2.4 INTERPROCESS COMMUNICATION (IPC)

The way in which the process communicate with each other is known as interprocess communication.

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information.

Independent processes:

- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.

Cooperating processes:

- A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system.
- Any process that shares data with other processes is a cooperating process.

Need:

a) **Information sharing**.

- Users may be interested in the same piece of information.
- Environment should allow concurrent access to such information.

b) **Computation speedup**.

If we want a particular task to run faster, break it into subtasks, each of which will be executing in parallel with the others.

c) **Modularity**.

Construct the system in a modular fashion i.e.) dividing the system functions into separate processes or threads.

d) **Convenience**.

- Even an individual user may work on many tasks at the same time
  - Example: a user may be editing, listening to music, and compiling in parallel.

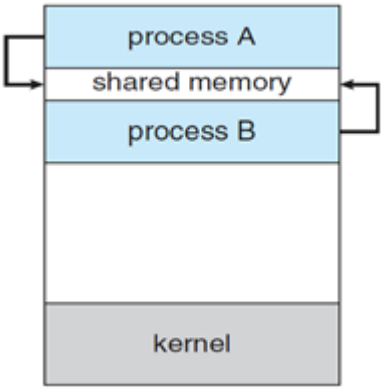
There are fundamental models of interprocess communication:

1) **Shared memory**

2) **Pipe**
3) **Semaphore**

1) Shared memory Systems

A region(portion) of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.



A shared-memory region resides in the address space of the process creating the shared-memory segment.

- System calls are required only to establish shared memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.
- Process ensures that they are not writing to the same location simultaneously.

Example: producer – consumer problem,

A **producer** process produces information that is consumed by a **consumer** process.

Shared memory is one of the solution to the producer – consumer problem.

- To allow producer and consumer processes to run concurrently.
- A buffer memory is used as shared region, where producer fills the item and consumer use/empties it.
- A producer can produce one item while the consumer is consuming another item.
- The consumer does not try to consume an item that has not yet been produced

Advantage:

- faster than message passing, no kernel assistance is required

Disadvantage:

- Data is shared by all users
- Shared memory suffers from cache coherency issues, which arise

because shared data migrate among the several caches

Types of Buffer.

i) Unbounded buffer

- There is no limit on the size of the buffer.
- The consumer may have to wait for new items, but the producer can always produce new items.

ii) Bounded buffer

- The buffer is of fixed buffer size.
- In this case, the consumer waits if the buffer is empty, and the producer wait if the buffer is full.

2. PIPES

Pipes are a fundamental Inter-Process Communication (IPC) mechanism in operating systems. A **pipe** is a unidirectional communication channel used for **interprocess communication** between **related processes** (like parent and child). They act as a conduit(channel) for data transfer between processes, allowing the output of one process to serve as the input for another.

Pipes are a type of IPC (Inter-Process Communication) technique that allows two or more processes to communicate with each other by creating a unidirectional or bidirectional channel between them. A pipe is a virtual communication channel that allows data to be transferred between processes, either one-way or two-way. Pipes can be implemented using system calls in most modern operating systems, including **Linux, macOS, and Windows**.

The **pipe()** system call in operating systems facilitates interprocess communication by creating a unidirectional communication channel between two processes. It allows one process to **write** data into the pipe, while another process can **read** from it. This mechanism is particularly useful for achieving coordination and data transfer between processes, such as in pipelines or filters.

Example: producer- consumer problem

Key concepts

- **Unidirectional flow**: Data flows in one direction only. A process writes to one end of the pipe (the write-end), and another process reads from the other end (the read-end).
- **Producer-consumer model**: This is the standard communication pattern for pipes, where one process produces data and another consumes it.
- **Byte stream**: Data in a pipe is treated as a stream of bytes, meaning there's no inherent knowledge of message boundaries.
- **FIFO (First-In, First-Out):** Pipes operate on a FIFO principle, like a queue.
- **Virtual file**: The operating system manages a pipe as a "virtual file" within main memory, providing a temporary storage space for data.
- **File descriptors**: When a pipe is created, the system call returns two file descriptors: one for reading from the pipe and one for writing to it.
    - **fd** is an integer array of size **2**, which will hold two file descriptors after the pipe is created:
    - **fd[0]** - This file descriptor is for reading from the pipe (the read end).
    - **fd[1]** - This file descriptor is for writing to the pipe (the write end).

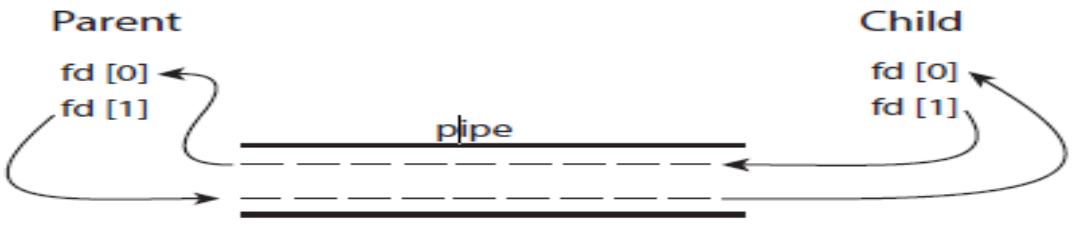


Figure File descriptors for an ordinary pipe.

