Running this on the instance we described, this code completes in roughly 18 seconds, delivering a significant speedup compared with our serial version. This is quite a nice performance gain for not a whole lot more code! You may also wish to experiment with a machine with more CPU cores to see if you can further improve the performance of this algorithm.

You may notice in this implementation that we still have some CPU-bound work happening in our parent process that is parallelizable. Our reduce operation takes thousands of dictionaries and combines them together. We can apply the partitioning logic we used on the original data set and split these dictionaries into chunks and combine them across multiple processes. Let's write a new `reduce` function that does that. In this function, we'll partition the list and call `reduce` on each chunk in a worker process. Once this completes, we'll keep partitioning and reducing until we have one dictionary remaining. (In this listing, we've removed the partition, map, and merge functions for brevity.)

Listing 6.9 Parallelizing the `reduce` operation

```
import asyncio
import concurrent.futures
import functools
import time
from typing import Dict, List
from chapter_06.listing_6_8 import partition, merge_dictiona
async def reduce(loop, pool, counters, chunk_size) -> Dict[s
    chunks: List[List[Dict]] = list(partition(counters, chun
    reducers = []
    while len(chunks[0]) > 1:
        for chunk in chunks:
            reducer = functools.partial(functools.reduce,
```

```
                merge_dictionaries, chunk)
            reducers.append(loop.run_in_executor(pool, reduc
        reducer_chunks = await asyncio.gather(*reducers)
        chunks = list(partition(reducer_chunks, chunk_size))
        reducers.clear()
    return chunks[0][0]


async def main(partition_size: int):
    with open('googlebooks-eng-all-1gram-20120701-a', encodi
        contents = f.readlines()
        loop = asyncio.get_running_loop()
        tasks = []
        with concurrent.futures.ProcessPoolExecutor() as poo
            start = time.time()

            for chunk in partition(contents, partition_size)
                tasks.append(loop.run_in_executor(pool, func

            intermediate_results = await asyncio.gather(*tas
            final_result = await reduce(loop, pool, intermed

            print(f"Aardvark has appeared {final_result['Aar

            end = time.time()
            print(f'MapReduce took: {(end - start):.4f} seco


if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))
```

1. Partition the dictionaries into parallelizable chunks.

2. Reduce each partition into a single dictionary.

3. Wait for all reduce operations to complete.

4. Partition the results again, and start a new iteration of the loop.

If we run this parallelized `reduce`, we may see some small performance gain or only a small gain, depending on the machine you run on. In this instance, the overhead of pickling the intermediate dictionaries and sending them to the children processes will eat away much of the time savings by running in parallel. This optimization may not do much to make this problem less troublesome; however, if our `reduce` operation was more CPU-intensive, or we had a much larger data set, this approach can yield benefits.

Our multiprocessing approach has clear performance benefits over a synchronous approach, but right now there isn't an easy way to see how many map operations we've completed at any given time. In the synchronous version, we would only need to add a counter we incremented for every line we processed to see how far along we were. Since multiple processes by default do not share any memory, how can we create a counter to track our job's progress?

ared data and locks

In chapter 1, we discussed the fact that, in multiprocessing, each process has its own memory, separate from other processes. This presents a challenge when we have shared state to keep track of. So how can we share data between processes if their memory spaces are all distinct?

Multiprocessing supports a concept called *shared memory objects*. A shared memory object is a chunk of memory allocated that a set of separate processes can access. Each process, as shown in figure 6.2, can then read and write into that memory space as needed.

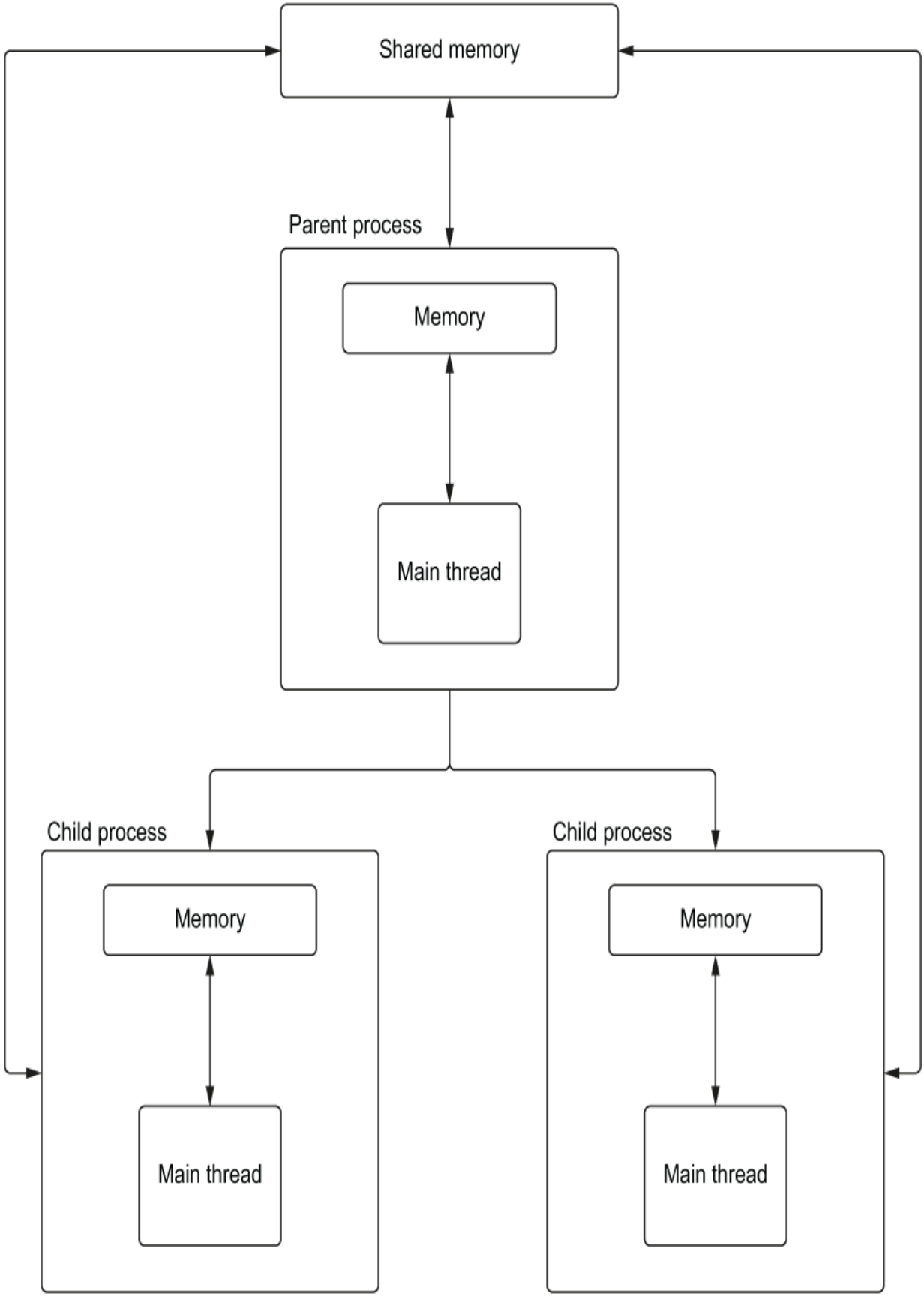
Shared memory
Parent process
Memory
Main thread
Child process
Memory
Main thread
Child process
Memory
Main thread

Figure 6.2 A parent process with two children processes, all sharing memory

Shared state is complicated and can lead to hard-to-reproduce bugs if not properly implemented. Generally, it is best to avoid shared state if possible. That said, sometimes it is necessary to introduce shared state. One such instance is a shared counter.

