Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2024-2

Publicación: 27 de Septiembre de 2024
Actividad evaluada

Actividad 4
Threading

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta: `Actividades/AC4`
- **Fecha máxima de entrega:** 04 de Octubre 23:59
- **Ejecución de actividad**: La Actividad será ejecutada **únicamente** desde la terminal del computador. **Los *paths* relativos utilizados en la Actividad deben ser coherentes con esta instrucción, y no pueden modificarse.**

Introducción

El Departamento de Ciencia de la Computación (DCC) quiere fomentar una vida deportiva entre sus estudiantes. Para esto va a proponer una **DCCarrera**, una competencia entre tres jugadores que consiste en llegar a la meta con una bandera que deberán capturar a la mitad de la carrera. Antes de solicitar la inscripción de sus estudiantes, te solicita generar una simulación de esta situación aplicando los conceptos de *threading* recién aprendidos.

Flujo del programa

El programa consiste en una competencia entre tres jugadores que correrán una carrera de 100 metros, donde a mitad de camino se encuentra una bandera que debe ser capturada y llevada hasta la meta.

Al llegar a los 50 metros, los jugadores intentan captura la bandera. Quien tenga la bandera se ve obligado a correr más cuidadosamente, bajando su velocidad.

Si un competidor no lleva la bandera, intentará robarla durante el resto de la carrera. El primer jugador que pase la meta **con la bandera** en su posesión, es el ganador de la carrera, obteniendo 10 puntos por su victoria y terminando la carrera. El puntaje de los otros dos jugadores dependerá de su posición al término de la carrera; un jugador sin bandera obtendrá 5 puntos si ya cruzó la meta y 0 puntos en caso contrario.

Esta actividad consta en completar 2 clases utilizando los contenidos de *Threadings*. La primera clase modela toda la situación de un competidor, mientras que la segunda modela la carrera. Ambas clases serán corregidas exclusivamente mediante el uso de *tests*.

Finalmente, debes asegurarte de entregar, como mínimo, el archivo que tenga el *tag* de `Entregar` en la siguiente sección. Los demás archivos no es necesario subir, pero tampoco se penalizará si se suben al repositorio personal.

Archivos

Archivos de código

En el directorio de la actividad encontrarás los siguientes archivos de código:

- Entregar Modificar `main.py`: Aquí encontrarás la definición básica de las clases que debes implementar y completar, junto con la simulación de una carrera. Al finalizar la actividad, debes asegurar que este archivo estar subido en el lugar de entrega correspondiente.
- No modificar `tests_publicos`: Carpeta que contiene diferentes `.py` para ir probando si lo desarrollado hasta el momento cumple con lo esperado. **En la última hoja del enunciado se encuentra un anexo de cómo ejecutar los *tests* por parte o todos**.

Entidades

No modificar `class Bandera`:
Clase que representa una bandera. Registra el nombre del jugador que la lleva.

- No modificar `def __init__(self) -> None`:
  Inicializa una instancia de bandera y guarda, como atributo, el nombre del dueño de la bandera. Inicialmente este valor parte como `None`.
- No modificar `def actualizar_dueño(self, nombre_jugador: str) -> None`:
  Actualiza el nombre del jugador que lleva a la bandera.

Modificar `class Jugador`:
Thread que representa a un competidor de la carrera. El programa no puede finalizar antes de que este *thread* termine su ejecución. Además, posee las variables de clase `TIEMPO_ESPERA`, `PORCENTAJE_MIN`, `PORCENTAJE_MAX`, `PROBABILIDAD_ROBAR`, `DISTANCIA_AVANZAR`, e incluye los siguientes métodos:

- Modificar `def __init__(self, nombre: str, bandera: Bandera, lock_bandera: Lock, senal_inicio: Event, senal_fin: Event, lock_carrera: Lock) -> None`:

  Inicializador de la clase. Debe asignar los siguientes atributos:

`self.daemon`	Modificar Un `bool` que determina si el *thread* es *daemon*.
`self.bandera`	Un objeto de la clase `Bandera` que representa la instancia de bandera que se debe capturar.
`self.senal_inicio`	Un `Event` que representa señal de inicio de la carrera.
`self.senal_fin`	Un `Event` que representa señal de término de la carrera.
`self.lock_bandera`	Un `Lock` que protege el recurso `bandera`. Este *lock* es compartido entre todos los jugadores.
`self.lock_carrera`	Un `Lock` que asegura que solo 1 jugador pueda modificar el estado de la carrera a la vez. Este *lock* es compartido entre todos los jugadores.
`self.rivales`	Una lista de objetos `Jugador` que representa a los competidores del jugador en la carrera.
`self.tiene_bandera`	Un `bool` que indica si el competidor tiene la bandera.
`self.puntaje`	Un `int` que registra el puntaje del jugador.
`self._posicion`	Un `int` que representa la posición del jugador en la carrera, en metros. Es una atributo privado que tiene un valor de 0 por defecto, y cuyo valor no puede superar los 100 metros.
`self._correr`	Un `bool` que indica si el jugador debe continuar corriendo.

