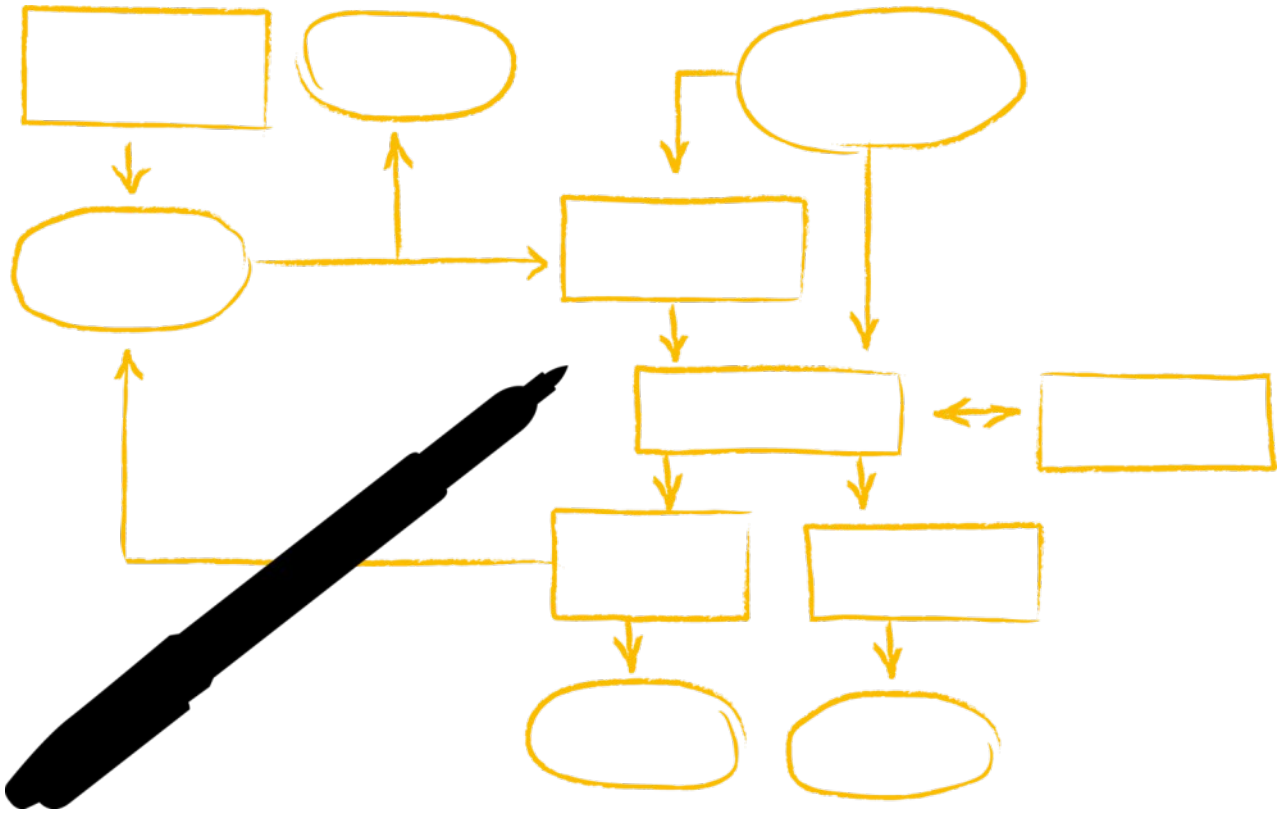
Многонишково програмиране

Основи

Владимир Панов

Ноември, 2014

Public

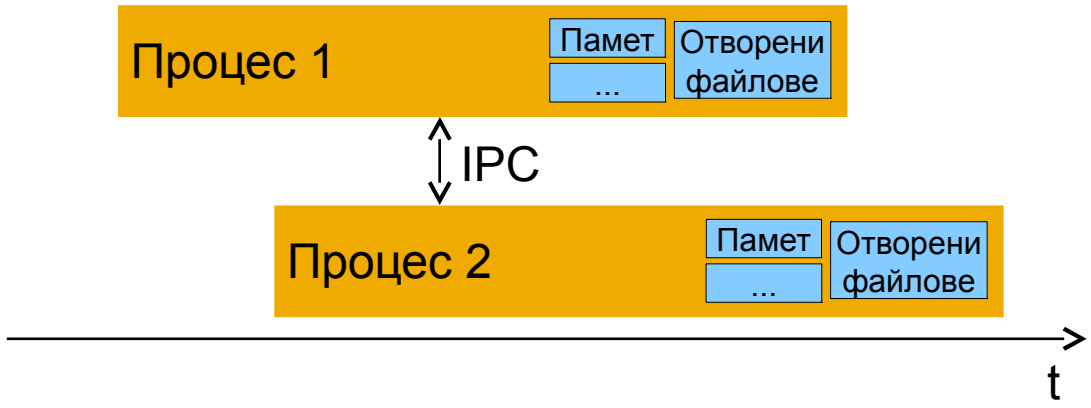


Процеси

Процес: Инстанция на компютърна програма в процес на изпълнение.

Процесите в рамките на една компютърна система:

- се изпълняват паралелно.
- са изолирани един от друг от операционната система.
- могат да комуникират помежду си само чрез специални средства (IPC – interprocess communication).

