1. Introduction to LINUX environment and related system programming

1. Introduction to LINUX

LINUX is a powerful, open-source operating system widely used for development, servers, and embedded systems. It is based on UNIX principles, offering a robust and multi-user environment.

Features of LINUX:

1. **Open Source**: The source code is freely available.
2. **Multitasking and Multiuser**: Supports multiple users and tasks simultaneously.
3. **Security**: Strong file permission and encryption mechanisms.
4. **Portability**: Runs on various hardware platforms.
5. **Shell Interface**: CLI-based interaction for efficient command execution.

2. LINUX Environment Components

1. **Kernel**: The core of the operating system that manages hardware and system calls.
2. **Shell**: A command-line interface (CLI) for interacting with the kernel.
3. **File System**: Hierarchical directory structure starting from the root (`/`).
4. **Process Management**: Handles creation, execution, and termination of processes.
5. **System Utilities**: Tools like `ls`, `cp`, `chmod`, and `top` for system management.

3. System Programming in LINUX

What is System Programming?

System programming involves writing code that interacts directly with the operating system, focusing on system-level tasks like process control, file handling, inter-process communication (IPC), and networking.

Key Aspects of System Programming:

1. **System Calls**: Functions provided by the OS kernel to interact with hardware and manage system resources.
2. **Libraries**: Standard libraries like `glibc` for efficient programming.
3. **Concurrency**: Multithreading and synchronization.

4. Important Topics in LINUX System Programming

a. File Management

System calls for file operations:

- `open()`, `close()`: Open/close files.
- `read()`, `write()`: Read/write data.
- `lseek()`: Move the file pointer.

Example:

```
c
Copy code
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd = open("example.txt", O_CREAT | O_WRONLY, 0644); // Create file
    if (fd < 0) {
        perror("open");
        return 1;
    }

    write(fd, "Hello, LINUX!", 13); // Write to file
    close(fd); // Close file
    return 0;
}
```

b. Process Management

System calls for process control:

- `fork()`: Create a new process.
- `exec()`: Replace the current process image.
- `wait()`: Wait for child processes.

Example:

```
c
Copy code
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child Process\n");
        execl("/bin/ls", "ls", NULL); // Execute "ls" command
    } else {
        wait(NULL); // Wait for child to finish
        printf("Parent Process\n");
    }

    return 0;
```

```
}
```

c. Signals

Signals are software interrupts to notify processes of events.

- `kill()`: Send a signal.
- Signal handlers: Custom functions to handle signals.

Example:

```
c
Copy code
#include <signal.h>
#include <stdio.h>

void handle_signal(int sig) {
    printf("Caught signal %d\n", sig);
}

int main() {
    signal(SIGINT, handle_signal); // Handle Ctrl+C
    while (1) {
        printf("Running...\n");
        sleep(1);
    }
    return 0;
}
```

d. Inter-Process Communication (IPC)

Mechanisms for communication between processes:

1. **Pipes**:
   - Anonymous pipes: Data flow between parent and child processes.
   - Named pipes (FIFOs): Data flow between unrelated processes.
2. **Shared Memory**:
   - Fastest IPC mechanism.
   - Shared regions of memory between processes.
3. **Message Queues**:
   - Messages sent and received between processes.
4. **Semaphores**:
   - Synchronization primitive.

Example of a Pipe:

```
c
Copy code
#include <unistd.h>
#include <stdio.h>
```

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

    if (fork() == 0) {
        close(fd[0]); // Close read end in child
        write(fd[1], "Hello, Parent!", 14); // Write to pipe
        close(fd[1]);
    } else {
        char buffer[20];
        close(fd[1]); // Close write end in parent
        read(fd[0], buffer, 20); // Read from pipe
        printf("Parent Received: %s\n", buffer);
        close(fd[0]);
    }

    return 0;
}
```

e. Networking

Programming for network communication:

- Sockets: Communication endpoints for networking.
- System calls: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`.

Example of a TCP Server:

