Operating systems (OSes)

Table of contents

Computer system architecture

▼ Introduction

There are a few types of computer system architecture:

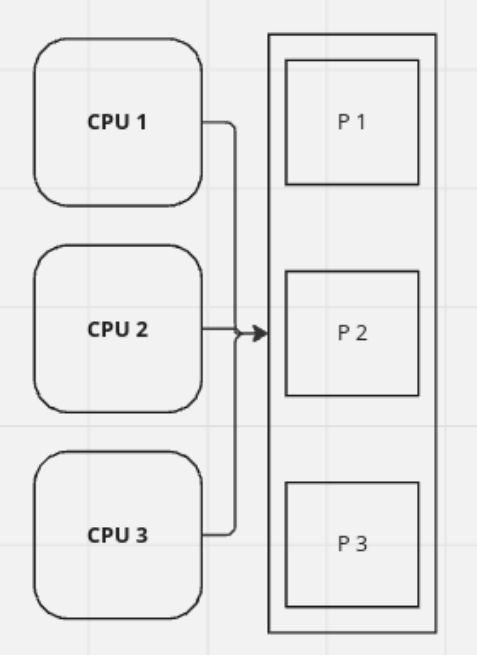
1. **Single processor system:** One main CPU capable of executing a general purpose instructions set, including interaction from a user.
2. **Multiprocessor systems:** also known as parallel or tightly coupled systems. Have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory and peripheral devices.

   None: General purpose CPU with multiple cores (e.g., one in a PC) can be considered as multiprocessor systems. it is typically classified as a Symmetric Multiprocessor System.

▼ Types of multiprocessor systems

▼ 1. Symmetric

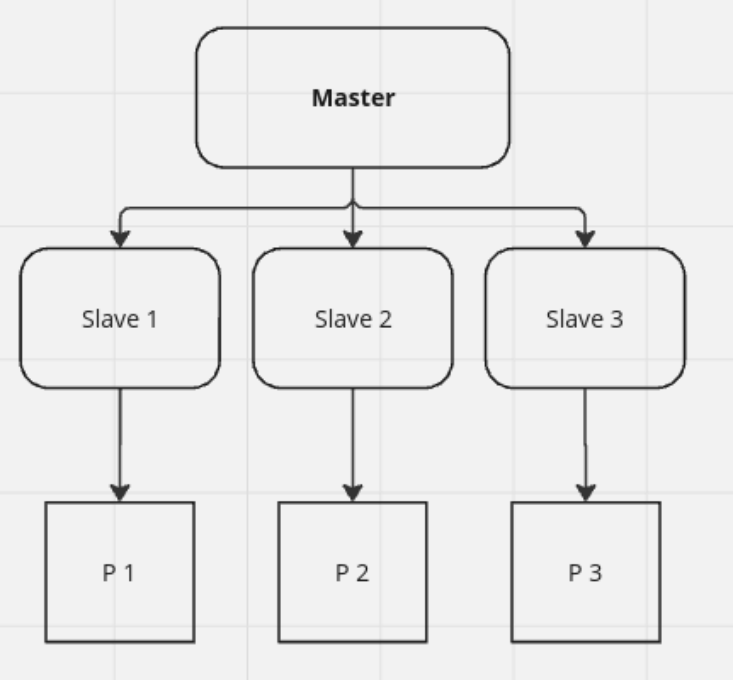
In symmetric systems all cores have equal capabilities (sometimes they are identical) and access to the same memory and system resources, They can execute any task independently or cooperatively.



Symmetric multiprocessor system scheme.

▼ 2. Asymmetric

In asymmetric systems cores have different capabilities or responsibilities. E.g., one core may handle I/O operations while the other computing.



Asymmetric multiprocessor system scheme.

▼ 3. Cluster

Like multiprocessor systems clusters are individual systems (not only CPUs) gathered together. Can be structured symmetrically or asymmetrically.

OS design and architecture

▼ Operating system services

An Operating System provides an environment for programs execution. It also provides several services to programs and users of these programs.

Services:

1. Command Line Interface (CLI) and Graphical User Interface (GIU).
2. Program execution: the OS must be able to load a program (executable code and data) into main memory, and run this program.
3. I/O operations: a user cannot control the O/I devices directly. Support drivers.
4. File System Manipulation: the file system is responsible for organizing the disk. Store file restrictions, files structure, write, delete, update files).
5. Cross process communication: a program that is in execution is a process. Processes may share data between each other.
6. Error detection, error stdout, error journaling.

7. Resource allocation: the OS must be able to allocate resources such as CPU time and memory among all active processes.
8. Protection and Security:
    a. Each process must run in isolated environment.
    b. One process must not have an ability to brake or access data of another processes.
    c. Prevent unauthorized access and record all attempts.

▼ Structure of OSes

▼ Simple (MS DOS)

About: Old MS DOS. It is not a truly layered structure as it exploits access to the hardware from all layers of programs. Old processors did not such thing as dual-mode, hence engineers could not implement safe design systems.

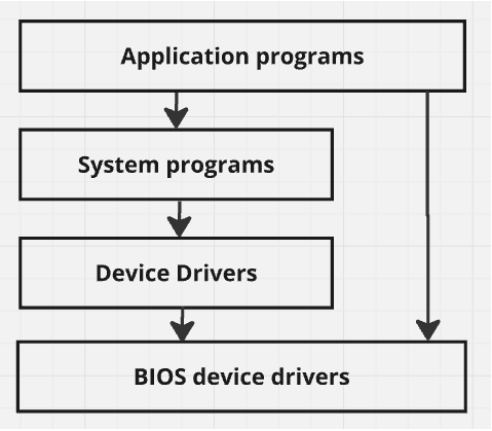
Pros:

- Easy development.
- Better performance.

Cons:

- Frequent system failures: If one program fails, entire OS crashes.
- Poor maintainability: as all layers of OS are tightly coupled, change in one layer can impact other layers heavily and making core unmaintainable over a period of time.

Scheme:



Simple OS structure scheme.

▼ Monolithic (early UNIX systems)

About: a central piece of code called kernel is responsible for all major operations of an OS. Such operations include file management, device management, etc. The kernel is the main components of an operating system and it provides all services of an OS to the

system and application programs. The kernel has access to all the resources and acts as an interface with application programs and the underlying hardware.

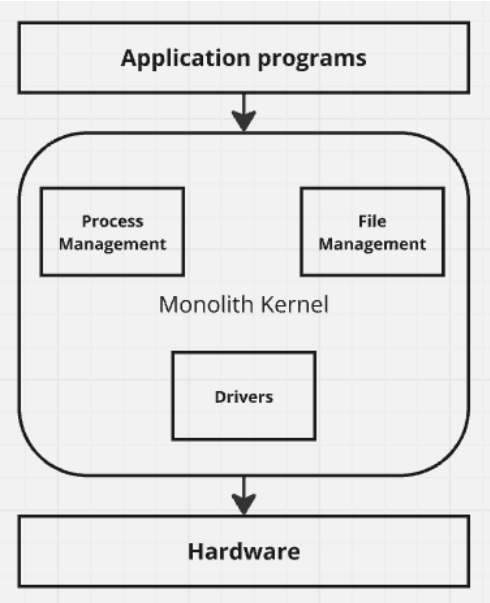
Pros:

- Easy to implement.
- Performance.

Cons:

- Crash prone: if one system program or user applications crashes, the kernel remains to operate.
- Difficult to enhance: It is difficult

Scheme:



Monolith OS structure scheme

▼ Layers structure

About: One way to achieve modularity is OSes. The OS is divided into layers each layer is isolated and responsible for layer-specific tasks only. In this, the bottom layer is the hardware and the topmost is the user interface.

Pros:

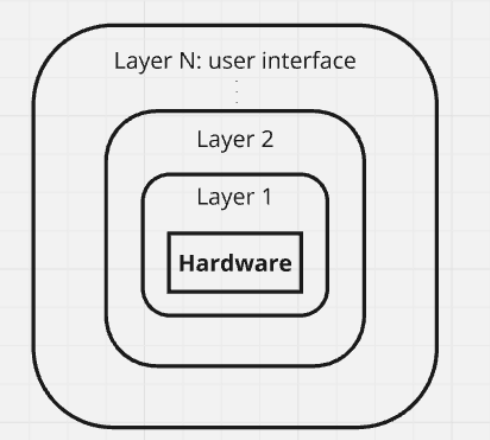
- **High customizable:** a new functionality is added to a certain layer without the need to change other layers.
- **Verifiable:** each layer can be debugged separately.

Cons:

- Less performance.

- Complex designing.

Scheme:



Layered OS structure scheme.

▼ Micro-Kernel

About: Instead of having a large kernel with many functionalities the kernel is kept small, and all the functionalities are migrated to the user-mode and run as system services. Hence, the system runs in two modes:

- **Kernel-mode:** responsible for messaging between user applications and hardware.
- **User-mode:** responsible for system services and user applications.

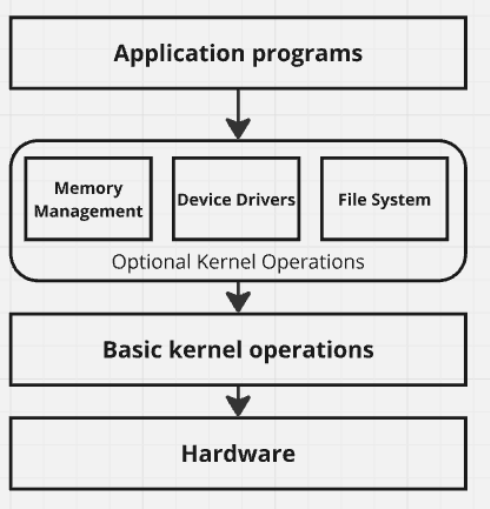
Pros:

- Reliable and stable.
- Maintainability.

Cons:

- Complex to design.
- Performance complications.

Scheme:



Micro-kernel OS structure scheme.

▼ Modules

About: A central kernel is responsible for major OS operations. Other functionality is present in term of modules which are loaded dynamically either at boot time or run time.

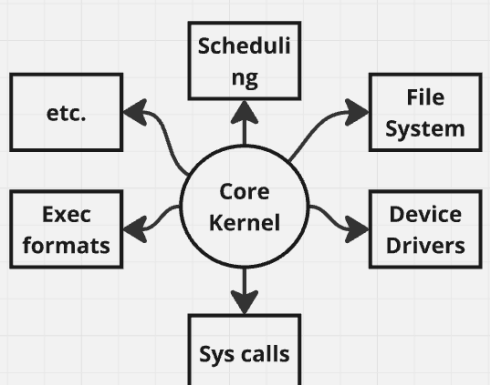
Pros:

- Customization.

Cons:

- Less performance.
- Complex design.

Scheme:



Modular OS structure scheme.

▼ Kernel

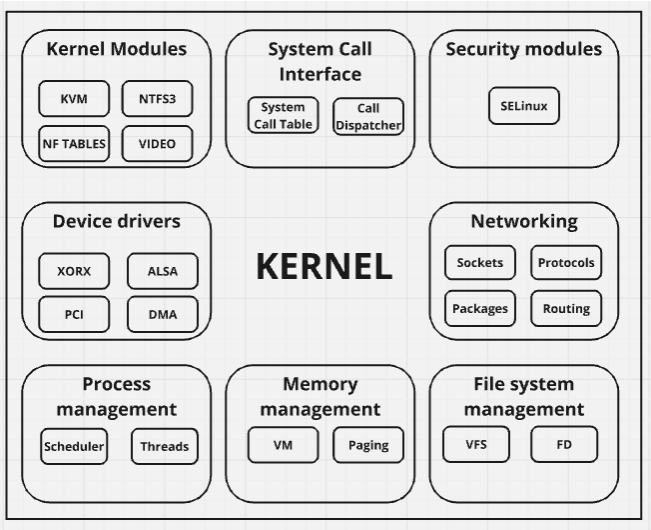
The OS Kernel is the core component of an operating system that acts as a bridge between applications and the actual data processing done at the hardware level. It is responsible for managing the system's resources, providing essential services to other parts of the operating

system and applications, and maintaining the separation between user space and kernel space. Key functions of the kernel include:

- **Process management:** Scheduling and coordinating processes
- **Memory management:** Allocating and deallocating memory for processes
- **Device management:** Controlling and interfacing with hardware devices
- **File system management:** Organizing and maintaining file structures
- **I/O management:** Handling input/output operations
- **Security and protection:** Enforcing access controls and system integrity

The kernel operates in a privileged mode with full access to hardware resources, ensuring efficient and secure system operation.

▼ Linux Kernel Structure



Linux Kernel and its components scheme

I / O (input / output)

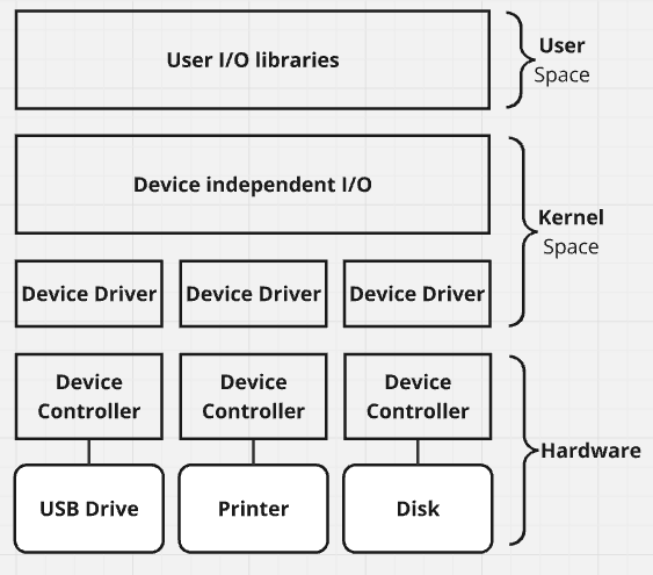
▼ Introduction

▼ I/O structure

I/O system is organized into a layered structure:

1. **User Level Libraries:** Provide simple interface to user program to perform input and output. Example: the `stdio` library for C and C++.
2. **Kernel Level Modules:** Provide a device driver to interact with the device controller and device independent I/O modules used by the device driver.

3. **Hardware:** Includes the actual hardware and hardware controller with interacts with the device driver and makes hardware alive.



I/O structure scheme

▼ Key concept

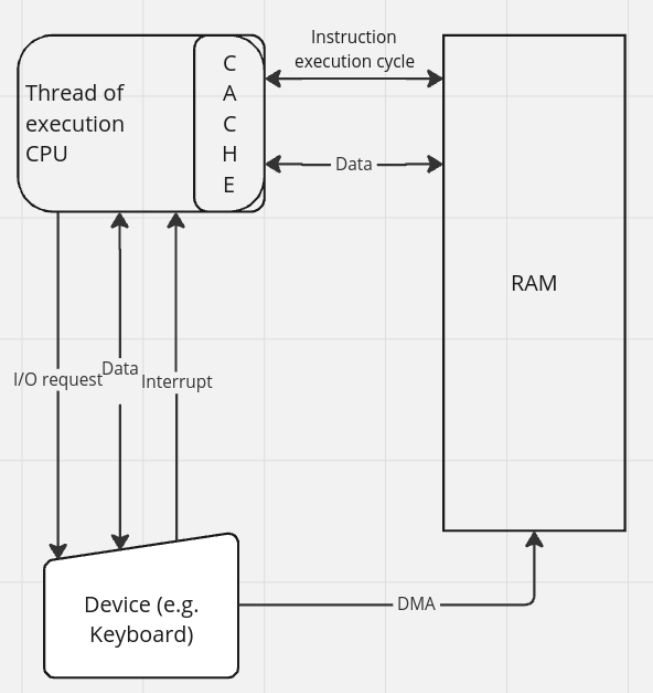
A key concept in the designing of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance.

Example: a program that reads input file as input should be able to read file on any floppy disk, on hard disk, or on CD-ROOM, without having to modify the program for each different device.

▼ I/O operations workflow

1. To start an I/O operation, device driver loads the appropriate registers within the device controller.
2. The device controller, in tern, examines the contents of these registers to determine what action to take.
3. The controller starts the data transfer from the device to it's local buffer.
4. Once the data transfer complete, the device controller informs the device driver via an interrupt that it has finished the operation.
5. The device driver then returns the control to the OS.

Communication with I/O device scheme:



Workflow of I/O operations.

I/O Hardware

Block and Char devices

- **Block device:** a device with which the driver communicates by sending entire blocks of data. (e.g., Hard drives, USB cameras, etc.).
- **Character devices:** a device with which the driver communicates by sending and receiving single character (bytes, octets). (e.g., serial ports, sound carts, etc.).

Device controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device.

The device controllers works like an interface between a device and a device driver. I/O units typically consist of a mechanical component and electronic component where electronic component is called a device controller.

There is always a device controller and a device driver for each device to communicate with the OS. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error handling if necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller.

Synchronous & Asynchronous I/O

- **Synchronous I/O:** in this scheme CPU execution waits while I/O processed.

- **Asynchronous I/O:** I/O processes concurrently with CPU execution.

▼ Communication to I/O devices

The CPU must have a way to pass information to and from I/O device. There is three approaches available to communicate with the CPU and device.

1. **Special Instruction I/O:**
    - Uses CPU instructions are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device and read form.
2. **Memory-mapped I/O:**
    - The same address space is shared by memory and I/O devices. The device is connected directly to certain memory locations so that I/O device can transfer block of data to/from memory without going through CPU.
    - The OS allocates memory buffer and inform the I/O device to use the buffer so send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.
    - Pros:
        - High speed as every instruction that can access memory can be used to manipulate the I/O device.
3. **Direct memory access (DMA):**
    - Slow I/O devices (e.g., keyboard) generate an interrupt to the CPU after each byte is transferred. If a fast device would generate an interrupt to the CPU after each byte is transferred it would cause a CPU overhead, as CPU would need to manage each interrupt.
    - Direct memory access allows to read and write to the memory without CPU involvement. DMA module itself controls exchange of data between main memory and I/O device. CPU is only involved at the beginning and end of transfer and interrupted after entire block has been transferred.
    - Requires a special hardware component called DMA controller (DMAC) that manages the data transfer and arbitrates across the system bus.

▼ Pooling & interrupting

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

Pooling: The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Interrupts: An interrupt is a signal to the microprocessor from a device that requires attention. A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

I/O Software

Device drivers

Device drivers are software modules that can be plugged into an OS to handle a particular device. OS takes help from device drivers to handle I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such way the code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device.

Device driver performs following tasks:

1. Accept request fro the device independent software above it.
2. Interact with the device controller to take and give I/O and perform required error handling.
3. Making sure that requests are executed successfully.

Interrupt handlers

An interrupt handler or interrupt service routine (ISR), it is a piece of software or more specifically a callback function in an OS or more specifically in a device driver, whose execution is triggered by the reception of an interrupt.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up processes that was waiting for an interrupt to happen.

The interrupt mechanism accepts an address — a number that select a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

Device independent I/O software

The main function of I/O independent software is to perform I/O functions that are common to all devices and provide a uniform interface to the user-level software. Though it is hard to write completely device independent software but it is possible to write some modules which are common among all the devices.

List of functions for device independent software:

1. Uniform interfacing for device drivers.

2. Device naming.
3. Device protection.
4. Providing a device-independent block size.
5. Buffering: data coming off the device cannot be stored in its final destination.
6. Storage allocation on block devices.
7. Allocation an releasing dedicated devices
8. Error reporting.

▼ User-Space I/O software

User-Space I/O software is represented by libraries which provide richer and simplifies interface to access the functionality of the kernel or ultimately interactive with the device drivers.

Most of user-space I/O software consist of library procedures with some exceptions like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system.

I/O libraries (e.g., `stdio`) are in user-space to provide an interface to the OS resident device-independent I/O SW. Example: `putchar()`, `getchar()`, `printf()`, and `scanf()` are example of user-space I/O library in C.

▼ Kernel I/O subsystem

Kernel I/O subsystem is responsible to provide many services related to I/O.

