MARKDOWN

```
# 🎯 Project Charter: HTTP Server (Basic)

## What You Are Building

A production-aware, standalone HTTP/1.1 static file server built from raw TCP sockets. You are creating a systems-level tool that maps URL paths to a local filesystem, detects MIME types, prevents security exploits like directory traversal, and manages multiple simultaneous clients using a bounded thread pool. By the end, you will have a high-performance binary capable of serving a modern website to multiple browsers concurrently.

## Why This Project Exists

Modern web development happens behind layers of framework abstractions that treat the network and the OS as "magic." Building an HTTP server from scratch exposes the "physics" of the web: how TCP byte streams are framed into messages, how the kernel manages file descriptors, and why threading models are the difference between a responsive service and a crashed one.

## What You Will Be Able to Do When Done

- **Master Socket Programming:** Implement the full TCP lifecycle (bind, listen, accept) and handle partial network reads/writes.

- **Implement Protocols:** Write a robust, adversarial-safe parser for HTTP/1.1 requests following RFC 7230.

- **Enforce Systems Security:** Implement path canonicalization to prevent "Directory Traversal" attacks that leak sensitive system files.

- **Architect Concurrency:** Build a work-queue based thread pool from scratch to handle hundreds of parallel connections.

- **Manage Resource Lifecycle:** Implement graceful shutdowns and connection "Keep-Alive" to maximize server efficiency.

## Final Deliverable

A single executable (written in C, Go, or Rust) composed of ~1,500 to 2,500 lines of code across modules for networking, parsing, and file I/O. The server will pass a "stress test" of 10,000 requests without leaking file descriptors and will serve a complex directory of HTML, CSS, JS, and binary images to any standard web browser.

## Is This Project For You?

**You should start this if you:**

- Understand pointers and manual memory management (if using C).
```

- Are comfortable with basic File I/O (open, read, write).

- Want to know exactly what happens when you type a URL into a browser.

- Are ready to debug low-level network issues using `curl`, `telnet`, and `netstat`.

Come back after you've learned:

- [TCP/IP Basics](https://beej.us/guide/bgnet/): Understanding IP addresses and port numbers.

- Basic C/Systems Syntax: If you don't know the difference between the stack and the heap, this will be frustrating.

Estimated Effort

| Phase | Time |

|-------|------|

| **Phase 1: TCP Server & HTTP Response** | ~3 hours |

| **Phase 2: HTTP Request Parsing** | ~4 hours |

| **Phase 3: Static File Serving & Security** | ~6 hours |

| **Phase 4: Thread Pool & Concurrency** | ~7 hours |

| **Total** | **~20 hours** |

Definition of Done

The project is complete when:

- The server successfully serves an `index.html` file and an image to a browser (Chrome/Firefox).

- `curl -I` returns correct headers including `Content-Type`, `Content-Length`, and `Date`.

- A request for `../../etc/passwd` returns a `403 Forbidden` or `404 Not Found`, never the file content.

- The server handles 50 concurrent requests simultaneously without dropping connections.

- The server shuts down gracefully on `SIGINT` (Ctrl+C), closing all sockets and joining all threads without leaks.

HTTP Server (Basic) — Build Your Own Static File Server

This project builds a fully functional HTTP/1.1 static file server from raw TCP sockets upward. You will implement the complete network stack lifecycle — binding, listening, accepting, reading, parsing, responding — then layer on HTTP protocol parsing per RFC 7230, secure static file serving with directory traversal prevention, and finally concurrent connection handling via a bounded thread pool with keep-alive and graceful shutdown.

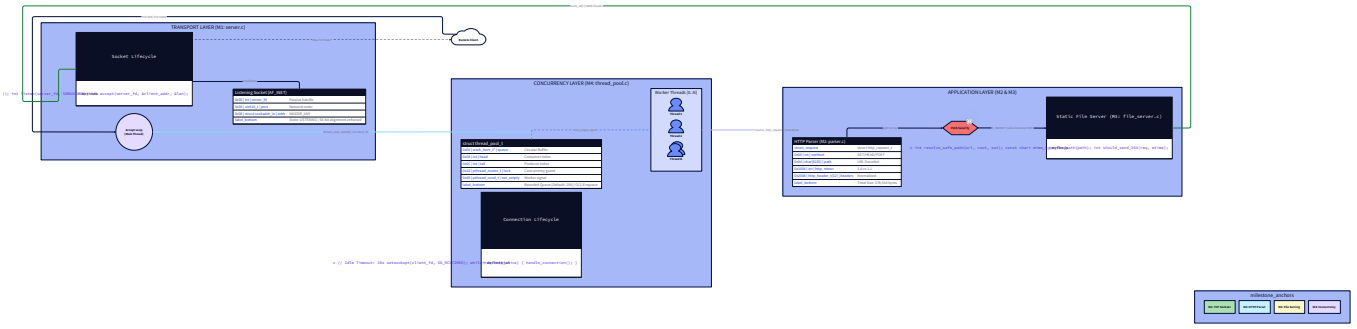
The project deliberately avoids all frameworks and HTTP libraries, forcing you to negotiate directly with the operating system's socket API, the filesystem, and the threading primitives. Every abstraction that web frameworks hide — partial reads, CRLF parsing, MIME detection, path canonicalization, connection lifecycle management — becomes visible and tangible.

By the end, you will have a production-aware mental model of what happens between a browser hitting Enter and pixels appearing on screen, grounded in syscalls, file descriptors, and cache lines rather than framework magic.

Milestone 1: TCP Server & HTTP Response

Where You Are in the System

Before you write a single line of code, orient yourself. You are about to build the absolute foundation of an HTTP server — the part that everything else rests on. In the full architecture of this project, you are building the **entry point**: the socket machinery that catches raw network connections before HTTP parsing, file serving, or concurrency concerns exist at all.



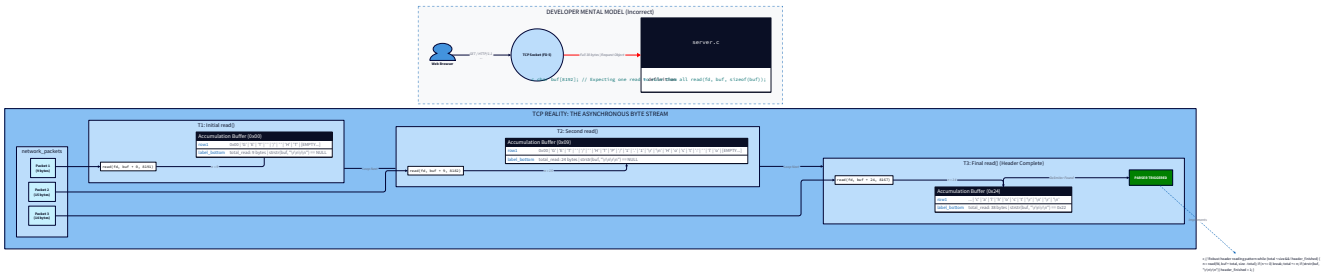
This milestone has one job: establish the plumbing. By the end, your server will open a network port, wait for a browser or `curl` to connect, receive raw bytes, and send back a hardcoded HTTP response. No

routing, no file reading, no concurrency — just the essential loop that every production web server on Earth runs underneath its abstractions.

The Fundamental Tension: TCP Is a Byte Pipe, Not a Message Pipe

Here's the misconception that breaks more networking code than any other: **"The client sends a request, so my `read()` call returns a complete request."** It feels obvious. You type a URL into a browser, it sends one HTTP request, your server should receive one HTTP request. The unit of thinking is a *message*. So naturally, `read()` gives you the message, right? Wrong. And understanding *why* this is wrong is the most important thing you will learn in this entire milestone. **TCP (Transmission Control Protocol) is a byte stream protocol.** The kernel has no concept of "messages." When the browser sends `GET / HTTP/1.1\r\nHost: localhost\r\n\r\n` — those 38 bytes — your kernel might deliver them in one `read()` call. Or three. Or seven. The kernel is working at the level of **receive buffer segments**, not HTTP messages. The factors that determine what each `read()` returns include:

- **Network packet fragmentation** — IP packets have a Maximum Transmission Unit (MTU) of ~1500 bytes on Ethernet. Large HTTP headers might span multiple packets.
- **TCP Nagle's algorithm** — Small writes from the client may be coalesced (batched) before transmission, or may not.
- **Kernel scheduling** — Your `read()` runs when the OS schedules your process. The receive buffer might have accumulated 1 byte or 8KB since your last read.
- **Localhost vs. real network** — On localhost, both processes share the same kernel and network stack, so data often arrives in one call. In production over a real network, partial delivery is the norm. This is the exact trap the Architect flagged. Code that passes every localhost test silently breaks on a production network because the testing environment hides the byte-stream nature of TCP.



The consequence is concrete: you must accumulate bytes in a buffer and search for the `\r\n\r\n` delimiter yourself. The HTTP specification says headers end with a blank line — that's `\r\n\r\n` in raw bytes. Your

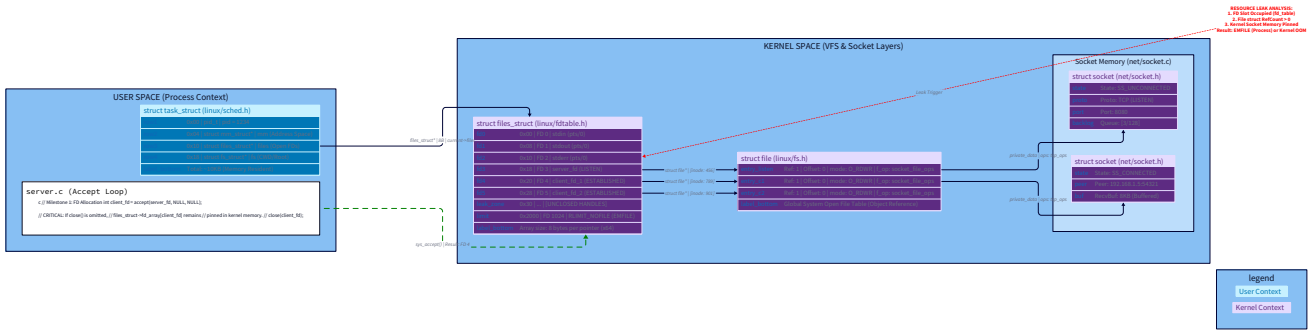
job is to keep calling `read()` until you've seen that sequence, appending each result into a growing buffer. Only then do you have a complete HTTP request header section. This one insight — TCP is a byte stream, HTTP is a structured message, and you must bridge the gap — is the foundation for the next three milestones and for virtually all network programming you will ever do.

File Descriptors: Your Handle to the OS

Before touching socket code, you need to understand **file descriptors** (FDs), because every socket you create is one. In Unix/Linux, the operating system manages access to hardware and resources on behalf of your program. Your program can't directly manipulate a network port or a disk file — it asks the kernel to do it, and the kernel hands back a small non-negative integer called a **file descriptor**. This integer is your process's handle to an open resource. Think of file descriptors as numbered slots in a per-process table the kernel maintains. Slot 0 is always `stdin` (keyboard input), slot 1 is `stdout` (terminal output), slot 2 is `stderr` (error output). When you open a file, create a socket, or create a pipe, the kernel allocates the next available slot and returns that number to you.

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);

// fd might be 3, 4, 5... — the kernel chose the next available slot
```

The crucial implication: **when you're done with a resource, you must call `close(fd)`.** If you forget, the kernel keeps the slot occupied. A typical Linux process can only have ~1024 file descriptors open at once (the soft limit; configurable with `ulimit -n`). A server that handles 1000 connections without closing them hits this ceiling and starts failing `accept()` calls with `EMFILE` ("too many open files"). This is a real production failure mode, not a theoretical concern.



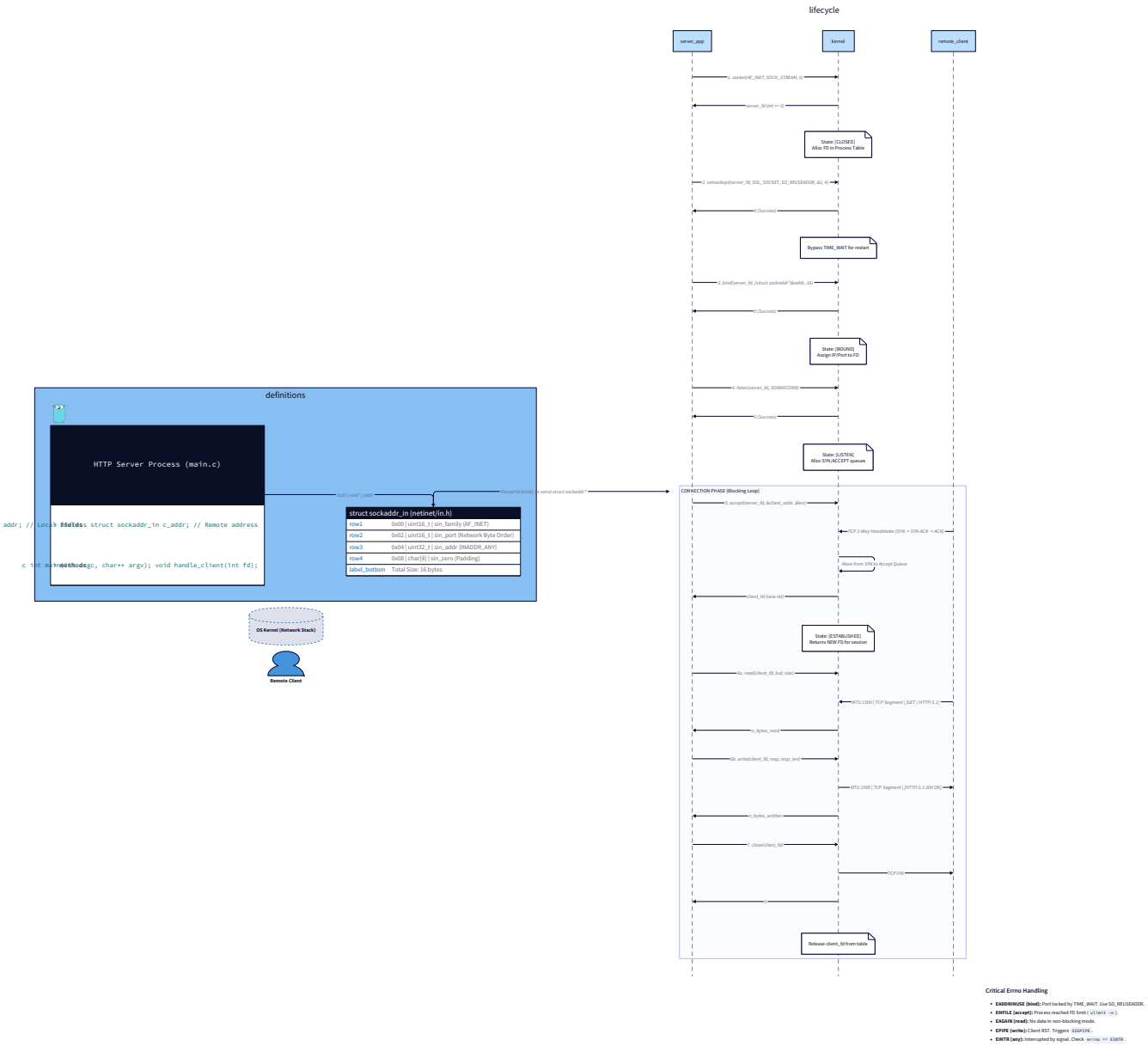
The reason file descriptors matter right now: your server will manage *two* kinds of file descriptors simultaneously:

- The **listening socket** — one FD, open for the server's lifetime, represents "this port is accepting connections"
- **Client connection sockets** — one FD *per accepted connection*, must be `close()`d after each response

---

The Seven-Step Socket Dance

Creating a TCP server requires exactly seven system calls in the right order. Each one transforms the state of your server and its connection to the OS. There's no skipping steps — the kernel enforces the order.



Let's walk through each step with its why.

Step 1: `socket()` — Create an Endpoint

```c
int server_fd = socket(AF_INET, SOCK_STREAM, 0);

if (server_fd < 0) {

    perror("socket");

    exit(1);

}
```

`socket()` creates a new socket and returns its file descriptor. The three parameters tell the kernel what kind of socket:

- `AF_INET` — Address Family Internet (IPv4). Use `AF_INET6` for IPv6.
- `SOCK_STREAM` — Stream socket, meaning TCP: ordered, reliable byte delivery. The alternative `SOCK_DGRAM` is UDP.
- `0` — Protocol. 0 means "pick the default for this family/type combo," which is TCP here. At this point, you have a socket, but it's not associated with any port or address. It's like having a phone but no phone number.

Step 2: `setsockopt()` — Enable Port Reuse

```c
int opt = 1;

setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

This step is optional but practically mandatory. Without it, after you stop and restart your server, the OS may reject `bind()` for 1–4 minutes with `EADDRINUSE`. This happens because TCP has a **TIME_WAIT** state: when a connection closes, the OS keeps the port reserved briefly to catch any delayed packets still in transit. `SO_REUSEADDR` tells the kernel "let me reuse this address even if there are TIME_WAIT connections on it." Every production server sets this. `SOL_SOCKET` means "this option applies to the socket layer itself" (as opposed to TCP-level or IP-level options). `SO_REUSEADDR` is the option name. The `&opt` is a pointer to the value — `1` means enable.

Step 3: `bind()` — Claim a Port

```c
struct sockaddr_in addr;

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

addr.sin_family = AF_INET;

addr.sin_addr.s_addr = INADDR_ANY;  // Listen on all interfaces

addr.sin_port = htons(8080);        // Port 8080, in network byte order

if (bind(server_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {

    perror("bind");

    exit(1);

}
```

`bind()` associates your socket with a specific IP address and port number. After this call, port 8080 on this machine is claimed by your process. The `struct sockaddr_in` encodes IPv4 address information. **Network byte order**: The `htons()` call is critical. CPUs can store multi-byte numbers in two ways: **big-endian** (most significant byte first, like humans write numbers) or **little-endian** (least significant byte first, which most modern x86 CPUs use). The network protocol standard mandates big-endian, called "network byte order." `htons()` — "host to network short" — converts a 16-bit port number from your CPU's native order to network byte order. Without it, you'd bind to port 36895 (the byte-flipped version of 8080) instead of 8080. `INADDR_ANY` (value 0) means "accept connections on any local IP address" — useful when your machine has multiple network interfaces.

Step 4: `listen()` — Start Accepting Connections

```c
if (listen(server_fd, SOMAXCONN) < 0) {

    perror("listen");

    exit(1);

}
```

`listen()` marks the socket as passive — it will receive incoming connections rather than initiate them. The second argument, the **backlog**, specifies how many pending connections the kernel should queue while your server is busy in `accept()`. `SOMAXCONN` is a constant (typically 128 or 4096 depending on OS) that requests the system maximum. For a production server, you'd tune this, but `SOMAXCONN` is correct for now. After `listen()`, your socket is in the **listening state**. The kernel's TCP state machine handles the three-way

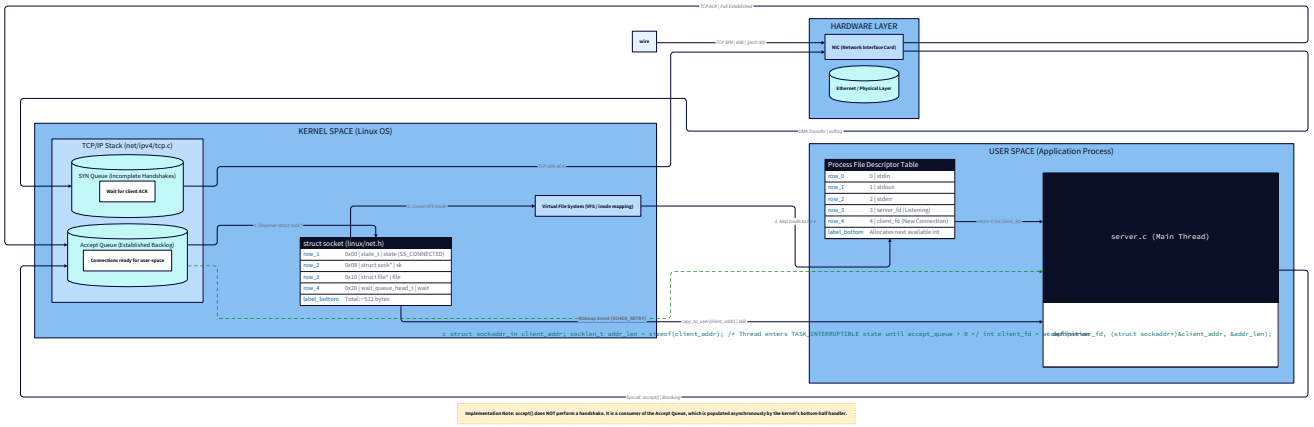
handshake (SYN → SYN-ACK → ACK) automatically for incoming clients. By the time `accept()` returns to you, the connection is already established at the TCP level — your code never sees raw TCP handshakes.

Step 5: `accept()` — Receive a Connection

```c
struct sockaddr_in client_addr;

socklen_t client_len = sizeof(client_addr);

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

if (client_fd < 0) {

    perror("accept");

    continue;  // Don't exit — try next connection

}
```

`accept()` blocks (pauses your program) until a client connects. When a client connects, it returns a **new file descriptor** — `client_fd` — that represents the specific connection to that client. Your original `server_fd` remains untouched, still listening for future connections. This is architecturally important: you always have two distinct roles:

- `server_fd` — the *listening* socket (one, permanent)
- `client_fd` — the *connection* socket (one per client, temporary) The `client_addr` is filled in by the kernel with the client's IP and port. You can inspect it to log where connections come from.



Step 6: `read()` / `recv()` and `write()` / `send()` — Communicate

Now you have `client_fd`. This is a bidirectional byte stream. You read from it (bytes the client sent) and write to it (bytes you're sending back). The partial-read problem lives here. We'll dedicate a full section to this below.

Step 7: `close()` — Release the Connection

```c
close(client_fd);  // MANDATORY after each connection
```

After your response is sent, close the client socket. This signals to the client that you're done (TCP FIN handshake), and releases the kernel file descriptor. Never skip this.

Three-Level View: What Happens When You Call `accept()`

To truly understand the socket lifecycle, look at all three levels simultaneously. **Level 1 — Your Code:** You call `accept(server_fd, ...)` and your program pauses. Eventually it returns with `client_fd = 5`. **Level 2 — OS/Kernel:** The kernel maintains two queues for your listening socket:

- **SYN queue** (incomplete connections): TCP handshakes in progress
- **Accept queue** (complete connections): fully-established connections waiting for your `accept()` call When a client's SYN arrives, the kernel adds it to the SYN queue, sends SYN-ACK, and waits for ACK. When ACK arrives, the connection moves to the Accept queue. Your `accept()` call dequeues the next entry from the Accept queue, creates a new socket file descriptor for it, and returns. If the Accept queue is empty, `accept()` blocks until it's not. **Level 3 — Hardware:** The network interface card (NIC) receives an Ethernet frame containing an IP packet containing a TCP segment. The NIC raises a hardware interrupt, the interrupt handler (running in kernel context) processes the TCP segment, updates the connection state machine, and places the data into the socket's receive buffer in kernel memory. The kernel wakes your `accept()` call by transitioning your thread from the "sleeping" state to "runnable." The hardware-to-userspace path: NIC → DMA into kernel buffer → TCP stack processes segment → wakes sleeping process → your code runs. This entire path takes on the order of 10–50 microseconds on a local network.

The Partial Read Problem — Solved

Now let's implement the read loop properly. The design goal: accumulate bytes from `client_fd` until we've seen `\r\n\r\n` (the HTTP header terminator), or until we've read a maximum safe amount.

```c
#define BUFFER_SIZE 8192

// Returns the number of bytes in the buffer on success, -1 on error/disconnect
// Sets *header_end to the position after \r\n\r\n if found, -1 if not found
ssize_t read_http_request(int client_fd, char *buf, size_t buf_size, int *header_end) {
    size_t total_read = 0;
    *header_end = -1;
    while (total_read < buf_size - 1) {  // Leave room for null terminator
        ssize_t n = read(client_fd, buf + total_read, buf_size - 1 - total_read);
        if (n < 0) {
            // EAGAIN/EWOULDBLOCK: no data yet (non-blocking mode, not our case here)
            // Other errors: real problem
            perror("read");
            return -1;
        }
        if (n == 0) {
            // Client disconnected before sending complete request
            return -1;
        }
        total_read += n;
        buf[total_read] = '\0';  // Null-terminate for string searching
        // Scan for the end-of-headers delimiter
        char *end = strstr(buf, "\r\n\r\n");
        if (end != NULL) {
            *header_end = (int)(end - buf) + 4;  // Position after the delimiter
            return (ssize_t)total_read;
        }
    }
```

```
    // Buffer full without finding delimiter — request too large
    return -1;
}
```

Walk through this carefully:

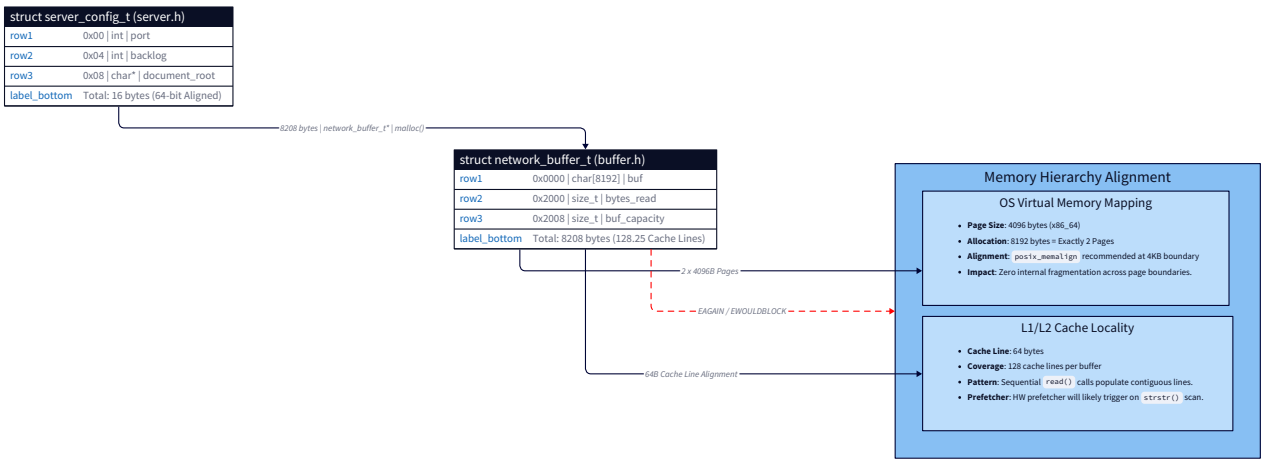
- **`read()` return values**: positive means "I read this many bytes," zero means "the client closed the connection," negative means "an error occurred." You must handle all three.
- **`buf + total_read`**: each call reads into the *next unused portion* of the buffer. This is how accumulation works — each call appends to where the last one left off.
- **`buf_size - 1 - total_read`**: we tell `read()` the maximum it can give us, shrinking as the buffer fills. The `- 1` reserves space for the null terminator we add after each read.
- **`strstr(buf, "\r\n\r\n")`**: scan the entire accumulated buffer for the delimiter. We scan from the start each time (slightly inefficient, but correct and simple for now).
- **Buffer size limit**: 8KB is the default maximum URI length in many HTTP servers (and the nginx default). Requests larger than this get rejected, which we'll implement properly in Milestone 2. This is the correct mental model: **`read()` is just "give me whatever bytes the kernel has right now."** Your code is responsible for interpreting those bytes as a structured protocol.

Building the Complete Server

Now let's assemble everything into a working server. We'll structure the code cleanly from the start — the habits you form here carry through all four milestones.

Data Structures and Memory Layout

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <signal.h>

#include <time.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define DEFAULT_PORT      8080

#define BACKLOG           SOMAXCONN

#define REQUEST_BUF_SIZE  8192

#define MAX_RESPONSE_SIZE 4096
```



```c
typedef struct {
    int      port;       // TCP port to bind
    int      server_fd;  // Listening socket file descriptor
} server_config_t;
```

Keep the struct minimal for Milestone 1. Later milestones will expand it with document root, thread pool size, and timeout configuration.

The Hardcoded HTTP Response

In Milestone 1, you're not reading files yet — you're sending a fixed response to prove the plumbing works. But the response must be a *valid* HTTP/1.1 response. Let's understand why each header matters.

```c
// Build a valid HTTP/1.1 200 OK response with Date header
// Returns bytes written to `buf`, or -1 on error
int build_hardcoded_response(char *buf, size_t buf_size) {
    // HTTP Date header format: "Day, DD Mon YYYY HH:MM:SS GMT"
    time_t now = time(NULL);
    struct tm *gmt = gmtime(&now);
    char date_str[64];
    strftime(date_str, sizeof(date_str), "%a, %d %b %Y %H:%M:%S GMT", gmt);
    const char *body =
        "<html><body>"
        "<h1>Hello from my HTTP server!</h1>"
        "</body></html>";
    int body_len = (int)strlen(body);
    int n = snprintf(buf, buf_size,
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html; charset=utf-8\r\n"
        "Content-Length: %d\r\n"
        "Date: %s\r\n"
        "Connection: close\r\n"
        "\r\n"
        "%s",
        body_len,
        date_str,
        body);
    return (n > 0 && (size_t)n < buf_size) ? n : -1;
}
```

Each header serves a specific role:

- **`HTTP/1.1 200 OK`** — The status line. `HTTP/1.1` is the protocol version. `200` is the numeric status code. `OK` is the human-readable reason phrase. The browser uses the numeric code; the phrase is for humans and logs.
- **`Content-Type: text/html; charset=utf-8`** — Without this, browsers guess the content type. Some browsers display HTML as raw text if they guess wrong. `charset=utf-8` specifies character encoding.
- **`Content-Length: N`** — The exact byte count of the body. The browser needs this to know where the response ends. Without it, the browser has to wait for the connection to close to know if more data is coming.
- **`Date:`** — Required by HTTP/1.1 (RFC 7231, Section 7.1.1.2). Servers should include it. We generate it from the system clock in UTC.
- **`Connection: close`** — Tells the client we'll close the connection after this response. In Milestone 4, you'll implement `keep-alive` to reuse connections. For now, `close` is correct.
- **Blank line (`\r\n`)** — Mandatory separator between headers and body. Every HTTP message has this blank line; without it, the body would be treated as another header.

Writing the Response — Partial Writes

Just as `read()` might give you fewer bytes than requested, `write()` might send fewer bytes than you asked. This is uncommon on localhost but can happen when the kernel's send buffer is full (which happens under load). The fix is the same: loop until all bytes are sent.

