14. Threads and Processes - Python in a Nutshell, 3rd Edition

Chapter 14. Threads and Processes

A *thread* is a flow of control that shares global state (memory) with other threads; all threads appear to execute simultaneously, although they are usually "taking turns" on a single processor/core. Threads are not easy to master, and multithreaded programs are often hard to test, and to debug; however, as covered in "Use Threading, Multiprocessing, or async Programming?", when used appropriately, multithreading may sometimes improve program performance in comparison to traditional "single-threaded" programming. This chapter covers the facilities that Python provides for dealing with threads, including the `threading`, `queue`, and `concurrent` modules.

A *process* is an instance of a running program. The operating system protects processes from one another. Processes that want to communicate must explicitly arrange to do so via *inter-process communication* (IPC) mechanisms. Processes may communicate via files (covered in Chapter 10) and databases (covered in Chapter 11). The general way in which processes communicate using data storage mechanisms, such as files and databases, is that one process writes data, and another process later reads that data back. This chapter covers the Python standard library modules `subprocess` and `multiprocessing`; the process-related parts of the module `os`, including simple IPC by means of *pipes*; and a cross-platform IPC mechanism known as *memory-mapped files*, which is supplied to Python programs by the module `mmap`.

Network mechanisms are well suited for IPC, as they work between processes that run on different nodes of a network, not just between ones that run on the same node. `multiprocessing` supplies some mechanisms that are suitable for IPC over a network; Chapter 17 covers low-level network mechanisms that provide a basis for IPC. Other, higher-level mechanisms, known as *distributed computing*, such as CORBA, DCOM/COM+, EJB, SOAP, XML-RPC, and .NET, can make IPC easier, whether locally or remotely. We do not cover distributed computing in this book.

Threads in Python

Python offers multithreading on platforms that support threads, such as Win32, Linux, and other variants of Unix. An action is known as *atomic* when it's guaranteed that no thread switching occurs between the start and the end of the action. In practice, in CPython, operations that *look* atomic (e.g., simple assignments and accesses) mostly *are* atomic, when executed on built-in types (augmented and multiple assignments, however, aren't atomic). Mostly, though, it's *not* a good idea to rely on atomicity. You might be dealing with an instance of a user-coded class rather than of a built-in type, so there might be implicit calls to Python code, making assumptions of atomicity unwarranted. Relying on implementation-dependent atomicity may lock your code into a specific implementation, hampering future upgrades. You're better off using the synchronization facilities covered in the rest of this chapter, rather than relying on atomicity assumptions.

Python offers multithreading in two flavors. An older and lower-level module, `_thread` (named `thread` in v2), has low-level functionality and is not recommended for direct use in your code; we do not cover `_thread` in this book. The higher-level module `threading`, built on top of `_thread`, is the recommended one. The key design issue in multithreading systems is how best to coordinate multiple threads. `threading` supplies several synchronization objects. Alternatively, the `queue` module is very useful for thread synchronization, as it supplies synchronized, thread-safe queue types, handy for communication and coordination between threads. The package `concurrent`, covered after `multiprocessing`, supplies a unified interface for communication and coordination that can be implemented by pools of either threads or processes.

The threading Module

The `threading` module supplies multithreading functionality. The approach of `threading` is similar to Java's, but locks and conditions are modeled as separate objects (in Java, such functionality is part of every object), and threads cannot be directly controlled from the outside (thus, no priorities, groups, destruction, or stopping). All methods of objects supplied by `threading` are atomic.

`threading` supplies classes dealing with threads: `Thread`, `Condition`, `Event`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, and `Timer` (and, in v3 only, `Barrier`). `threading` also supplies functions, including:

active_count	`active_count()` Returns an `int`, the number of `Thread` objects currently alive (not ones that have terminated or not yet started).
current_thread	`current_thread()` Returns a `Thread` object for the calling thread. If the calling thread was not created by `threading`, `current_thread` creates and returns a semi-dummy `Thread` object with limited functionality.
enumerate	`enumerate()` Returns a `list` of all `Thread` objects currently alive (not ones that have terminated or not yet started).
stack_size	`stack_size([size])` Returns the stack size, in bytes, used for new threads; `0` means "the system's default." When you pass `size`, that's what going to be used for new threads created afterward (on platforms that allow setting threads' stack size); acceptable values for `size` are subject to platform-specific constraints, such as being at least `32768` (or an even higher minimum, on some platforms) and (on some platforms) being a multiple of `4096`. Passing `size` as `0` is always acceptable and means "use the system's default." When you pass a value for `size` that is not acceptable on the current platform, `stack_size` raises a `ValueError` exception.

Thread Objects

A `Thread` instance `t` models a thread. You can pass a function to be used as `t`'s main function as an argument when you create `t`, or you can subclass `Thread` and override the `run` method (you may also override `__init__` but should not override other methods). `t` is not yet ready to run when you create it; to make `t` ready (active), call `t.start()`. Once `t` is active, it terminates when its main function ends, either normally or by propagating an exception. A `Thread` `t` can be a *daemon*, meaning that Python can terminate even if `t` is still active, while a normal (nondaemon) thread keeps Python alive until the thread terminates. The `Thread` class supplies the following constructor, properties, and methods:

Thread	`class Thread(name=None,target=None,args=(),kwargs={})` Always call `Thread` with named arguments: the number and order of parameters may change in the future, but the parameter names are guaranteed to stay. When you instantiate the class `Thread` itself, pass `target`: `t.run` calls `target(*args,**kwargs)`. When you extend `Thread` and override `run`, don't pass `target`. In either case, execution doesn't begin until you call `t.start()`. `name` is `t`'s name. If `name` is `None`, `Thread` generates a unique name for `t`. If a subclass *`T`* of `Thread` overrides `__init__`, *`T`*`.__init__` *must* call `Thread.__init__` on `self` before any other `Thread` method.
daemon	`t.daemon` `daemon` is a property, `True` when `t` is a daemon (i.e., the process can terminate even when `t` is still active; such a termination also ends `t`); otherwise, `daemon` is `False`. Initially, `t` is a daemon if and only if the thread that creates `t` is a daemon. You can assign to `t.daemon` only before `t.start`; assigning to `t.daemon` sets `t` to be a daemon if, and only if, you assign a true value.
is_alive	`t.is_alive()` Returns `True` when `t` is active (i.e., when `t.start` has executed and `t.run` has not yet terminated). Otherwise, `is_alive` returns `False`.
name	`t.name` `name` is a property returning `t`'s name; assigning `name` rebinds `t`'s name (`name` exists to help you debug; `name` need not be unique among threads).
join	`t.join(timeout=None)` Suspends the calling thread (which must not be `t`) until `t` terminates (when `t` is already terminated, the calling thread does not suspend). `timeout` is covered in "Timeout parameters". You can call `t.join` only after `t.start`. It's OK to call `join` more than once.
run	`t.run()` `run` is the method that executes `t`'s main function. Subclasses of `Thread` can override `run`. Unless overridden, `run` calls the `target` callable passed on `t`'s creation. Do *not* call `t.run` directly; calling `t.run` is the job of `t.start`!
start	`t.start()` `start` makes `t` active and arranges for `t.run` to execute in a separate thread. You must call `t.start` only once for any given thread object `t`; if you call it again, it raises an exception.

Thread Synchronization Objects

The `threading` module supplies several synchronization primitives, types that let threads communicate and coordinate. Each primitive type has specialized uses.

You may not need thread synchronization primitives

As long as you avoid having nonqueue global variables that change, and which several threads access, `queue` (covered in "The queue Module") can often provide all the coordination you need, and so can `concurrent` (covered in "The concurrent.futures Module"). "Threaded Program Architecture" shows how to use `Queue` objects to give your multithreaded programs simple and effective architectures, often without needing any explicit use of synchronization primitives.

Timeout parameters

The synchronization primitives `Condition` and `Event` supply `wait` methods that accept an optional `timeout` argument. A `Thread` object's `join` method also accepts an optional `timeout` argument, as do the `acquire` methods of locks. A `timeout` argument can be `None` (the default) to obtain normal blocking behavior (the calling thread suspends and waits until the desired condition is met). When it is not `None`, a `timeout` argument is a floating-point value that indicates an interval of time in seconds (`timeout` can have a fractional part, so it can indicate any time interval, even a very short one). If `timeout` seconds elapse, the calling thread becomes ready again, even if the desired condition has not been met; in this case, the waiting method returns `False` (otherwise, the method returns `True`). `timeout` lets you design systems that are able to overcome occasional anomalies in a few threads, and thus are more robust. However, using `timeout` may slow your program down: when that matters, be sure to measure your code's speed accurately.

Lock and RLock objects

`Lock` and `RLock` objects supply the same three methods. Here are the signatures and the semantics for an instance `L` of `Lock`:

acquire	`L.acquire(blocking=True)` When `blocking` is `True`, `acquire` locks `L`. When `L` is already locked, the calling thread suspends and waits until `L` is unlocked, then locks `L`. Even when the calling thread was the one that last locked `L`, it still suspends and waits until another thread releases `L`. When `blocking` is `False` and `L` is unlocked, `acquire` locks `L` and returns `True`. When `blocking` is `False` and `L` is locked, `acquire` does not affect `L` and returns `False`.
locked	`L.locked()` Returns `True` when `L` is locked; otherwise, returns `False`.
release	`L.release()` Unlocks `L`, which must be locked. When `L` is locked, any thread may call `L.release`, not just the thread that locked `L`. When more than one thread is blocked on `L` (i.e., has called `L.acquire`, found `L` locked, and is waiting for `L` to be unlocked), `release` wakes up an arbitrary waiting thread. The thread calling `release` does not suspend: it stays ready and continues to execute.

