TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – GJ (박현우)
uc820@naver.com

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 1번

오답노트

```c
#include<stdio.h>
#include<fcntl.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);

    /*
       text = buf;
       buf에 첫 문자가 c가 아니거나  두번 째 문자가 띄어쓰기가 아니면
       NULL 반환.
     */
    if( text[0] != 'c' || text[1] != ' ')
        return NULL;

    /*
       buf에 마지막 글자가 c가 아니거나 마지막에서 두 번째 글자가 .이 아니라면
       NULL 반환.
     */
    if( text[text_len - 1] != 'c' || text[text_len - 2] != '.')
        return NULL;

    /* 위에 조건을 다 만족하지 않으면
       filename에 buf값을 3번째 문자부터 차례대로 입력.
     */
    for(i = 2; i < text_len; i++){
        if(text[i] == ' ' || text[i] == '\t')
            return NULL;
        filename[i -2] = text [i];
    }

    return filename;

}
```

```c
int main(void){

    int fo;
    int fd, ret;
    char buf[1024];
    char *string = NULL;
    fd = open("myfifo", O_RDWR);  // pipe 열기

    fcntl(0, F_SETFL, O_NONBLOCK); // 키보드 입력에 대해 논 블락킹
    fcntl(fd, F_SETFL, O_NONBLOCK);  // 파이프 통신에 대해 논 블락킹

    for(;;){
        if( (ret = read(0, buf, sizeof(buf))) > 0 ){// 키보드 입력 받아 buf에 저
장
            buf[ret - 1] = 0;   // buf에 마지막 글자가 '\n'이니 띄어쓰기 없앰.
            printf("Keyboard Input : [%s]\n", buf);
            string = check_text(buf);   //check_text로 조건에 맞는 문자열 반환.
            printf("String : %s\n", string);
        }

        if( (ret = read(fd, buf, sizeof(buf))) > 0){ //파이프통신 입력 받음.

            buf[ret - 1] = 0;    // 위와 동일
            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);
        // string으된 파일을 하나 만듬.
        close(fo);
    }

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

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 4번

정답

파일

내가 쓴 답안

커널

오답노트

모든 것은 파일로 관리 된다는 것을 인지하고 있었으나,

문제를 잘못 읽은 것 같다.

Kernel이 모든 것을 관리한다고 생각했다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 8번

정답

ELF(Executalbe Linkable Format)

내가 쓴 답안

ext2

오답노트

이것도 문제를 잘못 읽었다. 아...

파일포맷이면 당연히 elf인데,

파일시스템...을 쓰다니 멍청했다.

앞으로는 더욱더 집중해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 11번 ★

정답

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

arch/x86/include/asm/current.h

여기서 get_current() 매크로를 살펴보면 ARM 의 경우에는 아래 파일에

include/asm-generic/current.h

thread_info->task 를 확인할 수 있다.
x86 의 경우에는 동일한 파일 위치에서
this_cpu_read_stable() 함수에 의해 동작한다.

이 부분을 살펴보면 아래 파일

arch/x86/include/asm/percpu.h 에서

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

내가 쓴 답안

current는 thread_info->task 이고,
현재 구동중인 task의 *task_struct이다.

오답노트

단순히 current는 현재 구동중인 task라고 생각했다.

x86이 다르고 arm이 다른데, 매크로를 통해서 관리 된다는

점도 제대로 파악을 못했다. 다시 소스 드라이빙하면서

재확인을 해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 21번

정답

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

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

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

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

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

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

내가 쓴 답안

zone_hi

오답노트

...ㅋㅋㅋㅋ 진짜 웃음만 나온다.

아는 내용인데 문제가 너무 많아서 인지 저렇게만 쓰고

넘어갔다.

제대로 확인 못한 나의 잘못이다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 22번

정답

물리 메모리의 최소 단위는 Page Frame이라 하며
이것을 SW 적 개념으로 구현한 구조체의 이름은 page다.

내가 쓴 답안

4k, mm_struct

오답노트

이것도 왜 저렇게 쓴지 모르겠다. 아무래도 정신상태가

온전치 못했나 보다.

Page frame과 4k, page라고 안 쓰고 뭘 했지...

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 25번

정답

task_struct 구조체에 있는 mm_struct 내에 start_code, end_code등으로 기록됨

