guv

Release 0.35.2

July 13, 2015

Contents

Note: The documentation is currently in very active developemnt and not yet complete. Please keep checking back for updates and filing issues for missing sections or suggestions for enhancement.

CHAPTER 1

Contents

1.1 How guv works

1.1.1 The "old" way of writing servers

The classic server design involves blocking sockets, `select.select()`, and spawning operating system threads for each new client connection. The only advantage of this method is the simplicity of its design. Although sufficient for serving a very small number of clients, system resources quickly get maxed out when spawning a large number of threads frequently, and `select.select()` doesn't scale well to a large number of open file descriptors.

An improvement on this design is using a platform-specific `poll()` mechanism such as `epoll()`, which handles polling a large number of file descriptors much more efficiently.

However, the thread issue remains. Typical solutions involve implementing the "reactor pattern" in an event loop using something like `epoll()`. The issue with this approach is that all code runs in a single thread and one must be careful not to block the thread in any way. Setting the socket file descriptors to non-blocking mode helps in this aspect, but effectively using this design pattern is difficult, and requires the cooperation of all parts of the system.

1.1.2 Coroutines, event loops, and monkey-patching

guv is an elegant solution to all of the problems mentioned above. It allows you to write highly efficient code that *looks* like it's running in its own thread, and *looks* like it's blocking. It does this by making use of greenlets instead of operating system threads, and globally monkey-patching system modules to cooperatively yield while waiting for I/O or other events. greenlets are extremely light-weight, and all run in a single operating system thread; switching between greenlets incurs very low overhead. Furthermore, only the greenlets that need switching to will be switched to when I/O or another event is ready; guv does not unnecessarily waste resources switching to greenlets that don't need attention.

For example, the `socket` module is one of the core modules which is monkey-patched by guv. When using the patched socket module, calls to `socket.read()` on a "blocking" socket will register interest in the file descriptor, then cooperatively yield to another greenlet instead of blocking the entire thread.

In addition, all monkey-patched modules are 100% API-compatible with the original system modules, so this allows existing networking code to run without modification as long as standard python modules are used. Code using C extensions will require simple modifications to cooperate with guv, since it is not possible to monkey-patch C code which may be making blocking function calls.

1.1.3 The hub and `trampoline()`

The "hub" (`guv.hubs.abc.AbstractHub`) is the core of guv and serves as the "scheduler" for greenlets. All calls to `spawn()` (and related functions) actually enqueue a request with the hub to spawn the greenlet on the next event loop iteration. The hub itself is a subclass of `greenlet.greenlet`

The hub also manages the underlying event loop (currently libuv only, but implementations for any event loop library, or even custom event loops can easily be written). Calls to monkey-patched functions actually register either a timer or the underlying file descriptor with libuv and switch ("yield") to the hub greenlet.

The core function which facilitates the process of registering the file descriptor of interest and switching to the hub is *`trampoline()`*. Examining the source code of included green modules reveals that this function is used extensively whenever interest in I/O events for a file descriptor needs to be registered. Note that this function does not need to be called by normal application code when writing code with the guv library; this is only part of the core inner working of guv.

Another important function provided by guv for working with greenlets is *`gyield()`*. This is a very simple function which simply yields the current greenlet, and registers a callback to resume on the next event loop iteration.

If you require providing support for a library which cannot make use of the patched python standard socket module (such as the case for C extensions), then it is necessary to provide a support module which calls either *`trampoline()`* or *`gyield()`* when there is a possibility that the C code will block for I/O.

For examples of support modules for common libraries, see the support modules provided in the `guv.support` package.

1.2 Library Support

The goal of guv is to support as many external libraries as possible such that no modification to application code is necessary. However, note that it is still required to use certain guv-specific constructs to take advantage of the concurrency (as demonstrated in the examples directory).

Quick overview:

- If your application code and any library dependencies are pure-python and use only standard library components like `socket`, `time`, `os`, etc., then your code is guaranteed to be compatible with guv.
- If your application code depends on libraries that make blocking I/O calls *from external C code* (such as is the case for many popular database drivers), then a support module must be available to make those specific libraries cooperative. Such modules can be found in the guv.support package and are all enabled by default if the library is installed.

Note: If your code is using only standard library components and is behaving in a non-cooperative way, this is considered a critical bug, which can be fixed by greenifying the appropriate standard library modules. Please submit a bug report to ensure that this issue is fixed as soon as possible.

