Item 52: Use `subprocess` to Manage Child Processes

Python has battle-hardened libraries for running and managing child processes. This makes it a great language for gluing together other tools, such as command-line utilities. When existing shell scripts get complicated, as they often do over time, graduating them to a rewrite in Python for the sake of readability and maintainability is a natural choice.

Child processes started by Python are able to run in parallel, enabling you to use Python to consume all of the CPU cores of a machine and maximize the throughput of programs. Although Python itself may be CPU bound (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"), it's easy to use Python to drive and coordinate CPU-intensive workloads.

Python has many ways to run subprocesses (e.g., `os.popen`, `os.exec*`), but the best choice for managing child processes is to use the `subprocess` built-in module. Running a child process with `subprocess` is simple. Here, I use the module's `run` convenience function to start a process, read its output, and verify that it terminated cleanly:

```
import subprocess

result = subprocess.run(
    ['echo', 'Hello from the child!'],
    capture_output=True,
    encoding='utf-8')

result.check_returncode()  # No exception means clean exit
print(result.stdout)
```

```
>>>
Hello from the child!
```

Note

The examples in this item assume that your system has the `echo`, `sleep`, and `openssl` commands available. On Windows, this may not be the case. Please refer to the full example code for this item to see specific directions on how to run these snippets on Windows.

Child processes run independently from their parent process, the Python interpreter. If I create a subprocess using the `Popen` class instead of the `run` function, I can poll child process status periodically while Python does other work:

```
proc = subprocess.Popen(['sleep', '1'])
while proc.poll() is None:
    print('Working...')
    # Some time-consuming work here
    ...

print('Exit status', proc.poll())
```

```
>>>
Working...
Working...
Working...
Working...
Exit status 0
```

Decoupling the child process from the parent frees up the parent process to run many child processes in parallel. Here, I do this by starting all the child processes together with `Popen` upfront:

```
import time

start = time.time()
sleep_procs = []
for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)
```

Later, I wait for them to finish their I/O and terminate with the `communicate` method:

```
for proc in sleep_procs:
    proc.communicate()

end = time.time()
delta = end - start
print(f'Finished in {delta:.3} seconds')
```

```
>>>
Finished in 1.05 seconds
```

If these processes ran in sequence, the total delay would be 10 seconds or more rather than the ~1 second that I measured.

You can also pipe data from a Python program into a subprocess and retrieve its output. This allows you to utilize many other programs to do work in parallel. For example, say that I want to use the `openssl` command-line tool to encrypt some data. Starting the child process with command-line arguments and I/O pipes is easy:

```
import os
def run_encrypt(data):
    env = os.environ.copy()
    env['password'] = 'zf7ShyBhZOraQDdE/FiZpm/m/8f9X+M1'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush()  # Ensure that the child gets input
    return proc
```

Here, I pipe random bytes into the encryption function, but in practice this input pipe would be fed data from user input, a file handle, a network socket, and so on:

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_encrypt(data)
    procs.append(proc)
```

The child processes run in parallel and consume their input. Here, I wait for them to finish and then retrieve their final output. The output is random encrypted bytes as expected:

```
for proc in procs:
    out, _ = proc.communicate()
    print(out[-10:])
```

```
>>>
b'\x8c(\xed\xc7m1\xf0F4\xe6'
b'\x0eD\x97\xe9>\x10h{\xbd\xf0'
b'g\x93)\x14U\xa9\xdc\xdd\x04\xd2'
```

It’s also possible to create chains of parallel processes, just like UNIX pipelines, connecting the output of one child process to the input of another, and so on. Here’s a function that starts the `openssl` command-line tool as a subprocess to generate a Whirlpool hash of the input stream:

```
def run_hash(input_stdin):
    return subprocess.Popen(
        ['openssl', 'dgst', '-whirlpool', '-binary'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
```

Now, I can kick off one set of processes to encrypt some data and another set of processes to subsequently hash their encrypted output. Note that I have to be careful with how the `stdout` instance of the upstream process is retained by the Python interpreter process that's starting this pipeline of child processes:

```
encrypt_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(100)

    encrypt_proc = run_encrypt(data)
    encrypt_procs.append(encrypt_proc)

    hash_proc = run_hash(encrypt_proc.stdout)
    hash_procs.append(hash_proc)

    # Ensure that the child consumes the input stream and
    # the communicate() method doesn't inadvertently steal
    # input from the child. Also lets SIGPIPE propagate to
    # the upstream process if the downstream process dies.
    encrypt_proc.stdout.close()
    encrypt_proc.stdout = None
```

The I/O between the child processes happens automatically once they are started. All I need to do is wait for them to finish and print the final output:

```
for proc in encrypt_procs:
    proc.communicate()
    assert proc.returncode == 0

for proc in hash_procs:
    out, _ = proc.communicate()
    print(out[-10:])
    assert proc.returncode == 0

>>>
b'\xe2j\x98h\xfd\xec\xe7T\xd84'
b'\xf3.i\x01\xd74|\xf2\x94E'
b'5_n\xc3-\xe6j\xeb[i'
```

If I'm worried about the child processes never finishing or somehow blocking on input or output pipes, I can pass the `timeout` parameter to the `communicate` method. This causes an exception to be raised if the child process hasn't finished within the time period, giving me a chance to terminate the misbehaving subprocess:

```
proc = subprocess.Popen(['sleep', '10'])
try:
    proc.communicate(timeout=0.1)
```

```
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Exit status', proc.poll())
```