내가 쓴 답안

task_struct 밑 mm_struct 밑 vm_area_struct 밑 vm_area

오답노트

잘못된 개념을 가지고 있던 것 같다. Stack, heap, data, text는

Start_code, end_code와 같은 걸로 기록되는데,

vm_area_struct는 세그멘테이션이니 , 가상메모리 관련 구조체인

저기에 기록 된다고 생각했다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 28, 29번

정답

C언어를 사용하기 위해서는 반드시 Stack이 필요하다.
Kernel 영역에서도 동작하는 코드가 올라가기 위한 Text 영역
전역 변수가 있는 Data 영역, 동적 할당하는 Heap,
지역 변수를 사용하는 Stack이 존재한다.
이는 역시 User 영역에서도 동일하므로 양쪽에 모두 메모리 공간이 구성된다.

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

내가 쓴 답안

VM과 PM을 연결 시키려면 뭔가 가상 주소를 물리 주소로 변환하는 방법이 필요하다. 이 때 사용되는 것이 '페이지 테이블'이다. 즉, 페이지 테이블은 가상 주소를 물리 주소로 변환하는 주소 변환 정보를 기록한 테이블이 필요하다.
예를 들어, 가상 주소 10000번지에 접근한다고 할 때 1024로 나눠 몫을 페이지 테이블의 인덱스로 사용한다. 나눈 값의 나머지는 해당 인덱스의 엔트리를 탐색하는 용도로 하고 해당 인덱스가 비어 있으면 free한 페이지 프레임를 할당 받는다. 그리고 페이지 테이블에 적재된 페이지 프레임 번호(trap_num)를 기록한다. 이러한 방식으로 페이지 테이블을 통해 가상 주소를 물리 주소로 변환하는 것을 페이징이라고 한다.

오답노트

... 29번을 28번에다가 썼다...

그리고 심지어 28번은 적어둔 내용인데,

아 진짜, 정신을 제대로 못 차렸었다... 아;;

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 30번

정답

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

내가 쓴 답안

페이징시 단점은 물리 메모리를 접근하기 위해 주소 변환 과정이 필요하다는 것이다. 그래서 이러한 과정을 하드웨어적으로 처리한 것을 MMU라고 부른다.

오답노트

음... 이것도 그냥 단순 기능만 쓴 것 같다.

제대로 파악하지 못하고 있었다.

이제 알았으니, 까먹지 말고 공부해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 31번

정답

Sector, 512 byte

내가 쓴 답안

디스크 블록, 4kb

오답노트

Filesystem에서 쓰이는 게 디스크블록인데,

이것도 오개념을 가지고 있었다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 32번

정답

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

내가 쓴 답안

오답노트

오답노트를 해보니, 잘못 쓴 부분도 너무 많고

문제를 잘못 푼 것도 많은 것 같다.

모듈 프로그래밍문제를 밑에서 풀었는데,

저렇게 공란으로 해둔걸 보면,,,

다음부터는 제대로 정신차리고 해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 33번

정답

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.

내가 쓴 답안

open() 함수가 호출 된다고 가정을 한다면, open()함수가 user단에서 호출이 되면 제일 먼저, ax 레지스터에 5가 셋팅이 된다. 그리고 __vector_start(IDT)에서 int 0x80이 호출되면서 user에서 kernel로 제어권이 넘어간다. 다음으로는 IDT에서 128번인 system call이 불려지고 ia32_sys_call_table에서 5번에 해당하는 함수가 최종적으로 동작하는 방식으로 system call mechanism이 작동한다.

오답노트

Open 시스템 콜 메커니즘에 대해

생각한 것과는 다른 것 같다.

다시 공부가 필요하다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 35번

정답

VFS는 Super Block인 ext2_sb_info와 같은 것들을 읽어 super_block에 채워넣는다.
그리고 읽고자 하는 파일에 대한 메타 정보를
ext2_inode와 같은 것들을 읽어 inode에 채운다.

이후에 디렉토리 엔트리인 ext2_dir_entry를 읽어 dentry 구조체에 채우고
현재 위치 정보에 대한 정보를 위해 path 구조체를 채운 이후
실제 File에 대한 상세한 정보를 기록하기 위해 file 구조체를 채운다.
각각 적절한 값을 채워넣어 실제 필요한 파일을 Task와 연결시킨다.

내가 쓴 답안

VFS는 구조체로 파일시스템들을 통합해둠으로 inode는 파일 시스템을 고려하지 않게 된다. 그렇기 때문에, 빈 inode 객체를 하나 만들고 파일시스템 내부 함수로 수퍼블록을 채운 뒤 리턴하고 sys_open()을 호출하여 빈 inode 객체를 채워서 다시 리턴한다. inode에 채워진 정보를 바탕으로 inode_opreations를 ext2면 ext2로 ext4면 ext4로 채운다. 이 방식을 채택하여 다양한 파일 시스템을 받을 수 있다.

오답노트

얼추 비슷하게 쓴 것 같지만,

부족한 부분에 대해서 더 공부를 해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 39번

정답

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

내가 쓴 답안

---
kernel 내에 irq_desc에 모니터에 해당하는 interrupt 주번호를 통해서 인터럽트를 공유한다.

오답노트

거의 맞았지만, 디테일한 부분을 추가해서

공부해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 41번

정답

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

내가 쓴 답안

intel의 경우는 ia32_sys_call_table이고
arm의 경우는 linux/arch/arm/kernel/calls.S
에 저장한다.

오답노트

System call번호가 정확히 어디 레지스터에

저장되는지를 잘 몰랐던 것 같다.

더 공부해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 43번

정답

Intel 의 경우엔 CR3
ARM 의 경우엔 CP15

내가 쓴 답안

intel cr3 , arm TTBR0, TTBR 1을 사용함.

오답노트

arm 관련 서적에서 page directory 레지스터를

TTBR0라고 본 것 같은데,

잘못 본 것 같다...

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 44번

