Лекция 13: Многопоточность и GIL

Сергей Лебедев

sergei.a.lebedev@gmail.com

7 декабря 2015 г.

II минимум

- *Процесс* — запущенная программа.
- У каждого процесса есть **изолированное** от других процессов состояние:
  - виртуальное адресное пространство,
  - указатель на исполняемую инструкцию,
  - стек вызовов,
  - системные ресурсы, например, открытые файловые дескрипторы.
- Процессы удобны для одновременного выполнения нескольких задач.
- Альтернативный способ: делегировать каждую задачу на выполнение *потоку*.

- Поток похож на процесс тем, что его исполнение происходит независимо от других потоков (и процессов).
- В отличие от процесса поток исполняется **внутри** процесса и разделяет с ним адресное пространство и системные ресурсы.
- Потоки удобны для одновременного выполнения нескольких задач, которым требуется доступ к разделяемому состоянию.
- Совместным выполнением нескольких процессов и потоков управляет операционная система, поочерёдно разрешая каждому процессу или потоку использовать сколько-то циклов процессора.

Модуль `threading`

- Поток в Python — это системный поток, то есть его выполнением управляет не интерпретатор, а операционная система.
- Создать поток можно с помощью класса `Thread` из модуля стандартной библиотеки `threading`.
- Пример[1]:

```
>>> import time
>>> def countdown(n):
...     for i in range(n):
...         print(n - i - 1, "left")
...         time.sleep(1)
...
>>> t = Thread(target=countdown, args=(3, ))
>>> t.start()
2 left
>>> 1 left
0 left
```

[1] Из книги D. Beazley & K. Jones «Python Cookbook», 3rd edtition.

- Альтернативный способ создания потока — наследование:

```
>>> class CountdownThread(Thread):
...     def __init__(self, n):
...         super().__init__()
...         self.n = n
...
...     def run(self):  # вызывается методом start.
...         for i in range(self.n):
...             print(self.n - i - 1, "left")
...             time.sleep(1)
...
>>> t = CountdownThread(3)
>>> t.start()
2 left
>>> 1 left
0 left
```

- Минус такого подхода в том, что он ограничивает переиспользование кода: функциональность класса CountdownThread можно использовать только в отдельном потоке.

- При создании потоку можно указать имя. По умолчанию оно `"Thread-N"`:

```
>>> Thread().name
'Thread-1'
>>> Thread(name="NumberCruncher").name
'NumberCruncher'
```

- У каждого активного потока есть идентификатор — неотрицательное число, уникальное для всех активных потоков.

```
>>> t = Thread()
>>> t.start()
>>> t.ident
4350545920
```

- Метод `join` позволяет дождаться завершения потока.
  - Выполнение вызывающего потока приостановится, пока не завершится поток `t`.
  - Повторные вызовы метода `join` не имеют эффекта.
  - Пример:

```
>>> t = Thread(target=time.sleep, args=(5, ))
>>> t.start()
>>> t.join()     # блокируется на 5 секунд
>>> t.join()     # выполняется моментально
```

- Проверить, выполняется ли поток, можно с помощью метода `is_alive`:

```
>>> t = Thread(target=time.sleep, args=(5, ))
>>> t.start()
>>> t.is_alive()
True
>>> t.is_alive()  # через 5 секунд
False
```

- Демон — это поток, созданный с аргументом `daemon=True`:

```
>>> t = Thread(target=time.sleep, args=(5, ),
...                daemon=True)
>>> t.start()
>>> t.daemon
True
```

- Отличие потока-демона от обычного потока в том, потоки-демоны **автоматически** уничтожаются при выходе из интерпретатора.
- Уничтожение потока-демона не подразумевает процедуру финализации, поэтому следует быть аккуратным при использовании демонов для задач, работающих с ресурсами.

- В Python нет встроенного механизма завершения потоков — это не случайность, а осознанное решение разработчиков языка.
- Корректное завершение потока часто связано с освобождением ресурсов, например:
  - поток может работать с файлом, дескриптор которого нужно закрыть,
  - или захватить примитив синхронизации.
- Для завершения потока обычно используют флаг:

```
class Task:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running:
            # ...
```

Набор примитивов синхронизации в модуле `threading` стандартный:

- `Lock` — обычный мьютекс, используется для обеспечения эксклюзивного доступа к разделяемому состоянию.
- `RLock` — рекурсивный мьютекс, разрешающий потоку, *владеющему* мьютексом, захватить мьютекс более одного раза.
- `Semaphore` — вариация мьютекса, которая разрешает захватить себя не более фиксированного числа раз.
- `BoundedSemaphore` — семафор, который следит за тем, что его захватили и отпустили одинаковое количество раз.

