Systems Programming

Semester 2, 2023

Dr Timothy Chappell

Tarang Janawalkar

Contents

Contents **1**

1 Operating Systems Overview **5**
1.1 What Operating Systems Do . . . 5
1.2 Computer System Organisation . . . 6
1.2.1 Device Controllers . . . 6
1.3 Computer Operation . . . 6
1.4 Interrupts . . . 7
1.4.1 Implementation . . . 7
1.4.2 Types of Interrupts . . . 7
1.5 Storage Structure . . . 8
1.5.1 Main Memory . . . 8
1.5.2 Registers . . . 8
1.5.3 Cache Management . . . 8
1.5.4 Secondary Storage . . . 9
1.5.5 Tertiary Storage . . . 10
1.5.6 Summary . . . 10
1.6 I/O Structure . . . 11
1.6.1 Direct Memory Access . . . 11
1.7 Computer-System Architecture . . . 12
1.7.1 Single-Processor Systems . . . 12
1.7.2 Multiprocessor Systems . . . 12
1.7.3 Clustered Systems . . . 17
1.8 Operating System Operations . . . 18
1.8.1 Computer Startup . . . 18
1.8.2 Multiprogramming . . . 18
1.8.3 Multitasking . . . 19
1.9 Dual-Mode Operation . . . 19
1.10 Multi-Mode Operation . . . 20
1.11 Timers . . . 20

2 Operating System Structures **21**
2.1 User and Operating System Interface . . . 22
2.1.1 Command Interpreters . . . 22
2.1.2 Graphical User Interface . . . 22
2.1.3 Touch Screen Interface . . . 23
2.2 System Calls . . . 23
2.3 Application Programming Interface . . . 23
2.3.1 System Call Interface . . . 24
2.3.2 System Call Parameter Passing . . . 25
2.4 Types of System Calls . . . 25
2.4.1 Process Control . . . 26
2.4.2 File Management . . . 26
2.4.3 Device Management . . . 26

2.4.4 Information Maintenance . . . 26
2.4.5 Communications . . . 26
2.4.6 Protection . . . 27
2.4.7 Examples of Windows and UNIX System Calls . . . 27
2.5 System Services . . . 27
2.6 Operating System Design and Implementation . . . 28
2.6.1 Mechanisms and Policies . . . 28
2.6.2 Implementation . . . 29
2.7 Operating System Structure . . . 29
2.7.1 Monolithic Structures . . . 29
2.7.2 Layered Structures . . . 30
2.7.3 Microkernel Structures . . . 31
2.7.4 Modules . . . 32
2.7.5 Hybrid Structures . . . 33

3 Processes 33
3.1 Process Concept . . . 33
3.1.1 The Process . . . 33
3.1.2 Process State . . . 35
3.1.3 Process Control Block . . . 35
3.1.4 Threads . . . 36
3.1.5 Process Representation in Linux . . . 36
3.2 Process Scheduling . . . 36
3.2.1 Scheduling Queues . . . 36
3.2.2 CPU Scheduling . . . 38
3.2.3 Context Switch . . . 38
3.3 Operations on Processes . . . 39
3.3.1 Process Creation . . . 39
3.3.2 Process Termination . . . 42
3.4 Interprocess Communication . . . 43
3.5 Shared Memory . . . 44
3.6 Message Passing . . . 45
3.6.1 Naming . . . 46
3.6.2 Synchronisation . . . 47
3.7 POSIX Shared Memory . . . 47
3.7.1 Pipes . . . 49

4 Threads 51
4.1 Overview . . . 51
4.1.1 Motivation . . . 51
4.1.2 Benefits . . . 52
4.2 Multicore Programming . . . 52
4.2.1 Programming Challenges . . . 53
4.3 Multithreading Models . . . 53
4.3.1 Many-to-One Model . . . 54
4.3.2 One-to-One Model . . . 54

4.3.3 Many-to-Many Model . . . 55
4.3.4 Two-Level Model . . . 55
4.4 Thread Libraries . . . 55
4.4.1 Pthreads . . . 56
4.5 Implicit Threading . . . 57
4.5.1 Thread Pools . . . 58
4.5.2 OpenMP . . . 58
4.5.3 Grand Central Dispatch . . . 59
4.6 Threading Issues . . . 59
4.6.1 |fork()| and |exec()| System Calls . . . 60
4.6.2 Signal Handling . . . 60
4.6.3 Thread Cancellation . . . 60
4.6.4 Thread-Local Storage . . . 61
4.6.5 Scheduler Activations . . . 61

5 Synchronisation 61
5.1 Background . . . 62
5.2 The Critical-Section Problem . . . 63
5.3 Peterson's Solution . . . 64
5.4 Hardware Support for Synchronisation . . . 65
5.4.1 Atomic . . . 66
5.5 Mutex Locks . . . 66
5.6 Semaphores . . . 67
5.6.1 Precedence . . . 68
5.6.2 Non-Busy Waiting . . . 68
5.6.3 Deadlock and Starvation . . . 69
5.6.4 Semaphore Problems . . . 69
5.7 Monitors . . . 70
5.8 Classical Problems of Synchronisation . . . 70
5.8.1 The Bounded-Buffer Problem . . . 71
5.8.2 The Readers-Writers Problem . . . 71
5.8.3 The Dining-Philosophers Problem . . . 72

6 Safety Critical Systems 74
6.1 Safety Critical Software . . . 74
6.2 Dependability . . . 74
6.3 Achieving Safety . . . 75
6.4 Risk Analysis . . . 75
6.4.1 Hazard and Risk . . . 75
6.4.2 Faults and Failures . . . 75
6.5 Safety Standards . . . 76
6.6 Real-Time Operating Systems . . . 76
6.7 DO-178B Standard . . . 77
6.7.1 Software Planning Process . . . 77
6.7.2 Software Development Process . . . 77
6.7.3 Software Verification Process . . . 78

6.7.4 Software Configuration Management Process . . . 78
6.7.5 Software Quality Assurance Process . . . 78
6.8 Programming Standards . . . 79
6.8.1 MISRA C . . . 79

7 Distributed Systems 79
7.1 Advantages of Distributed Systems . . . 79

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.

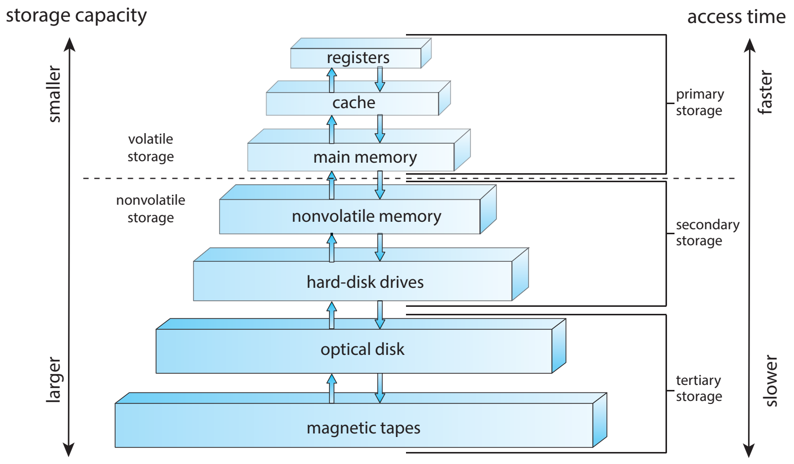
Programs are stored in secondary storage until they are loaded into memory. The most common forms of secondary storage are **hard-disk drives** (HDDs) and **nonvolatile memory** (NVM) **devices**.

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

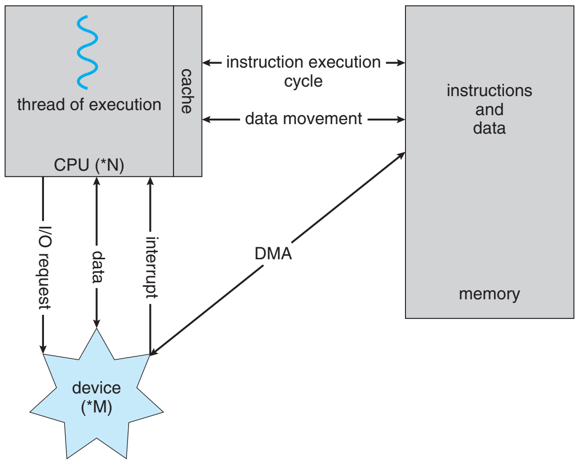
I/O is structured in one of two ways:

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.

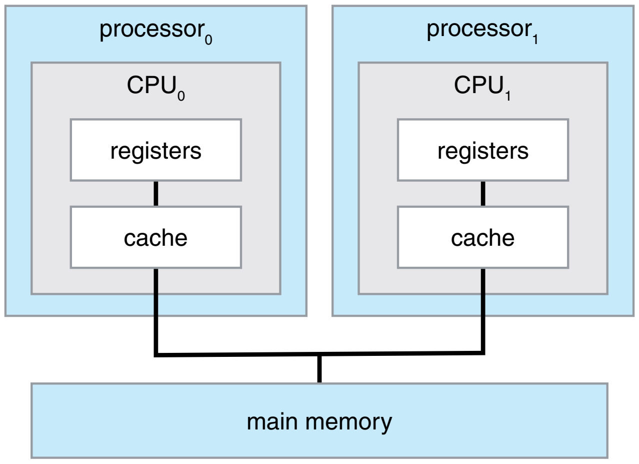
These systems also have other **special-purpose** processors such as device-specific processors for disk, keyboard, and graphics controllers. These processors run a limited instruction set and cannot run processes.

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**.

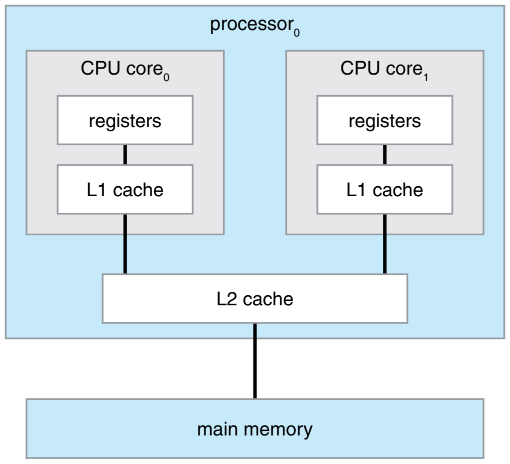
The benefit of this model is that many processes can be run **simultaneously**, without performance degradation. However, since CPUs are separate, one may be idle which another is busy, resulting in inefficiencies.

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.

