**Experiment -01**

**Aim**  -Implementation of Fork and EXEC in Unix Environment

**Objective:**

The objective of this lab session is to understand the concepts of process creation using fork() and executing different programs using exec() functions in Unix environment. This lab aims to provide hands-on experience in writing and executing programs utilizing these fundamental Unix system calls.

**Prerequisites**:

1. Basic knowledge of C programming language.

2. Understanding of Unix/Linux operating system environment.

**Equipment Required:**

1. Unix/Linux based operating system (preferably Ubuntu).

2. GCC compiler installed.

**Procedure:**

**1. Introduction:**

In this lab session, we will explore the concepts of process creation using the fork() system call and executing different programs using the exec() family of functions in a Unix environment. These are fundamental operations for managing processes in Unix-like operating systems.

**2. Writing the Fork Program:**

Open a text editor and create a new C program file (e.g., fork\_demo.c).

Write a C program to demonstrate the usage of the fork() system call. The program should include necessary header files and use the fork() function to create a child process.

Optionally, you can include code to print the process IDs (PID) of the parent and child processes for better understanding.

Save the file and close the text editor.

**3. Compiling and Executing the Fork Program:**

Open a terminal window.

Navigate to the directory where the C program file is saved.

Use the GCC compiler to compile the program. For example:

**Code:**

*gcc -o fork\_demo fork\_demo.c*

Execute the compiled program by running the generated executable. For example:

bash

**code:**

*./fork\_demo*

Observe the output which demonstrates process creation using fork().

**4. Writing the Exec Program:**

Create a new C program file (e.g., exec\_demo.c) or modify the existing one.

Write a C program to demonstrate the usage of the exec() system call. This program should include necessary header files and use the exec() function to execute a different program.

Optionally, you can experiment with different variants of the exec() function (e.g., execvp(), execl(), etc.) and explore passing arguments to the executed program.

Save the file and close the text editor.

**5. Compiling and Executing the Exec Program:**

Compile the program using GCC compiler. For example:

**Code:**

***gcc -o exec\_demo exec\_demo.c***

Execute the compiled program to observe the execution of the specified external program. For example:

**bash**

**code:**

./exec\_demo

Observe the output which demonstrates the execution of an external program using exec().

**6. Combining Fork and Exec:**

Create a new C program file (e.g., fork\_exec\_demo.c) or modify an existing one.

Write a C program that combines the fork() and exec() system calls to create a child process and execute an external program within it.

Save the file and close the text editor.

**7. Compiling and Executing the Combined Program:**

Compile the program using GCC compiler. For example:

**Copy code**

*gcc -o fork\_exec\_demo fork\_exec\_demo.c*

Execute the compiled program to observe the behavior of process creation and execution of an external program within the child process. For example:

**bash**

**code**

./fork\_exec\_demo

Observe the output which demonstrates the combined usage of fork() and exec().

**8. Experimentation and Exploration:**

Experiment with different combinations of fork() and exec() calls.Explore various options and arguments that can be passed to the exec() function.Try executing different external programs and observe the behavior.

**9. Conclusion:**

Through this lab session, you have gained practical experience in implementing fork() and exec() system calls in a Unix environment. These fundamental concepts are essential for understanding process management in Unix-like operating systems. Further experimentation and exploration are encouraged to deepen your understanding of these concepts.

**Experiment -02**

**Aim:**

The aim of this experiment is to implement signal handling mechanisms in a Unix environment to handle various signals sent to a process.

**Objective:**

The objective of this lab session is to gain hands-on experience in writing and executing programs utilizing signal handling mechanisms in Unix using signal() function and signal handlers.

**Prerequisites:**

Basic knowledge of C programming language.

Understanding of Unix/Linux operating system environment.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

Procedure:

**2. Writing the Signal Handling Program:**

Open a text editor and create a new C program file (e.g., signal\_handling.c).

Write a C program demonstrating the usage of signal() function to handle various signals.

Define signal handler functions to handle specific signals (e.g., SIGINT, SIGTERM).

Save the file.

**3. Compiling and Executing the Signal Handling Program:**

Open a terminal window.