To learn more about shared data, we'll take our MapReduce example from above and keep a counter of how many map operations we've completed. We'll then periodically output this number to show how far along we are to the user.

haring data and race conditions

Multiprocessing supports two kinds of shared data: values and array. A *value* is a singular value, such as an integer or floating-point number. An *array* is an array of singular values. The types of data that we can share in memory are limited by the types defined in the Python array module, available at https://docs.python.org/3/library/array .html#module-array.

To create a value or array, we first need to use the typecode from the array module that is just a character. Let's create two shared pieces of data—one integer value and one integer array. We'll then create two processes to increment each of these shared pieces of data in parallel.

Listing 6.10 Shared values and arrays

```
from multiprocessing import Process, Value, Array
```

```
def increment_value(shared_int: Value):
    shared_int.value = shared_int.value + 1


def increment_array(shared_array: Array):
    for index, integer in enumerate(shared_array):
        shared_array[index] = integer + 1


if __name__ == '__main__':
    integer = Value('i', 0)
    integer_array = Array('i', [0, 0])

    procs = [Process(target=increment_value, args=(integer,)
             Process(target=increment_array, args=(integer_a

    [p.start() for p in procs]
    [p.join() for p in procs]

    print(integer.value)
    print(integer_array[:])
```

In the preceding listing, we create two processes—one to increment our shared integer value and one to increment each element in our shared array. Once our two subprocesses complete, we print out the data.

Since our two pieces of data are never touched by different processes, this code works well. Will this code continue to work if we have multiple processes modifying the same shared data? Let’s test this out by creating two processes to increment a shared integer value in parallel. We’ll run this code repeatedly in a loop to see if we get

consistent results. Since we have two processes, each incrementing a shared counter by one, once the processes finish we expect the shared value to always be two.

Listing 6.11 Incrementing a shared counter in parallel

```
from multiprocessing import Process, Value


def increment_value(shared_int: Value):
    shared_int.value = shared_int.value + 1


if __name__ == '__main__':
    for _ in range(100):
        integer = Value('i', 0)
        procs = [Process(target=increment_value, args=(integ
                 Process(target=increment_value, args=(integ

        [p.start() for p in procs]
        [p.join() for p in procs]
        print(integer.value)
        assert(integer.value == 2)
```

While you will see different output because this problem is nondeterministic, at some point you should see that the result isn't always 2.

```
2
2
2
Traceback (most recent call last):
```

```
  File "listing_6_11.py", line 17, in <module>
    assert(integer.value == 2)
AssertionError
1
```

Sometimes our result is 1! Why is this? What we've encountered is called a *race condition*. A race condition occurs when the outcome of a set of operations is dependent on which operation finishes first. You can imagine the operations as racing against one another; if the operations win the race in the right order, everything works fine. If they win the race in the wrong order, bizarre behavior results.

So where is the race occurring in our example? The problem lies in that incrementing a value involves both read and write operations. To increment a value, we first need to read the value, add one to it, then write the result back to memory. The value each process sees in the shared data is entirely dependent on when it reads the shared value.

If the processes run in the following order, everything works fine, as seen in figure 6.3.

In this example, Process 1 increments the value just before Process 2 reads it and wins the race. Since Process 2 finishes second, this means that it will see the correct value of one and will add to it, producing the correct final value.

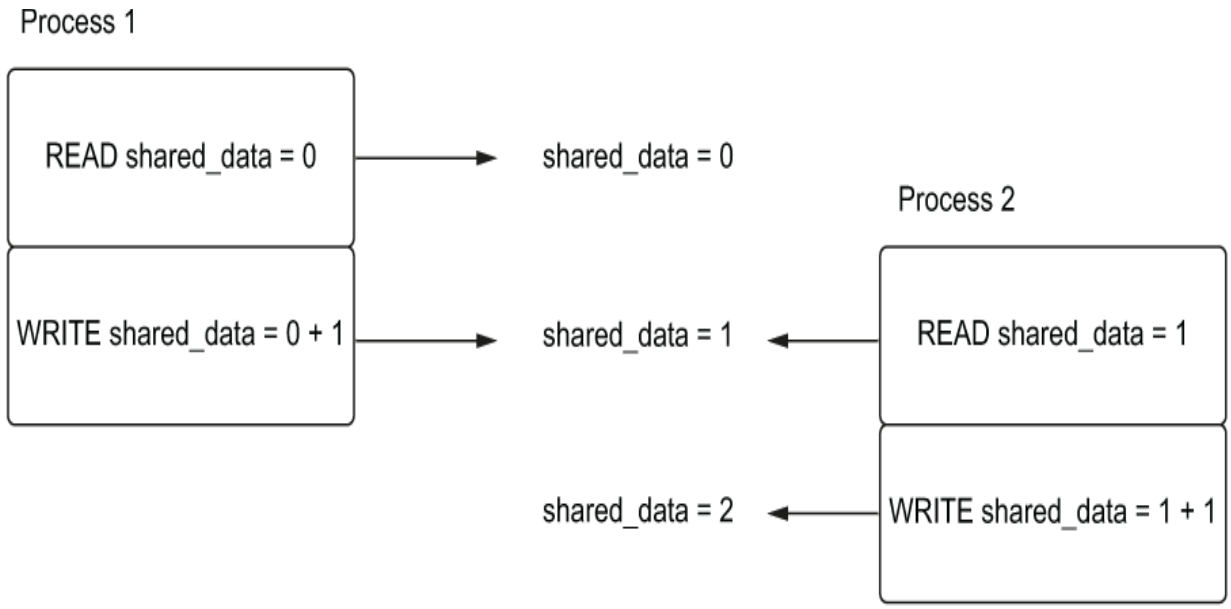

Figure 6.3 Successfully avoiding a race condition

What happens if there is a tie in our virtual race? Look at figure 6.4.

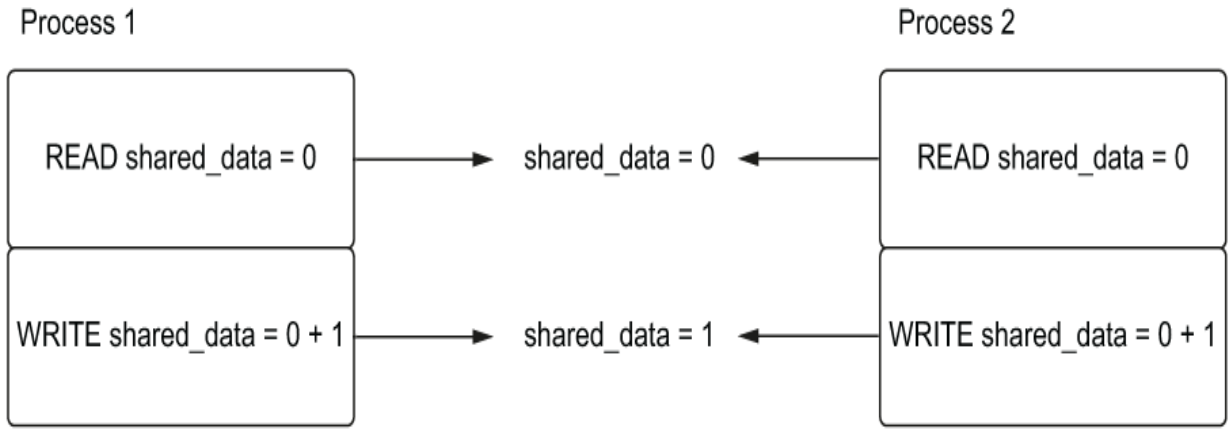

Figure 6.4 A race condition

In this instance, Processes 1 and 2 both read the initial value of zero. They then increment that value to 1 and write it back at the same time, producing the incorrect value.

