TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

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

1. 코드

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

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

    int fd, ret;
    int fd2;
    char buf[1024];
    mkfifo("myfifo");
    fd = open("myfifo",O_RDWR);

    for(;;){

        ret =read(fd, buf, sizeof(buf));
        buf[ret - 1] = 0;
        fd2 = open(buf, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        printf("Pipe Input : [%s]\n",buf);

    }

    return 0;
```

결과

```
phw@phw-Z20NH-AS51B5U:~/test2$ cat > myfifo
type.c
type.c
^C
phw@phw-Z20NH-AS51B5U:~/test2$ cat > myfifo
type.c
type.c
type.c
```

```
phw@phw-Z20NH-AS51B5U:~/test2$ ./a.out
Pipe Input : [type.c]
^C
phw@phw-Z20NH-AS51B5U:~/test2$ ls
1.c  a.out  myfifo  type.c
phw@phw-Z20NH-AS51B5U:~/test2$
```

2. 코드

```c
#include<stdio.h>
#include<signal.h>
#include<fcntl.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>


int clap_count = 0;  // signal 눌른 횟수

int u_clap_count = 0;
int count = 1;  //1 , 2 ,3 숫자 증가

void my_sig1(int signo){
    printf("Time over!\n");
    exit(0);
}
void my_sig2(int signo){
    u_clap_count++;
    printf("\nClap!\n");
}

void check_369(void){

    int x = count / 10;

    if( count % 10 == 3 || count % 10 == 6 || count % 10 == 9){
        clap_count++;
    }
    else if( count % 100 == 30 || count % 100 == 60 || count % 100 == 90){
        clap_count++;
    }

    printf("3 6 9 횟수 %d\n", clap_count);
}
```

```c
int check_num(char *msg){

    int num = atoi(msg);

    check_369();

    printf("num = %d\n", num);
    if( count == num && clap_count == 0){
        count++;
        printf("good\n");
        return 1;
    }else if( clap_count != 0 &&(clap_count == u_clap_count)){
        count++;
        clap_count = 0;
        u_clap_count = 0;
        printf("good\n");
        return 1;
    }else{
        return 0;
    }
}
void game_start(void){

    char buf[1024];
    int ret, c_ret;

    signal(SIGALRM, my_sig1);
    signal(SIGINT, my_sig2);

    printf(" 3 6 9 game is started !! \n");
re:
    alarm(2);
    ret = read(0, buf, sizeof(buf));
    buf[ret -1] = 0;

    if(c_ret = check_num(buf)){
        goto re;
    }
    else{
        printf("wrong answer\n");
        exit(0);
    }
    alarm(0);


}
```

2.

코드

```c
int main(void){


    game_start();
    return 0;
}
```

결과

```
3 6 9 횟수 0
num = 18
good
^C
Clap!

3 6 9 횟수 1
num = 0
good
20
3 6 9 횟수 0
num = 20
good
21
3 6 9 횟수 0
num = 21
good
22
3 6 9 횟수 0
num = 22
good
^C
Clap!

