



Chapter 1. Introduction

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018).
Operating System Concepts (10th ed.). Wiley, pp. 3-54.



Contents

- 1. Introduction to Operating Systems
- 2. Computer Organization
- 3. Operating System Operations
- 4. Operating-System Services
- 5. User and Operating-System Interface
- 6. System Calls
- 7. Operating System Structure
- 8. System Boot

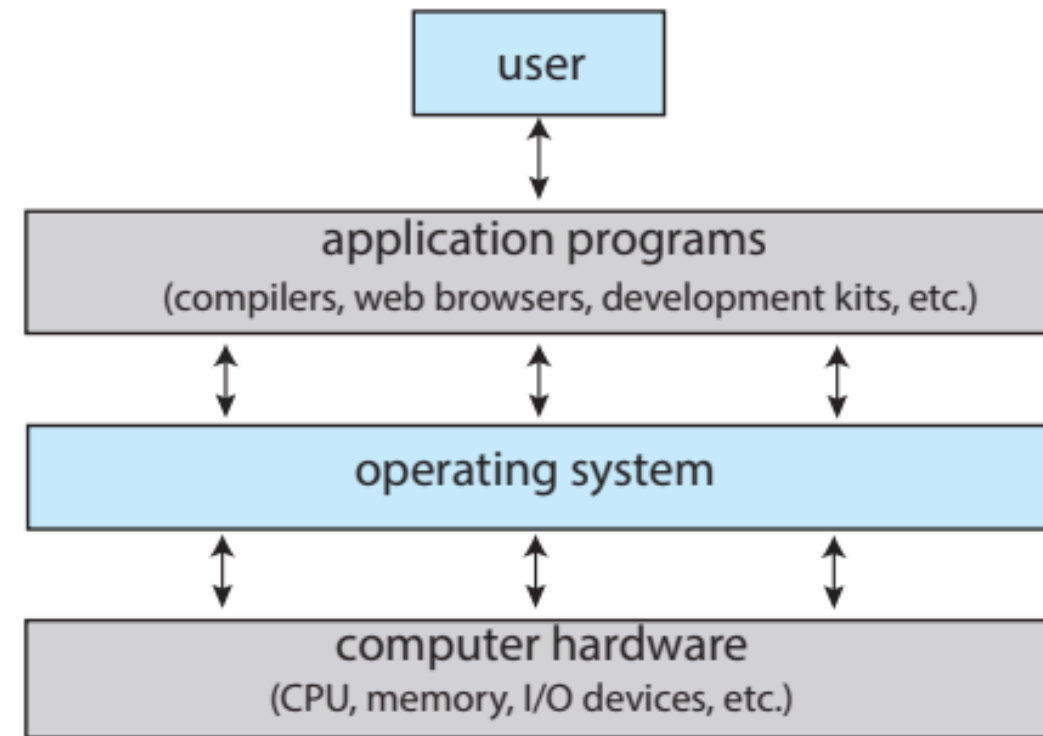
1. Introduction to Operating Systems

- An operating system is software that manages a computer's hardware.
- It provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.
- Operating systems are everywhere
 - From cars and home appliances, to smart phones, personal computers, enterprise computers, and cloud computing environments



What Operating System Do

- A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and a *user*.
 - Hardware: Provides the basic computing resources
 - The central processing unit (CPU), the memory, and the input/output (I/O) devices, for the system.
 - Operating system: Controls and coordinates use of hardware among various applications and users
 - Application programs: Define the ways in which these resources are used to solve users' computing problems.
 - Word processors, spreadsheets, compilers, and web browsers, ...
 - Users
 - People, machines, other computers



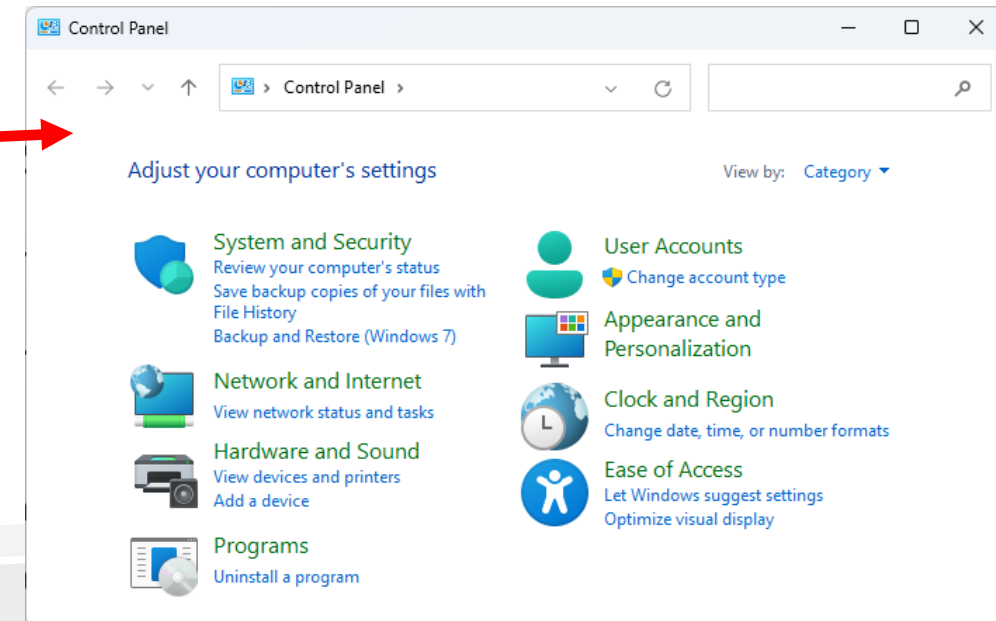
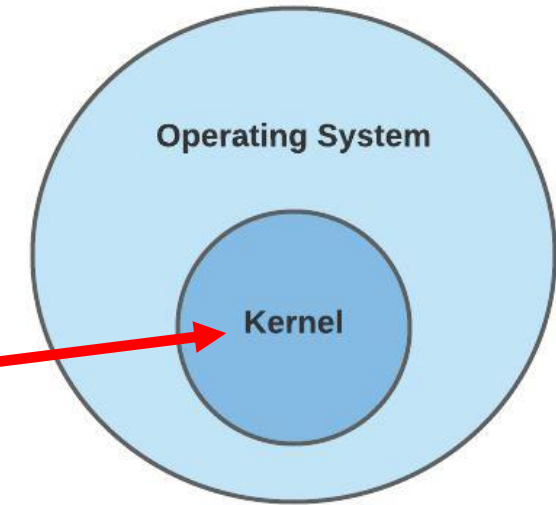


Definition of Operating Systems

- Operating system goals
 - Execute user programs and make solving user problems easier
 - Make the computer system convenient to use
 - Use the computer hardware/computing resources in an efficient manner
- Operating system definition
 - No universally accepted definition
 - Operating system is a **resource allocator** and **control program** making efficient use of hardware and managing execution of user programs.

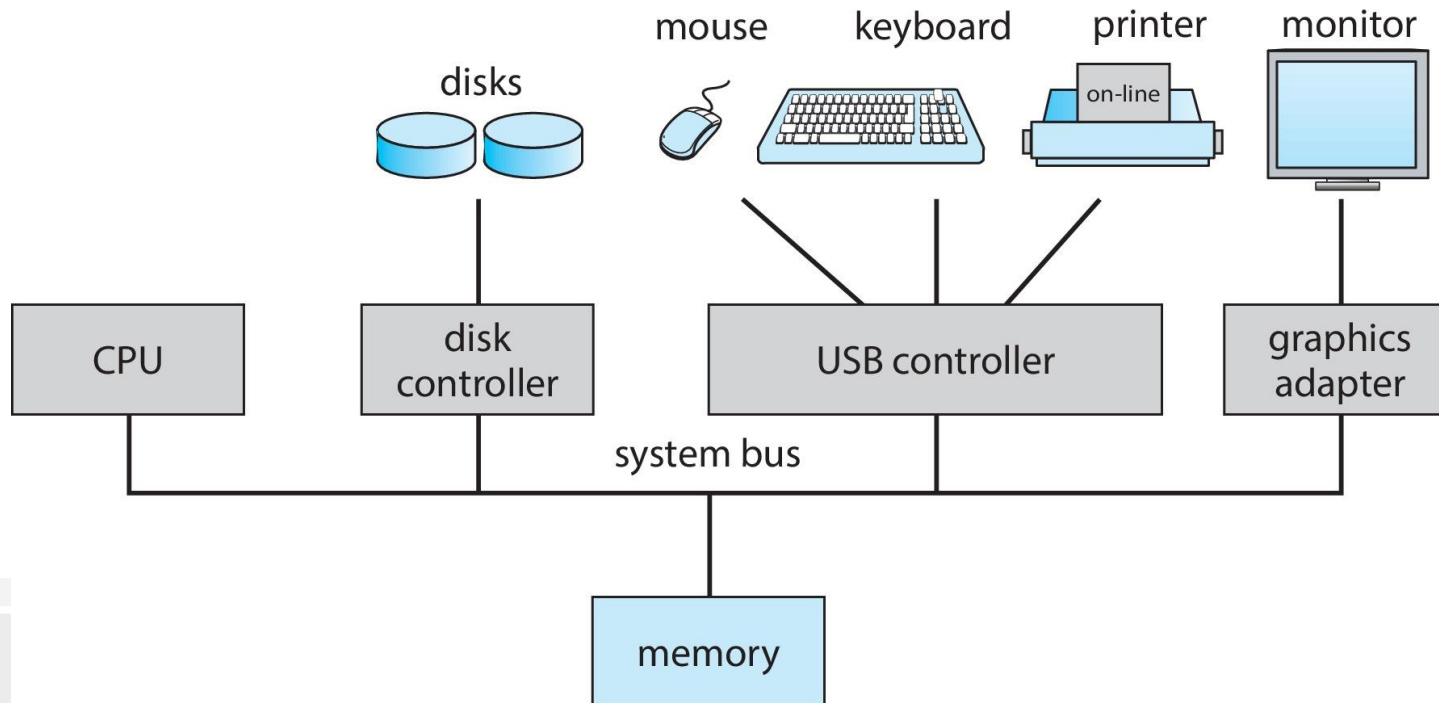
Definition of Operating Systems

- The operating system is the one program running at all times on the computer—usually called the **kernel**.
- Along with the kernel, there are two other types of programs:
 - System programs
 - Are associated with the operating system but are not necessarily part of the kernel
 - Application programs
 - Include all programs not associated with the operation of the system.



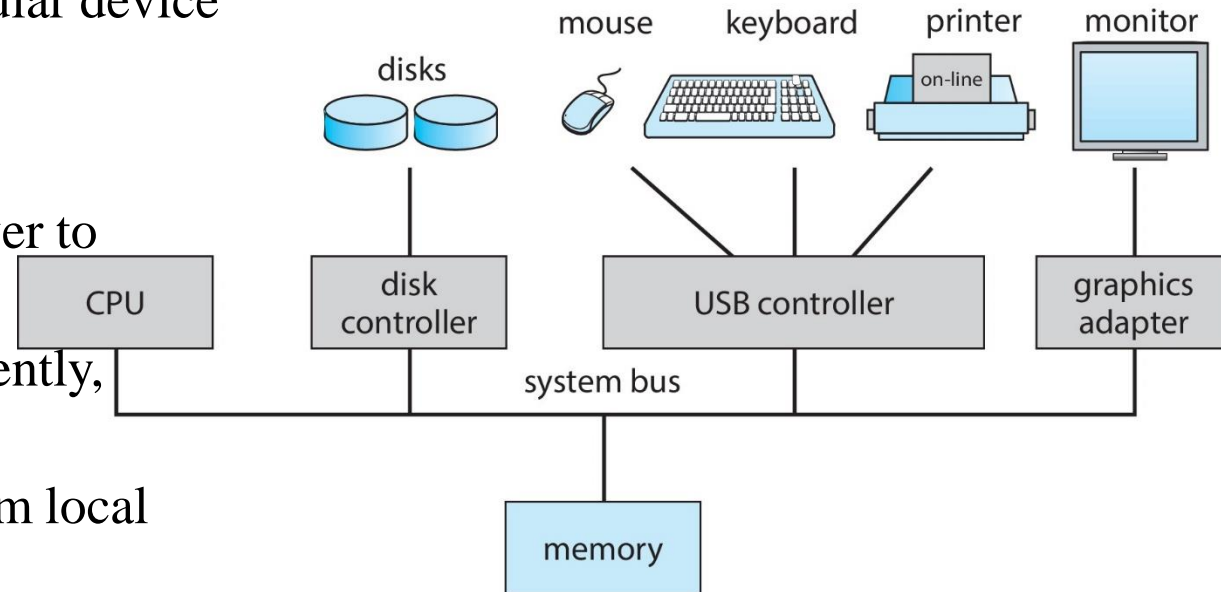
2. Computer-System Organization

- A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access between components and shared memory.

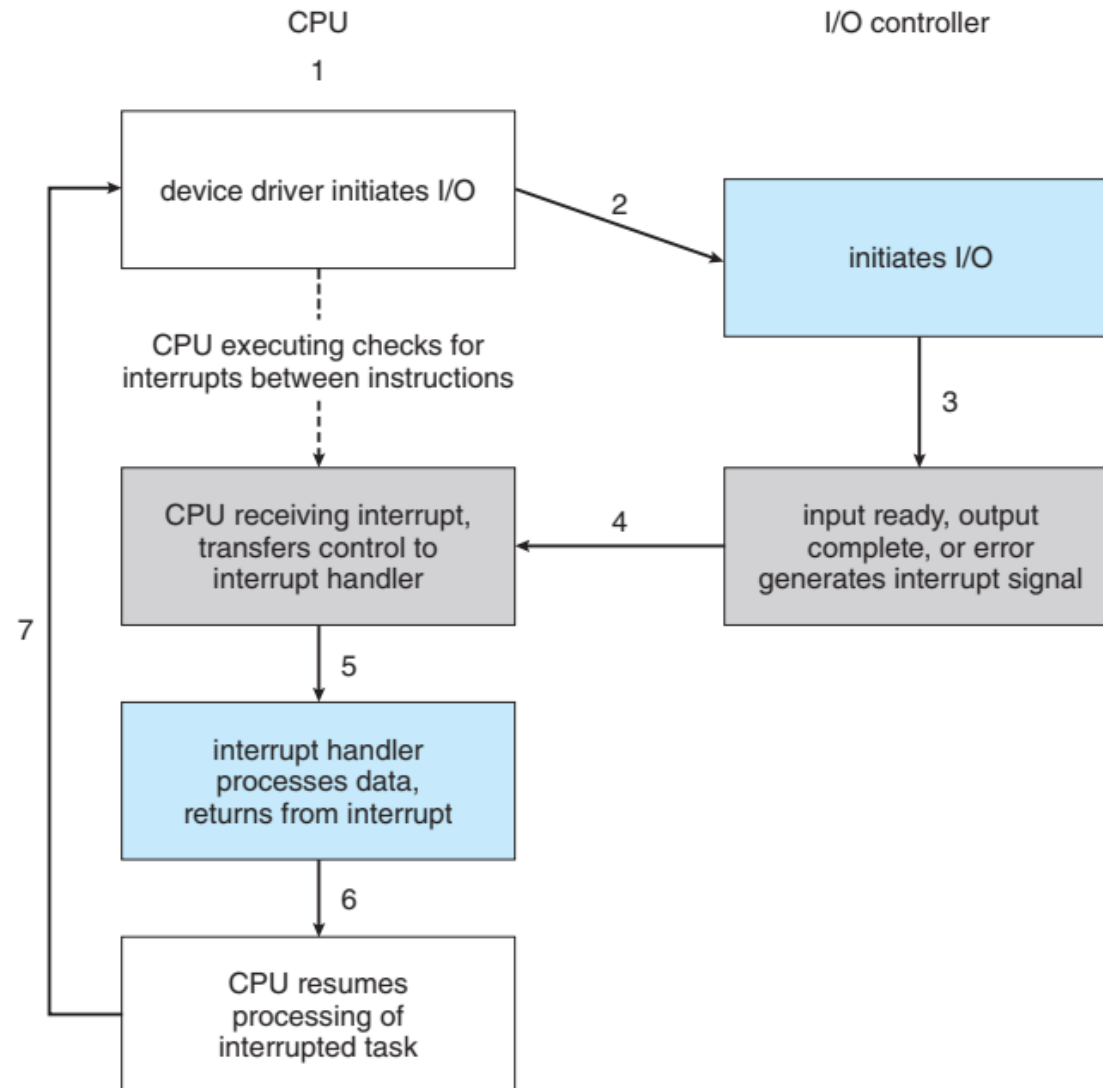


2. Computer-System Organization

- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- Each device controller type has an device driver to manage it.
- I/O devices and the CPU can execute concurrently, competing for memory cycles.
- CPU moves data from/to main memory to/from local buffers.
- I/O is from the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an **interrupt**.
- Operating Systems are interrupt driven.



