Clase 3

Sincronización por variables compartidas (locks y barreras)

Dentro de memoria compartida tenemos: variables compartidas, semáforos y monitores.

Los locks y las barreras son componentes clave para sincronizar y comunicar procesos.

- Locks: estructura usada para asegurar que solo un proceso tenga acceso a un recurso compartido en un momento dado. Se usan para evitar condiciones de carrera y garantizar que los procesos accedan a los recursos compartidos ordenadamente.

  Cuando un proceso quiere acceder a un recurso compartido, primero adquiere un lock asociado a este recurso. Si el lock está disponible, el proceso lo adquiere y accede. Una vez que termine, deberá liberar el lock.

- Barreras: punto de sincronización en el que varios procesos esperan hasta que todos hayan llegado antes de que cualquiera de ellos pueda continuar.

  Serán útiles cuando cuando se necesita que varios procesos alcancen cierto estado.

Busy waiting

Técnica en la que un proceso revisa repetidamente una condición hasta que se cumpla. Se usa generalmente con locks para esperar a que un lock se vuelva disponible.

La ventaja es que puede implementarse con instrucciones de cualquier procesador pero es ineficiente en multiprogramación.

Sección crítica

Una sección crítica es una parte específica del programa en la que múltiples procesos pueden acceder a recursos compartidos. Sin embargo, un solo proceso debe tener acceso a la sección crítica en un momento dado.

El objetivo de establecer secciones críticas es evitar problemas de concurrencia como condiciones de carrera y resultados inconsistentes. Para garantizarlo, se usan mecanismos como locks o semáforos.

Suponiendo que tengo una sección crítica en un proceso una solución de seguridad a las secciones críticas sería implementar sentencias de await.

Hay algunas propiedades a cumplir ante la presencia de secciones críticas

- Exclusión mutua: a lo sumo un proceso está en su SC.
- Ausencia de deadlock: si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.
- Ausencia de demora innecesaria: si un proceso trata de entrar a su SC y los otros están en sus SN o terminaron, el primero no está impedido de entrar a su SC.
- Eventual entrada: un proceso que intenta entrar a su SC tiene posibilidades de hacerlo.

Para la implementación de la sentencia `await` para las secciones críticas hay que tener en cuenta las definiciones de

1. Acción atómica incondicional: operación que se debe realizar en su totalidad o no realizarse en absoluto, sin importar si cierta condición se cumple o no. En el contexto de secciones críticas, se sugiere que cualquier solución al problema de la sección crítica se puede usar para implementar una acción atómica incondicional.
2. Acción atómica condicional: operación que se debe realizar si se cumple cierta condición, de lo contrario se deberá esperar.

Algoritmo tie-breaker

Se enfoca en garantizar un acceso justo y ordenado a una SC compartida por varios procesadores concurrentes. Este algoritmo aborda el problema de asegurar un acceso justo incluso en situaciones en las que el scheduler no sea fuertemente justo.

Problema de la sección crítica: situación en la que varios procesos comparten recursos compartidos y necesitan acceder a una SC de forma exclusiva y ordenada con el objeto de evitar conflictos y condiciones de carrera.

Cómo funciona

Cada proceso tiene una variable que indica que ha comenzado a ejecutar su protocolo de entrada a la sección crítica. Además hay una variable compartida y protegida que se usa para resolver empates y determinar cuál de los procesos debe ingresar primero a la sección crítica.

La variable compartida registrará cuál de los procesos fue el último en comenzar su entrada al protocolo de la SC.

El proceso que comenzó su protocolo de entrada más reciente se le da prioridad. Significando que si ambos procesos desean ingresar a la SC al mismo tiempo, se le dará prioridad al que comenzó su entrada más recientemente, atrasando al otro proceso.

Ventajas

Logra justa entrada a la sección crítica incluso en situaciones en las que el planificador no sea fuertemente justo.

Desventajas

Es más complejo de implementar y requiere el uso de variables compartidas y protegidas, lo que puede aumentar la complejidad y el riesgo de condiciones de carreras.

