**IPC :**

**Inter-Process Communication (IPC)** is a mechanism that allows **multiple processes** to **communicate and share data** with each other. Since processes run in separate memory spaces, IPC provides ways to exchange information **safely and efficiently**.

**Why is IPC Needed?**

1. **Data Sharing** – Allows processes to share data efficiently.
2. **Synchronization** – Helps coordinate actions between processes.
3. **Resource Management** – Prevents race conditions in shared resources.
4. **Modular Design** – Enables communication between independent modules of an application.

**Types of IPC Mechanisms**

| **IPC Method** | **Description** | **Usage** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Pipes** | Unidirectional communication channel between related processes. | Parent-child processes. |

|  |  |  |
| --- | --- | --- |
| **FIFOs (Named Pipes)** | Similar to pipes but work between unrelated processes. | Process communication via filesystem. |

|  |  |  |
| --- | --- | --- |
| **Message Queues** | Queue-based communication that allows multiple processes to read/write messages. | Asynchronous communication. |

|  |  |  |
| --- | --- | --- |
| **Shared Memory** | A memory region that multiple processes can access. | Fastest IPC method. |

|  |  |  |
| --- | --- | --- |
| **Semaphores** | Used for synchronization between processes. | Avoid race conditions. |

|  |  |  |
| --- | --- | --- |
| **Sockets** | Network-based IPC that works across systems. | Internet and local IPC. |

|  |  |  |
| --- | --- | --- |
| **Signals** | Simple asynchronous notifications between processes. | Process control (e.g., SIGKILL). |

**1.Example: IPC Using Pipes (Parent-Child Communication)**

#include <stdio.h>

#include <unistd.h>

int main() {

int fd[2]; // File descriptors for the pipe

char message[] = "Hello from parent!";

char buffer[50];

pipe(fd); // Create a pipe

if (fork() == 0) { // Child Process

close(fd[1]); // Close write end

read(fd[0], buffer, sizeof(buffer));

printf("Child received: %s\n", buffer);

close(fd[0]);

} else { // Parent Process

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

write(fd[1], message, sizeof(message));

close(fd[1]);

}

return 0;

}

**Explanation of Execution:**

1. The **parent process** creates a **pipe** and writes "Hello from parent!" into it.
2. The **child process** reads from the pipe and prints the received message.
3. The communication is **one-way (parent → child)** using the pipe.

**sample output** for the **IPC using Pipes** program:

Child received: Hello from parent!

**2.Inter-Process Communication (IPC) using FIFOs in C**

**FIFO (Named Pipe)** is a method of **Inter-Process Communication (IPC)** that allows processes to communicate using a **file-like interface**. Unlike unnamed pipes, FIFOs **persist as files** in the filesystem.

**Steps to Use FIFOs:**

1. **Create a FIFO file** using mkfifo().
2. **One process writes data** to the FIFO.
3. **Another process reads data** from the FIFO.

**Simple C Program for IPC using FIFO**

This example consists of **two programs**:

1. **Writer Process (producer.c)**
2. **Reader Process (consumer.c)**

**1. Producer (Writes to FIFO)**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#define FIFO\_FILE "myfifo"

int main() {

int fd;

char message[] = "Hello from Producer!";

// Create FIFO file

mkfifo(FIFO\_FILE, 0666);

// Open FIFO for writing

fd = open(FIFO\_FILE, O\_WRONLY);

// Write message to FIFO

write(fd, message, sizeof(message));

printf("Producer: Message sent!\n");

// Close FIFO

close(fd);

return 0;

}

1. **Consumer (Reads from FIFO)**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#define FIFO\_FILE "myfifo"

int main() {

int fd;

char buffer[100];

// Open FIFO for reading

fd = open(FIFO\_FILE, O\_RDONLY);

// Read message from FIFO

read(fd, buffer, sizeof(buffer));

printf("Consumer: Received message -> %s\n", buffer);

// Close FIFO

close(fd);

return 0;

}

**How to Compile and Run:**

1. **Open two terminals.**
2. **Compile the programs:**

**gcc producer.c -o producer**

**gcc consumer.c -o consumer**

3.Run the consumer first (waiting for message):

**./consumer**

4.Run the producer (sending message):

**./producer**

1. Output on the consumer's terminal:

**Consumer: Received message -> Hello from Producer!**

**3.Simple C Program for IPC using Shared Memory**

**Shared memory** is the fastest IPC mechanism, as it allows multiple processes to access a common memory segment directly.