List of services:

1. **Scheduling:** Schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The kernel I/O scheduler rearranges the order of the queue to improve overall system efficiency and the average response time experienced by applications.
2. **Buffering:** Kernel I/O subsystem maintains a buffer that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of data stream or to adapt between devices that have different data transfer size.
3. **Caching:** Kernel maintains cache memory which is the region of fast memory that hods up copies of data.
4. **Spooling and device reservation:** A spool is a buffer that holds output for a device, such as printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some OSes, spooling is manages by a system daemon process. In other OSes, it is handles by an in kernel thread.

5. **Error handling:** OS that uses protected memory can guard against many kinds of hardware and application errors.

▼ I/O Buffering

▼ Introduction

I/O buffering is a process of using temporary memory storage, knows as buffer, to hold data during I/O operations.

Buffering addresses several issues in computer systems:

1. **Speed mismatch:**
   - **Problem:** I/O devices (e.g., disks, printers) operate significantly slower than the CPU.
   - **Solution:** Buffers decouple the CPU from I/O device speeds by providing temporary storage for data.
2. **Data transfer granularity:**
   - **Problem:** I/O devices may transfer data in small chunks, while the CPU works better with larger, aligned blocks.
   - **Solution:** Buffers accumulate data to align transfer sizes.
3. **Overlapping computations and I/O:**
   - **Problem:** Without buffering, the CPU would remain idle while waiting for I/O operations to complete.
   - **Solution:** Buffers enable asynchronous operations, allowing computation and I/O to overlap.
4. **Smooth data flow:**
   - **Problem:** Interruptions in data streams (e.g., network packets) can cause inconsistencies.
   - **Solution:** Buffers smooth out data flow, absorbing bursts or delays.

▼ Workflow

- **Input buffering:** Temporary stores incoming data before passing it to the CPU or application.

  Example: Keyboard input is stored in a buffer before being processed. Network packets are buffered before being processed by the protocols stack.
- **Output buffering:** Temporary holds data generated by the CPU or applications before sending it to an output device.

  Example: Spooling print jobs into a buffer before sending them to the printer. Writing disk data in blocks rather than byte-by-byte.

- **Double buffering:** Uses two buffers. While one is being filled, the other is being processed or transmitted.

▼ Types of buffer

1. **Single buffer:** user to hold data temporary during I/O operations.

   Pros:

   - Simple to implement.
   - Reduces the speed mismatch between devices and the CPU.

   Cons:

   - Limits performance as only one operation can happen at a time (e.g., filling or processing

2. **Double buffer:** two buffers are used.

   - **Input buffer:** holds incoming data while another buffer is processed.
   - **Output buffer:** Prepares outgoing data while another buffer is send to the device.

   Pros:

   - Allows overlapping I/O and computation.
   - Increases throughput by reducing idle time.

   Cons:

   - Slightly higher memory usage.
   - Increased complexity.

3. **Circular buffer:** a fixed-size buffer where data is added to the end and removed from the front in a circular fashion.

   Pros:

   - Efficient use of buffer space.
   - Suitable for continuous data streams like audio or video.

   Cons:

   - Requires careful management to avoid overwriting unread data.

4. **Spooling:** a buffer-like mechanism where data is written to a **temporary storage area** (e.g., a disk file) before being sent to the I/O device.

   Pros:

   - Handles large data volumes effectively.
   - Decouples applications from slow output devices.

▼ Buffering strategies

1. **Unbuffered I/O:** Directly transfers data between application and the device.
2. **Fully buffered I/O:** Accumulates data in the buffer and processes it only when the buffer is full or explicitly flushed. (e.g., using a high-level APIs like `fprintf()`.
3. **Line buffered I/O:** Buffers data until a new line character is encountered. Common in interactive environments (e.g., terminal input).
4. **Block buffered I/O:** Buffers data in large blocks. Disk file systems like `ext4` use block buffering.

▼ Challenges with buffering

1. **Memory overhead:** Buffers consume RAM, which could be used for other processes.
2. **Buffers overflow:** Excessive data can overwrite buffers, leading to errors or vulnerabilities.
3. **Data loss:** Unflushed buffers can lead to data loss during unexpected shutdowns.

Virtual machines

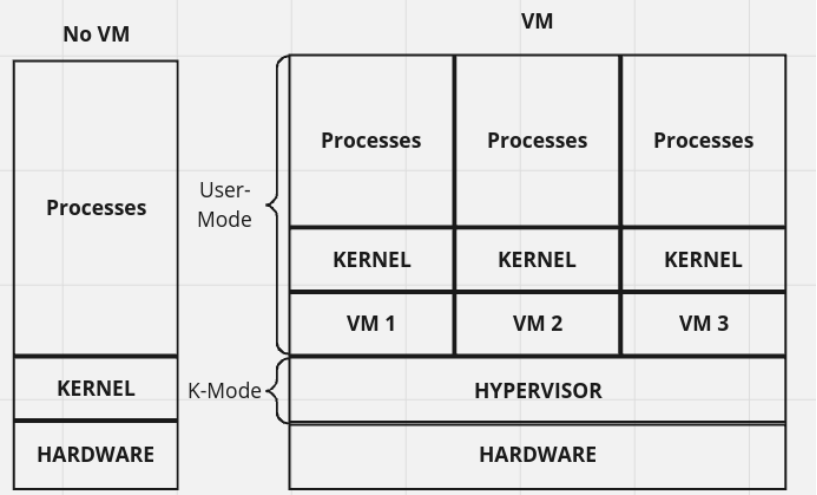
▼ Fundamental Idea

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (CPU, disk, memory, network interface, etc.) into several different execution environments, thereby creating the illusion that each separate execution environment is running on its own private computer.

A virtual machine is a software emulation of a physical computer, running its own OS, and using virtualized hardware.

Key principles of virtual machines:

- **Isolation**: Each virtual machine is isolated from each other, so running multiple VMs on a single host allows different environments to coexist without interference.
- **Hardware abstraction**: Virtual machines use virtualized hardware, which is managed by a hypervisor.



Virtualization from the hosts point of view.

▼ Implementation

A concrete implementation depends on the type of a virtual machine. In general, a virtual machine software (i.e. hypervisor) runs in a kernel-mode. Meanwhile, virtual machines themselves run in the user-mode.

Just as a physical machine has two modes, a virtual machine itself has two modes:

- a virtual kernel mode: a kernel-mode inside of a virtual machine.
- a virtual user mode: a user-mode inside of a virtual machine.

Both of these virtual machine modes run in the user-mode of the physical machine they are in.

▼ How do virtual machines work

Hypervisor:

A core of virtualization is the hypervisor (virtual machine monitor). The hypervisor is responsible for creating and managing virtual machines, allocation of system resources (CPU, memory, etc) to virtual machines, isolating virtual machines from each other.

Types of hypervisors:

1. Bare-metal:
    - Runs directly on the host machine (without OS).
    - Provides high performance and resource management.
    - Used in data centers and enterprise environments.

    Examples: VMware ESXi, MS Hyper-V, Xen, etc.
2. Hosted:
    - Runs on top of an existing OS.

- Treated as an application on the host OS, and VMs run withing this environment.
- Easy to setup but have lower performance.

Examples: Oracle VMBox, QEMU+KVM, etc.

▼ Types of virtual machines

There are two major types of virtual machines, each serving a different goal.

▼ System virtual machines

Provides a complete environment to run a full OS. It behaves like an independent physical machine with its own operating system, and is able to run multiple applications.

▼ Process virtual machines

Designed to run a single application in a virtual environment. It abstracts the execution of an individual program from the underlying system, allowing cross-platform compatibility for applications.

Instead of operating systems, process virtual machines use runtime environments which provide essential services that application needs.

Internal memory management: The process VM handles memory allocation and garbage collection within the application runtime environment, making it independent of the OS's memory management system.

Use case is to run programs in a platform-independent way (e.g., Java Virtual Machine (JMV), Python Virtual Machine (PVM).

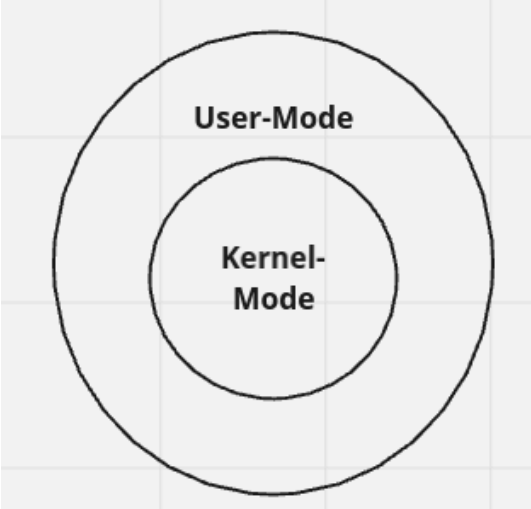
System calls

▼ OS modes (OS spaces) user and kernel

▼ Introduction

An operating system has two main operating modes for applications — user mode and kernel mode.

1. **User mode:**
   - **Purpose:** run general-purpose user applications.
   - **Restrictions:** limited access to system resources and various restrictions to prevent unauthorized access.
2. **Kernel mode:**
   - **Purpose:** provides the OS with privileged access to system resources and hardware.
   - **Privileges:** the kernel mode enables executions of privileged instructions, access memory directly and control hardware access.



OS program execution modes

▼ Context switching

If a program needs to perform a privileged operation (e.g., access the file system), it must request the OS to switch to kernel mode and perform the operation. The OS performs the operation in the kernel mode then returns the control to the user-mode process.

NOTE: when a program is execution in Kernel mode and if this program happens to crash during its executions then the entire system could crash.

▼ System calls

▼ Definition

When a user-mode process needs a privileged access this process makes a system call in order to switch to the kernel mode.

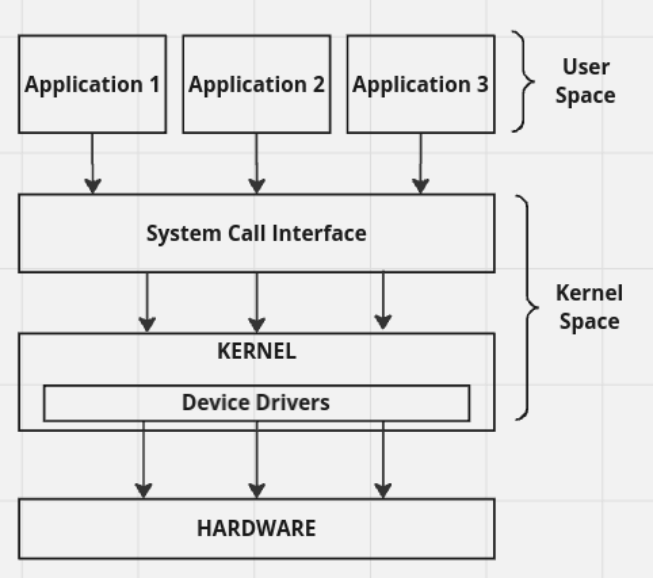
The call the process makes in order to switch modes in knows as the system call.

System call is the programmatic way in which a computer program requests a service from the kernel of the OS.

These calls are generally available as routines (a fixed set of machine instructions) written in C or C++.

▼ System calls interface

Application interact with the System Call Interface



System Call Interface in the OS (Linux, Ubuntu 22.04).

▼ Types

1. **Process control:**
   - end, abort, load, execute.
   - create process, terminate process.
   - get process attributes, set process attributes.
   - wait for time, signal event, wait event.
   - allocate, free memory.
2. **File manipulations:**
   - create file, delete file.
   - read, write a file.
   - open, close a file.
3. **Device manipulation:**
   - request, release a device.
   - read, write, reposition.
   - get, set device attributes.
   - logically attach and detach device.
4. **Information maintenance:**
   - get time or date; get, set system date.
   - get process, file or device data.

5. **Communications:**
    - used for communication between different processes and devices.
    - create, delete communication connection.
    - send, receive messages; transfer status information.
    - attach or detach remote devices.

▼ Types of programs (system & user)

▼ User programs

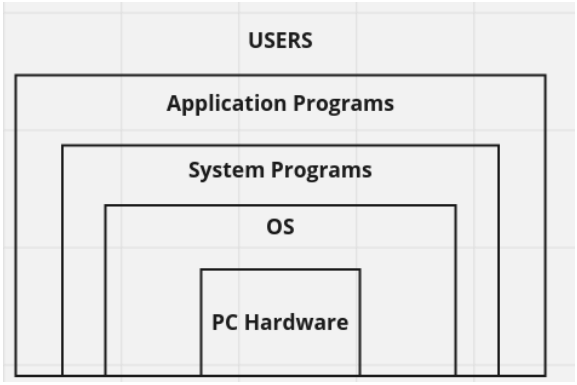
Purpose: User applications are designed to directly interact with users to perform user specific tasks.

Implementation: User applications are reliant on underlying system software (operating system) to provide essential services like I/O, memory management and process scheduling. Users control the entire application through interfaces (e.g., CLI or GUI).

▼ System programs

Purpose: Manage and control hardware and software resources of operating system. (e.g., device drivers, compilers, shells, etc.).

Implementation: May depend on another system software or the kernel for core services.



Programs hierarchy in OSes.

Memory

▼ Main memory (hardware)

▼ Introduction

Main memory, commonly known as **RAM (Random Access Memory)**, is a critical component in computer systems, bridging the gap between the **CPU** and **long-term storage**. Main memory temporarily holds data and instructions (machine code) that the CPU needs for executing programs, allowing the system to operate at high speed. Main memory is volatile, meaning it loses all the data after electrical power is lost.

▼ Structure and physical organization

Main memory is implemented using memory chips that are embedded on modules like DIMMs (Dual Inline Memory Module).

▼ Types of RAM (DRAM / SRAM)

Dynamic RAM (DRAM):

The most common type of memory, each memory cell consists of capacitor (to store bit) and a transistor (to control access to the cell).

- Widely used for main memory due to its cost efficiency and high density.
- Required periodic refreshing to maintain data, as the capacitors in DRAM cells lose charge over time.
- DRAM is slower than SRAM but can store more data in smaller area.

Static RAM (SRAM):

- Does not require refreshing, because it stores data using flip-flops circuits.
- Faster and more reliable than DRAM but it more expensive and larger.
- SRAM is used in CPU caching.

▼ Memory controller and buses

Memory controller:

- Manages data transferred between CPU and main memory.
- Handles tasks like: address decoding, data buffering, DRAM refreshing.
- In modern systems, the memory controller is often integrated into CPU chip to reduce latency, increase data throughput, and improve overall efficiency.

Data bus: controls the actual data being read form or written to memory. Typically, transfers a machine word per operation.

Address bus: carries the memory addresses that CPU uses to locate specific data in memory. The width of the address bus determines the maximum amount of memory the system can address (e.g. a 32-bit address but can address up to 4 GB of memory).

Control bus: Caries signals for read and write operations, memory refresh signals, and other control information necessary to manage memory operations.

▼ Data access in main memory workflow

When a CPU need to access data in main memory, it follows a sequence of steps using memory controller and busses.

1. **Address request:** the CPU places memory address on the address but. The address indicates a specific location in main memory where data should be read from or written

to.

2. **Address decoding:** The main memory controller decodes these addresses to locate the corresponding row and column within the memory chip.
3. **Data access:**
    - **Read operation:** the memory controller retrieves data from the specific memory cells and places it on the data bus, where CPU can access them.
    - **Write operation:** the CPU send data over the data bus, and the memory controller writes it to the specified locations in the memory cells.
4. **Refresh cycle (for DRAM):** the memory controller periodically refreshes the memory cells by reading and rewriting data to prevent data loss from capacitors discharge.

▼ Logical and physical address space

Main memory is a computer system is accessed using two primary types of addresses: logical (virtual) and physical.

Logical (Virtual) address space:

- Generated by the CPU while a program is running.
- Refers as virtual space because it represents the addresses that are seen by a program.
- Consists of all logical addresses that a process can use during its execution.

Physical address space:

- Actual addresses in the main memory.
- Represent a specific location in main memory where data or instructions reside.

▼ Virtual memory

▼ Definition

Virtual memory is a storage allocation scheme in which secondary memory can be addressed as though it were a part of the main memory. The addresses a program may use to refer to memory are distinguished from the addressed the memory system uses to identify physical storage sites and program guaranteed that addresses are transmitted automatically to corresponding machine addresses.

Virtual memory is a memory management technique that allows provides to programs appearance of memory as a long continues block, even if the physical memory is limited. Additionally, it allows OS to compensate for physical memory, enabling other (larger) application to run with less memory, and use secondary memory as it were the main memory.

The size of virtual memory is limited by the addressing scheme of the computer and the amount of secondary storage available.

Virtual Memory vs Physical Memory

Feature	Virtual Memory	Physical Memory (RAM)
Definition	An abstraction that extends the available memory by using the disk storage.	The actual hardware RAM that stores data in instructions currently being used by the CPU
Speed	Slower (due to I/O operations)	Faster (accessed directly by the CPU)
Location	On the hard drive or SSD	On the PC motherboard.
Capacity	Larger	Smaller
Data access	Indirect (via paging and swapping).	Direct (by CPU).
Volatility	Non-volatile	Volatile

▼ How does it work

This technique is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer.

- All memory references within a process are logical addresses that are dynamically converted into physical addresses as run time. This means that the process can be swapped in and out of the main memory such that it occupies different places in the main memory at different time during execution.
- A process can be broken into pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

▼ Memory Management Unit (MMU)

The memory management unit is a physical chip located on a CPU circuit providing a hardware support for memory virtualization in operating systems by translating virtual addresses into physical addresses.

MMU main functions:

1. Address translation (mapping): Translates logical addresses into physical addresses.
2. Dynamic address building: The MMU performs translation at runtime, allowing each process to operate in its own logical address space, which the MMU maps to available physical addresses in main memory.
3. Protection: Provides protection by ensuring that processes cannot access each other's memory, supporting process isolation.

Example: logical `0x2000` →physical `0xACF200`.

▼ Types of Virtual Memory

In a computer, virtual memory is managed my the Memory Management Unit (MMU), which is often build into the CPU. The CPU generated virtual addresses that the MMU translates into physical addresses.

There are two main types of virtual memory:

1. Pagination.
2. Segmentation.

▼ Dynamic loading

Dynamic loading is the technique that loads a programs's components into main memory only when needed, rather than all at once.

Dynamic loading workflow:

1. Program segmentation: a program is divided into smaller components, such as routines and functions.
2. On-demand loading: when the program needs to access a particular routine, the OS loads that specific routine into memory.
3. Code stub: if a routine is not loaded yet, the program uses a code stub, which is a small piece of code that triggers the OS to load the routine into memory.

Pros:

- Reduce memory usage: only necessary routines are loaded, leaving more memory available for other processes.
- Faster stat up: programs start more quickly because only essential (initial) components are loaded at the beginning.

▼ Dynamic linking

Dynamic linking is a process of linking a program to external libraries or modules at runtime rather than at compile time. It contrasts with static linking where all library core is compiles into the program executable code at compile time.

Dynamic linking workflow:

1. External libraries: the program refers external libraries or modules that are loaded into memory separately from the program, but does not include their code in the compiled binary.
2. Linking loader: when a program stats, the OS's linking loader dynamically links the program to these libraries.
3. Relocation table: the executable contains a relocation table with addresses for each dynamically linked function or variable, and the loader updates these references to

point to the memory addresses of the loaded libraries.

Pros:

- Reduces program size.
- Easy updates: independent libraries and modules can be updated separately.

Example: the `printf()` function in the C language. This function is dynamically loaded form the `libc.so` library at runtime.

▼ Shared libraries

Shared libraries are files containing code (functions, routines, etc) that multiple programs can use concurrently. They are commonly used in dynamic linking, allowing applications to use same library code loaded in memory.

How do shared libraries work:

1. Single copy in memory: the OS loads a shared library into memory once and maps it to the virtual address space of all programs that need it.
2. Reference counting: the OS keeps track of how many programs are using the shared library, ensuring that the library is unloaded when it’s not needed.
3. Position independent code (PIC): shared libraries are typically compiled as position independent code, allowing them to be loaded at any memory address.

▼ Swapping

Swapping is a memory management technique used by the OS to handle situations where the main memory (RAM) is fully occupied, and additional processes are needed to be executed. Swapping involves moving entire process or its pagers or its segments from the main memory to the secondary storage (disk).

