ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

«Начинать нужно с малого. И тогда, может быть, мы научимся творить настоящие чудеса.».
Чак Паланик, «Удушье»

Урок 12. (Lesson 11) Основы многопоточного программирования

- **Использование модуля _thread**

Модуль *_thread* обеспечивает достаточно простой набор методов для создания параллельных потоков. Пусть он менее функционально богат, но работать с ним существенно проще, нежели с модулем *threading*. Равно как и *threading*, модуль *_thread* обеспечивает переносимый интерфейс для создания параллельных легковесных потоков и управления ими. Рассмотрим два примера, один из которых реализован с использованием *threading*, а другой – *_thread*.

Листинг 1.

```
import _thread
import time
def child(tid):
    print('Hello from thread %d\n' % tid)
def parent():
    for i in range(5):
        # создание и запуск дочернего потока
        _thread.start_new_thread(child, (i+1,))
    #time.sleep(5)
    print('Parent thread finished')
parent()
```

Листинг 2.

```
import threading
import time
def child(tid):
    print('Hello from thread %d' % tid)
def parent():
    for i in range(5):
        # создание дочернего потока
        t = threading.Thread(target=child, args=(i+1,))
        t.start()  # запуск дочернего потока
    #time.sleep(5)
    print('Parent thread finished')
parent()
```

Разница, пока, в формате вызова потоков и в наличие у класса *Thread* метода *start()*, который нужно явно вызывать для запуска дочернего потока.

Запустите данные сценарии из командной строки. Сравните результаты работы кода в случае, когда инструкция *time.sleep(5)* закомментирована, и когда – нет.

Работа кода начинается с запуска функции *parent()*, внутри которой в цикле производится вызов функции запуска дочернего параллельного потока - *_thread.start_new_thread*. Ввиду того, что родительский и дочерний потоки никак не связаны, сразу после исполнения цикла работа родительского потока завершается. Завершение работы родительского потока проводит к автоматическому завершению дочернего потока. Поэтому вполне возможна ситуация, когда при запуске данных сценариев дочерние потоки просто ничего не успеют вывести в стандартный поток вывода.

Несколько модифицируем первый пример и запустим его на исполнение из среды *Python IDLE* и из командной строки.

```
import _thread
from tkinter import *
def child(tid):
    print('Hello from thread %d' % tid)
    root = Tk()
    Label(root, text='Hello from thread %d' % tid,
          font=('times', 20)).pack()
    root.mainloop()

def parent():
    for i in range(5):
        _thread.start_new_thread(child, (i,))
    input('quit - press Enter-key')
    print('Parent thread exiting.')

parent()
```

Результаты запуска данного сценария могут существенно различаться, так как сказывается наличие «***критической секции***», разделяемого ресурса – стандартного потока вывода и отсутствие использования механизмов, позволяющих родительскому потоку дожидаться завершения работы дочерних потоков.

Важным является тот факт, что функция *_thread.start_new_thread* не возвращает никакого значения, сигнализирующего о том, как был запущен или завершен дочерний поток. Кроме того, порожденный дочерний поток получается несколько изолированным – выброс объекта исключения в дочернем потоке не влияет на состояние родительского потока. Такая «***изоляция***» дочернего потока обеспечивается корректным использованием модели перехвата и обработки исключений, реализованной внутри функции *_thread.start_new_thread*.

```
import _thread
import time

def child(tid):
    print('Hello from thread %d' % tid)
    raise NameError('Hi-Hi...')  # генерируем исключение

def parent():
    for i in range(5):
        _thread.start_new_thread(child, (i,))
    input('quit - press Enter-key')
    time.sleep(5)
    print('Parent thread exiting.')

parent()
```

Подробно рассмотрим формат используемой функции для запуска дочерних потоков:

_thread.start_new_thread(<ссылка_на_функцию>, (<список_параметров>))

Существует несколько способов передачи параметра – ссылки на функцию: передача имени метода, передача lambda-выражения и передача выражения вызова связанного метода. Подробно рассмотрим все три способа.

```
import _thread

def function(x):
    print('%d ^ 12 = %d\n' % (x, x**12))

class AnyClass:
    def __init__(self, x):
        self.x = x
    def function(self):
        print('%d ^ 12 = %d\n' % (self.x, self.x**12))
# END class AnyClass

# передача параметра - ссылки
_thread.start_new_thread(function, (3,))
# передача параметра - lambda-выражения
_thread.start_new_thread((lambda: function(3)), ())
obj = AnyClass(3)
# передача параметра - ссылки на связанный метод
_thread.start_new_thread(obj.function, ())
print('Main process exiting.\n')
```

