```
    return response.status_code


@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    urls = ['https:// www .example .com' for _ in range(1000
    tasks = [loop.run_in_executor(None, functools.partial(ge
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

In the preceding listing, we eliminate creating our own `ThreadPoolExecutor` and using it in a context manager as we did before and, instead, pass in `None` as the executor. The first time we call `run_in_executor`, asyncio creates and caches a default thread pool executor for us. Each subsequent call to `run_in_executor` reuses the previously created default executor, meaning the executor is then global to the event loop. Shutdown of this pool is also different from what we saw before. Previously, the thread pool executor we created was shut down when we exited a context manager's `with` block. When using the default executor, it won't shut down until the event loop closes, which usually happens when our application finishes. Using the default thread pool executor when we want to use threads simplifies things, but can we make this even easier?

In Python 3.9, the `asyncio.to_thread` coroutine was introduced to further simplify putting work on the default thread pool executor. It takes in a function to run in a thread and a set of arguments to pass to that function. Previously, we had to use `functools.partial` to pass in arguments, so this makes our code a little cleaner.

It then runs the function with its arguments in the default thread pool executor and the currently running event loop. This lets us simplify our threading code even more. Using the `to_thread` coroutine eliminates using `functools.partial` and our call to `asyncio.get_running_loop`, cutting down our total lines of code.

Listing 7.7 Using the `to_thread` coroutine

```
import requests
import asyncio
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():
    urls = ['https:// www .example .com' for _ in range(1000
    tasks = [asyncio.to_thread(get_status_code, url) for url
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

So far, we've only seen how to run blocking code inside of threads. The power of combining threads with asyncio is that we can run other code while we're waiting for our threads to finish. To see how to run other code while threads are running, we'll revisit our example from chapter 6 of periodically outputting the status of a long-running task.

cks, shared data, and deadlocks

Much like multiprocessing code, multithreaded code is also susceptible to race conditions when we have shared data, as we do not control the order of execution. Any time you have two threads or processes that could modify a shared piece of non-thread-safe data, you'll need to utilize a lock to properly synchronize access. Conceptually, this is no different from the approach we took with multiprocessing; however, the memory model of threads changes the approach slightly.

Recall that with multiprocessing, by default the processes we create do not share memory. This meant we needed to create special shared memory objects and properly initialize them so that each process could read from and write to that object. Since threads *do* have access to the same memory of their parent process, we no longer need to do this, and threads can access shared variables directly.

This simplifies things a bit, but since we won't be working with shared `Value` objects that have locks built in, we'll need to create them ourselves. To do this, we'll need to use the threading module's `Lock` implementation, which is different from the one we used with multiprocessing. This is as easy as importing `Lock` from the threading module and calling its `acquire` and `release` methods around critical sections of code or using it in a context manager.

To see how to use locks with threading, let's revisit our task from chapter 6 of keeping track and displaying the progress of a long task. We'll take our previous example of making thousands of web requests and use a shared counter to keep track of how many requests we've completed so far.

Listing 7.8 Printing status of requests

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

counter_lock = Lock()
counter: int = 0



def get_status_code(url: str) -> int:
    global counter
    response = requests.get(url)
    with counter_lock:
        counter = counter + 1
    return response.status_code



async def reporter(request_count: int):
    while counter < request_count:
        print(f'Finished {counter}/{request_count} requests'
        await asyncio.sleep(.5)



@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        request_count = 200
        urls = ['https:// www .example .com' for _ in range(
        reporter_task = asyncio.create_task(reporter(request
        tasks = [loop.run_in_executor(pool, functools.partia
        results = await asyncio.gather(*tasks)
        await reporter_task
        print(results)


asyncio.run(main())
```

This should look familiar, as it is like the code we wrote to output progress of our `map` operation in chapter 6. We create a global `counter` variable as well as a `counter_lock` to synchronize access to it in critical sections. In our `get_status_code` function we acquire the lock when we increment the counter. Then, in our main coroutine we kick off a reporter background task that outputs how many requests we've finished every 500 milliseconds. Running this, you should see output similar to the following:

```
Finished 0/200 requests
Finished 48/200 requests
Finished 97/200 requests
Finished 163/200 requests
```

We now know the basics around locks with both multithreading and multiprocessing, but there is still quite a bit to learn about locking. Next, we'll look at the concept of *reentrancy*.

eentrant locks

Simple locks work well for coordinating access to a shared variable across multiple threads, but what happens when a thread tries to acquire a lock it has already acquired? Is this even safe? Since the same thread is acquiring the lock, this should be okay since this is single-threaded by definition and, therefore, thread-safe.

While this access should be okay, it does cause problems with the locks we have been using so far. To illustrate this, let’s imagine we have a recursive sum function that takes a list of integers and produces the sum of the list. The list we want to sum can be modified from multiple threads, so we need to use a lock to ensure the list we’re summing does not get modified during our sum operation. Let’s try implementing this with a normal lock to see what happens. We’ll also add some console output to see how our function is executing.

Listing 7.9 Recursion with locks

```
from threading import Lock, Thread
from typing import List

list_lock = Lock()


def sum_list(int_list: List[int]) -> int:
    print('Waiting to acquire lock...')
    with list_lock:
        print('Acquired lock.')
        if len(int_list) == 0:
            print('Finished summing.')
            return 0
        else:
            head, *tail = int_list
            print('Summing rest of list.')
            return head + sum_list(tail)


thread = Thread(target=sum_list, args=([1, 2, 3, 4],))
thread.start()
thread.join()
```

If you run this code, you'll see the following few messages and then the application will hang forever:

```
Waiting to acquire lock...
Acquired lock.
Summing rest of list.
Waiting to acquire lock...
```

Why is this happening? If we walk through this, we acquire `list_lock` the first time perfectly fine. We then unpack the list and recursively call `sum_list` on the remainder of the list. This then causes us to attempt to acquire `list_lock` a second time. This is where our code hangs because, since we already acquired the lock, we block forever trying to acquire the lock a second time. This also means we never exit the first `with` block and can't release the lock; we're waiting for a lock that will never be released!

Since this recursion is coming from the same thread that originated it, acquiring the lock more than once shouldn't be a problem as this won't cause race conditions. To support these use cases, the threading library provides *reentrant* locks. A reentrant lock is a special kind of lock that can be acquired by the same thread more than once, allowing that thread to “reenter” critical sections. The threading module provides reentrant locks in the `RLock` class. We can take our above code and fix the problem by modifying only two lines of code—the `import` statement and the creation of the `list_lock`:

```
from threading import Rlock

list_lock = RLock()
```

If we modify these lines our code will work properly, and a single thread will be able to acquire the lock multiple times. Internally, reentrant locks work by keeping a recursion count. Each time we acquire the lock from the thread that first acquired the lock, the count increases, and each time we release the lock it decreases. When the count is 0, the lock is finally released for other threads to acquire it.

Let’s examine a more real-world application to truly understand the concept of recursion with locks. Imagine we’re trying to build a thread-safe integer list class with a method to find and replace all elements of a certain value with a different value. This class will contain a normal Python list and a lock we use to prevent race conditions. We’ll pretend our existing class already has a method called `indices_of(to_ find:` `int)` that takes in an integer and returns all the indices in the list that match `to_find`. Since we want to follow the DRY (don’t repeat yourself) rule, we’ll reuse this method when we define our find-and-replace method (note this is not the technically the most efficient way to do this, but we’ll do it to illustrate the concept). This means our class and method will look something like the following listing.

Listing 7.10 A thread-safe list class

```
from threading import Lock
from typing import List


class IntListThreadsafe:

    def __init__(self, wrapped_list: List[int]):
        self._lock = Lock()
        self._inner_list = wrapped_list

    def indices_of(self, to_find: int) -> List[int]:
        with self._lock:
            enumerator = enumerate(self._inner_list)
            return [index for index, value in enumerator if

    def find_and_replace(self,
                         to_replace: int,
                         replace_with: int) -> None:
        with self._lock:
            indices = self.indices_of(to_replace)
            for index in indices:
                self._inner_list[index] = replace_with


threadsafe_list = IntListThreadsafe([1, 2, 1, 2, 1])
threadsafe_list.find_and_replace(1, 2)
```

If someone from another thread modifies the list during our `indices_of` call, we could obtain an incorrect return value, so we need to acquire the lock before we search for matching indices. Our `find_and_replace` method must acquire the lock for the same reason. However, with a normal lock we wind up hanging forever when we call `find_and_replace`. The find-and-replace method first acquires the lock and then calls another method, which tries to acquire the same lock. Switching to an `RLock` in this case will fix this problem because one call to `find_and_replace` will always acquire any locks from the same thread. This illustrates a generic formula for when you need to use reentrant locks. If you are developing a thread-safe class with a method A, which acquires a lock, and a method B that also needs to acquire a lock *and* call method A, you likely need to use a reentrant lock.

eadlocks

You may be familiar with the concept of *deadlock* from political negotiations on the news, where one party makes a demand of the other side, and the other side makes a counter-demand. Both sides disagree on the next step and negotiation reaches a standstill. The concept in computer science is similar in that we reach a state where there is contention over a shared resource with no resolution, and our application hangs forever.

The issue we saw in the previous section, where non-reentrant locks can cause our program to hang forever, is one example of a deadlock. In that case, we reach a state where we’re stuck in a standstill negotiation with ourselves, demanding to acquire a lock that is never released. This situation can also arise when we have two threads using more than one lock. Figure 7.1 illustrates this scenario: if thread `A` asks for a lock that thread `B` has acquired, and thread `B` is asking for a lock that `A` has acquired, we reach a standstill and a deadlock. In that instance, using reentrant locks won't help, as we have multiple threads stuck waiting on a resource the other thread holds.

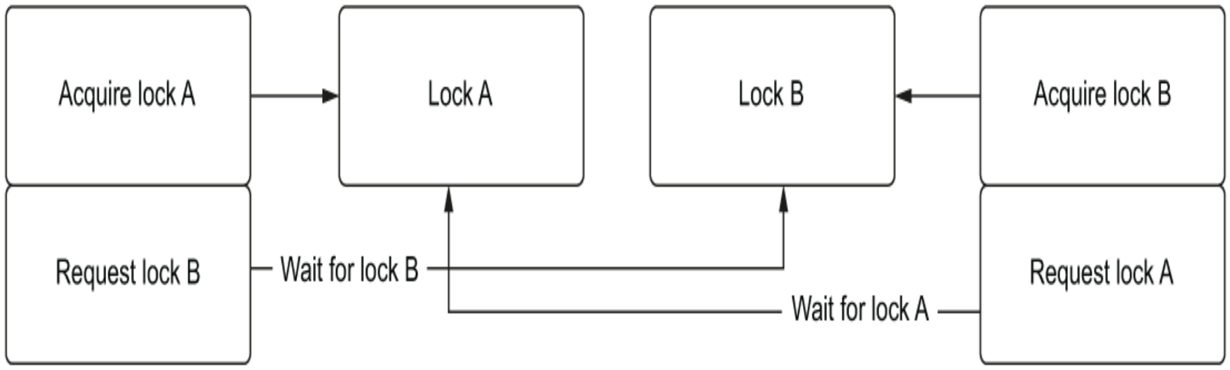

Figure 7.1 Threads 1 and 2 acquire locks A and B at roughly the same time. Then, thread 1 waits for lock B, which thread 2 holds; meanwhile, thread 2 is waiting for A, which thread 1 holds. This circular dependency causes a deadlock and will hang the application.

Let's look at how to create this type of deadlock in code. We'll create two locks, lock `A` and `B`, and two methods which need to acquire both locks. One method will acquire `A` first and then `B` and another will acquire `B` first and then `A`.

Listing 7.11 A deadlock in code

