inotify$_simple$

Release 1.3

Jun 20, 2022

Contents

1 Installation **3**

2 Introduction **5**

3 Example usage **7**

4 Tips and tricks **9**

4.1 Gracefully exit a blocking `read()` 9

5 Module reference **11**

6 Full source code **15**

Index **19**

Chris Billington, Jun 20, 2022

- *Installation*
- *Introduction*
- *Example usage*
- *Tips and tricks*
  - *Gracefully exit a blocking* `read()`
- *Module reference*
- *Full source code*

`inotify_simple` is a simple Python wrapper around inotify. No fancy bells and whistles, just a literal wrapper with ctypes. Only ~100 lines of code!

`inotify_init1()` is wrapped as a file-like object, *`INotify`*, holding the inotify file descriptor. *`read()`* reads available data from the file descriptor and returns events as *`Event`* namedtuples after unpacking them with the `struct` module. `inotify_add_watch()` and `inotify_rm_watch()` are wrapped with no changes at all, taking and returning watch descriptor integers that calling code is expected to keep track of itself, just as one would use inotify from C. Works with Python 2.7 and Python >= 3.2.

View on PyPI | Fork me on GitHub | Read the docs

CHAPTER 1

Installation

to install `inotify_simple`, run:

```
$ pip3 install inotify_simple
```

or to install from source:

```
$ python3 setup.py install
```

Note: If on Python < 3.4, you'll need the backported enum34 module.

CHAPTER 2

Introduction

There are many inotify python wrappers out there. I found them all unsatisfactory. Most are far too high-level for my tastes, and the supposed convenience they provide actually limits one from using inotify in ways other than those the author imagined. Others are C extensions, requiring compilation for different platforms and Python versions, rather than a pure python module using ctypes. This one is pretty low-level and really just does what inotify itself does and nothing more. So hopefully if I've written it right, it will remain functional well into the future with no changes, recompilation or attention on my part.

CHAPTER 3

Example usage

```python
import os
from inotify_simple import INotify, flags

os.mkdir('/tmp/inotify_test')

inotify = INotify()
watch_flags = flags.CREATE | flags.DELETE | flags.MODIFY | flags.DELETE_SELF
wd = inotify.add_watch('/tmp/inotify_test', watch_flags)

# Now create, delete and modify some files in the directory being monitored:
os.chdir('/tmp/inotify_test')
# CREATE event for a directory:
os.system('mkdir foo')
# CREATE event for a file:
os.system('echo hello > test.txt')
# MODIFY event for the file:
os.system('echo world >> test.txt')
# DELETE event for the file
os.system('rm test.txt')
# DELETE event for the directory
os.system('rmdir foo')
os.chdir('/tmp')
# DELETE_SELF on the original directory. # Also generates an IGNORED event
# indicating the watch was removed.
os.system('rmdir inotify_test')

# And see the corresponding events:
for event in inotify.read():
    print(event)
    for flag in flags.from_mask(event.mask):
        print('    ' + str(flag))
```

```
$ python example.py
Event(wd=1, mask=1073742080, cookie=0, name='foo')
```

(continues on next page)

(continued from previous page)

```
    flags.CREATE
    flags.ISDIR
Event(wd=1, mask=256, cookie=0, name='test.txt')
    flags.CREATE
Event(wd=1, mask=2, cookie=0, name='test.txt')
    flags.MODIFY
Event(wd=1, mask=512, cookie=0, name='test.txt')
    flags.DELETE
Event(wd=1, mask=1073742336, cookie=0, name='foo')
    flags.DELETE
    flags.ISDIR
Event(wd=1, mask=1024, cookie=0, name='')
    flags.DELETE_SELF
Event(wd=1, mask=32768, cookie=0, name='')
    flags.IGNORED
```

Note that the flags, since they are defined with an `enum.IntEnum`, print as what they are called rather than their integer values. However they are still just integers and so can be bitwise-ANDed and ORed etc with masks etc. The `from_mask()` method bitwise-ANDs a mask with all possible flags and returns a list of matches. This is for convenience and useful for debugging which events are coming through, but performance critical code should generally bitwise-AND masks with flags of interest itself so as to not do unnecessary checks.

