

# System Call, Process, Thread

2022년 1학기 고려대학교 컴퓨터학과 운영체제 실습 1

조교: 유문상 (anstkd07@gmail.com)

# 목차

## 1. 시스템 콜

1. 시스템 콜이란
2. 시스템 콜의 과정
3. 시스템 콜 사용해보기

## 2. 프로세스

1. Introduction
2. Fork()
3. Exec()
4. Wait()

## 3. 스레드

1. Introduction
2. Thread Creation
3. Thread Termination
4. Joining with a Terminated Thread

## 4. 과제 설명

### 부록1. 실습 환경 설정

# Tutorial 제도



# 1. System Call

# 시스템 콜이란?

**시스템 호출** 또는 **시스템 콜**(system call), 간단히 **시스콜**(syscall)은 운영 체제의 커널이 제공하는 서비스에 대해, 응용 프로그램의 요청에 따라 커널에 접근하기 위한 인터페이스이다. 보통 C나 C++과 같은 고급 언어로 작성된 프로그램들은 직접 시스템 호출을 사용할 수 없기 때문에 고급 API를 통해 시스템 호출에 접근하게 하는 방법이다. (위키피디아)

# 시스템 콜이란?

시스템 호출 또는 시스템 콜(system call), 간단히 시스콜(syscall)은 운영 체제의 커널이 제공하는 서비스에 대해, 응용 프로그램의 요청에 따라 커널에 접근하기 위한 인터페이스이다. 보통 C나 C++과 같은 고급 언어로 작성된 프로그램들은 직접 시스템 호출을 사용할 수 없기 때문에 고급 API를 통해 시스템 호출에 접근하게 하는 방법이다. (위키피디아)

Application Programming Interface



# 시스템 콜이란?

## Types of System Calls

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes

## Types of System Calls

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device information
  - set process, file, or device information

## Types of System Calls

- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

시스템  
위한 인  
게 하는

근하기  
근하

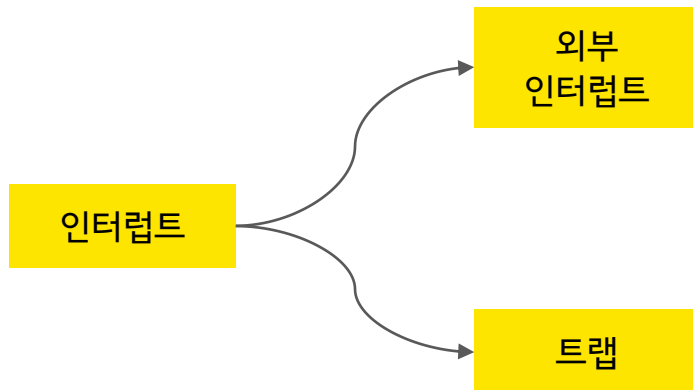
# 인터럽트

## 인터럽트

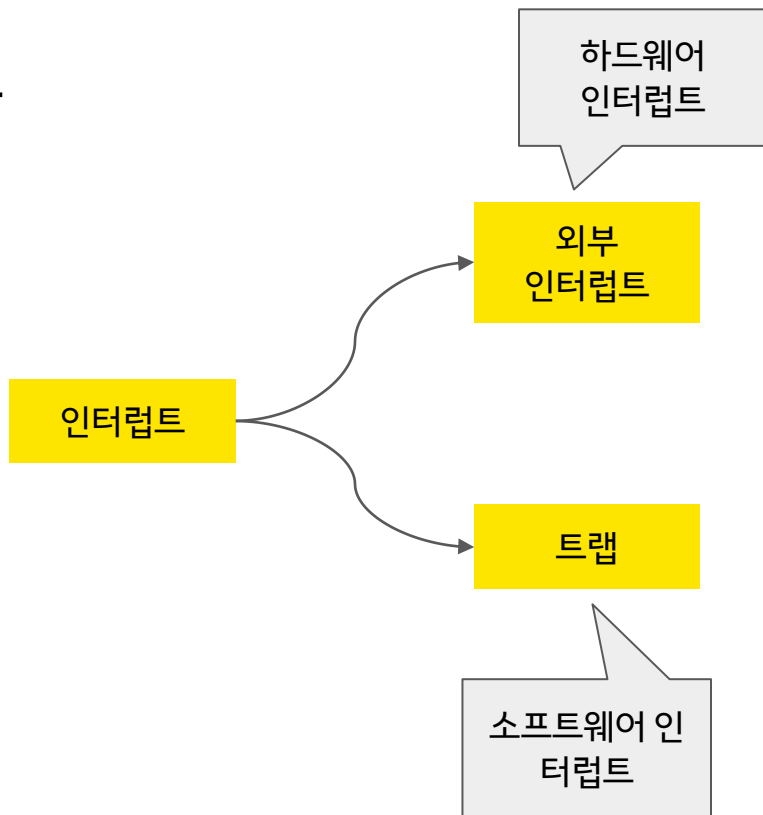
마이크로프로세서에서 인터럽트(interrupt)란 마이크로프로세서(CPU)가 프로그램을 실행하고 있을 때, 입출력 하드웨어 등의 장치에 예외상황이 발생하여 처리가 필요할 경우에 마이크로프로세서에게 알려 처리할 수 있도록 하는 것을 말한다. (위키 피디아)



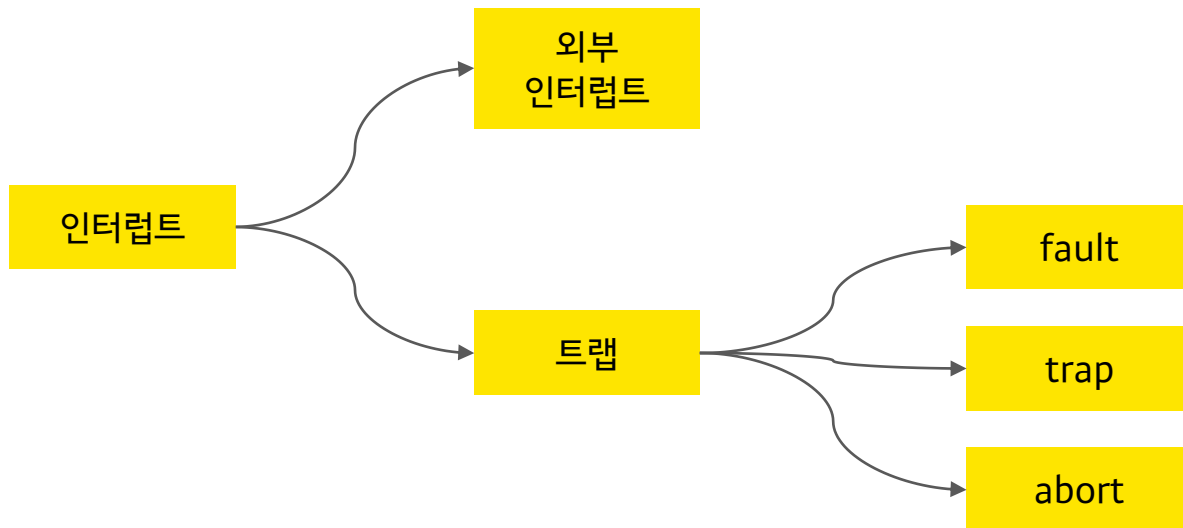
# 인터럽트



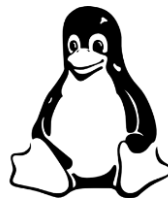
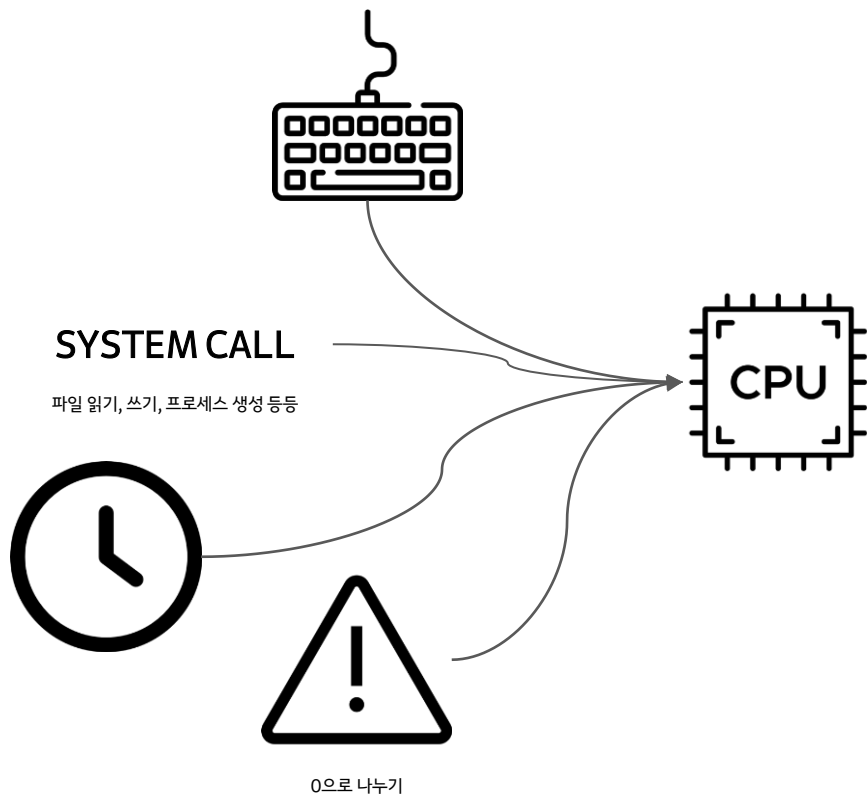
# 인터럽트



