

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №10
дисциплины «Анализ данных»

Выполнил:
Кожуховский Виктор Андреевич
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных систем
», очная форма обучения

(подпись)

Руководитель практики:
Воронкин Роман Александрович

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Тема: Синхронизация потоков в языке программирования Python

Цель: приобретение навыков использования примитивов синхронизации в языке программирования Python версии 3.x.

Порядок выполнения работы:

1. Изучил теоретический материал работы.
2. Создал общедоступный репозиторий на GitHub, в котором использована лицензия MIT и язык программирования Python.
3. Выполнил клонирование созданного репозитория.
4. Дополнил файл .gitignore необходимыми правилами для работы с IDE PyCharm.
5. Организовал свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создал проект в папке репозитория.
8. Выполнил индивидуальное задание.

Разработать приложение, в котором выполнить решение вычислительной задачи (например, задачи из области физики, экономики, математики, статистики и т. д.) с помощью паттерна “Производитель-Потребитель”.

Задача:

В буфете железнодорожной станции обслуживают клиентов два продавца. Интенсивность обслуживания одним продавцом составляет 0,5 человека в минуту. Посетители приходят в буфет со средним интервалом в 1 минуту. Если в момент прихода клиента все продавцы заняты, клиент встает в очередь, которая не может превышать 5 человек. Посетитель, не попавший в очередь, уходит в другой буфет. Определить вероятность отказа посетителю в обслуживании.

$$N = 2, l = 5, \lambda = 0.5 \text{ чел/мин}, T = 0.5 \text{ мин}, \mu = 1 / 0.5 = 2 \text{ мин}^{-1}$$

$$\rho = \lambda / \mu = 0.5 / 2 = 0.25, \rho / N = 0.25 / 2 = 0.125$$

$$P_0 = \left[\sum_{k=0}^N \frac{p^k}{k!} + \frac{p^{N+1}}{N \cdot N!} * \frac{1 - (p/N)^l}{1 - p/N} \right]^{-1}$$

$$P_{omk} = \left(\frac{p}{N}\right)^l * \frac{p^N}{N!} * P_0$$

Код решения:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import math
from threading import Thread, Barrier

barrier = Barrier(2)
result_p_0 = []
result_p_omk = []

def P_0(p, N, l):
    sum_series = sum([(p**k) / math.factorial(k) for k in range(N+1)])
    extra_term = ((p**(N+1)) / (N * math.factorial(N))) * \
        ((1 - (p/N)**l) / (1 - p/N))
    result_p_0.append((sum_series + extra_term)**(-1))
    barrier.wait()

def P_omk(p, N, l):
    barrier.wait()
    result_p_omk.append(
        ((p/N)**l) * ((p**N) / math.factorial(N)) * result_p_0[0])

def main():
    p = 0.25 # интенсивность трафика
    N = 2    # Количество продавцов
    l = 5    # Максимум в очереди

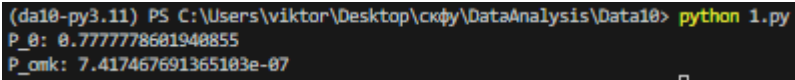
    thread1 = Thread(target=P_0, args=(p, N, l))
    thread2 = Thread(target=P_omk, args=(p, N, l))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    print("P_0:", result_p_0[0])
    print("P_omk:", result_p_omk[0])

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



```
(da10-py3.11) PS C:\Users\viktor\Desktop\скф\DataAnalysis\Data10> python 1.py
P_0: 0.7777778601940855
P_omk: 7.417467691365103e-07
```

Рисунок 1. Выполнение кода индивидуального задания 1

Для своего индивидуального задания лабораторной работы 2.23 необходимо организовать конвейер, в котором сначала в отдельном потоке вычисляется значение первой функции, после чего результаты вычисления должны передаваться второй функции, вычисляемой в отдельном потоке.

Потоки для вычисления значений двух функций должны запускаться одновременно.