```
>>>
Exit status -15
```

Things to Remember

✦ Use the `subprocess` module to run child processes and manage their input and output streams.

✦ Child processes run in parallel with the Python interpreter, enabling you to maximize your usage of CPU cores.

✦ Use the `run` convenience function for simple usage, and the `Popen` class for advanced usage like UNIX-style pipelines.

✦ Use the `timeout` parameter of the `communicate` method to avoid deadlocks and hanging child processes.

Item 53: Use Threads for Blocking I/O, Avoid for Parallelism

The standard implementation of Python is called CPython. CPython runs a Python program in two steps. First, it parses and compiles the source text into *bytecode*, which is a low-level representation of the program as 8-bit instructions. (As of Python 3.6, however, it's technically *wordcode* with 16-bit instructions, but the idea is the same.) Then, CPython runs the bytecode using a stack-based interpreter. The bytecode interpreter has state that must be maintained and coherent while the Python program executes. CPython enforces coherence with a mechanism called the *global interpreter lock* (GIL).

Essentially, the GIL is a mutual-exclusion lock (mutex) that prevents CPython from being affected by preemptive multithreading, where one thread takes control of a program by interrupting another thread. Such an interruption could corrupt the interpreter state (e.g., garbage collection reference counts) if it comes at an unexpected time. The GIL prevents these interruptions and ensures that every bytecode instruction works correctly with the CPython implementation and its C-extension modules.

The GIL has an important negative side effect. With programs written in languages like C++ or Java, having multiple threads of execution means that a program could utilize multiple CPU cores at the same time. Although Python supports multiple threads of execution, the GIL causes only one of them to ever make forward progress at a time. This means that when you reach for threads to do parallel computation and speed up your Python programs, you will be sorely disappointed.

For example, say that I want to do something computationally intensive with Python. Here, I use a naive number factorization algorithm as a proxy:

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

Factoring a set of numbers in serial takes quite a long time:

```
import time

numbers = [2139079, 1214759, 1516637, 1852285]
start = time.time()

for number in numbers:
    list(factorize(number))

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.399 seconds
```

Using multiple threads to do this computation would make sense in other languages because I could take advantage of all the CPU cores of my computer. Let me try that in Python. Here, I define a Python thread for doing the same computation as before:

```
from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number

    def run(self):
        self.factors = list(factorize(self.number))
```

Then, I start a thread for each number to factorize in parallel:

```
start = time.time()

threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)
```

Finally, I wait for all of the threads to finish:

```
for thread in threads:
    thread.join()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')
```

```
>>>
Took 0.446 seconds
```

Surprisingly, this takes even longer than running `factorize` in serial. With one thread per number, you might expect less than a 4x speedup in other languages due to the overhead of creating threads and coordinating with them. You might expect only a 2x speedup on the dual-core machine I used to run this code. But you wouldn't expect the performance of these threads to be worse when there are multiple CPUs to utilize. This demonstrates the effect of the GIL (e.g., lock contention and scheduling overhead) on programs running in the standard CPython interpreter.

There are ways to get CPython to utilize multiple cores, but they don't work with the standard `Thread` class (see Item 64: "Consider `concurrent.futures` for True Parallelism"), and they can require substantial effort. Given these limitations, why does Python support threads at all? There are two good reasons.

First, multiple threads make it easy for a program to seem like it's doing multiple things at the same time. Managing the juggling act of simultaneous tasks is difficult to implement yourself (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for an example). With threads, you can leave it to Python to run your functions concurrently. This works because CPython ensures a level of fairness between Python threads of execution, even though only one of them makes forward progress at a time due to the GIL.

The second reason Python supports threads is to deal with blocking I/O, which happens when Python does certain types of system calls.

A Python program uses system calls to ask the computer's operating system to interact with the external environment on its behalf. Blocking I/O includes things like reading and writing files, interacting with networks, communicating with devices like displays, and so on. Threads help handle blocking I/O by insulating a program from the time it takes for the operating system to respond to requests.

For example, say that I want to send a signal to a remote-controlled helicopter through a serial port. I'll use a slow system call (`select`) as a proxy for this activity. This function asks the operating system to block for 0.1 seconds and then return control to my program, which is similar to what would happen when using a synchronous serial port:

```
import select
import socket

def slow_systemcall():
    select.select([socket.socket()], [], [], 0.1)
```

Running this system call in serial requires a linearly increasing amount of time:

```
start = time.time()

for _ in range(5):
    slow_systemcall()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.510 seconds
```

The problem is that while the `slow_systemcall` function is running, my program can't make any other progress. My program's main thread of execution is blocked on the `select` system call. This situation is awful in practice. You need to be able to compute your helicopter's next move while you're sending it a signal; otherwise, it'll crash. When you find yourself needing to do blocking I/O and computation simultaneously, it's time to consider moving your system calls to threads.

Here, I run multiple invocations of the `slow_systemcall` function in separate threads. This would allow me to communicate with multiple serial ports (and helicopters) at the same time while leaving the main thread to do whatever computation is required:

```
start = time.time()
```

```
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

With the threads started, here I do some work to calculate the next helicopter move before waiting for the system call threads to finish:

```
def compute_helicopter_location(index):
    ...

for i in range(5):
    compute_helicopter_location(i)

for thread in threads:
    thread.join()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')
```

```
>>>
Took 0.108 seconds
```

The parallel time is ~5x less than the serial time. This shows that all the system calls will run in parallel from multiple Python threads even though they're limited by the GIL. The GIL prevents my Python code from running in parallel, but it doesn't have an effect on system calls. This works because Python threads release the GIL just before they make system calls, and they reacquire the GIL as soon as the system calls are done.

