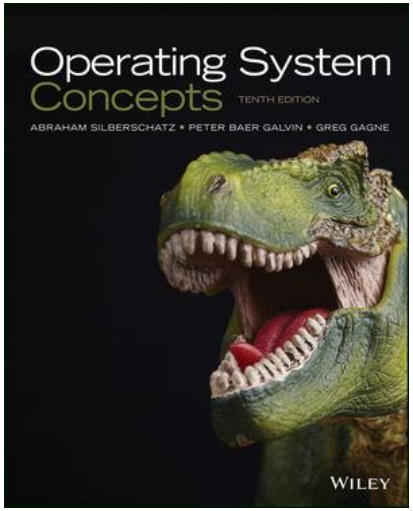
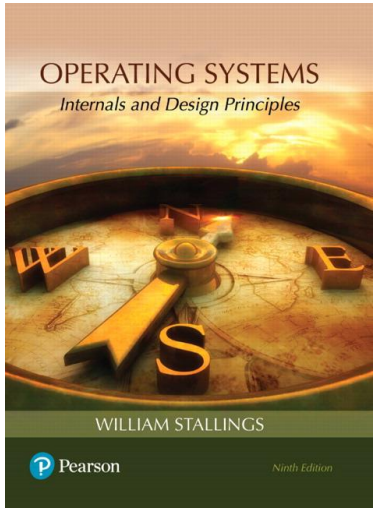
Sistemas Operativos

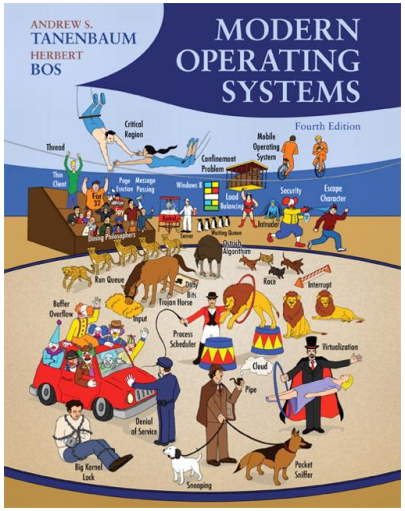
Operating Systems Structures

Disclaimer

- These slides and notes are based on the contents of the books:
  - Abraham Silberschatz, ” Operating System Concepts”, 10th Edition, Wiley, 2018;
  - William Stallings, “Operating Systems: Internals and Design Principles”, 9th Edition, Pearson, 2017;
  - Andrew S. Tanenbaum, “Modern Operating Systems”, 4th Edition, Pearson Education, 2014;
- The respective copyrights belong to their owners.







Outline

- Operating System Services
- System Calls
  - Definition
  - Classification
- Application Programming Interface (API)
- Operating Systems Implementation
- Operating Systems Structure
  - Simple Structure
  - Layered Approach
  - Microkernels
  - Modules
- Virtual Machines
- References

Operating-System Services

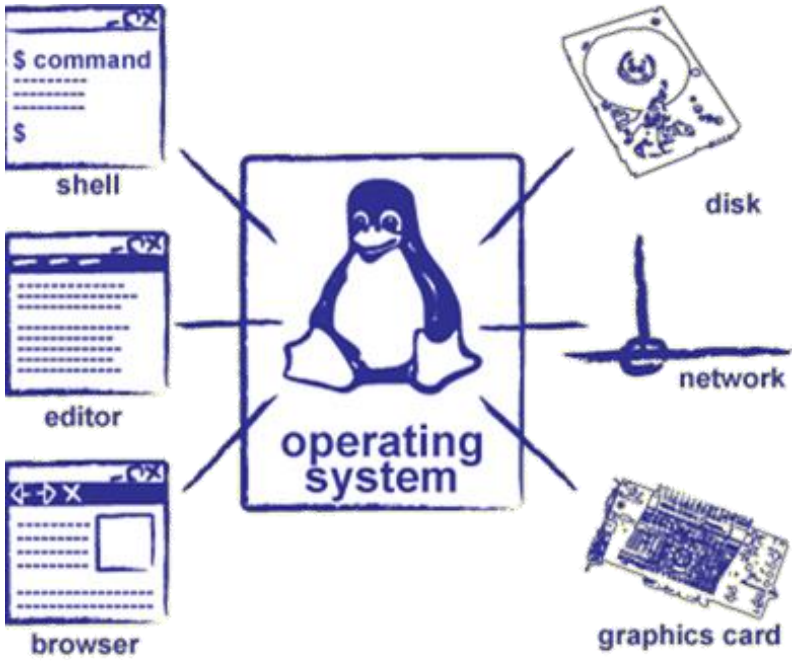
- An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.
- The specific **services** provided, of course, differ from one operating system to another, but we can identify common classes.
  - 1. User interface
  - 2. Program execution
  - 3. I/O operations
  - 4. File-system manipulation
  - 5. Communications
  - 6. Error detection
  - 7. Resource allocation
  - 8. Accounting
  - 9. Protection and security

Operating-System Services

- **1. User interface**
  - **Command-Line Interface (CLI)**
  - **Graphical User Interface (GUI)**
- **2. Program execution**
  - load a program into memory, run that program and end its execution, either normally or abnormally (indicating error).
- **3. I/O Operations**
  - users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **4. File-system manipulation**
  - read, write, search, list and delete files and directories based on permissions
- **5. Communications**
  - communication may occur between processes executing on the same computer or on different computer systems tied by a computer network.
- **6. Error detection**
  - for each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

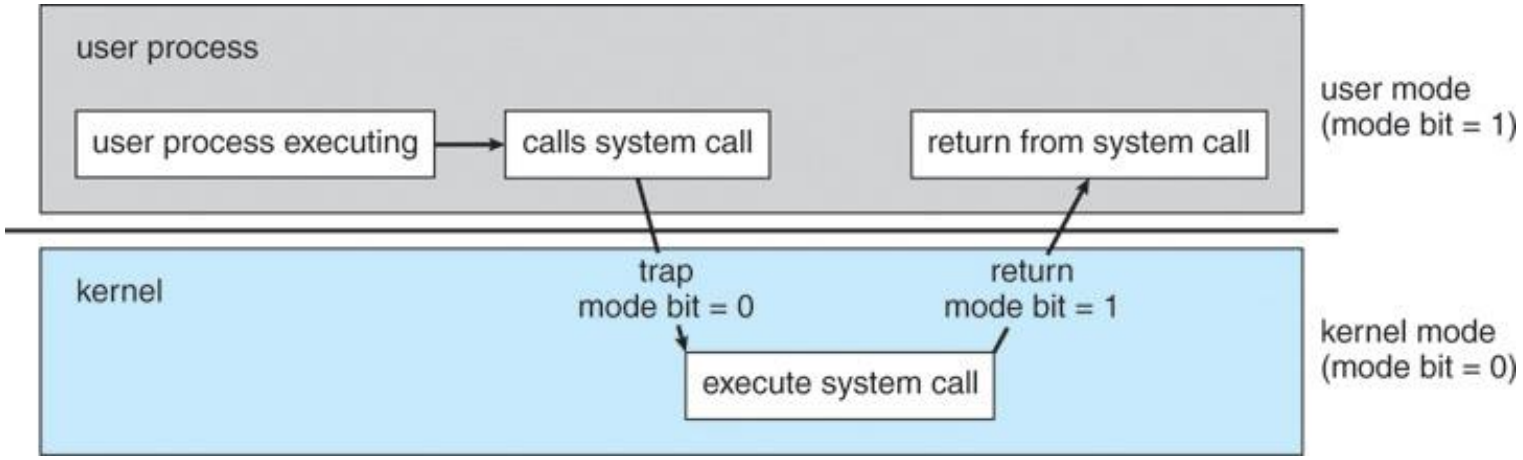
Operating-System Services

- **7. Resource allocation**
  - the following resources must be allocated between processes:
    - **CPU cycles;**
    - **Main Memory;**
    - **I/O devices.**
- **8. Accounting**
  - keep track of which users use, how much and what kinds of computer resources.
- **9. Protection and security.**
  - ensuring that all access to system resources is controlled. Security of the system from outsiders is also important.



