Systems Programming

Semester 2, 2023

Dr Timothy Chappell

Tarang Janawalkar

Contents

1 Operating Systems Overview

An operating system is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. The purpose of an operating system is to:

- execute user programs and make solving user problems easier
- make the computer system convenient to use
- use computer hardware in an efficient manner

1.1 What Operating Systems Do

A computer system is divided into four components:

- Hardware that provides basic computing resources, i.e., the CPU, memory, and I/O devices
- An operating system which controls and coordinates the use of hardware for applications and users
- Application programs that define how system resources are used to solve user computing problems
- Users that make use of the computer system. This includes people, machines, or other computers

We can also view a computer system as consisting of hardware, software, and data.

This hierarchy of components is layered such that users cannot directly access the hardware. As there are multiple different types of computer hardware, applications will typically rely on the operating system to manage the use of computer hardware so that applications can designed to operate on any hardware. Due to this, the operating system will typically restrict direct access to hardware resources.

The task of an operating system is to provide convenience, such that users do not have to worry about resource utilisation. Depending on the type of user, a computer system will prioritise one of the following:

- Shared computers such as mainframes or minicomputers are required to distribute resources to multiple users.
- Dedicated systems such as workstations have dedicated resources for a single user, but will frequently use shared resources (such as CPU cores or memory) from servers.
- Handheld devices (such as phones, tablets, and laptops) are resource poor in comparison to desktop devices, and are optimised for usability, portability, and battery life.
- Embedded devices often have little or no user interface, but access computing resources via alternate means, such as sensors in vehicles.

Definition 1.1 (Operating System)**.** An Operating System (OS) is a **resource allocator** that manages all resources in a computer system that resolves conflicting requests of resources by efficiently and fairly distributing resources.

An OS is also a **control program** that controls the execution of programs to prevent errors and improper use of it's system. The OS acts as a security layer between applications, such that one application cannot interfere with another, or bring down the entire system.

An OS must therefore be robust and reliable[1].

A more common definition is that the OS is the one program that runs at all times on the computer, which is known as the **kernel**, and is the core of the operating system. Everything else is either a **system program**, which are associated with the OS, or an **application program**.

1.2 Computer System Organisation

1.2.1 Device Controllers

A computer system consists of one or more CPUs and a number of **device controllers** connected through a common **bus** that provides access between components and shared memory. This bus is responsible for concurrent communication between the CPU and other devices.

Each device controller is responsible for a specific type of device, and depending on the device, may have more than one device attached. A device controller maintains a **local buffer** and a set of **special-purpose registers**. The device controller is responsible for moving data between the device and its local buffer, which is then moved to/from main memory by the CPU.

Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides a uniform interface to the rest of the operating system.

Device controllers inform the CPU that they have finished their operation by causing an **interrupt**.

1.3 Computer Operation

Consider a typical computer operation of performing I/O.

1. To start the operation, the device driver loads the appropriate registers within the device controller.
2. The device controller examines the contents of these registers to determine what actions to take.
3. The device controller will then start the transfer of data from the device to its local buffer.
4. Once the transfer is complete, the device controller informs the device driver that it has finished its operation.
5. The device driver then gives control back to the operating system, through an interrupt.

[1] This is far less onerous than requiring all applications to be error-free.

1.4 Interrupts

Operating systems are **interrupt driven**, and will respond to events as they occur.

Interrupts **transfer control** to an **interrupt service routine** (ISR) through the **interrupt vector**, which contains the address of all service routines, stored in low memory for quick access. A serice routine is simply a function, or piece of code, that is executed when an interrupt occurs.

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually through the system bus.

When an interrupt occurs, the CPU halts it's current execution and the interrupt architecture saves the address of the interrupted instruction, so that it can be resumed once the ISR has finished. The CPU then jumps to a fixed location in memory, which is the starting address of the ISR.

The interrupt architecture must also save the state information of the interrupted process, so that it can be restored once the ISR has finished.

1.4.1 Implementation

The basic interrupt mechanism is described below.

The CPU has an **interrupt-request line** that the CPU sense after executing every instruction. When the CPU detects that a controller has *asserted* a signal onto this line, it reads the interrupt number and jumps to the corresponding **interrupt-handler routine**.

The interrupt handler:

- saves any state it will change during its operation
- determines the cause of the interrupt
- performs the necessary processing
- restores the saved state
- executes a **return from interrupt** instruction, which returns the CPU to the execution state, prior to the interrupt

A device controller **raises** an interrupt by asserting a signal on the interrupt-request line, and the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, which then **clears** the interrupt by servicing the device.

1.4.2 Types of Interrupts

A **trap** or **exception** is a software-generated interrupt caused by an error or a user request, and is often used to communicate with the operating system. Software can trigger an interrupt through a special operation called a **system call** (or monitor call).

An operating system also makes a distinction between the following types of interrupts:

- For a **polled** interrupt, the operating system periodically queries a queue of interrupts, to see

if one needs to be serviced.

- In a **vectored** interrupt system, the interrupt vector table will interrupt the CPU to service the necessary interrupt.

1.5 Storage Structure

1.5.1 Main Memory

The CPU can only load instructions from memory, and therefore programs must first be loaded into memory before they can be executed. Computers run most of their programs from rewritable memory, called **main memory** (or **random-access memory** (RAM)), which means that it can both read and write to any location in memory.

Main memory is **volatile** and will lose its content when power is lost.

1.5.2 Registers

All forms of memory provide an array of **bytes** (or words) that can be individually accessed by a **memory address**. The CPU interacts with these memory locations through **load** or **store** instructions.

- A **load** instruction moves data from main memory into an internal register in the CPU
- A **store** instruction moves data from an internal register in the CPU to main memory

The operating system preserves the state of the CPU through **registers**. Registers can store data within the CPU, and are the fastest form of memory available to the CPU.

