CSCI 340 Operating Systems

Chapter 1: Introduction

Stewart Weiss

Table of Contents

Table of Contents

Table of Contents

Table of Contents

What Is This Course About?

This is an overview of computer operating systems. It is not an in-depth study of them.

The major topics that it covers are

- What operating systems do
- How computer systems are organized and structured, including the software-hardware interface
- How operating systems are structured
- Processes and their relationship to operating systems: process management, concurrency and parallelism, process synchronization, and deadlocks
- The memory hierarchy and its relationship to operating systems, including the operating system's role in managing memory
- Secondary storage and I/O devices and their relationship with operating systems
- A bit of the history of operating systems and how open source operating systems have influenced their development.

These slides are intended to accompany reading of the textbook, *Operating System Concepts, 10th Edition* by Silberschatz, Gagne, and Galvin.

This Chapter's Objectives

The goals of this chapter are to make sure that you understand

- **what** operating systems actually do and what they do not do,
- **how** operating systems do what they do,
- how operating systems **are structured**, and
- enough about the underlying hardware to give context to the tasks an operating system must perform, and how it interacts with that hardware layer, including
    - general organization of a computer system, including multiprocessor systems,
    - interrupts and their relationship to operating system functionality,
    - dual mode operation, and
    - storage and memory technologies and their impact on operating systems.

Computer System Components

In general, a **system** is a cohesive collection of interrelated and interdependent parts.

A **computer system** includes the **hardware** and the **software** that together make the aggregate usable by **users**, which include not just people, but machines or other **"things"**.

Computer System Components

In general, a **system** is a cohesive collection of interrelated and interdependent parts.

A **computer system** includes the **hardware** and the **software** that together make the aggregate usable by **users**, which include not just people, but machines or other **"things"**.

- **Hardware** includes processors, I/O devices, all types of memory and secondary storage.

Computer System Components

In general, a system is a cohesive collection of interrelated and interdependent parts.

A computer system includes the hardware and the software that together make the aggregate usable by users, which include not just people, but machines or other "things".

- Hardware includes processors, I/O devices, all types of memory and secondary storage.
- Software includes the operating system and the software that uses it, which is called the application layer.

Computer System Components

In general, a system is a cohesive collection of interrelated and interdependent parts.

A computer system includes the hardware and the software that together make the aggregate usable by users, which include not just people, but machines or other "things".

- Hardware includes processors, I/O devices, all types of memory and secondary storage.
- Software includes the operating system and the software that uses it, which is called the application layer.
- The operating system is the software that interacts directly with the hardware; applications do not.

Computer System Components

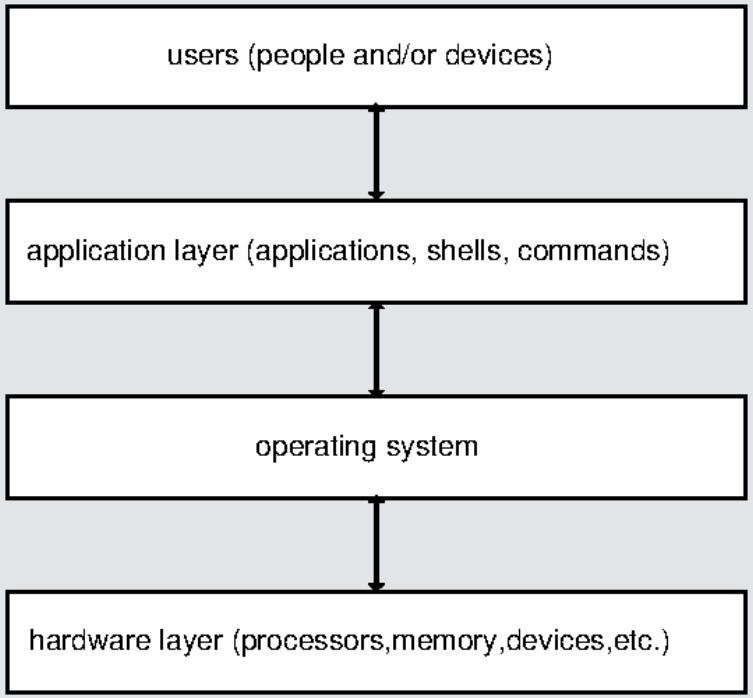
In general, a **system** is a cohesive collection of interrelated and interdependent parts.

A **computer system** includes the **hardware** and the **software** that together make the aggregate usable by **users**, which include not just people, but machines or other **"things"**.

- **Hardware** includes processors, I/O devices, all types of memory and secondary storage.
- **Software** includes the **operating system** and the software that uses it, which is called the **application layer**.
- The **operating system** is the software that interacts directly with the hardware; applications do not.
- **The application layer** includes software development tools, web browsers, all types of media editors and viewers, shells, commands, etc. ***User data is part of the application layer*** .

Layered Structure of Computer System

The components of a computer system form a sequence of layers:



Only adjacent layers have interactions with each other. For example,

- applications do not directly communicate with or control hardware, and
- users only interact with the application layer (which includes shells like `bash` in Linux.)

Operating System Requirements

The layered structure of a computer system implies that an operating system has three important responsibilities:

1. The operating system alone must **control the hardware resources**.

Operating System Requirements

The layered structure of a computer system implies that an operating system has three important responsibilities:

1. The operating system alone must **control the hardware resources**.
2. The operating system alone must enable and **control the execution** of all other software on the computer.

Operating System Requirements

The layered structure of a computer system implies that an operating system has three important responsibilities:

1. The operating system alone must **control the hardware resources**.
2. The operating system alone must enable and **control the execution** of all other software on the computer.
3. The operating system must give users the ability to **develop, run, and manage applications and their data**.

Operating System Requirements

The layered structure of a computer system implies that an operating system has three important responsibilities:

1. The operating system alone must control the hardware resources.
2. The operating system alone must enable and control the execution of all other software on the computer.
3. The operating system must give users the ability to develop, run, and manage applications and their data.

But these are not the only things for which operating systems are responsible.

Other Considerations

There are other considerations that affect what else operating systems is required to do.

Other Considerations

There are other considerations that affect what else operating systems is required to do.

There are requirements coming from **what users want**:

Other Considerations

There are other considerations that affect what else operating systems is required to do.

There are requirements coming from **what users want**:

- Users want their time on the computer to be spent **efficiently**.
- Users want their data to be **secure**.
- Users want using the computer to be as **convenient** and easy as possible.
- Users may want to be able to **share** data and applications **selectively with other users**.

Other Considerations

There are other considerations that affect what else operating systems is required to do.

There are requirements coming from **what users want**:

- Users want their time on the computer to be spent **efficiently**.
- Users want their data to be **secure**.
- Users want using the computer to be as **convenient** and easy as possible.
- Users may want to be able to **share** data and applications **selectively with other users**.

A computer, on some level, exists to produce value, directly or indirectly, for its owner. The more productive it is, the more value is created for the owner. This leads to additional requirements:

Other Considerations

There are other considerations that affect what else operating systems is required to do.

There are requirements coming from **what users want**:

- Users want their time on the computer to be spent **efficiently**.
- Users want their data to be **secure**.
- Users want using the computer to be as **convenient** and easy as possible.
- Users may want to be able to **share** data and applications **selectively with other users**.

A computer, on some level, exists to produce value, directly or indirectly, for its owner. The more productive it is, the more value is created for the owner. This leads to additional requirements:

- The computer resources should be **utilized as efficiently as possible**, maximizing the amount of work it performs per unit time.
- The computer resources should be **protected** from all possible intentional and unintentional abuses.
- It should be possible to **allocate resources as needed** among various users.

Additional Operating System Requirements

Preceding considerations lead to these additional tasks that operating systems must perform:

Additional Operating System Requirements

Preceding considerations lead to these additional tasks that operating systems must perform:

- An operating system should manage the resources of a computer system in such as way as to allow **reliable sharing of data and applications**.

Additional Operating System Requirements

Preceding considerations lead to these additional tasks that operating systems must perform:

- An operating system should manage the resources of a computer system in such as way as to allow **reliable sharing of data and applications**.
- An operating system should provide **security and protection** of all hardware and software on the computer system.

Additional Operating System Requirements

Preceding considerations lead to these additional tasks that operating systems must perform:

- An operating system should manage the resources of a computer system in such as way as to allow **reliable sharing of data and applications**.
- An operating system should provide **security and protection** of all hardware and software on the computer system.
- An operating system should provide robust **error handling and recovery**.

Additional Operating System Requirements

Preceding considerations lead to these additional tasks that operating systems must perform:

- An operating system should manage the resources of a computer system in such as way as to allow **reliable sharing of data and applications**.
- An operating system should provide **security and protection** of all hardware and software on the computer system.
- An operating system should provide robust **error handling and recovery**.
- An operating system should allocate resources fairly to users while trying to **maximize overall throughput, minimize response time** to as many users as possible, and **maximize the utilization** of all hardware resources in the system.

Operating System Services

The most common services performed by operating systems can be categorized as follows.

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security
- **I/O operations** - providing all I/O services to users and applications

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security
- **I/O operations** - providing all I/O services to users and applications
- **File systems** - creating and maintaining file systems and means of manipulating them

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security
- **I/O operations** - providing all I/O services to users and applications
- **File systems** - creating and maintaining file systems and means of manipulating them
- **Communication** - providing mechanisms that allow programs to communicate with each other

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security
- **I/O operations** - providing all I/O services to users and applications
- **File systems** - creating and maintaining file systems and means of manipulating them
- **Communication** - providing mechanisms that allow programs to communicate with each other
- **Error detection and recovery** - detecting and handling error conditions

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security
- **I/O operations** - providing all I/O services to users and applications
- **File systems** - creating and maintaining file systems and means of manipulating them
- **Communication** - providing mechanisms that allow programs to communicate with each other
- **Error detection and recovery** - detecting and handling error conditions
- **Protection and security** - preventing unauthorized or improper access to all resources, and protecting users from each other's attempts to invade privacy or corrupt data.

Operating System Services

The most common services performed by operating systems can be categorized as follows.

- **Program execution** - loading and executing programs; providing synchronization, communication, and security
- **I/O operations** - providing all I/O services to users and applications
- **File systems** - creating and maintaining file systems and means of manipulating them
- **Communication** - providing mechanisms that allow programs to communicate with each other
- **Error detection and recovery** - detecting and handling error conditions
- **Protection and security** - preventing unauthorized or improper access to all resources, and protecting users from each other's attempts to invade privacy or corrupt data.
- **Accounting** - monitoring and recording various performance and utilization metrics, for both users and the system.

Trying to Define "Operating System"

People disagree about the definition of the term "operating system."

Trying to Define "Operating System"

People disagree about the definition of the term "operating system."

- Some say it is just the program, often called the **kernel**, that is loaded into memory on start-up and remains in memory, controlling the computer, until it is shut down.

Trying to Define "Operating System"

People disagree about the definition of the term "operating system."

- Some say it is just the program, often called the kernel, that is loaded into memory on start-up and remains in memory, controlling the computer, until it is shut down.
- Others say it is the collection of programs, including the kernel, that provide services to applications and users, including all system programs. We discuss system programs shortly.

Trying to Define "Operating System"

People disagree about the definition of the term "operating system."

- Some say it is just the program, often called the **kernel**, that is loaded into memory on start-up and remains in memory, controlling the computer, until it is shut down.
- Others say it is the collection of programs, including the kernel, that provide services to applications and users, including all **system programs**. We discuss system programs shortly.

To remove all ambiguity in this course, we make the following definition: the **operating system is the kernel program and nothing more.**

System Programs

This is a fuzzy concept at best.

System programs are typically programs shipped or downloaded when the operating system is installed. For example:

- Software updaters/package managers - programs that track, download, and install updates to your operating system or other software or install new software.
- Compilers, linkers, loaders, debuggers, etc. - programs that enable you to build your own software.
- File management commands such as commands to navigate the file system, list directories, and so on.

System Programs

This is a fuzzy concept at best.

System programs are typically programs shipped or downloaded when the operating system is installed. For example:

- Software updaters/package managers - programs that track, download, and install updates to your operating system or other software or install new software.
- Compilers, linkers, loaders, debuggers, etc. - programs that enable you to build your own software.
- File management commands such as commands to navigate the file system, list directories, and so on.

On a Linux system, system programs include most of the commands that you typically type on the command line. To be more precise, these are commands that are found in the directories, `/bin`, `/usr/bin`, `/usr/sbin`, and `/sbin`.

Almost all of these programs can only be invoked as commands on the command line in a terminal window, but there are some that can (also) be invoked through a graphical user interface such as `Gnome` using menus or clickable icons.

System Programs

This is a fuzzy concept at best.

System programs are typically programs shipped or downloaded when the operating system is installed. **For example**:

- **Software updaters/package managers** - programs that track, download, and install updates to your operating system or other software or install new software.
- **Compilers, linkers, loaders, debuggers, etc.** - programs that enable you to build your own software.
- **File management commands** such as commands to navigate the file system, list directories, and so on.

On a **Linux system**, system programs include most of the commands that you typically type on the command line. To be more precise, these are commands that are found in the directories, `/bin`, `/usr/bin`, `/usr/sbin`, and `/sbin`.

Almost all of these programs can only be invoked as commands on the command line in a terminal window, but there are some that can (also) be invoked through a graphical user interface such as `Gnome` using menus or clickable icons.

Do not confuse system programs with applications. Applications are usually programs that you install. Some applications come "bundled" with the operating system for your convenience. Applications are typically installed in the directories `/usr/local/bin` or `/opt`.

System Program Activity

Take a look in the `/bin` directory on `eniac.cs.hunter.cuny.edu` to see what commands are in it:

- Login to `eniac.cs.hunter.cuny.edu`
- Type the command

  ```
  ls /bin
  ```

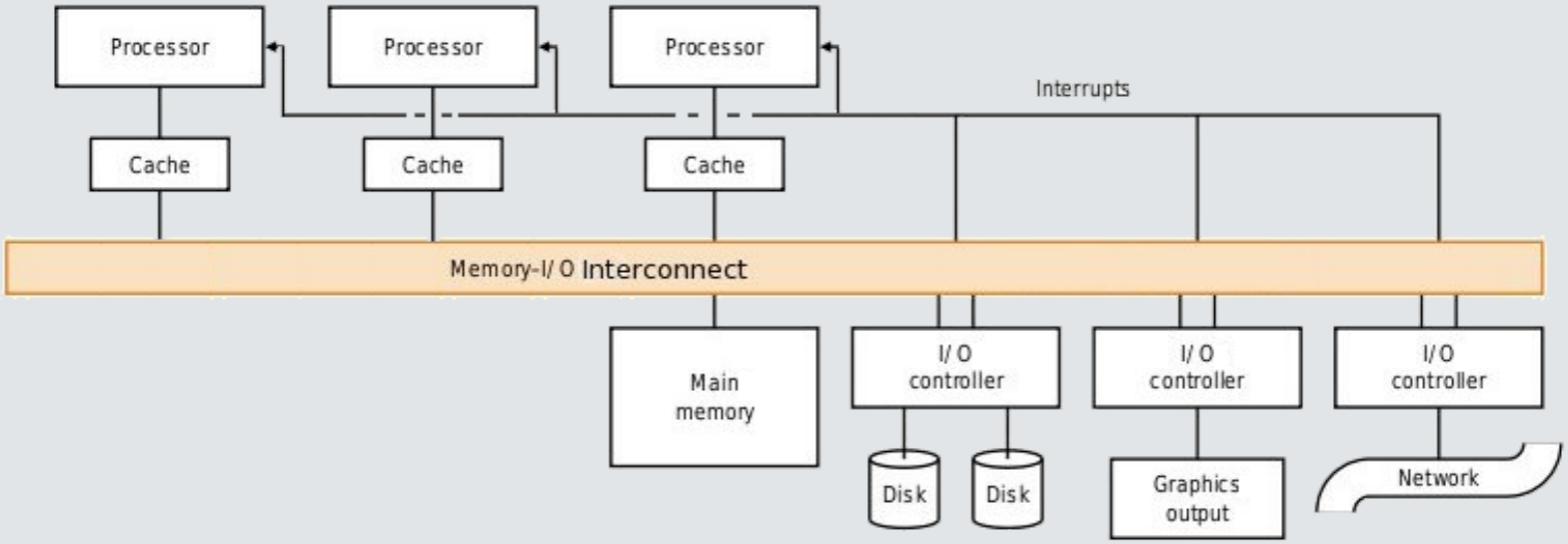
  and record three different commands that you see there. These are system programs.

