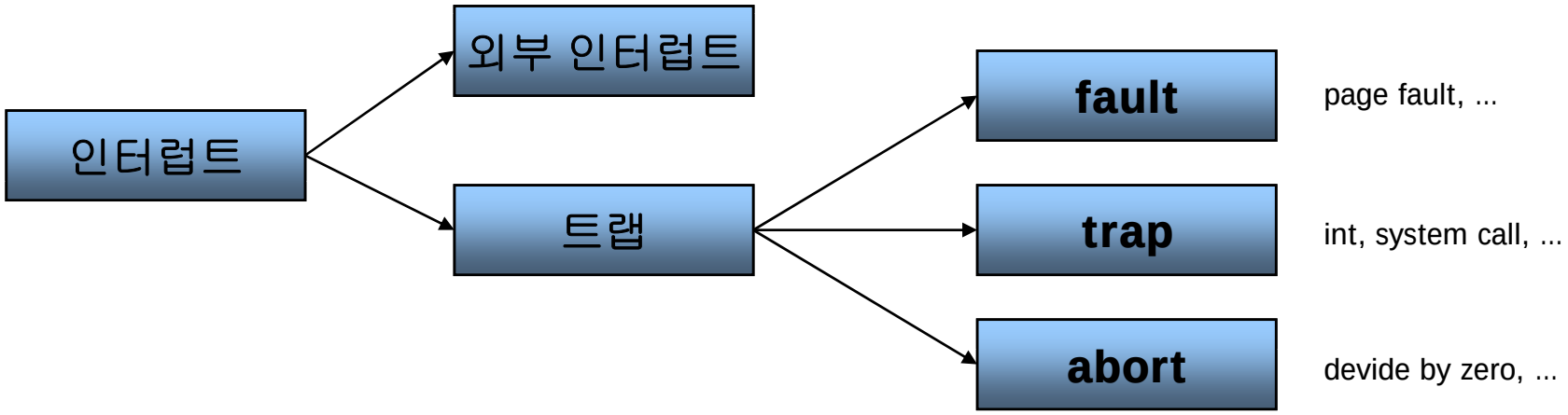
Interrupt, Module Programming

단국대학교
컴퓨터학과
2009
백승재
ibanez1383@dankook.ac.kr
http://embedded.dankook.ac.kr/~ibanez1383

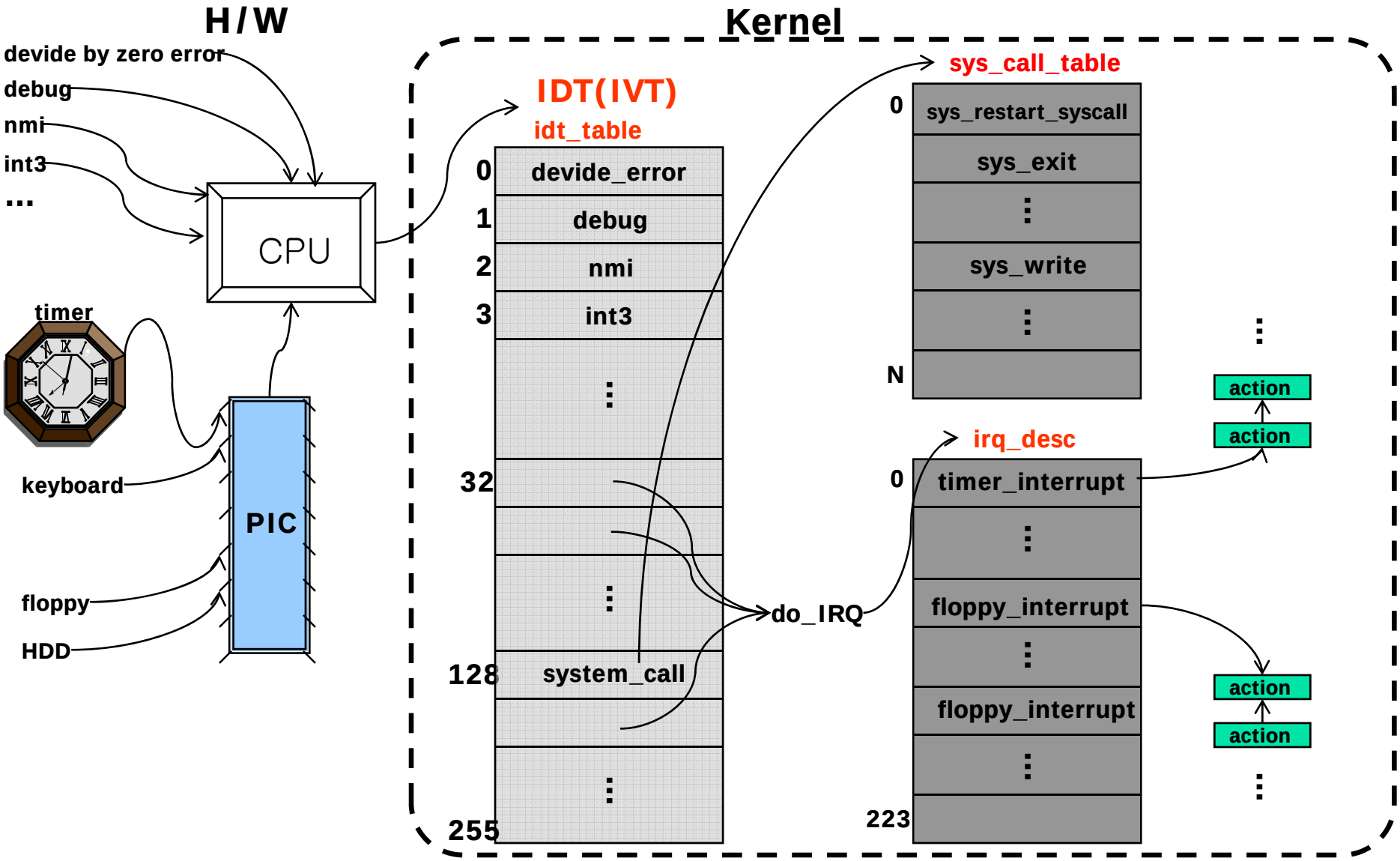
강의 목표

- Linux의 인터럽트 처리 과정 이해
- 시스템 호출 원리 파악
- 모듈 프로그래밍 기법 숙지

인터럽트의 분류


인터럽트
외부 인터럽트
트랩
fault
trap
abort
page fault, ...
int, system call, ...
devide by zero, ...