The semantics of an `RLock` object `r` are often more convenient (except in peculiar architectures where you need other threads to be able to release locks that a different thread has acquired). `RLock` is a *re-entrant* lock, meaning that, when `r` is locked, it keeps track of the *owning* thread (i.e., the thread that locked it—which, for an `RLock`, is also the only thread that can release it). The owning thread can call `r.acquire` again without blocking; `r` then just increments an internal count. In a similar situation involving a `Lock` object, the thread would block until some other thread releases the lock. For example, consider the following code snippet:

```
lock = threading.RLock()
global_state = []
def recursive_function(some, args):
    with lock:
# acquires lock, guarantees release at
end
        ...modify global_state...
        if more_changes_needed(global_state):
            recursive_function(other, args)
```

If `lock` was an instance of `threading.Lock`, `recursive_function` would block its calling thread when it calls itself recursively: the `with` statement, finding that the lock has already been acquired (even though that was done by the same thread), would block and wait…and wait… With a `threading.RLock`, no such problem occurs: in this case, since the lock has already been acquired *by the same thread*, on getting acquired again it just increments its internal count and proceeds.

An `RLock` object `r` is unlocked only when it's been released as many times as it has been acquired. An `RLock` is useful to ensure exclusive access to an object when the object's methods call each other; each method can acquire at the start, and release at the end, the same `RLock` instance. `try`/`finally` (covered in "try/finally") is one way to ensure that a lock is indeed released. A `with` statement, covered in "The with Statement", is usually better: all locks, conditions, and semaphores are context managers, so an instance of any of these types can be used directly in a `with` clause to `acquire` it (implicitly with blocking) and ensure it's `release`d at the end of the `with` block.

Condition objects

A `Condition` object `c` wraps a `Lock` or `RLock` object `L`. The class `Condition` exposes the following constructor and methods:

Condition	`class Condition(lock=None)` Creates and returns a new `Condition` object `c` with the lock `L` set to `lock`. If `lock` is `None`, `L` is set to a newly created `RLock` object.
acquire, release	`c.acquire(blocking=1) c.release()` These methods just call `L`'s corresponding methods. A thread must never call any other method on `c` unless the thread holds lock `L`.
notify, notify_all	`c.notify() c.notify_all()` `notify` wakes up one of the threads waiting on `c`. The calling thread must hold `L` before it calls `c.notify()`, and `notify` does not release `L`. The woken-up thread does not become ready until it can acquire `L` again. Therefore, the calling thread normally calls `release` after calling `notify`. `notify_all` is like `notify` but wakes up all waiting threads, not just one.
wait	`c.wait(timeout=None)` `wait` releases `L`, and then suspends the calling thread until some other thread calls `notify` or `notify_all` on `c`. The calling thread must hold `L` before it calls `c.wait()`. `timeout` is covered in "Timeout parameters". After a thread wakes up, either by notification or timeout, the thread becomes ready when it acquires `L` again. When `wait` returns `True` (meaning it has exited normally, not by timeout), the calling thread always holds `L` again.

Usually, a `Condition` object `c` regulates access to some global state `s` shared among threads. When a thread must wait for `s` to change, the thread loops:

```
with c:    while not is_ok_state(s):        c.wait()    do_some_work_using_state(s)
```

Meanwhile, each thread that modifies `s` calls `notify` (or `notify_all` if it needs to wake up all waiting threads, not just one) each time `s` changes:

```
with c:    do_something_that_modifies_state(s)    c.notify()    # or, c.notify_all()
# no need to call c.release(), exiting `with` intrinsically
does
```

You always need to acquire and release `c` around each use of `c`'s methods: doing so via a `with` statement makes using `Condition` instances less error-prone.

Event objects

Event objects let any number of threads suspend and wait. All threads waiting on `Event` object `e` become ready when any other thread calls `e.set()`. `e` has a flag that records whether the event happened; it is initially `False` when `e` is created. `Event` is thus a bit like a simplified `Condition`. Event objects are useful to signal one-shot changes, but brittle for more general use; in particular, relying on calls to `e.clear()` is error-prone. The `Event` class exposes the following methods:

Event	`class Event()` Creates and returns a new `Event` object `e`, with `e`'s flag set to `False`.
clear	`e.clear()` Sets `e`'s flag to `False`.
is_set	`e.is_set()` Returns the value of `e`'s flag, `True` or `False`.
set	`e.set()` Sets `e`'s flag to `True`. All threads waiting on `e`, if any, become ready to run.
wait	`e.wait(timeout=None)` If `e`'s flag is `True`, `wait` returns immediately. Otherwise, `wait` suspends the calling thread until some other thread calls `set`. `timeout` is covered in "Timeout parameters".

Semaphore objects

Semaphores (also known as *counting semaphores*) are a generalization of locks. The state of a `Lock` can be seen as `True` or `False`; the state of a `Semaphore` `s` is a number between `0` and some `n` set when `s` is created (both bounds included). Semaphores can be useful to manage a fixed pool of resources (e.g., 4 printers or 20 sockets), although it's often more robust to use `Queue`s for such purposes. The class `BoundedSemaphore` is very similar, but raises `ValueError` if the state ever becomes higher than the initial value: in many cases, such behavior can be a useful indicator of a coding bug.

Semaphore BoundedSemaphore `class Semaphore(n=1) class BoundedSemaphore(n=1)`

`Semaphore` creates and returns a semaphore object `s` with the state set to `n`; `BoundedSemaphore` is very similar, except that `s.release()` raises `ValueError` if the state becomes higher than `n`. A semaphore object `s` exposes the following methods:

acquire `s.acquire(blocking=True)`

When `s`'s state is `>0`, `acquire` decrements the state by `1` and returns `True`. When `s`'s state is `0` and `blocking` is `True`, `acquire` suspends the calling thread and waits until some other thread calls `s.release`. When `s`'s state is `0` and `blocking` is `False`, `acquire` immediately returns `False`.

release `s.release()`

When `s`'s state is `>0`, or when the state is `0` but no thread is waiting on `s`, `release` increments the state by `1`. When `s`'s state is `0` and some threads are waiting on `s`, `release` leaves `s`'s state at `0` and wakes up an arbitrary one of the waiting threads. The thread that calls `release` does not suspend; it remains ready and continues to execute normally.

Timer objects

A `Timer` object calls a specified callable, in a newly made thread, after a given delay. The class `Timer` exposes the following constructor and methods:

Timer `class Timer(interval,callable,args=(),kwargs={})`

Makes an object `t`, which calls `callable`, `interval` seconds after starting (`interval` is a floating-point number and can include a fractional part).

cancel `cancel()`

`t.cancel()` stops the timer and cancels the execution of its action, as long as `t` is still waiting (hasn't called its callable yet) when you call `cancel`.

start `start()`

`t.start()` starts `t`.

`Timer` extends `Thread` and adds the attributes `function`, `interval`, `args`, and `kwargs`.

A `Timer` is "one-shot"—`t` calls its callable only once. To call `callable` periodically, every `interval` seconds, here's a simple recipe:

```
class Periodic(threading.Timer):
  def __init__(self, interval, callable, args=(), kwargs={}):
    self.callable = callable
    threading.Timer.__init__(self, interval, self._f, args, kwargs
)

  def _f(self, *args, **kwargs):
    Periodic(self.interval, self.callable, args, kwargs).start()
    self.callable(*args, **kwargs)
```

Barrier objects (v3 only)

A `Barrier` is a synchronization primitive allowing a certain number of threads to wait until they've all reached a certain point in their execution, before all of them resume. Specifically, when a thread calls `b.wait()`, it blocks until the specified number of threads have done the same call on `b`; at that time, all the threads blocked on `b` are released.

The class `Barrier` exposes the following constructor, methods, and properties:

Barrier	`class Barrier(`*`num_threads`*`, `*`action`*`=None, `*`timeout`*`=None)` `action` is callable without arguments: if you pass this argument, it executes on any single one of the blocked threads when they are all unblocked. `timeout` is covered in "Timeout parameters".
abort	`abort()` *`b`*`.abort()` puts `Barrier` *`b`* in the *broken state*, meaning that any thread currently waiting resumes with a `threading.BrokenBarrierException` (the same exception also gets raised on any subsequent call to *`b`*`.wait()`). This is an emergency action typically used when a waiting thread is suffering some abnormal termination, to avoid deadlocking the whole program.
broken	`broken` `True` when `b` is in the broken state; otherwise, `False`.
n_waiting	`n_waiting` Number of threads currently waiting on `b`.
parties	`parties` The value passed as `num_threads` in the constructor of `b`.
reset	`reset()` Returns `b` to the initial, empty, nonbroken state; any thread currently waiting on `b`, however, resumes with a `threading.BrokenBarrierException`.

wait	`wait()` The first `b.parties-1` threads calling `b.wait()` block; when the number of threads blocked on `b` is `b.parties-1` and one more thread calls `b.wait()`, all the threads blocked on `b` resume. `b.wait()` returns an `int` to each resuming thread, all distinct and in `range(b.parties)`, in unspecified order; threads can use this return value to determine which one should do what next (though passing `action` in the `Barrier`'s constructor is simpler and often sufficient).