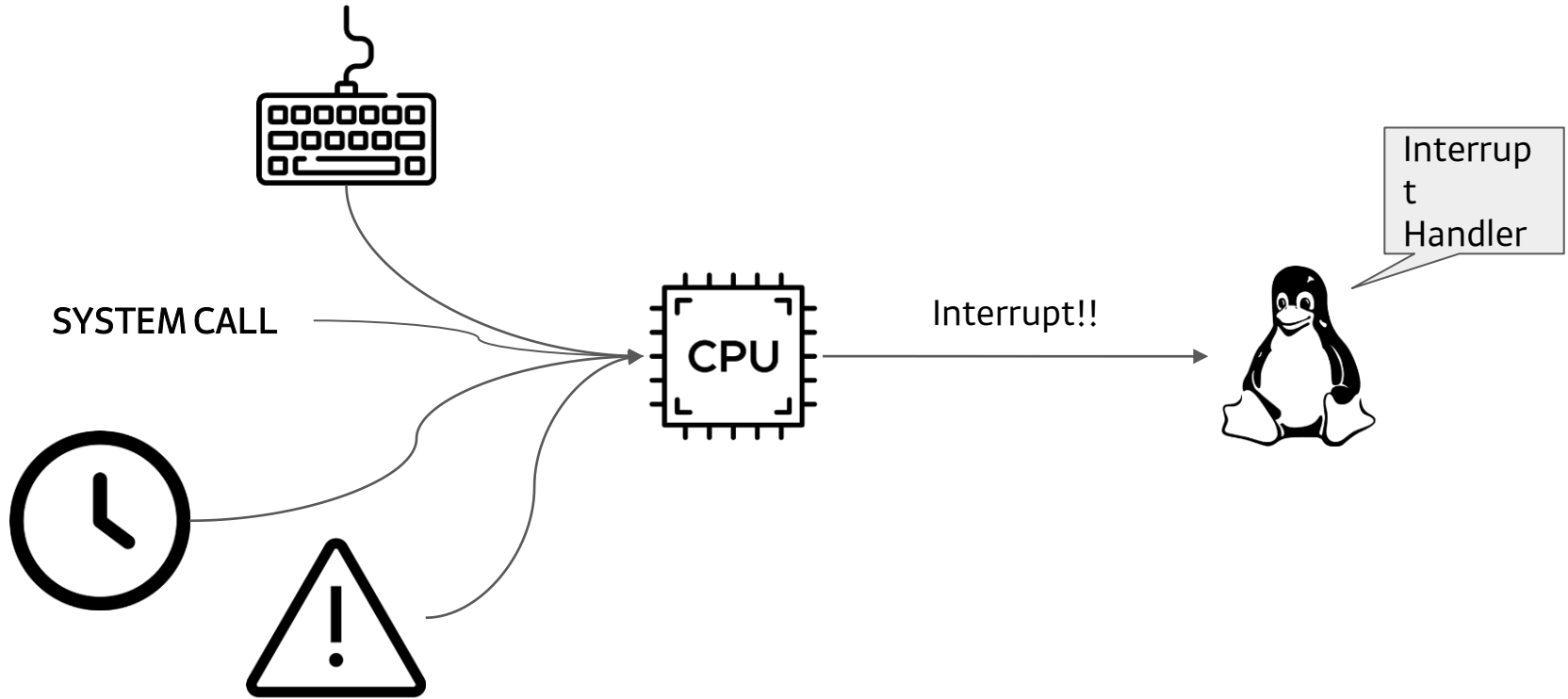
# 인터럽트



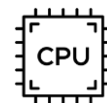
# 인터럽트



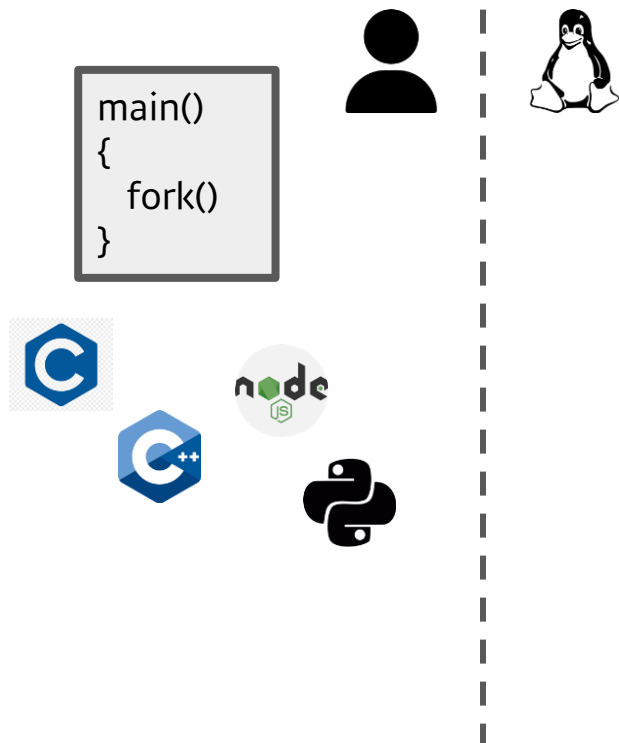
# 인터럽트



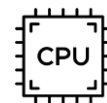
# 시스템 콜 처리 과정



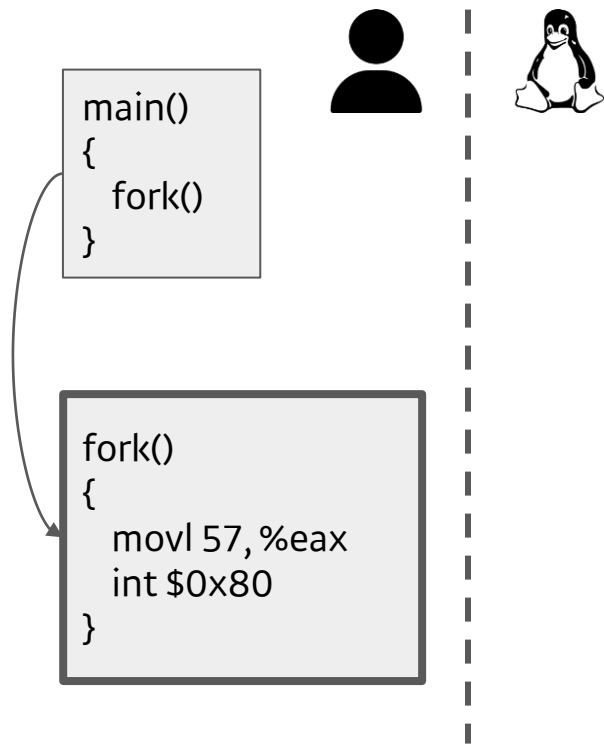
사용자 모드



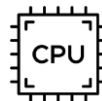
# 시스템 콜 처리 과정



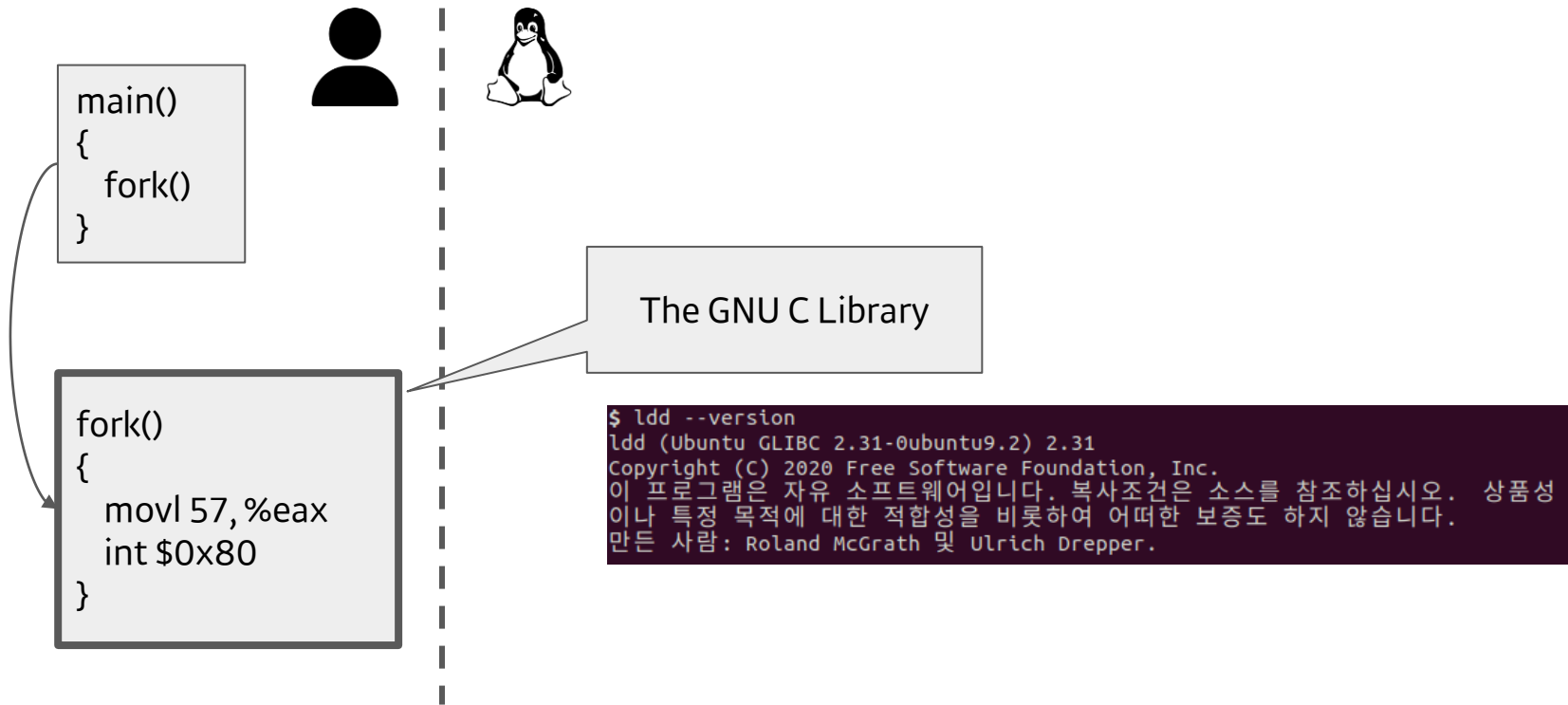
사용자 모드



# 시스템 콜 처리 과정

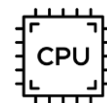


사용자 모드

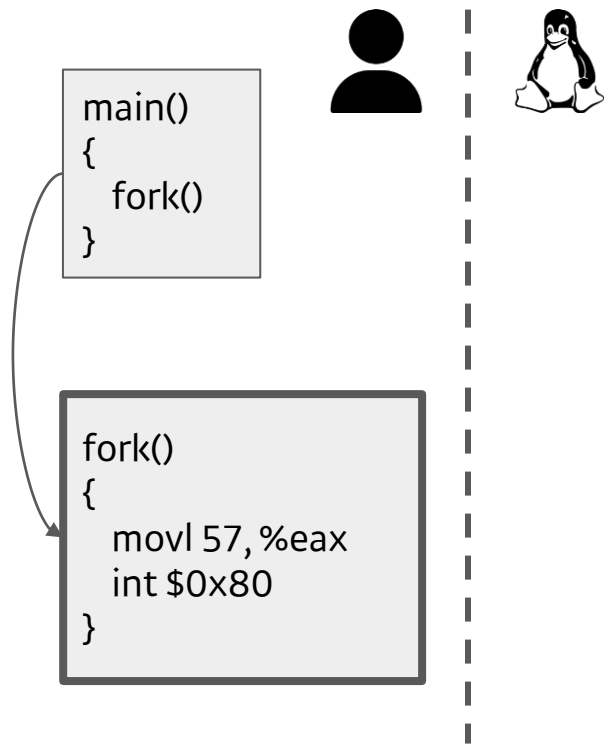




