Larbi Ben M'hidi University – Oum El Bouaghi

Faculty of Exact Sciences and Nature and Life Sciences

Department of Mathematics and Computer Science

Module: Operating Systems II

Lecturer: Dr. Moustafa Sadek KAHIL

2025 – 2026

Chapter III: Inter-Process Communication (IPC)

3.1 Introduction to Process Cooperation

In modern operating systems, multiple processes execute simultaneously. These processes can be:

- Independent: Execution neither affects nor is affected by other processes
- Cooperative: Processes work together to accomplish common goals

Why Do Processes Need to Cooperate?

- **Information Sharing:** Processes need to exchange data and information
- **Computational Speedup:** Parallel processing to reduce execution time
- **Modularity:** Dividing complex tasks among specialized processes
- **Resource Sharing:** Efficient use of hardware resources (files, devices, memory)
- **Distributed Computing:** Coordination across networked systems

Real-World Examples:

1. Example 1: Web Browser Architecture

   - Main Process (UI handling)

   - Renderer Process (web page display)

   - Network Process (downloads)

   - Plugin Process (Flash, PDF viewer)

   → All these processes communicate via IPC

2. Example 2: Database Management System

   - Client Handler Process (user queries)

- Query Optimizer Process

- Disk Manager Process (file I/O)

- Transaction Manager Process

→ Coordinate to process database transactions

3. Example 3: Compiler Pipeline
   - Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Code Generator
   → Each stage communicates with the next via IPC

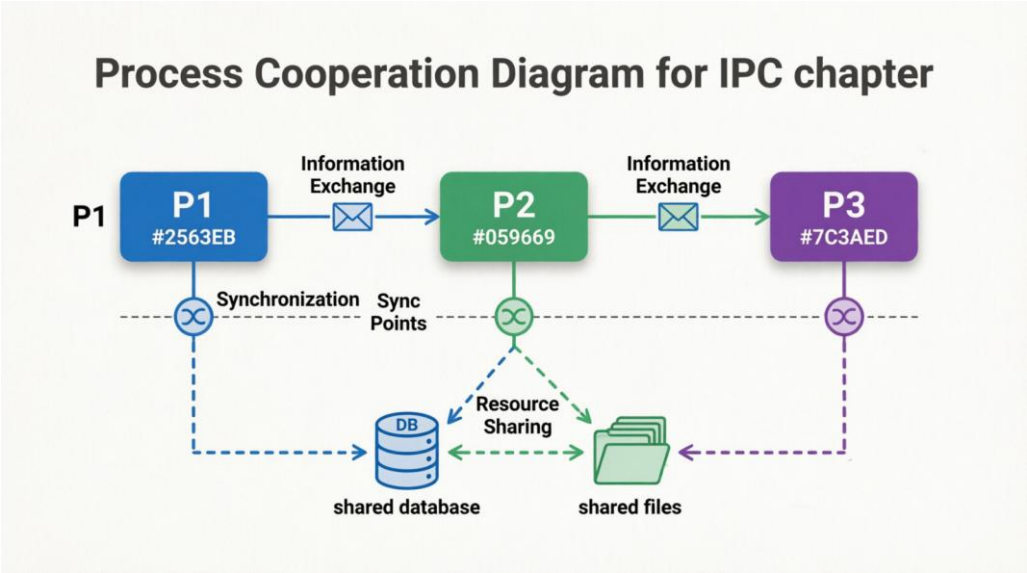

Figure 3.1: Illustration of processes P1, P2, P3 exchanging information, synchronizing at sync points, and sharing resources (database and files)

3.2 Shared Memory Communication

3.2.1 Concept and Principles

Shared memory is the fastest IPC mechanism where multiple processes access the same memory region. The operating system creates a memory segment that processes can attach to their address space.

