Operating System 실습

[4주차]

한양대학교 HANYANG UNIVERSITY

Data-Centric Systems Lab.

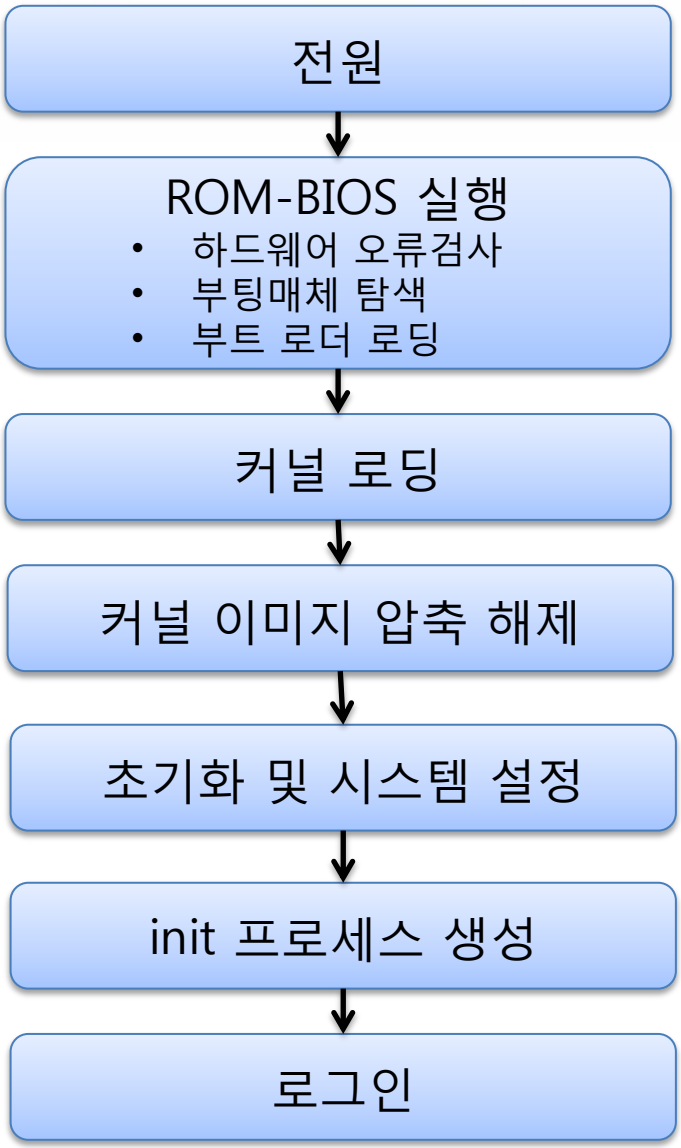
Contents

- Booting
- Interrupt
- System call
- Adding System Call

Booting

- 운영체제의 최소 부분을 주 메모리로 로딩해서 프로세서가 실행하도록 하는 것
  - 프로세서 초기화
  - 메모리 점검 및 초기화
  - 각종 하드웨어 점검 및 초기화
  - 커널 로딩 / 커널 자료 구조 등록 및 초기화
  - 사용 환경 조성

Booting (Cont.)



리눅스 시스템 부팅 과정

Booting (Cont.)

- BIOS (Basic Input / Output System)
  - 메모리의 특정 주소로 자동 로드 되어 실행되는 프로그램
  - 하드웨어와 소프트웨어 사이의 연결과 번역 기능을 담당하는 인터페이스
  - 리얼 모드 주소를 사용
    * 각종 장치 체크

