Portalocker Documentation

Release 2.6.0

Rick van Hattem

Oct 18, 2022

CONTENTS

1 portalocker - Cross-platform locking library **1**
- 1.1 Overview . . . 1
- 1.2 Redis Locks . . . 1
- 1.3 Python 2 . . . 2
- 1.4 Tips . . . 2
- 1.5 Links . . . 2
- 1.6 Examples . . . 3
- 1.7 Versioning . . . 4
- 1.8 Changelog . . . 4
- 1.9 License . . . 4
  - 1.9.1 portalocker package . . . 4
    - 1.9.1.1 Submodules . . . 4
    - 1.9.1.2 Module contents . . . 9
  - 1.9.2 tests package . . . 13
    - 1.9.2.1 Module contents . . . 13
  - 1.9.3 License . . . 16

2 Indices and tables **17**

Python Module Index **19**

Index **21**

CHAPTER

ONE

PORTALOCKER - CROSS-PLATFORM LOCKING LIBRARY

1.1 Overview

Portalocker is a library to provide an easy API to file locking.

An important detail to note is that on Linux and Unix systems the locks are advisory by default. By specifying the *-o mand* option to the mount command it is possible to enable mandatory file locking on Linux. This is generally not recommended however. For more information about the subject:

- https://en.wikipedia.org/wiki/File_locking
- http://stackoverflow.com/questions/39292051/portalocker-does-not-seem-to-lock
- https://stackoverflow.com/questions/12062466/mandatory-file-lock-on-linux

The module is currently maintained by Rick van Hattem <Wolph@wol.ph>. The project resides at https://github.com/WoLpH/portalocker . Bugs and feature requests can be submitted there. Patches are also very welcome.

1.2 Redis Locks

This library now features a lock based on Redis which allows for locks across multiple threads, processes and even distributed locks across multiple computers.

It is an extremely reliable Redis lock that is based on pubsub.

As opposed to most Redis locking systems based on key/value pairs, this locking method is based on the pubsub system. The big advantage is that if the connection gets killed due to network issues, crashing processes or otherwise, it will still immediately unlock instead of waiting for a lock timeout.

First make sure you have everything installed correctly:

```
pip install "portalocker[redis]"
```

Usage is really easy:

```
import portalocker

lock = portalocker.RedisLock('some_lock_channel_name')

with lock:
    print('do something here')
```

The API is essentially identical to the other `Lock` classes so in addition to the `with` statement you can also use `lock.acquire(...)`.

1.3 Python 2

Python 2 was supported in versions before Portalocker 2.0. If you are still using Python 2, you can run this to install:

```
pip install "portalocker<2"
```

1.4 Tips

On some networked filesystems it might be needed to force a *os.fsync()* before closing the file so it's actually written before another client reads the file. Effectively this comes down to:

```
with portalocker.Lock('some_file', 'rb+', timeout=60) as fh:
    # do what you need to do
    ...

    # flush and sync to filesystem
    fh.flush()
    os.fsync(fh.fileno())
```

1.5 Links

- **Documentation**
  - http://portalocker.readthedocs.org/en/latest/
- **Source**
  - https://github.com/WoLpH/portalocker
- **Bug reports**
  - https://github.com/WoLpH/portalocker/issues
- **Package homepage**
  - https://pypi.python.org/pypi/portalocker
- **My blog**
  - http://w.wol.ph/

1.6 Examples

To make sure your cache generation scripts don't race, use the *Lock* class:

```
>>> import portalocker
>>> with portalocker.Lock('somefile', timeout=1) as fh:
...     print('writing some stuff to my cache...', file=fh)
```

To customize the opening and locking a manual approach is also possible:

```
>>> import portalocker
>>> file = open('somefile', 'r+')
>>> portalocker.lock(file, portalocker.LockFlags.EXCLUSIVE)
>>> file.seek(12)
>>> file.write('foo')
>>> file.close()
```

Explicitly unlocking is not needed in most cases but omitting it has been known to cause issues: https://github.com/AzureAD/microsoft-authentication-extensions-for-python/issues/42#issuecomment-601108266

If needed, it can be done through:

```
>>> portalocker.unlock(file)
```

Do note that your data might still be in a buffer so it is possible that your data is not available until you *flush()* or *close()*.