```c
c
Copy code
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in address;
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);

    int client_fd = accept(server_fd, NULL, NULL);
    char buffer[1024] = {0};
    read(client_fd, buffer, 1024);
    printf("Received: %s\n", buffer);

    write(client_fd, "Hello, Client!", 14);
    close(client_fd);
```

```
    close(server_fd);

    return 0;
}
```

5. Tools for System Programming

1. **Editors**: `vim`, `nano`, or IDEs like VS Code.
2. **Compilers**: `gcc` or `clang`.
3. **Debuggers**: `gdb` for debugging system-level programs.
4. **Profilers**: `strace`, `ltrace`, and `perf` for performance analysis.

6. Key Benefits of LINUX System Programming

1. High control over hardware.
2. Efficient use of resources.
3. Broad applicability in networking, device drivers, and embedded systems.

2. Introduction to Various Networking Equipment

1. **Router**
    - **Purpose**: Connects multiple networks, directing data packets between them. Essential for connecting a local network to the internet.
    - **Common Configurations**: Setting IP addresses, enabling DHCP, configuring NAT, and setting up security (firewalls).
2. **Switch**
    - **Purpose**: Connects devices within a local network (LAN). Operates at Layer 2 (Data Link Layer) of the OSI model.
    - **Common Configurations**: VLAN setup, port security, spanning tree protocol (STP) configurations.
3. **Hub**
    - **Purpose**: Connects devices in a LAN but does not manage traffic like a switch. Operates at Layer 1 (Physical Layer).
    - **Configuration**: No configuration needed; it simply broadcasts data to all connected devices.
4. **Access Point (AP)**
    - **Purpose**: Extends wireless connectivity to devices in a network. Operates on Wi-Fi standards (802.11).
    - **Common Configurations**: SSID setup, security protocols (WPA3, WPA2), and channel selection.
5. **Modem**
    - **Purpose**: Converts signals between digital and analog for internet access.
    - **Configuration**: Depends on ISP settings; includes VLAN tagging, PPPoE settings, or DHCP.
6. **Firewall**
    - **Purpose**: Protects a network by controlling inbound and outbound traffic based on security rules.
    - **Common Configurations**: Rule creation, intrusion detection/prevention setup, and VPN configurations.
7. **Network Interface Card (NIC)**
    - **Purpose**: Allows a device to connect to a network. Comes in wired and wireless variants.
    - **Configuration**: Setting IP address, subnet mask, and gateway manually or using DHCP.
8. **Cable Types**
    - **Twisted Pair (Cat5e, Cat6)**: Common for Ethernet connections.
    - **Coaxial**: Used for cable internet.
    - **Fiber Optic**: High-speed, long-distance communication.

Configuration of a Computer Network

1. **Planning the Network**:
   - Determine network requirements (number of devices, type of connection, bandwidth needs).
   - Define IP addressing scheme (use private IPs, define subnet masks, and gateways).
2. **Configuring a Router**:
   - Access router's admin interface via a web browser or CLI.
   - Set up WAN (PPPoE, DHCP, or Static IP) and LAN settings.
   - Configure NAT for internet access.
   - Enable firewall and QoS if required.
3. **Configuring a Switch**:
   - Assign management IP address for remote access.
   - Set up VLANs for segmentation.
   - Enable STP to prevent loops.
   - Configure port mirroring for traffic analysis if needed.
4. **Setting up Wireless Access Points**:
   - Access AP via its management interface.
   - Configure SSID and encryption (WPA2/WPA3).
   - Set up DHCP if required or rely on router's DHCP service.
5. **Connecting Devices**:
   - Use proper cabling (Ethernet or fiber).
   - Assign IP addresses manually or enable DHCP for automatic configuration.
   - Ensure devices can communicate by pinging the gateway or another device on the network.
6. **Testing the Network**:
   - Test connectivity with ping, tracert, or other network tools.
   - Verify internet access and check for latency or packet loss.
7. **Monitoring and Maintenance**:
   - Use tools like Wireshark, NetFlow, or SNMP for real-time monitoring.
   - Regularly update firmware and check security settings.

3. Introduction to pipes and related system calls for pipe management

1. Understand the Concept

A **pipe** is a unidirectional communication channel:

- Data written to the pipe by one process can be read by another.
- A pipe can be created using the pipe() system call in Linux.

2. Use the pipe() System Call

- The pipe() system call creates a pipe.
- It returns two file descriptors:
  - **fd[0]**: Read end of the pipe.
  - **fd[1]**: Write end of the pipe.

3. Fork a Child Process

- Use the fork() system call to create a child process.
- Parent and child processes can communicate through the pipe.

4. Close Unused Ends

- In the parent process, close the read end of the pipe if it's only writing.
- In the child process, close the write end of the pipe if it's only reading.

5. Write and Read Data

- The parent process writes data to the pipe.
- The child process reads the data from the pipe.

6. Code Implementation

Below is an example in C:

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

int main() {
    int fd[2]; // File descriptors for the pipe
    pid_t pid;
    char write_msg[] = "Hello from parent!";
    char read_msg[100];

    // Step 2: Create the pipe
```

```
    if (pipe(fd) == -1) {
        perror("Pipe failed");
        return 1;
    }

    // Step 3: Fork a child process
    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    }

    if (pid > 0) { // Parent process
        // Step 4: Close unused read end
        close(fd[0]);

        // Step 5: Write to the pipe
        write(fd[1], write_msg, strlen(write_msg) + 1);
        close(fd[1]); // Close write end after writing

    } else { // Child process
        // Step 4: Close unused write end
        close(fd[1]);

        // Step 5: Read from the pipe
        read(fd[0], read_msg, sizeof(read_msg));
        printf("Child received: %s\n", read_msg);
        close(fd[0]); // Close read end after reading
    }

    return 0;
}
```