CHAPTER 4

Tips and tricks

4.1 Gracefully exit a blocking `read()`

It is common for an application to block indefinitely on a `read()` while waiting for events to be received. However it can be challenging to manage its shutdown.

When used without a timeout, the `read()` uses `poll()` on the internal file descriptor only. This call will return if there is an event on one of the file descriptors it monitors.

The idea to correctly unblock the `read()` is to add a file descriptor to be processed in addition to the internal descriptor. This new file descriptor will only be used to exit the application. Usually a `pipe()` is used for this purpose.

The *inotify_simple* module is not intended to have such a high-level mechanism. But it can be done directly in the user application by wrapping the calls to `INotify`.

Such a wrapper should:

- Have an `INotify` object.
- Have a `pipe()`.
- Run `select()` or `poll()` on both previous object descriptors.
- Retrieve the inotify events by calling `read()` with `timeout=0` (so the underlying `poll()` is not called).
- Send dummy data to the *pipe* when a stop has been requested.

An example wrapper class implementing `Thread` can be found below:

```python
import os
import select
import threading

from inotify_simple import INotify, masks, flags

class InotifyThread(threading.Thread):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, path):
        self.__path = path

        # Initialize the parent class
        threading.Thread.__init__(self)

        # Create an inotify object
        self.__inotify = INotify()

        # Create a pipe
        self.__read_fd, write_fd = os.pipe()
        self.__write = os.fdopen(write_fd, "wb")

    def run(self):
        # Watch the current directory
        self.__inotify.add_watch(self.__path, masks.ALL_EVENTS)

        while True:
            # Wait for inotify events or a write in the pipe
            rlist, _, _ = select.select(
                [self.__inotify.fileno(), self.__read_fd], [], []
            )

            # Print all inotify events
            if self.__inotify.fileno() in rlist:
                for event in self.__inotify.read(timeout=0):
                    flags = [f.name for f in flags.from_mask(event.mask)]
                    print(f"{event} {flags}")

            # Close everything properly if requested
            if self.__read_fd in rlist:
                os.close(self.__read_fd)
                self.__inotify.close()
                return

    def stop(self):
        # Request for stop by writing in the pipe
        if not self.__write.closed:
            self.__write.write(b"\x00")
            self.__write.close()
```

CHAPTER 5

Module reference

class `inotify_simple.`**`INotify`**(*inheritable=False*, *nonblocking=False*)

Bases: `_io.FileIO`

File-like object wrapping `inotify_init1()`. Raises `OSError` on failure. *`close()`* should be called when no longer needed. Can be used as a context manager to ensure it is closed, and can be used directly by functions expecting a file-like object, such as `select`, or with functions expecting a file descriptor via *`fileno()`*.

Parameters

- **`inheritable`** (*`bool`*) – whether the inotify file descriptor will be inherited by child processes. The default,‘‘False‘‘, corresponds to passing the `IN_CLOEXEC` flag to `inotify_init1()`. Setting this flag when opening filedescriptors is the default behaviour of Python standard library functions since PEP 446. On Python < 3.3, the file descriptor will be inheritable and this argument has no effect, one must instead use fcntl to set FD_CLOEXEC to make it non-inheritable.
- **`nonblocking`** (*`bool`*) – whether to open the inotify file descriptor in nonblocking mode, corresponding to passing the `IN_NONBLOCK` flag to `inotify_init1()`. This does not affect the normal behaviour of *`read()`*, which uses `poll()` to control blocking behaviour according to the given timeout, but will cause other reads of the file descriptor (for example if the application reads data manually with `os.read(fd)`) to raise `BlockingIOError` if no data is available.

