
multiprocessing Documentation

Release 3.5.0.0

Python Software Foundation

August 04, 2016

1 multiprocessing — Process-based parallelism	3
1.1 Introduction	3
1.2 Reference	7
1.3 Programming guidelines	29
1.4 Examples	32
2 Glossary	53
3 Indices and tables	55
Python Module Index	57

Contents:

multiprocessing — Process-based parallelism

1.1 Introduction

multiprocessing is a package that supports spawning processes using an API similar to the `threading` module. The *multiprocessing* package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the *multiprocessing* module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

Note: Some of this package’s functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the `multiprocessing.synchronize` module will be disabled, and attempts to import it will result in an `ImportError`. See [:issue:‘3770’](#) for additional information.

Note: Functionality within this package requires that the `__main__` module be importable by the children. This is covered in *Programming guidelines* however it is worth pointing out here. This means that some examples, such as the `multiprocessing.Pool` examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)

1.1.1 The `Process` class

In *multiprocessing*, processes are spawned by creating a *Process* object and then calling its `start()` method. *Process* follows the API of `threading.Thread`. A trivial example of a multiprocessing program is

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why (on Windows) the `if __name__ == '__main__'` part is necessary, see [Programming guidelines](#).

1.1.2 Exchanging objects between processes

`multiprocessing` supports two types of communication channel between processes:

Queues

The `Queue` class is a near clone of `Queue.Queue`. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe, but note that they must never be instantiated as a side effect of importing a module: this can lead to a deadlock! (see `threaded-imports`)

Pipes

The `Pipe()` function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv()) # prints "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by `Pipe()` represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

1.1.3 Synchronization between processes

`multiprocessing` contains equivalents of all the synchronization primitives from threading. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    print('hello world', i)
    l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Without using the lock output from the different processes is liable to get all mixed up.

1.1.4 Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then `multiprocessing` provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using `Value` or `Array`. For example, the following code

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
```

```

    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])

```

will print

```

3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

```

The 'd' and 'i' arguments used when creating num and arr are typecodes of the kind used by the array module: 'd' indicates a double precision float and 'i' indicates a signed integer. These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the *multiprocessing.sharedctypes* module which supports the creation of arbitrary ctypes objects allocated from shared memory.

Server process

A manager object returned by `Manager()` controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types `list`, `dict`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Queue`, `Value` and `Array`. For example,

```

from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    manager = Manager()

    d = manager.dict()
    l = manager.list(range(10))

    p = Process(target=f, args=(d, l))
    p.start()
    p.join()

