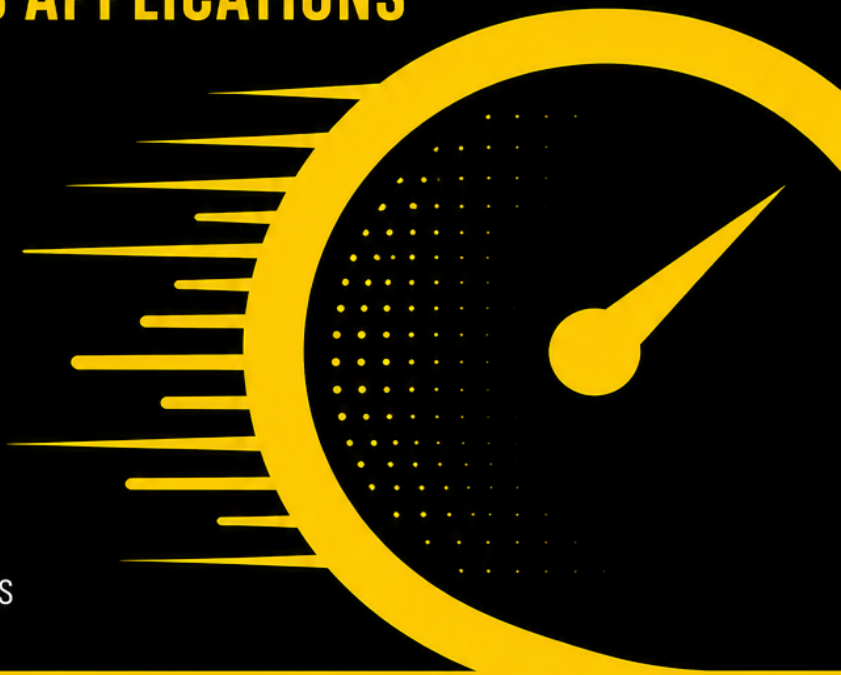
LIBUV MASTERY

BUILDING HIGH-PERFORMANCE ASYNCHRONOUS APPLICATIONS IN C

- EVENT-DRIVEN ARCHITECTURE
- CROSS-PLATFORM COMPATIBILITY
- SCALABLE PERFORMANCE
- PRACTICAL EXAMPLES REAL-WORLD PROJECTS



C

MIKHAIL SOKOLOV

Libuv Mastery

Building High-Performance Asynchronous Applications in C

Steve Publications

This book is available at https://leanpub.com/libuvmastery

This version was published on 2026-08-03

Contents

Building High-Performance Asynchronous Applications in C **1**

Introduction **2**
What You Will Learn 2
Who This Book Is For 3
How This Book Is Structured 3
A Note on Code Examples 3

Chapter 1: The World of Asynchronous I/O **5**
The C10K Problem and Beyond 5
Concurrency Models Compared 6
What libuv Is and Where It Lives 8
Design Goals of libuv 9
How to Read This Book 10

Chapter 2: Getting Started with libuv **12**
Obtaining libuv 12
Building Your First Project with CMake 13
The Minimal Event Loop Program 16
Understanding the Build Output 19
Verifying Your Installation 20

Chapter 3: Architecture of libuv **22**
The Platform Abstraction Layer 22
Handles vs Requests 22
The Event Loop Internals 22
Memory Allocation in libuv 22
Thread Safety Rules 22

Chapter 4: The Event Loop **23**
Anatomy of uv_run() 23

Loop Lifecycle 23
Custom Event Loops 23
Embedding Loops 23
Common Pitfalls 23

Chapter 5: Asynchronous I/O Fundamentals 24
The Callback Pattern 24
Error Handling in libuv 24
Handles Lifecycle 24
The Stream Abstraction 24
Resource Cleanup Patterns 24

Chapter 6: Timers and Periodic Operations 25
One-Shot Timers with uv_timer_t 25
Repeating Timers 25
Periodic Handles: Prepare and Check 25
Time Management in libuv 25
Building a Task Scheduler 25

Chapter 7: File System Operations 26
Synchronous vs Asynchronous File I/O 26
Reading and Writing Files 26
Directory Operations 26
The Thread Pool Behind fs Operations 26
File Descriptors and Ownership 26

