

PROYECTO 2: UNA SITUACIÓN COTIDIANA PARALELIZABLE

ALUMNO: MARTINEZ VILLEGAS PEDRO

SISTEMAS OPERATIVOS

GPO: 6



Premisa: Los eventos en el mundo en que vivimos se producen de forma masivamente paralela, por lo cual, la concurrencia puede presentarse en muchos casos, en ámbitos no necesariamente constreñidos al cómputo.

Por tanto, es de esperarse que –si existieran las primitivas de sincronización en el mundo real– podríamos aplicar lo estudiado en esta unidad a muchas otras situaciones.

1.- Identificación y descripción del problema

Después de pensar un rato y tomando de referencia un puesto de comida que está enfrente de mi casa decidí crear un Sistema de Pedidos en un Restaurante con Servicio a Domicilio. La situación que se modela es un sistema de pedidos en un restaurante con servicio a domicilio. El sistema involucra diferentes actores, como meseros, cocineros y repartidores, cada uno desempeñando un papel específico en el proceso de recibir, preparar y entregar pedidos.

Los meseros son responsables de tomar los pedidos de los clientes. Esta interacción se simula a través de una interfaz de usuario donde se solicita el nombre del cliente, su apellido, el pedido y su dirección. Luego, el mesero envía esta información al sistema. El sistema, representado por la clase *Restaurante*, gestiona los pedidos. Primero, verifica si el pedido es válido (esto porque se definió que, si se escribe en el campo pedido: 'nada', o simplemente se manda sin este campo lleno; no se preparará nada). Si es válido, agrega el pedido a una cola de pedidos pendientes y procede a preparar la comida.

El cocinero, representado por la función *preparar_comida*, toma los pedidos pendientes de la cola y los prepara. Una vez que la comida está lista, el pedido se mueve a una cola de pedidos listos para que, finalmente, los repartidores tomen los pedidos de la cola de pedidos listos y los entreguen a los clientes (Los pedidos se asignan a repartidores de manera aleatoria).

¿Qué tiene que ver la concurrencia con esto?

Bueno, funciona a través de tomar pedidos en un restaurante, prepararlos para su entrega y repartirlos. involucra múltiples procesos (meseros, cocineros, repartidores) que trabajan de manera simultánea e independiente para llevar a cabo diferentes etapas del proceso de pedido en el restaurante. La concurrencia es importante pues se asegura que los pedidos se manejen de manera eficiente y no se mezclen.

Hablando en lo real, los meseros pueden tomar pedidos de diferentes clientes al mismo tiempo. Mientras tanto, los cocineros pueden preparar múltiples platos de manera concurrente, y los repartidores pueden estar entregando pedidos a diferentes destinos simultáneamente.

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

Las consecuencias nocivas de la concurrencia pueden manifestarse en situaciones donde los procesos interdependientes no están adecuadamente sincronizados o coordinados. En nuestro planteamiento, las consecuencias nocivas pueden verse en:

- **Conflictos en la preparación de la comida:** Si los cocineros no están sincronizados adecuadamente, podrían intentar preparar el mismo platillo al mismo tiempo, lo que podría resultar en una sobrecarga de recursos o incluso en la preparación incorrecta de los platillos.
- **Asignación incorrecta de pedidos:** Si los repartidores no están coordinados, podrían terminar llevando el pedido incorrecto a la dirección incorrecta, lo que generaría insatisfacción en los clientes.
- **Situaciones de carrera en la interfaz de usuario:** Si varios clientes intentan realizar un pedido al mismo tiempo, podrían surgir problemas de concurrencia en la interfaz de usuario, lo que podría llevar a errores en la entrada de datos.

Para controlar estos eventos y prevenir las consecuencias nocivas de la concurrencia, utilicé técnicas de sincronización como colas y bloqueos en nuestro código. Esto ayuda a asegurar que los procesos se ejecuten de manera ordenada y que los recursos compartidos se manejen adecuadamente :).

