Параллелизм в Python

Асинхронный vs синхронный процесс

- **синхронный процесс** - задачи друг за другом
- **асинхронный процесс** - задачи стартуют и завершаются независимо от других

Конкурентность и параллелизм

- **Конкурентность** - задачи выполняются совместно
- **Параллелизм** - задачи выполняются отдельно

- синхронность блокирует операции (блокирующие)
- асинхронность не блокирует операции (неблокирующие)
- конкурентность - совместный прогресс (совместные)
- параллелизм - параллельный прогресс (параллельные)

Потоки vs процессы

- Процесс (process) - запущенная на выполнение программа, набор инструкций и текущее её состояние:
    - Образ машинного кода программы
    - Память (стек, куча)
    - Состояние процессора (регистры, режимы работы и т.п.)
    - Текущие исполняемые команды
    - Дескрипторы ОС и файловые, права доступа
    - и т.д.

- Поток (thread) - поток выполнения
    - Обычно существует внутри процесса (процесс может иметь один или несколько потоков выполнения).
    - Описывается машинным кодом, контекстом родительского процесса (в т.ч. общие данные для отдельных потоков) и своим локальным контекстом (например, различное состояние процессора и стека для нескольких потоков одного процесса, собственные неразделяемые данные отдельного процесса (thread-local data)).
    - Имеет меньший чем у процесса описывающий контекст (более "легковесный").
    - За счёт этой "легковесности" и разделения контекста переключение между потоками одного процесса происходит быстрее, чем между отдельными процессами.
    - ОС считает наименьшей единицей исполнения задачи поток (если потоки поддерживаются, иначе процесс) и переключает выполнение между ими.
    - Кроме потоков ядра системы (kernel threads, управляемых ОС) можно иногда также рассматриваются пользовательские потоки (user-space threads), которые реализованы не в ОС, а с помощью библиотеки или конкретного интерпретатора/компилятора.
    - Thread-safe код - код умеющий работать в многопоточном окружении (или не требует синхронизаций, или есть все необходимые).

Некоторые преимущества использования потоков:

- Программы могут оставаться отзывчивыми к пользовательским действиям, если интерфейс работает в отдельном потоке.
- Общие данные, с которыми можно работать из разных потоков.

Возникающие проблемы и задачи

- Запуск новых процессов и потоков
- Диспетчеризация выполняющихся задач (scheduling)

- Доступ к общим ресурсам и обмен информацией между выполняющимися задачами (синхронизация состояния и передача данных)
- Обработка ошибок
- Завершение выполнения

В общем случае:

- Запуск процессов и потоков выполняется через API предоставляемое операционной системой либо через библиотеки, которые являются обертками вокруг этого API.
- Диспетчеризацию выполняет ОС, программа может косвенно на это влиять выставлением параметров и типа диспетчеризации через API либо использовать пользовательскую библиотеку, которая будет реализовывать диспетчеризацию поверх механизмов ОС.
- Программе доступны (напрямую через ОС или через оборачивающие библиотеки) механизмы синхронизации доступа к общим ресурсам, а также передачи сообщений между отдельными процессами и потоками.
- ОС предоставляет механизмы передачи и получения сообщений (сигналов и т.п.) об ошибках, а также их обработку. Также необходимо предусматривать механизмы для восстановления программы после ошибки в отдельных потоках или совместному корректному завершению программы.
- ОС предоставляет механизмы синхронизации и ожидания завершения выполнения, а также получения статуса после завершнения. Необходимо учитывать различные возникающие ситуации (одна из задач зависла, где-то была ошибка, надо дождаться полного завершения или нет и т.д.).

Многопоточность в Python

Python поддерживает многопоточность, API для этого реализовано в библиотеке threading. Однако многопоточность в Python своеобразна.

Модуль threading

In [5]:

```
import threading
```

Потоки представлены классом Thread:

- Он описывает API по запуску и завершению выполнения потоков.
- Он может служить обёрткой для пользовательской функции, которую нужно запустить на выполнение в потоке.
- От него можно наследоваться и перекрывать метод run(), чтобы описать свой поток.

In [14]:

```
def worker(number):
    """thread worker function"""
    print(f"I'm student, doing lab {number + 1} ")
    return

threads = []
for i in range(4):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
I'm student, doing lab 1
I'm student, doing lab 2
I'm student, doing lab 3
I'm student, doing lab 4
```

In [15]:

```
import time
import random

def worker(number):
    """thread worker function"""
    seconds = random.randint(1, 10)
    time.sleep(seconds)
    print(f"I'm student, doing lab {number + 1} after sleeping {seconds}")
    return

threads = []
for i in range(4):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
I'm student, doing lab 3 after sleeping 1
I'm student, doing lab 1 after sleeping 2
I'm student, doing lab 4 after sleeping 2
I'm student, doing lab 2 after sleeping 4
```

Есть возможность определения текущего потока

In [17]:

```
import threading
import time

def worker():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(2)
    print(threading.currentThread().getName(), 'Exiting')

def my_service():
    print(threading.currentThread().getName(), 'Starting')
    time.sleep(3)
    print(threading.currentThread().getName(), 'Exiting')

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()
worker Starting
Thread-45 Starting
my_service Starting
worker Exiting
Thread-45 Exiting
my_service Exiting
```