How It Works:

1. OS creates a shared memory segment
2. Process A attaches the segment to its address space
3. Process B attaches the same segment
4. Both processes can now read/write to the same memory

5. Changes by one process are immediately visible to others

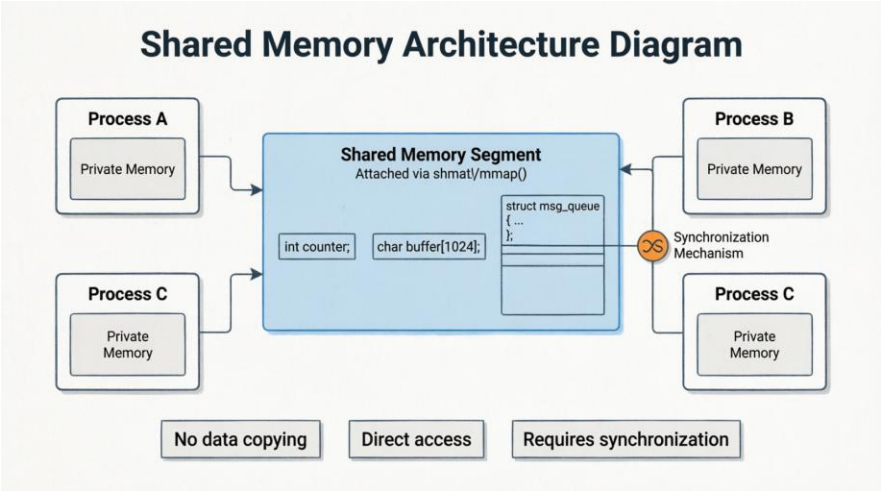

Figure 3.2: Multiple processes (A, B, C) with private memory, all connected to a central shared memory segment with synchronization mechanism

3.2.2 Implementation APIs

POSIX Shared Memory:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

// Create/open shared memory object
int shm_fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);

// Configure size
ftruncate(shm_fd, SIZE);

// Map to address space
char *ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

// Use shared memory
sprintf(ptr, "Hello from Process A");

// Cleanup
munmap(ptr, SIZE);
shm_unlink("/myshm");
```

System V Shared Memory:

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
// Create shared memory segment
int shmid = shmget(KEY, SIZE, IPC_CREAT | 0666);

// Attach to process address space
char *shm_ptr = shmat(shmid, NULL, 0);

// Use shared memory
strcpy(shm_ptr, "Hello from Process B");

// Detach
shmdt(shm_ptr);

// Remove segment
shmctl(shmid, IPC_RMID, NULL);
```

Producer-Consumer Problem

The Producer-Consumer problem is a fundamental synchronization challenge where:

- Producer: Generates data items
- Consumer: Processes data items
- Buffer: Finite-size storage between them

Problem Requirements:

1. Producer cannot add to full buffer
2. Consumer cannot remove from empty buffer
3. Only one process accesses buffer at a time (mutual exclusion)

Shared Data Structures:

```
#define BUFFER_SIZE 10

typedef struct {
   int data[BUFFER_SIZE];
   int in;   // Next write position
   int out;  // Next read position
   int count; // Number of items in buffer
} shared_buffer_t;

// Shared buffer
shared_buffer_t *buffer;
```

Synchronization Primitives:

```
semaphore mutex = 1;         // Mutual exclusion
semaphore empty = BUFFER_SIZE;  // Count of empty slots
semaphore full = 0;          // Count of filled slots
```

Producer Algorithm:

```
void producer() {
   while (TRUE) {
      int item = produce_item();

      sem_wait(&empty);    // Wait for empty slot
      sem_wait(&mutex);    // Enter critical section

      // Add item to buffer
      buffer->data[buffer->in] = item;
      buffer->in = (buffer->in + 1) % BUFFER_SIZE;
      buffer->count++;

      sem_post(&mutex);    // Exit critical section
      sem_post(&full);     // Signal: new item available
   }
}
```

