Operating Systems

A. Biswas

Architecture

Introduction

What is an operating system?

"Code" that:

- sits between **programs & hardware**
- sits between **different programs**
- sits between **different users**

What does it do?

Provides an **orderly and controlled** allocation of the **processors, memories and I/O devices** among the various programs competing for them.

Introduction

An operating system makes computers simple.

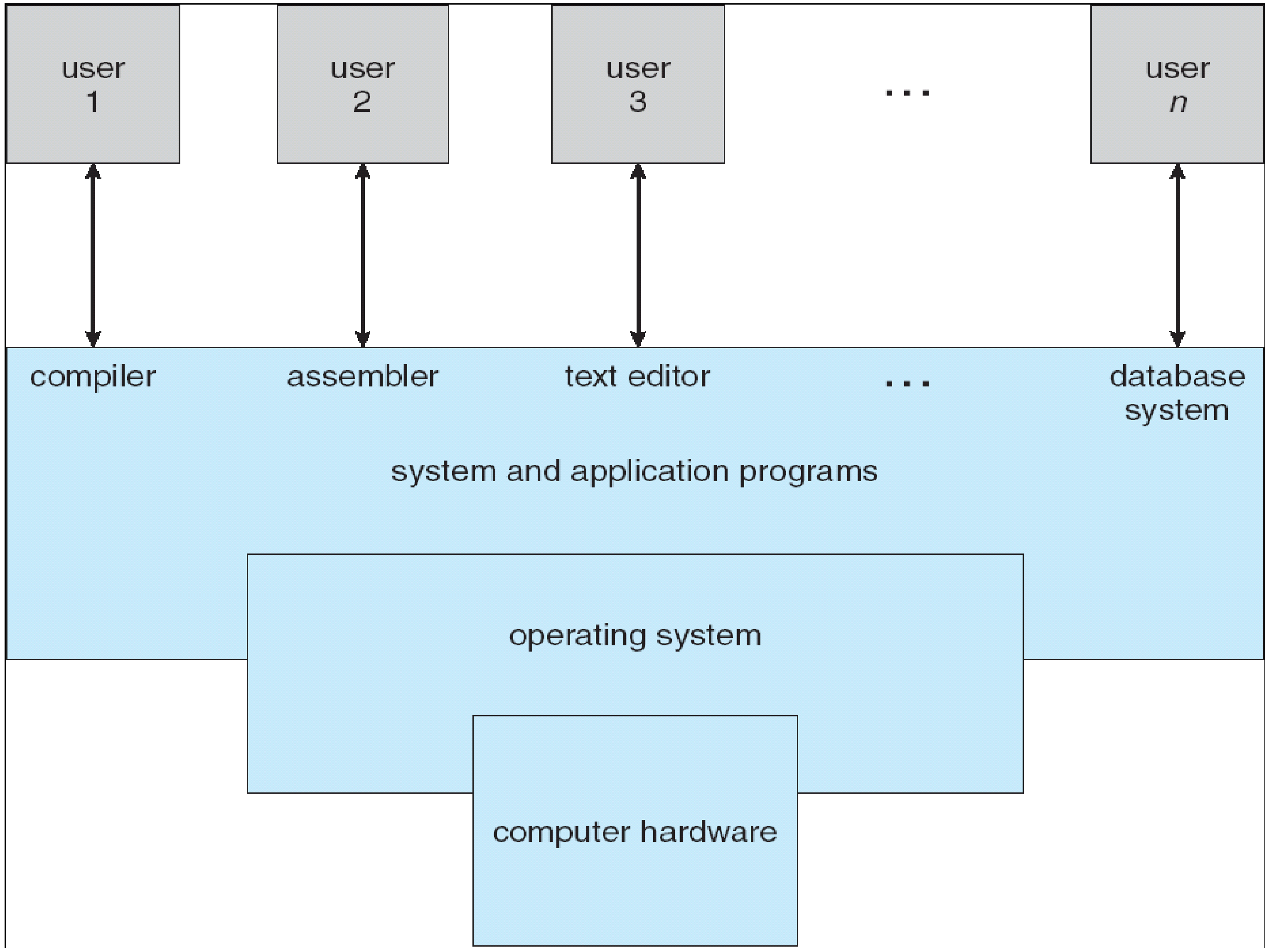
Resources:

- Allocation: finite resources and competing demands
- Protection: ensures some degree of safety and security
- Reclamation: voluntary at run time, implied at termination, involuntary and cooperative
- Virtualization: illusion of infinite, private resources

Services:

1. Abstraction
2. Simplification
3. Convenience
4. Standardization

Components of a Computer System



History of Operating Systems

- First generation 1945 – 1955
  – vacuum tubes, plug boards (no OS)

- Second generation 1955 – 1965
  – transistors, batch systems

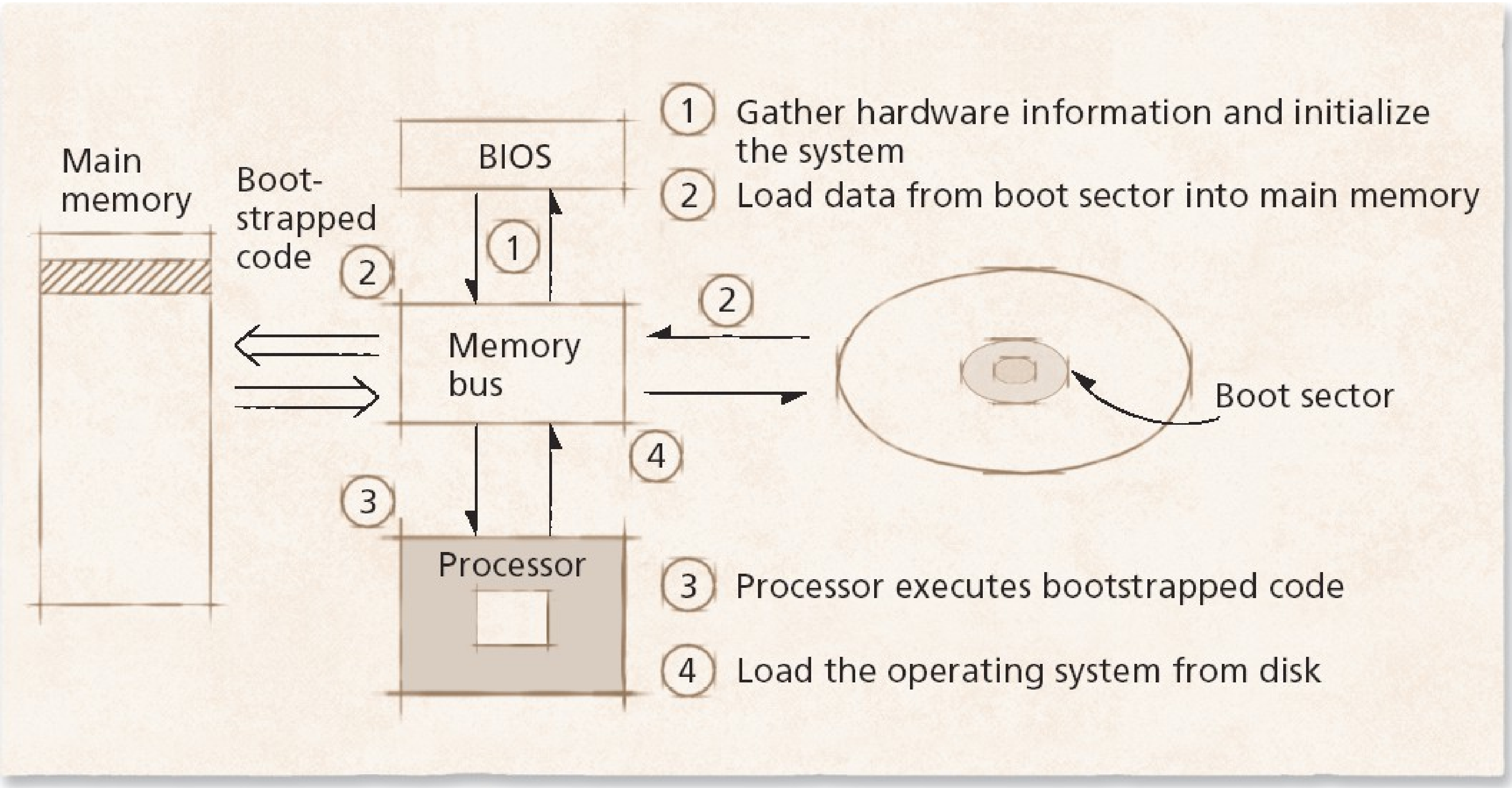
- Third generation 1965 – 1980
  – ICs and multiprogramming

- Fourth generation 1980 – present
  – personal computers, hand-held devices, sensors

Computer Startup

- **bootstrap program** is loaded at power-up
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

Computer Startup



1. Gather hardware information and initialize the system
2. Load data from boot sector into main memory
3. Processor executes bootstrapped code
4. Load the operating system from disk

Operating System Architectures

- **Today’s operating systems tend to be complex**
  - **Provide many services**
  - **Support variety of hardware and software**
  - **Operating system architectures help manage this complexity**
    - Organize operating system components
    - Specify privilege with which each component executes

Kernel Architecture

The kernel is the core of an operating system.
It is the software responsible for running programs
and providing secure access to the machine's hardware.

Kernel Architecture

Scheduling:

Since there are many programs, and resources are limited, the kernel also decides when and how long a program should run. This is called scheduling.

Kernel Architecture

Hardware Abstraction:

Accessing the hardware directly can be very complex, since there are many different hardware designs for the same type of component.

Kernels usually implement some level of hardware abstraction (a set of instructions universal to all devices of a certain type) to hide the underlying complexity from applications and provide a clean and uniform interface.

This helps application programmers to develop programs without having to know how to program for specific devices.

The kernel relies upon software drivers that translate the generic command into instructions specific to that device.

