Why your multiprocessing Pool is stuck (it's full of sharks!)

https://pythonspeed.com/articles/python-multiprocessing/

Itamar Turner-Trauring Thu Jul, 29 12:32

You're using `multiprocessing` to run some code across multiple processes, and it just—sits there. It's stuck.

You check CPU usage—nothing happening, it's not doing any work.

What's going on?

In many cases you can fix this with a single line of code—skip to the end to try it out—but first, it's time for a deep-dive into Python brokenness and the pain that is POSIX system programming, using exciting and not very convincing shark-themed metaphors!

Let's set the metaphorical scene: you're swimming in a pool full of sharks. (The sharks are a metaphor for processes.)

Next, you take a fork. (The fork is a metaphor for `fork()`.)

You stab yourself with the fork. Stab stab stab. Blood starts seeping out, the sharks start circling, and pretty soon you find yourself—dead(locked) in the water!

In this journey through space and time you will encounter:

- A mysterious failure wherein Python's `multiprocessing.Pool` deadlocks, mysteriously.
- The root of the mystery: `fork()`.
- A conundrum wherein `fork()` copying everything is a problem, and `fork()` not copying everything is also a problem.
- Some bandaids that won't stop the bleeding.
- The solution that will keep your code from being eaten by sharks.

Let's begin!

Introducing `multiprocessing.Pool`

Python provides a handy module that allows you to run tasks in a pool of processes, a great way to improve the parallelism of your program. (Note that none of these examples were tested on Windows; I'm focusing on the *nix platform here.)

```
from multiprocessing import Pool
from os import getpid

def double(i):
    print('I'm process', getpid())
    return i * 2

if __name__ == '__main__':
    with Pool() as pool:
        result = pool.map(double, [1, 2, 3, 4, 5])
        print(result)
```

If we run this, we get:

```
I'm process 4942
I'm process 4943
I'm process 4944
I'm process 4942
I'm process 4943
[2, 4, 6, 8, 10]
```

As you can see, the `double()` function ran in different processes.

Some code that ought to work, but doesn't

Unfortunately, while the `Pool` class is useful, it's also full of vicious sharks, just waiting for you to make a mistake. For example, the following perfectly reasonable code:

```
import logging
from threading import Thread
from queue import Queue
from logging.handlers import QueueListener, QueueHandler
from multiprocessing import Pool

def setup_logging():
    # Logs get written to a queue, and then a thread reads
    # from that queue and writes messages to a file:
    _log_queue = Queue()
    QueueListener(
        _log_queue, logging.FileHandler('out.log')).start()
    logging.getLogger().addHandler(QueueHandler(_log_queue))

    # Our parent process is running a thread that
    # logs messages:
    def write_logs():
        while True:
            logging.error('hello, I just did something')
    Thread(target=write_logs).start()

def runs_in_subprocess():
    print('About to log...')
    logging.error('hello, I did something')
    print('...logged')

if __name__ == '__main__':
    setup_logging()

    # Meanwhile, we start a process pool that writes some
    # logs. We do this in a loop to make race condition more
    # likely to be triggered.
    while True:
        with Pool() as pool:
            pool.apply(runs_in_subprocess)
```

Here's what the program does:

1. In the parent process, log messages are routed to a queue, and a thread reads from the queue and writes those messages to a log file.
2. Another thread writes a continuous stream of log messages.
3. Finally, we start a process pool, and log a message in one of the child subprocesses.

If we run this program on Linux, we get the following output:

```
About to log...
...logged
About to log...
...logged
About to log...
<at this point the program freezes>
```

Why does this program freeze?

How subprocesses are started on POSIX (the standard formerly known as Unix)

To understand what's going on you need to understand how you start subprocesses on POSIX (which is to say, Linux, BSDs, macOS, and so on).

1. A copy of the process is created using the `fork()` system call.
2. The child process replaces itself with a different program using the `execve()` system call (or one of its variants, e.g. `execl()`).

The thing is, there's nothing preventing you from just doing `fork()`. For example, here we `fork()` and then print the current process' process ID (PID):

```
from os import fork, getpid

print('I am parent process', getpid())
if fork():
    print('I am the parent process, with PID', getpid())
else:
    print('I am the child process, with PID', getpid())
```

When we run it:

```
I am parent process 3619
I am the parent process, with PID 3619
I am the child process, with PID 3620
```

As you can see both parent (PID 3619) and child (PID 3620) continue to run the same Python code.

Here's where it gets interesting: `fork()`-only is how Python creates process pools by default on Linux, and on macOS on Python 3.7 and earlier.

The problem with just `fork()` ing

So OK, Python starts a pool of processes by just doing `fork()`. This seems convenient: the child process has access to a copy of everything in the parent process' memory (though the child can't *change* anything in the parent anymore). But how exactly is that causing the deadlock we saw?

The cause is two problems with continuing to run code after a `fork()`-without-`execve()`:

1. `fork()` copies everything in memory.
2. But it doesn't copy *everything*.

