

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

**Отчет по лабораторной работе № 2.24
«Синхронизация потоков в языке
программирования Python»**

по дисциплине «Основы программной инженерии»


Выполнила:
Образцова Мария Дмитриевна,
2 курс, группа ПИЖ-б-о-21-1,
Проверил:
Доцент кафедры инфокоммуникаций,
Воронкин Р.А.

Ставрополь, 2023 г.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

Owner *

 obrMaria ▾

/


Repository name *

OPI_2.24


✔ OPI_2.24 is available.

Great repository names are short and memorable. Need inspiration? [Here are some ideas.](#)

Description (optional)

☒  Public

Anyone on the internet can see this repository. You choose who can commit.

☐  Private

You choose who can see and commit to this repository.

Initialize this repository with:

☒ Add a README file

This is where you can write a long description for your project. [Learn more about README files.](#)

Add .gitignore


.gitignore template: Python ▾


Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: MIT License ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set  main as the default branch. Change the default name in [your settings.](#)

 You are creating a public repository in your personal account.

3. Выполните клонирование созданного репозитория. Организуйте свой репозиторий в соответствии с моделью ветвления git-flow.

```
M@DESKTOP-UVM9NOL MINGW64 ~/Desktop (master)
$ git clone https://github.com/obrMaria/OPI_2.24.git
Cloning into 'OPI_2.24'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (5/5), done.

M@DESKTOP-UVM9NOL MINGW64 ~/Desktop (master)
$ cd OPI_2.24

M@DESKTOP-UVM9NOL MINGW64 ~/Desktop/OPI_2.24 (main)
$ git flow init

Which branch should be used for bringing forth production releases?
- main
Branch name for production releases: [main]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [C:/Users/M/Desktop/OPI_2.24/.git/hooks]

M@DESKTOP-UVM9NOL MINGW64 ~/Desktop/OPI_2.24 (develop)
$ |
```

Рисунок – клонирование созданного репозитория

```
1.py x
1 from threading import Condition, Thread
2 from queue import Queue
3 from time import sleep
4
5 cv = Condition()
6 q = Queue()
7
8
9 # Consumer function for order processing
3 usages
10 def order_processor(name):
11     while True:
12         with cv:
13             # Wait while queue is empty
14             while q.empty():
15                 cv.wait()
16             try:
17                 # Get data (order) from queue
18                 order = q.get()
19                 print(f'{name}: {order}')
20                 q.task_done()
21
22 if __name__ == "__main__":
23     for _ in range(3):
24         thread = Thread(target=order_processor, args=('thread %d' % _,))
25         thread.start()
26
27 thread 1: order 0
28 thread 2: order 1
29 thread 3: order 2
30 thread 1: order 3
31 thread 2: order 4
32 thread 3: order 5
33 thread 1: order 6
34 thread 2: order 7
35 thread 3: order 8
36 thread 1: order 9
37 thread 2: stop
38 thread 3: stop
39 thread 1: stop
40
41 Process finished with exit code 0
```

Рисунок – Условные переменные

```

2.py x
1 from threading import Thread, BoundedSemaphore
2 from time import sleep, time
3
4 ticket_office = BoundedSemaphore(value=3)
5
6
7 1 usage
8 def ticket_buyer(number):
9     start_service = time()
10     with ticket_office:
11         sleep(1)
12         print(f"client {number}, service time: {time() - start_service}")
13
14 if __name__ == "__main__":
15     buyer = [Thread(target=ticket_buyer, args=(i,)) for i in range(5)]
16     for b in buyer:
17         b.start()
18

```

ticket_buyer()

```

2 x
:
:
:

```

```

C:\Users\M\Desktop\OPI_2.24\PY\primer\venv\Scripts\python.exe
client 0, service time: 1.0010004043579102
client 2, service time: 1.0010030269622803client 1, service
client 3, service time: 2.0009965896606445
client 4, service time: 2.0019969940185547

```

Process finished with exit code 0

Рисунок – Семафоры

The image shows a Python IDE with a file named `3.py` open. The code defines a thread pool using `threading.Event` and `Thread`. The `worker` function prints its name when called. The `__main__` block clears the event, creates and starts four workers, prints the main thread, and sets the event. The output window shows the execution results, including the main thread and four workers.

```
2 from threading import Event
3
4 event = Event()
5
6
7 def worker(name: str):
8     event.wait()
9     print(f"Worker: {name}")
10
11
12 if __name__ == "__main__":
13     # Clear event
14     event.clear()
15     # Create and start workers
16     workers = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(4)]
17     for w in workers:
18         w.start()
19     print("Main thread")
20     event.set()
21
```

