Proyecto 1: Concurrencia en un Sistema de Reservas de Canchas Deportivas

Saldaña Navarrete Andrea Salgado Cruz Emiliano Roman

12 de abril de 2025

1. Descripción del problema

En este proyecto abordamos la problemática de reservar canchas deportivas en un entorno real, donde múltiples usuarios (simulados por hilos) compiten por horarios disponibles. Hemos elegido este problema porque, tanto en nuestra comunidad como en nuestra universidad, surgen conflictos cuando varios compañeros desean usar la misma cancha (fútbol, baloncesto, etc.) y no existe un sistema seguro que evite reservas duplicadas. Como consecuencia, pueden ocurrir desacuerdos, confusiones y hasta discusiones innecesarias. Nuestro objetivo es demostrar cómo, mediante el uso de mecanismos de concurrencia y sincronización, podemos resolver esta situación de manera ordenada y eficaz, asignando horarios de forma justa y evitando las condiciones de carrera.

Los principales problemas de **concurrencia** que deseamos evitar son:

- *Race condition*: Dos o más usuarios intentan reservar la misma cancha al mismo horario simultáneamente.
- *Inconsistencia de interfaz*: Que un usuario vea un horario como "libre" cuando en realidad ya ha sido tomado.
- *Reservas fantasma*: Un usuario considera disponible un horario, pero al momento de reservar, otro hilo ya lo ocupó.

Además, aunque existen eventos concurrentes donde el orden no es importante (por ejemplo, reservas en canchas u horarios totalmente distintos), la complejidad surge cuando los usuarios compiten por el mismo recurso y pueden incluso tener distintos niveles de **prioridad** (por ejemplo, usuarios VIP que desbancan la reserva de usuarios normales).

2. Mecanismo de sincronización empleado

Para garantizar la **exclusión mutua** al acceder o modificar la estructura de datos que representa las canchas y horarios, utilizamos la primitiva `Lock` de la librería estándar `threading` de Python. A continuación, describimos sus usos principales:

- *Reserva de una cancha*: El método `reservar` adquiere el `Lock` antes de verificar la disponibilidad y actualizar el estado.
- *Cancelación y/o modificación*: Estos métodos también adquieren el mismo `Lock` para evitar `race conditions`, garantizando que solo un hilo manipule a la vez la estructura global.
- **Manejo de prioridades (VIP)**: Para quienes cuenten con una prioridad más alta, el sistema verifica si la cancha está ocupada por un usuario de prioridad menor; en tal caso, la reserva se transfiere al VIP.

Para asegurar que el candado se libere automáticamente al finalizar la sección crítica, incluso si sucede alguna excepción, empleamos la forma

```
with self.lock:
    # Sección crítica
```

Gracias a ello, nuestras operaciones sobre el estado de las canchas se ejecutan de forma *atómica*.

Adicionalmente, usamos un segundo *lock* para los *logs*, para mantener el registro ordenado cuando varios hilos escriben simultáneamente en el archivo de bitácora.

3. Lógica general de la operación

El flujo global del sistema es:

1. **Inicialización**:
   - Creamos un objeto (`SistemaReservas`), responsable de la matriz/estructura que mantiene el estado de las canchas y horarios, y de los `locks` de sincronización (uno para la matriz de canchas y otro para el archivo de registro).
   - Inicializamos la matriz con valores que indiquen que todas las canchas y horarios están libres.
2. **Creación de hilos**:
   - Instanciamos varios hilos de tipo `Usuario`, cada uno con un identificador (`user_id`) y un atributo de *prioridad* que indicará si es VIP o usuario normal.
   - Cada hilo "simula" a un usuario que realizará acciones (`reservar`, `cancelar`, `modificar`) de manera concurrente.
3. **Ejecución de hilos**:
   - Cada hilo, en su método `run()`, realiza un conjunto de acciones. Por ejemplo, primero `reservar`, luego duerme unos segundos (`time.sleep`), y eventualmente `cancelar` o `modificar` su reserva.
   - Estas acciones invocan métodos protegidos con `lock` en `SistemaReservas`, donde se revisa el nivel de prioridad en caso de un conflicto de reserva.
4. **Interfaz de usuario (*Tkinter*)**:
   - En el hilo principal, se invoca una ventana de *Tkinter* que muestra en tiempo real la disponibilidad de canchas y horarios.
   - Esta interfaz se actualiza periódicamente leyendo la estructura compartida.
   - La interfaz muestra no solo si una cancha está libre u ocupada, si no que también empleamos colores distintos para diferencias reservas VIP de reservas normales. Además en la misma ventana se despliega una sección en la que se describe cada acción que ocurre (reservas, desbancamientos, cancelaciones), y además se muestran estadísticas de cuántas reservas exitosas ha hecho cada usuario.
5. **Finalización**:
   - Tras un número de acciones o al cerrarse la interfaz, se detiene el programa.
   - Se genera un **archivo de log** que registra todos los intentos de reserva/cancelación y sus resultados (*éxito* o *fracaso*).
   - Se presentan **estadísticas finales**, como el total de reservas exitosas, cuántas se desbancaron por un VIP, etc.