인터럽트와 트랩의 처리


H/W
devide by zero error
debug
nmi
int3
...
CPU
timer
keyboard
PIC
floppy
HDD
Kernel
IDT(IVT)
idt_table
0 devide_error
1 debug
2 nmi
3 int3
32
128 system_call
255
do_IRQ
sys_call_table
0 sys_restart_syscall
sys_exit
sys_write
N
irq_desc
0 timer_interrupt
floppy_interrupt
floppy_interrupt
223
action
action
action
action

irq_desc

- irq_desc
  - ✓ NR_IRQS(보통 224)개의 irq_desc_t 디스크립터의 배열
  - ✓ status : IRQ선의 상태 나타내는 flags

Flag name	Description
IRQ_INPROGRESS	A handler for the IRQ is being executed.
IRQ_DISABLED	The IRQ line has been deliberately disabled by a device driver.
IRQ_PENDING	An IRQ has occurred on the line; its occurrence has been acknowledged to the PIC, but it has not yet been serviced by the kernel.
IRQ_REPLAY	The IRQ line has been disabled but the previous IRQ occurrence has not yet been acknowledged to the PIC.
IRQ_AUTODETECT	The kernel uses the IRQ line while performing a hardware device probe.
IRQ_WAITING	The kernel uses the IRQ line while performing a hardware device probe; moreover, the corresponding interrupt has not been raised.
IRQ_LEVEL	Not used on the 80 x 86 architecture.
IRQ_MASKED	Not used.
IRQ_PER_CPU	Not used on the 80 x 86 architecture.

  - ✓ handler : IRQ선을 처리하는 PIC회로를 나타내는 hw_interrupt_type 디스크립터에 대한 ptr
  - ✓ action : IRQ가 발생할때 호출할 ISR
  - ✓ lock : IRQ디스크립터에 대한 spin lock

irqaction

- irqaction 디스크립터
  - ✓ handler : I/O device 용 ISR을 가리킴
  - ✓ flags : IRQ선과 I/O device간의 관계 설명
  - ✓ name : I/O device의 이름

Flag name	Description
SA_INTERRUPT	The handler must execute with interrupts disabled.
SA_SHIRQ	The device permits its IRQ line to be shared with other devices.
SA_SAMPLE_RANDOM	The device may be considered a source of events that occurs randomly; it can thus be used by the kernel random number generator. (Users can access this feature by taking random numbers from the */dev/random* and */dev/urandom* device files.)

  - ✓ dev_id : I/O device서 사용하는 private필드
  - ✓ next : irqaction 디스크립터 리스트의 다음 항목을 가리킴

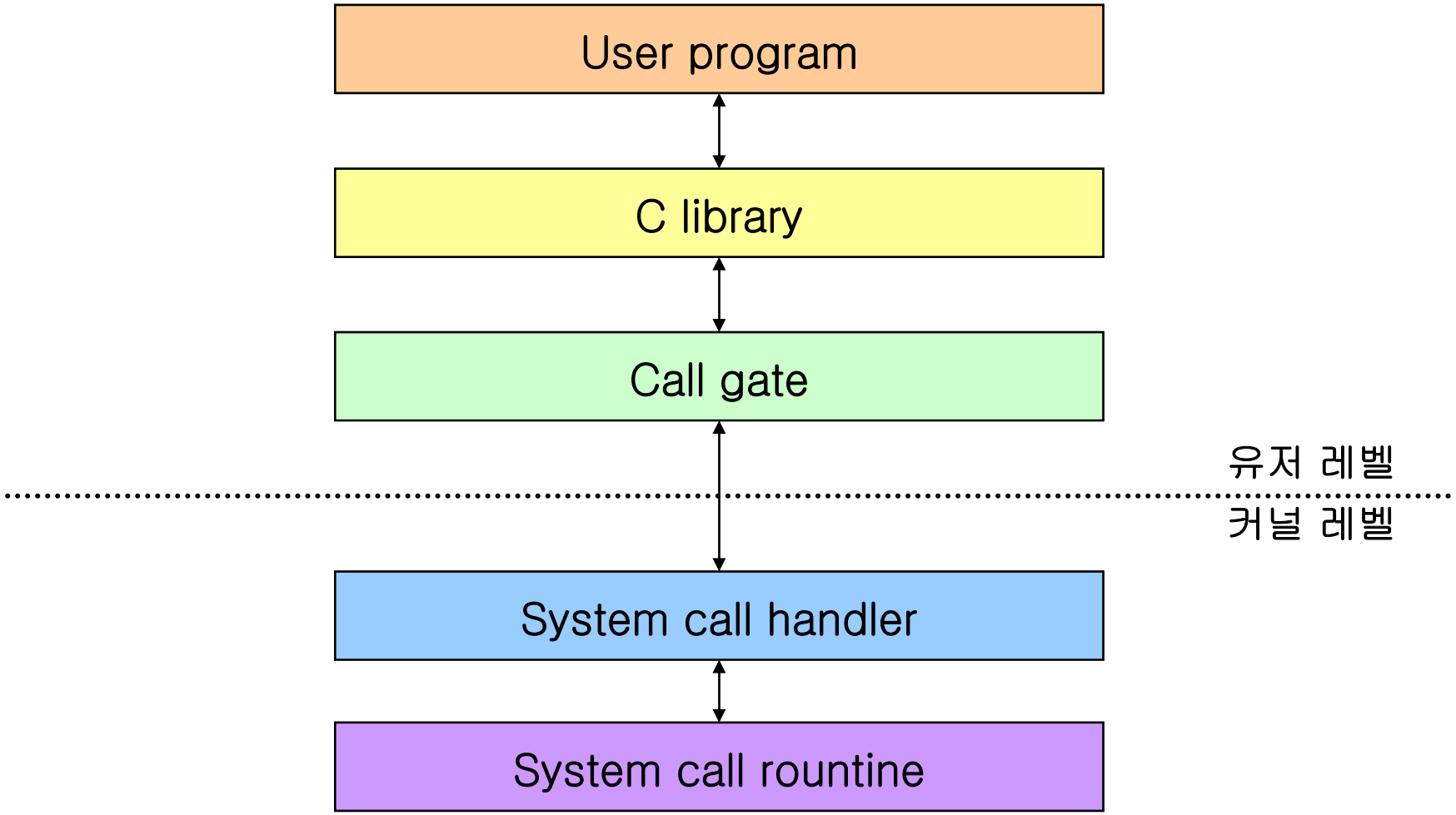
인터럽트 처리 과정

```
IRQn_interrupt:
    pushl $n-256
    jmp common_interrupt

common_interrupt:
    SAVE_ALL
    call do_IRQ
    jmp $ret_from_intr
```

