An Introduction to libuv

Release 2.0.0

Nikhil Marathe

December 01, 2015

CHAPTER

ONE

INTRODUCTION

This 'book' is a small set of tutorials about using libuv as a high performance evented I/O library which offers the same API on Windows and Unix.

It is meant to cover the main areas of libuv, but is not a comprehensive reference discussing every function and data structure. The official libuv documentation may be consulted for full details.

This book is still a work in progress, so sections may be incomplete, but I hope you will enjoy it as it grows.

1.1 Who this book is for

If you are reading this book, you are either:

1. a systems programmer, creating low-level programs such as daemons or network services and clients. You have found that the event loop approach is well suited for your application and decided to use libuv.
2. a node.js module writer, who wants to wrap platform APIs written in C or C++ with a set of (a)synchronous APIs that are exposed to JavaScript. You will use libuv purely in the context of node.js. For this you will require some other resources as the book does not cover parts specific to v8/node.js.

This book assumes that you are comfortable with the C programming language.

1.2 Background

The node.js project began in 2009 as a JavaScript environment decoupled from the browser. Using Google's V8 and Marc Lehmann's libev, node.js combined a model of I/O – evented – with a language that was well suited to the style of programming; due to the way it had been shaped by browsers. As node.js grew in popularity, it was important to make it work on Windows, but libev ran only on Unix. The Windows equivalent of kernel event notification mechanisms like kqueue or (e)poll is IOCP. libuv was an abstraction around libev or IOCP depending on the platform, providing users an API based on libev. In the node-v0.9.0 version of libuv libev was removed.

Since then libuv has continued to mature and become a high quality standalone library for system programming. Users outside of node.js include Mozilla's Rust programming language, and a variety of language bindings.

This book and the code is based on libuv version v1.3.0.

1.3 Code

All the code from this book is included as part of the source of the book on Github. Clone/Download the book, then build libuv:

```
cd libuv
./autogen.sh
./configure
make
```

There is no need to `make install`. To build the examples run `make` in the `code/` directory.

CHAPTER

TWO

BASICS OF LIBUV

libuv enforces an **asynchronous**, **event-driven** style of programming. Its core job is to provide an event loop and callback based notifications of I/O and other activities. libuv offers core utilities like timers, non-blocking networking support, asynchronous file system access, child processes and more.

2.1 Event loops

In event-driven programming, an application expresses interest in certain events and respond to them when they occur. The responsibility of gathering events from the operating system or monitoring other sources of events is handled by libuv, and the user can register callbacks to be invoked when an event occurs. The event-loop usually keeps running *forever*. In pseudocode:

```
while there are still events to process:
    e = get the next event
    if there is a callback associated with e:
        call the callback
```

Some examples of events are:

- File is ready for writing
- A socket has data ready to be read
- A timer has timed out

This event loop is encapsulated by `uv_run()` – the end-all function when using libuv.

The most common activity of systems programs is to deal with input and output, rather than a lot of number-crunching. The problem with using conventional input/output functions (`read`, `fprintf`, etc.) is that they are **blocking**. The actual write to a hard disk or reading from a network, takes a disproportionately long time compared to the speed of the processor. The functions don't return until the task is done, so that your program is doing nothing. For programs which require high performance this is a major roadblock as other activities and other I/O operations are kept waiting.

One of the standard solutions is to use threads. Each blocking I/O operation is started in a separate thread (or in a thread pool). When the blocking function gets invoked in the thread, the processor can schedule another thread to run, which actually needs the CPU.

The approach followed by libuv uses another style, which is the **asynchronous, non-blocking** style. Most modern operating systems provide event notification subsystems. For example, a normal `read` call on a socket would block until the sender actually sent something. Instead, the application can request the operating system to watch the socket and put an event notification in the queue. The application can inspect the events at its convenience (perhaps doing some number crunching before to use the processor to the maximum) and grab the data. It is **asynchronous** because the application expressed interest at one point, then used the data at another point (in time and space). It is **non-blocking** because the application process was free to do other tasks. This fits in well with libuv's event-loop approach, since the operating system events can be treated as just another libuv event. The non-blocking ensures that other events can continue to be handled as fast as they come in [1].

Note: How the I/O is run in the background is not of our concern, but due to the way our computer hardware works, with the thread as the basic unit of the processor, libuv and OSes will usually run background/worker threads and/or polling to perform tasks in a non-blocking manner.

Bert Belder, one of the libuv core developers has a small video explaining the architecture of libuv and its background. If you have no prior experience with either libuv or libev, it is a quick, useful watch.

libuv's event loop is explained in more detail in the documentation.

2.2 Hello World

With the basics out of the way, lets write our first libuv program. It does nothing, except start a loop which will exit immediately.

helloworld/main.c

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

int main() {
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    uv_loop_init(loop);

    printf("Now quitting.\n");
    uv_run(loop, UV_RUN_DEFAULT);

    uv_loop_close(loop);
    free(loop);
    return 0;
}
```

This program quits immediately because it has no events to process. A libuv event loop has to be told to watch out for events using the various API functions.

Starting with libuv v1.0, users should allocate the memory for the loops before initializing it with `uv_loop_init(uv_loop_t *)`. This allows you to plug in custom memory management. Remember to de-initialize the loop using `uv_loop_close(uv_loop_t *)` and then delete the storage. The examples never close loops since the program quits after the loop ends and the system will reclaim memory. Production grade projects, especially long running systems programs, should take care to release correctly.

2.2.1 Default loop

A default loop is provided by libuv and can be accessed using `uv_default_loop()`. You should use this loop if you only want a single loop.

Note: node.js uses the default loop as its main loop. If you are writing bindings you should be aware of this.

[1] Depending on the capacity of the hardware of course.

2.3 Error handling

Initialization functions or synchronous functions which may fail return a negative number on error. Async functions that may fail will pass a status parameter to their callbacks. The error messages are defined as UV_E* constants.

You can use the uv_strerror(int) and uv_err_name(int) functions to get a const char * describing the error or the error name respectively.

I/O read callbacks (such as for files and sockets) are passed a parameter nread. If nread is less than 0, there was an error (UV_EOF is the end of file error, which you may want to handle differently).

2.4 Handles and Requests

libuv works by the user expressing interest in particular events. This is usually done by creating a **handle** to an I/O device, timer or process. Handles are opaque structs named as uv_TYPE_t where type signifies what the handle is used for.

libuv watchers

```
/* Handle types. */
typedef struct uv_loop_s uv_loop_t;
typedef struct uv_handle_s uv_handle_t;
typedef struct uv_stream_s uv_stream_t;
typedef struct uv_tcp_s uv_tcp_t;
typedef struct uv_udp_s uv_udp_t;
typedef struct uv_pipe_s uv_pipe_t;
typedef struct uv_tty_s uv_tty_t;
typedef struct uv_poll_s uv_poll_t;
typedef struct uv_timer_s uv_timer_t;
typedef struct uv_prepare_s uv_prepare_t;
typedef struct uv_check_s uv_check_t;
typedef struct uv_idle_s uv_idle_t;
typedef struct uv_async_s uv_async_t;
typedef struct uv_process_s uv_process_t;
typedef struct uv_fs_event_s uv_fs_event_t;
typedef struct uv_fs_poll_s uv_fs_poll_t;
typedef struct uv_signal_s uv_signal_t;

/* Request types. */
typedef struct uv_req_s uv_req_t;
typedef struct uv_getaddrinfo_s uv_getaddrinfo_t;
typedef struct uv_getnameinfo_s uv_getnameinfo_t;
typedef struct uv_shutdown_s uv_shutdown_t;
typedef struct uv_write_s uv_write_t;
typedef struct uv_connect_s uv_connect_t;
typedef struct uv_udp_send_s uv_udp_send_t;
typedef struct uv_fs_s uv_fs_t;
typedef struct uv_work_s uv_work_t;

/* None of the above. */
typedef struct uv_cpu_info_s uv_cpu_info_t;
typedef struct uv_interface_address_s uv_interface_address_t;
typedef struct uv_dirent_s uv_dirent_t;
```

Handles represent long-lived objects. Async operations on such handles are identified using **requests**. A request is short-lived (usually used across only one callback) and usually indicates one I/O operation on a handle. Requests are used to preserve context between the initiation and the callback of individual actions. For example, an UDP socket is represented by a `uv_udp_t`, while individual writes to the socket use a `uv_udp_send_t` structure that is passed to the callback after the write is done.

Handles are setup by a corresponding:

```
uv_TYPE_init(uv_loop_t *, uv_TYPE_t *)
```

function.

Callbacks are functions which are called by libuv whenever an event the watcher is interested in has taken place. Application specific logic will usually be implemented in the callback. For example, an IO watcher's callback will receive the data read from a file, a timer callback will be triggered on timeout and so on.

2.4.1 Idling

Here is an example of using an idle handle. The callback is called once on every turn of the event loop. A use case for idle handles is discussed in Utilities. Let us use an idle watcher to look at the watcher life cycle and see how `uv_run()` will now block because a watcher is present. The idle watcher is stopped when the count is reached and `uv_run()` exits since no event watchers are active.

idle-basic/main.c

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

int64_t counter = 0;

void wait_for_a_while(uv_idle_t* handle) {
    counter++;

    if (counter >= 10e6)
        uv_idle_stop(handle);
}

int main() {
    uv_idle_t idler;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, wait_for_a_while);

    printf("Idling...\n");
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    uv_loop_close(uv_default_loop());
    return 0;
}
```

2.4.2 Storing context

In callback based programming style you'll often want to pass some 'context' – application specific information – between the call site and the callback. All handles and requests have a `void* data` member which you can set to the context and cast back in the callback. This is a common pattern used throughout the C library ecosystem. In addition `uv_loop_t` also has a similar data member.

CHAPTER

THREE

FILESYSTEM

Simple filesystem read/write is achieved using the `uv_fs_*` functions and the `uv_fs_t` struct.

Note: The libuv filesystem operations are different from socket operations. Socket operations use the non-blocking operations provided by the operating system. Filesystem operations use blocking functions internally, but invoke these functions in a thread pool and notify watchers registered with the event loop when application interaction is required.

All filesystem functions have two forms - *synchronous* and *asynchronous*.

The *synchronous* forms automatically get called (and **block**) if the callback is null. The return value of functions is a *libuv error code*. This is usually only useful for synchronous calls. The *asynchronous* form is called when a callback is passed and the return value is 0.

3.1 Reading/Writing files

A file descriptor is obtained using

```
int uv_fs_open(uv_loop_t* loop, uv_fs_t* req, const char* path, int flags, int mode, uv_fs_cb cb)
```

`flags` and `mode` are standard Unix flags. libuv takes care of converting to the appropriate Windows flags.

File descriptors are closed using

```
int uv_fs_close(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
```

Filesystem operation callbacks have the signature:

```
void callback(uv_fs_t* req);
```

Let's see a simple implementation of `cat`. We start with registering a callback for when the file is opened:

uvcat/main.c - opening a file

```
    // The request passed to the callback is the same as the one the call setup
    // function was passed.
    assert(req == &open_req);
    if (req->result >= 0) {
        iov = uv_buf_init(buffer, sizeof(buffer));
        uv_fs_read(uv_default_loop(), &read_req, req->result,
                   &iov, 1, -1, on_read);
    }
    else {
```

```
        fprintf(stderr, "error opening file: %s\n", uv_strerror((int)req->result));
    }
}
```

The `result` field of a `uv_fs_t` is the file descriptor in case of the `uv_fs_open` callback. If the file is successfully opened, we start reading it.

uvcat/main.c - read callback

```
void on_read(uv_fs_t *req) {
    if (req->result < 0) {
        fprintf(stderr, "Read error: %s\n", uv_strerror(req->result));
    }
    else if (req->result == 0) {
        uv_fs_t close_req;
        // synchronous
        uv_fs_close(uv_default_loop(), &close_req, open_req.result, NULL);
    }
    else if (req->result > 0) {
        iov.len = req->result;
        uv_fs_write(uv_default_loop(), &write_req, 1, &iov, 1, -1, on_write);
    }
}
```

In the case of a read call, you should pass an *initialized* buffer which will be filled with data before the read callback is triggered. The `uv_fs_*` operations map almost directly to certain POSIX functions, so EOF is indicated in this case by `result` being 0. In the case of streams or pipes, the `UV_EOF` constant would have been passed as a status instead.

