Operating Systems

3. OS Design and Structures

Hyunchan, Park

http://oslab.chonbuk.ac.kr

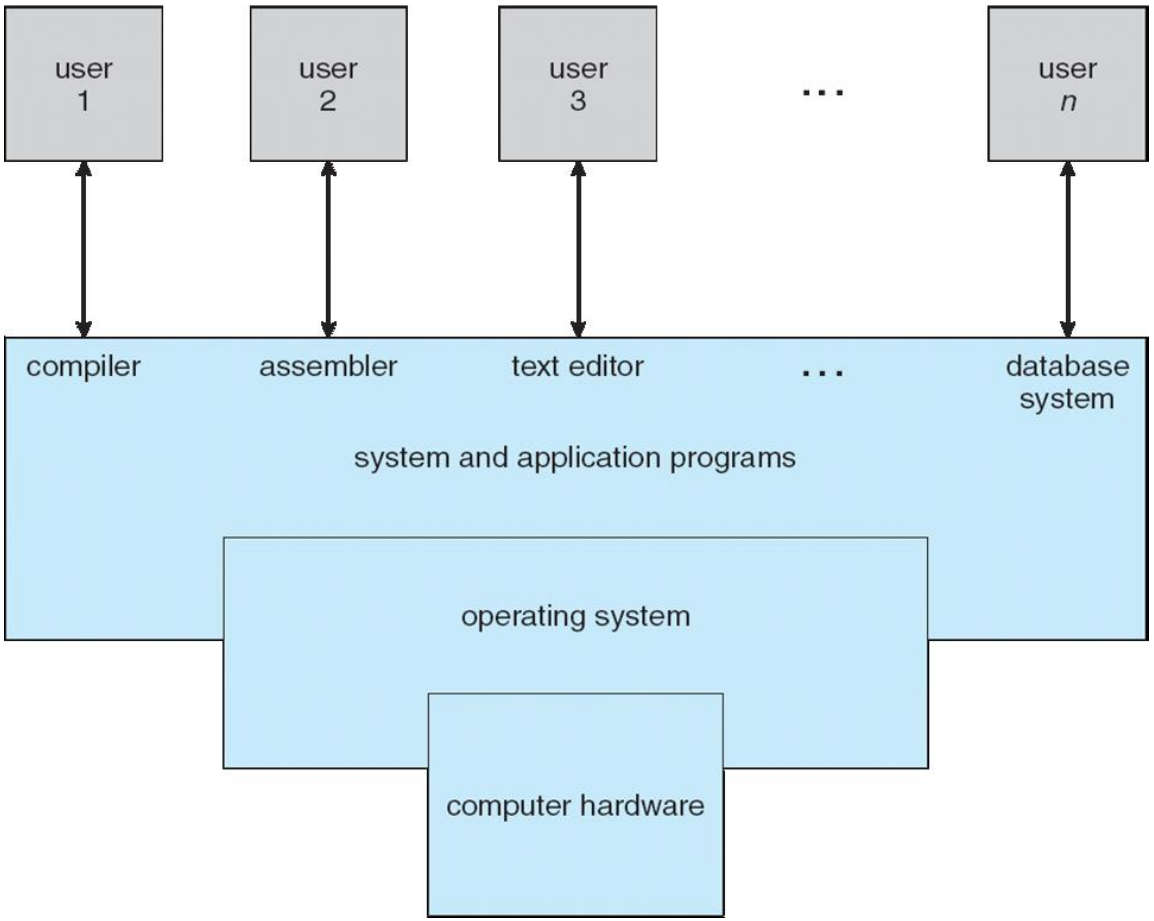
Division of Computer Science and Engineering

Chonbuk National University

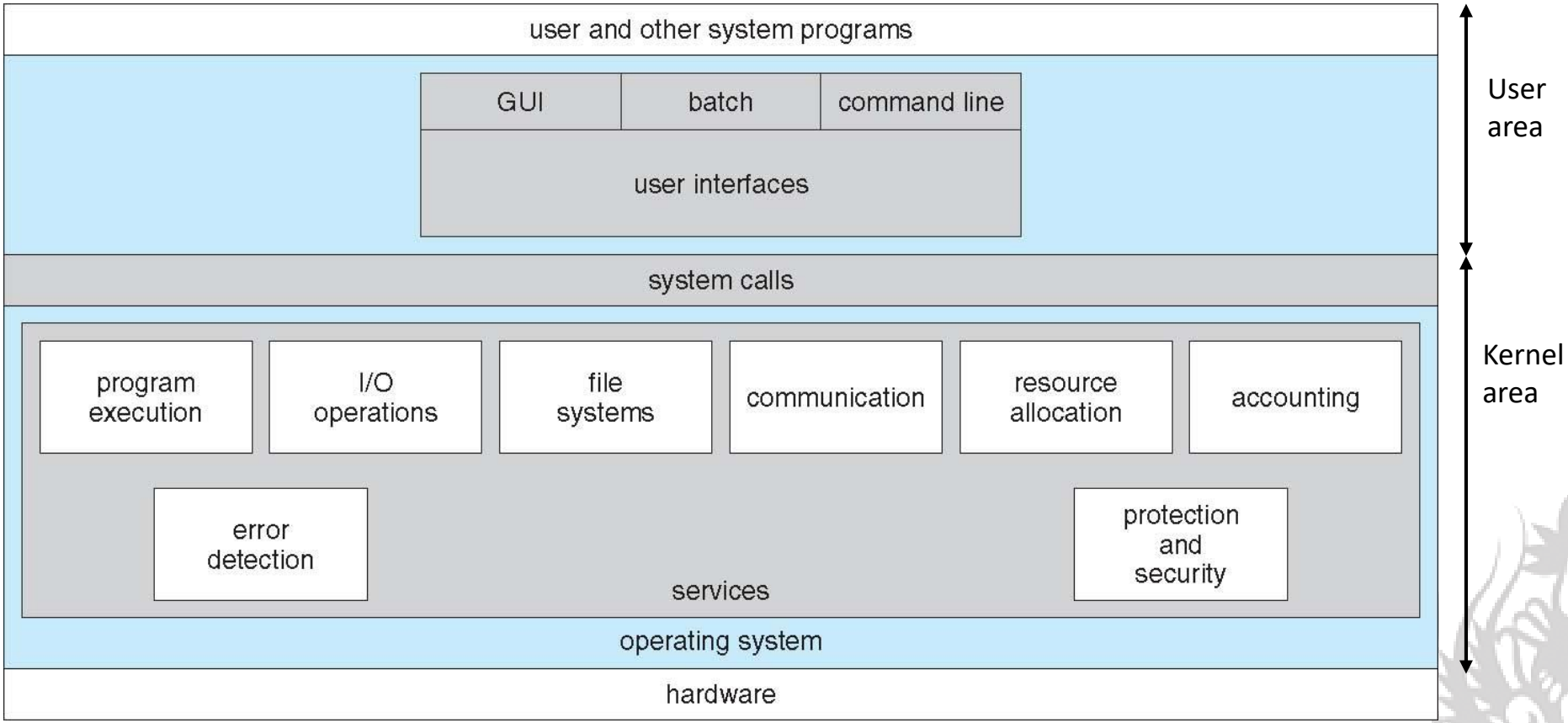
Contents

- Design considerations
  - Design goals: Properties
  - OS design principle: Mechanism and Policy
  - Methods for operating system design: Layering and Modularity
- Kernel structures
  - System call
  - Monolithic and Micro kernel
  - Hypervisor
- System programs

컴퓨터 시스템의 네 가지 요소



A View of Operating System Services



Design considerations

Design considerations

- 운영체제는 규모가 매우 크고 복잡한 소프트웨어
  - 설계 시 소프트웨어의 **"구조"**를 신중히 고려해야 함

- 좋은 설계를 통해 쉬워지는 것들.
  - 개발(develop)
  - 수정 및 디버깅(modify and debug)
  - 유지 보수(Maintain)
  - 확장(Extend)

