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 №7
дисциплины
«Объектно-ориентированное программирование»
Вариант № 3

Выполнил:
Левашев Тимур Рашидович
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 работы: “Управление потоками в Python”

Цель работы: приобретение навыков написания многопоточных приложений на языке программирования python версии 3.x.

Порядок выполнения работы:

Ссылка на Git репозиторий: https://github.com/mazy99/oop_pract_7

1. **Пример 1:** Создание и ожидание завершения работы потоков. Класс Thread.

Листинг программы:

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


from threading import Thread
from time import sleep, time

def func() -> None:
    for i in range(5):
        print(f"from child thread: {i}")
        sleep(0.5)

start_time: float = time()
print("\nБез использования join()\n")
th: Thread = Thread(target=func)
th.start()


for i in range(5):
    print(f"from main thread {i}")
    sleep(1)

end_time: float = time()
execution_time: float = end_time - start_time
print(f"Время выполнения функции без ожидания завершения потока:
{execution_time:.2f}\n")

start_time_2: float = time()
print("С использованием join()\n")
th1: Thread = Thread(target=func)
th1.start()
th2: Thread = Thread(target=func)
th2.start()

for i in range(5):
```

```
    print(f"from main thread {i}")
    sleep(1)

th1.join()
th2.join()

end_time_2: float = time()
execution_time_2: float = end_time_2 - start_time_2
print(f"Время выполнения функции с ожиданием завершения потока:
{execution_time_2:.2f}")
```

```
Без использования join()

from child thread: 0
from main thread 0
from child thread: 1
from main thread 1
from child thread: 2
from child thread: 3
from main thread 2
from child thread: 4
from main thread 3
from main thread 4

 Время выполнения функции без ожидания завершения потока 5.004549741744995
```

Рисунок 1 – Результат работы программы

```
С использованием join()

from child thread: 0
from child thread: 0
from main thread 0
from child thread: 1
from child thread: 1
from main thread 1
from child thread: 2
from child thread: 2
from child thread: 3
from child thread: 3
from main thread 2
from child thread: 4
from child thread: 4
from main thread 3
from main thread 4

 Время выполнения функции с ожиданием завершения потока 5.004159450531006
```

Рисунок 2 – Результат работы программы

2. Пример 2: Создание классов наследников от Thread.

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


from threading import Thread
from time import sleep, time

class CustomThread(Thread):
    def __init__(self, limit: int) -> None:
        Thread.__init__(self)
        self.limit_: int = limit

    def run(self) -> None:
        for i in range(self.limit_):
            print(f"from CustomThread: {i}")
            sleep(0.5)

if __name__ == "__main__":
    start_time: float = time()

    cth = CustomThread(10)
    cth.start()
    cth.join()

    end_time: float = time()
    execution_time: float = end_time - start_time

    print(f"\nВремя выполнения: {execution_time} секунд")
```

```
from CustomThread: 0
from CustomThread: 1
from CustomThread: 2
from CustomThread: 3
from CustomThread: 4
from CustomThread: 5
from CustomThread: 6
from CustomThread: 7
from CustomThread: 8
from CustomThread: 9

Время выполнения: 5.0058958530426025 секунд
```

Рисунок 3 – Результат выполнения программы

3. **Пример 3:** Принудительное завершение потока.

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


from threading import Thread, Lock
from time import sleep, time

lock: Lock = Lock()

stop_thread: bool = False

def infinit_worker() -> None:
    print("Start infinit_worker")
    while True:
        print("---> thread work")
        lock.acquire()

        if stop_thread is True:
            break
        lock.release()
        sleep(0.1)
    print("Stop infinit_worker()")

if __name__ == "__main__":
    th: Thread = Thread(target=infinit_worker)
    th.start()
    sleep(2)
    lock.acquire()
    stop_thread = True
    lock.release()
```

