Урок 8

Потоки

Введение в потоки. Введение в многопоточное программирование. Модуль threading. Примитивы синхронизации. Модуль Queue. Модуль multiprocessing.

На этом уроке рассмотрим модули и приемы программирования, используемые при разработке многопоточных приложений на Python. Изучим потоки управления, обмен сообщениями, многопоточную обработку данных. Чтобы приступить к библиотечным модулям, познакомимся с основными понятиями.

Основные понятия

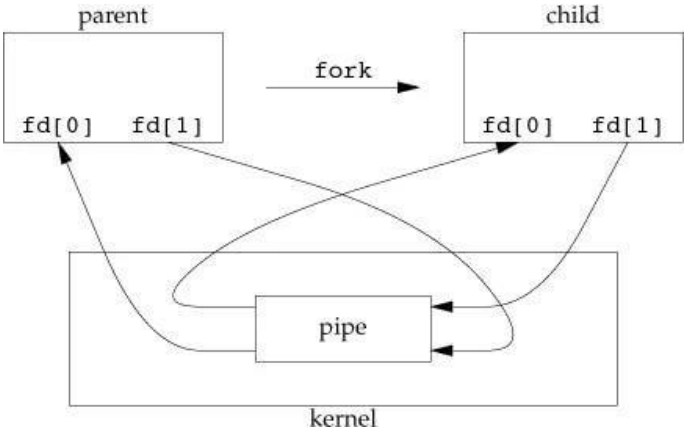
Выполняющаяся программа называется процессом. Он обладает параметрами, характеризующими его состояние:

- объем занимаемой памяти;
- список открытых файлов;
- программный счетчик, который ссылается на очередную выполняемую инструкцию;
- стек вызовов, используемый для хранения локальных переменных функций.

Обычно процесс выполняет инструкции одну за другой в единственном потоке управления. Его иногда называют главным потоком процесса. В каждый конкретный момент программа делает что-то одно.

Программа может создавать новые процессы с помощью библиотечных функций — таких, как представленные в модуле **os** или **subprocess** (например, **os.fork()**, **subprocess.Popen()** и другие). Такие процессы называют дочерними. Они выполняются независимо, и каждый из них имеет собственные характеристики и главный поток управления. Родительский процесс может выполнять свои операции, а его дочерние будут в фоне делать свою работу.

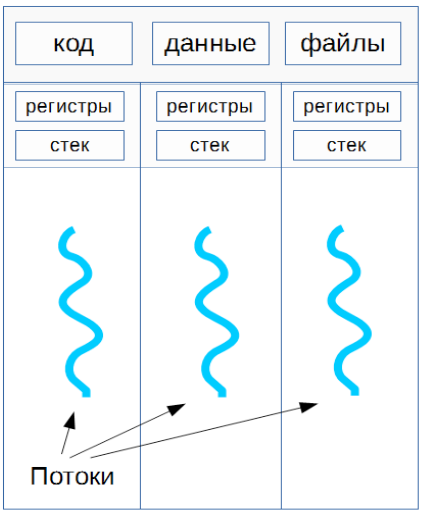
Процессы изолированы друг от друга, но могут обмениваться информацией, используя механизмы взаимодействия (**Interprocess Communication, IPC**). Одна из распространенных форм — обмен сообщениями (простыми буферами двоичных байтов). Чтобы принимать и передавать сообщения через неименованные каналы или сетевые соединения, используются простейшие операции — **send()** и **recv()**. Реже применяется механизм взаимодействий, основанный на отображаемых областях памяти (смотрите описание модуля **mmap**). Благодаря этой возможности процессы могут создавать разделяемые области памяти. Изменения в них будут доступны необходимым процессам.



Приложения могут создавать дочерние процессы для нескольких задач, и каждый будет отвечать за свою часть работы. Но есть и другой подход к разделению работы на задачи — он основан на использовании нескольких потоков управления. **Поток управления** напоминает процесс тем, что он выполняет собственную последовательность инструкций и имеет свой стек вызовов. Разница в том, что потоки выполняются в пределах процесса, создавшего их, и совместно используют данные и

системные ресурсы, выделенные процессу. Потоки удобно использовать, когда в приложении необходимо одновременно выполнять несколько задач, но при этом значительный объем информации должен быть доступен всем потокам. Один может обрабатывать файлы перед отправкой на сервер, другой в это время — обеспечивать взаимодействие с пользователем.

Однопоточный процесс

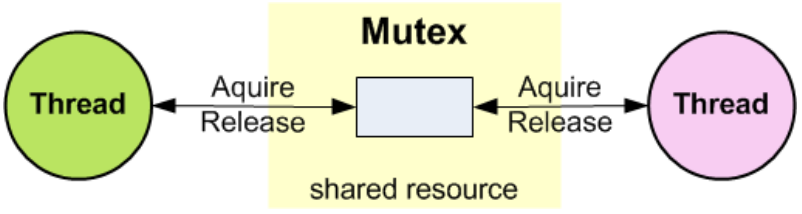


Многопоточный процесс

Когда одновременно выполняется несколько процессов или потоков управления, за распределение процессорного времени между ними отвечает операционная система. Она выделяет каждому процессу (или потоку) небольшой квант времени и быстро переключается между активными задачами, отдавая каждой из них определенное количество тактов процессора. Если в системе 10 активных процессов, ОС будет выделять каждому из них примерно 1/10 процессорного времени и быстро переключаться между ними. В системах, где более одного ядра процессора, операционная система сможет планировать работу так, чтобы все ядра оказались заняты примерно поровну параллельным выполнением процессов.

Основная сложность заключается в синхронизации и обеспечении доступа к совместно используемым данным. Попытка изменить данные одновременно из нескольких потоков может привести к их повреждению и к нарушению целостности состояния программы. Формально эта проблема известна как гонка за ресурсами.

Чтобы снять ее, в многопоточных программах выделяют **критические участки** программного кода и обеспечивают их выполнение под защитой взаимоисключающих **блокировок** (*мьютексов*) или похожих механизмов синхронизации. Если в разных потоках одновременно появится необходимость выполнить запись в один и тот же файл, для синхронизации их действий можно воспользоваться взаимоисключающей блокировкой. Пока один из потоков выполняет запись, остальные ждут завершения операции и только после него получают доступ к файлу.



Реализация синхронизации подобного рода обычно выглядит так:

```
write_lock = Lock()
...
# Критический участок, где выполняется запись
write_lock.acquire()
f.write("Here's some data.\n")
f.write("Here's more data.\n")
...
write_lock.release()
```

Есть шутка, которую приписывают Джейсону Уиттингтону (Jason Whittington): *«Зачем многопоточный цыпленок пересекает дорогу? Чтобы другую сторону перейти»*. Она отражает типичные проблемы, связанные с синхронизацией и многопоточным программированием. Но на практике подходы становятся очевидными и перестают вызывать недоумение.

Параллельное программирование в Python

Python поддерживает механизм обмена сообщениями и позволяет создавать многопоточные программы в большинстве систем. Многие программисты знакомы с интерфейсом потоков, но не все знают, что потоки управления в Python имеют существенные ограничения. Несмотря на минимальную поддержку многопоточных приложений, интерпретатор Python использует внутреннюю глобальную блокировку интерпретатора (Global Interpreter Lock, GIL). Из-за этого в каждый конкретный момент времени может работать только один поток. Вследствие этого программы на Python могут выполняться только на одном процессоре, независимо от их количества в системе. Глобальная блокировка GIL часто становится причиной жарких дебатов в сообществе Python, но вряд ли ее устранят в обозримом будущем.

