Задание 1. Моделирование параллельных платёжных операций

Данный код демонстрирует использование многопоточности для параллельной обработки финансовых транзакций, где создается пять потоков, каждый из которых имитирует обработку платежа с случайной задержкой от 1 до 3 секунд для эмуляции реального времени выполнения операций через внешние API. Потоки запускаются одновременно и выполняются независимо, что значительно ускоряет общее время обработки по сравнению с последовательным выполнением, а основной поток ожидает завершения всех операций через метод join перед выводом финального сообщения о successful обработке всех платежей.

```
import threading
import time
import random

def process_payment(transaction_id):
    print(f"📜 Обработка транзакции {transaction_id} начата")
    time.sleep(random.uniform(1, 3))  # имитация задержки API
    print(f"✅ Транзакция {transaction_id} завершена")

threads = []
for i in range(5):
    t = threading.Thread(target=process_payment, args=(i,))
    t.start()
    threads.append(t)

for t in threads:
    t.join()

print("💰 Все платежи обработаны!")
```

```
📜 Обработка транзакции 0 начата
📜 Обработка транзакции 1 начата
📜 Обработка транзакции 2 начата
📜 Обработка транзакции 3 начата
📜 Обработка транзакции 4 начата
✅ Транзакция 0 завершена
✅ Транзакция 3 завершена
✅ Транзакция 2 завершена
✅ Транзакция 4 завершена
✅ Транзакция 1 завершена
💰 Все платежи обработаны!
```

Задание 2. Получение курсов валют с нескольких источников

Данный код использует ThreadPoolExecutor для параллельного получения данных о курсах криптовалют с различных API-эндпоинтов CoinDesk, создавая отдельный поток для каждого URL-адреса и одновременно отправляя до трех запросов. Каждая функция get_rate измеряет время выполнения запроса, обрабатывает возможные ошибки соединения и возвращает название валюты вместе с продолжительностью запроса или сообщением об ошибке, после чего результаты выводятся по мере завершения работы потоков, демонстрируя эффективность многопоточного подхода для ускорения сбора данных из внешних источников.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import requests
import time

urls = [
    "https://api.coindesk.com/v1/bpi/currentprice/BTC.json",
    "https://api.coindesk.com/v1/bpi/currentprice/ETH.json",
    "https://api.coindesk.com/v1/bpi/currentprice/LTC.json"
]

def get_rate(url):
    start = time.perf_counter()
    try:
        print(f"Attempting to fetch data from: {url}")
        response = requests.get(url)
        response.raise_for_status() # Raise an HTTPError for bad responses (4xx or 5xx)
        end = time.perf_counter()
        return url.split("/")[-1], end - start
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data from {url}: {e}")
        return url.split("/")[-1], None


with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(get_rate, url) for url in urls]
```

```
    for future in as_completed(futures):
        name, duration = future.result()
        if duration is not None:
             print(f"{name}: ответ за {duration:.2f} сек")
        else:
             print(f"{name}: Не удалось получить данные")
```

```
Attempting to fetch data from: https://api.coindesk.com/v1/bpi/currentprice/BTC.json
Attempting to fetch data from: https://api.coindesk.com/v1/bpi/currentprice/ETH.json
Attempting to fetch data from: https://api.coindesk.com/v1/bpi/currentprice/LTC.json
Error fetching data from https://api.coindesk.com/v1/bpi/currentprice/BTC.json: HTTPSConnectionPool(host='api.coindesk.com',
BTC.json: Не удалось получить данные
Error fetching data from https://api.coindesk.com/v1/bpi/currentprice/ETH.json: HTTPSConnectionPool(host='api.coindesk.com',
ETH.json: Не удалось получить данные
Error fetching data from https://api.coindesk.com/v1/bpi/currentprice/LTC.json: HTTPSConnectionPool(host='api.coindesk.com',
LTC.json: Не удалось получить данные
```

Задание 3. Мониторинг выполнения задач в реальном времени

Данный код моделирует параллельные запросы к различным платежным системам с использованием ThreadPoolExecutor, где каждая имитированная функция API-вызова имеет случайную задержку от 1 до 4 секунд для эмуляции реального времени обработки транзакций. После запуска всех четырех запросов одновременно программа выводит сообщения о начале каждого запроса и затем асинхронно получает результаты по мере их готовности, демонстрируя эффективность многопоточного подхода для одновременного взаимодействия с несколькими внешними сервисами и сокращения общего времени ожидания ответов от различных платежных провайдеров.

