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 № 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: “Синхронизация потоков в языке программирования python”.

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

Порядок выполнения работы:

Ссылка на Git репозиторий: https://github.com/mazy99/oop_pract_8

1. **Пример 1:** Использование условных переменных (threading.Condition).

Листинг программы:

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


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


cv = Condition()
q = Queue()


def order_processor(name):
    while True:
        with cv:
            while q.empty():
                cv.wait()
            try:
                order = q.get_nowait()
                print(f"{name}: {order}")
                if order == "stop":
                    break
            except:
                pass
            sleep(0.1)

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

    for i in range(10):
```

```
        q.put(f"order {i}")

    for _ in range(3):
        q.put("stop")

    with cv:
        cv.notify_all()
```

```
thread 2: order 0
thread 3: order 1
thread 1: order 2
thread 2: order 3
thread 3: order 4
thread 1: order 5
thread 2: order 6
thread 3: order 7
thread 1: order 8
thread 2: order 9
thread 3: order 4
thread 1: order 5
thread 2: order 6
thread 3: order 7
thread 1: order 8
thread 2: order 9
thread 3: stop
thread 1: stop
thread 2: stop
```

Рисунок 1 – Результат вызова программы

2. **Пример 2:** Использование семафоров (threading.Semaphore).

Листинг программы:

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


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


ticket_office: BoundedSemaphore = BoundedSemaphore(value=3)

def ticket_buyer(number: int) -> None:
    start_service = time()
    with ticket_office:
        sleep(0.1)
```

```
        print(f"client {number}, service time: {time() - start_service}")

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

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

```
client 0, service time: 0.10026264190673828
client 1, service time: 0.10054540634155273
client 2, service time: 0.10051655769348145
client 3, service time: 0.20070385932922363
client 4, service time: 0.20106220245361328
```

Рисунок 2 – Результат выполнения программы

3. **Пример 3:** Использование событий (threading. Event).

Листинг программы:

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


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

event = Event()

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

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

    worker = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(5)]

    for w in worker:
        w.start()

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

```
Main thread
Worker: wrk 0
Worker: wrk 1
Worker: wrk 3
Worker: wrk 2
Worker: wrk 4
```

Рисунок 3 – Результат выполнения программы

4. **Пример 4:** Использование Барьера threading. Barrier.

Листинг программы:

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


from time import sleep
from threading import Barrier, Thread


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(f"Result: {sum(store)}")
```

```
Calc part1
Calc part2
Result: 23
```

Рисунок 4 – Результат выполнения программы

5. **Индивидуальное задание:** 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 $\varepsilon = 10^{-7}$.

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

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

Рисунок 5 – Исходный ряд, значение переменной x, функция приблизительно описывающая сумму ряда

Листинг программы:

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


from threading import Thread, Event

class SeriesPipeline:


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

        self.x: float = x
        self.eps : float = eps

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

        self.ready_event: Event = Event()

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

    def _first_worker(self, index: int) -> None:
        print(f"[Thread {index}] Начало вычисления ряда")
```

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

        while True:
            a_n = self._term(n)
            if abs(a_n) < self.eps:
                break
            s += a_n
            n += 1

        self.series_result = s
        print(f"[Thread-1] Сумма ряда = {s}")
        self.ready_event.set()

    def _second_function(self) -> float:
        return 1 / (2 + self.x)

    def _second_worker(self, index: int) -> None:
        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}")

    def run(self) -> None:
        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()


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

Листинг вызова программы:

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


from pipeline_sum_series import SeriesPipeline


if __name__ == "__main__":
    pipeline = SeriesPipeline(x=1.2)
    print(pipeline)

    pipeline.run()

    print("\n--- Итог ---")
    print(f"Сумма ряда     = {pipeline.series_result}")
    print(f"Аналитическое  = {pipeline.final_result}")
    print(f"Абсолютная погрешность    = {pipeline.series_result -
pipeline.final_result}")
```

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

x = 1.2
epsilon = 1e-07

Аналитическое выражение:
S = 1 / (2 + x)
[Thread 1] Начало вычисления ряда
[Thread-1] Сумма ряда = 0.31250004145136007
[Thread 2] Ожидание результата первой функции
[Thread 2] Результат второй функции = 0.3125