Блокировка GIL делает неэффективной реализацию в виде многопоточного приложения. Наличие дополнительных процессоров практически не дает преимуществ программе, которая большую часть времени проводит в ожидании событий. С другой стороны, разделение приложения, выполняющего объемные вычисления, на несколько потоков не только не дает преимуществ, но и снижает производительность программы. Это снижение окажется намного существеннее, чем вы могли бы предположить. Подобные программы лучше реализовывать в виде нескольких процессов и обеспечивать взаимодействие между ними, привлекая механизм обмена сообщениями.

Даже программисты, которые используют механизм потоков, зачастую находят странным их поведение при масштабировании. Многопоточный сетевой сервер может иметь прекрасную производительность при одновременной работе 100 потоков — и ужасную, когда их число увеличивается до 10 000. Вообще не стоит писать программы, запускающие 10 000 потоков, потому что каждый потребляет ресурсы системы и увеличивает нагрузку. Она связана с переключением контекста потоков, установкой блокировок и использованием других ресурсоемких механизмов. При этом все потоки вынуждены выполняться на единственном процессоре.

Подобные проблемы часто решают, реструктуризируя приложения и используя системы асинхронной обработки событий. Например, главный цикл событий может просматривать все каналы ввода–вывода при помощи модуля **select** и передавать асинхронные события большой коллекции

обработчиков ввода–вывода. Такой подход лег в основу ряда библиотечных модулей: **asyncio** и других, — а также популярных сторонних модулей: **Twisted**, **Tornado** и подобных.

Приступая к разработке многопоточных приложений на Python, в первую очередь необходимо выбрать механизм обмена сообщениями. При работе с потоками рекомендуется организовать приложение как коллекцию независимых потоков, обменивающихся данными с помощью очередей сообщений. Такой подход наименее подвержен ошибкам. При нем меньше требуются блокировки и другие механизмы синхронизации. Обмен сообщениями легко и естественно распространяется на сетевые взаимодействия и распределенные системы. Если есть часть программы, которая выполняется в виде отдельного потока и принимает сообщения, ее можно оформить в виде отдельного процесса или перенести на другой компьютер и посылать ей сообщения через сетевое соединение.

Модуль threading

Модуль **threading** содержит определение класса **Thread** и реализацию механизмов синхронизации, используемых в многопоточных программах.

Объекты класса Thread

Класс **Thread** используется для представления отдельного потока управления. Новый поток можно создать вызовом конструктора:

- **Thread(group=None, target=None, name=None, args=(), kwargs={})** — создает новый экземпляр класса **Thread**. Аргумент **group** всегда получает значение **None** и зарезервирован для использования в будущем. В аргументе **target** передается объект, который вызывается методом **run()** при запуске потока. По умолчанию задан как **None** — значит ничего вызываться не будет. Аргумент **name** определяет имя потока. По умолчанию генерируется уникальное имя вида **Thread-N**. В **args** передается кортеж позиционных аргументов для функции **target**, а в **kwargs** — словарь именованных аргументов для нее же.

Экземпляр **t** класса **Thread** поддерживает следующие методы и атрибуты:

- **t.start()** — запускает поток вызовом метода **run()** в отдельном потоке управления. Может вызываться только один раз;
- **t.run()** — вызывается при запуске потока. По умолчанию вызывает функцию **target**, которая была передана конструктору. Можно создать свой класс, производный от **Thread**, и определить в нем собственную реализацию метода **run()**;
- **t.join([timeout])** — ожидает завершения потока или истечения указанного интервала времени. Аргумент **timeout** определяет максимальный период ожидания в секундах — в виде числа с плавающей точкой. Поток не может присоединяться к самому себе. Ошибка — пытаться присоединиться к потоку до того, как он будет запущен;
- **t.is_alive()** — возвращает **True**, если поток **t** продолжает работу, и **False** в противном случае. Поток считается действующим от момента вызова метода **start()** до того, как завершится **run()**;
- **t.name** — имя потока. Этот атрибут используется только для идентификации и может принимать любые значения (желательно осмысленные, чтобы упростить отладку);
- **t.ident** — целочисленный идентификатор потока. Если поток еще не был запущен, этот атрибут содержит значение **None**;

- **t.daemon** — логический флаг, указывающий, будет ли поток демоническим. Значение этого атрибута должно устанавливаться до вызова метода **start()**. По умолчанию он получает значение, унаследованное от потока, создавшего его. Программа Python завершается, когда не осталось ни одного активного, недемонического потока управления. Любая программа имеет главный поток, представляющий первоначальный поток управления, который не является демоническим.

Рассмотрим пример того, как создавать и запускать функции (или другие вызываемые объекты) в отдельных потоках управления (файл **examples/01_thread/ 01_thread_simple.py**):

```
from threading import Thread
import time

def clock(interval):
    while True:
        print("Текущее время: %s" % time.ctime())
        time.sleep(interval)

t = Thread(target=clock, args=(15, ))
t.daemon = True
t.start()
```

И как определить тот же поток в виде класса:

```
from threading import Thread
import time

class ClockThread(Thread):
    def __init__(self, interval):
        super().__init__()
        self.daemon = True
        self.interval = interval

    def run(self):
        while True:
            print("Текущее время: %s" % time.ctime())
            time.sleep(self.interval)

t = ClockThread(15)
t.start()
```

Когда объявляется собственный класс потока, в котором переопределяется метод **__init__()**, важно вызвать конструктор базового класса **super().__init__()**, как показано в примере. Если этого не сделать, возникнет ошибка. Неверным будет и пытаться переопределить другие методы класса **Thread**, кроме **run()** и **__init__()**.

Настройка атрибута **daemon** в этих примерах является характерной операцией при работе с потоками, которые выполняют бесконечный цикл. Обычно интерпретатор Python ожидает завершения всех потоков, прежде чем завершиться самому. Но если есть никогда не завершающиеся фоновые потоки, такое поведение нежелательно. Значение **True** в атрибуте **daemon** позволяет интерпретатору завершиться сразу после выхода из главной программы. В этом случае демонические потоки просто уничтожаются.

Чтобы проследить работу потока после вызова метода **t.start()**, необходимо также добавить вызов метода **t.join()**. Иначе приложение завершится сразу после запуска потока.

Объекты класса Timer

Объекты класса **Timer** используются для вызова функций через определенное время.

- **Timer(interval, func [, args [, kwargs]])** — создает объект таймера, который вызывает функцию **func** через **interval** секунд. В аргументах **args** и **kwargs** передаются позиционные и именованные аргументы для функции **func**. Таймер не запускается, пока не будет вызван метод **start()**.

Экземпляр **t** класса **Timer** обладает следующими методами:

- **t.start()** — запускает таймер. Функция **func**, переданная конструктору **Timer()**, будет вызвана спустя указанное количество секунд после вызова этого метода;
- **t.cancel()** — останавливает таймер, если функция еще не была вызвана.

Объекты класса Lock

Простейшая блокировка (или взаимоисключающая) — это механизм синхронизации, имеющий два состояния: «закрыто» и «открыто». Чтобы изменить состояние блокировки, используются методы **acquire()** и **release()**. Если блокировка находится в состоянии «закрыто», любая попытка приобрести ее будет заблокирована до момента, пока она не будет освобождена. Если сразу несколько потоков управления пытаются приобрести блокировку, только один из них сможет продолжить работу, когда она освободится. Порядок продолжения работы заранее не определен.

Новый экземпляр класса **Lock** создается с помощью конструктора:

- **Lock()** — создает новый экземпляр блокировки, изначально находится в состоянии «открыто».