Consumer Algorithm:

```
void consumer() {
   while (TRUE) {
      sem_wait(&full);     // Wait for available item
      sem_wait(&mutex);    // Enter critical section

      // Remove item from buffer
      int item = buffer->data[buffer->out];
      buffer->out = (buffer->out + 1) % BUFFER_SIZE;
      buffer->count--;

      sem_post(&mutex);    // Exit critical section
      sem_post(&empty);    // Signal: empty slot available

      consume_item(item);
   }
}
```

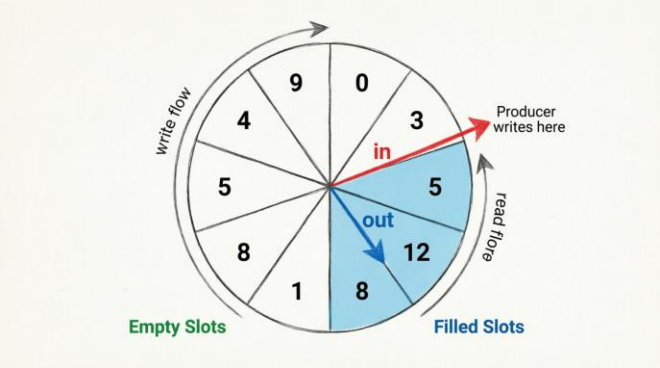

Figure 3.3: Circular ring showing filled and empty slots with producer (in) and consumer (out) pointers

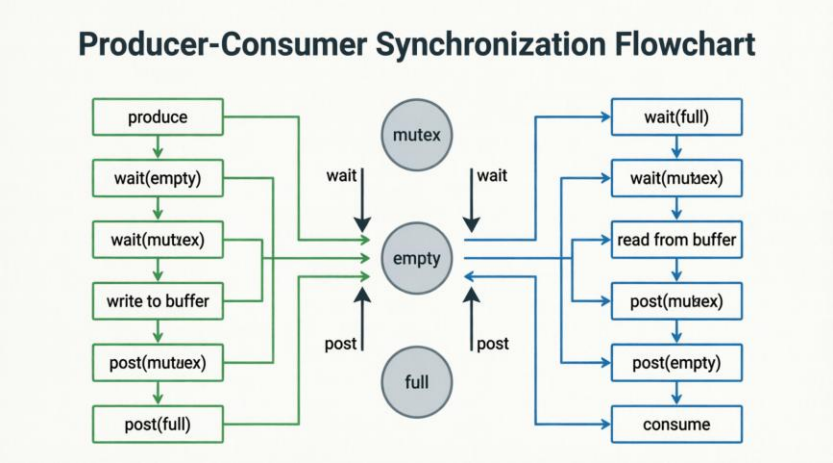

Figure 3.4: Parallel flowcharts for producer and consumer showing wait/post operations on three semaphores (mutex, empty, full)

3.2.4 Readers-Writers Problem

Problem Statement: Multiple processes need access to a shared resource (database, file, etc.):

- Readers: Only read data (can run concurrently)
- Writers: Modify data (need exclusive access)

Constraints:

1. Multiple readers can read simultaneously
2. Writers require exclusive access
3. No reader/writer access while writer is writing

Synchronization Challenge:

- Reader Priority: Can cause writer starvation
- Writer Priority: Can cause reader starvation

Implementation (Reader Priority):

```
int readcount = 0;
semaphore mutex = 1;     // Protect readcount
semaphore db = 1;        // Database access

// Reader Process
void reader() {
   while (TRUE) {
      sem_wait(&mutex);
      readcount++;
      if (readcount == 1)
         sem_wait(&db);  // First reader locks database
      sem_post(&mutex);

      // Reading is performed
      read_database();

      sem_wait(&mutex);
      readcount--;
      if (readcount == 0)
         sem_post(&db);  // Last reader unlocks database
      sem_post(&mutex);
   }
}

// Writer Process
void writer() {
   while (TRUE) {
      sem_wait(&db);     // Exclusive access

      // Writing is performed
      write_database();

      sem_post(&db);
   }
}
```

Figure 3.5: Three readers (R1, R2, R3) with concurrent read access, one writer (W1) requiring exclusive access, with mutex lock and readcount variable

3.3 Message Passing via Mailboxes/Ports

3.3.1 Concept and Architecture

Mailboxes (also called ports) provide indirect communication where processes send and receive messages through an intermediate queue managed by the operating system.