To create a cross platform bounded semaphore across multiple processes you can use the *BoundedSemaphore* class which functions somewhat similar to *threading.BoundedSemaphore*:

```
>>> import portalocker
>>> n = 2
>>> timeout = 0.1
```

```
>>> semaphore_a = portalocker.BoundedSemaphore(n, timeout=timeout)
>>> semaphore_b = portalocker.BoundedSemaphore(n, timeout=timeout)
>>> semaphore_c = portalocker.BoundedSemaphore(n, timeout=timeout)
```

```
>>> semaphore_a.acquire()
<portalocker.utils.Lock object at ...>
>>> semaphore_b.acquire()
<portalocker.utils.Lock object at ...>
>>> semaphore_c.acquire()
Traceback (most recent call last):
  ...
portalocker.exceptions.AlreadyLocked
```

More examples can be found in the tests.

1.7 Versioning

This library follows Semantic Versioning.

1.8 Changelog

Every release has a `git tag` with a commit message for the tag explaining what was added and/or changed. The list of tags/releases including the commit messages can be found here: https://github.com/WoLpH/portalocker/releases

1.9 License

See the LICENSE file.

Contents:

1.9.1 portalocker package

1.9.1.1 Submodules

portalocker.redis module

class portalocker.redis.**PubSubWorkerThread**(*pubsub*, *sleep_time*, *daemon=False*, *exception_handler=None*)

Bases: PubSubWorkerThread

run()

Method representing the thread'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.

class portalocker.redis.**RedisLock**(*channel: str*, *connection: Optional[Redis] = None*, *timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = False*, *thread_sleep_time: float = 0.1*, *unavailable_timeout: float = 1*, *redis_kwargs: Optional[Dict] = None*)

Bases: LockBase

An extremely reliable Redis lock based on pubsub with a keep-alive thread

As opposed to most Redis locking systems based on key/value pairs, this locking method is based on the pubsub system. The big advantage is that if the connection gets killed due to network issues, crashing processes or otherwise, it will still immediately unlock instead of waiting for a lock timeout.

To make sure both sides of the lock know about the connection state it is recommended to set the *health_check_interval* when creating the redis connection..

Parameters

- **channel** – the redis channel to use as locking key.

- **connection** (*or if you need to specify the redis*) – an optional redis connection if you already have one
- **connection** –
- **timeout** – timeout when trying to acquire a lock
- **check_interval** – check interval while waiting
- **fail_when_locked** – after the initial lock failed, return an error or lock the file. This does not wait for the timeout.
- **thread_sleep_time** – sleep time between fetching messages from redis to prevent a busy/wait loop. In the case of lock conflicts this increases the time it takes to resolve the conflict. This should be smaller than the *check_interval* to be useful.
- **unavailable_timeout** – If the conflicting lock is properly connected this should never exceed twice your redis latency. Note that this will increase the wait time possibly beyond your *timeout* and is always executed if a conflict arises.
- **redis_kwargs** – The redis connection arguments if no connection is given. The *DEFAULT_REDIS_KWARGS* are used as default, if you want to override these you need to explicitly specify a value (e.g. *health_check_interval=0*)

DEFAULT_REDIS_KWARGS = {'health_check_interval': 10}

acquire(*timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = None*)

channel: str

channel_handler(*message*)

check_interval: float

check interval while waiting for *timeout*

check_or_kill_lock(*connection*, *timeout*)

property client_name

close_connection: bool

connection: Optional[Redis]

fail_when_locked: bool

skip the timeout and immediately fail if the initial lock fails

get_connection() → Redis

pubsub: Optional[PubSub] = None

redis_kwargs: Dict[str, Any]

release()

thread: Optional[*PubSubWorkerThread*]

timeout: float

timeout when trying to acquire a lock

portalocker.constants module

Locking constants

Lock types:

- *EXCLUSIVE* exclusive lock
- *SHARED* shared lock

Lock flags:

- *NON_BLOCKING* non-blocking

Manually unlock, only needed internally

- *UNBLOCK* unlock

portalocker.constants.**LOCK_EX = 2**
exclusive lock

portalocker.constants.**LOCK_NB = 4**
non-blocking

portalocker.constants.**LOCK_SH = 1**
shared lock

portalocker.constants.**LOCK_UN = 8**
unlock

class portalocker.constants.**LockFlags**(*value*)
Bases: IntFlag

An enumeration.

