# 2장. 운영체제 시스템 구조

# 목표

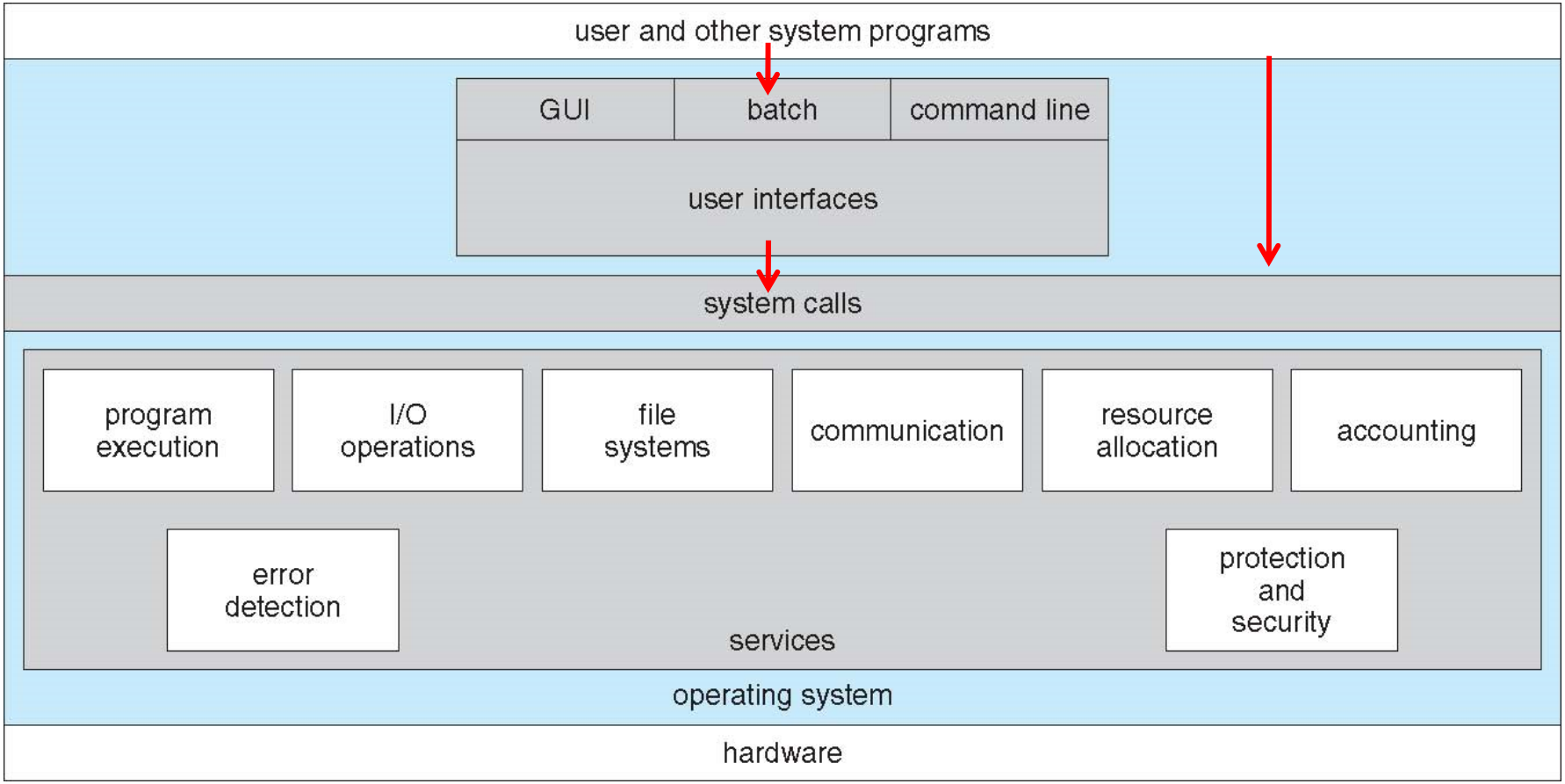
- 운영체제가 사용자, 프로세스 및 다른 시스템에게 **제공하는 서비스**를 기술
- 운영체제를 **구성하는 여러 방법**들을 논의
- 운영체제 **설치(install) 방법, 맞춤화(customize) 과정, 부팅 과정**을 설명

# 운영체제에 대한 관점

- 운영체제가 제공하는 **서비스** – 사용자/프로그래머 입장
- 운영체제가 제공하는 **인터페이스** – 프로그래머 입장
  - 시스템 호출(system call)
- 운영체제 **구성요소**(components)와 그들의 **상호 연결** – 설계자 입장

# 2.1 운영체제 서비스

- 운영체제 서비스 관점 – 프로그램 실행환경 제공



# 사용자를 위한 서비스

- 사용자 인터페이스(UI) – 거의 모든 운영체제가 제공
  - 명령어 라인 인터페이스(CLI)
  - 그래픽 사용자 인터페이스 (GUI)
  - 배치(batch) 인터페이스 – shell 프로그램
- 프로그램 실행(execution)
  - 프로그램을 메모리에 **적재**(load) 후 **실행**(run)시킴
  - 실행되는 프로그램을 **종료**시킴
    - 정상적 종료
    - 비정상적 종료 – 오류 등 요인
- 입출력 연산
  - 입출력  동작의 **효율과 보호**를 위해서
    - 사용자가 직접 입출력을 수행할 수 없게 함
  - 대신에 운영체제가 입출력 수행 서비스를 제공

# 사용자를 위한 서비스(계속)

- 파일 시스템 연산
  - 파일/디렉토리 read/write/create/delete
  - 파일 검색, 파일 목록 및 정보 출력
  - 파일/디렉토리 접근 권한 관리 등
- 통신
  - 프로세스 간에 정보를 교환
    - 동일 컴퓨터 내의 프로세스 간
    - 컴퓨터 네트워크로 연결된 다른 컴퓨터의 프로세스 간
  - 정보 교환 방법
    - 공유 메모리(shared memory) 경유 – 동일 컴퓨터
    - 메시지 전송(message passing) 방법 – 동일/다른 컴퓨터

# 사용자를 위한 서비스(계속)

- 오류 탐지
  - 오류 발생 요인 → 하드웨어/내부 인터럽트로 CPU에게 알림
    - CPU
    - 메모리 – 메모리 패리티 오류 등
    - 입출력 장치 – 네트워크 접속 실패, 프린터 종이 부족, 저장장치 패리티 오류 등
    - 사용자 프로그램 – overflow, 허가 받지 않은 위치의 메모리 접근 등
  - 오류가 발생할 때, 운영체제는 올바르고 일관성 있는 계산을 보장하기 위해 각 오류에 대해서 적절한 조치를 취해야 함.
    - 시스템 정지(halt)
    - 오류 발생 프로세스 종료(terminate) – 오류 코드 반환
    - 오류 원인 제거 후 재실행
  - 디버깅 툴 제공

# 시스템을 위한 서비스 - 효율적 동작

- 자원 할당(resource allocation)
  - 자원(resource)
    - **물리적 자원** - CPU cycle, main memory, 저장장치, 네트워크, 여러 입출력 장치 등
    - **추상적 자원** – 파일(file), 페이지(page), 프로세스(task), 프로토콜 등 운영체제가 관리를 위해 추상화한 객체
  - 운영체제는 다수의 사용자/다수의 프로세스에게 자원을 할당
  - 일부 자원은 특별한 할당 방법을 사용
  - 나머지 자원은 일반적인 request/release 방법을 사용
- 회계(accounting)
  - 컴퓨터 자원에 대한 사용 기록 – 어떤 자원을 얼마나 사용했는가
  - 사용 통계 또는 사용 요금 청구에 사용됨

# 시스템을 위한 서비스(계속)

- 보호(protection)와 보안(security)
  - **보호(protection)** – 한 프로세스가 다른 프로세스나 운영체제의 동작을 방해하지 않도록 시스템 자원에 대한 접근을 통제하는 것
  - **보안(security)** – 외부로부터의 부적합한 시스템 접근을 통제하는 것
    - 사용자 인증
    - 외부와 연결되는 입출력 장치의 부적합 접근 시도 방지
    - 침입탐지를 위한 모든 접속 기록

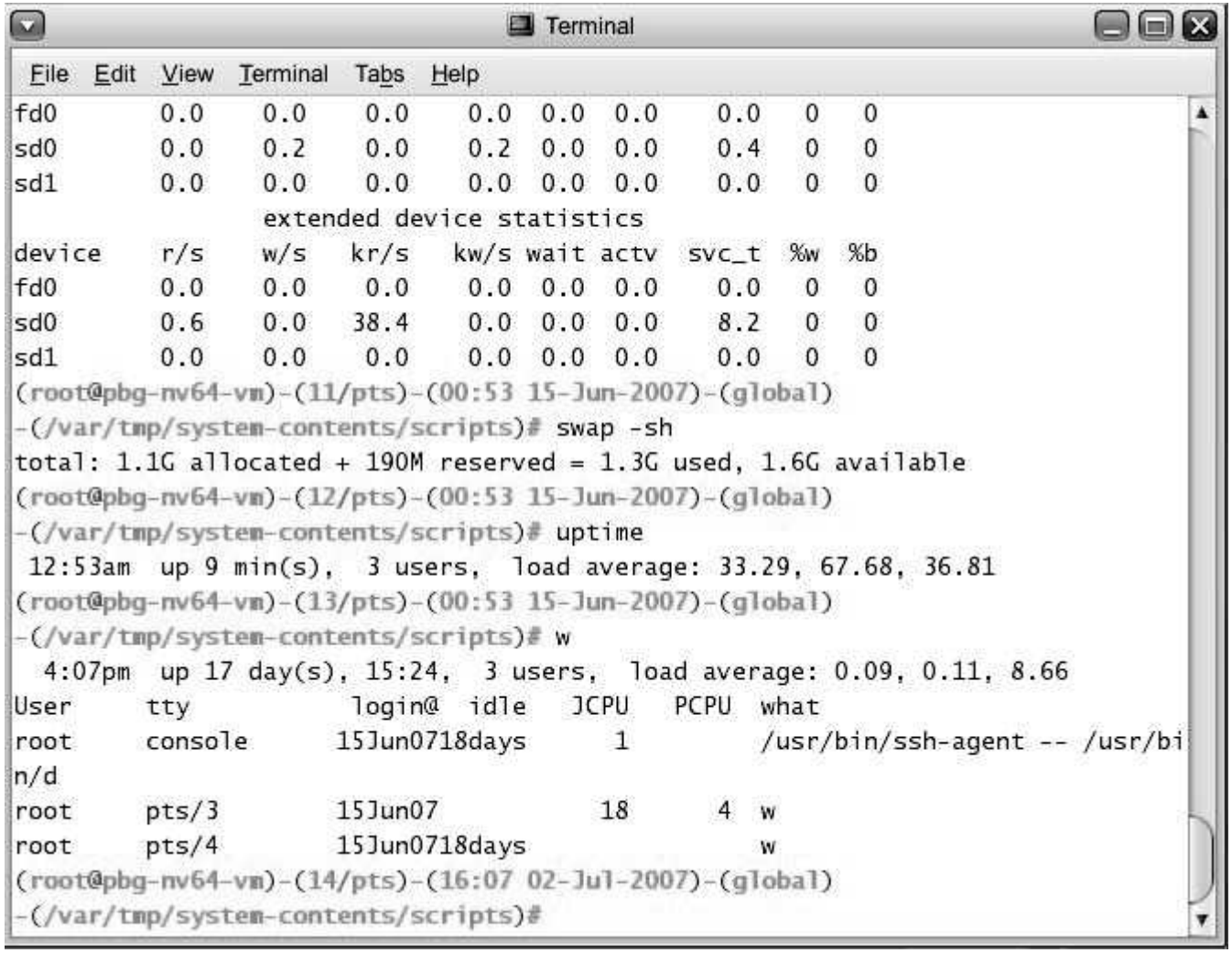
# 2.2 사용자 운영체제 인터페이스

- 명령어 해석기(command interpreter) – command line interface(CLI)
  - command를 입력 받아서 수행

