**Interprocess Communication (IPC)**

* **Independent Processes:**
  + These processes do not share any resources or data with other processes.
  + Their execution is unaffected by the execution of other processes.
  + They operate in isolation.
* **Cooperating Processes:**
  + These processes can share data, resources, or synchronize their execution with other processes.
  + Their execution can affect or be affected by other processes.

**Reasons for Cooperating Processes:**

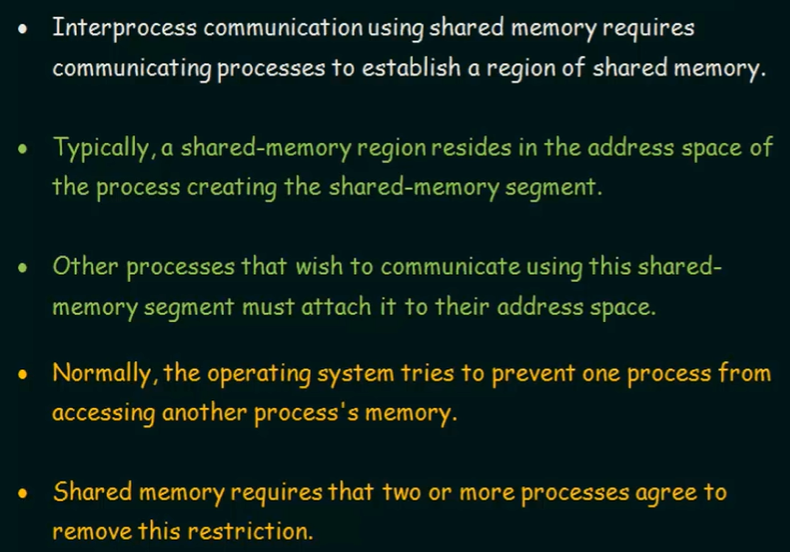
* **Information Sharing:**
  + Multiple processes may need to access and modify the same data.
  + Examples: A shared database, a collaborative document editor.
* **Computation Speedup:**
  + Dividing a complex task into smaller subtasks and executing them in parallel can significantly reduce execution time.
  + This is often achieved through multithreading or multiprocessing.
* **Modularity:**
  + Breaking down a system into smaller, cooperating processes enhances modularity and simplifies development and maintenance.
  + Each process can perform a specific function, improving code organization.
* **Convenience:**
  + Users may want to perform multiple tasks simultaneously, which can be accomplished through cooperating processes.
  + Examples: A web browser with multiple tabs, a media player with background tasks.

**Cooperating Processes**

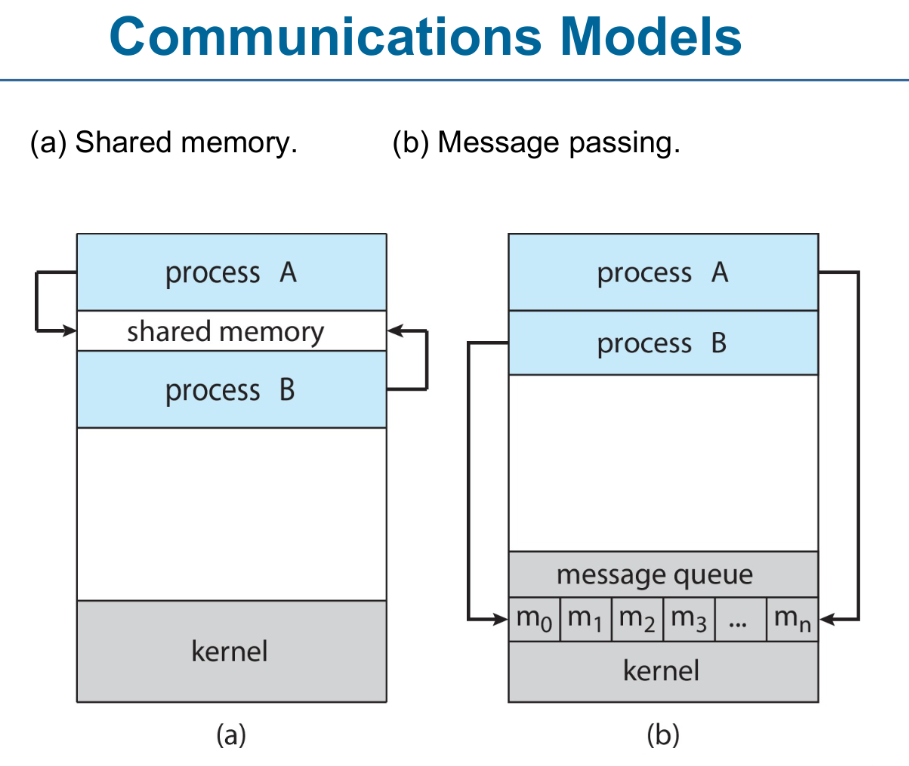
Cooperating processes require an **inter-process communication (IPC)** mechanism that will allow them to exchange data and information and synchronize their activities.