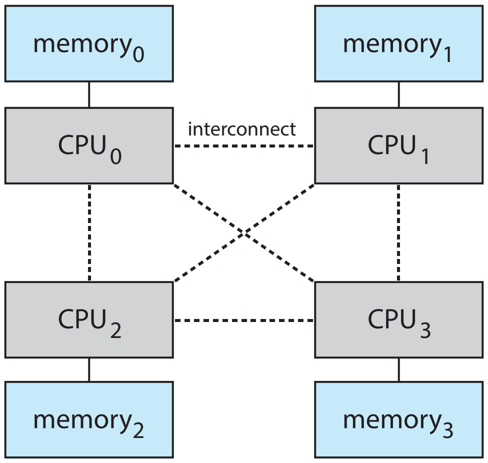
Each core has its own local cache, known as **level-1**, or **L1 cache**. In addition to this, each core shares a **level-2**, or **L2 cache**, that is local to the chip. Lower levels of cache are generally smaller, but have faster access times than higher level shared caches.

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

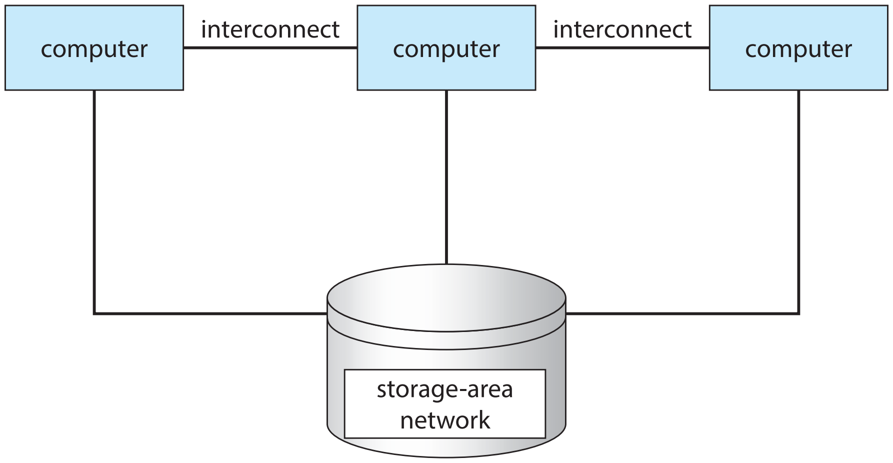
Due to their excellent scalability, NUMA systems are very popular on servers and high-performance computing systems.

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:

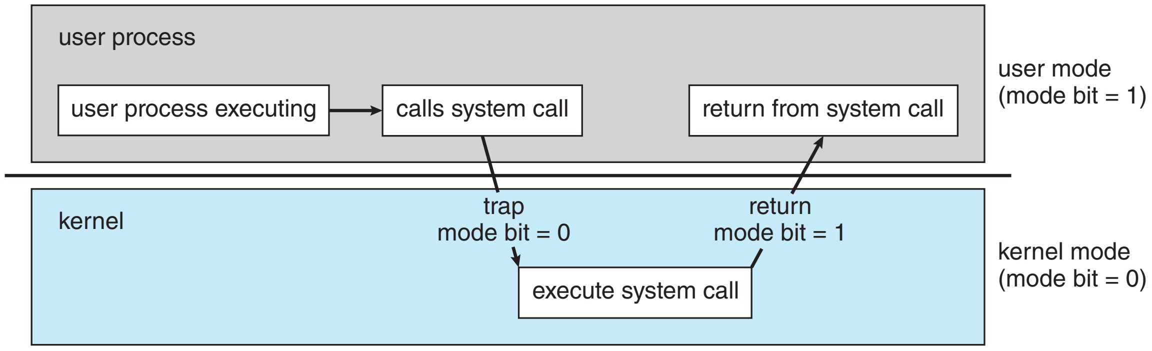
- **User mode** — used for executing user-defined code, and restricts direct access to hardware and special instructions.
- **Kernel mode** — used for executing operating system code, and allows direct access to hardware and all privileged instructions.

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:

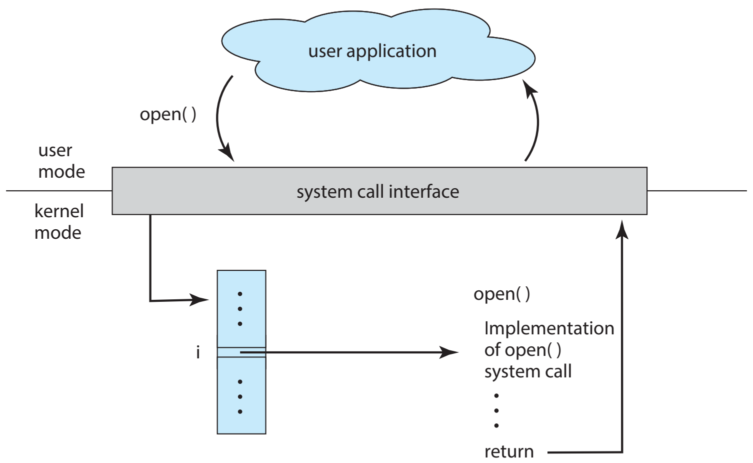
- **Program portability** — a program written to an API can be compiled on any system that supports that API
- **Ease of implementation** — making system calls manually may require more work and a deeper understanding of system functions

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:

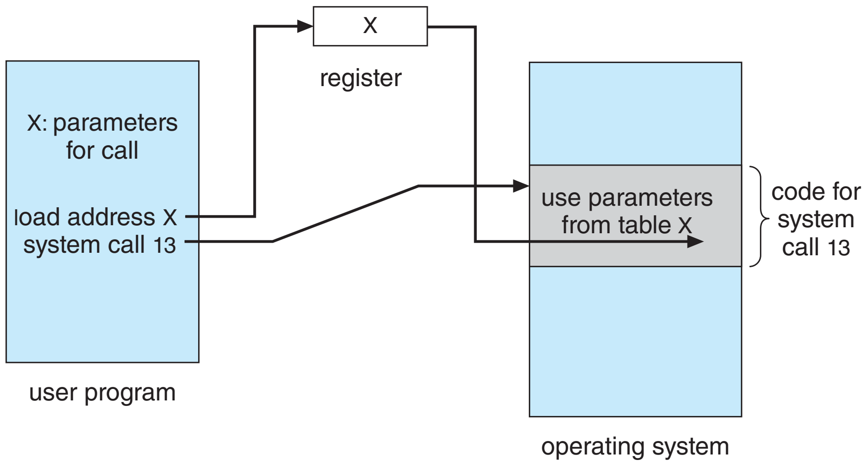


2.3.2 System Call Parameter Passing

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.

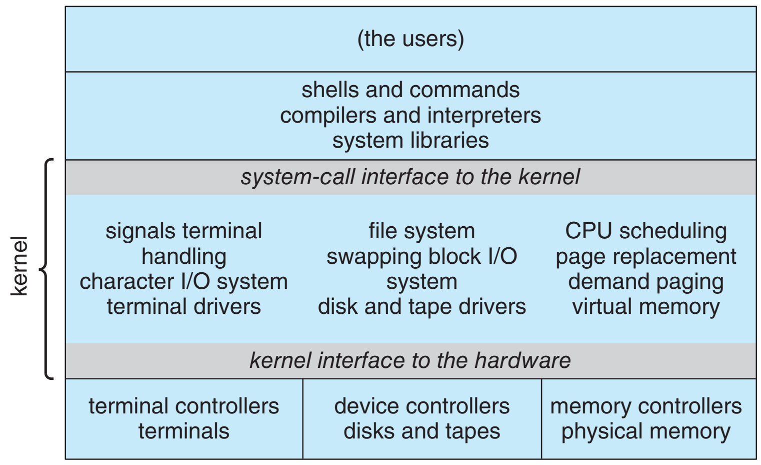
2.7.1 Monolithic Structures

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.

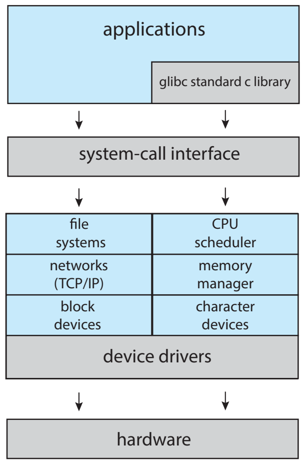
An example of such a structure is the original UNIX operating system, which consists of two separable parts:

- 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.

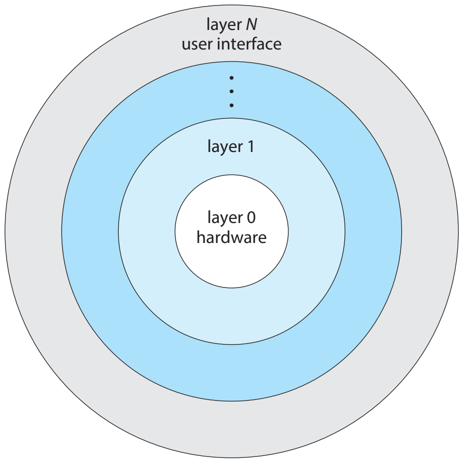


Monolithic kernels introduce very little overhead in the system-call interface, and communication within the kernel is very fast. Despite their simplicity, monolithic kernels are however difficult to implement and extend.

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.

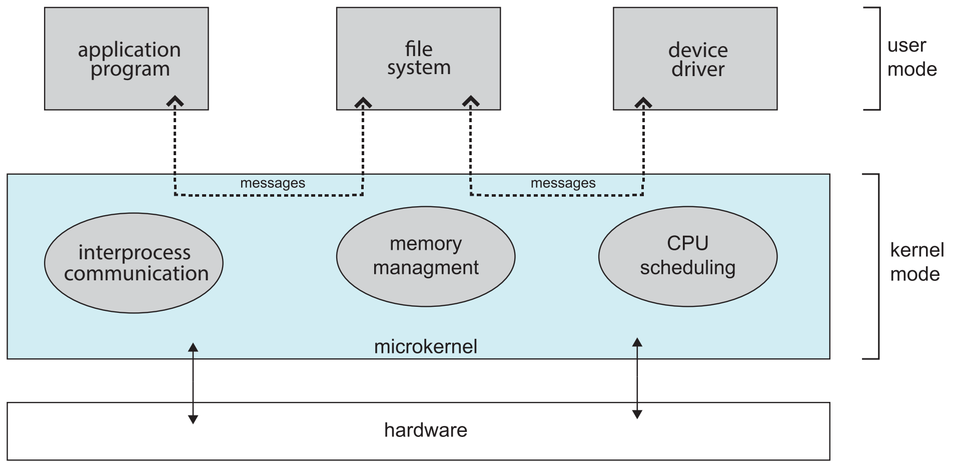
The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space, using **message passing**.

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**

3 Processes

A **process** is a program in execution, and is the unit of work in a modern computing system. Processes need resources to accomplish their task, including CPU time, memory, files, and I/O devices. These are typically allocated to processes while it is executing.

