# Fasteners Documentation

***Release 0.17.3***

**Joshua Harlow**

**May 16, 2022**

# CONTENTS

A python package that provides useful locks.

*Contents:*

CHAPTER

ONE

# LOCK

## 1.1 Classes

**class** fasteners.lock.**ReaderWriterLock**(*condition_cls=<class 'threading.Condition'>, current_thread_functor=None*)

A reader/writer lock.

This lock allows for simultaneous readers to exist but only one writer to exist for use-cases where it is useful to have such types of locks.

Currently a reader can not escalate its read lock to a write lock and a writer can not acquire a read lock while it is waiting on the write lock.

In the future these restrictions may be relaxed.

This can be eventually removed if http://bugs.python.org/issue8800 ever gets accepted into the python standard threading library...

**READER = 'r'**

Reader owner type/string constant.

**WRITER = 'w'**

Writer owner type/string constant.

**property has_pending_writers**

Returns if there are writers waiting to become the *one* writer.

**is_reader**()

Returns if the caller is one of the readers.

**is_writer**(*check_pending=True*)

Returns if the caller is the active writer or a pending writer.

**property owner**

Returns whether the lock is locked by a writer or reader.

**read_lock**()

Context manager that grants a read lock.

Will wait until no active or pending writers.

Raises a `RuntimeError` if a pending writer tries to acquire a read lock.

**write_lock**()

Context manager that grants a write lock.

Will wait until no active readers. Blocks readers after acquiring.

Guaranteed for locks to be processed in fair order (FIFO).

Raises a `RuntimeError` if an active reader attempts to acquire a lock.

## 1.2 Decorators

`fasteners.lock.`**`read_locked`**`(`**args*, ***kwargs*`)`

Acquires & releases a read lock around call into decorated method.

NOTE(harlowja): if no attribute name is provided then by default the attribute named '_lock' is looked for (this attribute is expected to be a `ReaderWriterLock`) in the instance object this decorator is attached to.

`fasteners.lock.`**`write_locked`**`(`**args*, ***kwargs*`)`

Acquires & releases a write lock around call into decorated method.

NOTE(harlowja): if no attribute name is provided then by default the attribute named '_lock' is looked for (this attribute is expected to be a `ReaderWriterLock` object) in the instance object this decorator is attached to.

`fasteners.lock.`**`locked`**`(`**args*, ***kwargs*`)`

A locking **method** decorator.

It will look for a provided attribute (typically a lock or a list of locks) on the first argument of the function decorated (typically this is the 'self' object) and before executing the decorated function it activates the given lock or list of locks as a context manager, automatically releasing that lock on exit.

NOTE(harlowja): if no attribute name is provided then by default the attribute named '_lock' is looked for (this attribute is expected to be the lock/list of locks object/s) in the instance object this decorator is attached to.

NOTE(harlowja): a custom logger (which will be used if lock release failures happen) can be provided by passing a logger instance for keyword argument `logger`.

## 1.3 Helper functions

`fasteners.lock.`**`try_lock`**`(`*lock*`)`

Attempts to acquire a lock, and auto releases if acquired (on exit).

CHAPTER

TWO

# PROCESS LOCK

Fasteners inter-process locks are cross-platform and are released automatically if the process crashes. They are based on the platform specific locking mechanisms:

- fcntl for posix (Linux and OSX)
- LockFileEx (via pywin32) and _locking (via msvcrt) for Windows

The intersection of fcntl and LockFileEx features is quite small, hence you should always assume that:

- Locks are advisory. They do not prevent the modification of the locked file by other processes.
- Locks can be unintentionally released by simply opening and closing the file descriptor, so lock files must be accessed only using provided abstractions.
- Locks are not reentrant. An attempt to acquire a lock multiple times can result in a deadlock or a crash upon a release of the lock.
- Reader writer locks are not upgradeable. An attempt to get a reader's lock while holding a writer's lock (or vice versa) can result in a deadlock or a crash upon a release of the lock.
- There are no guarantees regarding usage by multiple threads in a single process. The locks work only between processes.

