In figure 1.5, we sketch out the process and threads for listing 1.4. We create one child process that prints its process ID, and we also print out the parent process ID to prove that we are running different processes. Multiprocessing is typically best when we have CPU-intensive work.

Multithreading and multiprocessing may seem like magic bullets to enable concurrency with Python. However, the power of these concurrency models is hindered by an implementation detail of Python—the global interpreter lock.

derstanding the global interpreter lock

The *global interpreter lock*, abbreviated GIL and pronounced *gill*, is a controversial topic in the Python community. Briefly, the GIL prevents one Python process from executing more than one Python bytecode instruction at any given time. This means that even if we have multiple threads on a machine with multiple cores, a Python process can have only one thread running Python code at a time. In a world where we have CPUs with multiple cores, this can pose a significant challenge for Python developers looking to take advantage of multithreading to improve the performance of their application.

NOTE Multiprocessing can run multiple bytecode instructions concurrently because each Python process has its own GIL.

So why does the GIL exist? The answer lies in how memory is managed in CPython. In CPython, memory is managed primarily by a process known as *reference counting*. Reference counting works by keeping track of who currently needs access to a particular Python object, such as an integer, dictionary, or list. A reference count is an integer keeping track of how many places reference that particular object. When someone no longer needs that referenced object, the reference count is decremented,

and when someone else needs it, it is incremented. When the reference count reaches zero, no one is referencing the object, and it can be deleted from memory.

What is CPython?

CPython is the reference implementation of Python. By *reference implementation* we mean it is the standard implementation of the language and is used as the *reference* for proper behavior of the language. There are other implementations of Python such as Jython, which is designed to run on the Java Virtual Machine, and IronPython, which is designed for the .NET framework.

The conflict with threads arises in that the implementation in CPython is not thread safe. When we say CPython is not *thread safe*, we mean that if two or more threads modify a shared variable, that variable may end in an unexpected state. This unexpected state depends on the order in which the threads access the variable, commonly known as a *race condition*. Race conditions can arise when two threads need to reference a Python object at the same time.

As shown in figure 1.6, if two threads increment the reference count at one time, we could face a situation where one thread causes the reference count to be zero when the object is still in use by the other thread. The likely result of this would be an application crash when we try to read the potentially deleted memory.

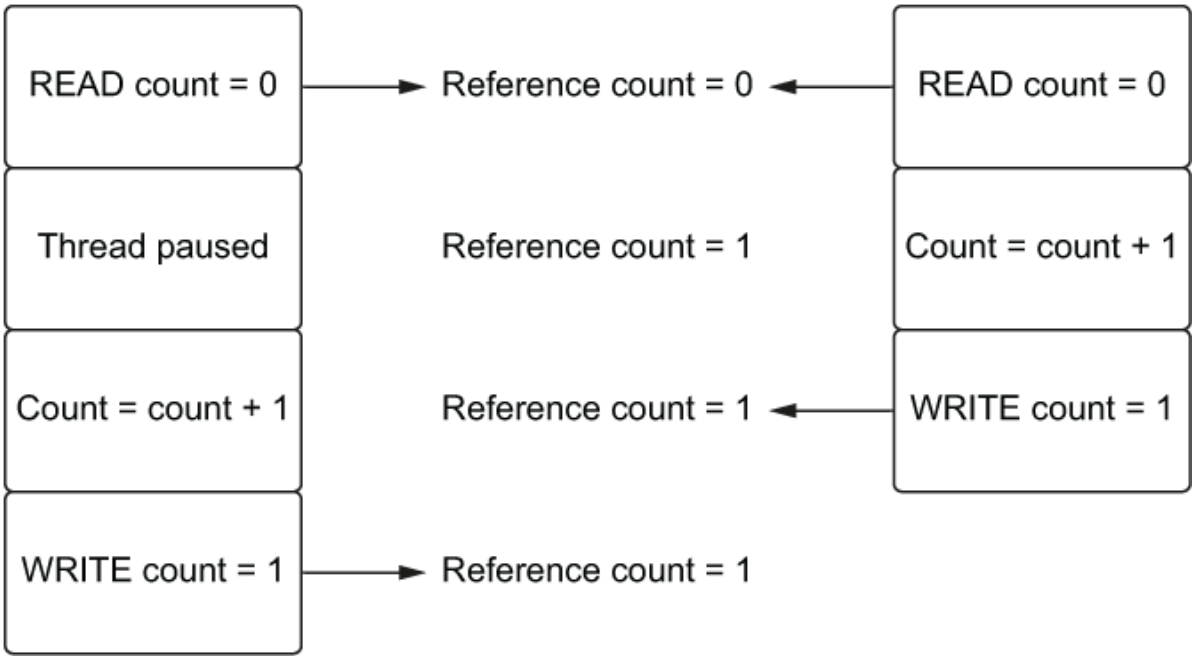

Figure 1.6 A race condition where two threads try to increment a reference count simultaneously. Instead of an expected count of two, we get one.

