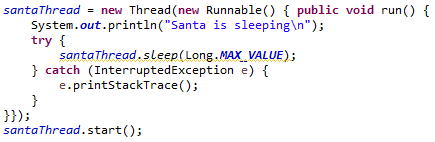
**Report of MiniProject**

Our group had to solve a problem and the solution had to be deadlock free. We had three options to choose of. Our group chose the Santa Claus problem.

In this report we are going to explain how our code is working, why it is deadlock free.

A deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a [lock](https://en.wikipedia.org/wiki/Lock_(computer_science)). Deadlocks are a common problem in [multiprocessing](https://en.wikipedia.org/wiki/Multiprocessing) systems, and [parallel computing](https://en.wikipedia.org/wiki/Parallel_computing) where software and hardware locks are used to arbitrate shared resources and implement [process synchronization](https://en.wikipedia.org/wiki/Synchronization_(computer_science)).

**Code**

****

Our program’s main thread is the *santaThread*, first of all, it goes to sleep immediately. We are starting this thread and put back to sleep if it is necessary, for a few times, it is overrided to avoid creating new threads every time, and be able to work with the same „Santa Claus”.

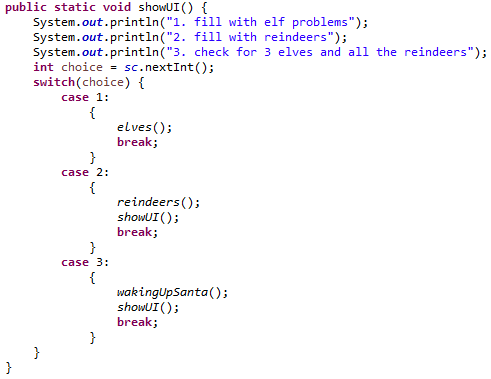
On the previous slice of code, our method of implementing threads can be seen.

We also implemented a very basic and simple command-line UI. The principle of the UI is to make the program controllable and more transparent for ourselves. The first option is filling the system with elves, the second option is filling the system with reindeers, and the third option is for waking up Santa Claus.

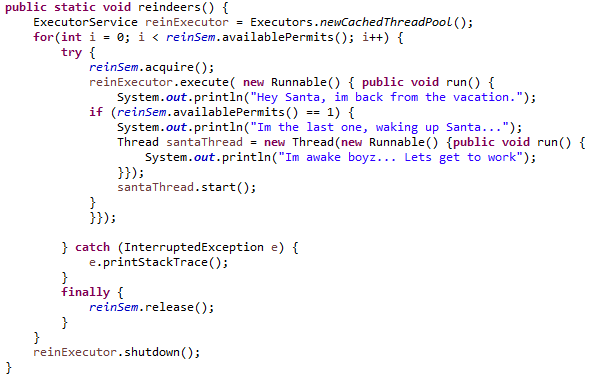
We used semaphores in our code.

A semaphore is a [variable](https://en.wikipedia.org/wiki/Variable_(programming)) or [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type) used to control access to a common resource by multiple [processes](https://en.wikipedia.org/wiki/Process_(computing)) and avoid [critical section](https://en.wikipedia.org/wiki/Critical_section) problems in a [concurrent](https://en.wikipedia.org/wiki/Concurrent_computing) system such as a [multitasking](https://en.wikipedia.org/wiki/Computer_multitasking) operating system.





Code of UI



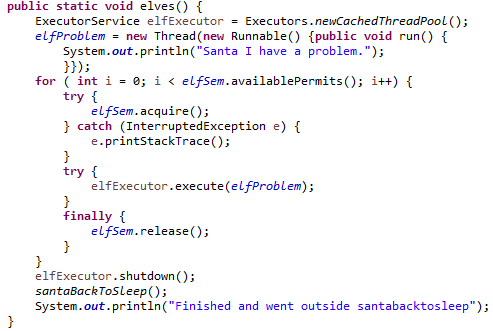
The last slice of code contains the implementation of the reindeers() function.

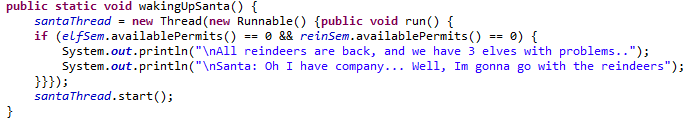
There, we are creating a new *ExecutorService* called *reinExecutor*. *ExecutorService* provides us a pool of threads, and we are able to assign tasks to them. In that case, we are using *Cached Thread Pools*.

