UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Seminario de solución de problemas de Sistemas Operativos
Profesor: Axel Daniel Saldivar Zaldivar
D07

Actividad 9

Práctica
Problemas de Concurrencia y Sincronización

Daniel Gaitan Chavez
219294005
2025B
4 Noviembre de 2025

Introducción

La **concurrencia** es un pilar fundamental de los sistemas operativos modernos, permitiendo que múltiples procesos progresen simultáneamente para maximizar el uso del CPU. Sin embargo, el acceso concurrente a recursos compartidos introduce riesgos críticos como las **Condiciones de Carrera** (donde el resultado depende del orden de ejecución) y los **Interbloqueos** (Deadlocks).

El objetivo de esta práctica es implementar soluciones robustas para los tres problemas clásicos de sincronización: **El Barbero Dormilón**, **La Cena de los Filósofos** y **Productor-Consumidor**. Para ello, se han desarrollado algoritmos en Python utilizando la librería threading para la gestión de hilos, y tkinter para crear una interfaz gráfica que permita visualizar en tiempo real el comportamiento de los semáforos, los bloqueos (mutex) y los cambios de estado de los procesos, superando la abstracción de la consola de comandos.

Los enlaces a todos los codigos usados aquí estan en el repositorio de Github junto con un Readme informativo con GIF

https://github.com/DanielGaitan1/Sistemas-operativos

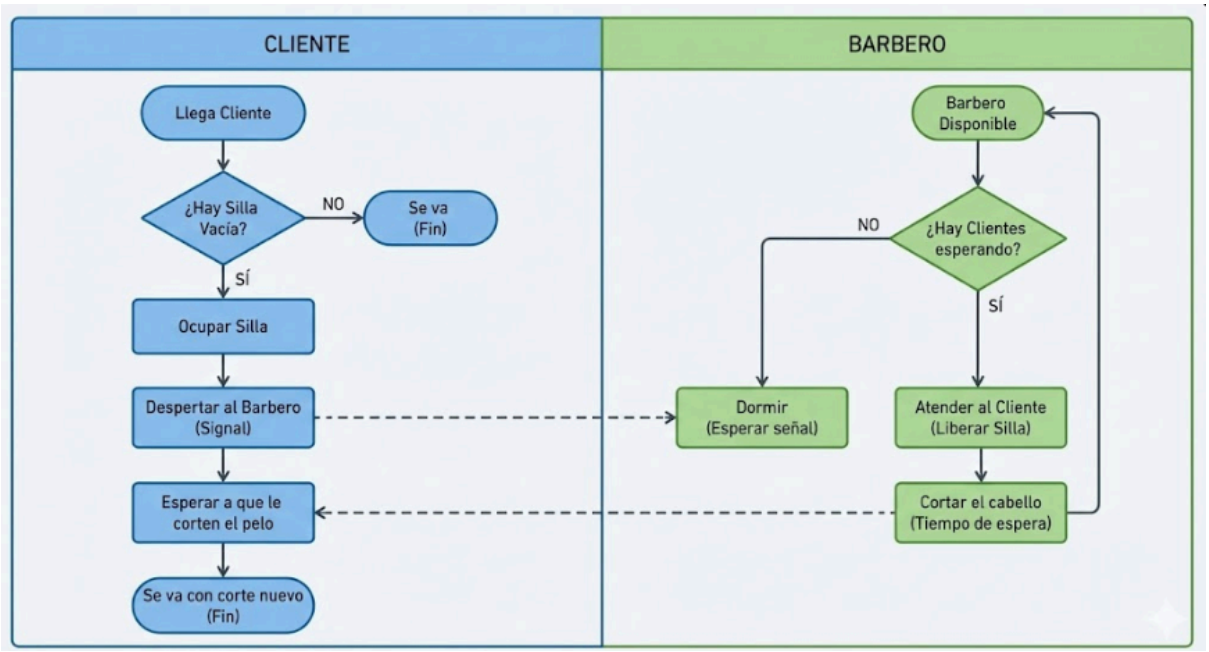
Desarrollo

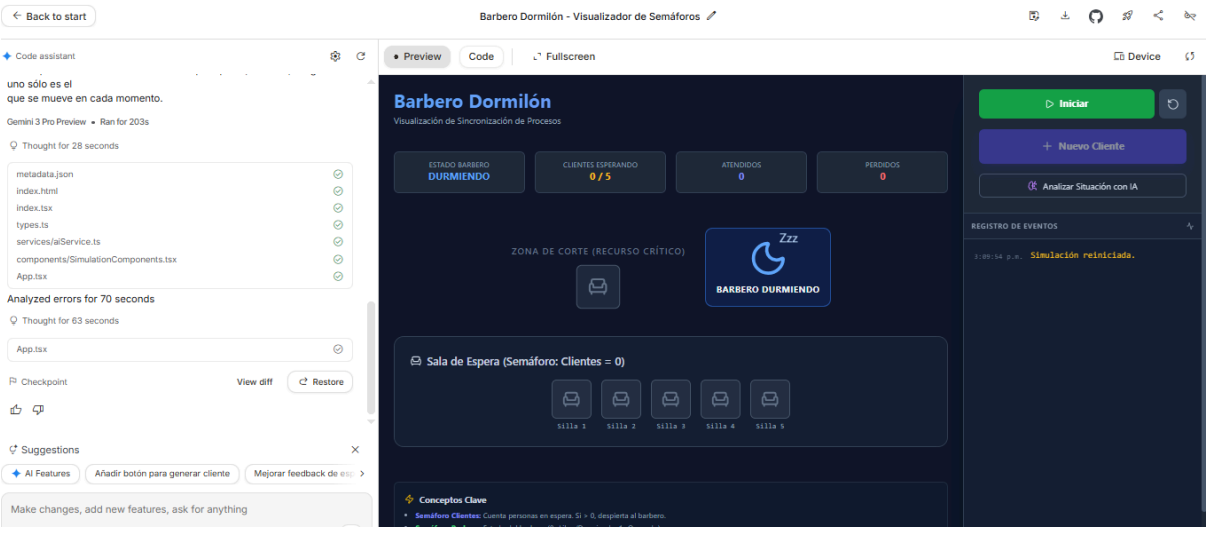
1. Barbero Dormilón

Descripción del Problema: Este escenario modela una situación de servicio con capacidad limitada. Tenemos un barbero (servidor), una silla de corte (recurso crítico) y una sala de espera con N sillas (buffer limitado). El desafío es coordinar que el barbero duerma si no hay clientes (para no desperdiciar CPU en espera activa) y que los clientes se vayan si la sala de espera está llena.

Implementación y Lógica: Se implementó una simulación visual donde los hilos representan al barbero y a los clientes generados aleatoriamente.

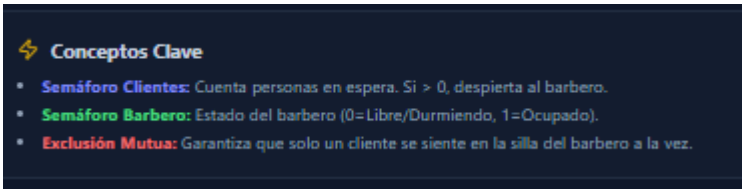
- **Sincronización:** Se utilizaron dos semáforos (sem_clientes_listos y sem_barbero_listo) para coordinar el "despertar" de los procesos.
- **Exclusión Mutua:** Se usó un objeto Lock (Mutex) para proteger la variable compartida clientes_esperando, asegurando que dos clientes no intenten ocupar la misma silla simultáneamente.
- **Visualización:** La interfaz muestra gráficamente el estado de las sillas (Gris=Vacía, Azul=Ocupada) y el estado del barbero (Rojo=Durmiendo, Verde=Trabajando), permitiendo verificar visualmente que nunca se excede la capacidad de la sala.