Key Characteristics:

- Decoupling: Sender and receiver don't need to know each other
- Asynchronous: Sender doesn't wait for receiver
- FIFO Ordering: Messages processed in arrival order
- OS-Managed: Operating system handles synchronization

Mailbox Components:

1. Unique Identifier (mailbox ID/name)
2. Message Queue (FIFO buffer)
3. Capacity Limit (maximum messages)
4. Access Control (permissions)

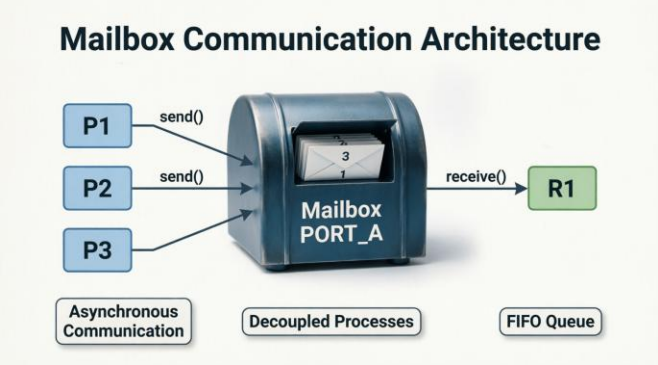

Figure 3.6: Central mailbox (PORT_A) with multiple sender processes (P1, P2, P3) sending messages and one receiver (R1) receiving, showing asynchronous, decoupled, FIFO communication

3.4.2 Mailbox Operations

Basic Operations:

```
// Send operation
status send(mailbox_id, message) {
   if (mailbox is full) {
      if (blocking_mode)
         wait_until_space_available();
      else
         return ERROR_BUFFER_FULL;
   }
```

```
    copy_message_to_mailbox(message);
    signal_waiting_receivers();
    return SUCCESS;
}

// Receive operation
status receive(mailbox_id, buffer) {
    if (mailbox is empty) {
        if (blocking_mode)
            wait_until_message_available();
        else
            return ERROR_NO_MESSAGE;
    }

    copy_message_to_buffer(buffer);
    signal_waiting_senders();
    return SUCCESS;
}
```

Advanced Operations:

```
// Non-blocking send
status try_send(mailbox_id, message) {
    if (mailbox is full)
        return ERROR_TRY_AGAIN;
    return send(mailbox_id, message);
}

// Receive with timeout
status receive_timeout(mailbox_id, buffer, timeout_ms) {
    if (mailbox is empty) {
        wait_with_timeout(timeout_ms);
        if (timeout_expired)
            return ERROR_TIMEOUT;
    }
    return receive(mailbox_id, buffer);
}

// Peek at next message without removing
status peek(mailbox_id, buffer) {
    if (mailbox is empty)
        return ERROR_NO_MESSAGE;

    copy_first_message_without_removal(buffer);
    return SUCCESS;
}
```

3.3.3 Communication Patterns

1. **One-to-One:** Producer → Mailbox → Consumer (Simple pipeline)
2. **Many-to-One (Server Pattern):** Client1, Client2, Client3 → Mailbox → Server (Request-response server)
3. **One-to-Many (Broadcast):** Sender → Mailbox1 → Receiver1, Mailbox2 → Receiver2 (Publish-subscribe)
4. **Many-to-Many:** Producer1,2,3 → Mailbox → Consumer1,2,3 (Work distribution)