1.2.1 List of Known Compatible Libraries

Pure-python libraries are guaranteed to be compatible with no additional support modules:

- All standard library modules which make blocking calls such as I/O calls on file descriptors (including `socket`, `smtplib`, etc) are automatically supported.
- boto
- Cassandra driver

- gunicorn (use with `-k guv.GuvWorker`)
- pg8000
- redis-py
- requests
- Many more. This list will be expanded as additional libraries are tested and *confirmed* to be compatible

Libraries containing C extensions which are currently supported:

- psycopg2

1.2.2 Writing support modules for external libraries

The idea behind guv is that everything runs in one OS thread (even monkey-patched `threading.Thread` objects!). Within this single thread, greenlets are used to switch between various functions efficiently. This means that any code making blocking calls will block the entire thread and prevent any other greenlet from running. For this reason, guv provides a monkey-patched standard library where all functions that can potentially block are replaced with their "greenified" counterparts that *yield* instead of blocking. The goal is to ensure that 100% of the standard library is greenified. If you encounter any part of the standard library that seems to be blocking instead of yielding, please file a bug report so this can be resolved as soon as possible.

The issue arises when using modules which make calls to compiled code that cannot be monkey-patched (for example, through C extensions or CFFI). This is the case for many popular database drivers or other network code which aim for maximum performance.

Some libraries provide mechanisms for the purpose of facilitating creating support modules for libraries such as guv. An excellent example is the high quality `psycopg2` database driver for PostgreSQL, written as a C extension. This library provides a very clean mechanism to call a callback before making any operations which could potentially block. This allows guv to *`trampoline()`* and register the connection's file descriptor if the I/O operation would block.

See the psycopg2 patcher for the implementation.

However, many libraries do not provide such a mechanism to simplify creating a support module. In such case, there are several strategies for making these libraries cooperative. In all cases, the end goal is the same: call *`trampoline()`*, which cooperatively yields and waits for the file descriptor to be ready for I/O.

Note: this section is incomplete.

1.3 Module Reference

1.3.1 `guv.const` - constants

Event Types

`guv.const.`**`READ`** **= 1**
: This is equivalent to `UV_READABLE`

`guv.const.`**`WRITE`** **= 2**
: This is equivalent to `UV_WRITABLE`

1.3.2 guv.event - event primitive for greenthreads

class guv.event.**Event**
Bases: object

An abstraction where an arbitrary number of greenlets can wait for one event from another

Events are similar to a Queue that can only hold one item, but differ in two important ways:

1. Calling *send()* never unschedules the current GreenThread
2. *send()* can only be called once; create a new event to send again.

They are good for communicating results between greenlets, and are the basis for how GreenThread.wait() is implemented.

```
>>> from guv import event
>>> import guv
>>> evt = event.Event()
>>> def baz(b):
...     evt.send(b + 1)
...
>>> _ = guv.spawn_n(baz, 3)
>>> evt.wait()
4
```

ready() → None
Return true if the *wait()* call will return immediately

Used to avoid waiting for things that might take a while to time out. For example, you can put a bunch of events into a list, and then visit them all repeatedly, calling *ready()* until one returns True, and then you can *wait()* on that one

send(*result=None*, *exc=None*) → None
Make arrangements for the waiters to be woken with the result and then return immediately to the parent

```
>>> from guv import event
>>> import guv
>>> evt = event.Event()
>>> def waiter():
...     print('about to wait')
...     result = evt.wait()
...     print('waited for {0}'.format(result))
>>> _ = guv.spawn(waiter)
>>> guv.sleep(0)
about to wait
>>> evt.send('a')
>>> guv.sleep(0)
waited for a
```

It is an error to call *send()* multiple times on the same event.

```
>>> evt.send('whoops')
Traceback (most recent call last):
...
AssertionError: Trying to re-send() an already-triggered event.
```

Use reset() between *send()* s to reuse an event object.

send_exception(**args*) → None
Same as *send()*, but sends an exception to waiters.

The arguments to send_exception are the same as the arguments to `raise`. If a single exception object is passed in, it will be re-raised when *`wait()`* is called, generating a new stacktrace.

```
>>> from guv import event
>>> evt = event.Event()
>>> evt.send_exception(RuntimeError())
>>> evt.wait()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "guv/event.py", line 120, in wait
    current.throw(*self._exc)
RuntimeError
```

If it's important to preserve the entire original stack trace, you must pass in the entire `sys.exc_info()` tuple.

```
>>> import sys
>>> evt = event.Event()
>>> try:
...     raise RuntimeError()
... except RuntimeError:
...     evt.send_exception(*sys.exc_info())
...
>>> evt.wait()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "guv/event.py", line 120, in wait
    current.throw(*self._exc)
  File "<stdin>", line 2, in <module>
RuntimeError
```

Note that doing so stores a traceback object directly on the Event object, which may cause reference cycles. See the `sys.exc_info()` documentation.

wait() → None

Wait until another greenthread calls *`send()`*

Returns the value the other coroutine passed to *`send()`*.

