THE STRIPBOARD C LIBRARY

Scheduling, Threads, Networking and Notifications for Your Programs

STRECK.AI STOCKHOLM

This book was typeset with TEX in the Computer Modern types, using the `taomac` macros. The illustrations were drawn in METAPOST.

First edition, September 2026.

CONTENTS

PREFACE

A program that does one thing at a time is easy to write and easy to read: it is a list of statements, and the machine works through them in order. Most programs worth writing do not have that luxury. A server has a hundred connections open and must answer whichever speaks next; a controller watches a sensor, a clock and a network at once; even a small tool that fetches a page wants to show that it is still alive while the network takes its time. The moment a program has two activities that proceed at their own pace, the straight line of statements that C gives you stops being enough, and you have to decide what to do about it.

There are three usual answers, and each buys something at a price. Callbacks cut every activity into fragments and hand the fragments to a loop; nothing blocks, and nothing reads like the thing it does, because the state that would have lived in local variables now lives in a structure you maintain by hand. Threads keep each activity whole and readable, and pay for it in shared memory: two threads that touch the same variable at the same time can leave it holding a value neither of them wrote, and the tools for preventing that—locks—bring deadlock, priority inversion, and a class of bug that reproduces once a month. Polling avoids both and burns a processor to do it.

Stripboard is a fourth answer, and this book teaches it. It is a small library for C99 programs, the same source on macOS, Linux and Windows, assembled out of four mechanisms that work better together than apart. The parts of a program talk to each other by posting *notifications* under names, so that a part need not know who is listening. Each activity is written as a plain sequential function—a *context*—that can say "receive the next request" and continue on the next line when the request has come, while the thread that carries it has meanwhile done a hundred other things; a *cooperative scheduler* is what makes that possible. The program waits for the outside world through the kernel's own *event queue,* so that a program with nothing to do costs nothing. And it uses *threads* where it wants more than one processor, under one rule that keeps them out of each other's memory.

This is a manual, not a commentary. It shows you what to call, what each call promises, what the promises are worth, and six complete programs that put them to work. It does not describe how the library is built.

How the book is arranged

The four mechanisms are taught one to a chapter, in the reversed order in which a program comes to need them, and each chapter's program uses only what the chapters before it have explained.

Chapter 1 is notifications, the smallest of the four, needing nothing from the operating system and setting the tone for everything after. Its program is a thermostat on one thread, in which sensors, a heater, a display and a logger meet by name and nothing else.

Chapter 2 is the cooperative scheduler: what a context is, how several of them share a thread, and the calls by which one parks and is resumed. Its program is the one every embedded board runs first, a blinker with two lights and a computation that does not disturb them.

Chapter 3 is kernel event queues: what a program does while it waits for a socket, and the six calls that look like the blocking socket calls they replace. Its program resolves a host name and downloads a page, with a second context that keeps ticking while the network is slow. The chapter ends with what a queue cannot hold—files, and names to resolve—which go to a pool of threads instead, and a second program that copies a file across a socket, reading the disk and driving the socket at once and stopping cleanly on Ctrl-C.

Chapter 4 is threads, and one rule about them that the whole library is built on. Its program is a web server with an acceptor and a pool of worker threads, each with a scheduler and a queue of its own and one context per connection.

Chapter 5 is all four at once: a fetch farm, complete in one process, that needs no network and could not be written sensibly with any three of the mechanisms.

Every chapter opens with a panel of the calls it introduces—the prototype, and what the call promises—so that the chapter can serve afterwards as the reference for those calls. Chapter 5's panel is the whole library, grouped by mechanism. Read the panels first if you like reference before narrative, or skip them and come back; the prose does not depend on them.

The programs are in `examples/`, one file each, and every one of them is read in full somewhere in these pages. They are small, they print what they are doing, and they check their own results. Run each program before you read its chapter and again afterwards; the difference between reading about a scheduler and watching one keep two lights on time while a prime count runs is most of the lesson.

Conventions

Three verbs in the library are spelled with a capital letter, because they are what a program written with it says most often: `Post` sends a notification, `Relinquish` hands the thread back, and `Resume` runs a context. Everything else is lower case

with a module prefix—`coop_`, `evio_`, `notify_`—or has the C11 spelling of the thread library, `thrd_`, `mtx_`, `cnd_`.

One header brings in the whole library:

```
#include "stripboard.h"
```

It also brings in the socket headers of the platform, so that `struct sockaddr_in`, `inet_pton` and `htons` are available after it. On Windows, include it before any `<windows.h>` of your own.

Listings are set in typewriter type and are taken from the files in the distribution. Where a listing has been shortened to fit the page it says so, usually with an ellipsis on a line of its own. Displayed transcripts are what the programs actually print; timings vary by a millisecond or two between machines, and the text says which numbers are meant to be identical everywhere. Every number in the text—the 152 requests, the 5.0-second timeout, the 183 072 primes, the 2.4 seconds of work done in 0.37—is produced by a program in the distribution, and `make check` reproduces them.

The word *we* appears where the two of us are working something out together, and *you* where you are being asked to do something—run a program, make a change, decide between two designs. Both are meant literally.

Stockholm, September 2026

CHAPTER ONE

NOTIFICATIONS

THE CALLS OF THIS CHAPTER

`notify_center * notify_default(void);`
The one center every program has without asking, created on first use. Thread-safe. A program that wants a private registry makes one with `notify_center_init` and disposes of it with `notify_center_destroy`.

`notify_observer * notify_observe(notify_center * c, const char * name, evio * queue, notify_fn fn, void * ctxt);`
Register `fn(n, ctxt)` for notifications named `name`; Λ as the name means every name. The `queue` says which thread to be called on, and is Λ in this chapter. Returns a handle, or Λ if the calling thread has neither a queue nor a scheduler.

`void notify_unobserve(notify_center * c, notify_observer * o);`
Remove an observer. A delivery already queued for it is dropped, not delivered.

`int Post(notify_center * c, const char * name, const void * sender, const void * data, size_t len);`
Announce one notification and return how many observers it reached. The `sender` is an identity the center never dereferences; the `len` bytes at `data` are copied when a delivery has to cross to another thread, so they may live on the caller's stack. Thread-safe, from anywhere.

`typedef void (*notify_fn)(const notification * n, void * ctxt);`
The shape of an observer. It receives the notification and the `ctxt` pointer it was registered with; both are valid only for the duration of the call.

Consider a house with a thermostat. A sensor measures the temperature; a display shows it; a heater comes on when the room is cold and goes off when it is warm; and somewhere a log records what happened. The sensor is the one that knows the temperature, so the obvious program has the sensor call the display, then call the heater, then call the log. That works until the day a second display is added, or the log is moved to another thread, or somebody wants the heater to react to a window sensor too, and each time the sensor's code changes although nothing about sensing has. The sensor has been made responsible for everybody who cares about its readings, and it is the wrong party to hold that list.

There is a second problem, quieter and worse. The display is updated before the heater decides anything, so the display shows the new temperature next to

a stale heater state; nothing in the program says that this ordering matters, and nothing would stop somebody from swapping two lines. The dependency is real and invisible.

The remedy is old and simple: keep the list somewhere else. The sensor announces "the temperature is 20.5" under a name, and whoever cares has registered, under that name, a function to be called. The sensor does not know who is listening and the listeners do not know who is speaking; they agree on the name, and on what the announcement carries. The place where the list lives is a *notification center*, the announcement is a *notification*, and a registered listener is an *observer*.

Fig. 1. The same four parts, before and after. On the left, the sensor holds a list of everybody who cares; on the right, the center holds it, and no part points at any other.

Three things change. The sensor is now shorter, and it stops changing: adding a display is one line somewhere else, and nothing the sensor can see. The compile-time dependencies collapse, since neither the sensor nor the display includes the other's header, and a program built this way can put its parts in any file, in any library, in any order. And the invisible ordering becomes visible, because the center calls observers in the order they registered, which is a rule you can read, in one place, in this chapter.

1.1. The thermostat

The house above is `examples/thermostat.c`, on one thread, in ninety lines. There are two sensors, indoor and outdoor, each a record with a name; a display; a heater that watches the indoor sensor only, switches itself on below 19 degrees and off above 21, and announces every change; and a logger that hears everything. None of them holds a pointer to any other. The parts meet in the notification center under two names, `temperature` and `heater`.

An observer is a function that receives the notification and a pointer of its own choosing. The display prints what it hears:

```
static void on_display(const notification * n, void * ctxt)
{
   const struct sensor * s = (const struct sensor *)n->sender;
   (void)ctxt;
   if (n->len != sizeof(double)) { return; }
   printf("  display %s %.1f\n", s->name, *(const double *)n->data);
}
```

A notification carries a name, a *sender,* some *data* and the data's length. This observer trusts the sender of a `temperature` notification to be a sensor, reads its name, and prints the reading after checking that the data is the size of a `double`. Section 1.3 says exactly what may be trusted about each of the four; for now, note the check on `n->len`, which is this observer declining to believe that a notification named `temperature` must carry a `double`.

The heater is fussier. It was given, as its own pointer, a record that says which sensor it watches, and it ignores the rest:

```
static void on_heater(const notification * n, void * ctxt)
{
   struct heater * h = (struct heater *)ctxt;
   double t;
   int on;
   if (n->sender != h->watches || n->len != sizeof t) { return; }
   t = *(const double *)n->data;
   on = t < 19.0 ? 1 : t > 21.0 ? 0 : h->on;       /* hysteresis */
   if (on != h->on) {
      h->on = on;
      printf("  heater  %s\n", on ? "on" : "off");
      Post(notify_default(), "heater", h, &on, sizeof on);
   }
}
```

Two things here are worth stopping on. The first is the test `n->sender != h->watches`: the heater filters by identity, comparing the sender pointer with the one sensor it cares about, which is how a program subscribes to *one* publisher of a name rather than all of them, and it costs one comparison. The heater never *dereferences* the sender. It does not need to know that a sender is a `struct sensor`; it needs to know only whether this is the sender it was told to watch. The second is the hysteresis, which is not about notifications at all but about the kind of bug this program would otherwise have: a heater that switched on below 20 and off above 20 would, in a room hovering at 20.0, switch on and off on every reading and post a notification each time. Two thresholds give the state a band in which it does not change, and almost every observer that turns readings into state wants some version of this.

When the heater does change, it *posts* a notification of its own, named `heater`, with itself as the sender and its new state as the data. An observer posting is completely ordinary, and it is how a program grows layers: raw readings in, a decision out, and whoever cares about the decision listens for `heater`

without ever seeing a temperature. The logger listens to every name there is, and counts:

```
static void on_log(const notification * n, void * ctxt)
{
   int * logged = (int *)ctxt;
   *logged += 1;
   printf("  log     %s\n", n->name);
}
```

The name Λ in place of a name means *every name,* which is what makes a logger, a tracer or a debugging monitor possible without touching the parts being watched. A sensor reading is a post, and the poster learns how many observers the post reached:

```
static int reading(const struct sensor * s, double t)
{
   printf("sensor  %-7s %.1f\n", s->name, t);
   return Post(notify_default(), "temperature", s, &t, sizeof t);
}
```

Notice where `t` lives: it is a parameter, on this function's stack, and its address is handed to `Post`. That is allowed, and Section 1.3 explains the rule that makes it allowed. Finally `main` registers the three observers, reads the sensors a few times, removes the display halfway, and reports:

```
coop_open(&sys, 0);                  /* observers live on a scheduler */
display = notify_observe(c, "temperature", NULL, on_display, NULL);
notify_observe(c, "temperature", NULL, on_heater, &heater);
notify_observe(c, NULL, NULL, on_log, &logged);

reading(&indoor, 20.5);
reading(&outdoor, 4.0);
reading(&indoor, 18.5);              /* the heater comes on */
reading(&indoor, 19.5);              /* and stays on */
reading(&indoor, 21.5);              /* and goes off */
notify_unobserve(c, display);
printf("(display removed)\n");
reached = reading(&indoor, 22.0);
```

The first line is the one piece of the program that is not about notifications. A center will only register an observer on a thread that has a *scheduler*—a `coop_system`, the subject of Chapter 2—because the scheduler is how the center knows which thread an observer belongs to, and Section 1.4 says why it needs to know. Nothing in the thermostat runs on the scheduler; for this chapter it is a line of setup, and `coop_close` at the end undoes it.

1.2. What it prints

```
sensor  indoor  20.5
  display indoor 20.5
  log     temperature
sensor  outdoor 4.0
  display outdoor 4.0
  log     temperature
sensor  indoor  18.5
  display indoor 18.5
  heater  on
  log     heater
  log     temperature
sensor  indoor  19.5
  display indoor 19.5
  log     temperature
sensor  indoor  21.5
  display indoor 21.5
  heater  off
  log     heater
  log     temperature
(display removed)
sensor  indoor  22.0
  log     temperature
8 notifications logged; the last reading reached 2 observers
```

The transcript already shows most of what there is to know about a notification center, and six lines of it are worth checking against the theory.

Each post reaches its observers in the order they registered. Display, heater, log: the order of the three `notify_observe` calls.

The outdoor reading reaches the display and the log but not the heater. The heater's sender check declined it. The center did not decline it—the notification was delivered, and the observer chose to do nothing, which is why the poster's count and the logger's count both include it.

The reading of 18.5 makes the log print `heater` *before it prints* `temperature`. This surprises people, and it is the right behaviour. The heater's post is an ordinary function call made from inside the heater's own observer; it runs to completion, log and all, before the outer post continues down its list to the logger. Posts nest, exactly the way calls nest, because on one thread that is what they are.

The reading of 19.5 changes nothing, so the heater says nothing: 19.5 is inside the band.

After the display is removed, the last reading reaches two observers rather than three, and the return value of `Post` says so. A program can use that count; a sensor with no listeners at all might switch itself off.

The logger counted eight notifications: six readings and two changes of the heater. The program's exit status checks that count and the count of two, which is how `make check` notices if this chapter ever stops being true.

1.3. Names, senders, data

The record an observer receives is

```
typedef struct notification
{
  const char * name;
  const void * sender;          /* an identity, never dereferenced here */
  const void * data;            /* a copy when delivered across threads */
  size_t len;
} notification;
```

and each field has a rule. The rules are what make the center usable by parts that do not trust each other.

The *name* is the address. Observers register for a name, or for every name with Λ; a post names exactly one; the center compares them as strings. Names are the whole of the agreement between two parts that share no header, so they deserve the care you would give a function name. The programs in this book use bare words for facts about the world (`temperature`, `heater`) and dotted words for messages within a subsystem (`server.quit`, `work.request`, `grant.2.1`). The dot means nothing to the center; it is punctuation for humans.

The *sender* is whatever the poster says it is—usually a pointer to the object on whose behalf the notification is posted—and the center treats it as a number. It is never dereferenced by the center, so it may be any pointer at all, including one to something that no longer exists by the time an observer looks. An observer that dereferences it, as the display does, does so on the strength of a convention about the name: every poster of `temperature` in this program is a live sensor. An observer that only compares it, as the heater does, needs no convention and cannot be wrong. Both uses are common, the second is the safer, and the first is the kind of promise worth writing down beside the name.

The *data* is `len` bytes at `data`, and here the center takes a position: the bytes are valid only during the observer's call, and they are *copied* whenever a delivery crosses to another thread, so that a poster may build them in a local variable—as `reading` does with `t`—and forget them the moment `Post` returns. In the thermostat every delivery is a direct call and the observer reads the poster's own variable; in a program with threads it reads a copy. Either way the pointer is dead when the observer returns, and an observer that wants to keep the data keeps its own copy.

The *length* is the observer's guard. A name promises nothing about what it carries; the center will happily deliver a notification named `temperature` carrying a `struct sockaddr`, because it has no idea what either is. Every observer in this book that reads `data` checks `len` first, and the ones that do not read `data`

ignore both. This costs a comparison and turns a class of crash into a silently ignored message.

1.4. Who runs an observer, and when

The thermostat's observers run inside `Post`, on the thread that posted, as ordinary calls. That is the simplest delivery and, in a program that lives on one thread, the right one. It is not the general case, and the `queue` argument of `notify_observe` is where the general case enters.

Some words are needed first, and they are the words of the rest of the book. A *thread* is what the operating system runs: a program has one at the start and may make more (Chapter 4). Each thread that takes part in the library has a *scheduler*, one `coop_system`, which keeps that thread's *contexts*—functions with stacks of their own, which take turns on the thread (Chapter 2)—and runs them in *rounds,* resuming each context that has something to do and then, when none has, waiting. What the scheduler waits *in* is a *queue,* one `evio`: the kernel's event queue for that thread (Chapter 3), which reports when a socket has something to say, and which also serves as the thread's *letter box*—a place where other threads may leave a function to be run on this thread, between two of its rounds. A thread need not have a queue; the thermostat has none. But a thread that talks to the network, or that other threads must be able to reach, has exactly one.

Now the rule. *An observer runs on the thread it was registered on, and on no other.* An observer registered with a queue is delivered to through that queue: the delivery is a function left in the queue's letter box, and the observer runs on the queue's thread, between rounds, with a copy of the data. An observer registered without a queue is called synchronously, inside `Post`, on the posting thread—which keeps the rule only when the posting thread is the observer's own. So a synchronous observer hears posts made from its own thread and no others; posts from other threads do not reach it and are not counted in the return value of `Post`. The thermostat, on one thread, hears everything.