`threading.Barrier` exists only in v3; in v2, you could implement it yourself, as shown, for example, on Stack Overflow.

Thread Local Storage

The `threading` module supplies the class `local`, which a thread can use to obtain *thread-local storage* (TLS), also known as *per-thread data*. An instance `L` of `local` has arbitrary named attributes that you can set and get, and stores them in a dictionary `L.__dict__` that you can also access. `L` is fully thread-safe, meaning there is no problem if multiple threads simultaneously set and get attributes on `L`. Most important, each thread that accesses `L` sees a disjoint set of attributes, and any changes made in one thread have no effect in other threads. For example:

```
                                              in main thread, setting zop to
import threadingL = threading.local()print('42                                ')L.zop =
                        in subthread, setting zop to
 42def targ():  print('23                            ')  L.zop = 23  print('
in subthread, zop is
now                       ', L.zop)t = threading.Thread(target=targ)t.start()t.join()
       in main thread, zop is              #          in main thread, setting zop to
print('now                         ', L.zop)prints:  # 42
  in subthread, setting zop to     in subthread, zop is now
# 23                              # 23                             #
in main thread, zop is now
42
```

TLS makes it easier to write code meant to run in multiple threads, since you can use the same namespace (an instance of `threading.local`) in multiple threads without the separate threads interfering with each other.

The queue Module

The `queue` module (named `Queue` in v2) supplies queue types supporting multithread access, with one main class, two subclasses, and two exception classes:

Queue	`class Queue(    maxsize=0)` `Queue`, the main class for module `queue`, implements a *First-In, First-Out* (*FIFO*) queue (the item retrieved each time is the one that was added earliest), and is covered in "Methods of Queue Instances". When `maxsize` is `>0`, the new `Queue` instance `q` is considered full when `q` has `maxsize` items. A thread inserting an item with the `block` option, when `q` is full, suspends until another thread extracts an item. When `maxsize` is `<=0`, `q` is never considered full, and is limited in size only by available memory, like normal Python containers.

LifoQueue	`class LifoQueue(maxsize=0)` `LifoQueue` is a subclass of `Queue`; the only difference is that `LifoQueue` is *Last-In, First-Out* (*LIFO*), which means the item retrieved each time is the one that was added most recently.
PriorityQueue	`class PriorityQueue(      maxsize=0)` `PriorityQueue` is a subclass of `Queue`; the only difference is that `PriorityQueue` implements a *priority* queue—the item retrieved each time is the smallest one currently in the queue. As there's no *key*= argument, you generally use, as queue items, pairs `(`*`priority`*`, `*`payload`*`)`, with low values of `priority` meaning earlier retrieval.
Empty	`Empty` is the exception that `q.get(False)` raises when `q` is empty.
Full	`Full` is the exception that `q.put(x,False)` raises when `q` is full.

Methods of Queue Instances

An instance `q` of the class `Queue` (or either of its subclasses) supplies the following methods, all thread-safe and guaranteed to be atomic:

empty	`q.empty()` Returns `True` if `q` is empty; otherwise, `False`.
full	`q.full()` Returns `True` if `q` is full; otherwise, `False`.
get, get_nowait	`q.get(block=True,timeout=None)` When `block` is `False`, `get` removes and returns an item from `q` if one is available; otherwise, `get` raises `Empty`. When `block` is `True` and `timeout` is `None`, `get` removes and returns an item from `q`, suspending the calling thread, if need be, until an item is available. When `block` is `True` and `timeout` is not `None`, `timeout` must be a number `>=0` (which may include a fractional part to specify a fraction of a second), and `get` waits for no longer than `timeout` seconds (if no item is yet available by then, `get` raises `Empty`). `q.get_nowait()` is like `q.get(False)`, which is also like `q.get(`*`timeout`*`=0.0)`. `get` removes and returns items in the same order as `put` inserted them (FIFO), if `q` is a direct instance of `Queue` itself; LIFO, if `q` is an instance of `LifoQueue`; smallest-first, if `q` is an instance of `PriorityQueue`.
put, put_nowait	`q.put(item,block=True,timeout=None)` When `block` is `False`, `put` adds `item` to `q` if `q` is not full; otherwise, `put` raises `Full`. When `block` is `True` and `timeout` is `None`, `put` adds `item` to `q`, suspending the calling thread, if need be, until `q` is not full. When `block` is `True` and `timeout` is not `None`, `timeout` must be a number `>=0` (which may include a fractional part to specify a fraction of a second), and `put` waits for no longer than `timeout` seconds (if `q` is still full by then, `put` raises `Full`). `q.put_nowait(item)` is like `q.put(item,False)`, also like `q.put(item,`*`timeout`*`=0.0)`.
qsize	`q.qsize()` Returns the number of items that are currently in `q`.

Moreover, `q` maintains an internal, hidden count of *unfinished tasks*, which starts at zero. Each call to `get` increments the count by one. To decrement the count by one, when a worker thread has finished processing a task, it calls `q.task_done()`. To synchronize on "all tasks done," call `q.join()`: `join` continues the calling thread when the count of unfinished tasks is zero; when the count is nonzero, `q.join()` blocks the calling thread, and unblocks later, when the count goes to zero.

You don't have to use `join` and `task_done` if you prefer to coordinate threads in other ways, but `join` and `task_done` do provide a simple, useful approach to coordinate systems of threads using a `Queue`.

`Queue` offers a good example of the idiom "It's easier to ask forgiveness than permission" (EAFP), covered in "Error-Checking Strategies". Due to multithreading, each nonmutating method of `q` (`empty`, `full`, `qsize`) can only be advisory. When some other thread executes and mutates `q`, things can change between the instant a thread gets the information from a nonmutating method and the very next moment, when the thread acts on the information. Relying on the "look before you leap" (LBYL) idiom is therefore futile, and fiddling with locks to try to fix things is a substantial waste of effort. Just avoid fragile LBYL code, such as:

```
                       no work to
if q.empty():    print('perform           ')else:    x = q.get_nowait()    work_on(x)
```

and instead use the simpler and more robust EAFP approach:

```
                                                        no work to
try:    x = q.get_nowait()except queue.Empty:    print('perform           ')else:
work_on(x)
```

The multiprocessing Module

The `multiprocessing` module supplies functions and classes you can use to code pretty much as you would for multithreading, but distributing work across processes, rather than across threads: the class `Process` (similar to `threading.Thread`) and classes for synchronization primitives (`BoundedSemaphore`, `Condition`, `Event`, `Lock`, `RLock`, `Semaphore`, and, in v3 only, `Barrier`—each similar to the class with the same names in module `threading`; also, `Queue`, and `JoinableQueue`, both similar to `queue.Queue`). These classes make it easy to take code written to use `threading`, and make a version that uses `multiprocessing` instead; you just need to pay attention to the differences we cover in "Differences Between Multiprocessing and Threading".

It's usually best to avoid sharing state among processes: use queues, instead, to explicitly pass messages among them. However, for those occasions in which you do need to share some state, `multiprocessing` supplies classes to access shared memory (`Value` and `Array`), and—more flexibly (including coordination among different computers on a network) though with more overhead—a `Process` subclass, `Manager`, designed to hold arbitrary data and let other processes manipulate that data via *proxy* objects. We cover state sharing in "Sharing State: Classes Value, Array, and Manager".

When you're writing new multiprocessing code, rather than porting code originally written to use `threading`, you can often use different approaches supplied by `multiprocessing`. The `Pool` class, in particular, can often simplify your code. We cover `Pool` in "Multiprocessing Pool".

Other advanced approaches, based on `Connection` objects built by the `Pipe` factory function or wrapped in `Client` and `Listener` objects, are, on the other hand, more flexible, but potentially more complex; we do not cover them further in this book. For more thorough coverage of `multiprocessing`, refer to the online docs and to good third-party online tutorials.

Differences Between Multiprocessing and Threading

You can port code written to use `threading` into a variant using `multiprocessing` instead—however, there are differences you must consider.

Structural differences

All objects that you exchange between processes (for example, via a queue, or an argument to a `Process`'s `target` function) are serialized via `pickle`, covered in "The pickle and cPickle Modules". Therefore, you can exchange only objects that can be thus serialized. Moreover, the serialized bytestring cannot exceed about 32 MB (depending on the platform), or else an exception gets raised; therefore, there are limits to the size of objects you can exchange.

Especially in Windows, child processes *must* be able to import as a module the main script that's spawning them. Therefore, be sure to guard all top-level code in the main script (meaning code that must not be executed again by child processes) with the usual `if __name__ == '__main__'` idiom, covered in "The Main Program".

If a process is abruptly killed (for example, via a signal) while using a queue, or holding a synchronization primitive, it won't be able to perform proper cleanup on that queue or primitive. As a result, that queue or primitive may get corrupted, causing errors in all other processes trying to use it.