**Two Models of IPC for cooperating processes:**

1. **Shared Memory:**
   * Cooperating Processes share a region of memory that they can read and write to.
   * This is a fast and efficient way to exchange large amounts of data.
   * Requires careful synchronization to prevent race conditions and data corruption.
   * The OS is responsible for setting up the shared memory region.
   * The communication is under the control of the users processes and not the operating system.
   * The processes themselves are responsible for synchronizing access to the shared memory.



1. **Message Passing:**
   * Cooperating Processes communicate by exchanging messages.
   * This model is suitable for distributed systems or when processes are located on different machines.
   * It is generally slower than shared memory but provides better protection and synchronization.
   * The OS is responsible for facilitating message delivery.
   * Subcategories of message passing include:
     + **Direct communication:** Processes name each other explicitly.
     + **Indirect communication:** Messages are sent and received through mailboxes or ports.



**Shared Memory**

A portion of the computer's memory is designated as "shared memory."

The operating system provides mechanisms to allow multiple processes to map this region into their address spaces.

Once a process has mapped the shared memory region, it can access it just like any other memory location. Multiple processes can have direct access to a common memory region.

**Producer-Consumer Problem:**

* **Paradigm:** It's a fundamental paradigm for understanding how cooperating processes can communicate and synchronize when one process (the producer) generates data and another process (the consumer) consumes that data.
* **Shared Buffer:** The producer and consumer share a common buffer (a data structure in memory) to exchange data.

**Key Challenges:**

* **Synchronization:** The producer and consumer must synchronize their access to the shared buffer to prevent data corruption.
* **Mutual Exclusion:** Only one process (producer or consumer) should be allowed to access the buffer at any given time.
* **Buffer Management:** The buffer must be managed to ensure that the producer doesn't overwrite data that hasn't been consumed and that the consumer doesn't try to consume data that hasn't been produced.

**Types of Buffers:**

1. **Unbounded-Buffer:**
   * **Definition:** An unbounded buffer has no practical limit on its size. The producer can always add data to the buffer, and the consumer can consume data at its own pace. The consumer may have to wait for new items, but the producer can always produce new items.
   * **Characteristics:**
     + Simpler to implement.
     + No risk of the producer blocking because the buffer is full.
     + Potentially consumes excessive memory if the producer produces data faster than the consumer can consume it.
   * **Practicality:** In real-world systems, a truly unbounded buffer is impossible due to memory limitations. However, some systems might use very large buffers to approximate an unbounded buffer.
2. **Bounded-Buffer:**
   * **Definition:** A bounded buffer has a fixed size. The consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
   * **Characteristics:**
     + More complex to implement due to the need for synchronization and buffer management.
     + Prevents excessive memory consumption.
     + Requires mechanisms to handle buffer full and buffer empty conditions.
   * **Synchronization:**
     + **Full Buffer:** If the buffer is full, the producer must wait until the consumer removes data.
     + **Empty Buffer:** If the buffer is empty, the consumer must wait until the producer adds data.
     + Because multiple processes can access the shared memory concurrently, careful synchronization is essential.
     + Without proper synchronization, **race conditions** can occur, leading to data corruption and unpredictable behavior.
     + Common synchronization techniques include:
       - Semaphores
       - Mutexes (mutual exclusion locks)
       - Condition variables