Связанный метод – это ссылка для вызова метода через экземпляр класса. Получается, что ссылка на метод связана со ссылкой на экземпляр класса. Данный механизм подразумевает предварительную настройку экземпляра класса для выполнения необходимой работы. В классе может быть реализовано несколько

методов и, соответственно, последовательно вызваны через *_thread.start_new_thread* могут быть несколько связанных методов одного класса.

Другой особенность механизма связанного метода является тот факт, что функции *_thread.start_new_thread* передается ссылка на оригинальный экземпляр объекта, а не его копия. Это позволяет обеспечивать дополнительные механизмы для взаимодействия потоков чрез атрибуты экземпляра класса. В самом простом случае мы имеем возможность для разбиения большой задачи на набор отдельных подзадач и выполнения этих подзадач в параллельных потоках исполнения. Рассмотрим простой пример нахождения значения суммы элементов числового от 1 до 10 с шагом равным единице.

```
import _thread, time

class AnyClass:
    def __init__(self, x=0):
        self.x = x
    def func(self, x):
        self.x += x
    def result(self):
        print(self.x)
# END class AnyClass

obj = AnyClass()
for i in range(1, 11):
    _thread.start_new_thread(obj.func, (i,))
time.sleep(5)
obj.result()
print('Main process exiting.\n')
```

Существенным недостатком данного примера является использование метода `time.sleep(5)` для формирования задержки в главном потоке, позволяющем «дождаться» окончания работы дочерних потоков. Величина задержки была сформирована случайным образом. Если указанный интервал ожидания окажется недостаточным для выполнения вычислений, то будет получен заведомо неверный результат.

Можно поэкспериментировать с количеством запускаемых потоков и величиной задержки. Например, найти сумму чисел от 1 до 100, формируя задержу всего в 1 микро секунду.

```
obj = AnyClass()
for i in range(1, 101):
    _thread.start_new_thread(obj.func, (i,))
time.sleep(0.000001)
obj.result()
print('Main process exiting.\n')
```

Однако увеличение количества запускаемых потоков приводит к исключению – отказу со стороны операционной системы. И, всего-то навсего, нужно было запустить каких-то 1000 потоков =).

```
for i in range(1, 1001):
    _thread.start_new_thread(obj.func, (i,))
```

```
Traceback (most recent call last):
  File "C:/Temp/test_thread_02c.py", line 14, in <module>
    _thread.start_new_thread(obj.func, (i,))
RuntimeError: can't start new thread
```

Внедрим в метод класса *func()* отладочную информацию и проанализируем действительное количество потоков, с запуском которых система справилась успешно. Вот полный текст измененного примера:

```
import _thread, time

class AnyClass:
    def __init__(self, x=0):
        self.x = x
    def func(self, x):
        print('Start thread number %d' % x)
        self.x += x
    def result(self):
        print(self.x)
# END class AnyClass

obj = AnyClass()
for i in range(1, 1001):
    _thread.start_new_thread(obj.func, (I,))
time.sleep(0.000001)
obj.result()
print('Main process exiting.\n')
```

В моём случае сбой произошел при запуске 688-го потока. Сами понимаете, визуально анализировать весь мусор, выводимый на экран, – это неблагодарное занятие. Однако попытка использования разделяемой памяти, например, списка для ведения протокола запуска дочерних потоков, тоже может привести к проявлению ошибки, так как у нас нет гарантии что два или более потока не попытаются обратиться к списку в один и тот же момент времени. Но, как ни странно, использование списка внутри объекта класса и отражение в нем события запуска потока привели к тому, что все 1000 дочерних потоков были запущены успешно. Но успешно они были запущены только один раз, повторные запуски примера привели к выбросу объекта исключения.