Экземпляр **lock** класса **Lock** поддерживает следующие методы:

- **lock.acquire([blocking])** — приобретает блокировку. Если она находится в состоянии «закрыто», этот метод приостанавливает работу потока, пока блокировка не будет освобождена. Если в аргументе **blocking** передать значение **False**, метод тут же вернет это значение, если блокировка не может быть приобретена, и **True** — если ее удалось приобрести;
- **lock.release()** — освобождает блокировку. Будет ошибкой пытаться вызвать этот метод, когда блокировка находится в состоянии «открыто» **или из другого потока, не из того, где вызывался метод** acquire().

Блокировка (мьютекс) реализуется на уровне языка Python (не C) и не хранит информацию о потоке, который захватил ее. В связи с этим возможна ситуация, когда один поток захватывает мьютекс, а другой освобождает его же (файл **examples/01_thread/02_thread_lock_hack.py**):

```
from threading import Thread, Lock
import time


 done = Lock()
```

```
def idle_release():
    print("Running release!")
    time.sleep(5)
    done.release()

done.acquire()
Thread(target=idle_release).start()
done.acquire()
print("Странное поведение мьютексов в Python...")
```

Пример использования блокировок — в файле **examples/01_thread/03_thread_lock.py**.

Объекты класса RLock

Реентерабельная блокировка (рекурсивная) — это механизм синхронизации, представляющий блокировку, которую в одном и том же потоке можно приобрести множество раз. Эта особенность позволяет потоку, владеющему блокировкой, выполнять вложенные операции **acquire()** и **release()**. В подобных ситуациях только самый внешний вызов метода **release()** действительно переведет блокировку в состояние «открыто».

Новый экземпляр класса **RLock** создается с помощью конструктора:

- **RLock()** — создает новый экземпляр реентерабельной блокировки.

Экземпляр **rlock** класса **RLock** поддерживает следующие методы:

- **rlock.acquire([blocking])** — приобретает блокировку. При необходимости работа потока приостанавливается, пока она не будет освобождена. Если перед вызовом метода блокировкой не владел ни один поток, она запирается, а уровень ее рекурсии устанавливается в значение 1. Если вызывающий поток уже владеет блокировкой, уровень рекурсии увеличивается на единицу, и метод тут же возвращает управление;

- **rlock.release()** — уменьшает уровень рекурсии. Если значение уровня рекурсии достигло нуля, блокировка переводится в состояние «открыто». В ином случае она остается закрытой. Эта функция должна вызываться только из потока, который владеет блокировкой.

Семафоры и ограниченные семафоры

Семафор — это механизм синхронизации, основанный на счетчике, который уменьшается при каждом вызове метода **acquire()** и увеличивается при каждом вызове **release()**. Если счетчик семафора достигает нуля, метод **acquire()** приостанавливает работу потока, пока другой поток не вызовет **release()**.

- **Semaphore([value])** — создает новый семафор. Аргумент **value** определяет начальное значение счетчика. При вызове без аргументов счетчик получает значение 1.

Экземпляр **s** класса **Semaphore** поддерживает следующие методы:

- **s.acquire([blocking])** — приобретает семафор. Если внутренний счетчик имеет значение больше нуля, этот метод уменьшает его на 1 и тут же возвращает управление. Если значение счетчика равно нулю, этот метод приостанавливает работу потока, пока другой поток не

вызовет метод **release()**. Аргумент **blocking** имеет тот же смысл, что и в методе **acquire()** экземпляров классов **Lock** и **RLock**.

- **s.release()** — увеличивает внутренний счетчик семафора на 1. Если перед вызовом метода счетчик был равен нулю и имеется другой поток, ожидающий освобождения семафора, этот поток возобновляет работу. Если сразу несколько потоков управления пытаются приобрести семафор, только в одном из них метод **acquire()** вернет управление. Порядок, в каком потоки смогут продолжить работу, заранее не определен.

- **BoundedSemaphore([value])** — создает новый семафор. Аргумент **value** определяет начальное значение счетчика. При вызове без аргументов счетчик получает значение 1. Ограниченный семафор **BoundedSemaphore** действует так же, как и обычный **Semaphore**, но количество вызовов метода **release()** не может превышать число вызовов **acquire()**.

Тонкое отличие семафоров от взаимоисключающих блокировок состоит в том, что семафоры могут использоваться в качестве сигналов. Методы **acquire()** и **release()** могут вызываться из разных потоков управления и обеспечивать взаимодействие между потоками поставщика и потребителя:

```
produced = threading.Semaphore(0)
consumed = threading.Semaphore(1)

def producer():
    while True:
        consumed.acquire()
        produce_item()
        produced.release()

def consumer():
    while True:
        produced.acquire()
        item = get_item()
        consumed.release()
```

Такой способ обмена сигналами можно реализовать с помощью переменных состояния.

Пример использования семафоров представлен в файле **examples/01_thread/04_thread_sem.py**.

Переменные состояния

Переменная состояния (**condition variable**) — это механизм синхронизации, надстроенный на уже имеющейся блокировке. Он используется потоками, когда требуется дождаться наступления определенного состояния или события. Переменные состояния обычно применяются в схемах «поставщик–потребитель», когда один поток производит данные, а другой обрабатывает их. Новый экземпляр класса **Condition** создается с помощью конструктора:

- **Condition([lock])** — создает новую переменную состояния. В необязательном аргументе **lock** передается экземпляр класса **Lock** или **RLock**. При вызове без аргумента для использования совместно с переменной состояния создается новый экземпляр класса **RLock**.

Экземпляр **cv** класса **Condition** поддерживает следующие методы:

- **cv.acquire(*args)** — приобретает блокировку, связанную с переменной состояния. Вызывает метод блокировки **acquire(*args)** и возвращает результат;

- **cv.release()** — освобождает блокировку, связанную с переменной состояния. Вызывает метод **release()** блокировки;

- **cv.wait([timeout])** — ожидает, пока будет получено извещение или пока истечет время ожидания. Этот метод должен вызываться только после того, как вызывающий поток приобретет блокировку. При этом блокировка освобождается, а поток приостанавливается, пока другим потоком не будет вызван метод **notify()** или **notifyAll()** переменной состояния. После возобновления метод тут же повторно приобретает блокировку и возвращает управление вызывающему потоку. В аргументе **timeout** передается число с плавающей точкой, определяющее предельное время ожидания в секундах. По истечении указанного интервала блокировка снова приобретается, и поток возобновляет работу;

- **cv.notify([n])** — возобновляет работу одного или более потоков, ожидающих изменения переменной состояния. Этот метод должен вызываться только после того, как поток приобретет блокировку. Ничего не делает, если отсутствуют потоки, ожидающие изменения этой переменной состояния. Аргумент **n** определяет количество потоков, которые смогут возобновить работу, и по умолчанию получает значение 1. Метод **wait()** не возвращает управление после возобновления потока, пока не сможет повторно приобрести блокировку;

- **cv.notify_all()** — возобновляет работу всех потоков, ожидающих изменения переменной состояния.

Рассмотрим пример применения переменной состояния. Его можно использовать как заготовку:

```
cv = threading.Condition()

def producer():
    while True:
        cv.acquire()
        produce_item()
        cv.notify()
        cv.release()

def consumer():
    while True:
        cv.acquire()
        while not item_is_available():
            cv.wait() # Ожидать появления нового элемента
        cv.release()
        consume_item()
```

Тонкость заключается в том, что при наличии нескольких потоков, ожидающих изменения одной переменной состояния, метод **notify()** может возобновить работу одного или более из них. Конкретное поведение часто зависит от операционной системы. Вследствие этого существует вероятность, что после возобновления работы поток обнаружит, что интересующее его состояние уже отсутствует. Это объясняет, например, почему в функции **consumer()** используется цикл **while**. Если поток возобновил работу, но элемент уже был обработан другим потоком, он опять переходит к ожиданию следующего извещения.

