Хороший вариант неблокирующего сокета

Этот путь называется select.

У ОС в Unix, также как в Linux и MacOS, есть несколько примитивов, которые позволяют выполнить очень хитрое действие.

Эти хитрые команды мы можем выполнить в терминале:

- select
- poll
- epoll (Linux)
- kqueue (FreeBSD, MacOS)

Функции команд

Сокет — это просто номер файлового дескриптора, который ОС выделила нам при создании сокета. Когда вы открываете файл (а сокет такой же файл), ОС внутри у себя в ядре создает сущность и выдает вам в ответ уникальный для текущего процесса идентификатор этой сущности: обычное число типа 3, 4 или 5.

Мы можем попросить ОС сообщать нам о изменении состояния того или иного файлового дескриптора. Условно, мы просим: «Если в дескриптор под номером 5 придет какая-то информация, и он станет доступен на чтение, сообщи мне, пожалуйста». Наша программа пока может спать, а не бесконечно крутиться в цикле. Как только событие произойдет, ОС сама вызовет нужную часть программы и тем самым продолжит ее выполнение.

Различия между командами

Select — самая старая команда и поддерживает только 1024 файловых дескриптора. **Poll** поддерживает гораздо больше.

Epoll и kqueue являются более совершенными технологиями и является аналогами друг друга. Если система не поддерживает epoll, она скатывается до poll, если не поддерживает poll, скатывается до select.

По результату работу эти команды не отличаются. На вход они принимают файловые дескрипторы и вызывают какую-то логику при наступлении определенных событий в этих дескрипторах.

Пример использования:

```
import socket, selectors

selector = selectors.DefaultSelector()  # выбираем наилучший механизм операционной системы по работе с событиями

sock = socket.socket()  # создаем TCP-сокет
sock.setblocking(False)  # делаем его неблокирующим

try:
    sock.connect(('xkcd.com', 80))  # асинхронно устанавливаем TCP-соединение с xkcd.com
except BlockingIOError:
    pass


# объявляем callback, который будет вызван при разблокировки сокета на запись.
# То есть после установки TCP-соединения
def connected():
    selector.unregister(sock)
    print('connected!')


# подписываемся на событие разблокировки сокета на запись
selector.register(sock, selectors.EVENT_WRITE, connected)


def loop():
    while True:  # event-loop
        events = selector.select()  # заблокируется, пока не произойдет какое-то зарегистрированное событие
        for event_key, event_mask in events:
            # event_mask - это число 1, 2 или 3. То есть 01, 10 или 11 в двоичном виде.
            # 01 - selectors.EVENT_READ, 10 - selectors.EVENT_WRITE, 11 -  selectors.EVENT_READ | selectors.EVENT_WRITE

            # event_key - информация о зарегистрированном сокете и о callback
            callback = event_key.data

            # callback - наша функция connected
            callback()


loop()  # запускаем вечный event-loop
```

этот пример доступен в [examples/sockets/async.py]

Разберем, что происходит в примере выше:

DefaultSelector — класс, который возвращает лучший из доступных механизмов на операционной системе. Вот так это работает:

```
if _can_use('kqueue'):
    DefaultSelector = KqueueSelector
elif _can_use('epoll'):
    DefaultSelector = EpollSelector
elif _can_use('devpoll'):
    DefaultSelector = DevpollSelector
elif _can_use('poll'):
    DefaultSelector = PollSelector
else:
    DefaultSelector = SelectSelector
```

Далее мы создали сокет и выполнили подключение к xkcd.com. В этот момент у нас выбросилось исключение BlockingIOException — подключение еще выполняется, но нам это сейчас не важно. Просто идем дальше, ничего не делая с исключением.

Дальше мы *подписываемся* на определенное событие в сокете. То есть говорим ОС: если этот сокет станет доступен на запись, то сообщи, пожалуйста, мне и вызови функцию *callback.*

Дальше мы переходим к самой интересной части — запускаем свой *Event Loop.*

Event Loop, или цикл событий — это цикл, который бесконечно ожидает появления каких-то событий и выполняет нужную часть программы при их возникновении. Он не проверяет постоянно наличие событий сам (в отличие от предыдущего плохого примера), а блокируется и ждет, пока ОС сообщит ему о наличии таких событий. Так он не нагружает CPU ненужными операциями.

Мы запустили while True и заблокировались на строчке *events = selector.select().* Как только соединение с нашим сокетом установится, *select()* вернет какие-то события (в нашем случае вернется только одно).

Из чего состоят возвращаемые *select()* события? *events* — это список, состоящий из кортежей длины 2.

Первым элементом в кортеже идет *event_key* — объект, который содержит *сокет,* на котором произошло событие, и *callback-функцию*, зарегистрованную на это событие на этом сокете.

Вторым элементом в кортеже идет *event_mask* — число 1, 2 или 3. Переводя эти числа в двоичный вид, получим 01, 10 или 11 соответственно. Это маска события, произошедшего на сокете:

- EventWrite (доступность сокета на запись) — 01
- EventRead (доступность сокета на чтение) — 10
- EventWrite | EventRead = 10 | 01 = 11 — доступность сокета как на чтение, так и на запись

Повторим:

1. мы начали подключение к сокету
2. подписались на событие доступности сокета на запись
3. запустили бесконечный цикл
4. заблокировались и стали дожидаться, пока нужное событие произойдет, и ОС сообщит нам об этом
5. выполнили callback-функцию, привязанную к этому сокету и к этому событию

Так и получается асинхронность. Мы начали выполнять операцию в один момент времени, а потом в другой — неподконтрольный нам — происходит выполнение другой функции.

В следующей карточке мы рассмотрим детальнее, что такое асинхронность.