Java нитки и паралелизам

Оперативни системи

Аудиториска вежба 5

Содржина

Конкурентност

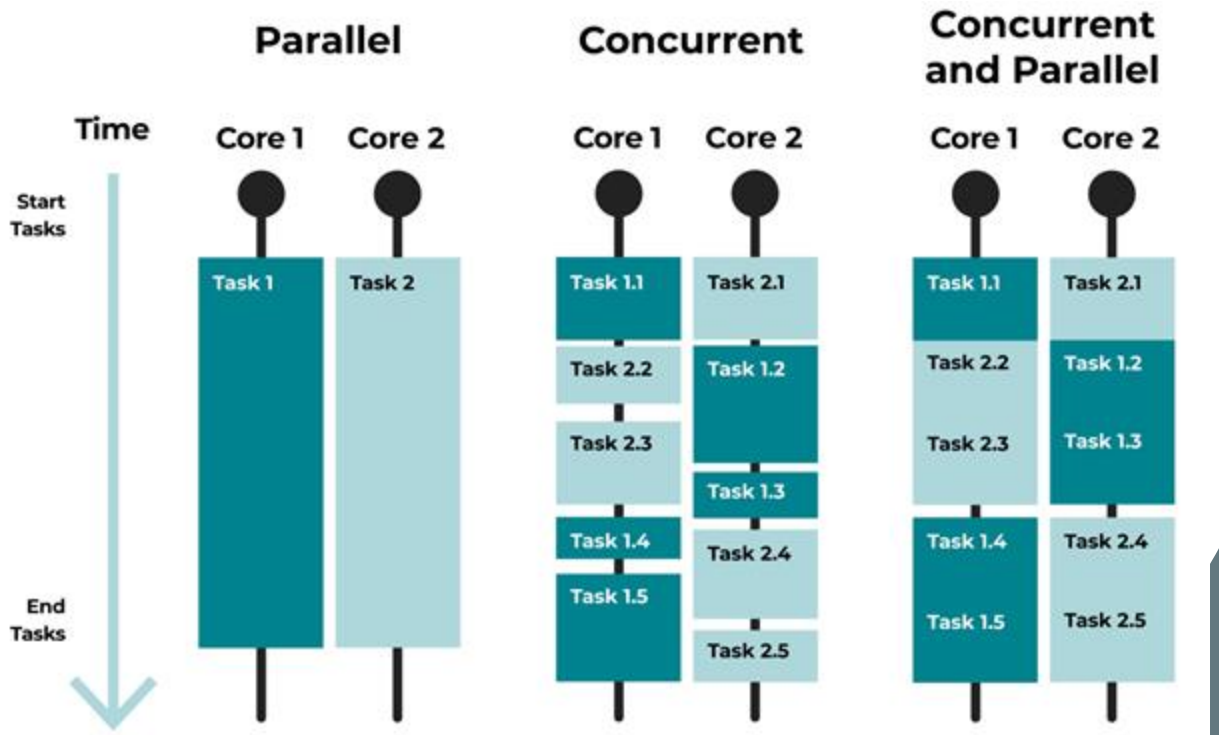
Конкурентност значи истовремено извршување на повеќе задачи

Контекст:

- Повеќе апликации;
- Повеќе процеси во една апликација;
- Повеќе нитки во еден процес;

Конкурентното извршување значи споделување на ресурси:

- Процесорско време
- Меморија
- Влезно/излезни уреди

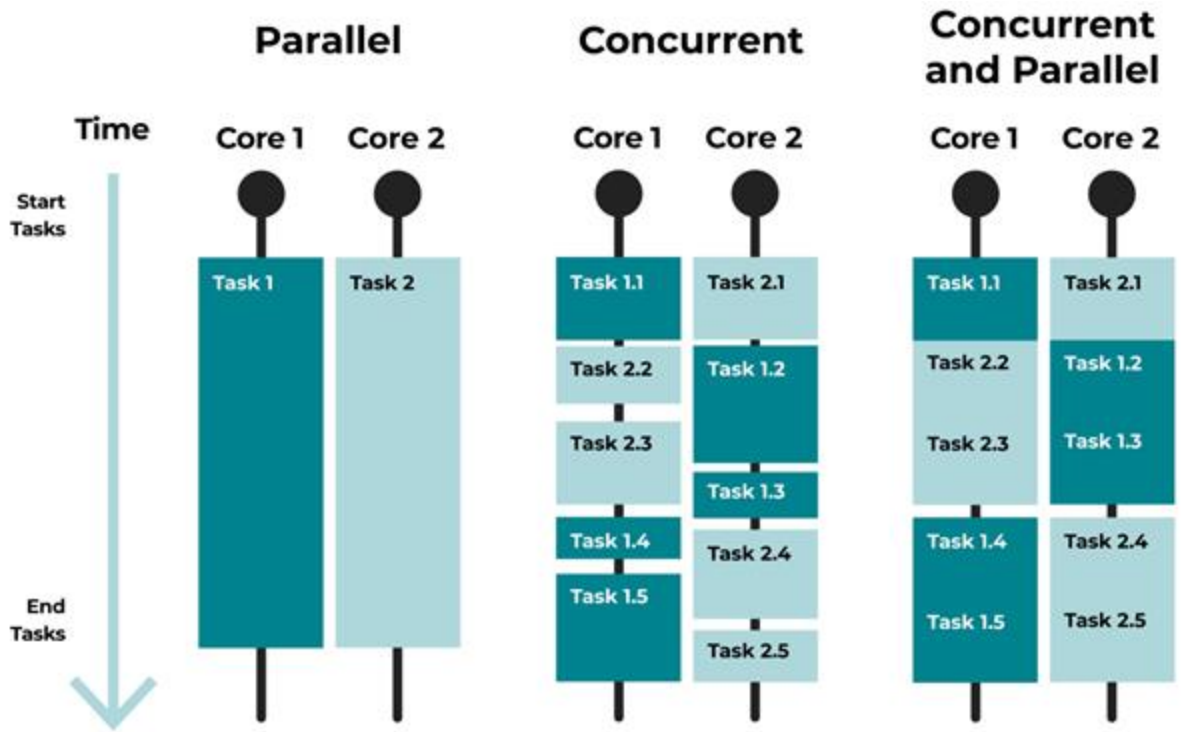


Конкурентност

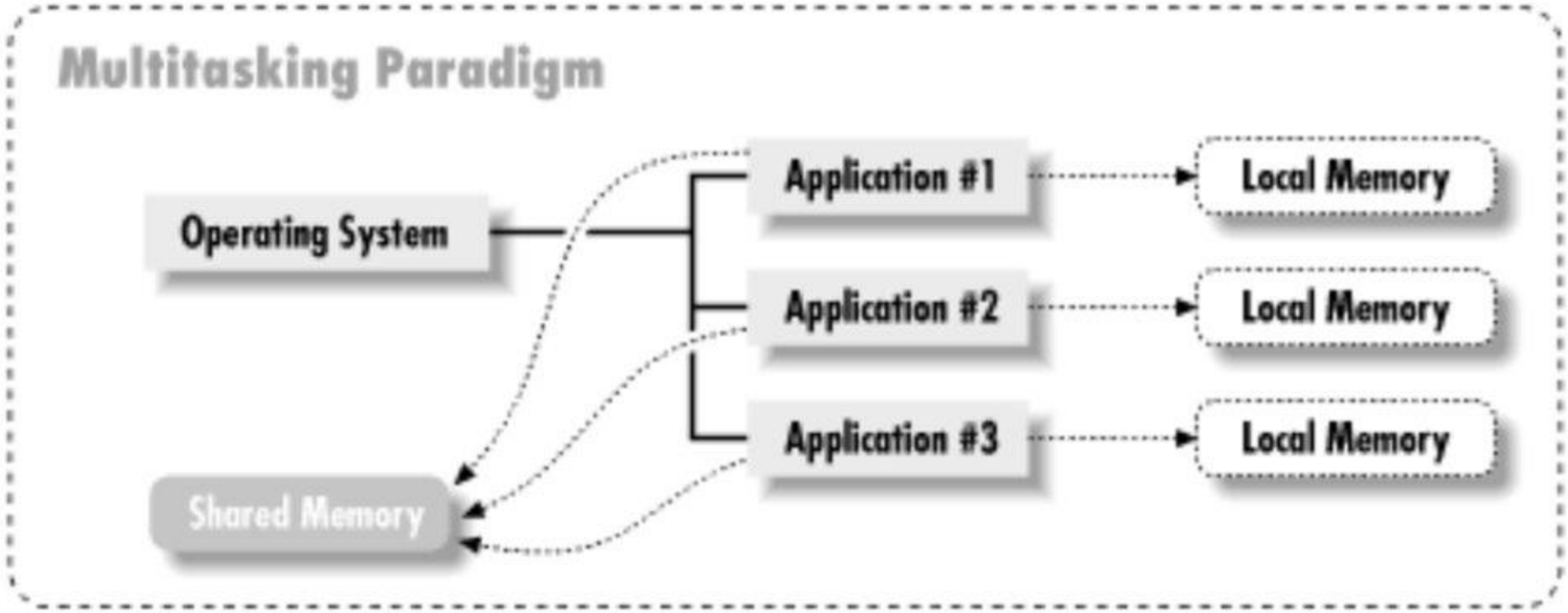
Конкурентност значи истовремено извршување на повеќе задачи

Проблеми од споделени ресурси:

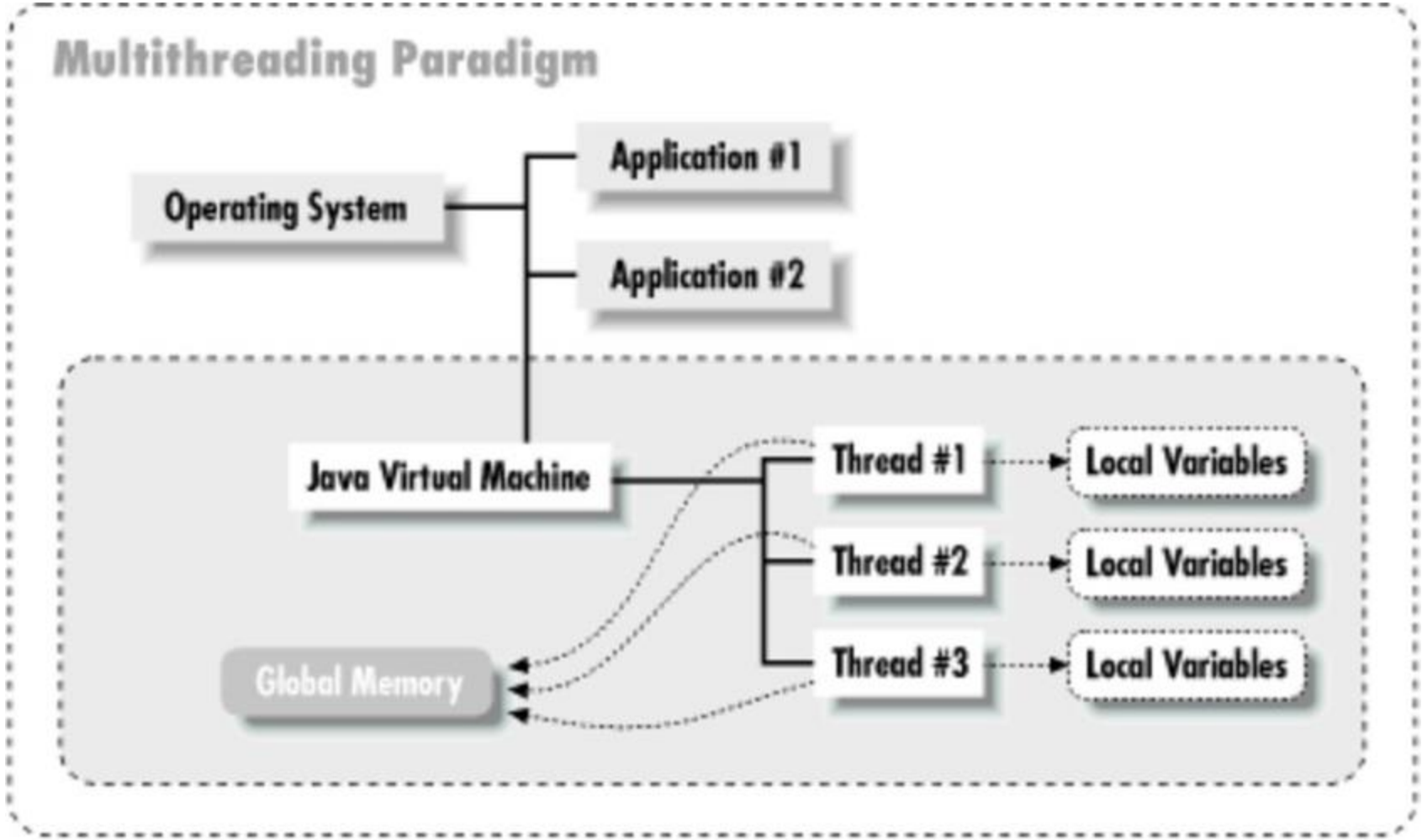
- Взаемно исклучување
- Потреба од синхронизација;
- Опасност од блокада



