select — Wait for I/O Efficiently

Purpose: Wait for notification that an input or output channel is ready.

The `select` module provides access to platform-specific I/O monitoring functions. The most portable interface is the POSIX function `select()`, which is available on Unix and Windows. The module also includes `poll()`, a Unix-only API, and several options that only work with specific variants of Unix.

Note

The new selectors module provides a higher-level interface built on top of the APIs in `select`. It is easier to build portable code using `selectors`, so use that module unless the low-level APIs provided by `select` are somehow required.

Using select()

Python's `select()` function is a direct interface to the underlying operating system implementation. It monitors sockets, open files, and pipes (anything with a `fileno()` method that returns a valid file descriptor) until they become readable or writable or a communication error occurs. `select()` makes it easier to monitor multiple connections at the same time, and is more efficient than writing a polling loop in Python using socket timeouts, because the monitoring happens in the operating system network layer, instead of the interpreter.

Note

Using Python's file objects with `select()` works for Unix, but is not supported under Windows.

The echo server example from the socket section can be extended to watch for more than one connection at a time by using `select()`. The new version starts out by creating a non-blocking TCP/IP socket and configuring it to listen on an address.

```
# select_echo_server.py

import select
import socket
import sys
import queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Bind the socket to the port
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address),
      file=sys.stderr)
server.bind(server_address)

# Listen for incoming connections
server.listen(5)
```

The arguments to `select()` are three lists containing communication channels to monitor. The first is a list of the objects to be checked for incoming data to be read, the second contains objects that will receive outgoing data when there is room in their buffer, and the third those that may have an error (usually a combination of the input and output channel objects). The next step in the server is to set up the lists containing input sources and output destinations to be passed to `select()`.

```
# Sockets from which we expect to read
inputs = [server]

# Sockets to which we expect to write
outputs = []
```

Connections are added to and removed from these lists by the server main loop. Since this version of the server is going to wait for a socket to become writable before sending any data (instead of immediately sending the reply), each output connection needs a queue to act as a buffer for the data to be sent through it.

```
# Outgoing message queues (socket:Queue)
message_queues = {}
```

The main portion of the server program loops, calling `select()` to block and wait for network activity.

```
while inputs:

    # Wait for at least one of the sockets to be
    # ready for processing
    print('waiting for the next event', file=sys.stderr)
    readable, writable, exceptional = select.select(inputs,
                                                    outputs,
                                                    inputs)
```

`select()` returns three new lists, containing subsets of the contents of the lists passed in. All of the sockets in the `readable` list have incoming data buffered and available to be read. All of the sockets in the `writable` list have free space in their buffer and can be written to. The sockets returned in `exceptional` have had an error (the actual definition of “exceptional condition” depends on the platform).

The “readable” sockets represent three possible cases. If the socket is the main “server” socket, the one being used to listen for connections, then the “readable” condition means it is ready to accept another incoming connection. In addition to adding the new connection to the list of inputs to monitor, this section sets the client socket to not block.

```
    # Handle inputs
    for s in readable:

        if s is server:
            # A "readable" socket is ready to accept a connection
            connection, client_address = s.accept()
            print('  connection from', client_address,
                  file=sys.stderr)
            connection.setblocking(0)
            inputs.append(connection)

            # Give the connection a queue for data
            # we want to send
            message_queues[connection] = queue.Queue()
```

The next case is an established connection with a client that has sent data. The data is read with `recv()`, then placed on the queue so it can be sent through the socket and back to the client.

```
        else:
            data = s.recv(1024)
            if data:
                # A readable client socket has data
                print('  received {!r} from {}'.format(
                    data, s.getpeername()), file=sys.stderr,
                )
                message_queues[s].put(data)
                # Add output channel for response
                if s not in outputs:
                    outputs.append(s)
```

A readable socket *without* data available is from a client that has disconnected, and the stream is ready to be closed.

```
            else:
                # Interpret empty result as closed connection
                print('  closing', client_address,
                      file=sys.stderr)
                # Stop listening for input on the connection
                if s in outputs:
                    outputs.remove(s)
                inputs.remove(s)
                s.close()

                # Remove message queue
                del message_queues[s]
```

```
            del message_queues[s]
```

