WebServ Evaluation Questions & Answers

HTTP Server Basics

An HTTP server works by:

1. Listening for incoming TCP connections on specified ports (commonly 80 for HTTP)
2. Accepting client connections and receiving HTTP request messages
3. Parsing the request (method, URI, headers, body)
4. Processing the request based on its method (GET, POST, DELETE, etc.)
5. Generating an appropriate HTTP response (status code, headers, body)
6. Sending the response back to the client
7. Either closing the connection or keeping it alive for future requests

Our implementation follows these principles with a non-blocking I/O model, allowing multiple concurrent client connections to be handled by a single process.

I/O Multiplexing Function

We use the `poll()` function for I/O multiplexing. This choice was made because:

- It scales better than `select()` when handling many file descriptors
- It's easier to use with dynamic sets of file descriptors
- It provides more detailed information about the events that occurred

How poll() Works

`poll()` works as follows:

1. It takes an array of `struct pollfd` structures, where each structure contains:

    - `fd` : The file descriptor to monitor
    - `events` : The events to watch for (POLLIN for reading, POLLOUT for writing)
    - `revents` : The events that actually occurred (filled by poll() upon return)

2. It also takes a timeout value:

    - Negative: Wait indefinitely until an event occurs
    - Zero: Return immediately (non-blocking poll)
    - Positive: Wait up to the specified number of milliseconds

3. It blocks until:

    - Events occur on one or more monitored file descriptors
    - The timeout expires
    - The call is interrupted by a signal

4. Upon return, it lets us know which file descriptors are ready for reading or writing by setting bits in the `revents` field of each pollfd structure.

Managing Server Accept and Client Read/Write

We use only one `poll()` call in our main loop to manage both server sockets (for accepting new connections) and client sockets (for reading/writing data).

Here's how we handle it:

1. We add all server listening sockets to the poll array with POLLIN
2. We add all client sockets to the poll array with POLLIN and/or POLLOUT depending on their state
3. We call poll() to wait for events on any of these sockets
4. When poll() returns, we check each socket with events:
    - If it's a server socket with POLLIN, we accept a new connection
    - If it's a client socket with POLLIN, we read data from it
    - If it's a client socket with POLLOUT, we write data to it

This approach allows us to efficiently handle multiple clients with a single thread, without blocking.

poll() in Main Loop

Our main loop in `Server::start()` contains a single `poll()` call that checks for both reading and writing at the same time. The code structure is:

```
while (_running) {
    // Check for both read and write events
    int ready = poll(_poll_fds, _poll_count, timeout);

    for (int i = 0; i < current_poll_count; ++i) {
        if (current_poll_fds[i].revents == 0) continue;

        if (current_poll_fds[i].revents & POLLIN) {
            // Handle reading (either accept a new connection or read client data)
            if (is_server_socket(current_poll_fds[i].fd)) {
                handleNewConnection(current_poll_fds[i].fd);
            } else {
                handleClientData(current_poll_fds[i].fd);
            }
        } else if (current_poll_fds[i].revents & POLLOUT) {
            // Handle writing to client
            handleClientWrite(current_poll_fds[i].fd);
        } else if (current_poll_fds[i].revents & (POLLERR | POLLHUP | POLLNVAL)) {
            // Handle errors
            removeClient(current_poll_fds[i].fd);
        }
    }
}
```

This approach ensures we handle both reading and writing within the same event loop, which is more efficient than having separate loops for each operation.

Read/Write Per Client Per poll()

After `poll()` indicates a socket is ready, we perform at most one read or one write per client. The relevant code in `handleClientData()` and `handleClientWrite()` shows this:

In `handleClientData()`:

```
void Server::handleClientData(int client_fd) {
    char buffer[4096];
    ssize_t bytes_read = recv(client_fd, buffer, sizeof(buffer) - 1, 0);

    if (bytes_read <= 0) {
        removeClient(client_fd);
        return;
    }

    // Process the read data...
}
```

In `handleClientWrite()`:

```
void Server::handleClientWrite(int client_fd) {
    Response& response = _client_responses[client_fd];
    const std::string& data = response.getData();

    // Calculate how much to send...

    ssize_t bytes_sent = send(client_fd, data.c_str() + bytes_sent_so_far,
                              data.length() - bytes_sent_so_far, 0);
    if (bytes_sent == -1) {
        removeClient(client_fd);
        return;
    }

    // Update the response state...
}
```

Each function performs exactly one read or write per call, and is only called once per client per `poll()` iteration.

Error Handling for Socket Operations

We check for errors in all socket operations and properly remove clients when errors occur:

Read/Recv Errors

```
ssize_t bytes_read = recv(client_fd, buffer, sizeof(buffer) - 1, 0);
if (bytes_read <= 0) {
    removeClient(client_fd);
    return;
}
```

Write/Send Errors

```
ssize_t bytes_sent = send(client_fd, data.c_str() + bytes_sent_so_far,
                         data.length() - bytes_sent_so_far, 0);
if (bytes_sent == -1) {
    removeClient(client_fd);
    return;
}
```

We check for both error returns (-1) and EOF conditions (0 for recv) to ensure robust error handling.