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $__KERNEL_DS,%edx
movl %edx,%ds
movl %edx,%es
```

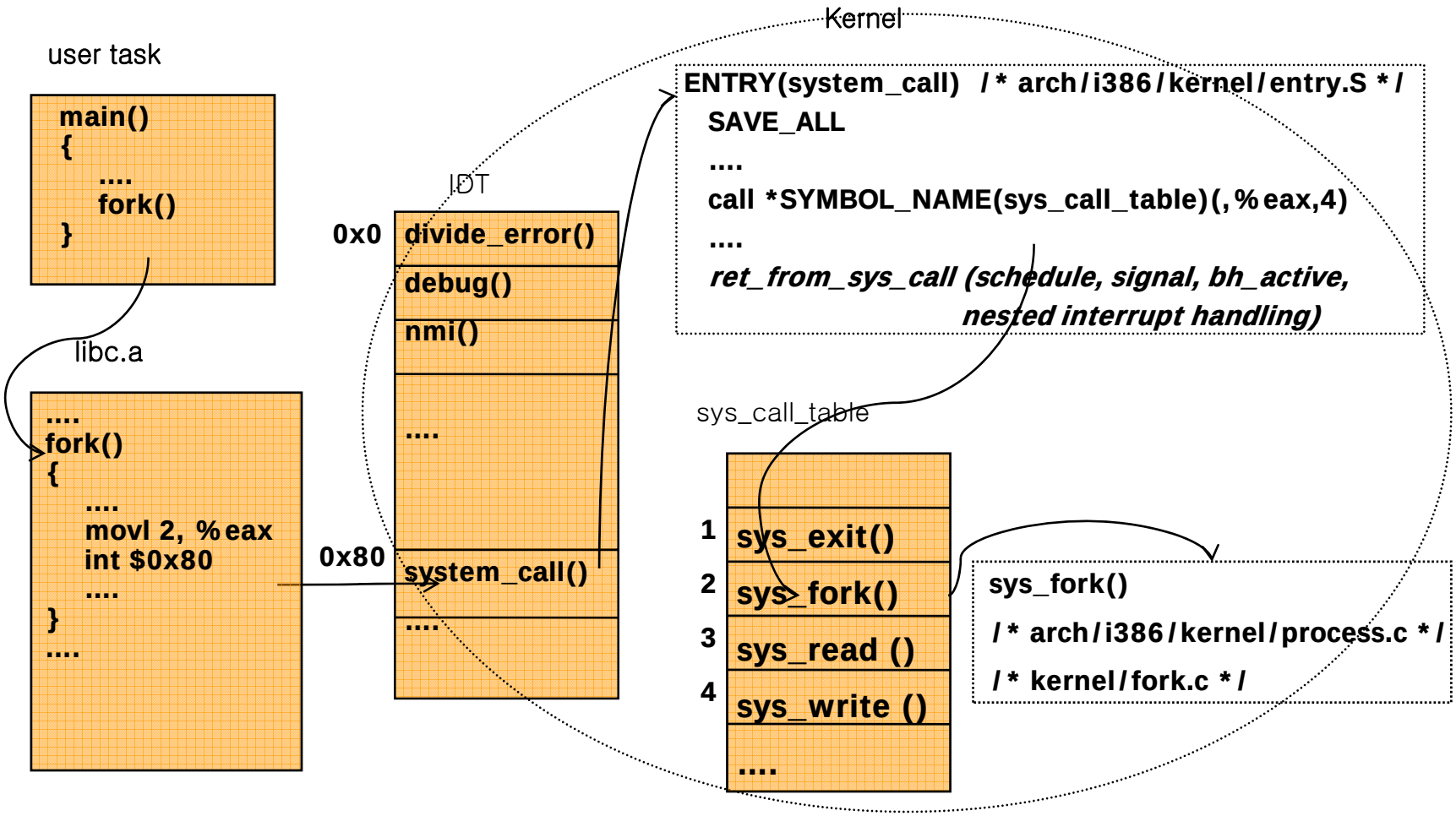
- do_IRQ()
  - ✓ 스택에는 return addr, SAVE_ALL로 저장한 reg값, 변환한 IRQ번호, INT발생 시 CU가 자동 저장한 reg
  - ✓ unsigned int do_IRQ(struct pt_regs regs)
  - ✓ pt_regs 처음 9필드는 SAVE_ALL로 저장한 reg값, 10번째는 orig_eax로 참조하는 필드=변환한 IRQ번호, 나머지는 CU가 자동 저장한 regls

시스템 호출 처리


User program
C library
Call gate
유저 레벨
커널 레벨
System call handler
System call rountine

시스템 호출 처리 과정

- 시스템 호출 처리 과정 예 : fork



시스템 호출 처리 관련

- system_call() → sys_system_call()
- asmlinkage
  - ✓ 함수의 argument를 stack을 통해 전달 받음
  - ✓ Asm내에서 C 함수를 호출할수 있도록 함
- 시스템 콜 번호
  - ✓ 각 시스템 콜에 시스템콜 번호 부여. 고유한 숫자
  - ✓ sys_call_table에 저장하고 있다
- 시스템 콜 핸들러
  - ✓ int $0x80 → 128번 vector의, 프로그래밍에 의한 예외발생
  - ✓ 순서
    - 시스템 콜 사용위해 커널에 인터럽트를 건다(eax에 syscall번호)
    - reg를 커널 모드 스택에 저장
    - 시스템콜 서비스 루틴 호출하여 처리
    - ret_from_sys_call()로 핸들러서 빠져 나옴
  - ✓ 적법한 매개변수, 퍼미션 인지 검사 필수
  - ✓ copy_to_user(), copy_from_user()사용

system_call 함수

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
1       SAVE_ALL
2       GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
3       jne tracesys
        cmpl $(NR_syscalls),%eax
4       jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
5       movl %eax,EAX(%esp)             # save the return
```

```
#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS),%edx; \
        movl %edx,%ds; \
        movl %edx,%es;
```