Kernel Architecture

Categories of kernels:

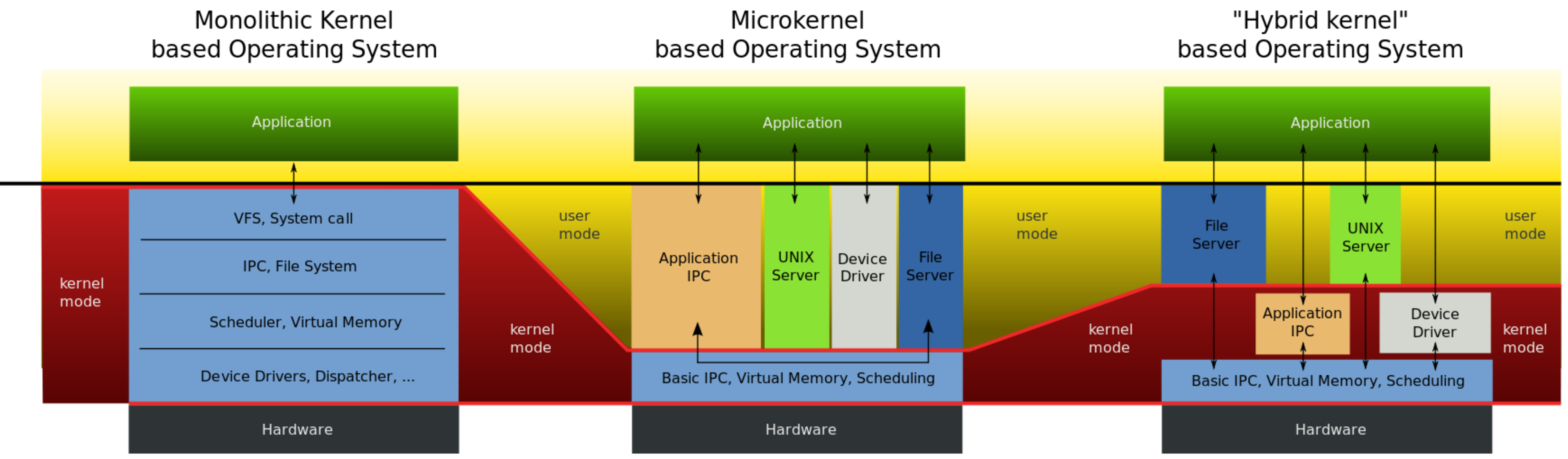
Four broad categories of kernels:

Monolithic kernels provide rich and powerful abstractions of the underlying hardware.

Microkernels provide a small set of simple hardware abstractions and use applications called servers to provide more functionality.

Exokernels provide minimal abstractions, allowing low-level hardware access. In exokernel systems, library operating systems provide the abstractions typically present in monolithic kernels.

Hybrid (*modified microkernels*) are much like pure microkernels, except that they include some additional code in kernelspace to increase performance.



Monolithic Kernel based Operating System
Microkernel based Operating System
"Hybrid kernel" based Operating System
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
kernel mode
user mode
Hardware
Application
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware
Application
File Server
UNIX Server
Application IPC
Device Driver
Basic IPC, Virtual Memory, Scheduling
Hardware

Kernel Architecture

Monolithic Kernel:

The monolithic approach is to define a high-level virtual interface over the hardware, with a set of primitives or system calls to implement operating system services such as process management, concurrency, and memory management in several modules that run in supervisor mode.

Kernel Architecture

Monolithic Kernel:

Even if every module servicing these operations is separate from the whole, the code integration is very tight and difficult to do correctly, and, since all the modules run in the same address space, a bug in one module can bring down the whole system.

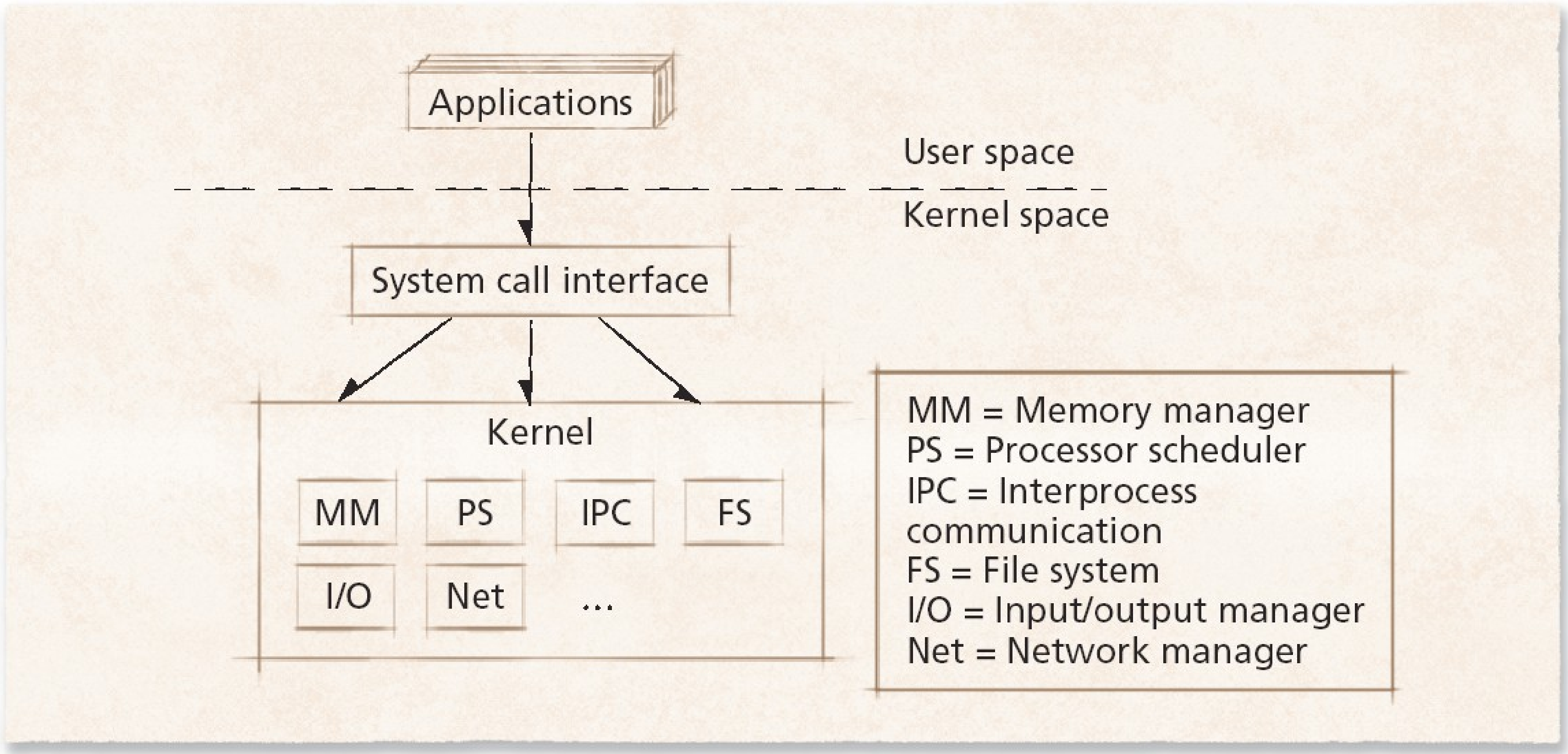
Kernel Architecture

Monolithic Kernel:

However, when the implementation is complete and trustworthy, the tight internal integration of components allows the low-level features of the underlying system to be effectively utilized, making a good monolithic kernel highly efficient.

Monolithic Architecture

Monolithic operating system kernel architecture



Monolithic Architecture

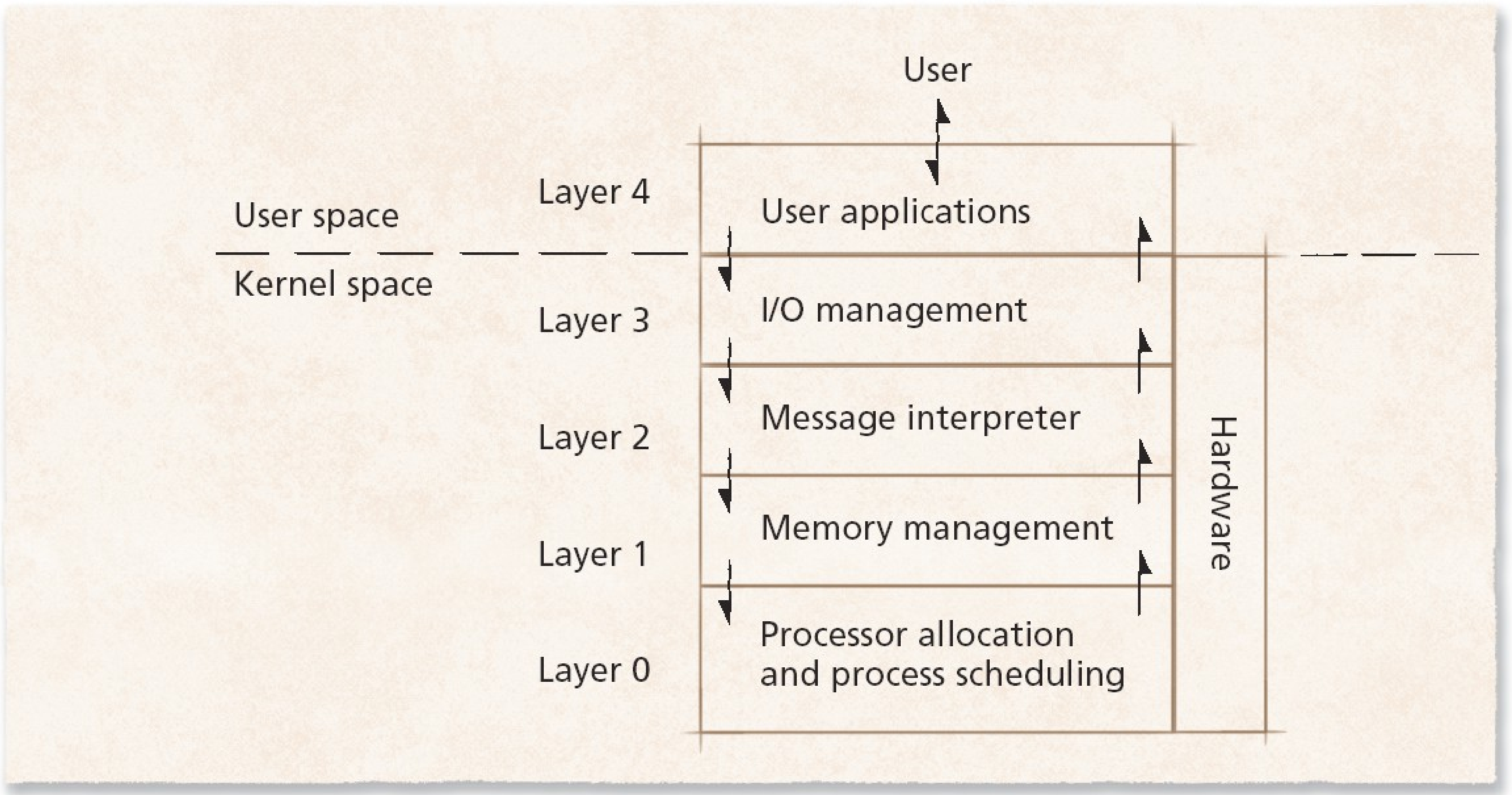
- Monolithic operating system
  - Every component contained in kernel
    - Any component can directly communicate with any other
  - Tend to be highly efficient
  - Disadvantage is difficulty determining source of subtle errors

