SYSTEM CALLS

- **System calls** provide an interface to the services made available by OS.
- System calls are generally available as **routines written in C and C++,** although certain **low-level tasks** may have to be written using **assembly-language** instructions.
- Example: consider 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

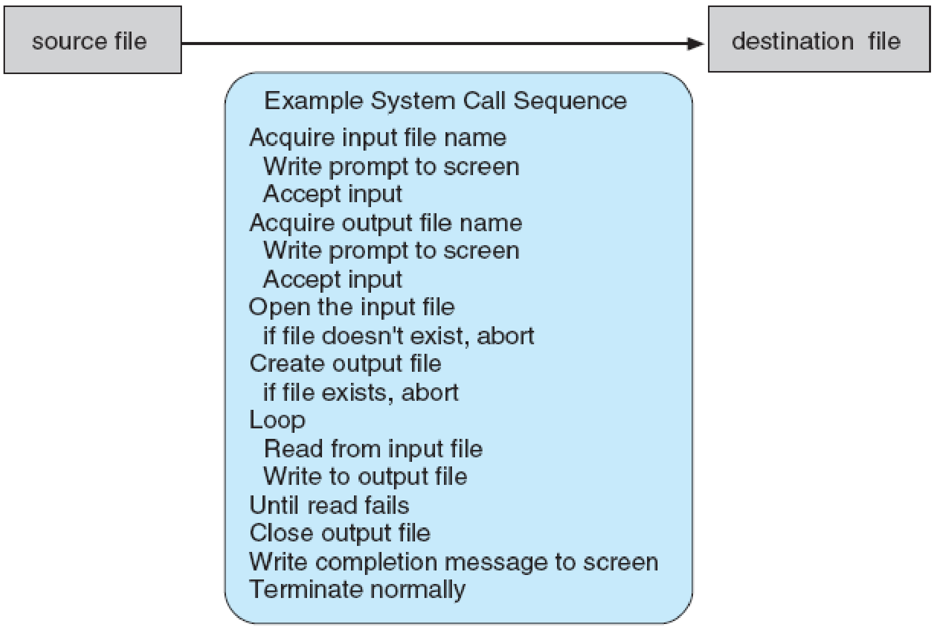


Figure 2.5 Example of how system calls are used.

- Systems execute thousands of system calls per second. Most programmers never see this level of detail.
- Application developers design programs according to **application programming interface (API)**.
- 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.

- Most common three APIs are :
    - **Windows API** for Windows systems
    - **POSIX API** for UNIX-based systems **(POSIX: Portable OS Interface in UNIX)**
    - **Java API** for programs that run on the Java virtual machine.
- Why would an application programmer prefer API rather than invoking actual system calls?
    - One benefit concerns 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.** API use is simple.
    - There exists a **strong correlation** between a function in the API and its associated system call within the kernel.
- For most programming languages, the run-time support system provides a **system call interface** that serves as the link to system calls made available by OS.
- 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 then invokes the intended system call in the OS kernel and returns the status of the system call and any return values.
- The caller need know nothing about how the system call is implemented or what it does during execution.

- Most of the details of the OS interface are hidden from the programmer by the API and are managed by the run-time support library.