Returns immediately if the event has already occurred.

```
>>> from guv import event
>>> import guv
>>> evt = event.Event()
>>> def wait_on():
...    retval = evt.wait()
...    print("waited for {0}".format(retval))
>>> _ = guv.spawn(wait_on)
>>> evt.send('result')
>>> guv.sleep(0)
waited for result
```

```
>>> evt.wait()
'result'
```

class guv.event.**TEvent**

Bases: `object`

A synchronization primitive that allows one greenlet to wake up one or more others. It has the same interface as `threading.Event` but works across greenlets.

An event object manages an internal flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true.

clear() → None
Reset the internal flag to false. Subsequently, threads calling *wait()* will block until *set()* is called to set the internal flag to true again.

isSet() → None
Return true if and only if the internal flag is true.

is_set() → None
Return true if and only if the internal flag is true.

rawlink(*callback*) → None
Register a callback to call when the internal flag is set to true

callback will be called in the `Hub`, so it must not use blocking *gevent API*. *callback* will be passed one argument: this instance.

ready() → None
Return true if and only if the internal flag is true.

set() → None
Set the internal flag to true. All greenlets waiting for it to become true are awakened. Greenlets that call *wait()* once the flag is true will not block at all.

unlink(*callback*) → None
Remove the callback set by *rawlink()*

wait(*timeout=None*) → None
Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls *set()* to set the flag to true, or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

Return the value of the internal flag (`True` or `False`).

class `guv.event.`**AsyncResult**
Bases: `object`

A one-time event that stores a value or an exception

Like *Event* it wakes up all the waiters when *set()* or *set_exception()* method is called. Waiters may receive the passed value or exception by calling *get()* method instead of *wait()*. An *AsyncResult* instance cannot be reset.

To pass a value call *set()*. Calls to *get()* (those that currently blocking as well as those made in the future) will return the value:

```
>>> result = AsyncResult()
>>> result.set(100)
>>> result.get()
100
```

To pass an exception call *set_exception()*. This will cause *get()* to raise that exception:

```
>>> result = AsyncResult()
>>> result.set_exception(RuntimeError('failure'))
>>> result.get()
Traceback (most recent call last):
 ...
RuntimeError: failure
```

AsyncResult implements `__call__()` and thus can be used as `link()` target:

```
>>> import gevent
>>> result = AsyncResult()
>>> gevent.spawn(lambda : 1/0).link(result)
>>> try:
...     result.get()
... except ZeroDivisionError:
...     print 'ZeroDivisionError'
ZeroDivisionError
```

exception
@property

Holds the exception instance passed to *set_exception()* if *set_exception()* was called. Otherwise `None`.

get(*block=True*, *timeout=None*) → None
Return the stored value or raise the exception.

If this instance already holds a value / an exception, return / raise it immediatelly. Otherwise, block until another greenlet calls *set()* or *set_exception()* or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

get_nowait() → None
Return the value or raise the exception without blocking.

If nothing is available, raise `gevent.Timeout` immediatelly.

rawlink(*callback*) → None
Register a callback to call when a value or an exception is set.

callback will be called in the `Hub`, so it must not use blocking gevent API. *callback* will be passed one argument: this instance.

ready() → None
Return true if and only if it holds a value or an exception

set(*value=None*) → None
Store the value. Wake up the waiters.

All greenlets blocking on *get()* or *wait()* are woken up. Sequential calls to *wait()* and *get()* will not block at all.

set_exception(*exception*) → None
Store the exception. Wake up the waiters.

All greenlets blocking on *get()* or *wait()* are woken up. Sequential calls to *wait()* and *get()* will not block at all.

successful() → None
Return true if and only if it is ready and holds a value

unlink(*callback*) → None
Remove the callback set by *rawlink()*

wait(*timeout=None*) → None
Block until the instance is ready.

If this instance already holds a value / an exception, return immediatelly. Otherwise, block until another thread calls *set()* or *set_exception()* or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

Return `value`.

1.3.3 `guv.greenpool` - greenthread pools

class `guv.greenpool.`**`GreenPool`**(*size=1000*)
Bases: `object`

Pool of greenlets/GreenThreads

This class manages a pool of greenlets/GreenThreads

`__init__`(*size=1000*)

Parameters **`size`** – maximum number of active greenlets

`free`() → None
Return the number of greenthreads available for use

If zero or less, the next call to *`spawn()`* or *`spawn_n()`* will block the calling greenthread until a slot becomes available.

`resize`(*new_size*) → None
Change the max number of greenthreads doing work at any given time

If resize is called when there are more than *new_size* greenthreads already working on tasks, they will be allowed to complete but no new tasks will be allowed to get launched until enough greenthreads finish their tasks to drop the overall quantity below *new_size*. Until then, the return value of free() will be negative.

