Synchronization

Concurrent and parallel programming

Programowanie współbieżne i równoległe

Academic year: 2018/19, Lecture 3

Paweł Lula, Cracow University of Economics, Poland
pawel.lula@uek.krakow.pl

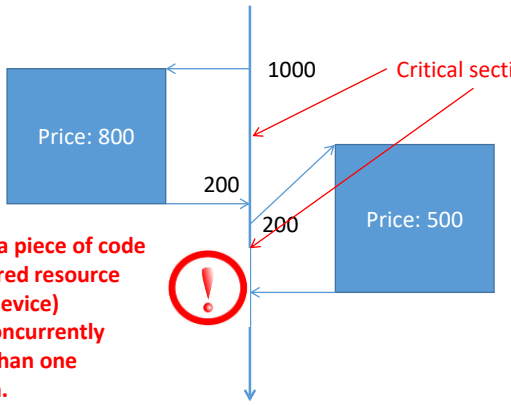
Synchronization

- Synchronization:
  - coordination of events in time;
  - harmonization of events, which allows to perform several operations properly at the same time
- Synchronization in concurrent programming = the mechanism which ensures proper results of the program regardless of the order in which actions from different tasks are executed.

Example – with synchronization

Common bank account for a couple

BALANCE: 1000

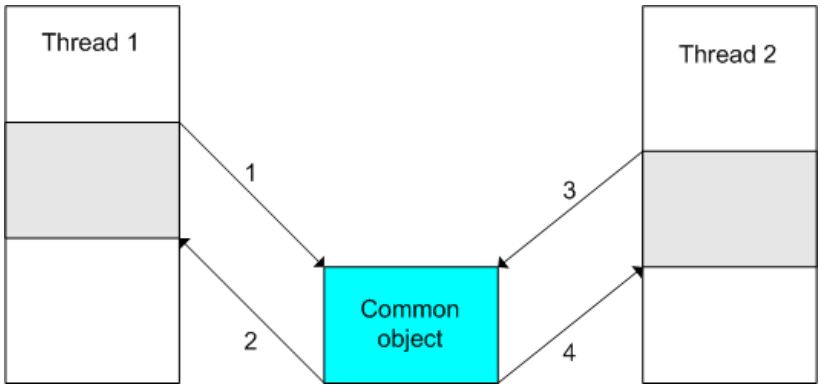


A critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

Critical section

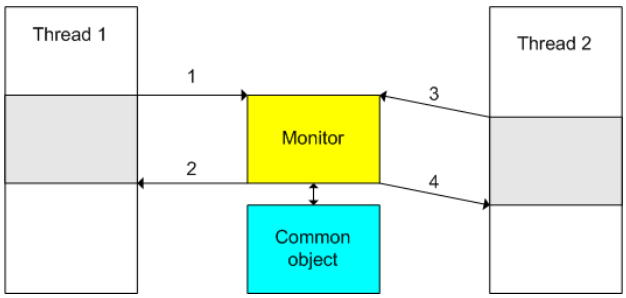
- **Critical section of a thread** – a part of the code from which common resources (variables, devices) are accessed and used.
- Only **one thread** can perform its critical section related to the same resource.

Schema of critical section



Monitor

- Monitor – a tool used for implementation of rules of access to common objects by threads.
- During the execution of critical section the thread occupies the monitor of the object related to this part of a code.



Monitor in Java language

synchronized (*object*)

//entering the monitor

{

...

critical section related to the *object*

...