Registers are often used to carry out operations such as addition, where the two operands are stored in two registers, before their sum can be computed and saved to another register or in memory.

The CPU also uses registers for storing other information such as the status of an operation, and the program counter, which is the address of the next instruction to be executed.

1.5.3 Cache Management

Caching is an important principle in computer systems, and is used to improve performance at many levels of a computer system. Caching refers to the temporary copying of data from a slower storage system into a faster storage system, where it can be accessed more quickly.

When some piece of information is required, we first check whether a copy of that data is in the cache.

- If so, we use the information directly from the cache
- If not, we use the information from the source, while placing a copy of that data into the cache, under the assumption that it will be needed again soon

Internal programmable registers provide high-speed cache for main memory. The programmer (or compiler) implements register allocation and replacement algorithms to decide which information

is kept in registers and which is kept in main memory.

Other caches are implemented in hardware. For example, most systems have an **instruction cache** to hold instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles for the instruction to be fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy.

Because caches have limited size, **cache management** is an important design problem. Careful selection of cache size and of a replacement policy can significantly improve performance.

The movement of information between levels of a storage hierarchy may be either **explicit** or **implicit**. For instance, data transfer from cache to the CPU and registers is usually a hardware function, with no operating system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

An Example Consider the following example where an integer A is to be incremented by 1, and is located in file B which resides on hard disk.

1. The operating system loads the file B from disk into main memory.
2. The operating system then load the integer A from main memory into the cache of an internal register.
3. The CPU performs the increment operation on the internal register.
4. The operating system then updates value of A from internal memory to the file B in main memory.
5. The operating system then writes the updated value of B back to disk.

Implications of Various Environments In a single processor system, where only one process executes at a time, this hierarchy poses no difficulties, as access to the integer A will always be to the copy at the **highest level** of the hierarchy.

In a **multitasking environment**, where multiple processes execute concurrently, extreme care must be taken to ensure that, if two or more processes are accessing the same data, then each process must access the **most recently updated** copy of the data.

Furthermore, in a **multiprocessor environment**, each CPU also contains a local cache. In such an environment, a copy of data may exist simultaneously in several caches. As these CPUs can execute in parallel, we must ensure that an update to data is propagated to all copies of the data in all caches.

This is known as **cache coherency**, and is usually a hardware level problem.

1.5.4 Secondary Storage

Systems with a **von Neumann architecture** fetch instructions from memory and store them in the **instruction register**. When this instruction is decoded, it may require addition operands to be fetched from memory, and stored into internal registers.

Ideally, we want programs and data to be stored in main memory permanently, to allow for fast access. However, main memory is usually too small to store all necessary programs and data permanently, and volatile.

Thus, most computer systems provide **secondary storage** as an extension of main memory. Secondary storage is nonvolatile and is used to store large amounts of data permanently. It is usually much slower than main memory.

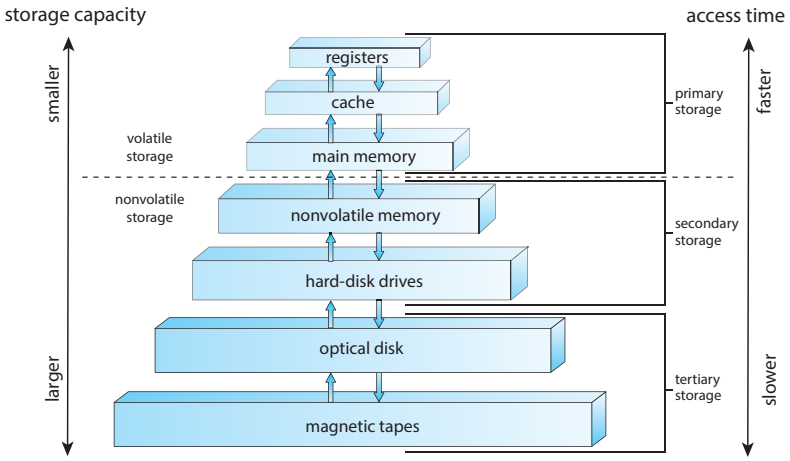
Programs are stored in secondary storage until they are loaded into memory. The most common forms of secondary storage are **hard-disk drives** (HDDs) and **nonvolatile memory** (NVM) **devices**.

1.5.5 Tertiary Storage

Large storage capacities can also be achieved through **tertiary storage**, which is used for data that is not frequently accessed, such as in archival storage, or for backup. Examples of this type of storage includes **magnetic tape** and **optical disk** storage.

1.5.6 Summary

A summary of the storage hierarchy is shown below.



From here onwards, the term **memory** will be used to refer to volatile storage, while **nonvolatile storage** (NVS) will be used to refer to nonvolatile storage.

The design of a complete storage system must balance,

- Cost: NVS is cheaper than memory
- Performance: memory is faster than NVS
- Volatility: memory loses its contents when power is lost

1.6 I/O Structure

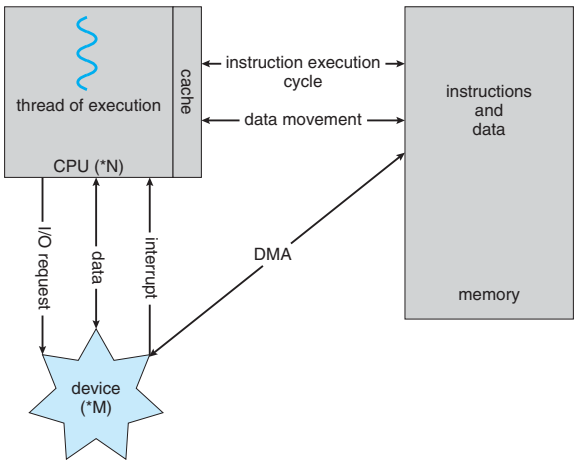
I/O is structured in one of two ways:

1. After I/O starts, control returns to the user program only upon I/O completion.
    - **Wait instructions** idle the CPU until the next interrupt
    - At most one I/O request is outstanding at a time, and no simultaneous I/O processing is possible
2. After I/O starts, control returns to the user program without waiting for I/O completion.
    - **System calls** request the OS to allow the user to wait for I/O completion
    - A **device-state table** contains entries for each I/O device, indicating its type, address, and state
    - The OS indexes into this table to determine the device status and modifies the table entry to include interrupt information

1.6.1 Direct Memory Access

The form of interrupt-driven I/O described above is sufficient for moving small amounts of data, but can produce high **overhead** when used for bulk data transfer.