Код:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import math
from threading import Thread, Barrier

epsilon = 1e-7
barrier = Barrier(3)

def func(x, result):
    sum = 0
    n = 0
    term = 1
    while abs(term) > epsilon:
        sum += term
        n += 1
        term = (-1)**n * x**(2 * n) / math.factorial(n)
    result.append(sum)
    barrier.wait()

def func2(x, result):
    sum = 0
    n = 1
    while True:
        term = 1 / (2 * n - 1) * ((x - 1) / (x + 1))**(2 * n - 1)
        if abs(term) < epsilon:
            break
        else:
            sum += term
            n += 1
    result.append(sum)
    barrier.wait()

def test(result1, result2):
    barrier.wait() # Ожидание, пока обе функции завершат свою работу, чтобы потом проверить правильность
результатов
    sum_func = result1[0]
    sum_func2 = result2[0]

    test1 = math.exp(-(-0.7)**2)
    test2 = 1/2 * math.log(0.6)

    print(f'Результат функции 1: {sum_func}')
    print(f'Контрольное значение для функции 1: {test1}')
    print(f'Результат функции 2: {sum_func2}')
    print(f'Контрольное значение для функции 2: {test2}')

    if abs(sum_func - test1) < epsilon:
        print("func: Верно.")
    else:
        print("func: Неверно.")

    if abs(sum_func2 - test2) < epsilon:
        print("func2: Верно.")
    else:
        print("func2: Неверно.")

def main():
    result1 = []
    result2 = []

    thread1 = Thread(target=func, args=(-0.7, result1))
    thread2 = Thread(target=func2, args=(0.6, result2))
    thread3 = Thread(target=test, args=(result1, result2))

    thread1.start()
    thread2.start()
```

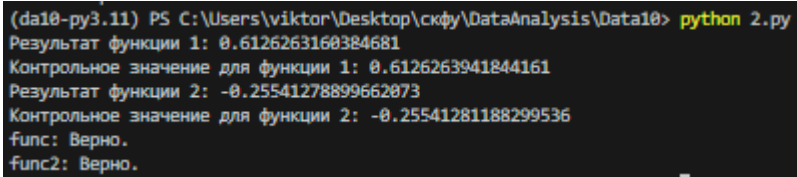
```

thread3.start()

thread1.join()
thread2.join()
thread3.join()