You may ask, “But our code is only one line. Why are there two operations!?” Under the hood, incrementing is written as two operations, which causes this issue. This makes it *non-atomic* or not *thread-safe*. This isn’t easy to figure out. An explanation of which operations are atomic and which operations are non-atomic is available at http://mng.bz/5Kj4.

These types of errors are tricky because they are often difficult to reproduce. They aren’t like normal bugs, since they depend on the order in which our operating system runs things, which is out of our control when we use multiprocessing. So how do we fix this nasty bug?

ynchronizing with locks

We can avoid race conditions by *synchronizing* access to any shared data we want to modify. What does it mean to synchronize access? Revisiting our race example, it means that we control access to any shared data so that any operations we have finish the race in an order that makes sense. If we’re in a situation where a tie between two operations could occur, we explicitly block the second operation from running until the first completes, guaranteeing operations to finish the race in a consistent manner. You can imagine this as a referee at the finish line, seeing that a tie is about to happen and telling the runners, “Hold up a minute. One racer at a time!” and picking one runner to wait while the other crosses the finish line.

One mechanism for synchronizing access to shared data is a *lock*, also known as a *mutex* (short for *mutual exclusion*). These structures allow for a single process to “lock” a section of code, preventing other processes from running that code. The locked section of code is commonly called a *critical section*. This means that if one process is executing the code of a locked section and a second process tries to access

that code, the second process will need to wait (blocked by the referee) until the first process is finished with the locked section.

Locks support two primary operations: *acquiring* and *releasing*. When a process acquires a lock, it is guaranteed that it will be the only process running that section of code. Once the section of code that needs synchronized access is finished, we release the lock. This allows other processes to acquire the lock and run any code in the critical section. If a process tries to run code that is locked by another process, acquiring the lock will block until the other process releases that lock.

Revisiting our counter race condition example, and using figure 6.5, let's visualize what happens when two processes try and acquire a lock at roughly the same time. Then, let's see how it prevents the counter from getting the wrong value.

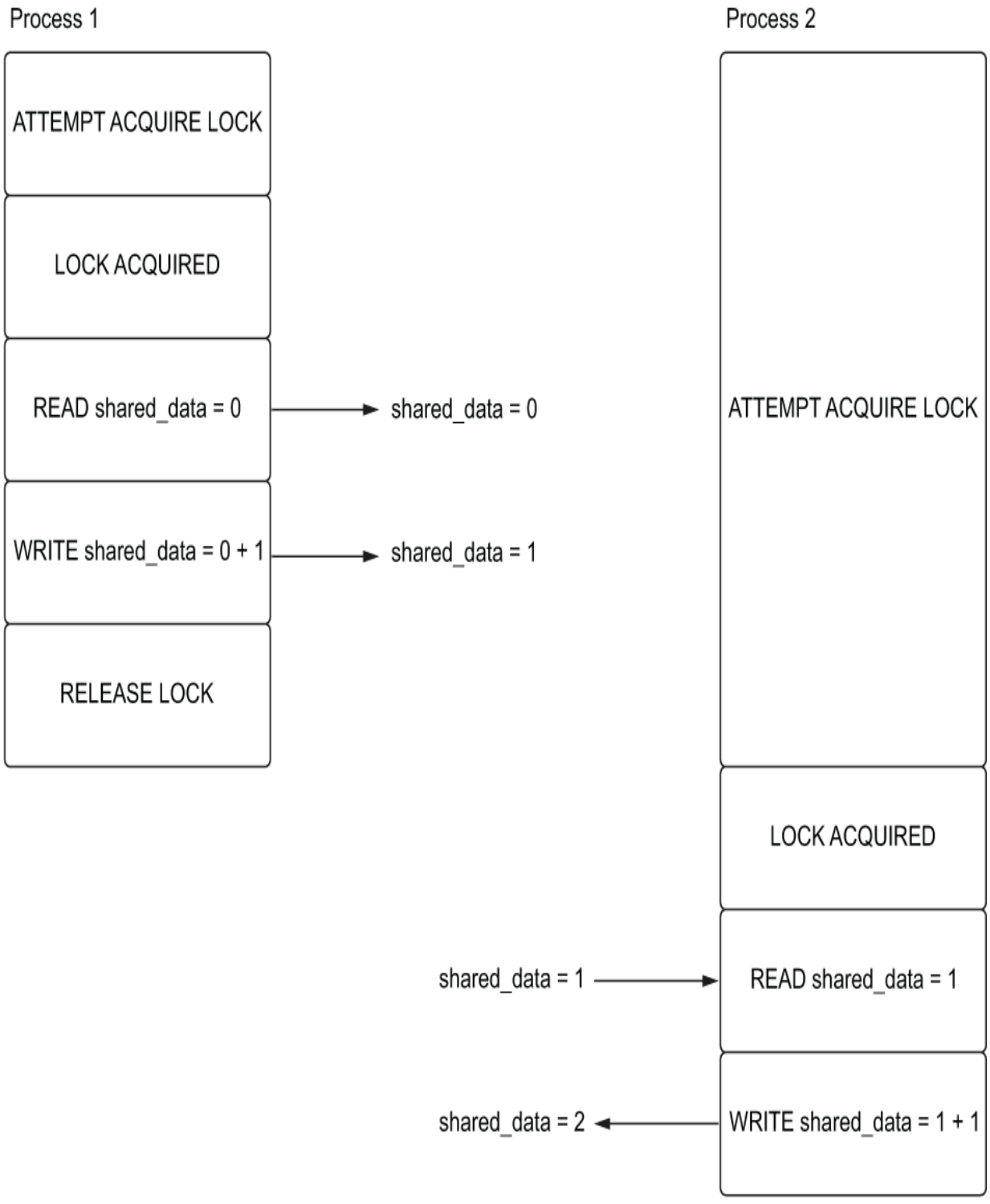

Figure 6.5 Process 2 is blocked from reading shared data until Process 1 releases the lock.

In this diagram, Process 1 first acquires the lock successfully and reads and increments the shared data. The second process tries to acquire the lock but is blocked from advancing further until the first process releases the lock. Once the first process releases the lock, the second process can successfully acquire the lock and increment the shared data. This prevents the race condition because the lock prevents more than one process from reading and writing the shared data at the same time.

So how do we implement this synchronization with our shared data? The multiprocessing API implementors thought of this and nicely included a method to get a lock on both value and array. To acquire a lock, we call `get_lock().acquire()` and to release a lock we call `get_lock().release()`. Using listing 6.12, let's apply this to our previous example to fix our bug.

