In the name of God

Sharif University of Technology

School of Computer Science

Operating Systems

Project

Dr. Zarei

Taha Entesari

95101117

In this project we aimed to simulate a natural selection experiment among a group of species. The goal of this project was to work with threads and processes and utilize the tools we had learned in the course to overcome issues such as race conditions and deadlocks. Alongside this report I have included the Java codes that I have written. At first I aimed to solve the threaded version of the project as it seems (and is) easier than the multi-process version.

1 Thread Implementation

To run this implementation, run the Main.java file in the ThreadImplementation package with your desired options (you must input your options into the code).
The code has a general controller and a table which both control the life cycle. The controller specifies the current state of the environment and the species base their actions on this state. After every t rounds the controller will ask the user whether or not to continue. The environment has 4 main states, namely intraTurn, intraTurnEnd, turnEnd and turnStart. The description of each state is as follows:

- intraTurn: In this stage every creature is free to move as many times as it desires. The ability to move is simulated with a random boolean. If the creature wants to move, it will choose at random one of the available adjacent cells (at most 8). At every round this stage takes up about 1 second and in this interval the creatures move at will.
- intraTurnEnd: This stage is added so that if any creature had requested to move and was in the wait list to acquire the lock (semaphore) to the table, could complete its actions.
- turnEnd: In this stage at first all overpopulated cells will sacrifice some creatures so that there is enough capacity for the rest and after that, adjacent cells will fight for survival (more on this later).
- turnStart: In this stage every creature that is left alive and is able to reproduce, will reproduce and will create one extra creature of the same kind in that cell.

1.1 intraTurn and intraTurnEnd

In my implementation the existence of the extra stage, *intraTurnEnd*, is crucial and without it there would be inconsistencies. Another way to implement this would be to preempt all threads that have requested to move but I choose the first method.
As mentioned, in the *intraTurn* stage every creature flips a coin and if successful, will want to move and to do so, it will first request for the table. The process of requesting for the table is implemented using a one lock semaphore and thus at every moment only a single process will have access to the table (to both read and write) and thus only a single process can move at once and others that have requested to do so, will be locked and must wait for the previous thread to finish and release the table. One could argue that reading the table might not need such a hard constraint and that threads should be free to read the table at any time but this causes an issue. The issue is that a creature might decide to move to an adjacent cell and choose a cell to move into, but after it has chosen to do so, a different species moves into that cell and thus this move cannot be made and thus to prevent such problems, I have issued a hard constraint on accessing the table.

1.2 turnEnd

The turnEnd stage is handled by the table (and not the creature itself) since this stage requires knowledge of the state of the whole table. The overpopulation death is obvious and does not need explanation. I will try to explain the death of a creature caused by its neighbours. As

mentioned in the project description, there are two remarks regarding this kind of death. The first remark forces that at first the weak creatures must die and the second remark imposes a weird rule on which adjacent cells can attack a given cell. These two remarks are acquired at the start of the program and are independent and can be turned on and off separately.

1.2.1 Remark 1

To handle this, I keep an array of the locations of each kind of species and thus at turnEnd when we are asked to kill off some creatures, I will start to read this array from the beginning, starting from the weakest creatures, and check if any cell must be executed completely.
The opposite of this remark is not clear exactly. That is, it is not obvious what would happen if we were to neglect this remark and thus I have implemented the following strategy:
The code starts to read the table from the first cell and traverses the table row by row and in each cell, checks if the inhabitants of that cell must be executed and does so if required and restarts this process (this step is necessary).

1.2.2 Remark 2

To handle this rule, I first get only the diagonal cell relevant to a cell and check if any of the creatures in those cells fit the rule. If so, I remove this adjacent cell from the list of cells that will be used to check if the initial cell must be killed of. The remainder of these diagonal cells are added to the non-diagonal adjacent cells and these cells are passed to the function that checks if a cell must be executed with respect to its adjacent cells.

1.3 turnStart

This stage is relatively simple. In this stage each creature that is alive starts a new thread (process) resembling a creature of that species. It is possible that a cell might be overpopulated after this stage but with respect to the rules of the project, that is not an issue.

