Receiving Unix Signals

Unix system event notifications normally interrupt an application, triggering their handler. When used with `asyncio`, signal handler callbacks are interleaved with the other coroutines and callbacks managed by the event loop. This results in fewer interrupted functions, and the resulting need to provide safe-guards for cleaning up incomplete operations.

Signal handlers must be regular callables, not coroutines.

```
# asyncio_signal.py

import asyncio
import functools
import os
import signal


def signal_handler(name):
    print('signal_handler({!r})'.format(name))
```

The signal handlers are registered using `add_signal_handler()`. The first argument is the signal and the second is the callback. Callbacks are passed no arguments, so if arguments are needed a function can be wrapped with `functools.partial()`.

```
event_loop = asyncio.get_event_loop()

event_loop.add_signal_handler(
    signal.SIGHUP,
    functools.partial(signal_handler, name='SIGHUP'),
)
event_loop.add_signal_handler(
    signal.SIGUSR1,
    functools.partial(signal_handler, name='SIGUSR1'),
)
event_loop.add_signal_handler(
    signal.SIGINT,
    functools.partial(signal_handler, name='SIGINT'),
)
```

This example program uses a coroutine to send signals to itself via `os.kill()`. After each signal is sent, the coroutine yields control to allow the handler to be run. In a normal application, there would be more places where application code yields back to the event loop and no artificial yield like this would be needed.

```
async def send_signals():
    pid = os.getpid()
    print('starting send_signals for {}'.format(pid))

    for name in ['SIGHUP', 'SIGHUP', 'SIGUSR1', 'SIGINT']:
        print('sending {}'.format(name))
        os.kill(pid, getattr(signal, name))
        # Yield control to allow the signal handler to run,
        # since the signal does not interrupt the program
        # flow otherwise.
        print('yielding control')
        await asyncio.sleep(0.01)
    return
```

The main program runs `send_signals()` until it has sent all of the signals.

```
try:
    event_loop.run_until_complete(send_signals())
finally:
    event_loop.close()
```

The output shows how the handlers are called when `send_signals()` yields control after sending a signal.

```
$ python3 asyncio_signal.py

starting send_signals for 21772
sending SIGHUP
yielding control
signal_handler('SIGHUP')
sending SIGHUP
yielding control
signal_handler('SIGHUP')
sending SIGUSR1
yielding control
signal_handler('SIGUSR1')
sending SIGINT
yielding control
signal_handler('SIGINT')
```

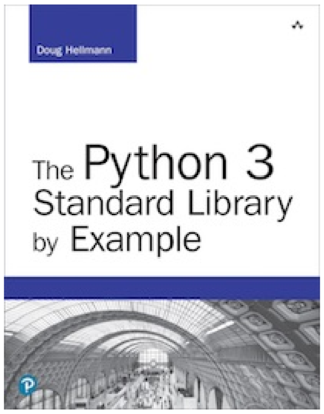
See also

- signal – Receive notification of asynchronous system events

Working with Subprocesses   Combining Coroutines with Threads and Processes

This page was last updated 2016-12-18.

Navigation
Working with Subprocesses
Combining Coroutines with Threads and Processes



Get the book

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

© Copyright 2019, Doug Hellmann