**SP Lab Project Report**



**Fall 2023**

**CSE-302L Systems Programming Lab**

Shahzad Bangash **(21PWCSE1980)**

Suleman Shah **(21PWCSE1983)**

Ali Asghar **(21PWCSE2059)**

Muhammad Shahab **(21PWCSE2074)**

“On our honor, as students of University of Engineering and Technology, we have neither given nor received unauthorized assistance on this academic work.”

Submitted to:

**Engr. Abdullah Hamid**

February 1, 2024

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

**Sections:**

* Introduction Page #3
* Web Servers Page #3
* Tools Used Page #4
* Methodology Page #4
* Sockets Page #4
* Code Explanation Page #4
* SAP-1 Operation Page #15
* Simulation Page #16
* Conclusion Page #20
* References Page #20

**Web Server on Linux**

**Introduction:**

A web server is software and hardware that uses HTTP (Hypertext Transfer Protocol) and other protocols to respond to client requests made over the World Wide Web. The main job of a web server is to display website content through storing, processing and delivering webpages to users. Besides HTTP, web servers also support SMTP (Simple Mail Transfer Protocol) and FTP (File Transfer Protocol), used for email, file transfer and storage.

**Linux for Web Servers:**

Web servers are software applications that serve content to clients over the internet. In the context of Linux, there are several popular web servers available, including Apache, NGINX, and Lighttpd. These servers are known for their performance, security, and flexibility.

**Apache** is the most widely used web server software in the world. It’s open-source and maintained by the Apache Software Foundation. Apache is known for its power and flexibility, and it has a wide range of modules that extend its functionality.

**NGINX** is another popular web server for Linux. It’s particularly known for its high performance and low memory usage, making it a good choice for high-traffic websites. NGINX can also be used as a reverse proxy, load balancer, and HTTP cache.

**Lighttpd** (pronounced “lighty”) is a lightweight web server, as its name suggests. It’s designed to be fast and efficient, making it an excellent choice for servers with less powerful hardware.

These web servers can serve static content (like HTML files, images, CSS, etc.) directly to the client. They can also work with other software to serve dynamic content. For example, they can work with a database server (like MySQL) and a server-side scripting language (like PHP) to generate web pages on the fly based on user input or other data.

Linux is a popular choice for running these web servers due to its stability, security, and open-source nature. It allows for fine-grained control over the server’s settings and can be customized to meet the specific needs of each deployment. Moreover, Linux’s robust command-line interface and strong community support make managing and troubleshooting web servers easier.

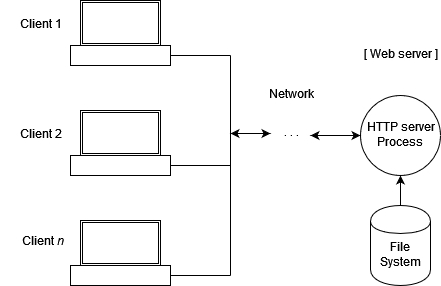


Figure 1, Block Diagram of a Web Server

**Sockets:**

A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Sockets are the only interprocess communication. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

