1 Getting to know asyncio

This chapter covers

- What asyncio is and the benefits it provides
- Concurrency, parallelism, threads, and processes
- The global interpreter lock and the challenges it poses to concurrency
- How non-blocking sockets can achieve concurrency with only one thread
- The basics of how event-loop-based concurrency works

Many applications, especially in today's world of web applications, rely heavily on I/O (input/output) operations. These types of operations include downloading the contents of a web page from the internet, communicating over a network with a group of microservices, or running several queries together against a database such as MySQL or Postgres. A web request or communication with a microservice may take hundreds of milliseconds, or even seconds if the network is slow. A database query could be time intensive, especially if that database is under high load or the query is complex. A web server may need to handle hundreds or thousands of requests at the same time.

Making many of these I/O requests at once can lead to substantial performance issues. If we run these requests one after another as we would in a sequentially run application, we'll see a compounding performance impact. As an example, if we're writing an application that needs to download 100 web pages or run 100 queries, each of which takes 1 second to execute, our application will take at least 100 seconds to run. However, if we were to exploit concurrency and start the downloads and wait simultaneously, in theory, we could complete these operations in as little as 1 second.

asyncio was first introduced in Python 3.4 as an additional way to handle these highly concurrent workloads outside of multithreading and multiprocessing. Properly utilizing this library can lead to drastic performance and resource utilization improvements for applications that use I/O operations, as it allows us to start many of these long-running tasks together.

In this chapter, we'll introduce the basics of concurrency to better understand how we can achieve it with Python and the asyncio library. We'll explore the differences between CPU-bound work and I/O-bound work to know which concurrency model best suits our specific needs. We'll also learn about the basics of processes and threads and the unique challenges to concurrency in Python caused by its global interpreter lock (GIL). Finally, we'll get an understanding of how we can utilize a concept called *non-blocking I/O* with an event loop to achieve concurrency using only one Python process and thread. This is the primary concurrency model of asyncio.

ıat is asyncio?

In a synchronous application, code runs sequentially. The next line of code runs as soon as the previous one has finished, and only one thing is happening at once. This model works fine for many, if not most, applications. However, what if one line of code is especially slow? In that case, all other code after our slow line will be stuck waiting for that line to complete. These potentially slow lines can block the application from running any other code. Many of us have seen this before in buggy user interfaces, where we happily click around until the application freezes, leaving us with a spinner or an unresponsive user interface. This is an example of an application being blocked leading to a poor user experience.

While any operation can block an application if it takes long enough, many applications will block waiting on I/O. I/O refers to a computer's input and output devices such as a keyboard, hard drive, and, most commonly, a network card. These operations wait for user input or retrieve the contents from a web-based API. In a synchronous application, we'll be stuck waiting for those operations to complete before we can run anything else. This can cause performance and responsiveness issues, as we can only have one long operation running at any given time, and that operation will stop our application from doing anything else.

One solution to this issue is to introduce concurrency. In the simplest terms, *concurrency* means allowing more than one task being handled at the same time. In the case of concurrent I/O, examples include allowing multiple web requests to be made at the same time or allowing simultaneous connections to a web server.

There are several ways to achieve this concurrency in Python. One of the most recent additions to the Python ecosystem is the asyncio library. *asyncio* is short for *asynchronous I/O*. It is a Python library that allows us to run code using an asynchronous programming model. This lets us handle multiple I/O operations at once, while still allowing our application to remain responsive.

So what is asynchronous programming? It means that a particular long-running task can be run in the background separate from the main application. Instead of blocking all other application code waiting for that long-running task to be completed, the system is free to do other work that is not dependent on that task. Then, once the long-running task is completed, we'll be notified that it is done so we can process the result.