In [19]:

```
import logging
import threading
```

```
import time

log_handler = logging.StreamHandler()
log_handler.setFormatter(
    logging.Formatter('[%(levelname)s] %(asctime)s (%(threadName)-
10s) %(message)s')
)
logger = logging.getLogger()
logger.handlers = []
logger.addHandler(log_handler)
logger.setLevel(logging.DEBUG)


def worker():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')


def my_service():
    logging.debug('Starting')
    time.sleep(0.3)
    logging.debug('Exiting')


logging.basicConfig(
    level=logging.DEBUG,
    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
)

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker)  # use default name

w.start()
w2.start()
t.start()
[DEBUG] 2021-03-25 22:07:58,987 (worker    ) Starting
[DEBUG] 2021-03-25 22:07:58,991 (Thread-47 ) Starting
[DEBUG] 2021-03-25 22:07:58,993 (my_service) Starting
[DEBUG] 2021-03-25 22:07:59,190 (worker    ) Exiting
[DEBUG] 2021-03-25 22:07:59,193 (Thread-47 ) Exiting
[DEBUG] 2021-03-25 22:07:59,296 (my_service) Exiting
```

Потоки демоны и не демоны

- По-умолчанию программы ждут, когда завершатся все созданные потоки.
- Иногда нужно, чтобы главный поток не ожидал завершения побочных.
- В таком случае можно дать потоку метку демона вызвав setDaemon(True).
- По-умолчанию этот флаг выключен.

In [20]:

```
def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
[DEBUG] 2021-03-25 22:11:18,778 (daemon    ) Starting
[DEBUG] 2021-03-25 22:11:18,779 (non-daemon) Starting
[DEBUG] 2021-03-25 22:11:18,781 (non-daemon) Exiting
[DEBUG] 2021-03-25 22:11:20,780 (daemon    ) Exiting
```

Если бы данный пример запускался из файла, то последнего лога бы не было.

Для ожидания завершения потока-демона можно воспользоваться функцией join().

In [21]:

```
def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join()
t.join()
[DEBUG] 2021-03-25 22:12:43,067 (daemon    ) Starting
[DEBUG] 2021-03-25 22:12:43,068 (non-daemon) Starting
[DEBUG] 2021-03-25 22:12:43,071 (non-daemon) Exiting
[DEBUG] 2021-03-25 22:12:45,072 (daemon    ) Exiting
```

Тут может быть разница при запуске из файла прошлого и этого примера.

По-умолчанию join() блокируется на неограниченное время, но можно указать timeout:

In [23]:

```
def daemon():
```

```
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join(1)
logging.debug('d.is_alive() {}'.format(d.is_alive()))
t.join()
[DEBUG] 2021-03-25 22:14:15,180 (daemon    ) Starting
[DEBUG] 2021-03-25 22:14:15,180 (non-daemon) Starting
[DEBUG] 2021-03-25 22:14:15,181 (non-daemon) Exiting
[DEBUG] 2021-03-25 22:14:16,181 (MainThread) d.is_alive() True
[DEBUG] 2021-03-25 22:14:17,183 (daemon    ) Exiting
```

Обход всех потоков

In [24]:

```
import random

def worker():
    """thread worker function"""
    t = threading.currentThread()
    pause = random.randint(1,5)
    logging.debug('sleeping %s', pause)
    time.sleep(pause)
    logging.debug('ending')
    return

for i in range(3):
    t = threading.Thread(target=worker)
    t.setDaemon(True)
    t.start()

main_thread = threading.currentThread()
for t in threading.enumerate():
    break
    if t is main_thread: # waiting for main thread causes deadlock
        continue
    logging.debug('joining %s', t.getName())
    t.join(1)
```

```
[DEBUG] 2021-03-25 22:15:32,992 (Thread-48 ) sleeping 3
[DEBUG] 2021-03-25 22:15:32,994 (Thread-49 ) sleeping 5
[DEBUG] 2021-03-25 22:15:32,995 (Thread-50 ) sleeping 1
[DEBUG] 2021-03-25 22:15:34,003 (Thread-50 ) ending
[DEBUG] 2021-03-25 22:15:35,999 (Thread-48 ) ending
[DEBUG] 2021-03-25 22:15:38,005 (Thread-49 ) ending
```

Есть возможность наследования от Thread

In [25]:

```
class MyThread(threading.Thread):

    def run(self):
        logging.debug('running')
        return

for i in range(5):
    t = MyThread()
    t.start()
```

```
[DEBUG] 2021-03-25 22:19:11,773 (Thread-51 ) running
[DEBUG] 2021-03-25 22:19:11,773 (Thread-52 ) running
[DEBUG] 2021-03-25 22:19:11,774 (Thread-53 ) running
[DEBUG] 2021-03-25 22:19:11,774 (Thread-54 ) running
[DEBUG] 2021-03-25 22:19:11,775 (Thread-55 ) running
```

In [28]:

```
class MyThreadWithArgs(threading.Thread):

    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None):
        threading.Thread.__init__(self, group=group, target=target, name=name
, daemon=daemon)
        self.args = args
        self.kwargs = kwargs

    def run(self):
        logging.debug('running with %s and %s', self.args, self.kwargs)

for i in range(5):
    t = MyThreadWithArgs(args=(i,), kwargs={'a':'A', 'b':'B'})
    t.start()
```

```
[DEBUG] 2021-03-25 22:20:51,250 (Thread-61 ) running with (0,) and {'a': 'A',
'b': 'B'}
[DEBUG] 2021-03-25 22:20:51,253 (Thread-62 ) running with (1,) and {'a': 'A',
'b': 'B'}
[DEBUG] 2021-03-25 22:20:51,254 (Thread-63 ) running with (2,) and {'a': 'A',
'b': 'B'}
[DEBUG] 2021-03-25 22:20:51,256 (Thread-64 ) running with (3,) and {'a': 'A',
'b': 'B'}
[DEBUG] 2021-03-25 22:20:51,256 (Thread-65 ) running with (4,) and {'a': 'A',
'b': 'B'}
```

Потоки и таймеры

- Timer - поток, выполняющий функцию не сразу, а после некоторого ожидания.
- Такой поток можно отменить в любой момент до начала.

