운영체제론 실습 3주차

정보보호연구실 @ 한양대학교

커널 (Kernel)

시스템을 통제하는 운영체제의 핵심

- **보안**
  - 컴퓨터 하드웨어 및 프로세스의 보안을 담당함
- **자원 관리**
  - **한정**된 시스템 자원을 관리하여 프로그램의 원활한 실행을 가능하게 함
- **추상화**
  - 하드웨어에 **직접 접근하지 않고** 추상화된 인터페이스를 하드웨어에 제공

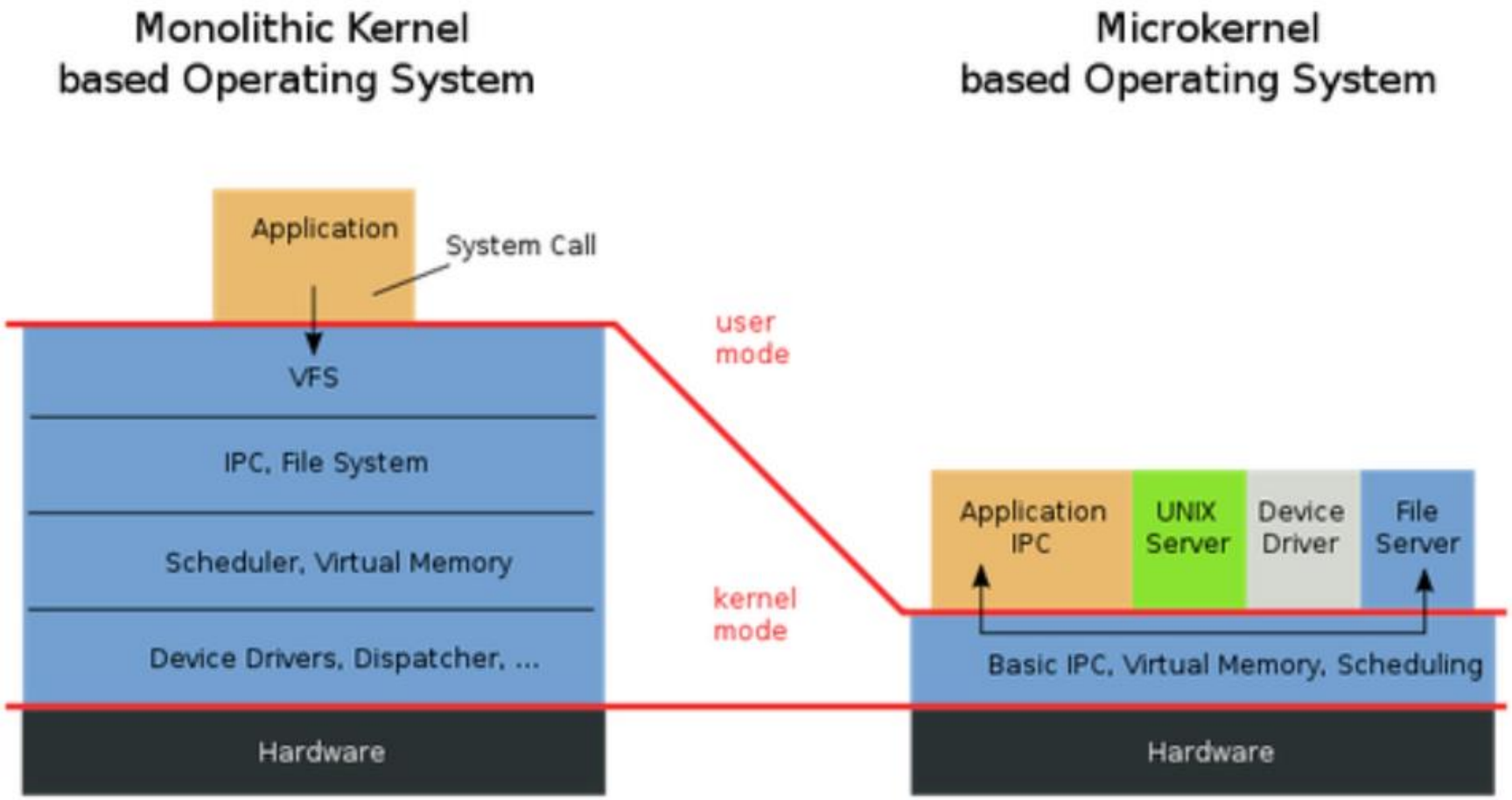
커널 (Kernel)

