COMPUTER NETWORK (CS F303)

LAB-SHEET – 5

Topic: Socket Programming- Concurrent Server

Learning Objectives:

Creating concurrent TCP server (handling multiple clients) using fork() and threads tem call

Handling multiple clients at the same time

There are two main classes of servers, iterative and concurrent. An iterative server iterates through each client, handling it one at a time. A concurrent server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the **fork** function, creating one child process for each client. An alternative technique is to use **threads** instead (i.e., light-weight processes).

TCPEchoServer_fork.c

A typical concurrent ECHO server (using fork has the following structure).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8898
#define MAX_CONN 10
#define BUFFER_SIZE 1024

void handle_client(int client_sock) {
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;
    // Echo data received from client
    while ((bytes_read = read(client_sock, buffer, sizeof(buffer))) > 0) {
        write(client_sock, buffer, bytes_read); // Send back the received
data
    }
    if (bytes_read == 0) {
        printf("Client disconnected.\n");
    } else if (bytes_read == -1) {
        perror("Error reading from socket");
    }
    close(client_sock);
}

int main() {
```

```
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    // Create socket
    if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket creation failed");
        exit(1);
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind socket to address
    if (bind(server_sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) == -1) {
        perror("Bind failed");
        close(server_sock);
        exit(1);
    }
    // Listen for incoming connections
    if (listen(server_sock, MAX_CONN) == -1) {
        perror("Listen failed");
        close(server_sock);
        exit(1);
    }
    printf("Server listening on port %d...\n", PORT);
    // Accept incoming connections and handle them concurrently
    while (1) {
        client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_addr_len);
        if (client_sock == -1) {
            perror("Accept failed");
            continue;
        }

printf("Client connected from %s:%d\n", inet_ntoa(client_addr.sin_addr),
           ntohs(client_addr.sin_port));
        // Create a child process to handle the client
        if (fork() == 0) {
            // Child process
            close(server_sock); // Child does not need the server socket
            handle_client(client_sock);
            exit(0);
        }
        close(client_sock); // Parent does not need the client socket
```

```
    }
    close(server_sock);
    return 0;
}
```

When a connection is established, accept returns, the **server calls fork**, and the child process services the client (on the connected new socket). The parent process waits for another connection (on the listening socket sockfd). The parent closes the connected socket since the child handles the new client.

TCPEchoServer_threads.c

A typical concurrent ECHO server (using threads has the following structure).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 8898
#define MAX_CONN 10
#define BUFFER_SIZE 1024

// Structure to pass client socket to the thread function
typedef struct {
    int client_sock;
    struct sockaddr_in client_addr;
} client_data_t;

// Thread function to handle each client
void *handle_client(void *arg) {
    client_data_t *data = (client_data_t *)arg;
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;
    // Handle client interaction
    while ((bytes_read = read(data->client_sock, buffer, sizeof(buffer))) >
0) {
        write(data->client_sock, buffer, bytes_read); // Echo the received
data
    }
    if (bytes_read == 0) {
        printf("Client disconnected from %s:%d\n", inet_ntoa(data-
>client_addr.sin_addr), ntohs(data->client_addr.sin_port));
    } else if (bytes_read == -1) {
```

```
        perror("Error reading from socket");
    }
    close(data->client_sock);
    free(data); // Free allocated memory for client data
    pthread_exit(NULL);
}
int main() {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    pthread_t thread_id;

    // Create server socket
    if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket creation failed");
        exit(1);
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);
    // Bind socket to the address
    if (bind(server_sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) == -1) {
        perror("Bind failed");
        close(server_sock);
        exit(1);
    }
    // Listen for incoming connections
    if (listen(server_sock, MAX_CONN) == -1) {
        perror("Listen failed");
        close(server_sock);
        exit(1);
    }
    printf("Server listening on port %d...\n", PORT);
    // Accept and handle client connections concurrently using threads
    while (1) {
        client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_addr_len);
        if (client_sock == -1) {
            perror("Accept failed");
            continue;
        }
        printf("Client connected from %s:%d\n",
inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
        // Allocate memory for client data and pass it to the thread
```

```
        client_data_t *data = malloc(sizeof(client_data_t));
        data->client_sock = client_sock;
        data->client_addr = client_addr;
        // Create a new thread for the client
        if (pthread_create(&thread_id, NULL, handle_client, data) != 0) {
            perror("Thread creation failed");
            free(data); // If thread creation fails, free the allocated
memory
            close(client_sock);
        }
        // Detach the thread so it can clean up resources when finished
        pthread_detach(thread_id);
    }
    close(server_sock);
    return 0;
}
```