Two consequences deserve a sentence each. First, a queued observer on the poster's *own* thread is also deferred: the delivery goes through the queue and runs between rounds, not inside `Post`. That is deliberate. A callback that ran inside `Post` would run inside whatever the poster was doing, possibly with the poster's data half-updated; a callback that runs between rounds runs with every context of the thread parked, which is the same peace and quiet a context has when it runs. Second, the rule is what makes the data a copy: a poster on one thread cannot keep its local variable alive until an observer on another thread has read it, so the center copies the data when it makes the delivery and frees the copy when the callback returns.

The rule is also the reason a program built this way can share state between threads without a lock, and the argument is short enough to give now and use

for the rest of the book. Whatever an observer touches is touched on one thread only, because no other thread ever runs the observer. State that several threads care about can therefore be given to one thread, as a set of observers, and the others reach it by posting; the state has an owner, and the owner runs one message at a time. Chapter 4's server keeps its log this way and Chapter 5's fetch farm keeps everything this way.

1.5. Order, nesting, and removal

Three promises about *when* things happen. They are short, and each one is something a program ends up depending on.

Two observers of the same name on the same thread are called in the order they registered, and deliveries queued to one observer arrive in the order they were posted. Nothing is promised about the relative order in which *different* observers on *different* threads see one notification, and a program that needs such an order should not be using a broadcast mechanism to get it.

A post made from inside a callback completes before the outer post continues, which the transcript showed. A callback may do anything at all: post, register a new observer, remove itself, remove the observer that is about to be called next. No lock of the center's is held while a callback runs, so there is no rule to remember here.

After `notify_unobserve` *returns on the observer's own thread, the callback will not run again,* and a notification that was queued for that observer before the removal is dropped rather than delivered. That qualifier is doing real work. On one thread, deliveries and the removal are serialised—the deliveries run between rounds, the removal runs in a context or between rounds, and neither can be halfway through when the other starts—so once the call returns, nothing can be pending. The observer's `ctxt` may therefore be freed as soon as `notify_unobserve` returns, provided the call was made on the observer's thread, which is why the thermostat can keep its heater record on the stack of `main`. From another thread the guarantee is weaker: a callback that had already begun may still be running when the removal returns, and the program must arrange for `ctxt` to outlive it. If you need the strong form, remove the observer from its own thread, which is one line and always possible. The waiting calls of the next section depend on the strong form and get it by construction.

1.6. The other way to hear

An observer is a callback, and a callback is a strange place to put an activity. Here is the shape it forces:

```
static void on_done(const notification * n, void * ctxt)
{
   struct state * st = (struct state *)ctxt;
   st->phase = FINISHED;                 /* ... and then what? */
}
```

Whatever happens next cannot happen here. It happens in some other function, driven by `st->phase`, and the sequence of the activity—first this, then wait, then that—is spread across a structure and a switch statement instead of being written down in order. Callbacks are exactly the thing the rest of the library exists to avoid, and the center offers a second way to hear, which is the one the rest of this book uses most. It is to *wait* for a notification: to say, in the middle of a function, "continue when `client.done` has been posted," and to continue there, on the next line, with the notification's data in hand:

```
notify_wait(c, "client.done", &bytes, sizeof bytes, -1);
printf("finished with %ld bytes\n", bytes);
```

A function that can stop in the middle of a line and continue later is a context, and contexts are the subject of the next chapter, which explains this call properly, together with the one trap that goes with it. What matters here is that the wait is not a new mechanism: it registers an ordinary observer, parks, and removes the observer again, on its own thread, which is why the strong form of the removal guarantee is what makes it safe.

The rule of thumb for choosing between the two ways is short. A callback suits a recipient that is a *thing* that reacts: a display that redraws, a log that appends, a counter that counts. Such a recipient has no sequence of its own; it has a state, and the notification changes it. A wait suits a recipient that is an *activity* that has reached a point where it cannot continue: it has asked for something, or it has work that only starts when a message arrives. The two mix freely, and the largest program in this book has both on the same thread.

1.7. What we gained, and what it cost

Look again at the ordering problem from the opening of this chapter, the display showing a new temperature beside a stale heater state. The thermostat still has that ordering, and the center did not magically fix it. What changed is that the fix is now available and cheap: the display can observe `heater` as well, or the heater can be registered first, or, best of all, the display can be told that it is showing two independent facts that arrive at their own times, which is what it was always doing.

The cost of the center is real and should be stated. A direct call is a call; a synchronous delivery is a string comparison and an indirect call. A post to m queued observers costs m small allocations and copies of the data, m acquisitions of the queues' mutexes, and m wake-ups on the observers' threads, each a system

call. For six observers and a reading every few seconds this is free. For a million events a second it is not, and a program with a hot path like that should call directly on the hot path and post the summaries. The center is a mechanism for the events a program has dozens of, not the bytes it has millions of.

The other cost is a genuine loss: you can no longer find every listener by asking your editor for the callers of a function. The names are the coupling now, and they are only as good as your discipline about them. Choose them the way you would choose the names of functions, and keep the list of names a program uses somewhere a reader can find it; Chapter 5 shows what such a list looks like for a program with five names.

CHAPTER TWO

THE COOPERATIVE SCHEDULER

THE CALLS OF THIS CHAPTER

`int coop_open(coop_system * s, size_t stack_size);`
`void coop_close(coop_system * s);`
Make a scheduler on the calling thread, and release it. A `stack_size` of 0 means `COOP_DEFAULT_STACK`, which is 128 kilobytes. Closing frees every context, finished or not, so nothing may still have an operation in flight.

`typedef void (*coop_entry)(int * cancel, void * ctxt);`
The shape of a context. It ends by returning. The flag `*cancel` becomes nonzero when somebody calls `coop_cancel` on it.

`coop_context * coop_spawn(coop_system * s,`
`coop_entry entry, void * ctxt);`
Start a context running `entry(&cancel, ctxt)` on `s`, from `s`'s own thread. It first runs in the next round. The pointer is valid only while the context cannot have ended.

`void coop_cancel(coop_context * c);`
Ask a context to stop: set its `*cancel`, and cut short a sleep. A context parked in a wait is *not* woken; it sees the flag when the wait returns.

`void Relinquish(void);`
Hand the thread back to the scheduler; due again in the next round. From the scheduler itself, does nothing.

`void coop_sleep_ms(int ms);  void coop_sleep_ns(int64_t ns);`
Park this context until a time. Ends early if the context is cancelled. Called outside a context, sleeps the thread.

`int64_t coop_now_ns(void);`
A monotonic clock in nanoseconds. It never returns 0 and never goes backwards, so it is the right clock for measuring intervals and for deadlines.

`coop_context * coop_current(void);`
`coop_system * coop_system_here(void);`
The context running on this thread, or Λ when the scheduler itself is running; and this thread's scheduler, or Λ. The scheduler's `user` field is yours, and is how a context finds its thread's own data.

`int coop_run(coop_system * s);`
`int coop_run_until(coop_system * s, const int * stop);`
Run rounds until no context is left; or until `*stop` is nonzero, idling even with an empty list. Both return 0, or −1 if nothing could ever make progress.

`int coop_step(coop_system * s);` `void Resume(coop_context * c);`
`void coop_reap(coop_system * s);`
One round, returning the number of contexts left or −1; run one context until it parks or ends; free the contexts that have ended. For a loop of your own (Section 2.9); `coop_run` is made of them.

`void coop_wait(void);`
`int coop_wait_until(int64_t deadline_ns);`
`void coop_wake(coop_context * c);`
Park this context until somebody wakes it, optionally with a deadline given as a `coop_now_ns` value (0 means none); the second form returns 0 if woken and 1 if the deadline passed first. Wake a waiting context, from its own thread only.

`void coop_set_idle(coop_system * s, coop_idle_fn f, void * arg);`
Install what the scheduler should do when it has nothing to run. The queue of Chapter 3 installs itself; you need this only for a wait of your own devising.

`long notify_wait(notify_center * c, const char * name,`
`void * buf, size_t cap, int timeout_ms);`
From a context: park until a notification named `name` is posted, copying up to `cap` bytes of its payload into `buf`. Returns the payload's *full* length, which may exceed `cap`; or −`EVIO_ETIMEDOUT` after `timeout_ms` milliseconds (−1 waits forever); or −1 outside a context. A notification posted before the call is not seen.

`int notify_expect(notify_center * c, const char * name,`
`void * buf, size_t cap, notify_expectation * e);`
`long notify_await(notify_expectation * e, int timeout_ms);`
`void notify_forget(notify_expectation * e);`
The same wait in two halves, for a context that must be listening before it asks: start listening now, into an expectation on the caller's own stack; then park until the notification arrives, or return at once if it already has, and stop listening. The third call stops listening without waiting. Expect and await from the same context.

Here is a small problem. A device has two lights that blink at different rates—say one every 500 milliseconds and one every 300—and a computation to get on with in between. Write the program.

With one thread and no library, the shape is forced. You cannot sleep, because sleeping stops the computation. So the program keeps the state of each light in a structure, looks at the clock, and decides:

```
for (;;) {
   int64_t now = clock_ms();
   if (now >= red.next) {
      red.on = !red.on;  show("red", red.on);
      red.next = now + 500;
   }
   if (now >= green.next) {
      green.on = !green.on;  show("green", green.on);
      green.next = now + 300;
   }
   compute_a_bit();
}
```

This works, and it is worth being precise about what is wrong with it, because "I don't like it" is not an argument. The state of each light is outside the light: `red.on` and `red.next` are not local variables of anything but fields of a structure that the loop maintains, and a light that wanted a more interesting pattern—on for 100 milliseconds, off for 400, then a double blink—would need a state variable and a switch, with the sequence spread across the cases rather than written down in order. The loop knows about every activity, so that adding a third light, or something that is not a light at all, means editing the loop, which becomes the place where everything in the program meets. And the program never sleeps: it spins, reading the clock, whether there is work or not, which on a laptop is a warm laptop and on a battery is a flat one.

Now here is the same light, written as a function:

```
static void blink(int * cancel, void * ctxt)
{
   struct led * led = (struct led *)ctxt;
   while (!*cancel) {
      led->on = !led->on;
      trace(led->name, led->on ? "on" : "off");
      coop_sleep_ms(led->period_ms);       /* cut short by coop_cancel */
   }
   led->on = 0;
   trace(led->name, "off, stopped");
}
```

A loop, with a sleep in it, exactly as you would write it if this light were the only thing in the world. There is no `next` field, because the moment to wake next is encoded in where the function has got to. There is no central loop to edit. And the program does sleep: while both lights are between edges, the thread sleeps and the process uses no processor at all. The thing that makes this possible is that the function has *a stack of its own*.

2.1. A context

A *context* is a function that has been given a stack of its own, and can therefore be left in the middle of its work and returned to later, at the same place, with all of its local variables intact. A thread is a context that the operating system switches for you, at moments of its own choosing; the contexts in this book are switched only when they ask, which is what *cooperative* means, and the difference is the whole point, in both directions.

Nothing interrupts you. Between two statements of a context, nothing else on that thread runs. There is no lock between contexts on one thread, ever, because there is no moment at which two of them are both in the middle of something; a data structure shared by five contexts on one thread needs no protection at all, and the state of a half-finished operation is where C keeps it anyway, in local variables.

Nothing rescues you. A context that computes for 200 milliseconds without asking to be switched holds the thread for 200 milliseconds, and every other context on it is late. Section 2.5 is about what to do instead.

To the program a context is its entry function, which `coop_spawn` gives a stack and a place on the scheduler's list, which first runs in the next round, and which ends by returning. Its `ctxt` is the pointer the spawner passed, which is how one function serves as two different lights. Its `*cancel` becomes nonzero when somebody calls `coop_cancel` on the context; it is a request, and the context decides when to honour it, as the lights do at the top of their loop.

Two consequences follow from the fact that contexts on a thread never run at the same time, and both are used constantly. First, a context may hold a pointer into another context's stack and use it safely for as long as that context is parked. This is not a trick; it is what lets the waiting calls of this chapter and the operations of the next keep their records on the stack of the context that is waiting, rather than in allocated memory. Second, the C library is used from all contexts of a thread as if from one: `errno`, `strtok` and the stdio buffers are shared in the ordinary sequential way, and thread-local variables are—as their name says—per thread, not per context, so a context must not expect one to follow it.

2.2. The blinker

The program is `examples/blinker.c`, and it has one thread, four contexts, and no kernel queue at all: two lights, a computation, and a supervisor that stops the lights after four seconds. It exists to show the scheduler's timing and nothing else, and it is the smallest program in which every feature of the scheduler can be seen at work.

A light is the function above, with two extra fields so that it can grade its own performance. Before each sleep it notes when the sleep should end, and afterwards it records how late it was:

```
due = coop_now_ns() + (int64_t)led->period_ms * 1000000;
coop_sleep_ms(led->period_ms);
if (!*cancel) {
   long late = (long)((coop_now_ns() - due) / 1000);
   if (late > led->worst_us) { led->worst_us = late; }
}
```

The red light has a period of 500 ms and the green one 300 ms, so their edges coincide every 1.5 seconds and drift apart otherwise, which makes a mistake in either easy to see. The computation counts the primes below a limit by trial division, and hands the thread back every 256 numbers:

```
static void compute(int * cancel, void * ctxt)
{
   long limit = *(long *)ctxt, count = 0, n;
   char text[64];
   for (n = 2; n < limit && !*cancel; n += 1) {
      long d;
      int prime = 1;
      for (d = 2; d * d <= n; d += 1) {
         if (n % d == 0) { prime = 0; break; }
      }
      count += prime;
      if ((n & 255) == 0) { Relinquish(); }   /* the lights get a turn */
   }
   snprintf(text, sizeof text, "%ld primes below %ld", count, limit);
   trace("cpu", text);
}
```

The second of the three capitalised verbs is `Relinquish`: the running context hands the thread back, and is due again in the next round. The job never sleeps and never waits; it only relinquishes, and would otherwise hold the processor for most of a second. The supervisor sleeps and then cancels:

```
static void supervise(int * cancel, void * ctxt)
{
   struct supervisor * sv = (struct supervisor *)ctxt;
   (void)cancel;
   coop_sleep_ms(sv->after_ms);
   trace("super", "stopping the leds");
   coop_cancel(sv->leds[0]);
   coop_cancel(sv->leds[1]);
}
```

and `main` opens a scheduler, spawns the four, runs until the list is empty, and closes:

```
coop_open(&s, 0);
t0 = coop_now_ns();
sv.leds[0] = coop_spawn(&s, blink, &red);
sv.leds[1] = coop_spawn(&s, blink, &green);
coop_spawn(&s, compute, &limit);
coop_spawn(&s, supervise, &sv);
r = coop_run(&s);                          /* until every context ends */
coop_close(&s);
```

With the limit at 2 500 000, the output begins

```
[  0.000] red    on
[  0.000] green  on
[  0.300] green  off
[  0.500] red    off
[  0.600] green  on
[  0.881] cpu    183072 primes below 2500000
[  0.900] green  off
[  1.000] red    on
[  1.201] green  on
[  1.501] red    off
[  1.501] green  off
```

and ends

```
[  4.000] super  stopping the leds
[  4.000] red    off, stopped
[  4.000] green  off, stopped
[  4.000] main   all contexts ended
red    8 edges, worst 0.357 ms late
green 14 edges, worst 0.200 ms late
```

Four things in the transcript are worth checking against what follows. Both lights print at 0.000, red before green: they were spawned runnable and the first round resumed them in the order they were spawned, before either had slept. Every edge during the computation landed on time, which is the claim the whole program exists to demonstrate: the prime count held the processor for the better part of a second, and in that time the lights produced five edges, the latest of which was a third of a millisecond behind where it belonged, because the computation's slices are short—256 numbers, a few dozen microseconds—and every round between slices tested both lights' timers. The count is right: there are 183 072 primes below 2 500 000, which you can confirm with a sieve. And the lights stopped at 4.000, not at their next edge, because `coop_cancel` makes a sleeping context due immediately, so both woke in the round after the supervisor's call, saw `*cancel` at the top of the loop, printed their last line, and returned. The program's exit status checks the two figures on the last two lines against a limit of 25 milliseconds, so the central claim is one that `make check` tests rather than one you have to take on trust.

2.3. Where the time goes

The interesting measurement is not in the program's output at all. Running it under `time` reports

```
0.88s user  0.01s system  22% cpu  4.002 total
```

The user time is the prime count and nothing else, which the transcript confirms: the count finished at 0.881. For the remaining three seconds the process was asleep between light edges, because a scheduler with nothing due and no queue sleeps the thread until the next timer. Compare that with the state-machine loop at the top of this chapter, which would report 4.0 seconds elapsed and 4.0 seconds of user time, because it never stops looking at the clock: the same output, and three seconds of a processor given away for nothing. That is the entire argument for a scheduler that knows when the next thing is due, and it is why every parking call in the library takes a deadline rather than being polled.

2.4. The round

A thread's contexts take turns in *rounds*. One round walks the list of contexts once, in the order they were spawned, and resumes every context that is due; each one runs until it parks or ends. Contexts that ended are then freed. Finally, if nothing is left to run, the thread waits—in the kernel queue if it has one, on a plain sleep if it does not—until the earliest timer, or until an event arrives.

That paragraph is the entire behaviour of the scheduler, and four things in it are promises a program can rely on. *Every due context runs once per round,* so a program cannot starve one activity by having another that is always ready. *The order is the order they were spawned,* and it does not change; the blinker's two lights print in that order at time zero for this reason. *A context spawned during a round runs in that round if the walk has not yet passed the end of the list, and in the next one otherwise*—either way, once. And *nothing else runs between two statements of a context*: observers, posted functions and the queue's own work all happen *between* rounds, never in the middle of one. The last promise is the one that pays for itself most often. It is why an observer and the contexts on its thread can share a structure with no lock, and why the fetch farm of Chapter 5 keeps its whole tally in a plain `struct` that three observers write and nothing guards.