커널과 사용자 (User) 모드

- **권한의 차이**
  - 커널 모드: **모든 자원** (드라이버, 메모리, CPU 등) 에 접근, 명령 가능
  - 사용자 모드: 접근 가능 영역이 **제한**됨 - 자원 침범 방지
  - 리눅스 커널은 **커널 모드에 어플리케이션을 제외한 모든 시스템 기능**이 몰려 있는 **모놀리식 (monolithic)** 커널

- **모드의 전환**
  - 프로세스의 실행 과정에서 **커널 영역의 기능을 사용**하는 경우가 발생
  - 이 과정에서 호출되는 것이 **시스템 콜 (system call)**

커널 (Kernel)


Monolithic Kernel
based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user
mode
kernel
mode
Microkernel
based Operating System
Application
IPC
UNIX
Server
Device
Driver
File
Server
Basic IPC, Virtual Memory, Scheduling
Hardware

시스템 콜 (System Call)

커널에 접근하기 위한 인터페이스 (접점)

- **사용자 모드에서 커널의 기능을 사용할 수 있도록 함**
  - 프로세스 제어, 파일 조작, 장치 관리, 정보 유지, 통신
- **시스템 콜 호출 시 사용자 → 커널 모드 변환**
- **호출이 처리되면 다시 커널 → 사용자 모드로 변환**
  - 결과값은 시스템 콜의 반환 값 (정수) 으로 전달

API (Appication Programming Interface)

POSIX API

- **UNIX 운영체제에 기반을 둔 표준 인터페이스**
  - 요청된 서비스에 대응하는 함수가 지정되어 있음
  - 표준을 두어 개발자들이 어플리케이션을 각기 다른 시스템에 이식하기 용이하게 함
  - open, close, read, write 함수 등

- **리눅스의 POSIX API를 준수하는 라이브러리 내의 함수들이 시스템 콜 함수를 호출하여 사용**

쉘(Shell) vs. API vs. 시스템 콜

- **사용자 인터페이스 vs. 응용 프로그램 인터페이스**
  - 쉘: 사용자 인터페이스 – 사용자와 운영체제간의 채널
  - API: 응용 프로그램 인터페이스 – 프로세스와 운영체제간의 채널
  - 시스템 콜: 운영체제 기능을 사용하기 위한 명령/함수

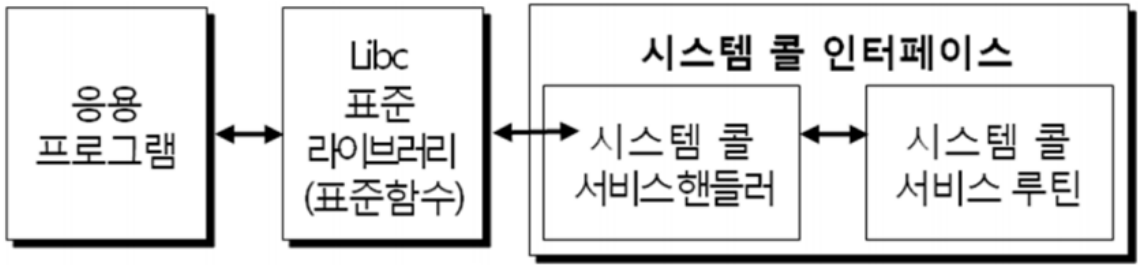
시스템 콜 동작원리

시스템 콜은 어떻게 실행될까?