In [29]:

```
def delayed():
    logging.debug('worker running')
    return

t1 = threading.Timer(3, delayed)
t1.setName('t1')
t2 = threading.Timer(3, delayed)
t2.setName('t2')

logging.debug('starting timers')
t1.start()
t2.start()

logging.debug('waiting before canceling %s', t2.getName())
time.sleep(2)
logging.debug('canceling %s', t2.getName())
t2.cancel()
logging.debug('done')
[DEBUG] 2021-03-25 22:21:38,837 (MainThread) starting timers
[DEBUG] 2021-03-25 22:21:38,841 (MainThread) waiting before canceling t2
[DEBUG] 2021-03-25 22:21:40,844 (MainThread) canceling t2
[DEBUG] 2021-03-25 22:21:40,845 (MainThread) done
[DEBUG] 2021-03-25 22:21:41,840 (t1        ) worker running
```

Общение между потоками по Event

- Класс, позволяющий синхронизировать действия в различных потоках.
- set() - установить внутренний флаг.
- clear() - снять внутренний флаг.
- wait() - ждать, пока будет установлен флаг (set()).
- Аргумент в wait(..) - на сколько секунд блокироваться. Возвращаемое значение - установлен ли проверяемый флаг.
- is_set() - неблокирущая проверка установленности флага.

In [31]:

```
def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    logging.debug('wait_for_event starting')
    event_is_set = e.wait()
    logging.debug('event set: %s', event_is_set)

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.is_set():
        logging.debug('wait_for_event_timeout starting')
        event_is_set = e.wait(t)
        logging.debug('event set: %s', event_is_set)
```

```
        if event_is_set:
            logging.debug('processing event')
        else:
            logging.debug('doing other work')


e = threading.Event()
t1 = threading.Thread(name='block',
                      target=wait_for_event,
                      args=(e,))
t1.start()

t2 = threading.Thread(name='non-block',
                      target=wait_for_event_timeout,
                      args=(e, 2))
t2.start()

logging.debug('Waiting before calling Event.set()')
time.sleep(3)
e.set()
logging.debug('Event is set')
[DEBUG] 2021-03-25 22:23:17,097 (block     ) wait_for_event starting
[DEBUG] 2021-03-25 22:23:17,098 (non-block ) wait_for_event_timeout starting
[DEBUG] 2021-03-25 22:23:17,098 (MainThread) Waiting before calling
Event.set()
[DEBUG] 2021-03-25 22:23:19,099 (non-block ) event set: False
[DEBUG] 2021-03-25 22:23:19,101 (non-block ) doing other work
[DEBUG] 2021-03-25 22:23:19,102 (non-block ) wait_for_event_timeout starting
[DEBUG] 2021-03-25 22:23:20,102 (MainThread) Event is set
[DEBUG] 2021-03-25 22:23:20,102 (block     ) event set: True
[DEBUG] 2021-03-25 22:23:20,103 (non-block ) event set: True
[DEBUG] 2021-03-25 22:23:20,104 (non-block ) processing event
```

В этом примере wait_for_event_timeout() проверяет статус события без бесконечного блокирования. wait_for_event() блокируется вызовом wait и не завершается пока статус не поменяется

Защита совместного доступа к ресурсам

- Стандартные типы данных защищены GIL (lists, dicts, etc).
- Но примитивные нет (integers, floats), а также пользовательские.
- При обращении из нескольких потоков могут возникать проблемы из-за одновременного доступа к данным.
- Общие данные нужно ограничивать.
- В модуле threading для этого можно воспользоваться классом Lock.

In [32]:

```
import random

class Counter(object):
    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.value = start
    def increment(self):
```

```
        logging.debug('Waiting for lock')
        self.lock.acquire()
        try:
            logging.debug('Acquired lock')
            self.value = self.value + 1
        finally:
            self.lock.release()

def worker(c):
    for i in range(4):
        pause = random.random()
        logging.debug('Sleeping %0.02f', pause)
        time.sleep(pause)
        c.increment()
    logging.debug('Done')

counter = Counter()
for i in range(2):
    t = threading.Thread(target=worker, args=(counter,))
    t.start()

logging.debug('Waiting for worker threads')
main_thread = threading.currentThread()
for t in threading.enumerate():
    if t is not main_thread:
        # there are other ipython threads so limit join wait to avoid eternal
waiting
        t.join(5)
logging.debug('Counter: %d', counter.value)
[DEBUG] 2021-03-25 22:29:57,586 (Thread-68 ) Sleeping 0.55
[DEBUG] 2021-03-25 22:29:57,587 (Thread-69 ) Sleeping 0.72
[DEBUG] 2021-03-25 22:29:57,587 (MainThread) Waiting for worker threads
[DEBUG] 2021-03-25 22:29:58,138 (Thread-68 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:58,139 (Thread-68 ) Acquired lock
[DEBUG] 2021-03-25 22:29:58,141 (Thread-68 ) Sleeping 0.46
[DEBUG] 2021-03-25 22:29:58,309 (Thread-69 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:58,310 (Thread-69 ) Acquired lock
[DEBUG] 2021-03-25 22:29:58,310 (Thread-69 ) Sleeping 0.20
[DEBUG] 2021-03-25 22:29:58,507 (Thread-69 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:58,509 (Thread-69 ) Acquired lock
[DEBUG] 2021-03-25 22:29:58,511 (Thread-69 ) Sleeping 0.32
[DEBUG] 2021-03-25 22:29:58,607 (Thread-68 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:58,609 (Thread-68 ) Acquired lock
[DEBUG] 2021-03-25 22:29:58,611 (Thread-68 ) Sleeping 0.45
[DEBUG] 2021-03-25 22:29:58,834 (Thread-69 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:58,836 (Thread-69 ) Acquired lock
[DEBUG] 2021-03-25 22:29:58,838 (Thread-69 ) Sleeping 0.70
[DEBUG] 2021-03-25 22:29:59,068 (Thread-68 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:59,070 (Thread-68 ) Acquired lock
[DEBUG] 2021-03-25 22:29:59,071 (Thread-68 ) Sleeping 0.60
[DEBUG] 2021-03-25 22:29:59,536 (Thread-69 ) Waiting for lock
```

```
[DEBUG] 2021-03-25 22:29:59,538 (Thread-69 ) Acquired lock
[DEBUG] 2021-03-25 22:29:59,539 (Thread-69 ) Done
[DEBUG] 2021-03-25 22:29:59,670 (Thread-68 ) Waiting for lock
[DEBUG] 2021-03-25 22:29:59,671 (Thread-68 ) Acquired lock
[DEBUG] 2021-03-25 22:29:59,673 (Thread-68 ) Done
[DEBUG] 2021-03-25 22:30:17,588 (MainThread) Counter: 8
```

С помощью acquire(False) можно просто узнать статус блокировки.

