



고려대학교
KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 2 Operating-System Structures

Heonchang Yu

Distributed and Cloud Computing Lab.

Contents

- Operating System Services
- User and Operating System Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating-System Specific
- Operating-System Design and Implementation
- Operating-System Structure
- Building and Booting an Operating System
- Operating-System Debugging

Objectives

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Apply tools for monitoring operating system performance.
- Design and implement kernel modules for interacting with a Linux kernel.

Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
 - ✓ User interface - Almost all operating systems have a user interface (UI)
 - ❖ graphical user interface (GUI) : a window system
 - ❖ Touch-screen interface : enabling users to slide their fingers across the screen or press buttons on the screen
 - ❖ command-line interface (CLI) : uses text commands
 - ✓ Program execution - The system must be able to load a program into memory, to run that program, and to end its execution, either normally or abnormally (indicating error)
 - ✓ I/O operations - A running program may require I/O, which may involve a file or an I/O device.
 - ✓ File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search for a given file, list file information, and include permissions management.

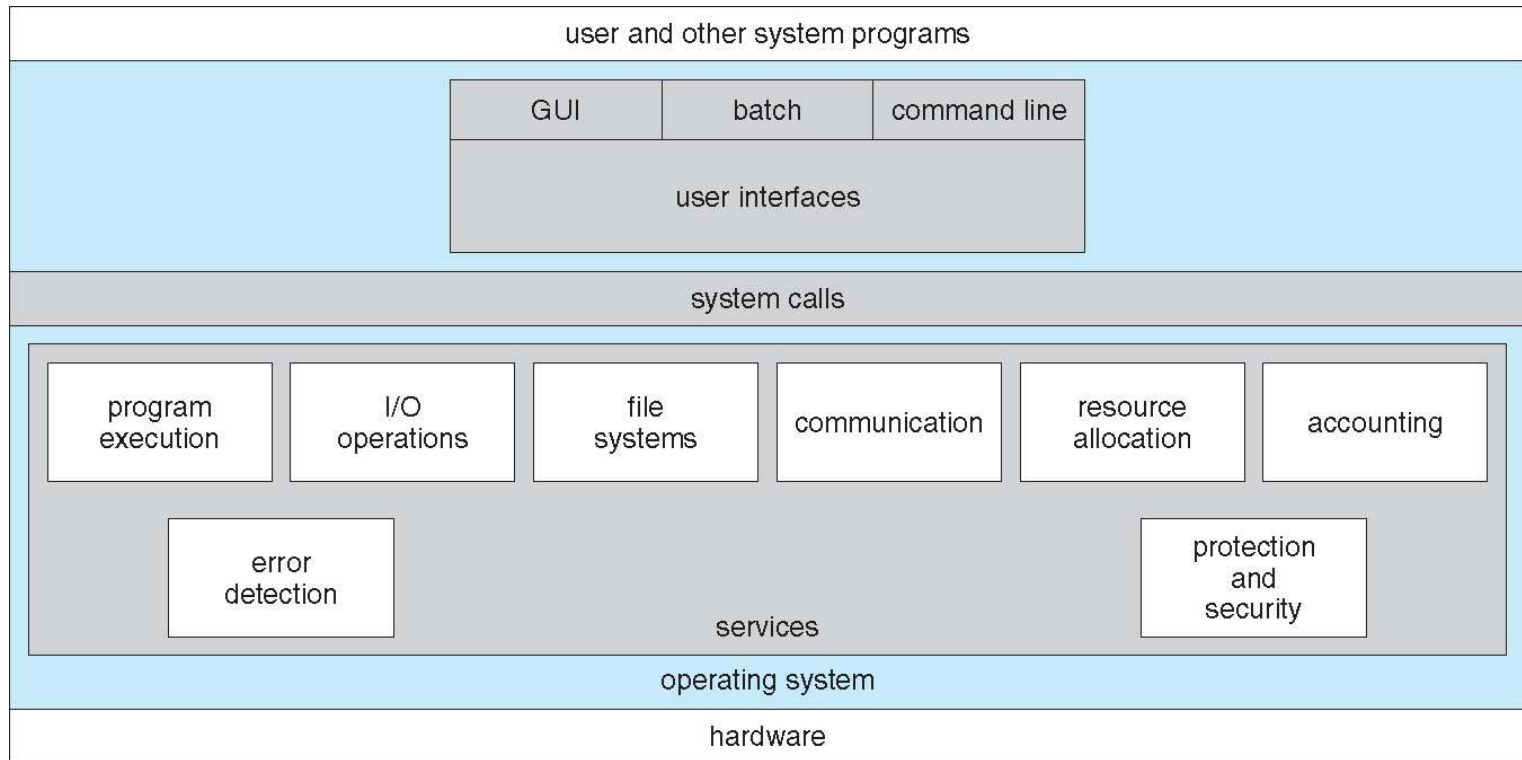
Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont):
 - ✓ Communications – occur between processes that are executing on the same computer or on different computer systems
 - ❖ Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
 - ❖ Error detection – OS needs to be detecting and correcting errors constantly.
 - ❖ May occur in the CPU and memory hardware, in I/O devices, and in the user program
 - ❖ For each type of error, OS should take the appropriate action to ensure correct and consistent computing

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
 - ✓ **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - ❖ Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have much more general request and release code.
 - ✓ **Logging** - To keep track of which programs use how much and what kinds of computer resources
 - ✓ **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information. Concurrent processes should not interfere with each other.
 - ❖ **Protection** involves ensuring that all access to system resources is controlled
 - ❖ **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
 - ❖ If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

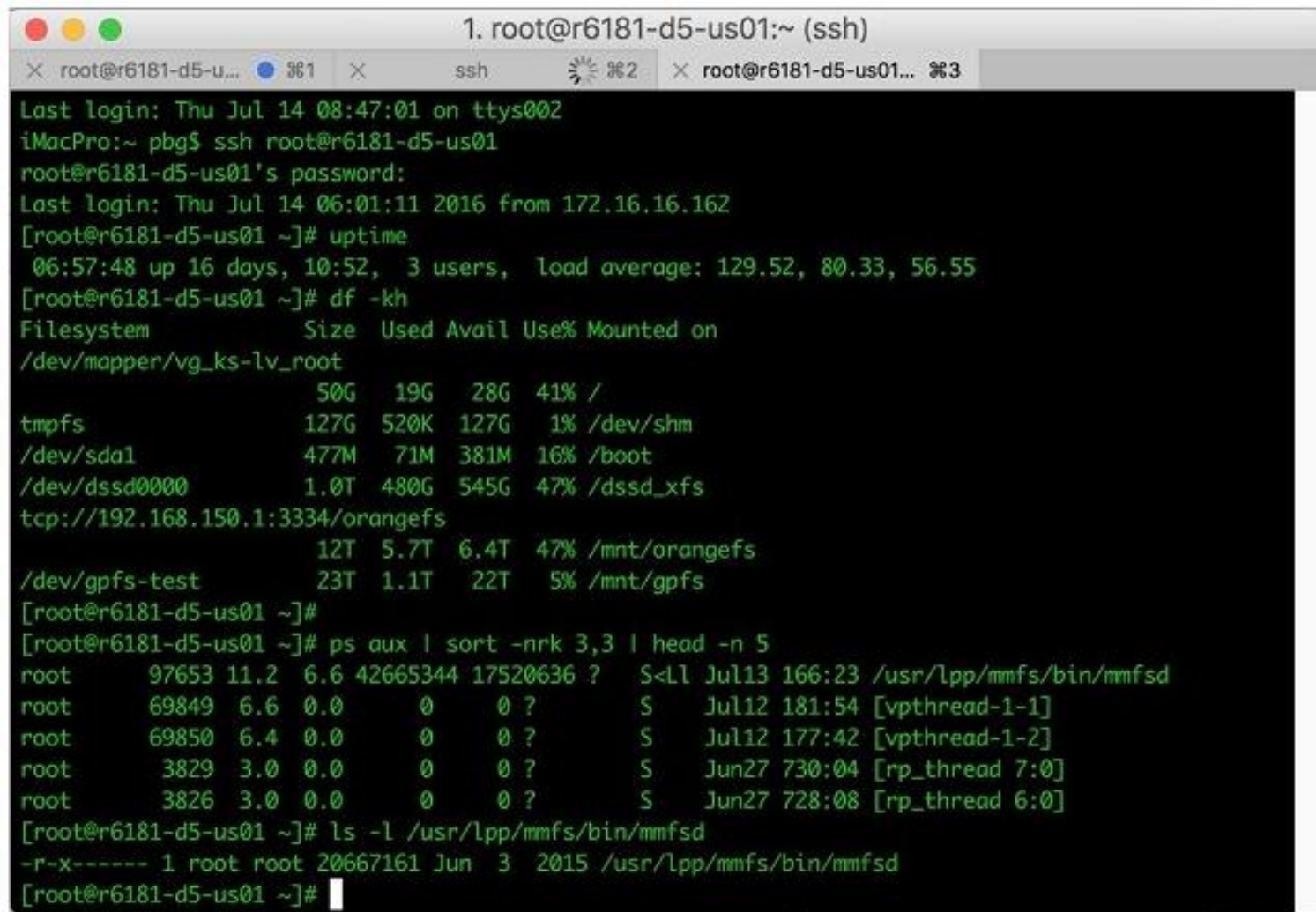
A View of Operating System Services



User and Operating System Interface - CLI

- CLI(command-line interface) allows direct command entry
 - ✓ Sometimes included in kernel, sometimes treated as a special program
 - ✓ On systems with multiple command interpreters – **shells**
 - ❖ Bourne-Again shell, C shell, Korn shell etc.
