МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное

учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

«Синхронизация потоков в языке программирования Python»

Отчет по лабораторной работе № 2.24

по дисциплине «Основы программной инженерии»

Выполнил студент группы ПИЖ-б-о-21-1

Гасанов Г. М. _____« » 2023г.

Подпись студента_____

Работа защищена « »_____20__г.

Проверил Воронкин Р.А. _____

(подпись)

Ставрополь 2023

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

Ход работы:

1. Изучить теоретический материал работы.
2. Проработайте примеры лабораторной работы.

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


from threading import Condition, Thread
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}")
with cv:
        cv.notify_all()
```

```
/Users/svetik/Desktop/OPI/OPI_LR_2.2
thread 1: order 0
thread 1: order 1
thread 1: order 2
thread 1: order 3
```

Рисунок 1 – Результат работы программы

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

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

ticket_office = BoundedSemaphore(value=3)
  def
ticket_buyer(number):
start_service = time()
with ticket_office:
        sleep(1)
        print(f"client {number}, service time: {time() - start_service}")
  if __name__ ==
"__main__":
    buyer = [Thread(target=ticket_buyer, args=(i,)) for i in range(5)]
     for b in
buyer:
        b.start()
```

```
/Users/svetik/Desktop/OPI/OPI_LR_2.24/PyCharm/
client 0, service time: 1.0050461292266846
client 1, service time: 1.00626802444458
client 2, service time: 1.0063509941101074
client 3, service time: 2.0113658905029297
client 4, service time: 2.0115408897399902

Process finished with exit code 0
```

Рисунок 2 – Результат работы программы

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

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

event = Event()
  def worker(name:
str):
    event.wait()
    print(f"Worker: {name}")
  if __name__ ==
"__main__":
    event.clear()
    workers = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(5)]
for w in workers:
        w.start()
```

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

```
/Users/svetik/Desktop/OPI/OPI_LR
Main thread
Worker: wrk 0Worker: wrk 1
Worker: wrk 4
Worker: wrk 3
Worker: wrk 2
```

Рисунок 3 – Результат работы программы

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

from threading import Timer from
time import sleep, time

timer = Timer(interval=3, function=lambda: print("Message from Timer!"))
timer.start()
```

```
/Users/svetik/Desktop/OPI/OPI_LR_2.2
Message from Timer!

Process finished with exit code 0
```

Рисунок 4 – Результат работы программы

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


from threading import Barrier, Thread from
time import sleep, time


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("Result: ", sum(store))
```

```
/Users/svetik/Deskto
Calc part1
Calc part2
Result:  23
```

Рисунок 5 – Результат работы программы

3. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

4. Выполните клонирование созданного репозитория.

5. Дополните файл .gitignore необходимыми правилами для работы с IDE PyCharm.

6. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.

7. Создайте проект PyCharm в папке репозитория.

8. Разработать приложение, в котором выполнить решение вычислительной задачи (например, задачи из области физики, экономики, математики, статистики и т. д.) с помощью паттерна “ПроизводительПотребитель”, условие которой предварительно необходимо согласовать с преподавателем.

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


import random from
queue import Queue
from threading import Lock, Thread
  def
manager():
    lock.acquire()  # Захватываем блокировку
    tasks = []
    while not q.empty():  # Пока очередь не пуста
task = q.get()  # Извлекаем задачу из очереди
        worker = random.choice(workers)  # Случайным образом выбираем работника
        print(f"Менеджер передал задачу '{task}' работнику {worker}")
tasks.append({              "Задача": task,
            "Работник": worker
        })


    lock.release()  # Освобождаем блокировку


    # Выводим информацию о нераспределенных задачах
for task in tasks:          if task["Работник"] is None:
            print(f"Задача '{task['Задача']}' ожидает выполнения")
  def worker():
    lock.acquire()      while
not q.empty():          task
= q.get()
        print(f"Работник {worker_id} выполняет задачу: '{task}'")
lock.release()          lock.acquire()      lock.release()
  if __name__ == "__main__":
    tasks = ["Задача 1", "Задача 2", "Задача 3", "Задача 4", "Задача 5"]
workers = ["Работник A", "Работник B", "Работник C"]      lock = Lock()      q
= Queue()


    # Заполняем очередь задачами      for
task in tasks:
        q.put(task)


    # Запускаем потоки работников      for
worker_id in workers:
        Thread(target=worker).start()


    # Запускаем поток менеджера
    Thread(target=manager).start()
```