```c
// Write all `len` bytes from `buf` to `fd`.
// Returns 0 on success, -1 on error.
int write_all(int fd, const char *buf, size_t len) {
    size_t total_written = 0;
    while (total_written < len) {
        ssize_t n = write(fd, buf + total_written, len - total_written);
        if (n <= 0) {
            return -1;  // Error or connection closed
        }
        total_written += n;
    }
    return 0;
}
```

This pattern — the "write-all" or "write loop" — is so common it appears in virtually every systems library. Python's `socket.sendall()`, Go's `io.WriteString()`, and Rust's `write_all()` trait method all implement exactly this loop so you don't have to.

Handling SIGPIPE — Don't Let the Client Kill Your Server

Here's a specific failure mode you must handle before your first test. **The scenario**: a client connects, you call `write()` to send the response, but the client has already disconnected (crashed, closed the tab, hit Ctrl+C). The client's side of the TCP connection is closed. What happens when you write to a socket whose remote end is closed? On Linux, the first `write()` succeeds (the kernel doesn't know yet that the client is gone). The kernel sends the data, the remote TCP stack sends back a RST (reset) packet. On your *second* `write()` to that socket, the kernel has now processed the RST and knows the connection is dead. Instead of returning an error from `write()`, the kernel delivers **SIGPIPE** — signal 13 — to your process. The default handler for SIGPIPE **terminates your process.** That's right: a single misbehaving client can kill your server.

The fix is to ignore SIGPIPE:

```c
// At server startup, before any accept() calls:

signal(SIGPIPE, SIG_IGN);
```

After this, when you try to write to a closed socket, `write()` returns -1 with `errno = EPIPE` instead of killing your process. You check the return value (you always check return values in systems code), log it, and close the client socket gracefully. An alternative when using `send()` instead of `write()`:

```c
// MSG_NOSIGNAL tells the kernel: don't send SIGPIPE on broken pipe

ssize_t n = send(fd, buf + total_written, len - total_written, MSG_NOSIGNAL);
```

`MSG_NOSIGNAL` is Linux-specific. `signal(SIGPIPE, SIG_IGN)` is portable and works on Linux, macOS, and BSDs. For this project, use `signal(SIGPIPE, SIG_IGN)` at startup.

The Complete `main()` Function

```c
int main(int argc, char *argv[]) {

    int port = DEFAULT_PORT;

    if (argc == 2) {

        port = atoi(argv[1]);

        if (port <= 0 || port > 65535) {

            fprintf(stderr, "Invalid port: %s\n", argv[1]);

            return 1;

        }

    }

    // --- SIGPIPE: must happen before any socket I/O ---

    signal(SIGPIPE, SIG_IGN);

    // --- Step 1: Create listening socket ---

    int server_fd = socket(AF_INET, SOCK_STREAM, 0);

    if (server_fd < 0) {

        perror("socket");

        return 1;

    }

    // --- Step 2: Enable port reuse ---

    int opt = 1;

    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {

        perror("setsockopt");

        close(server_fd);

        return 1;

    }

    // --- Step 3: Bind to port ---

    struct sockaddr_in addr;
```

```
    memset(&addr, 0, sizeof(addr));
    addr.sin_family      = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port        = htons((uint16_t)port);
    if (bind(server_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("bind");
        close(server_fd);
        return 1;
    }
    // --- Step 4: Start listening ---
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen");
        close(server_fd);
        return 1;
    }
    printf("Server listening on port %d\n", port);
    // --- Step 5: Accept loop ---
    while (1) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        int client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &client_len);
        if (client_fd < 0) {
            if (errno == EINTR) continue;  // Interrupted by signal, retry
            perror("accept");
            continue;  // Log and continue; don't crash the server
        }
        // Log the incoming connection
```

```
        char client_ip[INET_ADDRSTRLEN];

        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, sizeof(client_ip));

        printf("Connection from %s:%d (fd=%d)\n",

               client_ip, ntohs(client_addr.sin_port), client_fd);

        // --- Step 6a: Read the request ---

        char req_buf[REQUEST_BUF_SIZE];

        int header_end = -1;

        ssize_t bytes_read = read_http_request(client_fd, req_buf, REQUEST_BUF_SIZE,
&header_end);

        if (bytes_read > 0 && header_end > 0) {

            // Log the first line of the request

            char *first_line_end = strstr(req_buf, "\r\n");

            if (first_line_end) *first_line_end = '\0';

            printf("Request: %s\n", req_buf);

            if (first_line_end) *first_line_end = '\r';  // Restore

        }

        // --- Step 6b: Build and send the response ---

        char resp_buf[MAX_RESPONSE_SIZE];

        int resp_len = build_hardcoded_response(resp_buf, sizeof(resp_buf));

        if (resp_len > 0) {

            if (write_all(client_fd, resp_buf, (size_t)resp_len) < 0) {

                // Client disconnected before we finished — EPIPE, not a crash

                printf("Client disconnected during write (fd=%d)\n", client_fd);

            }

        }

        // --- Step 7: Close the connection ---

        close(client_fd);
```

```
        printf("Connection closed (fd=%d)\n", client_fd);
    }
    // Unreachable in this milestone, but good practice:
    close(server_fd);
    return 0;
}
```

Compiling and Testing

```bash
# Compile
gcc -Wall -Wextra -o http_server server.c

# Run on default port 8080
./http_server

# Run on a different port
./http_server 9090

# Test with curl (in another terminal)
curl -v http://localhost:8080/

# Test with telnet to see raw bytes
telnet localhost 8080
# Type: GET / HTTP/1.1
# Type: Host: localhost
# Press Enter twice
```

Expected `curl -v` output:

```
* Connected to localhost port 8080
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.x.x
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 53
< Date: Sat, 28 Feb 2026 12:00:00 GMT
< Connection: close
<
<html><body><h1>Hello from my HTTP server!</h1></body></html>
```

Verifying No File Descriptor Leaks

A critical correctness check: after many connections, your server must return to its baseline FD count. You can verify this on Linux:

```bash
# Get your server's PID

ps aux | grep http_server

# Check open FDs before connections

ls /proc/<PID>/fd | wc -l

# Hammer it with 1000 sequential connections

for i in $(seq 1000); do curl -s http://localhost:8080/ > /dev/null; done

# Check open FDs after — should be the same count as before

ls /proc/<PID>/fd | wc -l
```

If the count grows, you have a file descriptor leak. In this milestone's sequential design, the cause is always a missing `close(client_fd)` on an

error path. Every `if (something_failed)` block that exits before the normal `close(client_fd)` must also close the FD.

Hardware Soul: What the CPU and Kernel Are Doing

Let's apply the domain's hardware perspective to the operations in this milestone. `accept()` — blocking on empty queue: Your thread is in the `TASK_INTERRUPTIBLE` sleep state. The CPU runs other processes. When a SYN arrives, the NIC raises a hardware interrupt, the kernel interrupt handler runs (in kernel space), processes the TCP three-way handshake, moves the connection to the accept queue, and transitions your thread to `TASK_RUNNING`. Next time the scheduler runs, your thread wakes up and `accept()` returns. Zero CPU cycles wasted while waiting. `read()` from socket — memory access pattern: The kernel's receive buffer is in kernel virtual memory. When you call `read()`, the kernel copies bytes from its buffer into your userspace buffer (`req_buf`). This is a memory-to-memory copy with good locality if the request is small (fits in L1/L2 cache). For a typical 200-byte HTTP request, the entire operation — kernel buffer + your buffer — fits in L1 cache (typically 32–64KB). Latency: ~100ns for in-cache operations. `write()` to socket — send buffer: The kernel has a send buffer per socket (default ~87KB on Linux, controlled by `SO_SNDBUF`). Your `write()` copies data from your buffer into the send buffer and returns. The kernel's TCP stack asynchronously handles segmentation and transmission. This means `write()` is nearly always fast (just a memory copy). It only blocks if the send buffer is full — which happens when the network is congested or the client is reading slowly. `close()` — FD table update: Closing an FD is a kernel table update: decrement the reference count on the socket

structure, release the FD slot in the process's file descriptor table. If reference count hits zero (no other FD points to this socket), begin TCP FIN sequence. Very fast — no disk I/O, just memory writes. Branch prediction: The accept loop's `while(1)` is perfectly predictable (always taken). The error checks (`if (client_fd < 0)`) are almost never taken in normal operation — the branch predictor learns this quickly. The partial read loop's termination condition (`strstr` found `\r\n\r\n`) is taken once per request — predictable after the first connection.

Error Handling Philosophy for Systems Code

Notice that every syscall's return value is checked. This is not optional ceremony — it's load-bearing structure. Syscalls fail for real reasons:

Error	Cause	Correct Response
`EADDRINUSE` on `bind()`	Port already in use	Exit with clear error message
`EMFILE` on `accept()`	Too many open FDs	Log error, continue loop, investigate leak
`EINTR` on `accept()`	Signal interrupted the call	Retry immediately (loop `continue`)
`EPIPE` on `write()`	Client disconnected	Close `client_fd`, continue to next connection
`ECONNRESET` on `read()`	Client forcibly closed	Close `client_fd`, continue
The pattern: **server-fatal errors** (can't bind, can't listen) cause `exit()`. **Per-connection errors** (client misbehaved, disconnected) cause `close(client_fd)` and `continue`. Never let a misbehaving client crash the server — which is why `SIGPIPE` must be ignored globally.		
`perror("context string")` prints your context string followed by the system error string for `errno`. It's the fastest way to get diagnostic information. `strerror(errno)` gives you the string if you need to format it differently.		

Design Decision: Sequential vs. Concurrent Connections

In this milestone, your server handles one connection at a time. While serving client A, client B's `connect()` call succeeds (the kernel accepts it into the backlog queue), but B must wait until you finish A before you call `accept()` again.

Model	Throughput	Complexity	Used By
Sequential (this milestone) ✓	Low	Very low	dev tools, test servers
Thread-per-connection	Medium	Medium	Apache httpd (legacy)
Thread pool	High	Medium	nginx worker model
Event loop (epoll)	Very high	High	nginx, Node.js, Redis
You're building sequential *deliberately* to isolate the core socket machinery. Milestone 4 replaces this with a thread pool. Once you understand the sequential model perfectly, the concurrent model is just "run the inner loop in a separate thread" — and all the complexity is in the threading, not the I/O.			
The `SOMAXCONN` backlog helps here: even with sequential handling, the kernel queues up to `SOMAXCONN` connections, so brief bursts don't immediately reject connections. For a development and learning server, sequential is exactly right.			

Knowledge Cascade: What This Unlocks

Understanding TCP's byte-stream nature and the socket lifecycle opens doors in five directions simultaneously.

1. Every Language's Buffered I/O Makes Sense Now

Every language runtime wraps raw I/O in a buffered reader for exactly the reason you just discovered. Go's `bufio.Reader`, Python's `io.BufferedReader`, Java's `BufferedInputStream`, and C's `FILE*` (via `fread`) all maintain an internal accumulation buffer and expose "give me a line" or "give me N bytes" interfaces that hide the partial-read reality underneath. Now when you see `bufio.NewReader(conn)` in Go networking code, you know *exactly* what problem it solves and what the raw socket calls beneath it are doing.

2. Protocol Framing Is a Universal Problem

Every protocol that runs over TCP must solve the same framing problem: "where does one message end and the next begin?" The approaches are consistent across the industry:

- **HTTP/1.1**: headers end with `\r\n\r\n`, body length from `Content-Length` or chunked encoding
- **Redis RESP protocol**: length-prefixed (`*3\r\n$3\r\nSET\r\n...`)
- **PostgreSQL wire protocol**: 5-byte header with message type + 4-byte length
- **WebSockets**: 2–10 byte binary frame header with payload length
- **Protocol Buffers over TCP**: typically length-prefixed with a 4-byte varint Once you understand why HTTP uses `\r\n\r\n`, you can read any protocol specification and immediately identify its framing strategy.

3. Nagle's Algorithm and TCP_NODELAY

Now that you understand TCP as a byte stream, you can understand **Nagle's algorithm**: a TCP optimization that coalesces small writes before transmitting. If your application sends many small write() calls (say, HTTP headers one line at a time), Nagle's algorithm batches them into fewer packets, improving network efficiency. The tradeoff: it adds latency for small-packet protocols. This is why latency-sensitive applications (Redis, game servers, financial trading systems) call `setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &one, sizeof(one))` to disable Nagle's algorithm. Now that phrase means something concrete to you.

4. Non-Blocking I/O and Event Loops

The partial read loop you implemented works because this is *blocking* I/O: `read()` blocks until data arrives. The alternative — **non-blocking I/O** with `O_NONBLOCK` — makes `read()` return immediately with `EAGAIN` if no data is available. Systems like epoll (Linux) and kqueue (macOS) let you monitor thousands of sockets simultaneously and get notified when any of them have data. The partial-read problem doesn't disappear in non-blocking mode — it becomes *more* explicit. nginx, Node.js, Redis, and Go's net package all use this model. You'll understand io_uring (the next generation) when you reach it because you already understand why blocking `read()` is a bottleneck at scale.

5. The FD Lifecycle Is Universal

The file descriptor management discipline you're building — open, use, close, never leak — applies identically to files, pipes, sockets, timers (timerfd), event notifications (eventfd), and even GPU command buffers in Vulkan. The kernel's reference-counted resource model is the same everywhere. The `ulimit -n` ceiling is the same ceiling. The `/proc/<pid>/fd/`

directory lets you inspect any process's open resources, and now you know what you're looking at.

Common Mistakes That Will Burn You

Keep these in a mental checklist: 1. Forgetting `htons()` on the port. `bind()` will succeed — on the wrong port. You'll spend an hour wondering why `curl localhost:8080` times out before realizing your server is actually listening on port 36895. 2. Not handling `EINTR` on `accept()`. When a signal arrives (including SIGCHLD in Milestone 4's thread model), `accept()` returns -1 with `errno = EINTR`. If you exit the accept loop on any negative return, one signal kills your server. Always check `errno == EINTR` and retry. 3. Assuming `read()` returns a complete request. Works on localhost, fails in staging, fails in production, fails in any test that simulates network conditions. The partial-read loop is non-negotiable. 4. Missing `close(client_fd)` on error paths. Every early return inside the connection-handling block must close `client_fd` before returning. Draw the control flow on paper if needed. A leak here is a server that degrades after thousands of connections and becomes unfixable without a restart. 5. Ignoring `write()` return values. `write()` returning -1 with `EPIPE` is the normal case when a client disconnects. If you don't check return values, you don't know whether your response was delivered, and you may try to write to a dead connection repeatedly.

Acceptance Criteria Checklist

Before moving to Milestone 2, verify each of these manually:

- [ ] `./http_server 8080` starts and prints "Server listening on port 8080"

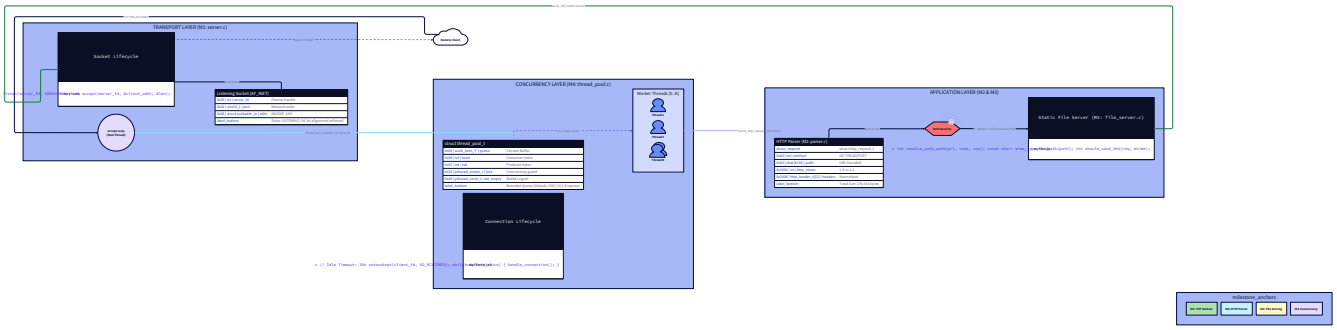
- [ ] `curl http://localhost:8080/` receives a valid `200 OK` response with all four required headers (`Content-Type`, `Content-Length`, `Date`, `Connection`)
- [ ] The HTML body is correct and `Content-Length` matches the actual body byte count
- [ ] `curl http://localhost:8081/` fails with "Connection refused" (confirms port is configurable)
- [ ] Server does not crash when you kill `curl` mid-request (SIGPIPE handled)
- [ ] After 1000 sequential `curl` requests, FD count in `/proc/<pid>/fd/` matches the count at startup
- [ ] Telnet test works: `telnet localhost 8080`, type `GET / HTTP/1.1\r\nHost: localhost\r\n\r\n`, receive full response
- [ ] Server logs the client IP and request line for each connection

---

Milestone 2: HTTP Request Parsing

Where You Are in the System

In Milestone 1, you built the socket plumbing: the server binds a port, accepts a TCP connection, reads raw bytes into a buffer, and sends a hardcoded response. You proved the plumbing works. Now those raw bytes need to mean something.



Right now, your server is holding a buffer that looks like this:

```
GET /index.html HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.88\r\nAccept: */*\r\n\r\n
```

That's just a `char[]`. Your server doesn't know the method, doesn't know the path, doesn't know whether to read a body. Before you can serve files (Milestone 3) or handle multiple connections properly (Milestone 4), you need a parser — code that transforms those raw bytes into

structured data your program can reason about. This milestone's job: turn the byte buffer into a struct. After this milestone, the rest of your server speaks in terms of `request.method`, `request.path`, and `request.headers["content-type"]` — not raw offsets and `strstr()` calls scattered across the codebase.

The Revelation: HTTP Parsing Is Adversarial

Here's what most developers think HTTP parsing is:

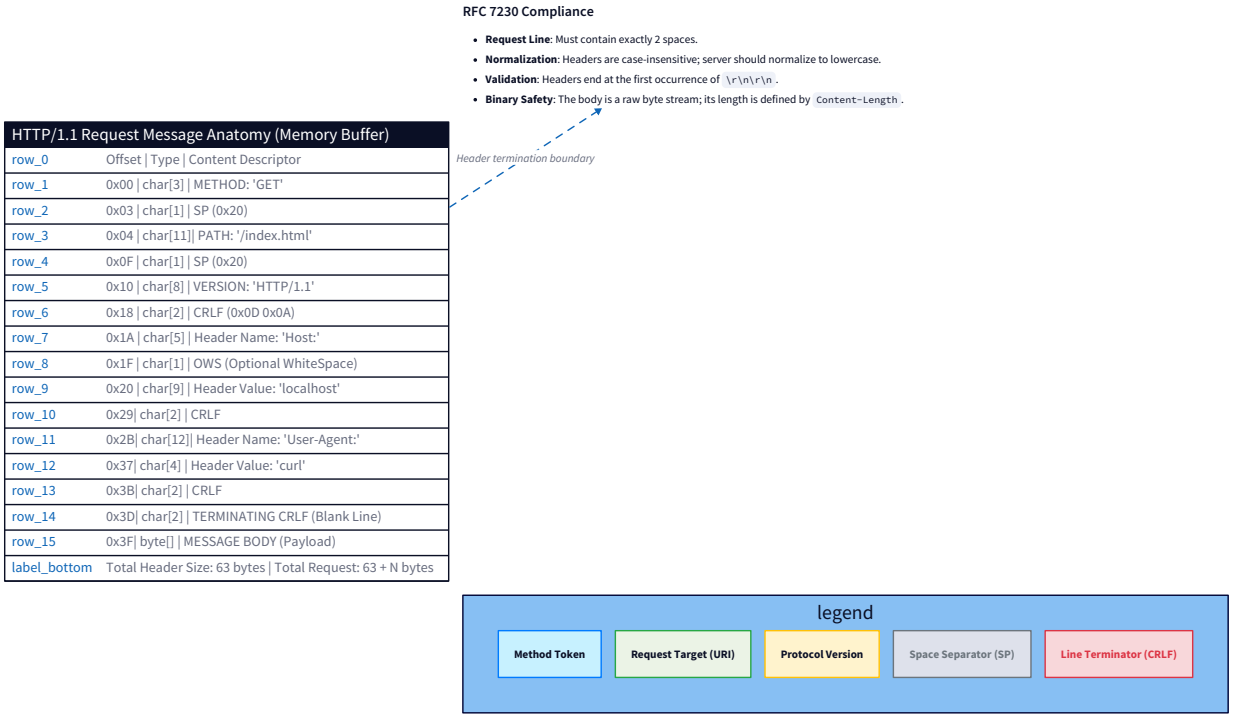
```c
// Seems obvious, right?
char *method = strtok(request_line, " ");
char *path   = strtok(NULL, " ");
char *version = strtok(NULL, "\r\n");
```

Three calls to `strtok()`, done. The method, path, and version are extracted. Ship it. This code works perfectly — in your tests, on your machine, with your browser. And then, in production, it silently fails or, worse, opens a security hole. Let's understand exactly why.

The Input You Won't See in Testing

Real HTTP clients are not all Chrome on macOS sending perfectly formatted requests. They include:

- **`telnet` and `netcat`** — send bare LF (`\n`) instead of CRLF (`\r\n`) because a human typed the request manually
- **Old HTTP/1.0 clients** — may send `GET /\n\n` with no headers at all
- **Malicious scanners** — deliberately send malformed requests to probe for vulnerabilities: `GET ` (no path), `GET /` (no version), requests with 100MB URIs
- **Proxies** — add extra headers, fold multi-line header values across lines (an HTTP/1.1 feature called "header folding," now deprecated but still encountered)
- **HTTP fuzzers** — systematically mutate every field looking for crashes Your `strtok()` parser treats all of these as undefined behavior. `strtok()` with `" "` as the delimiter will happily return whatever's between spaces — including a URI that's a gigabyte long — and your server will `malloc()` or buffer-overflow trying to hold it.



Why "Split on Spaces" Is a Security Vulnerability

Consider this request:

```
GET /../../../etc/passwd HTTP/1.1\r\nHost: localhost\r\n\r\n
```

Your `strtok()` parser extracts `/../../etc/passwd` as the path. If Milestone 3's file server doesn't also validate this, you've just served `/etc/passwd`. But even before that — your parser should apply length limits, method validation, and version checking. Every check your parser skips is a check that Milestone 3 has to retroactively add, and you will miss some. Now consider:

```
GET /valid-path HTTP/1.1\r\ncontent-type: application/json\r\nContent-Type:
text/html\r\n\r\n
```

The same header name appears twice, in different cases. Which one is the "real" `Content-Type`? A naive parser that uses case-sensitive string comparison won't even recognize `content-type` as the same header. RFC 7230 says header names are case-insensitive. Your parser must implement that. The real mental model: HTTP parsing is adversarial

string processing. Every field in an HTTP request is input from a potentially hostile source. Your parser's job is not just extraction — it's validation, normalization, and rejection.

The HTTP/1.1 Message Format

Before writing a parser, you need to know what you're parsing. Let's establish the exact byte-level structure of an HTTP/1.1 request, per RFC 7230. An HTTP/1.1 request has this structure:

```
{method} SP {request-target} SP {HTTP-version} CRLF
{header-name} ":" OWS {header-value} OWS CRLF
{header-name} ":" OWS {header-value} OWS CRLF
...
CRLF
[ message-body ]
```

Where:

- **SP** = a single space character (0x20). The spec says exactly one space between these three tokens.
- **CRLF** = `\r\n` (0x0D 0x0A). Each line, including the blank line that terminates headers, ends with CRLF.
- **OWS** = Optional WhiteSpace — zero or more spaces or tabs that may appear before or after a header value. Let's look at a complete example at the byte level:

```
G E T   /   H T T P / 1 . 1  \r \n
H o s t :   l o c a l h o s t  \r \n
U s e r - A g e n t :   c u r l  \r \n
\r \n
```

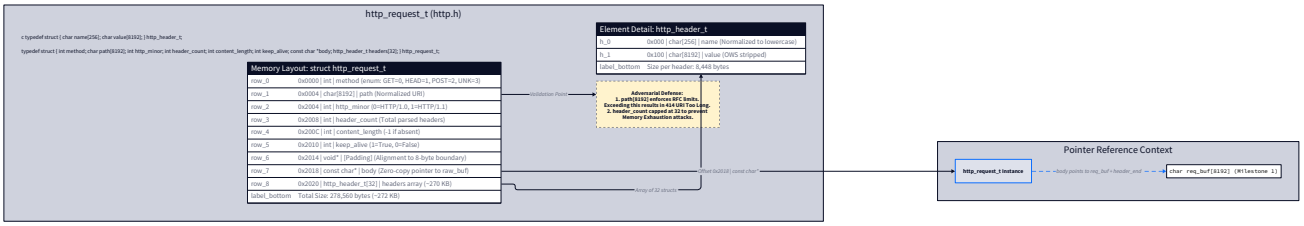
In hex, the first two bytes of the body separator are `0D 0A 0D 0A` — two CRLF sequences back-to-back. The first ends the last header line; the second is the blank line that terminates the header section. Three structural rules are critical to parse correctly:

1. **The request line is the first line.** Everything before the first CRLF is the request line.
2. **Each header is `name: value\r\n`.** The colon is the separator. Everything after the colon (minus OWS) is the value.
3. **A blank line (`\r\n` on its own) signals end-of-headers.** In raw bytes, that's `\r\n\r\n` following the last header. There is one important leniency the RFC recommends: **accept bare LF as a line terminator.** The spec says servers "SHOULD" accept bare LF in place of CRLF for historical reasons. When `telnet` users type a request manually, they often send bare LF. A robust parser strips any `\r` before checking for `\n`.

Designing the Parsed Request Structure

Before writing a single parsing function, design the data structure that will hold the result. Good structure design makes everything downstream simpler.

```c
#define MAX_HEADERS    32     // Maximum number of HTTP headers we'll accept
#define MAX_PATH_LEN   8192   // 8KB max URI length (matches nginx default)
#define MAX_HEADER_LEN 8192   // 8KB max per-header value length
#define METHOD_GET     0
#define METHOD_HEAD    1
#define METHOD_POST    2
#define METHOD_UNKNOWN 3
typedef struct {
    char  name[256];              // Header name (lowercase, normalized)
    char  value[MAX_HEADER_LEN];  // Header value (OWS stripped)
} http_header_t;
typedef struct {
    int          method;                  // METHOD_GET, METHOD_HEAD, METHOD_POST,
METHOD_UNKNOWN
    char         path[MAX_PATH_LEN];      // Requested path, e.g. "/index.html"
    int          http_minor;              // HTTP version minor (0 = HTTP/1.0, 1 =
HTTP/1.1)
    http_header_t headers[MAX_HEADERS];   // Parsed headers array
    int          header_count;            // Number of valid headers in array
    int          content_length;          // -1 if no Content-Length header
    int          keep_alive;              // 1 if connection should persist, 0 if close
    // Pointer into the original buffer — body starts here, length = content_length
    const char  *body;                    // NULL if no body
} http_request_t;
```

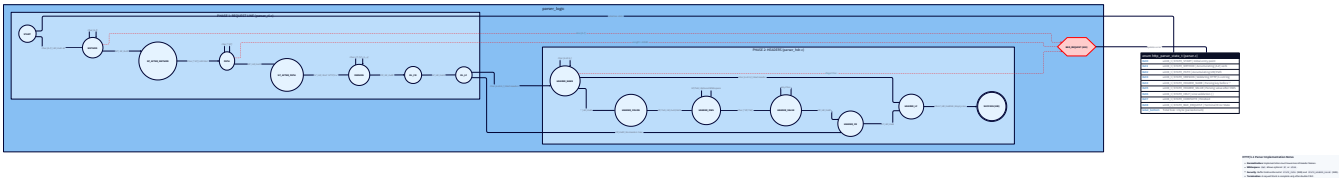


Several design decisions worth examining: Fixed-size arrays instead of dynamic allocation. `http_header_t headers[MAX_HEADERS]` uses 32 statically-allocated header slots. An alternative is `malloc()`ing a linked list or growing array. The fixed-array approach has two advantages: no allocation failures, and cache locality (all headers are contiguous in memory). The downside: we reject requests with more than 32 headers. In practice, real HTTP requests have 5–15 headers, so 32 is generous. nginx's default is 100, but HTTP/2 HPACK compression means that limit is rarely approached. Lowercase header names. We'll normalize all header names to lowercase during parsing. This is the "right" approach for case-insensitive comparison — normalize once at parse time, compare cheaply everywhere else. The alternative (compare case-insensitively every time you look up a header) means writing a `strcasecmp()` comparison for every header lookup, scattered across your codebase. `content_length` as -1 sentinel. `-1` means "no Content-Length header was present." `0` means the header was present and specified a zero-length body. This distinction matters: a response with `Content-Length: 0` and no body is valid and different from a response with no `Content-Length` at all. `body` as pointer into the original buffer. The body isn't copied — we point into the buffer we already read. This is the beginning of zero-copy parsing: instead of `malloc()`ing memory and copying the body, we just record where in the existing buffer the body

starts. For large request bodies (file uploads, POST data), this avoids a potentially multi-megabyte copy.

Building the State Machine

Now the core of this milestone: the parser itself. The right mental model for parsing HTTP is a **state machine** — a system that reads one byte (or token) at a time and transitions between well-defined states based on what it sees.



Why a state machine and not just `strtok()` and `sscanf()`? Three reasons:

1. **Partial input handling.** A state machine can be paused mid-parse (when a partial read arrives) and resumed when more bytes arrive. `strtok()` can't — it requires the complete string upfront.
2. **Error localization.** Each state enforces its own invariants. When you receive an invalid character, you know exactly which state you're in, which tells you exactly what's wrong with the request.
3. **Security through structure.** A state machine naturally imposes length limits (check at every transition) and rejects malformed input (illegal characters in the wrong state trigger immediate error). Our parser will work at the **line level** rather than the byte level, since HTTP is line-structured. We'll split the buffer into lines, parse the first line as the request line, and parse remaining lines (until the blank line) as headers.

Step 1: Parse the Request Line

The request line has exactly three fields separated by single spaces:

```
METHOD SP request-target SP HTTP-version CRLF
```

```c
// Parse an HTTP method string into our enum.
// Returns METHOD_GET, METHOD_HEAD, METHOD_POST, or METHOD_UNKNOWN.
static int parse_method(const char *s, size_t len) {
    if (len == 3 && memcmp(s, "GET",  3) == 0) return METHOD_GET;
    if (len == 4 && memcmp(s, "HEAD", 4) == 0) return METHOD_HEAD;
    if (len == 4 && memcmp(s, "POST", 4) == 0) return METHOD_POST;
    return METHOD_UNKNOWN;
}
// Parse "HTTP/1.X" version string.
// Returns HTTP minor version (0 or 1), or -1 on invalid format.
static int parse_http_version(const char *s, size_t len) {
    // Must be exactly "HTTP/1.X" — 8 characters
    if (len != 8) return -1;
    if (memcmp(s, "HTTP/1.", 7) != 0) return -1;
    if (s[7] == '0') return 0;
    if (s[7] == '1') return 1;
    return -1;
}
// Parse the HTTP request line into the request struct.
// `line` is a null-terminated string with the trailing CRLF already stripped.
// Returns 0 on success, an HTTP status code on failure (400, 414, 501).
int parse_request_line(const char *line, http_request_t *req) {
    // Find first space (separates method from path)
    const char *path_start = strchr(line, ' ');
    if (path_start == NULL) return 400;  // No space found: malformed
    size_t method_len = (size_t)(path_start - line);
    path_start++;  // Skip the space
```

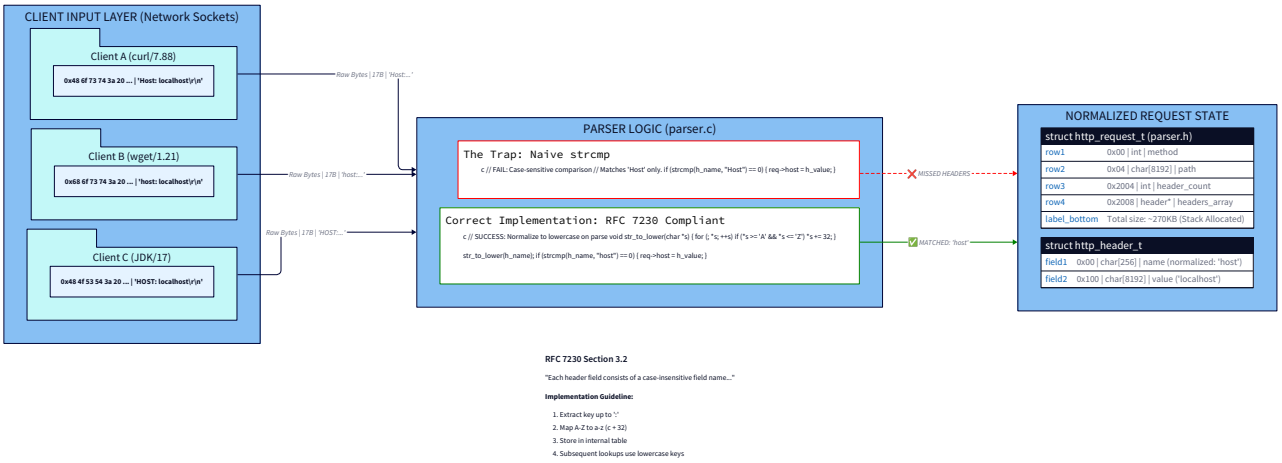
```
    // Find second space (separates path from version)
    const char *version_start = strchr(path_start, ' ');
    if (version_start == NULL) return 400;  // No version: malformed
    size_t path_len = (size_t)(version_start - path_start);
    version_start++;  // Skip the space
    // Check for extra spaces (method SP path SP version — exactly one space each)
    if (memchr(line, ' ', method_len) != NULL) return 400;
    // Validate and store method
    req->method = parse_method(line, method_len);
    if (req->method == METHOD_UNKNOWN) return 501;  // 501 Not Implemented
    // Validate path length before copying
    if (path_len == 0)              return 400;  // Empty path
    if (path_len >= MAX_PATH_LEN)   return 414;  // URI Too Long
    // Copy path and null-terminate
    memcpy(req->path, path_start, path_len);
    req->path[path_len] = '\0';
    // Validate HTTP version
    size_t version_len = strlen(version_start);
    req->http_minor = parse_http_version(version_start, version_len);
    if (req->http_minor < 0) return 400;  // Unknown version
    return 0;  // Success
}
```

Walk through the design choices: **`memcmp()` for method comparison, not `strcmp()`.** `strcmp()` works on null-terminated strings, which means it keeps reading until it hits `'\0'`. `memcmp()` with an explicit length is both safer (bounded) and faster (the compiler can often reduce it to a 3-byte or 4-byte integer comparison). **`strchr()` to find delimiters.** We use `strchr()` to find the first space (method/path boundary) and second space (path/version boundary). This is cleaner than `strtok()`, which has the critical flaw of modifying its input — `strtok()` replaces each delimiter with `'\0'`, which would corrupt our original buffer.

Method not in our list → 501, not 400. RFC 7231 specifies: "The server SHOULD respond with a 501 (Not Implemented) status code" for methods it doesn't implement, and "400 (Bad Request)" for malformed requests. The distinction matters for HTTP compliance. A `PATCH` request is a valid method — just not one we implement. A ` ` (spaces only) "method" is not valid at all. **414 for oversized URI.** The 414 status code means "URI Too Long." RFC 7230 recommends that servers respond with 414 when the request-target is longer than any URI the server is willing to process. Our limit of 8KB (8192 bytes) matches nginx's default and is a reasonable bound.

Step 2: Parse Headers

Headers are more complex than the request line because there are many of them, they have optional whitespace, their names are case-insensitive, and we need to extract specific ones (Host, Content-Length, Connection) while storing the rest for later use.