`running`() → None
Return the number of greenthreads that are currently executing functions in the GreenPool

`spawn`(*function*, **args*, ***kwargs*) → None
Run the *function* with its arguments in its own green thread

Returns the *`GreenThread`* object that is running the function, which can be used to retrieve the results.

If the pool is currently at capacity, `spawn` will block until one of the running greenthreads completes its task and frees up a slot.

This function is reentrant; *function* can call `spawn` on the same pool without risk of deadlocking the whole thing.

`spawn_n`(*function*, **args*, ***kwargs*) → None
Create a greenthread to run the *function* like *`spawn()`*, but return None

The difference is that *`spawn_n()`* returns None; the results of *function* are not retrievable.

`starmap`(*function*, *iterable*) → None
Apply each item in *iterable* to *function*

Each item in *iterable* must be an iterable itself, passed to the function as expanded positional arguments. This behaves the same way as `itertools.starmap()`, except that *func* is executed in a separate green thread for each item, with the concurrency limited by the pool's size. In operation, starmap consumes a constant amount of memory, proportional to the size of the pool, and is thus suited for iterating over extremely long input lists.

`waitall`() → None
Wait until all greenthreads in the pool are finished working

waiting() → None
Return the number of greenthreads waiting to spawn.

class guv.greenpool.**GreenPile**(*size_or_pool=1000*)
Bases: object

An abstraction representing a set of I/O-related tasks

Construct a GreenPile with an existing GreenPool object. The GreenPile will then use that pool's concurrency as it processes its jobs. There can be many GreenPiles associated with a single GreenPool.

A GreenPile can also be constructed standalone, not associated with any GreenPool. To do this, construct it with an integer size parameter instead of a GreenPool.

It is not advisable to iterate over a GreenPile in a different greenlet than the one which is calling spawn. The iterator will exit early in that situation.

__init__(*size_or_pool=1000*)

Parameters **size_or_pool** (*int or GreenPool*) – either an existing GreenPool object, or the size a new one to create

next() → None
Wait for the next result, suspending the current GreenThread until it is available

Raises StopIteration when there are no more results.

spawn(*func*, **args*, ***kwargs*) → None
Run *func* in its own GreenThread

The Result is available by iterating over the GreenPile object.

Parameters

- **func** (*Callable*) – function to call
- **args** – positional args to pass to *func*
- **kwargs** – keyword args to pass to *func*

1.3.4 guv.greenthread - cooperative threads

class guv.greenthread.**GreenThread**(*parent*)
Bases: greenlet.greenlet

The GreenThread class is a type of Greenlet which has the additional property of being able to retrieve the return value of the main function. Do not construct GreenThread objects directly; call *spawn()* to get one.

__init__(*parent*)

Parameters **parent** (*greenlet.greenlet*) – parent greenlet

cancel(**throw_args*) → None
Kill the GreenThread using *kill()*, but only if it hasn't already started running

After being canceled, all calls to *wait()* will raise *throw_args* (which default to greenlet.GreenletExit).

kill(**throw_args*) → None
Kill the GreenThread using *kill()*

After being killed all calls to *wait()* will raise *throw_args* (which default to greenlet.GreenletExit).

link (*func*, **curried_args*, ***curried_kwargs*) → None
Set up a function to be called with the results of the GreenThread

The function must have the following signature:

```
func(gt, [curried args/kwargs])
```

When the GreenThread finishes its run, it calls *func* with itself and with the curried arguments supplied at link-time. If the function wants to retrieve the result of the GreenThread, it should call wait() on its first argument.

Note that *func* is called within execution context of the GreenThread, so it is possible to interfere with other linked functions by doing things like switching explicitly to another GreenThread.

unlink (*func*, **curried_args*, ***curried_kwargs*) → None
Remove linked function set by *link()*

Remove successfully return True, otherwise False

wait () → None
Return the result of the main function of this GreenThread

If the result is a normal return value, *wait()* returns it. If it raised an exception, *wait()* will raise the same exception (though the stack trace will unavoidably contain some frames from within the GreenThread module).

guv.greenthread.**sleep** (*seconds=0*)
Yield control to the hub until at least *seconds* have elapsed

Parameters **seconds** (*float*) – time to sleep for

guv.greenthread.**spawn** (*func*, **args*, ***kwargs*)
Spawn a GreenThread

Execution control returns immediately to the caller; the created GreenThread is scheduled to be run at the start of the next event loop iteration, after other scheduled greenlets, but before greenlets waiting for I/O events.

