Aging in Operating Systems (OS) refers to a technique used in process scheduling to ensure that older, lower-priority processes eventually get a chance to execute. It helps prevent **starvation**, where certain processes (usually low-priority ones) are perpetually delayed in favor of higher-priority

Semaphore is a method of **Inter-Process Communication (IPC)** used for synchronizing access to shared resources between processes or threads to avoid race conditions. Semaphores help in managing resource allocation in a multi-process environment.

Process Duplication: This refers to creating a copy of a process, not an IPC mechanism.

Signal Handling: Signals are used to notify a process of certain events but are not typically considered an IPC method.

Process Scheduling: This is a way the operating system decides which process runs when, but it's not a method of communication between processes.

The **SIGCHLD** signal in Unix-like operating systems is sent by a child process to its parent process when the child process **terminates**, **stops**, or resumes after being stopped. It notifies the parent that it can now execute actions related to the child process, like cleaning up resources or retrieving the child's exit status.

The signal used by the kernel to inform a process that it has attempted an illegal operation, such as **division by zero**, is **SIGFPE**.

SIGFPE (Signal Floating Point Exception):

- **SIGFPE** stands for **Signal Floating Point Exception**, but it is used for various types of arithmetic errors, not just floating-point exceptions.
- This signal is raised when a process performs an illegal arithmetic operation, such as:
    - Division by zero.
    - Overflow or underflow in arithmetic operations.
    - Invalid floating-point operation.

In Unix-like operating systems, **orphaned processes** are processing whose parent process has terminated or exited while they are still running. These orphan processes are not left unattended; the operating system ensures they are handled properly by **re-parenting** them to a special process.

The main advantage of using **message queues** over **signals** for **Inter-Process Communication (IPC)** is that **message queues provide more flexibility and allow for data exchange** between processes, while **signals are limited to simple notifications** without data transfer.

Linux provides several mechanisms for **temporary storage of data** between **producer** and **consumer processes**, but one of the most commonly used is the **pipe**.

Pipes:

- **Pipes** in Linux provide a **unidirectional** communication channel between two processes. They allow a producer process to write data into the pipe and a consumer process to read from it. The data is temporarily stored in a buffer, allowing communication between processes running at different speeds.
- Pipes are especially useful for **short-term communication**, where the producer and consumer don't need the data to persist once it's read.

There are two types of pipes:

1. **Anonymous Pipes**: Used for communication between processes that have a parent-child relationship (created using fork()). Data written to the pipe by the producer process can be read by the consumer process.
2. **Named Pipes (FIFOs)**: Allow communication between unrelated processes. A named pipe persists in the filesystem and allows processes to exchange data by reading and writing to the same file.

Other Mechanisms:

- **Message Queues**: Asynchronous communication mechanism that allows processes to send and receive messages in a queue.
- **Shared Memory**: Provides a memory segment that can be shared between processes, allowing direct access to the data.
- **Sockets**: Used for communication over a network or between processes on the same machine, supporting both connection-oriented and connectionless communication.

Each of these mechanisms has its own use case depending on the requirements for persistence, communication speed, and complexity.

Both **mutex** (mutual exclusion) and **semaphore** are synchronization primitives used in concurrent programming to control access to shared resources by multiple threads or processes. Although they serve similar purposes, they differ in their behaviour and use cases.

1. Mutex (Mutual Exclusion)

A **mutex** is a locking mechanism used to ensure that only one thread or process can access a resource at a time. It allows *exclusive access* to a shared resource and is suitable when you need strict control over resource usage.

- **Characteristics of a Mutex:**
    - **Binary Lock:** A mutex has two states: locked (1) or unlocked (0).
    - **Ownership:** A thread that locks (acquires) the mutex becomes the owner of the mutex, and only that thread can unlock (release) it.
    - **Blocking:** If a thread tries to lock a mutex that is already locked by another thread, it is put to sleep (blocked) until the mutex is unlocked.

    - **Mutual exclusion:** Only one thread can own the mutex at any given time.
- **Use Case:** Mutexes are typically used when you need to protect critical sections, ensuring only one thread accesses the shared resource at any time.

2. Semaphore

A **semaphore** is a more general synchronization primitive that can be used to control access to a shared resource by more than one thread. It can allow multiple threads to access a resource concurrently, depending on the value of the semaphore.

- **Types of Semaphores:**
    - **Binary Semaphore:** Similar to a mutex, it can only take values 0 and 1 (locked/unlocked).
    - **Counting Semaphore:** A semaphore that can take non-negative integer values. It controls access to a resource pool, where the count indicates how many threads/processes can access the resource simultaneously.
- **Characteristics of a Semaphore:**
    - **Non-binary values:** Counting semaphores allow more than one thread to access the resource. The semaphore's value indicates how many resources are available.
    - **No ownership:** A semaphore doesn't enforce ownership, meaning any thread can signal (increment) or wait (decrement) the semaphore, unlike a mutex where only the owning thread can release it.
    - **Signal and Wait (P and V operations):**
        - **Wait (P operation)**: Decrements the semaphore. If the semaphore's value is positive, the thread can proceed. If it is zero, the thread is blocked until the value becomes positive.
        - **Signal (V operation):** Increments the semaphore, potentially unblocking a waiting thread.
- **Use Case:** Semaphores are useful when multiple instances of a resource need to be accessed by multiple threads (e.g., managing a resource pool like connections to a database or controlling access to a finite number of resources).

Key Differences:

Aspect	Mutex	Semaphore
Control	Provides exclusive access (binary lock)	Can allow multiple threads (counting)
Type	Binary (locked/unlocked)	Can be binary or counting
Ownership	Only the thread that locked it can unlock	No ownership; any thread can signal or wait
Use Case	Mutual exclusion (one thread accesses resource)	Resource management (multiple threads can access)

Aspect	Mutex	Semaphore
Blocking	Threads block if mutex is locked	Threads block if semaphore value is zero
Common Usage	Protecting critical sections (one user at a time)	Managing resource pools (multiple users)

Example of Usage:

- **Mutex:** Used to ensure that only one thread can modify a shared data structure, like a queue or file, at a time.
- **Semaphore:** Used to control access to a pool of resources, like limiting access to 10 connections in a database connection pool.

Summary:

- **Mutex** is simpler, used for mutual exclusion, and ensures that only one thread can access a resource at a time.
- **Semaphore** can be used to allow multiple threads to access a finite number of resources and can also function like a mutex when implemented as a binary semaphore.

meminfo

In Linux-based systems, /proc/meminfo provides detailed information about system memory usage. You can view this file by running the following command:

cat /proc/meminfo

Here are some common fields from /proc/meminfo and their descriptions:

- **MemTotal**: Total usable RAM (i.e., physical RAM minus some reserved bits and the kernel binary code).
- **MemFree**: The amount of physical RAM that is currently unused.
- **MemAvailable**: An estimate of how much memory is available for starting new applications, without swapping.
- **Buffers**: Memory used by the kernel buffers for I/O operations.
- **Cached**: Memory used by the page cache and slabs (recently accessed files).
- **SwapTotal**: Total swap space available.
- **SwapFree**: The amount of swap space that is currently unused.
- **Shmem**: Total shared memory (from tmpfs or shared memory segments).
- **Active**: Memory that has been used recently and is likely to be used again soon.
- **Inactive**: Memory that hasn't been used recently and can be swapped to disk if necessary.