Here you see a common pattern when writing asynchronous programs. The `uv_fs_close()` call is performed synchronously. *Usually tasks which are one-off, or are done as part of the startup or shutdown stage are performed synchronously, since we are interested in fast I/O when the program is going about its primary task and dealing with multiple I/O sources*. For solo tasks the performance difference usually is negligible and may lead to simpler code.

Filesystem writing is similarly simple using `uv_fs_write()`. *Your callback will be triggered after the write is complete*. In our case the callback simply drives the next read. Thus read and write proceed in lockstep via callbacks.

uvcat/main.c - write callback

```
void on_write(uv_fs_t *req) {
    if (req->result < 0) {
        fprintf(stderr, "Write error: %s\n", uv_strerror((int)req->result));
    }
    else {
        uv_fs_read(uv_default_loop(), &read_req, open_req.result, &iov, 1, -1, on_read);
    }
}
```

> **Warning:** Due to the way filesystems and disk drives are configured for performance, a write that 'succeeds' may not be committed to disk yet.

We set the dominos rolling in `main()`:

uvcat/main.c

```
int main(int argc, char **argv) {
    uv_fs_open(uv_default_loop(), &open_req, argv[1], O_RDONLY, 0, on_open);
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    uv_fs_req_cleanup(&open_req);
    uv_fs_req_cleanup(&read_req);
    uv_fs_req_cleanup(&write_req);
    return 0;
}
```

> **Warning:** The `uv_fs_req_cleanup()` function must always be called on filesystem requests to free internal memory allocations in libuv.

3.2 Filesystem operations

All the standard filesystem operations like `unlink`, `rmdir`, `stat` are supported asynchronously and have intuitive argument order. They follow the same patterns as the read/write/open calls, returning the result in the `uv_fs_t.result` field. The full list:

Filesystem operations

```
UV_EXTERN int uv_fs_close(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_open(uv_loop_t* loop,
                         uv_fs_t* req,
                         const char* path,
                         int flags,
                         int mode,
                         uv_fs_cb cb);
UV_EXTERN int uv_fs_read(uv_loop_t* loop,
                         uv_fs_t* req,
                         uv_file file,
                         const uv_buf_t bufs[],
                         unsigned int nbufs,
                         int64_t offset,
                         uv_fs_cb cb);
UV_EXTERN int uv_fs_unlink(uv_loop_t* loop,
                           uv_fs_t* req,
                           const char* path,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_write(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          const uv_buf_t bufs[],
                          unsigned int nbufs,
                          int64_t offset,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_mkdir(uv_loop_t* loop,
                          uv_fs_t* req,
```

```
                          const char* path,
                          int mode,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_mkdtemp(uv_loop_t* loop,
                            uv_fs_t* req,
                            const char* tpl,
                            uv_fs_cb cb);
UV_EXTERN int uv_fs_rmdir(uv_loop_t* loop,
                          uv_fs_t* req,
                          const char* path,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_scandir(uv_loop_t* loop,
                            uv_fs_t* req,
                            const char* path,
                            int flags,
                            uv_fs_cb cb);
UV_EXTERN int uv_fs_scandir_next(uv_fs_t* req,
                                 uv_dirent_t* ent);
UV_EXTERN int uv_fs_stat(uv_loop_t* loop,
                         uv_fs_t* req,
                         const char* path,
                         uv_fs_cb cb);
UV_EXTERN int uv_fs_fstat(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_rename(uv_loop_t* loop,
                           uv_fs_t* req,
                           const char* path,
                           const char* new_path,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_fsync(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_fdatasync(uv_loop_t* loop,
                              uv_fs_t* req,
                              uv_file file,
                              uv_fs_cb cb);
UV_EXTERN int uv_fs_ftruncate(uv_loop_t* loop,
                              uv_fs_t* req,
                              uv_file file,
                              int64_t offset,
                              uv_fs_cb cb);
UV_EXTERN int uv_fs_sendfile(uv_loop_t* loop,
                             uv_fs_t* req,
                             uv_file out_fd,
                             uv_file in_fd,
                             int64_t in_offset,
                             size_t length,
                             uv_fs_cb cb);
UV_EXTERN int uv_fs_access(uv_loop_t* loop,
                           uv_fs_t* req,
                           const char* path,
                           int mode,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_chmod(uv_loop_t* loop,
                          uv_fs_t* req,
```

```
                             const char* path,
                             int mode,
                             uv_fs_cb cb);
UV_EXTERN int uv_fs_utime(uv_loop_t* loop,
                             uv_fs_t* req,
                             const char* path,
                             double atime,
                             double mtime,
                             uv_fs_cb cb);
UV_EXTERN int uv_fs_futime(uv_loop_t* loop,
                              uv_fs_t* req,
                              uv_file file,
                              double atime,
                              double mtime,
                              uv_fs_cb cb);
UV_EXTERN int uv_fs_lstat(uv_loop_t* loop,
                             uv_fs_t* req,
                             const char* path,
                             uv_fs_cb cb);
UV_EXTERN int uv_fs_link(uv_loop_t* loop,
                            uv_fs_t* req,
                            const char* path,
                            const char* new_path,
                            uv_fs_cb cb);
```

3.3 Buffers and Streams

The basic I/O handle in libuv is the stream (`uv_stream_t`). TCP sockets, UDP sockets, and pipes for file I/O and IPC are all treated as stream subclasses.

Streams are initialized using custom functions for each subclass, then operated upon using

```
int uv_read_start(uv_stream_t*, uv_alloc_cb alloc_cb, uv_read_cb read_cb);
int uv_read_stop(uv_stream_t*);
int uv_write(uv_write_t* req, uv_stream_t* handle,
             const uv_buf_t bufs[], unsigned int nbufs, uv_write_cb cb);
```

The stream based functions are simpler to use than the filesystem ones and libuv will automatically keep reading from a stream when `uv_read_start()` is called once, until `uv_read_stop()` is called.

The discrete unit of data is the buffer – `uv_buf_t`. This is simply a collection of a pointer to bytes (`uv_buf_t.base`) and the length (`uv_buf_t.len`). The `uv_buf_t` is lightweight and passed around by value. What does require management is the actual bytes, which have to be allocated and freed by the application.

Error: THIS PROGRAM DOES NOT ALWAYS WORK, NEED SOMETHING BETTER**

To demonstrate streams we will need to use `uv_pipe_t`. This allows streaming local files [1]. Here is a simple tee utility using libuv. Doing all operations asynchronously shows the power of evented I/O. The two writes won't block each other, but we have to be careful to copy over the buffer data to ensure we don't free a buffer until it has been written.

The program is to be executed as:

[1] see *Pipes*

```
./uvtee <output_file>
```

We start off opening pipes on the files we require. libuv pipes to a file are opened as bidirectional by default.

uvtee/main.c - read on pipes

```c
int main(int argc, char **argv) {
    loop = uv_default_loop();

    uv_pipe_init(loop, &stdin_pipe, 0);
    uv_pipe_open(&stdin_pipe, 0);

    uv_pipe_init(loop, &stdout_pipe, 0);
    uv_pipe_open(&stdout_pipe, 1);

    uv_fs_t file_req;
    int fd = uv_fs_open(loop, &file_req, argv[1], O_CREAT | O_RDWR, 0644, NULL);
    uv_pipe_init(loop, &file_pipe, 0);
    uv_pipe_open(&file_pipe, fd);

    uv_read_start((uv_stream_t*)&stdin_pipe, alloc_buffer, read_stdin);

    uv_run(loop, UV_RUN_DEFAULT);
    return 0;
}
```

The third argument of `uv_pipe_init()` should be set to 1 for IPC using named pipes. This is covered in Processes. The `uv_pipe_open()` call associates the pipe with the file descriptor, in this case `0` (standard input).

We start monitoring `stdin`. The `alloc_buffer` callback is invoked as new buffers are required to hold incoming data. `read_stdin` will be called with these buffers.

uvtee/main.c - reading buffers

```c
void alloc_buffer(uv_handle_t *handle, size_t suggested_size, uv_buf_t *buf) {
    *buf = uv_buf_init((char*) malloc(suggested_size), suggested_size);
}

void read_stdin(uv_stream_t *stream, ssize_t nread, const uv_buf_t *buf) {
    if (nread < 0){
        if (nread == UV_EOF){
            // end of file
            uv_close((uv_handle_t *)&stdin_pipe, NULL);
            uv_close((uv_handle_t *)&stdout_pipe, NULL);
            uv_close((uv_handle_t *)&file_pipe, NULL);
        }
    } else if (nread > 0) {
        write_data((uv_stream_t *)&stdout_pipe, nread, *buf, on_stdout_write);
        write_data((uv_stream_t *)&file_pipe, nread, *buf, on_file_write);
    }

    if (buf->base)
        free(buf->base);
}
```

The standard `malloc` is sufficient here, but you can use any memory allocation scheme. For example, node.js uses its own slab allocator which associates buffers with V8 objects.

The read callback `nread` parameter is less than 0 on any error. This error might be EOF, in which case we close all the streams, using the generic close function `uv_close()` which deals with the handle based on its internal type. Otherwise `nread` is a non-negative number and we can attempt to write that many bytes to the output streams. Finally remember that buffer allocation and deallocation is application responsibility, so we free the data.

The allocation callback may return a buffer with length zero if it fails to allocate memory. In this case, the read callback is invoked with error UV_ENOBUFS. libuv will continue to attempt to read the stream though, so you must explicitly call `uv_close()` if you want to stop when allocation fails.

The read callback may be called with `nread = 0`, indicating that at this point there is nothing to be read. Most applications will just ignore this.

uvtee/main.c - Write to pipe

```
typedef struct {
    uv_write_t req;
    uv_buf_t buf;
} write_req_t;

void free_write_req(uv_write_t *req) {
    write_req_t *wr = (write_req_t*) req;
    free(wr->buf.base);
    free(wr);
}

void on_stdout_write(uv_write_t *req, int status) {
    free_write_req(req);
}

void on_file_write(uv_write_t *req, int status) {
    free_write_req(req);
}

void write_data(uv_stream_t *dest, size_t size, uv_buf_t buf, uv_write_cb cb) {
    write_req_t *req = (write_req_t*) malloc(sizeof(write_req_t));
    req->buf = uv_buf_init((char*) malloc(size), size);
    memcpy(req->buf.base, buf.base, size);
    uv_write((uv_write_t*) req, (uv_stream_t*)dest, &req->buf, 1, cb);
}
```

`write_data()` makes a copy of the buffer obtained from read. This buffer does not get passed through to the write callback trigged on write completion. To get around this we wrap a write request and a buffer in `write_req_t` and unwrap it in the callbacks. We make a copy so we can free the two buffers from the two calls to `write_data` independently of each other. While acceptable for a demo program like this, you'll probably want smarter memory management, like reference counted buffers or a pool of buffers in any major application.

