Lab 4: Inter-Process Communication (IPC)

Real-World Context

In operating systems, multiple processes often need to communicate to exchange data, synchronize tasks, or share resources. Inter-Process Communication (IPC) mechanisms like **pipes** and **FIFOs (named pipes)** enable this.

- **Pipes** allow communication between **related processes** (parent-child).
- **FIFOs** allow communication between **unrelated processes**.
- IPC ensures **data consistency**, **coordination**, and **efficient multitasking**.

Structured Use Case Challenge

Discover

• Execute all IPC programs to explore how processes communicate, coordinate, and synchronize.
• Use **unidirectional** and **bidirectional pipes** to observe the data flow between parent and child processes.
• Run **FIFO (named pipe)** programs in separate terminals to examine communication between unrelated processes.
• Observe **timestamps** in FIFO programs to analyze the timing and latency of message transmission.
• Execute **synchronous** and **asynchronous** message-passing programs to compare blocking and non-blocking communication behavior.
• Run **concurrent** programs (multiple senders → single receiver) to study parallel message transmission and interleaving of outputs.

Design

• **Unidirectional Pipe:** Implements one-way communication where the parent process sends data and the child process receives it.
• **Bidirectional Pipe:** Enables two-way communication between parent and child using two pipes.
• **FIFO (Named Pipe):** Facilitates communication between unrelated processes through a named file created in */tmp*.
• **FIFO with Timestamps:** Uses `gettimeofday()` to record message send and receive times for synchronization and latency analysis.
• **Synchronous Message Passing:** Both sender and receiver block until data transfer occurs, demonstrating controlled and coordinated communication.
• **Asynchronous Message Passing:** The sender continues execution without waiting for the receiver, while the receiver uses non-blocking reads (`O_NONBLOCK`) to fetch messages independently.
• **Concurrent Message Passing:** A single receiver simultaneously handles messages from multiple senders, illustrating parallel communication and message interleaving.

Validate

• Verify correct message flow in both unidirectional and bidirectional pipe programs.
• Confirm proper communication between unrelated processes through FIFO programs.
• Check FIFO timestamps to ensure correct message order and time difference between sending and receiving.
• In **synchronous programs**, observe that the sender waits until the receiver reads the message (blocked behavior).
• In **asynchronous programs**, ensure that the sender operates independently while the receiver reads messages whenever available.
• In **concurrent programs**, verify that the receiver displays intermixed messages from multiple senders, proving parallel communication.
• Test the closing of pipes and FIFOs to understand how IPC mechanisms handle process termination and resource release.

Concept Explanation

1. Pipes

- **Unidirectional:** One-way communication (Parent → Child).
- **Bidirectional:** Two-way communication (Parent ↔ Child) using two pipes.
- **Key points:**
  - Pipes are created using pipe(int fd[2]).
  - fd[0] → read end, fd[1] → write end.
  - Closing unused ends is essential to avoid blocking.

Example Programs

Example 1: Unidirectional Pipe (Parent → Child)

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

char buf1[] = "Message from Parent";
char buf2[80];

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

    if (fork() > 0) //child
  {
        write(fd[1], buf1, strlen(buf1)+1);
        close(fd[1]);
    } else
    {     //parent
        read(fd[0], buf2, sizeof(buf2));
        printf("Child read: %s\n", buf2);
        close(fd[0]);
    }
    return 0;
}
```

Expected Output:

Child read: Message from Parent

2. FIFOs (Named Pipes) Note: Use Two Terminal

- Allow communication between unrelated processes.
- Created using mkfifo(const char *path, mode_t mode).
- Processes open the FIFO using open(), then read/write like a regular file.
- Can include timestamps using gettimeofday() for precise timing of communication.

```
// C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666);
    char arr2[80];
    while (1)
    {
        fd = open(myfifo, O_WRONLY);
        fgets(arr2, 80, stdin);
        write(fd, arr2, strlen(arr2)+1);
        close(fd);
        }
    return 0;
}
```

```c
// C program to implement one side of FIFO
// This side reads first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd1;
    char * myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666);
    char str1[80];
    while (1)
    {
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);
        printf("User1: %s\n", str1);
        close(fd1);
    }
    return 0;
}
```