Конкурентност: парадигма на мултитаскинг



Конкурентност: парадигма на мултитрединг



Главни проблеми

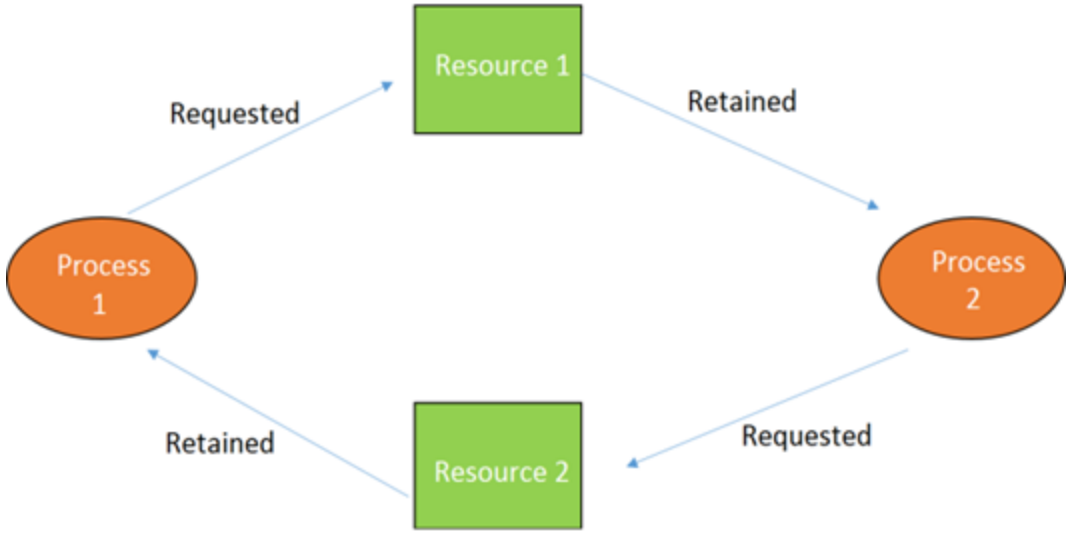
- Взаемно исклучување
- Ексклузивен пристап до неделливи, но споделени ресурси.
  - Пример: принтер;
- Заклучувања на ресурсите
- Условна синхронизација
- Обично се однесува на „потрошливи“ ресурси.
  - Пример: примерот на производител-потрошувач;
- Може да подразбира долго (и повторено) чекање.
- Потребни се структури за управување со задачите кои се во состојба на чекање.

Deadlock

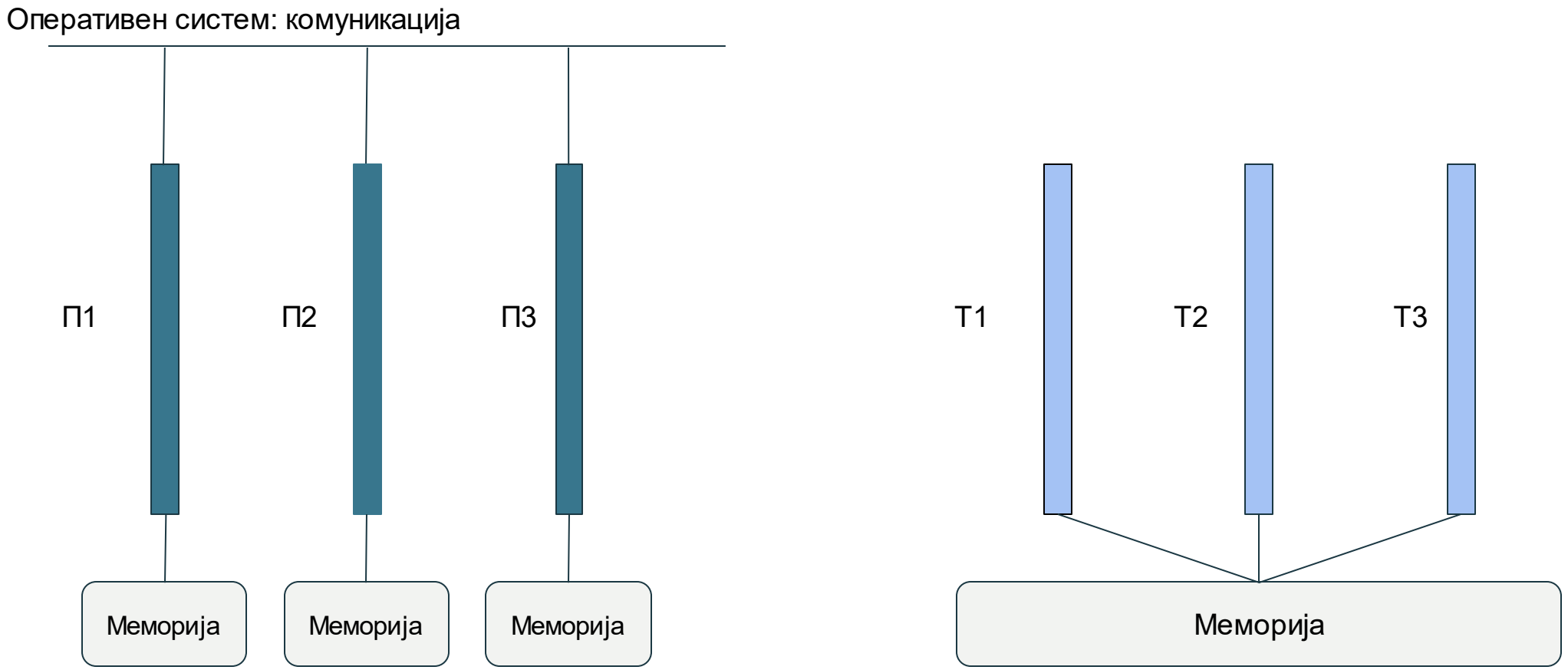
- Опасност од блокада доколку взаемно се чека на заклучени споделени ресурси
- Множество на процеси е во deadlock доколку секој процес во мночеството чека на настан, кој може да го предизвика само друг процес во истото множество.