EXCLUSIVE = 2
exclusive lock

NON_BLOCKING = 4
non-blocking

SHARED = 1
shared lock

UNBLOCK = 8
unlock

__annotations__ = {}

__module__ = 'portalocker.constants'

portalocker.exceptions module

exception portalocker.exceptions.**AlreadyLocked**(**args: Any*, *fh: Optional[IO] = None*, ***kwargs: Any*)

Bases: *LockException*

exception portalocker.exceptions.**BaseLockException**(**args: Any*, *fh: Optional[IO] = None*, ***kwargs: Any*)

Bases: Exception

LOCK_FAILED = 1

exception portalocker.exceptions.**FileToLarge**(**args: Any*, *fh: Optional[IO] = None*, ***kwargs: Any*)

Bases: *LockException*

exception portalocker.exceptions.**LockException**(**args: Any*, *fh: Optional[IO] = None*, ***kwargs: Any*)

Bases: *BaseLockException*

portalocker.portalocker module

portalocker.portalocker.**lock**(*file_: IO*, *flags:* LockFlags)

portalocker.portalocker.**unlock**(*file_: IO*)

portalocker.utils module

class portalocker.utils.**Lock**(*filename: ~typing.Union[str, ~pathlib.Path], mode: str = 'a', timeout: ~typing.Optional[float] = None, check_interval: float = 0.25, fail_when_locked: bool = False, flags: ~portalocker.constants.LockFlags = LockFlags.None, **file_open_kwargs*)

Bases: LockBase

Lock manager with built-in timeout

Parameters

- **filename** – filename
- **mode** – the open mode, 'a' or 'ab' should be used for writing
- **truncate** – use truncate to emulate 'w' mode, None is disabled, 0 is truncate to 0 bytes
- **timeout** – timeout when trying to acquire a lock
- **check_interval** – check interval while waiting
- **fail_when_locked** – after the initial lock failed, return an error or lock the file. This does not wait for the timeout.
- ****file_open_kwargs** – The kwargs for the *open(...)* call

fail_when_locked is useful when multiple threads/processes can race when creating a file. If set to true than the system will wait till the lock was acquired and then return an AlreadyLocked exception.

Note that the file is opened first and locked later. So using 'w' as mode will result in truncate _BEFORE_ the lock is checked.

acquire(*timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = None*) → IO

Acquire the locked filehandle

check_interval: float

check interval while waiting for *timeout*

fail_when_locked: bool

skip the timeout and immediately fail if the initial lock fails

fh: Optional[IO]

filename: str

flags: *LockFlags*

mode: str

release()

Releases the currently locked file handle

timeout: float

timeout when trying to acquire a lock

truncate: bool

portalocker.utils.**open_atomic**(*filename: Union[str, Path]*, *binary: bool = True*) → Iterator[IO]

Open a file for atomic writing. Instead of locking this method allows you to write the entire file and move it to the actual location. Note that this makes the assumption that a rename is atomic on your platform which is generally the case but not a guarantee.

http://docs.python.org/library/os.html#os.rename

```
>>> filename = 'test_file.txt'
>>> if os.path.exists(filename):
...     os.remove(filename)
```

```
>>> with open_atomic(filename) as fh:
...     written = fh.write(b'test')
>>> assert os.path.exists(filename)
>>> os.remove(filename)
```

```
>>> import pathlib
>>> path_filename = pathlib.Path('test_file.txt')
```

```
>>> with open_atomic(path_filename) as fh:
...     written = fh.write(b'test')
>>> assert path_filename.exists()
>>> path_filename.unlink()
```

1.9.1.2 Module contents

exception portalocker.**AlreadyLocked**(**args: Any*, *fh: Optional[IO] = None*, ***kwargs: Any*)

Bases: *LockException*

class portalocker.**BoundedSemaphore**(*maximum: int*, *name: str = 'bounded_semaphore'*, *filename_pattern: str = '{name}.{number:02d}.lock'*, *directory: str = '/tmp'*, *timeout=5*, *check_interval=0.25*)

Bases: LockBase

Bounded semaphore to prevent too many parallel processes from running

It's also possible to specify a timeout when acquiring the lock to wait for a resource to become available. This is very similar to threading.BoundedSemaphore but works across multiple processes and across multiple operating systems.

```
>>> semaphore = BoundedSemaphore(2, directory='')
>>> str(semaphore.get_filenames()[0])
'bounded_semaphore.00.lock'
>>> str(sorted(semaphore.get_random_filenames())[1])
'bounded_semaphore.01.lock'
```

acquire(*timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = None*) → Optional[*Lock*]

get_filename(*number*) → Path

get_filenames() → Sequence[Path]

get_random_filenames() → Sequence[Path]

lock: Optional[*Lock*]

release()

try_lock(*filenames: Sequence[Union[str, Path]]*) → bool

portalocker.**LOCK_EX: *LockFlags* = LockFlags.EXCLUSIVE**

Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

portalocker.**LOCK_NB: *LockFlags* = LockFlags.NON_BLOCKING**

Acquire the lock in a non-blocking fashion.