    print(d)
    print(l)

```

will print

```

{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

1.1.5 Using a pool of workers

The `Pool` class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)           # start 4 worker processes
    result = pool.apply_async(f, [10]) # evaluate "f(10)" asynchronously
    print(result.get(timeout=1))       # prints "100" unless your computer is *very* slow
    print(pool.map(f, range(10)))     # prints "[0, 1, 4, ..., 81]"
```

1.2 Reference

The `multiprocessing` package mostly replicates the API of the `threading` module.

1.2.1 Process and exceptions

class `multiprocessing.Process` (`[group[, target[, name[, args[, kwargs]]]]`], `daemon=None`)

Process objects represent activity that is run in a separate process. The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. `group` should always be `None`; it exists solely for compatibility with `threading.Thread`. `target` is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. `name` is the process name. By default, a unique name is constructed of the form `'Process-N1:N2:...:Nk'` where `N1,N2,...,Nk` is a sequence of integers whose length is determined by the *generation* of the process. `args` is the argument tuple for the target invocation. `kwargs` is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only `daemon` argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to `target`.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`Process.__init__()`) before doing anything else to the process.

Changed in version 3.3: Added the `daemon` argument.

run()

Method representing the process's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

start ()

Start the process's activity.

This must be called at most once per process object. It arranges for the object's `run ()` method to be invoked in a separate process.

join ([*timeout*])

If the optional argument *timeout* is `None` (the default), the method blocks until the process whose `join ()` method is called terminates. If *timeout* is a positive number, it blocks at most *timeout* seconds.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name

The process's name.

The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name. The initial name is set by the constructor.

is_alive ()

Return whether the process is alive.

Roughly, a process object is alive from the moment the `start ()` method returns until the child process terminates.

daemon

The process's daemon flag, a Boolean value. This must be set before `start ()` is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemon child processes.

Note that a daemon process is not allowed to create child processes. Otherwise a daemon process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemon processes have exited.

In addition to the `Threading.Thread` API, `Process` objects also support the following attributes and methods:

pid

Return the process ID. Before the process is spawned, this will be `None`.

exitcode

The child's exit code. This will be `None` if the process has not yet terminated. A negative value `-N` indicates that the child was terminated by signal `N`.

authkey

The process's authentication key (a byte string).

When `multiprocessing` is initialized the main process is assigned a random string using `os.random ()`.

When a `Process` object is created, it will inherit the authentication key of its parent process, although this may be changed by setting `authkey` to another byte string.

See *Authentication keys*.

sentinel

A numeric handle of a system object which will become "ready" when the process ends.

You can use this value if you want to wait on several events at once using `multiprocessing.connection.wait()`. Otherwise calling `join()` is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On Unix, this is a file descriptor usable with primitives from the `select` module.

New in version 3.3.

terminate()

Terminate the process. On Unix this is done using the `SIGTERM` signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated – they will simply become orphaned.

Warning: If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

Note that the `start()`, `join()`, `is_alive()`, `terminate()` and `exit_code` methods should only be called by the process that created the process object.

Example usage of some of the methods of `Process`:

exception multiprocessing.BufferTooShort

Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of `BufferTooShort` then `e.args[0]` will give the message as a byte string.

1.2.2 Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use `Pipe()` (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The `Queue`, `SimpleQueue` and `JoinableQueue` types are multi-producer, multi-consumer FIFO queues modelled on the `Queue.Queue` class in the standard library. They differ in that `Queue` lacks the `task_done()` and `join()` methods introduced into Python 2.5's `queue.Queue` class.

If you use `JoinableQueue` then you **must** call `JoinableQueue.task_done()` for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see *Managers*.

Note: `multiprocessing` uses the usual `Queue.Empty` and `Queue.Full` exceptions to signal a timeout. They are not available in the `multiprocessing` namespace so you need to import them from `queue`.

Warning: If a process is killed using `Process.terminate()` or `os.kill()` while it is trying to use a `Queue`, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

Warning: As mentioned above, if a child process has put items on a queue (and it has not used `JoinableQueue.cancel_join_thread()`), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See *Programming guidelines*.

For an example of the usage of queues for interprocess communication see *Examples*.

`multiprocessing.Pipe([duplex])`

Returns a pair (`conn1`, `conn2`) of `Connection` objects representing the ends of a pipe.

If `duplex` is `True` (the default) then the pipe is bidirectional. If `duplex` is `False` then the pipe is unidirectional: `conn1` can only be used for receiving messages and `conn2` can only be used for sending messages.

class `multiprocessing.Queue([maxsize])`

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual `Queue.Empty` and `Queue.Full` exceptions from the standard library's `Queue` module are raised to signal timeouts.

`Queue` implements all the methods of `Queue.Queue` except for `task_done()` and `join()`.

qsize()

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise `NotImplementedError` on Unix platforms like macOS where `sem_getvalue()` is not implemented.

empty()

Return `True` if the queue is empty, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

full()

Return `True` if the queue is full, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(obj[, block[, timeout]])

Put `obj` into the queue. If the optional argument `block` is `True` (the default) and `timeout` is `None` (the default), block if necessary until a free slot is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises the `queue.Full` exception if no free slot was available within that time. Otherwise (`block` is `False`), put an item on the queue if a free slot is immediately available, else raise the `queue.Full` exception (`timeout` is ignored in that case).

put_nowait(obj)

Equivalent to `put(obj, False)`.

get([block[, timeout]])

Remove and return an item from the queue. If optional args `block` is `True` (the default) and `timeout` is `None` (the default), block if necessary until an item is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises the `queue.Empty` exception if no item was available within that time. Otherwise (`block` is `False`), return an item if one is immediately available, else raise the `queue.Empty` exception (`timeout` is ignored in that case).

get_nowait()

get_no_wait()

Equivalent to `get(False)`.

`multiprocessing.Queue` has a few additional methods not found in `queue.Queue`. These methods are usually unnecessary for most code:

close()

Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()

Join the background thread. This can only be used after `close()` has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue's background thread. The process can call `cancel_join_thread()` to make `join_thread()` do nothing.

cancel_join_thread()

Prevent `join_thread()` from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see `join_thread()`.

class multiprocessing.SimpleQueue

It is a simplified `Queue` type, very close to a locked `Pipe`.

empty()

Return True if the queue is empty, False otherwise.

get()

Remove and return an item from the queue.

put(item)

Put `item` into the queue.

class multiprocessing.JoinableQueue([maxsize])

`JoinableQueue`, a `Queue` subclass, is a queue which additionally has `task_done()` and `join()` methods.

task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

join()

Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

1.2.3 Miscellaneous

multiprocessing.active_children()

Return list of all live children of the current process.

Calling this has the side affect of “joining” any processes which have already finished.

`multiprocessing.cpu_count()`

Return the number of CPUs in the system. May raise `NotImplementedError`.

`multiprocessing.current_process()`

Return the `Process` object corresponding to the current process.

An analogue of `threading.current_thread()`.

`multiprocessing.freeze_support()`

Add support for when a program which uses `multiprocessing` has been frozen to produce a Windows executable. (Has been tested with `py2exe`, `PyInstaller` and `cx_Freeze`.)

One needs to call this function straight after the `if __name__ == '__main__':` line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise `RuntimeError`.

If the module is being run normally by the Python interpreter then `freeze_support()` has no effect.

`multiprocessing.set_executable()`

Sets the path of the Python interpreter to use when starting a child process. (By default `sys.executable` is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes. (Windows only)

Note: `multiprocessing` contains no analogues of `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`, or `threading.local`.

1.2.4 Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using `Pipe()` – see also *Listeners and Clients*.

class `multiprocessing.Connection`

send (*obj*)

Send an object to the other end of the connection which should be read using `recv()`.

The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a `ValueError` exception.

recv()

Return an object sent from the other end of the connection using `send()`. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

fileno()

Return the file descriptor or handle used by the connection.

close()

Close the connection.

This is called automatically when the connection is garbage collected.

poll([timeout])

Return whether there is any data available to be read.

If `timeout` is not specified then it will return immediately. If `timeout` is a number then this specifies the maximum time in seconds to block. If `timeout` is `None` then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using `multiprocessing.connection.wait()`.

send_bytes(buffer[, offset[, size]])

Send byte data from an object supporting the buffer interface as a complete message.

If `offset` is given then data is read from that position in `buffer`. If `size` is given then that many bytes will be read from `buffer`. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a `ValueError` exception

recv_bytes([maxlength])

Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end has closed.

If `maxlength` is specified and the message is longer than `maxlength` then `OSError` is raised and the connection will no longer be readable.

Changed in version 3.3: This function used to raise a `IOError`, which is now an alias of `OSError`.

recv_bytes_into(buffer[, offset])

Read into `buffer` a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

`buffer` must be an object satisfying the writable buffer interface. If `offset` is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of `buffer` (in bytes).

If the buffer is too short then a `BufferTooShort` exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

For example:

Warning: The `Connection.recv()` method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message. Therefore, unless the connection object was produced using `Pipe()` you should only use the `recv()` and `send()` methods after performing some sort of authentication. See [Authentication keys](#).

Warning: If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

1.2.5 Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for `threading` module.

Note that one can also create synchronization primitives by using a manager object – see *Managers*.

class `multiprocessing.BoundedSemaphore` (`[value]`)

A bounded semaphore object: a clone of `threading.BoundedSemaphore`.

(On macOS, this is indistinguishable from *Semaphore* because `sem_getvalue()` is not implemented on that platform).

class `multiprocessing.Condition` (`[lock]`)

A condition variable: a clone of `threading.Condition`.

If *lock* is specified then it should be a *Lock* or *RLock* object from *multiprocessing*.

Changed in version 3.3: The `wait_for()` method was added.

class `multiprocessing.Event`

A clone of `threading.Event`. This method returns the state of the internal semaphore on exit, so it will always return `True` except if a timeout is given and the operation times out.

Changed in version 3.1: Previously, the method always returned `None`.

class `multiprocessing.Lock`

A non-recursive lock object: a clone of `threading.Lock`.

class `multiprocessing.RLock`

A recursive lock object: a clone of `threading.RLock`.

class `multiprocessing.Semaphore` (`[value]`)

A semaphore object: a clone of `threading.Semaphore`.

Note: On macOS, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

Note: If the `SIGINT` signal generated by Ctrl-C arrives while the main thread is blocked by a call to `BoundedSemaphore.acquire()`, `Lock.acquire()`, `RLock.acquire()`, `Semaphore.acquire()`, `Condition.acquire()` or `Condition.wait()` then the call will be immediately interrupted and `KeyboardInterrupt` will be raised.

This differs from the behaviour of `threading` where `SIGINT` will be ignored while the equivalent blocking calls are in progress.

1.2.6 Shared ctypes Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

`multiprocessing.Value` (`typecode_or_type`, `*args` [`, lock`])

Return a `ctypes` object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object.

`typecode_or_type` determines the type of the returned object: it is either a `ctypes` type or a one character typecode of the kind used by the `array` module. `*args` is passed on to the constructor for the type.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that *lock* is a keyword-only argument.

`multiprocessing.Array` (*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the `array` module. If *size_or_initializer* is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that *lock* is a keyword only argument.

Note that an array of `ctypes.c_char` has *value* and *raw* attributes which allow one to use it to store and retrieve strings.

The multiprocessing.sharedctypes module

The `multiprocessing.sharedctypes` module provides functions for allocating ctypes objects from shared memory which can be inherited by child processes.

Note: Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

`multiprocessing.sharedctypes.RawArray` (*typecode_or_type*, *size_or_initializer*)

Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the `array` module. If *size_or_initializer* is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use `Array()` instead to make sure that access is automatically synchronized using a lock.

`multiprocessing.sharedctypes.RawValue` (*typecode_or_type*, **args*)

Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the `array` module. **args* is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use `Value()` instead to make sure that access is automatically synchronized using a lock.

Note that an array of `ctypes.c_char` has *value* and *raw* attributes which allow one to use it to store and retrieve strings – see documentation for ctypes.

`multiprocessing.sharedctypes.Array` (*typecode_or_type*, *size_or_initializer*, **args*[, *lock*])

The same as `RawArray()` except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that *lock* is a keyword-only argument.

`multiprocessing.sharedctypes.Value` (*typecode_or_type*, **args*[, *lock*])

The same as `RawValue()` except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that *lock* is a keyword-only argument.

`multiprocessing.sharedctypes.copy` (*obj*)

Return a ctypes object allocated from shared memory which is a copy of the ctypes object *obj*.

`multiprocessing.sharedctypes.synchronized` (*obj*[, *lock*])

Return a process-safe wrapper object for a ctypes object which uses *lock* to synchronize access. If *lock* is `None` (the default) then a `multiprocessing.RLock` object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: `get_obj()` returns the wrapped object and `get_lock()` returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of `ctypes.Structure`.)

ctypes	sharedctypes using type	sharedctypes using typecode
<code>c_double(2.4)</code>	<code>RawValue(c_double, 2.4)</code>	<code>RawValue('d', 2.4)</code>
<code>MyStruct(4, 6)</code>	<code>RawValue(MyStruct, 4, 6)</code>	
<code>(c_short * 7)()</code>	<code>RawArray(c_short, 7)</code>	<code>RawArray('h', 7)</code>
<code>(c_int * 3)(9, 2, 8)</code>	<code>RawArray(c_int, (9, 2, 8))</code>	<code>RawArray('i', (9, 2, 8))</code>

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', 'hello world', lock=lock)
```

```
A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

p = Process(target=modify, args=(n, x, s, A))
p.start()
p.join()

print(n.value)
print(x.value)
print(s.value)
print([(a.x, a.y) for a in A])
```