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

Existen eventos concurrentes en nuestro problema en los que el orden relativo no es crítico. Por ejemplo:

- **Recepción de pedidos, meseros:** Los meseros pueden tomar los pedidos de diferentes clientes de manera independiente y simultánea. El orden en el que los meseros toman los pedidos no afecta el resultado final.
- **Preparación de platillos, cocineros:** Los cocineros pueden preparar los platillos en paralelo. No importa en qué orden se preparan los platillos, siempre y cuando se aseguren de que cada platillo se prepare correctamente.
- **Asignación de pedidos a repartidores:** Una vez que los pedidos están listos, pueden ser asignados a los repartidores de manera independiente. El orden en el que se asignan los pedidos a los repartidores no afecta la entrega final de los pedidos, siempre y cuando se entreguen bien.

2.- Implementación del modelo e introducción de mecanismos de sincronización

Para abordar el problema, utilice hilos (*'threads'*) para simular la concurrencia de los diferentes roles en el restaurante. Cada rol (mesero, cocinero, repartidor) se representa mediante un hilo separado, esto permite acciones simultáneas.

Puntos de Sincronización y Patrones:

- **Colas de Pedidos:** Utilicé colas (*'Queue'*) para almacenar y gestionar los pedidos pendientes y los pedidos listos.
- **Punto de Sincronización:** Los meseros ponen pedidos en la cola de pedidos pendientes y los cocineros sacan pedidos de esta cola para prepararlos. Los cocineros ponen pedidos en la cola de pedidos listos y los repartidores sacan pedidos de esta cola para entregarlos.

- Patrón de Sincronización: Se utilizan métodos de la cola como put y get para agregar y sacar elementos de manera segura en entornos multiproceso.

Bloqueo al Tomar Pedidos: Cuando los meseros toman pedidos de la interfaz, hay un bloqueo para asegurarse de que un mesero termine de tomar un pedido antes de que otro pueda empezar.

- Punto de Sincronización: Cuando un mesero está tomando un pedido, se bloquea el acceso a otros meseros para evitar que tomen pedidos al mismo tiempo.
- Patrón de Sincronización: Se utiliza un mecanismo de bloqueo (*'threading.Lock()'*) para garantizar que solo un mesero a la vez pueda tomar un pedido.

Impresión en la consola: Los diferentes roles (meseros, cocineros, repartidores) pueden estar corriendo simultáneamente y es importante que los mensajes se muestren de manera clara y ordenada en la consola.

- Punto de Sincronización: Cuando se imprime un mensaje, se asegura que los roles no se mezclen.
- Patrón de Sincronización: El uso de *'print'* es seguro para operaciones de escritura en la consola pues python maneja internamente la sincronización del flujo de salida.

Interfaz gráfica: La interfaz gráfica debe ser actualizada correctamente para reflejar los pedidos pendientes.

- Punto de Sincronización: La interfaz gráfica se ejecuta en un hilo separado y se sincroniza con el resto del programa para reflejar los cambios en los pedidos.
- Patrón de Sincronización: Se utiliza *'root.mainloop()'* para ejecutar el bucle principal de la interfaz gráfica, este gestiona la interacción del usuario y actualiza la interfaz de manera sincronizada.

3.- Descripción de los mecanismos de sincronización empleados

Como se ha mencionado anteriormente, utilice como mecanismos de sincronización los hilos, colas y el bloqueo:

- Hilos: Estos sirven para simular la concurrencia de los diferentes roles en el restaurante (meseros, cocineros, repartidores), permite que múltiples acciones ocurran simultáneamente, lo que es esencial para un ambiente donde distintas tareas deben realizarse al mismo tiempo.

- Colas: Se utilizan para almacenar y gestionar los pedidos pendientes y los pedidos listos; proporciona un mecanismo seguro y eficiente para la comunicación entre hilos, permitiendo que los diferentes roles compartan información de manera ordenada y sincronizada.
- Bloqueo: Garantizan que solo un mesero a la vez pueda tomar un pedido de la interfaz, evita que múltiples meseros intenten tomar pedidos al mismo tiempo, asegurando que la operación de tomar un pedido se complete de manera exclusiva.

4.- Lógica de operación

Identificación del Estado Compartido

El estado compartido se refiere a la información que debe ser accesible por múltiples hilos para llevar a cabo la operación del restaurante. En este caso, las colas actúan como un estado compartido.