A context is in one of four states, and every call in the library moves it between them:

state	*due again when*
runnable	always: the next round
sleeping	its time arrives, or it is cancelled
waiting	somebody wakes it; a deadline may be set too
finished	never; freed at the end of the round

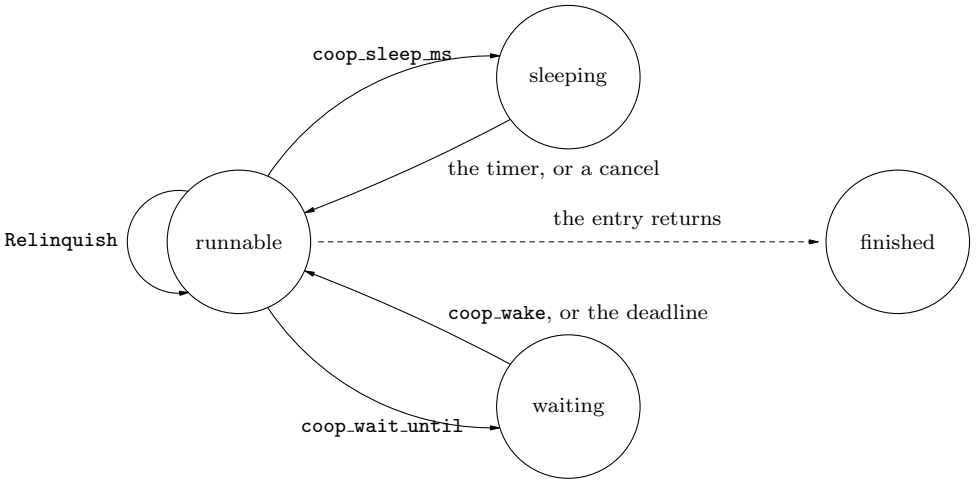

Fig. 2. The states of a context and what moves it between them. A context is born runnable and leaves the list by returning.

A context enters the runnable state at birth and re-enters it by `Relinquish`; it goes to sleeping through `coop_sleep_ms`, and to waiting, with or without a deadline, through everything else that parks—the notification waits of Section 2.7, the operations of the next chapter, and `coop_wait` itself. A cancelled context that was sleeping becomes due at once; a cancelled context that was waiting is left where it is, for the reason Section 2.6 gives. The blinker's lights spend their lives between runnable and sleeping; the connection contexts of Chapter 4 between runnable and waiting.

There are three ways to run the rounds, and a program of any size uses two of them.

Run to completion. The call `coop_run` repeats the round until no context is left, and returns 0. This is the shape of a program that has a job to do: the blinker, the client of Chapter 3, the main thread of a tool.

Run until told to stop. The call `coop_run_until` repeats the round until a flag becomes nonzero, and does not mind an empty list. That last clause is what makes it the shape of a service: a worker thread's list is empty precisely while it waits for the next piece of work, and a `coop_run` there would return immediately. The flag is set from an observer or a posted function on the same thread, so no lock is needed for it, and the standard ending of every such thread in this book is the pair

```
coop_run_until(&sys, &stop);        /* the working life */
coop_run(&sys);                     /* let live work finish */
```

whose second line runs rounds until every context has ended, so that nothing is in the middle of an operation when the scheduler is closed. Chapter 3 says why that matters.

Run rounds yourself. One round is `coop_step`, and a loop around it is either of the above. Calling it directly is what you do when the rounds have to be paced by something else—a frame clock, an external event loop, a test that wants to run exactly three rounds and look at the result. Section 2.9 goes further and writes the round itself.

A return of -1 from any of the three means *deadlock*: nothing ran, no timer is pending, and there is no queue, so nothing can ever wake the contexts that are waiting. The scheduler says so rather than spinning or blocking forever. Note the last condition. With a queue attached, the same situation is not a deadlock at all but a perfectly healthy idle service, waiting in the kernel for a connection; so a -1 from a program that has a queue means something has genuinely gone wrong. The blinker's `main` prints "deadlock" if `coop_run` returns -1, and it cannot happen there, because nothing in the blinker waits.

2.5. The polite loop

A `Relinquish` every 256 numbers is a number somebody chose, so it is worth knowing how to choose it. The rule is: *the longest a context goes without relinquishing is the worst lateness it inflicts on everybody else.* A slice of 256 trial divisions is a few dozen microseconds, so nothing else on the thread is ever more than a few dozen microseconds late, which is invisible for lights and comfortable even for a program driving hardware. A slice of 4096 would be sixteen times worse and still fine here. A slice of "one whole computation" is the bug you started with, and it is worth seeing what it looks like. Take the `Relinquish` out and run the same program, and the two numbers become

```
red    9 edges, worst 352.324 ms late
green 15 edges, worst 552.324 ms late
```

which is the computation's whole run, and the program takes 4.85 seconds to do four seconds of work.

The cost runs the other way: each `Relinquish` is a stack switch and a walk of the context list, well under a microsecond, but at 256 numbers per slice the program does about ten thousand of them a second and you can measure that if you look hard. The useful shape is a slice long enough that the switch is noise—say a hundred microseconds of work—and short enough that nobody else notices. Anything within a factor of ten of that is fine, which is why nobody tunes it.

One more consideration catches people. A relinquish makes a context due *in the next round,* not later; a context that only relinquishes never sleeps and never waits, so the scheduler always has something to run, so the thread never

idles. That is correct—the computation genuinely does want the processor—and the library takes care of the consequence: a thread that never idles still drains its letter box every round and still looks at the kernel regularly, so a notification posted to it from another thread arrives promptly rather than waiting for the computation to end. A program that would rather not have the computation on this thread at all puts it on a thread of its own, which is Section 4.8.

2.6. Cancelling, and ending

A context ends by returning from its entry function. That is all; there is no exit call, and no way to end another context directly.

Cancellation is a request, not a command. A call to `coop_cancel` sets the flag the entry received as `*cancel`, and, if the context was sleeping, makes its timer due so that the sleep ends now instead of at its natural time. The context notices when it chooses to, which for the lights is at the top of the loop and for the computation is once per number.

A *waiting* context—one parked in a notification wait, or in one of the next chapter's operations—is deliberately left alone by `coop_cancel`: the flag is set, but nothing wakes it. Its wait belongs to an operation that has its own idea of what a wake means, and a context that was woken for no reason would conclude that its event had happened, look, find nothing, and park again, having learned nothing about the cancellation. The convention is instead that *a context which must be cancellable while it waits gives its waits a deadline, and tests* `*cancel` *when they expire.* Chapter 4's acceptor does exactly this with a half-second deadline, and Chapter 5 puts a price on the deadline it chose.

Finally, a word about the pointers `coop_spawn` returns, because this is the one place where the library will let you hold a dangling pointer. A finished context is freed at the end of the round in which it ended, and the pointer dangles afterwards. The supervisor keeps pointers to the two lights, and that is safe *because the lights loop until cancelled and therefore cannot have ended.* It keeps no pointer to the computation, which may end at any moment, and needs none. The rule to carry away: hold a `coop_context *` only to a context that cannot yet have ended, which in practice means one that loops until cancelled or one you have not yet let run. If you need to know when one has ended, have it tell you—a flag, or a notification, as Chapter 3's client tells its heartbeat.

2.7. Waiting for a notification

Chapter 1 ended with a promise: that a function can say "continue when this has been posted" and continue there. A context is what makes that possible, and here is the smallest program that does it. One context bakes; the other waits to be told:

```
static void oven(int * cancel, void * ctxt)
{
   (void)cancel; (void)ctxt;
   trace("oven", "baking");
   coop_sleep_ms(300);
   Post(notify_default(), "bread.done", NULL, "sourdough", 10);
}

static void cook(int * cancel, void * ctxt)
{
   char kind[16];
   long n;
   (void)cancel; (void)ctxt;
   trace("cook", "waiting for bread.done");
   n = notify_wait(notify_default(), "bread.done", kind,
    sizeof kind, 1000);
   if (n < 0) { trace("cook", "gave up"); return; }
   trace("cook", kind);
}
```

The whole program is those two functions, a `main` that spawns the cook and then the oven and runs, and this:

```
[ 0.000] cook waiting for bread.done
[ 0.000] oven baking
[ 0.300] cook sourdough
```

Both contexts ran in the first round, in the order they were spawned. The cook stopped inside `notify_wait`; the oven stopped inside `coop_sleep_ms`; for three hundred milliseconds the thread had nothing to do and slept. Then the oven woke, posted, and the cook continued on the line after its wait with the payload in `kind`. Underneath, the wait registered a temporary observer with the name, parked the cook, and—when the observer was called, from inside the oven's `Post`—copied the payload into the cook's buffer, woke the cook, and removed the observer again. Nothing new was needed.

Three details of the return value matter in real programs. It is the payload's full length, not the number of bytes copied, so that a caller which must have the whole payload can compare the result with what it asked for and notice a buffer that was too small, rather than silently working with half a message. A timeout is not an error: `−EVIO_ETIMEDOUT` means the notification did not arrive in time, and a program is free to try again, give up, or treat the timeout as a tick—Chapter 3's heartbeat does the last of these, ten times a second. And a notification posted before the call is not seen, because the center delivers to the observers that exist at the moment of the post and keeps nothing for observers yet to come. That last fact is the one trap in the whole mechanism, and it deserves the rest of this section.

Listen before you ask. The order "register, then wait" is fine when the notification is going to arrive from somewhere in the future. It is not fine when *you are about to cause it.* Here is the bug. A context wants something from another part of the program; it posts a request, then waits for the answer:

```
Post(c, "work.request", me, &id, sizeof id);       /* ask */
notify_wait(c, "grant.2", NULL, 0, -1);            /* wait for it */
```

If the other part is quick—and it may be another thread, already running, microseconds away—the answer can be posted in the gap between those two lines. It is delivered to nobody, and the waiter waits forever. This is a lost wake-up, it happens perhaps one time in ten thousand, and it does not reproduce. The fix is to split the wait in two and start listening *first:*

```
notify_expect(c, "grant.2", NULL, 0, &e);          /* 1. listen */
Post(c, "work.request", me, &id, sizeof id);       /* 2. ask */
notify_await(&e, -1);                              /* 3. wait */
```

Now the gap is closed: whenever the answer is posted, whether before or after step 3 begins, an observer exists to receive it, and `notify_await` finds it already recorded and returns at once. A plain `notify_wait` is exactly `notify_expect` followed by `notify_await`, and it is the right call whenever you are not the cause of what you are waiting for. The expectation record lives on the caller's stack, and that is safe for a specific reason: the observer that `notify_expect` registers writes into it, and `notify_await` removes that observer before returning, on the observer's own thread, which is the strong form of the removal guarantee from Section 1.5. Two rules go with the pair. Expect and await from the same context, because the expectation names the context that will be woken. And follow every `notify_expect` with exactly one `notify_await` or one `notify_forget`, on every path out of the function, error paths included; an expectation left registered after its frame has gone is the one way to make this mechanism unsafe.

Name the answer. A request-and-answer exchange needs the answer to be addressed to the asker, and the address is a name. When there is exactly one asker a fixed name will do; when there are twelve, each needs its own, or one answer wakes all of them. The simplest arrangement, and the one the fetch farm of Chapter 5 uses, is that *the request carries the name to answer on:* the asker posts a string such as `grant.2.1`, built once when it started, and the answering side posts to whatever name it was handed, so it needs no table of who is waiting for what.

Wait for a state, not an answer. There is a second, gentler version of the lost wake-up, worth knowing because the fix is different. Sometimes a context waits for a state rather than an answer—"tell me when the download has finished"—and the download may have finished before the waiter ever existed. No amount of listening first helps, because there was no moment at which the waiter could have listened. The remedy is to keep the state in a plain flag as well, and use the notification only as the wake-up:

```
while (!job->done) {
   notify_wait(c, "client.done", &bytes, sizeof bytes, 100);
}
```

The flag is the truth; the notification saves the loop from polling; the timeout bounds the damage if the notification is missed entirely. A loop of this shape is correct whether the state changed before the loop started, during the wait, or never, and it is how the heartbeat of the next chapter is written.

2.8. Waiting for anything else

The two calls `coop_wait` and `coop_wake` are the primitive underneath every parking call in the library—the notification waits above, the operations of the next chapter—and they are exported so that you can add a wait of your own. The protocol has three steps. The context records where the answer will go and who to wake, typically in a small structure on its own stack holding `coop_current()`. It waits in a loop over a flag: `while (!done) coop_wait();`. And whoever has the answer sets the flag and calls `coop_wake` on the recorded context, on that context's own thread.

The loop in the second step is not decoration. A context may in principle be woken by something other than the event it is waiting for, and the loop makes that harmless; the same discipline applies to condition variables, and for the same reason. The qualification in the third step is the rule of Chapter 4, and it is what makes the whole protocol lock-free: a wake is a plain write to a field that only this thread reads, and the answer to "was I woken, or did my deadline pass?" is read back from that field with no lock, correctly, because the writer and the reader are the same thread. If the answer is produced on another thread, that thread does not call `coop_wake`; it leaves a function in the right thread's letter box—Chapter 4's `evio_post`—and *that* function calls `coop_wake`. Section 4.8 does exactly this to hand a computation to a thread of its own.

2.9. Writing your own loop

The round described in Section 2.4 is a reference, not a law. A simulation with a fixed frame period wants a loop that begins each frame on the clock, runs what is due, and sleeps out the remainder; a game loop wants the same with rendering in the remainder. Both are a page:

```
for (;;) {
   int64_t t0 = coop_now_ns();
   coop_context * c, * next;
   for (c = sys.first; c; c = next) {
      next = c->next;
      if (!c->done && due(c, t0)) { Resume(c); }
   }
   coop_reap(&sys);
   render();
   sleep_until(t0 + PERIOD_NS);
}
```

The scheduler's list is public for exactly this purpose: `sys.first` and `c->next` walk it, `c->done` says whether a context has ended, and `c->wake_at` and `c->waiting` are what `due` reads. Three rules are all such a loop must keep, and each is something `coop_step` does for you. Capture the successor before resuming, because a context may spawn or end while it runs; skipping this is the one way to make a custom loop crash. Reap between rounds, not during the walk, because a finished context's record is freed by `coop_reap` and freeing it during the walk would free the stack the walk is standing on. And call `Resume` from the scheduler's own context, never from inside a context, because `Resume` moves the processor from the scheduler onto a context and calling it from within one would leave the scheduler's idea of where to return pointing at the wrong place.

The predicate `due` is yours, and this is where a custom loop earns its keep: a frame loop might run only the contexts whose timers fall inside this frame and leave the rest for the next one, which is a scheduling policy the general loop has no way to express. A loop of this kind that also wants network I/O calls the queue's idle hook itself, once per frame, as `sys.idle(&sys, 0, sys.idle_arg)`; the 0 means "look, but do not block", which is what a frame loop wants, and it collects whatever the kernel has ready and runs whatever other threads have posted, without giving up the frame. That one line is the whole of what the queue needs from a scheduler, which is why this library's I/O can live inside somebody else's main loop.

2.10. How much stack

The second argument to `coop_open` is the stack size every context of that scheduler gets, and 0 selects 128 kilobytes. That is enough for the socket calls, for `printf` with floating point, and for the programs in this book, none of which comes close. Two things make a program want more. Deep recursion is the obvious one. The other is a large local array: a context that declares an 8-kilobyte request buffer, as Chapter 4's does, has spent a sixteenth of its stack on one line.

Count the local arrays on the deepest path and leave a comfortable margin, because the stacks are unguarded, and a context that overflows walks off the end of its allocation without warning.

The memory is reserved rather than touched, so a thousand contexts at 128 kilobytes reserve 128 megabytes of address space and use only the pages they actually write—a hundred contexts that each use four kilobytes cost about four hundred kilobytes, not thirteen megabytes. Still, if a program is going to hold ten thousand connections, sizing the stack to what those connections need is worth the five minutes it takes.