넘어온 매개 변수를 커널 모드 스택에 저장

```
badsys:
        movl $-ENOSYS,EAX(%esp)
        jmp ret_from_sys_call
```

1. 제어 유닛이 자동으로 저장한 eflags, cs, eip, ss, esp제외한 모든 reg를 스택에 저장
2. ebx reg에 current P의 디스크립터를 저장
3. current의 ptrace필드에 PT_TRACESYS플래그가 들어 있는지, 즉 디버거가 프로그램의 시스템콜 호출을 추적 중인지 검사 ➔ syscall_trace()를 처음, 마지막 두번 호출하게 됨
4. 올바른 syscall번호인지 검사. 잘못된 번호이면 바로 종료
5. dispatch table의 각 엔트리는 4바이트이므로, 시스템 콜 번호에 4를 곱한 후 + sys_call_table 시작주소를 더해서 ➔ 서비스 루틴의 ptr얻어와서 호출함 . 호출 종료되면 리턴값을 저장한 스택(사용자 모드에서의 eax)에 저장해놓고, syscall핸들러를 종료하는 ret_from_sys_call로 점프

복귀 과정

```
ENTRY(ret_from_sys_call)
        cli                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
```

```
tracesys_exit:
        call SYMBOL_NAME(syscall_trace)
        jmp ret_from_sys_call
```

```
tracesys:
        movl $-ENOSYS,EAX(%esp)
        call SYMBOL_NAME(syscall_trace)
        movl ORIG_EAX(%esp),%eax
        cmpl $(NR_syscalls),%eax
        jae tracesys_exit
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
```

```
signal_return:
        sti                             # we can get here from an interrupt handler
        testl $(VM_MASK),EFLAGS(%esp)
        movl %esp,%eax
        jne v86_signal_return
        xorl %edx,%edx
        call SYMBOL_NAME(do_signal)
        jmp restore_all
```

매개 변수와 주소 공간

- 매개 변수 확인
  - ✓ 매개 변수가 주소인 경우 검사 방법 두 가지
  - ✓ 선형 주소가 P 주소 공간에 속하는지,속하면 접근 권한이 있는지 검사
  - ✓ 선형 주소가 PAGE_OFFSET보다 낮은지 만 확인
  - ✓ access_ok()이용 system call에 전달한 주소 검사
- 프로세스 주소 공간 접근

Function	Action
get_user _ _get_user	Reads an integer value from user space (1, 2, or 4 bytes)
put_user _ _put_user	Writes an integer value to user space (1, 2, or 4 bytes)
copy_from_user _ _copy_from_user	Copies a block of arbitrary size from user space
copy_to_user _ _copy_to_user	Copies a block of arbitrary size to user space
strncpy_from_user _ _strncpy_from_user	Copies a null-terminated string from user space
strlen_user strnlen_user	Returns the length of a null-terminated string in user space
clear_user _ _clear_user	Fills a memory area in user space with zeros

새로운 시스템 호출 구현 (1/7)

■ 새로운 시스템 호출 구현

✓ 커널 수정 : 4 단계

1. 새로운 시스템 호출 번호 할당 (allocate syscall_number)
2. 새로운 시스템 호출 함수 sys_call_table[]에 등록
3. 새로운 시스템 호출 함수 커널에 구현
4. 커널 컴파일 및 리부팅

✓ 사용자 응용 작성 : 2단계

1. 새로운 시스템 호출을 사용하는 사용자 수준 응용 작성
2. 라이브러리로 사용자 응용 작성 (optional) : ar, ranlib

새로운 시스템 호출 구현 (2/7)

- 새로운 시스템 호출 구현 예 : newsyscall() 이라는 이름의 새로운 시스템 호출 구현

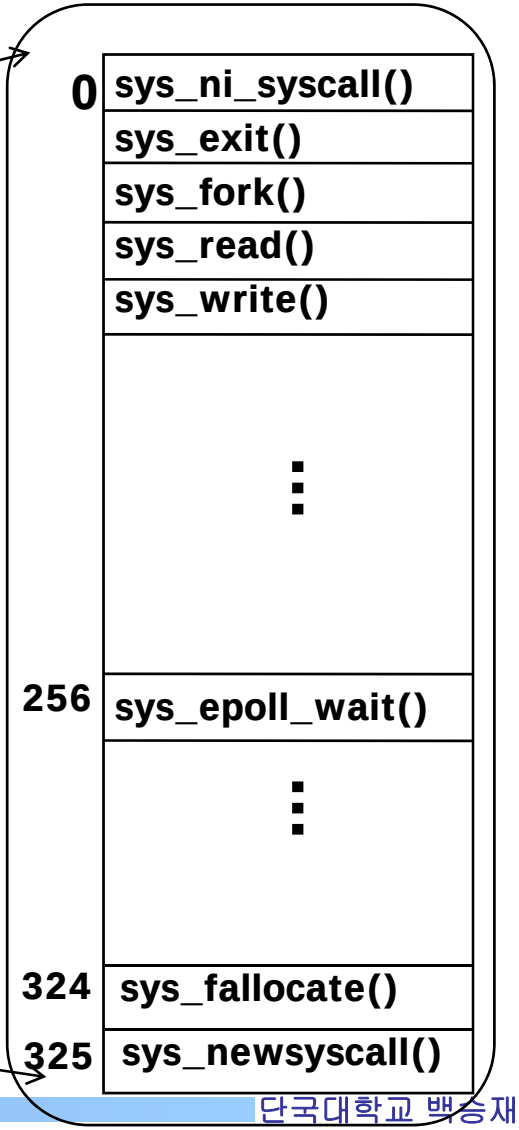
1. 새로운 시스템 호출 번호 할당

```
/* include/x86/unistd_32.h 파일의 내용 */

#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write              4
#define __NR_open               5
#define __NR_close              6
…
#define __NR_epoll_create       254
#define __NR_epoll_ctl          255
#define __NR_epoll_wait         256
…
#define __NR_fallocate          324
#define __NR_newsyscall         325

#define NR_syscalls    326
```