Navigate to the directory containing the C program file.

Compile the program using GCC compiler:

code

gcc -o signal\_handling signal\_handling.c

Execute the compiled program:

bash

code

./signal\_handling

**4. Testing Signal Handling:**

Run the compiled program in the terminal.

Send different signals to the running program using keyboard shortcuts or terminal commands (e.g., Ctrl+C for SIGINT, kill command for SIGTERM).

Observe how the program reacts to each signal and verify if the corresponding signal handler function is executed.

**5. Modifying Signal Handling Program:**

Modify the signal handling program to handle additional signals or customize the behavior of existing signal handlers.

Save the changes in the file.

**6. Compiling and Executing the Modified Program:**

Recompile the modified program using GCC compiler:

code

gcc -o signal\_handling signal\_handling.c

Execute the compiled program to observe the updated signal handling behavior.

**7. Experimentation and Exploration:**

Experiment with different signal handling scenarios and observe the behavior.

Explore advanced signal handling techniques such as blocking signals, using sigaction() function, etc.

Try handling specific signals related to process termination, errors, and interrupts.

**8. Conclusion:**

1. Summary of the gained experience and understanding of signal handling mechanisms in Unix environment.
2. Encouragement for further experimentation and exploration to deepen understanding and explore advanced signal handling techniques.
3. By following this procedure, participants will gain practical experience in implementing signal handling mechanisms in a Unix environment, thereby enhancing their understanding of pro

**Experiment -3**

**Aim:**

The aim of this experiment is to implement Pthreads (POSIX threads) in a Unix environment to create and manage multiple threads within a process.

**Objective:**

The objective of this lab session is to gain hands-on experience in writing and executing programs utilizing Pthreads library to create and manage concurrent threads in Unix.

**Prerequisites:**

Basic knowledge of C programming language.

Understanding of Unix/Linux operating system environment.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

Procedure:

1. **Introduction**:

Provide an overview of the experiment's purpose and objectives.

**2. Writing the Pthreads Program:**

Open a text editor and create a new C program file (e.g., pthread\_demo.c).

Write a C program demonstrating the usage of Pthreads library to create and manage multiple threads.

Include the pthread.h header file for Pthreads functions and data types.

Write a thread function that will be executed by each thread.

Save the file.

3**. Compiling the Pthreads Program:**

Open a terminal window.

Navigate to the directory containing the C program file.

Compile the program using GCC compiler with -pthread option to link the Pthreads library:

code

gcc -o pthread\_demo pthread\_demo.c -pthread

4. Executing the Pthreads Program:

Execute the compiled program:

bash

code

./pthread\_demo

**5. Observing Thread Execution:**

Observe the output of the program which demonstrates the execution of multiple threads concurrently.

Each thread should execute the thread function defined in the program.

**6. Modifying the Pthreads Program:**

Modify the Pthreads program to experiment with different thread creation and management scenarios.

Add synchronization mechanisms such as mutexes or semaphores to control access to shared resources among threads.

Save the changes in the file.

**7. Recompiling and Re-executing the Modified Program:**

Recompile the modified program using GCC compiler with -pthread option:

Copy code

gcc -o pthread\_demo pthread\_demo.c -pthread

Execute the compiled program again to observe the updated behavior.

**8. Experimentation and Exploration:**

Experiment with different thread creation techniques (e.g., pthread\_create() function).

Explore thread synchronization mechanisms (e.g., mutexes, semaphores, condition variables) to prevent race conditions and ensure thread safety.

Try implementing multithreaded applications with different concurrency patterns (e.g., producer-consumer, reader-writer).

**9. Conclusion:**

1. Summary of the gained experience and understanding of Pthreads library for concurrent programming in Unix environment.
2. Encouragement for further experimentation and exploration to deepen understanding and explore advanced thread management techniques.
3. By following this procedure, participants will gain practical experience in implementing Pthreads library in a Unix environment, thereby enhancing their understanding of concurrent programming and thread management in Unix-like operating systems.

**Experiment -04**

Aim:

The aim of this experiment is to implement an echo program using TCP in a client-server architecture with an iterative server that handles one client at a time.