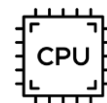
# 시스템 콜 처리 과정



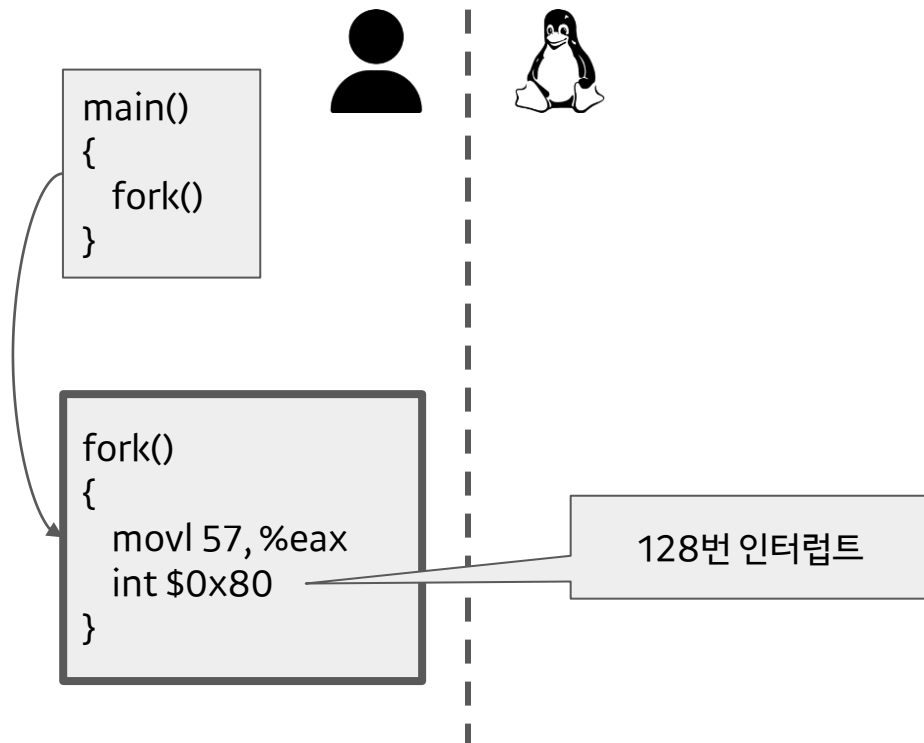
사용자 모드



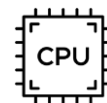
# 시스템 콜 처리 과정



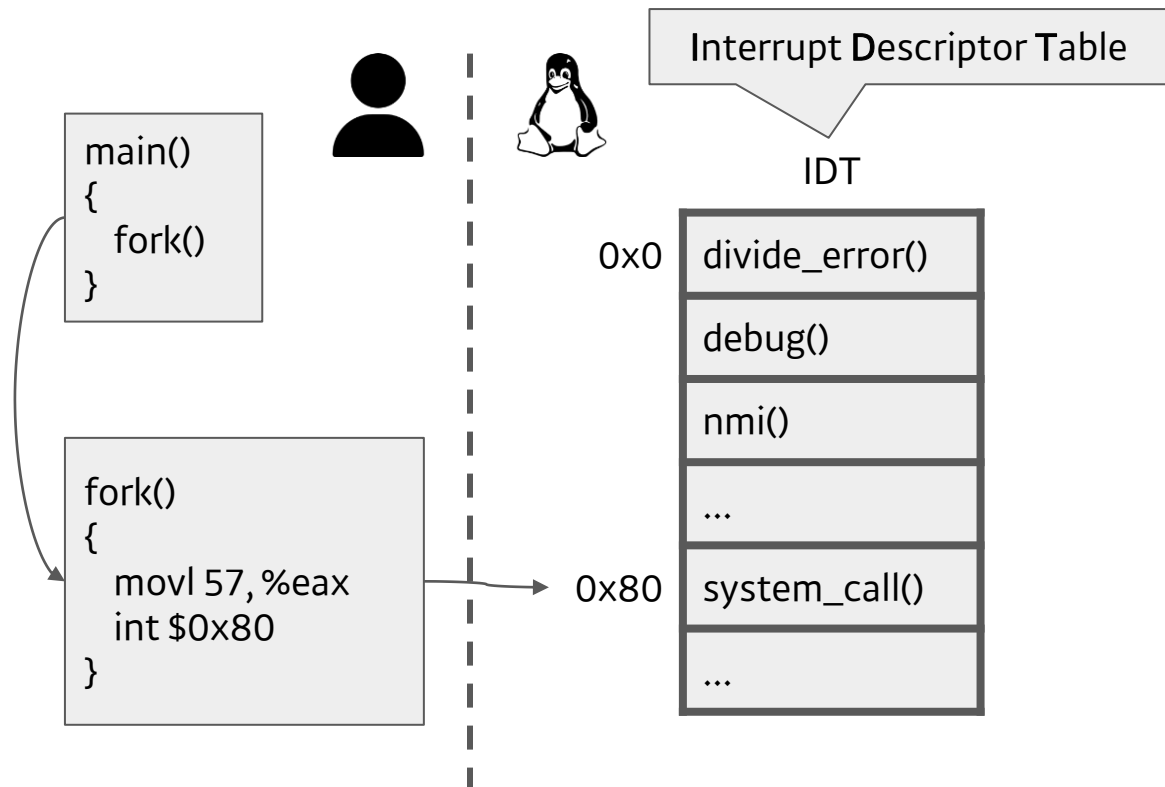
사용자 모드



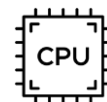
# 시스템 콜 처리 과정



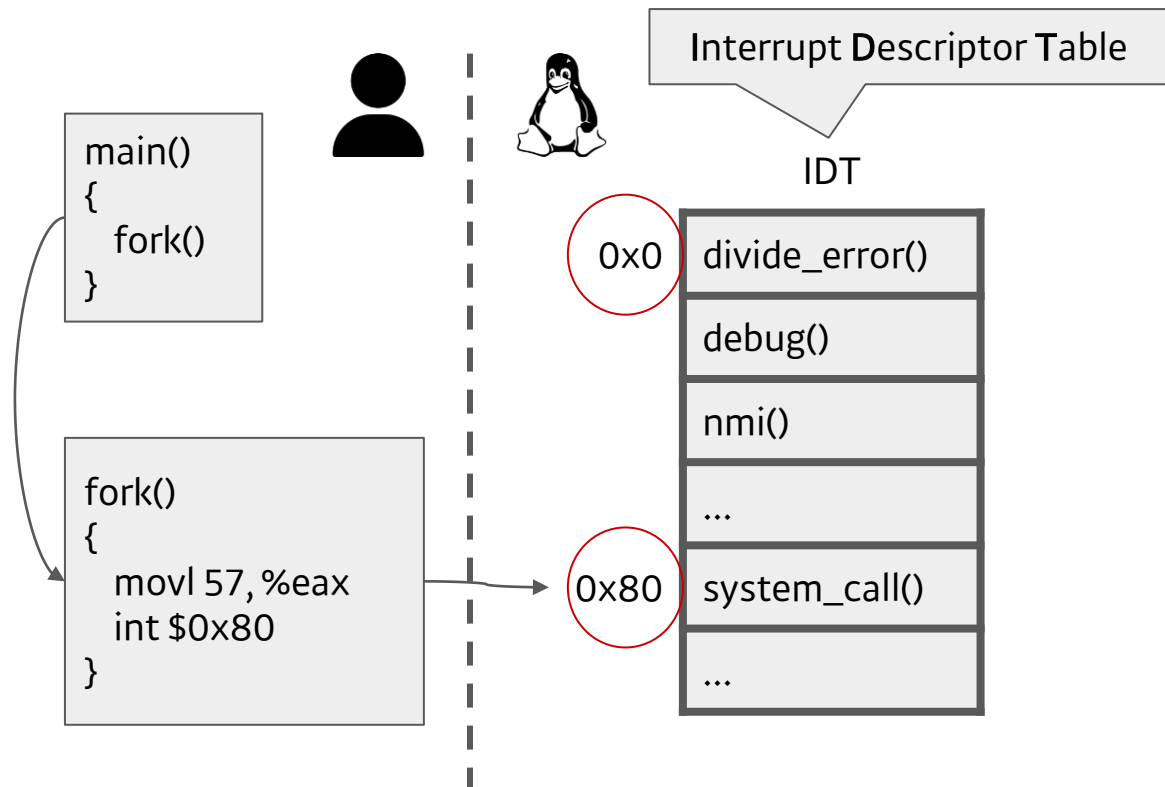
커널 모드



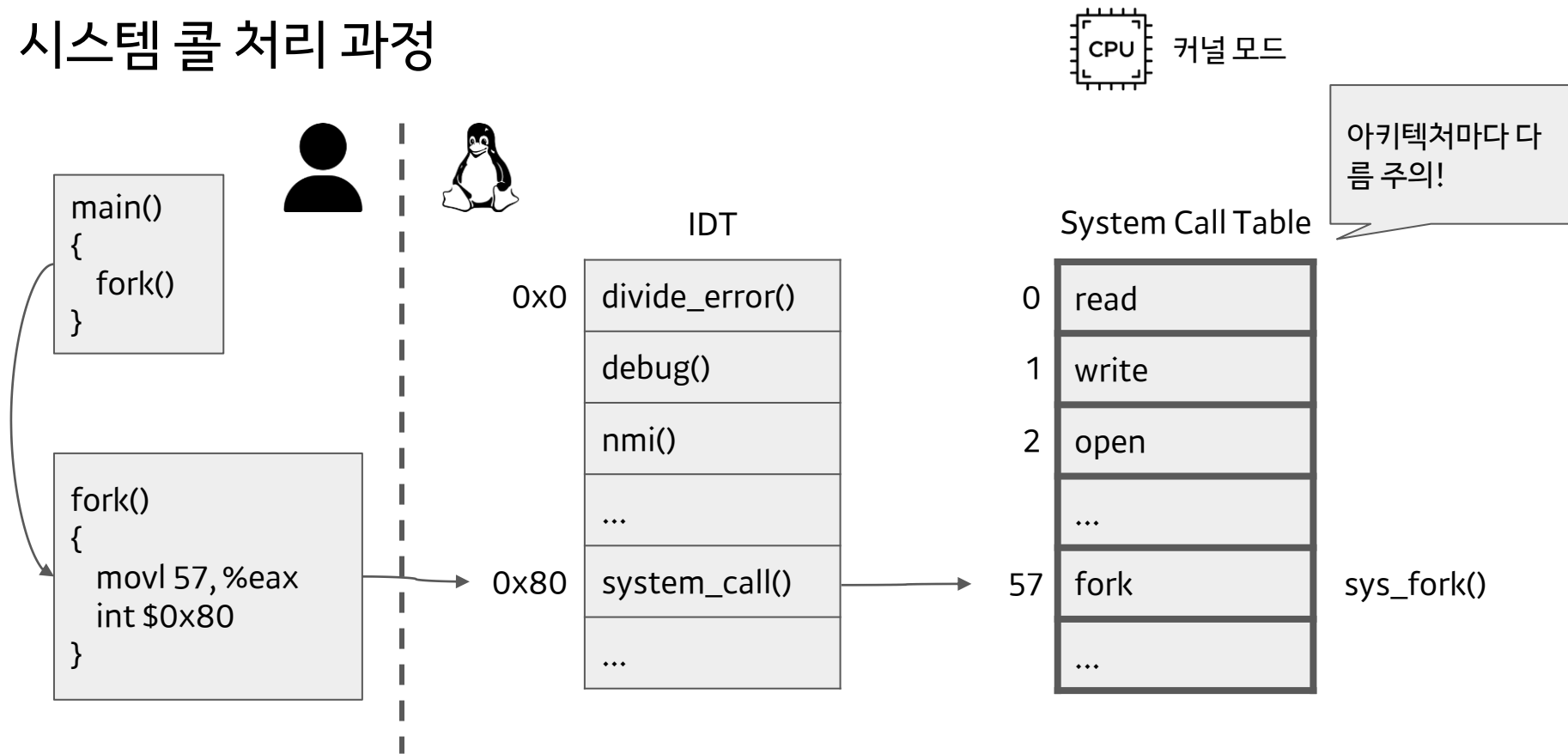
# 시스템 콜 처리 과정



커널 모드



# 시스템 콜 처리 과정



# ./arch/x86/kernel/idt.c

IDT

0x0	divide_error()
	debug()
	nmi()
	...
0x80	system_call()
	...