Returns GreenThread object which can be used to retrieve the return value of the function

Return type *GreenThread*

guv.greenthread.**spawn_n** (*func*, **args*, ***kwargs*)
Spawn a greenlet

Execution control returns immediately to the caller; the created greenlet is scheduled to be run at the start of the next event loop iteration, after other scheduled greenlets, but before greenlets waiting for I/O events.

This is faster than *spawn()*, but it is not possible to retrieve the return value of the greenlet, or whether it raised any exceptions. It is fastest if there are no keyword arguments.

If an exception is raised in the function, a stack trace is printed; the print can be disabled by calling `guv.debug.hub_exceptions()` with False.

Returns greenlet object

Return type greenlet.greenlet

guv.greenthread.**kill** (*g*, **throw_args*)
Terminate the target greenlet/GreenThread by raising an exception into it

Whatever that GreenThread might be doing, be it waiting for I/O or another primitive, it sees an exception right away.

By default, this exception is GreenletExit, but a specific exception may be specified. *throw_args* should be the same as the arguments to raise; either an exception instance or an exc_info tuple.

Calling *kill()* causes the calling greenlet to cooperatively yield.

Parameters **g** (*greenlet.greenlet or GreenThread*) – target greenlet/GreenThread to kill

guv.greenthread.**spawn_after**(*seconds*, *func*, **args*, ***kwargs*)
Spawn a GreenThread after *seconds* have elapsed

Execution control returns immediately to the caller.

To cancel the spawn and prevent *func* from being called, call *GreenThread.cancel()* on the returned GreenThread. This will not abort the function if it's already started running, which is generally the desired behavior. If terminating *func* regardless of whether it's started or not is the desired behavior, call *GreenThread.kill()*.

Returns GreenThread object which can be used to retrieve the return value of the function

Return type *GreenThread*

1.3.5 guv.patcher - monkey-patching the standard library

guv.patcher.**monkey_patch**(***modules*)
Globally patch/configure system modules to to be greenlet-friendly

If no keyword arguments are specified, all possible modules are patched. If keyword arguments are specified, the specified modules (and their dependencies) will be patched.

- Patching `socket` will also patch `ssl`
- Patching `threading` will also patch `_thread` and `queue`

It's safe to call monkey_patch multiple times.

Example:

```
monkey_patch(time=True, socket=True, select=True)
```

Parameters

- **time** (*bool*) – time module: patches sleep()
- **os** (*bool*) – os module: patches open(), read(), write(), wait(), waitpid()
- **socket** (*bool*) – socket module: patches socket, create_connection()
- **select** (*bool*) – select module: patches select()
- **threading** (*bool*) – threading module: patches local, Lock(), stack_size(), current_thread()
- **psycopg2** (*bool*) – psycopg2 module: register a wait callback to yield
- **cassandra** (*bool*) – cassandra module: set connection class to GuvConnection

guv.patcher.**original**(*modname*)
Return an unpatched version of a module

This is useful for guv itself.

Parameters **modname** (*str*) – name of module

guv.patcher.**is_monkey_patched**(*module*)
Check if the specified module is currently patched

Based entirely off the name of the module, so if you import a module some other way than with the import keyword (including import_patched), this might not be correct about that particular module

Parameters **module** (*module or str*) – module to check (moduble object itself, or its name str)

Returns True if the module is patched else False

Return type bool

guv.patcher.**inject**(*module_name*, *new_globals*, **additional_modules*)
Inject greenified modules into an imported module

This method imports the module specified in *module_name*, arranging things so that the already-imported modules in *additional_modules* are used when *module_name* makes its imports.

new_globals is either None or a globals dictionary that gets populated with the contents of the *module_name* module. This is useful when creating a "green" version of some other module.

additional_modules should be a collection of two-element tuples, of the form `(name: str, module: str)`. If it's not specified, a default selection of name/module pairs is used, which should cover all use cases but may be slower because there are inevitably redundant or unnecessary imports.

guv.patcher.**import_patched**(*module_name*, **additional_modules*, ***kw_additional_modules*)
Import patched version of module

Parameters **module_name** (*str*) – name of module to import

guv.patcher.**patch_function**(*func*, **additional_modules*)
Decorator that returns a version of the function that patches some modules for the duration of the function call

This should only be used for functions that import network libraries within their function bodies that there is no way of getting around.

1.3.6 guv.queue - greenthread-compatible queue

Synchronized queues

This module implements multi-producer, multi-consumer queues that work across greenlets, with the API similar to the classes found in the standard `queue` and `multiprocessing` modules.

A major difference is that queues in this module operate as channels when initialized with *maxsize* of zero. In such case, both `empty()` and `full()` return `True` and `put()` always blocks until a call to `get()` retrieves the item.

