Python. Лекция 11. Многопоточные вычисления.

О потоках управления

В современной операционной системе, даже не выполняющей ничего особенного, могут одновременно работать несколько **процессов** (processes). Например, при запуске программы запускается новый процесс. Функции для управления процессами можно найти в стандартном модуле `os` языка Python. Здесь же речь пойдет о потоках.

Потоки управления (threads) образуются и работают в рамках одного процесса. В однопоточном приложении (программе, которая не использует дополнительных потоков) имеется только один поток управления. Говоря упрощенно, при запуске программы этот поток последовательно исполняет встречаемые в программе операторы, направляясь по одной из альтернативных ветвей оператора выбора, проходит через тело цикла нужное число раз, выбирается к месту обработки исключения при возбуждении исключения. В любой момент времени интерпретатор Python знает, какую команду исполнить следующей. После исполнения команды становится известно, какой команде передать управление. Эта ниточка непрерывна в ходе выполнения программы и обрывается только по ее завершении.

Теперь можно представить себе, что в некоторой точке программы ниточка раздваивается, и каждый поток идет своим путем. Каждый из образовавшихся потоков может в дальнейшем еще несколько раз раздваиваться. (При этом один из потоков всегда остается главным, и его завершение означает завершение всей программы.) В каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Такое, казалось бы, незначительное усложнение механизма выполнения программы на самом деле требует качественных изменений в программе - ведь деятельность потоков должна быть согласована. Нельзя допускать, чтобы потоки одновременно изменяли один и тот же объект, результат такого изменения, скорее всего, нарушит целостность объекта.

Одним из классических средств согласования потоков являются объекты, называемые **семафорами**. Семафоры не допускают выполнения некоторого участка кода несколькими потоками одновременно. Самый простой семафор - **замок** (lock) или **mutex** (от английского mutually exclusive, взаимоисключающий). Для того чтобы поток мог продолжить выполнение кода, он должен сначала захватить замок. После захвата замка поток выполняет определенный участок кода и потом освобождает замок, чтобы другой поток мог его получить и пройти дальше к выполнению охраняемого замком участку программы. Поток, столкнувшись с занятым другим потоком замком, обычно ждет его освобождения.

Поддержка многопоточности в языке Python доступна через использование ряда модулей. В стандартном модуле `threading` определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров (классы замков `Lock`, `RLock` и класс `Semaphore`) и другие механизмы взаимодействия между потоками (классы `Event` и `Condition`), класс `Timer` для запуска функции по прошествии некоторого времени. Модуль `Queue` реализует очередь, которой могут пользоваться сразу несколько потоков. Для создания и (низкоуровневого) управления потоками в стандартном модуле `thread` определен класс `Thread`.

Пример многопоточной программы

В следующем примере создается два дополнительных потока, которые выводят на стандартный вывод каждый свое:

```
import threading

            def proc(n):
              print "Процесс", n

            p1 = threading.Thread(target=proc, name="t1", args=["1"])
            p2 = threading.Thread(target=proc, name="t2", args=["2"])
            p1.start()
            p2.start()
```

Сначала получается два объекта класса `Thread`, которые затем и запускаются с различными аргументами. В данном случае в потоках работает одна и та же функция `proc()`, которой передается один аргумент, заданный в именованном параметре `args` конструктора класса `Thread`. Нетрудно догадаться, что метод `start()` служит для запуска нового потока. Таким образом, в приведенном примере работают три потока: основной и два дополнительных (с именами `"t1"` и `"t2"`).

Функции модуля threading

В модуле `threading`, который здесь используется, есть функции, позволяющие получить информацию о потоках:

- `activeCount()` Возвращает количество активных в настоящий момент экземпляров класса `Thread`. Фактически, это `len(threading.enumerate())`.
- `currentThread()` Возвращает текущий объект-поток, то есть соответствующий потоку управления, который вызвал эту функцию. Если поток не был создан через модуль `threading`, будет возвращен объект-поток с сокращенной функциональностью (dummy thread object).
- `enumerate()` Возвращает список активных потоков. Завершившиеся и еще не начатые потоки не входят в список.