**Advantages of Shared Memory:**

* **Speed:** Shared memory is the fastest form of IPC because it avoids the overhead of copying data between processes.
* **Efficiency:** It's very efficient for exchanging large amounts of data.

**Real-World Examples:**

* **Print Spooler:** The producer is the application printing a document, and the consumer is the printer.
* **Web Server:** The producer is the network connection receiving requests, and the consumer is the web server processing those requests.
* **Multimedia Player:** The producer is the component reading data from a file or network stream, and the consumer is the component playing the audio or video.
* **Operating System Pipes:** Pipes in operating systems use a bounded buffer to transfer data between processes.
* **High-performance computing:** Applications that require fast data exchange between processes.
* **Database systems:** For sharing data between database processes.
* **Multimedia applications:** For sharing large buffers of audio or video data.
* A black text on a white background

  AI-generated content may be incorrect.**Image processing:** For sharing image data between processing modules.

A white screen with black text

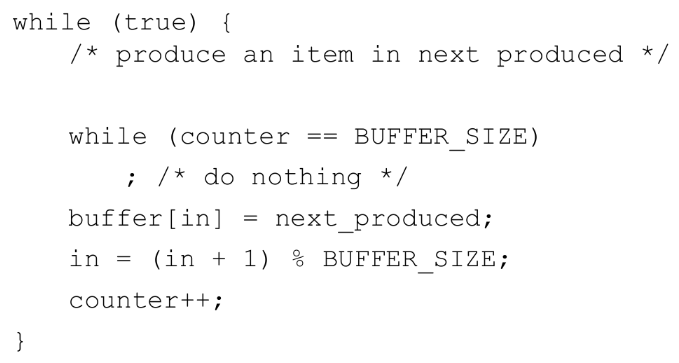
AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

* Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
* We can do so by having an integer counter that keeps track of the number of full buffers.
* Initially, counter is set to 0.
* The integer counter is incremented by the producer after it produces a new buffer.
* The integer counter is and is decremented by the consumer after it consumes a buffer.
* This solution will **require synchronization** to prevent race conditions because both processes are accessing and modifying the same variable/data.

A white background with black text

AI-generated content may be incorrect.

**Message Passing**

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space in distributed environments, where processes reside on different machines with no direct access to shared memory. The processes or threads communicate by exchanging messages. It uses **send**(message) & **receive**(message). Messages sent by a process can be of either fixed or variable size.

**Key Characteristics:**

* **Explicit Communication:** Processes must explicitly send and receive messages.
* **Decoupling:** Message passing often leads to looser coupling between processes, as they don't rely on shared memory.
* **Synchronization:** Message passing mechanisms often include built-in synchronization features.
* **Flexibility:** It can be used in various communication patterns (e.g., one-to-one, one-to-many).

**Types of Message Passing:**

* **Synchronous Message Passing:**
  + The sender waits until the receiver acknowledges the message.
  + This provides strong synchronization but can lead to blocking.
* **Asynchronous Message Passing:**
  + The sender sends the message and continues processing without waiting for acknowledgment.
  + This allows for greater concurrency but requires mechanisms to handle message delivery and ordering.

**Common Message Passing Mechanisms:**

* **Message Queues:**
  + Processes send and receive messages to and from queues.
  + Queues provide buffering and can handle asynchronous communication.
* **Sockets:**
  + Sockets enable communication over networks.
  + They are commonly used for client-server communication.
* **Message Passing Interface (MPI):**
  + MPI is a standardized library for message passing, widely used in high-performance computing.

