Cross-platform daemonization tools.

Release 0.1.0

Muterra, Inc

January 10, 2017

Contents

CHAPTER 1

What is Daemoniker?

Daemoniker provides a cross-platform Python API for running and signaling daemonized Python code. On Unix, it uses a standard double-fork procedure; on Windows, it creates an separate subprocess for `pythonw.exe` that exists independently of the initiating process.

Daemoniker also provides several utility tools for the resulting daemons. In particular, it includes cross-platform signaling capability for the created daemons.

1.1 Installing

Daemoniker requires **Python 3.5** or higher.

```
pip install daemoniker
```

1.2 Example usage

At the beginning of your script, invoke daemonization through the `daemoniker.Daemonizer` context manager:

```
from daemoniker import Daemonizer

with Daemonizer() as (is_setup, daemonizer):
    if is_setup:
        # This code is run before daemonization.
        do_things_here()

    # We need to explicitly pass resources to the daemon; other variables
    # may not be correct
    is_parent, my_arg1, my_arg2 = daemonizer(
        path_to_pid_file,
        my_arg1,
        my_arg2
    )

    if is_parent:
        # Run code in the parent after daemonization
        parent_only_code()

# We are now daemonized, and the parent just exited.
code_continues_here()
```

Signal handling works through the same `path_to_pid_file`:

```
from daemoniker import SignalHandler1

# Create a signal handler that uses the daemoniker default handlers for
# ``SIGINT``, ``SIGTERM``, and ``SIGABRT``
sighandler = SignalHandler1(path_to_pid_file)
sighandler.start()

# Or, define your own handlers, even after starting signal handling
def handle_sigint(signum):
    print('SIGINT received.')
sighandler.sigint = handle_sigint
```

These processes can then be sent signals from other processes:

```
from daemoniker import send
from daemoniker import SIGINT

# Send a SIGINT to a process denoted by a PID file
send(path_to_pid_file, SIGINT)
```

1.3 Comparison to `multiprocessing`, `subprocess`, etc

The modules included in the standard library for creating new processes from the current Python interpreter are intended for dependent subprocesses only. They will not continue to run if the current Python session is terminated, and when called from a Unix terminal in the background using `&`, etc, they will still result in the process being reaped upon terminal exit (this includes SSH session termination).

Daemonziation using daemoniker creates fully-independent, well-behaved processes that place no requirements on the launching terminal.

1.4 Comparison to `signal.signal`

For Unix systems, Daemoniker provides a lightweight wrapper around `signal.signal` and the poorly-named `os.kill` for the three signals (`SIGINT`, `SIGTERM`, and `SIGABRT`) that are both available on Windows and meaningful within the Python interpreter. On Unix systems, Daemoniker signal handling gives you little more than convenience.

On Windows systems, signal handling is poorly-supported at best. Furthermore, when sending signals to the independent processes created through Daemoniker, *all* signals sent to the process through `os.kill` will result in the target (daemon) process immediately exiting **without cleanup** (bypassing `try:/finally:` blocks, `atexit` calls, etc). On Windows systems, Daemoniker substantially expands this behavior, allowing Python processes to safely handle signals.

For more information on standard Windows signal handling, see:

1. Sending ^C to Python subprocess objects on Windows
2. Python send SIGINT to subprocess using os.kill as if pressing Ctrl+C
3. How to handle the signal in python on windows machine

CHAPTER 2

ToC

2.1 How it works

Daemonization is a little tricky to get right (and very difficult to test). Cross-platform "daemonization" is even less straightforward.

2.1.1 Daemonization: Unix

On Unix, daemonization with Daemoniker performs the following steps:

1. Create a PID file, failing if it already exists
2. Register cleanup of the PID file for program exit
3. Double-fork, dissociating itself from the original process group and any possible control terminal
4. Reset `umask` and change current working directory
5. Write its PID to the PID file
6. Close file descriptors
7. Redirect `stdin`, `stdout`, and `stderr`.

Note: To be considered a "well-behaved daemon", applications should also, at the least, handle termination through a `SIGTERM` handler (see Signal handling API for using Daemoniker for this purpose).

2.1.2 Daemonization: Windows

On Windows, "daemonization" with Daemoniker performs the following steps:

1. Find the currently-running Python interpreter and file.
2. Search for a copy of `pythonw.exe` within the same directory.
3. Create a PID file, failing if it already exists.
4. Save the current namespace and re-launch the script using `pythonw.exe`.
5. Bypass any already-completed code using an environment variable "switch".
6. Change current working directory.