Warning: If your program is meant to be used with other programs it may knowingly or unknowingly be writing to a pipe. This makes it susceptible to aborting on receiving a SIGPIPE. It is a good idea to insert:

```
signal(SIGPIPE, SIG_IGN)
```

in the initialization stages of your application.

3.4 File change events

All modern operating systems provide APIs to put watches on individual files or directories and be informed when the files are modified. libuv wraps common file change notification libraries [2]. This is one of the more inconsistent parts of libuv. File change notification systems are themselves extremely varied across platforms so getting everything working everywhere is difficult. To demonstrate, I'm going to build a simple utility which runs a command whenever any of the watched files change:

```
./onchange <command> <file1> [file2] ...
```

The file change notification is started using `uv_fs_event_init()`:

onchange/main.c - The setup

```c
int main(int argc, char **argv) {
    if (argc <= 2) {
        fprintf(stderr, "Usage: %s <command> <file1> [file2 ...]\n", argv[0]);
        return 1;
    }

    loop = uv_default_loop();
    command = argv[1];

    while (argc-- > 2) {
        fprintf(stderr, "Adding watch on %s\n", argv[argc]);
        uv_fs_event_t *fs_event_req = malloc(sizeof(uv_fs_event_t));
        uv_fs_event_init(loop, fs_event_req);
        // The recursive flag watches subdirectories too.
        uv_fs_event_start(fs_event_req, run_command, argv[argc], UV_FS_EVENT_RECURSIVE);
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The third argument is the actual file or directory to monitor. The last argument, `flags`, can be:

```
UV_FS_EVENT_WATCH_ENTRY = 1,
UV_FS_EVENT_STAT = 2,
UV_FS_EVENT_RECURSIVE = 4
```

`UV_FS_EVENT_WATCH_ENTRY` and `UV_FS_EVENT_STAT` don't do anything (yet). `UV_FS_EVENT_RECURSIVE` will start watching subdirectories as well on supported platforms.

The callback will receive the following arguments:

1. `uv_fs_event_t *handle` - The handle. The `path` field of the handle is the file on which the watch was set.
2. `const char *filename` - If a directory is being monitored, this is the file which was changed. Only non-`null` on Linux and Windows. May be `null` even on those platforms.
3. **`int flags` - one of `UV_RENAME` or `UV_CHANGE`, or a bitwise OR of** both.
4. `int status` - Currently 0.

In our example we simply print the arguments and run the command using `system()`.

[2] inotify on Linux, FSEvents on Darwin, kqueue on BSDs, ReadDirectoryChangesW on Windows, event ports on Solaris, unsupported on Cygwin

onchange/main.c - file change notification callback

```c
void run_command(uv_fs_event_t *handle, const char *filename, int events, int status) {
    char path[1024];
    size_t size = 1023;
    // Does not handle error if path is longer than 1023.
    uv_fs_event_getpath(handle, path, &size);
    path[size] = '\0';

    fprintf(stderr, "Change detected in %s: ", path);
    if (events & UV_RENAME)
        fprintf(stderr, "renamed");
    if (events & UV_CHANGE)
        fprintf(stderr, "changed");

    fprintf(stderr, " %s\n", filename ? filename : "");
    system(command);
}
```

CHAPTER

FOUR

NETWORKING

Networking in libuv is not much different from directly using the BSD socket interface, some things are easier, all are non-blocking, but the concepts stay the same. In addition libuv offers utility functions to abstract the annoying, repetitive and low-level tasks like setting up sockets using the BSD socket structures, DNS lookup, and tweaking various socket parameters.

The `uv_tcp_t` and `uv_udp_t` structures are used for network I/O.

4.1 TCP

TCP is a connection oriented, stream protocol and is therefore based on the libuv streams infrastructure.

4.1.1 Server

Server sockets proceed by:

1. `uv_tcp_init` the TCP handle.
2. `uv_tcp_bind` it.
3. Call `uv_listen` on the handle to have a callback invoked whenever a new connection is established by a client.
4. Use `uv_accept` to accept the connection.
5. Use *stream operations* to communicate with the client.

Here is a simple echo server

tcp-echo-server/main.c - The listen socket

```
int main() {
    loop = uv_default_loop();

    uv_tcp_t server;
    uv_tcp_init(loop, &server);

    uv_ip4_addr("0.0.0.0", DEFAULT_PORT, &addr);

    uv_tcp_bind(&server, (const struct sockaddr*)&addr, 0);
    int r = uv_listen((uv_stream_t*) &server, DEFAULT_BACKLOG, on_new_connection);
    if (r) {
        fprintf(stderr, "Listen error %s\n", uv_strerror(r));
```

```
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

You can see the utility function `uv_ip4_addr` being used to convert from a human readable IP address, port pair to the sockaddr_in structure required by the BSD socket APIs. The reverse can be obtained using `uv_ip4_name`.

Note: There are `uv_ip6_*` analogues for the ip4 functions.

Most of the setup functions are synchronous since they are CPU-bound. `uv_listen` is where we return to libuv's callback style. The second arguments is the backlog queue – the maximum length of queued connections.

When a connection is initiated by clients, the callback is required to set up a handle for the client socket and associate the handle using `uv_accept`. In this case we also establish interest in reading from this stream.

tcp-echo-server/main.c - Accepting the client

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status < 0) {
        fprintf(stderr, "New connection error %s\n", uv_strerror(status));
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

The remaining set of functions is very similar to the streams example and can be found in the code. Just remember to call `uv_close` when the socket isn't required. This can be done even in the `uv_listen` callback if you are not interested in accepting the connection.

4.1.2 Client

Where you do bind/listen/accept on the server, on the client side it's simply a matter of calling `uv_tcp_connect`. The same `uv_connect_cb` style callback of `uv_listen` is used by `uv_tcp_connect`. Try:

```
uv_tcp_t* socket = (uv_tcp_t*)malloc(sizeof(uv_tcp_t));
uv_tcp_init(loop, socket);

uv_connect_t* connect = (uv_connect_t*)malloc(sizeof(uv_connect_t));

struct sockaddr_in dest;
uv_ip4_addr("127.0.0.1", 80, &dest);

uv_tcp_connect(connect, socket, (const struct sockaddr*)&dest, on_connect);
```

where `on_connect` will be called after the connection is established. The callback receives the `uv_connect_t` struct, which has a member `.handle` pointing to the socket.

4.2 UDP

The User Datagram Protocol offers connectionless, unreliable network communication. Hence libuv doesn't offer a stream. Instead libuv provides non-blocking UDP support via the *uv_udp_t* handle (for receiving) and *uv_udp_send_t* request (for sending) and related functions. That said, the actual API for reading/writing is very similar to normal stream reads. To look at how UDP can be used, the example shows the first stage of obtaining an IP address from a DHCP server – DHCP Discover.

Note: You will have to run *udp-dhcp* as **root** since it uses well known port numbers below 1024.

udp-dhcp/main.c - Setup and send UDP packets

```
uv_loop_t *loop;
uv_udp_t send_socket;
uv_udp_t recv_socket;

int main() {
    loop = uv_default_loop();

    uv_udp_init(loop, &recv_socket);
    struct sockaddr_in recv_addr;
    uv_ip4_addr("0.0.0.0", 68, &recv_addr);
    uv_udp_bind(&recv_socket, (const struct sockaddr *)&recv_addr, UV_UDP_REUSEADDR);
    uv_udp_recv_start(&recv_socket, alloc_buffer, on_read);

    uv_udp_init(loop, &send_socket);
    struct sockaddr_in broadcast_addr;
    uv_ip4_addr("0.0.0.0", 0, &broadcast_addr);
    uv_udp_bind(&send_socket, (const struct sockaddr *)&broadcast_addr, 0);
    uv_udp_set_broadcast(&send_socket, 1);

    uv_udp_send_t send_req;
    uv_buf_t discover_msg = make_discover_msg();

    struct sockaddr_in send_addr;
    uv_ip4_addr("255.255.255.255", 67, &send_addr);
    uv_udp_send(&send_req, &send_socket, &discover_msg, 1, (const struct sockaddr *)&send_addr, on_se

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

Note: The IP address `0.0.0.0` is used to bind to all interfaces. The IP address `255.255.255.255` is a broadcast address meaning that packets will be sent to all interfaces on the subnet. port `0` means that the OS randomly assigns a port.

First we setup the receiving socket to bind on all interfaces on port 68 (DHCP client) and start a read on it. This will read back responses from any DHCP server that replies. We use the UV_UDP_REUSEADDR flag to play nice with any other system DHCP clients that are running on this computer on the same port. Then we setup a similar send socket and use `uv_udp_send` to send a *broadcast message* on port 67 (DHCP server).

It is **necessary** to set the broadcast flag, otherwise you will get an EACCES error [1]. The exact message being sent is not relevant to this book and you can study the code if you are interested. As usual the read and write callbacks will receive a status code of < 0 if something went wrong.

Since UDP sockets are not connected to a particular peer, the read callback receives an extra parameter about the sender of the packet.

nread may be zero if there is no more data to be read. If addr is NULL, it indicates there is nothing to read (the callback shouldn't do anything), if not NULL, it indicates that an empty datagram was received from the host at addr. The flags parameter may be UV_UDP_PARTIAL if the buffer provided by your allocator was not large enough to hold the data. *In this case the OS will discard the data that could not fit* (That's UDP for you!).

udp-dhcp/main.c - Reading packets

```
void on_read(uv_udp_t *req, ssize_t nread, const uv_buf_t *buf, const struct sockaddr *addr, unsigned
    if (nread < 0) {
        fprintf(stderr, "Read error %s\n", uv_err_name(nread));
        uv_close((uv_handle_t*) req, NULL);
        free(buf->base);
        return;
    }

    char sender[17] = { 0 };
    uv_ip4_name((const struct sockaddr_in*) addr, sender, 16);
    fprintf(stderr, "Recv from %s\n", sender);

    // ... DHCP specific code
    unsigned int *as_integer = (unsigned int*)buf->base;
    unsigned int ipbin = ntohl(as_integer[4]);
    unsigned char ip[4] = {0};
    int i;
    for (i = 0; i < 4; i++)
        ip[i] = (ipbin >> i*8) & 0xff;
    fprintf(stderr, "Offered IP %d.%d.%d.%d\n", ip[3], ip[2], ip[1], ip[0]);

    free(buf->base);
    uv_udp_recv_stop(req);
}
```

4.2.1 UDP Options

Time-to-live

The TTL of packets sent on the socket can be changed using uv_udp_set_ttl.

IPv6 stack only

IPv6 sockets can be used for both IPv4 and IPv6 communication. If you want to restrict the socket to IPv6 only, pass the UV_UDP_IPV6ONLY flag to uv_udp_bind [2].

[1] http://beej.us/guide/bgnet/output/html/multipage/advanced.html#broadcast

[2] on Windows only supported on Windows Vista and later.

Multicast

A socket can (un)subscribe to a multicast group using:

```
UV_EXTERN int uv_udp_set_membership(uv_udp_t* handle,
                                    const char* multicast_addr,
                                    const char* interface_addr,
                                    uv_membership membership);
```

where `membership` is `UV_JOIN_GROUP` or `UV_LEAVE_GROUP`.

The concepts of multicasting are nicely explained in this guide.

Local loopback of multicast packets is enabled by default [3], use `uv_udp_set_multicast_loop` to switch it off.

The packet time-to-live for multicast packets can be changed using `uv_udp_set_multicast_ttl`.

4.3 Querying DNS

libuv provides asynchronous DNS resolution. For this it provides its own `getaddrinfo` replacement [4]. In the callback you can perform normal socket operations on the retrieved addresses. Let's connect to Freenode to see an example of DNS resolution.

dns/main.c

```
int main() {
    loop = uv_default_loop();

    struct addrinfo hints;
    hints.ai_family = PF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = 0;

    uv_getaddrinfo_t resolver;
    fprintf(stderr, "irc.freenode.net is... ");
    int r = uv_getaddrinfo(loop, &resolver, on_resolved, "irc.freenode.net", "6667", &hints);

    if (r) {
        fprintf(stderr, "getaddrinfo call error %s\n", uv_err_name(r));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

If `uv_getaddrinfo` returns non-zero, something went wrong in the setup and your callback won't be invoked at all. All arguments can be freed immediately after `uv_getaddrinfo` returns. The *hostname*, *servname* and *hints* structures are documented in the getaddrinfo man page. The callback can be `NULL` in which case the function will run synchronously.

