Operating-System Structures

Contents of Lecture:

- Operating System Services
- System Calls
- Types of System Calls
- Operating System Structure

References for This Lecture:

- ✓ *Abraham Silberschatz, Peter Bear Galvin and Greg Gagne, Operating System Concepts, 9th Edition,* ***Chapter 2***

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users.

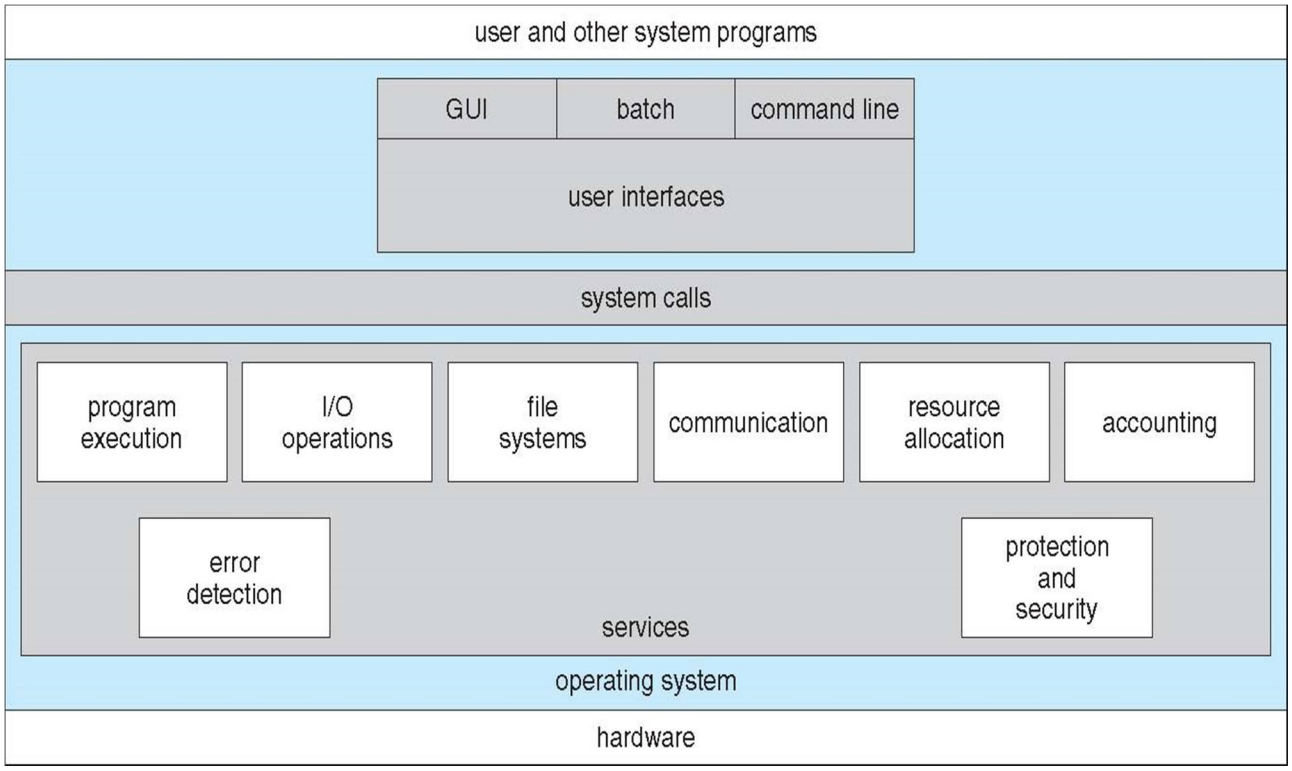


Figure 2.1 A view of operating system services.

- 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**).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch.**
  - ✓ **Program execution:** The system must be able to load a program into memory and to run that program, end 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. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - ✓ **Communications:** Processes may exchange information, on the same computer or between computers over a network.
    - Communications may be via shared memory or through message passing (packets moved by the OS).
  - ✓ **Error detection:** OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program.
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

- 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 - CPU cycles, main memory, file storage, I/O devices.
  - ✓ **Accounting:** To keep track of which users 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.