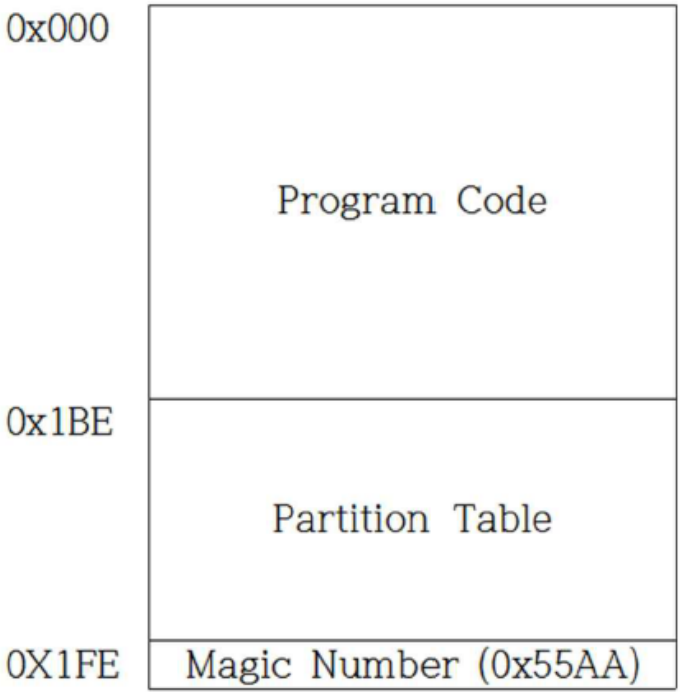
      CPU, 그래픽카드, RAM 등 장치 체크

      POST (Power On Self Test)
    * 하드웨어 장치를 초기화
    * 부트 섹터(MBR)에 접근

      BIOS 설정에 따라 : Floppy Disk, HDD, CD-ROM
    * 첫 번째 섹터 내용을 물리 주소 0x00007c00부터 시작하여 램으로 복사 한 후 코드 실행

Booting (Cont.)

- MBR(Master Boot Record)
  - 디바이스의 첫 섹터
    * 부트 프로그램 코드와 파티션 테이블 및 매직 넘버로 구성
    * 매직 넘버 : 0x55AA
      실제로 이 영역이 MBR이 맞는지 확인하는 값
    * 크기 : 512 bytes



Booting (Cont.)

- Boot Loader
  - 운영체제의 커널 이미지를 RAM으로 읽어 들이기 위해 BIOS에서 호출하는 프로그램
    * 플로피 디스크
      - 디스크의 첫 번째 섹터에 들어 있는 코드를 RAM으로 로딩해서 실행
    * 하드 디스크
      - 파티션 테이블을 통해 파티션의 첫 번째 섹터를 읽어 들이는 작은 프로그램을 담는다
      - LILO나 GRUB 같은 부트 로더로 교체하여 실행

✓ 커널 이미지를 메모리로 로딩

✓ setup() 함수 코드로 점프

Booting (Cont.)

- setup()
  - 하드웨어 장치 초기화
  - 임시 IDT(Interrupt Descriptor Table)와 임시 GDT(Global Descriptor Table)를 설정
  - 리얼 모드에서 보호 모드로 전환
  - startup_32() 어셈블리어 함수로 점프

Booting (Cont.)

- startup_32()
  - /linux/arch/x86/boot/compressed/head.S
    * decompress_kernel() 함수를 호출하여 커널 이미지 압축 해제
  - /linux/arch/x86/kernel/head.S
    * init process의 실행 환경 구성
    * start_kernel() 함수로 점프

Booting (Cont.)

- start_kernel()
  - 커널 구성 요소 초기화
    * trap / Interrupt 초기화
    * scheduler 초기화
    * softirq 초기화
    * kernel module 사용을 위한 초기화
    * fork 에 관한 초기화
    * kernel Cache 및 Buffer 초기화
    * /proc 디렉토리 초기화
    * init kernel thread 시작 → init process 실행

Interrupt

- 주변 장치나 CPU가 자신에게 발생한 사건을 리눅스 커널에게 알리는 메커니즘
  - 트랩
    * 현재 수행 중인 태스크와 관련된 동기적 사건
    * 에외처리 exception라고도 불림
    * divide_by_zero, 페이지 폴트, 비정상 주소 접근, 시스템 콜 등
  - 외부 인터럽트
    * 현재 수행중인 태스크와 관련없는 주변장치에서 발생된 비동기적 하드웨어 사건
    * 보드/마우스, 디스크, 네트워크, 클럭 등

Interrupt (Cont.)

- CPU가 발생시키는 예외
  - 0 : Divide Error
  - 1 : Debug
  - 2 : Not Used
    - 마스크가 불가능한 인터럽트(nmi 핀 사용)용으로 예약
  - 3 : Breakpoint
    - Int3
  - 4 : Overflow
  - 7 : Device Not Available
  - 13 : General Protection
  - 14 : Page Fault
  - 19 : SIMD Floating Point Exception
  - 20~31 : Reserved by Intel

Interrupt (Cont.)