System Calls

- **System calls**
  - **Definition**
  - the kernel makes its services available to the application programs through a large collection of entry points, known as **system calls:**
    - they consist in a request for the operating system to do something on behalf of the user's program

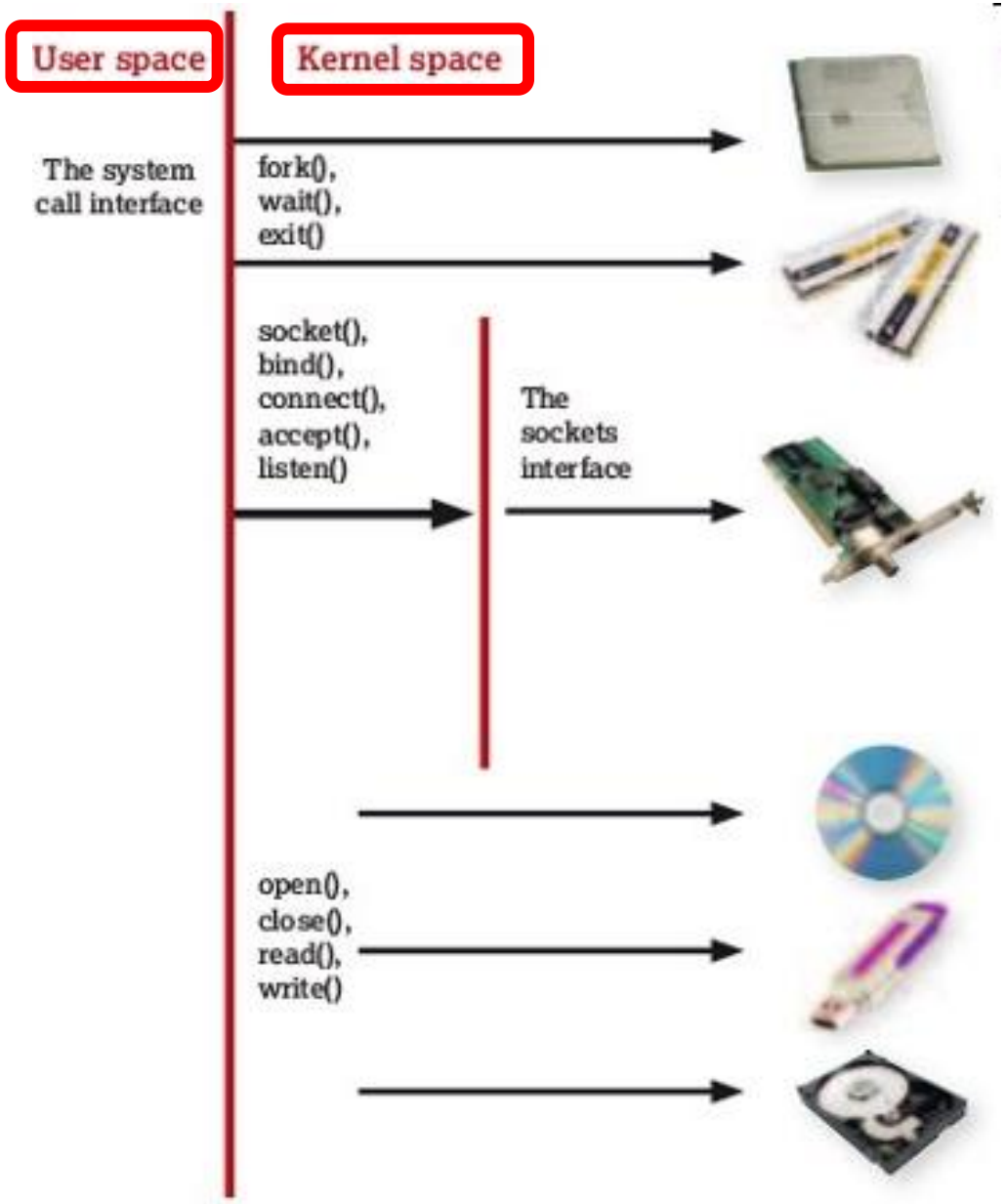


Abraham Silberschatz, " Operating System Concepts"

  - from a programmer's viewpoint, they look just like ordinary library of functions, although in reality a system call executes code in the kernel
  - the library functions invoke an OS **trap**, which is a software generated interrupt, to change the process execution mode to kernel

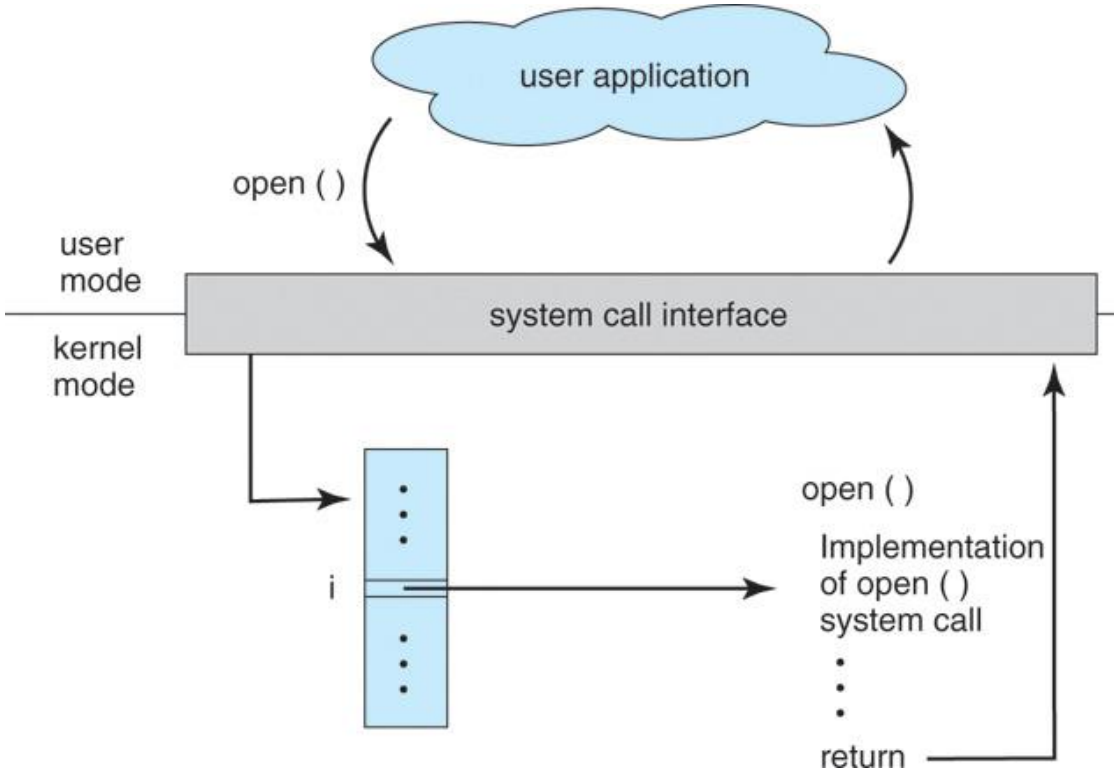
System Calls

- **System calls**
  - **Definition**
  - The library routines execute in user mode, but the system call interface is a special case of an interrupt handler.
  - These calls are generally available as routines written in C and C++, although certain low-level tasks, may need to be written using assembly-language instructions.



System Calls

- **System calls**

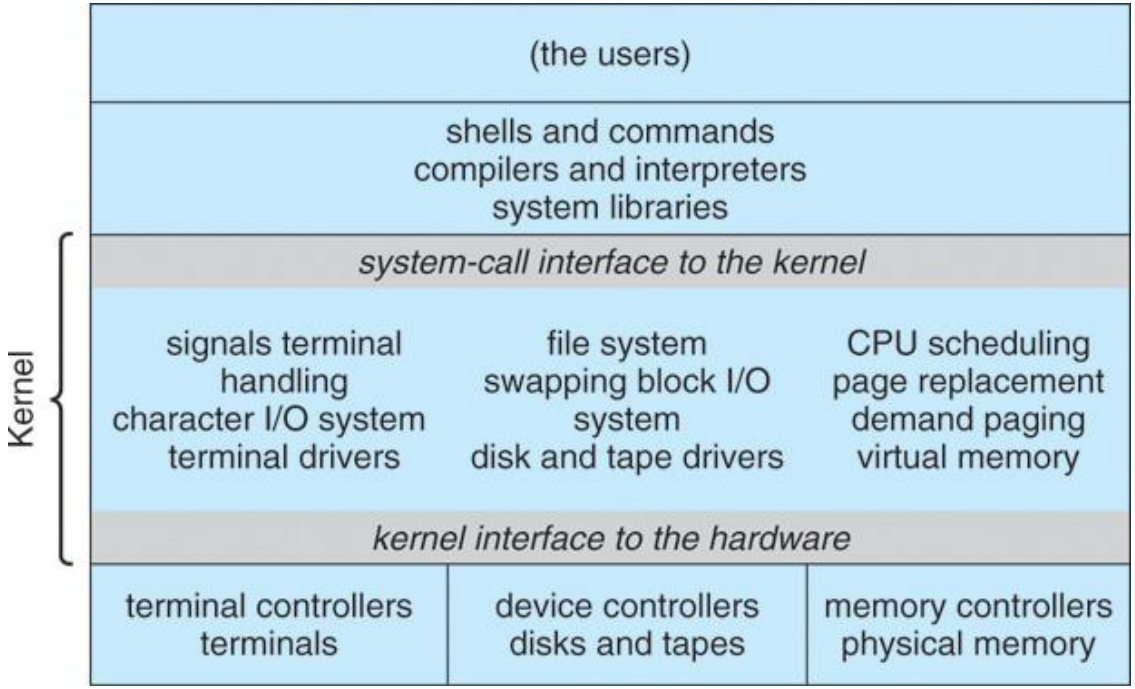


Abraham Silberschatz, ” Operating System Concepts”

System-call Interface: serves as the link to system calls made available by the operating system, intercepting function calls and invoking the necessary system call within the operating system.