 - ✓ The main function is to get and execute the next user-specified command.
 - ❖ create, delete, list, print, copy, execute, and so on.
 - ✓ These commands can be implemented in two general ways.
 - ❖ The CI itself contains the code to execute the command.
 - ❖ An alternative approach implements most commands through system programs.
 - » Programmers can add new commands to the system easily by creating new files with the proper names.

Bourne-Again Shell Command Interpreter



```
1. root@r6181-d5-us01:~ (ssh)
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem                Size      Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root   50G       19G   28G   41% /
tmpfs                     127G      520K   127G    1% /dev/shm
/dev/sda1                  477M       71M   381M   16% /boot
/dev/dssd0000              1.0T     480G   545G   47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                        12T     5.7T   6.4T   47% /mnt/orangefs
/dev/gpfs-test             23T     1.1T   22T    5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?    S<Ll  Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0      0      0 ?        S      Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0      0      0 ?        S      Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0      0      0 ?        S      Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0      0      0 ?        S      Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

Figure 2.2 The bash shell command interpreter in macOS.

User and Operating System Interface - GUI

- User-friendly graphical user interface
 - ✓ A mouse-based window-and-menu system
 - ✓ **Icons** represent programs, files, directories, and system functions.
 - ✓ Clicking a button on the mouse can invoke a program, select a file or directory (known as a **folder**) or pull down a menu that contains commands.
 - ✓ Invented in the early 1970s at Xerox PARC
- Graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s.
 - ✓ The adoption of the "Aqua" interface that appeared with Mac OS X.
 - ✓ Microsoft's first version of Windows was based on the addition of a GUI interface to the MS-DOS operating system.

참고로만 보면 된다

Touch-Screen Interfaces

참고로만 보면 된다

- A mouse-and-keyboard system is impractical for most mobile systems.
 - ✓ Smartphones and handheld tablet computers use a touch-screen interface.
- Users interact by making gestures on the touch screen – for example, pressing and swiping fingers across the screen.
- Most smartphones simulate a keyboard on the touch screen.



Figure 2.3 The iPhone touch screen.

The Mac OS GUI

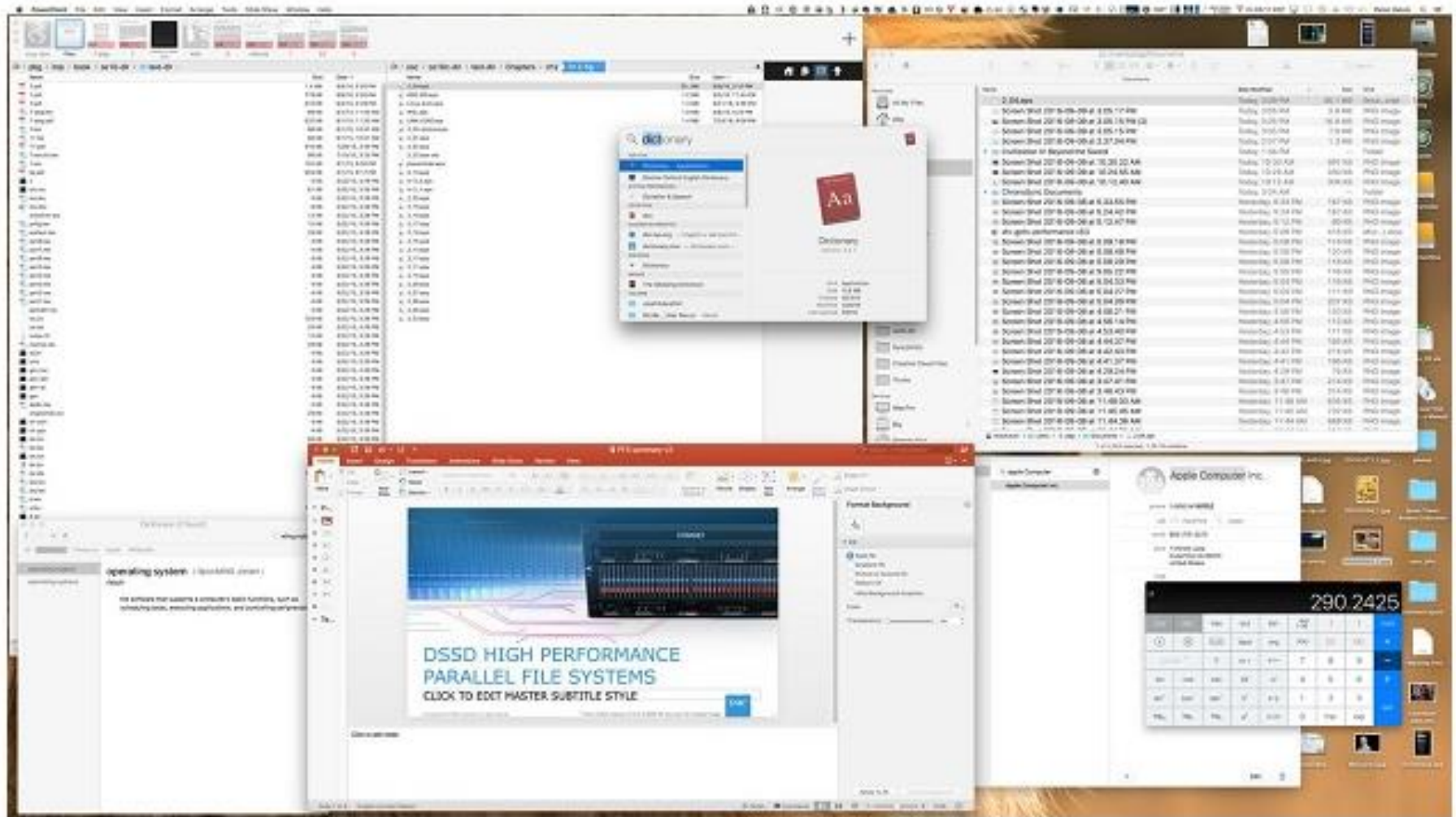


Figure 2.4 The macOS GUI.