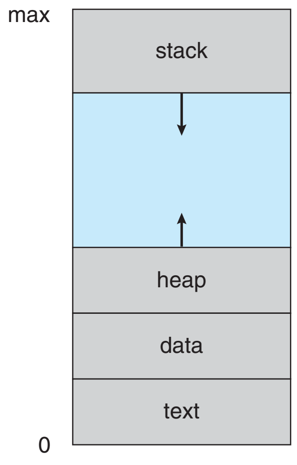
Modern operating systems support processes having multiple **threads** of control. On systems with multiple hardware processing cores, these threads can run in **parallel**. An important aspect of an operating system is how it **schedules** threads onto available processing cores.

3.1 Process Concept

Early computers were batch systems that executed **jobs**, followed by time-shared systems that ran **user programs** or **tasks**. On single-user systems, a user may be able to run several programs at one time, and even if that is not possible, the operating system may need to support internal programmed activities, such as memory management. Thus, the concept of a **process** was introduced to encapsulate these activities.

3.1.1 The Process

A process is a program in execution. The status of the current activity of a process is represented by the value of the **prorgam counter** and the contents of a processor's registers. The memory layout of a process is divided into multiple sections, as shown in the figure below.



To summarise, these sections include:

- **Text** — the executable code
- **Data** — global variables
- **Heap** — memory dynamically allocated during run time
- **Stack** — temporary data storage such as function parameters, return addresses, and local variables

Notice how the sizes of the text and data sections are **fixed**, while the sizes of the heap and stack sections may **shrink and grow dynamically** during program execution.

When a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack. When control is returned from the function, this activation record is popped off the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system.

The operating system must ensure that the stack and heap do not **overlap** in memory.

It is important to note that a program itself is not a process, but rather a **passive entity**, such as an **executable file**, that contains a list of instructions stored on disk. A process is an **active entity** with a program counter specifying the next instruction to execute, and a set of associated resources.

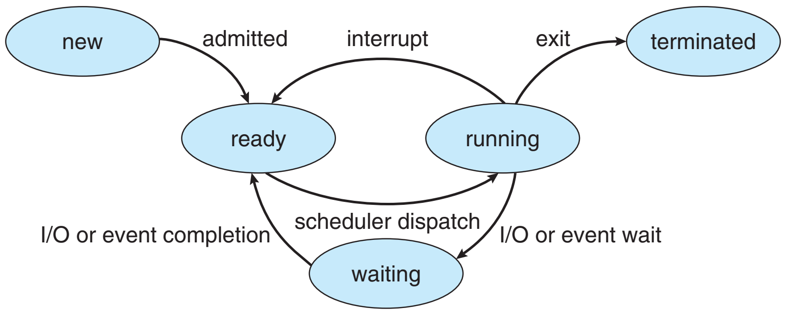
A **program** *becomes* a process when an executable file is loaded into memory, either through a GUI (i.e., mouse double-click), or through a command line interface. While **multiple processes** may be associated with the **same program**, these are considered as **separate** execution sequences, as they may have different data, heap, and stack sections. A process may also **spawn** many other processes as it runs.

3.1.2 Process State

As a process executes, it changes **state** according to its current activity:

- **New** — the process is being created
- **Running** — instructions are being executed
- **Waiting** — the process is waiting for some event to occur (such as an I/O completion or reception of a signal)
- **Ready** — the process is waiting to be assigned to a processor
- **Terminated** — the process has finished execution

It is important to realise that only one process can be **running** on any processor at any instant. A state diagram corresponding to these states is shown below:



3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (PCB) (also called a **task control block**). It contains many pieces of information associated with a specific process, including:

- **Process state** — new, ready, running, waiting, halted
- **Program counter** — the address of the next instruction to be executed
- **CPU registers** — contents of all process-centric registers (accumulators, index registers, stack pointers, etc.)
- **CPU-scheduling information** — process priority, pointers to scheduling queues, etc.
- **Memory-management information** — memory allocated to the process: base and limit registers, page tables, segment tables
- **Accounting information** — amount of CPU and real time used, time limits, account numbers, job or process numbers, etc.
- **I/O status information** — devices allocated to the process, list of open files, etc.

To summarise, the PCB serves as the **repository** for all information needed to start, or restart, a process.

3.1.4 Threads

The process model described above implies that a process is a program that performs a **single thread of execution**. This single thread of control allows the process to perform only one task at a time. Most modern operating systems have extended the process model to allow a process to have multiple threads of execution, and thus perform more than one task at a time. This is especially beneficial on multicore systems, where multiple threads can run in parallel. On such systems, the PCB is expanded to include information for each thread.

3.1.5 Process Representation in Linux

In Linux, a process is represented by a `task_struct` structure, which is defined in the `<linux/sched.h>` header file. This structure contains a large number of fields, including:

```
long state;                  /* state of the process */
struct sched entity se;      /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children;  /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm;        /* address space */
```

3.2 Process Scheduling

The objective of **multiprogramming** is to have some process running at all times, to maximise CPU utilisation. The objective of **time sharing** is to switch a CPU core among processes so frequently that users can interact with each program while it is running.

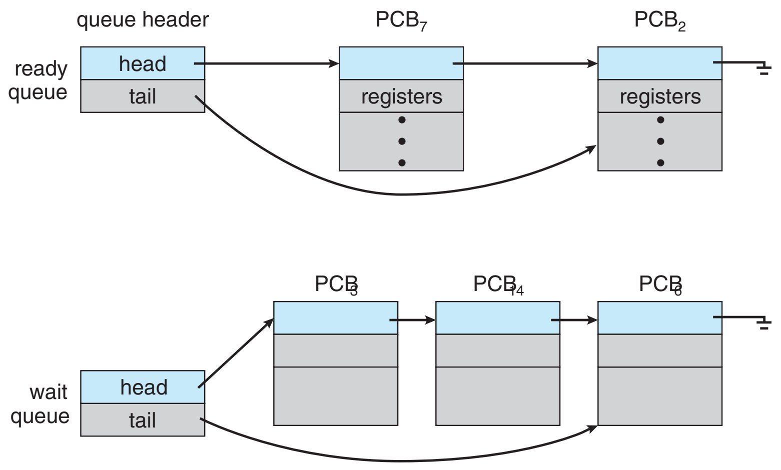
To meet these objectives, the **process scheduler** select an available process for execution on a core. Each CPU core runs one process at a time. If there are more processes than cores, the scheduler must wait until a core is free and can be rescheduled. The number of processes currently in memory is known as **degree of multiprogramming**

Balancing these objectives requires taking the general behaviour of the process into account.

- A process that spends more time doing I/O, is an **I/O bound process**
- A process that spends more time doing computation, is an **CPU bound process**

3.2.1 Scheduling Queues

The process scheduler maintains a set of **scheduling queues** for processes awaiting allocation of CPU time. These queues are typically implemented as **linked lists** of PCBs, with each queue having its own priority.



These queues include:

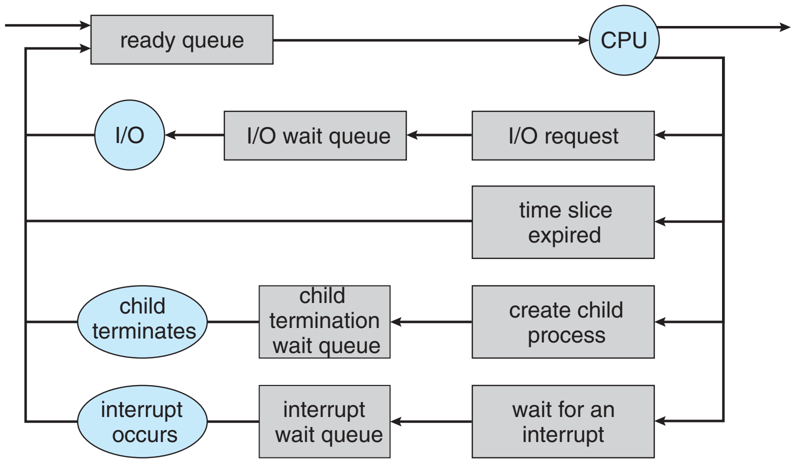
- **Job queue** — The set of **all** processes in the system
- **Ready queue** — As processes enter the system, they are placed in this queue which resides in main memory, where they are ready and waiting to execute
- **Wait queue** — The set of processes waiting for the occurrenece of an event, such as I/O completion

Processes are initially placed in the **ready queue** where they wait to be selected by scheduler, or are **dispatched**. When a processes is allocated to a CPU core, and is executing, one of several events may occur:

- The process may issue an I/O request and be placed in the *wait queue*
- The process may create a new child process and be placed in the *wait queue* while awaiting the child's termination
- The process may be removed forcibly from the CPU by the scheduler, either due to an interrupt or because that process's time slice has expired, and be placed back in the *ready queue*

In the first two cases, the process eventually returns to the ready queue. A process continutes this cycle until it terminates, at which point, it is removed from all queues and has its PCB and resources deallocated.

This is shown in the queueing diagram below:



3.2.2 CPU Scheduling

Processes migrate between the ready and wait queues throughout their lifetimes. The role of the **CPU scheduler** is to select from among the processes that are in the ready queue, and allocate a CPU core to one of them. This is known as **CPU scheduling**.

These schedulers are invoked at a different rate,

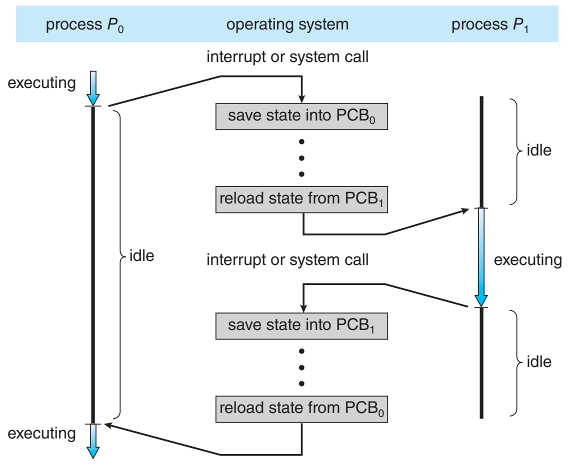
- The process scheduler is a **long-term scheduler**, as it is not invoked very frequently (seconds to minutes).
- The CPU scheduler is a **short-term scheduler**, as it is invoked frequently to prevent a single process from using a core for an extended period of time. The CPU scheduler often forcibly removes the CPU from a process every 100ms or faster.

Some operating systems have an intermediate form of scheduling known as **swapping**. This is a form of **medium-term scheduling**, where a process is removed from memory to reduce the degree of multiprogramming. This process can be reintroduced into memory later. A processes state is “swapped-out” from memory to disk, where its current status is saved, and later “swapped-in” to memory, with its status restored. This is useful when memory has been overcommitted and needs to be freed up.