Objective:

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a client and a server using TCP sockets in Unix environment.

Prerequisites:

Basic knowledge of C programming language.

Understanding of TCP/IP networking concepts.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

Procedure:

Below is a simple echo program with a client and an iterative server using TCP in C:

**TCP Echo Server (Iterative):**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[BUFFER\_SIZE] = {0};

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

// Initialize address structure

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Bind the socket to localhost port 8080

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0) {

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// Listen for incoming connections

if (listen(server\_fd, 3) < 0) {

perror("Listen failed");

exit(EXIT\_FAILURE);

}

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

// Accept a new connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t\*)&addrlen)) < 0) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Loop to echo messages back to client

while (1) {

int valread = read(new\_socket, buffer, BUFFER\_SIZE);

if (valread == 0) {

printf("Client disconnected\n");

break;

}

printf("Message from client: %s\n", buffer);

send(new\_socket, buffer, strlen(buffer), 0);

memset(buffer, 0, sizeof(buffer));

}

// Close the socket

close(new\_socket);

close(server\_fd);

return 0;

}

**TCP Echo Client:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int sock = 0;

struct sockaddr\_in serv\_addr;

char buffer[BUFFER\_SIZE] = {0};

char message[BUFFER\_SIZE];

// Create socket file descriptor

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

// Initialize server address structure

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary form

if(inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0) {

perror("Invalid address/ Address not supported");

exit(EXIT\_FAILURE);

}

// Connect to server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0) {

perror("Connection failed");

exit(EXIT\_FAILURE);

}

printf("Connected to server\n");

// Loop to send and receive messages

while (1) {

printf("Enter message: ");

fgets(message, BUFFER\_SIZE, stdin);

send(sock, message, strlen(message), 0);

int valread = read(sock, buffer, BUFFER\_SIZE);

printf("Server response: %s\n", buffer);

memset(buffer, 0, sizeof(buffer));

}

// Close socket

close(sock);

return 0;

}

Experiment -5

Aim:

The aim of this experiment is to implement an echo program using TCP in a client-server architecture with a concurrent server that can handle multiple clients simultaneously.

Objective:

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a client and a server using TCP sockets in a concurrent manner in Unix environment.

Prerequisites:

Basic knowledge of C programming language.

Understanding of TCP/IP networking concepts.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

**Procedure:**

**1. Introduction:**

Provide an overview of the experiment's purpose and objectives.

**2. Writing the Concurrent Server Program:**

Open a text editor and create a new C program file (e.g., server\_concurrent.c).

Write a C program for the concurrent server that listens for incoming connections, creates a new thread to handle each client connection, receives messages from clients, and echoes them back to the respective clients.

Save the file.

**3. Compiling the Concurrent Server Program:**

Open a terminal window.

Navigate to the directory containing the server concurrent C program file.

Compile the program using GCC compiler with pthread library linked:

Copy code

gcc -o server\_concurrent server\_concurrent.c -pthread

4. Writing the Client Program:

Open a text editor and create a new C program file (e.g., client.c).

Write a C program for the client that establishes a connection with the server, sends messages to the server, and receives echoed messages.

Save the file.

**5. Compiling the Client Program:**

Open a terminal window.

Navigate to the directory containing the client C program file.

Compile the program using GCC compiler:

Copy code

gcc -o client client.c

**6. Executing the Concurrent Server:**

Run the compiled concurrent server program:

bash

Copy code

./server\_concurrent

**7. Executing the Client:**

Run the compiled client program:

bash

Copy code

./client

**8. Observing Communication:**

Observe the communication between the client and the server.

Verify that the client's messages are echoed back by the server.

Ensure that the server can handle multiple clients concurrently without blocking.

9. Experimentation and Exploration:

Experiment with different messages of varying lengths.

Explore error handling and edge cases (e.g., handling disconnection).

Try modifying the server to handle multiple clients using different concurrency models (e.g., processes instead of threads).

**10. Conclusion:**

Summary of the gained experience and understanding of implementing an echo program with a client and a concurrent server using TCP.