Класс Thread

Экземпляры класса `threading.Thread` представляют потоки Python-программы. Задать действия, которые будут выполняться в потоке, можно двумя способами: передать конструктору класса исполняемый объект и аргументы к нему или путем наследования получить новый класс с переопределенным методом `run()`. Первый способ был рассмотрен в примере выше. Конструктор класса `threading.Thread` имеет следующие аргументы:

```
Thread(group, target, name, args, kwargs)
```

Здесь `group` - группа потоков (пока что не используется, должен быть равен `None`), `target` - объект, который будет вызван в методе `run()`, `name` - имя потока, `args` и `kwargs` - последовательность и словарь позиционных и именованных параметров (соответственно) для вызова заданного в параметре `target` объекта. В примере выше были использованы только позиционные параметры, но то же самое можно было выполнить и с применением именованных параметров:

```
import threading

            def proc(n):
              print "Процесс", n

            p1 = threading.Thread(target=proc, name="t1", kwargs={"n": "1"})
            p2 = threading.Thread(target=proc, name="t2", kwargs={"n": "2"})
            p1.start()
            p2.start()
```

То же самое можно проделать через наследование от класса `threading.Thread` с определением собственного конструктора и метода `run()`:

```
import threading

            class T(threading.Thread):
              def __init__(self, n):
                threading.Thread.__init__(self, name="t" + n)
                self.n = n
              def run(self):
                print "Процесс", self.n

            p1 = T("1")
            p2 = T("2")
            p1.start()
            p2.start()
```

Самое первое, что необходимо сделать в конструкторе - вызвать конструктор базового класса. Как и раньше, для запуска потока нужно выполнить метод `start()` объекта-потока, что приведет к выполнению действий в методе `run()`.

Жизнью потоков можно управлять вызовом методов:

- `start()` Дает потоку жизнь.
- `run()` Этот метод представляет действия, которые должны быть выполнены в потоке.
- `join([timeout])` Поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван. Параметр `timeout` (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод `join` был вызван. Вызывать `join()` некоторого потока можно много раз. Поток не может вызвать метод `join()` самого себя. Также нельзя ожидать завершения еще не запущенного потока. Слово "join" в переводе с английского означает "присоединить", то есть, метод, вызвавший `join()`, желает, чтобы поток по завершении присоединился к вызывающему метод потоку.
- `getName()` Возвращает имя потока. Для главного потока это `"MainThread"`.
- `setName(name)` Присваивает потоку имя `name`.
- `isAlive()` Возвращает истину, если поток работает (метод `run()` уже вызван, но еще не завершился).
- `isDaemon()` Возвращает истину, если поток имеет признак демона. Программа на Python завершается по завершении всех потоков, не являющихся демонами. Главный поток демоном не является.
- `setDaemon(daemonic)` Устанавливает признак `daemonic` того, что поток является демоном. Начальное значение этого признака заимствуется у потока, запустившего данный. Признак можно изменять только для потоков, которые еще не запущены.

В модуле `Thread` пока что не реализованы возможности, присущие потокам в Java (определение групп потоков, приостановка и прерывание потоков извне, приоритеты и некоторые другие вещи), однако они, скорее всего, будут созданы в недалеком будущем.

Таймер

Класс `threading.Timer` представляет действие, которое должно быть выполнено через заданное время. Этот класс является подклассом класса `threading.Thread`, поэтому запускается также методом `start()`. Следующий простой пример, печатающий на стандартном выводе `Hello, world`! поясняет сказанное:

```
def hello():
             print "Hello, world!"

           t = Timer(30.0, hello)
           t.start()
```

Замки

Простейший замок может быть реализован на основе класса `Lock` модуля `threading`. Замок имеет два состояния: он может быть или открыт, или заперт. В последнем случае им владеет некоторый поток. Объект класса `Lock` имеет следующие методы:

- `acquire([blocking=True])` Делает запрос на запирание замка. Если параметр `blocking` не указан или является истиной, то поток будет ожидать освобождения замка. Если параметр не был задан, метод не возвратит значения. Если `blocking` был задан и истинен, метод возвратит `True` (после успешного овладения замком). Если блокировка не требуется (то есть задан `blocking=False`), метод вернет `True`, если замок не был заперт и им успешно овладел данный поток. В противном случае будет возвращено `False`.
- `release()` Запрос на отпирание замка.
- `locked()` Возвращает текущее состояние замка (`True` - заперт, `False` - открыт). Следует иметь в виду, что даже если состояние замка только что проверено, это не означает, что он сохранит это состояние до следующей команды.