```c
// Convert a string to lowercase in-place.
// This is how we normalize header names for case-insensitive comparison.
static void str_to_lower(char *s) {
    for (; *s; ++s) {
        if (*s >= 'A' && *s <= 'Z') {
            *s = (char)(*s + 32);
        }
    }
}
// Strip leading and trailing OWS (Optional WhiteSpace: space and tab)
// from `s`, writing null-terminated result into `out` (max `out_size` bytes).
// Returns 0 on success, -1 if result doesn't fit.
static int strip_ows(const char *s, char *out, size_t out_size) {
    // Skip leading whitespace
    while (*s == ' ' || *s == '\t') s++;
    // Find end of string
    size_t len = strlen(s);
    // Strip trailing whitespace
    while (len > 0 && (s[len-1] == ' ' || s[len-1] == '\t')) len--;
    // Check it fits
    if (len >= out_size) return -1;
    memcpy(out, s, len);
    out[len] = '\0';
    return 0;
}
// Parse a single header line "Name: Value" (CRLF already stripped).
// Stores the result in req->headers[req->header_count] and increments header_count.
```

```c
// Returns 0 on success, -1 on malformed header (skip it), -2 on too many headers.
int parse_header_line(const char *line, http_request_t *req) {
    if (req->header_count >= MAX_HEADERS) return -2;
    // Find the colon separating name from value
    const char *colon = strchr(line, ':');
    if (colon == NULL) return -1;  // No colon: malformed, skip
    size_t name_len = (size_t)(colon - line);
    if (name_len == 0) return -1;  // Empty header name: malformed, skip
    if (name_len >= sizeof(req->headers[0].name)) return -1;  // Name too long
    http_header_t *h = &req->headers[req->header_count];
    // Copy header name and lowercase it for case-insensitive storage
    memcpy(h->name, line, name_len);
    h->name[name_len] = '\0';
    str_to_lower(h->name);
    // Strip OWS from value (everything after the colon)
    const char *value_raw = colon + 1;
    if (strip_ows(value_raw, h->value, sizeof(h->value)) < 0) {
        return -1;  // Value too long
    }
    req->header_count++;
    return 0;
}
// Look up a header by name (already lowercase).
// Returns pointer to header value string, or NULL if not found.
const char *request_get_header(const http_request_t *req, const char *name) {
    for (int i = 0; i < req->header_count; i++) {
        if (strcmp(req->headers[i].name, name) == 0) {
```

```
            return req->headers[i].value;
        }
    }
    return NULL;
}
```

The `str_to_lower()` function deserves attention. We're converting only ASCII letters (`A` – `Z` → `a` – `z`). The conversion is `c + 32` because in ASCII, uppercase letters are exactly 32 positions before their lowercase equivalents (e.g., `'A'` is 65, `'a'` is 97). This is faster than `tolower()` from `<ctype.h>` because `tolower()` checks locale settings at runtime. HTTP header names are defined as ASCII-only in RFC 7230, so locale-independence isn't a concern here. The `request_get_header()` function performs a linear search through the headers array. With at most 32 headers, this is O(32) — effectively O(1). We're calling this function a handful of times per request for specific headers (Host, Content-Length, Connection). A hash table would be faster asymptotically but adds significant complexity for negligible real-world benefit at this scale.

Step 3: Post-Processing — Extract Semantic Headers

After parsing all header lines, we need to extract the ones that control connection behavior and body reading:

C

```c
// After parsing all headers, extract semantic fields into the request struct.
// Call this once after the header loop completes.
void extract_semantic_headers(http_request_t *req) {
    // Content-Length: controls body reading
    const char *cl = request_get_header(req, "content-length");
    if (cl != NULL) {
        char *end;
        long val = strtol(cl, &end, 10);
        // Valid if: all characters consumed, non-negative, not astronomically large
        if (*end == '\0' && val >= 0 && val <= (64 * 1024 * 1024)) {
            req->content_length = (int)val;
        } else {
            req->content_length = -1;  // Invalid Content-Length: treat as absent
        }
    } else {
        req->content_length = -1;
    }
    // Connection: keep-alive semantics
    // HTTP/1.1 default is keep-alive; HTTP/1.0 default is close
    const char *conn = request_get_header(req, "connection");
    if (req->http_minor == 1) {
        // HTTP/1.1: default keep-alive, unless "Connection: close"
        req->keep_alive = (conn == NULL || strcasecmp(conn, "close") != 0) ? 1 : 0;
    } else {
        // HTTP/1.0: default close, unless "Connection: keep-alive"
        req->keep_alive = (conn != NULL && strcasecmp(conn, "keep-alive") == 0) ? 1 : 0;
    }
```

```
}
```

`strtol()` for Content-Length, not `atoi()`. `atoi()` has no error detection — if the input is `"abc"` or `""`, it returns 0 without any indication of failure. `strtol()` sets an end pointer: if `*end != '\0'` after the call, there were non-numeric characters. We also cap at 64MB — a server serving static HTML and CSS files has no business reading a 4GB request body. This limit prevents a slow client from sending `Content-Length: 9999999999` and holding your connection open while trickling bytes. `strcasecmp()` for `Connection` values. Header *names* we normalize to lowercase ourselves. Header *values* may be any case — the spec says method tokens and header values are case-insensitive where defined. `Connection: Close`, `Connection: CLOSE`, and `Connection: close` are all valid. `strcasecmp()` (POSIX) handles this. On Windows you'd use `_stricmp()` — worth knowing if you ever port this code.

Assembling the Full Parser

Now put the pieces together into a single function that takes the raw buffer from Milestone 1 and returns a populated `http_request_t`:

```c
// Parse a complete HTTP request from `buf` (null-terminated, total `buf_len` bytes).
// `header_end` is the offset just past the \r\n\r\n delimiter (from Milestone 1's reader).
// Fills `req` on success.
// Returns 0 on success, HTTP status code on failure (400, 414, 501).
int parse_http_request(const char *buf, size_t buf_len, int header_end,
                       http_request_t *req) {
    memset(req, 0, sizeof(*req));
    req->content_length = -1;
    // --- Line iterator ---
    // We'll walk through the buffer line by line.
    // A line ends with \n (we'll strip any preceding \r).
    const char *cursor = buf;
    const char *buf_end = buf + header_end;  // Don't go past headers
    int line_number = 0;
    char line_buf[8192];  // Scratch buffer for one line
    while (cursor < buf_end) {
        // Find end of this line
        const char *line_end = memchr(cursor, '\n', (size_t)(buf_end - cursor));
        if (line_end == NULL) break;  // No newline found, stop
        size_t line_len = (size_t)(line_end - cursor);
        // Strip trailing \r if present (handle both CRLF and LF)
        if (line_len > 0 && cursor[line_len - 1] == '\r') {
            line_len--;
        }
        // Blank line = end of headers (shouldn't happen since we stop at header_end,
        // but guard anyway)
        if (line_len == 0) {
```

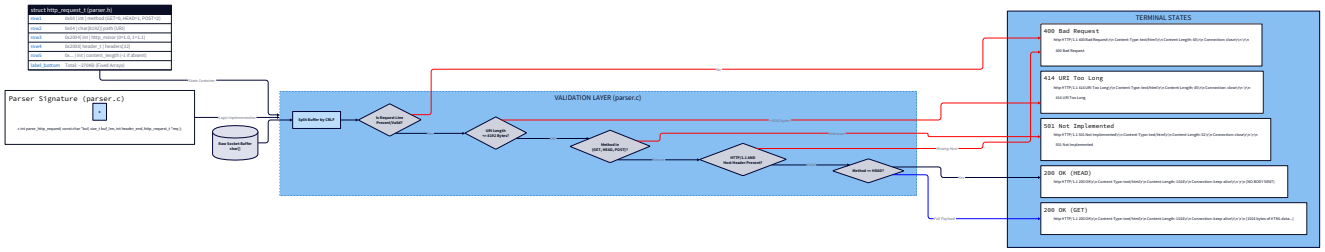
```
            cursor = line_end + 1;
            continue;
        }
        // Copy line into scratch buffer for safe manipulation
        if (line_len >= sizeof(line_buf)) return 400;  // Line too long
        memcpy(line_buf, cursor, line_len);
        line_buf[line_len] = '\0';
        if (line_number == 0) {
            // First line: request line
            int err = parse_request_line(line_buf, req);
            if (err != 0) return err;
        } else {
            // Subsequent lines: header fields
            // Silently skip malformed headers (RFC 7230 permits this)
            parse_header_line(line_buf, req);
        }
        line_number++;
        cursor = line_end + 1;
    }
    if (line_number == 0) return 400;  // No request line at all
    // Validate required HTTP/1.1 headers
    if (req->http_minor == 1) {
        const char *host = request_get_header(req, "host");
        if (host == NULL) return 400;  // Host header required in HTTP/1.1
    }
    // Extract semantic fields from headers
    extract_semantic_headers(req);
```

```
    // Set body pointer if Content-Length present
    if (req->content_length > 0) {
        req->body = buf + header_end;  // Body starts right after headers
        // Note: we don't read the body here — it may not be in the buffer yet.
        // Body reading is handled by the caller with a separate read loop.
    }
    return 0;  // Success
}
```

Several design decisions here: Line copying into `line_buf`. We copy each line into a temporary scratch buffer before parsing it. This protects the original buffer from modification and gives `parse_request_line()` and `parse_header_line()` a null-terminated string to work with. It costs one `memcpy()` per line (~100–500 bytes). For an HTTP server handling thousands of short requests, this is negligible. For a system parsing gigabytes of HTTP traffic (an API gateway or load balancer), you'd investigate zero-copy approaches — but that's the optimization ladder for later. Silently skip malformed headers. RFC 7230 says: "A server that receives an obs-fold in a request message that is not within a `message/http` container MUST either reject the message [...] or replace each received obs-fold with one or more SP octets." The general principle is that a receiver may choose to be lenient with headers it doesn't understand while being strict with the request line and Host header. We silently skip malformed header lines with a negative return from `parse_header_line()` — they don't cause a 400. Host header is required in HTTP/1.1. RFC 7230 Section 5.4: "A client MUST send a Host header field in all HTTP/1.1 request messages." A server "MUST

respond with a 400 (Bad Request) status code to any HTTP/1.1 request message that lacks a Host header field." We enforce this. HTTP/1.0 requests don't require Host, which is why we only check it when `http_minor == 1`.

Error Responses



Sending the right error response is part of the HTTP contract. Clients — especially automated ones — use status codes to decide what to do next. A browser receiving 400 won't retry; a proxy receiving 503 will retry with exponential backoff. Get the codes right.

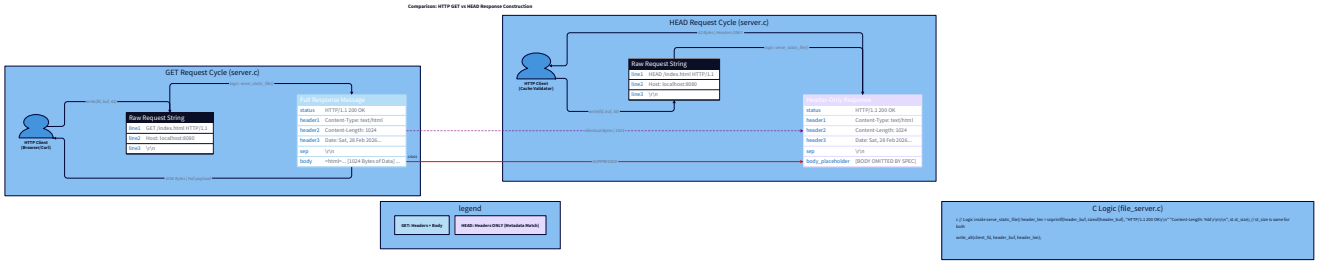
```c
// Send a simple HTTP error response with status code and HTML body.
// `client_fd` is the connection socket.
// `status_code` is e.g. 400, 404, 501.
// `reason` is the reason phrase, e.g. "Bad Request".
// `body` is a short HTML message body.
void send_error_response(int client_fd, int status_code,
                         const char *reason, const char *body) {
    char buf[2048];
    int body_len = (int)strlen(body);
    time_t now = time(NULL);
    struct tm *gmt = gmtime(&now);
    char date_str[64];
    strftime(date_str, sizeof(date_str), "%a, %d %b %Y %H:%M:%S GMT", gmt);
    int n = snprintf(buf, sizeof(buf),
        "HTTP/1.1 %d %s\r\n"
        "Content-Type: text/html; charset=utf-8\r\n"
        "Content-Length: %d\r\n"
        "Date: %s\r\n"
        "Connection: close\r\n"
        "\r\n"
        "%s",
        status_code, reason, body_len, date_str, body);
    if (n > 0 && (size_t)n < sizeof(buf)) {
        write_all(client_fd, buf, (size_t)n);
    }
    // Even if write fails, we close the fd in the caller — no resource leak
}
```

```
// Convenience function: send a 400 Bad Request
void send_400(int client_fd) {
    send_error_response(client_fd, 400, "Bad Request",
        "<html><body><h1>400 Bad Request</h1>"
        "<p>The server could not understand the request.</p>"
        "</body></html>");
}
// Convenience function: send a 501 Not Implemented
void send_501(int client_fd) {
    send_error_response(client_fd, 501, "Not Implemented",
        "<html><body><h1>501 Not Implemented</h1>"
        "<p>The server does not support this HTTP method.</p>"
        "</body></html>");
}
// Convenience function: send a 414 URI Too Long
void send_414(int client_fd) {
    send_error_response(client_fd, 414, "URI Too Long",
        "<html><body><h1>414 URI Too Long</h1>"
        "<p>The requested URI exceeds the server's limit.</p>"
        "</body></html>");
}
```

Map the error codes to their triggers:

Status Code	Trigger	RFC Reference
`400 Bad Request`	Malformed request line, missing Host header in HTTP/1.1, invalid Content-Length	RFC 7230 §3.1.1
`414 URI Too Long`	Request-target exceeds 8192 bytes	RFC 7231 §6.5.12
`501 Not Implemented`	HTTP method is syntactically valid but unrecognized	RFC 7231 §6.6.2

HEAD Requests: Same Headers, No Body



The HEAD method is semantically identical to GET except the server returns all the same headers but omits the response body. This is defined in RFC 7231 Section 4.3.2: "The HEAD method is identical to GET except that the server MUST NOT send a message body in the response." Why does HEAD exist? Two primary use cases:

- **Cache validation**: a client that cached a file can send HEAD to check if the `Last-Modified` or `ETag` changed, without downloading the full file.
- **Resource existence checks**: check if a URL returns 200 or 404 without paying for the transfer bandwidth. The implementation consequence for your server: in Milestone 3, where you actually serve files, the file-serving logic will look like this:

```c
// In your file-serving handler (preview — implemented in Milestone 3):

// Build the response headers the same way for both GET and HEAD.

// Then:

if (req->method != METHOD_HEAD) {

    // Only send the body for GET requests

    send_file_body(client_fd, file_fd, file_size);

}

// For HEAD, we fall through — headers were already sent, no body.
```

Right now, in Milestone 2's parser, all you need to do is correctly identify HEAD and store it as `METHOD_HEAD`. The body-suppression logic lives in the response path, not the parser. The parser's job is faithfully representing what the client asked for; the responder decides what to send back.

Reading the Request Body

For POST requests (and future methods like PUT and PATCH), the body appears after the blank line, and its length is specified by `Content-Length`. Reading the body is a separate step from reading the headers, because:

1. The body may not have arrived by the time you parse the headers
2. The body can be large — potentially megabytes — requiring its own loop
3. For GET and HEAD, there is no body to read Here's the body reading function:

```c
// Read exactly `content_length` bytes of request body from `client_fd`
// into `body_buf` (which must be at least content_length+1 bytes).
// `already_read` is any body bytes that arrived in the header read (rare but possible).
// Returns 0 on success, -1 on error or client disconnect.
int read_request_body(int client_fd, char *body_buf, int content_length,
                      const char *overflow, int overflow_len) {
    if (content_length <= 0) return 0;
    // Some body bytes may have arrived in the initial read, after the \r\n\r\n
    // Copy those first if present
    int already = overflow_len > content_length ? content_length : overflow_len;
    if (already > 0) {
        memcpy(body_buf, overflow, (size_t)already);
    }
    int remaining = content_length - already;
    int total = already;
    while (remaining > 0) {
        ssize_t n = read(client_fd, body_buf + total, (size_t)remaining);
        if (n <= 0) return -1;  // Disconnect or error
        total += (int)n;
        remaining -= (int)n;
    }
    body_buf[content_length] = '\0';
    return 0;
}
```

The `overflow` parameter handles a subtle case: if the client sent a small request body and it arrived in the same `recv()` call as the headers

(common on localhost), your Milestone 1 read loop may have already read part or all of the body into the header buffer. The bytes after the `\r\n\r\n` offset are "overflow" body bytes. We copy them first, then read the remainder from the socket. For this milestone, you'll likely only implement GET and HEAD, so body reading is forward-looking infrastructure. But building the function now means Milestone 3 can simply call it without revisiting the parsing layer.

Integration: Wiring the Parser into the Accept Loop

Here's how the parser integrates with the accept loop from Milestone 1. Replace the minimal handling in `main()`:

```c
// Updated connection handler — replaces the inner block of the accept loop
void handle_connection(int client_fd) {
    char req_buf[REQUEST_BUF_SIZE];
    int header_end = -1;
    // Phase 1: Read raw bytes until \r\n\r\n (from Milestone 1)
    ssize_t bytes_read = read_http_request(client_fd, req_buf,
                                           REQUEST_BUF_SIZE, &header_end);
    if (bytes_read < 0 || header_end < 0) {
        // Client disconnected or request too large
        close(client_fd);
        return;
    }
    // Phase 2: Parse the raw buffer
    http_request_t req;
    int parse_err = parse_http_request(req_buf, (size_t)bytes_read,
                                       header_end, &req);
    if (parse_err != 0) {
        // Parser returned an HTTP error code
        switch (parse_err) {
            case 400: send_400(client_fd); break;
            case 414: send_414(client_fd); break;
            case 501: send_501(client_fd); break;
            default:  send_400(client_fd); break;
        }
        close(client_fd);
        return;
    }
```

```
    // Phase 3: Log the parsed request
    const char *method_names[] = {"GET", "HEAD", "POST", "UNKNOWN"};
    printf("[%s] %s (HTTP/1.%d)\n",
           method_names[req.method], req.path, req.http_minor);
    const char *host = request_get_header(&req, "host");
    if (host) printf("  Host: %s\n", host);
    // Phase 4: Dispatch (Milestone 3 will add real file serving here)
    // For now, send the same hardcoded response as Milestone 1
    char resp_buf[MAX_RESPONSE_SIZE];
    int resp_len = build_hardcoded_response(resp_buf, sizeof(resp_buf));
    if (resp_len > 0) {
        // For HEAD requests: send only headers, no body
        // Temporarily: we send the whole thing and fix it in Milestone 3
        write_all(client_fd, resp_buf, (size_t)resp_len);
    }
    close(client_fd);
}
```

The three-phase structure — read, parse, dispatch — is the architecture you'll carry through Milestones 3 and 4. Each phase has a clear responsibility and returns a clear signal. Phase 1 gives you bytes; phase 2 gives you structure; phase 3 gives you behavior.

Testing the Parser

Manual testing with curl and telnet verifies the happy path. For a parser that handles adversarial input, you need to test the error paths too.

```bash
# Test 1: Valid GET request (should succeed)

curl -v http://localhost:8080/index.html

# Test 2: HEAD request (should receive headers only, no body)

curl -v -X HEAD http://localhost:8080/index.html

# Test 3: Unsupported method → 501

curl -v -X DELETE http://localhost:8080/index.html

# Test 4: Missing HTTP version → 400

# (telnet lets you type raw HTTP manually)

telnet localhost 8080

GET /                    # ← just press Enter (no HTTP version)

Host: localhost

# Test 5: No Host header in HTTP/1.1 → 400

printf "GET / HTTP/1.1\r\n\r\n" | nc localhost 8080

# Test 6: Very long URI → 414

# Generate an 8KB+ URI and send it

python3 -c "print('GET /' + 'a'*8200 + ' HTTP/1.1\r\nHost: localhost\r\n\r\n', end='')" |
nc localhost 8080

# Test 7: Bare LF instead of CRLF (should still work)

printf "GET / HTTP/1.1\nHost: localhost\n\n" | nc localhost 8080

# Test 8: Case-insensitive headers (should work)

curl -v -H "CONTENT-TYPE: text/plain" http://localhost:8080/

# Server should log "content-type: text/plain" (normalized lowercase)
```

For Test 5 (`printf` with no Host header): you'll see the `printf` output piped to `nc` (netcat), which sends the raw bytes to your server. The `\r\n` in the `printf` format string is interpreted by the shell, so you get actual CRLF bytes. This is a reliable way to send hand-crafted HTTP requests without a browser normalizing them. Verify FD leak behavior remains clean after error-path connections:

```bash
# Send 100 malformed requests and check FD count doesn't grow
for i in $(seq 100); do
    printf "INVALID\r\n\r\n" | nc -q1 localhost 8080 2>/dev/null
done
ls /proc/$(pgrep http_server)/fd | wc -l
```

Hardware Soul: What the CPU Does During Parsing

The HTTP parser is the most CPU-intensive component of a simple HTTP server. Let's look at what the hardware is actually doing. Memory access pattern — buffer locality. Your entire HTTP request (typically 200–2000 bytes) fits in L1 cache (32–64KB on modern CPUs). Every call to `strchr()`, `memcmp()`, and `memcpy()` within the parse is operating on L1-resident data. L1 access latency is ~1ns. The entire parse of a typical request takes a few microseconds — fast enough that parsing is never the bottleneck in a well-implemented server. Branch prediction during `str_to_lower()`. The character-by-character lowercase conversion in `str_to_lower()` has a branch: `if (*s >= 'A' && *s <= 'Z')`. In HTTP header names like `Content-Type`, the characters alternate between uppercase (first letter of each word) and lowercase. The branch predictor has a hard time here — the pattern is irregular. Modern CPUs handle this with a trick: the compiler often generates SIMD instructions for short string operations, processing 16 bytes at once with SSE/AVX. The trick: `_mm_sub_epi8` to subtract 'A', compare against 26 (the range of uppercase letters), and conditionally add 32 — all in one instruction on 16 bytes simultaneously. If you enable `-O2` optimization in GCC, the compiler may do this automatically. `strchr()` implementation. The `strchr()` call to

find the colon in a header line like `Content-Type: text/html` scans up to ~15 bytes. The glibc implementation uses SIMD to scan 16 or 32 bytes at a time. For a 15-byte scan, the SIMD overhead isn't worth it, and glibc falls back to a byte loop. But for longer strings (long header values), SIMD `strchr()` is measurably faster. Cache line granularity of the struct. Your `http_request_t` struct is large — with `MAX_HEADERS = 32` entries of `(256 + 8192)` bytes each, the struct approaches 270KB. That won't fit in L1 or L2 cache. However, you only access a handful of headers in practice (Host, Content-Length, Connection), and those are the first few entries in the array. The parser should fill headers in order of appearance, so the most-accessed headers (which tend to appear early in real requests) are in the earliest array slots — resident in the "hot" part of the cache. This is why the earlier design note about fixed arrays matters: a linked list of dynamically-allocated header nodes would scatter them across the heap, causing cache misses when iterating. The contiguous array keeps all headers in a predictable, prefetch-friendly layout.

Three-Level View: Parsing a Header Line

Take `"Content-Type: text/html"` and trace it through all three levels. Level 1 — Your code: `parse_header_line("Content-Type: text/html", &req)` runs. `strchr()` finds the colon at offset 12. `memcpy()` copies "Content-Type" into `h->name`. `str_to_lower()` converts it to "content-type". `strip_ows()` copies "text/html" into `h->value`. `req->header_count` increments. Level 2 — OS/Kernel: No syscalls during parsing. The bytes are already in your process's user-space buffer — the kernel handed them to you when `read()` returned. The OS scheduler may preempt your process during the parse if a

higher-priority process needs to run, but this is transparent. From the kernel's perspective, you're just a user-space process doing arithmetic on memory. Level 3 — Hardware: The CPU is executing integer instructions on cache-resident data. `strchr()` may be vectorized by the compiler/glibc to use SSE4.2's `PCMPEQB` instruction, which compares 16 bytes against the colon character in a single clock cycle. `memcpy()` for 12 bytes compiles to a few register moves — no separate loop. `str_to_lower()` for 12 characters executes 12 iterations of the loop, each testing two conditions and conditionally adding 32. With branch prediction warm (the alternating upper/lower pattern is learnable after a few iterations), mispredictions are rare. Total hardware cost: ~50–100 clock cycles, ~25ns at 2GHz.

Design Decisions: Parser Architecture

Approach	Pros	Cons	Used By
Line-at-a-time (this project) ✓	Simple, correct, debuggable	Two passes (find newlines, then parse)	Many simple HTTP servers
Byte-state-machine	Single pass, handles partial reads elegantly, easily pausable	Complex to write and debug	nginx, h2o, Hyper (Rust)
Regex-based	Very concise, declarative	Slow, security-sensitive regex edge cases	Python's `http.server`
`sscanf()`-based	Simple to write	No error recovery, buffer	Student projects

Approach	Pros	Cons	Used By
		overflow risk	
nginx uses a hand-written byte-level state machine — every byte transitions a `state` enum. This lets nginx parse across partial reads without accumulating a complete buffer first: you can call the parser with 50 bytes, pause, receive 50 more bytes, and resume. For our server, the line-at-a-time approach is correct because we accumulate the complete headers in Milestone 1's read loop before parsing. The state machine approach would be the next optimization if we needed to support very large headers or streaming HTTP/2 frames.			

Knowledge Cascade: What This Unlocks

Understanding HTTP request parsing at this level opens five doors.

1. State Machines for Protocol Parsing — Universal Pattern

The state-machine mental model you're building here is the same one used in every protocol parser ever written. JSON parsers (the JSON spec is a state machine over Unicode characters). SQL lexers (keywords, identifiers, string literals, operators — each a state). Compiler front-ends (tokenization is a state machine over source characters; parsing is a state machine over tokens). TLS record layer parsing. DNS wire format parsing. The moment you internalize "a protocol is a grammar, a parser is a state machine over that grammar, and every input byte drives a transition," you can read any protocol RFC and design a parser for it.

2. Case-Insensitive Comparison — Appearances Across Domains

The case-insensitivity problem you just solved in HTTP header names appears everywhere:

- **SQL**: column names and keywords (`SELECT` = `select` = `SeLeCt`). Database engines normalize identifiers on parse, exactly as you did with `str_to_lower()`.
- **DNS**: domain names are case-insensitive (`Example.COM` = `example.com`). DNS resolvers canonicalize to lowercase before lookup.
- **Windows/macOS filesystems**: paths are case-insensitive but case-preserving (`README.md` and `readme.md` refer to the same file). This is why your Milestone 3 path security code must be careful about case normalization.
- **Email headers** (MIME): `From:`, `To:`, `Subject:` are case-insensitive, just like HTTP. The general principle: **normalize at ingestion, compare at the canonical form**. One conversion, many cheap

comparisons — that's the right tradeoff.

3. Security Through Input Validation — Transfer to Injection Prevention

Every security vulnerability begins with trusting input format. The disciplines you're building here — check length before copying, validate charset before using the value, reject on unexpected characters, apply allow-lists rather than block-lists — transfer directly to:

- **SQL injection prevention**: validating query parameters before interpolation into SQL strings
- **XSS prevention**: validating and escaping HTML-context strings before rendering
- **Path traversal prevention** (which you'll build in Milestone 3): validating paths against a root before passing to `open()`
- **Format string vulnerabilities**: never passing user-controlled input as the format string to `printf()` The mental model: **treat all input as adversarial until proven safe**. Your parser proves safety by checking length, checking charset, checking structure, and rejecting anything that doesn't match the spec. Code that assumes "the client will send valid input" is code that will eventually run in a security incident postmortem.

4. Zero-Copy Parsing — nginx's Secret Weapon

You noticed that `body` in `http_request_t` is a `const char*` pointer into the original buffer, not a copy. This is the beginning of **zero-copy parsing**. nginx takes this much further: it never copies header names or values at all — it stores pairs of `{pointer, length}` (called `ngx_str_t`) that point directly into the receive buffer. Header name "Content-Type" in nginx is represented as a 12-byte pointer into the network buffer, not a separate 12-byte allocation. The performance implication: parsing a 500-byte HTTP request in nginx involves zero `malloc()` calls for the headers themselves. Every `malloc()` call acquires a lock (in glibc's default allocator) and is unpredictable in latency. Eliminating them during parsing removes a major source of latency variance. This is why nginx can handle hundreds of thousands of requests per second on a single core. Once you see why copying is expensive, you'll recognize `string_view` in C++, `&str` vs `String` in Rust, `bytes.Buffer` in Go, and Python's `memoryview` — all are language-level tools for "refer to existing bytes without copying."

5. The RFC As Primary Source — Reading Specifications

You've now implemented something directly against an RFC (RFC 7230). This skill — reading a standards document and translating its MUST/SHOULD/MAY language into code — is one of the most valuable engineering skills that textbooks rarely teach. The RFC tells you exactly when to return 400 vs. 501, exactly what OWS means, exactly which

headers are required. Now that you've done this once, you can read RFC 7231 (HTTP semantics), RFC 7232 (conditional requests, which Milestone 3 needs), RFC 7234 (caching), and RFC 7235 (authentication) and know how to implement them. The same reading skill applies to the POSIX socket spec, the TLS 1.3 spec (RFC 8446), the JSON spec (RFC 8259), and any other protocol you encounter.

Common Mistakes That Will Burn You

1. Using `strtok()` for anything in your parser. `strtok()` modifies its input by replacing delimiters with `'\0'`. It also uses global state, making it unsafe in any context where parsing might be interrupted or called from multiple threads. Use `strchr()`, `memchr()`, and explicit pointer arithmetic instead. 2. Case-sensitive header lookup. The single most common conformance bug: `request_get_header(&req, "Content-Length")` fails to find a header stored as `content-length`. Store lowercase at parse time, look up lowercase always. When in doubt, add a `DEBUG` assert that the name you're looking up is already lowercase. 3. Off-by-one in the CRLF check. When stripping the trailing `\r` before parsing a line: check `line_len > 0` *before* accessing `cursor[line_len - 1]`. Accessing `cursor[-1]` on an empty line is undefined behavior — and on a buffer that started at `buf[0]`, it reads the byte before your buffer. 4. Not validating Content-Length before reading body. If you accept `Content-Length: 9999999999` and try to `malloc()` that many bytes, you crash. If you try to `read()` that many bytes, you hold the connection open indefinitely. Apply a reasonable maximum (64MB for this server) and reject the request with 400 if the value exceeds it. 5. Forgetting bare LF handling. `telnet` sends bare LF. `curl` with `--http1.0` on some platforms sends bare LF. Test 7 in the

section above will fail on your server if you only strip `\r\n`. Your line-splitting code must handle both `\r\n` and `\n`. 6. Not handling the "no request line" case. An empty connection — client connects and immediately disconnects — results in zero bytes read. Your parser receives an empty buffer. `parse_request_line()` on an empty string should return 400, not segfault chasing a `NULL` return from `strchr()`.

Acceptance Criteria Checklist

Verify each of these before moving to Milestone 3:

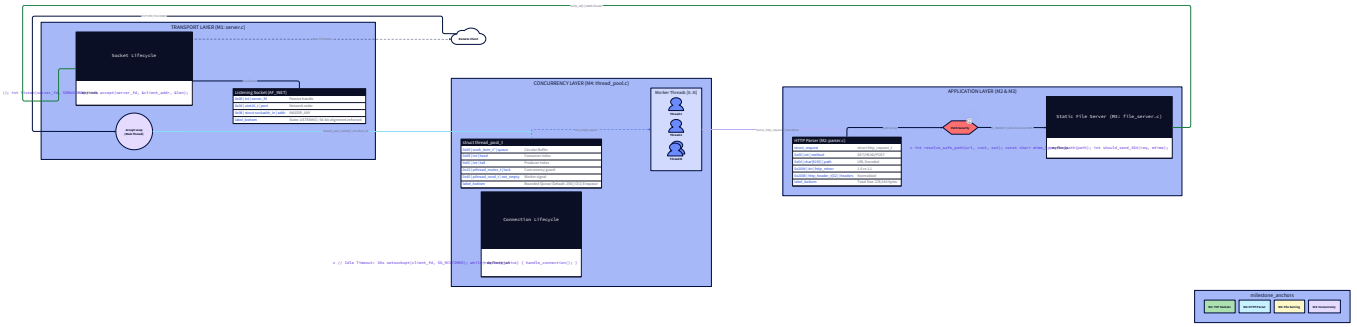
- [ ] `curl http://localhost:8080/` logs `[GET] / (HTTP/1.1)` and `Host: localhost`
- [ ] `curl -X HEAD http://localhost:8080/` is parsed as `METHOD_HEAD` (verify with log output)
- [ ] `curl -X DELETE http://localhost:8080/` returns `HTTP/1.1 501 Not Implemented`
- [ ] Malformed request `printf "NOTHTTP\r\n\r\n" | nc localhost 8080` returns `400 Bad Request`
- [ ] Request with 8201-byte URI returns `414 URI Too Long`
- [ ] HTTP/1.1 request missing `Host` header returns `400 Bad Request`
- [ ] Bare-LF request (no `\r`) is parsed correctly and returns `200 OK`
- [ ] Header `CONTENT-TYPE: text/plain` is stored as `content-type: text/plain` (case-normalized)
- [ ] Header value `Content-Type: text/html ` (extra spaces) has OWS stripped to `text/html`
- [ ] FD count after 100 malformed requests equals FD count before (no leaks on error paths)
- [ ] `request_get_header(&req, "host")` finds the Host header regardless of client's capitalization

---

Milestone 3: Static File Serving

Where You Are in the System

Your server can now accept TCP connections, read raw bytes off the socket, and parse those bytes into structured `http_request_t` data. You know the method, the path, the headers. Now for the first time, your server will do something *real* with that path: go to the filesystem, find a file, and send its contents back across the network.



This is the milestone where your HTTP server earns the label "web server." When this is done, you can point a browser at your server and see actual HTML, CSS, images, and JavaScript render correctly. Every file-serving feature a browser expects — correct content types, conditional caching, security boundaries — gets implemented here. There is one challenge in this milestone that towers above all others in importance. It is not MIME type detection, not `304 Not Modified` handling, not directory index serving. It is path security. Get this wrong and your server is not a web server — it's a file exfiltration tool waiting to be exploited.

The Revelation: String Prefix Checking Is Not Security

Here is the mental model that almost every developer reaches for when they first think about directory traversal prevention:

```c
// "Surely this is enough?"
if (strncmp(resolved_path, document_root, strlen(document_root)) == 0) {
    // Path starts with the root, so it must be inside the root. Right?
    serve_file(resolved_path);
}
```

This feels airtight. If the path starts with `/var/www/html`, it must be *inside* `/var/www/html`. The logic seems inescapable. Let's watch it get bypassed in five different ways.

Attack 1: The Classic `../` Traversal

A browser requests: `GET /../../etc/passwd HTTP/1.1` Your Milestone 2 parser stores this in `req.path` as `/../../etc/passwd`. You prepend the document root:

```
/var/www/html + /../../etc/passwd = /var/www/html/../../etc/passwd
```

Your `strncmp` check runs: does `/var/www/html/../../etc/passwd` start with `/var/www/html`? **Yes, it does.** The string prefix check passes. You call `open("/var/www/html/../../etc/passwd", O_RDONLY)` — which the kernel resolves as `/etc/passwd`. The contents of the system password file stream out across the network.

Attack 2: URL-Encoded Traversal

HTTP allows percent-encoding in URIs. The character `.` is ASCII 0x2E, which can be encoded as `%2e`. The character `/` is ASCII 0x2F, which can be encoded as `%2f`. So `../` becomes `%2e%2e%2f`. A browser requests: `GET /%2e%2e%2fetc%2fpasswd HTTP/1.1` If your parser stores the path without URL-decoding it, `strncmp` against the root might fail (the path doesn't look like it traverses). But when you concatenate and open, the kernel doesn't know about percent-encoding — it just opens the literal filename `%2e%2e%2fetc%2fpasswd`. Your server returns 404 (no such file). So percent-encoding actually harms the attacker here... unless your parser URL-decodes the path first. And a correct HTTP implementation *must* URL-decode the path before filesystem operations, because `%20` (space) is a legitimate encoding and `GET /my%20files/photo.jpg` must work. So the scenario is: your parser correctly URL-decodes `%2e%2e%2f` to `../`, and now you're back to Attack 1.

Attack 3: Symlink Bypass

Even if the path looks clean — `/var/www/html/images/photo.jpg`, no dots, no traversal — a symlink can redirect you. If an attacker (or even a careless administrator) created a symbolic link:

```bash
ln -s /etc /var/www/html/secret
```

Then `GET /secret/passwd HTTP/1.1` resolves as follows: your string prefix check sees `/var/www/html/secret/passwd`, which starts with `/var/www/html`. Check passes. You open it. The kernel follows the symlink: `/var/www/html/secret` points to `/etc`, so you open `/etc/passwd`.

Attack 4: Double Encoding

Some buggy URL processors decode `%25` to `%` in a first pass, then decode `%2e` to `.` in a second pass. So `%252e%252e%252f` (where `%25` encodes a literal `%`) decodes to `../` after two passes. If your URL decoder makes multiple passes or if there's a secondary layer of decoding (e.g., in a proxy), this attack works.

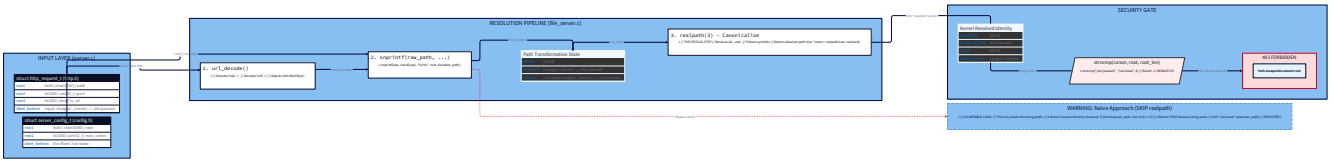
Attack 5: Unicode Normalization

On case-insensitive filesystems (macOS HFS+, Windows NTFS), the path `/VAR/WWW/HTML/../../ETC/PASSWD` might resolve correctly even if your check is case-sensitive. Your prefix check fails (different case), you reject it — but had you been on a case-insensitive system and not normalized case, this would be an inconsistency to exploit.

The Only Safe Approach: Canonicalize First, Check Second

The lesson from all five attacks is identical: **you cannot reason about path security by looking at the path string.** The path string is attacker-controlled input. The only trustworthy entity is the operating system's path resolver, which accounts for `..`, `.`, symlinks, and all other filesystem indirections. The safe algorithm has exactly three steps, in exactly this order:

```
1. URL-decode the request path
2. Canonicalize via realpath() → get the true absolute filesystem path
3. Check if the canonicalized path starts with the canonicalized document root
```



`realpath()` is the POSIX function that does everything the kernel does when resolving a path: it follows all symlinks, resolves all `.` and `..` components, and returns the absolute, canonical path. No matter how many layers of trickery an attacker stacks, `realpath()` returns what the kernel would actually open. Here is the critical constraint: `realpath()` requires the file to **exist**. If the file doesn't exist, `realpath()` returns `NULL` with `errno = ENOENT`. This means your path security check and your "file exists" check happen simultaneously — which is actually a feature, not a limitation.

