18.5.6. Subprocess

Source code: Lib/asyncio/subprocess.py

18.5.6.1. Windows event loop

On Windows, the default event loop is `SelectorEventLoop` which does not support subprocesses. `ProactorEventLoop` should be used instead. Example to use it on Windows:

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

See also: Available event loops and Platform support.

18.5.6.2. Create a subprocess: high-level API using Process

coroutine `asyncio.`**`create_subprocess_exec`**(**args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Create a subprocess.

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `AbstractEventLoop.subprocess_exec()` for other parameters.

Return a `Process` instance.

This function is a coroutine.

coroutine `asyncio.`**`create_subprocess_shell`**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Run the shell command *cmd*.

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `AbstractEventLoop.subprocess_shell()` for other parameters.

Return a `Process` instance.

It is the application's responsibility to ensure that all whitespace and meta-characters are quoted appropriately to avoid shell injection vulnerabilities. The

`shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

This function is a coroutine.

Use the `AbstractEventLoop.connect_read_pipe()` and `AbstractEventLoop.connect_write_pipe()` methods to connect pipes.

18.5.6.3. Create a subprocess: low-level API using subprocess.Popen

Run subprocesses asynchronously using the `subprocess` module.

coroutine `AbstractEventLoop.`**`subprocess_exec`**(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from one or more string arguments (character strings or bytes strings encoded to the filesystem encoding), where the first string specifies the program to execute, and the remaining strings specify the program's arguments. (Thus, together the string arguments form the `sys.argv` value of the program, assuming it is a Python script.) This is similar to the standard library `subprocess.Popen` class called with shell=False and the list of strings passed as the first argument; however, where `Popen` takes a single argument which is list of strings, `subprocess_exec()` takes multiple string arguments.

The *protocol_factory* must instantiate a subclass of the `asyncio.SubprocessProtocol` class.

Other parameters:

- *stdin*: Either a file-like object representing the pipe to be connected to the subprocess's standard input stream using `connect_write_pipe()`, or the constant `subprocess.PIPE` (the default). By default a new pipe will be created and connected.
- *stdout*: Either a file-like object representing the pipe to be connected to the subprocess's standard output stream using `connect_read_pipe()`, or the constant `subprocess.PIPE` (the default). By default a new pipe will be created and connected.
- *stderr*: Either a file-like object representing the pipe to be connected to the subprocess's standard error stream using `connect_read_pipe()`, or one of the constants `subprocess.PIPE` (the default) or `subprocess.STDOUT`. By default a new pipe will be created and connected. When

`subprocess.STDOUT` is specified, the subprocess's standard error stream will be connected to the same pipe as the standard output stream.

- All other keyword arguments are passed to `subprocess.Popen` without interpretation, except for *bufsize*, *universal_newlines* and *shell*, which should not be specified at all.

Returns a pair of `(transport, protocol)`, where *transport* is an instance of `BaseSubprocessTransport`.

This method is a coroutine.

See the constructor of the `subprocess.Popen` class for parameters.

coroutine `AbstractEventLoop.`**`subprocess_shell`**(*protocol_factory*, *cmd*, *, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from *cmd*, which is a character string or a bytes string encoded to the filesystem encoding, using the platform's "shell" syntax. This is similar to the standard library `subprocess.Popen` class called with `shell=True`.

The *protocol_factory* must instantiate a subclass of the `asyncio.SubprocessProtocol` class.

See `subprocess_exec()` for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* is an instance of `BaseSubprocessTransport`.

It is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

This method is a coroutine.

See also: The `AbstractEventLoop.connect_read_pipe()` and `AbstractEventLoop.connect_write_pipe()` methods.

18.5.6.4. Constants

`asyncio.subprocess.`**`PIPE`**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `create_subprocess_shell()` and `create_subprocess_exec()` and indicates that a pipe to the standard stream should be opened.

asyncio.subprocess.**STDOUT**

Special value that can be used as the *stderr* argument to `create_subprocess_shell()` and `create_subprocess_exec()` and indicates that standard error should go into the same handle as standard output.

asyncio.subprocess.**DEVNULL**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `create_subprocess_shell()` and `create_subprocess_exec()` and indicates that the special file `os.devnull` will be used.