Monolithic Architecture

- Examples:
  - Linux
  - UNIX (BSD, Solaris)
  - MS-DOS
  - MAC-OS till 8.6

Layered Architecture

Layers of the operating system



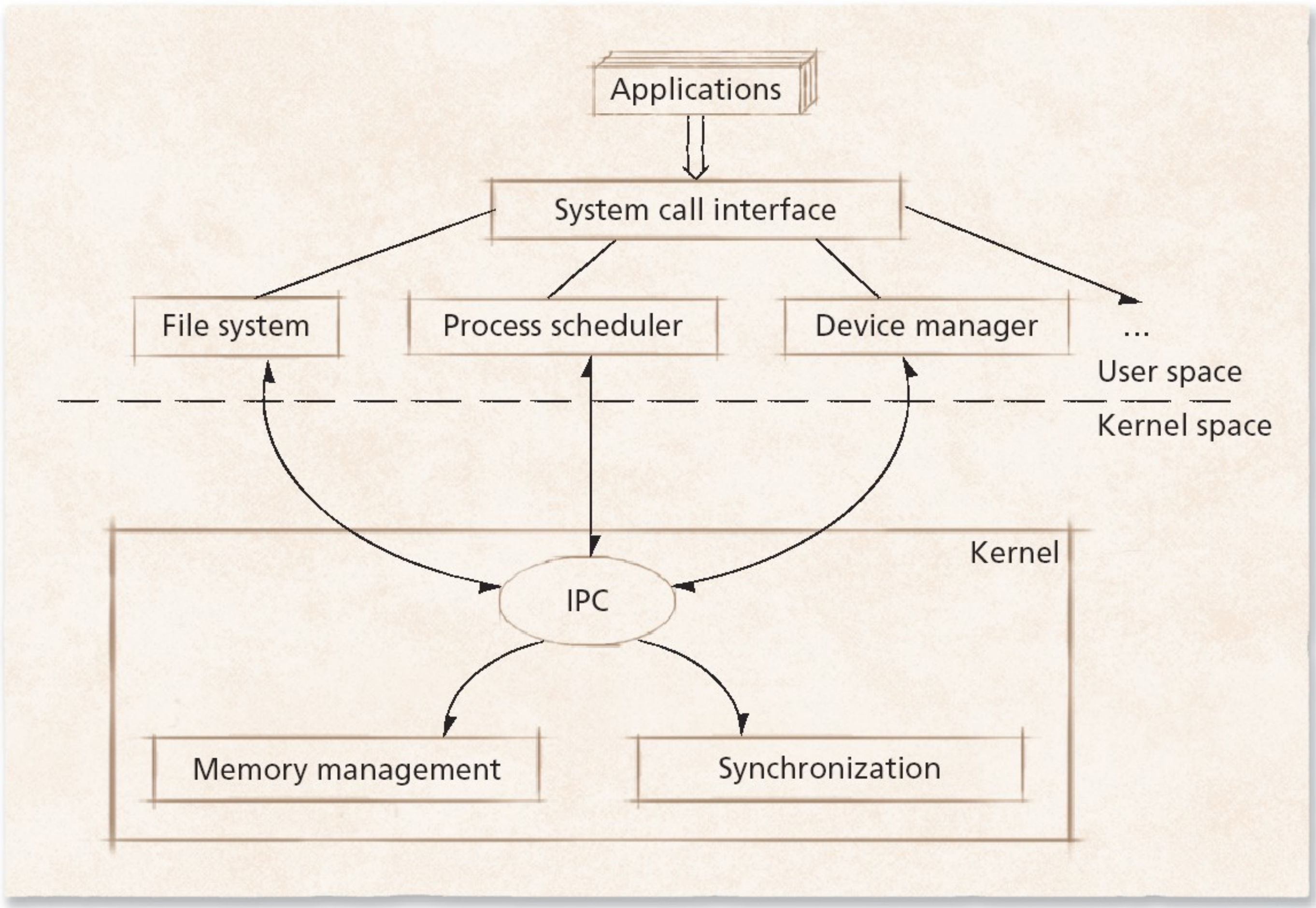
Layered Architecture

- **Layered approach to operating systems**
  - Tries to improve on monolithic kernel designs
    - Groups components that perform similar functions into layers
  - Each layer communicates only with layers immediately above and below it
  - Processes' requests might pass through many layers before completion
  - System throughput can be less than monolithic kernels
    - Additional methods must be invoked to pass data and control

Example: THE operating system

Microkernel Architecture

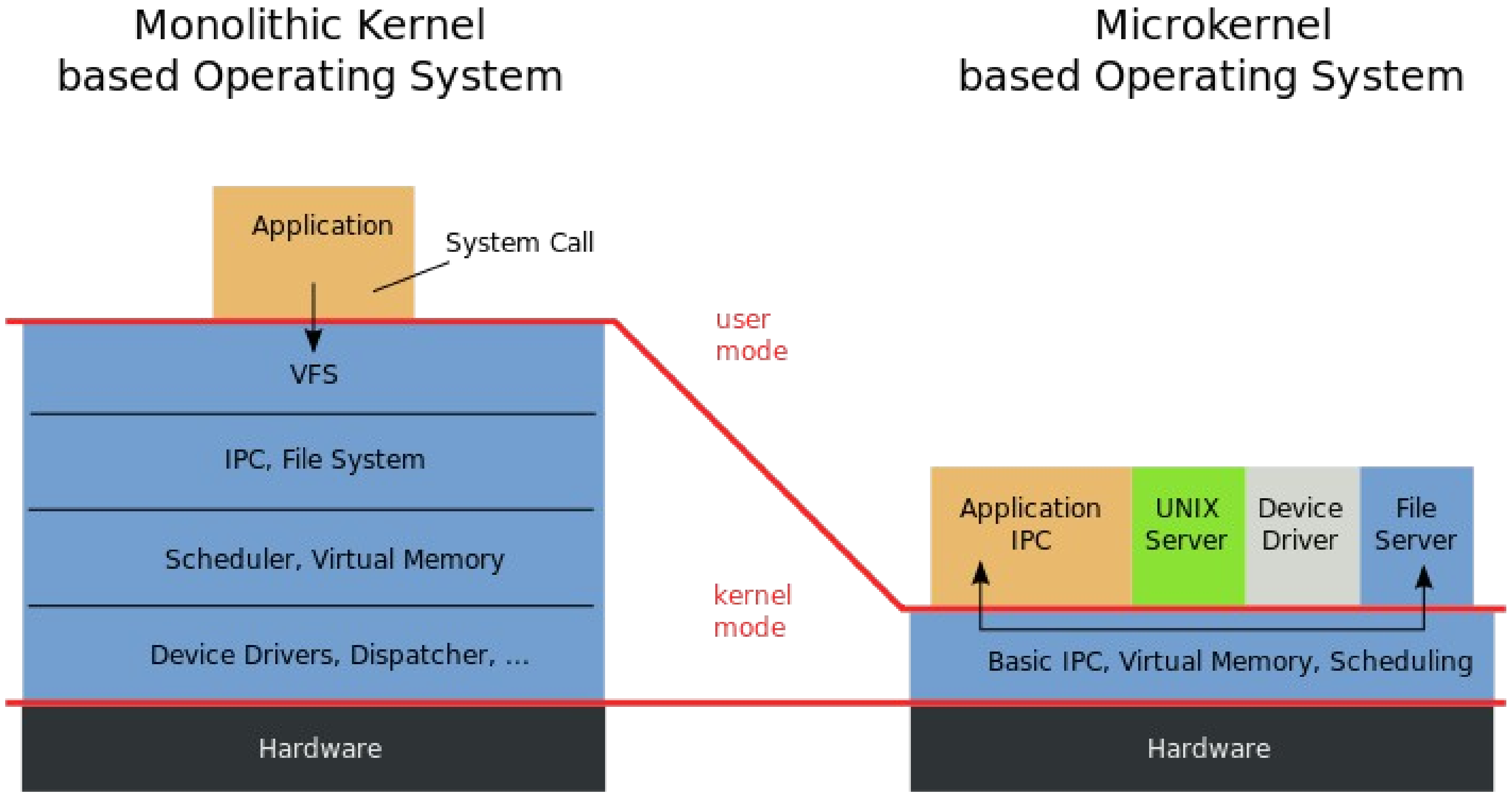
Microkernel operating system architecture



Microkernel Architecture

The microkernel approach is to define a very simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as thread management, address spaces and interprocess communication.

Microkernel Architecture


Monolithic Kernel based Operating System
Microkernel based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user mode
kernel mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware

Microkernel Architecture

All other services, those normally provided by the kernel such as networking, are implemented in user-space programs referred to as servers.

Microkernel Architecture

Servers are programs like any others, allowing the operating system to be modified simply by starting and stopping programs.

For a small machine without networking support, for instance, the networking server simply isn't started.

Microkernel Architecture

Under a traditional system this would require the kernel to be recompiled, something well beyond the capabilities of the average end-user.

In theory the system is also more stable, because a failing server simply stops a single program, rather than causing the kernel itself to crash.

Microkernel Architecture

- Microkernel operating system architecture
  - Provides only small number of services
    - Attempt to keep kernel small and scalable
  - High degree of modularity
    - Extensible, portable and scalable
  - Increased level of intermodule communication
    - Can degrade system performance