```
Start infinit_worker
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
---> thread work
Stop infinit_worker()
```

Рисунок 4 – Результат выполнения программы

4. Выполнение индивидуального задания: С использованием многопоточности для заданного значения x найти сумму ряда S с точностью члена ряда по абсолютному значению $\varepsilon = 10^{-7}$ и произвести сравнение полученной суммы с контрольными значениями функции y для двух бесконечных рядов.

$$S = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{2^{n+1}} = \frac{1}{2} - \frac{x}{4} + \frac{x^2}{8} - \ldots;$$

$$x = 1,2;\ y = \frac{1}{2+x}.$$

Рисунок 5 – Исходный ряд, значение x и функция для приблизительного вычисления суммы ряда

Листинг программы:

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


from threading import Thread


class Series:

    def __init__(self, x: float = 1.2, eps: float = 10 ** (-7)) -> None:
        self.x = x
        self.eps = eps
        self.lock = None

    def _term(self, n: int) -> float:
        return ((-1) ** n) * (self.x**n) / (2 ** (n + 1))

    def _worker(self, start: int, step: int, result_list: list, index: int) ->
None:
        print(f"[Thread {index}] Старт")
        sum_local = 0.0
        n = start
        a_n = self._term(n)
        count = 0
        while abs(a_n) >= self.eps:
            sum_local += a_n
```

```
            n += step
            a_n = self._term(n)
            count += 1
        result_list[index] = sum_local
        print(f"[Thread {index}] Завершил. Посчитано членов ряда: {count}")

    def calculate(self, num_threads: int = 4) -> float:
        threads = []
        results = [0.0] * num_threads
        for i in range(num_threads):
            thread = Thread(target=self._worker, args=(i, num_threads, results,
i))
            threads.append(thread)
            thread.start()

        for thread in threads:
            thread.join()

        return sum(results)

    def __str__(self) -> str:
        return (
            "Исходный ряд:\n"
            "S = Σ [(-1)^n * x^n / 2^(n+1)],  n = 0 .. ∞\n\n"
            f"Параметры:\n"
            f"x = {self.x}\n"
            f"epsilon = {self.eps}\n\n"
            "Аналитическое выражение суммы:\n"
            "S = 1 / (2 + x)"
        )

    def ex_value(self):
        return 1 / (2 + self.x)
```

Листинг вызова программы:

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


from time import time

from sum_series import Series

if __name__ == "__main__":
    series = Series()

    print(series)
```

```python
start_time = time()
s_numeric = series.calculate(num_threads=2)
end_time = time()

s_exact = series.ex_value()

print("\nРезультаты вычислений:")
print(f"Сумма ряда (численно)     = {s_numeric:.10f}")
print(f"Контрольное значение y    = {s_exact:.10f}")
print(f"Абсолютная погрешность    = {abs(s_numeric - s_exact):.2e}")
print(f"Время вычисления (многопоточно) = {end_time - start_time:.6f} сек")
```

```
Исходный ряд:
S = Σ [(-1)^n * x^n / 2^(n+1)],  n = 0 .. ∞

Параметры:
x = 1.2
epsilon = 1e-07

Аналитическое выражение суммы:
S = 1 / (2 + x)
[Thread 0] Старт
[Thread 0] Завершил. Посчитано членов ряда: 16
[Thread 1] Старт
[Thread 1] Завершил. Посчитано членов ряда: 15

Результаты вычислений:
Сумма ряда (численно)     = 0.3125000415
Контрольное значение y    = 0.3125000000
Абсолютная погрешность    = 4.15e-08
Время вычисления (многопоточно) = 0.000848 сек
(oop-pract-7) PS C:\учеба\ООП\7 лаба\oop_pract_7>
```

Рисунок 6 – Результат выполнения программы с использованием двух потоков

```
Исходный ряд:
S = Σ [(-1)^n * x^n / 2^(n+1)],  n = 0 .. ∞

Параметры:
x = 1.2
epsilon = 1e-07

Аналитическое выражение суммы:
S = 1 / (2 + x)
[Thread 0] Старт
[Thread 0] Завершил. Посчитано членов ряда: 8
[Thread 1] Старт
[Thread 1] Завершил. Посчитано членов ряда: 8
[Thread 2] Старт
[Thread 2] Завершил. Посчитано членов ряда: 8
[Thread 3] Старт
[Thread 3] Завершил. Посчитано членов ряда: 7