In the resolver callback, you can pick any IP from the linked list of `struct addrinfo(s)`. This also demonstrates `uv_tcp_connect`. It is necessary to call `uv_freeaddrinfo` in the callback.

[3] http://www.tldp.org/HOWTO/Multicast-HOWTO-6.html#ss6.1

[4] libuv use the system `getaddrinfo` in the libuv threadpool. libuv v0.8.0 and earlier also included c-ares as an alternative, but this has been removed in v0.9.0.

dns/main.c

```c
void on_resolved(uv_getaddrinfo_t *resolver, int status, struct addrinfo *res) {
    if (status < 0) {
        fprintf(stderr, "getaddrinfo callback error %s\n", uv_err_name(status));
        return;
    }

    char addr[17] = {'\0'};
    uv_ip4_name((struct sockaddr_in*) res->ai_addr, addr, 16);
    fprintf(stderr, "%s\n", addr);

    uv_connect_t *connect_req = (uv_connect_t*) malloc(sizeof(uv_connect_t));
    uv_tcp_t *socket = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, socket);

    uv_tcp_connect(connect_req, socket, (const struct sockaddr*) res->ai_addr, on_connect);

    uv_freeaddrinfo(res);
}
```

libuv also provides the inverse uv_getnameinfo.

4.4 Network interfaces

Information about the system's network interfaces can be obtained through libuv using `uv_interface_addresses`. This simple program just prints out all the interface details so you get an idea of the fields that are available. This is useful to allow your service to bind to IP addresses when it starts.

interfaces/main.c

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

int main() {
    char buf[512];
    uv_interface_address_t *info;
    int count, i;

    uv_interface_addresses(&info, &count);
    i = count;

    printf("Number of interfaces: %d\n", count);
    while (i--) {
        uv_interface_address_t interface = info[i];

        printf("Name: %s\n", interface.name);
        printf("Internal? %s\n", interface.is_internal ? "Yes" : "No");

        if (interface.address.address4.sin_family == AF_INET) {
            uv_ip4_name(&interface.address.address4, buf, sizeof(buf));
            printf("IPv4 address: %s\n", buf);
        }
        else if (interface.address.address4.sin_family == AF_INET6) {
```

```
            uv_ip6_name(&interface.address.address6, buf, sizeof(buf));
            printf("IPv6 address: %s\n", buf);
        }

        printf("\n");
    }

    uv_free_interface_addresses(info, count);
    return 0;
}
```

`is_internal` is true for loopback interfaces. Note that if a physical interface has multiple IPv4/IPv6 addresses, the name will be reported multiple times, with each address being reported once.

CHAPTER

FIVE

THREADS

Wait a minute? Why are we on threads? Aren't event loops supposed to be **the way** to do *web-scale programming*? Well... no. Threads are still the medium in which processors do their jobs. Threads are therefore mighty useful sometimes, even though you might have to wade through various synchronization primitives.

Threads are used internally to fake the asynchronous nature of all of the system calls. libuv also uses threads to allow you, the application, to perform a task asynchronously that is actually blocking, by spawning a thread and collecting the result when it is done.

Today there are two predominant thread libraries: the Windows threads implementation and POSIX's pthreads. libuv's thread API is analogous to the pthreads API and often has similar semantics.

A notable aspect of libuv's thread facilities is that it is a self contained section within libuv. Whereas other features intimately depend on the event loop and callback principles, threads are complete agnostic, they block as required, signal errors directly via return values, and, as shown in the *first example*, don't even require a running event loop.

libuv's thread API is also very limited since the semantics and syntax of threads are different on all platforms, with different levels of completeness.

This chapter makes the following assumption: **There is only one event loop, running in one thread (the main thread)**. No other thread interacts with the event loop (except using `uv_async_send`).

5.1 Core thread operations

There isn't much here, you just start a thread using `uv_thread_create()` and wait for it to close using `uv_thread_join()`.

thread-create/main.c

```
int main() {
    int tracklen = 10;
    uv_thread_t hare_id;
    uv_thread_t tortoise_id;
    uv_thread_create(&hare_id, hare, &tracklen);
    uv_thread_create(&tortoise_id, tortoise, &tracklen);

    uv_thread_join(&hare_id);
    uv_thread_join(&tortoise_id);
    return 0;
}
```

Tip: `uv_thread_t` is just an alias for `pthread_t` on Unix, but this is an implementation detail, avoid depending on it to always be true.

The second parameter is the function which will serve as the entry point for the thread, the last parameter is a `void *` argument which can be used to pass custom parameters to the thread. The function `hare` will now run in a separate thread, scheduled pre-emptively by the operating system:

thread-create/main.c

```
void hare(void *arg) {
    int tracklen = *((int *) arg);
    while (tracklen) {
        tracklen--;
        sleep(1);
        fprintf(stderr, "Hare ran another step\n");
    }
    fprintf(stderr, "Hare done running!\n");
}
```

Unlike `pthread_join()` which allows the target thread to pass back a value to the calling thread using a second parameter, `uv_thread_join()` does not. To send values use *Inter-thread communication*.

5.2 Synchronization Primitives

This section is purposely spartan. This book is not about threads, so I only catalogue any surprises in the libuv APIs here. For the rest you can look at the pthreads man pages.

5.2.1 Mutexes

The mutex functions are a **direct** map to the pthread equivalents.

libuv mutex functions

```
UV_EXTERN int uv_mutex_init(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_destroy(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_lock(uv_mutex_t* handle);
UV_EXTERN int uv_mutex_trylock(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_unlock(uv_mutex_t* handle);
```

The `uv_mutex_init()` and `uv_mutex_trylock()` functions will return 0 on success, and an error code otherwise.

If *libuv* has been compiled with debugging enabled, `uv_mutex_destroy()`, `uv_mutex_lock()` and `uv_mutex_unlock()` will `abort()` on error. Similarly `uv_mutex_trylock()` will abort if the error is anything *other than* `EAGAIN` or `EBUSY`.

Recursive mutexes are supported by some platforms, but you should not rely on them. The BSD mutex implementation will raise an error if a thread which has locked a mutex attempts to lock it again. For example, a construct like:

```
uv_mutex_lock(a_mutex);
uv_thread_create(thread_id, entry, (void *)a_mutex);
uv_mutex_lock(a_mutex);
// more things here
```

can be used to wait until another thread initializes some stuff and then unlocks `a_mutex` but will lead to your program crashing if in debug mode, or return an error in the second call to `uv_mutex_lock()`.

Note: Mutexes on linux support attributes for a recursive mutex, but the API is not exposed via libuv.

5.2.2 Locks

Read-write locks are a more granular access mechanism. Two readers can access shared memory at the same time. A writer may not acquire the lock when it is held by a reader. A reader or writer may not acquire a lock when a writer is holding it. Read-write locks are frequently used in databases. Here is a toy example.

locks/main.c - simple rwlocks

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

uv_barrier_t blocker;
uv_rwlock_t numlock;
int shared_num;

void reader(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_rdlock(&numlock);
        printf("Reader %d: acquired lock\n", num);
        printf("Reader %d: shared num = %d\n", num, shared_num);
        uv_rwlock_rdunlock(&numlock);
        printf("Reader %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

void writer(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_wrlock(&numlock);
        printf("Writer %d: acquired lock\n", num);
        shared_num++;
        printf("Writer %d: incremented shared num = %d\n", num, shared_num);
        uv_rwlock_wrunlock(&numlock);
        printf("Writer %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

int main()
{
    uv_barrier_init(&blocker, 4);

    shared_num = 0;
    uv_rwlock_init(&numlock);
```

```c
    uv_thread_t threads[3];

    int thread_nums[] = {1, 2, 1};
    uv_thread_create(&threads[0], reader, &thread_nums[0]);
    uv_thread_create(&threads[1], reader, &thread_nums[1]);

    uv_thread_create(&threads[2], writer, &thread_nums[2]);

    uv_barrier_wait(&blocker);
    uv_barrier_destroy(&blocker);

    uv_rwlock_destroy(&numlock);
    return 0;
}
```

Run this and observe how the readers will sometimes overlap. In case of multiple writers, schedulers will usually give them higher priority, so if you add two writers, you'll see that both writers tend to finish first before the readers get a chance again.

We also use barriers in the above example so that the main thread can wait for all readers and writers to indicate they have ended.

5.2.3 Others

libuv also supports semaphores, condition variables and barriers with APIs very similar to their pthread counterparts.

In addition, libuv provides a convenience function `uv_once()`. Multiple threads can attempt to call `uv_once()` with a given guard and a function pointer, **only the first one will win, the function will be called once and only once**:

```c
/* Initialize guard */
static uv_once_t once_only = UV_ONCE_INIT;

int i = 0;

void increment() {
    i++;
}

void thread1() {
    /* ... work */
    uv_once(once_only, increment);
}

void thread2() {
    /* ... work */
    uv_once(once_only, increment);
}

int main() {
    /* ... spawn threads */
}
```

After all threads are done, `i == 1`. libuv v0.11.11 onwards also added a `uv_key_t` struct and api for thread-local storage.

5.3 libuv work queue

`uv_queue_work()` is a convenience function that allows an application to run a task in a separate thread, and have a callback that is triggered when the task is done. A seemingly simple function, what makes `uv_queue_work()` tempting is that it allows potentially any third-party libraries to be used with the event-loop paradigm. When you use event loops, it is *imperative to make sure that no function which runs periodically in the loop thread blocks when performing I/O or is a serious CPU hog*, because this means that the loop slows down and events are not being handled at full capacity.

However, a lot of existing code out there features blocking functions (for example a routine which performs I/O under the hood) to be used with threads if you want responsiveness (the classic 'one thread per client' server model), and getting them to play with an event loop library generally involves rolling your own system of running the task in a separate thread. libuv just provides a convenient abstraction for this.

Here is a simple example inspired by node.js is cancer. We are going to calculate fibonacci numbers, sleeping a bit along the way, but run it in a separate thread so that the blocking and CPU bound task does not prevent the event loop from performing other activities.

queue-work/main.c - lazy fibonacci

```
void fib(uv_work_t *req) {
    int n = *(int *) req->data;
    if (random() % 2)
        sleep(1);
    else
        sleep(3);
    long fib = fib_(n);
    fprintf(stderr, "%dth fibonacci is %lu\n", n, fib);
}

void after_fib(uv_work_t *req, int status) {
    fprintf(stderr, "Done calculating %dth fibonacci\n", *(int *) req->data);
}
```

The actual task function is simple, nothing to show that it is going to be run in a separate thread. The `uv_work_t` structure is the clue. You can pass arbitrary data through it using the `void* data` field and use it to communicate to and from the thread. But be sure you are using proper locks if you are changing things while both threads may be running.

The trigger is `uv_queue_work`:

queue-work/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    uv_work_t req[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        req[i].data = (void *) &data[i];
        uv_queue_work(loop, &req[i], fib, after_fib);
    }

```

```
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The thread function will be launched in a separate thread, passed the `uv_work_t` structure and once the function returns, the *after* function will be called on the thread the event loop is running in. It will be passed the same structure.

