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 № 18

Выполнил:
Текеева Мадина Азрет-Алиевна
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 Р.А

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2025 г.

Тема: Управление потоками в Python.

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

Репозиторий: https://github.com/bickrosss/TM_oop_lab8

Порядок выполнения работы:

1. Пример работы с условной переменной.

```
from threading import Condition, Thread
from queue import Queue
from time import sleep

cv = Condition()
q = Queue()

# Consumer function for order processing
def order_processor(name):
    while True:
        with cv:
            # Wait while queue is empty
            while q.empty():
                cv.wait()
            try:
                # Get data (order) from queue
                order = q.get_nowait()
                print(f"{name}: {order}")
                # If get "stop" message then stop thread
                if order == "stop":
                    break
            except:
                pass
        sleep(0.1)

if __name__ == "__main__":
    # Run order processors
    Thread(target=order_processor, args=("thread 1",)).start()
    Thread(target=order_processor, args=("thread 2",)).start()
    Thread(target=order_processor, args=("thread 3",)).start()

    # Put data into queue
    for i in range(10):
        q.put(f"order {i}")

    # Put stop-commands for consumers
    for _ in range(3):
        q.put("stop")
```

```
    # Notify all consumers
    with cv:
        cv.notify_all()
```

2. Пример работы с семафором.

```
from threading import Thread, BoundedSemaphore
from time import sleep, time

ticket_office = BoundedSemaphore(value=3)

def ticket_buyer(number):
    start_service = time()
    with ticket_office:
        sleep(1)
        print(f"client {number}, service time: {time() - start_service}")

if __name__ == "__main__":
    buyer = [Thread(target=ticket_buyer, args=(i,)) for i in range(5)]

    for b in buyer:
        b.start()
```

3. Пример работы с Event-объектом.

```
from threading import Thread, Event
from time import sleep, time

event = Event()

def worker(name: str):
    event.wait()
    print(f"Worker: {name}")

if __name__ == "__main__":
    # Clear event
    event.clear()

    # Create and start workers
    workers = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(5)]
    for w in workers:
        w.start()

    print("Main thread")
    event.set()
```

4. Пример работы с таймером.

```
from threading import Timer
from time import sleep, time
```

```
timer = Timer(interval=3, function=lambda: print("Message from Timer!"))
timer.start()
```

5. Пример работы с классом Barrier.

```
from threading import Barrier, Thread
from time import sleep, time

br = Barrier(3)
store = []

def f1(x):
    print("Calc part1")
    store.append(x**2)
    sleep(0.5)
    br.wait()

def f2(x):
    print("Calc part2")
    store.append(x*2)
    sleep(1)
    br.wait()

if __name__ == "__main__":
    Thread(target=f1, args=(3,)).start()
    Thread(target=f2, args=(7,)).start()

    br.wait()

    print("Result: ", sum(store))
```

6. Выполнили задание 1.

С использованием многопоточности для заданного значения x необходимо организовать конвейер, в котором сначала в отдельном потоке вычисляется значение первой функции, после чего результат вычисления должны передаваться второй функции, вычисленной в отдельном потоке. Потоки для вычисления значений двух функций должны запускаться одновременно. Значения сумм рядов должны быть вычислены с тонностью ε=10^(-7).

$$S = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1} = x + \frac{x^3}{3} + \frac{x^5}{5} + \ldots;$$

$$x = 0,35;\ y = \ln \sqrt{\frac{1+x}{1-x}}.$$

Рисунок 1. Вариант 18

Листинг программы:

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

import math
from threading import Event, Thread


class ArctanSeriesPipeline:
    """
    Конвейер для вычисления:
    1. Суммы ряда: S = Σ x^(2n+1)/(2n+1) для n от 0 до ∞
    2. Функции: y = ln(sqrt((1+x)/(1-x))) = 0.5 * ln((1+x)/(1-x))

    Ряд представляет собой разложение в ряд Тейлора для 0.5*ln((1+x)/(1-x))
    """

    def __init__(self, x: float = 0.35, eps: float = 1e-7) -> None:
        """
        Инициализация конвейера.

        Args:
            x: Значение аргумента (по умолчанию 0.35 из задания)
            eps: Точность вычисления (по умолчанию 1e-7 из задания)
        """
        self.x: float = x
        self.eps: float = eps

        self.series_result: float | None = None
        self.final_result: float | None = None

        # Событие для синхронизации потоков
        self.ready_event: Event = Event()

    def _term(self, n: int) -> float:
        """
        Вычисление n-го члена ряда: x^(2n+1)/(2n+1)

        Args:
```

```
            n: Номер члена ряда (начиная с 0)

        Returns:
            Значение n-го члена ряда
        """
        return (self.x ** (2 * n + 1)) / (2 * n + 1)

    def _first_worker(self, index: int) -> None:
        """
        Первый поток: вычисление суммы ряда S = Σ x^(2n+1)/(2n+1)

        Args:
            index: Идентификатор потока для отладки
        """
        print(f"[Thread {index}] Начало вычисления суммы ряда для x = {self.x}")

        s: float = 0.0
        n: int = 0
        prev_s: float = 0.0

        # Основной цикл суммирования
        while True:
            # Вычисляем текущий член ряда
            a_n = self._term(n)
            s += a_n
            n += 1

            if abs(s - prev_s) < self.eps:
                print(f"[Thread {index}] Достигнута точность ε={self.eps} на
n={n}")
                break

            prev_s = s

            if n > 10000:
                print(f"[Thread {index}] Достигнут предел итераций: {n}")
                break

        self.series_result = s
        print(f"[Thread {index}] Сумма ряда S = {s:.10f} (вычислено {n} членов)")

        self.ready_event.set()

    def _second_function(self) -> float:
        """
        Аналитическое вычисление функции:
        y = ln(sqrt((1+x)/(1-x))) = 0.5 * ln((1+x)/(1-x))

        Returns:
            Точное значение функции
        """
```

```
        return 0.5 * math.log((1 + self.x) / (1 - self.x))

    def _second_worker(self, index: int) -> None:
        """
        Второй поток: вычисление аналитического значения.