CHAPTER THREE

KERNEL EVENT QUEUES

THE CALLS OF THIS CHAPTER

```
int evio_open(evio * io, coop_system * s);
    void evio_close(evio * io);
    evio * evio_here(void);
```
Create this thread's event queue on this thread's scheduler, and release it; and find the queue opened on the calling thread, or Λ. One queue per thread, opened after the scheduler and closed before it, when no context has an operation in flight.

```
evio_socket evio_socket_create(evio * io, int family, int type);
    int evio_attach(evio * io, evio_socket s);
    void evio_socket_close(evio_socket s);
```
A socket prepared for the queue, with `family` `AF_INET` or `AF_INET6` and `type` `SOCK_STREAM` or `SOCK_DGRAM`, or `EVIO_INVALID_SOCKET`; prepare a socket that arrived from elsewhere, on the thread that will use it; close a socket that no context is waiting on.

```
int evio_connect(evio * io, evio_socket s,
    const struct sockaddr * addr, socklen_t len, int timeout_ms);
```
Connect; 0 on success.

```
int evio_accept(evio * io, evio_socket listener,
    evio_socket * accepted, int timeout_ms);
```
Accept one connection into `*accepted`; 0 on success. The new socket belongs to no queue yet.

```
long evio_recv(evio * io, evio_socket s, void * buf,
    size_t len, int timeout_ms);
    long evio_send(evio * io, evio_socket s, const void * buf,
    size_t len, int timeout_ms);
```
Receive up to `len` bytes, returning the count, or 0 at end of stream; send *all* of `len` bytes and return `len`, without raising `SIGPIPE`. If the deadline passes after some bytes have gone the send returns how many, and −`EVIO_ETIMEDOUT` only if none did.

```
long evio_recvfrom(evio * io, evio_socket s, void * buf,
    size_t len, struct sockaddr * from, socklen_t * fromlen,
    int timeout_ms);
    long evio_sendto(evio * io, evio_socket s, const void * buf,
```

```
    size_t len, const struct sockaddr * to, socklen_t tolen,
    int timeout_ms);
```
One datagram in, with the sender's address; one datagram out.

```
int evio_resolve(evio * io, const char * host,
    const char * service, int family,
    struct sockaddr_storage * addrs, int cap);
```
Resolve a name to at most `cap` addresses in the resolver's order of preference, returning how many; runs on the pool of `fio.h`.

```
int evio_signal(evio * io, int signum, const char * name);
```
Deliver `signum` as a notification named `name`, posted on `io`'s thread between rounds, and suppress the signal's usual effect. On Windows `SIGINT` and `SIGTERM` are the console's Ctrl-C and close events.

```
long fio_call(evio * io, long (*fn)(void *), void * arg);
    fio_file fio_open(evio *, const char * path, int flags, int mode);
    long fio_pread(evio *, fio_file, void *, size_t, int64_t off);
    long fio_pwrite(evio *, fio_file, const void *, size_t, int64_t);
    int fio_close(evio *, fio_file); int fio_fsync(evio *, fio_file);
    int64_t fio_size(evio *, fio_file);
```
Run any blocking call, and read and write files, on a pool of threads so that the calling thread need not block; a context parks and its thread runs others meanwhile. Reads and writes name their offset; the pool is sized and joined by `fio_start` and `fio_stop`.

```
int fio_unlink(evio *, const char * path);
    int fio_rename(evio *, const char * from, const char * to);
    int fio_mkdir(evio *, const char * path, int mode);
    int fio_rmdir(evio *, const char * path);
    int fio_stat(evio *, const char * path, fio_info * out);
    int fio_readdir(evio *, const char * path, char * buf,
    size_t cap, int max);
```
A file's name rather than its contents, on the same pool: remove one, rename one (replacing the destination), make a directory and remove an empty one, stat a path into a `fio_info` (`size`, `mtime_ns`, `type`), and read a directory's entry names into `buf` as `evio_resolve` fills addresses.

```
int evio_last_error(void);
    const char * evio_strerror(int err);
```
The platform's last error (`errno` or `WSAGetLastError()`), and the text for a positive error code—which is the negation of what the operations return.

The blinker waited for timers and the cook waited for a notification, and both were things the scheduler could see for itself. Most programs are not so lucky. A program that fetches a page from the network sends a request and then has nothing to do until the answer comes, which may be a millisecond or ten seconds later, and during that time it would like to be useful—or at least to look alive. The classical C program calls `recv` and stops dead until the bytes arrive:

```
s = socket(AF_INET, SOCK_STREAM, 0);
connect(s, addr, len);
send(s, request, n, 0);
n = recv(s, buf, sizeof buf, 0);       /* the program stops here */
```

That is four lines long, it is correct, and it is the reason people write servers badly. The fourth line stops the thread, not for a scheduling quantum but for as long as the peer takes; if the program had a progress bar to draw, a second request to send, or a user pressing a key, none of it happens. If it wants to do anything meanwhile it needs a second thread, or a signal, or to give up the simple shape of "send, then receive" altogether and become a machine that reacts to events.

The kernel has an answer to this, and every operating system in this book gives the same answer under a different name. It keeps, on the program's behalf, a *queue* of the things that have happened to the program's sockets, and lets a thread sleep until the queue is not empty. A thread that waits in such a queue waits for *everything at once*—every socket it has, a timer, and a knock from another thread—and is woken by whichever happens first. The library's `evio` puts one face on the three queues, and this chapter is about that face: the six calls that look like the blocking socket calls they replace, what their return values mean, how timeouts behave, and a complete program that uses five of them. It ends with the things the queue *cannot* hold—files, and the resolving of names—which go instead to a small pool of threads, and a second program that copies a file across a socket with both at once.

3.1. The client

The chapter's program is `examples/client.c`, which takes a host, a path, a port and the address of a DNS server, and defaults to `example.com`, `/`, 80 and 8.8.8.8. Its `main` opens a scheduler and a queue, spawns two contexts, and runs the scheduler until both have ended:

```
coop_open(&s, 0);                      /* a scheduler on this thread */
evio_open(&io, &s);                    /* and a queue */
s.user = &io;                          /* so contexts can find it */
coop_spawn(&s, client, &job);          /* two activities */
coop_spawn(&s, heartbeat, &job);
coop_run(&s);                          /* until both have ended */
evio_close(&io);
coop_close(&s);
```

That is the shape of every thread in this book that does network work: a scheduler, a queue, some contexts, a loop, and the two closes in the reverse order of the opens. The first context does the work. It resolves the host, unless the host is already a dotted address, and then downloads the path; when it is done it sets a flag and posts a notification:

```
static void client(int * cancel, void * ctxt)
{
   struct job * job = (struct job *)ctxt;
   evio * io = (evio *)coop_system_here()->user;
   struct in_addr addr;
   (void)cancel;
   job->status = 1;
   if (inet_pton(AF_INET, job->host, &addr) != 1) {
      if (resolve(io, job->dns, job->host, &addr) != 0) { goto out; }
      ...
   }
   job->bytes = download(io, addr, job->port, job->host, job->path);
   if (job->bytes >= 0) { job->status = 0; }
out:
   job->done = 1;
   Post(notify_default(), "client.done", job, &job->bytes,
    sizeof job->bytes);
}
```

The second context waits for that notification, a tenth of a second at a time, and prints a line every time the wait times out:

```
static void heartbeat(int * cancel, void * ctxt)
{
   struct job * job = (struct job *)ctxt;
   int beats = 0;
   long bytes = 0, r = -1;
   while (!job->done && !*cancel) {
      r = notify_wait(notify_default(), "client.done", &bytes,
       sizeof bytes, 100);
      if (r != -EVIO_ETIMEDOUT) { break; }
      beats += 1;
      note("heartbeat %d: the thread is free while I/O waits", beats);
   }
   if (r == (long)sizeof bytes) {
      note("heartbeat: notified, client done with %ld bytes", bytes);
   }
}
```

Against a local server that takes a second to answer, the program's remarks, which go to standard error so that the page itself can go to a file, read

```
[ 0.000] http: connecting to 127.0.0.1:28081
[ 0.000] http: connected, sending GET /
[ 0.100] heartbeat 1: the thread is free while I/O waits
[ 0.201] heartbeat 2: the thread is free while I/O waits
      ...
[ 0.902] heartbeat 9: the thread is free while I/O waits
```

```
[ 1.001] http: done, 235 bytes
[ 1.001] heartbeat: notified, client done with 235 bytes
```

Nine heartbeats in the second the server took, and the last line a few microseconds after the client's: the notification was posted on the thread the heartbeat lives on, went through the queue, and woke the heartbeat between two rounds. With a name to resolve, the resolver's lines come first,

```
[ 0.000] dns: query 1 for example.com sent to 8.8.8.8
[ 0.021] dns: example.com is 172.66.147.243
[ 0.021] http: connecting to 172.66.147.243:80
```

and against a DNS server that never answers—192.0.2.1 is an address reserved for documentation and goes nowhere—the resolver tries three times, two seconds apart, while the heartbeat ticks sixty times:

```
[ 0.002] dns: query 1 for example.com sent to 192.0.2.1
[ 2.003] dns: timeout
[ 2.003] dns: query 2 for example.com sent to 192.0.2.1
[ 4.003] dns: timeout
[ 4.003] dns: query 3 for example.com sent to 192.0.2.1
[ 6.004] dns: timeout
```

Six seconds of elapsed time, sixty heartbeats, and—the number that matters—this from `time`:

```
0.01s user  0.00s system  0% cpu  6.004 total
```

Every wait in the program is a call that parks the calling context: the resolver's `evio_recvfrom` with its two-second timeout, the downloader's `evio_connect` and `evio_recv`, the heartbeat's `notify_wait`. While all of them are parked the thread is asleep in the kernel queue, and the program costs nothing; the moment a datagram arrives, or a timer is due, the thread wakes and the right context continues. A program that had polled for those six seconds would report six seconds of user time and produce exactly the same output.

3.2. What the queue does

The first answer anybody reaches for, faced with the four-line program at the top of the chapter, is to give each connection its own thread and let each thread block. The program stays readable and the operating system does the interleaving. For a dozen connections this is the right answer and you should use it. It stops being the right answer some way below ten thousand, for three reasons that get worse in that order: a thread's stack is a reservation of address space, a megabyte or eight by default, so ten thousand threads reserve ten gigabytes of which they will touch perhaps forty megabytes; a thread that is *woken* costs a trip through the kernel scheduler, and ten thousand connections that each speak once a second are ten thousand of those a second; and the moment two connections need to

touch the same thing—a cache, a counter, a session table—the program needs a lock, and it acquires all the problems the preface listed. So: threads, yes, but not one per activity, and Chapter 4 says how many. What holds the connections is the queue.

What the library wants from a queue is three things. It wants a thread to be able to *block* in it, with a timeout, so that a thread with nothing to do sleeps and a timer due in forty milliseconds still fires on time. It wants an event to say *which* context it concerns, so that the right context can be woken and no other. And it wants a way for *another thread* to end the wait, so that a thread asleep in its queue can be handed work; that is the letter box of Section 1.4, and it is Chapter 4's business. Each of the three kernels gives all three, differently, and the library makes them alike behind one interface.

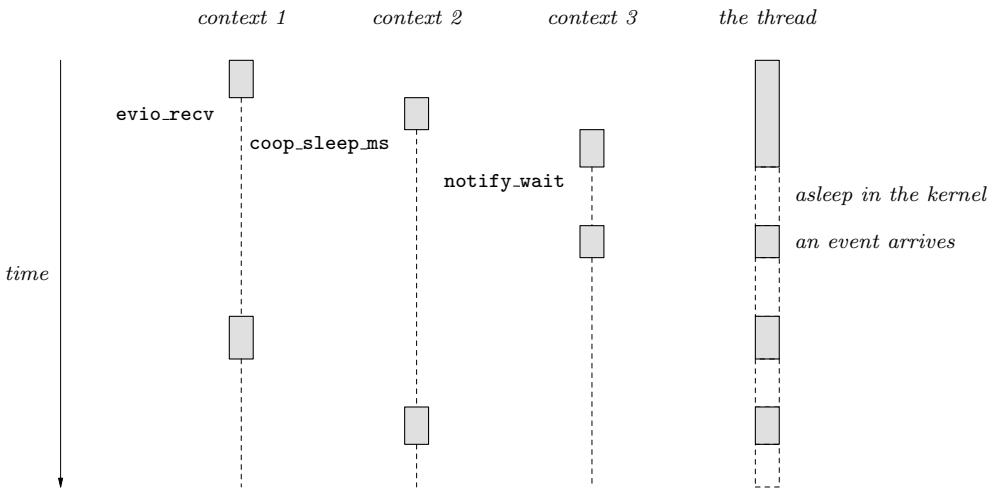

Fig. 3. One thread, three contexts, one queue. Time runs downward. Each context is a straight line of statements that stops where it must wait; the thread is never idle while any of them can proceed, and when all three are parked it sleeps in the kernel until an event arrives.

The interface presents *operations that complete.* Two of the three kernels report that a socket is *ready* and leave the program to do the read; the third does the read itself and reports that it is *done.* The first kind can be made to look like the second easily—try the operation, and if the socket is not ready wait for readiness and try again—and the second cannot be made to look like the first at all, so an interface that is to behave identically on all three must present the second. That is why the six operations carry buffers and timeouts rather than reporting states, and it is the last thing about the kernels that a program needs to know. A call such as

```
n = evio_recv(io, s, buf, sizeof buf, timeout_ms);
```

looks like `recv` and returns what `recv` would; but while it waits, the thread is in the kernel queue or running other contexts, and nothing that happened in between could have interleaved with the calling function's own statements.

3.3. The queue object, and sockets

One `evio` belongs to one thread, and it is opened on that thread's scheduler and closed before it. Opening it creates the kernel queue—and, on Windows, performs the one-time `WSAStartup`, so that a program using the library never calls it—and installs the queue as the thing the scheduler does when it has nothing else to do: a round that ends with every context parked ends in the kernel, waiting, until the earliest timer or the first event. The scheduler's `user` field is the tidy way for a context to find its thread's queue, as the client does with `coop_system_here()->user`; a program with one thread may equally well use a global, or call `evio_here()`, and a program with several threads will want the field.

A socket must be prepared before a queue can work with it, and there are two ways to prepare one. A fresh socket comes from `evio_socket_create`, already prepared for the queue you name; use it for everything you create yourself—outgoing connections, listening sockets, UDP sockets. A socket that already exists is prepared with `evio_attach`, and there is exactly one place a program needs that: a connection that was accepted on one thread and is going to be used on another. The thread that will use it calls `evio_attach` before its first operation, and Chapter 4's server does this on every connection it serves.

Two properties of a socket follow from which queue it belongs to, and both are easier to learn here than in a debugger. Operations on a socket are performed by the thread that owns the queue it was prepared for; a socket is not a shared object, and handing one to another thread means handing it over completely, the giver stopping and the receiver attaching. And on Windows a socket belongs to one completion port for its lifetime, with no way to detach it and attach it elsewhere. This is why the hand-over happens once, at accept time, and why `evio_accept` deliberately leaves the new socket unprepared: at the moment of the accept, nobody yet knows which thread will serve it.

The type `evio_socket` is an `int` on POSIX and a `SOCKET` on Windows, and `EVIO_INVALID_SOCKET` is the failure value for both. When a program needs to put a socket in a `void *` it casts through `intptr_t`, which is the one spelling that survives both, and the examples do so wherever a socket travels as a context's argument.

The library wraps the six calls that wait. It does not wrap the ones that do not, and it does not hide the socket layer, because there is no reason to: `evio.h` includes the platform's socket headers, so `struct sockaddr_in`, `htons`,

`inet_pton`, `bind`, `listen`, `getsockname` and `setsockopt` are all available and are all called directly. A listening socket is made the ordinary way:

```
listener = evio_socket_create(io, AF_INET, SOCK_STREAM);
memset(&addr, 0, sizeof addr);
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
addr.sin_port = 0;                  /* let the system choose the port */
bind(listener, (struct sockaddr *)&addr, sizeof addr);
listen(listener, 64);
getsockname(listener, (struct sockaddr *)&addr, &alen);
```

That fragment is from Chapter 5, and it does something worth copying: binding to port 0 asks the system for an unused port, and `getsockname` then says which one it got. A test, or a program that starts a service for its own use, should never hard-code a port number.

3.4. Six operations that complete

Each of the six is called from inside a context, parks the context until the kernel has done the work, and returns what the corresponding system call would have returned. That is the whole interface to the outside world, and what remains is the meaning of the return values.

Errors are returned, never printed and never signalled. An operation returns a byte count or 0 on success, and on failure the *negation* of the platform's error code: $-$`errno` on POSIX, $-$`WSAGetLastError()` on Windows, and $-$`EVIO_ETIMEDOUT` when a timeout ran out. So the test is always `if (n < 0)`, and the way to print one is always with the sign put back, as in

```
note("dns: recvfrom: %s", evio_strerror((int)-n));
```

The one value that is neither success nor failure is 0 from `evio_recv`, which means *end of stream*: the peer has closed its end and no more bytes are coming. A read loop therefore has three cases, and all three appear in the downloader of Section 3.6.

Every operation that can wait indefinitely takes a timeout in milliseconds, and -1 waits forever. A timeout is not an error condition; it is a normal outcome that a correct program is expected to handle, and it is how a program stays in control of its own time. Three properties make a timeout usable rather than merely available.

A timeout never loses data. If the operation had in fact completed a moment before the deadline was noticed, the completion wins and the bytes are returned. On the systems that report readiness this is trivially true, since a call that timed out never read anything and the data is still in the socket for the next call; on Windows the library waits for the cancelled operation to be reported before

returning, and if the report says the operation succeeded, those bytes are what you get.

A timeout is per call, not per transaction. Five seconds on `evio_recv` is five seconds without a single byte, and a peer that sends one byte every four seconds can hold a read loop open indefinitely. A program that cares about the total should compute a deadline once and pass the remaining time to each call.

A short timeout is the way to stay cancellable. Section 2.6 said that a context parked in a wait is not woken by `coop_cancel`; the flag is set, and the context sees it when the wait returns. So a context that must be stoppable while it waits gives its waits a deadline and looks at `*cancel` when they expire:

```
while (!*cancel) {
   int r = evio_accept(&io, listener, &c, 200);
   if (r == -EVIO_ETIMEDOUT) { continue; }    /* look at *cancel again */
   ...
}
```

That is the standard shape for a long-lived server context, and the price of it is one system call every 200 milliseconds while nothing is happening.

Two of the six deserve a note of their own. Sending is written to send *all* of its buffer, parking as often as it must and looping over short writes, because a partial send is almost never what a program means. Its timeout is therefore over the whole buffer, and it behaves the way a timeout that must not lose data has to: if the deadline passes after some bytes have already gone, the call returns *how many,* so that the next call can resume where it stopped, and it returns $-$`EVIO_ETIMEDOUT` only when nothing at all went. A caller that wants the plain partial form, one write and whatever fits, can still call `send` directly. Sending also never raises `SIGPIPE`: writing to a connection the peer has closed returns an error like any other failure, and a program needs no signal handling. And accepting hands back a socket that belongs to no queue, for the reason Section 3.3 gave: at the moment of the accept, the program may not yet have decided which thread is going to serve the connection, and on Windows that decision cannot be revised later.

