게 해서 bitmap 에 가장 처음으로 set 되어있는 비트가 가장 우선순위가 높은 것 이므로 그 우선순위 큐에 매달려 있는 태스크를 선택하여 for 문이나 while 문을 사용하지 않고도 스케줄링 작업이 고정시간 내에 완료하여 O(1)방식으로 처리할 수 있다.

[임베디드 애플리케이션 구현 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.

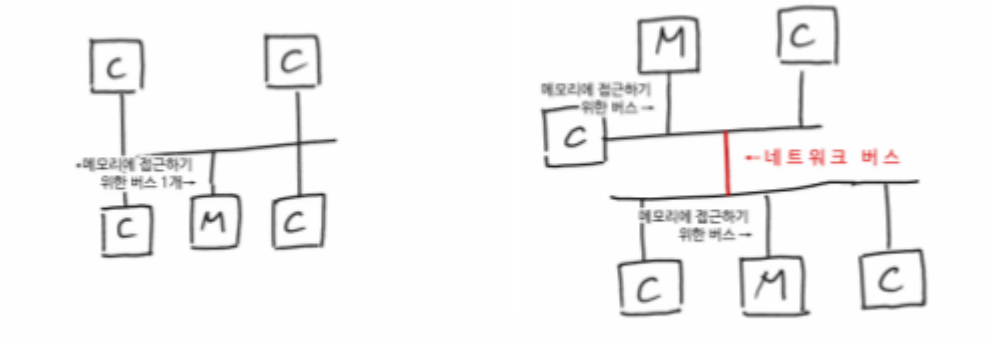
다음 사항을 고려하여 삽입하면 된다.

1) 새로 생성된 태스크는 부모 태스크가 존재하던 런 큐로 삽입한다. 이는 더 높은 캐시 친화력 (cache affinity)를 얻을 수 있기 때문이다.
2) 런 큐간의 부하가 균등하지 않은 경우 → 부하 균등 (load balancing) : 특정 CPU 가 많은 작업을 수행하고 있느라 매우 바빠 한가한 다른 CPU 에 태스크를 '이주'시켜서 성능 향상시키는 것을 말한다.
*. 하이퍼 스레딩
- fork()가 회로로 구현되어 있다 , 실제 CPU 는 4 개인데 8 개처럼 동작하게 만든것이다.
- 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.

[임베디드 애플리케이션 구현 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.

런 큐간의 부하가 균등하지 않은 경우 → 부하 균등 (load balancing) : 특정 CPU 가 많은 작업을 수행하고 있느라 매우 바빠 한가한 다른 CPU 에 태스크를 '이주'시켜서 성능 향상시키는 것을 말한다.
*. 하이퍼 스레딩
- fork()가 회로로 구현되어 있다 , 실제 CPU 는 4 개인데 8 개처럼 동작하게 만든것이다.
- 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.

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



- UMA - NUMA

UMA(Uniform Memory Access) = SMP(Symmetric Multi Processor)
: 메모리 접근하는 속도가 모두 같다. 모든 CPU 들이 메모리를 공유한다.
NUMA(Non-Uniform Memory Access)
: 메모리 접근 속도가 다르다. NUMA 에서 메모리 접근 타이밍이 다르다. 필요한 메모리가 반대쪽 메모리에 존재할 때 네트워크 버스를 타고 접근한다. 이기종 아키텍처에 사용되는 방식이다.

[임베디드 애플리케이션 구현 20] Kernel 의 Scheduling Mechanism 에서 Static Priority 와 Dynamic Priority 번호가 어떻게 되는지 적으시오.

task_struct 에서는 세개의 priority 가 관리된다.
dynamic priority (task_struct->prio)
static priority(task_struct->static_prio)
normal priority(task_struct->normal_prio)

#define NICE_TO_PRIO(nice) (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio) ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)PRIO_TO_NICE((p)->static_prio)

nice 는 유저레벨에 노출되는 값이며, priority 는 nice 와 연관된 커널 내부의 우선순위이다.
그 중 priority 100 ~ 139 까지가 CFS 의 관리 대상이 되는 태스크이며 이 값이 낮을 수록 높은 priority 를 의미한다.

0~99 까지의 rt priority 역시 값이 낮을 수록 높은 priority 를 의미하는데 여기서 헷갈릴 수 있는 부분이 task_struct->prio, static_prio, normal_prio 모두 저장된 값이 낮을수록 높은 prioroity 를 의미하지만 아래와 같이 setscheduler 류의 함수로 priority 를 변경하려고 할 때는 그 의미가 좀 달라진다.

즉, setscheduler 에 전달되는 prio 값과 이 값이 그대로 저장되는 rt_priority 의 값은 높을수록 최종적으로는 낮은 값으로 변환되어 높은 priority 의미하게 되는 것이다.

[임베디드 애플리케이션 구현 21] ZONE_HIGHMEM 에 대해 아는대로 기술하시오.

이 영역은 "상위 메모리"를 포함한다. 상위 메모리는 커널 주소 공간으로 영구적으로 매핑되지 않는다.
리눅스의 가상 주소공간과 물리메모리 공간을 1:1 로 연결한다면 아무리 메모리를 사용하려 해봐야 1GB 이상은 접근할 수가 없을 것이다. 가상 메모리는 4GB 인데 물리는 64GB 이면 현 시스템에선 약점이 되기

때문이다. 그래서 물리 메모리가 1GB 이상이라면 896MB 까지 커널의 가상주소공간과 1:1 로 연결해주고 나머지 부분은 필요할 때 동적으로 연결하여 사용하는 구조를 채택하였다. 여기서 896MB 이상의 메모리 영역을 ZONE_HIGHMEM 이라 부른다.
즉, 896MB 이상의 물리 메모리로 동적으로 매핑되는 페이지를 말하는 것이다. 지속적으로 커널 주소영역에 매핑되지 않는 페이지 프레임을 포함한다. 896MB 위쪽의 물리 메모리는 커널이 해당 메모리에 접근할 필요가 있을때만 일시적으로 커널 가상 메모리로 매핑된다.

high memory 가 처음 할당되면 그것은 직접 어드레싱 불가능하다. kmap(), kunmap(), kmap_atomic(), kunmap_atomic()을 이용해서 highmem 을 커널 가상 메모리로 일시적으로 매핑할 수 있다. 해당 메모리에 어드레싱 하려면

1. kmap()를 호출해서 메모리 페이지를 커널 페이지 테이블에 넣는다
2. kunmap()를 호출하기 전까지 주소는 유요하다
3. 이 페이지에 접근하기전에 kmap()~kunmap()를 호출하는 과정이 필요하다.

