Universidad Nacional Autónoma de México

Facultad de Ingeniería

Sistemas Operativos | Grupo 8

Tarea 1

Jiménez Olivo Evelin – 321184191

Lara Hernández Emmanuel – 321189536

Fecha: 16 de octubre de 2025

Simulación concurrente: Comedor de Gatos y Ratones

Informe técnico de entrega

1. Descripción general del problema

El objetivo de este trabajo fue desarrollar un programa concurrente que representara la convivencia de dos especies, gatos y ratones, en un comedor con un número limitado de platos. El desafío consiste en garantizar que ambos tipos de animales puedan comer respetando reglas estrictas de convivencia, evitando conflictos, manteniendo la exclusión mutua sobre los platos y asegurando que ninguna especie quede esperando indefinidamente.

Las reglas fundamentales son tres: sólo un animal puede ocupar un plato a la vez; nunca deben comer gatos y ratones al mismo tiempo, ya que si un gato ve a un ratón comiendo debe cazárselo; y finalmente, el sistema debe permitir la alternancia entre especies para evitar inanición.

Para modelar estas reglas se desarrollaron dos versiones del simulador. En la primera, llamada **sin cazar**, se impide completamente que gatos y ratones coincidan; los mecanismos de sincronización evitan el encuentro. En la segunda, denominada **con cazar**, si un ratón intenta comer mientras hay gatos activos, es eliminado inmediatamente, representando la acción de la caza.

2. Lenguaje y entorno de desarrollo

El simulador fue implementado en el lenguaje Python, versión 3.8 o superior, empleando únicamente librerías estándar: `threading` para la gestión de hilos y sincronización, `time` para el control de temporización y `random` para la generación de tiempos de espera aleatorios. El código es completamente portable

y ha sido probado en sistemas Windows y Linux. No requiere instalación de paquetes adicionales ni configuraciones externas.

3. Ejecución del programa

Para ejecutar cualquiera de las dos versiones se utiliza el comando:

```
py gatos_y_ratones-sin_cazar.py
py gatos_y_ratones-cazar.py
```

La primera instrucción ejecuta la versión que evita completamente la coincidencia entre especies, mientras que la segunda habilita la posibilidad de caza.

En la parte final de cada archivo se encuentra la función `simular()`, donde pueden modificarse los parámetros de la simulación, como el número de gatos, ratones, platos, duración de la simulación y semilla aleatoria. Por ejemplo, el valor por defecto de la función:

```
simular(k_gatos=3, l_ratones=5, m_platos=2, duracion=20, seed=7)
```

crea tres gatos, cinco ratones y dos platos, durante veinte segundos de ejecución. El programa imprime en consola el inicio y fin de cada comida, los cambios de turno y, en la versión con caza, los ratones eliminados.

4. Estrategia de sincronización y mecanismos utilizados

La solución se basa en tres mecanismos principales de sincronización: semáforos, mutex (candados) y variables de condición. Estos componentes trabajan en conjunto para mantener la coherencia del estado del sistema y coordinar la interacción de los hilos que representan a cada animal.

Uso de semáforos

El semáforo se utiliza para administrar el acceso a los platos, que son los recursos compartidos del comedor. En la versión sin caza, cada plato está protegido por un semáforo binario (`Semaphore(1)`), asegurando que sólo un hilo, sea gato o ratón, pueda ocuparlo a la vez. En la versión con caza, se emplea un semáforo contable (`Semaphore(m_platos)`), que representa el número total de platos disponibles. Cada vez que un animal empieza a comer, el programa ejecuta `acquire()` para ocupar un plato, y al terminar libera el recurso con `release()`. Con este mecanismo se logra exclusión mutua a nivel de recurso físico, evitando que dos animales se alimenten en el mismo plato simultáneamente.