- IDT(Interrupt Descriptor Table)
  - CPU는 인터럽트 발생시 PC값을 미리 정해진 특정 번지로 변경
  - 각각의 특정 번지에는 해당 인터럽트를 처리하는 인터럽트 핸들러를 가리킴

- Interrupt Handler
  - 인터럽트가 발생했을 때, 그 작업을 처리하는 함수
  - Interrupt Service Routine (ISR) 이라고도 함

Interrupt (Cont.)

- 리눅스는 외부 인터럽트와 트랩을 동일한 방식으로 처리
- idt_table
  - 외부 인터럽트와 트랩을 처리하기 위한 루틴
  - 0~31까지 32개의 엔트리를 CPU 트랩 핸들러 할당, 그 외 엔트리는 외부 인터럽트 핸들러를 위해 사용
- do_IRQ() 함수
  - idt_table에서 128번을 제외한 32~255번까지 do_IRQ() 함수가 등록
  - 외부 인터럽트 번호로 irq_desc 테이블을 인덱싱하여 해당하는 irq_desc_t 를 찾음
- irq_desc 테이블
  - 외부 인터럽트를 발생할 수 있는 라인은 한정적이기 때문에 외부 인터럽트를 위한 번호를 별도로 관리
- irq_desc_t
  - 하나의 인터럽트를 공유할 수 있도록 action 리스트를 유지

Interrupt (Cont.)

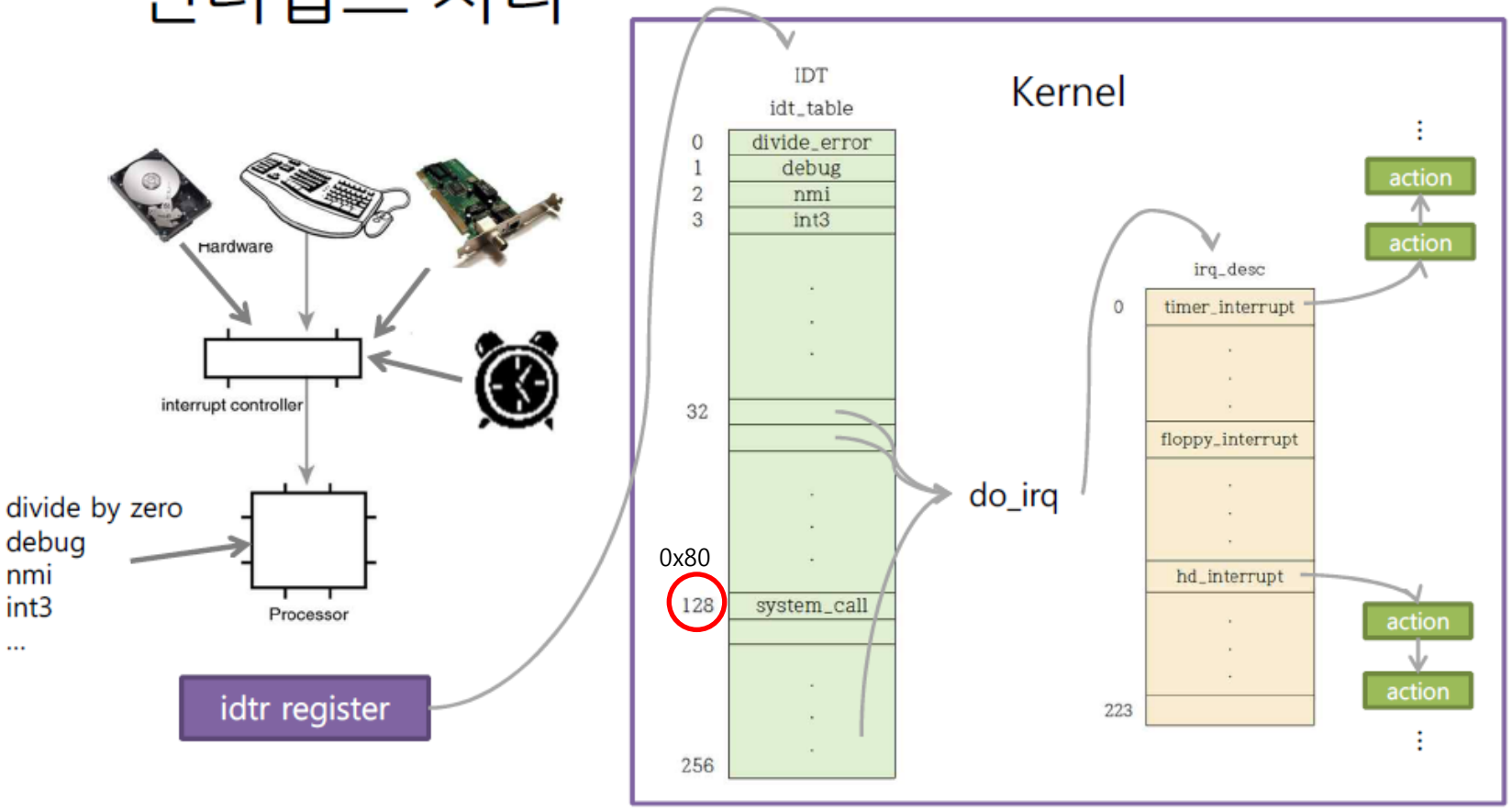
- PIC (Programmable Interrupt Controller)
  - 외부 인터럽트 발생시키는 주변 장치는 하드웨어적으로 PIC 칩 각 핀에 연결
  - PIC은 CPU의 한 핀과 연결