3.4.4 Example: Temperature Monitoring System

Scenario: Multiple sensors send temperature readings to a monitoring system.

```
// Message structure
typedef struct {
   int sensor_id;
   float temperature;
   timestamp_t time;
} temperature_msg_t;

// Create mailbox for temperature readings
mailbox_id_t temp_mailbox = mailbox_create("TEMP_MONITOR", 100);

// Sensor Process (Multiple instances)
void temperature_sensor(int sensor_id) {
   while (TRUE) {
      temperature_msg_t msg;
      msg.sensor_id = sensor_id;
      msg.temperature = read_sensor();
      msg.time = get_timestamp();

      send(temp_mailbox, msg);

      sleep(5); // Read every 5 seconds
   }
}

// Monitoring Process
void monitoring_system() {
   while (TRUE) {
      temperature_msg_t msg;
      receive(temp_mailbox, &msg);

      printf("Sensor %d: %.2f°C at %s\n",
           msg.sensor_id, msg.temperature,
           format_time(msg.time));

      if (msg.temperature > 40.0) {
         alert_high_temperature(msg);
      }
```

```
    }
}
```

3.4 Pipes

Pipes represent one of the oldest and most fundamental forms of Inter-Process Communication (IPC) in Unix-like operating systems. First introduced in Unix Version 3 (1973), pipes provide a unidirectional data channel that enables communication between related or unrelated processes through a producer-consumer model.

3.4.1 Anonymous Pipes (Unnamed Pipes)

Anonymous pipes are temporary, kernel-resident data buffers that exist only in memory and facilitate unidirectional communication between processes with a familial relationship (typically parent-child).

Key Properties:

- **Lifetime**: Exists only during the lifetime of the creating process and its descendants
- **Scope**: Limited to related processes (parent-child, siblings)
- **Direction**: Unidirectional (half-duplex) - data flows in one direction
- **Persistence**: No filesystem entry; exists only in kernel space
- **Capacity**: Typically 64KB buffer (system-dependent, configurable via /proc/sys/fs/pipe-buffer-size)

System Call Interface

```
#include <unistd.h>

int pipe(int fd[2]);
```

Parameters:

- fd[0]: File descriptor for reading (read end)
- fd[1]: File descriptor for writing (write end)

Return Value:

- Returns 0 on success
- Returns -1 on error (sets errno)

Operational Semantics

The pipe system call creates a kernel buffer and returns two file descriptors. The typical usage pattern involves:

1. **Creation**: Parent process calls pipe(fd)
2. **Fork**: Parent calls fork() to create child process
3. **Descriptor Management**:
    - Parent closes fd[0] (read end) - write-only
    - Child closes fd[1] (write end) - read-only
4. **Communication**: Data written to fd[1] can be read from fd[0]

Synchronization Properties:

- **Blocking I/O**: Read operations block when pipe is empty; write operations block when pipe is full
- **EOF Signaling**: When all write ends are closed, read returns 0 (end-of-file)
- **SIGPIPE Signal**: Writing to a pipe with no readers generates SIGPIPE (default: terminate process)

Example Implementation

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

int main() {
   int fd[2];
   char buffer[100];
   pid_t pid;

   pipe(fd);
   pid = fork();

   if (pid > 0) {  // Parent
      close(fd[0]);  // Close read end
      write(fd[1], "Hello Child!", 13);
      close(fd[1]);
   } else {  // Child
      close(fd[1]);  // Close write end
      read(fd[0], buffer, sizeof(buffer));
      printf("Received: %s\n", buffer);
      close(fd[0]);
   }
   return 0;
}
```

3.4.2 Named Pipes (FIFOs)

Named pipes, also known as FIFOs (First-In-First-Out), are special files in the filesystem that provide pipe-like communication between unrelated processes. Unlike anonymous pipes, they persist in the filesystem and can be accessed by any process with appropriate permissions.

Key Properties:

- **Lifetime**: Persists until explicitly removed (like regular files)
- **Scope**: Any processes on the same system (unrelated processes)
- **Direction**: Unidirectional (typically), though can be opened for both read and write
- **Persistence**: Filesystem entry with permissions (owner, group, mode)
- **Access Control**: Standard Unix file permissions apply

Creation Methods

Method 1: Command Line