Chapter 8: TCP Networking 27
TCP Servers 27
TCP Clients 27
Stream Operations on TCP 27
Advanced Socket Options 27
Building a Concurrent Echo Server 27

Chapter 9: UDP and DNS Resolution 28
UDP Sockets 28
Multicast Support 28
Async DNS with getaddrinfo 28
Reverse DNS and Hostname Resolution 28
Building a Simple UDP Chat Server 28

Chapter 10: Pipes, Streams, and TTY **29**
Unix Domain Sockets and Named Pipes 29
The Unified Stream Interface 29
Duplex Streams 29
TTY Support 29
Building an IPC Bridge 29

Chapter 11: Process Management and Signals **30**
Spawning Processes 30
Stdio Inheritance Patterns 30
Process Events 30
Signal Handling 30
Building a Process Supervisor 30

Chapter 12: Threading and Synchronization **31**
The Thread Pool 31
Creating Custom Threads 31
Mutexes and Locks 31
Condition Variables 31
Atomic Operations and Thread-Local Storage 31

Chapter 13: Polling and Custom File Descriptors **32**
The Poll Handle 32
Event Types and Their Meanings 32
Integrating External Libraries 32
File Descriptor Ownership 32
Building a Hybrid Reactor 32

Chapter 14: Error Handling, Debugging, and Testing **33**
Systematic Error Handling 33
Debugging libuv Applications 33
Memory Leak Detection 33
Testing Asynchronous Code 33
Core Dumps and Post-Mortem Analysis 33

Chapter 15: Performance Optimization **34**
Reducing Latency 34
Throughput Optimization 34
CPU Affinity and Multiple Loops 34
Profiling libuv Applications 34

Tuning for Production 34

Chapter 16: Portability Across Operating Systems 35
Platform Detection in libuv 35
Linux-Specific Behavior 35
macOS and BSD Considerations 35
Windows Portability 35
Writing Truly Portable Code 35

Chapter 17: Embedding libuv and Integration 36
Embedding libuv in Existing Applications 36
Integrating with SSL/TLS Libraries 36
Building on Top of libuv 36
Interfacing with C++ and Other Languages 36
Microservice Architecture with libuv 36

Chapter 18: Real-World Applications 37
Building an HTTP Server from Scratch 37
A WebSocket Implementation 37
A Load Balancer 37
Deployment Considerations 37

Conclusion: The Art of Event-Driven Programming 38
Patterns That Scale 38
When NOT to Use libuv 38
The Future of Async C 38
Continuing Your Journey 38

References 39

Building High-Performance Asynchronous Applications in C

This book takes you from basic C programming to advanced expertise with libuv, the cross-platform asynchronous I/O library that powers Node.js and countless other projects. Through clear explanations, complete working examples, and deep architectural insight, you will learn how to build scalable event-driven applications that handle thousands of concurrent connections efficiently on Linux, macOS, Windows, and beyond. No prior experience with asynchronous programming is required, only a solid foundation in C.

Introduction

In 2009, Ryan Dahl stood on stage at JSConf EU and delivered a presentation that would reshape server-side software development. He argued that the dominant model for building network servers was fundamentally broken: one thread per connection simply could not scale to thousands of simultaneous clients without exhausting system resources [1]. His solution combined Google's V8 JavaScript engine with an event loop built on libuv, creating Node.js and popularizing a programming style that would eventually influence virtually every major language and framework.

The library at the heart of this revolution was not glamorous or widely known outside certain circles. libuv is a C library, quiet and unassuming in its API design, yet responsible for powering some of the most important software on the internet. Node.js uses it. Deno uses it. Julia uses it. Redis has used variants of its design patterns. If you have ever benefited from a server handling thousands of concurrent users with a single process, chances are libuv was involved somewhere in the stack [2].

This book exists because libuv deserves more than cursory treatment. Most developers who use Node.js or other libuv-based runtimes never see the library directly. They interact with JavaScript abstractions that hide the complexity underneath. But for C programmers, and for anyone who wants to understand how asynchronous I/O actually works under the hood, libuv is an essential technology to master.