정답

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

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

내가 쓴 답안

기억장치는 RAM이건 하드디스크이건 한정된 자원을 아껴 사용해야 한다. 그래서, 메모리관리와 파일시스템 모두 내/외부 단편화의 최소화를 위해 노력해야 한다.

오답노트

메모리 할당이 필요한 이유가 아니라

메모리 관리가 필요한 이유를 쓴 것 같다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 45, 46번

정답

vmalloc()

kmalloc()

내가 쓴 답안

45. 같은 파일에 속한 디스크 블록들을 연속적으로 할당하지 않는 기법

46. 파일에게 연속된 디스크 블록을 할당하는 기법

오답노트

정의로 썼다.

함수로 쓰는 건지 몰랐다...

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 51번

정답

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

내가 쓴 답안

task_struct -> mm_struct -> file -> inode -> cdev -> dev 변수

오답노트

실제 Device의 Major Number와 Minor Number를 저장하는 변수는 어떤 구조체의 어떤 변수

커널에서 조금만 오개념만 잡혀도 참 어려운 것 같다.

후... 더 공부해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 52번 ★★

정답

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

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

chrdevs 배열에 Major Number에 해당하는 Index에
file_operations를 Wrapping한 구조체를 저장해둔다.

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

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

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

내가 쓴 답안

kernel에서 이 파일의 아이노드 객체를 통해서 주/부 번호를 얻고 장치 파일의 유형을 얻는다. 장치 파일의 유형을 통해 cdev_map 자료구조에 접근하여 주 번호를 인자로 cdev구조체를 검색한다. 검색 결과, 앞서 커널에 file_operations 구조체를 찾게 되고, 따라서 이 driver에서 작성한 함수로 file_operations가 채워진다. open()한 파일이 있으면 거기에 대한 하는 write나 read가 내가 만든 것으로 동작하는 것을 볼 수 있다.

오답노트

디테일함을 더 공부해야겠다.

아직 모르는 부분이 너무 많다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 55번

정답

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

내가 쓴 답안

위의 OoO 개념으로 Compile-Time에 Compiler Level에서 직접 수행한다.
Compiler가 직접 Compile-Time에 분석하므로
최적화의 수준이 높으면 높을수록 Compile Timing이 느려질 것이다.
그러나 성능만큼은 확실하게 보장할 수 있을 것이다.

오답노트

못 풀었다. 비순차 실행이 저런 건지 기억도 못했다.ㅋㅋㅋ

앞으로 잘 기억해둬야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 56번

정답

위의 OoO 개념으로 Compile-Time에 Compiler Level에서 직접 수행한다.
Compiler가 직접 Compile-Time에 분석하므로
최적화의 수준이 높으면 높을수록 Compile Timing이 느려질 것이다.
그러나 성능만큼은 확실하게 보장할 수 있을 것이다.

내가 쓴 답안

오답노트

이것도 못 풀었다.

컴파일러의 인스트럭션 스케쥴링에 대해서

이제 알게 됐으니 까먹지 말고 공부해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 58번

정답

Dynamic Instruction Scheduling

내가 쓴 답안

오답노트

하드웨어 만으로 인스트럭션 스케줄링을 하다니,

저런 방법이 있었군...

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 60번

정답

ISA(Instruction Set Architecture) : 명령어 집합

내가 쓴 답안

instruction cpu

오답노트

Cpu 명령어 인데, 같은 뜻 아닐까? ;;; ㅋㅋㅋ

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 61번

정답

Hyper Threading 은 Pentium 4 에서
최초로 구현된 SMT(Simultaneous Multi-Threading) 기술명이다.
Kernel 내에서 살펴봤던 Context를 HW 차원에서 지원하기 때문에
실제 1개 있는 CPU가 논리적으로 2개가 있는것처럼 보이게 된다.
HW상에서 회로로 이를 구현하는 것인데
Kernel에서 Context Switching에선 Register들을 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.

내가 쓴 답안

61. intel의 하이퍼스레딩은 소프트웨어에서
하는 sys_fork를 하드웨어적으로 했다고 보면
된다. 무슨 말인 즉, cpu 1개 짜리를 cpu 2개
처럼 작동하는 하드웨어 방식을 만든 것이다.
예를 들면, quad core의 thread는 4개인데 하
이퍼스레딩을 하면 thread가 8개가 된다.

오답노트

하드웨어에서 context switching을 소프트웨어가 아닌

하드웨어로 구현했다고 생각해야겠다.

Context switching 비용의 최소화

근데 용어가 너무 어렵다...

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 62번 ★★★★★

정답

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

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.

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.

내가 쓴 답안

오답노트

비슷한 내용으로 글을 쓴 것은 같으나,

아직 한참 부족하다...

너무 내용이 머리에 붕 떠있다.

이걸 보면서 정리하고 더 공부해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 64번

오답노트 Server