System Calls

- **System calls**



Abraham Silberschatz, ” Operating System Concepts”

System Calls

- **System calls**
  - **Classification**

General Class	Specific Class	System Call

File Structure	Creating a Channel	`create()`
		`open()`
		`close()`

	Input/Output	`read()` `write()`

	Access Control	`access()`
		`chmod()`
		`umask()`

System Calls

- **System calls**
  - **Classification**

General Class	Specific Class	System Call

Process	Creation and termination	`fork()`
		`wait()`
		`exit()`
		`exec()`

	Owner and Group	`getuid()` `getgid()`
	Identity	`getpid()` `getppid()`

	Control	`signal()`
		`kill()`
		`alarm()`

System Calls

- **System calls**
  - **Classification**

General Class	Specific Class	System Call
Interprocess Communication	Pipelines	`pipe()`

	Messages	`msget()`
		`msgsnd()`
		`msgrcv()`
		`msgctl()`

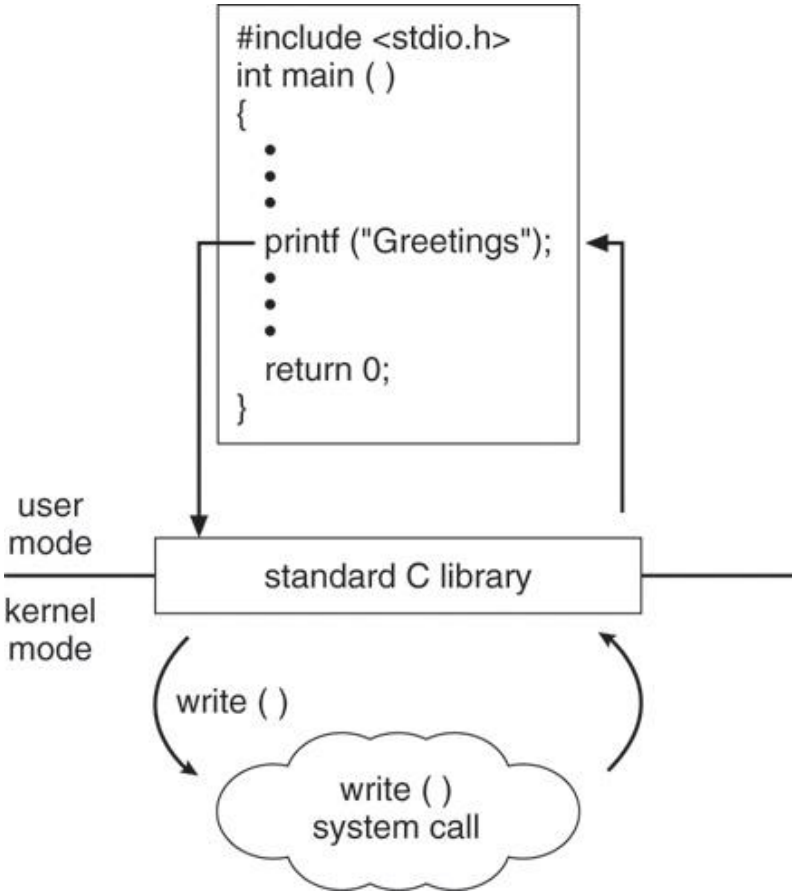
	Semaphores	`semget()` `semop()`

	Shared Memory	`shmget()`
		`shmat()`
		`shmdt()`

System Calls

- **System calls**
  - Some library functions have embedded system calls.
    - the library routine *printf* makes use of the system call *write*.

printf() -> printf() in the C library -> write() system call in kernel.



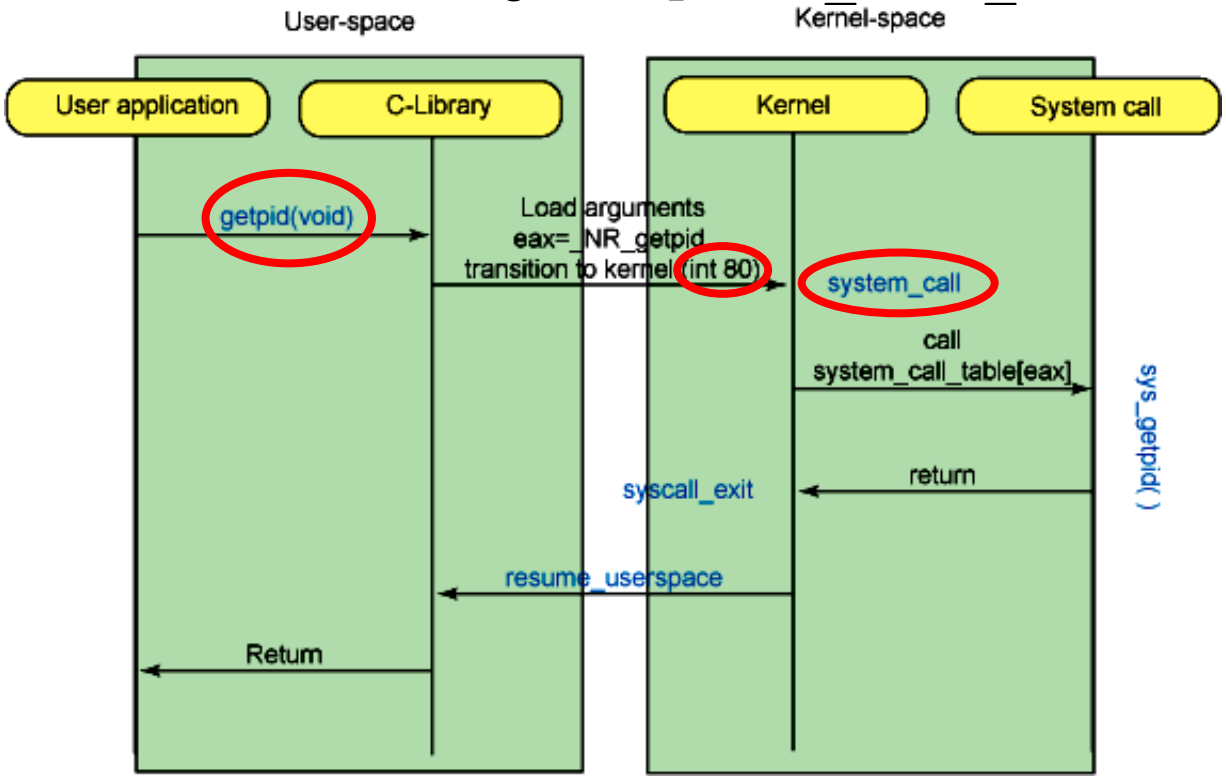
System Calls

- **System calls**
  - A system call can also be directly invoked
    - e.g. program that opens a file directly through a system call and copies its contents to standard output:

```
#include <fcntl.h>
int main()
{
    int fd, count; char buf[1000];
    fd=open("mydata", O_RDONLY);
    count = read(fd, buf, 1000);
    write(1, buf, count);
    close(fd);
}
```

System Calls

- **System calls**
  - A system call can also be directly invoked
  - when the C library has loaded the system call index and any arguments, a **software interrupt (trap)** is invoked (interrupt 0x80 in x86 architecture), which results in execution (through the **interrupt handler**) of the `system_call` function. After a few simple tests, the actual system call is invoked using the `system_call_table`.



Application Programming Interface (API)

- **Application Programming Interface (API)**
  - application developers design programs according to an ***Application Programming Interface* (API).**
  - 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.
    - the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.
  - Three of the most common APIs available to application programmers are the:
    - **Win32** API for Windows systems
    - **POSIX** API for POSIX-based systems (which includes virtually all versions of UNIX, Linux, and Mac os X)
    - **Java** API for designing programs that run on the Java virtual machine.