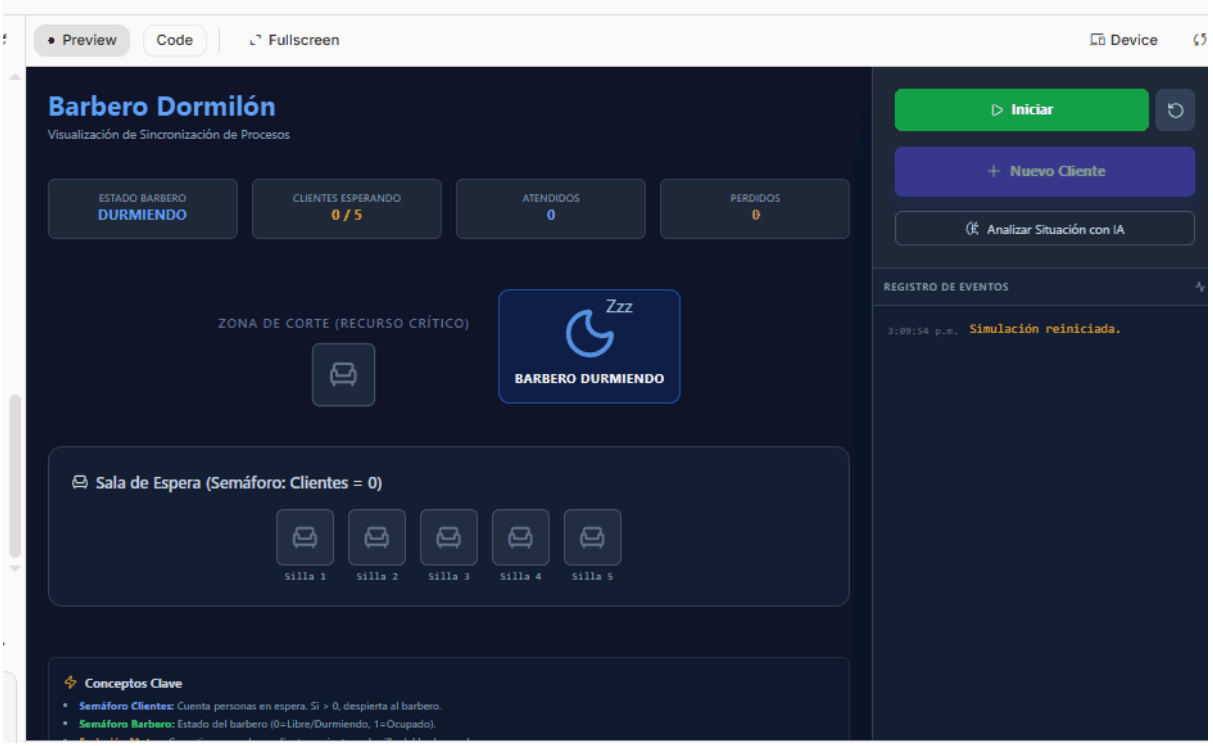




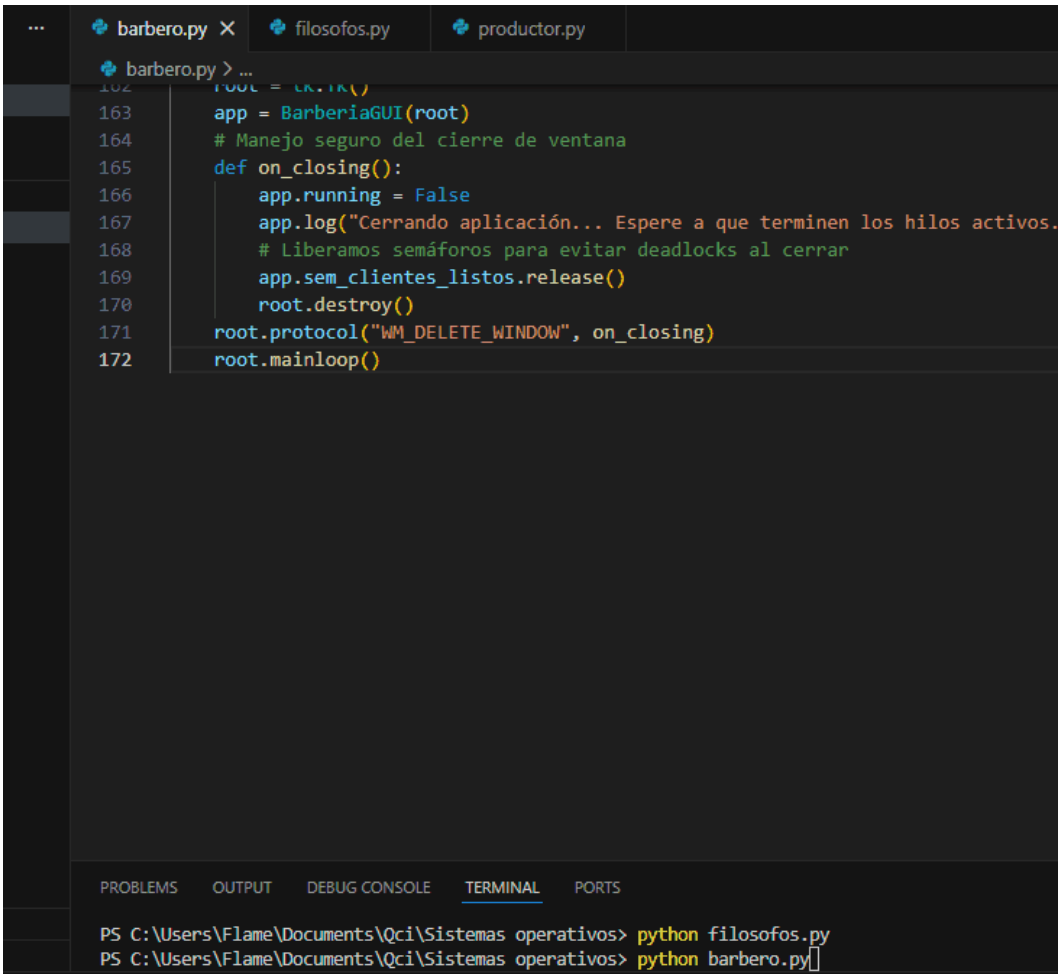
Nota: Para el desarrollo de la lógica visual y la depuración de hilos, se utilizaron herramientas de asistencia de código como Google AI Studio.