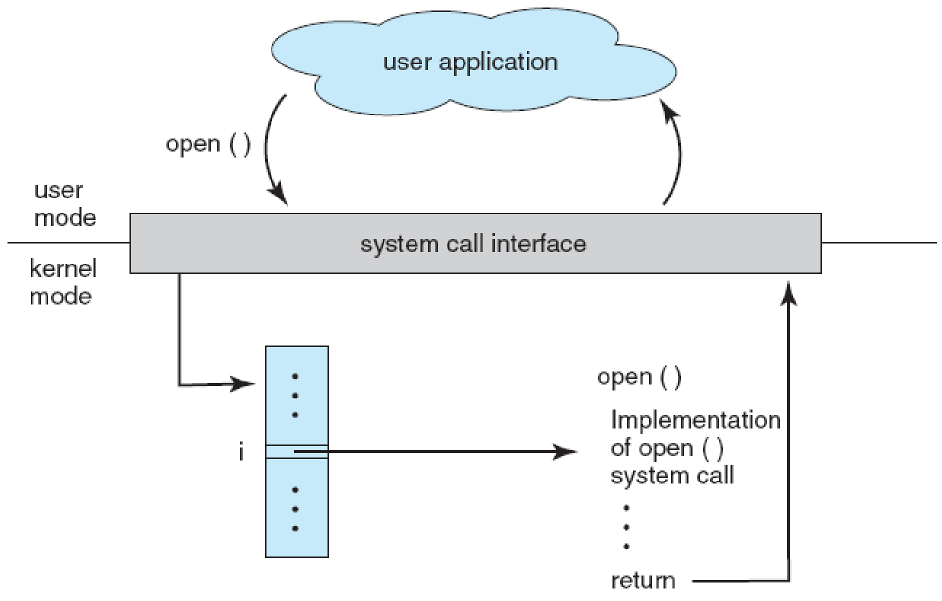


Figure 2.6 The handling of a user application invoking the `open()` system call.

- Three general methods are used to pass parameters to the OS for a system call
  - ➢ The simplest approach is to pass the parameters in **registers.**
  - ➢ If more parameters than registers, then parameters are generally stored in a **block or table** in memory and the address of the block is passed as a parameter in a register
  - ➢ Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the OS
- Some OS prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

TYPES OF SYSTEM CALLS

System calls can be grouped roughly into six major categories:

1. **Process control**
2. **File management**
3. **Device management**
4. **Information maintenance**
5. **Communication**
6. **Protection**

- Process control
    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for time
    - 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
- 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
- Communication
  - ◦ create, delete communication connection
  - ◦ send, receive messages
  - ◦ transfer status information
  - ◦ attach or detach remote devices
- Protection
  - ◦ set permission, get permission
  - ◦ allow user, deny user

1. Process Control

- A running program needs to be able to halt its execution either normally **end()** or abnormally **abort().**
- If currently running program is terminated abnormally, an error message is generated.
- The status is written to disk and may be examined by a **debugger** —a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem.
- **Alert the user** to the error and ask for guidance. Some systems may allow for **special recovery actions.**
- A process or job executing one program may want to **load()** and **execute() another program**.
- If both programs continue concurrently, it is **multiprogramming**. The system call specifically for this purpose **create_process()**
- **Process control** requires the ability to determine and reset the **attributes of a process**, including the **job's priority, its**

maximum allowable execution time, and so on (**get_process_attributes() and set_process_attributes()**).

- We may also want to terminate a job or process that we created (**terminate_process()**)
- We may want to wait for a certain amount of time to pass (**wait_time()**).
- More probably, we will want to wait for a specific event to occur (**wait_event()**). The jobs or processes should then signal when that event has occurred (**signal_event()**).
- To ensure the integrity of the data being shared, OS often provide system calls allowing a process to **lock** shared data. Then, no other process can access the data until the lock is released; **acquire_lock()** and **release_lock().**
- To start a new process, the **UNIX shell** executes a **fork()** system call. Then, the selected program is loaded into memory via an **exec()** system call, and the program is executed. When the process is done, it executes an **exit()** system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code.

2. File Management

- We may **create()** and **delete()** files. It requires the **name of the file** and perhaps some of the **file's attributes.**
- Once the file is created, we need to **open()** it and to use it. We may also **read(), write(), or reposition()**. Finally, we need to **close()** the file.
- We may need these **same sets of operations for directories**
- **File attributes** include the **file name, file type, protection codes, accounting information, and so on.**

- Two system calls, **get_file_attributes()** and **set_file_attributes(),** are required
- **Some OS** provide many more calls, such as calls for file **move()** and **copy()**. Others might provide an API that **performs those operations using other system calls**

3. Device Management

- The various **resources** controlled by OS can be thought of **as devices**.
- Some of these devices are **physical devices** (for example, disk drives), while others can be thought of as **abstract or virtual devices** (for example, files).
- A system with multiple users may require to first **request()** a device, to ensure exclusive use of it. After we are finished with the device, we **release()** it. These functions are similar to the open() and close() system calls for files.
- We can **read(), write(),** and **reposition()** the device, just as we can with files.
- The **similarity between I/O devices and files** merge the two into a combined **file–device structure**.
- **Same set of system calls is used on both files and devices**. I/O devices are identified by special file names

4. Information Maintenance

- To return the current time and date, **time()** and **date()**.
- Other system calls may return information about the system, such as the **number of current users**, the **version number of the OS**, the **amount of free memory or disk space, and so on**.

