Performance of subprocess.check_output vs subprocess.call

I've been using `subprocess.check_output()` for some time to capture output from subprocesses, but ran into some performance problems under certain circumstances. I'm running this on a RHEL6 machine.

The calling Python environment is linux-compiled and 64-bit. The subprocess I'm executing is a shell script which eventually fires off a Windows python.exe process via Wine (why this foolishness is required is another story). As input to the shell script, I'm piping in a small bit of Python code that gets passed off to python.exe.

While the system is under moderate/heavy load (40 to 70% CPU utilization), I've noticed that using `subprocess.check_output(cmd, shell=True)` can result in a significant delay (up to ~45 seconds) after the subprocess has finished execution before the check_output command returns. Looking at output from `ps -efH` during this time shows the called subprocess as `sh <defunct>`, until it finally returns with a normal zero exit status.

Conversely, using `subprocess.call(cmd, shell=True)` to run the same command under the same moderate/heavy load will cause the subprocess to return immediately with no delay, all output printed to STDOUT/STDERR (rather than returned from the function call).

Why is there such a significant delay only when `check_output()` is redirecting the STDOUT/STDERR output into its return value, and not when the `call()` simply prints it back to the parent's STDOUT/STDERR?

python linux subprocess wine

asked Aug 15 '14 at 20:14
greenlaw
355 1 5 15

have you tried the same code on a newer Python version or with `subprocess32` module, to see whether the unusual delay goes away i.e., there is a bug on the older version? – J.F. Sebastian Sep 6 '14 at 22:28

No I haven't, because my script requires several packages only available for 2.7.x. I have tried to reproduce the problem without my full script but have not yet been able to. If I can isolate and reproduce the problem with no library dependencies I will try your suggestion. – greenlaw Sep 8 '14 at 13:32

`subprocess32` works on Python 2.7 (posix systems) – J.F. Sebastian Sep 8 '14 at 13:34

2 Answers

Reading the docs, both `subprocess.call` and `subprocess.check_output` are use-cases of `subprocess.Popen`. One minor difference is that `check_output` will raise a Python error if the subprocess returns a non-zero exit status. The greater difference is emphasized in the bit about `check_output` (my emphasis):

> The full function signature is largely the same as that of the Popen constructor, *except that stdout is not permitted as it is used internally*. All other supplied arguments are passed directly through to the Popen constructor.

So how is `stdout` "used internally"? Let's compare `call` and `check_output`:

call

```
def call(*popenargs, **kwargs):
    return Popen(*popenargs, **kwargs).wait()
```

check_output

```
def check_output(*popenargs, **kwargs):
    if 'stdout' in kwargs:
        raise ValueError('stdout argument not allowed, it will be overridden.')
    process = Popen(stdout=PIPE, *popenargs, **kwargs)
    output, unused_err = process.communicate()
    retcode = process.poll()
    if retcode:
        cmd = kwargs.get("args")
        if cmd is None:
            cmd = popenargs[0]
        raise CalledProcessError(retcode, cmd, output=output)
    return output
```

communicate

Now we have to look at `Popen.communicate` as well. Doing this, we notice that for one pipe, `communicate` does several things which simply take more time than simply returning `Popen().wait()` , as `call` does.

For one thing, `communicate` processes `stdout=PIPE` whether you set `shell=True` or not. Clearly, `call` does not. It just lets your shell spout whatever... making it a security risk, as Python describes here.

Secondly, in the case of `check_output(cmd, shell=True)` (just one pipe)... whatever your subprocess sends to `stdout` is processed by a *thread* in the `_communicate` method. And `Popen` must join the thread (wait on it) before additionally waiting on the subprocess itself to terminate!

Plus, more trivially, it processes `stdout` as a `list` which must then be joined into a string.