if __name__ == "__main__":
    main()

```



```

(da10-py3.11) PS C:\Users\viktor\Desktop\сф\DataAnalysis\Data10> python 2.py
Результат функции 1: 0.6126263160384681
Контрольное значение для функции 1: 0.6126263941844161
Результат функции 2: -0.25541278899662873
Контрольное значение для функции 2: -0.25541281188299536
func: Верно.
func2: Верно.

```

Рисунок 2. Выполнение кода индивидуального задания 2

9. Зафиксировал сделанные изменения в репозитории.

10. Добавил отчет по лабораторной работе в формате PDF в папку doc репозитория. Зафиксировал изменения.

11. Выполнил слияние ветки для разработки с веткой master/main.

12. Отправил сделанные изменения на сервер GitHub.

Контрольные вопросы:

1. Каково назначение и каковы приемы работы с Lock-объектом.

Lock-объект может находиться в двух состояниях: захваченное (заблокированное) и не захваченное (не заблокированное, свободное). После создания он находится в свободном состоянии. Для работы с Lock-объектом используются методы `acquire()` и `release()`. Если Lock свободен, то вызов метода `acquire()` переводит его в заблокированное состояние. Повторный вызов `acquire()` приведет к блокировке инициировавшего это действие потока до тех пор, пока Lock не будет разблокирован каким-то другим потоком с помощью метода `release()`. Вывоз метода `release()` на свободном Lock-объекте приведет к вы抛су исключения `RuntimeError`. Lock-объекты поддерживают протокол менеджера контекста, это позволяет работать с ними через оператор `with`.

2. В чем отличие работы с RLock-объектом от работы с Lock-объектом.

В отличие от рассмотренного выше Lock-объекта RLock может освободить только тот поток, который его захватил. Повторный захват потоком уже захваченного RLock-объекта не блокирует его. RLock-объекты

поддерживают возможность вложенного захвата, при этом освобождение происходит только после того, как был выполнен `release()` для внешнего `acquire()`. Сигнатуры и назначение методов `release()` и `acquire()` `RLock`-объектов совпадают с приведенными для `Lock`, но в отличие от него у `RLock` нет метода `locked()`. `RLock`-объекты поддерживают протокол менеджера контекста.

3. Как выглядит порядок работы с условными переменными?

Основное назначение условных переменных – это синхронизация работы потоков, которая предполагает ожидание готовности некоторого ресурса и оповещение об этом событии. Наиболее явно такой тип работы выражен в паттерне `Producer-Consumer` (Производитель – Потребитель). Условные переменные для организации работы внутри себя используют `Lock`-или `RLock`-объекты, захватом и освобождением которых управлять не придется, хотя и возможно, если возникнет такая необходимость.

Порядок работы с условными переменными выглядит так:

- На стороне `Consumer`'а: проверить доступен ли ресурс, если нет, то перейти в режим ожидания с помощью метода `wait()`, и ожидать оповещение от `Producer`'а о том, что ресурс готов и с ним можно работать. Метод `wait()` может быть вызван с таймаутом, по истечении которого поток выйдет из состояния блокировки и продолжит работу.

- На стороне `Producer`'а: произвести работы по подготовке ресурса, после того, как ресурс готов оповестить об этом ожидающие потоки с помощью методов `notify()` или `notify_all()`. Разница между ними в том, что `notify()` разблокирует только один поток (если он вызван без параметров), а `notify_all()` все потоки, которые находятся в режиме ожидания.

4. Какие методы доступны у объектов условных переменных?

При создании объекта `Condition` вы можете передать в конструктор объект `Lock` или `RLock`, с которым хотите работать. Перечислим методы объекта `Condition` с кратким описанием:

`acquire(*args)` – захват объекта-блокировки.

`release()` – освобождение объекта-блокировки.

`wait(timeout=None)` – блокировка выполнения потока до оповещения о снятии блокировки. Через параметр `timeout` можно задать время ожидания оповещения о снятии блокировки. Если вызвать `wait()` на Условной переменной, у которой предварительно не был вызван `acquire()`, то будет выброшено исключение `RuntimeError`.

`wait_for(predicate, timeout=None)` – метод позволяет сократить количество кода, которое нужно написать для контроля готовности ресурса и ожидания оповещения.

`notify(n=1)` – снимает блокировку с остановленного методом `wait()` потока. Если необходимо разблокировать несколько потоков, то для этого следует передать их количество через аргумент `n`.

`notify_all()` – снимает блокировку со всех остановленных методом `wait()` потоков.

5. Каково назначение и порядок работы с примитивом синхронизации “семафор”?

Реализация классического семафора, предложенного Дейкстрой. Суть его идеи заключается в том, при каждом вызове метода `acquire()` происходит уменьшение счетчика семафора на единицу, а при вызове `release()` – увеличение. Значение счетчика не может быть меньше нуля, если на момент вызова `acquire()` его значение равно нулю, то происходит блокировка потока до тех пор, пока не будет вызван `release()`.

