Low-level Thread Support

sys includes low-level functions for controlling and debugging thread behavior.

Switch Interval

Python 3 uses a global lock to prevent separate threads from corrupting the interpreter state. After a configurable time interval, bytecode execution is paused and the interpreter checks if any signal handlers need to be executed. During the same check, the global interpreter lock (GIL) is also released by the current thread and then reacquired, with other threads given priority over the thread which has just released the lock.

The default switch interval is 5 milliseconds and the current value can always be retrieved with `sys.getswitchinterval()`. Changing the interval with `sys.setswitchinterval()` may have an impact on the performance of an application, depending on the nature of the operations being performed.

```python
# sys_switchinterval.py

import sys
import threading
from queue import Queue


def show_thread(q):
    for i in range(5):
        for j in range(1000000):
            pass
        q.put(threading.current_thread().name)
    return


def run_threads():
    interval = sys.getswitchinterval()
    print('interval = {:0.3f}'.format(interval))
    q = Queue()
    threads = [
        threading.Thread(target=show_thread,
                         name='T{}'.format(i),
                         args=(q,))
        for i in range(3)
    ]
    for t in threads:
        t.setDaemon(True)
        t.start()
    for t in threads:
        t.join()
    while not q.empty():
        print(q.get(), end=' ')
    print()
    return


for interval in [0.001, 0.1]:
    sys.setswitchinterval(interval)
    run_threads()
    print()
```

When the switch interval is less than the amount of time a thread takes to run to completion, the interpreter gives another thread control so that it runs for a while. This is illustrated in the first set of output where the interval is set to 1 millisecond.

For longer intervals, the active thread will be able to complete more work before it is forced to release control. This is illustrated by the order of the name values in the queue in the second example using an interval of 10 milliseconds.

```
$ python3 sys_switchinterval.py

interval = 0.001
T0 T1 T2 T1 T0 T2 T0 T1 T2 T1 T0 T2 T1 T0 T2

interval = 0.100
T0 T0 T0 T0 T0 T1 T1 T1 T1 T1 T2 T2 T2 T2 T2
```

Many factors other than the switch interval may control the context switching behavior of Python’s threads. For example, when a thread performs I/O it releases the GIL and may therefore allow another thread to take over execution.

Debugging

Identifying deadlocks can be one of the most difficult aspects of working with threads. `sys._current_frames()` can help by showing exactly where a thread is stopped.

```python
# sys_current_frames.py

import sys
import threading
import time

io_lock = threading.Lock()
blocker = threading.Lock()


def block(i):
    t = threading.current_thread()
    with io_lock:
        print('{} with ident {} going to sleep'.format(
            t.name, t.ident))
    if i:
        blocker.acquire()  # acquired but never released
        time.sleep(0.2)
    with io_lock:
        print(t.name, 'finishing')
    return


# Create and start several threads that "block"
threads = [
    threading.Thread(target=block, args=(i,))
    for i in range(3)
]
for t in threads:
    t.setDaemon(True)
    t.start()

# Map the threads from their identifier to the thread object
threads_by_ident = dict((t.ident, t) for t in threads)

# Show where each thread is "blocked"
time.sleep(0.01)
with io_lock:
    for ident, frame in sys._current_frames().items():
        t = threads_by_ident.get(ident)
        if not t:
            # Main thread
            continue
        print('{} stopped in {} at line {} of {}'.format(
            t.name, frame.f_code.co_name,
            frame.f_lineno, frame.f_code.co_filename))
```

The dictionary returned by `sys._current_frames()` is keyed on the thread identifier, rather than its name. A little work is needed to map those identifiers back to the thread object.

Since **Thread-1** does not sleep, it finishes before its status is checked. Since it is no longer active, it does not appear in the output. **Thread-2** acquires the lock `blocker`, then sleeps for a short period. Meanwhile **Thread-3** tries to acquire `blocker` but cannot because **Thread-2** already has it.

```
$ python3 sys_current_frames.py

Thread-1 with ident 123145307557888 going to sleep
Thread-1 finishing
Thread-2 with ident 123145307557888 going to sleep
Thread-3 with ident 123145312813056 going to sleep
Thread-3 stopped in block at line 18 of sys_current_frames.py
Thread-2 stopped in block at line 19 of sys_current_frames.py
```

See also

- threading – The threading module includes classes for creating Python threads.
- Queue – The Queue module provides a thread-safe implementation of a FIFO data structure.
- Reworking the GIL – Email from Antoine Pitrou to the python-dev mailing list describing the GIL implementation changes to introduce the switch interval.

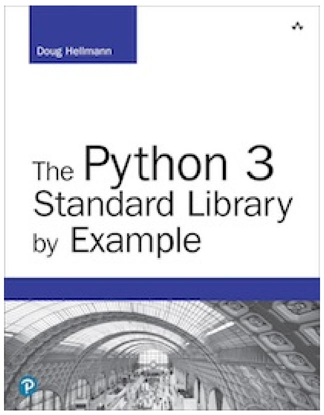
Exception Handling   Modules and Imports

Quick Links

This page was last updated 2016-12-29.

Navigation

Exception Handling
Modules and Imports



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index