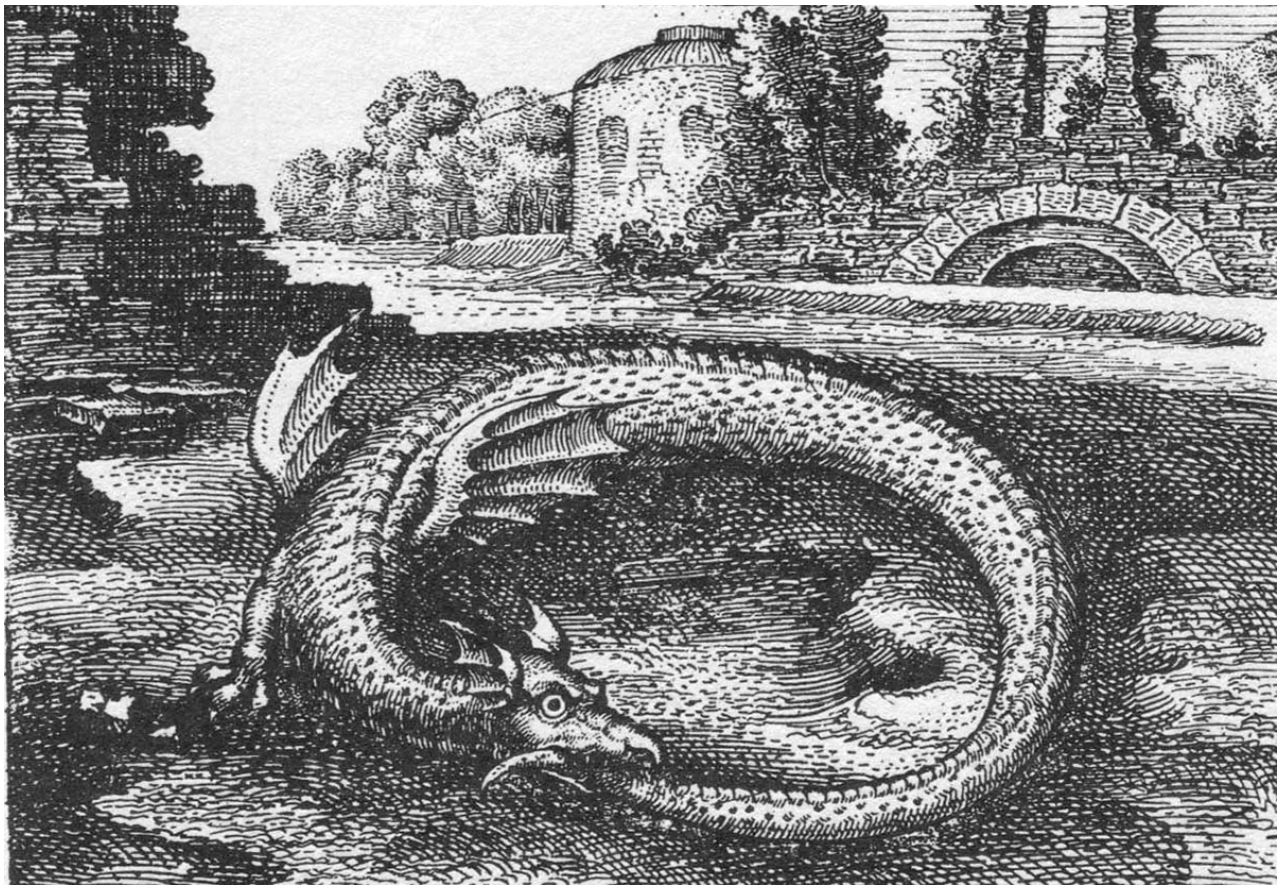
April 5, 2015

PyPy, Garbage Collection, And A Deadlock



I fixed a deadlock in PyMongo 3 and PyPy which, rarely, could happen in PyMongo 2 as well. Diagnosing the deadlock was educational and teaches us a rule about writing `__del__` methods—yet another tip about what to expect when you're expiring.

A Toy Example

This deadlocks in CPython:

```
import threading

lock = threading.Lock()

class C(object):
    def __del__(self):
        print('getting lock')
        with lock:
            print('releasing lock')
            pass

c = C()
with lock:
    del c
```

The statement `del c` removes the variable `c` from the namespace. The object that `c` had referred to has no more references, so CPython immediately calls its `__del__` method, which tries to get the lock. The lock is held, so the process deadlocks. It prints "getting lock" and hangs forever.

What if we swap the final two statements?:

```
del c
with lock:
    pass
```

This is fine. The `__del__` method completes and releases the lock before the next statement acquires it.