3.2.3 Context Switch

When the CPU switches to another process, the kernel performs a **state save** of the **current context** of the running process, then perform a **state restore** of that context when processing is complete. The **context** of a process is represented in the PCB of that process.

This process is known as a **context switch**.



Context switch time is pure overhead, as the system does no useful work while switching. More complex operating systems may require more information to be stored during a context switch, requiring more advanced memory-management techniques to be implemented.

3.3 Operations on Processes

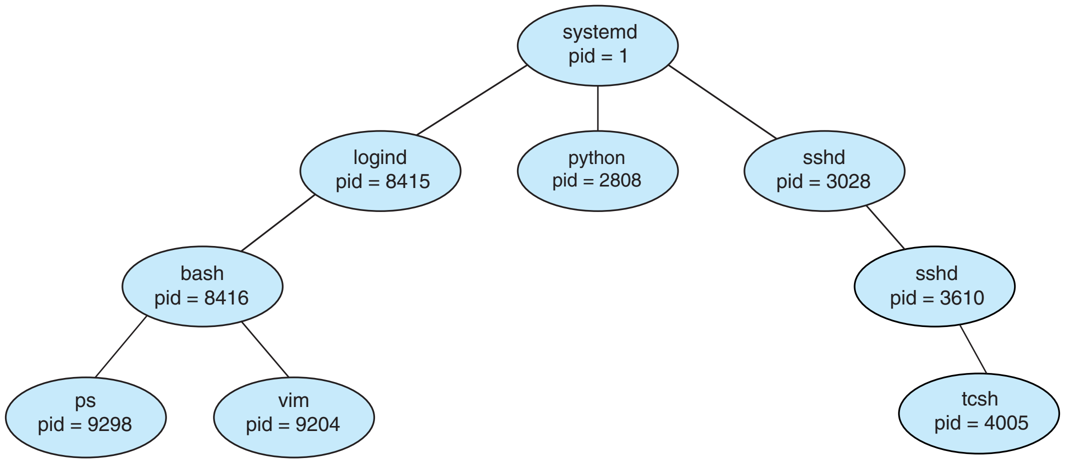
The processes in most systems can execute concurrently, and may be created and deleted dynamically. These systems must provide a mechanism for process creation and termination.

3.3.1 Process Creation

During execution, a process may create several new processes. The creating process is called the **parent process** and the new processes are known as **child processes**. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes using a **process identifier** (pid), represented by an integer. The pid is unique for each process in the system, and can be used as an index to access various attributes of a process within the kernel.

In Linux, the `systemd` process is the first process to be created, and has a pid of 1. This process is responsible for starting all other processes in the system, such as the `logind` and `sshd` processes.



In general, when a process creates a child process, the child will need certain resources to accomplish its task. This process may be able to obtain these resources directly from the operating system, or be constrained to a subset of the parent's resources. In addition to supplying physical and logical resources, the parent process may pass initialisation data (input) to the child process.

When a process creates a new process, two possibilities for execution exist:

- The parent continues to execute concurrently with its children
- The parent waits until some or all of its children have terminated

There are also two address-space possibilites for the new process:

- The child process is a duplicate of the parent process (both share the same data)
- The child process has a new program loaded into it

In UNIX, the `fork()` system call is used to duplicate the calling process and create a new process. This allows the parent process to easily communicate with its child process. The child may then use the `exec()` system call to replace its memory space with a new program.

The parent process can either create more children, or issue a `wait()` system call to wait for a child to terminate.

When the `fork()` system call is invoked, the function returns the child processes pid to the parent process, but returns 0 to the child process. This allows us to distinguish between the parent and child process.

Consider the following code:

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
```

```
 4  #include <unistd.h>
 5
 6  int main()
 7  {
 8      pid_t pid;
 9
10      /* fork a child process */
11      pid = fork();
12      // the parent process receives the child's pid
13      // the child process receives 0
14
15      if (pid < 0)
16      { /* error occurred */
17          fprintf(stderr, "Fork failed\n");
18          return 1;
19      }
20      else if (pid == 0)
21      { /* child process */
22          printf("Child:  fork() returned %5u PID %5u\n", pid, getpid());
23          execl("/usr/bin/ls", "ls", NULL);
24          printf("This line is never executed\n");
25      }
26      else
27      { /* parent process */
28          /* wait for children to terminate */
29          wait(NULL);
30          printf("Parent: fork() returned %5u PID %5u\n", pid, getpid());
31      }
32
33      return 0;
34  }
```

This program forks the parent process on line 11, and then waits for the child process to terminate. This program outputs the following:

```
Child:  fork() returned     0 PID 15515
create_process.c  create_process
Parent: fork() returned 15515 PID 15514
```

After the child process is created, the `exec()`[2] system call is used to replace the child's memory space with the `ls` program, causing the second print statement to not be executed.

The parent process can use the `wait()` system call to remove itself from the ready queue until the termination of the child process, after which the parent process can resume from where it left off.

[2]Note that `exec()` is often used to refer to a family of functions in C. While these functions refer to the same system call, they differ in the arguments they can accept. See the manpage for `exec` for more information.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement, and asks the operating system to delete it by using the `exit()` system call. At this point, the process may return a status value to its parent, which is used to determine if the child process completed successfully. All resources allocated to the process are then deallocated and reclaimed by the operating system.

A parent may terminate the execution of one or more of its children by using the `abort()` system call. This may be done for several reasons, including:

- The child process exceeded allocated resources
- The task assigned to the child is no longer required
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. As a result, all child processes are terminated — **cascading termination**

The following example demonstrates how a parent process might see the exit status of a child process.

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    pid = fork();

    if (pid < 0)
    { /* error occurred */
        fprintf(stderr, "Fork failed\n");
        return 1;
    }
    else if (pid == 0)
    { /* child process */
        // terminate the child process with exit code 1
        printf("Child PID %5u\n", getpid());
        exit(1);
    }
    else
    { /* parent process */
        // store the exit status of the child process
        int status;
        // return the pid of the child process that terminated
        pid = wait(&status);

        printf("PID %5u exited with status 0x%x\n", pid, status);

        // the exit code is located in the most significant byte of status
        printf("Exit code: %d\n", WEXITSTATUS(status));
    }

    return 0;
}
```

This program outputs the following:

```
Child PID 28453
PID 28453 exited with status 0x100
Exit code: 1
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain until the parent calls `wait()`, because the process table contains the process's exit status.

- A process that has terminated but whose parent has not yet called `wait()` is known as a **zombie process** All processes transition to this state when they terminate, but only for a short time.
- If a parent terminates without calling `wait()`, the child processes are **orphaned**. In this scenario, the operating system assigns the `systemd` process as the parent of any orphaned processes.

3.4 Interprocess Communication

Processes executing concurrently may either be **independent** or **cooperating** processes. Independent processes do not share data with other processes that are executing in the system, whereas cooperating processes can affect or be affected by other processes that are executing in the system.

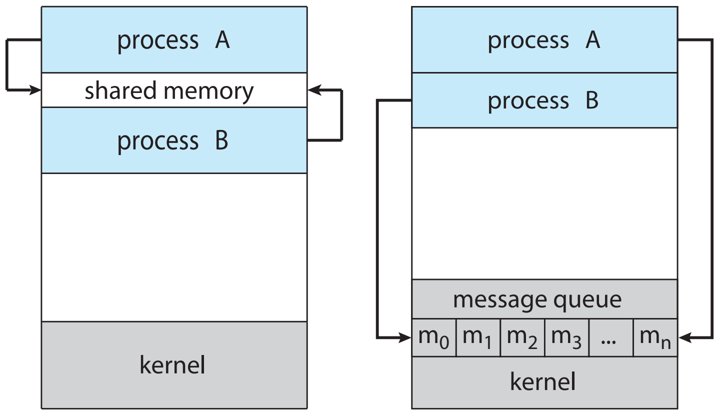
There are several reasons for providing an environment that allows process cooperation:

- **Information sharing** — processes may access the same piece of information
- **Computation speedup** — tasks may be subdivided and executed in parallel on multiple cores
- **Modularity** — dividing a system function into separate processes allows different parts of the system to be implemented independently

Cooperating processes require an **interprocess communication** (IPC) mechanism that allows them to exchange data and information. There are two models of IPC:

- **Shared memory** — processes exchange information by reading and writing data to a shared region of memory

- **Message passing** — processes communicate with one another by exchanging messages



Message passing is useful for exchanging small amounts of data, without needing to avoid conflict. Shared memory can be faster than message passing, as all accesses are treated as routine memory accesses, and no assistance from the kernel is required, aside from establishing the shared memory region.

3.5 Shared Memory

A common paradigm for shared memory is the **producer-consumer** problem. A **producer** process *produces* information that is *consumed* by the **consumer** process. For example, a server might provide web content, which is read by a client web browser.

A solution to this shared memory problem is to use a **buffer** that is filled by the producer, and emptied by the consumer. In such a system, the producer and consumer must be **synchronised** to ensure the consumer does not try to consume data that has not yet been produced.

Two types of buffers can be used:

- **Unbounded buffer** — the buffer has no fixed size; the producer can keep producing data, the consumer may need to wait if it is reading faster than the producer is writing
- **Bounded buffer** — the buffer has a fixed size; the producer must wait if the buffer is full, and the consumer may need to wait if it is reading faster than the producer

In either case, at least one process is waiting for the other to either write data, or to read data.

The shared buffer can be implemented using a circular array with two pointers `in` and `out`.

