Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №11
дисциплины «Программирование на языке Python»

Вариант_15_

Выполнила:
Маньшина Дарья Алексеевна
2 курс, группа ИТС-б-о-22-1,
11.03.02 «Инфокоммуникационные
технологии и системы связи»,
направленность (профиль)
«Инфокоммуникационные системы и
сети», очная форма обучения

_____

(подпись)

Руководитель практики:
Воронкин Р. А., канд. тех. наук,
доцент кафедры
инфокоммуникаций

_____

(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Тема: синхронизация потоков в языке программирования Python.

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

Ход работы:

1. Подготовка к выполнению работы. Клонирование репозитория и добавление пакетов black, isort, flake8.

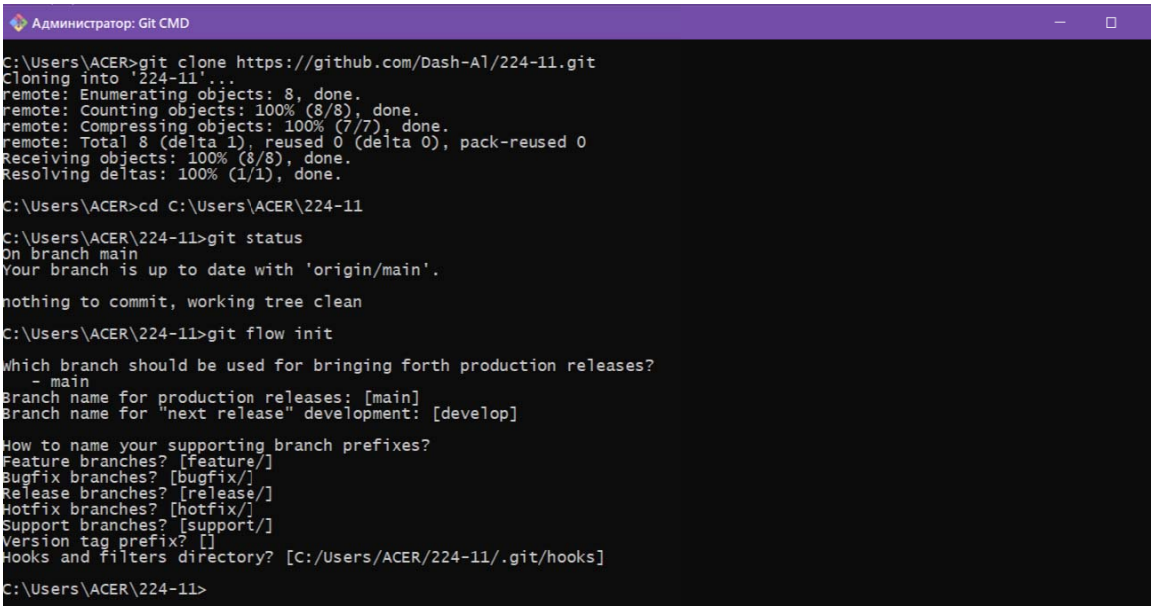

Рисунок 1. Клонирование репозитория

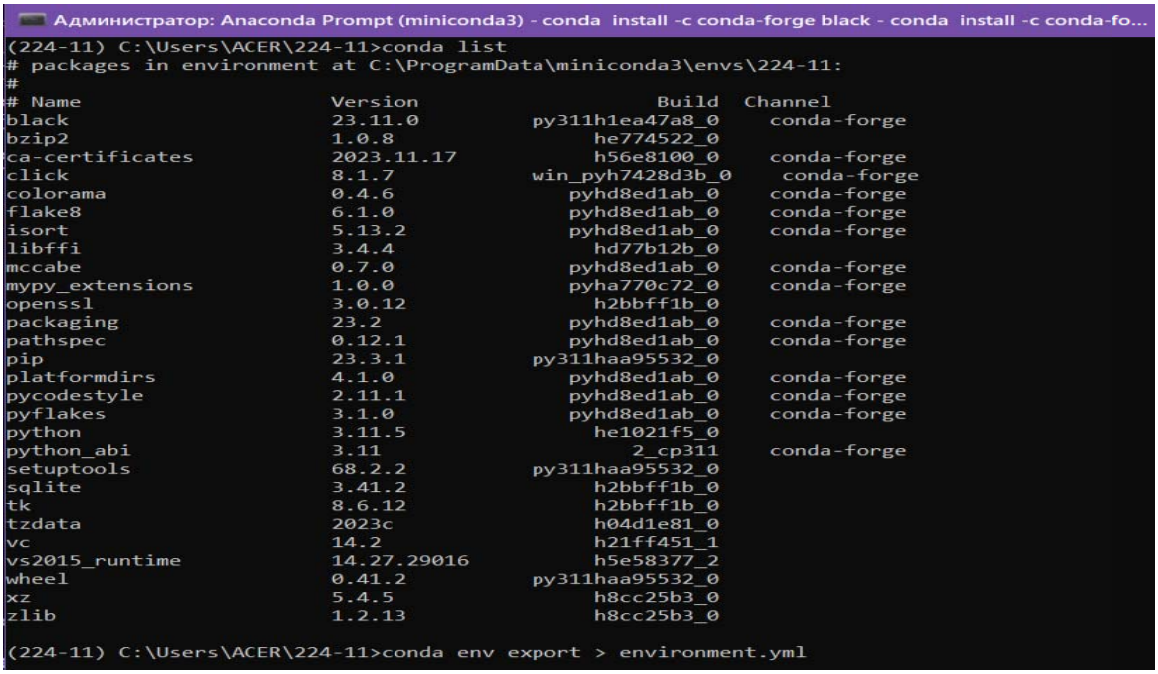

Рисунок 2. Установка пакетов

Проработаем примеры из методички.

Пример 1. В этом примере мы создаем функцию order_processor, которая может реализовывать в себе бизнес логику, например, обработку заказа. При этом, если она получает сообщение stop, то прекращает свое выполнение. В главном потоке мы создаем и запускаем три потока для обработки заказов. Запущенные потоки видят, что очередь пуста и “встают на блокировку” при вызове wait(). В главном потоке в очередь добавляются десять заказов и сообщения для остановки обработчиков, после этого вызывается метод notify_all() для оповещения всех заблокированных потоков о том, что данные для обработки есть в очереди.

Программа:

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

from threading import Condition, Thread
from queue import Queue
from time import sleep

cv = Condition()
q = Queue()
# Consumer function for order processing
def order_processor(name):
    while True:
        with cv:
            # Wait while queue is empty
            while q.empty():
                cv.wait()
            try:
                # Get data (order) from queue
                order = q.get_nowait()
                print(f"{name}: {order}")
                # If get "stop" message then stop thread
                if order == "stop":
                    break
            except:
                pass
            sleep(0.1)


if __name__ == "__main__":
    # Run order processors
    Thread(target=order_processor, args=("thread 1",)).start()
    Thread(target=order_processor, args=("thread 2",)).start()
    Thread(target=order_processor, args=("thread 3",)).start()

    # Put data into queue
    for i in range(10):
```

```
        q.put(f"order {i}")

