Concurrent and Distributed Systems
Synchronization Mechanisms

Mitre Flavia Antonia

December 12, 2019

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

1 Problem statement

1.1 Second problem

Implement the dining philosophers problem as follows:
a) using semaphores
b) using monitors
c) using locks

2 Implementation

2.1 Second problem

2.1.1 Semaphores

For solving the second problem with **semaphores** 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 **Semaphores** (one for every fork) is necessary.

  Every **fork** is a new **Semaphore** with one permit (only one Philosopher can grab it at a time).

  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.
  A new **Philosopher** instance is created.
  A new **thread** is created, for every philosopher.
  This threads are also **started**.

- **Philosopher**

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

  It contains the declaration for the **room** semaphore, used to block the Philosophers number that can hold the left fork at the same time to the maxim number of Philosophers minus one (here, 5 - 1 = 4).

  The function **writeAction** is used to write on the screen what action the Philosopher does (his number + his action).

  The **forks** on either sides of the Philosopher are two **Semaphores**.

  In the **constructor** for **Philosopher**:

  -The **room** semaphore sets the Philosophers number that can hold the left fork at the same time to 4 (permits value).

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

In the run method the while loops forever.

At first all the philosophers are **thinking**. We assume that this takes some **time**.
A philosopher will try to **acquire** his **left fork**. If a philosopher wants to pick up a fork, he must have **room** to do it.
Calling the left fork acquire function prevents it from being picked by other philosopher later.
The philosopher will try to **acquire** the **right fork**.
Calling the right fork acquire function prevents this fork from being picked by other philosopher.
If this fork is picked, it means a philosopher has both forks, and start **eating**. We assume that this takes some **time**.
After the philosopher eats, he **puts down** the **right fork**. The right fork is released.
Then, the philosopher **puts down** the **left fork**. The left fork is **released**.
This means that this philosopher holds no forks, so **room** is **released** for another one.

End of while loop.

Pseudocode

- **Semaphores:**

1. Philosopher i (0 <= i <= 4)
2. Semaphore array[0..4] $fork \leftarrow$ [1,1,1,1,1]
3. Semaphore room $\leftarrow$ 4
4. **loop forever**
5. think
6. wait(room)
7. wait(fork[i])
8. wait(fork[i+1])
9. eat
10. signal(fork[i+1])
11. signal(fork[i])
12. signal(room)
13. **end loop**

2.1.2 Monitors

For solving the second problem with **monitors** 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** (one for every fork) is necessary.

  Every **fork** is a new **Object**.

  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.
  A new **Philosopher** instance is created.
  The last philosopher has to pick up first the **right fork**, and then the **left fork**.
  A new **thread** is created, for every philosopher.
  This thread is also **started**.

- **Philosopher**

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

  The function **writeAction** is used to write on the screen what action the Philosopher does (his number + his action).

  The **forks** on either sides of the Philosopher are two **Objects**.

  In the **constructor** for **Philosopher** we access his left and right **forks** using "this".

  In the run method the while loops forever.

  At first all the philosophers are **thinking**. We assume that this takes some **time**.
  The **left fork** is synchronized and picked up.
  The **right fork** is synchronized and picked up.
  Calling the left fork acquire function prevents it from being picked by other philosopher later.
  It means a philosopher has both forks, and start **eating**.
  We assume that this takes some **time**.
  After the philosopher eats, he **puts down** the **right fork**.
  Then, the philosopher **puts down** the **left fork** and gets back to thinking.

End of while loop.

Pseudocode

- **Monitors:**

1. Philosopher i (0 <= i <= 4)
2. Object array[0..4] $fork$
3. Object $LeftFork$
4. Object $RightFork$
5. **loop forever**
6. think
7. synchronize(fork[i])
8. synchronize(fork[i+1])
9. eat
10. put down fork[i+1]
11. put down fork[i]
12. think
13. **end loop**

2.1.3 Locks

For solving the second problem with **locks** 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 **Locks** (one for every fork) is necessary.

  Every **fork** is a new **ReentrantLock**.