The results printed are

```
49
0.11111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

1.2.7 Managers

Managers provide a way to create data which can be shared between different processes. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

`multiprocessing.Manager()`

Returns a started *SyncManager* object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the `multiprocessing.managers` module:

class `multiprocessing.managers.BaseManager` (`[address[, authkey]]`)

Create a BaseManager object.

Once created one should call `start()` or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

`address` is the address on which the manager process listens for new connections. If `address` is `None` then an arbitrary one is chosen.

`authkey` is the authentication key which will be used to check the validity of incoming connections to the server process. If `authkey` is `None` then `current_process().authkey`. Otherwise `authkey` is used and it must be a string.

start (`[initializer[, initargs]]`)

Start a subprocess to start the manager. If `initializer` is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

get_server ()

Returns a `Server` object which represents the actual server under the control of the `Manager`. The `Server` object supports the `serve_forever()` method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey='abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` additionally has an `address` attribute.

connect ()

Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey='abc')
>>> m.connect()
```

shutdown ()

Stop the process used by the manager. This is only available if `start ()` has been used to start the server process.

This can be called multiple times.

register (typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]])

A classmethod which can be used for registering a type or callable with the manager class.

typeid is a “type identifier” which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be created using the `from_address ()` classmethod or if the *create_method* argument is `False` then this can be left as `None`.

proxytype is a subclass of `BaseProxy` which is used to create proxies for shared objects with this *typeid*. If `None` then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this *typeid* should be allowed to access using `BaseProxy._callMethod ()`. (If *exposed* is `None` then `proxytype._exposed_` is used instead if it exists.) In the case where no exposed list is specified, all “public methods” of the shared object will be accessible. (Here a “public method” means any attribute which has a `__call__ ()` method and whose name does not begin with `'_'`.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to *typeid* strings. (If *method_to_typeid* is `None` then `proxytype._method_to_typeid_` is used instead if it exists.) If a method’s name is not a key of this mapping or if the mapping is `None` then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name *typeid* which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is `True`.

`BaseManager` instances also have one read-only property:

address

The address used by the manager.

class multiprocessing.managers.SyncManager

A subclass of `BaseManager` which can be used for the synchronization of processes. Objects of this type are returned by `multiprocessing.Manager ()`.

It also supports creation of shared lists and dictionaries.

BoundedSemaphore ([value])

Create a shared `threading.BoundedSemaphore` object and return a proxy for it.

Condition ([lock])

Create a shared `threading.Condition` object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a `threading.Lock` or `threading.RLock` object.

Changed in version 3.3: The `wait_for ()` method was added.

Event ()

Create a shared `threading.Event` object and return a proxy for it.

Lock ()

Create a shared `threading.Lock` object and return a proxy for it.

Namespace ()

Create a shared `Namespace` object and return a proxy for it.

Queue ([maxsize])

Create a shared `Queue.Queue` object and return a proxy for it.

RLock ()

Create a shared `threading.RLock` object and return a proxy for it.

Semaphore ([value])

Create a shared `threading.Semaphore` object and return a proxy for it.

Array (typecode, sequence)

Create an array and return a proxy for it.

Value (typecode, value)

Create an object with a writable `value` attribute and return a proxy for it.

dict ()**dict (mapping)****dict (sequence)**

Create a shared `dict` object and return a proxy for it.

list ()**list (sequence)**

Create a shared `list` object and return a proxy for it.

Note: Modifications to mutable values or items in `dict` and `list` proxies will not be propagated through the manager, because the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# reassigning the dictionary, the proxy is notified of the change
lproxy[0] = d
```

Namespace objects

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `'_'` will be an attribute of the proxy and not an attribute of the referent:

Customized managers

To create one's own manager, one creates a subclass of `BaseManager` and uses the `register()` classmethod to register new types or callables with the manager class. For example:

```

from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    manager = MyManager()
    manager.start()
    maths = manager.Maths()
    print(maths.add(4, 3))          # prints 7
    print(maths.mul(7, 8))        # prints 56

```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```

>>> from multiprocessing.managers import BaseManager
>>> import queue
>>> queue = Queue.Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=(' ', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()

```

One client can access the server as follows:

```

>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')

```

Another client can also use it:

```

>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> Queue.get()
'hello'

```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```

>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('localhost', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()

```

1.2.8 Proxy Objects

A proxy is an object which *refers* to a shared object which lives (presumably) in a different process. The shared object is said to be the *referent* of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). A proxy can usually be used in most of the same ways that its referent can:

Notice that applying `str()` to a proxy will return the representation of the referent, whereas applying `repr()` will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. Note, however, that if a proxy is sent to the corresponding manager's process then unpickling it will produce the referent itself. This means, for example, that one shared object can contain a second:

Note: The proxy types in *multiprocessing* do nothing to support comparisons by value. So, for instance, we have:

One should just use a copy of the referent instead when making comparisons.

class multiprocessing.managers.**BaseProxy**

Proxy objects are instances of subclasses of *BaseProxy*.

_callmethod(*methodname*[, *args*[, *kwds*]])

Call and return the result of a method of the proxy's referent.

If *proxy* is a proxy whose referent is *obj* then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the *method_to_typeid* argument of *BaseManager.register()*.

If an exception is raised by the call, then is re-raised by `_callmethod()`. If some other exception is raised in the manager's process then this is converted into a `RemoteError` exception and is raised by `_callmethod()`.

Note in particular that an exception will be raised if *methodname* has not been *exposed*

An example of the usage of `_callmethod()`:

`__getvalue()`

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

`__repr__()`

Return a representation of the proxy object.

`__str__()`

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

1.2.9 Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the `Pool` class.

class `multiprocessing.Pool` (`[processes[, initializer[, initargs[, maxtasksperchild]]]`)

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by `cpu_count()` is used. If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

New in version 3.2: *maxtasksperchild* is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

Note: Worker processes within a `Pool` typically live for the complete duration of the `Pool`'s work queue. A frequent pattern found in other systems (such as Apache, `mod_wsgi`, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The *maxtasksperchild* argument to the `Pool` exposes this ability to the end user.

`apply` (`func`, `args`, `kwds`)

Call *func* with arguments *args* and keyword arguments *kwds*. It blocks until the result is ready. Given this blocks, `apply_async()` is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

`apply_async` (`func`, `args`, `kwds`, `callback`, `error_callback`)

A variant of the `apply()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

map (*func*, *iterable*[, *chunksizesize*])

A parallel equivalent of the *map* () built-in function (it supports only one *iterable* argument though). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksizesize* to a positive integer.

map_async (*func*, *iterable*[, *chunksizesize*[, *callback*[, *error_callback*]]])

A variant of the *map* () method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

imap (*func*, *iterable*[, *chunksizesize*])

A lazier version of *map* () .

The *chunksizesize* argument is the same as the one used by the *map* () method. For very long iterables using a large value for *chunksizesize* can make the job complete **much** faster than using the default value of 1.

Also if *chunksizesize* is 1 then the *next* () method of the iterator returned by the *imap* () method has an optional *timeout* parameter: *next* (timeout) will raise `multiprocessing.TimeoutError` if the result cannot be returned within *timeout* seconds.

imap_unordered (*func*, *iterable*[, *chunksizesize*])

The same as *imap* () except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be “correct”.)

starmap (*func*, *iterable*[, *chunksizesize*])

Like *map* () except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Hence an *iterable* of [(1,2), (3, 4)] results in [func(1,2), func(3,4)].

New in version 3.3.

starmap_async (*func*, *iterable*[, *chunksizesize*[, *callback*[, *error_back*]]])

A combination of *starmap* () and *map_async* () that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

New in version 3.3.

close ()

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

join()

Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

class multiprocessing.pool.AsyncResult

The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

get([timeout])

Return the result when it arrives. If `timeout` is not `None` and the result does not arrive within `timeout` seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

wait([timeout])

Wait until the result is available or until `timeout` seconds pass.

ready()

Return whether the call has completed.

successful()

Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)           # start 4 worker processes

    result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously
    print(result.get(timeout=1))        # prints "100" unless your computer is *very* slow

    print(pool.map(f, range(10)))      # prints "[0, 1, 4, ..., 81]"

    it = pool.imap(f, range(10))
    print(next(it))                    # prints "0"
    print(next(it))                    # prints "1"
    print(it.next(timeout=1))          # prints "4" unless your computer is *very* slow

    import time
    result = pool.apply_async(time.sleep, (10,))
    print(result.get(timeout=1))        # raises TimeoutError
```