System Calls

- System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

Example of System Calls:

- 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 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.

  - ✓ Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call.

  - ✓ Possible error conditions for each operation can require additional system calls.
    - When the program tries to open the input file, for example, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call).
    - If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call).
    - Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

  - ✓ When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call).

- ✓ Each read and write must return status information regarding various possible error conditions.
  - On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error).
  - The write operation may encounter various errors, depending on the output device (for example, no more disk space).

- ✓ Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in next figure (Figure 2.5).

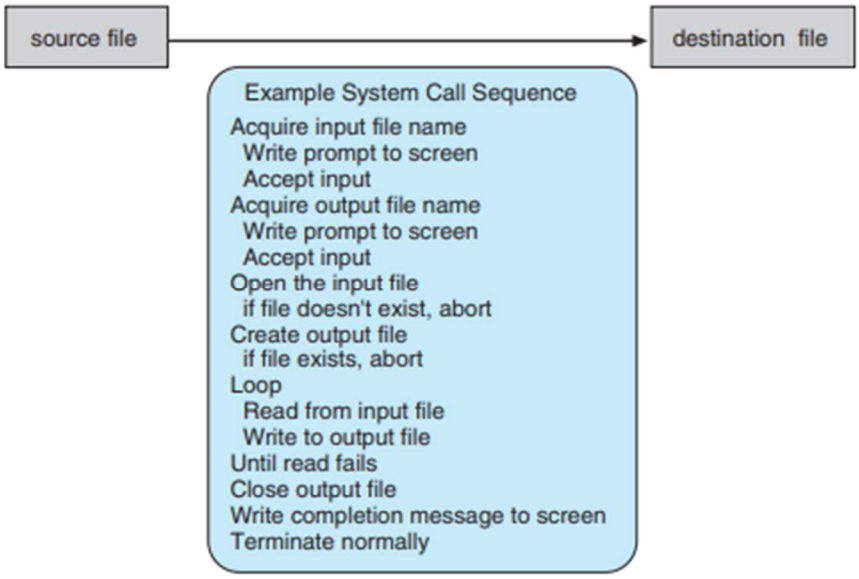


Figure 2.5 Example of how system calls are used

- ❖ As you can see, even simple programs may make heavy use of the operating system. Systems execute thousands of system calls per second.

- ❖ Most programmers never see this level of detail, however. Typically, 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.