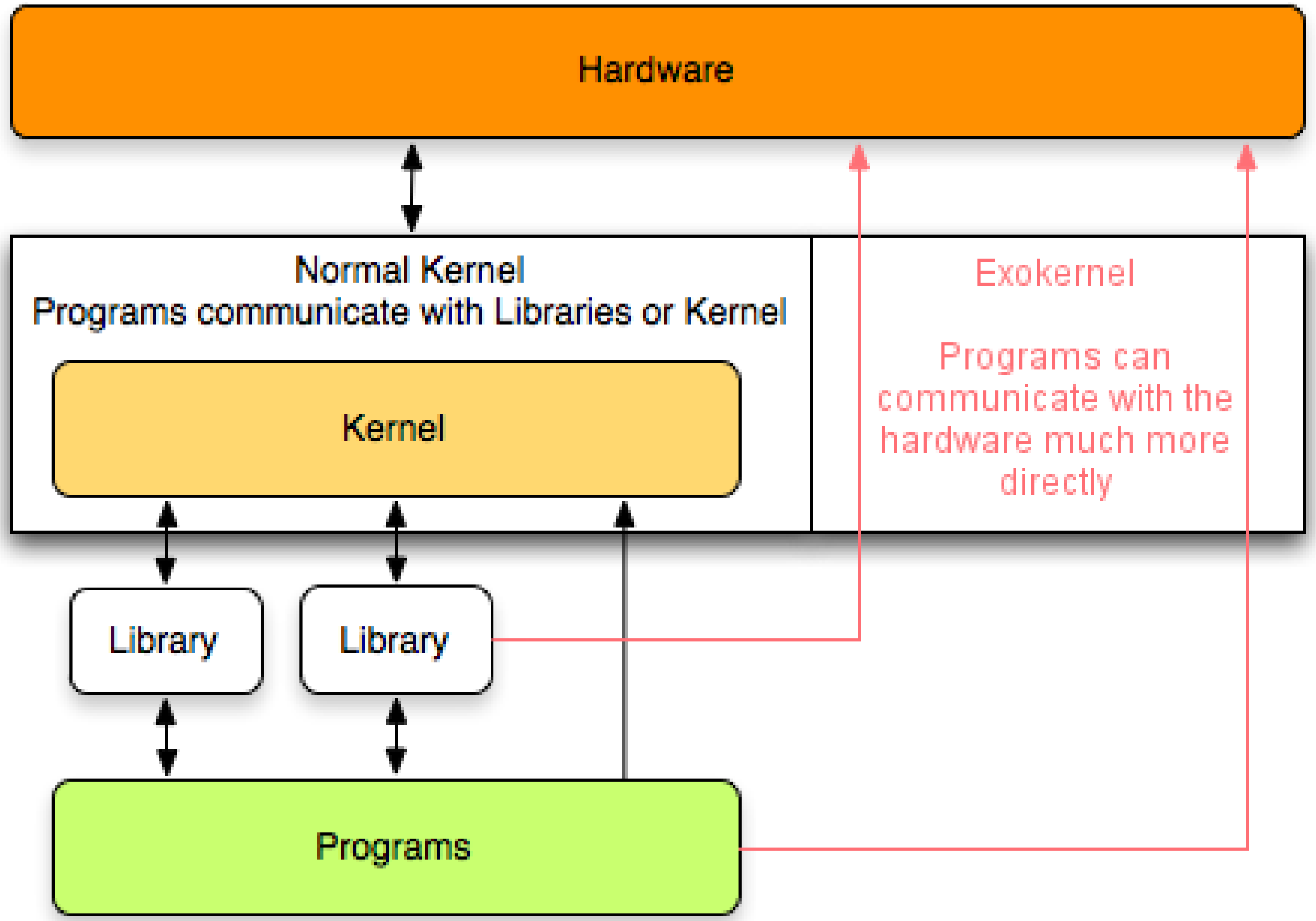
Microkernel Architecture

- Examples:
  - MACH
  - NT
  - EROS
  - MINIX

Exokernel

Exokernel is an operating system kernel developed by the MIT Parallel and Distributed Operating Systems group.

Exokernel


Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Library
Library
Programs
Exokernel
Programs can communicate with the hardware much more directly

Exokernel

Exokernels force as few abstractions as possible on developers, enabling them to make as many decisions as possible about hardware abstractions.

Exokernel

Exokernels are tiny, since functionality is limited to ensuring protection and multiplexing of resources, which are vastly simpler than conventional microkernels' implementation of message passing and monolithic kernels' implementation of abstractions.

Exokernel

Implemented applications are called library operating systems; they may request specific memory addresses, disk blocks, etc.

The kernel only ensures that the requested resource is free, and the application is allowed to access it.

Exokernel

This low-level hardware access allows the programmer to implement custom abstractions, and omit unnecessary ones, most commonly to improve a program's performance.

It also allows programmers to choose what level of abstraction they want, high, or low.

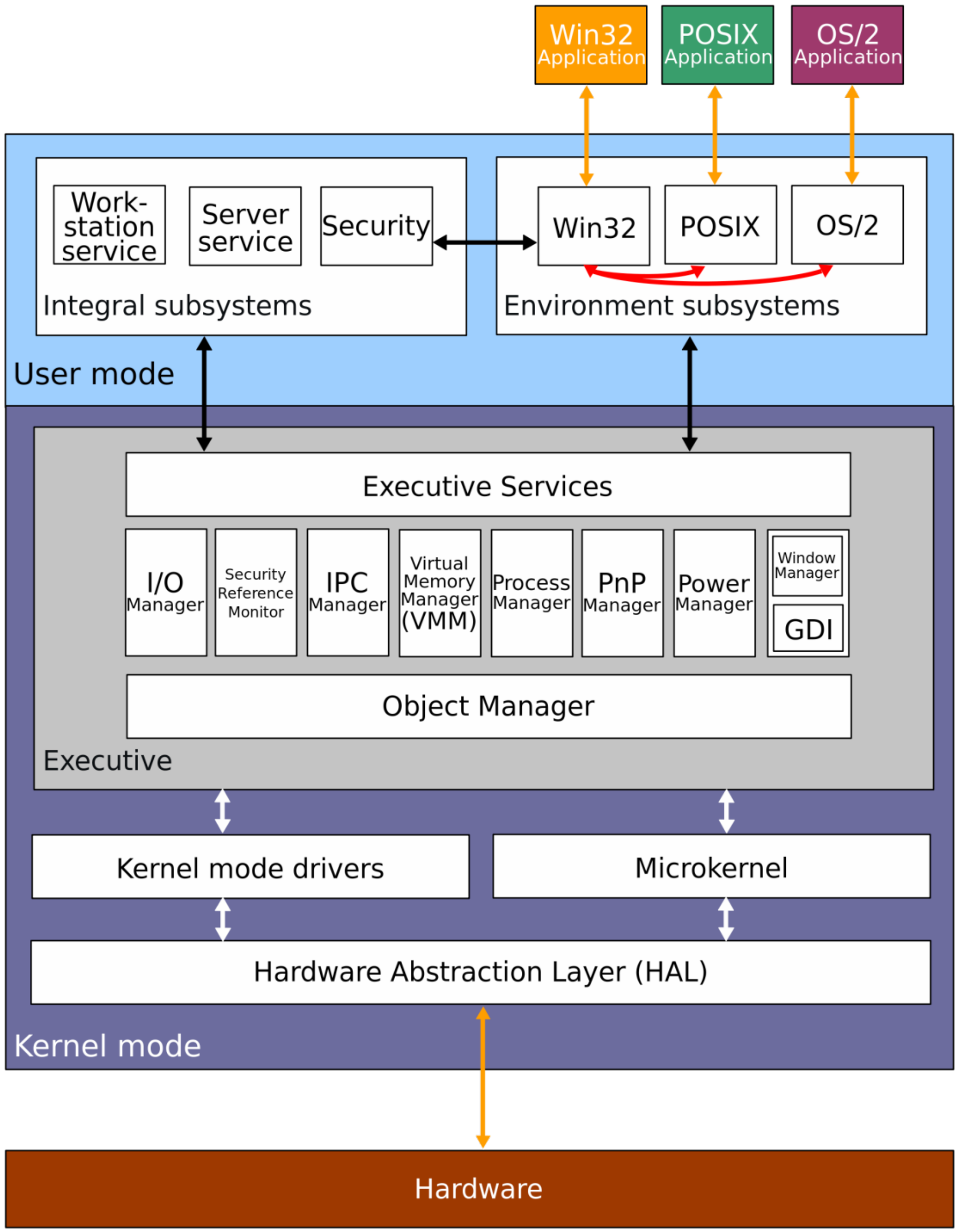
Hybrid Kernel

A hybrid kernel is one that combines aspects of both micro and monolithic kernels, but there is no exact definition.

Often, "hybrid kernel" means that the kernel is highly modular, but all runs in the same address space.

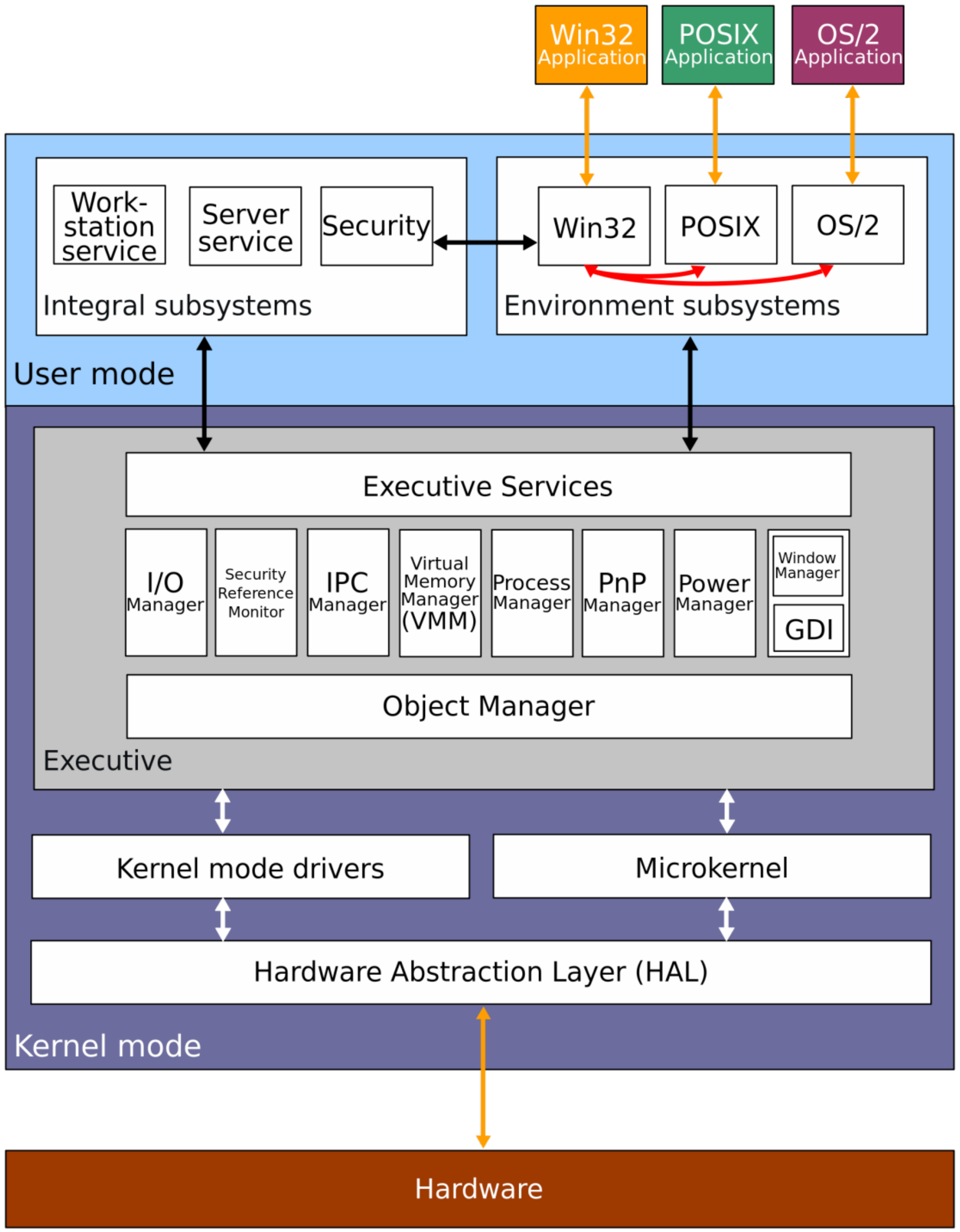
This allows the kernel avoid the overhead of a complicated message passing system within the kernel, while still retaining some microkernel-like features.