System Calls

API는 유저 친화적인 기능이고,
시스템 콜은 커널 친화적 기능이다

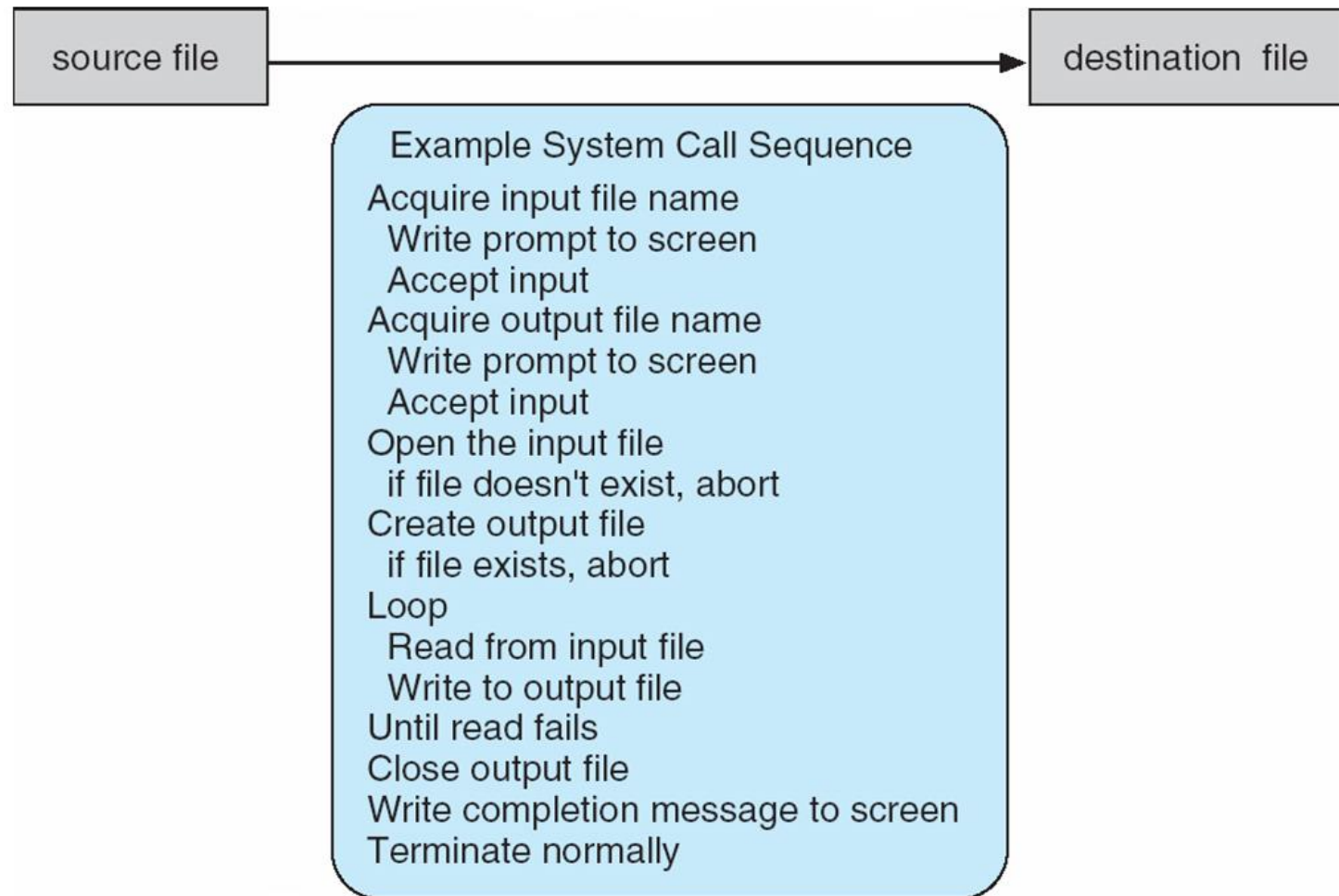
- Provide an interface to the services made available by the OS
- Are available as functions written in C and C++
- Application developers design programs according to an **Application Programming Interface (API)** rather than direct system calls.
- The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and macOS), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?
 - ✓ Program portability: An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API.
 - ✓ Actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.

System Calls

- We discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used:
 - ✓ Writing a simple program to read data from one file and copy them to another file.
 - ✓ The first input that the program will need is **the names of the two files**: the **input file** and the **output file**. These names can be specified in many ways, depending on the operating-system design. **One approach is to pass the names of the two files as part of the command** – for example, the UNIX `cp` command: `cp in.txt out.txt`
 - ✓ **A second approach is for the program to ask the user for the names.** In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. **On mouse-based and icon-based systems**, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. **This sequence requires many I/O system calls.**

Example of System Calls

- System call sequence to copy the contents of one file to another file



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:

<pre>#include <unistd.h></pre>		
<pre>ssize_t</pre>	<pre>read(int fd, void *buf, size_t count)</pre>	
return	function	parameters
value	name	

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`.

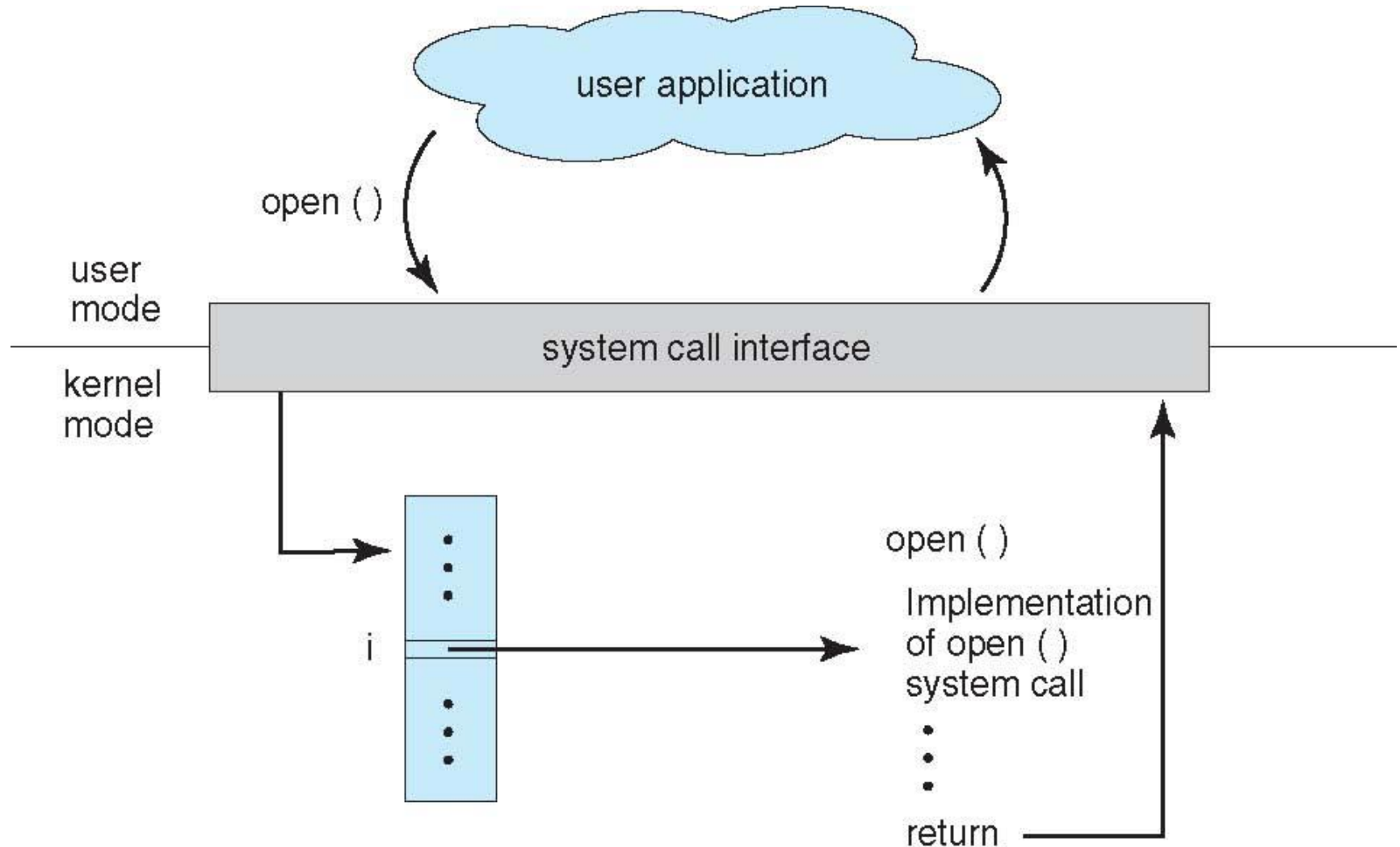
System Call Implementation

- Run-Time Environment (RTE): the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders
- RTE provides a **system-call interface** that serves as the link to system calls made available by the operating system.