In [33]:

```
def lock_holder(lock, stop_event):
    logging.debug('Starting')
    while not stop_event.is_set():
        lock.acquire()
        try:
            logging.debug('Holding')
            time.sleep(0.5)
        finally:
            logging.debug('Not holding')
            lock.release()
        time.sleep(0.5)
    return

def worker(lock):
    logging.debug('Starting')
    num_tries = 0
    num_acquires = 0
    while num_acquires < 3:
        time.sleep(0.5)
        logging.debug('Trying to acquire')
        have_it = lock.acquire(False)
        try:
            num_tries += 1
            if have_it:
                logging.debug('Iteration %d: Acquired',  num_tries)
                num_acquires += 1
            else:
                logging.debug('Iteration %d: Not acquired', num_tries)
        finally:
            if have_it:
                lock.release()
    logging.debug('Done after %d iterations', num_tries)


lock = threading.Lock()
stop_event = threading.Event()

holder = threading.Thread(target=lock_holder, args=(lock,stop_event), name='L
ockHolder')
holder.setDaemon(True)
holder.start()

worker = threading.Thread(target=worker, args=(lock,), name='Worker')
```

```
worker.start()

worker.join()
stop_event.set()
[DEBUG] 2021-03-25 22:42:12,224 (LockHolder) Starting
[DEBUG] 2021-03-25 22:42:12,225 (Worker    ) Starting
[DEBUG] 2021-03-25 22:42:12,225 (LockHolder) Holding
[DEBUG] 2021-03-25 22:42:12,726 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:12,727 (LockHolder) Not holding
[DEBUG] 2021-03-25 22:42:12,729 (Worker    ) Iteration 1: Not acquired
[DEBUG] 2021-03-25 22:42:13,231 (LockHolder) Holding
[DEBUG] 2021-03-25 22:42:13,233 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:13,235 (Worker    ) Iteration 2: Not acquired
[DEBUG] 2021-03-25 22:42:13,734 (LockHolder) Not holding
[DEBUG] 2021-03-25 22:42:13,737 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:13,739 (Worker    ) Iteration 3: Acquired
[DEBUG] 2021-03-25 22:42:14,236 (LockHolder) Holding
[DEBUG] 2021-03-25 22:42:14,241 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:14,243 (Worker    ) Iteration 4: Not acquired
[DEBUG] 2021-03-25 22:42:14,738 (LockHolder) Not holding
[DEBUG] 2021-03-25 22:42:14,745 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:14,747 (Worker    ) Iteration 5: Acquired
[DEBUG] 2021-03-25 22:42:15,240 (LockHolder) Holding
[DEBUG] 2021-03-25 22:42:15,249 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:15,250 (Worker    ) Iteration 6: Not acquired
[DEBUG] 2021-03-25 22:42:15,742 (LockHolder) Not holding
[DEBUG] 2021-03-25 22:42:15,751 (Worker    ) Trying to acquire
[DEBUG] 2021-03-25 22:42:15,753 (Worker    ) Iteration 7: Acquired
[DEBUG] 2021-03-25 22:42:15,754 (Worker    ) Done after 7 iterations
```

Re-entrant lock

- В случае с обычным Lock при попытке его захватить из того же потока будет провал
- Чтобы дать возможность в одном и том же потоке несколько раз захватывать лок, можно использовать класс RLock (reentrant lock).

In [34]:

```
lock = threading.Lock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
First try : True
Second try: False
```

In [37]:

```
lock = threading.RLock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
First try : True
Second try: True
```

Локи и контекстные менеджеры

In [38]:

```
def worker_with(lock):
    with lock:
        logging.debug('Lock acquired via with')

def worker_no_with(lock):
    lock.acquire()
    try:
        logging.debug('Lock acquired directly')
    finally:
        lock.release()

lock = threading.Lock()
w = threading.Thread(target=worker_with, args=(lock,))
nw = threading.Thread(target=worker_no_with, args=(lock,))

w.start()
nw.start()
[DEBUG] 2021-03-25 22:45:13,221 (Thread-70 ) Lock acquired via with
[DEBUG] 2021-03-25 22:45:13,226 (Thread-71 ) Lock acquired directly
```

worker_with и worker_no_with делают одно и то же

Синхронизация через condition

- Условная переменная.
- Для задач, когда нам нужно соединить потоки производителей и потребителей данных.
- Бывает полезно иметь возможность сигнализировать одному или нескольким потребителям.
- Класс Condition оборачивает Lock (или неявно создаёт, или ему можно передать).
- acquire() и release() вызывают соответствующие методы у обёрнутого лока.
- wait() отпускает лок и засыпает в ожидании notify() или notifyAll(), после чего пытается захватить лок.
- notify() - пробуждает один из ожидающих на wait() потоки.
- notifyAll() - пробуждает все ожидающие на wait() потоки.