3 6 9 횟수 1
num = 0
good
24
3 6 9 횟수 0
num = 24
good
225
3 6 9 횟수 0
num = 225
wrong answer
phw@phw-Z20NH-A551B5U:~/test2$ ./a.out
 3 6 9 game is started !!
Time over!
```

3

1) filesystem
2) memory
3) task
4) device
5) Network

4. Kernel

5.
* 사용자 임의대로 재구성이 가능하다.
  -> 사용자 마음대로 리눅스 코드를 customizing할 수가 있다.
* 열약한 환경에서도 HW 자원을 적절히 활용하여 동작한다.
  -> 리눅스는 하드웨어 환경에 맞게 최적화가 잘 되어 있어서 안정적이다.
* 커널의 크기가 작다.
  -> 윈도우보다 작지 실제로 커널이 작은 편이 아니다.
* 완벽한 멀티유저, 멀티태스킹 시스템
  -> 태스크 관리가 잘 되어있다.
* 뛰어난 안정성
  -> 네트워크 장비만 보더라도 안정성이 뛰어나다는 것을 알 수 있다.
* 빠른 업그레이드
  -> 오픈 소스이며 많은 개발자들이 참여하고 있어 업그레이드가 빠르다.
* 강력한 네트워크 지원
  -> 네트워크 장비는 다 리눅스다.
* 풍부한 소프트웨어
  -> GNU에 스톨만님께서 리눅스에서 가장 중요한 gcc 등 중요한 프로그램은 다 드심.

6. 32bit cpu 0~3 G – 사용자
3~4G – 커널

7. page fault가 발생하면 fault를 일으킨 명령어 주소를 eip에 넣어 놓고 page fault handling이 동작하여 페이지 할당을 위해 메모리에 재접근한다. 그리고 메모리 할당이 성공하면 eip를 꺼내 저장된 주소부터 다시 시작한다.

8.
ext2

9.
프로세스는 pid와 tgid가 같으며 스레드의 리더이다.
스레드는 프로세스와 tgid는 같으나 pid가 다르다.

10.
task_struct가 생성된다.

11.
current는 thread_info->task 이고,
현재 구동중인 task의 *task_struct이다.

12.
프로세스는 메모리를 공유하지 않으나, 스레드는 메모리 공유 하므로 세마포어, 스핀락, 뮤테스 같은 것을 해줘야 됨.

13.
문맥교환할 때, 태스크의 현재 실행 위치와 같은 정보를 기억해둔다. 즉, 레지스터에 태스크 정보를 저장하고 그 정보를 관리하기 위함.

14.
deadline이 가장 가까운 태스크를 먼저 스케줄링 하는 방식을 말한다. 이 스케줄링 방식은 rb_tree를 이용하여 태스크의 우선순위를 관리한다.

15.
인터럽트는 특정 사건이 발생하면 그 사건을 먼저 처리 하는 것인데, 만약 인터럽트가 허용되지 않아야 할 중요한 태스크인데 인터럽트가 허용되어 있다면 시스템 자체가 꼬일 것이다. 그러므로 태스크 상태에 관한 정의가 필요하다.

16.
O(N)은 이중연결리스트를 예로 들 수 있다.
시간(t)이 지날수록 cpu에서 처리해야 할 명령어 갯수가 증가하며, O(N)알고리즘은 태스크가 많아지면 소모시간을 예측할 수 없다.
반면, O(1)은 해쉬나 맵 알고리즘으로 시간(t)가 증가해도 cpu가 처리해야 할 명령어의 개수가 동일하다는 장점을 가진다.
하지만, O(N)과 O(1)을 비교해보면 cpu 명령어가 처리해야 할 양이 무식하게 많은 곳(ex.서버)에서는 O(1)이 더 좋다. 그 외의 경우는 O(N)이 좋다.

17.
2번에 위치해 있는 cpu에서 fork()하면 해당 cpu에 task가 배치되는 게 가장 좋다. 단, 해당 cpu의 Runqueue에서 태스크들이 적을 때에 해당한다.
왜냐하면, 각각의 cpu마다 cache가 있는데, 태스크가 적을 때 다른 cpu에 task를 넣게 되면 2번의 캐쉬도 깨지고 2번 task에서 fork가 된 task가 다른 cpu의 Runqueue로 들어가야 하기 때문에 치뤄야 할 비용적인 면이 너무 크기 때문이다.

18.
가장 한적한 cpu 2에 task를 배치하는 게 좋다. cpu에 너무 많은 프로세스가 있다면 load balance()함수를 통해서 속도를 고려하여 runqueue에 가장 적은 task의 수가 있는 곳으로 배치한다. 왜냐하면, 너무 많은 task가 있으면 대기 시간으로 인해 해당 task가 수행되는데 너무 오랜 시간이 걸리기 때문이다. 그래서 비용을 치루더라도 빨리 task가 수행될 수 있도록 cpu 2에 위치 시킨다.

19.
UMA(Uniform-Momory-Access)는 단일 메모리 구조로서 모든 cpu가 메모리 공유를 하며 cpu마다 메모리에 접근하는 속도가 같다.
반면에, NUMA는 메모리 접근 속도가 cpu마다 다르다
UMA의 경우 kernel 내부에 struct pglist_data로 된 변수 contig_page_data를 사용한다.

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;
#ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM *
    struct page *node_mem_map;
#ifdef CONFIG_PAGE_EXTENSION
    struct page_ext *node_page_ext;
#endif
#endif
#ifndef CONFIG_NO_BOOTMEM
    struct bootmem_data *bdata;
```