Application Programming Interface (API)

- **Application Programming Interface (API)**
  - **Advantages:**
    - 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.
    - Less detailed and easy to work than system calls. Regardless, there often exists a strong correlation between invoking a function in the API and its associated system call within the kernel.

Library	API
A reusable chunk of codes	A point of contact that allows interaction between running codes
It refers to the code itself	It refers to the interface
It is not an API by itself	It can be made of several libraries

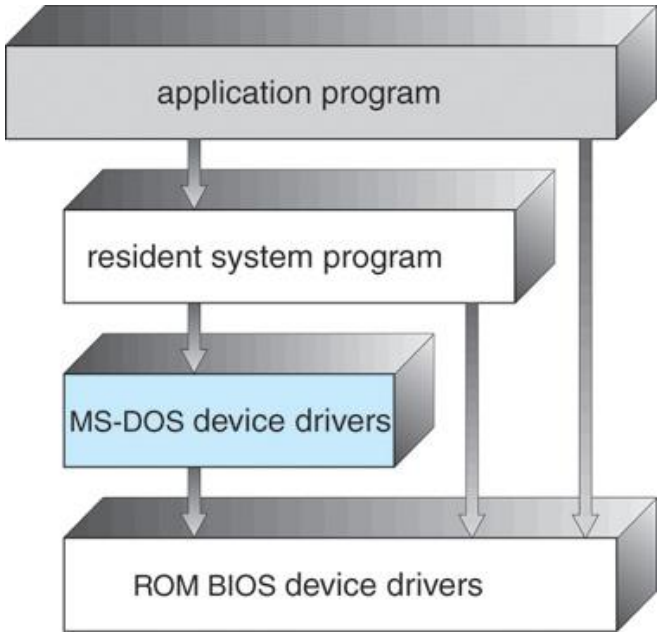
https://rapidapi.com/blog/api-vs-library/

Operating Systems Implementation

- Traditionally, operating systems have been written in assembly language.
- Nowadays, however, they are most commonly written in higher-level languages such as C or C++.
  - The Linux and Windows operating systems are written mostly in C
  - **Advantages** of using a higher-level language
    - code can be written faster
    - more compact
    - easier to understand
    - easier to port (to move to some other hardware)
  - **Disadvantages** of using a higher-level language
    - reduced speed
    - increased storage requirements

Operating Systems Structure

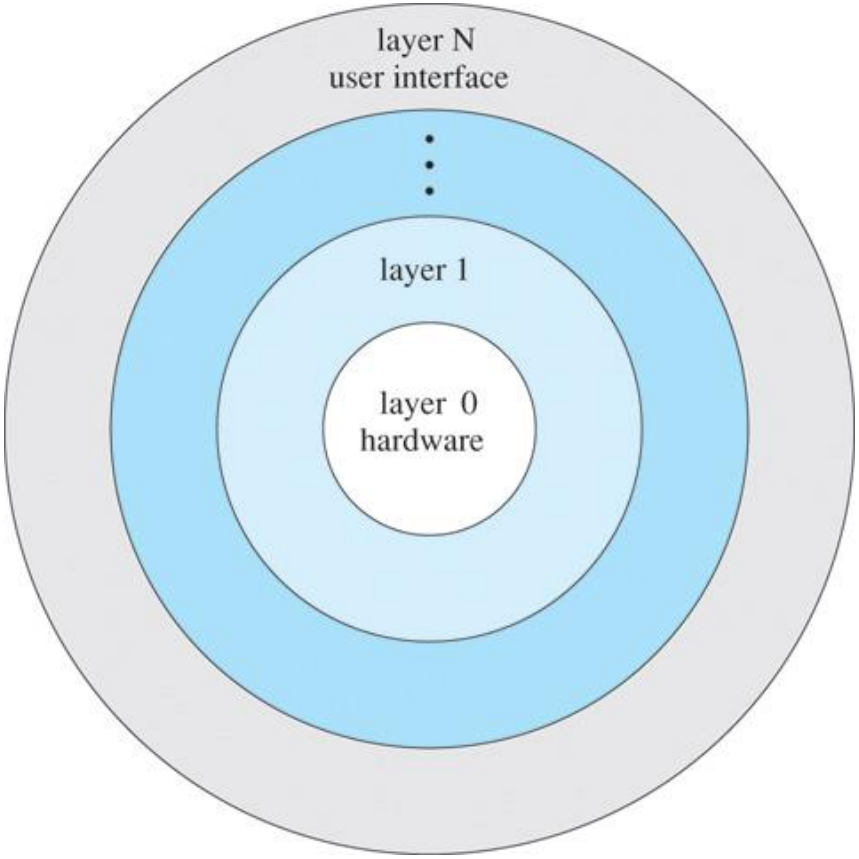
- **Simple Structure**
  - operating systems that do not have well-defined structures. Frequently, such operating systems (e.g. MS-DOS) started as small, simple, and limited systems and then grew beyond their original scope.
    - application programs are able to access the basic I/O routines to write directly to the display and disk drives



MS-DOS was limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS- DOS had no choice but to leave the base hardware accessible.

Operating Systems Structure

- **Layered Approach**
  - the operating system is broken up into a number of layers (levels):
    - **bottom layer (layer 0):** hardware;
    - **highest (layer N):** user interface.
  - layer M-consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers. Each layer is implemented with only those operations provided by lower level layers and does not need to know how these operations are implemented.

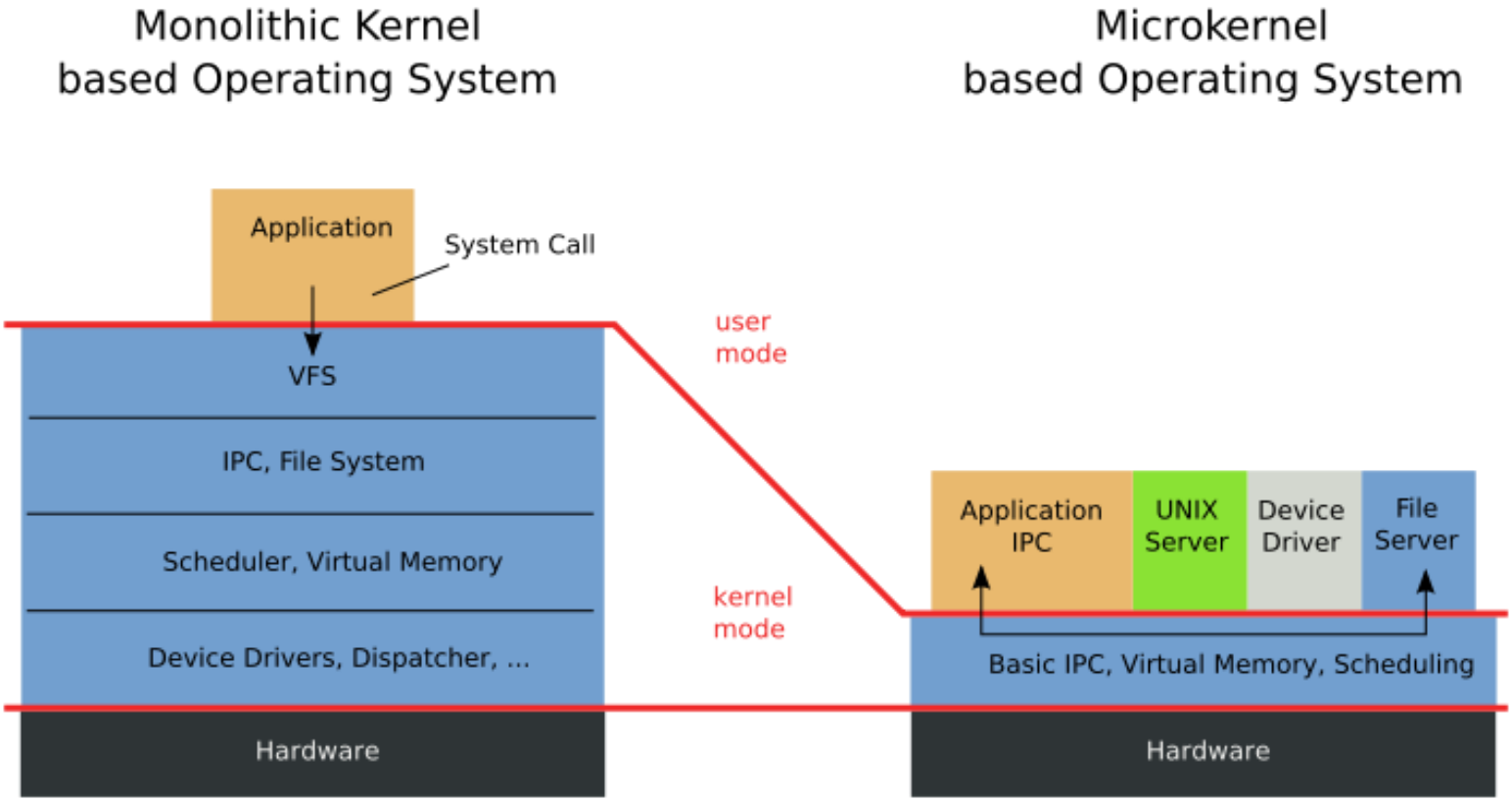


Operating Systems Structure

- **Layered Approach**
  - **Advantages:**
    - simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification.
    - each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
  - **Disadvantages:**
    - tend to be less efficient than other types.
      - For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.