./arch/x86/include/asm/irq\_vectors.h

```
#define IA32_SYSCALL_VECTOR 0x80
```

Intel Architecture

Distributed & C

```
68 /*
69  * The default IDT entries which are set up in trap_init() before
70  * cpu_init() is invoked. Interrupt stacks cannot be used at that point and
71  * the traps which use them are reinitialized with IST after cpu_init() has
72  * set up TSS.
73  */
74 static const __initconst struct idt_data def_idts[] = {
75     INTG(X86_TRAP_DE,      divide_error),
76     INTG(X86_TRAP_NMI,     nmi),
77     INTG(X86_TRAP_BR,      bounds),
78     INTG(X86_TRAP_UD,      invalid_op),
79     INTG(X86_TRAP_NM,      device_not_available),
80     INTG(X86_TRAP_OLD_MF,  coprocessor_segment_overrun),
81     INTG(X86_TRAP_TS,      invalid_TSS),
82     INTG(X86_TRAP_NP,      segment_not_present),
83     INTG(X86_TRAP_SS,      stack_segment),
84     INTG(X86_TRAP_GP,      general_protection),
85     INTG(X86_TRAP_SPURIOUS, spurious_interrupt_bug),
86     INTG(X86_TRAP_MF,      coprocessor_error),
87     INTG(X86_TRAP_AC,      alignment_check),
88     INTG(X86_TRAP_XF,      simd_coprocessor_error),
89
90     #ifdef CONFIG_X86_32
91         TSKG(X86_TRAP_DF,      GDT_ENTRY_DOUBLEFAULT_TSS),
92     #else
93         INTG(X86_TRAP_DF,      double_fault),
94     #endif
95     INTG(X86_TRAP_DB,      debug),
96
97     #ifdef CONFIG_X86_MCE
98         INTG(X86_TRAP_MC,      &machine_check),
99     #endif
100
101     SYSG(X86_TRAP_OF,      overflow),
102     #if defined(CONFIG_IA32_EMULATION)
103         SYSG(IA32_SYSCALL_VECTOR, entry_INT80_compat),
104     #elif defined(CONFIG_X86_32)
105         SYSG(IA32_SYSCALL_VECTOR, entry_INT80_32),
106     #endif
107 };
```

# ./arch/x86/entry/entry\_32.S (entry\_64\_compat.S)

```
/*
 * 32-bit legacy system call entry.
 *
 * 32-bit x86 Linux system calls traditionally used the INT $0x80
 * instruction. INT $0x80 lands here.
 *
 * This entry point can be used by any 32-bit perform system calls.
 * Instances of INT $0x80 can be found inline in various programs and
 * libraries. It is also used by the vDSO's __kernel_vsyscall
 * fallback for hardware that doesn't support a faster entry method.
 * Restarted 32-bit system calls also fall back to INT $0x80
 * regardless of what instruction was originally used to do the system
 * call. (64-bit programs can use INT $0x80 as well, but they can
 * only run on 64-bit kernels and therefore land in
 * entry_INT80_compat.)
 *
 * This is considered a slow path. It is not used by most libc
 * implementations on modern hardware except during process startup.
 *
 * Arguments:
 * eax system call number
 * ebx arg1
 * ecx arg2
 * edx arg3
 * esi arg4
 * edi arg5
 * ebp arg6
 */
ENTRY(entry_INT80_32)
    ASM_CLAC
    pushl %eax          /* pt_regs->orig_ax */
    SAVE_ALL pt_regs_ax=$-ENOSYS /* save rest */

    /*
     * User mode is traced as though IRQs are on, and the interrupt gate
     * turned them off.
     */
    TRACE_IRQS_OFF

    movl %esp, %eax
    call do_int80_syscall_32
.Lsyscall_32_done:
```

# ./arch/x86/entry/syscalls/syscall\_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0 common read sys_read
1 common write sys_write
2 common open sys_open
3 common close sys_close
4 common stat sys_newstat
5 common fstat sys_newfstat
6 common lstat sys_newlstat
7 common poll sys_poll
8 common lseek sys_lseek
9 common mmap sys_mmap
10 common mprotect sys_mprotect
11 common munmap sys_munmap
12 common brk sys_brk
13 64 rt_sigaction sys_rt_sigaction
14 common rt_sigprocmask sys_rt_sigprocmask
15 64 rt_sigreturn sys_rt_sigreturn/ptregs
16 64 ioctl sys_ioctl
17 common pread64 sys_pread64
18 common pwrite64 sys_pwrite64
19 64 readv sys_readv
20 64 writev sys_writev
21 common access sys_access
22 common pipe sys_pipe
23 common select sys_select
24 common sched_yield sys_sched_yield
25 common mremap sys_mremap
26 common msync sys_msync
27 common mincore sys_mincore
28 common madvise sys_madvise
29 common shmget sys_shmget
30 common shmat sys_shmat
31 common shmctl sys_shmctl
32 common dup sys_dup
```

```
51 common getsockname sys_getsockname
52 common getpeername sys_getpeername
53 common socketpair sys_socketpair
54 64 setsockopt sys_setsockopt
55 64 getsockopt sys_getsockopt
56 common clone sys_clone/ptregs
57 common fork sys_fork/ptregs
58 common vfork sys_vfork/ptregs
59 64 execve sys_execve/ptregs
60 common exit sys_exit
61 common wait4 sys_wait4
62 common kill sys_kill
63 common uname sys_newuname
64 common semget sys_semget
65 common semop sys_semop
66 common semctl sys_semctl
67 common shmctl sys_shmctl
68 common msgget sys_msgget
69 common msgsnd sys_msgsnd
70 common msgrcv sys_msgrcv
71 common msgctl sys_msgctl
72 common fcntl sys_fcntl
73 common flock sys_flock
74 common fsync sys_fsync
75 common fdatsync sys_fdatsync
76 common truncate sys_truncate
77 common ftruncate sys_ftruncate
78 common getdents sys_getdents
79 common getcwd sys_getcwd
80 common chdir sys_chdir
81 common fchdir sys_fchdir
82 common rename sys_rename
83 common mkdir sys_mkdir
```

```
303 common name_to_handle_at sys_name_to_handle_at
304 common open_by_handle_at sys_open_by_handle_at
305 common clock_adjtime sys_clock_adjtime
306 common syncfs sys_syncfs
307 64 sendmsg sys_sendmsg
308 common setns sys_setns
309 common getcpu sys_getcpu
310 64 process_vm_readv sys_process_vm_readv
311 64 process_vm_writev sys_process_vm_writev
312 common kcmp sys_kcmp
313 common finit_module sys_finit_module
314 common sched_setattr sys_sched_setattr
315 common sched_getattr sys_sched_getattr
316 common renameat2 sys_renameat2
317 common seccomp sys_seccomp
318 common getrandom sys_getrandom
319 common memfd_create sys_memfd_create
320 common kexec_file_load sys_kexec_file_load
321 common bpf sys_bpf
322 64 execveat sys_execveat/ptregs
323 common userfaultfd sys_userfaultfd
324 common membarrier sys_membarrier
325 common mlock2 sys_mlock2
326 common copy_file_range sys_copy_file_range
327 64 preadv2 sys_preadv2
328 64 pwritev2 sys_pwritev2
329 common pkey_mprotect sys_pkey_mprotect
330 common pkey_alloc sys_pkey_alloc
331 common pkey_free sys_pkey_free
332 common statx sys_statx
```



헤더 파일들이 있는 곳

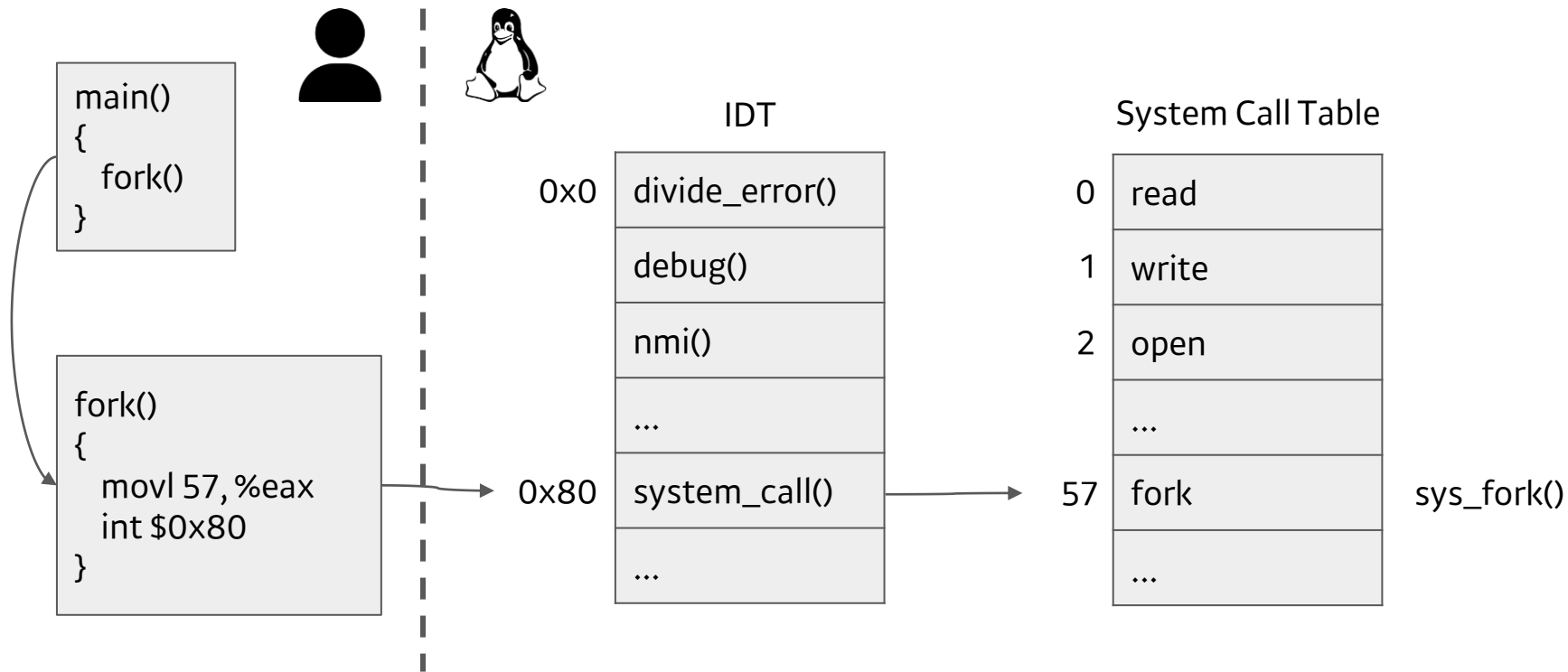
# ./include/linux/syscalls.h

```
asmlinkage long sys_exit(int error_code);
asmlinkage long sys_exit_group(int error_code);
asmlinkage long sys_wait4(pid_t pid, int __user *stat_addr,
                          int options, struct rusage __user *ru);
asmlinkage long sys_waitid(int which, pid_t pid,
                          struct siginfo __user *infop,
                          int options, struct rusage __user *ru);
asmlinkage long sys_waitpid(pid_t pid, int __user *stat_addr, int options);
asmlinkage long sys_set_tid_address(int __user *tidptr);
asmlinkage long sys_futex(u32 __user *uaddr, int op, u32 val,
                          struct timespec __user *utime, u32 __user *uaddr2,
                          u32 val3);
```

```
asmlinkage long sys_fork(void);
asmlinkage long sys_vfork(void);
#ifdef CONFIG_CLONE_BACKWARDS
asmlinkage long sys_clone(unsigned long, unsigned long, int __user *, unsigned
                          __user *);
#else
#ifdef CONFIG_CLONE_BACKWARDS3
asmlinkage long sys_clone(unsigned long, unsigned long, int, int __user *,
                          int __user *, unsigned long);
#else
asmlinkage long sys_clone(unsigned long, unsigned long, int __user *,
                          int __user *, unsigned long);
#endif
#endif
```