Interrupt-Driven I/O Cycle





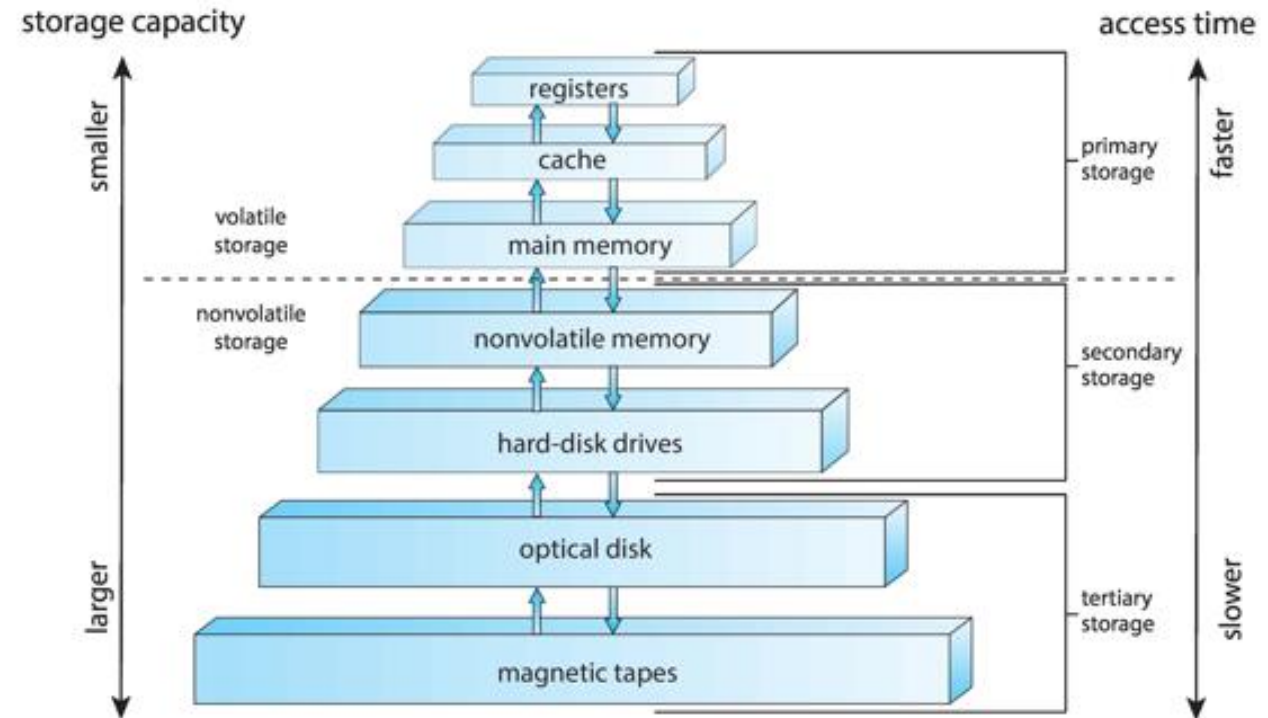
Interrupts

- Most CPUs have two interrupt request lines.
 - One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.
 - The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
 - The maskable interrupt is used by device controllers to request service.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Storage Structures

- Main memory – only large storage media that the CPU can access directly
 - Random access
 - Typically volatile
 - Typically random-access memory in the form of Dynamic Random-access Memory (DRAM)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
 - Hard Disk Drives (HDD)
 - Non Volatile Memory (NVM)
- Tertiary storage is used to store backup copies of material stored on other devices





Computer System Architecture

- CPU - The hardware that executes instructions.
- Processor - A physical chip that contains one or more CPUs.
- Core - The basic computation unit of the CPU.
- Multicore - Including multiple computing cores on the same CPU.
- Multiprocessor - Including multiple processors

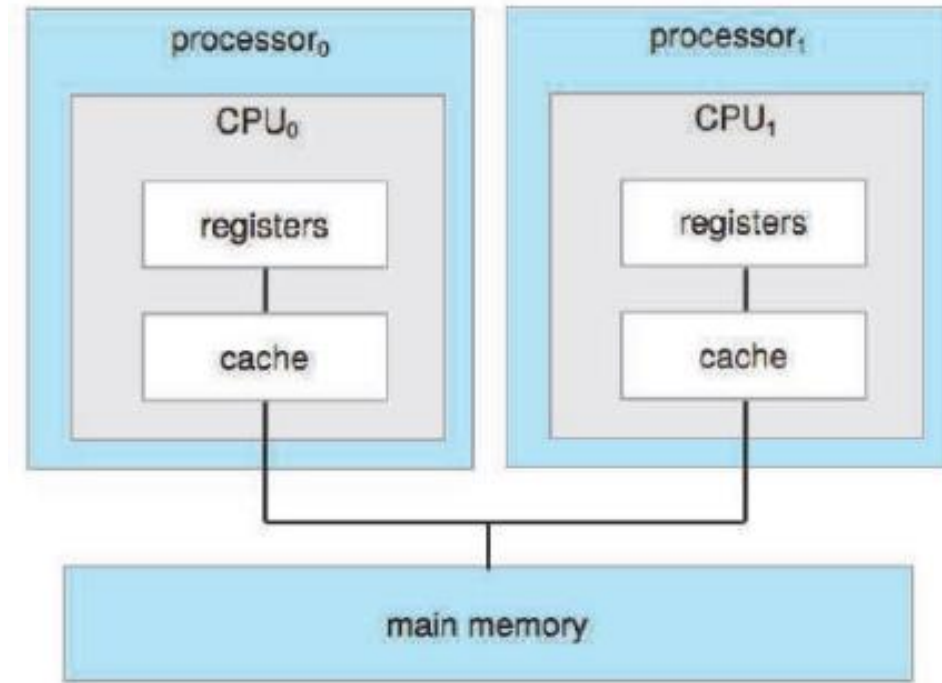


Single-Processor Systems

- Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and registers for storing data locally.
- The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These systems have other special-purpose processors as well.

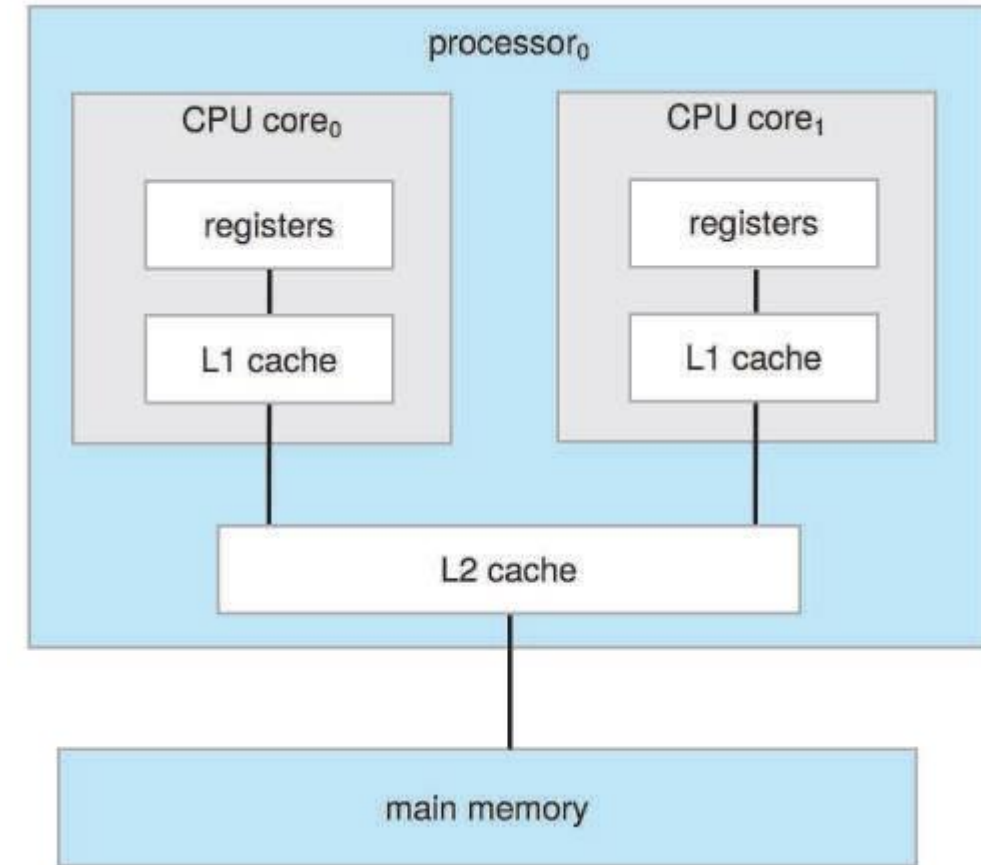
Symmetric Multiprocessing Architecture

- The most common multiprocessor systems use symmetric multiprocessing (SMP), in which each peer CPU processor performs all tasks.
- Each CPU processor has its own set of registers, as well as a private - or local - cache. However, all processors share physical memory over the system bus.
- The benefit of this model is that many processes can run simultaneously - N processes can run if there are N CPUs - without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies.



A Dual-Core Design

- Each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache.
- A level 2 (L2) cache is local to the chip but is shared by the two processing cores.
- Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared caches.
- A multicore processor with N cores appears to the operating system as N standard CPUs.
- All modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.





Multiprocessor Systems

- Most modern computers are **multiprocessor systems**.
- The definition of multiprocessor has evolved over time and now includes **multicore systems**, in which multiple computing cores reside on a single chip.
- Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.



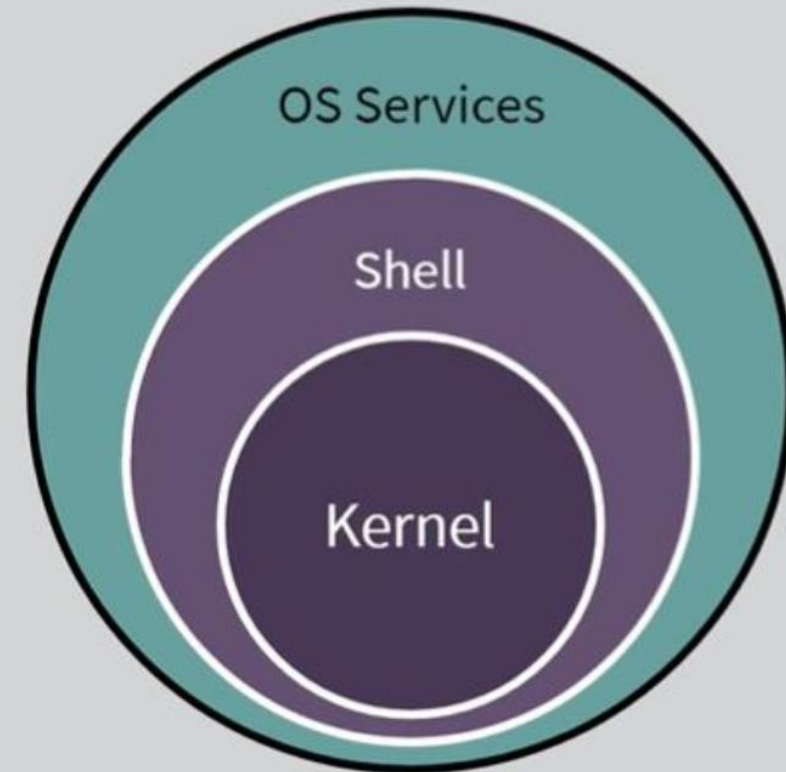
Resource Management

- An operating system is a resource manager.
 - Process management
 - Memory management
 - File-system management
 - Mass-storage management
 - Cache management
 - I/O system management

3. Operating System Operations

- The **kernel** is the core component of the operating system
 - Kernel is running at all time in the computer
- A **shell** is a program that acts as command line interpreter.
 - It processes commands and outputs the results. It interprets and processes the commands entered by the user.
 - In most Linux and Mac operating systems the default shell is Bash. While on Windows it's Powershell. Some other common examples of shells are Zsh and Fish.
- An operating system provides **an environment** for the execution of programs.
 - It makes certain **services** available to programs and to the users of those programs.

- The core component of the OS



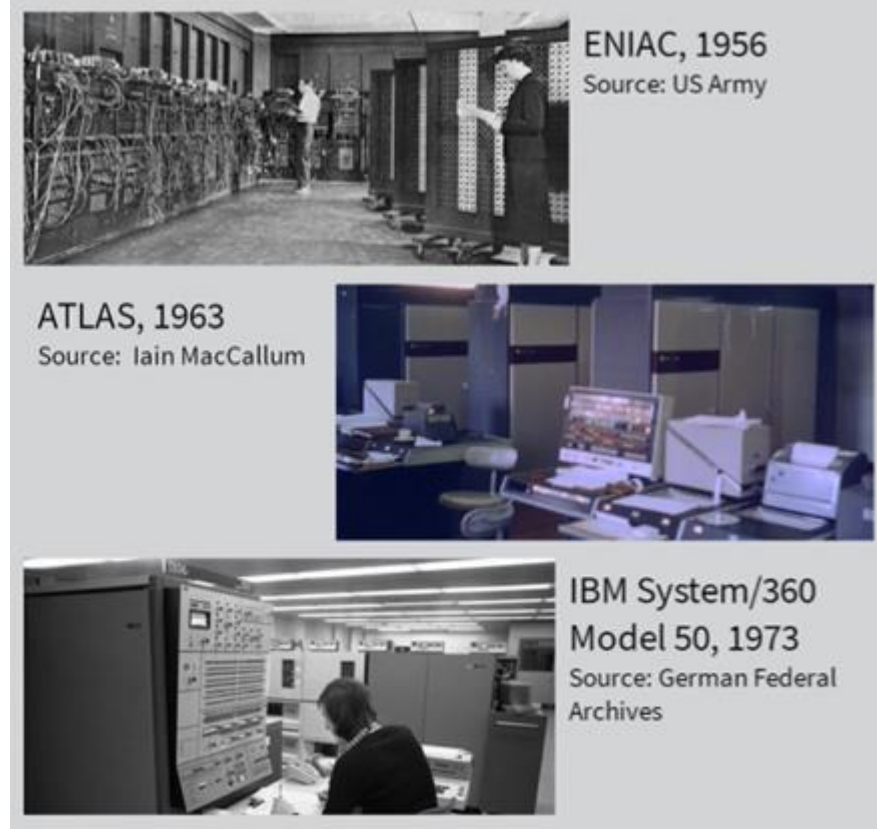


3. Operating System Operations

- Most modern OS have some common features:
 - **Multiprogramming**
 - **Time-Sharing (Multitasking)**
- Back in the old days, OS followed batch programming.

