cysignals Documentation

Release 1.11.2a0.dev0

Martin Albrecht, Jeroen Demeyer

Dec 01, 2021

CONTENTS

This is the documentation for **cysignals, a package to deal with interrupts and signal handling in Cython code**.

When writing Cython code, special care must be taken to ensure that the code can be interrupted with CTRL-C. Since Cython optimizes for speed, Cython normally does not check for interrupts. For example, code like the following cannot be interrupted in Cython:

```
while True:
    pass
```

While this is running, pressing CTRL-C has no effect. The only way out is to kill the Python process. On certain systems, you can still quit Python by typing CTRL-\ (sending a Quit signal) instead of CTRL-C. The package cysignals provides functionality to deal with this, see *Interrupt handling*.

Besides this, cysignals also provides Python functions/classes to deal with signals. These are not directly related to interrupts in Cython, but provide some supporting functionality beyond what Python provides, see *Signal-related interfaces for Python code*.

CHAPTER

ONE

INTERRUPT/SIGNAL HANDLING

Dealing with interrupts and other signals using `sig_check` and `sig_on`:

1.1 Interrupt handling

cysignals provides two related mechanisms to deal with interrupts:

- Use *sig_check()* if you are writing mixed Cython/Python code. Typically this is code with (nested) loops where every individual statement takes little time.
- Use *sig_on() and sig_off()* if you are calling external C libraries or inside pure Cython code (without any Python functions) where even an individual statement, like a library call, can take a long time.

The functions `sig_check()`, `sig_on()` and `sig_off()` can be put in all kinds of Cython functions: `def`, `cdef` or `cpdef`. You cannot put them in pure Python code (files with extension `.py`).

1.1.1 Basic example

The `sig_check()` in the loop below ensures that the loop can be interrupted by `CTRL-C`:

```
from cysignals.signals cimport sig_check
from libc.math cimport sin

def sine_sum(double x, long count):
    cdef double s = 0
    for i in range(count):
        sig_check()
        s += sin(i*x)
    return s
```

See the example directory for this complete working example.

Note: Cython `cdef` or `cpdef` functions with a return type (like `cdef int myfunc():`) need to have an except value to propagate exceptions. Remember this whenever you write `sig_check()` or `sig_on()` inside such a function, otherwise you will see a message like `Exception KeyboardInterrupt: KeyboardInterrupt() in <function name> ignored`.

1.1.2 Using `sig_check()`

`sig_check()` can be used to check for pending interrupts. If an interrupt happens during the execution of C or Cython code, it will be caught by the next `sig_check()`, the next `sig_on()` or possibly the next Python statement. With the latter we mean that certain Python statements also check for interrupts, an example of this is the `print` statement. The following loop *can* be interrupted:

```
>>> while True:
...     print("Hello")
```

The typical use case for `sig_check()` is within tight loops doing complicated stuff (mixed Python and Cython code, potentially raising exceptions). It is reasonably safe to use and gives a lot of control, because in your Cython code, a `KeyboardInterrupt` can *only* be raised during `sig_check()`:

```
from cysignals.signals cimport sig_check
def sig_check_example():
    for x in foo:
        # (one loop iteration which does not take a long time)
        sig_check()
```

This `KeyboardInterrupt` is treated like any other Python exception and can be handled as usual:

```
from cysignals.signals cimport sig_check
def catch_interrupts():
    try:
        while some_condition():
            sig_check()
            do_something()
    except KeyboardInterrupt:
        # (handle interrupt)
```

Of course, you can also put the `try/except` inside the loop in the example above.

The function `sig_check()` is an extremely fast inline function which should have no measurable effect on performance.

1.1.3 Using `sig_on()` and `sig_off()`

Another mechanism for interrupt handling is the pair of functions `sig_on()` and `sig_off()`. It is more powerful than `sig_check()` but also a lot more dangerous. You should put `sig_on()` *before* and `sig_off()` *after* any Cython code which could potentially take a long time. These two *must always* be called in **pairs**, i.e. every `sig_on()` must be matched by a closing `sig_off()`.

