Laboratory Practices

The dining philosophers problem (1 session)

Concurrency and Distributed Systems

Introduction

The goal of this practice is to design and implement different solutions to the deadlock problem. When you complete the proposed tasks you will have learned how to:

- Compile and execute concurrent programs.
- Detect deadlocks in a concurrent program.
- Design solutions to solve the deadlock problem.
- Implement those solutions.

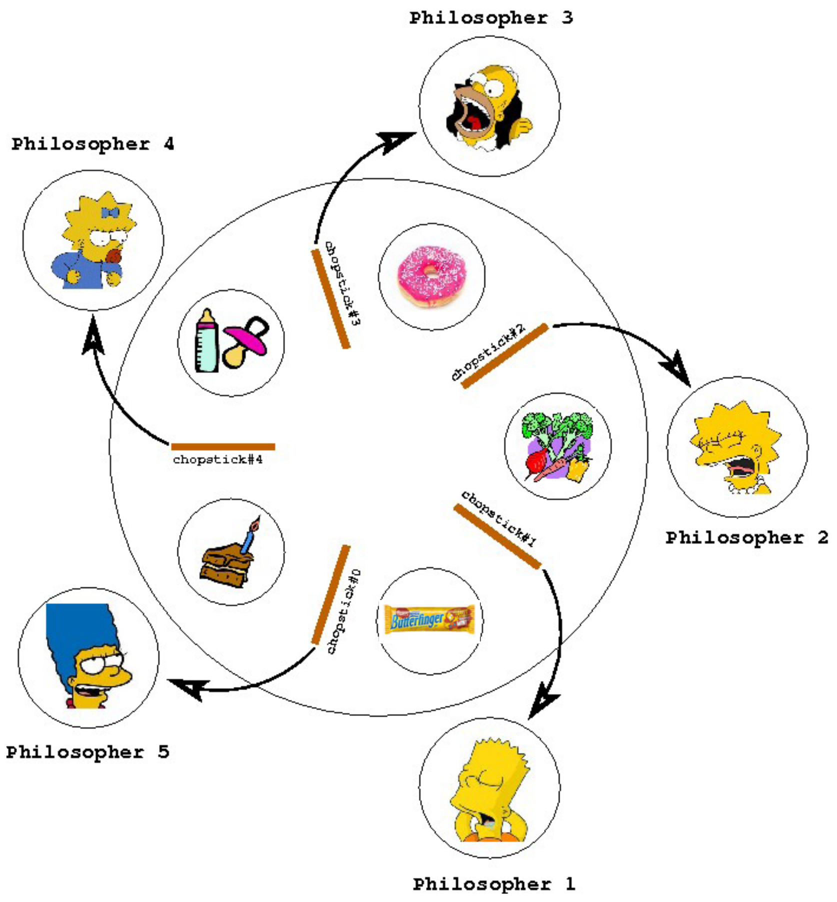
Java will be used as a programming language.

You will need approximately one week to complete the practice assignments. In the lab session your professor can help you with technical questions you might find. You might also need additional time to complete the practice. To that end you can use laboratory facilities during their free access schedule. You could also use your own personal computer.

Along the practice you will notice that there are exercises to complete. It is recommended to solve them and to annotate their results in the provided boxes, to facilitate the future study of the content of the practice.

The dining philosophers problem

During this practice we use the classical problem of the “five dining philosophers” as an outstanding example to illustrate deadlocks and their solutions. This problem is also described in unit "Deadlocks" of this course.



