Васильева Елизавета Валерьевна АТ-03
Технологии программирования. 13 задание task_02

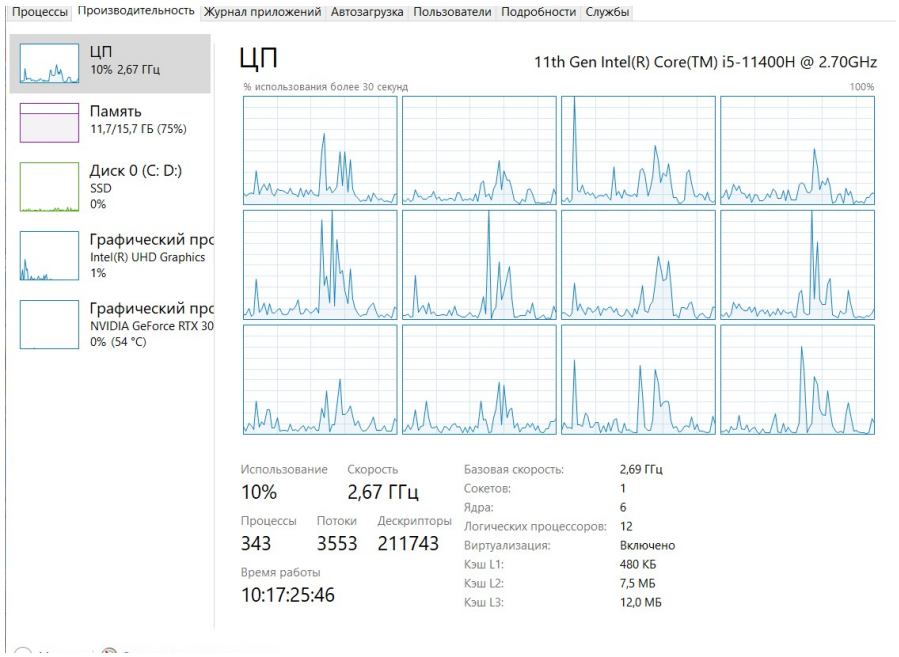
```python
import threading
import time
import multiprocessing
import math

# Функции для АТ-03

# запускать с n = 700003
def fibonacci(n):  # содержимое функции не менять
    """Возвращает последнюю цифру n-го числа Фибоначчи."""
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b % 10  # Возвращаем последнюю цифру


# запускать с f, a, b, n равными соответственно math.sin, 0, math.pi, 20000000
def trapezoidal_rule(f, a, b, n):  # содержимое функции не менять
    """Вычисляет определенный интеграл функции f от a до b методом трапеций с n шагами."""
    h = (b - a) / n
    integral = (f(a) + f(b)) / 2.0
    for i in range(1, n):
        integral += f(a + i * h)
    return integral * h  # Возвращаем значение интеграла
```

```python
def sequence():
    start_time = time.perf_counter()  # время старта
    fib_result = fibonacci(700003)  # вычисление fibonacci от значения 700003
    trap_result = trapezoidal_rule(math.sin, a: 0, math.pi, n: 20000000)  # вычисление трапеций
    end_time = time.perf_counter()  # время окончания

    print(f'fibonacci = {fib_result}')
    print(f'trapezoidal_rule = {trap_result}')
    print(f'sequence time: {end_time - start_time:0.2f} seconds\n')


def thread_fibonacci(result):
    result.append(fibonacci(700003))  # добавляем результат в список

def thread_trapezoidal(result):
    result.append(trapezoidal_rule(math.sin, a: 0, math.pi, n: 20000000))  # добавляем результат в список

def threads():
    start_time = time.perf_counter()  # время старта
    fib_result = []
    trap_result = []

    # Создаем потоки
    fib_thread = threading.Thread(target=thread_fibonacci, args=(fib_result,))
    trap_thread = threading.Thread(target=thread_trapezoidal, args=(trap_result,))

    fib_thread.start()
    trap_thread.start()
```