The Process class

The class `multiprocessing.Process` is very similar to `threading.Thread`, but, in addition to all of `Thread`'s attributes and methods, it supplies a few more:

authkey `authkey`

The process's authorization key, a bytestring: initialized to random bytes supplied by `os.urandom()`, but you can reassign it later if you wish. Used in the authorization handshake for advanced uses we do not cover in this book.

exitcode `exitcode`

`None` when the process has not exited yet; otherwise, the process's exit code: an `int`, `0` for success, `>0` for failure, `<0` when the process was killed.

pid `pid`

`None` when the process has not started yet; otherwise, the process's identifier as set by the operating system.

terminate `terminate()`

Kills the process (without giving it a chance to execute termination code, such as cleanup of queues and synchronization primitives; beware of the likelihood of causing errors when the process is using a queue or holding a primitive).

Differences in queues

The class `multiprocessing.Queue` is very similar to `queue.Queue`, except that an instance `q` of `multiprocessing.Queue` does *not* supply the methods `join` and `task_done`. When methods of `q` raise exceptions due to time-outs, they raise instances of `queue.Empty` or `queue.Full`. `multiprocessing` has no equivalents to `queue`'s `LifoQueue` and `PriorityQueue` classes.

The class `multiprocessing.JoinableQueue` does supply the methods `join` and `task_done`, but with a semantic difference compared to `queue.Queue`: with an instance `q` of `multiprocessing.JoinableQueue`, the process that calls `q.get` *must* call `q.task_done` when it's done processing that unit of work (it's not optional, as it would be when using `queue.Queue`).

All objects you `put` in multiprocessing queues must be serializable by `pickle`. There may be a small delay between the time you execute `q.put` and the time the object is available from `q.get`. Lastly, remember that an abrupt exit (crash or signal) of a process using `q` may leave `q` unusable for any other process.

Differences in synchronization primitives

In the `multiprocessing` module, the `acquire` method of the synchronization primitive classes `BoundedSemaphore`, `Lock`, `RLock`, and `Semaphore` has the signature `acquire(block=True, timeout=None)`; `timeout`'s semantics are covered in "Timeout parameters".

Sharing State: Classes Value, Array, and Manager

To use shared memory to hold a single primitive value in common among two or more processes, `multiprocessing` supplies the class `Value`; for a fixed-length array of primitive values, the class `Array`. For more flexibility (including nonprimitive values, and "sharing" among different systems joined by a network but sharing no memory) at the cost of higher overhead, `multiprocessing` supplies the class `Manager`, which is a subclass of `Process`.

The Value class

The constructor for the class `Value` has the signature:

Value `Value(typecode, *args, lock=True)`

`typecode` is a string defining the primitive type of the value, just like for module `array`, as covered in Table 15-3. (Alternatively, `typecode` can be a type from the module `ctypes`, mentioned in "ctypes", but this is rarely necessary.) `args` is passed on to the type's constructor: therefore, `args` is either absent (in which case the primitive is initialized as per its default, typically `0`) or a single value, which is used to initialize the primitive.

When `lock` is `True` (the default), `Value` makes and uses a new lock to guard the instance. Alternatively, you can pass as `lock` an existing `Lock` or `RLock` instance. You can even pass `lock=False`, but that is rarely advisable: when you do, the instance is not guarded (thus, it is not synchronized among processes) and is missing the method `get_lock`. If you do pass `lock`, you *must* pass it as a named argument, using `lock=something`.

An instance `v` of the class `Value` supplies the following attributes and methods:

get_lock `get_lock()`

Returns (but neither acquires nor releases) the lock guarding `v`.

---

value `value`

A read/write attribute, used to set and get `v`'s underlying primitive value.

To ensure atomicity of operations on `v`'s underlying primitive value, guard the operation in a `with v.get_lock():` statement. A typical example of such usage might be for augmented assignment, as in:

```
with v.get_lock():
    v.value += 1
```

If any other process does an unguarded operation on that same primitive value, however, even an atomic one such as a simple assignment like `v.value = x`, "all bets are off": the guarded operation and the unguarded one can get your system into a *race condition*. Play it safe: if any operation at all on `v.value` is not atomic (and thus needs to be guarded by being within a `with v.get_lock():` block), guard *all* operations on `v.value` by placing them within such blocks.

The Array class

The constructor for the class `Array` has the signature:

Array `Array(typecode, size_or_initializer, lock=True)`

A fixed-length array of primitive values (all items being of the same primitive type).

`typecode` is a string defining the primitive type of the value, just like for the module `array`, as covered in Table 15-3. (Alternatively, `typecode` can be a type from the module `ctypes`, mentioned in "ctypes", but this is rarely necessary.) `size_or_initializer` can be an iterable, used to initialize the array; alternatively, it can be an integer, used as the length of the array (in this case, each item of the array is initialized to `0`).

When `lock` is `True` (the default), `Array` makes and uses a new lock to guard the instance. Alternatively, you can pass as `lock` an existing `Lock` or `RLock` instance. You can even pass `lock=False`, but that is rarely advisable: when you do, the instance is not guarded (thus it is not synchronized among processes) and is missing the method `get_lock`. If you do pass `lock`, you *must* pass it as a named argument, using `lock=something`.

An instance `a` of the class `Array` supplies the following method:

get_lock `get_lock()`

Returns (but neither acquires nor releases) the lock guarding `a`.

`a` is accessed by indexing and slicing, and modified by assigning to an indexing or to a slice. `a` is fixed-length: therefore, when you assign to a slice, you must assign an iterable of the same length as the slice you're assigning to. `a` is also iterable.

In the special case where `a` was built with `typecode 'c'`, you can also access `a.value` to get `a`'s contents as a bytestring, and you can assign to `a.value` any bytestring no longer than `len(a)`. When `s` is a bytestring with `len(s)<len(a)`, `a.value = s` means `a[:len(s)+1] = s+b'\0'`; this mirrors the representation of `char`

strings in the C language, terminated with a `0` byte. For example:

```
                              four score and
a = multiprocessing.Array('c', b'seven              ')a.value = b'five'print(a.value)
                                         b'five\x00score and
   # prints b'five'print(a[:])      # prints seven'
```

The Manager class

`multiprocessing.Manager` is a subclass of `multiprocessing.Process`, with the same methods and attributes. In addition, it supplies methods to build an instance of any of the multiprocessing synchronization primitives, plus `Queue`, `dict`, `list`, and `Namespace`, the latter being a class that just lets you set and get arbitrary named attributes. Each of the methods has the name of the class whose instances it builds, and returns a *proxy* to such an instance, which any process can use to call methods (including special methods, such as indexing of instances of `dict` or `list`) on the instance held in the manager process.

Proxy objects pass most operators, and accesses to methods and attributes, on to the instance they proxy for; however, they don't pass on *comparison* operators—if you need a comparison, you need to take a local copy of the proxied object. For example:

```
p = some_manager.list()p[:] = [1, 2, 3]print(p == [1, 2, 3])       # prints False
, as it compares with pprint(list(p) == [1, 2, 3])  # prints True
, as it compares with
copy
```

The constructor of `Manager` takes no arguments. There are advanced ways to customize `Manager` subclasses to allow connections from unrelated processes (including ones on different computers connected via a network) and to supply a different set of building methods, but we do not cover them in this book. Rather, one simple, often-sufficient approach to using `Manager` is to explicitly transfer to other processes the proxies it produces, typically via queues, or as arguments to a `Process`'s `target` function.

For example, suppose there's a long-running, CPU-bound function `f` that, given a string as an argument, eventually returns a corresponding result; given a `set` of strings, we want to produce a `dict` with the strings as keys and the corresponding results as values. To be able to follow on which processes `f` runs, we also `print` the process ID just before calling `f`. Here's one way to do it:

Example 14-1.

```
import multiprocessing as mp

def f(s):

"""Run a long time, and eventually return a
result."""
    import time, random
                                   # simulate
    time.sleep(random.random()*2)  slowness
                # some computation or
    return s+s  other

def runner(s, d):
    print(os.getpid())
    d[s] = f(s)

def make_dict(set_of_strings):
    mgr = mp.Manager()
    d = mgr.dict()
    workers = []
    for s in set_of_strings:
        p = mp.Process(target=runner, args=(s, d))
        p.start()
        workers.append(p)
    for p in workers:
        p.join()
    return dict(d)
```

Multiprocessing Pool

In real life, beware of creating an unbounded number of worker processes, as we just did in Example 14-1. Performance benefits accrue only up to the number of cores in your machine (available by calling `multiprocessing.cpu_count()`), or a number just below or just above this, depending on such minutiae as your platform and other load on your computer. Making more worker processes than such an optimal number incurs substantial extra overhead to no good purpose.

As a consequence, it's a common design pattern to start a *pool* with a limited number of worker processes, and farm out work to them. The class `multiprocessing.Pool` handles the orchestration of this design pattern on your behalf.

The Pool class

The constructor for the class `Pool` has the signature:

Pool `Pool(processes=None, initializer=None, initargs=(), maxtasksperchild=None)`

Builds and returns an instance `p` of `Pool`. `processes` is the number of processes in the pool; it defaults to the value returned by `cpu_count()`. When `initializer` is not `None`, it's a function, called at the start of each process in the pool, with `initargs` as arguments, like `initializer(*initargs)`.