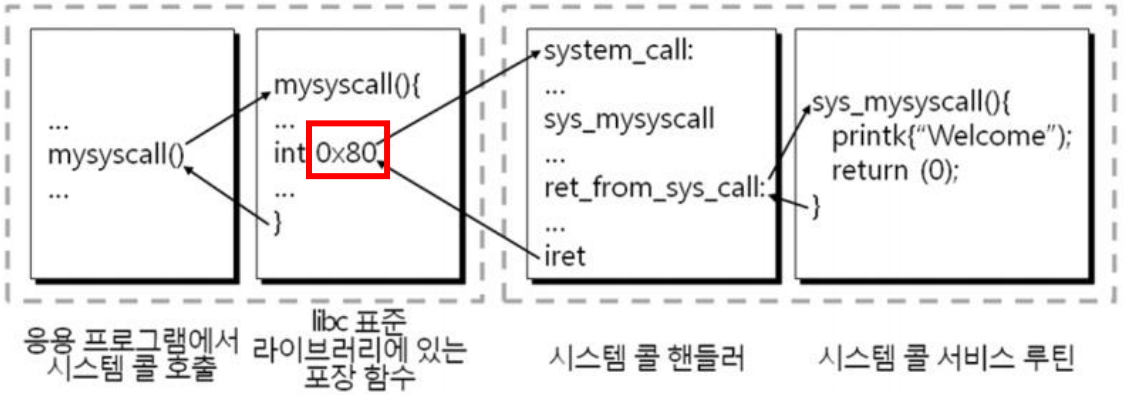
- 응용 프로그램에서 API 함수 호출 (GNU libc 라이브러리)
- API 함수 내부의 어셈블리 코드 실행 – 시스템 콜 발생
- 커널 모드 전환 – 시스템 콜 서비스 핸들러
  - 요청받은 시스템 콜에 대응하는 서비스 루틴 확인
- 서비스 루틴 호출 – 사용자 모드 전환

시스템 콜 동작개념도

❑ POSIX API에서 시스템 콜 사용의 개념도



❑ 시스템 콜 호출 시 내부동작 개념도

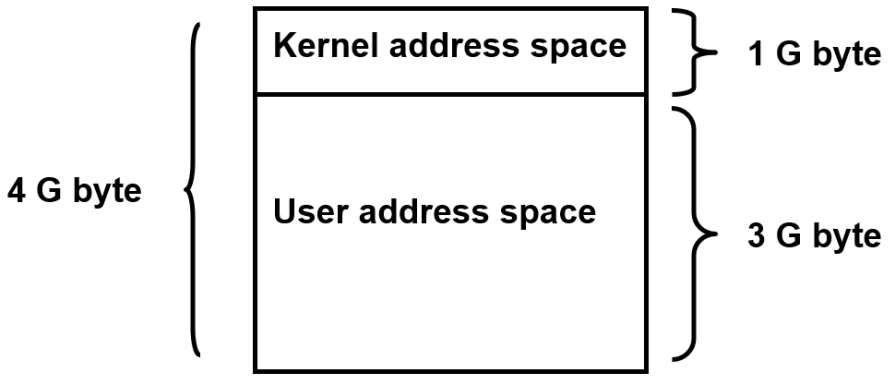


커널 프로그래밍

리눅스 커널의 핵심 기능 추가하기

- **커널 모듈을 구현하여 기능을 추가할 수 있음**

- **응용 프로그램 vs. 커널 프로그램**
  - 서로 다른 메모리 주소 영역에 프로그램 코드가 존재
  - 커널 프로그램은 **일반 라이브러리 사용 불가**
  - **작은** 스택 크기 – 메모리 사용에 주의!



커널 프로그래밍 주의 사항

- **이름 충돌 문제 (namespace pollution)**
  - 응용 프로그램: **프로그램 내부**에서만 함수&변수 이름 구별 필요
  - 커널 프로그램: 모듈만이 아니라 **커널 전체**에서 함수&변수 이름 구별 필요

- **모든 것이 가능한 커널**
  - **커널에서의 에러**는 시스템에 **치명적**인 결과를 발생시킴
  - **모든 에러 코드**를 꼼꼼히 검사하고 처리해야 함

프로그래밍 과정

- 새 시스템 콜 함수 작성

- 커널 Makefile 업데이트
- 시스템 콜 헤더 파일 업데이트
- 시스템 콜 테이블 업데이트

- 커널 컴파일 (일부)

- 구현된 시스템 콜 호출 및 확인

준비과정

설치한 커널 소스 코드 이동

- **소스 코드를 /usr/src (우분투 소스코드 디렉토리) 로 이동**
  - 설치한 커널 디렉토리 (linux-x.x.x) 의 상위 폴더로 이동
  - $ sudo mv linux-$(uname –r) /usr/src/

