Communication with threads. Thread starvation.

Concurrent and parallel programming

Lecture 5.
Academic year: 2018/19

1

Sending data to threads

- no possibility of data sending directly to run() or start() method,
- sending data by constructor parameters,
- sending data by parameters of methods calling before thread's starting

2

Thread results returning

- Techniques of returning of thread results:
  - testing a thread status,
  - waiting for thread completion (“join”),
  - using wait/notify commands,
  - sending a message about thread completion.

3

TESTING THREAD’S STATUS

4

```
12  public class Calculations implements Runnable {
13      protected double res = -1;
14      protected boolean finished = false;
15
16      public void run() {
17          try {
18                  Thread.sleep((int)(100*Math.random()));
19          }
20          catch (InterruptedException e) { }
21
22          res = Math.random();
23          finished = true;
24      }
25
26      public boolean getStatus() {
27
28          return finished;
29      }
30
31      public double getResult() {
32          return res;
33      }
34
35  }
36
```

5

```
12  public class Results01 {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19          Calculations calc1 = new Calculations();
20          Calculations calc2 = new Calculations();
21
22          Thread t1 = new Thread(calc1);
23          Thread t2 = new Thread(calc2);
24          t1.start();
25          t2.start();
26
27
28
29          System.out.println("Result 1: " + calc1.getResult());
30          System.out.println("Result 2: " + calc2.getResult());
31      }
32
33  }
```

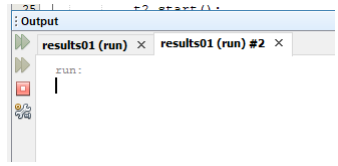
```
Output
results01 (run) ×   results01 (run) #2 ×
run:
Result 1: -1.0
Result 2: -1.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

6

After modification

```
12  public class Results01 {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19          Calculations calc1 = new Calculations();
20          Calculations calc2 = new Calculations();
21
22          Thread t1 = new Thread(calc1);
23          Thread t2 = new Thread(calc2);
24          t1.start();
25          t2.start();
26
            while(!calc1.getStatus()||!calc2.getStatus());
28
29          System.out.println("Result 1: " + calc1.getResult());
30          System.out.println("Result 2: " + calc2.getResult());
31      }
32
33  }
```

waiting for threads' completion



Starvation!!!

7

Starvation

- **Starvation** describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
- This happens when shared resources are made unavailable for long periods by "greedy" threads.

8

Solving a starvation problem

```
12  public class Calculations implements Runnable {
13      protected double res = -1;
14      protected boolean finished = false;
15
16      public void run() {
17          try {
18              Thread.sleep((int)(100*Math.random()));
19          }
20          catch (InterruptedException e) { }
21
22          res = Math.random();
23          finished = true;
24      }
25
26      public boolean getStatus() {
27          Thread.yield();
28          return finished;
29      }
30
31      public double getResult() {
32          return res;
33      }
34
```

allowing other threads to execute

```
Output
results01 (run) ×  results01 (run) #2 ×  results01 (run) #3 ×
run:
Result 1: 0.6649116687356895
Result 2: 0.3216566001828459
BUILD SUCCESSFUL (total time: 0 seconds)
```

9

WAITING FOR THREAD'S COMPLETION

10

Solution with “join” method

```
12  public class Results01 {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19          Calculations calc1 = new Calculations();
20          Calculations calc2 = new Calculations();
21
22          Thread t1 = new Thread(calc1);
23          Thread t2 = new Thread(calc2);
24          t1.start();
25          t2.start();
26
27          //while(!calc1.getStatus()||!calc2.getStatus());
28
29          try {
30              t1.join();
31              t2.join();
32          }
            catch (InterruptedException e) {};
34
35          System.out.println("Result 1: " + calc1.getResult());
36          System.out.println("Result 2: " + calc2.getResult());
37      }
38
39  }
```

11

WAIT/NOTIFY SOLUTION

12

Solution with wait/notify commands