- Another set of system calls is helpful in **debugging** a program. Many systems provide system calls to **dump()** memory (display the contents of memory for tracing). This provision is useful for debugging.
- OS keeps information about all its processes and may reset the process information **(get_process_attributes() and set_process_attributes()).**

5. Communication

- **Two common models of inter-process communication (IPC):**
  1. **Message passing model**
  2. **Shared-memory model.**
- In the **message-passing model**,
  - ➢ Communicating processes **exchange messages** with one another to transfer information.
  - ➢ Messages can be exchanged between the processes either **directly or indirectly through a common mailbox.**
  - ➢ Before communication can take place, a connection must be opened.
  - ➢ The name of the other communicator must be known
  - ➢ Be it another process on the same system or a process on another computer connected by a communications network.
  - ➢ Each computer in a network has a **host name** by which it is commonly known.
  - ➢ A host also has a network identifier, such as an **IP address**.
  - ➢ Each process has a **process name**, and this name is translated into an identifier

- The **get_hostid() and get_processid()** system calls do this translation.
- **open_connection()** and **close_connection()** system calls are used
- The recipient process usually must give its permission for communication to take place with an **accept_connection()** call. Otherwise **reject_connection()** or **wait_for_connection()** may be used
- The source of the communication, known as the **client**, and the receiving node, known as a **server,** then exchange messages by using **send_message()** and **receive_message()** system calls.
- The **close_connection()** call terminates the communication.

- In the **shared-memory model**,
  - **Normally, the OS tries to prevent one process from accessing another process's memory.**
  - **Shared memory requires that two or more processes agree to remove this restriction.**
  - Processes use **shared_memory_create() and shared_memory_attach()** system calls to create and gain access to regions of memory owned by other processes.
  - They can then exchange information by **read()** and **write()** in the shared areas.

- **Message passing is useful for exchanging smaller amounts of data**
- **It is also easier to implement**
- **It is used for inter-computer communication**

- **Shared memory allows maximum speed and convenience of communication**
- It can be used when transfer **takes place within a computer.**
- Problems exist, however, in the areas of **protection and synchronization between the processes sharing memory.**

6. Protection

- **Protection provides a mechanism for controlling access to the resources provided by a computer system.**
- System calls providing protection include **set_permission() and get_permission(),** which manipulate the permission settings of resources such as files and disks.
- The **allow_user() and deny_user()** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

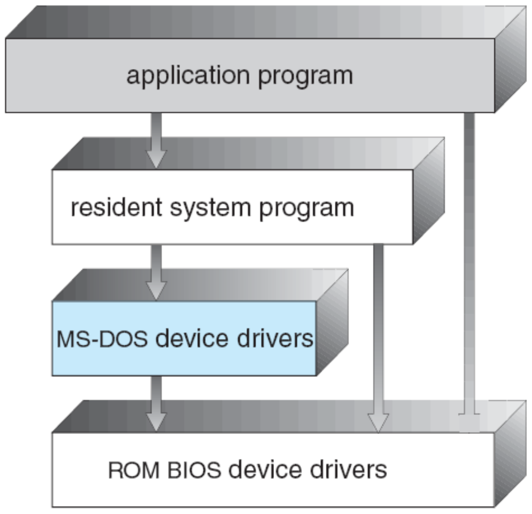
OPERATING SYSTEM STRUCTURE

1. **Simple Structure (Monolithic Structure)**
2. **Layered Approach**
3. **Micro-kernels**
4. **Modular approach**

1. Simple Structure (Monolithic Structure)

- It does **not have well-defined structures**.
- Such systems **started as small, simple, and limited systems and then grew beyond their original scope.**
- **MS-DOS** is an example of such a system.