- 디자인 목표 중에 좋은 것이란?
  - 설계하고자 하는 시스템의 목적과 관계가 있음

Design goals: Properties

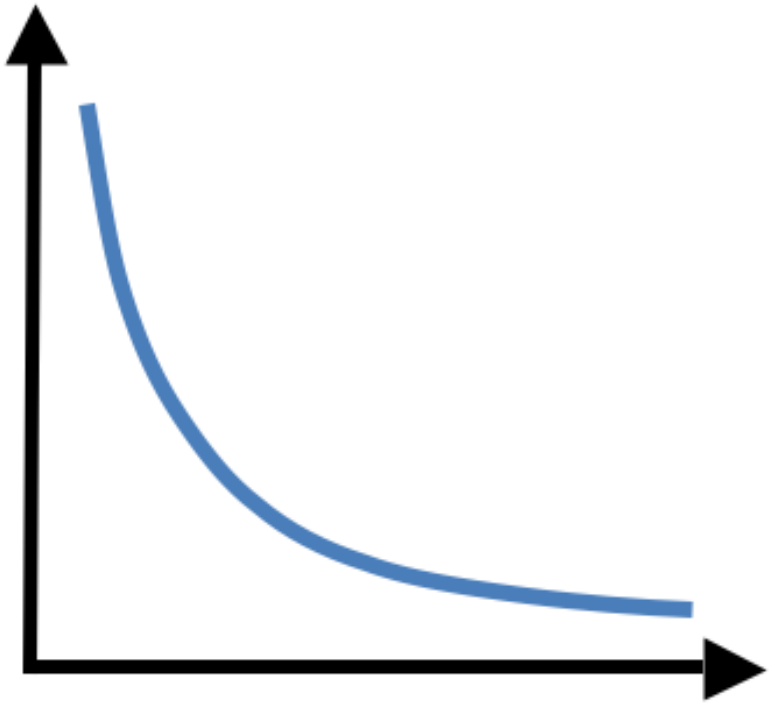
- Fairness
- Real-time
- High performance
- Scalability/Extensibility
- Stability/Reliability/Robustness
- Security/Integrity
- Usability
- Compatibility
- Energy consumption
- And so on …

General and Special Purpose OS

- General purpose OS
  - 일반적 용도의 OS
    - 윈도우, 리눅스, 안드로이드, iOS 등
  - 여러 특성들을 동시에 골고루 만족시켜주어야 함
    - 사용 환경, 사용자 어플리케이션 등을 특정할 수 없음
- Special purpose OS
  - 특수한 용도의 OS
    - 군용 장치, 발전소, IoT 장치, 센서 등
    - Mission critical system에 주로 사용
  - 특별한 요구 특성이 존재하고, 이를 만족시키는 것이 설계 목표
    - 발전소: Real-time, integrity
    - 군용 무기: Robustness, usability, stability

Trade-off between properties

- Trade-off
  - (동시에 달성할 수 없는 몇 개 조건을 취사 선택하여) 균형을 취하는 일
  - 예) 성능-공평성: 얼마나 자주 OS가 제어할 것인가? 1ms? 100ms?
    - 자주 제어하면? 성능 ↓ 공평성 ↑



trade-off

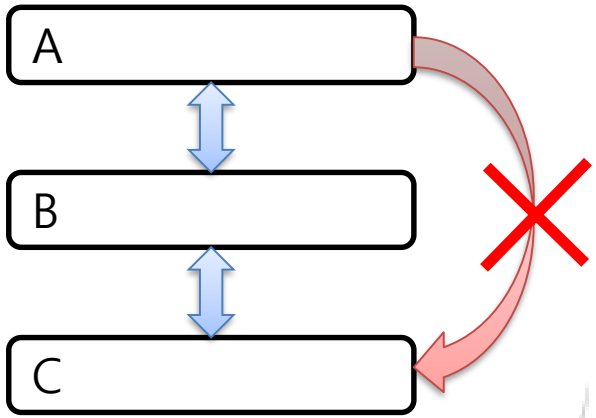
1.(미) (특히 타협을 가져오기 위한) 거래 (bargain)
2.교환 3.(거래에 의한) 협정
4.결정(arrangement) 5.이율 배반성

OS design principle

- Policy
  - Decide what to be done
    - 무엇이 되게 할 것인가?
  - Supposed to be higher level, and use mechanism
    - E.g. Complete fair distribution of CPU, real-time support for a specific task
- Mechanism
  - Determine how to do something
    - 무엇을 어떻게 할 것인가?
  - E.g. The concrete algorithms, data structures
- Policy를 실제로 달성하는 방식이 Mechanism
  - 설계를 위한 policy, 구현을 위한 mechanism

Layering

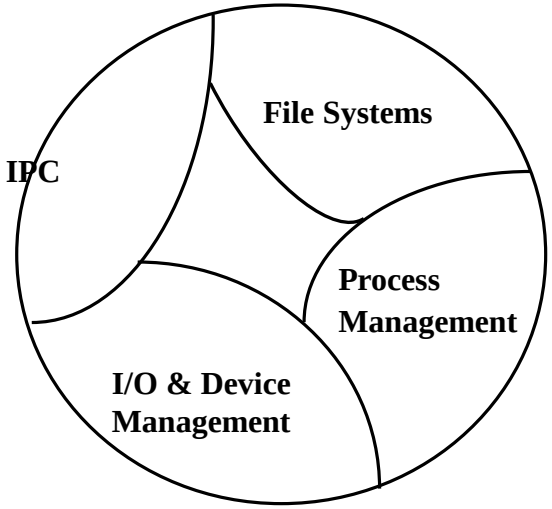
- OS의 복잡도를 낮추기 위한 방안
- 계층 별로 명확한(well-defined) 인터페이스 및 기능을 정의함
- 하나의 layer는 인접한 layer와만 통신
  - 위, 아래에 인접한 layer만과 통신하며, 2단계 이상 건너뛴 layer와 직접적으로 통신하지 않음



- 설계의 복잡도를 낮출 수는 있으나, 그로 인해서 overhead가 발생함
  - E.g. The 7-layers of the OSI model

