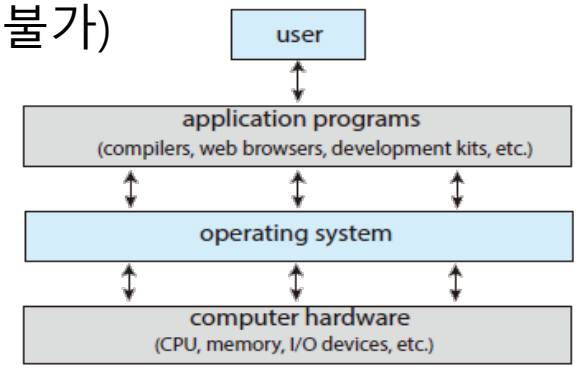
2.6 Why Applications Are Operating-System Specific

- Applications **compiled** on one OS are **not executable** on other OS.
  - (특정 OS 에서 컴파일된 응용은 다른 OS에서 실행 불가)
  - Each OS provides a unique set of **system calls**.
  - System calls are **part of the set of services** provided by OSs for use by applications.
    - (system call은 OS가 제공하는 서비스 집합의 일부)



- Application이 여러 OS에서 사용 가능하도록 하려면? 소스 코드 수준
  1. The application can be written in an **interpreted language** (such as **Python** or **Ruby**) that has an interpreter available for multiple OSs.
     - (interpreter 언어로 작성; interpreter는 여러 OS에서 사용 가능)
     - The interpreter **reads each line** of the source program, **executes equivalent instructions** on the native instruction set, **and calls** native OS calls.
       - (소스 프로그램의 line을 읽어서 상응하는 유사 기계어를 실행, OS call 호출)
     - **Performance** suffers relative to that for native applications, and
     - the **interpreter** provides **only a subset of each OS's features**, possibly **limiting the feature sets** of the associated applications.
       - (기계어로 구성된 native application에 비해 성능이 떨어지고, interpreter가 OS 기능의 일부만 제공하므로 관련 application의 기능이 제한됨)

2.6 Why Applications Are Operating-System Specific

- Application이 여러 OS에서 사용 가능하도록 하려면?
  - ② The application can be written in a language that includes a **virtual machine** containing the running application. (가상 머신 포함 언어)
    - 예) **Java**
    - The **virtual machine** is part of the language's full **RTE**. (RTE의 일부)
    - Java has an **RTE** that includes a **loader, byte-code verifier, and other components** that load the Java application into the Java virtual machine.
      - (java 응용을 java 가상 머신으로 load함)
    - **This RTE has been ported**, or developed, **for many OSs**, from mainframes to smartphones, and in theory **any Java app can run within the RTE** wherever it is available.
      - (많은 OS에 RTE 이식; 어떤 java 응용이든 RTE가 제공되면 실행 가능)
    - Systems of this kind have **disadvantages** similar to those of interpreters, discussed above.
      - (interpreter 와 같은 성능 저하 문제)

2.6 Why Applications Are Operating-System Specific

- Application이 여러 OS에서 사용 가능하도록 하려면?
  - ③ The application developer can use a **standard language** or **API** in which the compiler generates **binaries** in a machine- and OS- specific language. (어디서든 컴파일 가능)
    - (표준 언어나 API를 사용하여 compiler가 해당 machine-/OS-specific 언어로 이진 파일을 생성하도록 함)
    - The **application must be ported** to each OS on which it will run.
    - This **porting can be quite time consuming** and must be done for **each new version** of the application, with subsequent **testing and debugging**.
      - (실행될 OS에 port; 시간이 소요되고, 새 version마다 port 필요)
    - 예) **POSIX API and its set of standards**
      - for maintaining source-code compatibility
      - between different variants of **UNIX-like OSs**.

2.6 Why Applications Are Operating-System Specific

- Cross-platform applications 개발이 쉽지 않은 추가적 원인
  - **At the application level**
    - the **libraries** provided with the OS **contain APIs** to provide features like GUI interfaces, and (OS와 함께 제공되는 library는 GUI와 같은 기능 제공을 위해 API)
    - an **application** designed to call **one set of APIs** (say, those available from iOS on the Apple iPhone) **will not work** on an OS that does **not provide those APIs** (such as Android).
      - (한 API set을 호출하도록 설계된 응용은 이를 제공하지 않는 OS에서는 작동 안함)
  - **At lower levels**
    - **Each OS has a binary format** for applications that dictates the layout of the header, instructions, and variables. (OS별 이진 형식)
      - Those components need to be **at certain locations** in **specified structures** within an executable file so the **OS can open** the file and **load** the application for proper execution.
      - (실행 파일의 명시된 구조 내 특정 위치에 존재; OS가 open/load함)
    - CPUs have **varying instruction sets**, and **only applications** containing the appropriate instructions **can execute correctly**.
      - (CPU 명령 set이 다양; 해당 명령어를 포함하는 응용들만 제대로 수행됨)
    - OSs provide **system calls** that allow applications to **request various activities**, such as creating files and opening network connections.
      - Those **system calls vary** among OSs in many respects.
      - (OS가 application이 다양한 activity를 요청하는 system call을 제공함)

2.6 Why Applications Are Operating-System Specific

- Architectural difference(구조적 차이)를 해결하기 위한 노력
  - **Some approaches** that have helped address, though **not completely solve**, these architectural differences.

이진 실행 파일 수준

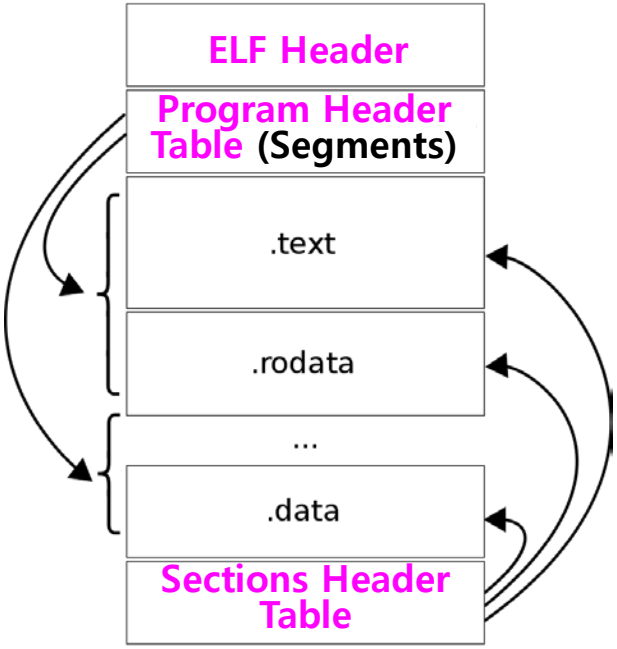
① **Linux** has adopted the **ELF format** for **binary executable files**.