To solve this problem, **direct memory access** (DMA) is used. This allows device controllers to transfer entire blocks of data directly to or from the device and main memory, without CPU intervention. Only one interrupt is generated per block, to indicate the operation has completed, rather than one interrupt per byte.



While the device controller is performing these operations, the CPU is idle, and can be used by another process.

1.7 Computer-System Architecture

Definitions

This section will use the following definitions:

Core The basic computation unit of the CPU

CPU The hardware that executes instructions

Processor A physical chip that contains one or more CPUs

Multicore Multiple cores on the same CPU

Multiprocessor Multiple processors

1.7.1 Single-Processor Systems

A single processor system is one which contains a single general purpose CPU with a single processing core.

Older computer systems used a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and accesses registers for storing data locally. The **CPU** is capable of executing a **general-purpose** instruction set, including instructions from processes.

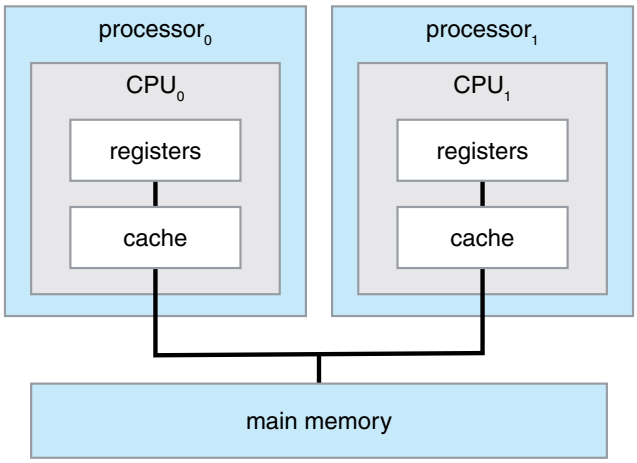
These systems also have other **special-purpose** processors such as device-specific processors for disk, keyboard, and graphics controllers. These processors run a limited instruction set and cannot run processes.

Sometimes these processors are managed by the OS, which sends them information about the next task and monitors their status.

In other systems, special-purpose processors are low-level components built into the hardware. The OS cannot communicate with these processors, as they execute jobs autonomously.

1.7.2 Multiprocessor Systems

Multiprocessor systems (or **parallel (tightly-coupled) systems**), are systems with two or more processors.



Multiprocessor systems have several advantages over single processor systems:

- **Increased throughput**: More work can be accomplished in less time
- **Economy of scale**: Multiprocessor systems are cheaper than equivalent multiple single processor systems
- **Increased reliability**: Graceful degradation or fault tolerance — if one CPU fails, the system can continue to operate

There are two types of multiprocessor systems:

- **Asymmetric multiprocessing**: Each processor is assigned a specific task, such as I/O or process scheduling
- **Symmetric multiprocessing**: Each processor performs all tasks, including OS activities

Symmetric Multiprocessing

The most common multiprocessor systems use symmetric multiprocessing (SMP), where each CPU performs all tasks, including operating system functions and user processes. Each CPU has its own registers and cache, but all processors share physical memory over the **same bus**.

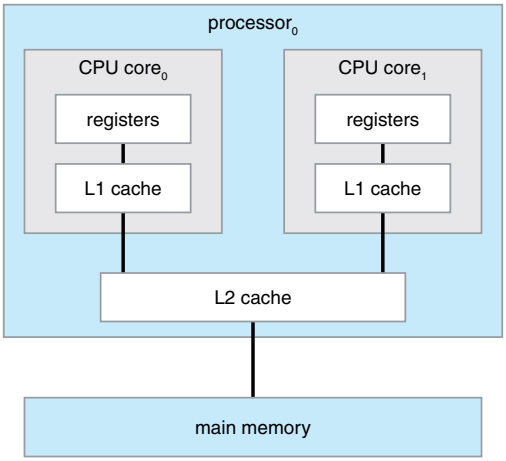
The benefit of this model is that many processes can be run **simultaneously**, without performance degradation. However, since CPUs are separate, one may be idle which another is busy, resulting in inefficiencies.

One solution to this problem is to share certain **data structures** between processors. This will allow processors and resources (such as memory) to be shared dynamically amongst processors, reducing workload variation between processors.

Such systems must properly **schedule** and **synchronise** access to shared resources, to avoid conflicts between processors.

Multicore Systems

Multicore systems are systems with multiple computing cores on the same chip (processor).



Multicore systems:

- are more efficient than systems with multiple chips with single cores, because on-chip communication is faster than between-chip communication.
- use significantly less power than multiple single core chips.