 - ✓ The system call interface intercepts function calls in the API and invokes the necessary system calls within OS.
 - ❖ Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
 - ✓ The system call interface invokes the intended system call in OS kernel and returns the status of the system call.

System Call Implementation

- The caller need know nothing about how the system call is implemented or what does during execution.
 - ✓ The caller need only obey the API and understand what OS will do as a result of the execution of that system call.
 - ✓ Most of the details of the OS interface are hidden from the programmer by API and are managed by the RTE.

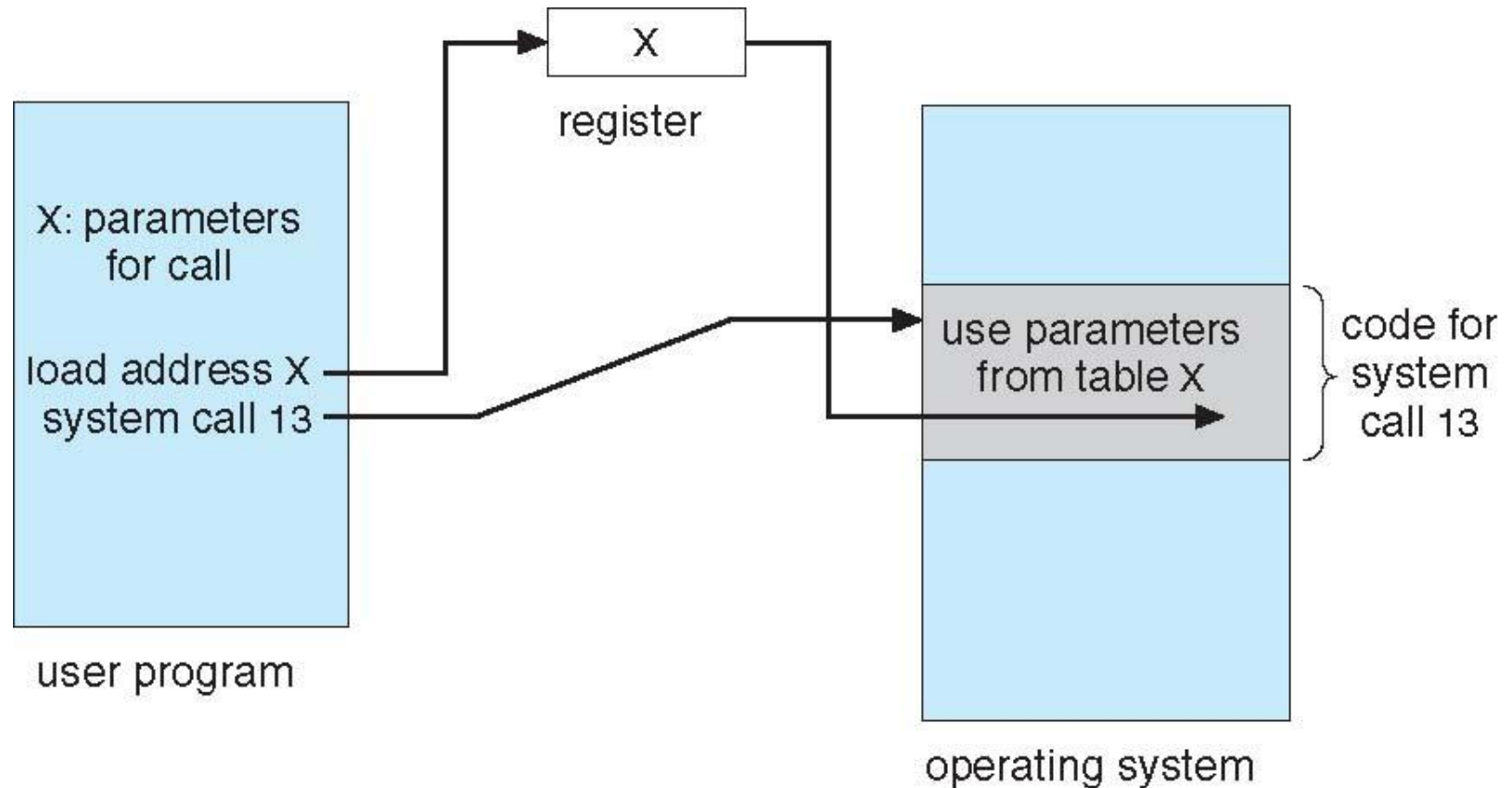
API – open () System Call



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
 - ✓ For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer.
- Three general methods used to pass parameters to the OS
 - Simplest: **pass the parameters in registers**
 - ✓ In some cases, there may be more parameters than registers.
 - Parameters are stored in a **block**, or **table**, in memory, and address of the block is passed as a parameter in a register.
 - ✓ This approach taken by Linux and Solaris
 - Parameters can be placed, or pushed, onto a **stack** by the program and popped off the stack by the operating system
 - **Block** and **stack methods** do not limit the number or length of parameters being passed

Passing of Parameters as a Table



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 attributes
 - set process, file, or device attributes

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

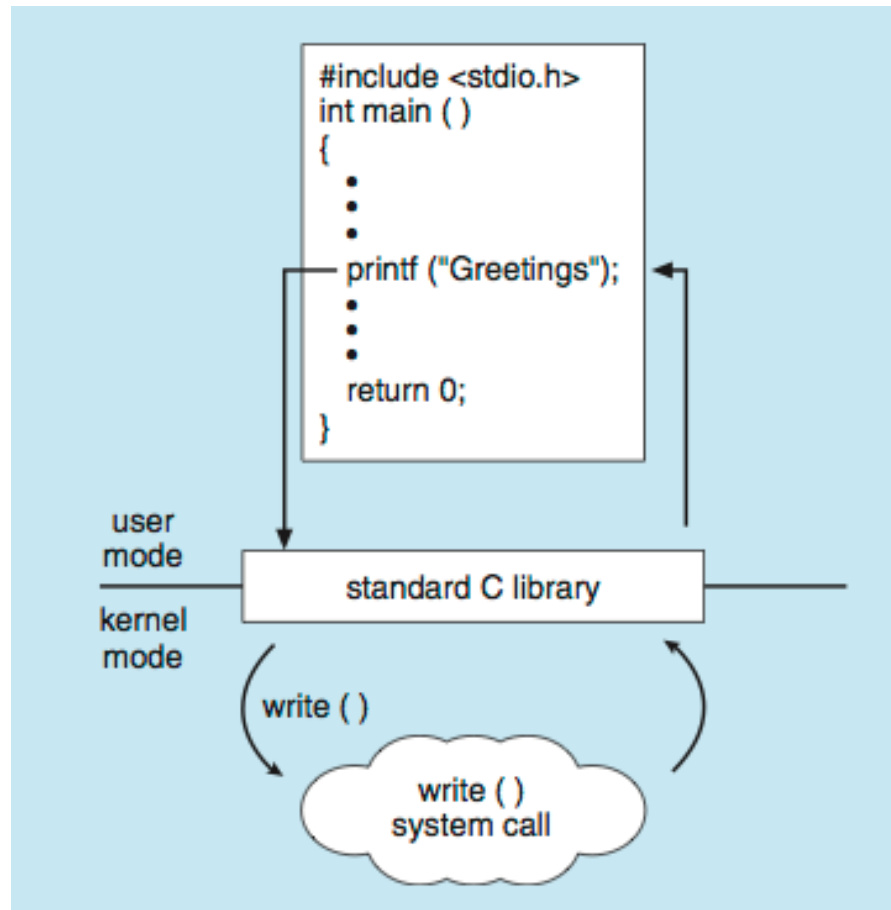
Types of System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

Example of Standard C Library

- C program invokes the `printf()`. The C library intercepts this call and invokes the necessary system call (`write()`) in OS. C library takes the value returned by `write()` and passes it back to the user program:



Example: Single-tasking system - Arduino

- ✓ Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events.
