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

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №8
дисциплины
«Объектно–ориентированное программирование»
Вариант 13

Выполнил:
Рябинин Егор Алексеевич
3 курс, группа ИВТ–б–о–23–2,
09.03.01 «Информатика и вычислительная техника»,
направленность (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем», очная форма обучения

_____
(подпись)

Проверил:
Доцент департамента цифровых, робототехнических систем и электроники института перспективной инженерии
Воронкин Роман Александрович
_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2025 г

Тема: Синхронизация потоков в языке программирования Python.

Цель: Приобретение навыков использования примитивов синхронизации в языке программирования Python версии 3.13.3.

Порядок выполнения работы:

Ссылка на репозиторий:

https://github.com/bohemiaaaaa/Lab8_Object-oriented-programming

Задание №1. С использованием многопоточности для заданного значения x 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 двух функций должны запускаться одновременно. Значение суммы рядов должны быть вычислены с точностью $\varepsilon = 10^{-7}$.

$$S = \sum_{n=1}^{\infty} \frac{1}{(2n-1)x^{2n-1}} = \frac{1}{x} + \frac{1}{3x^3} + \frac{1}{5x^5} + \ldots;$$

$$x = 3;\ y = \frac{1}{2}\ln\frac{x+1}{x-1}.$$

Листинг программы:

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


import math
import threading
from queue import Queue


def series_term(n: int, x: float) -> float:
    return 1.0 / ((2 * n - 1) * (x ** (2 * n - 1)))


def calculate_series(x: float, eps: float, out_queue: Queue) -> None:
    total_sum = 0.0
    n = 1

    while True:
        term = series_term(n, x)

        if abs(term) < eps:
            break

        total_sum += term
        n += 1

    out_queue.put(total_sum)
```

```python
def calculate_control(x: float, eps: float, in_queue: Queue) -> None:
    series_sum = in_queue.get()

    control_value = 0.5 * math.log((x + 1) / (x - 1))
    diff = abs(series_sum - control_value)

    print("Результаты вычислений:")
    print(f"Сумма ряда S = {series_sum:.10f}")
    print(f"Контрольное значение y = {control_value:.10f}")
    print(f"|S - y| = {diff:.2e}")

    if diff < eps:
        print(f"Точность достигнута: |S - y| < ε = {eps}")
    else:
        print(f"Точность не достигнута: |S - y| ≥ ε = {eps}")


def run_pipeline(x: float, eps: float) -> None:
    print("=" * 50)
    print("Конвейерное вычисление суммы ряда")
    print(f"x = {x}, ε = {eps}")
    print("=" * 50)

    queue = Queue()

    t1 = threading.Thread(
        target=calculate_series,
        args=(x, eps, queue),
        name="SeriesThread",
    )

    t2 = threading.Thread(
        target=calculate_control,
        args=(x, eps, queue),
        name="ControlThread",
    )

    t1.start()
    t2.start()

    t1.join()
    t2.join()
```

```
PS C:\Users\4isto\OOP_lab8> python tasks/task1.py
==================================================
Конвейерное вычисление суммы ряда
x = 3.0, ε = 1e-07
==================================================
Результаты вычислений:
Сумма ряда S = 0.3465735369
Контрольное значение y = 0.3465735903
|S - y| = 5.34e-08
Точность достигнута: |S - y| < ε = 1e-07
```

Рисунок 1 – Результат работы программы

Тесты для написанной программы:

Листинг программы:

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


import math
from queue import Queue

import pytest
from pipeline import calculate_series, series_term


def test_series_term_basic() -> None:
    x: float = 2.0
    n: int = 1
    expected: float = 1 / (1 * 2.0)
    assert series_term(n, x) == expected


@pytest.mark.parametrize(
    "n, x, expected",
    [
        (1, 2.0, 1 / (1 * 2.0)),
        (2, 2.0, 1 / (3 * 8.0)),
        (3, 3.0, 1 / (5 * 243.0)),
    ],
)
def test_series_term_parametrized(n: int, x: float, expected: float) -> None:
    assert series_term(n, x) == expected


def test_calculate_series_puts_value_in_queue() -> None:
    q: Queue[float] = Queue()
    x: float = 2.0
    eps: float = 1e-6
    calculate_series(x, eps, q)
    assert not q.empty()
    value: float = q.get()
    assert value > 0


def test_calculate_series_accuracy() -> None:
    q: Queue[float] = Queue()
    x: float = 2.0
    eps: float = 1e-6
    calculate_series(x, eps, q)
    series_sum: float = q.get()
    control_value: float = 0.5 * math.log((x + 1) / (x - 1))
    assert abs(series_sum - control_value) < 1e-5


def test_series_with_large_x() -> None:
    q: Queue[float] = Queue()
    x: float = 10.0
    eps: float = 1e-6
    calculate_series(x, eps, q)
    series_sum: float = q.get()
    control_value: float = 0.5 * math.log((x + 1) / (x - 1))
    assert abs(series_sum - control_value) < eps
```