In practice your function will probably look like:

```
from cysignals.signals cimport sig_on, sig_off
def sig_example():
    # (some harmless initialization)
    sig_on()
    # (a long computation here, potentially calling a C library)
    sig_off()
    # (some harmless post-processing)
    return something
```

It is possible to put `sig_on()` and `sig_off()` in different functions, provided that `sig_off()` is called before the function which calls `sig_on()` returns. The reason is that `sig_on()` is implemented using `setjmp()`, which requires that the stack frame is kept alive. Therefore, the following code is *invalid*:

```
# INVALID code because we return from function foo()
# without calling sig_off() first.
cdef foo():
    sig_on()

def f1():
    foo()
    sig_off()
```

But the following is valid since you cannot call `foo` interactively:

```
from cysignals.signals cimport sig_on, sig_off

cdef int foo():
    sig_off()
    return 2+2

def f1():
    sig_on()
    return foo()
```

For clarity however, it is best to avoid this.

A common mistake is to put `sig_off()` towards the end of a function (before the `return`) when the function has multiple `return` statements. So make sure there is a `sig_off()` before *every* `return` (and also before every `raise`).

Warning: The code inside `sig_on()` should be pure C or Cython code. If you call any Python code or manipulate any Python object (even something trivial like `x = []`), an interrupt can mess up Python's internal state. When in doubt, try to use *sig_check()* instead.

Also, when an interrupt occurs inside `sig_on()`, code execution immediately stops without cleaning up. For example, any memory allocated inside `sig_on()` is lost. See *Signal handling without exceptions* for ways to deal with this.

When the user presses `CTRL-C` inside `sig_on()`, execution will jump back to `sig_on()` (the first one if there is a stack) and `sig_on()` will raise `KeyboardInterrupt`. As with `sig_check()`, this exception can be handled in the usual way:

```
from cysignals.signals cimport sig_on, sig_off
def catch_interrupts():
    try:
        sig_on()  # This must be INSIDE the try
        # (some long computation)
        sig_off()
    except KeyboardInterrupt:
        # (handle interrupt)
```

It is possible to stack `sig_on()` and `sig_off()`. If you do this, the effect is exactly the same as if only the outer `sig_on()`/`sig_off()` was there. The inner ones will just change a reference counter and otherwise do nothing. Make sure that the number of `sig_on()` calls equal the number of `sig_off()` calls:

```
from cysignals.signals cimport sig_on, sig_off

def f1():
    sig_on()
    x = f2()
    sig_off()

cdef f2():
    sig_on()
    # ...
    sig_off()
    return ans
```

Extra care must be taken with exceptions raised inside `sig_on()`. The problem is that, if you do not do anything special, the `sig_off()` will never be called if there is an exception. If you need to *raise* an exception yourself, call a `sig_off()` before it:

```
from cysignals.signals cimport sig_on, sig_off
def raising_an_exception():
    sig_on()
    # (some long computation)
    if (something_failed):
        sig_off()
        raise RuntimeError("something failed")
    # (some more computation)
    sig_off()
    return something
```

Alternatively, you can use `try`/`finally` which will also catch exceptions raised by subroutines inside the `try`:

```
from cysignals.signals cimport sig_on, sig_off
def try_finally_example():
    sig_on()  # This must be OUTSIDE the try
    try:
        # (some long computation, potentially raising exceptions)
        return something
    finally:
        sig_off()
```

If you also want to catch this exception, you need a nested `try`:

```
from cysignals.signals cimport sig_on, sig_off
def try_finally_and_catch_example():
    try:
        sig_on()
        try:
            # (some long computation, potentially raising exceptions)
        finally:
            sig_off()
    except Exception:
        print("Trouble! Trouble!")
```

`sig_on()` is implemented using the C library call `setjmp()` which takes a very small but still measurable amount of time. In very time-critical code, one can conditionally call `sig_on()` and `sig_off()`:

```
from cysignals.signals cimport sig_on, sig_off
def conditional_sig_on_example(long n):
    if n > 100:
        sig_on()
    # (do something depending on n)
    if n > 100:
        sig_off()
```

This should only be needed if both the check (`n > 100` in the example) and the code inside the `sig_on()` block take very little time.

1.2 Handling other signals

Apart from handling interrupts, *sig_on()* provides more general signal handling. For example, it handles `alarm()` time-outs by raising an `AlarmInterrupt` (inherited from `KeyboardInterrupt`) exception.