Layering vs. Modularity

- Layering의 장점
  - Layer의 수정이 다른 layer와 독립적임

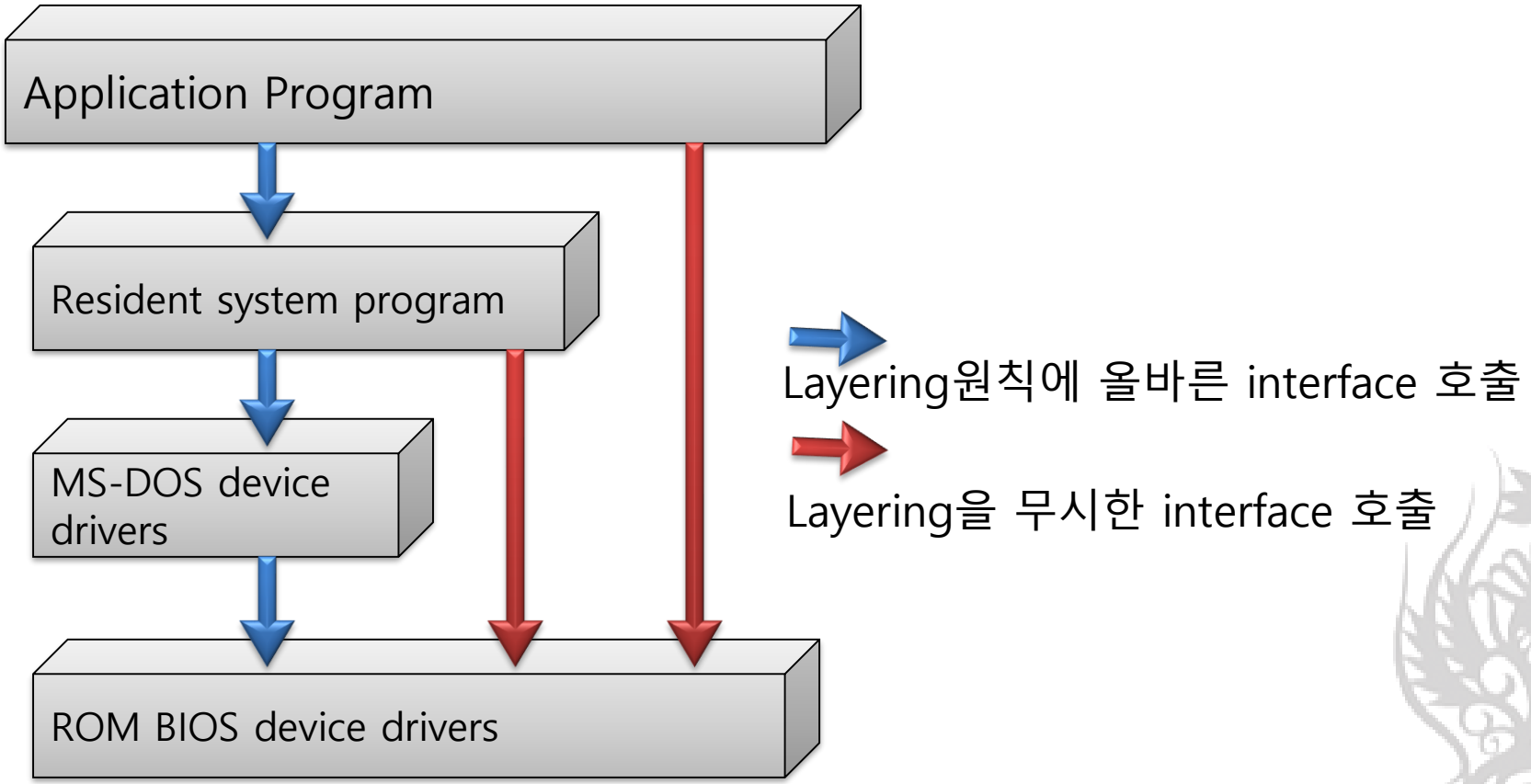


< Modularity >

< Layering >

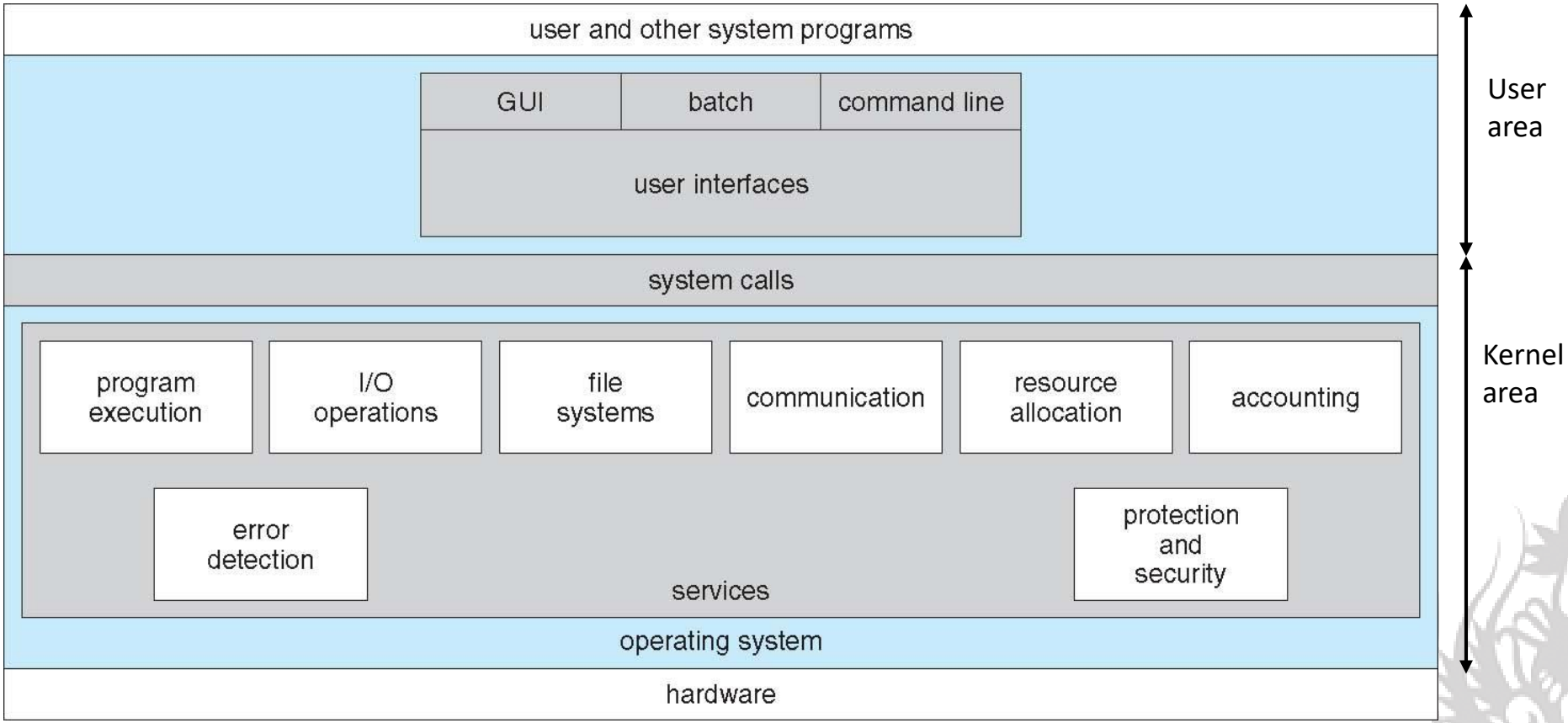
불완전한 Layering

< Example : MS-DOS. Interface is not well separated >



Kernel structures

A View of Operating System Services



System Call

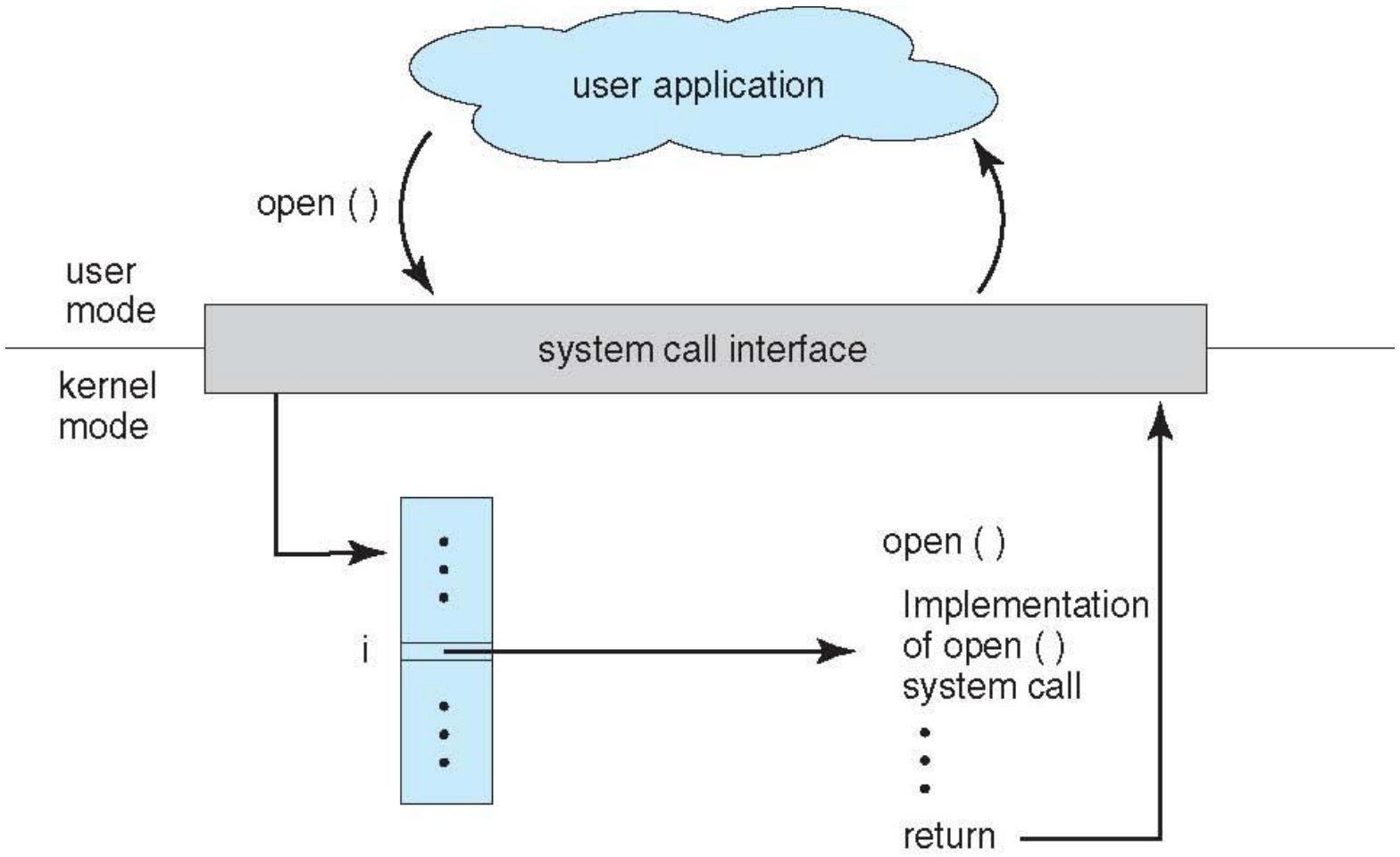
- User mode에서 kernel mode로 진입하기 위한 통로
  - 커널에서 제공하는 protected 서비스를 이용하기 위하여 필요
  - 시스템 콜의 예
    - Open() : a file or device, Write() : to file or device, Shm()
- 유저 프로그램은 보통 직접 시스템 콜을 이용하기보다,
  high-level **Application Programming Interface (API)** 를 이용
  - 예) C standard library 에서 시스템 콜을 직접 이용하고,
