INTEGRANTES:
- García Peñafort Rodrigo.
- Santana Sánchez Gustavo.

Introducción

1. Describan la situación que modelarán.

El problema planteado es una situación en la que somos roomies varios amigos (Paco, Yayo, Mau, Moi, Santana y Roy) y llega la hora de cocinar. Básicamente tenemos que compartir la estufa, sartenes, ollas, la licuadora, el microondas y un tostador. Nos tenemos que poner de acuerdo para usar de forma adecuada la estufa, ya que las ollas pueden abarcar el espacio de otro quemador y entonces alguien más tendría que esperar hasta que se libere ese espacio. También podría pasar que dos personas están intentando usar la licuadora al mismo tiempo, por lo que requerimos de coordinación.

2. ¿Dónde pueden verse las consecuencias nocivas de la concurrencia? ¿Qué eventos pueden ocurrir que queramos controlar?

Las consecuencias se generan cuando dos o más roomies quieren acceder a los mismos recursos al mismo tiempo y no hay coordinación. Los eventos que se pueden presentar son las condiciones de carrera (no se ponen de acuerdo para acceder al recurso); un uso inadecuado del recurso (alguien está usando el microondas y llega alguien más y lo abre); alguien obtiene un sartén u olla e intenta acceder a la estufa, pero no hay suficiente espacio.

3. ¿Hay eventos concurrentes para los cuales el ordenamiento relativo *no* resulta importante?

Existen eventos en los que el orden en que ocurren no afectan el resultado: Cuando alguien esta ocupando la licuadora, pero alguien más se encuentra usando el tostador; alguien está usando la estufa y alguien más el microondas. Básicamente cuando se encuentran haciendo diferentes acciones jamás va a afectar el resultado final porque no intervienen directamente con las acciones de alguien más.

Documentación

- Descripción de los mecanismos de sincronización empleados

Empleamos dos mecanismos de sincronización del paquete *java.util.concurrent*

- ***Semaphore***: controla el acceso a recursos compartidos con un cupo limitado
  - estufa: 4 permisos (4 quemadores)
  - olla y sarten: 2 permisos cada uno (2 disponibles)
- ***Reentrantlock***: controla el acceso exclusivo de ciertos recursos
  - Tostador, licuadora, microondas y lavaplatos.

De esta forma garantizamos que no haya más roomies (procesos) cocinando de lo que la estufa permite y solo una persona puede usar un electrodoméstico a la vez.

- Lógica de operación

  Cada hilo representa a un roomie que quiere preparar su comida.

- Identificación del estado compartido (¿cuáles son las variables o estructuras globales?)
  - ❖ private static final Semaphore estufa;
  - ❖ private static final Semaphore sarten;
  - ❖ private static final Semaphore olla;
  - ❖ private static final Lock tostador;
  - ❖ private static final Lock microondas;
  - ❖ private static final Lock licuadora;
  - ❖ private static final Lock lavaplatos;
- *Descripción algorítmica* del avance de cada hilo/proceso

  El hilo siempre genera su turina de cocina al azar que siempre empezará por *preparar* y *cocinar,* después puede hacer entre 1 o 3 actividades adicionales (usar el microondas, licuadora o tostador) y por último debe lavar sus utensilios.

  Siempre hay un tiempo de espera (entre 1 y 2 segundos) entre acciones.

  Cuando se está realizando una acción y requiere sincronización, entonces se realiza lo siguiente:

  - Se obtiene el recurso usando *acquire()*, para semáforos, o *lock()*, para recursos de accesos exclusivo.
  - Para simular que se está usando el recurso se emplea *Thread.sleep()* y dependiendo de la acción puede tardar más o menos tiempo.
  - Por último se libera el recurso mediante *release(),* para semáforos, o *unlock(),* para los recursos de accesos exclusivo.

  También se van mostrando los pasos con una función logEvento, la cuál se comunica con la interfaz gráfica y comunica el estado, nombre y acción del hilo (roomie) actual.

- Descripción de la interacción entre ellos (sea mediante los mecanismos de sincronización o de alguna otra manera)

  La interacción entre hilos se da únicamente a través del acceso a recursos compartidos. No se comunican directamente entre sí.