Computer Organization

We examine the physical components of a computer system and how they interact with each other.

The General Structure of a Typical Computer

A modern, general-purpose computer consists of one or more **CPUs**, one or more **memory units**, and a set of **device controllers**, all of which are connected by a common **bus**, usually called the **system bus**. The figure below illustrates how these components interconnect. Notice that there is a separate line labeled "interrupts", to be discussed shortly.



Main Components

CPUs

Most modern computers contain more than one processor. When several processors are integrated into a single chip, each one is called a **core** and the machine itself is called a **multi-core processor**.

Main Components

CPUs

Most modern computers contain more than one processor. When several processors are integrated into a single chip, each one is called a **core** and the machine itself is called a **multi-core processor**.

System Bus

The **system bus** is a single communication path that connects the major components of a computer system, combining the functionality of a **data bus**, an **address bus**, and a **control bus**. A system bus typically consists of many parallel wires. The number of wires is called the **width** of the bus.

Main Components

CPUs

Most modern computers contain more than one processor. When several processors are integrated into a single chip, each one is called a **core** and the machine itself is called a **multi-core processor**.

System Bus

The **system bus** is a single communication path that connects the major components of a computer system, combining the functionality of a **data bus**, an **address bus**, and a **control bus**. A system bus typically consists of many parallel wires. The number of wires is called the **width** of the bus.

Memory

Memory units are self-contained primary storage, i.e. random-access, devices. They are usually **volatile** and are connected directly to the system bus.

Memory units are controlled by a **memory controller**, which synchronizes access to the memory from the devices that want to access it, and responds to requests to transfer data to and from memory.

Main Components (Continued)

Device Controllers

A **device controller** is a special-purpose processor that controls a specific type of device, such as a disk or a keyboard. It may be connected to multiple instances of this type of device, or just one.

It typically has a number of special-purpose registers, buffer memory, and control logic that responds to specific instructions.

A device controller is responsible for moving data between the devices that it controls and its local memory. It responds to instructions submitted to it through the bus, and can query the status of the attached devices.

Main Components (Continued)

Device Controllers

A **device controller** is a special-purpose processor that controls a specific type of device, such as a disk or a keyboard. It may be connected to multiple instances of this type of device, or just one.

It typically has a number of special-purpose registers, buffer memory, and control logic that responds to specific instructions.

A device controller is responsible for moving data between the devices that it controls and its local memory. It responds to instructions submitted to it through the bus, and can query the status of the attached devices.

Because device controllers are special-purpose microprocessors with their own unique instruction sets, the software to control them is also highly specialized and specific to each different one. For example, the code needed for an Ultra320 SCSI disk controller is very different from the code needed for a Parallel ATA disk controller.

Main Components (Continued)

Device Controllers

A **device controller** is a special-purpose processor that controls a specific type of device, such as a disk or a keyboard. It may be connected to multiple instances of this type of device, or just one.

It typically has a number of special-purpose registers, buffer memory, and control logic that responds to specific instructions.

A device controller is responsible for moving data between the devices that it controls and its local memory. It responds to instructions submitted to it through the bus, and can query the status of the attached devices.

Because device controllers are special-purpose microprocessors with their own unique instruction sets, the software to control them is also highly specialized and specific to each different one. For example, the code needed for an Ultra320 SCSI disk controller is very different from the code needed for a Parallel ATA disk controller.

This is why the code written to control each controller is separated from the rest of the operating system and placed into its own module called a **device driver**.

Device Drivers

In a modern operating system, **every device controller has an associated device driver.**

Device Drivers

In a modern operating system, **every device controller has an associated device driver.**

A **device driver** is a program or software module that can "drive" that controller in much the same way that a person can drive a car. It "knows" the controller's instruction set and its interface and can send instructions to the device to perform actions, such as to read a number of blocks of data from a disk to transfer to a memory location, or to stop or start the device.

Device Drivers

In a modern operating system, **every device controller has an associated device driver.**

A **device driver** is a program or software module that can "drive" that controller in much the same way that a person can drive a car. It "knows" the controller's instruction set and its interface and can send instructions to the device to perform actions, such as to read a number of blocks of data from a disk to transfer to a memory location, or to stop or start the device.

Most operating systems have the ability to detect which device drivers they need and integrate them into the operating system itself as separate modules.

The primary purpose of device drivers is to provide a layer of abstraction, so that the operating system programmers do not need to know the details of the device controllers to write code to perform tasks using those devices.

Device Drivers

In a modern operating system, **every device controller has an associated device driver.**

A **device driver** is a program or software module that can "drive" that controller in much the same way that a person can drive a car. It "knows" the controller's instruction set and its interface and can send instructions to the device to perform actions, such as to read a number of blocks of data from a disk to transfer to a memory location, or to stop or start the device.

Most operating systems have the ability to detect which device drivers they need and integrate them into the operating system itself as separate modules.

The primary purpose of device drivers is to provide a layer of abstraction, so that the operating system programmers do not need to know the details of the device controllers to write code to perform tasks using those devices.

For example, a serial port might only present two "public" functions, one to send data and one to receive data. A device driver that implements these functions would communicate with the particular serial port controller installed on the computer. **Different serial ports have different instruction sets and architectures**, so each serial port may have a different device driver, but each device driver hides the hardware-specific differences and presents the same software interface to the software layer above. The operating system programmer just needs to know the common software interface that the device drivers present to them.

Computer System Operations

Peripheral devices such as keyboards, mice, network interfaces, and external storage devices can operate concurrently with the processors. For example, a program might be running on a CPU while a user types on a keyboard, or moves a mouse, or while data is arriving from the local network.

Computer System Operations

Peripheral devices such as keyboards, mice, network interfaces, and external storage devices can operate concurrently with the processors. **For example**, a program might be running on a CPU while a user types on a keyboard, or moves a mouse, or while data is arriving from the local network.

These concurrent activities usually involve moving data to or from memory, and/or may require that the CPU take some specific action in response to specific events.

Example. When the user types a "Control-C" on the keyboard while a terminal window has focus and a program is running in the terminal's command line (the shell), that program is typically terminated. Somehow, that Control-C must be detected and the fact of its occurrence transmitted to a program that can terminate the running program. A lot has to happen to make this work.

Computer System Operations

Peripheral devices such as keyboards, mice, network interfaces, and external storage devices can operate concurrently with the processors. **For example**, a program might be running on a CPU while a user types on a keyboard, or moves a mouse, or while data is arriving from the local network.

These concurrent activities usually involve moving data to or from memory, and/or may require that the CPU take some specific action in response to specific events.

Example. When the user types a "Control-C" on the keyboard while a terminal window has focus and a program is running in the terminal's command line (the shell), that program is typically terminated. Somehow, that Control-C must be detected and the fact of its occurrence transmitted to a program that can terminate the running program. A lot has to happen to make this work.

- What part of this is done by hardware? Which hardware?
- What part of this is done by software? Which software?

Computer System Operations

Peripheral devices such as keyboards, mice, network interfaces, and external storage devices can operate concurrently with the processors. For example, a program might be running on a CPU while a user types on a keyboard, or moves a mouse, or while data is arriving from the local network.

These concurrent activities usually involve moving data to or from memory, and/or may require that the CPU take some specific action in response to specific events.

Example. When the user types a "Control-C" on the keyboard while a terminal window has focus and a program is running in the terminal's command line (the shell), that program is typically terminated. Somehow, that Control-C must be detected and the fact of its occurrence transmitted to a program that can terminate the running program. A lot has to happen to make this work.

- What part of this is done by hardware? Which hardware?
- What part of this is done by software? Which software?

The key to all of this is the use of interrupts.

All modern operating systems are interrupt-driven; after the boot completes, they run only as a result of interrupts.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.
- The device controller responds by examining the contents of these registers and it determines (1) that it is a read operation and (2) which data must be read.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.
- The device controller responds by examining the contents of these registers and it determines (1) that it is a read operation and (2) which data must be read.
- The controller starts the transfer of the data from the device to its local buffer.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.
- The device controller responds by examining the contents of these registers and it determines (1) that it is a read operation and (2) which data must be read.
- The controller starts the transfer of the data from the device to its local buffer.
- When the transfer of data is complete, **the device controller informs the device driver that it has finished its operation**.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.
- The device controller responds by examining the contents of these registers and it determines (1) that it is a read operation and (2) which data must be read.
- The controller starts the transfer of the data from the device to its local buffer.
- When the transfer of data is complete, **the device controller informs the device driver that it has finished its operation**.
- The device driver runs again. It is responsible for the remainder of the work, which includes notifying the operating system that the I/O is complete and transferring the data to an appropriate place in memory.

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.
- The device controller responds by examining the contents of these registers and it determines (1) that it is a read operation and (2) which data must be read.
- The controller starts the transfer of the data from the device to its local buffer.
- When the transfer of data is complete, **the device controller informs the device driver that it has finished its operation**.
- The device driver runs again. It is responsible for the remainder of the work, which includes notifying the operating system that the I/O is complete and transferring the data to an appropriate place in memory.

How does the controller inform the device driver that it has finished its operation?

Typical I/O Read Operation

To illustrate, we discuss what happens when a running program issues a request to read data from a hard disk. The program's **read request** causes a device driver to run. We ignore the steps leading to this for now.

- The device driver loads the appropriate registers in the device controller in order to start the read operation. When it has done this, some other program is chosen to run on the CPU.
- The device controller responds by examining the contents of these registers and it determines (1) that it is a read operation and (2) which data must be read.
- The controller starts the transfer of the data from the device to its local buffer.
- When the transfer of data is complete, **the device controller informs the device driver that it has finished its operation**.
- The device driver runs again. It is responsible for the remainder of the work, which includes notifying the operating system that the I/O is complete and transferring the data to an appropriate place in memory.

How does the controller inform the device driver that it has finished its operation?

By issuing an **interrupt request**.

Interrupt Overview

What are interrupts and interrupt requests?

An **interrupt** is a temporary break in the continuity of the process running on the processor in order to respond to some condition needing attention.

When an interrupt takes place, the state of the processor is saved, and special code runs that **handles** the specific condition requiring attention. This code is called either an **interrupt service routine** (**ISR**) or an **interrupt handler**.

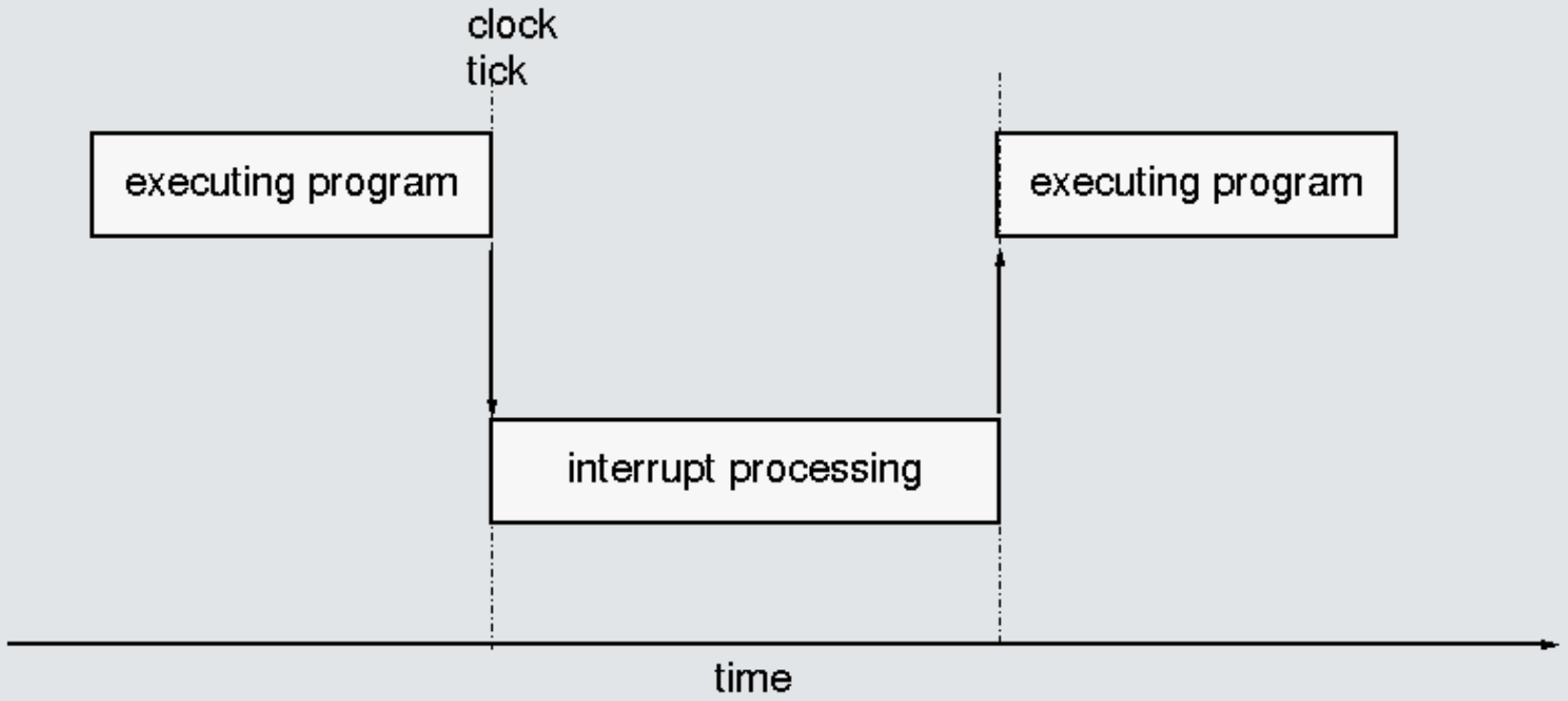
A device needing attention can request an interrupt by sending a signal to the processor. This signal is called an **interrupt request**. The acronym **IRQ** is short for interrupt request.

The distinction between interrupts, interrupt requests, and the events requiring attention is often blurred, and people may call any of them an **interrupt** for short. Thus, if a hard disk completed a read operation and issued an IRQ as a result of it, we might say it issued an interrupt, or it raised an interrupt, and we might also say the read event itself caused an interrupt.

Interrupts Visualized

Every computer has a clock or timer that causes interrupts at regular intervals. Every 1 millisecond or so, the clock sends an interrupt request so that the system can keep track of the time or do maintenance.

