Path Locking in Python3

Andrew Peel
Scram Software

MPUG, 2 October 2017

https://scramfs.com/path-locking-in-python3/

SCRAM SOFTWARE
SECURING YOUR DATA IN THE CLOUD

Context

- Python 3.4
- PyFilesystem v1 API

Path locking developed for ScramFS, an encrypted file system that applies encryption/decryption to files and directories on an insecure wrapped file system.

The Problem

Protect file system paths being concurrently accessed by multiple threads.

```
thread1:  movedir('/a/b/c', '/x/y/z')
thread2:  removedir('/a/b', force=True)
```

The Problem

Protect file system paths being concurrently accessed by multiple threads.

```
thread1:  movedir('/a/b/c', '/x/y/z')
thread2:  removedir('/a/b', force=True)
```

- Any code with multiple calls to the wrapped file system may be interrupted by a thread context switch.

```
if not wrapped_fs.isfile('/a/b.txt'):
    raise FileNotFoundError('/a/b.txt')
fh = wrapped_fs.open('/a/b.txt')
```

Solution overview — Shared Locking

For `atomic_method('/a/b/c')`, (*e.g.* `removedir()`),

- *Shared locks* on `'/a'` and `'/a/b'` to allow concurrent access for reading, and prevent other threads from acquiring an exclusive lock.
- *Exclusive lock* on `'/a/b/c'` to create an atomic operation.

Solution overview — Shared Locking

For `atomic_method('/a/b/c')`, (*e.g.* `removedir()`),

- *Shared locks* on `'/a'` and `'/a/b'` to allow concurrent access for reading, and prevent other threads from acquiring an exclusive lock.
- *Exclusive lock* on `'/a/b/c'` to create an atomic operation.

To avoid deadlocks, always acquire locks in sequence from `'/'`, and release in the reverse order.

- Acquire: `'/'`, `'/a'`, `'/a/b'`, `'/a/b/c'`
- Release: `'/a/b/c'`, `'/a/b'`, `'/a'`, `'/'`

Solution overview — Caveats

- For multiple threads in a single process. Not for multiple processes.
- Paths are locked, not files. It does not matter whether the file or directory exists on the wrapped FS.

Existing solutions

No existing shared locking libraries matched our needs:

- Python3 `threading` package does not provide a shared lock.
- `shared_lock` package (PyPi) is from OpenStack community.
  - Incompatible with Python3.4;
  - requires multiple new dependent packages;
  - unneeded complexity to support multiple processes and hosts.
- `prwlock` package (PyPi). For multi-process locks.

Our solution — Outline

```
self.path_locker = PathLocker()

lock = PathLock(path, self.path_locker)
lock.acquire_exclusive_lock(timeout, blocking)
# Atomic code block.
lock.release_exclusive_lock()

with lock.shared_lock():
    # Read-only code block.
```

Our solution – Outline

Our solution has three classes:

- `SharedLock`
- `PathLocker` – a dictionary associating a full path (key) with a `SharedLock` object (value).
- `PathLock('/a/b/c')` – to coordinate the `SharedLock` objects for each component of a path.

```
self._locks = [<lock on '/'>,
               <lock on '/a'>,
               <lock on '/a/b'>,]
```

Our solution — SharedLock

The shared lock is just some state recording which threads have the lock, protected by a `threading.Condition`.

```
self._shared_threads[thread_id] = count
self._exclusive_thread_id
self._exclusive_thread_count
self._condition = threading.Condition()
```

- The locks are *reentrant*. Need to record how many times each thread acquires and releases.

threading.Condition

threading.Condition is a lock with additional methods wait() and notify_all().

```
# Acquire our lock
with self._condition:
    while our_lock_not_available():
        self._condition.wait()
    get_our_lock()
# Release our lock
with self._condition:
    release_our_lock()
    self._condition.notify_all()
```

- wait() releases the condition lock; blocks until notified by another thread; acquires the lock.

Our solution — SharedLock

Problem: A thread terminates while holding a lock.

Use a timeout in `wait()`. When it returns, remove locks set by threads that no longer exist.

GOTCHA: Thread IDs are recycled!
`threading.Thread.ident` is the memory address of the `Thread` object.

- Create a unique ID attribute, combining the time and `ident`, that is added to the `Thread` object when it first acquires a lock.

Our solution — SharedLock

Example from acquire_shared_lock()

```
with self._condition:
    while not (self._exclusive_thread_id is None or
            self._exclusive_thread_id == this_thread):
        result = self._condition.wait(timeout)
        if not result:
            self._clean_stale_locks()
    # Lock is available
    count = self._shared_threads.get(this_thread)
    if count is None:
        self._shared_threads[this_thread] = 1
    else:
        self._shared_threads[this_thread] = count + 1
```

Our solution — PathLocker

```
self._dict['/a/b/c'] = <SharedLock on '/a/b/c'>
```

- Use a `weakref.WeakValueDictionary`.
  Only the `PathLock` objects contain non-weak references to `SharedLock` objects. Dictionary entries that are no longer needed are garbage collected.

`PathLocker.get_lock(path)`:

- Return the `SharedLock` object for `path` from the dictionary, adding a new one if needed.
- GOTCHA: Protect this with a lock so that two threads cannot make new locks for a path at the same time.

Our solution — PathLock

`__init__(path, path_locker)`:

- Get `SharedLocks` for `path` from `path_locker` and initialize `self._locks`

```
acquire_exclusive_lock():
    <SharedLock on '/'>.acquire_shared_lock()
    <SharedLock on '/a'>.acquire_shared_lock()
   <SharedLock on '/a/b'>.acquire_exclusive_lock()
```

- When an `acquire` times out, release the locks that have already been acquired.

`acquire_shared_lock()` acquires a shared lock on `'/a/b'`.

Release in the reverse order.

Demo