```
mkfifo /tmp/mypipe
# or
mknod /tmp/mypipe p
```

Method 2: System Call

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Parameters:

- pathname: Filesystem path for the FIFO
- mode: Permission bits (e.g., 0666 for read/write for all)

Operational Semantics

Named pipes follow FIFO (First-In-First-Out) ordering and exhibit blocking behavior:

Opening Semantics:

- **Read-only open (O_RDONLY)**: Blocks until a writer opens the FIFO
- **Write-only open (O_WRONLY)**: Blocks until a reader opens the FIFO
- **Read-write open (O_RDWR)**: Opens immediately without blocking

- **Non-blocking mode (O_NONBLOCK)**: Open returns immediately

Data Transfer:

- Writes up to PIPE_BUF bytes (typically 4096) are atomic
- Larger writes may interleave with other writers
- Data is preserved in order of writing

Example Implementation

Writer Process:

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

int main() {
   int fd = open("/tmp/mypipe", O_WRONLY);
   write(fd, "Hello via FIFO!", 16);
   close(fd);
   return 0;
}
```

Reader Process:

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

int main() {
   char buffer[100];
   int fd = open("/tmp/mypipe", O_RDONLY);
   read(fd, buffer, sizeof(buffer));
   printf("Received: %s\n", buffer);
   close(fd);
   return 0;
}
```

3.5 Socket Communication

3.5.1 Introduction to Sockets

Sockets are the most versatile IPC mechanism, enabling communication between:

- Processes on the same machine (UNIX domain sockets)
- Processes on different machines (network sockets)
- Processes across the Internet

Socket Definition:

A socket is an endpoint for communication, identified by:

5. IP Address: Machine identifier
6. Port Number: Process identifier (0-65535)
7. Protocol: TCP or UDP

Socket Analogy:

Think of a socket like a telephone:

- IP Address = Phone number of building
- Port = Extension number
- Protocol = Language spoken
- Socket connection = Phone call in progress

3.6.2 Socket Types

1. TCP Sockets (SOCK_STREAM)

- Reliable: Guaranteed delivery
- Ordered: Data arrives in sequence
- Connection-Oriented: Establish connection before data transfer
- Flow Control: Prevents overwhelming receiver
- Higher Latency: Connection setup overhead

Use Cases:

- Web browsing (HTTP/HTTPS), File transfer (FTP), Email (SMTP, IMAP), Remote access (SSH)

2. UDP Sockets (SOCK_DGRAM)

- Fast: No connection setup
- Low Overhead: Minimal headers
- Unreliable: Packets may be lost
- Unordered: May arrive out of sequence
- No Flow Control: Can overwhelm receiver

Use Cases:

- Video streaming, Online gaming, DNS queries, VoIP calls, IoT sensor data

3. UNIX Domain Sockets

- Fastest: No network overhead
- Secure: Local machine only
- File-based: Identified by filesystem path
- Full-duplex: Bidirectional communication

Use Cases:

- Database connections (PostgreSQL, MySQL), Web server to application server (Nginx ↔ PHP-FPM), Docker daemon communication, System services

3.5.3 TCP Socket Programming

Server-Side Lifecycle:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int create_server_socket(int port) {
    int server_fd;
    struct sockaddr_in address;

    // Step 1: Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Step 2: Configure socket options
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    // Step 3: Bind to address and port
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;  // Listen on all interfaces
    address.sin_port = htons(port);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }

    // Step 4: Listen for connections
    if (listen(server_fd, 5) < 0) {  // Queue up to 5 connections
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d\n", port);
    return server_fd;
}

int accept_client(int server_fd) {
    struct sockaddr_in client_addr;
```

```
    socklen_t client_len = sizeof(client_addr);

    // Step 5: Accept incoming connection
    int client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &client_len);
    if (client_fd < 0) {
        perror("Accept failed");
        exit(EXIT_FAILURE);
    }

    printf("Client connected: %s:%d\n",
           inet_ntoa(client_addr.sin_addr),
           ntohs(client_addr.sin_port));

    return client_fd;
}
```