Swapping workflow:

1. Process in memory: when a process is active it resides in the main memory.
2. Memory full: if the main memory is full and a higher priority process needs space, the OS moves a currently inactive or low-priority process (or its part) to the secondary storage area called a “swap space”.
3. Swapping back: when the swapped-out process needs to be executed again, the OS moves it back from the swap space to the main memory, possibly swapping another process out if necessary.

Pros:

- Efficient memory usage: allows to execute more processes than the physical memory can hold simultaneously.
- Prioritization: higher priority processes are keeps in the main memory.

Cons:

- Swap time: swapping is an expansive operation due to slow read/write speed of the secondary storage.
- Disk I/O overhead.

▼ Fragmentation

Memory fragmentation is a type of memory disruption that can lead to inefficient memory usage by leaving some blocks of memory unused.

Types of memory fragmentation:

1. **External fragmentation:** Occurs when memory is split into small, non-contiguous blocks that are too small to satisfy allocation requirements, even if the total free memory is sufficient.

   Example: If a process allocates a 4 KB block, but the free memory is scattered into multiple blocks 1 KB each, no new 4 KB process can be allocated despite sufficient total free memory.

2. Internal fragmentation: Occurs when memory is allocated in fixed-size blocks, and the process does not fully use the allocated space, leading to wasted memory within the block.

   Example: a process uses 3 KB of 4 KB, hence the fragmentation is 1 KB.

▼ Memory allocation

Memory allocation is how the OS assigns main memory to the processes. Proper memory allocation strategies ensure efficient use of RAM and optimize system performance.

Types of memory allocation:

1. **Contiguous**: each process is allocated a single contiguous block of memory.

   Pros:

   - Simple and fast to manage due to straightforward memory address calculation.

   Cons:

   - Lead to memory fragmentation: unused memory blocks scattered between allocation segments, limiting efficient memory use.

2. **Non-contiguous:** a process is allocated memory in multiple non-contiguous blocks. this method is more flexible and supports pagination and segmentation.

   Pros:

   - Reduces memory fragmentation.

   Cons:

- May increase memory access time due to additional complexity of managing non-contiguous blocks.

▼ Memory allocation strategies

Strategies used to allocate memory blocks for processes.

1. **First-fit:** The allocator scans memory and assigns the first available block that is large enough for the request.

   Pros:

   - Fast and Simple.

   Cons:

   - Tends to create external fragmentation over time.

2. **Best-fit:** The allocator finds the smallest available block that fits the request, minimizing wasted time.

   Pros:

   - Reduces wasted space in the allocated block.

   Cons:

   - Slower.
   - Might increase external fragmentation by leaving small gaps.

3. **Worst-fit:** The allocator chooses the largest available block, leaving the largest leftover place.

▼ Segmentation

▼ Introduction

Segmentation is a memory management technique that divides a process's memory into variable-size segments based on logical division within the program.

Key concepts of segmentation:

- **Logical division:** Memory divided based on logical segments of a process, reflecting how a program is structured (e.g, functions, arrays, modules).
- **Variable segment size:** Each segment has a custom size.
- **Segment table:** Each process has a segment table that stores the base address and limit (size) for each segment, allowing the OS to map logical segments addresses to physical memory locations.

NOTE: Modern operating systems do not use segmentation, instead they use pagination only.

▼ Segment tables

The segment table is a data structure that maps logical segments to physical memory. It holds two main pieces of information for each segment:

- **Base address:** The starting physical address of the segment in main memory.
- **Limit:** The length of the segment, defining its size.

Address translation using the segment table

- When a process references a memory location using logical address, the logical address contains segment number and the offset.
- The segment is used to index the segment table and retrieve the base address and the limit.
- The offset is used to calculate the physical address.

Example: If a logical address (e.g. `0x10AFB`) references segments 2 within an offset of 500.

1. The OS checks segment table entry for the segment 2 to find the base address (e.g. 1000).
2. The physical address is calculated as $1000 + 500 = 1500$.
3. the OS verifies that offset does not exceed the limit of the segment to prevent memory violation.

Pagination

Introduction

Pagination is a memory management scheme that eliminates the need for contiguous allocation of physical memory. Instead, it allows processes to be divided into fixed-size blocks of virtual memory called pages, which are located into equally-seized blocks in physical memory called frames.

Key concepts:

- Pages: Fixed-size blocks of a process's virtual (logical) memory.
- Frames: Fixed-size blocks of physical memory.
- Page table: A data structure used to map logical addresses to physical addresses.

Page tables

The page table is central for pagination process. It stores the mapping between logical page numbers (addresses) that is used by a process, and physical frame number (addresses) that is used by the hardware memory controller.

Page table structure: Page tables are often implemented as hash maps that values point to arrays, and keys are hashed virtual addresses. Each array is referred as the page entry. Each entry in a page corresponding to a logical page (key in a hash table) and contains the frame number (address) where the page resides in the physical memory.

Page Table Entry (PTE): Each entry in a page contains crucial information to manage and translate logical addresses into physical addresses.

Components of a page entry (PTE):

- **Frame number:** the number (address) of the physical frame holding the page.
- **Valid/Invalid bit:** indicates whether the page is currently in the main memory (valid) or swapped out to the disk (invalid).
- **Protection bit:** specifies permissions (e.g. read, write, execute) for page.
- **Referenced bit:** indicates if the page has been accessed recently (used for page replacement algorithms).
- **Dirty bit:** indicated if the page has been modified; used to decide if the page needs to be written back to the disk ?

▼ Types of page tables

1. **Hierarchical:** hierarchical is used to manage very large page tables which ban be implemented for modern systems with large address spaces. By splitting the page into sub-tables.

   How it works:

   - The logical address is divided into multiple parts, each part represents an index into different levels of the page table.
   - First-part index into first level table, which points to the second level table.
   - The process continues until the final table provides the frame number.

2. **Hashed:** Hashed tables are used for systems with large address space, such as 64-bit systems, where even hierarchical page table may become too large.

   How it works:

   - The virtual page number is hashed to create an index into a hash table.
   - Each entry in a hash table points to a linked list (or array) of page table entries that have the same hash value.
   - The linked list (or array) is searched linearly to find the correct entry.

3. **Inverted:** inverted pages are another method designed to address the high memory cost of traditional page tables, especially in large address spaces.

   How it works:

   - Mapping starts from a physical frame address back to logical page address, despite the actual address translation begins from virtual space and ends up with a physical memory frame.

- Whether or not the page is in the main memory, there is always place reserved for it.

▼ Shared pages

Shared pages allow multiple processes to access same physical memory frames. This is commonly used for shared libraries, or read-only code segments.

Implementation

The page table of each process has an entry pointing to the same frame in physical memory. Ensure that shared pages are accessed appropriately (e.g. protections bits are set as read-only).

▼ Demand paging

Demand paging is a virtual memory technique where pages are loaded into main memory only when they are needed during the course of a program execution, rather than preloading the entire process into the main memory.

Demand paging workflow:

1. **Page request:** the process requests the page, and the OS checks the page table to see if the page is active.
2. **Page fault:** if the page is not in the main memory, the page fault occurs.
3. **Page load:** the OS pauses the entire process and loads page into the memory.
4. **Process resumption:** after loading the page the OS continues the process execution.

Performance:

The performance of demand paging depends on the page fault rate and the cost of handling a page fault. The page demanding performance is calculated using the effective access time (EAT) equation:

$$EAC = (1 - p) * MemoryAccressTime + p * PageFaultServiceTime.$$

Where:

- p - probability of a page fault (page fault rate)
- Memory access time - time to access main memory.
- Page fault service time - includes the time to handle a page fault.

Example: given memory access time = 100 ms, page service tome = 10 ms (10_000_000 ns). For a page fault rate p = 0.001.

The $EAC = (1 - 0.001) * 100 + 0.001 * 10000000 \approx 10.099ns$

Demand paging performance can be optimize by efficient page replacement algorithms such as LRU (Least Recently Used).

▼ Thrashing

Trashing occurs when the system spend more time handling page faults than executing the actual process due to high page fault rate.

Solutions:

1. Working set model: keeps track of pages a process needs to over specific time interval and ensures that these pages stay in memory
2. Page fault frequency (PFF): monitor page fault rate and adjust the allocation of memory per process accordingly.

▼ Copy-on-Write (CoW)

Copy-On-Write (CoW) is an optimization strategy used in demand paging to efficiently handle process creation operations as `fork()`.

Copy-On-Write workflow:

1. **Initial sharing:** when a process is forked, both parent and child processes share the same physical memory pages.
2. **Making as read-only:** the shared pages are marked a read-only.
3. **Copy-on-Modification:** if either process attempts to modify a shared page, the OS makes a copy of page for the modifying process. This ensures that each process has its own copy after modification.

▼ Page replacement algorithms

▼ First-in-First-out (FIFO) replacement

Method: replaces page that has been in memory the longest.

Implementation: pages are stored in a queue, and the page in the front is replaced when a page fault occurs.

Pros: Simple to implement.

Cons: sub optimal performance, as the page access frequency is not considered.

Besady's anomaly: a situation when increasing the number of page frames can lead to more page faults, which is unexpected because adding more memory should ideally decrease page faults or keep them the same.

▼ Optimal page replacement

Methods: replace page that will not be used for the longest period of time in the future.

Implementation: the algorithm is theoretical.

Pros: Provides lowest possible page fault rate.

Cons: Not feasible to implement due to the need for future knowledge.

▼ Least Recently Used (LRU) replacement

Method: replace pages that has not been used for the longest time.

Implementation: requires tracking the usage of pages, typically by using counter or stack.

Pros: efficient in practice, as pages that have not been used recently are less likely to be required soon.

Cons: complex implementation and required hardware and software overhead.

Implementation example:

```
# Using a doubly linked list and a hash map for efficient LRU \
# implementation

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        # Hash map to store pages (key: page number, value: node)
        self.cache = {}
        # Doubly linked list to track LRU order
        self.order = DoublyLinkedList()

    def access_page(self, page):
        if page in self.cache:
            # Move the page to the front (most recently used)
            node = self.cache[page]
            self.order.move_to_front(node)
        else:
            # Page fault, need to load the page
            if len(self.cache) >= self.capacity:
                # Remove the least recently used page from the end
                lru_page = self.order.remove_from_end()
                del self.cache[lru_page]

            # Add the new page to the front
            new_node = self.order.add_to_front(page)
            self.cache[page] = new_node

class DoublyLinkedList:
    # Methods to add, remove, and move nodes within the list
```

▼ Least Frequently Used (LFU) replacement

Method: replace the page that has been accessed the least number of times.

Implementation: a counter is assigned to each page to track how often it is accessed.

Pros: ensures that infrequent pages are replaced.

Cons: can perform poorly if certain pages were heavily used at one time but are not longer required.

▼ Page buffering algorithm

▼ Introduction

The page buffering algorithm is used in OSes and database management systems as a key method to streamline data access and minimize disk I/O operations. It's largely used in virtual memory systems, where data is kept on secondary storage and brought into memory on demand.

The primary goal is to reduce latency assassinated with accessing the data on disk. The approach optimizes performance by intelligently buffering frequently visited pages in memory, minimizing the requirement for disk I/O operations.

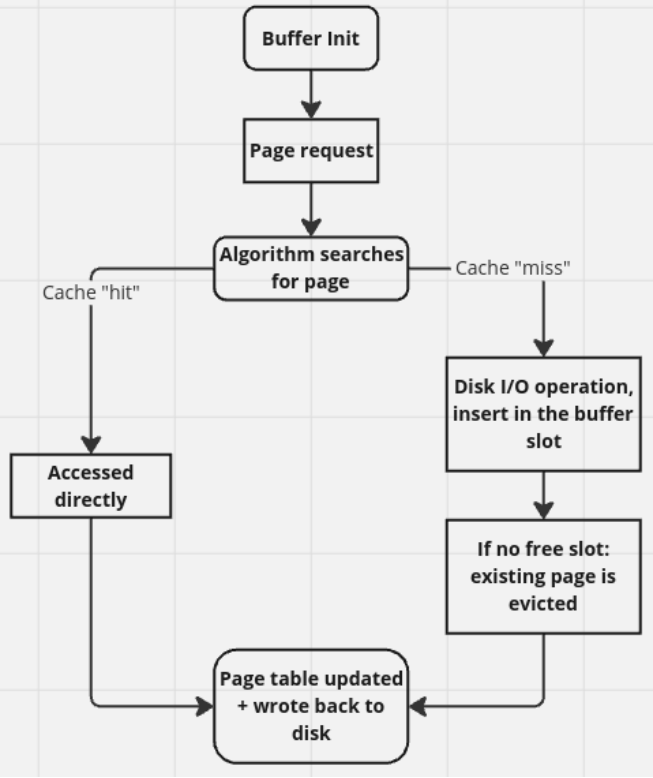
▼ Terminology

- **Buffer (Cache):** the technique is used in this algorithm keeps a portion of pages that are currently stored in on the disk in a buffer (cache) that is located in the main memory. The buffer serves as a short-term repository for frequently used pages.
- **Page request:** the OS determines if the page is already in the buffer when a process requests a specific page.
- **Eviction approach:** the page buffering algorithm uses eviction approach to free up space for freshly requested pages, because the page buffer has a finite capacity. A page replacement policy is used to determine which page or pages should be replaced to make room for new pages.
- **Locality of reference:** the notion of locality, which asserts the recently viewed pages are likely to be accesses again soon.

▼ Workflow

1. **Buffer initialization:** a portion of main memory, known as buffer or cache, is reserved to hold a subset of pages from secondary storage (disk) and it is initially empty.
2. **Page request:** the OS determines if a requested page is already in the buffer.
    a. Cache "hit": the page can be accesses directly.
    b. Cache "miss": the disk I/O operation is triggered to load the page.
3. **Buffer management:** the page buffering algorithm controls the buffer as pages as added. An eviction approach is used to free up space. In this process following page replacement algorithms can be used: FIFO, LRU, Clock algorithm.
4. **Access and Update:** a page may be directly read and changed in memory once it is in the buffer, obviating the requirements for disk I/O operation. Data consistency is ensured by the eventual propagation of any changes in the buffer back to secondary storage.

5. **Locality of reference:** recently viewed pages are likely to be accessed again soon. The program predicts future access by buffering these frequently used pages in memory.



Page buffering workflow scheme.

▼ Implementation

1. Data structure chosen to represent the buffer (cache) in memory.
2. Page table is maintained and updated. which stores all mapping of the virtual memory addresses and the corresponding pages in buffer.
3. The the initial state, the buffer is empty and the page table entries are initialized accordingly and the status bits are set correspondingly. The status bits indicate whether the page is currently in the buffer or not.
4. The page buffering algorithm decides whether the page is requested or not , after that it checks whether the page is already in the buffer or not.
5. The page buffering method adjusts continuously to processes' shifting access patterns. It forecasts future access patterns and modifies the buffer content dynamically.

▼ Frames allocation

▼ Introduction

In virtual memory systems, frames are fixed-size blocks of physical memory (RAM) into witch pages from processes virtual address space are loaded.

Key characteristics of frames:

- Fixed size: frames are the same size as pages (typically 4 KB), ensuring easy mapping of virtual pages to physical frames.
- Page mapping: each virtual page of a process is mapped to a frame in physical memory via the page table.
- Efficient use of memory: since frames and pages are the same size, there is no need for memory compaction.

▼ Allocation

Frame allocation determines how frames are assigned to processes in the system. When multiple processes compete for physical memory, the OS must decide how to divide the available frames among them.

Key questions in frame allocation:

- How many frames should each process have ?
- Which frames should be allocated to a process during execution ?
- What should happen when a process requires more frames than available ?

Minimum frames per process: each process requires a minimum number of frames to execute. For example, the number of frames required to hold all instructions for a single machine cycle (like a CPU instructions plus its operands).

Maximum frames per process: the total number of available frames in physical memory determines the maximum allocation.

▼ Algorithms

There are two main strategies in frames allocations.

Equal allocation: divide the total number of frames among all processes. Example: if 100 frames are available and there are 5 processes, each process gets 25 frames.

Pros:

- Simple implementation
- Equal allocation ensures no single process monopolizes memory.

Cons:

- Not efficient if processes have varying memory needs. (e.g. larger process may not get enough frames).

Proportional allocation: allocate frames based on the size of each process related to the total size of all processes. Example: a process A requires 10 frames, and the process B

requires 30 frames. If 100 frames are available, A gets $\frac{10}{40} * 100 = 25$ frames, and B gets $\frac{30}{40} * 100 = 75$ frames.

Pros:

- improved efficiency.

Cons:

- Can lead to starvation of smaller processes have insufficient frames.

▼ Global vs local allocation

The number of frames per process can be also dynamically changed. The frame allocation can be further divided into two strategies:

1. **Local allocation:** Each process is assigned a fixed number of frames, and when a page replacement is required the replacement occurs only within this process's allocated frames. A process can only replace its own frames, isolating it from other processes.

   Pros:

   - Prevent interference between processes.
   - Ensures predictable performance for individual processes.

   Cons:

   - Less flexible as memory requirements for process change over the time.

2. **Global allocation**: Frames are allocated to processes from a single, shared pool of frames. When a process need to perform a frame replacement, it can replace a frame belonging to any processes.

   Pros:

   - Dynamically adjusts to changing needs of processes.
   - More flexible as processes can use memory as required.

   Cons:

   - Can lead to process interference when a process affects another process by stealing its frames.
   - Unpredictable performance for individual processes.

▼ Working set model

The working set model is a memory management strategy that aims to allocate frames dynamically based on the current memory needs for each process. It helps in reducing thrashing by ensuring that a process has enough frames to hold its working set.

Working set definition: the working set of a process is the set of pages that it is actually using during a given time interval D. This set changes overtime as the process moves in

different phases of execution.

How it works:

1. Track recent page replacement: the OS monitors the page accessed by a process within the interval D.
2. Adjust frame allocation:
    a. If the working set grows, the process may need more frames.
    b. If the working set shrinks, the process may release frames for other processes.
3. Page fault reduction: by ensuring that the working set is fully in memory, the OS minimizes page faults.

▼ Memory layout of C programs

▼ Introduction

Process memory layout is a fundamental concept of how OS organizes virtual memory for running programs (processes). Each process in a computer system is provided with its own virtual memory space, which is divided into distinct sections for different purposes. This organization ensures isolation, efficiency, and scalability in multitasking environments.

A process memory layout refers to logical arrangement of memory allocated to a process during its execution. The memory is divided into distinct segments or regions, each serving a specific role. This segments include the text, data, heap, stack, shared libraries.

Key features:

- **Isolation**: Each process gets its own memory space, with the OS maps to physical memory.
- **Flexibility**: Virtual memory allows process to use memory without worrying about the underlying hardware components.
- **Protection**: The OS enforces access controls, ensuring processes cannot interfere with each other.

▼ Components of process memory layout

1. **Text segment:** Contains the compiled program code (executable binary code).
    - Marked as “read-only” to protect accidental or malicious modifications of the executable code.
    - Can be shared among processes running the same program to save memory.
    - Occupies the lowest part of the memory address space.
2. **Data segment:** Stores global variables, static variables, and initialized data.

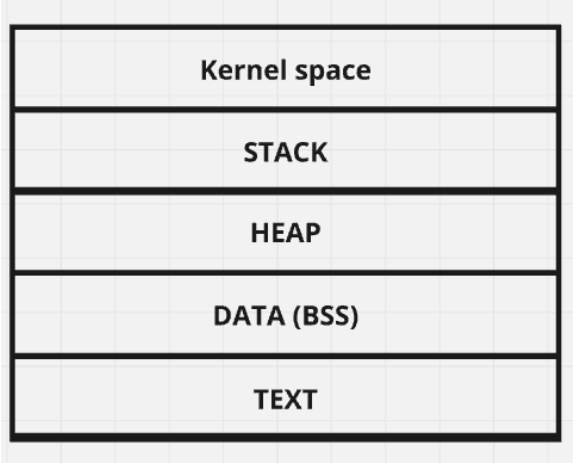
    Subdivision:

- Initialized data: Contains variables that are explicitly initialized in the program (e.g., `int x = 10;`.
- Uninitialized data (BSS): Contains uninitialized global and static variables. Process can modify the data in this segment during the execution. Initially, the OS initializes uninitialized variables to 0.