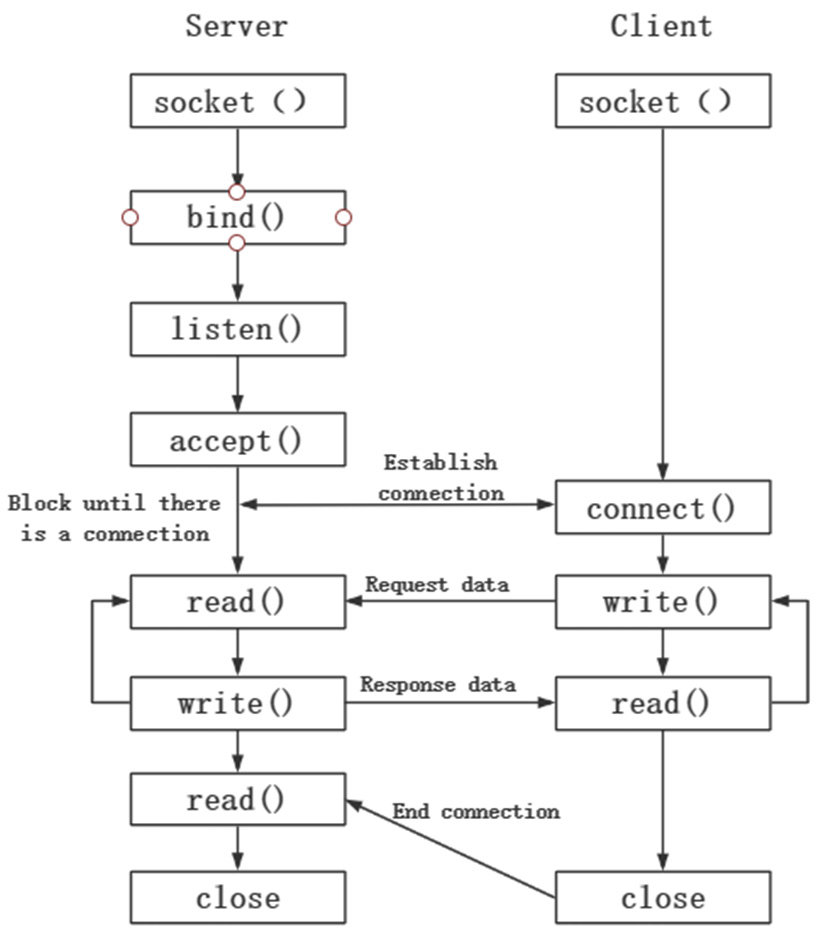


Figure 2, Block Diagram of a Simple Socket Communication

**Tools Used:**

* 1. **C language**

A high level, general purpose programming language.

* 1. **Ubuntu**

A famous distribution of Linux Operating System.

**Methodology:**

We focus on implementing the essential features that enable our server to interact with web clients. In a real-world scenario, a program would either offer a more complete HTTP implementation or interface with one of the many excellent web server implementations available, rather than directly providing HTTP services.

We also do not strive for full compliance with HTML specifications. Our goal is to generate simple HTML output that can be processed by popular web browsers.

We do not attempt to control the resources (such as the number of processes or memory usage) consumed by the server or its modules. Many multiprocess web server implementations service connections using a fixed pool of processes, rather than creating a new child process for each connection.

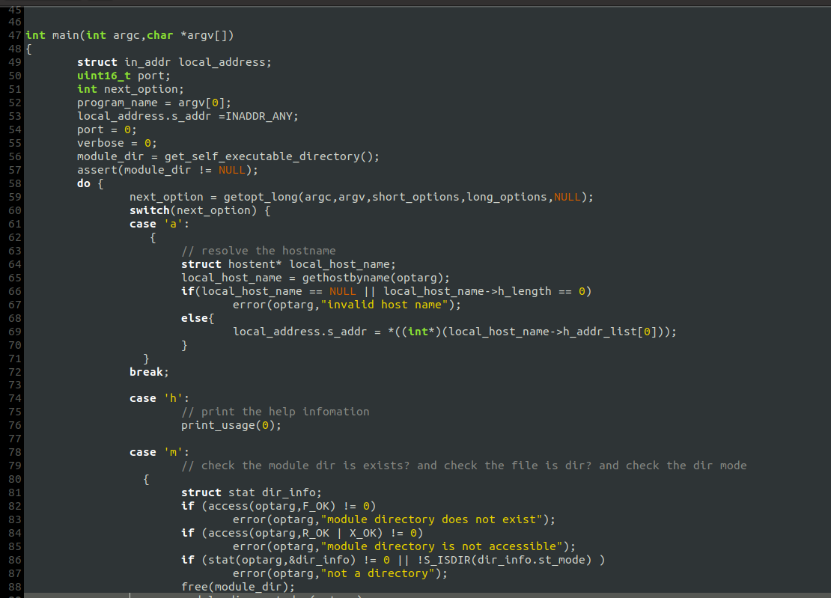
Our server loads the shared library for a server module each time it is requested and then immediately unloads it once the request is completed. A more efficient implementation would likely cache loaded modules.

**Code Explanation:**

The subsequent sections provide detailed explanations of each module.

**1. main.c:**

This is the main entry code of this project. It starts by parsing command-line options for the local address to bind to, the port, and the directory to load modules from. It also checks if the specified module directory exists and is accessible. If the verbose flag is set, it prints a message about where modules will be loaded from. Finally, it starts the server with the specified local address and port. This code is a good starting point for a web server, but it doesn’t handle HTTP requests or serve web pages, which would need to be added for a fully functional web server.