There are many other ways to deal with blocking I/O besides using threads, such as the `asyncio` built-in module, and these alternatives have important benefits. But those options might require extra work in refactoring your code to fit a different model of execution (see Item 60: "Achieve Highly Concurrent I/O with Coroutines" and Item 62: "Mix Threads and Coroutines to Ease the Transition to `asyncio`"). Using threads is the simplest way to do blocking I/O in parallel with minimal changes to your program.

Things to Remember

✦ Python threads can't run in parallel on multiple CPU cores because of the global interpreter lock (GIL).

- ✦ Python threads are still useful despite the GIL because they provide an easy way to do multiple things seemingly at the same time.
- ✦ Use Python threads to make multiple system calls in parallel. This allows you to do blocking I/O at the same time as computation.

Item 54: Use Lock to Prevent Data Races in Threads

After learning about the global interpreter lock (GIL) (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"), many new Python programmers assume they can forgo using mutual-exclusion locks (also called *mutexes*) in their code altogether. If the GIL is already preventing Python threads from running on multiple CPU cores in parallel, it must also act as a lock for a program's data structures, right? Some testing on types like lists and dictionaries may even show that this assumption appears to hold.

But beware, this is not truly the case. The GIL will not protect you. Although only one Python thread runs at a time, a thread's operations on data structures can be interrupted between any two bytecode instructions in the Python interpreter. This is dangerous if you access the same objects from multiple threads simultaneously. The invariants of your data structures could be violated at practically any time because of these interruptions, leaving your program in a corrupted state.

For example, say that I want to write a program that counts many things in parallel, like sampling light levels from a whole network of sensors. If I want to determine the total number of light samples over time, I can aggregate them with a new class:

```
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

Imagine that each sensor has its own worker thread because reading from the sensor requires blocking I/O. After each sensor measurement, the worker thread increments the counter up to a maximum number of desired readings:

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # Read from the sensor
        ...
        counter.increment(1)
```

Here, I run one worker thread for each sensor in parallel and wait for them all to finish their readings:

```
from threading import Thread

how_many = 10**5
counter = Counter()

threads = []
for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')
```

```
>>>
Counter should be 500000, got 246760
```

This seemed straightforward, and the outcome should have been obvious, but the result is way off! What happened here? How could something so simple go so wrong, especially since only one Python interpreter thread can run at a time?

The Python interpreter enforces fairness between all of the threads that are executing to ensure they get roughly equal processing time. To do this, Python suspends a thread as it's running and resumes another thread in turn. The problem is that you don't know exactly when Python will suspend your threads. A thread can even be paused seemingly halfway through what looks like an atomic operation. That's what happened in this case.

The body of the `Counter` object's `increment` method looks simple, and is equivalent to this statement from the perspective of the worker thread:

```
counter.count += 1
```

But the += operator used on an object attribute actually instructs Python to do three separate operations behind the scenes. The statement above is equivalent to this:

```
value = getattr(counter, 'count')
result = value + 1
setattr(counter, 'count', result)
```

Python threads incrementing the counter can be suspended between any two of these operations. This is problematic if the way the operations interleave causes old versions of value to be assigned to the counter. Here's an example of bad interaction between two threads, A and B:

```
# Running in Thread A
value_a = getattr(counter, 'count')
# Context switch to Thread B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Context switch back to Thread A
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

Thread B interrupted thread A before it had completely finished. Thread B ran and finished, but then thread A resumed mid-execution, overwriting all of thread B's progress in incrementing the counter. This is exactly what happened in the light sensor example above.

To prevent data races like these, and other forms of data structure corruption, Python includes a robust set of tools in the threading built-in module. The simplest and most useful of them is the Lock class, a mutual-exclusion lock (mutex).

By using a lock, I can have the Counter class protect its current value against simultaneous accesses from multiple threads. Only one thread will be able to acquire the lock at a time. Here, I use a with statement to acquire and release the lock; this makes it easier to see which code is executing while the lock is held (see Item 66: "Consider contextlib and with Statements for Reusable try/finally Behavior" for background):

```
from threading import Lock

class LockingCounter:
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

Now, I run the worker threads as before but use a LockingCounter instead:

```
counter = LockingCounter()

for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')
```

```
>>>
Counter should be 500000, got 500000
```

The result is exactly what I expect. Lock solved the problem.

Things to Remember

- ✦ Even though Python has a global interpreter lock, you're still responsible for protecting against data races between the threads in your programs.
- ✦ Your programs will corrupt their data structures if you allow multiple threads to modify the same objects without mutual-exclusion locks (mutexes).
- ✦ Use the Lock class from the threading built-in module to enforce your program's invariants between multiple threads.

Item 55: Use Queue to Coordinate Work Between Threads

Python programs that do many things concurrently often need to coordinate their work. One of the most useful arrangements for concurrent work is a pipeline of functions.

A pipeline works like an assembly line used in manufacturing. Pipelines have many phases in serial, with a specific function for each phase. New pieces of work are constantly being added to the beginning of the pipeline. The functions can operate concurrently, each working on the piece of work in its phase. The work moves forward as each function completes until there are no phases remaining. This approach is especially good for work that includes blocking I/O or subprocesses—activities that can easily be parallelized using Python (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism").