Пример использования переменных состояния представлен в файле **examples/01_thread/05_thread_condition_var.py**.

Работа с блокировками

При работе с любыми механизмами синхронизации — **Lock**, **RLock** или **Semaphore** — следует быть очень внимательными. Ошибки в управлении блокировками часто приводят к взаимоблокировкам потоков и состоянию гонки за ресурсами. Программный код, использующий блокировки, должен гарантировать их освобождение даже при возникновении исключений. Ниже приводится типичный пример такого программного кода:

```
try:
    lock.acquire()
    # критический раздел
    инструкции
    ...
finally:
    lock.release()
```

Все блокировки поддерживают протокол менеджера контекста, что позволяет писать более ясный код:

```
with lock:
    # критический раздел
    инструкции
    ...
```

В этом примере блокировка автоматически приобретается инструкцией **with** и освобождается, когда поток управления выходит за пределы контекста.

Не стоит писать программный код, который может обладать более чем одной блокировкой одновременно:

```
with lock_A:
    # критический раздел A
    инструкции
    ...

with lock_B:
    # критический раздел B
    инструкции
    ...
```

Такой код становится источником непонятных ***взаимоблокировок в программе***. Есть стратегии, позволяющие их избегать (например, иерархические блокировки), но лучше полностью отказаться от подобного стиля кодирования.

Приостановка и завершение потока

Потоки не имеют методов для принудительного завершения или приостановки. Это обусловлено сложностями разработки многопоточных программ. Если поток владеет блокировкой, из-за его принудительного завершения или приостановки все приложение может зависнуть. Кроме того, обычно нет возможности взять и «освободить все блокировки» по завершении, потому что при сложной процедуре синхронизации потоков бывает необходимо точно соблюсти последовательность операций приобретения и освобождения блокировок.

Если в программе потребуется возможность завершения или приостановки потока, ее придется реализовать самостоятельно. Обычно для этого поток проверяет в цикле свое состояние и определяет момент, когда он должен завершиться:

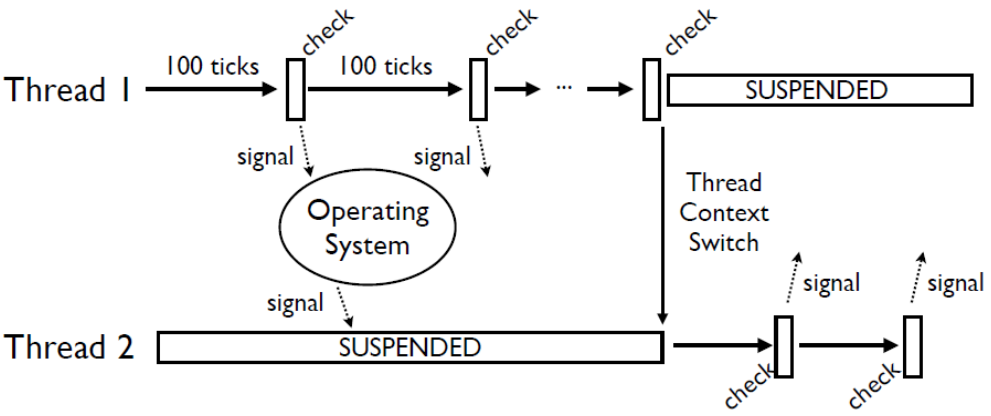
```
class StoppableThread(Thread):
    def __init__(self):
        Thread.__init__()
        self._terminate = False
        self._suspend_lock = Lock()

    def terminate(self):
        self._terminate = True

    def suspend(self):
        self._suspend_lock.acquire()

    def resume(self):
        self._suspend_lock.release()

    def run(self):
        while True:
        if self._terminate:
            break
        self._suspend_lock.acquire()
        self._suspend_lock.release()
        ...
```

Чтобы обеспечить надежную работу при таком подходе, поток не должен выполнять операции ввода–вывода, которые могут быть заблокированы. Если поток приостанавливается в ожидании поступления новых данных, он не может быть завершен, пока не закончит эту операцию. Поэтому желательно ограничивать ожидание в операциях ввода-вывода интервалом времени, использовать их неблокирующие версии и другие дополнительные возможности. Это позволит гарантировать, что проверка на необходимость завершения будет выполняться достаточно часто.

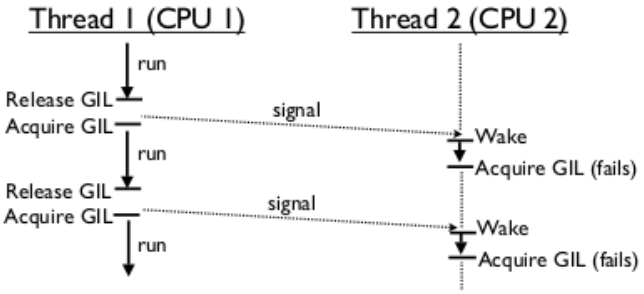
Глобальная блокировка интерпретатора

Интерпретатор Python выполняется под защитой блокировки, которая позволяет выполняться только одному потоку управления единовременно, даже если в системе несколько процессоров. Это обстоятельство существенно ограничивает выгоды, которые могло бы принести использование потоков в программах, выполняющих массивные вычисления. Фактически использование потоков в них часто ухудшает производительность по сравнению с однопоточными программами. Поэтому потоки управления должны использоваться только там, где основная задача — операции ввода-вывода, например, в сетевых серверах. Для массивных вычислений лучше применять модули расширений на языке C — они могут освободить блокировку интерпретатора и выполняться параллельно. Это при условии, что не будут взаимодействовать с интерпретатором после освобождения блокировки. Другой вариант — задействовать модуль **multiprocessing**. Он позволяет переложить работу на независимые дочерние процессы, которые не ограничиваются этой блокировкой.

До версии Python 3.2 в качестве базовой единицы таймаута **GIL** использует внутренний такт (**tick**) интерпретатора Python. Каждые 100 тактов осуществляется проверка глобальной переменной состояния.



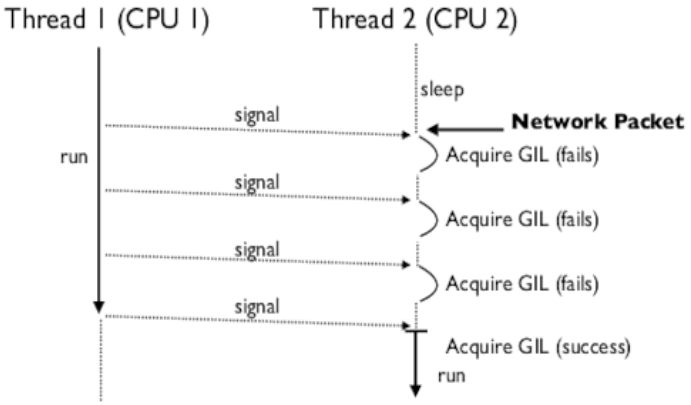
На многоядерной системе CPU-зависимые процессы переключаются одновременно (на разных ядрах), и происходит борьба за **GIL**:



Ожидающий поток при этом может сделать сотни безуспешных попыток захватить **GIL**.