- The server now uses **pthreads** to handle multiple client connections concurrently.
- Each time a client connects, a new thread is created with `pthread_create()`, and the thread is responsible for handling communication with that specific client.
- `client_data_t` is used to store the client's socket and address information, which is passed to the thread function.
- Threads are detached using `pthread_detach()` so that resources are automatically cleaned up once the thread finishes execution.

The main difference between using `fork()` and `pthreads` in handling multiple client connections is primarily in the concurrency model (process-based vs. thread-based), but there are some additional considerations as well.

1. Concurrency Model: Process vs. Threads

- **Using `fork()` (Process-based):**
    - Each time a client connects, the server creates a new child process using `fork()`.
    - Each process has its own memory space (each client gets a separate process, so no memory is shared between them unless explicitly managed using inter-process communication like shared memory or pipes).
    - The operating system is responsible for managing separate processes, and each process has its own stack, heap, and other resources.
    - Process creation is heavier, requiring more system resources (e.g., separate memory for each process, OS overhead for process scheduling).

- **Using `pthreads` (Thread-based):**
    - Instead of creating a new process, the server creates a new thread using `pthread_create()`.
    - All threads in a process share the same memory space. This makes thread management faster and more efficient in terms of memory and resource usage.
    - Threads are lighter than processes. They share the same address space, so they can communicate with each other easily (via shared memory).
    - Thread creation is generally faster and more efficient than process creation because there's no need for duplication of memory, file descriptors, etc.

2. Code and Resource Management Differences

Here are the key differences in terms of code and resource management:

1. Memory and Resources:

- **Forked Process:**
    - Each child process is independent with its own memory space, and the parent doesn't directly share data with the child unless explicitly handled via IPC mechanisms like pipes or shared memory.
    - Each process has a separate stack, and synchronization (if needed) must be handled through inter-process mechanisms.
- **Pthreads (Threads):**
    - Threads in a process share the same memory space, so they can easily communicate using shared variables and structures.
    - Synchronization (mutexes, condition variables) is needed to avoid data races when multiple threads access shared memory.
    - Threads are lighter in terms of resource usage compared to processes because they don't duplicate memory or system resources like file descriptors.

2. Thread/Process Lifecycle:

- **Forked Process:**
    - A new process is created when a client connects, and it runs independently. The child process exits once its task is done.
    - You need to wait for the process to complete (using `wait()` or `waitpid()`), or simply let the operating system clean up orphaned processes.
- **Pthreads (Threads):**
    - A new thread is created for each client, and the main thread can continue accepting other clients.
    - Threads are joined or detached. If you use `pthread_join()`, you can wait for the thread to complete; `pthread_detach()` allows threads to clean up automatically when done.

3. Code Differences in Handling Clients:

Fork-based Server Code:

```
if (fork() == 0) {  // Child process
    // Handle client
    handle_client(client_sock);
    exit(0);  // Child exits after handling client
}
```

- When you use `fork()`, the child process runs the code to handle the client, and the parent process goes back to accepting new connections.
- The child process exits once it's done, so there's no need for cleanup (the OS will handle that).

Thread-based Server Code:

```
pthread_t thread_id;
client_data_t *data = malloc(sizeof(client_data_t));
data->client_sock = client_sock;
data->client_addr = client_addr;
pthread_create(&thread_id, NULL, handle_client, data);
pthread_detach(thread_id);  // No need to join, thread will clean up itself
```

- Here, instead of forking, we create a thread using `pthread_create()`. The thread executes the client-handling function (`handle_client`).
- `pthread_detach()` is used to automatically clean up the thread when it's done. If we used `pthread_join()`, the parent would wait for the thread to finish.

4. Handling Synchronization:

- **Fork-based Process Server:**
    - Since each process is independent, no explicit synchronization is needed between processes. But if you're sharing data between processes (e.g., through shared memory), you will need synchronization mechanisms (semaphores, shared memory locks, etc.).
- **Thread-based Server:**
    - Because threads share the same address space, you must be cautious of race conditions. Shared data (such as global variables or structures passed to threads) should be protected using synchronization primitives like mutexes, semaphores, or condition variables.