Encouragement for further experimentation and exploration to deepen understanding and explore advanced networking concepts.

By following this procedure, participants will gain practical experience in implementing client-server communication using TCP sockets in a concurrent manner in Unix environment, thereby enhancing their understanding of network programming concepts.

Experiment -06

**Aim:**

The aim of this experiment is to implement an echo program using UDP in a client-server architecture with a concurrent server that can handle multiple clients simultaneously.

**Objective:**

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a client and a server using UDP sockets in a concurrent manner in Unix environment.

**Prerequisites:**

Basic knowledge of C programming language.

Understanding of UDP networking concepts.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

**Procedure:**

**1. Introduction:**

Provide an overview of the experiment's purpose and objectives.

**2. Writing the Concurrent Server Program:**

Open a text editor and create a new C program file (e.g., server\_concurrent\_udp.c).

Write a C program for the concurrent server that binds to a UDP socket, listens for incoming datagrams, creates a new thread to handle each client request, echoes the received message back to the client.

Save the file.

**3. Compiling the Concurrent Server Program:**

Open a terminal window.

Navigate to the directory containing the server concurrent UDP C program file.

Compile the program using GCC compiler with pthread library linked:

Copy code

gcc -o server\_concurrent\_udp server\_concurrent\_udp.c -pthread

**4. Writing the Client Program:**

Open a text editor and create a new C program file (e.g., client\_udp.c).

Write a C program for the client that sends a message to the server using UDP, receives the echoed message from the server, and prints it to the console.

Save the file.

**5. Compiling the Client Program:**

Open a terminal window.

Navigate to the directory containing the client UDP C program file.

Compile the program using GCC compiler:

Copy code

gcc -o client\_udp client\_udp.c

**6. Executing the Concurrent Server:**

Run the compiled concurrent server program:

bash

Copy code

./server\_concurrent\_udp

**7. Executing the Client:**

Run the compiled client program:

bash

Copy code

./client\_udp

**8. Observing Communication:**

Observe the communication between the client and the server using UDP.

Verify that the client's messages are echoed back by the server.

Ensure that the server can handle multiple clients concurrently without blocking.

**9.** **Experimentation and Exploration:**

Experiment with different me

Experiment -07

**Aim:**

The aim of this experiment is to create a TCP client-server application where the client can retrieve the current date and time from the server.

**Objective:**

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a TCP client and server to retrieve date and time information.

**Prerequisites:**

Basic knowledge of C programming language.

Understanding of TCP/IP networking concepts.

**Equipment Required:**

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

Procedure:

**1. Introduction:**

Provide an overview of the experiment's purpose and objectives.

**2. Writing the TCP Date and Time Server Program:**

Open a text editor and create a new C program file (e.g., tcp\_date\_time\_server.c).

Write a C program for the TCP server that listens for incoming client connections, retrieves the current date and time, and sends it to the client.

Save the file.

**3. Compiling the TCP Date and Time Server Program:**

Open a terminal window.

Navigate to the directory containing the server C program file.

Compile the program using GCC compiler:

Copy code

gcc -o tcp\_date\_time\_server tcp\_date\_time\_server.c

**4. Writing the TCP Date and Time Client Program:**

Open a text editor and create a new C program file (e.g., tcp\_date\_time\_client.c).

Write a C program for the TCP client that establishes a connection with the server, receives the date and time information, and displays it.

Save the file.

**5. Compiling the TCP Date and Time Client Program:**

Open a terminal window.

Navigate to the directory containing the client C program file.

Compile the program using GCC compiler:

Copy code

gcc -o tcp\_date\_time\_client tcp\_date\_time\_client.c

**6. Executing the TCP Date and Time Server:**

Run the compiled server program:

bash

Copy code

./tcp\_date\_time\_server

7. Executing the TCP Date and Time Client:

Run the compiled client program:

bash

Copy code

./tcp\_date\_time\_client

**8. Observing Date and Time Retrieval:**

Observe the communication between the client and server.

Verify that the client receives the current date and time information from the server.

**9. Experimentation and Exploration:**

Experiment with running the client from different machines to connect to the server.