The running program is interrupted and a routine runs that increments the count of ticks. When it finishes, the interrupted program resumes execution, as shown below.



Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).
4. It loads the PC with the starting address of the interrupt service routine for that device.

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).
4. It loads the PC with the starting address of the interrupt service routine for that device.
5. The contents of the remaining registers are saved in an appropriate place.

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).
4. It loads the PC with the starting address of the interrupt service routine for that device.
5. The contents of the remaining registers are saved in an appropriate place.
6. The interrupt service routine (ISR) runs.

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).
4. It loads the PC with the starting address of the interrupt service routine for that device.
5. The contents of the remaining registers are saved in an appropriate place.
6. The interrupt service routine (ISR) runs.
7. When the ISR finishes, the saved registers are restored and the PC is loaded with the saved value; the interrupted computation is resumed.

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).
4. It loads the PC with the starting address of the interrupt service routine for that device.
5. The contents of the remaining registers are saved in an appropriate place.
6. The interrupt service routine (ISR) runs.
7. When the ISR finishes, the saved registers are restored and the PC is loaded with the saved value; the interrupted computation is resumed.

Step 4 suggests that the CPU knows which device caused the interrupt. How does it know this?

Interrupts Step-By-Step

A simplification of what happens is as follows.

1. A device issues an interrupt request by sending a signal on the system bus.
2. The signal is received by the CPU.
3. The CPU saves the value of the program counter (PC).
4. It loads the PC with the starting address of the interrupt service routine for that device.
5. The contents of the remaining registers are saved in an appropriate place.
6. The interrupt service routine (ISR) runs.
7. When the ISR finishes, the saved registers are restored and the PC is loaded with the saved value; the interrupted computation is resumed.

Step 4 suggests that the CPU knows which device caused the interrupt. How does it know this?

In step 4, how does the system know the starting address to load ino the PC?

Finding and Running the Interrupt Service Routine

There are two different methods of determining which device caused the interrupt and where the starting address of its interrupt service routine is stored.

Finding and Running the Interrupt Service Routine

There are two different methods of determining which device caused the interrupt and where the starting address of its interrupt service routine is stored.

Method 1: Polling the Devices (Rarely Used)

An IRQ is received by the CPU but it does not indicate which device caused the interrupt.

There is a single ISR that runs when an interrupt occurs.

This ISR sends a signal to each device that amounts to the question, "did you just send an IRQ?". The first device to answer "yes" is the one that will be serviced. The ISR has code that causes a jump to the correct device driver.

Finding and Running the Interrupt Service Routine

There are two different methods of determining which device caused the interrupt and where the starting address of its interrupt service routine is stored.

Method 1: Polling the Devices (Rarely Used)

An IRQ is received by the CPU but it does not indicate which device caused the interrupt.

There is a single ISR that runs when an interrupt occurs.

This ISR sends a signal to each device that amounts to the question, "did you just send an IRQ?". The first device to answer "yes" is the one that will be serviced. The ISR has code that causes a jump to the correct device driver.

This method is called **polling** because it polls each device. **It is inefficient**.

Finding and Running the Interrupt Service Routine

There are two different methods of determining which device caused the interrupt and where the starting address of its interrupt service routine is stored.

Method 1: Polling the Devices (Rarely Used)

An IRQ is received by the CPU but it does not indicate which device caused the interrupt.

There is a single ISR that runs when an interrupt occurs.

This ISR sends a signal to each device that amounts to the question, "did you just send an IRQ?". The first device to answer "yes" is the one that will be serviced. The ISR has code that causes a jump to the correct device driver.

This method is called **polling** because it polls each device. **It is inefficient**.

Method 2: Vectored Interrupts (Almost Always Used)

Finding and Running the Interrupt Service Routine

There are two different methods of determining which device caused the interrupt and where the starting address of its interrupt service routine is stored.

Method 1: Polling the Devices (Rarely Used)

An IRQ is received by the CPU but it does not indicate which device caused the interrupt.

There is a single ISR that runs when an interrupt occurs.

This ISR sends a signal to each device that amounts to the question, "did you just send an IRQ?". The first device to answer "yes" is the one that will be serviced. The ISR has code that causes a jump to the correct device driver.

This method is called **polling** because it polls each device. **It is inefficient**.

Method 2: Vectored Interrupts (Almost Always Used)

The interrupt lines on the bus are **vectored**: each device can send its identity along with the IRQ. When the CPU receives the IRQ, it can extract the identity of the device that needs service. The identity is an integer value.

The system maintains a table, usually in low memory, that maps each integer device identity to the starting address of its ISR. This table is called the **interrupt vector** or the **interrupt vector table**. If there are many devices, sometimes the table is a linked list of ISR addresses.

Interrupt Masking

There are two types of interrupts: **maskable** and **non-maskable**.

A **maskable interrupt** is one that can be disabled temporarily. There is a register that contains a bit for each interrupt type, and that bit is used to determine whether or not to disable (i.e., mask) the interrupt. Maskable interrupts are non-critical.

A **non-maskable interrupt** is one that cannot be disabled. It must be serviced. It is not affected by the interrupt mask register. Examples are errors from memory and timer interrupts.

In some systems, disabling an interrupt means ignoring it completely. In others, it is possible to **defer** processing the interrupt if it is disabled. Deferred interrupts are saved, usually in a queue, and handled in a specific order at a future time.

Interrupt Priorities

Some interrupts have higher priorities than others: if the CPU is in the middle of servicing an interrupt request, and a higher priority request occurs, it should interrupt the current ISR. On the other hand, if an interrupt request has lower priority than the one being serviced by the CPU, it should be ignored.

Most systems have a method of **prioritizing interrupts**, either in hardware alone, or in both hardware and software.

Interrupt Priorities

Some interrupts have higher priorities than others: if the CPU is in the middle of servicing an interrupt request, and a higher priority request occurs, it should interrupt the current ISR. On the other hand, if an interrupt request has lower priority than the one being serviced by the CPU, it should be ignored.

Most systems have a method of **prioritizing interrupts**, either in hardware alone, or in both hardware and software.

Example

The **mask register** can be used to set the interrupt priority level of the processor using a left-to-right ordering of the mask bits.

- Each device has an associated priority level, and the ISR for that device runs at that priority level. The priority level is an integer corresponding to a bit position.
- **If an interrupt occurs whose bit is to the left of another one, it has higher priority**. If its mask bit is on, it is enabled, otherwise it is disabled.
- By turning off all bits to the right of a given bit, the processor masks all interrupts whose priority is lower than the given one.
- If a lower priority interrupt occurs, it is ignored. If one occurs that is at equal or higher priority, the currently running ISR is interrupted and the ISR for the new one runs.

Traps and Exceptions

In English, an **exception** is an uncommon event. In computing it is supposed to be an uncommon event.

An **exception** is a software-generated interrupt. It is caused by the execution of a software instruction. Examples of exceptions are

- floating-point errors such as divide-by-zero,
- attempts to execute invalid opcodes, and
- attempts to access memory locations outside of the process's allowed memory.

Traps and Exceptions

In English, an **exception** is an uncommon event. In computing it is supposed to be an uncommon event.

An **exception** is a software-generated interrupt. It is caused by the execution of a software instruction. Examples of exceptions are

- floating-point errors such as divide-by-zero,
- attempts to execute invalid opcodes, and
- attempts to access memory locations outside of the process's allowed memory.

The term **trap** is often used as a synonym for exception, but it is not. Traps are a specific type of exception.

- The Intel 80x86 architecture defines a trap as follows: A **trap** is a programmer-initiated (and consequently expected) transfer of control to a handler routine. (The `int` assembly instruction is a trap.)

- In the Linux kernel, a trap is a specific type of exception. There are other types of exceptions as well. On Intel chips, the `eip` register is queried to decide the type.

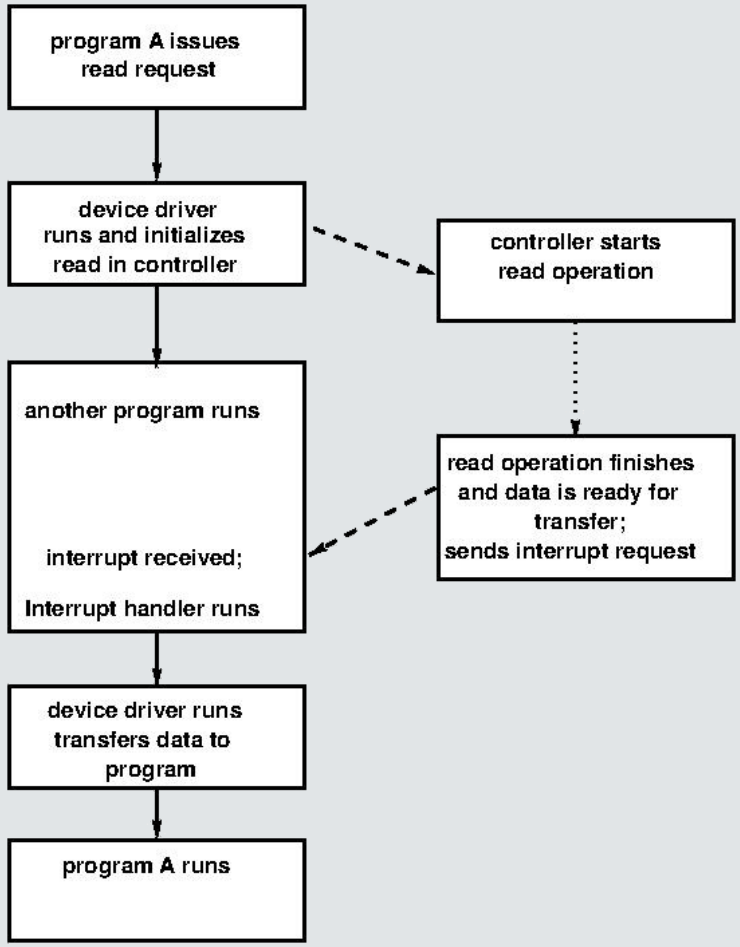
In short, traps are actual instructions coded into programs intentionally that cause an interrupt, whereas exceptions are any interrupts generated by software. We will see how traps are used in the next chapter, when we discuss **system calls**.

Interrupt-Driven I/O

Recall that modern operating systems are interrupt-driven; they only run as a result of interrupts.

All I/O occurs as a result of interrupts.

To illustrate this, in the figure to the right, we have augmented the sequence of steps described in the slide, Typical Read Operation to show how interrupts are used in the read operation.



Interrupt Activity

Take a look in the `/proc` directory on `eniac.cs.hunter.cuny.edu` to see what types of interrupts are hitting the CPUs recently:

- Login to `eniac.cs.hunter.cuny.edu`
- Navigate to the `/proc` directory:

```
cd /proc
```

- Make your terminal window as wide as possible.
- Then, using a command such as `more`, view the contents of the file named `interrupts`. There is a column for each logical CPU and a row for different interrupts.
- Which type of interrupt has occurred the most and on which CPU. Write what you find as instructed in class.

Storage

We review concepts of storage and the various types of storage used in computer systems.

Storage Concepts

Storage media and devices can be characterized in many ways, such as by their cost, reliability, capacity, and so on. We define the key properties of interest.

Storage Concepts

Storage media and devices can be characterized in many ways, such as by their cost, reliability, capacity, and so on. We define the key properties of interest.

- **Capacity** is total amount of stored information that a storage medium or device can hold. It is measured in either **bits** or **bytes**, both of which are defined in the next slide.

Storage Concepts

Storage media and devices can be characterized in many ways, such as by their cost, reliability, capacity, and so on. We define the key properties of interest.

- **Capacity** is total amount of stored information that a storage medium or device can hold. It is measured in either **bits** or **bytes**, both of which are defined in the next slide.
- **Volatility** refers to whether or not the information stored on a storage medium is retained when power is not continuously supplied to that medium. A **non-volatile** storage medium retains the information whereas a **volatile** one does not.

Storage Concepts

Storage media and devices can be characterized in many ways, such as by their cost, reliability, capacity, and so on. We define the key properties of interest.

- **Capacity** is total amount of stored information that a storage medium or device can hold. It is measured in either **bits** or **bytes**, both of which are defined in the next slide.
- **Volatility** refers to whether or not the information stored on a storage medium is retained when power is not continuously supplied to that medium. A **non-volatile** storage medium retains the information whereas a **volatile** one does not.
- **Access time** is the amount of time that it takes the medium or device to access the information at a given location. This time may vary depending on whether the access is to read information or to store it.

Storage Concepts

Storage media and devices can be characterized in many ways, such as by their cost, reliability, capacity, and so on. We define the key properties of interest.

- **Capacity** is total amount of stored information that a storage medium or device can hold. It is measured in either **bits** or **bytes**, both of which are defined in the next slide.
- **Volatility** refers to whether or not the information stored on a storage medium is retained when power is not continuously supplied to that medium. A **non-volatile** storage medium retains the information whereas a **volatile** one does not.
- **Access time** is the amount of time that it takes the medium or device to access the information at a given location. This time may vary depending on whether the access is to read information or to store it.
- **Accessibility** refers to two different types of access to the locations on a storage medium.
    - **Random access** media are those such that all locations can be accessed in about the same amount of time.
    - **Sequential access** media are those such that the information must be accessed in sequential order. This implies that the time to access a particular piece of information depends upon where that information is stored on the medium. Magnetic tape is an example of a sequential access medium.

Measuring Storage: Bits, Bytes, and Words

- A **bit** is the smallest unit of information. A bit has two possible values, which we represent as the numbers 0 and 1. The unit symbol for a bit is lowercase 'b'; *128b* means 128 bits.

Measuring Storage: Bits, Bytes, and Words

- A **bit** is the smallest unit of information. A bit has two possible values, which we represent as the numbers 0 and 1. The unit symbol for a bit is lowercase 'b'; *128b* means 128 bits.

- A **byte** is the **smallest addressable unit of storage** in a computer and consists of **eight bits**. There have been many different definitions of a byte, but over time, the eight-bit byte became the *de facto* standard. The unit symbol for a byte is uppercase 'B', e.g., *4096B* means 4096 bytes.

Measuring Storage: Bits, Bytes, and Words

- A **bit** is the smallest unit of information. A bit has two possible values, which we represent as the numbers 0 and 1. The unit symbol for a bit is lowercase 'b'; *128b* means 128 bits.

- A **byte** is the **smallest addressable unit of storage** in a computer and consists of **eight bits**. There have been many different definitions of a byte, but over time, the eight-bit byte became the *de facto* standard. The unit symbol for a byte is uppercase 'B', e.g., *4096B* means 4096 bytes.

- A **word** is the size of a given computer's **native unit of data**.

    - Typically, it is the size of a general-purpose register, the size of a machine instruction, and the size of the data chunk that is transferred to and from memory.
    - A word consists of one or more bytes. The **word length** or **word size** is the number of bits in a word.
    - A "64-bit" architecture has a word length of 64 bits or eight bytes. It has 64-bit registers and 64-bit memory addresses.

Measuring Storage: Larger Amounts

In practice, we often describe amounts of data and storage sizes that are orders of magnitude larger than bytes and words. We need to talk about thousands, millions, billions, trillions or more bytes, and so we use units whose sizes are commensurate with these magnitudes.

Below are commonly used units and their meanings.