Имеется еще один вариант замка - `threading.RLock`, который отличается от `threading.Lock` тем, что некоторый поток может запрашивать его запирание много раз. Отпирание такого замка должно происходить столько же раз, сколько было запираний. Это может быть полезно, например, внутри рекурсивных функций.

Когда нужны замки?

Замки позволяют ограничивать вход в некоторую область программы одним потоком. Замки могут потребоваться для обеспечения целостности структуры данных. Например, если для корректной работы программы требуется добавление определенного элемента сразу в несколько списков или словарей, такие операции в многопоточном приложении следует обставить замками. Вокруг атомарных операций над встроенными типами (операций, которые не вызывают исполнение какого-то другого кода на Python) замки ставить необязательно. Например, метод `append()` (встроенного) списка является атомарной операцией, а тот же метод, реализованный пользовательским классом, может требовать блокировок. В случае сомнений, конечно, лучше перестраховаться и поставить замки, однако следует минимизировать общее время действия замка, так как замок останавливает другие потоки, пытающиеся попасть в ту же область программы. Отсутствие замка в критической части программы, работающей над общими для двух и более потоков ресурсами, может привести к случайным, трудноуловимым ошибкам.

Тупиковая ситуация (deadlock)

Замки применяются для управления доступом к ресурсу, который нельзя использовать совместно. В программе таких ресурсов может быть несколько. При работе с замками важно хорошо продумать, не зайдет ли выполнение программы в **тупик** (deadlock) из-за того, что двум потокам потребуются одни и те же ресурсы, но ни тот, ни другой не смогут их получить, так как они уже получили замки. Такая ситуация проиллюстрирована в следующем примере:

```
import threading, time

                resource = {'A': threading.Lock(), 'B': threading.Lock()}

                def proc(n, rs):
                 for r in rs:
                   print "Процесс %s запрашивает ресурс %s" % (n, r)
                   resource[r].acquire()
                   print "Процесс %s получил ресурс %s" % (n, r)
                   time.sleep(1)
                 print "Процесс %s выполняется" % n
                 for r in rs:
                   resource[r].release()
                 print "Процесс %s закончил выполнение" % n

                p1 = threading.Thread(target=proc, name="t1", args=["1", "AB"])
                p2 = threading.Thread(target=proc, name="t2", args=["2", "BA"])
                p1.start()
                p2.start()
                p1.join()
                p2.join()
```

В этом примере два потока (t1 и t2) запрашивают замки к одним и тем же ресурсам (A и B), но в разном порядке, отчего получается, что ни у того, ни у другого не хватает ресурсов для дальнейшей работы, и они оба безнадежно повисают, ожидая освобождения нужного ресурса. Благодаря операторам `print` можно увидеть последовательность событий:

```
Процесс 1 запрашивает ресурс A
                Процесс 1 получил ресурс A
                Процесс 2 запрашивает ресурс B
                Процесс 2 получил ресурс B
                Процесс 1 запрашивает ресурс B
                Процесс 2 запрашивает ресурс A
```

Существуют методики, позволяющие избежать подобных тупиков, однако их рассмотрение не входит в рамки данной лекции. Можно посоветовать следующие приемы:

- построить логику приложения так, чтобы никогда не запрашивать замки к двум ресурсам сразу. Возможно, придется определить составной ресурс. В частности, к данному примеру можно было бы определить замок "AB" для указания эксклюзивного доступа к ресурсам A и B.
- строго упорядочить все ресурсы (например, по цене) и всегда запрашивать их в определенном порядке (скажем, начиная с более дорогих ресурсов). При этом перед заказом некоторого ресурса поток должен отказаться от заблокированных им более дешевых ресурсов.

