Universidad de Guadalajara

Centro Universitario De Ciencias Exactas E Ingenieritas CUCEI

Ingeniería Informática

Actividad 12 – Soluciones al interbloqueo, filósofos

Juan Antonio Ramírez Aguilar

Código: 212482507

Mtra. Becerra Velázquez Violeta del Rocío

Seminario de Solución de Problemas de Uso, Adaptación, Explotación de Sistemas Operativos

Indicé

Tabla de Ilustraciones

I. Introducción

En cualquier sistema, cada proceso necesita utilizar los recursos para ser implementado. Estos recursos podrían tomar la forma de dispositivos enchufados como dispositivos de entrada y salida estándar o la CPU de un sistema. Una vez finalizado el proceso, se libera el recurso que se utiliza. Sin embargo, cuando se producen múltiples procesos simultáneamente, pueden competir por recursos, lo que lleva a un bloqueo en el sistema operativo (OS). Los expertos en tecnología han identificado esta situación de interbloqueo en el sistema operativo como un problema común.

El bloqueo en el sistema operativo ocurre cuando dos o más procesos no pueden completar su ejecución porque cada uno está esperando un recurso en poder del otro. En este blog, profundizaremos en lo que es interbloqueo, con ejemplos detallados para ayudarte a entenderlo mejor. También cubriremos varios métodos para el manejo de Deadlock.

II. Concepto de interbloqueo

El bloqueo en el sistema operativo se refiere a una situación en la que más de uno o dos procesos o hilos no pueden continuar porque cada uno está esperando que el otro lance un recurso. En otras palabras, es un estado en el que un grupo de procesos se estanca de una manera que no pueden progresar. El bloqueo se produce generalmente en sistemas donde múltiples procesos compiten por recursos limitados, como el tiempo de CPU, la memoria o los dispositivos de entrada o salida.

Hay cuatro condiciones necesarias para que ocurra un punto muerto, a menudo referido como condiciones de bloqueo. Estas son las siguientes condiciones esenciales para el punto muerto.

1. **Exclusión mutua**: En esta condición, solo un método debe ser no compartible. Esto significa que solo se puede utilizar un proceso a la vez. Esta condición asegura que un recurso no pueda ser accedido o modificado simultáneamente por múltiples procesos.
2. **Mantener y esperar**: El proceso debe contener al menos un recurso mientras espera adquirir recursos adicionales. Esta condición puede conducir a una situación en la que los procesos contienen algunos recursos y esperan indefinidamente a otros.
3. **No hay preferencia**: Los recursos no pueden ser adelantados o retirados por la fuerza de un proceso. Esto significa que un recurso solo puede ser liberado voluntariamente por el proceso que lo mantiene. Si un proceso tiene un recurso y no puede completar su tarea debido a la espera de otro recurso, no liberará su recurso actual, contribuyendo a un posible punto muerto.
4. **Espera circular**: Debe haber una cadena circular de uno o más procesos, cada uno de los cuales está esperando un recurso sostenido por otro proceso en la cadena. Este patrón de espera circular significa que ningún proceso en el ciclo puede progresar porque otro proceso lo bloquea.

Como detectamos los puntos muertos

Identificar los bloqueos es esencial para una resolución eficiente. La detección de un bloqueo requiere comprobar continuamente la condición del sistema para identificar cualquier ocurrencia de bloqueo. Diferentes algoritmos, como el gráfico de asignación de recursos y el algoritmo del banquero, pueden ayudar en este procedimiento.

Los recursos y procesos del sistema se representan visualmente en el gráfico de asignación de recursos. Contiene nodos que representan procesos y recursos, conectados por bordes que muestran solicitudes y asignaciones de recursos. Al estudiar el diagrama de los bucles, podemos identificar posibles bloqueos.

El algoritmo del banquero funciona como un método para asignar recursos y prevenir bloqueos. Evalúa la seguridad del sistema mediante la ejecución de simulaciones de procesos para garantizar que todos los procesos puedan terminar sin quedarse atascado en un punto muerto. Si no es posible alcanzar un estado seguro, se establece un punto muerto.

Cuando se identifica un bloqueo, se pueden implementar pasos para resolverlo, permitiendo que los procesos continúen ejecutándose. La detección de bloqueos es una parte crucial de la gestión de Deadlocks, proporcionando información importante sobre el estado del sistema y ayudando a tomar decisiones bien informadas para resolverlos.

III. El problema de los filósofos

Edsger Wybe Dijkstra (1930–2002). Dijkstra nació el 11 de mayo de 1930 en Rotterdam, Holanda, hijo de un químico y una matemática. Estudió física y matemáticas en la Univ. de Leyden. El premio Turing otorgado por la ACM (Association for Computing Machinery, EEUU). Dijkstra escribió más de 1300 artículos, pero indudablemente hay tres contribuciones cuyo impacto está presente en numerosos ámbitos de la computación moderna.