3.5. Resolving a name

With the queue understood, the resolver is a short program that happens to speak a wire protocol. It speaks DNS directly, over UDP, because that is the natural use for `evio_sendto` and `evio_recvfrom` and because the protocol is small enough to read in one sitting. A message is at most 512 bytes, laid out as RFC 1035 describes: a twelve-byte header—an identifier chosen by the client and echoed by the server, sixteen bits of flags, and four counts that say how many questions, answers, authority records and additional records follow—then a question, then, in the reply, the answers. The client sets one flag, *recursion desired,* which asks the server to chase the name down on the client's behalf; the

low four bits of the flags in the reply carry a response code, of which 0 is success and 3 is "no such name".

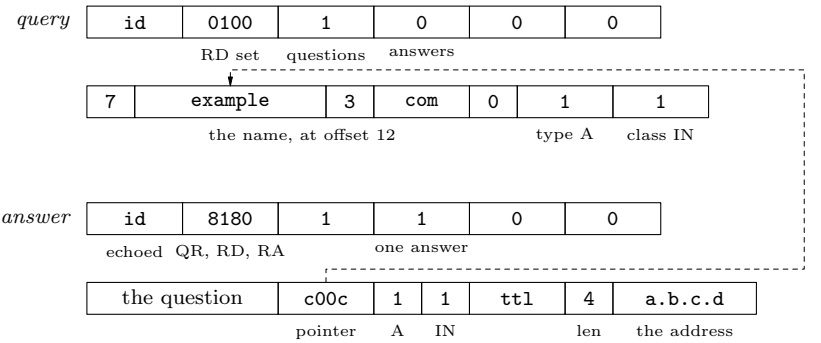

Fig. 4. A query for `example.com` and the shape of its answer. Every multi-byte field is big-endian; the answer repeats the question's name as a two-byte pointer back to offset 12.

A name is a sequence of labels, each a length byte followed by that many characters, ended by a zero byte, so `example.com` goes on the wire as `7example3com0`. The program builds and reads messages with byte arithmetic rather than a packed structure—packing is a compiler extension, and the wire's byte order is not the byte order of any of the three machines:

```
buf[0] = (unsigned char)(id >> 8);
buf[1] = (unsigned char)id;
buf[2] = 0x01;                  /* recursion desired */
buf[5] = 1;                     /* one question */
```

The one subtlety is compression. A name in the answer usually repeats the name in the question, and rather than spell it out the server may put, anywhere a label could start, two bytes whose top two bits are 11 and whose remaining fourteen are the offset in the message of the name to continue with. A parser that only wants to *skip* a name therefore stops at the first such pointer, since the pointer ends the name it appears in:

```
static int dns_skip_name(const unsigned char * p, int len, int off)
{
   while (off < len) {
      int l = p[off];
      if (l == 0) { return off + 1; }
      if ((l & 0xc0) == 0xc0) {               /* a compression pointer */
         return off + 2 <= len ? off + 2 : -1;
      }
      off += 1 + l;
   }
   return -1;
```

```
}
```

Nothing in the program ever follows a pointer, which is also why nothing in the program can be made to loop by a malicious one—a real hazard, and one that has produced real denial-of-service bugs in real resolvers.

The interesting part, for this chapter, is the retry loop, because it is what the whole interface exists to make writable. Three attempts, two seconds each, and an inner loop that discards answers to somebody else's question:

```
for (attempt = 1; attempt <= DNS_ATTEMPTS; attempt += 1) {
   uint16_t id = (uint16_t)((coop_now_ns() >> 10) + 7919 * attempt);
   int qlen = dns_build_query(query, (int)sizeof query, id, host);
   long n;
   n = evio_sendto(io, s, query, (size_t)qlen, (struct sockaddr *)&dst,
    sizeof dst, DNS_TIMEOUT_MS);
   if (n < 0) { note("dns: sendto: %s", evio_strerror((int)-n)); break; }
   note("dns: query %d for %s sent to %s", attempt, host, server);
   for (;;) {
      n = evio_recvfrom(io, s, answer, sizeof answer,
       (struct sockaddr *)&from, &fromlen, DNS_TIMEOUT_MS);
      if (n == -EVIO_ETIMEDOUT) { note("dns: timeout"); break; }
      if (n < 0) { ... goto out; }
      r = dns_parse_a(answer, (int)n, id, out);
      if (r == -4) { continue; }         /* somebody else's answer */
      if (r == 0) { result = 0; goto out; }
      ...
   }
}
```

There is no state machine, no callback, and no continuation. The timeout is a parameter. The retry is a `for` loop. The whole resolver is one function, written as if `evio_recvfrom` blocked, and you can read what it does by reading it downwards; the transcript of Section 3.1 against the black hole is this loop taking its three turns.

The identifier is not decoration. The socket is unconnected, so anything anyone sends to its port arrives, including the answer to attempt 1 turning up during attempt 2, and treating that as the answer to attempt 2 would be wrong in the case where the two attempts asked different questions, which a longer-lived resolver would do. The `continue` on -4 is that check. And the scan tolerates aliases with no special code: a recursive server puts the `CNAME` records first and the `A` record after them in the same section, and the scan walks past anything that is not an `A`.

3.6. Fetching a page

The download is a TCP connection, one request, and a loop until the server closes:

```
r = evio_connect(io, s, (struct sockaddr *)&dst, sizeof dst,
 HTTP_TIMEOUT_MS);
...
snprintf(request, sizeof request,
 "GET %s HTTP/1.1\r\nHost: %s\r\nUser-Agent: stripboard-client/1\r\n"
 "Connection: close\r\n\r\n", path, host);
n = evio_send(io, s, request, strlen(request), HTTP_TIMEOUT_MS);
...
for (;;) {
   n = evio_recv(io, s, buf, sizeof buf, HTTP_TIMEOUT_MS);
   if (n < 0) { ... break; }
   if (n == 0) { break; }                /* closed: response complete */
   fwrite(buf, 1, (size_t)n, stdout);
   total += n;
}
```

There are the three cases of a read loop. The header `Connection: close` asks the server to hang up after the response, which turns "the response is complete" into "`evio_recv` returned 0" and spares the client any parsing of `Content-Length` or chunked encoding. The response goes to standard output as it arrives, headers included; the program's own remarks go to standard error, so that the output can be redirected to a file and be the page.

3.7. The heartbeat, and a flag

The heartbeat is the smallest useful thing built out of Section 2.7. Each timeout of `notify_wait` is a tick, and the first delivery ends the loop with the byte count that the client posted. Note the flag `job->done` beside the notification, and the order in which the client sets the one and posts the other. If the client finished before the heartbeat's first `notify_wait` registered its observer—which happens when the connection is refused at once—the notification would have been posted to nobody, and a heartbeat that relied on it alone would tick forever. The flag is the state; the notification is the wake-up. A heartbeat that arrives to find the flag already set skips the wait, and one that is waiting when the flag is set is woken by the post that follows it.

Both contexts are on one thread, and the thread has a queue, so the delivery goes through the queue to the same thread and runs between the round in which the client posted and the one in which the heartbeat wakes—the deferred path of Section 1.4. The transcript shows the heartbeat's last line a few microseconds after the client's.

3.8. Lifetimes and limits

Almost everything in this chapter is a convenience. Two rules are not, and a program that breaks either has a bug that will not announce itself.

A context with an operation in flight must not be destroyed. While a context is parked inside one of the six operations, the kernel holds a pointer into that context's own stack and will write into it when the operation finishes or is cancelled. Freeing the stack meanwhile—by closing the scheduler, in practice—means the kernel writes into memory that has been given to something else, and nothing detects this at the point where it happens. The remedy is the pair of loops from Section 2.4, `coop_run_until` for the thread's working life and `coop_run` to let live operations finish, before the queue and the scheduler are closed. Give every operation a timeout and the drain is bounded by the longest of them.

Closing a socket under a context is allowed. This one is a permission, and it is the usual way to stop a stuck connection from elsewhere. Closing the socket makes the pending operation fail; the failure is delivered normally, the context wakes with an error, and the record on its stack is still there to receive it. It is the context that must not go away, not the socket.

Two limits follow from the design, and it is better to meet them here than in a debugger. *A context has one operation in flight at a time.* A parked context is woken without being told what woke it, so a context waiting for two things at once could not tell which had happened; full-duplex traffic on one socket is therefore two contexts, one per direction, which is also how you would write it by hand. *A socket takes one reader and one writer, not two of either.* Those two contexts, one reading and one writing the same socket, work on all three kernels; a third context that tries to wait in a direction already taken is turned away at once with `−EBUSY` rather than quietly displacing the context already there. Making Linux keep the two directions apart—`epoll` keeps a single interest record per descriptor, and the naive code lets the second waiter overwrite the first—was the one place the three kernels did not already agree, and the library reconciles them so that a program need not know which it is on. Neither limit troubles the programs in this book.

3.9. What the queue cannot hold

The queue holds sockets, and only sockets. This is not the library's limitation but the kernel's: a queue tells a program that a socket has become readable, and then the program reads without waiting, and the whole edifice rests on the read being instant once the readiness is known. A regular file breaks the premise, on every one of the three systems and each in its own way. Ask `epoll` to watch a file and it refuses outright; ask `kqueue` and it reports the file ready always, which is true and useless, because the read that follows still goes to the disk and still blocks; set `O_NONBLOCK` on a file and it does nothing, because a file is never "not ready." There is no portable call that waits for a file the way the queue

waits for a socket. The same is true of a name to resolve: `getaddrinfo` is a blocking library call with no descriptor to watch and no asynchronous form the three systems agree on.

So the library does the one thing that always works: it hands the blocking call to another thread, and lets that thread block while this one does not. A small pool of threads waits for work; a context that wants a file read gives the read to the pool and parks; a pool thread does the ordinary, blocking `pread`; and when it is done it posts the result back through the letter box of Section 1.4 to the context's own thread, which wakes the context between two rounds. It is Section 1.4's pattern exactly, turned to a new purpose: the pool thread is just another thread reaching into this one from outside, and the only door it uses is the one every outside thread uses. The context, meanwhile, has written a plain call that returns when the read is done, and its thread has served other contexts the whole time the disk was busy.

That pool is `fio.h`, and its primitive is `fio_call`, which runs one function on a pool thread and returns what it returned:

```
long fio_call(evio * io, long (*fn)(void * arg), void * arg);
```

Every file operation is one use of it, and so is `evio_resolve`. A program that has some other blocking call to make—a database library that knows nothing of contexts, a compression routine that runs for a second—gives it to `fio_call` directly and keeps its thread responsive. Outside a context there is nothing to keep responsive, so `fio_call` simply makes the call on the spot; the same code then works before the scheduler is running and inside it.

The file calls name their offset—`fio_pread` and `fio_pwrite`, after the POSIX calls that read and write at a position without a shared file pointer—so that two contexts may work on one open file at different places without a seek to trip over. A write is told to write all of its buffer, like `evio_send`, and a read returns fewer bytes than asked only at the end of the file, where it returns zero. There is no timeout on a file operation, and the reason is the same one that shows through everywhere in this design: a read that has entered the disk cannot be safely abandoned, so a call that could be cancelled would have to be one the kernel queue could hold, and it isn't. A file, once asked for, is waited for.

Reading and writing are not everything a program does to a file. The same pool carries the operations that act on a file's *name*: `fio_unlink` removes one, `fio_rename` renames or moves one—replacing an existing destination, as POSIX `rename` does and Windows must be asked to—`fio_mkdir` makes a directory and `fio_rmdir` removes one that is empty, `fio_stat` reports what a path is, and `fio_readdir` lists one. Each is one `fio_call` on a path, returns 0 or a negative error, and parks its context exactly as a read does. What `fio_stat` fills in is the handful of fields a program uses, not the platform's whole `struct stat`:

```
enum { FIO_OTHER, FIO_FILE, FIO_DIR };
typedef struct { int64_t size; int64_t mtime_ns; int type; } fio_info;
```

with `mtime_ns` in nanoseconds since the epoch—the wall clock, not the monotonic `coop_now_ns`. Reading a directory follows the shape of `evio_resolve` rather than the stateful `opendir`/`readdir` of C: one call fills a buffer with as many entry names as fit, each ended by a `NUL`, skips . and .., and returns how many—

```
char names[4096], * p = names;
int i, n = fio_readdir(io, dir, names, sizeof names, 256);
for (i = 0; i < n; i += 1) { puts(p); p += strlen(p) + 1; }
```

so a directory is read with no handle to open and close, and a buffer too small for the next name ends the walk with −`ERANGE` and the names so far in place. These six are conveniences over `fio_call`, not new machinery: a program that wanted an operation they do not name would write it as `mirror.c` writes its own cleanup, by handing the blocking call to the pool itself.

3.10. Names, and signals

Resolving a name is the pool's second customer, and it wears the same face as the socket calls around it:

```
struct sockaddr_storage addrs[4];
int n = evio_resolve(io, "example.com", "443", AF_UNSPEC, addrs, 4);
```

It fills in up to `cap` addresses in the order the resolver prefers them and returns how many, or a negative error whose text `evio_strerror` knows; a family of `AF_INET` or `AF_INET6` narrows it, and `AF_UNSPEC` takes whatever the name has. Under the surface it is `getaddrinfo` on a pool thread, so the calling context parks and its thread runs others while the network's own name servers are consulted—which can take as long as any fetch, and used to be the one blocking call left in an otherwise event-driven program.

Signals are the other thing that arrives from outside the program entirely, and the library folds them into the mechanism a program already has. A handler cannot safely do much, and least of all can it touch a scheduler running on some thread it interrupted; the classical answer is for the handler to set a flag and get out. Here the flag is a notification. Register a signal with

```
evio_signal(io, SIGINT, "app.interrupt");
```

and the signal's normal effect is suppressed; instead, the next time the queue's thread comes round, a notification named `app.interrupt` is posted, with the signal number as its payload, and whichever context waits for it or observes it wakes exactly as if another thread had posted it. Ctrl-C becomes a message. On Windows, where there are no `SIGINT` handlers of the POSIX kind, the same call catches the console's Ctrl-C and close events and delivers them the same way, so

that a program's shutdown path is written once. The server of Chapter 4 uses this to stop on Ctrl-C by the very route it already stops on a `/quit` request.

3.11. Copying a file across a socket

The chapter's second program, `examples/mirror.c`, puts the pool and the queue to work together. It copies a file to a second file through a loopback socket: a *sender* context reads the source from disk and sends it, a *receiver* context receives it and writes it to the copy, and a *ticker* prints how far along the copy is—all three on one thread, with no thread ever blocked. Every read and write of the disk is a call to the pool; every send and receive is a wait in the queue; and the ticker, which does neither, keeps printing throughout, which is the proof that the thread was free the whole time.

The sender is the heart of it. It accepts the receiver's connection, opens the source, and loops: read a piece from the file, send the piece down the socket, until the read returns zero at the end of the file.

```
while (!stop) {
   long n = fio_pread(&io, f, buf, sizeof buf, offset);
   if (n == 0) { break; }                      /* the end of the file */
   if (n < 0) { note("read: %s", evio_strerror((int)-n)); break; }
   n = evio_send(&io, s, buf, (size_t)n, TIMEOUT_MS);
   if (n < 0) { note("send: %s", evio_strerror((int)-n)); break; }
   offset += n;                    /* a short send: carry on from there */
}
```

Two waits, one line apart, and each of a different kind: `fio_pread` parks the sender while a pool thread reads the disk, and `evio_send` parks it while the kernel drains the socket to the receiver. Between them the sender's thread has the receiver and the ticker to run. The receiver is the mirror image, `evio_recv` then `fio_pwrite`, writing each arrival at the offset that counts the bytes so far. The accepted socket is attached to the queue before its first use—the hand-over of Section 3.3, here to the same thread that accepted it, which the accept still leaves to the program to do.

Copying sixty-four mebibytes, the program says

```
[ 0.455] 64 MiB from mirror.source to mirror.copy through localhost:50063
[ 0.456] sender: 67108864 bytes to send
[ 0.555]   28.8 MiB on disk
[ 0.658]   49.3 MiB on disk
[ 0.740] sender: done, 67108864 bytes sent
[ 0.858]   64.0 MiB on disk
[ 0.928] receiver: done, 67108864 bytes written
[ 0.960] checking both files at once
[ 1.173] mirror.source: 67108864 bytes, sum 2f7d6e10e4805c50
[ 1.173] mirror.copy: 67108864 bytes, sum 2f7d6e10e4805c50
the copy is identical: 64 MiB in 0.505 s, checked in 0.221 s
```

The ticker's lines, every tenth of a second, fall between the sender's and the receiver's, which is the whole point: while a piece was in the disk and another was in the socket, the thread had time to print. The check at the end is itself a use of the pool with something to prove. Two checker contexts open the two files and read them at once, a pool thread each, and reduce each to a sixty-four-bit sum; because the pool has more than one thread, the two reads truly overlap, and the run reads twice as much disk in not much more time than one file alone would take. That the two sums agree is the copy's warrant.

Ctrl-C during the copy is the signal example made real. The main thread registers `SIGINT` as the notification `mirror.interrupt` and observes it; the observer sets the `stop` flag the two loops test; the sender's next send fails as the receiver closes, both contexts end, and the half-written copy is removed with a `remove` run on the pool—a blocking call like any other, handed to `fio_call`. Interrupted at forty megabytes, the program leaves nothing behind and says so. A copy that stops cleanly on a keystroke, with no handler doing anything but posting a name, is a small thing that is usually a large one.