Тестирование финальной версии кода примера показали, что максимальное количество запускаемых параллельных потоков достаточно велико, но изменяется случайным образом. Вот этот код.

```
import _thread, time

class AnyClass:
    def __init__(self, x=0):
        self.protocol = []  # пустой список
        self.x = x
    def func(self, x):
        self.protocol.append(x)
        self.x += x
    def result(self):
        print(self.x)
# END class AnyClass

obj = AnyClass()
try:
    for i in range(1, 1001):
        _thread.start_new_thread(obj.func, (i,))
except RuntimeError:
    pass
time.sleep(0.000001)
obj.result()
print('Count %d' % len(obj.protocol))
print('Main process exiting.\n')
```

Давайте воспользуемся полученными навыками для решения практической задачи – численному интегрированию. Найдем приближенное значение определенного интеграла $\int_4^8 \frac{x \sin x dx}{2}$ методом правых прямоугольников. Код программы приведен ниже.

```
import _thread, time, math

class Calc:
    def __init__(self, x=0):
        self.x = x  # 'аккумулятор' результирующего значения
    def func(self, a, b, step):
        i = a
        res = 0  # 'аккумулятор' для формирования значений функции
                 # на отрезке интегрирования
        while i < b:
            # используем метод парвых прямоугольников
            res += step * (i * math.sin(i))/2
            i += step
        self.x += res
    def result(self):
        print(self.x)
# END class AnyClass
```

```
obj = Calc(0)
Xa = 4; Xb = 8; step = 0.00001
Xn = Xb - step
numThreads = 100  # количество параллельных потоков
intervX = (Xb - Xa) / numThreads
try:
    xa = Xa
    xb = Xa + intervX
    for i in range(0, numThreads):
        _thread.start_new_thread(obj.func, (xa, xb, step))
        xa += intervX
        xb += intervX
except RuntimeError:
    pass
time.sleep(10)
obj.result()
print('Main process exiting.\n')
```

Самое интересное заключается в том, что каждый раз при запуске программы будет получена новое значение, в пятнадцатом и следующих знаках, отличающееся от предыдущего. Данный фат объясняется не атомарность операции сложения, используемой при накоплении результата. Это значит, что не все потоки смогли правильно внести свой сформированный результат. Данный недостаток можно исправить, использую при формировании атомарные операции, либо механизмы блокировки, но это будет рассмотрено позже, сейчас же рассмотрим механизм, связанный с использование в качестве аккумулятора списка. Изменения вносятся только в класс *Calc*.

```
class Calc:
    def __init__(self, x=0):
        self.x = x  #
        self.arr = []  # в список будут заноситься найденные значения
                       # по отрезкам интегрирования
    def func(self, a, b, step):
        i = a
        res = 0  # 'аккумулятор' для значений функции
                 # на отрезке интегрирования
        while i < b:
            res += step * (i * math.sin(i))/2
            i += step
        self.arr.append(res)
    def result(self):
        self.x = sum(self.arr)  # суммируем найденные значения
        print(self.x)
# END class AnyClass
```

Для анализа свойств операции на предмет её атомарности либо, наоборот, не атомарности, в языке Python используются средства модуля *dis*. Например.

```
>>> import dis
>>> def incr():
     global x
     x += 1


>>> dis.dis(incr)
  3           0 LOAD_GLOBAL              0 (x)
              2 LOAD_CONST               1 (1)
              4 INPLACE_ADD
              6 STORE_GLOBAL             0 (x)
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
>>>
```

Метод *dis()* выполняет «дизассемблирование», то есть позволяет получить нечто вроде набора атомарных команд, соответствующих той или иной последовательности операций языка *Python*.

Однако мы немного увлеклись и впопыхах успели наступить грабли – выявить ряд фундаментальных проблем, связанных с многопоточным программированием. «Критическая секция» или разделяемые несколькими потоками данные – это одна из таких проблем. Одно из её решений связано с использованием таких инструментов синхронизации, как объекты блокировки. В модуле *_thread* определены объекты блокировки, реализующие интерфейс мьютексов, то есть являющиеся двоичными семафорами. Такой объект можно «***приобрести***» и «***освободить***». Приобретается блокировка – определяются права доступа к объекту. Сам по себе объект блокировки позволяет лишь определить логику доступа к тому объекту, но ни коим образом не контролирует доступ к самому объекту. То есть, если какой-нибудь поток не будет работать по определенным правилам доступа к объекту, то никакой пользы от таких правил нет. Давайте рассмотрим эти правила.