```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_in  si;
typedef struct sockaddr *   sp;

#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;
    pid_t pid;

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

    /*
      자식 프로세스가 죽으면 signal이 발생함.
     */
    act.sa_handler = read_cproc;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    state = sigaction(SIGCHLD, &act, 0);
```

```c
    for(;;){

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

        /*
           clnt를 계속 받기 위함.
         */
        if(clnt_sock == -1){
            printf("clnt die\n");
            continue;
        }
        else
            puts("New Client Connected!\n");

        pid = fork();

        /*
           자식 프로세스가 죽더라도 계속 for문을 돔.
         */
        if(pid == -1){
            printf("c proc dead\n");
            close(clnt_sock);
            continue;
        }

        if(!pid){ //자식 프로세스는 serv_sock을 닫음
            close(serv_sock);

            /*
               read로 clnt에서 보내는 문자열을 buf에 저장
               write로 1초마다 hello를 보냄.
             */
            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   // 부모 프로세스는 clnt_sock를 닫음
            close(clnt_sock);

    }
    close(serv_sock);


    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 64번

오답노트 Client

```
void write_proc(int sock, d *buf)
{
    char msg[32] = "Hi";

    for(;;)
    {
        write(sock, msg, strlen(msg));
        sleep(1);  // 1초마다 server에게 Hi를 보냄.
    }
}

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)  // 자식 프로세스는 write 전용
        write_proc(sock, (d *)&struct_data);
    else    // 부모 프로세스는 read 전용
        read_proc(sock, (d *)&struct_data);

    close(sock);

    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 65번

오답노트 shm.h , shmlib.c

```c
#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);
```

```c
#include "65_shm.h"