Generalización a N procesos

Para manejar N procesos en el algoritmo tie-breaker se establece un protocolo de entrada en cada proceso. Este protocolo tiene varias etapas y usa instancias de tie-breaker para determinar cuáles procesos avanzan.

- Cada proceso sigue un protocolo de entrada que consiste en un bucle que itera a través de N-1 etapas.
- Cada etapa se encarga de permitir que dos procesos avancen a la siguiente etapa.

La clave para garantizar la exclusión mutua en la SC es que “a lo sumo a un proceso a la vez se le permite ir por las n-1 etapas”. Esto significa que solo un proceso puede avanzar a través de las etapas del protocolo a la vez, lo que garantiza que solo un proceso pueda ingresar a la SC a la vez.

Esta solución garantiza que solo uno de los n procesos pueda acceder a la SC, incluso si hay múltiples procesos por ella. Además no es necesario un scheduling extremadamente justo para que funcione correctamente.

El algoritmo "Tie-Breaker" generalizado a n procesos utiliza un protocolo de entrada con múltiples etapas y "tie-breakers" para determinar cuál proceso avanza. La restricción clave es que solo un proceso puede avanzar a través de las etapas a la vez, lo que garantiza la exclusión mutua en la sección crítica. Esto asegura que, en cualquier momento, solo un proceso pueda acceder a la sección crítica, evitando problemas de concurrencia.

Lo malo es que el tie-breaker para n procesos es complejo y costoso por lo tanto se propone el **algoritmo Ticket.**

Algoritmo Ticket

1. A cada proceso se le asigna un número de ticket único antes de que intente acceder a la SC. Estos números de ticket se distribuyen de manera secuencial, comenzando desde 1 y aumentando en orden. Cada proceso toma un número de ticket mayor que el de cualquier otro proceso que esté esperando a ser atendido.
2. Una vez que un proceso tiene su número de ticket, espera su turno. El proceso no intenta ingresar a la SC hasta que su número de ticket sea el siguiente en la secuencia.
3. Para determinar si es su turno, un proceso comprueba si su número de ticket es el siguiente en la secuencia. Si lo es, el proceso entra en la sección crítica y realiza su trabajo.
4. Cuando un proceso ha terminado de ejecutar su SC, avanza su número de ticket al siguiente en la secuencia para permitir que otros procesos continúen.