- Затоа што сите процеси чекаат, никој од нив никогаш нема да го предизвика настанот кој би можел да го пробуди некој друг член од множеството, и сите процеси продолжуваат да чекаат засекогаш.

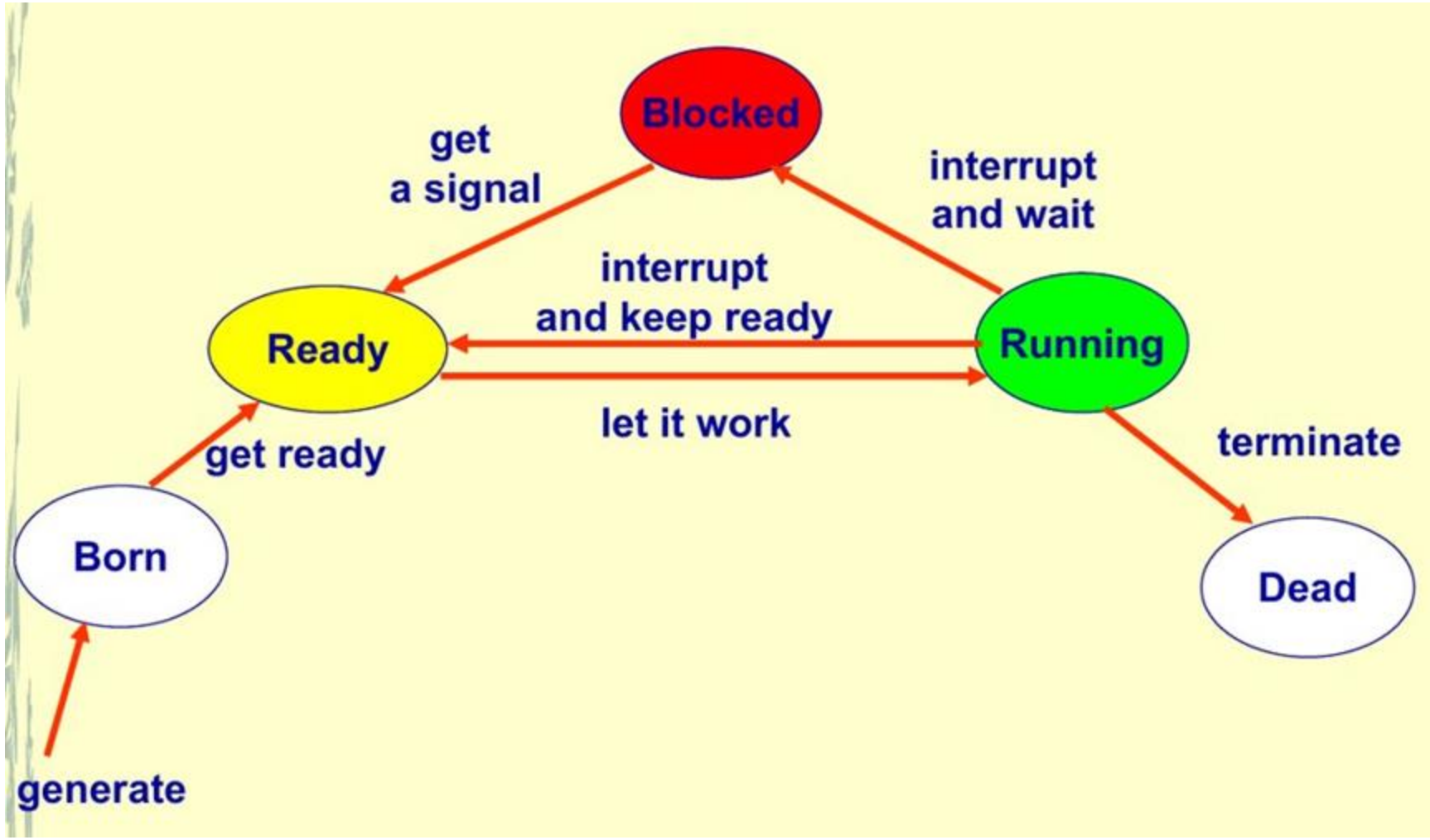
- Практичен пример: Филозофи кои вечераат.