Interrupt (Cont.)

- 인터럽트 처리 단계
  - /linux/arch/x86/kernel/entry_32.S
  - 수행 모드 변경
    * 사용자 수준(User Level) 에서 커널 수준(Kernel Level)
  - 현재 수행 중이던 태스크의 문맥 저장
    * SAVE_ALL
  - IDT를 통해 do_irq() 함수 실행
    * 인터럽트 처리 루틴 호출
  - 저장했던 문맥 복원
    * ret_from_intr (RESTORE_ALL)

Interrupt (Cont.)

- 인터럽트 처리

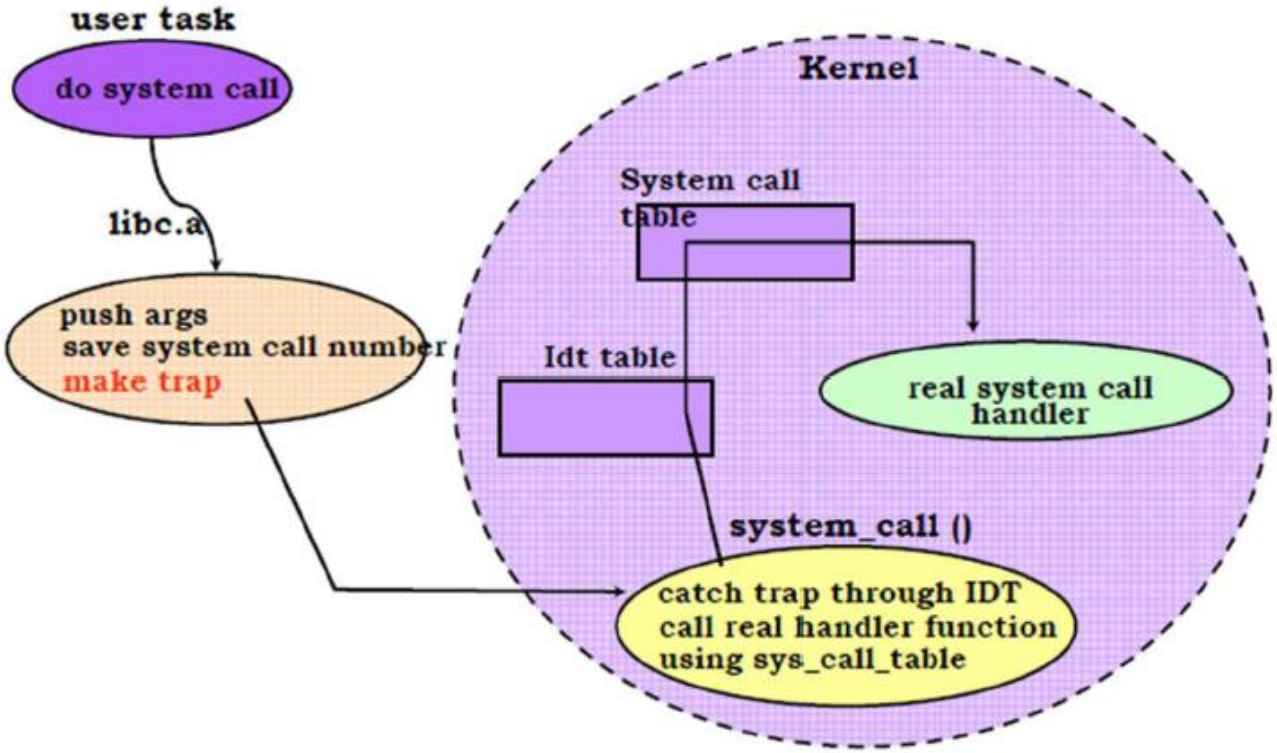


System Call

- 사용자 수준 응용 프로그램에게 커널이 자신의 서비스를 제공하는 인터페이스
- 사용자가 운영체제의 기능이나 모듈을 활용하기 위해서는 반드시 시스템 콜을 사용해야 함
- 커널로의 진입점(entry point)이라고 볼 수 있음
  - 예
    * sys_fork() : 새로운 태스크 생성
    * sys_read() : 파일의 내용을 읽음
    * sys_nice() : 현재 실행 중인 태스크의 실행 우선순위 제어
- 리눅스 커널은 각 시스템 콜을 함수(System Call Handler)로 구현해 놓고 각 시스템 콜이 요청되었을 때 대응되는 함수를 호출하여 서비스를 제공함

System Call (Cont.)

- 시스템 콜 처리



System Call (Cont.)