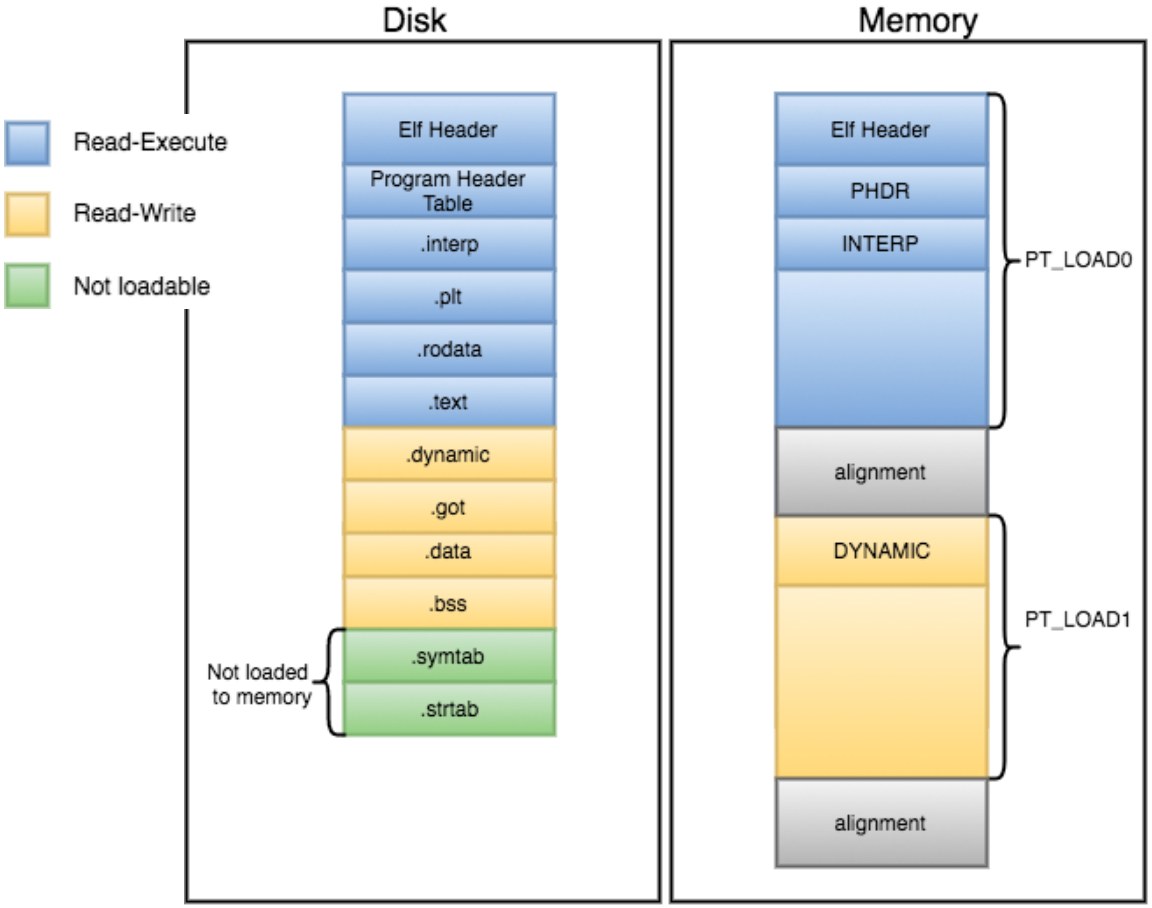
- **ELF**(Executable and Linkable Format)
  - include the **compiled machine code** and a **symbol table** containing **metadata** about functions and variables.
- Although **ELF** provides a **common standard** across Linux and UNIX systems, the **ELF format is not tied to any specific computer architecture**,
- so it does **not guarantee** that an executable file will run across different hardware platforms.
  - (ELF가 공통 표준을 제공하지만 특정 구조에 대한 언급이 아니라 다른 하드웨어 플랫폼에서 실행됨을 보장 못함)

63

2.5 Linkers and Loaders

- **ELF Header** : 바이너리에 대한 **일반적 정보**
- **Sections** : 작업 실행 파일을 빌드하기 위해 오브젝트 파일을 **링크하는 데 필요한 모든 정보** 포함 (링크 타임에 필요, 런타임에는 필요 없음)
- **Program Header (Segments)** : ELF 바이너리의 구조를 적절한 청크로 분류하여 실행 파일이 메모리에 로드되도록 준비 (링크 타임에는 프로그램 헤더가 필요 없음)





2.6 Why Applications Are Operating-System Specific

- Architectural difference(구조적 차이)를 해결하기 위한 노력
  - ② An **application binary interface (ABI)** is used.
    - to define how different components of **binary code** can **interface** for a given OS on a given architecture.
      - (이진코드의각구성요소들이주어진architecture 및 OS에서어떻게 interface하는지를정의)
    - specifies **low-level details**, including **address width**, methods of **passing parameters** to system calls, the **organization** of the run-time stack, the **binary format** of system libraries, and the **size** of data types, just to name a few.
    - is specified for a **given architecture.** (예: **ABI** for the **ARMv8 processor**)
    - If a **binary executable file** has been **compiled** and **linked** according to a **particular ABI**, it should be able to **run on different systems** that support that ABI.
      - (ABI에 따라 compile/link 했다면, ABI를 지원하는 다른 system에서 실행됨)
    - However, because a **particular ABI is defined for a certain OS** running on a given architecture, **ABIs do little** to provide cross-platform compatibility.
      - (특정 architecture에서 실행되는 특정 OS에 대해 정의; cross-platform 간 호환성을 제공하지 않음)

2.6 Why Applications Are Operating-System Specific

- 요약
  - Unless an interpreter, RTE, or binary executable file is **written** for and **compiled** on a **specific OS** on a **specific CPU** type (Intel x86 or ARMv8), the application will **fail to run**.
    - (interpreter, RTE, 이진파일이 특정 CPU type, 특정 OS에서 작성되고 compile 되지 않는다면 그 응용은 실행되지 않음)
  - Imagine the **amount of work** that is required for a program such as the **Firefox browser** to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.
    - (다양한 CPU, 다양한 OS에서 실행되는 browser에 요구되는 작업의 양은 어마함)

2.7 Operating System Design and Implementation

- Design and Implementation of OS
  - not "complete solutions", but some approaches have proven successful

2.7.1 Design Goals 시스템이 이러해야 한다!

- At the highest level, The **design** of the system
  - will be **affected by the choice** of **hardware and the type of system**.
  - traditional desktop/laptop, mobile, distributed, or real time.
- Beyond this highest design level, The **requirements** can be **divided**.

① **User goals**

- The system should be **convenient to use**, **easy to learn and to use**, **reliable**, **safe**, **and fast**.
- These specifications are **not particularly useful** in the system design, since there is **no general agreement** on how to achieve them.

② **System goals**

- A similar set of requirements can be **defined by the developers** who must design, create, maintain, and operate the system.
- The system should be **easy to design, implement, and maintain**; and it should be **flexible, reliable, error free, and efficient**.

- There is **no unique solution** for defining the requirements for an OS.

2.7 Operating System Design and Implementation

2.7.2 Mechanisms and Policies

- One important principle is the **separation** of **policy** from **mechanism**.
  - **Mechanisms** determine ***how*** to do something. (어떻게 할 것인가)
  - **Policies** determine ***what*** will be done. (무엇을 할 것인가)
  - 예) the timer construct (Section 1.4.3)
    - Timer is a **mechanism** for ensuring CPU protection, (CPU 보호를 위한 방법) deciding **how long the timer is to be set** for a particular user is a **policy** decision. (특정 사용자를 위해 타이머를 얼마나 길게/짧게 설정할 것인가)
  - 분리함으로써 융통성(**flexibility**)을 제공
  - **Policies** are likely to **change** across **places** or over **time**. (장소/시간에 따른 변경)
    - In the worst case, **each change** in policy would **require a change in the underlying mechanism**. (최악의 경우 정책에 따라 메커니즘의 변경)
    - A **general mechanism** flexible enough to **work across a range of policies** is preferable. (여러 정책에서 사용되기 충분한 융통성 있는 일반적 기법; 변경 최소화)
  - 예) **Mechanism for giving priority** to certain types of programs.
    - If the **mechanism** is properly separated from policy, it can be used either to support a **policy** decision that **I/O-intensive programs** should have priority over **CPU-intensive ones** or to support the **opposite policy**.