Planteamiento del problema: Cinco filósofos se sientan a la mesa, cada uno con un plato de spaghetti. El spaghetti es tan escurridizo que un filósofo necesita dos tenedores para comerlo. Entre cada dos platos hay un tenedor. En la figura 1 se muestra la representación de la mesa. La vida de un filósofo consta de períodos alternos de comer y pensar. Cuando un filósofo tiene hambre, intenta obtener un tenedor para su mano derecha, y otro para su mano izquierda, cogiendo uno a la vez y en cualquier orden.

IV. Propuestas para resolverlo

Soluciones que rompen la Espera Circular

La solución más común y elegante es modificar la forma en que los filósofos toman los tenedores para evitar que todos esperen al vecino, lo que genera el ciclo.

Solución Asimétrica (o de Jerarquía de Recursos)

- Mecanismo: Se asigna un orden o jerarquía a los tenedores (recursos).

- Funcionamiento: Cuatro de los cinco filósofos siguen la regla estándar (tomar el tenedor izquierdo y luego el derecho). Sin embargo, un filósofo (por ejemplo, el filósofo P5) debe tomar los tenedores en orden inverso (tomar el tenedor derecho y luego el izquierdo).
- Resultado: Esto rompe la simetría y, por lo tanto, la cadena de espera circular, ya que es imposible que todos los filósofos estén esperando por un recurso que su vecino tiene y que solo liberaría si tomara el que está en el otro extremo (la espera circular se interrumpe en P5).
- Implementación: Se usa principalmente con Semáforos (un semáforo por tenedor).

Solución con un "Controlador de Mesa" (Monitor o Mutex Central)

- Mecanismo: Introduce un recurso adicional que debe ser solicitado antes de intentar comer.
- Funcionamiento: Se utiliza un Semáforo general o un Monitor que limita el número de filósofos que pueden estar simultáneamente sentados a la mesa (o intentando recoger los tenedores) a N−1 (donde N es el número total de filósofos, es decir, 4 de 5).
- Resultado: Al limitar el acceso, se garantiza que al menos un filósofo siempre podrá obtener ambos tenedores y, por lo tanto, comer. Esto impide que los 5 filósofos puedan estar en la condición de "retención y espera" simultáneamente, rompiendo la espera circular.
- Implementación: Se usa un Monitor o un Semáforo extra.

Soluciones que rompen la Retención y Espera

Estas soluciones fuerzan a los filósofos a obtener ambos recursos (tenedores) de forma atómica (al mismo tiempo) o a no obtener ninguno.

Solución de "Recogida Atómica"

- Mecanismo: Se obliga a un filósofo a tomar ambos tenedores solo si ambos están disponibles, como una operación única. Si solo puede tomar uno, no toma ninguno.
- Funcionamiento: Requiere un mecanismo de sincronización más sofisticado, a menudo implementado con Monitores o un mutex global, que permite al filósofo inspeccionar el estado de ambos tenedores y adquirirlos de forma segura.

- Resultado: Si el filósofo solo puede tomar ambos o ninguno, nunca se queda "reteniendo" un tenedor mientras "espera" el otro, eliminando la condición de retención y espera.
- Implementación: Ideal para Monitores, ya que estos permiten encapsular las operaciones de tomar y soltar en una estructura de acceso exclusivo, permitiendo al filósofo cambiar de estado solo cuando es seguro.

Soluciones Avanzadas (Detección o Prevención más Compleja)

Uso de Semáforos con un Array de Estados (Solución de Dijkstra)

- Mecanismo: Un enfoque común usando semáforos y un array de estados (THINKING, HUNGRY, EATING) para cada filósofo.
- Funcionamiento: Un filósofo solo puede pasar al estado EATING si ambos vecinos no están comiendo. Se utiliza una función test() que verifica el estado de los vecinos. Si es seguro, se le permite comer y se señaliza un semáforo privado del filósofo.
- Resultado: Esta es una solución elegante que garantiza que un filósofo hambriento solo tome los tenedores si ambos están disponibles, previniendo el interbloqueo y la inanición.
- Implementación: Requiere Semáforos binarios (Mutex) para el acceso al estado compartido y Semáforos de conteo (o condicionales) para bloquear/desbloquear a cada filósofo individual.

Paso de Mensajes

- Mecanismo: En lugar de competir por recursos compartidos (los tenedores), los filósofos y los tenedores se comunican enviando mensajes.
- Funcionamiento: Los tenedores se modelan como procesos/agentes que responden a mensajes de SOLICITAR y LIBERAR. Esto es típico en arquitecturas distribuidas.
- Resultado: El interbloqueo se gestiona en la lógica del agente tenedor, que puede implementar una política para evitar el bloqueo (por ejemplo, cediendo un tenedor si ha sido solicitado por un filósofo que ha estado esperando más tiempo, aunque esto es más complejo).

V. Mi solución

Estrategia General: Algoritmo de Dijkstra

La solución implementada al problema de los Filósofos Comensales se basa en el Algoritmo de Dijkstra, utilizando el patrón Monitor y Semáforos para la sincronización de procesos concurrentes. Esta estrategia es reconocida por ser robusta ya que no solo previene el interbloqueo, sino que también garantiza la no inanición.

