Práctica 4

–

Arquitectura de memoria compartida

Nombre: David López Pereira

Índice de Contenidos

Índice de Figuras

1. Introducción

El **objetivo principal** de esta práctica es la realización del juego de la sabana africana, donde existen varios animales que compiten por ver quién es el ganador. Para la realización de esta práctica se utilizará el lenguaje de programación Python y cada animal del juego será un hilo, por lo que deberemos controlar las secciones criticas cuando se accedan a variables comunes entre hilos.

Para entender mejor los términos de hilo y sección critica, primero definiré los conceptos.

Un **hilo** es un subproceso que comparte espacio de memoria con otros como él y **la exclusión mutua** es la actividad que realiza el sistema operativo para evitar que dos o más procesos accedan al mismo tiempo a un espacio de memoria compartida denominado **sección critica**.

En el siguiente punto explicare la abstracción seguida para implementar el juego, mediante un diagrama de clases, donde explicare cada atributo en profundidad y también explicare cada requisito del juego, así como la manera que he tenido de solucionarlo.

2. Explicación del juego

2.1. Abstracción del juego

Como se puede ver en la Figura 1 el diagrama de clases del juego consiste en 8 clases, aunque solo me centrare en 7, ya que la clase **threading** es una clase ya implementada en el sistema que permite la obtención de estructuras como los **hilos** y los **locks,** estos últimos permiten la exclusión mutua en el sistema y los he nombrado mutex en los atributos donde los utilizo.

La primera clase que voy a tratar es la clase **Animal**, donde se guardan los atributos característicos de los animales como son la **especie**, el **número de la manada**, la **posición en filas y columnas**, el **estado**, si se encuentra vivo (True) o muerto (False), y la **velocidad**, que dependerá del número de la manada, con estos atributos podremos crear los distintos animales en la simulación.

La clase **Manada** es una **lista de animales** donde se añadirán tantos animales como número de integrantes hayamos definido en el constructor de la clase, especificando el numero de la manada en cada animal creado, además se debe pasar el estado del juego para poder crear los animales correctamente, en esta clase se encuentra otro atributo importante para el juego como es la **puntuación de cada manada**, inicializada a 0 y que ira variando a medida que los miembros de la manada realicen capturas, con este atributos decidiremos si se acaba la simulación o continua, este atributo contara con un **mutex** para realizar la correcta exclusión mutua del atributo.

La clase **Casilla** es de lo que este formado el tablero del juego por lo que sus atributos principales son un **animal**, un **mutex** para poder bloquear o liberar la casilla y un **booleano** que dirá si un animal está ocupando la posición o no.

Una de las clases más importantes es la clase **Juego**, esta clase tiene diversos atributos de distintos tipos, como son el **número de casillas** necesario para después poder crear una matriz de Casillas con un tamaño de numCasillas x numCasillas llamado **tablero**, las distintas **manadas de los animales** presentes en el juego, un booleano llamado **ganador** que permitirá saber si el juego sigue o no, la **manadaGanadora** para poder escribir por pantalla el resultado de la misma y un **mutex** para controlar el acceso a la variable ganador y otro para **controlar la entrada y salida** con el objetivo de que los mensajes escritos por pantalla salgan ordenados y no se amontonen unos encima de otros.

La clase **Cebra**, la clase **León** y la clase **Hiena** son muy similares entre si y son las más importantes del juego, ya que sin ellas no se podría jugar, para empezar todas estas clases presentan una herencia múltiple, heredan tanto de la clase **Animal** como de la clase **Threading.Thread**, además estas clases necesitan el atributo **juego** pasado por parámetro para poder controlar el atributo ganador y terminar su ejecución. La única diferencia notoria entre la clase de los animales es que las cebras no pueden cazar por lo que solo presentan el método run, mientras que las hienas y los leones si pueden.