But consider PyPy. It doesn't use reference counts: unreferenced objects live until the garbage collector frees them. The moment when objects are freed is unpredictable. If the GC happens to kick in while the lock is held, it will deadlock. We can force this situation:

```
del c
with lock:
    gc.collect()
```

Just like the first example, this prints "getting lock" and deadlocks.

The PyMongo Bug

A few weeks ago, I found a deadlock like this in my code for the upcoming PyMongo 3.0 release. From there, I discovered a far rarer deadlock in the current release as well.

I'll give you a little context so you can see how the bug arose. With PyMongo you stream results from the MongoDB server like:

```
for document in collection.find():
    print(document)
```

The `find` method actually returns an instance of the `Cursor` class, so you could write this:

```
cursor = collection.find()
for document in cursor:
    print(document)
```

As you iterate the cursor, it returns documents from its client-side buffer until the buffer is empty, then it fetches another big batch of documents from the server. After it returns the final document of the final batch, it raises `StopIteration`.

But what if your code throws an exception before then?

```
for document in cursor:
    1 / 0  # Oops.
```

The client-side cursor goes out of scope, but the server keeps a small amount of cursor state in memory for 10 minutes. PyMongo wants to clean this up promptly, by telling the server to close the cursor as soon as the client doesn't need it. The Cursor class's destructor is in charge of telling the server:

```
class Cursor(object):
    def __del__(self):
        if self.alive:
            self._mongo_client.close_cursor(self.cursor_id)
```

In order to send the message to the server, PyMongo 3.0 has to do some work: it gets a lock on the internal Topology class so it can retrieve the connection pool, then it locks the pool so it can check out a socket. In PyPy, we do this work at a wholly unpredictable moment: it's whenever garbage collection is triggered. If any thread is holding either lock at this moment, the process deadlocks.

(Some details: By default, objects with a `__del__` method are only freed by PyPy's garbage collector during a full GC, which is triggered when memory has grown 82% since the last full GC. So if you let an open cursor go out of scope, it won't be freed for some time.)

Diagnosis

I first found this deadlock in the unreleased code for PyMongo 3.0. Our test suite was occasionally hanging under PyPy in Jenkins. When I signaled the hanging test with Control-C it printed:

```
Exception KeyboardInterrupt in method __del__
of <pymongo.cursor.Cursor object> ignored
```

The exception is "ignored" and printed to stderr, as all exceptions in `__del__` are. Once it printed the error, the test suite resumed and completed. So I added two bits of debugging info. First, whenever a cursor was created it stored a stack trace so it could remember where it came from. And second, if it caught an exception in `__del__`, it printed the stored traceback and the current traceback:

```
class Cursor(object):
    def __init__(self):
        self.tb = ''.join(traceback.format_stack())
```

```
    def __del__(self):
        try:
            self._mongo_client.close_cursor(self.cursor_id)
        except:
            print('''
I came from:%s.
I caught:%s.
''' % (self.tb, ''.join(traceback.format_stack()))
```

The next time the test hung, I hit Control-C and it printed something like:

```
I came from:
Traceback (most recent call last):
  File "test/test_cursor.py", line 431, in test_limit_and_batch_size
    curs = db.test.find().limit(0).batch_size(10)
  File "pymongo/collection.py", line 828, in find
    return Cursor(self, *args, **kwargs)
  File "pymongo/cursor.py", line 93, in __init__
    self.tb = ''.join(traceback.format_stack())

I caught:
Traceback (most recent call last):
  File "pymongo/cursor.py", line 211, in __del__
    self._mongo_client.close_cursor(self.cursor_id)
  File "pymongo/mongo_client.py", line 908, in close_cursor
    self._topology.open()
  File "pymongo/topology.py", line 58, in open
    with self._lock:
```

Great, so a test had left a cursor open, and about 30 tests *later* that cursor's destructor hung waiting for a lock. It only hung in PyPy, so I guessed it had something to do with the differences between CPython's and PyPy's garbage collection systems.

I was doing the dishes that night when my mind's background processing completed a diagnosis. As soon as I thought of it I knew I had the answer, and I wrote a test that proved it the next morning.

The Fix

PyMongo 2's concurrency design is unsophisticated and the fix was easy. I followed the code path that leads from the cursor's destructor and saw two places it could take a lock. First, if it finds that the MongoClient was recently disconnected from the server, it briefly locks it to initiate a reconnect. I updated that code path to give up immediately if the client is disconnected—better to leave the cursor open on the server for 10 minutes than to risk a deadlock.