Потокобезопасный ~~и медленный~~ счётчик

- Все примитивы синхронизации реализуют единый интерфейс:
  - метод `acquire` захватывает примитив синхронизации,
  - а метод `release` отпускает его.
- Пример:

```python
class SharedCounter:
    def __init__(self, value):
      self._value = value
      self._lock = Lock()

    def increment(self, delta=1):
        self._lock.acquire()
        self._value += delta
        self._lock.release()

    def get(self):
        return self._value
```

- Все мьютексы и семафоры в модуле `threading` реализованы “с нуля” в терминах примитивного бинарного семафора[2]

```
typedef struct {
    char      locked;
    cond_t    lock_released;
    mutex_t   mut;
} lock_t;
```

- Мьютекс `mut` используется **только** для синхронизации доступа к полю `locked`.
- Забавное следствие: для мьютекса в Python не определено понятие *владеющего* потока, то есть поток может отпустить мьютекс, не захваченный им.

[2] http://bit.ly/cpython-thread

```
>>> done = Lock()
>>> def idle_release():
...     print("Running!")
...     time.sleep(15)
...     done.release()
...
>>> done.acquire()
True
>>> Thread(target=idle_release).start()
Running!
>>> done.acquire() and print("WAT?")
WAT?  # через 15 секунд.
```

[3] http://bit.ly/beazley-synchronization

Примитивы синхронизации: условные переменные

- `Condition` используется для отправки сигналов между потоками.
- Метод `wait` блокирует вызывающий поток, пока какой-то другой поток не вызовет метод `notify` или `notify_all`.

```
q = deque()
is_empty = Condition()

def producer():
    while True:
        is_empty.acquire()
        q.append(...)
        is_empty.notify()
        is_empty.release()
```

```
def consumer():
    while True:
        is_empty.acquire()
        while not q:  # !
            is_empty.wait()
        ... = q.popleft()
        is_empty.release()
```

- Поток может разблокироваться даже если метод `notify` не был вызван. Такое поведение называют *spurious wakeup*.

Функция `follow` читает сообщения из переданного ей соединения и кладёт их в очередь на обработку.

```
def follow(connection, connection_lock, q):
    try:
        while True:
            connection_lock.acquire()
            message = connection.read_message()
            connection_lock.release()
            q.put(message)
    except InvalidMessage:
        follow(connection, connection_lock, q)

follower = Thread(target=follow, args=...)
follower.start()
```

Вопрос

Что может пойти не так?

Чтобы минимизировать ошибки при использовании методов `acquire` `release`, все примитивы синхронизации поддерживают протокол менеджеров контекста.

```
def follow(connection, connection_lock, q):
    try:
        while True:
            with connection_lock:
                message = connection.read_message()
            q.put(message)
    except IOError:
        follow(connection, connection_lock, q)
```

Модуль queue

- Модуль `queue` реализует несколько потокобезопасных очередей:
  - `Queue` — FIFO очередь,
  - `LifoQueue` — LIFO очередь *aka* стек,
  - `PriorityQueue` — очередь, элементы которой — пары вида `(priority, item)`.
- Никаких особых изысков в реализации очередей нет: все методы, изменяющие состояние, работают “внутри” мьютекса.
- Класс `Queue` использует в качестве контейнера `deque`, а классы `LifoQueue` и `PriorityQueue` — список.

```python
def worker(q):
    while True:
        item = q.get()        # блокирующе ожидает следующий
        do_something(item)  #     элемент
        q.task_done()         # уведомляет очередь о выполнении
                              #     задания


def master(q):
    for item in source():
        q.put(item)

    # блокирующе ожидает, пока все элементы очереди
    # не будут обработаны
    q.join()
```

Модуль futures

- Модуль `concurrent.futures` содержит абстрактный класс `Executor` и его реализацию в виде пула потоков — `ThreadPoolExecutor`.
- Интерфейс исполнителя состоит всего из трёх методов:

```
>>> executor = ThreadPoolExecutor(max_workers=4)
>>> executor.submit(print, "Hello, world!")
Hello, world!
<Future at 0x102991278 state=running>
>>> list(executor.map(print, ["Knock?", "Knock!"]))
Knock?
Knock!
[None, None]
>>> executor.shutdown()
```

- Исполнители поддерживают протокол менеджеров контекста:

```
>>> with ThreadPoolExecutor(max_workers=4) as executor:
...     # ...
...
```

- Метод `Executor.submit` возвращает экземпляр класса `Future`, инкапсулирующего асинхронные вычисления.
- Что можно делать с `Future`?

```
>>> with ThreadPoolExecutor(max_workers=4) as executor:
...     f = executor.submit(sorted, [4, 3, 1, 2])
...
```