1.2.10 Listeners and Clients

Usually message passing between processes is done using queues or by using `Connection` objects returned by `Pipe()`.

However, the `multiprocessing.connection` module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for *digest authentication* using the `hmac` module, and for polling multiple connections at the same time.

`multiprocessing.connection.deliver_challenge` (*connection*, *authkey*)

Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise `AuthenticationError` is raised.

`multiprocessing.connection.answerChallenge` (*connection*, *authkey*)

Receive a message, calculate the digest of the message using *authkey* as the key, and then send the digest back.

If a welcome message is not received, then `AuthenticationError` is raised.

`multiprocessing.connection.Client` (*address*[, *family*[, *authenticate*[, *authkey*]]])

Attempt to set up a connection to the listener which is using address *address*, returning a `Connection`.

The type of the connection is determined by *family* argument, but this can generally be omitted since it can usually be inferred from the format of *address*. (See [Address Formats](#))

If *authenticate* is `True` or *authkey* is a string then digest authentication is used. The key used for authentication will be either *authkey* or `current_process().authkey` if *authkey* is `None`. If authentication fails then `AuthenticationError` is raised. See [Authentication keys](#).

class `multiprocessing.connection.Listener` ([*address*[, *family*[, *backlog*[, *authenticate*[, *authkey*]]]]])

A wrapper for a bound socket or Windows named pipe which is 'listening' for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

Note: If an address of '0.0.0.0' is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use '127.0.0.1'.

family is the type of socket (or named pipe) to use. This can be one of the strings 'AF_INET' (for a TCP socket), 'AF_UNIX' (for a Unix domain socket) or 'AF_PIPE' (for a Windows named pipe). Of these only the first is guaranteed to be available. If *family* is `None` then the family is inferred from the format of *address*. If *address* is also `None` then a default is chosen. This default is the family which is assumed to be the fastest available. See [Address Formats](#). Note that if *family* is 'AF_UNIX' and *address* is `None` then the socket will be created in a private temporary directory created using `tempfile.mkstemp()`.

If the listener object uses a socket then *backlog* (1 by default) is passed to the `listen()` method of the socket once it has been bound.

If *authenticate* is `True` (`False` by default) or *authkey* is not `None` then digest authentication is used.

If *authkey* is a string then it will be used as the authentication key; otherwise it must be `None`.

If *authkey* is `None` and *authenticate* is `True` then `current_process().authkey` is used as the authentication key. If *authkey* is `None` and *authenticate* is `False` then no authentication is done. If authentication fails then `AuthenticationError` is raised. See [Authentication keys](#).

accept ()

Accept a connection on the bound socket or named pipe of the listener object and return a `Connection` object. If authentication is attempted and fails, then `AuthenticationError` is raised.

close ()

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address

The address which is being used by the Listener object.

last_accepted

The address from which the last accepted connection came. If this is unavailable then it is `None`.

`multiprocessing.connection.wait(object_list, timeout=None)`

Wait till an object in `object_list` is ready. Returns the list of those objects in `object_list` which are ready. If `timeout` is a float then the call blocks for at most that many seconds. If `timeout` is `None` then it will block for an unlimited period.

For both Unix and Windows, an object can appear in `object_list` if it is

- a readable `Connection` object;
- a connected and readable `socket.socket` object; or
- the `sentinel` attribute of a `Process` object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

Unix: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if `select.select()` is interrupted by a signal, it can raise `OSError` with an error number of `EINTR`, whereas `wait()` will not.

Windows: An item in `object_list` must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a `fileno()` method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

New in version 3.3.

The module defines two exceptions:

exception `multiprocessing.connection.AuthenticationError`

Exception raised when there is an authentication error.

Examples

The following server code creates a listener which uses 'secret password' as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)      # family is deduced to be 'AF_INET'
listener = Listener(address, authkey=b'secret password')

conn = listener.accept()
print('connection accepted from', listener.last_accepted)

conn.send([2.25, None, 'junk', float])

conn.send_bytes(b'hello')

conn.send_bytes(array('i', [42, 1729]))

conn.close()
listener.close()
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array
```

```

address = ('localhost', 6000)
conn = Client(address, authkey=b'secret password')

print(conn.recv())           # => [2.25, None, 'junk', float]

print(conn.recv_bytes())     # => 'hello'

arr = array('i', [0, 0, 0, 0, 0])
print(conn.recv_bytes_into(arr)) # => 8
print(arr)                   # => array('i', [42, 1729, 0, 0, 0])

conn.close()

```

The following code uses `wait()` to wait for messages from multiple processes at once:

```

import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it. This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)

```

Address Formats

- An 'AF_INET' address is a tuple of the form (hostname, port) where *hostname* is a string and *port* is an integer.
- An 'AF_UNIX' address is a string representing a filename on the filesystem.
- An 'AF_PIPE' address is a string of the form `r'\\.\pipe\PipeName'`. To use `Client()` to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\\ServerName\pipe\PipeName'` instead.

Note that any string beginning with two backslashes is assumed by default to be an `'AF_PIPE'` address rather than an `'AF_UNIX'` address.

1.2.11 Authentication keys

When one uses `Connection.recv()`, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore `Listener` and `Client()` use the `hmac` module to provide digest authentication.

An authentication key is a string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of `current_process().authkey` is used (see `Process`). This value will automatically be inherited by any `Process` object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using `os.urandom()`.

1.2.12 Logging

Some support for logging is available. Note, however, that the `logging` package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

`multiprocessing.get_logger()`

Returns the logger used by `multiprocessing`. If necessary, a new one will be created.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

`multiprocessing.log_to_stderr()`

This function performs a call to `get_logger()` but in addition to returning the logger created by `get_logger`, it adds a handler which sends output to `sys.stderr` using format `'[% (levelname) s/% (processName) s] % (message) s'`.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pypm-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

In addition to having these two logging functions, the `multiprocessing` also exposes two additional logging level attributes. These are `SUBWARNING` and `SUBDEBUG`. The table below illustrates where these fit in the normal level hierarchy.

Level	Numeric value
SUBWARNING	25
SUBDEBUG	5

For a full table of logging levels, see the logging module.

These additional logging levels are used primarily for certain debug messages within the multiprocessing module. Below is the same example as above, except with SUBDEBUG enabled:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(multiprocessing.SUBDEBUG)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pypm-...
[INFO/SyncManager-...] manager serving at '/.../pypm-djGBXN/listener-...'
>>> del m
[SUBDEBUG/MainProcess] finalizer calling ...
[INFO/MainProcess] sending shutdown message to manager
[DEBUG/SyncManager-...] manager received shutdown message
[SUBDEBUG/SyncManager-...] calling <Finalize object, callback=unlink, ...
[SUBDEBUG/SyncManager-...] finalizer calling <built-in function unlink> ...
[SUBDEBUG/SyncManager-...] calling <Finalize object, dead>
[SUBDEBUG/SyncManager-...] finalizer calling <function rmtree at 0x5aa730> ...
[INFO/SyncManager-...] manager exiting with exitcode 0
```

1.2.13 The multiprocessing.dummy module

`multiprocessing.dummy` replicates the API of `multiprocessing` but is no more than a wrapper around the threading module.