7. Write its process handle to the PID file and register the file's cleanup for program exit.
8. Redirect `stdin`, `stdout`, and `stderr`.
9. Extract the old namespace.
10. Return the old namespace into the resumed "daemonized" process and allow the original process to exit.

Warning: Due to the implementation of signals on Windows (as well as their use within the CPython interpreter), any signals sent to this daughter process will result in its **immediate termination, without any cleanup.** That means no `atexit` calls, no `finally:` blocks, etc. See *Signals: Windows* below for more information, or see Signal handling API for using Daemoniker as a workaround.

2.1.3 Signals: Unix

Signal handling on Unix is very straightforward. The signal handler provided by `SigHandler1` provides a thin wrapper around the built-in `signal.signal` functionality. To maintain uniform cross-platform behavior, the `frame` argument typically passed to `signal.signal` callbacks is removed, but otherwise, Daemoniker is simply a convenience wrapper around `signal.signal` that includes several default signal handlers.

2.1.4 Signals: Windows

Signals on Windows are not natively supported by the operating system. They are included in the C runtime environment provided by Windows, but their role is substantially different than that in Unix systems. Furthermore, these signals are largely limited to transmission between parent/child processes, and because the "daemonization" process creates a fully-independent process group, every available signal (including the Windows-specific `CTRL_C_EVENT` and `CTRL_BREAK_EVENT`) result in immediate termination of the daughter process without cleanup.

To avoid this thoroughly undesirable behavior, Daemoniker uses the following workaround:

1. From the main thread of the (daemonized) Python script, launch a daughter **thread** devoted to signal handling.
2. From that daughter thread, launch a sleep-loop-forever daughter **process**.
3. Overwrite the PID file with the PID of the daughter process.
4. Wait for the daughter process to complete. If it was killed by a signal, its return code equals the number of the signal. Handle it accordingly.
5. For every signal received, create a new daughter process.

Additionally, to mimic the behavior of `signal.signal` and replicate Unix behavior, the default Daemoniker signal handlers call a `ctypes` API to raise an exception **in the main thread** of the parent script.

2.2 Daemonization API

Simple daemonization may be performed by directly calling the *`daemonize()`* funtion. In general, it should be the first thing called by your code (except perhaps `import` statements, global declarations, and so forth). If you need to do anything more complicated, use the *`Daemonizer`* context manager.

`daemonize`(*pid_file*, **args*, *chdir=None*, *stdin_goto=None*, *stdout_goto=None*, *stderr_goto=None*, *umask=0o027*, *shielded_fds=None*, *fd_fallback_limit=1024*, *success_timeout=30*, *strip_cmd_args=False*)

New in version 0.1.

The function used to actually perform daemonization. It may be called directly, but is intended to be used within the *Daemonizer* context manager.

> **Warning:** When used directly, all code prior to `daemonize()` will be repeated by the daemonized process on Windows systems. It is best to limit all pre-`daemonize` code to import statements, global declarations, etc. If you want to run specialized setup code, use the *Daemonizer* context manager.

Note: All `*args` must be pickleable on Windows systems.

Parameters

- **pid_file** (*str*) – The path to use for the PID file.
- ***args** – All variables to preserve across the daemonization boundary. On Windows, only these values (which will be returned by `daemonize`) are guaranteed to be persistent.
- **chdir** (*str*) – The path to use for the working directory after daemonizing. Defaults to the current directory, which can result in "directory busy" errors on both Unix and Windows systems. **This argument is keyword-only.**
- **stdin_goto** (*str*) – A filepath to redirect `stdin` into. A value of `None` defaults to `os.devnull`. **This argument is keyword-only.**
- **stdout_goto** (*str*) – A filepath to redirect `stdout` into. A value of `None` defaults to `os.devnull`. **This argument is keyword-only.**
- **stderr_goto** (*str*) – A filepath to redirect `stderr` into. A value of `None` defaults to `os.devnull`. **This argument is keyword-only.**
- **umask** (*int*) – The file creation mask to apply to the daemonized process. Unused on Windows. **This argument is keyword-only.**
- **shielded_fds** – An iterable of integer file descriptors to shield from closure. Unused on Windows. **This argument is keyword-only.**
- **fd_ballback_limit** (*int*) – If the file descriptor `resource` hard limit and soft limit are both infinite, this fallback integer will be one greater than the highest file descriptor closed. Unused on Windows. **This argument is keyword-only.**
- **success_timeout** – A numeric limit, in seconds, for how long the parent process should wait for acknowledgment of successful startup by the daughter process. Unused on Unix. **This argument is keyword-only.**
- **strip_cmd_args** (*bool*) – If the current script was started from a prompt using arguments, as in `python script.py --arg1 --arg2`, this value determines whether or not those arguments should be stripped when re-invoking the script. In this example, calling `daemonize` with `strip_cmd_args=True` would be re-invoke the script as `python script.py`. Unused on Unix. **This argument is keyword-only.**