`fd`

The inotify file descriptor returned by `inotify_init()`. You are free to use it directly with `os.read` if you'd prefer not to call *`read()`* for some reason. Also available as *`fileno()`*

`add_watch`(*path*, *mask*)

Wrapper around `inotify_add_watch()`. Returns the watch descriptor or raises an `OSError` on failure.

Parameters

- **`path`** (*`str, bytes, or PathLike`*) – The path to watch. Will be encoded with `os.fsencode()` before being passed to `inotify_add_watch()`.

- **`mask`** (*`int`*) – The mask of events to watch for. Can be constructed by bitwise-ORing *`flags`* together.

Returns watch descriptor

Return type int

`rm_watch`(*wd*)

Wrapper around `inotify_rm_watch()`. Raises `OSError` on failure.

Parameters **`wd`** (*`int`*) – The watch descriptor to remove

`read`(*timeout=None*, *read_delay=None*)

Read the inotify file descriptor and return the resulting *`Event`* namedtuples (wd, mask, cookie, name).

Parameters

- **`timeout`** (*`int`*) – The time in milliseconds to wait for events if there are none. If negative or `None`, block until there are events. If zero, return immediately if there are no events to be read.
- **`read_delay`** (*`int`*) – If there are no events immediately available for reading, then this is the time in milliseconds to wait after the first event arrives before reading the file descriptor. This allows further events to accumulate before reading, which allows the kernel to coalesce like events and can decrease the number of events the application needs to process. However, this also increases the risk that the event queue will overflow due to not being emptied fast enough.

Returns generator producing *`Event`* namedtuples

Return type generator

Warning: If the same inotify file descriptor is being read by multiple threads simultaneously, this method may attempt to read the file descriptor when no data is available. It may return zero events, or block until more events arrive (regardless of the requested timeout), or in the case that the *`INotify()`* object was instantiated with `nonblocking=True`, raise `BlockingIOError`.

`close`()

Close the file.

A closed file cannot be used for further I/O operations. close() may be called more than once without error.

`fileno`()

Return the underlying file descriptor (an integer).

class `inotify_simple.`**`Event`**(*wd*, *mask*, *cookie*, *name*)

Bases: `tuple`

A `namedtuple` (wd, mask, cookie, name) for an inotify event. On Python 3 the *`name`* field is a `str` decoded with `os.fsdecode()`, on Python 2 it is `bytes`.

`cookie`

Alias for field number 2

`mask`

Alias for field number 1

`name`

Alias for field number 3

`wd`

Alias for field number 0

inotify_simple.**parse_events**(*data*)

Unpack data read from an inotify file descriptor into *Event* namedtuples (wd, mask, cookie, name). This function can be used if the application has read raw data from the inotify file descriptor rather than calling *read()*.

Parameters **data** (*bytes*) – A bytestring as read from an inotify file descriptor.

Returns list of *Event* namedtuples

Return type list

class inotify_simple.**flags**

Inotify flags as defined in inotify.h but with IN_ prefix omitted. Includes a convenience method from_mask() for extracting flags from a mask.

ACCESS = 1
File was accessed

MODIFY = 2
File was modified

ATTRIB = 4
Metadata changed

CLOSE_WRITE = 8
Writable file was closed

CLOSE_NOWRITE = 16
Unwritable file closed

OPEN = 32
File was opened

MOVED_FROM = 64
File was moved from X

MOVED_TO = 128
File was moved to Y

CREATE = 256
Subfile was created

DELETE = 512
Subfile was deleted

DELETE_SELF = 1024
Self was deleted

MOVE_SELF = 2048
Self was moved

UNMOUNT = 8192
Backing fs was unmounted

Q_OVERFLOW = 16384
Event queue overflowed

IGNORED = 32768
File was ignored

ONLYDIR = 16777216
only watch the path if it is a directory

DONT_FOLLOW = 33554432
don't follow a sym link

EXCL_UNLINK = 67108864
exclude events on unlinked objects

MASK_ADD = 536870912
add to the mask of an already existing watch

ISDIR = 1073741824
event occurred against dir

ONESHOT = 2147483648
only send event once

class `inotify_simple.`**masks**
Convenience masks as defined in `inotify.h` but with `IN_` prefix omitted.