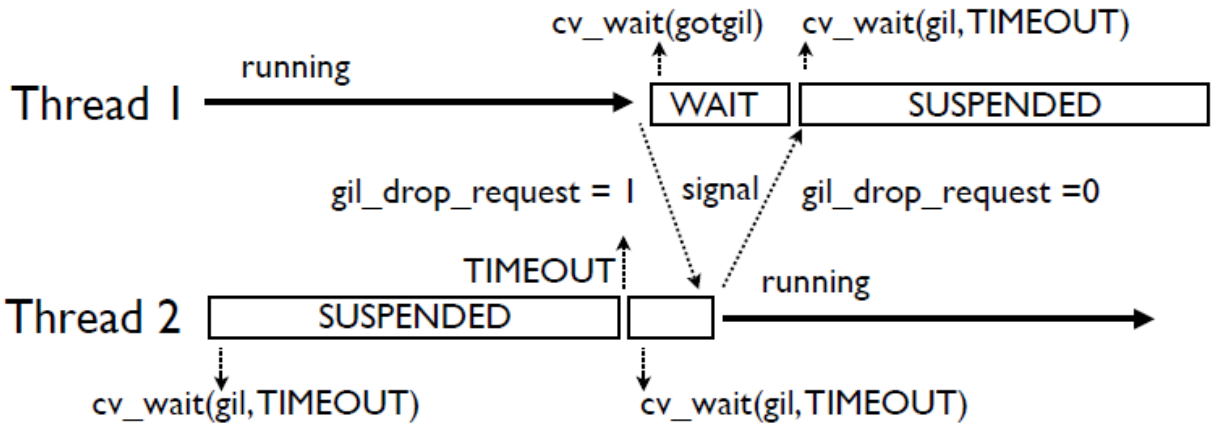
Происходит битва за две взаимоисключающие цели. Python запускает не больше одного потока в один момент, в то время как операционная система щедро переключает потоки, пытаясь извлечь максимальную выгоду из всех ядер.

Даже один CPU-зависимый поток порождает проблемы: увеличивает время отклика потока, осуществляющего ввод–вывод.



В результате этих особенностей старая версия **GIL** может внести существенные задержки в работу многопоточного кода.

Начиная с версии **Python 3.2** в качестве единицы таймаута используется **реальное время**. Также проверяется глобальная переменная состояния блокировки, но теперь это запрос на освобождения **GIL**. По умолчанию таймаут переключения потоков равен 5 мс. Такой принцип позволяет значительно ускорить работу многопоточного кода по сравнению с предыдущей версией **GIL**.



Тем не менее **GIL** присутствует во всех процессах и потоках, и поэтому реализация CPU-зависимых функций через потоки — не лучшее решение в Python. Но задачи обработки ввода-вывода, сетевых соединений можно решать при помощи потоков. Пример, демонстрирующий взаимосвязь **GIL** и потоков, представлен в файле **examples/03_gil/01_gil.py**.

Разработка многопоточных программ

На Python можно писать традиционные многопоточные программы, используя комбинации блокировок и других механизмов синхронизации. Но есть еще один стиль программирования, который обладает преимуществами перед всеми остальными: организовать многопоточную программу как коллекцию независимых задач, взаимодействующих с помощью очередей сообщений.

Модуль queue

Модуль **queue** (в Python 2 — **Queue**) реализует различные типы очередей, которые поддерживают возможность доступа из множества потоков и обеспечивают сохранность информации при обмене данными между несколькими потоками управления.

Модуль **queue** определяет три класса очередей:

- **Queue([maxsize])** — создает очередь типа **FIFO** (**first-in first-out** — первым пришел, первым вышел). Аргумент **maxsize** определяет максимальное количество элементов, которое может поместиться в очередь. При вызове без аргумента, или когда значение maxsize равно 0, размер очереди не ограничивается;
- **LifoQueue([maxsize])** — создает очередь типа **LIFO** (**last-in, first-out** —– последним пришел, первым вышел), которая известна как стек;
- **PriorityQueue([maxsize])** — создает очередь, в которой все элементы упорядочиваются по приоритетам, от низшего к высшему. Элементами очереди этого типа могут быть только кортежи вида (**priority**, **data**), где поле **priority** является числом.

Экземпляр **q** любого из классов очередей обладает следующими методами:

- **q.qsize()** — возвращает примерный размер очереди. Так как другие потоки могут добавлять и извлекать элементы, результат вызова этой функции не может считаться надежным;
- **q.empty()** — **True**, если в момент вызова очередь была пустой, и **False** — если нет;
- **q.full()** — **True**, если в момент вызова очередь была полной, и **False** в противном случае;
- **q.put(item [, block [, timeout]])** — добавляет элемент **item** в очередь. Если необязательный аргумент **block** имеет значение **True** (по умолчанию), при отсутствии свободного пространства в очереди вызывающий поток будет приостановлен. Когда в аргументе **block** передается значение **False**, в аналогичной ситуации в очереди будет вызвано исключение **Full**. Аргумент **timeout** определяет предельное время ожидания в секундах. По его истечении также появится исключение **Full**;
- **q.put_nowait(item)** — соответствует вызову **q.put(item, False)**;
- **q.get([block [, timeout]])** — удаляет и возвращает элемент из очереди. Если необязательный аргумент **block** имеет значение **True** (по умолчанию), при отсутствии элементов в очереди вызывающий поток будет приостановлен. Когда в аргументе **block** передается значение **False**, в аналогичной ситуации возникнет исключение **Empty**. Аргумент **timeout** определяет предельное время ожидания в секундах. По его истечении будет вызвано исключение **Empty**;
- **q.get_nowait()** — соответствует вызову **q.get(0)**;
- **q.task_done()** — используется потребителем, чтобы сообщить, что элемент очереди был обработан. Если этот метод используется, он должен вызываться один раз для каждого элемента, удаленного из очереди;
- **q.join()** — приостанавливает поток, пока не будут удалены и обработаны все элементы очереди. Возвращает управление только после того, как для каждого элемента очереди будет вызван метод **q.task_done()**.

Использование очереди в потоках

Разработку многопоточных программ часто можно упростить, используя очереди. Вместо того, чтобы использовать разделяемые данные, доступ к которым необходимо осуществлять под защитой блокировки, потоки могут обмениваться информацией с помощью очередей. В этой модели поток, занимающийся обработкой данных, обычно играет роль потребителя. Рассмотрим пример, иллюстрирующий эту концепцию (файл **examples/01_thread/06_thread_queue.py**):

```
from threading import Thread
from queue import Queue

class WorkerThread(Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.input_queue = Queue()

    def send(self, item):
        self.input_queue.put(item)

    def close(self):
```

```
        self.input_queue.put(None)
        self.input_queue.join()

    def run(self):
        while True:
            item = self.input_queue.get()
            if item is None:
                break
# Обработать элемент (вместо print могут быть полезные операции)
            print(item)
            self.input_queue.task_done()
# Конец. Сообщить, что сигнальная метка была принята, и выйти
        self.input_queue.task_done()

# Пример использования
w = WorkerThread()
w.start()
# Отправить элемент на обработку (с помощью очереди)
w.send("hello")
w.send("world")
w.close()
```

Стоит отметить некоторые **особенности** этого класса.

Во-первых, его программный интерфейс — это подмножество методов объектов класса **Connection**, которые создаются каналами в модуле **multiprocessing**. Это обеспечивает возможность дальнейшего расширения. Например, позднее обрабатывающий поток может быть вынесен в отдельный процесс без переделки программного кода, который посылает данные этому потоку.

Во-вторых, класс может завершать поток. Метод **close()** помещает в очередь сигнальную метку (значение **None**), которая в свою очередь вызывает завершение потока.

Модуль multiprocessing

Модуль **multiprocessing** предоставляет поддержку запуска задач в виде дочерних процессов, взаимодействий между ними и совместного использования данных, а также обеспечивает различные способы синхронизации. Программный интерфейс модуля имитирует интерфейс потоков, реализованный в модуле **threading**. Но важное отличие от потоков состоит в том, что процессы не имеют совместно используемых данных. Если процесс изменит данные, эти корректировки будут носить локальный характер для данного процесса.