CHAPTER FOUR

THREADS

THE CALLS OF THIS CHAPTER

`int thrd_create(thrd_t * t, thrd_start_t func, void * arg);`
`int thrd_join(thrd_t t, int * res);`
`int thrd_detach(thrd_t t);`
Start a thread running `int func(void *)`; wait for one and collect its result; or disclaim it. Creation returns `thrd_success`, `thrd_nomem` or `thrd_error`.

`thrd_t thrd_current(void);`
`int thrd_equal(thrd_t a, thrd_t b);`
`int thrd_sleep(const struct timespec * d, struct timespec * rem);`
`void thrd_yield(void);` `void thrd_exit(int res);`
Identity, and the three plain thread verbs, spelled as C11 spells them.

`int mtx_init(mtx_t * m, int type);`
`int mtx_lock(mtx_t * m);` `int mtx_trylock(mtx_t * m);`
`int mtx_unlock(mtx_t * m);` `void mtx_destroy(mtx_t * m);`
A mutex. The `type` is `mtx_plain` or `mtx_recursive`; write as though every mutex were plain. Trying returns `thrd_busy` when the mutex is held.

`int cnd_init(cnd_t * c);` `int cnd_signal(cnd_t * c);`
`int cnd_broadcast(cnd_t * c);`
`int cnd_wait(cnd_t * c, mtx_t * m);` `void cnd_destroy(cnd_t * c);`
A condition variable. Always wait in a `while` loop over the condition, never an `if`.

`void call_once(once_flag * f, void (*fn)(void));`
`THRD_LOCAL`
Run `fn` once per process however many threads race to call it; and declare a variable of which every thread has a private copy, as `static THRD_LOCAL int n;`.

`int evio_post(evio * io, evio_post_fn fn, void * arg);`
Run `fn(io, arg)` on `io`'s thread, between two of its scheduler rounds. Thread-safe, callable from anywhere, and the only way into a thread from outside. Returns 0 or a negative error.

The client of Chapter 3 had two contexts on one thread, and one thread is enough for a surprising amount: a thousand idle connections cost a thousand parked contexts and nothing else, and the client would be no busier resolving a hundred names than one. What one thread cannot do is use a second processor, and what

one thread should not do is let one piece of heavy work—a page rendered, a file compressed—hold up everything else on it. A machine with eight cores runs eight threads at once and one thread once; a context that spends 200 milliseconds in a tight loop makes every other context on its thread 200 milliseconds late, and the polite loop of Section 2.5 is a discipline you cannot impose on a library you did not write. For those two reasons, and only those two, a program wants threads, and this chapter is about how threads and the library get along.

They get along by keeping apart. Contexts give one thread the appearance of many activities; threads give a program more than one processor; the library keeps the two ideas separate with a single rule, stated in Section 4.3, which is that *a context never leaves the thread that made it, and the only way into a thread from outside is its letter box.* Everything else in the chapter—the small thread library, the server, its careful shutdown—follows from the rule. Neither reason above is "so that connections can proceed in parallel"; that is what the queue is for. Use threads for processors, not for waiting, and the number a program wants is roughly the number of cores it has, not the number of things it is doing.

4.1. The server

The chapter's program is `examples/server.c`, a web server that listens on a port (8080 by default) with n workers (4), answers every HTTP request with a small page that says which worker served it, and stops when a client asks for `/quit` or the terminal sends a Ctrl-C. Three hundred lines; one mutex. One thread accepts, n threads serve, and a connection goes from the one to the others as a function left in a worker's letter box.

Each worker has a scheduler, a queue, and one context per connection it is holding. Nothing about a connection is shared: from the moment the posted function runs on the worker, that worker's queue owns the socket and no other thread refers to it. Drive the server, with three workers, from a script that opens 150 connections at once, each sending one request and reading until the server closes; then park a connection that sends nothing, send a `HEAD` on another, check that it is answered while the idle one is still parked, wait for the idle one to be closed, and send `/quit`:

```
150 concurrent requests: 150 ok in 0.05s
served while an idle client is parked: True
idle client closed by server: True after 5.0s
acceptor: stopped
worker 1: stopped after 51 requests
worker 0: stopped after 50 requests
worker 2: stopped after 51 requests
served 152 requests
```

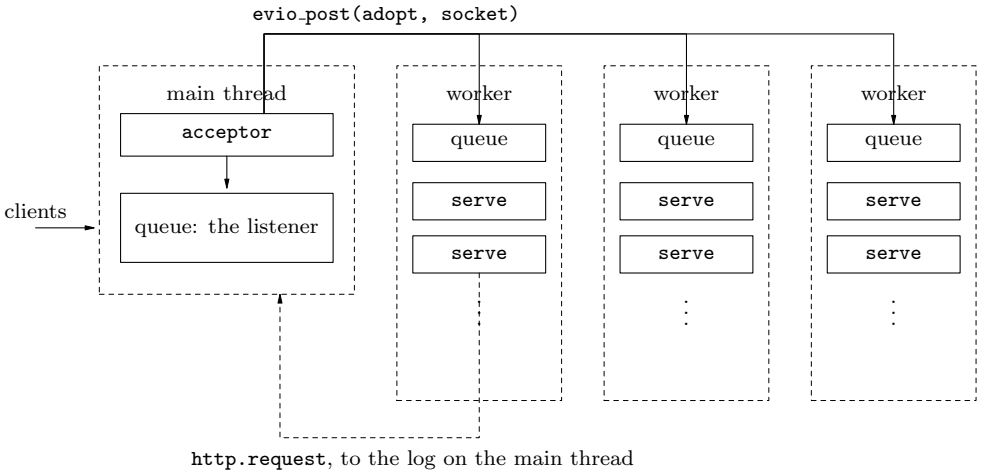

Fig. 5. The server. The acceptor takes connections on the main thread and posts each one to a worker's queue, round-robin; each worker spawns a context per connection; workers report back to the main thread by notification, so that the log has a single writer.

Every line there is a claim about the design. The 152 are the 150, the `HEAD` and the `/quit`; the split of 51, 51 and 50 is the round-robin doing what it says; the 5.0 seconds is the read timeout of a connection context, arriving exactly when it should; and the `True` on the second line is the central claim of the whole book, that one slow client parks one context and nobody else notices. The 0.05 seconds for 150 requests is where the design pays for itself twice: each operation is attempted before anything is registered with the kernel, so a socket with its request already waiting costs one system call and no parking at all, and the requests that did have to wait cost one parked context each, which is a few hundred bytes and no thread. The same numbers come out on all three systems. The rest of the chapter reads the program in the order things happen, after two sections on what it is built from.

4.2. A thread and its library

A thread is what the operating system runs. A program begins with one; `thrd_create` makes another, which runs a function of the program's choosing until the function returns, and `thrd_join` waits for that. Threads of one process share its memory, which is their whole attraction and their whole danger. Two threads that write the same variable at the same time can leave it holding a value neither of them wrote, and on a modern machine the ways this happens are more varied than most people expect: the write may be split; one thread's write may sit in a store buffer while the other reads memory; the compiler may have kept

the variable in a register across what you thought was a synchronisation point; the processor may have executed your two writes in the other order. "It is only an `int`" is not a defence. C says the program has undefined behaviour, and the machines agree.

The remedies are to keep the threads from writing at once—a *mutex*, which one thread holds at a time, so that the code between locking and unlocking runs alone—or to keep them from sharing. A mutex is cheap, correct and well understood. It is also the beginning of a long road: a program with many locks acquires deadlocks, lock ordering rules, contention, and the particular misery of a bug that depends on which of two threads was scheduled first. There is nothing wrong with a mutex around a counter. There is a great deal wrong with a program in which every data structure has one, and this library's position is that the second kind of program is avoidable.

The header `threads.h` is the part of C11's `<threads.h>` that the library and its examples need, spelled exactly as C11 spells it, so that a program written against it will compile unchanged against a C11 library that provides the real thing. C11 threads are still not universally available—notably not in Microsoft's C runtime—which is why the file exists.

this library	*POSIX*	*Windows*
`thrd_create`	`pthread_create`	`CreateThread`
`thrd_join`	`pthread_join`	`WaitForSingleObject`
`thrd_detach`	`pthread_detach`	`CloseHandle`
`thrd_current`	`pthread_self`	`GetCurrentThread`
`thrd_equal`	`pthread_equal`	`GetThreadId`
`thrd_sleep`	`nanosleep`	`Sleep`
`thrd_yield`	`sched_yield`	`SwitchToThread`
`thrd_exit`	`pthread_exit`	`ExitThread`
`mtx_init`, `mtx_lock`, ...	`pthread_mutex_*`	`CRITICAL_SECTION`
`cnd_init`, `cnd_wait`, ...	`pthread_cond_*`	`CONDITION_VARIABLE`
`call_once`	`pthread_once`	`InitOnceExecuteOnce`
`THRD_LOCAL`	`__thread`	`__declspec(thread)`

A *condition variable* lets a thread sleep until another thread says that something it cares about may have changed; it is used exactly once in these programs, in Section 4.4's start-up barrier. Running a function once per process however many threads race to call it is `call_once`, which is how `notify_default` creates the shared center without a lock in the common path. And `THRD_LOCAL` declares a variable of which every thread has its own copy, per thread and not per context.

Two of the mappings hide a trap, and since the point of the module is to hide traps, they are worth naming. A Windows critical section is always recursive, so that a thread may lock one twice without deadlocking; `mtx_plain` and `mtx_recursive` therefore behave identically there and differently on POSIX, and a program that wants the same behaviour everywhere writes as though every mutex were plain, never locking one twice and never relying on being

able to. And on Windows `thrd_current` returns a pseudo-handle, a constant meaning "whichever thread is asking", so two of them compare equal with `==` even when they were obtained on different threads; `thrd_equal` compares thread identifiers instead, and the practical rule is that a value from `thrd_current` is meaningful on the thread that obtained it, or when compared against a handle from `thrd_create`, and should not be stored and handed to somebody else.

4.3. The rule

Here is the rule that the rest of the library is built on. It is worth reading twice.

> *A context belongs to the thread that spawned it, and every call on it—spawning, cancelling, waking, resuming, and every parking call it makes—is made from that thread. A thread that wants contexts opens a scheduler of its own, and, if it talks to the network or must be reachable from other threads, a queue of its own. What crosses from one thread to another is never a context, but a message to the thread that owns it, and there are exactly two carriers:* `evio_post`*, which asks a thread's queue to run a function of your choosing between two of its rounds, and* `Post`*, which is built on it.*

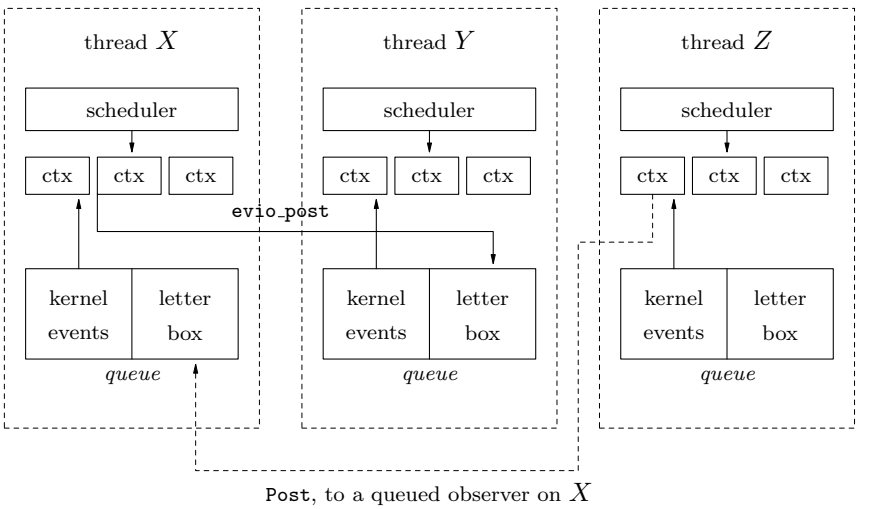

Fig. 6. Three threads keeping the rule. A context never leaves its column; the only way across is a function left in another thread's letter box, by `evio_post` directly or by `Post` through a queued observer.

The figure draws a program that keeps the rule. Each thread is a column: its scheduler with its list of contexts, its queue with the kernel's events on one side and the letter box on the other. Every arrow that crosses a column boundary ends in a letter box. Ordinary shared data is not drawn, because it is not the library's concern: the rule is about the library's objects, and a program may still share its own variables under a mutex, as the server does with its request counter.

What the rule buys is worth listing, because it is the argument for the whole design, and because every item on the list is something you get without writing any code for it. Nothing inside the library is locked on the fast path, because nothing on that path is shared: one thread reads and writes its own list of contexts, its own timers, its own flags. An observer's state is private to its thread, which is the guarantee Chapter 1 gave: a queued observer runs on the thread it was registered on, so whatever it touches is touched by one thread, however many threads post to it. A context's local variables are safe to hand to the kernel, because the queue writes into them from the same thread, between rounds, while the context is parked. And a timeout and a wake-up cannot race, because the parking call of Section 2.8 decides "was I woken, or did my deadline pass?" by reading a plain `int` with no lock, and it is correct precisely because the writer and the reader are the same thread.

The rule could have been relaxed. A runtime that moves contexts between threads to balance load is a reasonable thing to build, and several languages have one. It is not what this library does, for three reasons, in increasing order of weight. Thread-local storage would stop meaning anything: a context that reads a `THRD_LOCAL` variable, parks, and resumes on another thread now sees a different variable, which is not a bug a library can hide but a change to what C means. A Windows socket belongs to one completion port for life, so a context that moved would have to leave its sockets behind, which is to say it could not move while holding one, which is to say it could not move. And the cheap parts would stop being cheap: every item in the list above turns into a lock or an atomic, on the hottest path in the system. The price of not migrating is load balancing, and the server pays it in the crudest way available, handing connections to workers round-robin at accept time; a worker that draws a slow connection is unlucky, which for requests that are broadly similar costs nothing measurable. Chapter 5 takes the other approach and lets the workers ask for work when they are ready for it, which balances itself.

The rule is about the library's objects, not about your program's. Two threads may share a counter under a mutex, as the server does with the number of requests it has served; they may share a read-only table with no mutex at all; they may use every atomic and every lock-free queue you care to write. What the rule says is that you never have to do any of that *to use the library*: you never lock a scheduler, never lock a center, never wonder which thread an observer

runs on. That leaves three ways to share, and a program built on this library chooses between them constantly.

mechanism	*cost*	*when*
a mutex round the data	a lock per access	small, hot, no policy
`evio_post` to the owner	an allocation, a wake-up	the owner is one known thread
`Post` to observers	the above, per observer	the audience is not known

A mutex is right when the shared thing is small, is touched constantly, and has no rules attached to it: a counter, a statistics block, a cache with an obvious eviction policy. A posted function is right when there is a natural owner and you know who it is; the acceptor below knows exactly which worker should take a connection, so it posts a function to that worker, and the posted function runs on the worker's thread, between two of its rounds, with no lock held and with every context on that worker parked, so it may do anything at all, including spawn a context. A notification is right when you do not know, or should not know, who is listening; the server's workers report every request without knowing that the main thread is logging them. The trap is to reach for the mutex out of habit when the shared thing has a *policy*—rules about who gets what, in which order, under which conditions. Policy under a mutex ends up scattered across everyone who takes the lock. Policy in an owner ends up in one function that you can read, and Chapter 5 is an entire program built on that observation.

4.4. The shape of a thread

Every thread in this book that runs contexts has the same body, and the whole of this chapter is contained in it:

```
static int worker_main(void * arg)
{
   struct worker * w = (struct worker *)arg;
   coop_open(&w->sys, 0);                  /* 1. a scheduler */
   w->sys.user = w;
   evio_open(&w->io, &w->sys);             /* 2. a queue */
   notify_observe(notify_default(), "server.stop", &w->io, on_stop, w);
   report_ready(w, 0);                     /* 3. announce it exists */
   coop_run_until(&w->sys, &w->stop);      /* 4. idle in the queue */
   coop_run(&w->sys);                      /* 5. let live work finish */
   evio_close(&w->io);
   coop_close(&w->sys);
   return 0;
}
```

Open a scheduler; open a queue; register what this thread listens for; report in; run until told to stop; drain; close. Two of those steps are easy to get wrong, and both mistakes are the kind that work on your machine.

The first is step 3, which exists because of a race. The thread that starts the workers is about to post connections into their queues. A worker's queue

does not exist until that worker has run `evio_open`, and the order in which two threads reach two points is not something to assume. If the first connection arrives before the first worker has got that far, `evio_post` locks a mutex that was never initialised: on Linux this usually appears to work, because a zeroed `pthread_mutex_t` happens to be a valid unlocked mutex, and on Windows it corrupts the heap. So the workers count themselves in, and the starter waits, which is what a condition variable is for:

```
static void report_ready(void)
{
   mtx_lock(&ready_lock);
   ready_count += 1;
   cnd_signal(&ready_cnd);
   mtx_unlock(&ready_lock);
}

static void await_ready(int n)
{
   mtx_lock(&ready_lock);
   while (ready_count < n) { cnd_wait(&ready_cnd, &ready_lock); }
   mtx_unlock(&ready_lock);
}
```

Note the `while` rather than an `if`: `cnd_wait` may return without anybody having signalled, so the condition is re-tested. This is true of condition variables everywhere and is not negotiable.