In [40]:

```
def consumer(cond):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    t = threading.currentThread()
    with cond:
        cond.wait()
        logging.debug('Resource is available to consumer')

def producer(cond):
    """set up the resource to be used by the consumer"""
    logging.debug('Starting producer thread')
    with cond:
        logging.debug('Making resource available')
        cond.notifyAll()

condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer, args=(condition,))
c2 = threading.Thread(name='c2', target=consumer, args=(condition,))
```

```
p = threading.Thread(name='p', target=producer, args=(condition,))

c1.start()
time.sleep(2)
c2.start()
time.sleep(2)
p.start()
[DEBUG] 2021-03-25 22:46:36,537 (c1        ) Starting consumer thread
[DEBUG] 2021-03-25 22:46:38,539 (c2        ) Starting consumer thread
[DEBUG] 2021-03-25 22:46:40,543 (p         ) Starting producer thread
[DEBUG] 2021-03-25 22:46:40,545 (p         ) Making resource available
[DEBUG] 2021-03-25 22:46:40,548 (c2        ) Resource is available to
consumer
[DEBUG] 2021-03-25 22:46:40,550 (c1        ) Resource is available to
consumer
```

Семафоры для допуска нескольких потоков к ресурсу

- Семафор.
- Можно использовать для задач, когда нам нужно ограничить доступ к общему ресурсу, но можно пустить к нему несколько потоков.
- Пример: пул сетевых соединений или ограничение на количество одновременных загрузок.
- Имеет свой внутренний счётчик.
- Вызов acquire(), если счётчик больше 0, делает минус 1, иначе ждёт пока счётчик не станет больше нуля.
- Вызов release() увеличивает счётчик на единицу ("отдаёт обратно").

In [41]:

```
class ActivePool(object):
    def __init__(self):
        super(ActivePool, self).__init__()
        self.active = []
        self.lock = threading.Lock()
    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
            logging.debug('Running: %s', self.active)
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
            logging.debug('Running: %s', self.active)

def worker(s, pool):
    logging.debug('Waiting to join the pool')
    with s:
        name = threading.currentThread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)

pool = ActivePool()
s = threading.Semaphore(2)
```

```
for i in range(4):
    t = threading.Thread(target=worker, name=str(i), args=(s, pool))
    t.start()
```

```
[DEBUG] 2021-03-25 22:51:23,862 (0         ) Waiting to join the pool
[DEBUG] 2021-03-25 22:51:23,863 (1         ) Waiting to join the pool
[DEBUG] 2021-03-25 22:51:23,864 (2         ) Waiting to join the pool
[DEBUG] 2021-03-25 22:51:23,865 (3         ) Waiting to join the pool
[DEBUG] 2021-03-25 22:51:23,866 (0         ) Running: ['0']
[DEBUG] 2021-03-25 22:51:23,875 (1         ) Running: ['0', '1']
[DEBUG] 2021-03-25 22:51:23,975 (0         ) Running: ['1']
[DEBUG] 2021-03-25 22:51:23,978 (1         ) Running: []
[DEBUG] 2021-03-25 22:51:23,980 (3         ) Running: ['3']
[DEBUG] 2021-03-25 22:51:23,981 (2         ) Running: ['3', '2']
[DEBUG] 2021-03-25 22:51:24,082 (3         ) Running: ['2']
[DEBUG] 2021-03-25 22:51:24,083 (2         ) Running: []
```

Приватные данные потока

- thread-specific data
- threading.local() создаёт объект, через который можно работать с такими данными.
- Не видны из соседнего потока.

In [43]:

```
def show_value(data):
    try:
        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)


def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)

local_data = threading.local()
show_value(local_data)
local_data.value = 1000
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()
```

```
[DEBUG] 2021-03-25 22:51:53,665 (MainThread) No value yet
[DEBUG] 2021-03-25 22:51:53,667 (MainThread) value=1000
[DEBUG] 2021-03-25 22:51:53,670 (Thread-72 ) No value yet
[DEBUG] 2021-03-25 22:51:53,670 (Thread-73 ) No value yet
[DEBUG] 2021-03-25 22:51:53,672 (Thread-72 ) value=65
[DEBUG] 2021-03-25 22:51:53,674 (Thread-73 ) value=44
```

Отнаследуемся, чтобы задать значение по-умолчанию:

In [45]:

```
def show_value(data):
    try:
        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)

def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)

class MyLocal(threading.local):
    def __init__(self, value):
        logging.debug('Initializing %r', self)
        self.value = value

local_data = MyLocal(1000)
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()
[DEBUG] 2021-03-25 22:52:29,260 (MainThread) Initializing <__main__.MyLocal
object at 0x7f88c2ae84c0>
[DEBUG] 2021-03-25 22:52:29,261 (MainThread) value=1000
[DEBUG] 2021-03-25 22:52:29,261 (Thread-74 ) Initializing <__main__.MyLocal
object at 0x7f88c2ae84c0>
[DEBUG] 2021-03-25 22:52:29,262 (Thread-75 ) Initializing <__main__.MyLocal
object at 0x7f88c2ae84c0>
[DEBUG] 2021-03-25 22:52:29,262 (Thread-74 ) value=1000
[DEBUG] 2021-03-25 22:52:29,263 (Thread-75 ) value=1000
[DEBUG] 2021-03-25 22:52:29,263 (Thread-74 ) value=68
[DEBUG] 2021-03-25 22:52:29,264 (Thread-75 ) value=40
```

Только один поток

- Стандартный интерпретатор берёт на себя синхронизацию доступа к общим данным потоков.
- Делает он это с помощью GIL (global interpreter lock) - общий для всего интерпретатора механизм, который позволяет единомоментно выполняться только одному потоку.
- Из-за этого факта выполнение в несколько потоков может быть медленнее выполнения в один поток (борьба за общий ресурс).
- Это верно не для всех интерпретаторов, поэтому если пишутся программы с использованием threading, то их надо писать как будто GIL нет (использовать примитивы синхронизации и т.п.), чтобы не было проблем выполнением на других интерпретаторах (или, возможно, в будущем).
- GIL не защищает от абсолютно всех случаев, некоторые действия могут не защищаться глобальным локом (работа с примитивными типами) и в этим моменты два потока могу работать параллельно (и могут быть ошибки, в т.ч. поэтому доступ к общим данным надо всё равно синхронизировать).