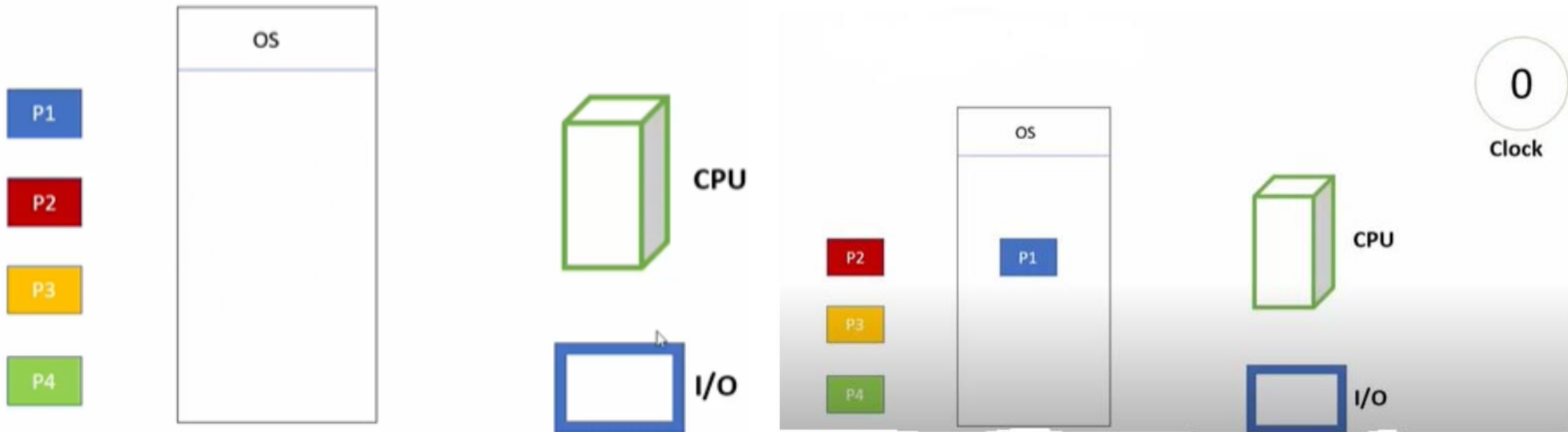
Batch Programming

- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.



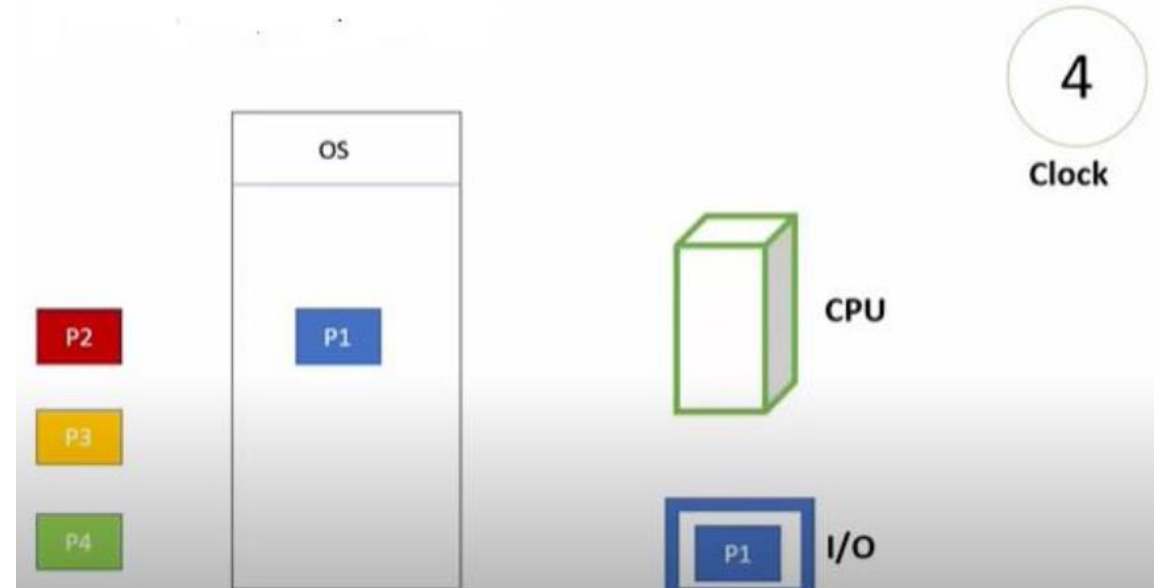
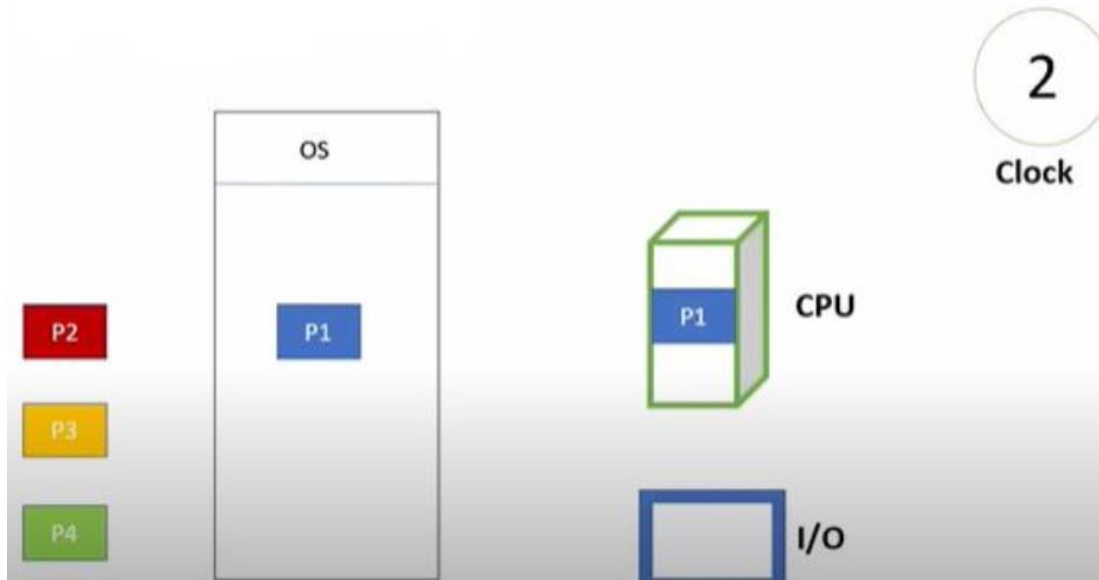


Batch Operating System





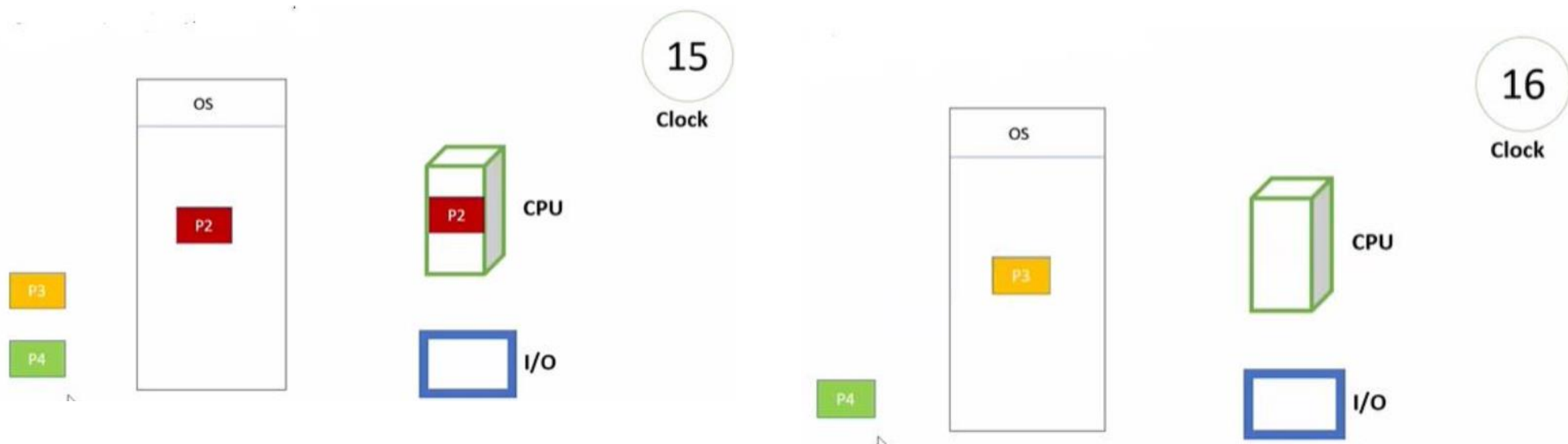
Batch Operating System



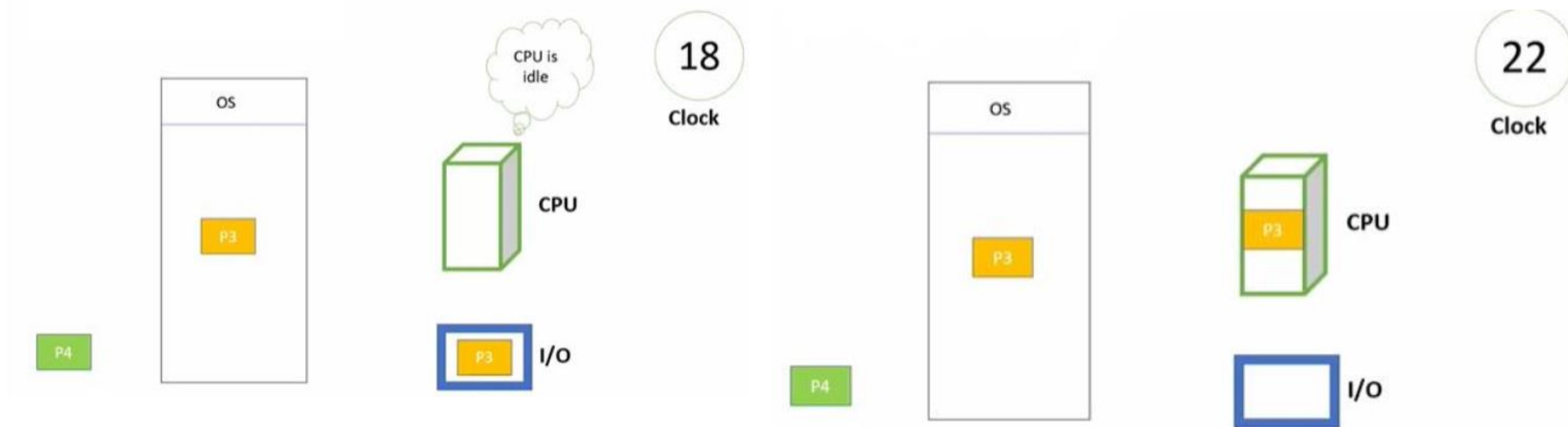
Batch Operating System



Batch Operating System

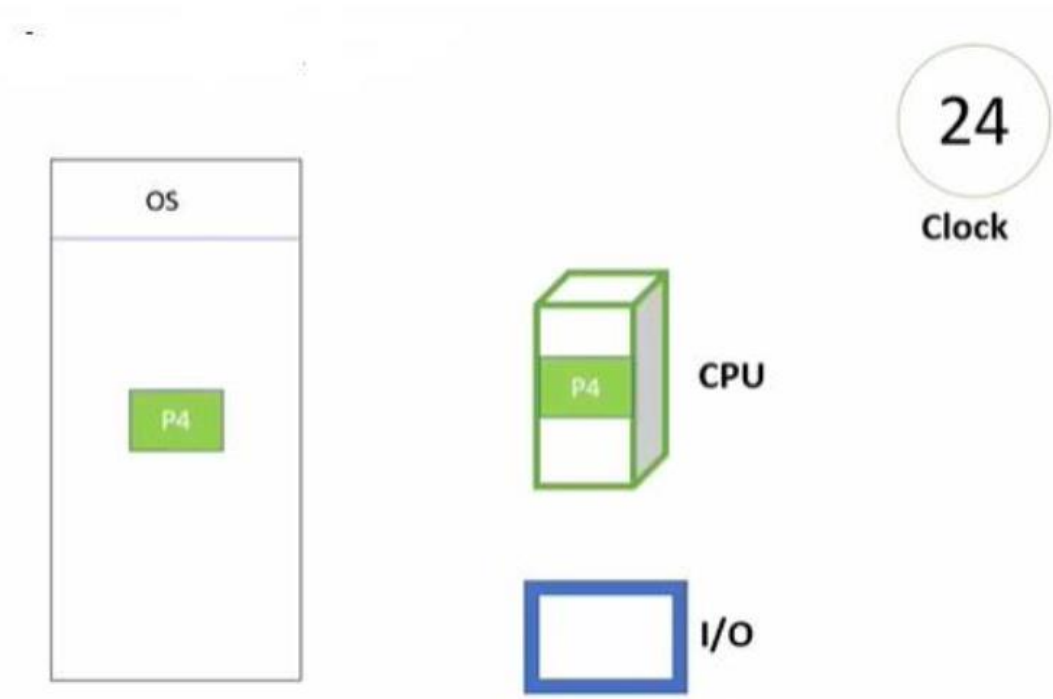


Batch Operating System





Batch Operating System



- Disadvantages
 - Zero interaction between user and computer
 - No mechanism to prioritize processes
 - Poor CPU usage (CPU is idles, waits for I/O completion)
- Very low in nature

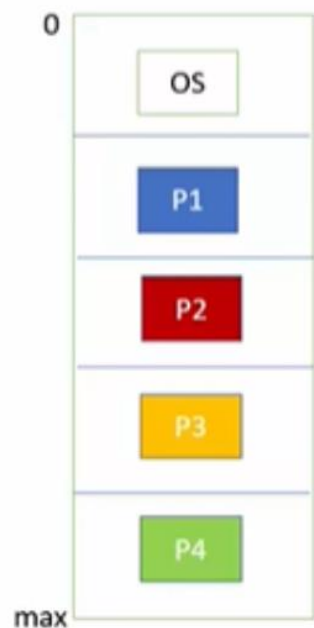
Modern Operating System Structure

- Multiprogramming
 - Multiprogramming represent the capability of running multiple programs by CPU
 - Modern computer system have lot's of processing power and lot's of resources enough to run multiple programs at the same time.