Listing 6.12 Acquiring and releasing a lock

```
from multiprocessing import Process, Value


def increment_value(shared_int: Value):
    shared_int.get_lock().acquire()
    shared_int.value = shared_int.value + 1
    shared_int.get_lock().release()


if __name__ == '__main__':
    for _ in range(100):
        integer = Value('i', 0)
        procs = [Process(target=increment_value, args=(integ
                 Process(target=increment_value, args=(integ
```

```
        [p.start() for p in procs]
        [p.join() for p in procs]
        print(integer.value)
        assert (integer.value == 2)
```

When we run this code, every value we get should be `2`. We've fixed our race condition! Note that locks are also context managers, and to clean up our code we could have written `increment_value` using a `with` block. This will acquire and release the lock for us automatically:

```
def increment_value(shared_int: Value):
    with shared_int.get_lock():
        shared_int.value = shared_int.value + 1
```

Notice that we have taken concurrent code and have just forced it to be sequential, negating the value of running in parallel. This is an important observation and is a caveat of synchronization and shared data in concurrency in general. To avoid race conditions, we must make our parallel code sequential in critical sections. This can hurt the performance of our multiprocessing code. Care must be taken to lock only the sections that absolutely need it so that other parts of the application can execute concurrently. When faced with a race condition bug, it is easy to protect all your code with a lock. This will "fix" the problem but will likely degrade your application's performance.

Sharing data with process pools

We've just seen how to share data within a couple of processes, so how do we apply this knowledge to process pools? Process pools operate a bit differently than creating processes manually, posing a challenge with shared data. Why is this?

When we submit a task to a process pool, it may not run immediately because the processes in the pool may be busy with other tasks. How does the process pool handle this? In the background, process pool executors keep a queue of tasks to manage this. When we submit a task to the process pool, its arguments are pickled (serialized) and put on the task queue. Then, each worker process asks for a task from the queue when it is ready for work. When a worker process pulls a task off the queue, it unpickles (deserializes) the arguments and begins to execute the task.

Shared data is, by definition, shared among worker processes. Therefore, pickling and unpickling it to send it back and forth between processes makes little sense. In fact, neither `Value` nor `Array` objects can be pickled, so if we try to pass the shared data in as arguments to our functions as we did before, we'll get an error along the lines of `can't pickle Value objects`.

To handle this, we'll need to put our shared counter in a global variable and somehow let our worker processes know about it. We can do this with *process pool initializers*. These are special functions that are called when each process in our pool starts up. Using this, we can create a reference to the shared memory that our parent process created. We can pass this function in when we create a process pool. To see how this works, let's create a simple example that increments a counter.

Listing 6.13 Initializing a process pool

```
from concurrent.futures import ProcessPoolExecutor
import asyncio
from multiprocessing import Value

shared_counter: Value


def init(counter: Value):
    global shared_counter
    shared_counter = counter


def increment():
    with shared_counter.get_lock():
        shared_counter.value += 1


async def main():
    counter = Value('d', 0)
    with ProcessPoolExecutor(initializer=init,
                             initargs=(counter,)) as pool:
        await asyncio.get_running_loop().run_in_executor(poo
        print(counter.value)


if __name__ == "__main__":
    asyncio.run(main())
```

❶ This tells the pool to execute the function init with the argument counter for each process.

We first define a global variable, `shared_counter`, which will contain the reference to the shared `Value` object we create. In our `init` function, we take in a `Value` and initialize `shared_counter` to that value. Then, in the main coroutine, we create the counter and initialize it to 0, then pass in our `init` function and our counter to the `initializer` and `initargs` parameter when creating the process pool. The `init` function will be called for each process that the process pool creates, correctly initializing our `shared_ counter` to the one we created in our main coroutine.

You may ask, "Why do we need to bother with all this? Can't we just initialize the global variable as `shared_counter: Value = Value('d', 0)` instead of leaving it empty?" The reason we can't do this is that when each process is created, the script we created it from is run again, per each process. This means that each process that starts will execute `shared_counter: Value = Value('d', 0)`, meaning that if we have 100 processes, we'd get 100 `shared_counter` values, each set to 0, resulting in some strange behavior.

Now that we know how to initialize shared data properly with a process pool, let's see how to apply this to our MapReduce application. We'll create a shared counter that we'll increment each time a map operation completes. We'll also create a `progress reporter1` task that will run in the background and output our progress to the console every second. For this example, we'll import some of our code around partitioning and reducing, to avoid repeating ourselves.

Listing 6.14 Keeping track of map operation progress

```
from concurrent.futures import ProcessPoolExecutor
import functools
import asyncio
```

```
from multiprocessing import Value
from typing import List, Dict
from chapter_06.listing_6_8 import partition, merge_dictiona

map_progress: Value


def init(progress: Value):
    global map_progress
    map_progress = progress


def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, count, _ = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)

    with map_progress.get_lock():
        map_progress.value += 1

    return counter


async def progress_reporter(total_partitions: int):
    while map_progress.value < total_partitions:
        print(f'Finished {map_progress.value}/{total_partiti
        await asyncio.sleep(1)
```

```
async def main(partiton_size: int):
    global map_progress

    with open('googlebooks-eng-all-1gram-20120701-a', encodi
        contents = f.readlines()
        loop = asyncio.get_running_loop()
        tasks = []
        map_progress = Value('i', 0)

        with ProcessPoolExecutor(initializer=init,
                                 initargs=(map_progress,)) a
            total_partitions = len(contents) // partiton_siz
            reporter = asyncio.create_task(progress_reporter

            for chunk in partition(contents, partiton_size):
                tasks.append(loop.run_in_executor(pool, func

            counters = await asyncio.gather(*tasks)

            await reporter

            final_result = functools.reduce(merge_dictionari

            print(f"Aardvark has appeared {final_result['Aar


if __name__ == "__main__":
    asyncio.run(main(partiton_size=60000))
```

The main change from our original MapReduce implementation, aside from initializing a shared counter, is inside our `map_frequencies` function. Once we have finished counting all words in that chunk, we acquire the lock for the shared counter and increment it. We also added a `progress_reporter` coroutine, which will run in the background and report how many jobs we've completed every second. When running this, you should see output similar to the following:

```
Finished 17/1443 map operations
Finished 144/1443 map operations
Finished 281/1443 map operations
Finished 419/1443 map operations
Finished 560/1443 map operations
Finished 701/1443 map operations
Finished 839/1443 map operations
Finished 976/1443 map operations
Finished 1099/1443 map operations
Finished 1230/1443 map operations
Finished 1353/1443 map operations
Aardvark has appeared 15209 times.
```

We now know how to use multiprocessing with asyncio to improve the performance of CPU-intensive work. What happens if we have a workload that has work that has both heavily CPU-bound and I/O-bound operations? We can use multiprocessing, but is there a way for us to combine the ideas of multiprocessing and a single-threaded concurrency model to further improve performance?

ıltiple processes, multiple event loops

While multiprocessing is mainly useful for CPU-bound tasks, it can have benefits for workloads that are I/O-bound as well. Let's take our example of running multiple SQL queries concurrently from listing 5.8 in the previous chapter. Can we use multiprocessing to further improve its performance? Let's look at what its CPU usage graph looks like on a single core, as illustrated in figure 6.6.