Barbero Dormilón
Visualización de Sincronización de Procesos
ESTADO BARBERO
DURMIENDO
CLIENTES ESPERANDO
0 / 5
ATENDIDOS
0
PERDIDOS
0
ZONA DE CORTE (RECURSO CRÍTICO)
Zzz
BARBERO DURMIENDO
Sala de Espera (Semáforo: Clientes = 0)
Silla 1
Silla 2
Silla 3
Silla 4
Silla 5

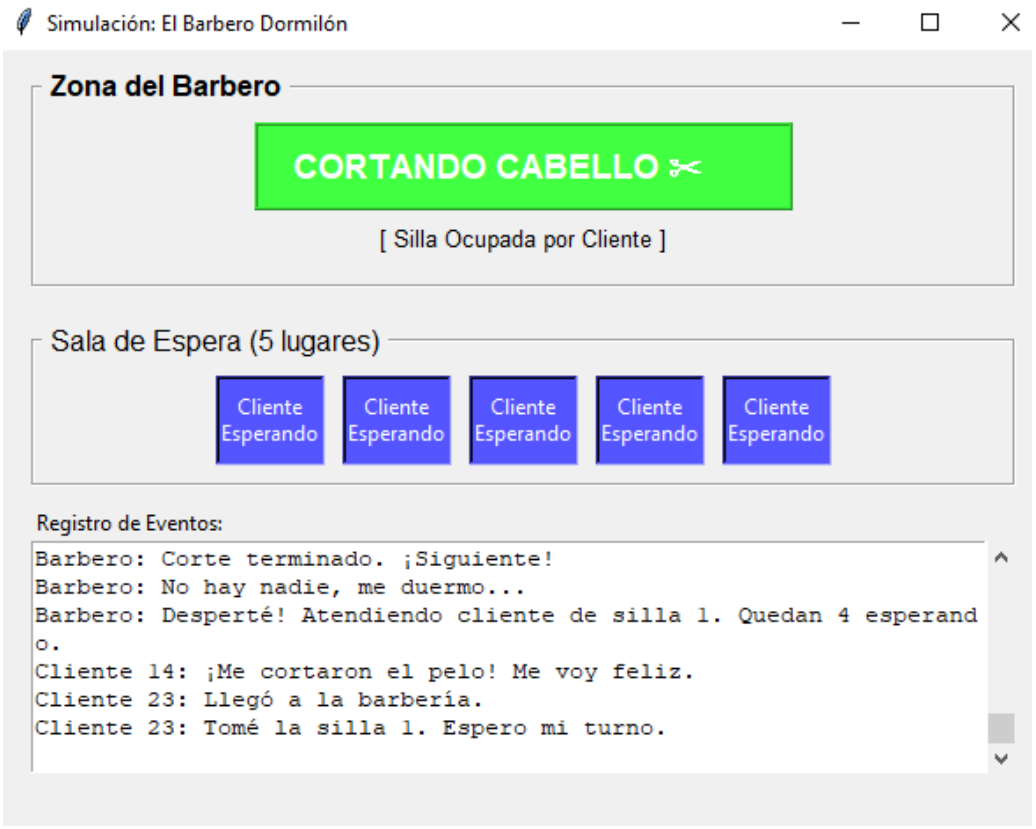

Conceptos Clave
Semáforo Clientes: Cuenta personas en espera. Si > 0, despierta al barbero.
Semáforo Barbero: Estado del barbero (0=Libre/Durmiendo, 1=Ocupado).
Exclusión Mutua: Garantiza que solo un cliente se siente en la silla del barbero a la vez.

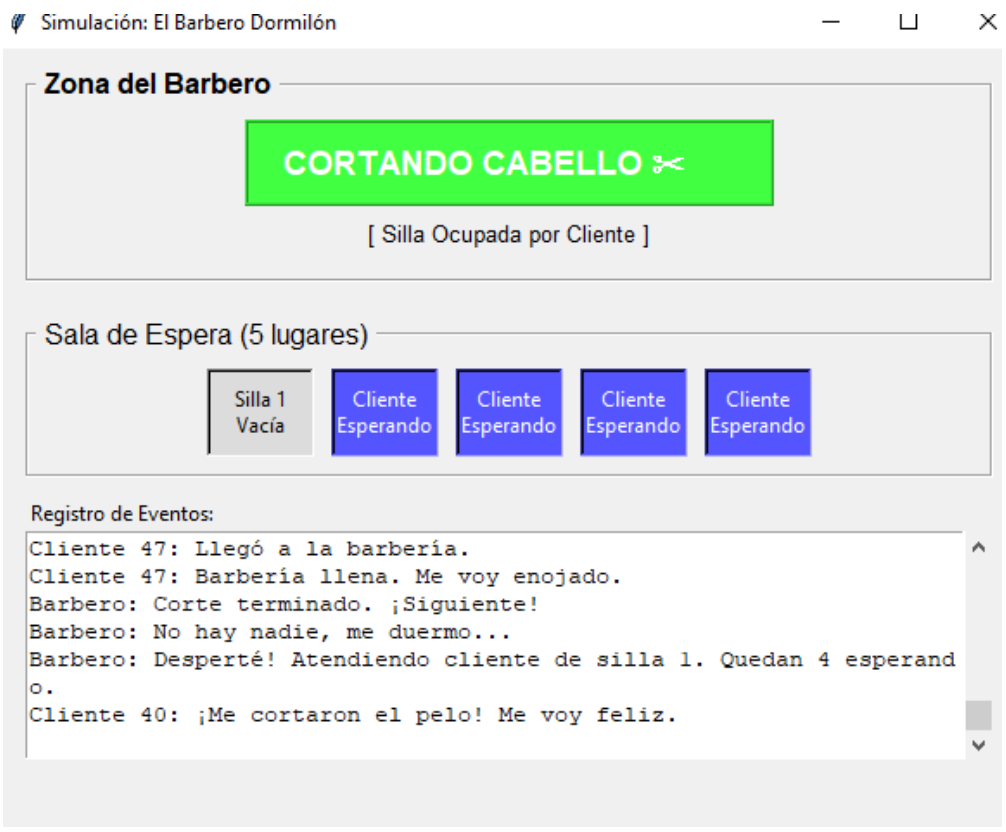

Preview
Code
Fullscreen
Device
Barbero Dormilón
Visualización de Sincronización de Procesos
Iniciar
Nuevo Cliente
Analizar Situación con IA
REGISTRO DE EVENTOS
3:09:54 p.m. Simulación reiniciada.
ESTADO BARBERO
DURMIENDO
CLIENTES ESPERANDO
0 / 5
ATENDIDOS
0
PERDIDOS
0
ZONA DE CORTE (RECURSO CRÍTICO)
Zzz
BARBERO DURMIENDO
Sala de Espera (Semáforo: Clientes = 0)
Silla 1
Silla 2
Silla 3
Silla 4
Silla 5
Conceptos Clave
Semáforo Clientes: Cuenta personas en espera. Si > 0, despierta al barbero.
Semáforo Barbero: Estado del barbero (0=Libre/Durmiendo, 1=Ocupado).

GIF que demuestra el proceso que se realizó en TKINTER



Capturas de pantalla


Simulación: El Barbero Dormilón
Zona del Barbero
CORTANDO CABELLO ✂
[Silla Ocupada por Cliente]
Sala de Espera (5 lugares)
Cliente Esperando
Cliente Esperando
Cliente Esperando
Cliente Esperando
Cliente Esperando
Registro de Eventos:
Barbero: Corte terminado. ¡Siguiente!
Barbero: No hay nadie, me duermo...
Barbero: Desperté! Atendiendo cliente de silla 1. Quedan 4 esperando.
Cliente 14: ¡Me cortaron el pelo! Me voy feliz.
Cliente 23: Llegó a la barbería.
Cliente 23: Tomé la silla 1. Espero mi turno.