}

//releasing the monitor

7

Example

```
class Thread01 implements Runnable {
   public void run()          {
         synchronized (System.out) {
               for (int i = 1; i <= 10; i++) {
                     try {
                           Thread.sleep(100);
                     }
                     catch (InterruptedException e) {
                     }
                     System.out.print("A");
               }
         }
   }
}
```

8

.....

```
class Thread02 implements Runnable {
   public void run() {
            synchronized (System.out) {
                     for (int i = 1; i <= 10; i++) {
                              try
                              {
                                       Thread.sleep(100);
                              }
                              catch(InterruptedException e) {
                              }
                              System.out.print("B");
                     }
            }
   }
}
```

9

.....

```
public class MyExample {
            public static void main(String [] args) {
                     Thread g = new Thread(new Thread01());
                     Thread p = new Thread(new Thread02());
                     g.start();
                     p.start();
                     System.out.println(“FINISHED!");
            }
}
```

Program’s output:

FINISHED!

AAAAAAAAAABBBBBBBBBB

it is also possible to obtain:

FINISHED!

BBBBBBBBBBAAAAAAAAAA

10

The description of *wait method*

```
synchronized (object) {
        ...
        wait()
        ...
}
```

- The *wait()* causes current thread to wait until another thread invokes the *notify()* method or the *notifyAll()* method for this object.

11

Metoda *notify*

```
synchronized (obiekt) {
        ...
        notify()
        ...
}
```

- The *notify()* wakes up a single thread that is waiting on this object's monitor.
- If many threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
- A thread waits on an object's monitor by calling the *wait()* method.

12

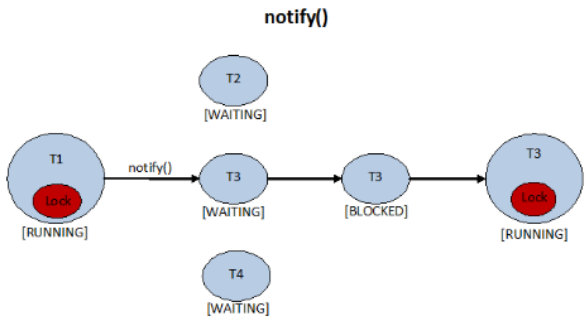
Metoda *notifyAll*

```
synchronized (obiekt)
{
        ...
        notifyAll()
        ...
}
```

- The java.lang.Object.notifyAll() wakes up all threads that are waiting on this object's monitor.
- The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.

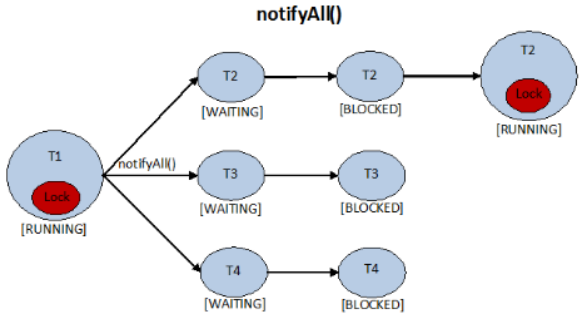
13

Difference Between notify And notifyAll



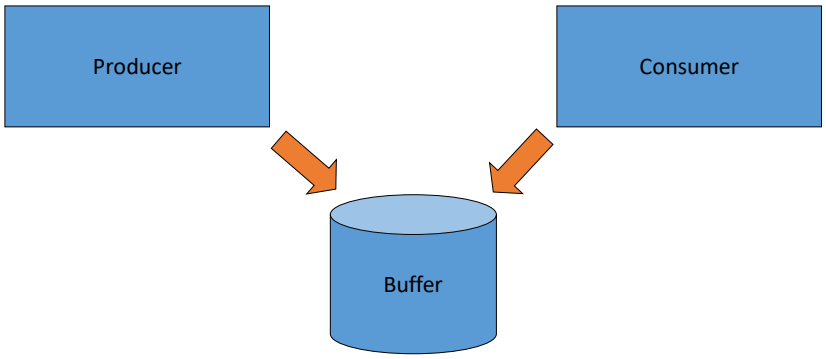
Only one thread will be notified randomly and gets the lock of the object.

Difference Between notify And notifyAll



All the threads which are waiting for this object lock are notified. All the notified threads will get the object lock on a priority basis.