1.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using `multiprocessing`.

1.3.1 All platforms

Avoid shared state

As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives from the `threading` module.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the *same* proxy.)

Joining zombie processes

On Unix when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's `Process.is_alive()` will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

On Windows many types from *multiprocessing* need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the `Process.terminate()` method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using `Process.terminate()` on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the “feeder” thread to the underlying pipe. (The child process can call the `Queue.cancel_join_thread()` method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be automatically be joined.

An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines round (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On Unix a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock
```

```
def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing `sys.stdin` with a “file like object”

multiprocessing originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the `multiprocessing.Process._bootstrap()` method — this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.devnull)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace `sys.stdin()` with a “file-like object” with output buffering. This danger is that if multiple processes call `close()` on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see [:issue:‘5155’](#), [:issue:‘5313’](#) and [:issue:‘5331’](#)

1.3.2 Windows

Since Windows lacks `os.fork()` it has a few extra restrictions:

More picklability

Ensure that all arguments to `Process.__init__()` are picklable. This means, in particular, that bound or unbound methods cannot be used directly as the `target` argument on Windows — just define a function and use that instead.

Also, if you subclass `Process` then make sure that instances will be picklable when the `Process.start()` method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that `Process.start()` was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such as starting a new process).

For example, under Windows running the following module would fail with a `RuntimeError`:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the “entry point” of the program by using `if __name__ == '__main__':` as follows:

```
from multiprocessing import Process, freeze_support

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module’s `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

1.4 Examples

Demonstration of how to create and use customized managers and proxies:

```
#
# This module shows how to use arbitrary callables with a subclass of
# `BaseManager`.
#
# Copyright (c) 2006–2008, R Oudkerk
# All rights reserved.
#

from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
```

```

import operator

##

class Foo(object):
    def f(self):
        print 'you called Foo.f()'
    def g(self):
        print 'you called Foo.g()'
    def _h(self):
        print 'you called Foo._h()'

# A simple generator function
def baz():
    for i in xrange(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ('next', '__next__')
    def __iter__(self):
        return self
    def next(self):
        return self._callmethod('next')
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print '-' * 20

    f1 = manager.Foo1()
    f1.f()

```

```

f1.g()
assert not hasattr(f1, '_h')
assert sorted(f1._exposed_) == sorted(['f', 'g'])

print '-' * 20

f2 = manager.Foo2()
f2.g()
f2._h()
assert not hasattr(f2, 'f')
assert sorted(f2._exposed_) == sorted(['g', '_h'])

print '-' * 20

it = manager.baz()
for i in it:
    print '<%d>' % i,
print

print '-' * 20

op = manager.operator()
print 'op.add(23, 45) =', op.add(23, 45)
print 'op.pow(2, 94) =', op.pow(2, 94)
print 'op.getslice(range(10), 2, 6) =', op.getslice(range(10), 2, 6)
print 'op.repeat(range(5), 3) =', op.repeat(range(5), 3)
print 'op._exposed_ =', op._exposed_

##

if __name__ == '__main__':
    freeze_support()
    test()

```

Using Pool:

```

#
# A test of `multiprocessing.Pool` class
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
    )

```

```

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

def f(x):
    return 1.0 / (x-5.0)

def pow3(x):
    return x**3

def noop(x):
    pass

#
# Test code
#

def test():
    print 'cpu_count() = %d\n' % multiprocessing.cpu_count()

    #
    # Create pool
    #

    PROCESSES = 4
    print 'Creating pool with %d processes\n' % PROCESSES
    pool = multiprocessing.Pool(PROCESSES)
    print 'pool = %s' % pool
    print

    #
    # Tests
    #

    TASKS = [(mul, (i, 7)) for i in range(10)] + \
            [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print 'Ordered results using pool.apply_async():'
    for r in results:
        print '\t', r.get()
    print

    print 'Ordered results using pool.imap():'
    for x in imap_it:
        print '\t', x
    print

```

```

print 'Unordered results using pool.imap_unordered():'
for x in imap_unordered_it:
    print '\t', x
print

print 'Ordered results using pool.map() --- will block till complete:'
for x in pool.map(calculatestar, TASKS):
    print '\t', x
print

#
# Simple benchmarks
#

N = 100000
print 'def pow3(x): return x**3'

t = time.time()
A = map(pow3, xrange(N))
print '\tmap(pow3, xrange(%d)):\n\t\t%s seconds' % \
      (N, time.time() - t)

t = time.time()
B = pool.map(pow3, xrange(N))
print '\tpool.map(pow3, xrange(%d)):\n\t\t%s seconds' % \
      (N, time.time() - t)

t = time.time()
C = list(pool.imap(pow3, xrange(N), chunksize=N//8))
print '\tlist(pool.imap(pow3, xrange(%d), chunksize=%d)):\n\t\t%s' \
      ' seconds' % (N, N//8, time.time() - t)

assert A == B == C, (len(A), len(B), len(C))
print

L = [None] * 1000000
print 'def noop(x): pass'
print 'L = [None] * 1000000'

t = time.time()
A = map(noop, L)
print '\tmap(noop, L):\n\t\t%s seconds' % \
      (time.time() - t)

t = time.time()
B = pool.map(noop, L)
print '\tpool.map(noop, L):\n\t\t%s seconds' % \
      (time.time() - t)

t = time.time()
C = list(pool.imap(noop, L, chunksize=len(L)//8))
print '\tlist(pool.imap(noop, L, chunksize=%d)):\n\t\t%s seconds' % \
      (len(L)//8, time.time() - t)

assert A == B == C, (len(A), len(B), len(C))
print

del A, B, C, L

```

```

#
# Test error handling
#

print 'Testing error handling:'

try:
    print pool.apply(f, (5,))
except ZeroDivisionError:
    print '\tGot ZeroDivisionError as expected from pool.apply()'
else:
    raise AssertionError, 'expected ZeroDivisionError'

try:
    print pool.map(f, range(10))
except ZeroDivisionError:
    print '\tGot ZeroDivisionError as expected from pool.map()'
else:
    raise AssertionError, 'expected ZeroDivisionError'

try:
    print list(pool.imap(f, range(10)))
except ZeroDivisionError:
    print '\tGot ZeroDivisionError as expected from list(pool.imap())'
else:
    raise AssertionError, 'expected ZeroDivisionError'

it = pool.imap(f, range(10))
for i in range(10):
    try:
        x = it.next()
    except ZeroDivisionError:
        if i == 5:
            pass
    except StopIteration:
        break
    else:
        if i == 5:
            raise AssertionError, 'expected ZeroDivisionError'

assert i == 9
print '\tGot ZeroDivisionError as expected from IMapIterator.next()'
print