  Gracias a los **semáforos** y **locks**, se evitan interferencias:

  - ■ Si un recurso está ocupado, el hilo debe **esperar** hasta que esté libre.

  - ■ Esto permite observar fenómenos como **espera activa, cola de espera y sincronización implícita**.

- Descripción del entorno de desarrollo, suficiente para reproducir una ejecución exitosa
  - ¿Qué lenguaje emplean? ¿Qué versión?

- ¿Qué bibliotecas más allá de las estándar del lenguaje?
- ¿Bajo qué sistema operativo / distribución lo desarrollaron y/o probaron?

Esta simulación fue desarrollada y probada en el siguiente entorno:

- **Lenguaje**: Java (para el backend) y python (interfaz gráfica)

- **Versión**: JDK 21.0.6, Python 3.13.2

- **Bibliotecas**:

    - Solo se usaron clases del *paquete estándar de Java*, principalmente:
        - java.util.concurrent.ExecutorService
        - java.util.concurrent.Executors
        - java.util.concurrent.Semaphore
        - java.util.concurrent.locks.Lock
        - java.util.concurrent.locks.ReentrantLock
        - java.util.Random
        - java.util.List
        - java.util.Collections
        - java.util.ArrayList
        - java.util.Arrays
        - java.util.concurrent.TimeUnit
    - En python:
        - tkinter (versión 8.6)
        - tkinter.ttk
        - subprocess
        - threading
        - collections.defaultdict

      Para este caso se tuvo que instalar la biblioteca tkinter empleando el comando: ***sudo dnf install python-tkinter***

- **Sistema operativo / distribución**:

    - Desarrollado y probado en:

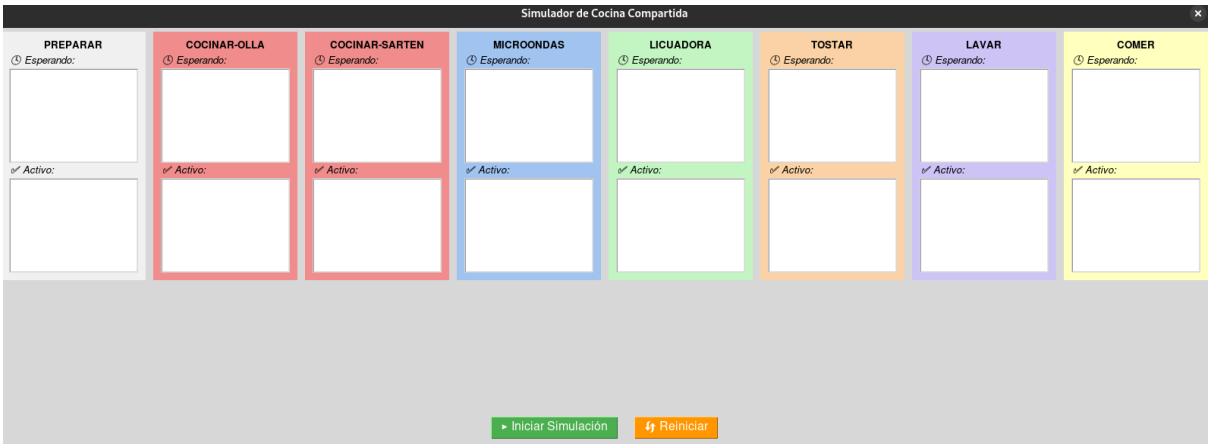
        - Fedora Linux 41 Workstation Edition

- Ejemplos o *pantallazos* de una ejecución exitosa.

