Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2021-1

22 de abril de 2021
Actividad Formativa

Actividad Formativa 3

Threading

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AF3/`
- **Hora del *push*:** 16:30

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Introducción

¡Caos en el campus San Joaquín! Incendios y choques han comenzado a aparecer en todas partes. Distintas compañías se comienzan a desplegar a lo largo del campus para contrarrestar el poder destructivo de **DCCatástrofe**. Debido al surgir caótico de los incidentes, los jefes de cada compañía buscan ayuda en la poca gente que hay dentro del campus y se encuentran ~~de completa casualidad~~ contigo. Deberás ayudar a entender cómo se comportan los accidentes para que las compañías puedan socorrer a todos sin que hayan heridos graves.

Para esto, deberás hacer una simulación de los accidentes con *threads*. Cada accidente será un *thread* distinto, y tendrás que encontrar la manera de representar el manejo de emergencias por las distintas compañías, además de entregar los recursos correctamente.

Flujo del programa

En esta actividad tendrás que completar dos archivos, `main.py` y `emergencias.py`. El archivo `main.py` se encargará de correr la simulación por completo, utilizando las clases que están en `emergencias.py`. La simulación deberá iterar sobre las catástrofes que van a ocurrir, cargadas desde `emergencias_db.csv` y mitigar cada una de ellos mediante la creación de *threads* personalizados para cada emergencia. Los *threads* necesitarán recursos entregados por las instancias de la clase `CompaniaServicio`, que se encuentra en `companias.py`.

Cada emergencia requiere una distinta cantidad de recursos, que dependerá de la gravedad de la emergencia.

Archivos

- `emergencias.py`: En este archivo están las clases `Emergencia`, `Incendio` y `Choque`, que tendrás que completar en la **parte I**.
- `main.py`: Este es el archivo principal del programa. Puedes ejecutarlo para probar el funcionamiento del programa completo. Contiene la clase `DCCatastrofe`, que tendrás que completar en la **parte II**.
- `companias.py`: En este archivo está la clase `CompaniaServicio`, que maneja los recursos utilizados para mitigar las emergencias. **No debes modificarlo**.
- `parametros.py`: Contiene los valores máximos que utilizará la clase `CompaniaServicio`. **No debes modificarlo**.
- `emergencias_db.csv`: En este archivo de texto se encuentran todas las emergencias que debes manejar. Cada línea sigue el siguiente formato, donde `tipo_emergencia` es un `str`: `"incendio"` o `"choque"`, y `mensaje` es un `str` con su aviso respectivo:

```
tipo_emergencia,mensaje
```

Compañías

La siguiente clase se encuentra en `companias.py` y proveerá los recursos utilizados para mitigar las emergencias, dependiendo si se necesita agua para los bomberos, banditas para el personal médico o donas para la policía. **Ya viene implementada y no debe ser modificada.**

`class CompaniaServicio`: Entregará los recursos para controlar cada `Emergencia`, representando a una compañía por recurso necesitado. Cada compañía puede entregar recursos para mitigar **sólo una emergencia a la vez**.

- `def __init__(self, tipo: str, recursos_max: dict)`: recibe el tipo de recurso a distribuir, pudiendo ser `"agua"`, `"banditas"` o `"donas"` y es guardado en el atributo `self.tipo`. También recibe `recursos_max`, que corresponde a un diccionario con llaves de un tipo de recurso específico y las capacidades máximas de almacenamiento de este recurso como valor respectivo. Con este diccionario y el `tipo` recibido como llave se extrae la capacidad máxima del recurso administrado y se guarda en `self.capacidad_maxima`. Además se define `self.stock`, que representa al stock de la compañía en un momento dado, es inicialmente igual a `self.capacidad_maxima` y nunca será mayor a este valor. Por último, se crea `self.disponibilidad`, que corresponde a un objeto de clase `Lock`.
- `def solicitar(self, cantidad: int)`: recibe la cantidad del recurso solicitado y revisa `self.stock` para ver si tiene la cantidad solicitada. En caso de no tener, se recarga igualando `self.stock` a

`self.capacidad_maxima` para posteriormente restarle la cantidad solicitada. Si tiene más stock de lo solicitado, simplemente se le resta la cantidad solicitada a `self.stock`.

Parte I: Construir Threads personalizados

Debes completar lo solicitado en las siguientes clases:

`class Emergencia`: Esta clase debe heredar de `Thread`.

Posee el siguiente **atributo global de clase**:

- `companias`: atributo global de clase tipo `dict`, cuyas llaves son `"agua"`, `"banditas"` y `"donas"`, y sus valores respectivos son instancias de clase `CompaniaServicio` con su atributo `self.tipo` igual a la llave respectiva.

