CPEN 333 Final Project : Part I Alternative (Multithreaded Game)

Group G6 : Muntakim Rahman and Tomaz Zlindra

December 6th, 2024

1 Redesign

Our main redesign of the program structure was to change the model of Inter-Process Communication (IPC) from message passing to shared-memory. This task was reduced to `producer-consumer` problems for each resource, in which the `producer` must wait for the `consumer` to yield the resource and vice-versa, in order for the program to be thread-safe.

Having removed the use of the `Queue`, `QueueHandler` classes, we aimed to achieve synchronization with `semaphores` for the *4* shared resources (i.e. ”`game_over`”, ”`score`”, ”`prey`”, ”`move`”) to indicate when they have been produced. All, except for `"game_over"` are protected via critical section using a `dict` of `mutexes`.

Please see the comment at the top of the `part1_alternative.py` file for a description of the implementation. This file will describe the benefits, disadvantages, and challenges of this.

2 Benefits

Since we were able to eliminate the need for *2* classes, this simplified the relationships between classes, as shown in Figure 1.

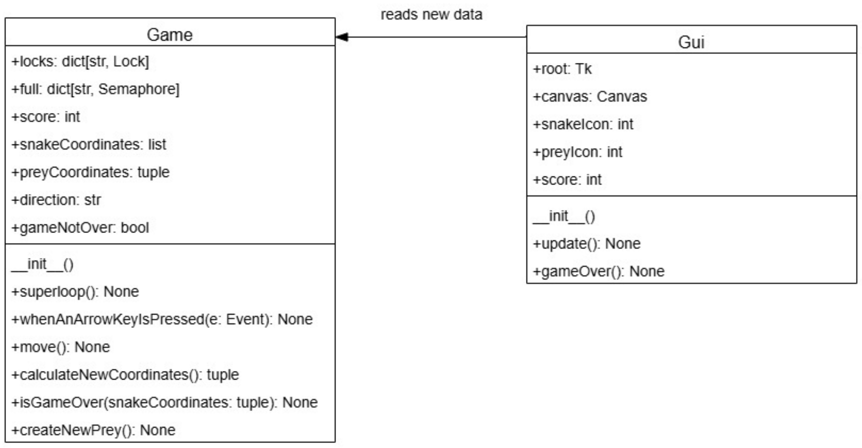

Figure 1: Part 1 Alternative - UML Relationship

2.1 Reader-Writer Abstraction

If we abstract this to be a `reader-writer` synchronization problem, a single writer / many readers can access access the resource at all times. We need to ensure there isn't a race condition between them. This is a problem we have addressed before (e.g. Lab 5).

We have designed this program to have the following constraints:

- $\{\text{Readers}\} \cup \{0,1\}$ where $\{\text{Writers}\} \cup \{0\}$
- $\{\text{Writers}\} \cup \{0,1\}$ where $\{\text{Readers}\} \cup \{0\}$
- $\{\text{Readers}\} \cup \{0\}$ where $\{\text{Writers}\} \cup \{1\}$

2.1.1 Producer Behaviour

The `game` instance acts as both a reader and a writer in different places. It reads the current value(s) of the shared resource and copies it for processing. It also writes the new data to the shared resource. When it is behaving as a `reader`, we don't need to acquire the `mutex` or release the `semaphore`. Race conditions become a possibility for cases where the shared resources are being modified.

3 Disadvantages

3.1 Additional Complexity

We have to ensure that each of the shared resources are accessed effectively. More locks means that there are more critical sections to manage among threads. We needed to ensure that critical sections were kept short and to the point to be able to navigate this added complexity.

In addition, we have the `full` semaphores to indicate when a new value is available. This condition must be evaluated by the `consumer` once it successfully acquires a `mutex`.

3.2 Storage

Along with the `mutexes` and `semaphores`, we also need to store the `game.preyCoordinates` data field for the `gui` instance. This wasn't necessary with the `queue` implementation.

4 Challenges

We initially spawned two threads for the `game.superloop()` and `gui.update()` methods. This was highly problematic since the program functioned as intended for initial gameplay. Critical sections seemed to be thread-safe, both theoretically and in practice. However, through extensive testing we observed this consistently caused the program to crash when the snake grew at a score $\sim$ *25-26*.

We realized why the original program design had the `QueueHandler.queueHandler()` method schedule itself with the `Tk.after(100)` method. From referring to the `Python` documentation, we realized that `Tkinter` widget updates must be done by the main thread.

(See The Python Software Foundation. (n.d.). Tkinter - Python interface to TCL/TK. Python Documentation. https://docs.python.org/3/library/tkinter.html#threading-model)

4.1 Solution

Instead of having the `gui.update()` method execute loop with spinlocks, we wrote it to be a conditional statement which schedules itself using the `Tk.after(100)` method. We implemented this with non-blocking acquires as well, such that the inability to enter a critical section wouldn't block other widget updates from occuring in the meantime.