3. **Heap segment:** Used for dynamic memory allocation during the runtime (e.g., `malloc()` in C).
   - Grows upward in memory (toward higher addresses) as more memory allocated.
   - Managed explicitly by a programmer or dynamically by the runtime environment.
4. **Stack segment:** Stores function call frames, including local variables, return addresses, and function arguments.
   - Grown downwards in memory as more function calls are made.
   - Automatically managed by the OS; memory is allocated when a function call is made and deallocated when it returns.
   - If the stack grows too large and collided with the heap, a stack overflow occurs.
5. **Shared libraries and memory-mapped segments:**
   - Shared libraries: store code for dynamically linked libraries (e.g., files with the "*.so" extension in Linux, and files with the "*.dll." extension in Windows.
   - Memory-mapped segments: used for efficient file I/O and inter-process communication. Allows processes to map files directly into memory, enabling fast access.
6. **Kernel space:** In systems with virtual memory, part of the address space is reserved for the kernel. Kernel space manages OS operations (e.g. handling interrupts, managing hardware resources). Not directly available to user-mode processes to ensure security and stability.

▼ Typical process memory layout

Tho process memory layout may be influence by a runtime environment of a process.



Typical process memory layout

▼ Example in Linux

In Linux, a process's virtual memory layout can be inspected using `/proc/[PID]/maps` or `pmap`.

Example of `/proc/self/maps` Output

Running `cat /proc/self/maps` for a running process might show:

```
00400000-0040b000 r-xp 00000000 08:01 123456 /bin/bash
0060a000-0060b000 r--p 0000a000 08:01 123456 /bin/bash
0060b000-0060c000 rw-p 0000b000 08:01 123456 /bin/bash
7ffd3a5be000-7ffd3a5df000 rw-p 00000000 00:00 0  [stack]
7ffddf5ee000-7ffddf5ff000 r-xp 00000000 08:01 654321 /lib/libc.so.
```

Explanation:

- **Text Segment**: `/bin/bash` executable code (`r-xp` indicates read, execute permissions).
- **Data Segment**: Writable section of the binary.
- **Stack**: Marked as `[stack]`.
- **Shared Libraries**: Dynamically loaded libraries like `libc.so.6`.

▼ Process runtime environment

▼ Introduction

A runtime environment bridges the gap between the high-level code written by programmers and the underlying system resources, enabling programs to execute consistently and effectively across different platforms.

A runtime environment (RTI): is a software layer that provides the necessary resources, tools, and abstraction to execute a program. It encompasses everything required for the program to run, including:

- Memory management
- I/O operations
- System Calls handling
- Error handling and exception processing
- Interactions with the OS and hardware.

Key features of runtime environments:

1. Platform abstraction: abstracts hardware and OS details so programs can run independently of the underlying platform.

   Example: Java Virtual Machine (JVM) enables Java applications to run on different OSes.

2. Execution context: provides a controlled execution environment where programs execute securely and predictably.
3. Dynamic features: enables features like dynamic linking, memory allocation, and garbage collection during program execution.

▼ Components

1. **Runtime libraries:** provide precompiled code and API that program can use at runtime.

   Example: `libc` in C/C++ programs provide standard functions like `printf()` and `malloc()`.

2. **Virtual machines:** executes programs in virtualized environment, ensuring platform independence.

   Example: Java Virtual Machine (JVM), .NET Common Language Runtime (CLR) for .NET applications.

3. **Memory management:** handles dynamic memory allocation, deallocation. and garbage collection. Ensures efficient use of memory and prevents memory leaks.
4. **Exception handler:** detects and handles runtime errors such as illegal memory access, divisions by 0 or stack overflow. Provides mechanisms like “try-catch” block to manage exceptions gracefully.
5. **System call interface:** mediates interactions between the program and the OS.

   Example: file I/O operations like `open()` and `read()` in Linux.

6. **Execution engine:** Interprets or compiles code into machine instructions for execution.

   Example:

   - Interpreter: execute code line-by-line (e.g. Python interpreter).
   - Just-in-Time (JIT) Compilers: dynamically compile high-level code during execution (e.g., JVM’s HotSpot JIT compiler).

▼ Types

1. **Native runtime environment:** directly interact with the hardware and OS.

   Example:

   - C runtime environment (CRI): provides runtime support for C programs via libraries like `glibc` (GNU C lib).
   - POSIX runtimes: support for applications written for POSIX-compliant systems.

2. **Managed runtime environment:** provides advanced features like garbage collection, dynamic typing, and secure execution.

   Example: Java Runtime Environment (JRE): includes the JVM, standard libraries, and runtime tools. Ensures platform independence for Java programs.

3. **Web runtime environment:** designed for execution web-based applications:

   Example:

   - JavaScript runtime (e.g., Node.js): executes JS outside the browser, enabling server-side applications.
   - WebAssembley (Wasm): allows high performance code to run in browser.

▼ Runtime environments workflow

Step-by-Step execution:

1. **Loading**: the programs and required libraries are loaded into memory.
2. **Initialization**: the runtime environment initializes global variables (DATA/BSS), sets up the stack and allocated heap space.
3. **Execution**: the code in interpreted or compiled into machine instructions by the execution engine.
4. **Resource management**: the runtime manages resources like memory, file handlers, and threads.
5. **Termination**: clean up memory and resources upon program completion.

File Systems

▼ File system

▼ Core concept

A file system is the method of organization and storing data on a storage device, such as hard drive, solid state drive, etc. It provides a hierarchical structure of directories and files, enabling efficient storage, retrieval, and management of data.

A file system comes with a driver that implements algorithms to perform operation with the file system (i.e., read, write, data blocks, etc.). A driver typically comes as an OS kernel library.

Key concepts of a file system:

- **File:** the basic unit of data storage, containing a sequence of bytes.
- **Directory (folder):** a container for files and other directories, organizing data into hierarchical structure.
- **Metadata:** information about a file, such as its size, creation date, and permissions.
- **File system driver:** the key component of a file system. It is software component that interacts with the storage device and implements the file system's logic.

▼ Core functionalities

- **Storage allocation:** assigning a space to file and directories.
- **File access:** locating and retrieving files based on their names and paths.
- **File creation and deletion.**
- **File modification:** updating content of existing files.
- **File permissions:** controlling access to files and directories.
- **Disk space management:** tracking free and unused space on the storage device.

▼ Types

- **FAT32:** An older file system that is still widely used for older devices and USB drives. It is simple and easy to use, but it has a number of limitations, such as a maximum file size of 4 GB and a maximum partition size of 32 GB
- **NTFS:** The default file system for Windows. It is more advanced than FAT32 and supports larger files and partitions. It also has features such as encryption, compression, and file system journaling
- **EXT4:** The latest version of the ext file system. It is more efficient and reliable than ext3, and it supports larger file sizes and partitions
- etc.

▼ Files

▼ Introduction

A file is a fundamental unit of storage in a file system, representing a sequence of bytes stored on a disk. Files abstract storage details, enabling users and programs to store and retrieve data conveniently.

Key characteristics:

- **Logical structure:** files are typically organized as sequence of bytes, regarding of the data they contain (text, binary, etc.).
- **Persistence:** files remain store on the disk even after the power supply is off.
- **Name:** a unique identifier used to reference the file.

▼ File structure

- **Unstructured files**: treated as a raw sequence of bytes. They can contain text, images. audio, etc. While they may have some internal structure, it is not as rigid as structured data.
- **Structured files:** are organized in a predefined formats, typically tubular or hierarchical. They adhere a specific data model or schema, which defines the structure and relationships between different data elements. This make them easily searchable and analyzable and machine-readable.

  Example:

  - JSON: a lightweight format of data interchange.
  - XML: uses tags to define the structure and content of files.
  - CSV: a simple text file where data is separated by commas.

Key differences between structured and unstructured files:

Feature	Structured	Unstructured
Organization	predefined format	no predefined format
Accessibility	easily searchable and analyzable.	more challenging to search and analyze.
Storage	often stored in database.	stored in file system
Example	JSON, XML, CSV	text documents, images, ...

Example in `etx4` file system, fines are represented as inodes (index nodes) containing metadata and pointers to data blocks.

▼ File attributes

Files have associated metadata, referred to as file attributes, which describe their properties.

Common file attributes:

1. Name: human-readable name of a file.
2. Identifier (Inode number): a unique number identifying the file wile within the file system.
3. Type: indicates the nature of the file (regular, link, directory, etc.).
4. Size: the size of a file in bytes.

5. Location: a pointer to the physical storage block where the file data is stored.
6. Protection: access permissions for different user categories. (e.g., owner, group, others). Example: `rwxr-x-r--`.
7. Time stamps: file creation, modification, access time.

▼ File operations

Operating systems provide a standard set of **file operations** that allow users and applications to interact with files.

Key Operations:

1. **Create**:
   - Creates a new file in the file system.
   - Example: `touch newfile.txt` in Linux.
2. **Open**:
   - Prepares a file for reading, writing, or both by creating a file descriptor.
   - Example: `open()` system call in Linux.
3. **Read**:
   - Retrieves data from the file starting at the current file pointer position.
   - Example:

     ```
     ssize_t bytes_read = read(fd, buffer, size);
     ```
4. **Write**:
   - Writes data to the file starting at the current file pointer position.
   - Example:

     ```
     size_t bytes_written = write(fd, buffer, size);
     ```
5. **Close**:
   - Releases resources associated with an open file descriptor.
   - Example:

     ```
     close(fd);
     ```
6. **Delete**:
   - Removes a file from the file system, freeing its storage blocks.

- Example: `rm file.txt` in Linux.

7. **Seek**:
   - Moves the file pointer to a specific position for reading or writing.
   - Example in Linux:

```
off_t offset = lseek(fd, position, SEEK_SET);
```

8. **Rename**:
   - Changes the name of the file.
   - Example: `mv oldname.txt newname.txt`.

▼ File types

Files can be classified based on their purpose and format.

Common file types:

1. **Regular files ("-"):** store user data such as text, images, videos, binary programs, etc.
2. **Directories ("d"):** special files that contain pointers to another files or directories.
3. **Special files:** represent hardware devices or provide special functionality.
   a. **Character device ("c"):** for character-by-character data transfer (e.g., `/dev/tty`).
   b. **Block device driver ("b"):** for block data transfer (e.g., `/dev/sda`).
4. **Symbolic links ("s"):** pointers to another file or directory.
   a. **Hard** links (not symbolic): direct reference to Inode, multiple names for the same data, cannot point to files on different file system, deleting the original file (Inode) does not effect hard links and the original data (as the original Inode is not deleted only the name). Use case: copying important data.
   b. **Soft** (symbolic) links: indirect reference via a path, deleting the original file (Inode) make a soft link invalid. Use case: creating a shortcut.
5. **Pipes and Sockets:** used for inter-process communications.

▼ Access methods

Files can be accessed using different methods depending on their structure and use case.

A. Sequential Access

- Data is read or written sequentially from the beginning to the end of the file.
- Simplest and most common method.
- Example:

```
FILE *fp = fopen("file.txt", "r");
while (fgets(buffer, size, fp)) {
    // Process line
}
fclose(fp);
```

B. Direct (Random) Access

- Allows reading or writing data at specific positions in the file.
- Useful for databases or large files where only specific parts need to be accessed.
- Example in Linux:

```
lseek(fd, offset, SEEK_SET);
read(fd, buffer, size);
```

C. Indexed Access

- Uses an index to locate file blocks, enabling efficient random access.
- Example: A database file with an index for quick lookups.

D. Memory-Mapped Files

- Maps a file directly into the process's virtual memory, enabling faster access.
- Example in Linux:

```
void *mapped = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
```

▼ Directories

▼ Introduction

A directory servers as a container for files and sub-directories, enabling hierarchical organization and efficient file management.

A directory is a special type file used by a file system to store metadata about files and other directories. It acts as a catalog or index for files, enabling users to locate, organize, and access data.

▼ Directory entries

A directory file contains entries referred as directory entries. Each entry in a directory contains information about a file or sub-directory such as filename, Inode or metadata that links to a file attributes, type (regular file, directory, symbolic link, etc.).

▼ Types of directory structures

File system can implement directories in different structures, ranging from simple flat layouts to complex hierarchical.

1. **Single-level directory:** organizes all files into a single flat directory.

   Pros: simplicity, efficient access.

   Cons: name conflicts, poor scalability.

2. **Two-level directory:** provides a separate directory for each user. Root directory contains sub-directories for each user.

   Pros: less name conflicts, improved organization.

   Cons: limited sharing, scalability issues.

3. **Tree-structured directory:** organizes directory and files hierarchically. forming a tree-like structure. The root `/` is the starting point. Sub-directories can contain files or other directories.

   Pros: efficient organization, scalability, file sharing via hard and soft links.

   Cons: complexity, path dependency: files must be accessed using a full path unless current working directory is set.

4. **Acycle graph directory:** allows directories and files to have multiple paths, supporting file sharing. Implements links (hard and soft) to allow multiple references to a the same file of directory. Ensure no cycles are formed in the graph.

   Pros: efficient sharing, space efficiency.

   Cons: complex maintenance, link breakage.

5. **General graph directory:** allows directories and files to form a general graph where cycles can exist.

   Pros: maximum flexibility, , efficient sharing.

   Cons: cycle detection, high complexity.

▼ Mounting file systems

▼ Mount / Unmount file system

Mounting is a process of making a file system accessible to the OS and its users by attaching it to a directory in the existing file system hierarchy. Once mounted, the content of the mounted file system appear as they are part of the main file system.

Steps to mount a file system:

1. Locate the file system: identify the storage device or or partition that holds the file system (e.g., `/dev/sda1`).
2. Specify a mount point: choose an empty directory in the file system where the new file system will be attached (e.g., `/mnt` or `/media/usb`).

3. Perform the mount: the OS integrates a new file system into the directory tree. (e.g., `mount /dev/sda1 /mnt`).

Types of mounts:

1. Temporary mount: the file system is mounted for the current session and is unmounted on reboot.
2. Persistent mount: the file system is configured to mount automatically at boot time, typically via entries in `/ets/fstab` in Linux.

Unmounting a file system:

1. Stop any processes that use the file system.
2. Unmount `umount` in Linux.

File sharing

File sharing refers to the ability of multiple users, devices or processes to access files concurrently. This can occur locally on the same system or across a network.

Types of file sharing:

1. Local sharing: multiple users on the same machine.
2. Network sharing: Network File Systems, Samba.

File sharing methods:

1. Shared directories: users with appropriate permissions can share files in a common directory.
2. Symbolic links: link points to shared file stored elsewhere in the file system.
3. Distributed file systems: NFS, SMB.

Remote file systems

Remote file systems allow files stored on one machine to be accesses and manipulated by another machine over a network as they were local.

Key protocols and technologies:

- NFS (Network File System): `mount -t nfs /server:shared /mnt`.
- SMB/CIFS (Server Message Block, Common internal file system): Commonly used for sharing on Windows machines. Supported on Linux via Samba.
- AFS: scalable with strong authentication.

Space allocation

Introduction

Space allocation strategies determine how the file system organizes and stored file's data blocks in a disk.

▼ Contiguous allocation

A file's data blocks are stored in consecutive location on the disk.

- The file system allocates a single, contiguous range of disk blocks for the entire file.
- Metadata: stores the starting block address and the length of the file.

Pros:

- Fast access science all blocks are allocated contiguously.
- Straightforward random access. Address = starting block + offset.
- Simple implementation.

Cons:

- External fragmentation.
- File growth issues. If a file grows beyond its allocated space, reallocation of a file to another contiguous region is costly (O(n)).

▼ Linked allocation

A file's data blocks are scattered across the disk and linked using pointers.

- Each file block contains the file data, and a pointer to the next block in the sequence (or a NULL pointer if the block is the last one).

Pros:

- Dynamic allocation reduces fragmentation.
- File growth dynamically without the need for reallocation.

Cons:

- Slow random access (O(n)).
- Pointer overhead: pointers consume additional storage space.
- Reliability: if a pointer is corrupted, the file chain may break, causing data loss.

▼ Indexed allocation

A file system maintains an index block that contains pointers to all the file's data blocks.

- The index block stores an array of pointers, each pointing to a data block of the file.
- Metadata includes the locations of the index block.

Pros:

- Fast access.
- Support large files.

Cons:

- Overhead: index block consuming additional space.
- Limited file size. A file cannot be larger than there are pointer to different blocks in the index block.

Variations:

1. Single-level indexing: one index block per file.
2. Multi-level indexing: entries in one index block can point to another index block, allowing large files.
3. Combined scheme (e.g., UNIX Inodes): uses direct pointers, single-level, and multi-level indexing.

▼ Comparison of allocation strategies

Feature	Contiguous	Linked	Indexed
Access speed	Fast (seq/random)	Slow (sequential only)	Fast (random only).
Space utilization	Poor (fragmentation)	Good	Good
File growth support	Poor (reallocation)	Good	Excellent
Overhead	Low	Moderate (pointers)	High (Index blocks)
Reliability	High	Low (pointer corruption)	High

▼ File System Drivers

▼ Introduction

A **file system driver (FSD)** is a special software module that acts as interface between the OS and a specific file system. It enables the OS to read from, write to, and manage storage devices formatted with various file systems.

Key characteristics:

1. **File system-specific:** each file system typically requires a dedicated driver.
2. **Kernel module:** in most modern OSes, FSD are implemented as kernel modules for efficient integration with the storage stack.
3. **Abstraction**: FSD abstracts the details of the file system providing a unifies interface foe file operations.

Role of file system drivers:

- File system management.
- File operations: read, write.
- Metadata handling.
- Disk space management: handle space allocation, fragmentation.

- Error handling.

▼ Architecture of file system drivers

The architecture of file system drovers integrated with the OS kernel and the I/O subsystem.

Layers in file system management:

1. **Application layer:** user-layer applications issue file operations through system calls (e.g., `read()` or `write()`, etc.).
2. **Virtual file system (VFS):** a kernel module that provides a uniform interface for file operations, regarding of the underlying file system. Translates generic operations into file system-specific calls.
3. **File system driver (FSD):** implements file system specific logic, interacting with storage medium based in the file system format.

   Example:

   - `ext4` driver for `ext4` file system.
   - `ntfs-3g` driver for NTFS in Linux.
4. **Block device driver:** manages communication with the physical storage device (e.g., SSD, HDD) by handling blocks of data.

▼ Types of file system drivers

- **Native FSD:** provided by the OS to support default file systems.
- **Third-party FSD.**
- **Network FSD:** enable remote file systems to be mounted and accesses as there were local.
- **Virtual FSD:** represents file system that do not rely on physical storage. Example: `procfs`, `tmpfc` in Linux.

Processes and Threads (Process management)

▼ Processes

▼ Process management in Linux

Definition

Key characteristics:

Process attribute State (process state):

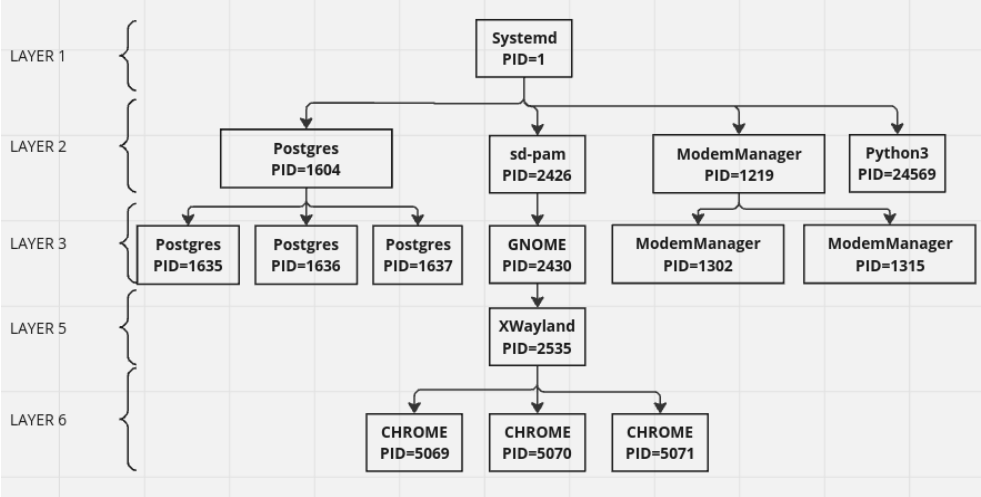
Each process has the state attribute, which can be checks at HTOP.