#
# Testing timeouts
#

print 'Testing ApplyResult.get() with timeout:',
res = pool.apply_async(calculate, TASKS[0])
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % res.get(0.02))
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print

```

```

print

print 'Testing IMapIterator.next() with timeout:',
it = pool.imap(calculatestar, TASKS)
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % it.next(0.02))
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')

print
print

#
# Testing callback
#

print 'Testing callback:'

A = []
B = [56, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

r = pool.apply_async(mul, (7, 8), callback=A.append)
r.wait()

r = pool.map_async(pow3, range(10), callback=A.extend)
r.wait()

if A == B:
    print '\tcallbacks succeeded\n'
else:
    print '\t*** callbacks failed\n\t\t%s != %s\n' % (A, B)

#
# Check there are no outstanding tasks
#

assert not pool._cache, 'cache = %r' % pool._cache

#
# Check close() methods
#

print 'Testing close():'

for worker in pool._pool:
    assert worker.is_alive()

result = pool.apply_async(time.sleep, [0.5])
pool.close()
pool.join()

assert result.get() is None

for worker in pool._pool:
    assert not worker.is_alive()

```

```

print '\tclose() succeeded\n'

#
# Check terminate() method
#

print 'Testing terminate():'

pool = multiprocessing.Pool(2)
DELTA = 0.1
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
pool.terminate()
pool.join()

for worker in pool._pool:
    assert not worker.is_alive()

print '\tterminate() succeeded\n'

#
# Check garbage collection
#

print 'Testing garbage collection:'

pool = multiprocessing.Pool(2)
DELTA = 0.1
processes = pool._pool
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]

results = pool = None

time.sleep(DELTA * 2)

for worker in processes:
    assert not worker.is_alive()

print '\tgarbage collection succeeded\n'

if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print 'Using processes'.center(79, '-')
    elif sys.argv[1] == 'threads':
        print 'Using threads'.center(79, '-')
    import multiprocessing.dummy as multiprocessing
    else:
        print 'Usage:\n\t%s [processes | threads]' % sys.argv[0]
        raise SystemExit(2)

test()

```

Synchronization types like locks, conditions and queues:

```
#
# A test file for the `multiprocessing` package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, random
from Queue import Empty

import multiprocessing          # may get overwritten

#### TEST_VALUE

def value_func(running, mutex):
    random.seed()
    time.sleep(random.random()*4)

    mutex.acquire()
    print '\n\t\t\t' + str(multiprocessing.current_process()) + ' has finished'
    running.value -= 1
    mutex.release()

def test_value():
    TASKS = 10
    running = multiprocessing.Value('i', TASKS)
    mutex = multiprocessing.Lock()

    for i in range(TASKS):
        p = multiprocessing.Process(target=value_func, args=(running, mutex))
        p.start()

    while running.value > 0:
        time.sleep(0.08)
        mutex.acquire()
        print running.value,
        sys.stdout.flush()
        mutex.release()

    print
    print 'No more running processes'

#### TEST_QUEUE

def queue_func(queue):
    for i in range(30):
        time.sleep(0.5 * random.random())
        queue.put(i*i)
    queue.put('STOP')

def test_queue():
    q = multiprocessing.Queue()

    p = multiprocessing.Process(target=queue_func, args=(q,))
    p.start()
```

```

o = None
while o != 'STOP':
    try:
        o = q.get(timeout=0.3)
        print o,
        sys.stdout.flush()
    except Empty:
        print 'TIMEOUT'

print

#### TEST_CONDITION

def condition_func(cond):
    cond.acquire()
    print '\t' + str(cond)
    time.sleep(2)
    print '\tchild is notifying'
    print '\t' + str(cond)
    cond.notify()
    cond.release()

def test_condition():
    cond = multiprocessing.Condition()

    p = multiprocessing.Process(target=condition_func, args=(cond,))
    print cond

    cond.acquire()
    print cond
    cond.acquire()
    print cond

    p.start()

    print 'main is waiting'
    cond.wait()
    print 'main has woken up'

    print cond
    cond.release()
    print cond
    cond.release()

    p.join()
    print cond

#### TEST_SEMAPHORE

def semaphore_func(sema, mutex, running):
    sema.acquire()

    mutex.acquire()
    running.value += 1
    print running.value, 'tasks are running'
    mutex.release()

```

```
random.seed()
time.sleep(random.random()*2)

mutex.acquire()
running.value -= 1
print '%s has finished' % multiprocessing.current_process()
mutex.release()

sema.release()

def test_semaphore():
    sema = multiprocessing.Semaphore(3)
    mutex = multiprocessing.RLock()
    running = multiprocessing.Value('i', 0)

    processes = [
        multiprocessing.Process(target=semaphore_func,
                               args=(sema, mutex, running))
        for i in range(10)
    ]

    for p in processes:
        p.start()

    for p in processes:
        p.join()

#### TEST_JOIN_TIMEOUT

def join_timeout_func():
    print '\tchild sleeping'
    time.sleep(5.5)
    print '\n\tchild terminating'

def test_join_timeout():
    p = multiprocessing.Process(target=join_timeout_func)
    p.start()

    print 'waiting for process to finish'

    while 1:
        p.join(timeout=1)
        if not p.is_alive():
            break
        print '.',
        sys.stdout.flush()

#### TEST_EVENT

def event_func(event):
    print '\t%sr is waiting' % multiprocessing.current_process()
    event.wait()
    print '\t%sr has woken up' % multiprocessing.current_process()

def test_event():
    event = multiprocessing.Event()
```

```

processes = [multiprocessing.Process(target=event_func, args=(event,))
              for i in range(5)]

for p in processes:
    p.start()

print 'main is sleeping'
time.sleep(2)

print 'main is setting event'
event.set()

for p in processes:
    p.join()

#### TEST_SHAREDVALUES

def sharedvalues_func(values, arrays, shared_values, shared_arrays):
    for i in range(len(values)):
        v = values[i][1]
        sv = shared_values[i].value
        assert v == sv

    for i in range(len(values)):
        a = arrays[i][1]
        sa = list(shared_arrays[i][:])
        assert a == sa

    print 'Tests passed'

def test_sharedvalues():
    values = [
        ('i', 10),
        ('h', -2),
        ('d', 1.25)
    ]
    arrays = [
        ('i', range(100)),
        ('d', [0.25 * i for i in range(100)]),
        ('H', range(1000))
    ]

    shared_values = [multiprocessing.Value(id, v) for id, v in values]
    shared_arrays = [multiprocessing.Array(id, a) for id, a in arrays]

    p = multiprocessing.Process(
        target=sharedvalues_func,
        args=(values, arrays, shared_values, shared_arrays)
    )
    p.start()
    p.join()

    assert p.exitcode == 0

####

```

```

def test(namespace=multiprocessing):
    global multiprocessing

    multiprocessing = namespace

    for func in [ test_value, test_queue, test_condition,
                 test_semaphore, test_join_timeout, test_event,
                 test_sharedvalues ]:

        print '\n\t##### %s\n' % func.__name__
        func()

    ignore = multiprocessing.active_children()          # cleanup any old processes
    if hasattr(multiprocessing, '_debug_info'):
        info = multiprocessing._debug_info()
        if info:
            print info
            raise ValueError, 'there should be no positive refcounts left'

if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
        namespace = multiprocessing
    elif sys.argv[1] == 'manager':
        print ' Using processes and a manager '.center(79, '-')
        namespace = multiprocessing.Manager()
        namespace.Process = multiprocessing.Process
        namespace.current_process = multiprocessing.current_process
        namespace.active_children = multiprocessing.active_children
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
        import multiprocessing.dummy as namespace
    else:
        print 'Usage:\n\t%s [processes | manager | threads]' % sys.argv[0]
        raise SystemExit, 2

test(namespace)

```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```

#
# Simple example which uses a pool of workers to carry out some tasks.
#
# Notice that the results will probably not come out of the output
# queue in the same in the same order as the corresponding tasks were
# put on the input queue.  If it is important to get the results back
# in the original order then consider using `Pool.map()` or
# `Pool.imap()` (which will save on the amount of code needed anyway).
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