- Поинтересоваться состоянием вычисления:

```
>>> f.running(), f.done(), f.cancelled()
(False, True, False)
```

- Блокирующе подождать результата вычисления:

```
>>> print(f.result())
[1, 2, 3, 4]
>>> print(f.exception())
None
```

- Добавить функцию, которая будет вызвана после завершения вычисления:

```
>>> f.add_done_callback(print)
<Future at 0x102edaac8 state=finished returned list>
```

Пример использования модуля futures: integrate

```
>>> import math
>>> def integrate(f, a, b, *, n_iter=1000):
...     acc = 0
...     step = (b - a) / n_iter
...     for i in range(n_iter):
...         acc += f(a + i * step) * step
...     return acc
...
>>> integrate(math.cos, 0, math.pi / 2)
1.0007851925466296
>>> integrate(math.sin, 0, math.pi)
1.9999983550656637
```

Пример использования модуля futures: integrate_async

```
>>> from functools import partial
>>> def integrate_async(f, a, b, *, n_jobs, n_iter=1000):
...     executor = ThreadPoolExecutor(max_workers=n_jobs)
...     spawn = partial(executor.submit, integrate, f,
...                     n_iter=n_iter // n_jobs)
...
...     step = (b - a) / n_jobs
...     fs = [spawn(a + i * step, a + (i + 1) * step)
...           for i in range(n_jobs)]
...     return sum(f.result() for f in as_completed(fs))
...
>>> integrate_async(math.cos, 0, math.pi / 2, n_jobs=2)
1.0003926476775074
>>> integrate_async(math.sin, 0, math.pi, n_jobs=2)
1.999995887664657
```

Инфраструктура для многопоточного программирования в Python

- Модули `threading`, `queue` и `concurrent.futures` реализуют привычные инструменты для || программирования на Python.
- Мы обсудили:
  - потоки,
  - мьютексы и семафоры,
  - события и условные переменные,
  - очереди,
  - пулы потоков.

Параллелизм и конкурентность

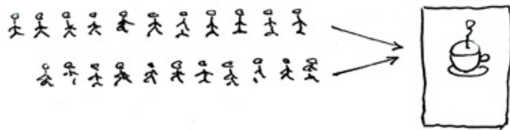
Сравним производительность последовательной и параллельной версий функции `integrate` с помощью “магической” команды `timeit`:

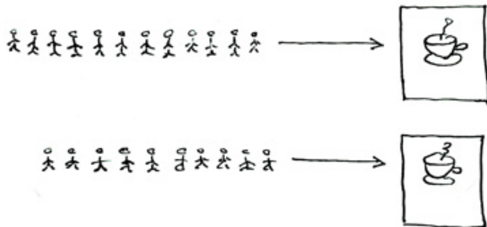
```
In [1]: %%timeit -n100
   ...: integrate(math.cos, 0, math.pi / 2,
   ...:           n_iter=10**6)
   ...:
100 loops, best of 3: 279 ms per loop

In [2]: %%timeit -n100
   ...: integrate_async(math.cos, 0, math.pi / 2,
   ...:                 n_iter=10**6,
   ...:                 n_jobs=2)
100 loops, best of 3: 283 ms per loop

In [3]: %%timeit -n100
   ...: integrate_async(math.cos, 0, math.pi / 2,
   ...:                 n_iter=10**6,
   ...:                 n_jobs=4)
100 loops, best of 3: 275 ms per loop
```

- GIL (global interpreter lock) — это мьютекс, который гарантирует, что в каждый момент времени только один поток имеет доступ к внутреннему состоянию интерпретатора.
- Python C API позволяет отпустить GIL, но это безопасно только при работе с объектами, не зависящими от интерпретатора Python.
- Например, все операции ввода/вывода в CPython отпускают GIL[4]:

```
// ...
Py_BEGIN_ALLOW_THREADS
err = close(fd);
if (err < 0)
    save_errno = errno;
Py_END_ALLOW_THREADS
// ...
```

[4] http://bit.ly/cpython-fileio


Concurrent = Two Queues One Coffee Machine


Parallel = Two Queues Two Coffee Machines
© Joe Armstrong 2013

- Ответ зависит от задачи.
- Наличие GIL делает невозможным использование потоков в Python для параллелизма: несколько потоков не ускоряют, а иногда даже замедляют работу программы.
- GIL не мешает использовать потоки для конкурентности при работе с вводом/выводом, например:

```
>>> from urllib.request import urlretrieve
>>> with ThreadPoolExecutor(max_workers=4) as executor:
...     with open("urls.txt", "w") as handle:
...         for url in handle:
...             executor.submit(urlretrieve, url)
...
```