In Python version 3.4, asyncio was first introduced with decorators alongside generator `yield from` syntax to define coroutines. A coroutine is a method that

can be paused when we have a potentially long-running task and then resumed when that task is finished. In Python version 3.5, the language implemented first-class support for coroutines and asynchronous programming when the keywords `async` and `await` were explicitly added to the language. This syntax, common in other programming languages such as C# and JavaScript, allows us to make asynchronous code look like it is run synchronously. This makes asynchronous code easy to read and understand, as it looks like the sequential flow most software engineers are familiar with. asyncio is a library to execute these coroutines in an asynchronous fashion using a concurrency model known as a *single-threaded event loop*.

While the name of asyncio may make us think that this library is only good for I/O operations, it has functionality to handle other types of operations as well by interoperating with multithreading and multiprocessing. With this interoperability, we can use `async` and `await` syntax with threads and processes making these workflows easier to understand. This means this library not only is good for I/O based concurrency but can also be used with code that is CPU intensive. To better understand what type of workloads asyncio can help us with and which concurrency model is best for each type of concurrency, let's explore the differences between I/O and CPU-bound operations.

ıat is I/O-bound and what is CPU-bound?

When we refer to an operation as I/O-bound or CPU-bound we are referring to the limiting factor that prevents that operation from running faster. This means that if we increased the performance of what the operation was bound on, that operation would complete in less time.

In the case of a CPU-bound operation, it would complete faster if our CPU was more powerful, for instance by increasing its clock speed from 2 GHz to 3 GHz. In the case of an I/O-bound operation, it would get faster if our I/O devices could handle more data in less time. This could be achieved by increasing our network bandwidth through our ISP or upgrading to a faster network card.

CPU-bound operations are typically computations and processing code in the Python world. An example of this is computing the digits of pi or looping over the contents of a dictionary, applying business logic. In an I/O-bound operation we spend most of our time waiting on a network or other I/O device. An example of an I/O-bound operation would be making a request to a web server or reading a file from our machine's hard drive.

Listing 1.1 I/O-bound and CPU-bound operations

```
import requests

response = requests.get('https:/ / www .example .com')     (1)

items = response.headers.items()

headers = [f'{key}: {header}' for key, header in items]    (2)

formatted_headers = '\n'.join(headers)                     (3)

with open('headers.txt', 'w') as file:
    file.write(formatted_headers)                          (4)
```

(1) I/O-bound web request

2. CPU-bound response processing

3. CPU-bound string concatenation

4. I/O-bound write to disk

I/O-bound and CPU-bound operations usually live side by side one another. We first make an I/O-bound request to download the contents of https:/ /www.example.com. Once we have the response, we perform a CPU-bound loop to format the headers of the response and turn them into a string separated by newlines. We then open a file and write the string to that file, both I/O-bound operations.

Asynchronous I/O allows us to pause execution of a particular method when we have an I/O operation; we can run other code while waiting for our initial I/O to complete in the background. This allows us to execute many I/O operations concurrently, potentially speeding up our application.

derstanding concurrency, parallelism, and multitasking

To better understand how concurrency can help our applications perform better, it is first important to learn and fully understand the terminology of concurrent programming. We'll learn more about what concurrency means and how asyncio uses a concept called multitasking to achieve it. Concurrency and parallelism are two concepts that help us understand how programming schedules and carries out various tasks, methods, and routines that drive action.

Concurrency

When we say two tasks are happening *concurrently*, we mean those tasks are happening at the same time. Take, for instance, a baker baking two different cakes. To

bake these cakes, we need to preheat our oven. Preheating can take tens of minutes depending on the oven and the baking temperature, but we don't need to wait for our oven to preheat before starting other tasks, such as mixing the flour and sugar together with eggs. We can do other work until the oven beeps, letting us know it is preheated.

We also don't need to limit ourselves from starting work on the second cake before finishing the first. We can start one cake batter, put it in a stand mixer, and start preparing the second batter while the first batter finishes mixing. In this model, we're switching between different tasks concurrently. This switching between tasks (doing something else while the oven heats, switching between two different cakes) is *concurrent* behavior.