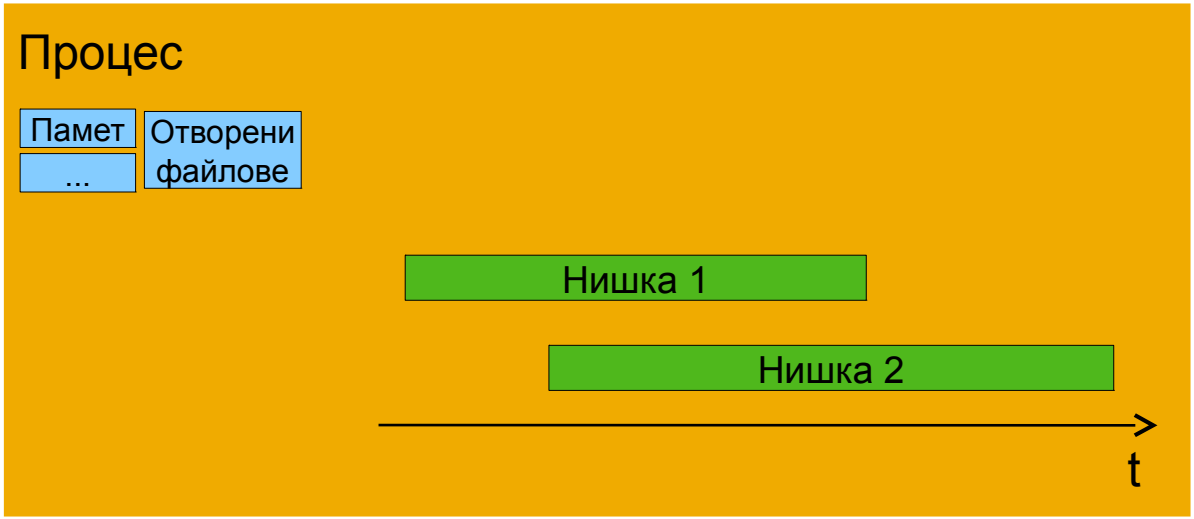


Нишки

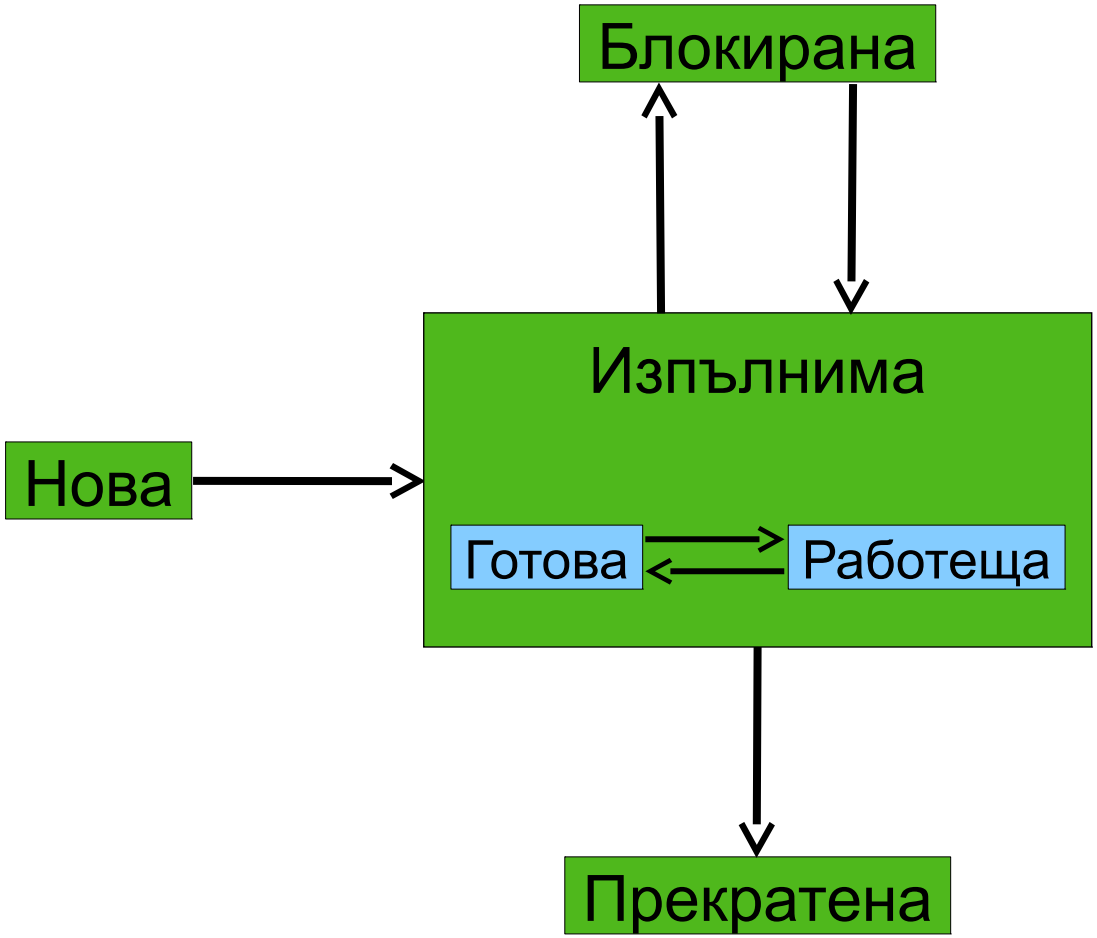
Нишка: Изпълнение на част от компютърна програма в рамките на един процес.