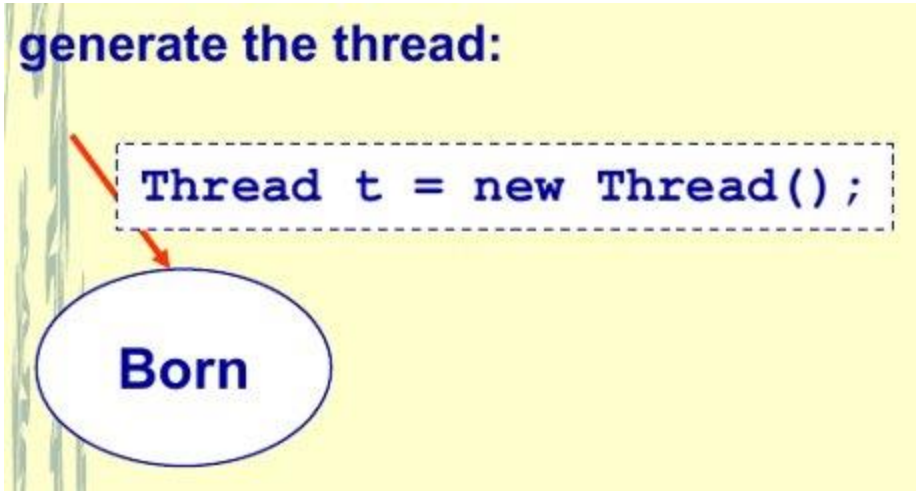
Процеси наспроти нитки

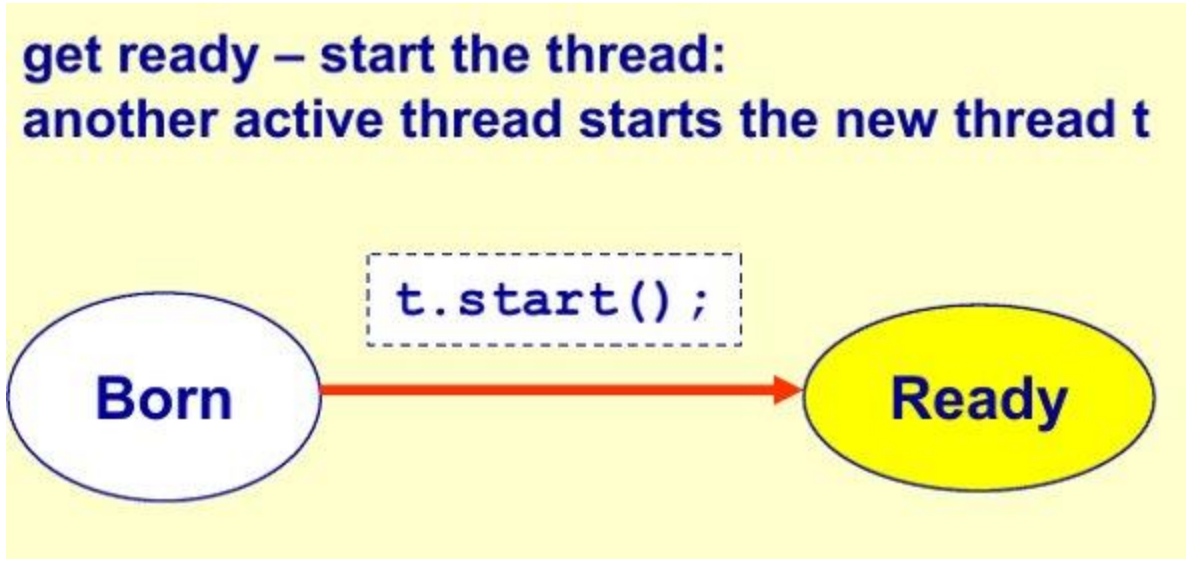


Животен циклус на нитки

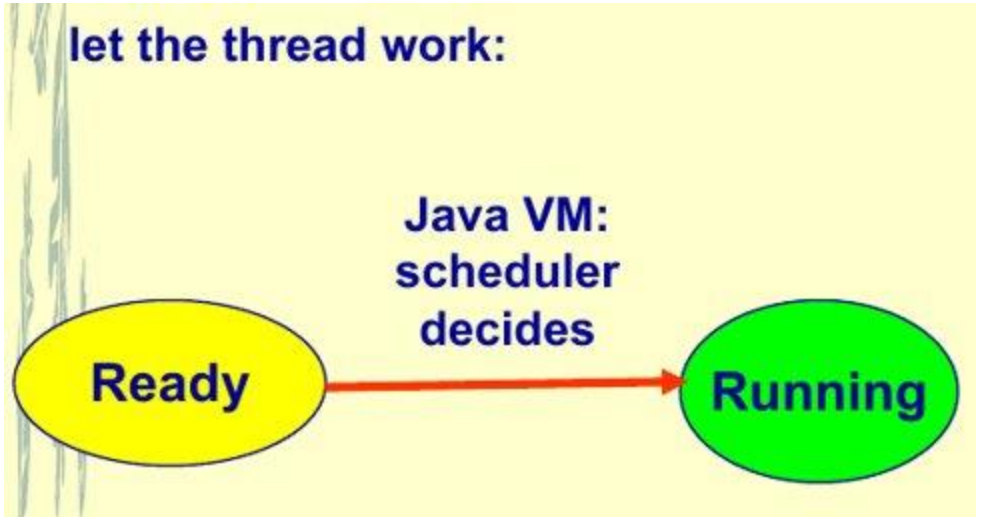


Животен циклус на нитки во детали





Животен циклус на нитки во детали



interrupt the thread
and keep it ready:

- Time slice ends
- Process with higher priority appears
- Process releases the processor: t.yield()

Животен циклус на нитки во детали

Interrupt and wait:

- Process goes to sleep: t.sleep()
- Process waits for the end of another thread u: u.join()
- Wait for the unlock of another object o: o.wait()

Get a signal and get ready:

- Time interval of sleep ends
- Another thread wakens t: t.interrupt()
- A thread whose end was waited for by the blocked thread „dies“
- Object o becomes unlocked

Моделот на нитки во Java

- Повеќе нитки во рамките на еден процес (JVM).
- Вградена синхронизација на исклучување во самиот јазик (секој објект има lock).
- Условна синхронизација базирана на монитори.
- JVMs може да искористат повеќе процесори.
- Постои стандардна нитка во секоја Java програма која ја извршува ‘main’ функцијата на програмата.
- Нови нитки може да се креираат и да се стартуваат.
- Нитките се објекти во Java.
- Возможно е да се направи хиерархија на класи од нитки.

Нитки во Java: Класата Thread

- Java ја содржи класата java.lang.Thread чиишто метод run() е наменет да ја содржи главната логика на секоја нитка.
- Класа T може да се изведе од Thread и да ја препокрие имплементацијата на run() методот. Инстанците на класата T се нитки и може да се стартуваат со повикување на методот start() од класата Thread.
- Методот start() го повикува методот run() – **не треба вие да го повикувате методот run().**
- Нитката ќе терминира кога методот run() method ќе заврши.

