

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ  
ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»  
ИНСТИТУТ ЦИФРОВОГО РАЗВИТИЯ**

**Отчёт о лабораторной работе №12 по дисциплине основы программной  
инженерии**

Выполнила:

Нестеренко Тамара Антоновна,  
2 курс, группа ПИЖ-б-о-20-1,

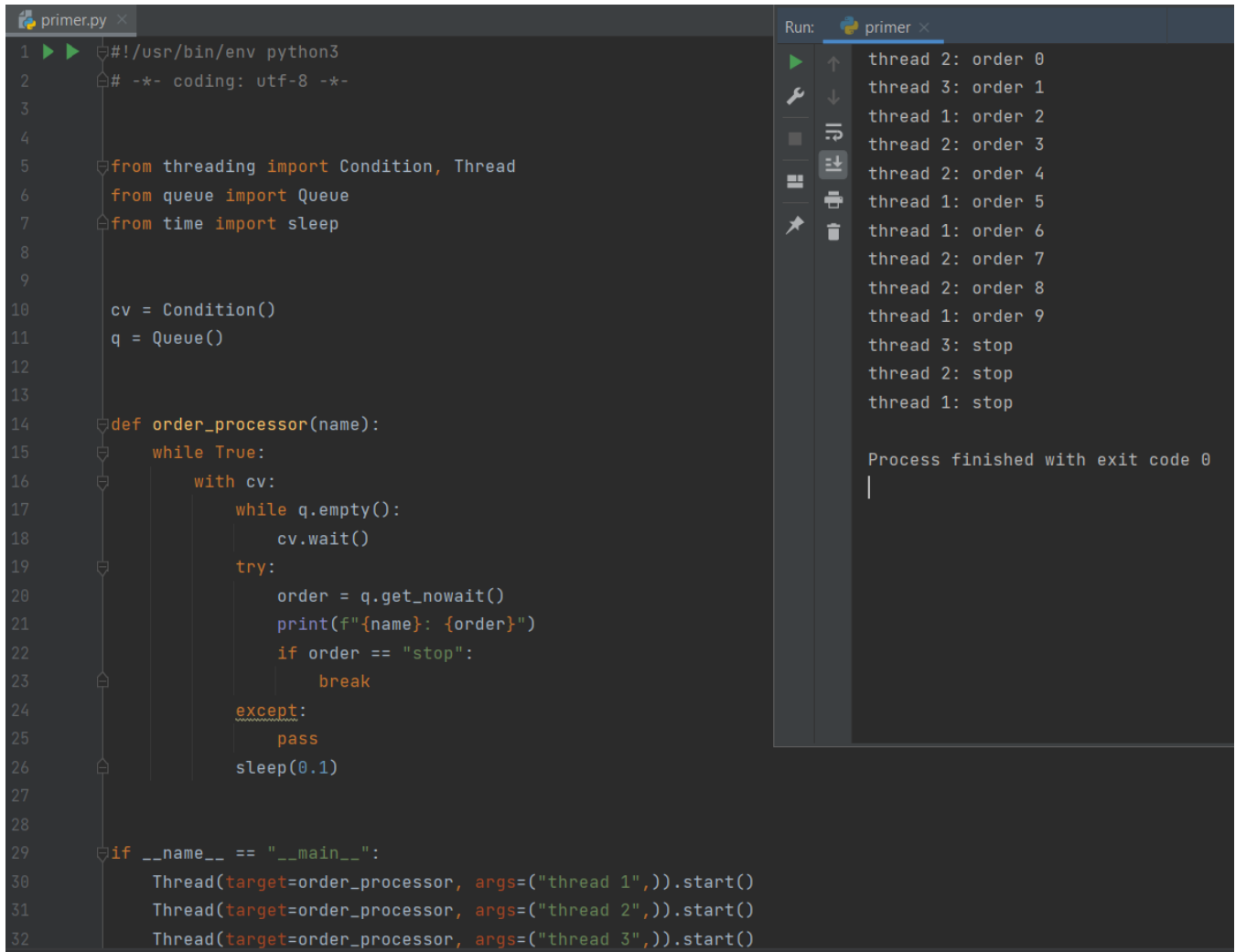
Проверил:

Доцент кафедры инфокоммуникаций,  
Воронкин Р.А.

Ставрополь, 2022 г.

# ВЫПОЛНЕНИЕ

## 1. Практическая часть



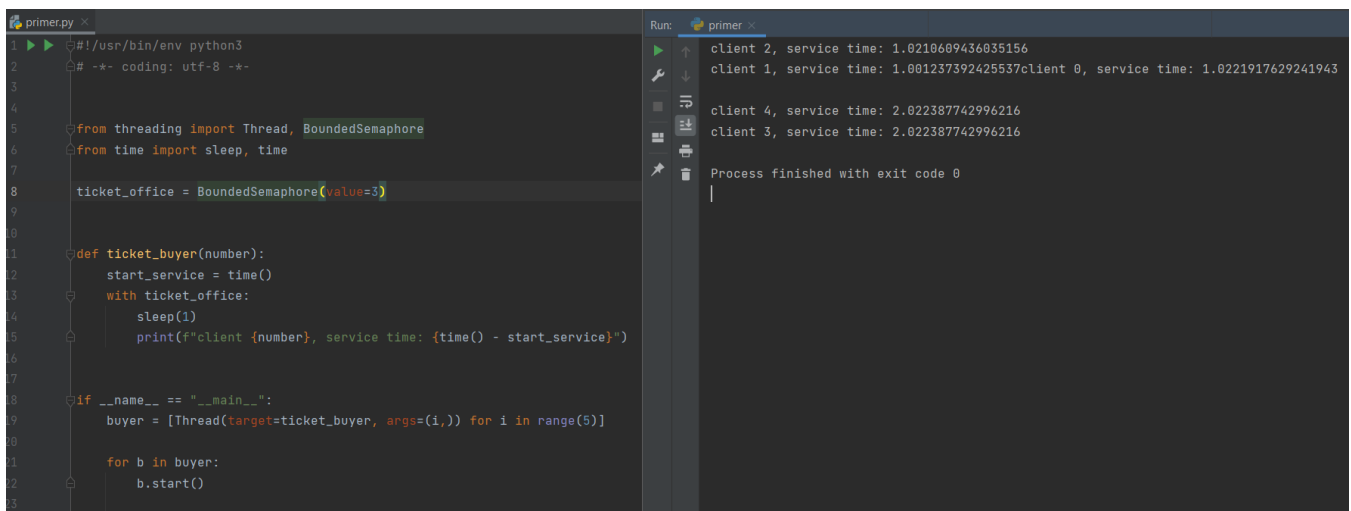
```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4
5 from threading import Condition, Thread
6 from queue import Queue
7 from time import sleep
8
9
10 cv = Condition()
11 q = Queue()
12
13
14 def order_processor(name):
15     while True:
16         with cv:
17             while q.empty():
18                 cv.wait()
19             try:
20                 order = q.get_nowait()
21                 print(f"{name}: {order}")
22                 if order == "stop":
23                     break
24             except:
25                 pass
26             sleep(0.1)
27
28
29 if __name__ == "__main__":
30     Thread(target=order_processor, args=("thread 1",)).start()
31     Thread(target=order_processor, args=("thread 2",)).start()
32     Thread(target=order_processor, args=("thread 3",)).start()
```

Run: primer x

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

Process finished with exit code 0
```

Рисунок 1 – Пример создания и ожидания завершения работы потоков. Класс Thread



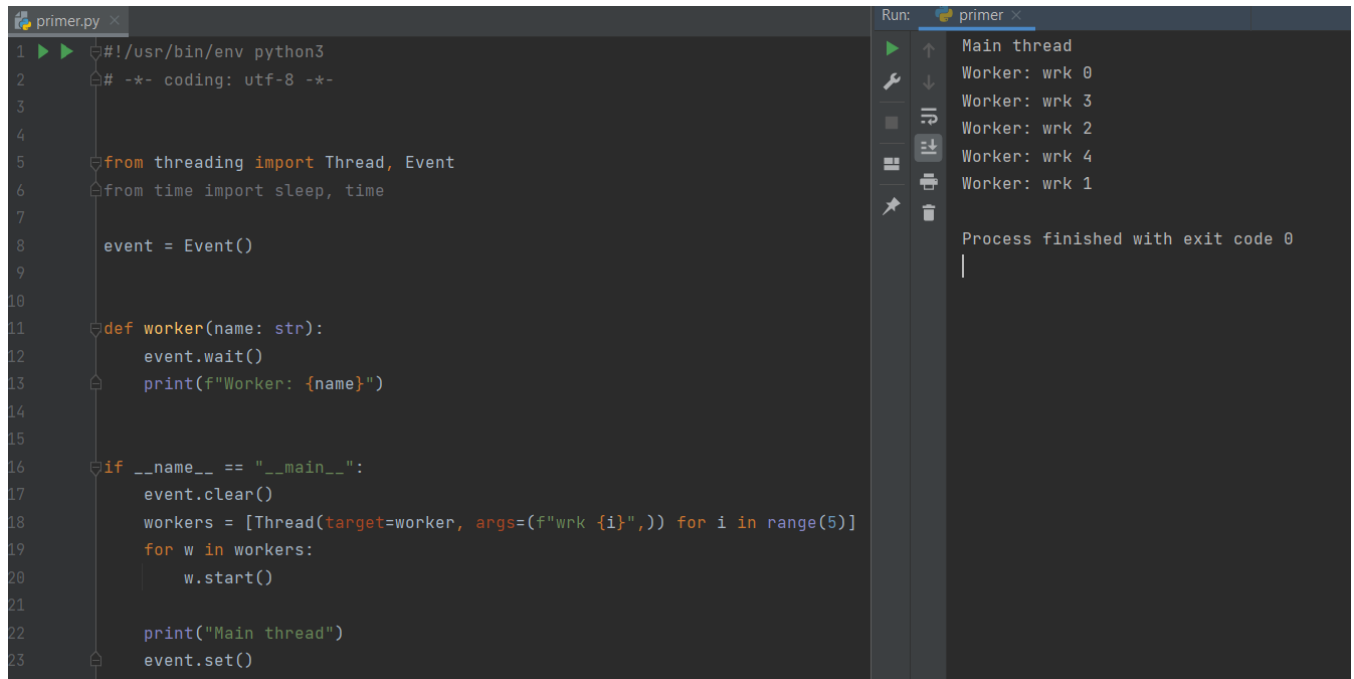
```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4
5 from threading import Thread, BoundedSemaphore
6 from time import sleep, time
7
8 ticket_office = BoundedSemaphore(value=3)
9
10
11 def ticket_buyer(number):
12     start_service = time()
13     with ticket_office:
14         sleep(1)
15     print(f"client {number}, service time: {time() - start_service}")
16
17
18 if __name__ == "__main__":
19     buyer = [Thread(target=ticket_buyer, args=(i,)) for i in range(5)]
20
21     for b in buyer:
22         b.start()
```

Run: primer x

```
client 2, service time: 1.0210609436035156
client 1, service time: 1.001237392425537client 0, service time: 1.0221917629241943
client 4, service time: 2.022387742996216
client 3, service time: 2.022387742996216

Process finished with exit code 0
```

Рисунок 2 – Пример использования метода join()



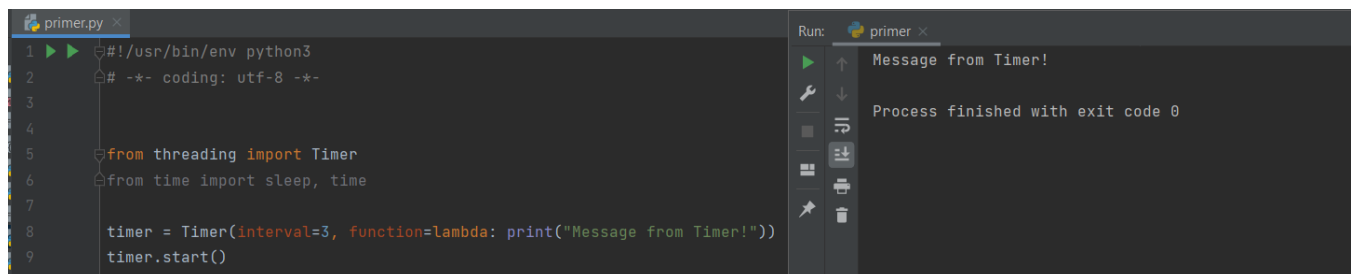
```
1  >>>#!/usr/bin/env python3
2  >>># -*- coding: utf-8 -*-
3
4
5  >>>from threading import Thread, Event
6  >>>from time import sleep, time
7
8  >>>event = Event()
9
10
11 >>>def worker(name: str):
12 >>>    event.wait()
13 >>>    print(f"Worker: {name}")
14
15
16 >>>if __name__ == "__main__":
17 >>>    event.clear()
18 >>>    workers = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(5)]
19 >>>    for w in workers:
20 >>>        w.start()
21
22 >>>    print("Main thread")
23 >>>    event.set()
```

Run: primer x

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

Process finished with exit code 0

Рисунок 3 – Пример использования метода is\_alive()



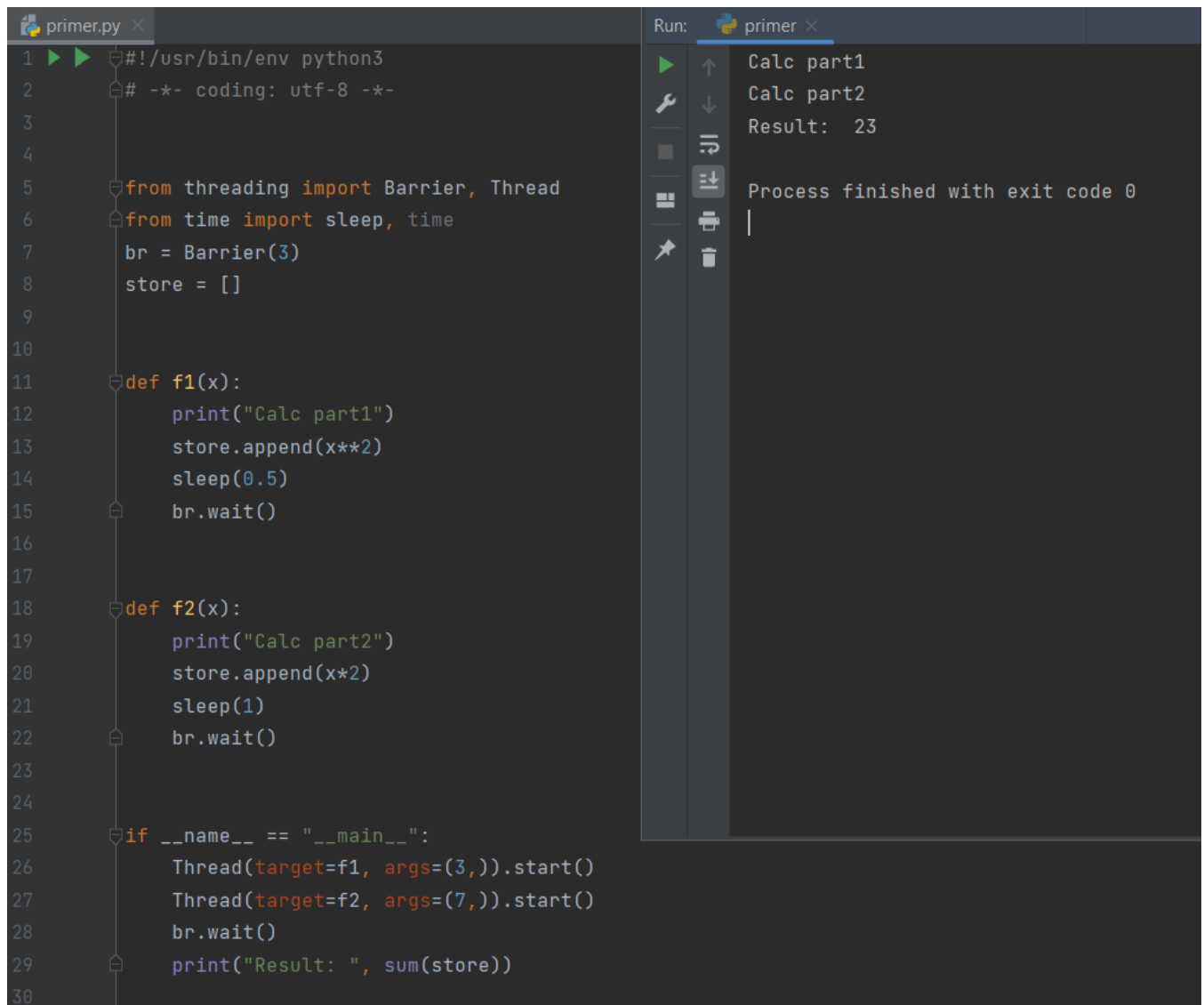
```
1  >>>#!/usr/bin/env python3
2  >>># -*- coding: utf-8 -*-
3
4
5  >>>from threading import Timer
6  >>>from time import sleep, time
7
8  >>>timer = Timer(interval=3, function=lambda: print("Message from Timer!"))
9  >>>timer.start()
```

Run: primer x

- Message from Timer!

Process finished with exit code 0

Рисунок 4 – Пример создания классов наследников от Thread



The image shows a code editor with a file named `primer.py` and a terminal window titled `Run: primer`. The code defines two functions, `f1` and `f2`, which use a `Barrier` to synchronize their execution. `f1` prints "Calc part1", appends  $x^2$  to a list, sleeps for 0.5 seconds, and then waits at the barrier. `f2` prints "Calc part2", appends  $x^2$  to the list, sleeps for 1 second, and then waits at the barrier. The main block starts two threads, `f1` with `args=(3,)` and `f2` with `args=(7,)`, waits for both to finish, and then prints the sum of the list.

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4
5  from threading import Barrier, Thread
6  from time import sleep, time
7  br = Barrier(3)
8  store = []
9
10
11 def f1(x):
12     print("Calc part1")
13     store.append(x**2)
14     sleep(0.5)
15     br.wait()
16
17
18 def f2(x):
19     print("Calc part2")
20     store.append(x*2)
21     sleep(1)
22     br.wait()
23
24
25 if __name__ == "__main__":
26     Thread(target=f1, args=(3,)).start()
27     Thread(target=f2, args=(7,)).start()
28     br.wait()
29     print("Result: ", sum(store))
30
```

The terminal output shows the execution of the script:

```
Calc part1
Calc part2
Result:  23
Process finished with exit code 0
```

Рисунок 5 – Пример принудительного завершения работы потока

```
1.py x
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4
5  from threading import Thread, Lock
6  import math
7  from queue import Queue
8
9  eps = .0000001
10 q = Queue()
11 lock = Lock()
12
13
14 def inf_sum(x):
15     lock.acquire()
16     a = 1
17     summa = math.cos(x)
18     i = 1
19     prev = 0
20     while abs(summa + prev) < eps:
21         a = a * (math.cos(2*x)) / 2
22         prev = summa

```

```
1 x
Enter the number to calculate: 1
The sum of an infinite series is: 0.5403023058681398
The calculated answer is: 0.5403023058681398
Process finished with exit code 0

```

Рисунок 6 – Пример решения индивидуального задания

## 2. Вопросы для защиты

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

Lock-объект может находиться в двух состояниях: захваченное (заблокированное) и не захваченное (не заблокированное, свободное). После создания он находится в свободном состоянии. Для работы с Lock-объектом используются методы `acquire()` и `release()`. Если Lock свободен, то вызов метода `acquire()` переводит его в заблокированное состояние. Повторный вызов `acquire()` приведет к блокировке инициировавшего это действие потока до тех пор, пока Lock не будет разблокирован каким-то другим потоком с помощью метода `release()`. Вывоз метода `release()` на свободном Lock-объекте приведет к выбросу исключения `RuntimeError`.

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

В отличие от рассмотренного выше Lock-объекта RLock может освободить только тот поток, который его захватил. Повторный захват потоком уже захваченного RLock-объекта не блокирует его. RLock-объекты поддерживают возможность вложенного захвата, при этом освобождение происходит только после того, как был выполнен `release()` для внешнего `acquire()`. Сигнатуры и назначение методов `release()` и `acquire()` RLock-объектов совпадают с приведенными для Lock, но в отличие от него у RLock нет метода `locked()`. RLock-объекты поддерживают протокол менеджера контекста.

### 3. Как выглядит порядок работы с условными переменными?

Порядок работы с условными переменными выглядит так:

1. На стороне Consumer'a: проверить доступен ли ресурс, если нет, то перейти в режим ожидания с помощью метода `wait()`, и ожидать оповещение от Producer'a о том, что ресурс готов и с ним можно работать. Метод `wait()` может быть вызван с таймаутом, по истечении которого поток выйдет из состояния блокировки и продолжит работу.

2. На стороне Producer'a: произвести работы по подготовке ресурса, после того, как ресурс готов оповестить об этом ожидающие потоки с помощью методов `notify()` или `notify_all()`. Разница между ними в том, что `notify()` разблокирует только один поток (если он вызван без параметров), а `notify_all()` все потоки, которые находятся в режиме ожидания.

### 4. Какие методы доступны у объектов условных переменных?

При создании объекта *Condition* вы можете передать в конструктор объект *Lock* или *RLock*, с которым хотите работать. Перечислим методы объекта *Condition* с кратким описанием:

- `acquire(*args)` – захват объекта-блокировки.
- `release()` – освобождение объекта-блокировки.
- `wait(timeout=None)` – блокировка выполнения потока до оповещения о снятии блокировки. Через параметр *timeout* можно задать время ожидания оповещения о снятии блокировки. Если вызвать `wait()` на Условной переменной, у которой предварительно не был вызван `acquire()`, то будет выброшено исключение *RuntimeError*.

- `wait_for(predicate, timeout=None)` – метод позволяет сократить количество кода, которое нужно написать для контроля готовности ресурса и ожидания оповещения. Он заменяет собой следующую конструкцию:

```
while not predicate():  
    cv.wait()
```

- `notify(n=1)` – снимает блокировку с остановленного методом `wait()` потока. Если необходимо разблокировать несколько потоков, то для этого следует передать их количество через аргумент `n`.
- `notify_all()` – снимает блокировку со всех остановленных методом `wait()` потоков.

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

С помощью семафоров удобно управлять доступом к ресурсу, который имеет ограничение на количество одновременных обращений к нему (например, количество подключений к базе данных и т.п.)

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

События по своему назначению и алгоритму работы похожи на рассмотренные ранее условные переменные. Основная задача, которую они решают – это взаимодействие между потоками через механизм оповещения. Объект класса `Event` управляет внутренним флагом, который сбрасывается с помощью метода `clear()` и устанавливается методом `set()`. Потоки, которые используют объект `Event` для синхронизации блокируются при вызове метода `wait()`, если флаг сброшен.

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

Модуль `threading` предоставляет удобный инструмент для запуска задач по таймеру – класс `Timer`. При создании таймера указывается функция, которая будет выполнена, когда он сработает. `Timer` реализован как поток, является наследником от `Thread`, поэтому для его запуска необходимо вызвать `start()`, если необходимо остановить работу таймера, то вызовите `cancel()`.

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

Последний инструмент для синхронизации работы потоков, который мы рассмотрим является Barrier. Он позволяет реализовать алгоритм, когда необходимо дождаться завершения работы группы потоков, прежде чем продолжить выполнение задачи.

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

Для решения определённого вида задач удобным будет каждый из способов, в зависимости от условий задачи, универсального способа нет.

Ссылка на репозиторий: [https://github.com/tamaranesterenko/Python\\_LR\\_12-2](https://github.com/tamaranesterenko/Python_LR_12-2)