Unit	Actual Number of Bytes	As Power of 2	Abbreviation
kilobyte	1024	2^10 bytes	1 KB
megabyte	1048576	2^20 bytes	1 MB
gigabyte	1073741824	2^30 bytes	1 GB
terabyte	1099511627776	2^40 bytes	1 TB
petabyte	1125899906842624	2^50 bytes	1 PB

Some computer manufacturers misuse these terms. For example, a hard disk vendor might say a disk has 500 gigabytes when it has 500 billion bytes.

Storage is measured in these binary-based units, but transmission rates, bandwidths, and other rates expressed as a function of bytes per unit time generally use the decimal approximations or express in bits per time. Thus, a bandwidth of 50MB/second means 50,000,000 bytes per second.

The Different Types of Storage

Modern computers have many different types of storage:

- **Primary storage** refers to storage that is directly accessible to the processor.
- **Secondary storage** refers to storage that is not directly accessible to the processor, but can be accessed through the I/O channels, and so is still considered to be **on-line**.
- **Tertiary storage** refers to storage that is not directly accessible to the processor, and cannot be accessed through the I/O channels without some intervention by a human or a mechanical system. Tertiary storage is therefore **off-line**[1].

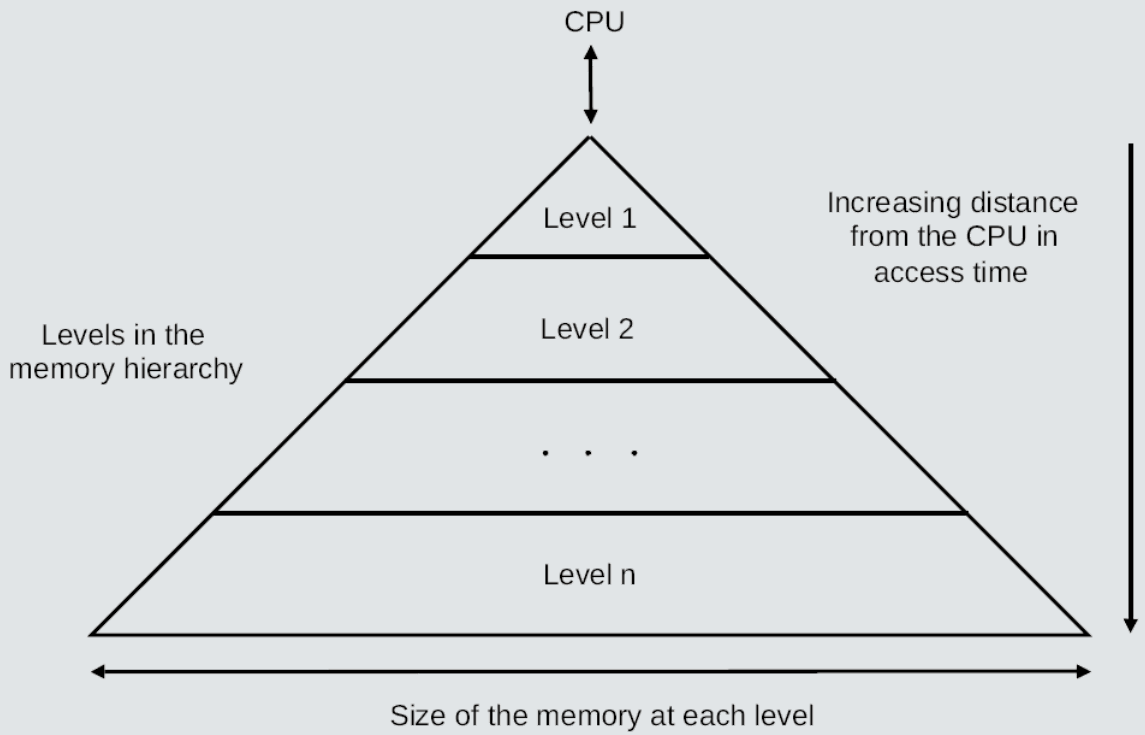
The different storage components of a general purpose computer system are as follows, listed in order of increasing access time:

Component	Type	Range of Access Times
CPU registers	primary storage	0.2 - 1.0 ns
CPU cache (levels 1, 2, and 3)	primary storage	0.5 - 2.5 ns
Random access memory	primary storage	50 - 70 ns
Solid-state drives	secondary storage	5000 – 50000 ns
Magnetic disks	secondary storage	5-20 million ns
Optical disks	tertiary storage	
Magnetic tape	tertiary storage	

1 There are tertiary storage systems that are semi-online because a robotic system can automatically attach them to the computer system.

The Memory Hierarchy

The preceding list of components forms a hierarchy in which components nearer the top are faster and have smaller capacity than those that are further away. For example, registers are above CPU caches, which are above random access memory, and these are all above magnetic disks. Tertiary storage is orders of magnitude slower than all of these.



Storage in the CPU

The CPU registers have the fastest access time of all storage in the computer system. However, their total capacity is small; a program cannot be stored in registers! In 2020, there were sixteen 64-bit general purpose registers in a typical desktop computer.

Programs are stored in main memory and data and instructions are brought into the CPU as needed.

Random Access Memory

Random Access Memory (**RAM**) is the memory in which the operating system, programs, and data reside when the computer is turned on. When a program is running, its executable image is stored in RAM. RAM is also called **primary memory**. RAM is volatile - it loses all data and programs when the machine loses power.

Random Access Memory

Random Access Memory (RAM) is the memory in which the operating system, programs, and data reside when the computer is turned on. When a program is running, its executable image is stored in RAM. RAM is also called primary memory. RAM is volatile - it loses all data and programs when the machine loses power.

- There are many different technologies for implementing RAM, but the most common is Dynamic Random Access Memory (DRAM). The alternative is Static Random Access Memory (SRAM).
- DRAM is dynamic because the information is stored in capacitors, which cannot retain a charge without being periodically refreshed with energy. DRAM's refresh cycles and the fact that reading the data destroys the capacitor contents make DRAM slower than SRAM. DRAM access times are roughly from 50 to 70ns.
- SRAM is very expensive and its capacity is orders of magnitude smaller than DRAM. Therefore it is not used for primary memory and instead is used for the various types of processor cache. SRAM access times are roughly from 0.5 to 2.5ns.

Magnetic Disk Drives and Solid-State Drives

Magnetic disks, commonly called **hard disks**, are the most common medium of secondary storage. Their capacities vary greatly, but they range from one to sixteen terabytes. Hard disk access times range from 5ms to 20ms. Expressed in ns, this is 5,000,000ns to 20,000,000ns, significantly slower than RAM of any kind.

Magnetic disks are non-volatile. They retain all information when the power is removed. They are where the file system resides, and they also provide a type of storage known as **swap** space, which is used in virtual memory systems to extend the capacity of primary memory. All operating system components are stored on the secondary storage medium, and all user programs and data.

Less common are solid-state drives, which are not mechanical and therefore last longer. They have smaller capacity and are more expensive per bit than magnetic disks. Because there are a limited number of possible writes and rewrites, they are more suited for storing mostly read-only files and data, such as the operating system software.

Tertiary Storage Media

Tertiary storage is used for backing up systems or archiving data. Sometimes optical disks are used, but their capacity is small and the ability to write and rewrite them is limited. Ordinary DVD disks can store no more than a few gigabytes and Blu Ray disks a few hundred gigabytes.

In contrast a magnetic tape can store several terabytes and many systems use robotic libraries to perform backups to tape and restores from tape. Recent advances have pushed capacity to over 500TB!

One can also purchase hard disks that are removable and external. These can be used for backing up a system. These external disks have the same range of capacities as internal disks and are just slightly slower because they are accessed through a slower bus than the internal disks.

Caches and Caching

What is **cache**?

Caches and Caching

What is cache?

In ordinary usage, a cache is a safe place to hide things. As a verb, to cache something is to hide it in a secure place.

In computer terminology, the noun "cache" generally means a fast but small storage component that is used by a device as a temporary holding area for data in order to improve performance. To cache something is to copy it into the cache.

Examples:

- A cache is used by the processor to hold frequently used data as well as instructions.
- A cache is used by hard disk drives and other secondary storage devices as a place to hold data being written to or read from the device.
- A cache is used by a web browser to store web pages downloaded from the web.

Processor Cache

Registers in the CPU are fast but there are not many of them. They cannot store frequently used data.

DRAM is vast in comparison, but it is slow to access. The processor only accesses its data by putting requests on the system bus and waiting. This slows down computations, as the processor must stall, waiting for data to be available.

SRAM is a much faster type of memory than DRAM, but because it is expensive and because it takes up more "real estate" on a chip, it cannot have the same large capacity as DRAM. However, smaller capacity SRAM can be placed on the processor chip as a compromise - it can store much more data than registers, and it is much faster than DRAM. It is used to create **processor cache**.

Processor Cache

Registers in the CPU are fast but there are not many of them. They cannot store frequently used data.

DRAM is vast in comparison, but it is slow to access. The processor only accesses its data by putting requests on the system bus and waiting. This slows down computations, as the processor must stall, waiting for data to be available.

SRAM is a much faster type of memory than DRAM, but because it is expensive and because it takes up more "real estate" on a chip, it cannot have the same large capacity as DRAM. However, smaller capacity SRAM can be placed on the processor chip as a compromise - it can store much more data than registers, and it is much faster than DRAM. It is used to create **processor cache**.

The **processor cache** is a level of the memory hierarchy between the CPU and main memory. It is is built using SRAM technology and is integrated into the CPU. It stores data and instructions that are frequently accessed, reducing accesses to the slower DRAM. When data is missing in the cache, it is copied into it from main memory.

Processor Cache

Registers in the CPU are fast but there are not many of them. They cannot store frequently used data.

DRAM is vast in comparison, but it is slow to access. The processor only accesses its data by putting requests on the system bus and waiting. This slows down computations, as the processor must stall, waiting for data to be available.

SRAM is a much faster type of memory than DRAM, but because it is expensive and because it takes up more "real estate" on a chip, it cannot have the same large capacity as DRAM. However, smaller capacity SRAM can be placed on the processor chip as a compromise - it can store much more data than registers, and it is much faster than DRAM. It is used to create **processor cache**.

The **processor cache** is a level of the memory hierarchy between the CPU and main memory. It is is built using SRAM technology and is integrated into the CPU. It stores data and instructions that are frequently accessed, reducing accesses to the slower DRAM. When data is missing in the cache, it is copied into it from main memory.

Questions such as how big the cache is, what size chunks are stored in it, how data is located in it, which data is replaced if it is full, what happens when data in the cache is changed, and so on, are the subject of a computer architecture course and are not answered here.

I/O and Direct Memory Access

Many devices are slow devices that handle small chunks of data. For example, keyboards and pointing devices deliver a byte at a time. Other devices may be required to deliver thousands of bytes in a single I/O operation.

I/O and Direct Memory Access

Many devices are slow devices that handle small chunks of data. For example, keyboards and pointing devices deliver a byte at a time. Other devices may be required to deliver thousands of bytes in a single I/O operation.

File operations typically cause the transfer of entire disk blocks, which can be anywhere from 1KB to 4KB or more. To transfer this much data using the interrupt mechanism for each byte or word of data would require a lot of CPU time and a lot of interrupts, slowing down the computer.

I/O and Direct Memory Access

Many devices are slow devices that handle small chunks of data. For example, keyboards and pointing devices deliver a byte at a time. Other devices may be required to deliver thousands of bytes in a single I/O operation.

File operations typically cause the transfer of entire disk blocks, which can be anywhere from 1KB to 4KB or more. To transfer this much data using the interrupt mechanism for each byte or word of data would require a lot of CPU time and a lot of interrupts, slowing down the computer.

Direct Memory Access (DMA) is a method of I/O that transfers data at a very high bandwidth with low overhead. It removes the processor from the operation of transferring large amounts of data to or from a device. The processor can continue to execute other instructions while the transfer takes place.

I/O and Direct Memory Access

Many devices are slow devices that handle small chunks of data. For example, keyboards and pointing devices deliver a byte at a time. Other devices may be required to deliver thousands of bytes in a single I/O operation.

File operations typically cause the transfer of entire disk blocks, which can be anywhere from 1KB to 4KB or more. To transfer this much data using the interrupt mechanism for each byte or word of data would require a lot of CPU time and a lot of interrupts, slowing down the computer.

Direct Memory Access (**DMA**) is a method of I/O that transfers data at a very high bandwidth with low overhead. It removes the processor from the operation of transferring large amounts of data to or from a device. The processor can continue to execute other instructions while the transfer takes place.

In DMA, the processor, under program control authorizes a device to take charge of the I/O transfers to memory, allowing it to be the **bus master** until the I/O is completed. A device with this capability is called a **DMA controller**.

I/O and Direct Memory Access

Many devices are slow devices that handle small chunks of data. For example, keyboards and pointing devices deliver a byte at a time. Other devices may be required to deliver thousands of bytes in a single I/O operation.

File operations typically cause the transfer of entire disk blocks, which can be anywhere from 1KB to 4KB or more. To transfer this much data using the interrupt mechanism for each byte or word of data would require a lot of CPU time and a lot of interrupts, slowing down the computer.

Direct Memory Access (**DMA**) is a method of I/O that transfers data at a very high bandwidth with low overhead. It removes the processor from the operation of transferring large amounts of data to or from a device. The processor can continue to execute other instructions while the transfer takes place.

In DMA, the processor, under program control authorizes a device to take charge of the I/O transfers to memory, allowing it to be the **bus master** until the I/O is completed. A device with this capability is called a **DMA controller**.

Many devices use DMA, including disk drive controllers, graphics cards, network cards and sound cards.

In some systems, such as those with a PCI bus, each device has its own internal DMA controller. In others, such as ISA, there is a central DMA controller.

DMA Operation

1. A program running on the CPU gives the DMA controller
    - a memory address,
    - the number of bytes to transfer,
    - a flag indicating whether it is a read or a write, and
    - the address of the I/O device and data involved in the I/O.

DMA Operation

1. A program running on the CPU gives the DMA controller
    - a memory address,
    - the number of bytes to transfer,
    - a flag indicating whether it is a read or a write, and
    - the address of the I/O device and data involved in the I/O.
2. The DMA controller becomes the **bus master** on the memory bus.

DMA Operation

1. A program running on the CPU gives the DMA controller
    - a memory address,
    - the number of bytes to transfer,
    - a flag indicating whether it is a read or a write, and
    - the address of the I/O device and data involved in the I/O.
2. The DMA controller becomes the **bus master** on the memory bus.
3. If it is an input operation, the device will then start sending data to the DMA controller, which will buffer the data, and store it in successive memory locations as it becomes available.

DMA Operation

1. A program running on the CPU gives the DMA controller
    - a memory address,
    - the number of bytes to transfer,
    - a flag indicating whether it is a read or a write, and
    - the address of the I/O device and data involved in the I/O.
2. The DMA controller becomes the **bus master** on the memory bus.
3. If it is an input operation, the device will then start sending data to the DMA controller, which will buffer the data, and store it in successive memory locations as it becomes available.
4. If it is an output operation, it buffers the data from memory and sends it to the I/O device as it becomes ready to receive it.

DMA Operation

1. A program running on the CPU gives the DMA controller
    - a memory address,
    - the number of bytes to transfer,
    - a flag indicating whether it is a read or a write, and
    - the address of the I/O device and data involved in the I/O.
2. The DMA controller becomes the **bus master** on the memory bus.
3. If it is an input operation, the device will then start sending data to the DMA controller, which will buffer the data, and store it in successive memory locations as it becomes available.
4. If it is an output operation, it buffers the data from memory and sends it to the I/O device as it becomes ready to receive it.
5. When the transfer is complete, the DMA controller relinquishes the bus and sends an interrupt to the processor.