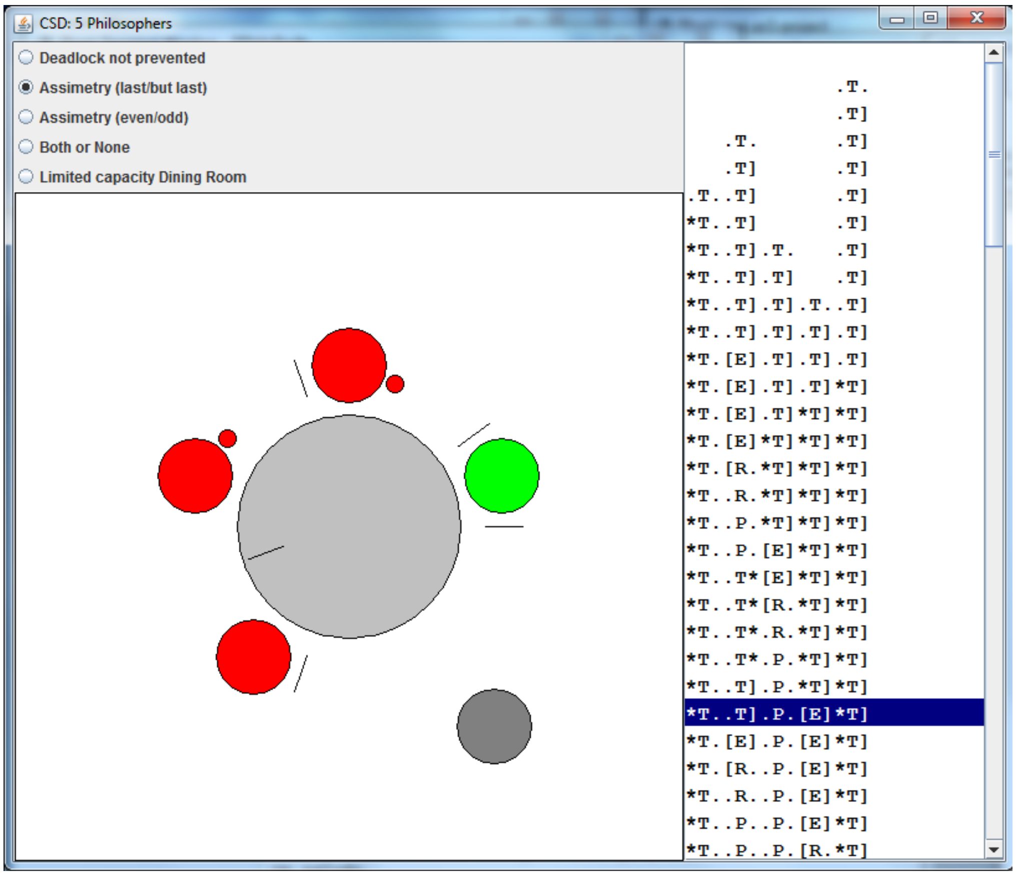
We say that there could be a deadlock if there is a chance that a deadlock appears, even if that situation is very unlikely to happen in practice. Using our dining philosophers problem, we can realize that deadlocks are more likely to appear if philosophers make a long pause after taking the first chopstick and before taking the second one, rather than if they take both chopsticks with no time in between. Since deadlocks appear or not depending on several factors, the simple observation of a deadlock free execution of a given program does not demonstrate its correctness about deadlocks.

Software provided

You are provided with a software environment where you can develop and execute solutions to the dining philosophers problem. The provided code does not solve the problem but it provides a skeleton to complete the requested implementations. The code as it is, can be compiled and executed. It is packaged into the file “PPhilo.jar”, which can be found in PoliformaT, in the section corresponding to Lab practice 2.

The user interface is quite similar to the one already used in last practice (shared pool). At the upper left corner of the window there are selection options. Selecting an option triggers the execution of the corresponding simulation. At the right-hand side, you can find text-based states that explain the simulation execution and an additional graphical representation of the simulation can be found at the left side. You can select a given text-based state and analyze its graphical representation.

Initially, only the simulation of the case "Deadlock not prevented" is complete and students must implement the rest of the simulations when working on the different practice activities.



( ) Deadlock not prevented
[✓ Assimetry (last/but last)]
( ) Assimetry (even/odd)
( ) Both or None
( ) Limited capacity Dining Room

```
            .T.
            .T]
   .T.      .T]
   .T]      .T]
.T..T]      .T]
*T..T]      .T]
*T..T].T.   .T]
*T..T].T]   .T]
*T..T].T].T..T]
*T..T].T].T].T]
*T.[E].T].T].T]
*T.[E].T].T]*T]
*T.[E].T]*T]*T]
*T.[E]*T]*T]*T]
*T.[R.*T]*T]*T]
*T..R.*T]*T]*T]
*T..P.*T]*T]*T]
*T..P.[E]*T]*T]
*T..T*[E]*T]*T]
*T..T*[R.*T]*T]
*T..T*.R.*T]*T]
*T..T*.P.*T]*T]
*T..T].P.*T]*T]
*T..T].P.[E]*T]
*T.[E].P.[E]*T]
*T.[R..P.[E]*T]
*T..R..P.[E]*T]
*T..P..P.[E]*T]
*T..P..P.[R.*T]
```

Next table is intended to help you understand the different possible states for a philosopher and their corresponding textual and graphical representations. Each philosopher state is represented by 3 characters. The central one describes the philosopher state (talking, pondering, eating, waiting) and the other 2 represent the chopstiks on his/her left and right.

