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
from time import sleep


def series_term(n: int, x: float) -> float:
    try:
        denominator = (2 * n - 1) * (x ** (2 * n - 1))
        return 1.0 / denominator
    except OverflowError:
        return 0.0


def calculate_series(
    x: float, epsilon: float, result_queue: Queue, stop_event:
threading.Event, cv=None
) -> None:
    total_sum = 0.0
    n = 1

    while not stop_event.is_set():
        term = series_term(n, x)
```

```
        if term == 0.0 or abs(term) < epsilon:
            result_queue.put(total_sum)

            if cv is not None:
                with cv:
                    cv.notify()

            stop_event.set()
            break

        total_sum += term
        n += 1
        sleep(0.001)

    if result_queue.empty():
        result_queue.put(total_sum)
        if cv is not None:
            with cv:
                cv.notify()


def calculate_control_value(
    x: float,
    data_queue: Queue,
    result_queue: Queue,
    stop_event: threading.Event,
    cv: threading.Condition,
) -> None:

    with cv:
        while data_queue.empty() and not stop_event.is_set():
            cv.wait(timeout=0.1)

    if not data_queue.empty():
        series_sum = data_queue.get()

        if x > 1:
            y = 0.5 * math.log((x + 1) / (x - 1))
        else:
            y = float("nan")

        result_queue.put(
            {
                "series_sum": series_sum,
                "control_value": y,
                "difference": abs(series_sum - y),
            }
        )
        stop_event.set()


if __name__ == "__main__":
    x = 3.0
    epsilon = 1e-7

    print("=" * 50)
    print(f"Вычисление суммы ряда для x = {x}")
    print(f"Точность ε = {epsilon}")
    print("-" * 50)

    series_queue = Queue()
    result_queue = Queue()
```

```
    stop_event = threading.Event()
    cv = threading.Condition()

    series_thread = threading.Thread(
        target=calculate_series,
        args=(x, epsilon, series_queue, stop_event, cv),
        name="SeriesCalculator",
    )

    control_thread = threading.Thread(
        target=calculate_control_value,
        args=(x, series_queue, result_queue, stop_event, cv),
        name="ControlCalculator",
    )

    series_thread.start()
    control_thread.start()
    series_thread.join(timeout=5)
    control_thread.join(timeout=5)

    if not result_queue.empty():
        results = result_queue.get()

        print("Результаты вычислений:")
        print(f"Сумма ряда S = {results['series_sum']:.10f}")
        print(f"Контрольное значение y = {results['control_value']:.10f}")
        print(f"Разница |S - y| = {results['difference']:.2e}")

        if results["difference"] < epsilon:
            print(f"Точность достигнута: |S - y| < ε = {epsilon}")
        else:
            print(f"Точность не достигнута: |S - y| ≥ ε = {epsilon}")
    else:
        print("Ошибка: результаты не получены")

    print("=" * 50)
```

```
PS C:\Users\4isto\OOP_lab8> python tasks/task1.py
==================================================
Вычисление суммы ряда для x = 3.0
Точность ε = 1e-07
--------------------------------------------------
Результаты вычислений:
Сумма ряда S = 0.3465735369
Контрольное значение y = 0.3465735903
Разница |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
import threading
```

```
from queue import Queue

import pytest
import task1
from task1 import calculate_control_value, calculate_series, series_term


class TestSeriesTerm:
    def test_series_term_correct_calculation(self):
        result = series_term(1, 3.0)
        expected = 1.0 / 3.0
        assert result == pytest.approx(expected, rel=1e-10)

    def test_series_term_overflow_handling(self):
        result = series_term(1000, 1.5)
        assert result == 0.0


class TestCalculateSeries:
    def test_calculate_series_converges_to_required_precision(self):
        result_queue = Queue()
        stop_event = threading.Event()

        calculate_series(3.0, 1e-7, result_queue, stop_event)

        assert not result_queue.empty()
        series_sum = result_queue.get()
        assert series_sum > 0.0

    def test_calculate_series_with_immediate_stop(self):
        result_queue = Queue()
        stop_event = threading.Event()
        stop_event.set()

        calculate_series(3.0, 1e-7, result_queue, stop_event)


class TestCalculateControlValue:
    def test_calculate_control_value_correctness(self):
        data_queue = Queue()
        result_queue = Queue()
        stop_event = threading.Event()
        cv = threading.Condition()

        test_series_sum = 0.346573590
        data_queue.put(test_series_sum)

        calculate_control_value(3.0, data_queue, result_queue, stop_event,
cv)

        assert not result_queue.empty()
        results = result_queue.get()

        expected_control = 0.5 * math.log((3 + 1) / (3 - 1))
        assert results["control_value"] == pytest.approx(expected_control,
rel=1e-10)
        assert results["series_sum"] == test_series_sum


class TestIntegration:
    def test_full_pipeline_meets_requirements(self):
        x = 3.0
        epsilon = 1e-7
```

```
        series_queue = Queue()
        result_queue = Queue()
        stop_event = threading.Event()
        cv = threading.Condition()

        series_thread = threading.Thread(
            target=task1.calculate_series, args=(x, epsilon, series_queue,
stop_event)
        )

        control_thread = threading.Thread(
            target=task1.calculate_control_value,
            args=(x, series_queue, result_queue, stop_event, cv),
        )

        series_thread.start()
        control_thread.start()

        import time

        time.sleep(0.1)

        with cv:
            cv.notify()

        series_thread.join(timeout=2)
        control_thread.join(timeout=2)

        assert not result_queue.empty()
        results = result_queue.get()

        assert "series_sum" in results
        assert "control_value" in results
        assert "difference" in results

        assert results["difference"] < epsilon
```

```
PS C:\Users\4isto\OOP_lab8> pytest
============================= test session starts =============================
platform win32 -- Python 3.11.0, pytest-8.3.5, pluggy-1.6.0 -- C:\Program Files\Python311\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\4isto\OOP_lab8
configfile: pyproject.toml
testpaths: tests
plugins: anyio-4.9.0
collected 6 items

tests/test_task1.py::TestSeriesTerm::test_series_term_correct_calculation PASSED
tests/test_task1.py::TestSeriesTerm::test_series_term_overflow_handling PASSED
tests/test_task1.py::TestCalculateSeries::test_calculate_series_converges_to_required_precision PASSED
tests/test_task1.py::TestCalculateSeries::test_calculate_series_with_immediate_stop PASSED
tests/test_task1.py::TestCalculateControlValue::test_calculate_control_value_correctness PASSED
tests/test_task1.py::TestIntegration::test_full_pipeline_meets_requirements PASSED

============================== 6 passed in 0.14s ==============================
```

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.