```c
// URL-decode a percent-encoded string.
// Writes decoded result into `out` (max `out_size` bytes including null terminator).
// Returns 0 on success, -1 if `out` is too small or encoding is invalid.
int url_decode(const char *src, char *out, size_t out_size) {
    size_t i = 0;
    size_t j = 0;
    while (src[i] != '\0') {
        if (j + 1 >= out_size) return -1;  // Output buffer full
        if (src[i] == '%') {
            // Expect exactly two hex digits following %
            if (src[i+1] == '\0' || src[i+2] == '\0') return -1;
            char hex[3] = { src[i+1], src[i+2], '\0' };
            char *end;
            long val = strtol(hex, &end, 16);
            if (*end != '\0') return -1;  // Non-hex characters
            if (val == 0) return -1;      // Null byte injection: %00 is a classic attack
            out[j++] = (char)val;
            i += 3;
        } else if (src[i] == '+') {
            // '+' means space in application/x-www-form-urlencoded, but NOT in URI paths.
            // In URI paths, '+' is a literal plus sign.
            out[j++] = '+';
            i++;
        } else {
            out[j++] = src[i++];
        }
    }
```

```
    out[j] = '\0';

    return 0;

}
```

Note the null byte check: `%00` decodes to ASCII 0, the null byte. In C, strings are null-terminated, so a path like `/etc/passwd%00.html` would pass extension-based MIME checking (looks like `.html`) but when the null byte is decoded, the resulting C string `"/etc/passwd\0.html"` passes to `open()` as `"/etc/passwd"`. This attack has been used in real vulnerabilities. Rejecting `%00` is essential. Now the complete path resolution and security check:

```c
#define MAX_PATH_SIZE 4096  // PATH_MAX on Linux

// Resolve a URL path to an absolute filesystem path, verifying it

// stays within the document root.

//

// `url_path`      — the request path from http_request_t.path (e.g. "/images/logo.png")

// `document_root` — canonicalized document root (from realpath() at startup)

// `resolved_out`  — output buffer for the safe filesystem path (PATH_MAX bytes)

//

// Returns 0 on success (safe path in resolved_out).

// Returns 403 if path escapes document root.

// Returns 404 if path does not exist on the filesystem.

// Returns 400 if path is malformed (URL decode failure, etc.).

int resolve_safe_path(const char *url_path, const char *document_root,

                      char *resolved_out) {

    // Step 1: URL-decode the path

    char decoded[MAX_PATH_SIZE];

    if (url_decode(url_path, decoded, sizeof(decoded)) < 0) {

        return 400;  // Malformed percent-encoding

    }

    // Step 2: Build the raw filesystem path by joining document_root + decoded_path.

    // We're not checking security yet — this is just string concatenation.

    char raw[MAX_PATH_SIZE * 2];

    int n = snprintf(raw, sizeof(raw), "%s%s", document_root, decoded);

    if (n < 0 || (size_t)n >= sizeof(raw)) {

        return 400;  // Path too long

    }

    // Step 3: Canonicalize via realpath().
```

```
    // This resolves all symlinks, .., ., and relative components.
    // Returns NULL if the path does not exist on the filesystem.
    char *canon = realpath(raw, resolved_out);
    if (canon == NULL) {
        // ENOENT = path doesn't exist → 404
        // EACCES = permission denied → 403
        // Other errors → treat as 404 (don't leak internal details)
        if (errno == ENOENT) return 404;
        if (errno == EACCES) return 403;
        return 404;
    }
    // Step 4: AFTER canonicalization, verify the resolved path is still
    // inside the document root. This is the security check.
    size_t root_len = strlen(document_root);
    if (strncmp(resolved_out, document_root, root_len) != 0) {
        // Path escaped the document root
        return 403;
    }
    // Ensure it's either exactly the root or the next char is '/'
    // (prevents matching /var/www-secret when root is /var/www)
    char next = resolved_out[root_len];
    if (next != '\0' && next != '/') {
        return 403;
    }
    return 0;  // Safe path is in resolved_out
}
```

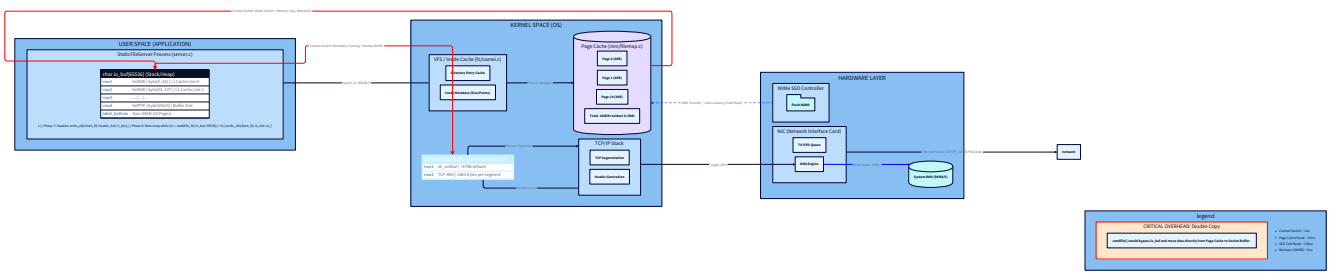
The final check — `next != '\0' && next != '/'` — prevents a subtle bypass. If your document root is `/var/www` and the resolved path is `/var/www-backup/passwords.txt`, a naive prefix check (`strncmp(resolved, root, root_len)`) would pass. Adding the check that the character after the root prefix must be `/` or end of string prevents this. **Initialize the document root at server startup**, not per-request:

```c
// In server initialization — run once when the server starts.

// Pass this canonical_root to resolve_safe_path() for every request.

char canonical_root[MAX_PATH_SIZE];

if (realpath(document_root_arg, canonical_root) == NULL) {

    perror("realpath (document root)");

    exit(1);  // Can't serve files without a valid document root

}

printf("Document root: %s\n", canonical_root);
```

Running `realpath()` on the document root at startup canonicalizes it once. If the document root contains a symlink (e.g., `/var/www/html` → `/data/www`), `canonical_root` will contain `/data/www` — the true path. Every per-request check then compares against this true path.

Three-Level View: What Happens When You Open a File

Take a request for `/images/photo.jpg` (100KB PNG image) and trace it through all three levels.



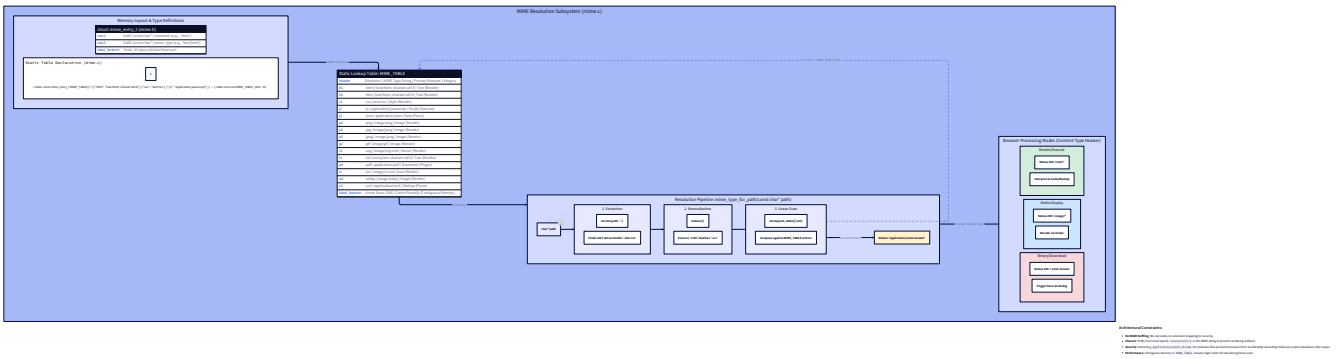
Level 1 — Your code: You call `resolve_safe_path("/images/photo.jpg", canonical_root, resolved)`, which calls `realpath()` and returns 0 with `resolved =`

`"/var/www/html/images/photo.jpg"`. You call `stat(resolved, &st)` to get the file size and modification time. You call `open(resolved, O_RDONLY)` to get a file descriptor. You call `read(file_fd, buf, buf_size)` in a loop to fill your buffer. You call `write(client_fd, ...)` to send the data. You call `close(file_fd)`. Level 2 — OS/Kernel: `realpath()` is implemented in user-space (glibc) but calls multiple `lstat()` syscalls — one per path component — to check each component for symlinks. For `/var/www/html/images/photo.jpg`, that's approximately five `lstat()` calls. `open()` triggers the kernel's VFS (Virtual File System) layer. The kernel looks up the file in the directory entry cache (dcache) — if `/var/www/html/images` has been recently accessed, the directory entry is already in kernel memory (hot). The kernel also checks the inode cache (icache): the inode stores the file's metadata (size, permissions, timestamps). On a busy server, both caches are hot and `open()` returns in microseconds without touching disk. `read()` triggers the kernel's page cache lookup. The page cache is a portion of RAM that holds recently-read disk blocks. If `photo.jpg` was read recently, its pages are in the page cache and `read()` is just a `memcpy()` from kernel memory to your user-space buffer. If the file is cold (not cached), the kernel issues disk I/O — an asynchronous read from the block device — and your process blocks until the data arrives. Level 3 — Hardware: For a cache-hot file: your `read()` triggers a context switch from user mode to kernel mode (the `read` syscall). The kernel finds the page cache entries for `photo.jpg`, copies them into your buffer via memory-to-memory copy. For a 100KB file, this is copying ~25 4KB pages. At memory bandwidth of ~20 GB/s (DDR4), 100KB copies in ~5 microseconds. Context switch overhead is ~1–2 microseconds. Total: ~7 microseconds for a cached read. For a cold file: the kernel submits an I/O request to the storage controller. An NVMe SSD responds in ~100

microseconds; a spinning HDD in ~5 milliseconds. Your process is moved off the CPU runqueue into a "waiting for I/O" state. The CPU runs other processes. When the storage interrupt fires, the kernel moves your process back to runnable. This is the 10ms vs 100ns gap — 100,000× difference between memory and disk — that the page cache exists to bridge.

MIME Types: Telling Browsers How to Interpret Bytes

Raw bytes have no intrinsic meaning. A sequence of bytes could be a UTF-8 HTML document, a JPEG image, a JavaScript program, or binary executable code. The browser has no way to tell which kind of content it received unless the server tells it. The **Content-Type header** carries this information.



MIME (**Multipurpose Internet Mail Extensions**) is a system originally designed for email attachments that HTTP adopted. A MIME type has the form `type/subtype`, optionally with parameters. `text/html; charset=utf-8` means "the type is text, the subtype is HTML, and the character encoding is UTF-8." The browser uses this to decide whether to render HTML, display an image, execute JavaScript, or prompt a download dialog. Getting MIME types wrong causes real user-visible failures:

- Serving `.js` files as `text/plain` causes browsers to refuse to execute the script (modern browsers block cross-origin scripts with wrong MIME types — **MIME-sniffing** protection)
- Serving `.css` as `text/plain` causes browsers to ignore the stylesheet
- Serving images as `application/octet-stream` causes download prompts instead of display The cleanest implementation is a sorted array of `{extension, mime_type}` pairs that you search with binary search or a simple linear scan. With only ~15 extensions to support, linear scan is fast enough (15 comparisons, all L1-resident):

```c
typedef struct {
    const char *extension;   // e.g. ".html" (with leading dot, lowercase)
    const char *mime_type;   // e.g. "text/html; charset=utf-8"
} mime_entry_t;

// MIME type table — searched linearly, so order doesn't matter.
// Extensions must be lowercase (we normalize before lookup).
static const mime_entry_t MIME_TABLE[] = {
    { ".html", "text/html; charset=utf-8"       },
    { ".htm",  "text/html; charset=utf-8"       },
    { ".css",  "text/css"                        },
    { ".js",   "application/javascript"          },
    { ".json", "application/json"                },
    { ".png",  "image/png"                       },
    { ".jpg",  "image/jpeg"                      },
    { ".jpeg", "image/jpeg"                      },
    { ".gif",  "image/gif"                       },
    { ".svg",  "image/svg+xml"                   },
    { ".txt",  "text/plain; charset=utf-8"       },
    { ".pdf",  "application/pdf"                 },
    { ".ico",  "image/x-icon"                    },
    { ".xml",  "application/xml"                 },
    { ".webp", "image/webp"                      },
};
static const int MIME_TABLE_LEN =
    (int)(sizeof(MIME_TABLE) / sizeof(MIME_TABLE[0]));
// Look up the MIME type for a filesystem path.
// Returns a MIME type string, or "application/octet-stream" as the fallback.
```

```
const char *mime_type_for_path(const char *path) {
    // Find the last '.' in the filename component
    const char *last_dot = strrchr(path, '.');
    if (last_dot == NULL) {
        return "application/octet-stream";
    }
    // Normalize extension to lowercase for comparison
    char ext_lower[32];
    size_t ext_len = strlen(last_dot);
    if (ext_len >= sizeof(ext_lower)) {
        return "application/octet-stream";  // Absurdly long extension
    }
    for (size_t i = 0; i < ext_len; i++) {
        char c = last_dot[i];
        ext_lower[i] = (c >= 'A' && c <= 'Z') ? (char)(c + 32) : c;
    }
    ext_lower[ext_len] = '\0';
    // Linear search — only ~15 entries, all in L1 cache
    for (int i = 0; i < MIME_TABLE_LEN; i++) {
        if (strcmp(ext_lower, MIME_TABLE[i].extension) == 0) {
            return MIME_TABLE[i].mime_type;
        }
    }
    return "application/octet-stream";  // Unknown extension: trigger download
}
```

`strrchr()` (string reverse character-find) finds the *last* occurrence of a character in a string. For a path like `/var/www/html/styles/main.min.css`, there are two dots. We want the *last* one to get `.css`, not `.min`. Using `strrchr()` instead of `strchr()` handles this correctly. `application/octet-stream` is the correct MIME type for "I don't know what this is." Browsers respond by presenting a download dialog, which is safer than trying to execute or display unknown binary content.

Conditional Requests: If-Modified-Since and 304 Not Modified

Once you understand that browsers and CDNs cache files, a new problem emerges: how does a browser know if the cached copy is still fresh? It could send `GET /logo.png` every time a user visits the page — but that's wasteful if the image hasn't changed in months. HTTP solves this with **conditional requests**. The protocol defines a handshake:

1. First request: server sends the file with a `Last-Modified` header indicating when the file was last changed
2. Subsequent requests: client sends `If-Modified-Since: <timestamp>` asking "has this changed since this time?"
3. If unchanged: server returns `304 Not Modified` with no body (just headers), saving bandwidth
4. If changed: server returns `200 OK` with the new file {{DIAGRAM:diag-m3-conditional-request-flow}} This optimization is significant in practice. For a page with 20 cacheable assets (CSS, JS, images), a returning visitor's browser might send 20 conditional requests. If none of the assets changed, all 20 get `304` responses. The browser loads the page entirely from cache — the responses contain no body bytes. Total bandwidth: a few kilobytes of headers vs. potentially megabytes of assets. The filesystem gives you the modification time of every file through `stat()`. The `struct stat` field `st_mtim` (or `st_mtime` on some platforms) contains the last modification time as a Unix timestamp (seconds since January 1, 1970 UTC). The HTTP `Last-Modified` and `If-Modified-Since` header format is **RFC 7231 HTTP-date format**: `"Day, DD Mon YYYY HH:MM:SS GMT"`. This is the same format we used for the `Date` header in Milestone 1.

```c
// Format a time_t as an HTTP-date string.
// Writes into `buf` (at least 64 bytes).
// Returns 0 on success, -1 on failure.
int format_http_date(time_t t, char *buf, size_t buf_size) {
    struct tm *gmt = gmtime(&t);
    if (gmt == NULL) return -1;
    size_t n = strftime(buf, buf_size, "%a, %d %b %Y %H:%M:%S GMT", gmt);
    return (n > 0) ? 0 : -1;
}
// Parse an HTTP-date string back to time_t.
// Returns the parsed time on success, (time_t)-1 on failure.
time_t parse_http_date(const char *date_str) {
    struct tm tm;
    memset(&tm, 0, sizeof(tm));
    // strptime parses a date string according to a format, inverse of strftime.
    // The %a (weekday), %d (day), %b (month name), %Y (year), %H:%M:%S (time)
    // format matches the standard HTTP-date format.
    const char *result = strptime(date_str, "%a, %d %b %Y %H:%M:%S GMT", &tm);
    if (result == NULL || *result != '\0') {
        return (time_t)-1;  // Parse failed
    }
    // timegm() converts struct tm (in UTC/GMT) to time_t.
    // mktime() assumes local time — WRONG for HTTP dates which are always UTC.
    return timegm(&tm);
}
```

`timegm()` is the inverse of `gmtime()`: it converts a `struct tm` interpreted as UTC into a `time_t`. **Do not use `mktime()` here.** `mktime()` interprets `struct tm` as local time — if your server is in a non-UTC

timezone, `mktime()` will give you the wrong result, causing incorrect cache validation. HTTP dates are always UTC. `timegm()` is available on Linux and macOS (as a BSD extension); if you need full POSIX portability, set the `TZ` environment variable temporarily or use a manual calculation. Now the conditional request check:

```c
// Check whether a conditional GET/HEAD should return 304 Not Modified.
// `file_mtime` is the file's last modification time (from stat).
// `req` is the parsed HTTP request.
// Returns 1 if 304 should be sent, 0 if the full response should be sent.
int should_send_304(const http_request_t *req, time_t file_mtime) {
    const char *ims = request_get_header(req, "if-modified-since");
    if (ims == NULL) return 0;  // No conditional header: send full response
    time_t client_time = parse_http_date(ims);
    if (client_time == (time_t)-1) return 0;  // Can't parse: send full response
    // The file has not been modified if its mtime <= the time the client
    // says it last received the file. Use <= not <, because HTTP-date
    // resolution is one second — a file modified in the same second as
    // the If-Modified-Since timestamp is considered unchanged.
    return (file_mtime <= client_time) ? 1 : 0;
}
```

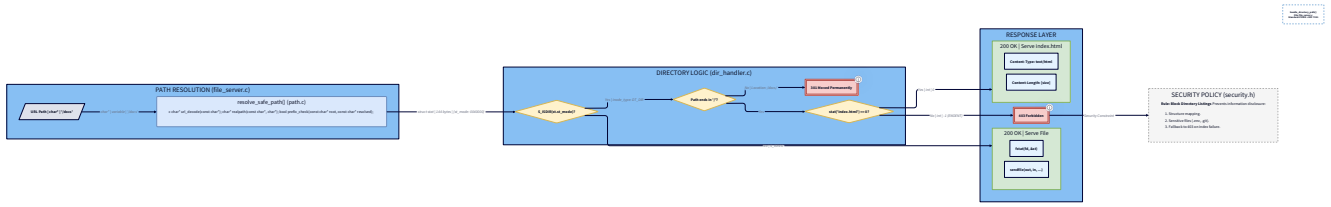
The `<=` comparison handles the one-second granularity of HTTP-date format. If a file was modified at timestamp T, the server sends `Last-Modified: <T formatted to second precision>`. On the next request, the client sends `If-Modified-Since: <T>`. If you used `<` instead of `<=`, you'd send a

full `200` response for a file modified at exactly the cached timestamp — wasting bandwidth and incorrectly indicating the file changed.

Directory Requests and Index Files

When a browser requests a URL ending in `/` — or a URL that maps to a directory — your server has a choice: list the directory contents (a **directory listing**), serve a default file, or refuse. Modern practice avoids directory listings in production (they expose file structure to attackers). The conventional behavior is:

1. If the path maps to a directory and `index.html` exists inside it → serve `index.html`
2. If the path maps to a directory and `index.html` does not exist → return 403 Forbidden
3. If the path maps to a regular file → serve it normally



The detection uses `stat()` and the `S_ISDIR()` macro:

C

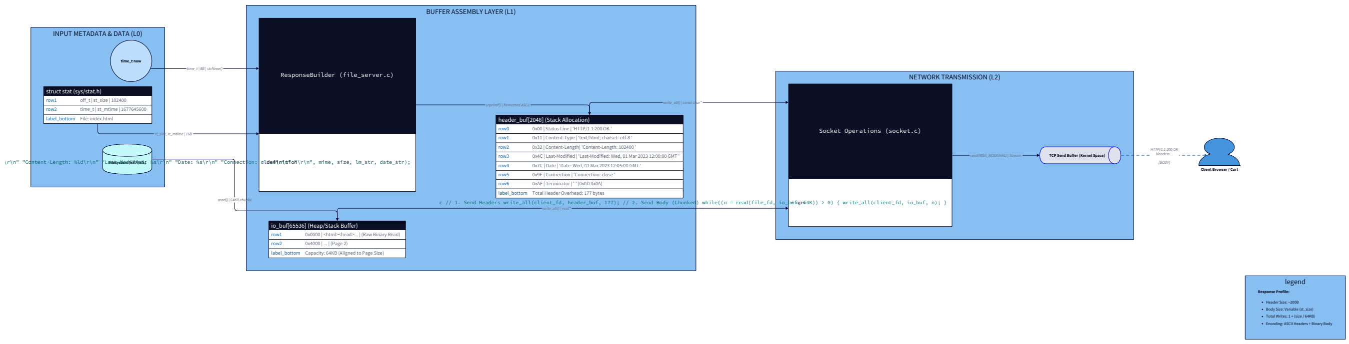
```c
// Detect if a path is a directory and append "/index.html" if so.
// Modifies `resolved_path` in-place (must have room for appending "/index.html").
// Returns:
//   0 = regular file, serve it
//   1 = was a directory, index.html appended, re-stat needed
//  -1 = directory with no index.html → send 403
//  -2 = stat() failed → send 404
int handle_directory_path(char *resolved_path, size_t path_buf_size,
                          struct stat *st) {
    if (!S_ISDIR(st->st_mode)) {
        return 0;  // Regular file — serve directly
    }
    // It's a directory — try to append /index.html
    size_t current_len = strlen(resolved_path);
    const char *index_suffix = "/index.html";
    size_t suffix_len = strlen(index_suffix);
    if (current_len + suffix_len + 1 > path_buf_size) {
        return -1;  // No room to append — refuse
    }
    // Ensure path doesn't already end with '/'
    if (resolved_path[current_len - 1] == '/') {
        // Overwrite the trailing slash
        strncat(resolved_path + current_len - 1,
                index_suffix, path_buf_size - current_len);
    } else {
        strncat(resolved_path + current_len,
                index_suffix, path_buf_size - current_len - 1);
```

```
    }

    // Re-stat the index.html path

    if (stat(resolved_path, st) < 0) {

        if (errno == ENOENT) return -1;  // No index.html → 403

        return -2;  // Other error → 404

    }

    if (S_ISDIR(st->st_mode)) {

        return -1;  // index.html is itself a directory? → 403

    }

    return 1;  // index.html found and stat'd — proceed to serve

}
```

`S_ISDIR(st->st_mode)` — the `stat()` syscall fills a `struct stat` with file metadata. The `st_mode` field is a bitmask encoding both the file type and permission bits. `S_ISDIR()` is a POSIX macro that extracts the file-type bits and checks if they indicate a directory. Similarly, `S_ISREG()` checks for regular files, `S_ISLNK()` for symbolic links (though `stat()` follows symlinks automatically — use `lstat()` to detect symlinks without following them).

The Complete File Serving Function

Now assemble all the pieces into the function that handles an HTTP request end-to-end: path resolution → security check → directory handling → conditional request check → response assembly → file transfer.




```
\r\n" "Content-Length: %ld\r\n" "Last-Modified: %s\r\n" "Date: %s\r\n" "Connection: close\r\n\r\n", mime, size, lm_str, date_str);
```

```
c // 1. Send Headers write_all(client_fd, header_buf, 177); // 2. Send Body (Chunked) while((n = read(file_fd, io_buf, 64K)) > 0) { write_all(client_fd, io_buf, n); }
```

```c
// Send a complete HTTP error response (reused from Milestone 2)
// — defined in your existing codebase.
void send_error_response(int client_fd, int status_code,
                         const char *reason, const char *body);

// Send the complete HTTP response for a file request.
// Handles GET and HEAD methods, conditional requests, all error cases.
// `req`           — the parsed HTTP request (from Milestone 2)
// `client_fd`     — the connection socket to write the response to
// `canonical_root`— the server's canonicalized document root
void serve_static_file(const http_request_t *req, int client_fd,
                       const char *canonical_root) {
    // ── Phase 1: Resolve and validate the filesystem path ──────────────
    char resolved[MAX_PATH_SIZE];
    int resolve_status = resolve_safe_path(req->path, canonical_root, resolved);
    if (resolve_status == 400) {
        send_error_response(client_fd, 400, "Bad Request",
            "<html><body><h1>400 Bad Request</h1>"
            "<p>Malformed request path.</p></body></html>");
        return;
    }
    if (resolve_status == 403) {
        send_error_response(client_fd, 403, "Forbidden",
            "<html><body><h1>403 Forbidden</h1>"
            "<p>Access to the requested resource is not permitted.</p>"
            "</body></html>");
        return;
    }
```

```
    if (resolve_status == 404) {
        send_error_response(client_fd, 404, "Not Found",
            "<html><body><h1>404 Not Found</h1>"
            "<p>The requested resource could not be found.</p>"
            "</body></html>");
        return;
    }
    // ── Phase 2: Stat the file ──────────────────────────────────────
    struct stat st;
    if (stat(resolved, &st) < 0) {
        int status = (errno == ENOENT) ? 404 : 403;
        const char *reason = (status == 404) ? "Not Found" : "Forbidden";
        const char *body = (status == 404)
            ? "<html><body><h1>404 Not Found</h1></body></html>"
            : "<html><body><h1>403 Forbidden</h1></body></html>";
        send_error_response(client_fd, status, reason, body);
        return;
    }
    // ── Phase 3: Handle directory paths ─────────────────────────────
    int dir_result = handle_directory_path(resolved, sizeof(resolved), &st);
    if (dir_result < 0) {
        // -1: directory with no index.html
        // -2: stat failed
        send_error_response(client_fd, 403, "Forbidden",
            "<html><body><h1>403 Forbidden</h1>"
            "<p>Directory listing is not permitted.</p></body></html>");
        return;
```

```
    }

    // If dir_result == 1, resolved now points to index.html and st is re-stat'd.

    // Verify it's a regular file (not a special file like /dev/random)

    if (!S_ISREG(st.st_mode)) {

        send_error_response(client_fd, 403, "Forbidden",

            "<html><body><h1>403 Forbidden</h1></body></html>");

        return;

    }

    // ── Phase 4: Determine MIME type ──────────────────────────────────

    const char *mime_type = mime_type_for_path(resolved);

    // ── Phase 5: Check conditional request (If-Modified-Since) ────────

    time_t file_mtime = st.st_mtime;

    if (should_send_304(req, file_mtime)) {

        // Build 304 response — headers only, no body

        char date_str[64];

        char lm_str[64];

        format_http_date(time(NULL), date_str, sizeof(date_str));

        format_http_date(file_mtime, lm_str, sizeof(lm_str));

        char resp[1024];

        int n = snprintf(resp, sizeof(resp),

            "HTTP/1.1 304 Not Modified\r\n"

            "Last-Modified: %s\r\n"

            "Date: %s\r\n"

            "Connection: %s\r\n"

            "\r\n",

            lm_str, date_str,

            req->keep_alive ? "keep-alive" : "close");
```

```
        if (n > 0) write_all(client_fd, resp, (size_t)n);
        return;
    }

    // ── Phase 6: Open the file ──────────────────────────────────
    int file_fd = open(resolved, O_RDONLY);
    if (file_fd < 0) {
        int status = (errno == ENOENT) ? 404 : 403;
        const char *reason = (status == 404) ? "Not Found" : "Forbidden";
        const char *body = (status == 404)
            ? "<html><body><h1>404 Not Found</h1></body></html>"
            : "<html><body><h1>403 Forbidden</h1></body></html>";
        send_error_response(client_fd, status, reason, body);
        return;
    }

    // ── Phase 7: Build and send response headers ────────────────
    char date_str[64];
    char lm_str[64];
    format_http_date(time(NULL), date_str, sizeof(date_str));
    format_http_date(file_mtime, lm_str, sizeof(lm_str));
    long file_size = (long)st.st_size;
    char header_buf[2048];
    int header_len = snprintf(header_buf, sizeof(header_buf),
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: %s\r\n"
        "Content-Length: %ld\r\n"
        "Last-Modified: %s\r\n"
        "Date: %s\r\n"
```

```
        "Connection: %s\r\n"
        "\r\n",
        mime_type,
        file_size,
        lm_str,
        date_str,
        req->keep_alive ? "keep-alive" : "close");
    if (header_len <= 0 || (size_t)header_len >= sizeof(header_buf)) {
        close(file_fd);
        send_error_response(client_fd, 500, "Internal Server Error",
            "<html><body><h1>500 Internal Server Error</h1></body></html>");
        return;
    }
    if (write_all(client_fd, header_buf, (size_t)header_len) < 0) {
        close(file_fd);
        return;  // Client disconnected
    }
    // ── Phase 8: Send the file body (skip for HEAD requests) ──────────
    if (req->method == METHOD_HEAD) {
        close(file_fd);
        return;  // HEAD: headers sent, no body
    }
    // Transfer file contents in chunks
    char io_buf[65536];  // 64KB I/O buffer — matches typical page cache read size
    ssize_t bytes_read;
    while ((bytes_read = read(file_fd, io_buf, sizeof(io_buf))) > 0) {
        if (write_all(client_fd, io_buf, (size_t)bytes_read) < 0) {
```

```
            break;  // Client disconnected mid-transfer — just stop
        }
    }
    close(file_fd);
}
```

The 64KB Buffer Choice

The I/O buffer size of 65536 bytes (64KB) is not arbitrary. Several hardware and OS factors converge at this size:

- **Page cache granularity**: The kernel reads files in units of its memory pages (4KB on x86). The `read()` syscall, when serving from the page cache, can transfer up to the buffer size in one call. 64KB = 16 pages — a good batch size.
- **TCP send buffer**: The kernel's TCP send buffer default is typically 87KB. Sending 64KB at a time keeps the pipeline full without overflowing.
- **L1 cache pressure**: 64KB fits in L2 cache (typically 256KB–512KB) without evicting the rest of your working set from L1. Larger buffers start competing with other hot data. At smaller buffer sizes (4KB, 8KB), the overhead of the read→write loop — two syscalls per iteration, with their associated mode switches between user space and kernel space — becomes a bottleneck. At larger buffer sizes (256KB+), the buffer itself starts evicting hot data from cache.

TOCTOU: The Security Race You Can't Fully Win

Even with perfect `realpath()` usage, there is a theoretical security vulnerability worth understanding: the **Time-Of-Check-Time-Of-Use (TOCTOU)** race condition. Here's the scenario:

1. Your code calls `realpath("/var/www/html/safe.txt")` → verifies path is inside root ✓
2. An attacker (with filesystem access) replaces `/var/www/html/safe.txt` with a symlink to `/etc/passwd`
3. Your code calls `open("/var/www/html/safe.txt")` → now opens `/etc/passwd` The check (`realpath`) and the use (`open`) are two separate operations. Between them, the filesystem can change. This is the time-of-check-time-of-use problem. In practice, exploiting this requires the attacker to already have filesystem write access to your web root — if they do, they've already won (they can just read the file directly, or plant malicious content). So TOCTOU in this specific scenario is not a meaningful threat model for a static file server. However, the TOCTOU concept itself is one of the most important patterns in systems security, appearing in:

- **setuid programs**: check permissions as real user, use resources as root user
- **Database isolation levels**: read-then-write transactions with intervening modifications (phantom reads)
- **Container escape vulnerabilities**: checking container identity before acting on it
- **File upload handlers**: validate file type, then move — attacker replaces file between steps You cannot eliminate TOCTOU without atomic operations. The Linux kernel provides `openat()` with directory file descriptors, which reduces (but doesn't eliminate) the window. For our static file server, the threat model doesn't require solving TOCTOU — but knowing it exists makes you a more careful systems programmer.

Hardware Soul: Cache Lines and File Transfer

`realpath()` — multiple syscalls: Each `lstat()` call inside `realpath()` crosses the user/kernel boundary (~1–2 microseconds for the context switch) and performs a dcache lookup. For a path with N components, this is N × 2µs. For `/var/www/html/images/photo.jpg` (5 components), `realpath()` costs approximately 10 microseconds — dominated by system call overhead, not computation. **`stat()` memory access pattern**: `struct stat` is 144 bytes on Linux x86-64. It fits entirely in L1 cache. You call `stat()` once per request; the kernel fills it in during the syscall. There's no cache-line bouncing concern here — it's a small, write-once, read-several-times structure. **File read loop — cache line efficiency**: Your 64KB read buffer crosses 1024 cache lines (64 bytes each). The CPU's hardware prefetcher recognizes the sequential access pattern — you read from offset 0, 64, 128, ... in order — and prefetches the next cache line before you need it. This is a **stream access pattern**: the most cache-friendly access pattern possible, achieving near-peak memory bandwidth. The prefetcher would fail on random access (serving random bytes from different files), which is why file serving is faster than you might expect even when the data isn't already in L1. **`write()` and socket buffers**: When you call `write(client_fd, io_buf, 65536)`, the kernel copies your buffer into the socket's send buffer (in kernel memory), then returns. The TCP stack asynchronously segments and transmits from the send buffer. If the send buffer is full (the client is reading slowly), `write()` blocks. For a local loopback connection, the kernel's loopback interface operates at memory bandwidth speeds — easily 10+ GB/s. The bottleneck for local file serving is never the network. **The case for `sendfile()`**: In the serve loop above, you perform two copies for each chunk of file data:

1. `read()`: disk/page-cache → your userspace buffer (kernel mode → user mode copy)
2. `write()`: your userspace buffer → socket send buffer (user mode → kernel mode copy) The `sendfile()` syscall (Linux-specific) eliminates both copies. It tells the kernel: "copy from this file FD directly to this socket FD without touching userspace." The kernel can use DMA (Direct Memory Access) to copy from the page cache to the NIC's DMA buffer directly, bypassing the CPU entirely for the data movement. For large files on high-bandwidth connections, `sendfile()` doubles throughput and halves CPU usage. Your current implementation is educationally valuable because it shows you exactly where the copies happen. After this milestone, `sendfile()` is the natural next optimization.

Integrating File Serving into the Accept Loop

Update `handle_connection()` from Milestone 2 to dispatch to `serve_static_file()`:

C