Multiprogramming

Multi-Programming

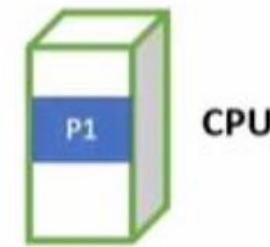
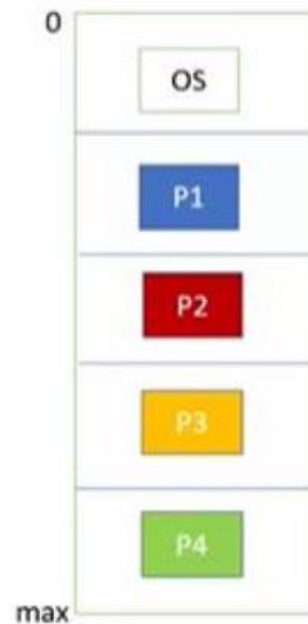


CPU



I/O

Multi-Programming



CPU

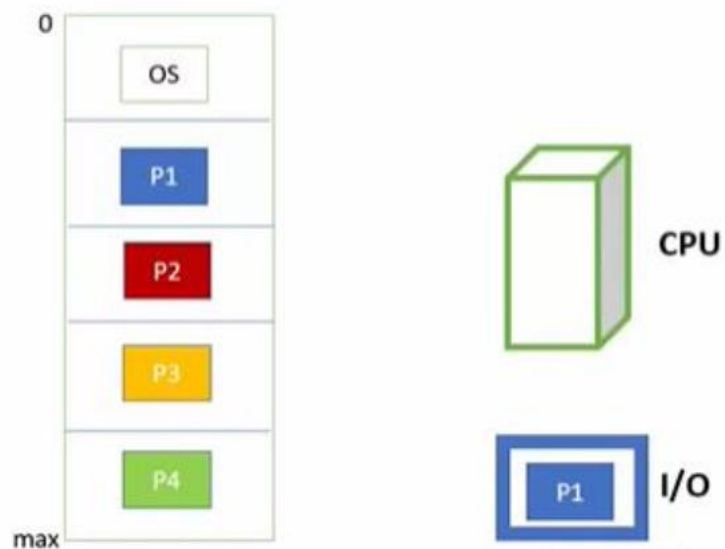


I/O

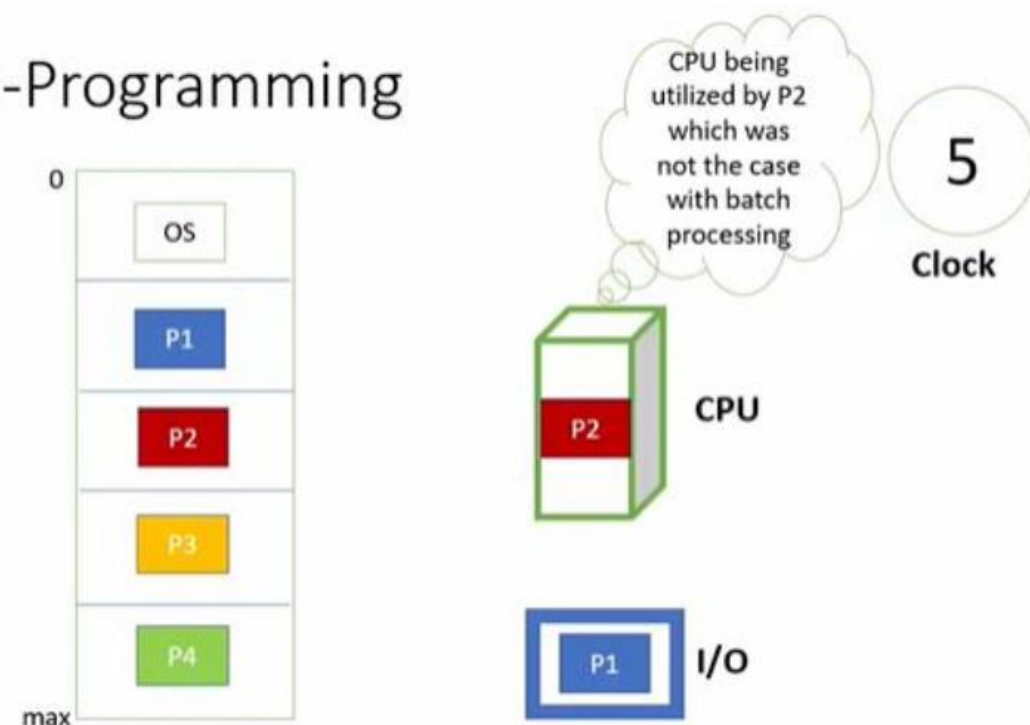
0
Clock

Multiprogramming

Multi-Programming

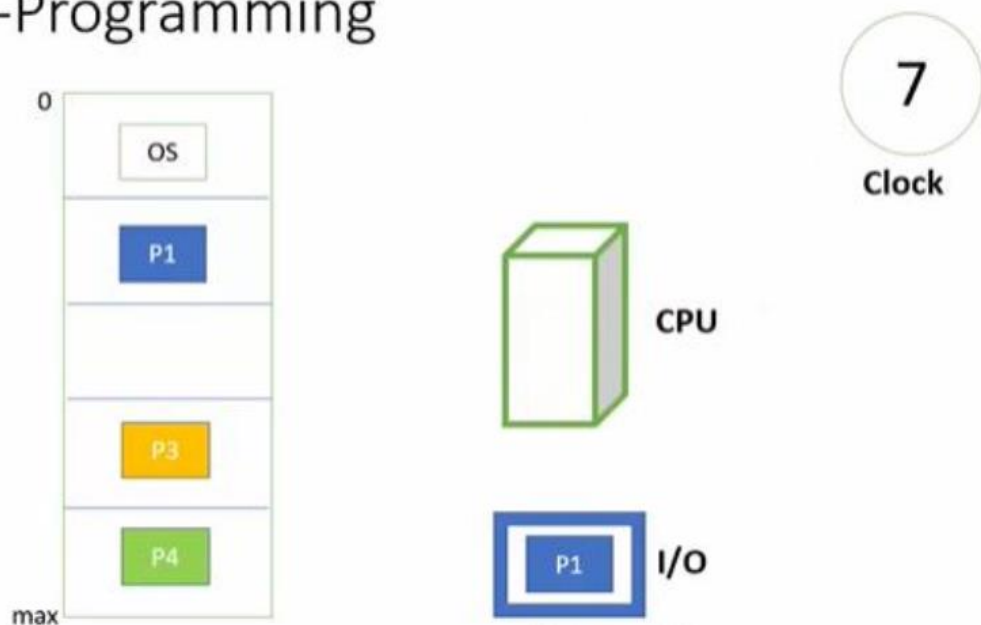


Multi-Programming

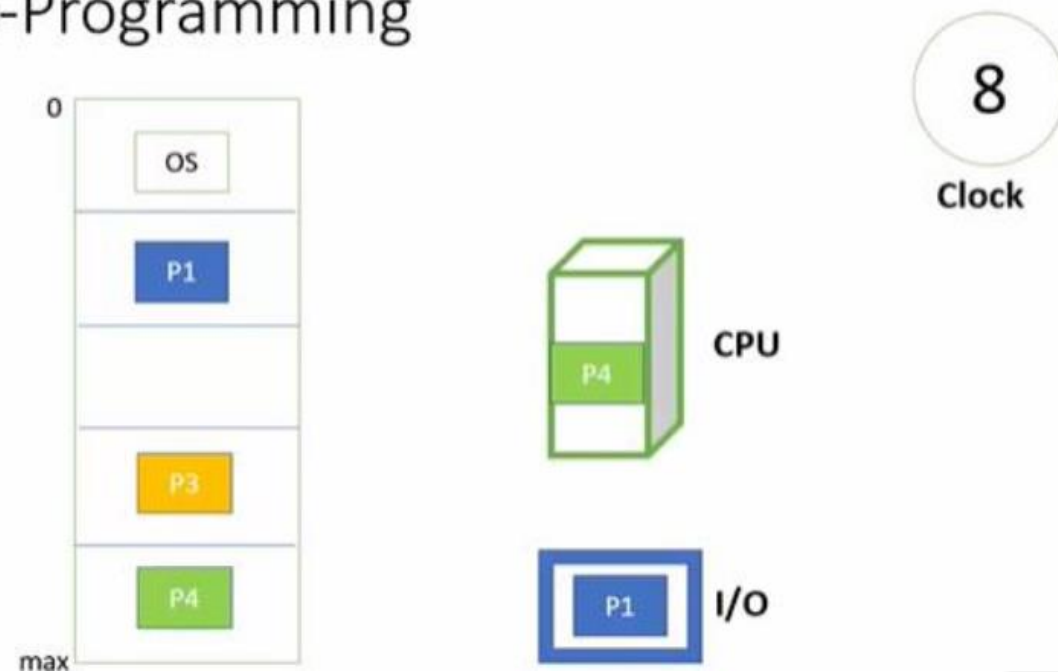


Multiprogramming

Multi-Programming

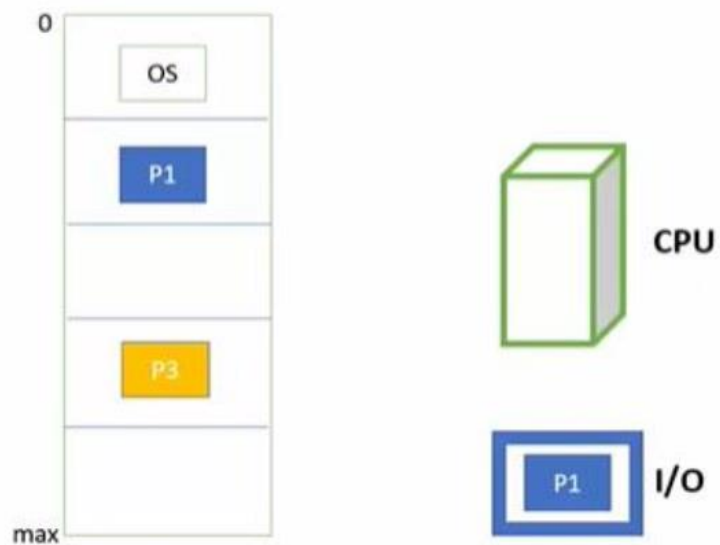


Multi-Programming



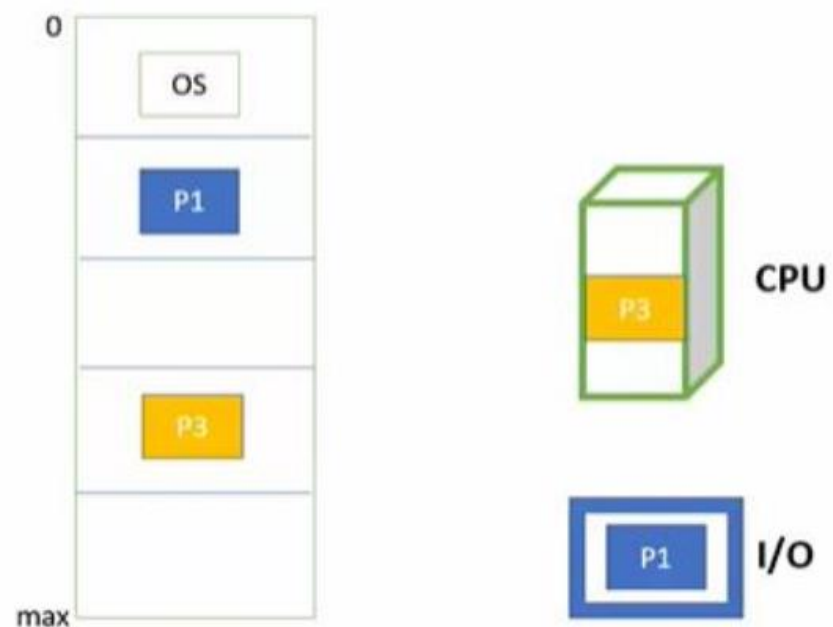
Multiprogramming

Multi-Programming



9
Clock

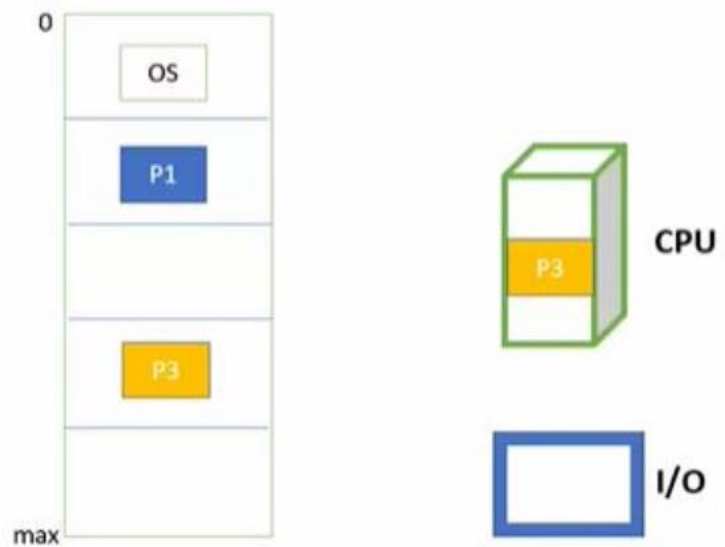
Multi-Programming



10
Clock

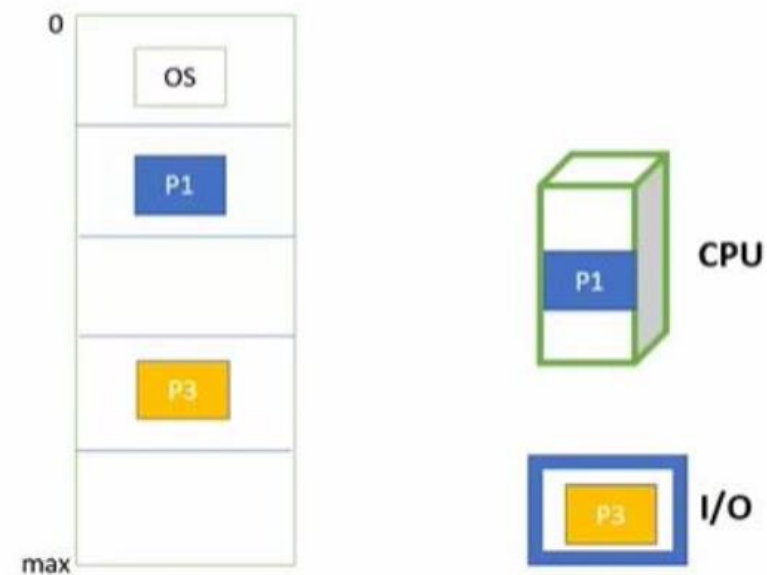
Multiprogramming

Multi-Programming



10
Clock

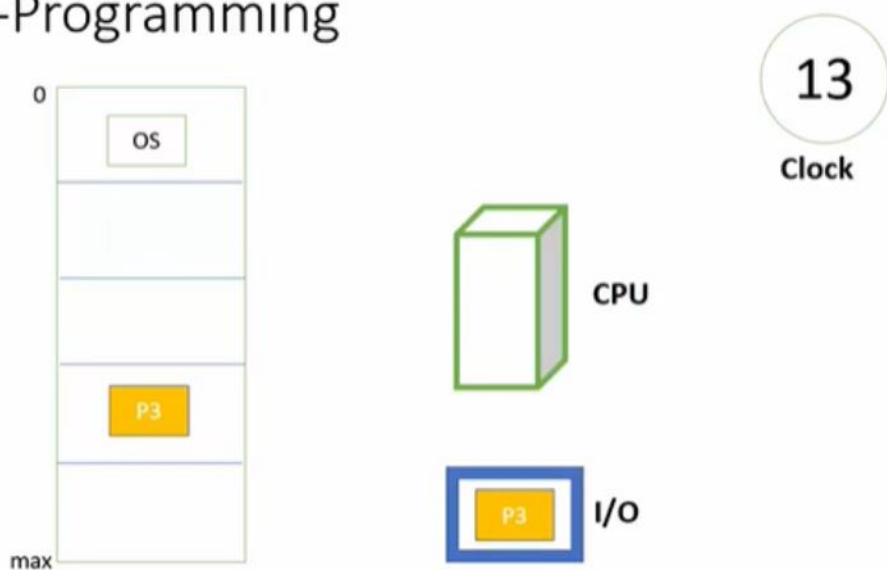
Multi-Programming



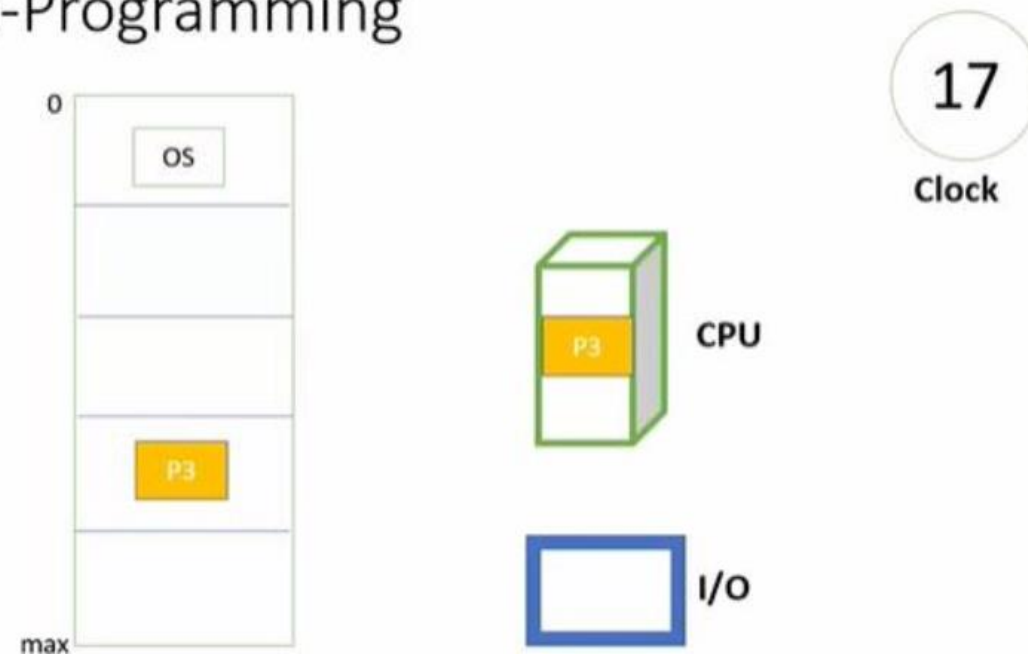
11
Clock

Multiprogramming

Multi-Programming

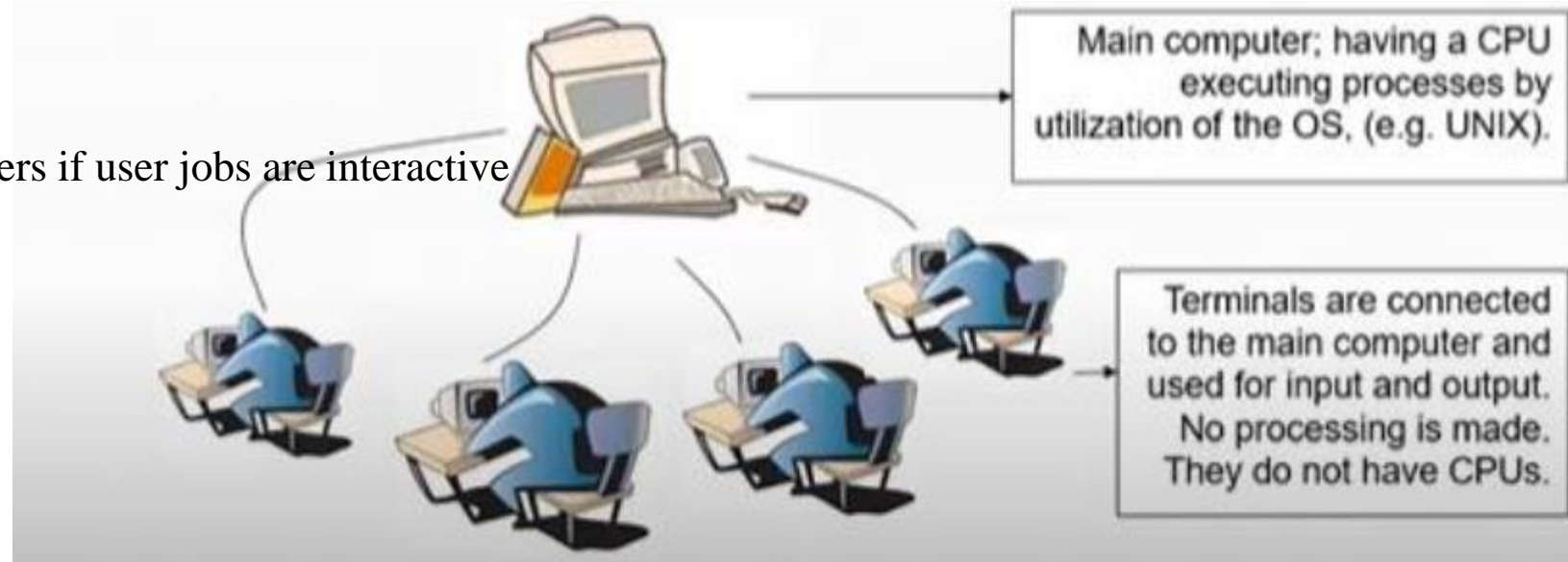


Multi-Programming