Simulación: El Barbero Dormilón
Zona del Barbero
CORTANDO CABELLO ✂
[Silla Ocupada por Cliente]
Sala de Espera (5 lugares)
Silla 1 Vacía
Cliente Esperando
Cliente Esperando
Cliente Esperando
Cliente Esperando
Registro de Eventos:
Cliente 47: Llegó a la barbería.
Cliente 47: Barbería llena. Me voy enojado.
Barbero: Corte terminado. ¡Siguiente!
Barbero: No hay nadie, me duermo...
Barbero: Desperté! Atendiendo cliente de silla 1. Quedan 4 esperando.
Cliente 40: ¡Me cortaron el pelo! Me voy feliz.

2. La Cena de los Filósofos

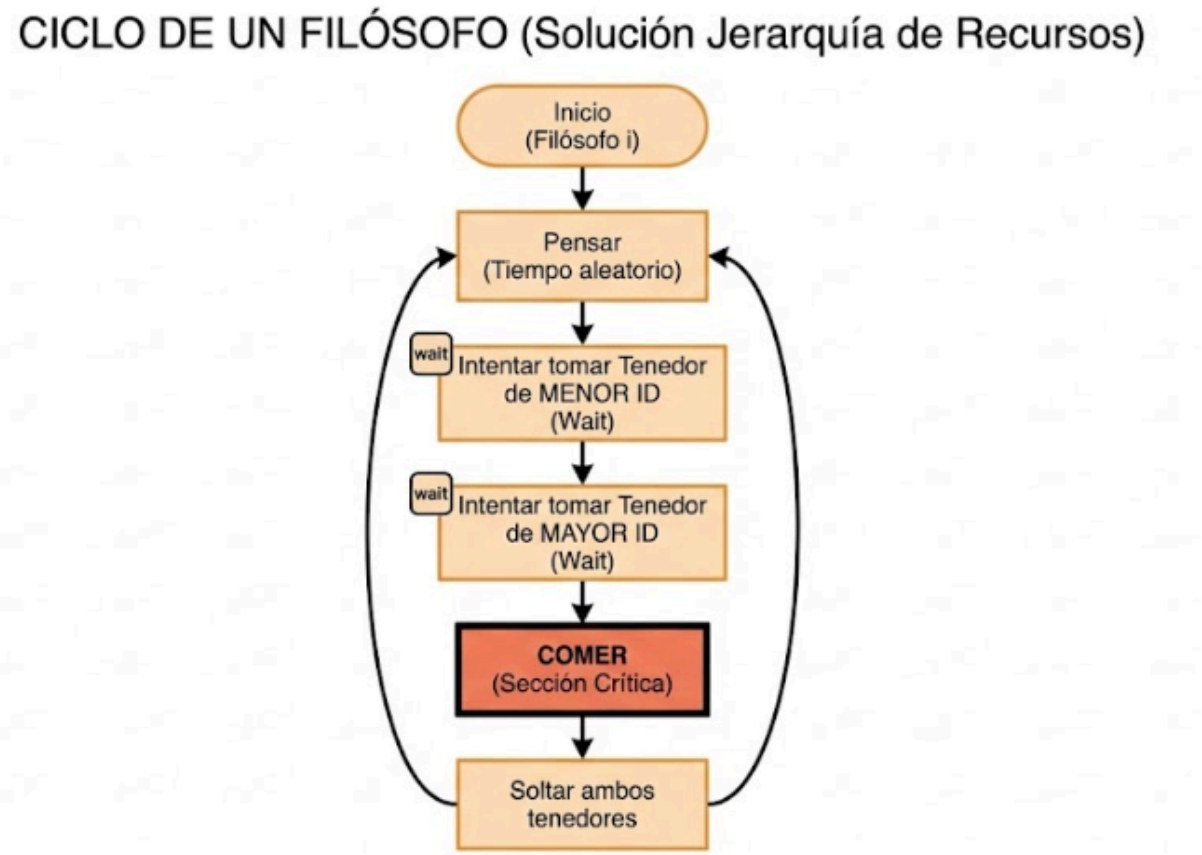
Descripción del Problema: Cinco filósofos se sientan alrededor de una mesa y alternan entre pensar y comer. Para comer, necesitan dos tenedores (recursos compartidos a su izquierda y derecha). El problema clásico es evitar el **Deadlock** (todos toman el tenedor izquierdo al mismo tiempo y esperan eternamente el derecho) y la **Inanición** (un filósofo nunca logra comer).

Implementación y Solución al Deadlock: La interfaz gráfica representa la mesa redonda, cambiando el color de los filósofos según su estado: Blanco (Pensando), Amarillo (Hambriento) y Verde (Comiendo).

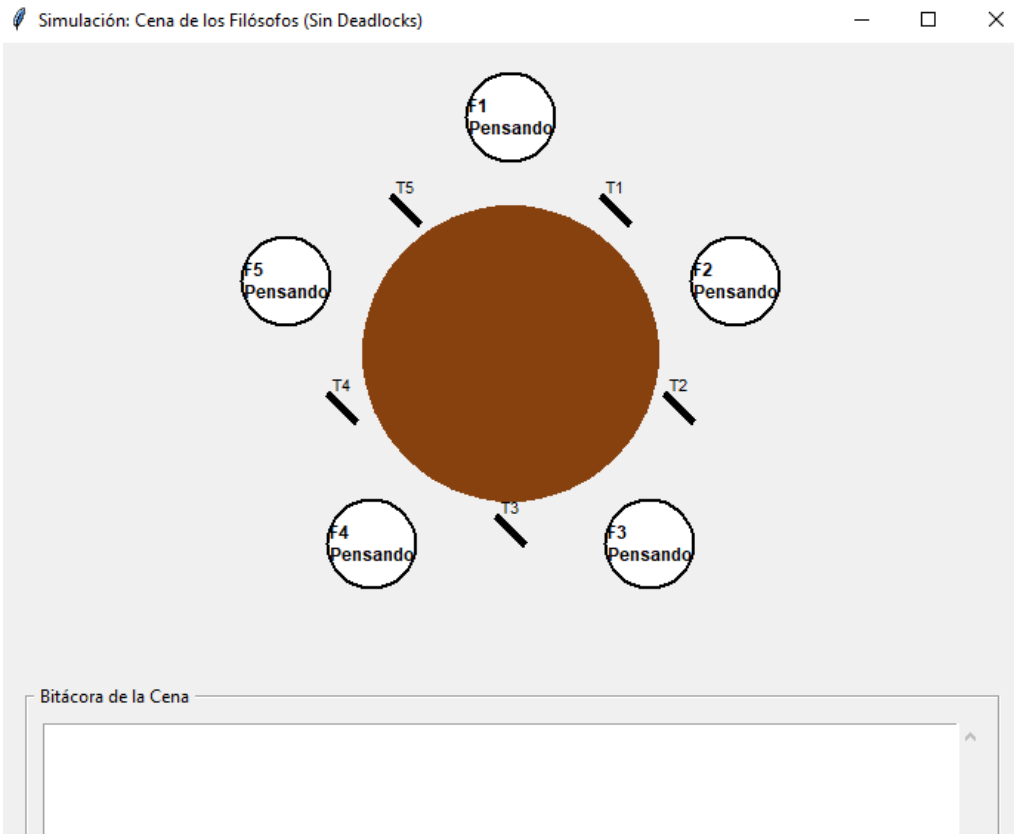
Detalles Técnicos (Solución SDE): Para garantizar que el sistema nunca se trabe, no permití que los filósofos tomaran los tenedores aleatoriamente. Implementé una solución basada en **Jerarquía de Recursos**:

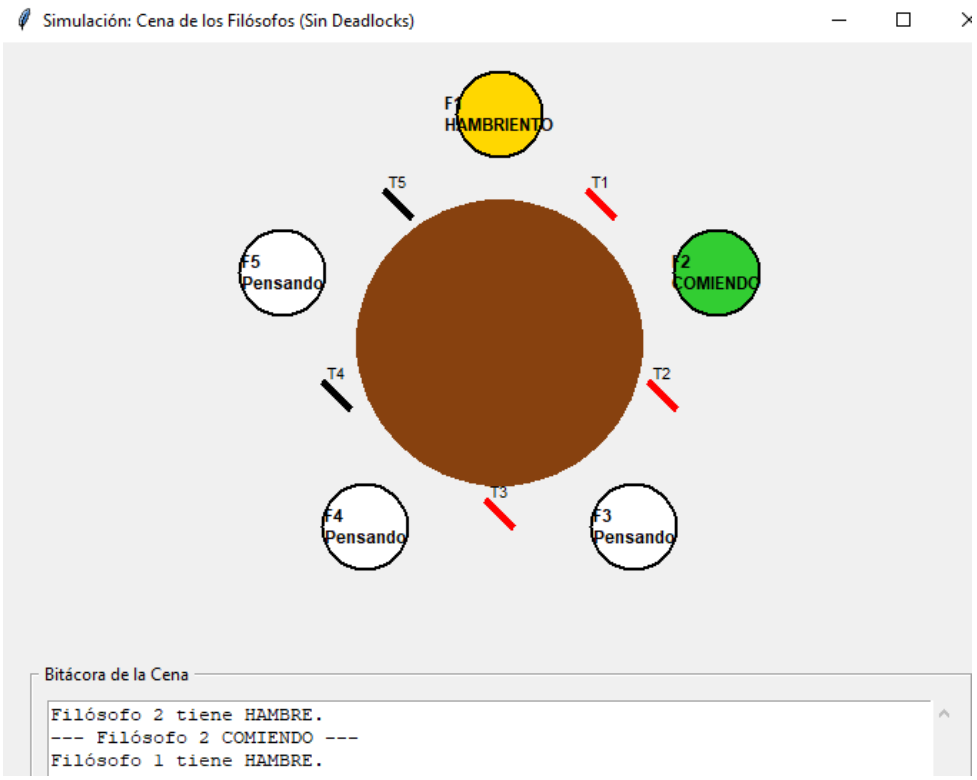
> *"Los filósofos no toman 'izquierda y derecha' ciegamente. Siempre intentan adquirir primero el bloqueo (Mutex) del tenedor con el **número de identificación menor**. Esto rompe matemáticamente la posibilidad de 'Espera Circular', que es una de las 4 condiciones necesarias para el Deadlock según William Stallings."*

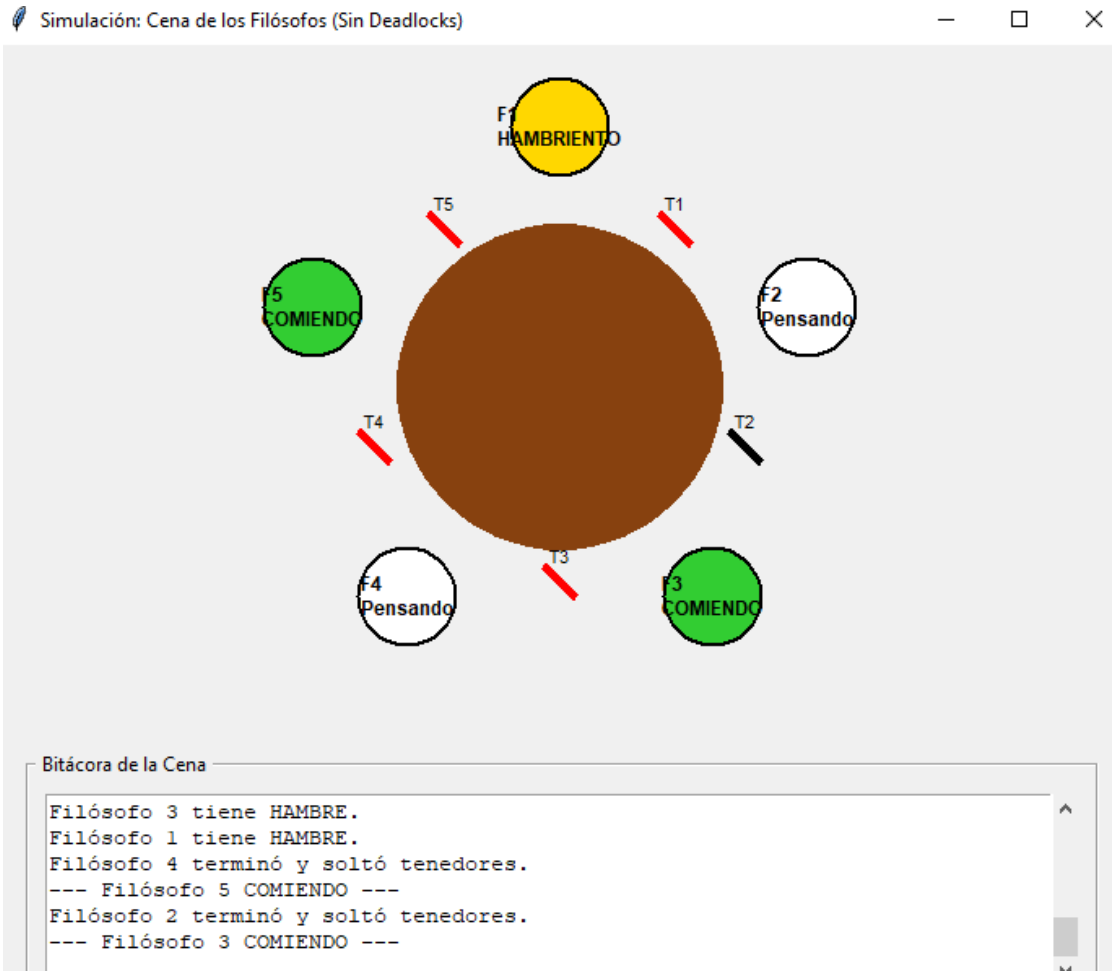
Gracias a esta lógica, la simulación puede correr indefinidamente sin congelarse.