```

```

import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print 'Unordered results:'

```

```
for i in range(len(TASKS1)):
    print '\t', done_queue.get()

# Add more tasks using `put()`
for task in TASKS2:
    task_queue.put(task)

# Get and print some more results
for i in range(len(TASKS2)):
    print '\t', done_queue.get()

# Tell child processes to stop
for i in range(NUMBER_OF_PROCESSES):
    task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
    test()
```

An example of how a pool of worker processes can each run a SimpleHTTPRequestHandler instance while sharing a single listening socket.

```
#
# Example where a pool of http servers share a single listening socket
#
# On Windows this module depends on the ability to pickle a socket
# object so that the worker processes can inherit a copy of the server
# object. (We import `multiprocessing.reduction` to enable this pickling.)
#
# Not sure if we should synchronize access to `socket.accept()` method by
# using a process-shared lock -- does not seem to be necessary.
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import os
import sys

from multiprocessing import Process, current_process, freeze_support
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

if sys.platform == 'win32':
    import multiprocessing.reduction    # make sockets pickable/inheritable

def note(format, *args):
    sys.stderr.write('[%s]\t%s\n' % (current_process().name, format%args))

class RequestHandler(SimpleHTTPRequestHandler):
    # we override log_message() to show which process is handling the request
    def log_message(self, format, *args):
        note(format, *args)

def serve_forever(server):
```

```

note('starting server')
try:
    server.serve_forever()
except KeyboardInterrupt:
    pass

def runpool(address, number_of_processes):
    # create a single server object -- children will each inherit a copy
    server = HTTPServer(address, RequestHandler)

    # create child processes to act as workers
    for i in range(number_of_processes-1):
        Process(target=serve_forever, args=(server,)).start()

    # main process also acts as a worker
    serve_forever(server)

def test():
    DIR = os.path.join(os.path.dirname(__file__), '..')
    ADDRESS = ('localhost', 8000)
    NUMBER_OF_PROCESSES = 4

    print 'Serving at http://%s:%d using %d worker processes' % \
        (ADDRESS[0], ADDRESS[1], NUMBER_OF_PROCESSES)
    print 'To exit press Ctrl-' + ['C', 'Break'][sys.platform=='win32']

    os.chdir(DIR)
    runpool(ADDRESS, NUMBER_OF_PROCESSES)

if __name__ == '__main__':
    freeze_support()
    test()

```

Some simple benchmarks comparing *multiprocessing* with threading:

```

#
# Simple benchmarks for the multiprocessing package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, multiprocessing, threading, Queue, gc

if sys.platform == 'win32':
    _timer = time.clock
else:
    _timer = time.time

delta = 1

#### TEST_QUEUESPEED

def queuespeed_func(q, c, iterations):
    a = '0' * 256

```

```
c.acquire()
c.notify()
c.release()

for i in xrange(iterations):
    q.put(a)

q.put('STOP')
```

```
def test_queuespeed(Process, q, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = Process(target=queuespeed_func, args=(q, c, iterations))
        c.acquire()
        p.start()
        c.wait()
        c.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = q.get()

        elapsed = _timer() - t

        p.join()

    print iterations, 'objects passed through the queue in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed
```

```
#### TEST_PIPESPEED
```

```
def pipe_func(c, cond, iterations):
    a = '0' * 256
    cond.acquire()
    cond.notify()
    cond.release()

    for i in xrange(iterations):
        c.send(a)

    c.send('STOP')
```

```
def test_pipespeed():
    c, d = multiprocessing.Pipe()
    cond = multiprocessing.Condition()
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2
```

```

    p = multiprocessing.Process(target=pipe_func,
                               args=(d, cond, iterations))

    cond.acquire()
    p.start()
    cond.wait()
    cond.release()

    result = None
    t = _timer()

    while result != 'STOP':
        result = c.recv()

    elapsed = _timer() - t
    p.join()

    print iterations, 'objects passed through connection in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed

#### TEST_SEQSPEED

def test_seqspeek(seq):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

    t = _timer()

    for i in xrange(iterations):
        a = seq[5]

    elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed

#### TEST_LOCK

def test_lockspeed(l):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

    t = _timer()

    for i in xrange(iterations):
        l.acquire()
        l.release()

    elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'

```

```

    print 'average number/sec:', iterations/elapsed

#### TEST_CONDITION

def conditionspeed_func(c, N):
    c.acquire()
    c.notify()

    for i in xrange(N):
        c.wait()
        c.notify()

    c.release()

def test_conditionspeed(Process, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        c.acquire()
        p = Process(target=conditionspeed_func, args=(c, iterations))
        p.start()

        c.wait()

        t = _timer()

        for i in xrange(iterations):
            c.notify()
            c.wait()

        elapsed = _timer()-t

        c.release()
        p.join()

    print iterations * 2, 'waits in', elapsed, 'seconds'
    print 'average number/sec:', iterations * 2 / elapsed

####

def test():
    manager = multiprocessing.Manager()

    gc.disable()

    print '\n\t##### testing Queue.Queue\n'
    test_queuespeed(threading.Thread, Queue.Queue(),
                    threading.Condition())
    print '\n\t##### testing multiprocessing.Queue\n'
    test_queuespeed(multiprocessing.Process, multiprocessing.Queue(),
                    multiprocessing.Condition())
    print '\n\t##### testing Queue managed by server process\n'
    test_queuespeed(multiprocessing.Process, manager.Queue(),
                    manager.Condition())

```

```

print '\n\t##### testing multiprocessing.Pipe\n'
test_pipespeed()

print

print '\n\t##### testing list\n'
test_seqspeek(range(10))
print '\n\t##### testing list managed by server process\n'
test_seqspeek(manager.list(range(10)))
print '\n\t##### testing Array("i", ..., lock=False)\n'
test_seqspeek(multiprocessing.Array('i', range(10), lock=False))
print '\n\t##### testing Array("i", ..., lock=True)\n'
test_seqspeek(multiprocessing.Array('i', range(10), lock=True))

print

print '\n\t##### testing threading.Lock\n'
test_lockspeed(threading.Lock())
print '\n\t##### testing threading.RLock\n'
test_lockspeed(threading.RLock())
print '\n\t##### testing multiprocessing.Lock\n'
test_lockspeed(multiprocessing.Lock())
print '\n\t##### testing multiprocessing.RLock\n'
test_lockspeed(multiprocessing.RLock())
print '\n\t##### testing lock managed by server process\n'
test_lockspeed(manager.Lock())
print '\n\t##### testing rlock managed by server process\n'
test_lockspeed(manager.RLock())

print

print '\n\t##### testing threading.Condition\n'
test_conditionspeed(threading.Thread, threading.Condition())
print '\n\t##### testing multiprocessing.Condition\n'
test_conditionspeed(multiprocessing.Process, multiprocessing.Condition())
print '\n\t##### testing condition managed by a server process\n'
test_conditionspeed(multiprocessing.Process, manager.Condition())

gc.enable()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()

```

Glossary

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode.

CPython The canonical implementation of the Python programming language. The term “CPython” is used in contexts when necessary to distinguish this implementation from others such as Jython or IronPython.