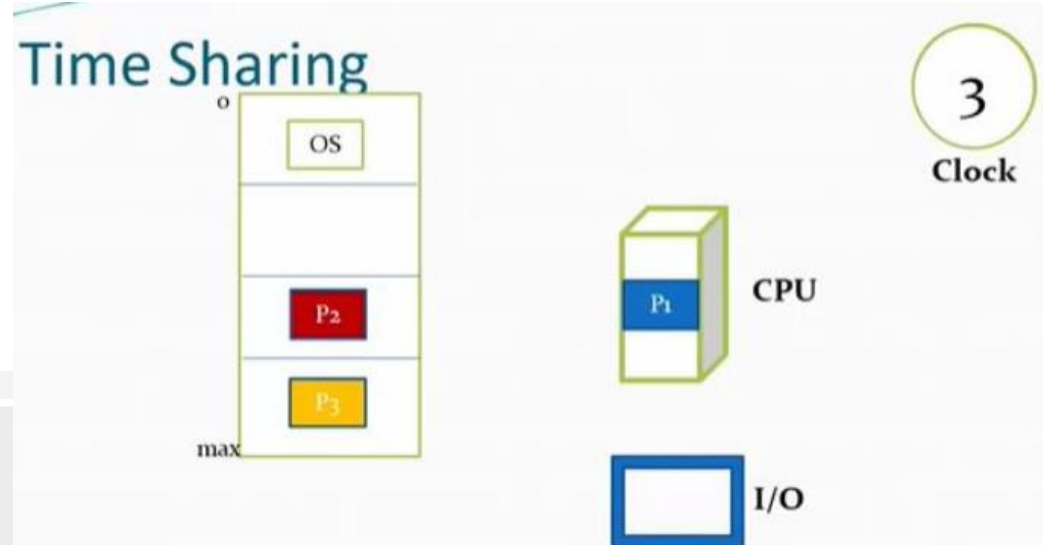
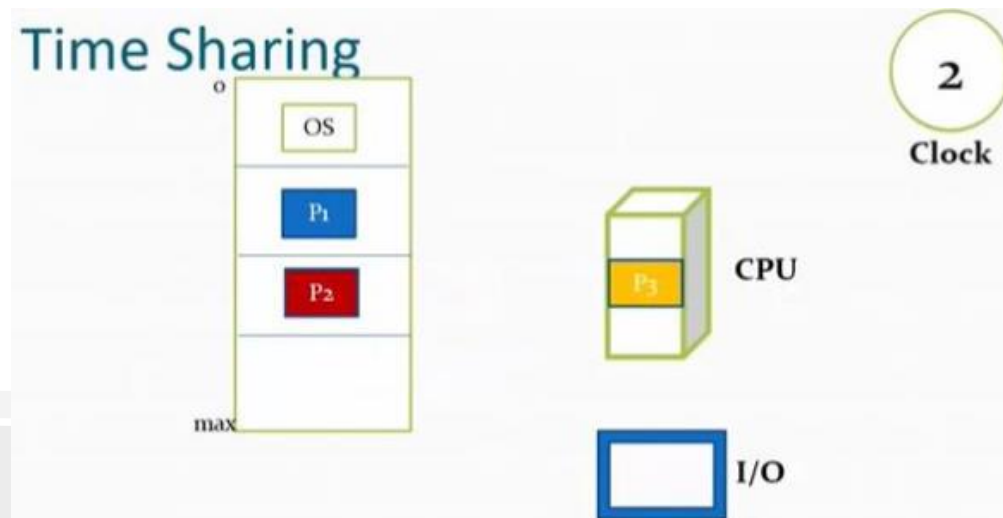
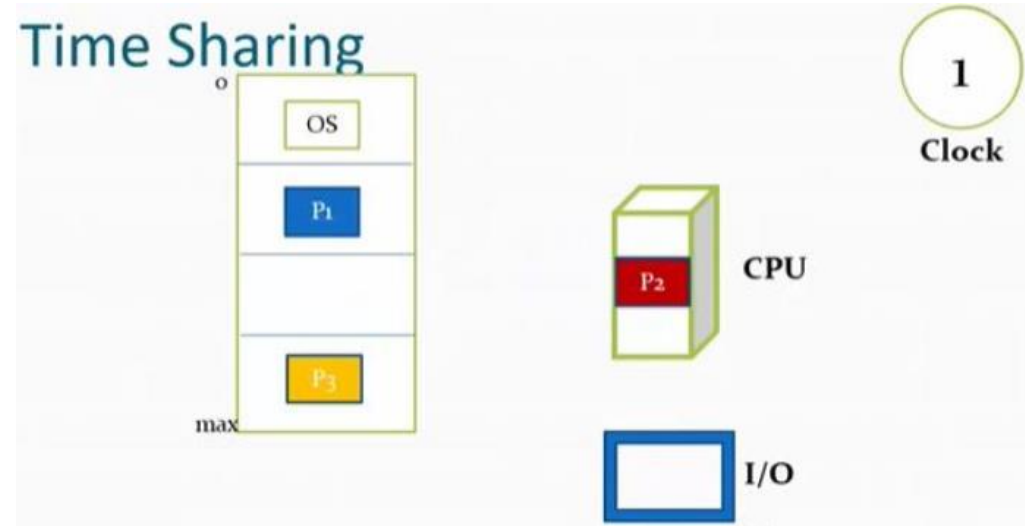
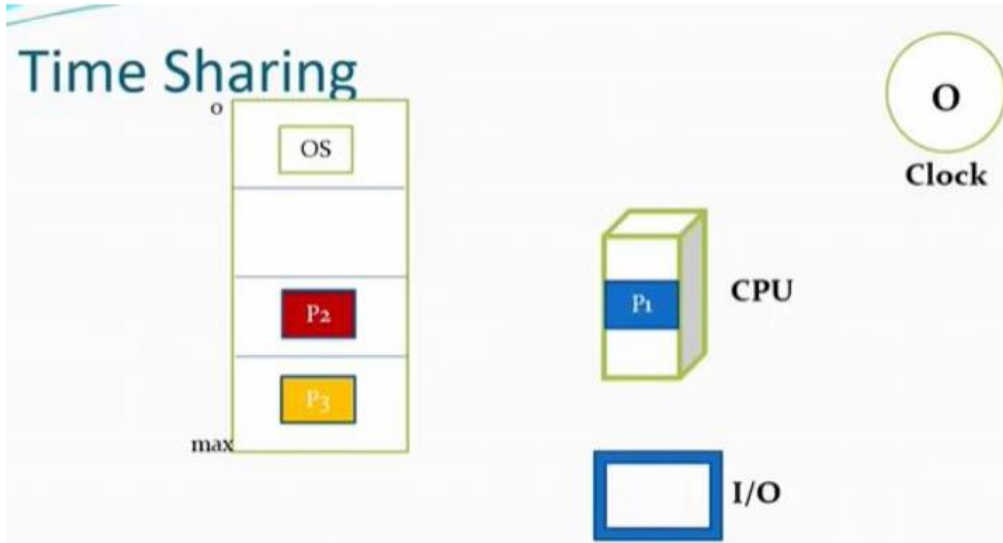


Time-Sharing System

- Supports interactive use
 - Multiple terminals into one machine
 - Each user has illusion of entire machine to himself/herself
 - Optimized response time
- Time slicing
 - Divide CPU equally among users
 - CPU can move among programs and users if user jobs are interactive



Time-Sharing System

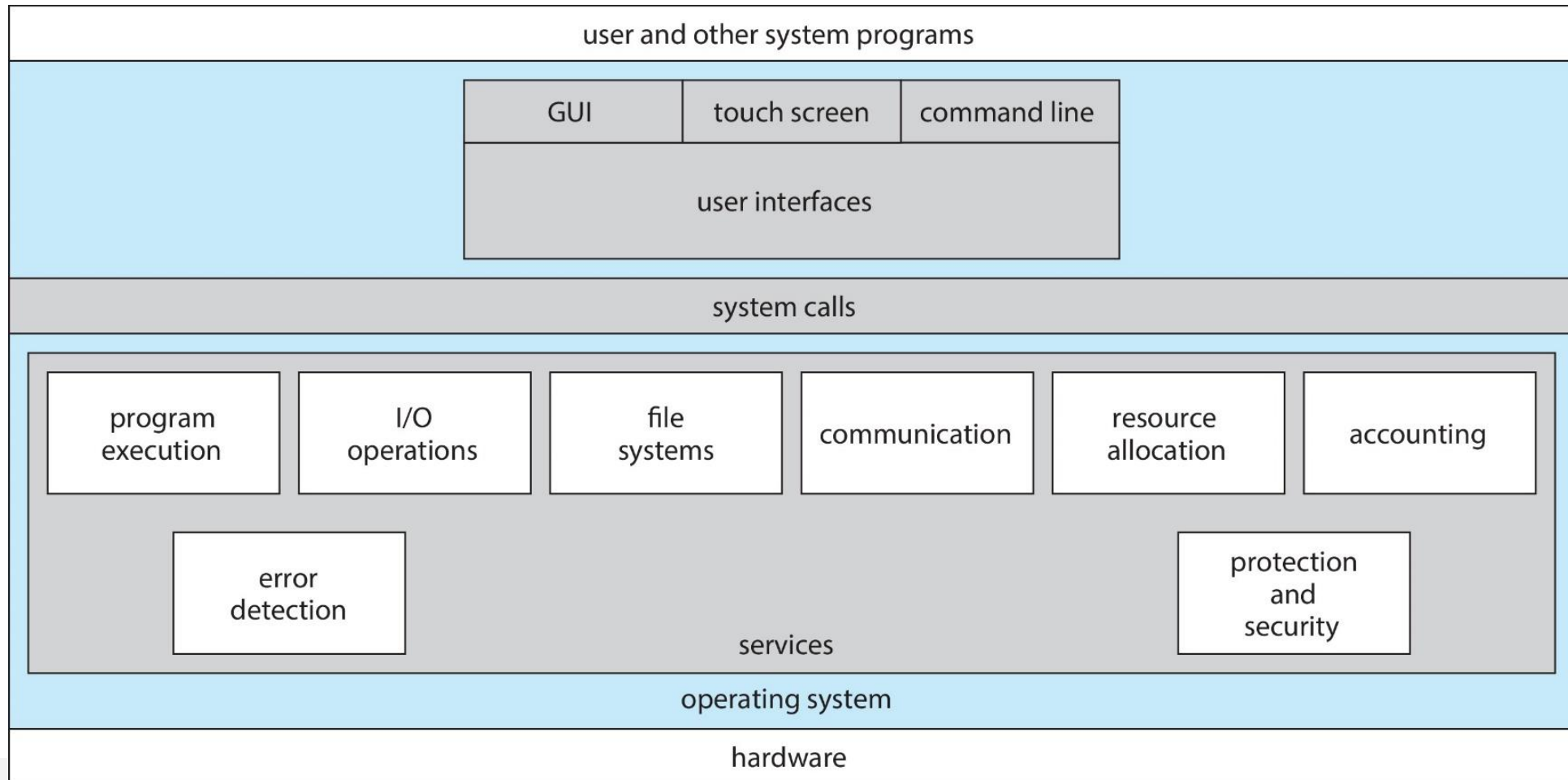




4. Operating-System Services

- We can view an operating system from several vantage points.
 - On the services that the system provides
 - On the interface that it makes available to users and programmers
 - On its components and their interconnections

4. Operating-System Services





4. 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), touch-screen, 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.