```c
// Updated handle_connection — replaces the dispatch section from Milestone 2
void handle_connection(int client_fd, const char *canonical_root) {
    char req_buf[REQUEST_BUF_SIZE];
    int header_end = -1;
    ssize_t bytes_read = read_http_request(client_fd, req_buf,
                                           REQUEST_BUF_SIZE, &header_end);
    if (bytes_read < 0 || header_end < 0) {
        close(client_fd);
        return;
    }
    http_request_t req;
    int parse_err = parse_http_request(req_buf, (size_t)bytes_read,
                                       header_end, &req);
    if (parse_err != 0) {
        switch (parse_err) {
            case 400: send_error_response(client_fd, 400, "Bad Request",
                          "<html><body><h1>400 Bad Request</h1></body></html>");
                      break;
            case 414: send_error_response(client_fd, 414, "URI Too Long",
                          "<html><body><h1>414 URI Too Long</h1></body></html>");
                      break;
            case 501: send_error_response(client_fd, 501, "Not Implemented",
                          "<html><body><h1>501 Not Implemented</h1></body></html>");
                      break;
            default:  send_error_response(client_fd, 400, "Bad Request",
                          "<html><body><h1>400 Bad Request</h1></body></html>");
        }
```

```
        close(client_fd);
        return;
    }
    // Only serve GET and HEAD — POST and others get 501 (parser sets METHOD_UNKNOWN)
    if (req.method != METHOD_GET && req.method != METHOD_HEAD) {
        send_error_response(client_fd, 501, "Not Implemented",
            "<html><body><h1>501 Not Implemented</h1></body></html>");
        close(client_fd);
        return;
    }
    serve_static_file(&req, client_fd, canonical_root);
    close(client_fd);
}
```

Pass `canonical_root` from your server initialization (where you called `realpath()` on the document root at startup) into every connection handler. In Milestone 4, this will move into a shared server configuration struct accessible to all worker threads.

Testing Static File Serving

Create a test document root with a representative set of files:

```bash
mkdir -p /tmp/www/images

echo '<html><body><h1>Home</h1></body></html>' > /tmp/www/index.html

echo 'body { color: red; }' > /tmp/www/style.css

echo 'console.log("hello");' > /tmp/www/app.js

cp /usr/share/pixmaps/some-icon.png /tmp/www/images/logo.png  # or any PNG

echo '{"status": "ok"}' > /tmp/www/api.json

# Run server

./http_server 8080 /tmp/www
```

BASH

```
# Test 1: Serve HTML with correct Content-Type

curl -v http://localhost:8080/

# Expect: HTTP/1.1 200 OK, Content-Type: text/html; charset=utf-8

# Test 2: Serve CSS

curl -v http://localhost:8080/style.css

# Expect: Content-Type: text/css

# Test 3: Last-Modified header present

curl -v http://localhost:8080/index.html | grep -i last-modified

# Expect: Last-Modified: <some valid HTTP date>

# Test 4: Conditional request — first fetch, then conditional

LMOD=$(curl -sI http://localhost:8080/index.html | grep -i last-modified | cut -d' ' -f2-)

curl -v -H "If-Modified-Since: $LMOD" http://localhost:8080/index.html

# Expect: HTTP/1.1 304 Not Modified, no body

# Test 5: 404 for missing file

curl -v http://localhost:8080/does-not-exist.html

# Expect: HTTP/1.1 404 Not Found

# Test 6: Directory traversal — MUST return 403

curl -v http://localhost:8080/../../../etc/passwd

# Expect: HTTP/1.1 403 Forbidden (curl may normalize the path — use nc)

printf "GET /../../../etc/passwd HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc localhost 8080

# Expect: 403 Forbidden, no file content

# Test 7: URL-encoded traversal

printf "GET /%2e%2e%2f%2e%2e%2fetc%2fpasswd HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc
localhost 8080

# Expect: 403 Forbidden

# Test 8: Directory without index.html

mkdir /tmp/www/empty-dir
```

```
curl -v http://localhost:8080/empty-dir/

# Expect: 403 Forbidden

# Test 9: Directory with index.html

curl -v http://localhost:8080/

# Expect: 200 OK with the content of /tmp/www/index.html

# Test 10: HEAD request returns headers, no body

curl -v -X HEAD http://localhost:8080/index.html

# Expect: 200 OK with Content-Length header, zero bytes of body

# Test 11: Binary file (PNG) served correctly

curl -s http://localhost:8080/images/logo.png | file -

# Expect: PNG image data — not corrupted binary
```

Test 11 is important: verify that binary files are transmitted intact. A naive server that tries to process binary content as text can corrupt images. Ensure your read/write loop treats the data as raw bytes — which it does, since you're using `read()` and `write()` with `char` buffers, where every byte value 0–255 is valid.

Verifying Security Properties

BASH

```
# Verify realpath() is actually being called — strace on Linux

strace -e trace=lstat,open,read ./http_server 8080 /tmp/www &

printf "GET /../../../etc/passwd HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc localhost 8080

# Look for lstat() calls on path components in strace output

# Should NOT see open("/etc/passwd")

# Verify symlink traversal is blocked

ln -s /etc /tmp/www/secret-link

curl -v http://localhost:8080/secret-link/passwd

# Expect: 403 Forbidden

# realpath() resolves the symlink to /etc/passwd, which is outside /tmp/www
```

Design Decisions: Path Security Approaches

Approach	Security Level	Complexity	Used By
String prefix check only	None (bypassable)	Trivial	Buggy student servers
Manual `..` stripping	Weak (URL encoding bypasses)	Medium	Some legacy CGI servers
`realpath()` + prefix check (this project) ✓	Strong for file server	Low	nginx, Apache httpd
`chroot()` jail	Very strong (OS-enforced)	High	High-security file servers
`openat()` with O_NOFOLLOW	Strong (no TOCTOU)	Medium	Modern security-conscious servers
`chroot()` changes a process's view of the root filesystem — inside a chroot jail, the process cannot navigate above its "root." This is OS-enforced containment: even a compromised `realpath()` implementation can't escape a chroot. The downside is that it requires the server to be set up carefully (all needed libraries and files must be inside the jail) and typically requires root privileges to call `chroot()`. nginx and Apache can use `chroot` in production configurations.			
`openat()` with `O_NOFOLLOW` (don't follow symlinks on the final component) combined with `O_PATH` fd-based path traversal is the most modern approach — it eliminates TOCTOU by holding directory fds across the traversal. This is how container runtimes like `runc` handle their filesystem operations.			
For this milestone, `realpath()` + prefix check is the right choice: correct, auditable, and teachable.			

Knowledge Cascade: What This Unlocks

1. Defense in Depth — The Resolve-Then-Check Pattern Everywhere

The principle you just applied — canonicalize first, check second — is the foundation of every defense-in-depth filesystem security mechanism: **`chroot()` jails**: The kernel's `chroot()` syscall changes what "/" means for a process. Every path lookup inside a chroot is automatically constrained to the jail directory. The implementation is: when resolving a path, if the process's root is `/jail`, every absolute path lookup starts at `/jail`. `realpath()` inside the jail cannot escape because the kernel won't let it. Same principle: resolve identity through the system, then trust the result. **Linux namespaces and container filesystems**: Docker and container runtimes use mount namespaces to create a private filesystem view per container. When a process inside a container opens `/etc/passwd`, the kernel resolves that path in the container's namespace, which maps to a container-specific overlay filesystem layer. The security model is: trust the kernel's path resolution, not the string. **`seccomp` sandboxes**: Instead of path-level security, seccomp operates on syscall level — but the principle is the same. Don't trust the syscall arguments (the string passed to `open()`); instead, constrain which syscalls are permitted at all, forcing the program to operate within a defined safety envelope. Wherever you see security through containment — browser sandboxes (Chrome's site isolation uses process separation and IPC), database privilege levels (EXECUTE permission on stored procedures without SELECT access to underlying tables), AWS IAM policies — the pattern is: don't trust the name or the request, verify the resolved identity.

2. `sendfile()` and Zero-Copy I/O

Your current file transfer code has two copies per chunk: page cache → user buffer (`read()`), user buffer → socket send buffer (`write()`). Visualize those two copies as two context switches and two memcpy calls per 64KB chunk. `sendfile(output_fd, input_fd, offset, count)` is a Linux syscall that performs the transfer inside the kernel — no userspace buffer needed. The kernel can use DMA to move data directly from the disk controller's memory to the NIC's DMA buffer, bypassing the CPU for the data path entirely. For a 1MB file served over a fast connection:

- **With read/write**: 16 read() calls + 16 write() calls + 2MB of memcpy = ~30µs of CPU time
- **With sendfile()**: 1 syscall, kernel handles DMA = ~5µs of CPU time This is the technique behind nginx's legendary efficiency. nginx's inner loop for static file serving is essentially: open the file, `sendfile()` to the socket, close the file. No userspace buffer, no memcpy. The kernel does everything. The concept extends: `splice()` transfers data between two file descriptors (not necessarily a file and a socket), `tee()` duplicates pipe data, and `copy_file_range()` copies between two regular files — all in kernel space, all avoiding the userspace round-trip. Go's `net/http` server uses `sendfile()` automatically when serving files via `http.ServeFile()`. Rust's `tokio` async runtime exposes it through `tokio::fs::File`. Understanding why it exists — because the double-copy in read+write is wasteful physics — lets you recognize when to reach for it.

3. Browser Caching Architecture

The `If-Modified-Since` / `Last-Modified` / `304` flow you just implemented is the oldest layer of the web's caching stack. Understanding it bottom-up unlocks the entire architecture:

- **`Last-Modified` + `If-Modified-Since`**: Time-based validation. Granularity: one second. Weakness: if a file changes and reverts within one second, the cache won't invalidate. What you just built.
- **`ETag` + `If-None-Match`**: Hash-based validation. The server sends a hash of the file content as the `ETag`. Clients send `If-None-Match: <etag>` on subsequent requests. Hash comparison is unambiguous — any change, regardless of timestamp granularity, produces a different ETag. nginx computes ETags as `mtime-size` (modification time XOR file size).
- **`Cache-Control`**: Proactive caching directives. `Cache-Control: max-age=3600` tells the browser "don't even ask for 1 hour — assume it's fresh." No request is sent at all. `Cache-Control: no-cache` means "always validate but reuse if 304." `Cache-Control: no-store` means "never cache this."
- **CDN invalidation**: CDNs (Cloudflare, Fastly, AWS CloudFront) cache your responses at edge nodes globally. When you update a file, you send a "purge" API call to invalidate the CDN cache. The CDN then uses `If-Modified-Since` or `ETag` to revalidate with your origin server. The mechanics are exactly what you built — just distributed.
- **Service Workers**: A JavaScript-controlled cache layer in the browser that intercepts fetch requests. Service workers can implement arbitrary caching logic — but they use the same `Response` objects with the same `Last-Modified` and `ETag` semantics. Understanding the HTTP layer means understanding what service workers are working with.

4. MIME Types as Content Negotiation — The Simple Case of a Deep Concept

The `Content-Type` header you're setting per file extension is the simplest form of HTTP **content negotiation**. The general principle: the client and server negotiate what representation of a resource to use. MIME types for static files require no negotiation — the server picks the type based on the file. But the same infrastructure supports richer negotiation:

- **`Accept-Language: en-US, fr;q=0.9`**: Client prefers US English, accepts French at 90% quality. Server picks the best available language version of the resource.
- **`Accept-Encoding: gzip, deflate, br`**: Client can decompress these encodings. Server sends a gzip-compressed response with `Content-Encoding: gzip`, saving 70–80% of bandwidth for text files.
- **`Accept: application/json, text/html`**: REST APIs use this to serve JSON to API clients and HTML to browsers from the same URL.
- **API versioning via `Accept`**: `Accept: application/vnd.myapi.v2+json` — the client requests a specific API version through the Accept header. GitHub's API works this way. Every one of these is a generalization of the same concept you implemented: use metadata about the resource and the client's capabilities to pick the right representation.

5. `stat()` and the Unix Inode Model

`stat()` is a window into the Unix inode model. An **inode** (index node) is the kernel data structure that stores file metadata: size, permissions (`st_mode`), owner (`st_uid`), group (`st_gid`), timestamps (`st_atime` , `st_mtime` , `st_ctime`), and the block pointers that tell the kernel where the file's data lives on disk. Notably, the inode does *not* store the filename — that lives in the directory entry, which is a separate structure that maps names to inode numbers. This separation explains seemingly strange behaviors:

- You can have two filenames that point to the same inode (`hard links`). Both names share the same metadata, the same content, the same modification time.
- A file can be "deleted" (the directory entry removed) but its inode persists until all file descriptors to it are closed. This is why `unlink()` doesn't immediately free disk space if a process has the file open — the inode stays alive.
- `stat()` and `lstat()` differ: `stat()` follows symlinks to the target's inode; `lstat()` returns the symlink's own inode (with `st_mode` indicating it's a symlink). Understanding inodes is the foundation for understanding filesystem implementation (how SQLite manages its database file, how PostgreSQL manages tablespace files), for understanding why `mv` within a filesystem is instantaneous (renames the directory entry, doesn't move inode data), and for understanding why cross-filesystem `mv` is slow (must copy all data, create new inode, delete old entry).

Common Mistakes That Will Burn You

1. Calling `realpath()` on the document root per request, not at startup. `realpath()` on the document root doesn't change between requests (it's your configuration). Calling it per request adds 5–10 `lstat()` syscalls on every single request. Call it once at startup, store the result, pass it to every connection handler. 2. Checking the prefix before calling `realpath()`. The entire point of `realpath()` is that the raw path string cannot be trusted. Any check before `realpath()` is checking attacker-controlled data and provides no security guarantee. Canonicalize first. Always. 3. Text-mode reading for binary files. On Linux, `open()` with `O_RDONLY` gives you raw bytes — there's no "text mode" vs "binary mode" distinction at the syscall level. However, if you ever use `fopen()` with `"r"` instead of `"rb"`, some platforms (Windows) translate `\r\n`

sequences to `\n` in text mode, corrupting binary files. Use `open()` + `read()` directly (as in the code above), or use `fopen(..., "rb")` if you use stdio. Never use `"r"` for files that might be binary. 4. Using `st_mtime` without understanding its granularity. On some filesystems (FAT32, older HFS+), file modification times have 2-second or 1-second granularity. On ext4 and modern filesystems, `st_mtim.tv_nsec` gives nanosecond precision. But HTTP-date format only has 1-second resolution. When comparing `st_mtime` against an `If-Modified-Since` timestamp, use only the second-precision `st_mtime` (not `st_mtim.tv_nsec`), and use `<=` comparison as shown above. 5. Not closing `file_fd` on all error paths. Every early return after `file_fd = open(...)` must include `close(file_fd)`. Draw the control flow. If `write_all()` of the headers fails (client disconnected), you break out of the function — have you closed `file_fd`? Check every branch. 6. Forgetting to handle the path prefix edge case. The check `strncmp(resolved, root, root_len) == 0` passes for both `/var/www` (the root itself) and `/var/www-backup` (a different directory). Always verify that the character at `resolved[root_len]` is either `'/'` or `'\0'`. 7. Using `atoi()` for Content-Length in Milestone 2 (carry-forward) You already handle this, but if you missed it: `atoi("-1")` returns `-1`, and `atoi("9999999999999")` overflows silently on 32-bit `int`. `strtol()` with bounds checking is the only safe approach.

Acceptance Criteria Checklist

Before proceeding to Milestone 4, verify each item:

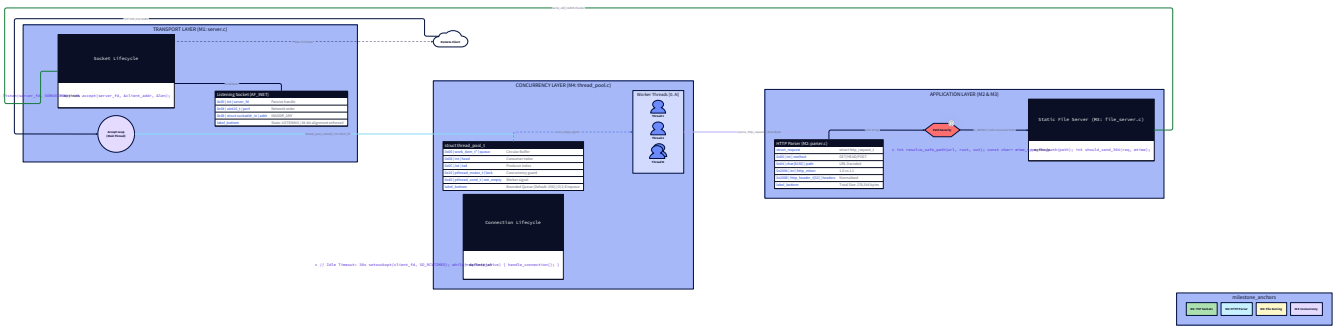
- [ ] `./http_server 8080 /tmp/www` starts, prints the canonical document root path
- [ ] `curl http://localhost:8080/` returns `200 OK` with `Content-Type: text/html; charset=utf-8` and the content of `index.html`
- [ ] `curl http://localhost:8080/style.css` returns `Content-Type: text/css`
- [ ] `curl http://localhost:8080/app.js` returns `Content-Type: application/javascript`

- [ ] `curl http://localhost:8080/images/logo.png` returns `Content-Type: image/png` and binary-intact PNG data
- [ ] All responses include a correctly formatted `Last-Modified` header (day-of-week, day, mon, year, time, "GMT")
- [ ] `curl -H "If-Modified-Since: <valid future date>" http://localhost:8080/index.html` returns `304 Not Modified` with no body
- [ ] `curl http://localhost:8080/missing.html` returns `404 Not Found` with an HTML error body
- [ ] `printf "GET /../../../etc/passwd HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc localhost 8080` returns `403 Forbidden`, never file contents
- [ ] `printf "GET /%2e%2e%2f%2e%2e%2fetc%2fpasswd HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc localhost 8080` returns `403 Forbidden`
- [ ] A symlink inside the document root pointing outside it returns `403 Forbidden`
- [ ] `curl http://localhost:8080/empty-dir/` (directory without `index.html`) returns `403 Forbidden`
- [ ] `curl -X HEAD http://localhost:8080/index.html` returns `200 OK` with correct `Content-Length` and zero bytes of body
- [ ] FD count after 1000 requests (mix of 200, 304, 404, 403) returns to baseline — no leaks

Milestone 4: Concurrent Connections

Where You Are in the System

You have built three-quarters of a real HTTP server. Your server binds a port, reads raw bytes off TCP connections, parses them into structured HTTP requests, and serves files with proper security validation. What you have right now is a fully correct sequential server — one that would serve a single developer testing a local site with zero problems. What you do not have yet is a server that could survive contact with reality.



Reality looks like this: a user opens a browser and loads a page with 20 assets (HTML, CSS, JavaScript files, images). The browser opens

multiple connections simultaneously to fetch those assets in parallel. A second user hits the site at the same moment. Your sequential accept loop — which finishes one connection completely before calling `accept()` again — makes everyone wait in line. The second user's browser times out before it gets a response. Your server, from the outside, appears broken. This milestone fixes that by introducing concurrency: the ability to handle multiple connections at the same time. You will build two models — thread-per-connection first (simple, reveals the problem) and then a bounded thread pool (the real solution) — and layer on top the three features that make concurrent connection management work in production: HTTP/1.1 keep-alive, per-connection idle timeouts, and graceful shutdown.

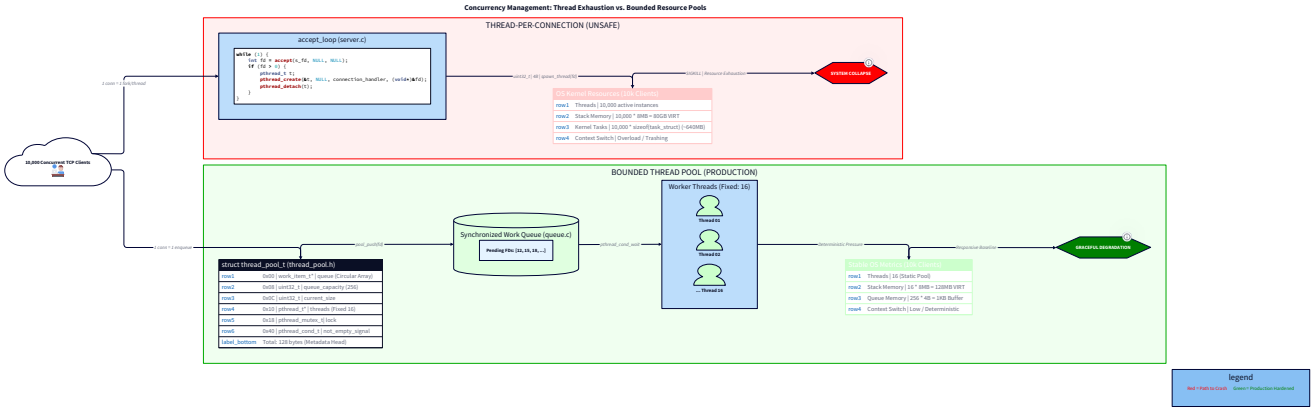
The Revelation: "Just Spawn a Thread" Is Not a Solution

Before writing any threading code, you need to understand why the obvious approach is dangerous. The obvious approach is: when a new connection arrives, spawn a thread to handle it. Done. Each client gets a thread. Threads can run concurrently. Problem solved. This works perfectly in testing. You connect with curl, it gets a thread, the thread handles the request, everything is fine. You connect with ten curls in parallel, ten threads handle them simultaneously. Still fine. Now an attacker runs this:

```bash
# Slowloris: open 10,000 connections and send HTTP headers one byte every 20 seconds
# Never complete the request — hold the connections open indefinitely
python3 slowloris.py --port 8080 --connections 10000 --sleep 20
```

The Slowloris attack, discovered in 2009 by Robert "RSnake" Hansen, exploits exactly the "just spawn a thread" model. Each connection gets a thread. The thread blocks in `read()` waiting for the complete HTTP headers. The attacker sends one byte every 20 seconds — just enough to keep the connection alive and prevent your timeout (if you have one). With 10,000 connections, you have 10,000 threads. Here is what 10,000 threads cost on Linux: Each thread, by default, gets an **8MB stack**. This is the kernel's default (`ulimit -s` reports it). The stack is virtual memory — it doesn't all map to physical RAM immediately — but the kernel still reserves the virtual address space. On a 64-bit system with 256GB of virtual address space, you might think this is fine. But:

- 10,000 threads × 8MB stack = **80GB of virtual address space** reserved
- Kernel overhead per thread: ~64KB for the task_struct and related kernel data structures = **640MB of kernel memory**
- Context switches between 10,000 threads: the scheduler now has 10,000 runnable (or blocked) threads to manage — scheduling overhead becomes visible
- The Linux default maximum number of threads (`/proc/sys/kernel/threads-max`) is often around 32,000. With 10,000 connections, you're already at 31% of the system limit. A real attack can push past it entirely, causing `pthread_create()` to fail with `EAGAIN` — your server can no longer handle even legitimate connections



The resource exhaustion is not hypothetical. It's the exact failure mode that took down Apache web servers in 2009 before patches were deployed. Apache's default model was thread-per-connection (actually process-per-connection, which is worse). Slowloris exploited the OS limit on concurrent processes, not the attack's bandwidth requirements. The attacking machine needed almost no resources; the defending server was overwhelmed. The insight: thread creation is not free, and thread count must be bounded. A bounded thread pool eliminates this entire attack surface. If your pool has 16 threads, the 17th connection waits in a queue. Queued connections consume no threads, no kernel task_structs, no 8MB stacks. The 10,000th Slowloris connection just adds to the queue (which you can also bound — more on that shortly). Your server continues serving legitimate requests normally. But there is a second revelation hiding inside the first. Even

with a bounded pool, keep-alive connections without idle timeouts create a slow leak. If your 16 threads are all handling keep-alive connections from browsers that opened them 10 minutes ago and haven't sent a second request, all 16 threads are blocked in `read()`. New legitimate connections sit in the queue forever. The pool is full of zombie connections — technically alive, sending no data, consuming a thread each. The complete solution requires three interlocking pieces: a bounded thread pool, per-connection idle timeouts, and a keep-alive loop that correctly reuses connections when appropriate and closes them when not.

POSIX Threads: The API You Will Use

[[EXPLAIN:pthread-api|POSIX Threads (pthreads) — creation, mutexes, and condition variables]] The pthreads API is POSIX's standardized interface for multi-threading in C. Linux implements it via the `<pthread.h>` header and the `libpthread` library (link with `-lpthread`). You need four concepts from it.

Thread creation: `pthread_create()`

```c
#include <pthread.h>

// Signature:
int pthread_create(
    pthread_t *thread,        // OUTPUT: handle to the new thread
    const pthread_attr_t *attr, // Thread attributes (NULL for defaults)
    void *(*start_fn)(void*), // Function the thread will run
    void *arg                 // Argument passed to start_fn
);

// Returns 0 on success, error code on failure (EAGAIN = too many threads, ENOMEM = out of memory)
```

The new thread starts executing `start_fn(arg)` immediately, concurrently with your calling thread. Both threads run in the same process, sharing the same heap memory, global variables, and file descriptors. That shared access is what makes synchronization necessary. **Mutexes: `pthread_mutex_t`** A mutex (**mut**ual

exclusion lock) is a synchronization primitive that ensures only one thread accesses a shared resource at a time. Imagine a bathroom key on a hook — only the person holding the key can enter; everyone else waits.

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  // Static initialization

pthread_mutex_lock(&lock);   // Acquire lock — blocks if another thread holds it

// ... access shared data here ...

pthread_mutex_unlock(&lock); // Release lock — wakes one waiting thread
```

Any code that reads or writes shared state (a counter, a log file, a work queue) must hold the mutex while doing so. Code that doesn't hold the mutex and accesses shared data has a **data race** — the result is undefined behavior in C, meaning anything can happen: corrupted data, crashes, or apparently correct behavior that breaks under load. **Condition variables:** `pthread_cond_t` A condition variable lets a thread sleep until some condition becomes true. Used together with a mutex, it allows "wait until there is work to do" — essential for thread pool worker loops.

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// In a producer thread: signal that work is available

pthread_mutex_lock(&lock);

// ... add work to shared queue ...

pthread_cond_signal(&cond);  // Wake one sleeping consumer

pthread_mutex_unlock(&lock);

// In a consumer thread: wait for work

pthread_mutex_lock(&lock);

while (queue_is_empty(&queue)) {

    pthread_cond_wait(&cond, &lock);

    // pthread_cond_wait atomically: releases lock AND sleeps

    // When signaled: reacquires lock AND returns

    // The while loop (not if) handles spurious wakeups

}

// ... take work from queue ...

pthread_mutex_unlock(&lock);
```

The `while` loop around `pthread_cond_wait()` is not optional ceremony. POSIX permits **spurious wakeups** — a thread can wake from `pthread_cond_wait()` even without being signaled. Always re-check the condition after waking. **Thread joining and detaching**

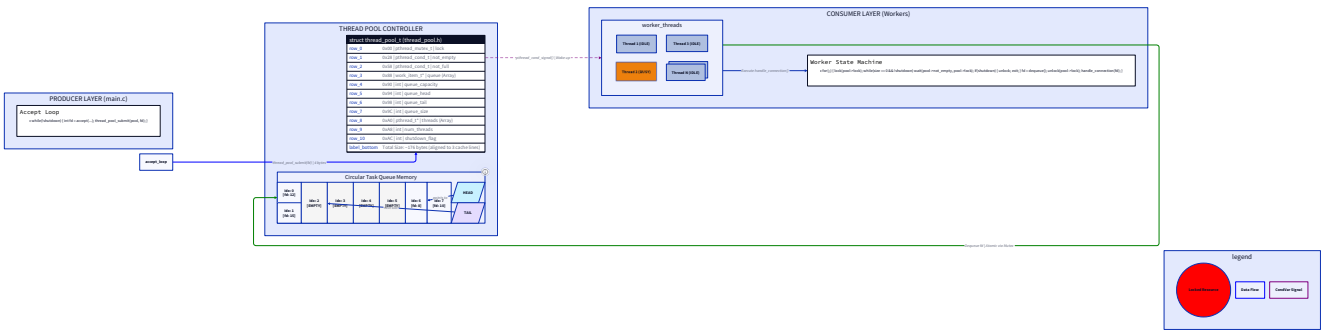
```c
pthread_join(thread, NULL);   // Block until thread exits, then clean up its resources

pthread_detach(thread);       // Thread cleans up its own resources when it exits
```

You must either join or detach every thread. A thread that is neither joined nor detached is a resource leak — its kernel task_struct remains allocated until the process exits.

The Thread Pool Architecture

Now build the actual solution. A thread pool consists of:

1. **A fixed array of worker threads** — created at startup, they run a loop waiting for work
2. **A shared work queue** — a bounded queue of file descriptors waiting to be handled
3. **A mutex protecting the queue** — ensures threads don't corrupt the queue by accessing it simultaneously
4. **A condition variable** — lets workers sleep efficiently when the queue is empty
5. **A shutdown flag** — signals workers to exit cleanly



```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define POOL_SIZE_DEFAULT    16
#define QUEUE_SIZE_DEFAULT   256
#define IDLE_TIMEOUT_SECS    30

// A single item in the work queue: a client file descriptor to serve
typedef struct {
    int client_fd;
} work_item_t;

// The thread pool and its shared state
typedef struct {
    // --- Work queue (circular buffer) ---
    work_item_t  *queue;          // Fixed-size circular buffer of work items
    int           queue_capacity; // Maximum items in queue
    int           queue_head;     // Index of next item to consume
    int           queue_tail;     // Index of next slot to produce into
    int           queue_size;     // Current number of items in queue

    // --- Synchronization ---
    pthread_mutex_t lock;         // Protects all queue fields and shutdown flag
```

```
    pthread_cond_t  not_empty;    // Signaled when an item is added to empty queue
    pthread_cond_t  not_full;     // Signaled when an item is removed from full queue
    // --- Worker threads ---
    pthread_t    *threads;        // Array of worker thread handles
    int           num_threads;    // Number of worker threads
    // --- Lifecycle ---
    int           shutdown;       // 1 = workers should exit after draining queue
    // --- Shared statistics (protected by lock) ---
    int           active_connections;  // Number of connections currently being served
    long          total_requests;      // Cumulative request count for access log
    // --- Server configuration (read-only after init, no lock needed) ---
    const char   *document_root;  // Canonical document root path
    int           idle_timeout;   // Per-connection idle timeout in seconds
} thread_pool_t;
```

Memory Layout

The work queue is a **circular buffer** — a fixed-size array where the head and tail pointers wrap around. When the tail reaches the end, it wraps back to index 0. This gives O(1) enqueue and dequeue without any allocation.