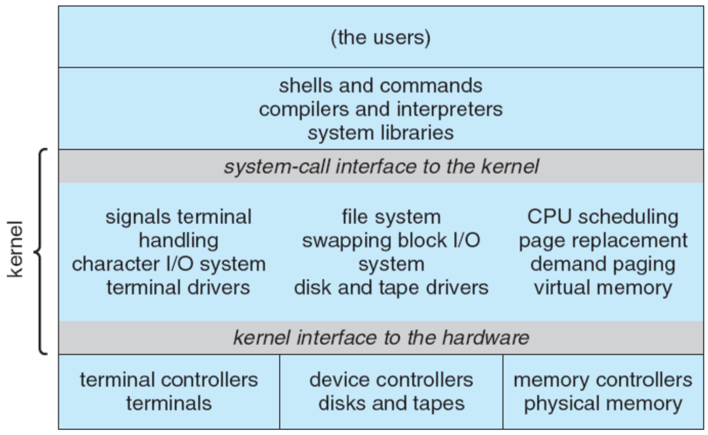
- It was originally designed and implemented by a few people who had no idea that it would become so popular.
- It was written to provide the most functionality in the least space, so it was not divided into modules carefully.
- In MS-DOS, the interfaces and levels of functionality are not well separated.
- Application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- Such freedom leaves MS-DOS **vulnerable to malicious programs**, causing entire system crashes when user programs fail.
- MS-DOS was also **limited by the hardware of its era**.



MS DOS Structure

- Another example of limited structuring is the **original UNIX** OS.
- UNIX initially was limited by hardware functionality.
- It consists of **two separable parts: the kernel and the system programs**
- **Kernel is the core part of an OS**

- The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.



Traditional UNIX system structure

- Everything below the system-call interface and above the physical hardware is the kernel.
- **The kernel provides file systems, CPU scheduling, memory management, and other OS functions through system calls**.
- A large amount of functionality to be combined into one level.
- This monolithic structure was **difficult to implement and maintain**.

2. Layered Approach

- OS can be broken into pieces that are smaller and more appropriate than those allowed by the original MS DOS or UNIX.
- OS can then retain much greater control over the computer and over the applications that make use of that computer.
- Implementers have more freedom in changing the inner working of the system and in creating modular OS

- Under a top-down approach, the overall functionality and features are determined and are separated into components.
- Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.
- Layered approach, in which the OS is broken into a number of layers (levels)
- The bottom layer is the hardware (Layer 0)
- Highest is the user interface (Layer N)

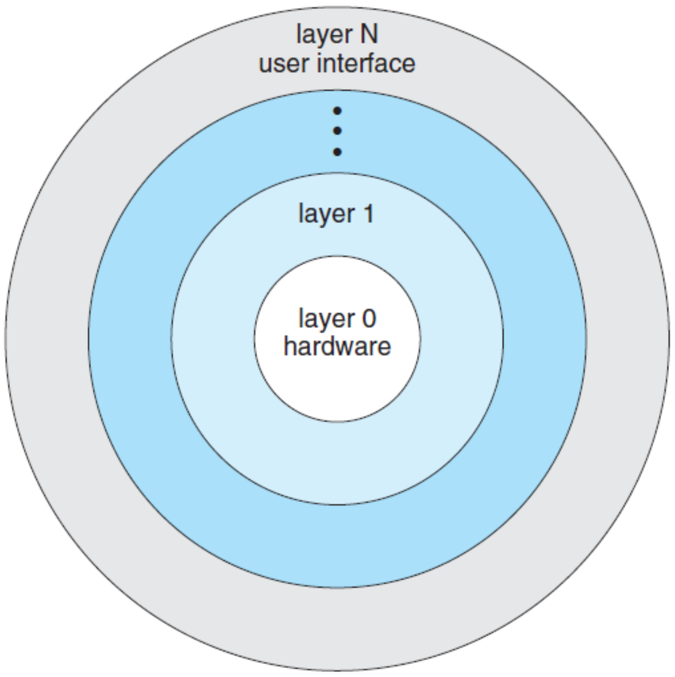


Figure 2.13 A layered operating system.

- Layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.
- A typical OS 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.

- **Advantage of the layered approach is 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.
- The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions.
- Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on.
- 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.
- Each layer is implemented with only those operations provided by lower level layers.
- A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
- **The major difficulty with the layered approach involves appropriately defining the various layers.**
- Because a layer can use only lower-level layers, careful planning is necessary.
- Problem with layered implementations is that they tend to be **less efficient than other types.**

- At each layer, the parameters may be modified; data may need to be passed, and so on.
- Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a non-layered system.

3. Micro-kernels

- As UNIX expanded, the kernel became large and difficult to manage.
- **Micro-kernel** method structures OS by removing all nonessential components from the kernel and implementing them as system and user level programs.
- The result is a smaller kernel.
- There is a common agreement regarding which services should remain in the kernel and which should be implemented in user space.
- Micro-kernels provide minimal process and memory management, in addition to a communication facility.
- The main function of the micro kernel is to provide a communication facility between the client program and the various services that are also running in user space.
- **Communication is provided by *message passing model***
- The client program and service never interact directly. Rather they communicate indirectly by exchanging messages with the micro-kernel.
- **Benefit of the microkernel approach** is ease of extending the OS.
- All new services are added to user space and consequently do not require modification of the kernel.
- It is easier to port from one hardware design to another.

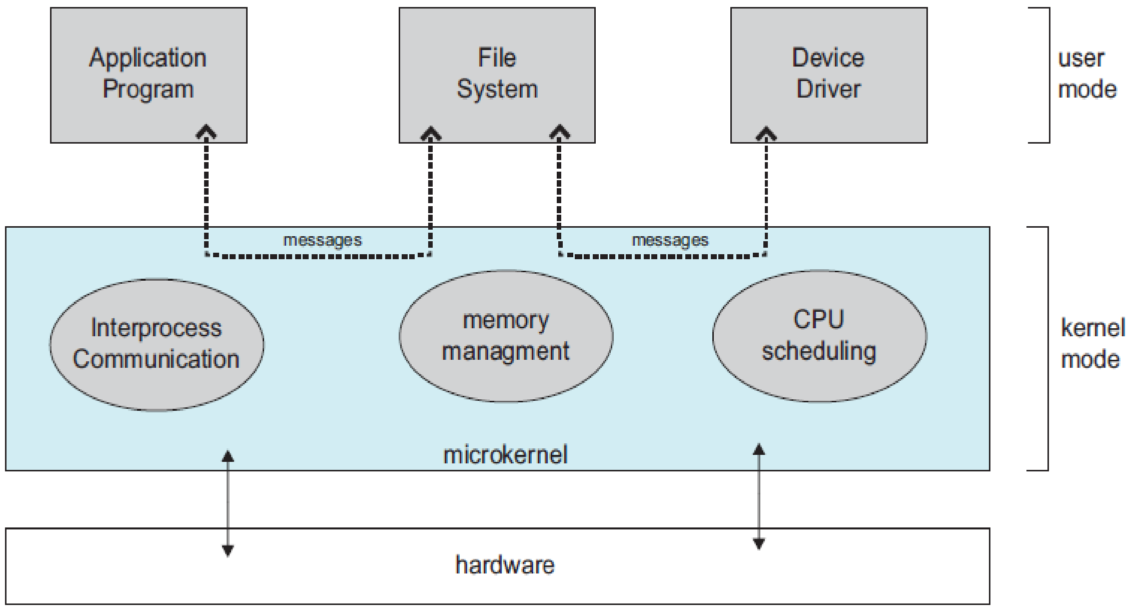


Figure 2.14 Architecture of a typical microkernel.

- The microkernel also provides more security and reliability, since most services are running as user, rather than kernel-processes.
- If a service fails, the rest of the OS remains untouched.
- **Unfortunately, micro-kernels can suffer from performance decreases due to increased system function overhead.**
- Eg: Windows NT.

4. Modular approach

- The best current methodology for OS design involves using object-oriented programming techniques to create a modular kernel.
- Here, the kernel has a set of core components and links in additional services either during boot time or during run time.