CLOSE = 24
helper event mask equal to `flags.CLOSE_WRITE | flags.CLOSE_NOWRITE`

MOVE = 192
helper event mask equal to `flags.MOVED_FROM | flags.MOVED_TO`

ALL_EVENTS = 4095
bitwise-OR of all the events that can be passed to *`add_watch()`*

CHAPTER 6

Full source code

Presented here for ease of verifying that this wrapper is as sensible as it claims to be (comments stripped - see source on github to see comments).

```
from sys import version_info, getfilesystemencoding
import os
from enum import Enum, IntEnum
from collections import namedtuple
from struct import unpack_from, calcsize
from select import poll
from time import sleep
from ctypes import CDLL, get_errno, c_int
from ctypes.util import find_library
from errno import EINTR
from termios import FIONREAD
from fcntl import ioctl
from io import FileIO

PY2 = version_info.major < 3
if PY2:
    fsencode = lambda s: s if isinstance(s, str) else s.
↪encode(getfilesystemencoding())
    IntEnum = type('IntEnum', (long, Enum), {})
else:
    from os import fsencode, fsdecode

__version__ = '1.3.5'

__all__ = ['Event', 'INotify', 'flags', 'masks', 'parse_events']

_libc = None

def _libc_call(function, *args):
    while True:
        rc = function(*args)
```

(continues on next page)

(continued from previous page)

```
        if rc != -1:
            return rc
        errno = get_errno()
        if errno != EINTR:
            raise OSError(errno, os.strerror(errno))

Event = namedtuple('Event', ['wd', 'mask', 'cookie', 'name'])

_EVENT_FMT = 'iIII'
_EVENT_SIZE = calcsize(_EVENT_FMT)

class INotify(FileIO):

    fd = property(FileIO.fileno)

    def __init__(self, inheritable=False, nonblocking=False):
        try:
            libc_so = find_library('c')
        except RuntimeError:
            libc_so = None
        global _libc; _libc = _libc or CDLL(libc_so or 'libc.so.6', use_errno=True)
        O_CLOEXEC = getattr(os, 'O_CLOEXEC', 0)
        flags = (not inheritable) * O_CLOEXEC | bool(nonblocking) * os.O_NONBLOCK
        FileIO.__init__(self, _libc_call(_libc.inotify_init1, flags), mode='rb')
        self._poller = poll()
        self._poller.register(self.fileno())

    def add_watch(self, path, mask):
        path = str(path) if hasattr(path, 'parts') else path
        return _libc_call(_libc.inotify_add_watch, self.fileno(), fsencode(path),␣
↪mask)

    def rm_watch(self, wd):
        _libc_call(_libc.inotify_rm_watch, self.fileno(), wd)

    def read(self, timeout=None, read_delay=None):
        data = self._readall()
        if not data and timeout != 0 and self._poller.poll(timeout):
            if read_delay is not None:
                sleep(read_delay / 1000.0)
            data = self._readall()
        return parse_events(data)

    def _readall(self):
        bytes_avail = c_int()
        ioctl(self, FIONREAD, bytes_avail)
        if not bytes_avail.value:
            return b''
        return os.read(self.fileno(), bytes_avail.value)

def parse_events(data):
    pos = 0
    events = []
    while pos < len(data):
        wd, mask, cookie, namesize = unpack_from(_EVENT_FMT, data, pos)
        pos += _EVENT_SIZE + namesize
        name = data[pos - namesize : pos].split(b'\x00', 1)[0]
```

(continues on next page)