Colas de Pedidos:

- cola_pedidos_pendientes: Almacena los pedidos tomados por los meseros que están pendientes de preparación por los cocineros.
- cola_pedidos_listos: Almacena los pedidos preparados por los cocineros que están listos para ser entregados por los repartidores.

Estas colas permiten a los meseros, cocineros y repartidores interactuar y compartir información de manera ordenada y segura.

Descripción Algorítmica del Avance de Cada Hilo/Proceso

Cada hilo (mesero, cocinero, repartidor) tiene un ciclo de trabajo específico:

- Mesero:
 1. Espera a que el usuario ingrese un pedido en la interfaz.
 2. Toma el pedido y lo agrega a la cola de pedidos pendientes.
- Cocinero:
 1. Espera a que haya pedidos pendientes en la cola de pedidos pendientes.
 2. Toma un pedido de la cola de pedidos pendientes.
 3. Prepara el platillo.
 4. Agrega el pedido preparado a la cola de pedidos listos.

- Repartidor:
 1. Espera a que haya pedidos listos en la cola de pedidos listos.
 2. Toma un pedido de la cola de pedidos listos.
 3. Entrega el pedido a la dirección indicada.

Descripción de la Interacción entre Ellos

La interacción entre los roles se facilita mediante el uso de las colas y el bloqueo:

- Los meseros toman pedidos y los colocan en la cola de pedidos pendientes, los cocineros toman pedidos de la cola de pedidos pendientes y los colocan en la cola de pedidos listos una vez que están preparados y los repartidores toman pedidos de la cola de pedidos listos para entregarlos a las direcciones proporcionadas.

El bloqueo se utiliza cuando los meseros toman pedidos de la interfaz para asegurarse de que solo un mesero a la vez pueda tomar un pedido, evitando posibles conflictos.

5.- Descripción del entorno de desarrollo, suficiente para reproducir una ejecución exitosa

¿Qué lenguaje emplee?

- Decidí usar python para resolver lo que plantee, es necesario tener una versión 3 en adelante para probar el código, esto para que sea compatible con lo que se usó en el.

¿Qué bibliotecas más allá de las estándar del lenguaje?

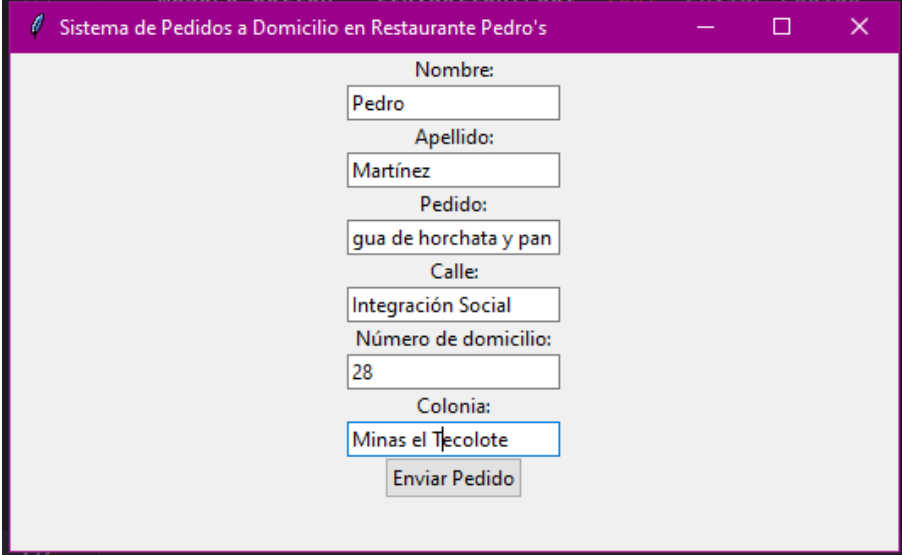
- Sólo la biblioteca tkinter, para la interfaz gráfica.

¿Bajo qué sistema operativo / distribución lo desarrollaron y/o probaron?