Семафоры поддерживают протокол менеджера контекста.

Для работы с семафорами в Python есть класс `Semaphore`, при создании его объекта можно указать начальное значение счетчика через параметр `value`. `Semaphore` предоставляет два метода:

`acquire(blocking=True, timeout=None)` – если значение внутреннего счетчика больше нуля, то счетчик уменьшается на единицу и метод возвращает `True`. Если значение счетчика равно нулю, то вызвавший данный метод поток блокируется, до тех пор, пока не будет кем-то вызван метод

release(). Дополнительно при вызове метода можно указать параметры blocking и timeout, их назначение совпадает с acquire() для Lock.

release() – увеличивает значение внутреннего счетчика на единицу.

Существует ещё один класс, реализующий алгоритм семафора BoundedSemaphore, в отличие от Semaphore, он проверяет, чтобы значение внутреннего счетчика было не больше того, что передано при создании объекта через аргумент value, если это происходит, то выбрасывается исключение ValueError.

С помощью семафоров удобно управлять доступом к ресурсу, который имеет ограничение на количество одновременных обращений к нему (например, количество подключений к базе данных и т.п.)

6. Каково назначение и порядок работы с примитивом синхронизации “событие”?

События по своему назначению и алгоритму работы похожи на рассмотренные ранее условные переменные. Основная задача, которую они решают – это взаимодействие между потоками через механизм оповещения. Объект класса Event управляет внутренним флагом, который сбрасывается с помощью метода clear() и устанавливается методом set(). Потоки, которые используют объект Event для синхронизации блокируются при вызове метода wait(), если флаг сброшен.

Методы класса Event:

is_set() – возвращает True если флаг находится в взведенном состоянии.

set() – переводит флаг в взведенное состояние.

clear() – переводит флаг в сброшенное состояние.

wait(timeout=None) – блокирует вызвавший данный метод поток если флаг соответствующего Event-объекта находится в сброшенном состоянии. Время нахождения в состоянии блокировки можно задать через параметр timeout.

7. Каково назначение и порядок работы с примитивом синхронизации “таймер”?

Модуль `threading` предоставляет удобный инструмент для запуска задач по таймеру – класс `Timer`. При создании таймера указывается функция, которая будет выполнена, когда он сработает. `Timer` реализован как поток, является наследником от `Thread`, поэтому для его запуска необходимо вызвать `start()`, если необходимо остановить работу таймера, то вызовите `cancel()`.

Конструктор класса `Timer`:

`Timer(interval, function, args=None, kwargs=None)` Параметры:

`interval` – количество секунд, по истечении которых будет вызвана функция `function`. `function` – функция, вызов которой нужно осуществить по таймеру.

`args, kwargs` – аргументы функции `function`.

Методы класса `Timer`:

`cancel()` – останавливает выполнение таймера

Пример работы с таймером:

8. Каково назначение и порядок работы с примитивом синхронизации “барьер”?

Он позволяет реализовать алгоритм, когда необходимо дождаться завершения работы группы потоков, прежде чем продолжить выполнение задачи.

Параметры:

`parties` – количество потоков, которые будут работать в рамках барьера.

`action` – определяет функцию, которая будет вызвана, когда потоки будут освобождены (достигнут барьера).

`timeout` – таймаут, который будет использовать как значение по умолчанию для методов `wait()`.

Свойства и методы класса:

`wait(timeout=None)` – блокирует работу потока до тех пор, пока не будет получено уведомление либо не пройдет время указанное в `timeout`.

`reset()` – переводит `Barrier` в исходное (пустое) состояние. Потокам, ожидающим уведомления, будет передано исключение `BrokenBarrierError`.

`abort()` – останавливает работу барьера, переводит его в состояние “разрушен” (`broken`). Все текущие и последующие вызовы метода `wait()` будут завершены с ошибкой с выбросом исключения `BrokenBarrierError`.

`parties` – количество потоков, которое нужно для достижения барьера.

`n_waiting` – количество потоков, которое ожидает срабатывания барьера.

`broken` – значение флага равное `True` указывает на то, что барьер находится в “разрушенном” состоянии.

9. Сделайте общий вывод о применении тех или иных примитивов синхронизации в зависимости от решаемой задачи.

Выбор примитива синхронизации зависит от задачи:

`Lock/RLock` для эксклюзивного доступа.

Условные переменные для ожидания изменения состояния.

Семафоры для ограничения числа активных потоков.

События для сигнализации состояний.

Таймеры для выполнения действий по времени.

Барьеры для координации групп потоков.