[임베디드 애플리케이션 구현 22] 물리 메모리의 최소 단위를 무엇이라고 하며 크기가 얼마인가 ?
그리고 이러한 개념을 SW 적으로 구현한 구조체의 이름은 무엇인가 ?

Page frame 이라고 하며 4KB 로 지정 되어 있다. 리눅스는 시세템 내의 모든 물리 메모리에 접근 가능해야 한다. 이를 위해 모든 페이지 프레임 당 하나씩 page 구조체가 존대한다. 이는 시스템이 부팅되는 순간에 구축되어 역시 물리 메모리 특정 위치에 저장된다. 이 위치는 node_mem_map 이라는 전역 배열을 통해 접근 할 수 있다.

[임베디드 애플리케이션 구현 23] Linux Kernel 은 외부 단편화와 메모리 부하를 최소화하기 위해 Buddy 할당자를 사용한다. Buddy 할당자의 Algorithm 을 자세히 기술하시오.

 버디 할당자는 zone 구조체에 존재하는 free_area 배열을 통해 구축된다. zone 당 하나씩 버디가 존재한다. free_area 는 10 개의 엔트리를 가진다. 0~9 까지의 숫자는 free_area 가 관리하는 할당의 크기를 나타낸다. 4MB(2 0~9 * 4KB 의 크기를 가진다. nr_free 라는 free 한 페이지 개수를 나타내는 변수와 list_head 라는 연결리스트가 존재한다. map 이라는 변수의 비트맵에 페이지 프레임 단위로 봤을 때의 상태를 저장한다.

만일 연속적인 페이지 4 개가 필요하다면, 4 개인 리스트를 살펴본다. 없을 경우 8 개로 올라가서 하나를 할당 받고, 나머지 4 개는(있다면…) 현재 사용할 수 있는 4 개 프레임 리스트 뒤로 이어지게 한다. 만약 2 개를 할당해달라고 요청했는데 2 개 리스트에도, 4 개 리스트에도 없다면, 8 개 리스트를 참조하여 존재할 시에 일단 8 개짜리에서 2 개를 할당하고, 그 나머지 6 개 프레임을 4 개, 2 개로 쪼개서 각각의 페이지 프레임을 기존의 리스트에 이어 준다.

[임베디드 애플리케이션 구현 24] 앞선 문제에 이어 내부 단편화를 최소화 하기 위해 Buddy 에서 Page 를 받아 Slab 할당자를 사용한다. Slab 할당자는 어떤식으로 관리되는지 기술하시오.

 페이지 프레임 크기가 클수록 내부 단편화로 인한 낭비되는 공간이 큰데, 버디를 이용하여 공간을 쪼개게 되면 공간을 쪼개는 작업이 속도를 느리게 하는 문제점이 있다. 이 문제를 해결하기 위한 것이 슬랩할당자이다. 페이지 프레임의 크기가 4KB 일 때, 64byte 크기의 메모리를 공간을 요청하면 미리 4KB 페이지 프레임을 할당 받은 뒤 이 공간을 64byte 분할해두어 이 공간을 떼어주도록 한다. 일종의 캐시로 사용하는 것인데 이런 메모리 관리하는 정책을 슬랩 할당자라 부른다.
 자주 할당하고 해제되는 크기의 cache 를 가지고 있어야 내부 단편화를 최소화 시킬 수 있다. 2 의 승수 크기의 cache 를 128KB 유지한다. 다양한 크기의 캐시를 관리하는 kmem_cache 라는 자료구조가 정의되어 있다. 슬랩 할당자는 위의 함수 외에도 외부 인터페이스 함수로 kmalloc() / kfree()를 제공한다. kmalloc()함수를 이용ㅇ해 한번에 할당 받을 수 있는 최대 크기는 128K 혹은 4MB 이며 할당된 공간은 물리적으로 연속이라는 특징을 가진다.

[임베디드 애플리케이션 구현 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)

task_struct 에서 관리한다. task_struct > mm_struct 구조체에 저장된다.

[임베디드 애플리케이션 구현 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 >밑에 unsigned long 으로 7 개 변수가 선언되어 있다.