Client-Side Connection:

```
int connect_to_server(const char *ip, int port) {
    int sock;
    struct sockaddr_in server_addr;

    // Step 1: Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Step 2: Configure server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);

    // Convert IP string to binary
    if (inet_pton(AF_INET, ip, &server_addr.sin_addr) <= 0) {
        perror("Invalid address");
        exit(EXIT_FAILURE);
    }

    // Step 3: Connect to server
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection failed");
        exit(EXIT_FAILURE);
    }

    printf("Connected to server %s:%d\n", ip, port);
    return sock;
}
```

Data Transfer:

```
// Send data
ssize_t send_data(int sock, const char *data, size_t len) {
    ssize_t total_sent = 0;

    while (total_sent < len) {
        ssize_t sent = send(sock, data + total_sent, len - total_sent, 0);
        if (sent < 0) {
            perror("Send failed");
            return -1;
        }
        total_sent += sent;
    }

    return total_sent;
}

// Receive data
ssize_t receive_data(int sock, char *buffer, size_t len) {
    ssize_t total_received = 0;

    while (total_received < len) {
        ssize_t received = recv(sock, buffer + total_received, len - total_received, 0);
        if (received < 0) {
            perror("Receive failed");
            return -1;
        }
        if (received == 0) {
            // Connection closed by peer
            break;
        }
        total_received += received;
    }

    return total_received;
}
```

Complete Server Example:

```
void run_echo_server(int port) {
    int server_fd, client_fd;
    char buffer[1024];

    server_fd = create_server_socket(port);

    while (1) {
        printf("Waiting for connection...\n");
        client_fd = accept_client(server_fd);
```

```
        // Handle client
        while (1) {
            ssize_t bytes_read = recv(client_fd, buffer, sizeof(buffer), 0);
            if (bytes_read <= 0) {
                printf("Client disconnected\n");
                break;
            }

            buffer[bytes_read] = '\0';
            printf("Received: %s\n", buffer);

            // Echo back
            send(client_fd, buffer, bytes_read, 0);
        }

        close(client_fd);
    }

    close(server_fd);
}
```

Complete Client Example:

```
void run_echo_client(const char *ip, int port) {
    int sock;
    char buffer[1024];

    sock = connect_to_server(ip, port);

    // Send messages
    const char *messages[] = {"Hello", "World", "Echo", "Test"};

    for (int i = 0; i < 4; i++) {
        printf("Sending: %s\n", messages[i]);
        send(sock, messages[i], strlen(messages[i]), 0);

        // Receive response
        ssize_t bytes_read = recv(sock, buffer, sizeof(buffer), 0);
        buffer[bytes_read] = '\0';
        printf("Echo: %s\n\n", buffer);
    }

    close(sock);
}
```