Нитки во Java: Класата Thread

```
public class Main1 {
 public static void main(String[] args) throws
InterruptedException {
 T obj = new T();
   obj.start();
   //obj runs in parallel
   //with the main thread
 }
}

class T extends Thread {
 public void run() {
 // thread's main logic
 }
}
```

Нитки во Java: Интерфејсот Runnable

- Поради ограничувањата на наследување само од една класа во Java, нов механизам е потребен кога нитката треба да наследува од друга класа.

- Интерфејсот Runnable го содржи методот run() – класата Thread всушност го имплементира интерфејсот Runnable.

- Класа T2 може да наследи од друга класа и да го имплементира интерфејсот Runnable.

- Инстанца од класата T2 ја даваме како аргумент во конструкторот на класата Thread, за да креираме Thread објект.

Нитки во Java: Интерфејсот Runnable

```
public class Main2 {
    public static void main(String[] args) {
      Runnable obj = new T2();
      Thread tobj = new Thread(obj);
      tobj.start();
      // tobj runs in parallel
      // with the main thread
    }
}

class Base {}

class T2 extends Base implements Runnable {
  public void run() {
    // thread's main logic
  }
}
```

Животен циклус на нитки во Java

Креирање: нитка која се извршува (t0) креира нова нитка:

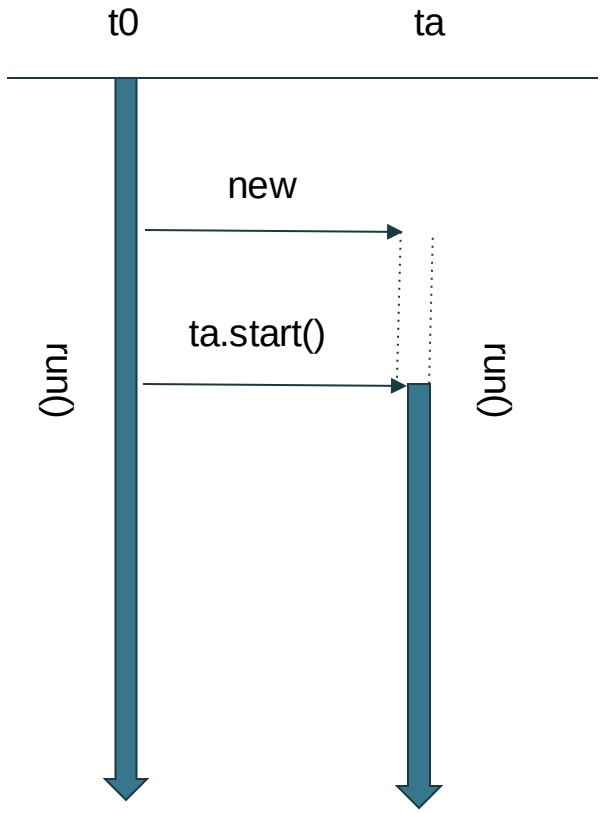
- ta = new ThreadA1();

Стартување: нитката која се извршува ја стартува новата нитка:

- ta.start();

Извршување: JVM распоредувачот го повикува методот run().

Терминирање: run() методот ја завршува работата.



Пример за извршување на нитки

```
public class ThreadBasicTest {
   public static void main(String[] args) {
       Thread ta = new ThreadA();
       Thread tb = new ThreadB();
       ta.start();
       tb.start();
       System.out.println("Main done");
   }
}
class ThreadA extends Thread {
   public void run() {
       for (int i = 1; i <= 20; i ++) {
           System.out.println("A: " + i);
       }
       System.out.println("A done");
   }
}
class ThreadB extends Thread {
   public void run() {
       for (int i = -1; i >= -20; i --) {
           System.out.println("\t\tB: " + i);
       }
       System.out.println("B done");
   }
}
```

```
Main done
A: 1
A: 2
A: 3
A: 4
A: 5
A: 6
A: 7
A: 8
        B: -1
        B: -2
        …
        B: -14
        B: -15
        B: -16
        B: -17
        B: -18
        B: -19
        B: -20
        B done
A: 9
A: 10
…
A: 17
A: 18
A: 19
A: 20
A done
```

```
Main done
        B: -1
A: 1
A: 2
        B: -2
        B: -3
        B: -4
A: 3
        B: -5
… (наизменично)
        B: -14
A: 13
        B: -15
A: 14
        B: -16
A: 15
        B: -17
A: 16
        B: -18
A: 17
        B: -19
A: 18
        B: -20
A: 19
        B done
A: 20
A done
```

Прекинување на извршувањето на нитките

Нитка којашто е жива останува жива се додека:

- run() не заврши нормално;
- run() не заврши со грешка;
- destroy() не биде повикан на нишката;
- терминира програмата;
- Кога методот run() на нитката терминира, нитката ги ослободува сите locks.
- destroy() е драстичен метод, не ги ослободува locks и некои JVMs не го имплементираат.
- Се препорачува прекинување (interrupt) на нитките наместо уништување.

Прекинување на извршувањето на нитките

- Една нитка може да го прекине извршувањето на друга нитка со повикување на interrupt() методот. Прекинатата нитка го користи методот interrupted() to за да ја тестира и расчисти состојба по прекин.
- Пример:
  - Нитка 1 го има кодот:

```
thread2.interrupt();
```

  - Нитка 2 го има кодот:

```
while (!interrupted()) {
   // normal execution
}
```

  - Нитка 2 може да се прекине.

Паралелно извршување на нитки

```
public class UseJoin {
   public static void main(String[] args) {
        Count c = new Count();
        c.start();
        try {
            c.join();
            System.out.println("Result = " + c.getResult());
        } catch(InterruptedException e) {
        }
   }
}

class Count extends Thread {
   private long result;
   public void run() {
        result = count();
   }
   public long getResult() {
        return result;
   }
   public long count() {
        long r = 0;
        for (r = 0; r < 100000; r++);
        return r;
   }
}
```

- Во програма со повеќе нитки, никогаш не знаеме која нитка ќе заврши прва (или кога ќе заврши).
- Можеме да ги чекаме нитките да завршат со користење на методот join(), од класата Thread.