There are fewer cases for the writable connections. If there is data in the queue for a connection, the next message is sent. Otherwise, the connection is removed from the list of output connections so that the next time through the loop `select()` does not indicate that the socket is ready to send data.

```
    # Handle outputs
    for s in writable:
        try:
            next_msg = message_queues[s].get_nowait()
        except queue.Empty:
            # No messages waiting so stop checking
            # for writability.
            print('  ', s.getpeername(), 'queue empty',
                  file=sys.stderr)
            outputs.remove(s)
        else:
            print('  sending {!r} to {}'.format(next_msg,
                                                s.getpeername()),
                  file=sys.stderr)
            s.send(next_msg)
```

Finally, if there is an error with a socket, it is closed.

```
    # Handle "exceptional conditions"
    for s in exceptional:
        print('exception condition on', s.getpeername(),
              file=sys.stderr)
        # Stop listening for input on the connection
        inputs.remove(s)
        if s in outputs:
            outputs.remove(s)
        s.close()

        # Remove message queue
        del message_queues[s]
```

The example client program uses two sockets to demonstrate how the server with `select()` manages multiple connections at the same time. The client starts by connecting each TCP/IP socket to the server.

```
# select_echo_multiclient.py

import socket
import sys

messages = [
    'This is the message. ',
    'It will be sent ',
    'in parts.',
]
server_address = ('localhost', 10000)

# Create a TCP/IP socket
socks = [
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
]

# Connect the socket to the port where the server is listening
print('connecting to {} port {}'.format(*server_address),
      file=sys.stderr)
for s in socks:
    s.connect(server_address)
```

Then it sends one piece of the message at a time via each socket and reads all responses available after writing new data.

```
for message in messages:
    outgoing_data = message.encode()

    # Send messages on both sockets
```

```
    for s in socks:
        print('{}: sending {!r}'.format(s.getsockname(),
                                        outgoing_data),
              file=sys.stderr)
        s.send(outgoing_data)

    # Read responses on both sockets
    for s in socks:
        data = s.recv(1024)
        print('{}: received {!r}'.format(s.getsockname(),
                                         data),
              file=sys.stderr)
        if not data:
            print('closing socket', s.getsockname(),
                  file=sys.stderr)
            s.close()
```

Run the server in one window and the client in another. The output will look like this, with different port numbers.

```
$ python3 select_echo_server.py
starting up on localhost port 10000
waiting for the next event
  connection from ('127.0.0.1', 61003)
waiting for the next event
  connection from ('127.0.0.1', 61004)
waiting for the next event
  received b'This is the message. ' from ('127.0.0.1', 61003)
  received b'This is the message. ' from ('127.0.0.1', 61004)
waiting for the next event
  sending b'This is the message. ' to ('127.0.0.1', 61003)
  sending b'This is the message. ' to ('127.0.0.1', 61004)
waiting for the next event
   ('127.0.0.1', 61003) queue empty
   ('127.0.0.1', 61004) queue empty
waiting for the next event
  received b'It will be sent ' from ('127.0.0.1', 61003)
  received b'It will be sent ' from ('127.0.0.1', 61004)
waiting for the next event
  sending b'It will be sent ' to ('127.0.0.1', 61003)
  sending b'It will be sent ' to ('127.0.0.1', 61004)
waiting for the next event
   ('127.0.0.1', 61003) queue empty
   ('127.0.0.1', 61004) queue empty
waiting for the next event
  received b'in parts.' from ('127.0.0.1', 61003)
waiting for the next event
  received b'in parts.' from ('127.0.0.1', 61004)
  sending b'in parts.' to ('127.0.0.1', 61003)
waiting for the next event
   ('127.0.0.1', 61003) queue empty
  sending b'in parts.' to ('127.0.0.1', 61004)
waiting for the next event
   ('127.0.0.1', 61004) queue empty
waiting for the next event
  closing ('127.0.0.1', 61004)
  closing ('127.0.0.1', 61004)
waiting for the next event
```