1. **Running / Runnable (R)**: running is using CPU at the moment. Runnable is the process that is expecting to get CPU time.
2. **Uninterruptible Sleep (D)**: Process is waiting for the I/O operation to finish (this process cannot be killed).
3. **interruptible Sleep (S)**: Process is waiting for an event (e.g., user input) and does not utilize the CPU time.
4. **Stopped (T)**: Process has received the `SIGSTOP` or `SIGTSTP` and stopped working. (Can be resumed by calling `SIGNCONT`.
5. **Zombie (Z)**: Process has received the kill signal, and it's terminated.

▼ Operation on processes - Process creation

A process can create several new processes, via a create-process system call, during the course of execution.

The created process is called "parent process", and created process are called "children processes". Each of child processes can create its own processes forming a tree of processes.



Example of process a tree (Ubuntu 22.04)

To check the full tree in Linux run the `pstree` command.

When a new process is created there are two possible ways of execution:

1. The parent process continues to execute concurrently with its children.
2. The parent process waits for their children process to complete.

And there are two possible ways of managing address space:

1. The child process is a duplicate of its parent, hence the child process has the program and data.

2. The child process has a new program loaded into it.

▼ Operation on processes - Process termination

Standard process termination:

1. A process terminates when it finished its final statement and asks the OS to delete it by using the `exit()` system call.
2. After the `exit()` system call was called the process returns a status value (usually an integer) to its parent process.
3. All the resources of the process — including physical and virtual memory, open files, and I/O buffers — are freed (deallocated) by the OS.

Killing a process (explicitly terminating) by making a certain system call:

One process can terminate another process via an appropriate system call. This type of system calls only might be invoked only by parent processes or privileged processes (e.g., root process that is being executed in a kernel mode.).

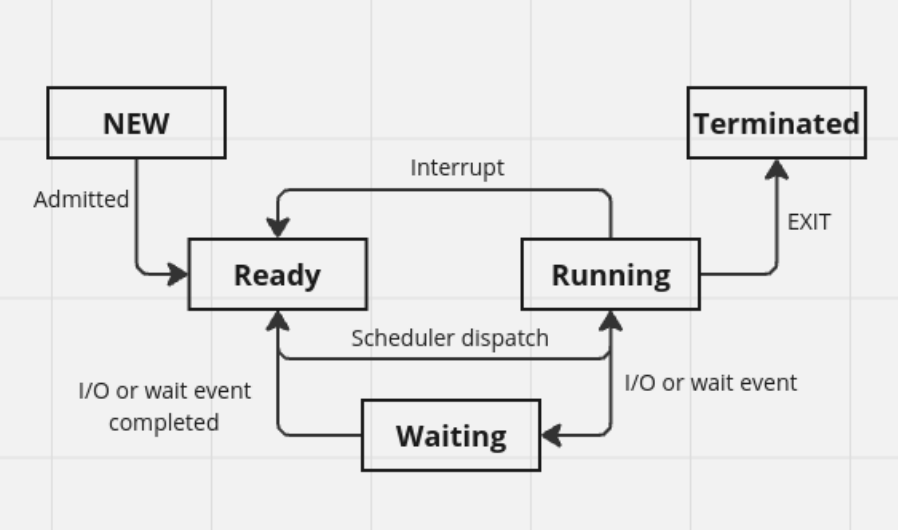
A parent may kill its child processes in such cases as:

- The child has exceeded its usage of some of the resources that has been allocated. (To determine whether it has been occurred the parent must have a mechanism to inspect the sate of its children.).
- The task assigned to the child is no longer required.
- The parent itself terminates. (only on some OSes).

▼ Process state during execution scheme

A process has several states in its life cycle:

- **New**: newly created process.
- **Ready**: process is waiting for CPU time.
- **Running**: the process is being executed right now.
- **Waiting**: the process is waiting for a signal or I/O to complete.
- **Terminated**: the process is being destroyed by the OS.



Process state during execution.

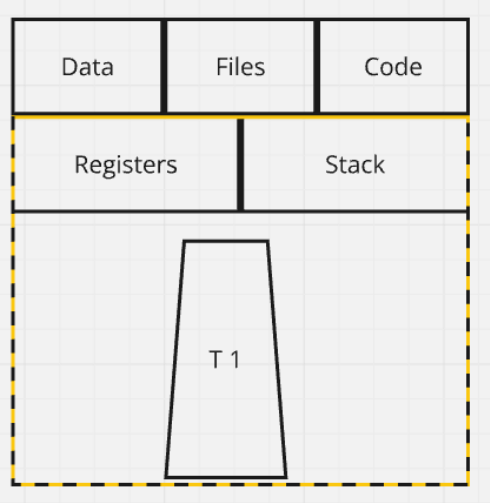
Threads

Thread in a basic unit of CPU utilization. Each thread has Thread Control Block, similar to a process. In Linux thread even use the same data structure as processes called `task_struct`. A thread shares its resources (code section, data section and other OS resources as open files and signals) with other thread within the same process.

A traditional (heavyweight) processes has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Single threaded process

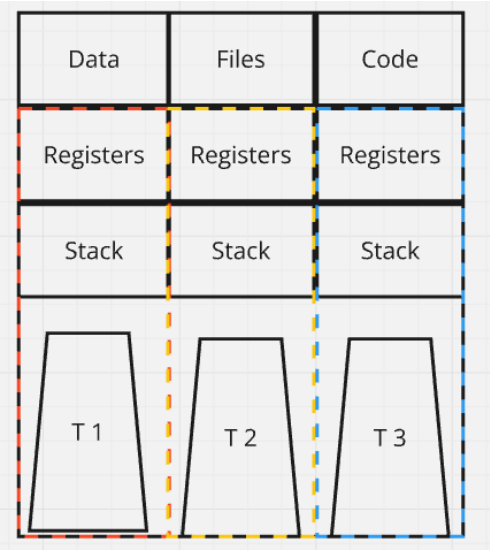
One process has one thread. A sing-threaded process run on only one CPU (CPU cores). No matter how many CPUs (CPU cores) are available.



Single threaded process.

▼ Multi-threaded process

Each of threads within one process have its own registers and stacks. Multiple tasks can be performed at a time with the help of these threads. Therefore, multi-threaded processes are more efficient than single threaded processes.



Multi threaded process.

▼ Benefits of multi-threaded processes

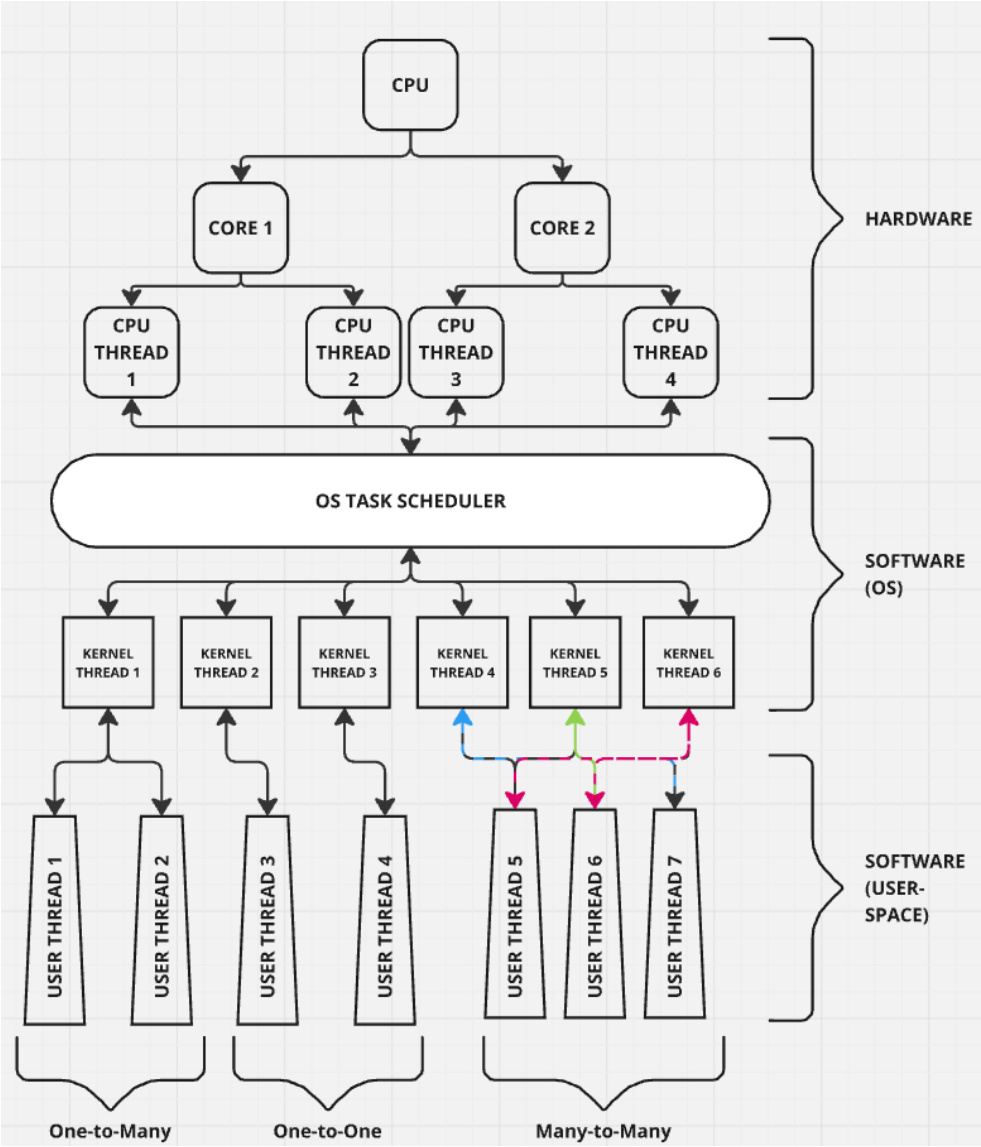
The benefits of multi-threaded processes can be broken into four major categories.

1. **Responsiveness:** multi-threading allows application to run even if one part of the application is blocked or performing a long running operation, thereby increasing responsiveness to the user.
2. **Resource sharing:** By default, threads share memory and the resources of the process they belong to. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space. So there is no need to think about data exchange between threads.
3. **Economy:** Allocating memory and resources for a process is costly. Threads are less expansive to create in comparison to processes, as creating a new thread the OS does not need to create another memory space address, and load data to it.
4. **Utilizing multi-processor architecture:** The benefits of multi-threading can be greatly increased in multi-processor architectures, where thread may be running in parallel on different processors (CPU cores). A single-threaded process can run only ob on CPU, no matter haw many are available. Multi-threading on a multi-CPU machine increases concurrency.

▼ Types of threads

This topic covers software threads, bot not CPU thread also knows a logical processors.

Software threads are created and managed by the Operating System or a program, and they are scheduled by the OS to run on CPU threads (logical processors).



Threads hierarchy in OSes.

In modern OSes the relationship model between user threads and kernel threads is primarily determined by the OS. The OS kernel implements a specific threading model, and software engineers cannot change it directly.

By the way, Linux uses one-to-one relationship model between kernel thread and user threads.

▼ User-threads

User threads supported above kernel threads and managed without kernel support (OS scheduler).

User threads are not directly visible to the OS. The OS only knows about the kernel threads. User threads execution flow is controlled by a programmer by utilizing third party libraries such as `POSIX`.

Only some languages libraries allow thread creating to form one(kernel)-to-many(user) threads (example `asyncio` from Python). These user thread (also, known as *green threads* or *coroutines)* are manages entirely in user space, and the OS is not aware of them. These threads are mapped to one kernel thread.

▼ Kernel-threads

Kernel threads are fully managed by the OS scheduler. These threads are visible to the OS, and the OS kernel decides when each kernel thread runs on the CPU. Each kernel thread is treated as a separate task, and the OS applies its scheduling algorithms (like CFS in Linux) to manage their execution.

▼ Multithreading models and Hyperthreading

Ultimately, there must be a relationship between user threads and kernel threads. Multithreading models are type of relationship that can be between user threads and kernel threads.

▼ Many-to-One

Characteristics:

- Many user threads are accessing one kernel thread.
- User thread management is done by thread management libraries is user space.

Downsides:

- The entire process can be blocked if a thread makes a blocking system call.
- Because only one user thread can access the kernel kernel at a time, multiple user thread are unable to run in true parallel on multiprocessors (different CPU thread / CPU logical processors).

▼ One-to-One

Characteristics:

- Maps each user thread to a kernel thread.
- Provides more concurrency than many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Allow true parallelism in multiprocessor systems.

Downsides:

- Creating a user thread requires creating a corresponding kernel thread.

- Because the overhead of creating kernel threads can be burden the performance of an application. Most implementations of this model restricts the number of thread supported by the system.

▼ Many-to-Many

Characteristics:

- Multiplexes many user-level threads to a small of equal number of kernel threads.
- The number of kernel threads may be specific either a particular application or a particular machine.
- Developers can create as many user thread as needed, and corresponding kernel threads can run in parallel on multiprocessor systems.
- When a thread performs a blocking system call, the OS can schedule another kernel thread to execute.

Downsides:

- Complex development.

▼ Hyperthreading (simultaneous multithreading SMT)

Simultaneous multithreading is a technique when CPU producers design processors (CPU cores) architecture the way that a single processor (CPU core) is able to execute two or more sets of instructions at the same time (in parallel).

The OS sees hyperthreaded processors (CPU thread / logical processors) as independent processors.

Hyperthreading system allow their processor cores resources to become multiple logical cores for performance.

It enables the processor (CPU core) to execute two thread, or set of instruction) at the same time. Since hyperthreading allows to stream to be executed at the same time (in parallel), it is almost like having two separate processors (CPU cores) working together.

▼ Run a program on a specific logical processor (CPU thread)

It is possible to run an executable code on a predefined CPU thread (logical processor).

Thread affinity allows to “pin” a specific software thread to run on a particular CPU core (logical processor / CPU thread). This is done by using system calls like `schd_setaffinity()` in Linux or tools like `taskset`.

```
taskset -c 0 ./my_program
```

▼ Control Blocks (PCB / TCB)

▼ Process Control Block (PCB)

Processes and threads are managed through a specialized data structures. These data structures help to track all the information about processes and threads that is required to manage their execution.

Process Control Block (PCB) is a data structure maintained by the OS for every process. It contains all the important information about a specific process, allowing the OS to manage and control the process during its life cycle (new, read, running, I/O waiting, terminated).

Information stored in PCB:

PCB field	Description
Process ID (PID):	Unique identifier of a process.
Process State:	The current process state (e.g., ready, ...).
Program Counter (PC):	Pointer to the next instruction to be executed.
Memory management information:	Information about the process's memory. E.g., pointers to it's page table, or segment table, memory regions (text, data, heap, stack).
Priority:	Priority of the process, which influences the CPU scheduler.
CPU scheduling information:	CPU scheduling queue information, nice value, CPU burst time, etc.
I/O status:	List of I/O devices allocated to the specific program.

▼ Process Control Block in Linux

PCB in Linux:

In **Linux**, the information for each process is stored in a kernel data structure called `task_struct`, which acts as the **PCB**. This structure contains all the necessary fields to manage processes.

Example: `task_struct` **(Linux PCB)**:

```
struct task_struct {
        // Process ID
    pid_t pid;
    // Process state (running, waiting, etc.)
    long state;
    // Pointer to memory information
    struct mm_struct *mm;
    // Scheduling information
    struct sched_entity se;
    // Pointer to open file descriptors
    struct files_struct *files;
```

```
    // Signal handling information
    struct signal_struct *signal;
    // ... (many more fields)
};
```

In this example:

- `pid`: The process ID.
- `state`: The current state of the process (e.g., `TASK_RUNNING`, `TASK_INTERRUPTIBLE`).
- `mm`: Points to the process's **memory map** (`mm_struct`), which stores information about the process's address space (code, data, stack).
- `files`: A list of open file descriptors for the process.
- `sched_entity`: Information related to CPU scheduling (priority, time slice, etc.).

▼ Thread Control Block (TCB)

A Thread Control Block (TCB) is similar to a PCB, but it contains thread-specific information. Threads are lightweight units of execution that share the same resources (like memory and open files) within a process. Each thread has its own TCB to manage its execution state.

Key Functions of TCB:

- The TCB allows the OS to **track** and **control** individual threads.
- It stores **thread-specific data**, such as the thread's program counter, stack pointer, and registers.
- It ensures that each thread can be **paused** and **resumed** independently of other threads within the same process.

TCB field	Description
Thread ID (PID):	Unique identifier for the thread (within the process).
Thread State:	The current state of the thread (e.g., running, waiting, terminated).
Program Counter (PC):	Points to the next instruction to be executed by the thread.
Stack Pointer:	Points to the thread's stack, which is unique for each thread.
CPU Registers:	The contents of CPU registers used by the thread.
Priority:	The priority of the thread for scheduling purposes.

TCB field	Description
Thread Context:	The current execution context of the thread (e.g., general-purpose registers).
Parent Process Info:	Information about the parent process to which the thread belongs.
Signal Mask:	Mask to determine which signals are blocked for this thread.

▼ Thread Control Block in Linux

In **Linux**, threads are treated similarly to processes, and `task_struct` also serves as the TCB. Threads in Linux are represented as **tasks** (lightweight processes), and each thread has its own `task_struct` entry.

- Linux assigns both processes and threads a **PID** (Process ID). However, threads are also given a TID (Thread ID), which differentiates them from other threads within the same process.

Example of TCB in `task_struct` (Linux):

Linux doesn't have a separate structure for TCBs because **threads are treated as tasks**. Therefore, both the **processes** and **threads** share the same `task_struct` structure in Linux.

For threads:

- TID (Thread ID) is stored in the `pid` field of the `task_struct`.
- Thread-specific information like the program counter, stack pointer, and CPU registers is stored just like in processes, but multiple `task_struct` entries will point to the same shared resources (e.g., memory, file descriptors).

▼ System calls `fork()`, `exec()`.

▼ System call `fork()`.

The `fork()` system call is used by a parent process in order to crate a separate, duplicate child process. The newly create child process is going to have a different process ID (PID) from its parent process.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
    fork(); // Create child process.
    fork(); // Create child process.
    fork(); // Create child process.
    printf("Hello FORKS! PID=%d\n", getpid());
```

```
    return 0;
}
```

▼ System call `exec()`.

When the `exec()` system call in invoked, the program specified in the parameters to the `exec()` system call will replace the entire process — including all threads. In this case the process will not change its ID (PID), because the `exec()` system call replaces the process (code, data, file descriptors, PC, registers, etc) but does not create a new process.

Example of `exec()` usage.

PS: `exec()` has many implementations and one of them if `execv()`, this implementation enables to pass vector of arguments.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv)
{
    printf("PID of exec1=%d\n", getpid());
    char *args[] = {"Hello", "exec", "!", NULL};
    execv("./compiled/exec2", args);
    printf("Back to exec1\n");
    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
    printf("We are now in exec2.c PID=%d\n", getpid());
    return 0;
}
```

Note, the semantics of `fork()` and `exec()` system call change in multithreading programs.

▼ Semantics of `fork()` system call in multithreaded programs

In single threaded programs, the `fork()` system call creates a child process that a full duplicate of the parent process. It inherits all of the parent's memory, file descriptors, and other resources. Essentially, the child process is an exact copy of a parent process, but they both continue executing independently after the `fork()`.

However, in **multithreaded program**, semantics change because there are multiple threads within a parent process, and the behavior of `fork()` need an account for that.

When `fork()` is invoked in multithreaded process, only the calling thread is duplicated in child process. The other threads in the parent process are not copied to the child process. This has several consequences:

- Parent Process: Continues execution as usual with all of its threads.
- Child Process: Only has the thread that invoked `fork()`; the other threads from the parent do not exist in the child.