Output:

```
C:\Users\M\Desktop\OPI_2.24\PY\primer\venv\Scripts\python.exe
Main thread
Worker: wrk 0
Worker: wrk 1
Worker: wrk 3
Worker: wrk 4Worker: wrk 2

Process finished with exit code 0
```

Рисунок – События

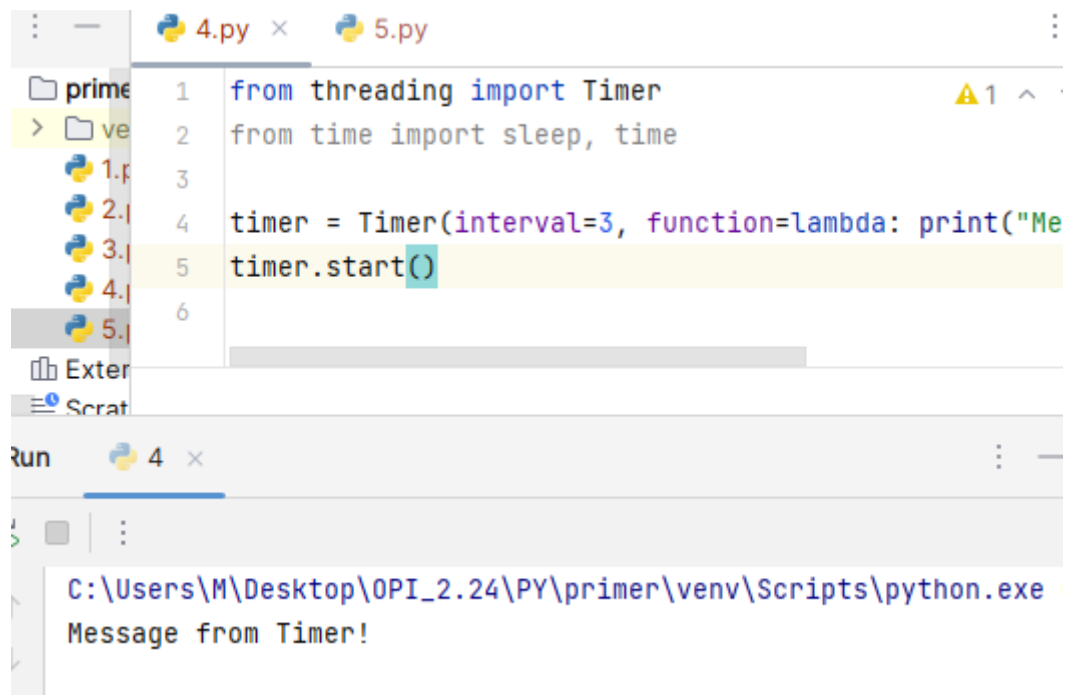
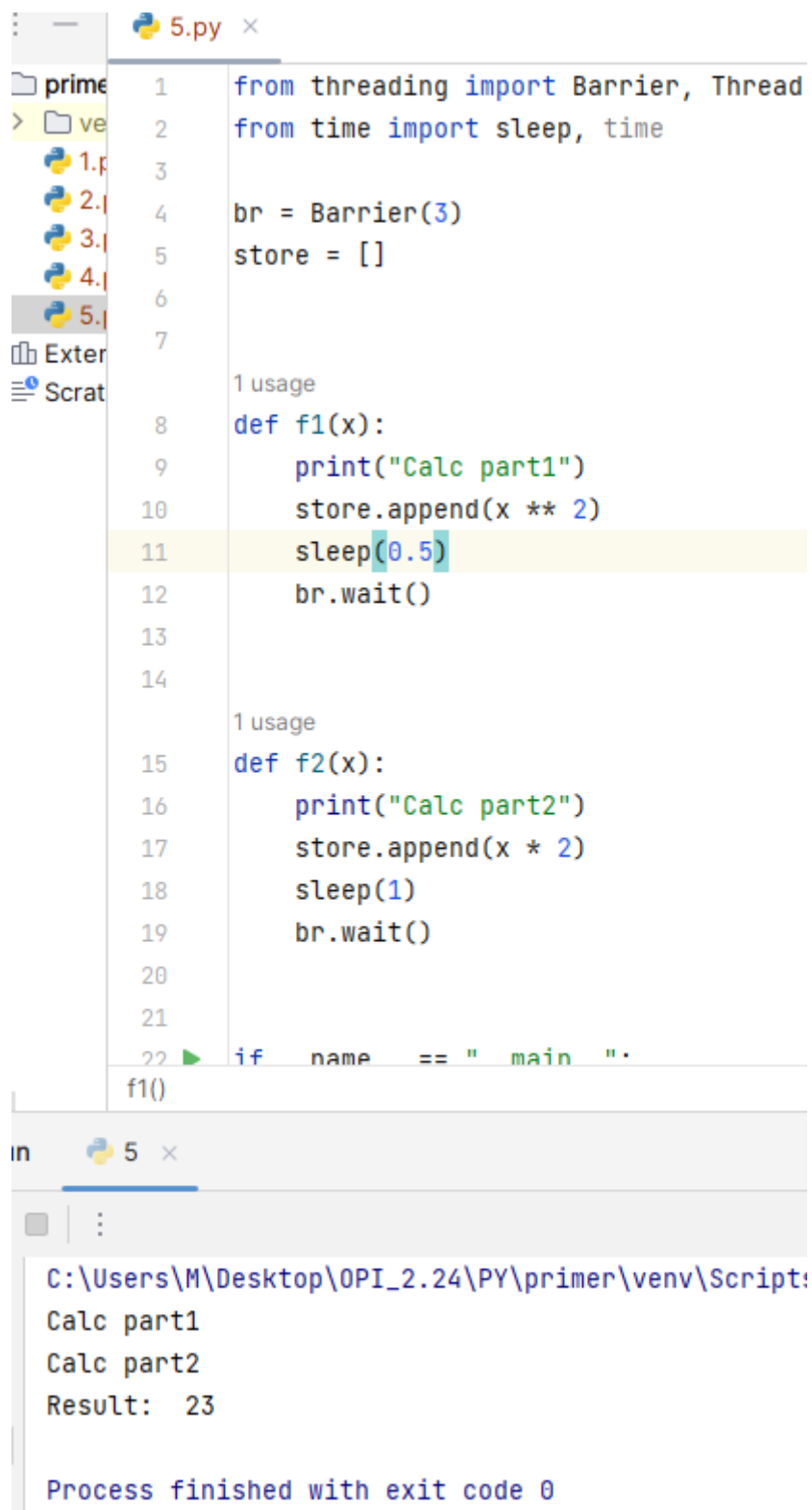