- **source 와 build 파일을 변경된 경로로 링크**
  - $ cd /usr/src/linux-$(uname -r) ← 커널 디렉토리로 이동
  - $ sudo ln –Tfs /usr/src/linux-$(uname -r) /lib/modules/$(uname -r)/source
  - $ sudo ln –Tfs /usr/src/linux-$(uname -r) /lib/modules/$(uname -r)/build

시스템 콜 함수 작성

작업 위치: /usr/src/linux-$(uname -r)

- **hello 디렉토리 만들기**
  - $ sudo mkdir hello
- **hello.c 파일 생성**
  - $ cd hello ← hello 디렉토리로 이동
  - $ sudo vi hello.c ← vim 에디터 사용자
  - $ sudo nano hello.c ← nano 에디터 사용자

시스템 콜 함수 작성

작업 위치: /usr/src/linux-$(uname -r)/hello

- **sys_hello 함수 작성하기**
  - asmlinkage: 어셈블리 단계에서 이 함수를 지칭할 keyword 지정 명령어

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

asmlinkage long sys_hello(void)
{
        printk("SYSCALL HELLO IS CALLED!₩n");
        return 0;
}
```

시스템 콜 함수 작성

작업 위치: /usr/src/linux-$(uname -r)/hello

- **hello.c 파일의 Makefile 작성**
  - $ sudo *vi/nano* Makefile ← 철자주의!

```
obj-y := hello.o
```

시스템 콜 추가

작업 위치: /usr/src/linux-$(uname -r)

- **Makefile 업데이트**
  - $ sudo *vi/nano* Makefile ← 철자주의!

- **Makefile 에서 core–y 부분에 hello 디렉토리 추가**

```
export MODULES_NSDEPS := $(extmod-prefix)modules.nsdeps

ifeq ($(KBUILD_EXTMOD),)
core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ hello/

vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
                     $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
```

시스템 콜 추가

작업 위치: /usr/src/linux-$(uname -r)

- 시스템 콜 헤더 파일 업데이트
  - $ cd include/linux/
  - $ sudo *vi/nano* syscalls.h

- #endif /*CONFIG_(...)_WRAPPER*/ 상단에 새 시스템 콜 함수 추가

```
 * not implemented -- see kernel/sys_ni.c
 */
asmlinkage long sys_ni_syscall(void);

asmlinkage long sys_hello(void);

#endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
```

시스템 콜 추가

작업 위치: /usr/src/linux-$(uname -r)

- 시스템 콜 테이블 업데이트
  - $ cd arch/x86/entry/syscalls
  - $ sudo *vi/nano* syscall_64.tbl

- 새 시스템 콜 함수를 테이블 맨 밑 항목에 추가 ← 해당 숫자 기억!

```
433     common  fspick                  __x64_sys_fspick
434     common  pidfd_open              __x64_sys_pidfd_open
435     common  clone3                  __x64_sys_clone3/ptregs
436     common  hello                   sys_hello

# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
```

커널 컴파일

작업 위치: /usr/src/linux-$(uname -r)

- 커널 일부의 이미지 생성
  - $ sudo make bzImage –j $(nproc)
  - $ sudo cp arch/x86/boot/bzImage /boot/vmlinuz-$(uname -r)
- 가상머신 재부팅
  - $ sudo reboot

추가된 시스템 콜 확인

작업 위치: 실습 작업 디렉토리 (사용자 모드)

- 시스템 콜 호출 프로그램 작성
  - $ *vi/nano* test_sys_hello.c

```
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdio.h>
int main( )                          hello의 시스템 콜 테이블 번호
{                                          ↓
        long int i = syscall(436);
        printf("SYSCALL::SYS_HELLO::RETVAL=%ld₩n", i);
        return 0;
}
```

추가된 시스템 콜 확인

작업 위치: 실습 작업 디렉토리 (사용자 모드)

- **호출 프로그램 컴파일 및 실행**
  - $ gcc test_sys_hello.c –o test_sys_hello
  - $ ./test_sys_hello
- **출력 내용 확인**
  - SYSCALL::SYS_HELLO::RETVAL=**0**
  - 0이 나올 경우 시스템 콜이 정상적으로 작동 중

추가된 시스템 콜 확인

작업 위치: 실습 작업 디렉토리 (사용자 모드)

- **커널에 출력된 로그 확인하기**
  - $ dmesg
  - SYSCALL HELLO IS CALLED! 확인