This behavior might cause issues:

- The child process might not be in a consistent state if parent threads were manipulating shared data. Since, only one on parent threads are duplicated, it is possible that the child process could inherit data that is partially modified or in an inconsistent state due to other thread that were in the middle of their operation.
- Since other threads are not copied, any resources that they were holding (like file descriptors, locks, etc.) are not duplicated in the child process, which can lead to deadlocks or undefined behavior if the child process tries to access them.

▼ Semantics of `exec()` system call in multithreaded programs

In a **single-threaded** process (program), `exec()` simply replaces the process image with a new program, and execution continues in the new program.

In multithreaded process (program), when one of the thread calls `exec()`, following occurs:

1. All threads are terminated: Before the new program (executable) is loaded, the kernel terminates all threads except the one calling `exec()`. This is because the new program is single-threaded initially (it starts with only one main thread).
2. The calling thread's memory space and other resources are replaces with the new program's code and data.
3. After the `exec()` call, the new program executes as if it were a single-threaded process.

This behavior might cause issues:

- If one of a multithreaded program's thread calls `exec()` all threads will be terminated (also, known as thread cancellation), and a new program will start in a single threaded process.
- Any other resources held by other threads (looks, open file descriptors, network connections) may not be properly released if these threads are terminated abruptly by the `exec()` call. This could leak to resource leaks or inconsistent system states.

▼ Dealing with `fork()` and `exec()` in multithreaded programs

Best practice: Use `fork()` followed by `exec()` carefully in multithreaded programs. A common pattern to cal `fork()` and right after `exec()` in the child process. In this case, any

inconsistent state in child process caused by missing thread is irrelevant because `exec()` will replace the entire process anyway.

There are alternatives like `posix_spawn()`. Which is designed for spawning new processes and can be safer alternative to `fork()` and `exec()` in multithreaded environments, as it handles some of complexity internally.

▼ Thread cancellation

Threads cancellation is the process when threads are being terminated. The thread that is going to be cancelled (terminated) often referred as a target thread. There are a few types of thread cancellation:

1. Asynchronous cancellation: One thread immediately terminated the target thread.
2. Deferred (graceful) cancellation: The target thread periodically checks (special flag) whether it should terminate, allowing an opportunity to terminate itself in orderly fashion.

Issues to consider cancelling a thread:

- When a resources have been allocated to a cancelled thread: When a target thread is terminated the OS will free system wide resources but might fail free all the resources (e.g., opened sockets).
- A target thread is cancelled while updating data it's sharing with other thread:

Inter-process communication (IPC)

▼ Introduction

A concurrently executing processes in the OS can either:

- **Independent** processes — They cannot affect or be affected by the other processes executing in the system.
- **Cooperating** processes — They can affect or be affected be the other processes executing in the system.

Any process that shares data with other processes is a cooperative process.

▼ Inter-process communication (IPC)

Definition

The inter-process communication mechanism is used when two or more processes need to share data or send messages to each other.

Inter process communication allows to:

- Send and receive messages.
- Share data and synchronize execution.

Why needed ?

Since processes have separate virtual memory address spaces they cannot access to the shared data directly like threads. However, sometimes processes need to cooperate and exchange data.

Examples:

1. A web process (process 1) communicates with a db server (process 2) to store or retrieve data.
2. A parent process (e.g. shell) spawns a child processes and needs to synchronize them.

When to use ?

- Information sharing between processes.
- Computational speed up by braking a task into smaller pieces, executing pieces in parallel and joining the result.
- Modularity — designing a system with different modules working together.

▼ Choosing the right IPC mechanism

- **Pipes**: Great for simple **parent-child process communication** or **command-line piping**.
- **Message Queues**: Ideal for **asynchronous message passing** and more structured communication.
- **Shared Memory**: The **fastest IPC**, suitable for sharing **large amounts of data** between processes, but requires synchronization (e.g., semaphores).
- **Semaphores**: Used for **synchronization** between processes, particularly when accessing shared resources.
- **Signals**: Best for sending **asynchronous notifications** between processes, like termination requests.
- **Sockets**: Essential for **network communication** between processes, either locally or across a network.

▼ IPC systems

▼ 1. Shared memory system

Shared memory allows direct access to the same memory address space by multiple processes.

Typically, shared memory region resides in the process memory address space which initializes creation of the shared memory address space. Other processes that wish to communicate using this shared-memory segment must attach to their address space.

(Normally, the OS does not allow a process to attach other process memory). Shared memory requires that two or more process are agreed to remove this restriction.

Buffer sizing:

1. Unbounded buffer: space is dynamically reallocated.
2. Bounded buffer: fixed size.

▼ 2. Message passing system

Message passing systems provide a mechanism to allow processes to communicate and to synchronize actions without sharing the same address space. It is particularly useful in in distributed environments where communicating process can reside on different computers connected by a network.

A message passing facility provides two operations: send and receive messages. Messages sent by a process can be either fixed or dynamic size. Fixed size messages are easier in development but sometimes not convenient in user and vise versa for the dynamic size messages.

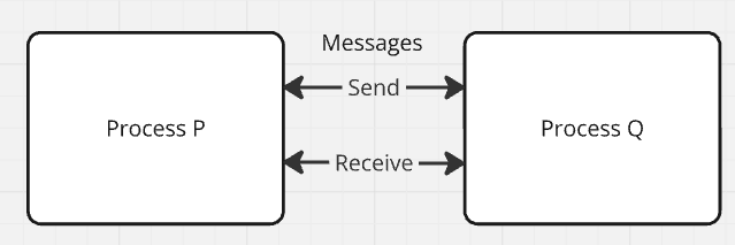
If a process Q want to communicate with process P, they must send messages to and receive from each other. A communication link must exist between them.

This link can be implemented in a variety of

- Direct or indirect communication.
- Synchronous or asynchronous communication.
- Automatic of explicit buffering.

▼ 2.1 Message passing system direct communication

Direct communication: Each process that want to communicate with other processes must explicitly tell the name of the recipient process.



IPC Direct communication paradigm scheme.

```
// Synchronous
send(P, "Message")
revieve(Q, "Message")

// Asynchronous
```

```
send(P, "Message")
revieve(id, "Message")
```

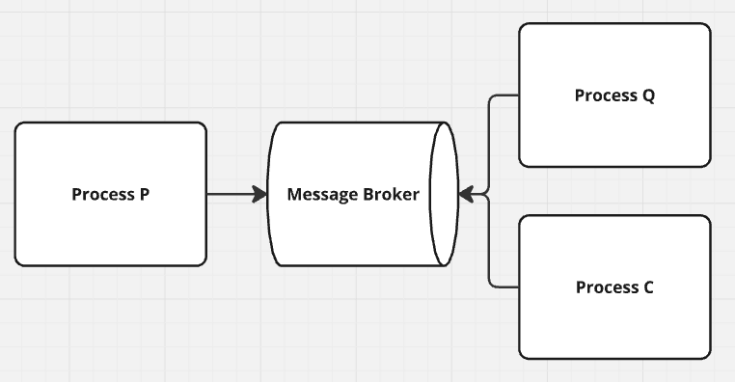
In this paradigm:

- A link is established automatically between between two processes.
- Between each pair of processes there is only one link.
- Symmetry in addressing. The sender and the receiver must name each other to communicate.
- In asynchronous communication the receiver receives messages from any process.

Main disadvantage of either synchronous or asynchronous direct communication is high coupling (lack of modularity).

▼ 2.2 Message passing system indirect communication

Indirect process communication: The messages are send to end received from an intermediary (e.g., message broker, socket). An intermediary can be viewed abstractly as an object into which messages can be places by processes and from which messages can be removed. Two processes can communicate only if the process es have share intermediary.



IPC indirect communication paradigm scheme (Producer-Consumer model).

```
send(intermediary, "Message")
revieve(intermediary, "Message")
```

In this paradigm:

- A link is established between a pair of processes only if they are both have the same intermediary.
- A link may be associated with two or more processes.

- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one intermediary.

Receiving messages policy is dependent on a concrete implementation. Following strategies can be chosen:

- Allow at most one process at a time to execute the `receive()` command.
- Implementing collision solving algorithm in the intermediary e.g., Round Robbin, etc.

An intermediary can be owned either by a process or the OS.

▼ 2.3 Sync and async communication

In inter-process communication (IPC), synchronous and asynchronous communication are two primary models for processes to interact with each other.

Synchronous communication: Best for real-time interactions, where immediate responses are crucial, and the sender process can afford to wait.

- **Blocking:** The sender process waits for the receiver process to acknowledge the message before proceeding.
- **Real-time:** Communication happens immediately, and the sender is blocked until the receiver processes the message.
- **Examples:** Remote Procedure Calls (RPC), Pipes, Sockets (with blocking operations)

Asynchronous communication: Best for decoupled systems, where processes can work independently, and there's no need for immediate responses. This approach is ideal for high-throughput, distributed systems.

- **Non-blocking:** The sender process doesn't wait for the receiver to acknowledge the message. It can continue with other tasks.
- **Message-based:** Communication relies on message queues or message brokers.
- **Examples:**
  - Message Queues.
  - Sockets (with non-blocking operations).
  - Signals.

▼ 3. Clint-Server Systems

Sockets: Socket is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets — one for each process. A socket is identified by an IP address concatenated with a port number.

The server waits for an incoming client request by listening a specified port. Once, a request is received, the server accepts a connection from the client to complete the connection.

Servers implementing specific services e.g., HTTP server, SSH server, etc..

Connection establishment:

1. When a client process wants to establish a connection, the OS assigns a port to the process (*free port numbers > 1024 > well known port numbers)* and the IP address is concatenated usually automatically.
2. The OS makes a network request to the destination socket.
3. The server receives the request and the process working on the socket is responsible for processing the connection.

If the same client process wants to connect other sockets the OS assigns different socket to the same process, hence ope process can have multiple sockets.

▼ Remote procedure (call) systems

Remote procedure system is a form of inter-process communication. in that different to IPC processes have different address spaces: if on the same host machine, they have distinct virtual address spaces, even though the physical address space is the same; while if they are on different hosts, the physical address space is different.

▼ Remote Procedure Calls (RPC)

Remote Procedure Calls (RPC) is a protocol that one program can use to request a service from a program located in another computer or network without having to understand the network's details. Because RPC is dealing with processes that are executing on separate systems, the message passing communication scheme must be used.

▼ RPC workflow

Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function.

The function is executed as required, and any output is send back to the requester in a separate message.

▼ Semantics of remote calls

The semantics of RPC allow a client to invoke a procedure on a remote host as it would invoke a procedure locally.

Steps:

1. The RPC system encapsulates the details that allow communication to take place b providing a stub on the client side. Typically, a separate stub exists for each separate remote procedure.
2. When a client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. The stub locates the port

on the remote system and marshes the parameters. Each service is identifies by a separate port number.

3. Parameters marshaling involves packaging the parameters into a form that can be transferred over a network.
4. The stub transmits a message to the server using message passing system.
5. A similar stub on the remote server receives the message and invokes the procedure on the server.
6. If necessary, return a values are passed back to the client using the same technique.

▼ Common issues using RPC

Using Remote Procedure Call might go together with some common issues.

List of issues:

1. **Issue:** Different in Data representation between client and server machines.

   Example: representation of 32-bit integer. Some machines (known as big-endian) use the high memory address to store the most significant byte, while the other systems (known as little-endian) store the least significant byte in the high memory address.

   Solution: RPC systems define a machine-independent representation on data. One such data representation is known as external data representation (XDR). On the client side, parameter marshaling involves converting the machine dependent data into XDR before they are send to the server. On the server side, the XDR data are unmarshalled and converted to machine-dependent representation for the server.

2. **Issue:** Whereas local procedure calls might fail under extreme circumstances, RPCs can fail, or be duplicated and executed more then once, as a result of common network mistakes.

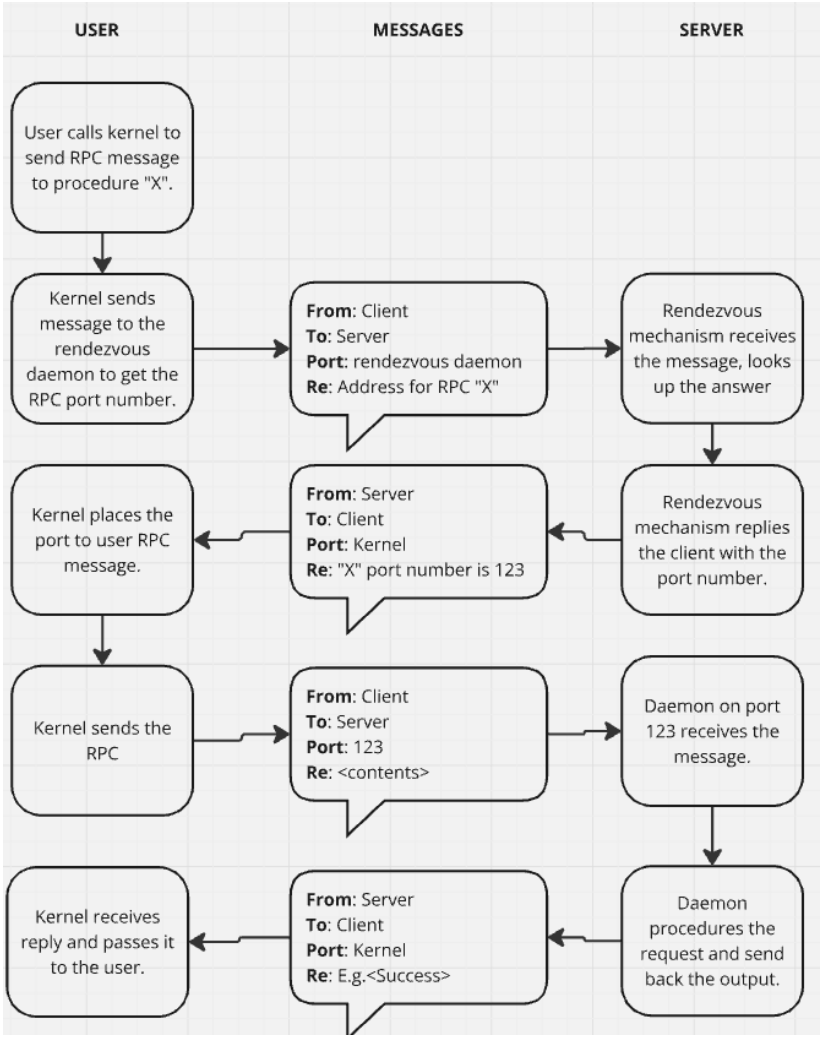
   Solution: The OS must ensure that RPCs are executed exactly once, rather than at most once. Most RPC have exactly one functionality. Implementing the acknowledgement system.

3. **Issue:** RPC client does not know the binding of the RPC port number on the server. Hence, how the client knows on the port to call in order to call an appropriate procedure call ?

   Solution 1: The binding information is predetermined, in the form of fixed port numbers during program (RPC stub) compilation. Therefore, the server cannot change port numbers during the runtime. (This solution is less flexible).

   Solution 2: Dynamically port number binding using rendezvous mechanism. Typically, OSes provide rendezvous mechanism (also called a matchmaker) daemon on a fixed RPC port. A client sends a message containing the RPC name to the rendezvous daemon requesting the port number of the RPC it needs to execute, and the RPC calls can be send to that port until the process terminates (or the server crashes). Even, if

the port number changes the rendezvous daemon will always help the client to find the right port number. (This solution is more flexible).



RPC workflow with rendezvous mechanism.

▼ Types (with C examples)

There are several IPC mechanisms in **Linux** (and most other Unix-based OSes). They vary in how they work, their complexity,

▼ Pipes

Pipes: Pipes are one of the simplest and most commonly used IPC mechanisms. They provide a unidirectional communication channel between processes, where one process writes data to the pipe and another reads from it.

Key characteristics:

1. **Anonymous pipes** are created within the same process or between a parent and child process.

2. They are commonly used in **command-line operations** to pass data between processes. Example: `ls | grep "txt"` the `|` means a pipe between two processes "ls" and "grep".

Example in C:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);  // Create a pipe

    if (fork() == 0) {
        // Child process
        close(fd[0]);  // Close reading end
        write(fd[1], "Hello from child\n", 17);
        close(fd[1]);
    } else {
        // Parent process
        close(fd[1]);  // Close writing end
        char buffer[100];
        read(fd[0], buffer, sizeof(buffer));
        printf("Parent received: %s", buffer);
        close(fd[0]);
    }
    return 0;
}
```

- A **pipe** is created between the parent and child processes.
- The child writes a message to the pipe, and the parent reads it.

Named pipes:

- Named pipes (FIFOs) are similar to anonymous pipes but exist as files in the filesystem, making them useful for communication between unrelated processes.
- Named pipes allow communication between processes that do not share a parent-child relationship. `mkfifo mypipe  # Create a named pipe`.

▼ Message Queues

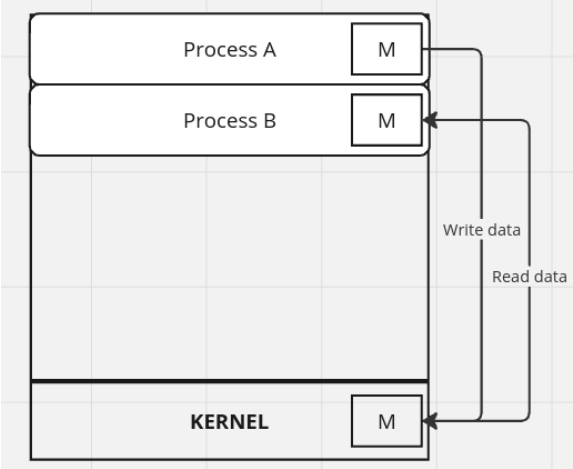
Message Queues: allow processes to exchange messages in a queue. Unlike pipes, message queues support:

1. Asynchronous communication (messages can be sent and read at different times).

2. Multiple messages with different types.

Key characteristics:

1. Messages are **queued** in a **first-in, first-out (FIFO)** manner.
2. Processes can send and receive messages **without blocking**.



Usage of Message Queue scheme

Example in C:

To use message queues in Linux, the **System V IPC** API provides system calls like `msgget()`, `msgsnd()`, and `msgrcv()`.

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

struct message {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key = ftok("queuefile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct message msg;
    msg.msg_type = 1;
    sprintf(msg.msg_text, "Hello from message queue");

    msgsnd(msgid, &msg, sizeof(msg), 0);
```

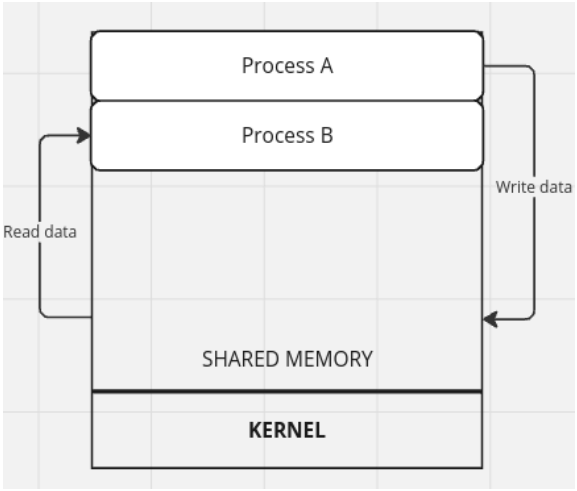
```
    printf("Message sent: %s\n", msg.msg_text);

    msgrcv(msgid, &msg, sizeof(msg), 1, 0);
    printf("Message received: %s\n", msg.msg_text);

        // Destroy message queue
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}
```

▼ Shared Memory

Shared Memory: is the fastest form of IPC because it allows direct access to the same memory space by multiple processes. Instead of passing data through pipes or messages, processes can read and write to a shared memory segment.

Key characteristics:

1. **F**astest IPC because no data needs to be copied between processes.
2. Ideal for large amounts of data..
3. Requires synchronization (e.g., using semaphores) to avoid race conditions.



Usage of shared memory scheme

Example in C:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>
```

```
int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    char *str = (char*) shmat(shmid, (void*)0, 0);

    strcpy(str, "Hello from shared memory");

    printf("Data written: %s\n", str);

    // Detach from shared memory
    shmdt(str);

    // Destroy the shared memory
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

In this example:

1. Shared memory is created using `shmget()`, and a process attaches to it with `shmat()`.
2. Processes can directly read from and write to the shared memory segment.