In short, even with minimal arguments, `check_output` spends a lot more time in Python processes than `call` does.

answered Sep 6 '14 at 18:50
Joseph8th
283 1 9

I don't think that's a security risk; the Python documentation just warns against using shell=True when building commands from unsanitized input. But I see your point about the additional complexity of running check_output. I don't think I'm going to get a complete answer to this question without providing some exact reproduction cases, so yours is the closest. – greenlaw Sep 8 '14 at 13:46

@greenlaw: this answer doesn't explain ~45 seconds delay. Also, I suspect that threads are used only on Windows and only if more than one stream is redirected i.e., `check_output(cmd, shell=True)` does *not* use threads. – J.F. Sebastian Sep 9 '14 at 10:16

Let's look at the code. The .check_output has the following wait:

```
    def _internal_poll(self, _deadstate=None, _waitpid=os.waitpid,
            _WNOHANG=os.WNOHANG, _os_error=os.error, _ECHILD=errno.ECHILD):
        """Check if child process has terminated.  Returns returncode
        attribute.

        This method is called by __del__, so it cannot reference anything
        outside of the local scope (nor can any methods it calls).

        """
        if self.returncode is None:
            try:
                pid, sts = _waitpid(self.pid, _WNOHANG)
                if pid == self.pid:
                    self._handle_exitstatus(sts)
            except _os_error as e:
                if _deadstate is not None:
                    self.returncode = _deadstate
                if e.errno == _ECHILD:
                    # This happens if SIGCLD is set to be ignored or
                    # waiting for child processes has otherwise been
                    # disabled for our process.  This child is dead, we
                    # can't get the status.
                    # http://bugs.python.org/issue15756
                    self.returncode = 0
        return self.returncode
```

The .call waits using the following code:

```
    def wait(self):
        """Wait for child process to terminate.  Returns returncode
        attribute."""
        while self.returncode is None:
            try:
                pid, sts = _eintr_retry_call(os.waitpid, self.pid, 0)
            except OSError as e:
                if e.errno != errno.ECHILD:
                    raise
                # This happens if SIGCLD is set to be ignored or waiting
                # for child processes has otherwise been disabled for our
                # process.  This child is dead, we can't get the status.
                pid = self.pid
                sts = 0
            # Check the pid and loop as waitpid has been known to return
            # 0 even without WNOHANG in odd situations.  issue14396.
            if pid == self.pid:
                self._handle_exitstatus(sts)
        return self.returncode
```

Notice that bug related to internal_poll. It is viewable at http://bugs.python.org/issue15756. Pretty much exactly the issue you are running into.

Edit: The other potential issue between .call and .check_output is that .check_output actually cares about stdin and stdout and will try to perform IO against both pipes. If you are running into a process that get's itself into a zombie state it is possible that a read against a pipe in a defunct state is causing the hang you are experiencing.

In most cases zombie states get cleaned up pretty quickly, but, they will not if for instance they are interrupted while in a system call (like read or write). Of course the read/write system call should itself be interrupted as soon as the IO can no longer be performed, but, it is possible that you are hitting some sort of race condition where things are getting killed in a bad order.

The only way that I can think of to determine which is the cause in this case is for you to either add debugging code to the subprocess file or to invoke the python debugger and initiate a backtrace when you run into the condition you are experiencing.

edited Sep 3 '14 at 16:52   answered Sep 2 '14 at 17:35

Claris
1,552 8 17

2 Well, not exactly...the bug comments state that affected code would hang indefinitely, whereas my code does eventually return after a significant delay. – greenlaw Sep 3 '14 at 14:31

@Claris: a process is a zombie if it exits but its status has not been read yet (by its parent). In this case, `sh` is a zombie because, the parent python process hangs on `p.stdout.read()` call that may happen if `sh` spawns its own children that inherit its stdout e.g., `call('(sleep 5; echo abc) &', shell=True)` should return immediately but `check_output('(sleep 5; echo abc) &', shell=True)` should return only in 5 seconds. – J.F. Sebastian Sep 9 '14 at 10:23

@greenlaw: have you tried to set SIGALRM to look at the stacktrace if the child hangs for debugging purposes? – J.F. Sebastian Sep 9 '14 at 10:25