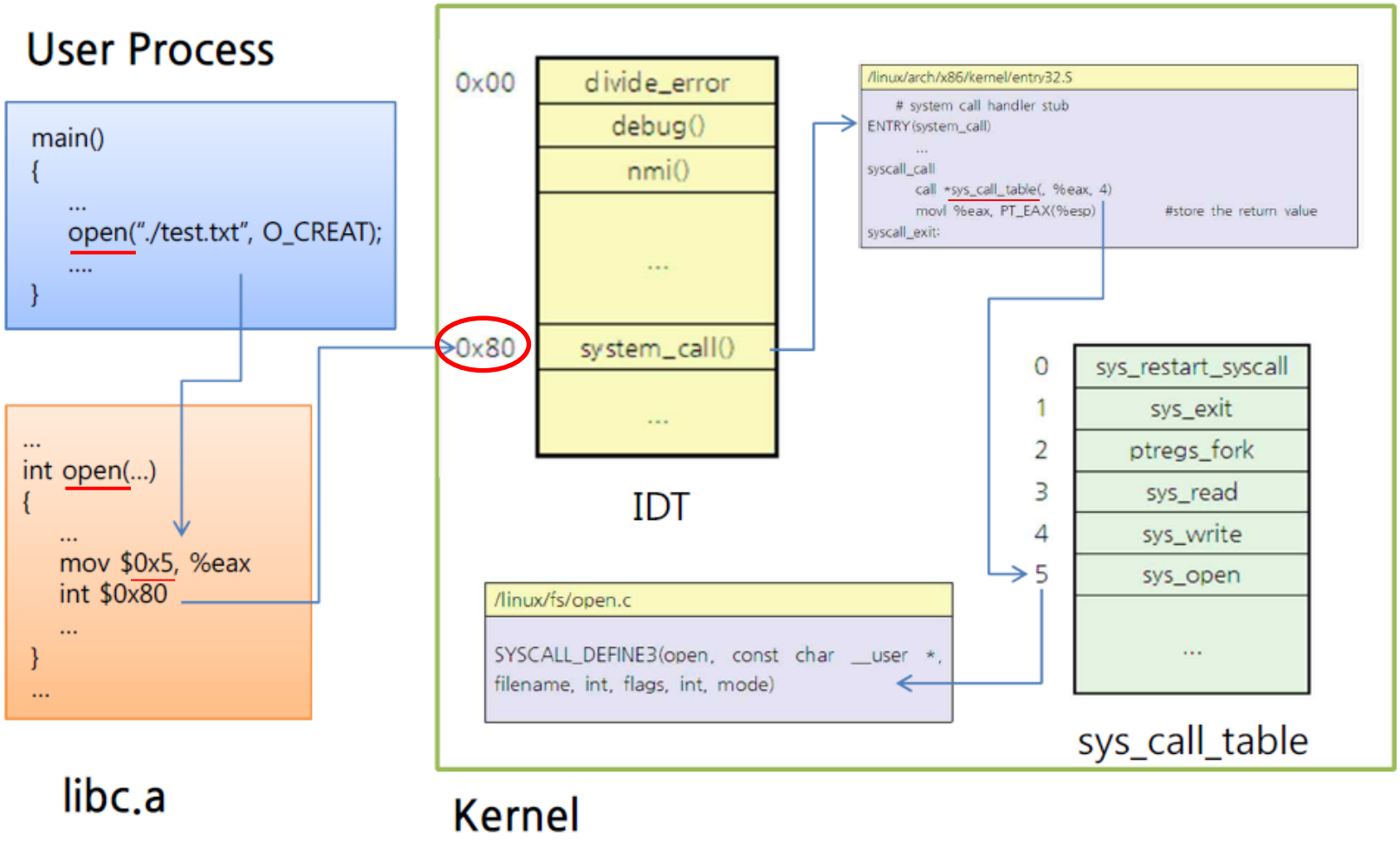
- 시스템콜 처리

  - Save System Call Number
    * eax 레지스터에 시스템 콜 번호 저장

  - Make trap
    * Using int 0x80
    * IDT 를 통해 system_call() 함수 실행
    * sys_call_table 을 통해 핸들러 함수 실행

  - Return System Call Handler
    * ret_from_sys_call()

System Call (Cont.)



Adding System Call

- Step 1 : Register the new sys_func() in sys_call_table[]
- Step 2 : Implement a new function in the kernel
- Step 3 : Compile and Reboot

Adding System Call

- /linux/arch/x86/syscalls/syscall_32.tbl

```
350 341     i386    name_to_handle_at       sys_name_to_handle_at
351 342     i386    open_by_handle_at       sys_open_by_handle_at           compat_sys_open_by_handle_at
352 343     i386    clock_adjtime           sys_clock_adjtime               compat_sys_clock_adjtime
353 344     i386    syncfs                  sys_syncfs
354 345     i386    sendmmsg                sys_sendmmsg                    compat_sys_sendmmsg
355 346     i386    setns                   sys_setns
356 347     i386    process_vm_readv        sys_process_vm_readv            compat_sys_process_vm_readv
357 348     i386    process_vm_writev       sys_process_vm_writev           compat_sys_process_vm_writev
358 349     i386    kcmp                    sys_kcmp
359 350     i386    finit_module            sys_finit_module
360 351     i386    helloworld              sys_helloworld
```

Adding System Call

- /linux/include/linux/syscalls.h

```
871 asmlinkage long sys_process_vm_writev(pid_t pid,
872                                       const struct iovec __user *lvec,
873                                       unsigned long liovcnt,
874                                       const struct iovec __user *rvec,
875                                       unsigned long riovcnt,
876                                       unsigned long flags);
877
878 asmlinkage long sys_kcmp(pid_t pid1, pid_t pid2, int type,
879                          unsigned long idx1, unsigned long idx2);
880 asmlinkage long sys_finit_module(int fd, const char __user *uargs, int flags);
881 asmlinkage long sys_helloworld(void);
```

Adding System Call

- /linux/kernel/helloworld.c

```
1 #include<linux/kernel.h>
2 #include<linux/linkage.h>
3 #include<linux/unistd.h>
4
5 asmlinkage long sys_helloworld(void)
6 {
7         printk("Hello World!\n");
8 }
9
~
```

sys_helloworld()는 커널 수준에서 수행되는 함수이므로 사용자 수준에서 수행되는 표준 C 라이브러리를 사용할 수 없다.