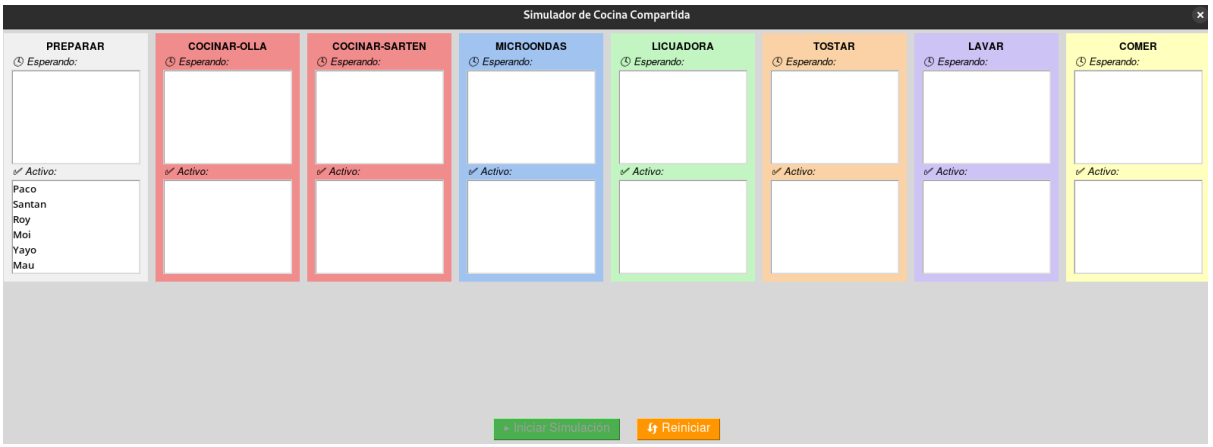
    - Invocación:

```
javac CocinaCompartida.java
python interfazCocinaCompartida.py
```

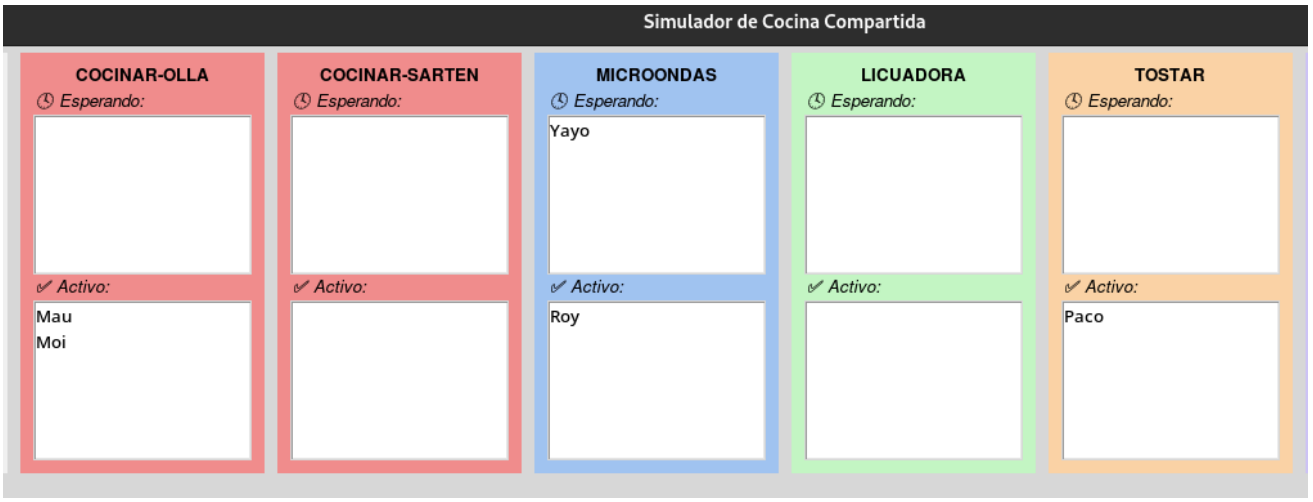
- Interfaz sin iniciar el programa aún:



- Iniciamos el programa: Todos empiezan a preparar sus ingredientes



- Transcurso de la ejecución



- La mayoría se encuentran en el área de lavar, por lo que es el último paso antes de terminar

- Final de la ejecución

- Otros ejemplos más de cómo finaliza la ejecución

De esta forma podemos observar que el orden en el que terminan es diferente, por lo que siempre dependerá de las acciones aleatorias que se asignen durante la ejecución del programa.