Семафоры

Семафоры (их иногда называют семафорами Дийкстры (Dijkstra) по имени их изобретателя) являются более общим механизмом синхронизации потоков, нежели замки. Семафоры могут допустить в критическую область программы сразу несколько потоков. Семафор имеет счетчик запросов, уменьшающийся с каждым вызовом метода

`acquire()` и увеличивающийся при каждом вызове `release()`. Счетчик не может стать меньше нуля, поэтому в таком состоянии потокам приходится ждать, как и в случае с замками, пока значение счетчика не увеличится.

Конструктор класса `threading.Semaphore` принимает в качестве (необязательного) аргумента начальное состояние счетчика (по умолчанию оно равно 1, что соответствует замку класса `Lock`). Методы `acquire()` и `release()` действуют аналогично описанным выше одноименным методам у замков.

Семафор может применяться для охраны ограниченного ресурса. Например, с его помощью можно вести **пул** соединений с базой данных. Пример такого использования семафора (заимствован из документации к Python) дан ниже:

```
from threading import BoundedSemaphore
            maxconnections = 5
            # Подготовка семафора
            pool_sema = BoundedSemaphore(value=maxconnections)

            # Внутри потока:

            pool_sema.acquire()
            conn = connectdb()
            # ... использование соединения ...
            conn.close()
            pool_sema.release()
```

Таким образом, применяется не более пяти соединений с базой данных. В примере использован класс `threading.BoundedSemaphore`. Экземпляры этого класса отличаются от экземпляров класса `threading.Semaphore` тем, что не дают сделать `release()` больше, чем сделан `acquire()`.

События

Еще одним способом коммуникации между объектами являются **события**. Экземпляры класса `threading.Event` могут быть использованы для передачи информации о наступлении некоторого события от одного потока одному или нескольким другим потокам. Объекты-события имеют внутренний флаг, который может находиться в установленном или сброшенном состоянии. При своем создании флаг события находится в сброшенном состоянии. Если флаг в установленном состоянии, ожидания не происходит: поток, вызвавший метод `wait()` для ожидания события, просто продолжает свою работу. Ниже приведены методы экземпляров класса `threading.Event`:

- `set()` Устанавливает внутренний флаг, сигнализирующий о наступлении события. Все ждущие данного события потоки выходят из состояния ожидания.
- `clear()` Сбрасывает флаг. Все события, которые вызывают метод `wait()` этого объекта-события, будут находиться в состоянии ожидания до тех пор, пока флаг сброшен, или по истечении заданного таймаута.
- `isSet()` Возвращает состояние флага.
- `wait([timeout])` Переводит поток в состояние ожидания, если флаг сброшен, и сразу возвращается, если флаг установлен. Аргумент `timeout` задает таймаут в секундах, по истечении которого ожидание прекращается, даже если событие не наступило.

Составить пример работы с событиями предлагается в качестве упражнения.

Условия

Более сложным механизмом коммуникации между потоками является механизм условий. Условия представляются в виде экземпляров класса `threading.Condition` и, подобно только что рассмотренным событиям, оповещают потоки об изменении некоторого состояния. Конструктор класса `threading.Condition` принимает необязательный параметр, задающий замок класса `threading.Lock` или `threading.RLock`. По умолчанию создается новый экземпляр замка класса `threading.RLock`. Методы объекта-условия описаны ниже:

- `acquire(...)` Запрашивает замок. Фактически вызывается одноименный метод принадлежащего объекту-условию объекта-замка.
- `release()` Снимает замок.
- `wait([timeout])` Переводит поток в режим ожидания. Этот метод может быть вызван только в том случае, если вызывающий его поток получил замок. Метод снимает замок и блокирует поток до появления объявлений, то есть вызовов методов `notify()` и `notifyAll()` другими потоками. Необязательный аргумент `timeout` задает таймаут ожидания в секундах. При выходе из ожидания поток снова запрашивает замок и возвращается из метода `wait()`.
- `notify()` Выводит из режима ожидания один из потоков, ожидающих данные условия. Метод можно вызвать, только овладев замком, ассоциированным с условием. Документация предупреждает, что в будущих реализациях модуля из целей оптимизации этот метод будет прерывать ожидание сразу нескольких потоков. Сам по себе метод `notify()` не приводит к продолжению выполнения ожидавших условия потоков, так как этому препятствует занятый замок. Потоки получают управление только после снятия замка потоком, вызвавшим метод `notify()`.
- `notifyAll()` Этот метод аналогичен методу `notify()`, но прерывает ожидание всех ждущих выполнения условия потоков.