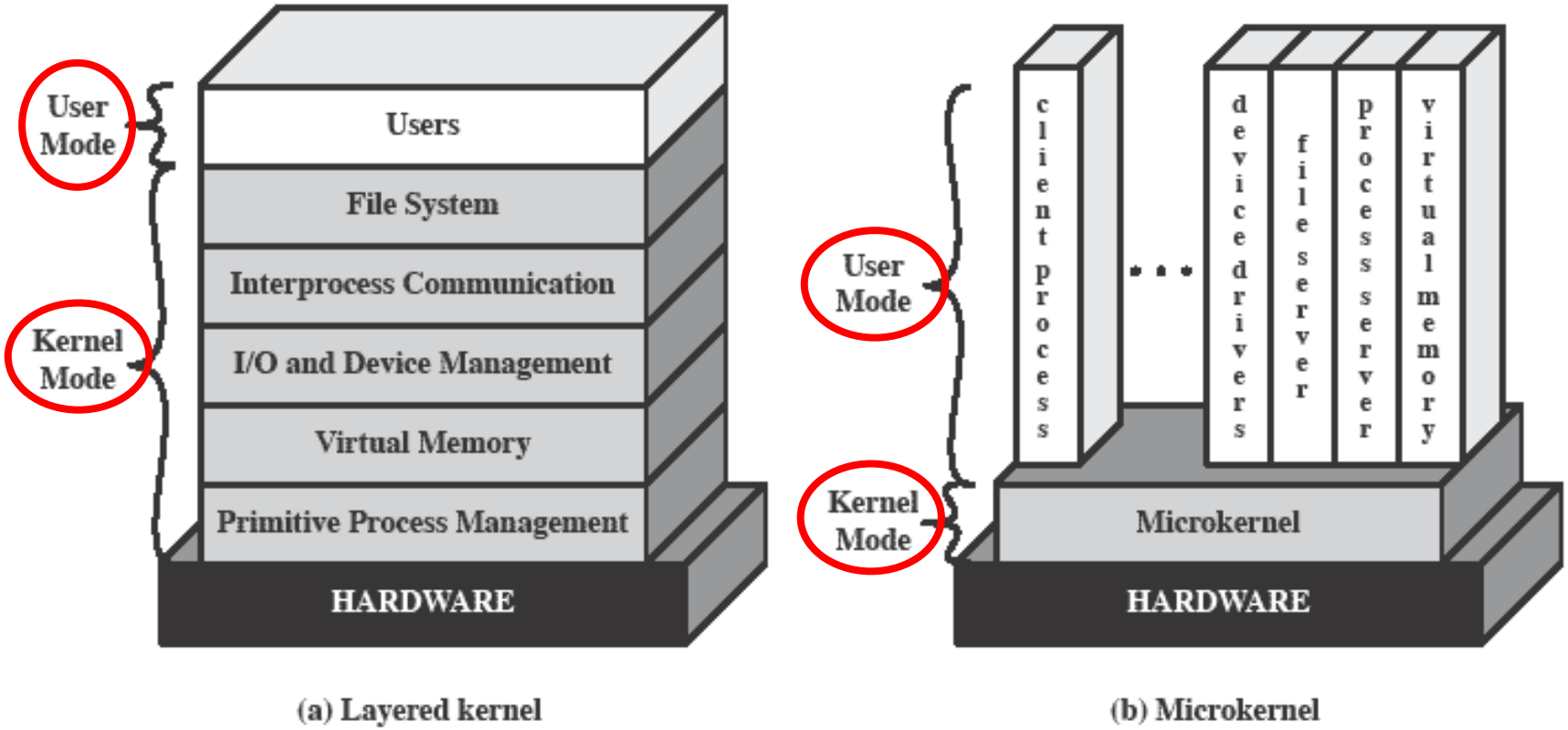
Operating Systems Structure

- **Microkernels**
  - This method removes all nonessential components from the kernel and implements them as system and user-level programs. The result is a smaller kernel.



Operating Systems Structure

- **Microkernels**
  - Layered versus Microkernel



Operating Systems Structure

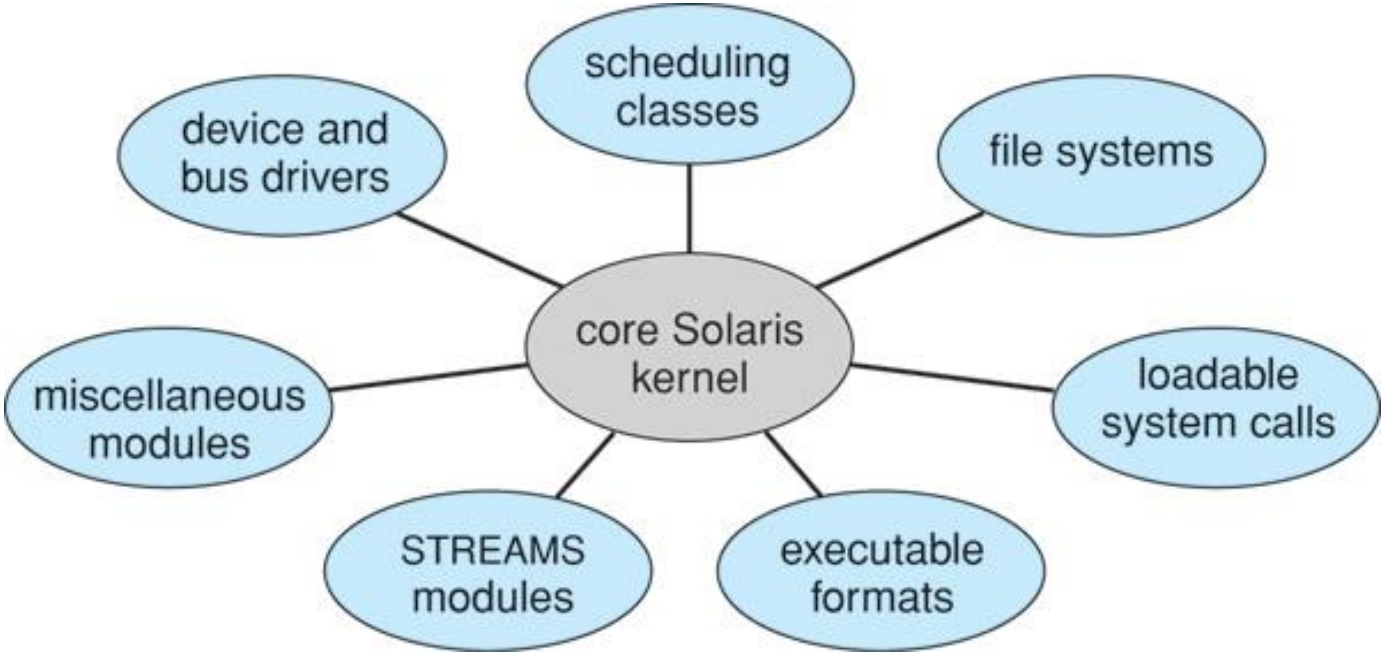
- **Microkernels**
  - **Advantages**
    - ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
    - More security and reliability, since most services are running as user-rather than kernel-processes. If a service fails, the rest of the operating system remains untouched.
  - **Disadvantages**
    - performance decreases due to increased system function overhead.