```
from threading import Lock, Thread
import time

lock_a = Lock()
lock_b = Lock()
```

```
def a():
    with lock_a:                                                    ❶
        print('Acquired lock a from method a!')
        time.sleep(1)                                               ❷
        with lock_b:                                                ❸
            print('Acquired both locks from method a!')


def b():
    with lock_b:                                                    ❸
        print('Acquired lock b from method b!')
        with lock_a:                                                ❶
            print('Acquired both locks from method b!')
thread_1 = Thread(target=a)
thread_2 = Thread(target=b)
thread_1.start()
thread_2.start()
thread_1.join()
thread_2.join()
```

❶ Acquire lock A.

❷ Sleep for 1 second; this ensures we create the right conditions for deadlock.

❸ Acquire lock B.

When we run this code, we’ll see the following output, and our application will hang forever:

```
Acquired lock a from method a!
Acquired lock b from method b!
```

We first call method `A` and acquire lock `A`, then we introduce an artificial delay to give method `B` a chance to acquire lock `B`. This leaves us in a state where method `A` holds lock `A` and method `B` holds lock `B`. Next, method `A` attempts to acquire lock `B`, but method `B` is holding that lock. At the same time, method `B` tries to acquire lock `A`, but method `A` is holding it, stuck waiting for `B` to release its lock. Both methods are stuck waiting on one another to release a resource, and we reach a standstill.

How do we handle this situation? One solution is the so-called “ostrich algorithm,” named for the situation (although ostriches don’t *actually* behave this way) where an ostrich sticks its head in the sand whenever it senses danger. With this strategy, we ignore the problem and devise a strategy to restart our application when we encounter the issue. The driving idea behind this approach is if the issue happens rarely enough, investing in a fix isn’t worth it. If you remove the `sleep` from the above code, you’ll only rarely see deadlock occur, as it relies on a very specific sequence of operations. This isn’t really a fix and isn’t ideal but is a strategy used with deadlocks that rarely occur.

However, in our situation there is an easy fix, where we change the locks in both methods to always be acquired in the same order. For instance, both methods `A` and `B` can acquire lock `A` first then lock `B`. This resolves the issue, as we’ll never acquire locks in an order where a deadlock could occur. The other option would be to refactor the locks so we use only one instead of two. It is impossible to have a deadlock with one lock (excluding the reentrant deadlock we saw earlier). Overall, when dealing with multiple locks that you need to acquire, ask yourself, “Am I acquiring these locks in a consistent order? Is there a way I can refactor this to use only one lock?”

We’ve now seen how to use threads effectively with asyncio and have investigated more complex locking scenarios. Next, let’s see how to use threads to integrate asyncio into existing synchronous applications that may not work smoothly with asyncio.

ent loops in separate threads

We have mainly focused on building applications that are completely implemented from the bottom up with coroutines and asyncio. When we’ve had any work that does not fit within a single-threaded concurrency model, we have run it inside of threads or processes. Not all applications will fit into this paradigm. What if we’re working in an existing synchronous application and we want to incorporate asyncio?

One such situation where we can run into this scenario is building desktop user interfaces. The frameworks to build GUIs usually run their own event loop, and the event loop blocks the main thread. This means that any long-running operations can cause the user interface to freeze. In addition, this UI event loop will block us from creating an asyncio event loop. In this section, we’ll learn how to use multithreading to run multiple event loops at the same time by building a responsive HTTP stress-testing user interface in Tkinter.

ntroducing Tkinter

Tkinter is a platform-independent desktop graphical user interface (GUI) toolkit provided in the default Python installation. Short for “Tk interface,” it is an interface to the low-level Tk GUI toolkit that is written in the tcl language. With the creation of the Tkinter Python library, Tk has grown into a popular way for Python developers to build desktop user interfaces.

Tkinter has a set of “widgets,” such as labels, text boxes, and buttons, that we can place in a desktop window. When we interact with a widget, such as entering text or pressing a button, we can trigger a function to execute code. The code that runs in response to a user action could be as simple as updating another widget or triggering another operation.

Tkinter, and many other GUI libraries, draw their widgets and handle widget interactions through their own event loops. The event loop is constantly redrawing the application, processing events, and checking to see if any code should run in response to a widget event. To get familiar with Tkinter and its event loop, let’s create a basic `hello` `world` application. We’ll create an application with a “say hello” button that will output “Hello there!” to the console when we click on it.

Listing 7.12 “Hello world” with Tkinter