새로운 시스템 호출 구현 (3/7)

2. 새로운 시스템 호출 함수 sys_call_table[]에 등록

```
/* arch/x86/kernel/syscall_table_32.S 파일의 내용 */

ENTRY(sys_call_table)
        .long sys_restart_syscall
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open
        .long sys_close
        …
        .long sys_epoll_create
        .long sys_epoll_ctl
        .long sys_epoll_wait
        …
        .long sys_fallocate
        .long sys_newsyscall
```



새로운 시스템 호출 구현 (4/7)

3. 새로운 시스템 호출 함수 커널에 구현

```
/* kernel/newfile.c 파일의 내용 */
#include <linux/unistd.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/sched.h>

asmlinkage long sys_newsyscall(void)
{
        printk("<0>Hello Linux, I'm in Kernel\n");
        return 0;
}
EXPORT_SYMBOL_GPL(sys_newsyscall);
```

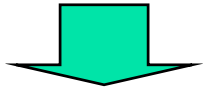
☞ printf()가 아니라 printk() 임에 주의

새로운 시스템 호출 구현 (5/7)

4. 커널 컴파일 및 리부팅

- 커널 컴파일 전에 makefile을 다음과 같이 수정해야 한다.

```
/* kernel/Makefile 의 변경 전 내용 */
obj-y    = sched.o fork.o exec_domain.o panic.o printk.o profile.o \
           exit.o itimer.o time.o softirq.o resource.o \
           sysctl.o capability.o ptrace.o timer.o user.o \
           signal.o sys.o kmod.o workqueue.o pid.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
```



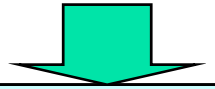
```
/* kernel/Makefile 의 변경 후 내용 */
obj-y    = sched.o fork.o exec_domain.o panic.o printk.o profile.o \
           exit.o itimer.o time.o softirq.o resource.o \
           sysctl.o capability.o ptrace.o timer.o user.o \
           signal.o sys.o kmod.o workqueue.o pid.o \
           rcupdate.o extable.o params.o posix-timers.o \
           kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
           hrtimer.o rwsem.o newfile.o
```

- 커널 컴파일 후 재부팅

새로운 시스템 호출 구현 (6/7)

1. 새로운 시스템 호출을 사용하는 사용자 수준 응용 작성

```
/* /usr/include/asm-x86/unistd_32.h */
……
#define __NR_signalfd          321
#define __NR_timerfd           322
#define __NR_eventd            323
#define __NR_fallocate         324
```



```
/* /usr/include/asm-x86/unistd_32.h */
……
#define __NR_signalfd          321
#define __NR_timerfd           322
#define __NR_eventd            323
#define __NR_fallocate         324
#define __NR_newsyscall        325
```

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

int main(void)
{
        syscall(325);
        return 0;
}
```

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

int main(void)
{
        syscall(__NR_newsyscall);
        return 0;
}
```

syscall 매크로

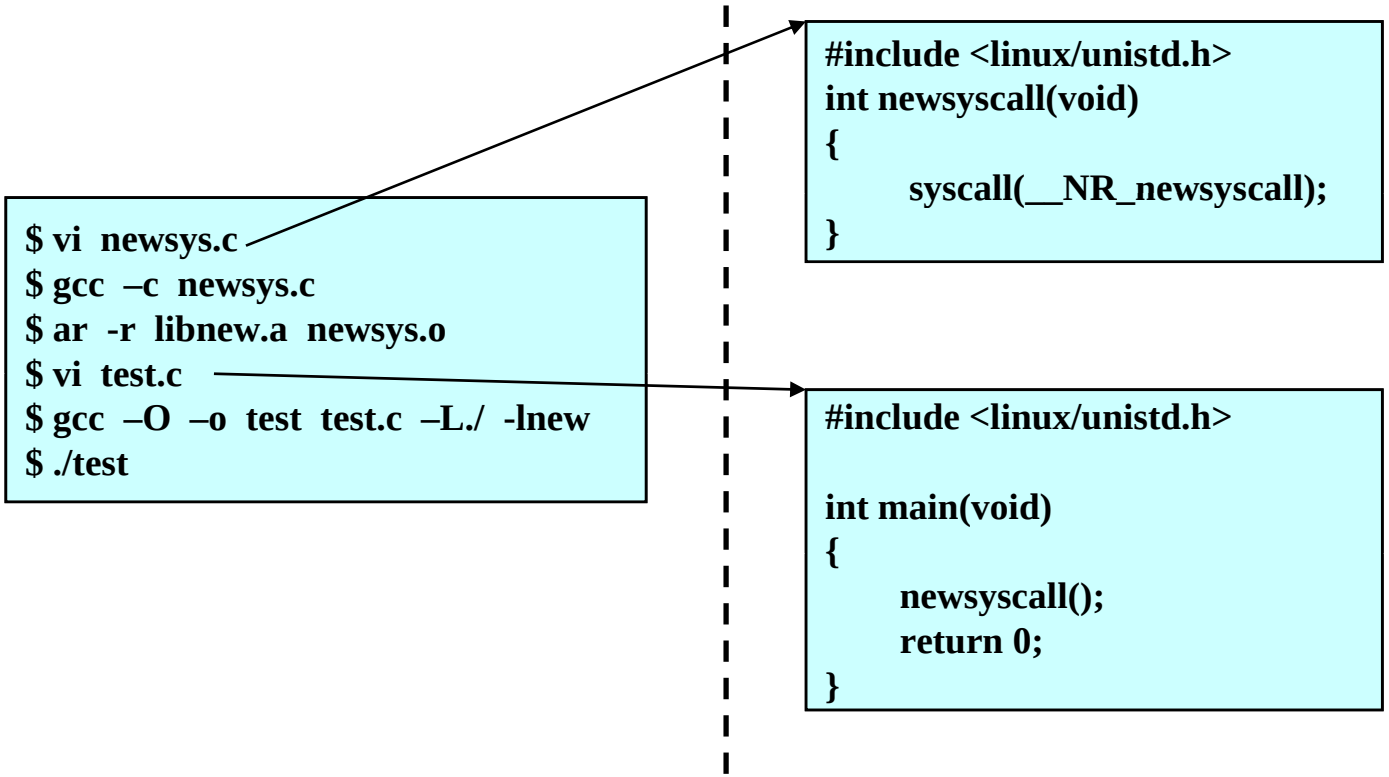
```
/* /usr/include/unistd.h */
extern long int syscall (long int __sysno, ...) __THROW;
```

```
/* glibc/sysdeps/unix/sysv/linux/i386/syscall.S */
 .text
ENTRY (syscall)

      PUSHARGS_6            /* Save register contents.  */
      _DOARGS_6(44)         /* Load arguments.  */
      movl 20(%esp), %eax   /* Load syscall number into %eax.  */
      ENTER_KERNEL          /* Do the system call.  */
      POPARGS_6             /* Restore register contents.  */
      cmpl $-4095, %eax     /* Check %eax for error.  */
      jae SYSCALL_ERROR_LABEL /* Jump to error handler if error.  */
L(pseudo_end):
      ret                   /* Return to caller.  */

PSEUDO_END (syscall)
```

```
/* glibc/sysdeps/unix/sysv/linux/i386/sysdep.h */
# define ENTER_KERNEL int $0x80
```