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

20.
static priority - 0 ~ 99 번
dynimic priority - 100~ 139번

21. zone_hi

22.
4k, mm_struct

23.
Buddy 할당자는 물리 메모리에 4KB보다 큰 용량을 할당할 때 사용된다. 그리고 zone 구조체에 free_area[]로 구축된다. free_area 구조체에 free_list 배열은 list_head 구조체로 되어 있다. free_list의 인덱스 0은 order 0을 의미하고 인덱스1은 order 1을 의미한다. 위와 같은 방식으로 4KB는 order 0, 8KB는 order 1로 맵핑이 되어 있다. 만약에 6kb 용량을 할당하려고 한다면 order 1에서 2page가 비어 있는 곳으로 가서 페이지를 할당하고 free_area.free_list를 갱신한다.

24. 4KB의 공간을 미리 만들고 32byte씩 공간을 쪼개서 자주 사용되면 cache형식으로 가지고 있는다.
kmem_cache를 통해서 다양한 cache를 생성할 수 있다.

25.
task_struct 밑 mm_struct 밑 vm_area_struct 밑 vm_area

26. task_struct 밑 mm_struct 밑 vm_area_struct로 관리.

27. a.out 실행파일은 linux에서 elf 포맷이고 text에서 header만 읽고 header의 내용에 따라 물리 메모리에 올릴지 말지를 결정한다.

28.

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.

30.
페이징시 단점은 물리 메모리를 접근하기 위해 주소 변환 과정이 필요하다는 것이다. 그래서 이러한 과정을 하드웨어적으로 처리한 것을 MMU라고 부른다.

31.
디스크 블록, 4kb

32.

33.
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.

34.
Super block은 rootfile system으로 전체 파일의 위치 정보를 가지고 있다.

35.
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.

36.
외부 인터럽트와 trap으로 나뉘며, 외부 인터럽트는 태스크와 관련 없는 주변장치에서 발생된 비동기적 하드웨어 사건이다. trap은 태스크와 관련이 있는 소프트웨어적 사건으로 나뉜다.

37.
fault, trap, abort로 나뉨.

38.
fault는 page fault, trap은 system call, abort는 divide by zero 이다.

39.
kernel 내에 irq_desc에 모니터에 해당하는 interrupt 주번호를 통해서 인터럽트를 공유한다.

40.
__vector_start

41.
intel의 경우는 ia32_sys_call_table이고 arm의 경우는 linux/arch/arm/kernel/calls.S에 저장한다.

42.
pgd

43.
intel cr3 , arm TTBR0, TTBR 1을 사용함.

44.
기억장치는 RAM이건 하드디스크이건 한정된 자원을 아껴 사용해야 한다. 그래서, 메모리관리와 파일시스템 모두 내/외부 단편화의 최소화를 위해 노력해야 한다.

45. 같은 파일에 속한 디스크 블록들을 연속적으로 할당하지 않는 기법

46. 파일에게 연속된 디스크 블록을 할당하는 기법

47.
세마포어는 공유된 자원을 여러 프로세스가 접근하는 것을 막는 것.
뮤테스는 공유된 자원을 여러 스레드가 접근하는 것을 막는 것이다.

48. insmod

49. rmmod

50.
thread_union은 kernel stack으로서 context switching을 관리하기 위한 구조체이다. arm 기준으로 kernel stack은 8KB을 할당 받고 thread_union 밑에 thread_info에서 프로세스에 관한 레지스터 관련 정보를 저장한다.

51.
task_struct -> mm_struct -> file -> inode -> cdev -> dev 변수

52.
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.

53. 드라이버나 외부 인터페이스 함수의 메모리 배치 때문에 이 메커니즘 방식이 필요로 하다. free_pages()함수나 get_free_pages()함수를 통해서 free_area에 비어 있는 곳을 찾고 device가 필요한 메모리 공간을 kmalloc()과 kfree()를 하면서 물리 메모리에 순차 배치를 시킨다.