For writing wrappers to blocking libraries, a common *pattern* is to use a baton to exchange data.

Since libuv version *0.9.4* an additional function, `uv_cancel()`, is available. This allows you to cancel tasks on the libuv work queue. Only tasks that *are yet to be started* can be cancelled. If a task has *already started executing, or it has finished executing*, `uv_cancel()` **will fail**.

`uv_cancel()` is useful to cleanup pending tasks if the user requests termination. For example, a music player may queue up multiple directories to be scanned for audio files. If the user terminates the program, it should quit quickly and not wait until all pending requests are run.

Let's modify the fibonacci example to demonstrate `uv_cancel()`. We first set up a signal handler for termination.

queue-cancel/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        fib_reqs[i].data = (void *) &data[i];
        uv_queue_work(loop, &fib_reqs[i], fib, after_fib);
    }

    uv_signal_t sig;
    uv_signal_init(loop, &sig);
    uv_signal_start(&sig, signal_handler, SIGINT);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

When the user triggers the signal by pressing `Ctrl+C` we send `uv_cancel()` to all the workers. `uv_cancel()` will return `0` for those that are already executing or finished.

queue-cancel/main.c

```
void signal_handler(uv_signal_t *req, int signum)
{
    printf("Signal received!\n");
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        uv_cancel((uv_req_t*) &fib_reqs[i]);
    }
    uv_signal_stop(req);
}
```

For tasks that do get cancelled successfully, the *after* function is called with `status` set to `UV_ECANCELED`.

queue-cancel/main.c

```
void after_fib(uv_work_t *req, int status) {
    if (status == UV_ECANCELED)
        fprintf(stderr, "Calculation of %d cancelled.\n", *(int *) req->data);
}
```

`uv_cancel()` can also be used with `uv_fs_t` and `uv_getaddrinfo_t` requests. For the filesystem family of functions, `uv_fs_t.errorno` will be set to `UV_ECANCELED`.

Tip: A well designed program would have a way to terminate long running workers that have already started executing. Such a worker could periodically check for a variable that only the main process sets to signal termination.

5.4 Inter-thread communication

Sometimes you want various threads to actually send each other messages *while* they are running. For example you might be running some long duration task in a separate thread (perhaps using `uv_queue_work`) but want to notify progress to the main thread. This is a simple example of having a download manager informing the user of the status of running downloads.

progress/main.c

```
uv_loop_t *loop;
uv_async_t async;

int main() {
    loop = uv_default_loop();

    uv_work_t req;
    int size = 10240;
    req.data = (void*) &size;

    uv_async_init(loop, &async, print_progress);
    uv_queue_work(loop, &req, fake_download, after);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The async thread communication works *on loops* so although any thread can be the message sender, only threads with libuv loops can be receivers (or rather the loop is the receiver). libuv will invoke the callback (`print_progress`) with the async watcher whenever it receives a message.

Warning: It is important to realize that since the message send is *async*, the callback may be invoked immediately after `uv_async_send` is called in another thread, or it may be invoked after some time. libuv may also combine multiple calls to `uv_async_send` and invoke your callback only once. The only guarantee that libuv makes is – The callback function is called *at least once* after the call to `uv_async_send`. If you have no pending calls to `uv_async_send`, the callback won't be called. If you make two or more calls, and libuv hasn't had a chance to run the callback yet, it *may* invoke your callback *only once* for the multiple invocations of `uv_async_send`. Your callback will never be called twice for just one event.

progress/main.c

```c
void fake_download(uv_work_t *req) {
    int size = *((int*) req->data);
    int downloaded = 0;
    double percentage;
    while (downloaded < size) {
        percentage = downloaded*100.0/size;
        async.data = (void*) &percentage;
        uv_async_send(&async);

        sleep(1);
        downloaded += (200+random())%1000; // can only download max 1000bytes/sec,
                                           // but at least a 200;
    }
}
```

In the download function, we modify the progress indicator and queue the message for delivery with `uv_async_send`. Remember: `uv_async_send` is also non-blocking and will return immediately.

progress/main.c

```c
void print_progress(uv_async_t *handle) {
    double percentage = *((double*) handle->data);
    fprintf(stderr, "Downloaded %.2f%%\n", percentage);
}
```

The callback is a standard libuv pattern, extracting the data from the watcher.

Finally it is important to remember to clean up the watcher.

progress/main.c

```c
void after(uv_work_t *req, int status) {
    fprintf(stderr, "Download complete\n");
    uv_close((uv_handle_t*) &async, NULL);
}
```

After this example, which showed the abuse of the `data` field, bnoordhuis pointed out that using the `data` field is not thread safe, and `uv_async_send()` is actually only meant to wake up the event loop. Use a mutex or rwlock to ensure accesses are performed in the right order.

Note: mutexes and rwlocks **DO NOT** work inside a signal handler, whereas `uv_async_send` does.

One use case where `uv_async_send` is required is when interoperating with libraries that require thread affinity for their functionality. For example in node.js, a v8 engine instance, contexts and its objects are bound to the thread that the v8 instance was started in. Interacting with v8 data structures from another thread can lead to undefined results. Now consider some node.js module which binds a third party library. It may go something like this:

1. In node, the third party library is set up with a JavaScript callback to be invoked for more information:

```
var lib = require('lib');
lib.on_progress(function() {
    console.log("Progress");
});
```

```
    lib.do();

    // do other stuff
```

2. `lib.do` is supposed to be non-blocking but the third party lib is blocking, so the binding uses `uv_queue_work`.
3. The actual work being done in a separate thread wants to invoke the progress callback, but cannot directly call into v8 to interact with JavaScript. So it uses `uv_async_send`.
4. The async callback, invoked in the main loop thread, which is the v8 thread, then interacts with v8 to invoke the JavaScript callback.

CHAPTER

SIX

PROCESSES

libuv offers considerable child process management, abstracting the platform differences and allowing communication with the child process using streams or named pipes.

A common idiom in Unix is for every process to do one thing and do it well. In such a case, a process often uses multiple child processes to achieve tasks (similar to using pipes in shells). A multi-process model with messages may also be easier to reason about compared to one with threads and shared memory.

A common refrain against event-based programs is that they cannot take advantage of multiple cores in modern computers. In a multi-threaded program the kernel can perform scheduling and assign different threads to different cores, improving performance. But an event loop has only one thread. The workaround can be to launch multiple processes instead, with each process running an event loop, and each process getting assigned to a separate CPU core.

6.1 Spawning child processes

The simplest case is when you simply want to launch a process and know when it exits. This is achieved using `uv_spawn`.

spawn/main.c

```
uv_loop_t *loop;
uv_process_t child_req;
uv_process_options_t options;
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "mkdir";
    args[1] = "test-dir";
    args[2] = NULL;

    options.exit_cb = on_exit;
    options.file = "mkdir";
    options.args = args;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    } else {
        fprintf(stderr, "Launched process with ID %d\n", child_req.pid);
    }
```

```
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

Note: options is implicitly initialized with zeros since it is a global variable. If you change options to a local variable, remember to initialize it to null out all unused fields:

```
uv_process_options_t options = {0};
```

The uv_process_t struct only acts as the handle, all options are set via uv_process_options_t. To simply launch a process, you need to set only the file and args fields. file is the program to execute. Since uv_spawn uses execvp internally, there is no need to supply the full path. Finally as per underlying conventions, **the arguments array has to be one larger than the number of arguments, with the last element being NULL**.

After the call to uv_spawn, uv_process_t.pid will contain the process ID of the child process.

The exit callback will be invoked with the *exit status* and the type of *signal* which caused the exit.

spawn/main.c

```
void on_exit(uv_process_t *req, int64_t exit_status, int term_signal) {
    fprintf(stderr, "Process exited with status %" PRId64 ", signal %d\n", exit_status, term_signal);
    uv_close((uv_handle_t*) req, NULL);
```

It is **required** to close the process watcher after the process exits.

6.2 Changing process parameters

Before the child process is launched you can control the execution environment using fields in uv_process_options_t.

6.2.1 Change execution directory

Set uv_process_options_t.cwd to the corresponding directory.

6.2.2 Set environment variables

uv_process_options_t.env is a null-terminated array of strings, each of the form VAR=VALUE used to set up the environment variables for the process. Set this to NULL to inherit the environment from the parent (this) process.

6.2.3 Option flags

Setting uv_process_options_t.flags to a bitwise OR of the following flags, modifies the child process behaviour:

- UV_PROCESS_SETUID - sets the child's execution user ID to uv_process_options_t.uid.
- UV_PROCESS_SETGID - sets the child's execution group ID to uv_process_options_t.gid.

Changing the UID/GID is only supported on Unix, uv_spawn will fail on Windows with UV_ENOTSUP.

- `UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS` - No quoting or escaping of `uv_process_options_t.args` is done on Windows. Ignored on Unix.
- `UV_PROCESS_DETACHED` - Starts the child process in a new session, which will keep running after the parent process exits. See example below.

6.3 Detaching processes

Passing the flag `UV_PROCESS_DETACHED` can be used to launch daemons, or child processes which are independent of the parent so that the parent exiting does not affect it.

detach/main.c

```
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "sleep";
    args[1] = "100";
    args[2] = NULL;

    options.exit_cb = NULL;
    options.file = "sleep";
    options.args = args;
    options.flags = UV_PROCESS_DETACHED;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    }
    fprintf(stderr, "Launched sleep with PID %d\n", child_req.pid);
    uv_unref((uv_handle_t*) &child_req);

    return uv_run(loop, UV_RUN_DEFAULT);
```

Just remember that the handle is still monitoring the child, so your program won't exit. Use `uv_unref()` if you want to be more *fire-and-forget*.

6.4 Sending signals to processes

libuv wraps the standard `kill(2)` system call on Unix and implements one with similar semantics on Windows, with *one caveat*: all of `SIGTERM`, `SIGINT` and `SIGKILL`, lead to termination of the process. The signature of `uv_kill` is:

```
uv_err_t uv_kill(int pid, int signum);
```

For processes started using libuv, you may use `uv_process_kill` instead, which accepts the `uv_process_t` watcher as the first argument, rather than the pid. In this case, **remember to call** `uv_close` on the watcher.

6.5 Signals

libuv provides wrappers around Unix signals with some Windows support as well.