Нишките в рамките на един процес:

- се изпълняват паралелно.
- имат пълен достъп до всички ресурси на процеса (няма изолация).
- трябва да се грижат сами за синхронизацията помежду си.



Състояние на нишка



Процеси срещу нишки

	Процеси	Нишки
Стартиране	Бавно	Относително бързо
Изолация	Да	Не
Синхронизация	Лесна (под грижата на ОС, чрез IPC)	Трудна (под грижата на самата програма*)
Комуникация	Бърза (чрез IPC)	Мигновена (чрез общи ресурси)

* т.е. на приложния програмист

Какъв е смисълът?

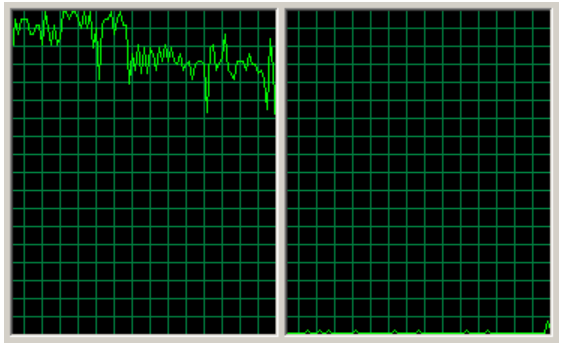
Целта на многозадачността е да се уплътни употребата на хардуерните ресурси, които могат да се позват паралелно, напр.:

- процесор и вход/изход
- няколко процесора
- вход/изход от няколко външни устройства
- вход/изход от няколко външни устройства и няколко процесора

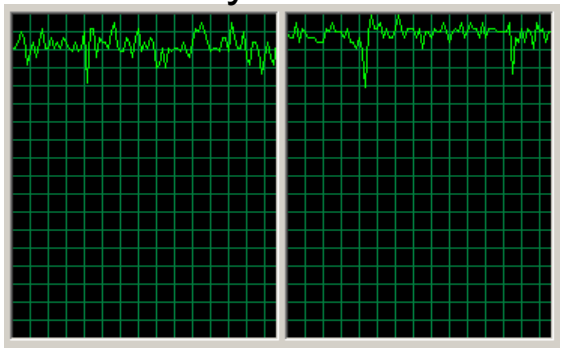
Нишките са измислени за да компенсират недостатъците на процесите там, където са приложими (в рамките на един процес).

Нишките не са измислени за да улеснят живота на програмиста.

Лошо:



Хубаво:



Увод

JVM първоначално има една нишка, наречена **main**. В нея се извиква методът **main(String args[])**.

Има два вида нишки:

- Обикновени – JVM се прекратява, когато и последната обикновена нишка бъде прекратена.
- Демонови – тези нишки не са критерий за прекратяването на JVM.

Събирането на боклук (garbage collection) се случва в специална демонова нишка.

Нишките в Java са инстанции на класове, наследници на **java.lang.Thread**.

Имплементиране и инстанциираne

Методът **run()** съдържа изпълнявания в нишката код.

```
public class MyThread extends Thread {
    public void run() {
        // кодът на нишката
    }
}

// инстанциираne:
Thread myThread = new MyThread();  // нишката е в състояние "нова"
```

Имплементиране и инстанцииране (алтернатива)

Нишка може да се имплементира и чрез интерфейса **java.lang.Runnable**.

```
public class MyRunnable implements Runnable extends MyBaseClass {
    public void run() {
        // кодът на нишката
    }
}

// инстанцииране:
Thread myThread = new Thread(new MyRunnable());
```

Стартиране и прекратяване

Нишките се стартират (преминават в изпълнимо състояние) чрез извикване на метода **start()**.

Една нишка може да бъде прекратена само по един начин: методът **run()** да върне управлението.

```
Thread myThread = ...;
myThread.start();
// нишката ще бъде прекратена
// след като методът run() завърши изпълнението си
```

Стартиране на няколко нишки
Пример

```
public class ThreadTest {

  private static class MyThread1 extends Thread {
    public void run() {
      System.out.println("Нишка 1 се стартира.");
      for (int i = 0; i < 5; i++) {
        System.out.println("Нишка 1 върши работа.");
        Math.cbrt(Double.MIN_VALUE);
      }
      System.out.println("Нишка 1 се прекратява.");
    }
  }

  private static class MyThread2 extends Thread {
    public void run() {
      System.out.println("Нишка 2 се стартира.");
      for (int i = 0; i < 5; i++) {
        System.out.println("Нишка 2 върши работа.");
        Math.cbrt(Double.MAX_VALUE);
      }
      System.out.println("Нишка 2 се прекратява.");
    }
  }

  public static void main(String args[]) {
    Thread myThread1 = new MyThread1();
    Thread myThread2 = new MyThread2();
    myThread1.start();
    myThread2.start();
  }
}
```

Възможен изход:

```
Нишка 1 се стартира.
Нишка 2 се стартира.
Нишка 2 върши работа.
Нишка 1 върши работа.
Нишка 2 върши работа.
Нишка 1 върши работа.
Нишка 2 върши работа.
Нишка 1 върши работа.
Нишка 2 върши работа.
Нишка 2 се прекратява.
Нишка 1 върши работа.
Нишка 1 върши работа.
Нишка 1 върши работа.
Нишка 1 върши работа.
Нишка 1 се прекратява.
```

Отстъпване на други нишки

Когато една нишка прецени, че е свършила достатъчно работа към даден момент, тя може да уведоми JVM, че е ОК процесорът да бъде даден на друга нишка.