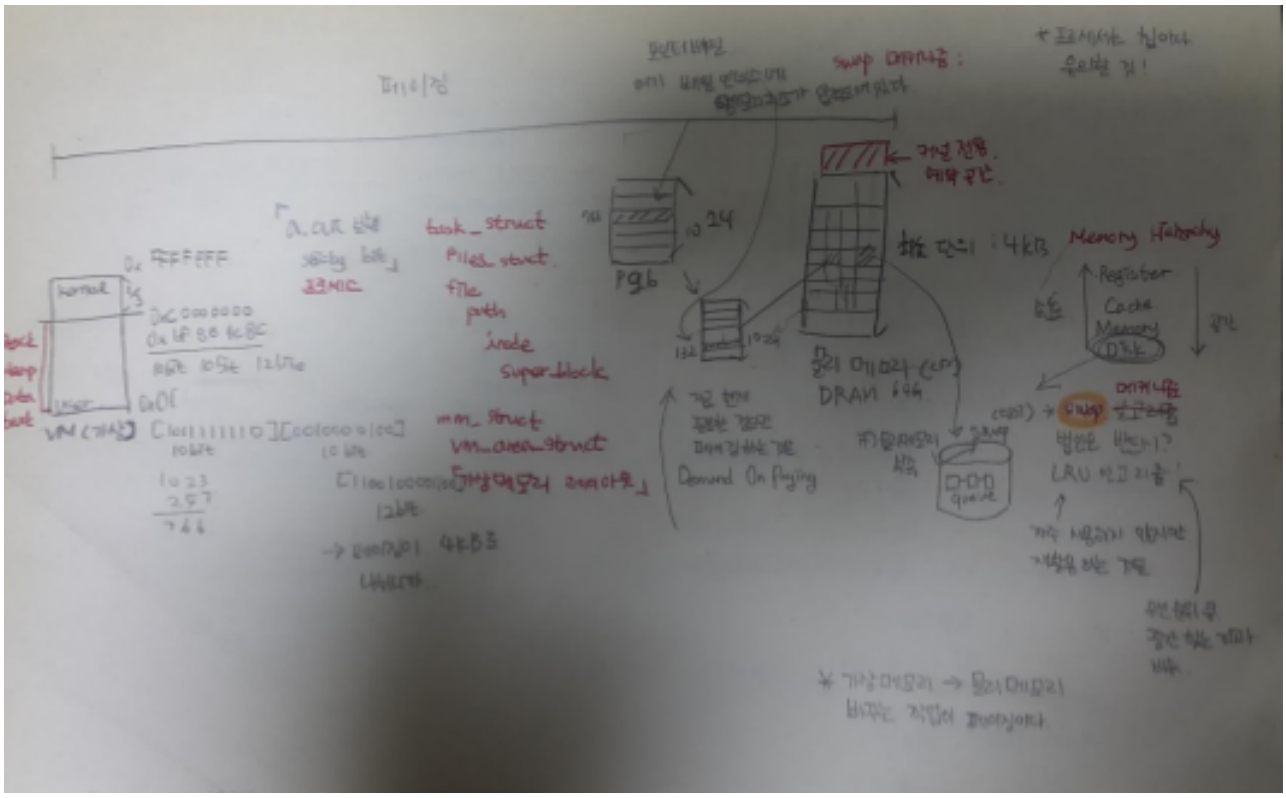
```
unsigned long def_flags;
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk, start_stack;
unsigned long arg_start, arg_end, env_start, env_end;

unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */
```

[임베디드 애플리케이션 구현 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 ?

[임베디드 애플리케이션 구현 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.

리눅스 자체도 파일이기 때문이다. 메모리에 올라가야 프로세스가 되고 작업이 진행되기 때문이다.

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



[임베디드 애플리케이션 구현 30] MMU(Memory Management Unit)의 주요 핵심 기능을 모두 적고 간략히 설명하시오.

가상주소를 사용하는 운영체제가 원활히 수행될 수 잇도록 하기 위해 가상 주소로부터 물리주소로의 변환을 담당하는 하드웨어이다.

[임베디드 애플리케이션 구현 31] 하드디스크의 최소 단위를 무엇이라 부르고 그 크기는 얼마인가 ?

Page frame 이라고 하며 4KB 로 지정 되어 있다.

[임베디드 애플리케이션 구현 32] Character 디바이스 드라이버를 작성할 때 반드시 Wrapping 해야 하는 부분이 어디인가 ? (Task 구조체에서 부터 연결된 부분까지를 쭉 이어서 작성하라)

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

[임베디드 애플리케이션 구현 34] task_struct 에서 super_block 이 하는 역할은 무엇인가 ?

파일시스템의 전체적인 정보를 가지고 있다.
루트 시스템 정보를 가지고 있어 기준을 알려준다.

[임베디드 애플리케이션 구현 35] VFS(Virtual File System)이 동작하는 Mechanism 에 대해 서술하시오.

[임베디드 애플리케이션 구현 36] Linux Kernel 에서 Interrupt 를 크게 2 가지로 분류한다. 그 2 가지에 대해 각각 기술하고 간략히 설명하시오.

- 외부 인터럽트 : 현재 수행중인 태스크와 관련 없는 주변장치에서 발생된 비동기적인 하드웨어적인 사건
- 트랩 : 내부 인터럽트. 소프트웨어적인 사건. 예외처리(exception handling)라고도 한다. 0 으로 나누는 연산(devide_by_zero), 세그멘테이션 결함, 페이지 결함, 보호 결함, 시스템 호출 등이 있다.

즉, 인터럽트의 발생 기준은 CPU 내부에서 감지하는지, 외부에서 감지하는지에 따라 분류한다.

[임베디드 애플리케이션 구현 37] 내부 인터럽트는 다시 크게 3 분류로 나눌 수 있다. 3 가지를 분류하시오.
> 내부 인터럽트는 fault, trap, abort 로 나뉘게 된다.

[임베디드 애플리케이션 구현 38] 앞선 문제에서 분류한 녀석들의 특징에 대해 기술하시오.

fault - ex)page, fault
fault 를 일으킨 명령어 주소를 eip 에 넣어 두었다가 해당 핸들러가 종료되고 나면 eip 에 저장되어 있는 주소부터 다시 수행을 시작한다.
trap - ex)int, system call
trap 을 일으킨 명령어의 다음 주소를 eip 에 넣어두었다가 그 다음부터 다시 수행한다.
abort - ex)devide by zero
이는 심각한 에러인 경우이므로 eip 값을 저장해야할 필요가 없으며 현재 태스크를 강제종료시키면 된다.

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

[임베디드 애플리케이션 구현 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.

IDT 테이블에서 0x80 에 해당하는 함수(system call()) 호출한다. 이 함수는 eax 의 값을 인덱스로 ia32_sys_call_table 을 탐색하여 함수의 포인터를 얻어온다.

[임베디드 애플리케이션 구현 41] 38 에서 User Space 에서 System Call 번호를 전달한다. Intel Machine 에서는 이를 어디에 저장하는가 ? 또한 ARM Machine 에서는 이를 어디에 저장하는가 ?