Parallelism

While concurrency implies that multiple tasks are in process simultaneously, it does not imply that they are running together in parallel. When we say something is running *in parallel*, we mean not only are there two or more tasks happening concurrently, but they are also executing at the same time. Going back to our cake baking example, imagine we have the help of a second baker. In this scenario, we can work on the first cake while the second baker works on the second. Two people making batter at once is parallel because we have two distinct tasks running concurrently (figure 1.1).

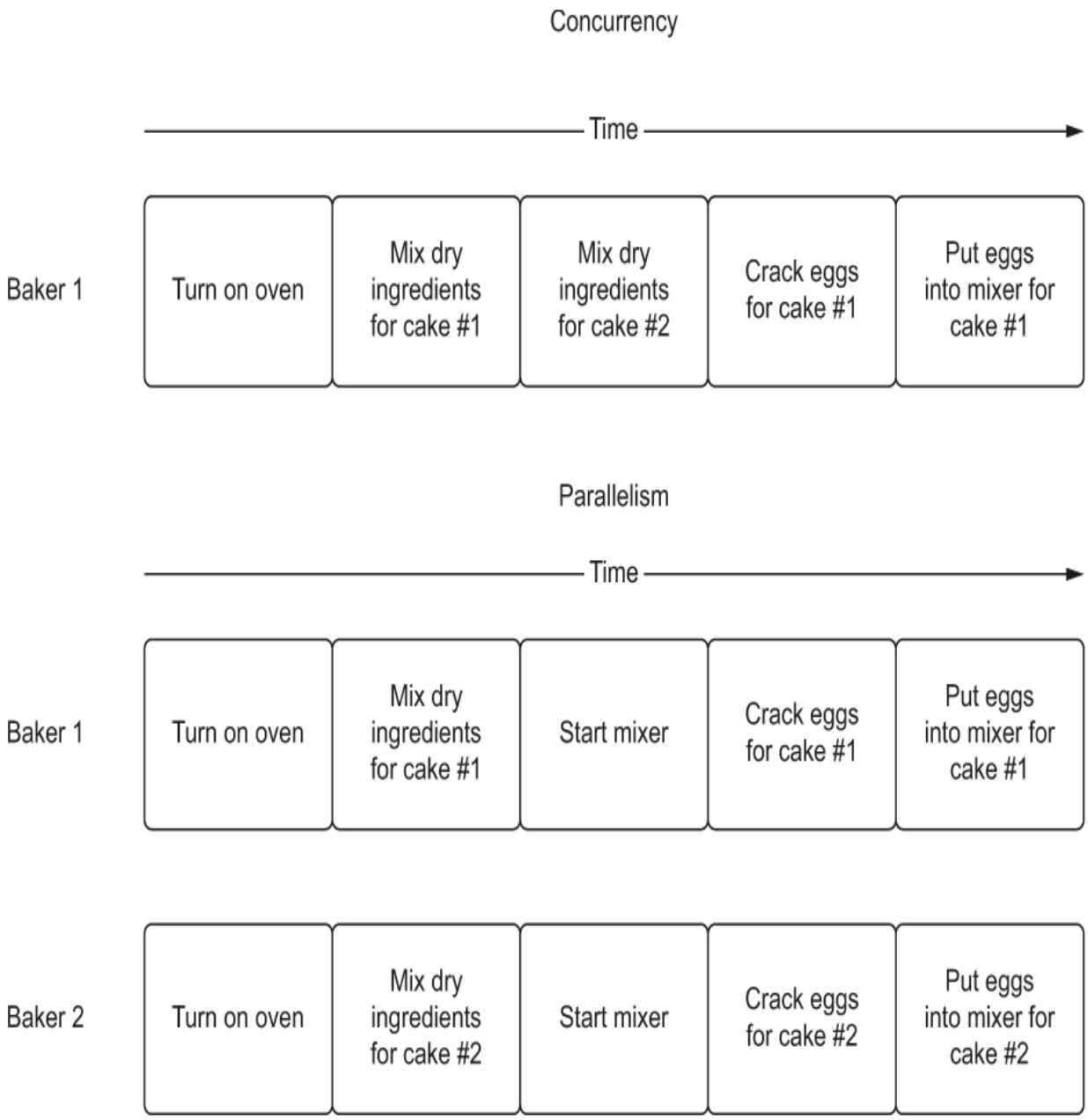

Figure 1.1 With *concurrency*, we have multiple tasks happening at the same time, but only one we're actively doing at a given point in time. With *parallelism*, we have multiple tasks happening and are actively doing more than one simultaneously.