Отстъпването е пожелателно и за неопределено време, т.е. JVM може веднага да върне управлението на нишката, ако няма други желаещи.

```
public void run() {
    ... // работа
    Thread.yield();
    ... // още работа
}
```

Спане на нишка

Когато една нишка прецени, че трябва да спре изпълнението си за известно време, тя може да поспи. Така тя освобождава процесора за други нишки.

Спането е пожелателно – нишката може да бъде прекъсната от друга нишка, или пък да бъде оставена от JVM да спи малко повече или малко по-малко от пожеланото време.

```
public void run() {
    ... // работа
    try {
        Thread.sleep(2000);   // в милисекунди
    } catch (InterruptedException e) {
        // прекъснати от друга нишка насред съня
    }
    ... // още работа
}
```

Присъединяване към нишка

Една нишка може да се присъедини към друга, т.е. да я изчака да завърши.

```
Thread anotherThread = ...;

public void run() {
  ...
  try {
    anotherThread.join();
    ... // anotherThread гарантирано е прекратена
  } catch (InterruptedException e) {
    // тази нишка е прекъсната от друга нишка насред чакането
    // anotherThread може и да не е прекратена!
  }
}
```

Присъединяване към нишка (алтернатива)

```
Thread anotherThread = ...;

public void run() {
    ...
    try {
         anotherThread.join(5000);    // чакаме най-много 5 s
    } catch (InterruptedException e) {
        // прекъснати от друга нишка насред чакането
    }
    if (anotherThread.isAlive()) {
        // не сме дочакали,
        // anotherThread може и да не е прекратена
    }
    ...
}
```

Прекъсване на нишка

Прекъсването е генерален механизъм, който дава възможност на една нишка да привлече вниманието на друга. Това става чрез извикване на метода **Thread.interrupt()** на прекъсваната нишка.

Прекъсваната нишка може:

- да игнорира прекъсването.
- да изпълни някакво действие (напр. прекратяване).

Някои блокиращи методи (**Thread.sleep(…)**, **Thread.join(…)**, **Object.wait(…)** и др.) изхвърлят **InterruptedException**, ако нишката бъде прекъсната по време на тяхното изпълнение.

В останалите случаи нишката трябва сама да проверява дали е била прекъсната чрез един от следните методи:

- **Thread.interrupted()** – проверява дали нишката е прекъсната и изчиства статуса на прекъсването (нишката престава да се води прекъсната).
- **Thread.isInterrupted()** – само проверява дали нишката е прекъсната.

Прекъсване на нишка с цел прекратяване (пример)

```
class InterruptibleThread extends Thread {
  public void run() {
    try {
      while (true) {
        System.out.println("Прекъсваемата нишка върши работа.");
        for (int i = 0; i < 50000000; i++) {
          if (interrupted()) {
            System.out.println("Прекъсваемата нишка беше прекъсната насред работа.");
            return;
          }
          Math.cbrt(Double.MAX_VALUE);
        }
        System.out.println("Прекъсваемата нишка ще поспи за около 5 секунди.");
        Thread.sleep(5000);
      }
    } catch (InterruptedException e) {
      System.out.println("Прекъсваемата нишка беше прекъсната докато е блокирана.");
      return;
    }
  }
}

...

Thread intThread = new InterruptibleThread();
intThread.start();
...
intThread.interrupt();
intThread.join();
System.out.println("Прекъсваемата нишка е успешно прекратена.");
```

Приоритети на нишки

Всяка една нишка има приоритет – число между **Thread.MIN_PRIORITY** (1) и **Thread.MAX_PRIORITY** (10).

Колкото по-голям е приоритетът на една нишка, толкова повече процесорно време ще получава тя (ако въобще има недостиг).

Приоритетите на нишките се задават от самата програма (чрез метода **Thread.setPriority(int)**), но решението кога точно една нишка да се изпълнява и кога да отстъпи на други нишки е на JVM.

Програмистът не трябва да прави никакви приоритетно-базирани предположения относно изпълнението на нишките.

Кога има нужда от синхронизиране?

Синхронизация се налага винаги, когато един ресурс може да бъде достъпван от повече от една нишка в даден момент.

Ресурси, които подлежат на синхронизиране:

- Член-променливи
- Статични променливи
- Локални променливи, реферирaщи нелокални обекти
- Отворени файлове
- Отворени мрежови сокети
- GUI ресурси
- Общо взето всичко

Ресурси, които няма нужда да се синхронизират:

- Локални променливи, но само ако са от примитивен тип или реферират новосъздадени обекти, до които други нишки все още нямат достъп.