▼ Semaphores

Semaphores: are used for synchronization between processes. They are not designed to exchange data, but to coordinate access to shared resources and avoid race conditions in shared memory.

Key characteristics:

1. Semaphores maintain a counter that tracks access to shared resources.
2. Processes can wait for a semaphore (decrease the counter) or signal it (increase the counter).

Example in C:

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

int main() {
    key_t key = ftok("semfile", 65);
    int semid = semget(key, 1, 0666 | IPC_CREAT);
```

```
    // Initialize semaphore value to 1
    semctl(semid, 0, SETVAL, 1);

        // Wait operation
    struct sembuf p = {0, -1, 0};

    // Signal operation
    struct sembuf v = {0, 1, 0};

    semop(semid, &p, 1);  // Wait (lock)
    printf("Critical Section Start\n");
    sleep(2);
    printf("Critical Section End\n");

    // Signal (unlock)
    semop(semid, &v, 1);

        // Remove semaphore
    semctl(semid, 0, IPC_RMID);
    return 0;
}
```

In this example:

- A semaphore is used to control access to a critical section, ensuring that only one process enters the section at a time.

▼ Signals

Signals: are a way for processes to send notifications to each other or to themselves. A signal is a simple asynchronous notification that tells a process to perform some action (e.g., terminate, pause, or handle a custom event).

Key characteristics:

1. Signals are used to notify a process about asynchronous events (e.g., division by zero, termination requests).
2. A process can catch and handle signals using signal handlers.

Common Signals in Linux:

- `SIGKILL`: Terminates a process immediately.
- `SIGTERM`: Requests a process to terminate gracefully.
- `SIGINT`: Sent when a user interrupts the process (e.g., pressing `Ctrl+C`).

Example in C:

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

void handle_signal(int sig) {
    printf("Caught signal %d\n", sig);
}

int main() {
        // Register signal handler for SIGINT
    signal(SIGINT, handle_signal);

    while (1) {
        printf("Running... Press Ctrl+C to stop.\n");
        sleep(1);
    }
    return 0;
}
```

In this example:

- The process registers a signal handler for the SIGINT signal (triggered by `Ctrl+C`), which allows the process to handle the signal gracefully.

▼ Sockets

Sockets: are a mechanism for network-based IPC. They enable communication between processes on the same machine or different machines over a network (using protocols like TCP/IP or UDP).

Key characteristics:

1. Sockets are bidirectional and can be used for both local IPC and network communication.
2. Commonly used in client-server models.

Example in C:

A typical client-server program using sockets involves the following steps:

1. Server creates a socket, binds to a port, listens for connections, and accepts a client.
2. Client connects to the server and exchanges data.

```c
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in address = {AF_INET, htons(8080), INADDR_ANY};
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));

    listen(server_fd, 3);

    int client_fd = accept(server_fd, NULL, NULL);
    char *message = "Hello from server";
    send(client_fd, message, strlen(message), 0);

    close(client_fd);
    close(server_fd);
    return 0;
}
```

Multiprogramming and Multitasking (time sharing)

▼ Introduction

▼ Multitasking in single processor systems

In a single-processor system only one processor can execute a program (set of instructions) at a time. When a program is waiting for some I/O operation to be completed the CPU is blocked by the program, and does not complete any useful job, just being idle.

Multitasking allows to distribute valuable CPU time among all tasks (processes) by utilizing some data structures and CPU time planning algorithms for each task individually.

▼ Multitasking in multiprocessor systems

In multiprocessor system the CPU scheduler plays crucial role by distributing task no only among a singe CPU, but among all CPU cores (processors).

Note: OS sees every logical processor (CPU thread), as a single independent processor.

▼ Main idea

The main Idea behind multitasking is the CPU time must be utilize properly between all tasks.

▼ Components of the OS scheduling process

1. Task (Process / Kernel Thread): A single unit of execution.
    a. Process Control Block:
        i. PID.
        ii. Program Counter.

iii. Machine Code.

iv. Data.

v. etc...

2. Task schedulers:
    a. Long-term scheduler.
    b. Medium-term scheduler.
    c. Short-term scheduler.
    d. Dispatcher.
3. CPU processor (logical processor / CPU thread).

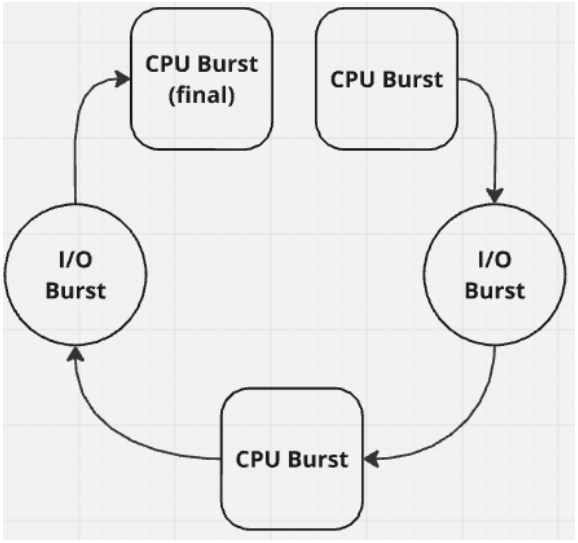
▼ CPU and I/O burst cycles

A process execution consists of two states:

- CPU Execution state: The machine instructions are being executes at a time.
- I/O wait state: The process is waiting for some data to be written or received in order to continue execution. It this time the program instruction are not being executes.

Process execution cycle.

In a final CPU burst there is a system request to terminate execution of the process.



Process execution cycle (CPU burst, I/O burst).

▼ Components of the OS Scheduler

▼ 1. Long-term scheduler (Job Scheduler)

- **Role**: The long-term scheduler is responsible for deciding which processes (jobs) to admit into system for execution. When a program starts (like launching application),

the long-term scheduler decides whether the process should be loaded into memory or kept waiting (especially in systems with limited resources).

- **When:** It operates infrequently because process creation is relatively rare compared to other scheduling events.
- **Tasks:**
  - Decide which processes are loaded into memory for execution.
  - Creates the Process Control Block (PCB), which holds the essential information about the process (like PID, Program Counter, State, etc.).
- **Example:** "User start a new program, the long-term scheduler decides to create a PCB for this program and admits it into the system for execution".

▼ 2. Medium-term scheduler (Swapping Manager)

- **Role**: This scheduler is responsible for memory management and sometimes swapping processes in and out of the main memory. If a system is running low on memory, the medium-term scheduler will decide to suspend or swap out a process to free up space, and then swap it back later when memory becomes available.
- **When**: Operates occasionally, when memory needs to be freed up.
- **Tasks**:
  - Temporary removes (swaps out) processes from memory is the file system is overloaded, and brings them back later.
  - Manages processes that are waiting (int the blocked state) until they are ready to run again.

▼ 3. Short-term scheduler (CPU scheduler)

- **Role**: The sort-term scheduler is responsible for deciding which process gets the CPU next. It works with ready queue, a queue of processes that are ready to run (runnable) (in ready state) and selects one based on scheduling algorithm (e.g. CFS, RR, Priority Scheduling, etc.).
- **When**: Operates frequently, typically every few milliseconds (on every context switch).

  There are four circumstances of context switching. When a process switches:

  1. From the running state to the waiting state.
  2. From the running state to the ready state (e.g, when an interrupt occurs).
  3. From waiting the waiting state to ready state (e.g, I/O completion).
  4. When a process terminates.
- **Tasks**:
  - Chooses the next process from the ready queue and gives it the CPU.

    - Works at the level of kernel threads.
- **Relation to PCB**: The sort-term scheduler uses the Process Control Block (PCB) to access the state (like registers, program counter, etc) of the next process to run.

▼ 4. Context switcher (Dispatcher)

- **Role**: The dispatcher is the components responsible for performing the context switch. After the short-term scheduler selects the next process to run, the dispatcher takes over and loads the process's state (PCB) into the CPU registers. It perform the actual switching of execution form one process to another.
- **When**: Every time the process switch happens. The time it take to switch the context is known as dispatch latency.
- **Tasks**:
    - Loads the next process's context (registers, program counter, etc.) into the CPU.
    - Performs the context switching so selected process could run on CPU processor (hardware).

▼ 5. CPU (Hardware)

- **Role**: The actual physical CPU executes the machine instructions of the process. The CPU has multiple cores and each core has a logical processor (CPU thread), which execute the machine instructions.
- **When**: Constantly executing instructions.
- **Tasks**:
    - Execute the instructions of the currently running process.

▼ Task scheduling step-by-step

When a new application is being launched the OS make following steps to launch and execute the application (program):

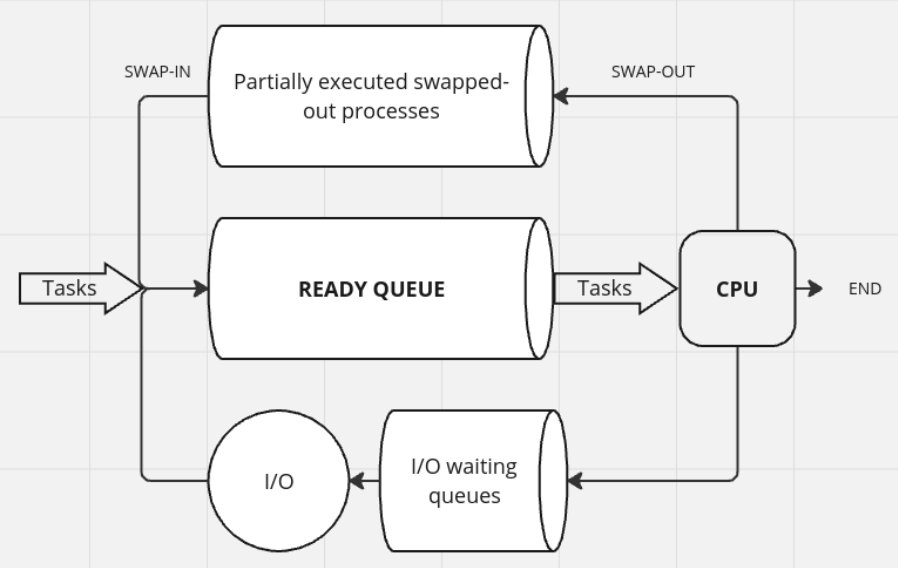
1. **Long-term Scheduler:**
    - **What it does**: when a new program starts, the long-term scheduler decides to admit the process into the system, initializes the Process Control Block (PCB), and allocates resources (like memory).
    - **Result:** The process is now in the ready queue, waiting to be scheduler by the short-term scheduler.
2. **Medium-term scheduler:**
    - **What it does:** If there is a memory shortage, the medium-term scheduler swaps out some processes and may bring them back when memory is available. This component manages processes that are swapped or blocked.

- **Result:** It ensures efficient use of memory by managing process suspension and resumption.

3. **Short-term scheduler (CPU scheduler)**:
   - **What it does:** selects the next process (kernel thread) to execute from the ready queue, using the system scheduling algorithm.
   - **Result:** The selected process's PCB is passed to the dispatcher to perform the context switch.
4. **Context switching (Dispatcher):**
   - **What it does:** The dispatcher loads the process's state form the PCB into the CPU's register and switches the CPU processor to execute the selected process.
   - **Result:** The selected process begins to execute on a CPU thread (logical processor).
5. **Execution on CPU thread (logical processor):**
   - **What it does:** Execute the machine instructions of the process selected by the short-term scheduler and switched by the dispatcher.
   - **Result:** The process runs on physical CPU, executing it's code until its preempted, completed or blocked.

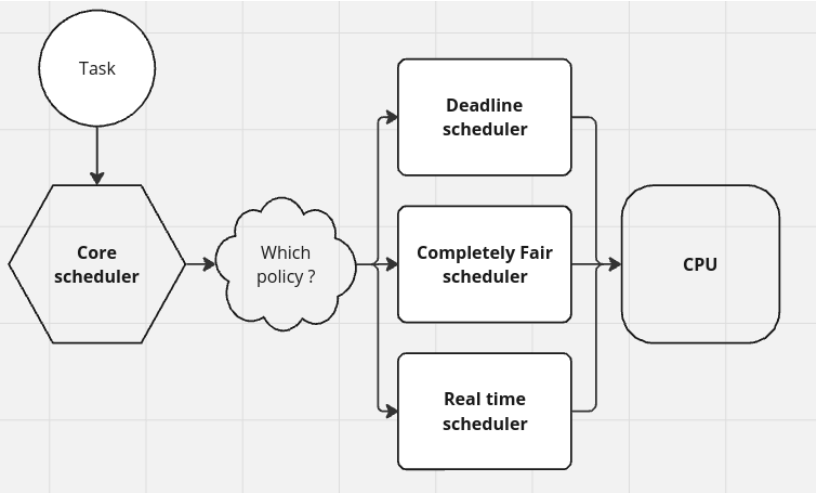
▼ Process flow scheme

When a running task stops being executed (e.g., waiting for I/O operation to be done) the scheduler swaps the task to a task that is ready to execute, and putting the I/O waiting task to another data structure (e.g., queue).



Process flow scheme

▼ Workflow



Task scheduling workflow.

▼ CPU scheduling

Note: Do NOT confuse CPU scheduling and task scheduling in Operating Systems. Despite of being related, these are different components with separate purposes in an Operating Systems.

▼ Introduction

- **Task Scheduler** (often referred to simply as the **scheduler**): This is a higher-level concept in the operating system, and its role is to decide **which process or task** should be executed by the CPU at any given time. A task can represent a **process** or a **thread** (depending on the operating system).
- **CPU Scheduler**: This is a lower-level part of the task scheduling system. Once the task scheduler selects a process or thread to run, the **CPU scheduler** is responsible for managing the **allocation of the CPU** (or CPU cores, if we're in a multicore system) to those tasks.

The term "CPU scheduling" is often used to refer to the entire scheduling process of assigning **CPU resources** to different tasks.

In many contexts, both **task scheduling** and **CPU scheduling** are considered part of the **same system**—but the CPU scheduler is more specifically concerned with how tasks are mapped to the CPU.

▼ Definition

CPU scheduling is the process by which the OS decides which tasks (processes or threads) will run on the CPU and for how long. In systems with multiple cores, it also involves deciding which CPU core will handle each task.

Since **CPU time** is a finite resource, the operating system needs to carefully manage how it is distributed among various tasks. This process of assigning the CPU to different tasks is called **scheduling**.

▼ Preemptive (forced) multitasking

▼ Preemptive multitasking in Task scheduling

Preemptive scheduling takes place when CPU time can be taken from a process. Usually, when CPU scheduler follows a preemptive algorithm.

▼ Cooperative (voluntary) multitasking

▼ Cooperative multitasking in Task scheduling

Cooperative scheduling happens in task scheduling takes place under two circumstances:

1. When a process switches from running state to the waiting state.
2. When a process in being terminated.

The CPU time cannot be taken from a process until the process goes to the waiting state or terminates. Otherwise, its preemptive.

▼ Context switching (dispatching)

Context switching is the process of saving the state of a currently running process or thread and restoring the state of another process or thread. The OS scheduler decides when to **context switch** to give each process or thread its turn to run on the CPU.

The content switching is executed by the dispatcher (a module in the task scheduler).

▼ Steps in context switching

Interrupts cause the OS to change a CPU from its current task and run kernel routine.

1. **Save Context**: The state of the currently running process or thread is saved in its **PCB/TCB** (including the program counter, CPU registers, and other execution context).
2. **Restore Context**: The OS restores the context of the next process or thread by loading its saved state from its **PCB/TCB**.
3. **Switch Execution**: The CPU starts executing the next process or thread.

The **PCB/TCB** plays a crucial role in storing the **context** during a switch so that the OS can seamlessly resume execution at the point where the process/thread was paused.

Notes:

- Context switching is a pure overhead because the system does no useful work during switching.
- The speed of context switching vary from machine to machine, depending on the memory speed, number of registers that must be copied, and the existence of special instruction.
- Typical speed is a few milliseconds.

▼ Scheduling Criteria

Scheduling criteria are metrics that an OS considers when selecting which task to tun on the CPU processor, and for how long. These criteria help to design and estimate scheduling algorithms efficiency.

▼ List of criteria:

1. **CPU utilization:**
    - **What it means:** it measures the percentage of time the CPU is actually busy doing work (i.e., to idle).
    - **Goal**: The scheduling algorithm should try to keep the CPU as **busy as possible**, without leaving it idle when there are tasks to be processed.
    - **Ideal outcome**: Maximize CPU utilization to approach **100%** (though 100% is unrealistic due to context switches, I/O waits, etc.).
2. **Throughput:**
    - **What it means**: It is the number of processes completed per unit of time (e.g., how many processes finish in one second).
    - **Goal:** The scheduling algorithm should maximize throughput by making sure a high number of processes finish in a given time period.
    - **Ideal outcome:** More processes completed in less time (i.e., higher throughput).
3. **Turnaround time:**
    - **What it means**: The total time taken from the moment a process enters the system until it is completed (includes all waiting, execution, and I/O times).
    - **Formula**: $T_{turnaround} = T_{completion} - T_{arival}$.
    - **Goal:** Minimize turnaround time so processes finish faster.
    - **Ideal outcome:** A scheduling algorithm that minimizes the overall time a process spends in the system (quick response and completion).
4. **Waiting time:**
    - **What it means**: The total time a process spends in the ready queue, waiting to be scheduled to run on the CPU (does not include execution time or I/O wait time).
    - **Formula:** $T_{waiting} = T_{turnaround} - T_{burst}$
    - **Goal:** The scheduler should minimize the waiting time of each process in the ready queue.
    - **Ideal outcome:** Shorter waiting times mean less idle time for processes waiting to execute.

5. **Response time:**
    - **What it means**: The amount of time it takes from when a request (process) is submitted until the first response is produced (i.e., until the process starts executing, not when it completes).
    - **Formula:** $T_{response} = T_{firstrun} - T_{arrival}$.
    - **Goal:** For interactive systems (like user applications or GUIs), it's important to minimize response time to make the system feel fast and responsive
    - **Ideal outcome:** The scheduler should ensure processes start executing quickly after they're submitted, even if they don’t finish right away.
6. **Fairness:**
    - **What it means**: Ensuring that all processes get an equal or fair share of the CPU time and resources, without starvation (where a process waits forever while others get CPU time).
    - **Goal:** Every process should get some share of CPU time, and high-priority or short tasks shouldn’t monopolize the CPU.
    - **Ideal outcome:** Fair allocation of CPU time to prevent any one process from starving or being treated unfairly.
7. **Context switching overhead:**
    - **What it means**: A context switch happens when the OS switches the CPU from running one process to another. This involves saving the state of the current process and loading the state of the next one.
    - **Goal:** Minimize context switch overhead, since context switching takes CPU time without doing productive work.
    - **Ideal outcome:** Efficient scheduling algorithms should minimize the number of context switches to reduce unnecessary CPU time overhead.

▼ Balancing criteria

In real-world scheduling, some criteria can conflict with others. For example:

- Maximizing throughput may increase waiting time for some processes.
- Minimizing response time (for interactive tasks) could decrease overall CPU utilization.
- Fairness might result in longer turnaround times for some processes.

Because of these trade-offs, different scheduling algorithms prioritize **different criteria** depending on the system's needs:

- Batch systems (where non-interactive processes run) might prioritize throughput and CPU utilization.

- Interactive systems (like desktop OS or web servers) might prioritize response time and fairness.

▼ Task scheduling algorithms

Task scheduling algorithm are responsible to tackle following questions:

- For how long a process should be executed.
- How to determine when a process needs to be swapped.
- How to determine after what time a process that had been swapped need to continue its execution.
- etc...

And the efficiency of each algorithm is measured by the scheduling criteria.

▼ Formulas:

1. Task completion time and Task completion timeline:

$$T = \{t_n \mid \sum_{p=1}^{p_n}(d + b)\}$$

   - P = process number.
   - d = dispatch latency.
   - b = burst time

Preemptive algorithm:

1. **Waiting time** = Total waiting time - No of ms. process had completed - Arrival time.
2. **Waiting average** = sum(processes waiting time) / number of processes.