When `maxtasksperchild` is not `None`, it's the maximum number of tasks executed in each process in the pool. When a process in the pool has executed that many tasks, it terminates, and a new process starts and joins the pool. When `maxtasksperchild` is `None` (the default), processes live as long as the pool.

An instance `p` of the class `Pool` supplies the following methods (all of them must be called only in the process that built instance `p`):

apply `apply(func, args=(), kwds={})`

In an arbitrary one of the worker processes, runs `func(*args, **kwds)`, waits for it to finish, and returns `func`'s result.

apply_async `apply_async(func, args=(), kwds={}, callback=None)`

In an arbitrary one of the worker processes, starts running `func(*args, **kwds)`, and, without waiting for it to finish, immediately returns an `AsyncResult` (see "The AsyncResult class") instance, which eventually gives `func`'s result, when that result is ready. When `callback` is not `None`, it's a function called (in a separate thread in the process that calls `apply_async`), with `func`'s result as the only argument, when that result is ready; `callback` should execute rapidly, or otherwise it blocks the process. `callback` may mutate its argument if that argument is mutable; `callback`'s return value is irrelevant.

close `close()`

No more tasks can be submitted to the pool. Worker processes terminate when they're done with all outstanding tasks.

imap `imap(func, iterable, chunksize=1)`

Returns an iterator calling `func` on each item of `iterable`, in order. `chunksize` determines how many consecutive items are sent to each process; on a very long `iterable`, a large `chunksize` can improve performance. When `chunksize` is `1` (the default), the returned iterator has a method `next` (even on v3, where the canonical name of the iterator's method is `__next__`), which optionally accepts a `timeout` argument (a floating-point value in seconds), and raises `multiprocessing.TimeoutError` should the result not yet be ready after `timeout` seconds.

imap_unordered `imap_unordered(func, iterable, chunksize=1)`

Same as `imap`, but the ordering of the results is arbitrary (this can sometimes improve performance, when you don't care about the order the results are iterated on).

join	`join()` Waits for all worker processes to exit. You must call `close` or `terminate` before you call `join`.
map	`map(func, iterable, chunksize=1)` Calls `func` on each item of `iterable`, in order, in worker processes in the pool; waits for them all to finish, and returns the list of results. `chunksize` determines how many consecutive items are sent to each process; on a very long `iterable`, a large `chunksize` can improve performance.
map_async	`map_async(func, iterable, chunksize=1, callback=None)` Arranges for `func` to be called on each item of `iterable` in worker processes in the pool; without waiting for any of this to finish, immediately returns an `AsyncResult` (see "The AsyncResult class") instance, which eventually gives the list of `func`'s results, when that list is ready. When `callback` is not `None`, it's a function and gets called (in a separate thread in the process that calls `map_async`) with the list of `func`'s results, in order, as the only argument, when that list is ready; `callback` should execute rapidly, or otherwise it blocks the process. `callback` may mutate its list argument; `callback`'s return value is irrelevant.
terminate	`terminate()` Terminates all worker processes immediately, without waiting for them to complete work.

For example, here's a `Pool`-based approach to perform the same task as in Example 14-1:

```
import multiprocessing as mp

def f(s):
    """Run a long time, and eventually return a
    result."""
    import time, random
                                          # simulate
    time.sleep(random.random()*2)   slowness
                  # some computation or
    return s+s    other

def runner(s):
    print(os.getpid())
    return s, f(s)

def make_dict(set_of_strings):
    with mp.Pool() as pool:
        d = dict(pool.imap_unordered(runner, set_of_strings
))
        return d
```

The AsyncResult class

The methods `apply_async` and `map_async` of the class `Pool` return an instance of the class `AsyncResult`. An instance `r` of the class `AsyncResult` supplies the following methods:

get `get(timeout=None)`

Blocks and returns the result when ready, or re-raises the exception raised while computing the result. When `timeout` is not `None`, it's a floating-point value in seconds; `get` raises `multiprocessing.TimeoutError` should the result not yet be ready after `timeout` seconds.

ready `ready()`

Does not block; returns `True` if the result is ready; otherwise, returns `False`.

successful `successful()`

Does not block: returns `True` if the result is ready and the computation did not raise an exception; returns `False` if the computation raised an exception. If the result is not yet ready, `successful` raises `AssertionError`.

wait `wait(timeout=None)`

Blocks and waits until the result is ready. When `timeout` is not `None`, it's a `float` in seconds: `wait` raises `multiprocessing.TimeoutError` should the result not yet be ready after `timeout` seconds.

The concurrent.futures Module

The `concurrent` package supplies a single module, `futures`. `concurrent.futures` is in the standard library only in v3; to use it in v2, download and install the backport with `pip2 install futures` (or, equivalently, `python2 -m pip install futures`).

`concurrent.futures` supplies two classes, `ThreadPoolExecutor` (using threads as workers) and `ProcessPoolExecutor` (using processes as workers), which implement the same abstract interface, `Executor`. Instantiate either kind of pool by calling the class with one argument, `max_workers`, specifying how many threads or processes the pool should contain. You can omit `max_workers` to let the system pick the number of workers (except that you have to explicitly specify `max_workers` to instantiate `ThreadPoolExecutor` for the v2 backport, only).

An instance `e` of an `Executor` class supports the following methods:

map `map(func, *iterables, timeout=None)`

Returns an iterator `it` whose items are the results of `func` called with one argument from each of the `iterables`, in order (using multiple worker threads or processes to execute `func` in parallel). When `timeout` is not `None`, it's a `float` number of seconds: should `next(it)` not produce any result in `timeout` seconds, raises `concurrent.futures.TimeoutError`.

In v3 only, you may specify (by name) argument `chunksize`: ignored for a `ThreadPoolExecutor`, for a `ProcessPoolExecutor` it sets how many items of each iterable in `iterables` are passed to each worker process.

shutdown	`shutdown(wait=True)` No more calls to `map` or `submit` allowed. When `wait` is `True`, `shutdown` blocks until all pending futures are done; when `False`, `shutdown` returns immediately. In either case, the process does not terminate until all pending futures are done.
submit	`submit(func, *a, **k)` Ensures `func(*a, **k)` executes on an arbitrary one of the pool's processes or threads. Does not block, but rather returns a `Future` instance.

Any instance of an `Executor` is also a context manager, and therefore suitable for use on a `with` statement (`__exit__` being like `shutdown(wait=True)`).

For example, here's a `concurrent`-based approach to perform the same task as in Example 14-1:

```
import concurrent.futures as cf

def f(s):

"""run a long time and eventually return a
result"""
    ...

def runner(s):
    return s, f(s)

def make_dict(set_of_strings):
    with cf.ProcessPoolExecutor() as e:
        d = dict(e.map(runner, set_of_strings))
    return d
```

The `submit` method of an `Executor` returns a `Future` instance. A `Future` instance `f` supplies the methods described in Table 14-1.

Table 14-1.

add_done_callback	`add_done_callback(func)` Add callable `func` to `f`; `func` is called, with `f` as the only argument, when `f` completes (i.e., is cancelled or finishes).
cancel	`cancel()` Tries cancelling the call; returns `False` when the call is being executed and cannot be cancelled; otherwise, returns `True`.
cancelled	`cancelled()` Returns `True` when the call was successfully cancelled; otherwise, returns `False`.

done	`done()` Returns True when the call is completed (i.e., is finished or succesfully cancelled).
exception	`exception(timeout=None)` Returns the exception raised by the call, or `None` if the call raised no exception. When `timeout` is not `None`, it's a `float` number of seconds to wait; if the call hasn't completed after `timeout` seconds, raises `concurrent.futures.TimeoutError`; if the call is cancelled, raises `concurrent.futures.CancelledError`.
result	`result(timeout=None)` Returns the call's result. When `timeout` is not `None`, it's a `float` number of seconds; if the call hasn't completed after `timeout` seconds, raises `concurrent.futures.TimeoutError`; if the call is cancelled, raises `concurrent.futures.CancelledError`.
running	`running()` Returns `True` when the call is executing and cannot be cancelled.

The `concurrent.futures` module also supplies two functions:

as_completed	`as_completed(fs, timeout=None)` Returns an iterator `it` over the `Future` instances that are the items of iterator `fs`. If there are duplicates in `fs`, each is only yielded once. `it` yields one completed future at a time, as they complete; if `timeout` is not `None`, it's a `float` number of seconds, and—should it ever happen that no new future can be yielded after `timeout` seconds from the previous one—`as_completed` raises `concurrent.futures.Timeout`.

wait `wait(fs, timeout=None, return_when=ALL_COMPLETED)`

Waits for the `Future` instances that are the items of iterator `fs`. Returns a named 2-tuple of `set` s: the first `set`, named `done`, contains the futures that completed (meaning that they either finished or were cancelled) before `wait` returned. The second `set`, named `not_done`, contains yet-uncompleted futures.

`timeout`, if not `None`, is a `float` number of seconds, the maximum time `wait` lets elapse before returning (when `timeout` is `None`, `wait` returns only when `return_when` is satisfied, no matter the elapsed time before that happens).

