Homework 4

Ludvig Larsson

1 Implementation

For homework 4 I decided to implement dining in hell, where 5 Devils feed each other since no one can feed themselves. The implementation is done in Java.

I stared by defining how I wanted to implement, start and handle the threads. There is a `class Devil` which can be instantiated to act as a person around the table. This class implements the `runnable` interface providing parallelism and the threads are created in my main class, `Inferno`. Threads are created by instantiating the Devil class and passing it to the `Thread` constructor. Lastly, `Thread.run()` is invoked and all threads are running concurrently as desired.

I did not want to complicate the solution too much in the monitor, which is holding all building blocks for the table interactions as its state in form of fields. There is only one the implicit conditional variable used, and a thread will sleep on it unconditionally when it is allowed eat and will be awaken by another thread currently serving. There is a FIFO implemented `ArrayList` holding threads that wait until they can be fed that will correspond to the conditional variable waiting queue to show which threads are being blocked at the moment. To ensure fairness (starvation) I have made a circular dependency between the threads. Each person with `id = x` is instructed to only feed the person with `id = (x+1 % 5) + 1` (id's are 1-5). This allows servant threads to simply check if its servant for the first person in waiting queue and if so, feed it by using `notify()`. Threads are terminated once all threads have signaled to the monitor that they are done eating, in that case, when all threads are serving, an exception is thrown from the monitor that breaks the run loop.

To solve the deadlock situation and fairness there are two key implementation details to consider. To avoid deadlock there is a limit which allows at most half of the people to wait for food simultaneously. Threads have a private flag which decides its next action, eat or serve, that is distributed half and half initially. The flag is altered after each attempt to eat or serve, unless a thread is done eating so it always serves. Deadlock avoidance in this scenario is ensured based on the limitation of people waiting for food (all can't wait on cond var) and there is also a check that e.g. `id:3` doesn't add itself to the wait queue if `id:4` is already waiting. Without this deadlock could occur, since only the first person in the queue is notified, `id:4`, and that becomes impossible if `id:3` is also waiting.

2 Discussion

I decided to not implement additional conditional variables, considering the complexity it would bring and if I'm not completely wrong, there will be low busy-waiting since threads have a specific thread to serve. It could be a linear busy-waiting time (time-slices) for all threads not allowed to serve at the very moment, but since each thread is blocked sleeping between each attempt to serve or eat, the CPU loss is reduced significantly. There is also concurrency involved as the threads do not execute in the same order every run, execution traces are altered, as can be seen in the program output (order threads got to eat in). One thing to mention is that there are not an excessive amount of execution traces since there are a few constraints on feeding implemented that partially reduce the concurrency, but each thread feeds equal number of times as another which is the fairness I strived for! It is also not an over constrained monitor considering there are no loops in the synchronized methods and there is a blocking wait queue on the conditional variable.