3. Performance Considerations:

- **Forking (Processes):**
    - Forking processes is more expensive in terms of time and system resources.
    - Creating and managing processes involves more memory overhead (copy-on-write mechanism) and context-switching is heavier compared to threads.

    - If the server needs to handle a large number of clients, forking might introduce performance issues due to the overhead of creating and destroying processes.
- **Pthreads (Threads):**
    - Threads are lightweight and faster to create and manage, with lower overhead.
    - Since threads share memory, communication between threads is faster (they can directly access shared variables).
    - However, the developer must ensure thread safety, especially when modifying shared data.

4. Scalability:

- **Forking:**
    - Scaling with `fork()` can be less efficient, especially with many clients, due to the overhead of managing multiple processes.
    - On systems with limited process resources (like file descriptors), you may hit system limits more quickly.
- **Pthreads:**
    - Threads are more scalable for a larger number of concurrent clients due to their lightweight nature.
    - Thread-based servers generally provide better performance and scalability for applications requiring high concurrency.

5. Error Handling and Debugging:

- **Forking:**
    - Each child process runs independently, so debugging and error handling can be trickier. You need to handle errors across multiple processes, often using signals and inter-process communication.
- **Pthreads:**
    - Debugging is easier because threads share memory space, but the synchronization of threads can lead to subtle issues like race conditions or deadlocks. Proper error handling mechanisms (like checking `pthread_create()` returns) must be in place.

Switching from `fork()` to `pthreads` mainly affects **how concurrency is handled**. While both methods achieve similar functionality in a TCP server (handling multiple clients concurrently), using threads (`pthreads`) provides **lower overhead** and **more efficient memory usage**. Threads are typically preferred for higher-performance and scalable server applications. However, they require more careful synchronization and attention to shared memory access to avoid issues like data races.

To summarize:

- **Using** `fork()`: Processes, heavier memory footprint, independent address spaces, suitable for lower concurrency.
- **Using** `pthreads`: Threads, lighter memory footprint, shared memory, suitable for high concurrency and better resource utilization.

Note: you need to use -**pthread** option while compiling the **TCPEchoServer_threads.c** code**.**
The **`-pthread`** option is necessary for**:**

1. **Defining threading-related macros** for thread safety during compilation.
2. **Linking to the** `pthread` **library** during the linking phase of the compilation process.

Thus, it ensures that the program is both compiled and linked correctly to support threads.

Exercise #1

a. Use and modify the TCP Echo client created in Lab3 to create a client which connects to either of the servers.
b. Use and modify to TCPEchoServer_fork.c and TCPEchoServer_threads.c to create a concurrent TCPEchoServer_ft.c which can either run using *fork* or *thread* mode (you may use command line arguments to specify the mode)
c. Test the server program created above using the TCP Echo client program.
d. Which server mode do you think is more efficient? Justify you answer.

Exercise#2

Develop a simple **key-value () store** using TCP sockets where clients connect simultaneously to the server for inserting and retrieving the key. Your application must support the following features:

- The client takes user input to **get/put/delete keys** and passes these requests over the server.
- The server maintains a data structure of **key-value** pairs in a file (**database.txt**) as the persistent database and returns a suitable reply to the server.
- When the request from the client is of the form **"put key value",** the server must store the key-value pair in its data structures, provided the key does not already exist. Put requests on an existing key must return an error message.
- When the client sends a request of the form **"get key"**, the value stored against the key must be returned to the client. If the key is not found, a suitable error message must be returned by the server.
- The client should be able to issue a delete request (on an existing key) as follows: **"del key"**.
- For all requests that succeed, the server must either return the value (in case of get), or the string **"OK"** (in case of put/del).
- You may assume that all keys are integers and the corresponding values are strings. When the user inputs "Bye", the server should reply **"Goodbye"** and close the client connection.
- Upon receiving this message from the server, your client program terminates. However, the server must keep the database in a file for subsequent run of client program.
- Note that your server must be able to communicate with multiple active clients at a time. When multiple clients talk to the server, one client should be able to see the data stored by the other clients.

xx—oo—xx