2.7 Operating System Design and Implementation

```
void func1() {
  ...
  if (시스템 에러)
     printf("Error! - errNo(%d)₩n", errNo);
}

void func2() {
  ...
  if (시스템 에러)
     printf("Error! - errNo(%d)₩n", errNo);
}
```

```
void func1() {
  ...
  if (시스템 에러)
     PrnErr(errNo);
}

void func2() {
  ...
  if (시스템 에러)
     PrnErr(errNo);
}

void PrnErr(int errNo) {
     printf("Error! - errNo(%d)₩n", errNo);
}
```

- **Mechanism** : 오류 상황을 파악하기위해 메시지 사용
- **Policy** : 시스템 에러 상황에서 Error! 메시지와 오류 번호 출력
  오류 번호만 출력
  정책이 바뀌어 시스템 에러 시 시스템 종료

2.7 Operating System Design and Implementation

2.7.2 Mechanisms and Policies

- OS에서의 적용
  - Microkernel-based OSs (Section 2.8.3) (분리)
    - **take the separation** of mechanism and policy to one extreme by implementing **a basic set of primitive building blocks**.
    - **These blocks** are almost **policy free**, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves.
  - Windows Microsoft (밀접)
    - has **closely encoded** both mechanism and policy into the system **to enforce a global look and feel** across all devices that run the Windows OS.
    - **All applications have similar interfaces**, because the interface itself is built into the kernel and system libraries.
- **Policy decisions** are important for all **resource allocation**.
  - Whenever it is necessary to decide **whether or not to allocate a resource**, a policy decision must be made.
    - (자원 할당 여부를 결정해야 할 때마다 Policy decision 필요) CPU를 적게 쓰는 작업 고르기
  - Whenever the question is ***how*** rather than ***what***, it is a **mechanism** that must be determined. CPU를 적게? or 많이 쓴 작업?

2.7 Operating System Design and Implementation

2.7.3 Implementation

- OS design 후 implementation!
  - **Early OSs** were written in **assembly language**.
  - **Now**, most are written in **higher-level languages(C or C++)** with small amounts of the system written in assembly language.
  - In fact, **more than one higher level language** is often used.
    - The **lowest levels of the kernel** might be written in **assembly** language and **C**.
    - **Higher-level routine**s might be written in **C and C++.**
    - **System libraries** might be written in **C++** or even **higher-level languages**.
  - 예) Android
    - **Its kernel** is written **mostly in C** with some assembly language.
    - Most Android **system libraries** are written in **C or C++,**
    - Its **application frameworks**, which provide the **developer interface** to the system, are written mostly in **Java**.
    - We cover Android's architecture in more detail in Section 2.8.5.2.

2.7 Operating System Design and Implementation

2.7.3 Implementation

- **Advantages** of implementing an OS in a **higher-level language**
  - The **code** can be written **faster**, is more **compact**, and is **easier** to understand and debug.
  - An OS is far **easier to port** to other hardware.
    - This is particularly **important for OSs** that are intended to **run on several different hardware systems**, such as small embedded devices, Intel x86 systems, and ARM chips running on phones and tablets.

- **Disadvantages** of implementing an OS in a **higher-level language**
  - are **reduced speed** and **increased storage requirements**.
  - However, not a major issue in today's systems.
    - For large programs a **modern compiler** can perform **complex analysis** and **apply sophisticated optimizations** that produce excellent code.
    - Modern processors have **deep pipelining** and **multiple functional units** that can handle the **details of complex dependencies** much more easily than can the human mind.

2.7 Operating System Design and Implementation

2.7.3 Implementation

- Major performance improvements in OSs are more likely to be the result of **better data structures and algorithms** than of excellent assembly-language code.
  - (OS의 성능은 우수한 assembly 코드보다 나은 데이터구조와 알고리즘의 결과임)
- Although OSs are large, only a **small amount of the code** is critical to high performance;
  - (OS의 일부 코드만 성능에 영향)
  - interrupt handlers, I/O manager, memory manager, and CPU scheduler are probably the most critical routines.
- After the system is written and is working correctly, **bottlenecks** can be identified and can be refactored to operate more efficiently.

2.8 Operating System Structure

- A system as **large and complex** as a modern OS must be **engineered carefully** if it is to function properly and be modified easily.
  - (제대로 동작하고 쉽게 수정되도록 신중히 제작되어야 함)

- A common approach is to **partition** the **task** into **small components, or modules**, rather than have one single system.
  - Each of these **modules** should be a **well-defined portion** of the system, with carefully defined interfaces and functions.
    - (신중히 정의된 interface와 function을 가진 시스템의 잘 정의된 부분)
  - When you program :
    - rather than placing all of your code in the main() function, you instead **separate logic into a number of functions**.
    - (main 함수에 전체 코드를 넣는 대신, 여러 함수로 분리)

2.8 Operating System Structure

2.8.1 Monolithic Structure

- **The simplest** structure for organizing an OS is **no structure at all**.
  - Place **all of the functionality** of the kernel into **a single, static binary file** that **runs in a single address space**.
  - known as a **monolithic structure**
  - is a common technique for designing OSs.

- 예) original UNIX OS
  - Figure 2.12

2.8 Operating System Structure

2.8.1 Monolithic Structure

- 예) original UNIX OS
  - consists of **two separable parts**: the **kernel** and the **system programs**.
    - An **enormous amount of functionality** to be combined into one single address space.
    - (커널의 너무 많은 기능들이 하나의 주소 공간으로 결합함)

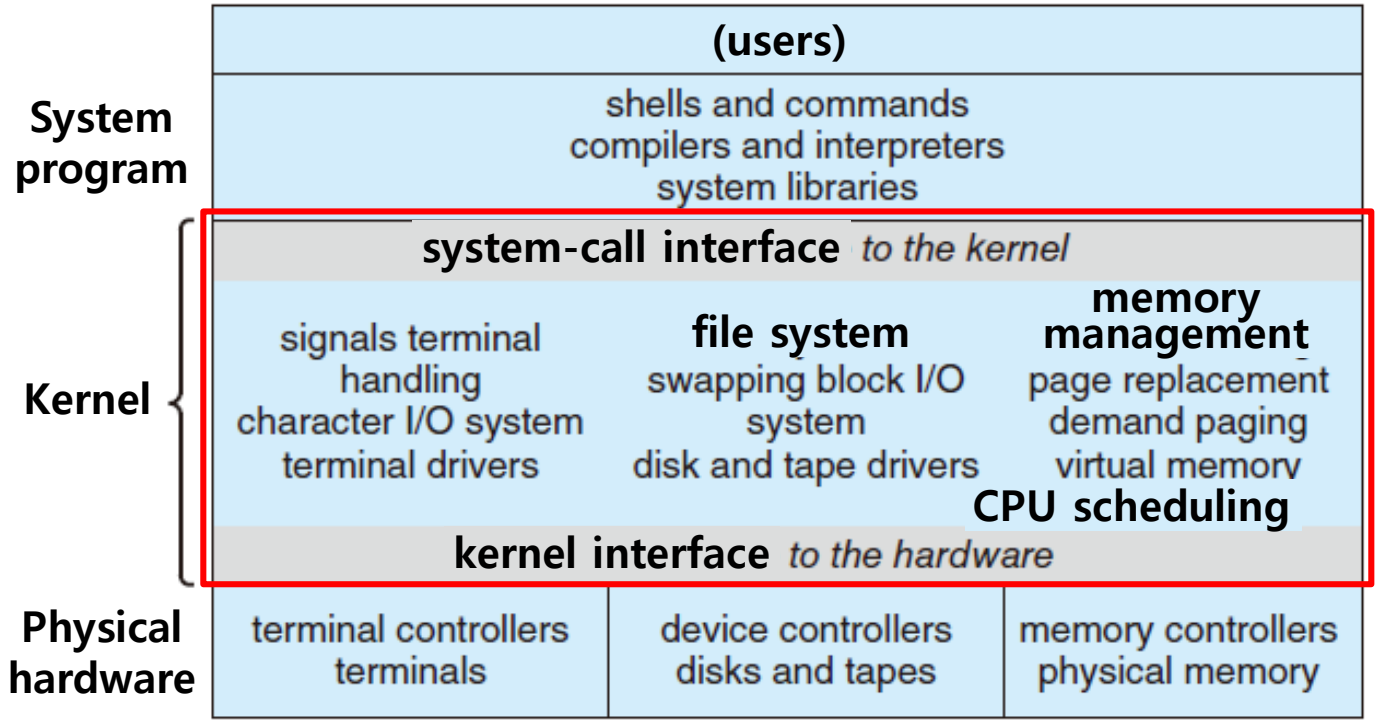


Figure 2.12 Traditional UNIX system structure

2.8 Operating System Structure

2.8.1 Monolithic Structure

- The **Linux OS** is based on UNIX and is structured similarly.
  - Figure 2.13
    - **Applications** typically use the **glibc standard C library** when communicating with the **system call interface** to the kernel.
  - The Linux kernel is **monolithic** in that it runs entirely in kernel mode in a single address space.
  - It does have a **modular design** that allows the **kernel** to be **modified** during run time.