Returns `*args`

```
>>> from daemoniker import daemonize
>>> daemonize('pid.pid')
```

class Daemonizer

New in version 0.1.

A context manager for more advanced daemonization. Entering the context assigns a tuple of (boolean `is_setup`, callable `daemonizer`) to the `with Daemonizer() as target:` target.

```
from daemoniker import Daemonizer

with Daemonizer() as (is_setup, daemonizer):
    if is_setup:
        # This code is run before daemonization.
        do_things_here()

    # We need to explicitly pass resources to the daemon; other variables
    # may not be correct
    is_parent, my_arg1, my_arg2 = daemonizer(
        'path/to/pid/file.pid',
        my_arg1,
        my_arg2,
        ...,
        **daemonize_kwargs
    )

    # This allows us to run parent-only post-daemonization code
    if is_parent:
        run_some_parent_code_here()

# We are now daemonized and the parent has exited.
code_continues_here(my_arg1, my_arg2)
```

When used in this manner, the *Daemonizer* context manager will return a boolean `is_setup` and a wrapped *daemonize()* function.

Note: Do not include an `else` clause after `is_setup`. It will not be run on Unix:

```
from daemoniker import Daemonizer

with Daemonizer() as (is_setup, daemonizer):
    if is_setup:
        # This code is run before daemonization.
        do_things_here()
    else:
        # This code will never run on Unix systems.
        do_no_things_here()

    ...
```

Note: To prevent resource contention with the daemonized child, the parent process must be terminated via `os._exit` when exiting the context. You must perform any cleanup inside the `if is_parent:` block.

__enter__()

Entering the context will return a tuple of:

```
with Daemonizer() as (is_setup, daemonizer):
```

`is_setup` is a `bool` that will be True when code is running in the parent (pre-daemonization) process.

`daemonizer` wraps *daemonize()*, prepending a bool `is_parent` to its return value. To prevent accidental manipulation of already-passed variables from the parent process, it also replaces them with

`None` in the parent caller. It is otherwise identical to *`daemonize()`*. For example:

```
from daemoniker import Daemonizer

with Daemonizer() as (is_setup, daemonizer):
    if is_setup:
        my_arg1 = 'foo'
        my_arg2 = 'bar'

    is_parent, my_arg1, my_arg2 = daemonizer(
        'path/to/pid/file.pid',
        my_arg1,
        my_arg2
    )

    # This code will only be run in the parent process
    if is_parent:
        # These will return True
        my_arg1 == None
        my_arg2 == None

    # This code will only be run in the daemonized child process
    else:
        # These will return True
        my_arg1 == 'foo'
        my_arg2 == 'bar'

# The parent has now exited. All following code will only be run in
# the daemonized child process.
program_continues_here(my_arg1, my_arg2)
```

`__exit__`()
Exiting the context will do nothing in the child. In the parent, leaving the context will initiate a forced termination via `os._exit` to prevent resource contention with the daemonized child.

2.3 Signal handling API

class `SignalHandler1`(*pid_file*, *sigint=None*, *sigterm=None*, *sigabrt=None*)
New in version 0.1.

Handles signals for the daemonized process.

Parameters

- **`pid_file`** (*`str`*) – The path to the PID file.
- **`sigint`** – A callable handler for the `SIGINT` signal. May also be `daemoniker.IGNORE_SIGNAL` to explicitly ignore the signal. Passing the default value of `None` will assign the default `SIGINT` handler, which will simply `raise daemoniker.SIGINT` **within the main thread.**
- **`sigterm`** – A callable handler for the `SIGTERM` signal. May also be `daemoniker.IGNORE_SIGNAL` to explicitly ignore the signal. Passing the default value of `None` will assign the default `SIGTERM` handler, which will simply `raise daemoniker.SIGTERM` **within the main thread.**
- **`sigabrt`** – A callable handler for the `SIGABRT` signal. May also be `daemoniker.IGNORE_SIGNAL` to explicitly ignore the signal. Passing the de-

fault value of `None` will assign the default `SIGABRT` handler, which will simply `raise daemoniker.SIGABRT` **within the main thread.**

Warning: There is a slight difference in handler calling between Windows and Unix systems. In every case, the default handler will always `raise` from within **the main thread.** However, if you define a custom signal handler, on Windows systems it will be called from within a daughter thread devoted to signal handling. This has two consequences:

1. All signal handlers must be threadsafe
2. On Windows, future signals will be silently dropped until the callback completes

On Unix systems, the handler will be called from within the main thread.