    (예) UNIX 셸, MS-DOS
  - 구현 방법
    - 커널에 포함되어 구현, 또는
    - 작업이 시작되거나, 로그온할 때에(대화형) 수행되는 특수한 (시스템) 프로그램으로 구현
  - 여러 명령어 해석기를 제공하는 시스템에서는 선택하여 사용 가능
    - UNIX/Linux: Bourne shell (sh), C shell (csh), bash, ksh …
- 명령어 처리의 구현
  - 내장 명령어(built-in command)
    - 명령어 해석기에 명령어 처리 코드를 포함
  - 유틸리티 명령어(utility)
    - 명령어가 명령어 실행 파일을 나타내며,
    - 명령어 해석기는 단순히 이 파일을 적재하여 실행

    → 새로운 명령어를 쉽게 추가 가능 (셸의 수정 불필요)

# (예) Bourne Shell 명령어 해석기



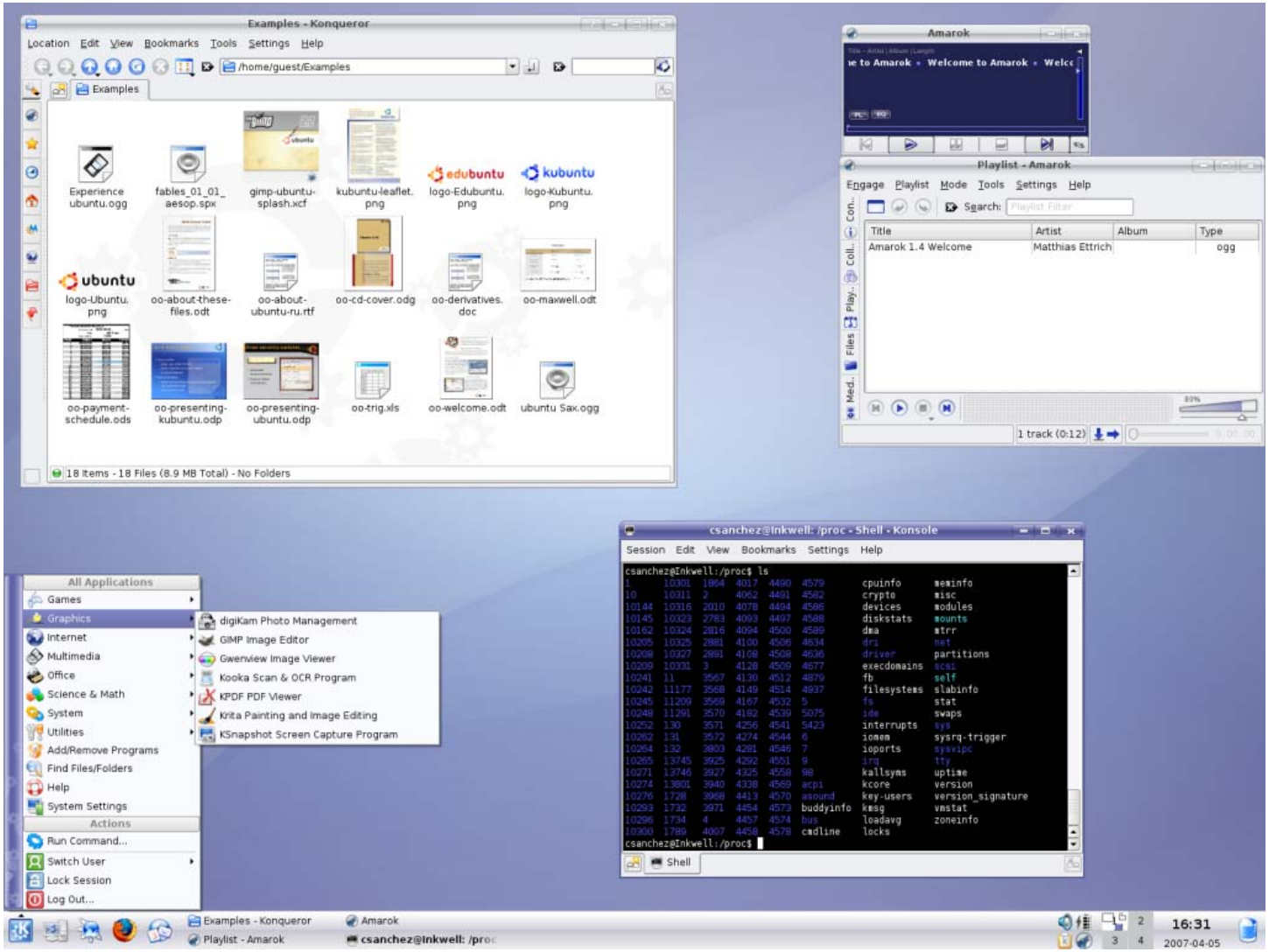
# 그래픽 사용자 인터페이스(GUI)

- 그래픽 사용자 인터페이스
  - 사용자 친화적 인터페이스
    - window, icon, menu, pointing device(mouse) → WIMP
  - 1970년대 초 Xerox PARC 연구소에서 발명
    - 1973년 Xerox Alto 컴퓨터에서 처음 사용
  - 1980년대 Apple Macintosh 개인용 컴퓨터에 사용
    - GUI가 급속히 보급되는 계기가 됨
    - Mac OS X 의 Aqua GUI – shell도 제공
  - Windows
    - 초창기의 Windows는 MS-DOS 기반, 현재 Windows 10
  - UNIX 운영체제 – GUI도 제공. 필요 시 GUI 사용 가능
    - X-windows 기반 GUI – 원격 GUI 가능
    - CDE(common desktop environment), KDE, GNOME desktop
  - 모바일 시스템 - 터치스크린 인터페이스

# Mac OS X GUI


Finder File Edit View Go Window Help
Mon 10:50 AM
Documents
DEVICES
iDisk
Macintosh HD
Backup
SHARED
Leo's MacBook
Aki's Mac mini
Bea's MacBook
Dad's PC
Office Mac Pro
PLACES
Desktop
liam
Applications
Documents
SEARCH FOR
Today
Yesterday
Past Week
All Images
All Movies
All Documents
Jones Family
Portable Document Format (PDF)
Marine Life
Name
Date Modified
Size
Kind
The Collector
Today, 8:23 AM
10.6 MB
Pages Publication
Cougars Scrapbook
Yesterday, 2:15 PM
4.2 MB
Pages Publication
New Zealand Fossils
Yesterday, 10:00 AM
448 KB
Keynote Document
Extreme 2008
Sunday, October 14, 2007, 6:48 PM
884 KB
Portable Document Format (PDF)
Car Purchase Comparison
Sunday, October 14, 2007, 11:38 AM
40 KB
Numbers Document
Canoe Fun.jpg
Saturday, October 13, 2007, 7:36 PM
2.7 MB
JPEG image
Jones Family
Saturday, October 13, 2007, 5:53 PM
768 KB
Portable Document Format (PDF)
Marine Life
Thursday, October 11, 2007, 3:20 PM
26.1 MB
Keynote Document
Gardner Letter
Wednesday, October 10, 2007, 2:40 PM
320 KB
Pages Publication
Southside Jazz Fest
Tuesday, October 9, 2007, 2:41 PM
32 KB
Portable Document Format (PDF)
Mountain Bike for Sale
Tuesday, September 25, 2007, 10:02 AM
72 KB
Portable Document Format (PDF)
Investor Newsletter
Saturday, September 22, 2007, 6:18 PM
6.8 MB
Pages Publication
1 of 12 selected, 213.74 GB available
Show In Finder
Napa Travel Brochure
2nd Grade Scrapbook
Mountain Bike Ad
Investor Newsletter
Redwoods Quarterly
Benefits Overview
Samantha's Journal

# UNIX/Linux KDE GUI



# 사용자 인터페이스의 선택

- 명령어 라인 인터페이스(CLI)
  - 시스템 관리자, 파워 유저가 많이 사용 – 작업에 빨리 접근, 효율적
  - 어떤 시스템은 GUI를 통해 일부 기능만 사용 가능하며
    나머지 작업은 CLI를 사용해야 함
  - 프로그램 기능이 있어서 반복적 작업에 효과적 → **셸 스크립트**
- 그래픽 사용자 인터페이스(GUI)
  - 사용하기 쉬움
  - 예전의 Mac OS는 CLI를 제공하지 않았으나,
    현재의 UNIX 커널을 기반으로 만든 Mac OS X는 CLI도 제공함
- 사용자 인터페이스는 OS의 직접적인 기능이 아님
  - 사용자 인터페이스는 사용자마다 다를 수도 있음
  - 대개, 운영체제 시스템 구조에서 제외됨

# 2.3 시스템 호출(system call)

- 시스템 호출(system call)
  - 운영체제가 제공하는 서비스에 대한 프로그래밍 인터페이스 제공
  - 운영체제 프로그래밍 인터페이스
    - 대개 C/C++와 같은 고급 언어 루틴 형태로 제공

      → 프로그래머가 시스템 호출에 대한 C/C++ 함수를 호출하여 사용
    - 저수준의 작업은 **시스템 호출 명령어**를 포함하는 어셈블리 프로그램으로 작성
    - MS-DOS는 직접 시스템 호출 명령어를 사용하여 운영체제 서비스를 요청하도록 되어 있음 → 어셈블리 프로그램
- 시스템 호출 명령어 = 소프트웨어 인터럽트 명령어