For example, say I want to build a system that will take a constant stream of images from my digital camera, resize them, and then add them to a photo gallery online. Such a program could be split into three phases of a pipeline. New images are retrieved in the first phase. The downloaded images are passed through the resize function in the second phase. The resized images are consumed by the upload function in the final phase.

Imagine that I've already written Python functions that execute the phases: `download`, `resize`, `upload`. How do I assemble a pipeline to do the work concurrently?

```
def download(item):
    ...

def resize(item):
    ...

def upload(item):
    ...
```

The first thing I need is a way to hand off work between the pipeline phases. This can be modeled as a thread-safe producer–consumer queue (see Item 54: "Use `Lock` to Prevent Data Races in Threads" to understand the importance of thread safety in Python; see Item 71: "Prefer `deque` for Producer–Consumer Queues" to understand queue performance):

```
from collections import deque
from threading import Lock

class MyQueue:
    def __init__(self):
        self.items = deque()
        self.lock = Lock()
```

The producer, my digital camera, adds new images to the end of the deque of pending items:

```
    def put(self, item):
        with self.lock:
            self.items.append(item)
```

The consumer, the first phase of the processing pipeline, removes images from the front of the `deque` of pending items:

```
    def get(self):
        with self.lock:
            return self.items.popleft()
```

Here, I represent each phase of the pipeline as a Python thread that takes work from one queue like this, runs a function on it, and puts the result on another queue. I also track how many times the worker has checked for new input and how much work it's completed:

```
from threading import Thread
import time

class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

The trickiest part is that the worker thread must properly handle the case where the input queue is empty because the previous phase hasn't completed its work yet. This happens where I catch the `IndexError` exception below. You can think of this as a holdup in the assembly line:

```
    def run(self):
        while True:
            self.polled_count += 1
            try:
                item = self.in_queue.get()
            except IndexError:
                time.sleep(0.01)  # No work to do
            else:
                result = self.func(item)
                self.out_queue.put(result)
                self.work_done += 1
```

Now, I can connect the three phases together by creating the queues for their coordination points and the corresponding worker threads:

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
```

```
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

I can start the threads and then inject a bunch of work into the first phase of the pipeline. Here, I use a plain `object` instance as a proxy for the real data required by the `download` function:

```
for thread in threads:
    thread.start()

for _ in range(1000):
    download_queue.put(object())
```

Now, I wait for all of the items to be processed by the pipeline and end up in the `done_queue`:

```
while len(done_queue.items) < 1000:
    # Do something useful while waiting
    ...
```

This runs properly, but there's an interesting side effect caused by the threads polling their input queues for new work. The tricky part, where I catch `IndexError` exceptions in the `run` method, executes a large number of times:

```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print(f'Processed {processed} items after '
      f'polling {polled} times')

>>>
Processed 1000 items after polling 3035 times
```

When the worker functions vary in their respective speeds, an earlier phase can prevent progress in later phases, backing up the pipeline. This causes later phases to starve and constantly check their input queues for new work in a tight loop. The outcome is that worker threads waste CPU time doing nothing useful; they're constantly raising and catching `IndexError` exceptions.

But that's just the beginning of what's wrong with this implementation. There are three more problems that you should also avoid. First, determining that all of the input work is complete requires yet another busy wait on the `done_queue`. Second, in `Worker`, the `run` method will execute forever in its busy loop. There's no obvious way to signal to a worker thread that it's time to exit.

Third, and worst of all, a backup in the pipeline can cause the program to crash arbitrarily. If the first phase makes rapid progress but the second phase makes slow progress, then the queue connecting the first phase to the second phase will constantly increase in size. The second phase won't be able to keep up. Given enough time and input data, the program will eventually run out of memory and die.

The lesson here isn't that pipelines are bad; it's that it's hard to build a good producer–consumer queue yourself. So why even try?

Queue to the Rescue

The `Queue` class from the `queue` built-in module provides all of the functionality you need to solve the problems outlined above.

`Queue` eliminates the busy waiting in the worker by making the `get` method block until new data is available. For example, here I start a thread that waits for some input data on a queue:

```
from queue import Queue

my_queue = Queue()

def consumer():
    print('Consumer waiting')
    my_queue.get()              # Runs after put() below
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

Even though the thread is running first, it won't finish until an item is put on the `Queue` instance and the `get` method has something to return:

```
print('Producer putting')
my_queue.put(object())          # Runs before get() above
print('Producer done')
thread.join()

>>>
Consumer waiting
Producer putting
Producer done
Consumer done
```

To solve the pipeline backup issue, the `Queue` class lets you specify the maximum amount of pending work to allow between two phases.

This buffer size causes calls to put to block when the queue is already full. For example, here I define a thread that waits for a while before consuming a queue:

```
my_queue = Queue(1)             # Buffer size of 1

def consumer():
    time.sleep(0.1)             # Wait
    my_queue.get()              # Runs second
    print('Consumer got 1')
    my_queue.get()              # Runs fourth
    print('Consumer got 2')
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

The wait should allow the producer thread to put both objects on the queue before the consumer thread ever calls get. But the Queue size is one. This means the producer adding items to the queue will have to wait for the consumer thread to call get at least once before the second call to put will stop blocking and add the second item to the queue:

```
my_queue.put(object())          # Runs first
print('Producer put 1')
my_queue.put(object())          # Runs third
print('Producer put 2')
print('Producer done')
thread.join()
```