- ✓ To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a [sketch](#)) from the PC to the Arduino's flash memory via a USB connection.
- ✓ The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a **boot loader** loads the sketch into a specific region in the Arduino's memory.

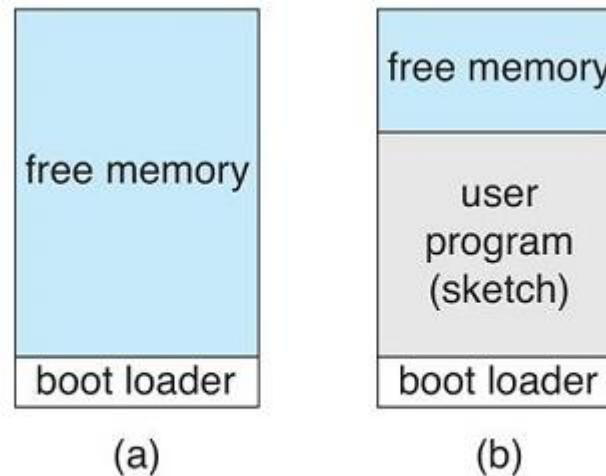


Figure 2.9 Arduino execution. (a) At system startup. (b) Running a sketch.

Example: Multitasking system - FreeBSD

- Derived from Berkeley Unix
- When a user logs on to the system, the shell of the user's choice is run.
- Since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed. (Figure 2.10)
- To start a new process, the shell executes a `fork()` system call.
 - ✓ The selected program is loaded into memory via an `exec()` system call, and the program is executed.
 - ✓ The shell either waits for the process to finish or runs the process "in the background".

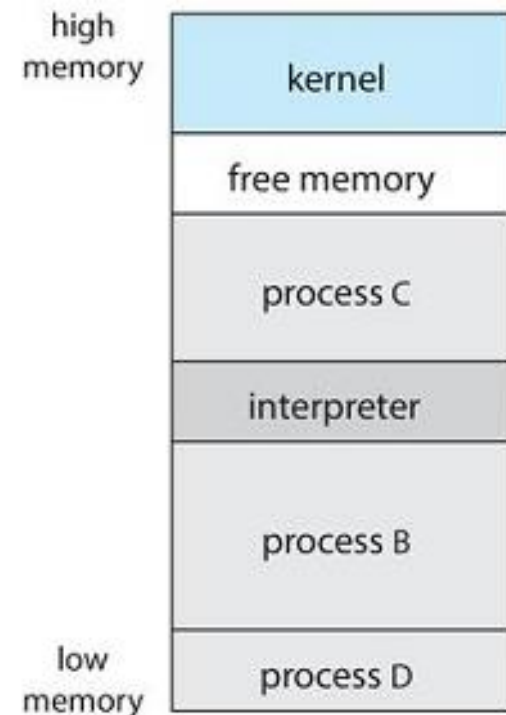


Figure 2.10 FreeBSD running multiple programs.

System Services

- System services provide a convenient environment for program development and execution. They can be divided into these categories:
 - File manipulation
 - Status information
 - File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Background services
- Most users' view of the operating system is defined by the application and system programs, rather than by the actual system calls.

System Services

- Also known as **system utilities**
- Some of them are simply user interfaces to system calls; others are considerably more complex.
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
 - ✓ Some programs ask the system for the date, time, amount of available memory, disk space, number of users, or similar status information.
 - ✓ Others are more complex, providing detailed performance, logging, and debugging information.
 - ✓ Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI.
 - ✓ Some systems support a registry - used to store and retrieve configuration information

System Services

- File modification
 - ✓ Text editors to create and modify the contents of files stored on disk
 - ✓ Special commands to search contents of files or perform transformations of the text
- Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
- Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay loaders, debugging systems for higher-level or machine language
- Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems
 - ✓ Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another
- Background services
 - ✓ Some of these processes terminate after completing their tasks, while others continue to run until the system is halted.
 - ✓ Known as **services**, **subsystems**, **daemons**

Linkers and Loaders

- Procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core
 - ✓ **Compiled** into object files that are designed to be loaded into any physical memory location, a format known as a **relocatable object file**.
 - ✓ **Linker** combines these relocatable object files into a single binary **executable file**.
 - ✓ **Loader** is used to load the binary executable file into memory. Activity associated with linking and loading is **relocation**.

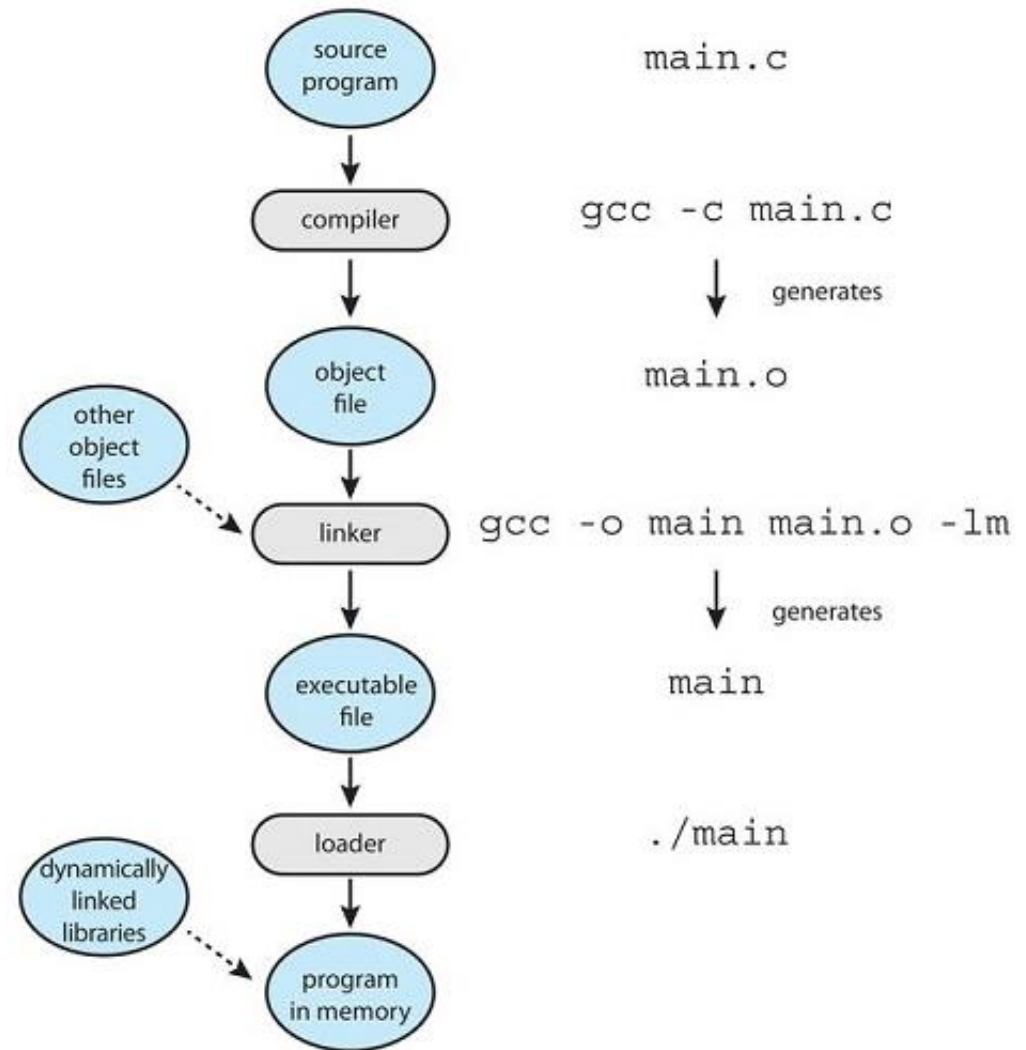


Figure 2.11 The role of the linker and loader.