54.

```
static ssize_t mydrv_read(struct file *file, char *buf, size_t count, loff_t *ppos){

    int i, sum;

    if( (buf == NULL) || (count <0))
        return -EINVAL;
    if( (mydrv_write_offset - mydrv_read_offset) <= 0)
        return 0;
    count = MIN( (mydrv_write_offset - mydrv_read_offset), count);

    if( copy_to_user(buf, mydrv_data + mydrv_read_offset, count))
        return -EFAULT;
    mydrv_read_offset += count;

    sum = (count * (count + 1))/2;

    return sum;
}
```

```
static int mydrv_release(struct inode *inode, struct file *file){
//  printk("%s\n", __FUNCTION__);
    printk("\n-------Finalize Device Driver\n");
    return 0;
}
```

```
int factorial(int n)
{
       if(n<=1) return 1;
          return n*factorial(n-1);
}

static ssize_t mydrv_write(struct file *file, const char *buf, size_t count, loff_t *
ppos){

    int sum = 0;

    if( (buf == NULL) || (count <0))
        return -EINVAL;
    if( count+mydrv_write_offset >= MYDRV_MAX_LENGTH){
        /* driver space is too small */
        return 0;
    }


    if( copy_from_user(mydrv_data + mydrv_write_offset, buf, count))
        return -EFAULT;
    mydrv_write_offset += count;

    sum = factorial(count);

    return sum;
}
```

코드

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

int main(void){

    int fd, ret, ret2;
    char buf[1024];

    if( (fd = open("/dev/mydevicefile", O_RDWR)) < 0 ){
        return -1;
    }

    ret = read(fd, buf, 10);


    fprintf(stderr, "%d\n", ret);

    ret2 = write(fd, buf, 5);

    fprintf(stderr, "%d\n", ret2);
    close(fd);

    return 0;
}
```

결과

```
[25942.296285] mydrv_open
[25942.296310]
               -------Finalize Device Driver
phw@phw-Z20NH-AS51B5U:~/test/kmp$ sudo ./a.out
15
120
```

53. 드라이버나 외부 인터페이스 함수의 메모리 배치 때문에 이 메커니즘 방식이 필요로 하다. free_pages()함수나 get_free_pages()함수를 통해서 free_area에 비어 있는 곳을 찾고 device가 필요한 메모리 공간을 kmalloc()과 kfree()를 하면서 물리 메모리에 순차 배치를 시킨다.

57.
Intel 은 CISC 구조로 반도체 다이 사이즈가 커서
다양한 Functional Unit 들이 많이 올라갈 수 있고
그로 인하여 전력 소모가 커질 수 밖에 없다.
ARM 은 RISC 구조로 다이 사이즈가 작아서
CISC 만큼 많은 양의 Functional Unit 이 올라가지 못하지만
전력 소모가 적다는 장점과 Load/Store 아키텍처로 구성된다.

58.
MMU

59.
파이프라인은 cpu에 하나씩 있는 cache에서 분기명령어 예측이 자주 틀리면 파이프라인이 깨지게 된다.

60.
instruction cpu

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

62.

```
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespaces */
```

위에 files에서

```
  struct file __rcu * fd_array[NR_OPEN_DEFAULT];
```

예를 들어, 3번 index라면 fd_array[3]으로 이동하고

```
struct file {
    union {
        struct llist_node   fu_llist;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path     f_path;
    struct inode        *f_inode;   /* cached valu
    const struct file_operations    *f_op;
```

File구조체에서 f_path를 따라

```
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
};
```

Dentry 포인터로 dentry 정보에 접근한다.

```
struct dentry {
    /* RCU lookup touched fields */
    unsigned int d_flags;       /* protected by d_lock */
    seqcount_t d_seq;       /* per dentry seqlock */
    struct hlist_bl_node d_hash;    /* lookup hash list */
    struct dentry *d_parent;    /* parent directory */
    struct qstr d_name;
    struct inode *d_inode;      /* Where the name belongs to - NULL
                     * negative */
    unsigned char d_iname[DNAME_INLINE_LEN];    /* small names */

    /* Ref lookup also touches following */
    struct lockref d_lockref;   /* per-dentry lock and refcount */
    const struct dentry_operations *d_op;
    struct super_block *d_sb;   /* The root of the dentry tree */
    unsigned long d_time;       /* used by d_revalidate */
    void *d_fsdata;         /* fs-specific data */

    struct list_head d_lru;     /* LRU list */
    struct list_head d_child;   /* child of parent list */
    struct list_head d_subdirs; /* our children */
    /*
```

Dentry에서 *d_inode로 접근하고