Рисунок – Таймеры



```
5.py x
1 from threading import Barrier, Thread
2 from time import sleep, time
3
4 br = Barrier(3)
5 store = []
6
7
8 1 usage
9 def f1(x):
10     print("Calc part1")
11     store.append(x ** 2)
12     sleep(0.5)
13     br.wait()
14
15 1 usage
16 def f2(x):
17     print("Calc part2")
18     store.append(x * 2)
19     sleep(1)
20     br.wait()
21
22 if name == " main ":
    f1()
```

```
5 x
C:\Users\M\Desktop\OPI_2.24\PY\primer\venv\Script:
Calc part1
Calc part2
Result: 23

Process finished with exit code 0
```

Рисунок – Барьеры

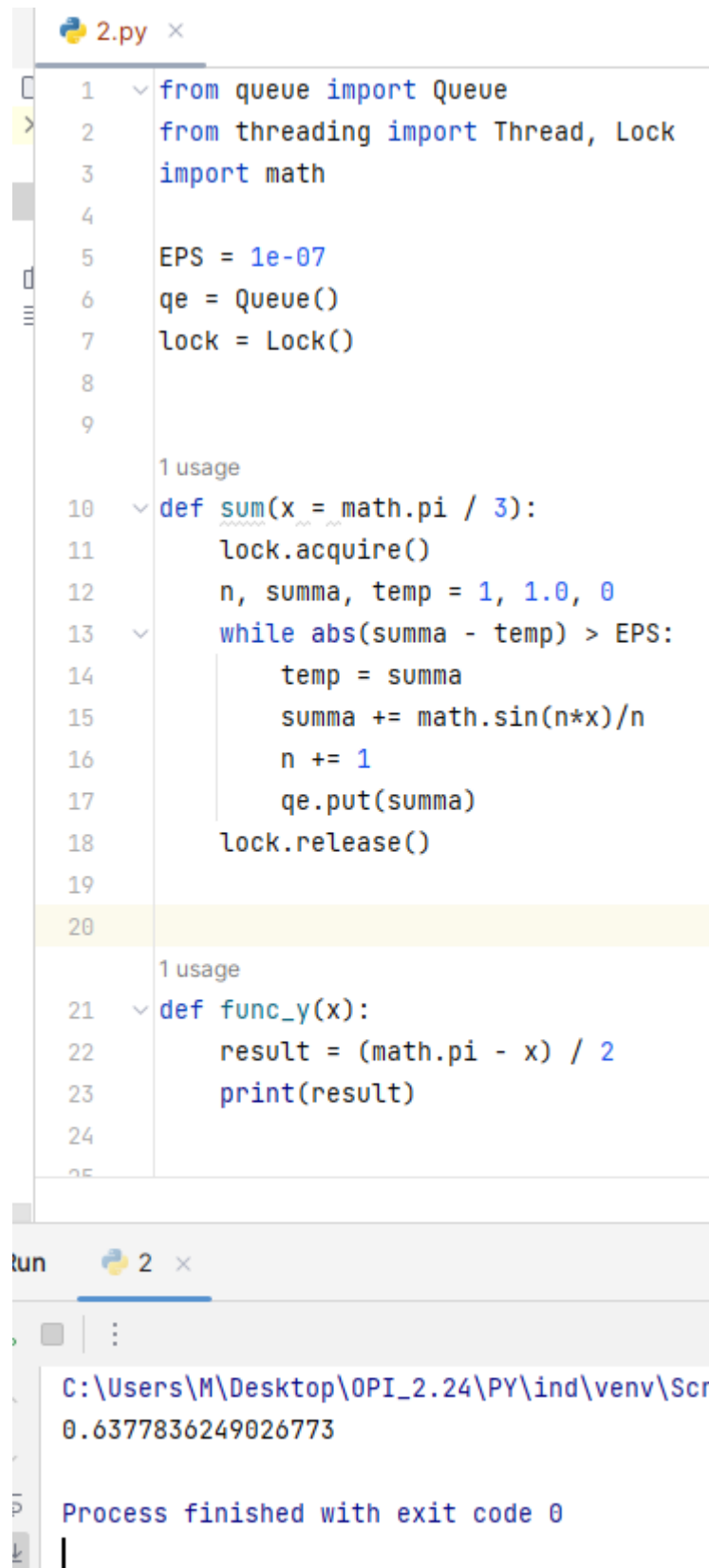
Разработать приложение, в котором выполнить решение вычислительной задачи (например, задачи из области физики, экономики, математики, статистики и т. д.) с помощью паттерна “Производитель-

Потребитель”, условие которой предварительно необходимо согласовать с преподавателем.

```
15 def consumer():
16     lock.acquire()
17     ls = []
18     while not q.empty():
19         s = q.get()
20         r = random.choice(["вопрос остался открыт", "вопрос решен", "гудок идет"])
21         print(f"звонок №: {s[1]} столкнулся с проблемой: {s[0]}, Результат: {r}")
22         ls.append(
23             {
24                 "№": s[1],
25                 "Проблема": s[0],
26                 "Результат": r
27             }
28         )
29     for i in ls:
consumer() > while not q.empty()

in 1 x 123 x
C:\Users\M\Desktop\OPI_2.24\PY\ind\venv\Scripts\python.exe C:\Users\M\Desktop\OPI_2.24\PY\ind\1.p
звонок №: 369 столкнулся с проблемой: сломался счетчик, Результат: вопрос остался открыт
звонок №: 645 столкнулся с проблемой: не проходит оплата, Результат: гудок идет
звонок №: 557 столкнулся с проблемой: нет света/газа/воды, Результат: гудок идет
звонок №: 43 столкнулся с проблемой: сломался счетчик, Результат: вопрос решен
звонок №: 586 столкнулся с проблемой: нет света/газа/воды, Результат: вопрос решен
звонок №: 646 столкнулся с проблемой: нет света/газа/воды, Результат: вопрос остался открыт
звонок № 369 ожидает оператора
звонок № 646 ожидает оператора
Done
```

Для своего индивидуального задания лабораторной работы 2.23 необходимо организовать конвейер, в котором сначала в отдельном потоке вычисляется значение первой функции, после чего результаты вычисления должны передаваться второй функции, вычисляемой в отдельном потоке. Потоки для вычисления значений двух функций должны запускаться одновременно.