Producer-Consumer problem



Producer-Consumer problem, version I

```
class Pojemnik
{
   private int liczba;
   synchronized public int pobierz()      {
            System.out.println("Z pojemnika pobierana jest wartosc: " +
                                                liczba);
            return liczba;
   }
   synchronized public void wstaw(int liczba) {
            System.out.println("Do pojemnika wstawiana jest wartosc: " +
                                                liczba);
            this.liczba = liczba;
   }
}
```

17

Producer-Consumer problem, version I

```
class Producent implements Runnable {
   Pojemnik p;
   Producent(Pojemnik p) {
            this.p = p;
   }
   public void run() {
            for (int i = 1; i <= 10; i++) {
                     try {
                              Thread.sleep((int) (100 * Math.random()));
                     }
                     catch (InterruptedException e) { }
                     p.wstaw(i);
            }
   }
}
```

18

Producer-Consumer problem, version I

```
class Konsument implements Runnable {
   Pojemnik p;
   Konsument(Pojemnik p) {
         this.p = p;
   }
   public void run() {
         for (int i = 1; i <= 10; i++) {
               try {
                     Thread.sleep((int) (100 * Math.random()));
               }
               catch (InterruptedException e){ }
               p.pobierz();
         }
   }
}
```

19

Producer-Consumer problem, version I

```
public class ProducentKonsument1 {
   public static void main(String[] args) {
         Pojemnik poj = new Pojemnik();
         Producent prod = new Producent(poj);
         Konsument kons = new Konsument(poj);
         Thread watek1 = new Thread(prod);
         Thread watek2 = new Thread(kons);
         watek1.start();
         watek2.start();
   }
}
```

20

Producer-Consumer problem, version I

Efekt działania programu:

```
Do pojemnika wstawiana jest wartosc: 1
Z pojemnika pobierana jest wartosc: 1
Do pojemnika wstawiana jest wartosc: 2
Do pojemnika wstawiana jest wartosc: 3
Do pojemnika wstawiana jest wartosc: 4
Z pojemnika pobierana jest wartosc: 4
Do pojemnika wstawiana jest wartosc: 5
Do pojemnika wstawiana jest wartosc: 6
Do pojemnika wstawiana jest wartosc: 7
Do pojemnika wstawiana jest wartosc: 8
Z pojemnika pobierana jest wartosc: 8
Do pojemnika wstawiana jest wartosc: 9
Do pojemnika wstawiana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
```


Producer-Consumer problem, version II

```
class Pojemnik {
    private int liczba;
    private boolean czyJestWartoscWPojemniku = false;
    public int pobierz() {
            while (!czyJestWartoscWPojemniku) ; //pusta petla
            System.out.println("Z pojemnika pobierana jest wartosc: " + liczba);
            czyJestWartoscWPojemniku = false;
            return liczba;
    }
    public void wstaw(int liczba) {
            while (czyJestWartoscWPojemniku) ; //pusta petla
            System.out.println("Do pojemnika wstawiana jest wartosc: " + liczba);
            this.liczba = liczba;
            czyJestWartoscWPojemniku = true;
    }
}
```

Producer-Consumer problem, version II

Efekt działania programu:

```
Do pojemnika wstawiana jest wartosc: 1
Z pojemnika pobierana jest wartosc: 1
Do pojemnika wstawiana jest wartosc: 2
Z pojemnika pobierana jest wartosc: 2
Do pojemnika wstawiana jest wartosc: 3
Z pojemnika pobierana jest wartosc: 3
Do pojemnika wstawiana jest wartosc: 4
Z pojemnika pobierana jest wartosc: 4
Do pojemnika wstawiana jest wartosc: 5
Z pojemnika pobierana jest wartosc: 5
Do pojemnika wstawiana jest wartosc: 6
Z pojemnika pobierana jest wartosc: 6
Do pojemnika wstawiana jest wartosc: 7
Z pojemnika pobierana jest wartosc: 7
Do pojemnika wstawiana jest wartosc: 8
Z pojemnika pobierana jest wartosc: 8
Do pojemnika wstawiana jest wartosc: 9
Z pojemnika pobierana jest wartosc: 9
Do pojemnika wstawiana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
```