Ризици при користење на нитки

Безбедносни ризици (Коректност)

- Во отсуство на добра синхронизација, распоредот на операциите во повеќе нитки е непредвидлив, некогаш и изненадувачки.
- Чест безбедносен ризик е услов на трка (race condition).

Ризици на активност (Deadlock)

- Ваквите ризици настануваат кога некоја активност во нитките ќе влезе во таква состојба од која е трајно невозможно да продолжи со извршување.
  - Бесконечна јамка
- Ако нитката А чека на некој ресурс кој го има нитката Б, и нитката Б никогаш не го ослободи, тогаш нитката А ќе чека засекогаш.

Ризици при користење на нитки

Ризици поврзани со перформанси

- Лошо време на услуга, респонзивност, responsiveness, пропусна моќ, искористеност на ресурси или скалабилност
- Поголем overhead при извршување
- Повеќе промени на контекст
- Зачувување и враќање на контекст за извршување на нитка, загуба на локалноста, процесорско време потрошено на распоредување на нитки наместо нивно извршување
- Споделени податоци меѓу нитките
- Синхронизациските механизми може да ги влошат оптимизациите на кодот од компајлерот, да го инвалидираат мемориският кеш, и да направат синхронизациски метеж на мемориската магистрала

Споделени ресурси

Ресурси кои можат да бидат пристапувани конкурентно од повеќе нитки или процеси

- Променливи
- Мемориски локации (покажувачи)
- Влезно/излезни уреди и операции

Споделени ресурси

Паралелно пристапување до споделени ресурси може да доведе до неконзистентност кај таквите ресурси

- Неконзистентни вредности
- Недетерминистичко однесување
- Редоследот и начинот на пристап до споделени ресурси потребно е да се синхронизира

Атомичност

За една операција велиме дека е атомична ако: или сите операции од коишто е составена се извршуваат успешно, или ниту една операција не се извршува.

- Атомичноста ни овозможува подреденост.
- Конкурентното извршување може да се нарече дека е подредено доколу извршувањето на повеќе нитки е гарантирано дека одговара на некакво сериско извршување на тие задачи.
- Ни овозможува предвидливи резултати од операциите.

Атомичност

Пример: var ++

Операции:

- Прочитај ја вредноста на променливата var од нејзината мемориска локација во процесорските регистри.
- Зголеми ја вредноста на променливата.
- Запиши ја инкрементираната вредност во мемориската локација на променливата.

Објансување:

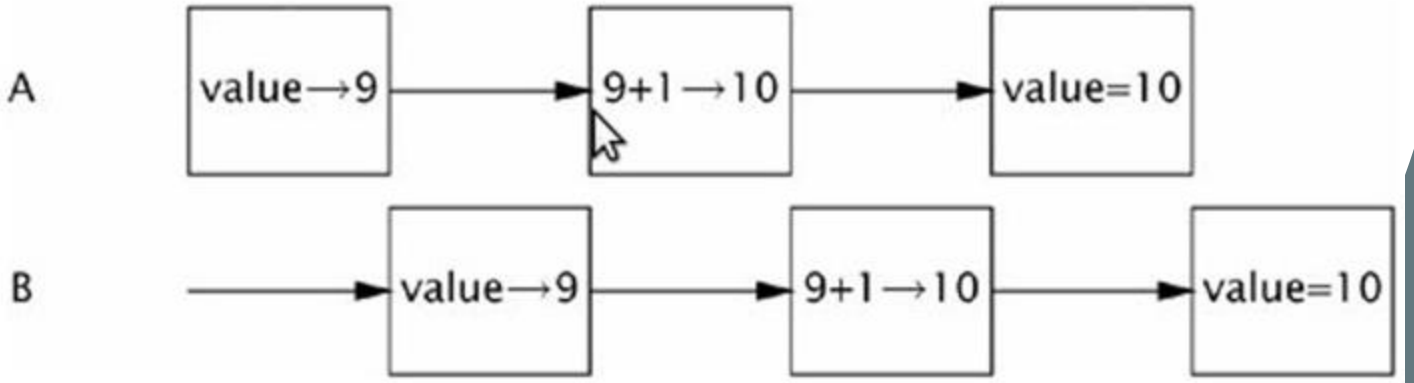
- Повеќе од еден процесорски циклус.
- Процесот / нитката може да биде прекинат од распоредувачот по било која од горенаведените операции.
- Затоа, операторот ++ не е атомичен, има непредвидлив резултат во околина каде што се извршуваат повеќе нитки.

Услови на трка (race conditions)

- Услови на трка настануваат кога повеќе нитки изведуваат серија на акции врз споделени ресурси, и повеќе различни исходи може да се добијат како резултат на редоследот на извршување на активностите од секоја нитка.

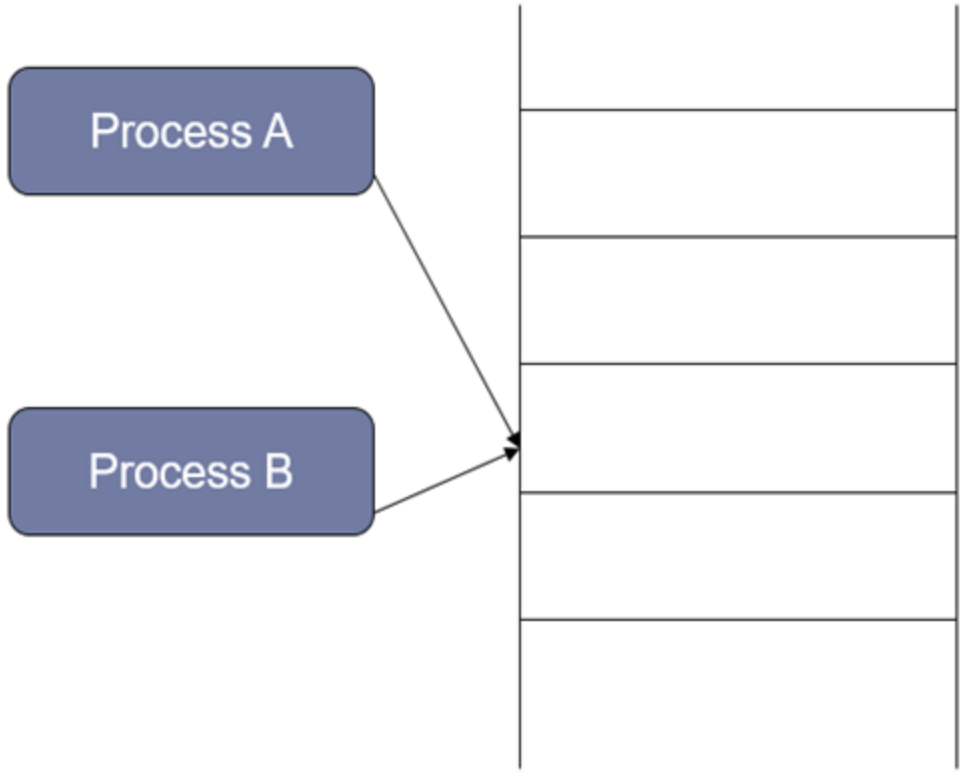
```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    public int getNext() {
        return value++;
    }
}
```