Note: On Windows, `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` signals are both converted to `SIGINT` signals internally. This also applies to custom signal handlers.

```
>>> from daemoniker import SignalHandler1
>>> sighandler = SignalHandler1('pid.pid')
```

sigint
The current handler for `SIGINT` signals. This must be a callable. It will be invoked with a single argument: the signal number. It may be re-assigned, even after calling *start()*. Deleting it will restore the default `Daemoniker` signal handler; **to ignore it, instead assign** `daemoniker.IGNORE_SIGNAL` **as the handler.**

sigterm
The current handler for `SIGTERM` signals. This must be a callable. It will be invoked with a single argument: the signal number. It may be re-assigned, even after calling *start()*. Deleting it will restore the default `Daemoniker` signal handler; **to ignore it, instead assign** `daemoniker.IGNORE_SIGNAL` **as the handler.**

sigabrt
The current handler for `SIGABRT` signals. This must be a callable. It will be invoked with a single argument: the signal number. It may be re-assigned, even after calling *start()*. Deleting it will restore the default `Daemoniker` signal handler; **to ignore it, instead assign** `daemoniker.IGNORE_SIGNAL` **as the handler.**

start()
Starts signal handling. Must be called to receive signals with the `SignalHandler`.

stop()
Stops signal handling. Must be called to stop receive signals. `stop` will be automatically called:

1. at the interpreter exit, and
2. when the main thread exits.

`stop` is idempotent. On Unix systems, it will also restore the previous signal handlers.

IGNORE_SIGNAL
A constant used to explicitly declare that a *SignalHandler1* should ignore a particular signal.

send(*pid_file*, *signal*)
New in version 0.1.

Send a `signal` to the process at `pid_file`.

Parameters

- **pid_file** (*str*) – The path to the PID file.

- **signal** – The signal to send. This may be either:
  1. an *instance* of one of the *ReceivedSignal* exceptions, for example: `daemoniker.SIGINT()` (see *SIGINT*)
  2. the *class* for one of the *ReceivedSignal* exceptions, for example: `daemoniker.SIGINT` (see *SIGINT*)
  3. an integer-like value, corresponding to the signal number, for example: `signal.SIGINT`

```
>>> from daemoniker import send
>>> from daemoniker import SIGINT
>>> send('pid.pid', SIGINT)
```

2.4 Exception API

exception DaemonikerException

All of the custom exceptions in `Daemoniker` subclass this exception. As such, an application can catch any `Daemoniker` exception via:

```
try:
    code_goes_here()

except DaemonikerException:
    handle_error_here()
```

exception SignalError

These errors are only raised if something goes wrong internally while handling signals.

exception ReceivedSignal

Subclasses of `ReceivedSignal` exceptions are raised by the default signal handlers. A `ReceivedSignal` will only be raise directly:

1.if the actual signal number passed to the callback does not match its expected value.

2.if, on Windows, the signal handling daughter process terminates abnormally.

Note: Calling `int()` on a `ReceivedSignal` class or instance, or a class or instance of any of its subclasses, will return the signal number associated with the signal.

---

exception SIGINT

Raised for incoming `SIGINT` signals. May also be used to *send()* signals to other processes.

Attr SIGNUM The signal number associated with the signal.

exception SIGTERM

Raised for incoming `SIGTERM` signals. May also be used to *send()* signals to other processes.

Attr SIGNUM The signal number associated with the signal.

exception SIGABRT

Raised for incoming `SIGABRT` signals. May also be used to *send()* signals to other processes.

Attr SIGNUM The signal number associated with the signal.

2.4.1 Exception hierarchy

The `Daemoniker` exceptions have the following inheritance:

```
DaemonikerException
    SignalError
    ReceivedSignal
        SIGINT
        SIGTERM
        SIGABRT
```

Symbols

D

I

R

S