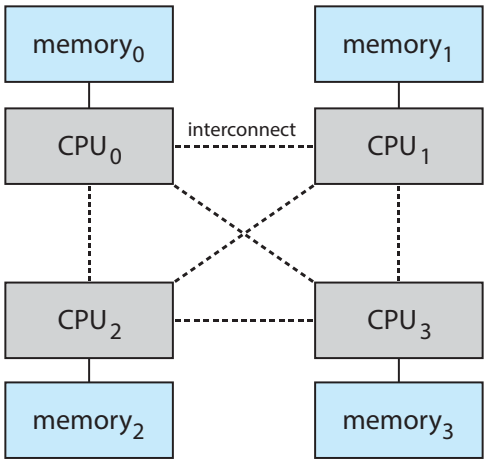
Each core has its own local cache, known as **level-1**, or **L1 cache**. In addition to this, each core shares a **level-2**, or **L2 cache**, that is local to the chip. Lower levels of cache are generally smaller, but have faster access times than higher level shared caches.

A multicore processor with N cores appears to the operating system as N standard CPUs. This requires operating system designers and application programmers to make efficient use of these additional processing cores.

Non-Uniform Memory Access Multiprocessing Systems

While adding additional CPUs to multiprocessor systems increases computing power, contention for the system bus creates a bottleneck and limits performance. An alternative approach is to provide each CPU (or groups of CPUs) with its own **local memory** that is accessed via a small but fast, local bus. These CPUs are connected by a **shared system interconnect**, so that all CPUs share one physical address space.

This is known as a **non-uniform memory access** (NUMA) architecture.



The advantages and disadvantages of NUMA multiprocessing architectures are as follows:

- CPUs have fast access to local memory and require no contention over the system interconnect
- NUMA can be scaled more effectively as more CPUs are added
- There is increased **latency** when accessing **remote memory** across the system interconnect

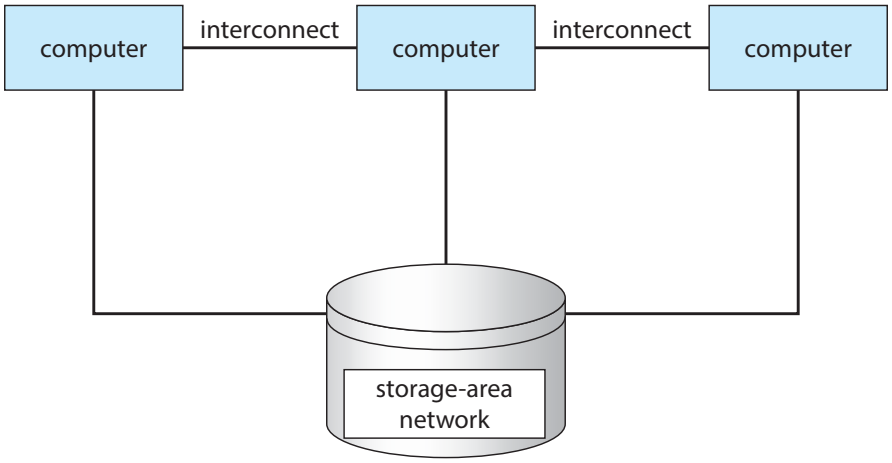
Due to their excellent scalability, NUMA systems are very popular on servers and high-performance computing systems.

Blade Servers

Blade servers are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. These servers consist of multiple independent multiprocessor systems, that runs its own operating system.

1.7.3 Clustered Systems

Clustered systems are another type of multiprocessor system that are composed of two or more individual systems, called **nodes**. Each system is typically a multicore system, and the system is **loosely coupled**. Clustered computers share storage via a **storage-area network** (SAN) and are usually connected via a **local-area network** (LAN).



High-Availability Service

The purpose of a clustered system is to provide a **high availability service**, that is, the ability to operate even if one or more nodes fail. This is achieved by adding a level of redundancy in the system. In this system, a layer of cluster software runs on the cluster nodes to monitor one or more nodes, such that if the monitored machine fails, the monitoring machine can take ownership of its resources.

The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems are called **fault tolerant** if they can suffer a failure of a single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and corrected.

Clustering can be structured asymmetrically or symmetrically:

- **Asymmetric clustering** has one machine in **hot-standby mode**, while the other runs applications. The hot-standby host machine monitors the active server, so that if it fails, it will become the active server.
- **Symmetric clustering** has multiple hosts running applications, while monitoring each other. This structure is more efficient as it uses all available hardware, but only if more than one application is running.

High-Performance Computing

As a cluster consists of several computer systems connected via a network, clusters can also provide **high-performance computing** (HPC) environments. Such systems supply significantly greater computational power because they can run applications concurrently on several computers.

This is primarily useful for applications that utilise **parallelisation**, which is the process of breaking down a large task into smaller components that run on individual cores in a computer. These tasks are designed to then be recombined to produce the final result.

Parallel Clustering

Parallel clusters allow multiple hosts to access the same data on a shared storage over a **wide-area network** (WAN). Such systems require specialised versions of software that can support simultaneous data access by multiple hosts, to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager** (DLM), is included in some cluster technologies.

1.8 Operating System Operations

1.8.1 Computer Startup

When a computer is turned on or rebooted, the first program it runs is a **bootstrap program** (**firmware**), which then loads the operating system. This program is stored in **read only memory** (ROM) or **electrically erasable programmable read only memory** (EEPROM). This storage is infrequently written to and is nonvolatile.

This program will search for an operating system kernel within all connected hard disks, optical drives, or USBs, and load the first one it finds into memory. The order in which this search is conducted is known as the **boot sequence**, and can be configured in the **basic input/output system** (BIOS).

Some services are provided outside of the kernel, by system programs, that are loaded into memory at boot time to become **system daemons**, which run while the kernel is running. On Linux, the first system program is "`systemd`", and it starts many other daemons.

Once this is completed, the system is fully booted, and waits for some event to occur. As discussed earlier, events are signalled via interrupts.