```
import tkinter
from tkinter import ttk

window = tkinter.Tk()
window.title('Hello world app')
window.geometry('200x100')


def say_hello():
    print('Hello there!')
```

```
hello_button = ttk.Button(window, text='Say hello', command=
hello_button.pack()

window.mainloop()
```

This code first creates a Tkinter window (see figure 7.2) and sets the application title and window size. We then place a button on the window and set its command to the `say_hello` function. When a user presses this button, the `say_hello` function executes, printing out our message. We then call `window.mainloop()` that starts the Tk event loop, running our application.

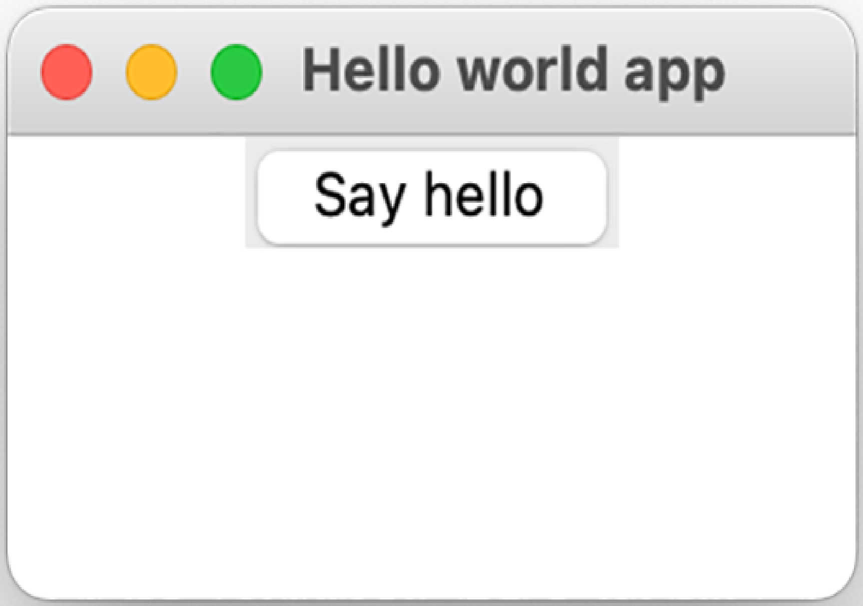

Figure 7.2 The “hello world” application from listing 7.12

One thing to note here is that our application will block on `window.mainloop()`. Internally, this method runs the Tk event loop. This is an infinite loop that is checking for window events and constantly redrawing the window until we close it. The Tk event loop has interesting parallels to the asyncio event loop. For example, what happens if we try to run blocking work in our button’s command? If we add a 10-second delay to the `say_hello` function with `time.sleep(10)`, we’ll start to see a problem: our application will freeze for 10 seconds!

Much like asyncio, Tkinter runs *everything* in its event loop. This means that if we have a long-running operation, such as making a web request or loading a large file, we'll block the tk event loop until that operation finishes. The effect on the user is that the UI hangs and becomes unresponsive. The user can't click on any buttons, we can't update any widgets with status or progress, and the operating system will likely display a spinner (like the example in figure 7.3) to indicate the application is hanging. This is clearly an undesirable, unresponsive user interface.

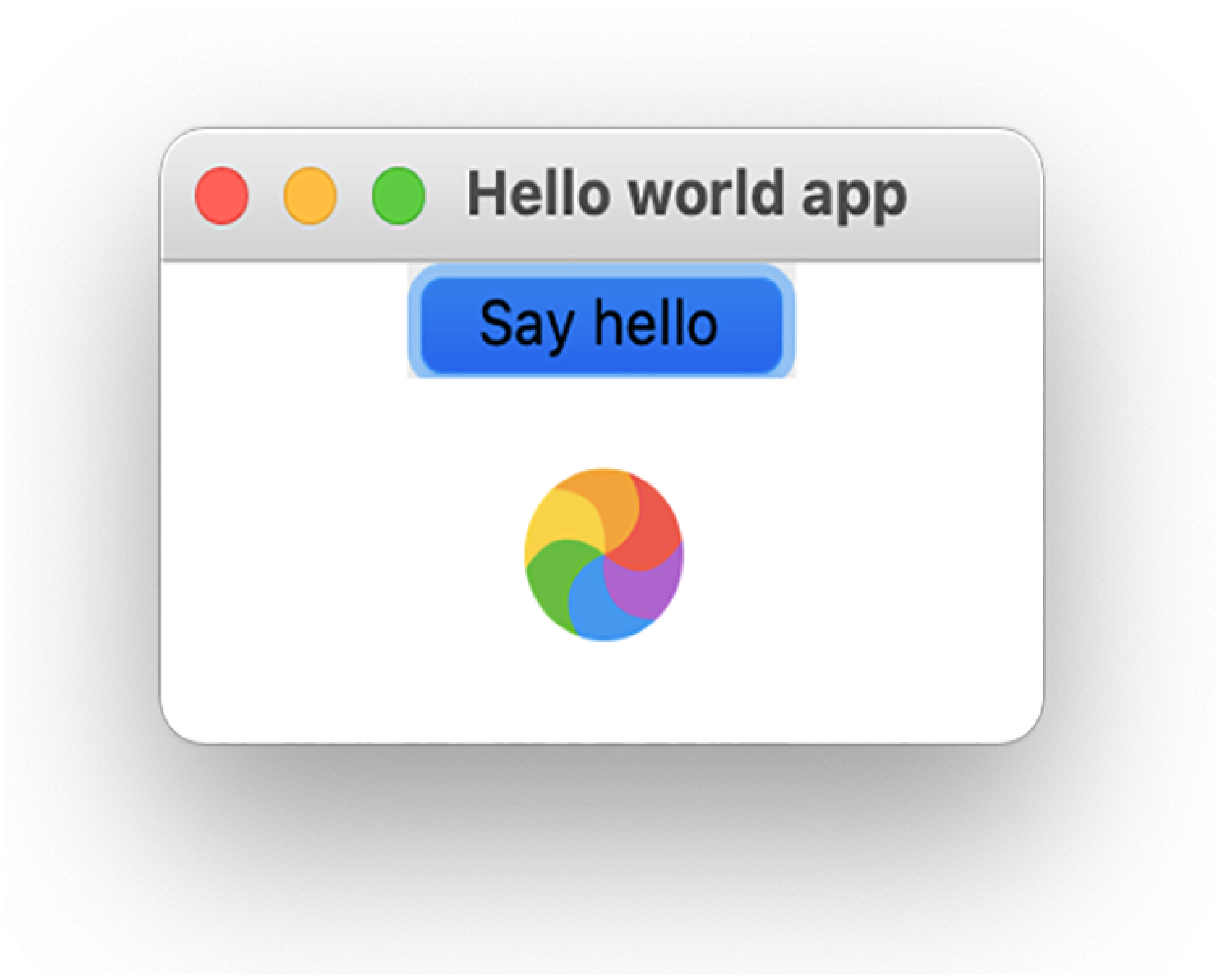

Figure 7.3 The dreaded "beach ball of doom" occurs as we block the event loop on a Mac.

This is an instance where asynchronous programming can, in theory, help us out. If we can make asynchronous requests that don't block the tk event loop, we can avoid this problem. This is trickier than it may seem as Tkinter is not asyncio-aware, and you can't pass in a coroutine to run on a button click. We could try running two event loops at the same time in the