```
Queue capacity = 8, current state: head=2, tail=5, size=3
Index: 0     1     2     3     4     5     6     7
       [  ] [  ] [fd=9][fd=12][fd=15][    ][    ][  ]
                  ↑head               ↑tail
Items waiting: fd=9, fd=12, fd=15
```

When `queue_size == queue_capacity`, the queue is full. A new connection arriving while the queue is full gets rejected with 503.

Initialization

```c
int thread_pool_init(thread_pool_t *pool, int num_threads, int queue_capacity,
                     const char *document_root, int idle_timeout) {
    memset(pool, 0, sizeof(*pool));
    pool->queue = calloc((size_t)queue_capacity, sizeof(work_item_t));
    if (!pool->queue) return -1;
    pool->queue_capacity = queue_capacity;
    pool->num_threads    = num_threads;
    pool->document_root  = document_root;
    pool->idle_timeout   = idle_timeout;
    pthread_mutex_init(&pool->lock, NULL);
    pthread_cond_init(&pool->not_empty, NULL);
    pthread_cond_init(&pool->not_full, NULL);
    pool->threads = calloc((size_t)num_threads, sizeof(pthread_t));
    if (!pool->threads) { free(pool->queue); return -1; }
    for (int i = 0; i < num_threads; i++) {
        int rc = pthread_create(&pool->threads[i], NULL, worker_thread_fn, pool);
        if (rc != 0) {
            // Failed to create thread — initiate shutdown of already-created threads
            pool->shutdown = 1;
            pthread_cond_broadcast(&pool->not_empty);
            for (int j = 0; j < i; j++) pthread_join(pool->threads[j], NULL);
            free(pool->threads);
            free(pool->queue);
            return -1;
        }
    }
```

```
    return 0;

}
```

`pthread_cond_broadcast()` wakes *all* sleeping threads (unlike `signal()` which wakes one). Used during shutdown to ensure all workers wake up and check the shutdown flag.

The Worker Thread Loop

```c
static void *worker_thread_fn(void *arg) {

    thread_pool_t *pool = (thread_pool_t *)arg;

    for (;;) {

        // --- Wait for work ---

        pthread_mutex_lock(&pool->lock);

        while (pool->queue_size == 0 && !pool->shutdown) {

            pthread_cond_wait(&pool->not_empty, &pool->lock);

        }

        if (pool->shutdown && pool->queue_size == 0) {

            // Shutdown: no work left — this worker exits

            pthread_mutex_unlock(&pool->lock);

            return NULL;

        }

        // --- Dequeue one item ---

        work_item_t item = pool->queue[pool->queue_head];

        pool->queue_head = (pool->queue_head + 1) % pool->queue_capacity;

        pool->queue_size--;

        pool->active_connections++;

        // Signal that queue has space (for the accept loop, if it was blocked)

        pthread_cond_signal(&pool->not_full);

        pthread_mutex_unlock(&pool->lock);

        // --- Handle the connection (OUTSIDE the lock) ---

        handle_connection(item.client_fd, pool);

        // --- Update stats ---

        pthread_mutex_lock(&pool->lock);

        pool->active_connections--;
```

```
        pthread_mutex_unlock(&pool->lock);

    }

}
```

The critical discipline: **hold the lock only while touching shared data, never while doing I/O or long computation.** The actual connection handling — which involves network reads, file opens, and potentially seconds of waiting — happens completely outside the mutex. If you held the mutex during connection handling, all other threads would be blocked waiting for it. You'd have effectively serialized your "concurrent" server.

Enqueue: Submitting Work

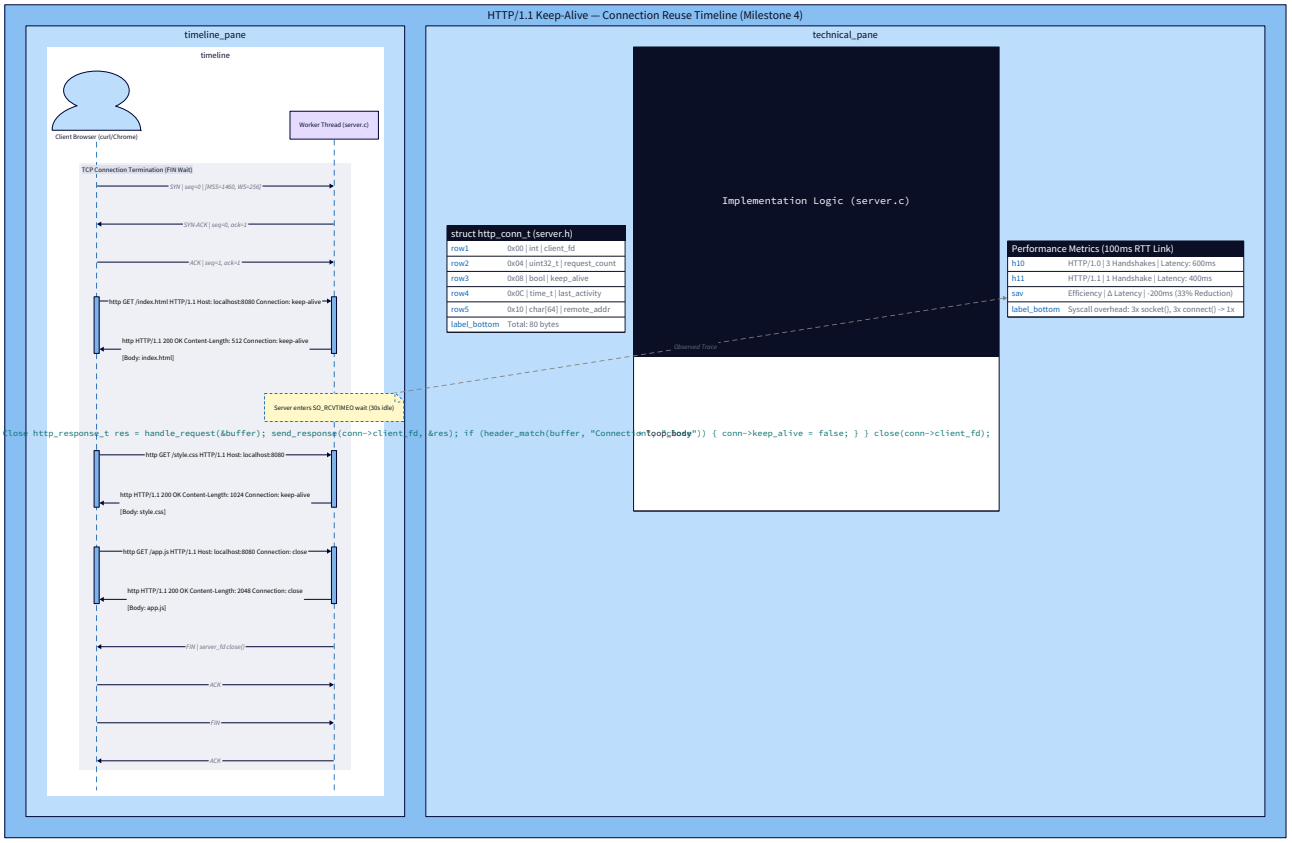
```c
// Submit a client connection to the thread pool for handling.
// Returns 0 on success, -1 if pool is full (caller should send 503 and close fd).
int thread_pool_submit(thread_pool_t *pool, int client_fd) {
    pthread_mutex_lock(&pool->lock);
    if (pool->shutdown) {
        pthread_mutex_unlock(&pool->lock);
        return -1;  // Server is shutting down
    }
    if (pool->queue_size >= pool->queue_capacity) {
        // Queue is full — reject this connection
        pthread_mutex_unlock(&pool->lock);
        return -1;
    }
    pool->queue[pool->queue_tail] = (work_item_t){ .client_fd = client_fd };
    pool->queue_tail = (pool->queue_tail + 1) % pool->queue_capacity;
    pool->queue_size++;
    // Wake one sleeping worker
    pthread_cond_signal(&pool->not_empty);
    pthread_mutex_unlock(&pool->lock);
    return 0;
}
```

When `thread_pool_submit()` returns -1, the accept loop must send a `503 Service Unavailable` response and close the file descriptor. This is the graceful overload response: tell the client the server is temporarily busy, rather than silently dropping the connection or crashing.

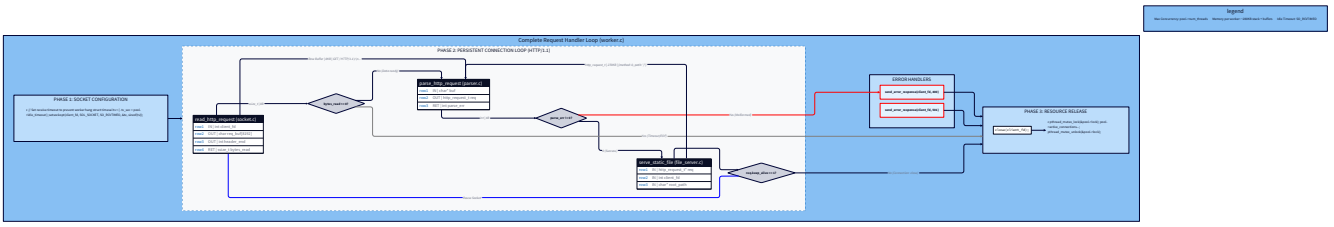
```c
// Send a minimal 503 response — used when the pool is full.
// We send the smallest valid HTTP response to minimize time holding the fd.
static void send_503(int client_fd) {
    const char *resp =
        "HTTP/1.1 503 Service Unavailable\r\n"
        "Content-Type: text/html\r\n"
        "Content-Length: 63\r\n"
        "Connection: close\r\n"
        "\r\n"
        "<html><body><h1>503 Service Unavailable</h1></body></html>";
    // Best-effort write — if client already disconnected, EPIPE is ignored
    write(client_fd, resp, strlen(resp));
    close(client_fd);
}
```

HTTP/1.1 Keep-Alive: The Connection Reuse Loop

HTTP/1.0 opened a new TCP connection for every single request/response cycle — a three-way handshake for each asset, before sending anything. For a page with 50 assets, that's 50 TCP handshakes. On a cross-continental connection with 100ms RTT, that's 5 seconds of pure handshake overhead before any real data transfers. HTTP/1.1 introduced **persistent connections** (keep-alive): by default, the connection stays open after a response, and the client sends the next request on the same socket. The overhead of TCP connection establishment — one handshake for the entire session — is paid once.



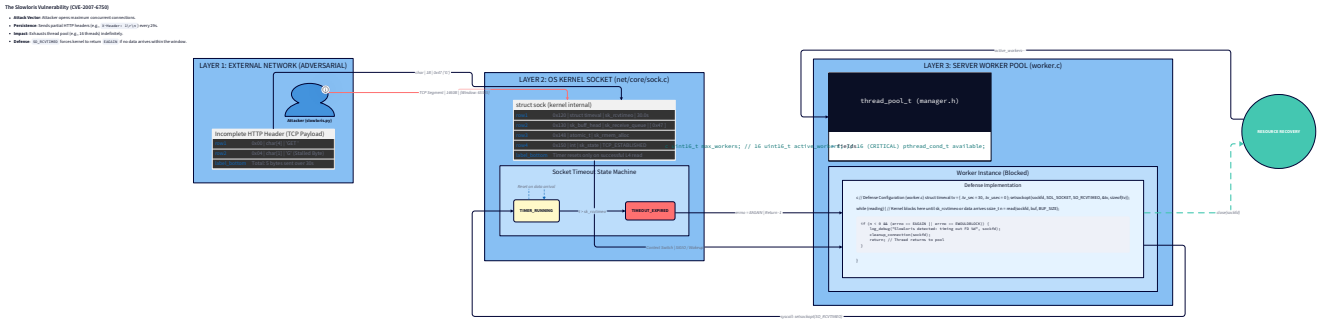
From your Milestone 2 parser, you already track `req.keep_alive` and set the `Connection: keep-alive` or `Connection: close` response header. Now you need to make the connection actually persist when keep-alive is requested. The keep-alive loop wraps the request-read → parse → serve cycle inside a `while` loop on the same `client_fd`. Here is the complete connection handler that replaces the simple sequential handler:



```c
void handle_connection(int client_fd, thread_pool_t *pool) {
    // Set the receive timeout on the socket.
    // If no data arrives within idle_timeout seconds, read() returns EAGAIN/EWOULDBLOCK.
    struct timeval tv;
    tv.tv_sec  = pool->idle_timeout;
    tv.tv_usec = 0;
    setsockopt(client_fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
    // Keep-alive loop: serve multiple requests on one connection
    for (;;) {
        // --- Phase 1: Read the request ---
        char req_buf[REQUEST_BUF_SIZE];
        int header_end = -1;
        ssize_t bytes_read = read_http_request(client_fd, req_buf,
                                               REQUEST_BUF_SIZE, &header_end);
        if (bytes_read < 0 || header_end < 0) {
            // Timeout (EAGAIN), client disconnect (0 bytes), or read error
            // In all cases: close the connection, return thread to pool
            break;
        }
        // --- Phase 2: Parse ---
        http_request_t req;
        int parse_err = parse_http_request(req_buf, (size_t)bytes_read,
                                           header_end, &req);
        if (parse_err != 0) {
            send_error_by_code(client_fd, parse_err);
            break;  // Close after error response
        }
```

```
        // --- Phase 3: Update shared stats (briefly hold lock) ---
        pthread_mutex_lock(&pool->lock);
        pool->total_requests++;
        long req_num = pool->total_requests;
        pthread_mutex_unlock(&pool->lock);
        // --- Phase 4: Serve the request ---
        if (req.method == METHOD_GET || req.method == METHOD_HEAD) {
            serve_static_file(&req, client_fd, pool->document_root);
        } else {
            send_error_response(client_fd, 501, "Not Implemented",
                "<html><body><h1>501 Not Implemented</h1></body></html>");
            break;  // Don't keep-alive after 501
        }
        // --- Phase 5: Decide whether to continue ---
        if (!req.keep_alive) {
            break;  // Connection: close — done
        }
        // Otherwise: loop back, wait for the next request on this connection
    }
    close(client_fd);
}
```

The `SO_RCVTIMEO` socket option is the idle timeout mechanism. When you set it, `recv()` and `read()` will return -1 with `errno = EAGAIN` (or `EWOULDBLOCK`) if no data arrives within the specified time. Your `read_http_request()` function from Milestone 1 already returns -1 on any error, so the timeout causes it to return -1, breaking the keep-alive loop and closing the connection. The worker thread is now free to handle a new connection.

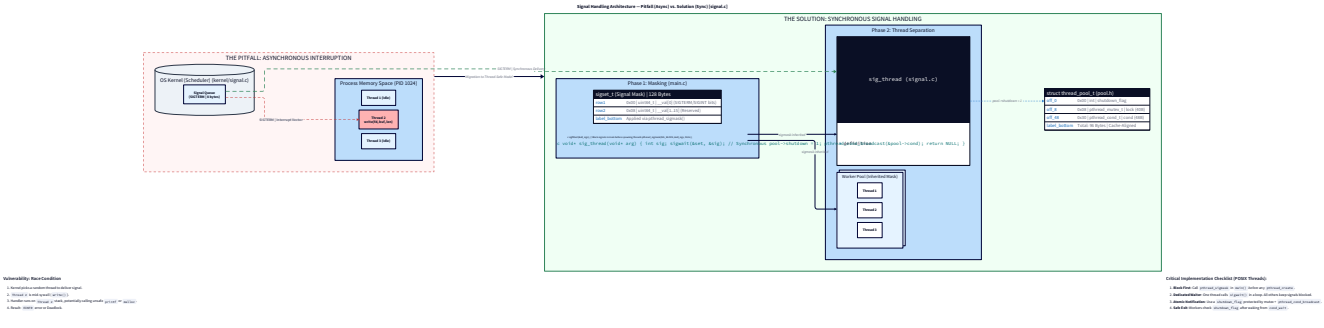


Why Break After Non-200 Responses?

Notice the `break` after sending a 501 response. This is deliberate. If a client sends a malformed or unsupported request and receives a 4xx or 5xx response, the right behavior is usually to close the connection. The client clearly has a problem — letting it keep the connection alive and send more broken requests wastes server resources and potentially enables abuse. The exceptions are 404 and 304, where the client is well-behaved but just requesting something that doesn't exist or hasn't changed. For those, keep-alive is fine.

Signal Handling in Multi-Threaded Programs

When your server was single-threaded, signal handling was straightforward: `signal(SIGPIPE, SIG_IGN)` at startup, and you're done. Multi-threaded programs have a complication: **POSIX delivers signals to an unpredictable thread.**



The POSIX specification says: when a signal is delivered to a process (as opposed to a specific thread via `pthread_kill()`), the kernel picks one of the threads that doesn't have the signal blocked. It may pick any thread — the accept loop thread, a worker thread, any of them. This creates a subtle problem for SIGTERM

and SIGINT (your graceful shutdown signals). If a worker thread receives SIGTERM, it might be in the middle of writing a file to the network. The signal handler runs asynchronously. If your signal handler calls `shutdown_server()` which modifies shared state while the worker thread isn't holding any lock, you have a data race inside the signal handler. The safe solution for multi-threaded programs is the **dedicated signal thread** pattern:

1. In `main()`, before creating any threads, **block all signals** using `pthread_sigmask()`
2. Create the thread pool — worker threads inherit the blocked signal mask
3. Create a dedicated signal thread that uses `sigwait()` to wait for signals synchronously

```c
// Block all signals in the main thread before spawning workers.

// All threads created after this inherit the blocked mask.

sigset_t all_signals;

sigfillset(&all_signals);

pthread_sigmask(SIG_BLOCK, &all_signals, NULL);

// Create the thread pool (workers inherit blocked signal mask)

thread_pool_init(&pool, POOL_SIZE_DEFAULT, QUEUE_SIZE_DEFAULT, root, IDLE_TIMEOUT_SECS);

// Create the dedicated signal-handling thread

pthread_t signal_thread;

pthread_create(&signal_thread, NULL, signal_handler_thread, &pool);

// Now run the accept loop in main thread

accept_loop(&server_fd, &pool);
```

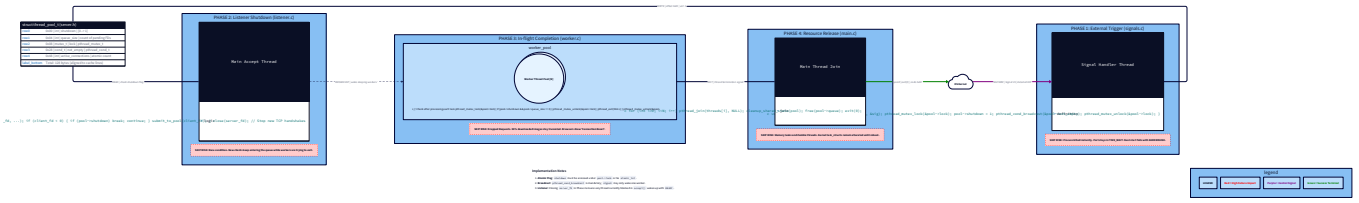
The signal-handling thread uses `sigwait()` — a blocking call that waits for a signal from the specified set and returns it synchronously, without running any asynchronous signal handler:

```c
static void *signal_handler_thread(void *arg) {
    thread_pool_t *pool = (thread_pool_t *)arg;
    // Wait for termination signals
    sigset_t waitset;
    sigemptyset(&waitset);
    sigaddset(&waitset, SIGTERM);
    sigaddset(&waitset, SIGINT);
    // Still ignore SIGPIPE globally to avoid broken-pipe kills
    signal(SIGPIPE, SIG_IGN);
    int sig;
    sigwait(&waitset, &sig);  // Blocks until SIGTERM or SIGINT arrives
    printf("\n[%s received] Initiating graceful shutdown...\n",
           sig == SIGTERM ? "SIGTERM" : "SIGINT");
    // Initiate graceful shutdown
    thread_pool_shutdown(pool);
    return NULL;
}
```

`sigwait()` is safe to call from a regular thread context — no async-signal-safety concerns, no restriction on what you can do before or after it. The signal is consumed by `sigwait()` and delivered synchronously to this thread. No other thread is interrupted.

Graceful Shutdown: The Four-Phase Dance

Graceful shutdown means: stop accepting new work, finish all in-flight work, then exit cleanly. It is not `kill -9` and it is not `exit(0)` in the signal handler.



The four phases: **Phase 1: Stop accepting new connections.** The accept loop must notice the shutdown signal and stop calling `accept()`. Use an `atomic_int` or mutex-protected flag that the accept loop checks between iterations. **Phase 2: Close the listening socket.** This causes any pending `accept()` call (if the accept loop is sleeping in it) to return with `EBADF`. New clients attempting to connect now get "connection refused" — a clear signal that the server is down, not hanging. **Phase 3: Set the pool shutdown flag and wake all workers.** Workers that are sleeping in `pthread_cond_wait()` need to wake up and check the shutdown flag. **Phase 4: Join all workers.** Wait for every worker thread to finish its current connection and exit. Only after all workers have joined can you safely free resources and call `exit(0)`.

```c
// Global flag for the accept loop to check — written by signal thread,
// read by accept loop. Using volatile int is NOT sufficient for thread safety
// but using atomic_int or a mutex-protected int is.
// We use a mutex-protected int to keep the pattern consistent with the pool.
void thread_pool_shutdown(thread_pool_t *pool) {
    pthread_mutex_lock(&pool->lock);
    pool->shutdown = 1;
    pthread_cond_broadcast(&pool->not_empty);  // Wake all sleeping workers
    pthread_mutex_unlock(&pool->lock);
    // Workers will drain the queue then exit when they see shutdown=1 and queue is empty
}
void thread_pool_join(thread_pool_t *pool) {
    for (int i = 0; i < pool->num_threads; i++) {
        pthread_join(pool->threads[i], NULL);
    }
    printf("All worker threads exited cleanly.\n");
    free(pool->threads);
    free(pool->queue);
    pthread_mutex_destroy(&pool->lock);
    pthread_cond_destroy(&pool->not_empty);
    pthread_cond_destroy(&pool->not_full);
}
```

The Accept Loop with Shutdown Awareness

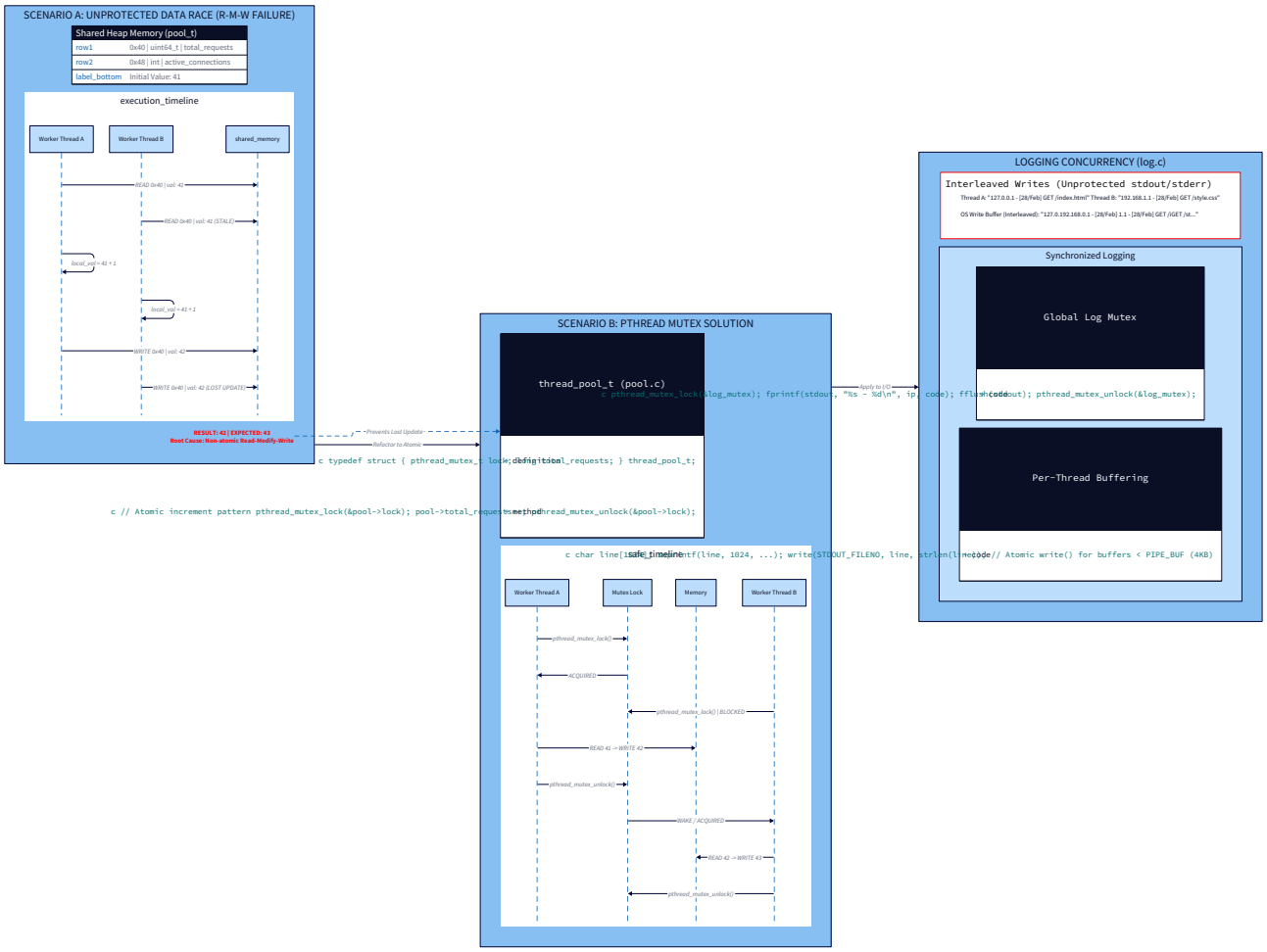
```c
// A global server_fd — needed by signal handler to close the listening socket
static volatile int g_server_fd = -1;

void accept_loop(int *server_fd_ptr, thread_pool_t *pool) {
    g_server_fd = *server_fd_ptr;
    for (;;) {
        // Check shutdown flag BEFORE calling accept
        pthread_mutex_lock(&pool->lock);
        int shutting_down = pool->shutdown;
        pthread_mutex_unlock(&pool->lock);
        if (shutting_down) break;
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        int client_fd = accept(*server_fd_ptr,
                               (struct sockaddr*)&client_addr, &client_len);
        if (client_fd < 0) {
            if (errno == EINTR || errno == EBADF) break;  // Signal or fd closed
            if (errno == EMFILE || errno == ENFILE) {
                // Out of file descriptors — log and briefly yield
                fprintf(stderr, "accept: out of file descriptors!\n");
                usleep(100000);  // 100ms back-off
                continue;
            }
            perror("accept");
            continue;
        }
        // Try to submit to pool
```

```
        if (thread_pool_submit(pool, client_fd) < 0) {
            send_503(client_fd);  // Pool full or shutting down
        }
    }
    close(*server_fd_ptr);
    *server_fd_ptr = -1;
    g_server_fd = -1;
    printf("Accept loop exited.\n");
}
```

The `EMFILE` case — `errno = EMFILE` means "too many open files" (the process hit its file descriptor limit) — is important and often missed. When this happens, `accept()` fails, but the OS has already completed the TCP handshake for the waiting client. Repeatedly calling `accept()` to get this error doesn't drain the accept queue — the client remains waiting. The `usleep(100000)` back-off gives the server a moment to close other file descriptors (connections finishing in the worker pool) before trying again.

Shared State and Mutex Protection

Your server now has state shared between multiple threads: the request counter, the connection counter, and any shared output (like an access log). Every shared write creates a potential data race.



A **data race** in C is not just "sometimes wrong results." The C11 standard (and the POSIX threading specification) defines any concurrent access to the same memory location where at least one access is a write, without synchronization, as **undefined behavior** — the most severe category in C. The compiler is permitted to assume data races don't exist, which means it can reorder instructions, cache values in registers, and produce behavior that makes no sense from a sequential reasoning perspective. The most common manifestation: a 64-bit counter incremented by two threads. Even though the increment seems atomic, it compiles to `load → add → store`. Two threads can both load the old value, both add 1, and both store back — net result: the counter only incremented by 1 instead of 2. With the compiler free to reorder, the actual behavior can be even stranger. Here is the shared state in the pool struct, with protection analysis:

```c
// Things protected by pool->lock:
//   pool->queue_*         — modified by accept loop and workers
//   pool->shutdown        — written by signal thread, read by workers and accept loop
//   pool->active_connections — incremented/decremented by workers
//   pool->total_requests  — incremented by workers
// Things that DO NOT need the lock (read-only after initialization):
//   pool->document_root   — set once at startup, never modified
//   pool->idle_timeout    — set once at startup, never modified
//   pool->num_threads     — set once at startup, never modified
//   pool->threads[]       — array set at startup; individual elements not modified
```

Accessing `pool->document_root` from multiple worker threads without a lock is safe because it's read-only: the workers only read it, they never write it. The mutex only needs to protect data that is both written and read concurrently. For the access log (writing to a file or stdout), mutex protection is essential:

```c
static pthread_mutex_t log_mutex = PTHREAD_MUTEX_INITIALIZER;

void log_request(const char *client_ip, const http_request_t *req,
                 int status_code, long bytes_sent) {
    time_t now = time(NULL);
    struct tm *gmt = gmtime(&now);
    char time_buf[64];
    strftime(time_buf, sizeof(time_buf), "%d/%b/%Y:%H:%M:%S +0000", gmt);
    // Without this lock, output from two threads interleaves in the same line
    pthread_mutex_lock(&log_mutex);
    printf("%s - - [%s] \"%s %s HTTP/1.%d\" %d %ld\n",
           client_ip, time_buf,
           method_name(req->method), req->path, req->http_minor,
           status_code, bytes_sent);
    fflush(stdout);
    pthread_mutex_unlock(&log_mutex);
}
```

Without the `log_mutex`, two threads writing to stdout simultaneously produce interleaved garbage — half of thread A's line, half of thread B's, all on the same output line. This looks harmless but signals sloppy concurrency discipline. If the log is wrong, you can't trust your own server's behavior reports.

Pool Sizing: The Math That Governs Everything

The pool size is not a guess. For an I/O-bound workload like file serving, there is a formula from **Little's Law** and **Amdahl's Law** that gives you a principled starting point. For a CPU-bound workload (cryptography, compression, rendering), the optimal thread count is:

```
pool_size = num_cpu_cores
```

Adding more threads than cores causes context-switch overhead without improving throughput — the CPU is already fully utilized. For an I/O-bound workload (file serving, database queries), threads spend most of their time waiting. The formula is:

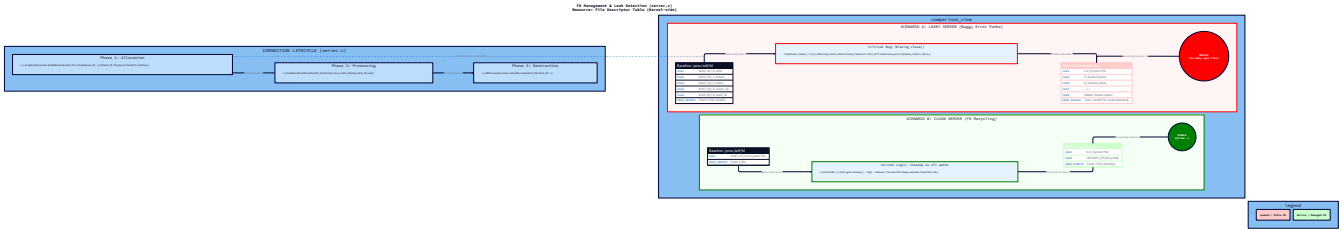
```
pool_size = num_cores × (1 + wait_time / compute_time)
```

Where `wait_time` is the time a request spends blocked on I/O (file reads from disk, network reads from client) and `compute_time` is the time spent actually executing CPU instructions (parsing, path resolution, MIME lookup). For a static file server serving files from an NVMe SSD:

- Compute time per request: ~10–50µs (mostly parsing and path operations)
- Wait time per request: ~100µs for an SSD read, or ~5ms for a spinning disk, or ~0µs for page-cached files For page-cached files (common for popular content):
- `pool_size = num_cores × (1 + ~1µs / ~30µs) ≈ num_cores × 1.03 ≈ num_cores` For SSD-served cold files:
- `pool_size = num_cores × (1 + 100µs / 30µs) ≈ num_cores × 4.3` The default of 16 is a reasonable production starting point for a 4-core machine serving SSD-backed files with some cache warming. Monitor your active connection count and adjust: if `active_connections` is consistently near `pool_size` under load, increase the pool.

FD Leak Detection Under Load

The acceptance criterion requires that after 10,000 sequential connections, the open FD count returns to baseline. This check is your comprehensive correctness test for all four milestones combined.



FD leaks in a concurrent server are more insidious than in a sequential one because the leaky code path might only execute under specific race conditions:

- A worker thread gets a client_fd, starts serving, the client disconnects mid-response → the error path exits without `close(client_fd)`
- A `pthread_create()` fails while client_fd is already assigned → cleanup path skips the close

- A keep-alive loop exits due to timeout but closes only once when it should be fine (this is OK — one close per fd) The test script:

```bash
#!/bin/bash

SERVER_PID=$(pgrep http_server)

ROOT_PATH=/tmp/www

echo "Starting FD baseline..."

BASE_FDS=$(ls /proc/$SERVER_PID/fd 2>/dev/null | wc -l)

echo "Baseline: $BASE_FDS open FDs"

echo "Sending 10,000 connections..."

for i in $(seq 10000); do

    curl -s http://localhost:8080/ > /dev/null

done

# Give server time to close any in-flight FDs

sleep 2

FINAL_FDS=$(ls /proc/$SERVER_PID/fd 2>/dev/null | wc -l)

echo "Final: $FINAL_FDS open FDs"

if [ "$FINAL_FDS" -le "$((BASE_FDS + 2))" ]; then

    echo "PASS: No FD leak detected (within tolerance of 2)"

else

    echo "FAIL: Leaked $((FINAL_FDS - BASE_FDS)) FDs"

    echo "Current open FDs:"

    ls -la /proc/$SERVER_PID/fd/

fi
```

A tolerance of 2 accounts for FDs that might be mid-close at the moment you sample — perfectly normal timing variation. Growth of 3+ is a real leak. On Linux, you can identify what types of FDs are leaking:

```bash
# Show the types of all open FDs for your server

ls -la /proc/$(pgrep http_server)/fd/

# symlinks point to: socket:[inode] (sockets), /path/to/file (files), pipe:[inode] (pipes)

# If you see many socket:[...] entries that should have been closed, you have a socket leak
```

Complete Main: Wiring It All Together

Here is the complete `main()` that integrates all four milestones:

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <signal.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#define DEFAULT_PORT       8080

#define DEFAULT_POOL_SIZE  16

#define DEFAULT_QUEUE_SIZE 256

#define DEFAULT_TIMEOUT    30

#define MAX_PATH_SIZE      4096

extern void accept_loop(int *server_fd, thread_pool_t *pool);

extern int  thread_pool_init(thread_pool_t*, int, int, const char*, int);

extern void thread_pool_join(thread_pool_t*);

static thread_pool_t g_pool;

int main(int argc, char *argv[]) {

    int port = DEFAULT_PORT;

    const char *doc_root = ".";  // Default: current directory

    if (argc >= 2) port      = atoi(argv[1]);

    if (argc >= 3) doc_root  = argv[2];

    // --- Canonicalize document root (once, at startup) ---

    char canonical_root[MAX_PATH_SIZE];

    if (realpath(doc_root, canonical_root) == NULL) {
```

```
        perror("realpath (document root)");
        return 1;
    }
    printf("Document root: %s\n", canonical_root);
    // --- Block all signals before creating any threads ---
    // Workers inherit this blocked mask, so signals go to the signal thread only
    sigset_t all_sigs;
    sigfillset(&all_sigs);
    pthread_sigmask(SIG_BLOCK, &all_sigs, NULL);
    // SIGPIPE: ignore globally (must be done via sigaction, not masked)
    struct sigaction sa_ignore = { .sa_handler = SIG_IGN };
    sigaction(SIGPIPE, &sa_ignore, NULL);
    // --- Initialize thread pool ---
    if (thread_pool_init(&g_pool, DEFAULT_POOL_SIZE, DEFAULT_QUEUE_SIZE,
                         canonical_root, DEFAULT_TIMEOUT) < 0) {
        fprintf(stderr, "Failed to initialize thread pool\n");
        return 1;
    }
    printf("Thread pool: %d workers, queue capacity %d\n",
           DEFAULT_POOL_SIZE, DEFAULT_QUEUE_SIZE);
    // --- Create signal handler thread ---
    pthread_t sig_thread;
    pthread_create(&sig_thread, NULL, signal_handler_thread, &g_pool);
    pthread_detach(sig_thread);  // Will exit after handling one signal
    // --- Create the listening socket ---
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) { perror("socket"); return 1; }
```

```c
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family      = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port        = htons((uint16_t)port);
    if (bind(server_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("bind"); close(server_fd); return 1;
    }
    if (listen(server_fd, SOMAXCONN) < 0) {
        perror("listen"); close(server_fd); return 1;
    }
    printf("Server listening on port %d\n", port);
    // --- Run the accept loop (blocks until shutdown) ---
    accept_loop(&server_fd, &g_pool);
    // --- Wait for all workers to finish ---
    thread_pool_join(&g_pool);
    printf("Server shutdown complete.\n");
    return 0;
}
```

Compiling the Multi-Threaded Server

```bash
# -lpthread links the POSIX threads library

# -pthread is preferred on modern GCC: sets linker flags AND compiler flags

# (-D_REENTRANT makes some library functions thread-safe)

gcc -Wall -Wextra -pthread -O2 \

    -o http_server \

    main.c socket.c parser.c file_server.c thread_pool.c

# Test graceful shutdown

./http_server 8080 /tmp/www &

SERVER_PID=$!

sleep 1

# Send 20 concurrent requests

for i in $(seq 20); do curl -s http://localhost:8080/ & done

# Immediately send SIGTERM — server should finish in-flight requests then exit

kill -SIGTERM $SERVER_PID

wait $SERVER_PID

echo "Exit code: $?"  # Should be 0
```

Three-Level View: A Request Under Concurrency

Take a GET request that arrives while 5 other requests are in flight and trace it through all three levels. Level 1 — Your code: The main thread's `accept_loop()` calls `accept()` and gets `client_fd = 12`. It calls `thread_pool_submit(&pool, 12)`, which acquires the pool mutex, adds fd 12 to the circular queue, increments `queue_size`, and signals `not_empty`. Worker thread 3 (which was sleeping in `pthread_cond_wait()`) wakes up, dequeues

`client_fd = 12`, increments `active_connections`, releases the lock, and calls `handle_connection(12, &pool)`. Meanwhile, the main thread is already back in `accept()`, accepting the next connection. Level 2 — OS/Kernel: `pthread_cond_signal()` doesn't run the sleeping thread immediately — it marks thread 3 as runnable and adds it to the kernel scheduler's run queue. The kernel's scheduler will run it when it gets a CPU timeslice (typically within ~1ms on a lightly loaded system, or immediately if a CPU is idle). The two threads — main thread in `accept()` and worker 3 in `handle_connection()` — run on separate CPU cores simultaneously: true parallelism on a multi-core machine. When worker 3 calls `read(client_fd, ...)`, the kernel checks the socket's receive buffer. If the HTTP request data is already there (common for localhost), it copies it to user space immediately. If not, the kernel moves thread 3 to the "waiting for I/O" state and puts another runnable thread on that CPU core. Zero CPU cycles wasted while waiting for network data. Level 3 — Hardware: `pthread_mutex_lock()` compiles to a `LOCK CMPXCHG` instruction (on x86) or `LDXR`/`STXR` (on ARM). These are atomic compare-and-swap instructions — hardware-enforced operations that compare and modify memory in a single, uninterruptible cycle. The CPU's cache coherence protocol (MESI) ensures that all cores see the same value for the mutex: when core 1 acquires the mutex (sets it from 0 to 1 with `CMPXCHG`), the cache coherence protocol invalidates the L1/L2 cache lines holding the mutex on all other cores, forcing them to reload from L3 or main memory. This cross-core coordination costs ~40–100 clock cycles (20–50ns at 2GHz) — the true cost of one mutex lock/unlock cycle. This is why the law "hold the lock only as long as necessary" matters at the hardware level: every cycle you hold the mutex is a cycle another core spends stalled waiting for the cache line.