- No modificar `def posicion(self) -> int:`
  Property encargada de manejar la posición del jugador.

- No modificar `def dist_avance(self) -> int:`
  Property encargada de manejar la velocidad de avance del jugador. La velocidad baja en un 50 % si el jugador lleva una bandera.

- No modificar `def agregar_rival(self, rival: Jugador) -> None:`
  Método encargado de guardar una referencia al rival del jugador en la carrera.

  IMPORTANTE: Este método crea una referencia circular, que en sí misma no es un problema, sin embargo, dado que Python no permite la creación de métodos y atributos privados, esto debe usarse con mucho cuidado.

- No modificar `def avanzar(self) -> None:`
  Método que simula el movimiento del jugador. Aumenta la posición en una cantidad aleatoria entre el `PORCENTAJE_MIN` % y `PORCENTAJE_MAX` % de la velocidad del jugador y luego imprime su nueva posición.

- Modificar `def perder_bandera(self) -> None:`
  Método encargado de simular la pérdida de la bandera. Si el jugador tiene la bandera, este método debe cambiar el atributo `tiene_bandera` a `False` e imprimir un mensaje notificando la pérdida de la bandera. En caso que el jugador no tuviera la bandera, este método no hace nada.

- Modificar `def capturar_bandera(self) -> None:`
  Método encargado de simular la captura de la bandera. En primer lugar, este método debe asegurarse de que todos los rivales del jugador pierdan la bandera, utilizando el método `perder_bandera` de cada rival. Luego, actualiza el nombre del jugador que tiene la bandera. Finalmente, se debe modificar el atributo `tiene_bandera` del jugador actual a `True`.

- Modificar `def intentar_capturar_bandera(self) -> None:`
  Método que simula el **intento** de capturar la bandera a mitad de la carrera. Para esto, se debe intentar adquirir el `lock_bandera`, pero **NO** debe quedarse esperando a que se libere el `lock` si no logra obtenerlo. En caso de obtener el `lock` y la bandera no tenga un dueño todavía, debe llamar al método `capturar_bandera`, imprimir un mensaje avisando que se logró la captura y debe liberar el `lock_bandera`. En caso de obtener el `lock`, pero la bandera ya tiene un dueño, simplemente se libera el `lock`.

  Para este método te entregamos los siguientes ***hints***:

  - El método `acquire()` de la clase `Lock` recibe como parámetro el booleano *blocking*, que le indica si debe quedarse esperando a que se libere el *lock* (`True`) o si debe intentarlo solo una vez y seguir su ejecución (`False`).
  - El método `acquire()` de la clase `Lock` retorna `True` si el *lock* fue adquirido correctamente y `False` en caso contrario.
  - Para más información sobre `Lock` y `acquire`, puedes revisar la documentación de `Lock`.

- Modificar `def intentar_robar_bandera(self) -> bool:`
  Método que simula el **intento** de robar la bandera a otro jugador. Debe lograr robar la bandera con una probabilidad de `PROBABILIDAD_ROBAR` %. Si se logra el robo se debe capturar la bandera, asegurándose de que solo un jugador pueda capturarla al mismo tiempo. Se imprime un mensaje avisando que se robó la bandera y finalmente se debe retornar `True`. En caso de que falle el robo, sólo se debe retornar `False`.

- No modificar `def correr_primera_mitad(self) -> None`:
  Método encargado de simular la forma en que corre el jugador durante la primera mitad de la carrera. El jugador debe avanzar y luego el *thread* debe detenerse por `TIEMPO_ESPERA` segundos.

- Modificar `def correr_segunda_mitad(self) -> bool`:
  Método encargado de simular la forma en que corre el jugador durante la segunda mitad de la carrera. Primero se debe adquirir el `lock_carrera`, si este ya está tomado, el jugador tendrá que esperar que este sea liberado.