В следующем примере условия используются для оповещения потоков о прибытии новой порции данных (организуется связь производитель - потребитель, producer - consumer):

```
import threading

            cv = threading.Condition()

            class Item:
              """Класс-контейнер для элементов, которые будут потребляться
              в потоках"""
              def __init__(self):
                self._items = []
              def is_available(self):
                return len(self._items) > 0
              def get(self):
                return self._items.pop()
              def make(self, i):
                self._items.append(i)

            item = Item()

            def consume():
              """Потребление очередного элемента (с ожиданием его появления)"""
              cv.acquire()
              while not item.is_available():
                cv.wait()
              it = item.get()
              cv.release()
              return it

            def consumer():
```

```
  while True:
    print consume()

def produce(i):
  """Занесение нового элемента в контейнер и оповещение потоков"""
  cv.acquire()
  item.make(i)
  cv.notify()
  cv.release()

p1 = threading.Thread(target=consumer, name="t1")
p1.setDaemon(True)
p2 = threading.Thread(target=consumer, name="t2")
p2.setDaemon(True)
p1.start()
p2.start()
produce("ITEM1")
produce("ITEM2")
produce("ITEM3")
produce("ITEM4")
p1.join()
p2.join()
```

В этом примере условие `cv` отражает наличие необработанных элементов в контейнере `item`. Функция `produce()` "производит" элементы, а `consume()`, работающая внутри потоков, "потребляет". Стоит отметить, что в приведенном виде программа никогда не закончится, так как имеет бесконечный цикл в потоках, а в главном потоке - ожидание завершения этих потоков. Еще одна особенность - признак демона, установленный с помощью метода `setDaemon()` объекта-потока до его старта.

Очередь

Процесс, показанный в предыдущем примере, имеет значение, достойное отдельного модуля. Такой модуль в стандартной библиотеке языка Python есть, и он называется `Queue`.

Помимо исключений - `Queue.Full` (очередь переполнена) и `Queue.Empty` (очередь пуста) - модуль определяет класс `Queue`, заведующий собственно очередью.

Собственно, здесь можно привести аналог примера выше, но уже с использованием класса `Queue.Queue`:

```
import threading, Queue

item = Queue.Queue()

def consume():
  """Потребление очередного элемента (с ожиданием его появления)"""
  return item.get()

def consumer():
  while True:
    print consume()

def produce(i):
  """Занесение нового элемента в контейнер и оповещение потоков"""
  item.put(i)

p1 = threading.Thread(target=consumer, name="t1")
p1.setDaemon(True)
p2 = threading.Thread(target=consumer, name="t2")
p2.setDaemon(True)
p1.start()
p2.start()
produce("ITEM1")
produce("ITEM2")
```

```
produce("ITEM3")
produce("ITEM4")
p1.join()
p2.join()
```

Следует отметить, что все блокировки спрятаны в реализации очереди, поэтому в коде они явным образом не присутствуют.

Модуль thread

По сравнению с модулем `threading`, модуль `thread` предоставляет низкоуровневый доступ к потокам. Многие функции модуля `threading`, который рассматривался до этого, реализованы на базе модуля `thread`. Здесь стоит сделать некоторые замечания по применению потоков вообще. Документация по Python предупреждает, что использование потоков имеет особенности:

- Исключение `KeyboardInterrupt` (прерывание от клавиатуры) может быть получено любым из потоков, если в поставке Python нет модуля `signal` (для обработки сигналов).
- Не все встроенные функции, блокированные ожиданием ввода, позволяют другим потокам работать. Правда, основные функции вроде `time.sleep()`, `select.select()`, метод `read()` файловых объектов не блокируют другие потоки.
- Невозможно прервать метод `acquire()`, так как исключение `KeyboardInterrupt` возбуждается только после возврата из этого метода.
- Нежелательно, чтобы главный поток завершался раньше других потоков, так как не будут выполнены необходимые деструкторы и даже части `finally` в операторах `try-finally`. Это связано с тем, что почти все операционные системы завершают приложение, у которого завершился главный поток.