An interesting difference, made possible because of GreenThreads, is that `qsize()`, `empty()`, and `full()` *can* be used as indicators of whether the subsequent `get()` or `put()` will not block. The new methods *`LightQueue.getting()`* and *`LightQueue.putting()`* report on the number of GreenThreads blocking in `put()` or `get()` respectively.

exception guv.queue.**Full**
Bases: `Exception`

Exception raised by Queue.put(block=0)/put_nowait().

exception guv.queue.**Empty**
Bases: `Exception`

Exception raised by Queue.get(block=0)/get_nowait().

class guv.queue.**Queue**(*maxsize=None*)
Bases: *`guv.queue.LightQueue`*

Create a queue object with a given maximum size

If *maxsize* is less than zero or `None`, the queue size is infinite.

`Queue(0)` is a channel, that is, its `put()` method always blocks until the item is delivered. (This is unlike the standard `queue.Queue`, where 0 means infinite size).

In all other respects, this Queue class resembles the standard library, `queue.Queue`.

join() → None
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.

task_done() → None
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.

class guv.queue.**PriorityQueue**(*maxsize=None*)
Bases: *guv.queue.Queue*

A subclass of *Queue* that retrieves entries in priority order (lowest first)

Entries are typically tuples of the form: `(priority number, data)`.

class guv.queue.**LifoQueue**(*maxsize=None*)
Bases: *guv.queue.Queue*

A subclass of *Queue* that retrieves most recently added entries first

class guv.queue.**LightQueue**(*maxsize=None*)
Bases: `object`

This is a variant of Queue that behaves mostly like the standard *Queue*. It differs by not supporting the `task_done()` or `join()` methods, and is a little faster for not having that overhead.

empty() → None
Return `True` if the queue is empty, `False` otherwise.

full() → None
Return `True` if the queue is full, `False` otherwise.

`Queue(None)` is never full.

get(*block=True*, *timeout=None*) → None
Remove and return an item from the queue.

If optional args *block* is true 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() → None
Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the `queue.Empty` exception.

getting() → None
Returns the number of GreenThreads that are blocked waiting on an empty queue.

put(*item*, *block=True*, *timeout=None*) → None
Put an item into the queue.

If optional arg *block* is true 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(*item*) → None
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the `queue.Full` exception.

putting() → None
Returns the number of GreenThreads that are blocked waiting to put items into the queue.

qsize() → None
Return the size of the queue.

resize(*size*) → None
Resizes the queue's maximum size.

If the size is increased, and there are putters waiting, they may be woken up.

1.3.7 `guv.semaphore` - greenthread-compatible semaphore

class `guv.semaphore.`**BoundedSemaphore**(*value=1*)
Bases: *guv.semaphore.Semaphore*

A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, ValueError is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

release(*blocking=True*) → None
Release a semaphore, incrementing the internal counter by one. If the counter would exceed the initial value, raises ValueError. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

The *blocking* argument is for consistency with *CappedSemaphore* and is ignored

class `guv.semaphore.`**CappedSemaphore**(*count*, *limit*)
Bases: `object`

A blockingly bounded semaphore.

Optionally initialize with a resource *count*, then *acquire()* and *release()* resources as needed. Attempting to *acquire()* when *count* is zero suspends the calling greenthread until count becomes nonzero again. Attempting to *release()* after *count* has reached *limit* suspends the calling greenthread until *count* becomes less than *limit* again.

This has the same API as `threading.Semaphore`, though its semantics and behavior differ subtly due to the upper limit on calls to *release()*. It is **not** compatible with `threading.BoundedSemaphore` because it blocks when reaching *limit* instead of raising a ValueError.

It is a context manager, and thus can be used in a with block:

```
sem = CappedSemaphore(2)
with sem:
  do_some_stuff()
```

balance
@property

An integer value that represents how many new calls to *acquire()* or *release()* would be needed to get the counter to 0. If it is positive, then its value is the number of acquires that can happen before the next acquire would block. If it is negative, it is the negative of the number of releases that would be required in order to make the counter 0 again (one more release would push the counter to 1 and unblock acquirers). It takes into account how many greenthreads are currently blocking in *acquire()* and *release()*.

acquire(*blocking=True*) → None
Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release() to make it larger than zero. This is done with proper interlocking so that if multiple acquire() calls are blocked, release() will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with blocking set to true, do the same thing as when called without arguments, and return true.

When invoked with blocking set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

bounded() → None
Returns true if a call to release would block.

locked() → None
Returns true if a call to acquire would block.

release(*blocking=True*) → None
Release a semaphore. In this class, this behaves very much like an *acquire()* but in the opposite direction.

Imagine the docs of *acquire()* here, but with every direction reversed. When calling this method, it will block if the internal counter is greater than or equal to *limit*.

class guv.semaphore.**Semaphore**(*value=1*)
Bases: object

An unbounded semaphore

Optionally initialize with a resource *count*, then *acquire()* and *release()* resources as needed. Attempting to *acquire()* when count* is zero suspends the calling greenthread until *count* becomes nonzero again.

