Concurrent and Distributed Systems
Concurrent Objects

Mitre Flavia Antonia

December 21, 2019

Third Year
Group: C.E.N. 3.1
Lecturer: Costin Bădică
Teaching assistant: Cristinel Ungureanu

1 Problem statement

1.1 First problem

Implement the dining philosophers problem using channels.

2 Implementation

2.1 First problem

2.1.1 Channels

For solving the dining philosophers problem with **channels** I've created the classes:

- **DiningPhilosophers**

  This class contains the **main** of the program.
  We have 5 philosophers, so an **array** of 5 **Philosophers** is necessary.
  We also have 5 **forks** (a fork between two philosophers), so an array of 5 **Objects** is necessary.
  The **forks** communicate through **channels** with the philosophers. We also need an array of 5 fork channels.
  Every **fork** is a new **Object**.

  Every **fork channel** is a new **Channel** of type boolean.

  We count the forks from right to left, as this is the logical order of staying at a table.
  The **right fork** is the first fork numbered.
  The **left fork** is the next fork numbered.
  This happens for every philosopher's fork. The same for fork channels.
  A new **Philosopher** instance is created.
  The fifth philosopher has to pick up the right fork and then the left fork to avoid deadlocks.
  A new **thread** is created, for every philosopher.
  This threads are also **started**.

- **Philosopher**

  This class implements **runnable** (overrides the **run** method for threads).

  The **forks** on either sides of the Philosopher are two **Objects**.
  For the **forks** on either sides of the Philosopher there is a **Channel**.
  In the **constructor** for **Philosopher**:

  -We access his first and second **forks** using "this".

  -We access his left and right **fork channels** using "this".

  At first all the philosophers are **thinking**. We assume that this takes some **time**.

In the run method the while loops forever.

The **first fork** is synchronised. If a philosopher wants to pick up a fork, it must be free.
The **first fork** is picked up by sending a message to its channel.
The **second fork** is synchronised. If a philosopher wants to pick up a fork, it must be free.
The **second fork** is picked up by sending a message to its channel.
The **second fork** is put down, so the message is received to its channel.
The **first fork** is put down, so the message is received to its channel.

End of while loop.

The receive function (called inside the while loop) can throw an exception, so the catch clause is added here.

- **Channel**

The dining philosophers problem should be solved using **asynchronous communication**, because if the message can not be received at the moment the channel should not be blocked (otherwise it causes a deadlock). The message should be stored temporary until the receiver gets to receive it.

This class has the generic type T.
Will be used in the implementation with Boolean type (the messages sent will have the value "true" every time, because we don't care about the message itself, only about the fact that it is sent).

The global variable **ready** is used to check if the channel is ready to **send/receive** a message.

This class contains two methods: **Send** and **Receive**.

The **Send** method is synchronised and of type void.
It has the following parameters: the sent **message**, the **side** of philosopher where the fork is taken from (left or right) and the **number of the philosopher** that sends the message (that picks up a fork).
According to the philosopher number and the **side** of philosopher where the fork is taken from, a message is printed (which fork is picked by who, if the philosopher eats + sleep etc).
Notifies potential receivers when it is ready.

The **Receive** method is synchronised and of type void.
It has the following parameters: the **side** of philosopher where the fork is taken from (left or right) and the **number of the philosopher** that receives the message (that puts down a fork).
According to the philosopher number and the **side** of philosopher where the fork is taken from, a message is printed (which fork is put down by who, if the philosopher thinks + sleep etc).
Expects a message to be available on the channel, when it is ready.
Releases the channel.