4. Operating-System Services

- **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



4. Operating-System Services

- **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.
- **Logging** - 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

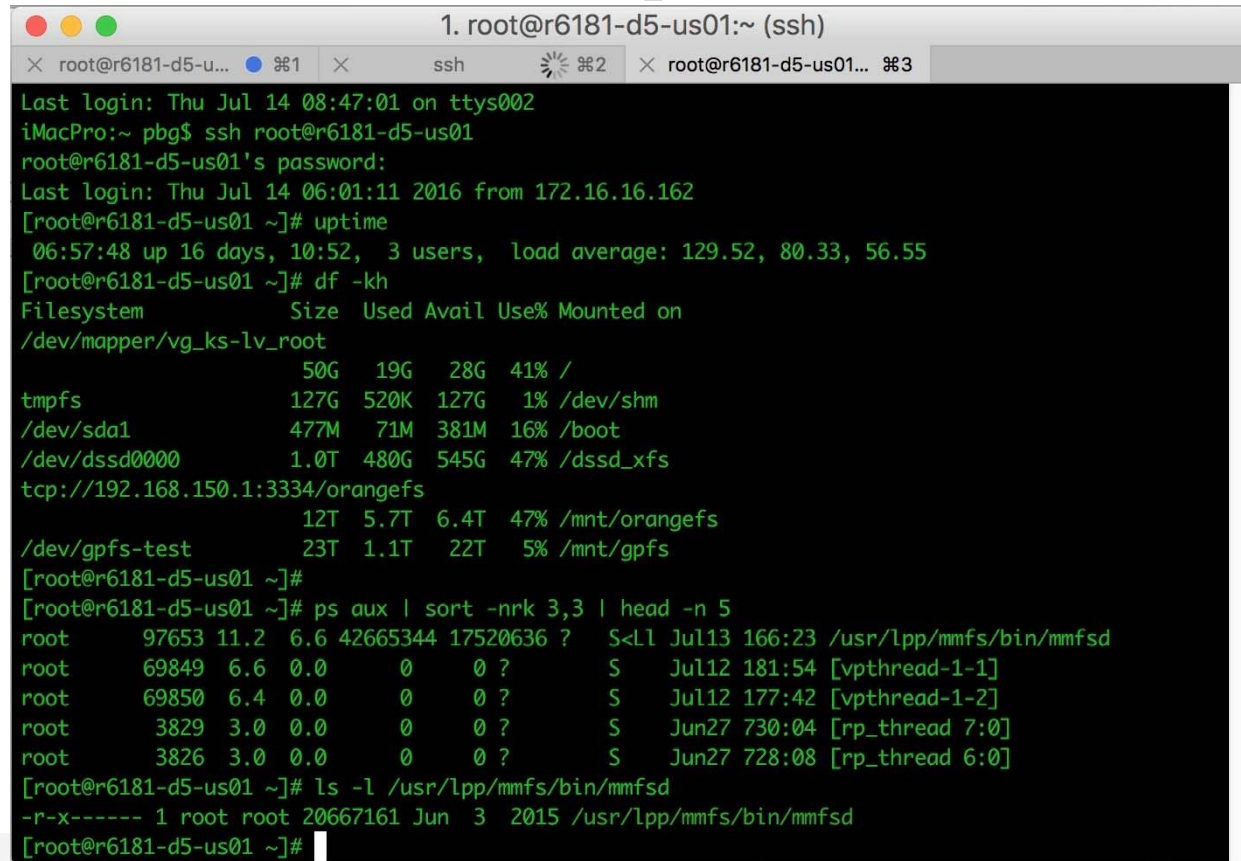


5. User and Operating-System Interface

- Command Line Interpreter (CLI)
 - CLI allows direct command entry
 - Sometimes implemented in kernel, sometimes by systems program
 - Sometimes multiple flavors implemented – **shells**
 - Primarily fetches a command from user and executes it
 - Sometimes commands built-in, sometimes just names of programs
 - If the latter, adding new features doesn't require shell modification

5. User and Operating-System Interface

- Bourne shell command interpreter



```
1. root@r6181-d5-us01:~ (ssh)
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/mmfsc
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/mmfsc
-r-x----- 1 root root 20667161 Jun  3 2015 /usr/lpp/mmfs/bin/mmfsc
[root@r6181-d5-us01 ~]#
```

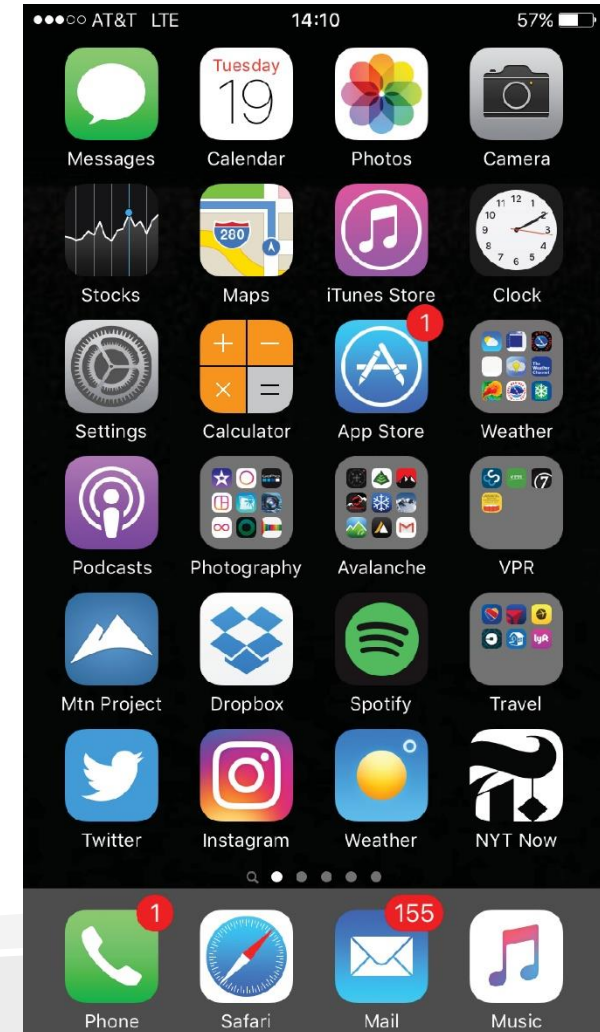


5. User and Operating-System Interface

- Graphical User Interface (GUI)
 - User-friendly **desktop** metaphor interface
 - Usually mouse, keyboard, and monitor
 - Icons represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder))
 - Invented at Xerox PARC
 - Many systems now include both CLI and GUI interfaces
 - Microsoft Windows is GUI with CLI “command” shell
 - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
 - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

5. User and Operating-System Interface

- Touchscreen Interfaces
 - Touchscreen devices require new interfaces
 - Mouse not possible or not desired
 - Actions and selection based on gestures
 - Virtual keyboard for text entry
- Voice commands



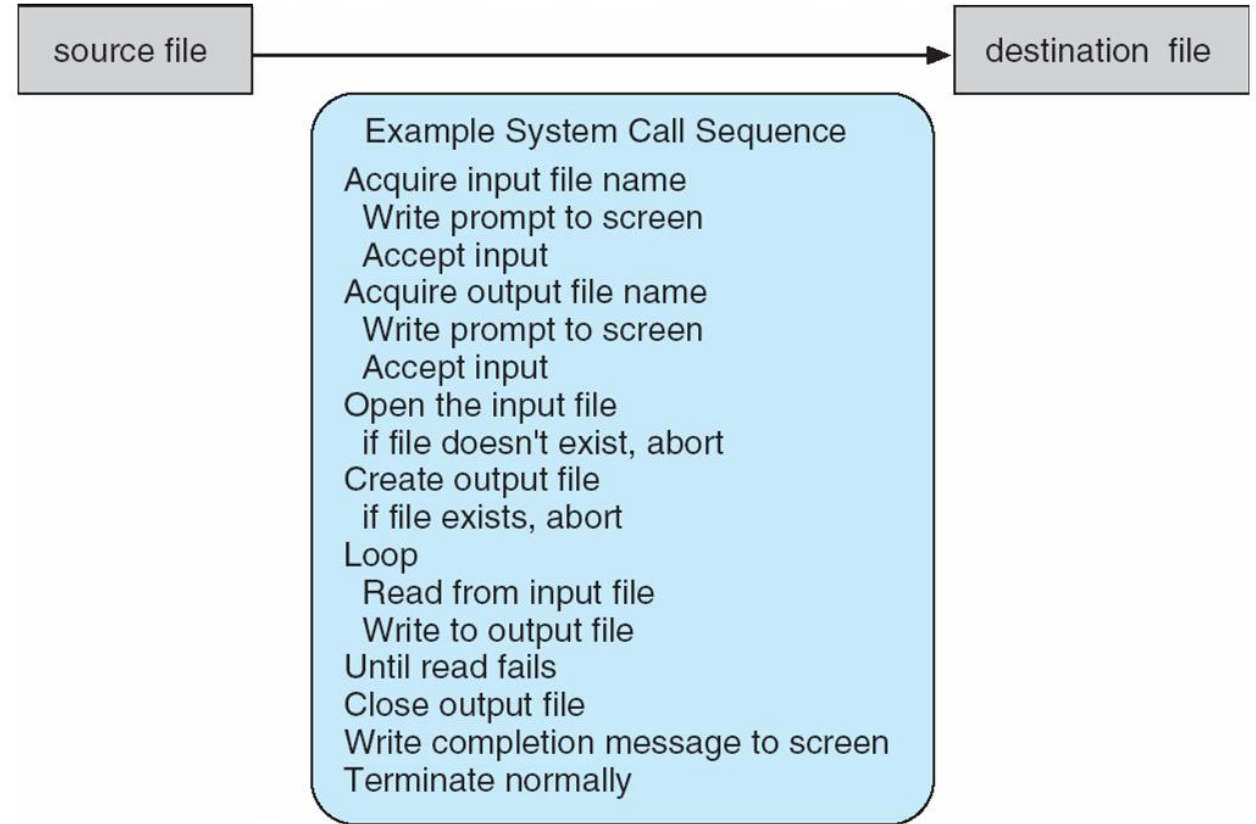


6. System Calls

- Programming interface to the services provided by the OS
 - System call is the programmatic way in which a program requests a service from the kernel of the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs are
 - Win32 API for Windows
 - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
 - Java API for the Java virtual machine (JVM)

6. System Calls

- Example of system calls:
 - System call sequence to copy the contents of one file to another file



6. System Calls

- 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>	
<div></div>	<div></div>	<div></div>
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 into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

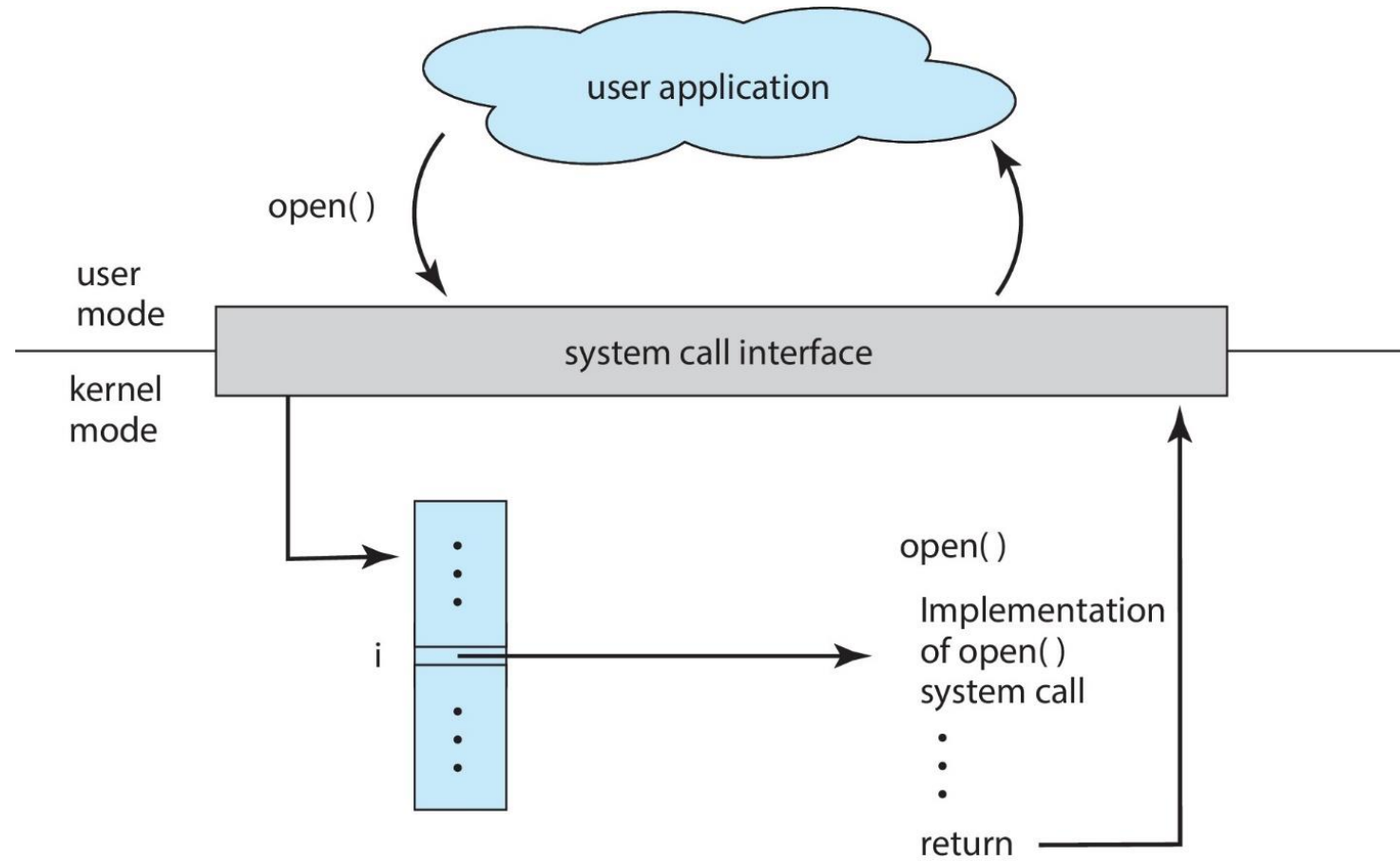
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.