[임베디드 애플리케이션 구현 42] Paging Mechanism 에서 핵심이 되는 Page Directory 는 mm_struct 의 어떤 변수가 가지고 있는가 ?

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

[임베디드 애플리케이션 구현 44] 커널 내부에서 메모리 할당이 필요한 이유는 무엇인가 ?

모든 컴퓨터는 기계어 기반으로 돌아간다. 따라서 특정 작업이 수행되려면 디스크도 메모리에 올라가야 하기 때문이다. 파일시스템은 디스크를 물리적인 구조로 보지 않는다. 논리적인 디스크 블록들의 집합으로 본다.

[임베디드 애플리케이션 구현 45] 메모리를 불연속적으로 할당하는 기법은 무엇인가 ?

불연속 할당의 경우 파일에 속한 디스크 블록들이 어디에 위치하고 있는지에 대한 정보를 기록해 두어야 하는데, 그 방법으로 블록체인 기법, 인덱스 블록 기법, FAT(File Allocation Table) 등이 있다.

블록체인 기법
같은 파일에 속한 디스크 블록들을 체인으로 연결해 놓는 방법으로 각 블록에 포인터를 두어 다음 블록의 위치를 기록한다. Linked list 와 유사하며 포인터를 저장해야 하므로 실질적으로 4KB 를 다 사용하지 못한다. file name, start, size 를 저장한다. 첫 번째 디스크 블록에 가면 포인터를 이용해 다음 블록의 위치를 찾아 갈 수 있다. lseek()같은 시스템 콜을 사용할 때 파일의 끝 부분을 읽으려는 경우에 앞 부분의 데이터 블록을 읽어야 하고 중간 블록이 유실되면 나머지 데이터까지 모두 잃게 되는 단점이 있다.

인덱스 블록 기법
블록들에 대한 위치 정보들을 기록한 인덱스 블록을 따로 사용하는 방법이다. file name, index, size 를 index block 에 저장하여 각 인덱스들은 디스크 블록을 가리키고 있다. lseek()를 사용하여 파일의 끝 부분을 접근할 때 데이터 블록을 일일이 읽을 필요가 없지만 인덱스 블록이 유실되면 데이터 전체가 소실되는 단점이 있다.

FAT 기법
같은 파일에 속해 있는 블록들의 위치를 FAT 라는 자료구조에 기록해 놓는 방법이다. 블록체인 기법과 인덱스 블록 기법의 개념을 합친 기법으로, 파일시스템 전체적으로 하나의 FAT 이 존재하는 것이다. 이 구조에서 FF 는 파일의 끝을 의미하며 0 은 free 상태를 나타낸다. start 인덱스가 있어 데이터가 중간에 유실되어도 복원이 가능한 장점이 있지만, FAT 구조의 유실은 파일시스템 내의 모든 파일이 소실되는 단점이 있다.

[임베디드 애플리케이션 구현 46] 메모리를 연속적으로 할당하는 기법은 무엇인가 ?

sequential 로 파일에게 연속된 디스크 블록을 할당하는 방법이다, 간혹 흩어져 있는 디스크들을 디스크 조각 모음으로 연속 할당으로 다시 배치해주기도 한다.

[임베디드 애플리케이션 구현 47] Mutex 와 Semaphore 의 차이점을 기술하시오.

프로세스가 실행될 때 다른 프로세스가 동작하는 프로세스에 접근하게 된다면 실행중인 프로세스의 데이터는 꼬이게 될 것이다. 데이터가 꼬일 수 있는 지점 critical section 에 lock 을 걸게 되면 데이터가 꼬이는 것을 방지할 수 있는데 lock 을 거는 방식이 2 개가 있는데 semaphore 과 spin lock 이 있다. semaphore 는 lock 이 걸리면 lock 이 풀릴 때 까지 접근하지 못해 lock 이 없는 곳을 찾아다니며, 남은 메모리가 없을 때 대기열이 존재한다. 프로세스 대기열이 있다는 것은 프로세스가 wait queue 로 빠진다는 것이고 context switching 을 해야한다는 것이니 semaphore 는 복잡하고 연산이 오래 걸리는 프로세스들을 처리할 때 걸게 된다.

[임베디드 애플리케이션 구현 48] module_init() 함수 호출은 언제 이루어지는가 ?

모듈이 시작될 때. 컴파일이 완료 된 후에 insmod 라는 명령어를 통해 이뤄진다. 이 명령은 인자로 주어진 파일을 커널에 넣기 위한 공간을 할당받고 모듈을 삽입한다. 또한 이때 모듈에서 사용하는 함수나 extern 한 변수 등을 실제 커널 정보를 참조하여 연결시켜 준다. 그런 뒤 모듈에 구현된 module_init() 매크로에 정의된 함수를 호출해준다.

[임베디드 애플리케이션 구현 49] module_exit() 함수 호출은 언제 이루어지는가 ?

모듈이 종료될 때. 적재된 모듈을 해제하는 병령은 rmmod 로 해당 모듈에 module_exit()매크로를 통해 정의되어 있는 함수를 호출하고 모듈이 차지하고 있던 공간을 회수한다.

[임베디드 애플리케이션 구현 50] thread_union 에 대해 기술하시오.
태스크가 생성되면 리눅스는 task_struct 구조체와 커널 스택을 할당하게 된다. 태스크 당 할당되는 커널 스택은 thread_union 이라 불리며, thread_info 구조체를 포함하고 있다. 이 구조체 안에는 해당 태스크의 task_struct 를 가르키는 포인터와 스케줄링의 필요성 여부를 나타내는 플래그, 태스크의 포맷을 나타내는 exec_domain 등의 필드가 존재한다.
커널 스택 안에 현재 레지스터의 값들을 구조체를 이용하여 일목요연하게 저장한다.

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

[임베디드 애플리케이션 구현 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)

아이노드 객체에는 i_rdev 필드가 존재하는데 이 i_rdev 필드에 주 번호(장치 종류)와 부 번호(장치 개수)를 저장한다.