This is API-compatible with threading.Semaphore.

It is a context manager, and thus can be used in a with block:

```
sem = Semaphore(2)
with sem:
  do_some_stuff()
```

If not specified, *value* defaults to 1.

It is possible to limit acquire time:

```
sem = Semaphore()
ok = sem.acquire(timeout=0.1)
# True if acquired, False if timed out.
```

balance
@property

An integer value that represents how many new calls to *acquire()* or *release()* would be needed to get the counter to 0. If it is positive, then its value is the number of acquires that can happen before the next acquire would block. If it is negative, it is the negative of the number of releases that would be required in order to make the counter 0 again (one more release would push the counter to 1 and unblock acquirers). It takes into account how many greenthreads are currently blocking in *acquire()*.

acquire(*blocking=True*, *timeout=None*) → None
Acquire a semaphore

This function behaves like `threading.Lock.acquire()`.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release() to make it larger than zero. This is done with proper interlocking so that if multiple acquire() calls are blocked, release() will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with blocking set to true, do the same thing as when called without arguments, and return true.

When invoked with blocking set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

bounded() → None
Returns False; for consistency with *CappedSemaphore*.

locked() → None
Returns true if a call to acquire would block.

release(*blocking=True*) → None
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

The *blocking* argument is for consistency with CappedSemaphore and is ignored

1.3.8 `guv.hubs.switch` - facilities for cooperative yielding

guv.hubs.switch.**gyield**(*switch_back=True*)
Yield to other greenlets

This is a cooperative yield which suspends the current greenlet and allows other greenlets to run by switching to the hub.

- If *switch_back* is True (default), the current greenlet is resumed at the beginning of the next event loop iteration, before the loop polls for I/O and calls any I/O callbacks. This is the intended use for this function the vast majority of the time.
- If *switch_back* is False, the hub will will never resume the current greenlet (use with caution). This is mainly useful for situations where other greenlets (not the hub) are responsible for switching back to this greenlet. An example is the Event class, where waiters are switched to when the event is ready.

Parameters **switch_back** (*bool*) – automatically switch back to this greenlet on the next event loop cycle

guv.hubs.switch.**trampoline**(*fd*, *evtype*, *timeout=None*, *timeout_exc=<class 'guv.timeout.Timeout'>*)
Jump from the current greenlet to the hub and wait until the given file descriptor is ready for I/O, or the specified timeout elapses

If the specified *timeout* elapses before the socket is ready to read or write, *timeout_exc* will be raised instead of *trampoline()* returning normally.

When the specified file descriptor is ready for I/O, the hub internally calls the callback to switch back to the current (this) greenlet.

Conditions:

- must not be called from the hub greenlet (can be called from any other greenlet)
- *evtype* must be either *READ* or *WRITE* (not possible to watch for both simultaneously)

Parameters

- **fd** (*int*) – file descriptor
- **evtype** (*int*) – either the constant *READ* or *WRITE*
- **timeout** (*float*) – (optional) maximum time to wait in seconds
- **timeout_exc** (*Exception*) – (optional) timeout Exception class

1.3.9 guv.timeout - universal timeouts

exception guv.timeout.**Timeout**(*seconds=None*, *exception=None*)

Bases: BaseException

Raise *exception* in the current greenthread after *timeout* seconds.

When *exception* is omitted or None, the *Timeout* instance itself is raised. If *seconds* is None, the timer is not scheduled, and is only useful if you're planning to raise it directly.

Timeout objects are context managers, and so can be used in with statements. When used in a with statement, if *exception* is False, the timeout is still raised, but the context manager suppresses it, so the code outside the with-block won't see it.

pending

@property

True if the timeout is scheduled to be raised

__init__(*seconds=None*, *exception=None*)

Parameters

- **seconds** (*float*) – timeout seconds
- **exception** – exception to raise when timeout occurs

cancel() → None

If the timeout is pending, cancel it

If not using Timeouts in with statements, always call cancel() in a finally after the block of code that is getting timed out. If not canceled, the timeout will be raised later on, in some unexpected section of the application.

start() → None

Schedule the timeout. This is called on construction, so it should not be called explicitly, unless the timer has been canceled.

guv.timeout.**with_timeout**(*seconds*, *function*, **args*, ***kwds*)

Wrap a call to some (yielding) function with a timeout

If the called function fails to return before the timeout, cancel it and return a flag value.

1.3.10 guv.websocket - websocket server

class guv.websocket.**WebSocketWSGI**(*handler*)
Bases: object

Wraps a websocket handler function in a WSGI application.