To demonstrate the effect of the GIL on multithreaded programming, let's examine the CPU-intensive task of computing the *n*th number in the Fibonacci sequence. We'll use a fairly slow implementation of the algorithm to demonstrate a time-intensive operation. A proper solution would utilize memoization or mathematical techniques to improve performance.

Listing 1.5 Generating and timing the Fibonacci sequence

```
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
```

```
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    print(f'fib({number}) is {fib(number)}')


def fibs_no_threading():
    print_fib(40)
    print_fib(41)


start = time.time()

fibs_no_threading()

end = time.time()

print(f'Completed in {end - start:.4f} seconds.')
```

This implementation uses recursion and is overall a relatively slow algorithm, requiring exponential *O*(*2^N*) time to complete. If we are in a situation where we need to print two Fibonacci numbers, it is easy enough to synchronously call them and time the result, as we have done in the preceding listing.

Depending on the speed of the CPU we run on, we will see different timings, but running the code in listing 1.5 will yield output similar to the following:

```
fib(40) is 63245986
fib(41) is 102334155
```

```
Completed in 65.1516 seconds.
```

This is a fairly long computation, but our function calls to `print_fibs` are independent from one another. This means that they can be put in multiple threads that our CPU can, in theory, run concurrently on multiple cores, thus, speeding up our application.

Listing 1.6 Multithreading the Fibonacci sequence

```
import threading
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

def fibs_with_threads():
    fortieth_thread = threading.Thread(target=print_fib, arg
    forty_first_thread = threading.Thread(target=print_fib,

    fortieth_thread.start()
    forty_first_thread.start()

    fortieth_thread.join()
    forty_first_thread.join()
```

```
start_threads = time.time()

fibs_with_threads()

end_threads = time.time()

print(f'Threads took {end_threads - start_threads:.4f} secon
```

In the preceding listing, we create two threads, one to compute `fib(40)` and one to compute `fib(41)` and start them concurrently by calling `start()` on each thread. Then we make a call to `join()`, which will cause our main program to wait until the threads finish. Given that we start our computation of `fib(40)` and `fib(41)` simultaneously and run them concurrently, you would think we could see a reasonable speedup; however, we will see an output like the following even on a multi-core machine.

```
fib(40) is 63245986
fib(41) is 102334155
Threads took 66.1059 seconds.
```

Our threaded version took almost the same amount of time. In fact, it was even a little slower! This is almost entirely due to the GIL and the overhead of creating and managing threads. While it is true the threads run concurrently, only one of them is allowed to run Python code at a time due to the lock. This leaves the other thread in a waiting state until the first one completes, which completely negates the value of multiple threads.

s the GIL ever released?

Based on the previous example, you may be wondering if concurrency in Python can ever happen with threads, given that the GIL prevents running two lines of Python concurrently. The GIL, however, is not held forever such that we can't use multiple threads to our advantage.

The global interpreter lock is released when I/O operations happen. This lets us employ threads to do concurrent work when it comes to I/O, but not for CPU-bound Python code itself (there are some notable exceptions that release the GIL for CPU-bound work in certain circumstances, and we'll look at these in a later chapter). To illustrate this, let's use an example of reading the status code of a web page.

Listing 1.7 Synchronously reading status codes

```
import time
import requests


def read_example() -> None:
    response = requests.get('https:/ / www .example .com')
    print(response.status_code)


sync_start = time.time()

read_example()
read_example()

sync_end = time.time()
```

```
print(f'Running synchronously took {sync_end - sync_start:.4
```

In the preceding listing, we retrieve the contents of example.com and print the status code twice. Depending on our network connection speed and our location, we'll see output similar to the following when running this code:

```
200
200
Running synchronously took 0.2306 seconds.
```

Now that we have a baseline for what a synchronous version looks like, we can write a multithreaded version to compare to. In our multithreaded version, in an attempt to run them concurrently, we'll create one thread for each request to example.com.