Use `uv_signal_init()` to initialize a handle and associate it with a loop. To listen for particular signals on that handler, use `uv_signal_start()` with the handler function. Each handler can only be associated with one signal number, with subsequent calls to `uv_signal_start()` overwriting earlier associations. Use `uv_signal_stop()` to stop watching. Here is a small example demonstrating the various possibilities:

signal/main.c

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

uv_loop_t* create_loop()
{
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    if (loop) {
      uv_loop_init(loop);
    }
    return loop;
}

void signal_handler(uv_signal_t *handle, int signum)
{
    printf("Signal received: %d\n", signum);
    uv_signal_stop(handle);
}

// two signal handlers in one loop
void thread1_worker(void *userp)
{
    uv_loop_t *loop1 = create_loop();

    uv_signal_t sig1a, sig1b;
    uv_signal_init(loop1, &sig1a);
    uv_signal_start(&sig1a, signal_handler, SIGUSR1);

    uv_signal_init(loop1, &sig1b);
    uv_signal_start(&sig1b, signal_handler, SIGUSR1);

    uv_run(loop1, UV_RUN_DEFAULT);
}

// two signal handlers, each in its own loop
void thread2_worker(void *userp)
{
    uv_loop_t *loop2 = create_loop();
    uv_loop_t *loop3 = create_loop();

    uv_signal_t sig2;
    uv_signal_init(loop2, &sig2);
    uv_signal_start(&sig2, signal_handler, SIGUSR1);

```

```
    uv_signal_t sig3;
    uv_signal_init(loop3, &sig3);
    uv_signal_start(&sig3, signal_handler, SIGUSR1);

    while (uv_run(loop2, UV_RUN_NOWAIT) || uv_run(loop3, UV_RUN_NOWAIT)) {
    }
}

int main()
{
    printf("PID %d\n", getpid());

    uv_thread_t thread1, thread2;

    uv_thread_create(&thread1, thread1_worker, 0);
    uv_thread_create(&thread2, thread2_worker, 0);

    uv_thread_join(&thread1);
    uv_thread_join(&thread2);
    return 0;
}
```

Note: `uv_run(loop, UV_RUN_NOWAIT)` is similar to `uv_run(loop, UV_RUN_ONCE)` in that it will process only one event. UV_RUN_ONCE blocks if there are no pending events, while UV_RUN_NOWAIT will return immediately. We use NOWAIT so that one of the loops isn't starved because the other one has no pending activity.

Send `SIGUSR1` to the process, and you'll find the handler being invoked 4 times, one for each `uv_signal_t`. The handler just stops each handle, so that the program exits. This sort of dispatch to all handlers is very useful. A server using multiple event loops could ensure that all data was safely saved before termination, simply by every loop adding a watcher for `SIGINT`.

6.6 Child Process I/O

A normal, newly spawned process has its own set of file descriptors, with 0, 1 and 2 being `stdin`, `stdout` and `stderr` respectively. Sometimes you may want to share file descriptors with the child. For example, perhaps your applications launches a sub-command and you want any errors to go in the log file, but ignore `stdout`. For this you'd like to have `stderr` of the child be the same as the stderr of the parent. In this case, libuv supports *inheriting* file descriptors. In this sample, we invoke the test program, which is:

proc-streams/test.c

```
#include <stdio.h>

int main()
{
    fprintf(stderr, "This is stderr\n");
    printf("This is stdout\n");
    return 0;
}
```

The actual program `proc-streams` runs this while sharing only `stderr`. The file descriptors of the child process are set using the `stdio` field in `uv_process_options_t`. First set the `stdio_count` field to the number of file descriptors being set. uv_process_options_t.stdio is an array of uv_stdio_container_t, which is:

```
typedef struct uv_stdio_container_s {
  uv_stdio_flags flags;

  union {
    uv_stream_t* stream;
    int fd;
  } data;
} uv_stdio_container_t;
```

where flags can have several values. Use UV_IGNORE if it isn't going to be used. If the first three stdio fields are marked as UV_IGNORE they'll redirect to /dev/null.

Since we want to pass on an existing descriptor, we'll use UV_INHERIT_FD. Then we set the fd to stderr.

proc-streams/main.c

```
int main() {
    loop = uv_default_loop();

    /* ... */

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_IGNORE;
    child_stdio[2].flags = UV_INHERIT_FD;
    child_stdio[2].data.fd = 2;
    options.stdio = child_stdio;

    options.exit_cb = on_exit;
    options.file = args[0];
    options.args = args;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

If you run proc-stream you'll see that only the line "This is stderr" will be displayed. Try marking stdout as being inherited and see the output.

It is dead simple to apply this redirection to streams. By setting flags to UV_INHERIT_STREAM and setting data.stream to the stream in the parent process, the child process can treat that stream as standard I/O. This can be used to implement something like CGI.

A sample CGI script/executable is:

cgi/tick.c

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

int main() {
    int i;
    for (i = 0; i < 10; i++) {
        printf("tick\n");
        fflush(stdout);
        sleep(1);
    }
    printf("BOOM!\n");
    return 0;
}
```

The CGI server combines the concepts from this chapter and Networking so that every client is sent ten ticks after which that connection is closed.

cgi/main.c

```c
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        invoke_cgi_script(client);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
```

Here we simply accept the TCP connection and pass on the socket (*stream*) to `invoke_cgi_script`.

cgi/main.c

```c
    args[1] = NULL;

    /* ... finding the executable path and setting up arguments ... */

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_INHERIT_STREAM;
    child_stdio[1].data.stream = (uv_stream_t*) client;
    child_stdio[2].flags = UV_IGNORE;
    options.stdio = child_stdio;

    options.exit_cb = cleanup_handles;
    options.file = args[0];
    options.args = args;
```

```
    // Set this so we can close the socket after the child process exits.
    child_req.data = (void*) client;
    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
```

The `stdout` of the CGI script is set to the socket so that whatever our tick script prints, gets sent to the client. By using processes, we can offload the read/write buffering to the operating system, so in terms of convenience this is great. Just be warned that creating processes is a costly task.

6.7 Pipes

libuv's `uv_pipe_t` structure is slightly confusing to Unix programmers, because it immediately conjures up `|` and pipe(7). But `uv_pipe_t` is not related to anonymous pipes, rather it is an IPC mechanism. `uv_pipe_t` can be backed by a Unix Domain Socket or Windows Named Pipe to allow multiple processes to communicate. This is discussed below.

6.7.1 Parent-child IPC

A parent and child can have one or two way communication over a pipe created by settings `uv_stdio_container_t.flags` to a bit-wise combination of `UV_CREATE_PIPE` and `UV_READABLE_PIPE` or `UV_WRITABLE_PIPE`. The read/write flag is from the perspective of the child process.

6.7.2 Arbitrary process IPC

Since domain sockets [1] can have a well known name and a location in the file-system they can be used for IPC between unrelated processes. The D-BUS system used by open source desktop environments uses domain sockets for event notification. Various applications can then react when a contact comes online or new hardware is detected. The MySQL server also runs a domain socket on which clients can interact with it.

When using domain sockets, a client-server pattern is usually followed with the creator/owner of the socket acting as the server. After the initial setup, messaging is no different from TCP, so we'll re-use the echo server example.

pipe-echo-server/main.c

```
int main() {
    loop = uv_default_loop();

    uv_pipe_t server;
    uv_pipe_init(loop, &server, 0);

    signal(SIGINT, remove_sock);

    int r;
    if ((r = uv_pipe_bind(&server, "echo.sock"))) {
        fprintf(stderr, "Bind error %s\n", uv_err_name(r));
        return 1;
    }
    if ((r = uv_listen((uv_stream_t*) &server, 128, on_new_connection))) {
```

[1] In this section domain sockets stands in for named pipes on Windows as well.

```
        fprintf(stderr, "Listen error %s\n", uv_err_name(r));
        return 2;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

We name the socket `echo.sock` which means it will be created in the local directory. This socket now behaves no different from TCP sockets as far as the stream API is concerned. You can test this server using socat:

```
$ socat - /path/to/socket
```

A client which wants to connect to a domain socket will use:

```
void uv_pipe_connect(uv_connect_t *req, uv_pipe_t *handle, const char *name, uv_connect_cb cb);
```

where `name` will be `echo.sock` or similar.

6.7.3 Sending file descriptors over pipes

The cool thing about domain sockets is that file descriptors can be exchanged between processes by sending them over a domain socket. This allows processes to hand off their I/O to other processes. Applications include load-balancing servers, worker processes and other ways to make optimum use of CPU. libuv only supports sending **TCP sockets or other pipes** over pipes for now.

To demonstrate, we will look at a echo server implementation that hands of clients to worker processes in a round-robin fashion. This program is a bit involved, and while only snippets are included in the book, it is recommended to read the full code to really understand it.

The worker process is quite simple, since the file-descriptor is handed over to it by the master.

multi-echo-server/worker.c

```
uv_loop_t *loop;
uv_pipe_t queue;
int main() {
    loop = uv_default_loop();

    uv_pipe_init(loop, &queue, 1 /* ipc */);
    uv_pipe_open(&queue, 0);
    uv_read_start((uv_stream_t*)&queue, alloc_buffer, on_new_connection);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

`queue` is the pipe connected to the master process on the other end, along which new file descriptors get sent. It is important to set the `ipc` argument of `uv_pipe_init` to 1 to indicate this pipe will be used for inter-process communication! Since the master will write the file handle to the standard input of the worker, we connect the pipe to `stdin` using `uv_pipe_open`.

multi-echo-server/worker.c

```
void on_new_connection(uv_stream_t *q, ssize_t nread, const uv_buf_t *buf) {
    if (nread < 0) {
        if (nread != UV_EOF)
            fprintf(stderr, "Read error %s\n", uv_err_name(nread));
        uv_close((uv_handle_t*) q, NULL);
```

```
        return;
    }

    uv_pipe_t *pipe = (uv_pipe_t*) q;
    if (!uv_pipe_pending_count(pipe)) {
        fprintf(stderr, "No pending count\n");
        return;
    }

    uv_handle_type pending = uv_pipe_pending_type(pipe);
    assert(pending == UV_TCP);

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(q, (uv_stream_t*) client) == 0) {
        uv_os_fd_t fd;
        uv_fileno((const uv_handle_t*) client, &fd);
        fprintf(stderr, "Worker %d: Accepted fd %d\n", getpid(), fd);
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

First we call `uv_pipe_pending_count()` to ensure that a handle is available to read out. If your program could deal with different types of handles, `uv_pipe_pending_type()` can be used to determine the type. Although `accept` seems odd in this code, it actually makes sense. What `accept` traditionally does is get a file descriptor (the client) from another file descriptor (The listening socket). Which is exactly what we do here. Fetch the file descriptor (`client`) from `queue`. From this point the worker does standard echo server stuff.

Turning now to the master, let's take a look at how the workers are launched to allow load balancing.

multi-echo-server/main.c

```
struct child_worker {
    uv_process_t req;
    uv_process_options_t options;
    uv_pipe_t pipe;
} *workers;
```

The `child_worker` structure wraps the process, and the pipe between the master and the individual process.

multi-echo-server/main.c

```
void setup_workers() {
    round_robin_counter = 0;

    // ...

    // launch same number of workers as number of CPUs
    uv_cpu_info_t *info;
    int cpu_count;
    uv_cpu_info(&info, &cpu_count);
    uv_free_cpu_info(info, cpu_count);
```

```

    child_worker_count = cpu_count;

    workers = calloc(sizeof(struct child_worker), cpu_count);
    while (cpu_count--) {
        struct child_worker *worker = &workers[cpu_count];
        uv_pipe_init(loop, &worker->pipe, 1);

        uv_stdio_container_t child_stdio[3];
        child_stdio[0].flags = UV_CREATE_PIPE | UV_READABLE_PIPE;
        child_stdio[0].data.stream = (uv_stream_t*) &worker->pipe;
        child_stdio[1].flags = UV_IGNORE;
        child_stdio[2].flags = UV_INHERIT_FD;
        child_stdio[2].data.fd = 2;

        worker->options.stdio = child_stdio;
        worker->options.stdio_count = 3;

        worker->options.exit_cb = close_process_handle;
        worker->options.file = args[0];
        worker->options.args = args;

        uv_spawn(loop, &worker->req, &worker->options);
        fprintf(stderr, "Started worker %d\n", worker->req.pid);
    }
}
```

In setting up the workers, we use the nifty libuv function `uv_cpu_info` to get the number of CPUs so we can launch an equal number of workers. Again it is important to initialize the pipe acting as the IPC channel with the third argument as 1. We then indicate that the child process' `stdin` is to be a readable pipe (from the point of view of the child). Everything is straightforward till here. The workers are launched and waiting for file descriptors to be written to their standard input.