4. Descripción algorítmica del avance de cada hilo

Cada **hilo Usuario** sigue este esquema:

1. **Inicio**:
   - Recibe su `user_id`, la referencia a `SistemaReservas`, su `prioridad` y una lista de posibles acciones (`reservar`, `cancelar`, `modificar`, ...).
2. **Ciclo de acciones** (puede repetirse N veces):
   a) **Elegir acción**: Toma al azar (o según un guion) la operación a realizar.
   b) **Elegir cancha y horario**: Selecciona aleatoriamente la cancha y el horario a utilizar.
   c) **Invocar el método de `SistemaReservas`**:
      - `reservar(user_id, priority, cancha, horario)`

- `cancelar(user_id, cancha, horario)`
- `modificar(user_id, priority, cancha_orig, hora_orig, cancha_dest, hora_dest)`

d) **Retardo**: Usamos `time.sleep` (0.5 a 2 segundos) para simular "tiempo de decisión" y forzar **concurrencia real**.

e) **Log**: Se escribe el resultado en un archivo, indicando si fue exitoso o fallido y, en caso de conflicto, si se "desbancó" a otro usuario por prioridad.

3. **Fin**:
   - Al completar su número de acciones asignadas, el hilo termina (sale de `run()`).

Debido a que varios hilos repiten este patrón en *paralelo*, surgen naturalmente las condiciones de carrera, resueltas gracias al uso de `lock` y a la comparación de `prioridad`.

5. Descripción de la interacción entre hilos

- **Hilos sin comunicación directa**: No se envían mensajes entre ellos, sino que interactúan a través de `SistemaReservas`.
- **Acceso concurrente con prioridades**: Cuando un hilo VIP llama `reservar`, el sistema revisa si la cancha-horario está libre o si está ocupada por alguien con *menor* prioridad; de ser así, "desbanca" esa reserva y otorga el espacio al VIP.
- **Estado compartido**: Consiste en la matriz que marca cada *cancha* y *horario* como ocupado o libre (y por quién), más su prioridad. También se lleva un *log* estructurado para registrar acciones.

```
[Usuario 1 (VIP)] ->(Lock) ->[SistemaReservas] <- (Lock) <- [Usuario 2 (Normal)]
```

6. Entorno de desarrollo

6.1. Lenguaje y versión

- **Lenguaje**: Python (versión 3.10 o superior).
- Librería estándar `threading` para crear hilos y utilizar `Lock`, `Semaphore`, etc.
- `time`, `random`, `tkinter` se incluyen también en la librería estándar de Python.

6.2. Bibliotecas utilizadas

- `threading`: Manejo de hilos (`Thread`) y mecanismos de sincronización (`Lock`).
- `time` y `random`: Para retardos y selección aleatoria de acciones, canchas, horarios, etc.
- `tkinter`: Para la interfaz gráfica de usuario, donde se representa la ocupación de cada cancha.

6.3. Sistemas operativos y pruebas

- **Windows 11 (x64)**: Desarrollado y probado en esta plataforma, con Python 3.10.

6.4. Ejecución

Para ejecutar nuestro programa, basta con:

1. Ubicarse en la carpeta donde se encuentran los archivos de código fuente (`main.py`, `sistema_reservas.py`, `usuario.py`, `gui.py`).
2. Asegurarse de tener **Python 3.10 o superior**.
3. Ejecutar:

```
python main.py
```

4. Al cabo de unos instantes, aparecerá la ventana de `tkinter` con la interfaz gráfica.

Se generará también el archivo `acciones.log` con el registro de todas las operaciones (reservas, cancelaciones, desbancamientos, etc.) realizadas por los hilos de usuario.

7. Resultados y conclusiones

7.1. Resultados obtenidos

- **Archivo de log**: Se generan registros con cada intento de reserva, cancelación o modificación, detallando el ID y prioridad del usuario. Al revisar, se observa que no se duplican reservas en la misma cancha-horario a menos que la prioridad del reservante sea superior.
- **Interfaz gráfica**: En la ventana de `tkinter`, se visualiza cómo se marcan canchas como "Ocupadas" o "Libres". En caso de que un VIP desplace a un usuario normal, se actualiza la interfaz en tiempo real. Adicional a esto hemos establecido colores a cada usuario, las celdas que estén de color rosado pertenecen a usuarios con prioridad 2, es decir, usuarios VIP, mientras que las celdas que se encuentren coloreadas con un fondo celeste pertenecen a usuarios normales.
- **Estadísticas**: Al final de la ejecución, se obtiene el número de reservas totales, cuántas fueron exitosas y cuántas sufrieron una colisión de prioridad.

8. Conclusiones

Con nuestro proyecto, hemos brindado una solución clara y novedosa a la problemática de reservar canchas deportivas, haciendo uso de `Lock` para evitar condiciones de carrera. Además, ofrecemos al usuario una interfaz intuitiva y atractiva que permite visualizar el estado de cada cancha en tiempo real. El uso de prioridad para usuarios VIP añade una capa de complejidad que ilustra cómo puede manejarse la concurrencia con distintos niveles de acceso, enriqueciendo la experiencia y demostrando la versatilidad de los mecanismos de sincronización en problemas reales.