```
>>>
Producer put 1
Consumer got 1
Producer put 2
Producer done
Consumer got 2
Consumer done
```

The Queue class can also track the progress of work using the task_done method. This lets you wait for a phase's input queue to drain and eliminates the need to poll the last phase of a pipeline (as with the done_queue above). For example, here I define a consumer thread that calls task_done when it finishes working on an item:

```
in_queue = Queue()
```

```
def consumer():
    print('Consumer waiting')
    work = in_queue.get()       # Runs second
    print('Consumer working')
    # Doing work
    ...
    print('Consumer done')
    in_queue.task_done()        # Runs third

thread = Thread(target=consumer)
thread.start()
```

Now, the producer code doesn't have to join the consumer thread or poll. The producer can just wait for the `in_queue` to finish by calling `join` on the `Queue` instance. Even once it's empty, the `in_queue` won't be joinable until after `task_done` is called for every item that was ever enqueued:

```
print('Producer putting')
in_queue.put(object())        # Runs first
print('Producer waiting')
in_queue.join()               # Runs fourth
print('Producer done')
thread.join()
```

```
>>>
Consumer waiting
Producer putting
Producer waiting
Consumer working
Consumer done
Producer done
```

I can put all these behaviors together into a `Queue` subclass that also tells the worker thread when it should stop processing. Here, I define a `close` method that adds a special *sentinel* item to the queue that indicates there will be no more input items after it:

```
class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)
```

Then, I define an iterator for the queue that looks for this special object and stops iteration when it's found. This `__iter__` method also calls `task_done` at appropriate times, letting me track the progress of work on the queue (see Item 31: "Be Defensive When Iterating Over Arguments" for details about __iter__):

```
    def __iter__(self):
        while True:
            item = self.get()
            try:
                if item is self.SENTINEL:
                    return  # Cause the thread to exit
                yield item
            finally:
                self.task_done()
```

Now, I can redefine my worker thread to rely on the behavior of the ClosableQueue class. The thread will exit when the for loop is exhausted:

```
class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)
```

I re-create the set of worker threads using the new worker class:

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    StoppableWorker(resize, resize_queue, upload_queue),
    StoppableWorker(upload, upload_queue, done_queue),
]
```

After running the worker threads as before, I also send the stop signal after all the input work has been injected by closing the input queue of the first phase:

```
for thread in threads:
    thread.start()
```

```
for _ in range(1000):
    download_queue.put(object())

download_queue.close()
```

Finally, I wait for the work to finish by joining the queues that connect the phases. Each time one phase is done, I signal the next phase to stop by closing its input queue. At the end, the `done_queue` contains all of the output objects, as expected:

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')

for thread in threads:
    thread.join()
```

```
>>>
1000 items finished
```

This approach can be extended to use multiple worker threads per phase, which can increase I/O parallelism and speed up this type of program significantly. To do this, first I define some helper functions that start and stop multiple threads. The way `stop_threads` works is by calling `close` on each input queue once per consuming thread, which ensures that all of the workers exit cleanly:

```
def start_threads(count, *args):
    threads = [StoppableWorker(*args) for _ in range(count)]
    for thread in threads:
        thread.start()
    return threads

def stop_threads(closable_queue, threads):
    for _ in threads:
        closable_queue.close()

    closable_queue.join()

    for thread in threads:
        thread.join()
```

Then, I connect the pieces together as before, putting objects to process into the top of the pipeline, joining queues and threads along the way, and finally consuming the results:

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()

download_threads = start_threads(
    3, download, download_queue, resize_queue)
resize_threads = start_threads(
    4, resize, resize_queue, upload_queue)
upload_threads = start_threads(
    5, upload, upload_queue, done_queue)

for _ in range(1000):
    download_queue.put(object())

stop_threads(download_queue, download_threads)
stop_threads(resize_queue, resize_threads)
stop_threads(upload_queue, upload_threads)

print(done_queue.qsize(), 'items finished')
```

```
>>>
1000 items finished
```

Although `Queue` works well in this case of a linear pipeline, there are many other situations for which there are better tools that you should consider (see Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- ✦ Pipelines are a great way to organize sequences of work—especially I/O-bound programs—that run concurrently using multiple Python threads.
- ✦ Be aware of the many problems in building concurrent pipelines: busy waiting, how to tell workers to stop, and potential memory explosion.
- ✦ The `Queue` class has all the facilities you need to build robust pipelines: blocking operations, buffer sizes, and joining.

Item 56: Know How to Recognize When Concurrency Is Necessary

Inevitably, as the scope of a program grows, it also becomes more complicated. Dealing with expanding requirements in a way that maintains clarity, testability, and efficiency is one of the most difficult parts of programming. Perhaps the hardest type of change to handle is moving from a single-threaded program to one that needs multiple concurrent lines of execution.