A screen shot of a computer program

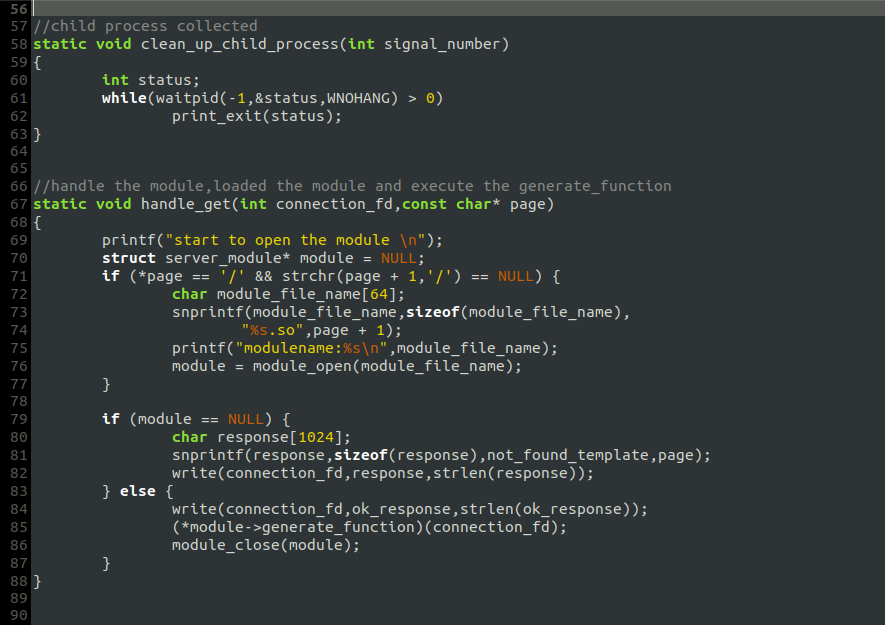
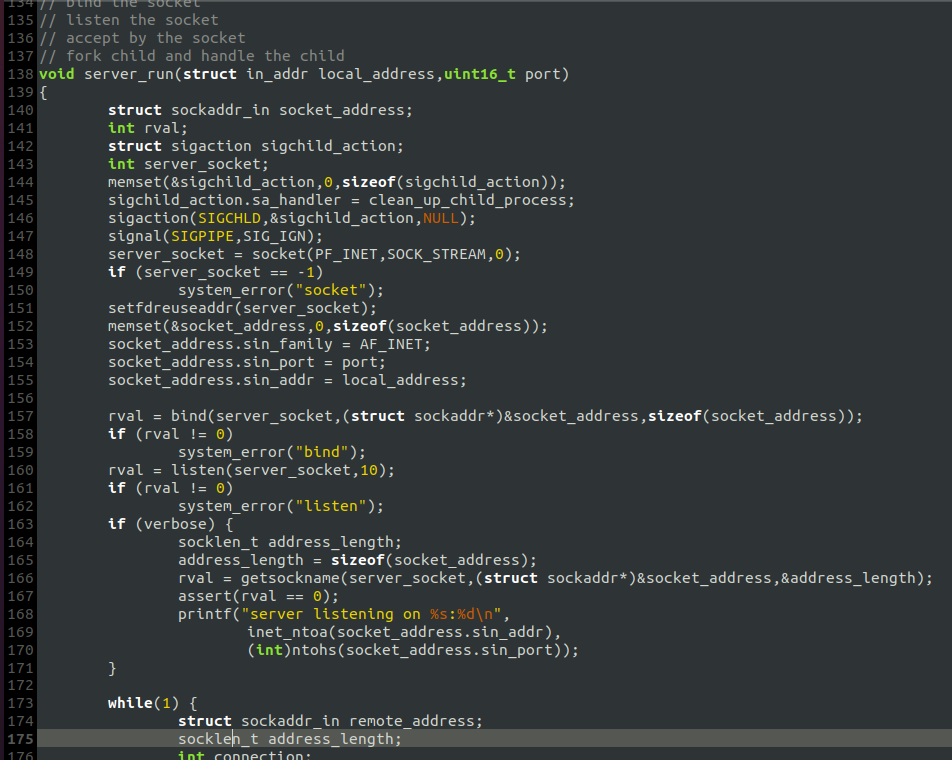
Description automatically generated**Code:**