Putting this into terms of applications run by our operating system, let's imagine it has two applications running. In a system that is only concurrent, we can switch

between running these applications, running one application for a short while before letting the other one run. If we do this fast enough, it gives the appearance of two things happening at once. In a system that is parallel, two applications are running simultaneously, and we're actively running two things concurrently.

The concepts of concurrency and parallelism are similar (figure 1.2) and slightly confusing to differentiate, but it is important to understand what makes them distinct from one another.

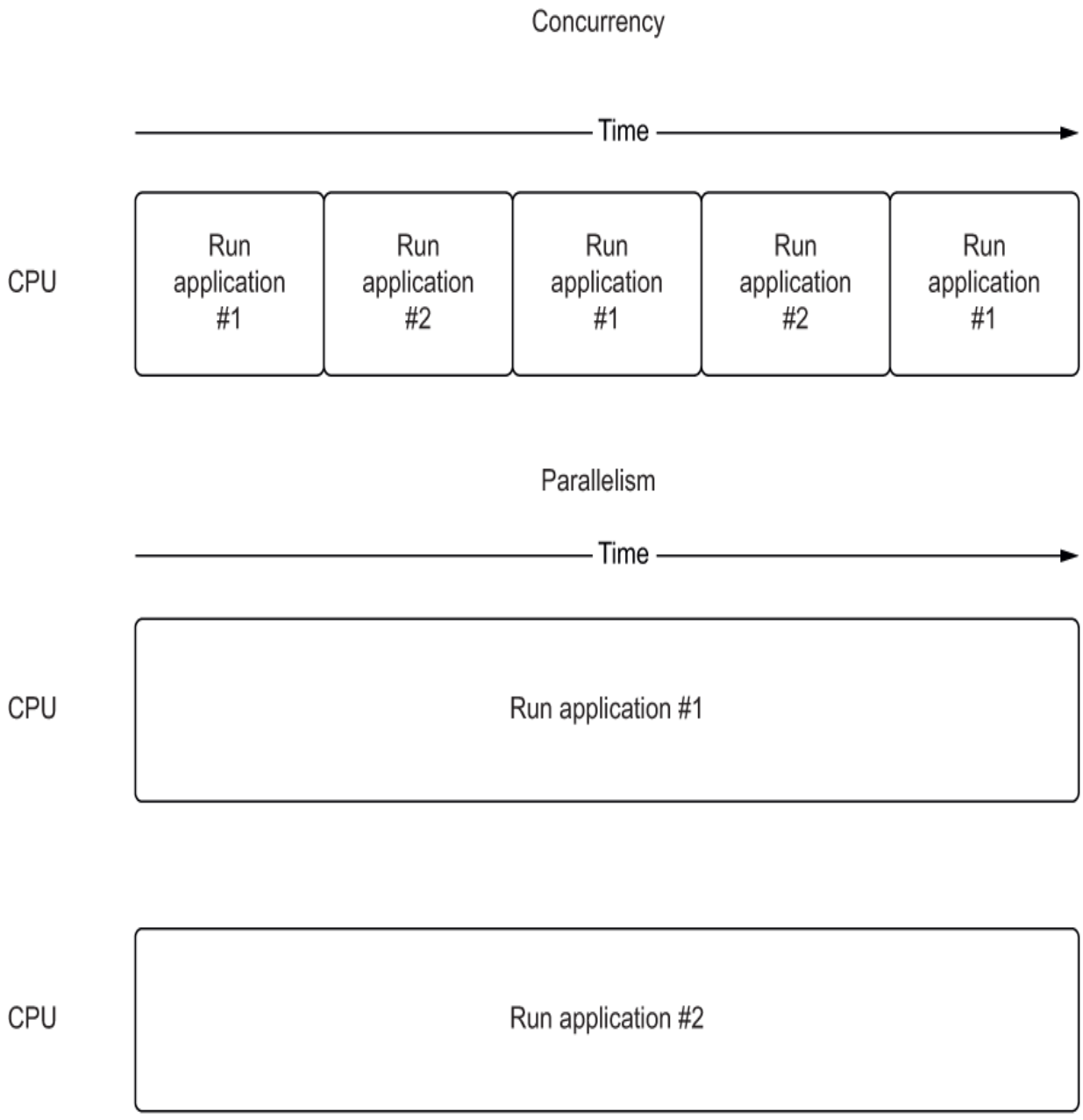

Figure 1.2 With concurrency, we switch between running two applications. With parallelism, we actively run two applications simultaneously.

The difference between concurrency and parallelism

Concurrency is about multiple tasks that can happen independently from one another. We can have concurrency on a CPU with only one core, as the operation will employ

preemptive multitasking (defined in the next section) to switch between tasks. Parallelism, however, means that we must be executing two or more tasks at the same time. On a machine with one core, this is not possible. To make this possible, we need a CPU with multiple cores that can run two tasks together.

While parallelism implies concurrency, concurrency does not always imply parallelism. A multithreaded application running on a multiple-core machine is both concurrent and parallel. In this setup, we have multiple tasks running at the same time, and there are two cores independently executing the code associated with those tasks. However, with multitasking we can have multiple tasks happening concurrently, yet only one of them is executing at a given time.

What is multitasking?

Multitasking is everywhere in today's world. We multitask while making breakfast by taking a call or answering a text while we wait for water to boil to make tea. We even multitask while commuting to work, by reading a book while the train takes us to our stop. Two main kinds of multitasking are discussed in this section: *preemptive multitasking* and *cooperative multitasking*.

PREEMPTIVE MULTITASKING

In this model, we let the operating system decide how to switch between which work is currently being executed via a process called *time slicing*. When the operating system switches between work, we call it *preempting*.

How this mechanism works under the hood is up to the operating system itself. It is primarily achieved through using either multiple threads or multiple processes.

