Obliczenia równoległe i rozproszone w JAVIE

Michał Kozłowski

30 listopada 2003

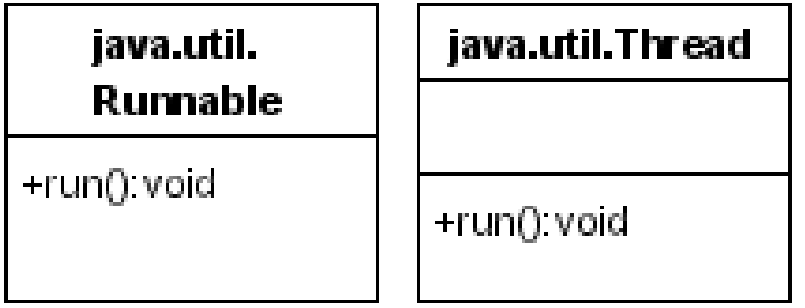
Wątki w JAVIE

- **Reprezentacja wątków jako obiektów**
- Uruchamianie i zatrzymywanie wątków
- Realizacja wątków
- Ograniczenia
- Mechanizmy komunikacji i synchronizacji

Wątki jako obiekty

- Wątkiem w JAVIE może być:

- Obiekt klasy dziedziczącej po **Thread**
- Obiekt klasy implementującej **Runnable**



Przykład

```
public class MyThread extends Thread {
public SimpleThread(String str) { super(str); }
public void run() {
  for (int i = 0; i < 10; i++)
  {
     System.out.println(i);
     try
    {
        sleep((long)(Math.random() * 1000));
    }
    catch (InterruptedException e) {} }
 System.out.println("DONE! " );
  }
}
```

```
public class MyThread2 implements Runnab
   private Thread t = null;

public void start()
    {
      if (t==null)
      {
        t = new Thread(this); t.start();
      }
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            try {
                sleep((long)(Math.random() * 1000))
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! ");
    }
}
```

Sterowanie wątkami

- Wątki uruchamiamy metodą `start()`
- Wątek kończy się wraz z końcem metody `run()`
- Nie należy używać `stop()`,`suspend()`, `resume()`

Priorytety

- JAVA oferuje 10 priorytetów dla wątków
- Ustalanie metodą `setPriority()`
- Warto używać tylko trzech:
  - Thread.MIN_PRIORITY
  - Thread.MAX_PRIORITY
  - Thread.NORM_PRIORITY

Realizacja wątków

Maszyna wirtualna korzysta z mechanizmów wątków udostępnianych przez System Operacyjny

- **Zalety**
  - Dobra wydajność
  - Jedyny sposób na wykorzystanie wielu procesorów
- **Wady**
  - Programista musi być przygotowany na różne scenariusze

Możliwe scenariusze

- Wątki mogą być wywłaszczane lub pracować na zasadzie współpracy
  - Trzeba synchronizować wątki (bo można zostać wywłaszczonym w dowolnym momencie)
  - Trzeba zadbać o to, żeby wątek czasami oddawał procesor (np. przy blokujących operacjach I/O czy explicite zrzekając się procesora – metoda `yield()`
- Priorytety mogą nie działać

Synchronizacja wątków

- Wątki korzystają ze wspólnej przestrzeni adresowej – mogą widzieć i modyfikować te same obiekty

- JAVA udostępnia mechanizm synchronizacji podobny do monitorów

Obiekty jako monitory

- Każdy **obiekt** jest monitorem
- Słowo kluczowe **synchronized**

```
synchronized (obiekt)
{
  // Synchronized Code
}
```

- W danym momencie `Synchronized Code` wykonywać się może tylko w jednym wątku

Wzajemne wykluczanie

- Sekcje krytyczne najlepiej realizować synchronizując się na obiekcie reprezentującym klasę (mamy gwarancję, że jest tylko jeden)

```
synchronized (Mutex.getClass()
{
  //sekcja krytyczna
}
```

Metody synchronizowane

- Słowo kluczowe **synchronized** można stosować też tak:

```
class A {
  public method X() synchronized
  { //anything }
}
```

- Co jest równoważne

```
class A {
  public method X()
  { synchronized (this)
    {//anything} }
}
```

- Statyczne metody ozanczone **synchronized** synchronizują się na `this.getClass()`

wait() i notify()

- Wywołanie `wait()` na danym obiekcie powoduje wstrzymanie wątku dopóki nie zostanie wykonane `notify()` na tym samym obiekcie

- Wątek wywołujący `wait()` musi być właścicielem monitora na którym wykonujemy oczekiwanie (czyli być w bloku synchronizowanym na tym obiekcie). Metoda `wait()` zwalnia monitor, aby możliwe było wykonanie `notify()`

notify() i *notifyAll()*

- `notify()` budzi jeden, arbitralnie wybrany wątek
- `notifyAll()` budzi wszystkie wątki oczekujące na danym obiekcie

Obudzony wątek (wątki) rywalizuje z pozostałymi o blokadę monitora. **Nie** ma żadnej preferencji dla budzonych wątków

- Oznacza to, że konstrukcję postaci:

```
if (!warunek) wait(); {Tu warunek spełniony}
```

- Należy zastąpić przez:

```
while (!warunek) wait(); {Tu warunek spełniony}
```

Przykład 1 - semafor

```
public synchronized void P() throws
  InterruptedException
  {   while( count <= 0 )
       this.wait();
       --count;
  }

public synchronized void V()
{
  int current_count = count;
  count++;
  if (current_count == 0) notifyAll();
}
```

Przykład 2 – czytelnicy i pisarze

```
public class RWLocks {

  private int[] readersLock = new int[1];
  private int[] writersLock = new int[1];
  private int readers = 0;
  private int waitingReaders = 0;
  private int waitingWriters = 0;
  private int  writers = 0;
```

ReadLock()

```
public  void readLock() throws InterruptedException{
   synchronized (this)      {
       if (writers+waitingWriters==0)          {
       readers++; return;
       }
     waitingReaders++;
   }
   synchronized (readersLock)     {
       readersLock.wait();
    }
   synchronized(this)  {
      readers++;
      waitingReaders--;
    }
}
```

ReadUnlock()

```
public void readUnlock() throws InterruptedException
{
   synchronized(this)
    {
      readers--;
       if (readers>0 || waitingWriters==0) return;
   }
    synchronized (writersLock) {
       writersLock.notify();
   }
}
```

WriteLock()

```
public void writeLock() throws InterruptedException
{
  synchronized(this)
  {
       if (readers+writers==0) {
              writers++;
              return;
       }
       waitingWriters++;
   }
  synchronized (writersLock)      {
       writersLock.wait();
   }
   synchronized(this)  {
       waitingWriters--;
       writers++;
   }
}
```

WriteUnlock()

```
public void writeUnlock()throws InterruptedException
{
   synchronized (this)
   {
        writers--;
   }
    synchronized (readersLock)
   {
       readersLock.notifyAll();
   }
}
```

Czy można inaczej?

- `Wait()` i `notify()` wymagają aby być właścicielem monitora – stąd pozornie niepotrzebne sekcje **synchronized**

- Gdyby `wait()` mogło uwalniać inny lock niż obiektu na rzecz którego jest wywoływany – całość możnaby zapisać jako metody synchronizowane

Wątki a bytecode

RMI