```
struct inode {
    umode_t         i_mode;
    unsigned short      i_opflags;
    kuid_t          i_uid;
    kgid_t          i_gid;
    unsigned int        i_flags;

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl    *i_acl;
    struct posix_acl    *i_default_acl;
#endif

    const struct inode_operations   *i_op;
    struct super_block  *i_sb;
    struct address_space    *i_mapping;

#ifdef CONFIG_SECURITY
    void            *i_security;
#endif

    /* Stat data, not accessed from path walking *
    unsigned long       i_ino;
```

i_sb인 super_block 으로 간다.

```
struct super_block {
    struct list_head    s_list;     /* Keep this first */
    dev_t           s_dev;      /* search index; _not_ kdev_t */
    unsigned char       s_blocksize_bits;
    unsigned long       s_blocksize;
    loff_t          s_maxbytes; /* Max file size */
    struct file_system_type *s_type;
    const struct super_operations   *s_op;
    const struct dquot_operations   *dq_op;
    const struct quotactl_ops   *s_qcop;
    const struct export_operations *s_export_op;
    unsigned long       s_flags;
    unsigned long       s_iflags;   /* internal SB_I_* flags */
    unsigned long       s_magic;
    struct dentry       *s_root;
    struct rw_semaphore s_umount;
    int         s_count;
    atomic_t        s_active;
```

Super_block 구조체에서 실행 시킨 파일의 정보를 저장한다

위와 같은 방식으로 실행 시킨 파일 포맷부터 시작해서 super_block에 정보를 넣고 inode를 채워서 리턴하여 file 정보를 얻어 온다. 이후에는 파일의 elf 포맷의 헤더를 읽고 pglist_data에 접근해서 필요한 물리 메모리양을 구하고 zone 자료구조로 가상 메모리와 맵핑을 한다. 그리고 demand on paging을 하여 cow로 디스크 블록을 물리 메모리에 올린다. 위와 같은 방식을 진행하면 내가 실행 시킨 파일이 돌아간다.

63.

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

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

    struct stat buf;
    char ch;
    stat(argv[1], &buf);
    if(S_ISDIR(buf.st_mode)) // 디렉토리
        ch = 'd';
    if(S_ISREG(buf.st_mode)) // 레귤러 일반 파일
        ch = '-';
    if(S_ISFIFO(buf.st_mode))  // 파이프
        ch = 'p';
    if(S_ISLNK(buf.st_mode))  // 링크 바로가기
        ch = 'l';
    if(S_ISSOCK(buf.st_mode)) // 소켓
        ch = 's';
    if(S_ISCHR(buf.st_mode)) //캐릭터 디바이스
        ch = 'c';
    if(S_ISBLK(buf.st_mode)) // 블락 디바이스
        ch = 'b';

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

    return 0;
}
```

66.
home에서 mkdir kernel 디렉토리 만들고 cd kernel 후에

1) wget https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/linux-4.4.tar.gz로 웹에서 해당 파일을 get

2) tar -zxvf linux-4.4.tar.gz 압출 풀기

3)cd linux-4.4로 옮겨서 설치파일 30 확인.

4) ctag랑 cscope 설치

5) ./vimrc와 mkcscope.sh 파일 코드 작성

6) ctags -R 후에 chmod 755 ~/mkcscope.sh로 권한 설정

7) vi -t task_struct한 후에 144번으로 이동

1~ 7과정을 하면 struct task_strcut {} 구조체를 볼 수 있다.

67.
TASK A와 TASK B가 있으면, 두 개의 kernel stack이 할당된다. 그리고 RQ에 A,B TASK가 들어간다. 일반적으로는 CFS 스케줄링 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 스케줄링 시간이 끝나 완전히 프로세스가 종료되면 expired상태가 되며 만약 실행해야 할 일이 남아 있다면, WQ로 빠져 다음 스케줄링 시간 까지 대기한다. RQ에서 스케줄링을 기다리던 B는 A가 끝나면, A와 마찬가지로 kernel stack에 자신의 정보를 저장하고 cpu에 올라가 active 상태가 된다. 위와 같은 방식을 아주 빠르게 진행한다면, Multi-tasking한다.