7. Explanation

1. **pipe(fd)**: Creates a pipe with fd[0] for reading and fd[1] for writing.
2. **fork()**: Creates a child process.
3. **Parent Process**:
    - Closes the read end (fd[0]).
    - Writes data to the write end (fd[1]).
4. **Child Process**:
    - Closes the write end (fd[1]).
    - Reads data from the read end (fd[0]).
5. Communication is complete, and ends are closed.

8. Compile and Run

bash
Copy code
gcc -o pipe_example pipe_example.c
./pipe_example

Expected Output

bash
Copy code
Child received: Hello from parent!

---

1. Include Necessary Libraries

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

- `stdio.h`: For input/output functions like `printf`.
- `unistd.h`: Provides system calls like `pipe`, `fork`, `read`, and `write`.
- `string.h`: For string manipulation functions like `strlen`.

---

2. Declare Pipe and Variables

```
int fd[2]; // File descriptors for the pipe
pid_t pid;
char write_msg[] = "Hello from parent!";
char read_msg[100];
```

- `fd[2]`: Array to hold file descriptors. `fd[0]` is the read end, and `fd[1]` is the write end of the pipe.
- `pid`: Stores the process ID returned by `fork`.
- `write_msg`: The message the parent will send to the child.
- `read_msg`: A buffer for the child to store the message read from the pipe.

---

3. Create a Pipe

```
if (pipe(fd) == -1) {
    perror("Pipe failed");
```

```
    return 1;
}
```

- The `pipe` system call creates a unidirectional communication channel and assigns file descriptors for reading and writing.
- If `pipe` returns `-1`, an error occurred, and the program exits with an error message using `perror`.

---

4. Fork a New Process

```
pid = fork();
if (pid < 0) {
    perror("Fork failed");
    return 1;
}
```

- The `fork` system call creates a child process.
- `pid`:
  - `< 0`: Fork failed.
  - `> 0`: Parent process (the returned `pid` is the child's PID).
  - `== 0`: Child process (returned PID is 0).

---

5. Parent Process

```
if (pid > 0) { // Parent process
    close(fd[0]); // Close unused read end
    write(fd[1], write_msg, strlen(write_msg) + 1);
    close(fd[1]); // Close write end after writing
}
```

- **Close unused read end (`fd[0]`)**:
  - The parent writes to the pipe, so it doesn't need the read end.
- **Write to the pipe**:
  - `write(fd[1], write_msg, strlen(write_msg) + 1)` writes the message (including the null terminator `+1`) to the write end of the pipe.
- **Close write end (`fd[1]`)**:
  - Closing the write end signals to the child that no more data will be sent.

---

6. Child Process

```
else { // Child process
    close(fd[1]); // Close unused write end
```

```
    read(fd[0], read_msg, sizeof(read_msg));
    printf("Child received: %s\n", read_msg);
    close(fd[0]); // Close read end after reading
}
```

- **Close unused write end (`fd[1]`)**:
  - The child reads from the pipe, so it doesn't need the write end.
- **Read from the pipe**:
  - `read(fd[0], read_msg, sizeof(read_msg))` reads data from the read end into the `read_msg` buffer.
- **Print the received message**:
  - `printf("Child received: %s\n", read_msg)` outputs the message received by the child.
- **Close read end (`fd[0]`)**:
  - Closing the read end releases resources and signals that the child is done reading.

---

7. Return 0

```
return 0;
```

- Indicates that the program executed successfully.