Кога има нужда от синхронизиране? (пример)

Нека две нишки (А и Б) едновременно извикват следния метод:

```
private int lastId = 0;
public int getUniqueId() {
  lastId++;
  return lastId;
}
```

Извършител	Действие	Стойност на *lastId*
-	Начало	0
Нишка А	*lastId*++	1
Нишка Б	*lastId*++	2
Нишка А	*return lastId*	2
Нишка Б	*return lastId*	2

Двете нишки получават един и същ “уникален” идентификатор.

Синхронизация на метод

```
synchronized return_type method(parameters) {
   ... // работа
}
```

Само един синхронизиран метод от даден клас може да се изпълнява върху една инстанция на класа в даден момент.

Всички останали нишки, чието изпълнение е стигнало до синхронизиран метод за същата инстанция, са блокирани през това време и не използват процесор.

Когато нишката завърши изпълнението на синхронизирания метод, някоя от чакащите нишки ще бъде събудена. Другите ще продължат да бъдат блокирани.

Не може да се предвиди коя точно от чакащите нишки ще бъде събудена, макар и вероятността нишка с по-висок приоритет да бъде събудена е по-голяма.

Корекция на примера

```
public synchronized int getUniqueId() { ...
```

Извършител	Действие	Стойност на *lastId*
-	Начало	0
Нишка А	влиза в синхронизирания метод	0
Нишка Б	блокира	0
Нишка А	*lastId*++	1
Нишка А	*return lastId*	1
Нишка А	излиза от синхронизирания метод	1
Нишка Б	влиза в синхронизирания метод	1
Нишка Б	*lastId*++	2
Нишка Б	*return lastId*	2
Нишка Б	излиза от синхронизирания метод	2

Ключалка (монитор)

Всеки обект в Java има неявна ключалка (наричана още монитор). Ключалката и методите за манипулирането и се намират в **java.lang.Object.**

В даден момент само една нишка може да държи даден обект заключен (да притежава монитора му).

Ако една нишка се опита да заключи обект, който в момента е заключен от друга нишка, то тя ще бъде блокирана.

В момента, в който една нишка отключи даден обект (освободи монитора му), някоя от блокираните нишки, чакащи да заключат обекта, ще бъде събудена и заедно с това ще заключи обекта. Ако няма други чакащи нишки, обектът ще остане отключен.

Една нишка може да заключи повече от един обект в даден момент (да притежава мониторите на повече от един обект).

Ключалките в Java са рекурсивни – ако една нишка се опита да заключи обект, който вече самата тя е заключила, то тя няма да се самоблокира.

Синхронизация на блок по обект

Случаи, за които синхронизацията на метод не е подходяща:

- Нуждаещото се от синхронизация парче код е малка част от метод.
- Повече от една инстанция на един клас или няколко класа трябва да се синхронизират.
- Парче код има нужда да се синхронизира по повече от един ресурс.

```
Object obj = ...;
...
synchronized (obj) {  // заключване на obj
  ...  // obj е заключен
}  // отключване на obj
```

Синхронизация на блок по обект (пример)

```
void methodA(List<Integer> list) {
  // трябва да изтрием първия елемент
  synchronized (list) {
    if (!list.empty()) list.remove(0);
  }
}
void methodB(List<Integer> list) {
  synchronized (list) {
    float average = 0;
    if (list.size() > 0) {
      for (Integer i : list) average += i.intValue();
      // ако няма синхронизация, тук би могло да делим на 0:
      average /= list.size();
    }
    ... // ако няма синхронизация, average може вече да е с невалидна стойност
  }
}
```

Връзка между двата вида синхронизация

```
class C {

  synchronized T1 methodA() {

    ...

  }

  synchronized static T2 methodB() {

    ...

  }

}
```

```
class C {

  T1 methodA() {
    synchronized (this) {
      ...
    }
  }

  static T2 methodB() {
    synchronized (C.class) {
      ...
    }
  }

}
```

Ключова дума volatile

Компилаторите на Java (javac и JIT) оптимизират достъпа до често ползвани променливи. Стойността на променливата се държи (кешира) в регистър на централния процесор, докато се ползва. Междувременно:

- Стойността на оптимизираната променлива не се обновява в паметта.
- Промените в стойността на променливата в паметта, извършени от други нишки, се игнорират.

Ако една променлива бъде декларирана с ключовата дума **volatile**, то тя никога няма да бъде кеширана в регистър. Резултатът от всички атомарни операции с тази променлива ще бъдат налични за всички нишки веднага.

Употребата на **volatile** променливи е по-бърза от синхронизация, защото не изисква нито проверка за притежаване на ключалка, нито заключване, нито блокиране.

Ключова дума volatile

Пример

```
public class StoppableThread
    extends Thread {
  private boolean mustStop = false;

  public void run() {
    while (!mustStop) {
      ...
    }
  }

  public void stop() {
    this.mustStop = true;
  }
}
```

1. Нишка **`StoppableThread`** започва изпълнението си, прочитайки стойността на променливата **`mustStop`** (**`false`**).
2. Стойността на **`mustStop`** се записва в регистър и се чете от там с предположението, че няма да бъде променена от друга нишка.
3. Друга нишка извиква **`stop()`**.
4. Нишката **`StoppableThread`** игнорира новата стойност на **`mustStop`**, използвайки кешираната в регистър стойност.