What You Will Learn

By the end of this book you will be able to:

- Design and implement event-driven applications in C that scale to thousands of concurrent connections
- Understand every major libuv API, including TCP and UDP networking, file system operations, process management, signals, threading, and IPC
- Debug and profile libuv applications using standard tools like valgrind, strace, and GDB

- Write portable code that works correctly on Linux, macOS, Windows, FreeBSD, and other supported platforms
- Integrate libuv into existing projects or build new systems from the ground up with production-grade reliability

The journey begins with fundamentals. We will start by understanding why asynchronous I/O matters, what problems it solves, and how different concurrency models compare. Then we will install libuv, write our first programs, and gradually build toward complex real-world applications like HTTP servers, load balancers, and process supervisors.

Who This Book Is For

This book assumes you know C at a working level: you understand pointers, structs, memory management, and basic system calls. You do not need prior experience with asynchronous programming, networking, or event loops. If you have written blocking socket code in C before, this book will show you how to reimplement that code without threads-per-connection overhead while maintaining readability and correctness.

If you are a Node.js developer curious about what happens beneath the JavaScript layer, this book will give you that understanding. If you are a systems programmer looking for a portable way to write high-performance networked software, libuv provides exactly that capability.

How This Book Is Structured

The chapters progress from basic concepts to advanced techniques. The first few chapters establish foundation: what libuv is, how to install it, how the event loop works at a conceptual level. Middle chapters dive into specific APIs one by one, each with complete working examples you can compile and run. Later chapters cover threading, debugging, performance optimization, portability concerns, and architectural patterns for production systems.

The final chapters bring everything together with real-world projects: building an HTTP server from scratch using only libuv primitives, implementing a WebSocket handler, creating a load balancer that distributes traffic across backend servers. These are not toy examples. They are complete applications written in the style you should use for your own production code.

A Note on Code Examples

Every code example in this book is complete and self-contained. You will never see ellipses where important logic was omitted, never encounter a "fill in the rest yourself" situation. Each program compiles with standard C compilers and links against libuv without modification. Build instructions accompany every example, including full CMakeLists.txt files for projects that use CMake.

Code is explained thoroughly. Every significant line has surrounding prose that describes what it does and why it is written that way. This is not a reference manual where you look up API signatures and infer usage from context. It is a guided learning experience that walks you through the reasoning behind every design decision.

Let us begin.

Chapter 1: The World of Asynchronous I/O

The C10K Problem and Beyond

In the year 2004, a paper titled "The C10K problem" was published that would define a challenge for network programmers for decades to come. The problem statement was simple: how do you build a server that can handle ten thousand concurrent connections on commodity hardware [3]? At the time, this seemed ambitious. Most web servers of the era handled hundreds or maybe a few thousand connections before performance degraded dramatically.

The fundamental issue lay in how operating systems managed threads and file descriptors. The dominant server architecture was straightforward: for each incoming client connection, spawn a new thread to handle that client's requests. This model is intuitive and easy to reason about. Each thread runs independently, blocking on I/O operations without affecting other clients.

But threads are expensive. On typical systems of the early 2000s, each thread consumed at least 1 megabyte of stack space plus kernel data structures. Ten thousand threads would require roughly 10 GB of memory just for stacks, far beyond what commodity servers could provide. Even with smaller stacks, the context switching overhead between thousands of threads degraded performance severely. The CPU spent more time switching between threads than actually processing requests [4].

The solution that emerged was not to abandon concurrency but to reimagine it. Instead of one thread per connection, use a single thread that manages many connections simultaneously. This is the event-driven model, and it relies on a fundamental shift in how programs interact with I/O operations.

In a blocking model, when your code calls read() on a socket, the calling thread stops executing until data arrives. In an asynchronous or non-blocking model, the call returns immediately with whatever data is available right now, or with a signal that no data is ready yet. The program then does something else

productive while waiting, and comes back to handle the data when it actually arrives.

This approach reduces memory usage dramatically because you need only one or a few threads regardless of how many connections exist. Each connection requires minimal bookkeeping: a socket file descriptor, some buffers, and a callback function to invoke when events occur. Ten thousand connections might consume a few megabytes instead of ten gigabytes [5].

The C10K problem has long been "solved" in the sense that modern servers routinely handle far more than ten thousand concurrent connections. But the underlying challenge persists in new forms. Today's applications face the C100K problem, or even higher. Real-time features, WebSockets, microservice architectures with thousands of internal connections, IoT platforms managing millions of devices: all of these require efficient handling of massive connection counts [6].

Concurrency Models Compared

Understanding why libuv matters requires understanding the landscape of concurrency models. Let us examine the major approaches and their trade-offs.