Adding System Call

- /linux/kernel/Makefile

```
#
# Makefile for the linux kernel.
#

obj-y     = fork.o exec_domain.o panic.o printk.o \
            cpu.o exit.o itimer.o time.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o task_work.o \
            rcupdate.o extable.o params.o posix-timers.o \
            kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
            hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
            notifier.o ksysfs.o cred.o \
            async.o range.o groups.o lglock.o smpboot.o helloworld.o

ifdef CONFIG_FUNCTION_TRACER
# Do not trace debug files and internal ftrace files
```

- Compile a kernel

Adding System Call

- System call test source (User Program)

```
1 #include<stdio.h>
2 #include<linux/unistd.h>
3
4 void main(int argc, char *argv[])
5 {
6         syscall(351);
7 }
8
~
```

- Compile & execute

- $dmesg

```
[   15.182891] init: plymouth-stop pre-start process (1153) terminated with stat
us 1
[   26.357511] ISO 9660 Extensions: Microsoft Joliet Level 3
[   26.372541] ISO 9660 Extensions: RRIP_1991A
[   88.386581] Hello World!
```

Adding System Call

- 시스템 콜
  - 커널에게 어떤 서비스를 요청할 때 사용
    - 예 : fork(), read()…
  - 커널의 정보를 얻기 위해 사용
    - 예 : getpid(), stat()…

```
1  #include <linux/kernel.h>
2  #include <linux/linkage.h>
3  #include <linux/unistd.h>
4  #include <linux/sched.h>
5
6  asmlinkage long sys_gettaskinfo(void)
7  {
8      printk("[*]PID : %d \n", current->pid);
9      printk("[*]TGID : %d \n", current->tgid);
10     printk("[*]PPID : %d \n", current->parent->pid);
11     printk("[*]STATE : %d \n", current->state);
12     printk("[*]PRIORITY : %d \n", current->prio);
13     printk("[*]POLICY : %d \n", current->policy);
14     printk("[*]Number of MAJOR FAULT : %d \n", current->maj_flt);
15     printk("[*]Number of MINOR FAULT : %d \n", current->min_flt);
16
17     return 0;
18 }
```

현재 실행중인 태스크의 정보를 출력한다. 이를 위해 현재 실행 중인 task_struct의 각 내용을 얻어 와서 출력한다. 다행히 리눅스 커널 내부에는 현재 실행 중인 태스크의 task_struct를 가리키는 변수가 존재하는데 이것이 current라는 포인터 변수이다. 앞 예제에서 수행 했던 것과 동일한 과정을 거쳐 시스템 콜을 추가해보자.

Adding System Call

- Test Source (User Program)

```c
#include <stdio.h>
#include <linux/unistd.h>

void main(int argc, char *argv[])
{
    syscall(351);//helloworld
    syscall(352);//gettaskinfo
}
```

Adding System Call

```c
#include <linux/kernel.h>
#include <linux/linkage.h>
#include <linux/unistd.h>
#include <asm/uaccess.h>

asmlinkage long sys_show_mult(int x, int y, int *res)
{
    int error, compute;
    int i;
    error = access_ok(VERIFY_WRITE, res, sizeof(*res));
    if(error < 0)
    {
        printk("error in cdang\n");
        printk("error is %d\n", error);
        return error;
    }

    compute = x * y;
    printk("compute is %d\n", compute);
    i = copy_to_user(res, &compute, sizeof(int));

    return 0;
}
```

곱셈한 결과를 사용자 수준 공간에 전달하기 위해 이 프로그램은 리눅스가 제공하는 copy_to_user()라는 매크로를 사용하였다. 이 매크로는 include/asm/uaccess.h에 정의되어 있다. 그리고 copy_to_user()를 사용하기 전에 res라는 사용자 공간에 쓰기가 가능한지 확인하기 위해 access_ok()라는 커널 내부 함수를 사용하였다.

Adding System Call

- Test Source (User Program)

```c
#include <stdio.h>
#include <linux/unistd.h>

int main(void)
{
    int mult_ret = 0;
    int x = 2, y = 5;
    int i;

    i = syscall(353, x, y, &mult_ret);
    printf("x is %d, y is %d, ret is %d \n", x, y , mult_ret);

    return 0;
}
```

HW #1

- 보고서는 다음과 같은 내용을 포함해야 합니다.
  - HW1-1 Kernel Compile 과정 스크린 샷
    * uname –a
  - HW1-2
    * System Call 추가 예제 3개 실행 결과 스크린 샷, 소스
    * System Call 동작 과정 설명
    * helloworld, gettaskinfo, show_mult