COOPERATIVE MULTITASKING

In this model, instead of relying on the operating system to decide when to switch between which work is currently being executed, we explicitly code points in our application where we can let other tasks run. The tasks in our application operate in a model where they *cooperate*, explicitly saying, “I’m pausing my task for a while; go ahead and run other tasks.”

The benefits of cooperative multitasking

asyncio uses cooperative multitasking to achieve concurrency. When our application reaches a point where it could wait a while for a result to come back, we explicitly mark this in code. This allows other code to run while we wait for the result to come back in the background. Once the task we marked has completed, we in effect “wake up” and resume executing the task. This gives us a form of concurrency because we can have multiple tasks started at the same time but, importantly, not in parallel because they aren’t executing code simultaneously.

Cooperative multitasking has benefits over preemptive multitasking. First, cooperative multitasking is less resource intensive. When an operating system needs to switch between running a thread or process, it involves a *context switch*. Context switches are intensive operations because the operating system must save information about the running process or thread to be able to reload it.

A second benefit is *granularity*. An operating system knows that a thread or task should be paused based on whichever scheduling algorithm it uses, but that might not be the best time to pause. With cooperative multitasking, we explicitly mark the areas that are the best for pausing our tasks. This gives us some efficiency gains in that we are only switching tasks when we explicitly know it is the right time to do so. Now

that we understand concurrency, parallelism, and multitasking, we'll use these concepts to understand how to implement them in Python with threads and processes.

derstanding processes, threads, multithreading, and multiprocessing

To better set us up to understand how concurrency works in the Python world, we'll first need to understand the basics about how threads and processes work. We'll then examine how to use them for multithreading and multiprocessing to do work concurrently. Let's start with some definitions around processes and threads.

rocess

A *process* is an application run that has a memory space that other applications cannot access. An example of creating a Python process would be running a simple "hello world" application or typing `python` at the command line to start up the REPL (read eval print loop).