The threads-per-connection model is the simplest to understand. When a client connects, you create a new thread that handles all communication with that client using blocking I/O calls:

```
// Simplified threads-per-connection server
void *handle_client(void *arg) {
    int client_fd = *(int *)arg;
    free(arg);

    char buffer[4096];
    ssize_t n;
    while ((n = read(client_fd, buffer, sizeof(buffer))) > 0) {
        write(client_fd, buffer, n); // Echo back
    }
    close(client_fd);
    return NULL;
}

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

```
    // bind(server_fd), listen(server_fd)...

    while (1) {
        int client_fd = accept(server_fd, NULL, NULL);
        pthread_t thread;
        int *fd_ptr = malloc(sizeof(int));
        *fd_ptr = client_fd;
        pthread_create(&thread, NULL, handle_client, fd_ptr);
        pthread_detach(thread);
    }
}
```

This code is readable and straightforward. Each client handler runs in isolation. But as we discussed, it does not scale. Memory usage grows linearly with connection count, and context switching overhead becomes prohibitive at high concurrency levels.

The select/poll model improves on this by using a single thread that monitors multiple file descriptors for activity. The select() system call blocks until at least one of the monitored descriptors is ready for I/O:

```
// Simplified select-based server
fd_set read_fds;
FD_SET(server_fd, &read_fds);

while (1) {
    FD_ZERO(&read_fds);
    FD_SET(server_fd, &read_fds);
    // Add client fds to read_fds...

    select(max_fd + 1, &read_fds, NULL, NULL, NULL);

    if (FD_ISSET(server_fd, &read_fds)) {
        int client_fd = accept(server_fd, NULL, NULL);
        // Add client_fd to monitored set
    }
    // Read from ready client fds...
}
```

This approach uses constant memory regardless of connection count. But select() has limitations: it caps at FD_SETMAX file descriptors (often 1024), requires scanning the entire fd set on each call, and modifies the sets in place requiring reinitialization. The poll() system call improves slightly by removing the fd limit, but still requires linear scanning of all monitored descriptors [7].

Modern Linux introduced epoll, which addresses these inefficiencies. Instead of scanning all file descriptors, epoll maintains an internal data structure that tracks only active events. When you call epoll_wait(), it returns immediately with a list of ready descriptors without requiring you to pass the entire set each time:

```
// Simplified epoll-based server
int epfd = epoll_create1(0);
struct epoll_event event;
event.events = EPOLLIN;
event.data.fd = server_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &event);

while (1) {
    struct epoll_event events[MAX_EVENTS];
    int n = epoll_wait(epfd, events, MAX_EVENTS, -1);