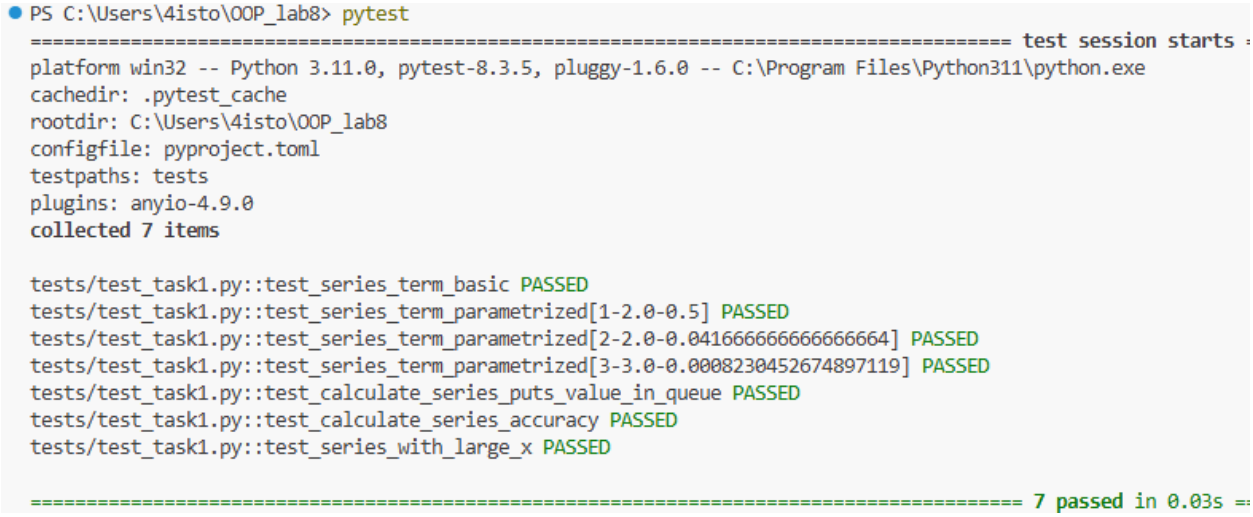

Рисунок 2 – Результат работы тестов

Контрольные вопросы:

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

Назначение Lock-объекта – обеспечение взаимного исключения (mutual exclusion) для защиты общих ресурсов от одновременного доступа нескольких потоков. Приемы работы: метод acquire() для захвата блокировки, release() для освобождения. Рекомендуется использовать контекстный менеджер with для автоматического управления блокировкой. Если Lock уже захвачен, вызов acquire() блокирует поток до его освобождения.

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

RLock (reentrant lock) позволяет одному и тому же потоку многократно захватывать блокировку без само-блокировки, в то время как обычный Lock при повторном вызове acquire() тем же потоком приведет к deadlock. RLock требуется освобождать столько же раз, сколько он был захвачен. Это полезно в рекурсивных функциях или когда код, защищенный блокировкой, вызывает другой метод, который тоже требует той же блокировки.

3. Как выглядит порядок работы с условными переменными?

Порядок работы с условными переменными (Condition): поток-потребитель захватывает связанную блокировку, проверяет условие (например, доступность данных в очереди), и если условие не выполнено,

вызывает wait(), освобождая блокировку и переходя в состояние ожидания. Поток-производитель, подготовив данные, захватывает блокировку, изменяет общее состояние и вызывает notify() или notify_all() для пробуждения ожидающих потоков. После пробуждения потребитель повторно проверяет условие.

4. Какие методы доступны у объектов условных переменных?

У объектов Condition доступны методы: acquire() для захвата связанной блокировки; release() для её освобождения; wait(timeout=None) для ожидания уведомления; wait_for(predicate, timeout=None) для ожидания выполнения предиката; notify(n=1) для пробуждения одного или n ожидающих потоков; notify_all() для пробуждения всех ожидающих потоков. Обычно используются вместе с контекстным менеджером with.

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

Назначение семафора – ограничение количества потоков, одновременно имеющих доступ к ресурсу. Семафор имеет внутренний счётчик. Метод acquire() уменьшает счётчик; если счётчик равен нулю, поток блокируется до вызова release() другим потоком, который увеличивает счётчик. BoundedSemaphore дополнительно контролирует, чтобы счётчик не превышал начальное значение. Используется, например, для ограничения числа подключений к ресурсу.

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

Назначение события (Event) – уведомление одного или нескольких потоков о наступлении какого-либо события. Объект Event содержит внутренний флаг. Метод set() устанавливает флаг, clear() сбрасывает. Потоки вызывают wait() и блокируются, пока флаг не установлен. Метод is_set() проверяет состояние флага. После установки флага все ожидающие потоки разблокируются. Используется для простой координации потоков без сложных условий.

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

Назначение таймера (Timer) – выполнение функции по истечении заданного интервала времени. Таймер создаётся как Timer(interval, function, args, kwargs), затем запускается методом start(). Это отдельный поток, который ждёт указанный интервал, после чего выполняет переданную функцию. Метод cancel() позволяет отменить выполнение, если таймер ещё не сработал. Используется для отложенных задач и периодических действий.

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

Назначение барьера (Barrier) – синхронизация группы потоков в определённой точке выполнения. Барьер инициализируется числом потоков parties. Каждый поток, достигнув барьера, вызывает wait() и блокируется, пока все parties потоков не вызовут wait(). Затем все они одновременно разблокируются. Метод reset() сбрасывает барьер, abort() переводит его в "сломанное" состояние. Используется, когда нужно дождаться готовности всех потоков перед продолжением.

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

Выбор примитива синхронизации зависит от конкретной задачи: Lock и RLock используются для исключительного доступа к ресурсам. Condition применяется в шаблоне производитель-потребитель для ожидания условий. Semaphore ограничивает количество одновременных обращений к ресурсу. Event подходит для простых уведомлений между потоками. Timer используется для отложенного выполнения задач. Barrier синхронизирует группу потоков в общей точке. В реальных приложениях часто комбинируют несколько примитивов для решения комплексных задач многопоточности, выбирая наиболее простой и подходящий по семантике инструмент.

Вывод: в ходе лабораторной работы были приобретены навыки использования примитивов синхронизации в языке программирования Python версии 3.13.3.