У модуля **multiprocessing** широкие возможности, что делает его одной из самых крупных и сложных встроенных библиотек. Здесь невозможно во всех подробностях описать каждую особенность модуля. Мы рассмотрим основные его части и примеры использования. Опытные программисты смогут взять их за основу и распространить приемы на более сложные задачи.

Процессы

Все функциональные возможности модуля **multiprocessing** направлены на работу с процессами, которые описываются следующим классом:

- **Process ([group [, target [, name [, args [, kwargs]]]]])** — класс, представляющий задачу, запущенную в дочернем процессе.

Параметры всегда должны передаваться конструктору в виде именованных аргументов. В **target** передается объект, поддерживающий возможность вызова, который будет выполнен при запуске процесса. В **args** — кортеж позиционных аргументов для функции **target**, а в **kwargs** — словарь именованных аргументов для объекта **target**. Если опустить аргументы **args** и **kwargs**, объект **target** будет вызван без аргументов. В аргументе **name** передается строка с описательным именем процесса. Аргумент **group** не используется и всегда принимает значение **None**. Он присутствует лишь для полной имитации создания потоков с помощью модуля **threading**.

Экземпляр **p** класса **Process** обладает следующими методами:

- **p.is_alive()** — возвращает **True**, если процесс **p** продолжает работу;
- **p.join([timeout])** — ожидает завершения процесса **p**. Аргумент **timeout** определяет период ожидания. Присоединяться к процессу в ожидании его завершения можно неограниченное число раз, но будет ошибкой, если процесс попытается присоединиться к себе самому;
- **p.run()** — метод, который вызывается в дочернем процессе при его запуске. По умолчанию вызывает объект **target**, который был передан конструктору класса **Process**. При желании можно создать свой класс, производный от **Process**, и определить в нем собственную реализацию метода **run()**;
- **p.start()** — запускает дочерний процесс и вызывает в нем метод **p.run()**;
- **p.terminate()** — принудительно завершает процесс. При вызове этого метода дочерний процесс завершается немедленно, без заключительных процедур. Если **p** создал свои дочерние процессы, они превратятся в «зомби». Этот метод требует осторожного обращения. Если процесс **p** установил блокировку или вовлечен во взаимодействия с другими процессами, его принудительное завершение может вызвать взаимоблокировку процессов или повреждение данных.

Экземпляр **p** класса **Process** обладает следующими атрибутами:

- **p.authkey** — ключ аутентификации процесса. Если значение не было определено явно, в этот атрибут записывается 32-символьная строка, сгенерированная функцией **os.urandom()**. Назначение этого ключа в том, чтобы обеспечить безопасность низкоуровневых операций взаимодействия между процессами, которые выполняются через сетевые соединения. Взаимодействия через такие соединения будут возможны, только если с обоих концов используется один и тот же ключ аутентификации;
- **p.daemon** — логический флаг, указывающий, будет ли дочерний процесс демоническим. Демонический процесс завершается автоматически — вместе с процессом Python, создавшим его. Кроме того, демонический процесс не может создавать дочерние. Значение атрибута **p.daemon** должно устанавливаться до вызова метода **p.start()**;
- **p.exitcode** — целочисленный код завершения процесса. Если процесс продолжает выполняться, этот атрибут будет содержать значение **None**. Отрицательное значение **-N** означает, что процесс был завершен сигналом **N**;
- **p.name** — имя процесса;
- **p.pid** — целочисленный идентификатор процесса.

Следующий пример демонстрирует, как создавать и запускать функции (или другие вызываемые объекты) в отдельном процессе (файл **examples/02_multiprocess/01_proc_simple.py**):

```
import multiprocessing
import time

def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)

if __name__ == "__main__":
    p = multiprocessing.Process(target=clock, args=(15,))
    p.start()
```

И как определить свой класс, производный от **Process** (файл **examples/02_multiprocess/02_proc_subclass.py**):

```
import multiprocessing
import time

class ClockProcess(multiprocessing.Process):
    def __init__(self, interval):
        multiprocessing.Process.__init__(self)
        self.interval = interval

    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)

if __name__ == "__main__":
    p = ClockProcess(15)
    p.start()
```

В обоих примерах дочерние процессы должны выводить текущее время каждые 15 секунд. Для межплатформенной совместимости новые процессы должны создаваться только основной программой, как показано в примерах. В UNIX это условие можно не соблюдать, но в Windows — обязательно. Кроме того, в Windows предыдущие примеры желательно запускать в командной оболочке (**cmd.exe**), так как интегрированная среда разработки может не предоставлять возможностей взаимодействия с дочерними процессами (**PyCharm** под Windows не имеет такого ограничения).

Взаимодействие между процессами

Модуль **multiprocessing** поддерживает два основных способа взаимодействия процессов: каналы и очереди. Оба реализованы на основе механизма передачи сообщений. Причем интерфейс очередей очень близко имитирует подобный из многопоточных программ.

- **Queue([maxsize])** — создает очередь для организации обмена сообщениями между процессами. Аргумент **maxsize** определяет максимальное количество элементов, которые можно поместить в очередь. При вызове без аргумента ее размер не ограничивается.

Внутренняя реализация очередей основана на использовании каналов и блокировок. Для передачи данных из очереди в канал запускается вспомогательный поток управления.

Экземпляр **q** класса **Queue** обладает следующими методами:

- **q.cancel_join_thread()** — предотвращает автоматическое присоединение к фоновому потоку при завершении процесса. Благодаря этому не будет блокироваться процесс в вызове метода **join_thread()**;

- **q.close()** — закрывает очередь, запрещая добавление новых элементов. После вызова этого метода фоновый вспомогательный поток продолжит запись данных из очереди в канал и завершится, как только очередь будет исчерпана. Этот метод вызывается автоматически, когда экземпляр **q** утилизируется сборщиком мусора. Операция закрытия очереди не генерирует признак окончания передачи данных и не вызывает исключение на принимающей стороне. Если принимающий процесс (потребитель) находится в ожидании в методе **get()**, закрытие очереди на стороне передающего процесса (поставщика) не приведет к выходу из метода с признаком ошибки на стороне потребителя;

- **q.empty()** — возвращает **True**, если в момент вызова очередь **q** была пустой. Если есть другие процессы и потоки, добавляющие новые элементы в очередь, результат вызова этой функции не может считаться надежным. Между моментом получения результата и моментом его проверки в очередь могут быть добавлены новые элементы;

- **q.full()** — **True**, если в момент вызова очередь **q** была полной. Результат этой функции также нельзя считать надежным в многопоточных приложениях (см. описание **q.empty()**);

- **q.get([block [, timeout]])** — возвращает элемент из очереди **q**. Если она пуста, процесс приостанавливается до появления в ней элемента. Аргумент **block** управляет режимом блокировки и по умолчанию имеет значение **True**. Если в нем передать **False**, при попытке получить элемент из пустой очереди метод будет вызывать исключение **queue.Empty**. Аргумент **timeout** используется, когда блокировка разрешена. Если в течение указанного интервала времени в очереди не появится сообщений, будет вызвано исключение **queue.Empty**;

- **q.get_nowait()** — то же, что и **q.get(False)**;

- **q.join_thread()** — выполняет присоединение к фоновому потоку очереди. Может использоваться, чтобы дождаться момента, когда очередь будет исчерпана после вызова метода **q.close()**. По умолчанию вызывается всеми процессами, которые не являются создателями очереди **q**. Такое поведение можно изменить, вызвав метод **q.cancel_join_thread()**;