Услови на трка (race conditions)

Printer Spooler



Критични региони и взаемно исклучување

- Критичен регион е дел од кодот на процесот/нитката којтшто пристапува споделени променливи, споделени датотенки или друг тип на споделени мемориски објекти.

- Кога повеќе процеси/нитки се извршуваат во системот, мора да го регулираме пристапот до критичниот регион и да дозволиме само еден процес/нитка да пристапи до регионот во даден момент.

- Ова се нарекува взаемно исклучување.

Взаемно исклучување: Решение за услови на трка

Mutex

- Овозможува атомично извршување на операции
- Во Java, java.util.concurrent.locks.Lock имплементацијата делува како mutex со користење на следните методи:
  - lock()
- Го добива lock-от.
- Ако lock-от не е достапен, моменталната нитка ќе биде блокирана, не се распоредува и чека се додека не го добие lock-от.
  - unlock()
- Нитката го ослободува lock-от.

Взаемно исклучување: Монитор

- Користењето на мутекс често резултира со грешки, бидејќи е резултат на дисциплината и знаењето на програмерите.

Монитор

- Секој објект во Java содржи монитор
- Обезбедува взаемно исклучување при пристап на критични региони од кодот
- Synchronized
- Означуваме метод или блок од кодот со synchronized
- Само една нитка во даден момент има дозвола да извршува критична секција од кодот синхронизиран со монитор.

Пример за монитор

```
@ThreadSafe
public class SafeSequence {
   private int value;

   public synchronized int getNext() {
       return value++;
   }
}

@ThreadSafe
class SafeSequence {
   private int value;

   public int getNext() {
       synchronized(this) {
           return value++;
       }
   }
}
```

Сценарија за синхронизација

```
SafeSequence a = new SafeSequence();
SafeSequence b = new SafeSequence();
```

- **Случај 1:**
  - Нитката 1 ја повикува методата a.getNext() и Нитката 2 ја повикува методата b.getNext()
- Двете нитки користат различни инстанци кои се синхронизирани со различни монитори.
- Нема критичен регион во овој случај, и изршувањето на нитките ќе биде паралелно.

- **Случај 2:**
  - Нитката 1 и Нитката 2 ја повикуваат методата a.getNext()
- Двете нитки чекаат на методот којшто е синхронизиран од истиот монитор (на инстанцата a), па во овој случај методите ќе бидат повикани еден по друг.
- Ќе има взаемно исклучување во критичниот регион.
- Но, редоследот на извршување на нитките не може да се предвиди.

Пример: Синхронизација на статички метод

```
@ThreadSafe
class SafeSequence {
   private static int value;

   public static synchronized int getNext() {
       return value++;
   }
}
```

```
@ThreadSafe
class SafeSequence {
   private static int value;

   public static int getNext() {
       synchronized(SafeSequence.class) {
           return value++;
       }
   }
}
```

Пример: Синхронизација на статички методи

```
SafeSequence a = new SafeSequence();
SafeSequence b = new SafeSequence();
```

- **Случај 1:** Нитката 1 ја повикува методата a.getNext() и Нитката 2 ја повикува методата b.getNext()
  - Методот getNext() е статички, што значи дека е ист за сите инстанци на класата, односно сите инстанци го делегираат повикот до:

    ```
    SafeSequence.getNext();
    ```
- Двете нитки чеккат на метод којшто е синхронизиран од истиот монитор (SafeSequence.class монитор), па методите ќе бидат повикани еден по друг во различните нитки
- Ќе има взаемно исклучување во критичниот регион
- Но, редоследот на извршување не може да се предвиди

Синхронизација

- Зависности на операциите кои се извршуваат во различни нитки
- Една нитка треба да го чека резултатот од друга нитка

Семафори

- Класата во Java java.util.concurrent.Semaphore
- Концептуално, семафорот одржува множество на дозволи
- Секој повик на Semaphore.acquire() блокира доколку е неопходно се додека нема достапни дозволи, и откако ќе стане достапна дозволата нитката ја зема
- Секој повик на Semaphore.release() додава дозвола во множеството, потенцијално ослободува блокирана нитка која чека дозвола со acquire().
- Се користи за да се ограничи бројот на нитки кои можат да пристапат одреден ресурс (физички или логички)
- Од тие причини, мутексот некогаш се нарекува бинарен семафор.

Пример: Семафор

Иницијализација на семафор

Semaphore empty = new Semaphore(1);

Користење:

- Thread 1 – Producer
  ```
  empty.acquire();
  putItems(buffer);
  ```

- Thread 2 – Consumer
  ```
  if(noMoreItems()) {
        empty.release();
  }
  Item = getItem(buffer);
  ```

Deadlock- Блокада

Се случува кога:

- Има взаемно исклучување
- Ресурсите може да се доделат на најмногу една нитка
- Нитката оди по принципот „задржи и чекај“
- Нитките држат одредени ресурси и бараат други истовеременo
- Има циркуларно чекање
- Постои циклус во кој секоја нитка чека на ресурс којшто е доделен на друга нитка

ПРАШАЊА