DMA Operation

1. A program running on the CPU gives the DMA controller
    - a memory address,
    - the number of bytes to transfer,
    - a flag indicating whether it is a read or a write, and
    - the address of the I/O device and data involved in the I/O.
2. The DMA controller becomes the **bus master** on the memory bus.
3. If it is an input operation, the device will then start sending data to the DMA controller, which will buffer the data, and store it in successive memory locations as it becomes available.
4. If it is an output operation, it buffers the data from memory and sends it to the I/O device as it becomes ready to receive it.
5. When the transfer is complete, the DMA controller relinquishes the bus and sends an interrupt to the processor.

Because the DMA controller **owns the bus** during a transfer, the CPU will not be able to access memory. If the CPU or the cache controller needs to access memory, it will be delayed. There are methods of avoiding this.

Computer System Architecture

Modern computers have multiple processors. We review the different ways in which these are organized within a computer system.

Single Processor Systems

For decades, most computers have had just a single CPU, whose main components are

- an arithmetic-logic unit,
- registers, and
- a control unit.

Together these are sometimes called a CPU core.

Single-processor computers also have other special-purpose processors, such as graphics accelerators, DMA controllers, and disk controllers. These are not under the control of the operating system and they do not execute programs in general.

In 2020, most new computers containing multiple processors.

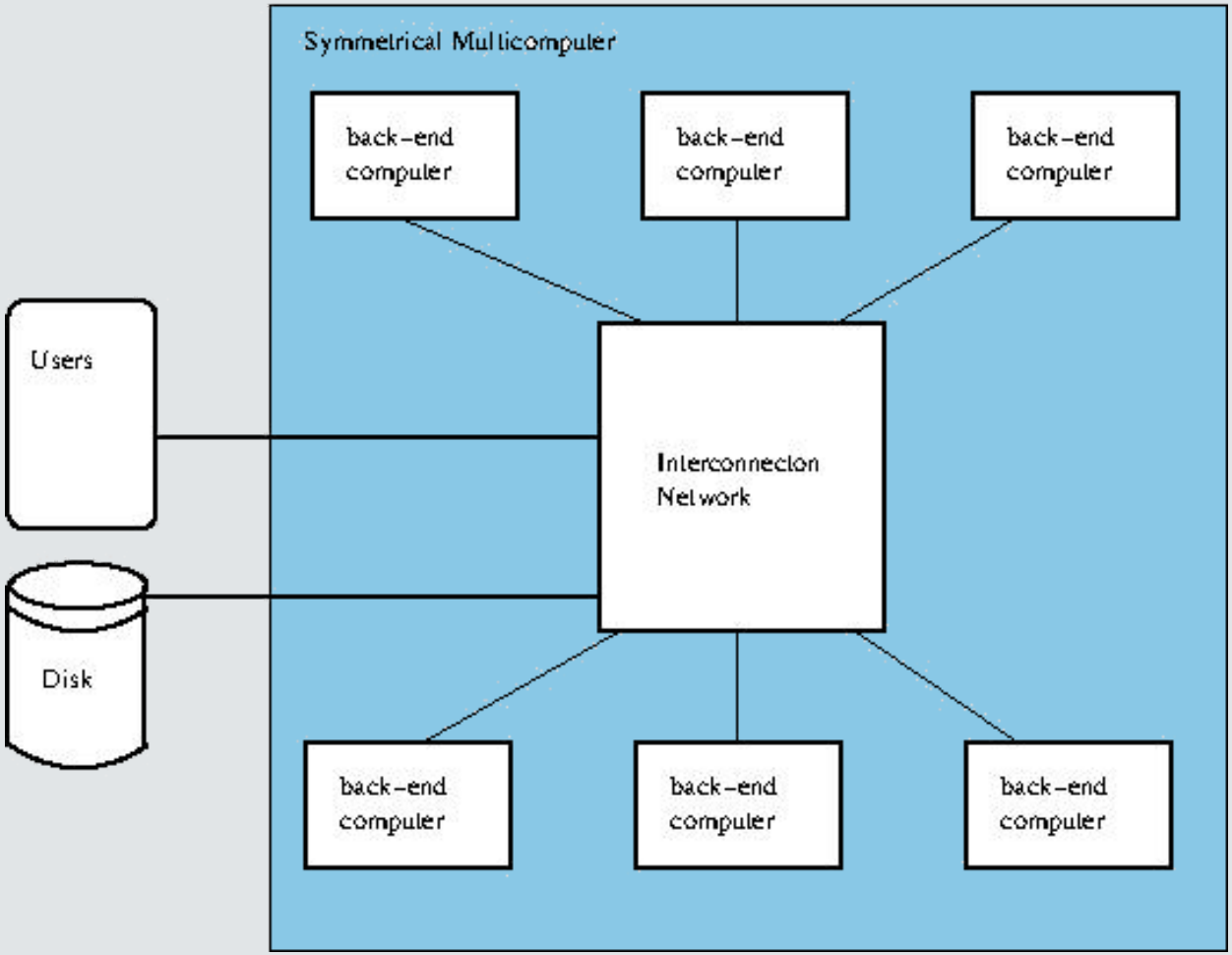
Multicomputers and Multiprocessors

A **multicomputer** is a computer with multiple CPUs **that do not share memory**. Each CPU has its own memory address space and can access only what is in this memory, which is called its **private memory**.

In contrast, a **multiprocessor** is a computer with multiple processors and a **shared memory**. In a multiprocessor, the same address generated on two different processors refers to the same memory location.

A Multicomputer Architecture

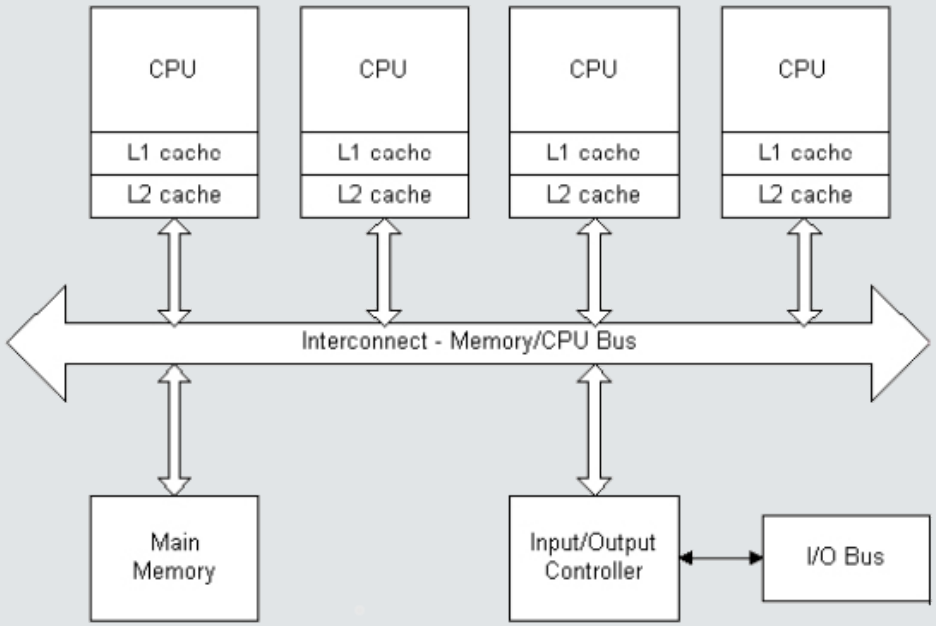
The figure below illustrates one possible way to arrange the multiple computers with respect to each other in a multicomputer.



Symmetric Multiprocessors

When the processors are identical to each other, memory is shared, and access time to memory is the same for each, the computer is called a **symmetric multiprocessor** (**SMP**) or a **uniform memory-access processor** (**UMA**).

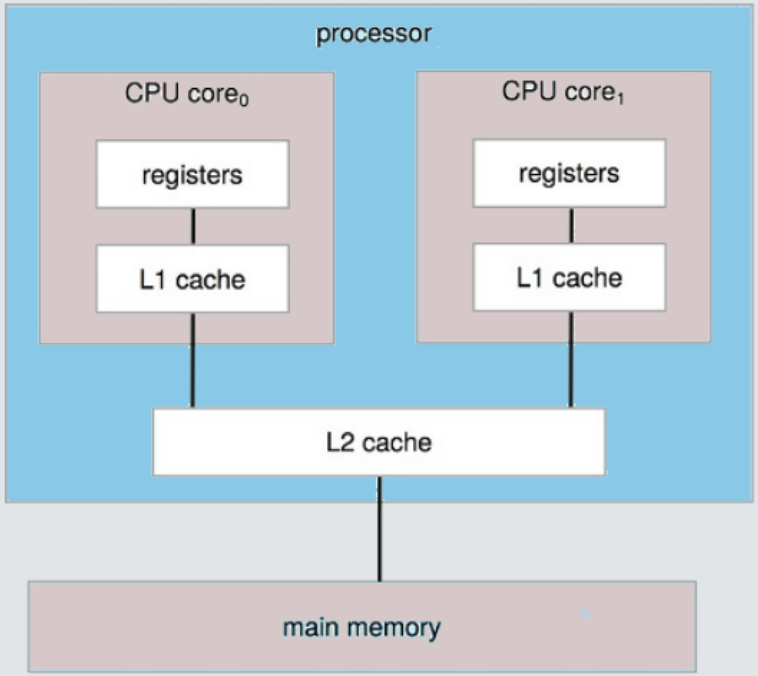
The figure below illustrates a 4-CPU symmetric multiprocessor, each containing two levels of cache.



Multi-core SMP

Some SMPs are manufactured as a single chip or **socket** containing two or more CPUs. The CPUs in this case are always called cores. The advantage of the single chip design is speed: on-chip communication is faster than between-chip communication. Also, one chip with multiple cores uses significantly less power than multiple single-core chips.

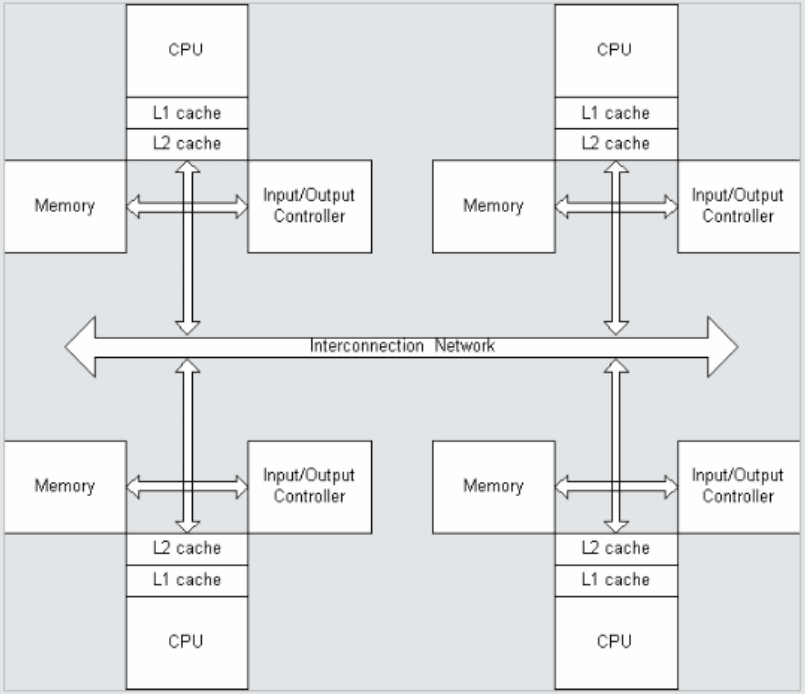
The figure below illustrates a multi-core processor with two CPUs, each containing one level of cache and sharing the second level cache.



Non-Uniform Memory Access Multiprocessors

A **non-uniform memory access** (**NUMA**) multiprocessor is one in which each CPU has a local memory that can be accessed through a fast local bus. The CPUs are connected by a shared interconnection network that enables all CPUs to share one memory address space.

Memory access is non-uniform because when a CPU accesses its local memory, it is fast and contention-free, but when it accesses the local memory of another CPU, it is slower because the access is through the shared interconnection network.



Clustered Systems

A **clustered computer system** is a collection of independent computers that

- are connected by a local area network (LAN),
- share a common secondary storage device, and
- are integrated in such a way that a single computational job may be distributed among them.

The computers are usually called **nodes**.

Clusters can be used to provide **high availability** so that if one node goes down, the system can continue without failing.

Alternatively, when they are used to run a single job, they are running a **parallel computation**.

Asymmetric clusters have one node in **hot-standby mode**. The configuration is very much like the symmetric multicomputer shown in Multicomputers except that each back-end computer is a separate node.

Symmetric clusters have multiple nodes running applications, monitoring each other are for high-performance computing (HPC)

Program Execution

We look at the ways in which the operating system supports program execution.

System Start-Up: Stage 1

Before the operating system can do any work, the computer must be started and initialized. This process is known as **bootstrapping** or simply **booting** the computer[1].

1 "To pull yourself up by your bootstraps" is the phrase people use to mean, "to elevate yourself without any outside help." In the early days, starting a computer was like magic- a program needed to be loaded into memory and then run, but it needed to put itself there first, so the program needed to get itself into memory and run itself without outside help. People likened this to bootstrapping.

System Start-Up: Stage 1

Before the operating system can do any work, the computer must be started and initialized. This process is known as **bootstrapping** or simply **booting** the computer[1].

- When power is turned on, a small program stored in the computer's **Read-Only-Memory** (**ROM**) is executed. This code on many computers was known as the **BIOS**, but it is being replaced by newer software known as **UEFI** (**Unified Extensible Firmware Interface**).

1 "To pull yourself up by your bootstraps" is the phrase people use to mean, "to elevate yourself without any outside help." In the early days, starting a computer was like magic- a program needed to be loaded into memory and then run, but it needed to put itself there first, so the program needed to get itself into memory and run itself without outside help. People likened this to bootstrapping.

System Start-Up: Stage 1

Before the operating system can do any work, the computer must be started and initialized. This process is known as **bootstrapping** or simply **booting** the computer[1].

- When power is turned on, a small program stored in the computer's **Read-Only-Memory** (**ROM**) is executed. This code on many computers was known as the **BIOS**, but it is being replaced by newer software known as **UEFI** (**Unified Extensible Firmware Interface**).

- It initializes and checks all of the hardware, performing system checks such as memory and disk checks, and then loads into memory a program stored on the hard disk in a fixed location. This program is the second stage in the boot process.

1 "To pull yourself up by your bootstraps" is the phrase people use to mean, "to elevate yourself without any outside help." In the early days, starting a computer was like magic- a program needed to be loaded into memory and then run, but it needed to put itself there first, so the program needed to get itself into memory and run itself without outside help. People likened this to bootstrapping.

System Start-Up: Stage 1

Before the operating system can do any work, the computer must be started and initialized. This process is known as **bootstrapping** or simply **booting** the computer[1].

- When power is turned on, a small program stored in the computer's **Read-Only-Memory** (**ROM**) is executed. This code on many computers was known as the **BIOS**, but it is being replaced by newer software known as **UEFI** (**Unified Extensible Firmware Interface**).
- It initializes and checks all of the hardware, performing system checks such as memory and disk checks, and then loads into memory a program stored on the hard disk in a fixed location. This program is the second stage in the boot process.
- The second-stage program runs and loads the operating system kernel into memory, performing configuration, and transfers control to the kernel.