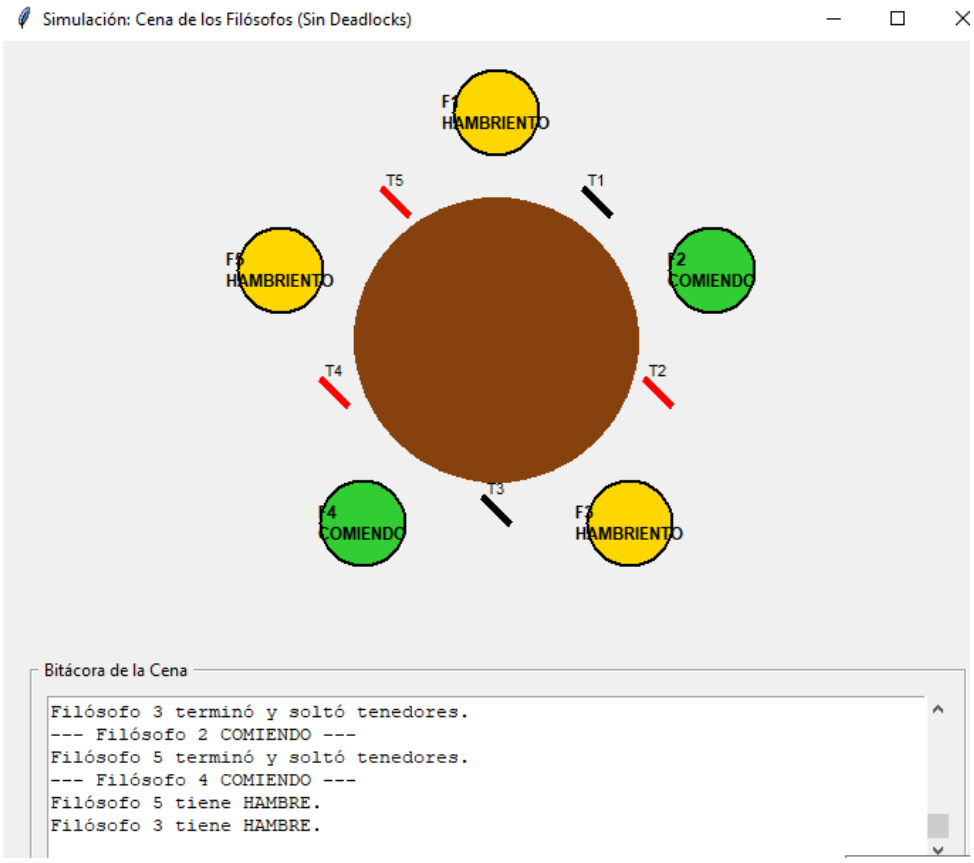


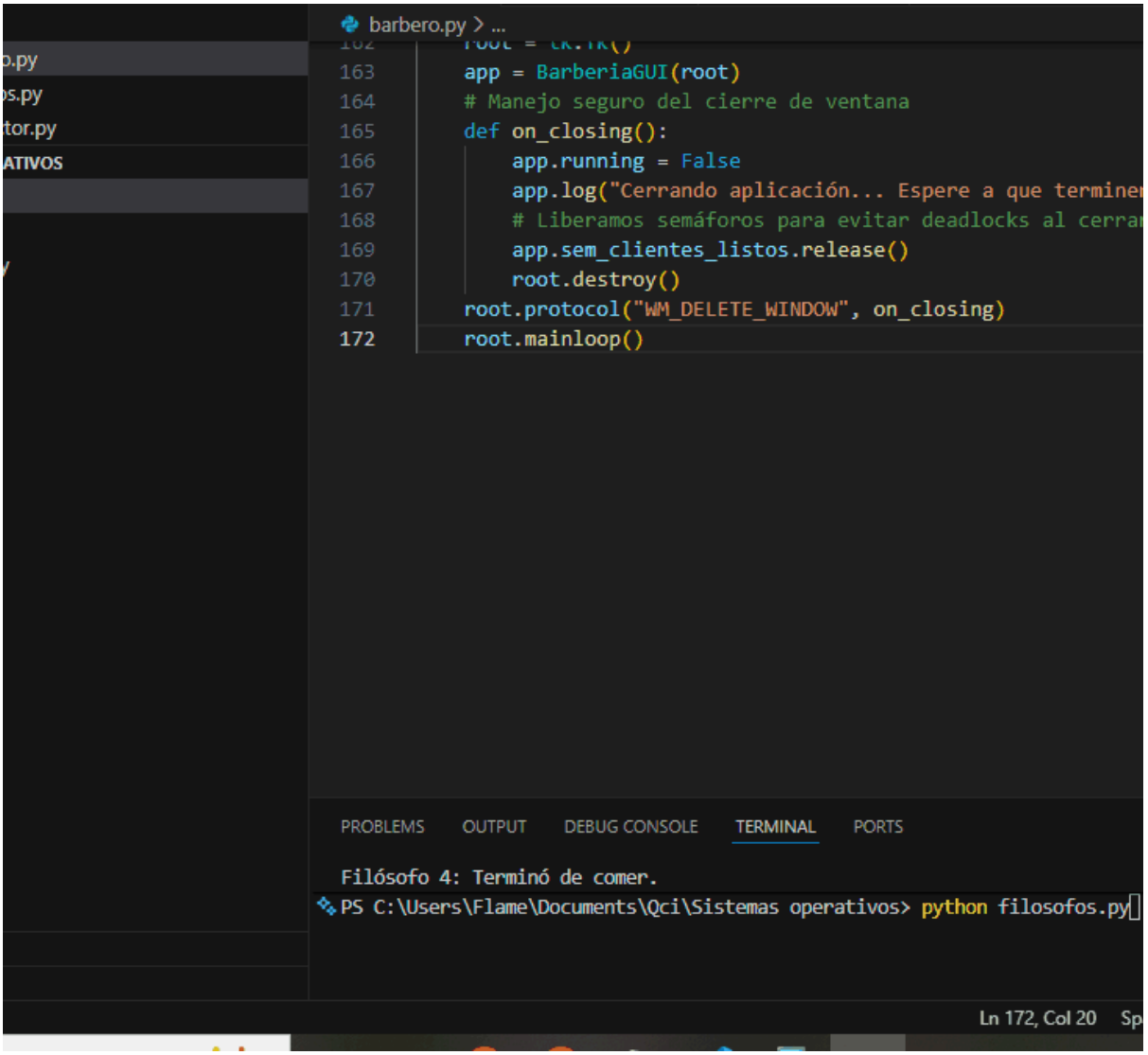
Capturas de pantalla






Simulación: Cena de los Filósofos (Sin Deadlocks)
F1
HAMBRIENTO
T5
T1
F5
COMIENDO
F2
Pensando
T4
T2
T3
F4
Pensando
F3
COMIENDO
Bitácora de la Cena
Filósofo 3 tiene HAMBRE.
Filósofo 1 tiene HAMBRE.
Filósofo 4 terminó y soltó tenedores.
--- Filósofo 5 COMIENDO ---
Filósofo 2 terminó y soltó tenedores.
--- Filósofo 3 COMIENDO ---


Simulación: Cena de los Filósofos (Sin Deadlocks)
F1
HAMBRIENTO
T5
T1
F5
HAMBRIENTO
F2
COMIENDO
T4
T2
T3
F4
COMIENDO
F3
HAMBRIENTO
Bitácora de la Cena
Filósofo 3 terminó y soltó tenedores.
--- Filósofo 2 COMIENDO ---
Filósofo 5 terminó y soltó tenedores.
--- Filósofo 4 COMIENDO ---
Filósofo 5 tiene HAMBRE.
Filósofo 3 tiene HAMBRE.