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

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

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

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

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 65번

오답노트 recv.c , send.c

```c
#include "65_shm.h"

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

    /*
       shared Memory 생성 0x888로 생성
     */
    mid = CreateSHM(0x888);

    /*
       물리메모리의 주소를 얻음
     */
    p = GetPtrSHM(mid);

    getchar();

    /*
       파일에 내용을 읽어와서 출력
     */
    printf("name : %s\nbuf : %s\n", p -> name, p -> buf);

    FreePtrSHM(p);

    return 0;
}
```

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

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

    /*
       메모리아이디, 페이지프레임의 아이디값을 얻음(권한을 얻음)
     */
    mid = OpenSHM(0x888);

    /*
       진짜 쉐어드 메모리의 포인터 값을 얻음, 공유 메모리의 물리주소를 얻음.
     */
    p = GetPtrSHM(mid);

    getchar();

    /*
       argv[1]로 얻은 임의의 파일명으로 fd를 얻음
     */
    fd = open(argv[1], O_RDONLY);

    // 해당 fd에 내용을 읽어 buf에 저장
    ret = read(fd, buf, sizeof(buf));

    //마지막 문자를 0으로 해서 문자의 끝을 NULL로 지정.
    buf[ret - 1] = 0;

    // p에 argv[1]과 buf복사
    strcpy(p -> name, argv[1]);
    strcpy(p -> buf, buf);

    //shared Memory 해제. 시간이 걸림.
    FreePtrSHM(p);

    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 67번

정답

CPU마다 RQ, WQ가 1개씩 존재한다.
active 배열과 expired 배열이 존재해서
어떤 우선순위의 Process가 현재 올라가 있는지 bitmap을 체크하게되며
queue에는 만들어진 task_struct가 들어가 있게 된다.
즉, bitmap을 보고 우선순위에 해당하는 것이 존재하면
빠르게 queue[번호]로 접근해서 해당 task_struct를 RQ에 넣거나
주어진 Time Slice가 다했는데 수행할 작업이 남아 있다면
RQ에서 WQ로 집어넣는등의 작업을 수행한다.
결국 Scheduling이란 작업이 Multi-Tasking을 지원하는데 있어 핵심인 기술이다.
여러 Task들을 동시다발적으로 동작하는것처럼 보이게하는 트릭이라 할 수 있겠다.

내가 쓴 답안

TASK A와 TASK B가 있으면, 두 개의 kernel stack이 할당된다. 그리고 RQ에 A,B TASK가 들어간다. 일반적으로는 CFS 스케쥴링 방식에 의해서 태스크가 관리된다. A가 먼저 cpu를 점유했다면, A의 thread_union 밑에 thread_info의 cpu_context_save에 A의 현재 위치 및 정보가 저장된다. A는 RQ에서 active 상태가 되면서 CPU를 점유하게 된다. 그리고 해당 스케쥴링 시간이 끝나 완전히 프로세스가 종료되면 expired상태가 되며 만약 실행해야 할 일이 남아 있다면, WQ로 빠져 다음 스케쥴링 시간 까지 대기한다. RQ에서 스케쥴링을 기다리던 B는 A가 끝나면, A와 마찬가지로 kernel stack에 자신의 정보를 저장하고 cpu에 올라가 active 상태가 된다. 위와 같은 방식을 아주 빠르게 진행한다면, Multi-tasking한다.

오답노트

Time slice와 bitmap 내용을 추가하면

완벽할 것 같다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 76번

오답노트

```c
#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);
        /*
           프로세스가 정상 종료되었다면, 하위 8비트에는 0이 저장되고
           상위 8비트에는 종료되게 한 exit 함수의 인수가 기록된다.

           하위 8비트가 0이면 정상종료.
         */
        if( (status & 0xff) == 0)
            printf("(정상 종료)status : 0x%x\n", WEXITSTATUS(status));
        /*
           반면 비정상 종료하면 상위 8비트에 0이 저장되고 하위 8비트에
           프로세스를 종료한 시그널의 번호가 저장된다.

           status를 오른쪽으로 8비트 밀면 상위 8비트가 0이므로
           8비트와 and연산해서 0이되면 비정상종료.
         */
        else if( ((status >> 8) & 0xff) == 0)
            printf("(비정상 종료)status : 0x%x\n", status & 0x7f);

    }
    else if(pid ==0)
        abort();  // abort는 kill 6임. SIGABRT
    else{
        perror("fork() ");
        exit(-1);
    }
    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 77번

오답노트