Event loop

Docs

Ядро любого приложения, которое использует asyncio.

Он представляет собой набор асинхронных задач, колбеков, I/O операций и запущенных подзадач.

Фьючерсы

Docs

Объекты в которых хранится текуший результат выполнения какой-либо задачи.

Сопрограммы (корутины)

Корутины - потоки выполнения кода, которые организуются поверх аппаратных потоков.

Поток выполнения кода - последовательность операций.

В определенный момент эта последовательность может быть приостановлена, а вместо нее начинает выполнение другая последовательность.

До Python 3.5 корутины реализовывались с помощью расширенных генераторов.

Docs

Начиная с python 3.5 корутины объявляются с помощью async/await синтаксиса.

Принципы async/await:

- async def синтаксис представляет или нативный корутин, или асинхронный генератор
- ключевое слово await возвращает контроль основному циклу событий
- внутри async функции возвращать значение можно как и в обычных функциях, то есть через return, yield или вообще не писать
- если используется yield, то создается асинхронный генератор
- при зададании асинхронной функции недопустимо использование yield from - будет вызвана синтаксическая ошибка
- await можно использовать только в теле корутина

In [3]:

```
async def f(x):

    y = await z(x)  # OK - `await` and `return` allowed in coroutines

    return y



async def g(x):

    yield x  # OK - this is an async generator
```

```
async def m(x):
    yield from gen(x)  # No - SyntaxError


def m(x):
    y = await z(x)  # Still no - SyntaxError (no `async def` here)
    return y


  File "<ipython-input-3-42a15f3a066e>", line 9
    yield from gen(x)  # No - SyntaxError
    ^
SyntaxError: 'yield from' inside async function
```

Внимамание! Примеры ниже для IPython. В вашей программе код может отличаться

In [4]:

```
import asyncio


async def worker():
    print('Worker starting')
    await asyncio.sleep(0)
    print('Worker exiting')


async def service():
    print('Service starting')
    await asyncio.sleep(0)
    print('Service exiting')


await worker()
await service()
Worker starting
```

Worker exiting

Service starting

Service exiting

Есть возможность запустить в параллель с переключением между контекстами

In [5]:

```
import asyncio
from ipykernel.eventloops import register_integration

@register_integration('asyncio')
def loop_asyncio(kernel):
    '''Start a kernel with asyncio event loop support.'''
    loop = asyncio.get_event_loop()

    def kernel_handler():
        loop.call_soon(kernel.do_one_iteration)
        loop.call_later(kernel._poll_interval, kernel_handler)

    loop.call_soon(kernel_handler)
    try:
        if not loop.is_running():
            loop.run_forever()
    finally:
        loop.run_until_complete(loop.shutdown_asyncgens())
        loop.close()
```

In [6]:

```
import asyncio

loop = asyncio.get_event_loop()
```

```
async def worker():
    print('Worker starting')
    await asyncio.sleep(0)
    print('Worker exiting')

async def service():
    print('Service starting')
    await asyncio.sleep(0)
    print('Service exiting')



tasks = [loop.create_task(worker()), loop.create_task(service())]
wait_object = asyncio.wait(tasks)
asyncio.run_coroutine_threadsafe(wait_object, loop)

#loop.run_until_complete(wait_object)
#loop.close()
```

Out[6]:

```
<Future at 0x7ff7b44a0d90 state=pending>
Worker starting
Service starting
Worker exiting
Service exiting
```

Пояснение

- Объявляем пару простейших корутин
- Корутины могут быть запущены только из другой корутины или обернуты в задачу с помощью метода create_task()
- Объединяем, используя wait()
- Отправка на выполнение в цикл событий используя run_until_complete()

Когда мы используем await в какой-либо корутине, мы говорим, что корутина может отдать управление обратно event loop, который перейдет к следующей задаче.

In [7]:

```
import time

loop = asyncio.get_event_loop()

start = time.time()


def log_time():

    return f'{(time.time() - start)} seconds from start'


async def worker1():

    print('Worker1 starting')

    await asyncio.sleep(2)

    print(f'Worker1 exiting. {log_time()}')


async def worker2():

    print('Worker2 starting')

    await asyncio.sleep(2)

    print(f'Worker2 exiting. {log_time()}')


async def worker3():

    print('Worker3 starting')

    await asyncio.sleep(1)

    print(f'Worker3 exiting. {log_time()}')




tasks = [loop.create_task(worker1()), loop.create_task(worker2()), loop.create_task(worker3())]

wait_object = asyncio.wait(tasks)

asyncio.run_coroutine_threadsafe(wait_object, loop)
```

Out[7]:

<Future at 0x7ff7b4421a60 state=pending>

Worker1 starting

Worker2 starting

Worker3 starting

Worker3 exiting. 1.005699634552002 seconds from start

Worker1 exiting. 2.0052363872528076 seconds from start

Worker2 exiting. 2.0054433345794678 seconds from start

Колбеки

In [8]:

```
import time

loop = asyncio.get_event_loop()

start = time.time()


def log_time():

    return f'{(time.time() - start)} seconds from start'


async def worker1():

    print('Worker1 starting')

    await asyncio.sleep(2)

    print(f'Worker1 exiting. {log_time()}')




print(log_time())

tasks = [loop.create_task(worker1())]

wait_object = asyncio.wait(tasks)

asyncio.run_coroutine_threadsafe(wait_object, loop)

print(log_time())
```

