Operating Systems

Homework Assignment #5

Due 19.5.2016, 23:50

In this assignment you will implement Conway's Game of Life using threading and synchronization.

The Game of Life is a cellular automation, represented by a matrix of bits. Each cell in the matrix has two possible states: *alive* or *dead*. The automation proceeds in generations, represented by steps. In each step, every cell interacts with its eight neighbors to update its state. If it's alive, it either dies or lives on. If it's dead, it either remains dead or becomes alive.

At each step, the following transitions occur:

- Any **live** cell with fewer than two **live** neighbors dies, as if caused by under-population.
- Any **live** cell with more than three **live** neighbors dies, as if by overcrowding.
- Any **dead** cell with exactly three live neighbors becomes a live cell, as if by reproduction.
- Otherwise, the cell's state remains the same.

Part 1

Write a program `gol.o` implementing Conway's Game of Life. The program will:

- The program receives two parameters: file name, and steps (an integer).
- Create an **`int`** matrix according to the file (a *square* matrix):
  - Assume file size is exactly a power of four. Each byte in the file represents a single cell in the matrix. A byte value of 0 is 0, any other byte value is 1.
  - The matrix should be a **global** variable.
  - Do not define the matrix regularly - you will probably receive a segmentation fault.
  - Instead, dynamically allocate array of arrays, e.g.:
    - `mat = (int**) malloc(sizeof(int*) * SIZE);`
    - `for (i = 0; i < SIZE; ++i) {`
    - `mat[i] = (int*) malloc(sizeof(int) * SIZE);`
  - Fill the matrix with **binary** values from the file as described above.
  - **Do not** try to improve performance, etc. Define, allocate and load the matrix exactly as stated.
- Execute steps of the game (according to the 2nd parameter).
  - In each step, update each cell of the matrix to its new state.
  - Measure the time (in milliseconds) the entire execution took, i.e., all steps of the automation.
  - **Do not** measure the allocation, loading, cleanup, etc. – only the automation itself.
- Make sure your program works correctly.
  - Use a small matrix, and a small number of steps.
  - Execute various sample matrices check the results.
  - Use google to find various simulators you can experiment with to validate your results.
  - Results should not be mixed! All cells update at once, so the new state of a certain cell should have **no effect** on the state of other cells. Cells updated to generation i are updated according to the state of cells in generation $i-1$. You may use a helper matrix for this, but **do not** copy matrices in each step (i.e., use pointers).

Part 2

Write a program `pgol.o` extending part 1 to use tasks and threading. The program will:

- Use tasks to execute each step.
  - A task is an object consisting of 4 parameters: (x, y, dx, dy). Meaning, a task is simply some part of the global matrix.
  - A task can be executed. To execute a task, the following rules apply:
    - If $dx == 1$ and $dy == 1$, the task represents a single cell – update the cell (x, y) according to its neighbors (per the rules given above).
    - Otherwise, split the task into 4 separate **equal** tasks, different parts of the original.
- Each step of the automation starts by placing a single task in a queue - $(0,0, SIZE, SIZE)$
  - To proceed, dequeue a task from the queue and execute it.
  - If the task was split into 4 other tasks, enqueue them into the queue.
  - Proceed until the queue is empty and no further tasks will be enqueued to it – the step is done.
- Use mutex and condition variables to implement the global queue of tasks.
  - The queue should support multiple enqueuers and multiple dequeuers.
  - No busy-wait! Use condition variables to avoid spinning, etc.
  - Usually multiple items will be enqueued at once (multiple tasks). Thus, for the enqueue operation, the queue should allow insertion of multiple items with a **single** lock/unlock section. Do not forget to call the condition variable appropriately.
  - Make sure the lock is held for as little time as possible.
- Receive another (3rd) argument – the number of threads to spawn (in addition to the main thread).
  - All created threads should be **identical**.
  - Each thread will repeatedly dequeue a task from the global queue, and execute it.
  - When the execution is done, the thread dequeues the next task, etc.
  - Threads never finish – when the main thread finishes, all threads will exit along with it.
- Use the main thread to control the execution of steps.
  - To start each step, the main thread enqueues a starting task, as described above.
  - The main thread then waits until all tasks are done.
  - Once all tasks are done, the main thread continues executing the next step by placing a new starting task in the queue.
  - **Do not** place tasks before the previous step is finished, or the automation will be corrupted!
  - To determine if all tasks are done:
    - Read about $\_\_sync\_fetch\_and\_add$.
    - Whenever a thread finishes executing a single cell task, atomically increment a shared (global) counter. **Do not lock!**
    - When that counter reaches $SIZE * SIZE$, the step is finished (reset counter).
    - Don't busy-wait! The thread that finishes last (check the result of the fetch & add operation) should signal the main thread with a condition variable, which can then reset the counter and prepare the next step.
  - Implement the main thread efficiently – does it need to lock the queue?
- Measure the time (in milliseconds) the entire execution took.
  - As before, measure only the execution itself (just before inserting the first task to the queue by the main thread, until finishing with the last step).

Guidelines

- Execute all code for time measurement on **nova**.
- Use only the system calls learned in class.
- Do not forget to check the return values of all functions.
- **Do not forget** to comment out any auxiliary code (debug prints, etc.) in the final submission!
    - **Submit:** Two C files: `gol.c`, `pgol.c`.
        - A simple excel graph comparing the measurements of both parts. Execute part 2 for 1, 2, 4, and 8 threads for a total of 5 bars. Each should be the average of 5 executions.
        - A text file (*answers.txt*) containing:
            - A possible explanation to the results you've seen (hint: nova has 4 CPUs).
            - A suggestion for improving the runtime of part 2 (in a different way, using the tools learned in class).