portalocker.**LOCK_SH: *LockFlags* = LockFlags.SHARED**

Place a shared lock. More than one process may hold a shared lock for a given file at a given time.

portalocker.**LOCK_UN: *LockFlags* = LockFlags.UNBLOCK**

Remove an existing lock held by this process.

class portalocker.**Lock**(*filename: ~typing.Union[str, ~pathlib.Path]*, *mode: str = 'a'*, *timeout: ~typing.Optional[float] = None*, *check_interval: float = 0.25*, *fail_when_locked: bool = False*, *flags: ~portalocker.constants.LockFlags = LockFlags.None*, ***file_open_kwargs*)

Bases: LockBase

Lock manager with built-in timeout

Parameters

- **filename** – filename
- **mode** – the open mode, 'a' or 'ab' should be used for writing
- **truncate** – use truncate to emulate 'w' mode, None is disabled, 0 is truncate to 0 bytes
- **timeout** – timeout when trying to acquire a lock
- **check_interval** – check interval while waiting
- **fail_when_locked** – after the initial lock failed, return an error or lock the file. This does not wait for the timeout.
- ****file_open_kwargs** – The kwargs for the *open(...)* call

fail_when_locked is useful when multiple threads/processes can race when creating a file. If set to true than the system will wait till the lock was acquired and then return an AlreadyLocked exception.

Note that the file is opened first and locked later. So using 'w' as mode will result in truncate _BEFORE_ the lock is checked.

acquire(*timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = None*) → IO

Acquire the locked filehandle

check_interval: float

check interval while waiting for *timeout*

fail_when_locked: bool

skip the timeout and immediately fail if the initial lock fails

release()

Releases the currently locked file handle

timeout: float

timeout when trying to acquire a lock

exception portalocker.**LockException**(**args: Any*, *fh: Optional[IO] = None*, ***kwargs: Any*)

Bases: *BaseLockException*

class portalocker.**LockFlags**(*value*)

Bases: IntFlag

An enumeration.

EXCLUSIVE = 2

exclusive lock

NON_BLOCKING = 4

non-blocking

SHARED = 1

shared lock

UNBLOCK = 8

unlock

class portalocker.**RLock**(*filename*, *mode='a'*, *timeout=5*, *check_interval=0.25*, *fail_when_locked=False*, *flags=LockFlags.None*)

Bases: *Lock*

A reentrant lock, functions in a similar way to threading.RLock in that it can be acquired multiple times. When the corresponding number of release() calls are made the lock will finally release the underlying file lock.

acquire(*timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = None*) → IO

Acquire the locked filehandle

check_interval: float

check interval while waiting for *timeout*

fail_when_locked: bool

skip the timeout and immediately fail if the initial lock fails

fh: Optional[IO]

filename: str

flags: *LockFlags*

mode: str

release()

Releases the currently locked file handle

timeout: float

timeout when trying to acquire a lock

truncate: bool

class portalocker.**RedisLock**(*channel: str*, *connection: Optional[Redis] = None*, *timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = False*, *thread_sleep_time: float = 0.1*, *unavailable_timeout: float = 1*, *redis_kwargs: Optional[Dict] = None*)

Bases: LockBase

An extremely reliable Redis lock based on pubsub with a keep-alive thread

As opposed to most Redis locking systems based on key/value pairs, this locking method is based on the pubsub system. The big advantage is that if the connection gets killed due to network issues, crashing processes or otherwise, it will still immediately unlock instead of waiting for a lock timeout.

To make sure both sides of the lock know about the connection state it is recommended to set the *health_check_interval* when creating the redis connection..

Parameters

- **channel** – the redis channel to use as locking key.
- **connection** (*or if you need to specify the redis*) – an optional redis connection if you already have one
- **connection** –
- **timeout** – timeout when trying to acquire a lock
- **check_interval** – check interval while waiting
- **fail_when_locked** – after the initial lock failed, return an error or lock the file. This does not wait for the timeout.
- **thread_sleep_time** – sleep time between fetching messages from redis to prevent a busy/wait loop. In the case of lock conflicts this increases the time it takes to resolve the conflict. This should be smaller than the *check_interval* to be useful.

