# 2. LAB ON IPC, MESSAGING, AND PUBLISH/SUBSCRIBE

## 1. Objective of the Lab

To understand and implement:

- **Inter-Process Communication (IPC)**
- **Messaging systems**
- **Publish/Subscribe (Pub/Sub) model**
  used in **cloud-based distributed applications**.

### 2.1 Inter-Process Communication (IPC)

**Inter-Process Communication (IPC)** is a mechanism provided by an **Operating System (OS)** that allows **different processes to communicate with each other and share data**.

---

## Why IPC is Needed

- One process **cannot directly access** another process's data
- IPC enables **coordination, data sharing, and synchronization**

**SIMPLY :** IPC is a mechanism that allows **processes to communicate and share data**

with each other.

### ◆ Common IPC methods:

- **Pipes**
- Message Queues
- Shared Memory
- Sockets

# Lab Part 1: IPC Using Message Queue (Conceptual Steps)

### Step 1: Create two processes

- Process A (Sender)
- Process B (Receiver)

### Step 2: Sender creates a message

- Example: `"Hello from Process A"`

### Step 3: Sender places message in message queue

### Step 4: Receiver reads message from queue

### Step 5: Message is processed and removed

📌 **Result:** Successful IPC using message queues

## Program:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main() {
    int pipefd[2];
    char buffer[50];
    pipe(pipefd);
    if (fork() == 0) {
        close(pipefd[0]);
        char msg[] = "Hello from child";
        write(pipefd[1], msg, strlen(msg) + 1);
        close(pipefd[1]);
    }
else
 {
        close(pipefd[1]);
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);
        close(pipefd[0]);
    }
    return 0;
}
```

Explanation:

# Program Purpose

This program demonstrates **Inter-Process Communication (IPC)** using a **pipe** between a **parent process** and a **child process**.

# Concept Used

✓ **Pipe** – one-way communication channel
✓ **fork()** – creates a child process
✓ **read() / write()** – data transfer

# Step-by-Step Explanation

### 1. Header Files

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

- `stdio.h` → for `printf`
- `unistd.h` → for `pipe()`, `fork()`, `read()`, `write()`
- `string.h` → for `strlen()`

### 2. Variable Declaration

```
int pipefd[2];
char buffer[50];
```

- `pipefd[0]` → read end of pipe
- `pipefd[1]` → write end of pipe
- `buffer` → stores received message

### 3. Create Pipe

```
pipe(pipefd);
```

Creates a pipe with:

- One **read end**
- One **write end**

### 4. Create Child Process

```
if (fork() == 0)
```

- `fork()` creates a new process
- Returns `0` in **child**
- Returns **child PID** in **parent**

# 5. Child Process Code

```
close(pipefd[0]);
char msg[] = "Hello from child";
write(pipefd[1], msg, strlen(msg) + 1);
close(pipefd[1]);
```

🔻 Child:

- Closes **read end**
- Writes message to pipe
- Closes **write end**

# 6. Parent Process Code

```
close(pipefd[1]);
read(pipefd[0], buffer, sizeof(buffer));
printf("Parent received: %s\n", buffer);
close(pipefd[0]);
```

🔻 Parent:

- Closes **write end**
- Reads message from pipe
- Prints received message
- Closes **read end**

# 7. Output

```
Parent received: Hello from child
```

---

# 8. Communication Flow

```
Child Process → Pipe → Parent Process
```

# Key Points for Viva

- Pipe is **unidirectional**
- Used for **related processes**
- `pipefd[0]` = read
- `pipefd[1]` = write
- IPC is achieved after `fork()`

# 2 Messaging in Cloud Computing

- **Messaging allows** asynchronous communication **between applications using** messages**.**

## Producer

A **Producer** is a process or component that **creates data** and **sends it** to a shared resource.

## Consumer

A **Consumer** is a process or component that **receives and uses data** produced by the producer.

🔷 Components:

- **Producer** – sends messages
- **Message Broker** – stores/routes messages
- **Consumer** – receives messages

🔷 Examples:

- RabbitMQ
- Apache Kafka
- AWS SQS
- Azure Service Bus

# Lab Part 2: Messaging Model (Producer–Consumer)

### Step 1: Start Message Broker

Example: RabbitMQ / Kafka (conceptual)

### Step 2: Create Producer

- Sends messages like:
  - Order details
  - User notifications

### Step 3: Broker stores messages

- Ensures reliability

### Step 4: Create Consumer

- Fetches messages asynchronously

### Step 5: Consumer processes messages

- Logs / stores / triggers action

📌 **Result:** Asynchronous messaging achieved

Program:

```
#include <mqueue.h>
#include <stdio.h>
#include <string.h>
int main() {
   mqd_t mq = mq_open("/testmq", O_CREAT | O_RDWR, 0644, NULL);
   char msg[] = "Hello via Message Queue";
   mq_send(mq, msg, strlen(msg)+1, 0);

   char buffer[50];
   mq_receive(mq, buffer, 50, NULL);
   printf("Received: %s\n", buffer);

   mq_close(mq);
   mq_unlink ("/testmq");
   return 0;
}
```

Explanation:

# Program Purpose

This program demonstrates **Inter-Process Communication (IPC)** using a **POSIX Message Queue**, where a message is **sent and received** through a queue.

# Concept Used

✓ POSIX Message Queue

# Meaning of POSIX

**POSIX** stands for:
☞ **Portable Operating System Interface**

It is a set of standards to ensure **compatibility across UNIX/Linux systems**.

---

# Simple Definition

☞ A **POSIX message queue** allows one process to **send messages** and another process to **receive messages** using a **named queue managed by the operating system**.

✓ Asynchronous communication
✓ Kernel-managed message storage

# Header Files

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

- `mqueue.h` → message queue functions
- `stdio.h` → input/output
- `string.h` → string length

# Step-by-Step Explanation

## 1. Create / Open Message Queue

```
mqd_t mq = mq_open("/testmq", O_CREAT | O_RDWR, 0644, NULL);
```

- `/testmq` → queue name (must start with `/`)
- `O_CREAT` → creates queue if not present
- `O_RDWR` → read & write permission
- `0644` → access permissions
- `NULL` → default attributes

📌 Returns **message queue descriptor**

### 2. Create Message

```
char msg[] = "Hello via Message Queue";
```

Message to be sent.

### 3. Send Message

```
mq_send(mq, msg, strlen(msg)+1, 0);
```

- Sends message into queue
- `strlen(msg)+1` includes null character
- `0` → message priority

### 4. Receive Message

```
char buffer[50];
mq_receive(mq, buffer, 50, NULL);
```

- Receives message from queue
- Stores it in `buffer`

### 5. Display Message

```
printf("Received: %s\n", buffer);
```

### 6. Close and Delete Queue

```
mq_close(mq);
mq_unlink("/testmq");
```

- `mq_close()` → closes queue
- `mq_unlink()` → deletes queue from system

# Output

```
Received: Hello via Message Queue
```

## 3 Publish/Subscribe Model

In Pub/Sub:

- Publishers send messages to a **topic**
- Subscribers receive messages from that topic
- Publisher and subscriber are **loosely coupled**

🔻 Examples:

- Google Cloud Pub/Sub
- Apache Kafka
- AWS SNS

# Lab Part 3: Publish / Subscribe Model

### Step 1: Create a Topic

Example: `"cloud-notifications"`

### Step 2: Create Publisher

- Publishes messages to topic

### Step 3: Create Subscribers

- Subscriber 1: Email service
- Subscriber 2: SMS service
- Subscriber 3: Logging service

### Step 4: Publish Message

Example:

```
"Server CPU usage high"
```

### Step 5: All subscribers receive message simultaneously

📌 **Result:** One-to-many communication

# 7. Comparison Table

| Feature | IPC | Messaging | Publish/Subscribe |
|---|---|---|---|
| Communication | Direct | Queue-based | Topic-based |
| Coupling | Tight | Loose | Very loose |
| Scalability | Low | Medium | High |
| Cloud Usage | Low | High | Very High |

**Program:**

## Program Purpose

This program demonstrates the **Publish / Subscribe communication model**

using **ZeroMQ (ZMQ)**, where:

- One **Publisher** sends messages
- Multiple **Subscribers** receive messages
- Communication is **one-to-many**

# Concept Used

✓ Publish / Subscribe Model
✓ Asynchronous Messaging
✓ ZeroMQ Library

# PUBLISHER PROGRAM (publisher.py)

```
import time
import zmq
```

### Step 1: Import Required Modules

- `time` → to pause between messages
- `zmq` → ZeroMQ messaging library (A **ZeroMQ message** is the data that is exchanged between a **sender and receiver** through **ZeroMQ sockets**.)