4. Framing Protocol: WAP for Character Count

Steps to Implement the Character Count Protocol

1. Understand the Problem

- **Sender**: Constructs frames with a leading character count byte.
- **Receiver**: Reads frames based on the count byte and processes the content.

2. Key Components

- **Sender**:
  - Calculate the length of the message.
  - Add the length as the first byte.
  - Send the frame.
- **Receiver**:
  - Read the first byte to determine the length of the frame.
  - Extract the rest of the frame based on the length.

3. Plan the Implementation

- Write a program that:
  1. Simulates the sender creating frames.
  2. Simulates the receiver reading frames.

4. Code Implementation

Here's an example in C:

c
Copy code

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

// Function to simulate the sender
void sender(const char *messages[], int num_messages, char frames[][100]) {
    for (int i = 0; i < num_messages; i++) {
        int length = strlen(messages[i]);    // Calculate the message length
        frames[i][0] = length;               // First byte is the length
        strcpy(frames[i] + 1, messages[i]); // Copy the message after the length
    }
}

// Function to simulate the receiver
void receiver(char frames[][100], int num_frames) {
    for (int i = 0; i < num_frames; i++) {
```

```
        int length = frames[i][0]; // Read the first byte as length
        printf("Frame %d (Length: %d): %.*s\n", i + 1, length, length, frames[i] + 1);
    }
}

int main() {
    const char *messages[] = {"Hello", "World", "Character Count Protocol"};
    int num_messages = sizeof(messages) / sizeof(messages[0]);
    char frames[10][100]; // Array to store frames

    // Step 1: Sender creates frames
    sender(messages, num_messages, frames);

    // Step 2: Receiver processes frames
    printf("Receiver Output:\n");
    receiver(frames, num_messages);

    return 0;
}
```

5. Explanation of the Code

1. **Sender Function**:
    - Calculates the length of each message.
    - Adds the length as the first byte of the frame.
    - Appends the message after the length byte.
2. **Receiver Function**:
    - Reads the first byte of each frame to get the message length.
    - Extracts and prints the message using the length.
3. **Main Function**:
    - Simulates messages as input.
    - Calls the sender and receiver functions to demonstrate the protocol.

6. Compilation and Execution

- Compile the code:

  bash
  Copy code
  ```
  gcc -o char_count_protocol char_count_protocol.c
  ```

- Run the program:

  bash
  Copy code
  ```
  ./char_count_protocol
  ```

7. Expected Output

plaintext
Copy code
Receiver Output:
Frame 1 (Length: 5): Hello
Frame 2 (Length: 5): World
Frame 3 (Length: 26): Character Count Protocol

8. Notes

- The frames array is used to simulate communication. In a real scenario, frames could be sent over a network or written to a file.
- The protocol assumes that frames are correctly formatted with the first byte indicating the length.

--

1. Include Necessary Libraries

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

- **`stdio.h`**: Provides functions like `printf` for displaying output.
- **`string.h`**: Provides string manipulation functions like `strlen` and `strcpy`.

---

2. Define the `sender` Function

```
void sender(const char *messages[], int num_messages, char frames[][100]) {
    for (int i = 0; i < num_messages; i++) {
        int length = strlen(messages[i]);     // Calculate the message length
        frames[i][0] = length;                // First byte is the length
        strcpy(frames[i] + 1, messages[i]); // Copy the message after the
length
    }
}
```

What the sender does:

- **Parameters**:
    - `messages`: An array of strings to be framed.
    - `num_messages`: The number of messages.
    - `frames`: A 2D array where each row represents a frame containing the length and message.
- **For Each Message**:
    1. Compute the **length** of the message using `strlen`.

2. Store the **length** in the first byte of the frame (`frames[i][0]`).
3. Copy the message to the frame starting from the second byte (`frames[i] + 1`) using `strcpy`.

The result is that each row in the `frames` array contains:

- The first byte as the message length.
- The rest as the actual message.

---

3. Define the `receiver` Function

```
void receiver(char frames[][100], int num_frames) {
    for (int i = 0; i < num_frames; i++) {
        int length = frames[i][0]; // Read the first byte as length
        printf("Frame %d (Length: %d): %.*s\n", i + 1, length, length,
frames[i] + 1);
    }
}
```

What the receiver does:

- **Parameters**:
  - `frames`: A 2D array of frames to process.
  - `num_frames`: The number of frames to process.
- **For Each Frame**:
  1. Retrieve the **length** of the message from the first byte of the frame (`frames[i][0]`).
  2. Use `printf` to display:
     - The frame number.
     - The message length.
     - The message itself using `frames[i] + 1`, formatted with `%.*s` to ensure only the specified length is displayed.

---

4. Define the `main` Function

