Capítulo 5: Sincronización de acceso a los objetos compartidos

Los programas multi-hilo extienden el modelo tradicional, de un solo subproceso de programación de manera que cada hilo proporciona un único flujo secuencial de la ejecución de las instrucciones compuestas familiares. Si un programa tiene hilos independientes que operan en subconjuntos completamente separados de la memoria, podemos razonar sobre cada hilo por separado. En este caso, el razonamiento acerca de los hilos independientes difiere poco del razonamiento acerca de una serie de programas independientes, de un único subproceso.

Sin embargo, la mayoría de los programas multi-hilo tienen tanto el estado por hilo (por ejemplo, registros de pila y un hilo) como el estado compartido (por ejemplo, variables compartidas en el montón). Los hilos cooperantes de lectura y escritura comparten estados.

Compartir el estado es útil porque permite que los hilos se comuniquen, coordinen el trabajo y compartan información.

Por desgracia, cuando los hilos comparten estado, escribir programas multi-hilo se vuelve mucho más difícil. La mayoría de los programadores están acostumbrados a pensar "secuencialmente" al razonar los programas. Por ejemplo, a menudo razonamos acerca de la serie de estados atravesados por un programa que se ejecuta en una secuencia de instrucciones. Sin embargo, este modelo secuencial de razonamiento no funciona en los programas de hilos que comparten estados, por tres razones:

1. La ejecución del programa depende de los posibles intercambios de acceso a hilos de estado compartido.
2. La ejecución del programa puede ser no determinista.
3. Los compiladores y hardware del procesador pueden cambiar el orden de las instrucciones.

Teniendo en cuenta estos desafíos, el código multi-hilo puede introducir errores sutiles. En este capítulo se describe un enfoque estructurado para la sincronización de estados compartidos en programas multi-hilo. En lugar de la dispersión de acceso a estos recursos compartidos en todo el programa e intentar un razonamiento ad hoc sobre lo que ocurre cuando se accede a los hilos, se intercalan de diversas maneras. Un mejor enfoque consiste en: (1) la estructura del programa para facilitar el razonamiento acerca de la concurrencia, y (2) utiliza un conjunto primitivas de sincronización de Habitación para controlar el acceso a estos recursos compartidos. Este enfoque da un poco de libertad, pero si se siguen constantemente las reglas que describe en este capítulo, a continuación, el razonamiento sobre programas con estado compartido se vuelve mucho más simple.

5.1 Retos o Desafíos

Comenzamos este capítulo con el principal reto de la programación multi-hilo: la ejecución de un programa multi-hilo depende del intercalado de acceso a diferentes hilos de la memoria compartida, que pueden hacer que sea difícil razonar acerca de depuración de estos programas. En particular, la ejecución de hilos cooperantes puede verse afectada por las ***condiciones de carrera***.

5.1.1 Condiciones de Carrera

Una condición de carrera se produce cuando el comportamiento de un programa depende de la intercalación de las operaciones de diferentes hilos. En efecto, los hilos corren una carrera entre sus operaciones y los resultados de la ejecución del programa depende de quién gane la carrera.

Ejemplos:

- Supongamos que estamos corriendo un programa con dos hilos, los cuales son

  Hilo A: x = 2; Hilo B: x = 3. El posible resultado de x puede ser 2 o 3, dependiendo de cuál de los dos hilos gane la carrera.

- Supongamos ahora, que inicializamos y=12, y corremos un programa que tiene los siguientes hilos:

  Hilo A: x = y + 1; Hilo B: y = y * 2; Los posibles resultados de x pueden ser x = 13 o x = 25 dependiendo cuál de los dos hilos gane la carrera.

5.1.2 Operaciones Atómicas

Teniendo en cuenta los ejemplos anteriores, podemos hablar acerca de las ***operaciones atómicas***, operaciones indivisibles que no pueden ser intercaladas o subdividirlas por otras operaciones.