  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.
  A new **Philosopher** instance is created.
  The last philosopher picks up first the right fork, then the left fork.
  This helps avoiding a **livelock** (without this constraint, it is possible that every philosopher will first pick up the left fork, then will try to pick up the right fork, and they will all fail, so they will repeat this process on and on).
  A new **thread** is created, for every philosopher.
  This thread is also **started**.

- **Philosopher**

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

  The function **writeAction** is used to write on the screen what action the Philosopher does (his number + his action).

  The **forks** on either sides of the Philosopher are two **Locks**.

  In the **constructor** for **Philosopher**

  We access his left and right **forks** using "this".

  The while loops forever.

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

  A philosopher will try to pick up (using **trylock**) his **left fork**.
  If the philosopher **didn't succeed** to pick up the **left fork**, the **continue** statement enables going back to the start of the loop.
  Else, he will try to pick up (using **trylock**) his **right fork**.
  If the philosopher **didn't succeed** to pick up the **right fork**, he will **put back** his **left fork** and start **thinking**, also, the **continue** statement enables going back to the start of the loop.
  Else, he will start **eating** for some **time**, then, he will **put back** the **right fork**, then the **left fork** and start **thinking** again.

  End of while loop.

Pseudocode

- **Locks:**

  1. Philosopher i (0 <= i <= 4)
  2. ReentrantLock array[0..4] $fork$
  3. Lock $LeftFork$
  4. Lock $RightFork$
  5. **loop forever**
  6. think
  7. trylock(fork[i])
  8. **if** trylock(fork[i]) = false
  9. go back to start of loop
  10. **end if**
  11. trylock(fork[i+1])

12. **if** trylock(fork[i+1]) = false
13. unlock(fork[i])
14. go back to start of loop
15. **end if**
16. eat
17. put right fork down
18. put left fork down
19. think
20. **end loop**

3 Experimental Data

3.1 Second problem

3.1.1 Semaphores

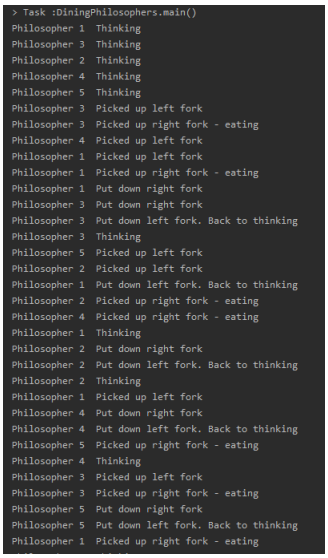

Figure 1: First output

At first all the philosophers are thinking. The third philosopher picks the left fork, then picks the right fork and eats. The fourth philosopher picks the left fork. It will try to pick the right fork, but that is not possible, because the third philosopher is still eating with that fork. The first philosopher picks the left fork, then the right fork (it was available) and eats, then puts down first the right fork. The third philosopher puts down the right fork, then the left fork and starts thinking. The fifth philosopher picks up the left fork. The second philosopher picks up the left fork. The first philosopher puts down the left fork and starts thinking. The second philosopher picks up the right fork (immediately after it was released by the first philosopher) and starts eating. The fourth philosopher picks the right forks and starts eating. The first thinks. The second put down the right and the left fork and starts thinking. The first picks the left fork. The fourth puts down the right and the left fork and gets back to thinking. The fifth philosopher grabs the right fork (immediately after being released by the fourth philosopher) and starts eating. The fourth thinks.