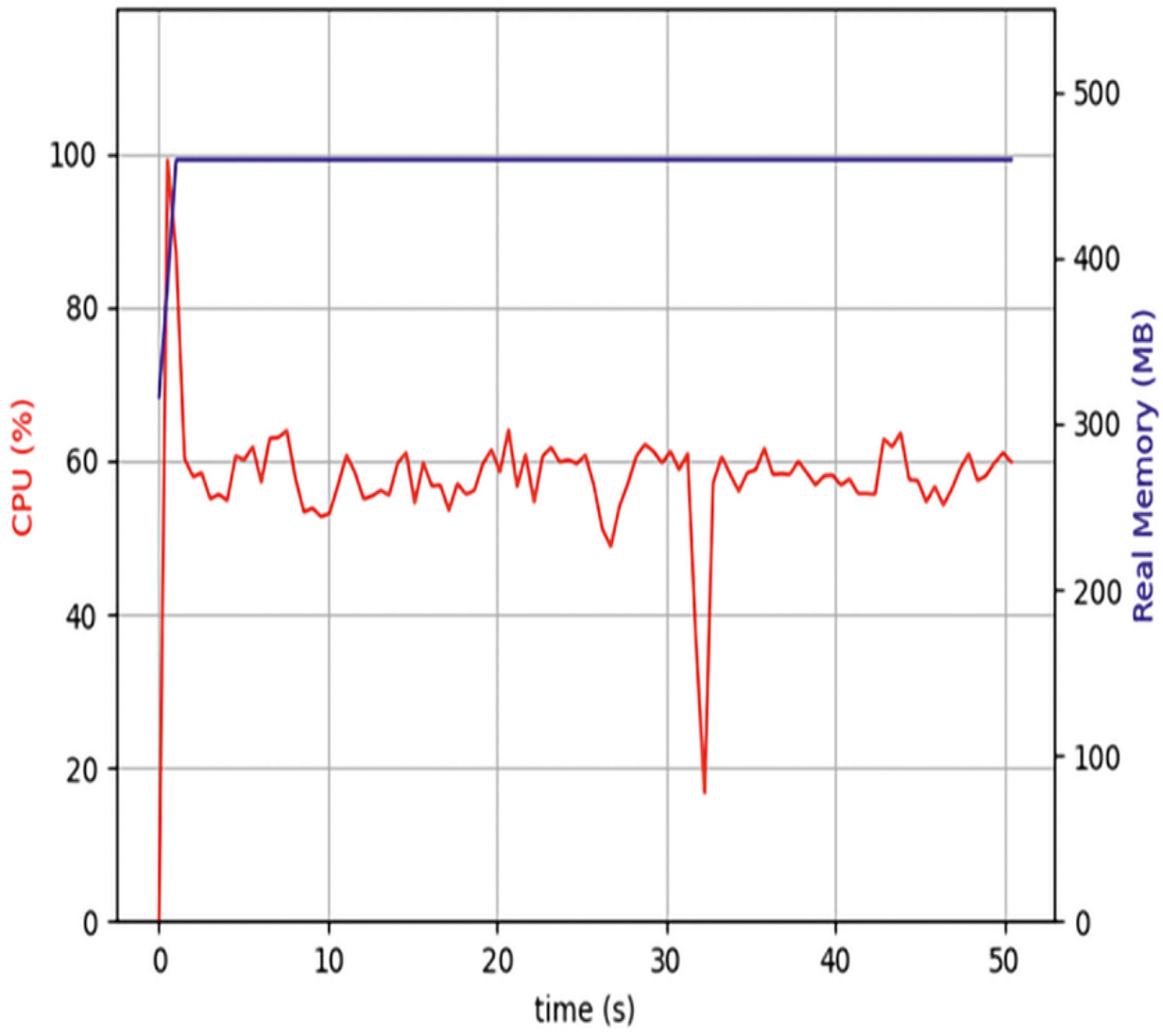

Figure 6.6 The CPU utilization graph for the code in listing 5.8

While this code is mostly making I/O-bound queries to our database, there is still a significant amount of CPU utilization happening. Why is this? In this instance, there is work happening to process the raw results we get from Postgres, leading to higher CPU utilization. Since we're single-threaded, while this CPU-bound work is happening, our event loop isn't processing results from other queries. This poses a potential throughput issue. If we issue 10,000 SQL queries concurrently, but we can only process one result at a time, we may end up with a backlog of query results to process.

Is there a way for us to improve our throughput by using multiprocessing? Using multiprocessing, each process has its own thread and its own Python interpreter. This opens up the opportunity to create one event loop per each process in our pool. With this model, we can distribute our queries over several processes. As seen in figure 6.7, this will spread the CPU load across multiple processes.

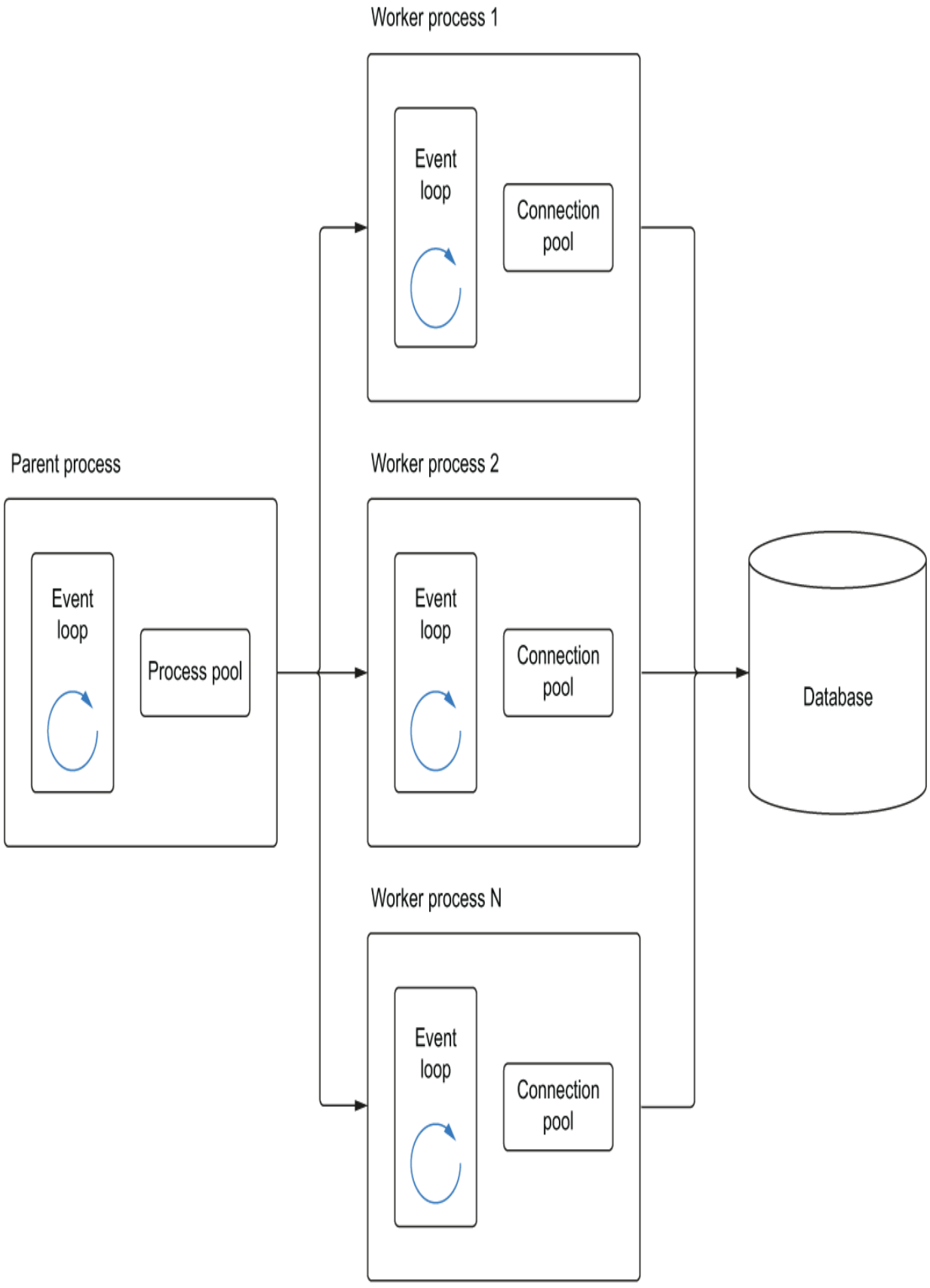
Worker process 1
Event loop
Connection pool
Parent process
Worker process 2
Event loop
Process pool
Event loop
Connection pool
Database
Worker process N
Event loop
Connection pool