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Design goals
 - Start by defining goals and specifications
 - Affected by choice of hardware, type of system
- User goals and System goals
 - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation

- Important principle to separate
 - Policy:** What will be done
 - Mechanism:** How to do something
- The separation of policy and mechanism is important for flexibility.
 - Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism.
 - The timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

Operating System Design and Implementation

- Early operating systems were written in assembly language.
- Now, most are written in a higher-level languages, C or C++.
 - Lowest levels of kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages.
- Advantages of using a higher-level language
 - The code can be written faster, is more compact, and is easier to understand and debug.
 - An operating system is far easier to port – to move to some other hardware.
- Disadvantages of implementing an operating system in a higher-level language
 - reduced speed and increased storage requirements

OS Structure: Monolithic Structure

- The simplest structure for organizing an operating system is no structure at all.
- A common technique for designing operating systems
- The UNIX OS consists of two separable parts.
 - Systems programs
 - The kernel
 - ✓ Separated into a series of interfaces and device drivers
 - ✓ Everything below the system-call interface and above the physical hardware
 - ✓ Provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls

OS Structure: Monolithic Structure – UNIX

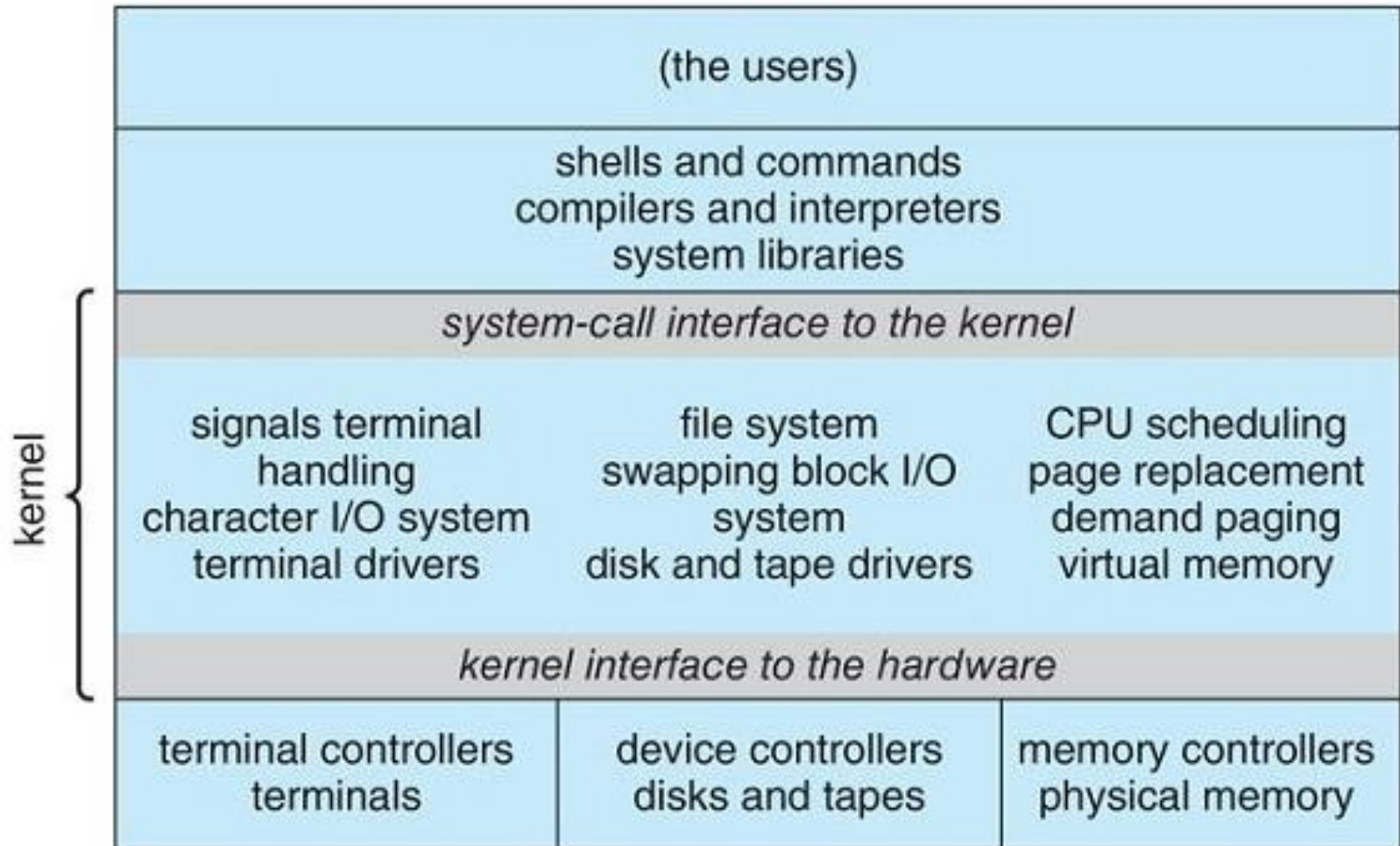


Figure 2.12 Traditional UNIX system structure.

OS Structure: Monolithic Structure – Linux

- Applications use `glibc` standard C library when communicating with system call interface to kernel.
- Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend.
- There is little overhead in system-call interface, and communication within kernel is fast.
- Despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows.