Para terminar la abstracción, el juego contara con una clase main donde se pedirán los valores de numero de casillas iniciales y se comenzara el juego mediante el método **start()** para cada uno de los animales de la simulación que llamara al método sobrescrito run de cada animal y más tarde esperara con **join()** para terminar la ejecución del programa. Esta será la clase que ejecutar para que funcione el juego.

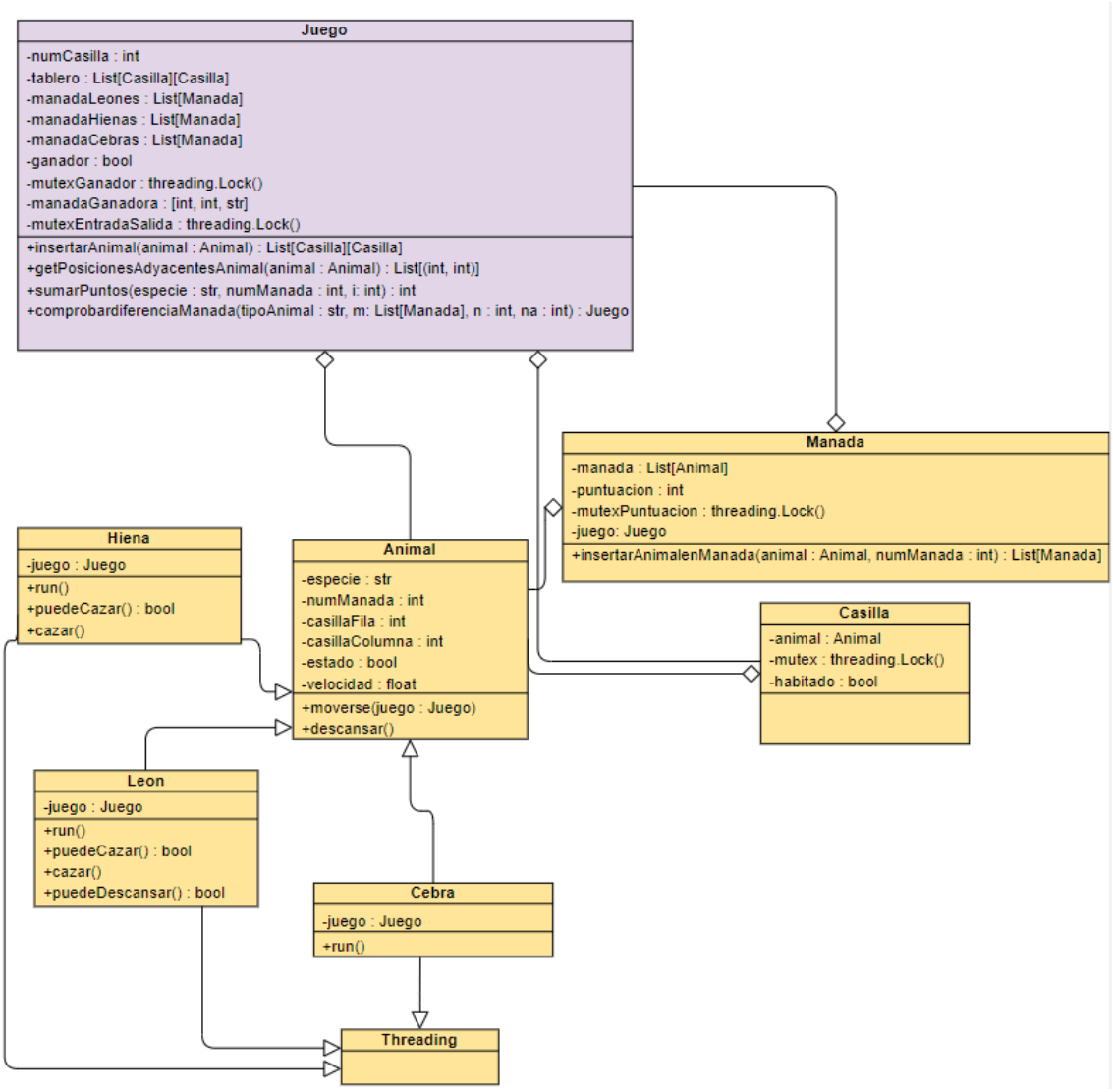

Figura 1 : Diagrama de clases de la sabana africana

2.2. Requisitos del juego

A continuación, enumerare los requisitos principales del juego para poder explicar uno a uno cómo los he cumplido:

1. Los animales deben ser hilos.
2. Velocidades diferentes por manada, leones más rápidos, luego cebras y por último hienas.
3. Animales deben descansar siendo los leones los que más descansen.
4. Debe haber al menos dos manadas, el numero de cebras es 6 veces el de los leones y el de hienas 3 veces.
5. Los leones cazan cebras siempre y hienas solo si tienen igualdad o superioridad numérica en casillas adyacentes.
6. Las hienas solo cazan cebras si tienen superioridad en casillas adyacentes
7. Ningún animal en la misma casilla.
8. Una manada ganadora al llegar a 20 puntos, una cebra vale 1 punto y una hiena 2.

9. Cuando se caza una cebra se debe crear otra de la misma manada.
10. Los movimientos y capturas en exclusión mutua.
11. Todos los movimientos permitidos, incluido el no hacer nada.
12. Se deben mostrar los movimientos.
13. Los animales deben aparecer juntos y desplazarse de manera grupal.
14. No se permiten variables globales.

El primero de los requisitos lo he solucionado mediante la **herencia** a la clase **Threading.Thread**, gracias a heredar de esta clase, el animal se convierte en un hilo y puedo sobrescribir el método **run** para que realice lo que quiero.

En cuanto a las velocidades para las diferentes manadas, lo que he hecho es a la hora de insertar los animales en las distintas manadas con su número, dividir ese numero de manada por 10000 en el caso de los leones, 9000 en el caso de las cebras y 8000 para las hienas, el motivo de estos números es que el león debe ser el más rápido por eso su número es el más grande seguido de cebras y hienas, pero como el numero de la manada puede ser el 0 he sumado un pequeño número, el más pequeño a los leones y el más grande para hienas para así controlar que la manada 0 también cumpla esta restricción y una vez calculado el numero lo establezco al atributo del animal mediante un set(), y por último en el método run de cada animal he realizado un sleep() con la velocidad del animal.

Para que el león sea el animal que más descansa he realizado un random y mediante condiciones al **león** le he dado un **75 %** de probabilidad de descansar mientras que para **las cebras y hienas** es un **33%**, el método descansar cuenta con un random de floats de 0 a 1 en el que el numero elegido pasara a un sleep.

A la hora de asignar el numero de manadas he puesto como mínimo 2 manadas y el numero de leones también es mínimo 2 (1 por manada) y como máximo el numero de casillas, una vez elegido el numero de leones para obtener el numero de cebras se multiplica por 6 y para obtener el numero de hienas por 3. En este punto encontré una dificultad con el numero de animales por manada, ya que al ser mínimo 2 algunos animales no los insertaba correctamente en la manada, para solucionar esto decidí implementar el método **comprobardiferenciaManada**() que calcula la diferencia entre los agregados a las manadas y el número total y así añadirlos a las manadas con números de manadas más pequeños, pudiendo así obtener manadas con distintos números de integrantes.

Para cumplir la restricción número 5 y 6 añadí el método **puedeCazar**() que devolverá un booleano indicando si se cumple las restricciones de caza, comprobando las casillas adyacentes al animal.

La restricción número 7 la cumplí mediante el **booleano habitado** presente en la clase Casilla, así cada vez que quiera insertar un animal en una casilla compruebo si está ocupada mediante el mutex de cada casilla.

El bucle while de ejecución de cada hilo tiene como condición que la variable ganadora no sea True, en el momento que una manada llegue a 20 puntos o 21, siendo el resultado 21 si la manada tiene 19 puntos y caza a una hiena, se bloqueara la variable ganadora mediante su mutex y se cambiara el valor del booleano a True, además la variable manadaGanadora obtendrá el valor del numero de la manada y el tipo de manada (leones o hienas).