`return_when` controls when, exactly, `wait` returns; it must be one of three constants supplied by module `concurrent.futures:`

`ALL_COMPLETED`

Return when all futures finish or are cancelled

`FIRST_COMPLETED`

Return when any future finishes or is cancelled

`FIRST_EXCEPTION`

Return when any future raises an exception; should no future raise an exception, becomes equivalent to `ALL_COMPLETED`

Threaded Program Architecture

A threaded program should always try to arrange for a *single* thread to deal with any given object or subsystem that is external to the program (such as a file, a database, a GUI, or a network connection). Having multiple threads that deal with the same external object is possible, but can often cause gnarly problems.

When your threaded program must deal with some external object, devote a thread to such dealings, using a `Queue` object from which the external-interfacing thread gets work requests that other threads post. The external-interfacing thread can return results by putting them on one or more other `Queue` objects. The following example shows how to package this architecture into a general, reusable class, assuming that each unit of work on the external subsystem can be represented by a callable object. (In examples, remember: in v2, the module `queue` is spelled `Queue`; the class `Queue` in that module is spelled with an uppercase `Q` in both v2 and v3).

```
import threading, queue
class ExternalInterfacing(threading.Thread):
    def __init__(self, external_callable, **kwds):
        threading.Thread.__init__(self, **kwds)
# could use
`super`
        self.daemon = True
        self.external_callable = external_callable
        self.work_request_queue = queue.Queue()
        self.result_queue = queue.Queue()
        self.start()
    def request(self, *args, **kwds):
        """called by other threads as external_callable would
        be"""
        self.work_request_queue.put((args,kwds))
        return self.result_queue.get()
    def run(self):
        while True:
            a, k = self.work_request_queue.get()
            self.result_queue.put(self.external_callable(*a, **k))
```

Once some `ExternalInterfacing` object `ei` is instantiated, any other thread may call `ei.request` just as it would call `external_callable` without such a mechanism (with or without arguments as appropriate). The advantage of the `ExternalInterfacing` mechanism is that all calls upon `external_callable` are serialized. This means they are performed by just one thread (the thread object bound to `ei`) in some defined sequential order, without overlap, race conditions (hard-to-debug errors that depend on which thread happens to get there first), or other anomalies that might otherwise result.

If several callables need to be serialized together, you can pass the callable as part of the work request, rather than passing it at the initialization of the class `ExternalInterfacing`, for greater generality. The following example shows this more general approach:

```
import threading, queue
class Serializer(threading.Thread):
    def __init__(self, **kwds):
        threading.Thread.__init__(self, **kwds)
# could use
`super`
        self.daemon = True
        self.work_request_queue = queue.Queue()
        self.result_queue = queue.Queue()
        self.start()
    def apply(self, callable, *args, **kwds):
        """called by other threads as callable would
        be"""
        self.work_request_queue.put((callable, args, kwds))
        return self.result_queue.get()
    def run(self):
        while True:
            callable, args, kwds = self.work_request_queue.get()
            self.result_queue.put(callable(*args, **kwds))
```

Once a `Serializer` object `ser` has been instantiated, any other thread may call `ser.apply( external_callable)` just as it would call `external_callable` without such a mechanism (with or without further arguments as appropriate). The `Serializer` mechanism has the same advantages as `ExternalInterfacing`, except that all calls to the same or different callables wrapped by a single `ser` instance are now serialized.

The user interface of the whole program is an external subsystem, and thus should be dealt with by a single thread —specifically, the main thread of the program (this is mandatory for some user interface toolkits, and advisable even when not mandatory). A `Serializer` thread is therefore inappropriate. Rather, the program's main thread should deal only with user-interface issues, and farm out actual work to worker threads that accept work requests on a `Queue` object and return results on another. A set of worker threads is generally known as a *thread pool*. As shown in the following example, all worker threads should share a single queue of requests and a single queue of results, since the main thread is the only one to post work requests and harvest results:

```
import threading
class Worker(threading.Thread):
    IDlock = threading.Lock()
    request_ID = 0
    def __init__(self, requests_queue, results_queue, **kwds):
        threading.Thread.__init__(self, **kwds)
        self.daemon = True
        self.work_request_queue = requests_queue
        self.result_queue = results_queue
        self.start()
    def perform_work(self, callable, *args, **kwds):

"""called by main thread as callable would be, but w/o
return"""
        with self.IDlock:
            Worker.request_ID += 1
            self.work_request_queue.put(
                (Worker.request_ID, callable, args, kwds))
            return Worker.request_ID
    def run(self):
        while True:
            request_ID, callable, a, k = self.work_request_queue.get()
            self.result_queue.put((request_ID, callable(*a, **k)))
```

The main thread creates the two queues, and then instantiates worker threads as follows:

```
import queue
requests_queue = queue.Queue()
results_queue = queue.Queue()
for i in range(number_of_workers):
    worker = Worker(requests_queue, results_queue
)
```

Whenever the main thread needs to farm out work (execute some callable object that may take substantial elapsed time to produce results), the main thread calls `worker.perform_work(callable)` , much as it would call `callable` without such a mechanism (with or without further arguments as appropriate). However, `perform_work` does not return the result of the call. Instead of the results, the main thread gets an `id` that identifies the work request. If the main thread needs the results, it can keep track of that `id`, since the request's results are tagged with that `id` when they appear. The advantage of this mechanism is that the main thread does not block waiting for the callable's lengthy execution to complete, but rather becomes ready again at once and can immediately return to its main business of dealing with the user interface.

The main thread must arrange to check the `results_queue`, since the result of each work request eventually appears there, tagged with the request's `id`, when the worker thread that took that request from the queue finishes computing the result. How the main thread arranges to check for both user interface events and the results coming back from worker threads onto the results queue depends on what user interface toolkit is used or—if the user interface is text-based—on the platform on which the program runs.

A widely applicable, though not always optimal, general strategy is for the main thread to *poll* (check the state of the results queue periodically). On most Unix-like platforms, the function `alarm` of the module `signal` allows polling. The `Tkinter` GUI toolkit supplies method `after`, which is usable for polling. Some toolkits and platforms afford more effective strategies (letting a worker thread alert the main thread when it places some result on the results queue), but there is no generally available, cross-platform, cross-toolkit way to arrange for this. Therefore, the following artificial example ignores user interface events and just simulates work by evaluating random expressions, with random delays, on several worker threads, thus completing the previous example:

```
import random, time
def make_work():
    return '{} {} {}'.format(random.randrange(2,10),
        random.choice(('+', '-', '*', '/', '%', '**')),
        random.randrange(2,10))
def slow_evaluate(expression_string):
    time.sleep(random.randrange(1,5))
    return eval(expression_string)
workRequests = {}
def showResults():
    while True:
        try: id, results = results_queue.get_nowait()
        except queue.Empty: return
        print('Result {}: {} -> {}'.format(
            id, work_requests[id], results))
        del work_requests[id]
for i in range(10):
    expression_string = make_work()
    id = worker.perform_work(slow_evaluate, expression_string)
    work_requests[id] = expression_string
    print('Submitted request {}: {}'.format(id, expression_string
))
    time.sleep(1)
    showResults()
while work_requests:
    time.sleep(1)
    showResults()
```

Process Environment

The operating system supplies each process `P` with an *environment*, a set of variables whose names are strings (most often, by convention, uppercase identifiers) and whose contents are strings. In "Environment Variables", we cover environment variables that affect Python's operations. Operating system shells offer ways to examine and modify the environment via shell commands and other means mentioned in “Environment Variables”.

Process environments are self-contained

The environment of any process `P` is determined when `P` starts. After startup, only `P` itself can change `P`’s environment. Changes to `P`’s environment affect only `P` itself: the environment is *not* a means of inter-process communication (IPC). Nothing that `P` does affects the environment of `P`’s parent process (the process that started `P`), nor of those of child processes *previously* started from `P` and now running, nor of processes unrelated to `P`. Child processes of `P` normally get a copy of `P`’s environment as their starting environment. In this narrow sense, changes to `P`’s environment do affect child processes that `P` starts *after* such changes.

The module `os` supplies the attribute `environ`, a mapping that represents the current process’s environment. `os.environ` is initialized from the process environment when Python starts. Changes to `os.environ` update the current process’s environment if the platform supports such updates. Keys and values in `os.environ` must be strings. On Windows (but not on Unix-like platforms), keys into `os.environ` are implicitly uppercased. For example, here’s how to try to determine which shell or command processor you’re running under:

```
import os
shell = os.environ.get('COMSPEC')
if shell is None: shell = os.environ.get('SHELL')
if shell is None: shell = 'an unknown command processor'
print('Running under', shell)
```

When a Python program changes its environment (e.g., via `os.environ['X']='Y'`), this does not affect the environment of the shell or command processor that started the program. As already explained—and for all programming languages including Python—changes to a process’s environment affect only the process itself, not other processes that are currently running.

Running Other Programs

You can run other programs via functions in the `os` module or (at a higher and usually preferable level of abstraction) with the `subprocess` module.

Running Other Programs with the os Module

The best way for your program to run other processes is usually with the `subprocess` module, covered in “The Subprocess Module”. However, the `os` module also offers several ways to do this, which, in some rare cases, may be simpler.

The simplest way to run another program is through the function `os.system`, although this offers no way to control the external program. The `os` module also provides a number of functions whose names start with `exec`. These functions offer fine-grained control. A program run by one of the `exec` functions replaces the current program (i.e., the Python interpreter) in the same process. In practice, therefore, you use the `exec` functions mostly on platforms that let a process duplicate itself by `fork` (i.e., Unix-like platforms). `os` functions whose names start with `spawn` and `popen` offer intermediate simplicity and power: they are cross-platform and not quite as simple as `system`, but simple and usable enough for many purposes.