23

Producer-Consumer problem, version III

```
class Pojemnik {
   private int liczba;
   private boolean czyJestWartoscWPojemniku = false;
   synchronized public int pobierz() {
         if (!czyJestWartoscWPojemniku)
               try {
                     wait();
               }
               catch (InterruptedException e) { }

         System.out.println("Z pojemnika pobierana jest wartosc: " +
                                                liczba);
         czyJestWartoscWPojemniku = false;
         notify();
         return liczba;
   }
```

24

Producer-Consumer problem, version III

```
  synchronized public void wstaw(int liczba) {
        if (czyJestWartoscWPojemniku)
              try {
                    wait();
              }
              catch (InterruptedException e) {     }

        System.out.println("Do pojemnika wstawiana jest wartosc: „
                                                + liczba);
        czyJestWartoscWPojemniku = true;
        this.liczba = liczba;
        notify();
  }
}
```

25

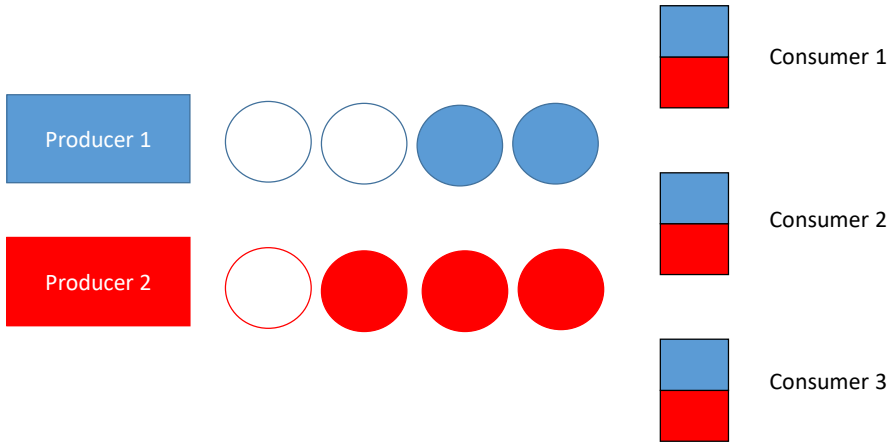
Producer-Consumer problem, version III

Efekt działania programu:

```
Do pojemnika wstawiana jest wartosc: 1
Z pojemnika pobierana jest wartosc: 1
Do pojemnika wstawiana jest wartosc: 2
Z pojemnika pobierana jest wartosc: 2
Do pojemnika wstawiana jest wartosc: 3
Z pojemnika pobierana jest wartosc: 3
Do pojemnika wstawiana jest wartosc: 4
Z pojemnika pobierana jest wartosc: 4
Do pojemnika wstawiana jest wartosc: 5
Z pojemnika pobierana jest wartosc: 5
Do pojemnika wstawiana jest wartosc: 6
Z pojemnika pobierana jest wartosc: 6
Do pojemnika wstawiana jest wartosc: 7
Z pojemnika pobierana jest wartosc: 7
Do pojemnika wstawiana jest wartosc: 8
Z pojemnika pobierana jest wartosc: 8
Do pojemnika wstawiana jest wartosc: 9
Z pojemnika pobierana jest wartosc: 9
Do pojemnika wstawiana jest wartosc: 10
Z pojemnika pobierana jest wartosc: 10
```

26

Homework



The order of requests' realization the same as the order of requests' sending

Thank you for your attention!

Paweł Lula, Cracow University of Economics, Poland
pawel.lula@uek.krakow.pl