Proper Return Value Checking

We properly check return values from socket operations:

- For `recv()`, we check for:
  - `-1` : Error occurred
  - `0` : Connection closed by peer
  - `> 0` : Data received successfully
- For `send()`, we check for:
  - `-1` : Error occurred
  - `>= 0` : Bytes successfully sent (may be less than requested)

No Direct errno Checking

We don't directly check errno after read/write operations. Instead, we use the return values from the system calls to determine success or failure. This is a cleaner approach that avoids potential race conditions with the errno global variable.

I/O Only Through poll()

All I/O operations on sockets are performed only after `poll()` indicates the socket is ready. We never read from or write to a socket unless poll() has indicated it's ready for that operation.

This ensures we never block on I/O operations and properly respect the event-driven architecture.

poll() Flow Diagram

Here's a visual representation of our poll-based I/O flow:

```
┌─────────────────────────────────────────────────────────────┐
│                      Main Server Loop                        │
└────────────────────────────┬────────────────────────────────┘
                             │
                             ▼
┌─────────────────────────────────────────────────────────────┐
│  poll(_poll_fds, _poll_count, timeout)                       │
│  Wait for socket events (POLLIN, POLLOUT, POLLERR, etc.)     │
└────────────────────────────┬────────────────────────────────┘
                             │
                             ▼
┌─────────────────────────────────────────────────────────────┐
│  Check revents for each socket                               │
└───────────┬─────────────────┬─────────────────┬─────────────┘
            │                 │                 │
            ▼                 ▼                 ▼
┌───────────────────┐ ┌─────────────────┐ ┌───────────────────┐
│ POLLIN (Read)     │ │ POLLOUT (Write) │ │ POLLERR/POLLHUP   │
└─────────┬─────────┘ └────────┬────────┘ └─────────┬─────────┘
          │                    │                    │
          ▼                    ▼                    ▼
┌───────────────────┐ ┌─────────────────┐ ┌───────────────────┐
│ handleClientData  │ │ handleClientWrite│ │ removeClient      │
└─────────┬─────────┘ └────────┬────────┘ └───────────────────┘
          │                    │
          ▼                    ▼
┌───────────────────┐ ┌─────────────────┐
│ Read data with    │ │ Write data with │
│ recv()            │ │ send()          │
└─────────┬─────────┘ └────────┬────────┘
          │                    │
          ▼                    ▼
┌───────────────────┐ ┌─────────────────┐
│ Append to request │ │ Update bytes sent│
└─────────┬─────────┘ └────────┬────────┘
          │                    │
          ▼                    ▼
┌───────────────────┐ ┌─────────────────┐
│ If request        │ │ If response      │
│ complete:         │ │ complete:        │
│ → Process request │ │ → Reset for next │
│ → Prepare response│ │   request or     │
│ → Add POLLOUT     │ │ → Close if needed│
└───────────────────┘ └─────────────────┘
```

Client Socket State Machine

```
          ┌────────────────────────────────────────────┐
          │                                            │
          │                                            │
          ▼                                            │
┌────────────────────┐     ┌──────────────────────┐   │
│                    │     │                      │   │
│   POLLIN only      │────►│   POLLIN | POLLOUT   │───┘
│  (Reading state)   │     │  (Reading+Writing)   │
│                    │◄────│                      │
└────────────────────┘     └──────────────────────┘

         ▲                            │
         │                            │
         │                            ▼

┌────────────────────┐     ┌──────────────────────┐
│  New Connection    │     │                      │
│                    │     │  Connection Closed   │
└────────────────────┘     └──────────────────────┘
```

Data Flow for a Complete HTTP Exchange

```
┌──────────┐                                  ┌──────────┐
│  Client  │                                  │  Server  │
└────┬─────┘                                  └────┬─────┘
     │                                             │
     │  TCP connection                             │
     ├────────────────────────────────────────────►
     │                                             │
     │  HTTP request (might be in multiple chunks) │
     ├────────────────────────────────────────────► POLLIN
     │                                             │ recv() ←┐
     │                                             │         │ until request
     │                                             │         │ is complete
     │                                             │         ┘
     │                                             │
     │               process request               │
     │                                             │
     │                                             │ add POLLOUT
     │                                             │
     │  HTTP response (might be in multiple chunks)│
     │◄─────────────────────────────────────────────┤ POLLOUT
     │                                             │ send() ←┐
     │                                             │         │ until response
     │                                             │         │ is complete
     │                                             │         ┘
     │                                             │
     │  Close connection or ready for next request │
     │◄─────────────────────────────────────────────┤ switch to POLLIN
     │                                             │ or close socket
```

This event-driven approach allows the server to efficiently handle thousands of connections simultaneously with a single thread by only performing I/O operations when the kernel confirms they're ready.

Compilation

The project compiles cleanly without any re-link issues. We use a proper Makefile with appropriate dependencies to ensure clean compilation.