Sincronización por variables compartidas

Locks y barreras

**Problema de la sección crítica:** Implementar acciones atómicas en software (locks)

**Barrera:** Punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.

En la técnica de **busy waiting** un proceso chequea una condición varias veces hasta que sea verdadera. Tiene la ventaja de implementarse con instrucciones de cualquier procesador ( no necesita herramientas extras ), aunque es ineficiente en multiprogramación ( varios procesos comparten el procesador). Es aceptable si cada proceso se ejecuta en su procesador.

Problema de la sección crítica

Imagen que contiene Texto

Descripción generada automáticamente

La solución que usemos a este problema necesita cumplir con una serie de propiedades:

* **Exclusión mutua:** A lo sumo un proceso está en su SC.
* **Ausencia de deadlock**: Si 2 o más procesos tratan de entrar a su SC, al menos uno tendrá éxito.
* **Ausencia de demora innecesaria:** Si un proceso trata de entrar a su SC y los otros están en su SNC o terminaron, el primero puede entrar sin problemas a su SC.
* **Eventual entrada:** Los procesos que intentan entrar a su SC van a poder entrar en algún momento.

Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional < S; >

Para una acción atómica condicional <await (B) S; >. Aunque esto es ineficiente porque un proceso está en spinning continuamente saliendo y entrando a SC hasta que otro altere una variable referenciada en B.

Si S es skip y B cumple ASV <await (B): > puede implementarse con un while (not B) skip.

Para solucionar la ineficiencia se pone un pequeño delay al salir de la SC para darle la posibilidad a otro proceso para que pueda usarla.



**Solución hardware:** Deshabilitar las interrupciones, cuando entro a la SC deshabilitar las interrupciones del sistema. Esto funciona para monoprocesador, pero es ineficiente porque durante la SC no se usa la multiprogramación. Es una solución incorrecta en multiprocesador.

**Solución de grano grueso:**

Texto

Descripción generada automáticamente

Uso sentencias await, de esta forma un proceso tiene que esperar a que el otro proceso no esté en su SC. De forma atómica marca que entró a la SC ( lo que está dentro del await) y el otro tiene que esperar a que el primer proceso salga, esto funciona porque como está dentro de un await el otro proceso no puede acceder a la variable hasta que el primer proceso termina de modificarla.

* Cumple la **exclusión mutua** por la forma en cómo se construyó.
* Cumple **ausencia de deadlock** porque para que haya deadlock ambas variables deberían ser true y esto no puede darse.
* **No hay demora innecesaria** porque si SC1 está fuera de SC o terminó, in1 es false, si SC2 está tratando de entrar a SC in1 es true. Por ende, no hay demora innecesaria.
* Cumple la **eventual entrada** porque si SC1 trata de entrar a su SC y no puede, SC2 está en SC. Un proceso en su SC eventualmente sale, por lo que el otro proceso podrá entrar. Puede darse el caso que el SC1 sea más rápido que SC2 y llegue a entrar a su SC cuando acaba de salir, haciendo que SC2 nunca entre. Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito las guardas son true con infinita frecuencia, por lo que para cumplir la eventual entrada se necesita una política fuertemente fair.

Con n procesos se cambian las variables in1 e in2 por una variable **lock**. Los procesos cuando entran a su SC marcan lock como true ( con exclusión mutua ) y el resto de los procesos esperan a que lock sea false. ( marca como ocupado, cuando sale la marca como libre ).

Así quedaría la solución generalizada a n procesos:

Texto

Descripción generada automáticamente

Cumple las 4 propiedades de la misma forma que la solución para 2 procesos. Cumple las 3 sin problema y la cuarta se cumple dependiendo de la política de scheduling.

**Solución de grano fino:**

Tiene como objetivo hacer atómico el await de grano grueso. La idea es usar instrucciones como Test y Set, Fetch y add o compare y swap, que están disponibles en casi todos los procesadores.

Solución tipo spin locks

Los procesos se quedan iterando mientras esperan que se limpie lock. Cumple las 4 propiedades si su política es fuertemente fair. Una política débilmente fair es aceptable ( rara vez todos quieren entrar a la SC a la vez ).Texto

Descripción generada automáticamente con confianza media

TS escribe siempre en lock aunque el valor no cambie, eso es mala para la memoria y la arquitectura y otras cosas, así que se usa mejor un Test-Test-And-Set. Con esto la memory contention se reduce, pero no desaparece, ya que cuando lock pasa a false posiblemente todos intenten hacer TS.