To learn more about the complications of locking on different platforms we recommend the following resources:

- File locking in Linux (blog post)
- On the Brokenness of File Locking (blog post)
- Everything you never wanted to know about file locking (blog post)
- Record Locking (course notes)
- Windows NT Files – Locking (pywin32 docs)
- _locking (Windows Dev Center)
- LockFileEx function (Windows Dev Center)

## 2.1 Classes

`fasteners.process_lock.`**`InterProcessLock`**
alias of `fasteners.process_lock._FcntlLock`

**`class`** `fasteners.process_lock.`**`_InterProcessLock`**(*path*, *sleep_func=<built-in function sleep>*, *logger=None*)
An interprocess lock.

**`DELAY_INCREMENT = 0.01`**
Default increment we will use (up to max delay) after each attempt before next attempt to acquire the lock. For example if 3 attempts have been made the calling thread will sleep (0.01 * 3) before the next attempt to acquire the lock (and repeat).

**`MAX_DELAY = 0.1`**
Default maximum delay we will wait to try to acquire the lock (when it's busy/being held by another process).

**`acquire`**(*blocking=True*, *delay=0.01*, *max_delay=0.1*, *timeout=None*)
Attempt to acquire the given lock.

**Parameters**

- **`blocking`** (*`bool`*) – whether to wait forever to try to acquire the lock
- **`delay`** (*`int/float`*) – when blocking this is the delay time in seconds that will be added after each failed acquisition
- **`max_delay`** (*`int/float`*) – the maximum delay to have (this limits the accumulated delay(s) added after each failed acquisition)
- **`timeout`** (*`int/float`*) – an optional timeout (limits how long blocking will occur for)

**Returns** whether or not the acquisition succeeded

**Return type** bool

**`exists`**()
Checks if the path that this lock exists at actually exists.

**`release`**()
Release the previously acquired lock.

`fasteners.process_lock.`**`InterProcessReaderWriterLock`**
alias of `fasteners.process_lock._FcntlInterProcessReaderWriterLock`

**`class`** `fasteners.process_lock.`**`_InterProcessReaderWriterLock`**(*path*, *sleep_func=<built-in function sleep>*, *logger=None*)
An interprocess readers writer lock.

**`DELAY_INCREMENT = 0.01`**
Default increment we will use (up to max delay) after each attempt before next attempt to acquire the lock. For example if 3 attempts have been made the calling thread will sleep (0.01 * 3) before the next attempt to acquire the lock (and repeat).

**`MAX_DELAY = 0.1`**
Default maximum delay we will wait to try to acquire the lock (when it's busy/being held by another process).

**`acquire_read_lock`**(*blocking=True*, *delay=0.01*, *max_delay=0.1*, *timeout=None*)
Attempt to acquire a reader's lock.

**Parameters**

- **`blocking`** (*`bool`*) – whether to wait forever to try to acquire the lock
- **`delay`** (*`int/float`*) – when blocking this is the delay time in seconds that will be added after each failed acquisition
- **`max_delay`** (*`int/float`*) – the maximum delay to have (this limits the accumulated delay(s) added after each failed acquisition)
- **`timeout`** (*`int/float`*) – an optional timeout (limits how long blocking will occur for)

**Returns** whether or not the acquisition succeeded

**Return type** bool

**`acquire_write_lock`**(*blocking=True*, *delay=0.01*, *max_delay=0.1*, *timeout=None*)

Attempt to acquire a writer's lock.

**Parameters**

- **`blocking`** (*`bool`*) – whether to wait forever to try to acquire the lock
- **`delay`** (*`int/float`*) – when blocking this is the delay time in seconds that will be added after each failed acquisition
- **`max_delay`** (*`int/float`*) – the maximum delay to have (this limits the accumulated delay(s) added after each failed acquisition)
- **`timeout`** (*`int/float`*) – an optional timeout (limits how long blocking will occur for)

**Returns** whether or not the acquisition succeeded

**Return type** bool

**`release_read_lock`**()

Release the reader's lock.

**`release_write_lock`**()

Release the writer's lock.

## 2.2 Decorators

`fasteners.process_lock.`**`interprocess_locked`**(*path*)

Acquires & releases a interprocess lock around call into decorated function.

`fasteners.process_lock.`**`interprocess_read_locked`**(*path*)

Acquires & releases an interprocess read lock around the call into the decorated function

`fasteners.process_lock.`**`interprocess_write_locked`**(*path*)

Acquires & releases an interprocess read lock around the call into the decorated function

# CHAPTER THREE

# EXAMPLES

## 3.1 Inter-process locks

**Note:** Launch multiple of these at the same time to see the lock(s) in action.

**Warning:** There are no guarantees regarding usage by multiple threads in a single process with these locks (you will have to ensure single process safety yourself using traditional thread based locks). In other words this lock works **only** between processes.