새로운 시스템 호출 구현 (7/7)

2. 라이브러리로 사용자 응용 작성



시스템 호출 구현 확장 (1/7)

- 인자 전달
- 기존 시스템 호출 분석
- 커널 정보 얻기
- 모듈 프로그래밍을 이용한 시스템 호출 구현 => 모듈 프로그래밍 장 참조

☞ ***Just Do It*** (百見不如一打)

시스템 호출 구현 확장 (2/7)

- 인자 전달 : show_mult(arg1, arg2, result)
  1. 새로운 시스템 호출 번호 할당 : 192번
  2. 새로운 시스템 호출 함수 sys_call_table[]에 등록
  3. 새로운 시스템 호출 함수 커널에 구현

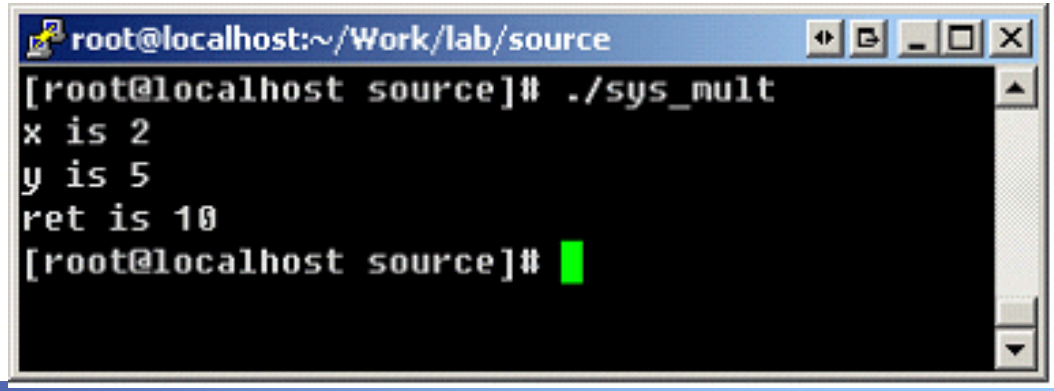
```
#include<linux/unistd.h>
#include<linux/kernel.h>
#include<asm-x86/uaccess.h>
asmlinkage int sys_show_mult(int x, int y, int* res)
{
   int error, compute;
   int i;
    error = access_ok(VERIFY_WRITE,res,sizeof(*res));
   if(error < 0)
   {
      printk("error in cdang\n");
      printk("error is %d\n",error);
      return error;
   }
   compute = x*y;
   printk("computeis %d\n",compute);
   i=  copy_to_user(res,&compute,sizeof(int));
   return 0;
}
```

  4. 커널 컴파일 및 리부팅

시스템 호출 구현 확장 (3/7)

- 인자 전달 : show_mult(arg1, arg2, result)
  1. 사용자 수준 응용

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

int main(void)
{
   int mult_ret = 0;
   int x = 2,y=5;
   int i;
   i=syscall(325,x,y,&mult_ret);
   printf("x is %d\ny is %d\nret is %d\n",x,y,mult_ret);

   return 0;
}
```



시스템 호출 구현 확장 (4/7)

- 커널 정보 얻기 : gettaskinfo()
  - ✓ header

```
//mystat.h

#include<linux/kernel.h>
#include<linux/sched.h>

struct mystat
{
    pid_t pid;
    pid_t ppid;
    int stat;
    int priority;
    int policy;
    long utime;
    long stime;
    long starttime;
    unsigned long min_flt;
    unsigned long maj_flt;
    long open_files;
};
```

시스템 호출 구현 확장 (5/7)

- 커널 정보 얻기 : gettaskinfo()
  - ✓ 커널 함수

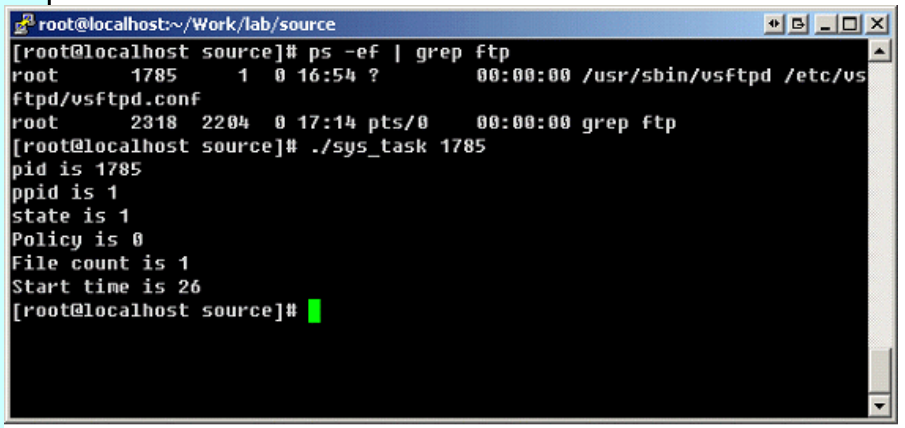
```
asmlinkage int sys_gettaskinfo(int id, struct mystat *user_buf)
{
   struct mystat *buf;
   int i, cnt = 0;
   struct task_struct *search;
   struct file *fp;
   search = &init_task;
   while(search->pid != id)
   {
   //  search = next_task(search);  //아래 list_entry 대신 사용가능
      search = list_entry((search)->tasks.next,struct task_struct,tasks);
      if(search->pid == init_task.pid)
      {
         printk("init_task\n");
         return -1;
      }
   }
```