Types of pipes

Two types of pipe

a) **Ordinary Pipes**
b) **Named Pipes**

a) Ordinary Pipes (unnamed pipe/Anonymous Pipes):

Ordinary pipes are a fundamental mechanism in operating systems, for enabling communication between processes. They are also known as anonymous pipes or unnamed pipes.

- **Ordinary pipes** allow **simple, one-way communication** between **parent and child processes**.
- They are widely used in Linux systems for **data exchange** in pipelines and process communication.
- Easy to implement and understand, but limited to **related processes** and **one direction**.

Mechanism:

- The pipe is created before `fork()`, so both parent and child share the pipe.
- It has **two ends**:
  - `fd[0]` → Read end
  - `fd[1]` → Write end
- One process **writes** data into the pipe, the other **reads** from it.
- It is stored temporarily in a **kernel buffer**.

Unidirectional communication

- Ordinary pipes primarily facilitate unidirectional communication, meaning data flows in one direction only.
- One end of the pipe is designated for writing (producer), and the other end is for reading (consumer).
  - Ordinary pipes are commonly used for communication between related processes, primarily a parent process and its child processes created via the `fork()` system call.
  - The parent process typically creates the pipe, and the child process inherits the file descriptors for the pipe's read and write ends

Ordinary pipes allow two processes to communicate in producer–consumer fashion:

- the producer writes to one end of the pipe (the **write end**)
- and the consumer reads from the other end (the **read end**).

Ordinary pipes are unidirectional, allowing only one-way communication

Advantages

- **Simplicity**: A relatively straightforward way for processes to communicate.
- **Efficiency**: Can transfer data quickly with minimal overhead.
- **Reliability**: Can detect transmission errors and ensure correct data delivery.

Disadvantages

- **Unidirectional**: Ordinary pipes are unidirectional by default; two-way communication requires two pipes.
- **Limited Capacity**: Pipes have a fixed-size buffer, which can limit the amount of data transferred at once.
- **No Broadcasting**: Cannot broadcast data to multiple receivers.
- **Requires Related Processes** (**Ordinary Pipes**): Ordinary pipes require a parent-child relationship between processes.
- **No Message Boundaries**: Data is treated as a stream of bytes, without knowledge of message boundaries.

Example:

- The command `ls -l | grep "txt"` uses a pipe to send the output of `ls -l` (list files and directories) as input to `grep "txt"` (filter lines containing "txt"), effectively listing only files and directories with the "txt" extension.

b) Named Pipes (FIFOs):

- Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for Communication.
- **Named Pipe** (also called **FIFO**) is a **special type of file** used for **unidirectional interprocess communication (IPC)**. Unlike ordinary (unnamed) pipes.

- Which are **used to establish two-way communications between two unrelated programs**
- Multiple processes can access this special file for reading and writing like any ordinary file.
- To achieve bidirectional communication, you would typically need to set up two separate ordinary pipes, one for each direction

Mechanism:

1. A FIFO special file is created using mkfifo() or the shell command mkfifo.
2. One process opens it for **writing**, the other for **reading**.
3. The kernel buffers data written to the FIFO until it is read.
4. Data flows like a **first-in, first-out** queue.

Advantages of Named Pipes (FIFOs)

Advantage	**Description**
1.Inter-process Communication (IPC)	Named pipes allow **communication between unrelated processes**, unlike anonymous pipes which work only for parent-child processes.
2. Simplicity	Easier to implement and use compared to more complex IPC mechanisms like sockets or shared memory.
3.Bidirectional Communication	With two pipes (one for each direction), full-duplex communication can be achieved.
4. File System Presence	They appear as a special file in the file system, allowing **easy access via file paths** and standard file I/O operations.
5. Blocking Behavior	The blocking nature (writer waits for reader and vice versa) can be used to synchronize processes easily.
6. Language Agnostic	Can be used across programs written in different programming languages (as long as they support file I/O).
7. Security Control	File system permissions can be applied to control access to the named pipe.

Disadvantages of Named Pipes (FIFOs)

Disadvantage	Description
1. Limited to Same Host	Named pipes **cannot be used over a network**, unlike sockets. Communication is restricted to the same machine.
2. No Persistence	Data is not stored permanently; it exists only while being transferred and read. Once read, it's gone.
3. Buffer Size Limits	Pipes have a **limited buffer size**. If the buffer fills and the reader is slow or absent, the writer blocks.
4. One-Way by Default	Named pipes are unidirectional by default. Full-duplex communication requires setting up **two separate pipes**.
5.Synchronization Overhead	Requires manual coordination between reader and writer processes to avoid deadlocks or data inconsistency.
6. No Random Access	Pipes are **stream-based** — you cannot seek or jump to a specific position in the data.
7.Performance Overhead	Compared to shared memory, pipes can be **slower** due to kernel involvement in every read/write operation.