```
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
asmlinkage long sys_readahead(int fd, loff_t offset, size_t count);
asmlinkage long sys_readv(unsigned long fd,
                          const struct iovec __user *vec,
                          unsigned long vlen);
asmlinkage long sys_write(unsigned int fd, const char __user *buf,
                          size_t count);
asmlinkage long sys_writev(unsigned long fd,
                          const struct iovec __user *vec,
                          unsigned long vlen);
asmlinkage long sys_pread64(unsigned int fd, char __user *buf,
                          size_t count, loff_t pos);
asmlinkage long sys_pwrite64(unsigned int fd, const char __user *buf,
                          size_t count, loff_t pos);
```

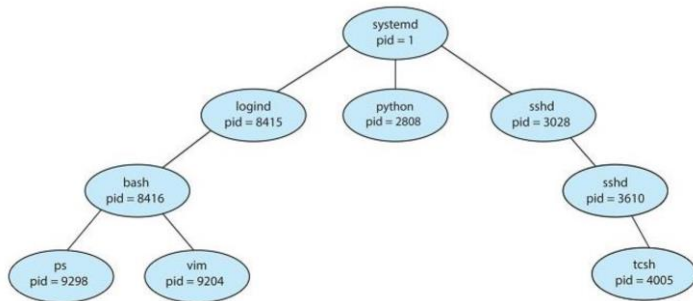
# 시스템 콜 처리 과정



## 2. Process

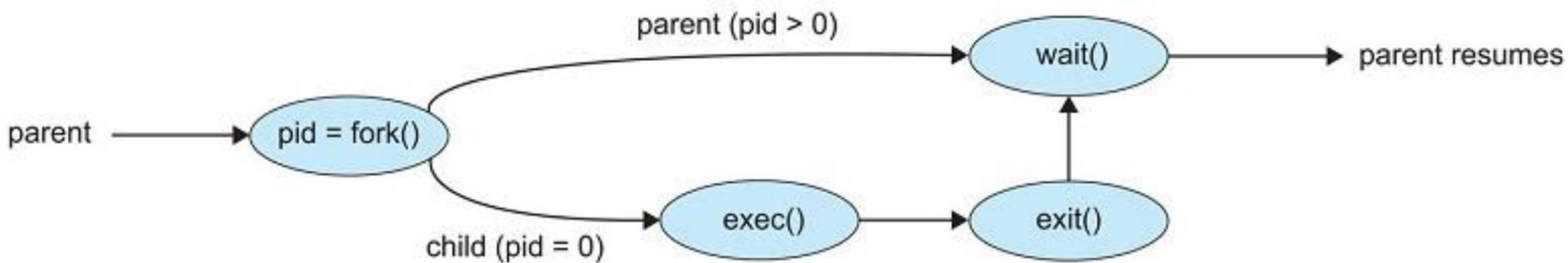
# Process Introduction

- 본 실습은 Chapter3. Process 슬라이드 22쪽~27쪽의 내용을 다룹니다.
- 실습의 목적 : 우리가 쓰는 컴퓨터의 프로세스가 어떻게 생성되고 관리되는지 관찰한다.
- 프로세스는 실행 중에 여러 개의 새로운 프로세스를 생성할 수 있다.
  - 생성한 프로세스를 부모 프로세스로, 생성된 프로세스를 자식 프로세스로 부름.
  - 프로세스는 프로세스 식별자(pid)로 구분됨.



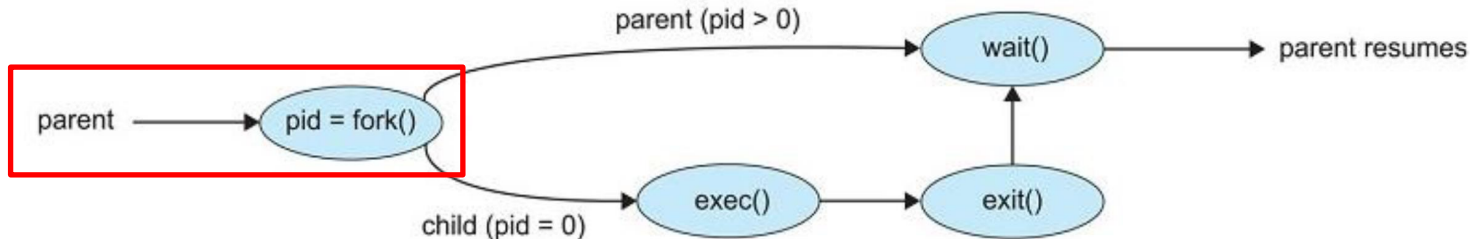
# Process Introduction

- 새로운 프로세스를 생성(-> fork()) 한 후에 상황...
  - 부모 프로세스는 무엇을 할까?
    - 부모가 계속해서 자식과 병렬로 실행되거나
    - 부모가 자식이 끝날 때까지 기다리거나 -> wait()
  - 자식 프로세스의 상태는 어떨까?
    - 부모 프로세스와 같거나
    - 새로운 프로세스를 로드하거나 -> exec()



# fork()

- fork() system call creates a new process (p24)
- 부모 프로세스의 복제본을 생성.
- 부모 프로세스는...
  - 자식 프로세스의 pid 를 리턴 받거나, 실패 시 -1 을 리턴 받음.
- 자식 프로세스는...
  - 0 을 리턴받음.
- 위 성질을 이용해 분기문으로 각 프로세스가 할 일을 구분하는 기법이 자주 쓰임!



# fork()

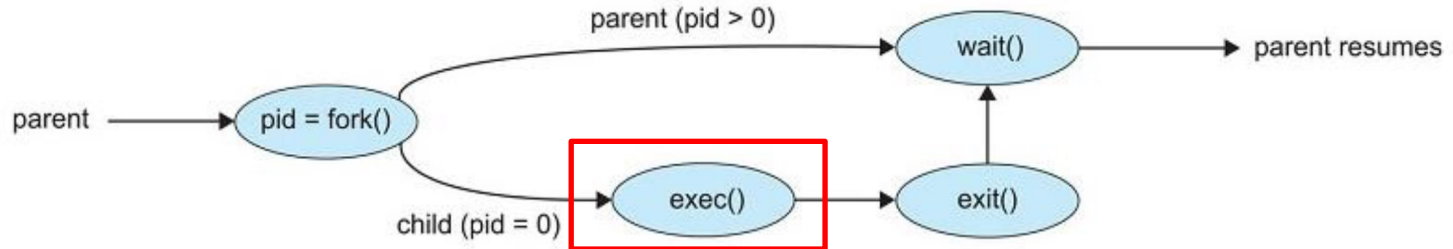
- Process-Exercise/exercise/fork.c
- gcc -o fork fork.c
- ./fork

```
ps-practice: ~/Process-Exercises/exercise |main ? :2 |  
+ ./fork1  
PID=3321 (child) idata =333 istack =666  
PID=3320 (parent) idata =111 istack =222
```

```
#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <stdlib.h>  
static int idata = 111; /*Allocated in data*/  
  
int main(int argc , char *argv[])  
{  
    int istack =222; /*Allocated in stack*/  
    pid_t childPid;  
    switch(childPid = fork())  
    {  
        case -1:  
            exit(childPid);  
        case 0:  
            idata *= 3;  
            istack *= 3;  
            break;  
        default:  
            sleep(3);  
            break;  
    }  
    printf("PID=%ld %s idata =%d istack =%d\n", (long)getpid(),  
           (childPid==0) ? "(child)" : "(parent)", idata , istack);  
    exit(childPid);  
    return 0;  
}
```

# exec()

- `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution. (p24)





# exec()

```
int execl(const char *pathname, const char *arg, ...  
/* , (char *) NULL */);
```

None of the above returns on success; all return `-1` on error

- `exec()` 함수에는 `execl()`, `execve()` 등 여러 파생 함수들이 존재하는데, 목적에 따라 파라미터만 다를 뿐 ‘기존 프로세스의 프로그램을 대치한다’는 본질은 같음!
- `pathname` = 대치할 프로그램의 `pathname`을 string 형태로 넣음.
- `args` = 대치된 프로세스에서 실행할 커맨드를 넣음. 여러 개 가능.

# exec()

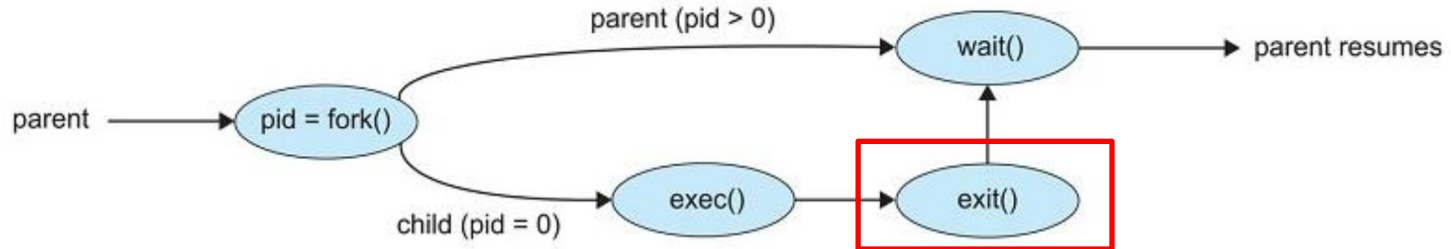
- Process-Exercise/exercise/exec.c
- gcc -o exec exec.c
- ./exec

```
os-practice: ~/Process-Exercises/exercise |main ? :2 |
+ ./exec
start
parent
child
exec exec.c fork1 fork.c
bye
```

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main() {
6      pid_t pid;
7      printf("start\n");
8      pid = fork();
9
10     if (pid > 0) {
11         printf("parent\n");
12         sleep(1);
13     }
14     else if (pid == 0) {
15         printf("child\n");
16         execl("/bin/ls", "ls", NULL);
17         printf("fail to execute ls -l \n");
18     }
19     else {
20         printf("parent fail to fork\n");
21     }
22
23     printf("bye\n");
24     return 0;
25 }
26
```

# exit()

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using `exit()`. (p27)
  - Return a status value to its waiting parent process (via `wait()` system call)
  - All the resources of process are deallocated by the OS



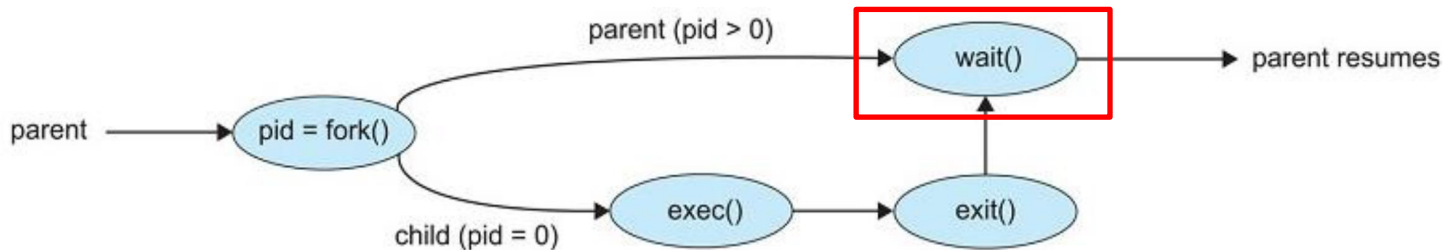
# exit()

- Status 인자를 통해서 wait 함수에게 자식 프로세스의 종료 결과 상태를 전달 가능.
- 정상적으로 종료되었으면 0, 아니면 0 외의 값으로 전달된다.

```
#include <unistd.h>  
  
void exit(int status);
```

# wait()

- Wait for termination of a child process, returning the pid(p27)
  - A process that has terminated, but whose parent has not yet called wait() is known as a zombie process.
  - If parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans



# wait()

- Status 파라미터 = exit으로 종료된 자식 프로세스의 상태를 받을 수 있음.
- 문제 없으면 자식의 프로세스 id를, 문제 있으면 -1 이 반환됨.

```
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

Returns process ID of terminated child, or -1 on error

# wait()

- Process-Exercise/exercise/wait.c
- gcc -o wait wait.c
- ./wait

```
os-practice: ~/Process-Exercises/exercise |main ? :3 |  
→ ./exitwait  
parent waiting...  
child: i'm child  
child bye  
parent bye
```