- **unavailable_timeout** – If the conflicting lock is properly connected this should never exceed twice your redis latency. Note that this will increase the wait time possibly beyond your *timeout* and is always executed if a conflict arises.
- **redis_kwargs** – The redis connection arguments if no connection is given. The *DEFAULT_REDIS_KWARGS* are used as default, if you want to override these you need to explicitly specify a value (e.g. *health_check_interval=0*)

DEFAULT_REDIS_KWARGS = {'health_check_interval': 10}

acquire(*timeout: Optional[float] = None*, *check_interval: Optional[float] = None*, *fail_when_locked: Optional[bool] = None*)

channel: str

channel_handler(*message*)

check_or_kill_lock(*connection*, *timeout*)

property client_name

close_connection: bool

connection: Optional[Redis]

get_connection() → Redis

pubsub: Optional[PubSub] = None

redis_kwargs: Dict[str, Any]

release()

thread: Optional[*PubSubWorkerThread*]

timeout: float

timeout when trying to acquire a lock

portalocker.**lock**(*file_: IO*, *flags:* LockFlags)

Lock a file. Note that this is an advisory lock on Linux/Unix systems

portalocker.**open_atomic**(*filename: Union[str, Path]*, *binary: bool = True*) → Iterator[IO]

Open a file for atomic writing. Instead of locking this method allows you to write the entire file and move it to the actual location. Note that this makes the assumption that a rename is atomic on your platform which is generally the case but not a guarantee.

http://docs.python.org/library/os.html#os.rename

```
>>> filename = 'test_file.txt'
>>> if os.path.exists(filename):
...     os.remove(filename)
```

```
>>> with open_atomic(filename) as fh:
...     written = fh.write(b'test')
>>> assert os.path.exists(filename)
>>> os.remove(filename)
```

```
>>> import pathlib
>>> path_filename = pathlib.Path('test_file.txt')
```

```
>>> with open_atomic(path_filename) as fh:
...     written = fh.write(b'test')
>>> assert path_filename.exists()
>>> path_filename.unlink()
```

portalocker.**unlock**(*file_: IO*)

Unlock a file

1.9.2 tests package

1.9.2.1 Module contents

class portalocker_tests.tests.**LockResult**(*exception_class: Union[Type[Exception], NoneType] = None*, *exception_message: Union[str, NoneType] = None*, *exception_repr: Union[str, NoneType] = None*)

Bases: object

__annotations__ = {'exception_class': typing.Union[typing.Type[Exception], NoneType], 'exception_message': typing.Union[str, NoneType], 'exception_repr': typing.Union[str, NoneType]}

__dataclass_fields__ = {'exception_class': Field(name='exception_class',type=typing.Union[typing.Type[Exception], NoneType],default=None,default_factory=<dataclasses._MISSING_TYPE object>,init=True, repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD), 'exception_message': Field(name='exception_message',type=typing.Union[str, NoneType],default=None,default_factory=<dataclasses._MISSING_TYPE object>,init=True, repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD), 'exception_repr': Field(name='exception_repr',type=typing.Union[str, NoneType],default=None,default_factory=<dataclasses._MISSING_TYPE object>,init=True, repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD)}

__dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,order=True, unsafe_hash=False,frozen=False)

__dict__ = mappingproxy({'__module__': 'portalocker_tests.tests', '__annotations__': {'exception_class': typing.Union[typing.Type[Exception], NoneType], 'exception_message': typing.Union[str, NoneType], 'exception_repr': typing.Union[str, NoneType]}, 'exception_class': None, 'exception_message': None, 'exception_repr': None, '__dict__': <attribute '__dict__' of 'LockResult' objects>, '__weakref__': <attribute '__weakref__' of 'LockResult' objects>, '__doc__': 'LockResult(exception_class: Union[Type[Exception], NoneType] = None, exception_message: Union[str, NoneType] = None, exception_repr: Union[str, NoneType] = None)', '__dataclass_params__': _DataclassParams(init=True,repr=True, eq=True,order=True,unsafe_hash=False,frozen=False), '__dataclass_fields__': {'exception_class': Field(name='exception_class',type=typing.Union[typing.Type[Exception], NoneType],default=None,default_factory=<dataclasses._MISSING_TYPE object>,init=True, repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD), 'exception_message': Field(name='exception_message',type=typing.Union[str, NoneType],default=None,default_factory=<dataclasses._MISSING_TYPE object>,init=True, repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD), 'exception_repr': Field(name='exception_repr',type=typing.Union[str, NoneType],default=None,default_factory=<dataclasses._MISSING_TYPE object>,init=True, repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD)}, '__init__': <function __create_fn__.<locals>.__init__>, '__repr__': <function __create_fn__.<locals>.__repr__>, '__eq__': <function __create_fn__.<locals>.__eq__>, '__lt__': <function __create_fn__.<locals>.__lt__>, '__le__': <function __create_fn__.<locals>.__le__>, '__gt__': <function __create_fn__.<locals>.__gt__>, '__ge__': <function __create_fn__.<locals>.__ge__>, '__hash__': None})

__eq__(*other*)

Return self==value.