[임베디드 애플리케이션 구현 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.

[임베디드 애플리케이션 구현 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.

[임베디드 애플리케이션 구현 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!

[임베디드 애플리케이션 구현 55] OoO(Out-of-Order)인 비순차 실행에 대해 기술하라.

[임베디드 애플리케이션 구현 56] Compiler 의 Instruction Scheduling 에 대해 기술하라.

[임베디드 애플리케이션 구현 57] CISC Architecture 와 RISC Architecture 에 대한 차이점을 기술하라.

[임베디드 애플리케이션 구현 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 어느정도의 Instruction Scheduling 을 HW 의 힘만으로 수행할 수 있다. 이러한 것을 무엇이라 부르는가 ?

[임베디드 애플리케이션 구현 59] Pipeline 이 깨지는 경우에 대해 자세히 기술하시오.

call 이나 jmp 를 CPU Instruction(명령어) 레벨에서 분기 명령어라고 하고 이들은 CPU 파이프라인에 매우 치명적인 손실을 가져다 준다. 즉, 파이프라인을 깨지게 한다.
기본적으로 아주 단순한 CPU 의 파이프라인을 설명하자면 아래와 같은 3 단계로 구성된다.
1. Fetch - 실행해야할 명령어를 물어옴
2. Decode - 어떤 명령어인지 해석함
3. Execute - 실제 명령어를 실행시킴
파이프라인이 짧은 것부터 긴 것이 5 단계 ~ 수십 단계로 구성된다. ARM, Intel 등등 다양한 프로세서들 모두 마찬가지이다. 그런데 왜 jmp 나 call 등의 분기 명령어가 문제가 될까?
기본적으로 분기 명령어는 파이프라인을 때려부수기 때문이다. 이 뜻은 위의 가장 단순한 CPU 가 실행까지 3 clock 을 소요하는데 파이프라인이 깨지니, 쓸데없이 또 다시 3 clock 을 버려야 하는 것을 의미한다. 만약 파이프라인의 단계가 수십 단계라면, 분기가 여러 번 발생하면 파이파이프라인 단계 x 분기 횟수만큼 CPU clock 을 낭비하게 된다. 즉 성능면에서 안 좋아지게 된다.

[임베디드 애플리케이션 구현 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 ?

[임베디드 애플리케이션 구현 61] Intel 의 Hyper Threading 기술에 대해 상세히 기술하시오.

[임베디드 애플리케이션 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.

[임베디드 애플이케이션 구현 63] 파일의 종류를 확인하는 프로그램을 작성하시도록 하시오.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>

int main(int argc, char **argv)
{
    struct stat buf;
    struct passwd *pw;
    struct group *gr;
    char ch;
    char perm[11] = "----------";
    char rwx[4] = "rwx";
    int i;
    stat(argv[1], &buf);
    if(S_ISDIR(buf.st_mode))
        ch = 'd';
    if(S_ISREG(buf.st_mode))
        perm[0] = '-';
    if(S_ISFIFO(buf.st_mode))
        perm[0] = 'p';
    if(S_ISLNK(buf.st_mode))
        perm[0] = 'l';
    if(S_ISSOCK(buf.st_mode))
        perm[0] = 's';
    if(S_ISCHR(buf.st_mode))
```

```
        perm[0] = 'C';
    if(S_ISBLK(buf.st_mode))
        perm[0] = 'b';
     for(i=0; i<9; i++)
       if((buf.st_mode >>(8-i)) &1)
        perm[i+1] = rwx[i%3];
   printf("%s", perm);
   printf("%lu", buf.st_nlink);
   pw = getpwuid(buf.st_uid);
   printf("%s", pw->pw_name);
   gr = getgrgid(buf.st_gid);
   printf("%s", gr->gr_name);

   return 0;
}
```

[임베디드 애플리케이션 구현 64] 서버와 클라이언트가 1 초 마다 Hi, Hello 를 주고 받게 만드시오.
[서버 소스]

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

typedef struct sockaddr_in si;
typedef struct sockaddr * sap;

#define BUF_SIZE      1024

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

int main (int argc, char **argv)
{
  int serv_sock, clnt_sock;
  int read_cnt, fd;
  si serv_addr, clnt_addr;
  socklen_t clnt_addr_size;
  char msg_1[BUF_SIZE] = {0};
  char msg[] = "Hello";

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

   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, (sap)&serv_addr, sizeof(serv_addr)) == -1)
    err_handler("bind() error");

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

  clnt_addr_size = sizeof(clnt_addr);
  clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);

  if(clnt_sock == -1)
    err_handler("accept() error");

  write(clnt_sock, msg, sizeof(msg));
  //clnt_sock = 0;

  for(;;) // 현재 보내기만 할 것이므로 여러 명을 받을 필요 없어서 for 대신 while 로 하는 것이 더..
  {
    read_cnt = read(clnt_sock, msg_1, BUF_SIZE);
    if(read_cnt < BUF_SIZE)
    {
      write(clnt_sock, msg_1, read_cnt);
       break;
    }
      write(clnt_sock, msg_1, BUF_SIZE);
  }

  shutdown(clnt_sock, SHUT_WR);
  read(clnt_sock, msg_1, BUF_SIZE);
  printf("msg form client : %s ₩n", msg_1);

  close(fd);
  close(clnt_sock);
  close(serv_sock);

  return 0;
}

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

typedef struct sockaddr_in   si;
typedef struct sockaddr *  sap;

#define BUF_SIZE      32

void err_handler(char *msg)
```

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

int main(int argc, char **argv)
{
  int sock, fd;
  int str_len;
  si serv_addr;
  char msg[BUF_SIZE] = {0};
  char msg_1[] = "Hi";