- Альтернативный подход к организации конкурентной работы с вводом/выводом основан на использовании паттернов *реактор* и *проактор*.

```
import asyncio

async def echo(source, target):
    while True:
        line = await source.readline()  # ->
        if not line:
            break
        target.write(line)

loop = asyncio.get_event_loop()
server = asyncio.start_server(echo, port=8080)
loop.create_task(server)

try:
    loop.run_forever()
finally:
    server.close()
    loop.close()
```

[5]Модуль asyncio появился сравнительно недавно. Его вдохновители вне стандартной библиотеки Python: twisted, tornado, gevent.

```
In [2]: %%cython
   ...: from libc.math cimport cos
   ...: def integrate(f, double a, double b, long n_iter):
   ...:     #           ^ мы используем C-версию функции
   ...:     cdef double acc = 0
   ...:     cdef double step = (b - a) / n_iter
   ...:     cdef long i
   ...:     with nogil:
   ...:         for i in range(n_iter):
   ...:             acc += cos(a + i * step) * step
   ...:     return acc

In [3]: %%timeit -n100
   ...: integrate_async(math.cos, 0, math.pi / 2,
   ...:                 n_iter=10**6, n_jobs=2)
100 loops, best of 3: 9.58 ms per loop

In [4]: %%timeit -n100
   ...: integrate_async(math.cos, 0, math.pi / 2,
   ...:                 n_iter=10**6, n_jobs=4)
100 loops, best of 3: 7.95 ms per loop
```

Модуль `multiprocessing`

- Можно использовать вместо потоков процессы.
- У каждого процесса будет свой GIL, но он не помешает им работать параллельно.
- За работу с процессами в Python отвечает модуль `multiprocessing`:

```
>>> import multiprocessing as mp
>>> p = mp.Process(target=countdown, args=(5, ))
>>> p.start()
>>> 4 left
3 left
2 left
1 left
0 left
>>> p.name, p.pid
('Process-2', 65130)
>>> p.daemon
False
>>> p.join()
>>> p.exitcode
0
```

- Модуль реализует базовые примитивы синхронизации: мьютексы, семафоры, условные переменные.
- Для организации взаимодействия между процессами можно использовать `Pipe` — основанное на сокете соединение между двумя процессами:

```
>>> def ponger(conn):
...     conn.send("pong")
...
>>> parent_conn, child_conn = mp.Pipe()
>>> p = mp.Process(target=ponger,
...                args=(child_conn, ))
>>> p.start()
>>> parent_conn.recv()
'pong'
>>> p.join()
```

- Альтернативно два и более процессов можно соединить через очередь `Queue` или `JoinableQueue` — аналоги потокобезопасных очередей из модуля `queue`.

Реализация функции `integrate_async` на основе пула потоков работала 275 мс, попробуем использовать пул процессов:

```
In [1]: from concurrent.futures import ProcessPoolExecutor

In [2]: def integrate_async(f, a, b, *, n_jobs, n_iter=1000):
   ...:     executor = ProcessPoolExecutor(
   ...:         max_workers=n_jobs)
   ...:     spawn = partial(executor.submit, integrate, f,
   ...:                     n_iter=n_iter // n_jobs)
   ...:
   ...:     step = (b - a) / n_jobs
   ...:     fs = [spawn(a + i * step, a + (i + 1) * step)
   ...:           for i in range(n_jobs)]
   ...:     return sum(f.result() for f in as_completed(fs))
   ...:

In [3]: %%timeit -n100
   ...: integrate_async(math.cos, 0, math.pi / 2,
   ...:                 n_iter=10**6, n_jobs=4)
100 loops, best of 3: 154 ms per loop
```

- Пакет `joblib` реализует параллельный аналог цикла **for**, который удобно использовать для параллельного выполнения независимых задач.

```
from joblib import Parallel, delayed

def integrate_async(f, a, b, *, n_jobs, n_iter=1000,
                    backend=None):
    step = (b - a) / n_jobs
    with Parallel(n_jobs=n_jobs,
                  backend=backend) as parallel:
        fs = (delayed(integrate)(a + i * step,
                                 a + (i + 1) * step,
                                 n_iter=n_iter // n_jobs)
              for i in range(n_jobs))
        return sum(parallel(fs))
```

- В качестве значения аргумента `backend` можно указать `"threading"` или `"multiprocessing"`.

- GIL — это глобальный мьютекс, который ограничивает возможности использования потоков для параллелизма в программах на CPython.
- Для программ, использующих, в основном, операции ввода/вывода, GIL не страшен: в CPython эти операции отпускают GIL.
- Для программ, нуждающихся в параллелизме, для повышения производительности есть варианты:
  - писать критическую функциональность на C или Cython или
  - использовать модуль `multiprocessing`.