❖ **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:

```
#include <unistd.h>

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

return value — function name — parameters

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.

❖ Three of the most common APIs available to application programmers are:

1. Windows (Win32) API for Windows systems.
2. The POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OS X).
3. The Java API for programs that run on the Java virtual machine (JVM).

❖ A programmer accesses an API via a library of code provided by the operating system.

✓ In the case of UNIX and Linux for programs written in the C language, the library is called libc.

Note that

The system-call names used throughout this text are generic examples. Each operating system has its own name for each system call

System Call Implementation

- For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) 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 the operating system.

- 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 operating-system 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
  - ✓ Just needs to obey API and understand what OS will do as a result call
  - ✓ Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

- The relationship between an API, the system-call interface, and the operating system is shown in next figure (Figure 2.6), which illustrates how the operating system handles a user application invoking the open() system call.

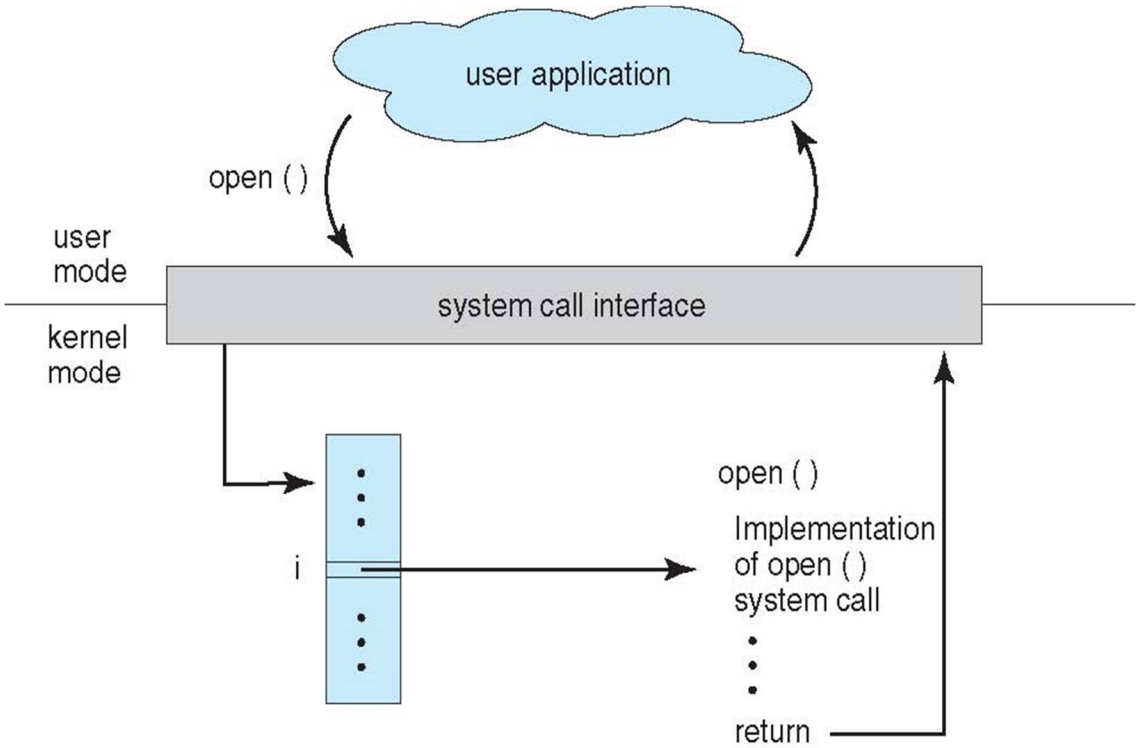


Figure 2.6 The handling of a user application invoking the open() system call

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

- Three general methods used to pass parameters to the OS
  1. Simplest: pass the parameters in registers
     - In some cases, may be more parameters than registers

  2. Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register, next figure (Figure 2.7).
     - This approach taken by Linux and Solaris

  3. Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.

Some operating systems prefer the block and stack methods do not limit the number or length of parameters being passed

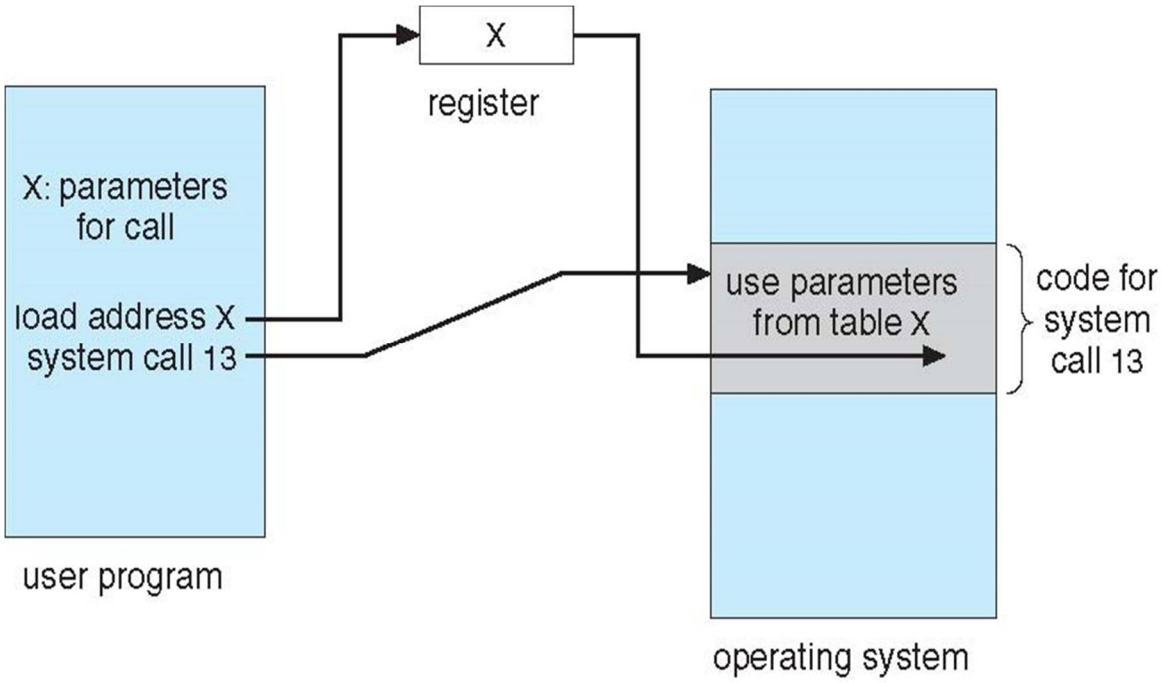


Figure 2.7 Passing of parameters as a table

Types of System Calls

- Process control
  - ✓ create process, terminate process
  - ✓ end, abort
  - ✓ load, execute
  - ✓ get process attributes, set process attributes
  - ✓ wait for time
  - ✓ wait event, signal event
  - ✓ allocate and free memory
  - ✓ Dump memory if error
  - ✓ **Debugger** for determining **bugs, single step** execution
  - ✓ **Locks** for managing access to shared data between processes

- File management
  - ✓ create file, delete file