The client output shows the data being sent and received using both sockets.

```
$ python3 select_echo_multiclient.py
connecting to localhost port 10000
('127.0.0.1', 61003): sending b'This is the message. '
('127.0.0.1', 61004): sending b'This is the message. '
('127.0.0.1', 61003): received b'This is the message. '
('127.0.0.1', 61004): received b'This is the message. '
('127.0.0.1', 61003): sending b'It will be sent '
('127.0.0.1', 61004): sending b'It will be sent '
('127.0.0.1', 61003): received b'It will be sent '
('127.0.0.1', 61004): received b'It will be sent '
```

```
('127.0.0.1', 61003): sending b'in parts.'
('127.0.0.1', 61004): sending b'in parts.'
('127.0.0.1', 61003): received b'in parts.'
('127.0.0.1', 61004): received b'in parts.'
```

Non-blocking I/O With Timeouts

`select()` also takes an optional fourth parameter, which is the number of seconds to wait before breaking off monitoring if no channels have become active. Using a timeout value lets a main program call `select()` as part of a larger processing loop, taking other actions in between checking for network input.

When the timeout expires, `select()` returns three empty lists. Updating the server example to use a timeout requires adding the extra argument to the `select()` call and handling the empty lists after `select()` returns.

```
# select_echo_server_timeout.py

    readable, writable, exceptional = select.select(inputs,
                                                    outputs,
                                                    inputs,
                                                    timeout)

    if not (readable or writable or exceptional):
        print('  timed out, do some other work here',
              file=sys.stderr)
        continue
```

This “slow” version of the client program pauses after sending each message, to simulate latency or other delay in transmission.

```
# select_echo_slow_client.py

import socket
import sys
import time

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)
print('connecting to {} port {}'.format(*server_address),
      file=sys.stderr)
sock.connect(server_address)

time.sleep(1)

messages = [
    'Part one of the message.',
    'Part two of the message.',
]
amount_expected = len(''.join(messages))

try:

    # Send data
    for message in messages:
        data = message.encode()
        print('sending {!r}'.format(data), file=sys.stderr)
        sock.sendall(data)
        time.sleep(1.5)

    # Look for the response
    amount_received = 0

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data), file=sys.stderr)

finally:
```

```
finally:
    print('closing socket', file=sys.stderr)
    sock.close()
```

Running the new server with the slow client produces:

```
$ python3 select_echo_server_timeout.py
starting up on localhost port 10000
waiting for the next event
  timed out, do some other work here
waiting for the next event
  connection from ('127.0.0.1', 61144)
waiting for the next event
  timed out, do some other work here
waiting for the next event
  received b'Part one of the message.' from ('127.0.0.1', 61144)
waiting for the next event
  sending b'Part one of the message.' to ('127.0.0.1', 61144)
waiting for the next event
('127.0.0.1', 61144) queue empty
waiting for the next event
  timed out, do some other work here
waiting for the next event
  received b'Part two of the message.' from ('127.0.0.1', 61144)
waiting for the next event
  sending b'Part two of the message.' to ('127.0.0.1', 61144)
waiting for the next event
('127.0.0.1', 61144) queue empty
waiting for the next event
  timed out, do some other work here
waiting for the next event
closing ('127.0.0.1', 61144)
waiting for the next event
  timed out, do some other work here
```

And this is the client output:

```
$ python3 select_echo_slow_client.py
connecting to localhost port 10000
sending b'Part one of the message.'
sending b'Part two of the message.'
received b'Part one of the '
received b'message.Part two'
received b' of the message.'
closing socket
```

Using poll()

The `poll()` function provides similar features to `select()`, but the underlying implementation is more efficient. The trade-off is that `poll()` is not supported under Windows, so programs using `poll()` are less portable.