- Windows 10

6.- Ejemplos o pantallazos de una ejecución exitosa

El registro de un pedido normal se ve así, se llenan los campos a través de GUI y se envían para procesarlos:



Sistema de Pedidos a Domicilio en Restaurante Pedro's

Nombre:
Pedro

Apellido:
Martínez

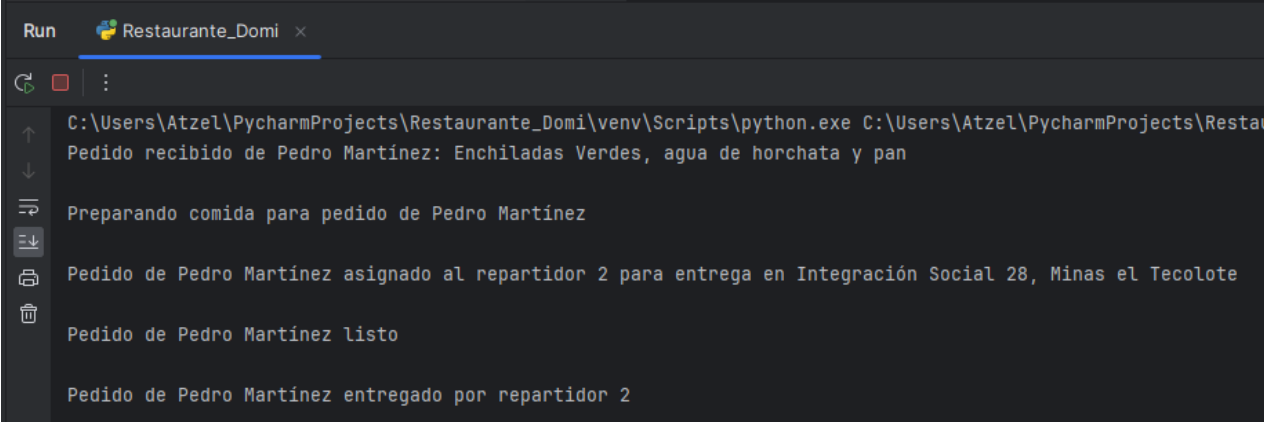
Pedido:
gua de horchata y pan

Calle:
Integración Social

Número de domicilio:
28

Colonia:
Minas el Tecolote

Enviar Pedido



```
Run Restaurante_Domi x
C:\Users\Atzel\PycharmProjects\Restaurante_Domi\venv\Scripts\python.exe C:\Users\Atzel\PycharmProjects\Resta
Pedido recibido de Pedro Martínez: Enchiladas Verdes, agua de horchata y pan

Preparando comida para pedido de Pedro Martínez

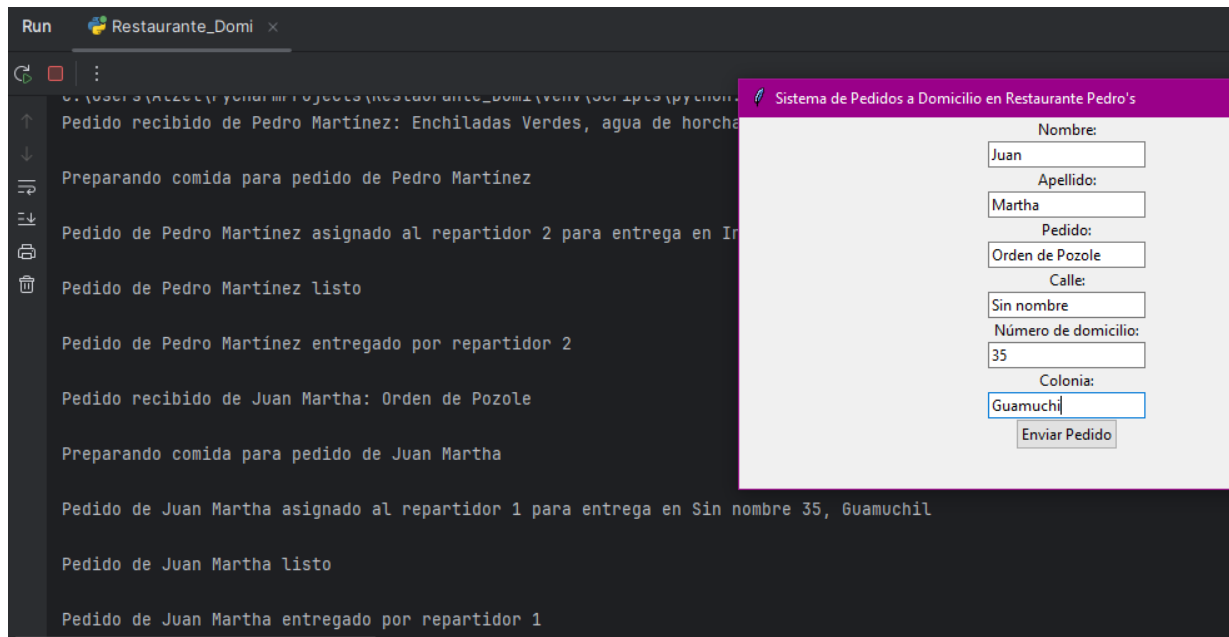
Pedido de Pedro Martínez asignado al repartidor 2 para entrega en Integración Social 28, Minas el Tecolote

Pedido de Pedro Martínez listo

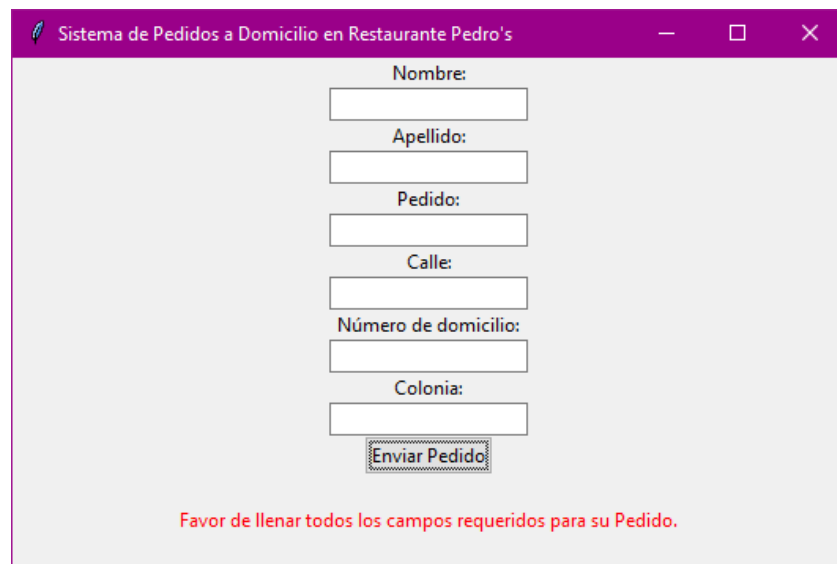
Pedido de Pedro Martínez entregado por repartidor 2
```

En base a los datos de la GUI, empieza el código a hacer lo requerido, se asignan tiempos aleatorios, el repartidor que toma la orden es aleatorio, etc.

Otro ejemplo:



Hay algunas cosas que implemente para que fuera un poco mas 'completo', como por ejemplo que si no se llena ningún dato y se presiona el botón para enviar el pedido, aparezca una advertencia:



Había agregado cosas como que aunque no pongan nombre pero que llenen lo demás, tome en cuenta el pedido; Si no agregan nada en el campo de pedido, se tomará como 'Sin orden, e incluso que se llenaran los pedidos desde la terminal. Pero terminé descartando eso o por redundante o porque estaba mejor en GUI.

Conclusión

Después de trabajar en este proyecto, puedo decir que ha sido una experiencia bastante desafiante. Para alguien como yo, que no tiene mucha experiencia en programación, entender y crear un sistema de este tipo requirió un esfuerzo considerable.

La concurrencia en la programación, especialmente cuando se trabaja con hilos, añade un nivel de complejidad adicional. Coordinar la interacción entre diferentes partes del programa y asegurarse de que todo funcione de manera sincronizada es una tarea que requiere cuidado y atención.

Además, el diseño de la interfaz gráfica de usuario también presentó sus propios desafíos. Aprender a utilizar la biblioteca tkinter y crear una interfaz que sea fácil de entender y utilizar para los usuarios fue un proceso que demandó tiempo y paciencia.