GIL See *global interpreter lock*.

global interpreter lock The lock used by Python threads to assure that only one thread executes in the *CPython virtual machine* at a time. This simplifies the CPython implementation by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines. Efforts have been made in the past to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity), but so far none have been successful because performance suffered in the common single-processor case.

virtual machine A computer defined entirely in software. Python’s virtual machine executes the *bytecode* emitted by the bytecode compiler.

Indices and tables

- `genindex`
- `modindex`
- `search`

m

multiprocessing, 3
multiprocessing.connection, 24
multiprocessing.dummy, 29
multiprocessing.managers, 17
multiprocessing.pool, 22
multiprocessing.sharedctypes, 15

Symbols

`__repr__()` (multiprocessing.managers.BaseProxy method), 22
`__str__()` (multiprocessing.managers.BaseProxy method), 22
`_callmethod()` (multiprocessing.managers.BaseProxy method), 21
`_getvalue()` (multiprocessing.managers.BaseProxy method), 22

A

`accept()` (multiprocessing.connection.Listener method), 25
`active_children()` (in module multiprocessing), 11
`address` (multiprocessing.connection.Listener attribute), 25
`address` (multiprocessing.managers.BaseManager attribute), 18
`answerChallenge()` (in module multiprocessing.connection), 25
`apply()` (multiprocessing.pool.multiprocessing.Pool method), 22
`apply_async()` (multiprocessing.pool.multiprocessing.Pool method), 22
`Array()` (in module multiprocessing), 15
`Array()` (in module multiprocessing.sharedctypes), 15
`Array()` (multiprocessing.managers.SyncManager method), 19
`AsyncResult` (class in multiprocessing.pool), 24
`AuthenticationError`, 26
`authkey` (multiprocessing.Process attribute), 8

B

`BaseManager` (class in multiprocessing.managers), 17
`BaseProxy` (class in multiprocessing.managers), 21
`BoundedSemaphore` (class in multiprocessing), 14
`BoundedSemaphore()` (multiprocessing.managers.SyncManager method), 18
`BufferTooShort`, 9

bytecode, 53

C

`cancel_join_thread()` (multiprocessing.Queue method), 11
`Client()` (in module multiprocessing.connection), 25
`close()` (multiprocessing.Connection method), 13
`close()` (multiprocessing.connection.Listener method), 25
`close()` (multiprocessing.pool.multiprocessing.Pool method), 23
`close()` (multiprocessing.Queue method), 11
`Condition` (class in multiprocessing), 14
`Condition()` (multiprocessing.managers.SyncManager method), 18
`connect()` (multiprocessing.managers.BaseManager method), 17
`Connection` (class in multiprocessing), 12
`copy()` (in module multiprocessing.sharedctypes), 16
`cpu_count()` (in module multiprocessing), 11
`CPython`, 53
`current_process()` (in module multiprocessing), 12

D

`daemon` (multiprocessing.Process attribute), 8
`deliver_challenge()` (in module multiprocessing.connection), 24
`dict()` (multiprocessing.managers.SyncManager method), 19

E

`empty()` (multiprocessing.Queue method), 10
`empty()` (multiprocessing.SimpleQueue method), 11
`Event` (class in multiprocessing), 14
`Event()` (multiprocessing.managers.SyncManager method), 18
`exitcode` (multiprocessing.Process attribute), 8

F

`fileno()` (multiprocessing.Connection method), 13
`freeze_support()` (in module multiprocessing), 12

full() (multiprocessing.Queue method), 10

G

get() (multiprocessing.pool.AsyncResult method), 24

get() (multiprocessing.Queue method), 10

get() (multiprocessing.SimpleQueue method), 11

get_logger() (in module multiprocessing), 28

get_no_wait() (multiprocessing.Queue method), 10

get_nowait() (multiprocessing.Queue method), 10

get_server() (multiprocessing.managers.BaseManager method), 17

GIL, 53

global interpreter lock, 53

I

imap() (multiprocessing.pool.multiprocessing.Pool method), 23

imap_unordered() (multiprocessing.pool.multiprocessing.Pool method), 23

is_alive() (multiprocessing.Process method), 8

J

join() (multiprocessing.JoinableQueue method), 11

join() (multiprocessing.pool.multiprocessing.Pool method), 24

join() (multiprocessing.Process method), 8

join_thread() (multiprocessing.Queue method), 11

JoinableQueue (class in multiprocessing), 11

L

last_accepted (multiprocessing.connection.Listener attribute), 25

list() (multiprocessing.managers.SyncManager method), 19

Listener (class in multiprocessing.connection), 25

Lock (class in multiprocessing), 14

Lock() (multiprocessing.managers.SyncManager method), 18

log_to_stderr() (in module multiprocessing), 28

M

map() (multiprocessing.pool.multiprocessing.Pool method), 23

map_async() (multiprocessing.pool.multiprocessing.Pool method), 23

multiprocessing (module), 3

multiprocessing.connection (module), 24

multiprocessing.dummy (module), 29

multiprocessing.Manager() (in module multiprocessing.sharedctypes), 17

multiprocessing.managers (module), 17

multiprocessing.Pool (class in multiprocessing.pool), 22

multiprocessing.pool (module), 22

multiprocessing.sharedctypes (module), 15

N

name (multiprocessing.Process attribute), 8

Namespace() (multiprocessing.managers.SyncManager method), 19

P

pid (multiprocessing.Process attribute), 8

Pipe() (in module multiprocessing), 10

poll() (multiprocessing.Connection method), 13

Process (class in multiprocessing), 7

put() (multiprocessing.Queue method), 10

put() (multiprocessing.SimpleQueue method), 11

put_nowait() (multiprocessing.Queue method), 10

Q

qsize() (multiprocessing.Queue method), 10

Queue (class in multiprocessing), 10

Queue() (multiprocessing.managers.SyncManager method), 19

R

RawArray() (in module multiprocessing.sharedctypes), 15

RawValue() (in module multiprocessing.sharedctypes), 15

ready() (multiprocessing.pool.AsyncResult method), 24

recv() (multiprocessing.Connection method), 12

recv_bytes() (multiprocessing.Connection method), 13

recv_bytes_into() (multiprocessing.Connection method), 13

register() (multiprocessing.managers.BaseManager method), 18

RLock (class in multiprocessing), 14

RLock() (multiprocessing.managers.SyncManager method), 19

run() (multiprocessing.Process method), 7

S

Semaphore (class in multiprocessing), 14

Semaphore() (multiprocessing.managers.SyncManager method), 19

send() (multiprocessing.Connection method), 12

send_bytes() (multiprocessing.Connection method), 13

sentinel (multiprocessing.Process attribute), 8

set_executable() (in module multiprocessing), 12

shutdown() (multiprocessing.managers.BaseManager method), 18

SimpleQueue (class in multiprocessing), 11

starmap() (multiprocessing.pool.multiprocessing.Pool method), 23

`starmap_async()` (multiprocessing.pool.multiprocessing.Pool method), 23

`start()` (multiprocessing.managers.BaseManager method), 17

`start()` (multiprocessing.Process method), 7

`successful()` (multiprocessing.pool.AsyncResult method), 24

`synchronized()` (in module multiprocessing.sharedctypes), 16

`SyncManager` (class in multiprocessing.managers), 18

T

`task_done()` (multiprocessing.JoinableQueue method), 11

`terminate()` (multiprocessing.pool.multiprocessing.Pool method), 23

`terminate()` (multiprocessing.Process method), 9

V

`Value()` (in module multiprocessing), 14

`Value()` (in module multiprocessing.sharedctypes), 16

`Value()` (multiprocessing.managers.SyncManager method), 19

virtual machine, 53

W

`wait()` (in module multiprocessing.connection), 26

`wait()` (multiprocessing.pool.AsyncResult method), 24