Объект блокировки создается при помощи метода *_thread.allocate_lock()*, который может быть «приобретен» при помощи вызова метода *acquire()* и освобожден вызова *release()*.

```
import _thread, time

def func1(numb, value):
    for i in range(value):
        time.sleep(1)
        mutex.acquire()
        print('thread[%d] => %d' % (numb, i))
        mutex.release()

def main():
    global mutex
```

```
    mutex = _thread.allocate_lock()
    for i in range(5):
        _thread.start_new_thread(func1, (i,5))
    time.sleep(6)
    print('Main process exiting.\n')

if __name__ == '__main__':
    mutex = None
    main()
```

Запустите данный пример, и вы увидите, что вывод осуществляется корректно. Но вспомним о том, что если в системе будет запущен поток, не реализующий правила синхронизации, то результат работы будет искажен.

```
import _thread, time

def func1(numb, value):
    for i in range(value):
        time.sleep(1)
        mutex.acquire()
        print('thread[%d] => %d' % (numb, i))
        mutex.release()

def func2(numb, value):
    """ Метод, нарушающий правила синхронизации доступа к stdout """
    for i in range(value):
        time.sleep(1)
        print('thread[%d] => %d' % (numb, i))

def main():
    global mutex
    mutex = _thread.allocate_lock()
    for i in range(5):
        _thread.start_new_thread(func1, (i,3))
        _thread.start_new_thread(func2, (i,3))
    time.sleep(6)
    print('Main process exiting.\n')

if __name__ == '__main__':
    mutex = None
    main()
```

Объект блокировки можно сравнить с неким аналогом функции *time.sleep()*, либо *os.system('pause')*, которая принудительно останавливает выполнение потока, пытающегося выполнить «***захват***» объекта блокировки, если тот уже «***захвачен***» другим потоком. Продолжение работы ожидающего потока возможно только в случае, когда объект блокировки будет «***освобожден***».

Проверка состояния объекта блокировки выполняется при помощи метода *locked()* (закрыт), возвращающий логическое значение *True* для «***захваченного***» объекта синхронизации.

Существенным недостатком модуля *_thread* по сравнению с модулем *threading* является отсутствие встроенных средств для выполнения процедуры ожидания завершения работы дочерних потоков. Возможно несколько вариантов организации процесса ожидания работы дочерних потоков, основанных на использовании объектов блокировки.

Рассмотрим следующие примеры, реализующие данные варианты.

```
import _thread

# объект синхронизации, используемый для синхронизации доступа
# из дочерних потоков к разделяемому ресурсу – потоку sys.stdout
mutex = _thread.allocate_lock()
# список объектов синхронизации, используемых главным потоком
# для анализа состояния дочерних потоков
arraymutex = [_thread.allocate_lock() for i in range(5)]

def func1(numb, value):
    # функция, запускаемая в дочернем потоке
    # numb – номер дочернего потока
    # value – количество прогонов цикла внутри потока
    for i in range(value):
        mutex.acquire()
        print('thread[%d] => %d' % (numb, i))
        mutex.release()
    # захват объекта блокировки главного потока
    arraymutex[numb].acquire()

def main():
    # цикл запуска дочерних потоков
    for i in range(5):
        _thread.start_new_thread(func1, (i,5))

    # цикл перебора элементов списка объектов блокировки
    for mutex in arraymutex:
        # пока блокировка не снята - pass...
        # главный поток 'ожидает захвата' всех элементов
        # списка объектов блокировки
        while not mutex.locked(): pass
    # главный поток достигает данной точки программы только после
    # того, как будут 'захвачены' все объекты списка arraymutex
    print('Main thread exiting')

if __name__ == '__main__':
    main()
# END ------------------------------------------------------------
```

```
import _thread
mutex = _thread.allocate_lock()  # объект синхронизации
# список признаков состояния дочерних потоков, используемых
# главным потоком
arraymutex = [False] * 5

def func1(numb, value):
    for i in range(value):
        mutex.acquire()
        print('thread[%d] => %d' % (numb, i))
        mutex.release()
    # дочерний поток номер numb изменяет 'свой' элемент
    # списка признаков состояния дочерних потоков
    arraymutex[numb] = True

def main():
    # цикл запуска дочерних потоков
    for i in range(5):
        _thread.start_new_thread(func1, (i,5))

    # цикл перебора элементов списка признаков блокировки
    # пока все дочерние потоки не изменят состояние связанного
    # с ними элемента списка признаков - pass...
    while False in arraymutex: pass
    print('Main thread exiting')

if __name__ == '__main__':
    main()
# END ---------------------------------------------------------------
```

```
import _thread, time
# объект синхронизации для доступа к sys.stdout
mutex = _thread.allocate_lock()
numbthreads = 5  # количество запускаемых дочерних потоков
# список объектов блокировки потоков, используемых главным потоком
# для анализа состояния дочерних потоков
arraymutex = [_thread.allocate_lock() for i in range(numbthreads)]

def func1(numb, value, mutex):
    for i in range(value):
        # для взаимодействия  с объектом синхронизации
        # используется менеджер контекста, позволяющий скрыть вызовы
        # методов acquire() и release()
        with mutex:  # вызов менеджера контекста объекта блокировки
            # скрытый вызов mutex.acquire()
            print('thread[%d] => %d' % (numb, i))
            # крытый вызов mutex.release()
    # После завершения процедуры вывода данных в sys.stdout,
    # дочерний поток номер numb изменяет 'свой' элемент
```

```
    # списка объектов блокировки
    arraymutex[numb].acquire()

def main():
    # цикл запуска дочерних потоков
    for i in range(5):
        _thread.start_new_thread(func1, (i,5, mutex))

    # цикл перебора элементов списка объектов блокировки,
    # пока все дочерние потоки не изменят состояние связанного
    # с ними элемента списка, ждать...
    while not all(mutex.locked() for mutex in arraymutex):
time.sleep(0.1)
    print('Main thread exiting')

if __name__ == '__main__':
    main()
# END ---------------------------------------------------------------
```