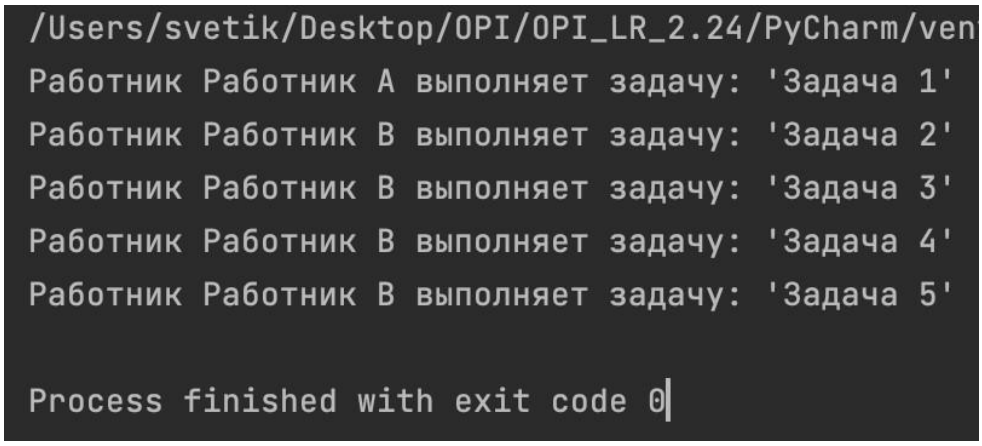

Рисунок 6 – Результат работы программы

9. 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.

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


from threading import Lock, Thread
from queue import Queue import
math


EPS = .0000001 q
= Queue() lock =
Lock()
  def sum_func(x,
q):
    summa = 1.0      temp = 0      n
= 1      while abs(summa - temp) >
EPS:          temp = summa
        summa += math.sin(n*x) / n
n += 1
    q.put(summa)
  def check_func(x,
q):      summa =
q.get()
    res = - math.log(2 * math.sin(0.5 * x))


    print(f"Sum is {summa}")
print(f"Check: {res}")
  if __name__ ==
'__main__':
    x = math.pi
    th1 = Thread(target=sum_func, args=(x, q)).start()
th2 = Thread(target=check_func, args=(x, q)).start()
```

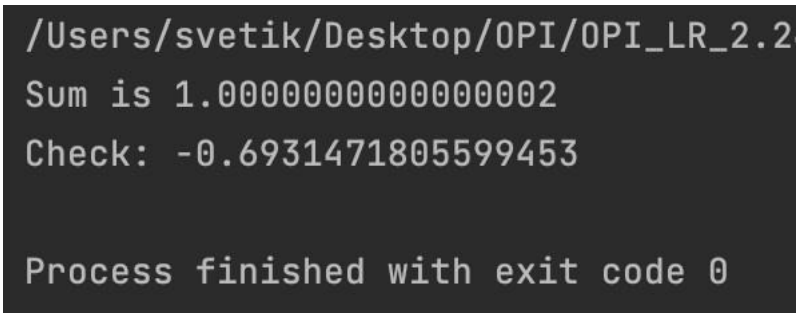

Рисунок 7 – Результат работы программы

10. Зафиксируйте сделанные изменения в репозитории.

11. Выполните слияние ветки для разработки с веткой main (master).

12. Отправьте сделанные изменения на сервер GitHub.

Контрольные вопросы:

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

Lock-объект может находится 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 отличии 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 отличии от него у *RLock* нет метода *locked()*. *RLock*-объекты поддерживают протокол менеджера контекста.

3. Как выглядит порядок работы с условными переменными?

Основное назначение условных переменных – это синхронизация работы потоков, которая предполагает ожидание готовности некоторого ресурса и оповещение об этом событии. Наиболее явно такой тип работы выражен в паттерне *Producer-Consumer* (Производитель – Потребитель). Условные переменные для организации работы внутри себя используют *Lock*- или *RLock*-объекты, захватом и освобождением которых управлять не придется, хотя и возможно, если возникнет такая необходимость.

4. Какие методы доступны у объектов условных переменных?

- `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:
- `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 отличии 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 равное *True* указывает на то, что барьер находится в “разрушенном” состоянии.