If the code inside `sig_on()` would generate a segmentation fault or call the C function `abort()` (or more generally, raise any of SIGSEGV, SIGILL, SIGABRT, SIGFPE, SIGBUS), this is caught by the interrupt framework and an exception is raised (`RuntimeError` for SIGABRT, `FloatingPointError` for SIGFPE and the custom exception `SignalError`, based on `BaseException`, otherwise):

```
from libc.stdlib cimport abort
from cysignals.signals cimport sig_on, sig_off

def abort_example():
    sig_on()
    abort()
    sig_off()
```

```
>>> abort_example()
Traceback (most recent call last):
...
RuntimeError: Aborted
```

This exception can be handled by a `try/except` block as explained above. A segmentation fault or `abort()` unguarded by `sig_on()` would simply terminate the Python Interpreter. This applies only to `sig_on()`, the function `sig_check()` only deals with interrupts and alarms.

Instead of `sig_on()`, there is also a function `sig_str(s)`, which takes a C string `s` as argument. It behaves the same as `sig_on()`, except that the string `s` will be used as a string for the exception. `sig_str(s)` should still be closed by `sig_off()`. Example Cython code:

```
from libc.stdlib cimport abort
from cysignals.signals cimport sig_str, sig_off

def abort_example_with_sig_str():
    sig_str("custom error message")
    abort()
    sig_off()
```

Executing this gives:

```
>>> abort_example_with_sig_str()
Traceback (most recent call last):
...
RuntimeError: custom error message
```

With regard to ordinary interrupts (i.e. SIGINT), `sig_str(s)` behaves the same as `sig_on()`: a simple `KeyboardInterrupt` is raised.

1.3 Further topics in interrupt/signal handling

1.3.1 Testing interrupts

When writing documentation, one sometimes wants to check that certain code can be interrupted in a clean way. The best way to do this is to use `cysignals.alarm()`.

The following is an example of a doctest demonstrating that the SageMath function `factor()` can be interrupted:

```
>>> from cysignals.alarm import alarm, AlarmInterrupt
>>> try:
...     alarm(0.5)
...     factor(10**1000 + 3)
... except AlarmInterrupt:
...     print("alarm!")
alarm!
```

If you use the SageMath doctesting framework, you can instead doctest the exception in the usual way (the Python `doctest` module exits whenever a `KeyboardInterrupt` is raised in a doctest). To avoid race conditions, make sure that the calls to `alarm()` and the function you want to test are in the same doctest:

```
>>> alarm(0.5); factor(10**1000 + 3)
Traceback (most recent call last):
...
AlarmInterrupt
```

1.3.2 Signal handling without exceptions

There are several more specialized functions for dealing with interrupts. As mentioned above, `sig_on()` makes no attempt to clean anything up (restore state or freeing memory) when an interrupt occurs. In fact, it would be impossible for `sig_on()` to do that. If you want to add some cleanup code, use `sig_on_no_except()` for this. This function behaves *exactly* like `sig_on()`, except that any exception raised (like `KeyboardInterrupt` or `RuntimeError`) is not yet passed to Python. Essentially, the exception is there, but we prevent Cython from looking for the exception. Then `cython_check_exception()` can be used to make Cython look for the exception.

Normally, `sig_on_no_except()` returns 1. If a signal was caught and an exception raised, `sig_on_no_except()` instead returns 0. The following example shows how to use `sig_on_no_except()`:

```
def no_except_example():
    if not sig_on_no_except():
        # (clean up messed up internal state)

        # Make Cython realize that there is an exception.
```

(continues on next page)

(continued from previous page)

```
        # It will look like the exception was actually raised
        # by cython_check_exception().
        cython_check_exception()
    # (some long computation, messing up internal state of objects)
    sig_off()
```

There is also a function `sig_str_no_except(s)` which is analogous to `sig_str(s)`.

Note: See the file src/cysignals/tests.pyx for more examples of how to use the various `sig_*()` functions.

1.3.3 Releasing the Global Interpreter Lock (GIL)

All the functions related to interrupt and signal handling do not require the Python GIL (if you don't know what this means, you can safely ignore this section), they are declared `nogil`. This means that they can be used in Cython code inside `with nogil` blocks. If `sig_on()` needs to raise an exception, the GIL is temporarily acquired internally.

If you use C libraries without the GIL and you want to raise an exception before calling *sig_error()*, remember to acquire the GIL while raising the exception. Within Cython, you can use a with gil context.