  Teniendo el `lock_carrera`, el jugador deberá verificar:

  1. Si la señal de fin de la carrera fue levantada: significa que el jugador perdió la carrera, por lo que debe modificar su atributo `self._correr` a `False`, para dejar de correr y además debe retornar `False`.
  2. Si la posición del jugador es igual a 100 y tiene la bandera: debe avisar que la carrera terminó, modificar su atributo `self._correr` para dejar de correr y retornar `True`.
  3. Si el jugador no tiene la bandera: debe intentar robarla.

  Después de haber verificado todo lo anterior, se debe liberar el `lock_carrera`, luego el jugador avanzará en la carrera mediante el llamado de `self.avanzar` y se pausará el *thread* por `TIEMPO_ESPERA` segundos.

- Modificar `def run(self) -> None`:
  Método encargado de la ejecución del *Thread*. Debe realizar los siguientes pasos:

  1. Al iniciarse la ejecución del *thread*, el jugador debe esperar hasta que se avise el inicio de la carrera.
  2. Mientras no supere los 50 metros, debe correr utilizando método `correr_primera_mitad`.
  3. Tras superar los 50 metros, debe intentar capturar la bandera mediante el método `intentar_capturar_bandera`.
  4. Finalmente, mientras su atributo `self._correr` sea verdadero, debe correr la segunda mitad de la carrera utilizando el método `correr_segunda_mitad`.

Modificar `class Carrera`:
Thread que representa una carrera entre 3 jugadores. El programa principal debe esperar que este *thread* finalice para terminar con su programa. Además, posee los siguientes métodos:

- Modificar `def __init__(self, juga_1: Jugador, juga_2: Jugador, juga_3: Jugador, senal_inicio: Event, senal_fin: Event) -> None`:

  Inicializador de la clase. Debe asignar a los jugadores como rivales y además definir los siguientes atributos:

`self.daemon`	Modificar Un `bool` que determina si el thread es daemon.
`self.senal_inicio`	Un `Event` que representa señal de inicio de la carrera.
`self.senal_fin`	Un `Event` que representa señal de término de la carrera.
`self.jugador_1`	Un `Jugador` que corresponde uno de los competidores de la carrera.
`self.jugador_2`	Un `Jugador` que corresponde uno de los competidores de la carrera.
`self.jugador_3`	Un `Jugador` que corresponde uno de los competidores de la carrera.

  Además de lo anterior, este método se encarga de asignar los rivales de cada jugador.

- Modificar `def run(self) -> None`:
  Método encargado de la ejecución del *thread*. En primer lugar, debe iniciar los *threads* de los jugadores. Luego, debe avisar el inicio de la carrera y esperar hasta que todos los jugadores terminen su ejecución.
- No modificar `def entregar_ganador(self) -> None`
  Método encargado de calcular el puntaje de los jugadores tras finalizar una carrera. Luego imprime el resultado de la competencia.

Notas

- **No puedes hacer *import* de otras librerías externas a las entregadas en el archivo.**
- Recuerda que la ubicación de tu entrega es en **tu repositorio de Git**. En la rama (*branch*) por defecto del repositorio: `main`.
- Se recomienda completar la actividad en el orden del enunciado.
- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Si aparece un error inesperado, ¡léelo! Intenta interpretarlo y/o buscarlo en Google.
- Se recomienda probar tu código con los *tests* y ejecutando `main.py`, este último se ofrece un pequeño código donde se prueba la carrera con 3 jugadores.

Objetivo de la actividad

- Aplicar conocimientos de *Threading* para permitir la concurrencia de código.
- Aplicar conceptos de *Lock* para controlar el acceso a código crítico.
- Utilizar señales y el método `join` para permitir la interacción entre *threads* cuando corresponda.
- Probar código mediante la ejecución de *test* y del archivo `main.py`.

Ejecución de *tests*

En esta actividad se provee de varios archivos `.py` los cuáles contiene diferentes *tests* que ayudan a validar el desarrollo de la actividad. Para ejecutar estos *tests*, **primero debes posicionar tu terminal/consola en la carpeta de la actividad (`Actividades/AC4`)**. Luego, desde esta misma, debes escribir el siguiente comando para ejecutar todos los *tests* de la actividad:

- `python3 -m unittest discover tests_publicos -v -b`

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes hacerlo si escribes lo siguiente en la terminal/consola:

- `python3 -m unittest -v -b tests_publicos.test_jugador`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `Jugador`.
- `python3 -m unittest -v -b tests_publicos.test_carrera`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `Carrera`.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.