**Steps for Shared Memory IPC**

1. **Create a shared memory segment** using shmget().
2. **Attach to the shared memory** using shmat().
3. **Write to or read from the shared memory**.
4. **Detach from shared memory** using shmdt().
5. **Remove the shared memory** when done.

**Shared Memory Example (Two Programs)**

This example consists of:

1. **Writer (Producer)** – Writes data to shared memory.
2. **Reader (Consumer)** – Reads data from shared memory.

**1. Producer (Writes to Shared Memory) :**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <sys/ipc.h>**

**#include <sys/shm.h>**

**#include <string.h>**

**#define SHM\_SIZE 1024 // Shared memory size**

**int main() {**

**key\_t key = 1234; // Unique key for shared memory**

**int shmid;**

**char \*shmaddr;**

**// Create shared memory segment**

**shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT);**

**if (shmid == -1) {**

**perror("shmget failed");**

**exit(1);**

**}**

**// Attach to shared memory**

**shmaddr = (char \*)shmat(shmid, NULL, 0);**

**if (shmaddr == (char \*)-1) {**

**perror("shmat failed");**

**exit(1);**

**}**

**// Write message to shared memory**

**strcpy(shmaddr, "Hello from Producer!");**

**printf("Producer: Message written to shared memory.\n");**

**// Detach from shared memory**

**shmdt(shmaddr);**

**return 0;**

**}**

**Consumer (Reads from Shared Memory) :**

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024 // Shared memory size

int main() {

key\_t key = 1234; // Same key as producer

int shmid;

char \*shmaddr;

// Locate shared memory segment

shmid = shmget(key, SHM\_SIZE, 0666);

if (shmid == -1) {

perror("shmget failed");

exit(1);

}

// Attach to shared memory

shmaddr = (char \*)shmat(shmid, NULL, 0);

if (shmaddr == (char \*)-1) {

perror("shmat failed");

exit(1);

}

// Read message from shared memory

printf("Consumer: Received message -> %s\n", shmaddr);

// Detach from shared memory

shmdt(shmaddr);

// Remove shared memory segment

shmctl(shmid, IPC\_RMID, NULL);

return 0;

}

**How to Compile and Run**

1. **Open two terminals**.
2. **Compile both programs:**

gcc producer.c -o producer

gcc consumer.c -o consumer

Run the producer first (writes to shared memory):

./producer

Output:

Producer: Message written to shared memory.

Run the consumer next (reads from shared memory):

./consumer

Output:

Consumer: Received message -> Hello from Producer!

**4.Simple C Program for IPC using Message Queues**

**Message Queues** allow processes to communicate by sending and receiving messages via the kernel. Unlike pipes, message queues **persist** even after the process exits.

**Steps for IPC using Message Queues**

1. **Create a message queue** using msgget().
2. **Define a message structure** with a type and content.
3. **Write messages using msgsnd()** and **read them using msgrcv()**.
4. **Remove the message queue** when done using msgctl().

**Shared Structure for Messages**

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

**1. Producer (Sender) - Writes to Message Queue**

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

#define MSG\_KEY 1234

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

int msgid;

struct msg\_buffer message;

// Create message queue

msgid = msgget(MSG\_KEY, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("msgget failed");

exit(1);

}

// Prepare the message

message.msg\_type = 1; // Message type must be > 0

strcpy(message.msg\_text, "Hello from Producer!");

// Send the message

msgsnd(msgid, &message, sizeof(message.msg\_text), 0);

printf("Producer: Message sent.\n");

return 0;

}

**2. Consumer (Receiver) - Reads from Message Queue :**

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#define MSG\_KEY 1234

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

int msgid;

struct msg\_buffer message;

// Get the message queue

msgid = msgget(MSG\_KEY, 0666);

if (msgid == -1) {

perror("msgget failed");

exit(1);

}

// Receive the message

msgrcv(msgid, &message, sizeof(message.msg\_text), 1, 0);

printf("Consumer: Received message -> %s\n", message.msg\_text);

// Delete the message queue

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

**How to Compile and Run**

1. **Open two terminals.**
2. **Compile both programs:**

gcc producer.c -o producer

gcc consumer.c -o consumer

Run the consumer first (waiting for a message):

./consumer

Run the producer (sending the message):

./producer

**Output** on the consumer's terminal:

Consumer: Received message -> Hello from Producer!