Warning: The GIL should never be released or acquired inside a `sig_on()` block. If you want to use a `with nogil` block, put both `sig_on()` and `sig_off()` inside that block. When in doubt, choose to use `sig_check()` instead, which is always safe to use.

1.4 Debugging Python crashes

If `cysignals` is imported, it sets up a hook which triggers when Python crashes. For example, it would be triggered on a segmentation fault outside a `sig_on()` block.

When a crash happens, first a simple C backtrace is printed if supported by the C library on the system. Then GDB is run to print a much more complete backtrace (except on OS X, where running a debugger requires special privileges). For your convenience, these GDB backtraces are also saved to a logfile.

Finally, this familiar message is shown:

```
This probably occurred because a *compiled* module has a bug
in it and is not properly wrapped with sig_on(), sig_off().
Python will now terminate.
```

1.4.1 Environment variables

There are several environment variables which influence this:

CYSIGNALS_CRASH_QUIET
: If set, be completely quiet whenever a crash happens. No backtrace or other message is shown and GDB is not run.

CYSIGNALS_CRASH_NDEBUG
: If set, disable the GDB backtrace. The simple backtrace is still shown.

`CYSIGNALS_CRASH_LOGS`
: The directory where the logs of the crashes are stored. If this is empty, disable storing of crash logs. The default is `cysignals_crash_logs` in the current directory.

`CYSIGNALS_CRASH_DAYS`
: Automatically delete crash logs older than this many days in the directory where crash logs are stored. A negative value means that logs are never deleted. The default is 7 days if `CYSIGNALS_CRASH_LOGS` is unset and -1 days (never delete) otherwise.

CHAPTER

TWO

ERROR HANDLING

Defining error callbacks for external libraries using `sig_error`:

2.1 Error handling in C libraries

Some C libraries can produce errors and use some sort of callback mechanism to report errors: an external error handling function needs to be set up which will be called by the C library if an error occurs.

The function `sig_error()` can be used to deal with these errors. This function may only be called within a `sig_on()` block (otherwise the Python interpreter will crash hard) after raising a Python exception. You need to use the Python/C API for this and call `sig_error()` after calling some variant of `PyErr_SetObject()`. Even within Cython, you cannot use the `raise` statement, because then the `sig_error()` will never be executed. The call to `sig_error()` will use the `sig_on()` machinery such that the exception will be seen by `sig_on()`.

A typical error handler implemented in Cython would look as follows:

```
from cysignals.signals cimport sig_error
from cpython.exc cimport PyErr_SetString

cdef void error_handler(char *msg):
    PyErr_SetString(RuntimeError, msg)
    sig_error()
```

Exceptions which are raised this way can be handled as usual by putting the `sig_on()` in a `try/except` block. For example, the package cypari2 provides a wrapper around the number theory library PARI/GP. The error handler has a callback which turns errors from PARI/GP into Python exceptions of type `PariError`. This can be handled as follows:

```
from cysignals.signals cimport sig_on, sig_off
def handle_pari_error():
    try:
        sig_on()  # This must be INSIDE the try
        # (call to PARI)
        sig_off()
    except PariError:
        # (handle error)
```

SageMath uses this mechanism for libGAP, GLPK, NTL and PARI.

CHAPTER

THREE

SIGNAL-RELATED INTERFACES FOR PYTHON CODE

cysignals provides further support for system calls related to signals:

3.1 Python interface to signal handlers

In this module, we distinguish between the "OS-level" signal handler and the "Python-level" signal handler.

The Python function `signal.signal()` sets both of these: it sets the Python-level signal handler to the function specified by the user. It also sets the OS-level signal handler to a specific C function which calls the Python-level signal handler.

The Python `signal` module does not allow access to the OS-level signal handler (in particular, it does not allow one to temporarily change a signal handler if the OS-level handler was not the Python one).

class cysignals.pysignals.**SigAction**

An opaque object representing an OS-level signal handler.

The only legal initializers are `signal.SIG_DFL` (the default), `signal.SIG_IGN` and another `SigAction` object (which is copied).

EXAMPLES:

```
>>> from cysignals.pysignals import SigAction
>>> SigAction()
<SigAction with sa_handler=SIG_DFL>
>>> import signal
>>> SigAction(signal.SIG_DFL)
<SigAction with sa_handler=SIG_DFL>
>>> SigAction(signal.SIG_IGN)
<SigAction with sa_handler=SIG_IGN>
>>> A = SigAction(signal.SIG_IGN)
>>> SigAction(A)
<SigAction with sa_handler=SIG_IGN>
>>> SigAction(A) == A
True
```

TESTS:

```
>>> SigAction(42)
Traceback (most recent call last):
...
TypeError: cannot initialize SigAction from <... 'int'>
```

class cysignals.pysignals.**changesignal**

Context to temporarily change a signal handler.