Let me demonstrate how you might encounter this problem with an example. Say that I want to implement Conway's Game of Life, a classic illustration of finite state automata. The rules of the game are simple: You have a two-dimensional grid of an arbitrary size. Each cell in the grid can either be alive or empty:

```
ALIVE = '*'
EMPTY = '-'
```

The game progresses one tick of the clock at a time. Every tick, each cell counts how many of its neighboring eight cells are still alive. Based on its neighbor count, a cell decides if it will keep living, die, or regenerate. (I'll explain the specific rules further below.) Here's an example of a 5 × 5 Game of Life grid after four generations with time going to the right:

```
  0   |   1   |   2   |   3   |   4
----- | ----- | ----- | ----- | -----
-*--- | --*-- | --**- | --*-- | -----
--**- | --**- | -*--- | -*--- | -**--
---*- | --**- | --**- | --*-- | -----
----- | ----- | ----- | ----- | -----
```

I can represent the state of each cell with a simple container class. The class must have methods that allow me to get and set the value of any coordinate. Coordinates that are out of bounds should wrap around, making the grid act like an infinite looping space:

```
class Grid:
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)
    def get(self, y, x):
        return self.rows[y % self.height][x % self.width]

    def set(self, y, x, state):
        self.rows[y % self.height][x % self.width] = state

    def __str__(self):
        ...
```

To see this class in action, I can create a Grid instance and set its initial state to a classic shape called a glider:

```
grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
print(grid)
```

```
>>>
---*-----
----*----
--***----
---------
---------
```

Now, I need a way to retrieve the status of neighboring cells. I can do this with a helper function that queries the grid and returns the count of living neighbors. I use a simple function for the get parameter instead of passing in a whole Grid instance in order to reduce coupling (see Item 38: "Accept Functions Instead of Classes for Simple Interfaces" for more about this approach):

```
def count_neighbors(y, x, get):
    n_ = get(y - 1, x + 0)  # North
    ne = get(y - 1, x + 1)  # Northeast
    e_ = get(y + 0, x + 1)  # East
    se = get(y + 1, x + 1)  # Southeast
    s_ = get(y + 1, x + 0)  # South
    sw = get(y + 1, x - 1)  # Southwest
    w_ = get(y + 0, x - 1)  # West
    nw = get(y - 1, x - 1)  # Northwest
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Now, I define the simple logic for Conway's Game of Life, based on the game's three rules: Die if a cell has fewer than two neighbors, die if a cell has more than three neighbors, or become alive if an empty cell has exactly three neighbors:

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY     # Die: Too few
        elif neighbors > 3:
            return EMPTY     # Die: Too many
    else:
        if neighbors == 3:
            return ALIVE     # Regenerate
    return state
```

I can connect `count_neighbors` and `game_logic` together in another function that transitions the state of a cell. This function will be called each generation to figure out a cell's current state, inspect the neighboring cells around it, determine what its next state should be, and update the resulting grid accordingly. Again, I use a function interface for `set` instead of passing in the `Grid` instance to make this code more decoupled:

```
def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)
```

Finally, I can define a function that progresses the whole grid of cells forward by a single step and then returns a new grid containing the state for the next generation. The important detail here is that I need all dependent functions to call the `get` method on the previous generation's `Grid` instance, and to call the `set` method on the next generation's `Grid` instance. This is how I ensure that all of the cells move in lockstep, which is an essential part of how the game works. This is easy to achieve because I used function interfaces for `get` and `set` instead of passing `Grid` instances:

```
def simulate(grid):
    next_grid = Grid(grid.height, grid.width)
    for y in range(grid.height):
        for x in range(grid.width):
            step_cell(y, x, grid.get, next_grid.set)
    return next_grid
```

Now, I can progress the grid forward one generation at a time. You can see how the glider moves down and to the right on the grid based on the simple rules from the `game_logic` function:

```
class ColumnPrinter:
    ...

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate(grid)

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

This works great for a program that can run in one thread on a single machine. But imagine that the program's requirements have changed—as I alluded to above—and now I need to do some I/O (e.g., with a socket) from within the `game_logic` function. For example, this might be required if I'm trying to build a massively multiplayer online game where the state transitions are determined by a combination of the grid state and communication with other players over the Internet.

How can I extend this implementation to support such functionality? The simplest thing to do is to add blocking I/O directly into the `game_logic` function:

```
def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...
```

The problem with this approach is that it's going to slow down the whole program. If the latency of the I/O required is 100 milliseconds (i.e., a reasonably good cross-country, round-trip latency on the

Internet), and there are 45 cells in the grid, then each generation will take a minimum of 4.5 seconds to evaluate because each cell is processed serially in the `simulate` function. That's far too slow and will make the game unplayable. It also scales poorly: If I later wanted to expand the grid to 10,000 cells, I would need over 15 minutes to evaluate each generation.

The solution is to do the I/O in parallel so each generation takes roughly 100 milliseconds, regardless of how big the grid is. The process of spawning a concurrent line of execution for each unit of work—a cell in this case—is called *fan-out*. Waiting for all of those concurrent units of work to finish before moving on to the next phase in a coordinated process—a generation in this case—is called *fan-in*.

Python provides many built-in tools for achieving fan-out and fan-in with various trade-offs. You should understand the pros and cons of each approach and choose the best tool for the job, depending on the situation. See the items that follow for details based on this Game of Life example program (Item 57: "Avoid Creating New `Thread` Instances for On-demand Fan-out," Item 58: "Understand How Using `Queue` for Concurrency Requires Refactoring," Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency," and Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- ✦ A program often grows to require multiple concurrent lines of execution as its scope and complexity increases.
- ✦ The most common types of concurrency coordination are fan-out (generating new units of concurrency) and fan-in (waiting for existing units of concurrency to complete).
- ✦ Python has many different ways of achieving fan-out and fan-in.

Item 57: Avoid Creating New `Thread` Instances for On-demand Fan-out

Threads are the natural first tool to reach for in order to do parallel I/O in Python (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"). However, they have significant downsides when you try to use them for fanning out to many concurrent lines of execution.

To demonstrate this, I'll continue with the Game of Life example from before (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for background and the implementations of various functions and classes below). I'll use threads to solve the latency problem caused by doing I/O in the game_logic function. To begin, threads require coordination using locks to ensure that assumptions within data structures are maintained properly. I can create a subclass of the Grid class that adds locking behavior so an instance can be used by multiple threads simultaneously:

```
from threading import Lock

ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

class LockingGrid(Grid):
    def __init__(self, height, width):
        super().__init__(height, width)
        self.lock = Lock()

    def __str__(self):
        with self.lock:
            return super().__str__()

    def get(self, y, x):
        with self.lock:
            return super().get(y, x)

    def set(self, y, x, state):
        with self.lock:
            return super().set(y, x, state)
```

Then, I can reimplement the simulate function to *fan out* by creating a thread for each call to step_cell. The threads will run in parallel and won't have to wait on each other's I/O. I can then *fan in* by waiting for all of the threads to complete before moving on to the next generation:

```
from threading import Thread

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...
```

```
def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)

def simulate_threaded(grid):
    next_grid = LockingGrid(grid.height, grid.width)

    threads = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            thread = Thread(target=step_cell, args=args)
            thread.start()  # Fan out
            threads.append(thread)

    for thread in threads:
        thread.join()       # Fan in

    return next_grid
```

I can run this code using the same implementation of `step_cell` and the same driving code as before with only two lines changed to use the `LockingGrid` and `simulate_threaded` implementations:

```
class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)            # Changed
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_threaded(grid)  # Changed

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

This works as expected, and the I/O is now parallelized between the threads. However, this code has three big problems:

- The Thread instances require special tools to coordinate with each other safely (see Item 54: "Use Lock to Prevent Data Races in Threads"). This makes the code that uses threads harder to reason about than the procedural, single-threaded code from before. This complexity makes threaded code more difficult to extend and maintain over time.
- Threads require a lot of memory—about 8 MB per executing thread. On many computers, that amount of memory doesn't matter for the 45 threads I'd need in this example. But if the game grid had to grow to 10,000 cells, I would need to create that many threads, which couldn't even fit in the memory of my machine. Running a thread per concurrent activity just won't work.
- Starting a thread is costly, and threads have a negative performance impact when they run due to context switching between them. In this case, all of the threads are started and stopped each generation of the game, which has high overhead and will increase latency beyond the expected I/O time of 100 milliseconds.

This code would also be very difficult to debug if something went wrong. For example, imagine that the game_logic function raises an exception, which is highly likely due to the generally flaky nature of I/O:

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem with I/O')
    ...
```

I can test what this would do by running a Thread instance pointed at this function and redirecting the sys.stderr output from the program to an in-memory StringIO buffer:

```
import contextlib
import io
```

```
fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    thread = Thread(target=game_logic, args=(ALIVE, 3))
    thread.start()
    thread.join()

print(fake_stderr.getvalue())
```

```
>>>
Exception in thread Thread-226:
Traceback (most recent call last):
  File "threading.py", line 917, in _bootstrap_inner
    self.run()
  File "threading.py", line 865, in run
    self._target(*self._args, **self._kwargs)
  File "example.py", line 193, in game_logic
    raise OSError('Problem with I/O')
OSError: Problem with I/O
```

An `OSError` exception is raised as expected, but somehow the code that created the `Thread` and called `join` on it is unaffected. How can this be? The reason is that the `Thread` class will independently catch any exceptions that are raised by the `target` function and then write their traceback to `sys.stderr`. Such exceptions are never re-raised to the caller that started the thread in the first place.

Given all of these issues, it's clear that threads are not the solution if you need to constantly create and finish new concurrent functions. Python provides other solutions that are a better fit (see Item 58: "Understand How Using `Queue` for Concurrency Requires Refactoring," Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency", and Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- ✦ Threads have many downsides: They're costly to start and run if you need a lot of them, they each require a significant amount of memory, and they require special tools like `Lock` instances for coordination.
- ✦ Threads do not provide a built-in way to raise exceptions back in the code that started a thread or that is waiting for one to finish, which makes them difficult to debug.

Item 58: Understand How Using Queue for Concurrency Requires Refactoring

In the previous item (see Item 57: "Avoid Creating New Thread Instances for On-demand Fan-out") I covered the downsides of using Thread to solve the parallel I/O problem in the Game of Life example from earlier (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for background and the implementations of various functions and classes below).

The next approach to try is to implement a threaded pipeline using the Queue class from the queue built-in module (see Item 55: "Use Queue to Coordinate Work Between Threads" for background; I rely on the implementations of ClosableQueue and StoppableWorker from that item in the example code below).

Here's the general approach: Instead of creating one thread per cell per generation of the Game of Life, I can create a fixed number of worker threads upfront and have them do parallelized I/O as needed. This will keep my resource usage under control and eliminate the overhead of frequently starting new threads.

To do this, I need two ClosableQueue instances to use for communicating to and from the worker threads that execute the game_logic function:

```
from queue import Queue

class ClosableQueue(Queue):
    ...

in_queue = ClosableQueue()
out_queue = ClosableQueue()
```

I can start multiple threads that will consume items from the in_queue, process them by calling game_logic, and put the results on out_queue. These threads will run concurrently, allowing for parallel I/O and reduced latency for each generation:

```
from threading import Thread

class StoppableWorker(Thread):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def game_logic_thread(item):
    y, x, state, neighbors = item
    try:
        next_state = game_logic(state, neighbors)
    except Exception as e:
        next_state = e
    return (y, x, next_state)