  - ✓ open, close file
  - ✓ read, write, reposition
  - ✓ get and 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 and set process, file, or device attributes

- Communications
  - ✓ create, delete communication connection
  - ✓ send, receive messages if message passing model to host name or process name
  - ✓ From client to server
  - ✓ Shared-memory model create and gain access to memory regions
  - ✓ transfer status information
  - ✓ attach and detach remote devices

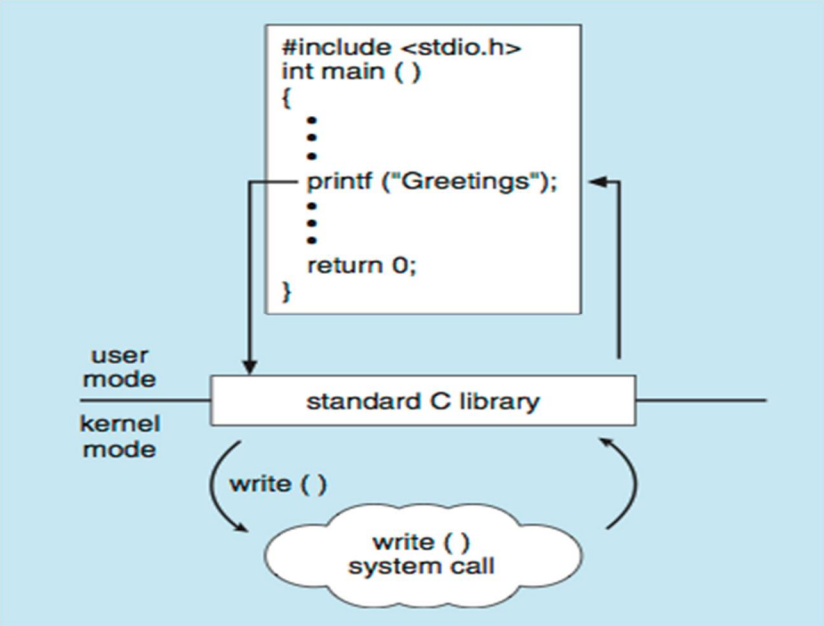
- Protection
  - ✓ Control access to resources
  - ✓ Get and set permissions
  - ✓ Allow and deny user access

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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Standard C Library Example

- The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, C program invokes the printf() statement.
  - ✓ The C library intercepts this call and invokes the necessary system call (or calls) in the operating system in this instance, the write() system call. The C library takes the value returned by write() and passes it back to the user program. This is shown below:



Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - ✓ Simple structure – MS-DOS
  - ✓ More complex -- UNIX
  - ✓ Layered – an abstrcation
  - ✓ Microkernel –Mach

Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