It is in `on_new_connection` (the TCP infrastructure is initialized in `main()`), that we accept the client socket and pass it along to the next worker in the round-robin.

multi-echo-server/main.c

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_write_t *write_req = (uv_write_t*) malloc(sizeof(uv_write_t));
        dummy_buf = uv_buf_init("a", 1);
        struct child_worker *worker = &workers[round_robin_counter];
        uv_write2(write_req, (uv_stream_t*) &worker->pipe, &dummy_buf, 1, (uv_stream_t*) client, NULL
        round_robin_counter = (round_robin_counter + 1) % child_worker_count;
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

The `uv_write2` call handles all the abstraction and it is simply a matter of passing in the handle (`client`) as the right argument. With this our multi-process echo server is operational.

Thanks to Kyle for pointing out that `uv_write2()` requires a non-empty buffer even when sending handles.

CHAPTER

SEVEN

ADVANCED EVENT LOOPS

libuv provides considerable user control over event loops, and you can achieve interesting results by juggling multiple loops. You can also embed libuv's event loop into another event loop based library – imagine a Qt based UI, and Qt's event loop driving a libuv backend which does intensive system level tasks.

7.1 Stopping an event loop

`uv_stop()` can be used to stop an event loop. The earliest the loop will stop running is *on the next iteration*, possibly later. This means that events that are ready to be processed in this iteration of the loop will still be processed, so `uv_stop()` can't be used as a kill switch. When `uv_stop()` is called, the loop **won't** block for i/o on this iteration. The semantics of these things can be a bit difficult to understand, so let's look at `uv_run()` where all the control flow occurs.

src/unix/core.c - uv_run

```
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  int timeout;
  int r;
  int ran_pending;

  r = uv__loop_alive(loop);
  if (!r)
    uv__update_time(loop);

  while (r != 0 && loop->stop_flag == 0) {
    uv__update_time(loop);
    uv__run_timers(loop);
    ran_pending = uv__run_pending(loop);
    uv__run_idle(loop);
    uv__run_prepare(loop);

    timeout = 0;
    if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
      timeout = uv_backend_timeout(loop);

    uv__io_poll(loop, timeout);
```

`stop_flag` is set by `uv_stop()`. Now all libuv callbacks are invoked within the event loop, which is why invoking `uv_stop()` in them will still lead to this iteration of the loop occurring. First libuv updates timers, then runs pending timer, idle and prepare callbacks, and invokes any pending I/O callbacks. If you were to call `uv_stop()` in any of them, `stop_flag` would be set. This causes `uv_backend_timeout()` to return `0`, which is why the loop does not block on I/O. If on the other hand, you called `uv_stop()` in one of the check handlers, I/O has already finished and is not affected.

`uv_stop()` is useful to shutdown a loop when a result has been computed or there is an error, without having to ensure that all handlers are stopped one by one.

Here is a simple example that stops the loop and demonstrates how the current iteration of the loop still takes places.

uvstop/main.c

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

int64_t counter = 0;

void idle_cb(uv_idle_t *handle) {
    printf("Idle callback\n");
    counter++;

    if (counter >= 5) {
        uv_stop(uv_default_loop());
        printf("uv_stop() called\n");
    }
}

void prep_cb(uv_prepare_t *handle) {
    printf("Prep callback\n");
}

int main() {
    uv_idle_t idler;
    uv_prepare_t prep;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, idle_cb);

    uv_prepare_init(uv_default_loop(), &prep);
    uv_prepare_start(&prep, prep_cb);

    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    return 0;
}
```

CHAPTER

EIGHT

UTILITIES

This chapter catalogues tools and techniques which are useful for common tasks. The libev man page already covers some patterns which can be adopted to libuv through simple API changes. It also covers parts of the libuv API that don't require entire chapters dedicated to them.

8.1 Timers

Timers invoke the callback after a certain time has elapsed since the timer was started. libuv timers can also be set to invoke at regular intervals instead of just once.

Simple use is to init a watcher and start it with a `timeout`, and optional `repeat`. Timers can be stopped at any time.

```
uv_timer_t timer_req;

uv_timer_init(loop, &timer_req);
uv_timer_start(&timer_req, callback, 5000, 2000);
```

will start a repeating timer, which first starts 5 seconds (the `timeout`) after the execution of `uv_timer_start`, then repeats every 2 seconds (the `repeat`). Use:

```
uv_timer_stop(&timer_req);
```

to stop the timer. This can be used safely from within the callback as well.

The repeat interval can be modified at any time with:

```
uv_timer_set_repeat(uv_timer_t *timer, int64_t repeat);
```

which will take effect **when possible**. If this function is called from a timer callback, it means:

- If the timer was non-repeating, the timer has already been stopped. Use `uv_timer_start` again.
- If the timer is repeating, the next timeout has already been scheduled, so the old repeat interval will be used once more before the timer switches to the new interval.

The utility function:

```
int uv_timer_again(uv_timer_t *)
```

applies **only to repeating timers** and is equivalent to stopping the timer and then starting it with both initial `timeout` and `repeat` set to the old `repeat` value. If the timer hasn't been started it fails (error code `UV_EINVAL`) and returns -1.

An actual timer example is in the *reference count section*.

8.2 Event loop reference count

The event loop only runs as long as there are active handles. This system works by having every handle increase the reference count of the event loop when it is started and decreasing the reference count when stopped. It is also possible to manually change the reference count of handles using:

```
void uv_ref(uv_handle_t*);
void uv_unref(uv_handle_t*);
```

These functions can be used to allow a loop to exit even when a watcher is active or to use custom objects to keep the loop alive.

The latter can be used with interval timers. You might have a garbage collector which runs every X seconds, or your network service might send a heartbeat to others periodically, but you don't want to have to stop them along all clean exit paths or error scenarios. Or you want the program to exit when all your other watchers are done. In that case just unref the timer immediately after creation so that if it is the only watcher running then `uv_run` will still exit.

This is also used in node.js where some libuv methods are being bubbled up to the JS API. A `uv_handle_t` (the superclass of all watchers) is created per JS object and can be ref/unrefed.

ref-timer/main.c

```
uv_loop_t *loop;
uv_timer_t gc_req;
uv_timer_t fake_job_req;

int main() {
    loop = uv_default_loop();

    uv_timer_init(loop, &gc_req);
    uv_unref((uv_handle_t*) &gc_req);

    uv_timer_start(&gc_req, gc, 0, 2000);

    // could actually be a TCP download or something
    uv_timer_init(loop, &fake_job_req);
    uv_timer_start(&fake_job_req, fake_job, 9000, 0);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

We initialize the garbage collector timer, then immediately `unref` it. Observe how after 9 seconds, when the fake job is done, the program automatically exits, even though the garbage collector is still running.

8.3 Idler pattern

The callbacks of idle handles are invoked once per event loop. The idle callback can be used to perform some very low priority activity. For example, you could dispatch a summary of the daily application performance to the developers for analysis during periods of idleness, or use the application's CPU time to perform SETI calculations :) An idle watcher is also useful in a GUI application. Say you are using an event loop for a file download. If the TCP socket is still being established and no other events are present your event loop will pause (**block**), which means your progress bar will freeze and the user will face an unresponsive application. In such a case queue up and idle watcher to keep the UI operational.

idle-compute/main.c

```
uv_loop_t *loop;
uv_fs_t stdin_watcher;
uv_idle_t idler;
char buffer[1024];

int main() {
    loop = uv_default_loop();

    uv_idle_init(loop, &idler);

    uv_buf_t buf = uv_buf_init(buffer, 1024);
    uv_fs_read(loop, &stdin_watcher, 0, &buf, 1, -1, on_type);
    uv_idle_start(&idler, crunch_away);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

Here we initialize the idle watcher and queue it up along with the actual events we are interested in. `crunch_away` will now be called repeatedly until the user types something and presses Return. Then it will be interrupted for a brief amount as the loop deals with the input data, after which it will keep calling the idle callback again.

idle-compute/main.c

```
void crunch_away(uv_idle_t* handle) {
    // Compute extra-terrestrial life
    // fold proteins
    // computer another digit of PI
    // or similar
    fprintf(stderr, "Computing PI...\n");
    // just to avoid overwhelming your terminal emulator
    uv_idle_stop(handle);
}
```

8.4 Passing data to worker thread

When using `uv_queue_work` you'll usually need to pass complex data through to the worker thread. The solution is to use a `struct` and set `uv_work_t.data` to point to it. A slight variation is to have the `uv_work_t` itself as the first member of this struct (called a baton [1]). This allows cleaning up the work request and all the data in one free call.

```
struct ftp_baton {
    uv_work_t req;
    char *host;
    int port;
    char *username;
    char *password;
}
```

```
ftp_baton *baton = (ftp_baton*) malloc(sizeof(ftp_baton));
baton->req.data = (void*) baton;
```

[1] I was first introduced to the term baton in this context, in Konstantin Käfer's excellent slides on writing node.js bindings – http://kkaefer.github.com/node-cpp-modules/#baton

```
baton->host = strdup("my.webhost.com");
baton->port = 21;
// ...

uv_queue_work(loop, &baton->req, ftp_session, ftp_cleanup);
```

Here we create the baton and queue the task.

Now the task function can extract the data it needs:

```
void ftp_session(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    fprintf(stderr, "Connecting to %s\n", baton->host);
}

void ftp_cleanup(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    free(baton->host);
    // ...
    free(baton);
}
```

We then free the baton which also frees the watcher.

8.5 External I/O with polling

Usually third-party libraries will handle their own I/O, and keep track of their sockets and other files internally. In this case it isn't possible to use the standard stream I/O operations, but the library can still be integrated into the libuv event loop. All that is required is that the library allow you to access the underlying file descriptors and provide functions that process tasks in small increments as decided by your application. Some libraries though will not allow such access, providing only a standard blocking function which will perform the entire I/O transaction and only then return. It is unwise to use these in the event loop thread, use the libuv-work-queue instead. Of course, this will also mean losing granular control on the library.

The `uv_poll` section of libuv simply watches file descriptors using the operating system notification mechanism. In some sense, all the I/O operations that libuv implements itself are also backed by `uv_poll` like code. Whenever the OS notices a change of state in file descriptors being polled, libuv will invoke the associated callback.

Here we will walk through a simple download manager that will use libcurl to download files. Rather than give all control to libcurl, we'll instead be using the libuv event loop, and use the non-blocking, async multi interface to progress with the download whenever libuv notifies of I/O readiness.

uvwget/main.c - The setup

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>
#include <curl/curl.h>

uv_loop_t *loop;
CURLM *curl_handle;
uv_timer_t timeout;
```

```
}

int main(int argc, char **argv) {
    loop = uv_default_loop();

    if (argc <= 1)
        return 0;

    if (curl_global_init(CURL_GLOBAL_ALL)) {
        fprintf(stderr, "Could not init cURL\n");
        return 1;
    }

    uv_timer_init(loop, &timeout);

    curl_handle = curl_multi_init();
    curl_multi_setopt(curl_handle, CURLMOPT_SOCKETFUNCTION, handle_socket);
    curl_multi_setopt(curl_handle, CURLMOPT_TIMERFUNCTION, start_timeout);

    while (argc-- > 1) {
        add_download(argv[argc], argc);
    }

    uv_run(loop, UV_RUN_DEFAULT);
    curl_multi_cleanup(curl_handle);
    return 0;
}
```

The way each library is integrated with libuv will vary. In the case of libcurl, we can register two callbacks. The socket callback `handle_socket` is invoked whenever the state of a socket changes and we have to start polling it. `start_timeout` is called by libcurl to notify us of the next timeout interval, after which we should drive libcurl forward regardless of I/O status. This is so that libcurl can handle errors or do whatever else is required to get the download moving.