68.
ls -l /dev/mydevicefile

69.
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.

70.
프로세스는 task_struct로 되어있다. 여기서 task_struct의 디스크에서 용량이 100G라고 한다면, 이 파일을 실행하기 위해서는 물리 메모리가 엄청나게 커야 한다.하지만 현 시스템에서는 100G dram을 사용하기에는 현실적으로 어렵다. 그렇기 때문에, 물리 메모리의 한계를 극복하기 위해서 도입된 녀석이 'VM'이다. 평균 64 bit cpu의 경우 가상 메모리가 16EB까지 커버가 가능하다. 아무리 큰 용량을 가진 파일이라 할 지라도 가상 메모리 시스템으로 인해 모두 커버할 수 있게 되었다.

72.

코드

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

int main(void){

    DIR *dp;
    int i = 0;
    struct dirent *p;
    dp = opendir(".");
    while(p = readdir(dp)){

        if(p->d_name[0] == '.')
            continue;
        printf("%-16s", p->d_name);
        if( (i+1)%5 == 0)
            printf("\n");
        i++;
    }
    printf("\n");
    closedir(dp);

    return 0;
}
```

결과

```
temp2.txt       chat_serv       10_clone.c      10_fork_pt.c    tar.c
execve9.c       a.out           execve6.c       chat_clnt.c     7_echo_client.c
7_op_server.c   1_8.c           9_mpecho_clnt.c mycp            2_1.c
semlib.c        op_server       m_serv          9_gethostbyaddr.ctemp1.txt
5_queue.c       6_kill.c        7_inet_ntoa.c   data3.txt       fork2.c
wait4.c         res.tar         file_io2.c      3_2.c           3_1.c
4_goto.c        8_gethostbyname.cfile_io4.c      4_game.c        file_io3.c
6_test.c        fork3.c         4_setjmp.c      file_io11.c     chat_clnt
serv            m_clnt          kill            7_op_client.c   4_signal.c
mbr.txt         test1           fork1.c         mycp.c          6_basic_client.c
file_io5.c      sem.h.gch       execve.c        file_client     4_signal6.c
2_2_2.c         b.txt           6_convert_endian.cnet_game        data2.txt
execve4.c       6_basic_server.c369_game        chat_serv.c     wait3.c
9_mpecho.serv.c data1.txt       dup.c           6_sigaction1.c  2_2.c
1_6.c           file_io6.c      file_io10.c     web             file_io8.c
1_1.c           2_3.c           1_7.c           wait2.c         chat
1_2.c           file_io7.c      fork4.c         bbb.txt         clnt
6_inet_addr.c   a.txt           fifo.c          receive.txt     execve7.c
sem             tar_free.c      temp.txt        8_file_server.c 2_4.c
3_3.c           data.txt        10_thread.c     4_signal5.c     wait.c
file_io9.c      6_socket_fd.c   8_file_client.c network.h       newpgm.c
fork_exe.c      4_setjmp2.c     6_read_client.c sem.h           1_5.c
7_inet_aton.c   newpgm          10_pthread.c    test            file_server
tmep2.txt       op_client       1_3.c           7_echo_server.c 6_thread.c
c.txt           myfifo          fork9.c         execve5.c       5_adv_tech4.c
myfifo2.c       1_4.c           3_4.c
```

73. set follow-fork-mode child

74.
copy on write 의 약자로 프로세스가 실행 될 때, 가상 메모리에서 물리메모리로 페이징된다. 페이징 과정 중 demand on paging을 호출하면 디스크 블록이 페이지 프레임에 복사되고 쓰여지는 방식을 cow architecture라고 한다.

75.
 blocking 연산은 입력이 들어오기 까지 무한히 대기하고 non-blocking 연산은 콜센터에서 대기자가 많으면 나중에 연락을 주는 방식처럼 일단은 다 입력을 받고 순서대로 처리하는 연산 방식이다.

79.
goto의 경우 함수끼리의 점프가 불가능하다. 이것을 해결 해주기 위해 setjmp,longjmp가 도입되었다.