Hardware Soul: Concurrency Costs

Thread creation overhead. `pthread_create()` involves a system call (`clone()` on Linux) that creates a new kernel task_struct, allocates a kernel stack (8KB), and maps the user-space stack (8MB virtual, but initially only one 4KB page is actually mapped — the rest page-faults in as needed). Total cost: ~10–50µs. The thread pool amortizes this by creating threads at startup, not per-request. Context switch cost. When the scheduler switches between threads — which happens every ~1–4ms on a standard Linux configuration (`CONFIG_HZ = 250` or `1000`) — it must save all CPU registers (16 general-purpose registers, SSE/AVX state, etc.) for the outgoing thread and restore them for the incoming thread. On x86-64, this is ~80 cache lines of state. Cost: ~1–5µs per context switch. With 16 threads, context switching is infrequent and well within budget. With 1000 threads, it becomes the dominant cost. False sharing. The `pool->active_connections` and `pool->total_requests` counters are adjacent in memory, likely sharing a 64-byte cache line. When worker thread A increments `active_connections` and worker thread B increments `total_requests` simultaneously — both cache lines get invalidated on both cores even though they're touching different variables. This false sharing causes unnecessary cache coherence traffic. Production thread pools (Go's runtime, jemalloc's arena structure) use `__attribute__((aligned(64)))` or padding to place each frequently-written field on its own cache line. For this server at 16 threads, it's not a measurable issue — but understanding it prepares you for high-performance work. Socket option `SO_RCVTIMEO` — the implementation. Setting `SO_RCVTIMEO` stores a `struct timeval` in the socket's kernel data structure. When `read()` is called on the socket, the kernel's net/socket

layer starts a timer. If data doesn't arrive within the timeout, the timer fires (via the kernel's timer interrupt) and wakes the blocked thread with `EAGAIN`. The cost is zero unless the timeout actually fires — it's a passive kernel structure, not a polling loop.

Design Decisions: Concurrency Models

Model	Max Connections	Complexity	Latency	Used By
Sequential (Milestone 1)	1	Minimal	Serialized	Dev tools
Thread-per-connection	OS thread limit (~32K)	Low	Low	Apache (legacy)
Bounded thread pool (this milestone) ✓	pool_size active, unlimited queued	Medium	Low	nginx workers, Java Servlet
epoll event loop (single thread)	100K+	High	Very low	nginx master loop, Redis, Node.js
io_uring + async	100K+	Very high	Minimal	Modern high-performance servers
The thread pool is the right choice for this project: it handles the real concurrency challenges (bounded resources, keep-alive, graceful shutdown) while remaining understandable. The epoll model — where a single thread monitors thousands of sockets and handles I/O events without blocking — is the next step on the optimization ladder, but it requires rearchitecting how you think about connection state. Understanding the thread pool model first makes the epoll model's tradeoffs legible.				

Testing Concurrent Behavior

```bash
# Build with thread sanitizer to catch data races:
gcc -Wall -Wextra -pthread -fsanitize=thread -g \
    -o http_server_tsan *.c
# --- Test 1: Parallel connections work correctly ---
./http_server 8080 /tmp/www &
for i in $(seq 50); do
    curl -s http://localhost:8080/index.html > /tmp/response_$i.html &
done
wait
# All responses should be identical
md5sum /tmp/response_*.html | awk '{print $1}' | sort | uniq -c
# Expected: one unique md5 with count 50
# --- Test 2: Pool exhaustion returns 503 ---
# Send 300 slow connections (pool default is 16)
# Use a client that keeps connection open without completing request
python3 - <<'EOF'
import socket, time, threading
def slow_connection(n):
    s = socket.socket()
    s.connect(('localhost', 8080))
    s.send(b'GET / HTTP/1.1\r\nHost: localhost\r\n')
    # Don't send the final \r\n — connection stays open
    time.sleep(5)
    s.close()
threads = [threading.Thread(target=slow_connection, args=(i,)) for i in range(300)]
```

```
for t in threads: t.start()

time.sleep(1)

# Now try a normal request — should get 503

import urllib.request

try:

    resp = urllib.request.urlopen('http://localhost:8080/')

    print(f'Got {resp.status} (expected 503)')

except urllib.error.HTTPError as e:

    print(f'Got {e.code} (correct: 503)' if e.code == 503 else f'Wrong code: {e.code}')

EOF

# --- Test 3: Keep-alive reuse ---

# HTTP/1.1 with keep-alive uses one TCP connection for multiple requests

curl -v --http1.1 http://localhost:8080/index.html \

         http://localhost:8080/style.css \

         http://localhost:8080/app.js

# In verbose output, look for:

# "Re-using existing connection" — curl reused the TCP connection

# --- Test 4: Idle timeout closes lingering connections ---

# Connect but don't send anything — should be closed after IDLE_TIMEOUT seconds

nc -v localhost 8080

# Just wait. After 30 seconds, the server should close the connection

# You'll see nc print "Connection closed by foreign host"

# --- Test 5: Graceful shutdown ---

./http_server 8080 /tmp/www &

SERVER_PID=$!

# Start a slow download (10MB file) that takes several seconds

dd if=/dev/urandom of=/tmp/www/bigfile.bin bs=1M count=10
```

```
curl -s http://localhost:8080/bigfile.bin > /dev/null &

CURL_PID=$!

sleep 1

# Send SIGTERM while curl is downloading

kill -SIGTERM $SERVER_PID

wait $CURL_PID

echo "curl exit code: $?"  # Should be 0 — download completed before server exit

# --- Test 6: Thread sanitizer for data races ---

./http_server_tsan 8080 /tmp/www &

for i in $(seq 200); do curl -s http://localhost:8080/ > /dev/null & done

wait

kill $(pgrep http_server_tsan)

# Check stderr for "DATA RACE" reports — there should be none

# --- Test 7: FD leak under sustained load ---

SERVER_PID=$(pgrep http_server)

BASE=$(ls /proc/$SERVER_PID/fd | wc -l)

for i in $(seq 10000); do curl -s http://localhost:8080/ > /dev/null; done

sleep 2

FINAL=$(ls /proc/$SERVER_PID/fd | wc -l)

echo "Base: $BASE, Final: $FINAL"

[ $((FINAL - BASE)) -le 2 ] && echo "PASS" || echo "FAIL: leaked $((FINAL - BASE)) FDs"
```

The Thread Sanitizer (`-fsanitize=thread`) is a compiler-level instrumentation tool that detects data races at runtime. It instruments every memory access and records which thread last wrote each memory location. When two threads access the same memory without synchronization, it reports the race with a complete stack trace

showing both threads. Running the TSan build under load is the definitive check that your mutex discipline is correct.

Knowledge Cascade: What This Unlocks

1. Connection Pool Sizing — The Universal Pattern

The bounded thread pool you just built is the same pattern used everywhere resources are finite and must be shared: **Database connection pools** (HikariCP, PgBouncer, pgpool-II): A PostgreSQL database can handle ~100–500 concurrent connections before performance degrades. An application server with 1,000 threads would saturate the database with 1,000 simultaneous connections. A connection pool (typically 10–50 connections) multiplexes the 1,000 application threads over the available database connections — exactly the same queue-and-wait model you built, with the same 503-equivalent (connection acquisition timeout). **HTTP client pools** (Go's `http.DefaultTransport`, Python's `requests.Session`, Java's `HttpClient`): Outgoing HTTP requests to external APIs go through a connection pool. `MaxIdleConnsPerHost` in Go limits how many idle connections to keep warm per target host. The idle timeout that you implemented per-connection is the same as `IdleConnTimeout` in Go's transport. **Goroutine limiting in Go** (`golang.org/x/sync/semaphore`, `errgroup`): Go goroutines are cheap (2KB stack), but unbounded goroutine creation is still a Slowloris-style attack surface. Production Go services use `semaphore.Weighted` or worker pool patterns — exactly what you built, with goroutines instead of pthreads. **Kubernetes pod autoscaling** (Horizontal Pod Autoscaler): The HPA watches CPU/memory utilization and adjusts the pod count. The underlying math is the same I/O-bound formula: `replicas = ceil(current_utilization / target_utilization)`. More replicas = bigger "pool" of workers. The HPA is Little's Law applied to a distributed system. The math that governs all of these — **Little's Law** (`L = λW`, where `L` is average queue length, `λ` is arrival rate, and `W` is average time in system) — was established by John Little in 1961. Understanding it as you've just implemented it makes every discussion of "pool size tuning" and "capacity planning" concrete rather than abstract.

2. Why nginx Uses Event Loops Instead of Thread Pools

Your thread pool model handles concurrency by giving each connection a thread. nginx's "worker" model handles concurrency differently: each worker process runs a single event loop using **epoll** (Linux's scalable I/O multiplexing interface), monitoring hundreds of thousands of sockets simultaneously with one thread. The thread pool model hits a wall at roughly 1,000–10,000 concurrent connections — not because threads are expensive per se, but because:

- Each blocked thread consumes a kernel task_struct whether it's actively processing or waiting for I/O
- Context switching between thousands of threads adds scheduler overhead
- Memory pressure from many stacks can cause TLB churn The epoll model breaks this wall by reversing the model: instead of "give each connection a thread," you "have one thread handle all connections,

switching between them when I/O is ready." epoll watches a set of file descriptors and tells you which ones have data available — so your one thread only wakes up when there is actual work to do, never burning CPU waiting. The C10K problem (handling 10,000 concurrent connections) was the precise point where thread-per-connection broke and event loops became necessary. Dan Kegel's 1999 paper "The C10K Problem" documented this breakdown and catalyzed the design of epoll (Linux 2.5.44, 2002). Since then, the C10M problem (10 million connections) has been solved by io_uring (Linux 5.1, 2019), which takes event loops further by allowing truly asynchronous I/O submission — no context switches even to check if I/O is ready. Every async/await keyword in Python, Rust, JavaScript, Go, and Kotlin traces back to this same insight that you've just hit the wall of: blocking-per-connection doesn't scale. Your thread pool is the last viable step before the event-loop revolution.

3. Lock Granularity — Why Databases Use Row Locks, Not Table Locks

You protected your shared queue with a single mutex. This is a **coarse-grained lock** — one lock for all shared state. It's correct, simple to reason about, and fast enough for your server's concurrency level. As concurrency scales, coarse-grained locks become bottlenecks. Every thread that wants to enqueue or dequeue must wait for the one global lock. If the critical section (the time between lock and unlock) is short, lock contention is low and it doesn't matter. If the critical section is long — or many threads contend simultaneously — the lock becomes a serialization point that eliminates the benefit of concurrency. PostgreSQL uses **row-level locking** instead of table-level locking: `UPDATE table SET x=1 WHERE id=5` locks only the row with `id=5`, allowing concurrent `UPDATE table SET x=2 WHERE id=7` on a different row. The lock granularity matches the actual contention scope. jemalloc, the memory allocator used by Firefox and FreeBSD, uses **per-CPU arena locking**: instead of one global lock for all allocations, each CPU core has its own arena with its own lock. Threads allocating memory on the same core share a lock; threads on different cores never contend. This reduces lock contention roughly proportionally to the number of CPU cores. The general lesson: choose your lock granularity by mapping it to the actual contention scope. A global mutex is correct. A per-connection mutex (protecting that connection's state only) allows full parallelism. A per-shard mutex (protecting 1/N of the shared data) is in between. Understanding the tradeoffs starts here, with your single-mutex thread pool.

4. Graceful Shutdown as a Distributed Systems Primitive

The four-phase shutdown sequence — stop accepting → drain in-flight → notify workers → wait for completion — is not just a multi-threading pattern. It is the fundamental primitive for any system that must handle state-changing operations without data loss. **Kubernetes pod termination** (`terminationGracePeriodSeconds`): When Kubernetes sends SIGTERM to a pod, it starts a grace period timer (default 30 seconds). The application should stop accepting new requests, complete in-flight requests, and exit cleanly. If it doesn't exit within `terminationGracePeriodSeconds`, Kubernetes sends SIGKILL. This is exactly your four-phase pattern with a deadline. **Rolling deployments**: When deploying a new version of a service, you stop traffic to old pods (stop accepting), wait for in-flight requests to complete (drain), then replace the old pod with the new one. The deployment controller implements your four-phase pattern at the infrastructure level. **Blue-green deployments**: Route traffic to the new ("green") environment. The old ("blue")

environment stops accepting new traffic but finishes serving its in-flight requests. Once drained, blue is terminated. Same pattern, distributed across environments. **Database connection draining** (PostgreSQL `pg_terminate_backend()`): When a DBA needs to take a database offline for maintenance, they call `pg_terminate_backend()` on sessions and then wait for active transactions to commit or roll back. Forcibly killing a transaction mid-way leaves the database in an inconsistent state — exactly why graceful shutdown (let transactions complete) matters. Every time you see "draining," "quiescing," or "graceful termination" in a distributed systems discussion, it's your four-phase pattern scaled up.

5. The C10K/C10M Problem and the async/await Revolution

Dan Kegel's 1999 "C10K Problem" paper described the failure mode you just implemented a defense against: a server that spawns one thread or process per connection breaks at roughly 10,000 concurrent connections on typical hardware. The paper asked: can we handle 10,000 connections simultaneously on a single machine? The answer required rethinking the programming model. epoll (2002), kqueue (BSD, 2000), IOCP (Windows NT 3.5) all emerged as kernel interfaces that let a single thread efficiently monitor thousands of connections. But writing epoll-based code in C is famously difficult — you manage connection state explicitly across non-blocking callbacks, fighting what's called "callback hell." The solution to callback hell is async/await — syntactic sugar that makes non-blocking I/O look like sequential code. Python's `asyncio`, JavaScript's Promise/async, Rust's `tokio`, Go's goroutines (which hide the event loop behind a scheduler that looks like threads) — all of them are the language ecosystem's answer to "how do we get the performance of epoll without the programmer experience of raw epoll?" Understanding why your thread pool hits a wall at scale — and why the alternative (one thread watching thousands of fds via epoll) is both faster and harder to code — is the historical context that makes async/await meaningful. You haven't just built a thread pool: you've

built the exact system that motivated the most significant paradigm shift in server-side programming of the past 25 years.

Common Mistakes That Will Burn You

1. Holding the mutex while doing network I/O. This is the most common threading mistake in server code. If you call `read()`, `write()`, or `serve_static_file()` while holding `pool->lock`, all other threads are blocked waiting for the lock for the entire duration of the I/O operation. Your "16-thread concurrent server" behaves like a sequential server. The rule: acquire the lock, modify the queue (a few pointer updates), release the lock, then do the I/O. **2. Checking the shutdown flag without the lock.**

```c
// WRONG — data race:

if (pool->shutdown) break;

// CORRECT:

pthread_mutex_lock(&pool->lock);

int s = pool->shutdown;

pthread_mutex_unlock(&pool->lock);

if (s) break;
```

Reading `pool->shutdown` without the mutex is a data race — undefined behavior per the C11 standard. The compiler may optimize away the repeated read, caching it in a register and never re-reading from memory ("the value can't change, it's not volatile"). An alternative is `atomic_int` with `atomic_load_explicit()` using the appropriate memory ordering — but using the mutex you already have is simpler and equally correct. **3. Forgetting `pthread_detach()` or `pthread_join()` for every thread.** The signal handler thread in our design is `pthread_detach()`'d — it runs once, handles one signal, and exits. Every `pthread_create()` must be matched by either `pthread_join()` (to collect exit status and free resources) or `pthread_detach()` (to let the thread clean up itself). An unjoined, undetached thread is a zombie: its task_struct stays in the kernel until the process exits. **4. Using `signal()` instead of `pthread_sigmask()` for multi-threaded signal masking.** `signal()` sets the disposition (what happens when a signal arrives) but does not control which thread receives it. Only `pthread_sigmask()` controls the signal mask per-thread. Setting `signal(SIGTERM, handler)` in a multi-threaded program means SIGTERM calls `handler` in whichever thread the kernel picks — which might be a worker thread in the middle of `write()`. Use `pthread_sigmask()` to block signals in all threads, then use `sigwait()` in a dedicated signal thread. **5. The `while` loop, not `if`, around `pthread_cond_wait()`.**

```c
// WRONG — spurious wakeup causes incorrect behavior:
if (pool->queue_size == 0) {
    pthread_cond_wait(&pool->not_empty, &pool->lock);
}
// here we assume queue is non-empty — WRONG if it was a spurious wakeup
// CORRECT:
while (pool->queue_size == 0 && !pool->shutdown) {
    pthread_cond_wait(&pool->not_empty, &pool->lock);
}
```

POSIX permits spurious wakeups from `pthread_cond_wait()`. A thread can return from the wait even if no signal was sent. Always re-check the condition in a `while` loop. 6. Not setting `SO_RCVTIMEO` before the keep-alive loop. If you set the timeout option inside the loop on each iteration, you incur an extra syscall per request. Set it once per connection, before the `for(;;)` loop. The timeout applies to all subsequent reads on that socket. 7. Not sending 503 when the pool queue is full — just closing the FD. Silently closing a TCP connection (without sending an HTTP response) leaves the client hanging, waiting for data that never arrives. The client's browser shows a "connection reset" error with no explanation. Sending a 503 response takes milliseconds and gives the client actionable information ("server busy, try later"). Always send a response before closing.

Acceptance Criteria Checklist

Before calling this project complete, verify each of these:

- [ ] `./http_server 8080 /tmp/www` starts with "Thread pool: 16 workers, queue capacity 256" (or configured values)

- [ ] `for i in $(seq 50); do curl -s http://localhost:8080/ & done; wait` — all 50 responses are `200 OK` with correct content
- [ ] Under Slowloris-style load (300 slow connections), the 301st connection receives `503 Service Unavailable` rather than timing out silently
- [ ] `curl -v --http1.1 http://localhost:8080/index.html http://localhost:8080/style.css` shows "Re-using existing connection" in verbose output
- [ ] An idle connection is closed after `idle_timeout` seconds (default 30) — verified with `nc localhost 8080` (no request sent)
- [ ] `Connection: close` in a request causes the server to close after responding — verified with `curl -H "Connection: close" -v http://localhost:8080/`
- [ ] SIGTERM causes graceful shutdown: in-flight downloads complete, server then exits with code 0
- [ ] SIGINT (Ctrl+C) causes the same graceful shutdown behavior as SIGTERM
- [ ] The thread sanitizer build (`-fsanitize=thread`) reports zero data races under parallel load
- [ ] FD count after 10,000 sequential connections matches baseline (within tolerance of 2)
- [ ] `pool->total_requests` counter accurately reflects total served requests (verify via server stats output or log)
- [ ] Server logs show correct client IP and request line from each worker thread, without interleaving (log mutex prevents garbling)
- [ ] After all workers are joined on shutdown, `valgrind --leak-check=full` reports no memory leaks
- [ ] Thread pool size and queue capacity are configurable (compile-time constants at minimum, runtime arguments preferred)

TDD

A socket-level HTTP/1.1 server built from first principles: raw TCP socket lifecycle, adversarial HTTP parsing per RFC 7230, security-hardened static file serving with path canonicalization, and concurrent connection handling via a bounded thread pool with keep-alive and graceful shutdown. Every abstraction that frameworks hide — partial reads, CRLF parsing, MIME detection, realpath security, mutex-protected shared state — is made explicit and implementation-grade.

MODULE SPECIFICATION: TCP Server & HTTP Response (http-server-basic-m1)

---

1. Module Charter

This module establishes the foundational network "Entry Point" for the HTTP server. It is responsible for the complete lifecycle of a TCP listening socket and the sequential handling of incoming client connections.

Core Responsibilities:

- Initialize a passive TCP socket (the "Listening Socket") bound to a configurable port.
- Implement the "Seven-Step Socket Dance": `socket`, `setsockopt`, `bind`, `listen`, `accept`, `read/write`, `close`.
- Solve the "Partial Read Problem" by accumulating raw bytes into a fixed-size buffer until the HTTP header delimiter (`\r\n\r\n`) is detected.
- Construct a standards-compliant (RFC 7230) hardcoded HTTP/1.1 200 OK response.
- Ensure robust resource management by preventing file descriptor (FD) leaks and handling broken pipe signals (SIGPIPE).

What this module does NOT do:

- It does **not** parse the HTTP request (it treats it as raw bytes).
- It does **not** handle concurrent connections (it is strictly sequential).
- It does **not** serve files from disk (the response is hardcoded in memory).

Invariants:

- Every successfully `accept()`'ed file descriptor MUST be `close()`'d before the next `accept()` call.
- The server must never crash due to a client disconnecting mid-write (SIGPIPE).

---

2. File Structure

The implementation shall follow this numbered creation order:

1. `server.h`: Data structures and function prototypes.
2. `utils.c`: Helper functions for robust I/O (`write_all`, `read_http_request`).
3. `http.c`: HTTP response construction logic.
4. `server.c`: The core socket lifecycle and accept loop.
5. `main.c`: CLI entry point and signal initialization.

---

3. Complete Data Model

3.1 Server Configuration Struct

This struct persists for the lifetime of the server process.

Field	Type	Offset (64-bit)	Description
`port`	`int`	0x00	The TCP port to bind (default 8080).
`server_fd`	`int`	0x04	The listening socket file descriptor.
`is_running`	`int`	0x08	Boolean flag for the accept loop.
`padding`	`char[4]`	0x0C	Alignment padding to 16-byte boundary.

Total Size: 16 bytes.

3.2 Constants

- `REQUEST_BUF_SIZE` : 8192 (8KB). Caps memory usage per request. Matches standard URI/header limits.
- `BACKLOG` : `SOMAXCONN` . The kernel-level queue for pending connections.
- `MAX_RESPONSE_SIZE` : 4096 (4KB). Sufficient for hardcoded headers and small HTML body.

---

4. Interface Contracts

4.1 `ssize_t read_http_request(int client_fd, char *buf, size_t buf_size, int *header_end)`

- **Purpose**: Accumulate bytes until `\r\n\r\n` is found.
- **Parameters**:
  - `client_fd` : The socket to read from.
  - `buf` : The destination buffer.
  - `buf_size` : Maximum capacity.
  - `header_end` : [OUT] Offset in `buf` where headers end (start of body).
- **Return**: Total bytes read on success, `0` on client disconnect, `-1` on error or buffer overflow.

4.2 `int write_all(int fd, const char *buf, size_t len)`

- **Purpose**: Ensure all bytes are written despite partial writes or interrupts.

- **Return**: `0` on success, `-1` on error.

4.3 `int build_hardcoded_response(char *buf, size_t buf_size)`

- **Purpose**: Populate `buf` with a valid HTTP response string.
- **Includes**: `Date` (RFC 1123 format), `Content-Type`, `Content-Length`, `Connection: close`.
- **Return**: Length of response string or `-1` on overflow.

---

5. Algorithm Specification

5.1 The Socket Dance (Phase 1-2)

1. **socket()**: Call `socket(AF_INET, SOCK_STREAM, 0)`.
2. **setsockopt()**: Set `SO_REUSEADDR` to `1` on `SOL_SOCKET`. This is mandatory for rapid server restarts.
3. **bind()**:
   - Initialize `struct sockaddr_in`.
   - Use `htons(port)` for the port (Network Byte Order).
   - Use `INADDR_ANY` for the IP.
4. **listen()**: Set backlog to `SOMAXCONN`.

5.2 The Accumulation Loop (Phase 3)

{{DIAGRAM:tdd-diag-1|HTTP Accumulation Flow|Flowchart of read() calls searching for delimiter}}

1. Initialize `total_read = 0`.
2. While `total_read < buf_size - 1`:
   - `n = read(client_fd, buf + total_read, buf_size - 1 - total_read)`.
   - If `n <= 0`: Return `n` (Error or Disconnect).
   - `total_read += n`.
   - `buf[total_read] = '\0'` (Null-terminate for `strstr`).
   - `found = strstr(buf, "\r\n\r\n")`.
   - If `found`:
     - Calculate `*header_end = (found - buf) + 4`.
     - Return `total_read`.
3. If loop exits without `found`: Return `-1` (Request too large).

5.3 The Sequential Accept Loop

1. Call `accept(server_fd, ...)` which blocks.

2. Log client IP/Port via `inet_ntop` and `ntohs`.
3. Invoke `read_http_request`.
4. If success: Invoke `build_hardcoded_response` and `write_all`.
5. **CRITICAL**: Call `close(client_fd)` in all branches (success or failure) before looping.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`EADDRINUSE`	`bind()`	Exit process with code 1.	Yes (Console)
`EINTR`	`accept()`	`continue` the loop immediately.	No
`EMFILE`	`accept()`	`sleep(1)`, log error, `continue`.	No
`EPIPE`	`write()`	Stop writing, `close(client_fd)`, log disconnect.	No
Buffer Overflow	`read_http_request`	`close(client_fd)`, do not respond.	No (Server logs only)

---

7. Implementation Sequence with Checkpoints

Phase 1: Socket Setup (1 hour)

- Implement `socket_init(int port)`.
- **Checkpoint**: Run `netstat -tulpn | grep :8080`. You should see the process in `LISTEN` state.

Phase 2: Accept Loop & Logging (0.5 hours)

- Implement the infinite loop calling `accept()`.
- **Checkpoint**: Connect via `nc localhost 8080`. Server should print "Connection from 127.0.0.1".

Phase 3: The Accumulator (1 hour)

- Implement `read_http_request`.
- **Checkpoint**: Use `telnet localhost 8080`. Type `GET /` and press Enter once. Server should NOT respond. Type `\r\n\r\n`. Server should now acknowledge receipt.

Phase 4: Response & write_all (1 hour)

- Implement `write_all` and `build_hardcoded_response`.
- Handle `SIGPIPE` using `signal(SIGPIPE, SIG_IGN)`.
- **Checkpoint**: `curl -v http://localhost:8080`. You should see full headers, the HTML body, and `* Connection #0 to host localhost left intact`.

---

8. Test Specification

8.1 Happy Path: curl

- **Command**: `curl -v http://localhost:8080/`
- **Requirement**: HTTP 200 OK, `Content-Length` matches body size, `Date` matches current UTC time.

8.2 Failure: Delimiter Search

- **Command**: `printf "GET / HTTP/1.1\r\n" | nc -q 2 localhost 8080`
- **Requirement**: Server should hang waiting for the final `\r\n`, then close connection (timeout or client quit). It must NOT send a 200 OK for a partial header.

8.3 Resource Leak Test

- **Command**: `for i in {1..1000}; do curl -s http://localhost:8080/ > /dev/null; done`
- **Requirement**: Check `ls /proc/$(pgrep http_server)/fd | wc -l`. The count must be identical before and after the loop.

---

9. Performance Targets

Operation	Target	How to Measure
Request Latency	< 2ms	`curl -w "%{time_total}\n" -o /dev/null -s http://localhost:8080/`
FD Baseline	< 5	`ls /proc/self/fd | wc -l` (stdin, out, err, server_fd + 1)
Throughput	> 2000 req/sec	`wrk -t1 -c1 -d10s http://localhost:8080/` (Sequential)

---

10. Hardware Soul & Memory Layout

Cache Line Optimization

The `char req_buf[8192]` is the most "touched" memory area.

- **Alignment**: The buffer should be 64-byte aligned to prevent "false sharing" and ensure efficient cache line fills (though less critical in single-threaded mode, it's good practice).
- **Access Pattern**: `strstr` scans sequentially. Modern CPUs (x86_64) will prefetch the next cache lines (64B segments) automatically during the scan.

Memory Layout Table

Memory Segment	Object	Size	Life Span
Stack	`req_buf`	8192 B	Connection Duration
Stack	`resp_buf`	4096 B	Response Generation
Static	`server_config`	16 B	Process Lifetime
Kernel	`socket_buffer`	~87 KB	Socket Lifetime

---

11. Concurrency Specification: The Barrier

This module uses a **Blocked Sequential Model**.

{{DIAGRAM:tdd-diag-2|Sequential Blocking|Timeline showing Accept -> Read -> Write -> Close}}

1. The `accept()` call blocks the main thread.
2. While processing, any other clients are held in the **Kernel TCP Backlog**.
3. If the backlog (e.g., 128) fills up, new clients will receive "Connection Refused" (ECONNREFUSED).
4. **Benefit**: No race conditions, no mutexes, trivial to debug.
5. **Constraint**: A slow client (Slowloris) can hang the server by connecting and sending nothing. This will be solved in Milestone 4.

```c
/* interface contract signatures */

int server_init(server_config_t *config, int port);

ssize_t read_http_request(int client_fd, char *buf, size_t buf_size, int *header_end);

int write_all(int fd, const char *buf, size_t len);

int build_hardcoded_response(char *buf, size_t buf_size);

void handle_client(int client_fd);
```

MODULE SPECIFICATION: HTTP Request Parsing (http-server-basic-m2)

1. Module Charter

This module serves as the semantic gateway of the server, transforming raw, untrusted byte streams from the network (Milestone 1) into a structured, validated `http_request_t` object. It acts as the primary defense against protocol-level attacks and malformed inputs.

Core Responsibilities:

- Implement an adversarial-safe parser for HTTP/1.1 request lines and headers.
- Enforce strict RFC 7230 structural constraints (CRLF line endings, single space delimiters).
- Normalize header names to lowercase for case-insensitive lookup.
- Extract semantic control fields: `Content-Length` (for body framing), `Connection` (for keep-alive), and `Host` (mandatory in HTTP/1.1).
- Prevent buffer overruns and resource exhaustion via hard URI and header length limits.
- Generate and transmit standards-compliant HTTP error responses (400, 414, 501) for invalid inputs.

Downstream Dependencies:

- Milestone 3 (Static File Serving) relies on the `path` and `method` fields for resource location.
- Milestone 4 (Concurrency) relies on the `keep_alive` flag to manage connection persistence.

Invariants:

- The parser MUST NOT modify the original request buffer (except for temporary null-termination during line-at-a-time processing).
- The parser MUST perform zero dynamic memory allocations (`malloc`) to ensure predictable latency and prevent heap fragmentation.

- All extracted strings (path, header values) MUST be null-terminated within the `http_request_t` structure.

---

2. File Structure

The implementation is realized in the following sequence:

1. `http_types.h`: Definition of enums, constants, and the monolithic `http_request_t` struct.
2. `http_parser.h`: Prototypes for parsing and header lookup functions.
3. `http_parser.c`: Implementation of the state-machine/line-iterator parsing logic.
4. `http_errors.h`: Error response templates and constants.
5. `http_errors.c`: Implementation of `send_error_response` and convenience wrappers.

---

3. Complete Data Model

3.1 Enumerations

```c
typedef enum {
    METHOD_GET,
    METHOD_HEAD,
    METHOD_POST,
    METHOD_UNKNOWN
} http_method_t;
```

3.2 Header Entry Struct

Each header is stored in a fixed-size slot.

Field	Type	Size	Description
`name`	`char[256]`	256 B	Lowercase normalized header name.
`value`	`char[8192]`	8192 B	OWS-stripped header value.

3.3 The HTTP Request Struct (`http_request_t`)

This structure is designed for stack allocation within the connection handler. Note the significant size due to fixed-buffer constraints (approx 270KB).

Field	Type	Offset	Description
`method`	`int`	0x00	`http_method_t` enum value.
`path`	`char[8192]`	0x04	Parsed URI path (null-terminated).
`http_minor`	`int`	0x2004	0 for 1.0, 1 for 1.1.
`headers`	`http_header_t[32]`	0x2008	Contiguous array of header pairs.
`header_count`	`int`	0x44008	Current valid entries in `headers`.
`content_length`	`int`	0x4400C	-1 if absent, else byte count.
`keep_alive`	`int`	0x44010	1 if persistent, 0 if close.
`body`	`const char*`	0x44018	Pointer into the original M1 buffer.

Total Size: ~278,560 Bytes. *Note: Ensure the server's thread stack size (default 8MB on Linux) is respected. If spawning many threads in M4, use `pthread_attr_setstacksize` if 270KB per thread is problematic.*

{{DIAGRAM:tdd-diag-7|Request Struct Memory Layout|Visualization of the 270KB stack-allocated struct and its alignment}}

---

4. Interface Contracts

4.1 `int parse_http_request(const char *buf, size_t buf_len, int header_end, http_request_t *req)`

- **Inputs**: Raw buffer, total bytes read, and the offset of `\r\n\r\n`.
- **Outputs**: 0 on success, or an HTTP error code (400, 414, 501) on failure.
- **Contract**: Rejects any request with more than 32 headers or lines exceeding 8KB.

4.2 `const char* request_get_header(const http_request_t *req, const char *name)`

- **Inputs**: Request struct and a **lowercase** name to search for.
- **Return**: Pointer to the value string, or `NULL` if not found.
- **Complexity**: O(N) where N is `header_count`.

4.3 `void send_error_response(int client_fd, int status, const char *reason, const char *body)`

- **Inputs**: Socket FD, status code (e.g., 414), phrase (e.g., "URI Too Long"), and HTML body.
- **Invariants**: Must include `Date`, `Content-Length`, and `Connection: close`.

---

5. Algorithm Specification

5.1 Request Line Parsing (`parse_request_line`)

The request line is the first line of the buffer. It must follow the format `METHOD SP URI SP VERSION`.

1. **Token 1 (Method)**:
   - Locate the first ` ` (space) using `memchr`.
   - Length check: Must be between 3 and 4 characters for supported methods.
   - Use `memcmp` to map `GET`, `HEAD`, `POST` to enums. If valid but unsupported, return `501`.
2. **Token 2 (URI)**:
   - Locate the second ` ` (space).
   - Distance between spaces is the `path_len`.
   - **Boundary Check**: If `path_len >= 8192`, return `414`.
   - Copy to `req->path` and null-terminate.
3. **Token 3 (Version)**:
   - Must be exactly `HTTP/1.0` or `HTTP/1.1`.
   - Verify prefix `HTTP/1.`.
   - Set `req->http_minor` based on the final digit. If not '0' or '1', return `400`.