System calls in Unix/Linux

- **pipe() :** Creates a pipe and returns two file descriptors, one for the read end and one for the write end.
- **fork() :** Creates a child process, allowing both parent and child to access the pipe through inherited file descriptors.
- **close() :** Closes a file descriptor, releasing the pipe resource when all ends are closed.
- **read() :** Reads data from the pipe's read end.
- **write() :** Writes data to the pipe's write end.

Working of pipes:

1. **Creation**: A process creates a pipe using the `pipe()` system call. This establishes a read-end and a write-end, represented by file descriptors.

2. **Communication Setup**: For successful communication, the writing process should close the read end of the pipe, and the reading process should close the write end.

3. **Data Flow**: The writing process uses the `write()` system call to send data to the pipe's write-end. The reading process uses the `read()` system call to receive data from the pipe's read-end.

4. **Blocking**: If a process attempts to read from an empty pipe, it will be blocked (suspended) until data is written to the pipe. Conversely, if the pipe's buffer becomes full, a writing process will be blocked until space becomes available.

5. Closing the file descriptor: Both the open file descriptors must be closed at the end of the process to execute the program successfully. Zero is returned on success by the close() system call.

3. SEMAPHORES

Definition

A semaphore is an integer variable that controls access to a resource. It uses two atomic operations:

- wait() (also called **P** or **down**): Decrements the semaphore. If the value becomes negative, the process is blocked.
- signal() (also called **V** or **up**): Increments the semaphore. If other processes are waiting, one may be unblocked.

Semaphores are used to control access to a limited number of resources. They maintain a counter representing the number of available resources. Threads decrement the counter when acquiring a resource and increment it when releasing. If the counter is zero, threads attempting to acquire a resource are blocked until a resource becomes available.

Mechanism:

A semaphore uses an **integer counter**:

- **wait (P operation)**: Decreases the semaphore value.
  - If the value is **> 0**, the process continues.
  - If the value is **0**, the process is **blocked**.
- **signal (V operation)**: Increases the semaphore value.
  - Wakes up one of the blocked processes (if any).

Two standard operation

a) Wait( ) :

- used to test the semaphore's value.
- Decrements the semaphore's value.
- If the value becomes negative, the process is blocked.
- Denoted using P symbol

Definition of wait():

```
wait(P)
{
    while (P <= 0);                    // busy wait process itself blocked
    P--;
}
```

b)Signal( ) :

- used to increment the semaphore's value.
- It gives signal to the waiting process to access it. (if the value is negative)
- It activates a process blocked on the semaphore
- Denoted using V symbol

Definition of signal():

```
signal(V)
{
    V++;
}
```

wait and signal operations is to control the number of processes that can access a critical section of code or a shared resource i.e., if one process modifies the semaphore value, no other process can modify the semaphore value at the same time.

Example:

When a common resource to be access is given to the critical section. When a process is wants to use the resource it will give wait() here semaphore value is decremented. The process start using the resources. After using the resource signal() is called which increment the semaphore value. So that the process waiting for the resource can access that resources.

Advantage:

- Semaphores help prevent data corruption.
- Semaphores ensure proper synchronization between concurrent processes

Types of Semaphores:

- **Binary Semaphore:**
  - This is also known as a mutex lock.
  - locks that provide mutual exclusion.
  - It two values 0 and 1.
  - Its value is initialized to 1.
  - It is used to implement the solution of critical section problems with multiple processes and a single resource.

- **Counting Semaphore:**
  - Used to control access to a resource that has multiple instances.
  - Initialized to the number of resources available.
  - Its value can range over an unrestricted domain.

Semaphore Implementation

- Busy waiting problem is overcome by modify the definition of the wait() and signal() operations
- When a process executes the wait() operation and if the semaphore value is not positive, it must wait. (rather than busy in waiting)
- The process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

Definition of **wait**() semaphore :

```
wait(semaphore *P)
{
        ------
        -----
        block();
}
```

Definition of **signal**() semaphore:

```
signal(semaphore *V)
{
    ---
    ----
    wakeup(V);

}
```

- Blocked process should be restarted.
- The process is restarted by a wakeup() operation when some other process executes a signal() operation.
- wakeup() which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue.
- Integer value of the semaphore is the list of processes.

- When a process wait, it is added to the list.
- A signal() operation removes one waiting process from the list awakens that process using **wakeup(V)** operation.

System Calls Used in Linux (System V Semaphores):

Linux supports two main semaphore APIs:

1. **System V Semaphores**

- **semget()** – Create a semaphore set or access an existing one.
- **semop()** – Perform operations like wait (P) or signal (V).
- **semctl()** – Control operations (remove, set values, etc.)
- Suitable for complex multi-process synchronization.

2. **POSIX Semaphores**

- Created using sem_open() (named) or sem_init() (unnamed)
- Operated with sem_wait() and sem_post()
- Easier to use and more portable

Importance of semaphores:

- **Process Synchronization**: Ensuring multiple processes safely share resources.
- **Mutual Exclusion**: Allowing only one process access to a critical section.
- **Resource Management**: Controlling access to a finite number of resources.
- **Deadlock Prevention**: Regulating the order processes acquire resources.
- **Solving Classic Synchronization Problems**: Used for problems like the Producer-Consumer and Dining Philosophers problems.