Operating systems 2 project documentation

Project description:

Solving the N-Queens problem using four threads involves parallelizing the task of finding valid configurations of N queens on an N×N chessboard. The goal is to distribute the workload across multiple threads to potentially speed up the solution-finding process. Here's a description of the process:

Divide the chessboard: The first step is to divide the N×N chessboard into four equal sections. Each section will be assigned to a separate thread for independent processing.

Thread initialization: Initialize four threads, each responsible for finding valid queen configurations in its assigned section of the chessboard.

Thread execution: Each thread will independently execute the solving algorithm to find valid configurations of queens in its section. The algorithm follows the steps described in the sequential solution, but each thread operates on its designated portion of the chessboard.

Parallel search: Each thread searches for valid queen positions in its section concurrently with the other threads. They explore different possibilities simultaneously, reducing the overall search time.

Thread synchronization: Since each thread operates independently, there needs to be a mechanism to synchronize their progress and ensure that they do not duplicate work or miss potential solutions. Synchronization can be achieved using synchronization primitives like locks, barriers, or message passing between the threads.

Solution collection: As each thread finds valid configurations, it adds them to a shared data structure or communicates them to a central thread responsible for collecting the solutions.

Thread termination: Once a thread completes searching its assigned section, it terminates its execution.

Combine solutions: After all threads have finished their execution, the solutions collected by each thread are combined to obtain the final list of valid configurations.

What we have actually did:

In the N-Queens problem, the main task is to find a valid configuration of N queens on an N×N chessboard, where no two queens can attack each other. To solve this problem, we typically follow these steps:

Initialize the chessboard: Create an empty N×N chessboard, represented as a 2D array or a data structure.

Start with the first column: Begin with the leftmost column (column 0) and proceed to the next column (column 1).

Place a queen in a valid position: Iterate through each row in the current column and try placing a queen in each row. Check if the queen's position is valid, considering the constraints (no two queens should be able to attack each other). If a valid position is found, mark it as a queen's position on the chessboard.

Move to the next column: Once a queen is successfully placed in the current column, move to the next column (column 2) and repeat steps 3 and 4 recursively.

5.Continue placing queens and backtracking: Repeat steps 3 and 4 until all N queens are placed on the chessboard. If at any point, it becomes impossible to place a queen in a valid position, backtrack to the previous column and try a different position. Backtracking involves undoing the previous queen placement and exploring alternative possibilities.

Find all solutions: Keep track of all valid configurations where all N queens are successfully placed on the chessboard. This can be achieved by maintaining a list or collection of valid solutions.

Continue exploring: After finding a solution or exhausting all possibilities, continue exploring other valid configurations by backtracking further or exploring different paths.

Return the solutions: Once all valid configurations have been found, return the list of solutions or display them as desired.

Team members roles:

Board: Peter shaker, mina Milad.

Pieces: Rowyda Raafat, Mary Yousry

Testing the code: Mariam Eid, Youstina Gamal.

Documentation: Rahma Ehab.

Gui: Peter shaker.

n-queen-t file:

The n-queenst class provides a multithreaded solution for the N-Queens problem. It utilizes threads to explore different regions of the chessboard concurrently in an attempt to find a valid placement of N queens such that no two queens threaten each other. **Class Members**

- **ID (String):**
  - **Description: Unique identifier for the thread.**
  - **Access Modifier: Private ○ Purpose: Distinguishes each thread instance.**
- **N (int):**
  - **Description: Size of the chessboard (number of queens).**
  - **Access Modifier: Static ○ Purpose: Defines the dimensions of the chessboard.**
- **R (int):**
  - **Description: Starting row for the thread.**
  - **Access Modifier: Private ○ Purpose: Specifies the initial row for the thread's exploration.**
- **C (int):**
  - **Description: Starting column for the thread.**

- Access Modifier: Private ○ Purpose: Specifies the initial column for the thread's exploration.

- key (Reentrant Lock):
  - Description: Lock for synchronization.
  - Access Modifier: Public ○ Purpose: Ensures exclusive access to critical sections of the code.
- Answer Found (Boolean):
  - Description: Volatile Boolean flag.
  - Access Modifier: Public ○ Purpose: Indicates whether a valid solution has been found.

Gui Update file:

The `Gui Update` class appears to be part of a graphical user interface (GUI) implementation for the N-Queens problem solution. This class extends the `SwingWorker` class, which is designed for background processing and GUI updates. Here's an explanation of the key components and functionality:

Class Members:

- ID (String): ○ Stores a unique identifier for the thread associated with this instance.
- N (int): ○ Represents the size of the chessboard (number of queens).
- i (int): ○ Represents the current row index.
- col (int):
  - Represents the current column index.
- done (Boolean): ○ Indicates whether the GUI update is complete.
- board (int[][]): ○ Represents the current state of the chessboard.
- Operation (int):
  - Specifies the type of operation (0 for placement, 1 for removal).

`doInBackground` Method:

Description:

- This method is called in the background thread to perform GUI updates.
- Switches between two operations based on the value of `Operation`:
  - Operation 0 (Queen Placement):
    - Calls `nqueenT.Safe` method to check if placing a queen is safe.

- If safe, updates the GUI by adding an image (specified by `ID`) to the corresponding `Panel`.
  - Operation 1 (Queen Removal):
    - Iterates through the `Panel` components and removes the label with the specified `ID`.

`done` Method:

Description:

- Executed when the background task is done.
- Currently, it does nothing. You might add additional actions after the background task completion.

`returnDone` Method:

- Description:
  - Returns the value of the `done` flag, indicating whether the GUI update is complete.

Additional Explanation:

- Image Selection:
  - The `switch` statement selects an image (`Image im`) based on the `ID` (thread identifier).

- The images are presumably loaded from `MainSup` class (not provided) using identifiers like "White," "Green," etc.

- **GUI Updates:**
  - GUI updates involve creating a new `JLabel` with the selected image and adding it to the corresponding `Panel`.
  - `Thread.sleep` is used to introduce delays for visual effects.