same thread, but this won't work. Both Tkinter and asyncio are single-threaded—this idea is the same as trying to run two infinite loops in the same thread at the same time, which can't be done. If we start the asyncio event loop before the Tkinter event loop, the asyncio event loop will block the Tkinter loop from running, and vice versa. Is there a way for us to run an asyncio application alongside a single-threaded application?

We can in fact combine these two event loops to create a functioning application by running the asyncio event loop in a separate thread. Let's look at how to do this with an application that will responsively update the user on the status of a long-running task with a progress bar.

uilding a responsive UI with asyncio and threads

First, let's introduce our application and sketch out a basic UI. We'll build a URL stress test application. This application will take a URL and many requests to send as input. When we press a submit button, we'll use aiohttp to send out web requests as fast as we can, delivering a predefined load to the web server we choose. Since this may take a long time, we'll add a progress bar to visualize how far along we are in the test. We'll update the progress bar after every 1% of total requests are finished to show progress. Further, we'll let the user cancel the request if they'd like. Our UI will have a few widgets, including a text input for the URL to test, a text input for the number of requests we wish to issue, a start button, and a progress bar. We'll design a UI that looks like the illustration in figure 7.4.

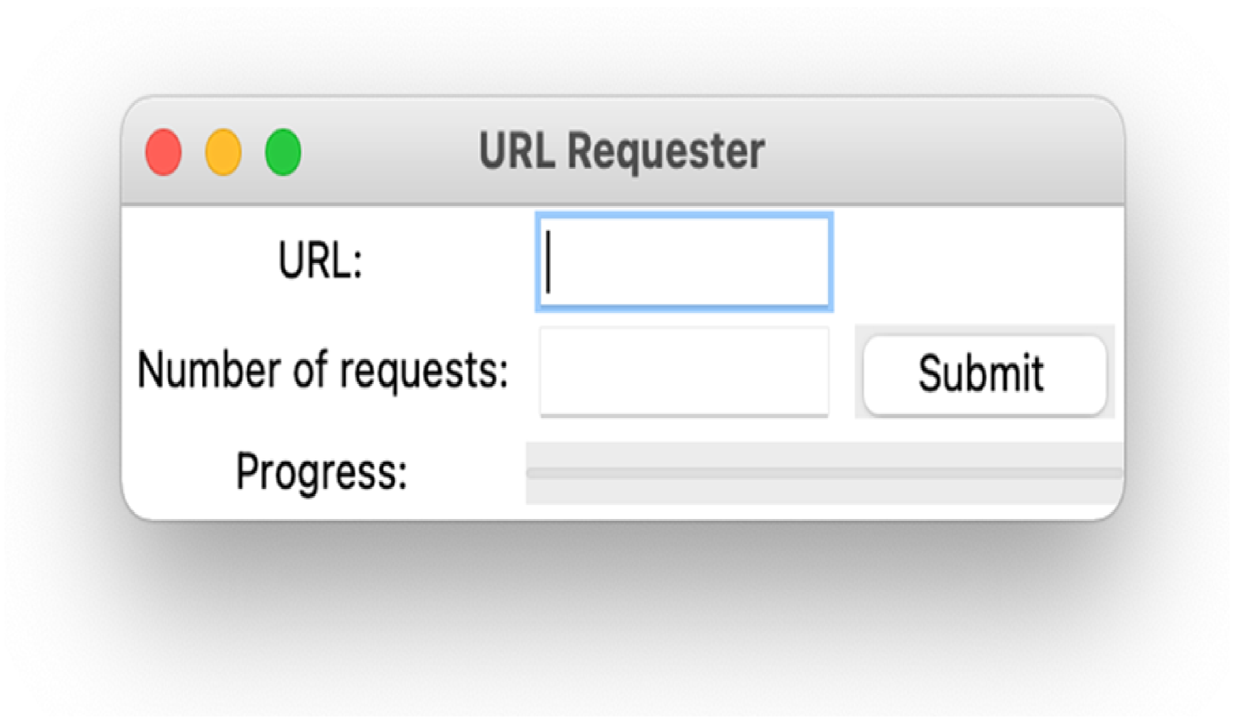

Figure 7.4 The URL requester GUI

Now that we have our UI sketched out, we need to think through how to have two event loops running alongside one another. The basic idea is that we'll have the Tkinter event loop running in the main thread, and we'll run the asyncio event loop in a separate thread. Then, when the user clicks "Submit," we'll submit a coroutine to the asyncio event loop to run the stress test. As the stress test is running, we'll issue commands from the asyncio event loop back to the Tkinter event loop to update our progress. This gives us an architecture that looks like the drawing in figure 7.5.

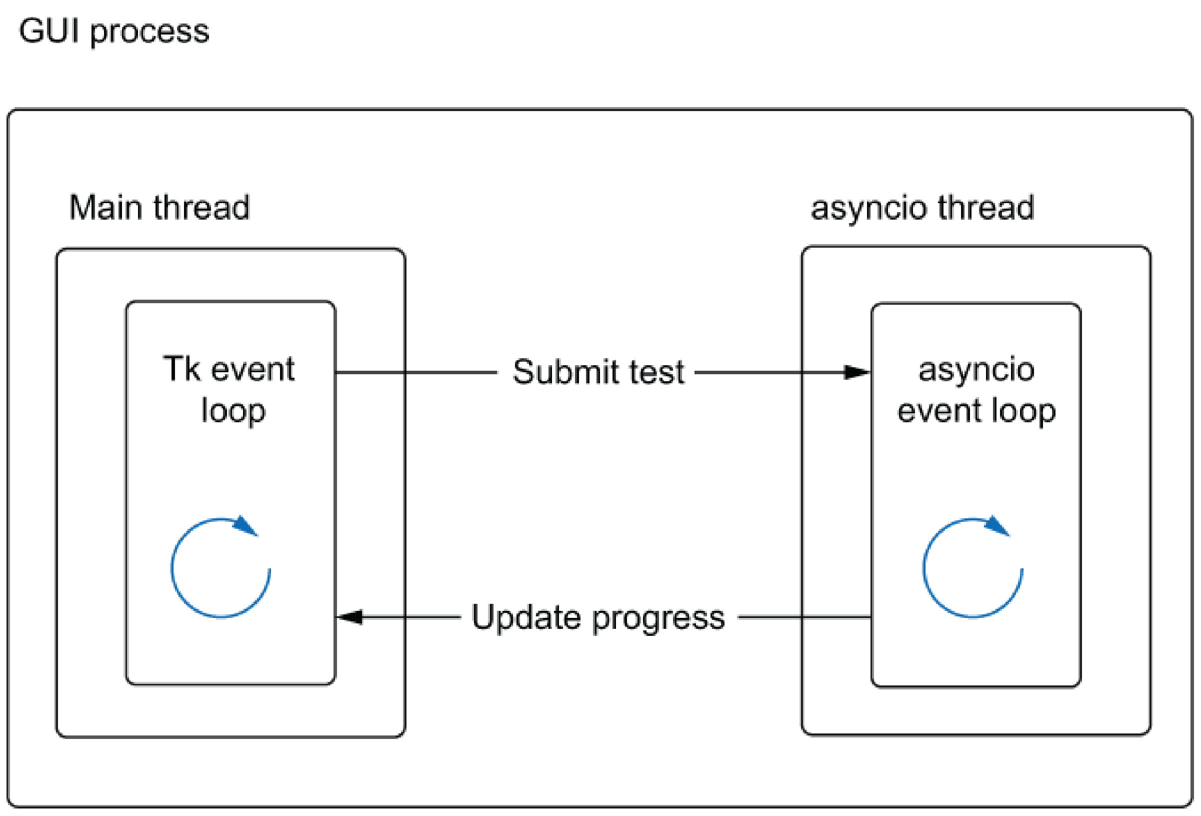

Figure 7.5 The tk event loop submits a task to the asyncio event loop, which runs in a separate thread.