Ако променливата **`mustStop`** бъде декларирана като **`volatile`**, то нейната стойност няма да бъде кеширана в регистър, и нишката ще спре.

Ключова дума volatile

Атомарни операции

Атомарни операции (сами по себе си) са:

- Присвояване на нова стойност на променливата.
- Прочитане на стойността на променливата.

Атомарни операции **НЕ** са:

- Присвояване на нова стойност на променливата от израз, който зависи от старата стойност.
  - `++, --, +=, -=, *=, /=, %=`

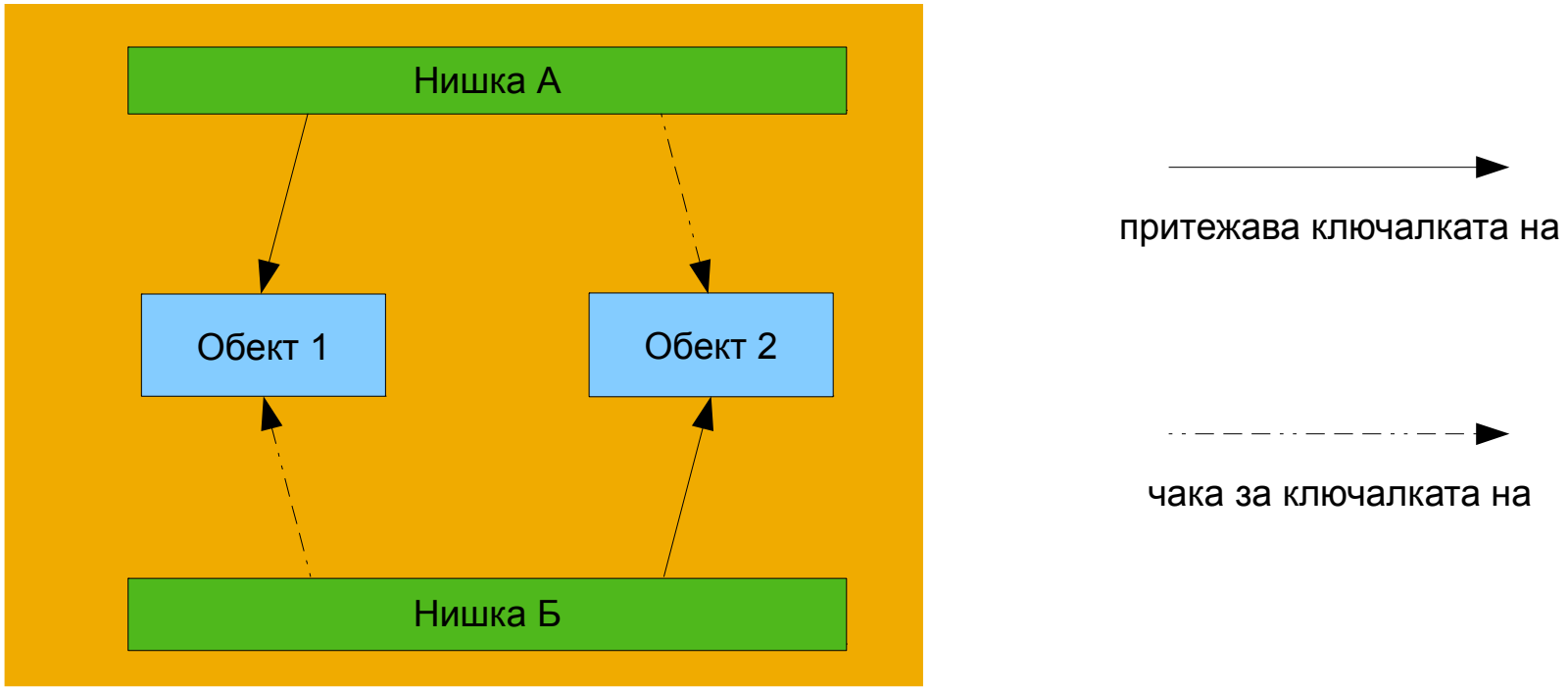
Ако се нуждаете от съставни атомарни операции:

- използвайте **synchronized** блок; или
- използвайте класовете от пакета **java.util.concurrent**

вместо **volatile** променливи.

Мъртва хватка

Мъртва хватка се получава, когато две или повече нишки се блокират една друга, всяка от тях притежаваща ключалка, от която друга нишка има нужда, но чакайки за ключалка, която някоя от другите нишки притежава.



Мъртва хватка
Пример

Пешо и Гошо много обичат да ядат салата. Двамата седят на една маса един срещу друг. Те използват един общ оливерник със шишенце за олио и шишенце за оцет.

Всеки един от двамата държи да си подправи салатата с олио и оцет преди да я изяде. Всеки един от двамата държи да си подправя салатата едновременно с олиото и с оцета. И двамата първо посягат към дясностоящото шишенце.

- „Пешо“ и „Гошо“ всъщност са едновременно изпълняващи се нишки,
- които в безкраен цикъл получават обект („салата“),
- върху който трябва да извършат някакви действия („подправяне“ и „изяждане“ - различни за всяка от нишките).
- Едното от действията („подправянето“) се нуждае от изключителната и едновременна употреба на два споделени ресурса („олио“ и „оцет“).