{{DIAGRAM:tdd-diag-8|Request Line Tokenization|Step-by-step pointers moving through the first line}}

5.2 Header Normalization and Storage (`parse_header_line`)

Executed for every line after the first until a blank line is hit.

1. **Split**: Locate first `:` using `strchr`. If no colon, skip line (robustness).
2. **Name Processing**:
   - Copy prefix (name) to `headers[i].name`.
   - Invoke `str_to_lower` on the field name. This facilitates O(1) case-insensitive comparisons later.
3. **Value Processing (OWS Strip)**:
   - Pointer `p` starts at `colon + 1`.
   - While `*p == ' ' || *p == '\t'`, increment `p`.

- Locate end of string; backtrack to remove trailing spaces.
- Copy to `headers[i].value`.

4. **Counter**: Increment `req->header_count`. If `> 32`, stop parsing further headers.

5.3 Semantic Extraction (`extract_semantic_headers`)

Post-parse logic to set control flags.

1. **Host**:
   - `request_get_header(req, "host")`.
   - If `http_minor == 1` and `host == NULL`, return `400` (Required in HTTP/1.1).
2. **Content-Length**:
   - `val_str = request_get_header(req, "content-length")`.
   - Use `strtol(val_str, &end, 10)`. If `*end != '\0'` or value `< 0`, set `content_length = -1`.
3. **Keep-Alive**:
   - If `http_minor == 1`: Default `1`, set `0` only if `Connection: close`.
   - If `http_minor == 0`: Default `0`, set `1` only if `Connection: keep-alive`.

---

6. Error Handling Matrix

Error Condition	Status Code	Detection Point	Handling
Path > 8191 chars	414	`parse_request_line`	Call `send_414()`, close socket
Method is `DELETE`	501	`parse_request_line`	Call `send_501()`, close socket
Missing `Host` (1.1)	400	`extract_semantic_headers`	Call `send_400()`, close socket
Space in URI	400	`parse_request_line`	RFC violation, 400
Invalid `Content-Length`	400	`extract_semantic_headers`	Reject request
Header too long	Skip	`parse_header_line`	Skip specific header, continue

---

7. Implementation Sequence with Checkpoints

Phase 1: Structs & Constants (0.5 hours)

- Define `http_request_t` and `http_header_t` in `http_types.h`.
- Set `MAX_PATH_LEN 8192` and `MAX_HEADERS 32`.
- **Checkpoint**: Compile a test file that `sizeof(http_request_t)` and prints it. Ensure it is ~270KB.

Phase 2: String Helpers (1 hour)

- Implement `str_to_lower` and `strip_ows`.
- Implement `url_decode` (basic version: convert `%XX` to char).
- **Checkpoint**: Test `strip_ows(" value ")` returns `"value"`.

Phase 3: The Parser (2 hours)

- Implement `parse_request_line`.
- Implement `parse_header_line`.
- Implement the main loop in `parse_http_request`.
- **Checkpoint**: Pass a raw buffer of a valid GET request. Verify `req.method` is `METHOD_GET` and `req.path` is correct.

Phase 4: Error Dispatch (1 hour)

- Implement `send_error_response` and convenience wrappers (`send_400`, etc.).
- Integrate into the main server loop from M1.
- **Checkpoint**: Use `curl -X DELETE http://localhost:8080`. Server should return 501.

---

8. Test Specification

8.1 Happy Path: Header Normalization

- **Input**: `GET / HTTP/1.1\r\nHOST: localhost\r\nUser-Agent: test\r\n\r\n`
- **Assertion**: `request_get_header(req, "host")` returns `"localhost"`. `req->http_minor == 1`.

8.2 Edge Case: Bare LF

- **Input**: `GET /index.html HTTP/1.1\nHost: localhost\n\n`
- **Assertion**: Parser treats `\n` as line terminator, correctly extracts path `/index.html`.

8.3 Attack: URI Too Long

- **Input**: `GET /` + ('A' * 9000) + `HTTP/1.1\r\n...`
- **Assertion**: `parse_http_request` returns `414`. `send_error_response` is called with 414.

8.4 Attack: Null Byte Injection

- **Input**: `GET /index.html%00.php HTTP/1.1\r\n...`
- **Assertion**: `url_decode` detects `%00` and returns an error (400), or the resulting string is correctly terminated and handled as `/index.html`.

---

9. Performance Targets

Operation	Target	Measurement Method
Request Parsing	< 5µs	`clock_gettime(CLOCK_MONOTONIC)` around `parse_http_request`
Normalization	< 1ns / char	Microbenchmark of `str_to_lower`
Memory Allocation	0 bytes	`valgrind --tool=massif` or manual audit (No `malloc` in scope)

---

10. Hardware Soul: Cache and Pipeline

10.1 Cache Locality

Because `http_request_t` is contiguous, the first few headers (usually the most important like `Host` and `Content-Length`) will likely share cache lines. When `request_get_header` iterates, the CPU's **L1 Data Cache Prefetcher** will pull the next header entries into the cache before the loop even reaches them.

10.2 Branch Prediction

In `str_to_lower`, the check `if (c >= 'A' && c <= 'Z')` is a branch. In standard HTTP headers, most characters are lowercase. The **Branch Predictor** in modern CPUs (like Intel's TAGE predictor) will learn this and "speculate" that the character is already lowercase, executing the add-32 path only when necessary.

{{DIAGRAM:tdd-diag-9|Header Search Pipeline|Visualizing L1 cache hits during the linear search of the headers array}}

---

11. State Machine Specification

The parser moves through the following states per connection:

1. **STATE_REQ_LINE**: Expecting Method, Path, and Version. Transition to `STATE_HEADERS` on first CRLF.
2. **STATE_HEADERS**: Extracting key-value pairs. Transition to `STATE_SEMANTIC` on `\r\n\r\n`.
3. **STATE_SEMANTIC**: Post-parse validation (Host check, etc.).
4. **STATE_COMPLETE**: Parsed successfully.
5. **STATE_ERROR**: Invalid input detected. Send 4xx/5xx and terminate.

{{DIAGRAM:tdd-diag-10|Parser States|State machine showing transitions and error exits}}

---

12. Implementation Path (C Details)

Warning: `strtok` is Forbidden. Use `memchr` and pointer arithmetic. `strtok` is not thread-safe (uses internal static state) and modifies the source string. While M1 creates a copy, using `memchr` is a better systems-programming habit for immutable buffers.

Handling CRLF vs LF:

```c
// Implementation snippet for line splitting

const char *line_start = cursor;

const char *line_end = memchr(cursor, '\n', remaining_len);

if (line_end) {

    size_t actual_len = line_end - line_start;

    if (actual_len > 0 && line_start[actual_len - 1] == '\r') {

        actual_len--; // Strip \r

    }

    // process line_start with actual_len...

}
```

Normalization Snippet:

```c
void str_to_lower(char *s) {
    for (; *s; ++s) {
        // Bitwise trick: 'A' | 0x20 == 'a'
        // Only applies if it is actually an uppercase letter
        if (*s >= 'A' && *s <= 'Z') *s |= 0x20;
    }
}
```

MODULE SPECIFICATION: Static File Serving (http-server-basic-m3)

1. Module Charter

This module implements the core "Payload Delivery" logic of the HTTP server. It is responsible for mapping abstract URL paths to concrete filesystem resources while enforcing strict security boundaries.

Core Responsibilities:

- **Path Sanitization**: URL-decode incoming paths and reject malicious payloads (e.g., null-byte injections).
- **Security Enforcement**: Use `realpath()` canonicalization to prevent directory traversal attacks by ensuring resolved paths reside strictly within the configured `document_root`.
- **Resource Resolution**: Handle directory-to-index mapping (auto-serving `index.html`) and detect file metadata via `stat()`.
- **Protocol Compliance**: Implement conditional GETs using `If-Modified-Since` and `Last-Modified` headers to support browser caching (304 Not Modified).
- **Content Negotiation**: Map file extensions to MIME types using a static lookup table.
- **Efficient I/O**: Stream file data from the kernel page cache to the network socket using a performance-tuned 64KB buffer loop.

What this module does NOT do:

- It does **not** manage connection persistence (Keep-Alive), which is handled by the M4 orchestrator.
- It does **not** handle directory listings (it returns 403 if `index.html` is missing).
- It does **not** perform compression (gzip/brotli).

Invariants:

- Every file opened via `open()` MUST be `close()`'d regardless of whether the `write()` to the socket succeeds or fails.
- The server MUST NOT follow symbolic links that point outside the `document_root`.
- All timestamps in HTTP headers MUST be in GMT/UTC per RFC 7231.

---

2. File Structure

The implementation follows this numbered creation order:

1. `mime_types.h`: Static MIME type mapping table and lookup logic.
2. `path_utils.h / .c`: `url_decode` and `resolve_safe_path` logic.
3. `date_utils.h / .c`: RFC 1123 date formatting/parsing using `timegm`.
4. `file_handler.h`: Prototypes for the main serving entry point.
5. `file_handler.c`: Implementation of the 8-phase serving orchestration.

---

3. Complete Data Model

3.1 MIME Entry Struct

Used in a static array to map extensions to types.

Field	Type	Offset	Description
`extension`	`const char*`	0x00	e.g., ".html", ".png" (includes dot).
`mime_type`	`const char*`	0x08	e.g., "text/html; charset=utf-8".

Total Size: 16 bytes.

3.2 Internal Constants

- `MAX_PATH_SIZE`: 4096 (Defined by `PATH_MAX` in `limits.h`).
- `IO_BUFFER_SIZE`: 65536 (64KB). Aligned with modern page cache/TCP window scaling.
- `HTTP_DATE_SIZE`: 64. Buffer size for "Wed, 21 Oct 2015 07:28:00 GMT".

{{DIAGRAM:tdd-diag-13|MIME Table Layout|Memory representation of the static MIME_TABLE array}}

---

4. Interface Contracts

4.1 `int url_decode(const char *src, char *out, size_t out_size)`

- **Purpose**: Convert `%XX` hex sequences to raw bytes and reject `%00`.
- **Input**: `src` (untrusted URL path), `out` (destination), `out_size` (max capacity).
- **Return**: `0` on success, `-1` on malformed hex or null-byte detection.
- **Security**: If `%00` is detected, return `-1` immediately to prevent filename truncation attacks.

4.2 `int resolve_safe_path(const char *url_path, const char *canonical_root, char *resolved_out)`

- **Purpose**: The "Security Firewall." Converts URL to absolute path and checks boundaries.
- **Input**: Raw URL path, canonicalized document root.
- **Output**: Writes absolute filesystem path to `resolved_out`.
- **Return Codes**:
  - `0`: Success.
  - `400`: Malformed encoding.
  - `403`: Traversal attempt (escaped root).
  - `404`: Resource does not exist (via `realpath` failure).

4.3 `void serve_static_file(const http_request_t *req, int client_fd, const char *canonical_root)`

- **Purpose**: Main entry point. Orchestrates the 8 phases of file delivery.
- **Contract**: Rejects non-regular files (pipes, sockets, devices) with 403.
- **Performance**: Uses a stack-allocated 64KB buffer for the I/O loop.

---

5. Algorithm Specification

5.1 The Resolve-Then-Check Algorithm

{{DIAGRAM:tdd-diag-14|Path Resolution Sequence|Flow from raw URL to canonical path validation}}

1. **Decode**: Invoke `url_decode()` on `req->path`.
2. **Concatenate**: `snprintf` the `canonical_root` and `decoded_path` into a temporary `raw_path` buffer.
3. **Canonicalize**: Call `realpath(raw_path, resolved_out)`.
   - If `NULL`:

        - If `errno == ENOENT`, return `404`.
        - If `errno == EACCES`, return `403`.
        - Else, return `404`.
4. **Boundary Check**:
    - `root_len = strlen(canonical_root)`.
    - Perform `strncmp(resolved_out, canonical_root, root_len)`.
    - If no match, return `403`.
5. **Character Boundary Check**:
    - If `resolved_out[root_len]` is not `\0` AND not `/`, return `403`. (Prevents matching `/var/www-secret` against `/var/www`).

5.2 Directory Index Logic (`handle_directory_path`)

1. Call `stat(resolved_path, &st)`.
2. If `S_ISDIR(st.st_mode)`:
    - Check if `resolved_path` + `/index.html` exceeds `MAX_PATH_SIZE`.
    - Append `/index.html`.
    - Call `stat()` again on the new path.
    - If second `stat()` fails, return `-1` (triggers 403 Forbidden).
3. Return `0` (Success, serving index).

5.3 64KB I/O Stream Loop

{{DIAGRAM:tdd-diag-15|Zero-Allocation I/O Loop|Relationship between Kernel Page Cache and userspace buffer}}

1. `int file_fd = open(resolved, O_RDONLY)`.
2. While `(bytes_read = read(file_fd, io_buf, 65536)) > 0`:
    - Invoke `write_all(client_fd, io_buf, bytes_read)`.
    - If `write_all` fails (client disconnected), break loop.
3. `close(file_fd)`. *Note: This utilizes the "Stream Pattern". For sequential reads, the CPU prefetcher will pull subsequent 64B cache lines into L1 while the current 64B is being written to the socket.*

---

6. Error Handling Matrix

Error Condition	Detected By	Recovery	User-Visible Response
`%00` in URL	`url_decode`	Stop processing.	`400 Bad Request`
Symlink to `/etc`	`resolve_safe_path`	`realpath` resolves to `/etc`, `strncmp` fails.	`403 Forbidden`
File does not exist	`realpath`	Check `errno == ENOENT`.	`404 Not Found`
Missing `index.html`	`stat` (2nd call)	Fail directory resolution.	`403 Forbidden`
Client quit mid-file	`write_all`	Close `file_fd`, return thread to pool.	N/A (Socket closed)
Header buffer overflow	`snprintf`	Return code check on `snprintf`.	`500 Internal Server Error`

7. Implementation Sequence with Checkpoints

Phase 1: URL & Path Security (1.5 hours)

- Implement `url_decode` with hex-to-char logic.
- Implement `resolve_safe_path` using `realpath()`.
- **Checkpoint**: Test `resolve_safe_path("/../../etc/passwd", "/var/www", out)`. It must return `403` or `404`, never `0`.

Phase 2: Metadata & MIME (1 hour)

- Define `MIME_TABLE` in `mime_types.h`.
- Implement `mime_type_for_path` with `strrchr` extension extraction.
- **Checkpoint**: `mime_type_for_path("test.PNG")` must return `image/png`.

Phase 3: Conditional GETs (1.5 hours)

- Implement `format_http_date` and `parse_http_date`.
- Implement `should_send_304` comparing `st_mtime` and `If-Modified-Since`.
- **Checkpoint**: Set a file's mtime to 1 hour ago. Send an `If-Modified-Since` header with that timestamp. Server must return `304`.

Phase 4: The Orchestrator (2 hours)

- Implement `serve_static_file` linking all previous phases.
- Implement the 64KB `read/write` loop.
- **Checkpoint**: `curl http://localhost:8080/index.html`. Verify headers and body are fully delivered.

---

8. Test Specification

8.1 Security: Traversal Prevention

- **Input**: `GET /%2e%2e%2f%2e%2e%2fetc%2fshadow`
- **Requirement**: HTTP 403 or 404. Audit `strace` to ensure `open("/etc/shadow")` was never called.

8.2 Logic: Directory Indexing

- **Structure**: `/www/images/index.html` exists.
- **Input**: `GET /images/`
- **Requirement**: Returns content of `/www/images/index.html`. `Content-Type: text/html`.

8.3 Efficiency: 64KB Alignment

- **Input**: Request a 10MB binary file.
- **Requirement**: Monitor memory usage. Peak RSS must not grow significantly (staying within the 64KB buffer limit).

---

9. Performance Targets

Operation	Target	Measurement
Path Resolution	< 15µs	`CLOCK_MONOTONIC` across `resolve_safe_path`.
Throughput (L1 Cached)	> 800 MB/s	`wget` on localhost for 1GB file.
MIME Lookup	< 500ns	Micro-benchmark of `mime_type_for_path`.
Allocations	0 malloc/req	`valgrind --tool=memcheck` check.

---

10. Hardware Soul & Systems View

10.1 The Three-Level View

- **Application**: Your C code calls `serve_static_file`.
- **OS/Kernel**:
  - `realpath()` performs `lstat()` on every directory component. This is a metadata-heavy operation.
  - `read()` triggers the kernel to check the **Page Cache**. If found, the kernel uses its own memory-to-memory copy.
  - `write()` fills the **TCP Send Buffer**. If the window is full, the thread blocks in the kernel, yielding the CPU.
- **Hardware**:
  - **DMA (Direct Memory Access)**: When the file is cold, the Disk Controller moves data to RAM without CPU involvement.
  - **Cache Line Locality**: The `io_buf[65536]` is accessed sequentially. The CPU's L2 prefetcher fetches the next cache line (64B) while the current line is being processed by the `write()` syscall.

{{DIAGRAM:tdd-diag-16|Three-Level View: File Serving|Interaction between NIC, CPU Cache, and Page Cache}}

10.2 Cache Line Alignment

The `io_buf` should ideally be aligned to a 64-byte boundary.

```c
// System-level alignment for optimal cache line fills
char io_buf[IO_BUFFER_SIZE] __attribute__((aligned(64)));
```

This ensures that every `read()` call fills exactly 1024 complete cache lines, preventing "partial cache line" updates which can trigger unnecessary cache coherence traffic on multi-core systems (even though M3 is single-request, the hardware still performs these checks).

10.3 Syscall Overhead: realpath()

`realpath()` is expensive because it involves multiple `lstat()` calls. For a path like `/a/b/c/d.txt`, there are 4 `lstat()` calls. At ~2µs per syscall, `realpath()` is the primary latency contributor for small requests. **Optimization Note**: The `canonical_root` is resolved once at startup to avoid re-calculating the base directory's components on every request.

---

11. Implementation Path (C Details)

11.1 Date Handling (UTC)

```c
// MUST use timegm for UTC, as mktime assumes local timezone

time_t parse_http_date(const char *date_str) {

    struct tm tm;

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

    if (strptime(date_str, "%a, %d %b %Y %H:%M:%S GMT", &tm) == NULL) return -1;

    return timegm(&tm);

}
```

11.2 File Descriptor Safety

```c
int file_fd = open(path, O_RDONLY);

if (file_fd < 0) { /* handle error */ }


// ... send headers ...


ssize_t n;

while ((n = read(file_fd, buf, sizeof(buf))) > 0) {

    if (write_all(client_fd, buf, n) < 0) break;

}


// Invariant: Close is guaranteed

close(file_fd);
```

11.3 MIME Lookup with strrchr

```c
const char *ext = strrchr(path, '.');

if (!ext) return "application/octet-stream";

// Note: strrchr finds the LAST dot, handling "archive.tar.gz" correctly by finding ".gz"
```

MODULE SPECIFICATION: Concurrent Connections (http-server-basic-m4)

1. Module Charter

This module transforms the server from a sequential, single-request processor into a high-concurrency production-grade engine. It implements a bounded thread pool architecture to manage system resources predictably while serving multiple clients in parallel. The module is responsible for the transition from one-shot connections to HTTP/1.1 persistent (keep-alive) sessions, enforced by kernel-level idle timeouts (`SO_RCVTIMEO`). It establishes a robust "Signal Handling" architecture using a dedicated thread to ensure that `SIGTERM` and `SIGINT` trigger a graceful four-phase shutdown rather than immediate process termination. Crucially, it manages all shared state (request counters, connection slots) through strict mutex synchronization to prevent data races.

Core Responsibilities:

- Manage a fixed-size pool of worker threads and a circular FIFO work queue.
- Implement the HTTP/1.1 Keep-Alive loop allowing multiple requests per TCP session.
- Protect shared server statistics and the access log using mutexes.
- Defend against resource exhaustion (Slowloris) via idle timeouts.
- Orchestrate graceful shutdown: stop accepting, drain queue, join threads.
- Handle `EMFILE` (file descriptor exhaustion) with exponential backoff.

Invariants:

- No shared variable (e.g., `active_connections`) shall be modified without holding the pool mutex.
- The `pool->lock` MUST NEVER be held during any blocking I/O operation (`read`, `write`, `open`).
- Every `pthread_create` must be balanced by a `pthread_join` or `pthread_detach`.

2. File Structure

The implementation follows this numbered creation order:

1. `concurrency_types.h` : Definition of `thread_pool_t` and `work_item_t` .
2. `thread_pool.c` : Core logic for initialization, worker loops, and submission.
3. `connection_handler.c` : Revised `handle_connection` with Keep-Alive loop and `SO_RCVTIMEO` .
4. `signals.c` : Dedicated signal handling thread logic.
5. `main.c` : Final integration of signal masking and pool orchestration.

---

3. Complete Data Model

3.1 Work Item Struct

A minimal descriptor for tasks in the queue.

Field	Type	Offset	Description
`client_fd`	`int`	0x00	The file descriptor of the accepted connection.
`padding`	`char[4]`	0x04	Alignment to 8-byte boundary.

3.2 Thread Pool Struct (`thread_pool_t`)

This struct contains the entire concurrency state. It is 64-byte aligned to optimize cache line utilization.

Field	Type	Offset (64-bit)	Description
Queue Management			
`queue`	`work_item_t*`	0x00	Pointer to heap-allocated circular buffer.
`queue_capacity`	`int`	0x08	Max items the queue can hold.
`queue_size`	`int`	0x0C	Current number of items in queue.
`queue_head`	`int`	0x10	Next item to dequeue.
`queue_tail`	`int`	0x14	Next slot for enqueue.
Synchronization			
`lock`	`pthread_mutex_t`	0x18	Global pool lock (approx 40 bytes).
`not_empty`	`pthread_cond_t`	0x40	Signaled when work is added.
`not_full`	`pthread_cond_t`	0x70	Signaled when space is cleared.
Workers & Life			
`threads`	`pthread_t*`	0xA0	Array of thread handles.
`num_threads`	`int`	0xA8	Number of worker threads.
`shutdown`	`int`	0xAC	Shutdown flag (0 or 1).
Config & Stats			
`doc_root`	`char*`	0xB0	Document root (read-only).
`idle_timeout`	`int`	0xB8	Seconds until idle close.
`active_conns`	`int`	0xBC	Count of threads currently serving.
`total_reqs`	`long`	0xC0	Cumulative request counter.

Cache Note: `active_conns` and `total_reqs` are placed at the end to minimize "False Sharing" contention with the queue pointers which are modified at much higher frequency.

{{DIAGRAM:tdd-diag-21|Memory Layout of thread_pool_t|Visualizing offsets and 64-byte alignment boundaries}}

---

4. Interface Contracts

4.1 `int thread_pool_init(thread_pool_t *pool, int threads, int queue_size, const char *root, int timeout)`

- **Purpose**: Initialize all sync primitives and spawn worker threads.
- **Return**: `0` on success, `-1` on any failure (e.g., `pthread_create` error).
- **Invariants**: If thread creation fails midway, it must cleanup all previously created threads before returning.

4.2 `int thread_pool_submit(thread_pool_t *pool, int client_fd)`

- **Purpose**: Push a new FD into the circular queue.
- **Return**: `0` on success, `-1` if queue is full or pool is shutting down.
- **Locking**: Must acquire `pool->lock`. Must signal `not_empty`.

4.3 `void thread_pool_shutdown(thread_pool_t *pool)`

- **Purpose**: Set shutdown flag and broadcast to all workers.
- **Contract**: Does not join threads; merely signals intent.

4.4 `void handle_connection(int client_fd, thread_pool_t *pool)`

- **Purpose**: Revised request handler with Keep-Alive for-loop.
- **Contract**: Must set `SO_RCVTIMEO` before entering the loop. Must check `req.keep_alive` after each response.

---

5. Algorithm Specification

5.1 The Worker Loop (`worker_thread_fn`)

1. **Wait**:
   - Lock `pool->lock`.
   - While `pool->queue_size == 0` AND `pool->shutdown == 0`:
     - `pthread_cond_wait(&pool->not_empty, &pool->lock)`.
2. **Shutdown Check**:
   - If `pool->shutdown == 1` AND `pool->queue_size == 0`, unlock and `pthread_exit`.
3. **Dequeue**:
   - Copy `pool->queue[pool->queue_head]` to local variable.
   - `pool->queue_head = (pool->queue_head + 1) % pool->queue_capacity`.

- `pool->queue_size--`.
- `pool->active_conns++`.
- `pthread_cond_signal(&pool->not_full)`.
- Unlock `pool->lock`.

4. **Execute**:
   - Call `handle_connection(local_fd, pool)`.
5. **Cleanup**:
   - Lock `pool->lock`.
   - `pool->active_conns--`.
   - Unlock `pool->lock`.

{{DIAGRAM:tdd-diag-22|Worker Thread State Machine|Transitions between Idle, Waiting, and Serving}}

5.2 HTTP/1.1 Keep-Alive Loop

Within `handle_connection`:

1. **Timeout Setup**: `setsockopt(client_fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv))`.
2. **Loop**:
   - Call `read_http_request`.
   - If `read` returns `-1` (including `EAGAIN` from timeout), `break` loop.
   - Call `parse_http_request`.
   - If success, call `serve_static_file`.
   - If `req.keep_alive == 0`, `break` loop.
   - Else, update stats and continue to next `read`.
3. **Exit**: `close(client_fd)`.

5.3 Dedicated Signal Thread (`signal_handler_thread`)

1. Clear and populate `sigset_t` with `SIGINT` and `SIGTERM`.
2. Call `sigwait(&set, &sig)`.
3. On signal:
   - Call `thread_pool_shutdown()`.
   - `close(listening_socket_fd)` to break the main accept loop.

---

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
`EAGAIN` / `EWOULDBLOCK`	`read()`	Idle timeout. Break keep-alive loop, `close(fd)`.	No
`EMFILE`	`accept()`	Log error. `usleep(100000)` (100ms backoff) and `continue`.	No
Queue Full	`thread_pool_submit`	Invoke `send_503(fd)` then `close(fd)`.	Yes (503 Error)
`pthread_create` fail	`thread_pool_init`	Set `shutdown=1`, broadcast, join created threads, return -1.	No (Server fails start)
`EINTR`	`accept()`	Common if debugger attached. `continue`.	No

7. Implementation Sequence with Checkpoints

Phase 1: Shared State & Initialization (2 hours)

- Define `thread_pool_t`. Implement `thread_pool_init` and the circular queue logic.
- **Checkpoint**: Run test that initializes a pool of 4 threads and verifies that 4 threads are visible in `htop` for the process.

Phase 2: Worker Logic (2 hours)

- Implement `worker_thread_fn` with `pthread_cond_wait`. Implement `thread_pool_submit`.
- **Checkpoint**: Submit 10 dummy tasks (print sleep) to a pool of 2. Verify tasks are processed concurrently and the queue drains correctly.

Phase 3: Keep-Alive & Timeouts (2 hours)

- Refactor `handle_connection` to include the `SO_RCVTIMEO` and the `for(;;)` loop.
- **Checkpoint**: `telnet localhost 8080`. Send one request. Wait 30 seconds. Verify the server closes the connection automatically.

Phase 4: Graceful Shutdown (1 hour)

- Implement `sigwait` thread and `thread_pool_join`.

- **Checkpoint**: Start server, hammer it with `wrk`, then hit `Ctrl+C`. Verify "All worker threads exited cleanly" is printed and process exits with code 0.

---

8. Test Specification

8.1 Load Test (Concurrency)

- **Tool**: `wrk -t12 -c400 -d30s http://localhost:8080/index.html`
- **Assertion**: No 500 errors. Success rate 100%. Throughput > 5000 req/sec (on standard hardware).

8.2 Slowloris Defense

- **Action**: Open 50 connections with `nc`, send `GET /`, then do nothing.
- **Assertion**: After 30s, server must close all 50 connections. Server remains responsive to new `curl` requests during this time.

8.3 Data Race Check

- **Tool**: `gcc -fsanitize=thread`
- **Action**: Run the server under load with thread sanitizer.
- **Assertion**: Zero reports of "Data race" or "Unlock of unheld mutex".

---

9. Performance Targets

Operation	Target	Measurement
Lock Contention	< 5% CPU	`perf record` monitoring `pthread_mutex_lock`.
Keep-Alive Handshake Savings	~1.2ms / req	Comparison of 100 seq reqs with vs without Keep-Alive.
Context Switch Overhead	< 2µs / task	`latency-top` or `perf stat` (cs).
FD Leakage	0.00%	Baseline FD count after 100k requests.

---

10. Hardware Soul: Concurrency & Physics

10.1 Cache Line Bouncing (MESI Protocol)

When thread A modifies `pool->queue_size`, the CPU core invalidates that cache line in the L1/L2 caches of all other cores. When thread B tries to read `pool->queue_size`, it experiences a cache miss and must fetch the line from L3. This is the "cost of synchronization." By holding the lock only for the few nanoseconds required to increment a pointer, we minimize the window where cores are fighting over the same cache lines.

10.2 Atomic Instructions

The `pthread_mutex_lock` is implemented using hardware atomic instructions (e.g., `LOCK CMPXCHG` on x86). This is a "bus lock" or "cache lock" that ensures no other processor can modify the memory address during the operation. This takes ~~20-50ns. If the lock is already held, the kernel puts the thread to sleep (~~`futex`~~ syscall), which is much more expensive (~~2-5µs).

{{DIAGRAM:tdd-diag-23|The Futex Path|Sequence: Lock Attempt -> Fail -> Kernel Sleep -> Signal -> Wakeup}}

---

11. Concurrency Specification: Lock Ordering

To prevent **Deadlock**, the following hierarchy is enforced:

1. `pool->lock` (The primary gatekeeper).
2. `log_mutex` (The access log lock).

Illegal Pattern: Holding `log_mutex` and then attempting to acquire `pool->lock`. **Mandatory Pattern**: Always acquire `pool->lock` first, then `log_mutex`.

Note: Worker threads currently do not need to hold both simultaneously. They acquire `pool->lock` to update request stats, release it, then acquire `log_mutex` to write the access log line.

---

12. Implementation Path (C Details)

12.1 The 503 Overload Response

```c
static void send_503(int client_fd) {
    const char *msg = "HTTP/1.1 503 Service Unavailable\r\n"
                      "Content-Type: text/html\r\n"
                      "Content-Length: 42\r\n"
                      "Connection: close\r\n\r\n"
                      "<h1>503</h1><p>Server Queue Full</p>";
    write_all(client_fd, msg, strlen(msg));
}
```

12.2 Signal Masking Snippet

```c
// MUST do this before spawning ANY threads
sigset_t set;
sigfillset(&set);
pthread_sigmask(SIG_BLOCK, &set, NULL);
// Worker threads will inherit this mask and ignore SIGINT/SIGTERM,
// allowing our dedicated thread to catch them via sigwait().
```

12.3 Robust Dequeue

```c
// Inside worker loop

pthread_mutex_lock(&pool->lock);

while(pool->queue_size == 0 && !pool->shutdown) {

    pthread_cond_wait(&pool->not_empty, &pool->lock);

}

// Re-check after waking (Spurious Wakeup handling)

if (pool->shutdown && pool->queue_size == 0) {

    pthread_mutex_unlock(&pool->lock);

    pthread_exit(NULL);

}
```

Project Structure: HTTP Server (Basic)

Directory Tree

MARKDOWN

```
http-server/
├── include/                # Header files (Definitions & Interfaces)
│   ├── concurrency_types.h # M4: Thread pool & work structs
│   ├── date_utils.h        # M3: RFC 1123 date formatting
│   ├── file_handler.h      # M3: Static file serving logic
│   ├── http_errors.h       # M2: Error response templates
│   ├── http_parser.h       # M2: Request parsing prototypes
│   ├── http_types.h        # M2: HTTP enums and request struct
│   ├── mime_types.h        # M3: Extension-to-MIME mapping table
│   ├── path_utils.h        # M3: URL decoding & path safety
│   ├── server.h            # M1: Core socket structures
│   ├── thread_pool.h       # M4: Pool management functions
│   ├── connection_handler.h # M4: Connection lifecycle orchestration
│   └── signals.h           # M4: Signal handling thread interface
├── src/                    # Source files (Implementations)
│   ├── connection_handler.c # M4: Keep-alive loop & timeout logic
│   ├── date_utils.c        # M3: timegm and strftime helpers
│   ├── file_handler.c      # M3: 64KB stream I/O loop
│   ├── http.c              # M1: Hardcoded response builder
│   ├── http_errors.c       # M2: send_error_response implementation
│   ├── http_parser.c       # M2: Line-based state machine
│   ├── main.c              # M1/M4: Entry point & orchestrator
│   ├── path_utils.c        # M3: realpath() & traversal checks
│   ├── server.c            # M1: Socket dance (bind/listen)
│   ├── signals.c           # M4: Synchronous sigwait() thread
```

```
│   ├── thread_pool.c       # M4: Worker loops & sync primitives
│   └── utils.c             # M1: write_all & partial read logic
├── build/                   # Compiled binaries (Build Artifact)
├── Makefile                 # Build system (Build Artifact)
├── .gitignore               # Version control exclusions
└── README.md               # Project overview and testing guide
```

Creation Order

1. **Project Scaffolding** (30 min)
    - Create `include/`, `src/`, and `build/` directories.
    - Set up `Makefile` with `-pthread` and `-Wall -Wextra` flags.
    - Initialize `server.h` with `server_config_t`.
2. **Milestone 1: The Socket Foundation** (2 hours)
    - Implement `src/utils.c` (`read_http_request`, `write_all`).
    - Implement `src/server.c` (`socket_init`, `bind`, `listen`).
    - Implement `src/http.c` (hardcoded response).
    - Create basic `src/main.c` accept loop.
3. **Milestone 2: Protocol Parsing** (3 hours)
    - Define `include/http_types.h` (`http_request_t`).
    - Implement `src/http_parser.c` (`parse_request_line`, `parse_header_line`).
    - Implement `src/http_errors.c` (`send_400`, `send_414`, `send_501`).
    - Integrate parser into the connection handler.
4. **Milestone 3: Payload Delivery & Security** (4 hours)
    - Implement `src/path_utils.c` (`url_decode`, `resolve_safe_path`).
    - Implement `src/date_utils.c` (RFC 1123 UTC formatting).
    - Define `include/mime_types.h` table.
    - Implement `src/file_handler.c` (The 8-phase serving orchestrator).
5. **Milestone 4: High Concurrency & Hardening** (4 hours)
    - Implement `src/thread_pool.c` (Mutex/CondVar work queue).
    - Implement `src/signals.c` (Dedicated signal thread logic).

    - Refactor `src/connection_handler.c` to include the Keep-Alive loop.
    - Update `src/main.c` to initialize pool and mask signals.

File Count Summary

- **Total Header Files**: 12
- **Total Source Files**: 12
- **Build/Config Files**: 3
- **Total Directories**: 3 (root, include, src)
- **Estimated Lines of Code**: ~1,450 LOC (Implementation logic + Comments)