Operating Systems Structure

- **Modules**
  - kernel has a set of core components and dynamically links in additional services either during boot time or during run time.
  - uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.
  - The overall result resembles
    - a **layered system** in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module.
    - a **microkernel approach** in that the primary module has only core functions and knowledge of how to load and communicate with other modules;

Operating Systems Structure

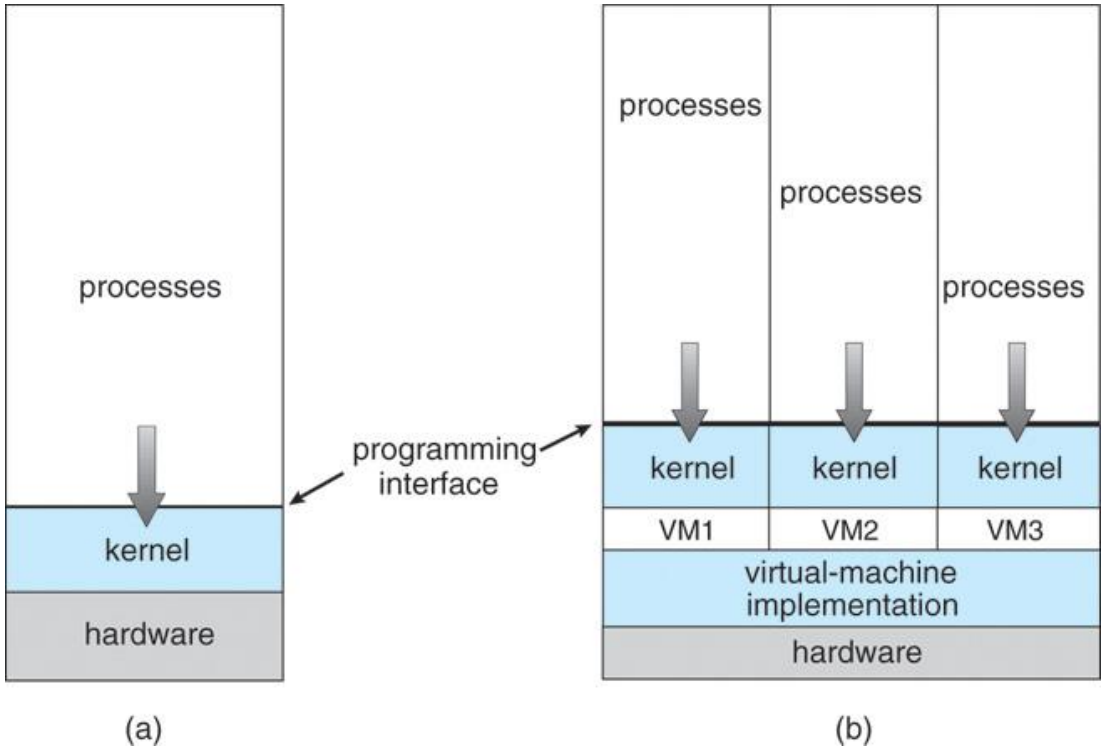
- **Modules**



Abraham Silberschatz, ” Operating System Concepts”

Virtual Machines

- **Virtual Machines**
  - abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer



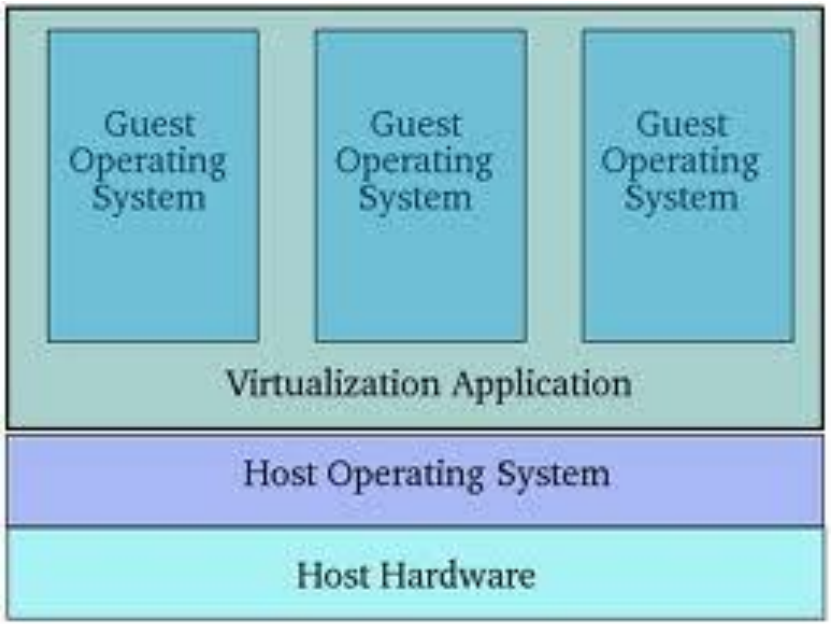
Nonvirtual machine **Virtual machine.**

Virtual Machines

- **Virtual Machines**
  - **Implementation**
    - **Just as the physical machine has two modes, so must the virtual machine.** Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode.
    - Such a transfer can be accomplished as follows.
      1. When a system call, for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine.
      2. When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call.
      3. It can then restart the virtual machine, noting that it is now in virtual kernel mode.

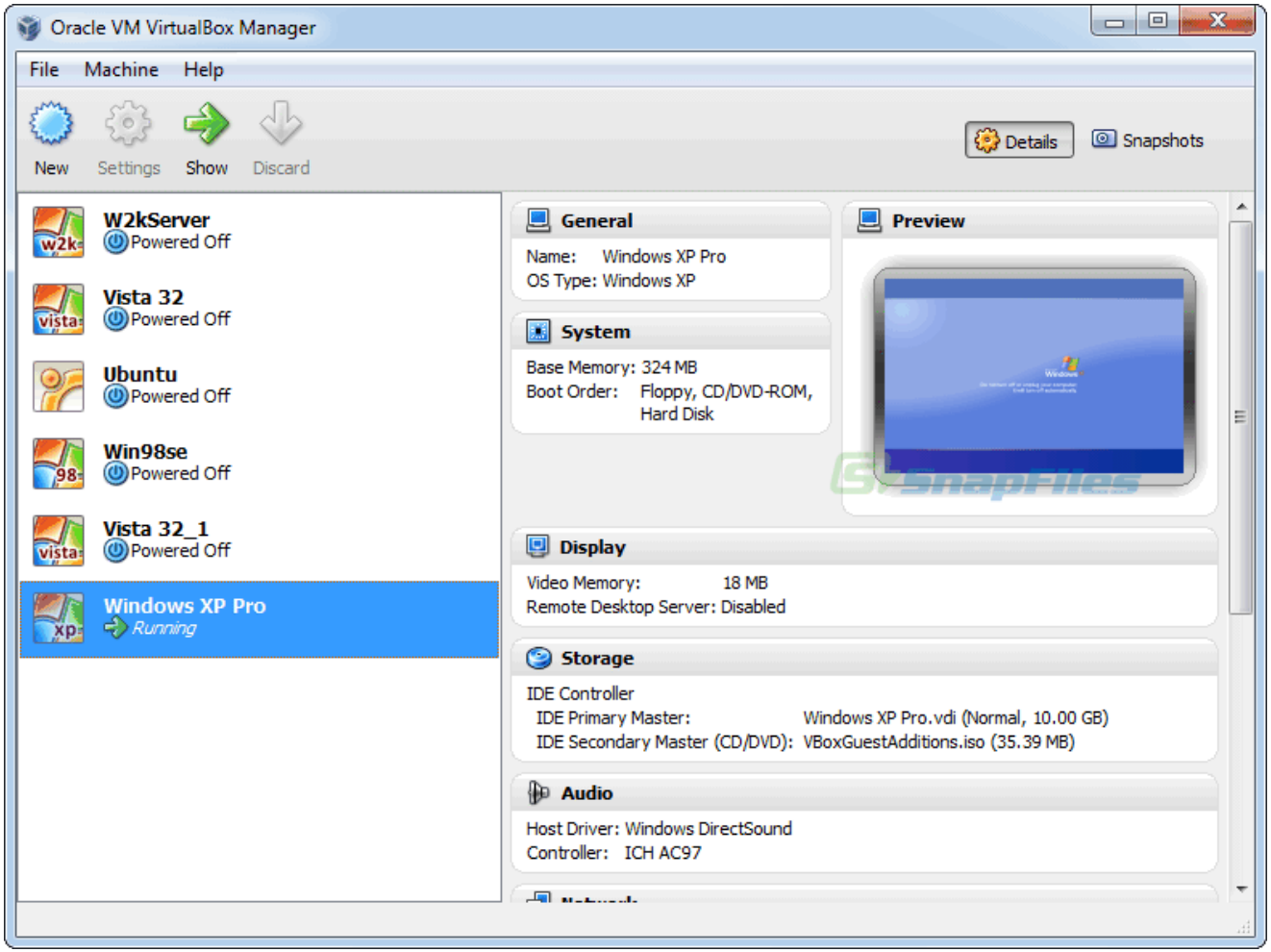
Virtual Machines

- **Virtual Machines**
  - **VirtualBox**
    - Oracle VM VirtualBox (formerly Sun VirtualBox) is an x86 virtualization software package, now developed by Oracle Corporation as part of its family of virtualization products.
    - Oracle VM VirtualBox is installed on an existing host operating system as an application allowing additional guest operating systems, each known as a Guest OS, to be loaded and run (each with its own virtual environment).