1 "To pull yourself up by your bootstraps" is the phrase people use to mean, "to elevate yourself without any outside help." In the early days, starting a computer was like magic- a program needed to be loaded into memory and then run, but it needed to put itself there first, so the program needed to get itself into memory and run itself without outside help. People likened this to bootstrapping.

System Start-Up: Stage 2

The last stage in booting is performed by the operating system kernel.

System Start-Up: Stage 2

The last stage in booting is performed by the operating system kernel.

- The operating system does more configuration, and then starts up system services.

System Start-Up: Stage 2

The last stage in booting is performed by the operating system kernel.

- The operating system does more configuration, and then starts up system services.
- Services are provided by programs known as **system daemons**. Examples of daemons in Unix are programs such as **sshd**, which listens for ssh connections, **logind**, which listens for login attempts on the console, and **systemd**, which brings up and maintains all services for the computer users.

System Start-Up: Stage 2

The last stage in booting is performed by the operating system kernel.

- The operating system does more configuration, and then starts up system services.
- Services are provided by programs known as **system daemons**. Examples of daemons in Unix are programs such as **sshd**, which listens for ssh connections, **logind**, which listens for login attempts on the console, and **systemd**, which brings up and maintains all services for the computer users.
- After all daemons are started, the kernel then waits for a user to login, or it runs whatever applications or system programs have been specified as start-up programs. When this state is reached, booting is complete.

Origins of Multiprogramming

Early computer systems were **single-user systems**. The operating system would control execution of one program at a time, initializing it, letting it run to completion, and then starting the next. **One program was in memory at any time**.

Origins of Multiprogramming

Early computer systems were **single-user systems**. The operating system would control execution of one program at a time, initializing it, letting it run to completion, and then starting the next. **One program was in memory at any time**.

Why did this change?

Origins of Multiprogramming

Early computer systems were **single-user systems**. The operating system would control execution of one program at a time, initializing it, letting it run to completion, and then starting the next. **One program was in memory at any time**.

Why did this change?

When a program makes a request for I/O, the device cannot respond immediately; it takes time to perform I/O. The program must wait until the I/O operation is completed.

Origins of Multiprogramming

Early computer systems were **single-user systems**. The operating system would control execution of one program at a time, initializing it, letting it run to completion, and then starting the next. **One program was in memory at any time**.

Why did this change?

When a program makes a request for I/O, the device cannot respond immediately; it takes time to perform I/O. The program must wait until the I/O operation is completed.

It **idles in the CPU**, periodically checking whether the I/O is complete, or waiting for the device to send an interrupt. This is a waste of the CPU.

Origins of Multiprogramming

Early computer systems were **single-user systems**. The operating system would control execution of one program at a time, initializing it, letting it run to completion, and then starting the next. **One program was in memory at any time**.

Why did this change?

When a program makes a request for I/O, the device cannot respond immediately; it takes time to perform I/O. The program must wait until the I/O operation is completed.

It **idles in the CPU**, periodically checking whether the I/O is complete, or waiting for the device to send an interrupt. This is a waste of the CPU.

It would be a better use of the CPU if another program could be run while the first was waiting for its I/O to complete.

Multiprogramming

By keeping more than one program in memory at a time, the operating system can switch among the memory-resident programs so that the CPU always has a program that is ready to run: when a program makes a request that makes it wait, the operating system can switch to another runnable program. In this way it keeps the CPU busy.

Multiprogramming

By keeping more than one program in memory at a time, the operating system can switch among the memory-resident programs so that the CPU always has a program that is ready to run: when a program makes a request that makes it wait, the operating system can switch to another runnable program. In this way it keeps the CPU busy.

An operating system that can keep more than one program in memory at a time so that more than one program is capable of running at any time, is called a **multiprogramming system**.

All modern operating systems are multiprogramming systems.

Multiprogramming

By keeping more than one program in memory at a time, the operating system can switch among the memory-resident programs so that the CPU always has a program that is ready to run: when a program makes a request that makes it wait, the operating system can switch to another runnable program. In this way it keeps the CPU busy.

An operating system that can keep more than one program in memory at a time so that more than one program is capable of running at any time, is called a **multiprogramming system**.

All modern operating systems are multiprogramming systems.

Technically, the term **multiprogramming** means **keeping more than one program in memory at a time**[1], and the **degree of multiprogramming** is **the number of programs** in memory at any time.

1 Many websites mistakenly state that multiprogramming means running multiple programs at the same time.

Multiprogramming

By keeping more than one program in memory at a time, the operating system can switch among the memory-resident programs so that the CPU always has a program that is ready to run: when a program makes a request that makes it wait, the operating system can switch to another runnable program. In this way it keeps the CPU busy.

An operating system that can keep more than one program in memory at a time so that more than one program is capable of running at any time, is called a **multiprogramming system**.

All modern operating systems are multiprogramming systems.

Technically, the term **multiprogramming** means **keeping more than one program in memory at a time**[1], and the **degree of multiprogramming** is **the number of programs** in memory at any time.

The purpose of multiprogramming is to increase **CPU utilization**, which is defined as **the fraction of time that the CPU does useful work**.

1 Many websites mistakenly state that multiprogramming means running multiple programs at the same time.

Batch Processing

A **batch job** is a program that can run without user intervention. This is called "**running in the background**".

Batch Processing

A **batch job** is a program that can run without user intervention. This is called "**running in the background**".

Batch jobs typically run for a long time and process large amounts of data.

Batch Processing

A **batch job** is a program that can run without user intervention. This is called "**running in the background**".

Batch jobs typically run for a long time and process large amounts of data.

They are often run in high-performance computing centers, where users submit the jobs to a work queue to be scheduled.

Batch Processing

A **batch job** is a program that can run without user intervention. This is called "**running in the background**".

Batch jobs typically run for a long time and process large amounts of data.

They are often run in high-performance computing centers, where users submit the jobs to a work queue to be scheduled.

These computers use **batch processing** to run the jobs. **Batch processing** is a particular type of multiprogramming system in which several programs are memory-resident, and jobs are **run to completion**.

Batch Processing

A **batch job** is a program that can run without user intervention. This is called "**running in the background**".

Batch jobs typically run for a long time and process large amounts of data.

They are often run in high-performance computing centers, where users submit the jobs to a work queue to be scheduled.

These computers use **batch processing** to run the jobs. **Batch processing** is a particular type of multiprogramming system in which several programs are memory-resident, and jobs are **run to completion**.

Jobs submitted to a batch processing system may wait a long time to run because all jobs before them must run to completion.

Batch Processing

A **batch job** is a program that can run without user intervention. This is called "**running in the background**".

Batch jobs typically run for a long time and process large amounts of data.

They are often run in high-performance computing centers, where users submit the jobs to a work queue to be scheduled.

These computers use **batch processing** to run the jobs. **Batch processing** is a particular type of multiprogramming system in which several programs are memory-resident, and jobs are **run to completion**.

Jobs submitted to a batch processing system may wait a long time to run because all jobs before them must run to completion.

Batch systems are not used for interactive computing.

Interactive Computing

Think about how many different **applications** are typically "**open**" on your computer (and remember that your phone **is** a computer) at a given time.

- What do we mean by "open" above?

Interactive Computing

Think about how many different **applications** are typically "**open**" on your computer (and remember that your phone **is** a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means **it is running**.

Interactive Computing

Think about how many different **applications** are typically "**open**" on your computer (and remember that your phone **is** a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means **it is running**.
- What are "applications"?

Interactive Computing

Think about how many different **applications** are typically "**open**" on your computer (and remember that your phone **is** a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means **it is running**.
- What are "applications"? Applications are programs.

Interactive Computing

Think about how many different **applications** are typically "**open**" on your computer (and remember that your phone **is** a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means **it is running**.
- What are "applications"? Applications are programs.

So what this all means is that each of these programs is running "at the same time" and responding to your inputs, whether they are touches to the screen, keys typed on a keyboard, words spoken, or a mouse clicked. If there is a single CPU, how is this possible?

Interactive Computing

Think about how many different applications are typically "open" on your computer (and remember that your phone is a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means it is running.
- What are "applications"? Applications are programs.

So what this all means is that each of these programs is running "at the same time" and responding to your inputs, whether they are touches to the screen, keys typed on a keyboard, words spoken, or a mouse clicked. If there is a single CPU, how is this possible?

Time-sharing.

Interactive Computing

Think about how many different applications are typically "open" on your computer (and remember that your phone is a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means it is running.
- What are "applications"? Applications are programs.

So what this all means is that each of these programs is running "at the same time" and responding to your inputs, whether they are touches to the screen, keys typed on a keyboard, words spoken, or a mouse clicked. If there is a single CPU, how is this possible?

Time-sharing.

Time-sharing is a type of multiprogramming in which each program in memory gets a very small slice of time on the CPU, perhaps no more than a few milliseconds, in some well-defined order. The time slices are so small that you cannot see that the switches take place.

You get a chance to interact with each program.

Interactive Computing

Think about how many different applications are typically "open" on your computer (and remember that your phone is a computer) at a given time.

- What do we mean by "open" above? We mean that you see "the application" on the screen, which means it is running.
- What are "applications"? Applications are programs.

So what this all means is that each of these programs is running "at the same time" and responding to your inputs, whether they are touches to the screen, keys typed on a keyboard, words spoken, or a mouse clicked. If there is a single CPU, how is this possible?

Time-sharing.

Time-sharing is a type of multiprogramming in which each program in memory gets a very small slice of time on the CPU, perhaps no more than a few milliseconds, in some well-defined order. The time slices are so small that you cannot see that the switches take place.

You get a chance to interact with each program.

Time-sharing is what makes interactive computing environments possible, and the term is usually associated with interactive computing, although it is more general than this. The objective is to make each user think their program is the only program running!

Time-sharing Infrastructure

Time-sharing requires more infrastructure in the operating system:

Time-sharing Infrastructure

Time-sharing requires more infrastructure in the operating system:

- CPU scheduling, to decide which program runs next on the CPU, and whether to remove a program from the CPU that has used too much time;

Time-sharing Infrastructure

Time-sharing requires more infrastructure in the operating system:

- CPU scheduling, to decide which program runs next on the CPU, and whether to remove a program from the CPU that has used too much time;
- Memory management - deciding which programs are in memory, when to temporarily remove them to secondary storage because they are not doing anything lately, how much memory each needs;

Time-sharing Infrastructure

Time-sharing requires more infrastructure in the operating system:

- CPU scheduling, to decide which program runs next on the CPU, and whether to remove a program from the CPU that has used too much time;
- Memory management - deciding which programs are in memory, when to temporarily remove them to secondary storage because they are not doing anything lately, how much memory each needs;
- Resource management - deciding which program gets to use which device (when you click on a mouse, which program gets the click?), which devices can be shared, and so on.

Time-sharing Infrastructure

Time-sharing requires more infrastructure in the operating system:

- CPU scheduling, to decide which program runs next on the CPU, and whether to remove a program from the CPU that has used too much time;
- Memory management - deciding which programs are in memory, when to temporarily remove them to secondary storage because they are not doing anything lately, how much memory each needs;
- Resource management - deciding which program gets to use which device (when you click on a mouse, which program gets the click?), which devices can be shared, and so on.
- Timers, lists of active and inactive processes (running programs) and other structures to make this all possible.

Multitasking

Multitasking is a specific form of time-sharing in which the running programs, which we call **processes**, or **tasks**, may be cooperating to solve a single problem, sharing resources to accomplish this.

Time-sharing is slicing up the CPU's time into small pieces and giving them to processes, whether or not they are working together.

Multi-tasking is the term generally used when the processes are working together.

Multitasking includes a form of parallel computing in which programs are multithreaded, and the individual threads share the CPU.

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

The Solution:

- The CPU has a **mode bit** that can be in one of two states:
  - **user mode** (mode bit == 1)
  - **kernel mode**, also called **privileged mode** or **supervisor mode** (mode bit == 0).

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

The Solution:

- The CPU has a **mode bit** that can be in one of two states:
  - **user mode** (mode bit == 1)
  - **kernel mode**, also called **privileged mode** or **supervisor mode** (mode bit == 0).
- The CPU can determine the state of the mode bit.

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

The Solution:

- The CPU has a **mode bit** that can be in one of two states:
  - **user mode** (mode bit == 1)
  - **kernel mode**, also called **privileged mode** or **supervisor mode** (mode bit == 0).
- The CPU can determine the state of the mode bit.
- Some instructions are designated as **privileged instructions** that **can only be executed in kernel mode**.

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

The Solution:

- The CPU has a **mode bit** that can be in one of two states:
  - **user mode** (mode bit == 1)
  - **kernel mode**, also called **privileged mode** or **supervisor mode** (mode bit == 0).
- The CPU can determine the state of the mode bit.
- Some instructions are designated as **privileged instructions** that **can only be executed in kernel mode**.
- Any attempt to execute a privileged instruction in user mode results in a trap.

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

The Solution:

- The CPU has a **mode bit** that can be in one of two states:
  - **user mode** (mode bit == 1)
  - **kernel mode**, also called **privileged mode** or **supervisor mode** (mode bit == 0).
- The CPU can determine the state of the mode bit.
- Some instructions are designated as **privileged instructions** that **can only be executed in kernel mode**.
- Any attempt to execute a privileged instruction in user mode results in a trap.
- There is a privileged instruction that can change the state of the mode bit.

Dual-Mode Operation

The Problem:

The kernel needs to be able to execute all instructions and protect and control all resources, whereas user programs should only be allowed to execute certain instructions.

The Solution:

- The CPU has a **mode bit** that can be in one of two states:
  - **user mode** (mode bit == 1)
  - **kernel mode**, also called **privileged mode** or **supervisor mode** (mode bit == 0).
- The CPU can determine the state of the mode bit.
- Some instructions are designated as **privileged instructions** that **can only be executed in kernel mode**.
- Any attempt to execute a privileged instruction in user mode results in a trap.
- There is a privileged instruction that can change the state of the mode bit.

Taken together, these features are known as **dual-mode operation**.

Protection Levels

Some types of privileged instructions are the following:

- setting the value of a timer or a clock
- instructions to perform I/O
- disabling interrupts
- changing the values of certain system-controlled registers
- changing the mode bit to kernel mode

In the Intel x86 architecture, there were about fifteen different privileged instructions[1].

1 According to https://manybutfinite.com/post/cpu-rings-privilege-and-protection.

Protection Levels

Some types of privileged instructions are the following:

- setting the value of a timer or a clock
- instructions to perform I/O
- disabling interrupts
- changing the values of certain system-controlled registers
- changing the mode bit to kernel mode

In the Intel x86 architecture, there were about fifteen different privileged instructions[1].

Some processors provide more than two modes. In the Intel chips, there are four privilege rings, numbered 0, 1, 2, and 3, with 0 being the most privileged and 3, the least.

The kernel runs in ring 0, and user programs in ring 3. Some virtual machine software arranges for the guest virtual machine to run in ring 1.

1 According to https://manybutfinite.com/post/cpu-rings-privilege-and-protection.

Use of Dual-Mode Operation

When the system is booted, the kernel runs in kernel mode. After it is booted, all user processes are run in user mode.

Dual-mode operation makes it possible for user programs to request services from the operating system without compromising security.

Use of Dual-Mode Operation

When the system is booted, the kernel runs in kernel mode. After it is booted, all user processes are run in user mode.