    for (int i = 0; i < n; i++) {
        if (events[i].data.fd == server_fd) {
            // Accept new connection
        } else {
            // Handle ready client
        }
    }
}
```

epoll scales efficiently to hundreds of thousands of connections. But this is Linux-specific. Other platforms have their own mechanisms: kqueue on BSD and macOS, IOCP on Windows, event ports on Solaris. Each has different semantics, different APIs, different quirks [8].

This is where libuv enters the picture. It provides a single, consistent API that works across all these platforms underneath. You write your code once using libuv's abstraction, and libuv uses epoll on Linux, kqueue on macOS, IOCP on Windows, and so forth. The same source compiles and runs efficiently everywhere.

What libuv Is and Where It Lives

libuv is a multi-platform C library focused on asynchronous I/O. Its primary design goal is to provide a portable event loop that abstracts away platform-specific I/O mechanisms while maintaining high performance [9]. The library

was originally developed for Node.js by Ben Noordhuis and others, starting around 2011 when the Node.js team needed Windows support. At that time, Node.js used libev on Unix systems, but libev did not support Windows. Rather than write a custom Windows backend for Node.js directly, the team extracted the I/O layer into its own library: libuv [10].

Today libuv is maintained independently and used by many projects beyond Node.js:

- **Node.js**: The runtime that popularized JavaScript on the server side uses libuv for all non-blocking I/O, file system operations, DNS resolution, and child process management.
- **Deno**: Ryan Dahl's successor to Node.js also uses libuv underneath its TypeScript/JavaScript runtime.
- **Julia**: The high-performance numerical computing language uses libuv for asynchronous I/O capabilities.
- **uvloop**: A fast implementation of Python's asyncio event loop written in Cython that wraps libuv directly.
- **Luvit**: A Lua toolkit built on libuv for concurrent networking applications.
- **Various C/C++ projects**: Countless smaller projects use libuv as their networking and async I/O foundation.

The library is released under the MIT license, making it freely usable in both open-source and proprietary software. It follows semantic versioning since version 1.0.0, guaranteeing backward compatibility within major versions [11].

Design Goals of libuv

Understanding libuv's design goals helps explain many API decisions that might otherwise seem arbitrary. The library was shaped by several core principles:

Portability first. libuv must work on Linux, macOS, Windows, FreeBSD, OpenBSD, and other platforms with identical behavior. This is not an afterthought or a secondary concern. It is the primary constraint that drives every design decision. If an API cannot be implemented consistently across all target platforms, it does not go into libuv [12].

Simplicity of use. The API should be straightforward for experienced C programmers. Conceptual models like handles versus requests are kept

minimal and consistent. Error handling follows predictable patterns: negative return values indicate errors, callbacks receive status parameters that can be checked.

Performance without compromise. Abstraction must not come at the cost of performance. libuv uses the best available mechanism on each platform: epoll with edge-triggered mode on Linux, kqueue on BSD systems, IOCP on Windows. It avoids unnecessary copying and allocations in hot paths [13].

Single-threaded event loop. The core event loop is designed to run on a single thread. This simplifies reasoning about concurrent access: callbacks execute one at a time, eliminating data races within the callback layer. Thread safety concerns are confined to well-defined boundaries where worker threads interact with the main loop [14].

Non-blocking by default. Every I/O operation in libuv is designed around non-blocking semantics. Even operations that cannot truly be non-blocking on the underlying platform (like file system operations) are made asynchronous through a thread pool so they never block the event loop [15].

These design goals sometimes conflict. Portability can limit what features are available. Single-threaded design requires careful handling of CPU-bound work. But the resulting trade-offs produce a library that is reliable, efficient, and genuinely cross-platform.

How to Read This Book

This book follows a deliberate learning progression. We start with installation and basic concepts, then gradually introduce more complex topics. Each chapter builds on previous ones, so reading in order is recommended.

Chapters 1 through 4 establish foundation: why asynchronous I/O matters, how libuv is structured internally, how the event loop works. Without this understanding, the APIs will feel like disconnected functions to memorize. With it, they become tools that make sense within a coherent architecture.

Chapters 5 through 13 cover individual API areas: timers, file system operations, networking (TCP, UDP, DNS), pipes and IPC, process management, signals, threading, polling. Each chapter includes complete working examples you can compile, run, modify, and study. These are the core chapters where you will spend most of your time.

Chapters 14 through 17 address production concerns: debugging, performance optimization, cross-platform portability, embedding libuv in larger systems. These topics matter when you move from learning examples to real applications that must be reliable, fast, and maintainable.

Chapter 18 brings everything together with substantial projects: a complete HTTP server built from libuv primitives, a WebSocket implementation, a load balancer. These demonstrate how all the individual pieces fit together in realistic scenarios.

You can approach this book in two ways. The first is linear reading: work through every chapter sequentially, compiling and running every example as you go. This is the recommended approach for thorough learning. The second is reference-style: skim the early chapters for context, then jump to specific API chapters relevant to your immediate needs, returning to foundational material when concepts are unclear.

Either way, the key to mastering libuv is hands-on practice. Read the explanations, but more importantly write code. Modify the examples. Break things and fix them. Build something of your own. That is how you move from understanding libuv intellectually to using it fluently in production systems.

Chapter 2: Getting Started with libuv

Obtaining libuv

Before writing any code, you need libuv installed on your system. There are several approaches depending on your platform and preferences.

On Linux distributions, libuv is available through package managers. On Debian and Ubuntu systems, install it with apt:

```
sudo apt-get install libuv1-dev
```

This installs both the development headers and the shared library. On Fedora and RHEL derivatives:

```
sudo dnf install libuv-devel
```

On Arch Linux:

```
sudo pacman -S libuv
```

On macOS, Homebrew provides libuv:

```
brew install libuv
```

On Windows, you can use vcpkg or Conan as package managers, or build from source using CMake. The vcpkg approach is straightforward:

```
vcpkg install libuv
```

For full control over the version and build configuration, building from source is always an option. Download the source from GitHub:

```
git clone https://github.com/libuv/libuv.git
cd libuv
git checkout v1.52.1  # Or the latest stable tag
```

On Unix-like systems, you can build using autotools or CMake:

```
# Using autotools (Unix only)
./autogen.sh
./configure
make -j$(nproc)
sudo make install

# Using CMake (cross-platform)
mkdir build && cd build
cmake ..
cmake --build .
sudo cmake --install .
```

On Windows, only CMake is supported:

```
mkdir build && cd build
cmake .. -G "Visual Studio 17 2022" -A x64
cmake --build . --config Release
cmake --install . --config Release
```

The choice between static and dynamic linking affects how you distribute your application. With dynamic linking, the libuv shared library must be present on the target system. With static linking, libuv code is embedded directly into your executable, eliminating the runtime dependency but increasing binary size. For development, dynamic linking is usually simpler. For deployment to systems where you cannot guarantee libuv installation, static linking provides reliability [16].

Building Your First Project with CMake

CMake has become the standard build system for C and C++ projects using libuv. It handles platform differences automatically, finds installed libraries, and generates appropriate build files for your compiler.

Create a project directory and set up the basic structure:

```
mkdir libuv-hello
cd libuv-hello
mkdir src
```

Create a minimal CMakeLists.txt file in the project root:

```
cmake_minimum_required(VERSION 3.16)
project(libuv_hello C)

set(CMAKE_C_STANDARD 11)
set(CMAKE_C_STANDARD_REQUIRED ON)

# Find libuv
find_package(uv REQUIRED)

# Create executable
add_executable(hello src/main.c)

# Link against libuv
target_link_libraries(hello PRIVATE uv_a)
```

The find_package(uv REQUIRED) command searches for libuv on your system. When you installed libuv via package manager or built from source with cmake –install, CMake can locate it automatically. The target name uv_a refers to the static library variant. If you prefer dynamic linking, use uv_shared instead:

```
target_link_libraries(hello PRIVATE uv_shared)
```

Now create src/main.c with a minimal program that initializes libuv and runs the event loop for exactly one iteration without doing any work:

```c
#include <stdio.h>
#include <uv.h>

int main(void) {
    /* Create and initialize an event loop */
    uv_loop_t* loop = uv_default_loop();

    if (loop == NULL) {
        fprintf(stderr, "Failed to create event loop\n");
        return 1;
    }

    printf("Event loop created successfully.\n");
    printf("Loop address: %p\n", (void*)loop);

    /* Run the event loop once with no blocking */
    int result = uv_run(loop, UV_RUN_NOWAIT);

    printf("uv_run returned: %d\n", result);

    /* Clean up the loop */
    if (uv_loop_close(loop) != 0) {
        fprintf(stderr, "Failed to close event loop. "
                "There may be open handles.\n");
        return 1;
    }

    printf("Event loop closed. Program exiting.\n");
    return 0;
}
```

Let us examine this code carefully. The uv_default_loop() function creates and returns a pointer to the default event loop. This is a convenience function that Node.js and other single-loop applications use. Under the hood, it calls uv_loop_init() on a statically allocated uv_loop_t structure [17].

The loop can be NULL only if memory allocation fails internally, which is extremely rare but worth checking. The printf statement showing the loop address confirms we have a valid pointer.

The uv_run() function executes the event loop. We pass UV_RUN_NOWAIT as the mode, which means: poll for events once but do not block if no events are pending. Since we have not registered any handles or requests with the loop, there is nothing to process. The function returns 0 immediately, indicating the loop has no active work [18].

Finally, uv_loop_close() releases all resources held by the loop. It returns

an error code if there are still open handles that have not been closed. This is a safety check: forgetting to close handles before closing the loop indicates a resource leak. In this minimal example we know there are no open handles, so the call succeeds and prints 0.

Build and run the project:

```
mkdir build && cd build
cmake ..
cmake --build .
./hello
```

Expected output:

```
Event loop created successfully.
Loop address: 0x55a3b2c1d800
uv_run returned: 0
Event loop closed. Program exiting.
```

If you see this output, libuv is installed correctly and your build environment is configured properly.

The Minimal Event Loop Program

Our first example did nothing useful. Let us write a program that actually uses the event loop by scheduling a timer that fires after one second. This demonstrates the fundamental pattern of libuv programming: initialize handles, register callbacks, run the loop, let callbacks execute when events occur.

Create a new file src/timer_hello.c:

```c
#include <stdio.h>
#include <uv.h>

/* Timer callback invoked when the timeout expires */
static void on_timer(uv_timer_t* handle) {
    printf("Timer fired! One second has passed.\n");

    /* Stop the timer so it does not fire again */
    uv_timer_stop(handle);

    /* Stop the event loop after this callback completes */
    uv_stop(handle->loop);
}

int main(void) {
    uv_loop_t* loop = uv_default_loop();
    if (loop == NULL) {
        fprintf(stderr, "Failed to create event loop\n");
        return 1;
    }

    /* Create a timer handle */
    uv_timer_t timer;

    /* Initialize the timer with the loop */
    int r = uv_timer_init(loop, &timer);
    if (r != 0) {
        fprintf(stderr, "uv_timer_init failed: %s\n", uv_strerror(r));
        return 1;
    }

    printf("Timer initialized. Starting one-shot timer for 1000ms...\n");

    /* Start the timer: callback, timeout in ms, repeat interval (0 = no
    ↪  repeat) */
    r = uv_timer_start(&timer, on_timer, 1000, 0);
    if (r != 0) {
        fprintf(stderr, "uv_timer_start failed: %s\n", uv_strerror(r));
        return 1;
    }

    printf("Running event loop. Waiting for timer...\n");

    /* Run the event loop until uv_stop() is called */
    r = uv_run(loop, UV_RUN_DEFAULT);
    if (r != 0) {
        fprintf(stderr, "uv_run failed: %s\n", uv_strerror(r));
        return 1;
    }

```

```
    printf("Event loop stopped. Cleaning up.\n");

    /* Close the loop (timer was already stopped by the callback) */
    if (uv_loop_close(loop) != 0) {
        fprintf(stderr, "Failed to close event loop\n");
        return 1;
    }

    printf("Done.\n");
    return 0;
}
```

This program demonstrates the core libuv pattern. We create a timer handle using uv_timer_init(), which registers the handle with the event loop but does not activate it yet. Then we call uv_timer_start() to configure when the timer fires: after 1000 milliseconds, with no repetition (the last parameter being 0).

When the timer expires, libuv invokes our on_timer callback from within the event loop thread. Inside the callback we stop the timer and then call uv_stop() to request that the loop exit. The loop does not exit immediately: it finishes processing any remaining callbacks from this iteration, then returns from uv_run(). This ensures clean shutdown without leaving operations in an inconsistent state [19].

Update CMakeLists.txt to build this example instead:

```
cmake_minimum_required(VERSION 3.16)
project(libuv_timer_hello C)

set(CMAKE_C_STANDARD 11)
set(CMAKE_C_STANDARD_REQUIRED ON)

find_package(uv REQUIRED)

add_executable(timer_hello src/timer_hello.c)
target_link_libraries(timer_hello PRIVATE uv_a)
```

Build and run:

```
cd build
cmake ..
cmake --build .
./timer_hello
```

Output:

```
Timer initialized. Starting one-shot timer for 1000ms...
Running event loop. Waiting for timer...
Timer fired! One second has passed.
Event loop stopped. Cleaning up.
Done.
```

Notice that between "Waiting for timer" and "Timer fired" approximately one second elapses. During that time, the uv_run() call is blocking, but it is not busy-waiting. It is sleeping efficiently, using the operating system's event notification mechanism (epoll_wait on Linux, kevent on macOS, GetQueuedCompletionStatus on Windows) to wake up only when the timer expires or another event occurs [20].

Understanding the Build Output

When you compile libuv programs, several flags and linking details matter for correctness and debugging. Let us examine what happens when CMake builds our project.

The key compiler flags include:

- **-std=c11**: Uses the C11 standard, which provides features like anonymous structs and improved thread support that modern code relies on.
- **-Wall -Wextra**: Enables comprehensive warnings. libuv's own build uses aggressive warning flags, and you should too to catch potential issues early.
- **-g**: Includes debug symbols for GDB and other debugging tools. Always include this during development.

For linking, the critical component is the libuv library itself. On most Unix systems, the linker also needs to connect with platform-specific libraries that libuv uses internally:

- On Linux: pthread (for threading), rt (for high-resolution timers)
- On macOS: CoreFoundation and CoreServices (for kqueue and other system facilities)
- On Windows: ws2_32, mswsock, userenv (for sockets and I/O completion ports)

When you use CMake's find_package(uv REQUIRED), these dependencies are handled automatically. If you compile manually with gcc, you would need to specify them explicitly. A typical manual compilation command on Linux looks like:

```
gcc -std=c11 -Wall -g src/timer_hello.c -luv -lpthread -lrt -o timer_hello
```

On macOS:

```
gcc -std=c11 -Wall -g src/timer_hello.c -luv -framework CoreServices -o
↪  timer_hello
```

CMake abstracts these platform differences, which is one reason it is the recommended build approach.

Verifying Your Installation

Before moving on to more complex examples, confirm your libuv installation is working correctly with a few checks.

First, verify the installed version:

```
# On systems with pkg-config
pkg-config --modversion libuv
```

This should output the version number, such as 1.52.1.

Second, check that headers are accessible:

```
# Find the uv.h header
pkg-config --cflags libuv
```

This outputs include paths like -I/usr/include.

Third, confirm linking works by compiling a small test program directly:

```
echo '#include <uv.h>
int main(void) {
    return uv_version();
}' > test_uv.c && \
gcc test_uv.c $(pkg-config --cflags --libs libuv) -o test_uv && \
./test_uv && echo "OK" || echo "FAIL"
```

If all three checks pass, your environment is ready for the rest of this book. If any check fails, revisit the installation steps for your platform. Common issues include missing development packages (libuv1-dev on Debian/Ubuntu, libuv-devel on Fedora) or the library being installed in a non-standard location that CMake cannot find.

In the next chapter we will dive into libuv's internal architecture: how handles and requests work, how the event loop processes events, how platform abstraction enables portability. Understanding this architecture makes every subsequent API easier to learn because you will see the patterns that connect them all together.

Chapter 3: Architecture of libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Platform Abstraction Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Handles vs Requests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Event Loop Internals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Memory Allocation in libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Thread Safety Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 4: The Event Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Anatomy of uv_run()

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Loop Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Custom Event Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Embedding Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Common Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 5: Asynchronous I/O Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Callback Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Error Handling in libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Handles Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Stream Abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Resource Cleanup Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 6: Timers and Periodic Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

One-Shot Timers with uv_timer_t

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Repeating Timers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Periodic Handles: Prepare and Check

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Time Management in libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building a Task Scheduler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 7: File System Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Synchronous vs Asynchronous File I/O

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Reading and Writing Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Directory Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Thread Pool Behind fs Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

File Descriptors and Ownership

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 8: TCP Networking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

TCP Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

TCP Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Stream Operations on TCP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Advanced Socket Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building a Concurrent Echo Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 9: UDP and DNS Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

UDP Sockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Multicast Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Async DNS with getaddrinfo

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Reverse DNS and Hostname Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building a Simple UDP Chat Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 10: Pipes, Streams, and TTY

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Unix Domain Sockets and Named Pipes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Unified Stream Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Duplex Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

TTY Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building an IPC Bridge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 11: Process Management and Signals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Spawning Processes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Stdio Inheritance Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Process Events

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Signal Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building a Process Supervisor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 12: Threading and Synchronization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Thread Pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Creating Custom Threads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Mutexes and Locks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Condition Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Atomic Operations and Thread-Local Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 13: Polling and Custom File Descriptors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Poll Handle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Event Types and Their Meanings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Integrating External Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

File Descriptor Ownership

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building a Hybrid Reactor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 14: Error Handling, Debugging, and Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Systematic Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Debugging libuv Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Memory Leak Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Testing Asynchronous Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Core Dumps and Post-Mortem Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 15: Performance Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Reducing Latency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Throughput Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

CPU Affinity and Multiple Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Profiling libuv Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Tuning for Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 16: Portability Across Operating Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Platform Detection in libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Linux-Specific Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

macOS and BSD Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Windows Portability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Writing Truly Portable Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 17: Embedding libuv and Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Embedding libuv in Existing Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Integrating with SSL/TLS Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building on Top of libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Interfacing with C++ and Other Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Microservice Architecture with libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Chapter 18: Real-World Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Building an HTTP Server from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

A WebSocket Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

A Load Balancer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Deployment Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Conclusion: The Art of Event-Driven Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Patterns That Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

When NOT to Use libuv

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

The Future of Async C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

Continuing Your Journey

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/libuvmastery.