```
int main() {
    const char *messages[] = {"Hello", "World", "Character Count Protocol"};
    int num_messages = sizeof(messages) / sizeof(messages[0]);
    char frames[10][100]; // Array to store frames

    // Step 1: Sender creates frames
    sender(messages, num_messages, frames);

    // Step 2: Receiver processes frames
    printf("Receiver Output:\n");
```

```
    receiver(frames, num_messages);

    return 0;
}
```

What the main function does:

1. **Define the messages**:
   - `messages[]` contains the strings to be transmitted.
   - `num_messages` calculates the number of messages using `sizeof`.
2. **Define the frames**:
   - `frames[10][100]` is a 2D array with 10 rows and space for 100 characters in each row. It stores the frames created by the sender.
3. **Call the sender**:
   - The sender converts the `messages` into frames and stores them in `frames`.
4. **Call the receiver**:
   - The receiver processes the frames and prints the message details.

Program Output

```
mathematica
Copy code
Receiver Output:
Frame 1 (Length: 5): Hello
Frame 2 (Length: 5): World
Frame 3 (Length: 23): Character Count Protocol
```

5. WAP to Implement Framing Protocol: Byte Stuffing

Steps to Implement Byte Stuffing

1. Understand the Concept

- **Special Characters**:
  - **Start-of-frame (SOF)**: Indicates the beginning of a frame (e.g., '@').
  - **Escape (ESC)**: Used to indicate that the next character is part of the payload, not a control character (e.g., '#').
- **Sender**:
  - Adds SOF at the beginning of the frame.
  - Replaces each SOF and ESC in the payload with an ESC followed by the special character.
- **Receiver**:
  - Reads the frame and removes the ESC before special characters in the payload.

2. Define the Key Functions

- **Sender**: Adds SOF and performs byte stuffing.
- **Receiver**: Detects SOF, interprets ESC sequences, and reconstructs the original message.

3. Code Implementation

Below is an example program in C:

c
Copy code
```c
#include <stdio.h>
#include <string.h>

#define SOF '@'   // Start-of-frame marker
#define ESC '#'   // Escape character

// Function to perform byte stuffing at the sender's side
void sender(const char *message, char *stuffed_frame) {
    int j = 0;
    stuffed_frame[j++] = SOF; // Add SOF at the start of the frame

    for (int i = 0; message[i] != '\0'; i++) {
        if (message[i] == SOF || message[i] == ESC) {
            stuffed_frame[j++] = ESC; // Add escape character
        }
        stuffed_frame[j++] = message[i]; // Add the actual character
    }
```

```
    stuffed_frame[j++] = SOF; // Add SOF at the end of the frame
    stuffed_frame[j] = '\0';  // Null-terminate the stuffed frame
}

// Function to perform byte unstuffing at the receiver's side
void receiver(const char *stuffed_frame, char *original_message) {
    int j = 0;
    for (int i = 1; stuffed_frame[i] != SOF; i++) { // Skip the initial SOF
        if (stuffed_frame[i] == ESC) {
            i++; // Skip the escape character
        }
        original_message[j++] = stuffed_frame[i];
    }
    original_message[j] = '\0'; // Null-terminate the original message
}

int main() {
    const char *message = "Hello @World# Protocol";
    char stuffed_frame[100], original_message[100];

    // Step 1: Perform byte stuffing
    sender(message, stuffed_frame);
    printf("Stuffed Frame: %s\n", stuffed_frame);

    // Step 2: Perform byte unstuffing
    receiver(stuffed_frame, original_message);
    printf("Original Message: %s\n", original_message);