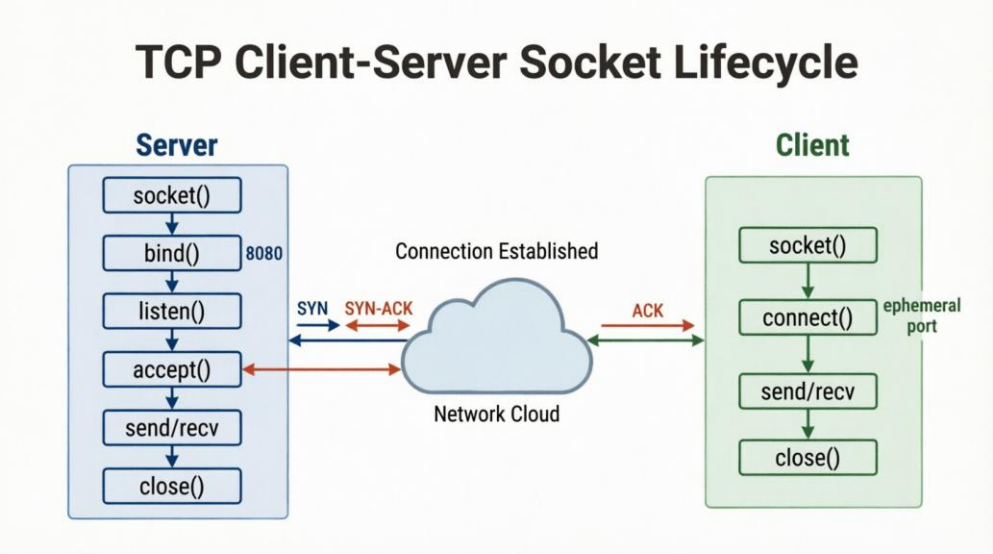

Figure 3.7: Server flow (socket→bind→listen→accept→send/recv→close) and Client flow (socket→connect→send/recv→close) with TCP handshake (SYN, SYN-ACK, ACK) in the middle

3.5.4 UDP Socket Programming

UDP Server:

```
void run_udp_server(int port) {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    char buffer[1024];

    // Create UDP socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    // Bind
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port);

    bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr));

    printf("UDP Server listening on port %d\n", port);

    // No listen/accept - just receive
    while (1) {
        socklen_t client_len = sizeof(client_addr);
        ssize_t len = recvfrom(sockfd, buffer, sizeof(buffer), 0,
                               (struct sockaddr *)&client_addr, &client_len);
```

```
        buffer[len] = '\0';
        printf("Received from %s:%d: %s\n",
               inet_ntoa(client_addr.sin_addr),
               ntohs(client_addr.sin_port),
               buffer);

        // Send response
        const char *response = "ACK";
        sendto(sockfd, response, strlen(response), 0,
               (struct sockaddr *)&client_addr, client_len);
    }

    close(sockfd);
}
```

UDP Client:

```
void run_udp_client(const char *ip, int port) {
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[1024];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    inet_pton(AF_INET, ip, &server_addr.sin_addr);

    // Send without connection
    const char *msg = "Hello UDP";
    sendto(sockfd, msg, strlen(msg), 0,
           (struct sockaddr *)&server_addr, sizeof(server_addr));

    // Receive response
    socklen_t server_len = sizeof(server_addr);
    ssize_t len = recvfrom(sockfd, buffer, sizeof(buffer), 0,
                           (struct sockaddr *)&server_addr, &server_len);

    buffer[len] = '\0';
    printf("Response: %s\n", buffer);

    close(sockfd);
}
```

3.5.5 UNIX Domain Sockets

```
void run_unix_socket_server(const char *socket_path) {
    int server_fd, client_fd;
```

```
    struct sockaddr_un addr;

    // Remove existing socket file
    unlink(socket_path);

    // Create socket
    server_fd = socket(AF_UNIX, SOCK_STREAM, 0);

    // Configure address
    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, socket_path, sizeof(addr.sun_path) - 1);

    // Bind
    bind(server_fd, (struct sockaddr *)&addr, sizeof(addr));

    // Listen
    listen(server_fd, 5);

    printf("UNIX socket server listening on %s\n", socket_path);

    // Accept and handle
    client_fd = accept(server_fd, NULL, NULL);

    // Communication...
    close(client_fd);
    close(server_fd);
    unlink(socket_path);
}
```