Solución con algoritmo Tie-breaker

Texto

Descripción generada automáticamente

Es un algoritmo que requiere scheduling solo débilmente fair y no usa instrucciones especiales, aunque es más complejo.

Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la SC y una variable compartida para romper empates para ver cual entró último.

Si el algoritmo tiene n procesos se vuelve complejo ( engorroso y difícil de seguir, créeme se lo que te digo) y costoso en tiempo.

Solución con Algoritmo ticket

Se reparten números y se espera a que sea el turno. Los procesos que toman un número mayor que el de cualquier otro que espera ser atendido, luego esperan hasta que todos los procesos anteriores hayan sido atendidos.

Solución de grano grueso con algoritmo ticket:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Un **potencial problema** es que los valores próximo y turno son ilimitados.

Cumple todas las propiedades ( no hay deadlock ni demora porque los turnos son únicos y el ticket es un invariante global ). Además, la eventual entrada se cumple con scheduling débilmente fair.

Para pasarlo a grano fino el **await** puede implementarse con busy waiting y el incremento de **próximo** puede ser un load/store normal

El problema es que el **ticket** que se asigna al principio es difícil de implementar, puede usarse FA pero si no existe hay que simular con una SC y la solución puede no ser fair.

Algoritmo bakery

Cada proceso que trata de ingresar recorre los números de los demás y se autoasigna uno mayor. Luego espera a que su número sea el menor de los que esperan. **Los procesos se chequean entre ellos y no contra un global.** Es un algoritmo más complejo pero es fair y no usa instrucciones especiales, además no necesita un contador global próximo que se entrega a cada proceso al llegar a la SC.

**Solución de grano grueso**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Esta solución no es implementable directamente porque la asignación a turno[i] exige calcular el máximo de valores y el await referencia una variable compartida dos veces.

Una solución de grano fino asigna a varios procesos el mismo valor de turno, para arreglar esto se usa un doble chequeo ( si dos procesos tienen el mismo turno, se desempata por id ).

Sincronización Barrier

Una barrera es un punto de demora a la que deben llegar todos los procesos antes de permitirles pasar y continuar su ejecución. Depende la aplicación las barreras pueden necesitar reutilizarse más de una vez.

**Contador compartido**

Texto

Descripción generada automáticamente

N procesos necesitan encontrarse en una barrera. Cada proceso incrementa una variable cantidad al llegar, cuando cantidad es n los procesos pueden pasar. La sentencia await se usa para sincronización por condición.

Se puede implementar con:

FA(cantidad,1);

while (cantidad <> n) skip;

Un problema es que esta solución solo puede usarse una sola vez, para que se use muchas veces hay ver como y cuando reiniciar el valor cantidad a 0.

Si no existe FA hay que distribuir cantidad usando n variables arreglo. El await pasaría a ser un esperar que todas las posiciones del arreglo tengan algo. Esto reintroduce la contención de memoria y es ineficiente.

**Flags y coordinadores**

Puede usarse un conjunto de valores adicionales y un proceso más ( un coordinador ) para que les diga a los procesos cuando pueden continuar. Cada worker espera un único valor y el coordinador un arreglo de arribo para esperar que los procesos lleguen y un proceso continuar para avisarle a los procesos que sigan.

Solución de grano fino.Texto, Carta

Descripción generada automáticamente con confianza media

Como problemas está que requiere un proceso ( y un procesador ) extra y el tiempo de ejecución del coordinador es proporcional a n.

Una solución está en combinar las acciones de workers y coordinadores haciendo que cada worker sea un coordinador. Por ejemplo, con workers en forma de árbol las señales de arribo van hacia arriba y las de continuar hacia abajo.

El problema es que los procesos juegan diferentes roles.

**Barrera simétrica**

barrera de la manteca voladora; una barrera simétrica, pero en cada etapa se vincula con flacos que no se vinculó nadie mas, etapa 1, se vinculan con distancia 1 {1-2, 3-4..} etapa 2 se distancian con 2 de distancia {1-3, 2-4} y así van con norma de distancia 2^(s-1) con s siendo la etapa

Una barrera simétrica para n procesos se construye a partir de pares de barreras simples para 2 procesos.

Defectos de la sincronización por busy waiting

Los protocolos busy waiting son complejos y sin clara separación entre variables de sincronización y las usadas para resultados. Es difícil diseñar para probar corrección, incluso la verificación es compleja cuando se incrementa el número de procesos. Además, es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso spinning puede ser usado de forma más productiva por otro proceso.

Por eso necesitamos **herramientas para diseñar protocolos de sincronización.**