  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, (sap)&serv_addr, sizeof(serv_addr)) == -1)
      err_handler("connect() error");

   write(sock, "Hi", 3);
   //sock = 0;

   str_len = read(sock, msg, sizeof(msg)-1);

   if(str_len == -1)
     err_handler("read() error!");

   printf("msg form serv :%s ₩n", msg);
   close(sock);

  return 0;
}
```

[임베디드 애플리케이션 구현 65] Shared Memory 를 통해 임의의 파일을 읽고 그 내용을 공유하도록 프로그래밍하시오.

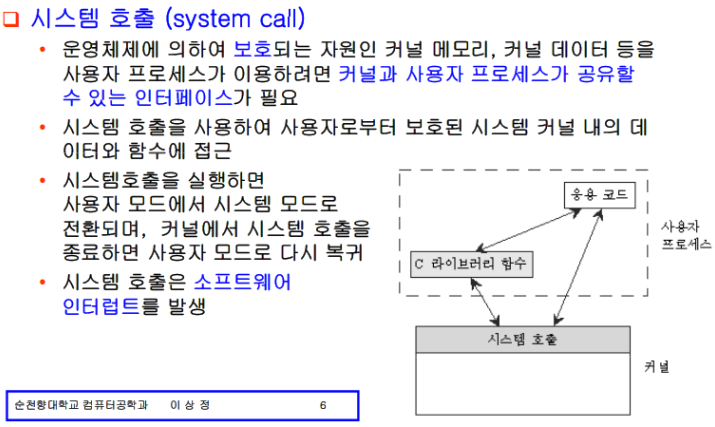
[임베디드 애플이케이션 구현 66] 자신이 사용하는 리눅스 커널의 버전을 확인해보고 https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/$(자신의 버전).tar.gz 를 다운받아보고 이 압축된 파일을 압축 해제하고 task_struct 를 찾아보도록 한다. 일련의 과정을 기술하면 된다.

[임베디드 애플이케이션 구현 67] Multi-Tasking 의 원리에 대해 서술하시오. (Run Queue, Wait Queue, CPU 에 기초하여 서술하시오)
Context switching 을 기반으로 multi-tasking 이 일어난다. 프로세스 A 와 B 가있다고 가정해보자. 이

때, 프로세스는 A 와 B 를 왔다갔다 하면서 작업이 이루어진다. 이는 CPU 를 얻기 위해서 경쟁하는 것이다. CPU 의 프로세스를 얻어야 실행될 수 있기 때문이다. CPU 는 오로지 한 순간에 한 가지 연산만 수행하기 때문에 진행되지 않는 프로세스는 wait queue 에 빠지고 진행 중인 프로세스는 run queue 에 있게 된다. 이 때, 어떤 프로세스가 먼저 실행될 지 우선순위가 설정이 되고 우선순위마다 주어진 할당시간이 다르다. 중간에 끊기면 A 값이 B 로 옮겨지는 경우가 생기기 때문에 task_struct 에 현재 상태를 저장한다.
이 때, 제어권이 돌아오면 run queue 가 되면서 저장되어 있던 레지스터가 실행된다.

[임베디드 애플리케이션 구현 68] 현재 삽입된 디바이스 드라이버의 리스트를 보는 명령어는 무엇인가 ?

[임베디드 애플리케이션 구현 69] System Call Mechanism 에 대해 기술하시오.



커널 모드 진입할 때, 커널은 유저 모드 프로그램에서 커널 모드로 직접 접근을 허락하지 않는다.
시스템콜은 유저가 어떤 상황을 요청하면 커널에서 처리하고 다시 유저에게 돌려준다.
system call : 유저가 커널에게 요청하는 작업을 의미한다.
커널로 진입할 수 있는 방법은 interrupt, trap, system call 이 있다.

[임베디드 애플이케이션 구현 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. Demanding-page 는 실제로 필요한 page 만 물리 메모리로 가져오는 방식을 이야기한다. 필요 page 에 접근하려 하면, 결국 가상 메모리 주소에 대응하는 물리 메모리 주소를 찾아내어, 물리 메모리 주소를 얻어와 하는데, 이 때 필요한 것이 페이지 테이블(page table)이다.

가상 메모리를 사용하면서 생기는 부가 장점으로 다음과 같은 것들이 있다.

1. 공유 메모리 사용

2. Copy-on-write 메커니즘

부모 프로세스를 fork 하여 자식 프로세스를 생성하였을 때, 처음에는 같은 메모리를 사용하도록 하다가 그곳에 Write 가 발생하였을 때 메모리를 copy 하는 것으로 이것 또한 공유 메모리처럼 같은 프레임을 가리키도록 하였다가 복사가 되었을때 새로운 프레임을 할당하면 된다.

3. Memory mapped file

file 을 접근하는 것을 메모리 접근하듯이 페이지를 할당하여 할 수 있도록 하는 것이며, 메모리 접근 속도가 훨씬 더 빠르므로 효율적이라 할 수 있다.

[임베디드 애플리케이션 구현 71] 인자로 파일을 입력 받아 해당 파일의 앞 부분 5 줄을 출력하고 추가적으로 뒷 부분의 5 줄을 출력하는 프로그램을 작성하시오.

[임베디드 애플리케이션 구현 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>

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); //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);
}
```

[임베디드 애플리케이션 구현 73] Linux 에서 fork()를 수행하면 Process 를 생성한다. 이때 부모 프로세스를 gdb 에서 디버깅하고자하면 어떤 명령어를 입력해야 하는가 ?
set fork-follow-mode parent
set fork-follow-mode child / 자식 프로세스의 경우

[임베디드 애플리케이션 구현 74] C.O.W Architecture 에 대해 기술하시오.
  Copy On Write 의 약자로, fork()함수로 인해 자식 프로세스가 만들어졌을 때 fork()를 포함한 밑의 명령어들을 복사하여 실행하게 되는데 밑의 명령어들을 수행하기 위해 필요한 변수 값 같은 것들이 필요할 때 복사가 이루어지게 된다. 즉, 메모리에 쓰기를 사용할 때 복사가 이루어지게 된다. text 가 먼저 필요해서 text 를 복사하고(기계어를 돌기 위해서 먼저 복사해야 한다), 그 다음은 stack 영역을, 다음은 data 를 복사한다. 즉, 메모리 쓰기 작업할 때 복사한다.