8.249282836914062e-05 seconds from start

0.00028014183044433594 seconds from start

Worker1 starting

Worker1 exiting. 2.004279613494873 seconds from start

In [9]:

```
import time
loop = asyncio.get_event_loop()
start = time.time()


def log_time():
    return f'{(time.time() - start)} seconds from start'


async def worker1():
    print('Worker1 starting')
    await asyncio.sleep(2)
    print(f'Worker1 exiting. {log_time()}')


def callback(result):
    print(f'Hello from callback {log_time()}')



print(f'Hello from the first line {log_time()}')
tasks = [loop.create_task(worker1())]
wait_object = asyncio.wait(tasks)
# wait_object.add_done_callback(log_time)
fut = asyncio.run_coroutine_threadsafe(wait_object, loop)
fut.add_done_callback(callback)
print(f'Hello from the last line {log_time()}')
```

Hello from the first line 0.00024056434631347656 seconds from start

Hello from the last line 0.0005934238433837891 seconds from start

Worker1 starting

Worker1 exiting. 2.006852865219116 seconds from start

Hello from callback 2.007354974746704 seconds from start

В асинхронном мире мы не можем быть уверены, что код, который мы написали, выполнится последовательно

In [10]:

```
import random

loop = asyncio.get_event_loop()

start = time.time()


def worker(id):
    seconds = random.randint(1, 2)
    time.sleep(seconds)
    print(f'Sync worker {id} done since {log_time()}')


async def worker_coroutine(id):
    seconds = random.randint(1, 2)
    await asyncio.sleep(seconds)
    print(f'Async worker {id} done since {log_time()}')



for i in range(5):
    worker(i)


async_tasks = [loop.create_task(worker_coroutine(i)) for i in range(5)]

wait_object = asyncio.wait(tasks)

asyncio.run_coroutine_threadsafe(wait_object, loop)
```

Sync worker 0 done since 2.0023434162139893 seconds from start

Sync worker 1 done since 3.003688097000122 seconds from start

Sync worker 2 done since 5.004488706588745 seconds from start

Sync worker 3 done since 7.0068278312683105 seconds from start

Sync worker 4 done since 8.0081307888031 seconds from start

Out[10]:

<Future at 0x7ff7b4421370 state=pending>

Async worker 2 done since 9.020669221878052 seconds from start

Async worker 3 done since 9.020865440368652 seconds from start

Async worker 4 done since 9.022058010101318 seconds from start

Async worker 0 done since 10.021488904953003 seconds from start

Async worker 1 done since 10.021677494049072 seconds from start

Более реальные примеры

In [11]:

```
import time
import urllib.request
import asyncio
import aiohttp


URL = 'https://api.github.com/events'
MAX_CLIENTS = 3



def fetch_sync(pid):
    print('Fetch sync process {} started'.format(pid))
    start = time.time()
    response = urllib.request.urlopen(URL)
    datetime = response.getheader('Date')
```

```
    print('Process {}: {}, took: {:.2f} seconds'.format(
        pid, datetime, time.time() - start))

    return datetime


async def fetch_async(pid):
    async with aiohttp.ClientSession() as session:
        print('Fetch async process {} started'.format(pid))
        async with session.get(URL) as response:
            datetime = response.headers.get('Date')
            print('Process {}: {}, took: {:.2f} seconds'.format(pid, datetime, time.time() - start))



def synchronous():
    start = time.time()
    for i in range(1, MAX_CLIENTS + 1):
        fetch_sync(i)
    print("Process took: {:.2f} seconds".format(time.time() - start))



print('Synchronous:')
synchronous()

print('Asynchronous:')
```

```
loop = asyncio.get_event_loop()

start = time.time()

tasks = [asyncio.create_task(fetch_async(i)) for i in range(1, MAX_CLIENTS + 1)]

wait_object = asyncio.wait(tasks)

asyncio.run_coroutine_threadsafe(wait_object, loop)
```

Synchronous:

```
Fetch sync process 1 started

Process 1: Fri, 02 Apr 2021 05:40:52 GMT, took: 0.39 seconds

Fetch sync process 2 started

Process 2: Fri, 02 Apr 2021 05:40:52 GMT, took: 0.15 seconds

Fetch sync process 3 started

Process 3: Fri, 02 Apr 2021 05:40:52 GMT, took: 0.14 seconds

Process took: 0.68 seconds
```

Asynchronous:

```
Out[11]:

<Future at 0x7ff7ad63cc10 state=pending>

Fetch async process 1 started

Fetch async process 2 started

Fetch async process 3 started

Process 2: Fri, 02 Apr 2021 05:40:53 GMT, took: 0.39 seconds

Process 1: Fri, 02 Apr 2021 05:40:53 GMT, took: 0.39 seconds

Process 3: Fri, 02 Apr 2021 05:40:53 GMT, took: 0.39 seconds

In [13]:

from collections import namedtuple

import time

import asyncio

import aiohttp
```

```
Service = namedtuple('Service', ('name', 'url', 'ip_attr'))

SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'query')
)



async def fetch_ip(service):
    start = time.time()
    async with aiohttp.ClientSession() as session:
        print('Fetching IP from {}'.format(service.name))
        async with session.get(service.url) as response:
            datetime = response.headers.get('Date')
            json_response = await response.json()
            ip = json_response[service.ip_attr]
            return '{} finished with result: {}, took: {:.2f} seconds'.format(
                service.name, ip, time.time() - start)

async def get_all():
    futures = [fetch_ip(service) for service in SERVICES]
    done, pending = await asyncio.wait(futures)
    for future in done:
        print(future.result())

ioloop = asyncio.get_event_loop()
wait_object = asyncio.wait(asyncio.create_task(get_all()))
asyncio.run_coroutine_threadsafe(wait_object, loop)
```

Out[13]:

<Future at 0x7ff7ad6503d0 state=pending>

Fetching IP from ip-api

Fetching IP from ipify

ipify finished with result: 37.214.62.45, took: 0.56 seconds

ip-api finished with result: 37.214.62.45, took: 0.12 seconds

Таймауты

In [14]:

```
from collections import namedtuple
import time
import asyncio
from concurrent.futures import FIRST_COMPLETED
import aiohttp


Service = namedtuple('Service', ('name', 'url', 'ip_attr'))


SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'query')
)



async def fetch_ip(service):
    start = time.time()
    async with aiohttp.ClientSession() as session:
        print('Fetching IP from {}'.format(service.name))
        async with session.get(service.url) as response:
            datetime = response.headers.get('Date')
```

```
            json_response = await response.json()
            ip = json_response[service.ip_attr]
            return '{} finished with result: {}, took: {:.2f} seconds'.format(
                service.name, ip, time.time() - start)

async def get_all(timeout):
    futures = [fetch_ip(service) for service in SERVICES]
    done, pending = await asyncio.wait(futures, timeout=timeout, return_when=FIRST_COMPLETED)
    for future in pending:
        future.cancel()
    for future in done:
        print(future.result())

ioloop = asyncio.get_event_loop()
wait_object = asyncio.wait(asyncio.create_task(get_all(0.05)))
asyncio.run_coroutine_threadsafe(wait_object, loop)

wait_object = asyncio.wait(asyncio.create_task(get_all(0.2)))
asyncio.run_coroutine_threadsafe(wait_object, loop)
Out[14]:
<Future at 0x7ff7ac5c4190 state=pending>
Fetching IP from ip-api
Fetching IP from ipify
Fetching IP from ip-api
Fetching IP from ipify
ip-api finished with result: 37.214.62.45, took: 0.10 seconds
```

Обработка ошибок

Чтиво про обработку ошибок:

https://www.roguelynn.com/words/asyncio-exception-handling/

Async IO дизайн паттерны

Цепочка корутин

Основная фича корутин, что они могут быть соединены друг с другом. Это достигается за счет того, что они могут ждать друг друга.

Это позволяет разбить программу на маленькие управляемые корутины

In [15]:

```
import asyncio

import random

import time


async def part1(n: int) -> str:

    i = random.randint(0, 10)

    print(f"part1({n}) sleeping for {i} seconds.")

    await asyncio.sleep(i)

    result = f"result{n}-1"

    print(f"Returning part1({n}) == {result}.")

    return result


async def part2(n: int, arg: str) -> str:

    i = random.randint(0, 10)

    print(f"part2{n, arg} sleeping for {i} seconds.")

    await asyncio.sleep(i)

    result = f"result{n}-2 derived from {arg}"

    print(f"Returning part2{n, arg} == {result}.")

    return result


async def chain(n: int) -> None:

    start = time.perf_counter()
```

```
    p1 = await part1(n)
    p2 = await part2(n, p1)
    end = time.perf_counter() - start
    print(f"-->Chained result{n} => {p2} (took {end:0.2f} seconds).")

async def main(*args):
    await asyncio.gather(*(chain(n) for n in args))

loop = asyncio.get_event_loop()
wait_object = asyncio.wait(asyncio.create_task(main(*[1, 2, 3])))
asyncio.run_coroutine_threadsafe(wait_object, loop)
```

Out[15]:

```
<Future at 0x7ff7ac5ab5b0 state=pending>
part1(1) sleeping for 8 seconds.
part1(2) sleeping for 6 seconds.
part1(3) sleeping for 6 seconds.
Returning part1(2) == result2-1.
part2(2, 'result2-1') sleeping for 3 seconds.
Returning part1(3) == result3-1.
part2(3, 'result3-1') sleeping for 10 seconds.
Returning part1(1) == result1-1.
part2(1, 'result1-1') sleeping for 1 seconds.
Returning part2(2, 'result2-1') == result2-2 derived from result2-1.
-->Chained result2 => result2-2 derived from result2-1 (took 9.00 seconds).
Returning part2(1, 'result1-1') == result1-2 derived from result1-1.
-->Chained result1 => result1-2 derived from result1-1 (took 9.00 seconds).
```

Очередь

In [ ]:

```
import asyncio
import itertools as it
import os
import random
import time

async def makeitem(size: int = 5) -> str:
    return os.urandom(size).hex()

async def randsleep(a: int = 1, b: int = 5, caller=None) -> None:
    i = random.randint(0, 10)
    if caller:
        print(f"{caller} sleeping for {i} seconds.")
    await asyncio.sleep(i)

async def produce(name: int, q: asyncio.Queue) -> None:
    n = random.randint(0, 10)
    for _ in it.repeat(None, n):  # Synchronous loop for each single producer
        await randsleep(caller=f"Producer {name}")
        i = await makeitem()
        t = time.perf_counter()
        await q.put((i, t))
        print(f"Producer {name} added <{i}> to queue.")

async def consume(name: int, q: asyncio.Queue) -> None:
    while True:
        await randsleep(caller=f"Consumer {name}")
        i, t = await q.get()
```

```
        now = time.perf_counter()
        print(f"Consumer {name} got element <{i}>"
              f" in {now-t:0.5f} seconds.")
        q.task_done()


async def main(nprod: int, ncon: int):
    q = asyncio.Queue()
    producers = [asyncio.create_task(produce(n, q)) for n in range(nprod)]
    consumers = [asyncio.create_task(consume(n, q)) for n in range(ncon)]
    await asyncio.gather(*producers)
    await q.join()  # Implicitly awaits consumers, too
    for c in consumers:
        c.cancel()



loop = asyncio.get_event_loop()
wait_object = asyncio.wait(asyncio.create_task(main(2, 4)))
asyncio.run_coroutine_threadsafe(wait_object, loop)
```

Чтиво

- https://realpython.com/async-io-python/#where-does-async-io-fit-in
- https://www.roguelynn.com/words/asyncio-exception-handling/

https://docs.python.org/3/library/asyncio-task.html#coroutines