El método **cazar** satisface las restricciones 8 y 9, para realizar una captura primero compruebo las posiciones adyacentes ocupadas, si no tiene ninguna ocupada entonces no puede cazar, si tiene alguna posición la elijo aleatoriamente y compruebo con el método puedeCazar que cumpla la restricción de caza si la cumple bloqueo la casilla donde esta la presa para que ningún otro animal pueda interaccionar con esta casilla, establezco su estado a muerto para que el animal no pueda moverse y marco la casilla como desocupada, luego bloqueo la casilla donde se encuentra el cazador y marco la casilla del cazador como desocupada y en la casilla donde estaba la presa establezco el animal cazador marcando esa casilla como habitada, libero las dos casillas y establezco los atributos fila y columna del cazador para que coincidan con la nueva casilla. Una vez el animal ha sido cazado si se trata de una cebra se crea una nueva cebra con el numero de manada guardado en un paso anterior, se inserta el animal en la simulación y se suma los puntos por captura a la manada, 1 en este caso, una vez sumado se hace un start() de la nueva cebra para que comience a moverse por el juego, si se trata de una hiena se suman 2 puntos por captura y se acaba la ejecución del método. Así genero una nueva cebra y modifico el valor de la variable puntuación satisfaciendo las dos restricciones 8 y 9 y parte de las capturas de la restricción 10.

Para cumplir la restricción 10 realizo el método moverse este método obtiene el tablero actual y marca las posiciones adyacentes libres del animal, si no hay ninguna libre el animal no hace nada y se queda en la misma posición, si hay alguna libre elige la posición aleatoriamente y bloquea la casilla correspondiente a esa posición, una vez bloqueada marca la posición inicial como desocupada, guarda en las variables de fila y columna del animal en la posición nueva, bloqueo la nueva casilla y establezco el animal en ella y su estado pasa ha habitado, para terminar libero los mutex de las casillas y así controlo los movimientos con exclusión mutua.

El animal en el bucle de ejecución elige un numero aleatorio, este número representará lo que hacer, dependiendo del número hará una acción o ninguna, dándole así la oportunidad al animal de no realizar ninguna acción, una vez que complete la acción decidirá de manera aleatoria si descansar o no.

Al principio del programa se mostrará cada animal insertado en su posición, así como una matriz donde se puede ver de forma visual el estado del juego antes de iniciar, una vez iniciado se imprimirá por pantalla cada movimiento y cada captura con el animal que lo realiza y las posiciones donde lo realiza. Antes de realizar un print se bloqueará el mutex del juego de entrada y salida y una vez realizado se liberará.

La restricción número 13 no he sido capaz de terminarla por falta de tiempo, había pensado la idea de insertar el primer elemento de la manada en una posición aleatoria y en función de esa posición insertar los demás miembros en las posiciones adyacentes.

La última restricción la he cumplido mediante la implementación de getters y setters en cada clase para así no modificar los atributos privados de cada clase. En Python no existen atributos privados, pero por seguridad en el código he decidido ocultarlos de la parte publica de la API mediante la inserción de __ al inicio del nombre de la variable, así me aseguro 100% la no utilización de variables globales.

4. Conclusión

Uno de los principales problemas que he tenido a la hora de realizar la práctica ha sido a la hora de evitar **interbloqueos (deadlocks)**, esto ocurre cuando un conjunto de hilos está esperando un recurso que solo puede generar o liberar otro hilo de ese conjunto que está esperando, por lo que se quedan eternamente esperando.

Gracias a la realización de esta práctica he podido comprender el funcionamiento de los hilos y como proteger las secciones críticas de los mismos, para obtener el comportamiento deseado. Además, me ha permitido mejorar mis habilidades con el lenguaje de programación Python, ya que he tenido que buscar mucha información acerca de los hilos y locks.