The `exec` and `spawn` functions run a given executable file, given the executable file’s path, arguments to pass to it, and optionally an environment mapping. The `system` and `popen` functions execute a command, which is a string passed to a new instance of the platform's default shell (typically */bin/sh* on Unix, *cmd.exe* on Windows). A *command* is a more general concept than an *executable file*, as it can include shell functionality (pipes, redirection, built-in shell commands) using the shell syntax specific to the current platform. `os` provides the following functions:

execl, execle, execlp, execv, execve, execvp, execvpe

```
execl(path,*args)
execle(path,*args)
execlp(path,*args)
execv(path,args)
execve(path,args,env)
execvp(path,args)
execvpe(path,args,env)
```

These functions run the executable file (program) indicated by string `path`, replacing the current program (i.e., the Python interpreter) in the current process. The distinctions encoded in the function names (after the prefix `exec`) control three aspects of how the new program is found and run:

- Does `path` have to be a complete path to the program's executable file, or can the function accept a name as the `path` argument and search for the executable in several directories, as operating system shells do? `execlp`, `execvp`, and `execvpe` can accept a `path` argument that is just a filename rather than a complete path. In this case, the functions search for an executable file of that name along the directories listed in `os.environ['PATH']`. The other functions require `path` to be a complete path to the executable file for the new program.

- Are arguments for the new program accepted as a single sequence argument `args` to the function or as separate arguments to the function? Functions whose names start with `execv` take a single argument `args` that is the sequence of the arguments to use for the new program. Functions whose names start with `execl` take the new program's arguments as separate arguments (`execle`, in particular, uses its last argument as the environment for the new program).

- Is the new program's environment accepted as an explicit mapping argument `env` to the function, or is `os.environ` implicitly used? `execle`, `execve`, and `execvpe` take an argument `env` that is a mapping to use as the new program's environment (keys and values must be strings), while the other functions use `os.environ` for this purpose.

Each `exec` function uses the first item in `args` as the name under which the new program is told it's running (for example, `argv[0]` in a C program's `main`); only `args[1:]` are arguments proper to the new program.

popen

`popen(cmd,mode='r',buffering=-1)`

Runs the string command `cmd` in a new process `P` and returns a file-like object `f` that wraps a pipe to `P`'s standard input or from `P`'s standard output (depending on `mode`). `mode` and `buffering` have the same meaning as for Python's `open` function, covered in "Creating a "file" Object with io.open". When `mode` is `'r'` (the default), `f` is read-only and wraps `P`'s standard output. When mode is `'w'`, `f` is write-only and wraps `P`'s standard input.

The key difference of `f` with respect to other file-like objects is the behavior of method `f.close`. `f.close()` waits for `P` to terminate and returns `None`, as `close` methods of file-like objects normally do, when `P`'s termination is successful. However, if the operating system associates an integer error code `c` with `P`'s termination, indicating that `P`'s termination was unsuccessful, `f.close()` returns `c`. On Windows systems, `c` is a signed integer return code from the child process.

spawnv, spawnve	`spawnv(mode,path,args` `)` `spawnve(mode,path,args,env)`
	
	These functions run the program indicated by `path` in a new process `P`, with the arguments passed as sequence `args`. `spawnve` uses mapping `env` as `P`'s environment (both keys and values must be strings), while `spawnv` uses `os.environ` for this purpose. On Unix-like platforms only, there are other variations of `os.spawn`, corresponding to variations of `os.exec`, but `spawnv` and `spawnve` are the only two that also exist on Windows.
	
	`mode` must be one of two attributes supplied by the `os` module: `os.P_WAIT` indicates that the calling process waits until the new process terminates, while `os.P_NOWAIT` indicates that the calling process continues executing simultaneously with the new process. When `mode` is `os.P_WAIT`, the function returns the termination code `c` of `P`: `0` indicates successful termination, `c` less than `0` indicates `P` was killed by a *signal*, and `c` greater than `0` indicates normal but unsuccessful termination. When `mode` is `os.P_NOWAIT`, the function returns `P`'s process ID (or on Windows, `P`'s process handle). There is no cross-platform way to use `P`'s ID or handle; platform-specific ways (not covered further in this book) include the function `os.waitpid` on Unix-like platforms and third-party extension package PyWin32 on Windows.
	
	For example, your interactive program can give the user a chance to edit a text file that your program is about to read and use. You must have previously determined the full path to the user's favorite text editor, such as *c:\\windows\\notepad.exe* on Windows or */usr/bin/vim* on a Unix-like platform. Say that this path string is bound to variable `editor` and the path of the text file you want to let the user edit is bound to `textfile`:
	
	`import os`
	`os.spawnv(os.P_WAIT, editor, [editor, textfile])`
	
	The first item of the argument `args` is passed to the program being spawned as "the name under which the program is being invoked." Most programs don't look at this, so you can usually place just about any string here. Just in case the editor program does look at this special first argument, passing the same string `editor` that is used as the second argument to `os.spawnv` is the simplest and most effective approach.

system	`system(cmd)` Runs the string command `cmd` in a new process and returns `0` when the new process terminates successfully. When the new process terminates unsuccessfully, `system` returns an integer error code not equal to `0`. (Exactly what error codes may be returned depends on the command you're running: there's no widely accepted standard for this.)

Note that `popen` is deprecated in v2 (although it was never removed), then reimplemented in v3 as a simple wrapper over `subprocess.Popen`.

The Subprocess Module

The subprocess module supplies one very broad class: `Popen`, which supports many diverse ways for your program to run another program.

Popen `class Popen( args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0)`

`Popen` starts a subprocess to run a distinct program, and creates and returns an object `p`, representing that subprocess. The `args` mandatory argument and the many optional named arguments control details of how the subprocess is to run.

When any exception occurs during the subprocess creation (before the distinct program starts), `Popen` re-raises that exception in the calling process with the addition of an attribute named `child_traceback`, which is the Python traceback object for the subprocess. Such an exception would normally be an instance of `OSError` (or possibly `TypeError` or `ValueError` to indicate that you've passed to `Popen` an argument that's invalid in type or value).

What to run, and how: args, executable, shell

`args` is a sequence (normally a list) of strings: the first item is the path to the program to execute, and the following items, if any, are arguments to pass to the program (`args` can also be just a string, when you don't need to pass arguments). `executable`, when not `None`, overrides `args` in determining which program to execute. When `shell` is true, `executable` specifies which shell to use to run the subprocess; when `shell` is true and `executable` is `None`, the shell used is */bin/sh* on Unix-like systems (on Windows, it's `os.environ['COMSPEC']`).

Subprocess files: stdin, stdout, stderr, bufsize, universal_newlines, close_fds

`stdin`, `stdout`, and `stderr` specify the subprocess's standard input, output, and error files, respectively. Each may be `PIPE`, which creates a new pipe to/from the subprocess; `None`, meaning that the subprocess is to use the same file as this ("parent") process; or a file object (or file descriptor) that's already suitably open (for reading, for the standard input; for writing, for the standard output and standard error). `stderr` may also be `STDOUT`, meaning that the subprocess's standard error must use the same file as its standard output. `bufsize` controls the buffering of these files (unless they're already open), with the same semantics as the same argument to the `open` function covered in "Creating a "file" Object with io.open" (the default, `0`, means "unbuffered"). When `universal_newlines` is true, `stdout` and `stderr` (unless they're already open) are opened in "universal newlines" (`'rU'`) mode, covered in "mode". When `close_fds` is true, all other files (apart from standard input, output, and error) are closed in the subprocess before the subprocess's program or shell is executed.

Other arguments: preexec_fn, cwd, env, startupinfo, creationflags

When `preexec_fn` is not `None`, it must be a function or other callable object, and gets called in the subprocess before the subprocess's program or shell is executed (only on Unix-like system, where the call happens after `fork` and before `exec`).

When `cwd` is not `None`, it must be a string that gives the path to an existing directory; the current directory gets changed to `cwd` in the subprocess before the subprocess's program or shell is executed.

When `env` is not `None`, it must be a mapping (normally a dictionary) with strings as both keys and values, and fully defines the environment for the new process.

`startupinfo` and `creationflags` are Windows-only arguments passed to the `CreateProcess` Win32 API call used to create the subprocess, for Windows-specific purposes (they are not covered further in this book, which focuses on cross-platform uses of Python).

Attributes of subprocess.Popen instances

An instance `p` of class `Popen` supplies the following attributes:

`args`

(v3 only) `Popen`'s `args` argument (string or sequence of strings).

`pid`

The process ID of the subprocess.

`returncode`

`None` to indicate that the subprocess has not yet exited; otherwise, an integer: `0` for successful termination, `>0` for termination with an error code, or `<0` if the subprocess was killed by a signal.

`stderr`, `stdin`, `stdout`

When the corresponding argument to `Popen` was `subprocess.PIPE`, each of these attributes is a file object wrapping the corresponding pipe; otherwise, each of these attributes is `None`. Use the `communicate` method of `p`, not reading and writing to/from these file objects, to avoid possible deadlocks.