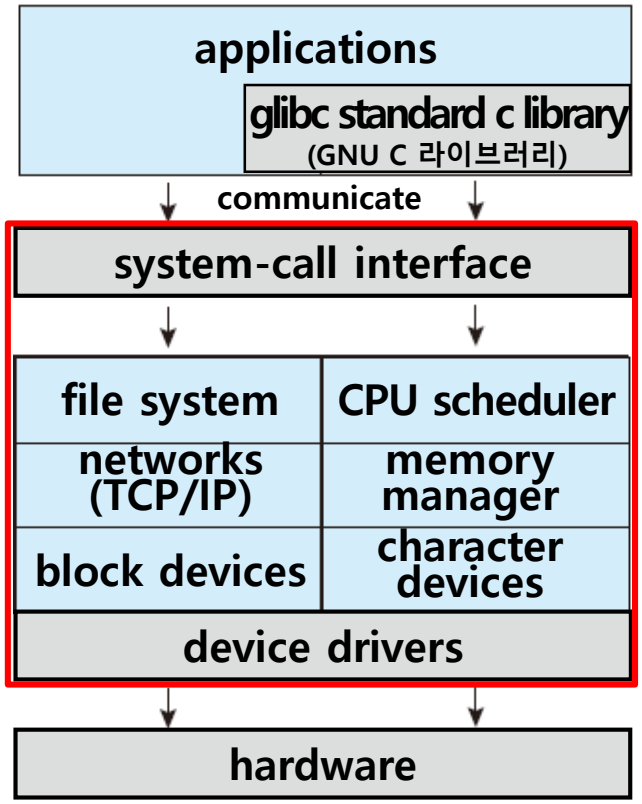


Figure 2.13 Linux system structure

2.8 Operating System Structure

2.8.1 Monolithic Structure

- Monolithic kernels are **difficult to implement and extend**.
  - (구현, 확장이 어려움)
- **Simplicity**
- **performance advantage** (성능에서의 이점)
  - There is very **little overhead** in the system-call interface, and **communication** within the kernel is **fast**.
  - (system-call interface에서 overhead가 거의 없고, 커널 내 통신이 빠름)
- Their **speed and efficiency** explains why we still see evidence of this structure in the UNIX, Linux, and Windows OSs.

- **tightly coupled system**
  - **monolithic approach,** because **changes** to one part of the system can have wide-ranging **effects** on other parts.
- **loosely coupled system**

2.8 Operating System Structure

2.8.2 Layered Approach

- Alternatively, we could design a **loosely coupled system**.
  - Such a system is **divided** into **separate, smaller components** that have specific and limited functionality. (시스템을 개별적, 작은 요소들로 구분)
  - All these components together comprise the kernel.
    - (모든 요소들이 커널을 구성)
- **Advantage**
  - **Changes** in one component **affect only that component**, and no others, allowing system implementers **more freedom** in creating and changing the inner workings of the system.
- A system can be made **modular** in many ways.
  - One method is the **layered approach**.
  - Figure 2.14
    - OS is broken into a **number of layers (levels)**.
    - The **bottom** layer (**layer 0**) is the **hardware**
    - The **highest** layer (**layer N**) is the **user interface**.

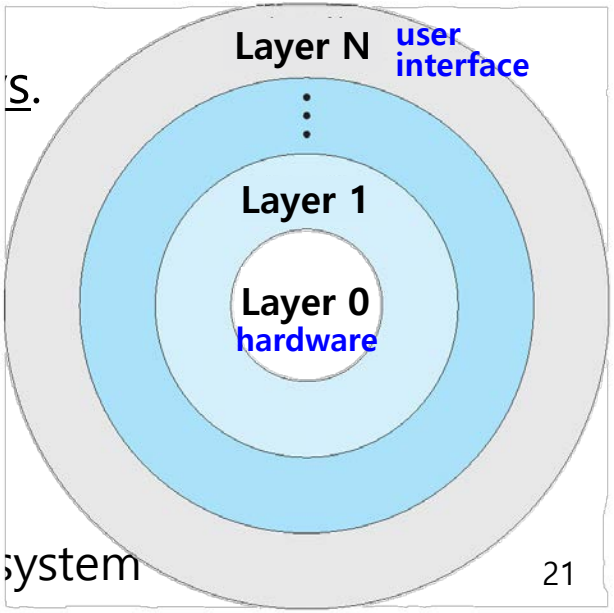


Figure 2.14
A layered operating system

2.8 Operating System Structure

2.8.2 Layered Approach

- An **OS layer** is an implementation of an **abstract object** (추상화 객체) made up of **data** and the **operations** that can manipulate those data.
  - A **typical OS layer(layer M)** consists of **data** structures and a set of **functions** that can be invoked by higher-level layers.
  - **Layer M**, in turn, can **invoke** operations on lower-level layers.
    - (계층은 data + function →상위 계층은 하위 계층의 연산(기능)을 호출)
- The main **advantage** of the layered approach
  - **simplicity** of construction and debugging.
    - The **first layer(layer 0)** can be debugged **without any concern** for the rest of the system, because it uses only the basic hardware (which is **assumed correct**) to implement its functions. (hardware는 정확하다고 가정)
      - (Layer 0는 hardware만 사용하여 구현; 다른 부분을 신경 쓰지 않고 디버깅함)
    - Once the first layer is debugged, **its correct functioning can be assumed** while the second layer is debugged, and so on.
      - (단계별로 이전/하위 단계가 정확하게 동작함을 가정)
    - **If an error is found** during the debugging of a particular layer, **the error must be on that layer**, because the layers below it are already debugged.
      - (error 발견 시 해당 층에서의 문제; 이전/하위 단계는 이미 디버거 되었음)

2.8 Operating System Structure

2.8.2 Layered Approach

- A layer does **not** need to know **how** these operations are implemented; **it needs to know only what these operations do**.
  - (계층별로 이전/하위 계층에서 제공하는 연산이 어떻게 동작하는지 몰라도 되며, 그 연산이 무엇을 하는지만 알면 됨)
  - Hence, **each layer hides** the existence of **certain data structures, operations, and hardware** from higher level layers.
    - (각 계층은 상위 계층에 특정 정보들을 숨김)
- Layered systems have been successfully used in **computer networks** (such as TCP/IP) and **web applications**.
- Nevertheless, **relatively few** OSs use a **pure layered approach**.