Figure 6.7 A parent process creates a process pool. The parent process then creates workers, each with its own event loop.

While this won't make our I/O throughput increase, it will increase how many query results we can process at a time. This will increase the overall throughout of our application. Let's take our example from listing 5.7 and use it to create this architecture, as shown in listing 6.15.

Listing 6.15 One event loop per process

```
import asyncio
import asyncpg
from util import async_timed
from typing import List, Dict
from concurrent.futures.process import ProcessPoolExecutor


product_query = \
    """
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_
JOIN product_size as ps on ps.product_size_id = s.product_si
WHERE p.product_id = 100"""
```

```
async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)


@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)


def run_in_new_loop(num_queries: int) -> List[Dict]:
    async def run_queries():
        async with asyncpg.create_pool(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       password='password',
                                       database='products',
                                       min_size=6,
                                       max_size=6) as pool:
            return await query_products_concurrently(pool, n

    results = [dict(result) for result in asyncio.run(run_qu
    return results


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    pool = ProcessPoolExecutor()
    tasks = [loop.run_in_executor(pool, run_in_new_loop, 100
```

```
    all_results = await asyncio.gather(*tasks)
    total_queries = sum([len(result) for result in all_resul
    print(f'Retrieved {total_queries} products the product d


if __name__ == "__main__":
    asyncio.run(main())
```

❶ Run queries in a new event loop, and convert them to dictionaries.

❷ Create five processes each with their own event loop to run queries.

❸ Wait for all query results to complete.

We create a new function: `run_in_new_loop`. This function has an inner coroutine, `run_queries`, which creates a connection pool and runs the number of queries we specify concurrently. We then call `run_queries` with `asyncio.run`, which creates a new event loop and runs the coroutine.

One thing to note here is that we convert our results into dictionaries because asyncpg record objects cannot be pickled. Converting to a data structure that is serializable ensures that we can send our result back to our parent process.

In our main coroutine, we create a process pool and make five calls to `run_in_new_loop`. This will concurrently kick off 50,000 queries—10,000 per each of five processes. When you run this, you should see five processes launched quickly, followed by each of these processes finishing at roughly the same time. The runtime of the entire application should take slightly longer than the slowest process. When running this on an eight-core machine, this script was able to complete in roughly 13

seconds. Going back to our previous example from chapter 5, we made 10,000 queries in about 6 seconds. This output means we were getting a throughput of roughly 1,666 queries per second. With the multiprocessing and multiple event loop approach, we completed 50,000 queries in 13 seconds, or roughly 3,800 queries per second, more than doubling our throughput.

ary

- We've learned how to run multiple Python functions in parallel with a process pool.
- We've learned how to create a process pool executor and run Python functions in parallel. A process pool executor lets us use asyncio API methods such as `gather` to run multiple processes concurrently and wait for the results.
- We've learned how to solve a problem with MapReduce using process pools and asyncio. This workflow not only applies to MapReduce but can be used in general with any CPU-bound work that we can split into multiple smaller chunks.
- We've learned how to share state between multiple processes. This lets us keep track of data that is relevant for subprocesses we kick off, such as a status counter.
- We've learned how to avoid race conditions by using locks. Race conditions happen when multiple processes attempt to access data at roughly the same time and can lead to hard-to reproduce bugs.
- We've learned how to use multiprocessing to extend the power of asyncio by creating an event loop per each process. This has the potential to improve performance of workloads that have a mixture of CPU-bound and I/O-bound work.

7 Handling blockingwork with threads

This chapter covers

- Reviewing the multithreading library
- Creating thread pools to handle blocking I/O
- Using `async` and `await` to manage threads
- Handling blocking I/O libraries with thread pools
- Handling shared data and locking with threads
- Handling CPU-bound work in threads

When developing a new I/O-bound application from scratch, asyncio may be a natural technology choice. From the beginning, you'll be able to use non-blocking libraries that work with asyncio, such as asyncpg and aiohttp, as you begin development. However, greenfields (a project lacking constraints imposed by prior work) development is a luxury that many software developers don't have. A large portion of our work may be managing existing code using blocking I/O libraries, such as requests for HTTP requests, psycopg for Postgres databases, or any number of blocking libraries. We may also be in a situation where an asyncio-friendly library does not yet exist. Is there a way to get the performance gains of concurrency while still using asyncio APIs in these cases?

Multithreading is the solution to this question. Since blocking I/O releases the global interpreter lock, this enables the possibility to run I/O concurrently in separate threads. Much like the multiprocessing library, asyncio exposes a way for us to utilize pools of threads, so we can get the benefits of threading while still using the asyncio APIs, such as `gather` and `wait`.

In this chapter, we'll learn how to use multithreading with asyncio to run blocking APIs, such as requests, in threads. In addition, we'll learn how to synchronize shared data like we did in the last chapter and examine more advanced locking topics such as *reentrant locks* and *deadlocks*. We'll also see how to combine asyncio with synchronous code by building a responsive GUI to run a HTTP stress test. Finally, we'll look at the few exceptions for which threading can be used for CPU-bound work.

roducing the threading module

Python lets developers create and manage threads via the threading module. This module exposes the `Thread` class, which, when instantiated, accepts a function to run in a separate thread. The Python interpreter runs single-threaded within a process, meaning that only one piece of Python bytecode can be running at one time even if we have code running in multiple threads. The global interpreter lock will only allow one thread to execute code at a time.

This seems like Python limits us from using multithreading to any advantage, but there are a few cases in which the global interpreter lock is released, the primary one being during I/O operations. Python can release the GIL in this case because, under the hood, Python is making low-level operating system calls to perform I/O. These system calls are outside the Python interpreter, meaning that no Python bytecode needs to run while we're waiting for I/O to finish.