__ge__(*other*)

Return self>=value.

__gt__(*other*)

Return self>value.

__hash__ = None

__init__(*exception_class: Optional[Type[Exception]] = None*, *exception_message: Optional[str] = None*, *exception_repr: Optional[str] = None*) → None

__le__(*other*)

Return self<=value.

__lt__(*other*)

Return self<value.

__module__ = 'portalocker_tests.tests'

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

exception_class: Optional[Type[Exception]] = None

exception_message: Optional[str] = None

exception_repr: Optional[str] = None

portalocker_tests.tests.**exclusive_lock**(*filename*, ***kwargs*)

portalocker_tests.tests.**lock**(*filename: str*, *fail_when_locked: bool*, *flags:* LockFlags) → *LockResult*

portalocker_tests.tests.**shared_lock**(*filename*, ***kwargs*)

portalocker_tests.tests.**shared_lock_fail**(*filename*, ***kwargs*)

portalocker_tests.tests.**test_acquire_release**(*tmpfile*)

portalocker_tests.tests.**test_blocking_timeout**(*tmpfile*)

portalocker_tests.tests.**test_class**(*tmpfile*)

portalocker_tests.tests.**test_exceptions**(*tmpfile*)

portalocker_tests.tests.**test_exclusive_processes**(*tmpfile*, *fail_when_locked*)

portalocker_tests.tests.**test_exlusive**(*tmpfile*)

portalocker_tests.tests.**test_lock_fileno**(*tmpfile*)

portalocker_tests.tests.**test_nonblocking**(*tmpfile*)

portalocker_tests.tests.**test_release_unacquired**(*tmpfile*)

portalocker_tests.tests.**test_rlock_acquire_release**(*tmpfile*)

portalocker_tests.tests.**test_rlock_acquire_release_count**(*tmpfile*)

portalocker_tests.tests.**test_shared**(*tmpfile*)

portalocker_tests.tests.**test_shared_processes**(*tmpfile*, *fail_when_locked*)

portalocker_tests.tests.**test_simple**(*tmpfile*)

portalocker_tests.tests.**test_truncate**(*tmpfile*)

portalocker_tests.tests.**test_utils_base**()

portalocker_tests.tests.**test_with_timeout**(*tmpfile*)

portalocker_tests.tests.**test_without_fail**(*tmpfile*)

portalocker_tests.tests.**test_without_timeout**(*tmpfile*)

portalocker_tests.test_combined.**test_combined**(*tmpdir*)

portalocker_tests.temporary_file_lock.**test_temporary_file_lock**(*tmpfile*)