Listing 1.8 Multithreaded status code reading

```
import time
import threading
import requests


def read_example() -> None:
    response = requests.get('https:/ / www .example .com')
    print(response.status_code)


thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)
```

```
thread_start = time.time()

thread_1.start()
thread_2.start()

print('All threads running!')

thread_1.join()
thread_2.join()

thread_end = time.time()

print(f'Running with threads took {thread_end - thread_start
```

When we execute the preceding listing, we will see output like the following, depending again on our network connection and location:

```
All threads running!
200
200
Running with threads took 0.0977 seconds.
```

This is roughly two times faster than our original version that did not use threads, since we've run the two requests at roughly the same time! Of course, depending on your internet connection and machine specs, you will see different results, but the numbers should be directionally similar.

So how is it that we can release the GIL for I/O but not for CPU-bound operations? The answer lies in the system calls that are made in the background. In the case of I/O, the low-level system calls are outside of the Python runtime. This allows the GIL to be released because it is not interacting with Python objects directly. In this case, the GIL is only reacquired when the data received is translated back into a Python object. Then, at the operating-system level, the I/O operations execute concurrently. This model gives us concurrency but not parallelism. In other languages, such as Java or C++, we would get true parallelism on multi-core machines because we don't have the GIL and can execute simultaneously. However, in Python, because of the GIL, the best we can do is concurrency of our I/O operations, and only one piece of Python code is executing at a given time.

syncio and the GIL

asyncio exploits the fact that I/O operations release the GIL to give us concurrency, even with only one thread. When we utilize asyncio we create objects called *coroutines*. A coroutine can be thought of as executing a lightweight thread. Much like we can have multiple threads running at the same time, each with their own concurrent I/O operation, we can have many coroutines running alongside one another. While we are waiting for our I/O-bound coroutines to finish, we can still execute other Python code, thus, giving us concurrency. It is important to note that asyncio does not circumvent the GIL, and we are still subject to it. If we have a CPU-bound task, we still need to use multiple processes to execute it concurrently (which can be done with asyncio itself); otherwise, we will cause performance issues in our application. Now that we know it is possible to achieve concurrency for I/O with only a single thread, let's dive into the specifics of how this works with non-blocking sockets.

w single-threaded concurrency works

In the previous section, we introduced multiple threads as a mechanism for achieving concurrency for I/O operations. However, we don't need multiple threads to achieve this kind of concurrency. We can do it all within the confines of one process and one thread. We do this by exploiting the fact that, at the system level, I/O operations can be completed concurrently. To better understand this, we'll need to dive into how sockets work and, in particular, how non-blocking sockets work.

Vhat is a socket?

A *socket* is a low-level abstraction for sending and receiving data over a network. It is the basis for how data is transferred to and from servers. Sockets support two main operations: sending bytes and receiving bytes. We write bytes to a socket, which will then get sent to a remote address, typically some type of server. Once we've sent those bytes, we wait for the server to write its response back to our socket. Once these bytes have been sent back to our socket, we can then read the result.

Sockets are a low-level concept and are fairly easy to understand if you think of them as mailboxes. You can put a letter in your mailbox that your letter carrier then picks up and delivers to the recipient's mailbox. The recipient opens their mailbox and your letter. Depending on the contents, the recipient may send you a letter back. In this analogy, you may think of the letter as the data or bytes we want to send. Consider that the act of putting a letter into the mailbox is writing the bytes to a socket, and opening the mailbox to read the letter is reading bytes from a socket. The letter carrier can be thought of as the transfer mechanism over the internet, routing the data to the correct address.

In the case of getting the contents from example.com as we saw earlier, we open a socket that connects to example.com's server. We then write a request to get the contents to that socket and wait for the server to reply with the result: in this case, the HTML of the web page. We can visualize the flow of bytes to and from the server in figure 1.7.