  80x86: INT 80 … interrupt

  IBM System/390: SVC *n* … supervisor call

  680x0: trap 4 … trap

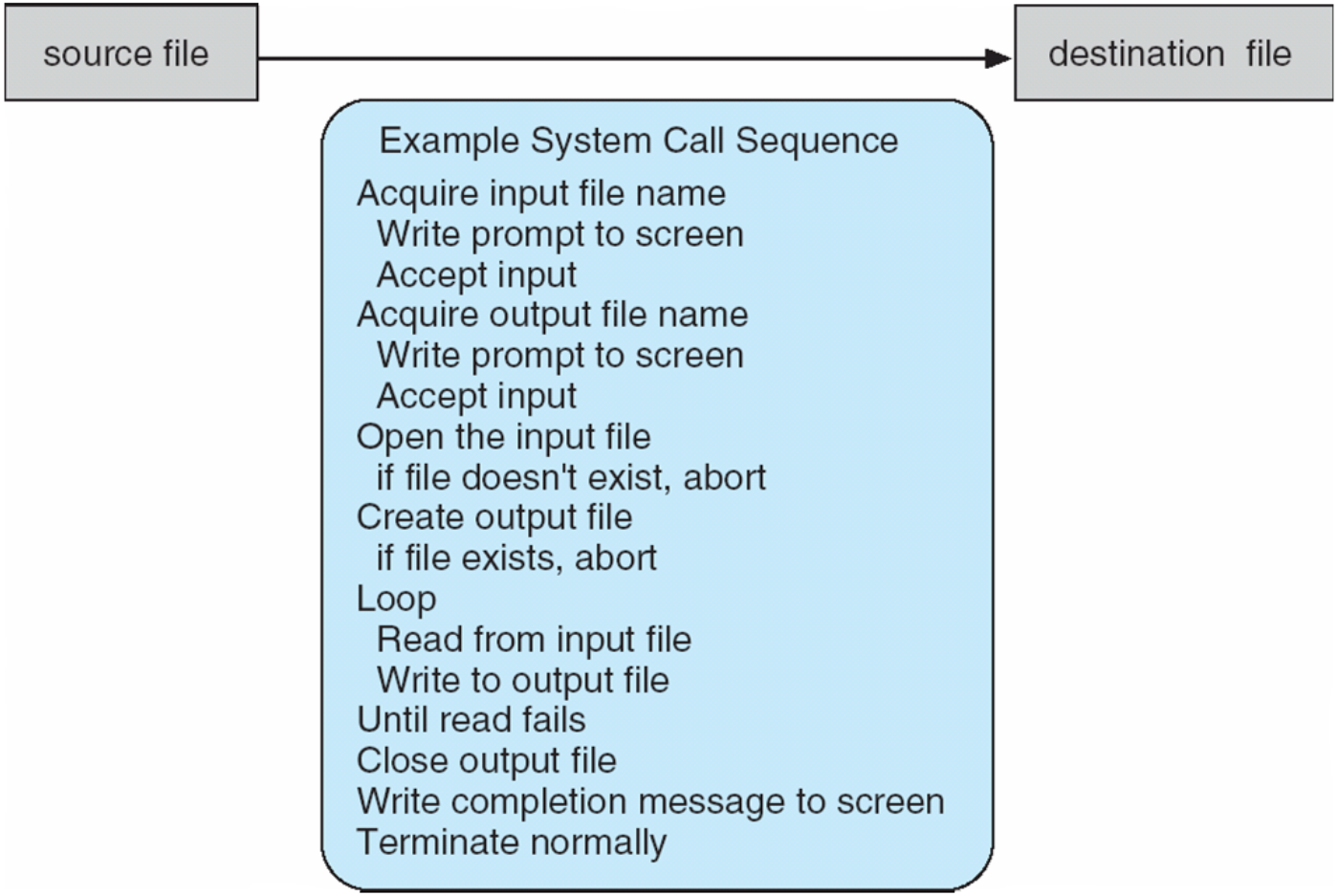
  ARM: SWI *n* … software interrupt
  - 오퍼랜드 숫자는 소프트웨어 인터럽트의 종류를 구분하는 데 사용

# 응용 프로그래밍 인터페이스(API)

- API (Application Programming Interface)
  - 응용 프로그래머에서 사용 가능한 함수의 집합을 지정
  - API 함수에서는 응용 프로그래머를 대신하여 필요한 실제 시스템 호출들을 호출함.
- 시스템 호출보다 API를 더 많이 사용하는 이유
  - 프로그램 이식성(portability)
  - 사용하기 쉬움
- 대표적인 API
  - Windows API
  - POSIX API – UNIX, Linux, Mac OS X
  - Java API – Java virtual machine (JVM)
  - …