2 Process Implementation

Just like the *Thread Implementation*, to run this program, run the Main.java file in the ProcessImplementation package and change the options as desired.
The stages that were mentioned in the *Thread Implementation* are exact copies here and thus I will not repeat them. The main difference of these two implementations is the communication. In the thread implementation each thread could easily access the controller and the table and read data from it but in this implementation that is not an option. For the communication needed between the processes I have used two methods. The first one which handles the main communication is using the input/output stream of the process that has been started in the table. Using these streams the processes send their requests to move, reproduce and to get the current stage of the turn. I have also used a second method of communication and that is communication via writing to a file.

2.1 Process Start

The processes are started in the table object and the table then keeps the input and output stream of the process for further use. For every process that is started a *communicator* thread is also started that is constantly listening to the process and gives such requests to the table object

2.2 Communication

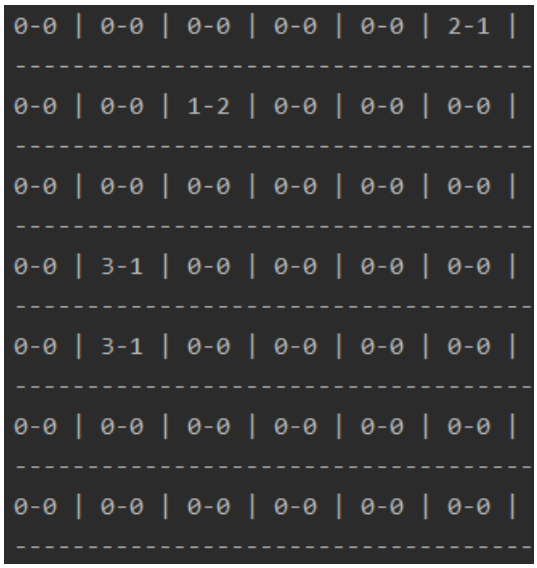
As mentioned, there are two methods of communication. The first is communicating via a java stream. Using these streams the process sends requests such as requesting to acquire the table or release it (acquire/release the semaphore), to move the creature or to tell the table that the creature has reproduced. After such a request has been sent to the table, the creature process will wait for the response of the table and the table will provide the required results to the creature process using its output stream.
The second method of communication is used for when the table requests some data from the creature. These data are limited to the state of the creature, i.e. whether or not the creature has finished its *turnEnd*, *turnStart* or *intraTurnEnd*. One more data is also communicated using this method and that is whether or not the creature has been killed (the creature constantly checks a file to see if it has been killed or not).

2.3 Consistency

Even thought this implementation is called *Process Implementation*, I have used threads for keeping the consistency of the table data. Every request that a creature makes is handled by a single constant thread and thus just as before, consistency holds since the previous *Thread Implementation* had consistency.

3 Results

The following figure shows a sample result of the table that the program produces:



The first number of each cell shows the species type and the second number specifies the number of creatures in that cell. The above picture is the result of running the program with the sample options provided in the project description for a single round. As it can bee seen, after one round two creatures have been killed (one of type 1 and one of type 2) and the remaining creature of type 1 has reproduced. The program outputs tables like the above after each important stage that can be used to check the validity of the code.
It seems that under all circumstances (all combinations of rule 1 and 2 being active and inactive), the stronger creatures always survive and kill all the other creatures. This result seems logical but the reason that I am a bit skeptical is that the project description had mentioned that "we enforce these two rules so that the stronger creatures would survive". A reason why this might be happening is that perhaps the designer of the project had something else in mind about what it meant for rule 1 to be inactive (I have described what I have implemented for this case in 1.2.1)

Notes on running the code

I have coded this project using the Jetbrains IntelliJ IDE and have exported the project using the IDE. The reason I am telling this is that it is necessary for the project files to stay where they are if the code is to be run without any problems and to run the project it is enough to load this project in IntelliJ.
If you intend to run this project in some other way, keep in mind that two things (as far as I know at the moment) are needed for the code to run. One is the processFiles folder at the base folder of the project and the other, is the path to the SingleSpeciesProcess.java file. The second one is needed to compile the source and run it when the table object wants to start a new creature process (it is used in two places. One in Main.java and the other in Table.java both in the ProcessImplementation package).