This should be used as follows:

```
with changesignal(sig, action):
    ...
```

Inside the context, code behaves as if `signal.signal(sig, action)` was called. When leaving the context, the signal handler is restored to what it was before. Both the Python-level and OS-level signal handlers are restored.

EXAMPLES:

```
>>> from cysignals.pysignals import changesignal
>>> import os, signal
>>> def handler(*args):
...     print("got signal")
>>> _ = signal.signal(signal.SIGQUIT, signal.SIG_IGN)
>>> with changesignal(signal.SIGQUIT, handler):
...     os.kill(os.getpid(), signal.SIGQUIT)
got signal
>>> os.kill(os.getpid(), signal.SIGQUIT)
>>> with changesignal(signal.SIGQUIT, handler):
...     setossignal(signal.SIGQUIT, signal.SIG_DFL)
...     raise Exception("just testing")
Traceback (most recent call last):
...
Exception: just testing
>>> os.kill(os.getpid(), signal.SIGQUIT)
```

class cysignals.pysignals.**containsignals**

Context to revert any changes to given signal handlers and block those signals.

This should be used as follows:

```
with containsignals(signals):
    ...
```

where `signals` is a list of signals (by default, all signals numbered from 1 to 31 except for `SIGKILL` and `SIGSTOP`, which cannot be handled).

When entering the context, the current handlers of those signals are saved. They are restored when exiting the context. This is mainly meant to prevent unwanted changes to signal handlers that other code may make. Both the Python-level and OS-level signal handlers are saved and restored.

Also, the signals from the list `signals` are blocked. So any newly-installed signal handlers are prevented from being triggered.

EXAMPLES:

```
>>> from cysignals.pysignals import containsignals
>>> import os, signal
>>> def handler(*args):
...     print("got signal")
>>> _ = signal.signal(signal.SIGBUS, handler)
>>> with containsignals([signal.SIGBUS]):
```

(continues on next page)

(continued from previous page)

```
...     _ = signal.signal(signal.SIGBUS, signal.SIG_DFL)
...     # This signal is delivered when exiting the context
...     os.kill(os.getpid(), signal.SIGBUS)
...     print("no signal yet")
no signal yet
got signal
```

The same example but now containing all signals:

```
>>> with containsignals() as C:
...     print("blocked {0} signals".format(len(C.oldhandlers)))
...     _ = signal.signal(signal.SIGBUS, signal.SIG_DFL)
...     # This signal is delivered when exiting the context
...     os.kill(os.getpid(), signal.SIGBUS)
...     print("no signal yet")
blocked 29 signals
no signal yet
got signal
```

This time, we send a signal which is not contained. We set a new handler, which is not blocked or changed by the context:

```
>>> def fancyhandler(*args):
...     print("fancy!")
>>> with containsignals([signal.SIGINT]):
...     _ = signal.signal(signal.SIGBUS, fancyhandler)
...     os.kill(os.getpid(), signal.SIGBUS)
fancy!
>>> os.kill(os.getpid(), signal.SIGBUS)
fancy!
```

cysignals.pysignals.**getossignal**(*sig*)

Get the OS-level signal handler.

This returns an opaque object of type *SigAction* which can only be used in a future call to *setossignal()*.

EXAMPLES:

```
>>> from cysignals.pysignals import getossignal
>>> import signal
>>> getossignal(signal.SIGINT)
<SigAction with sa_handler=0x...>
>>> getossignal(signal.SIGUSR1)
<SigAction with sa_handler=SIG_DFL>
>>> def handler(*args): pass
>>> _ = signal.signal(signal.SIGUSR1, handler)
>>> getossignal(signal.SIGUSR1)
<SigAction with sa_handler=0x...>
```

Check whether a signal is handled by the Python signal handler:

```
>>> from cysignals.pysignals import python_os_handler
>>> getossignal(signal.SIGUSR1) == python_os_handler
True
```

(continues on next page)