```c
#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;

    /*
       부모 프로세스를 종료.
     */
    if(fork() > 0)
        exit(0);

    setsid();  // 프로세스 상태를 ? 로 만든다.
    chdir("/");  // 실행 시켜야 할 파일이 있으니 root로 보냄.
    umask(0); // 권한 설정
    for(i =0; i< 64; i++)  // 리눅스는 기본 64개 파일을 열으니 전부 닫음.
        close(i);
    signal(SIGCHLD, SIG_IGN); // 자식이 생기면 무시하라 - signal manual
    return 0;

}


int main(void){

    daemon_init();

    //데몬에서 동작시키고자 하는 작업을 이곳에서 작성한다.
    for(;;)
        ;
    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 78번

오답노트

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

int main(void){

    signal(SIGINT, SIG_IGN);  // SIGINT는 무시.
    for(;;)
        pause();  // 시그널을 기다린다.
    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 83번

오답노트

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


int main(int argc, char *argv[]){

    if(argc != 2){
        printf("Usasge: exe dir_name\n");
        exit(-1);
    }

    mkdir(argv[1], 0755);// 디렉토리 만들기

    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 84번

오답노트

```c
#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;     // 길이를  4 ~ 7까지

    for(i = 0; i<len; i++)
        buf[i] = rand() % 26 + 97;    //  97에서 122까지

    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);// char 8개 동적할당 받음.
        strcpy(dname[i], rand_name());  // buf값 dname에 복사
    }
}

int main(void){

    int i;
    char *dname[3] = {0};

    srand(time(NULL));
    make_rand_dname(dname);   // char형 포인터 배열로 랜덤 이름 3개 만듬.

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

    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 85번

오답노트

```c
void make_rand_file(void){

    int i, j, fd, len, cnt;
    char buf[11] = "\0";

    len = rand() % 6 + 5;   // len 5 ~ 10
    cnt = rand() % 4 + 2;  //  cnt 2 ~ 5

    for(i = 0; i<cnt; i++){
        for( j = 0; j<len; j++)   // 5 ~ 10 랜덤하게 buf 이름 만듬.
            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++){  // 3번 반복
        chdir(dname[i]);   //dname에 해당하는 이름으로 디렉토리 이동
        make_rand_file();  // 랜덤이름 파일 만들기
        chdir("..");  // 상위 디렉토리로 이동
    }
}

int main(void){

    int i;
    char *dname[3] = {0};

    srand(time(NULL));
    make_rand_dname(dname);   // char형 포인터 배열로 랜덤 이름 3개 만듬.

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

    lookup_dname(dname);
    return 0;
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 86번

오답노트

```c
void find_abc(void){

    struct dirent *p;
    struct stat buf;
    DIR *dp;
    int i, len;

    dp = opendir("."); // 현재 디렉토리를 열고 디렉토리 포인터를 받음.

    while(p = readdir(dp)){  // 개방한 디렉토리를 읽는다.

        stat(p->d_name, &buf);  //해당 디렉토리 상태를 buf에 저장한다.
        len = strlen(p->d_name);  // 디렉토리 이름 길이

        for(i = 0; i< len; i++){  // a 또는 b 또는 c가 있는지 확인한다.
            if(!strncmp(&p->d_name[i], "a", 1) ||
                    !strncmp(&p->d_name[i], "a", 1) ||
                    !strncmp(&p->d_name[i], "a", 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("..");
    }
}
```

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 94번

정답

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

내가 쓴 답안

오답노트

33번에서 open() 했던 메커니즘을 그대로

썼으면 되는 건데, 아쉽다.

더 자세히 알게 됐으니, 더 공부를 해야겠다.

1. 리눅스 커널 & 시스템 & 네트워크 프로그래밍 99번

정답

r7

내가 쓴 답안

arm TTBR0, TTBR 1을 사용함.

오답노트

R7 다신 잊어 먹지 않으리다.