[임베디드 애플리케이션 구현 75] Blocking 연산과 Non-Blocking 연산의 차이점에 대해 기술하시오.
non blocking 은 처리할 일이 많으면 예약해놓고 순차적으로 처리해준다. blocking 인 경우, 지정된 동작이 될 때까지 다른 프로세스에 제어권을 넘기지 않고 기다리고 있다. 그래서 다수가 빠르게 통신할 때에는 non-blocking 이 좋고 순차적으로 진행 되어야 할 때는 blocking 이 좋다. 즉, blocking 연산은 하나

의 연산을 다 끝내고 다음 연산을 처리하는 동기 처리이고, non-blocking 연산은 상황에 따라 연산을 처리하는 비동기 처리이다.

[임베디드 애플리케이션 구현 76] 자식이 정상 종료되었는지 비정상 종료되었는지 파악하는 프로그램을 작성하시오.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

void term_status(int status) // 정상종료인지 비정상인지 확인
{
   if(WIFEXITED(status))
     printf("(exit)status : 0x%x₩n", WEXITSTATUS(status));
   else if(WTERMSIG(status))
     printf("(signal)status : 0x%x, %s₩n", status &0x7f, WCOREDUMP(status) ? "core dumped"
: "");
}

int main(void)
{
  pid_t pid;
  int status;
  if((pid = fork()) > 0 )
    {
    wait(&status);
    term_status(status);
     }
  else if( pid == 0)
    abort();                  //여기를 exit 로 작성하면 정상 종료로 바꾸어서 확인 할 수 있다.
  else
  {
     perror("fork()");
     exit(-1);
   }
  return 0;
}
```

[임베디드 애플리케이션 구현 77] 데몬 프로세스를 작성하시오. 잠시 동안 데몬이 아니고 영구히 데몬이 되게 하시오.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.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;
}
```

[임베디드 애플리케이션 구현 78] SIGINT 는 무시하고 SIGQUIT 을 맞으면 죽는 프로그램을 작성하시오.

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

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

[임베디드 애플리케이션 구현 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.
 함수 호출은 스택을 만든다. 그러나 goto 는 스택을 할당 해지할 능력이 없어 메인 함수에서 나와 따로 함수를 만들 경우 쓸 수가 없다.

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

jmp_buf env;

int func(void)
{

        printf("I'm func!\n");
        longjmp(env,1);
}

int main(void)
{
        int ret;
        if((ret=setjmp(env)) == 0)
        {
                func();
        }
        else if(ret>0 )
        {
                printf("hi\n");
```

```
        }

        return 0;
}
```

[임베디드 애플리케이션 구현 80] 리눅스에서 말하는 File Descriptor(fd)란 무엇인가 ?
파일을 관리하기 위해 운영체제가 필요로 하는 파일의 정보를 가지고 있는 것이다. 핵심 철학인 '모든 것은 파일이다.' 때문에 가능한 것이다. 일반적인 정규파일(Regular File)에서부터 디렉토리(Directory), 소켓(Socket), 파이프(PIPE), 블록 디바이스, 캐릭터 디바이스 등등 모든 객체들은 파일로써 관리된다. 시스템에서 프로세스가 이 파일들을 접근할 때에 파일 디스크립터(File Descriptor)라는 개념을 이용한다.

[임베디드 애플리케이션 구현 81] stat(argv[2], &buf)일때 stat System Call 을 통해 채운 buf.st_mode 의 값에 대해 기술하시오.
파일의 크기, 권한, 생성일시, 최종 변경일등 파일의 상태나 정보를 얻는 함수로 다음과 같은 값을 얻을 수 있다.

S_ISDIR → 디렉토리
S_ISREG → 파일
S_ISFIFO → 파이프
S_ISLNK → 바로가기 파일
S_ISSOCK → 소켓
S_ISCHR → 캐릭터 디바이스
S_ISBLK → 블록 디바이스

[임베디드 애플리케이션 구현 82] 프로세스들은 최소 메모리를 관리하기 위한 mm, 파일 디스크립터인 fd_array, 그리고 signal 을 포함하고 있는데 그 이유에 대해 기술하시오.
pid 를 공유하고 있는 모든 쓰레드(쓰레드 그룹)들간에 시그널을 공유하는 매커니즘이 필요하기 때문이다.

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

[임베디드 애플리케이션 구현 84] 이번에는 랜덤한 이름(길이도 랜덤)을 가지도록 디렉토리를 3 개 만들어보자! (너무 길면 힘드니까 적당한 크기로 잡도록함)

[임베디드 애플리케이션 구현 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)

[임베디드 애플리케이션 구현 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.

[임베디드 애플리케이션 구현 87] 클라우드 기술의 핵심인 OS 가상화 기술에 대한 질문이다. OS 가상화에서 핵심에 해당하는 3 가지를 기술하시오.
1. 가상화는 서버의 이용률을 높이고 관리 부하를 줄일 수 있다.
2. 가상화는 각 사용자의 수행 환경을 다른 환경들로부터 고립시킬 수 있다.
3. 가상화는 여러 물리 자원들을 단일한 가상 자원으로 집합할 수 있다.

[임베디드 애플리케이션 구현 88] 반 인원이 모두 참여할 수 있는 채팅 프로그램을 구현하시오.

[임베디드 애플리케이션 구현 89] 88 번 답에 도배를 방지 기능을 추가하시오.

[임베디드 애플리케이션 구현 90] 89 번 조차도 공격할 수 있는 프로그램을 작성하시오.

[임베디드 애플리케이션 구현 91] 네트워크 상에서 구조체를 전달할 수 있게 프로그래밍 하시오.

[임베디드 애플리케이션 구현 92] 91 번을 응용하여 Queue 와 Network 프로그래밍을 연동하시오.

[임베디드 애플리케이션 구현 93] Critical Section 이 무엇인지 기술하시오.
임계 영역이라고도 하며, 둘 이상의 쓰레드(쓰레드는 메모리를 공유하기 때문)가 동시에 접근해서는 안되는 공간이다. 여러 테스크가 동시에 접근해서 정보가 꼬일 수 있는 공간이다.

[임베디드 애플리케이션 구현 94] 유저에서 fork() 를 수행할때 벌어지는 일들 전부를 실제 소스 코드 차원에서 해석하도록 하시오.
유저가 fork()를 구동 시키면 C library 로 가고 시스템 콜을 불러 제어권이 커널로 넘어간다. 결국은 do_fork()가 함수를 구동 시켜주고 kernel_thread()에서 관리한다.