(continued from previous page)

```
>>> _ = signal.signal(signal.SIGUSR1, signal.SIG_IGN)
>>> getossignal(signal.SIGUSR1) == python_os_handler
False
>>> getossignal(signal.SIGABRT) == python_os_handler
False
```

TESTS:

```
>>> getossignal(None)
Traceback (most recent call last):
...
TypeError: an integer is required
>>> getossignal(-1)
Traceback (most recent call last):
...
OSError: [Errno 22] Invalid argument
```

cysignals.pysignals.**setossignal**(*sig*, *action*)

Set the OS-level signal handler to `action`, which should either be `signal.SIG_DFL` or `signal.SIG_IGN` or a *SigAction* object returned by an earlier call to *getossignal()* or *setossignal()*.

Return the old signal handler.

EXAMPLES:

```
>>> from cysignals.pysignals import setossignal
>>> import os, signal
>>> def handler(*args): print("got signal")
>>> _ = signal.signal(signal.SIGHUP, handler)
>>> os.kill(os.getpid(), signal.SIGHUP)
got signal
>>> pyhandler = setossignal(signal.SIGHUP, signal.SIG_IGN)
>>> pyhandler
<SigAction with sa_handler=0x...>
>>> os.kill(os.getpid(), signal.SIGHUP)
>>> setossignal(signal.SIGHUP, pyhandler)
<SigAction with sa_handler=SIG_IGN>
>>> os.kill(os.getpid(), signal.SIGHUP)
got signal
>>> setossignal(signal.SIGHUP, signal.SIG_DFL) == pyhandler
True
```

TESTS:

```
>>> setossignal(signal.SIGHUP, None)
Traceback (most recent call last):
...
TypeError: cannot initialize SigAction from <... 'NoneType'>
>>> setossignal(-1, signal.SIG_DFL)
Traceback (most recent call last):
...
OSError: [Errno 22] Invalid argument
```

cysignals.pysignals.**setsignal**(*sig*, *action*, *osaction=None*)

Set the Python-level signal handler for signal `sig` to `action`. If `osaction` is given, set the OS-level signal handler to `osaction`. If `osaction` is `None` (the default), change only the Python-level handler and keep the OS-level handler.

Return the old Python-level handler.

EXAMPLES:

```
>>> from cysignals.pysignals import *
>>> def handler(*args): print("got signal")
>>> _ = signal.signal(signal.SIGSEGV, handler)
>>> A = getossignal(signal.SIGILL)
>>> _ = setsignal(signal.SIGILL, getsignal(signal.SIGSEGV))
>>> getossignal(signal.SIGILL) == A
True
>>> _ = setossignal(signal.SIGILL, getossignal(signal.SIGSEGV))
>>> import os
>>> os.kill(os.getpid(), signal.SIGILL)
got signal
>>> setsignal(signal.SIGILL, signal.SIG_DFL)
<function handler at 0x...>
>>> _ = setsignal(signal.SIGALRM, signal.SIG_DFL, signal.SIG_IGN)
>>> os.kill(os.getpid(), signal.SIGALRM)
>>> _ = setsignal(signal.SIGALRM, handler, getossignal(signal.SIGSEGV))
>>> os.kill(os.getpid(), signal.SIGALRM)
got signal
```

TESTS:

```
>>> setsignal(-1, signal.SIG_DFL)
Traceback (most recent call last):
...
OSError: [Errno 22] Invalid argument
```

3.2 Interface to the `pselect()` and `sigprocmask()` system calls

This module defines a class *PSelecter* which can be used to call the system call `pselect()` and which can also be used in a `with` statement to block given signals until *PSelecter.pselect()* is called.

3.2.1 Waiting for subprocesses

One possible use is to wait with a **timeout** until **any child process** exits, as opposed to `os.wait()` which doesn't have a timeout or `multiprocessing.Process.join()` which waits for one specific process.