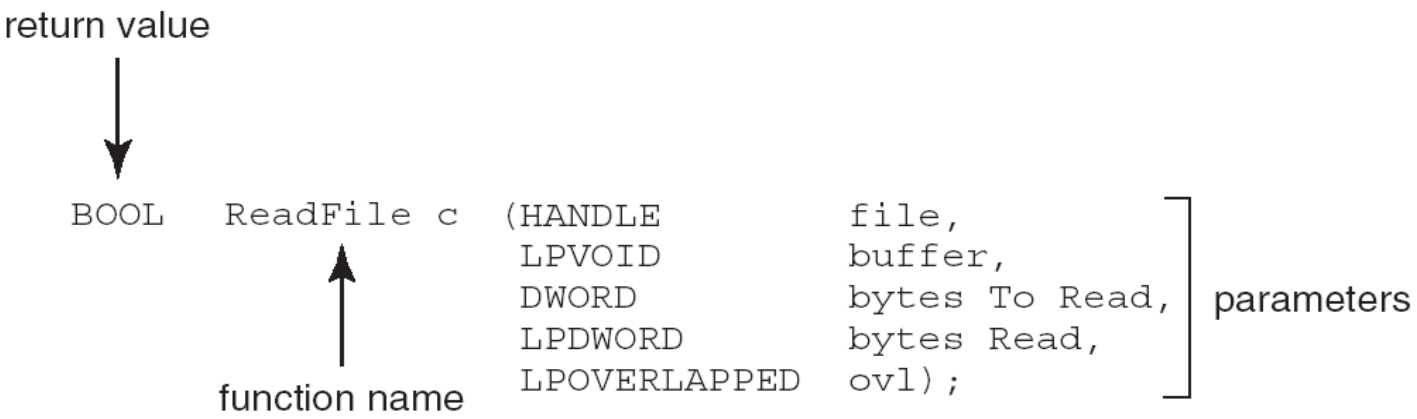
# 시스템 호출 사용 예

- (예) copy 동작에서 사용하는 시스템 호출



# 표준 API의 예

- Win32 - ReadFile() function




- UNIX/Linux – read() function

```
#include <unistd.h>

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

return value | function name | parameters

# 시스템 호출의 구현

- 각 시스템 호출에 번호가 부여됨
  - OS 커널에서 시스템 호출 처리 루틴의 주소 테이블의 인덱스로 이 번호를 사용함

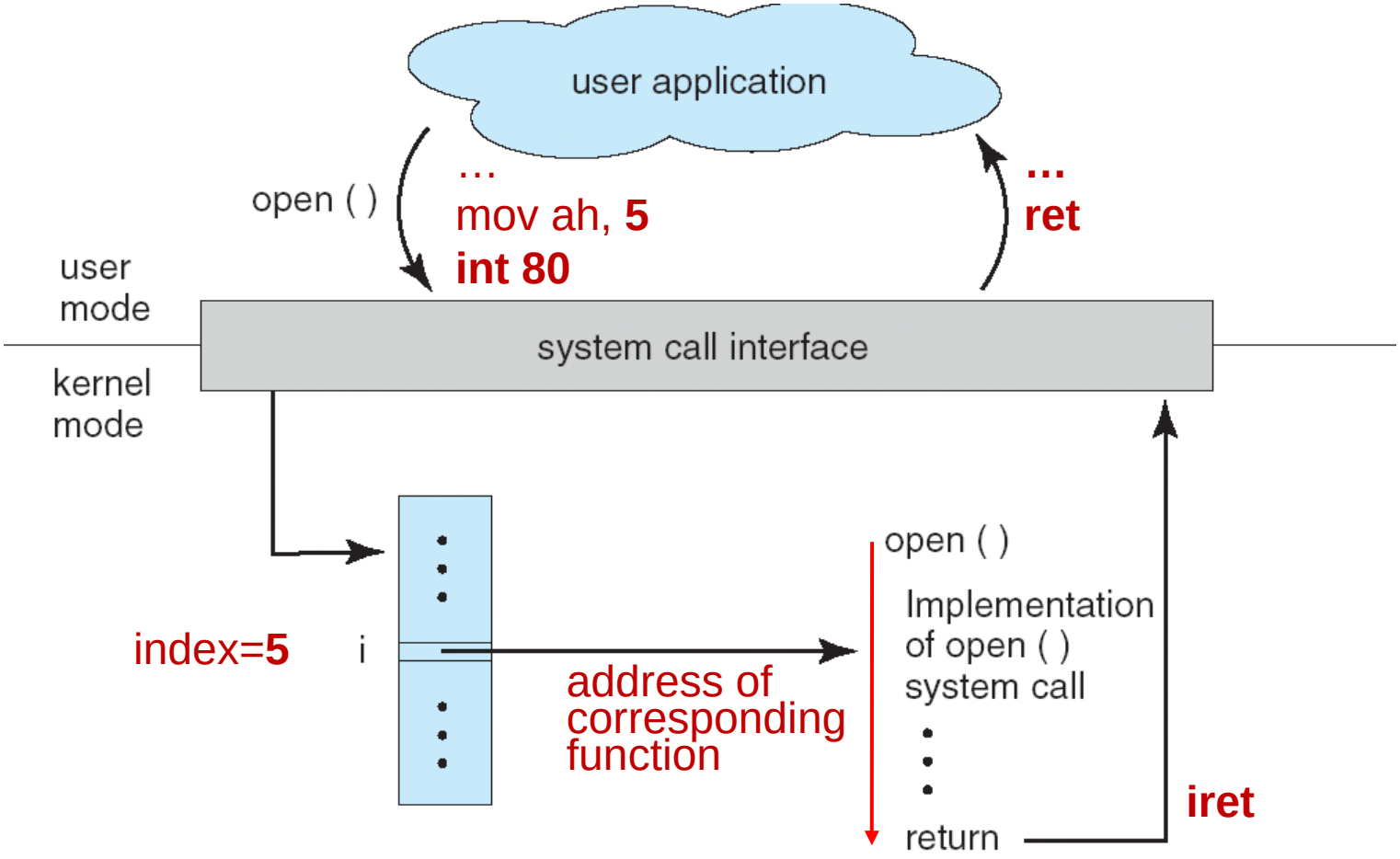
    (ex) mov ah, 5

    int 80

| | | |
|---|---|---|
| 1 | exit | terminate the current process |
| 2 | fork | create a child process |
| 3 | read | read from a file descriptor |
| 4 | write | write to a file descriptor |
| 5 | open | open a file or device |
| 6 | close | close a file descriptor |

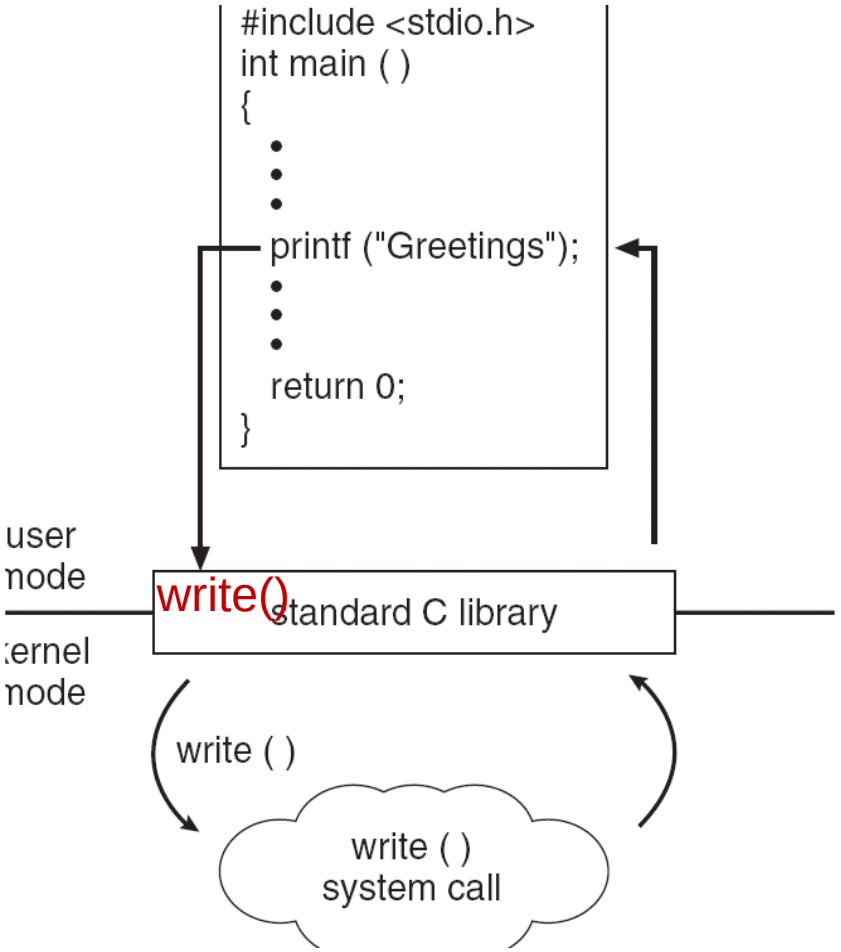
- 시스템 호출 인터페이스
  - 매개 변수 전달 받음
  - 관련된 시스템 호출을 호출
  - status와 반환 값 반환
- 운영체제 인터페이스의 상세 내용은 API에 의해서 숨겨짐
  - 대개 run-time support library에 의해서 관리됨
  - 사용자는 API 매개변수 규칙을 준수하고, 시스템 호출의 결과로 OS가 수행하는 작업에 대해서 이해하면 됨

# 시스템 호출의 OS 내에서의 처리



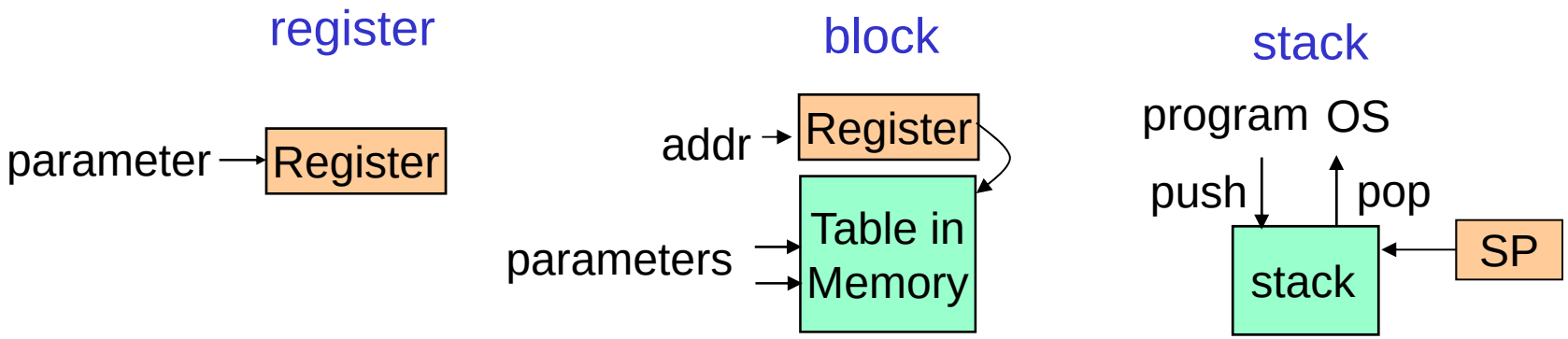
# 표준 C 라이브러리 함수 동작 예

- (예) C 프로그램이 printf() 함수를 호출하면, printf() 함수에서 write() 시스템 호출을 호출함



# 시스템 호출 매개변수 전달

- 시스템 호출 매개변수 전달 방법
  - **레지스터(register)** – 레지스터에 매개변수 저장, 개수 제한
    - 매개변수가 많으면 블록을 사용하여 추가 전달
  - **블록(block)** – 메모리 블록에 매개변수를 저장하고, 블록의 주소를 레지스터를 통해 전달
  - **스택(stack)** – 응용 프로그램이 시스템 스택에 push, 운영체제에서 pop
    - (실제로는 pop없이 스택을 접근하여 매개변수를 참조함)



# 2.4 시스템 호출의 유형

- 시스템 호출의 중요 범주
  - 프로세스 제어
  - 파일 관리
  - 장치(device) 관리
  - 정보 유지보수
  - 통신 및 보호
- 프로세스 제어
  - 프로세스 생성/적재/실행/종료, 중지
  - 프로세스 속성 읽기/설정, 메모리 할당/반납(free)
  - 시간 대기, 사건 대기, 사건 알림
- 파일 관리
  - 파일 생성, 삭제, 열기, 닫기, 읽기, 쓰기, 위치변경
  - 파일 속성 읽기/설정

# 시스템 호출 유형(계속)

- 장치 관리
  - 장치 요구(request)/반납(release)/읽기/쓰기/위치 변경
  - 장치 속성 읽기/설정, 논리적 부착(attach)/분리(detach)
- 정보 유지보수
  - 시간, 날짜, 시스템 데이터, 프로세스/파일/장치 속성
- 통신
  - 통신 연결 생성/삭제
  - 메시지 송신, 수신, 상태 정보 전달
  - 원격 장치 부착(attach)/분리(detach)
- 보호
  - 역사적으로 다중 사용자/다중 프로그램 환경에서 고려됨
  - 네트워크와 인터넷 연결로 모든 컴퓨터 시스템에서 보호를 고려해야 함
  - 자원 접근 제어 – 허가권 설정

# Windows와 Unix 시스템 호출 예

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

# POSIX 시스템 호출 (API)

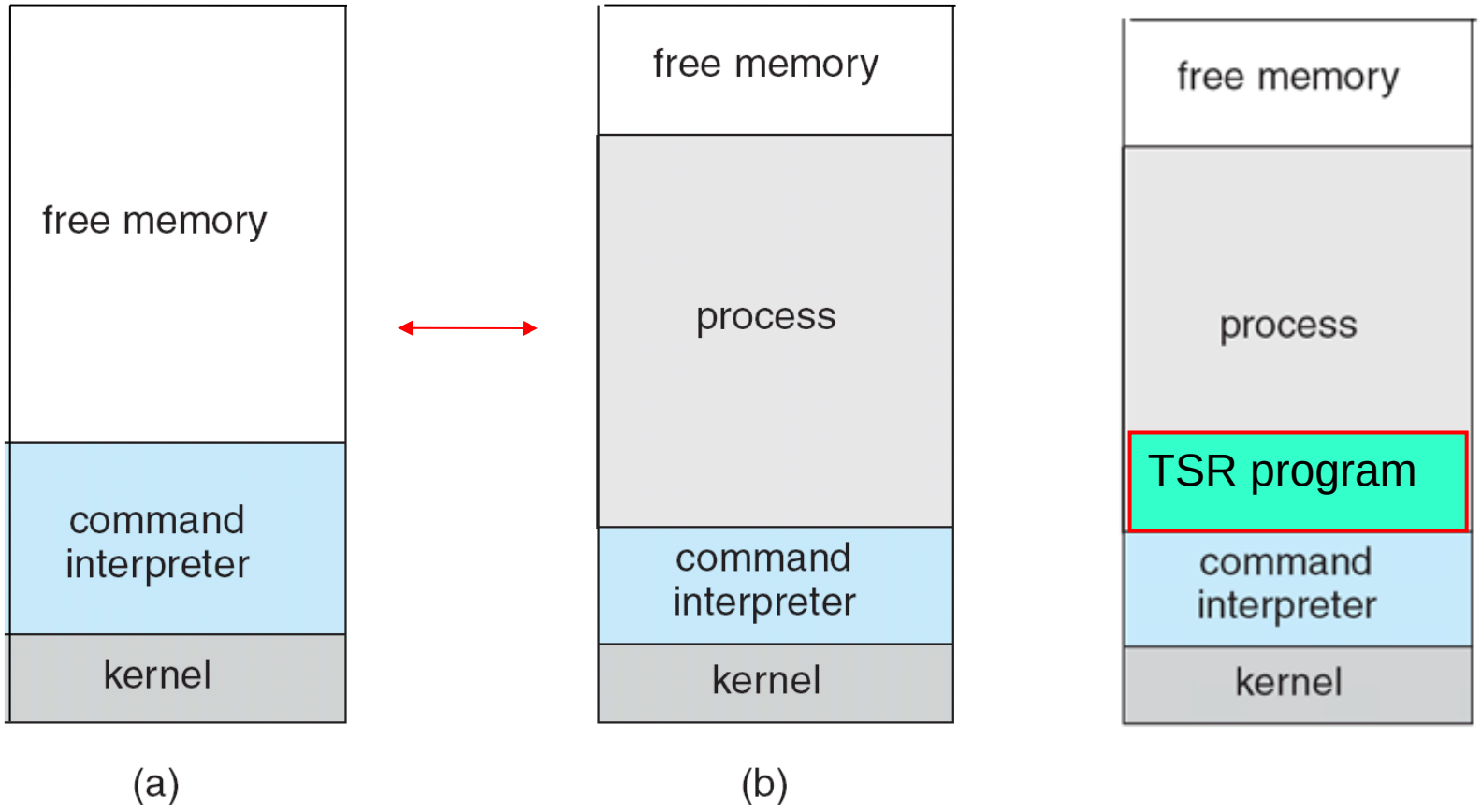
- Portable OS Interface (POSIX) 표준 – UNIX System V에 기반

| Call | Description |
|---|---|
| pid = fork() | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, opts) | Wait for a child to terminate |
| s = wait(&status) | Old version of waitpid |
| s = execve(name, argv, envp) | Replace a process core image |
| exit(status) | Terminate process execution and return status |
| size = brk(addr) | Set the size of the data segment |
| pid = getpid() | Return the caller's process id |
| pid = getpgrp() | Return the id of the caller's process group |
| pid = setsid() | Create a new session and return its proc. group id |
| l = ptrace(req, pid, addr, data) | Used for debugging |
| s = sigaction(sig, &act, &oldact) | Define action to take on signals |
| s = sigreturn(&context) | Return from a signal |
| s = sigprocmask(how, &set, &old) | Examine or change the signal mask |
| s = sigpending(set) | Get the set of blocked signals |
| s = sigsuspend(sigmask) | Replace the signal mask and suspend the process |
| s = kill(pid, sig) | Send a signal to a process |