This new architecture includes communication across threads. We need to be careful about race conditions in this situation, especially since the asyncio event loop is *not* thread-safe! Tkinter is designed with thread safety in mind, so there are fewer concerns with calling it from a separate thread (in Python 3+ at least; we'll look more closely at this soon).

We may be tempted to submit coroutines from Tkinter using `asyncio.run`, but this function blocks until the coroutine we pass in finishes and will cause the Tkinter application to hang. We'll need a function which submits a coroutine to the event loop without any blocking. There are a few new asyncio functions to learn that are both non-blocking and have thread safety built in to submit this kind of work properly. The first is a method on the asyncio event loop named `call_soon_threadsafe`. This function takes in a Python function (not a coroutine) and schedules it to execute it in a thread-safe manner at the next iteration of the asyncio event loop. The second function is `asyncio.run_ coroutine_threadsafe`. This function takes in a coroutine and submits it to run in a thread-safe manner, immediately returning a future that we can use to access a result of the coroutine. Importantly, and confusingly, this future is *not* an asyncio future but rather from the `concurrent.futures` module. The logic behind this is that asyncio futures are not thread-safe, but `concurrent.futures` futures are. This `future` class does however have the same functionality as the future from the asyncio module.

Let's start defining and implementing a few classes to build our stress test application based on what we described above. The first thing we'll build is a stress test class. This class will be responsible for starting and stopping one stress test and keeping track of how many requests have completed. Its constructor will take in a URL, an asyncio event loop, the number of desired requests to make, and a progress updater callback. We'll call this callback when we want to trigger a progress bar update. When we get to implementing the UI, this callback will trigger an update to the progress bar. Internally, we'll calculate a refresh rate, which is the rate at which we'll execute the callback. We'll default this rate to every 1% of the total requests we plan to send.

Listing 7.13 The stress test class