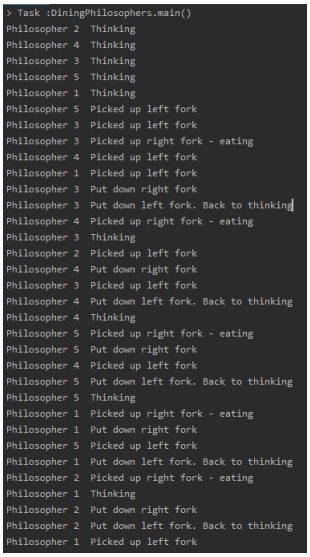

Figure 2: Second output

At first all the philosophers are thinking. The fifth philosopher picks up the left fork. The third philosopher picks up the left fork, then the right fork and eats. The fourth philosopher picks up the left fork. The first philosopher picks up the left fork. The third philosopher puts down the right and the left fork and starts thinking. The fourth philosopher picks up the right fork and starts eating. The third philosopher thinks. The second philosopher picks up the left fork. The fourth philosopher puts down the right fork and the third picks it up (the left fork for the third philosopher is the same as the right forks for the fourth philosopher). The fourth philosopher puts down the left fork and thinks. The fifth philosopher picks up the right fork and eats, then puts the right fork down. The fourth philosopher grabs the left forks (available because the fifth just put it down). The fifth philosopher puts the left fork down and thinks. This process never stops, the only way to stop it is manually in the used tool.

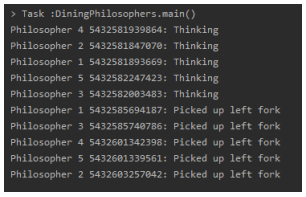

Figure 3: Third output

Another way that the philosophers problem can have a point to stop is a deadlock (which 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.

Figure 4: Fourth output

This is another example of a deadlock, where the blocking happens after some other actions (the first philosopher and the fifth philosopher manage to eat). But in the end all the philosophers have picked up the left fork and every one of them will be waiting for the right fork forever.

3.1.2 Monitors

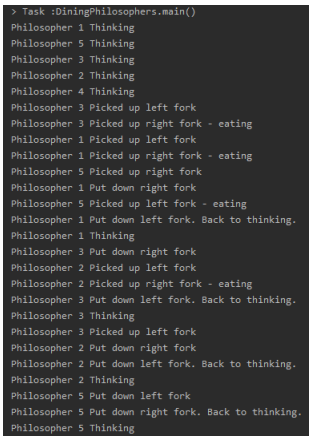

Figure 5: First output

At first all the philosophers think for some time.
The third philosopher picks up the left fork and then the right fork and starts eating. The first philosopher picks up the left fork and then the right fork and starts eating. The fifth philosopher picks up the right fork.
The first philosopher puts down the right fork.
The fifth philosopher picks up the left fork and starts eating (avoiding deadlocks because the last philosopher picks up first the right fork, then the left fork). The first philosopher puts down the left fork and starts thinking.
The third philosopher puts down the right fork, the second philosopher picks it up (the second's philosopher left fork is the same as the third's philosopher right fork) and then the right fork and eats.
The third philosopher puts down the left fork, thinks again.
The first philosopher puts down the left fork and thinks again.

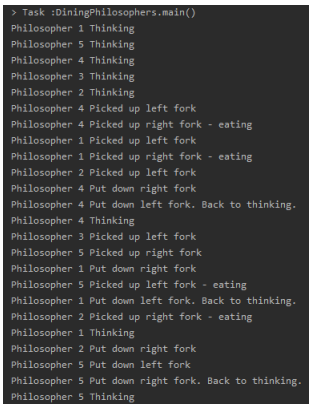

Figure 6: Second output

At first all the philosophers think for some time.
The fourth philosopher picks up the left fork and then the right fork and starts eating. The first philosopher picks up the right fork and starts eating.
The second philosopher picks up the left fork.
The fourth philosopher puts down the right fork and then the left fork and starts thinking. The third philosopher picks up the left fork.
The fifth philosopher picks up the right fork.
The first philosopher puts down the left fork.
The fifth philosopher picks up the left fork and starts eating.
The first philosopher puts down the left fork and starts thinking.
The second philosopher the left fork and starts eating.
The first philosopher thinks. The second philosopher puts down the right fork.
The fifth philosopher puts down the left fork, then the right fork and starts thinking.