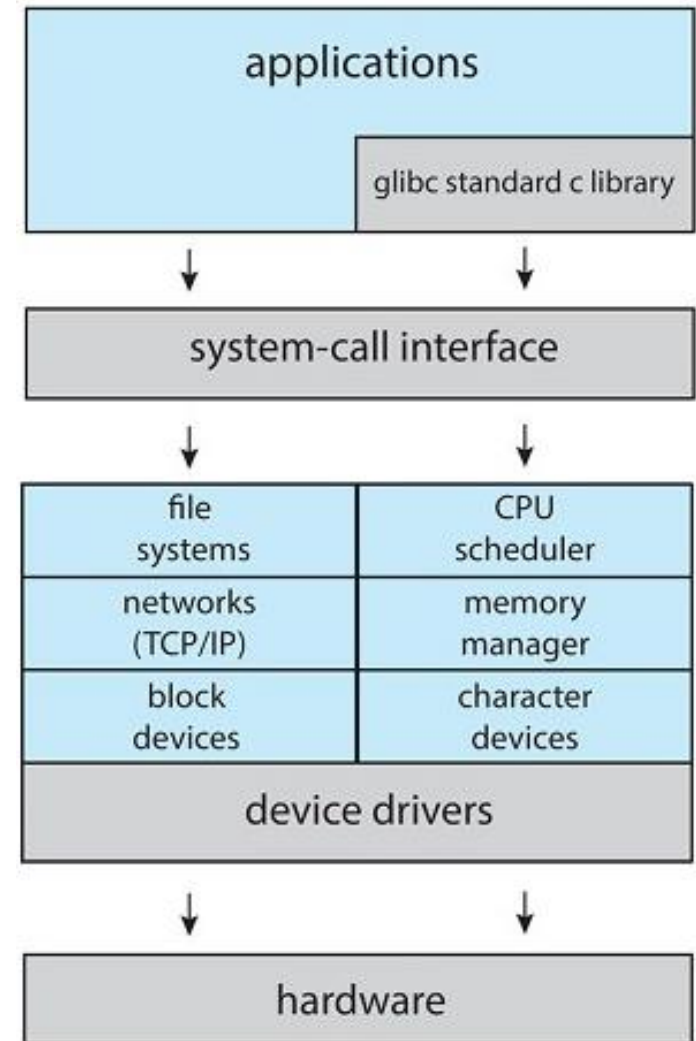
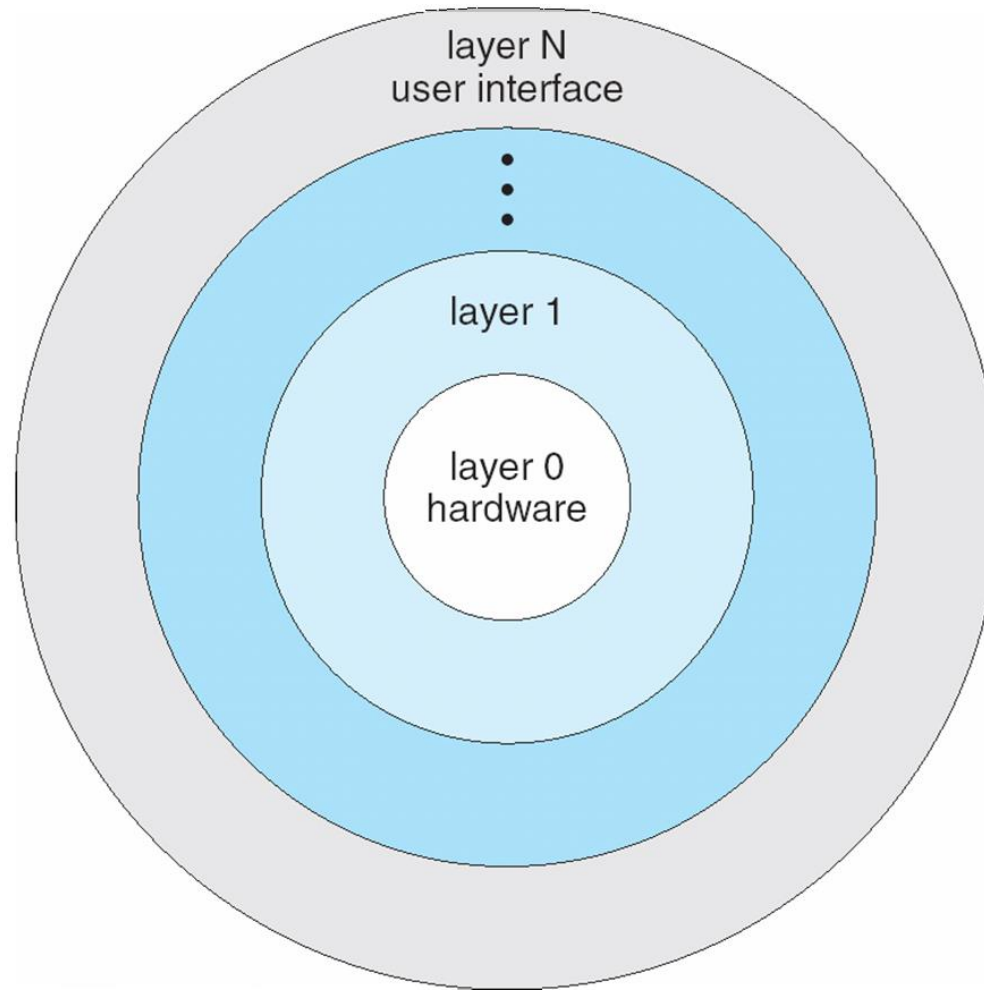


Figure 2.13 Linux system structure.

OS Structure: Layered Approach

- The monolithic approach is often known as a **tightly coupled system**. Alternatively, a **loosely coupled system** is divided into separate, smaller components that have specific and limited functionality.
- The operating system is divided into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected so that each uses functions (operations) and services of only lower-level layers
- The main advantage of the layered approach is simplicity of construction and debugging.
- Relatively few OSs use a pure layered approach. One reason involves the challenges of appropriately defining the functionality of each layer.

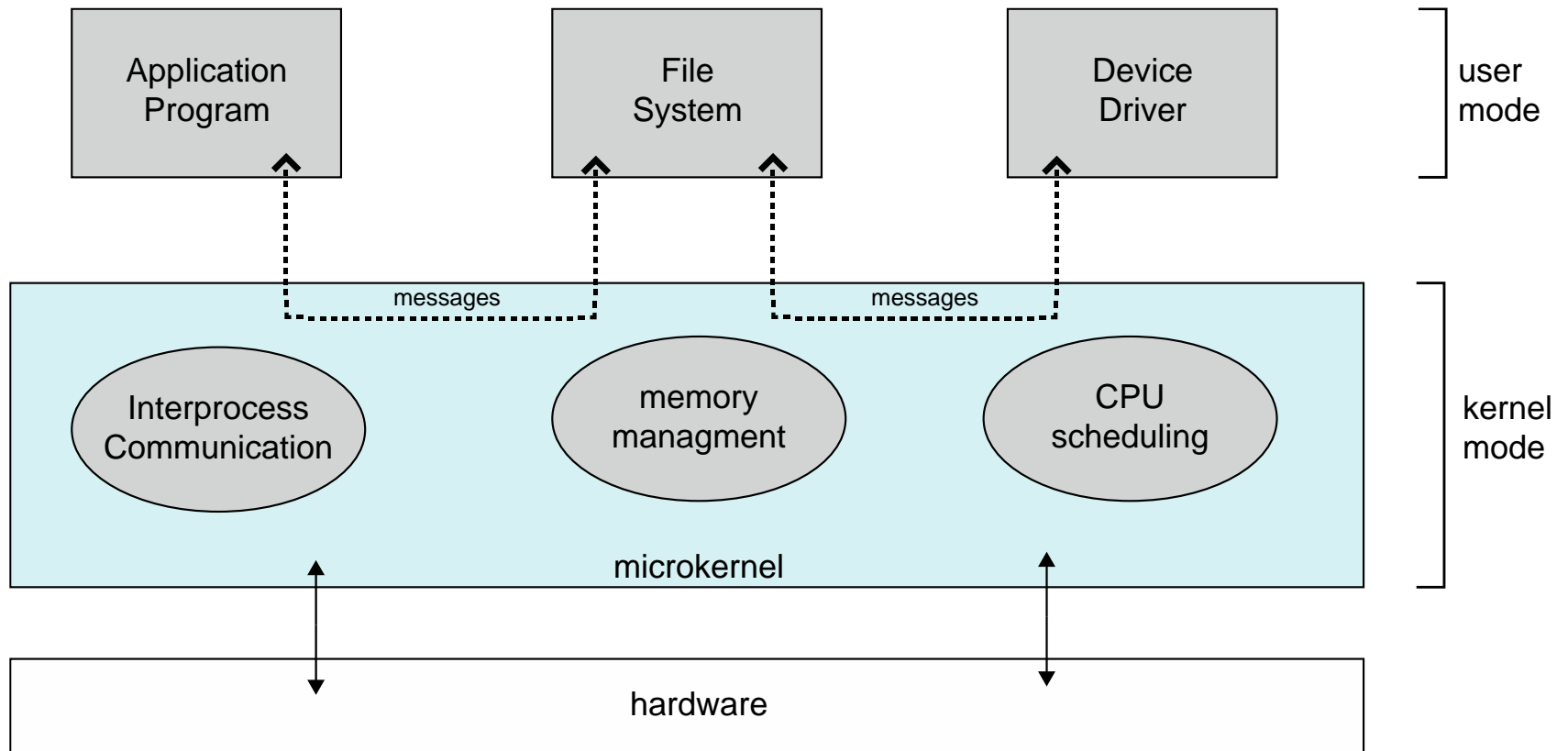
OS Structure: Layered Operating System



OS Structure: Microkernels

- Modularized the kernel using the **microkernel** approach
 - Mach: at Carnegie Mellon University, mid-1980s
- Removing all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces
- The main function of the microkernel is to provide communication between the client program and the various services that are running in user space. (The client program and service never interact directly)

Microkernel System Structure



OS Structure: Microkernels

– Benefits:

- Makes extending the operating system easier
- All new services are added to user space and consequently do not require modification of the kernel.