```
#define BUFFER_SIZE 4

// Shared memory
```

```
int buffer[BUFFER_SIZE];

int in = 0;  // Index for producer
int out = 0; // Index for consumer

void produce(int *produced)
{
    while (1)
    {
        // Wait until consumer has consumed next index
        while (((in + 1) % BUFFER_SIZE) == out)
            ; // Buffer is full, do nothing

        // Buffer has space, produce item
        buffer[in] = *produced;

        // Update index
        in = (in + 1) % BUFFER_SIZE;
    }
}

void consume(int *consumed)
{
    while (1)
    {
        // Wait until producer has produced next index
        while (in == out)
            ; // Buffer is empty, do nothing

        // Buffer has item, consume it
        *consumed = buffer[in];

        // Update index
        out = (out + 1) % BUFFER_SIZE;
    }
}
```

As the producer cannot write to the buffer when the consumer points to the next location to be written, the total number of items the buffer can hold is `BUFFER_SIZE - 1`.

3.6 Message Passing

Message passing provides a mechanism to allow proesses to communicate and to synchronise their actions without sharing the same address space. It is particularly useful when communicating processes reside on different computers that are connected by a network.

A message passing facility provides two operations:

- `send(message)` — send a message to another process
- `receive(message)` — receive a message from another process

Messages sent by a process can be either **fixed** or **variable** in size. If only fixed-size messages are sent, the system-level implementation is more straightforward, however the task of programming the application becomes more difficult. Conversely, if variable-sized messages are sent, the system-level implementation is more complex, but the task of programming the application becomes easier.

If processes P and Q wish to communicate, they must

- establish a **communication link** between them
- exchange messages via **send/receive** primitives

These links can be implemented in a variety of ways:

- physically — shared memory, hardware bus
- logically — direct or indirect communication, synchronous or asynchronous communication, automatic or explicit buffering

3.6.1 Naming

Processes that want to communicate must be able to refer to each other. This can be done either **directly** or **indirectly**.

Under direct communication, each process must explicity name the recipient or sender of the communication:

- `send(P, message)` — send a message to process P
- `receive(Q, message)` — receive a message from process Q

In this scheme, a link is established between each pair of processes that wish to communicate. This allows for a **symmetric** or **asymmetric** link to be established. The above example uses symmetric addressing, but it is possible for the recipient to not name the sender:

- `send(P, message)` — send a message to process P
- `receive(id, message)` — receive a message from any process; the variable *id* is set to the name of the sender

The disadvantage of direct communication is the limited modularity of the resulting system. If a process changes identifier, it may be necessary to update all processes that communicate with it.

In contrast, **indirect communication** allows messages to be passed to and from **mailboxes** or **ports**. Messages can be placed into mailboxes and received by other processes that have access to this shared mailbox. The `send()` and `receive()` primitives are then extended to include the name of the mailbox:

- `send(A, message)` — send a message to mailbox A
- `receive(A, message)` — receive a message from mailbox A

In this method, we must decide how many processes can read from a mailbox, or if they must take turns.

3.6.2 Synchronisation

Communication can be either **synchronous** or **asynchronous**, also known as **blocking** and **non-blocking** respectively. In relation to message passing, this means:

- **Blocking send**. The sending process must wait until the sent message is received by the receiving process
- **Non-blocking send**. The sending process can send a message and resume operation immediately
- **Blocking receive**. The receiving process must wait until a message is available
- **Non-blocking receive**. The receiving process can receive a valid message or null

If both send and receive are blocking, there is a **rendezvous** between the sender and receiver.

3.7 POSIX Shared Memory

This section explores a mechanism for shared memory provided by the POSIX API. POSIX shared memory is organised using **memory-mapped** files that associate a region of shared memory with a file. A process must first create a shared-memory object using the `shm_open()` system call:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fcntl.h>

#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;

    /* name of the shared memory object */
    const char *name = "OS";

    /* strings written to shared memory */
    const char *message = "Hello";

    /* shared memory file descriptor */
    int fd;

    /* pointer to shared memory object */
    char *ptr;

    /* create the shared memory object */
    // params:
    // - name of the shared memory object
    // - flags - create the shared memory object if it does not exist,
    //           allow reading and write
    // - file-access permission
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);

    /* establish a memory-mapped file containing the shared memory object */
    // params:
    // - start address
    // - length of the mapping
    // - memory protection - page can be read and written to
    // - flags - mapping is shared
    // - file descriptor
    // - offset from the beginning of the file
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* write to the shared memory object */
    sprintf(ptr, "%s", message);
    ptr += strlen(message);

    return 0;
}
```

The consuming process can then read the contents of this shared memory using the code below.

```c
#include <stdio.h>
#include <stdlib.h>

#include <fcntl.h>

#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
```

```

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;

    /* name of the shared memory object */
    const char *name = "OS";

    /* shared memory file descriptor */
    int fd;

    /* pointer to shared memory object */
    char *ptr;

    /* open the shared memory object */
    // params:
    // - name of the shared memory object
    // - flags - allow reading only
    // - file-access permission
    fd = shm_open(name, O_RDONLY, 0666);

    /* establish a memory-mapped file containing the shared memory object */
    // params:
    // - start address
    // - length of the mapping
    // - memory protection - page can be read and written to
    // - flags - mapping is shared
    // - file descriptor
    // - offset from the beginning of the file
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

3.7.1 Pipes

Pipes allow two processes to communicate with each other through message passing. The producer writes to the **write end** of the pipe, and the consumer reads from the **read end** of the pipe. As a result, pipes are **unidirectional**, and only allow one-way communication. Below is an example of a pipe being used to communicate between a parent and child process.

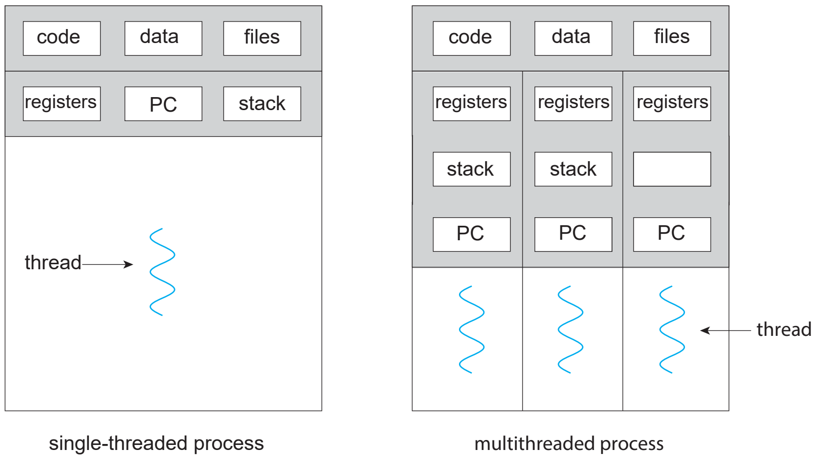
```c
#include <stdio.h>
#include <string.h>

#include <sys/types.h>
#include <unistd.h>

#define BUFFER_SIZE 25

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Hello";
    char read_msg[BUFFER_SIZE];

    int fd[2]; // fd[0] - read, fd[1] - write
    pid_t pid;

    if (pipe(fd) == -1)
    {
        fprintf(stderr, "Pipe failed");
        return 1;
    }

    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork failed");
        return 1;
    }

    if (pid > 0) // parent process
    {
        close(fd[0]); // close read end

        // write to pipe
        write(fd[1], write_msg, strlen(write_msg) + 1);
        printf("Writing: \"%s\" to pipe.\n", write_msg);

        close(fd[1]); // close write end
    }
    else // child process
    {
        close(fd[1]); // close write end

        // read from pipe
        read(fd[0], read_msg, BUFFER_SIZE);
        printf("Reading \"%s\" from pipe.\n", read_msg);