```
#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <stdlib.h>  
#include <sys/wait.h>  
  
int main(int argc , char *argv[])  
{  
    pid_t pid;  
    int status;  
  
    pid = fork();  
  
    if (pid > 0) {  
        printf("parent waiting...\n");  
        wait(&status);  
    }  
    else if(pid == 0) {  
        sleep(1);  
        printf("child: i'm child\n");  
    }  
    else {  
        printf("parent: fork failed\n");  
    }  
  
    printf((pid == 0) ? "child bye\n" : "parent bye\n");  
  
    return 0;  
}
```

# 3. Thread

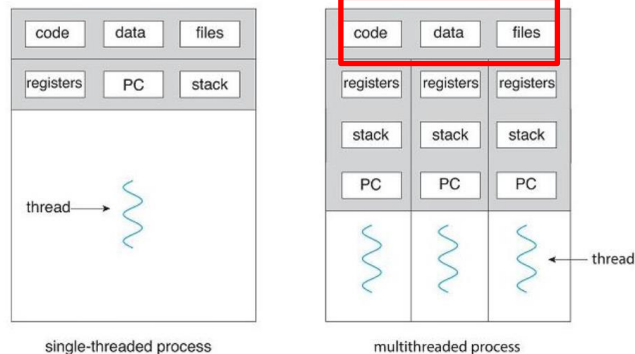


# Thread Introduction

- 본 실습은 Chapter4. Thread 슬라이드와 연결됩니다.
- 실습의 목적
  - 쓰레드를 이용하는 가장 큰 이유인 멀티코어 환경에서 parallel 한 작동을 관찰한다.
  - 쓰레드의 흐름을 제어해 본다.

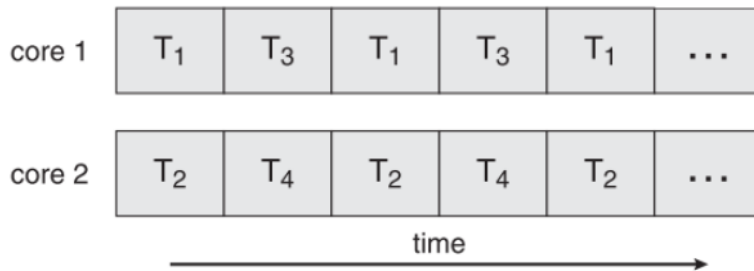
# Thread Introduction

- Thread = a light weight process
  - A basic unit of CPU utilization
  - It shares with other threads belonging to the same process its code section, data section, and other operating system resources
- 쓰레드의 이점1. 쓰레드 간의 정보 공유가 편리함.
- 프로세스만 쓴다면?
  - 프로세스 간의 정보 공유가 어려움.
  - 수업 시간에 배웠듯, 별도의 IPC 기법이 필요함.



# Thread Introduction

- Thread = a light weight process
  - A basic unit of CPU utilization
  - It shares with other threads belonging to the same process its code section, data section, and other operating system resources
- 쓰레드의 이점2. On a multiprocessor system, multiple threads can be executed in parallel(Multithreaded programming)



# Thread Creation

- `pthread_create()` : 새로운 스레드를 생성함.
- 유의사항!
  - 어떤 스레드가 cpu를 점유할 것인지에 대해서는 책임이 없다.
  - 또한 병렬 작업 수행 시, 하나의 자원에 대해 여러 스레드가 조작해야 하는 상황이 발생할 수 있다 = race condition

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start)(void *), void *arg);
```

Return 0 on success, or a positive error number on error

# Thread Termination

- `pthread_exit()` : 호출한 스레드를 종료한다.
- 첫 번째 인자로 리턴할 값을 넣을 포인터를 파라미터로 넣는다.

```
#include <pthread.h>
```

```
void pthread_exit(void *retval);
```

# create()

- Thread-Exercise/exercise/create.c
- gcc -o create create.c **-pthread**
- ./create

```
+ ./create
0 : Hello world!
6 : Hello world!
7 : Hello world!
2 : Hello world!
3 : Hello world!
4 : Hello world!
5 : Hello world!
1 : Hello world!
8 : Hello world!
10 : Hello world!
12 : Hello world!
15 : Hello world!
16 : Hello world!
14 : Hello world!
13 : Hello world!
11 : Hello world!
9 : Hello world!
17 : Hello world!
18 : Hello world!
27 : Hello world!
20 : Hello world!
21 : Hello world!
22 : Hello world!
25 : Hello world!
24 : Hello world!
23 : Hello world!
```

```
#include<stdlib.h>
#include<pthread.h>
#include<stdio.h>

#define NUM_THREADS 30

void *hello_thread(void *arg) {
    printf("%ld : Hello world!\n", (long)arg);
}

main() {
    pthread_t tid[NUM_THREADS];
    int i, status;
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&tid[i], NULL, hello_thread, (void *)i);
    }
    if (status != 0) {
        printf("Thread creation failed: %d\n", status);
        exit(1);
    }
    pthread_exit(NULL);
}
```

# Joining with a Terminated Thread

- `pthread_join()` : 특정 스레드가 종료될 때까지 기다렸다가 다음 코드를 수행한다.
- 기다린 스레드의 리턴 값을 받을 수 있다.
- 첫 번째 인자로 기다릴 스레드를, 두 번째 인자로 리턴받을 변수를 파라미터로 넣는다.

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number on error

# join()

- Thread-Exercise/ exercise/join.c
- gcc -o join join.c **-pthread**
- ./join

```
+ ./join
pid : 3984
thread 0 is created...
thread 1 is created...
thread 2 is created...
thread 3 is created...
thread 4 is created...
thread 4 terminated...
thread 3 terminated...
thread 1 terminated...
thread 0 terminated...
thread 0 is joined...
thread 0 return value: 5
thread 1 is joined...
thread 1 return value: 5
thread 2 terminated...
thread 2 is joined...
thread 2 return value: 5
thread 3 is joined...
thread 3 return value: 5
thread 4 is joined...
thread 4 return value: 5
```

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define NUM_THREADS 5
pthread_t tid[NUM_THREADS];

void *hello_thread(void *arg) {
    int i = 0;
    while(1) {
        if(i==5) {
            break;
        }
        i++;
        sleep(1);
    }
    printf("thread %ld terminated...\n", (intptr_t) arg);
    pthread_exit( (void*)(intptr_t)i);
}

void main() {
    long int i, status;
    printf("pid : %d\n", getpid());

    for(i=0; i<NUM_THREADS; i++) {
        status = pthread_create(&tid[i], NULL, hello_thread, (void *) (intptr_t)i);
        if (status != 0) {
            fprintf(stderr, "create thread error: %s\n", strerror(status));
        }
        printf("thread %ld is created...\n", i);
    }

    for(i=0; i<NUM_THREADS; i++) {
        void *ret;
        status = pthread_join(tid[i], &ret);
        printf("thread %ld is joined...\n", i);
        printf("thread %ld return value: %ld\n", i, (intptr_t)ret);
    }
}
```



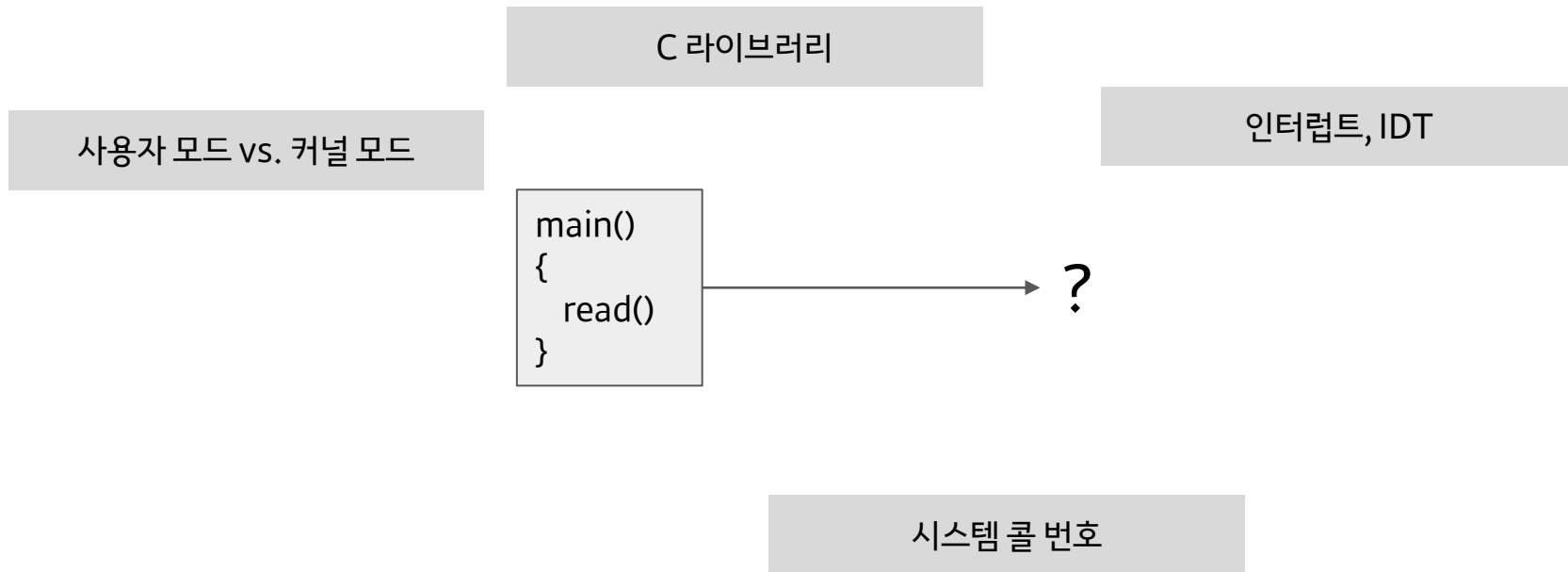
# 과제 설명

# System Call 과제

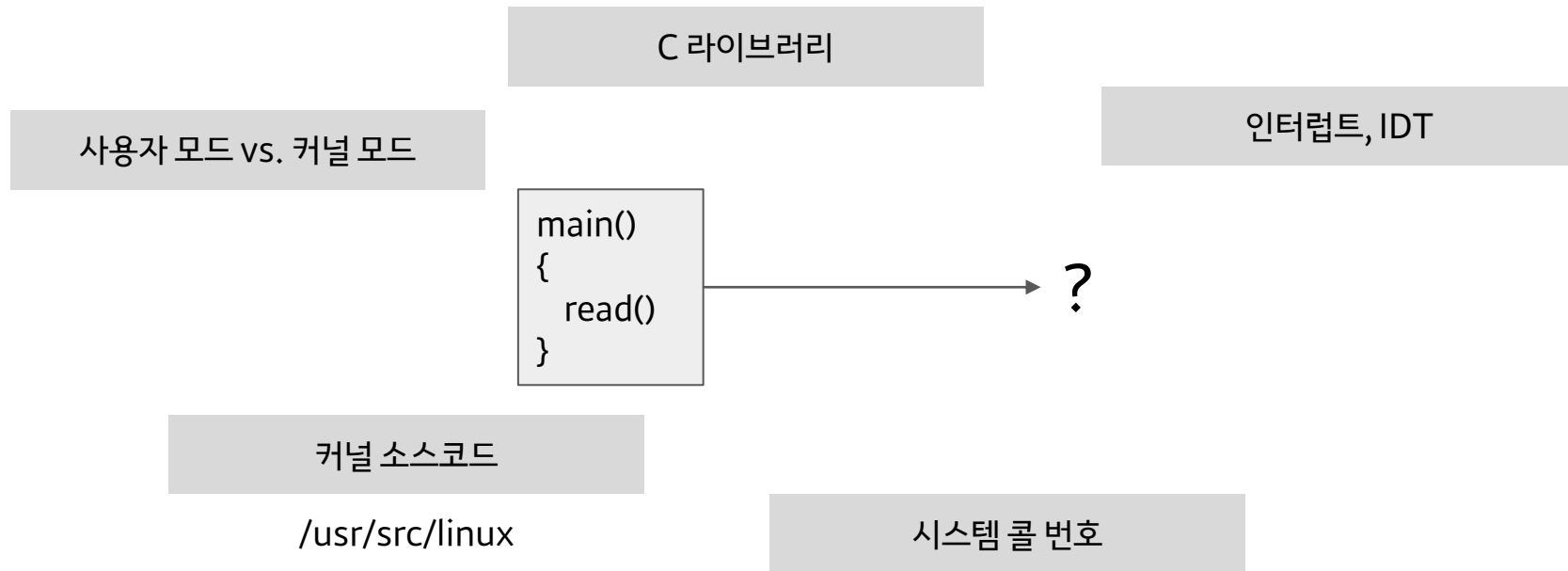
## 과제1-1. 시스템 콜 과정 이해하기



## 과제1-1. 시스템 콜 과정 이해하기



## 과제1-1. 시스템 콜 과정 이해하기



## 과제1-2. 새로운 시스템 콜 추가하기