State	**Txt**	**Graphical representation**
Inactive (philosopher has terminated)		
Waiting for a seat (enter)	.*.	Red circle far from the table.
Talking at the table	.T.	Red circle close to the table.
Talking at the table, with the right hand side chopstick.	.T]	Red circle close to the table, a line at the right side represents the chopstick.
Talking at the table, with the left hand side chopstick.	[T.	Red circle close to the table, a line at the left side represents the chopstick.
Eating	[E]	Green circle close to the table. Two lines represent both chopsticks.
Pondering	.P.	Grey circle far from the table.

Talking at the table, with no chopstick, waiting for the right hand side chopstick.	.T*	Red circle close to the table, little red circle at the right side represents that chopstick is requested.
Talking at the table, with no chopstick, waiting for the chopstick on his/her left.	*T.	Red circle close to the table, little red circle at the left side showing that left chopstick is waited for.
Talking at the table, with the chopstick on his/her right and waiting for the chopstick on his/her left.	*T]	Red circle close to the table, little red circle at the left side and a line at the right side.
Talking at the table, with the chopstick on his/her left and waiting for the chopstick on his/her right.	[T*	Red circle close to the table, little red circle at the right side and a line at the left side.
Talking at the table, waiting for both chopsticks.	*T*	Red circle close to the table. Two little red circles at both sides.
Sitting at the table, resting after eating, but still holding the left chopstick.	[R.	White circle close to the table with a line at the left side. NOTE: Right chopstick must be dropped before dropping the left one.
Sitting at the table, resting after eating and before pondering again.	.R.	White circle close to the table.

Philosopher at the upper part of the table is philosopher number 0, and the other philosophers are numbered clockwise (1, 2, 3, 4). Text messages are shown with philosophers ordered from 4 to 0. For instance, the following text state:

Philosopher	4	3	2	1	0
State	`*T.`	`.T]`	`.P.`	`[E]`	`*T]`

Should be interpreted as follows:

- Philosopher 4 is talking and waiting for the left chopstick.
- Philosopher 3 is talking and it has its right chopstick.
- Philosopher 2 is pondering.
- Philosopher 1 is eating.
- Philosopher 0 is talking. He has his right chopstick but is waiting for the other one (Philosopher 1 has it).

Notice that the code must invoke the required methods of *state* (object of the class *StateManager*) so that the user interface correctly reflects each philosopher state. You can see how this is done in the class *RegularTable*. The following table shows the methods of *StateManager* that perform the change from one state to another. It must be taken into account that some actions are only valid for some types of tables.

Method of *StateManager* class	**Philosopher's action**
begin	Starts
end	Ends
wenter	Waits to sit at the table (enter)
enter	Sits at the table (enters)
exit	Gets up from the table (exits)
takeR	Takes the chopstick on his/her right
takeL	Takes the chopstick on his/her left
takeLR	Takes both chopsticks
eat	Starts to eat
ponder	Starts to ponder (think)
wtakeR	Waits to take the chopstick on his/her right
wtakeL	Waits to take the chopstick on his/her right
wtakeLR	Waits to take both chopsticks
dropR	Drops the chopstick on his/her right
dropL	Drops the chopstick on his/her left

Activity 0

The main goal of this activity is to realize that deadlocks might appear with the provided code as is. Students should also realize that deadlocks are difficult problems mainly because deadlock probabilities are usually very low (but very important).

You should invest some minutes studying the files and code provided in the jar file. Study class *Philo* and class *RegularTable* (which implements interface Table).

class	description
Philo	Instances of this class are the threads that perform philosophers' activities.
RegularTable	This class implements a monitor that controls how philosophers access chopsticks. It ensures mutual exclusion but it does not ensure a deadlock free execution.

Now, pay attention to the following piece of code from class *Philo*, method run():

```
table.takeR(id); delay(msegDelay); table.takeL(id);
```

In order to increase the deadlock probability, there is a delay after taking the first chopstick and before taking the second one. The number of milliseconds to wait can be set as a command line argument to the program. Accepted values range from 1 to 10. If there is no argument, the argument is invalid or if it is out of range, a default value of 10 is used.