`fork()` copies everything in memory

When you do a `fork()`, it copies everything in memory. That includes any globals you've set in imported Python modules.

For example, your `logging` configuration:

```
import logging
from multiprocessing import Pool
from os import getpid

def runs_in_subprocess():
    logging.info(
        'I am the child, with PID {}'.format(getpid()))

if __name__ == '__main__':
    logging.basicConfig(
        format='GADZOOKS %(message)s', level=logging.DEBUG)

    logging.info(
        'I am the parent, with PID {}'.format(getpid()))

    with Pool() as pool:
        pool.apply(runs_in_subprocess)
```

When we run this program, we get:

```
GADZOOKS I am the parent, with PID 3884
GADZOOKS I am the child, with PID 3885
```

Notice how child processes in your pool inherit the parent process' logging configuration, even if that wasn't your intention! More broadly, *anything* you configure on a module level in the parent is inherited by processes in the pool, which can lead to some unexpected behavior.

But `fork()` doesn't copy everything

The second problem is that `fork()` doesn't actually copy everything. In particular, one thing that `fork()` *doesn't* copy is threads. Any threads running in the parent process do not exist in the child process.

```
from threading import Thread, enumerate
from os import fork
from time import sleep

# Start a thread:
Thread(target=lambda: sleep(60)).start()

if fork():
    print('The parent process has {} threads'.format(
        len(enumerate())))
else:
    print('The child process has {} threads'.format(
        len(enumerate())))
```

When we run this program, we see the thread we started didn't survive the `fork()`:

```
The parent process has 2 threads
The child process has 1 threads
```

The mystery is solved

Here's why that original program is deadlocking—with their powers combined, the two problems with `fork()`-only create a bigger, sharkier problem:

1. Whenever the thread in the parent process writes a log messages, it adds it to a `Queue`. That involves acquiring a lock.
2. If the `fork()` happens at the wrong time, the lock is copied in an acquired state.
3. The child process copies the parent's logging configuration—including the queue.
4. Whenever the child process writes a log message, it tries to write it to the queue.
5. That means acquiring the lock, but the lock is already acquired.
6. The child process now waits for the lock to be released.
7. The lock will never be released, because the thread that would release it wasn't copied over by the `fork()`.

In simplified form:

```
from os import fork
from threading import Lock

# Lock is acquired in the parent process:
lock = Lock()
lock.acquire()

if fork() == 0:
    # In the child process, try to grab the lock:
    print('Acquiring lock...')
    lock.acquire()
    print('Lock acquired! (This code will never run)')
```

Band-aids and workarounds

There are some workarounds that could make this a little better.

For module state, the `logging` library could have its configuration reset when child processes are started by `multiprocessing.Pool`. However, this doesn't solve the problem for all the *other* Python modules and libraries that set some sort of module-level global state. Every single library that does this would need to fix itself to work with `multiprocessing`.

For threads, locks could be set back to released state when `fork()` is called (Python has a ticket for this.) Unfortunately this doesn't solve the problem with locks created by C libraries, it would only address locks created directly by Python. And it doesn't address the fact that those locks don't really make sense anymore in the child process, whether or not they've been released.

Luckily, there is a better, easier solution.

The real solution: stop plain `fork()`ing

In Python 3 the `multiprocessing` library added new ways of starting subprocesses. One of these does a `fork()` followed by an `execve()` of a completely new Python process. That solves our problem, because module state isn't inherited by child processes: it starts from scratch.

Enabling this alternate configuration requires changing just two lines of code in your program before any other import or usage of `multiprocessing`; basically, the very first thing your application does should be:

```
from multiprocessing import set_start_method
set_start_method('spawn')
```

That changes things globally for all code in your program, so if you're maintaining a library the polite thing to do is use the "spawn" method just for your own pools, like this:

```
from multiprocessing import get_context

def your_func():
    with get_context('spawn').Pool() as pool:
        # ... everything else is unchanged
```

That's it: do that and all the problems we've been going over won't affect you. (See the documentation on contexts for details.)

But this still requires *you* to do the work. And it requires every Python user who trustingly follows the examples in the documentation to get confused why their program sometimes breaks.

The current default is broken, and in an ideal world Python would document that, or better yet change it to no longer be the default.

Learning more

My explanation here is of course somewhat simplified: for example, there is state other than threads that `fork()` doesn't copy. Here are some additional resources:

- Read the Linux man page for `fork()` to learn about other things it doesn't copy.
- Rachel By The Bay's post on why threads and processes don't mix and a followup are where I originally learned about this problem—and I promptly forgot about it until I encountered a related issue in production code.
- Some mathematical analysis of your chances of getting eaten by a shark and a followup.

Stay safe, and watch out for sharks and bad interactions between threads and processes! 🦈🦈

Thanks to Terry Reedy for pointing out the need for `if __name__ == '__main__'`.