An echo server built on `poll()` starts with the same socket configuration code used in the other examples.

```
# select_poll_echo_server.py
```

```
import select
import socket
import sys
import queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Bind the socket to the port
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address),
      file=sys.stderr)
server.bind(server_address)
```

```
# Listen for incoming connections
server.listen(5)

# Keep up with the queues of outgoing messages
message_queues = {}
```

The timeout value passed to `poll()` is represented in milliseconds, instead of seconds, so in order to pause for a full second the timeout must be set to `1000`.

```
# Do not block forever (milliseconds)
TIMEOUT = 1000
```

Python implements `poll()` with a class that manages the registered data channels being monitored. Channels are added by calling `register()` with flags indicating which events are interesting for that channel. The full set of flags is listed in the table below.

Event Flags for poll()

Event	Description
POLLIN	Input ready
POLLPRI	Priority input ready
POLLOUT	Able to receive output
POLLERR	Error
POLLHUP	Channel closed
POLLNVAL	Channel not open

The echo server will be setting up some sockets just for reading and others to be read from or written to. The appropriate combinations of flags are saved to the local variables `READ_ONLY` and `READ_WRITE`.

```
# Commonly used flag sets
READ_ONLY = (
    select.POLLIN |
    select.POLLPRI |
    select.POLLHUP |
    select.POLLERR
)
READ_WRITE = READ_ONLY | select.POLLOUT
```

The `server` socket is registered so that any incoming connections or data triggers an event.

```
# Set up the poller
poller = select.poll()
poller.register(server, READ_ONLY)
```

Since `poll()` returns a list of tuples containing the file descriptor for the socket and the event flag, a mapping from file descriptor numbers to objects is needed to retrieve the `socket` to read or write from it.

```
# Map file descriptors to socket objects
fd_to_socket = {
    server.fileno(): server,
}
```

The server's loop calls `poll()` and then processes the "events" returned by looking up the socket and taking action based on the flag in the event.

```
while True:

    # Wait for at least one of the sockets to be
    # ready for processing
    print('waiting for the next event', file=sys.stderr)
    events = poller.poll(TIMEOUT)

    for fd, flag in events:

        # Retrieve the actual socket from its file descriptor
```

```
            s = fd_to_socket[fd]
```

As with `select()`, when the main server socket is “readable,” that really means there is a pending connection from a client. The new connection is registered with the `READ_ONLY` flags to watch for new data to come through it.

```
            # Handle inputs
            if flag & (select.POLLIN | select.POLLPRI):

                if s is server:
                    # A readable socket is ready
                    # to accept a connection
                    connection, client_address = s.accept()
                    print('  connection', client_address,
                          file=sys.stderr)
                    connection.setblocking(0)
                    fd_to_socket[connection.fileno()] = connection
                    poller.register(connection, READ_ONLY)

                    # Give the connection a queue for data to send
                    message_queues[connection] = queue.Queue()
```

Sockets other than the server are existing clients and `recv()` is used to access the data waiting to be read.

```
                else:
                    data = s.recv(1024)
```

If `recv()` returns any data, it is placed into the outgoing queue for the socket, and the flags for that socket are changed using `modify()` so `poll()` will watch for the socket to be ready to receive data.

```
                    if data:
                        # A readable client socket has data
                        print('  received {!r} from {}'.format(
                            data, s.getpeername()), file=sys.stderr,
                        )
                        message_queues[s].put(data)
                        # Add output channel for response
                        poller.modify(s, READ_WRITE)
```

An empty string returned by `recv()` means the client disconnected, so `unregister()` is used to tell the `poll` object to ignore the socket.

```
                    else:
                        # Interpret empty result as closed connection
                        print('  closing', client_address,
                              file=sys.stderr)
                        # Stop listening for input on the connection
                        poller.unregister(s)
                        s.close()

                        # Remove message queue
                        del message_queues[s]
```

The `POLLHUP` flag indicates a client that “hung up” the connection without closing it cleanly. The server stops polling clients that disappear.

```
            elif flag & select.POLLHUP:
                # Client hung up
                print('  closing', client_address, '(HUP)',
                      file=sys.stderr)
                # Stop listening for input on the connection
                poller.unregister(s)
                s.close()
```