2.8 Operating System Structure

2.8.2 Layered Approach

- **Pure layered approach** 방식의 OS가 잘 사용되지 않는 이유
  1. **challenges** of appropriately **defining the functionality** of each layer.
     - (적절한 계층별 기능 정의의 어려움)
  2. **Overall performance** of such systems **is poor** due to the **overhead** of requiring a **user program** to traverse through multiple layers to obtain an OS service.
     - (OS 서비스를 위해 user program이 여러 계층을 거쳐야 하는 overhead; 전체적 성능 저하)

- **Some layering** is **common** in contemporary OSs, however.
  - (어느 정도의 계층화가 공통적임)
  - These systems have **fewer layers** with **more functionality**,
  - **providing** most of the advantages of modularized code while **avoiding** the problems of layer definition and interaction.
    - (모듈화 코드의 장점을 제공하면서, 계층 정의 및 상호작용의 문제점을 피함)

2.8 Operating System Structure

2.8.3 Microkernels

- As UNIX expanded, the **kernel became large and difficult** to manage.

- **Smaller kernel (Mach)**
  - In the mid-1980s, researchers at Carnegie Mellon University
  - **modularized the kernel** using the **microkernel approach**.
  - structures the OS by **removing all nonessential components** from the kernel and **implementing** them as **user level programs** that reside **in separate address spaces**.
    - (중요하지 않은 모든 요소들을 커널에서 제거하여 user level program으로 구현)
  - **Microkernels** provide minimal **process** and **memory** management, in addition to a **communication** facility.
  - Figure 2.15

2.8 Operating System Structure

2.8.3 Microkernels

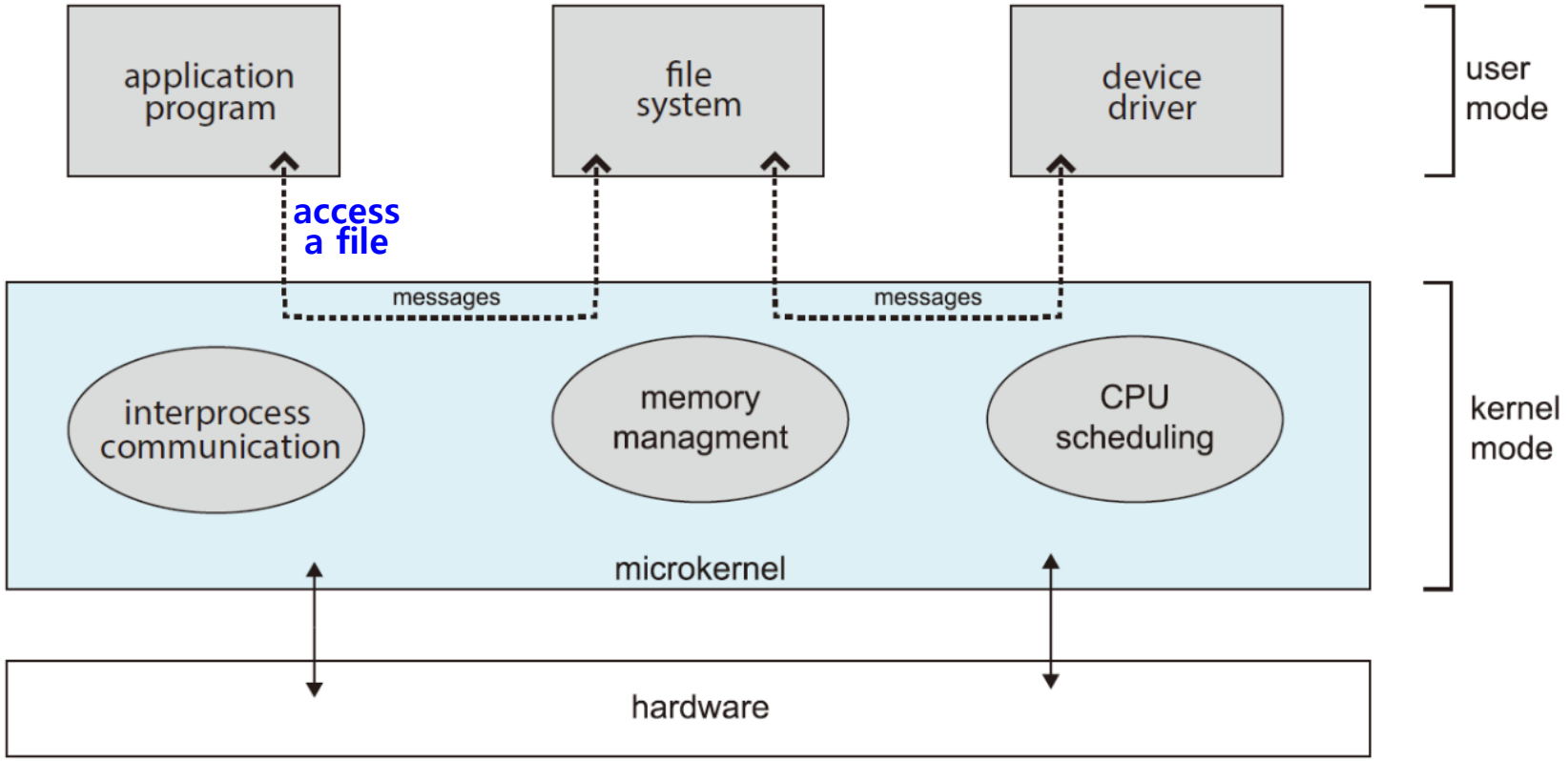


Figure 2.15 Architecture of a typical microkernel

2.8 Operating System Structure

2.8.3 Microkernels

- **Main function** of the microkernel
  - (message passing 을 통한 간접적인 통신을 제공)
  - provide **communication** between the client program and the various services that are also **running in user space**.
    - Communication is provided through **message passing**. (Section 2.3.3.5)
    - If the client program wishes to **access a file**, it must **interact** with the file server.
  - The client program and service **never interact directly**.
    - **They communicate indirectly** by exchanging messages with the microkernel.

- One **benefit** of the microkernel approach
  - It makes **extending** the OS **easier**. (OS 확장이 쉬움)
    - All new **services** are added to **user space** and consequently do **not require modification** of the kernel. (new service는 user space에 추가; 커널 수정 불필요)
  - The resulting OS is **easier to port** from one hardware design to another.
  - The microkernel also provides **more security and reliability**, since most services are running as user processes.
    - (대부분의 서비스가 user process로 실행되므로 보안/신뢰성 제공)

2.8 Operating System Structure

2.8.3 Microkernels

- Microkernel OS의 예
  - **Darwin**
    - Kernel component of the **macOS and iOS**
    - consists of **two kernels**, one of which is the **Mach microkernel**. (2.8.5.1)
  - **QNX**
    - a **real-time OS** for **embedded systems**.
    - The QNX Neutrino microkernel provides **services** for message passing and process scheduling.
    - handles **low-level network communication** and **hardware interrupts**.
    - **All other services** are provided by **standard processes** that run outside the kernel **in user mode**.

2.8 Operating System Structure

2.8.3 Microkernels

- Unfortunately, the **performance** of microkernels can **suffer** due to **increased system-function overhead**. (성능 문제)
  - ① When two user-level services must communicate, **messages must be copied** between the services, which reside in **separate address spaces**.
    - (두 사용자 서비스가 통신하는 경우, 서비스가 독립된 주소 공간에 존재하므로, 서비스 간 메시지가 복사되어야 함)
  - ② The **OS may have to switch** from one **process** to the next to exchange the messages.
    - (메시지 교환을 위한 프로세스 간 전환(switch) 필요)
  - The **overhead** involved in copying messages and switching between processes **has been the largest impediment** to the growth of microkernel-based OSs.
    - (두 가지 문제(overhead)가 microkernel 기반 OS의 성장에 장애가 됨)

2.8 Operating System Structure

2.8.3 Microkernels

- History of **Windows NT**
  - The **first release** had a **layered microkernel organization**.
  - This version's **performance** was **low** compared with that of Windows 95.
  - **Windows NT 4.0** partially corrected the performance problem by **moving** layers from user space to kernel space and **integrating** them more closely.