Мъртва хватка
Пример (Пешо)

```
public void run() {
    while (true) {
        System.out.println("Пешо получава салата.");
        synchronized (this.oil) {
            System.out.println("Пешо взе олиото.");
            synchronized (this.vinegar) {
                System.out.println("Пешо взе оцета.");
                System.out.println("Пешо си подправя салатата.");
            }
        }
        System.out.println("Пешо похапва салата.");
    }
}
```

Мъртва хватка
Пример (Гошо)

```
public void run() {
     while (true) {
          System.out.println("Гошо получава салата.");
          synchronized (this.vinegar) {
               System.out.println("Гошо взе оцета.");
               synchronized (this.oil) {
                    System.out.println("Гошо взе олиото.");
                    System.out.println("Гошо си подправя салатата.");
               }
          }
          System.out.println("Гошо похапва салата.");
     }
}
```

Мъртва хватка
Пример (изпълнение)

Рано или късно се стига до:

Пешо получава салата.
Гошо получава салата.
Гошо взе оцета.
Пешо взе олиото.

Мъртва хватка

- Нишките не могат да бъдат прекратявани отвън.
- Ключалките не могат да бъдат отнемани насилствено.

=> Единственият изход от мъртва хватка е рестартиране на JVM.

Обикновено местата, където се заключват ресурсите, не се намират близо едно до друго в програмния код. В общия случай всяка от нишките заема двете ключалки в различни методи на различни класове, като единият метод е извикан непряко от другия.

=> В една проста мъртва хватка с два ресурса участват до четири непряко свързани едно с друго парчета програмен код. Използвайте изгледа Debug на Eclipse за да откриете веднага две от тях. За другите две - проследете стека на извикванията на всяка от нишките.

Мъртва хватка
Решение

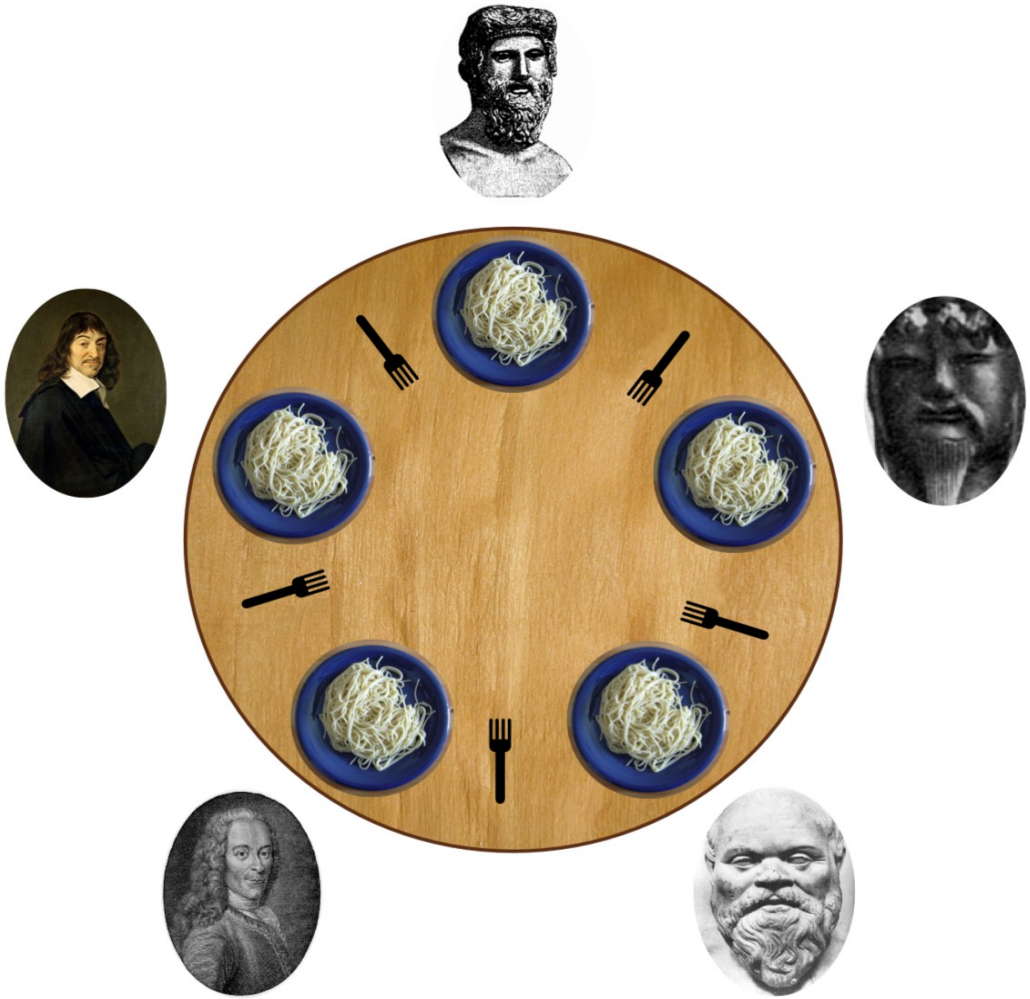
Технологичното решение на проста мъртва хватка с **n** ресурса е ключалките на ресурсите да взимат в един и същи ред от всички нишки.

