Programação Paralela (Prof. Fernando Castor)
Alunos: Edwin Marinho e Thomas Cristanis

Trabalho 03

Do livro AMP, devem ser feitos (no mínimo!) os seguintes exercícios: Capítulo 1. 1, 2, 6, 8

Exercise 1. The dining philosophers problem was invented by E. W. Dijkstra, a concurrency pioneer, to clarify the notions of deadlock and starvation freedom. Imagine five philosophers who spend their lives just thinking and feasting. They sit around a circular table with five chairs. The table has a big plate of rice. How- ever, there are only five chopsticks (in the original formulation forks) available, as shown in Fig. 1.5. Each philosopher thinks. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him. If a philosopher can pick up both chopsticks, he can eat for a while. After a philosopher finishes eating, he puts down the chopsticks and again starts to think.

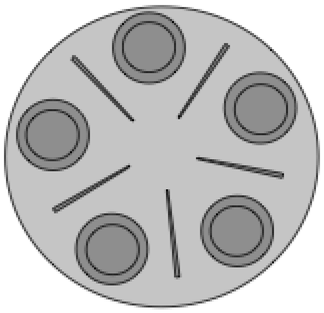

Figure 1.5 Traditional dining table arrangement according to Dijkstra.

1. Write a program to simulate the behavior of the philosophers, where each philosopher is a thread and the chopsticks are shared objects. Notice that you must prevent a situation where two philosophers hold the same chopstick at the same time.
2. Amend your program so that it never reaches a state where philosophers are deadlocked, that is, it is never the case that each philosopher holds one chopstick and is stuck waiting for another to get the second chopstick.
3. Amend your program so that no philosopher ever starves.
4. Write a program to provide a starvation-free solution for any number of philosophers n.

Arquivo: Chopstick.java

```
public class Chopstick {
  public boolean busy;
}
```

Arquivo: Table.java

```
import java.util.LinkedList;
import java.util.List;

public class Table {
  List<Chopstick> chopsticks = new LinkedList<>();

  public synchronized boolean pickChopsticks(Philosopher philosopher) {
    if (chopsticks.get(philosopher.leftChopstickIndex).busy
        || chopsticks.get(philosopher.rightChopstickIndex).busy) {
      return false;
    }

    chopsticks.get(philosopher.leftChopstickIndex).busy = true;
    chopsticks.get(philosopher.rightChopstickIndex).busy = true;
    return true;
  }

  public void releaseChopsticks(Philosopher philosopher) {
    chopsticks.get(philosopher.leftChopstickIndex).busy = false;
    chopsticks.get(philosopher.rightChopstickIndex).busy = false;
  }
}
```

Arquivo: Philosopher.java

```
import java.util.Random;

public class Philosopher implements Runnable {

  public int id;
  public int leftChopstickIndex;
  public int rightChopstickIndex;
  public Table table;

  @Override
  public void run() {
    while (true) {
      Random random = new Random();
      int thinkingTime = random.nextInt(2000);
      try {
        Thread.sleep(thinkingTime);
      } catch (InterruptedException e) {
        System.out.println("An error has ocurred on thread sleep.");
      }

      while (!this.table.pickChopsticks(this)) {
        // wait someone to release
      }

      this.eat();
      this.table.releaseChopsticks(this);
    }
  }

  public void eat() {
    System.out.println("Philosoper " + this.id + " is eating...");
  }
}
```

Arquivo: DiningPhilosophers.java

```
import java.util.LinkedList;
import java.util.List;

public class DiningPhilosophers {
  private static final int NUMBER_OF_PHILOSOPHERS = 8;

  public void execute() {
    Table table = new Table();
    table.chopsticks.add(new Chopstick());

    List<Philosopher> philosophers = new LinkedList<Philosopher>();
    for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
      table.chopsticks.add(new Chopstick());
      Philosopher philosopher = new Philosopher();
      philosopher.id = i + 1;
      philosopher.leftChopstickIndex = i;
      philosopher.rightChopstickIndex = (i == NUMBER_OF_PHILOSOPHERS - 1) ? 0 : (i + 1);
      philosopher.table = table;
      philosophers.add(philosopher);

      Thread thread = new Thread(philosopher);
      thread.start();
    }
  }

  public static void main(String[] args) {
    DiningPhilosophers diningPhilosophers = new DiningPhilosophers();
    diningPhilosophers.execute();
  }
}
```

Exercise2. For each of the following, state whether it is a safety or liveness property. Identify the bad or good thing of interest.

> *A safety property states that some "bad thing" never happens. For example, a traffic light never displays green in all directions, even if the power fails.*

A liveness property states that a particular good thing will happen. For example, a red traffic light will eventually turn green.

1. Patrons are served in the order they arrive.
   liveness

2. What goes up must come down.
   safety

3. If two or more processes are waiting to enter their critical sections, at least one succeeds.
   liveness

4. If an interrupt occurs, then a message is printed within one second.
   safety

5. If an interrupt occurs, then a message is printed.
   safety

6. The cost of living never decreases.
   safety

7. Two things are certain: death and taxes.
   liveness

8. You can always tell a Harvard man.
   liveness

Exercise 6. Use Amdahl's law to answer the following questions:

- Suppose the sequential part of a program accounts for 40% of the program's execution time on a single processor. Find a limit for the overall speedup that can be achieved by running the program on a multiprocessor machine.
  $p = 60/100 = 0{,}6$
  $n = \infty$

  $S = 1 / ((1 - p) + (p / n))$
  $= 1 / ((1 - 0{,}6) + (0{,}6 / \infty))$
  $= 1 / (0{,}4 + 0)$
  $= 1/ 0{,}4$
  $= 2{,}5$

- Now suppose the sequential part accounts for 30% of the program's computation time. Let sn be the program's speedup on n processes, assuming the rest of the program is perfectly parallelizable. Your boss tells you to double this speedup: the revised program should have speedup s'n > sn/2. You advertize for a programmer to replace the sequential part with an improved version that runs k times faster. What value of k should you require?
  sn = 1 / 0,3

  s'n = (1 / 0,3) / 2
  s'n = 6,66
  s'n = 1 / x
  6,66 = 1 / x
  x = 1 / 6,66
  x = 0,15

  0,3 / k = 0,15
  k = 0,3 / 0,15 = 2

- Suppose the sequential part can be sped up three-fold, and when we do so, the modified program takes half the time of the original on n processors. What fraction of the overall execution time did the sequential part account for? Express your answer as a function of n.
  [...]

Exercise 8. You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's Law, explain how you would decide which to buy for a particular application.
[...]