The handling for writable sockets looks like the version used in the example for `select()`, except that `modify()` is used to change the flags for the socket in the poller, instead of removing it from the output list.

```
            elif flag & select.POLLOUT:
                # Socket is ready to send data,
                # if there is any to send
```

```
                # If there is any to send:
                try:
                    next_msg = message_queues[s].get_nowait()
                except queue.Empty:
                    # No messages waiting so stop checking
                    print(s.getpeername(), 'queue empty',
                          file=sys.stderr)
                    poller.modify(s, READ_ONLY)
                else:
                    print('  sending {!r} to {}'.format(
                        next_msg, s.getpeername()), file=sys.stderr,
                    )
                    s.send(next_msg)
```

And, finally, any events with POLLERR cause the server to close the socket.

```
            elif flag & select.POLLERR:
                print('  exception on', s.getpeername(),
                      file=sys.stderr)
                # Stop listening for input on the connection
                poller.unregister(s)
                s.close()

                # Remove message queue
                del message_queues[s]
```

When the poll-based server is run together with `select_echo_multiclient.py` (the client program that uses multiple sockets), this is the output.

```
$ python3 select_poll_echo_server.py
starting up on localhost port 10000
waiting for the next event
waiting for the next event
waiting for the next event
waiting for the next event
  connection ('127.0.0.1', 61253)
waiting for the next event
  connection ('127.0.0.1', 61254)
waiting for the next event
  received b'This is the message. ' from ('127.0.0.1', 61253)
  received b'This is the message. ' from ('127.0.0.1', 61254)
waiting for the next event
  sending b'This is the message. ' to ('127.0.0.1', 61253)
  sending b'This is the message. ' to ('127.0.0.1', 61254)
waiting for the next event
('127.0.0.1', 61253) queue empty
('127.0.0.1', 61254) queue empty
waiting for the next event
  received b'It will be sent ' from ('127.0.0.1', 61253)
  received b'It will be sent ' from ('127.0.0.1', 61254)
waiting for the next event
  sending b'It will be sent ' to ('127.0.0.1', 61253)
  sending b'It will be sent ' to ('127.0.0.1', 61254)
waiting for the next event
('127.0.0.1', 61253) queue empty
('127.0.0.1', 61254) queue empty
waiting for the next event
  received b'in parts.' from ('127.0.0.1', 61253)
  received b'in parts.' from ('127.0.0.1', 61254)
waiting for the next event
  sending b'in parts.' to ('127.0.0.1', 61253)
  sending b'in parts.' to ('127.0.0.1', 61254)
waiting for the next event
('127.0.0.1', 61253) queue empty
('127.0.0.1', 61254) queue empty
waiting for the next event
  closing ('127.0.0.1', 61254)
waiting for the next event
  closing ('127.0.0.1', 61254)
waiting for the next event
```

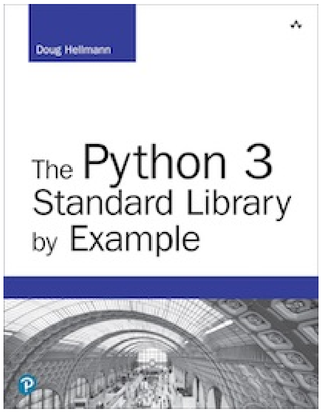
Platform-specific Options

Less portable options provided by `select` are `epoll`, the *edge polling* API supported by Linux; `kqueue`, which uses BSD's *kernel queue*; and `kevent`, BSD's *kernel event* interface. Refer to the operating system library documentation for more detail about how they work.

See also

- Standard library documentation for select
- `selectors` – Higher-level abstraction on top of `select`.
- Socket Programming HOWOTO – An instructional guide by Gordon McMillan, included in the standard library documentation.
- `socket` – Low-level network communication.
- `SocketServer` – Framework for creating network server applications.
- `asyncio` – Asynchronous I/O framework
- *Unix Network Programming, Volume 1: The Sockets Networking API, 3/E* By W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Published by Addison-Wesley Professional, 2004. ISBN-10: 0131411551
- *Foundations of Python Network Programminng, 3/E* By Brandon Rhodes and John Goerzen. Published by Apress, 2014. ISBN-10: 1430258543

This page was last updated 2016-12-18.



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?

Other Writing

- Blog
- The Python Standard Library By Example