- **Использование модуля _threading**

Модуль *_threading* предоставляет высокоуровневый интерфейс для работы с потоками. Внутри данного модуля используются скрытые вызовы средств модуля *_thread*, предназначенные для создания объектов – потоков, объектов синхронизации и управления ими.

```
import threading

def worker():
    print('Hello fron Worker')

threads = []
for i in range(10):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

- **Модель взаимодействия потоков**

При разработке приложений с графическим интерфейсом пользователя (*GUI*), интерфейсу отводится роль «клиента», обеспечивающего процесс взаимодействия с «сервером». Таким образом, средства построения и отображения *GUI* должны запускаться в дочернем потоке.

Задача «сервера» сводится к управлению работой приложения, запуску всех зависимых приложений в дочерних потоках и контролю и обработке возникающих ошибок.

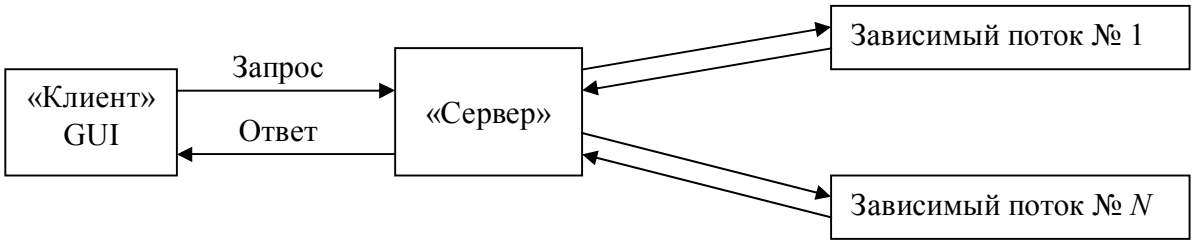

Рисунок 1. Модель взаимодействия потоков в приложении с графическим интерфейсом пользователя

Модель, представленная на рис.1 предполагает следующий обобщенный сценария работы приложения:

1. Запуск приложения.
2. Запуск главной функции приложения «сервера».
3. Анализ ошибок старта главного потока приложения «сервера». Если старт приложения был произведен с ошибками, то работа приложения аварийно завершается с выдачей соответствующего приложения.
4. Запуск зависимых дочерних потоков, предназначенных для выполнения операций для работы с данными.
5. Если дочерние потоки стартовали успешно, то, как один из вариантов, переходим к циклу обработки событий интерфейса пользователя

Базовый каркас для построения такого приложения формируется из следующей обобщенной схемы.

```
def main():
    '''
    Главная функция - метод приложения, отвечающий за запуск
    'сервера' и 'клиента' приложения.
    '''
    pass

if __name__ == '__main__':
    main()
```