        close(fd[0]); // close read end
    }

    return 0;
}
```

4 Threads

The process model described above implies that a process is a program with a **single thread of execution**. However, most modern operating systems allow processes to have multiple threads of execution. Identifying opportunities for parallelism in a program is a very important in multicore systems with multiple CPUs.

4.1 Overview

A **thread** is a basic unit of CPU utilisation, consisting of a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. If a process has multiple threads of control, it can perform more than one task at a time.



4.1.1 Motivation

Many modern applications are designed to leverage processing on multicore systems to perform multiple tasks in parallel across multiple computing cores. For example, a web server may have one thread to listen for incoming requests, and another thread to process these requests. Most operating system kernels are also multi-threaded.

As process creation is time consuming and resource intensive, it is more efficient to use one process with multiple threads.

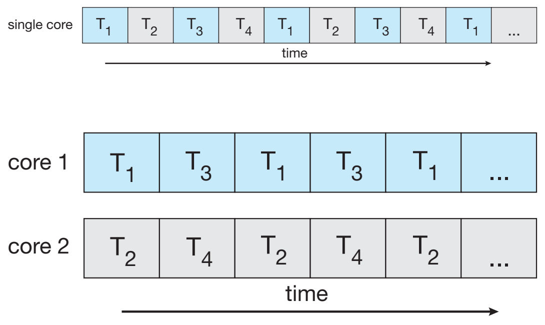
4.1.2 Benefits

There are several benefits to multi-threaded programming:

- **Responsiveness** — may allow continued execution if part of a process is blocked. This is especially important for user interfaces.
- **Resource sharing** — threads share resources of the process to which they belong, including memory and files. This is easier than using shared memory or message passing.
- **Economy** — creating and managing threads is generally faster than creating and managing processes. Context switching between threads incurs less overhead than between processes.
- **Scalability** — multi-threaded programs can take advantage of multiprocessor architectures, as threads can be run in parallel on different processing cores.

4.2 Multicore Programming

Consider the following application with four threads. On a system with a single computing core, concurrency is achieved by time slicing the CPU between the threads executing only one thread at a time. On a system with multiple cores, concurrency is achieved by running the threads in parallel on different cores, because the system can assign a separate thread to each core.



Note the distinction between **parallelism** and **concurrency**.

- A **concurrent** system supports more than one task by allowing all tasks to make process. On a single core system, the scheduler switches between tasks to achieve this.
- A **parallel** system can perform more than one task simultaneously.

Thus, a concurrent system does not imply it is also parallel. There are two types of parallelism:

- **Data parallelism** — distributes subsets of the same data across multiple cores, performing the same operation on each core.

- **Task parallelism** — distributes tasks (threads) across multiple computing cores, each of which performs a unique operation.

Note that these two types of parallelism are not mutually exclusive, and an application may use a combination of both.

4.2.1 Programming Challenges

Multicore systems introduce several challenges for programmers to make better use of multiple computing cores. Designers of oeprating systems must write their own scheduling altoritms that use multiple cores to allow parallel execution of threads. In general, there are five areas that present challenges:

- **Identifying tasks** — identifying tasks that can be divided into separate, concurrent tasks. Ideally, these tasks are independent of one another, and can run on individual cores.
- **Balance** — ensuring that all cores perform an equal amount of work. When one task contributes significantly less value than others, it may be necessary to use a separate execution core.
- **Data splitting** — dividing data into subsets that can be operated on in parallel.
- **Data dependency** — ensuring that tasks which access the same data are synchronised.
- **Testing and debugging** — testing and debugging multi-threaded programs is more difficult than single-threaded programs.

Amdahl's Law

Amdahl's Law is a formula that identifies performance gains from adding additional cores, to an application that has both serial and parallel components. If S is the fraction of the application that must be performed serially on a system with N cores, then

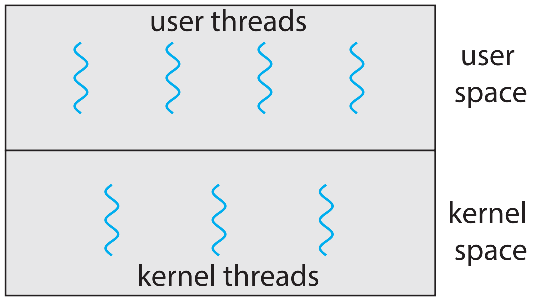
$$\text{speedup} = \frac{1}{S + \frac{1-S}{N}}.$$

4.3 Multithreading Models

Support for threads can be provided either at the user level, for **user threads**, or at the kernel level, for **kernel threads**. User threads are managed by user-level threads libraries such as:

- POSIX Pthreads
- Windows threads
- Java threads

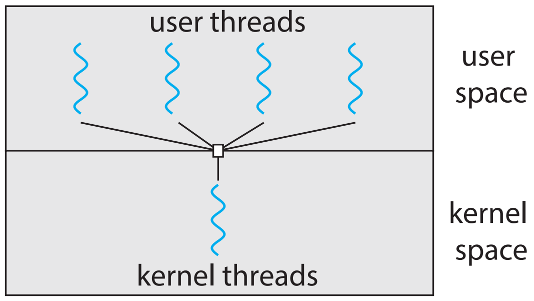
Kernel threads are managed directly by the operating system, and this is the case for most modern operating systems, such as Windows, Linux, and macOS.



For the user space to utilise multiple cores, a relationship must be established between user threads and kernel threads. The following sections will explore various models that achieve this.

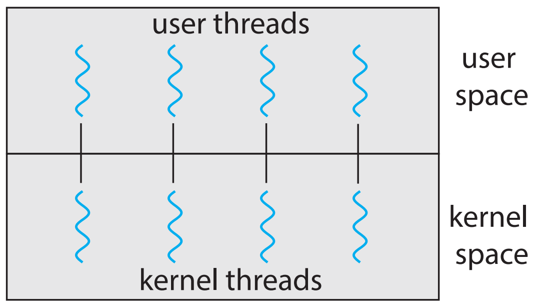
4.3.1 Many-to-One Model

The many-to-one model maps many user-level threads to a single kernel thread. Thread managemen is done by the thread library in user space, so it is efficient. However, an entire process will be blocked if a thread makes a blocking system call. Additionally, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.



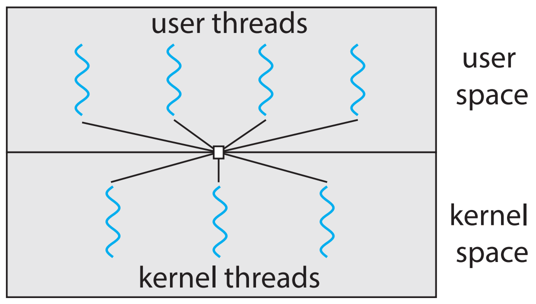
4.3.2 One-to-One Model

The one-to-one model maps each user thread to a kernel thread, and is the most common approach. This allows multiple threads to run in parallel on multicore systems. However, creating a user thread requires creating a kernel thread, which is expensive and thus a large number of kernel threads may affect performance.



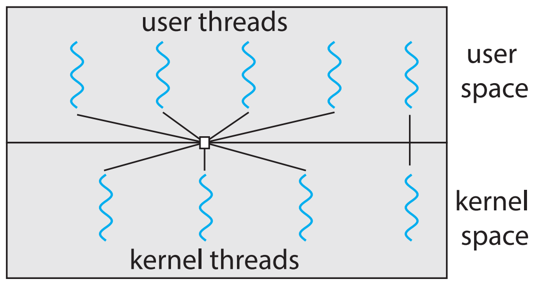
4.3.3 Many-to-Many Model

The many-to-many model multiplexes any number of user threads to an equal or smaller number of kernel threads. This model allows the operating system to create a sufficient number of kernel threads for a particular application or machine. Although this model is the most flexible, it is also the most difficult to implement.



4.3.4 Two-Level Model

A two-level model is similar to the many-to-many model, but allows a user thread to be bound to a kernel thread.



4.4 Thread Libraries

Thread libraries provide the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread:

- User-level library, where support is limited to the user space.
- Kernel-level library, which is supported by the operating system.

Three thread libraries are in use today:

- POSIX Pthreads — a threads extension of the POSIX standard which may be provided as either a user-level or kernel-level library.
- Windows threads — a kernel-level library that is part of the Windows API.
- Java threads — a user-level library provided managed directly in Java programs.

There are two strategies for creating multiple threads:

- **Asynchronous threading** — the parent thread continues to execute concurrently and independently with the child thread it created. This is useful for tasks that require little data sharing between threads.
- **Synchronous threading** — the parent thread waits until the child thread completes before continuing. This is useful for tasks that require significant data sharing between threads, where a parent thread may need to combine the results of multiple child threads.

4.4.1 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronisation. This is a specification for thread behaviour, not an implementation. Systems that implement the Pthreads specification include UNIX-type operating systems such as Linux and macOS.

The following program calculates the sum of the first n integers using a multi-threaded program that follows the Pthreads API.

```c
#include <stdio.h>
#include <stdlib.h>

#include <pthread.h>

unsigned int sum; /* data shared by thread(s) */

void *runner(void *param); /* function called by threads */

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <integer value>\n", argv[0]);
        return -1;
    }

    /* thread identifier */
    pthread_t tid;

    /* set of thread attributes */
    pthread_attr_t attr;

    /* set default attributes for thread */
    pthread_attr_init(&attr);

    /* create thread */
    // params:
    // - thread identifier
    // - thread attributes
    // - function to be executed by thread
    // - function arguments
    pthread_create(&tid, &attr, runner, argv[1]);

    /* wait for thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %u\n", sum);
}

/* The thread begins control in this function */
void *runner(void *param)
{
    unsigned int i, upper = atoi((char *)param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

In this program, the `sum` variable is shared between the parent and child threads. Each child thread begins control within the `runner` function.

The parent thread waits for all child threads to complete by calling the `pthread_join()` function.

4.5 Implicit Threading

To address the challenges of creating and managing threads, many programming languages transfer these responsibilities to compilers and run-time libraries. This is known as **implicit threading**. The programmer must then identify tasks, rather than threads, that can be executed in parallel.

These tasks are usually defined as functions that can be mapped to a separate thread, typically using the many-to-many model. Three common approaches to implicit threading are discussed below.

4.5.1 Thread Pools

A **thread pool** is a collection of worker threads that are created at start-up and placed into a pool, where they wait for work. When a task is to be performed, a thread is removed from the pool and assigned to the task. When the task is complete, the thread returns to the pool and waits for more work.

This approach has the following advantages:

- Overhead of thread creation is avoided by servicing requests using an existing thread.
- The number of threads available can be bounded to the size of the pool.
- The separation of task submission from thread management allows for different strategies for scheduling tasks on threads. For example, a task can be scheduled to execute periodically.

Thread pools are supported by the Windows API.

4.5.2 OpenMP

OpenMP is a set of compiler directives and an API for C, C++, and Fortran that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel. These regions are defined using *#pragma* directives as shown below.

```
#include <stdio.h>

#include <omp.h>

int main(int argc, char *argv[])
{
    /* sequential code */
    int N = 10;
    int a[N], b[N], c[N];

/* parallel code */
#pragma omp parallel
    {
        printf("Parallel region\n");
    }

#pragma omp parallel for
    {
        for (int i = 0; i < N; i++)
```

```
            c[i] = a[i] + b[i];
    }

    /* sequential code */
    for (int i = 0; i < N; i++)
        printf("%d\n", c[i]);

    return 0;
}
```

In this code, the *#pragma omp parallel* directive creates as many threads as cores in the system, and all threads execute the code within the parallel region. To parallelise loops, the *#pragma omp parallel for* directive can be used to distribute the iterations of a for loop across threads.

OpenMP also allows developers to specify the number of threads manually, and identify whether data should be shared or private to each thread. OpenMP is available on several open-source and commercial compilers for Linux, Windows, and macOS systems.

4.5.3 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed by Apple for macOS and iOS. It is a combination of a run-time library, API, and language extension that allows developers to identify tasks that can be executed in parallel. GCD manages most of the details of threading.

GCD schedules tasks for run-time execution by placing them in a **dispatch queue**. When a task is ready to execute, it is removed from this queue and assigned to an available thread from a thread pool. There are two types of dispatch queues:

- **Serial queues** — tasks are executed in FIFO order, and a task must be completed before the next task is executed. This queue is called the **main queue** of a process.
- **Concurrent queues** — tasks are also executed in FIFO order, but multiple tasks may be removed at a time, allowing them to be executed in parallel. These tasks are prioritised into three system wide queues.

In C, GCD defines a language extension called a **block** that is a self-contained unit of work specified by a caret (^):

```
^{ /* code */ }
```

4.6 Threading Issues

This section discusses issues that arise when designing multi-threaded programs.

4.6.1 `fork()` and `exec()` System Calls

The semantics of the `fork()` system call in a multi-threaded program may not be well defined. If a thread calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? In some UNIX systems, two versions of the `fork()` system call are defined to address this ambiguity.

In the case of a thread calling `exec()`, the new program specified will usually replace the entire process, including all threads.

4.6.2 Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred. For example, illegal memory access or division by 0. Signals may be received both synchronously and asynchronously, and follow the same pattern:

1. A signal is generated by a particular event.
2. The signal is delivered to a process.
3. The signal is handled by either a default or user-defined signal handler.

Every signal has a default handler that the kernel runs when handling that signal. This may be overriden by a user-defined signal handler. Some signals may be ignored, whereas some signals terminate the program.

For single-threaded applications, signals are delivered to the process. However, it is not clear where a signal should be delivered in a multi-threaded application. One of the following options exist:

- Deliver the signal to the thread to which the signal applies.
- Deliver the signal to every thread in the process.
- Deliver the signal to certain threads in the process.
- Assign a specific thread to receive all signals for the process.

4.6.3 Thread Cancellation

Thread cancellation terminates a thread before it has completed. For example, if multiple threads are concurrently searching a large data structure, and one thread finds the result, the other threads may be cancelled.

The thread to be cancelled is called the **target thread**, and there are two ways to cancel this thread:

- **Asynchronous cancellation** — terminates the target thread immediately.
- **Deferred cancellation** — allows the target thread to periodically check if it should be cancelled.

This poses a difficulty when resources are shared between threads. For example, if a thread is cancelled while it is modifying data shared with other threads, it is not possible to reclaim all resources used by the target thread, as other threads may be using them.

In Pthreads, the `pthread_cancel()` function indicates a request to cancel the target thread, which may be deferred or asynchronous. Program cancellation occurs when the thread reaches its **cancellation point**. A cancellation point can be specified using `pthread_testcancel()`, which checks if a cancellation request has been made.

A **cleanup handler** may be invoked to release any resources before the thread is terminated.

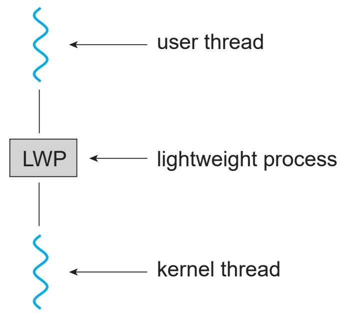
4.6.4 Thread-Local Storage

Thread-local storage allows threads to have a separate copy of shared data in a process. This is useful when the thread the developer has no control over the thread creation process.