Multiple processes can run on a single machine. If we are on a machine that has a CPU with multiple cores, we can execute multiple processes at the same time. If we are on a CPU with only one core, we can still have multiple applications running simultaneously, through time slicing. When an operating system uses time slicing, it will switch between which process is running automatically after some amount of time. The algorithms that determine when this switching occurs are different, depending on the operating system.

hread

Threads can be thought of as lighter-weight processes. In addition, they are the smallest construct that can be managed by an operating system. They do not have their own memory as does a process; instead, they share the memory of the process that created them. Threads are associated with the process that created them. A process will always have at least one thread associated with it, usually known as the *main thread*. A process can also create other threads, which are more commonly known as *worker* or *background* threads. These threads can perform other work concurrently alongside the main thread. Threads, much like processes, can run alongside one another on a multi-core CPU, and the operating system can also switch between them via time slicing. When we run a normal Python application, we create a process as well as a main thread that will be responsible for running our Python application.

Listing 1.2 Processes and threads in a simple Python application

```
import os
import threading

print(f'Python process running with process id: {os.getpid()
total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'Python is currently running {total_threads} thread(s
print(f'The current thread is {thread_name}')
```

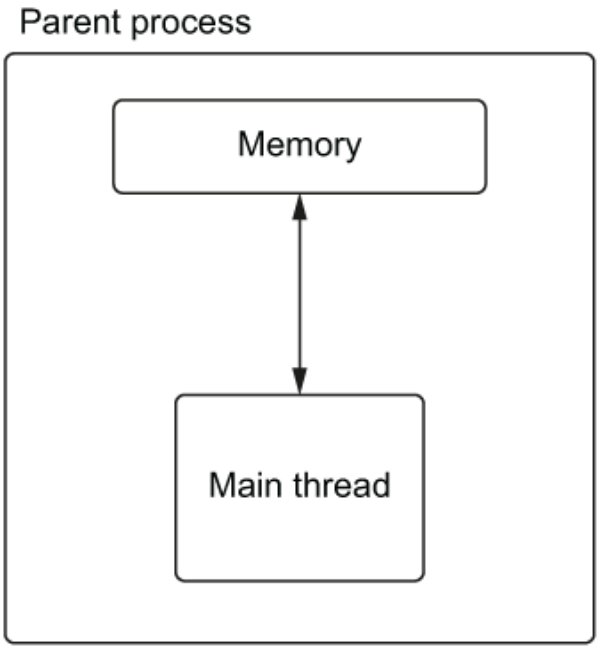

Figure 1.3 A process with one main thread reading from memory

In figure 1.3, we sketch out the process for listing 1.2. We create a simple application to show us the basics of the main thread. We first grab the process ID (a unique identifier for a process) and print it to prove that we indeed have a dedicated process running. We then get the active count of threads running as well as the current thread's name to show that we are running one thread—the main thread. While the process ID will be different each time this code is run, running listing 1.2 will give output similar to the following:

```
Python process running with process id: 98230
Python currently running 1 thread(s)
The current thread is MainThread
```

Processes can also create other threads that share the memory of the main process. These threads can do other work concurrently for us via what is known as *multithreading*.