To get a better sense of how to create and run threads in the context of blocking I/O, we'll revisit our example of an echo server from chapter 3. Recall that to handle multiple connections, we needed to switch our sockets to non-blocking mode and use the `select` module to watch for events on the sockets. What if we were working

with a legacy codebase where non-blocking sockets weren't an option? Could we still build an echo server that can handle more than one client at a time?

Since a socket's `recv` and `sendall` are I/O-bound methods, and therefore release the GIL, we should be able to run them in separate threads concurrently. This means that we can create one thread per each connected client and read and write data in that thread. This model is a common paradigm in web servers such as Apache and is known as a *thread-per-connection* model. Let's give this idea a try by waiting for connections in our main thread and then creating a thread to echo for each client that connects.

Listing 7.1 A multithreaded echo server

```
from threading import Thread
import socket


def echo(client: socket):
    while True:
        data = client.recv(2048)
        print(f'Received {data}, sending!')
        client.sendall(data)


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as se
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR
    server.bind(('127.0.0.1', 8000))
    server.listen()
    while True:
        connection, _ = server.accept()
```

```
            thread = Thread(target=echo, args=(connection,))
            thread.start()
```

❶ Block waiting for a client to connect.

❷ Once a client connects, create a thread to run our echo function.

❸ Start running the thread.

In the preceding listing, we enter an infinite loop listening for connections on our server socket. Once we have a client connected, we create a new thread to run our `echo` function. We supply the thread with a `target` that is the `echo` function we want to run and `args`, which is a tuple of arguments passed to `echo`. This means that we'll call `echo(connection)` in our thread. Then, we start the thread and loop again, waiting for a second connection. Meanwhile, in the thread we created, we loop forever listening for data from our client, and when we have it, we echo it back.

You should be able to connect an arbitrary amount of telnet clients concurrently and have messages echo properly. Since each `recv` and `sendall` operates in a separate thread per client, these operations never block each other; they only block the thread they are running in.

This solves the problem of multiple clients being unable to connect at the same time with blocking sockets, although the approach has some issues unique to threads. What happens if we try to kill this process with CTRL-C while we have clients connected? Does our application shut down the threads we created cleanly?

It turns out that things don't shut down quite so cleanly. If you kill the application, you should see a `KeyboardInterrupt` exception thrown on `server.accept()`,

but your application will hang as the background thread will keep the program alive. Furthermore, any connected clients will still be able to send and receive messages!

Unfortunately, user-created threads in Python do not receive `KeyboardInterrupt` exceptions; only the main thread will receive them. This means that our threads will keep running, happily reading from our clients and preventing our application from exiting.

There are a couple approaches to handle this; specifically, we can use what are called *daemon* threads (pronounced *demon*), or we can come up with our own way of canceling or "interrupting" a running thread. Daemon threads are a special kind of thread for long-running background tasks. These threads won't prevent an application from shutting down. In fact, when only daemon threads are running, the application will shut down automatically. Since Python's main thread is not a daemon thread, this means that, if we make all our connection threads daemonic, our application will terminate on a `KeyboardInterrupt`. Adapting our code from listing 7.1 to use daemonic threads is easy; all we need to do is set `thread.daemon = True` before we run `thread.start()`. Once we make that change, our application will terminate properly on CTRL-C.

The problem with this approach is we have no way to run any cleanup or shutdown logic when our threads stop, since daemon threads terminate abruptly. Let's say that on shutdown we want to write out to each client that the server is shutting down. Is there a way we can have some type of exception interrupt our thread and cleanly shut down the socket? If we call a socket's `shutdown` method, any existing calls to `recv` will return `zero`, and `sendall` will throw an exception. If we call `shutdown` from the main thread, this will have the effect of interrupting our client

threads that are blocking a `recv` or `sendall` call. We can then handle the exception in the client thread and perform any cleanup logic we'd like.

To do this, we'll create threads slightly differently than before, by subclassing the `Thread` class itself. This will let us define our own thread with a `cancel` method, inside of which we can shut down the client socket. Then, our calls to `recv` and `sendall` will be interrupted, allowing us to exit our `while` loop and close out the thread.

The `Thread` class has a `run` method that we can override. When we subclass `Thread`, we implement this method with the code that we want the thread to run when we start it. In our case, this is the `recv` and `sendall` echo loop.

Listing 7.2 Subclassing the thread class for a clean shutdown

```
from threading import Thread
import socket


class ClientEchoThread(Thread):

    def __init__(self, client):
        super().__init__()
        self.client = client

    def run(self):
        try:
            while True:
                data = self.client.recv(2048)
                if not data:
```

```
                    raise BrokenPipeError('Connection closed
                print(f'Received {data}, sending!')
                self.client.sendall(data)
        except OSError as e:
            print(f'Thread interrupted by {e} exception, shu

    def close(self):
        if self.is_alive():
            self.client.sendall(bytes('Shutting down!', enco
            self.client.shutdown(socket.SHUT_RDWR)


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as se
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR
    server.bind(('127.0.0.1', 8000))
    server.listen()
    connection_threads = []
    try:
        while True:
            connection, addr = server.accept()
            thread = ClientEchoThread(connection)
            connection_threads.append(thread)
            thread.start()
    except KeyboardInterrupt:
        print('Shutting down!')
        [thread.close() for thread in connection_threads]
```

❶ If there is no data, raise an exception. This happens when the connection was closed by the client or the connection was shut down.

❷ When we have an exception, exit the run method. This terminates the thread.

❸ Shut down the connection if the thread is alive; the thread may not be alive if the client closed the connection.

❹ Shut down the client connection for reads and writes.

❺ Call the close method on our threads to shut down each client connection on keyboard interrupt.

We first create a new class, `ClientEchoThread`, that inherits from `Thread`. This class overrides the `run` method with the code from our original `echo` function, but with a few changes. First, we wrap everything in a `try` `catch` block and intercept `OSError` exceptions. This type of exception is thrown from methods such as `sendall` when we close the client socket. We also check to see if the data from `recv` is `0`. This happens in two cases: if the client closes the connection (someone quits telnet, for example) or when we shut down the client connection ourselves. In this case we throw a `BrokenPipeError` ourselves (a subclass of `OSError`), execute the print statement in the except block, and exit the `run` method, which shuts down the thread.

We also define a `close` method on our `ClientEchoThread` class. This method first checks to see if the thread is alive before shutting down the client connection. What does it mean for a thread to be "alive," and why do we need to do this? A thread is alive if its `run` method is executing; in this case this is true if our run method does not throw any exceptions. We need this check because the client itself may have closed the connection, resulting in a `BrokenPipeError` exception in the `run` method before we call `close`. This means that calling `sendall` would result in an exception, as the connection is no longer valid.

Finally, in our main loop, which listens for new incoming connections, we intercept `KeyboardInterrupt` exceptions. Once we have one, we call the `close` method on each thread we've created. This will send a message to the client, assuming the connection is still active and shut down the connection.

Overall, canceling running threads in Python, and in general, is a tricky problem and depends on the specific shutdown case you're trying to handle. You'll need to take special care that your threads do not block your application from exiting and to figure out where to put in appropriate interrupt points to exit your threads.