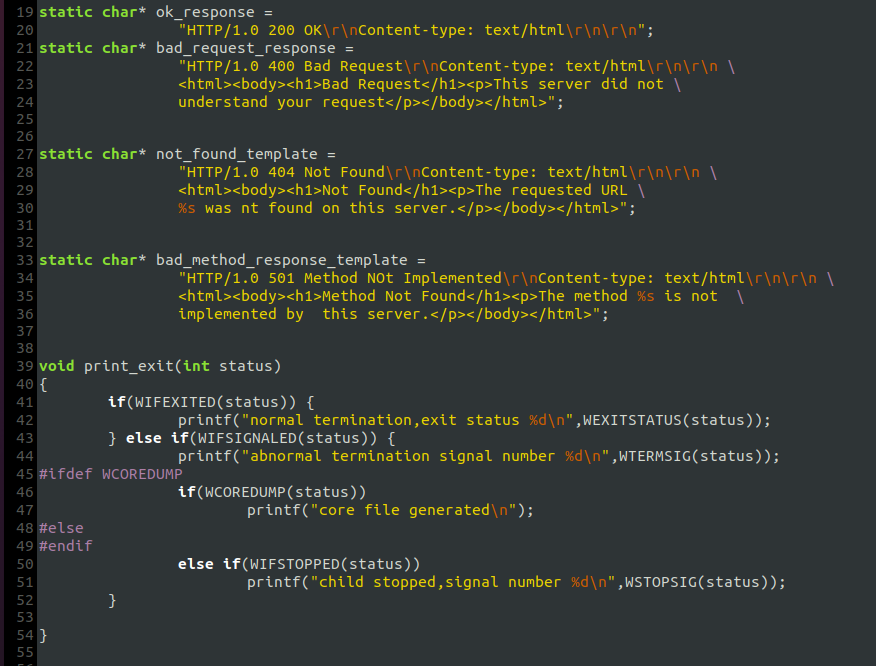
A computer screen shot of a program code

Description automatically generated

**2. server.c:**

The server.c file contains several functions for running a web server. The server\_run function is the main entry point, which starts the server and accepts connections on a specified local address and port. It handles each client connection in a child process, invoking **handle\_connection** to process HTTP requests. The server only processes HTTP 1.0 and 1.1 GET requests, responding with appropriate HTTP result codes for other requests or errors. If a well-formed GET request is received, handle\_get is called to service it, attempting to load a server module corresponding to the requested page. If successful, it sends a 200 result code and invokes the module’s module\_generate function to generate and send the HTML source for a web page. The server\_run function also installs clean\_up\_child\_process as the signal handler for SIGCHLD to clean up terminated child processes.

**Code:**

****

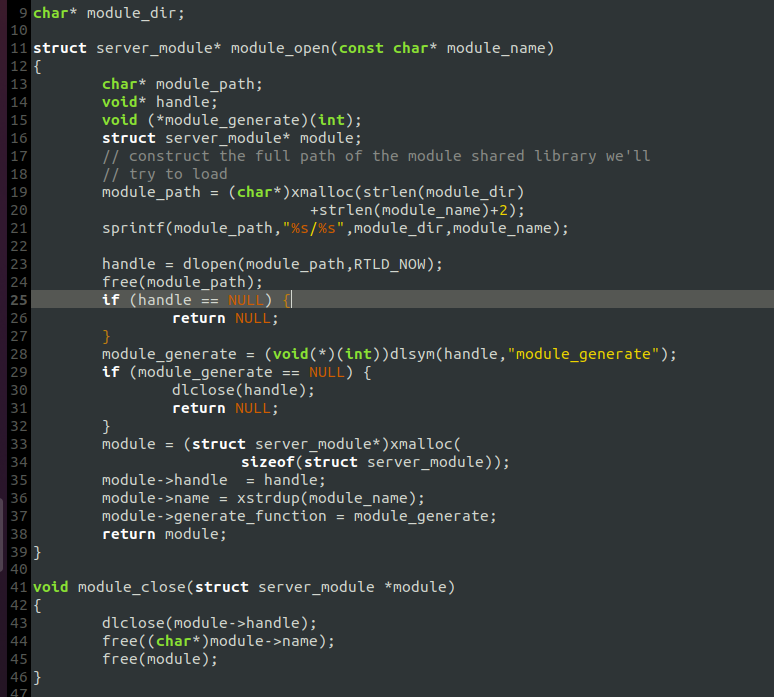
**A computer screen shot of a program code

Description automatically generated**

**3. module.c:**

Module.c code is for managing server modules in a web server. The module\_open function takes a module name as input, constructs the full path of the module’s shared library, and attempts to load it using dlopen. If successful, it looks up the module\_generate function in the shared library using dlsym. If this function is found, a new server\_module structure is allocated and initialized with the handle of the shared library, the module name, and the module\_generate function. The module\_close function takes a server\_module structure as input, closes the shared library using dlclose, and frees the memory allocated for the module name and the server\_module structure.

**Code:**



**4. other scripts:**

The remaining scripts include:

1. **libsockets.c**

contains functions for manipulating sockets

1. **common.c**

contains functions of general utility that are used throughout the program.

1. **Modules implementation code**

contains implementation of modules(total 4).

