microhttpd Roadmap & Checklist (Step-by-Step)

October 16, 2025

Overview

A practical, incremental path to build the multithreaded HTTP/1.1 file server. Each phase ends with a clear "Definition of Done" and small test commands you can run immediately.

1 Phase 0 — Workspace Bootstrapping (30–45 min)

0.1 Repo layout

```
microhttpd/
  Makefile
  src/
    main.c
    http_parse.c  http_parse.h
    fs.c          fs.h
    log.c         log.h
```

0.2 Makefile (minimal): compile all `.c` in `src/` with `-std=c11 -Wall -Wextra -O2`.

0.3 Main flags: parse `-p`, `-d`. Default: port 8080, root ".".

0.4 Open listen socket: `socket, setsockopt(SO_REUSEADDR), bind, listen`.

0.5 Definition of Done: binary runs and prints "listening on 0.0.0.0:8080".

2 Phase 1 — Single-Connection Echo (15–30 min)

1.1 Accept one client: `accept4(..., SOCK_CLOEXEC)`.

1.2 Blocking read/write loop: read bytes, immediately write back (temporary).

1.3 Definition of Done: `nc 127.0.0.1 8080` echoes back text.

3 Phase 2 — Minimal HTTP Parser + 200 OK (1–2 h)

2.1 Parser skeleton (`http_parse.c`): parse request line (METHOD, TARGET, VERSION), then headers until CRLFCRLF.

2.2 Limits: max start-line 4KiB, headers 16KiB; reject on overflow (400).

2.3 Routing: if METHOD not GET/HEAD → 405; if VERSION != HTTP/1.1 → 400.

2.4 Respond 200: always return a tiny "Hello" page to confirm formatting.

2.5 Definition of Done: `curl -v http://127.0.0.1:8080/` shows 200 and your page.

4 Phase 3 — Map Target to Files (1–2 h)

3.1 Path normalization (`fs.c`): percent-decode safe chars, reject "..", build absolute path under ROOT, `realpath()` both ROOT and target; ensure target has ROOT prefix.

3.2 Directories: if directory and missing trailing slash → 301 redirect to "/.../".

3.3 Index: if directory and has `index.html`, serve it; else generate simple HTML listing.

3.4 MIME: basic table by extension, default `application/octet-stream`.

3.5 HEAD: same headers as GET, zero-length body.

3.6 Definition of Done: static files (`.html/.txt/.png`) served; bad paths give 404; dir redirect works.

5 Phase 4 — Keep-Alive + Connection Loop (45–60 min)

4.1 Connection loop: after finishing a response, attempt to parse the next request from the same socket.

4.2 Timeouts: set `SO_RCVTIMEO` or use `poll()` with 5s idle limit.

4.3 Connection header: default persistent; close if `Connection: close` is present or on parse error.

4.4 Cap: optional max 100 requests/connection.

4.5 Definition of Done: `curl -v --keepalive-time 2 http://127.0.0.1:8080/` reuses TCP.

6 Phase 5 — Thread Pool & Work Queue (2–3 h)

5.1 Introduce workq

```
struct job { int client_fd; struct sockaddr_storage peer; };

struct workq {
  struct job *ring; size_t cap, head, tail, count;
  pthread_mutex_t mtx;
  pthread_cond_t not_empty, not_full;
};
```

5.2 APIs: `workq_init(cap)`, `enqueue(job)`, `dequeue()`, `workq_destroy()`.

5.3 Acceptor thread: `accept4()` then `enqueue()` (blocks when full → natural backpressure).

5.4 Workers (N): `dequeue()` and run the full connection lifecycle (keep-alive loop).

5.5 Per-thread buffers: allocate request buffer & path buffer once per thread to avoid malloc churn.

5.6 Definition of Done: with `-w N`, multiple concurrent cURLs are served in parallel.

7 Phase 6 — Logging, Errors, Hardening (1–2 h)

6.1 Logging: one line per request (ISO timestamp, peer IP, method, target, status, bytes, ms, user-agent).

6.2 Error pages: small HTML bodies for 400/404/405/413/500.

6.3 Resource limits: header size cap; idle timeout; sanitize directory listings (HTML-escape).

6.4 Definition of Done: malformed requests yield 400 with a body; logs look consistent under load.

8 Phase 7 — Functional Tests (ongoing, 30–60 min)

- `curl -I /` (HEAD), `curl -v /does-not-exist` (404), nested dirs, large files.
- Long URLs to test limit handling; paths with encoded slashes, spaces, unicode.
- Symlink inside root; symlink escape attempt (must fail).

9 Phase 8 — Load & Stability Tests (1–2 h)

- **Concurrency**: many small files: `for i in {1..200}; do curl -s http://127.0.0.1:8080/ & done; wait`
- **Backpressure**: temporarily set small queue capacity; ensure acceptor blocks when full.
- **File descriptors**: `ulimit -n 256`; observe graceful handling near limits.

10 Phase 9 — Polish (optional, 1–3 h)

- **Directory listing UX**: size, mtime, simple CSS inline.
- **Conditional**: `Last-Modified` and simple `If-Modified-Since` handling.
- **Range (206)**: single-range support for resumable downloads.

Mini Implementation Order (Files & Functions)

1. `main.c`: parse flags, open socket, single accept/handle.
2. `http_parse.c`: `parse_request()`, `get_header()`, limits.
3. `fs.c`: `safe_join_and_realpath()`, `is_dir()`, `serve_file()`, `render_listing()`.
4. `log.c`: `log_request()`, `ts_iso8601()`.
5. Thread pool: `workq.c`, `acceptor.c`, `worker.c`.

Quick Reference Snippets

Socket Setup (IPv4 any)

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
int one = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
struct sockaddr_in sa = {0};
sa.sin_family = AF_INET; sa.sin_port = htons(port); sa.sin_addr.s_addr = htonl
    (INADDR_ANY);
bind(fd, (struct sockaddr*)&sa, sizeof(sa));
listen(fd, 128);
```

Worker Loop Skeleton

```
for (;;) {
  ssize_t n = recv(client, buf + used, sizeof(buf)-used, 0);
  if (n <= 0) break;
  used += n;
  int parsed = parse_request(buf, used, &req);
  if (parsed < 0) { send_400(client); break; }
  if (parsed == 0) continue; // need more bytes
  handle_request(client, &req); // may read nothing from body in this server
  compact_or_reset_buffer(buf, &used, parsed); // keepalive: keep unread bytes
}
```

Debugging Tips

- Run with `strace -f -e trace=network,read,write ./microhttpd ...` to verify I/O.
- Print parser state transitions for tricky requests; fuzz with random headers.
- Watch `/proc/$PID/fd` to catch descriptor leaks while load testing.

Definition of "MVP Done"

- Serves files/dirs under ROOT with GET/HEAD, redirects dir without trailing "/".
- Persistent connections; times out idle clients; bounded memory and descriptors.
- Thread pool serves multiple clients concurrently; logs each request.