1.8.2 Multiprogramming

Users of a system typically want to run multiple programs at the same time, rather than having one program keeping the CPU or I/O devices busy at all times. **Multiprogramming** allows an operating system to increase CPU utilisation, and satisfy user requirements, by organising jobs (code and data) such that the CPU always has one to execute. In such a system, a program in execution is called a **process**.

The operating system keeps a subset of processes in memory, where the CPU executes processes one at a time, switching between processes when the current process no longer requires the CPU or is waiting for I/O. This ensures the CPU is never idle as long as there are processes to execute. This is known as **process scheduling**.

1.8.3 Multitasking

Multitasking (or **timesharing**) is a logical extension of multiprogramming, in which a CPU switches between multiple processes frequently, providing the illusion that multiple processes are executing simultaneously. For instance, the time a user takes to type a command or click a mouse is incredibly slow for a computer, and hence the CPU may switch to another process while waiting for the user to provide input.

Multitasked systems require additional considerations:

- **Interactive** systems require fast **response times** (less than 1s), so that users do not have to wait for long periods of time.
- Having several processes in memory requires **memory management**.
- If several processes are ready to be executed, **CPU scheduling** is required to decide which process to execute next.
- If multiple processes are executing concurrently, **process synchronisation** is required to ensure that processes do not interfere with each other.

For processes that are larger than **physical memory**, **virtual memory** may be used to execute processes that are stored partially in memory. This arrangement of memories addresses memory usage constraints.

*Both multiprogramming and multitasking systems must provide a **file system** to allow processes to access data stored on secondary storage. In addition to this, they must **protect resources** from inappropriate use, provide mechanisms to process **synchronisation and communication**, and ensure that processes do not get stuck in a **deadlock**.*

1.9 Dual-Mode Operation

As the operating system and users share hardware and software resources, the operating system must ensure that malicious programs cannot cause other programs (or the operating system itself) to execute incorrectly. To distinguish between the execution of operating system code and user-defined code, we can use **dual-mode** operation:

- **User mode** — used for executing user-defined code, and restricts direct access to hardware and special instructions.
- **Kernel mode** — used for executing operating system code, and allows direct access to hardware and all privileged instructions.

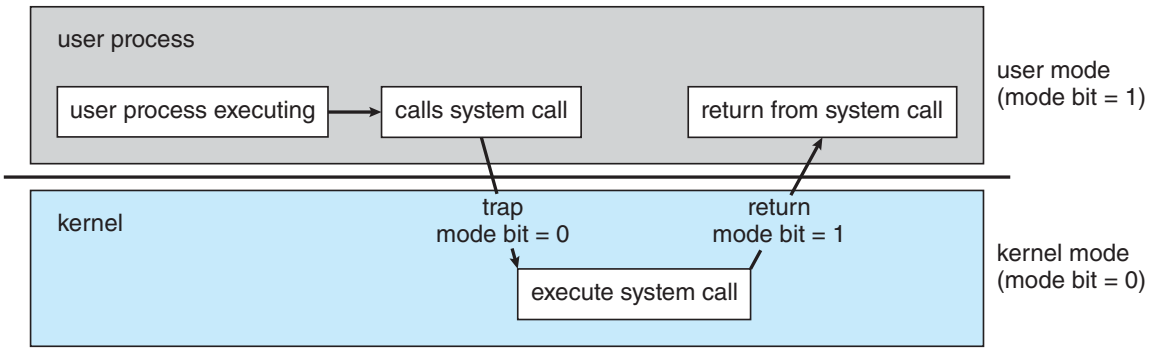
A **mode bit** is used to indicate the current mode of operation; 0, when the system is in kernel mode, and 1, when the system is in user mode. At system boot time, hardware starts in kernel

mode, and the operating system (when it is loaded), starts user applications in user mode.

When a trap or interrupt occurs, the hardware switches from user mode to kernel mode, and always switches to user mode *before* returning control to the user program.

This also allows us to designate certain machine instructions as **privileged instructions**, which can only be executed in kernel mode. For example, the instruction to switch from user mode to kernel mode is a privileged instruction.

An example is shown in the figure below.



1.10 Multi-Mode Operation

The dual-mode concept can be extended to include multiple modes of operation, where each mode has a different level of privilege. One such example of this is with CPUs that support virtualisation, where a separate mode is used to indicate when the **virtual machine manager** (VMM) is in control of the system.

1.11 Timers

To ensure that the operating system maintains control over the CPU, a **timer** is used to prevent a user program from running indefinitely.

- The operating system configures a timer to interrupt the CPU after a specific period of time, before transferring control to the user
- The timer is decremented for every clock tick
- An interrupt is generated when the timer reaches 0
- The operating system decides whether to regain control of the CPU, or allow the program to continue running

2 Operating System Structures

An operating system provides an environment for executing programs and services to programs and users. One set of operating system services provides functions that are helpful to the **user**:

- **User interface** — almost all operating systems have a **user interface** (UI) that is either **command-line** or **graphical**. This interface can be interacted with via the keyboard or mouse. Touchscreens also provide **touch screen interfaces**.
- **Program execution** — The system must be able to load programs into memory to run them, and also end their execution, either normally, or abnormally (due to an error).
- **I/O operations** — A running program may require I/O, which may involve a file or an I/O device. The operating system provides a uniform interface to I/O devices.
- **File-system manipulation** — Programs need to read and write files or directories, create or delete them by name, search for files, list file information, and manage permissions and ownership.
- **Communications** — Processes may exchange information, on the same computer or between computers over a network. Communications may be via **shared memory**, in which two or more processes read and write to a shared section of memory, or through **message passing**, where packets of information in predefined formats are moved between processes by the operating system.
- **Error detection** — The operating system must detect and correct errors constantly.
  - Errors may occur in the CPU and memory hardware (memory errors or power failures), I/O devices (parity errors or connection failures on a network, or lack of paper in a printer), and in user programs (division by zero or invalid memory access).
  - The operating system must take the appropriate action to ensure correct and consistent computing.
  - Debugging facilities can enhance the user's and programmer's abilities to efficiently use the system.