--- Итог ---
Сумма ряда     = 0.31250004145136007
Аналитическое  = 0.3125
Абсолютная погрешность    = 4.145136006661332e-08
```

Рисунок 6 – Результат выполнения программы

Ответы на контрольные вопросы:

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

Назначение Lock (блокировки) — обеспечить эксклюзивный доступ к общему ресурсу (критической секции) только одному потоку в данный момент времени, предотвращая состояние гонки. Приёмы работы:

1) Создать объект lock = threading.Lock()
2) Перед входом в критическую секцию вызвать lock.acquire()
3) После работы в критической секции вызвать lock.release()
4) Или использовать контекстный менеджер with lock: для автоматического захвата и освобождения.

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

Lock — это базовая блокировка, которую один поток может захватить только один раз; повторный вызов acquire() тем же потоком приведёт к deadlock. RLock (реентерабельная блокировка) позволяет одному и тому же потоку захватывать блокировку несколько раз без deadlock, но требуется столько же вызовов release(), сколько было acquire().

3. Как выглядит порядок работы с условными переменными? Условная переменная (Condition) используется для ожидания выполнения какого-либо условия одним потоком, пока другой поток его не изменит. Порядок работы:

1) Поток, ожидающий условие, вызывает cond.acquire(), затем cond.wait(), что временно отпускает блокировку.

2) Другой поток, изменивший условие, вызывает cond.acquire(), затем cond.notify() или cond.notify_all() для пробуждения ожидающих потоков, и cond.release().

3) После пробуждения первый поток повторно захватывает блокировку и проверяет условие.

4. Какие методы доступны у объектов условных переменных? Основные методы:

1) acquire() — захватить блокировку.

2) release() — освободить блокировку.

3) wait(timeout=None) — ожидать уведомления, временно отпуская блокировку.

4) notify(n=1) — пробудить до n ожидающих потоков.

5) notify_all() — пробудить все ожидающие потоки.

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

Назначение семафора — ограничить количество потоков, одновременно имеющих доступ к ресурсу. Счётчик семафора указывает, сколько потоков могут одновременно войти. Порядок работы:

1) Создать семафор: sem = threading.Semaphore(value=N)

2) При входе в защищённую область: sem.acquire() (уменьшает счётчик; если 0 — блокирует)

3) При выходе: sem.release() (увеличивает счётчик)

4) Также есть BoundedSemaphore, который предотвращает случайное превышение начального значения.

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

Событие (Event) используется для оповещения одного или нескольких потоков о наступлении какого-либо события. Поток может ждать события, а другой поток — устанавливать его. Порядок:

1) Создать: event = threading.Event()

2) Ожидать: event.wait() (блокирует поток, пока флаг не будет установлен)

3) Установить флаг: event.set() (пробуждает все ждущие потоки)

4) Сбросить флаг: event.clear()

5) Проверить статус: event.is_set()

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

Таймер (Timer) — это поток, который запускает указанную функцию после истечения заданного промежутка времени. Порядок работы:

1) Создать: timer = threading.Timer(interval, function, args=None, kwargs=None)

2) Запустить: timer.start()

3) При необходимости отменить до срабатывания: timer.cancel() Таймер является подклассом Thread.

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

Барьер (Barrier) используется для синхронизации заданного количества потоков в определённой точке: все потоки должны дождаться друг друга, прежде чем продолжить выполнение. Порядок работы:

1) Создать: barrier = threading.Barrier(parties) — parties количество потоков для ожидания

2) Каждый поток вызывает barrier.wait()

3) Как только последний поток вызовет wait(), все потоки одновременно разблокируются

4) Можно использовать для координации этапов параллельной обработки

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

Общий вывод:

1) Lock/RLock — для эксклюзивного доступа к ресурсу (один поток за раз).

2) Semaphore — для ограничения количества одновременных доступов к ресурсу.

3) Event — для однократного оповещения потоков о наступлении события.

4) Condition — для сложных сценариев ожидания условия с возможностью уведомления.

5) Barrier — для синхронизации нескольких потоков в одной точке.

6) Timer — для отложенного выполнения задачи. Выбор примитива зависит от конкретной задачи: простой мьютекс — Lock, ожидание события — Event, координация группы потоков — Barrier, ограничение пула — Semaphore.

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