Since `SIGCHLD` is ignored by default, we first need to install a signal handler for `SIGCHLD`. It doesn't matter what it does, as long as the signal isn't ignored:

```
>>> import signal
>>> def dummy_handler(sig, frame):
...     pass
>>> _ = signal.signal(signal.SIGCHLD, dummy_handler)
```

We wait for a child created using the `subprocess` module:

```
>>> from cysignals.pselect import PSelecter
>>> from subprocess import *
>>> with PSelecter([signal.SIGCHLD]) as sel:
...     p = Popen(["sleep", "1"])
...     _ = sel.sleep()
>>> p.poll()  # p should be finished
0
```

Now using the `multiprocessing` module:

```
>>> from cysignals.pselect import PSelecter
>>> from multiprocessing import *
>>> import time
>>> with PSelecter([signal.SIGCHLD]) as sel:
...     p = Process(target=time.sleep, args=(1,))
...     p.start()
...     _ = sel.sleep()
...     p.is_alive()  # p should be finished
False
```

class cysignals.pselect.**PSelecter**

This class gives an interface to the `pselect` system call.

It can be used in a `with` statement to block given signals such that they can only occur during the *pselect()* or *sleep()* calls.

As an example, we block the `SIGHUP` and `SIGALRM` signals and then raise a `SIGALRM` signal. The interrupt will only be seen during the *sleep()* call:

```
>>> from cysignals import AlarmInterrupt
>>> from cysignals.pselect import PSelecter
>>> import os, signal, time
>>> with PSelecter([signal.SIGHUP, signal.SIGALRM]) as sel:
...     os.kill(os.getpid(), signal.SIGALRM)
...     time.sleep(0.5)  # Simply sleep, no interrupt detected
...     try:
...         _ = sel.sleep(1)  # Interrupt seen here
...     except AlarmInterrupt:
...         print("Interrupt OK")
Interrupt OK
```

Warning: If `SIGCHLD` is blocked inside the `with` block, then you should not use `Popen().wait()` or `Process().join()` because those might block, even if the process has actually exited. Use non-blocking alternatives such as `Popen.poll()` or `multiprocessing.active_children()` instead.

__enter__()

Block signals chosen during `__init__()` in this `with` block.

OUTPUT: `self`

TESTS:

Test nesting, where the inner `with` statements should have no influence, in particular they should not unblock signals which were already blocked upon entering:

```
>>> from cysignals import AlarmInterrupt
>>> from cysignals.pselect import PSelecter
>>> import os, signal
>>> with PSelecter([signal.SIGALRM]) as sel:
...     os.kill(os.getpid(), signal.SIGALRM)
...     with PSelecter([signal.SIGFPE]) as sel2:
...         _ = sel2.sleep(0.1)
...     with PSelecter([signal.SIGALRM]) as sel3:
...         _ = sel3.sleep(0.1)
...     try:
...         _ = sel.sleep(0.1)
...     except AlarmInterrupt:
...         print("Interrupt OK")
Interrupt OK
```

__exit__(**args*)

Reset signal mask to what it was before *__enter__()*.

EXAMPLES:

Install a `SIGCHLD` handler:

```
>>> import signal
>>> def child_handler(sig, frame):
...     global got_child
...     got_child = 1
>>> _ = signal.signal(signal.SIGCHLD, child_handler)
>>> got_child = 0
```

Start a process which will cause a `SIGCHLD` signal:

```
>>> import time
>>> from multiprocessing import *
>>> from cysignals.pselect import PSelecter, interruptible_sleep
>>> w = PSelecter([signal.SIGCHLD])
>>> with w:
...     p = Process(target=time.sleep, args=(0.25,))
...     t0 = time.time()
...     p.start()
```

This `sleep` should be interruptible now:

```
>>> interruptible_sleep(1)
>>> t = time.time() - t0
>>> (0.2 <= t <= 0.9) or t
True
>>> got_child
1
>>> p.join()
```

pselect(*rlist*, *wlist*, *xlist*, *timeout*)

Wait until one of the given files is ready, or a signal has been received, or until `timeout` seconds have past.

INPUT:

- `rlist` – (default: `[]`) a list of files to wait for reading.
- `wlist` – (default: `[]`) a list of files to wait for writing.
- `xlist` – (default: `[]`) a list of files to wait for exceptions.
- `timeout` – (default: `None`) a timeout in seconds, where `None` stands for no timeout.

OUTPUT: A 4-tuple `(rready, wready, xready, tmout)` where the first three are lists of file descriptors which are ready, that is a subset of `(rlist, wlist, xlist)`. The fourth is a boolean which is `True` if and only if the command timed out. If `pselect` was interrupted by a signal, the output is `([], [], [], False)`.

See also:

Use the *sleep()* method instead if you don't care about file descriptors.