```
import asyncio
from concurrent.futures import Future
from asyncio import AbstractEventLoop
from typing import Callable, Optional
from aiohttp import ClientSession
```

```
class StressTest:

    def __init__(self,
                 loop: AbstractEventLoop,
                 url: str,
                 total_requests: int,
                 callback: Callable[[int, int], None]):
        self._completed_requests: int = 0
        self._load_test_future: Optional[Future] = None
        self._loop = loop
        self._url = url
        self._total_requests = total_requests
        self._callback = callback
        self._refresh_rate = total_requests // 100

    def start(self):
        future = asyncio.run_coroutine_threadsafe(self._make
        self._load_test_future = future

    def cancel(self):
        if self._load_test_future:
            self._loop.call_soon_threadsafe(self._load_test_

    async def _get_url(self, session: ClientSession, url: st
        try:
            await session.get(url)
        except Exception as e:
            print(e)
        self._completed_requests = self._completed_requests
        if self._completed_requests % self._refresh_rate ==
                or self._completed_requests == self._total_r
            self._callback(self._completed_requests, self._t

    async def _make_requests(self):
        async with ClientSession() as session:
            reqs = [self._get_url(session, self._url) for _
            await asyncio.gather(*reqs)
```

❶ Start making the requests, and store the future, so we can later cancel if needed.

❷ If we want to cancel, call the cancel function on the load test future.

❸ Once we've completed 1% of requests, call the callback with the number of completed requests and the total requests.

In our `start` method, we call `run_coroutine_threadsafe` with `_make_requests` that will start making requests on the asyncio event loop. We also keep track of the future this returns in `_load_test_future`. Keeping track of this future lets us cancel the load test in our `cancel` method. In our `_make_requests` method we create a list coroutines to make all our web requests, passing them into `asyncio.gather` to run them. Our `_get_url` coroutine makes the request, increments the `_completed_requests` counter, and calls the callback with the total number of completed requests if necessary. We can use this class by simply instantiating it and calling the `start` method, optionally canceling by calling the `cancel` method.

One interesting thing to note is that we didn't use any locking around the `_` `completed` `requests` counter despite updates happening to it from multiple coroutines. Remember that asyncio is single-threaded, and the asyncio event loop only runs a piece of Python code at any given time. This has the effect of making incrementing the counter atomic when used with asyncio, despite it being non-atomic when happening between multiple threads. asyncio saves us from many kinds of race conditions that we see with multithreading but not all. We'll examine this more in a later chapter.

Next, let's implement our Tkinter GUI to use this load tester class. For code cleanliness, we'll subclass the `TK` class directly and initialize our widgets in the constructor. When a user clicks the start button, we'll create a new `StressTest` instance and start it. The question now becomes what do we pass in as a callback to our `StressTest` instance? Thread safety becomes an issue here as our callback will be called in the worker thread. If our callback modifies shared data from the worker thread that our main thread can also modify, this could cause race conditions. In our case, since Tkinter has thread safety built in and all we're doing is updating the progress bar, we should be okay. But what if we needed to do something with shared data? Locking is one approach, but if we could run our callback in the main thread, we'd avoid any race conditions. We'll use a generic pattern to demonstrate how to do this, though updating the progress bar directly should be safe.

One common pattern to accomplish this is to use a shared thread-safe queue from the `queue` module. Our asyncio thread can put progress updates into this queue. Then, our Tkinter thread can check this queue for updates in its thread, updating the progress bar in the correct thread. We'll need to tell Tkinter to poll the queue in the main thread to do this.

Tkinter has a method that lets us queue up a function to run after a specified time increment in the main thread called `after`. We'll use this to run a method that asks the queue if it has a new progress update (listing 7.14). If it does, we can update the progress bar safely from the main thread. We’ll poll the queue every 25 milliseconds to ensure we get updates with reasonable latency.

Is Tkinter really thread-safe?

If you search for Tkinter and thread safety, you'll find a lot of conflicting information. The threading situation in Tkinter is quite complicated. This is in part because, for several years, Tk and Tkinter lacked proper thread support. Even when threaded mode was added, it had several bugs that have since been fixed. Tk supports both non-threaded and threaded modes. In non-threaded mode, there is no thread safety; and using Tkinter from anything other than the main thread is inviting a crash. In older versions of Python, Tk thread safety was not turned on; however, in versions of Python 3 and later, thread safety is turned on by default and we have thread-safe guarantees. In threaded mode, if an update is issued from a worker thread, Tkinter acquires a mutex and writes the update event to a queue for the main thread to later process. The relevant code where this happens is in CPython in the `Tkapp_Call` function in `Modules/_tkinter.c`.

Listing 7.14 The Tkinter GUI

```
from queue import Queue
from tkinter import Tk
from tkinter import Label
from tkinter import Entry
from tkinter import ttk
from typing import Optional
from chapter_07.listing_7_13 import StressTest


class LoadTester(Tk):
    def __init__(self, loop, *args, **kwargs):
        Tk.__init__(self, *args, **kwargs)
        self._queue = Queue()
        self._refresh_ms = 25

        self._loop = loop
        self._load_test: Optional[StressTest] = None
        self.title('URL Requester')

        self._url_label = Label(self, text="URL:")
        self._url_label.grid(column=0, row=0)

        self._url_field = Entry(self, width=10)
        self._url_field.grid(column=1, row=0)

        self._request_label = Label(self, text="Number of re
        self._request_label.grid(column=0, row=1)

        self._request_field = Entry(self, width=10)
        self._request_field.grid(column=1, row=1)

        self._submit = ttk.Button(self, text="Submit", comma
        self._submit.grid(column=2, row=1)

        self._pb_label = Label(self, text="Progress:")
        self._pb_label.grid(column=0, row=3)

        self._pb = ttk.Progressbar(self, orient="horizontal"
        self._pb.grid(column=1, row=3, columnspan=2)
    def _update_bar(self, pct: int):
        if pct == 100:
            self._load_test = None
            self._submit['text'] = 'Submit'
        else:
            self._pb['value'] = pct
            self.after(self._refresh_ms, self._poll_queue)