Use it like this:

```
@websocket.WebSocketWSGI
def my_handler(ws):
    from_browser = ws.wait()
    ws.send("from server")
```

The single argument to the function will be an instance of *WebSocket*. To close the socket, simply return from the function. Note that the server will log the websocket request at the time of closure.

class guv.websocket.**WebSocket**(*sock*, *environ*, *version=76*)
Bases: object

A websocket object that handles the details of serialization/deserialization to the socket.

The primary way to interact with a *WebSocket* object is to call *send()* and *wait()* in order to pass messages back and forth with the browser. Also available are the following properties:

path The path value of the request. This is the same as the WSGI PATH_INFO variable, but more convenient.

protocol The value of the Websocket-Protocol header.

origin The value of the 'Origin' header.

environ The full WSGI environment for this request.

__init__(*sock*, *environ*, *version=76*)

Parameters

- **socket** (eventlet.greenio.GreenSocket) – The guv socket
- **environ** – The wsgi environment
- **version** – The WebSocket spec version to follow (default is 76)

close() → None
Forcibly close the websocket; generally it is preferable to return from the handler method.

send(*message*) → None
Send a message to the browser.

message should be convertable to a string; unicode objects should be encodable as utf-8. Raises socket.error with errno of 32 (broken pipe) if the socket has already been closed by the client.

wait() → None
Waits for and deserializes messages.

Returns a single message; the oldest not yet processed. If the client has already closed the connection, returns None. This is different from normal socket behavior because the empty string is a valid websocket message.

1.3.11 guv.wsgi - WSGI server

guv.wsgi.**serve**(*server_sock*, *app*, *log_output=True*)
Start up a WSGI server handling requests from the supplied server socket

This function loops forever. The *sock* object will be closed after server exits, but the underlying file descriptor will remain open, so if you have a dup() of *sock*, it will remain usable.

Parameters

- **server_sock** – server socket, must be already bound to a port and listening
- **app** – WSGI application callable

guv.wsgi.**format_date_time**(*timestamp*)
Format a unix timestamp into an HTTP standard string

CHAPTER 2

Introduction

guv is a fast networking library and WSGI server (like gevent/eventlet) for **Python >= 3.2 and pypy3**

The event loop backend is pyuv_cffi, which aims to be fully compatible with the pyuv interface. pyuv_cffi is fully supported on CPython and pypy3. libuv >= 1.0.0 is required.

Asynchronous DNS queries are supported via dnspython3. To forcefully disable greendns, set the environment variable `GUV_NO_GREENDNS` to any value.

guv currently only runs on POSIX-compliant operating systems, but Windows support is not far off and can be added in the near future if there is a demand for this.

This library is actively maintained and has a zero bug policy. Please submit issues and pull requests, and bugs will be fixed immediately.

This project is under active development and any help is appreciated.

CHAPTER 3

Quickstart

Since guv is currently in alpha release state and under active development, it is recommended to pull often and install manually:

```
git clone https://github.com/veegee/guv.git
cd guv
python setup.py install
```

Note: libuv >= 1.0.0 is required. This is the first stable version but is a recent release and may not be available in Debian/Ubuntu stable repositories, so you must compile and install manually.

3.1 Serve your WSGI app using guv directly

```
import guv; guv.monkey_patch()
import guv.wsgi

app = <your WSGI app>

if __name__ == '__main__':
    server_sock = guv.listen(('0.0.0.0', 8001))
    guv.wsgi.serve(server_sock, app)
```

3.2 Serve your WSGI app using guv with gunicorn

```
gunicorn -w 4 -b 127.0.0.1:8001 -k guv.GuvWorker wsgi_app:app
```

Note: you can use wrk to benchmark the performance of guv.

3.3 Crawl the web: efficiently make multiple "simultaneous" requests

```
import guv; guv.monkey_patch()
import requests

def get_url(url):
    print('get_url({})'.format(url))
    return requests.get(url)
```

```
def main():
    urls = ['http://gnu.org'] * 10
    urls += ['https://eff.org'] * 10

    pool = guv.GreenPool()
    results = pool.starmap(get_url, zip(urls))

    for i, resp in enumerate(results):
        print('{}: done, length: {}'.format(i, len(resp.text)))

if __name__ == '__main__':
    main()
```

CHAPTER 4

Guarantees

This library makes the following guarantees:

- Semantic versioning is strictly followed
- Compatible with Python >= 3.2.0 and PyPy3 >= 2.3.1 (Python 3.2.5)

CHAPTER 5

Testing

guv uses the excellent **tox** and **pytest** frameworks. To run all tests, run in the project root:

```
$ pip install pytest
$ py.test
```

g

Symbols

A

B

C

E

F

G

I

J

K

L