Освен технологична част, това решение има следните изисквания:

- Група, която да дефинира реда на заемане на на ресурсите (напр. архитекти).
- Документация, която да прави този ред пределно ясен:
  - както в рамките на екипа, разработващ дадения продукт,
  - така и на външни екипи, които го използват в своите продукти и се очаква да заключват неговите ресурси.
- Редовен одит на програмния код, който да проверява дали редът се спазва.

Мъртва хватка

Задачата с обядващите философи



За повече информация, потърсете:
Dining philosophers problem
в английската Уикипедия.

Изчакване и уведомяване

Този механизъм позволява имплементирането на следния сценарий:

- Една или повече нишки чакат някакво събитие да настъпи.
- Друга нишка ги уведомява, когато събитието настъпи.

Всеки обект може да се използва за събитие.

Чакането е блокиране на нишката докато настъпи събитието. То се извършва чрез извикване на един от методите **Object.wait(...)**. Тези методи връщат управлението, когато настъпи събитието (понякога и по-рано – напр. ако нишката е прекъсната).

Уведомяването за настъпване на събитието се извършва чрез извикване на някой от методите **Object.notify()** и **Object.notifyAll()**. notify() събужда една от чакащите нишки (ако има такива въобще), а notifyAll() – всички.

Всички от споменатите методи може да се извикват само от нишка, която е заключила обекта-събитие. Ключалката се освобождава по време на изпълнението на метод wait(…) и се получава пак при неговото приключване.

Изчакване и уведомяване (пример)

Склад за продукти:
`Collection<Product> store`

Производител:
Нишка, която произвежда продукти и ги доставя в склада (добавя инстанции на **`Product`** в **`store`**).

Потребител:
Нишка, която взима продукти от склада (ако са налични) и ги консумира (премахва налични инстанции на **`Product`** от **`store`** и извършва някаква работа с тях).

Докато складът е празен, нишката-потребител трябва да чака. Нишката-производител трябва да уведоми чакащите потребители (ако има такива) при доставка на нови продукти в склада.

Изчакване и уведомяване (пример) - производител

```
public void run() {
  while (…) {
    Product newProduct = this.produce();
    synchronized (store) {
      store.add(newProduct);
      store.notify();
    }
  }
}
```

Изчакване и уведомяване (пример) - потребител

```
public void run() {
  while (…) {
    synchronized (store) {
      for (Product product : store) {
        this.consume(product);
      }
      store.clear();
      try {
        store.wait();
      } catch (InterruptedException e) {
      }
    }
  }
}
```

Фалшиво събуждане

В много редки случаи е възможно **Object.wait(...)** да върне управлението без друга нишка да е извикала **Object.notify*()**. Събуждане без събитието да е настъпило се нарича фалшиво.

След събуждане кодът на нишката трябва да подходи скептично към наличието на събитието. То трябва да се провери или да не се приема за даденост.

Фалшиво събуждане се получава, когато системната библиотека за управление на нишки (намираща се под JVM) загуби за момент поглед върху събитието. След такава ситуация не е сигурно дали през този момент събитието не е настъпило. Ако все пак събитието се е случило и чакащата нишка не бъде събудена, то най-вероятно ще се получи съвсем истинска мъртва хватка.

Фалшиво събуждане
Примери

```
store.wait();
Product product = store.iterator().next();
// В случай на фалшиво събуждане store ще е празна и горният ред ще изхвърли NoSuchElementException.
-----------------------------------------------------------
while (store.isEmtpy()) {
    store.wait();
}
// Последното събуждане със сигурност не е било фалшиво.
-----------------------------------------------------------
store.wait();
for (Product product : store) {
}
// Ако последното събуждане е било фалшиво, то цикълът просто няма да има нито една итерация.
```

Цена и общи съвети

Колкото по-често и за колкото по-дълго нишките се блокират, толкова по-неефективно се използват изчислителните ресурси.

Влизането в синхронизиран блок е относително бавна операция, дори и когато обектът е отключен.

Целта на нишките е да се оползотворяват изчислителните ресурси възможно най-добре, но това трябва да се случва чрез използване на възможно най-малка част от тях за служебни дейности (т.е. синхронизация).

Общи съвети:

- Синхронизирайте според най-лошия възможен сценарий.

Но ако имате възможност (подредени по приоритет):

- Синхронизирайте за възможно най-кратко.
  - Избягвайте вход/изход в синхронизиран блок.
- Синхронизирайте възможно най-рядко.

Допълнителни материали

Java tutorial за конкурентно програмиране:

https://docs.oracle.com/javase/tutorial/essential/concurrency/

… или просто потърсете следните ключови думи:

Java tutorial threads

Благодаря за вниманието!

Домашно (по желание):

1. Подобрете примерът за склад така, че да са приложени генералните съвети възможно най-добре.
2. Подобрете примерът за склад така, че да са възможни повече от една нишка-потребител и повече от една нишка-производител.

Работещ (но неоптимизиран) пример може да намерите тук:
http://java.voidland.org/Store-0.1.src.zip

Пращайте предложенията си на:
vladimir.panov@sap.com

Речник за някои неочевидни преводи

Български	Английски
прекратяване/прекратен	termination/terminated
изпълним	runnable
работещ/изпълняващ се	running
фалшиво събуждане	spurious wake-up
мъртва хватка	deadlock
рекурсивна ключалка	reentrant/recursive lock