    유저에겐 보다 편리한 인터페이스의 서비스를 제공함
- Three most common APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

System Call

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

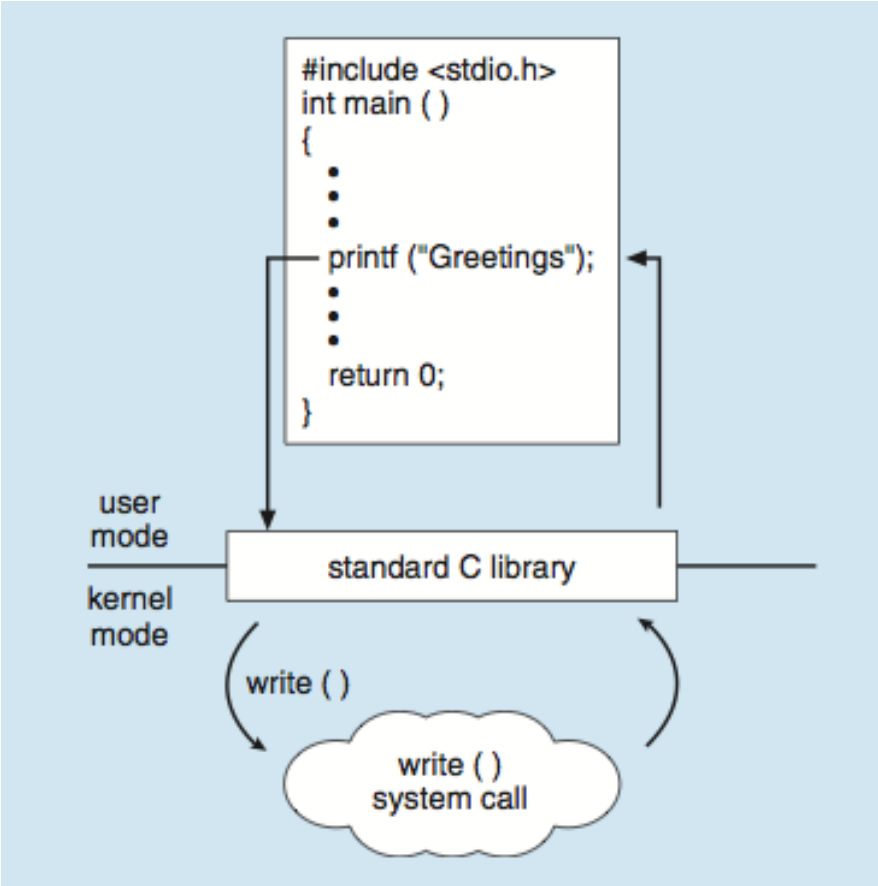
	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shmget()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

System Call


user application
open ( )
user mode
system call interface
kernel mode
i
open ( )
Implementation of open ( ) system call
return

Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

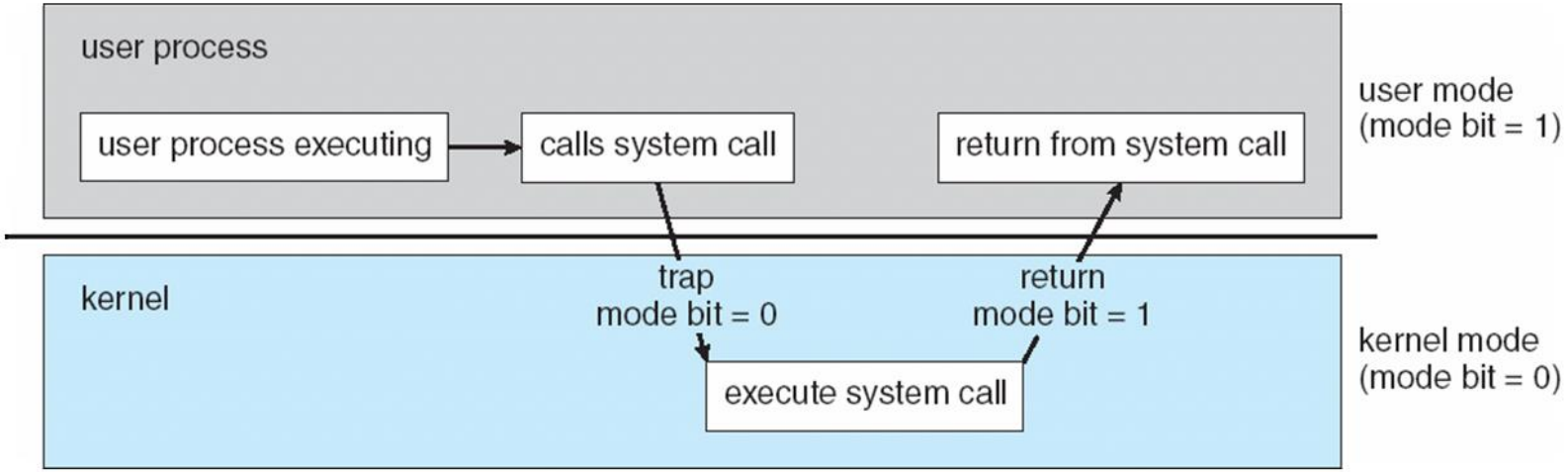
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