1.9.3 License

```
Copyright 2022 Rick van Hattem

Redistribution and use in source and binary forms, with or without modification, are␣
↪permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of␣
↪conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list␣
↪of conditions and the following disclaimer in the documentation and/or other materials␣
↪provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be␣
↪used to endorse or promote products derived from this software without specific prior␣
↪written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY␣
↪EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES␣
↪OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT␣
↪SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,␣
↪INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED␣
↪TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR␣
↪BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN␣
↪CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY␣
↪WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

portalocker, 9
portalocker.constants, 6
portalocker.exceptions, 7
portalocker.portalocker, 7
portalocker.redis, 4
portalocker.utils, 7
portalocker_tests.temporary_file_lock, 15
portalocker_tests.test_combined, 15
portalocker_tests.tests, 13

INDEX

Symbols

__annotations__ (*portalocker.constants.LockFlags attribute*), 6
__annotations__ (*portalocker_tests.tests.LockResult attribute*), 13
__dataclass_fields__ (*portalocker_tests.tests.LockResult attribute*), 13
__dataclass_params__ (*portalocker_tests.tests.LockResult attribute*), 13
__dict__ (*portalocker_tests.tests.LockResult attribute*), 13
__eq__() (*portalocker_tests.tests.LockResult method*), 14
__ge__() (*portalocker_tests.tests.LockResult method*), 14
__gt__() (*portalocker_tests.tests.LockResult method*), 14
__hash__ (*portalocker_tests.tests.LockResult attribute*), 14
__init__() (*portalocker_tests.tests.LockResult method*), 14
__le__() (*portalocker_tests.tests.LockResult method*), 14
__lt__() (*portalocker_tests.tests.LockResult method*), 14
__module__ (*portalocker.constants.LockFlags attribute*), 6
__module__ (*portalocker_tests.tests.LockResult attribute*), 14
__repr__() (*portalocker_tests.tests.LockResult method*), 14
__weakref__ (*portalocker_tests.tests.LockResult attribute*), 14

A

acquire() (*portalocker.BoundedSemaphore method*), 9
acquire() (*portalocker.Lock method*), 10
acquire() (*portalocker.redis.RedisLock method*), 5
acquire() (*portalocker.RedisLock method*), 12
acquire() (*portalocker.RLock method*), 10
acquire() (*portalocker.utils.Lock method*), 7
AlreadyLocked, 7, 9

B

BaseLockException, 7
BoundedSemaphore (*class in portalocker*), 9

C

channel (*portalocker.redis.RedisLock attribute*), 5
channel (*portalocker.RedisLock attribute*), 12
channel_handler() (*portalocker.redis.RedisLock method*), 5
channel_handler() (*portalocker.RedisLock method*), 12
check_interval (*portalocker.Lock attribute*), 10
check_interval (*portalocker.redis.RedisLock attribute*), 5
check_interval (*portalocker.RLock attribute*), 11
check_interval (*portalocker.utils.Lock attribute*), 8
check_or_kill_lock() (*portalocker.redis.RedisLock method*), 5
check_or_kill_lock() (*portalocker.RedisLock method*), 12
client_name (*portalocker.redis.RedisLock property*), 5
client_name (*portalocker.RedisLock property*), 12
close_connection (*portalocker.redis.RedisLock attribute*), 5
close_connection (*portalocker.RedisLock attribute*), 12
connection (*portalocker.redis.RedisLock attribute*), 5
connection (*portalocker.RedisLock attribute*), 12

D

DEFAULT_REDIS_KWARGS (*portalocker.redis.RedisLock attribute*), 5
DEFAULT_REDIS_KWARGS (*portalocker.RedisLock attribute*), 12

E

exception_class (*portalocker_tests.tests.LockResult attribute*), 14

exception_message (*portalocker_tests.tests.LockResult attribute*), 14
exception_repr (*portalocker_tests.tests.LockResult attribute*), 15
EXCLUSIVE (*portalocker.constants.LockFlags attribute*), 6
EXCLUSIVE (*portalocker.LockFlags attribute*), 10
exclusive_lock() (*in module portalocker_tests.tests*), 15

F

fail_when_locked (*portalocker.Lock attribute*), 10
fail_when_locked (*portalocker.redis.RedisLock attribute*), 5
fail_when_locked (*portalocker.RLock attribute*), 11
fail_when_locked (*portalocker.utils.Lock attribute*), 8
fh (*portalocker.RLock attribute*), 11
fh (*portalocker.utils.Lock attribute*), 8
filename (*portalocker.RLock attribute*), 11
filename (*portalocker.utils.Lock attribute*), 8
FileToLarge, 7
flags (*portalocker.RLock attribute*), 11
flags (*portalocker.utils.Lock attribute*), 8

G

get_connection() (*portalocker.redis.RedisLock method*), 5
get_connection() (*portalocker.RedisLock method*), 12
get_filename() (*portalocker.BoundedSemaphore method*), 9
get_filenames() (*portalocker.BoundedSemaphore method*), 9