```
11      */
12      public class Calculations implements Runnable {
13          protected double res = -1;
14          protected boolean finished = false;
15
16          protected Object o;
17
18          Calculations(Object o) {
19              this.o = o;
20          }
21
22          public void run() {
23              try {
24                      Thread.sleep((int)(100*Math.random()));
25              }
26              catch (InterruptedException e) { }
27
28              res = Math.random();
29              finished = true;
30
31              synchronized(o) {
32                  o.notifyAll();
33              }
34          }
35
36          public boolean getStatus() {
37              //Thread.yield();
38              return finished;
39          }
40
41          public double getResult() {
42              return res;
43          }
44
45      }
```

13

```
12      public class Results02 {
13
14          /**
15           * @param args the command line arguments
16           */
17          public static void main(String[] args) {
18              // TODO code application logic here
19              Object o = new Object();
20              Calculations calc1 = new Calculations(o);
21              Calculations calc2 = new Calculations(o);
22
23              Thread t1 = new Thread(calc1);
24              Thread t2 = new Thread(calc2);
25              t1.start();
26              t2.start();
27
28              synchronized(o) {
29                  while(!calc1.getStatus()||!calc2.getStatus()) {
30                      try {
31                          o.wait();
32                      }
33                      catch(InterruptedException e)     {};
34                  }
35              }
36
37
38              System.out.println("Result 1: " + calc1.getResult());
39              System.out.println("Result 2: " + calc2.getResult());
40          }
41
42      }
```

14

LISTENERS

Listener interface

```
11  */
    public interface Listener {
        public void handleMessage();
14  }
```

```
13  public class Calculations implements Runnable {
14      protected double res = -1;
15      protected boolean finished = false;
16      ArrayList listeners = new ArrayList();
17
18      public void run() {
19          try {
20                  Thread.sleep((int)(100*Math.random()));
21          }
22          catch (InterruptedException e) { }
23          res = Math.random();
24          finished = true;
25          sendMessages();
26      }
27       public boolean getStatus() {
28          //Thread.yield();
29          return finished;
30      }
31      public double getResult() {
32          return res;
33      }
34      public synchronized void addListener(Listener l) {
35          listeners.add(l);
36      }
37      public synchronized void removeListener(Listener l) {
38          listeners.remove(l);
39      }
40      public synchronized void sendMessages() {
41          ListIterator iter = listeners.listIterator();
42          while(iter.hasNext()) {
43              Listener l = (Listener) iter.next();
44              l.handleMessage();
45          }
46      }
47  }
48
```

17

```
12  public class Manager implements Listener {
13      Thread t1, t2;
14      Calculations calc1, calc2;
15
16      Manager() {
17          calc1 = new Calculations();
18          calc2 = new Calculations();
19
20          calc1.addListener(this);
21          calc2.addListener(this);
22
23          t1 = new Thread(calc1);
24          t2 = new Thread(calc2);
25          t1.start();
26          t2.start();
27      }
28
29      public void handleMessage() {
30          if(!calc1.getStatus()||!calc2.getStatus()) return;
31
32          System.out.println("Result 1: " + calc1.getResult());
33          System.out.println("Result 2: " + calc2.getResult());
34      }
35
36  }
```

18