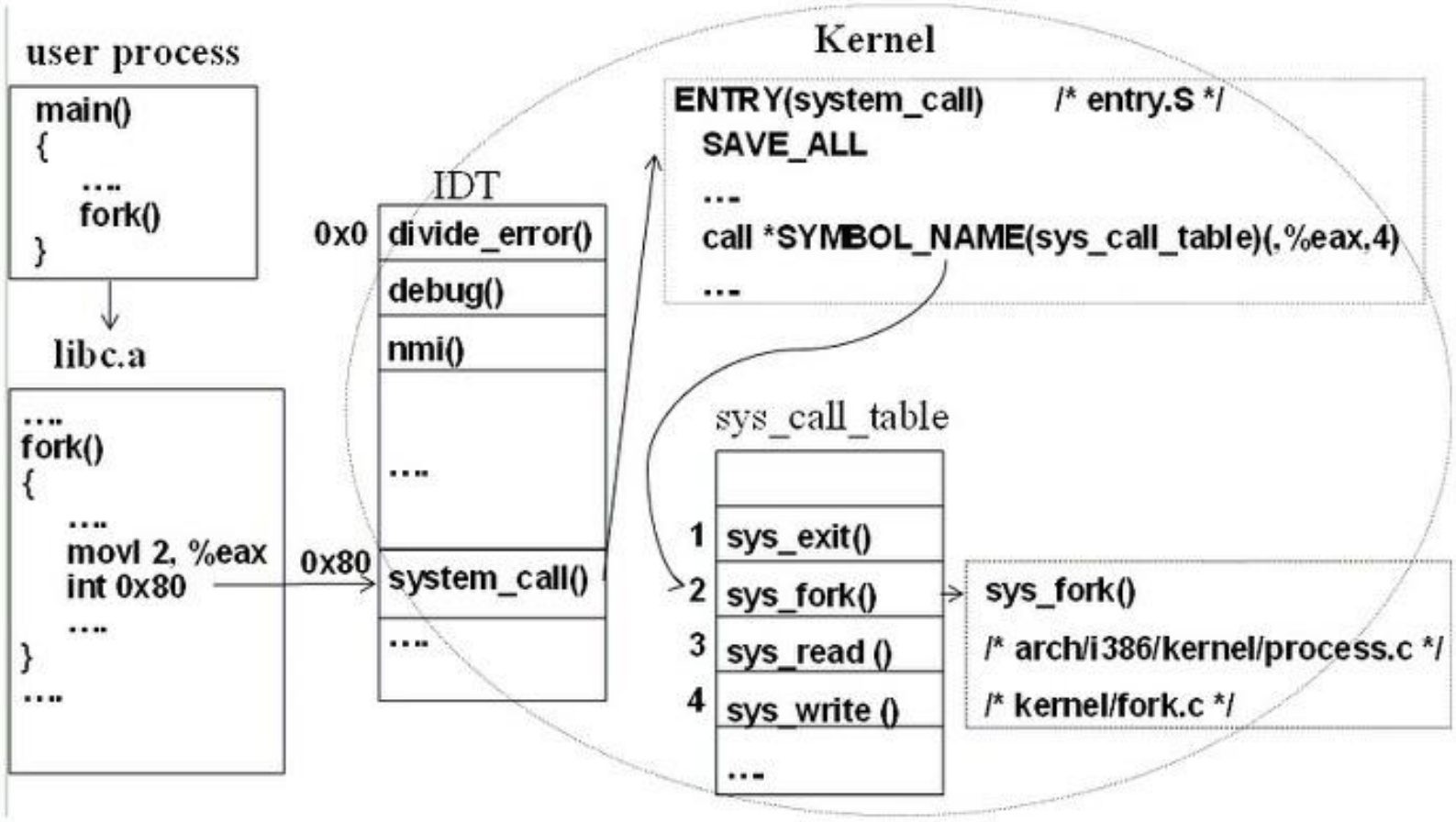
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Transition from User to Kernel Mode

- Trap을 통해서 Mode 전환
  - System call 을 위해 약속된 Trap을 유저 프로세스가 호출
    - Intel: Interrupt 0x80
  - CPU: 내부의 mode bit을 11 (user)에서 00 (kernel) 으로 변경하고, 커널의 Trap Handler 를 호출
  - OS: Trap Handler (=system call handler)가 전달된 정보에 따라 커널 내부 함수를 호출
  - OS: 작업 종료 후 mode를 user로 변경하고 다시 유저 프로세스 수행




user process
main()
{
....
fork()
}
libc.a
....
fork()
{
....
movl 2, %eax
int 0x80
....
}
....
0x80
IDT
0x0
divide_error()
debug()
nmi()
....
system_call()
....
Kernel
ENTRY(system_call) /* entry.S */
SAVE_ALL
...
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
...
sys_call_table
1 sys_exit()
2 sys_fork()
3 sys_read ()
4 sys_write ()
...
sys_fork()
/* arch/i386/kernel/process.c */
/* kernel/fork.c */

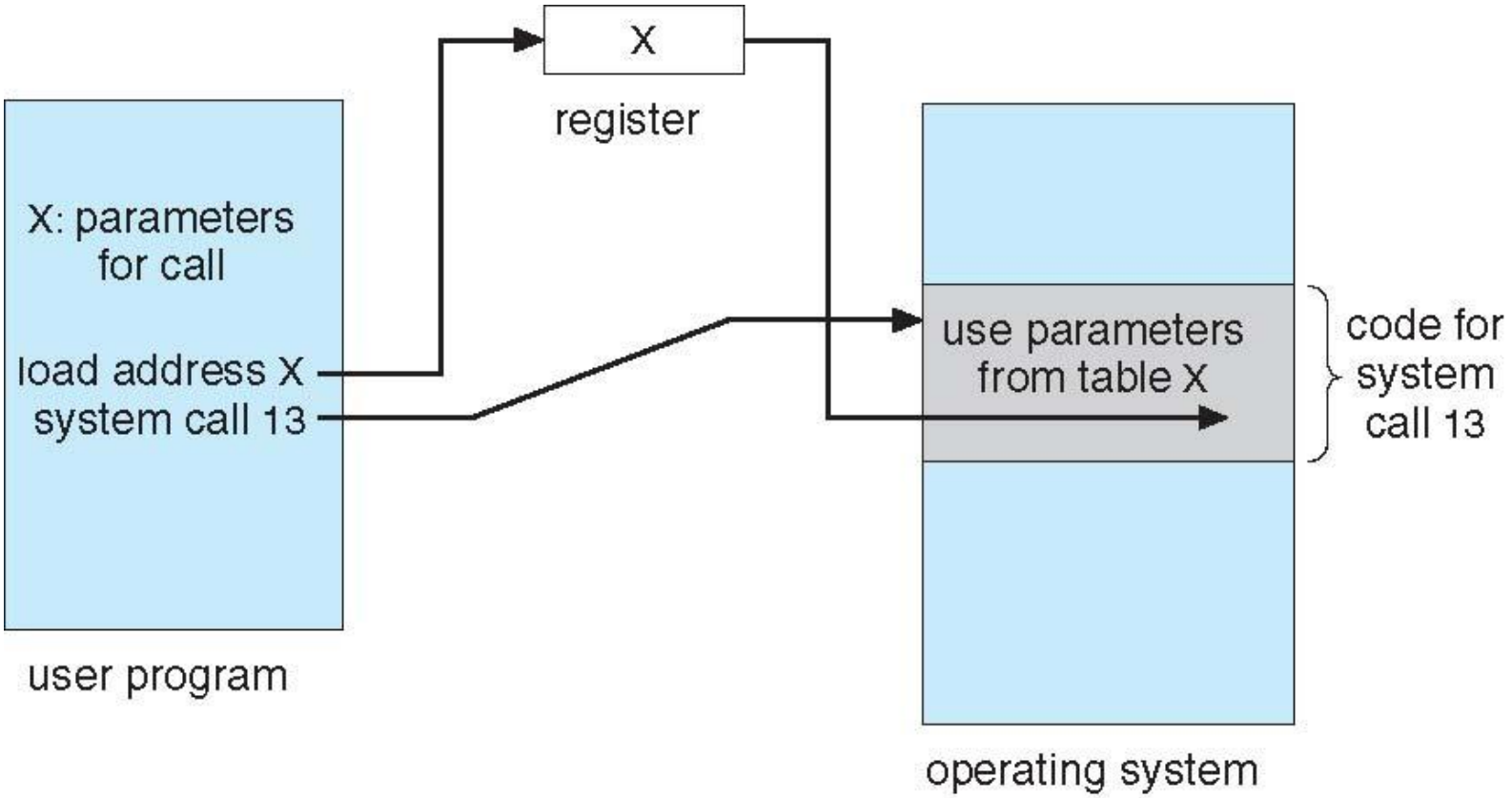
In kernel: do_sys_open() in fs/open.c

```
1021 long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
1022 {
1023         struct open_flags op;
1024         int fd = build_open_flags(flags, mode, &op);
1025         struct filename *tmp;
1026
1027         if (fd)
1028                 return fd;
1029
1030         tmp = getname(filename);
1031         if (IS_ERR(tmp))
1032                 return PTR_ERR(tmp);
1033
1034         fd = get_unused_fd_flags(flags);
1035         if (fd >= 0) {
1036                 struct file *f = do_filp_open(dfd, tmp, &op);
1037                 if (IS_ERR(f)) {
1038                         put_unused_fd(fd);
1039                         fd = PTR_ERR(f);
1040                 } else {
1041                         fsnotify_open(f);
1042                         fd_install(fd, f);
1043                 }
1044         }
1045         putname(tmp);
1046         return fd;
1047 }
```