| residual = alarm(seconds) | Set the alarm clock signal |
| s = pause() | Suspend the caller until the next signal |
| fd = creat(name, mode) | Obsolete way to create a new file |
| fd = mknod(name, mode, addr) | Create a regular, special, or directory i-node |

| | |
|---|---|
| fd = open(file, how, ...) | Open a file for reading, writing or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| pos = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |
| s = fstat(fd, &buf) | Get a file's status information |
| fd = dup(fd) | Allocate a new file descriptor for an open file |
| s = pipe(&fd[0]) | Create a pipe |
| s = ioctl(fd, request, argp) | Perform special operations on a file |
| s = access(name, amode) | Check a file's accessibility |
| s = rename(old, new) | Give a file a new name |
| s = fcntl(fd, cmd, ...) | File locking and other operations |
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| Call | Description |
| s = umount(special) | Unmount a file system |
| s = sync() | Flush all cached blocks to the disk |
| s = chdir(dirname) | Change the working directory |
| s = chroot(dirname) | Change the root directory |
| s = chmod(name, mode) | Change a file's protection bits |
| uid = getuid() | Get the caller's uid |

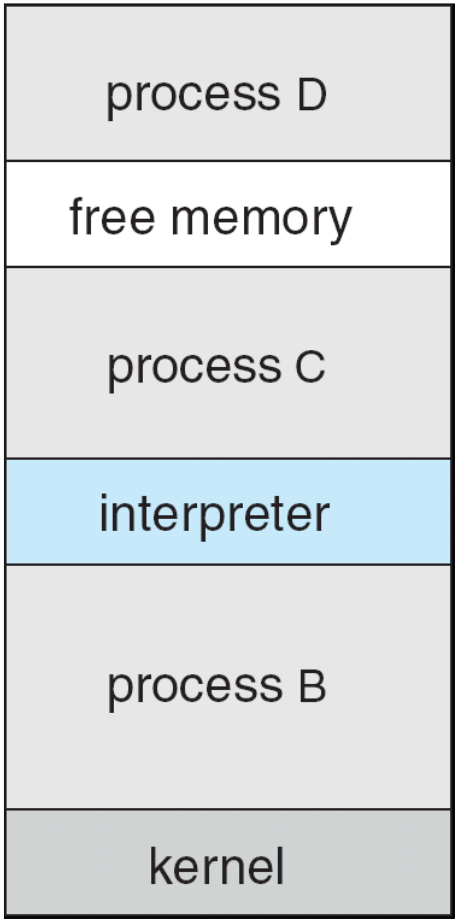
# MS-DOS 실행: single-tasking 시스템

**At System Start-up** **Running a Program**




(cf) **TSR(terminate and stay resident)** MS-DOS system call – hooks an interrupt

# UNIX 실행 : multi-tasking 시스템




- Multitasking 시스템
  - 셸이 다른 프로그램이 실행되는 동안 계속하여 수행 가능
    → 후면 처리(background processing)
  - 여러 개의 작업을 동시에 처리 가능
  - fork() 와 exec() 시스템 호출 사용하여 구현

# 2.5 시스템 프로그램

- 시스템 프로그램
  - 프로그램 개발과 실행을 위해 편리한 환경을 제공하는 프로그램
    - 파일 관리: copy, delete, rename ...
    - 상태 정보: date, available memory/disk space, # of users
    - 파일 변경: text editor, transformation of the text
    - 프로그래밍 언어 지원: compiler, assembler, interpreter
    - 프로그램 적재/실행: loader, linkage editor, debugger
    - 통신: telnet, mail, ftp, …
    - 서비스, deamon, 서브시스템: 시스템이 정지할 때까지 특정 서비스 제공을 위하여 백그라운드로 계속해서 실행되는 프로세스
      – 네트워크 deamon
- 응용 프로그램
  - 일반적인 문제 해결이나 연산 처리에 유용한 프로그램
- 운영체제에 대한 대부분의 사용자의 관점은 system call이라기 보다는 시스템 프로그램과 응용 프로그램에 의해 정의됨

# 2.6 운영체제의 설계와 구현

- 설계 목표(goal)와 명세(specification) 정의
  - 하드웨어와 시스템 유형(batch, time shared, single user, multi-user, distributed, real time, general purpose )의 선택에 영향 받음
- 요구조건 지정 (설계 목표)
  - **사용자 목표** – 사용하기 쉽고, 배우기 쉽고, 신뢰성 있고, 안전하고, 빠름
  - **시스템 목표** – 설계, 구현, 유지 보수가 쉽고, 유연성, 신뢰성, 무오류, 효율적이어야 함.
  - 요구 조건은 애매하고 다양하게 해석될 수 있으며, 일반적으로 합의된 사항은 없음
- 운영체제의 설계 및 구현
  - 모든 요구 조건을 해결하는 완전한 해결책은 없지만, 성공이 입증된 접근법은 있다.
  - 소프트웨어 공학에서 개발된 일반적인 원칙을 사용
  - 내부 구조는 운영체제마다 다를 수 있다.

# 기법(Mechanism)과 정책(Policy)

- Mechanism과 Policy – 둘을 분리하는 것이 중요한 원칙임
  - **Mechanism**: 어떻게 할 것인가? (How)
  - **Policy**: 무엇을 할 것인가? (What)
- (예) CPU 보호 방법
  - mechanism: CPU 보호를 위하여 타이머 구조를 사용 ➔ 바뀌지 않는 것
  - policy: 특정 사용자를 위한 타이머 양을 결정하는 것 ➔ 변경 가능
- Mechanism과 Policy의 분리는 flexibility를 위해서 중요함
  - policy는 시간이 지남에 따라서 변경될 수 있음.
  - mechanism은 정책 변경에 민감하지 않는 일반적인 것이 바람직함
  - 시스템 매개 변수의 재정의에 의해서 policy 변경이 이루어지도록 함
    - UNIX 초기 : 시분할 스케줄러
    - Solaris 최근 버전 : loadable table에 의해 제어되는 스케줄러