    return 0;
}
```

4. Explanation of the Code

1. **Sender Function**:
   - Adds an SOF at the start and end of the frame.
   - Scans the message for SOF and ESC.
   - Adds an ESC before any SOF or ESC found in the message.
   - Constructs the stuffed frame.
2. **Receiver Function**:
   - Skips the initial SOF.
   - Detects ESC and skips it before adding the next character.
   - Stops processing upon encountering the final SOF.
3. **Main Function**:
   - Defines a test message.
   - Calls the sender and receiver functions.
   - Displays the stuffed frame and the reconstructed message.

5. Compilation and Execution

- Compile the program:

  bash
  Copy code
  ```
  gcc -o byte_stuffing byte_stuffing.c
  ```

- Run the program:

  bash
  Copy code
  ```
  ./byte_stuffing
  ```

6. Expected Output

plaintext
Copy code
```
Stuffed Frame: @Hello #@World## Protocol@
Original Message: Hello @World# Protocol
```

7. Notes

- The SOF and ESC characters are predefined in the code. You can customize them as needed.
- Ensure the receiver processes the frame correctly by interpreting the ESC character.
- The program simulates communication between a sender and receiver for simplicity.

6. WAP to Implement Framing Protocol: Bit Stuffing

Steps to Implement Bit Stuffing

1. Understand the Concept

- **Flag Sequence**: A fixed bit pattern (e.g., 01111110) marks the start and end of a frame.
- **Bit Stuffing Rule**:
  - If five consecutive 1s appear in the data, insert a 0 immediately after them.
- **Sender**:
  - Adds the flag at the start and end of the frame.
  - Stuffs a 0 after five consecutive 1s in the data.
- **Receiver**:
  - Detects the flag.
  - Removes the stuffed 0 after every five consecutive 1s.

2. Key Steps

1. Read the input data as a binary string.
2. Add flag sequences.
3. Perform bit stuffing (insert a 0 after five 1s).
4. Simulate the receiver removing stuffed bits.

3. Code Implementation

Below is an example program in C:

c
Copy code
```
#include <stdio.h>
#include <string.h>

#define FLAG "01111110"

// Function to perform bit stuffing at the sender's side
void sender(const char *data, char *stuffed_data) {
    int count = 0, j = 0;

    // Add the flag sequence at the start of the frame
    strcpy(stuffed_data, FLAG);
    j += strlen(FLAG);

    for (int i = 0; data[i] != '\0'; i++) {
        if (data[i] == '1') {
            count++;
        } else {
```

```
            count = 0;
        }

        // Add the current bit to the stuffed data
        stuffed_data[j++] = data[i];

        // Stuff a '0' after five consecutive '1's
        if (count == 5) {
            stuffed_data[j++] = '0';
            count = 0;
        }
    }

    // Add the flag sequence at the end of the frame
    strcpy(stuffed_data + j, FLAG);
    j += strlen(FLAG);
    stuffed_data[j] = '\0'; // Null-terminate the stuffed data
}

// Function to perform bit unstuffing at the receiver's side
void receiver(const char *stuffed_data, char *original_data) {
    int count = 0, j = 0;

    // Skip the initial flag sequence
    int start = strlen(FLAG);

    for (int i = start; stuffed_data[i] != '\0'; i++) {
        // Stop at the final flag sequence
        if (strncmp(stuffed_data + i, FLAG, strlen(FLAG)) == 0) {
            break;
        }

        if (stuffed_data[i] == '1') {
            count++;
        } else {
            count = 0;
        }

        // Add the current bit to the original data
        original_data[j++] = stuffed_data[i];

        // Skip the stuffed '0' after five consecutive '1's
        if (count == 5 && stuffed_data[i + 1] == '0') {
            i++;
            count = 0;
        }
```

```
    }
    original_data[j] = '\0'; // Null-terminate the original data
}

int main() {
    const char *data = "0111111011111100001111111"; // Input binary string
    char stuffed_data[100], original_data[100];

    // Step 1: Perform bit stuffing
    sender(data, stuffed_data);
    printf("Stuffed Data: %s\n", stuffed_data);

    // Step 2: Perform bit unstuffing
    receiver(stuffed_data, original_data);
    printf("Original Data: %s\n", original_data);