Non preemptive algorithm:

1. **Turnaround time** = Finish execution time - arrival time.
2. **Waiting time =** turnaround time - burst time.

Algorithms:

▼ FCFS (First-Come, First-Served) algorithm

Semantics: The process that request the CPU first gets the CPU time until it's terminated or requested I/O. Meanwhile, other processes are waiting for execution in a ready queue. Consequently, the algorithm is cooperative (non preemptive).

Implementation details:

- **Data Structure:** a simple queue (FIFO).

Scheduling criteria:

- CPU utilization:
- Throughput:
- Turnout time:
- **Waiting time:** depending on the order and burst time.
- Response time: low (average).

Issues:

1. FCFS is not convenient for time sharing systems, where each task needs to be executed without waiting for too long.
2. Tasks with large burst time can be a bottleneck, because it would significantly increase the waiting time for other processes.

▼ SJF (Shortest Job First) algorithm

Semantics: the process that will utilize CPU for the shortest period of time (smallest burst time) will be the first to get the CPU. If these are two or more task have the same burst time, in this scenario, the FCFS algorithm is used to resolve the collision.

The SJF can be preemptive. In such case, the short-term scheduler can take CPU away from a running process, when a newly added into the ready queue process has the shorter burst time that the currently running process. Therefore, a process with the shortest burst time runs until it terminates or goes to the I/O queue.

Also, SJF can be cooperative. In case where, there are two or more tasks in the ready queue that have the same burst time.

Note: the SJF is a variation of the priority scheduling algorithm where the priority is inverse burst time of a process.

Scheduling criteria:

- Waiting time = Total waiting time - No. of time units process had executed - Arrival time.

Issues:

- The algorithm relies on the burst time of processes, but it is complicated to calculate. One of the ways to calculate the next CPU burst time is to approximate.
- The algorithm cannot be implemented on the short-term scheduling level only.

▼ Priority Scheduling algorithm

Semantics: A priority is associated with each process and the short-term scheduler selects the process with the highest priority. Equal priority tasks are scheduled by the FCFS algorithm.

Priority scheduling algorithm can be preemptive and non preemptive (cooperative):

- **Preemptive**: The algorithm preempts the CPU from a task if a newly added in the ready queue task has higher priority than the currently execution task.
- **Non preemptive:** The algorithm puts the new process in the head of the ready queue, but does not preempt a currently execution process.

Scheduling criteria:

- ...

Issues:

- Indefinite blocking (starvation): A process that is ready to run but waiting for CPU can be considered as blocked. The algorithm can leave low priority tasks blocked indefinitely. The issues can be solve using the aging technique. Aging can be represented as a special variable by which the task priority is corrected. The value increases when a task is waiting for execution.

▼ RR (Round Robbin) algorithm

This algorithm is primary designed for time sharing systems and it is similar to the FCFS algorithm, but preemption added to switch between processes. A small unit of time, called a time quantum or time slice is defined (generally from 10 up to 100 ms.).

Semantics: A newly created process gets a time slice assigned, and placed to the circular ready queue. The low-term scheduler selects one process at a time in queue order. The selected process is being executed only for its slice time, then the scheduler preempts the CPU and the process is placed back to the circular queue, in case, it did not terminate. The cycle continues round by round until the circular queue is empty.

Implementation details:

- Data structure: Ready queue (FIFO).
- A new process is added to the tail of the queue.
- The sort-term scheduler picks the first (queue head) process from the queue, sets the time interrupt after one time slice, and send signal to the dispatcher to switch processes.
- The CPU processor start execution:
  - [process has terminated]: if a process CPU burst time is less than the time slice (quantum) the process terminates it executing and releases the CPU voluntary.
  - [process has not been terminated]: A non terminated process is placed back to the ready queue. And the process state is save in the PCB.
- the short-term scheduler selects the next process to execute and sends signal to the dispatcher to make a context switch.

Scheduling criteria:

- Turnaround time = Completion time - arrival time.
- Waiting time = Turnaround time - burst time. Or Waiting time = Last Start time - Arrival time - (Preemption * Time Slice). Where Preemption is the number of time the process was preempted before final execution.

Issues:

- Complexity in finding the right time quantum (time slice). Otherwise, the RR algorithm will work as the FCFS algorithm.

▼ Multilevel queue algorithms

Semantics: Multilevel queue (MLQ) scheduling algorithm is a CPU scheduling technology that divides the ready queue into several separate queues based on type or priority of the processes (Tasks). Each queue can be managed with its own scheduling algorithm, depending on the characteristics of the process it holds.

In multilevel queue system, processes (tasks) are permanently assigned to one queue based on characteristics, such as process type (system, interactive, batch), priority or memory requirements. One assigned to a queue, a process does not move between queues.

The multilevel queue scheduler is responsible for both distributing tasks into the appropriate queues based on their type and priority and managing the scheduling policy across the queues (inter-queue scheduling), usually done by the preemptive priority scheduling algorithm.

Intra-Queue scheduling: Once, the process in their respective queues, the intra-queue scheduling is handled by the specific scheduling algorithm.

Implementation details:

- Multiple queues are created for different processes, each queue managed by its own scheduling algorithm. Each queue has different priority.
- The OS (core scheduler) moves processes (task) into different queues, usually based on the priority of each queue.

▼ Multilevel feedback-queue algorithm

Preamble: Multilevel feedback-queue scheduling (MLFQ) is an advanced type of CPU scheduling algorithms that build upon the multilevel queue (MLQ) scheduling concept. However, unlike traditional MLQ, where processes (tasks) are permanently assigned to a single queue, MLFQ allows processes (task) to move between different queues based on their behavior and CUP usage patterns.

For example, if a process uses too much CPU time, it will be moved to lower priority queue. in addition, a process that waits to too long in a low-priority queue can be moved to a higher-priority queue. This form of aging prevents process (task) starvation.

Semantics: Distribute new processes among different priority queues, each process can be reassigned to a different queue based on specific process parameters. Each queue follows its own scheduling algorithm.

Implementation details:

- The number of queues: Each queue has a lever of priority.
- The scheduling algorithm for each queue: each queue requires its own scheduling algorithm.
- The algorithm which is responsible for determining whether upgrade a process to a higher or lower priority queue.
- The algorithm that is responsible for determination of a queue to which a new process must be assigned.

Process synchronization

▼ Process synchronization

The orderly execution of cooperating processes that share logical address space. So data consistency is maintained.

▼ Critical section

Critical section is a code segments within a process where the process changes shared-memory objects (i.e., common variables, updating a table, writing a file, etc.).

If a process is executing code in a critical section, no other processes is allowed to execute code in their critical sections. (No process are executing in its critical section at the same time).

Critical section problem

The critical section problem is to design a protocol that the processes can use to cooperate.

▼ Implementation details:

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.

Example:

```
do {
    if Entry section
    {
```

```
        critical section code
    }
    Exit section
    {
        exit section code
    }
    Remainter section
    {
        remainder code
    } while (1);
```

▼ Requirements for critical section problem solution

1. **Mutual exclusion:**

   Definition: Only one process can be in its critical section at any time.

   Example: If a process P. is executing in its critical section, then no other process can be executing in their critical section.

2. **Progress:**

   Definition: When no process is in its critical section, any processes waiting to enter their critical section should be able to **make progress** in a finite amount of time.

   Example: If Process A has completed its critical section and Process B and C want to enter, the system should allow one of them to proceed based on some defined criteria without waiting indefinitely.

3. **Bounded waiting:**

   Definition: There should be a **limit on the number of times** other processes are allowed to enter their critical sections after a process has requested to enter its own critical section.

   Example: If Process A requests access to the critical section, it should eventually be able to enter, even if other processes frequently request access.

Solutions

▼ Peterson’s Solution

A classical software-based solution to the critical-section problem. This solution may not work correctly on modern computer architectures.

```
do {
    // Entry critical section
    flag[Pi] = True;
    turn = Pj; // Global
    // There is no code execution in the while loop
```

```
    // The while loop holds the futher execution of the program.
    // As long as this condition is True it will never go to the
    // critical section.
    while (flag[Pj] && turn[Pj] == True);

    // Critical section
    // Exit critical section
    flag[Pi] = False;
    // Reminder code
} while (True)
```

```
do {
    // Entry critical section
    flag[Pj] = True;
    turn = Pi; // Global
    // There is no code execution in the while loop
    // The while loop holds the futher execution of the program.
    // As long as this condition is True it will never go to the
    // critical section.
    while (flag[Pi] && turn[Pi] == True);

    // Critical section
    // Exit critical section
    flag[Pj] = False;
    // Reminder code
} while (True)
```

▼ Synchronization mechanisms for space-sharing systems

These are common solutions to the critical section problem.

▼ Mutexes (Mutual Exclusion Locks)

Description

Test and Set Lock

```
bool TestAndSet(bool *target)
{
    bool rv = *target;
    *target = true;
```

```
    return rv;
};
```

```
do
{
    while (TestAndSet(&lock)); // do nothing.
    // Critical section;
    // Exit section
    lock = False;
    // Remainder section;
} while (True)
```

Pros:

- Satisfy the mutual exclusion.

Cons:

- Bounded waiting is not satisfied.

▼ Semaphores

Description

Semaphore proposed by Edgers Dijkstra, it is a technique to manage current processes by using a simple integer value, which knows as a semaphore. Semaphore is a software based solution.

Semaphores are non-negative

```
P(Semaphore S) {
 while (S <= 0); // no execution.
 S--;
}
```

```
S(Semaphore S) {
    S++;
}
```

Note: All the modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed invisibly. That is, when one process modifies the semaphore value, no other processes can modify that same value.

Types of semaphores:

- Binary semaphore: The value of a binary semaphore can change only between 0 and 1. On some systems binary semaphores are known as *mutex locks*, as they are that provide mutual exclusion. If the value is 0 the shared resource is being used by another process (requesting process has to wait). If the value is 1 the requesting process is free to execute the critical section.
- Counting semaphore: Its value can range over an unrestricted domain, It is used to control access to a resource that have multiple instances.

Issues of semaphores:

- **Requires busy waiting:**

  Description: While one process is in its critical section, other processes have to loop continuously in their entry section. Busy waiting wastes CPU cycles that some other processes might be able to use productively. (Also called a spin lock).

  Solution: Make processes block themselves instead of continuously executing in the entry loop and wasting CPU time. The block operation places a process into a waiting queue associated with the semaphore, and state of the process is switched to the waiting state. Then the control is transferred to the CPU scheduler that selects which process to execute next. Therefore, the CPU can be used by another processes.

- **Deadlocks and starvation** (if self blocking operation is implemented):

  Description: After implementing self blocking by moving waiting processes to a waiting queue and transferring the control to the CPU scheduler, there are several more issues may arise, such as deadlocks and process starvation.

  Deadlock. If a process 0 executes (`wait(S)` and then `wait(Q)`) and at the same time the process 1 execute (`wait(Q)`, and then `wait(S)`) the deadlock arises because the process 0 cannot execute the `wait(Q)` instruction as the Q semaphore is taken by the process 1, meanwhile the process 1 cannot execute the instruction `wait(S)` as the process 0 took the semaphore S. Therefore, two processes are unable to execute further instruction, and they are locked forever.

Process 0	Process 1	Comment
`wait(S)`	`wait(Q)`	So far everything is fine.
`wait(Q)`	`wait(S)`	Process 0 cannot get the Q semaphore, as long as it is already taken by the process 1. And the process 1 cannot take the semaphore S, as long as it is already taken by the process 0.
...	...	Critical section instructions.
`signal(S)`	`signal(Q)`	Supposed to increment semaphore, but this instruction will never execute in both processes.

Process 0	Process 1	Comment
`signal(Q)`	`signal(S)`	Supposed to increment semaphore, but this instruction will never execute in both processes.

▼ Monitors

Description

Monitors are a high-level synchronization construct used to control access to shared resources and avoid conflicts in cooperative programming. Unlike low-level mechanisms like semaphores and mutex, a monitors provide a structured way to handle mutual exclusion and conditional synchronization within a single construct, making them easier to use and less error-prone.

A monitor is essentially a class or module that encapsulates shared data and provides controlled access to it. It consists of:

- Shared Variables: The data that needs synchronized access.
- Procedures/Methods: Functions within the monitor that operate on shared data.
- Mutual Exclusion: Only one process (or thread) can execute any monitor procedure at a time, ensuring safe access to shared data.
- Condition Variables: Special variables used to implement conditional waiting within a monitor. Condition variables allow processes to wait for a certain condition to be true before proceeding.

In a monitor-based solution, only one thread can be active inside the monitor at any given time, and condition variables handle waiting and signaling to ensure that threads only proceed when it's safe.

When a process enters a monitor, it gains exclusive access, and no other process can enter until it leaves. This provides **automatic mutual exclusion** for all procedures in the monitor.

Unlike mutex and semaphores, monitors use **condition variables** for **conditional synchronization**. A condition variable allows a thread to **wait for a specific condition** to become true.

- **wait()**: A process calling `wait()` on a condition variable releases the monitor lock and goes to sleep until another process signals the condition.
- **signal()**: A process calling `signal()` on a condition variable wakes up one of the waiting processes (if any), allowing it to proceed once the monitor is free.

Issues of monitors

- **Limited to High-Level Languages**: Monitors are generally a feature of high-level languages and are not available directly in low-level languages like C.

- **Limited Control**: While easier to use, monitors provide less fine-grained control over synchronization compared to low-level mechanisms like semaphores or spinlocks.
- **Potential for Deadlock**: Incorrect use of condition variables can lead to deadlocks if processes are not signaled properly.

▼ Barriers

▼ Condition variables

▼ Classic problems of synchronization

▼ The bounded-buffer (producer-consumer) problem

Description:

The bounded-buffer problem, also known as the producer-consumer problem, is a classic synchronization problem in operating systems. It illustrates the challenges of process synchronization when multiple processes (or threads) share resources—in this case, a bounded buffer with a limited capacity.

The problem involves two types of processes:

1. **Producers**: These processes generate data (or items) and place them into the shared buffer in empty slots.
2. **Consumers**: These processes remove data (or items) from the buffer to process them.

The bounded-buffer problem aims to coordinate the actions of producers and consumers so they can safely share the buffer without conflicts and without wasting time or resources.

1. The producer must not insert data when the buffer is full.
2. The consumer must not try remove data when the buffer is empty.
3. The producer and consumer should not add or remove data simultaneously.

Requirements for solution:

1. Mutual exclusion: Only one process (either a producer or a consumer) can access the buffer at the same time.
2. No starvation: Both producer and consumer should get a fair opportunity to access the buffer without being indefinitely delayed.
3. Synchronization: Producer must wait if the buffer is full. Consumer must wait is the buffer is empty.

Solution:

Initialization of semaphores: there are three semaphores are going to be used:

- **m (mutex)** — a binary semaphore which is used to acquire and release lock.

- **empty** — a counting semaphore whose initial value is the number of empty slots in the buffer. (Initially all slots in the buffer are empty).
- **full** — a counting semaphore whose initial value is 0. Counts the number of full slots in the buffer.

```
do
{
    wait(Empty); // decrease empty
    wait(Mutex); // acqure lock
    /* Add data to the buffer */
    signal(Mutex); // release lock
    signal(Full); // increase full
} while (True);
```

```
do
{
    wait(Full); // decrement full
    wait(Mutext); // aquire lock
    /* Read data from the buffer */
    signal(Mutex); // release lock
    signal(Empty); // increment empty
} while (True);
```

▼ The readers and writers problem

Description:

The Readers-Writers problem is a classic synchronization issue in operating systems that arises when multiple processes (or threads) need to access shared data, such as a database or file. Some processes only need to read the data (readers), while others need to write or update the data (writers). The goal of the Readers-Writers problem is to allow concurrent read operations while ensuring that writes are done exclusively to avoid data inconsistency.

Problem overview:

- **Readers**: Processes that only read the shared data. Multiple readers should be allowed to read concurrently without interference since reading doesn't alter the data.
- **Writers**: Processes that modify or update the shared data. Only **one writer** should be allowed access to the data at any time to ensure data integrity.

Types of the readers-writers problem:

There are three main variations of the Reader-Writer problem, each prioritizing different needs:

1. First Readers-Writers problem (Reader Priority): This allows any reader to access the shared data as long as there is no writer actively writing, which can cause **writer starvation** if there's a constant flow of readers.
2. Second Readers-Writers problem (Writer Priority): Once a writer requests access, no new readers are allowed to start reading until the writer has completed. This can cause **reader starvation** if writers are frequently requesting access.
3. Third Readers-Writers problem (Fair solution): Ensures **fairness** between readers and writers by using **queue-based scheduling** or **fair access mechanisms**. Both readers and writers get a chance to access the data in the order they arrive.

Solution using semaphores (reader priority):

Initialize semaphores:

1. mutex — a binary semaphore that initialized to 1, and used to ensure mutual exclusion when the read count is updated. i.e. when any reader enters or exits from the critical section.
2. wrt — a semaphore (initialized to 1) common to both reader and writer processes.
3. read_count — and integer variable (initialized to 0) that keeps the tack of how many processes are currently reading the object.

```
do
{
/* Writer requests for crtical
section */
wait(wrt);
// Perform write;
// exit section
signal(wrt)
} while (True);
```

```
do
{
wait(mutex);
read_count++; // increase number of readers
if (reader_count == 1)
{
wait(wrt); // This ensures no writer can enter if there is ever 1 read
}
signal(mutex); // other readers can enter while this one is reading.
// Reading section (critical section for readers)
```

```
// Code to read the shared data
wait(mutex);
read_count--;
if (reader_count == 0)
{
signal(wrt);
}
signal(mutex);
} while (True);
```

▼ The dining philosophers problem

Description:

The Dining Philosophers Problem is a classic synchronization problem that demonstrates the challenges of allocating shared resources among multiple processes in a way that avoids deadlock and starvation. It models a situation where multiple processes (philosophers) need to share limited resources (forks), which leads to potential conflicts.

The problem was formulated by Edsger Dijkstra and illustrates issues in concurrent programming and process synchronization, especially in cases where processes need exclusive access to resources that are shared.

Problem overview:

- Processes (philosophers): a fixed number of processes. Has two states work on shared resourced (eat), and do not work on shared resources (think).
- Resources (forks): a fixed number of shared resources.

The problem arises when a process in order to work on shared resources is required two resources simultaneously, hence the number of shared resources is less than the number of processes that are able to utilize these resources at the same time.

Requirements for solution:

- No Deadlock: The system should be designed so that philosophers don’t end up in a state where each is waiting for a fork held by another, causing deadlock.
- No Starvation: Every philosopher should eventually get a chance to eat, avoiding indefinite waiting for resources.
- Concurrency: Philosophers should be able to eat concurrently when possible (for instance, if their neighbors are not eating).

Solution using semaphores:

Each (fork) is a shared data. The shared data access can be regulated using binary mutex.

```
# Initialize forks as semaphores (one for each fork)
forks = [Semaphore(1) for _ in range(5)]
def philosopher(i):
    while True:
        think()                  # Philosophers are thinking
        # Pick up forks with lower-numbered fork first
        if i % 2 == 0:
            forks[i].wait()      # Pick up left fork
            forks[(i+1) % 5].wait()  # Pick up right fork
        else:
            forks[(i+1) % 5].wait()  # Pick up right fork
            forks[i].wait()      # Pick up left fork
        eat()                    # Philosophers are eating
        # Release both forks
        forks[i].signal()        # Put down left fork
        forks[(i+1) % 5].signal()  # Put down right fork
```

Possible resolution of the deadlock problem:

- Allow only a limited number of processes to work on the shared data. So if all processes acquire locks simultaneously there will the last lock which will allow processes to execute their critical sections in order.
- Allow process to start its critical sections only if two shared data resources are available simultaneously.
- Asynchronous solution: allow odd processes to start to work on the a certain number of shared resources (left forks), whereas an even processes are allowed to start to work on the opposite number of shared resources (right forks).

💡 **Credits:**

1. https://www.geeksforgeeks.org/
2. https://www.tutorialspoint.com/operating_system
3. https://www.nesoacademy.org/