Listing 1.3 Creating a multithreaded Python application

```
import threading


def hello_from_thread():
    print(f'Hello from thread {threading.current_thread()}!'


hello_thread = threading.Thread(target=hello_from_thread)
hello_thread.start()

total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'Python is currently running {total_threads} thread(s
print(f'The current thread is {thread_name}')

hello_thread.join()
```

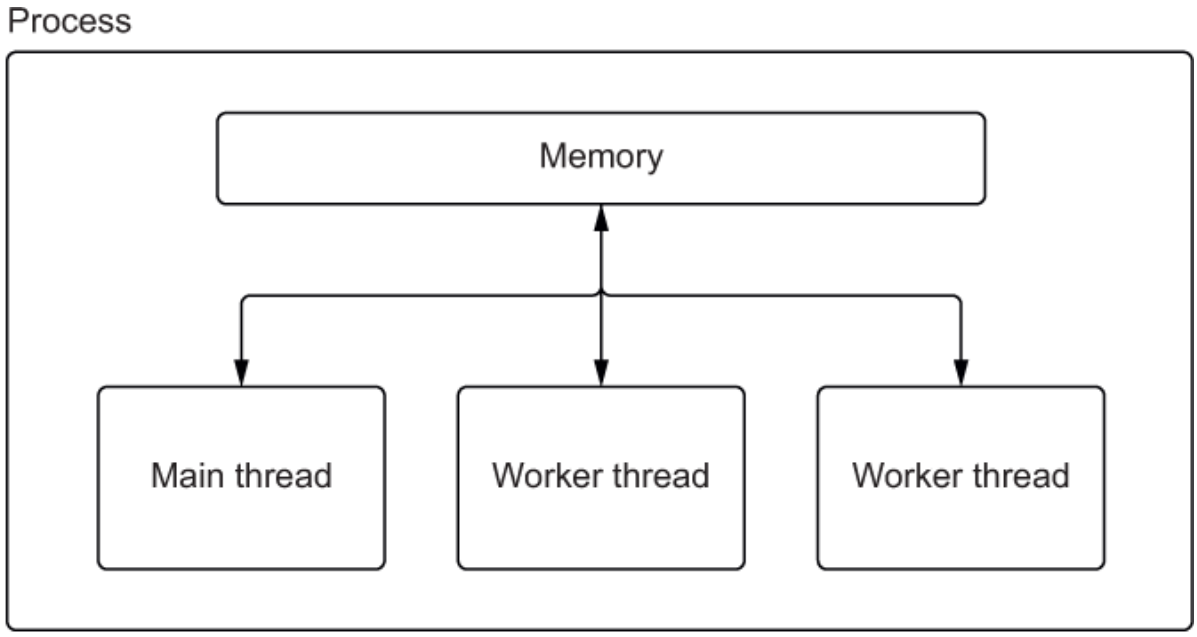

Figure 1.4 A multithreaded program with two worker threads and one main thread, each sharing the process's memory

In figure 1.4, we sketch out the process and threads for listing 1.3. We create a method to print out the name of the current thread and then create a thread to run that method. We then call the `start` method of the thread to start running it. Finally, we call the `join` method. `join` will cause the program to pause until the thread we started completed. If we run the previous code, we'll see output similar to the following:

```
Hello from thread <Thread(Thread-1, started 123145541312512)
Python is currently running 2 thread(s)
The current thread is MainThread
```

Note that when running this you may see the *hello from thread* and *python is currently running 2 thread(s)* messages print on the same line. This is a *race*

condition; we'll explore a bit about this in the next section and in chapters 6 and 7.

Multithreaded applications are a common way to achieve concurrency in many programming languages. There are a few challenges in utilizing concurrency with threads in Python, however. Multithreading is only useful for I/O-bound work because we are limited by the global interpreter lock, which is discussed in section 1.5.

Multithreading is not the only way we can achieve concurrency; we can also create multiple processes to do work concurrently for us. This is known as *multiprocessing*. In multiprocessing, a parent process creates one or more child processes that it manages. It can then distribute work to the child processes.

Python gives us the multiprocessing module to handle this. The API is similar to that of the threading module. We first create a process with a `target` function. Then, we call its `start` method to execute it and finally its `join` method to wait for it to complete running.