```
printk("Hello World!\n");  
printk("%d + %d = %d\n", a, b, a+b);
```

“자신의 학번을 printk 함수를 사용해 출력하는 시스템 콜  
그 결과를 dmesg로 확인하기”

시스템 콜 이름: `print_student_id`

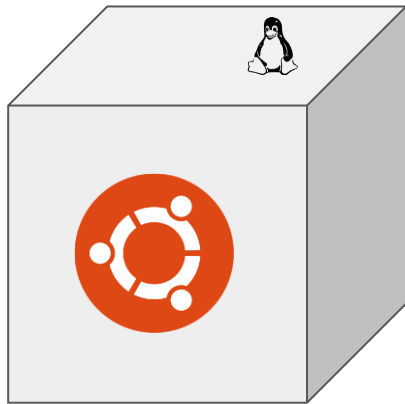
호출되는 함수 이름: `sys_print_student_id(void)`

## 과제1-2. 새로운 시스템 콜 추가하기

dmesg로 결과 확인하기

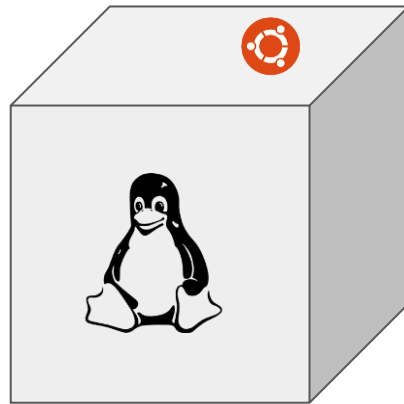
```
[ 90.012650] My student id is 2020021465
```

# 새로운 커널 만들기



Ubuntu 18.04 Server (Linux 4.15.0)

시스템 콜을 추가한 Linux 4.15.10



Linux 4.15.10 + 새로운 시스템 콜



# 시스템 콜 함수를 구현하고 난 후에

vim /usr/src/linux/kernel/Makefile

ucount.o 뒤에 new\_syscall.o 추가

cd /usr/src/linux

sudo make -j 4

sudo make modules\_install

sudo make install

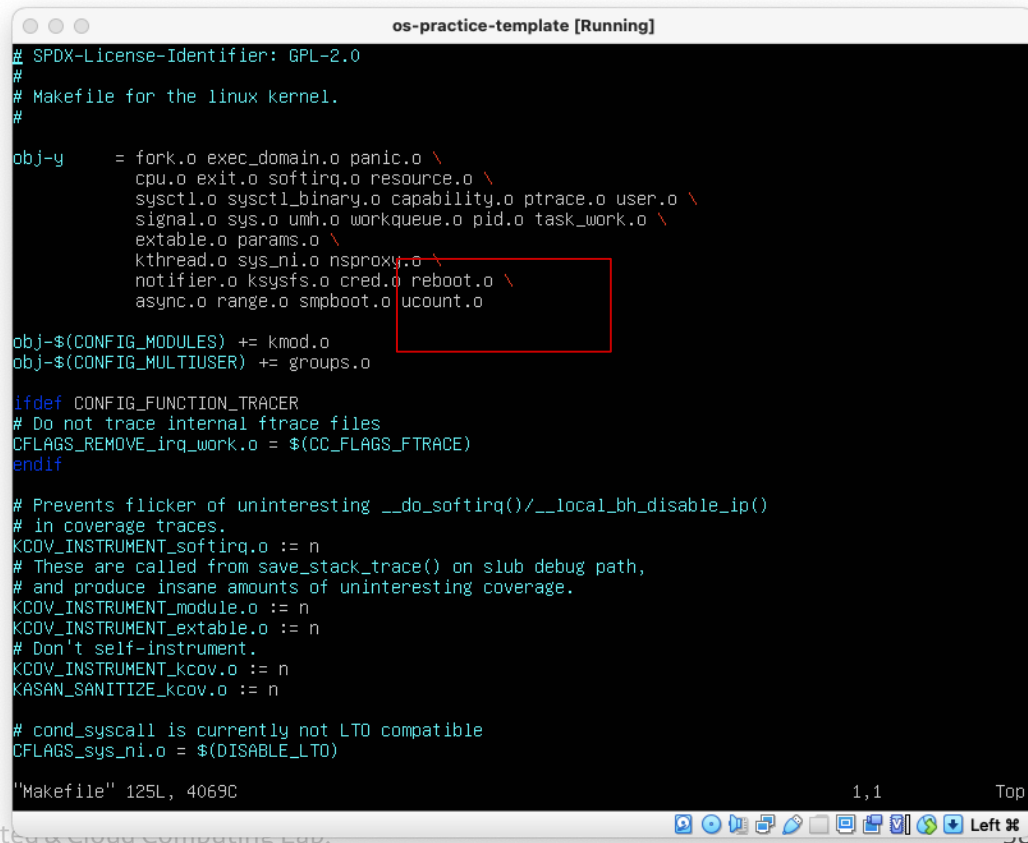
sudo reboot

cd ~

gcc -o assignment assignment.c

./assignment

dmesg



```
os-practice-template [Running]
# SPDX-License-Identifier: GPL-2.0
#
# Makefile for the linux kernel.
#
obj-y      = fork.o exec_domain.o panic.o \
             cpu.o exit.o softirq.o resource.o \
             sysctl.o sysctl_binary.o capability.o ptrace.o user.o \
             signal.o sys.o umh.o workqueue.o pid.o task_work.o \
             extable.o params.o \
             kthread.o sys_ni.o nsproxy.o \
             notifier.o ksysfs.o cred.o reboot.o \
             async.o range.o smpboot.o ucount.o

obj-$(CONFIG_MODULES) += kmod.o
obj-$(CONFIG_MULTIUSER) += groups.o

ifdef CONFIG_FUNCTION_TRACER
# Do not trace internal ftrace files
CFLAGS_REMOVE_irq_work.o = $(CC_FLAGS_FTRACE)
endif

# Prevents flicker of uninteresting __do_softirq()/__local_bh_disable_ip()
# in coverage traces.
KCOV_INSTRUMENT_softirq.o := n
# These are called from save_stack_trace() on slub debug path,
# and produce insane amounts of uninteresting coverage.
KCOV_INSTRUMENT_module.o := n
KCOV_INSTRUMENT_extable.o := n
# Don't self-instrument.
KCOV_INSTRUMENT_kcov.o := n
KASAN_SANITIZE_kcov.o := n