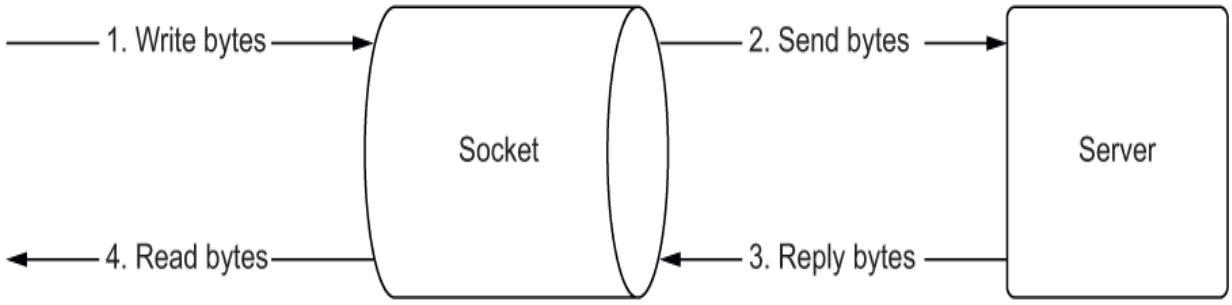

Figure 1.7 Writing bytes to a socket and reading bytes from a socket

Sockets are *blocking* by default. Simply put, this means that when we are waiting for a server to reply with data, we halt our application or *block* it until we get data to read. Thus, our application stops running any other tasks until we get data from the server, an error happens, or there is a timeout.

At the operating system level, we don't need to do this blocking. Sockets can operate in *non-blocking* mode. In non-blocking mode, when we write bytes to a socket, we can just fire and forget the write or read, and our application can go on to perform other tasks. Later, we can have the operating system tell us that we received bytes and deal with it at that time. This lets the application do any number of things while we wait for bytes to come back to us. Instead of blocking and waiting for data to come to us, we become more reactive, letting the operating system inform us when there is data for us to act on.

In the background, this is performed by a few different event notification systems, depending on which operating system we're running. asyncio is abstracted enough that it switches between the different notification systems, depending on which one our operating system supports. The following are the event notification systems used by specific operating systems:

- *kqueue*—FreeBSD and MacOS
- *epoll*—Linux
- *IOCP (I/O completion port)*—Windows

These systems keep track of our non-blocking sockets and notify us when they are ready for us to do something with them. This notification system is the basis of how asyncio can achieve concurrency. In asyncio's model of concurrency, we have only one thread executing Python at any given time. When we hit an I/O operation, we hand it over to our operating system's event notification system to keep track of it for us. Once we have done this handoff, our Python thread is free to keep running other Python code or add more non-blocking sockets for the OS to keep track of for us. When our I/O operation finishes, we "wake up" the task that was waiting for the result and then proceed to run any other Python code that came after that I/O operation. We can visualize this flow in figure 1.8 with a few separate operations that each rely on a socket.