Dual-mode operation makes it possible for user programs to request services from the operating system without compromising security.

- A user program makes a **system call** to request a service for which it does not have privilege, such as reading from a device.

Use of Dual-Mode Operation

When the system is booted, the kernel runs in kernel mode. After it is booted, all user processes are run in user mode.

Dual-mode operation makes it possible for user programs to request services from the operating system without compromising security.

- A user program makes a **system call** to request a service for which it does not have privilege, such as reading from a device.
- The system call causes a **trap**, which causes a transfer to a kernel **system call handler** that handles all system calls, and causes the mode bit to be set to 0 (kernel mode). (In Linux this system call handler is written in assembly language.)

Use of Dual-Mode Operation

When the system is booted, the kernel runs in kernel mode. After it is booted, all user processes are run in user mode.

Dual-mode operation makes it possible for user programs to request services from the operating system without compromising security.

- A user program makes a **system call** to request a service for which it does not have privilege, such as reading from a device.
- The system call causes a **trap**, which causes a transfer to a kernel **system call handler** that handles all system calls, and causes the mode bit to be set to 0 (kernel mode). (In Linux this system call handler is written in assembly language.)
- The **system call handler** saves registers, determines which system call was made, and runs the code that performs the service. When that code finishes, it returns to the system call handler.

Use of Dual-Mode Operation

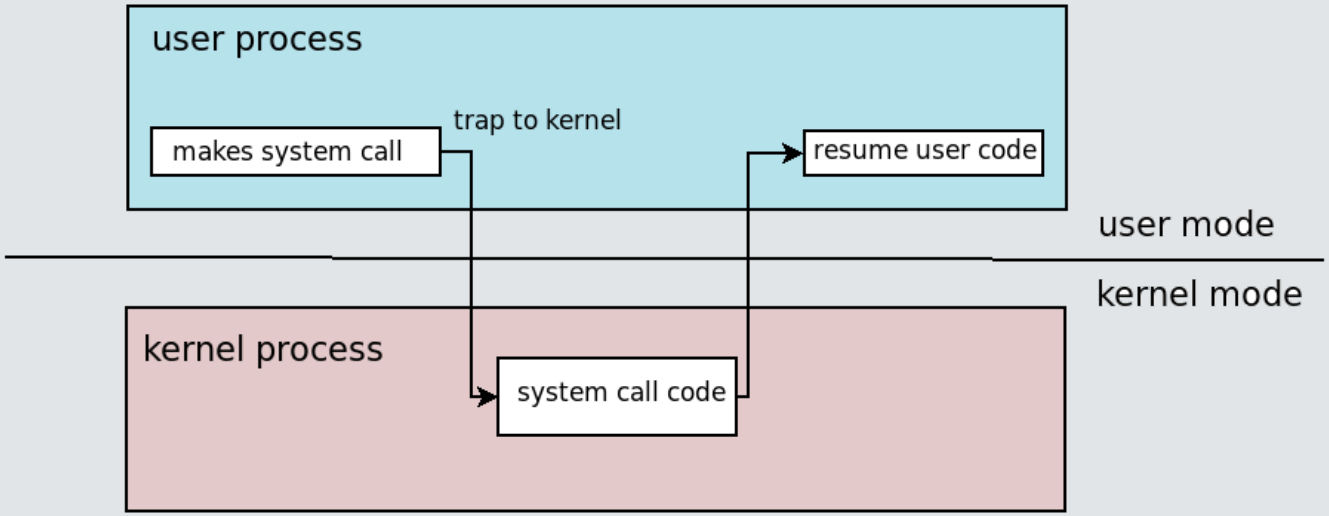
When the system is booted, the kernel runs in kernel mode. After it is booted, all user processes are run in user mode.

Dual-mode operation makes it possible for user programs to request services from the operating system without compromising security.

- A user program makes a **system call** to request a service for which it does not have privilege, such as reading from a device.
- The system call causes a **trap**, which causes a transfer to a kernel **system call handler** that handles all system calls, and causes the mode bit to be set to 0 (kernel mode). (In Linux this system call handler is written in assembly language.)
- The **system call handler** saves registers, determines which system call was made, and runs the code that performs the service. When that code finishes, it returns to the system call handler.
- When the system call handler is finished, the mode bit is set to 1 (user mode) and user code is run again.

System Call Schematic in Linux

The figure below illustrates conceptually what happens when a user process requests services from the kernel by using a system call. A trap takes place, the mode is changed from user mode to kernel mode, the kernel executes, and when it returns, the mode is changed back to user mode and user code resumes.



Timers

A **timer** is a hardware component within the processor that can be set to generate an interrupt after a set amount of time.

Timers

A **timer** is a hardware component within the processor that can be set to generate an interrupt after a set amount of time.

The use of timers plays a crucial role in managing program execution:

- It prevents user processes from holding the CPU forever in infinite loops.
- It prevents user processes from never returning control to the operating system.
- It ensures fair use of the CPU and good response times for interactive processes.

Timers

A **timer** is a hardware component within the processor that can be set to generate an interrupt after a set amount of time.

The use of timers plays a crucial role in managing program execution:

- It prevents user processes from holding the CPU forever in infinite loops.
- It prevents user processes from never returning control to the operating system.
- It ensures fair use of the CPU and good response times for interactive processes.

A **variable timer** can be implemented by a fixed-rate clock and a counter:

- The kernel sets the counter.
- Each clock tick decrements the counter.
- When the counter reaches 0, an interrupt is generated.

Timers

A **timer** is a hardware component within the processor that can be set to generate an interrupt after a set amount of time.

The use of timers plays a crucial role in managing program execution:

- It prevents user processes from holding the CPU forever in infinite loops.
- It prevents user processes from never returning control to the operating system.
- It ensures fair use of the CPU and good response times for interactive processes.

A **variable timer** can be implemented by a fixed-rate clock and a counter:

- The kernel sets the counter.
- Each clock tick decrements the counter.
- When the counter reaches 0, an interrupt is generated.

To control a user process with a timer,

- the kernel sets the timer before running the process;
- if a timer interrupt occurs, control is transferred automatically to the kernel;
- the kernel's service routine handles the interrupt, deciding whether to give the process more time or run another process.

Resource Management

The operating system manages all computer resources. We examine the various resources and how the operating system manages them.

Process Management: What are Processes?

A **process** is a program in execution.

Process Management: What are Processes?

A **process** is a program in execution.

A program is not a process. Multiple copies of the same program can be run simultaneously by the same or different users.

For example, there can be many instances of `bash`, `firefox`, or the `SSH` daemon, `sshd` running on a shared computer.

Each instance has its own "state", including the program counter, specifying the location of next instruction to execute.

Process Management: What are Processes?

A **process** is a program in execution.

A program is not a process. Multiple copies of the same program can be run simultaneously by the same or different users.

For example, there can be many instances of `bash`, `firefox`, or the `SSH` daemon, `sshd` running on a shared computer.

Each instance has its own "state", including the program counter, specifying the location of next instruction to execute.

Some programs are **multithreaded** and when they are run, there are multiple processes, called **threads**, associated with the single program, each having its own program counter.

Process Management: What are Processes?

A **process** is a program in execution.

A program is not a process. Multiple copies of the same program can be run simultaneously by the same or different users.

For example, there can be many instances of `bash`, `firefox`, or the `SSH` daemon, `sshd` running on a shared computer.

Each instance has its own "state", including the program counter, specifying the location of next instruction to execute.

Some programs are **multithreaded** and when they are run, there are multiple processes, called **threads**, associated with the single program, each having its own program counter.

At any given time there can be hundreds of different processes running on even a modestly small, shared computer system.

Process Management: What are Processes?

A **process** is a program in execution.

A program is not a process. Multiple copies of the same program can be run simultaneously by the same or different users.

For example, there can be many instances of `bash`, `firefox`, or the `SSH` daemon, `sshd` running on a shared computer.

Each instance has its own "state", including the program counter, specifying the location of next instruction to execute.

Some programs are **multithreaded** and when they are run, there are multiple processes, called **threads**, associated with the single program, each having its own program counter.

At any given time there can be hundreds of different processes running on even a modestly small, shared computer system.

Each of these processes must be managed individually by the operating system.

In short, a process is the **unit of work** in an operating system - it is the entity that the operating system manages.

Process Activity

- What else is part of the state of a process? List five different components.
- Besides the applications and programs mentioned already, what are more examples of commonly shared programs? Name three more.
- Login to a `cslab` host and use the command `ps -efw` to view the list of running processes. Find more examples of programs that are being run by multiple processes.

Process Management

What kinds of things related to processes need managing?

Process Management

What kinds of things related to processes need managing?

1. Processes need resources to accomplish their tasks. These include:
    - processor time,
    - memory,
    - access to I/O devices,
    - access to files

Process Management

What kinds of things related to processes need managing?

1. Processes need resources to accomplish their tasks. These include:
    - processor time,
    - memory,
    - access to I/O devices,
    - access to files
2. Processes are created and destroyed by the operating system.
    - Process creation requires passing initialization data to the process,
    - Process termination requires reclaiming **reusable resources**. (CPU time, for example, is not reusable, but memory is.)

Process Management

What kinds of things related to processes need managing?

1. Processes need resources to accomplish their tasks. These include:
    - processor time,
    - memory,
    - access to I/O devices,
    - access to files
2. Processes are created and destroyed by the operating system.
    - Process creation requires passing initialization data to the process,
    - Process termination requires reclaiming **reusable resources**. (CPU time, for example, is not reusable, but memory is.)

In general, operating systems are responsible for

- creating and deleting both user and system processes,
- scheduling processes and threads on the CPU(s),
- suspending and resuming processes,
- providing mechanisms for process synchronization, and
- providing mechanisms for process communication

The Memory Resource

Primary memory is the only storage accessible to the processor, and its size is finite.

The Memory Resource

Primary memory is the only storage accessible to the processor, and its size is finite.

On a typical general purpose computer, based on a von Neumann architecture,

- during the instruction-fetch cycle, the CPU fetches instructions from main memory, and
- during the data-fetch cycle, it reads and writes data from main memory.

The Memory Resource

Primary memory is the only storage accessible to the processor, and its size is finite.

On a typical general purpose computer, based on a von Neumann architecture,

- during the instruction-fetch cycle, the CPU fetches instructions from main memory, and
- during the data-fetch cycle, it reads and writes data from main memory.

This implies that

- programs must be in main memory in order to be executed.

The Memory Resource

Primary memory is the only storage accessible to the processor, and its size is finite.

On a typical general purpose computer, based on a von Neumann architecture,

- during the instruction-fetch cycle, the CPU fetches instructions from main memory, and
- during the data-fetch cycle, it reads and writes data from main memory.

This implies that

- programs must be in main memory in order to be executed.
- and that when a program is executing, it can only access data when that data is in main memory; data on disk must be copied into main memory first.

The Memory Resource

Primary memory is the only storage accessible to the processor, and its size is finite.

On a typical general purpose computer, based on a von Neumann architecture,

- during the instruction-fetch cycle, the CPU fetches instructions from main memory, and
- during the data-fetch cycle, it reads and writes data from main memory.

This implies that

- programs must be in main memory in order to be executed.
- and that when a program is executing, it can only access data when that data is in main memory; data on disk must be copied into main memory first.

The preceding facts raise many questions, such as:

- Which programs should be in memory?
- When should programs be loaded?
- When should data needed by a process be loaded into main memory? How much of it?
- How much memory should be allocated to each process?
- Where in memory should processes be kept?
- Can they be removed and brought back in again? To the same location or not?

Memory Management

The operating system needs to address the questions raised in the preceding slide. It must provide memory management services.

Memory Management

The operating system needs to address the questions raised in the preceding slide. It must provide **memory management** services.

In general, the operating system is responsible for

- keeping track of which parts of memory are currently being used and by which processes;
- deciding which processes and data should be in memory,
- deciding which memory-resident processes and data may be removed from memory to secondary storage, possibly only temporarily, and
- providing mechanisms to allocate and deallocate memory for processes

while **optimizing CPU utilization and response time to processes and their users**.

Files and File Systems

From a user's perspective, a **file** is a container that stores software, either programs (in source or object format) or data. Data files may be text or binary.

From the operating system's perspective, a **file** is a named collection of related information that is recorded on secondary storage.

It is the smallest logical unit of storage in a computer.

Files and File Systems

From a user's perspective, a **file** is a container that stores software, either programs (in source or object format) or data. Data files may be text or binary.

From the operating system's perspective, a **file** is a named collection of related information that is recorded on secondary storage.

It is the smallest logical unit of storage in a computer.

Users generally think of **file systems** as a hierarchical, tree-like structure whose internal nodes are directories and whose external nodes are non-directory-files, but this is just the user's **logical view**.

Files and File Systems

From a user's perspective, a **file** is a container that stores software, either programs (in source or object format) or data. Data files may be text or binary.

From the operating system's perspective, a **file** is a named collection of related information that is recorded on secondary storage.

It is the smallest logical unit of storage in a computer.

Users generally think of **file systems** as a hierarchical, tree-like structure whose internal nodes are directories and whose external nodes are non-directory-files, but this is just the user's **logical view**.

This is not what a file system is.

Files and File Systems

From a user's perspective, a **file** is a container that stores software, either programs (in source or object format) or data. Data files may be text or binary.

From the operating system's perspective, a **file** is a named collection of related information that is recorded on secondary storage.

It is the smallest logical unit of storage in a computer.

Users generally think of **file systems** as a hierarchical, tree-like structure whose internal nodes are directories and whose external nodes are non-directory-files, but this is just the user's **logical view**.

This is not what a file system is.

A **file system** is an abstraction that supports the creation, deletion, and modification of files, and organization of files into directories.

- It also supports control of access to files and directories and manages the disk space accorded to it.
- It is actually a flat structure sitting on a linear storage device such as a disk partition.

File System Management

Storage devices can contain many different types of physical media, with distinct physical properties. (Think about optical disks versus magnetic tapes versus magnetic disks.)

The devices themselves can have unique characteristics such as **access speed**, **capacity**, **data-transfer rate**, and **access method**.

When users and their programs work with files on these devices, they should not need to know the details of the media and the devices on which these files are stored.

File System Management

Storage devices can contain many different types of physical media, with distinct physical properties. (Think about optical disks versus magnetic tapes versus magnetic disks.)

The devices themselves can have unique characteristics such as **access speed**, **capacity**, **data-transfer rate**, and **access method**.

When users and their programs work with files on these devices, they should not need to know the details of the media and the devices on which these files are stored.

A file system is a set of structures that an operating system creates on a storage device to hide these details and provide the higher level abstraction that the user sees.

The operating system provides a programming interface to the application layer for working with files and directories.

File System Management

Storage devices can contain many different types of physical media, with distinct physical properties. (Think about optical disks versus magnetic tapes versus magnetic disks.)

The devices themselves can have unique characteristics such as access speed, capacity, data-transfer rate, and access method.

When users and their programs work with files on these devices, they should not need to know the details of the media and the devices on which these files are stored.

A file system is a set of structures that an operating system creates on a storage device to hide these details and provide the higher level abstraction that the user sees.

The operating system provides a programming interface to the application layer for working with files and directories.