  - ✓ Not divided into modules
  - ✓ Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

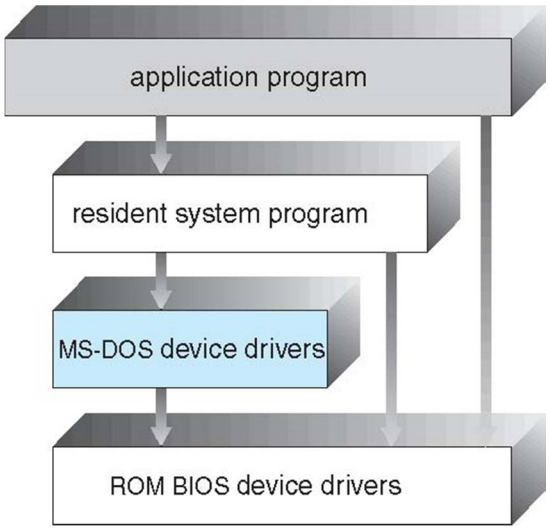


Figure 2.11 MS-DOS layer structure.

Non Simple Structure -- UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.

- The UNIX OS consists of two separable parts:
  - ✓ Systems programs
  - ✓ The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

❖ Traditional UNIX System Structure beyond simple but not fully layered

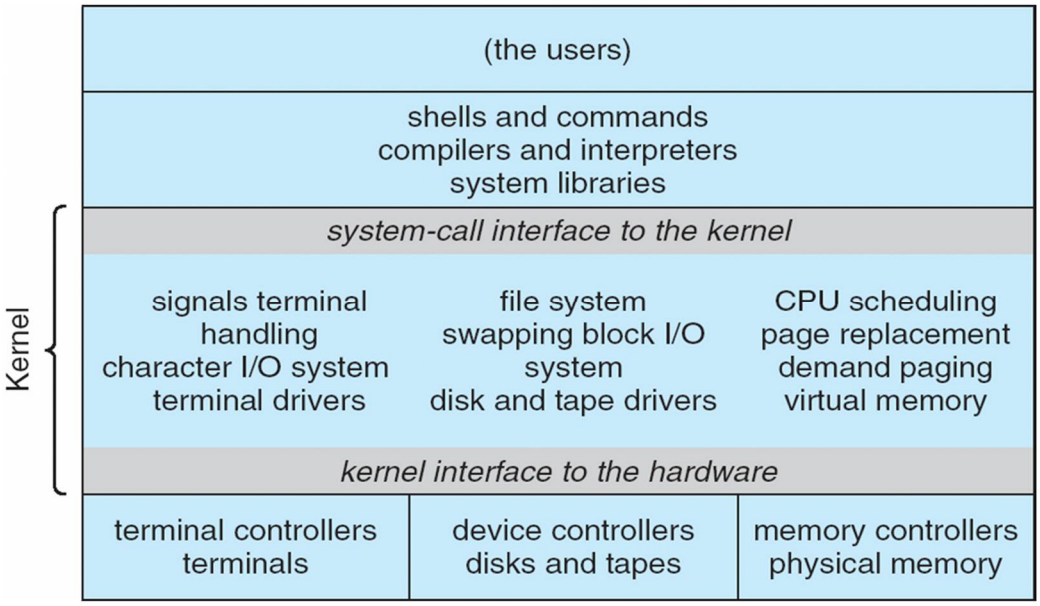


Figure 2.12 Traditional UNIX system structure.

Layered Approach

❖ The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

❖ With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

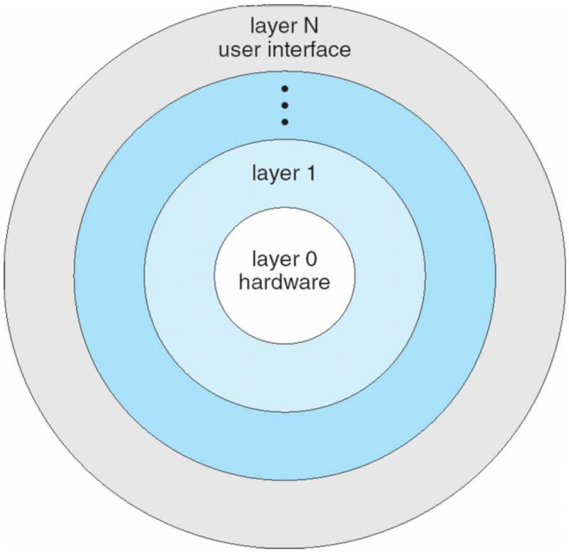


Figure 2.13 A layered operating system.

Microkernel System Structure

- Moves as much from the kernel into user space
- Mach example of microkernel
  - ✓ Mac OS X kernel (Darwin) partly based on Mach
- Communication takes place between user modules using message passing

- Benefits:
  - ✓ Easier to extend a microkernel
  - ✓ Easier to port the operating system to new architectures
  - ✓ More reliable (less code is running in kernel mode)
  - ✓ More secure

- Detriments:
  - ✓ Performance overhead of user space to kernel space communication

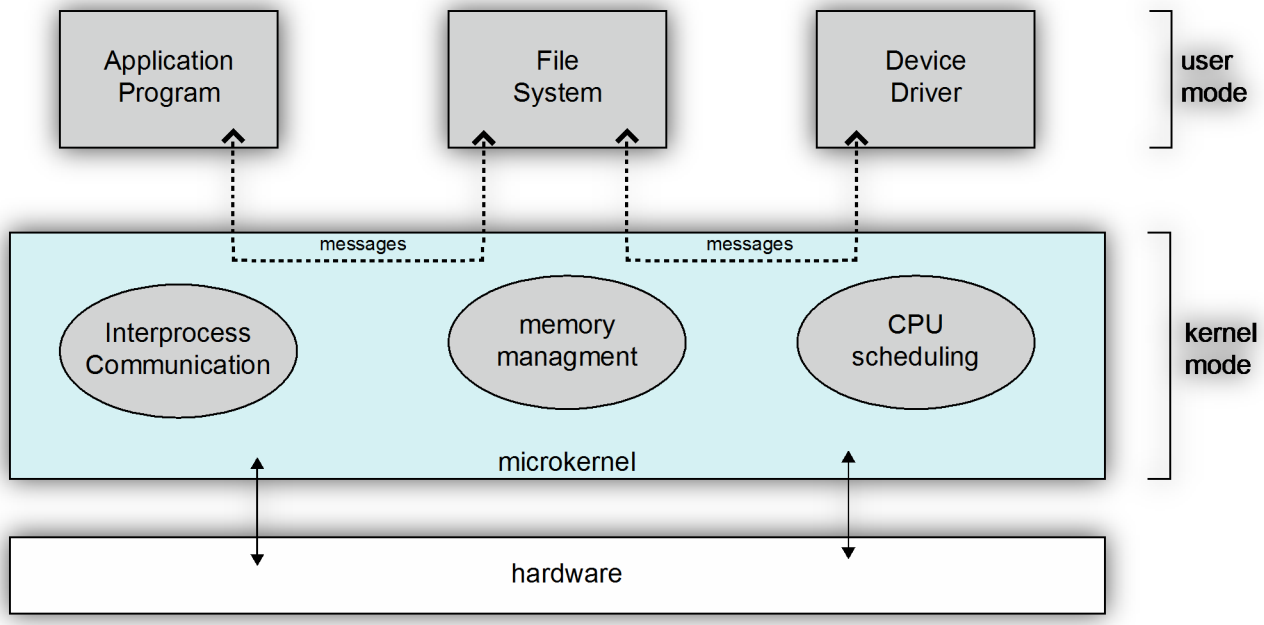


Figure 2.14 Architecture of a typical microkernel.