- **q.put(item [, block [, timeout]])** — добавляет элемент **item** в очередь. Если она заполнена до предела, процесс приостанавливается до появления места. Аргумент **block** управляет режимом блокировки и по умолчанию имеет значение **True**. Если в этом аргументе передать **False**, при попытке добавить элемент в заполненную очередь метод будет вызывать исключение **queue.Full**. Аргумент **timeout** определяет предельное время ожидания свободного места в очереди, когда блокировка разрешена. По истечении этого интервала будет вызвано исключение **queue.Full**;

- **q.put_nowait(item)** — то же, что и **q.put(item, False)**;

- **q.qsize()** — возвращает примерное количество элементов, находящихся в очереди. Результат этой функции также нельзя считать надежным, потому что между моментом получения результата и его проверки могут быть добавлены новые элементы или извлечены

существующие. В некоторых системах этот метод может вызывать исключение **NotImplementedError**;

- **JoinableQueue([maxsize])** — создает необособленный процесс очереди, доступной для совместного использования. Очереди этого типа похожи на **Queue**, но позволяют потребителю известить поставщика, что элементы были благополучно обработаны. Передача извещений реализована на основе разделяемых семафоров и переменных состояния;

Экземпляр **q** класса **JoinableQueue** обладает теми же методами, что и экземпляр класса **Queue**, а также дополнительными:

- **q.task_done()** — используется потребителем, чтобы сообщить, что элемент очереди, полученный методом **q.get()**, был обработан. Вызывает исключение **ValueError**, если количество вызовов этого метода превышает число элементов, извлеченных из очереди;

- **q.join()** — используется поставщиком, чтобы дождаться момента, когда будут обработаны все элементы очереди. Этот метод приостанавливает процесс, пока для каждого элемента в очереди не будет вызван метод **q.task_done()**.

Следующий пример демонстрирует, как реализовать процесс, который в бесконечном цикле получает элементы из очереди и обрабатывает их. Поставщик добавляет элементы в очередь и ожидает, пока они будут обработаны (файл **examples/02_multiprocess/03_proc_queue.py**):

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        # Обработать элемент
        print(item) # <- Здесь может быть обработка элемента
        # Сообщить о завершении обработки
        input_q.task_done()

def producer(sequence, output_q):
    for item in sequence:
        output_q.put(item)      # Добавить элемент в очередь

if __name__ == '__main__':
    q = multiprocessing.JoinableQueue()
    # Запустить процесс-потребитель
    cons_p = multiprocessing.Process(target=consumer, args=(q,))
    cons_p.daemon = True
    cons_p.start()

    # Воспроизвести элементы.
    # sequence — последовательность элементов, которые передаются потребителю.
    # На практике вместо переменной можно использовать генератор
    # или воспроизводить элементы каким-либо другим способом.
    sequence = [1,2,3,4]
    producer(sequence, q)
    q.join()    # Дождаться, пока все элементы не будут обработаны
```

В этом примере процесс-потребитель запускается как демонический, потому что он выполняется в бесконечном цикле — а нам необходимо, чтобы он завершался вместе с главной программой. Если

этого не сделать, программа зависнет. Чтобы в процессе-поставщике определить момент, когда все элементы будут успешно обработаны, используется очередь типа **JoinableQueue**. Это гарантирует метод **join()** — если забыть его вызвать, процесс-потребитель будет завершен еще до того, как успеет обработать все элементы в очереди.

Добавлять и извлекать элементы очереди могут сразу несколько процессов. Если данные должны получать сразу несколько процессов-потребителей, это можно реализовать, как показано ниже (файл **examples/02_multiprocess/04_proc_consumers.py**):

```
if __name__ == '__main__':
    q = multiprocessing.JoinableQueue()
    # Запустить несколько процессов-потребителей
    cons_p1 = multiprocessing.Process(target=consumer, args=(q, ))
    cons_p1.daemon = True
    cons_p1.start()
    cons_p2 = multiprocessing.Process(target=consumer, args=(q, ))
    cons_p2.daemon = True
    cons_p2.start()

    # Воспроизвести элементы.
    # Переменная sequence представляет последовательность элементов, которые
    # будут передаваться потребителю. На практике вместо переменной можно
    # использовать генератор или воспроизводить элементы другим
    # способом.
    sequence = [1, 2, 3, 4]
    producer(sequence, q)

    # Дождаться, пока все элементы не будут обработаны
    q.join()
```

Каждый элемент, помещаемый в очередь, преобразуется в последовательность байтов и отправляется процессу через канал или сетевое соединение. С точки зрения производительности лучше послать небольшое количество крупных объектов, чем много маленьких.

Чтобы поставщик извещал потребителей, что элементов больше не будет и надо завершить работу, можно использовать сигнальную метку — специальное значение, которое извещает об окончании работы. Рассмотрим, как использовать значение **None** в качестве сигнальной метки (файл **examples/02_multiprocess/05_proc_queue_flag.py**):

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        if item is None:
            break
        print(item)             # Обработать элемент (здесь может быть полезная
обработка элемента)
    print("Потребитель завершил работу")

def producer(sequence, output_q):
    for item in sequence:
        output_q.put(item)     # Добавить элемент в очередь
```

```
if __name__ == '__main__':
    q = multiprocessing.Queue()
                                # Запустить процесс-потребитель
    cons_p = multiprocessing.Process(target=consumer, args=(q, ))
    cons_p.start()

                                # Воспроизвести элементы.
    sequence = [1, 2, 3, 4]
    producer(sequence, q)

    q.put(None)     # Сообщить о завершении, поместив в очередь сигнальную метку
    cons_p.join()   # Дождаться, пока завершится процесс-потребитель
```

Следует помнить, что необходимо добавить в очередь по одной сигнальной метке для каждого потребителя. Если было запущено три процесса-потребителя, извлекающих элементы из очереди, процесс-поставщик должен добавить в очередь три сигнальные метки, чтобы завершить работу всех потребителей.

Для обмена сообщениями между процессами вместо очередей можно использовать каналы.

- **Pipe([duplex])** — создает канал между процессами и возвращает кортеж (**conn1**, **conn2**), где поля **conn1** и **conn2** являются объектами класса **Connection** и представляют концы канала. По умолчанию создается двунаправленный канал. Если в аргументе **duplex** передать значение **False**, объект **conn1** можно будет использовать только для чтения, а **conn2** — только для записи. Функция **Pipe()** должна вызываться до создания и запуска объектов класса **Process**, которые будут пользоваться каналом.

Экземпляр **con** класса **Connection**, возвращаемый функцией **Pipe()**, обладает следующими методами и атрибутами:

- **con.close()** — закрывает соединение. Вызывается автоматически, когда объект **con** утилизируется сборщиком мусора;
- **con.fileno()** — возвращает целочисленный дескриптор файла, идентифицирующий соединение;
- **con.poll([timeout])** — возвращает **True** при наличии данных в канале. Аргумент **timeout** определяет предельное время ожидания. При вызове без аргумента метод немедленно возвращает результат. Если в аргументе **timeout** передать значение **None**, метод будет ожидать появления данных неопределенно долго;
- **con.recv()** — извлекает из канала объект, отправленный методом **con.send()**. Если с другой стороны соединение было закрыто и в канале нет данных, вызывает исключение **EOFError**;
- **con.recv_bytes([maxlength])** — принимает строку байтов сообщения, отправленную методом **con.send_bytes()**. Аргумент **maxlength** определяет максимальное количество байтов, которые требуется принять. Если входящее сообщение длиннее заданного значения, вызывается исключение **IOError**, после чего последующие операции чтения из канала невозможны. Если с другой стороны соединение было закрыто и в канале нет данных, возникает исключение **EOFError**;
- **con.recv_bytes_into(buffer [, offset])** — принимает строку байтов сообщения и сохраняет ее в объекте **buffer**, который должен поддерживать интерфейс буферов, доступных для записи (как объект типа **bytearray**). Аргумент **offset** определяет смещение в байтах от начала буфера,

куда будет записано сообщение. Возвращает количество прочитанных байтов. Если длина сообщения превышает объем доступного пространства в буфере, будет выброшено исключение **BufferTooShort**;

- **con.send(obj)** — отправляет объект через соединение. Аргумент **obj** может быть любым объектом, совместимым с модулем **pickle**;
- **con.send_bytes(buffer [, offset [, size]])** — отправляет строку байтов из буфера через соединение. Аргумент **buffer** может быть любым объектом, поддерживающим интерфейс буферов. Аргумент **offset** определяет смещение в байтах от начала буфера, а **size** — количество байтов, которые требуется отправить. Данные отправляются в виде одного сообщения и могут быть приняты одним вызовом метода **con.recv_bytes()**.