# Start the threads upfront
threads = []
for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, in_queue, out_queue)
    thread.start()
    threads.append(thread)
```

Now, I can redefine the simulate function to interact with these queues to request state transition decisions and receive corresponding responses. Adding items to in_queue causes *fan-out*, and consuming items from out_queue until it's empty causes *fan-in*:

```
ALIVE = '*'
EMPTY = '-'

class SimulationError(Exception):
    pass

class Grid:
    ...

def count_neighbors(y, x, get):
    ...

def simulate_pipeline(grid, in_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            neighbors = count_neighbors(y, x, grid.get)
            in_queue.put((y, x, state, neighbors))  # Fan out

    in_queue.join()
    out_queue.close()
    next_grid = Grid(grid.height, grid.width)
    for item in out_queue:                              # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid
```

The calls to `Grid.get` and `Grid.set` both happen within this new `simulate_pipeline` function, which means I can use the single-threaded implementation of `Grid` instead of the implementation that requires `Lock` instances for synchronization.

This code is also easier to debug than the `Thread` approach used in the previous item. If an exception occurs while doing I/O in the `game_logic` function, it will be caught, propagated to the `out_queue`, and then re-raised in the main thread:

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem with I/O in game_logic')
    ...

simulate_pipeline(Grid(1, 1), in_queue, out_queue)
```

```
>>>
Traceback ...
OSError: Problem with I/O in game_logic

The above exception was the direct cause of the following
➥exception:

Traceback ...
SimulationError: (0, 0)
```

I can drive this multithreaded pipeline for repeated generations by calling `simulate_pipeline` in a loop:

```
class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_pipeline(grid, in_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
for thread in threads:
    thread.join()
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --------- | --*-*---- | --------- | ----*----
--***---- | --------- | ---**---- | --------- | --*-*----
--------- | --------- | ---*----- | --------- | ---**----
--------- | --------- | --------- | --------- | ---------
```

The results are the same as before. Although I've addressed the memory explosion problem, startup costs, and debugging issues of using threads on their own, many issues remain:

- The simulate_pipeline function is even harder to follow than the simulate_threaded approach from the previous item.
- Extra support classes were required for ClosableQueue and StoppableWorker in order to make the code easier to read, at the expense of increased complexity.
- I have to specify the amount of potential parallelism—the number of threads running game_logic_thread—upfront based on my expectations of the workload instead of having the system automatically scale up parallelism as needed.
- In order to enable debugging, I have to manually catch exceptions in worker threads, propagate them on a Queue, and then re-raise them in the main thread.

However, the biggest problem with this code is apparent if the requirements change again. Imagine that later I needed to do I/O within the count_neighbors function in addition to the I/O that was needed within game_logic:

```
def count_neighbors(y, x, get):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...
```

In order to make this parallelizable, I need to add another stage to the pipeline that runs count_neighbors in a thread. I need to make sure that exceptions propagate correctly between the worker threads and the main thread. And I need to use a Lock for the Grid class in order to ensure safe synchronization between the worker threads (see Item 54: "Use Lock to Prevent Data Races in Threads" for background and Item 57: "Avoid Creating New Thread Instances for On-demand Fan-out" for the implementation of LockingGrid):

```
def count_neighbors_thread(item):
    y, x, state, get = item
    try:
        neighbors = count_neighbors(y, x, get)
    except Exception as e:
        neighbors = e
    return (y, x, state, neighbors)

def game_logic_thread(item):
    y, x, state, neighbors = item
    if isinstance(neighbors, Exception):
        next_state = neighbors
    else:
        try:
            next_state = game_logic(state, neighbors)
        except Exception as e:
            next_state = e
    return (y, x, next_state)

class LockingGrid(Grid):
    ...
```

I have to create another set of Queue instances for the count_neighbors_thread workers and the corresponding Thread instances:

```
in_queue = ClosableQueue()
logic_queue = ClosableQueue()
out_queue = ClosableQueue()

threads = []
```

```
for _ in range(5):
    thread = StoppableWorker(
        count_neighbors_thread, in_queue, logic_queue)
    thread.start()
    threads.append(thread)

for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, logic_queue, out_queue)
    thread.start()
    threads.append(thread)
```

Finally, I need to update `simulate_pipeline` to coordinate the multiple phases in the pipeline and ensure that work fans out and back in correctly:

```
def simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            item = (y, x, state, grid.get)
            in_queue.put(item)          # Fan out

    in_queue.join()
    logic_queue.join()                  # Pipeline sequencing
    out_queue.close()

    next_grid = LockingGrid(grid.height, grid.width)
    for item in out_queue:              # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid
```

With these updated implementations, now I can run the multiphase pipeline end-to-end:

```
grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
```

```
columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
for thread in threads:
    logic_queue.close()
for thread in threads:
    thread.join()
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Again, this works as expected, but it required a lot of changes and boilerplate. The point here is that `Queue` does make it possible to solve fan-out and fan-in problems, but the overhead is very high. Although using `Queue` is a better approach than using `Thread` instances on their own, it's still not nearly as good as some of the other tools provided by Python (see Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency" and Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

✦ Using `Queue` instances with a fixed number of worker threads improves the scalability of fan-out and fan-in using threads.

✦ It takes a significant amount of work to refactor existing code to use `Queue`, especially when multiple stages of a pipeline are required.

✦ Using `Queue` fundamentally limits the total amount of I/O parallelism a program can leverage compared to alternative approaches provided by other built-in Python features and modules.