# Parallel Programming

Exercise 1, Concurrency – Dining Philosophers

03.03.2020

## 1 Task Description

The goal of this task to implement the dining philosophers program in a naive way, detect and analyze a deadlock and provide countermeasures in order to finally arrive at a valid solution for the problem. Be prepared to present your solution (in an informal setting, by showing the code on the beamer), explain the problem and answer questions, particularly those asked in this document.

For grading, put together a document which contains answers to the questions asked in subtask 2 and pack together your solution and the answer document.

In your programming language of choice, implement a program which reads in:

- the number of the philosophers at the table (in the description we refer to this integer as $n$)
- the maximal 'thinking time' of philosophers (in the description we refer to this integer as $thinkingTime$)
- the maximal 'eating time' of philosophers (in the description we refer to this integer as $eatingTime$)

The setup is as follows:

- The philosophers are identified by variables $p_0 \dots p_{n-1}$ (e.g. philosopher at index 0 is referred to as $p_0$).
- The philosophers sit around a table, $p_1$ is sitting between $p_0$ and $p_2$, while $p_{n-1}$ is sitting between $p_{n-2}$ and $p_0$.
- Between each of the philosophers is a fork denoted as $f_0 \dots f_{n-1}$. $f_0$ is between $p_{n-1}$ and $p_0$ whereas $f_{n-1}$ is between $p_{n-1}$ and $p_{n-2}$.

Each philosopher runs in its own thread. Be careful not have any race conditions on the shared resources (the forks). On the main thread wait for keyboard inputs (e.g. `readline`) which indicate to shutdown the process cooperatively. If a philosopher is busy while shutdown is required, let him finish his tasks and turn down the thread after that.

Each philosopher repeatedly performs the following operations:

- Think for a random period of time (in milliseconds) between 0 and $thinkingTime$
- Take the fork to the left (for $p_i$ this means $f_i$). If the fork is used at the moment wait until it gets available.
- Take the fork to the right, if not available wait (do not give back the left fork).
- Now the philosopher eats a random period of time (in milliseconds) between 0 and $eatingTime$.
- Finished eating, the philosopher puts back the forks in arbitrary order.

The following pseudocode sketches the procedure. Note that take is a blocking procedure and only returns if the specified fork is available. Additionally some information is printed.

```
int index = index of the philosopher to simulate
bool running = true
while(running)
{
     int t = random between 0 and thinkTime
     sleep(t)
     print('phil' + index + 'finished thinking')

     take fork[index]
     printf('phil' + index + 'took first fork: index)
     take fork [(index + 1) mod n]

     printf('phil' + index + 'took second fork: index)

     int t2 = random between 0 and eatingTime
     sleep(t)
     printf('phil' + index + ' is done eating')

     putBackForks(index,(index+1) mod n)

}
```

## 2 Subtask 1: A naive implementation

Implement the problem as sketched above. Test your program and find out that your program is trapped in a deadlock. Try to debug this situation and analyze it by using your debugger (if possible in your environment). Next, experiment with the input values and test how long it takes to arrive at a deadlock. Try different input values and think about possible reasons for different runtimes of the program until it ends up in a deadlock. Note that it is not required to detect the deadlock programmatically. Of course, in practice, deadlocks need to be prevented by using appropriate locking shemes. Still, debugging and analyzing deadlocks is a very useful tool. In the next subtask we will work out a solution for the deadlock problem.

## 3 Subtask 2: Deadlock prevention

Why does the deadlock occur? Answer the following questions:

- What are the necessary conditions for deadlocks (discussed in the lecture) [0.5 points]?
- Why does the initial solution lead to a deadlock (by looking at the deadlock conditions) [0.5 points]?

Prevent the deadlock by removing *Circular Wait* condition:

- Switch the order in which philosophers take the fork by using the following scheme: Odd philosophers start with the left fork, while even philosophers start with the right hand [6 points]. **Make sure to use concurrency primitives correctly!**
- Does this strategy resolve the deadlock and why [1 point]?
- Measure the time spent in waiting for fork and compare it to the total runtime [3 points].
- Can you think of other techniques for deadlock prevention?
- Make sure to always shutdown the program cooperatively and to always cleanup all allocated resources [2 points]

Make sure that your final solution compiles, **runs without race conditions or deadlock and performs a proper cooperative shutdown**. Use as many looks needed but as little locks as possible. Excessive placement of useless locks is considered as a mistake (e.g. each method is marked as synchronized for no reason). Please note that this exercise requires a real non-trivial solution to the problem. e.g. one single lock which runs all operations sequentially is not what we want here ;). Pack the final solution into your submission and upload it to moodle.

# 4 Files to submit

Submit your solution as zip file including the following files:

- `philosophers.{c,cpp,cs,fs}` Final implementation using the deadlock resolution strategy defined in Subtask 2.
- `report.pdf` Solutions to the answers

Please do not add .git or temporary build files such as obj etc in the zip! Upload your final solution till 11.3, 1800.