Another set of operating system functions exist to ensure efficient operation of the system via resource sharing:

- **Resource allocation** — When multiple users or multiple jobs running concurrently, resources must be allocated to each of them.
  - some resources (CPU cycles, main memory, and file storage), may have a special allocation code
  - others (such as I/O devices) may have general request and release codes
- **Accounting** (**logging**) — Keeping track of which programs use how much and what kinds of computer resources. This is valuable for system administration where a system can be fine-tuned to improve performance.

- **Protection and security** — The owners of information stored in a multi-user or networked system must be able to control access to that information.
  - Concurrent processes should not interfere with each other or the operating system itself
  - **Protection** ensures that all access to system resources is controlled
  - **Security** requires authentication from external users. This extends to defending external I/O devices from invalid access attempts
  - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

2.1 User and Operating System Interface

There are many ways for users to interface with the operating system.

2.1.1 Command Interpreters

Most operating systems, including, Linux, UNIX, and Windows, treat command interpreters as a **special program** that is running when a process is initiated. On systems with multiple command interpreters, these are known as **shells**.

For example, on UNIX and Linux systems, users may choose among several shells including the **C shell**, **Bourne-Again shell**, **Korn shell**, and others.

The main function of a command interpreter is to fetch and execute user-specified commands. These commands can be implemented in two general ways:

- **Built-in commands** — Commands that are interpreted directly by the command interpreter and do not require the execution of another program.
- **System programs** — The command interpreter does not understand the command, and uses the command to identify a file to be loaded into memory and executed.

If the latter, adding new commands to the system is as simple as writing a new program, and modifying existing programs does not require shell modification.

2.1.2 Graphical User Interface

Rather than entering commands directly via a command-line interface, users can use the mouse, keyboard, and monitor to interact with images and icons on the screen (the desktop). Clicking mouse buttons may invoke additional actions that can provide information, display options, execute functions, open directories (folders), and so on.

Many systems now include both a CLI and GUI:

- **macOS** is implemented on the UNIX kernel, and provides an *Aqua* GUI and a command-line interface
- **Windows** provides a standard GUI and a CLI

- **UNIX and Linux** provide CLI shells with optional GUIs such as *K Desktop Environment* (KDE), or the GNOME desktop by the GNU project.

2.1.3 Touch Screen Interface

As a command-line or mouse-and-keyboard system is impractical for mobile systems, phones, tablets, and other mobile devices use a touch screen interface. These devices require the user to interact with the screen directly using gestures and a virtual keyboard.

2.2 System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, or sometimes Assembly.

These functions are typically accessed via a high-level **application programming interface** (API), rather than direct system calls.

Observe the following example of a system call sequence where we wish to copy a file's contents into another file:

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

2.3 Application Programming Interface

Even a simple program makes heavy use of the operating system. For this reason, application programmers design programs according to an **application programming interface** (API). This API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. On UNIX systems, we can use the `man` command to view the API for a given function.

Some examples of APIs include:

- the **Windows API** for Windows systems
- the **POSIX API** for UNIX, Linux, and macOS

A programmer accesses an API via a **library** of code, provided by the operating system. In the case of UNIX and Linux, programs written in the C language use a library called **libc**.

There are two main reasons for programming according to an API:

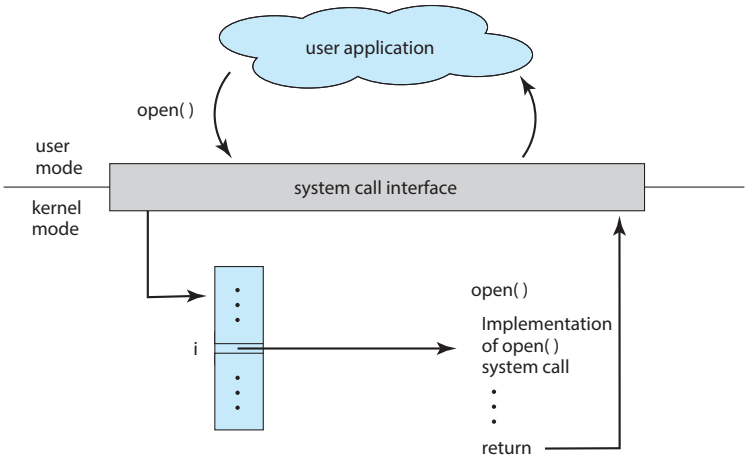
- **Program portability** — a program written to an API can be compiled on any system that supports that API
- **Ease of implementation** — making system calls manually may require more work and a deeper understanding of system functions

2.3.1 System Call Interface

An important factor in handling system calls is the **run-time environment** (RTE), which is a suite of software needed to execute applications written in a particular programming language, including compilers, interpreters, libraries and loaders. The RTE provides a **system-call interface** that serves as a link to system calls made available by the operating system.

The system-call interface **intercepts** function calls in the API and invokes the necessary system calls within the operating system. Typically a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system-call interface invokes the intended system call in the operating system kernel and returns the status of the system call with any return values.

