Computer Science & Information Technology Department NED University of Engineering and Technology, Karachi.

Report Title: Developing a Minimal Web Server with Core System Calls

Complex Computing Problem (CCP)

Course Title: Operating Systems CT-353

Course Instructor: Dr. Maria Andleeb

Group Members:

Afifa Siddique DT-22003

Ashna Iqbal Sheikh DT-22019

1. Introduction

1.1 Purpose of the Report

This report explores the practical use of system calls in building a basic web server. It highlights how system-level operations, particularly those related to networking and process control, are essential in processing HTTP requests and managing client sessions.

1.2 Report Scope

The server implementation utilizes key Linux system calls such as socket(), bind(), listen(), and accept() for network communication, and fork() for handling multiple clients simultaneously. It also discusses how these calls align with operating system fundamentals, like managing system resources and isolating processes.

1.3 System Call Overview

System calls serve as an interface between user applications and the kernel. In this web server project, they are used for:

- **Socket Handling:** Creating sockets to manage TCP communication.
- **Process Handling:** Using separate processes to manage each client session.
- **Data Exchange:** Reading requests and delivering HTTP responses.

2. Target Operating System and Application Domain

2.1 Selected Operating System

Linux was chosen for its powerful system call interface and reliable networking capabilities. It is a common choice for both small and large-scale web server deployments.

2.2 Application Area

The project is within the **Networking domain**, focusing on:

- **Client-Server Model:** Handling client requests and responses over TCP.
- **Concurrent Processing:** Using process isolation for simultaneous connections.
- **Efficient Resource Use:** Ensuring optimal use and release of sockets and memory.

3. Application of System Calls

3.1 Categories of System Calls Used

- **Networking:**
    - socket(): Opens communication endpoints.
    - bind(): Assigns the socket to a specific IP and port.
    - listen(): Prepares the socket to accept connections.
    - accept(): Receives client connections.
- **Process Management:**
    - fork(): Creates child processes to manage individual clients.
    - wait(): Handles cleanup of terminated child processes.
- **File Handling:**
    - read(): Receives data from the client.
    - write()/send(): Sends HTTP responses.

3.2 System Calls Explained

1. Networking System Calls

➔ **socket()**
Use: Creates a socket for TCP/IP communication.
Syntax: int socket(int domain, int type, int protocol);

➔ **bind()**
Use: Links the socket to an IP and port.
Example:

```
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()**
Use: Enables the socket to accept connections.

```
listen(server_fd, 5);
```

➔ **accept()**
Use: Accepts an incoming client connection.

```
int client_fd = accept(server_fd, (struct sockaddr*) &client_addr, &addrlen);
```

2. Process Management System Calls

fork()

Use: Creates a new process for each client.

```
 pid_t pid = fork();
if (pid == 0) {
   handle_client(client_fd);
   exit(0);
} else {
   close(client_fd);
}
```

3. File Management System Calls

read() and write()

Use: Handle HTTP data transmission.

```
 char buffer[1024];
read(client_fd, buffer, sizeof(buffer));
write(client_fd, response, strlen(response));
```

4. Code Implementation

4.1 Web Server Code Example

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>



void handle_client(int client_fd) {

    char buffer[1024];

    read(client_fd, buffer, 1024);

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



    // HTTP response

    char response[] =

        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html\r\n\r\n"
        "<html><body><h1>Hello, World!</h1></body></html>";
    send(client_fd, response, strlen(response), 0);
    close(client_fd);
}

int main() {
    int server_fd;
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

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

    // Step 2: Bind socket to port 8080
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }
```

```
// Step 3: Listen for connections
if (listen(server_fd, 10) < 0) {
    perror("Listen failed");
    exit(EXIT_FAILURE);
}
printf("Server listening on port 8080...\n");

// Step 4: Accept and handle clients
while (1) {
    int client_fd = accept(server_fd, (struct sockaddr *)&address, &addrlen);
    if (client_fd < 0) {
        perror("Accept failed");
        continue;
    }

    // Handle client in child process
    if (fork() == 0) {
        close(server_fd);
        handle_client(client_fd);
        exit(0);
    }
    close(client_fd);
}

return 0;}
```

5. Evaluation of Objectives

5.1 Achievements

- **Basic Server Functionality:**
  The server processes HTTP requests and sends responses using essential system calls.
- **Concurrent Processing:**
  Each client is handled by a separate process using fork(), ensuring non-blocking operation.
- **Resource Handling:**
  Calls like close() ensure efficient release of used resources.
- **Fault Tolerance:**
  Process isolation ensures that failures in one session do not affect others.
- **Operating System Integration:**
  The implementation reflects key OS principles like resource efficiency and modular design.

5.2 Limitations Identified

- **Scalability:**
  Forking for every client is resource-intensive. Alternatives like threading or async I/O could improve performance.
- **Limited Protocol Support:**
  Only basic GET requests are supported; lacks HTTPS or other HTTP methods.
- **Lack of Robust Logging and Error Handling:**
  Enhanced error reporting and logging mechanisms are needed.
- **Security Deficiencies:**
  Without SSL/TLS, the server cannot securely transmit data.

6. Conclusion

This report highlights the effective use of system calls in building a basic yet functional web server. By leveraging system-level calls for networking and process handling, the project successfully demonstrates key operating system concepts such as concurrency, modularity, and resource management. Future enhancements could focus on scalability (via threads or async I/O) and security (via HTTPS).