Exercise 0.1: Execute the program 10 times (selecting option "Deadlock not prevented"), changing the delay time T and write down how many times there is a deadlock.

T	Number of deadlocks
1	
5	
10	

NOTE.- After the execution, a message DEADLOCK or OK appears on the console. This can be also checked by analyzing the final state (if everything goes well, the philosophers will disappear and all the chopsticks will be on the table, and in case of deadlock all philosophers will be waiting on the table).

Exercise 0.2: How does the value of T influence the probability of a deadlock?

Activities

We propose 4 different options to prevent deadlocks.

Version	Description
1	Philosophers always take first the chopstick on their right and then the chopstick on their left, except for philosopher 4, who takes the chopstick on his left first.
2	Even-numbered philosophers take the chopstick on their right first, while odd-numbered philosophers start by taking the chopstick on their left.
3	Philosophers take both chopsticks or they take none of them.
4	Only 4 philosophers can sit at the table at the same time.

In the next activities you are asked to implement and test those solutions.
It is important to mention again that each solution must invoke the appropriate *StateManager* methods, so that the user interface reflects the real table state.
The *StateManager* implementation verifies that no illegal actions are done. For instance, if we try to drop a chopstick that was not taken, the program (the StateManager class) shows an error and aborts.

Versions 1 and 2: Asymmetry

Both versions 1 and 2 are based on asymmetry. To that end, there is a class named ***LefthandedPhilo***, corresponding to a philosopher that takes their chopsticks in the opposite order of that of Philo.

Exercise 1.1: Complete the class ***LefthandedPhilo*** and make the appropriate modifications to the class **PPhilo**, to solve both version 1 and version 2. Keep in mind that in order to make the code work in the correct way, the class *LefthandedPhilo* has to extend Philo.

NOTE.- Keep in mind that, independently from the order that the chopsticks are taken, the right chopstick has to be dropped before the left one, since this is a restriction imposed by the class StateManager (i.e., by the graphical interface).

Exercise 1.2: Check that deadlock can no longer occur by executing several times the program, in both the *Asymmetry (last/but last)* version and the *Asymmetry (even/odd)* version.

<u>Questions</u>
1) Which Coffman conditions are broken with the proposed solution, for versions 1 and 2?

Circular wait

2) Is it guaranteed that deadlocks will never appear?

Yes, because we broke a Coffman condition, and a deadlock cannot occur if some Coffman condition is broken

3) From the point of view of solving the deadlock problem, is it important that the *LefthandedPhilo* philosophers are placed in particular positions, or could they be in any position?

There could be in any position, the important thing is that they break the circular wait condition

Version 3: All or nothing

In version 3 of the deadlock solutions proposed in the table above, the mechanism to prevent deadlocks is based on assuring that “philosophers either take two chopsticks or, if any of them is taken by someone else, they do not take any chopstick”. In this case, philosophers must request chopsticks using the takeLR operation of the table. In this operation (to be implemented by students), the philosopher will either take both chopsticks at once or, if any chopstick has already been taken, wait without taking any.

Exercise 2: Modify the code as needed to give solution to version 3.

Questions:

1) Which Coffman conditions are broken with the proposed solution for version 3?

Hold and wait

2) Have you used the same *RegularTable* as in previous versions? Or have you used a different table? Why?

I have used RegularTable, after implementing `takeLR`

3) Have you used the same philosopher *Philo* as in version 1? Have you used the *LefthandedPhilo* philosopher? Have you implemented a new type of philosopher? Why?

I have used BothOrNonePhilo, which runst `takeLR` instead of `takeL` and `takeR`

Version 4: table capacity

Finally, version 4 proposes as a mechanism to avoid deadlocks that at most 4 philosophers can be sitting at the table. In order to do that, you should use methods enter/exit of the table, so then philosophers must request to enter/exit from the table in each iteration.

Exercise 3.1: Modify the code as needed to give solution to version 4.

Questions:

1) Which Coffman conditions are broken with the proposed solution for version 4?

Circular wait

2) Have you used the same *RegularTable* as in previous versions? Or have you used a different table? Why?

No, I have used `LimitedTable`, which limits the number of philosophers to 4 by using `wait()` and `notifyAll()`

3) Have you used the same philosopher *Philo* as in version 1? Have you implemented a new type of philosopher? Why?

I have implemented `LimitedPhilo`, which calls `table.enter()` before taking the forks, and `table.exit()` after eating and before starting to ponder