Thus, the caller does not need to know anything about how the system call is implemented, rather it only needs to know obey the API and understand what the operating system will do as a result of this call. Below is an example of a user application invoking the `open()` system call:

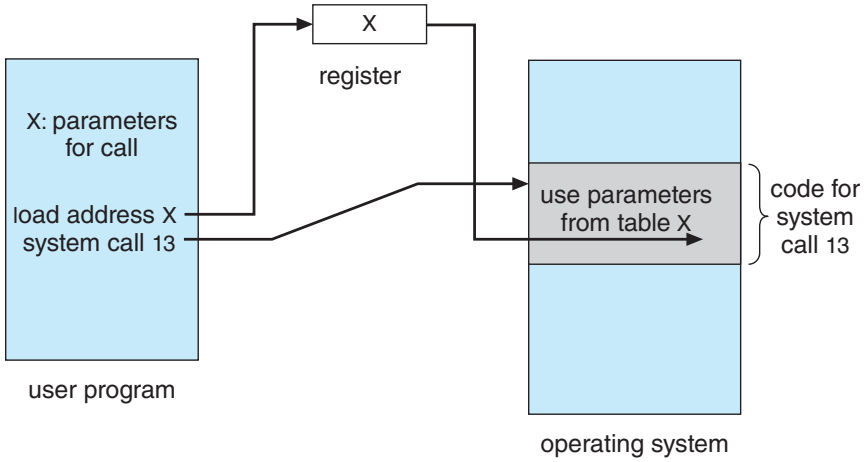


2.3.2 System Call Parameter Passing

Often more information is required than simply the identity of the desired system call. There are three general methods used to pass parameters to the operating system:

- **Registers** — pass parameters to registers
- **Blocks** — parameters are stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register
- **Stack** — parameters are pushed onto the stack, and popped by the operating system

The latter two approaches are preferred as they do not limit the number or length of parameters being passed.



2.4 Types of System Calls

System calls can be grouped into six major categories:

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

Some examples of these types of system calls are shown in the following sections.

2.4.1 Process Control

- create process, terminate process
- load, execute
- get process attributes, set process attributes
- wait event, signal event
- allocate and free memory

2.4.2 File Management

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes

2.4.3 Device Management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

2.4.4 Information Maintenance

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

2.4.5 Communications

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

2.4.6 Protection

- get file permissions
- set file permissions

2.4.7 Examples of Windows and UNIX System Calls

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

2.5 System Services

System services, also known as system programs or utilities, provide a convenient environment for program development and execution. Some are simply user interfaces to system calls, while others are considerably more complex. Most users' view of the operating system is defined by system programs, not the actual system calls.

- **File management**. Programs that create, delete, copy, rename, print, list, or access and manipulate, files and directories.
- **Status information**. Programs that ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Other programs provide detailed performance, logging, and debugging information. This information is typically

formatted and outputted to the user.

Some systems also support a **registry**, which is used to store and retrieve configuration information.

- **File modification**. Text editors may create and modify the content of files stored on disk. There may be special commands to search files or perform transformations of the text.
- **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available to download.
- **Program loading and execution**. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.

  Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They also allow users to send information to other machines.
- **Background services**. All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running processes are called services, subsystems, or daemons.

Along with these system programs, most operating systems are supplied with application programs that are designed to perform common operations.

2.6 Operating System Design and Implementation

2.6.1 Mechanisms and Policies

An important principle is the separation of **policy** from **mechanism**. Policies determine *what* will be done, and mechanisms determine *how* to do something.

The separation of policy and mechanism is important for **flexibility**. Policies are likely to change over time, and should not require a change in the underlying mechanism.

Microkernel based systems take this principle to the extreme, by only implementing a basic set of policy-free building blocks, so that more advanced mechanisms and policies can be added via user-created kernel modules.

In contrast, the Windows operating system and macOS, closely encode both mechanism and policy into the system to enforce a global look and feel across the system. All applications will therefore have similar interfaces, because the interface itself is built into the kernel and system libraries.

2.6.2 Implementation

Operating systems are typically implemented using several **high-level languages** such as C, C++, and assembly. While the kernel might be written assembly and C, higher-level routines and system libraries may be written using C++.

Some reasons for using higher-level languages are described below:

- Code is faster to write
- Code is more compact
- Code is easier to understand
- Code is easier to debug

In addition to this, improvements in compiler technology will improve the generated code for the entire operating system.

Another advantage is that operating systems can be **ported** to other hardware if they are written in a higher-level language. This is particularly important for operating systems intended to run on different hardware systems, such as embedded devices, Intel x86 systems, and ARM chips in mobile devices.

2.7 Operating System Structure

A system as large and complex as an operating system must be engineered carefully if it is to function properly and be modified easily. We will discuss the evolution of various structures in the following sections.

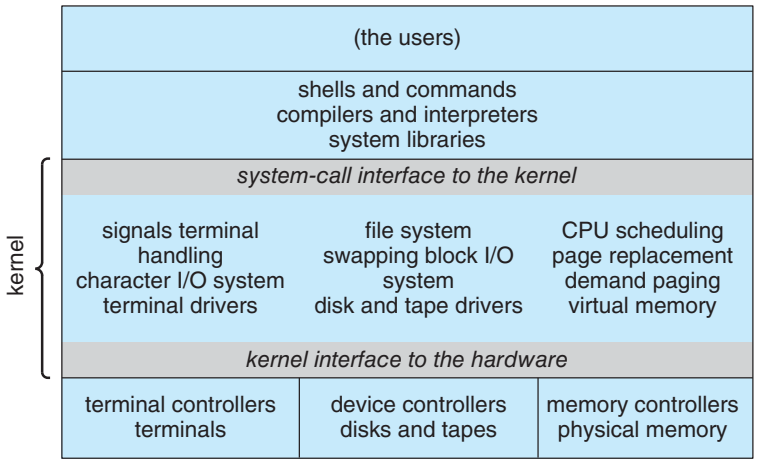
2.7.1 Monolithic Structures

The monolithic structure is the **simplest** structure for organising an operating system, as it uses **no structure** at all. All functionality of the kernel is placed into a single, static binary file, that runs in a single address space.