Hybrid Kernel



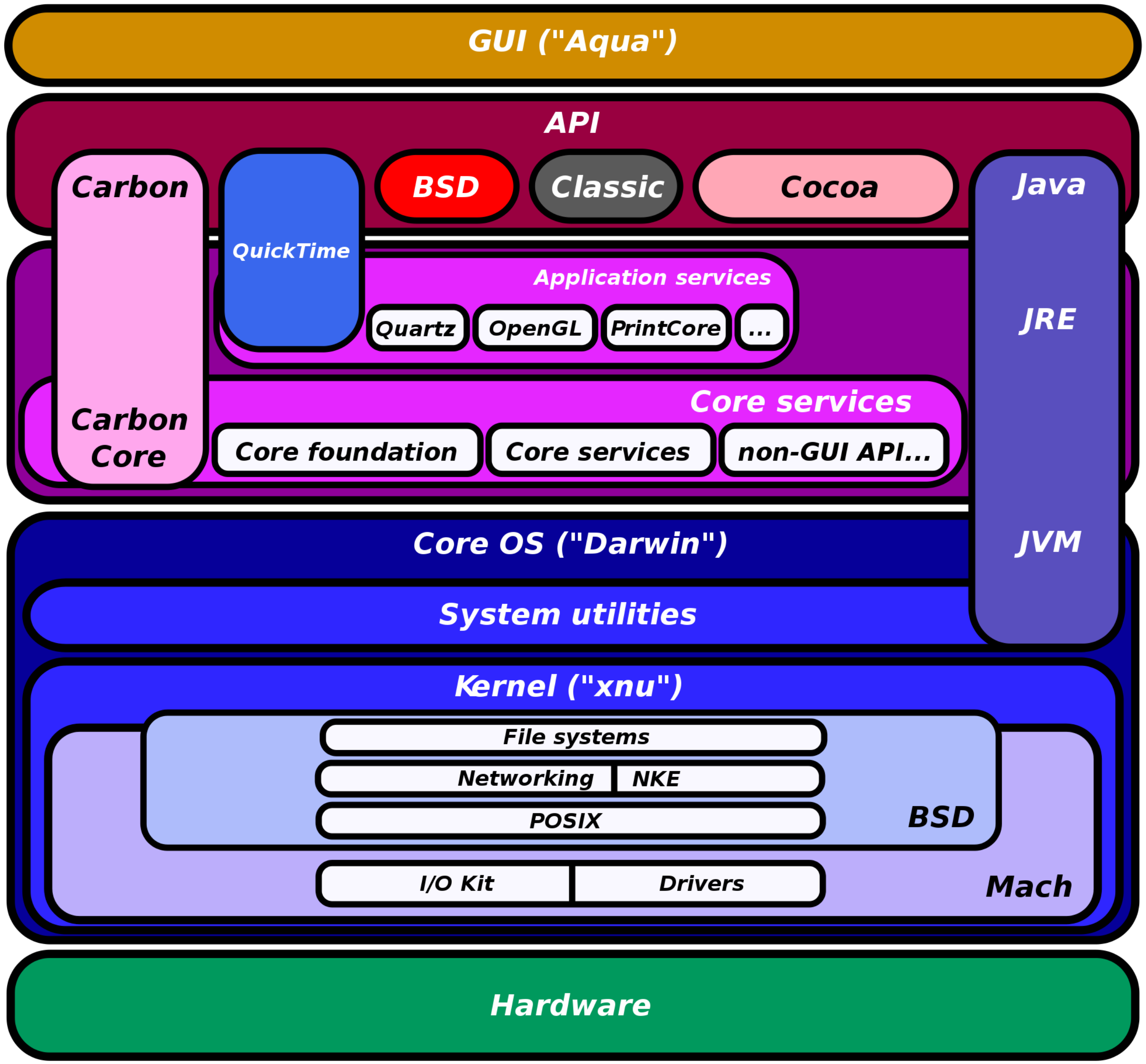
A small microkernel limited to core functions such as first-level interrupt handling, thread scheduling and synchronization primitives. This allows for the possibility of using either direct procedure calls or interprocess communication (IPC) to communicate between modules, and hence for the potential location of modules in different address spaces (for example in either kernel space or server processes).

Hybrid Kernel

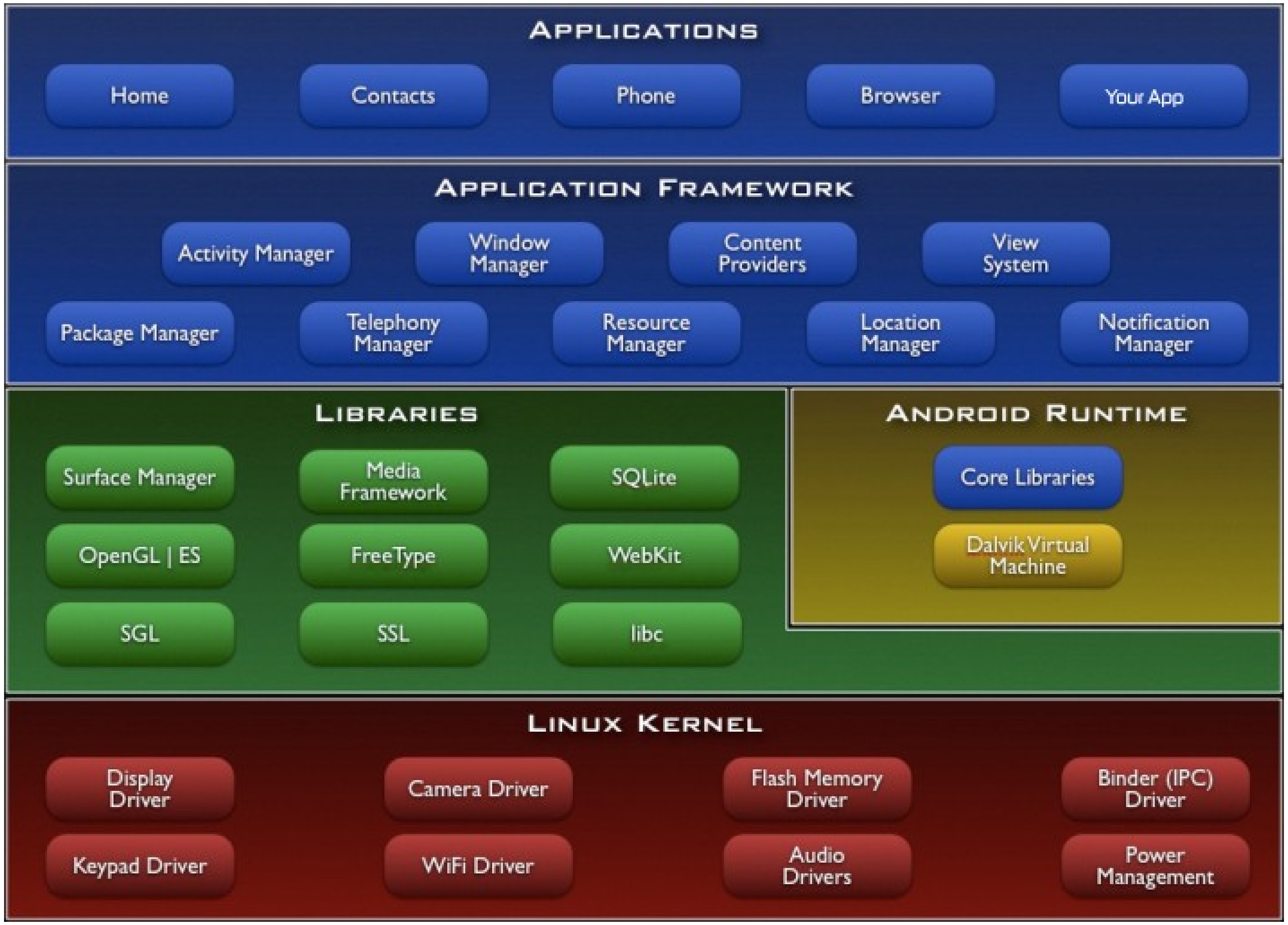


The reason NT is not a micro-kernel system is that nearly all of the subsystems providing system services, including the entire Executive, run in kernel mode (in the same address space as the microkernel itself), rather than in user-mode server processes, as would be the case with a microkernel design.

Hybrid Kernel



Android Architecture



Android Architecture

Linux Kernel

The basic layer is the Linux kernel. The whole Android OS is built on top of the Linux 2.6 Kernel with some further architectural changes made by Google. It is this Linux that interacts with the hardware and contains all the essential hardware drivers.

Drivers are programs that control and communicate with the hardware. For example, consider the Bluetooth function. All devices has a Bluetooth hardware in it. Therefore the kernel must include a Bluetooth driver to communicate with the Bluetooth hardware.

Android Architecture

Linux Kernel

The Linux kernel also acts as an abstraction layer between the hardware and other software layers. Android uses the Linux for all its core functionality such as Memory management, process management, networking, security settings etc. As the Android is built on a most popular and proven foundation, it made the porting of Android to variety of hardware, a relatively painless task.

Android Architecture

Libraries

The next layer is the Android’s native libraries. It is this layer that enables the device to handle different types of data. These libraries are written in c or c++ language and are specific for a particular hardware.

Some of the important native libraries include the following:

Surface Manager: It is used for compositing window manager with off-screen buffering. Off-screen buffering means you cant directly draw into the screen, but your drawings go to the off-screen buffer. There it is combined with other drawings and form the final screen the user will see. This off screen buffer is the reason behind the transparency of windows.

Android Architecture

Media framework: Media framework provides different media codecs allowing the recording and playback of different media formats

SQLite: SQLite is the database engine used in android for data storage purposes

WebKit: It is the browser engine used to display HTML content

OpenGL: Used to render 2D or 3D graphics content to the screen

Android Architecture

Android Runtime
Android Runtime consists of Dalvik Virtual machine and Core Java libraries.

Dalvik Virtual Machine
It is a type of JVM used in android devices to run apps and is optimized for low processing power and low memory environments. Unlike the JVM, the Dalvik Virtual Machine doesn’t run .class files, instead it runs .dex files. .dex files are built from .class file at the time of compilation and provides hifger efficiency in low resource environments. The Dalvik VM allows multiple instance of Virtual machine to be created simultaneously providing security, isolation, memory management and threading support. It is developed by Dan Bornstein of Google.

Core Java Libraries
These are different from Java SE and Java ME libraries. However these libraries provides most of the functionalities defined in the Java SE libraries.

Android Architecture

Application Framework
These are the blocks that our applications directly interacts with. These programs manage the basic functions of phone like resource management, voice call management etc. As a developer, you just consider these are some basic tools with which we are building our applications.

Important blocks of Application framework are:

Activity Manager: Manages the activity life cycle of applications

Content Providers: Manage the data sharing between applications

Telephony Manager: Manages all voice calls. We use telephony manager if we want to access voice calls in our application.