Визуализация работы потоков

Следующий пример иллюстрирует параллельность выполнения потоков, используя возможности библиотеки графических примитивов `Tkinter` (она входит в стандартную поставку Python). Несколько потоков наперегонки увеличивают размеры прямоугольника некоторого цвета. Цветом победившего потока окрашивается кнопка Go:

```
import threading, time, sys
            from Tkinter import Tk, Canvas, Button, LEFT, RIGHT, NORMAL, DISABLED

            global champion

            # Задается дистанция, цвет полосок и другие параметры
            distance = 300
            colors = ["Red","Orange","Yellow","Green","Blue","DarkBlue","Violet"]
            nrunners = len(colors)      # количество дополнительных потоков
            positions = [0] * nrunners  # список текущих позиций
            h, h2 = 20, 10              # параметры высоты полосок

            def run(n):
              """Программа бега n-го участника (потока)"""
              global champion
              while 1:
                for i in range(10000):           # интенсивные вычисления
                  pass
                graph_lock.acquire()
                positions[n] += 1                # передвижение на шаг
                if positions[n] == distance:     # если уже финиш
                  if champion is None:           # и чемпион еще не определен,
                    champion = colors[n]         # назначается чемпион
                  graph_lock.release()
                  break
                graph_lock.release()
```

```
def ready_steady_go():
  """Инициализация начальных позиций и запуск потоков"""
  graph_lock.acquire()
  for i in range(nrunners):
    positions[i] = 0
    threading.Thread(target=run, args=[i,]).start()
  graph_lock.release()

def update_positions():
  """Обновление позиций"""
  graph_lock.acquire()
  for n in range(nrunners):
    c.coords(rects[n], 0, n*h, positions[n], n*h+h2)
  tk.update_idletasks()  # прорисовка изменений
  graph_lock.release()

def quit():
  """Выход из программы"""
  tk.quit()
  sys.exit(0)

# Прорисовка окна, основы для прямоугольников и самих прямоугольников,
# кнопок для пуска и выхода
tk = Tk()
tk.title("Соревнование потоков")
c = Canvas(tk, width=distance, height=nrunners*h, bg="White")
c.pack()
rects = [c.create_rectangle(0, i*h, 0, i*h+h2, fill=colors[i])
         for i in range(nrunners)]
go_b = Button(text="Go", command=tk.quit)
go_b.pack(side=LEFT)
quit_b = Button(text="Quit", command=quit)
quit_b.pack(side=RIGHT)

# Замок, регулирующий доступ к функции пакета Tk
graph_lock = threading.Lock()

# Цикл проведения соревнований
while 1:
  go_b.config(state=NORMAL), quit_b.config(state=NORMAL)
  tk.mainloop()              # Ожидание нажатия клавиш
  champion = None
  ready_steady_go()
  go_b.config(state=DISABLED), quit_b.config(state=DISABLED)
  # Главный поток ждет финиша всех участников
  while sum(positions) < distance*nrunners:
    update_positions()
  update_positions()
  go_b.config(bg=champion)     # Кнопка окрашивается в цвет победителя
  tk.update_idletasks()
```

Примечание:

Эта программа использует некоторые возможности языка Python 2.3 (встроенную функцию `sum()` и списковые включения), поэтому для ее выполнения нужен Python версии не меньше 2.3.

Заключение

Навыки параллельного программирования необходимы любому профессиональному программисту. Одним из вариантов организации (псевдо) параллельного программирования является многопоточное программирование (другой вариант, более свойственный Unix-системам - многопроцессное программирование - здесь не рассматривается). В обычной (однопоточной) программе действует всего один поток управления, а в многопоточной одновременно могут работать несколько потоков.

Параллельное программирование требует тщательной отработки взаимодействия между потоками управления. Некоторые участки кода необходимо ограждать от одновременного использования двумя различными потоками, дабы не нарушить целостность изменяемых структур данных или логику работы с внешними ресурсами. Для ограждения участков кода используются замки и семафоры.

Стандартная библиотека Python предоставляет довольно неплохой набор возможностей для многопоточного программирования в модулях `threading` и `thread`, а также некоторые полезные вспомогательные модули (например, `Queue`).