System Call Parameter Passing

- 파라미터 전달을 위한 세 가지 방법
  - 레지스터 이용: 빠르고 단순함
    - 그러나 사용 가능한 레지스터 개수 이상의 파라미터가 있을 수 있음
  - 메모리 블록(연속된 메모리 공간)에 파라미터들을 저장하고, 시작 주소만을 레지스터에 넣어 전달
    - 리눅스, 솔라리스 등에서 사용
  - 스택을 이용: 유저 프로그램은 파라미터를 push, OS는 pop 해서 사용

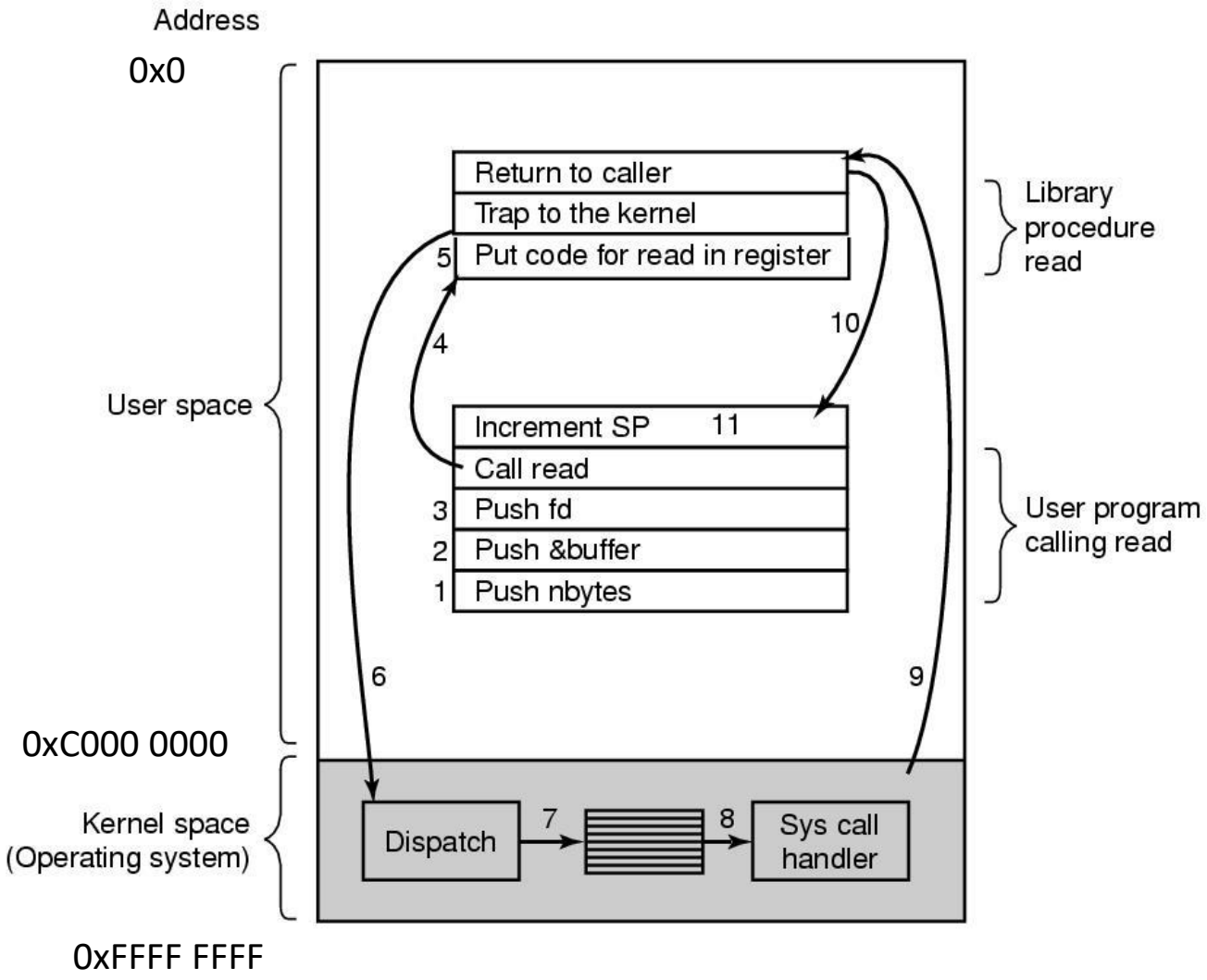
Parameter Passing via Memory block or table



Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

전체 정리: Real System Call



Kernel Structures

Monolithic kernel

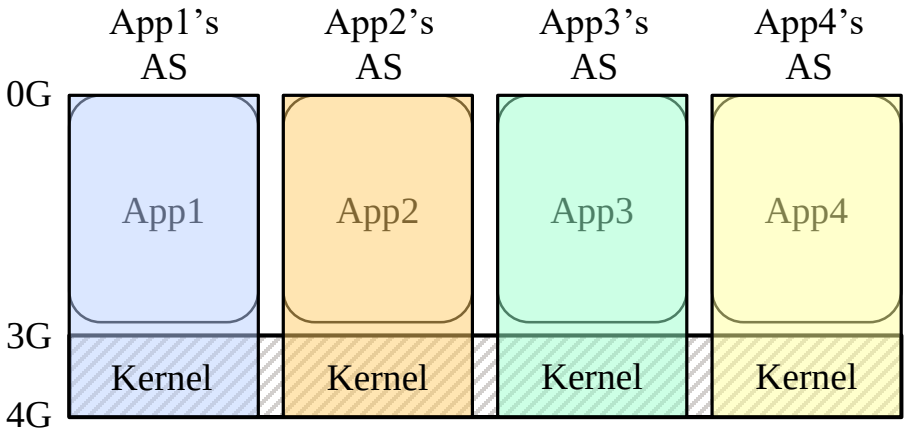
Micro kernel

Hybrid systems

Hypervisor

Monolithic kernel

- 특징
  - 커널의 모든 서비스가 같은 주소 공간에 위치
  - 어플리케이션은 자신의 주소공간에 커널 코드 영역을 매핑하여 커널 서비스를 이용
  - H/W 계층에 관한 단일한 abstraction을 정의
    - 라이브러리나 어플리케이션에게도 단일한 인터페이스 제공



Monolithic 커널에서의 주소공간 매핑
(X86-32bit, ARM-32bit)

Address Space

- 논리적 실체나 물리적 실체에 대응되는 주소의 범위를 정의한 공간 wikipedia
  - 같은 주소 공간에 있는 경우, 주소를 이용하여 접근 가능
  - 주소 공간을 알지 못하는 경우, 그 주소공간에 포함된 실체에 접근할 수 없음
    - 접근할 수 있는 다른 방법을 제공하여야만 접근 가능
- 예) 아파트 주소
  - 301호
  - 101동 301호
  - 102동 301호
  - 현대아파트 101동 301호
  - LH 아파트 101동 301호