### 3.1.1 Lock API

Using a decorator:

```
import time

import fasteners

@fasteners.interprocess_locked('/tmp/tmp_lock_file')
def test():
    for i in range(10):
        print('I have the lock')
        time.sleep(1)

print('Waiting for the lock')
test()
```

Using a context manager:

```
import time

import fasteners

def test():
    for i in range(10):
        with fasteners.InterProcessLock('/tmp/tmp_lock_file'):
            print('I have the lock')
            time.sleep(1)

test()
```

Acquiring and releasing manually:

```
import time

import fasteners

def test():
    a_lock = fasteners.InterProcessLock('/tmp/tmp_lock_file')
    for i in range(10):
        gotten = a_lock.acquire(blocking=False)
        try:
            if gotten:
                print('I have the lock')
                time.sleep(0.2)
            else:
                print('I do not have the lock')
                time.sleep(0.1)
        finally:
            if gotten:
                a_lock.release()

test()
```

### 3.1.2 Reader Writer Lock API

Reader lock using a decorator:

```
import time

import fasteners

@fasteners.interprocess_read_locked('/tmp/tmp_lock_file')
def test():
    for i in range(10):
        print('I have the readers lock')
        time.sleep(1)

print('Waiting for the lock')
test()
```

Writer lock using a context manager:

```
import time

import fasteners

def test():
    for i in range(10):
        with fasteners.InterProcessReaderWriterLock('/tmp/tmp_lock_file').write_
↪lock():
            print('I have the writers lock')
            time.sleep(1)

test()
```

Acquiring and releasing manually:

```
import time

import fasteners

def test():
    a_lock = fasteners.InterProcessReaderWriterLock('/tmp/tmp_lock_file')
    for i in range(10):
        gotten = a_lock.acquire_read_lock(blocking=False)
        try:
            if gotten:
                print('I have the readers lock')
                time.sleep(0.2)
            else:
                print('I do not have the readers lock')
                time.sleep(0.1)
        finally:
            if gotten:
                a_lock.release_read_lock()

test()
```

## 3.2 Inter-thread locks

### 3.2.1 Lock API

Using a decorator:

```
import threading

import fasteners

class NotThreadSafeThing(object):
    def __init__(self):
        self._lock = threading.Lock()

    @fasteners.locked
    def do_something(self):
        print("Doing something in a thread safe manner")

o = NotThreadSafeThing()
o.do_something()
```

Multiple locks using a single decorator:

```
import threading

import fasteners

class NotThreadSafeThing(object):
    def __init__(self):
        self._locks = [threading.Lock(), threading.Lock()]

    @fasteners.locked(lock='_locks')
    def do_something(self):
```

(continues on next page)

(continued from previous page)

```
        print("Doing something in a thread safe manner")

o = NotThreadSafeThing()
o.do_something()
```

Manual lock without blocking:

```
import threading

import fasteners

t = threading.Lock()
with fasteners.try_lock(t) as gotten:
    if gotten:
        print("I got the lock")
    else:
        print("I did not get the lock")
```

### 3.2.2 Reader Writer lock API

Using a context manager:

```
import random
import threading
import time

import fasteners

def read_something(ident, rw_lock):
    with rw_lock.read_lock():
        print("Thread %s is reading something" % ident)
        time.sleep(1)

def write_something(ident, rw_lock):
    with rw_lock.write_lock():
        print("Thread %s is writing something" % ident)
        time.sleep(2)

rw_lock = fasteners.ReaderWriterLock()
threads = []
for i in range(0, 10):
    is_writer = random.choice([True, False])
    if is_writer:
        threads.append(threading.Thread(target=write_something,
                                        args=(i, rw_lock)))
    else:
        threads.append(threading.Thread(target=read_something,
                                        args=(i, rw_lock)))

try:
    for t in threads:
        t.start()
finally:
    while threads:
```

(continues on next page)

(continued from previous page)

```
        t = threads.pop()
        t.join()
```

CHAPTER

# FOUR

## INDICES AND TABLES

- genindex
- modindex
- search

# INDEX