Pseudocode

- **Channels:**

1. Philosopher i (0 <= i <= 4)
2. Object array[0..4] $fork$
3. Channel array[0..4] $forkchannels$
4. boolean message
5. **loop forever**
6. think
7. forks[i] $\rightarrow$ message
8. forks[i+1] $\rightarrow$ message
9. eat
10. forks[i+1] $\leftarrow$ true
11. forks[i] $\leftarrow$ true
12. think
13. **end loop**

3 Experimental Data

3.1 First problem

3.1.1 Channels

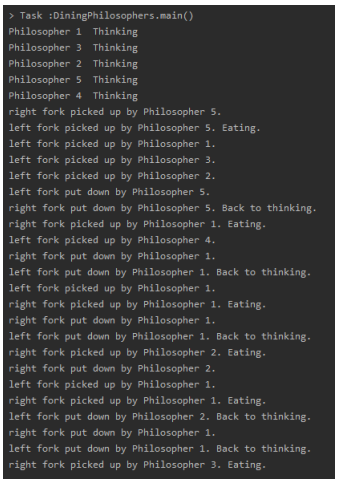

Figure 1: First output

At first all the philosophers are thinking. The fifth philosopher picks up the right fork (avoiding deadlocks), then the left fork and eats. The first philosopher eats multiple times, the second and the third eat too. They never pick up the same fork. Also they don't put down a fork without eating.

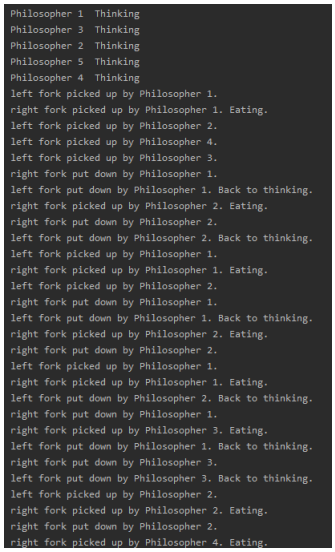

Figure 2: Second output
At first all the philosophers are thinking. The first four philosophers eat.

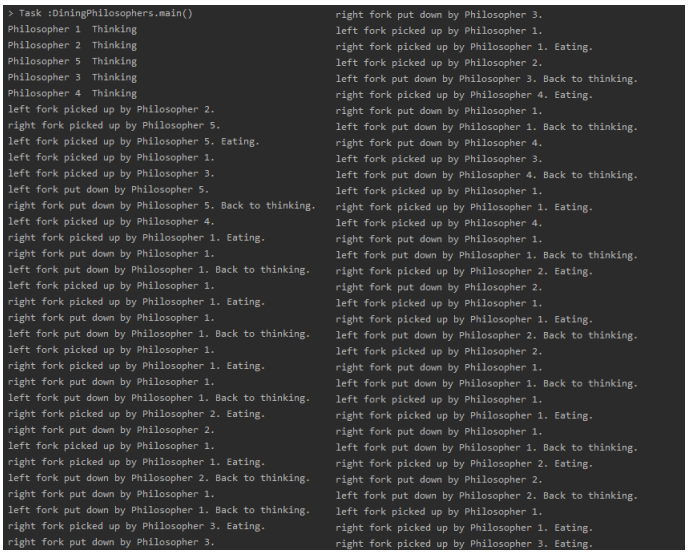

Figure 3: Third output
All the philosophers eat, but reaching this state takes some time, because only philosophers next to the ones that recently ate can eat next. Starvation is avoided.
The fifth philosopher eats first.
The first philosopher eats more often.

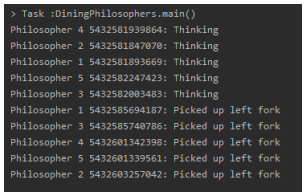

Figure 4: Fourth output

The dining philosophers problem can have a point to stop: a deadlock (this problem was obviously solved). This is the classical example of a deadlock, where all the philosophers have picked up the left fork and every one of them will be waiting for the right fork forever.

4 Conclusions

The dining philosophers problem can be solved in many ways, including channels.

A channel connects a transmitting process with a receiving process.
Usually, a channel has a type that describes the type of messages that can be transported by the respective channel.
A channel is synchronous.

For this problem the values of the messages do not matter, boolean values equal to true are used. When messages arrive, it means that the forks are free and the philosopher will use them. They are released by sending each one a message on the respective channels.

Channels guarantee that two philosophers can never pick up the same forks by synchronising them, and also that the last philosopher can't pick up the left fork (avoiding deadlocks) if the rest already have picked up one.
After some time all philosophers have the chance to eat.
The algorithm is efficient in case of forks conflicts.