  - microkernel 기반 OS – mechanism과 policy의 극단적 분리
  - Mac OS와 Windows – mechanism과 policy를 함께 작성됨
    - interface가 kernel과 system library에 포함 ➔ global look and feel

# 시스템 구현

- 운영체제의 구현
  - 초기의 운영체제는 어셈블리 언어로 작성됨
  - 현재의 대부분의 운영체제는 고급 언어로 작성됨
    - *MULTICS*: PL/1;
    - *UNIX, Linux, Windows*: C
  - 커널의 저수준 코드는 여전히 어셈블리 언어로 작성됨
    - 장치 드라이버, 레지스터 상태 저장 및 복구 등
- system programs의 구현
  - C, C++, interpreted languages (Perl, Python, shell script ...)로 작성됨

# 고급 언어 구현의 장단점

- 장점
  - 코드를 빠르게 작성
  - 간결한 코드
  - 이해와 디버깅이 쉬움
  - 이식(port)하기 훨씬 쉬움
- 주장되는 단점:
  - 속도가 느려짐, 소요 메모리가 증가됨 → 현재는 문제가 되지 않음
- 문제가 안 되는 이유
  - 현대의 최적화 컴파일러는 일반적인 어셈블리 언어 프로그래머보다 훨씬 우수한 코드를 생성할 수 있다.
  - OS의 주된 성능 향상은 더 좋은 **자료구조와 알고리즘**에 의한 것이다.

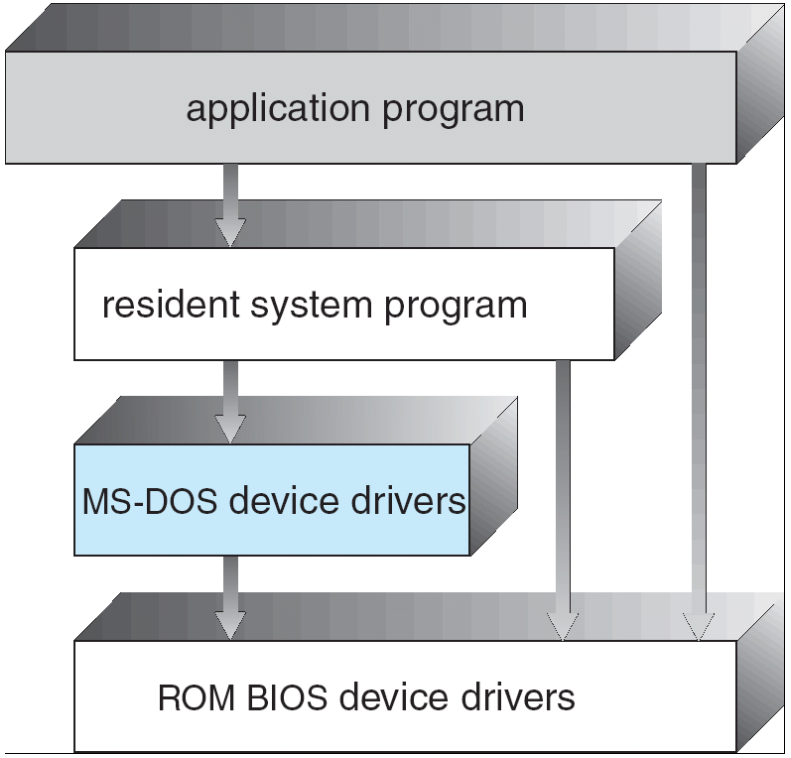
# 2.7 운영체제 구조

- Simple structure
  - **monolithic** 구조 – 많은 기능들이 하나의 계층으로 구현됨
  - 많은 상용 운영체제가 잘 정의된 구조를 갖지 않음
    - small, simple, limited system으로 시작되어, 원래 범위 이상으로 발전함
- Layered structure
  - 운영체제가 여러 계층으로 구분됨
  - 각 계층은 하위 계층 위에 구축됨
- Microkernel
  - 필수적이 아닌 구성 요소를 커널에서 모두 제거하고 시스템 및 사용자 수준 프로그램으로 구현함 ➔ smaller kernel

# Simple Structure – MS-DOS

- MS-DOS
  - 최소 공간에 최대 기능을 제공하도록 작성
    → 모듈들로 구분되지 않음
  - 인터페이스와 functionality 계층이 잘 분리되어 있지 않음
    → 운영체제를 거치지 않고 BIOS를 사용하여 입출력 가능
- 기능이 제한적인 8088 CPU 사용
  - no dual mode
  - no HW protection

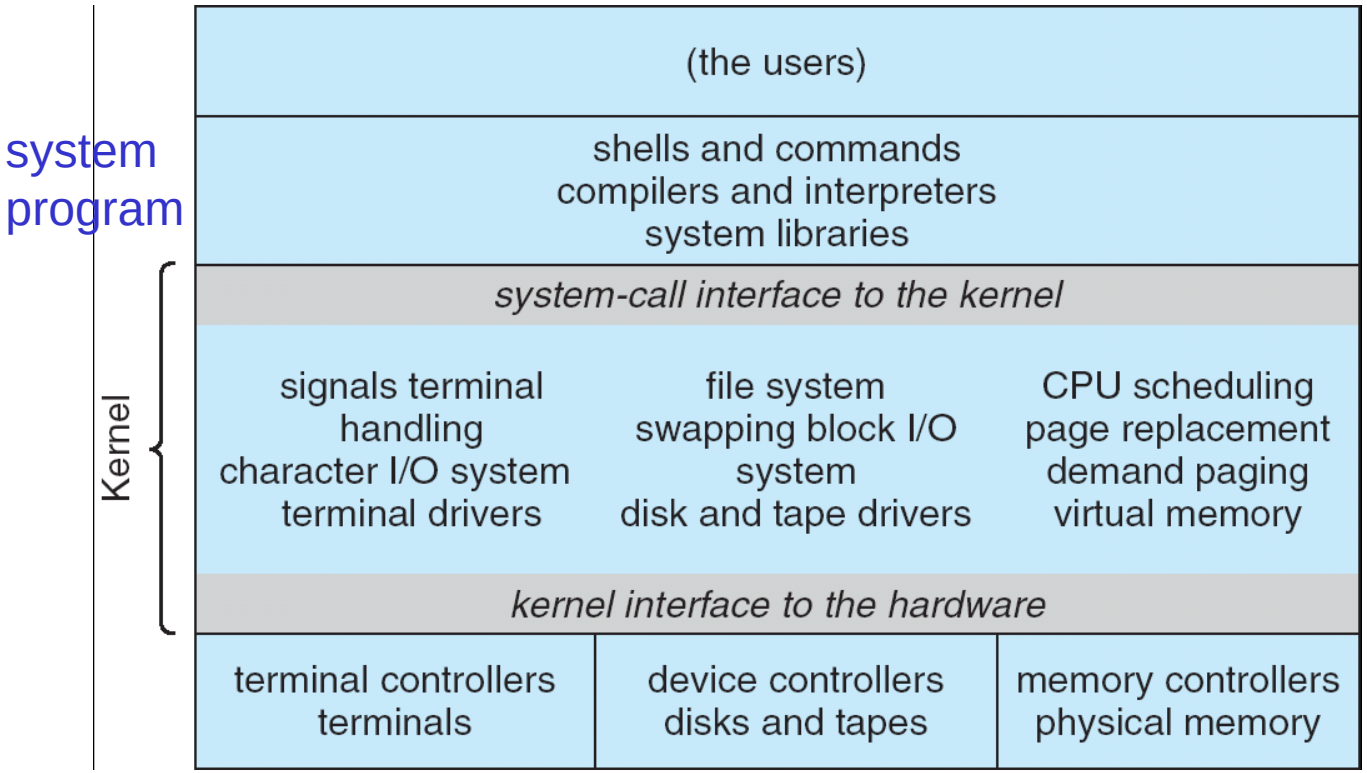
  → 악의적이거나 오류가 있는 프로그램에 취약함



# Simple Structure - UNIX

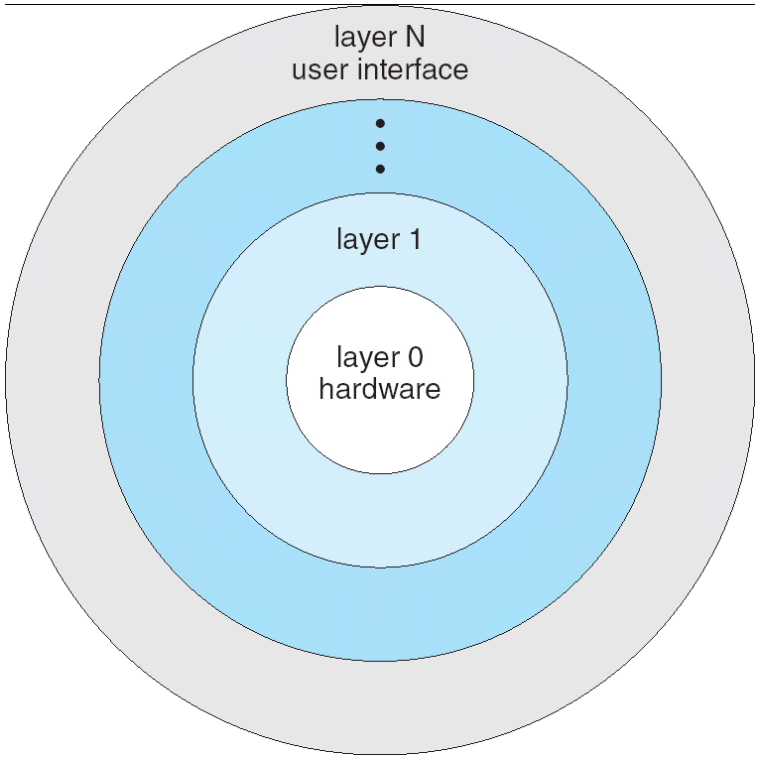
- UNIX
  - 초기에 하드웨어 기능에 제한이 있었으며 제한된 구조를 가짐
- UNIX의 구성
  - Kernel과 System program으로 구성
  - Kernel은 여러 인터페이스와 장치 드라이버로 분리되어 확장됨
- UNIX Kernel
  - the system-call interface 아래와  physical hardware 위의 모든 부분
  - 한 계층에서 많은 기능을 제공
    - the file system, CPU scheduling, memory management, and other operating-system functions
- monolithic 구조의 장단점
  - 장점 – 성능(커널 내부 통신 오버헤드가 거의 없음)
  - 단점 –  구현과 유지보수가 어려움

# UNIX System Structure



# Layered Approach

- Layered 구조
  - 최하위 계층(layer 0) = 하드웨어
  - 최상위 계층(layer N) = 사용자 인터페이스
- Modularity
  - 각 계층은 하위계층에서 제공하는 함수와 서비스만 사용함

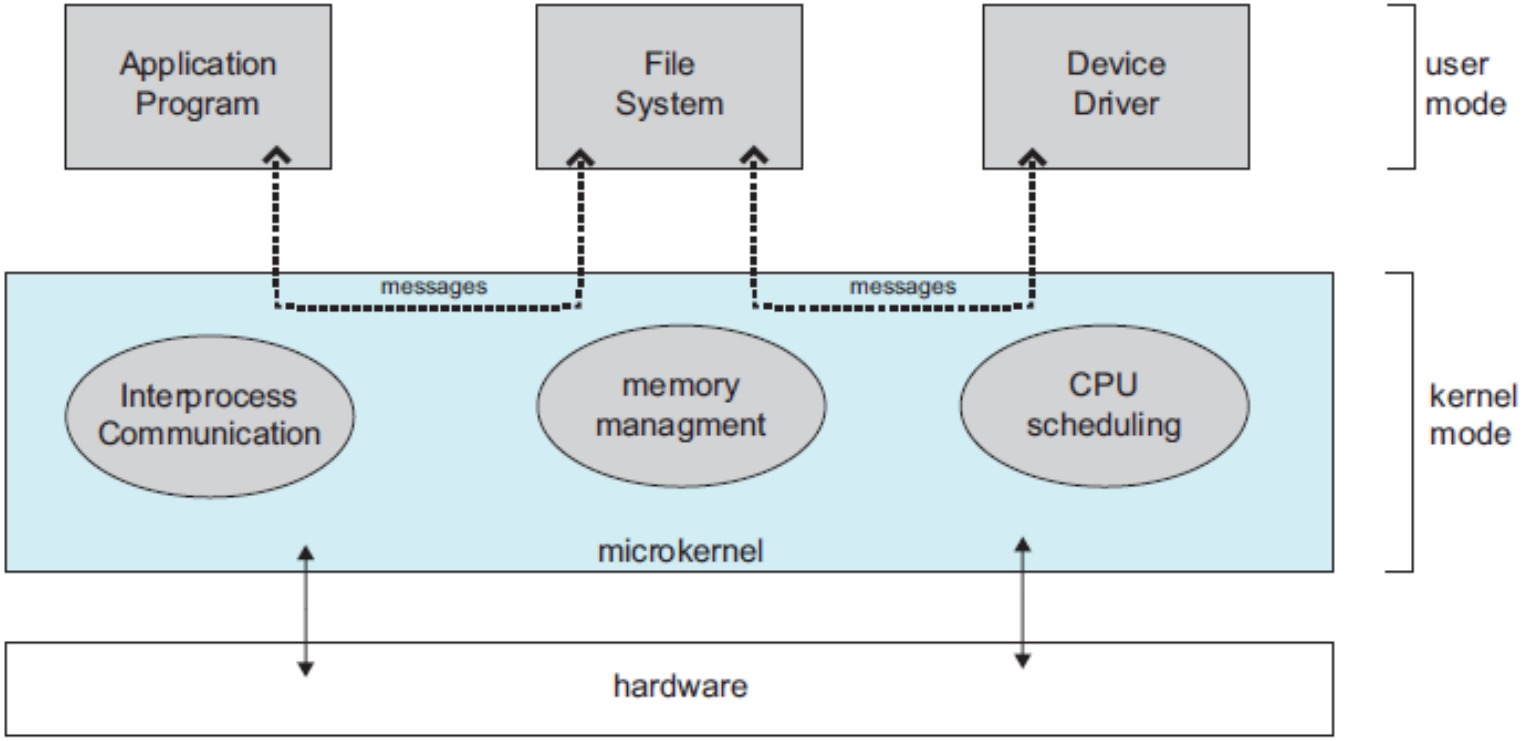


# Layered Approach의 장단점

- 장점 – 구현과 디버깅이 간단함
  - 하위계층의 연산이 어떻게(how) 구현 되었는지 알 필요가 없음
  - 하위계층 연산이 무슨(what) 동작을 하는 지만 알면 됨
- 단점
  - 각 계층을 적절히 정의하는 것이 어렵고, 명확하지 않을 수 있음
  - 덜 효율적 – 계층별 시스템 호출 오버헤드
  - 이러한 제한점으로 최근에 이 방법에 대한 부정적 평가
    → 많은 기능을 가진 적은 수의 계층으로 설계

# Microkernels

- Microkernel
  - 커널의 필수적이 아닌 많은 부분을 사용자 공간으로 이동 → small kernel
- Microkernel의 기본 기능
  - 사용자 모듈과 사용자 공간에서 수행하는 서비스 간에  통신 기능 제공
  - **message passing** 사용



# Microkernel의 장단점

- Microkernel 개발 이유
  - UNIX 커널이 확장되면서 관리하기가 어려워짐
  - Carnegie Mellon University (CMU)에서 이를 해결하기 위해 microkernel 방식의 Mach 커널 개발
    - Tru64 UNIX, MacOS X (Darwin kernel), QNX에서 사용
- 장점:
  - 확장이 용이 – 새로운 서비스는 사용자 공간에 추가
  - 운영체제 이식이 용이 – 작은 커널이므로 변경 부분이 적음
  - 높은 신뢰성과 보안성
    - 커널 모드에서 수행되는 코드가 적고
    - 대부분의 서비스는 사용자 프로세스로 실행됨

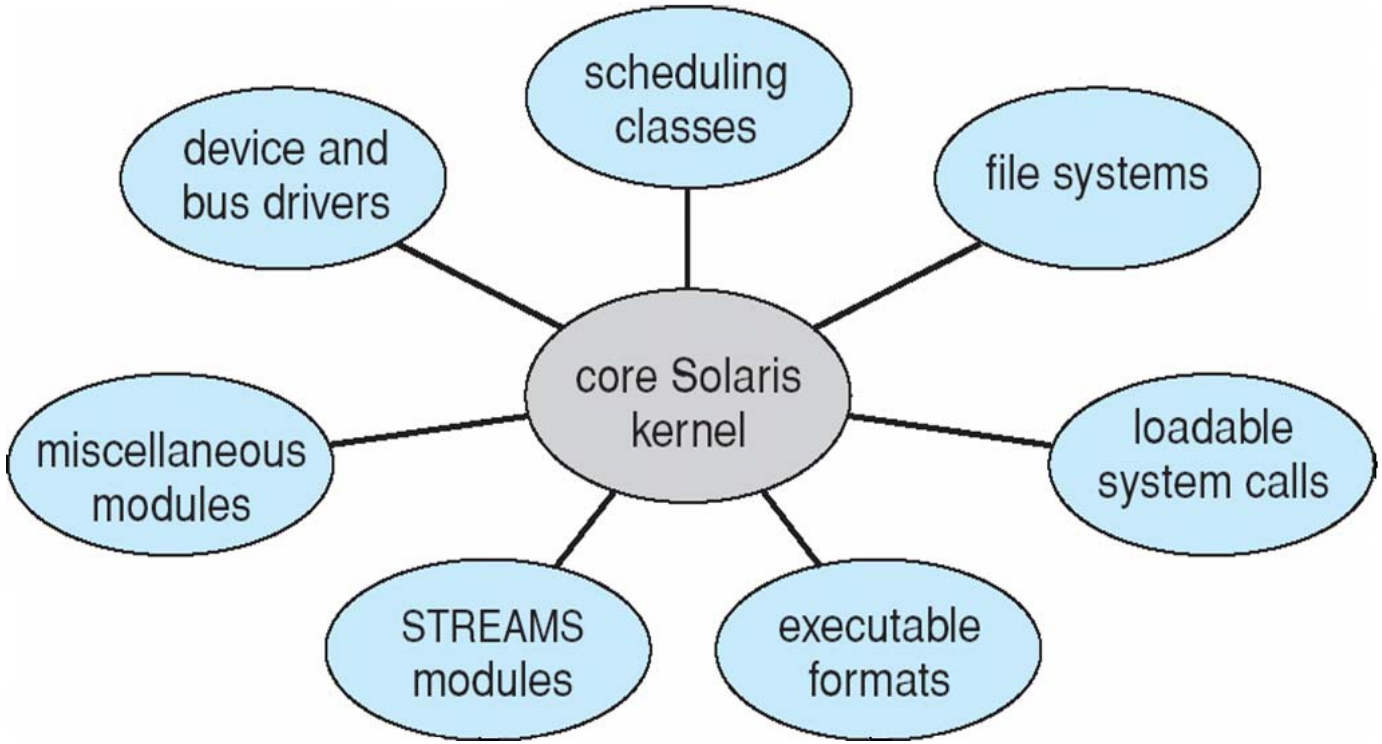
    → 서비스가 잘못되더라도 다른 부분에 영향이 없음
- 단점
  - 시스템 함수 오버헤드 – 사용자 공간과 커널 공간 간의 통신 오버헤드

# Modules

- kernel 구성
  - a set of core components + loadable kernel modules
- loadable kernel module
  - 부팅 또는 실행시간 동안 동적으로 적재되어 커널 기능을 확장함
  - 대부분의 현대 운영체제(Solaris, Linux 포함)에서 구현됨
- 모듈 인터페이스의 특징
  - 각 core component가 분리됨
  - 알려진 인터페이스를 통하여 다른 component/module과 통신을 함