Location Manager: Location management, using GPS or cell tower

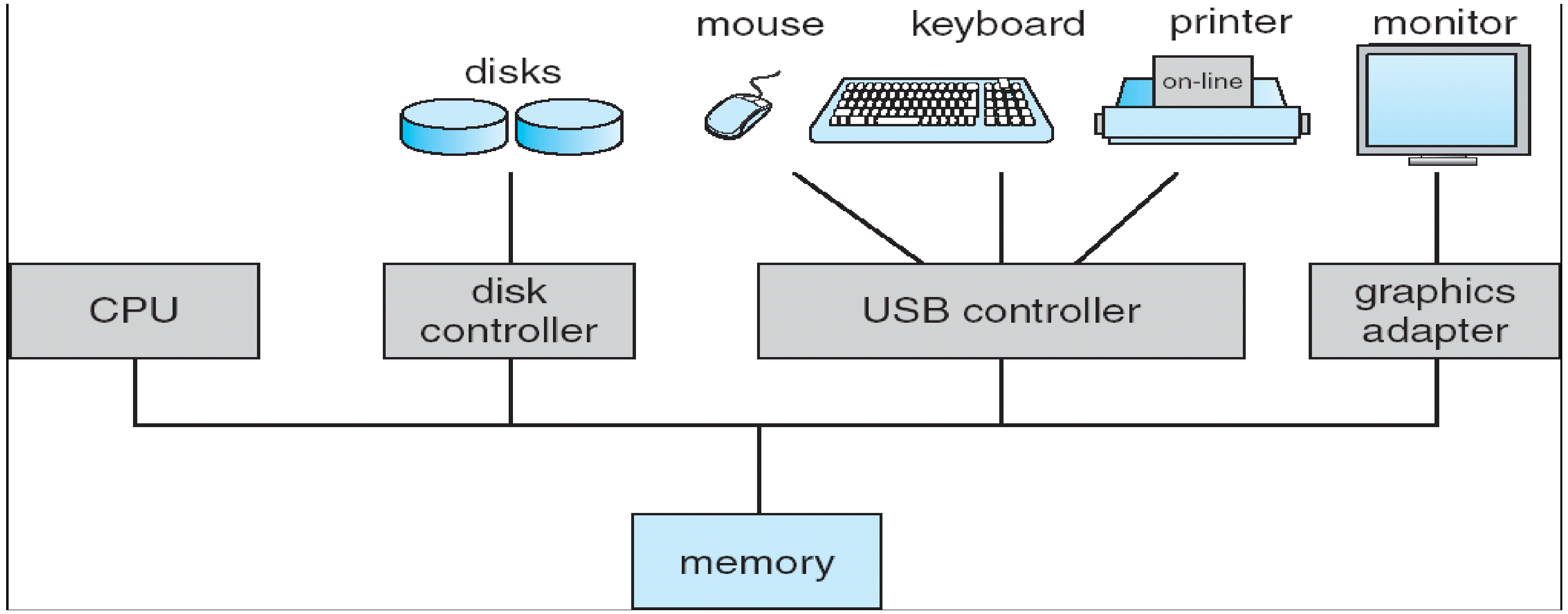
Resource Manager: Manage the various types of resources we use in our Application

Android Architecture

Applications
Applications are the top layer in the Android architecture and this is where our applications are gonna fit. Several standard applications comes pre-installed with every device, such as:
SMS client app
Dialer
Web browser
Contact manager
As a developer we are able to write an app which replace any existing system app. That is, you are not limited in accessing any particular feature. You are practically limitless and can whatever you want to do with the android (as long as the users of your app permits it). Thus Android is opening endless opportunities to the developer.

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Computer System Operation

- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- 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*.

Interrupts

- Suspends the normal sequence of execution

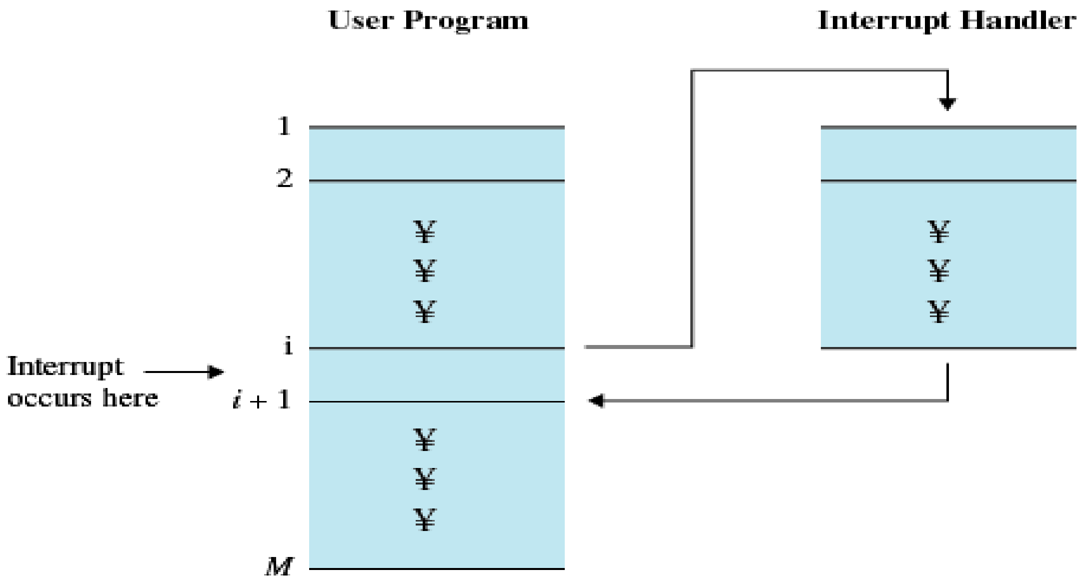


Figure 1.6 Transfer of Control via Interrupts

Simple Interrupt Processing

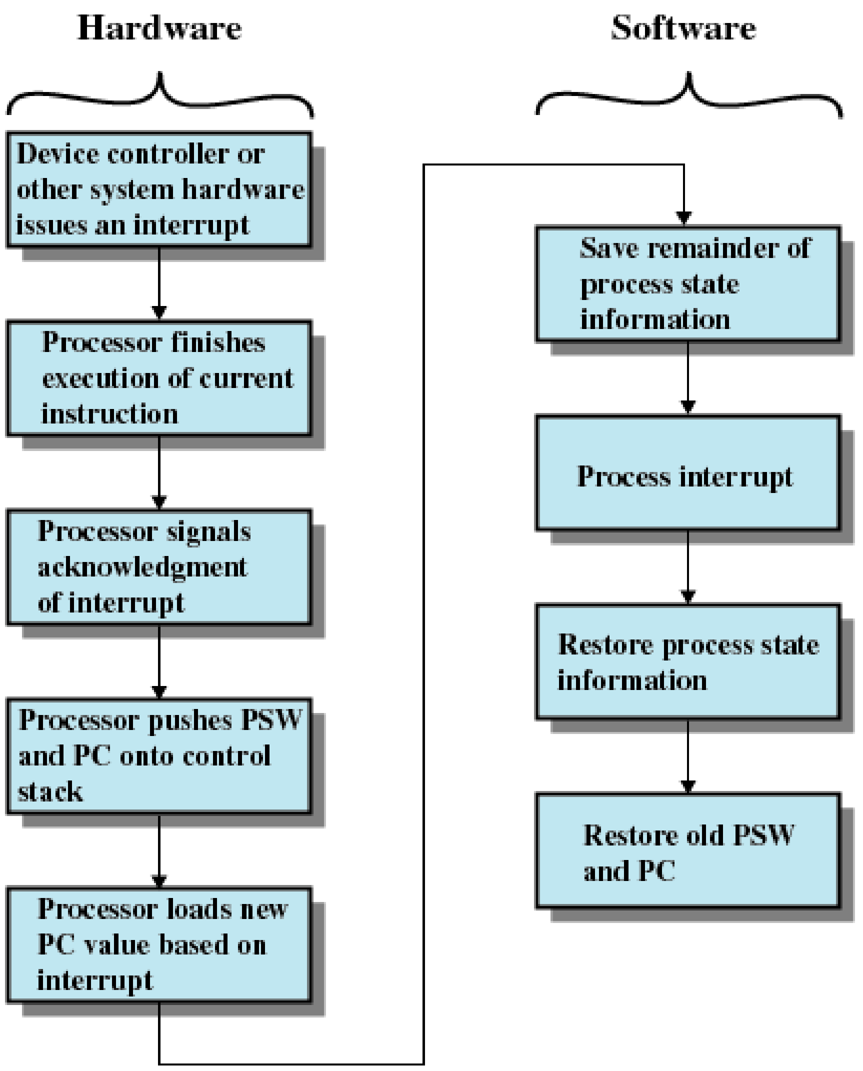
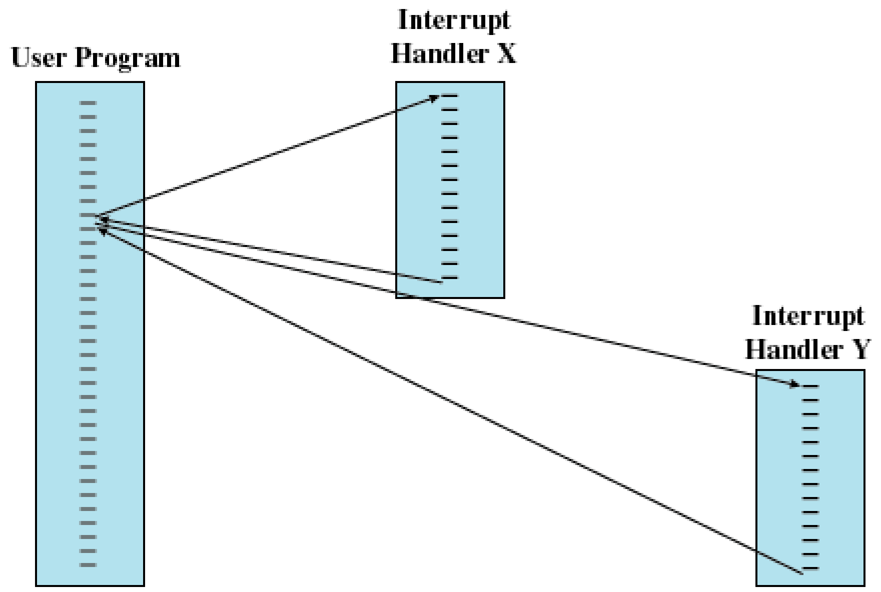

Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

Figure 1.10 Simple Interrupt Processing

Multiple Interrupts

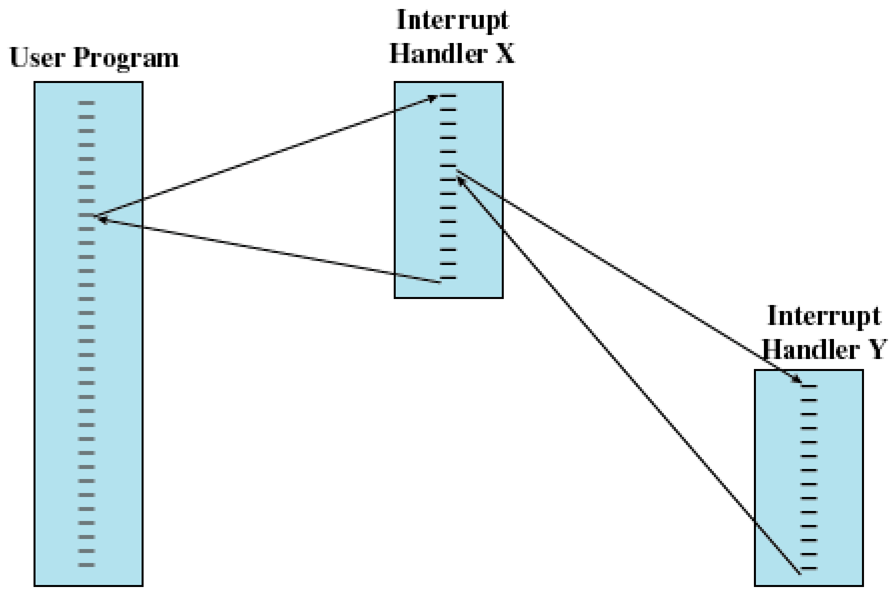
- Disable interrupts while an interrupt is being processed