---

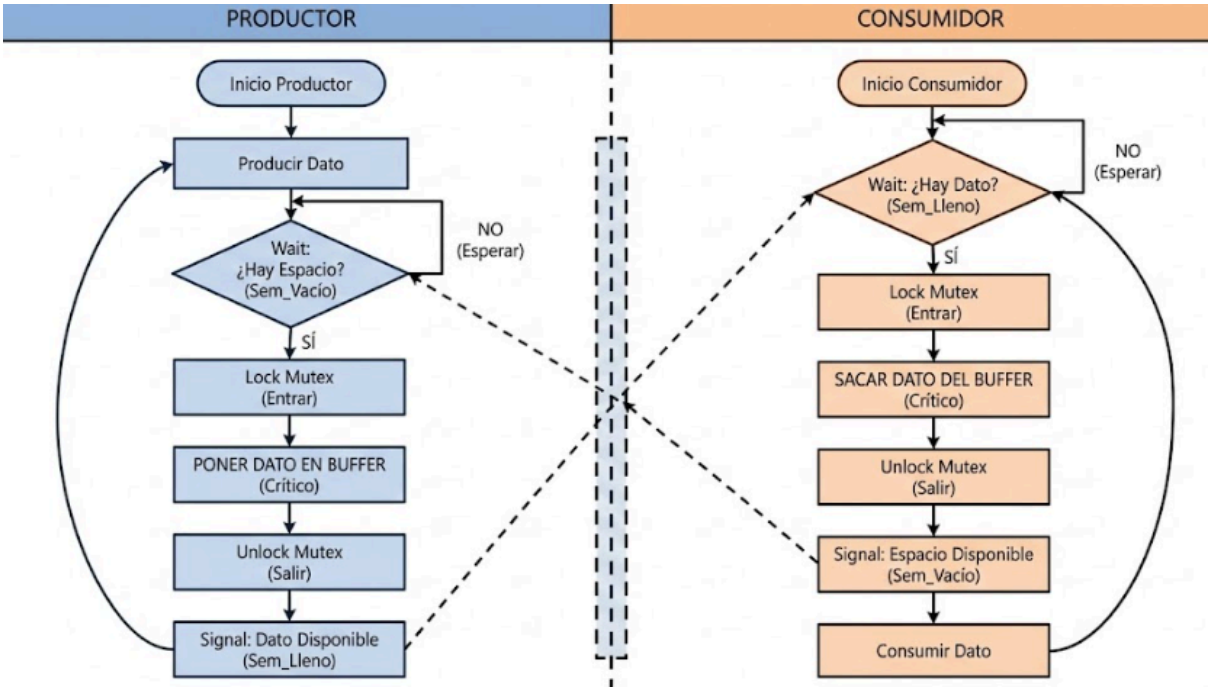
3. Productor - Consumidor

Descripción del Problema: Este problema modela dos procesos que comparten un buffer de tamaño fijo. El **Productor** genera datos y los coloca en el buffer, mientras que el **Consumidor** los retira. El reto es asegurar que el productor no intente agregar datos si el buffer está lleno (Overflow) y que el consumidor no intente retirar si está vacío (Underflow).

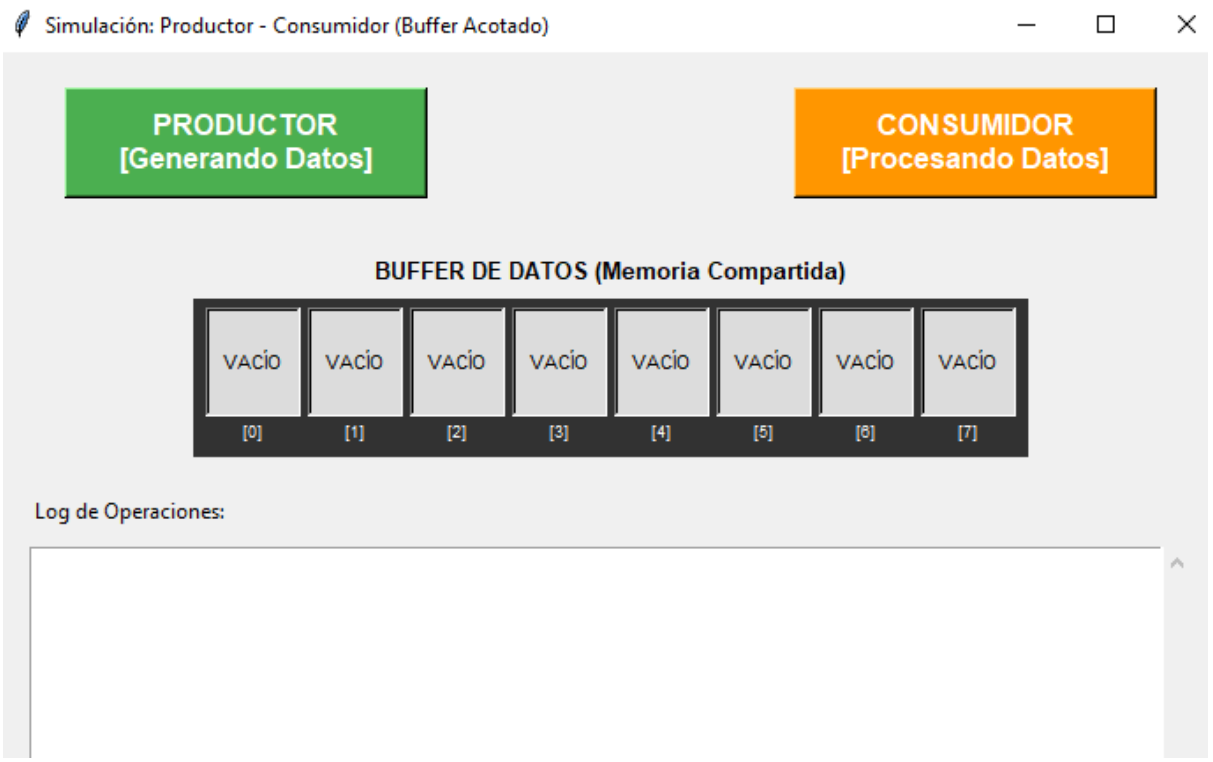
Implementación y Lógica: Se diseñó una interfaz que simula una cinta transportadora (Cola Circular o FIFO).

- **Semáforos Contadores:**
  - sem_espacios_vacios: Inicializado en N (tamaño buffer). Decrementa cuando el productor pone algo.
  - sem_items_disponibles: Inicializado en 0. Incrementa cuando hay datos listos para consumir.

- **Manejo de Estados:** La interfaz visualiza claramente los estados de bloqueo. Si el buffer se llena, el Productor cambia a color Rojo (Estado de Espera/Bloqueado) hasta que el consumidor libera un espacio, demostrando visualmente el concepto de sincronización condicional.