This data has a similar visibility to static variables, but it is unique to each thread.

4.6.5 Scheduler Activations

Systems implementing the many-to-many and two-level threading models require communication between the kernel and thread library to allow the number of kernel threads allocated to the program to be adjusted dynamically. This is done using an intermediate data structure between user and kernel threads, known as a **lightweight process** (LWP).



To the user-thread library, the LWP appears as a virtual processor on which the process can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is the kernel thread that is scheduled by the operating system.

One scheme for communication is known as **scheduler activation**. In this scheme, the kernel provids an application with a set of virtual processors (LWPs). The kernel informs an application about events using an **upcall** that are handled by the thread library in an **upcall handler**.

5 Synchronisation

A cooperating process is a process that can affect other processes in a system. These processes allow concurrent access of data which may result in data inconsistency. This section explores the various mechanisms that can be used to ensure consistency.

5.1 Background

Processes executing concurrently or in parallel may be interrupted at any point in their execution, even if that process has partially completed an operation.

Consider an updated producer-consumer shared memory implementation, where the producer and consumer access a shared variable `count` to allow the entire buffer to be utilised. This variable counts the number of items not yet consumed by the consumer.

```c
#define BUFFER_SIZE 4

// Shared memory
int buffer[BUFFER_SIZE];

// Shared variable
int count = 0; // Items to be consumed

int in = 0;  // Index for producer
int out = 0; // Index for consumer

void produce(int *produced)
{
    while (1)
    {
        // Wait until consumer has consumed next index
        while (count == BUFFER_SIZE)
            ; // Buffer is full, do nothing

        // Buffer has space, produce item
        buffer[in] = *produced;

        // Update index
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consume(int *consumed)
{
    while (1)
    {
        // Wait until producer has produced next index
        while (count == 0)
            ; // Buffer is empty, do nothing

        // Buffer has item, consume it
        *consumed = buffer[in];

        // Update index
        out = (out + 1) % BUFFER_SIZE;
        count--;
    }
}
```

Assuming `count = 5` initially, and the increment and decrement operations are implemented in Assembly like so:

```
register1 = count;
register1 = register1 + 1;
count = register1;

register2 = count;
register2 = register2 - 1;
count = register2;
```

then if during the concurrent execution of this program, the CPU executes these instructions in the following manner:

```
register1 = count;         // producer, register1 = 5
register1 = register1 + 1; // producer, register1 = 6
// interrupt
register2 = count;         // consumer, register2 = 5
register2 = register2 - 1; // consumer, register2 = 4
// interrupt
count = register1;         // producer, count = 6
// interrupt
count = register2;         // consumer, count = 4
```

then, the resulting value of `count` will incorrectly be 4, rather than 5.

The situation where multiple processes manipulate the value of the same variable concurrently, where the order in which access to the variable occurs is important, is known as a **race condition**.

To guard against the race condition encountered above, processes need to be synchronised.

5.2 The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_{n-1}\}$, each of which has a segment of code, called a **critical section**, in which the process may access and update data shared with at least one other process. In this system, no two processes may be simultaneously inside their critical sections.

The solution to this problem splits the program into the following sections:

- entry section
- critical section
- exit section
- remainder section

Each process requests permission to enter its critical section in the entry section. Once it has finished executing its critical section, the process may be followed by an exit section, to indicate that it has finished. The remainder section consists of any code that is not critical.

This solution must also satisfy the following three requirements:

- **Mutual exclusion** — if process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress** — if no process is currently executing in its critical section, then only those processes not in their remainder section may participate in the decision of which process will enter its critical section next. This decision must be made in a finite amount of time.
- **Bounded waiting** — there exists a bound on the number of times that a process requests to enter its critical section before that request is granted. This prevents a process from waiting forever to enter its critical section.

Here we assume that each process executes at a nonzero speed but there is no assumption concerning the relative speed of the n processes.

There are two approaches to handle critical sections depending on the kernel:

- **Preemptive kernel** — the kernel may pre-empt (interrupt) a process while it is executing in kernel mode.
- **Non-preemptive kernel** — the kernel does not pre-empt (interrupt) a process while it is executing in kernel mode.

While the non-preemptive kernel is free of race conditions, it may be more repsonsive as there is no risk that a process will run forever.

5.3 Peterson's Solution

Peterson's solution is a two process solution to the critical section problem which uses two variables to control access to the critical section. An implementation of Peterson's solution is shown below.

```c
#include <pthread.h>

volatile int turn = 0;
volatile int flag[2] = {0, 0};
```

```c

void *P(void *arg)
{
    int i = (int *)arg; // this process
    int j = 1 - i;      // the other process

    /* entry section */
    flag[i] = 1; // i wants to enter critical section
    turn = j;    // but first let j enter critical section
    while (flag[j] && turn == j)
        ;        // wait for j to leave critical section

    /* critical section */

    /* exit section */
    flag[i] = 0; // i finished critical section

    /* remainder section */
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1, thread2;

    // create two threads
    pthread_create(&thread1, NULL, P, (void *)0);
    pthread_create(&thread2, NULL, P, (void *)1);

    // wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

A problem with this solution is that the operating system is free to reorder instructions to optimise runtime, resulting in both threads entering their critical sections at the same time.

5.4 Hardware Support for Synchronisation

The above solution is a *software-based* solution as it does not rely on any special support from the operating system or hardware, to ensure mutual exclusion.

As will be discussed in the next section, locks are a common mechanism for ensuring mutual exclusion. Locks can be used to prevent a process from entering its critical section, if another process is already in its critical section.

5.4.1 Atomic

An **atomic operation** is an indivisible operation that cannot be interrupted by the operating system.

Atomic variables provide atomic operations on basic data types such as integers, and can be modified by multiple threads without race conditions.

5.5 Mutex Locks

A **mutex lock** (mutual exclusion lock) is a lock that can be used to ensure mutual exclusion. A mutex lock protects critical sections and thus prevents race conditions. A mutex lock has two operations:

- `acquire()` — if the lock has been released, acquire it, otherwise wait until it is.
- `release()` — release the lock.

A thread cannot enter its critical section until it has acquired the mutex lock and must release the mutex lock when it exits its critical section. These functions can be implemented as follows:

```
acquire()
{
    while (!available)
        ; // busy wait

    available = false;
}

release()
{
    available = true;
}

do
{
    acquire();
    // critical section
    release();
    // remainder section
} while (true);
```

The calls to `acquire()` and `release()` must be **atomic**.

A problem with this implementation is that it requires **busy waiting** where a thread waits for a condition to met without yielding control to the operating system. This is wasteful as it consumes

CPU cycles that could be used by other threads.

This particular scenario is known as a **spinlock**, as lock contention is low, and the thread is likely to acquire the lock after a short duration. Here waiting is more efficient than performing a context switch to another thread.

5.6 Semaphores

Semaphores are similar to mutex locks, but allow multiple threads to access a shared resource simultaneously. Semaphores can be accessed by two atomic operations:

- `wait()` — if the value of the semaphore is nonzero, decrement it and continue, otherwise, wait until it is.
- `signal()` — increment the value of the semaphore.

The number a semaphore is initialised to is the number of threads that are allowed to access the shared resource simultaneously. If this number is equal to 1, the semaphore is called a **binary semaphore**, and is equivalent to a mutex lock. When a semaphore is initialised to a value greater than 1, it is called a **counting semaphore**. Semaphores can be implemented as follows:

```
wait(S)
{
    while (S <= 0)
        ; // busy wait

    S--;
}

signal(S)
{
    S++;
}

do
{
    wait(S);
    // critical section
    signal(S);
    // remainder section
} while (true);
```

where both `wait()` and `signal()` must be atomic.

5.6.1 Precedence

If a process must be executed before another process, a semaphore can be used to enforce this precedence. Given processes P_1 and P_2 with statements S_1 and S_2 respectively, if P_1 must be executed before P_2, then the following code can be used:

```
// P1
S1;
signal(S);

// P2
wait(S);
S2;
```

where S is a binary semaphore initialised to 0. This code requires P_1 to signal S before P_2 can execute.

5.6.2 Non-Busy Waiting

Both mutex locks and the above implementation of the semaphore require threads to busy wait. This can be avoided by **suspending** threads that need to wait for a semaphore. This is done by maintaining a queue of suspended threads, where each suspended thread transfers control to the CPU scheduler. When a thread has released a semaphore, it must wake up a thread from the queue. An implementation of this is shown below:

```
typedef struct
{
    int value;
    struct process *list;
} semaphore;

void wait(semaphore *S)
{
    S->value--;
    if (S->value < 0)
    {
        // add this process to S->list;
        sleep();
    }
}

void signal(semaphore *S)
{
    S->value++;
    if (S->value <= 0)
    {
        // remove a process from S->list;
        wakeup();
    }
}

do
{
    wait(S);
    // critical section
    signal(S);
    // remainder section
} while (true);
```

In this implementation, semaphore values may be negative to allow the number of waiting threads to be tracked as the absolute value of the semaphore value.

5.6.3 Deadlock and Starvation

A **deadlock** occurs when two or more threads are waiting for an event that can only be caused by one of the waiting threads. In the following example, both thread 0 and 1 are waiting for the other thread to release the semaphore, resulting in a deadlock.

```
// thread 0
wait(S0);
wait(S1);
signal(S0);
signal(S1);

// thread 1
wait(S1);
wait(S0);
signal(S1);
signal(S0);
```

A **starvation** occurs when a thread is perpetually denied access to a resource. This can occur when a thread has a lower priority than other threads, and thus is never scheduled.

5.6.4 Semaphore Problems

Semaphores may be misused in the following ways:

- If a `signal()` operation is executed before a `wait()` operation, the mutual-exclusion requirement will be violated.
- If a `wait()` operation is executed twice without an intervening `signal()` operation, the thread will permanently block on the second call to `wait()` as the semaphore value will be unavailable.

- If either `wait()` or `signal()` is omitted, mutual exclusion may be violated, or the thread will block indefinitely.

5.7 Monitors

Monitors are a high-level abstraction that provide a convenient and effective mechanism for process synchronisation. This reduces the risk of programmer error when using semaphores.

A monitor is an **abstract data type** (ADT) that encapsulates data with a set of methods that operate on that data. A monitor is defined as follows:

```
monitor monitor-name
{
    // shared variable declarations
    // procedures that operate on the shared variables
    procedure P1(...) { ... }
    ...
    procedure Pn(...) { ... }

    // initialisation code
    initialisation(...) { ... }
}
```

where each operation within this monitor is mutually exclusive. All internal variables must only be accessed by the procedures within the monitor, and local variables must be private to each procedure. The monitor maintains a queue of threads waiting to enter the monitor, and only one thread may be executing within the monitor at any time.

This is not sufficient to model certain synchronisation problems, and therefore, monitors also provide the **condition construct**.