- layered 구조와 유사하지만 더 유연성이 있음
  - 잘 정의되고 보호된 인터페이스를 가지는 점에서 layered 구조와 유사
  - 모듈은 임의의 다른 모듈을 호출할 수 있으므로 유연성이 있음
- microkernel approach과 유사하지만 더 효율적
  - 핵심 모듈은 핵심 기능만 가지고 있는 점에서 microkernel 구조와 유사
  - 모듈이 message passing을 사용하지 않으므로 더 효율적

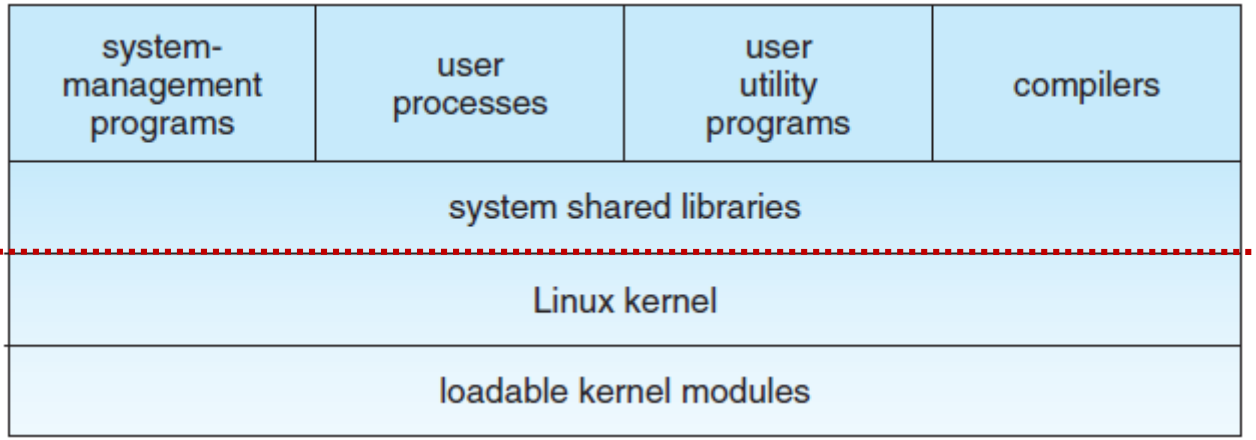
# 예: Solaris loadable modules

7 loadable kernel modules

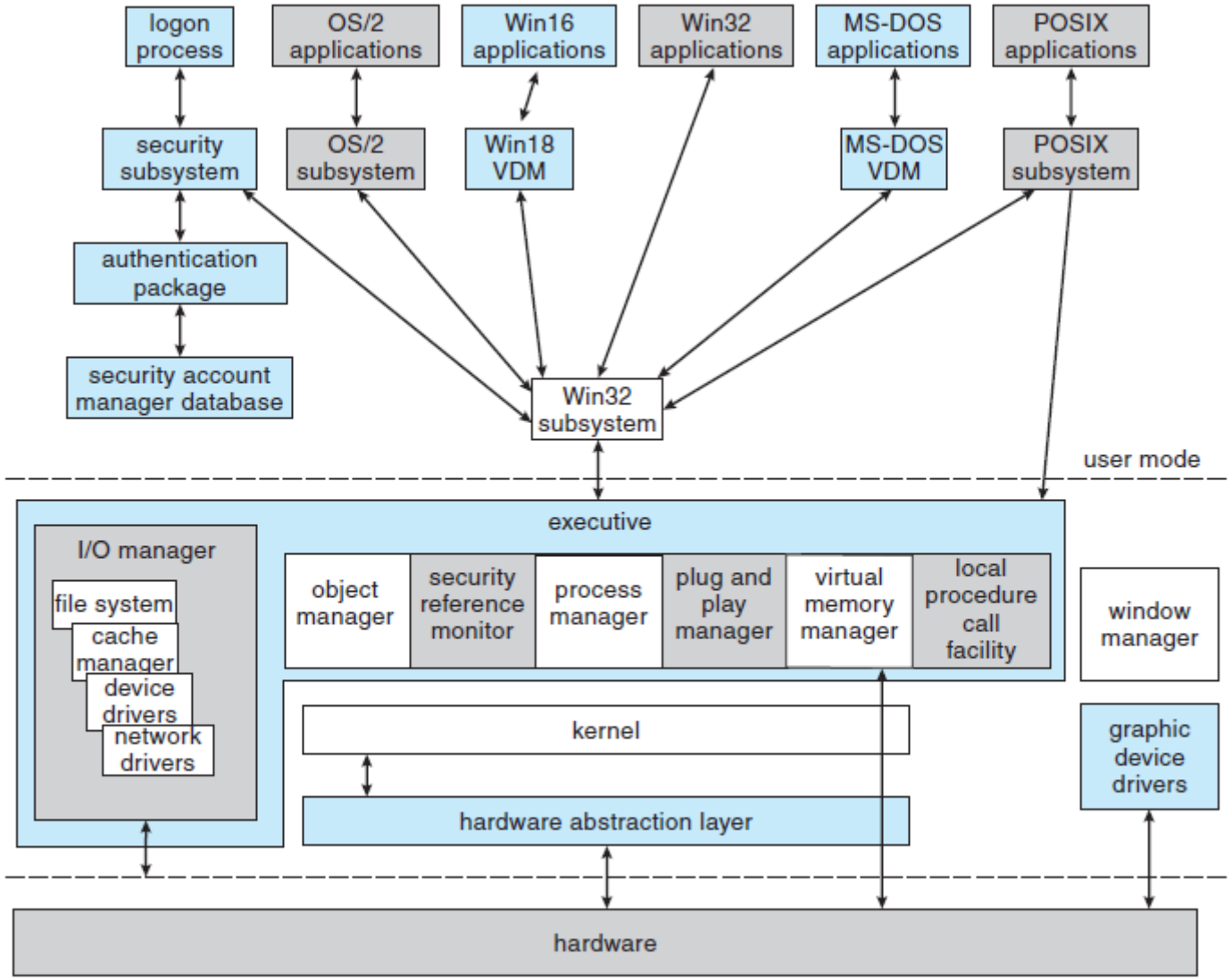


# Hybrid Systems

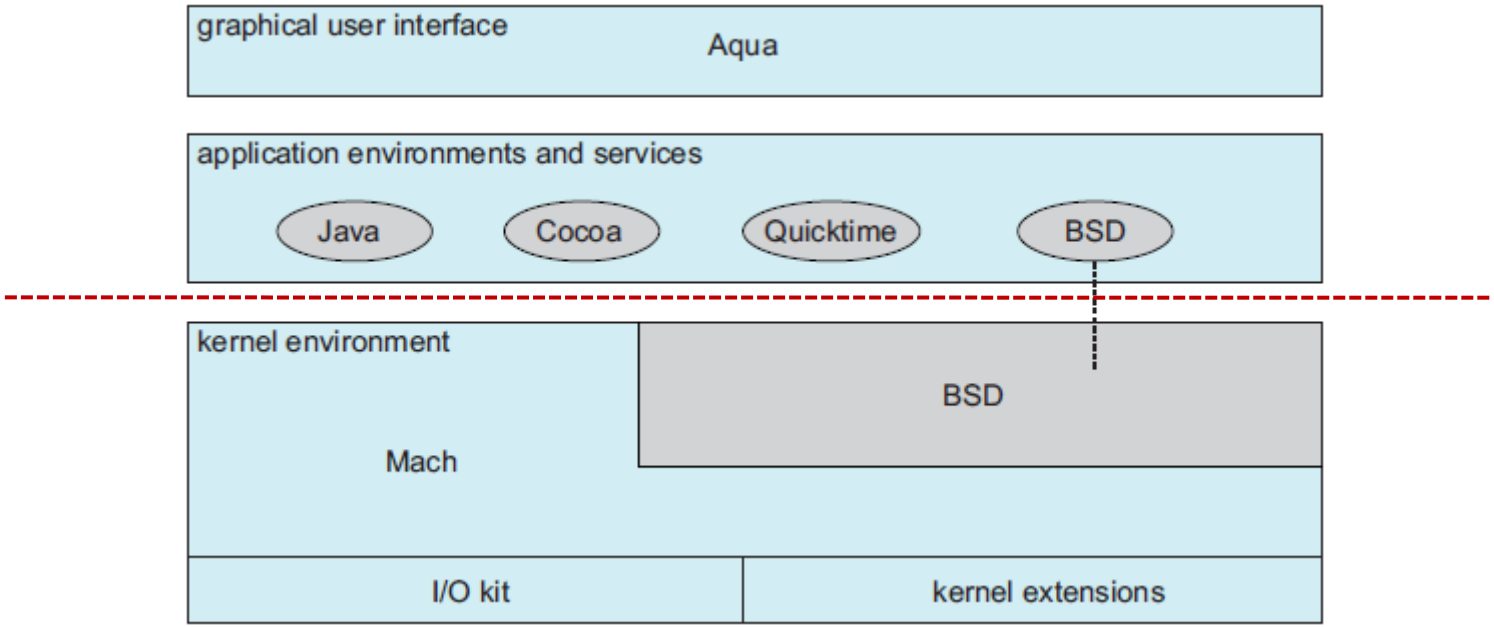
- Linux



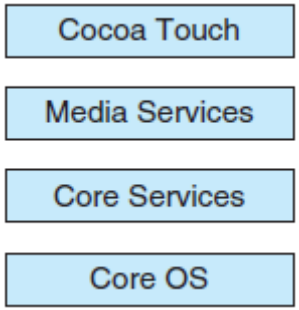
## Windows


logon process
OS/2 applications
Win16 applications
Win32 applications
MS-DOS applications
POSIX applications
security subsystem
OS/2 subsystem
Win18 VDM
MS-DOS VDM
POSIX subsystem
authentication package
security account manager database
Win32 subsystem
user mode
executive
I/O manager
file system
cache manager
device drivers
network drivers
object manager
security reference monitor
process manager
plug and play manager
virtual memory manager
local procedure call facility
window manager
kernel
graphic device drivers
hardware abstraction layer
hardware

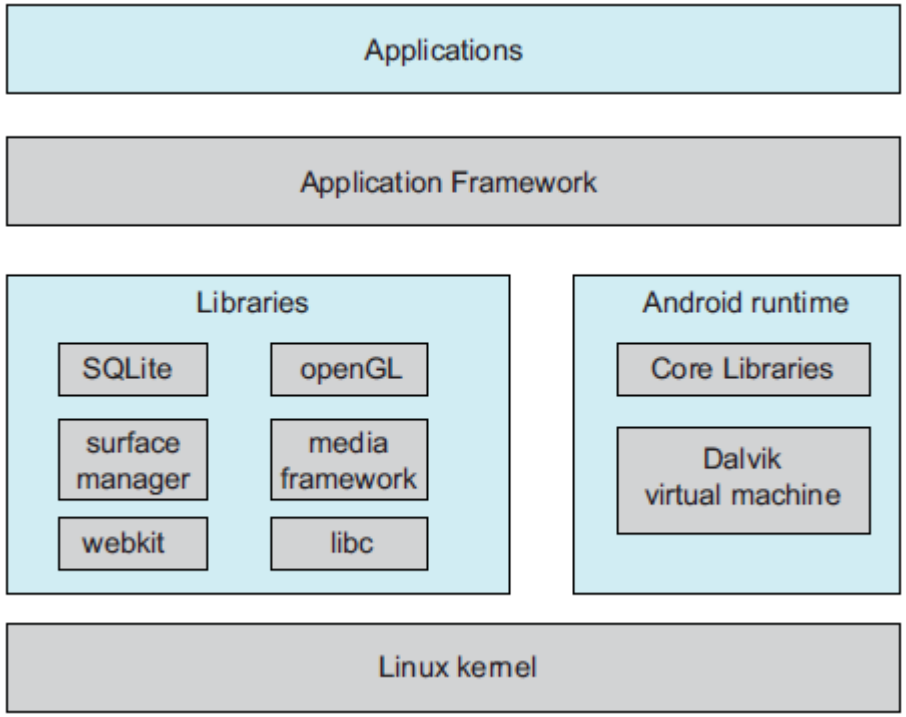
## Mac OS X


graphical user interface
Aqua
application environments and services
Java
Cocoa
Quicktime
BSD
kernel environment
BSD
Mach
I/O kit
kernel extensions

## iOS


Cocoa Touch
Media Services
Core Services
Core OS

## Android


Applications
Application Framework
Libraries
SQLite
openGL
surface manager
media framework
webkit
libc
Android runtime
Core Libraries
Dalvik virtual machine
Linux kernel

# 2.8 운영체제 디버깅

- 디버깅(debugging)
  - 오류(bug라고도 함)를 발견하고 수정하는 것
- 장애(Failure) 분석
  - log file – 프로세스가 실패하면 오류 정보를 기록
  - core dump 파일 – 오류 발생 프로세스의 메모리 내용을 저장
    - 초창기의 컴퓨터에서 메모리를 core라고 부름
  - crash dump file – 커널 장애시에 커널 메모리 내용을 저장
    - 커널 장애를 충돌(crash)라고 부름
- 성능 조정
  - 병목 지점(bottleneck)을 제거하여 시스템 성능을 최적화 가능
  - OS의 bottleneck을 발견하기 위하여 시스템 성능 측정 및 표시 유틸리티를 사용해야 함 (ex) top (unix), windows task manager
- *DTrace*
  - 사용자 프로세스와 커널에 동적으로 탐색점을 추가할 수 있는 유틸리티
  - Solaris, FreeBSD, Mac OS X에서 제공됨

# Solaris 10 dtrace – System Call을 추적

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                       U
  0   -> _XEventsQueued                    U
  0     -> _X11TransBytesReadable          U
  0     <- _X11TransBytesReadable          U
  0     -> _X11TransSocketBytesReadable    U
  0     <- _X11TransSocketBytesreadable    U
  0     -> ioctl                           U
  0       -> ioctl                         K
  0         -> getf                        K
  0           -> set_active_fd             K
  0           <- set_active_fd             K
  0         <- getf                        K
  0         -> get_udatamodel              K
  0         <- get_udatamodel              K
...
  0         -> releasef                    K
  0           -> clear_active_fd           K
  0           <- clear_active_fd           K
  0           -> cv_broadcast              K
  0           <- cv_broadcast              K
  0         <- releasef                    K
  0       <- ioctl                         K
  0     <- ioctl                           U
  0   <- _XEventsQueued                    U
  0 <- XEventsQueued                       U
```

