“You must be at least this tall”:

Threads, fork(), Deadlocks and Tests

Patrick Schemitz, solute GmbH
KA Python Meetup 2018-07-10

billiger.de

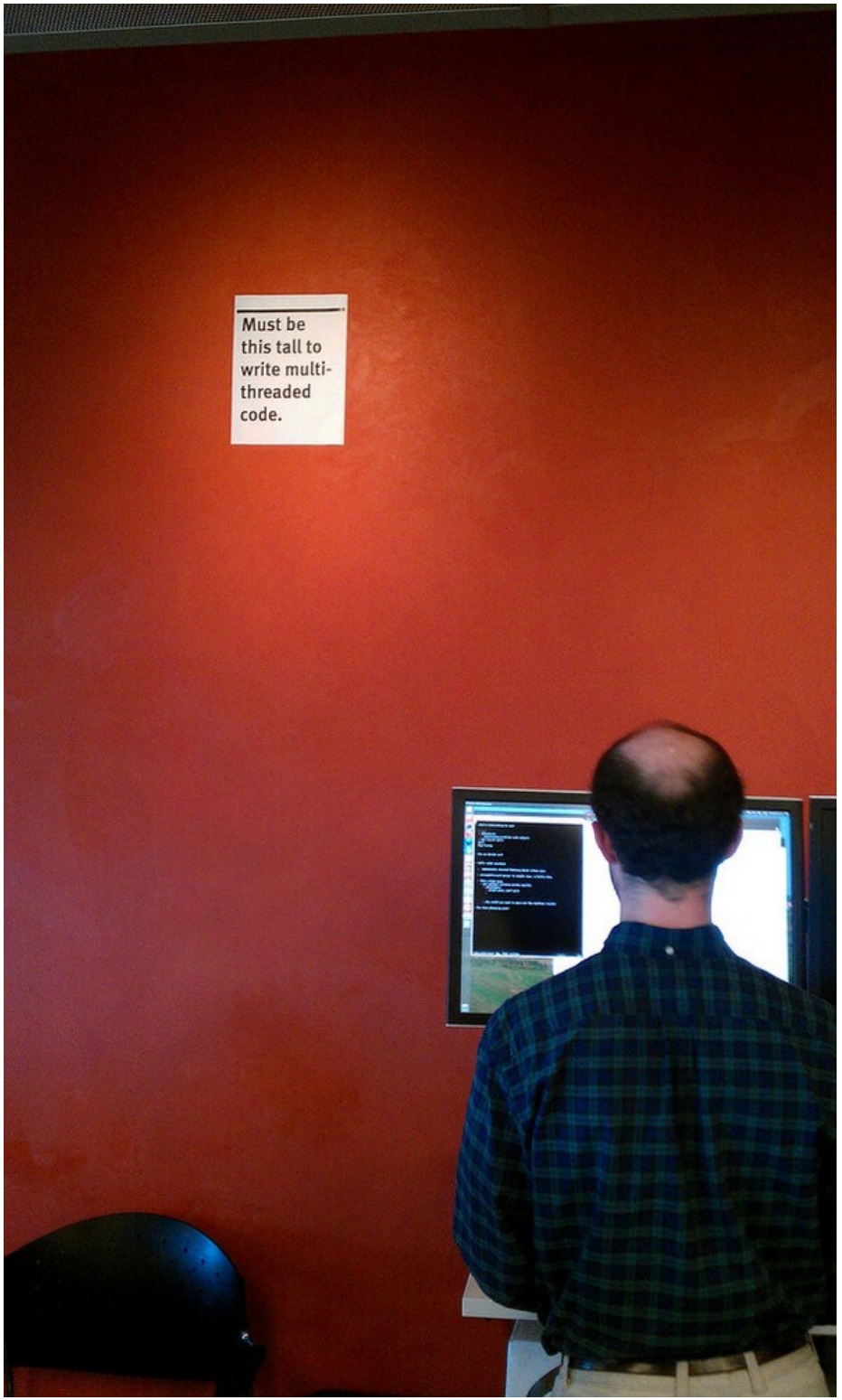


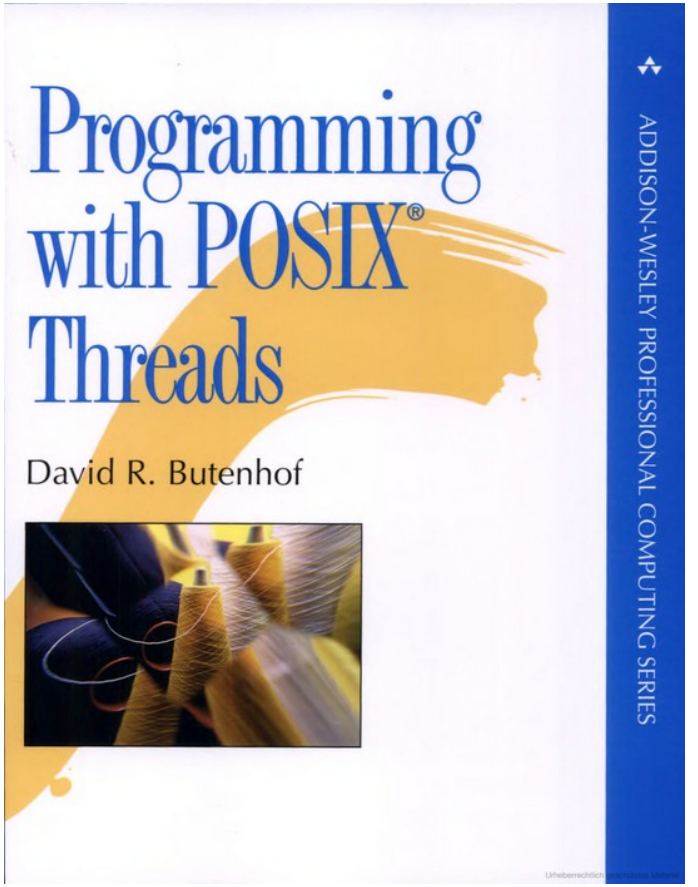
Table of Contents

- Threads and Processes
- threading and multiprocessing
- Real World Example: SOLR Index Updater
- Processes and Testing

billiger.de

Threads and Processes

- Threads implicitly share their memory (except stack), processes don't
- Threads arrived late in Unix/ POSIX (pthreads: late nineties)
- Implicit memory sharing is dangerous & difficult → lots of locking required
- Java: **`synchronized {}`** block protection
- Python: threads honour the GIL → no CPU gain



multiprocessing v. threading

- Interface of multiprocessing.Process and threading.Thread look identical:

```
def print_val(val):
    print(val)

import threading
t = threading.Thread(target=print_val, args=("Hello",))
t.start()
t.join()

import multiprocessing as mp
p = mp.Process(target=print_val, args=("Hello",))
p.start()
p.join()
```

multiprocessing v. threading

- Behaviour may differ, though:

```
def append_val(lst, val):
    lst.append(val)

l = []
t = threading.Thread(target=append_val,
                     args=(l, "Hello",))
t.start()
t.join()
print(l)

l = []
p = mp.Process(target=append_val, args=(l, "Hello",))
p.start()
p.join()
print(l)
```

Unix Interface to Processes I

- Also used in the intestines of multiprocessing
- **`fork(2)`** is a syscall that is *called once* but **returns twice**:
  - In the original process, returns new process id
  - In the new process, returns 0
- **`waitpid(2)`** is a syscall that waits for a process to terminate (blocking or non-blocking)
- What does it look like?

Unix Interface to Processes II

```
pid = os.fork()
if pid == 0:
    # child process ...
    os._exit(exitcode)  # not sys.exit()!
else:
    # parent process ...
    child_pid, exitstatus = os.waitpid(pid, 0)
    exitcode = os.WEXITSTATUS(exitstatus)
```

- New process is a copy on write (COW) memory image of original process
- Python: ref counting makes COW virtually useless
- File handles are duplicated, sockets inherited