Methods of subprocess.Popen instances

An instance `p` of class `Popen` supplies the following methods.

communicate	`p.communicate(input=None)` Sends the string `input` as the subprocess's standard input (when `input` is not `None`), then reads the subprocess's standard output and error files into in-memory strings `so` and `se` until both files are finished, and finally waits for the subprocess to terminate and returns a pair (two-item tuple) `(so, se)`. In v3, also accepts an optional `timeout` argument.
poll	`p.poll()` Checks if the subprocess has terminated, then returns `p.returncode`.
wait	`p.wait()` Waits for the subprocess to terminate, then returns `p.returncode`. In v3, also accepts an optional `timeout` argument.

The mmap Module

The `mmap` module supplies memory-mapped file objects. An `mmap` object behaves similarly to a bytestring, so you can often pass an `mmap` object where a bytestring is expected. However, there are differences:

- An `mmap` object does not supply the methods of a string object.
- An `mmap` object is mutable, while string objects are immutable.
- An `mmap` object also corresponds to an open file and behaves polymorphically to a Python file object (as covered in "File-Like Objects and Polymorphism").

An `mmap` object `m` can be indexed or sliced, yielding bytestrings. Since `m` is mutable, you can also assign to an indexing or slicing of `m`. However, when you assign to a slice of `m`, the righthand side of the assignment statement must be a bytestring of exactly the same length as the slice you're assigning to. Therefore, many of the useful tricks available with list slice assignment (covered in "Modifying a list") do not apply to `mmap` slice assignment.

The `mmap` module supplies a factory function that is slightly different on Unix-like systems and on Windows:

mmap `mmap(filedesc,length,tagname='', access=None, offset=None)` *Windows*

`mmap(filedesc,length,flags=MAP_SHARED,`
`    =PROT_READ | PROT_WRITE,`
`protaccess=None, offset=0)` *Unix*

Creates and returns an `mmap` object `m` that maps into memory the first `length` bytes of the file indicated by file descriptor `filedesc`. `filedesc` must normally be a file descriptor opened for both reading and writing (except, on Unix-like platforms, when the argument `prot` requests only reading or only writing). (File descriptors are covered in "File Descriptor Operations".) To get an `mmap` object `m` for a Python file object `f`, use `m=mmap.mmap(f.fileno(),length)`. `filedesc` can be `-1` to map anonymous memory.

On Windows, all memory mappings are readable and writable, and shared among processes, so that all processes with a memory mapping on a file can see changes made by other such processes. On Windows only, you can pass a string `tagname` to give an explicit tag name for the memory mapping. This tag name lets you have several memory mappings on the same file, but this is rarely necessary. Calling `mmap` with only two arguments has the advantage of keeping your code portable between Windows and Unix-like platforms.

On Unix-like platforms only, you can pass `mmap.MAP_PRIVATE` as `flags` to get a mapping that is private to your process and copy-on-write. `mmap.MAP_SHARED`, the default, gets a mapping that is shared with other processes so that all processes mapping the file can see changes made by one process (same as on Windows). You can pass `mmap.PROT_READ` as the `prot` argument to get a mapping that you can only read, not write. Passing `mmap.PROT_WRITE` gets a mapping that you can only write, not read. The bitwise-OR `mmap.PROT_READ|mmap.PROT_WRITE` gets a mapping you can both read and write.

You can pass named argument `access`, instead of `flags` and `prot` (it's an error to pass both `access` and either or both of the other two arguments). The value for `access` can be one of `ACCESS_READ` (read-only), `ACCESS_WRITE` (write-through, the default on Windows), or `ACCESS_COPY` (copy-on-write).

You can pass named argument `offset` to start the mapping after the beginning of the file; offset must be an `int`, `>=0`, multiple of `ALLOCATIONGRANULARITY` (or, on Unix, of `PAGESIZE`).

Methods of mmap Objects

An `mmap` object `m` supplies the following methods:

close `m.close()`

Closes the file of `m`.

find	`m.find(sub,start=0, end=None)`	Returns the lowest `i>=start` such that `sub==m[i:i+len(sub)]` (and `i+len(sub)-1<=end`, ng when you pass `end`). If no such `i` exists, `m.find` returns `-1`. This is the same behavior as the `find` method of string objects, covered in Table 8-1.
flush	`m.flush([offset,n])`	Ensures that all changes made to `m` also exist on `m`'s file. Until you call `m.flush`, it's uncertain whether the file reflects the current state of `m`. You can pass a starting byte offset `offset` and a byte count `n` to limit the flushing effect's guarantee to a slice of `m`. Pass both arguments, or neither: it is an error to call `m.flush` with exactly one argument.
move	`m.move(dstoff,srcoff,n)`	Like the slice assignment `m[dstoff:dstoff+n]=m[srcoff:srcoff+n]`, but potentially faster. The source and destination slices can overlap. Apart from such potential overlap, `move` does not affect the source slice (i.e., the `move` method *copies* bytes but does not *move* them, despite the method's name).
read	`m.read(n)`	Reads and returns a string `s` containing up to `n` bytes starting from `m`'s file pointer, then advances `m`'s file pointer by `len(s)`. If there are fewer than `n` bytes between `m`'s file pointer and `m`'s length, returns the bytes available. In particular, if `m`'s file pointer is at the end of `m`, returns the empty string `''`.
read_byte	`m.read_byte()`	Returns a byte string of length `1` containing the byte at `m`'s file pointer, then advances `m`'s file pointer by `1`. `m.read_byte()` is similar to `m.read(1)`. However, if `m`'s file pointer is at the end of `m`, `m.read(1)` returns the empty string `''`, while `m.read_byte()` raises a `ValueError` exception.
readline	`m.readline()`	Reads and returns one line from the file of `m`, from `m`'s current file pointer up to the next `'\n'`, included (or up to the end of `m` if there is no `'\n'`), then advances `m`'s file pointer to point just past the bytes just read. If `m`'s file pointer is at the end of `m`, `readline` returns the empty string `''`.
resize	`m.resize(n)`	Changes the length of `m` so that `len(m)` becomes `n`. Does not affect the size of `m`'s file. `m`'s length and the file's size are independent. To set `m`'s length to be equal to the file's size, call `m.resize(m.size())`. If `m`'s length is larger than the file's size, `m` is padded with null bytes (`\x00`).
rfind	`rfind(sub, start=0, end=None)`	Returns the highest `i>=start` such that `sub==m[i:i+len(sub)]` (and `i+len(sub)-1<=end`, when you pass `end`). If no such `i` exists, `m.rfind` returns `-1`. This is the same behavior as the `rfind` method of string objects, covered in Table 8-1.

seek	`m.seek(pos,how=0)`	Sets the file pointer of `m` to the integer byte offset `pos`. `how` indicates the reference point (point `0`): when `how` is `0`, the reference point is the start of the file; when `1`, `m`'s current file pointer; when `2`, the end of `m`. A `seek` that tries to set `m`'s file pointer to a negative byte offset, or to a positive offset beyond `m`'s length, raises a `ValueError` exception.
size	`m.size()`	Returns the length (number of bytes) of the file of `m`, not the length of `m` itself. To get the length of `m`, use `len(m)`.
tell	`m.tell()`	Returns the current position of the file pointer of `m` as a byte offset from the start of `m`'s file.
write	`m.write(str)`	Writes the bytes in `str` into `m` at the current position of `m`'s file pointer, overwriting the bytes that were there, and then advances `m`'s file pointer by `len(str)`. If there aren't at least `len(str)` bytes between `m`'s file pointer and the length of `m`, `write` raises a `ValueError` exception.
write_byte	`m.write_byte(byte)`	Writes `byte`, which must be an int in v3, a single-character bytestring in v2, into mapping `m` at the current position of `m`'s file pointer, overwriting the byte that was there, and then advances `m`'s file pointer by `1`. When `x` is a single-character bytestring in v2, `m.write_byte(x)` is similar to `m.write(x)`. However, if `m`'s file pointer is at the end of `m`, `m.write_byte(x)` silently does nothing, while `m.write(x)` raises a `ValueError` exception. Note that this is the reverse of the relationship between `read` and `read_byte` at end-of-file: `write` and `read_byte` raise `ValueError`, while `read` and `write_byte` don't.

Using mmap Objects for IPC

The way in which processes communicate using `mmap` is similar to how IPC uses files: one process writes data and another process later reads the same data back. Since an `mmap` object rests on an underlying file, you can also have some processes doing I/O directly on the file (as covered in "The io Module"), while others use `mmap` to access the same file. You can choose between `mmap` and I/O on file objects on the basis of convenience: the functionality is the same, and performance is roughly equivalent. For example, here is a simple program that uses file I/O to make the contents of a file equal to the last line interactively typed by the user:

```
fileob = open('xxx','w')
while True:
    data = input('Enter some text:')
    fileob.seek(0)
    fileob.write(data)
    fileob.truncate()
    fileob.flush()
```

And here is another simple program that, when run in the same directory as the former, uses `mmap` (and the `time.sleep` function, covered in Table 12-2) to check every second for changes to the file and print out the file's new contents:

```
import mmap, os, time
mx = mmap.mmap(os.open('xxx',os.O_RDWR), 1)
last = None
while True:
    mx.resize(mx.size())
    data = mx[:]
    if data != last:
        print(data)
        last = data
    time.sleep(1)
```