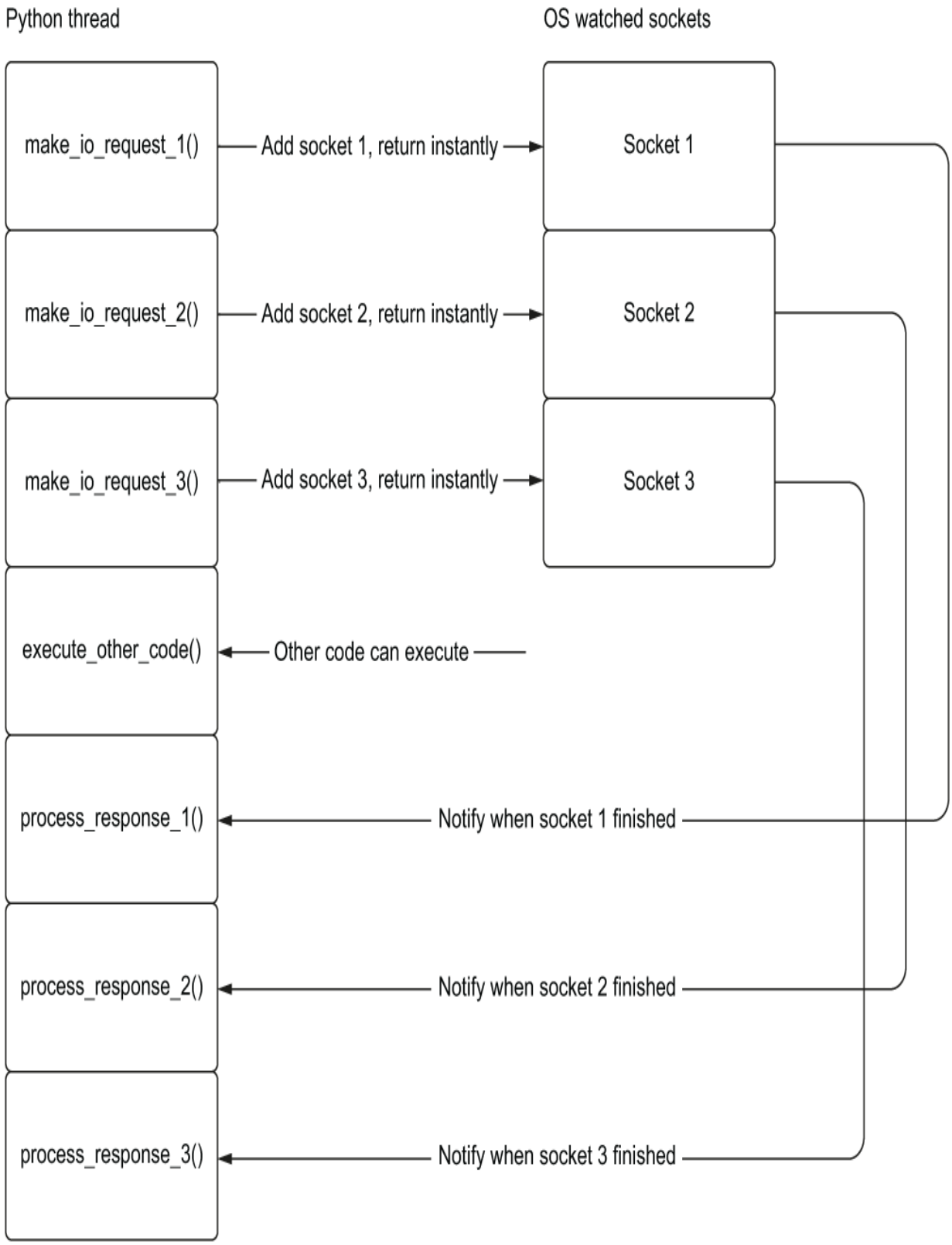

Figure 1.8 Making a non-blocking I/O request returns immediately and tells the O/S

to watch sockets for data. This allows `execute_other_code()` to run right away instead of waiting for the I/O requests to finish. Later, we can be alerted when I/O is complete and process the response.

But how do we keep track of which tasks are waiting for I/O as opposed to ones that can just run because they are regular Python code? The answer lies in a construct called an event loop.

w an event loop works

An event loop is at the heart of every asyncio application. *Event loops* are a fairly common design pattern in many systems and have existed for quite some time. If you've ever used JavaScript in a browser to make an asynchronous web request, you've created a task on an event loop. Windows GUI applications use what are called message loops behind the scenes as a primary mechanism for handling events such as keyboard input, while still allowing the UI to draw.