        Args:
            index: Идентификатор потока для отладки
        """
        print(f"[Thread {index}] Ожидание результата первой функции...")

        self.ready_event.wait()

        assert self.series_result is not None
        self.final_result = self._second_function()

        print(f"[Thread {index}] Результат второй функции =
{self.final_result:.10f}")

    def run(self) -> None:
        """
        Запуск конвейера вычислений.
        """
        print("=" * 50)
        print("ЗАПУСК КОНВЕЙЕРА ВЫЧИСЛЕНИЙ")
        print("=" * 50)

        t1: Thread = Thread(target=self._first_worker, args=(1,))
        t2: Thread = Thread(target=self._second_worker, args=(2,))

        t1.start()
        t2.start()

        t1.join()
        t2.join()

        print("\n" + "=" * 50)
        print("КОНВЕЙЕР ВЫПОЛНЕН")
        print("=" * 50)

    def get_absolute_error(self) -> float | None:
        """
        Вычисление абсолютной погрешности.

        Returns:
            Абсолютная погрешность или None, если результаты не вычислены
        """
        if self.series_result is not None and self.final_result is not None:
            return abs(self.series_result - self.final_result)
        return None
```

```
    def get_relative_error(self) -> float | None:
        """
        Вычисление относительной погрешности.

        Returns:
            Относительная погрешность или None, если результаты не вычислены
        """
        if self.series_result is not None and self.final_result is not None:
            if self.final_result != 0:
                return abs((self.series_result - self.final_result) /
self.final_result)
        return None

    def __str__(self) -> str:
        """
        Строковое представление задачи.

        Returns:
            Описание задачи с формулами
        """
        return (
            "\n" + "=" * 60 + "\n"
            "ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ: ВЫЧИСЛЕНИЕ С ИСПОЛЬЗОВАНИЕМ
МНОГОПОТОЧНОСТИ\n"
            "=" * 60 + "\n\n"
            "Ряд для вычисления:\n"
            "S = Σ [x^(2n+1) / (2n+1)], n = 0 .. ∞\n"
            f"  при x = {self.x}\n\n"
            "Аналитическое выражение:\n"
            "y = ln(√((1+x)/(1-x))) = 0.5 * ln((1+x)/(1-x))\n\n"
            f"Точность вычислений: ε = {self.eps}\n"
            "=" * 60 + "\n"
        )
```

Листинг вызова:

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

from pipeline_arctan_series import ArctanSeriesPipeline

if __name__ == "__main__":
    # Создаем конвейер с параметрами из задания
    pipeline = ArctanSeriesPipeline(x=0.35, eps=1e-7)

    # Выводим описание задачи
    print(pipeline)

    # Запускаем конвейер
    pipeline.run()
```

```python
print("\n--- РЕЗУЛЬТАТЫ ВЫЧИСЛЕНИЙ ---")

if pipeline.series_result is not None and pipeline.final_result is not None:
    # Выводим результаты с высокой точностью
    print(f"Сумма ряда (приближенно): {pipeline.series_result:.12f}")
    print(f"Аналитическое значение:   {pipeline.final_result:.12f}")

    # Вычисляем и выводим погрешности
    error = pipeline.series_result - pipeline.final_result
    abs_error = abs(error)

    print(f"\nРазность (S - y):          {error:.2e}")
    print(f"Абсолютная погрешность:   {abs_error:.2e}")

    # Вычисляем относительную погрешность
    if pipeline.final_result != 0:
        rel_error = abs_error / abs(pipeline.final_result)
        print(f"Относительная погрешность: {rel_error:.2e}")

    # Проверяем достижение заданной точности
    print(f"\nЗаданная точность: ε = {pipeline.eps}")
    if abs_error < pipeline.eps:
        print("✓ Требуемая точность достигнута!")
    else:
        print("✗ Требуемая точность не достигнута")

# Выводим информацию о событии
print(f"\nСобытие ready_event установлено: {pipeline.ready_event.is_set()}")

print("\n" + "=" * 60)
```

```
================================================
ЗАПУСК КОНВЕЙЕРА ВЫЧИСЛЕНИЙ
================================================
[Thread 1] Начало вычисления суммы ряда для x = 0.35
[Thread 1] Достигнута точность ε=1e-07 на n=7
[Thread 1] Сумма ряда S = 0.3654437434 (вычислено 7 членов)
[Thread 2] Ожидание результата первой функции...
[Thread 2] Результат второй функции = 0.3654437543

================================================
КОНВЕЙЕР ВЫПОЛНЕН
================================================

--- РЕЗУЛЬТАТЫ ВЫЧИСЛЕНИЙ ---
Сумма ряда (приближенно): 0.365443743440
Аналитическое значение:   0.365443754271

Разность (S - y):          -1.08e-08
Абсолютная погрешность:   1.08e-08
Относительная погрешность: 2.96e-08

Заданная точность: ε = 1e-07
✓ Требуемая точность достигнута!

Событие ready_event установлено: True

================================================
```

Рисунок 1. Результат выполнения программы

Вывод: в ходе выполнения лабораторной работы были приобретены навыки использования примитивов синхронизации в языке программирования Python версии 3.x.