```c
#include<fcntl.h>
#include<stdlib.h>
#include<setjmp.h>
#include<stdio.h>


jmp_buf env1, env2;

void test(int ret, void (*p)(void)){

    if(ret ==0)
        longjmp(env1, 1);    // 2번째 인자는 리턴값
    p();
}

void test2(void){
    longjmp(env2, 1);
}

int main(void){
    int ret1, ret2;

    if( (ret1 = setjmp(env1)) == 0 ){  // setjmp = goto: , env = lable:
        printf("this1\n");              // setjmp초기 리턴값 0
        test(ret1, test2);
    }

    else if(ret1 > 0){
        printf("error1\n");

        if( (ret2 = setjmp(env2)) == 0){
            printf("this2\n");
            test(ret1, test2);
        }
        else if(ret2 > 0)
            printf("error2\n");
    }

    return 0;
}
```

80.
리눅스에서 fd는 파일 인덱스의 위치정보이다.

81.
buf.st_mode를 통해서 현재 이 파일이 디렉토리, 소켓 등과 같은 현재 파일의 상태를알려주는 상태 변수이다.

82. 프로세스를 물리 메모리에 다 직접 맵핑하기에는 너무 자원이 부족하다. 그래서 가상 메모리라는 시스템을 만들어 mm으로 프로세스마다 가상 메모리를 하나씩 넣어 프로세스의 용량이 크더라도 얼마든지 가상메모리로 커버하고 vm_area_struct로 vm area를 관리한다.
fd_array의 경우 open()의 인덱스에 해당함. 커널은 별도의 정보를 제공하지 않고 이 인덱스 정보만을 가지고 유저가 파일을 제어할 수 있게 해주는 역할을 한다.
signal의 경우는 비동기적 사건을 처리하게 해준다. 즉, 특정한 사건이 발생했을 때 프로세스보다 먼저 처리해야 할 일들을 처리하고 해당 사건이 끝나면 다시 프로세스를 동작시키기 위해서 사용한다.

87. cpu 가상화, 메모리 가상화, I/O 가상화

88.Server

```
#include"network.h"
#include"time.h"

extern jmp_buf env;

extern int clnt_cnt;

extern int ban_cnt;

extern int clnt_socks[MAX_CLNT];

extern net_data n_data[MAX_CLNT];

extern int *ban_ip[MAX_CLNT];

extern pthread_mutex_t mtx;
```

```
int main(int argc, char **argv){

	double time;
	int serv_sock, clnt_sock;
	si serv_addr, clnt_addr;
	ty start, end;

	socklen_t addr_size;
	pthread_t t_id;

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

	pthread_mutex_init(&mtx, NULL);

	serv_sock = socket(PF_INET, SOCK_STREAM, 0);

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

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

	if(bind(serv_sock , (sp)&serv_addr, sizeof(serv_addr)) == -1)
		err_handler("bind() error");
	if(listen(serv_sock, 10) == -1)
		err_handler("listen() error");
```

```
	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(;;){
		gettimeofday(&start,NULL);
		addr_size = sizeof(clnt_addr);
		clnt_sock = accept(serv_sock, (sp)&clnt_addr, &addr_size);


		pthread_mutex_lock(&mtx);

		n_data[clnt_cnt].clnt_sock = clnt_sock;
		n_data[clnt_cnt].n_clnt_addr = clnt_addr;

		pthread_mutex_unlock(&mtx);

		pthread_create(&t_id, NULL, clnt_handler
				,(net_data *)&n_data[clnt_cnt]);

		pthread_detach(t_id);

		printf("Connected Client IP: %s₩n", inet_ntoa(clnt_addr.sin_addr));
		clnt_cnt++;
```

```
		gettimeofday(&end,NULL);
		time = get_runtime(start,end);
		if( (int)time % 60 == 0){
			block_ip_init();
		}
	}

	close(serv_sock);

	return 0;
}
```

88. Client

```
#include"network.h"

char name[NAME_SIZE] = "[DEFAULT]";
char msg[BUF_SIZE] = {0};

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

void *send_msg(void *arg){

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

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

if(!strcmp(msg, "q₩n") || !strcmp(msg, "Q₩n")){
close(sock);
exit(0);
}
sprintf(name_msg, "%s %s",name, msg);
write(sock, name_msg, strlen(name_msg));