```
12  public class Results03 {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          Manager m = new Manager();
19          // TODO code application logic here
20      }
21
22  }
23
```

19

Homework

- Solve TSP (Travelling Salesman Problem) using genetic algorithms
- http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5
- Chromosome evaluation should be performed by threads
- Input data: a list of cities

- Distributed version – as exam project (not obligatory)

20

GENETIC ALGORITHMS

Genetic algorithms

GA is a method used for solving optimization problems:

$$f(X_1, X_2, ..., X_N) \rightarrow max$$

or

$$f(X_1, X_2, ..., X_N) \rightarrow min$$

GA shows the development of the population of chromosomes:

C_1 = [01101010000011110001001000010000100]
C_2 = [10010000100001000101111110011111001]
. . . . . .
C_K = [00100001011111101101001110010000100]

Types of chromosomes

Binary

1	0	0	1	1	1	0	0	1	0

Integer

4	2	2	7	1	11	0	2	6	7

Integer (values represent the order of objects /permutation/)

4	2	3	7	1	9	6	5	10	8

Real

3,2	7,1	0,2	-1,5	-4,8	1,3	0,2	-0,9	1,0	0,0

Symbols

D	A	A	C	B	A	C	A	D	C

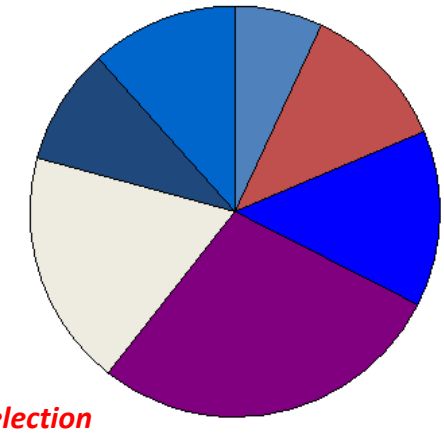
The fitness function

The fitness function – a tool for chromosome evaluation:
chromosome-score = fitness(chromosome)
the greater value of f.f. – the better chromosome (solution)

1 0 ..... 0 1	FF_1
1 1 ..... 0 0	FF_2
0 0 ..... 0 1	FF_3
0 0 ..... 1 0	FF_4
1 0 ..... 0 0	FF_5
.....	.....
1 1 ..... 0 1	FF_K

The selection process

1 0 ..... 0 1	FF_1
1 1 ..... 0 0	FF_2
0 0 ..... 0 1	FF_3
0 0 ..... 1 0	FF_4
1 0 ..... 0 0	FF_5
.....	.....
1 1 ..... 0 1	FF_K



Roulette wheel selection

Crossover and mutation

Crossover operator:

A=[011101100100]
B=[010110111100]

C=[011110111100]

Mutation operator:

A=[0***0***1011100100]
B=[0***1***1011100100]

Methods of problem representation – finding optimal function parameters

Function:

$$f(X_1, X_2, X_3, X_4) \rightarrow max$$

Representation as a binary chromosome:

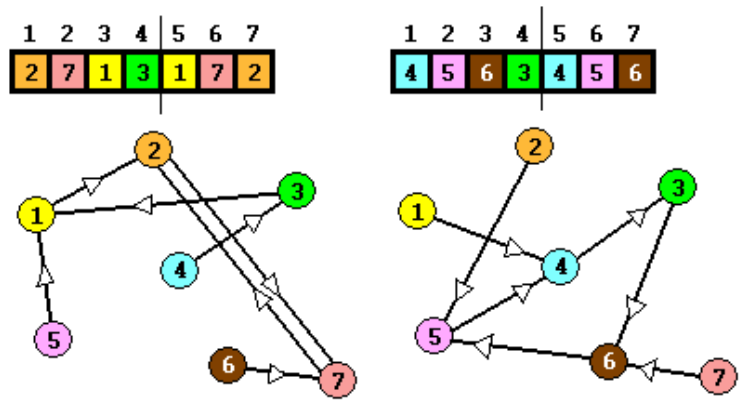
C_i = [10010000100001000101110011110011101]

←—x_1—→←—x_2—→←—x_3—→←—x_4—→

Representation as a real chromosome:

C_i = [4,332; -8,0; 12,1; -32]

Methods of problem representation – the optimal order



Methods of problem representation – variable selection

	Features / Variables		*FF_i=MaxError- $Error_i$ or: $FF_i = 1 / Error_i$*
	$X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ X_7$		
Population of chromosomes:			
1 0 0 1 0 1 0	$Model_1$	$Error_1$	FF_1
1 1 1 1 0 0 0	$Model_2$	$Error_2$	FF_2
0 0 0 1 0 1 1	$Model_3$	$Error_3$	FF_3
0 0 1 1 1 0 0	$Model_4$	$Error_4$	FF_4
1 0 0 0 0 0 1	$Model_5$	$Error_5$	FF_5
1 0 1 1 1 1 0	$Model_6$	$Error_6$	FF_6
1 1 1 0 0 1 1	$Model_7$	$Error_7$	FF_7

Genetic algorithm

Definition the method of problem representation
Creating an initial population of chromosomes

```
for i = 1:numberOfGenerations
{
        Selection
        Crossover and mutation
}
```

Decoding the best chromosome