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
{ TICKET: proximo > 0 ^ (□i: 1≤ i ≤ n: (SC[i] está en su SC) □ (turno[i]== proximo) ^
(turno[i] >0) □ (□j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}
process SC [i: 1..n]
   while (true)
   < turno[i] = numero; numero = numero +1; >
< await turno[i] == proximo; >
sección crítica;
< proximo = proximo + 1; >
sección no crítica;
}
}
```

Uno de los problemas potenciales es que los valores de `próximo` y `turno` podrían teóricamente aumentar indefinidamente a medida que los procesos obtienen números de ticket y avanzan en la secuencia. En la práctica esto podría llevar a números grandes y poco eficientes.

Para abordar ese problema se sugiere que los valores de `próximo` y `turno` se reseteen a valores pequeños, en lugar de incrementarse indefinidamente. Ayuda a mantener los números de ticket en un rango manejable y evita números extremadamente grandes.

El algoritmo Ticket garantiza ciertas propiedades importantes

- Invariante global, predicado Ticket: el predicado TICKET es una invariante global. Significa que es un propiedad que se mantiene en todas las etapas de ejecución. La propiedad TICKET se refiere al hecho de que, debido a la asignación de números de ticket y al manejo de `próximo` y `turno` de manera atómica, solo un proceso tiene permitido acceder a la SC en un momento dado.

- Ausencia de deadlock y demora innecesaria: el algoritmo ticket garantiza la ausencia de situaciones deadlock y evita demoras innecesarias. Se debe a que los valores de `turno` son únicos y se usan para determinar cuál proceso tiene prioridad para acceder a la SC.
- Entrada garantizada: con un scheduling débilmente justo se asegura que eventualmente todos los procesos tengan la oportunidad de entrar a la sección crítica. O sea que, a pesar de la competencia, cada proceso tendrá la oportunidad de ejecutar la SC.

La expresión booleana utilizada para esperar `await` la entrada a la sección crítica puede implementarse mediante "busy waiting" (espera activa), lo que significa que un proceso verifica repetidamente una condición hasta que se cumple.

El aumento de "próximo" se puede implementar como una operación normal de carga/guardado (load/store) porque, en cualquier momento, solo un proceso puede estar ejecutando su protocolo de salida.

Algoritmo Backery

Es otra solución a la SC que garantiza un acceso justo y ordenado a la SC para múltiples procesos, funcionando de la siguiente forma

1. Cada proceso que intenta ingresar a la SC recorre los números asignados a los demás procesos y se autoasigna un número mayor que cualquier otro número en uso.
2. Una vez que un proceso tiene su número asignado, espera a que su número sea el menor entre todos los proceso que están esperando ingresar a la SC.
3. Los proceso se comprueban entre sí y se aseguran de que el número asignado a un proceso sea el menor de todos los números de los procesos que esperan. Esto significa que el proceso con el número más bajo es el siguiente en ingresar a la SC.
4. Cuando un proceso ha terminado de ejecutar su SC, se retira y permite que otros procesos continúen. Luego, cede el turno a otro proceso, ya que su número ya no es el más bajo.

Ventajas

- Más complejo que otros enfoques pero garantiza un acceso justo y ordenado.
- No requiere un contador global o una variable compartida para mantener un seguimiento del próximo proceso que puede ingresar a la sección crítica. En cambio, usa números únicos asignados a cada proceso y una comparación entre procesos para determinar el orden de entrada.
- Asegura que ningún proceso quede esperando indefinidamente y que todos tengan la oportunidad de acceder a la sección crítica de manera justa.

Sincronización Barrier

Mecanismo de sincronización que consiste en establecer un punto de demora en la ejecución en el que todos los procesos deben llegar antes de que se les permita continuar su ejecución. La idea detrás de esto es asegurarse de que todos los procesos hayan alcanzado un estado específico o hayan completado una fase particular antes de que cualquiera de ellos pueda avanzar.

Funcionamiento

1. Puede haber momentos en los que múltiples procesos estén haciendo tareas de manera independiente pero necesitan coordinarse en algún momento para garantizar que ciertas condiciones

se cumplan antes de avanzar.

2. Una barrera es como una puerta que bloquea el avance de todos los procesos hasta que todos ellos lleguen a la barrera. Cuando un proceso llega a la barrera, se detiene y espera a que los demás procesos también lleguen.
3. Una vez que todos los procesos han llegado a la barrera, se desbloquea y permite que todos los procesos continúen su ejecución.
4. Dependiendo de la aplicación, las barreras pueden necesitar reutilizarse más de una vez. Por ejemplo, en algoritmos iterativos, los procesos necesitan sincronizarse en cada iteración antes de continuar con la siguiente.

Importancia

Son útiles en situaciones en las que se necesita una sincronización específica entre procesos o n algoritmos iterativos donde los procesos deben sincronizarse en cada iteración para actualizar y compartir datos antes de avanzar a la siguiente iteración además ayudan de evitar condiciones de carrera.

