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 accessed via 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.

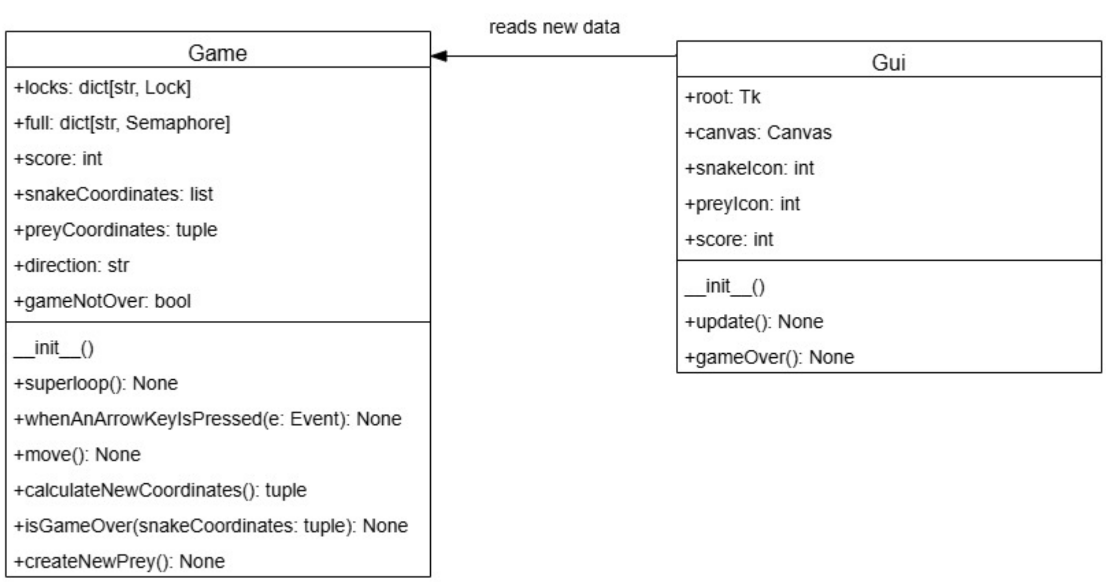

Figure 1: Part 1 Alternative - UML Relationship

3 Disadvantages

A single producer / consumer is accessing the resource at all times and 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 implemented this rather simply, and aren't treating them as `reader-writer` for our synchronization problem (i.e. $\{\text{Producers, Consumers}\} \cup \{0, 1\}$).

3.1 Complexity

That said, there is added complexity from the original design. We now have to ensure that each of the *4* shared resources are accessed effectively. More locks means that there are more critical sections to manage among threads. This in turn, leads to more potential complexity. We needed to ensure that critical sections were kept short and to the point. *If we abstract this as a `reader-writer` problem momentarily : `readers` access the current data and use it for their needs, `writers` update the new data.*
Having the `full` semaphores to indicate when a new value is available adds another layer of complexity. This condition must be evaluated by the `consumer` once it successfully acquires a lock.

Note that 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

3.2 Overhead

We also need to store the `game.preyCoordinates` data field for the `gui` instance to access for its `Tkinter` widget, which wasn't necessary with the `queue` implementation. Here, we are also updating the coordinates every *100 ms*. This is unnecessary as it should only be done upon change. The `Queue.get_nowait()` method had addressed this issue.

3.2.1 Future Improvements

To improve this solution, we should treat this as a `reader-writer` synchronization problem in which multiple `readers` can be present at any given time. This will likely improve performance and allow perform Improvements such that it is near identical to the original implementation with the `Queue`.

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 *~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 conditionally loop in a spinlock, we wrote it to be a conditional statement which schedules itself with the `Tk.after(100)` method. We implemented this with non-blocking acquires as well, such that widget updates wouldn't trap the thread in a spinlock (i.e. while waiting) and block other widget updates from occuring in the meantime.