    # Put stop-commands for consumers
    for _ in range(3):
        q.put("stop")

    # Notify all consumers
    with cv:
        cv.notify_all()
```

Результат:

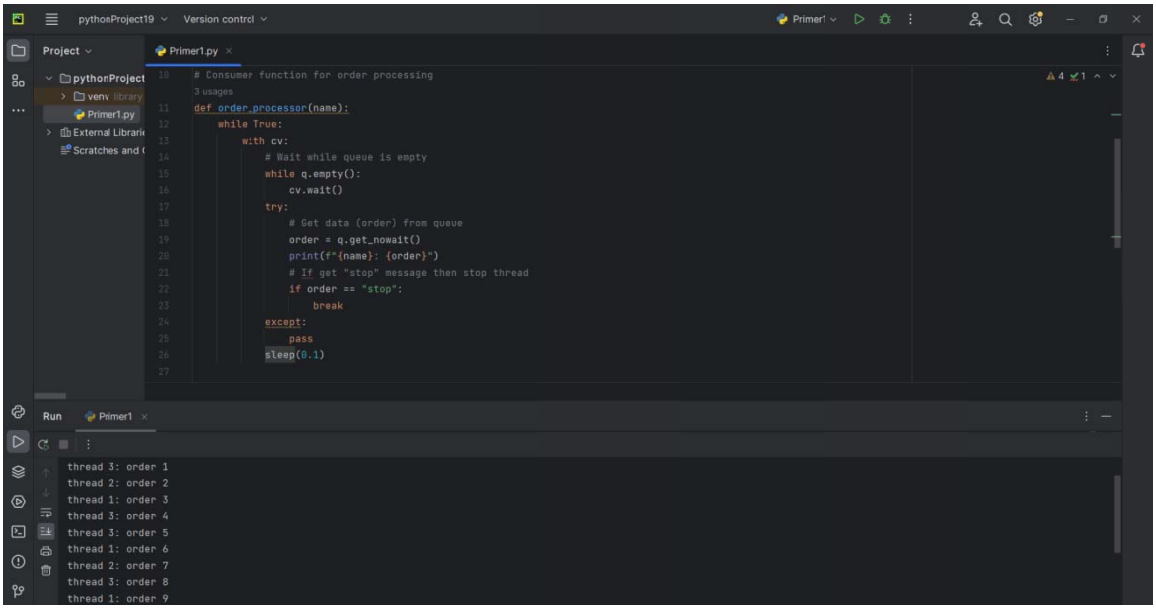

Рисунок 3. Результат примера №1

Пример 2. Работа с Event-объектом

Программа:

```
#!/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__":
    # Clear event
    event.clear()
    # Create and start workers
```

```
workers = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(5)]
for w in workers:
    w.start()
print("Main thread")
event.set()
```

Результат:

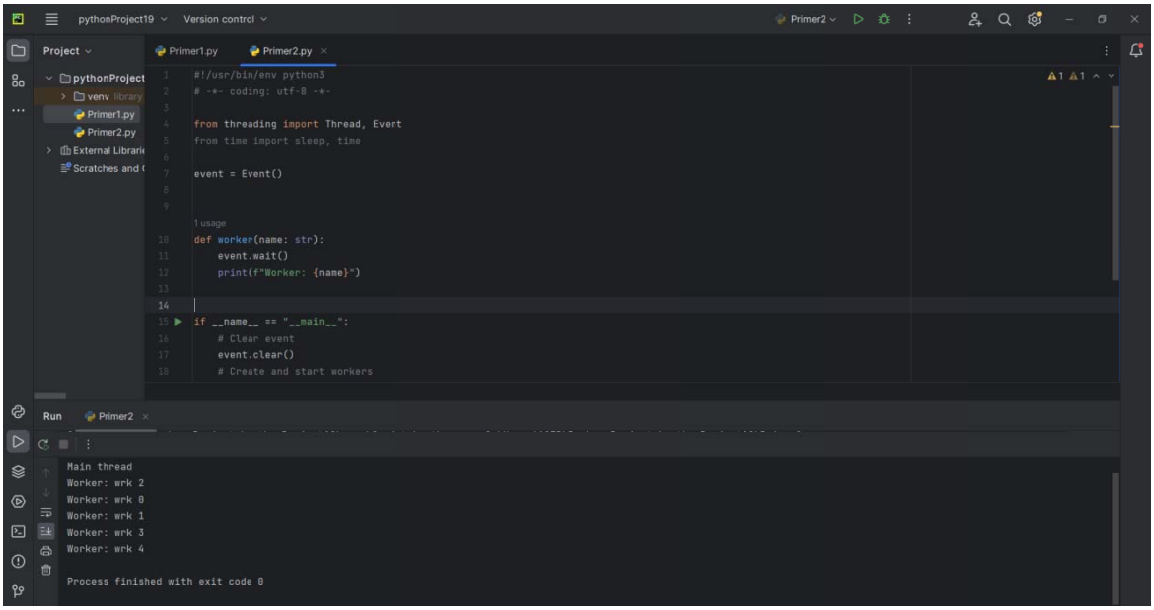

Рисунок 4. Результат примера №2

Пример 3. Работа с таймером

Программа:

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

if __name__ == "__main__":
    from threading import Timer
    from time import sleep, time
    timer = Timer(interval=3, function=lambda: print("Message from Timer!"))
    timer.start()
```

Результат:

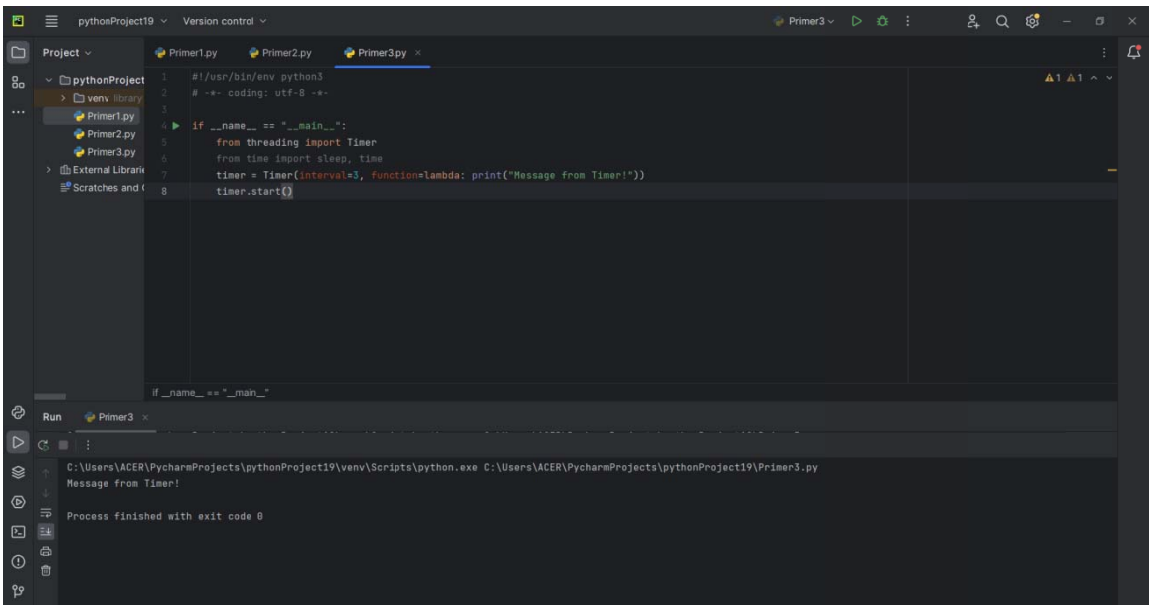

Рисунок 5. Результат примера №3

Пример 4. Работа с классом Barrier

Программа:

```
#!/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))
```

Результат:

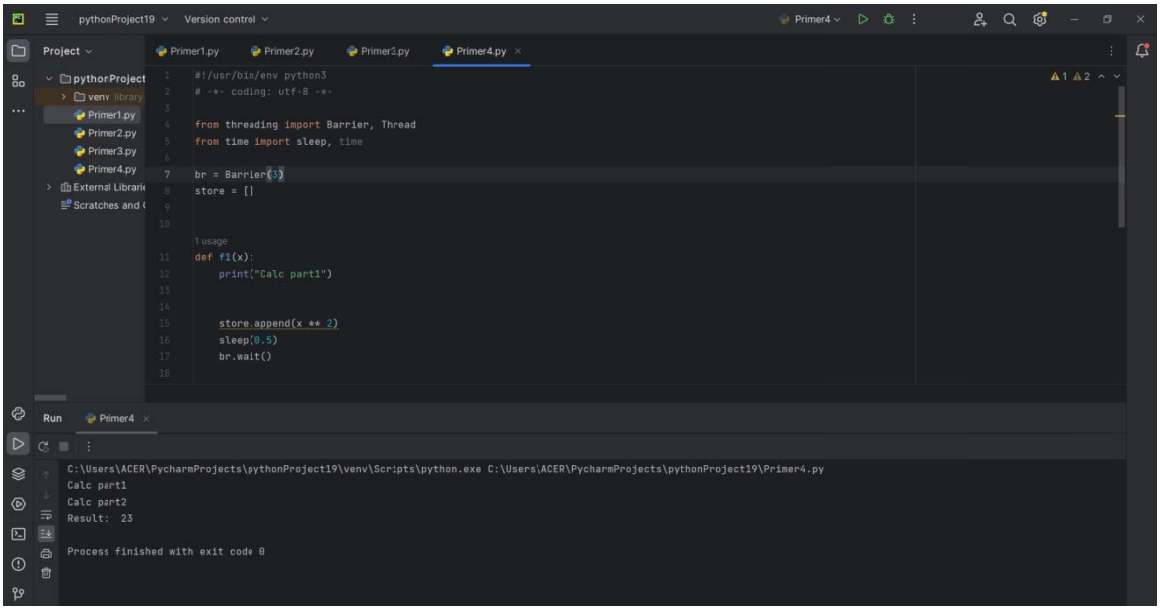

Рисунок 6. Результат примера №4

Решим индивидуальное задание

Условие: 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.

Программа:

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

# 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.

import threading
import math
class Function1(threading.Thread):
    def __init__(self, x, epsilon):
        threading.Thread.__init__(self)
        self.x = x
```

```
        self.epsilon = epsilon
        self.result = 0
    def run(self):
        n = 0
        term = (self.x ** (2 * n)) / math.factorial(2 * n)
        while abs(term) > self.epsilon:
            self.result += term
            n += 1
            term = (self.x ** (2 * n)) / math.factorial(2 * n)
class Function2(threading.Thread):
    def __init__(self, value):
        threading.Thread.__init__(self)
        self.value = value
        self.result = 0
    def run(self):
        # Пример второй функции: экспоненциальная функция от значения первой функции
        self.result = math.exp(self.value)
def main():
    x = 1/2
    epsilon = 1e-7
    thread1 = Function1(x, epsilon)
    thread2 = Function2(0)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    result1 = thread1.result
    result2 = thread2.result
    print(f"Результат первой функции: {result1}")
    print(f"Результат второй функции: {result2}")
if __name__ == "__main__":
    main()
```

Результат:

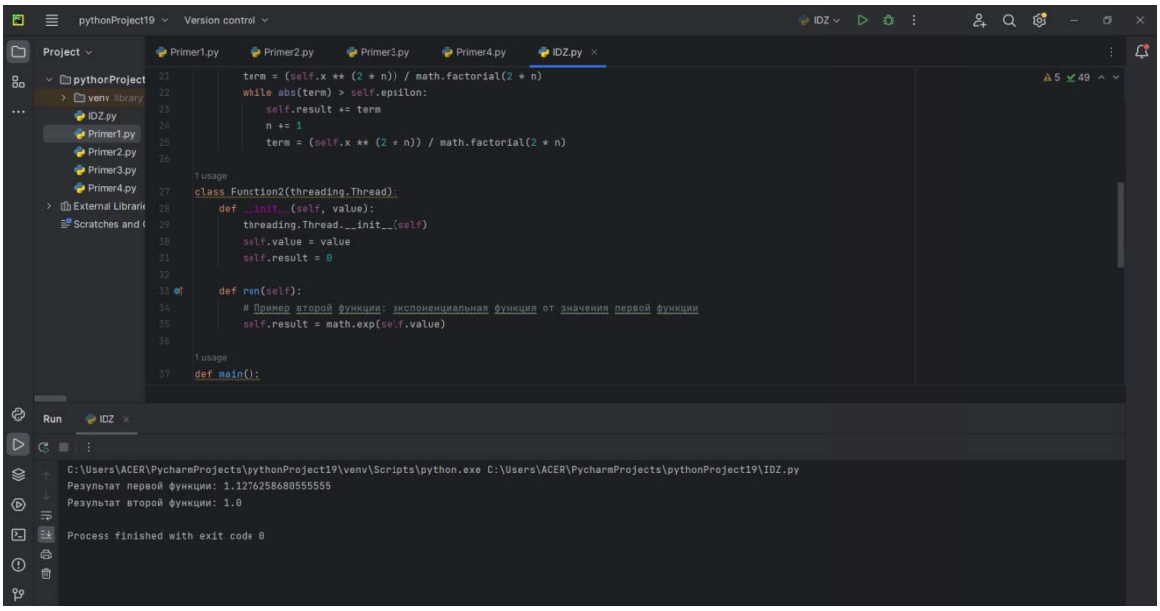

Рисунок 7. Результат индивидуального задания

Вывод: в ходе лабораторной работы приобретены навыки использования примитивов синхронизации в языке программирования Python версии 3.x.

Контрольные вопросы

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

Lock-объект используется для блокировки доступа к разделяемому ресурсу, чтобы обеспечить его последовательное использование. Приемы работы с ним включают:

Создание объекта Lock: lock = threading.Lock()

Блокировка ресурса: lock.acquire()

Освобождение ресурса: lock.release()

Когда один поток блокирует ресурс, другие потоки ожидают, пока блокировка не будет снята.

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

RLock (ReentrantLock) отличается от Lock тем, что позволяет потокам повторно захватывать блокировку, даже если они уже удерживают ее. Это полезно, когда поток выполняет сложную логику внутри блокировки и может потребовать дополнительной блокировки для выполнения определенной части кода.

3. Как выглядит порядок работы с условными переменными?

Работа с условными переменными включает:

Создание: cv = threading.Condition(lock)

Ожидание переменной: cv.acquire(), cv.wait() или cv.notify()

Освобождение переменной: cv.release() или cv.notifyAll()

Условные переменные используются для управления доступом к общим ресурсам в многопоточных приложениях.

4. Какие методы доступны у объектов условных переменных?

Методы условных переменных включают:

acquire() и release() - для блокировки и разблокировки переменной

wait() и notify() - для ожидания изменения переменной и уведомления потока о том, что изменение произошло– notifyAll() - для уведомления всех потоков о том, что переменная изменилась

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

Семафор используется как примитив синхронизации для управления доступом потоков к ресурсу. Он состоит из счетчика и набора “дверей”, которые потоки могут открывать и закрывать. Семафоры полезны, когда нужно контролировать количество потоков, одновременно работающих с ресурсом.

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

Событие используется для оповещения других потоков о том, что произошло какое-то событие. Событие может быть установлено в “поднятое” состояние и “опущено”, и другие потоки могут ожидать его. Доступ к событию осуществляется через методы acquire() и release().

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

Таймер используется для создания задержек или периодических событий в многопоточном приложении. Доступ к таймеру осуществляется через его методы start() и cancel().

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

Барьер используется для синхронизации потоков при достижении определенного места в коде. Барьер устанавливается в определенное место в коде, и все потоки останавливаются, пока все потоки не достигнут барьера. После этого все потоки продолжают работу. Доступ к барьеру осуществляется через метод wait().