Árboles en sincronización barrier

En algunos casos, especialmente cuando se trabaja con un número significativo de procesos, puede ser ineficiente utilizar un solo proceso (llamado coordinador) para administrar la barrera, ya que esto podría introducir cuellos de botella y aumentar el tiempo de ejecución. Para abordar este problema, se pueden utilizar estructuras de árbol para organizar los procesos de manera más eficiente.

Cuando se usan árboles en la sincronización de barreras, los procesos se organizan en forma de un árbol en lugar de una estructura lineal. Cada procesos tiene uno o más procesos padre y uno o más hijos en el árbol.

Los procesos envían señales de arribo hacia arriba en el árbol y señales de continuar hacia abajo. Esto significa que los procesos notifican a sus padres cuando llegan a la barrera y esperan a que todos los procesos hijos lleguen antes de continuar.

Barrera simétrica

- En una barrera simétrica, se construyen pares de barreras simples para dos procesos. Cada par de barreras consta de dos procesos: W[i] y W[j].
- Cada proceso dentro del par primero espera a que el otro proceso haya llegado al punto de sincronización. Esto se logra mediante la expresión `await (arribo[i] == 0);`, que indica que el proceso `i` espera a que la variable `arribo[i]` sea igual a cero, lo que significa que el otro proceso aún no ha llegado.
- Una vez que ambos procesos han llegado, ambos establecen sus respectivas variables de `arribo` en 1 para indicar que han llegado.
- Luego, cada proceso espera a que el otro proceso también haya establecido su variable de `arribo` en 1, lo que se hace con la expresión `await (arribo[j] == 1);`. Esto garantiza que ambos procesos estén listos antes de continuar.
- Finalmente, ambos procesos restablecen sus variables de `arribo` a 0 para estar listos para futuras sincronizaciones.

Combinando Barreras Simétricas para n Procesos:

- Para construir una barrera simétrica para n procesos a partir de pares de barreras para dos procesos, se organizan los procesos en una estructura de árbol.
- En cada etapa de la sincronización, los procesos se dividen en pares, y cada par realiza una barrera simétrica como se describió anteriormente.
- En la siguiente etapa, los procesos se agrupan nuevamente en pares diferentes, y repiten el proceso de barrera simétrica.
- Esto se repite durante $\log_2(n)$ etapas, donde n es el número de procesos. Cada etapa involucra a procesos que están a una distancia de 2^(s-1) entre sí, donde s es el número de etapa.
- Una vez que todos los procesos han pasado por todas las etapas, la barrera simétrica completa se ha cumplido y todos los procesos pueden continuar su ejecución.

Butterfly Barrier:

- Si el número de procesos n es una potencia de 2, esta estructura se denomina "Butterfly Barrier", ya que los procesos se organizan en el patrón de un árbol binario que se asemeja a la forma de una mariposa.

En resumen, una barrera simétrica es una técnica que utiliza pares de barreras simples para dos procesos para sincronizar un grupo de procesos de manera eficiente. Estas barreras se combinan en etapas sucesivas para garantizar que todos los procesos lleguen a un punto de sincronización antes de continuar su ejecución.

Busy waiting

La sincronización por "busy waiting" es una técnica en la programación concurrente en la que un proceso o hilo continúa ejecutándose mientras espera que se cumpla una condición. Aunque esta técnica puede ser efectiva en algunos casos, tiene ciertos defectos y desafíos asociados:

Defectos de la sincronización por busy waiting:

1. **Complejidad de los Protocolos:** Los protocolos de sincronización basados en "busy waiting" pueden volverse complejos, especialmente cuando se tienen varios procesos o hilos trabajando en conjunto. Esto se debe a que no hay una clara separación entre las variables utilizadas para la sincronización y las que se utilizan para realizar cálculos. Esta falta de separación puede hacer que sea difícil diseñar y depurar el código de manera efectiva.
2. **Dificultad en la Verificación y Prueba de Corrección:** Debido a la complejidad de los protocolos de sincronización por "busy waiting," puede ser difícil verificar su corrección y realizar pruebas exhaustivas. La verificación se vuelve aún más compleja a medida que se aumenta el número de procesos o hilos involucrados. Los errores en la sincronización pueden conducir a problemas difíciles de identificar, como condiciones de carrera y bloqueos.
3. **Ineficiencia en Multiprogramación:** En un entorno de multiprogramación, donde varios procesos comparten recursos de manera intercalada, la sincronización por "busy waiting" puede ser ineficiente. Un procesador que ejecuta un proceso en modo "busy waiting" está ocupado esperando que se cumpla una condición en lugar de realizar cálculos útiles. Esto puede llevar a una utilización subóptima de los recursos del sistema.

Dado que la sincronización por "busy waiting" presenta desafíos en términos de complejidad y eficiencia, es importante contar con herramientas adecuadas para diseñar protocolos de sincronización de manera efectiva. Estas herramientas pueden incluir técnicas de diseño estructurado, análisis de código y herramientas de depuración avanzadas.