```
from concurrent.futures import ThreadPoolExecutor
import random
import time

def simulate_api_call(api_name):
    duration = random.uniform(1, 4)
    time.sleep(duration)
    return f"{api_name} ответил за {duration:.2f} сек"

apis = ["СБП", "Мир", "Visa", "MasterCard"]

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = {executor.submit(simulate_api_call, api): api for api in apis}
    for future in futures:
        api = futures[future]
        print(f"📤 Запущен запрос к {api}")

    for future in futures:
        print("🔔", future.result())
```

```
📤 Запущен запрос к СБП
📤 Запущен запрос к Мир
📤 Запущен запрос к Visa
📤 Запущен запрос к MasterCard
🔔 СБП ответил за 1.06 сек
🔔 Мир ответил за 2.63 сек
🔔 Visa ответил за 2.90 сек
🔔 MasterCard ответил за 1.66 сек
```

Задание 4. Многопоточная обработка заявок

Этот код имитирует асинхронную запись транзакций в лог-файл.

Функция `simulate_transactions` создает 20 случайных транзакций с разными типами, суммами и статусами. Каждая транзакция записывается в файл `transactions.log` с помощью `aiofiles` для асинхронной работы с файлами, добавляя небольшую задержку между операциями.

Код использует асинхронные возможности для эффективной записи данных без блокировки основного потока выполнения.

```
import threading
import queue
import time
import random

def worker(q):
    while True:
        try:
            tx = q.get(timeout=2)
        except queue.Empty:
            break
        print(f"{threading.current_thread().name} обрабатывает {tx}")
```

```
        time.sleep(random.uniform(1, 2))
        q.task_done()

q = queue.Queue()
for i in range(10):
    q.put(f"txn_{i}")

threads = [threading.Thread(target=worker, args=(q,)) for _ in range(3)]
for t in threads:
    t.start()
for t in threads:
    t.join()

print("✅ Все заявки обработаны.")
```

```
Thread-17 (worker) обрабатывает txn_0Thread-18 (worker) обрабатывает txn_1
Thread-19 (worker) обрабатывает txn_2

Thread-17 (worker) обрабатывает txn_3
Thread-18 (worker) обрабатывает txn_4
Thread-19 (worker) обрабатывает txn_5
Thread-17 (worker) обрабатывает txn_6
Thread-19 (worker) обрабатывает txn_7
Thread-18 (worker) обрабатывает txn_8
Thread-17 (worker) обрабатывает txn_9
✅ Все заявки обработаны.
```

Задание 5. Контроль доступа к счёту клиента

Данный код демонстрирует использование блокировок для обеспечения потокобезопасности при одновременном доступе к общему финансовому ресурсу, где три потока выполняют случайные операции пополнения и списания средств с общего баланса в диапазоне от -1000 до +1000 единиц. Применение конструкции with lock гарантирует атомарность каждой операции изменения баланса, предотвращая состояние гонки и обеспечивая корректность итогового результата после выполнения всех транзакций, что критически важно для финансовых систем с параллельным доступом к общим данным.

```
import threading
import random
import time

balance = 10000
lock = threading.Lock()

def update_balance(name):
    global balance
    for _ in range(5):
        with lock:
            delta = random.randint(-1000, 1000)
            balance += delta
            print(f"{name} изменил баланс на {delta:+}, итого: {balance}")
        time.sleep(random.uniform(0.5, 1))

threads = [threading.Thread(target=update_balance, args=(f"Поток-{i}",)) for i in range(3)]
for t in threads:
    t.start()
for t in threads:
    t.join()

print("💰 Итоговый баланс:", balance)
```

```
Поток-0 изменил баланс на -747, итого: 9253
Поток-1 изменил баланс на -384, итого: 8869
Поток-2 изменил баланс на -777, итого: 8092
Поток-1 изменил баланс на +358, итого: 8450
Поток-2 изменил баланс на +966, итого: 9416
Поток-0 изменил баланс на -1, итого: 9415
Поток-1 изменил баланс на +681, итого: 10096
Поток-2 изменил баланс на +49, итого: 10145
Поток-0 изменил баланс на +417, итого: 10562
Поток-1 изменил баланс на -576, итого: 9986
Поток-2 изменил баланс на -391, итого: 9595
Поток-0 изменил баланс на -859, итого: 8736
Поток-0 изменил баланс на +459, итого: 9195
Поток-1 изменил баланс на +104, итого: 9299
Поток-2 изменил баланс на +636, итого: 9935
💰 Итоговый баланс: 9935
```

Задание 6. Ограничение числа одновременных операций

Данный код демонстрирует использование семафора для ограничения одновременного доступа к ресурсу в многопоточной среде. Создается семафор с значением 2, что позволяет одновременно работать только двум потокам, в то время как остальные ожидают своей очереди. Шесть потоков-задач пытаются выполнить функцию обработки, но благодаря семафору одновременно работают не более двух задач, что обеспечивает контролируемый доступ к общему ресурсу и предотвращает перегрузку системы.

```
import threading
import time
import random

sem = threading.Semaphore(2)

def process(name):
    with sem:
        print(f"{name} начал обработку")
        time.sleep(random.uniform(1, 3))
        print(f"{name} завершил")

threads = [threading.Thread(target=process, args=(f"Задача-{i}",)) for i in range(6)]
for t in threads:
    t.start()
for t in threads:
    t.join()
```

```
Задача-0 начал обработку
Задача-1 начал обработку
Задача-1 завершил
Задача-2 начал обработку
Задача-0 завершил
Задача-3 начал обработку
Задача-3 завершил
Задача-4 начал обработку
Задача-2 завершил
Задача-5 начал обработку
Задача-5 завершил
Задача-4 завершил
```

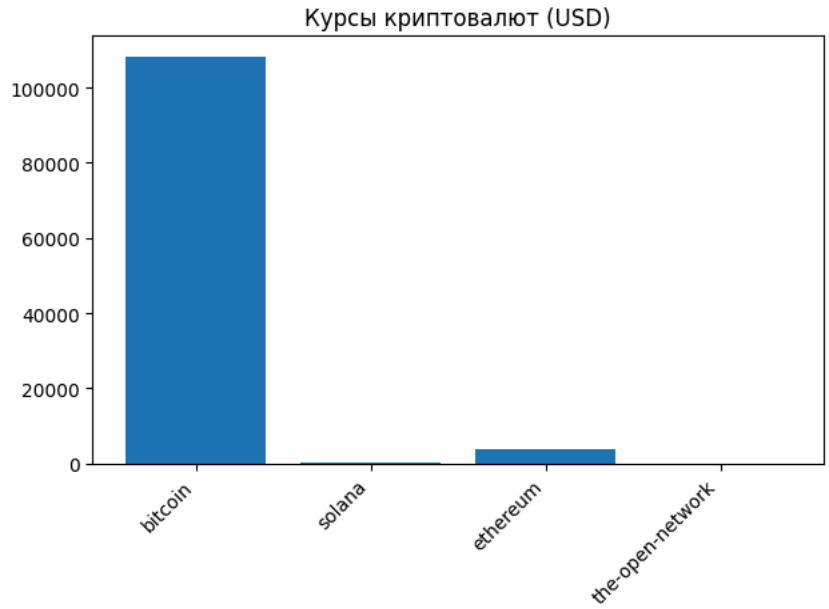
Задание 7. Многопоточный сбор курсов криптовалют с визуализацией

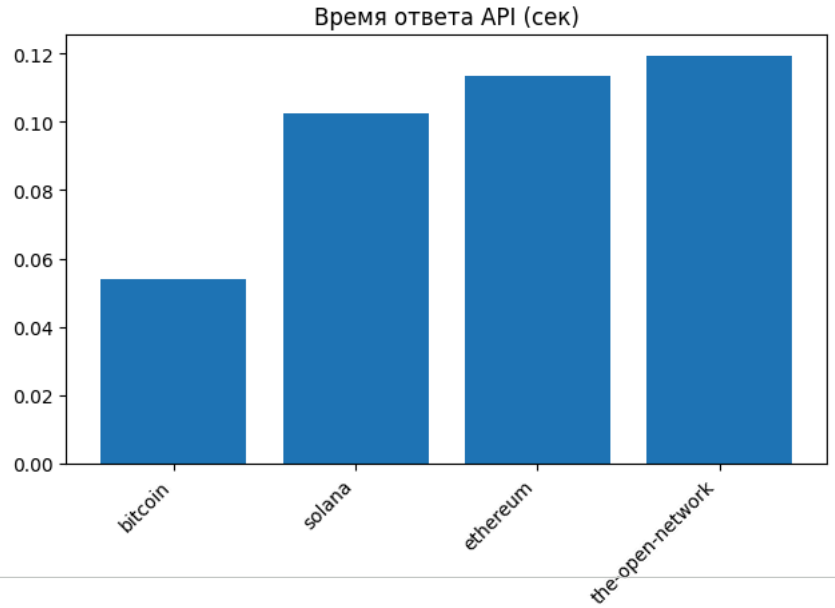
Данный код демонстрирует параллельное получение актуальных курсов криптовалют через CoinGecko API с использованием ThreadPoolExecutor для одновременного выполнения нескольких HTTP-запросов. Программа запрашивает цены в долларах США для четырех криптовалют (Bitcoin, Ethereum, Solana и Toncoin), измеряет время выполнения каждого запроса и фильтрует результаты с отсутствующими данными. Полученные данные сохраняются в DataFrame и визуализируются в виде двух столбчатых диаграмм: первая отображает текущие цены криптовалют, а вторая показывает время ответа API для каждого запроса, что позволяет оценить эффективность параллельного выполнения.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import requests
import pandas as pd
import matplotlib.pyplot as plt
import time

# Use the correct CoinGecko API id for Toncoin
coins = ["bitcoin", "ethereum", "solana", "the-open-network"]
API = "https://api.coingecko.com/api/v3/simple/price"

def get_price(coin):
    params = {"ids": coin, "vs_currencies": "usd"}
    start = time.perf_counter()
    r = requests.get(API, params=params)
    r.raise_for_status() # Raise an HTTPError for bad responses (4xx or 5xx)
    data = r.json()
    # Check if the coin data is in the response
    if coin in data and "usd" in data[coin]:
        price = data[coin]["usd"]
    else:
        # Handle cases where the coin data or USD price is missing
        print(f"Warning: Could not get USD price for {coin}")
        price = None # Or handle as an error, depending on requirements

    duration = time.perf_counter() - start
    return coin, price, duration

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = [executor.submit(get_price, c) for c in coins]
    # Filter out results where price was not obtained
```

```
        results = [f.result() for f in as_completed(futures)]
        results = [r for r in results if r[1] is not None]


    df = pd.DataFrame(results, columns=["Coin", "Price (USD)", "Time (s)"])
    print(df)

    plt.bar(df["Coin"], df["Price (USD)"])
    plt.title("Курсы криптовалют (USD)")
    plt.xticks(rotation=45, ha='right') # Rotate labels for better readability
    plt.tight_layout() # Adjust layout to prevent labels overlapping
    plt.show()

    plt.bar(df["Coin"], df["Time (s)"])
    plt.title("Время ответа API (сек)")
    plt.xticks(rotation=45, ha='right') # Rotate labels for better readability
    plt.tight_layout() # Adjust layout to prevent labels overlapping
    plt.show()
```

```
                Coin  Price (USD)  Time (s)
0            bitcoin    108311.00  0.054077
1             solana       185.36  0.102476
2           ethereum      3855.63  0.113506
3  the-open-network         2.15  0.119495
```





Задание 9. Расчёт комиссий в отдельных процессах

Данный код демонстрирует использование многопроцессорности для параллельного расчета комиссий по различным суммам платежей. Создается пул процессов по количеству доступных ядер процессора, что позволяет одновременно выполнять вычисления для разных сумм, имитируя задержку обработки в полсекунды для каждой операции. Функция расчета комиссии применяет случайную процентную ставку от 1% до 3% к каждой сумме, и результаты выводятся в виде округленных значений, показывая эффективность распараллеливания вычислительных задач для ускорения массовых финансовых расчетов.

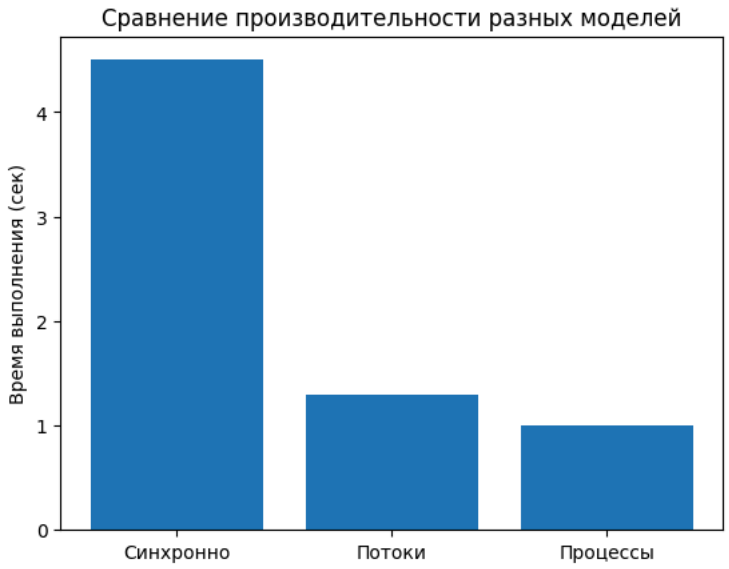
```
from multiprocessing import Pool, cpu_count
import random
import time

def calc_fee(amount):
    time.sleep(0.5)
    return amount * random.uniform(0.01, 0.03)

amounts = [random.randint(10_000, 100_000) for _ in range(10)]

with Pool(processes=cpu_count()) as pool:
    results = pool.map(calc_fee, amounts)

print("💸 Комиссии:", [round(r, 2) for r in results])
```

```
💸 Комиссии: [745.61, 471.78, 2421.33, 615.63, 1441.27, 2799.58, 265.95, 1204.09, 874.08, 1737.72]
```

Финальная визуализация: сравнение времени

Данный код создает столбчатую диаграмму для наглядного сравнения времени выполнения операций в трех различных моделях программирования: синхронной, многопоточной и многопроцессорной. На графике демонстрируется значительное преимущество параллельных подходов над последовательным выполнением, где многопроцессорная обработка показывает наилучший результат в 1 секунду по сравнению с 4.5 секундами у синхронного метода, визуализируя эффективность распараллеливания задач для оптимизации производительности.

```
import matplotlib.pyplot as plt

labels = ["Синхронно", "Потоки", "Процессы"]
times = [4.5, 1.3, 1.0]

plt.bar(labels, times)
plt.title("Сравнение производительности разных моделей")
plt.ylabel("Время выполнения (сек)")
plt.show()
```



Чтобы изменить содержимое ячейки, дважды нажмите на нее (или выберите "Ввод")

Чтобы изменить содержимое ячейки, дважды нажмите на нее (или выберите "Ввод")

```
Напишите программный код или сгенерируйте его с помощью искусственного интеллекта.
```