```
context = zmq.Context()
```

### Step 2: Create ZMQ Context

- Context manages **sockets**
- Required for all ZMQ programs

```
socket = context.socket(zmq.PUB)
```

### Step 3: Create Publisher Socket

- `zmq.PUB` → publisher type socket
- Used to **send messages**

```
socket.bind("tcp://*:5555")
```

### Step 4: Bind Socket to Port

- `*` → accepts connections from any IP
- `5555` → port number
- Publisher **waits for subscribers**

```
while True:
```

### Step 5: Start Infinite Loop

- Sends messages continuously

```
msg = "Hello Subscribers"
print("Publishing:", msg)
```

### Step 6: Create Message

- Message content to be published

```
socket.send_string(msg)
```

### Step 7: Send Message

- Publishes message to **all subscribers**

```
time.sleep(1)
```

### Step 8: Delay

- Sends one message every second

## SUBSCRIBER PROGRAM (subscriber.py)

```
import zmq
```

### Step 1: Import ZMQ Library

```
context = zmq.Context()
```

### Step 2: Create ZMQ Context

---

```
socket = context.socket(zmq.SUB)
```

### Step 3: Create Subscriber Socket

- `zmq.SUB` → subscriber socket
- Used to **receive messages**

---

```
socket.connect("tcp://localhost:5555")
```

### Step 4: Connect to Publisher

- Connects to publisher on same machine
- Uses port `5555`

---

```
socket.subscribe("")
```

### Step 5: Subscribe to Topics

- `""` (empty string) means:
  ✓ Receive **all messages**

- Can be filtered using topic names

---

```
while True:
```

### Step 6: Infinite Loop

- Continuously listens for messages

---

```
message = socket.recv_string()
print("Received:", message)
```

### Step 7: Receive Message

- Receives message from publisher
- Prints it on screen

---

# Output Example

### Publisher Output

```
Publishing: Hello Subscribers
```

### Subscriber Output

```
Received: Hello Subscribers
```

---

# Communication Flow

```
Publisher → Topic → Subscriber(s)
```

---

# Important Points for Viva

- PUB does **not know** how many subscribers exist
- Subscribers must connect **before messages**
- Communication is **asynchronous**
- Used in **cloud, IoT, event-driven systems**

# Real-World Cloud Use Cases

✓ Live notifications
✓ Stock price updates
✓ Chat applications
✓ IoT sensor broadcasting

---

# Conclusion

This program demonstrates **Publish/Subscribe messaging using ZeroMQ**, where one publisher **broadcasts messages** and multiple subscribers **receive them simultaneously**, a key communication model in **cloud computing systems**.

**Publisher (publisher.py)**

```
import time
import zmq
context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5555")

while True:
    msg = "Hello Subscribers"
    print("Publishing:", msg)
    socket.send_string(msg)
    time.sleep(1)
```

**Subscriber (subscriber.py)**

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect("tcp://localhost:5555")
socket.subscribe("")
while True:
    message = socket.recv_string()
    print("Received:", message)
```

# Python code for the Above Programs

## 1. IPC Using Message Queue (Python)

We use **multiprocessing.Queue** for IPC.

Program 1: Sender Process

```
from multiprocessing import Process, Queue

def sender(q):
    q.put("Hello from Process A")

if __name__ == "__main__":
    q = Queue()
    p = Process(target=sender, args=(q,))
    p.start()
    p.join()
```

Program 2: Receiver Process

```
from multiprocessing import Queue

if __name__ == "__main__":
    q = Queue()
    q.put("Hello from Process A")
    message = q.get()
    print("Received:", message)
```

📌 **Output**

```
Received: Hello from Process A
```

---

## 2. Messaging Model (Producer – Consumer)

### Producer Code

```
from queue import Queue
q = Queue()
q.put("Order Placed")
print("Producer sent message")
```

### Consumer Code

```
from queue import Queue
q = Queue()
q.put("Order Placed")
msg = q.get()
print("Consumer received:", msg)
```

📌 **Output**

```
Producer sent message
Consumer received: Order Placed
```

---

## 3. Publish / Subscribe Model (Simple Simulation)

### Publisher

```
def publish(topic, message, subscribers):
    print("Publishing:", message)
    for sub in subscribers:
        sub(topic, message)
```

### Subscribers

```
def email_service(topic, message):
    print("Email received:", message)

def sms_service(topic, message):
```

```
    print("SMS received:", message)

subscribers = [email_service, sms_service]

publish("alerts", "Server CPU High",
subscribers)
```

📌 **Output**

```
Publishing: Server CPU High
Email received: Server CPU High
SMS received: Server CPU High
```

---

## 4. Lab Observation

- IPC uses **direct queue**
- Messaging uses **producer → queue → consumer**
- Pub/Sub sends **one message to many subscribers**

---

## 5. Lab Conclusion

Simple Python programs demonstrate **IPC, Messaging, and Publish/Subscribe**, which form the backbone of **cloud-based microservices communication**.