3.1.3 Locks

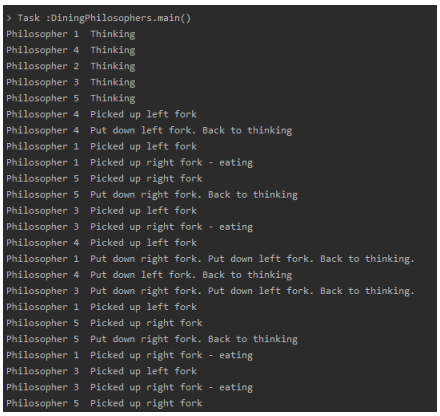

Figure 7: First output

At first all the philosophers think for some time.
The fourth philosopher picks up the left fork and then puts it down and thinks again (he didn't succeed to pick up the right fork).
The first philosopher picks up the left fork and then the right fork and starts eating.
The fifth philosopher picks up the right fork and then puts it down (he didn't succeed to pick up the left fork). Picking up the right fork first assures avoiding a livelock.
The third philosopher picks up the left fork and then the right fork and starts eating.
The fourth philosopher picks up the left fork.
The first philosopher puts down the left right fork and then the left fork and starts thinking again.
The fourth philosopher puts down the left fork and thinks again (he didn't succeed to pick up the right fork).
The third philosopher puts down the right fork and then the left fork and starts thinking again.
The first philosopher picks up the left fork.
The fifth philosopher picks up the right and then puts it down (he didn't succeed to pick up the left fork).
And so on. This algorithm doesn't avoid starvation, only two philosophers get to eat. The rest can pick only one fork and then put it down.

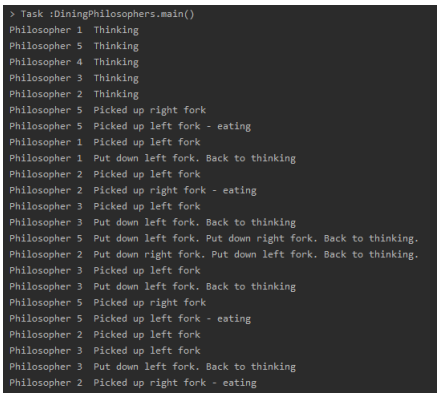

Figure 8: Second output

At first all the philosophers think for some time.
The fifth philosopher picks up the right fork and then the left fork and starts eating. Picking up the right fork first assures avoiding a livelock.
The first philosopher picks up the left fork and then puts it down (he didn't succeed to pick up the right fork).
The second philosopher picks up the left fork and then the right fork and starts eating. The third philosopher picks up the left fork and then puts it down (he didn't succeed to pick up the right fork).
The fifth philosopher puts down the right fork and then the left fork and starts thinking again. The second philosopher puts down the left fork and then the right fork and starts thinking again. And so on. This algorithm doesn't avoid starvation, only two philosophers get to eat. The rest can pick only one fork and then put it down. The fifth philosopher eats by picking up first the right fork and then the left fork.

4 Conclusions

The dining philosophers problem can be solved in many ways, including using semaphores, monitors or locks.

The synchronisation mechanisms are used because they can guarantee that a philosopher eats only when he has two forks, two philosophers can never pick up the same forks simultaneous, avoid deadlocks and starvation, and also, are efficient in case of forks conflicts.
Still, there are some differences between them.

Semaphores guarantee that two philosophers can never pick up the same forks by making the forks semaphores, and also that the last philosopher can't pick up the left fork (avoiding deadlocks) if the rest already have picked up one (the room semaphore permits only the number of philosophers minus one to pick up the left fork at a time).
Semaphores also avoid starvation (all philosophers have the chance to eat) and are efficient in case of forks conflicts.

Monitors guarantee that two philosophers can never pick up the same forks by synchronising them. Also a constraint is needed. The last philosopher can't pick up the left fork first (avoiding deadlocks), so he must pick the right one first.
Monitors also avoid starvation (all philosophers have the chance to eat) and are efficient in case of forks conflicts.

Locks avoid deadlocks, livelocks and are efficient in case of forks conflicts, but don't avoid starvation (not all philosophers get to eat).
A deadlock-free solution is not necessarily starvation-free.