```
long _do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr,
              unsigned long tls)
{
        struct task_struct *p;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the event
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if ((clone_flags & CSIGNAL) != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }

        p = copy_process(clone_flags, stack_start, stack_size,
                         child_tidptr, NULL, trace, tls);
        /*
         * Do this prior waking up the new thread - the thread pointer
         * might get invalid after that point, if the thread exits quickly.
         */
        if (!IS_ERR(p)) {
                struct completion vfork;
                struct pid *pid;

                trace_sched_process_fork(current, p);

                pid = get_task_pid(p, PIDTYPE_PID);
                nr = pid_vnr(pid);

                if (clone_flags & CLONE_PARENT_SETTID)
```

```
                put_user(nr, parent_tidptr);

        if (clone_flags & CLONE_VFORK) {
    p->vfork_done = &vfork;
                init_completion(&vfork);
                get_task_struct(p);
        }

        wake_up_new_task(p);

        /* forking complete and child started to run, tell ptracer */
        if (unlikely(trace))
                ptrace_event_pid(trace, pid);

        if (clone_flags & CLONE_VFORK) {
                if (!wait_for_vfork_done(p, &vfork))
                        ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
        }

        put_pid(pid);
    } else {
        nr = PTR_ERR(p);
    }
    return nr;
}
```

[임베디드 애플리케이션 구현 95] 리눅스 커널의 arch 디렉토리에 대해서 설명하시오.

[임베디드 애플리케이션 구현 96] 95 번 문제에서 arm 디렉토리 내부에 대해 설명하도록 하시오.

[임베디드 애플리케이션 구현 97] 리눅스 커널 arch 디렉토리에서 c6x 가 무엇인지 기술하시오.

[임베디드 애플리케이션 구현 98] Intel 아키텍처에서 실제 HW 인터럽트를 어떤 함수를 가지고 처리하게 되는지 코드와 함께 설명하시오.

[임베디드 애플리케이션 구현 99] ARM 에서 System Call 을 사용할 때 사용하는 레지스터를 적으시오.

[임베디드 애플리케이션 구현 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.

이번 기간에 제 자신을 반성하게 만드는 말은 '인간의 욕심은 끝이 없고 같은 실수를 반복한다.' 라고 생각합니다. 그리고 수많은 자기 합리화가 제 자신을 깍아 먹었다고 생각합니다. 다시 이해해보겠다고, 책을 다시 보아도 제대로 기억나지 않고 이해도 못한 채 결국 과제만 제출하는데 급급했었습니다. 아무리 자는 시간을 줄인다고 해도 효율이 제대로 나지 않고, 그저 밖에 보이는 결과만 집착했던 것 같습니다. 제대로 이해하지도 수업을 따라가지도 못한채 그저 '자는 시간 줄인 것'과 '과제'라는 것에만 집착하여 나는 이만큼 열심히 했다는 자기 합리화가 쌓여서 지금의 제 상태를 만들었다고 생각합니다. '나는 열심히 하고 있어.', '나는 비관련 전공자야.'라는 자기 합리화와 보이는 것에 대한 욕심이 같은 실수를 만들었다고 생각합니다.

저는 여전히 코드를 작성하지 못하여 프로그래밍은 제대로 된 결과가 나오지 않고, 거기다 커널은 제대로 하지도 못했습니다. 남아서 공부할 때, 정말 열심히 하는 학우들을 보면 저는 정말 죽도록 열심히 하지 않았다는 것을 매번 깨닫게 됩니다. 정말 죽도록 열심히 했다면 지금의 시험 결과는 달라져 있으리라고 생각

합니다. 같은 시간이 주어지는데, 이 정도밖에 못했다는 것은 정말 죽도록 열심히 하지 않았다는 뜻이라고 생각합니다.
과정도 중요하지만 결과가 현 시점에서는 더 중요하다고 생각합니다. 결국은 취직을 위한 것이고 취직을 하기 위해선 제가 얼마나 회사에 메리트가 있는가를 보여줄 수 있어야 하니까요. 하지만 이런 마음가짐과 상태로 나아간다면 지금과 앞으로의 전 얼마나 이점이 있을지 장담할 수 없으리라 생각됩니다. 결과가 나오지 않는 이유는 변명밖에 되지 않을 것이라고 생각합니다.
비관련 전공자라는 단어는 저번에도 느꼈듯이 버리고 다시 시작하겠습니다. 자는 시간을 더 줄이지는 못하더라도 더 효과적으로 공부할 수 있도록 노력하겠습니다. 그리고 이해와 응용을 할 수 있도록 더 제자신을 집중하고 주변에 집중을 흐트려트릴 수 있는 것은 다 버리도록 하겠습니다. 이번에 이해 못한 커널과 잘 사용하지 못하는 C 언어와 자료구조를 시간 날 때마다 다시 연습할 것이고, 수학적 문제도 찬찬히 보아서 앞으로 학습의 준비를 하겠습니다. 마음가짐을 바꾸고 다시 시작할 수 있도록 하겠습니다. 다시 저런 생각이 들 때마다 자신을 채찍질 할 수 있도록 노력하겠습니다.

시험 결과가 미흡하여 열심히 가르쳐주시는 선생님께 죄송합니다.