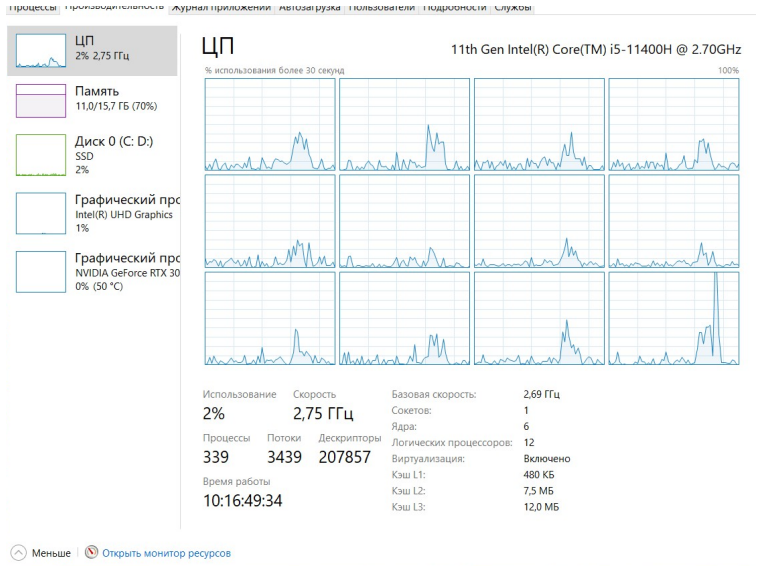
```
    fib_thread.join()  # ожидание завершения потока для fibonacci
    trap_thread.join()  # ожидание завершения потока для трапеций

    end_time = time.perf_counter()  # время окончания

    print(f'fibonacci = {fib_result[0]}')
    print(f'trapezoidal_rule = {trap_result[0]}')
    print(f'threads time: {end_time - start_time:0.2f} seconds\n')


def process_fibonacci(queue):
    queue.put(fibonacci(700003))  # добавляем результат в очередь

def process_trapezoidal(queue):
    queue.put(trapezoidal_rule(math.sin, a: 0, math.pi, n: 20000000))  # добавляем результат в очередь

def processes():
    start_time = time.perf_counter()  # время старта
    queue = multiprocessing.Queue()

    # Создаем процессы
    fib_process = multiprocessing.Process(target=process_fibonacci, args=(queue,))
    trap_process = multiprocessing.Process(target=process_trapezoidal, args=(queue,))

    fib_process.start()
    trap_process.start()

    fib_process.join()  # ожидание завершения процесса для fibonacci
    trap_process.join()  # ожидание завершения процесса для трапеций
```

```
    # Получаем результаты из очереди
    fib_result = queue.get()  # получаем результат из очереди
    trap_result = queue.get()  # получаем результат из очереди

    # Перепутываем результаты
    fib_result, trap_result = trap_result, fib_result  # Обмен значениями

    end_time = time.perf_counter()  # время окончания

    print(f'fibonacci = {fib_result}')
    print(f'trapezoidal_rule = {trap_result}')
    print(f'processes time: {end_time - start_time:0.2f} seconds\n')


if __name__ == '__main__':
    sequence()
    threads()
    processes()

```

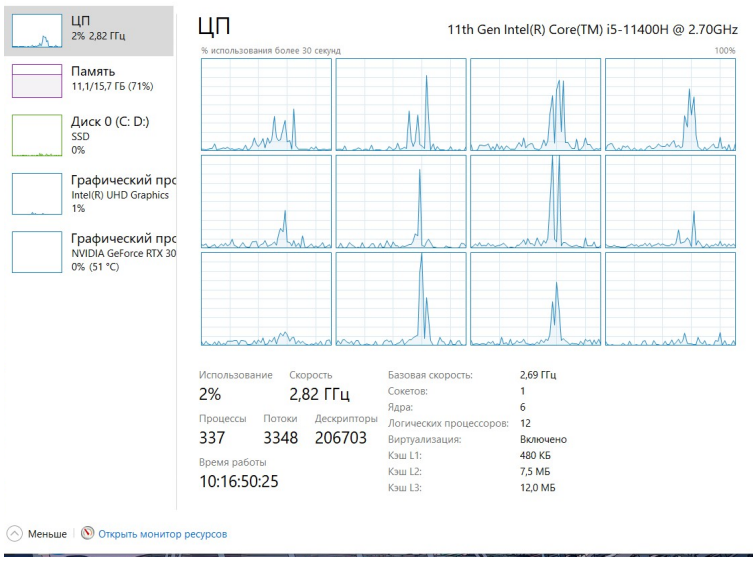
1) последовательно

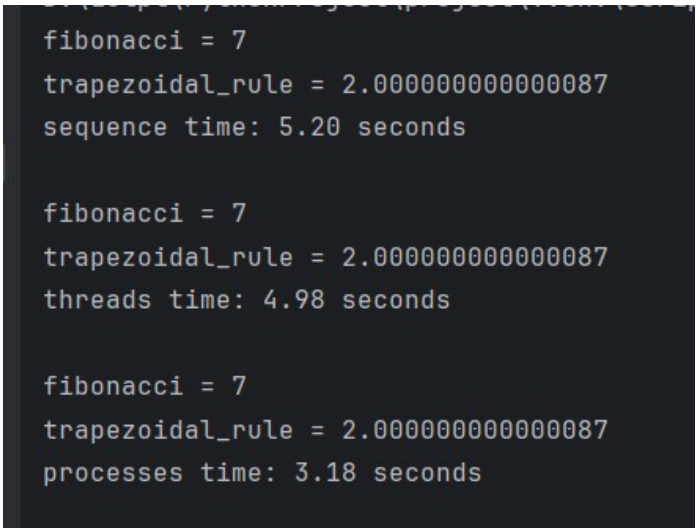


2) на потоках



3) через процессы





Последовательное выполнение занимает больше всего времени. Многопроцессорное выполнение является самым быстрым, что говорит о том, что использование нескольких процессов может значительно ускорить выполнение, особенно для задач, требующих значительных вычислительных ресурсов, таких как вычисление интегралов. Многопроцессорность позволяет обойти ограничения GIL, так как каждый процесс имеет свой собственный интерпретатор Python и память. Это объясняет, почему многопроцессорное выполнение оказалось быстрее. Если задачи требуют значительных вычислительных ресурсов и могут быть распараллелены, использование многопроцессорности будет более эффективным, но только для больших задач.