# cond_syscall is currently not LTO compatible
CFLAGS_sys_ni.o = $(DISABLE_LTO)

"Makefile" 125L, 4069C
1,1 Top
```

# Process, Thread 과제

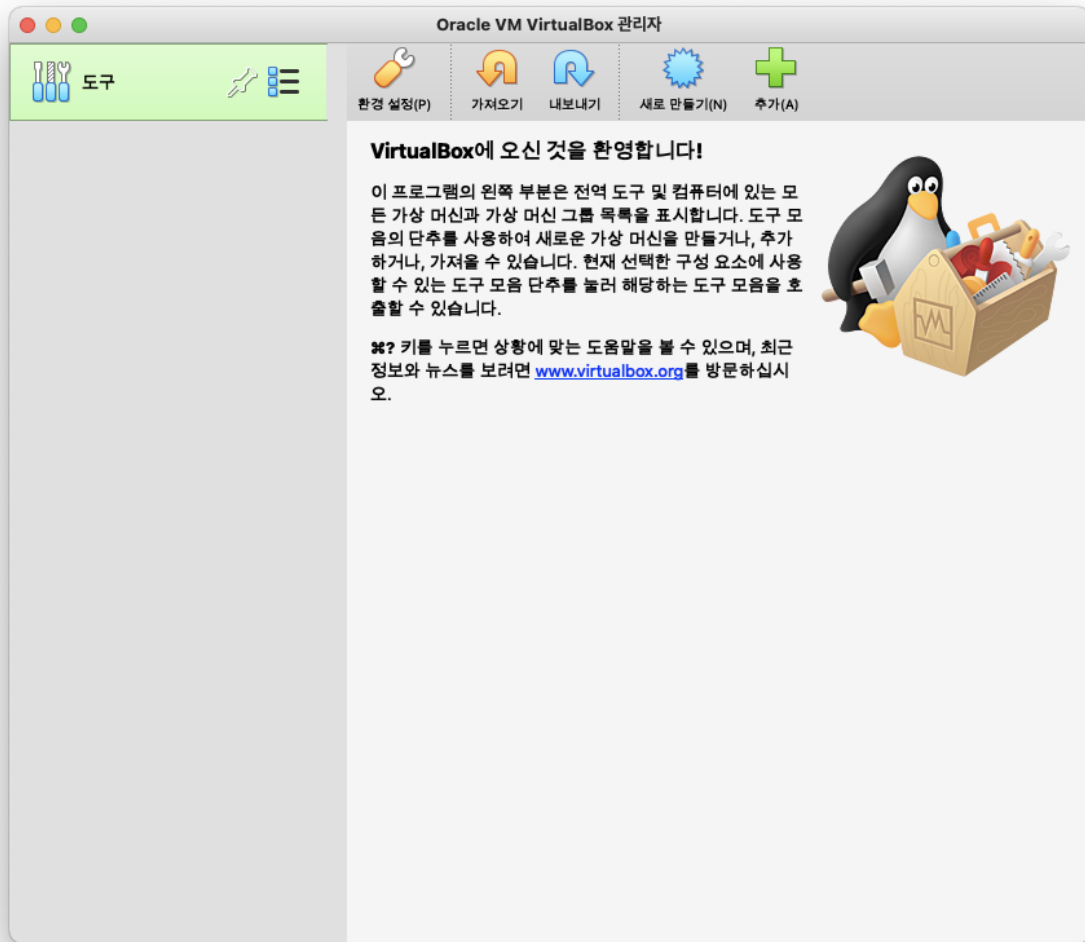
## 과제2. 프로세스, 스레드 퀴즈 풀기

- 제공하는 스켈레톤 코드에 <?문제번호/> 형태로 빈칸이 뚫려 있다.
- 또한 코드의 맨 밑을 확인하면 문제마다 기대되는 아웃풋이 주석으로 표현되어 있다.
- 각 문제마다 빈칸에 들어갈 코드를 작성하고, 완성한 코드와 출력 결과 스크린샷을 찍어서 제출한다.

# 부록1. 실습 환경 virtualbox 설정

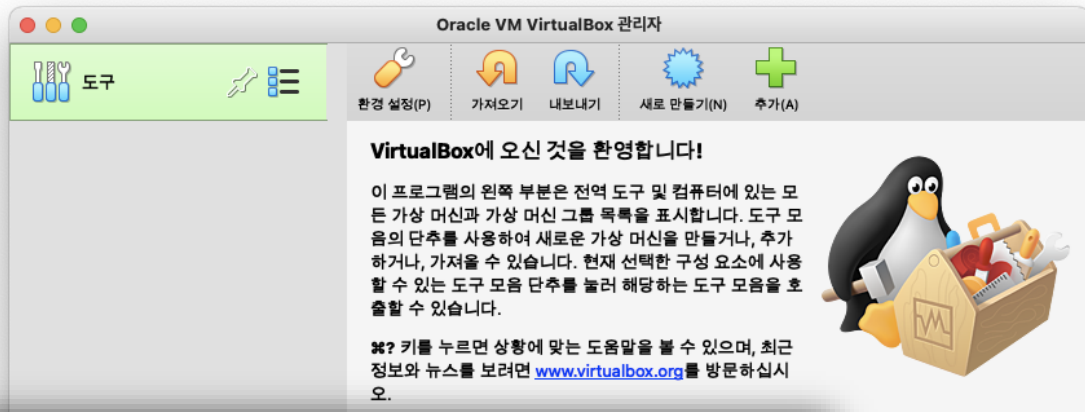
# 가상머신 실행시키기

1. Oracle VM VirtualBox [다운로드](#)
2. os-practice-template.ova 다운로드
3. 가져오기



# 가상머신 실행시키기

1. Oracle VM VirtualBox 다운로드
2. os-practice-template.ova 다운로드
3. 가져오기



## 가져올 가상 시스템

가상 시스템을 가져올 원본을 선택하십시오. 로컬 파일 시스템의 OVF 파일을 가져오거나 클라우드 서비스 공급자에서 클라우드 가상 머신을 가져올 수 있습니다.

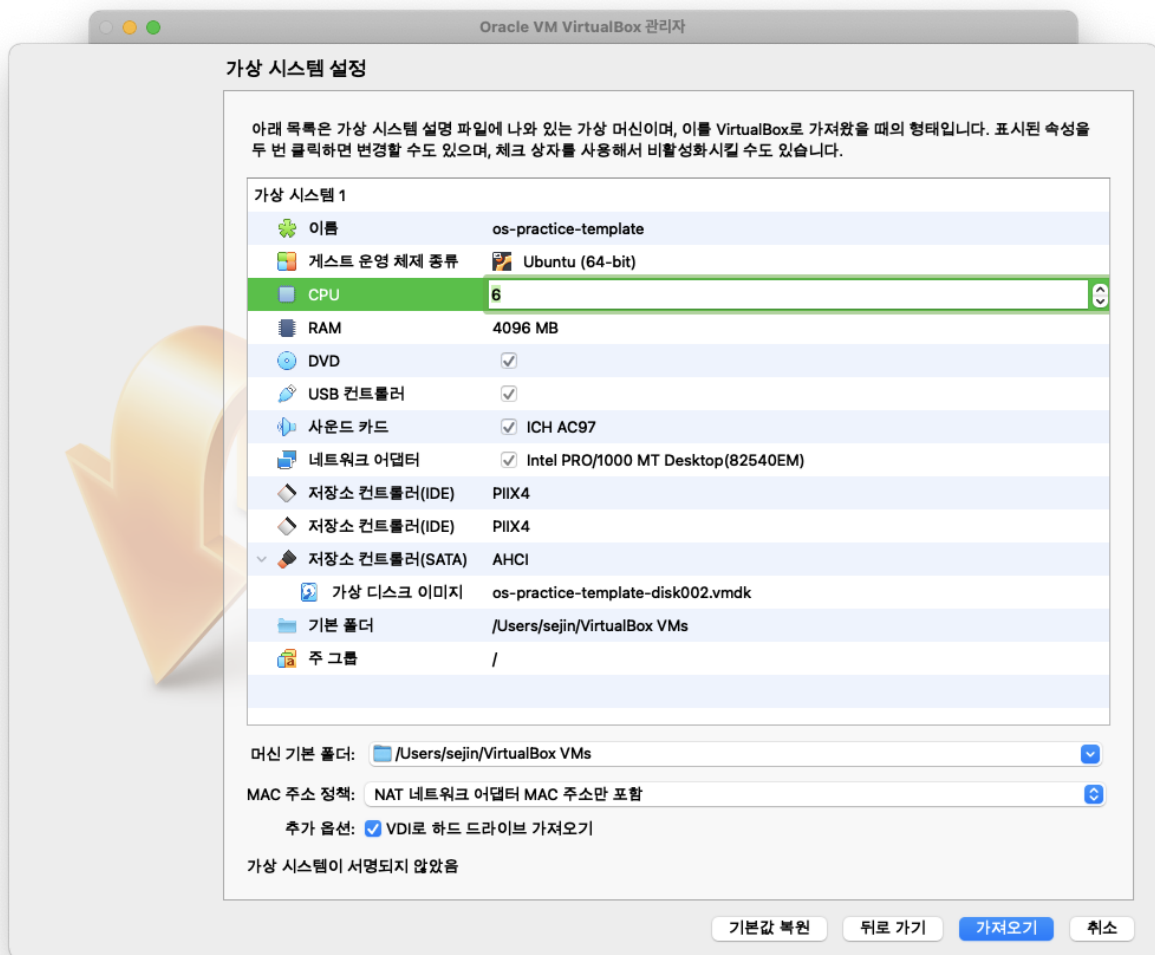
원본:

가상 시스템을 가져올 파일을 선택하십시오. VirtualBox는 열린 가상화 형식(OVF)으로 저장된 가상 시스템을 가져올 수 있습니다. 계속 진행하려면 가져올 파일을 선택하십시오.

파일:

# 가상머신 불러오기

호스트 OS에 맞게 CPU 조절 가능  
CPU를 많이 할당하면 컴파일 속도↑



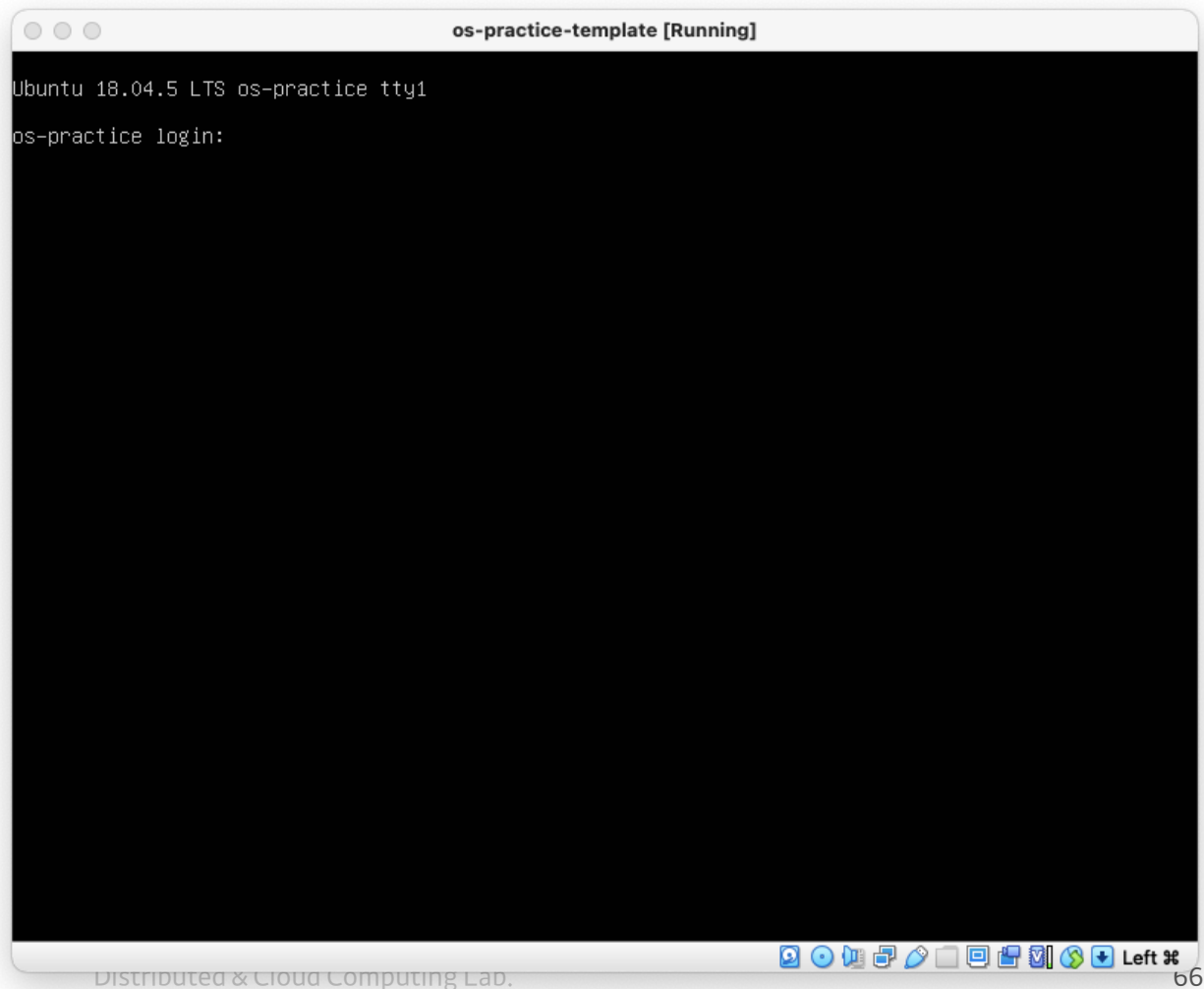
# 가상머신 시작하기





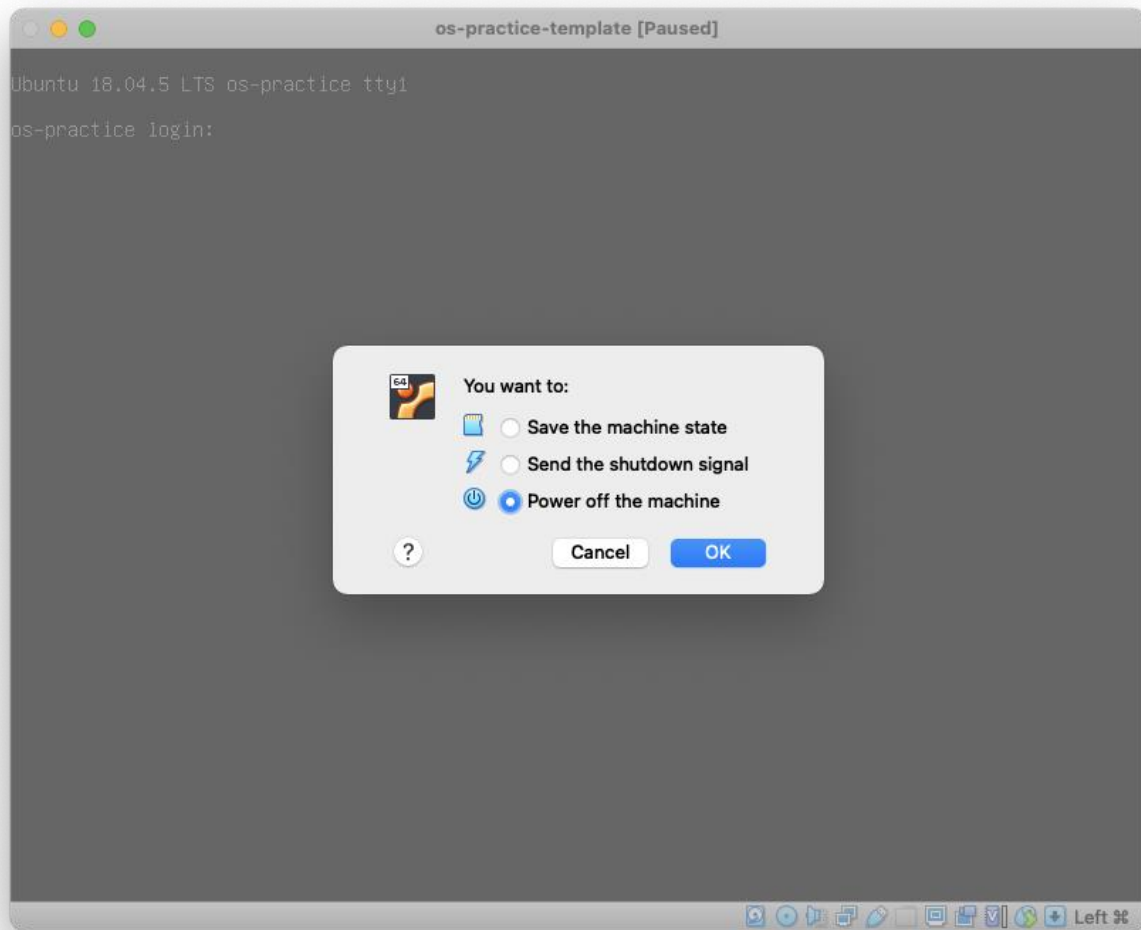
# 가상머신 시작하기

ID: guest  
PW: 1234



# 가상머신 종료하기

특수한 상황이 아니라면  
Power off the machine



감사합니다