# 2.9 운영체제 생성

- 운영체제 구성(Configuration)
  - 운영 체제는 다양한 구성을 가진 모든 종류의 시스템에서 실행되도록 설계되었다.
  - 운영 체제는 특정 컴퓨터를 위해 구성(configuration)되어야 함
- 운영체제 생성 (SYSGEN)
  - 운영체제 배포 – 디스크, CD-ROM, DVD-ROM, 또는 ISO image 형태
  - 하드웨어 구성 정보(CPU type, memory size, devices, OS options) – 파일을 읽거나, 직접 하드웨어를 검사하여 얻음
  - 운영체제 3가지 생성 방법
    - 소스코드 수정 후 완전히 다시 컴파일
    - 구성 테이블 생성 → 미리 컴파일된 라이브러리에서 필요한 모듈 선택 후 링크 – 코드가 재컴파일 되지 않음.
    - 완전한 테이블 방식 – 모듈 선택이 실행 시에 일어남

# 2.10 시스템 부트

- Bootstrapping(booting)
  - 커널을 적재하여 컴퓨터를 시작하는 절차
- Bootstrap loader
  - 커널을 찾아서 메모리에 적재하고 수행을 시작하는 일을 하는 ROM에 저장된 코드
  - 대개 시스템을 진단(diagnostic)하는 작업을 수행하고, 시스템 전체를 초기화 한 후에 운영체제를 시작시킴
- Firmware
  - bootstrap 코드의 변경이 가능하도록 ROM대신 EPROM에 저장함
  - 하드웨어와 소프트웨어의 중간 특성을 가져서 firmware라고 부름
  - 실행속도가 RAM에서 실행하는 것보다 느려서 대개 빠른 수행을 위해서 RAM에 복사하여 실행함
- 운영체제는 대개 디스크에 저장하지만, 일부 시스템은(smartphone, game console 등) 운영체제 전체를 firmware(EPROM/flash메모리)에 저장함

# 시스템 부트 과정

- 다단계 부팅 과정
  1. bootstrap loader :  reset location in ROM
     - 부트 디스크의 boot block을 메모리에 적재 후 실행
  2. simple boot code : single block at block 0 of a boot disk.(boot block)
     - 나머지 bootstrap 프로그램(full bootstrap 프로그램)에 대한 디스크 주소와 길이만 알고 있는 간단한 코드
  3. complex boot code (full bootstrap program)
     - 파일 시스템을 탐색하여 OS 커널을 찾고, 메모리에 적재하여 실행을 시작
     - (예) GRUB (Grand Unified Bootloader)
  4. operating system kernel
     - 운영체제가 커널을 실행하고 프로세스를 시작함
- UEFI (Unified Extensible Firmware Interface)
  - BIOS를 대체할 목적으로 개발됨
  - 파일 시스템을 지원하여 boot sector에 의존없이 OS 커널 부팅 가능