The most basic event loop is extremely simple. We create a queue that holds a list of events or messages. We then loop forever, processing messages one at a time as they come into the queue. In Python, a basic event loop might look something like this:

```
from collections import deque

messages = deque()

while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

In asyncio, the event loop keeps a queue of tasks instead of messages. Tasks are wrappers around a coroutine. A coroutine can pause execution when it hits an I/O-bound operation and will let the event loop run other tasks that are not waiting for I/O operations to complete.

When we create an event loop, we create an empty queue of tasks. We can then add tasks into the queue to be run. Each iteration of the event loop checks for tasks that need to be run and will run them one at a time until a task hits an I/O operation. At that time the task will be "paused," and we instruct our operating system to watch any sockets for I/O to complete. We then look for the next task to be run. On every iteration of the event loop, we'll check to see if any of our I/O has completed; if it has, we'll "wake up" any tasks that were paused and let them finish running. We can visualize this as follows in figure 1.9: the main thread submits tasks to the event loop, which can then run them.

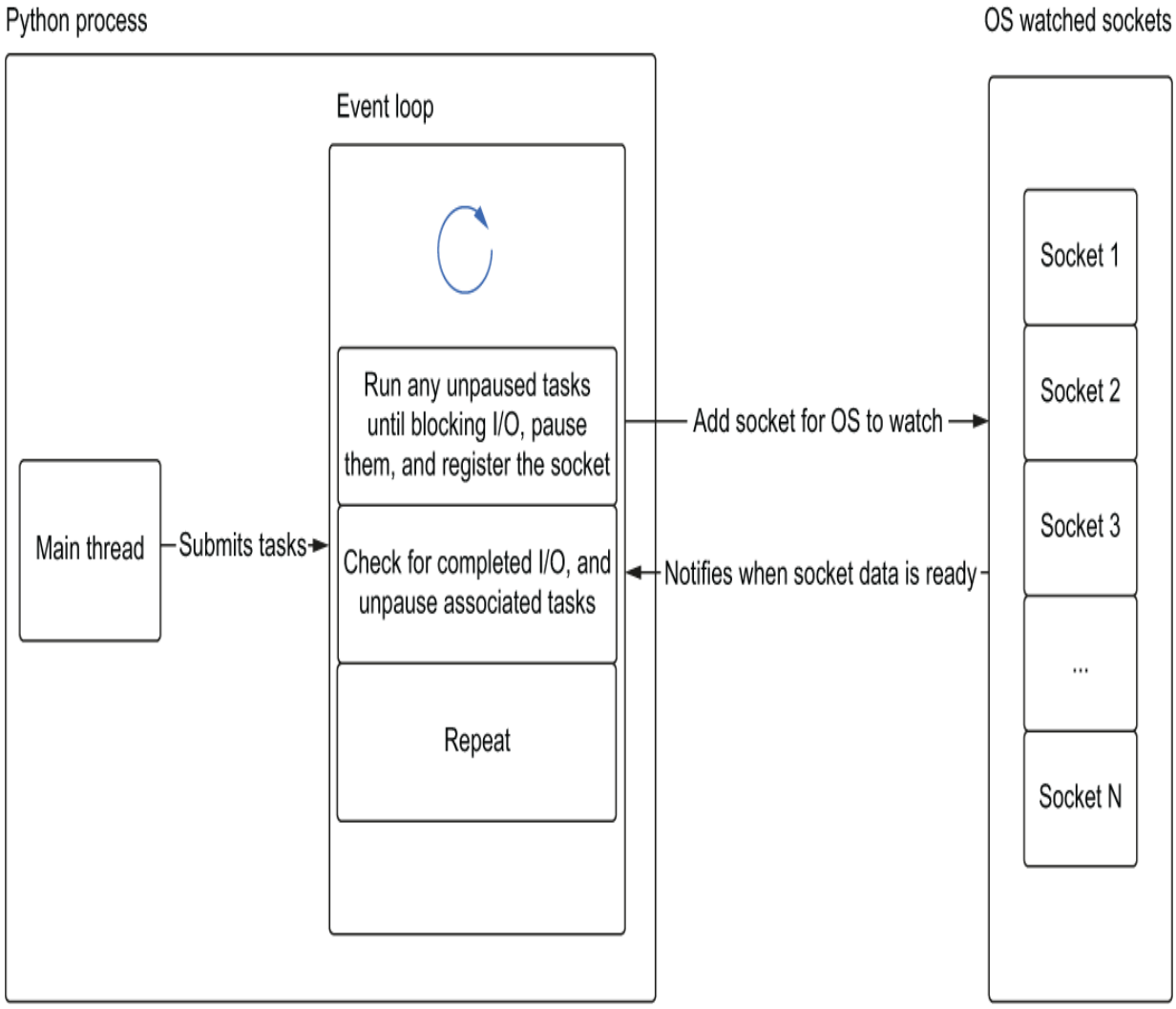

Figure 1.9 An example of a thread submitting tasks to the event loop

To illustrate this, let's imagine we have three tasks that each make an asynchronous web request. Imagine these tasks have a bit of code to do setup, which is CPU-bound, then they make a web request, and they follow with some CPU-bound postprocessing code. Now, let's submit these tasks to the event loop simultaneously. In pseudocode, we would write something like this:

```
def make_request():
    cpu_bound_setup()
```

```
    io_bound_web_request()
    cpu_bound_postprocess()