With the help of *Cached Thread Pools*, **if there is an idle thread waiting on the queue, then the task producer hands off the task to that thread. Otherwise, the executor creates a new thread to handle that task.** It uses the *SynchronousQueue.*

With a for loop, we are filling the system up, with reindeers. It runs until the number of the available permits of *reinSem*. That means only 9 reindeers can exist in the system. In the *try*, a reindeer is going to acquire a permission from the *reinSem* semaphore. After that, executing a new *Runnable*. If the *reinSem* semaphore’s available permit is 1, that means the last reindeer is back, then the last reindeer is going to wake up Santa. Finally the current *reinSem*, is released.

The same goes for elves.

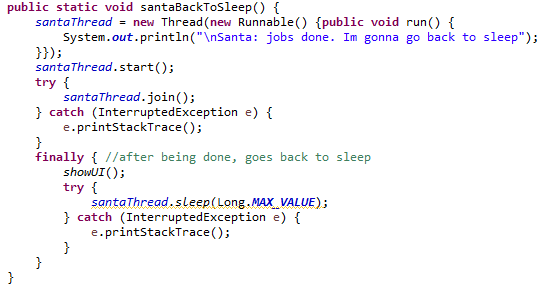




The *wakingUpSanta* function has a problem here. As it can be seen, we wanted to solve the waking up Santa problem, with an *if* statement, where we are checking the available permits. Theoretically, it should work, but in the *elves()* and the *reindeers()* functions, the semaphore permissions are going to be released every time a for loop is going to run. Hence, the *availablePermit()* function’s return value is going to be the same as the predefined value, in our case 9 and 3.

Unfortunetaly we realised that late, and we ran out of time.

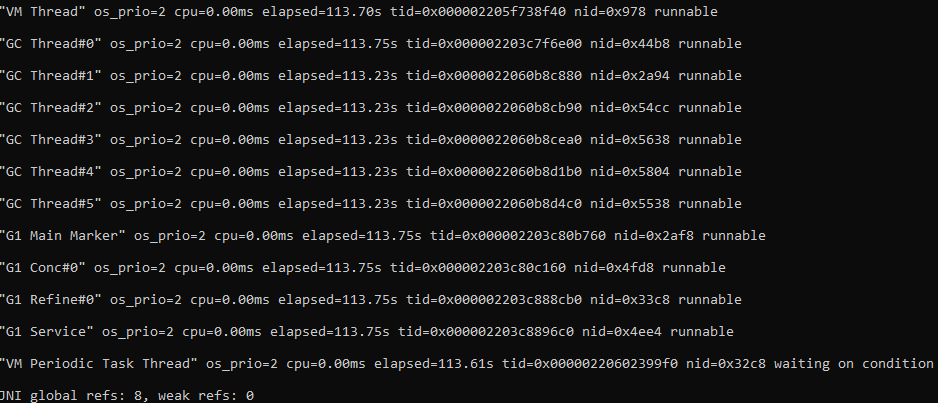
After Santa is finished with solving the elves problems, he goes back to sleep. That’s why we are invoking the *santaBackToSleep()* function.



First, the *santaThread* starts. In the *try*, *santaThread.join()* waits until the thread has dead. If *santaThread* has died already, finally, *santaThread* is goind to sleep again.

**Conclusion**

We tried to solve the problem to the best of our knowledge. We only used the mandatory amount of threads, and tried to run only one thread at the time. If there were more than one thread running at the same time, it would not be a problem in our case, because were not using a variable, or an array to write and read from it, the threads were just printing some text to the console. Furthermore, we ran jstack to analyze deadlocks in the system.

The output of jstack

Even though, we were not using semaphores to prevent racing conditions, we thought is a good and easy idea to implement reindeers and elves. Sad to say, it is not working at this state.

Since everything is runnable, and no deadlock had been found according to *jstack*, as we best know, that proves there are no deadlocks in the system.

https://en.wikipedia.org/wiki/Deadlock

<https://www.baeldung.com/java-executors-cached-fixed-threadpool>

<https://en.wikipedia.org/wiki/Semaphore_(programming)>

Daniel Vechiu

Kristof Soos

Fatih Ozkavaf

Laurentiu Bogdan

Sandra-Georgiana Ene