(continued from previous page)

```
        events.append(Event(wd, mask, cookie, name if PY2 else fsdecode(name)))
    return events

class flags(IntEnum):
    ACCESS = 0x00000001
    MODIFY = 0x00000002
    ATTRIB = 0x00000004
    CLOSE_WRITE = 0x00000008
    CLOSE_NOWRITE = 0x00000010
    OPEN = 0x00000020
    MOVED_FROM = 0x00000040
    MOVED_TO = 0x00000080
    CREATE = 0x00000100
    DELETE = 0x00000200
    DELETE_SELF = 0x00000400
    MOVE_SELF = 0x00000800

    UNMOUNT = 0x00002000
    Q_OVERFLOW = 0x00004000
    IGNORED = 0x00008000

    ONLYDIR = 0x01000000
    DONT_FOLLOW = 0x02000000
    EXCL_UNLINK = 0x04000000
    MASK_ADD = 0x20000000
    ISDIR = 0x40000000
    ONESHOT = 0x80000000

    @classmethod
    def from_mask(cls, mask):
        return [flag for flag in cls.__members__.values() if flag & mask]

class masks(IntEnum):
    CLOSE = flags.CLOSE_WRITE | flags.CLOSE_NOWRITE
    MOVE = flags.MOVED_FROM | flags.MOVED_TO

    ALL_EVENTS  = (flags.ACCESS | flags.MODIFY | flags.ATTRIB | flags.CLOSE_WRITE |
        flags.CLOSE_NOWRITE | flags.OPEN | flags.MOVED_FROM | flags.MOVED_TO |
        flags.CREATE | flags.DELETE| flags.DELETE_SELF | flags.MOVE_SELF)
```

Index

A

ACCESS (*inotify_simple.flags attribute*), 13
add_watch() (*inotify_simple.INotify method*), 11
ALL_EVENTS (*inotify_simple.masks attribute*), 14
ATTRIB (*inotify_simple.flags attribute*), 13

C

CLOSE (*inotify_simple.masks attribute*), 14
close() (*inotify_simple.INotify method*), 12
CLOSE_NOWRITE (*inotify_simple.flags attribute*), 13
CLOSE_WRITE (*inotify_simple.flags attribute*), 13
cookie (*inotify_simple.Event attribute*), 12
CREATE (*inotify_simple.flags attribute*), 13

D

DELETE (*inotify_simple.flags attribute*), 13
DELETE_SELF (*inotify_simple.flags attribute*), 13
DONT_FOLLOW (*inotify_simple.flags attribute*), 13

E

Event (*class in inotify_simple*), 12
EXCL_UNLINK (*inotify_simple.flags attribute*), 14

F

fd (*inotify_simple.INotify attribute*), 11
fileno() (*inotify_simple.INotify method*), 12
flags (*class in inotify_simple*), 13

I

IGNORED (*inotify_simple.flags attribute*), 13
INotify (*class in inotify_simple*), 11
ISDIR (*inotify_simple.flags attribute*), 14

M

mask (*inotify_simple.Event attribute*), 12
MASK_ADD (*inotify_simple.flags attribute*), 14
masks (*class in inotify_simple*), 14
MODIFY (*inotify_simple.flags attribute*), 13
MOVE (*inotify_simple.masks attribute*), 14
MOVE_SELF (*inotify_simple.flags attribute*), 13
MOVED_FROM (*inotify_simple.flags attribute*), 13
MOVED_TO (*inotify_simple.flags attribute*), 13

N

name (*inotify_simple.Event attribute*), 12

O

ONESHOT (*inotify_simple.flags attribute*), 14
ONLYDIR (*inotify_simple.flags attribute*), 13
OPEN (*inotify_simple.flags attribute*), 13

P

parse_events() (*in module inotify_simple*), 12

Q

Q_OVERFLOW (*inotify_simple.flags attribute*), 13

R

read() (*inotify_simple.INotify method*), 12
rm_watch() (*inotify_simple.INotify method*), 12

U

UNMOUNT (*inotify_simple.flags attribute*), 13

W

wd (*inotify_simple.Event attribute*), 12