- Easier to port from one hardware design to another
- Provides more security and reliability

– Challenges:

- The performance of microkernels can suffer due to increased system-function overhead.
- ✓ Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely.

OS Structure: Hybrid Systems

- Very few operating systems adopt a single, strictly defined structure.
 - ✓ OS combines different structures, resulting in hybrid systems that address performance, security, and usability issues.
 - ✓ Linux is monolithic, because having the operating system in a single address space provides very efficient performance.
 - ✓ Windows is largely monolithic, but it retains some behavior typical of microkernel systems.

OS Structure: Hybrid Systems – macOS/iOS

- Apple's macOS is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile OS designed for the iPhone smartphone and iPad tablet computer. Architecturally, macOS and iOS have much in common.
 - ✓ **User experience layer** defines the software interface that allows users to interact with the computing devices. macOS uses the ***Aqua*** user interface, which is designed for a mouse or trackpad, whereas iOS uses the ***Springboard*** user interface, which is designed for touch devices.
 - ✓ **Application frameworks layer** includes the ***Cocoa*** (for developing **macOS** app) and ***Cocoa Touch*** (for **iOS**) frameworks, which provide an API for the Objective-C and Swift programming languages.
 - ✓ **Core frameworks** defines frameworks that support graphics and media.
 - ✓ **Kernel environments** (known as **Darwin**) includes the Mach microkernel and the BSD UNIX kernel.

OS Structure: Hybrid Systems – macOS/iOS

- 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 operating system is generally much more restricted to developers than macOS.

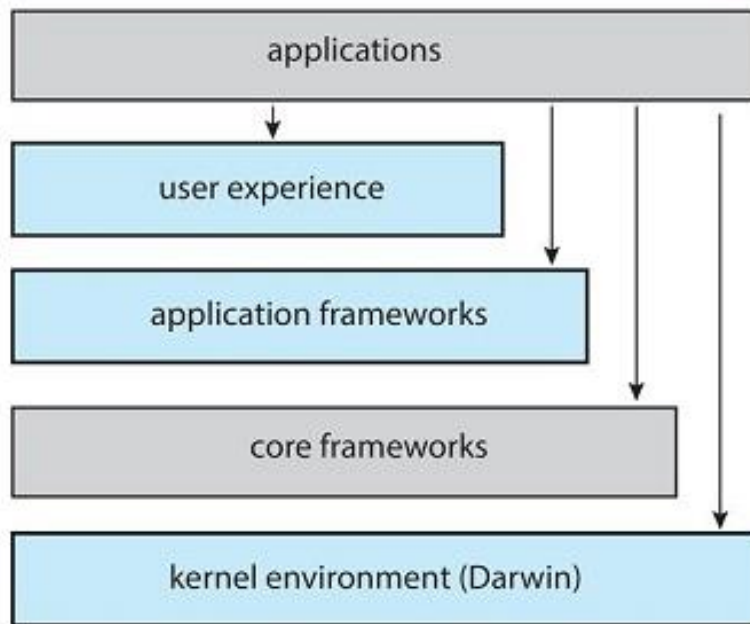


Figure 2.16 Architecture of Apple's macOS and iOS operating systems.

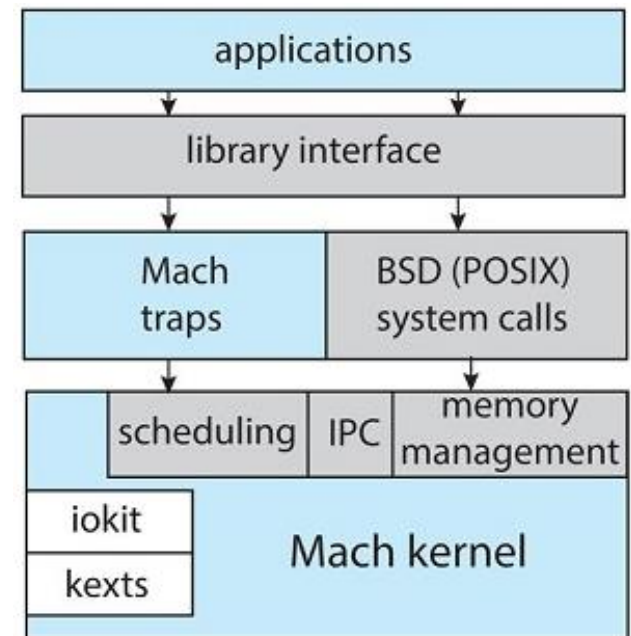


Figure 2.17 The structure of Darwin.

OS Structure: Hybrid Systems – Android

- Designed by Open Handset Alliance (led primarily by Google)
 - ✓ Android runs on a variety of mobile platforms and is open-sourced, but iOS is designed to run on Apple mobile devices and is close-sourced.
- Is similar to iOS in that it is a layered stack of software
- Software designers for Android devices develop applications in the Java language (But, do not use the standard Java API). Google has designed a separate Android API for Java development.
- Java applications are compiled into a form that can execute on the Android RunTime ART (a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities).
- The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).

OS Structure: Hybrid Systems – Android

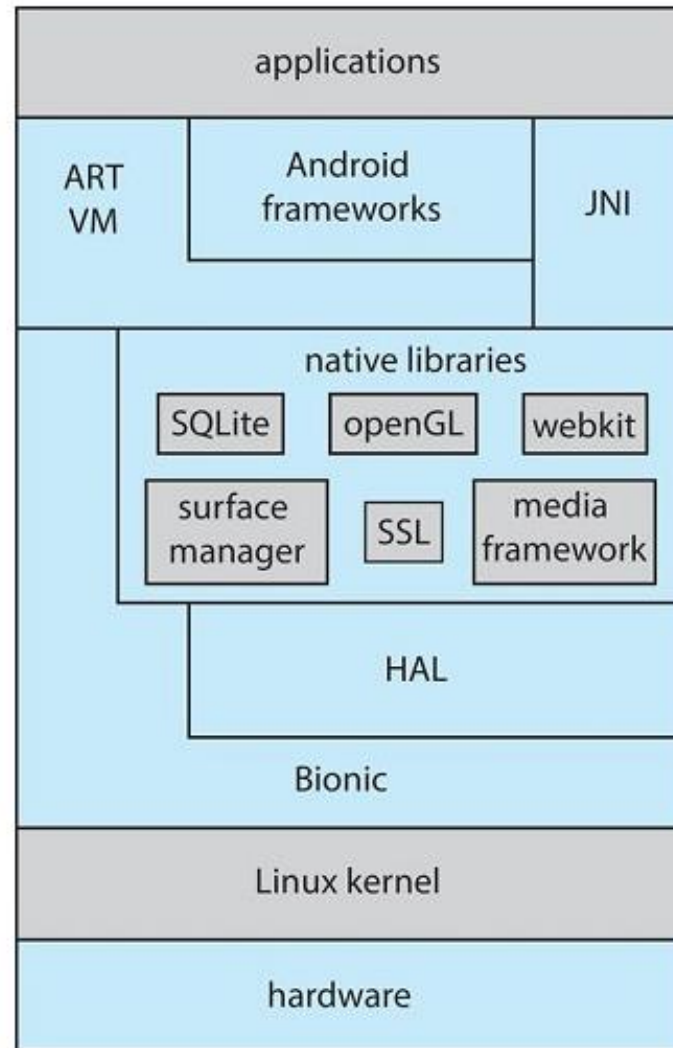


Figure 2.18 Architecture of Google's Android.