}
return NULL;
}
```

```
void *recv_msg(void *arg){

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

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

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

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

return NULL;
}
```

88. Client

```
int main(int argc, char **argv){

int sock;
si serv_addr;
pthread_t snd_thread, rcv_thread;
void *thread_ret;
if(argc != 4){
printf("Usage: %s <IP> <port> <name>₩n", argv[0]);
exit(1);
}

sprintf(name, "[%s]", argv[3]);
sock = socket(PF_INET, SOCK_STREAM, 0);

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

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

if(connect(sock, (sp)&serv_addr, sizeof(serv_addr)) == -1)
err_handler("connect() error!");

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

pthread_join(snd_thread, &thread_ret);
pthread_join(rcv_thread, &thread_ret);
close(sock);
return 0;
}
```

91. struct serv

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

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

#define BUF_SIZE            32

#endif
```

```c
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");
```

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

91. struct clnt

```
void read_proc(int sock, d *buf)
{
    for(;;)
    {
        int len = read(sock, buf, BUF_SIZE);

        if(!len)
            return;

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

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

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

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

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

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

결과

```
msg from serv: 3, 7.700000
^CExited!
```

```
struct.data = 3, struct.fdata = 7.700000
load_ratio = 2326767676.767677
```

93.
critcal section은 여러 스레드가 서로 cpu의 자원을 점유하면서 data영역이 꼬일 수 있는 구간을 말한다.

(보류)94. 유저에서 fork()를 수행할 때 버러지는 일들 전부를 소스 코드 차원에서 해석하라.

95.
arch는 cpu 모델에 따라서 바뀌어야 할 코드들이 들어 있다. (context switching code)

96.

- arm (하위 호환을 하지 않음 – 종류가 많은 이유)
- Exynos (삼성), omap (TI), zynq (자이링스)
- s3c24xx (삼성)
- keystone (TI DSP), 2 (레이더용)
- ipc18xx, ipc 32xx (NXP) (차량용)
- bcm (라즈베리파이)
- davinci (TI) (블랙박스, 스마트 tv, CCTV)
- stm32 (차량용)
- tegra (NVIDIA)

97. TI DSP 관련 내용

98.
do_irq함수로 하드웨어 인터럽트가 처리된다.
인터럽트 번호가 __vector_start 테이블에서 do_irq의 인자로 vec에 들어오고 vec_to_irq 테이블에서 해당 인터럽트가 호출되면서 하드웨어 인터럽트가 실행됨.

```
void do_irq(int vec, struct pt_regs *fp)
{
    int irq = vec_to_irq(vec);
    if (irq == -1)
        return;
    asm_do_IRQ(irq, fp);
}
```

99.
arm TTBR0, TTBR 1을 사용함.

100.

2개월이라는 시간이 짧을 수도 있지만, 그 사이에 정말 공부하는 자세와 실력이 부쩍 늘었다고 생각합니다. 첫 한 달은 늦은 시각까지 계속 C언어와 자료구조를 잘하고 싶어서 고민했고 고민한 만큼 실력이 좋아졌습니다.

두 달 째에는 시스템 프로그래밍이 다소 생소했지만 그냥 재미있었습니다. 그리고 리눅스 커널 내부구조를 시작하면서는 좀 어려웠지만, 선생님께서 '이건 미국 대학원이나 가야 하는 것'이라는 말씀을 해주셔서 정말 잘하고 싶어 새벽까지 공부하고 A4용지에 책의 내용을 옮겨가며 읽고 다시 읽고를 반복을 하다보니 지금은 정말 대략적으로 커널은 이렇게 동작하는 것이라는 것도 깨우칠 수 있었다고 생각합니다.

앞으로는 이것보다 더 어려운 과목들과 프로젝트라는 끝판왕이 기다리고 있지만, 지금보다 더 열심히 열정을 불사질러 완성도 있는 작품을 만들고 이 분야에서 전문성을 높혀 제 값어치를 엄청나게 올리고 싶은 마음입니다. 이 마음이 흔들리지 않고 프로젝트를 완성하는 그 순간까지 지치지 않고 열심히 하고 싶습니다. 감사합니다.