**Advantages:**

* **Suitable for Distributed Systems:** Message passing is well-suited for systems where processes reside on different machines.
* **Reduced Data Corruption:** By avoiding shared memory, it reduces the risk of data corruption due to race conditions.
* **Improved Modularity:** It promotes modularity and encapsulation.

**Disadvantages:**

* **Overhead:** Message passing can introduce overhead due to message serialization, transmission, and deserialization.
* **Complexity:** Designing and implementing robust message passing systems can be complex.

**Message Size:**

* **Fixed Size:**
  + Some message passing systems impose a fixed size limit on messages.
  + This can simplify buffer management and improve performance in certain scenarios.
* **Variable Size:**
  + Other systems allow messages of variable size, providing greater flexibility for data exchange.
  + However, variable-size messages can introduce complexity in buffer management and message handling.

**1. How are links established?**

Links between processes are established through **inter-process communication (IPC) mechanisms**. The method depends on the operating system and the type of communication. Common ways to establish links include:

* **Direct Communication**: Processes explicitly name each other to communicate (e.g., using process IDs).
* **Indirect Communication**: Processes communicate through intermediary objects like **mailboxes (ports)** or **message queues**.
* **Shared Memory**: Processes share a common memory segment to exchange data.
* **Pipes**: One process writes data, and another reads it (usually in parent-child relationships).
* **Sockets**: Network-based communication for processes on the same or different machines.

**2. Can a link be associated with more than two processes?**

* **Direct Communication**: Typically, a link connects **only two processes** (point-to-point communication).
* **Indirect Communication**: Yes, a **link can connect multiple processes** if they communicate through a shared object like a **message queue**, **mailbox**, or **shared memory**.

Example: A mailbox can receive messages from **many senders** and deliver them to **many receivers**.

**3. How many links can there be between every pair of communicating processes?**

* **Direct Links**: Usually **one** direct communication link exists between any pair of processes.
* **Indirect Links**: **Multiple links** can exist if processes communicate through different shared objects (e.g., several mailboxes or queues). A program could open two sockets between itself and another program.

**4. What is the capacity of a link?**

The **capacity** of a link refers to how much data it can hold before it becomes full. It depends on the **IPC mechanism**:

* **Zero Capacity**: No buffering; the sender must wait for the receiver to accept the message (synchronous communication).
* **Bounded Capacity**: A fixed buffer size where messages can be queued; the sender waits if the buffer is full.
* **Unbounded Capacity**: An unlimited buffer where messages are always accepted (rare in practice due to memory constraints).

**5. Is the size of a message that the link can accommodate fixed or variable?**

* **Fixed-size messages**: The system imposes a **fixed length** for each message (e.g., **32 bytes**, **64 bytes**).
* **Variable-size messages**: The system allows messages of **different sizes**, offering more flexibility.

**Example**:

* **Pipes** usually have a **fixed** buffer size.
* **Message queues** can support **variable** message sizes.

**6. Is a link unidirectional or bi-directional?**

* **Unidirectional**: Data flows in **one direction** (e.g., traditional pipes in Unix/Linux).
* **Bi-directional**: Data can flow in **both directions** (e.g., **sockets** or **named pipes**).

**Example**:

* **Pipes**: Typically **unidirectional**.
* **Sockets**: **Bi-directional** communication.

**Communication Links**

**1. Physical Implementation:**

These refer to the actual hardware mechanisms that facilitate communication between processes.