```
condition x, y;
```

A condition variable has two operations:

- `wait()` — a thread that invokes an operation is suspended until another thread invokes `signal()`.
- `signal()` — resumes a process (if any) which called `wait()`. This function may be called multiple times, and has no effect if no threads are suspended.

5.8 Classical Problems of Synchronisation

This section explores three classical problems of synchronisation, and provides solutions using the concepts discussed above.

5.8.1 The Bounded-Buffer Problem

In this problem, processes produce and consume data from a shared buffer of size n. The solution to this problem requires the following shared data:

```
#define BUFFER_SIZE n
semaphore mutex = 1; // controls access to critical section
semaphore empty = N; // counts empty buffer slots
semaphore full = 0;  // counts full buffer slots
```

The producer and consumer can be implemented as follows:

```
// producer
while (true)
{
    /* produce an item */
    wait(empty);
    wait(mutex);
    /* add item to buffer */
    signal(mutex);
    signal(full);
}

// consumer
while (true)
{
    wait(full);
    wait(mutex);
    /* remove item from buffer */
    signal(mutex);
    signal(empty);
    /* consume the item */
}
```

5.8.2 The Readers-Writers Problem

In this problem, a data set is shared among a number of concurrent processes, where some processes only read the data, and others can both read and write the data. In the following solution, the data set is protected by two semaphores:

```
semaphore mutex = 1;    // control access to read_count variable
semaphore rw_mutex = 1; // controls access to data
int read_count = 0;     // number of readers accessing data
```

The reader and writer processes can be implemented as follows:

```
// reader
while (true)
{
    wait(mutex); // ensure exclusive access to read_count
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);  /* first reader locks data */
    signal(mutex);
    /* read data */
    wait(mutex); // ensure exclusive access to read_count
    read_count--;
    if (read_count == 0)
        signal(rw_mutex); /* last reader unlocks data */
    signal(mutex);
}

// writer
while (true)
{
    wait(rw_mutex); // ensure exclusive access to data
    /* write data */
    signal(rw_mutex);
}
```

5.8.3 The Dining-Philosophers Problem

In this problem, n philosophers sit around a table with a bowl of rice in the centre. These philosophers spend their lives thinking and eating. Each philosopher has a plate of food in front of them, with chopsticks between each plate.

This problem has the following constraints:

- When a philosopher is hungry, they will try to pick up the two chopsticks adjacent to their plate, one at a time. They cannot take a chopstick that is held by a neighbouring philosopher.
- When a philosopher has two chopsticks, they may eat for a finite amount of time.
- When a philosopher is finished eating, they must put down both chopsticks and think.
- When a philosopher is thinking, they do not need any chopsticks.

The first solution to this problem uses semaphores. Consider the shared data:

```
semaphore chopstick[n];
```

where all elements of this array are initialised to 1. The philosopher processes can be implemented as follows:

```
// philosopher i
while (true)
{
    wait(chopstick[i]);
    wait(chopstick[(i + 1) % n]);
    /* eat */
    signal(chopstick[i]);
    signal(chopstick[(i + 1) % n]);
    /* think */
}
```

This solution is correct, but may result in **starvation** as a philosopher may never be able to acquire both chopsticks if other philosophers are constantly eating. Additionally, if all philosophers are hungry at the same time, they may all acquire one chopstick and wait forever for the other, resulting in a **deadlock**.

To solve the problem of deadlock, a monitor can be used.

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING } state[n];
    condition self[n];

    void pickup(int i)
    {
        state[i] = HUNGRY;      // ith philosopher is hungry
        test(i);                // try to eat
        if (state[i] != EATING) // if unable to eat
            self[i].wait();     // wait to be signalled
    }

    void putdown(int i)
    {
        state[i] = THINKING;   // ith philosopher is thinking
        test((i + n - 1) % n); // allow left neighbour to eat if hungry
        test((i + 1) % n);     // allow right neighbour to eat if hungry
    }

    void test(int i)
    {
        // if neither neighbour is eating
        // and ith philosopher is hungry
        if ((state[(i + n - 1) % n] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % n] != EATING))
        {
            state[i] = EATING;
            self[i].signal();  // signal ith philosopher to eat
        }
    }

    initialisation()
    {
        for (int i = 0; i < n; i++)
            state[i] = THINKING;
    }
}
```

This solution ensures that a philosopher only picks up a chopstick if both of their neighbours are not eating. This prevents deadlock as there will always be at least one philosopher that can eat.

6 Safety Critical Systems

Safety critical systems are systems whose failure may result in one of the following outcomes:

- illness, serious injury, or death
- damage to or loss of equipment or property
- damage to the environment

The increase of software in safety critical systems necessitates the proper design and implementation of these systems.

6.1 Safety Critical Software

Software is not inherently safe or unsafe. However the use of software in a safety critical system may contribute to an unsafe situation. Such software is considered safety critical.

Low risk systems are safety related, whereas high risk systems are safety critical.

6.2 Dependability

Dependability is the most important property of a system critical system. It reflects the user's confidence in a system. The costs of system failure are often very high, and so dependability covers a range of attributes such as:

- **Reliability** — The ability to deliver services as specified.
- **Availability** — The ability to deliver services when required.

- **Safety** — The ability to operate without catastrophic failure.
- **Security** — The ability to protect itself against accidental or deliberate intrusion.

Both reliability and availability are measured as a probability of failure over a given time period.

6.3 Achieving Safety

- **Hazard avoidance** — The system should be designed to avoid hazards.
- **Hazard detection and removal** — The system should be designed to detect hazards and remove them before they cause accidents.
- **Damage limitation** — The system should be designed to limit the damage caused by hazards.

In complex systems, accidents typically occur as a result of multiple failures. As such, designing a system to be completely safe is difficult. **Dependability costs** increase exponentially as increasing levels of dependability are required, and therefore a comprimise must be made between design cost and dependability.

6.4 Risk Analysis

Failures in safety critical systems can be caused by:

- **Hardware failure** — Manufacturing, end-of-life, or design error.
- **Software failure** — Specification, design, or implementation errors.
- **Operational failure** — Human or socio-technical errors.

6.4.1 Hazard and Risk

- A **hazard** is a situation that has the potential to cause harm.
- **Risk** enacapsulates the probability that a hazard will lead to an accident, and the severity of the consequences of that accident. Risk is a product of the probability of accident and the severity of the consequences.

Risk analysis involves identifying hazards and limiting the risks associated with them.

6.4.2 Faults and Failures

A **fault** is an event that may lead to a failure, whereas a **failure** is the inability of a system or component to perform its required function within specified limits.

Failures may be the result of one or more faults, and there are several ways to trace a failure back to its cause:

- **Failure mode and effects analysis** (FMEA) — A systematic way of identifying the possible faults that may lead to a failure, and the consequences of that failure.

- **Failure tree analysis** (FTA) — A graphical technique that can be used to analyse the combination of faults that may lead to a failure.

6.5 Safety Standards

Many standards have been developed to ensure the safety of software in particular domains. One standard is the **International Electrotechnical Commission** (IEC) 61508 standard, which is a generic standard for functional safety. This standard has the following views on risk:

- Zero risk can never be reached, but the risk can be reduced to an acceptable level.
- Non-tolerable risks must be reduced as lot as reasonably possible.
- Optimal cost effective safety is achieved when addressed in the entire safety lifecycle.

IEC 61508 defines three successive tiers of safety assessments:

- Safety Instrumented Systems (SIS) — The entire system that is composed of Safety Instrumented Functions (SIF).
- Safety Instrumented Functions (SIF) — A function implemented by the SIS that is intended to achieve or maintain a safe state for the process.
- Safety Integrity Level (SIL) — A measure of the safety integrity requirements of a particular SIF. SIL 1 is low risk and therefore allows for a higher probability of failure, and SIL 4 is high risk and therefore requires a Lower probability of failure.

6.6 Real-Time Operating Systems

Real-time operating systems (RTOS) are operating systems that are designed to meet the requirements of real-time applications. The following considerations must be made when designing an RTOS:

- **Tasking**
  - Tasks terminating or deleting
  - Overflows within a kernel's storage area for task control blocks
  - Tasks exceeding stack size limits
- **Scheduling**
  - Deadlocks
  - Resource starvation
  - Unbounded execution times
- **Memory and I/O Access**
  - Inappropriate pointer usage

  - Data erasure
  - Unauthorised access
- **Queueing**
  - Overflows
- **Interrupt and Error Handling**
  - Lack of error handling
  - Lack of interrupt handling
  - Improper protection of superisor tasks

6.7 DO-178B Standard

The DO-178B standard is a software standard for airborne systems. It consists of five main processes:

- Software planning
- Software development
- Software verification
- Software configuration management
- Software quality assurance

6.7.1 Software Planning Process

The software planning process determins what needs to be done to produce safe and requirements-based software. It's expected outputs are:

- A plan for software aspects of certification
- A plan for software development
- A plan for software verification
- A plan for software configuration management
- A plan for software quality assurance

6.7.2 Software Development Process

The software development process is broken into four sub-processes:

- **Software requirements** — High-level requirements in relation to functionality, performance, and safety.

- **Software design** — Low-level requirements used to implement source code.
- **Software coding** — Production of source code from the design process.
- **Software integration** — Integration of software into a real-time environment.

This process is iterative, and the expected outputs are:

- Software requirements data
- Software design description
- Source code
- Executable object code

6.7.3 Software Verification Process

Software verification consists of:

- verification of software requirements
- validation of software design

these can be achieved through reviews, analysis, and testing. This process outputs:

- Software verification cases and procedures
- Software verification results

6.7.4 Software Configuration Management Process

The software configuration management process is used to establish secure and effective configuration control for all artifacts. The following activities may be conducted:

- Identification of configuration items
- Change control
- Baseline establishment
- Software archival

This process outputs:

- Software configuration index
- Software lifecycle environment configuration index

6.7.5 Software Quality Assurance Process

The software quality assurance process provides assurance that the software life cycle process will yield quality software. Each process is analysed to demonstrate that processes produce expected outputs. Changes to originally proposed plans are reported, evaluated, and resolved to ensure process integrity.

6.8 Programming Standards

While many languages are used in safety critical systems, certain languages may be preferred for various purposes. Many C standards also subset the language to remove features that are considered unsafe.

6.8.1 MISRA C

MISRA C is a set of guidelines on a subset of the C language in critical systems. MISRA C has three levels of compliance:

- **Mandatory** — These rules must be followed without exception.
- **Required** — These rules must be followed but may be violated in certain situations. These violations must also be documented.
- **Advisory** — These rules should be followed but are not mandatory.

MISRA C is used for error prevention, rather than error detection, and violation of a guideline does not necessarily mean that a program is incorrect.