    return 0;
}
```

4. Explanation of the Code

1. **Sender Function**:
   - Adds the flag sequence at the beginning and end of the frame.
   - Scans the input data for five consecutive 1s and inserts a 0 after them.
   - Constructs the stuffed data frame.
2. **Receiver Function**:
   - Skips the initial flag sequence.
   - Reads the frame and reconstructs the original data by removing the stuffed 0 after five consecutive 1s.
   - Stops processing upon detecting the final flag sequence.
3. **Main Function**:
   - Defines an input binary string.
   - Calls the sender to stuff the data.
   - Calls the receiver to unstuff and reconstruct the original data.
   - Displays both stuffed and original data.

5. Compilation and Execution

- Compile the program:

  bash
  Copy code
  ```
  gcc -o bit_stuffing bit_stuffing.c
  ```

- Run the program:

bash
Copy code
./bit_stuffing

6. Expected Output

plaintext
Copy code
Stuffed Data: 01111110011111001111101000011111001111110
Original Data: 0111111011111100001111111

7. Notes

- The FLAG sequence is predefined as 01111110 but can be customized.
- This implementation assumes the input data is a valid binary string.
- The program simulates a simple sender-receiver communication for bit stuffing.

7. WAP to Implement Error Detection: LRC and Checksum

1. LRC (Longitudinal Redundancy Check)

Concept:

- Data is divided into blocks of equal length.
- Perform bitwise XOR on corresponding bits across all blocks to generate the LRC.
- The LRC is appended to the transmitted data.

Steps:

1. Divide data into blocks of fixed size.
2. Calculate LRC by XORing each column of bits.
3. Append LRC to the data.

Code Implementation for LRC:

c
Copy code
```
#include <stdio.h>
#include <string.h>

void calculateLRC(char data[][9], int rows, char *lrc) {
    int colSum[8] = {0};

    // Calculate LRC
    for (int col = 0; col < 8; col++) {
        for (int row = 0; row < rows; row++) {
            colSum[col] ^= (data[row][col] - '0'); // XOR each bit column-wise
        }
        lrc[col] = colSum[col] + '0'; // Convert back to character
    }
    lrc[8] = '\0'; // Null-terminate the LRC
}

int main() {
    char data[4][9] = { // Example binary data (8 bits per block)
        "11001101",
        "10101010",
        "11110000",
        "00001111"
    };

    char lrc[9];
    calculateLRC(data, 4, lrc);
```

```
    printf("Input Data Blocks:\n");
    for (int i = 0; i < 4; i++) {
        printf("%s\n", data[i]);
    }

    printf("LRC: %s\n", lrc);

    return 0;
}
```

2. Checksum

Concept:

- Data is divided into blocks of fixed size.
- All blocks are added together (modulo 2^n, where n is the block size).
- Complement of the sum is the checksum.
- Checksum is appended to the transmitted data.

Steps:

1. Divide data into blocks of equal size.
2. Add all blocks together.
3. Calculate 1’s complement of the sum as the checksum.
4. Append checksum to the data.

Code Implementation for Checksum:

c
Copy code

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

// Function to calculate checksum
unsigned int calculateChecksum(int data[], int n) {
    unsigned int sum = 0;

    // Add all data blocks
    for (int i = 0; i < n; i++) {
        sum += data[i];
    }

    // Calculate the 1's complement of the sum
    unsigned int checksum = ~sum;
    return checksum;
```

```
}

int main() {
    int data[] = {0x1234, 0x5678, 0x9ABC, 0xDEF0}; // Example data (16-bit blocks)
    int n = sizeof(data) / sizeof(data[0]);

    // Calculate checksum
    unsigned int checksum = calculateChecksum(data, n);

    printf("Input Data Blocks:\n");
    for (int i = 0; i < n; i++) {
        printf("0x%X\n", data[i]);
    }

    printf("Checksum: 0x%X\n", checksum);

    return 0;
}
```

3. Explanation of the Code

For LRC:

1. **Data Blocks**: Input is divided into 8-bit blocks.
2. **LRC Calculation**:
   - Each column of bits is XORed across all blocks.
   - The result is the LRC, which is appended to the data.

For Checksum:

1. **Data Blocks**: Input is treated as 16-bit words.
2. **Sum**: All blocks are added together.
3. **Complement**: The 1's complement of the sum is the checksum.
4. **Append**: The checksum is added to the data for verification.

4. Compilation and Execution

Compile:

bash
Copy code
```
gcc -o lrc lrc.c
gcc -o checksum checksum.c
```

Execute:

bash
Copy code
./lrc
./checksum

5. Expected Output

For LRC:

plaintext
Copy code
Input Data Blocks:
11001101
10101010
11110000
00001111
LRC: 10010010

For Checksum:

plaintext
Copy code
Input Data Blocks:
0x1234
0x5678
0x9ABC
0xDEF0
Checksum: 0xDCB2

6. Notes

- Both methods are basic error detection mechanisms and assume no data corruption during processing.
- **LRC** is more suitable for character-oriented data, while **Checksum** is used in block-oriented systems (e.g., TCP/IP).