Explore error handling and edge cases (e.g., handling connection failures).

Try modifying the server to handle multiple client connections concurrently.

**10. Conclusion:**

Summary of the gained experience and understanding of implementing a TCP client-server application to retrieve date and time information.

Encouragement for further experimentation and exploration to deepen understanding and explore advanced networking concepts.

By following this procedure, participants will gain practical experience in writing TCP client-server applications to retrieve date and time information, thereby enhancing their understanding of network programming concepts

Experiment -08

**Aim:**

The aim of this experiment is to create a UDP client-server application where the client can retrieve the current date and time from the server.

**Objective:**

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a UDP client and server to retrieve date and time information.

**Prerequisites:**

Basic knowledge of C programming language.

Understanding of UDP networking concepts.

**Equipment Required:**

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

**Procedure:**

1.**Writing the UDP Date and Time Server Program:**

Open a text editor and create a new C program file (e.g., udp\_date\_time\_server.c).

Write a C program for the UDP server that binds to a UDP socket, listens for incoming datagrams from clients, retrieves the current date and time, and sends it back to the client.

Save the file.

**UDP Date and Time Server:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int sockfd;

struct sockaddr\_in servaddr, cliaddr;

char buffer[BUFFER\_SIZE];

time\_t current\_time;

char\* time\_string;

// Creating socket file descriptor

if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

memset(&cliaddr, 0, sizeof(cliaddr));

// Filling server information

servaddr.sin\_family = AF\_INET; // IPv4

servaddr.sin\_addr.s\_addr = INADDR\_ANY;

servaddr.sin\_port = htons(PORT);

// Bind the socket with the server address

if (bind(sockfd, (const struct sockaddr\*)&servaddr, sizeof(servaddr)) < 0) {

perror("Bind failed");

exit(EXIT\_FAILURE);

}

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

int len, n;

len = sizeof(cliaddr); //len is value/resuslt

while (1) {

n = recvfrom(sockfd, (char\*)buffer, BUFFER\_SIZE, MSG\_WAITALL, (struct sockaddr\*)&cliaddr, &len);

buffer[n] = '\0';

// Get the current date and time

current\_time = time(NULL);

time\_string = ctime(&current\_time);

// Send the date and time to the client

sendto(sockfd, (const char\*)time\_string, strlen(time\_string), MSG\_CONFIRM, (const struct sockaddr\*)&cliaddr, len);

}

return 0;

}

**UDP Date and Time Client:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int sockfd;

char buffer[BUFFER\_SIZE];

struct sockaddr\_in servaddr;

// Create socket file descriptor

if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

// Filling server information

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(PORT);

servaddr.sin\_addr.s\_addr = INADDR\_ANY;

int n, len;

printf("Sending request to server...\n");

sendto(sockfd, (const char\*)"Date and time request", strlen("Date and time request"), MSG\_CONFIRM, (const struct sockaddr\*)&servaddr, sizeof(servaddr));

n = recvfrom(sockfd, (char\*)buffer, BUFFER\_SIZE, MSG\_WAITALL, (struct sockaddr\*)&servaddr, &len);

buffer[n] = '\0';

printf("Date and time received from server: %s\n", buffer);

close(sockfd);

return 0;

}

**Conclusion:**

Summary of the gained experience and understanding of implementing a UDP client-server application to retrieve date and time information.

Encouragement for further experimentation and exploration to deepen understanding and explore advanced networking concepts.

By following this procedure, participants will gain practical experience in writing UDP client-server applications to retrieve date and time information, thereby enhancing their understanding of network programming concepts.

Experiment -09

Aim:

The aim of this experiment is to demonstrate the implementation of client and server routines using I/O multiplexing, allowing multiple I/O operations to be monitored simultaneously.

Objective:

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a client and a server using I/O multiplexing techniques.

Prerequisites:

Basic knowledge of C programming language.

Understanding of socket programming and basic networking concepts.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

Procedure:

1. Writing the Multiplexed Server Program:

Open a text editor and create a new C program file (e.g., server\_multiplexed.c).

Write a C program for the server that uses I/O multiplexing (select or poll) to handle multiple client connections concurrently.