- Such a strategy uses **dynamically loadable modules** and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS.
- The idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running.
- Linking services dynamically is preferable than adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
- For example, the Solaris OS structure is organized around a core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers

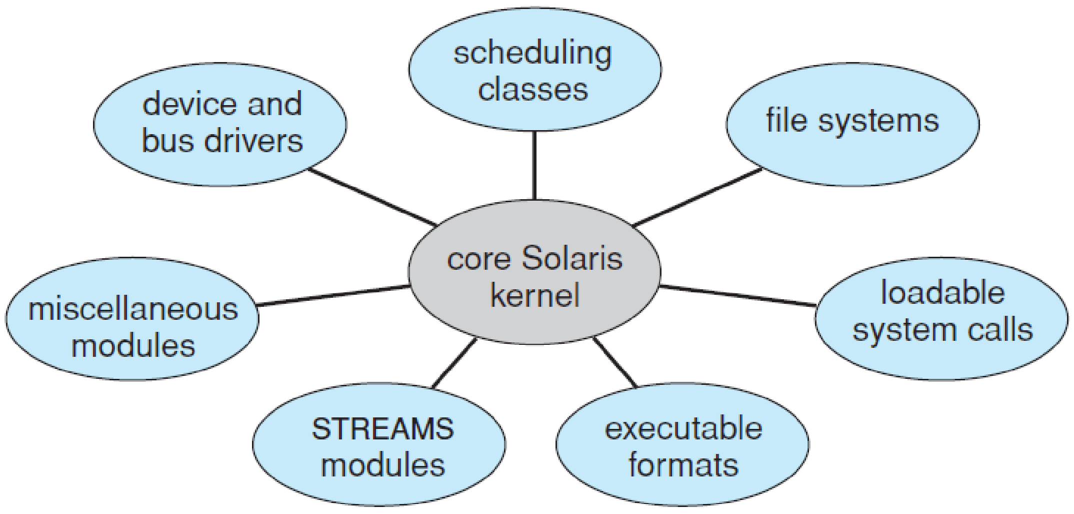


Figure 2.15 Solaris loadable modules.

- 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.
- Furthermore, the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules
- But it is more efficient, because modules do not need to invoke message passing in order to communicate.
- Linux also uses loadable kernel modules, primarily for supporting device drivers and file systems.

SYSTEM BOOT PROCESS

- The procedure of starting a computer by loading the kernel is known **as** ***booting*** the system.
- On most computer systems, a small piece of code known as the **bootstrap program or bootstrap loader** locates the kernel, loads it into main memory, and starts its execution.
- Some computer systems, such as PCs, use a **two-step process** in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.
- When a CPU is powered up or rebooted, the instruction register is loaded with a predefined memory location, and execution starts there.
- At that location is the initial bootstrap program. This program is in the form of read-only memory (**ROM**),

because the RAM is in an unknown state at system startup.
- ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.
- **The bootstrap program can perform a variety of tasks.**
- One task is to run diagnostics to determine the state of the machine.
- If the diagnostics pass, the program can continue with the booting steps.
- It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory.
- **Some systems-such as cellular phones, PDAs, and game consoles-store the entire OS in ROM.**
- Storing in ROM is suitable for small OS
- A **problem** with this approach is that changing the bootstrap code requires changing the ROM hardware chips.
- Some systems resolve this problem by using erasable programmable read-only memory (**EPROM**), which is read only except when explicitly given a command to become writable.
- **All forms of ROM are also known as firmware, since their characteristics fall somewhere between those of hardware and those of software.**
- A **problem with firmware** in general is that executing code there is slower than executing code in RAM.
- Some systems store OS in firmware and copy it to RAM for fast execution.
- A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

- For large OS or for systems that change frequently, the bootstrap loader is stored in firmware, and the OS is on disk.
- In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block**
- The program stored in the boot block may be sophisticated enough to load the entire OS into memory and begin its execution.
- More typically, it is simple code (as it fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program.
- A disk that has a boot partition is called a **boot disk or system disk.**