Debes completar el siguiente **método**:

- `def __init__(self, aviso: str, numero_catastrofe: int)`: debe recibir el aviso y el número de catastrofe y guardarlo como atributos con **los mismos nombres**. Además, posee `self.gravedad`, que se calcula como un número aleatorio entre **1** y **10**. Debes hacer lo necesario para que **herede de `Thread` correctamente**.

`class Incendio`: Debe heredar de `Emergencia`. Debes completar los siguientes métodos.

- `def __init__(self, aviso: str, numero_catastrofe: int)`:
  - deberá pasar los argumentos correspondientes a la **superclase**
- `def llamar_bomberos(self)`: Este método **se demora**[1] una cantidad de tiempo igual a `self.gravedad` en apagar el incendio, para finalmente imprimir la catástrofe con su número respectivo, como finalizada.
- `def run(self)`:
  - Imprime el inicio de la catástrofe junto con su número de catástrofe y su aviso. A continuación calcula su `agua_necesaria` y `banditas_necesarias`. **[Ya implementado]**
  - Luego, se debe trabajar con el `Lock` de la instancia `CompaniaServicio` de tipo `agua`, y llamar al método `solicitar` con la cantidad de agua requerida.
  - A continuación, debe trabajar con el `Lock()` de la instancia `CompaniaServicio` de tipo `banditas`, y llamar al método `solicitar` con la cantidad de banditas requeridas.
  - Finalmente, se debe apagar el incendio llamando a `self.llamar_bomberos()`.

[1] Puedes usar la librería `time`

`class Choque`: Debe heredar de emergencia. Debes completar los siguientes métodos.

- `def __init__(self, aviso: str, numero_catastrofe: int)`:
  - deberá pasar los argumentos correspondientes a la **superclase**
- `def atender_heridos(self)`: Este método **se demora** una cantidad de tiempo igual a `self.gravedad` en atender a los heridos, y luego imprime la catástrofe con su número respectivo, como finalizada.
- `def run(self) -> None`:
  - Imprime el inicio la catástrofe junto con su número de catástrofe y su aviso. A continuación, se calculan sus `donas_necesarias` y `banditas_necesarias`. [**Ya implementado**]
  - Luego, debe trabajar con el `Lock` de la instancia `CompaniaServicio` de tipo `banditas`, y llamar al método `solicitar` con la cantidad de banditas requerida.
  - A continuaciæon, debe trabajar con el `Lock` de la instancia de `CompaniaServicio` de tipo `"donas"`, y llamar al método `solicitar` con la cantidad de donas requeridas.
  - Finalmente, se deben atender a los heridos llamando a `self.atender_heridos()`.

TIP: Puedes correr el modulo `emergencias.py` para saber si los threads se instancian bien. Siéntete libre de editarlo y modificar la sección `if __name__ == "__main__"` dentro de `emergencias.py`.

Parte II: Iniciar simulación

En esta parte deberás completar el archivo `main.py`, específicamente dentro de `DCCatastrofe`:

`class DCCatastrofe`: esta clase maneja la simulación creando las distintas emergencias.

- `def cargar_emergencias(self) -> List`: maneja `emergencias_db.csv` y retorna una **lista de tuplas**, donde en cada tupla el primer elemento es el **tipo** y el segundo es el **aviso**, de una emergencia. [**Ya implementado**]
- `def iniciar_simulacion(self) -> None`: debe cargar las emergencias con el método `self.cargar_emergencias()`, iniciar cada una de las emergencias con un **intervalo de aleatorio entre 1 y 3 segundos** usando un contador para cada una de ellas. Finalmente, deberás hacer que el programa espere a que finalicen todas las emergencias para dar el mensaje final de término de esta `DCCatastrofe`.

Ejemplo

Un ejemplo de cómo podría verse una implementación correcta, en la consola:

```
Inicio de catastrofe N°10:
Alguien ha iniciado un incendio en el DCC luego de demasiadas actividades de AmongUs

recargando agua...

Rescatando personas del choque 7!
Fin de catastrofe N°7

Rescatando personas del choque 9!
Fin de catastrofe N°9

Inicio de catastrofe N°11:
Coti tuvo un accidente camino al LittleCaesars

recargando banditas...

Inicio de catastrofe N°12:
El carrito de la biblioteca ha chocado con un arbol!

recargando donas...
```

OJO: Las emergencias no necesariamente se solucionan en orden, y tampoco se llamarían a los recursos en orden, ya que como hay elementos aleatorios, puede que tu output no se vea exactamente igual. Pero debe parecerse en el sentido de recargar recursos, iniciar y finalizar recursos se imprima concurrentemente.

Objetivos

- Comprender el uso de `Thread`s personalizados
- Utilizar `Lock`s de manera correcta
- Trabajar una simulación con concurrencia