In general, the operating system is responsible for the following file system-related activities:

- creating and deleting files and directories
- managing properties of files and directories
- mapping files onto secondary storage
- providing security and privacy of files and directories

File System Activity

- Explain what it means to create a file system on some medium such as a disk. What "things" are created?
- Find the command(s) that can create a file system in Linux. List them.
- What commands can create
    - a FAT32 file system on a USB drive?
    - an NTFS file system on a hard drive?
    - an ext4 file system on a hard drive?

Storage Management

The input to almost all programs is stored in files on secondary storage devices. The exceptions are those interactive programs whose inputs are mostly provided by a user on a console.

Storage Management

The input to almost all programs is stored in files on secondary storage devices. The exceptions are those interactive programs whose inputs are mostly provided by a user on a console.

Most programs write their output and their temporary results to secondary storage as well.

Storage Management

The input to almost all programs is stored in files on secondary storage devices. The exceptions are those interactive programs whose inputs are mostly provided by a user on a console.

Most programs write their output and their temporary results to secondary storage as well.

The entire file system resides on secondary storage.

Storage Management

The input to almost all programs is stored in files on secondary storage devices. The exceptions are those interactive programs whose inputs are mostly provided by a user on a console.

Most programs write their output and their temporary results to secondary storage as well.

The entire file system resides on secondary storage.

If secondary storage is not managed well, reading files, writing files, accessing files, and many other operations will be slowed down or, still worse, made impossible because there is no space left on a device.

What kinds of operations must an operating system manage with respect to secondary storage?

Storage Management

The input to almost all programs is stored in files on secondary storage devices. The exceptions are those interactive programs whose inputs are mostly provided by a user on a console.

Most programs write their output and their temporary results to secondary storage as well.

The entire file system resides on secondary storage.

If secondary storage is not managed well, reading files, writing files, accessing files, and many other operations will be slowed down or, still worse, made impossible because there is no space left on a device.

What kinds of operations must an operating system manage with respect to secondary storage?

In general, the operating system is responsible for the following storage-related activities:

- mounting and unmounting file systems onto the directory hierarchy
- managing the free space on the storage devices
- allocating blocks of free storage upon request
- scheduling the order in which disk actions are performed
- partitioning the disks
- protecting storage devices from unauthorized access

Caching Revisited

Recall from the slide introducing caches that, in general, a cache is a temporary storage area used to improve performance:

- When data and/or instructions are first accessed, they are copied temporarily from a slower, larger capacity storage component into a faster, smaller storage component (the cache), and then accessed from that cache.
- When the data is needed again, the cache is checked first to determine if it is there.
    - If so, the data is accessed directly from the cache, saving time.
    - If not, the data is copied from the slower storage to the cache and then accessed there.

Cache Management

The hardware generally manages processor cache, not the operating system. But the operating system must manage the parts of main memory used as a cache for the hard disks. This includes such actions as:

- choosing the size of the cache
- deciding when the cache is full, which data in it should be replaced
- if data in the cache is modified, deciding when to copy the changes back to the storage from which they came.

Cache Management

The hardware generally manages processor cache, not the operating system. But the operating system must manage the parts of main memory used as a cache for the hard disks. This includes such actions as:

- choosing the size of the cache
- deciding when the cache is full, which data in it should be replaced
- if data in the cache is modified, deciding when to copy the changes back to the storage from which they came.

But there may be more to handle...

Cache Management

The hardware generally manages processor cache, not the operating system. But the operating system must manage the parts of main memory used as a cache for the hard disks. This includes such actions as:

- choosing the size of the cache
- deciding when the cache is full, which data in it should be replaced
- if data in the cache is modified, deciding when to copy the changes back to the storage from which they came.

But there may be more to handle...

When a computer has multiple processors, **a copy of a data item might exist in multiple caches, each in a different processor**. The system has to make sure that an update to a data item in one cache is immediately reflected in all other caches where a copy of that item exists.

A system in which this is true has cache coherency. Hardware handles cache coherency in processor cache, but the operating system must handle it in caches in main memory.

The I/O Subsystem

Modern operating systems provide **device-independent I/O** to users and user programs. **Device independent I/O** is I/O such that all I/O devices are accessible to programs without the program's needing to specify the device in advance.

For example, the code in a program that opens a file and then reads it, is the same whether the file is on an internal hard disk, a USB device, or a magnetic tape.

In UNIX, for example, the `read()` system call has the same arguments regardless of whether it is reading from a file, a pipe, a terminal device, or a network interface.

The I/O Subsystem

Modern operating systems provide **device-independent I/O** to users and user programs. **Device independent I/O** is I/O such that all I/O devices are accessible to programs without the program's needing to specify the device in advance.

For example, the code in a program that opens a file and then reads it, is the same whether the file is on an internal hard disk, a USB device, or a magnetic tape.

In UNIX, for example, the `read()` system call has the same arguments regardless of whether it is reading from a file, a pipe, a terminal device, or a network interface.

The device driver interface is also device-independent. This means that all device drivers, regardless of the device type or manufacturer, are required to provide the same interface to the kernel. They implement a standard interface. This makes maintaining the kernel code easier.

Protection and Security

Operating systems must also provide both **protection** and **security**.

Protection is the set of **policies and mechanisms** for controlling access by processes and users to the resources managed by the operating system.

Examples of Controlling Access:

- Defining which users are allowed to view the system log files
- Preventing user processes from writing to secondary storage directly.

Protection and Security

Operating systems must also provide both **protection** and **security**.

Protection is the set of **policies and mechanisms** for controlling access by processes and users to the resources managed by the operating system.

Examples of Controlling Access:

- Defining which users are allowed to view the system log files
- Preventing user processes from writing to secondary storage directly.

Security is the set of **policies and mechanisms** for the defense of the computer system against internal and external attacks.

Examples of Attacks:

- denial-of-service, worms, viruses, identity theft, theft of service

Protection and Security Management

There is a broad range of methods of providing protection and security in modern operating systems, as well as on-going research into operating system security.

Unix and its derivatives, as well as **Apple** and **Windows** operating systems share a common paradigm:

Protection and Security Management

There is a broad range of methods of providing protection and security in modern operating systems, as well as on-going research into operating system security.

Unix and its derivatives, as well as **Apple** and **Windows** operating systems share a common paradigm:

- **User identities**, such as user names and numeric ids, are assigned to each user.

Protection and Security Management

There is a broad range of methods of providing protection and security in modern operating systems, as well as on-going research into operating system security.

Unix and its derivatives, as well as **Apple** and **Windows** operating systems share a common paradigm:

- **User identities**, such as user names and numeric ids, are assigned to each user.
- A system of **authentication**, such as passwords, two-factor authentication, cryptographic key-pairs, and so on, is used to authenticate logins.

Protection and Security Management

There is a broad range of methods of providing protection and security in modern operating systems, as well as on-going research into operating system security.

Unix and its derivatives, as well as **Apple** and **Windows** operating systems share a common paradigm:

- **User identities**, such as user names and numeric ids, are assigned to each user.
- A system of **authentication**, such as passwords, two-factor authentication, cryptographic key-pairs, and so on, is used to authenticate logins.
- User identities are associated with **privileges** and **access rights** for all files and other entities needing privilege, and **processes of that user inherit those privileges**.

Protection and Security Management

There is a broad range of methods of providing protection and security in modern operating systems, as well as on-going research into operating system security.

Unix and its derivatives, as well as **Apple** and **Windows** operating systems share a common paradigm:

- **User identities**, such as user names and numeric ids, are assigned to each user.
- A system of **authentication**, such as passwords, two-factor authentication, cryptographic key-pairs, and so on, is used to authenticate logins.
- User identities are associated with **privileges** and **access rights** for all files and other entities needing privilege, and **processes of that user inherit those privileges**.
- **Group identifiers**, such as a group id, allow sets of users to be defined and to control what they can do collectively to controlled files and other entities. Processes of a user in a group inherit those privileges.

Protection and Security Management

There is a broad range of methods of providing protection and security in modern operating systems, as well as on-going research into operating system security.

Unix and its derivatives, as well as **Apple** and **Windows** operating systems share a common paradigm:

- **User identities**, such as user names and numeric ids, are assigned to each user.
- A system of **authentication**, such as passwords, two-factor authentication, cryptographic key-pairs, and so on, is used to authenticate logins.
- User identities are associated with **privileges** and **access rights** for all files and other entities needing privilege, and **processes of that user inherit those privileges**.
- **Group identifiers**, such as a group id, allow sets of users to be defined and to control what they can do collectively to controlled files and other entities. Processes of a user in a group inherit those privileges.
- **Privilege escalation** (such as `sudo` in Linux) allows a user to change to an effective identity that has greater privilege.

Distributed Systems

The definition of a **distributed system** has evolved over many years. What is common to most definitions is that a **distributed system**

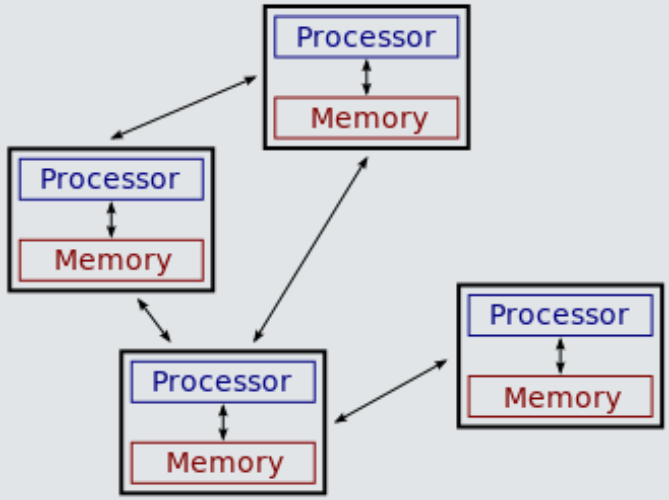
- has several autonomous computational elements, called **nodes**, each of which has its own local memory, and that
- the nodes communicate via message passing.

Distributed Systems

The definition of a **distributed system** has evolved over many years. What is common to most definitions is that a **distributed system**

- has several autonomous computational elements, called **nodes**, each of which has its own local memory, and that
- the nodes communicate via message passing.

Distributed systems can be used to solve large computational problems or to provide and coordinate the collective, shared resources of the system to individual users. The figure below illustrates a distributed system.



Image, modified, from Miym, license: CC BY-SA 3.0; Wikipedia. https://commons.wikimedia.org/w/index.php?curid=7317036

Network and Distributed Operating Systems

Network and distributed operating systems are types of operating systems whose objective is to unify the collective resources of a distributed computer system so that users can access the resources of any node on the network.

Network and Distributed Operating Systems

Network and distributed operating systems are types of operating systems whose objective is to unify the collective resources of a distributed computer system so that users can access the resources of any node on the network.

A **network operating system** provides an environment in which a user can access remote resources by either logging in to the remote machine or transferring those resources from the remote machine to the local machine.

All popular, general-purpose operating systems, such as *Linux*, *MacOS X*, and *Windows*, as well as embedded operating systems such as *Android* and *IOS*, are network operating systems.

Network and Distributed Operating Systems

Network and distributed operating systems are types of operating systems whose objective is to unify the collective resources of a distributed computer system so that users can access the resources of any node on the network.

A **network operating system** provides an environment in which a user can access remote resources by either logging in to the remote machine or transferring those resources from the remote machine to the local machine.

All popular, general-purpose operating systems, such as *Linux*, *MacOS X*, and *Windows*, as well as embedded operating systems such as *Android* and *IOS*, are network operating systems.

In contrast, a **distributed operating system** is an operating system that allows users to access remote resources in the same way that they access local resources. It creates the illusion that only a single operating system controls the network.

There are no modern entirely distributed operating systems, although many operating systems have distributed subsystems. For example, the **Network File System** (**NFS**) is a distributed file system used in Linux, and *Apache*'s **Hadoop distributed file system** (**HDFS**) is a distributed file system written in Java.

Free and Open Source Software

Roughly put,

> **open source software** is software with source code that anyone can **inspect**, **modify**, and **enhance**.[1]

and **free software** is software that

> respects users' freedom and community. Roughly, it means that the users have the freedom to run, copy, distribute, study, change and improve the software."[2]

1 OpenSource.com, https://opensource.com/resources/what-open-source
2 Richard Stallman, The Free Software Foundation. https://www.gnu.org/philosophy/free-sw.html

Free and Open Source Software

Roughly put,

> **open source software** is software with source code that anyone can **inspect**, **modify**, and **enhance**.[1]

and **free software** is software that

> respects users' freedom and community. Roughly, it means that the users have the freedom to run, copy, distribute, study, change and improve the software."[2]

"Free" has nothing to do with the monetary price of the software. The difference between free software and open source software is partly operational and partly philosophical.

- Open source software is software that must comply with one or more approved open source licenses. Free software is software that grants its users the four essential freedoms, defined by the Free Software Foundation.

1 OpenSource.com, https://opensource.com/resources/what-open-source
2 Richard Stallman, The Free Software Foundation. https://www.gnu.org/philosophy/free-sw.html

Free and Open Source Software

Roughly put,

> **open source software** is software with source code that anyone can **inspect**, **modify**, and **enhance**.[1]

and **free software** is software that

> respects users' freedom and community. Roughly, it means that the users have the freedom to run, copy, distribute, study, change and improve the software."[2]

"Free" has nothing to do with the monetary price of the software. The difference between free software and open source software is partly operational and partly philosophical.

- Open source software is software that must comply with one or more approved open source licenses. Free software is software that grants its users the four essential freedoms, defined by the Free Software Foundation.

There is a long and interesting story about the evolution of free software and open source software. Much of it is due to Richard Stallman, who zealously promoted the idea and who founded the Free Software Foundation and started the GNU Project.

1 OpenSource.com, https://opensource.com/resources/what-open-source
2 Richard Stallman, The Free Software Foundation. https://www.gnu.org/philosophy/free-sw.html

Free and Open Source Operating Systems

UNIX was open source from the outset. For a short while there were conflicts, but eventually, several versions of it remained free and open source.

Free and Open Source Operating Systems

UNIX was open source from the outset. For a short while there were conflicts, but eventually, several versions of it remained free and open source.

- Linux, BSD, Solaris and many other UNIX derivative operating systems are free and open source. You can inspect their source code; you can download and install them without any cost.
- For example, you can download the entire **Ubuntu Bionic** kernel code from git.launchpad.
- The core of **MacOS X** is open source.

Free and Open Source Operating Systems

UNIX was open source from the outset. For a short while there were conflicts, but eventually, several versions of it remained free and open source.

- Linux, BSD, Solaris and many other UNIX derivative operating systems are free and open source. You can inspect their source code; you can download and install them without any cost.
- For example, you can download the entire Ubuntu Bionic kernel code from git.launchpad.
- The core of MacOS X is open source.

Why is this important?

Free and Open Source Operating Systems

UNIX was open source from the outset. For a short while there were conflicts, but eventually, several versions of it remained free and open source.

- Linux, BSD, Solaris and many other UNIX derivative operating systems are free and open source. You can inspect their source code; you can download and install them without any cost.
- For example, you can download the entire Ubuntu Bionic kernel code from git.launchpad.
- The core of MacOS X is open source.

Why is this important?

- The availability of source code means we can study and inspect operating systems by looking at their code.
- Questions that could only be answered in the past by looking at documentation or by inferring from the behavior of an operating system, can now be answered by direct examination of code.

References

1. Randall Hyde. *The Art of Assembly Language*. No Starch Press. 2003.

2. Abraham Silberschatz, Greg Gagne, Peter B. Galvin. *Operating System Concepts*, 10th Edition. Wiley Global Education, 2018.