We've now seen a couple ways to manage threads manually ourselves, creating a thread object with a `target` function and subclassing `Thread` and overriding the `run` method. Now that we understand threading basics, let's see how to use them with asyncio to work with popular blocking libraries.

ing threads with asyncio

We now know how to create and manage multiple threads to handle blocking work. The drawback of this approach is that we must individually create and keep track of threads. We'd like to be able to use all the asyncio-based APIs we've learned to wait for results from threads without having to manage them ourselves. Like process pools from chapter 6, we can use *thread pools* to manage threads in this manner. In this section, we'll introduce a popular blocking HTTP client library and see how to use threads with asyncio to run web requests concurrently.

ntroducing the requests library

The *requests library* is a popular HTTP client library for Python, self-described as "HTTP for humans." You can view the latest documentation for the library at

https://requests.readthedocs.io/en/master/. Using it, you can make HTTP requests to web servers much like we did with aiohttp. We'll use the latest version (as of this writing, version 2.24.0). You can install this library by running the following `pip` command:

```
pip install -Iv requests==2.24.0
```

Once we've installed the library, we're ready to make some basic HTTP requests. Let's start out by making a couple of requests to example.com to retrieve the status code, as we did earlier with aiohttp.

Listing 7.3 Basic usage of requests

```
import requests


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


url = 'https:// www .example .com'
print(get_status_code(url))
print(get_status_code(url))
```

The preceding listing executes two `HTTP` `GET` requests in series. Running this, you should see two `200` outputs. We didn't create a HTTP session here, as we did with aiohttp, but the library does support this as needed to keep cookies persistent across different requests.

The requests library is blocking, meaning that each call to `requests.get` will stop any thread from executing other Python code until the request finishes. This has implications for how we can use this library in asyncio. If we try to use this library in a coroutine or a task by itself, it will block the entire event loop until the request finishes. If we had a HTTP request that took 2 seconds, our application wouldn't be able to do anything other than wait for those 2 seconds. To properly use this library with asyncio, we must run these blocking operations inside of a thread.

ntroducing thread pool executors

Much like process pool executors, the `concurrent.futures` library provides an implementation of the `Executor` abstract class to work with threads named `ThreadPoolExecutor`. Instead of maintaining a pool of worker processes like a process pool does, a thread pool executor will create and maintain a pool of threads that we can then submit work to.

While a process pool will by default create one worker process for each CPU core our machine has available, determining how many worker threads to create is a bit more complicated. Internally, the formula for the default number of threads is `min(32, os.cpu_count() + 4)`. This causes the maximum (upper) bound of worker threads to be 32 and the minimum (lower) bound to be 5. The upper bound is set to 32 to avoid creating a surprising number of threads on machines with large amounts of CPU cores (remember, threads are resource-expensive to create and maintain). The lower bound is set to 5 because on smaller 1–2 core machines, spinning up only a couple of threads isn't likely to improve performance much. It often makes sense to create a few more threads than your available CPUs for I/O-bound work. For example, on an 8-core machine the above formula means we'll create 12 threads.

While only 8 threads can run concurrently, we can have other threads paused waiting for I/O to finish, letting our operating resume them when I/O is done.

Let's adapt our example from listing 7.3 to run 1,000 HTTP requests concurrently with a thread pool. We'll time the results to get an understanding of what the benefit is.

Listing 7.4 Running requests with a thread pool

```
import time
import requests
from concurrent.futures import ThreadPoolExecutor


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


start = time.time()

with ThreadPoolExecutor() as pool:
    urls = ['https:// www .example .com' for _ in range(1000
    results = pool.map(get_status_code, urls)
    for result in results:
        print(result)

end = time.time()

print(f'finished requests in {end - start:.4f} second(s)')
```

On an 8-core machine with a speedy internet connection, this code can execute in as little as 8–9 seconds with the default number of threads. It is easy to write this synchronously to understand the impact that threading has by doing something, as in the following:

```
start = time.time()

urls = ['https:// www .example .com' for _ in range(1000)]

for url in urls:
    print(get_status_code(url))

end = time.time()

print(f'finished requests in {end - start:.4f} second(s)')
```

Running this code can take upwards of 100 seconds! This makes our threaded code a bit more than 10 times faster than our synchronous code, giving us a pretty big performance bump.

While this is clearly an improvement, you may remember from chapter 4, on aiohttp, that we were able to make 1,000 requests concurrently in less than 1 second. Why is this so much slower than our threading version? Remember that our maximum number of worker threads is limited to 32 (that is, the number of CPUs plus 4), meaning that by default we can only run a maximum of 32 requests concurrently. We can try to get around this by passing in `max_workers=1000` when we create our thread pool, as in the following:

```
with ThreadPoolExecutor(max_workers=1000) as pool:
    urls = ['https:// www .example .com' for _ in range(1000
    results = pool.map(get_status_code, urls)
    for result in results:
        print(result)
```

This approach can yield some improvements, as we now have one thread per each request we make. However, this still won't come very close to our coroutine-based code. This is due to the resource overhead associated with threads. Threads are created at the operating-system level and are more expensive to create than coroutines. In addition, threads have a context-switching cost at the OS level. Saving and restoring thread state when a context switch happens eats up some of the performance gains obtained by using threads.

When you're determining the number of threads to use for a particular problem, it is best to start small (the amount of CPU cores plus a few is a good starting point), test it, and benchmark it, gradually increasing the number of threads. You'll usually find a "sweet spot," after which the run time will plateau and may even degrade, no matter how many more threads you add. This sweet spot is usually a fairly low number relative to the requests you want to make (to make it clear, creating 1,000 threads for 1,000 requests probably isn't the best use of resources).

Thread pool executors with asyncio

Using thread pool executors with the asyncio event loop isn't much different than using `ProcessPoolExecutors`. This is the beauty of having the abstract `Executor` base class in that we can use the same code to run threads or processes

by only having to change one line of code. Let's adapt our example of running 1,000 HTTP requests to use `asyncio.gather` instead of `pool.map`.

Listing 7.5 Using a thread pool executor with asyncio

```
import functools
import requests
import asyncio
from concurrent.futures import ThreadPoolExecutor
from util import async_timed


def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        urls = ['https:// www .example .com' for _ in range(
        tasks = [loop.run_in_executor(pool, functools.partia
        results = await asyncio.gather(*tasks)
        print(results)

asyncio.run(main())
```

We create the thread pool as we did before, but instead of using `map` we create a list of tasks by calling our `get_status_code` function with

`loop.run_in_executor`. Once we have a list of tasks, we can wait for them to finish with `asyncio.gather` or any of the other asyncio APIs we learned earlier.

Internally, `loop.run_in_executor` calls the thread pool executor's `submit` method. This will put each function we pass in onto a queue. Worker threads in the pool then pull from the queue, running each work item until it completes. This approach does not yield any performance benefits over using a pool without asyncio, but while we're waiting for `await` `asyncio.gather` to finish, other code can run.

Default executors

Reading the asyncio documentation, you may notice that the `run_in_executor` method's `executor` parameter can be `None`. In this case, `run_in_executor` will use the event loop's *default executor*. What is a default executor? Think of it as a reusable singleton executor for your entire application. The default executor will always default to a `ThreadPoolExecutor` unless we set a custom one with the `loop.set_default_executor` method. This means that we can simplify the code from listing 7.5, as shown in the following listing.

Listing 7.6 Using the default executor

```
import functools
import requests
import asyncio
from util import async_timed


def get_status_code(url: str) -> int:
    response = requests.get(url)
```