Listing 1.4 Creating multiple processes

```
import multiprocessing
import os


def hello_from_process():
    print(f'Hello from child process {os.getpid()}!')
if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_fro
    hello_process.start()
```

```
    print(f'Hello from parent process {os.getpid()}')

    hello_process.join()
```

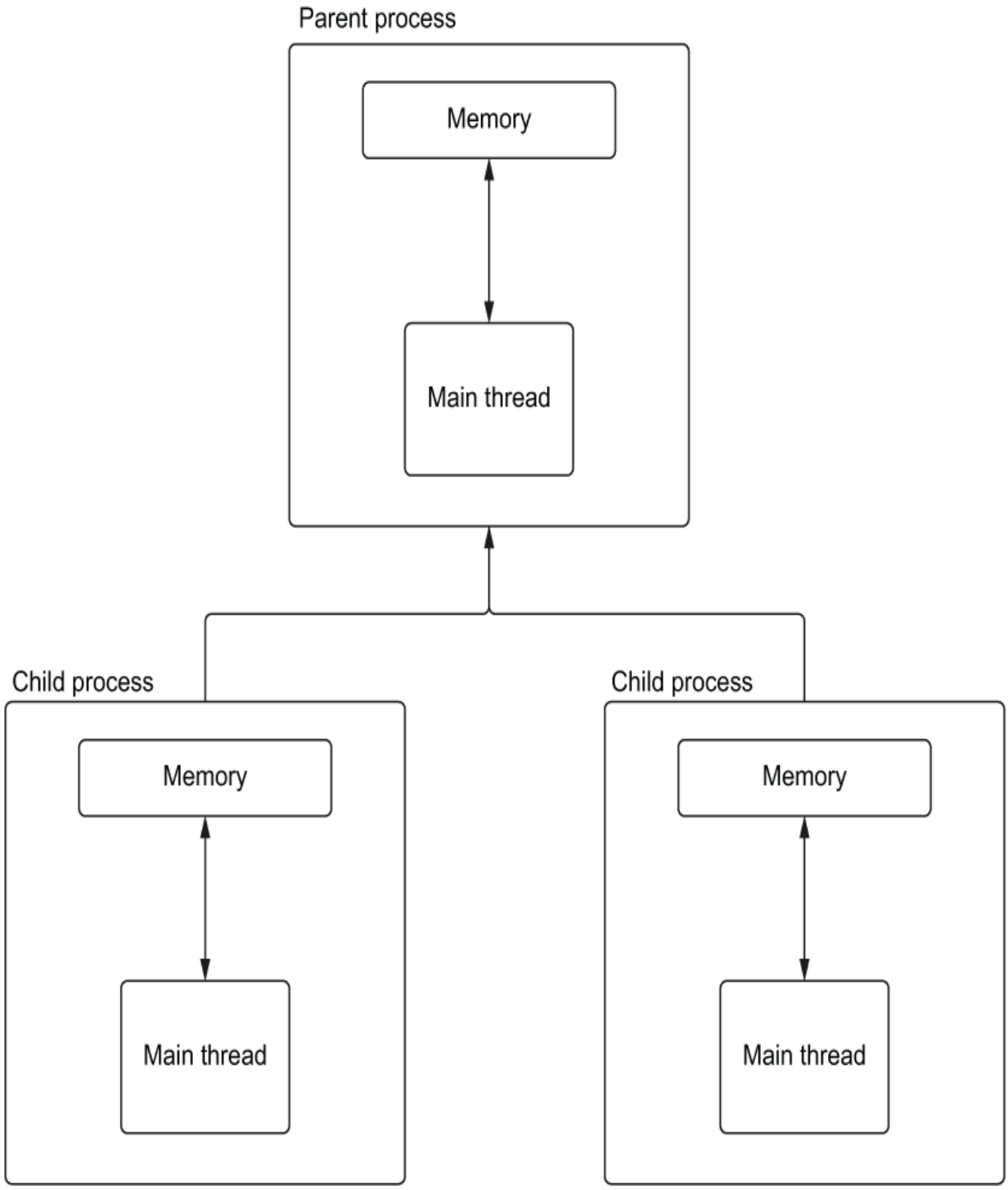

Figure 1.5 An application utilizing multiprocessing with one parent process and two child processes