**System Calls Used:**

Following System Calls are used in this project:

* **socket:** For creating a socket
* **closes**: For destroying a socket
* **connect**: To creates connection between two sockets
* **bind**: To labels a server socket with an address
* **listen**: To configures a socket to accept conditions
* **recv:** to receive messages from a socket.
* **accept**: To accepts a connection and creates a new socket for the connection
* **fork**: For making a child process
* **waitpid**: For waiting a child process
* **execv**: For executing a module by a child process
* **write**: For executing a module by a child process
* **stat**: For getting info about certain parameters
* **close**: For closing a file descriptor

**Output:**

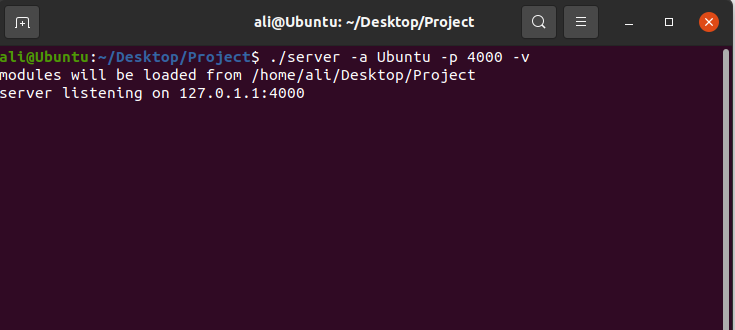
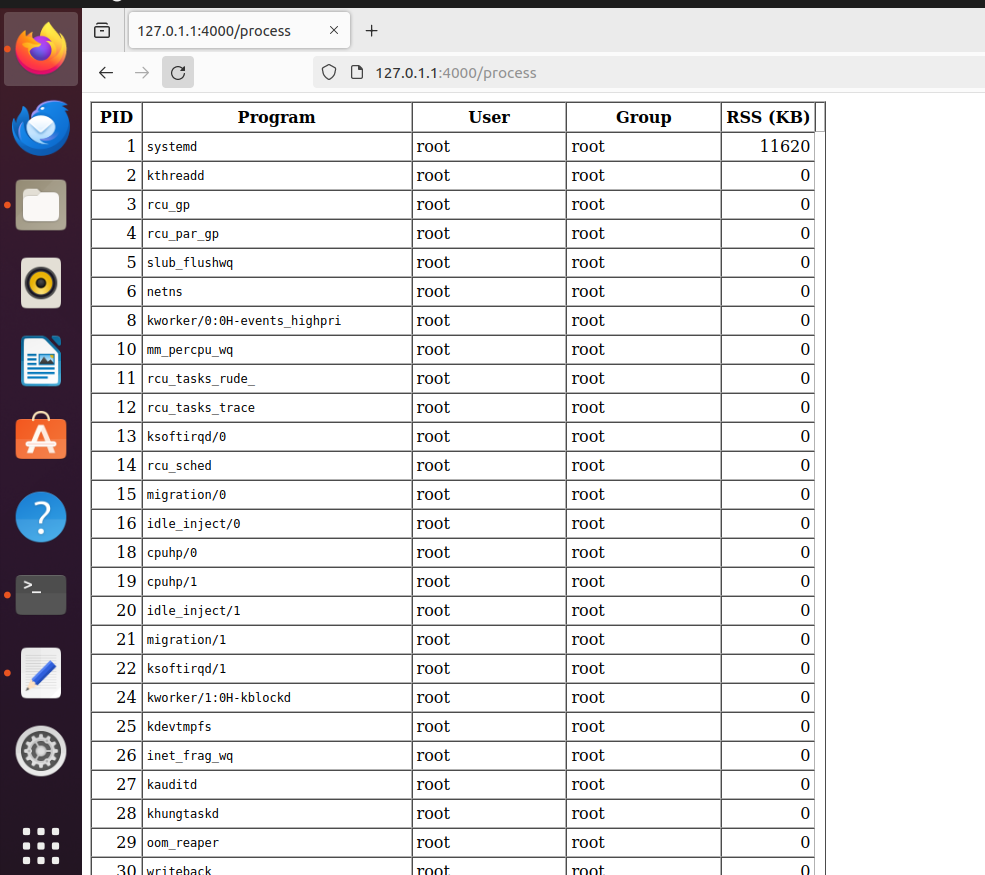


Figure 19, Running our program on terminal



Accessing the web server from a browser

Conclusion:

We successfully created a web server in Linux using C language. This server can host a web page and any kind of module can be integrated into this system in future.

References:

[1] Advanced Linux Programming

Mark Mitchell, Jeffrey Oldham, and Alex Samuel