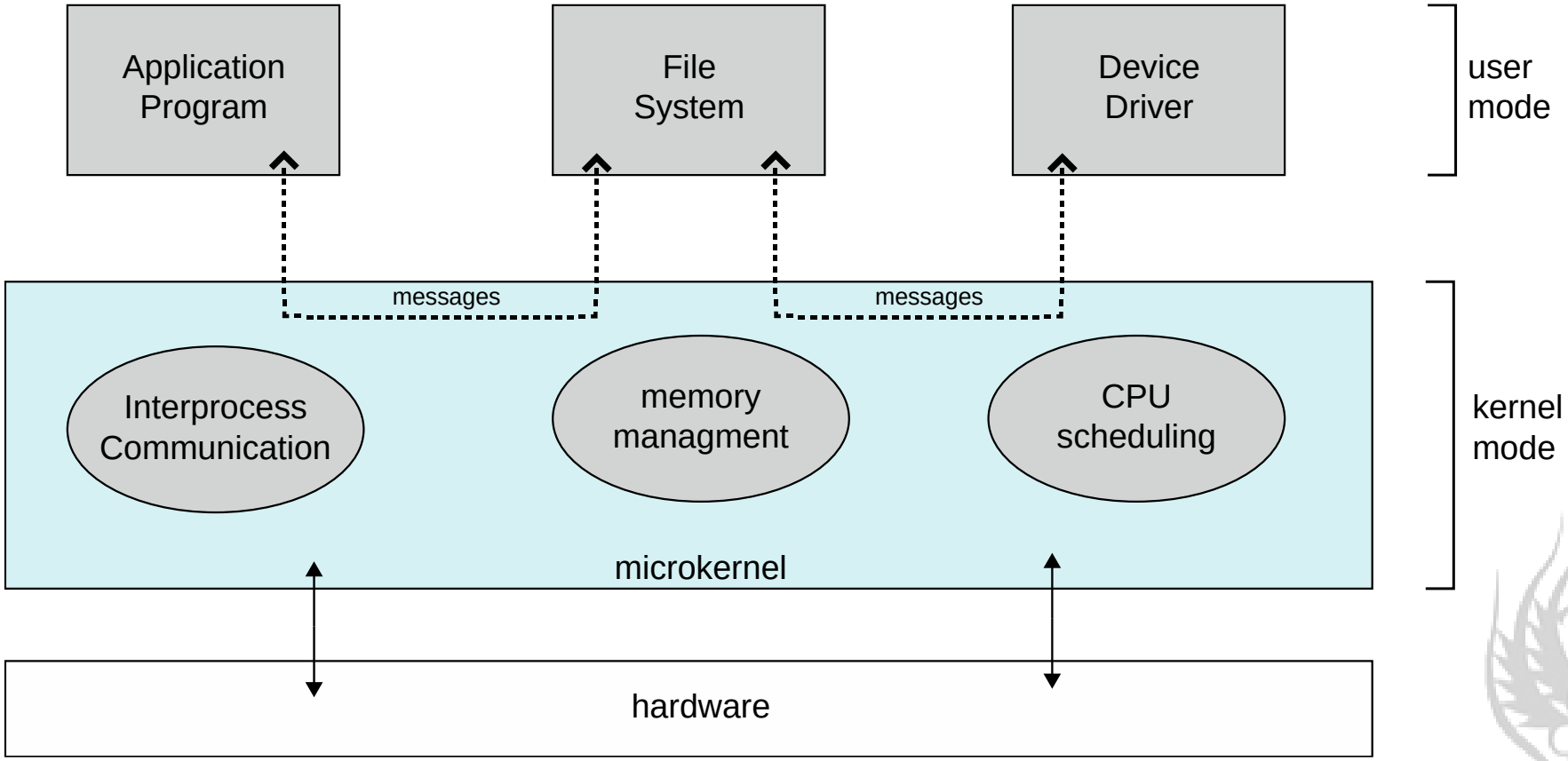
Monolithic kernel (Cont.)

- 장점
  - 어플리케이션과 커널 서비스가 같은 주소 공간에 위치하기 때문에, 시스템 콜 및 커널 서비스 간의 데이터 전달 시에 오버헤드가 적음
- 단점
  - 모든 서비스 모듈이 하나의 바이너리로 이루어져 있기 때문에 일부분의 수정이 전체에 영향을 미침
  - 각 모듈이 유기적으로 연결되어 있기 때문에 커널 크기가 커질수록 유지 보수가 어려움
  - 한 모듈의 버그가 시스템 전체에 영향을 끼침

Micro Kernel

- 특징
  - 커널 서비스를 기능에 따라 모듈화 하여 각각 독립된 주소 공간에서 실행
  - 이러한 모듈을 서버라 하며, 서버들은 독립된 프로세스로 구현
  - 커널의 동작은 서버들간의 통신 (message passing)으로 수행됨
  - 마이크로 커널은 서버들 간의 통신(IPC), 어플리케이션의 서비스 콜 전달과 같은 단순한 기능만을 제공
  - 예) Mach (1985). Mac OS X 커널(Darwin)은 Mach를 부분적으로 기반함

Microkernel System Structure



Micro Kernel(Cont.)

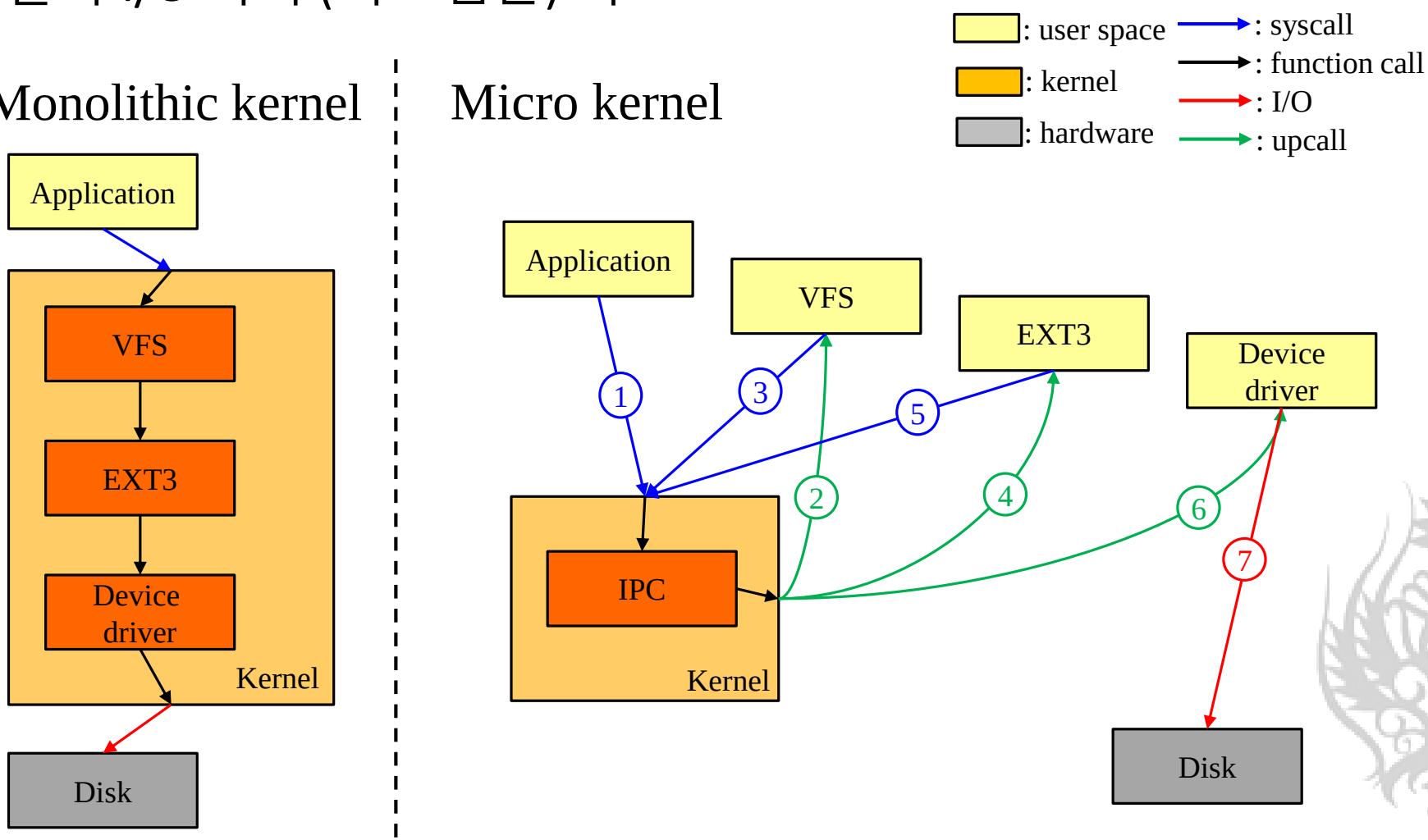
- 장점
  - 각 커널 서비스 서버가 따로 분리되어, 서로 간의 의존성이 낮음
    - Monolithic 커널 보다 독립적인 개발이 가능
    - 커널의 개발 및 유지 보수가 상대적으로 용이
  - 각 커널 서비스 서버의 간단한 시작/종료 가능
    - 불필요한 서비스의 서버는 종료
      - 많은 메모리 및 CPU utilization확보 가능
  - 이론적으로 micro 커널이 monolithic보다 안정적
    - 문제 있는 서비스는 서버를 재시작하여 해결
  - 각 서버가 protected memory에서 실행되므로 검증이 필요한 S/W 분야에 적합함
    - 임베디드 로봇 산업, 의료 컴퓨터 분야