18.5.6.5. Process

class asyncio.subprocess.**Process**

A subprocess created by the `create_subprocess_exec()` or the `create_subprocess_shell()` function.

The API of the `Process` class was designed to be close to the API of the `subprocess.Popen` class, but there are some differences:

- There is no explicit `poll()` method
- The `communicate()` and `wait()` methods don't take a *timeout* parameter: use the `wait_for()` function
- The *universal_newlines* parameter is not supported (only bytes strings are supported)
- The `wait()` method of the `Process` class is asynchronous whereas the `wait()` method of the `Popen` class is implemented as a busy loop.

This class is not thread safe. See also the Subprocess and threads section.

coroutine **wait**()

Wait for child process to terminate. Set and return `returncode` attribute.

This method is a coroutine.

> **Note:** This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use the `communicate()` method when using pipes to avoid that.

coroutine **communicate**(*input=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. The type of *input* must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`.

If a `BrokenPipeError` or `ConnectionResetError` exception is raised when writing *input* into stdin, the exception is ignored. It occurs when the process exits before all data are written into stdin.

Note that if you want to send data to the process's stdin, you need to create the Process object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

This method is a coroutine.

> **Note:** The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.4.2: The method now ignores `BrokenPipeError` and `ConnectionResetError`.

`send_signal`(*signal*)

Sends the signal *signal* to the child process.

> **Note:** On Windows, `SIGTERM` is an alias for `terminate()`. `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

`terminate`()

Stop the child. On Posix OSs the method sends `signal.SIGTERM` to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

`kill`()

Kills the child. On Posix OSs the function sends `SIGKILL` to the child. On Windows `kill()` is an alias for `terminate()`.

`stdin`

Standard input stream (`StreamWriter`), `None` if the process was created with `stdin=None`.

`stdout`

Standard output stream (`StreamReader`), `None` if the process was created with `stdout=None`.

stderr

Standard error stream (`StreamReader`), `None` if the process was created with `stderr=None`.

Warning: Use the `communicate()` method rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to streams pausing reading or writing and blocking the child process.

pid

The identifier of the process.

Note that for processes created by the `create_subprocess_shell()` function, this attribute is the process identifier of the spawned shell.

returncode

Return code of the process when it exited. A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (Unix only).

18.5.6.6. Subprocess and threads

asyncio supports running subprocesses from different threads, but there are limits:

- An event loop must run in the main thread
- The child watcher must be instantiated in the main thread, before executing subprocesses from other threads. Call the `get_child_watcher()` function in the main thread to instantiate the child watcher.

The `asyncio.subprocess.Process` class is not thread safe.

See also: The Concurrency and multithreading in asyncio section.

18.5.6.7. Subprocess examples

18.5.6.7.1. Subprocess using transport and protocol

Example of a subprocess protocol using to get the output of a subprocess and to wait for the subprocess exit. The subprocess is created by the `AbstractEventLoop.subprocess_exec()` method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

@asyncio.coroutine
def get_date(loop):
    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by the protocol DateProtocol,
    # redirect the standard output into a pipe
    create = loop.subprocess_exec(lambda: DateProtocol(exit_future),
                                  sys.executable, '-c', code,
                                  stdin=None, stderr=None)
    transport, protocol = yield from create

    # Wait for the subprocess exit using the process_exited() method
    # of the protocol
    yield from exit_future

    # Close the stdout pipe
    transport.close()

    # Read the output which was collected by the pipe_data_received()
    # method of the protocol
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date(loop))
print("Current date: %s" % date)
loop.close()
```

18.5.6.7.2. Subprocess using streams

Example using the `Process` class to control the subprocess and the `StreamReader` class to read from the standard output. The subprocess is created by the `create_subprocess_exec()` function:

```
import asyncio.subprocess
import sys

@asyncio.coroutine
def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess, redirect the standard output into a pipe
    create = asyncio.create_subprocess_exec(sys.executable, '-c', code
                                            stdout=asyncio.subprocess.
    proc = yield from create

    # Read one line of output
    data = yield from proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit
    yield from proc.wait()
    return line

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date())
print("Current date: %s" % date)
loop.close()
```