(a) Sequential interrupt processing

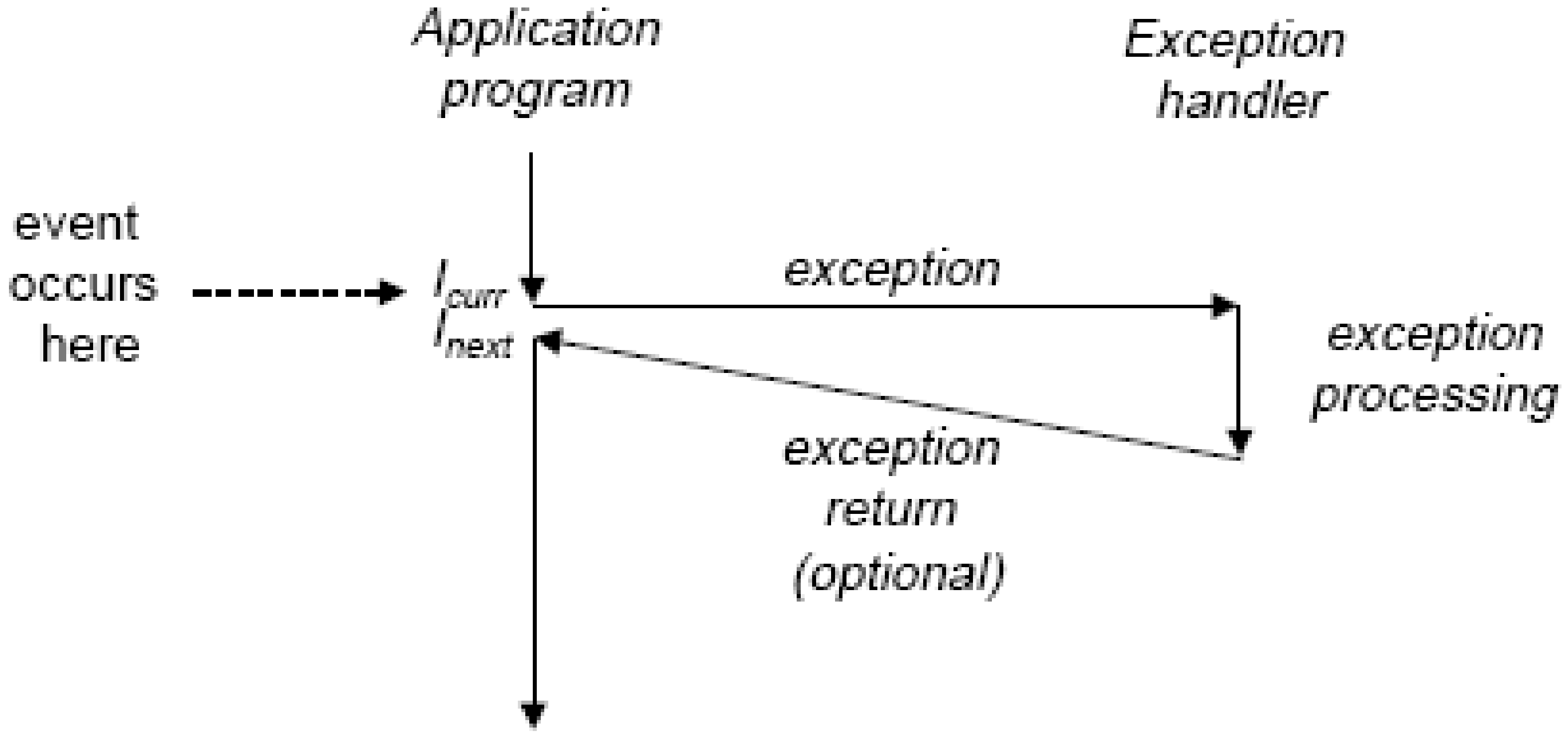
Multiple Interrupts

- Define priorities for interrupts

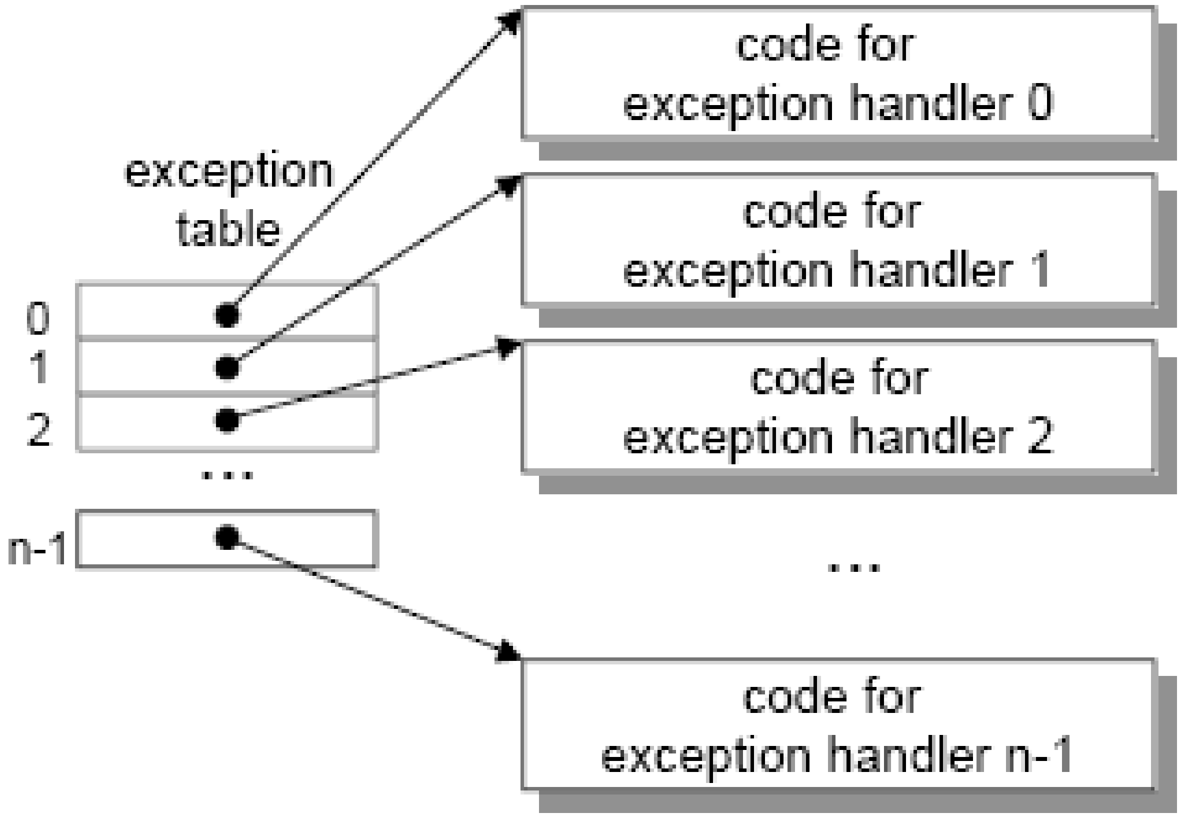


(b) Nested interrupt processing

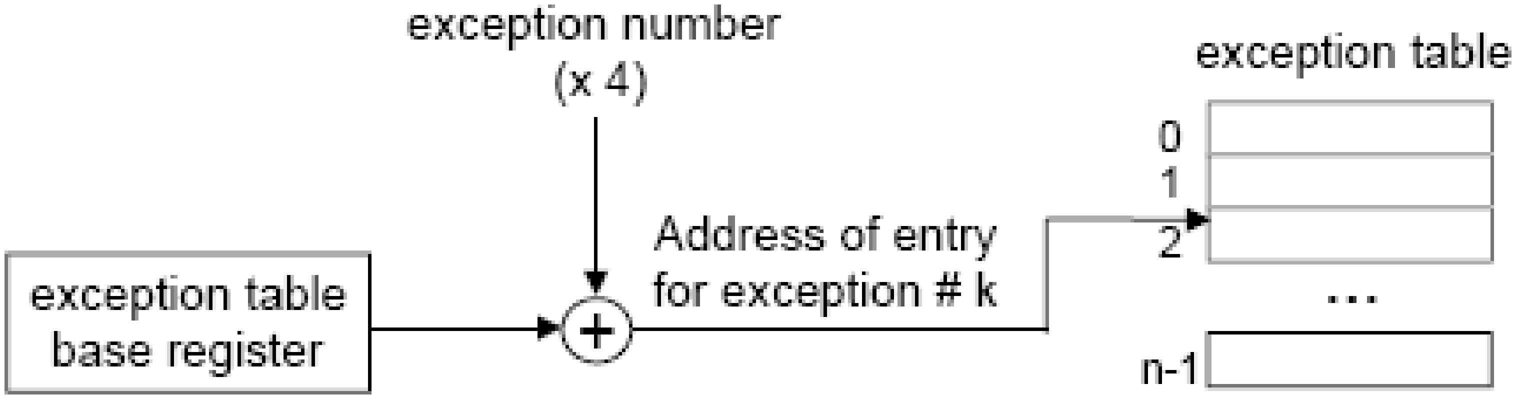
Exception: Abrupt change in the control flow in response to some change in processor state.