Результаты вычислений:
Сумма ряда (численно)     = 0.3125000415
Контрольное значение y    = 0.3125000000
Абсолютная погрешность    = 4.15e-08
```

Рисунок 7 – Результат выполнения программы с использованием четырех потоков

```
Исходный ряд:
S = Σ [(-1)^n * x^n / 2^(n+1)],  n = 0 .. ∞

Параметры:
x = 1.2
epsilon = 1e-07

Аналитическое выражение суммы:
S = 1 / (2 + x)
[Thread 0] Старт
[Thread 0] Завершил. Посчитано членов ряда: 31

Результаты вычислений:
Сумма ряда (численно)     = 0.3125000415
Контрольное значение y    = 0.3125000000
Абсолютная погрешность    = 4.15e-08
```

Рисунок 8 – Результат выполнения программы в однопоточном режиме

Ответы на контрольные вопросы:

1. **Что такое синхронность и асинхронность?**

Синхронность — это режим выполнения кода, при котором операции выполняются последовательно, одна за другой. Каждая следующая операция

начинает выполняться только после завершения предыдущей. Асинхронность — это подход, при котором операции могут быть запущены, и программа не ждёт их завершения, а продолжает выполнять другой код. По завершении операции вызывается callback или срабатывает await. Это позволяет эффективно работать с I/O-задачами, не блокируя основной поток.

2. **Что такое параллелизм и конкурентность?**

Конкурентность — это способ организации выполнения нескольких задач так, что они выполняются за счёт переключения контекста, создавая видимость одновременной работы (например, в одном потоке или ядре процессора). Параллелизм — это фактическое одновременное выполнение нескольких задач на разных ядрах или процессорах. В Python из-за GIL истинный параллелизм потоков для CPU-задач ограничен.

3. **Что такое GIL? Какое ограничение накладывает GIL?**

GIL (Global Interpreter Lock) — это механизм в реализации CPython, который позволяет выполняться только одному потоку Python за раз, даже на многоядерных системах. Это ограничение накладывает основное ограничение на параллельное выполнение CPU-интенсивных задач в потоках. Однако для I/O-операций (сеть, файлы) потоки всё равно полезны, так как во время ожидания I/O GIL отпускается.

4. **Каково назначение класса Thread?**

Класс Thread из модуля threading предназначен для создания и управления потоками выполнения внутри одного процесса. Он позволяет запускать код в отдельном потоке, что даёт возможность выполнять задачи конкурентно, особенно полезно для I/O-операций, не блокируя основной поток.

5. **Как реализовать в одном потоке ожидание завершения другого потока?**

Для этого используется метод join() объекта потока. Вызов thread.join() в другом потоке (чаще в основном) блокирует выполнение до тех пор, пока поток thread не завершит свою работу.

6. **Как проверить факт выполнения потоком некоторой работы?** Можно использовать метод is_alive(), который возвращает True, если поток ещё выполняется. Для более сложной логики контроля используют примитивы синхронизации, например threading.Event: поток устанавливает событие по завершении работы, а другой поток проверяет это событие методом is_set().

7. Как реализовать приостановку выполнения потока на некоторый промежуток времени?

Используется функция time.sleep(seconds) из модуля time. Также можно использовать threading.Timer для запуска функции через заданный интервал, но это создаёт новый поток.

8. Как реализовать принудительное завершение потока?

В Python нет безопасного метода для принудительного завершения потока. Рекомендуемый подход — использовать флаг (например, переменную-событие threading.Event), который поток периодически проверяет и корректно завершает свою работу. Методы suspend(), resume() и stop() устарели или отсутствуют из-за опасности нарушения состояния программы.

9. Что такое потоки-демоны? Как создать поток-демон?

Потоки-демоны — это фоновые потоки, которые не препятствуют завершению основной программы. Если остаются только демон-потоки, интерпретатор завершает их принудительно. Создать поток-демон можно, установив атрибут daemon в True перед запуском (thread.daemon = True) или передав параметр daemon=True в конструктор Thread.

Вывод: в ходе выполнения работы были получены навыки написания многопоточных приложений на языке программирования python версии 3.x.