```
   buf = (char*)kmalloc(sizeof(struct mystat),GFP_KERNEL);
   if(buf == NULL)
   {
      printk("buf is NULL\n");
      return -1;
   }
   buf->pid = search->pid;
   buf->ppid = search->parent->pid;
   buf->stat = search->state;
   buf->priority = search->prio;
   buf->policy = search ->policy;
   buf->utime = search->utime;
   buf->stime = search->stime;
   buf->starttime = search->start_time.tv_sec;
   buf->min_flt = search->min_flt;
   buf->maj_flt = search->maj_flt;
   for(i = 0; i<32; i++) {
      if(  (search->files->fd_array[i]) !=NULL) {
         cnt++;
      }
   }
   buf->open_files = cnt;
   copy_to_user(user_buf,buf,sizeof(struct mystat));
   return 0;
}
```

시스템 호출 구현 확장 (6/7)

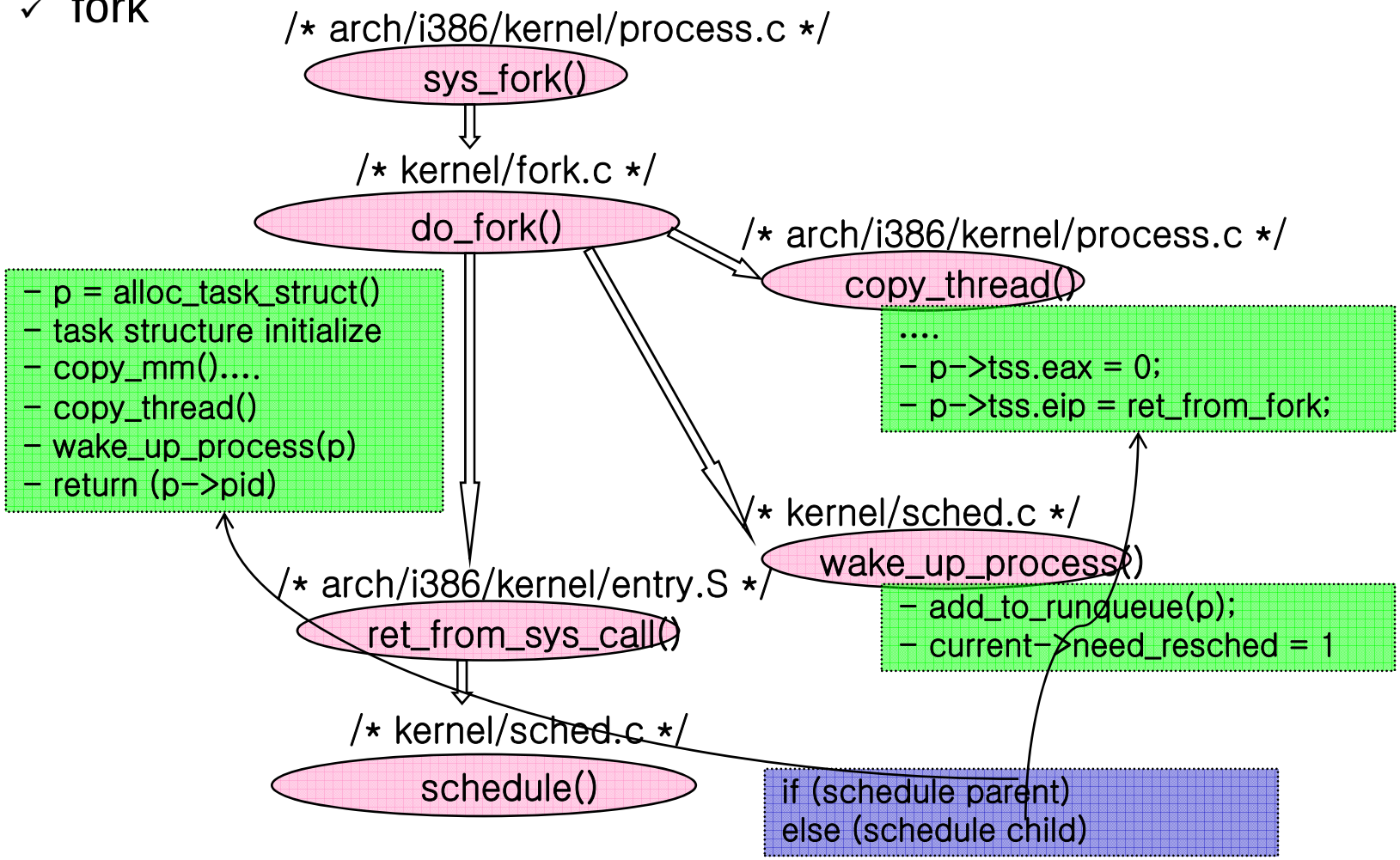
- 커널 정보 얻기 : gettaskinfo()
  - ✓ 사용자 수준 응용