Example – SOLR Index Updater

- Task: feed updates from spool dir into SOLRCloud cluster
- Digests linewise JSON delta file (0..1 lines per doc)
- Lots of converting → lots of CPU
- HTTP POST to SOLR already in its own thread (I/O bound so GIL is no issue)
- (Many) more updates → updater is CPU bound
- Thus, processes not threads
- Nagios bright red → we need a solution, now!

billiger.de

Index Updater Worker Processes

- New delta file arrives → we fork() worker processes
- Each worker knows its number and n_workers, i.e. knows which lines to process:

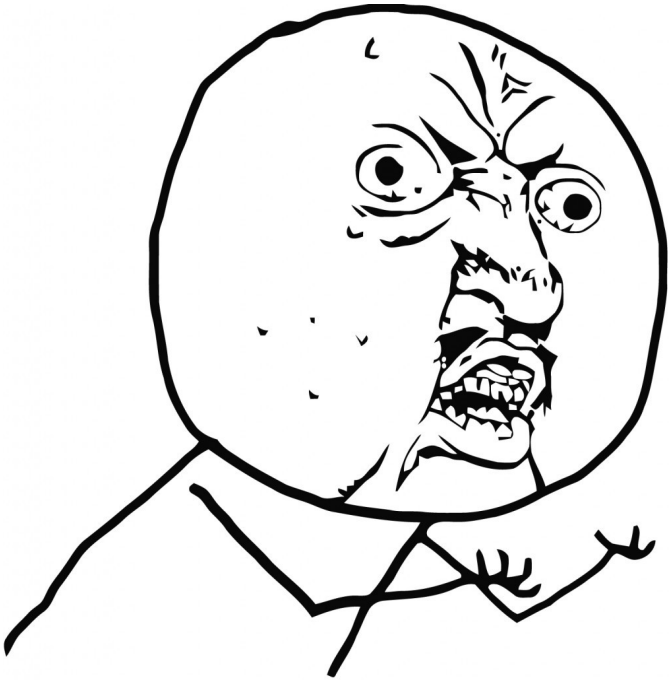
```
def updater_main(fname, n_workers, worker_num):
    log.info(“worker %i of %i”, worker_num, n_workers)
    #...
    for i, line in enumerate(open(fname)):
        if i % n_workers != worker_num:
            continue
        # actually process line
```

- Each worker gets its own POST thread

billiger.de

Index Updater Worker Processes

- Worked fine… most of the time
- Occasional deadlocks!
  Child processes would lock up,
  parent waits forever.



- Caused by...

billiger.de

Index Updater Worker Processes

- Worked fine… most of the time
- Occasional deadlocks!
  Child processes would lock up,
  parent waits forever.

- Caused by… **our monitoring thread**!

billiger.de

fork() and Threads

- Thread creation *after* fork() is fine, but…
- POSIX says: A process forked from a multithreaded process may basically just call execv()
- CERN says: “Mixing Python modules multiprocessing and threading along with logging error/debug messages with module logging is a very bad idea which leads to unexpected process stalling.”

fork() and Threads

- Python Bugtracker says: “The python logging module uses a lock to surround many operations, in particular. This causes deadlocks in programs that use logging, fork and threading simultaneously.”

- What POSIX actually says: “A process shall be created with a single thread. If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the exec functions is called. [THR]”

Processes and py.test

- py.test works well with multiprocessing:

```
def print_val(val):
    print(val)

def test_mp():
    p = multiprocessing.Process(
        target=print_val,
        args=("unittest mp",),
    )
    p.start()
    p.join()
    assert p.exitcode == 0
```

- But what about os.fork() style processes?

fork() and py.test

```
def print_val(val):
    print(val)

def test_fork():
    pid = os.fork()
    if pid == 0:
        print_val("unittest fork")
        os._exit(0)
    else:
        child_pid, exitcode = os.waitpid(pid, 0)
        assert child_pid == pid
        assert os.WEXITSTATUS(exitcode) == 0
```

- Use os._exit() instead of sys.exit() b/c SystemExit
- No coverage for child process!

Summary

- If you can, use multiprocessing instead of os.fork() and os.waitpid()
- Because multiprocessing plays nice with py.test
- No multiprocessing or fork() after thread creation
- Unless the child just os.exec()utes another program

- You're **NOT** Tall Enough.