Работа с каналами мало чем отличается от действий с очередями. Пример решения предыдущей задачи: передачи данных между поставщиком и потребителем на основе каналов (файл **examples/02_multiprocess/06_proc_pipes.py**):

```
import multiprocessing

# Получает элементы из канала
def consumer(pipe):
    output_p, input_p = pipe
    input_p.close() # Закрыть конец канала, доступный для записи
    while True:
        try:
            item = output_p.recv()
        except EOFError:
            break
        print(item) # Обработать элемент. Замените print фактической обработкой
                    # Завершение
    print("Потребитель завершил работу")

# Создает элементы и помещает их в канал
# sequence - итерируемый объект с элементами, которые требуется обработать
def producer(sequence, input_p):
    for item in sequence:
        input_p.send(item)     # Послать элемент в канал

if __name__ == '__main__':
    output_p, input_p = multiprocessing.Pipe()
# Запустить процесс-потребитель
    cons_p = multiprocessing.Process(target=consumer, args=((output_p, input_p),
))
    cons_p.start()
# Закрыть в поставщике конец канала, доступный для чтения
    output_p.close()

# Отправить элементы
    sequence = [1,2,3,4]
    producer(sequence, input_p)
    input_p.close()     # Сообщить об окончании, закрыв конец канала, доступный
для записи
    cons_p.join()       # Дождаться, пока завершится процесс-потребитель
```

Особое внимание надо уделять корректному управлению концами канала. Если один из них не используется поставщиком или потребителем, его следует закрыть. Поэтому в примере процесс-поставщик закрывает конец канала, доступный для чтения, а процесс-потребитель — доступный для записи. Если забыть выполнить одну из этих операций, программа может зависнуть при вызове метода **recv()** в процессе-потребителе. Операционная система ведет подсчет ссылок на каналы, и чтобы вызвать исключение **EOFError**, канал должен быть закрыт с обоих концов. Если сделать это только со стороны поставщика, результата не будет, пока потребитель не закроет канал со своего конца.

Каналы могут использоваться как средство обмена сообщениями в двух направлениях. Они позволяют писать программы, реализующие модель обмена «запрос-ответ», которая обычно применяется во взаимодействиях типа «клиент-сервер» или в вызовах удаленных процедур. Пример реализации такого типа взаимодействий (файл **examples/02_multiprocess/07_proc_pipes_client_server.py**):

```
import multiprocessing

# Серверный процесс
def adder(pipe):
    server_p, client_p = pipe
    client_p.close()
    while True:
        try:
            x, y = server_p.recv()
        except EOFError:
            break
        result = x + y
        server_p.send(result)
    print("Сервер завершил работу")     # Завершение

if __name__ == '__main__':
    server_p, client_p = multiprocessing.Pipe()
    # Запустить серверный процесс
    adder_p = multiprocessing.Process(target=adder, args=((server_p, client_p),
))
    adder_p.start()
    server_p.close()     # Закрыть серверный канал в клиенте
    # Послать серверу несколько запросов
    client_p.send((3, 4))
    print(client_p.recv())
    client_p.send(("Hello", "World"))
    print(client_p.recv())
    client_p.close()     # Конец. Закрыть канал
    adder_p.join()       # Дождаться, пока завершится серверный процесс
```

В этом примере функция **adder()** запускается как серверный процесс, ожидающий поступления сообщений на своем конце канала. Получив сообщение, сервер обрабатывает его и отправляет результаты обратно в канал. Не забывайте, что методы **send()** и **recv()** используют модуль **pickle** для сериализации и десериализации объектов. В примере сервер получает кортеж (**x, y**) и возвращает результат операции **x + y**. В более сложных приложениях могут использоваться вызовы удаленных процедур, для чего может потребоваться создать пул процессов.

Пример, демонстрирующий взаимосвязь **GIL** и процессов, представлен в файле **examples/03_gil/02_gil_proc.py**.

Общие советы по использованию модуля multiprocessing

Модуль **multiprocessing** — один из самых сложных и мощных модулей в стандартной библиотеке Python. Общие советы, которые помогут при работе с ним:

- Внимательно прочитайте официальную документацию, прежде чем создавать крупное приложение. В ней описывается множество коварных проблем, с которыми можно столкнуться;
- Обязательно убедитесь, что все типы данных, которые участвуют в обмене между процессами, совместимы с модулем **pickle**;
- Старайтесь не использовать механизмы совместного использования данных и учитесь пользоваться механизмами передачи сообщений и очередями. При применении механизмов обмена сообщениями вам не придется беспокоиться о синхронизации, блокировках и других проблемах. Такой подход обладает лучшей масштабируемостью, если процессов становится больше;
- Не пользуйтесь глобальными переменными в функциях, которые предназначены для работы в отдельных процессах. Лучше передавать параметры явно;
- Старайтесь не смешивать поддержку механизма потоков и процессов в одной программе;
- Особое внимание обращайте на то, как завершаются процессы. Предпочтительнее явно закрывать процессы и предусматривать четко оформленную процедуру их завершения, а не полагаться на механизм сборки мусора или принудительное завершение дочерних процессов с использованием операции **terminate()**;
- Несмотря на то, что этот модуль можно использовать в операционной системе Windows, следует тщательно проработать официальную документацию. Модуль **multiprocessing** предусматривает реализацию собственного клона функции **fork()** из UNIX для запуска новых процессов в Windows. Эта функция создает копию окружения родительского процесса и отправляет ее дочернему процессу с помощью канала. В целом, этот модуль больше подходит для использования в UNIX;
- И самое главное — старайтесь сделать реализацию максимально простой.

На данном занятии мы рассмотрели основы организации и взаимодействия параллельных задач, построенных на потоках и процессах. Уделили внимание глобальной блокировке интерпретатора и его воздействию на работу многопоточного приложения.

Практическое задание

1. На клиентской стороне реализовать прием и отправку сообщений с помощью потоков в P2P-формате (обмен сообщениями между двумя пользователями).

Итогом выполнения практических заданий первой части продвинутого курса Python стал консольный мессенджер. Усовершенствуем его во второй части: реализуем взаимосвязь мессенджера с базами данных и создадим для него графический пользовательский интерфейс.

Дополнительные материалы

1. Андрей Светлов. Загадочный GIL.
2. Хабрахабр. Правильное использование QThread.
3. Python Concurrency Cheatsheet.
4. Синхронизация потоков в Python.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. David Beazley. Understanding the Python GIL.
3. David Beazley. Inside the New GIL.
4. Giancarlo Zaccone. Python Parallel Programming Cookbook (каталог «Дополнительные материалы»).
5. Jan Palach. Parallel Programming with Python (каталог «Дополнительные материалы»).
6. Бизли Дэвид. Python. Подробный справочник. 4-е издание (каталог «Дополнительные материалы»).