An example of such a structure is the original UNIX operating system, which consists of two separable parts:

- the kernel
- system programs

The kernel represents everything below the system-call interface and above the physical hardware. It provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.



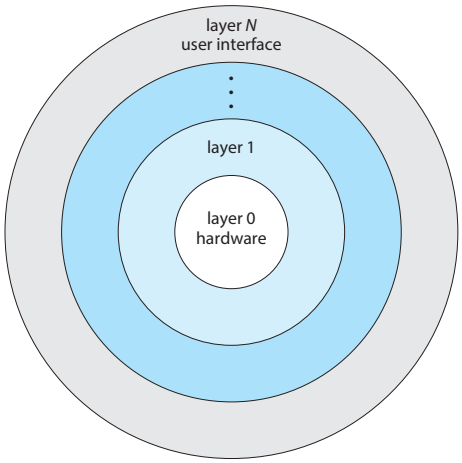
The Linux operating system is based on UNIX and is structured similarly. Applications typically use the `glibc` standard C library when communicating with the kernel (through the system-call interface). The Linux kernel is monolithic, but has a **modular design** that allows the kernel to be modified during runtime.

Monolithic kernels introduce very little overhead in the system-call interface, and communication within the kernel is very fast. Despite their simplicity, monolithic kernels are however difficult to implement and extend.

2.7.2 Layered Structures

The monolithic approach is a **tightly coupled** system because changes to one part of the system have widespread effects on other parts of the system. Alternatively, we may consider a **loosely coupled** system which is divided into separate, smaller components that have specific and limited functionality. All these components comprise the kernel.

One way to achieve modularity is to use a **layered approach**, where the operating system is divided into a number of layers (levels). The bottom layer (layer 0) is the hardware, and the highest (layer N) is the user interface. This is shown below.



Each layer uses functions (operations) and services of only **lower-level layers**. This simplifies debugging and system verification.

Layered systems are used in computer networks (such as TCP/IP) and web applications, however relatively few operating systems use a purely layered approach due to the challenges of appropriately defining the functionality of each layer.

2.7.3 Microkernel Structures

As UNIX expanded, the kernel became large and difficult to manage, and thus a system called **Mach** was developed to modularise the kernel using the **microkernel** approach. The Mac OS X kernel, Darwin, is a well-known example of a microkernel operating system based on Mach.

In a microkernel structure, **all nonessential components** are removed from the kernel and are instead implemented as **user-level programs** that reside in separate address spaces. Typically, microkernels provide minimal process and memory management, in addition to a communication facility, resulting in a small (micro) kernel.

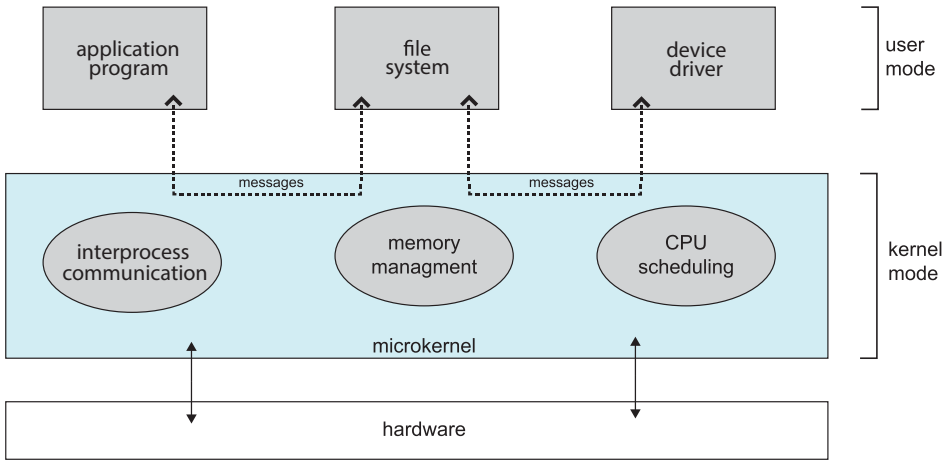
The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space, using **message passing**.

There are several benefits of using a microkernel system:

- Easy to extend — new services added to the user space rarely require modification to the kernel. Changes to the kernel are also smaller
- Easy to port to new architectures
- More reliable and secure — most services are running as user processes, rather than kernel processes

One of the main disadvantages of microkernels is the **performance overhead** associated with user space to kernel space communication. Communication between two user processes requires

messages to be copied twice; once from the sending process to the kernel, and again from the kernel to the receiving process.



2.7.4 Modules

Most modern operating systems use **loadable kernel modules** (LKMs). Here, the kernel has a set of core components and can link in additional services via modules, either at boot time, or during run time. This design is common to modern implementatios of UNIX, such as Linux, macOS, and Solaris, and also Windows.

In this structure, the kernel only provides core services, while other services are implemented **dynamically**, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, as the latter requires recompiling the kernel every time a change is made.

The overall result resembles a layered system as each kernel section has defined, protected interfaces; however, it is more flexible than a layered system, as any module can call any other module.

This approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient as modules do not need to invoke message passing in order to communicate.

Linux uses loadable kernel modules, primarily for device drivers. LKMs can be “inserted” into the kernel as the system is **booted** or during run time. If a device is not present, it can be dynamically loaded.

2.7.5 Hybrid Structures

Most modern systems combine several of the above approaches, resulting in hybrid systems that address performance, security, and usability issues. For example,

- Linux is monolithic to provide performance, and modular to allow dynamic loading of kernel services

- Windows is also monotlithic, but retains the behaviour of a microkernel system, including support for separate subsystem *personalities* that run as user-mode processes
- Mac OS X is based on the Mach microkernel, and includes dynamically loadable modules called **kernel extensions**