The image shows a Python IDE with a file named `2.py` open. The code defines a `sum` function that uses a queue and a lock to calculate a sum, and a `func_y` function that prints a result. Below the code editor, a terminal window shows the execution path and the output of the program.

```
1  from queue import Queue
2  from threading import Thread, Lock
3  import math
4
5  EPS = 1e-07
6  qe = Queue()
7  lock = Lock()
8
9
10 1 usage
11  def sum(x = math.pi / 3):
12      lock.acquire()
13      n, summa, temp = 1, 1.0, 0
14      while abs(summa - temp) > EPS:
15          temp = summa
16          summa += math.sin(n*x)/n
17          n += 1
18          qe.put(summa)
19      lock.release()
20
21 1 usage
22  def func_y(x):
23      result = (math.pi - x) / 2
24      print(result)
25
```

Execution output:

```
C:\Users\M\Desktop\OPI_2.24\PY\ind\venv\Scr
0.6377836249026773

Process finished with exit code 0
```

ВОПРОСЫ

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()`. Вызов 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:

- На стороне Consumer'а: 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.

- 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.

`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”?

Реализация классического семафора, предложенного Дейкстрой. Суть его идеи заключается в том, при каждом вызове метода `acquire()` происходит уменьшение счетчика семафора на единицу, а при вызове `release()` – увеличение. Значение счетчика не может быть меньше нуля, если на момент вызова `acquire()` его значение равно нулю, то происходит блокировка потока до тех пор, пока не будет вызван `release()`.

Семафоры поддерживают протокол менеджера контекста.

Для работы с семафорами в Python есть класс `Semaphore`, при создании его объекта можно указать начальное значение счетчика через параметр `value`. `Semaphore` предоставляет два метода:

- `acquire(blocking=True, timeout=None)` – если значение внутреннего счетчика больше нуля, то счетчик уменьшается на единицу и метод возвращает `True`. Если значение счетчика равно нулю, то вызвавший данный метод поток блокируется, до тех пор, пока не будет кем-то вызван метод `release()`. Дополнительно при вызове метода можно указать параметры `blocking` и `timeout`, их назначение совпадает с `acquire()` для `Lock`.
- `release()` – увеличивает значение внутреннего счетчика на единицу.

Существует ещё один класс, реализующий алгоритм семафора `BoundedSemaphore`, в отличие от `Semaphore`, он проверяет, чтобы значение внутреннего счетчика было не больше того, что передано при создании объекта через аргумент `value`, если это происходит, то выбрасывается исключение `ValueError`.

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.

Методы класса `Event`:

- `is_set()` – возвращает `True` если флаг находится в взведенном состоянии.
- `set()` – переводит флаг в взведенное состояние.
- `clear()` – переводит флаг в сброшенное состояние.
- `wait(timeout=None)` – блокирует вызвавший данный метод поток если флаг соответствующего `Event`-объекта находится в сброшенном состоянии. Время нахождения в состоянии блокировки можно задать через параметр `timeout`.

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()`.

Конструктор класса Timer:

Timer(interval, function, args=None, kwargs=None)

Параметры:

- interval – количество секунд, по истечении которых будет вызвана функция function.
- function – функция, вызов которой нужно осуществить по таймеру.
- args, kwargs – аргументы функции function.

Методы класса Timer:

cancel() – останавливает выполнение таймера

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.

Конструктор класса:

Barrier(parties, action=None, timeout=None)

Параметры:

- parties – количество потоков, которые будут работать в рамках барьера.
- action – определяет функцию, которая будет вызвана, когда потоки будут освобождены (достигнут барьера).
- timeout – таймаут, который будет использовать как значение по умолчанию для методов wait().

Свойства и методы класса:

wait(timeout=None) – блокирует работу потока до тех пор, пока не будет получено уведомление либо не пройдет время указанное в timeout.

`reset()` – переводит `Barrier` в исходное (пустое) состояние. Потокам, ожидающим уведомления, будет передано исключение `BrokenBarrierError`.

`abort()` – останавливает работу барьера, переводит его в состояние “разрушен” (`broken`). Все текущие и последующие вызовы метода `wait()` будут завершены с ошибкой с выбросом исключения `BrokenBarrierError`.

`parties` – количество потоков, которое нужно для достижения барьера.

`n_waiting` – количество потоков, которое ожидает срабатывания барьера.

`broken` – значение флага равно `True` указывает на то, что барьер находится в “разрушенном” состоянии.

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

Блокировка используется для основной защиты совместно используемых ресурсов. Многократные потоки могут попытаться получить блокировку, но только один поток может фактически содержать ее в любой момент времени. В то время как тот поток содержит блокировку, другие потоки должны ожидать. Существует несколько различных типов блокировок, отличаясь в основном по тому, что потоки делают при ожидании для получения их.

Семафор во многом как блокировка, за исключением того, что конечное число потоков может содержать его одновременно. Семафоры могут думаться как являющийся во многом как груды маркеров. Многократные потоки могут взять эти маркеры, но, когда нет ни одного оставленного, поток должен ожидать, пока другой поток не возвращает тот.