get_random_filenames() (*portalocker.BoundedSemaphore method*), 9

L

Lock (*class in portalocker*), 9
Lock (*class in portalocker.utils*), 7
lock (*portalocker.BoundedSemaphore attribute*), 9
lock() (*in module portalocker*), 12
lock() (*in module portalocker.portalocker*), 7
lock() (*in module portalocker_tests.tests*), 15
LOCK_EX (*in module portalocker*), 9
LOCK_EX (*in module portalocker.constants*), 6
LOCK_FAILED (*portalocker.exceptions.BaseLockException attribute*), 7
LOCK_NB (*in module portalocker*), 9
LOCK_NB (*in module portalocker.constants*), 6
LOCK_SH (*in module portalocker*), 9
LOCK_SH (*in module portalocker.constants*), 6
LOCK_UN (*in module portalocker*), 9
LOCK_UN (*in module portalocker.constants*), 6
LockException, 7, 10
LockFlags (*class in portalocker*), 10
LockFlags (*class in portalocker.constants*), 6
LockResult (*class in portalocker_tests.tests*), 13

M

mode (*portalocker.RLock attribute*), 11
mode (*portalocker.utils.Lock attribute*), 8
module
    portalocker, 9
    portalocker.constants, 6
    portalocker.exceptions, 7
    portalocker.portalocker, 7
    portalocker.redis, 4
    portalocker.utils, 7
    portalocker_tests.temporary_file_lock, 15
    portalocker_tests.test_combined, 15
    portalocker_tests.tests, 13

N

NON_BLOCKING (*portalocker.constants.LockFlags attribute*), 6
NON_BLOCKING (*portalocker.LockFlags attribute*), 10

O

open_atomic() (*in module portalocker*), 12
open_atomic() (*in module portalocker.utils*), 8

P

portalocker
    module, 9
portalocker.constants
    module, 6
portalocker.exceptions
    module, 7
portalocker.portalocker
    module, 7
portalocker.redis
    module, 4
portalocker.utils
    module, 7
portalocker_tests.temporary_file_lock
    module, 15
portalocker_tests.test_combined
    module, 15
portalocker_tests.tests
    module, 13
pubsub (*portalocker.redis.RedisLock attribute*), 5
pubsub (*portalocker.RedisLock attribute*), 12
PubSubWorkerThread (*class in portalocker.redis*), 4

R

redis_kwargs (*portalocker.redis.RedisLock attribute*), 5
redis_kwargs (*portalocker.RedisLock attribute*), 12
RedisLock (*class in portalocker*), 11

RedisLock (*class in portalocker.redis*), 4
release() (*portalocker.BoundedSemaphore method*), 9
release() (*portalocker.Lock method*), 10
release() (*portalocker.redis.RedisLock method*), 5
release() (*portalocker.RedisLock method*), 12
release() (*portalocker.RLock method*), 11
release() (*portalocker.utils.Lock method*), 8
RLock (*class in portalocker*), 10
run() (*portalocker.redis.PubSubWorkerThread method*), 4

S

SHARED (*portalocker.constants.LockFlags attribute*), 6
SHARED (*portalocker.LockFlags attribute*), 10
shared_lock() (*in module portalocker_tests.tests*), 15
shared_lock_fail() (*in module portalocker_tests.tests*), 15

T

test_acquire_release() (*in module portalocker_tests.tests*), 15
test_blocking_timeout() (*in module portalocker_tests.tests*), 15
test_class() (*in module portalocker_tests.tests*), 15
test_combined() (*in module portalocker_tests.test_combined*), 15
test_exceptions() (*in module portalocker_tests.tests*), 15
test_exclusive_processes() (*in module portalocker_tests.tests*), 15
test_exlusive() (*in module portalocker_tests.tests*), 15
test_lock_fileno() (*in module portalocker_tests.tests*), 15
test_nonblocking() (*in module portalocker_tests.tests*), 15
test_release_unacquired() (*in module portalocker_tests.tests*), 15
test_rlock_acquire_release() (*in module portalocker_tests.tests*), 15
test_rlock_acquire_release_count() (*in module portalocker_tests.tests*), 15
test_shared() (*in module portalocker_tests.tests*), 15
test_shared_processes() (*in module portalocker_tests.tests*), 15
test_simple() (*in module portalocker_tests.tests*), 15
test_temporary_file_lock() (*in module portalocker_tests.temporary_file_lock*), 15
test_truncate() (*in module portalocker_tests.tests*), 15
test_utils_base() (*in module portalocker_tests.tests*), 15
test_with_timeout() (*in module portalocker_tests.tests*), 15
test_without_fail() (*in module portalocker_tests.tests*), 15
test_without_timeout() (*in module portalocker_tests.tests*), 15
thread (*portalocker.redis.RedisLock attribute*), 5
thread (*portalocker.RedisLock attribute*), 12
timeout (*portalocker.Lock attribute*), 10
timeout (*portalocker.redis.RedisLock attribute*), 5
timeout (*portalocker.RedisLock attribute*), 12
timeout (*portalocker.RLock attribute*), 11
timeout (*portalocker.utils.Lock attribute*), 8
truncate (*portalocker.RLock attribute*), 11
truncate (*portalocker.utils.Lock attribute*), 8
try_lock() (*portalocker.BoundedSemaphore method*), 9

U

UNBLOCK (*portalocker.constants.LockFlags attribute*), 6
UNBLOCK (*portalocker.LockFlags attribute*), 10
unlock() (*in module portalocker*), 13
unlock() (*in module portalocker.portalocker*), 7