Uso de mutex (candado)

El mutex o `Lock` es el responsable de proteger el acceso a las variables de estado compartidas, como los contadores de animales activos y en espera (`activos_gatos`, `activos_ratones`, `esperando_gatos`, `esperando_ratones`) y la variable `turno`, que indica qué especie tiene prioridad para comer. El mutex garantiza que sólo un hilo pueda modificar estas variables a la vez, evitando condiciones de carrera y manteniendo la coherencia del estado global. Todo el código crítico que altera estas variables se encuentra dentro de bloques `with cond:`, donde la condición asocia internamente el candado con las operaciones de espera y notificación.

Uso de variables de condición

La `Condition` se utiliza para suspender temporalmente la ejecución de los hilos que no pueden continuar según las reglas. Por ejemplo, si un gato intenta entrar mientras hay ratones comiendo, debe esperar a que la condición de exclusión se libere; y de forma análoga, un ratón debe esperar mientras haya gatos activos. El hilo en espera libera el candado y se bloquea hasta que otro hilo invoque `notify_all()`, lo que ocurre cuando cambia el turno o cuando se libera un plato. De esta manera, las variables de condición permiten coordinar los accesos sin necesidad de bucles de espera activa y garantizan que las especies alternen su uso del comedor.

5. Funcionamiento de las dos versiones

En la versión **sin cazar**, los gatos y ratones nunca coinciden dentro del comedor. Cuando una especie termina su tanda, el turno cambia automáticamente si la otra está en espera. El resultado es un comportamiento perfectamente ordenado y sin conflictos.

En cambio, en la versión **con cazar**, la simulación es más realista. Si un ratón intenta comer mientras existen gatos activos, el programa detecta la violación de la regla y marca a ese ratón como cazado, deteniendo su hilo. De esta forma se cumple la regla "si un gato ve a un ratón comiendo, se lo come". Esta implementación utiliza una comprobación inmediata en el método `raton_entra()`, lo que impide que el hilo llegue a tomar un plato o modificar los contadores de animales activos. Además, para evitar bloqueos, se utiliza una adquisición no bloqueante del semáforo (`acquire(blocking=False)`) en lugar de una llamada que mantenga el candado ocupado.

6. Refinamientos y prevención de inanición

Ambas versiones implementan una política de alternancia basada en la variable `turno`. Cuando una especie termina de comer y la otra tiene hilos en espera, el turno cambia automáticamente, asegurando que ninguna quede indefinidamente bloqueada. Este esquema elimina la inanición y promueve la equidad entre especies. Además, todas las operaciones críticas se encuentran protegidas por el candado global, garantizando integridad de los datos incluso en presencia de múltiples hilos activos.

7. Posibles mejoras y observaciones

El sistema actual no mantiene un orden FIFO entre hilos de una misma especie; el orden de acceso depende del planificador del sistema operativo. Una posible mejora sería incorporar colas explícitas de espera para cada especie. También podría ampliarse la lógica de caza para permitir que un gato, al entrar, detecte y elimine ratones que ya estaban comiendo, lo que implicaría un control más detallado de los platos ocupados. Finalmente, los mensajes impresos por varios hilos pueden entrelazarse en la consola; esto podría solucionarse con un candado dedicado a las impresiones o mediante registro en archivo.

8. Conclusión

Ambas versiones cumplen correctamente con las reglas del problema y utilizan de manera coordinada los mecanismos clásicos de sincronización. Los semáforos garantizan la exclusión mutua sobre los recursos físicos (los platos), el mutex protege las variables compartidas del estado del sistema, y las variables de condición gestionan la comunicación y alternancia entre hilos, asegurando que las reglas de convivencia se mantengan sin bloqueos ni inanición.

La versión **sin cazar** representa un sistema preventivo en el que las especies nunca coinciden, mientras que la versión **con cazar** añade un comportamiento más natural en el que los ratones pueden ser cazados si rompen las reglas. En ambos casos, el diseño refleja un correcto entendimiento de la concurrencia, la sincronización y el uso disciplinado de los mecanismos de control en Python.