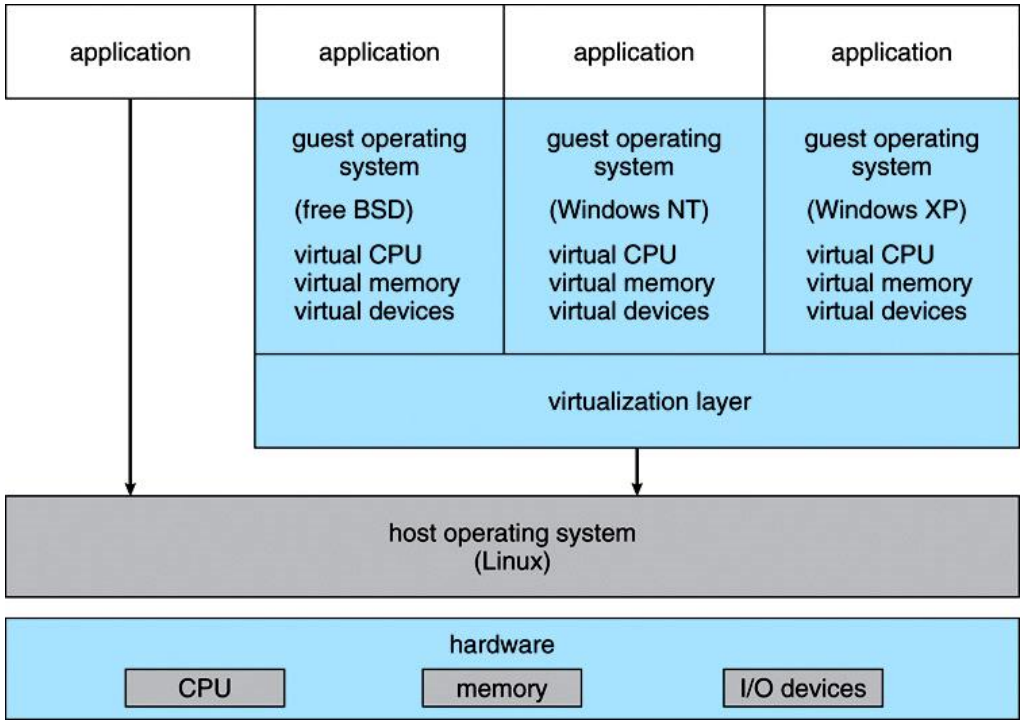
Virtual Machines

- **Virtual Machines**
  - **VirtualBox**



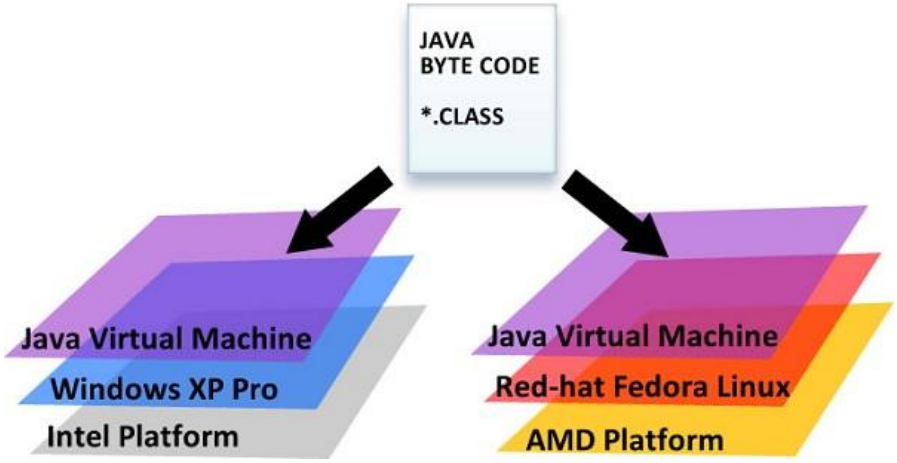
Virtual Machines

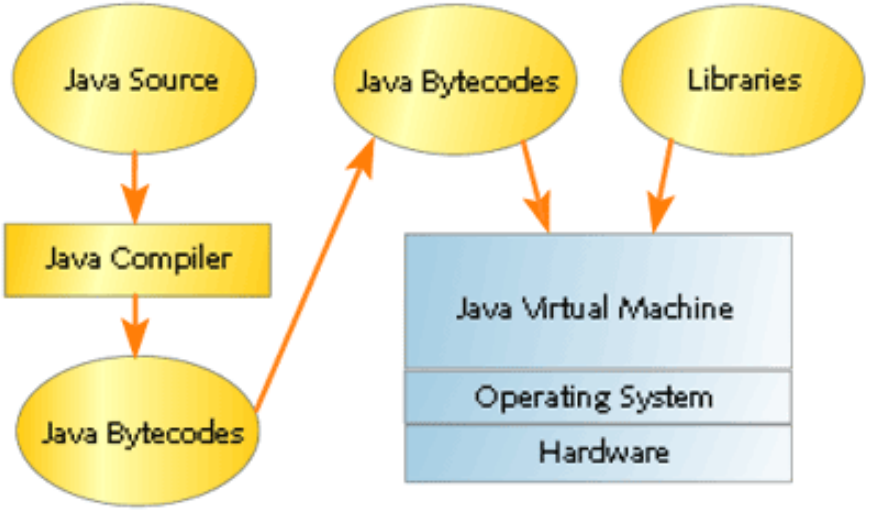
- **Virtual Machines**
  - **VMWare**
    - commercial application that abstracts Intel 80X86 hardware into isolated virtual machines.
    - runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent VMs



Virtual Machines

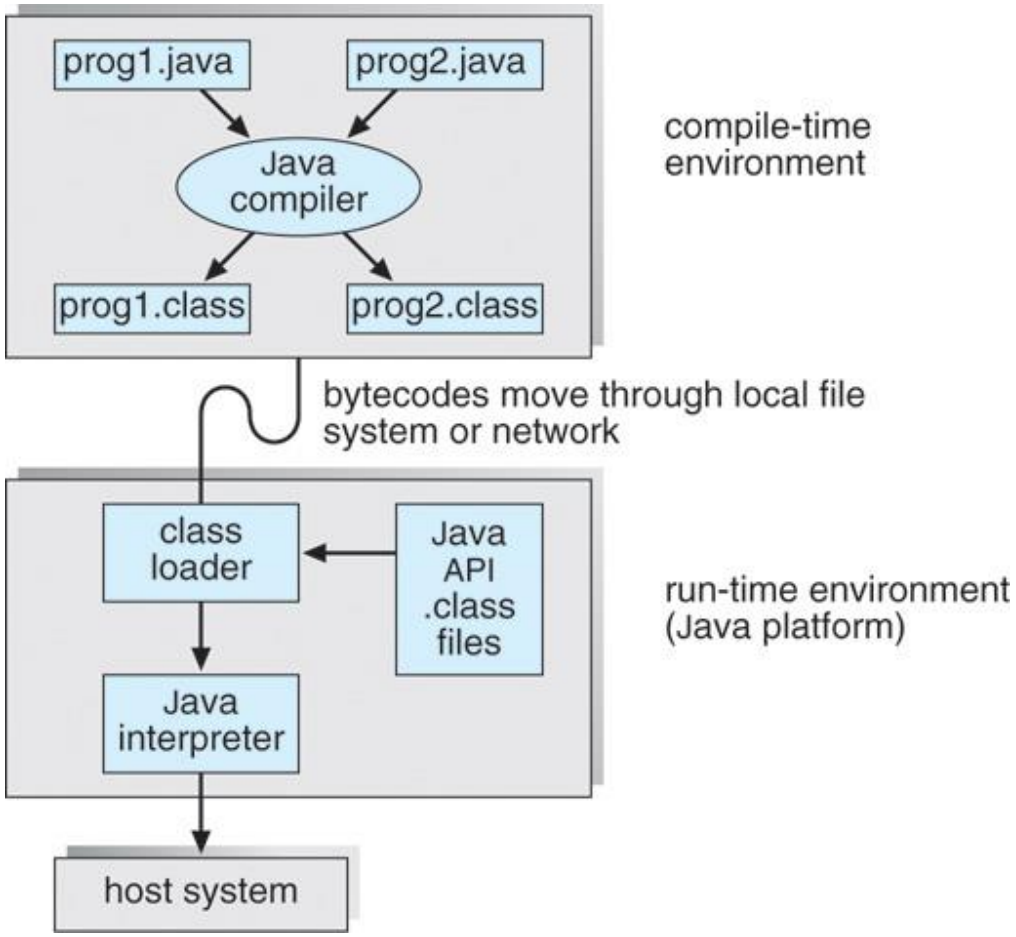
- **Virtual Machines**
  - **Java Virtual Machine (JVM)**
    - The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a web browser.
    - Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs.