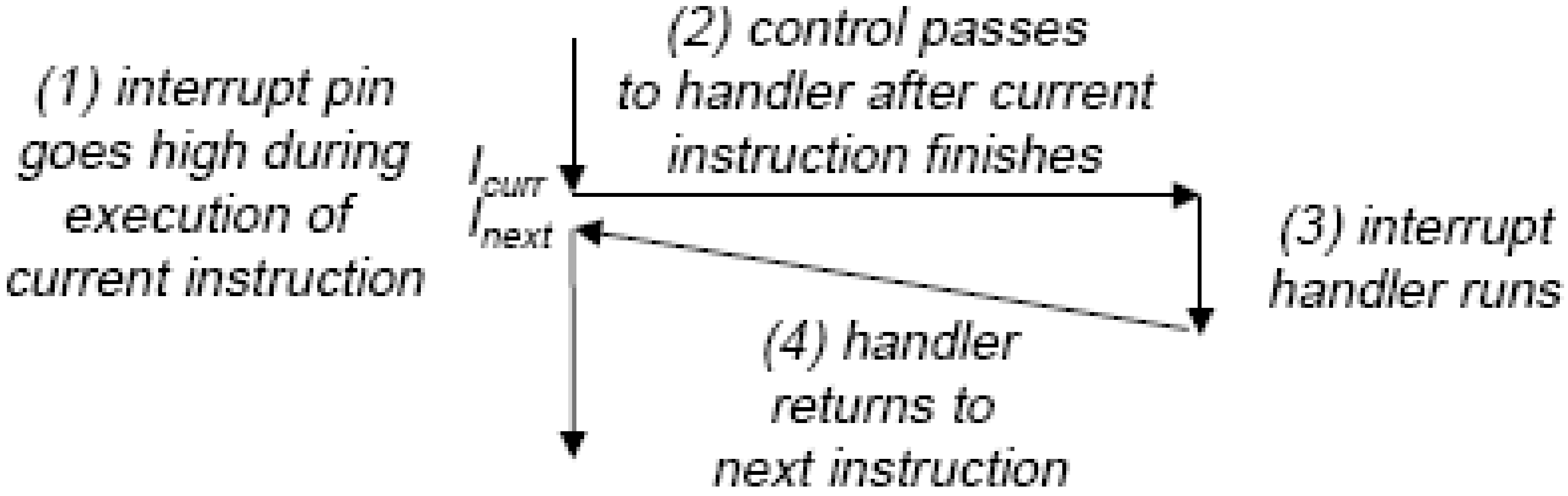
Exception table:At system boot time (when the computer is reset or powered on) the operating system allocates and initializes a jump table called an *exception table*, so that entry k contains the address of the handler for exception k.


exception table
0
1
2
...
n-1
code for exception handler 0
code for exception handler 1
code for exception handler 2
...
code for exception handler n-1

The exception number is an index into the exception table, whose starting address is contained in a special CPU register called the *exception table base register*.



Interrupts **occur** ***asynchronously*** **as a result of signals from I/O devices that are external to the processor.**
Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction.
Exception handlers for hardware interrupts are often called ***interrupt handlers*****.**

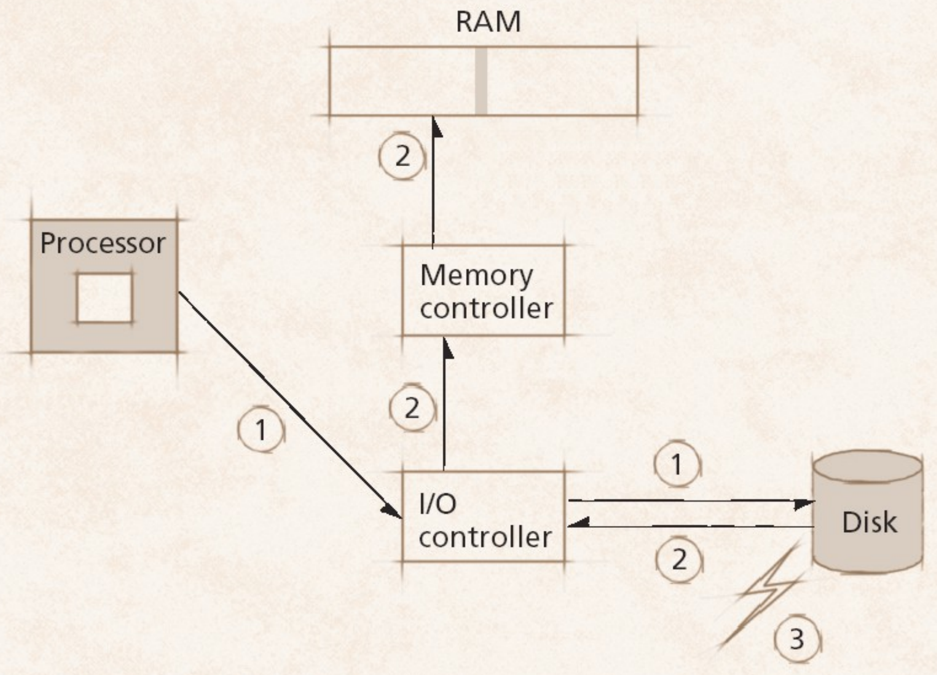

(1) interrupt pin goes high during execution of current instruction
(2) control passes to handler after current instruction finishes
I_{curr}
I_{next}
(3) interrupt handler runs
(4) handler returns to next instruction

Direct Memory Access (DMA)

- **DMA improves data transfer between memory and I/O devices**
  - Devices and controllers transfer data to and from main memory directly
  - Processor is free to execute software instructions
  - DMA channel uses an I/O controller to manage data transfer
    - Notifies processor when I/O operation is complete
  - Improves performance in systems that perform large numbers of I/O operations (e.g., mainframes and servers)

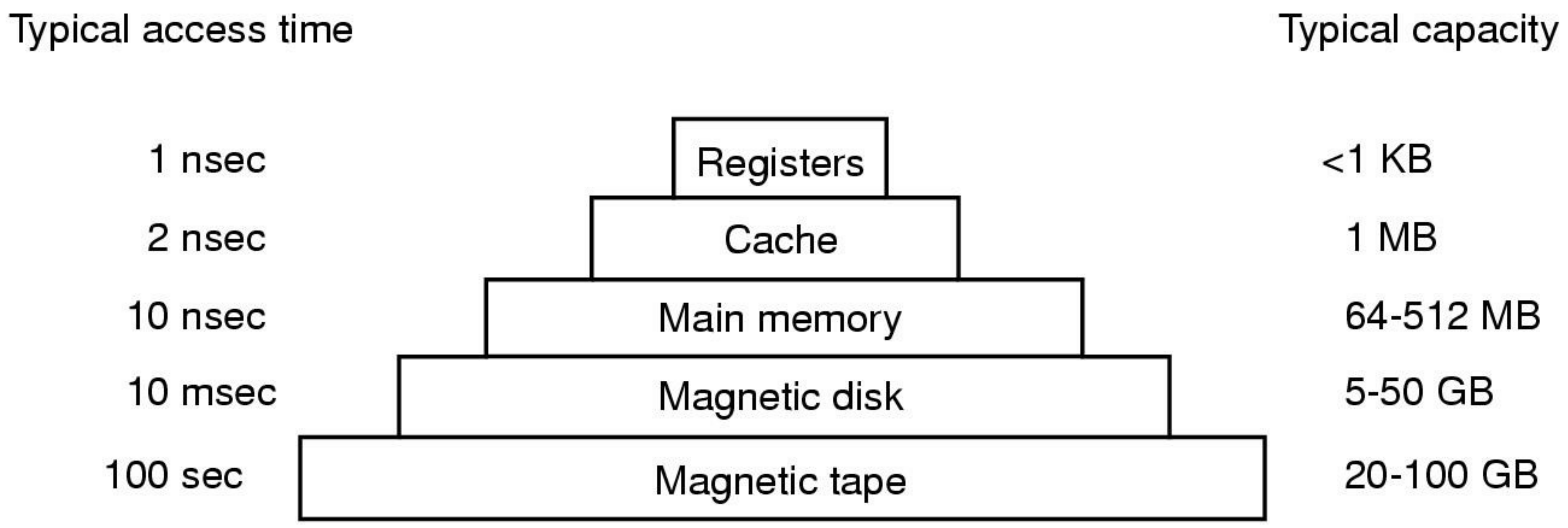
Direct Memory Access (DMA)

Direct memory access (DMA).



1. A processor sends an I/O request to the I/O controller, which sends the request to the disk. The processor continues executing instructions.
2. The disk sends data to the I/O controller; the data is placed at the memory address specified by the DMA command.
3. The disk sends an interrupt to the processor to indicate that the I/O is done.

Memory-Storage Hierarchy



Operating System Structure

- **Multiprogramming needed for efficiency**
  - **Single user cannot keep CPU and I/O devices busy at all times**
  - **Multiprogramming organizes jobs (code and data) so CPU always has one to execute**
  - **A subset of total jobs in system is kept in memory**
  - **One job selected and run via job scheduling**
  - **When it has to wait (for I/O for example), OS switches to another job**

Operating System Structure

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

Operating System Structure

- **Real-time systems** rigid time requirements are placed, often used as a control device for a dedicated application.

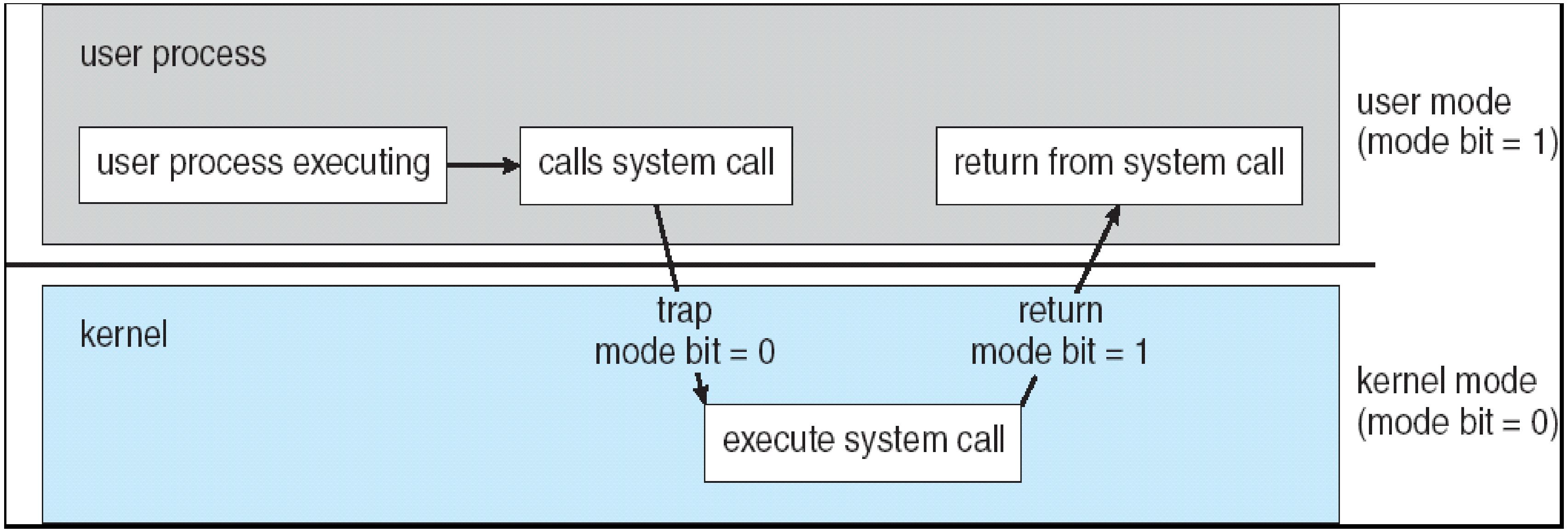
- Two flavors:
  - **Hard real-time system**: guarantees task to be completed in time. No virtual memory. Hence time sharing is not possible.
  - **Soft real-time system**: critical real-time task gets priority over other tasks, delays are bounded.

Operating-System Operations

- **Dual-mode operation allows OS to protect itself and other system components**
  - **User mode and kernel mode**
  - **Mode bit provided by hardware**
    - **provides ability to distinguish when system is running user code or kernel code**
    - **some instructions designated as privileged, only executable in kernel mode**
    - **system call changes mode to kernel, return from call resets it to user**

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)