task_one = make_request()
task_two = make_request()
task_three = make_request()
```

All three tasks start with CPU-bound work and we are single-threaded, so only the first task starts executing code, and the other two are left waiting to run. Once the CPU-bound setup work is finished in Task 1, it hits an I/O-bound operation and will pause itself to say, “I’m waiting for I/O; any other tasks waiting to run can run.”

Once this happens, Task 2 can begin executing. Task 2 starts its CPU-bound code and then pauses, waiting for I/O. At this time both Task 1 and Task 2 are waiting concurrently for their network request to complete. Since Tasks 1 and 2 are both paused waiting for I/O, we start running Task 3.

Now imagine once Task 3 pauses to wait for its I/O to complete, the web request for Task 1 has finished. We’re now alerted by our operating system’s event notification system that this I/O has finished. We can now resume executing Task 1 while both Task 2 and Task 3 are waiting for their I/O to finish.

In figure 1.10, we show the execution flow of the pseudocode we just described. If we look at any vertical slice of this diagram, we can see that only one CPU-bound piece of work is running at any given time; however, we have up to two I/O-bound operations happening concurrently. This overlapping of waiting for I/O per each task is where the real time savings of asyncio comes in.

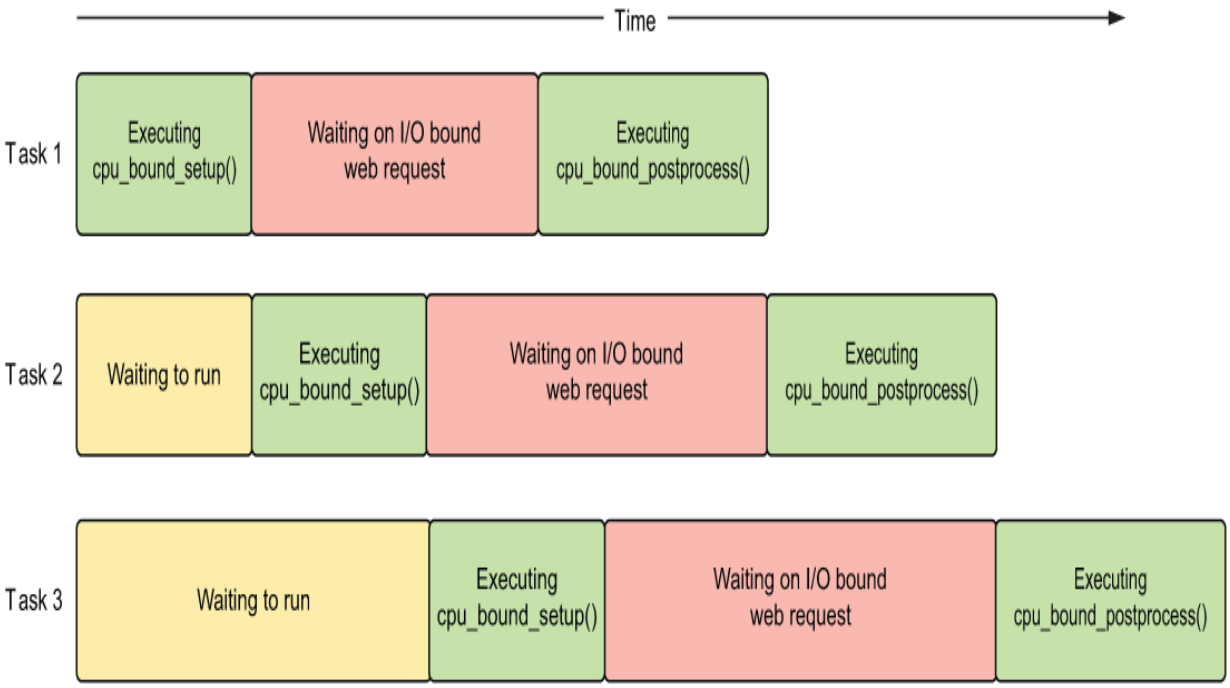

Figure 1.10 Executing multiple tasks concurrently with I/O operations

ary

- CPU-bound work is work that primarily utilizes our computer's processor whereas I/O-bound work primarily utilizes our network or other input/output devices. asyncio primarily helps us make I/O-bound work concurrent, but it exposes APIs for making CPU-bound work concurrent as well.
- Processes and threads are the basic most units of concurrency at the operating system level. Processes can be used for I/O and CPU-bound workloads and threads can (usually) only be used to manage I/O-bound work effectively in Python due to the GIL preventing code from executing in parallel.
- We've seen how, with non-blocking sockets, instead of stopping our application while we wait for data to come in, we can instruct the operating system to tell us when data has come in. Exploiting this is part of what allows asyncio to achieve concurrency with only a single thread.

- We’ve introduced the event loop, which is the core of asyncio applications. The event loop loops forever, looking for tasks with CPU-bound work to run while also pausing tasks that are waiting for I/O.