Estrategia de Prevención

La solución se centra en romper la condición de Retención y Espera (una de las cuatro Condiciones de Coffman necesarias para el interbloqueo).

- Principio: Un filósofo solo puede adquirir los recursos (los dos tenedores) si ambos están disponibles simultáneamente. Si solo uno está disponible, no toma ninguno.
- Mecanismo: La clase Mesa actúa como un monitor central que controla el acceso a la acción de "comer". Los filósofos no acceden directamente a los tenedores; en su lugar, le piden permiso a la Mesa.

Componentes de Sincronización

Componente	Implementación en Python	Función y Propósito
Monitor/Mutex	self.mutex = threading.Lock()	Garantiza la Exclusión Mutua. Asegura que solo un filósofo a la vez pueda entrar en las funciones que modifican el estado (tomar_tenedores, dejar_tenedores y test), previniendo condiciones de carrera.
Array de Estado	self.estado = [0] * self.N	Mantiene el estado actual de cada filósofo (0: Pensando, 1: Hambriento, 2: Comiendo). El monitor utiliza este array para tomar decisiones seguras.

Semáforos Individuales	self.semaforo_filosofos[i]	Actúan como variables de condición. Un filósofo se bloquea en su semáforo individual si no puede comer, y solo puede ser despertado por un vecino.

El Corazón del Algoritmo: La Función test(i)

La función test(i) es la lógica central de la prevención de interbloqueo. Es la única función que puede permitir a un filósofo empezar a comer.

```
# ============================================================================
# Funcion que Dijkstra recomendo para verificar si el filosofo puede comer
# ============================================================================
def test(self, i):
    # Solo se puede comer si esta hambriento
    if self.estado[i] == 1 and self.estado[self.der(i)] != 2 and self.estado[self.izq(i)] != 2:
        self.estado[i] = 2 # Pasa a comer
        # Libera el semaforo del filosofo
        self.semaforo_filosofos[i].release()
```

Imagen 1. Implementación de la función test.

- Condición de Seguridad: Solo se ejecuta si el filósofo i está Hambriento (== 1) Y ninguno de sus vecinos inmediatos está Comiendo (!= 2).
- Prevención de Interbloqueo: Si se cumplen las condiciones, el filósofo pasa a Comiendo (= 2) y su semáforo es liberado (release()), permitiéndole salir de su espera. Si las condiciones no se cumplen, el filósofo permanece bloqueado en estado Hambriento, sin retener recursos, lo que elimina el deadlock.

Flujo de Ejecución y No Inanición

La interacción entre el filósofo y la mesa garantiza que todos tengan oportunidad de comer:

- Al Tener Hambre (tomar_tenedores(i)): El filósofo entra al monitor (mutex.acquire()), se declara Hambriento (estado[i] = 1), e intenta comer con test(i). Si no es seguro, libera el mutex y se bloquea en su semáforo individual (acquire()), esperando.
- Al Dejar Recursos (dejar_tenedores(i)): Después de comer, el filósofo libera el mutex, establece su estado a Pensando (estado[i] = 0) y, crucialmente, llama a test() para sus dos vecinos (izq(i) y der(i)).

- No Inanición: Esta llamada explícita a test() por parte del vecino que acaba de soltar los recursos asegura que cualquier filósofo hambriento adyacente sea evaluado y despertado inmediatamente si las condiciones lo permiten. Esto garantiza que nadie espere indefinidamente.
- Cálculo Circular de Vecinos: Las funciones izq(i) y der(i) ((i - 1 + N) % N y (i + 1) % N) aseguran el correcto manejo de los índices circulares, lo cual fue vital para la terminación limpia de la simulación.

VI. Links del Proyecto

Link al código del proyecto:

https://github.com/IngJuanRamirez/Solucion-al-problema-de-los-filosofos

Link al video de demostración:

https://drive.google.com/file/d/1dtjQ3UOCg5VjZ0eo23cV1A9UIPmJnnq2/view?usp=sharing

VII. Conclusión

El interbloqueo o deadlock, es un problema que surge al tratar de utilizar más hilos en un proceso. Como todo problema, también tiene muchas formas de solucionarlo dependiendo de cómo se estén usando los procesos. Como ingenieros es necesario aprender los varios tipos de soluciones que existen para poder aplicarlos en nuestra vida profesional.

También vimos el problema de los filósofos, empleado por Dijkstra para explicar el problema de los interbloqueos, la inanición y como los procesos consumen recursos. Aprendí que además de los semáforos, también se pueden crear condiciones para que los procesos ni siquiera consuman recursos si no existen espacios disponibles, evitando la inanición.

VIII. Referencias

- Irmagr. (2021, August 8). *El problema de los 5 filósofos*. Medium. https://medium.com/@irmagr1467/el-problema-de-los-5-fil%C3%B3sofos-9d0710b8ea46
- *What is Deadlock in OS?* (n.d.). Theknowledgeacademy.com. Retrieved October 25, 2025, from https://www.theknowledgeacademy.com/blog/deadlock-in-os/