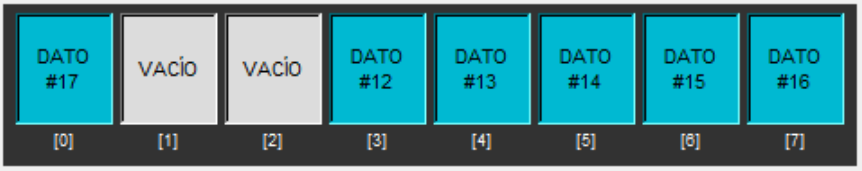
Capturas de pantalla



PRODUCTOR
[Trabajando]

CONSUMIDOR
[Procesando]

BUFFER DE DATOS (Memoria Compartida)



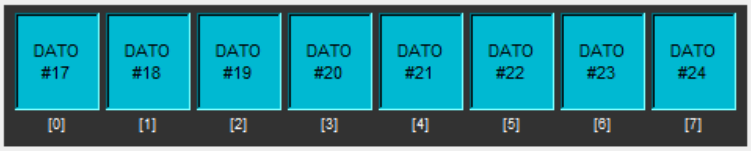
Log de Operaciones:

```
Productor: Creó #9 en slot [0]
Productor: Creó #10 en slot [1]
Consumidor: Retiró #6 del slot [5]
Productor: Creó #11 en slot [2]
Consumidor: Retiró #7 del slot [6]
Consumidor: Retiró #8 del slot [7]
Productor: Creó #12 en slot [3]
Productor: Creó #13 en slot [4]
Productor: Creó #14 en slot [5]
Consumidor: Retiró #9 del slot [0]
Productor: Creó #15 en slot [6]
Consumidor: Retiró #10 del slot [1]
Productor: Creó #16 en slot [7]
Productor: Creó #17 en slot [0]
Consumidor: Retiró #11 del slot [2]
```

PRODUCTOR
[ESPERANDO ESPACIO]

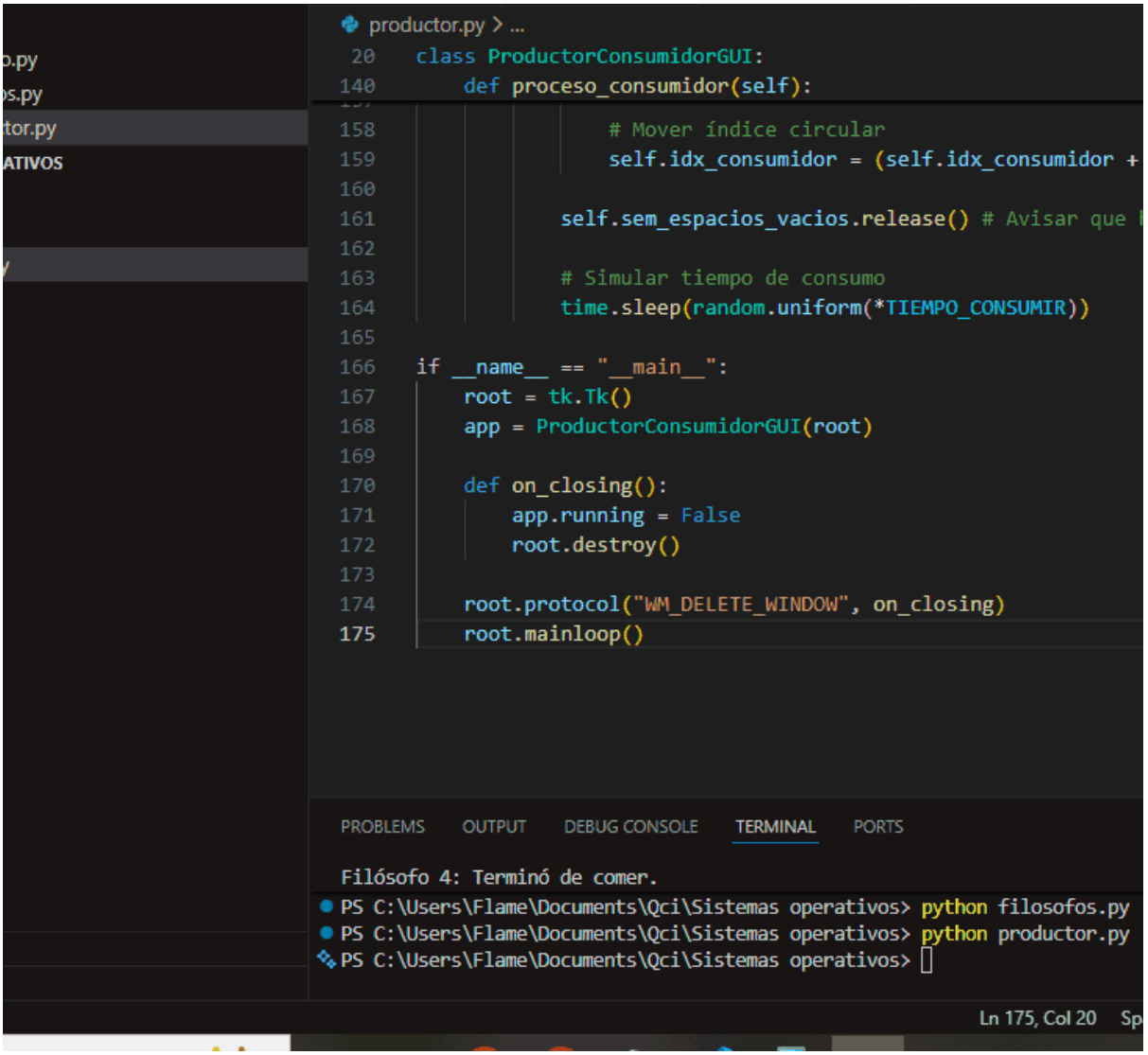
CONSUMIDOR
[Procesando]

BUFFER DE DATOS (Memoria Compartida)



Log de Operaciones:

```
Productor: Creó #16 en slot [7]
Productor: Creó #17 en slot [0]
Consumidor: Retiró #11 del slot [2]
Productor: Creó #18 en slot [1]
Productor: Creó #19 en slot [2]
Consumidor: Retiró #12 del slot [3]
Consumidor: Retiró #13 del slot [4]
Productor: Creó #20 en slot [3]
Productor: Creó #21 en slot [4]
Consumidor: Retiró #14 del slot [5]
Productor: Creó #22 en slot [5]
Consumidor: Retiró #15 del slot [6]
Productor: Creó #23 en slot [6]
Consumidor: Retiró #16 del slot [7]
Productor: Creó #24 en slot [7]
```



Conclusión

La realización de esta práctica permitió visualizar conceptos abstractos de sistemas operativos que son difíciles de apreciar solo con código en consola. A través de la implementación gráfica con tkinter, pude observar en tiempo real cómo los **semáforos** actúan como mecanismos de señalización para despertar procesos dormidos, y cómo los **mutex** protegen las secciones críticas para mantener la integridad de los datos.

Como aprendizaje clave para mi formación como ingeniero, destaco la importancia de las estrategias de diseño para prevenir **Deadlocks**. Comprobé que soluciones simples como la jerarquización de recursos (usada en los Filósofos) son más efectivas y robustas que intentar detectar y recuperar fallos una vez ocurridos. Estas habilidades son directamente aplicables al desarrollo de software backend escalable, donde la gestión eficiente de hilos y recursos compartidos determina el rendimiento de la aplicación.