* **Shared Memory:**
  + In systems where processes reside on the **same machine**, shared memory can be used as a physical communication link.
  + Processes write and read messages to and from a shared memory region.
  + This is very fast but requires careful synchronization mechanisms like **semaphores,** **mutexes, or condition variables** to prevent race conditions.
* **Hardware Bus:**
  + On a single computer, a hardware bus can be used for communication between devices or processes.
  + This is common in embedded systems or device drivers.
  + Used in multiprocessor systems where CPUs share a communication bus.
  + Fast but can suffer from contention when multiple processes access the bus simultaneously.
* **Network:**
  + In distributed systems, a network (e.g., Ethernet, Wi-Fi, Internet) is used as the physical communication link.
  + Messages are transmitted as packets over the network.
  + This introduces network latency and potential for packet loss.

**2. Logical Implementation (Message Passing):**

Logical links define how processes interact at a **conceptual level**, independent of the physical medium.

* **Direct or Indirect:**
  + **Direct Communication:**
    - Processes explicitly name each other when sending and receiving messages.
    - Links are automatically established when processes communicate.
    - Each link is between exactly two processes.
    - A unique link exists per process pair.
    - The link may be unidirectional, but is usually bi-directional.
    - Example: send(P, message) sends a message to process P.
    - Limitation: Not scalable for systems with many processes.
    - Examples: Sockets, pipes.
  + **Indirect Communication:**
    - Processes communicate through a shared intermediary, such as a message queue, mailbox/port.
    - Processes send messages to the intermediary, and the intermediary delivers them to the intended recipient.
    - This decouples processes and provides greater flexibility.
    - Each mailbox has a unique id.
    - Processes can communicate only if they share a mailbox.
    - A link may be associated with many processes.
    - Each pair of processes may share several communication links.
    - Link may be unidirectional or bi-directional.
    - **4 Operations:**
      * **create** a new mailbox (port)
      * **send** and **receive** messages through mailbox
      * **delete** a mailbox
    - Primitives are defined as:
      * send(A, message) – send a message to mailbox A
      * receive(A, message) – receive a message from mailbox A
    - Send and receive are flexible, for example **four options** if mailbox full:
      * Wait indefinitely
      * Wait at most n milliseconds
      * Return immediately
      * Temporarily cache a message
* **Synchronization:**
  + **Synchronous Communication:**
    - Blocking is considered synchronous
    - Blocking send -- the sender is blocked until the message is received
    - Blocking receive -- the receiver is blocked until a message is available
    - Examples: Rendezvous communication, remote procedure calls (RPCs).
  + **Asynchronous Communication:**
    - Non-blocking is considered asynchronous
    - Non-blocking send -- the sender sends the message and continue
    - Non-blocking receive -- the receiver receives:
    - A valid message, or Null message
    - Examples: Message queues, email.
  + **Rendezvous:**
    - It is a form of synchronous message passing.
    - Both processes (sender and receiver) must wait until the other is ready.
    - The message transfer only occurs when both processes are synchronized—this is called a rendezvous point.
    - Until both processes are ready:
      * The sender is blocked (waiting) if no receiver is available.
      * The receiver is blocked if there is no message to consume.
    - If two processes using rendezvous communication are waiting for each other to send a message, and neither process is willing to send until it receives, a deadlock can occur.
    - Examples of rendezvous communication include message queues, pipes and Remote Procedure Call (RPC).
* **Buffering:**
  + **Automatic Buffering:**
    - The message passing system automatically buffers messages during transmission.
    - This provides a degree of decoupling and can improve performance.
    - However, it can also introduce complexity in buffer management.
    - **The system provides temporary storage for messages until received.**
      * **Zero Capacity:** No buffer—sender must wait if the receiver is not ready (rendezvous).
      * **Bounded Capacity:** Limited buffer— finite length of n messages, sender waits if the buffer is full.
      * **Unbounded Capacity:** Infinite buffer— infinite length, sender never waits.
  + **Explicit Buffering:**
    - User-controlled message queues. The sender or receiver process explicitly manages message buffers.
    - Processes manually manage how much data to store.
    - This provides greater control but requires more complex programming.
    - Example: a program that allocates a buffer to hold the incoming data from a socket.