```
#include"mystat.h"
#include<linux/unistd.h>

int main(int argc, char* argv[])
{
   int task_number;
   struct mystat* mybuf;
   if(argc !=2)
   {
      printf("Usage : a.out pid \n");
      exit(1);
   }
   task_number = atoi(argv[1]);
   mybuf = (char*)malloc(sizeof(struct mystat));
   if(mybuf == NULL)
      exit(1);
   syscall(326,task_number,mybuf);
   printf("pid is %d\n",(int)mybuf->pid);
   printf("ppid is %d\n",(int)mybuf->ppid);
   printf("state is %d\n",(int)mybuf->stat);
   printf("Policy is %d\n",(int)mybuf->policy);
   printf("File count is %d\n",mybuf->open_files);
   printf("Start time is %d\n",mybuf->starttime);
   return 0;
}
```



시스템 호출 구현 확장 (7/7)

- 기존 시스템 호출 분석
  - ✓ fork



모듈 프로그래밍의 동기

- Linux에서 모듈 프로그래밍이 개발된 이유
  - ✓ Linux는 모노리딕 커널(monolithic kernel)
    - 모든 기능이 커널 내부에 구현되어 있다.
    - 사소한 커널 변경(trivial modification)에도 커널 컴파일과 리부팅이 필요
    - 커널의 크기가 너무 크다.
    - 커널의 일부 기능은 거의 사용되지 않지만, 메모리에 상주하고 있다.
  - ✓ 모듈 프로그래밍 module: steps toward micro-kernelized Linux
    - 커널의 일부 기능을 커널에서 빼고 모듈로 구현
    - 그 기능이 필요할 때만 메모리에 적재
    - 작고 깔끔한 커널 가능 (small and clean kernel)
    - 메모리의 효율적 이용
    - 커널 기능의 수정 때마다 컴파일 및 리부팅의 필요가 없다.

모듈 프로그래밍의 범위

■ 모듈 프로그래밍의 범위

- ✓ 하드웨어 종속적인 부분과 초기화 부분을 제외한 거의 모든 커널 기능을 모듈로 만들 수 있다.
- ✓ 주로 파일 시스템과 디바이스 드라이버

- ✓ 현재 Linux가 지원하는 모듈 인터페이스

file system	register_filesystem, unregister_filesystem read_super, put_super
block device driver	register_blkdev, unregister_blkdev open, release
character device driver	register_chrdev, unregister_chrdev open, release
network device driver	register_netdev, unregister_netdev open, close
exec domain	register_exec_domain, unregister_exec_domain load_binary, personality
binary format	register_binfmt, unregister_binfmt load_binary
....	

모듈의 관리

- 모듈의 메모리 적재
  - ✓ insmod, lsmod, rmmod, modprobe

```
#insmod  fat.o
#lsmod
Module:     #pages :   Used by
fat              6              0
#rmmod  fat
```

  - ✓ kerneld (kmod) : 자동 모듈 적재 데몬
    - eg: mount -t msdos /dev/fd0 /mnt => transparent load fat & msdos modules

모듈 프로그래밍 예 (1/3) - 2.6

- 2.6 커널을 위한 hello_module

```
#include <linux/kernel.h>
#include <linux/module.h>

int hello_module_init(void)
{
        printk(KERN_EMERG "Hello Module~! I'm in Kernel₩n");
        return 0;
}

void hello_module_cleanup(void)
{
        printk("<0>Bye Module~!₩n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);

MODULE_LICENSE("GPL");
```

모듈 프로그래밍 예 (2/3) – 2.6

- 2.6 커널에서 모듈 컴파일을 위한 Makefile

```
O_TARGET        := hello_module.ko
obj-m           := hello_module.o

KERNEL_DIR      := /lib/modules/$(shell uname -r)/build
MODULE_DIR      := /lib/modules/$(shell uname -r)/kernel/hello_module
PWD             := $(shell pwd)

default :
        $(MAKE)  -C  $(KERNEL_DIR)  SUBDIRS=$(PWD)  modules
install :
        mkdir  -p  $(MODULE_DIR)
        cp  -f  $(O_TARGET)  $(MODULE_DIR)
clean :
        $(MAKE)  -C  $(KERNEL_DIR)  SUBDIRS=$(PWD)  clean
```

모듈 프로그래밍 예 (3/3) – 2.6

- 2.6 커널에서 모듈 테스트를 위한 명령어 수행

```
$ vi Makefile
$ vi hello_module.c
$ ls
Makefile        hello_module.c
$ make
$ ls
Makefile        hello_module.c        hello_module.ko        ……
$ insmod hello_module.ko
Hello Module~!  I'm in Kernel
$ rmmod hello_module
Bye Module~!
$ make install
…
$ ls  –l  /lib/modules/2.6.18/kernel/hello_module
hello_module.ko
$ make clean
…
$ ls
Makefile        hello_module.c
```

모듈 프로그램 : system call wrapper (1/3) – 2.6

- 2.6 커널에서 모듈을 통한 system call wrapper

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm-i386/unistd.h>
#include <asm-i386/pgtable.h>

unsigned long **sys_call_table;
unsigned long **locate_sys_call_table(void)
{
        unsigned long temp;
        unsigned *p;
        unsigned long **sys_table;
        for ( temp = 0xc0000000; temp < 0xd0000000; temp+= sizeof(void *)){
                p = (unsigned long *)temp;
                if( p[__NR_close] == (unsigned long)sys_close){
                        sys_table = (unsigned long **)p;
                        return &sys_table[0];
                }
        }
        return NULL;
}
asmlinkage int (*original_call)(const char *, int, int);
asmlinkage int sys_our_open(const char *filename, int flags, int mode){
        printk("<0>open system call₩n");
        return (original_call(filename, flags, mode));
}
```

모듈 프로그램 : system call wrapper (2/3) – 2.6


```
#define __flush_tlb_single(addr) ₩
        __asm__ __volatile__("invlpg (%0)" ::"r" (addr) : "memory")
int syscall_hooking_init(void)
{
        if( (sys_call_table = locate_sys_call_table()) == NULL){
                printk("<0> Can't find sys_call_table₩n");
                return -1;
        }
        pgd_t *pgd = pgd_offset_k((unsigned long)sys_call_table);
        pud_t *pud;         pmd_t *pmd;         pte_t *pte;
        if( pgd_none(*pgd))     return NULL;
        pud = pud_offset(pgd, (unsigned long)sys_call_table);
        if( pud_none(*pud))     return NULL;
        pmd = pmd_offset(pud, (unsigned long)sys_call_table);
        if( pmd_none(*pmd))   return NULL;
        if( pmd_large(*pmd)){
                pte = (pte_t *)pmd;
        }else{
                pte = pte_offset_kernel(pmd, (unsigned long)sys_call_table);
        }
        pte->pte_low |= _PAGE_KERNEL;
        __flush_tlb_single((unsigned long)sys_call_table);
        printk("<0> sys_call_table is loaded at %p₩n", sys_call_table);

        original_call = (void *)sys_call_table[__NR_open];
        sys_call_table[__NR_open] = (void *)sys_our_open;

        printk("<0> Module Init₩n");
        return 0;
}
```

모듈 프로그램 : system call wrapper (3/3) – 2.6


```
void syscall_hooking_cleanup(void)
{
        sys_call_table[__NR_open] = original_call;
        printk("<0> Module cleanup₩n");
}

module_init(syscall_hooking_init);
module_exit(syscall_hooking_cleanup);
MODULE_LICENSE("GPL");
```