The second is the pair of loops in steps 4 and 5, which is Section 2.4's pair and Section 3.8's first rule. The first loop is the thread's working life: it runs rounds until the flag is set, and it does not mind an empty list, because on a worker the list is empty precisely while it waits for the next connection, with nothing to run and the thread asleep in its queue. When the stop flag is set—by a notification delivered on this thread, so there is no race on the flag—the loop returns, but the list may still hold contexts in the middle of a connection. The second loop runs rounds until it is empty, and only after that is it safe to close the queue and the scheduler, because a context blocked in an operation has handed the kernel a pointer into its stack, and the stack must outlive the operation. The flag itself is set from an observer, on this thread, so it needs no lock either:

```
static void on_stop(const notification * n, void * ctxt)
{
   (void)n;
   ((struct worker *)ctxt)->stop = 1;
}
```

One notification, broadcast once, stops every worker in the pool.

The main thread has the same shape with a different middle. It opens its scheduler and queue, registers two observers on its queue—which is the `queue` argument that Chapter 1 left as Λ, and says "deliver to me on this thread, between its rounds"—and listens:

```
notify_observe(notify_default(), "http.request", &main_io, on_request, NULL);
notify_observe(notify_default(), "server.quit", &main_io, on_quit, NULL);
evio_signal(&main_io, SIGINT, "server.quit");     /* Ctrl-C: the same */
evio_signal(&main_io, SIGTERM, "server.quit");

listener = listen_on(port);
```

The two `evio_signal` calls are Section 3.10 put to use: Ctrl-C and a `kill` arrive as the same `server.quit` that a client's `/quit` posts, so the careful shutdown below serves all three without another line of code. The function `listen_on` creates the socket with `evio_socket_create` on the main queue, sets `SO_REUSEADDR` on POSIX so that a restart need not wait out `TIME_WAIT`, binds, and listens with a backlog of 128. It deliberately does *not* set `SO_REUSEADDR` on Windows, where the option means something else entirely: it lets a second process steal the port from under the first. Then the workers start, the main thread waits for all of them to report in, and only then does it spawn the acceptor and run its own scheduler:

```
acceptor_ctx = coop_spawn(&main_sys, acceptor, (void *)(intptr_t)listener);
coop_run(&main_sys);                    /* returns once the acceptor ends */
```

4.5. Accepting and handing over

```
static void acceptor(int * cancel, void * ctxt)
{
   evio_socket listener = (evio_socket)(intptr_t)ctxt;
   int next = 0;
   while (!*cancel) {
      evio_socket c;
      int r = evio_accept(&main_io, listener, &c, ACCEPT_POLL_MS);
      if (r == -EVIO_ETIMEDOUT) { continue; }      /* look at *cancel again */
      if (r < 0) {
         fprintf(stderr, "accept: %s\n", evio_strerror(-r));
         coop_sleep_ms(100);
         continue;
      }
      evio_post(&workers[next].io, adopt, (void *)(intptr_t)c);
      next = (next + 1) % nworkers;
   }
   printf("acceptor: stopped\n");
}
```

Three details, none of them obvious. The half-second timeout is the cancellation protocol of Section 3.4 in its natural habitat: a context that must be stoppable while it waits gives its waits a deadline and looks at `*cancel` when they expire, and the acceptor pays one system call per half second of idleness for the privilege. A failed accept is not fatal, because running out of descriptors is a temporary condition and a server that exits on it is a server that a client can kill; the loop logs, pauses briefly so that the log does not become the problem, and carries on. And the socket travels through an `intptr_t`, the one spelling that survives both an `int` and an unsigned pointer-sized handle.

The hand-over is one line, `evio_post`, and on the worker the posted function spawns the connection's context:

```
static void adopt(evio * io, void * arg)
{
   if (coop_spawn(io->sys, serve, arg) == NULL) {
      evio_socket_close((evio_socket)(intptr_t)arg);
   }
}
```

It runs on the worker's thread, between two of its rounds, with every context on that worker parked. The spawn appends the new context to the worker's list and the worker's next round runs it; if the spawn fails, out of memory, the socket is closed rather than leaked, which is the sort of thing that is easy to leave out and impossible to notice. Nothing else about the connection is shared: the worker's queue owns the socket from the moment the function runs.

The obvious alternative—every worker accepting on the one listening socket—is common on POSIX and would have removed the posting. It was not chosen because it does not survive Windows: a socket belongs to one completion port for its lifetime, and n workers with n queues cannot all accept on it. The idiomatic Windows shape, one queue shared by n threads, does not fit either, because then a completion is taken by whichever thread happens to be free, and a context would resume on a thread other than its own, which the rule forbids. Handing the socket over is a few lines and works everywhere, so it is what all three platforms do.

4.6. Serving a connection

The connection context attaches the socket to its worker's queue, reads until it has a complete request head, answers, closes, and reports:

```
static void serve(int * cancel, void * ctxt)
{
   evio_socket s = (evio_socket)(intptr_t)ctxt;
   struct worker * w = (struct worker *)coop_system_here()->user;
   char request[REQUEST_MAX + 1], body[512], head[512];
   char method[16] = "", target[256] = "";
   int len = 0, blen, hlen, r;
   long served;
   (void)cancel;

   r = evio_attach(&w->io, s);            /* Windows: bind it to this queue */
   if (r != 0) { evio_socket_close(s); return; }

   while (len < REQUEST_MAX && find_header_end(request, len) == NULL) {
      long n = evio_recv(&w->io, s, request + len,
       (size_t)(REQUEST_MAX - len), READ_TIMEOUT_MS);
      if (n <= 0) { evio_socket_close(s); return; } /* closed/error/timeout */
      len += (int)n;
   }
   request[len] = '\0';
   sscanf(request, "%15s %255s", method, target);
   ...
```

The worker record is found through the scheduler's `user` field, which the thread body set; that is what the field is for. The attach is the second half of Section 3.3's story about accepted sockets: on POSIX it costs almost nothing, and on Windows it associates the socket with *this worker's* queue, which is why the acceptor deliberately left it unassociated. The read loop stops at the first blank line, at 8 kilobytes, or after five seconds without a byte, and in the last two cases the connection is simply closed. The five-second limit is per `evio_recv`, not per request, which is Section 3.4's second property in practice; a client that sends one byte every four seconds can hold a context for as long as it likes, which is the kind of thing a real server guards against with a deadline for the whole request. This one does not, and it costs the other clients nothing but one parked context.

The answer is a small page with a correct content length and a `Connection: close`; a `HEAD` request gets the head without the body. Then two counters are kept, and the difference between them is the whole of Section 4.3's advice in four lines:

```
mtx_lock(&stats_lock);
total_served += 1;
served = total_served;
mtx_unlock(&stats_lock);
w->served += 1;
```

The counter of requests served by the whole process is written by every worker, so it is under a mutex; the counter per worker is written only by its own thread, so it is not. Reaching for a lock on the second would be harmless and wrong: harmless because it costs little, wrong because it teaches the next reader that

the program does not know which of its data is shared. Then the context reports what it did, and, if the target was `/quit`, starts the end:

```
Post(notify_default(), "http.request", w, &report, sizeof report);
if (strcmp(target, "/quit") == 0) {
   Post(notify_default(), "server.quit", w, NULL, 0);
}
```

The report is a small structure with the worker number, the request number, the method and the target, copied by the center because the delivery crosses to the main thread, and printed by the main thread's observer:

```
static void on_request(const notification * n, void * ctxt) /* on main */
{
   const struct request_report * r = (const struct request_report *)n->data;
   (void)ctxt;
   if (n->len != sizeof *r) { return; }
   printf("worker %d: %s %s (#%ld)\n", r->worker, r->method, r->target,
    r->served);
}
```

One writer, so the lines cannot interleave, and no lock, because the observer runs on one thread. This is the third way of sharing, and the one the library is built for: the log is state that many threads want to add to, so it is given an owner, and the others send it messages. The observer runs on the main thread between rounds, with the copy of the report the center made; the worker that posted it went on to close its socket without waiting. The length check is Chapter 1's guard, because the center makes no promise about what a name carries.

4.7. Stopping, in order

Shutdown is where servers are usually weakest, because it is the path nobody exercises. This one is a sequence, and each step waits for the one before.

1. The context that served `/quit` answers, closes its socket, and posts `server.quit`—or a Ctrl-C or a `kill` posts the same notification through `evio_signal`, and the rest is identical.
2. The main thread's observer, on the main thread, cancels the acceptor. Within half a second the acceptor's `evio_accept` times out, the loop sees `*cancel`, and the acceptor returns.
3. The main thread's `coop_run` returns, because its list of contexts is now empty. The main thread closes the listening socket and posts `server.stop` once; each worker's observer hears it on its own thread and sets that worker's flag.
4. Each worker's `coop_run_until` returns. Its list may still hold contexts in the middle of a connection, so `coop_run` then runs rounds until they have all finished—at most five seconds, the read timeout. Only then does it close its queue and scheduler.

5. The main thread joins the workers, closes its own queue and scheduler, and prints the total.

Why so careful? Two of those steps are load-bearing. Step 2 must precede step 3 so that no connection is accepted after the decision to stop; otherwise the acceptor could post a socket into the queue of a worker that has already gone, which is a use-after-free with a one-in-a-thousand reproduction rate. Step 4 must complete before the queue is closed, which is Section 3.8's first rule and the one that bites hardest: a context parked in `evio_recv` has handed the kernel a pointer into its own stack, and closing the scheduler would free that stack. The cost of the care is that a client which connects and then says nothing delays shutdown by its read timeout, and for a server that stops once in its life that is the right trade. Every step crosses a thread boundary through a letter box or a join, and none through a shared variable without a lock; the flags `w->stop` are written by observers on their own threads and read by the loops of the same threads.

4.8. A thread for arithmetic

The second reason for threads was a computation that does not belong on the thread it was asked for on: too long to be polite, or in a library that cannot be made to relinquish. The pattern for it is fifteen lines, and it is the protocol of Section 2.8 with the letter box as the carrier. A context hands the work to a thread of its own, parks, and is woken when the answer is back:

```
struct job
{
  coop_context * ctx;                  /* who to wake */
  evio * io;                           /* whose thread to wake it on */
  long limit;
  long result;
  int done;
};

static void finish(evio * q, void * arg)      /* on the context's thread */
{
   struct job * j = (struct job *)arg;
   (void)q;
   j->done = 1;
   coop_wake(j->ctx);
}

static int hard_work(void * arg)              /* a plain thread */
{
   struct job * j = (struct job *)arg;
   j->result = count_primes(j->limit);
   evio_post(j->io, finish, j);
   return 0;
}
```

```
static void offload(int * cancel, void * ctxt)
{
   struct job j;
   thrd_t t;
   ...
   j.ctx = coop_current();
   j.io = evio_here();
   j.limit = 2500000;
   j.done = 0;
   if (thrd_create(&t, hard_work, &j) != thrd_success) { return; }
   while (!j.done) { coop_wait(); }
   thrd_join(t, NULL);
   trace("offload", "primes:", j.result);
}
```

The job record lives on the context's stack, which is safe for the reason Section 2.1 gave: the context is parked and cannot return while the thread is running. The worker thread never touches the scheduler; its last act is one `evio_post`, and the `coop_wake` happens back on the right thread, between two of its rounds. Run it beside a context that ticks every 200 milliseconds:

```
[ 0.000] ticker tick 1
[ 0.000] offload handing the count to a thread, limit 2500000
[ 0.200] ticker tick 2
[ 0.401] ticker tick 3
[ 0.601] ticker tick 4
[ 0.801] ticker tick 5
[ 0.867] offload primes: 183072

0.86s user  0.00s system  99% cpu  0.869 total
```

Compare that with Chapter 2, which did the same count on the same thread as its lights: there, 0.88 seconds of computation stretched across the program's whole four seconds, and the lights kept their timing by the computation's good manners. Here the computation is on a processor of its own, the ticker keeps its timing because the main thread is idle, and the elapsed time is the computation's own. Ninety-nine per cent of a processor for 0.87 seconds is two processors doing about one processor's work, which is what you would expect when one of them is mostly asleep. That is when to reach for a thread: not because something must happen concurrently, which contexts do, but because something must happen *on another processor,* or because it is code you cannot persuade to relinquish.

4.9. What it is not

The server speaks enough HTTP to be driven by a browser and no more. It does not keep connections alive, read request bodies, serve files, or speak TLS. Each of those is a program of its own, and none of them needs anything from the library that this one has not already used—which was the point of writing it. If you want a place to start, keep-alive is the interesting one: it changes the read loop, the response headers, and step 4 of the shutdown, and nothing else. What the server does show is the whole discipline of a multi-threaded program in this style: threads that own things, a letter box on each, messages across, and one mutex for the one number everybody counts.

CHAPTER FIVE

EVERYTHING AT ONCE

THE CALLS OF THIS CHAPTER

`notify_default` `notify_center_init` `notify_center_destroy`
`notify_observe` `notify_unobserve` `Post` `notify_wait`
`notify_expect` `notify_await` `notify_forget`
Notifications (Chapter 1; Section 2.7 for the waits). Five types go with them: the `notification` an observer receives, the `notify_center` it is registered with, the `notify_observer` handle that removes it, the `notify_fn` type of the callback, and the `notify_expectation` record of a wait in two halves.

`coop_open` `coop_close` `coop_spawn` `coop_cancel`
`Relinquish` `Resume` `coop_step` `coop_run` `coop_run_until`
`coop_reap` `coop_current` `coop_system_here` `coop_now_ns`
`coop_sleep_ns` `coop_sleep_ms` `coop_wait` `coop_wait_until`
`coop_wake` `coop_set_idle`
The scheduler (Chapter 2). One `coop_system` per thread; contexts on it take turns; `COOP_DEFAULT_STACK` is the stack each one gets when you pass 0.

`evio_open` `evio_close` `evio_here` `evio_socket_create`
`evio_attach` `evio_socket_close` `evio_connect` `evio_accept`
`evio_recv` `evio_send` `evio_recvfrom` `evio_sendto`
`evio_resolve` `evio_signal` `evio_post` `evio_last_error`
`evio_strerror`
The queue (Chapter 3, and Section 4.3 for `evio_post`). One `evio` per thread that does I/O or must be reachable from other threads. The failed socket is `EVIO_INVALID_SOCKET` and the timeout is $-$`EVIO_ETIMEDOUT`; a resolver failure is $-$(`EVIO_ERESOLVE`+ *code*). A backend can be forced with `EVIO_KQUEUE`, `EVIO_EPOLL` or `EVIO_IOCP`, if you ever need to.

`fio_call` `fio_open` `fio_close` `fio_pread` `fio_pwrite`
`fio_fsync` `fio_size` `fio_unlink` `fio_rename` `fio_mkdir`
`fio_rmdir` `fio_stat` `fio_readdir` `fio_start` `fio_stop`
The pool (Chapter 3, Sections 3.9–3.11). The threads that block so a queue's thread need not: files—contents and names alike—name resolution, and any other blocking call, one per `fio_call`. A file is a `fio_file`, invalid as `FIO_INVALID_FILE`, and `fio_stat` reports a path as a `fio_info`; the pool holds `FIO_DEFAULT_THREADS` until `fio_start` says otherwise.

`thrd_create` `thrd_join` `thrd_detach` `thrd_current`
`thrd_equal` `thrd_sleep` `thrd_yield` `thrd_exit`

`mtx_init` `mtx_lock` `mtx_trylock` `mtx_unlock` `mtx_destroy`
`cnd_init` `cnd_signal` `cnd_broadcast` `cnd_wait` `cnd_destroy`
`call_once` `THRD_LOCAL`
Threads (Chapter 4). C11 spellings, and C11 meanings. The status codes are `thrd_success`, `thrd_busy`, `thrd_nomem`, `thrd_error` and `thrd_timedout`; the mutex types are `mtx_plain`, `mtx_recursive` and `mtx_timed`; a once flag is initialised with `ONCE_FLAG_INIT`.

`STRIPBOARD_VERSION` `STRIPBOARD_VERSION_MAJOR`
`STRIPBOARD_VERSION_MINOR` `WIN32_LEAN_AND_MEAN`
Version macros. The last is defined by the library before it includes `<windows.h>`, so that `winsock2.h` wins; define it yourself if you include `<windows.h>` first.

Every piece is now in place: notifications and the rule that an observer runs on its own thread; the scheduler, its parking calls, and the two halves of a wait; the kernel queue and its letter box; threads that own things. The last program, `examples/harvest.c`, uses all of them at once. It fetches a list of pages over HTTP, as fast as the far end will allow, and reports what it got, and it needs no network and no other program: the service it fetches from is a thread of its own, listening on a port the system picks.

That last decision is worth defending, because a program that talks to itself sounds like a toy. It is the opposite. A fetch farm that needs an external server can only be run by somebody who has one; this one runs everywhere, on every platform, in every build, and its numbers mean the same thing each time. The server thread is also, on its own, a small lesson in the same techniques from the other side, and the two halves together are the smallest program in which each of the four mechanisms is doing something that none of the other three could do for it.

5.1. Owners and askers

Chapter 4 showed two ways for threads to agree about shared state, and this program is about the second. The first is the mutex, which lets any thread touch the state as long as it holds the lock; the server's request counter is kept that way. The second is to give the state an owner, and to have every other thread *ask*: the owner is a set of observers registered with one thread's queue, the state is whatever those observers keep, and the delivery rule of Section 1.4—an observer runs on its own thread—means that the state is touched by one thread only, without a lock, because no other thread ever runs the code that touches it.