Our downloader is to be invoked as:

```
$ ./uvwget [url1] [url2] ...
```

So we add each argument as an URL

uvwget/main.c - Adding urls

```
void add_download(const char *url, int num) {
    char filename[50];
    sprintf(filename, "%d.download", num);
    FILE *file;

    file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening %s\n", filename);
        return;
    }

    CURL *handle = curl_easy_init();
    curl_easy_setopt(handle, CURLOPT_WRITEDATA, file);
    curl_easy_setopt(handle, CURLOPT_URL, url);
    curl_multi_add_handle(curl_handle, handle);
```

```
    fprintf(stderr, "Added download %s -> %s\n", url, filename);
}
```

We let libcurl directly write the data to a file, but much more is possible if you so desire.

`start_timeout` will be called immediately the first time by libcurl, so things are set in motion. This simply starts a libuv timer which drives `curl_multi_socket_action` with `CURL_SOCKET_TIMEOUT` whenever it times out. `curl_multi_socket_action` is what drives libcurl, and what we call whenever sockets change state. But before we go into that, we need to poll on sockets whenever `handle_socket` is called.

uvwget/main.c - Setting up polling

```
void start_timeout(CURLM *multi, long timeout_ms, void *userp) {
    if (timeout_ms <= 0)
        timeout_ms = 1; /* 0 means directly call socket_action, but we'll do it in a bit */
    uv_timer_start(&timeout, on_timeout, timeout_ms, 0);
}

int handle_socket(CURL *easy, curl_socket_t s, int action, void *userp, void *socketp) {
    curl_context_t *curl_context;
    if (action == CURL_POLL_IN || action == CURL_POLL_OUT) {
        if (socketp) {
            curl_context = (curl_context_t*) socketp;
        }
        else {
            curl_context = create_curl_context(s);
            curl_multi_assign(curl_handle, s, (void *) curl_context);
        }
    }

    switch (action) {
        case CURL_POLL_IN:
            uv_poll_start(&curl_context->poll_handle, UV_READABLE, curl_perform);
            break;
        case CURL_POLL_OUT:
            uv_poll_start(&curl_context->poll_handle, UV_WRITABLE, curl_perform);
            break;
        case CURL_POLL_REMOVE:
            if (socketp) {
                uv_poll_stop(&((curl_context_t*)socketp)->poll_handle);
                destroy_curl_context((curl_context_t*) socketp);
                curl_multi_assign(curl_handle, s, NULL);
            }
            break;
        default:
            abort();
    }

    return 0;
}
```

We are interested in the socket fd `s`, and the `action`. For every socket we create a `uv_poll_t` handle if it doesn't exist, and associate it with the socket using `curl_multi_assign`. This way `socketp` points to it whenever the callback is invoked.

In the case that the download is done or fails, libcurl requests removal of the poll. So we stop and free the poll handle.

Depending on what events libcurl wishes to watch for, we start polling with `UV_READABLE` or `UV_WRITABLE`. Now libuv will invoke the poll callback whenever the socket is ready for reading or writing. Calling `uv_poll_start` multiple times on the same handle is acceptable, it will just update the events mask with the new value. `curl_perform` is the crux of this program.

uvwget/main.c - Driving libcurl.

```
void curl_perform(uv_poll_t *req, int status, int events) {
    uv_timer_stop(&timeout);
    int running_handles;
    int flags = 0;
    if (status < 0)                      flags = CURL_CSELECT_ERR;
    if (!status && events & UV_READABLE) flags |= CURL_CSELECT_IN;
    if (!status && events & UV_WRITABLE) flags |= CURL_CSELECT_OUT;

    curl_context_t *context;

    context = (curl_context_t*)req;

    curl_multi_socket_action(curl_handle, context->sockfd, flags, &running_handles);
    check_multi_info();
}
```

The first thing we do is to stop the timer, since there has been some progress in the interval. Then depending on what event triggered the callback, we set the correct flags. Then we call `curl_multi_socket_action` with the socket that progressed and the flags informing about what events happened. At this point libcurl does all of its internal tasks in small increments, and will attempt to return as fast as possible, which is exactly what an evented program wants in its main thread. libcurl keeps queueing messages into its own queue about transfer progress. In our case we are only interested in transfers that are completed. So we extract these messages, and clean up handles whose transfers are done.

uvwget/main.c - Reading transfer status.

```
void check_multi_info(void) {
    char *done_url;
    CURLMsg *message;
    int pending;

    while ((message = curl_multi_info_read(curl_handle, &pending))) {
        switch (message->msg) {
        case CURLMSG_DONE:
            curl_easy_getinfo(message->easy_handle, CURLINFO_EFFECTIVE_URL,
                              &done_url);
            printf("%s DONE\n", done_url);

            curl_multi_remove_handle(curl_handle, message->easy_handle);
            curl_easy_cleanup(message->easy_handle);
            break;

        default:
            fprintf(stderr, "CURLMSG default\n");
            abort();
        }
    }
}
```

8.6 Check & Prepare watchers

TODO

8.7 Loading libraries

libuv provides a cross platform API to dynamically load shared libraries. This can be used to implement your own plugin/extension/module system and is used by node.js to implement `require()` support for bindings. The usage is quite simple as long as your library exports the right symbols. Be careful with sanity and security checks when loading third party code, otherwise your program will behave unpredictably. This example implements a very simple plugin system which does nothing except print the name of the plugin.

Let us first look at the interface provided to plugin authors.

plugin/plugin.h

```
#ifndef UVBOOK_PLUGIN_SYSTEM
#define UVBOOK_PLUGIN_SYSTEM

// Plugin authors should use this to register their plugins with mfp.
void mfp_register(const char *name);

#endif
```

You can similarly add more functions that plugin authors can use to do useful things in your application [2]. A sample plugin using this API is:

plugin/hello.c

```
#include "plugin.h"

void initialize() {
    mfp_register("Hello World!");
}
```

Our interface defines that all plugins should have an `initialize` function which will be called by the application. This plugin is compiled as a shared library and can be loaded by running our application:

```
$ ./plugin libhello.dylib
Loading libhello.dylib
Registered plugin "Hello World!"
```

Note: The shared library filename will be different depending on platforms. On Linux it is `libhello.so`.

This is done by using `uv_dlopen` to first load the shared library `libhello.dylib`. Then we get access to the `initialize` function using `uv_dlsym` and invoke it.

[2] mfp is My Fancy Plugin

plugin/main.c

```c
#include "plugin.h"

typedef void (*init_plugin_function)();

void mfp_register(const char *name) {
    fprintf(stderr, "Registered plugin \"%s\"\n", name);
}

int main(int argc, char **argv) {
    if (argc == 1) {
        fprintf(stderr, "Usage: %s [plugin1] [plugin2] ...\n", argv[0]);
        return 0;
    }

    uv_lib_t *lib = (uv_lib_t*) malloc(sizeof(uv_lib_t));
    while (--argc) {
        fprintf(stderr, "Loading %s\n", argv[argc]);
        if (uv_dlopen(argv[argc], lib)) {
            fprintf(stderr, "Error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin_function init_plugin;
        if (uv_dlsym(lib, "initialize", (void **) &init_plugin)) {
            fprintf(stderr, "dlsym error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin();
    }

    return 0;
}
```

`uv_dlopen` expects a path to the shared library and sets the opaque `uv_lib_t` pointer. It returns 0 on success, -1 on error. Use `uv_dlerror` to get the error message.

`uv_dlsym` stores a pointer to the symbol in the second argument in the third argument. `init_plugin_function` is a function pointer to the sort of function we are looking for in the application's plugins.

8.8 TTY

Text terminals have supported basic formatting for a long time, with a pretty standardised command set. This formatting is often used by programs to improve the readability of terminal output. For example `grep --colour`. libuv provides the `uv_tty_t` abstraction (a stream) and related functions to implement the ANSI escape codes across all platforms. By this I mean that libuv converts ANSI codes to the Windows equivalent, and provides functions to get terminal information.

The first thing to do is to initialize a `uv_tty_t` with the file descriptor it reads/writes from. This is achieved with:

```c
int uv_tty_init(uv_loop_t*, uv_tty_t*, uv_file fd, int readable)
```

Set `readable` to true if you plan to use `uv_read_start()` on the stream.

It is then best to use `uv_tty_set_mode` to set the mode to *normal* which enables most TTY formatting, flow-control and other settings. Other modes are also available.

Remember to call `uv_tty_reset_mode` when your program exits to restore the state of the terminal. Just good manners. Another set of good manners is to be aware of redirection. If the user redirects the output of your command to a file, control sequences should not be written as they impede readability and `grep`. To check if the file descriptor is indeed a TTY, call `uv_guess_handle` with the file descriptor and compare the return value with `UV_TTY`.

Here is a simple example which prints white text on a red background:

tty/main.c

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

uv_loop_t *loop;
uv_tty_t tty;
int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, 1, 0);
    uv_tty_set_mode(&tty, UV_TTY_MODE_NORMAL);

    if (uv_guess_handle(1) == UV_TTY) {
        uv_write_t req;
        uv_buf_t buf;
        buf.base = "\033[41;37m";
        buf.len = strlen(buf.base);
        uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    }

    uv_write_t req;
    uv_buf_t buf;
    buf.base = "Hello TTY\n";
    buf.len = strlen(buf.base);
    uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    uv_tty_reset_mode();
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The final TTY helper is `uv_tty_get_winsize()` which is used to get the width and height of the terminal and returns `0` on success. Here is a small program which does some animation using the function and character position escape codes.

tty-gravity/main.c

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

uv_loop_t *loop;
uv_tty_t tty;
uv_timer_t tick;
```

```c
uv_write_t write_req;
int width, height;
int pos = 0;
char *message = "  Hello TTY  ";

void update(uv_timer_t *req) {
    char data[500];

    uv_buf_t buf;
    buf.base = data;
    buf.len = sprintf(data, "\033[2J\033[H\033[%dB\033[%luC\033[42;37m%s",
                            pos,
                            (unsigned long) (width-strlen(message))/2,
                            message);
    uv_write(&write_req, (uv_stream_t*) &tty, &buf, 1, NULL);

    pos++;
    if (pos > height) {
        uv_tty_reset_mode();
        uv_timer_stop(&tick);
    }
}

int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, 1, 0);
    uv_tty_set_mode(&tty, 0);

    if (uv_tty_get_winsize(&tty, &width, &height)) {
        fprintf(stderr, "Could not get TTY information\n");
        uv_tty_reset_mode();
        return 1;
    }

    fprintf(stderr, "Width %d, height %d\n", width, height);
    uv_timer_init(loop, &tick);
    uv_timer_start(&tick, update, 200, 200);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The escape codes are:

Code	Meaning
2 J	Clear part of the screen, 2 is entire screen
H	Moves cursor to certain position, default top-left
n B	Moves cursor down by n lines
n C	Moves cursor right by n columns
m	Obeys string of display settings, in this case green background (40+2), white text (30+7)

As you can see this is very useful to produce nicely formatted output, or even console based arcade games if that tickles your fancy. For fancier control you can try ncurses.

CHAPTER NINE

ABOUT

Nikhil Marathe started writing this book one afternoon (June 16, 2012) when he didn't feel like programming. He had recently been stung by the lack of good documentation on libuv while working on node-taglib. Although reference documentation was present, there were no comprehensive tutorials. This book is the output of that need and tries to be accurate. That said, the book may have mistakes. Pull requests are encouraged. You may also email him if you find an error.

Nikhil is indebted to Marc Lehmann's comprehensive man page about libev which describes much of the semantics of the two libraries.

This book was made using Sphinx and vim.

9.1 Licensing

CHAPTER TEN

ALTERNATE FORMATS

The book is also available in: