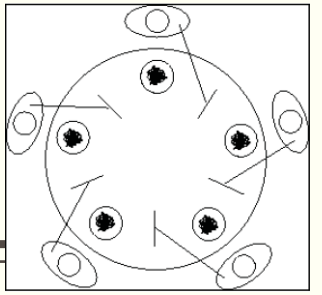
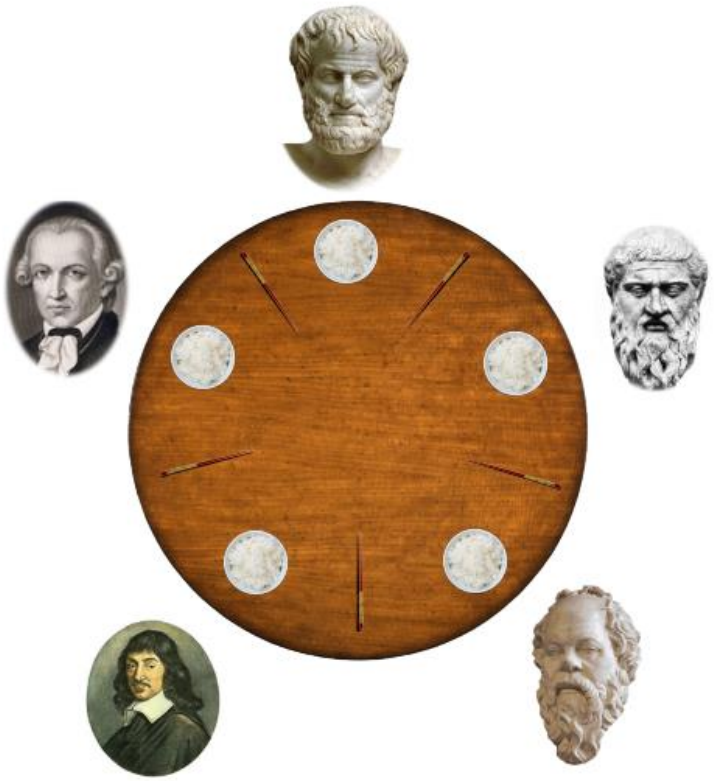
СИНХРОНІЗАЦІЯ ПОТОКІВ

Питання 12.3.

Задача про філософів, які обідають





- Придумана Едсгаром Дейкстрою, офіційно сформульована Тоні Хоаром.
- Розглядаємо 5 філософів, що сидять а круглим столом та їдять спагетті з тарілок.
  - Між тарілками розташовано 5 вилок, якими можна їсти їжу, проте для споживання потрібні ДВІ вилки.
  - Філософ може їсти (якщо має вилки зліва та справа) або думати.
  - Коли філософ бере вилку, сусіди повинні очікувати, поки ця вилка не буде покладена назад.
- Кожен з 5 філософів узяв ліву вилку і тепер очікує, коли стане доступною права.
  - Оскільки жоден з філософів не покладе вилку, поки не з'їсть їжу, виникне ***взаємоблокування (deadlock)***.
  - Ця задача ілюструє основну проблему, з якою можна стикнутись при проєктуванні власних багатопоточних систем, що покладаються на стандартні примітиви синхронізації (locks).
  - Наші вилки виступають системними ресурсами, а кожен філософ представляє конкуруючий процес.

Реалізація задачі

```python
import threading
import time
import random

class Philosopher(threading.Thread):

  def __init__(self, name, leftFork, rightFork):
    print("{} сів за стіл".format(name))
    threading.Thread.__init__(self, name=name)
    self.leftFork = leftFork
    self.rightFork = rightFork

  def run(self):
    print("{} почав думати".format(threading.currentThread().getName()))
    while True:
      time.sleep(random.randint(1,5))
      print("{} закінчив думати".format(threading.currentThread().getName()))
      self.leftFork.acquire()
      time.sleep(random.randint(1,5))
      try:
        print("{} отримав виделку зліва".format(threading.currentThread().getName()))

        self.rightFork.acquire()
        try:
          print("{} володіє обома виделками, зараз їсть".format(threading.currentThread().getName()))
        finally:
          self.rightFork.release()
          print("{} віддав праву виделку".format(threading.currentThread().getName()))
      finally:
        self.leftFork.release()
        print("{} віддав ліву виделку".format(threading.currentThread().getName()))
```

Продовження програми та її вивід

```
33 fork1 = threading.RLock()
34 fork2 = threading.RLock()
35 fork3 = threading.RLock()
36 fork4 = threading.RLock()
37 fork5 = threading.RLock()
38
39 philosopher1 = Philosopher("Кант", fork1, fork2)
40 philosopher2 = Philosopher("Арістотель", fork2, fork3)
41 philosopher3 = Philosopher("Спіноза", fork3, fork4)
42 philosopher4 = Philosopher("Маркс", fork4, fork5)
43 philosopher5 = Philosopher("Рассел", fork5, fork1)
44
45 philosopher1.start()
46 philosopher2.start()
47 philosopher3.start()
48 philosopher4.start()
49 philosopher5.start()
50
51 philosopher1.join()
52 philosopher2.join()
53 philosopher3.join()
54 philosopher4.join()
55 philosopher5.join()
```

- Вивід програми:

```
Кант сів за стіл
Арістотель сів за стіл
Спіноза сів за стіл
Маркс сів за стіл
Кант почав думатиРассел сів за стіл
Арістотель почав думати
Спіноза почав думати

Маркс почав думати
Рассел почав думати
Арістотель закінчив думати
Спіноза закінчив думати
Кант закінчив думати
Арістотель отримав виделку зліва
Маркс закінчив думати
Рассел закінчив думати
Кант отримав виделку зліва
Спіноза отримав виделку зліва
Рассел отримав виделку зліва
Маркс отримав виделку зліва
```

- Далі зависає

Стан гонитви (Race conditions)

- Стандартне визначення:
  - *Стан гонитви – поведінка електронної, програмної або іншої системи, коли вивід залежить від послідовності або тривалості інших неконтрольованих подій.*
- Припустимо, що є банківський рахунок з £2,000 на ньому.
  - Очікуємо отримати бонус £5,000 за виправлення помилки в контексті багатопоточності, яка коштувала замовнику мільйони.
  - Також уявімо, що потрібно заплатити за оренду житла (£1,000) в той же день.
- Розглянемо ситуацію, коли додаток матиме два процеси: Process A (знімання коштів) та Process B (внесення коштів на рахунок).
  - Нехай Process B зчитує баланс рахунку - £2,000.
  - Якщо Process A розпочав знімання грошей відразу після запуску транзакції з Process B, він побачить стартовий баланс £2,000.
  - Process B завершить свою транзакцію та коректно додасть £5,000 до стартового балансу – отримаємо £7,000.
  - Проте оскільки Process A запустив свою транзакцію при початковому балансі £2,000, остаточний баланс буде £1,000.

Вирішення

- Виконувати транзакції з блокуванням.
  - Огорнемо код, який виконує зчитування балансу та його оновлення, за допомогою блокування.
  - Так Process A першим отримає замок для обох операцій, а потім – Process B.
  - Проте синхронізація за допомогою замків перетворює багатопоточний код в однопоточний в місці огортання.
- ***Критична секція*** – будь-яка частина коду, яка отримує доступ або змінює спільний ресурс.
  - Вона не може виконуватись більш, ніж одним процесом в кожен момент часу.
  - Наприклад, пишемо код для банківського додатку.
  - Частина, яка йде від початкового зчитування даних рахунку до внесення змін у нього, може вважатись критичною секцією.
  - Саме при багатопоточному виконанні коду з критичної секції виникає гонка даних.
- Гонитва даних може негативно вплинути і на файлову систему.
  - Наприклад, два процеси будуть намагатись змінити файл.
  - Без контролю за допомогою синхронізації файлова структура може порушитись, файл пошкодиться та втратить свою користь з двома процесами, які в нього записують.

Критичні системи життєзабезпечення

- Один з найгірших випадків програмних помилок, пов'язаних з гонитвою даних, стало програмне забезпечення радіотерапевтичних машин Therac-25.
  - Гонитви даних стало достатньо, щоб спричинити смерть принаймні трьох пацієнтів, які лікувались у цій машині.
- Гонитва даних може спричинити програмні помилки, які дуже важко відстежити та налагодити.
  - Розуміння базових підходів до синхронізації критично важливе для створення потокобезпечних, високопродуктивних програм мовою Python.
  - У модулі threading присутні багато примітивів синхронізації, які допомагають у багатьох конкурентних ситуаціях.

Метод join

```python
import threading
import time

def ourThread(i):
  print("Потік {} запущено\n".format(i))
  time.sleep(i*2)
  print("Потік {} завершено\n".format(i))

def main():
  thread = threading.Thread(target=ourThread, args=(1,))
  thread.start()

  print("Чи потік 1 завершено?\n")

  thread2 = threading.Thread(target=ourThread, args=(2,))
  thread2.start()
  thread2.join()

  print("Потік 2 точно завершив роботу!")

if __name__ == '__main__':
  main()
```

- Об'єкт thread дозволяє відкладати завершення виконання задачі за допомогою методу join().
  - Блокує батьківський потік до того моменту, поки він не підтвердить завершення своєї роботи.
  - Причини можуть бути як природними, так і пов'язаними з виникненням необробленого виключення.
- Крім того, що метод join надає швидкий спосіб забезпечення порядку виконання коду, він може відмінити всі переваги від багатопоточності.
- Що отримали від багатопоточності для thread2?
  - Ми приєднали (joined) його відразу після запуску і загалом заблокували головний потік, поки thread2 не завершить виконання.
  - Багатопоточний додаток перетворився, фактично, в однопоточний під час виконання thread2.

Блокування (Locks)

```python
import threading
import time
import random

counter = 1
lock = threading.Lock()

def workerA():
  global counter
  lock.acquire()
  try:
    while counter < 1000:
      counter += 1
      print("Worker A збільшує лічильник до {}".format(counter))

  finally:
    lock.release()

def workerB():
  global counter
  lock.acquire()
  try:
    while counter > -1000:
      counter -= 1
      print("Worker B зменшує лічильник до {}".format(counter))

  finally:
    lock.release()
```

- Базовий механізм доступу до спільних ресурсів з багатьох потоків виконання.
  - Lock – примітив синхронізації в Python.
  - Можливі стани: "locked" або "unlocked", отримати блокування можна в стані "unlocked".

Блокування (Locks)

```
30 def main():
31   t0 = time.time()
32   thread1 = threading.Thread(target=workerA)
33   thread2 = threading.Thread(target=workerB)
34
35   thread1.start()
36   thread2.start()
37
38   thread1.join()
39   thread2.join()
40
41   t1 = time.time()
42
43   print("Час виконання {}".format(t1-t0))
44
45 if __name__ == '__main__':
46   main()
```

- При початковому запуску потоків вони змагаються за отримання замка (lock), щоб інкрементувати лічильник до 1000 або декрементувати до -1000.
  - Замок буде відкрито (release) тільки тоді, коли один з потоків завершив свою задачу.

```python
import threading
import time

class myWorker():

  def __init__(self):
    self.a = 1
    self.b = 2
    self.rlock = threading.RLock()

  def modifyA(self):
    with self.rlock:
      print("Modifying A : RLock Acquired: {}".format(self.rlock._is_owned()))
      print("{}".format(self.rlock))
      self.a = self.a + 1
      time.sleep(5)

  def modifyB(self):
    with self.rlock:
      print("Modifying B : RLock Acquired: {}".format(self.rlock._is_owned()))
      print("{}".format(self.rlock))
      self.b = self.b - 1
      time.sleep(5)

  def modifyBoth(self):
    with self.rlock:
      print("Rlock acquired, modifying A and B")
      print("{}".format(self.rlock))
      self.modifyA()
      print("{}".format(self.rlock))
      self.modifyB()
    print("{}".format(self.rlock))


workerA = myWorker()
workerA.modifyBoth()
```

RLocks

- Reentrant-locks, or RLocks – синхронізаційний примітив, який працює схоже до стандартного блокування, проте може отримуватись потоком багато разів, якщо цей потік вже володіє ним.
  - Нехай thread-1 отримує RLock, лічильник в примітиві збільшиться на 1 кожного разу, коли thread-1 знову отримуватиме цей замок.
  - Якщо thread-2 спробував отримати RLock, він буде очікувати, поки лічильник Rlock дійде до нуля 0.
- У чому користь?
  - Наприклад, при бажанні мати потокобезпечний доступ до методу в класі, який отримує доступ до інших методів класу.

Вивід програми

```
Rlock acquired, modifying A and B
<locked _thread.RLock object owner=4780 count=1 at 0x0000029C11D56CD8>
Modifying A : RLock Acquired: True
<locked _thread.RLock object owner=4780 count=2 at 0x0000029C11D56CD8>
<locked _thread.RLock object owner=4780 count=1 at 0x0000029C11D56CD8>
Modifying B : RLock Acquired: True
<locked _thread.RLock object owner=4780 count=2 at 0x0000029C11D56CD8>
<unlocked _thread.RLock object owner=0 count=0 at 0x0000029C11D56CD8>
```

- На кожному кроці виводимо стан нашого Rlock.
  - Після його отримання в функції modifyBoth(), власником встановлюється головний потік, а лічильник збільшується на 1.
  - Коли потім викликається modifyA(), лічильник Rlock знову збільшується на 1, а необхідні обчислення здійснюються до того, як modifyA() звільнить Rlock.
  - До моменту звільнення функцією modifyA() об’єкту Rlock спостерігаємо зменшення лічильника на 1 та негайне збільшення до 2 знову в функції modifyB().
  - Наприкінці, коли завершує своє виконання modifyB(), вона звільняє Rlock, після чого те ж робить і функція modifyBoth().

Умова (Condition)

- Синхронізаційний примітив, який очікує сигналу від іншого потоку.
  - Наприклад, інший потік закінчив виконання, а поточний потік може продовжувати роботу.
  - Найбільш поширений сценарій використання – виробник/споживач.
  - Виробник публікує повідомлення в чергу та сповіщає інші потоки (споживачі), що повідомлення очікують на споживання в черзі.
- Створимо 2 окремих класи, успадковані від класу thread: Publisher та Subscriber.
  - Виробник виконає задачу з публікації нових цілих чисел у цілочисельний масив, а потім сповістить підписників, що існує нове ціле число, яке можна отримати з масиву.

Клас Publisher

```
class Publisher(threading.Thread):

  def __init__(self, integers, condition):
    self.condition = condition
    self.integers = integers
    threading.Thread.__init__(self)

  def run(self):
    while True:
      integer = random.randint(0,1000)
      self.condition.acquire()
      print("Condition Acquired by Publisher: {}".format(self.name))
      self.integers.append(integer)
      print("Publisher {} appending to array: {}".format(self.name, integer))
      self.condition.notify()
      print("Condition Released by Publisher: {}".format(self.name))
      self.condition.release()
      time.sleep(1)
```

- Функція run() загалом входить у нескінченний цикл та генерує випадкові числа з проміжку від 0 до 999.
  - Далі отримуємо умову (condition), після чого додаємо згенероване ціле число в масив цілих чисел.
  - Тепер сповістимо підписників, що з'явився новий елемент в масиві та звільняємо умову (release the condition).

Клас Subscriber

```
def main():
  integers = []
  condition = threading.Condition()

  # Our Publisher
  pub1 = Publisher(integers, condition)
  pub1.start()

  # Our Subscribers
  sub1 = Subscriber(integers, condition)
  sub2 = Subscriber(integers, condition)
  sub1.start()
  sub2.start()

  ## Joining our Threads
  pub1.join()
  consumer1.join()
  consumer2.join()

if __name__ == '__main__':
  main()
```

```
class Subscriber(threading.Thread):

  def __init__(self, integers, condition):
    self.integers = integers
    self.condition = condition
    threading.Thread.__init__(self)

  def run(self):
    while True:
      self.condition.acquire()
      print("Condition Acquired by Consumer: {}".format(self.name))
      while True:
        if self.integers:
          integer = self.integers.pop()
          print("{} Popped from list by Consumer: {}".format(integer, self.name))
          break
        print("Condition Wait by {}".format(self.name))
        self.condition.wait()
      print("Consumer {} Releasing Condition".format(self.name))
      self.condition.release()
```

- Визначаємо одного виробника та два підписника.
  - Запускаємо виробника та підписників, а також з'єднуємо (join) потоки, щоб програма не переривалась до того моменту, як потоки зможуть розпочати роботу.

Результати виконання

```
Condition Acquired by Publisher: Thread-1
Publisher Thread-1 appending to array: 108
Condition Released by Publisher: Thread-1
Condition Acquired by Consumer: Thread-2
108 Popped from list by Consumer: Thread-2
Consumer Thread-2 Releasing Condition
Condition Acquired by Consumer: Thread-2
Condition Wait by Thread-2
Condition Acquired by Consumer: Thread-3
Condition Wait by Thread-3
Condition Acquired by Publisher: Thread-1
Publisher Thread-1 appending to array: 563
...
```

- Коли виробник отримує умову, число дописується в масив, а потім відбувається сповіщення щодо умови та її звільнення.
  - Як тільки умова була виголошена, починається боротьба між підписниками за неї.
  - Коли хтось з них перемагає, він просто виймає ("pop«) це число з масиву.

Семафори

- Мають внутрішній лічильник, який інкрементується чи декрементується, як тільки здійснено виклик на отримання або звільнення замка.
  - При ініціалізації значення лічильника за замовчуванням дорівнює 1.
  - Семафор не можна отримати, якщо лічильник буде від'ємним.
- Нехай блок коду захищений семафором із значенням лічильника 2.
  - Якщо один потік отримує семафор, значення його лічильника зменшується на 1.
  - Якщо інший потік отримує семафор, лічильник набуде значення 0.
  - Тепер, якщо семафор буде відхиляти звернення на отримання від інших потоків, поки один з початкових потоків не викличе метод release() та лічильник не буде збільшено на 1.
- Створимо просту програму для продажу квитків.
  - Передбачено 4 окремих потоки, кожен з яких намагається продати якомога більше квитків з визначеної кількості.

Клас TicketSeller

```python
import threading
import time
import random

class TicketSeller(threading.Thread):
  ticketsSold = 0

  def __init__(self, semaphore):
    threading.Thread.__init__(self);
    self.sem = semaphore
    print("Продавець квитків почав роботу")

  def run(self):
    global ticketsAvailable
    running = True
    while running:
      self.randomDelay()

      self.sem.acquire()
      if(ticketsAvailable <= 0):
        running = False
      else:
        self.ticketsSold = self.ticketsSold + 1
        ticketsAvailable = ticketsAvailable - 1
        print("{} Продано квиток ({} залишилось)".format(self.getName(), ticketsAvailable))
      self.sem.release()
    print("Продавець квитків {} продав загалом {} квитків".format(self.getName(),
          self.ticketsSold))

  def randomDelay(self):
    time.sleep(random.randint(0,4)/4)
```

- Спочатку напишемо клас ***TicketSeller***, який міститиме внутрішній лічильник кількості проданих квитків.
  - У конструкторі ініціалізуємо потік та посилання на семафор.
  - У функції run() намагаємось отримати цей семафор.
  - Якщо кількість квитків для продажу перевищує 0, додаємо 1 до кількості проданих квитків ticketSeller та зменшуємо ticketsAvailable на 1.
- Після цього звільняємо семафор та виводимо прогрес роботи.

Моделюємо роботу продавців

```
# семафор
semaphore = threading.Semaphore()
# початкова кількість квитків
ticketsAvailable = 200

# масив продавців
sellers = []
for i in range(4):
  seller = TicketSeller(semaphore)
  seller.start()
  sellers.append(seller)

# чекаємо всіх продавців
for seller in sellers:
  seller.join()
```

```
Thread-2 Продано квиток (22 залишилось)
Thread-3 Продано квиток (21 залишилось)
Thread-4 Продано квиток (20 залишилось)
Thread-4 Продано квиток (19 залишилось)
Thread-4 Продано квиток (18 залишилось)
Thread-4 Продано квиток (17 залишилось)
Thread-4 Продано квиток (16 залишилось)
Thread-3 Продано квиток (15 залишилось)
Thread-5 Продано квиток (14 залишилось)
Thread-2 Продано квиток (13 залишилось)
Thread-3 Продано квиток (12 залишилось)
Thread-4 Продано квиток (11 залишилось)
Thread-2 Продано квиток (10 залишилось)
Thread-5 Продано квиток (9 залишилось)
Thread-5 Продано квиток (8 залишилось)
Thread-4 Продано квиток (7 залишилось)
Thread-2 Продано квиток (6 залишилось)
Thread-2 Продано квиток (5 залишилось)
Thread-2 Продано квиток (4 залишилось)
Thread-3 Продано квиток (3 залишилось)
Thread-5 Продано квиток (2 залишилось)
Thread-5 Продано квиток (1 залишилось)
Thread-4 Продано квиток (0 залишилось)
Продавець квитків Thread-3 продав загалом 43 квитків
Продавець квитків Thread-2 продав загалом 50 квитків
Продавець квитків Thread-4 продав загалом 51 квитків
Продавець квитків Thread-5 продав загалом 56 квитків
```

- Якщо видалити введене блокування потоку (закоментувати self.randomDelay у функції run()), при запуску програми потік, який першим отримає семафор, скоріше за все, продасть усі квитки.
  - Потік, який першим здобуде семафор, може розпродати квитки до того моменту, поки запити на семафор виконають інші потоки.

Прикріплені семафори (Bounded semaphores)

- Прикріплений семафор перевіряє, чи поточне значення лічильника не перевищує його початкове значення.
  - Якщо перевищує, викидається виняток ValueError.
  - Зазвичай семафори використовуються для збалансування використання обмежених ресурсів.
- Якщо семафор звільняється надто багато разів, це знак щодо наявності бага.
  - Якщо значення не задається, за умовчанням вона рівне 1.
  - Прикріплені семафори зазвичай можна побачити у реалізаціях веб-сервера чи бази даних з метою захисту від вичерпання ресурсу у випадку одночасного напливу клієнтів для підключення або деякої іншої дії.
- Загалом використання прикріплених семафорів замість звичайних є кращою ідеєю.
  - Якщо замінити відповідний код у прикладі з Semaphore на threading.BoundedSemaphore(4) та перезапустити програму, отримаємо практично таку ж поведінку, за винятком захищеності від дуже простих помилок програмування, які інакше можуть залишитись необробленими.

Події (Events)

- За допомогою подій один потік зазвичай буде сигналізувати про настання події, а інший – активно прослуховуватиме цей сигнал.
  - Event-події в своїй основі мають булевий прапорець.
  - У потоках можемо постійно опитувати цей об’єкт-подію щодо його стану.
  - Потім обирається реакція на зміну стану цього прапорця.
- Мова Python не має реальних нативних механізмів для «вбивства» потоків.
  - Проте за допомогою об’єктів-подій потоки можуть працювати лише стільки часу, скільки об’єкт-подія буде залишатись невизначеною (unset).
  - У системах з доступним сигналом SIGKILL це не так важливо, проте корисно в деяких випадках, коли потрібно обережно припинити роботу, чекаючи, поки потік закінчить приготування до зупинки.

Event-об’єкт володіє 4 публічними функціями, які можуть його змінювати та використовувати

```python
import threading
import time

def myThread(myEvent):
  while not myEvent.is_set():
    print("Waiting for Event to be set")
    time.sleep(1)
  print("myEvent has been set")

def main():
  myEvent = threading.Event()
  thread1 = threading.Thread(target=myThread, args=(myEvent,))
  thread1.start()
  time.sleep(10)
  myEvent.set()

if __name__ == '__main__':
  main()
```

- ***isSet():*** перевіряє, чи визначено подію
- ***set():*** визначає подію
- ***clear():*** «скидує» об’єкт-подію
- ***wait():*** блокує роботу, поки внутрішній прапорець буде мати значення true

Бар’єри

```
1 import threading
2 import time
3 import random
4
5 class myThread(threading.Thread):
6
7     def __init__(self, barrier):
8       threading.Thread.__init__(self)
9       self.barrier = barrier;
10
11    def run(self):
12      print("Thread {} working on something".format(threading.current_thread()))
13      time.sleep(random.randint(1,10))
14      print("Thread {} is joining {} waiting on Barrier".format(
15            threading.current_thread(), self.barrier.n_waiting))
16      self.barrier.wait()
17
18      print("Barrier has been lifted, continuing with work")
19
20
21 barrier = threading.Barrier(4)
22
23 threads = []
24
25 for i in range(4):
26   thread = myThread(barrier)
27   thread.start()
28   threads.append(thread)
29
30 for t in threads:
31     t.join()
```

- Синхронізаційні примітиви, представлені в Python 3, які вирішують проблему, що розв’язується великою кількістю умов та семафорів.
  - Бар’єри встановлюють контрольні точки для груп потоків, які можуть працювати лише до досягнення цих точок.
- Використаємо бар’єри для блокування роботи потоків до моменту досягнення всіма потоками деякої точки виконання.

Вивід програми

```
Thread <myThread(Thread-2, started 2600)> working on something
Thread <myThread(Thread-3, started 16420)> working on something
Thread <myThread(Thread-4, started 14600)> working on something
Thread <myThread(Thread-5, started 8068)> working on something
Thread <myThread(Thread-4, started 14600)> is joining 0 waiting on Barrier
Thread <myThread(Thread-5, started 8068)> is joining 1 waiting on Barrier
Thread <myThread(Thread-2, started 2600)> is joining 2 waiting on Barrier
Thread <myThread(Thread-3, started 16420)> is joining 3 waiting on Barrier
>> >Barrier has been lifted, continuing with work

Barrier has been lifted, continuing with work
Barrier has been lifted, continuing with work
Barrier has been lifted, continuing with work
```

- Наші 4 потоки виводять повідомлення, що чимось зайняті, а потім один за одним випадковим чином запускають очікування об’єкта-бар’єра.
  - Як тільки 4й потік починає очікування, програма майже миттєво завершується, оскільки всі 4 потоки підійшли до бар’єру.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Екзекутори та пули потоків