The third verb, `Post`, is the verb of the second way. A context on any thread posts a request; the owner's observer runs between the owner's rounds, changes the state, and posts an answer; the asker, which was listening for the answer,

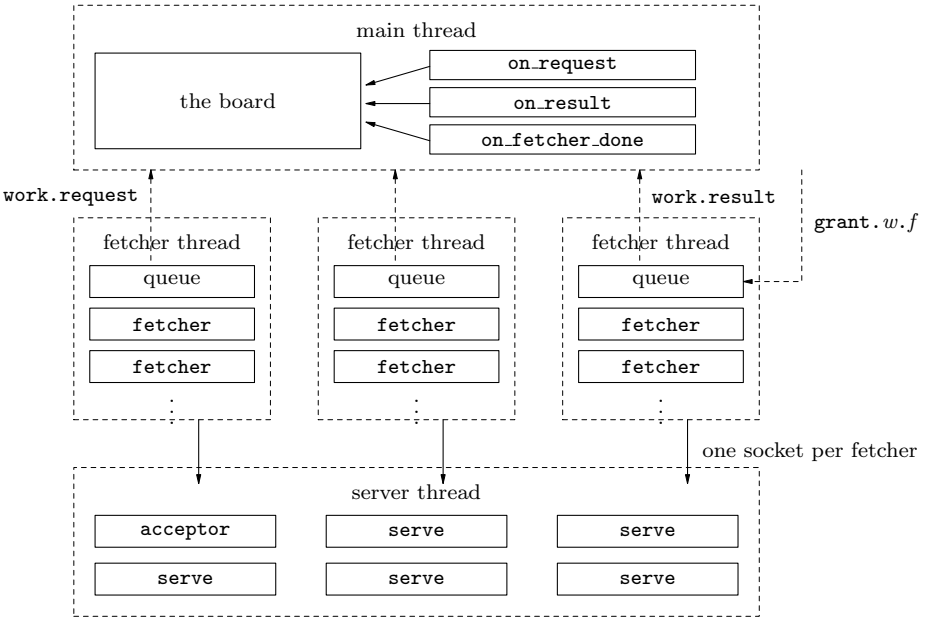

Fig. 7. The fetch farm. The main thread owns the work list and the tally and never touches a socket; each fetcher thread runs several fetcher contexts, which ask for work by name and wait for their own answer; the server thread runs one context per connection. Every arrow that crosses a thread boundary is a notification.

is woken. In the fetch farm the work list and the tally are the state, the main thread is the owner, and a dozen fetcher contexts on three threads ask.

Three kinds of thread, then, and they do not resemble each other. The main thread owns the list of pages still to fetch and the tally of what has come back, never touches a socket, and does all of its work in three observers: one that hands out a page, one that records a result, and one that counts the fetchers as they finish. The fetcher threads—three of them, by default—each run four fetcher contexts, and each context is a loop: ask for a page, fetch it, report it, repeat. Twelve pages are therefore in flight at once, on three threads, which is the arithmetic the program exists to demonstrate. The server thread runs an acceptor context and one context per open connection, and each connection sleeps for forty milliseconds before answering, which stands for the work of producing a page.

Every arrow between threads in that description is a notification, and there are only five names in the whole program:

`work.request`	a fetcher wants a page; the payload is the name it wants the answer on

`grant.`w`.`f	the answer: a path, or empty if there is none left
`work.result`	one page finished: status, bytes, microseconds
`fetcher.done`	one fetcher context has stopped
`farm.stop`	broadcast: every thread should wind up

That list is the program's interface, and it is worth writing down for any program built this way: it is the equivalent of a header file for the parts that do not include each other's headers.

5.2. The main thread owns the work

Here is the state that twelve fetchers on three threads share:

```
static struct
{
  int handed_out;
  int finished;                         /* fetchers that have stopped */
  int pages;                            /* results counted */
  int failures;
  long bytes;
  long per_thread[MAX_THREADS];
  long slowest;
  char slowest_path[PATH_MAX_LEN];
  int done;                             /* the main loop's stop flag */
} board;
```

There is no mutex, no atomic, and no volatile, and the reason is the rule of Chapter 4: every one of the three observers that touches `board` is registered on the main thread's queue, so all three run on the main thread, between two of its rounds, one at a time. A request posted from a fetcher thread while an observer is in the middle of its work waits in the main thread's letter box until the observer has returned; the fetcher that posted it has meanwhile gone on to its next line. The board has exactly one writer.

Handing out work is nine lines, and Section 2.7 has already described the idea: the request carries the name to answer on, and an empty path means there is nothing left.

```
static void on_request(const notification * n, void * ctxt)
{
   const char * answer = (const char *)n->data;
   struct grant g;
   (void)ctxt;
   if (n->len == 0 || answer[n->len - 1] != '\0') { return; }
   memset(&g, 0, sizeof g);
   if (board.handed_out < npages) {          /* empty path: nothing left */
      board.handed_out += 1;
      snprintf(g.path, sizeof g.path, "/page/%d", board.handed_out);
   }
   Post(notify_default(), answer, &board, &g, sizeof g);
}
```

Three things in that handler are the habits of the whole book. It checks `n->len` before believing the payload, and here the check is load-bearing rather than decorative, because the payload is about to be used as a string and the handler insists on a terminating zero rather than trusting one to be there. It answers by posting to a name it was *given,* so it needs to know nothing about who asked. And it says "there is no work" with an empty payload field rather than with silence, so the asker learns of the end of the work in the same way it learns of everything else, on the line after its wait.

Recording a result is the same shape, and shows what a single-writer tally can afford to do: it keeps a per-thread count, a byte total, a failure count, and the identity of the slowest page, all in ordinary C.

```
static void on_result(const notification * n, void * ctxt)
{
   const struct result * r = (const struct result *)n->data;
   (void)ctxt;
   if (n->len != sizeof *r) { return; }
   board.pages += 1;
   board.bytes += r->bytes;
   if (r->status != 200) { board.failures += 1; }
   if (r->thread >= 0 && r->thread < MAX_THREADS) {
      board.per_thread[r->thread] += 1;
   }
   if (r->micros > board.slowest) {
      board.slowest = r->micros;
      memcpy(board.slowest_path, r->path, sizeof board.slowest_path);
   }
   ...
}
```

Write that with a mutex and it is the same code with two more lines. Write it with twelve threads each updating a shared structure and it is the same code with a lock, a lock ordering rule, and an argument about whether `slowest` and `slowest_path` can disagree. They cannot here, because nothing can interleave between those two assignments.

The third observer notices when the last fetcher has stopped, broadcasts `farm.stop`, and sets the flag the main loop is watching:

```
static void on_fetcher_done(const notification * n, void * ctxt)
{
   (void)n; (void)ctxt;
   board.finished += 1;
   if (board.finished == nthreads * nfetchers) {
      Post(notify_default(), "farm.stop", &board, NULL, 0);
      board.done = 1;
   }
}
```

and the main thread's whole working life is one line,

```
coop_run_until(&sys, &board.done);   /* the board lives here */
```

with no contexts of its own. It sits in its kernel queue, wakes when a notification arrives, runs the observer, and goes back to sleep. This is the second shape of Section 2.4, and it is why `coop_run_until` tolerates an empty list.

5.3. A fetcher is a straight line

Here is one of the twelve, entire:

```
static void fetcher(int * cancel, void * ctxt)
{
   struct slot * f = (struct slot *)ctxt;
   struct farm * w = f->farm;
   notify_center * c = notify_default();

   while (!*cancel) {
      notify_expectation e;
      struct grant g;
      struct result res;
      int64_t start;
      long bytes = 0;

      notify_expect(c, f->answer, &g, sizeof g, &e);
      Post(c, "work.request", w, f->answer, strlen(f->answer) + 1);
      if (notify_await(&e, -1) != (long)sizeof g) { break; }
      if (g.path[0] == '\0') { break; }          /* the list is done */

      start = coop_now_ns();
      memset(&res, 0, sizeof res);
      res.thread = w->id;
      res.status = fetch_one(&w->io, w->port, g.path, &bytes);
      res.bytes = bytes;
      res.micros = (long)((coop_now_ns() - start) / 1000);
      memcpy(res.path, g.path, sizeof res.path);
      Post(c, "work.result", w, &res, sizeof res);
   }
   Post(c, "fetcher.done", w, NULL, 0);
}
```

Read it as a description of a job rather than as code and it says exactly what a person doing the job would say: ask for the next one, and if there is no next one, stop; fetch it; say what happened; ask again. There is no state machine, no completion handler, and no shared cursor into the work list. The whole of the fetcher's state—which page it has, when it started, what came back—is in local variables of this function, which is where a C programmer expects it to be.

Three things in it are the book's habits, and each of them is doing real work. The expectation is registered before the request is posted, which is Section 2.7's pattern exactly, and the lost wake-up it guards against is not hypothetical here: the main thread may answer within microseconds of the request arriving, and if the fetcher posted first and only then registered to listen, the grant could be

delivered to nobody. The name is this fetcher's own—`f->answer` is `grant.2.1`, thread two, fetcher one, built once when the context started—because a name shared between the four fetchers of a thread would let one grant wake all four, and four fetchers would fetch the same page, which is the kind of bug that leaves the totals slightly wrong and no evidence of why. And the end of the work arrives as data, not as silence: an empty path is the answer to "is there more?", it comes back on the same line as everything else, and nothing has to be cancelled to shut the fetchers down in the normal case.

The fetch itself is the client of Chapter 3 with the DNS taken out: create a socket, connect with a five-second timeout, send the request, receive until the server closes, and read the status out of the first line. While this context is inside any of those calls, the other three fetchers on its thread are using the thread.

5.4. The server is the same idea from the other side

The server thread opens a scheduler and a queue like any other, binds to port 0 so that the system picks an unused port, and finds out which one it got:

```
addr.sin_port = 0;                  /* let the system choose the port */
if (listener == EVIO_INVALID_SOCKET
 || bind(listener, (struct sockaddr *)&addr, sizeof addr) != 0
 || listen(listener, 64) != 0
 || getsockname(listener, (struct sockaddr *)&addr, &alen) != 0) {
   report_ready();
   return 1;
}
srv.port = ntohs(addr.sin_port);
```

Its acceptor is Chapter 4's, with a shorter poll, and there is no hand-over to another thread: this server has one thread, so the connection context is spawned on the spot.

```
int r = evio_accept(&srv.io, listener, &c, 200);
if (r == -EVIO_ETIMEDOUT) { continue; }    /* look at *cancel again */
if (r < 0) { coop_sleep_ms(20); continue; }
if (coop_spawn(&srv.sys, serve, (void *)(intptr_t)c) == NULL) {
   evio_socket_close(c);
}
```

The connection context reads the request, sleeps for the page's worth of work, and answers:

```
coop_sleep_ms(work_ms);             /* "producing the page": only this
                                       context waits, not the thread */
```

That one line is the whole demonstration on the server side. Twelve connections are open; each is sleeping forty milliseconds; the thread they share is idle for essentially the whole time. Forty milliseconds is paid once for all twelve, not twelve times. The server counts its open connections so that it can say so afterwards, and, being one thread, needs no lock for the counter:

```
srv.live += 1;                        /* this thread only: no lock */
if (srv.live > srv.peak) { srv.peak = srv.live; }
```

5.5. Starting, and stopping

Both ends of the program's life are the patterns of Chapter 4, and this is what they look like when there is nothing else in the way.

Starting is a barrier, twice. The main thread starts the server thread and waits for it to report in, because until it has, `srv.port` is not yet a port number. Then it starts the fetcher threads and waits for all of them, because until they have opened their queues, nothing may post into them.

```
thrd_create(&srv.thread, server_main, NULL);
await_ready(1);
...
for (i = 0; i < nthreads; i += 1) { thrd_create(...); }
await_ready(1 + nthreads);
```

Stopping is the reverse, and it happens by itself. The fetchers stop because the work ran out; the last one to stop causes `farm.stop` to be broadcast; each fetcher thread and the server thread hears it on its own thread, sets its own flag, leaves `coop_run_until`, drains with `coop_run`, and closes its queue and scheduler. The main thread joins them all and prints the totals. The one context that has to be told is the acceptor, because it is parked in `evio_accept` and no page-fetching will ever end that wait:

```
static void on_server_stop(const notification * n, void * ctxt)
{
   (void)n; (void)ctxt;
   srv.stop = 1;
   if (srv.acceptor) { coop_cancel(srv.acceptor); }
}
```

which is Section 2.6's rule in one line: the cancel sets the flag, and the 200-millisecond accept timeout is what makes the acceptor look at it. The next section puts a price on that 200 milliseconds.

5.6. What it prints

```
60 pages of 40 ms, 4 fetchers on each of 3 threads, server on port 57531
[ 0.083] 20 pages fetched
[ 0.164] 40 pages fetched
[ 0.205] 60 pages fetched
60 pages, 9882 bytes, 0 failed, in 0.366 s
thread 0 fetched 20 pages
thread 1 fetched 20 pages
thread 2 fetched 20 pages
server answered 60 requests, 12 of them open at once
40 ms of page work each, 2.400 s in all, done in 0.366 s
slowest page /page/9 at 41208 us
```

The last two lines are the point. Sixty pages, each costing the server forty milliseconds of work, is 2.4 seconds of work; it took 0.366 seconds. The peak of twelve connections open at once is the four fetchers on each of three threads, all in flight together, which is where the factor came from.

The split of twenty, twenty and twenty is worth a remark, because nothing in the program arranges it. There is no round-robin here and no attempt at fairness: every fetcher asks for the next page whenever it is ready for one, so a thread that happened to be slow would simply ask less often. Work handed out on request balances itself, which is the cheapest form of load balancing there is and the reason a program should prefer it to any scheme that decides in advance.

Vary the two arguments and the arithmetic is visible:

fetchers	*threads*	*in flight*	*elapsed*
1	1	1	2.598 s
2	1	2	1.378 s
4	1	4	0.772 s
4	2	8	0.487 s
4	3	12	0.365 s
8	3	24	0.286 s

Every one of those is within a few milliseconds of $2.4/n + 0.2$ seconds, where n is the number in flight. The $2.4/n$ is the service's work, divided by how much of it is in flight at once. The constant 0.2 is the acceptor noticing that it has been cancelled: it is parked in an `evio_accept` with a 200-millisecond deadline, and on average it waits out most of one. Shorten that deadline to 20 milliseconds and the constant becomes 0.05; that is the trade Section 3.4 described, priced. The one row that says something else is the first: one fetcher, one page at a time, is 2.598 seconds for 2.4 seconds of work, about three milliseconds a page of asking and answering across two threads. Twelve in flight hides all of it.

5.7. What each mechanism was for

It is worth taking the program apart, because the argument for using four mechanisms is that removing any one of them makes it worse in a specific way.

Without the scheduler, each fetcher would be a state machine, or a thread. As a state machine, the twelve-line loop of Section 5.3 becomes a structure and a switch, and the local variables `start`, `bytes` and `res` become fields somebody maintains. As a thread, twelve becomes twelve threads and the count no longer scales: the program's concurrency would be bounded by how many threads it is willing to have, rather than by how many pages it wants in flight.

Without the queue, a fetcher blocked in `recv` would block its thread and the other three fetchers on it. Twelve in flight would need twelve threads again.

Without threads, everything would be on one processor. This program would barely notice, because it is not doing arithmetic; change the fetchers to parse or checksum what they download and it would notice immediately, and the change is one line: more threads, fewer fetchers each.

Without notifications, the work list and the tally would be shared memory under a lock, touched by twelve fetchers on three threads. That is the version most programs are, and it is not unwritable—it is a mutex around `board` and care about where it is taken. What it costs is that the *policy*—which page goes to whom, what counts as a failure, when the work is finished—stops being in one place. There is no function in the mutex version where the hand-out policy could be written, because there is no one who sees the requests in order; fairness, priorities, a limit per host, an account of who waited how long, each of these is a line in `on_request` here and a redesign around a condition variable there. The owner is the shape to choose when the state has a policy, and `Post` is the whole of what the askers need to know about it.

None of the four is doing anything clever. The point is that they compose: a fetcher is a straight line *because* the scheduler lets it park, its parks are cheap *because* the queue holds them, the threads are useful *because* nothing in the library is shared between them, and the sharing that does happen is safe *because* it goes through one owner by name.

5.8. Where to take it

The program is a hundred and fifty lines longer than it needs to be for what it does, and every one of those lines is somewhere to start. Replace the port number with a host and a resolver—Chapter 3's, unchanged—and the farm fetches from the internet; add a count per host on the board, and have `on_request` decline to hand out a page whose host is at its limit, and it becomes a polite crawler. Put a failed path back on the work list instead of counting it, with an attempt count, and give up after three; that is four lines in `on_result`, and it is a change to the policy only, since no fetcher is touched. Do something with the pages: parsing them is arithmetic, which means it belongs on the fetcher's thread with

a `Relinquish` in its inner loop, or on a thread of its own by the pattern of Section 4.8. And change what the pages are, because nothing in the main thread or the fetchers knows that this is HTTP: replace `fetch_one` with anything that turns a string into a result and the rest of the program is unchanged, which is the last thing the four mechanisms buy, and the one that is hardest to notice until you need it.

That is also where this book ends, because it is where the four ideas meet. A fetcher is a context, written as the sequential function it is; it waits in the kernel queue of its thread, which costs nothing while it waits; the threads own their fetchers and the main thread owns the work, and none of them touches the others' memory; and the only thing that crosses between them is a notification with a name.

INDEX

STRECK.AI STOCKHOLM