Save the file.

2. Compiling the Multiplexed Server Program:

Open a terminal window.

Navigate to the directory containing the server C program file.

Compile the program using GCC compiler:

Copy code

gcc -o server\_multiplexed server\_multiplexed.c

4. Writing the Multiplexed Client Program:

Open a text editor and create a new C program file (e.g., client\_multiplexed.c).

Write a C program for the client that establishes a connection with the server.

Save the file.

5. Compiling the Multiplexed Client Program:

Open a terminal window.

Navigate to the directory containing the client C program file.

Compile the program using GCC compiler:

Copy code

gcc -o client\_multiplexed client\_multiplexed.c

6. Executing the Multiplexed Server:

Run the compiled server program:

bash

Copy code

./server\_multiplexed

7. Executing the Multiplexed Client:

Run the compiled client program:

bash

Copy code

./client\_multiplexed

8. Observing Multiplexed Communication:

Observe the communication between the client and server using I/O multiplexing.

Verify that the server can handle multiple client connections concurrently.

9. Experimentation and Exploration:

Experiment with different numbers of client connections to observe the scalability of the multiplexed server.

Explore different I/O multiplexing mechanisms (select, poll) and compare their performance.

Try modifying the server to handle different types of client requests (e.g., echoing messages, file transfer).

10. Conclusion:

Summary of the gained experience and understanding of implementing client and server routines using I/O multiplexing techniques.

Encouragement for further experimentation and exploration to deepen understanding and explore advanced networking concepts.

By following this procedure, participants will gain practical experience in writing client and server routines demonstrating I/O multiplexing, thereby enhancing their understanding of advanced socket programming concepts.

**Experiment -10**

**Aim:**

The aim of this experiment is to create a simple echo client-server application using Unix domain stream sockets.

**Objective:**

The objective of this lab session is to gain hands-on experience in writing and executing programs to establish communication between a client and a server using Unix domain stream sockets.

**Prerequisites:**

Basic knowledge of C programming language.

Understanding of Unix domain sockets and basic networking concepts.

Equipment Required:

Unix/Linux based operating system (preferably Ubuntu).

GCC compiler installed.

**Procedure:**

**2. Writing the Echo Server Program:**

Open a text editor and create a new C program file (e.g., server\_unix\_stream.c).

Write a C program for the echo server that listens for client connections, receives messages from clients, and echoes them back to the clients.

Save the file.

**3. Compiling the Echo Server Program:**

Open a terminal window.

Navigate to the directory containing the server C program file.

Compile the program using GCC compiler:

Copy code

gcc -o server\_unix\_stream server\_unix\_stream.c

**4. Writing the Echo Client Program:**

Open a text editor and create a new C program file (e.g., client\_unix\_stream.c).

Write a C program for the echo client that establishes a connection with the server, sends messages to the server, and receives echoed messages.

Save the file.

**5. Compiling the Echo Client Program:**

Open a terminal window.

Navigate to the directory containing the client C program file.

Compile the program using GCC compiler:

Copy code

gcc -o client\_unix\_stream client\_unix\_stream.c

**6. Executing the Echo Server:**

Run the compiled server program:

bash

Copy code

./server\_unix\_stream

7. Executing the Echo Client:

Run the compiled client program:

bash

Copy code

./client\_unix\_stream

**8. Observing Communication:**

Observe the communication between the client and server using Unix domain stream sockets.

Verify that the client's messages are echoed back by the server.

**9. Experimentation and Exploration:**

Experiment with running multiple instances of the client to observe how the server handles concurrent connections.

Explore error handling and edge cases (e.g., handling connection failures).

Try modifying the server to handle different types of client requests (e.g., echoing messages, file transfer).

**10. Conclusion:**

Summary of the gained experience and understanding of implementing an echo client and server using Unix domain stream sockets.

Encouragement for further experimentation and exploration to deepen understanding and explore advanced socket programming concepts.

By following this procedure, participants will gain practical experience in writing echo client and server programs using Unix domain stream sockets, thereby enhancing their understanding of Unix domain socket programming concepts.