- By the time Windows XP was designed, **Windows** architecture had become more **monolithic** than microkernel.

2.8 Operating System Structure

2.8.4 Modules

- The **best current methodology for OS design** involves using **loadable kernel modules** (**LKM**s).
  - The **kernel** has a set of **core components** and can **link** in **additional services** via modules, either at **boot** time or during **run** time.
    - (커널이 핵심 요소들을 가지고, boot/run time 시 추가 서비스들을 모듈로 연결)
  - This type of design is **common** in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows. (현대적 OS 구현)
  - The idea of the design is for the **kernel** to provide **core services**, while **other services** are implemented **dynamically**, as the kernel is running.
    - (커널이 핵심 서비스 제공, 다른 서비스들은 커널 실행 동안 동적으로 구현)
    - **Linking services dynamically** is preferable to **adding** new features **directly** to the kernel, which would require **recompiling** the kernel every time a change was made.
      - (서비스 동적 연결이 새 기능을 커널에 직접 추가하는 것보다 나음; 직접 추가는 변경 시마다 recompile 필요)
  - 예) We might **build** CPU scheduling and memory management algorithms **directly** into the kernel and then **add support** for different file systems by way of **loadable modules**.

2.8 Operating System Structure

2.8.4 Modules

- Layered system과 유사
  - **Each kernel section** has defined, protected **interfaces**.
    - (각 커널 영역이 defined, protected 인터페이스를 가짐; 계층 구조와 유사)
  - But it is **more flexible** than a layered system, because **any module can call** any other module.
    - (임의의 다른 모듈을 호출할 수 있으므로 더 유연함)

- Microkernel approach와 유사
  - **Primary module** has only **core** functions and **knowledge** of how to load and communicate with other modules.
    - (주 모듈이 핵심 기능을 가지고 다른 모듈을 적재하고 통신하는 방법을 앎; 마이크로 커널과 유사)
  - But it is **more efficient**, because modules do **not** need to **invoke message passing** in order to communicate.
    - (message passing 사용하지 않으므로 더 효율적임)

2.8 Operating System Structure

2.8.4 Modules

- **Linux** uses **loadable kernel modules**, primarily for supporting **device drivers** and **file systems**.
  - (리눅스는 device driver, file system 지원을 위해 LKM 사용함)
  - LKMs can be "**inserted**" into the kernel as the **system is started** (or booted) or during **run time**, such as when a **USB device is plugged** into a running machine.
    - (LKM은 시스템 시작, 실행 중(USB 장치 삽입)에 insert 됨)
  - If the Linux kernel **does not have** the necessary driver, it can be dynamically loaded. (필요한 driver가 없다면, 동적으로 적재 가능)
  - LKMs can be **removed** from the kernel **during run time** as well. (동적 제거)
- For Linux, LKMs allow a **dynamic and modular kernel**, while maintaining the **performance benefits** of a monolithic system.
  - (동적 모듈식 커널을 허용하면서 monolithic system의 성능 이점을 유지함)

2.8 Operating System Structure

2.8.5 Hybrid Systems

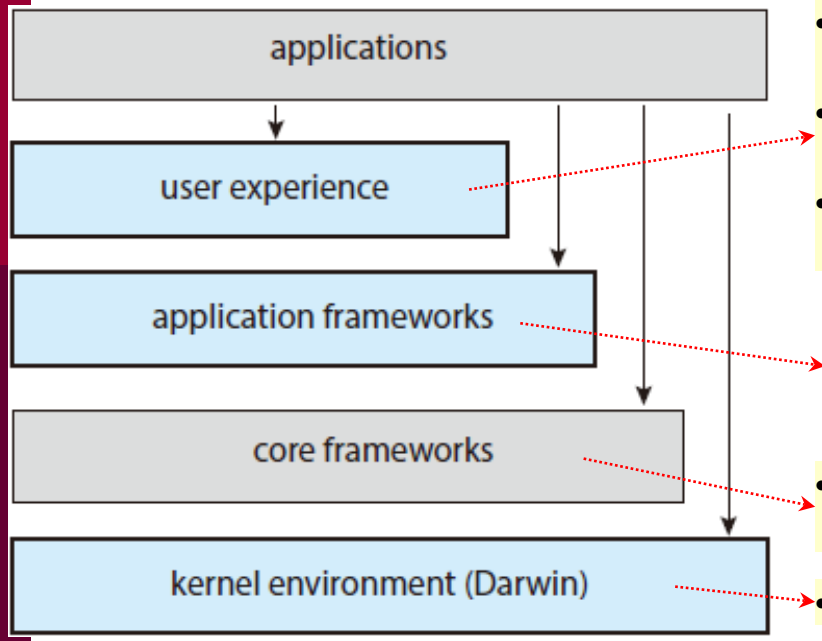
- In practice, very few OSs adopt a single, strictly defined structure.
  - (하나의 엄격히 정의된 구조를 채택한 OS는 거의 없음)

- Instead, they **combine** different structures, resulting in **hybrid systems** that address performance, security, and usability issues.
  - **Linux**
    - **monolithic**, because having the OS in a single address space provides very efficient performance.
    - also **modular**, so that **new functionality** can be **dynamically added** to the kernel.
  - **Windows**
    - is largely **monolithic** as well (again primarily for performance reasons),
    - but it retains some behavior typical of **microkernel** systems, including providing support for separate subsystems (known as OS personalities) that run as user-mode processes.
    - provide support for **dynamically loadable kernel modules**.

2.8 Operating System Structure

2.8.5 Hybrid Systems

- 2.8.5.1 Apple's macOS and iOS
  - **macOS OS** is designed to run primarily on **desktop and laptop** computer systems,
  - **iOS** is a **mobile OS** designed for the **iPhone smartphone** and **iPad tablet** computer.
  - Architecturally, **macOS and iOS** have much in **common**.



- defines the **software interface** that allows **users** to **interact** with the computing devices.
- **macOS** uses the **Aqua user interface** (designed for a **mouse or trackpad**)
- **iOS** uses the **Springboard user interface** (designed for **touch devices**)

- **provide an API** for the Objective-C and Swift programming languages. (응용 개발을 위한 API 제공)
- **Cocoa** for **macOS** applications, **Cocoa Touch** for **iOS**

- defines **frameworks** that support **graphics and media** including **Quicktime and OpenGL**.

- includes the **Mach microkernel** and the **BSD UNIX kernel**.

Figure 2.16 Architecture of Apple's macOS and iOS OSs

2.8 Operating System Structure

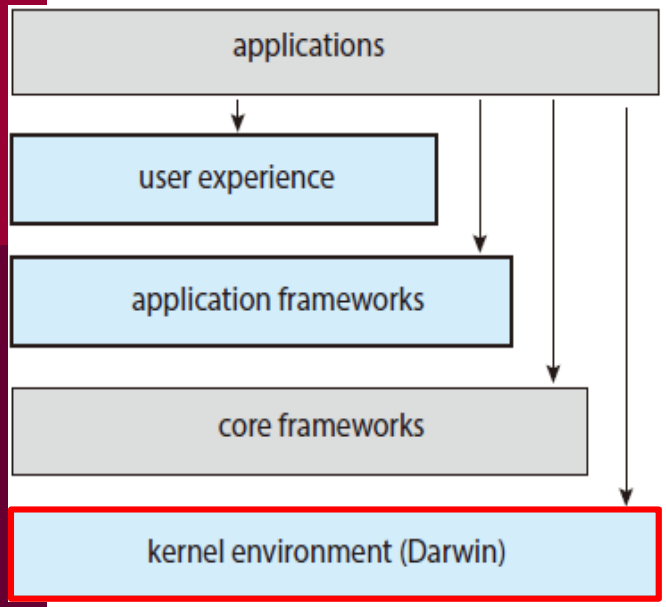
2.8.5 Hybrid Systems

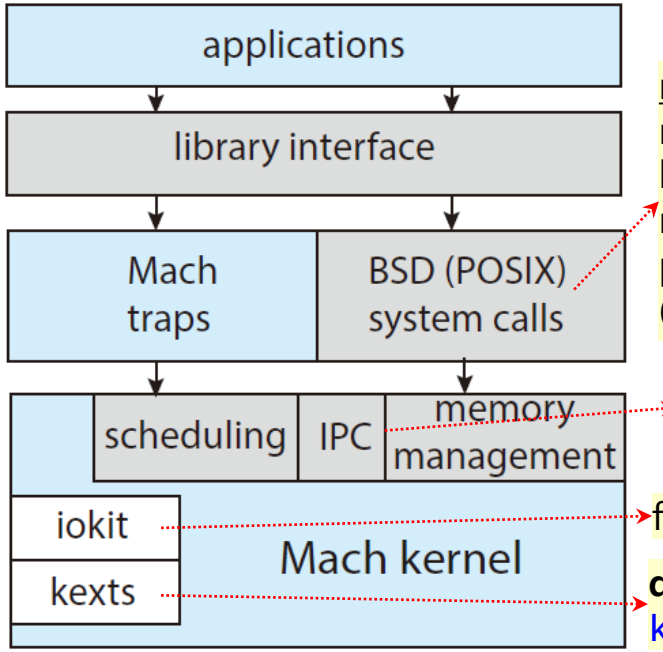
- 2.8.5.1 Apple's macOS and iOS
  - Some significant **distinctions** between **macOS and iOS**
    - Because **macOS** is intended for **desktop and laptop** computer systems, it is **compiled** to run on **Intel architectures**.
    - **iOS** is designed for **mobile devices** and thus is **compiled** for **ARM-based architectures**.
    - **iOS kernel** has been modified somewhat to address specific features and needs of **mobile systems**, such as **power management** and **aggressive memory management**.
    - **iOS** has **more stringent security settings** than macOS.
    - **iOS** is generally much **more restricted to developers** than macOS and may even be **closed to developers**. (개발자에게 더 제한적이고, 폐쇄적임)
    - For example, **iOS restricts access** to POSIX and BSD APIs on iOS, whereas they are openly **available** to developers on **macOS**.
      - (iOS 에서 POSIX, BSD API에 접근을 제한함)

2.8 Operating System Structure

2.8.5 Hybrid Systems

- 2.8.5.1 Apple's macOS and iOS
  - Darwin is a layered system that consists primarily of the **Mach microkernel** and the **BSD UNIX kernel**.
  - **Two system-call interfaces**
    - **Mach system calls (known as traps)**
    - **BSD system calls** (which provide POSIX functionality).





rich set of libraries that includes not only **the standard C library** but also libraries that provide **networking, security, and programming language support** (to name just a few).

message passing and **remote procedure calls** (RPCs)

for development of **device drivers**

dynamically loadable modules kernel extensions(kexts)

Figure 2.17 The structure of Darwin

2.8 Operating System Structure

2.8.5 Hybrid Systems

- 2.8.5.1 Apple's macOS and iOS
  - Much of the **functionality** provided by Mach is available through **kernel abstractions**, which include **tasks** (a Mach process), **threads**, **memory** objects, and **ports** (used for IPC).
    - (Mach에서 제공하는 기능들은 커널 추상화를 통해 제공됨)
  - To address **overhead of message passing** between different services, **Darwin combines** Mach, BSD, the I/O kit, and any kernel extensions **into a single address space**.
    - (서비스 간 message passing의 overhead를 처리하기 위해 단일 주소 공간으로 통합)
    - Thus, Mach is **not a pure microkernel** in the sense that various subsystems run in user space.
    - **Message passing** within Mach still does **occur**, but **no copying** is necessary, as the services have access to the same address space.
      - (message passing 은 일어나지만 message copy는 하지 않음)

2.8 Operating System Structure

2.8.5 Hybrid Systems

- 2.8.5.2 Android
  - was designed by the Open Handset Alliance (led primarily by Google)
  - was developed for **Android smartphones** and **tablet computers**.
  - runs on a **variety of mobile platforms** and is **open-sourced**, partly explaining its **rapid rise in popularity**.
    - **iOS** is designed to run on **Apple mobile devices** and is **close-sourced**,
  - Figure 2.18
    - It is a **layered stack** of software that provides a **rich set of frameworks** supporting **graphics, audio, and hardware features**.

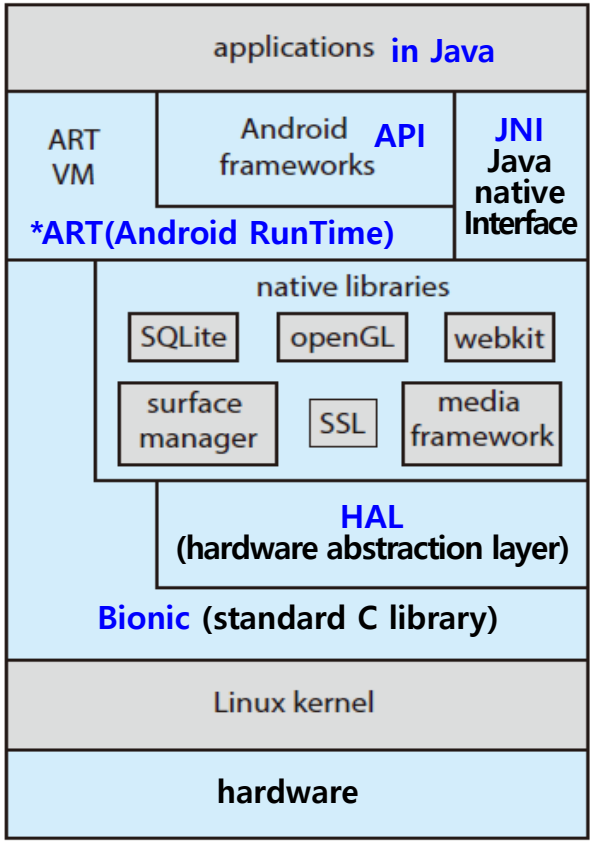
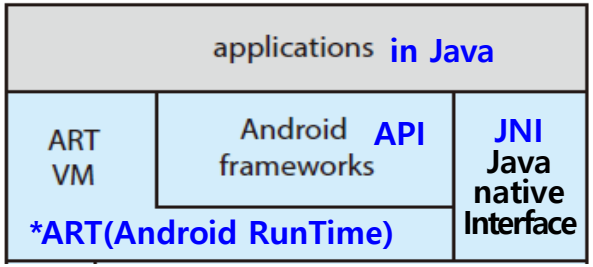


Figure 2.18
Architecture of Google's Android

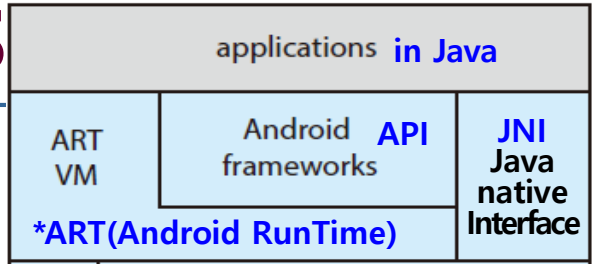
2.8 Operating System Structure

2.8.5 Hybrid Systems

- 2.8.5.2 Android
  - Software designers for Android devices develop applications in the **Java language**, but they do **not** generally **use** the **standard Java API**.
    - (개발자들은 Java 언어로 개발, 표준 Java API 대신 별도의 Android API 사용)
  - Google has designed a **separate Android API** for Java development.
    - **Java applications are compiled** into a form that can execute on the **ART**.
      - (자바 응용은 ART에서 실행되는 형식으로 컴파일됨)
    - **ART(Android RunTime)** : **a virtual machine** designed for Android and **optimized** for **mobile devices** with limited memory and CPU processing capabilities.
      - (제한된 메모리 CPU 처리 능력을 갖는 모바일 장치에 최적화된 가상 머신)
    - Java programs are **first compiled to a Java bytecode .class file** and then **translated into an executable .dex file**.
      - (Java 프로그램 → bytecode로 컴파일(.class) → 실행파일(.dex)로 변환 → ART에서 실행)

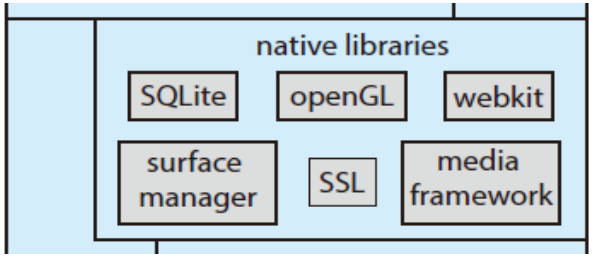


2.8 Operating System S



2.8.5 Hybrid Systems

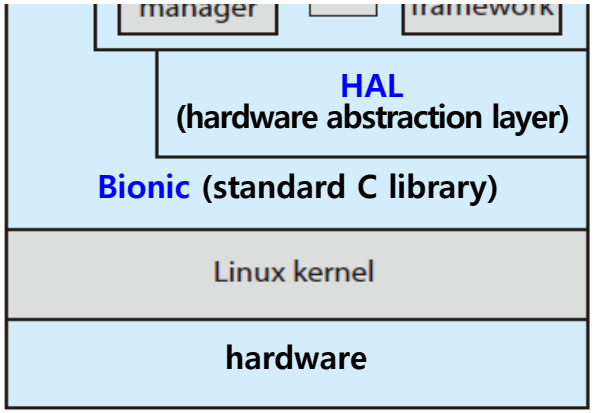
- 2.8.5.2 Android
  - Whereas many **Java virtual machines** perform **just-in-time (JIT) compilation** to improve application efficiency, **ART performs ahead-of-time (AOT) compilation**.
    - AOT compilation allows **more efficient** application execution as well as **reduced power consumption**, features that are crucial **for mobile systems**.
  - Android developers can also write **Java programs** that use the **Java native Interface(JNI)**
    - which allows **developers** to bypass the virtual machine and instead **write Java programs** that can **access specific hardware features**.
      - (가상 머신을 우회하여 특정 하드웨어 기능에 접근)
    - Programs written using JNI are generally **not portable** from one hardware device to another.
  - **Set of native libraries** available for Android applications
    - **web browsers** (**webkit**)
    - **database** support (**SQLite**)
    - **network** support such as secure sockets (**SSLs**).



2.8 Operating System Structure

2.8.5 Hybrid Systems

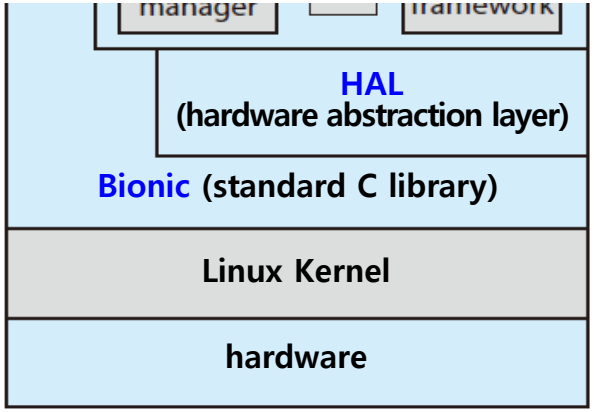
- 2.8.5.2 Android
  - Because Android can run on an almost **unlimited number of hardware devices**, Google has chosen to **abstract the physical hardware** through the **hardware abstraction layer(HAL)**.
    - (수많은 하드웨어 장치에서 실행되므로 물리적 하드웨어를 추상화함)
    - camera, GPS chip, and other sensors
    - HAL provides **applications** with a **consistent view** independent of specific hardware.
      - (특정 하드웨어와 상관없는 일관성 있는 view를 제공함)
    - This feature allows **developers** to write programs that are **portable across** different hardware platforms.
      - (다른 하드웨어 플랫폼에 이식 가능한 프로그램 작성이 가능해짐)



2.8 Operating System Structure

2.8.5 Hybrid Systems

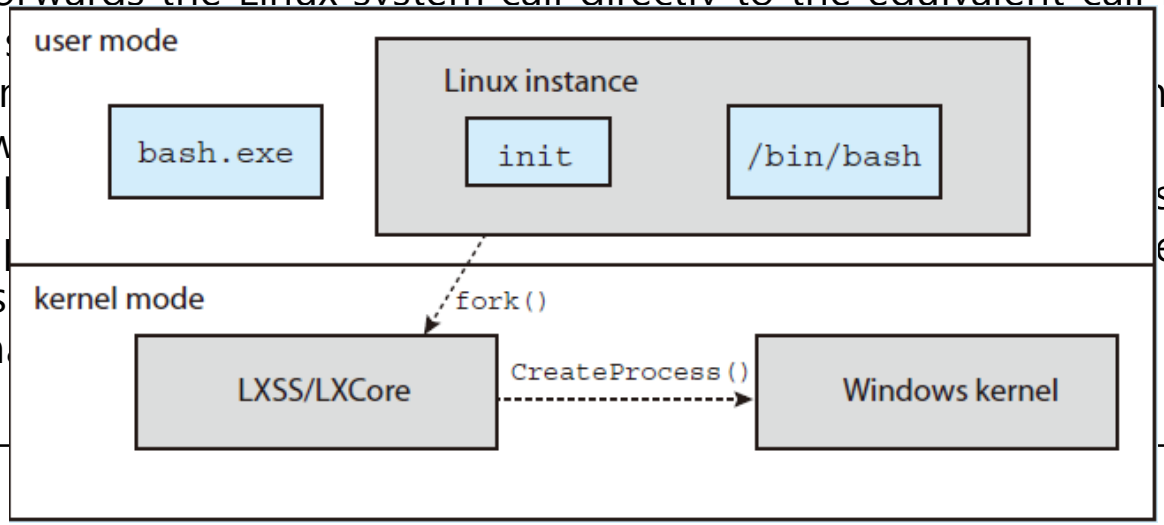
- 2.8.5.2 Android
  - Google instead developed the **Bionic standard C library** for Android.
    - (Linux의 GNU C 라이브러리(glic) 대신 Bionic 표준 C 라이브러리 개발)
    - Not only does Bionic have a **smaller memory footprint** than glibc, but it also has been designed for the **slower CPUs** that characterize **mobile devices**.
  - At the **bottom** of Android's software stack is the **Linux kernel**.
    - Google has **modified the Linux kernel** used in Android in a variety of areas **to support** the special needs of **mobile systems**, such as **power management**.
    - It has also made changes in **memory management and allocation** and has added a **new form of IPC** known as Binder (Section 3.8.2.1).



WINDOWS SUBSYSTEM FOR LINUX

Windows uses a hybrid architecture that provides subsystems to emulate different OS environments. These user-mode subsystems communicate with theWindows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux (WSL), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application bash.exe, which presents the userwith a bash shell running Linux. Internally, the WSL creates a Linux instance consisting of the init process, which in turn creates the bash shell running the native Linux application /bin/bash. Each of these processes runs in a Windows Pico process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS servicemust provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some



similar but not identical. Wher
will invoke the similar Window
The Linux fork() provides an ill
similar to fork() but does not
in WSL, the LXSS service does
CreateProcess() to do the rem
behavior of WSL.