Virtual Machines

- **Virtual Machines**
  - **Java Virtual Machine**
    - **Implementation**

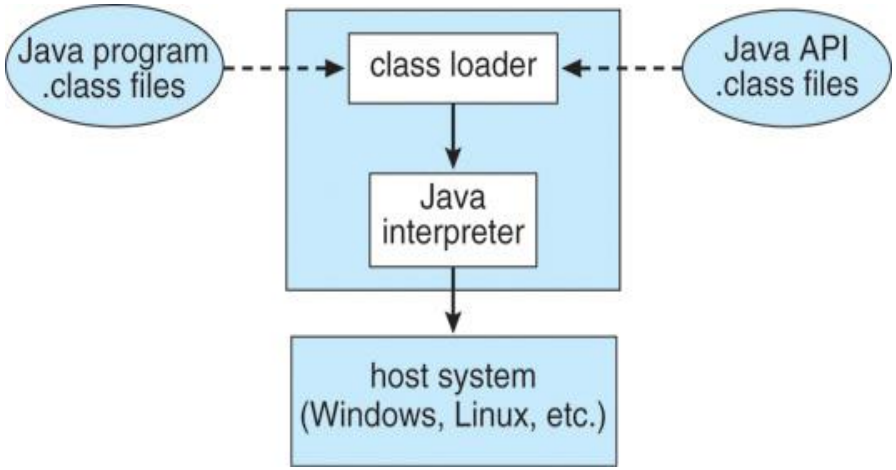


Abraham Silberschatz, " Operating System Concepts"

Just-in-time (JIT) compiler: the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system, and cached so that subsequent invocations of a method are performed using the native machine instructions and the bytecode operations need not be interpreted all over again.

Virtual Machines

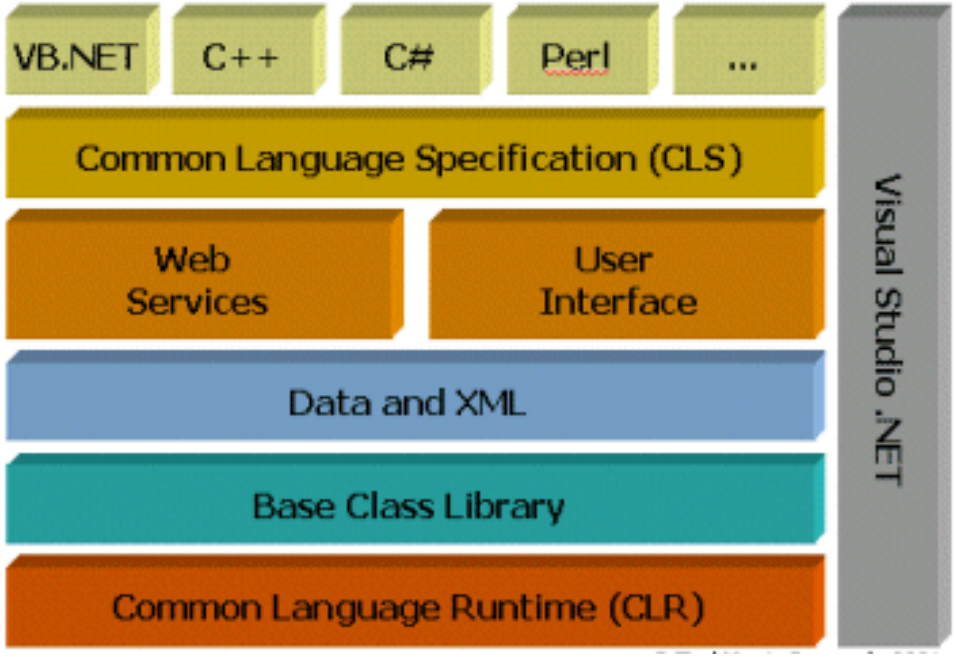
- **Virtual Machines**
  - **Java Virtual Machine**
    - **Implementation**
      - the **class loader** loads the compiled **.class** files from both the Java program and the Java API for execution by the Java interpreter.
      - After a class is loaded, the verifier checks that the .class file is a valid Java bytecode and does not overflow or underflow the stack. If the class passes verification, it is run by the **Java interpreter**.



Abraham Silberschatz, " Operating System Concepts"

Virtual Machines

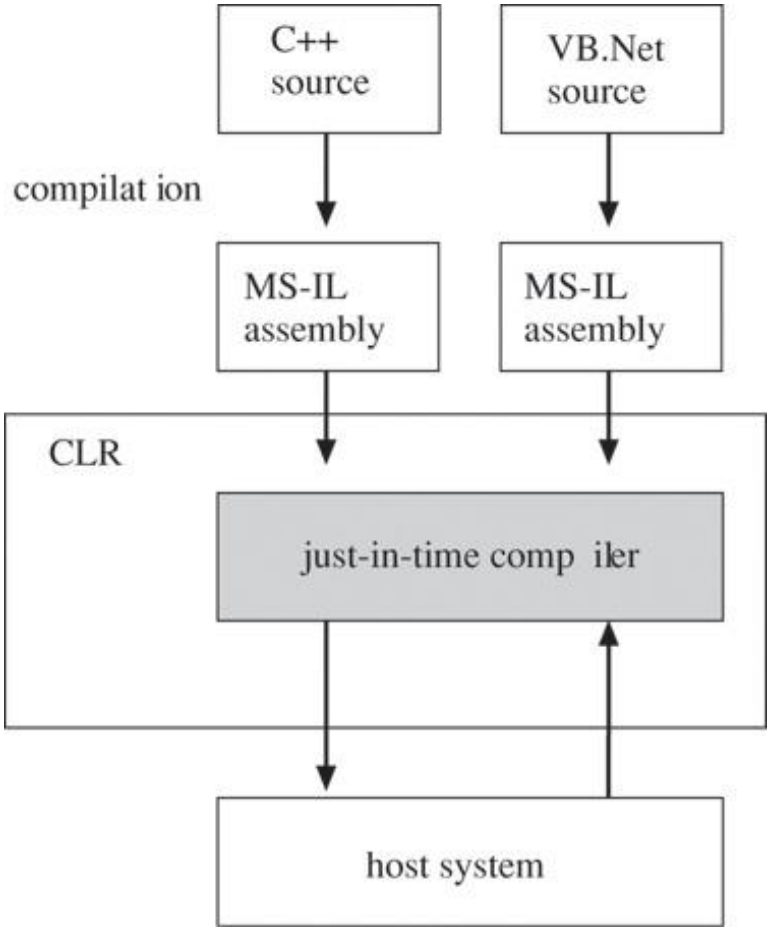
- **Virtual Machines**
  - **.Net Framework**
    - program written on the .Net framework needs not worry about the specifics of the hardware
    - Any architecture implementing .NET will be able to successfully execute the program, since it provides a virtual machine as an intermediary between the executing program and the underlying architecture



Virtual Machines

CLR - Common Language Runtime

- **Virtual Machines**
  - **.Net Framework**
    - **Implementation**
      - Programs written in languages such as C# and VB .NET are compiled into an intermediate architecture independent language called **Microsoft Intermediate Language (MS-IL)**
        - these compiled files, known as assemblies have an extension **.exe or .dll**
      - Upon the execution of the program, the **CRL (Common Language Runtime)** loads the assemblies and converts the MS-IL instructions into native code using just-in-time compilation



Abraham Silberschatz, ” Operating System Concepts”

References

- Abraham Silberschatz, ” Operating System Concepts”, 10th Edition, Wiley, 2018;
- William Stallings, “Operating Systems: Internals and Design Principles”, 9th Edition, Pearson, 2017;
- Andrew S. Tanenbaum, “Modern Operating Systems”, 4th Edition, Pearson Education, 2014;