6. System Calls

- System call implementation
 - Typically, a number is associated with each system call
 - **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 status of the system call and any return values
 - The caller needs 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)

6. System Calls





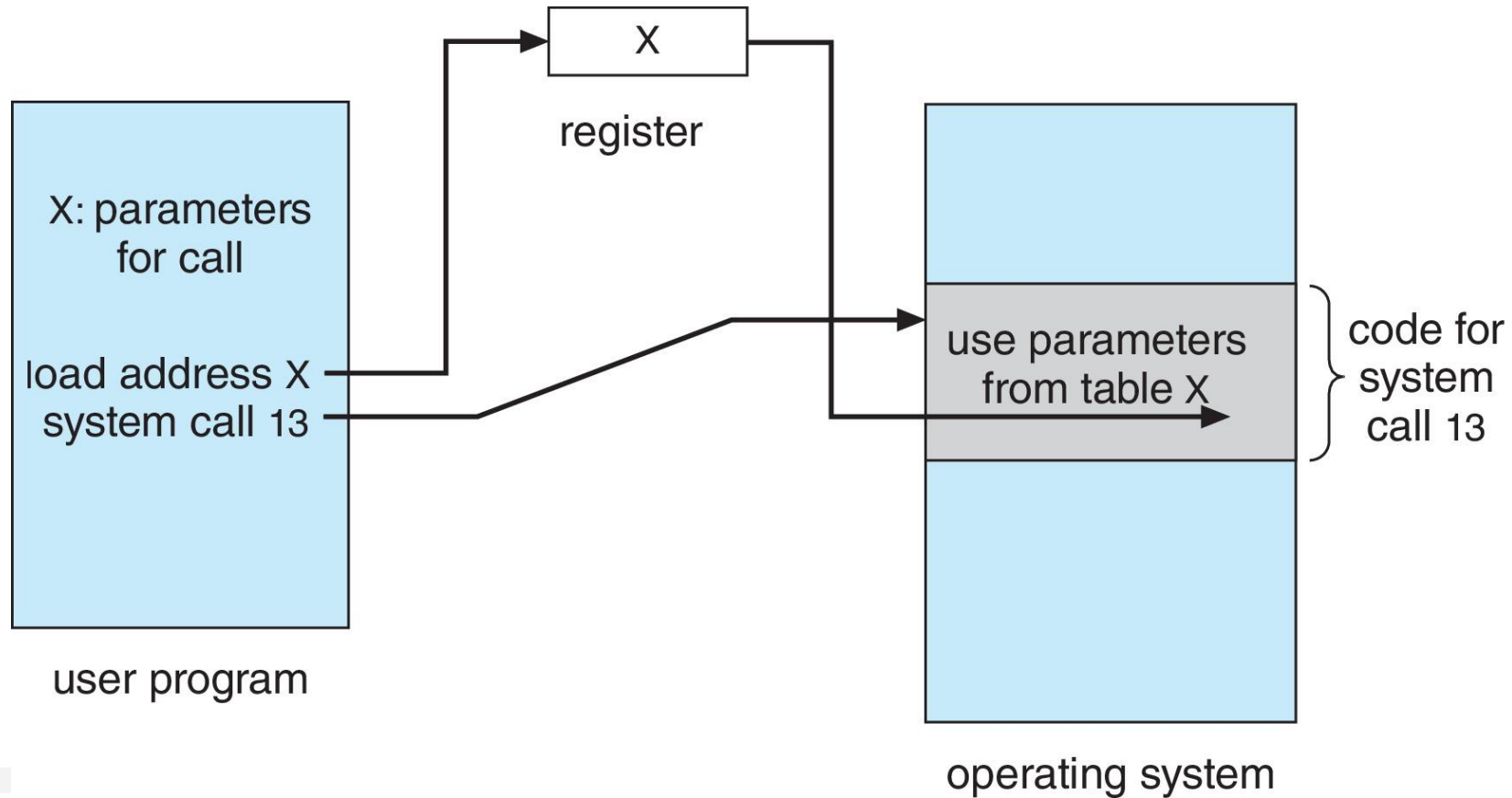
6. System Calls

- System call parameter passing
 - Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in registers
 - In some cases, may be more parameters than registers
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 - Parameters placed, or pushed, onto the 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

6. System Calls

- Parameter passing via table





6. System Calls

- 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



6. System Calls

- 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



6. System Calls

- 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



6. System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

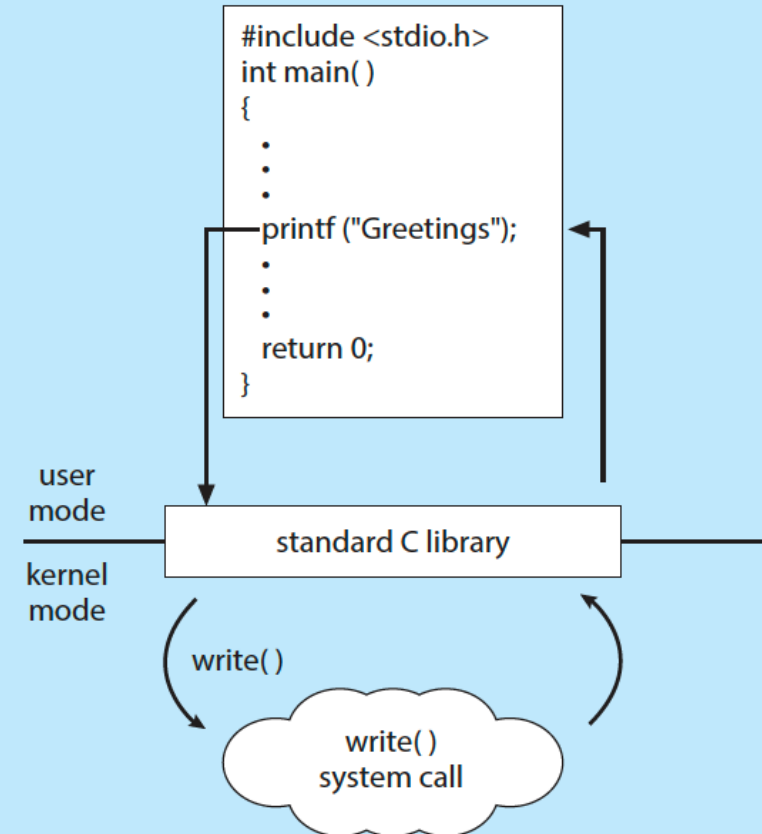
	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance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

6. System Calls

- C program invoking **printf()** library call, which calls **write()** system call.

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a 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:





7. Operating System Structure

- Monolithic structure
- Layered Approach
- Microkernel
- Modules
- Hybrid Systems

7. Operating System Structure

Monolithic Structure

- The simplest structure for organizing an operating system is no structure at all.
 - Place all of the functionality of the kernel into a single, static binary file that runs in a single address space.
 - Is a common technique for designing operating systems.

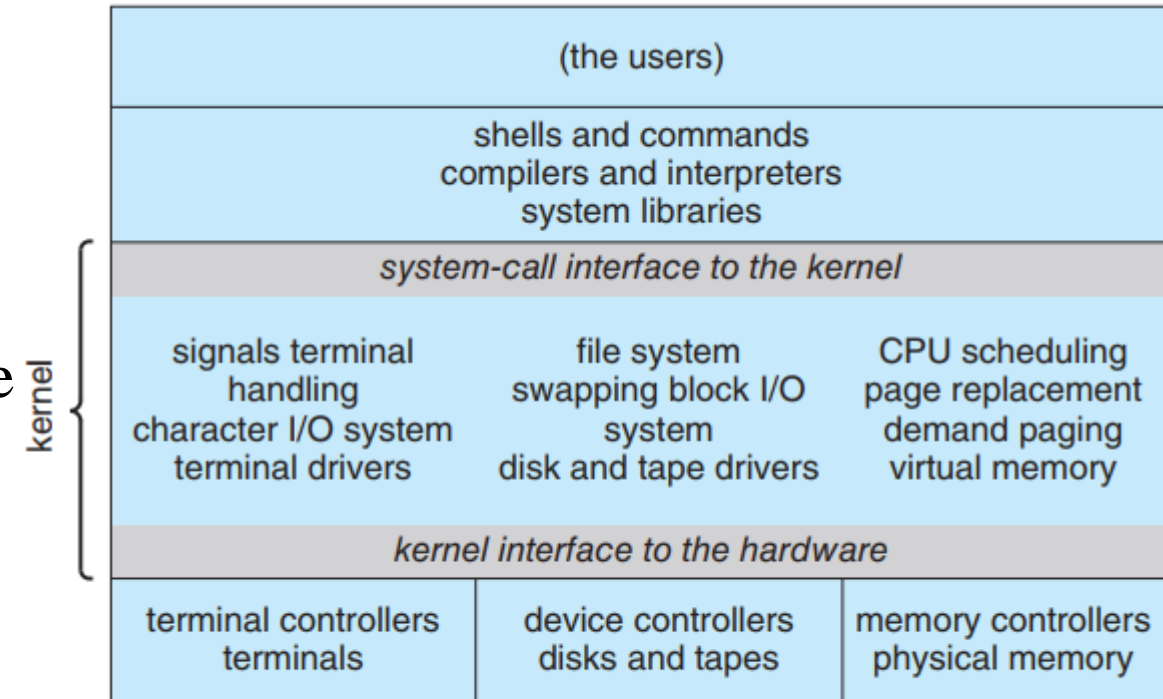


Figure 2.12 Traditional UNIX system structure.

7. Operating System Structure

Monolithic Structure

- The Linux operating system is based on UNIX and is structured similarly
 - Applications typically use the glibc standard C library when communicating with the system call interface to the kernel.
- The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but it does have a modular design that allows the kernel to be modified during run time.
- Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore, 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 operating systems.

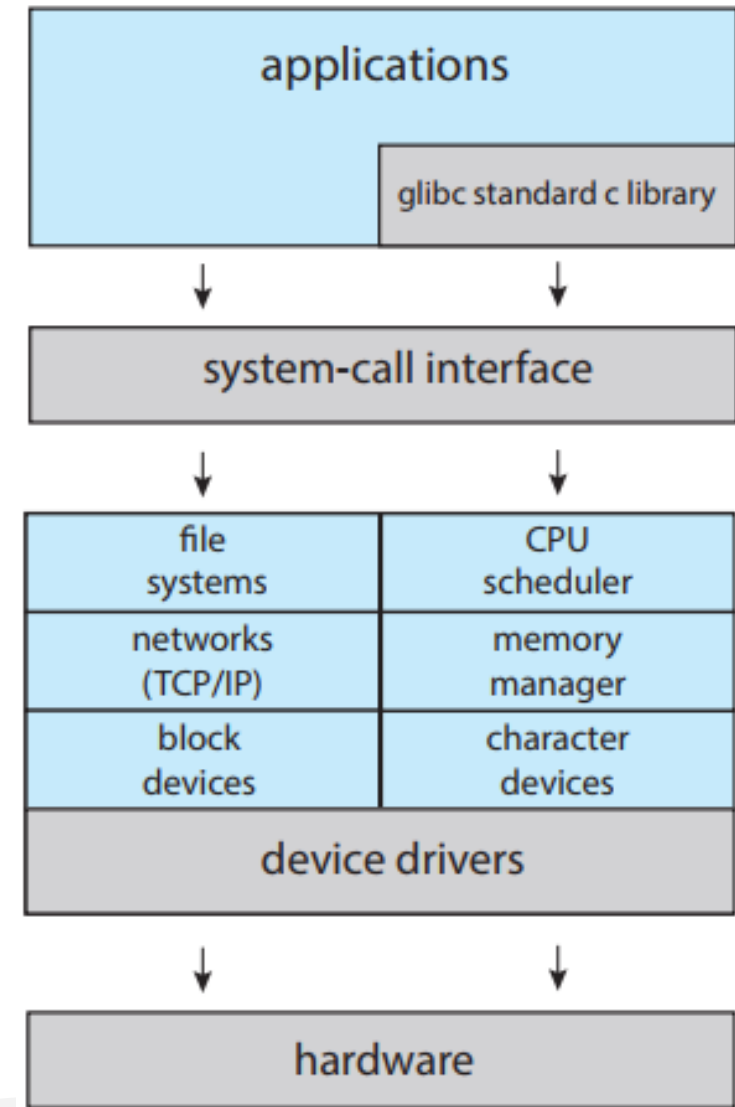


Figure 2.13 Linux system structure.

7. Operating System Structure

Layered Approach

- The monolithic approach is often known as a **tightly coupled** system because changes to one part of the system can have wide-ranging effects on other parts.
- Alternatively, we could design a **loosely coupled** system. Such a system is divided into separate, smaller components that have specific and limited functionality.
 - The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.
 - A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels).
 - The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

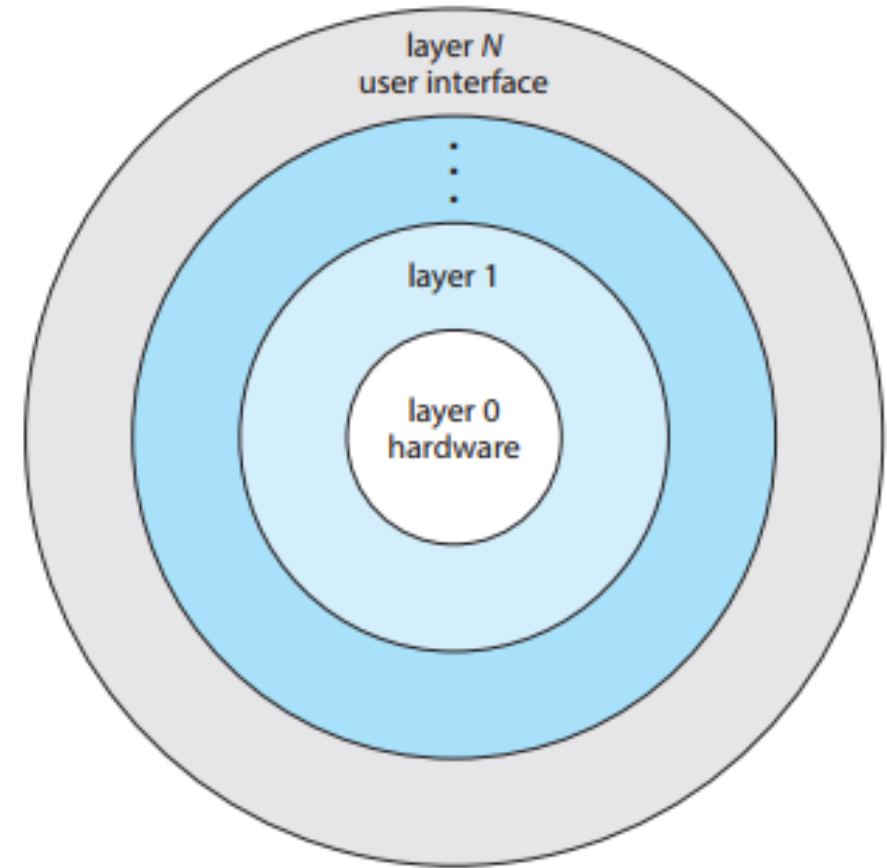


Figure 2.14 A layered operating system.

7. Operating System Structure

Layered Approach

- A layer M consists of data structures and a set of functions that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.
 - The main advantage of the layered approach is simplicity of construction and debugging.
 - Each layer is implemented only with 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.

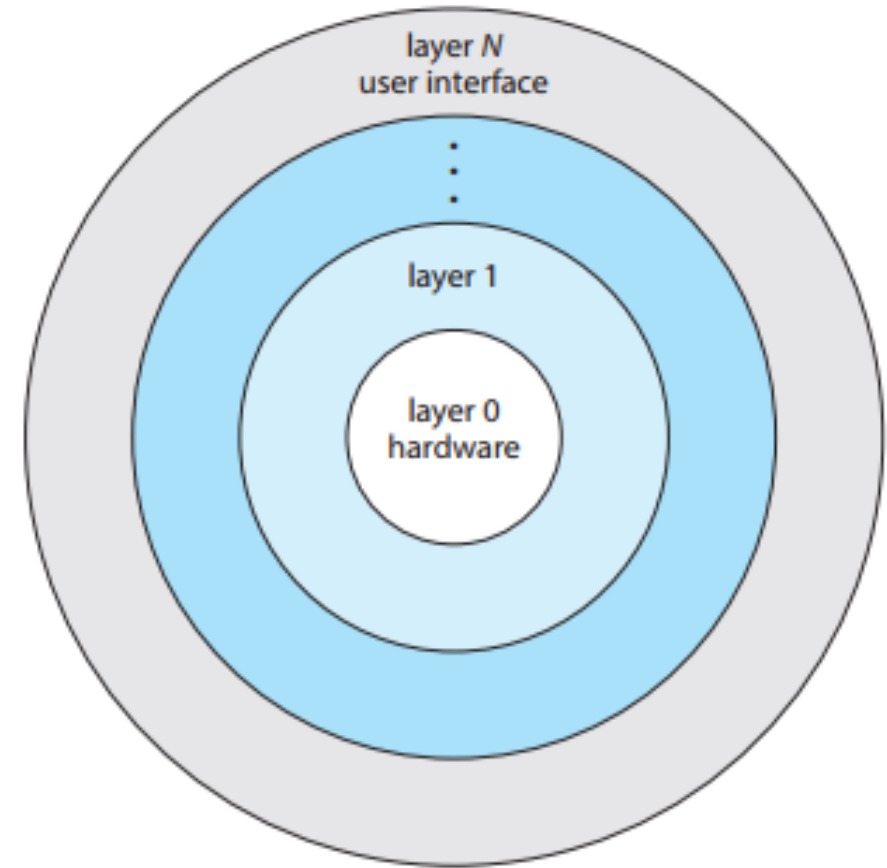


Figure 2.14 A layered operating system.