Second, if the client is *not* disconnected, the cursor destructor locks the connection pool to check out a socket. Here, there's no easy way to avoid the lock, so I came at the problem from the other side: how do I prevent a GC while the pool is locked? If the pool is never locked at the beginning of a GC, then the cursor destructor can safely lock it. The fix is here, in `Pool.reset`:

```
class Pool:
    def reset(self):
        sockets = None
        with self.lock:
```

```
            sockets = self.sockets
            self.sockets = set()

        for s in sockets:
            s.close()
```

This is the one place we allocate data while the pool is locked. Allocating the new set while holding the lock could trigger a garbage collection, which could destroy a cursor, which could attempt to lock the pool *again*, and deadlock. So I moved the allocation outside the lock:

```
    def reset(self):
        sockets = None
        new_sockets = set()
        with self.lock:
            sockets = self.sockets
            self.sockets = new_sockets

        for s in sockets:
            s.close()
```

Now, the two lines of `reset` that run while holding the lock can't trigger a garbage collection, so the cursor destructor knows it isn't called by a GC that interrupted this section of code.

And what about PyMongo 3? The new PyMongo's concurrency design is much superior, but it spends much *more* time holding a lock than PyMongo 2 does. It locks its internal Topology class whenever it reads or updates information about your MongoDB servers. This makes the deadlock trickier to fix.

I borrowed a technique from the MongoDB Java Driver: I deferred the job of closing cursors to a background thread. Now, when an open cursor is garbage collected, it doesn't immediately tell the server. Instead, it safely adds its ID to a list. Each MongoClient has a thread that runs once a second checking the list for new cursor IDs. If there are any, the thread safely takes the locks it needs to send the message to the server—unlike the garbage collector, the cursor-cleanup thread cooperates normally with your application's threads when it needs a lock.

What To Expect When You're Expiring

I already knew that a `__del__` method:

- Must not reference globals or builtins, see my "normal accidents" article.
- Must not access threadlocals, to avoid a refleak in Python 2.6 and older (see the bug that cost me a month).

Now, add a third rule:

- It must not take a lock.

Weakref callbacks must follow these three rules, too.

The Moral Of The Story Is....

Don't use `__del__` if you can possibly avoid it. Don't design APIs that rely on it. If you maintain a library like PyMongo that has already committed to such an API, you must follow the rules above impeccably.

Image: Ouroboros, Michael Maier (1568–1622).

Categories: Mongo, Programming, Python
Tags: best

← Response to "Asynchronous Python and Databases" | Yangshan Plants His Hoe: Audio →

A. Jesse Jiryu Davis Mod ➚ Dan Crosta • a year ago

Yeah, that's what I'm going with. It's too late for PyMongo, but save yourselves! Don't commit to an API that requires __del__ to implement!

▲ | ▼ • Reply • Share ›

Yuriy Taraday • a year ago

This line is wrong:
> If any thread is holding either lock at this moment, the process deadlocks.
I've checked it: https://gist.github.com/YorikS...
Error occurs if thread that happened to be interrupted by GC holds lock, not any thread.

▲ | ▼ • Reply • Share ›

A. Jesse Jiryu Davis Mod ➚ Yuriy Taraday • a year ago

Hi Yuriy. The situation in PyMongo is something like this:

https://gist.github.com/ajdavi...

While thread A holds a lock the interpreter switches to thread B. Thread B runs, triggers a GC, and tries to get the same lock. Now the whole process is deadlocked. The linked script deadlocks in PyPy.

▲ | ▼ • Reply • Share ›

Yuriy Taraday ➚ A. Jesse Jiryu Davis • a year ago

Note that in your example target() thread has no effect since it has to release lock (and finish) before lock is taken in main thread. The script you provided doesn't deadlock on my PyPy (although it's not very fresh 2.3.1)

Update: Just realized that you used RLock there. That explains why it doesn't deadlock: GC is run in the main thread that holds the lock and so __del__ just confirms that lock is already taken and proceeds.

▲ | ▼ • Reply • Share ›

grahamd • a year ago

Django once had a similar locking issue with garbage collection related to their signals mechanism. The application would still run but the garbage collector would stop being called as it had deadlocked. As a result the memory used by the process would keep increasing as the garbage collector wasn't cleaning up anything due to not being run. We created a little bit of code which would create a background thread to monitor the number of objects tracked by the gc module, plus whether the garbage collector was even running. That way we could confirm whether the garbage collector was hanging was the issue and if the user was using an effected Django version point out the existing problem and suggest they upgrade.

▲ | ▼ • Reply • Share ›

A. Jesse Jiryu Davis Mod ➚ grahamd • a year ago

Wow, weird, how interesting.

• Reply • Share ›