Micro Kernel(Cont.)

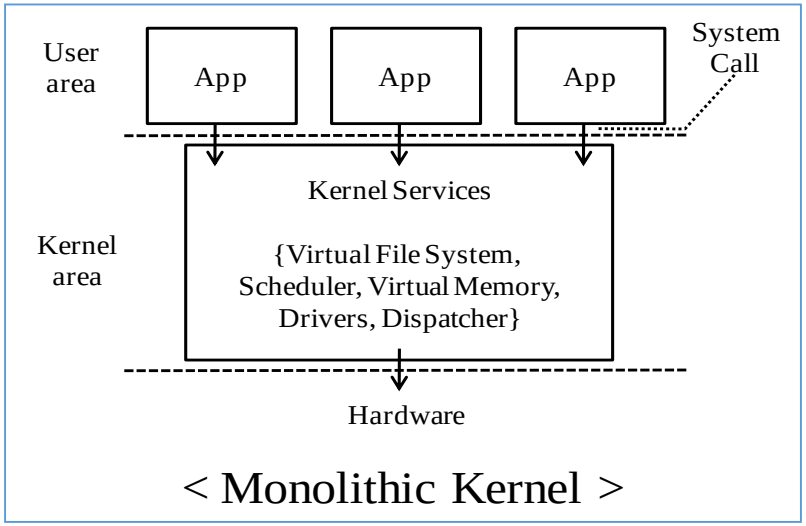
- 단점
  - Monolithic 커널보다 낮은 성능을 보임
    - 독립된 서버들 간의 통신 및 Context switching

Monolithic kernel vs. Micro kernel

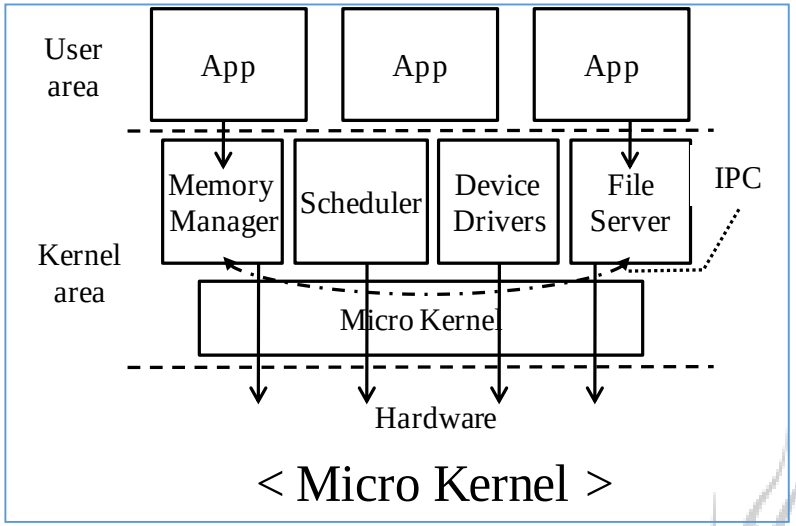
- 블록 I/O 처리 (시스템콜) 비교



Monolithic kernel and Micro Kernel



< Monolithic Kernel >



< Micro Kernel >

Hybrid Systems

- 대다수의 modern operating systems 은 정확히 특정 모델을 따르는 것이 아니라, Hybrid 구조를 사용
  - 리눅스와 솔라리스는 하나의 커널 주소 공간을 사용하는 monolithic kernel.
    - 그러나 기능의 확장을 위한 동적 모듈 제공 (loadable kernel module)
  - 윈도우도 monolithic 구조가 기본이지만, 특정 서브 시스템은 micro kernel 구조처럼 server 사용 가능
- Apple Mac OS X
  - Aqua UI + Cocoa programming environment (system program)
  - 커널은 Mach microkernel 기반으로 BSD UNIX 의 서브시스템을 이용
  - Loadable kernel module 도 사용 가능

Modules

- Loadable kernel modules
  - 각 컴포넌트(모듈, 서비스)를 분리하여 개발 가능
  - 필요할 때마다 모듈 단위로 커널에 로드해서 사용 가능
  - 객체지향적 접근 방식
    - 컴포넌트 간의 접근은 well-known (pre-defined) interface를 통해 수행
  - 일반적으로 Monolithic 커널에 기능 확장성, 유연성을 부여하는 용도로 사용
    - Linux, Solaris, etc

System Software

System Software

- 사용자에게 편리한 개발 및 수행 환경을 제공하는 SW
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- 대부분의 사용자/어플리케이션은 시스템 SW를 통해 시스템 콜을 수행함
  - C 라이브러리 함수: printf(), fread(), fwrite() 등

System Software

- UI
  - 텍스트 혹은 그래픽 기반의 사용자 인터페이스를 제공
- Program loading and execution
  - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- Communications
  - 프로세스, 사용자, 컴퓨터들 간의 접근 메커니즘을 제공함
    - SSH, Samba
- Background Services
  - 부팅 중에 실행되어 디스크 검사, 에러 로깅 등을 담당
  - 서비스, 서브 시스템, 데몬 등으로 알려짐

OS and Kernel

- OS와 kernel에 대한 두 가지 관점
  - OS = Kernel (수업에서는 이 관점으로)
  - OS = Kernel + windows system(GUI) + library
- Kernel
  - 운영체제의 핵심 부분으로, 자원할당, 하드웨어 인터페이스, 보안등을 담당
    - 예) Linux, Darwin, Windows NT kernel
    - 커널 + 타 시스템 소프트웨어 = 배포본 (일반적으로 OS 라 불림)
      - 예) Ubuntu, CentOS, OS X, 윈도우 10
- Windows system
  - 윈도우형태의 그래픽 사용자 인터페이스
    - 예) X Window (KDE, Gnome), Desktop Window Manager
- Library
  - 서브루틴과 자주 사용되는 함수들의 집합
    - 예) libc, win32.dll

Relation of Hardware, O/S and Application