7. Operating System Structure

Microkernels

- The original UNIX system had a monolithic structure. As UNIX expanded, the kernel became large and difficult to manage.
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach.
 - This method structures the operating system by removing all nonessential components from the kernel and implementing them as user level programs that reside in separate address spaces. The result is a smaller kernel.

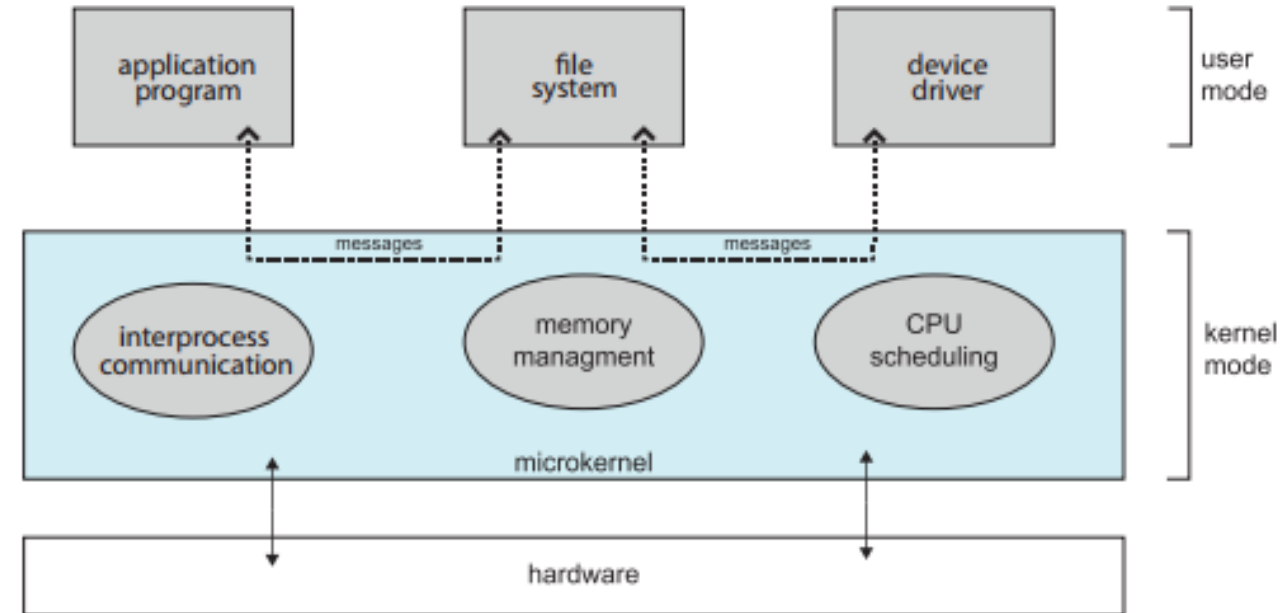


Figure 2.15 Architecture of a typical microkernel.

7. Operating System Structure

Microkernels

- Typically, microkernels provide minimal process and memory management, in addition to a communication facility.
 - The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
 - Communication is provided through message passing,
- One benefit of the microkernel approach is that it makes extending the operating system easier.

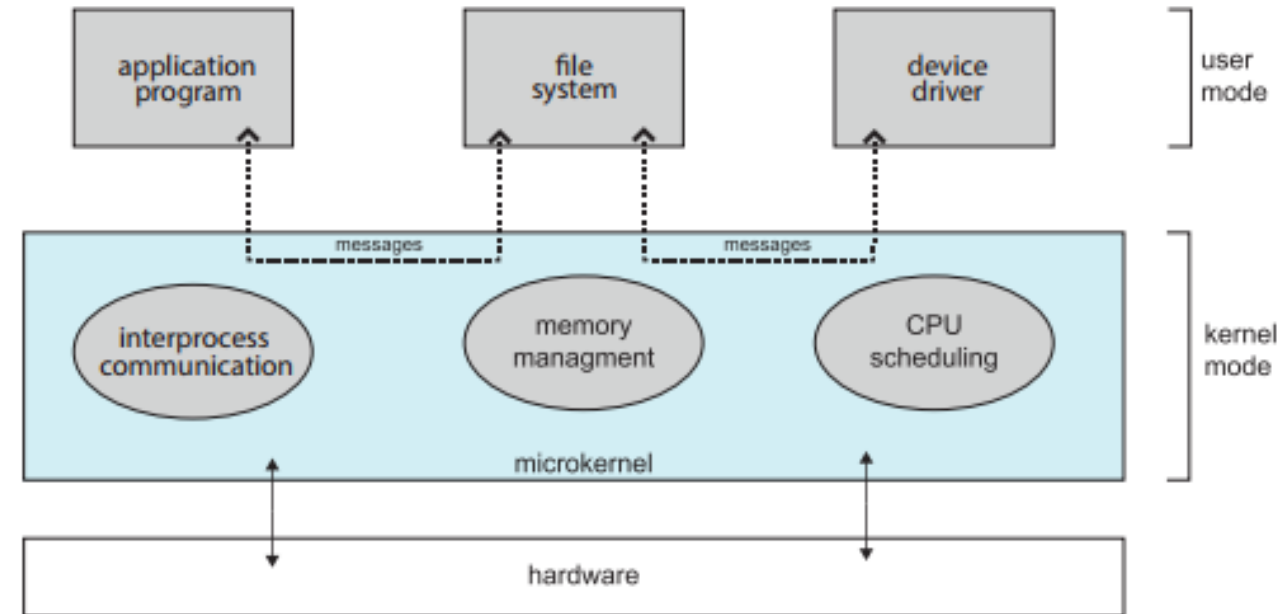


Figure 2.15 Architecture of a typical microkernel.

7. Operating System Structure

Microkernels

- One benefit of the microkernel approach is that it makes extending the operating system easier.
 - 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.
 - The resulting operating system is easier to port from one hardware design to another.
 - The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

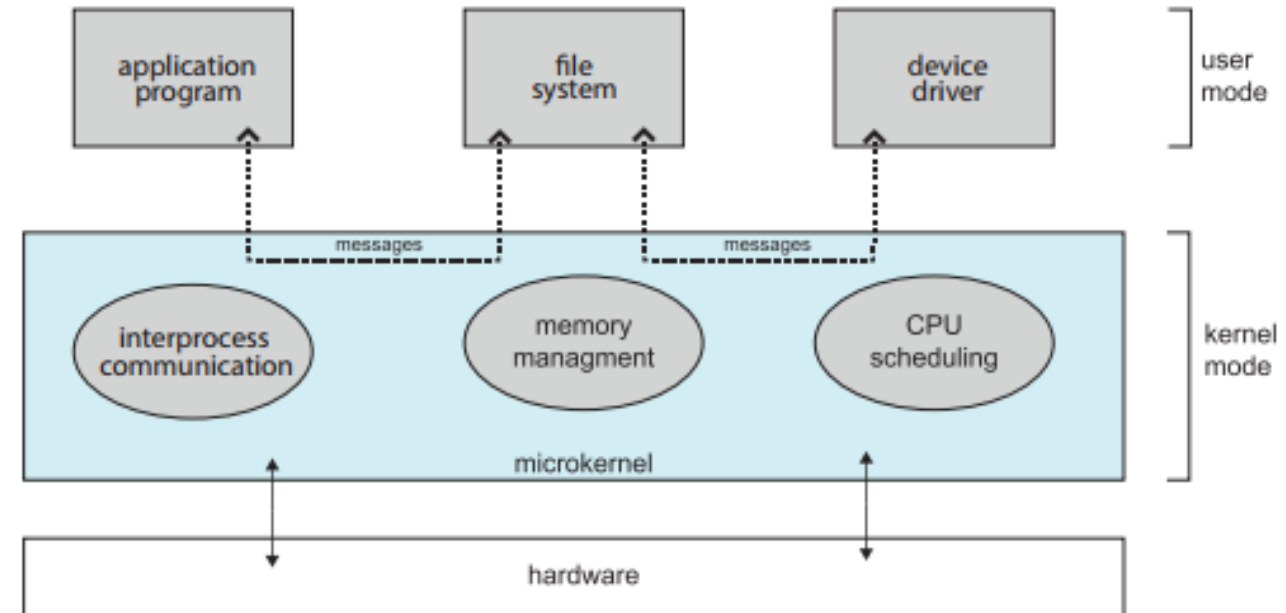


Figure 2.15 Architecture of a typical microkernel.

7. Operating System Structure

Microkernels

- The best-known illustration of a microkernel operating system is *Darwin*, the kernel component of the macOS and iOS operating systems.
- QNX : A real-time operating system for embedded systems
- Unfortunately, the performance of microkernels can suffer due to increased system-function overhead.
 - When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces.

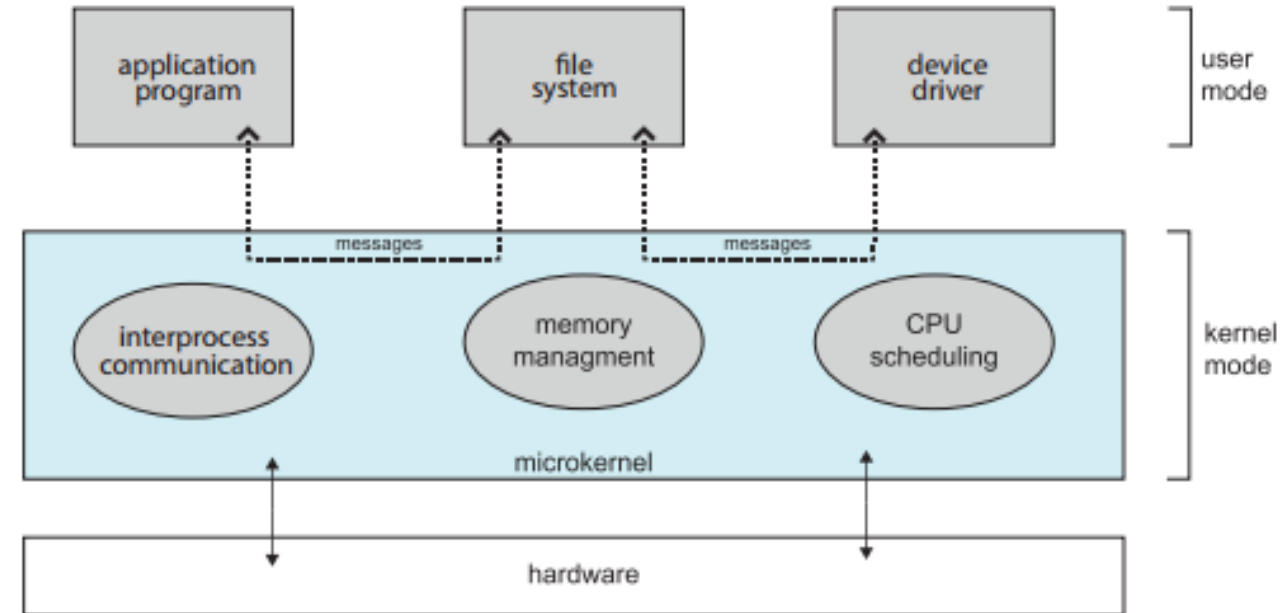


Figure 2.15 Architecture of a typical microkernel.

7. Operating System Structure

Microkernels

- Windows NT: The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95.
- Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely.
- By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel.

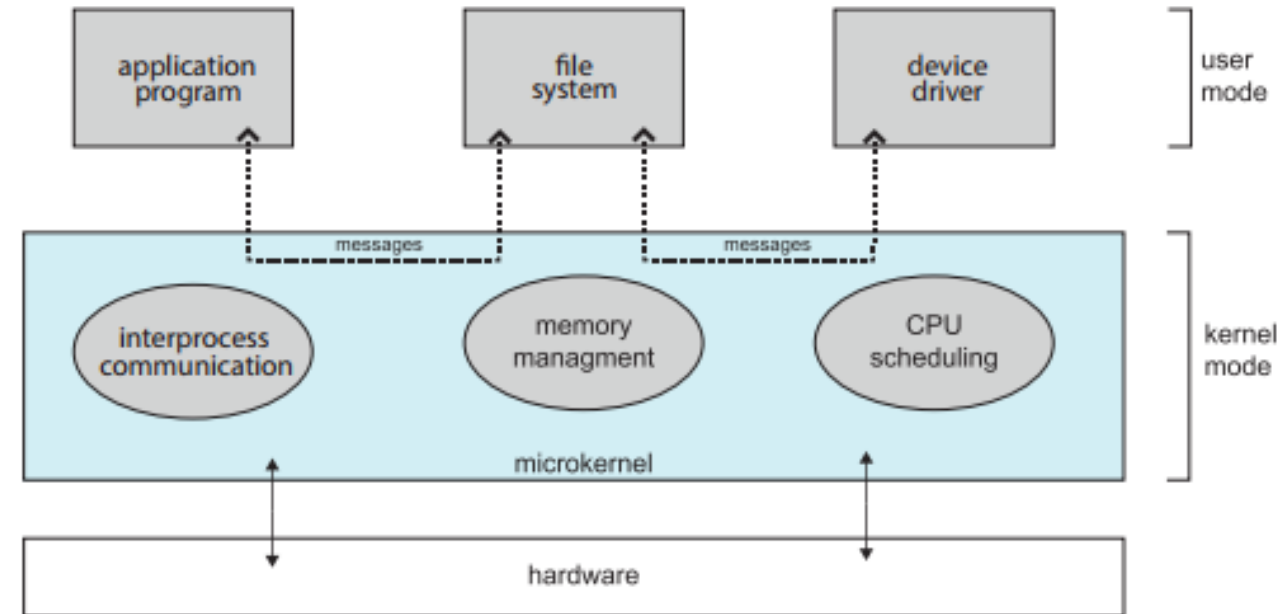


Figure 2.15 Architecture of a typical microkernel.



7. Operating System Structure

Modules

- Perhaps the best current methodology for operating-system design involves using **loadable kernel modules (LKMs)**.
 - Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time.
 - This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.
- 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 to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
 - Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.



7. Operating System Structure

Hybrid Systems

- In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine 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. However, it also modular, so that new functionality can be dynamically added to the kernel.
- Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system personalities) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules.



8. System Boot

- How does the hardware know where the kernel is or how to load that kernel?
- The process of starting a computer by loading the kernel is known as **booting** the system. On most systems, the boot process proceeds as follows:
 - 1. A small piece of code known as the **bootstrap program** or **boot loader** locates the kernel.
 - 2. The kernel is loaded into memory and started.
 - 3. The kernel initializes hardware.
 - 4. The root file system is mounted.



8. System Boot

- Some computer systems use a multistage boot process:
 - When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run.
 - This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block**.
 - The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.



8. System Boot

- Many recent computer systems have replaced the BIOS-based boot process with **UEFI** (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks.
 - UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.
- Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks.
 - In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine—for example, inspecting memory and the CPU and discovering devices. If the diagnostics pass, the program can continue with the booting steps.
 - The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be **running**.



8. System Boot

- **GRUB** is an open-source bootstrap program for Linux and UNIX systems. Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted.
 - As an example, the following are kernel parameters from the special Linux file `/proc/cmdline`, which is used at boot time:

`BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic`

`root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92`

`BOOT_IMAGE` is the name of the kernel image to be loaded into memory

`root` specifies a unique identifier of the root file system.