    def _queue_update(self, completed_requests: int, total_r
        self._queue.put(int(completed_requests / total_reque

    def _poll_queue(self):
        if not self._queue.empty():
            percent_complete = self._queue.get()
            self._update_bar(percent_complete)
        else:
            if self._load_test:
                self.after(self._refresh_ms, self._poll_queu

    def _start(self):
        if self._load_test is None:
            self._submit['text'] = 'Cancel'
            test = StressTest(self._loop,
                              self._url_field.get(),
                              int(self._request_field.get())
                              self._queue_update)
            self.after(self._refresh_ms, self._poll_queue)
            test.start()
            self._load_test = test
        else:
            self._load_test.cancel()
            self._load_test = None
            self._submit['text'] = 'Submit'
```

❶ In our constructor, we set up the text inputs, labels, submit button, and progress bar.

❷ When clicked, our submit button will call the _start method.

❸ The update bar method will set the progress bar to a percentage complete value from 0 to 100. This method should only be called in the main thread.

❹ This method is the callback we pass to the stress test; it adds a progress update to the queue.

❺ Try to get a progress update from the queue; if we have one, update the progress bar.

❻ Start the load test, and start polling every 25 milliseconds for queue updates.

In our application's constructor, we create all the widgets we need for the user interface. Most notably, we create `Entry` widgets for the URL to test and the number of requests to run, a submit button, and a horizontal progress bar. We also use the `grid` method to arrange these widgets in the window appropriately.

When we create the submit button widget, we specify the command as the `_start` method. This method will create a `StressTest` object and starts running it unless we already have a load test running, in which case we will cancel it. When we create a `StressTest` object, we pass in the `_queue_update` method as a callback. The `StressTest` object will call this method whenever it has a progress update to issue. When this method runs, we calculate the appropriate percentage and put this into the queue. We then use Tkinter's `after` method to schedule the `_poll_queue` method to run every 25 milliseconds.

Using the queue as a shared communication mechanism instead of directly calling `_update_bar` will ensure that our `_update_bar` method runs in the Tkinter event loop thread. If we don't do this, the progress bar update would happen in the asyncio event loop as the callback is run within that thread.

Now that we've implemented the UI application, we can glue these pieces all together to create a fully working application. We'll create a new thread to run the event loop in the background and then start our newly created `LoadTester` application.

Listing 7.15 The load tester app

```
import asyncio
from asyncio import AbstractEventLoop
from threading import Thread
from chapter_07.listing_7_14 import LoadTester


class ThreadedEventLoop(Thread):                          ❶
    def __init__(self, loop: AbstractEventLoop):
        super().__init__()
        self._loop = loop
        self.daemon = True

    def run(self):
        self._loop.run_forever()


loop = asyncio.new_event_loop()

asyncio_thread = ThreadedEventLoop(loop)
asyncio_thread.start()                                    ❷
```

```
app = LoadTester(loop)                              ❸
app.mainloop()
```

❶ We create a new thread class to run the asyncio event loop forever.

❷ Start the new thread to run the asyncio event loop in the background.

❸ Create the load tester Tkinter application, and start its main event loop.

We first define a `ThreadedEventLoopClass` that inherits from `Thread` to run our event loop. In this class's constructor, we take in an event loop and set the thread to be a daemon thread. We set the thread to be daemon because the asyncio event loop will block and run forever in this thread. This type of infinite loop would prevent our GUI application from shutting down if we ran in non-daemon mode. In the thread's `run` method, we call the event loop's `run_forever` method. This method is well named, as it quite literally just starts the event loop running forever, blocking until we stop the event loop.

Once we've created this class, we create a new asyncio event loop with the `new_event_loop` method. We then create a `ThreadedEventLoop` instance, passing in the loop we just created and start it. This creates a new thread with our event loop running inside of it. Finally, we create an instance of our `LoadTester` app and call the `mainloop` method, kicking off the Tkinter event loop.

When we run a stress test with this application, we should see the progress bar update smoothly without freezing the user interface. Our application remains responsive, and we can click cancel to stop the load test whenever we please. This technique of running the asyncio event loop in a separate thread is useful for building responsive

GUIs, but also is useful for any synchronous legacy applications where coroutines and asyncio don’t fit smoothly.

We’ve now seen how to utilize threads for various I/O-bound workloads, but what about CPU-bound workloads? Recall that the GIL prevents us from running Python bytecode concurrently in threads, but there are a few notable exceptions to this that let us do some CPU-bound work in threads.

ing threads for CPU-bound work

The global interpreter lock is a tricky subject in Python. The rule of thumb is multithreading only makes sense for blocking I/O work, as I/O will release the GIL. This is true in most cases but not all. To properly release the GIL and avoid any concurrency bugs, the code that is running needs to avoid interacting with Python objects (dictionaries, lists, Python integers, and so on). This can happen when a large portion of our libraries’ work is done in low-level C code. There are a few notable libraries, such as hashlib and NumPy, that perform CPU-intensive work in pure C and release the GIL. This enables us to use multithreading to improve the performance of certain CPU-bound workloads. We’ll examine two such instances: hashing sensitive text for security and solving a data analysis problem with NumPy.

Iultithreading with hashlib

In today’s world, security has never been more important. Ensuring that data is not read by hackers is key to avoiding leaking sensitive customer data, such as passwords or other information that can be used to identify or harm them.

Hashing algorithms solve this problem by taking a piece of input data and creating a new piece of data that is unreadable and unrecoverable (if the algorithm is secure) to