EXAMPLES:

The file `/dev/null` should always be available for reading and writing:

```
>>> from cysignals.pselect import PSelecter
>>> f = open(os.devnull, "r+")
>>> sel = PSelecter()
>>> sel.pselect(rlist=[f])
([<...'/dev/null'...>], [], [], False)
>>> sel.pselect(wlist=[f])
([], [<...'/dev/null'...>], [], False)
```

A list of various files, all of them should be ready for reading. Also create a pipe, which should be ready for writing, but not reading (since nothing has been written):

```
>>> import os, sys
>>> f = open(os.devnull, "r")
>>> g = open(sys.executable, "r")
>>> (pr, pw) = os.pipe()
>>> r, w, x, t = PSelecter().pselect([f,g,pr,pw], [pw], [pr,pw])
>>> len(r), len(w), len(x), t
(2, 1, 0, False)
```

Checking for exceptions on the pipe should simply time out:

```
>>> sel.pselect(xlist=[pr,pw], timeout=0.2)
([], [], [], True)
```

TESTS:

It is legal (but silly) to list the same file multiple times:

```
>>> r, w, x, t = PSelecter().pselect([f,g,f,f,g])
>>> len(r)
5
```

Invalid input:

```
>>> PSelecter().pselect([None])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: an integer is required
```

Open a file and close it, but save the (invalid) file descriptor:

```
>>> f = open(os.devnull, "r")
>>> n = f.fileno()
>>> f.close()
>>> PSelecter().pselect([n])
Traceback (most recent call last):
...
OSError: ...
```

sleep(*timeout*)

Wait until a signal has been received, or until `timeout` seconds have past.

This is implemented as a special case of *pselect()* with empty lists of file descriptors.

INPUT:

- `timeout` – (default: `None`) a timeout in seconds, where `None` stands for no timeout.

OUTPUT: A boolean which is `True` if the call timed out, False if it was interrupted.

EXAMPLES:

A simple wait with timeout:

```
>>> from cysignals.pselect import PSelecter
>>> sel = PSelecter()
>>> sel.sleep(timeout=0.1)
True
```

0 or negative time-outs are allowed, `sleep` should then return immediately:

```
>>> sel.sleep(timeout=0)
True
>>> sel.sleep(timeout=-123.45)
True
```

cysignals.pselect.**get_fileno**(*f*)

Return the file descriptor of `f`.

INPUT:

- `f` – an object with a `.fileno` method or an integer, which is a file descriptor.

OUTPUT: A C `long` representing the file descriptor.

EXAMPLES:

```
>>> from os import devnull
>>> from cysignals.pselect import get_fileno
>>> get_fileno(open(devnull)) > 2
True
>>> get_fileno(42)
42
>>> get_fileno(None)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: an integer is required
>>> get_fileno(-1)
Traceback (most recent call last):
...
ValueError: Invalid file descriptor
>>> get_fileno(2**30)
Traceback (most recent call last):
...
ValueError: Invalid file descriptor
```

cysignals.pselect.**interruptible_sleep**(*seconds*)

Sleep for `seconds` seconds or until a signal arrives. This behaves like `time.sleep` from Python versions <= 3.4 (before **PEP 475**).

EXAMPLES:

```
>>> from cysignals.pselect import interruptible_sleep
>>> interruptible_sleep(0.5)
```

We set up an alarm handler doing nothing and check that the alarm interrupts the sleep:

```
>>> import signal, time
>>> def alarm_handler(sig, frame):
...     pass
>>> _ = signal.signal(signal.SIGALRM, alarm_handler)
>>> t0 = time.time()
>>> _ = signal.alarm(1)
>>> interruptible_sleep(2)
>>> t = time.time() - t0
>>> (0.9 <= t <= 1.9) or t
True
```

TESTS:

```
>>> interruptible_sleep(0)
>>> interruptible_sleep(-1)
Traceback (most recent call last):
...
ValueError: sleep length must be non-negative
```

Reset the signal handlers:

```
>>> from cysignals import init_cysignals
>>> _ = init_cysignals()
```

CHAPTER

FOUR

LINKS

- cysignals on the Python package index: https://pypi.org/project/cysignals/
- cysignals code repository and issue tracker on GitHub: https://github.com/sagemath/cysignals
- cysignals documentation on Read the Docs: https://cysignals.readthedocs.io

PYTHON MODULE INDEX

C

INDEX

Symbols

C

E

G

I

M

P

S