En la mayoría de las arquitecturas modernas, escribir o leer una palabra de 32 bits desde o a la memoria es una operación atómica. Por lo tanto, el análisis previo razonó sobre el intercalado de cargas y almacenamientos atómicos en la memoria.

Por el contrario, una lectura o escritura no es siempre una operación atómica. Dependiendo de la implementación de hardware, si dos hilos almacenan el valor de un registro de coma flotante de 64 bits a una dirección de memoria, el resultado final podría ser el primer valor, el segundo valor, o una mezcla de los dos.

5.1.3 Demasiada Leche

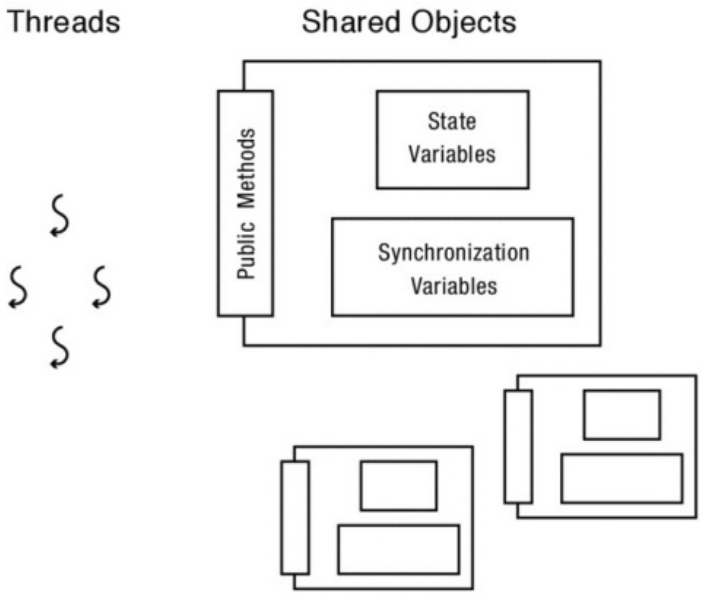
Aunque se podría, en principio, la razón cautelosa acerca de la posible intercalación de lectura y escritura atómicas de diferentes hilos, es porque hacerlo es complicado y propenso a errores. Más tarde, se presentará un mayor nivel de abstracción para la sincronización de hilos, pero primero se ilustrarán los problemas con el uso de lectura y escritura atómicas que tienen un problema simple llamado, "demasiada leche". Los modelos de problemas de demasiada leche de dos compañeros que comparten un refrigerador y que - como buenos compañeros – se aseguran de que el refrigerador esté siempre bien abastecido con leche. Cuando uno ve que el refrigerador no tiene leche, va a la tienda a comprar más; si no le avisa a su compañero dejándole una “notita” él también ira a comprar leche y se produce este exceso de leche en el refrigerador.

5.1.4 Discusión

Suponiendo que el compilador y el procesador ejecutan instrucciones de un programa en orden, la prueba anterior muestra que es posible idear una solución al problema en cuestión, que es a la vez segura y viable usando nada más que las operaciones de carga y almacenamiento atómicas en la memoria compartida. Aunque la solución que presentamos sólo funciona para los dos compañeros (n igual a 2), no es una generalización. Existe un algoritmo, llamado algoritmo de Peterson, que funciona con cualquier número fijo de n hilos.

5.2 La estructura de los objetos compartidos

Décadas de trabajo han desarrollado un enfoque mucho más simple para escribir programas multi-hilo usando cargas y almacenamientos atómicos. Este enfoque extiende la modularidad de la programación orientada a objetos para programas multi-hilo. Como ilustra la figura, un programa multi-hilo se construye a partir de los objetos compartidos y un conjunto de hilos que operan en ellos.



Los objetos compartidos son objetos que se pueden acceder de manera segura por varios subprocesos. Todos estos recursos compartidos en un programa - incluyendo variables asignados en la pila (por ejemplo, objetos asignados con malloc o new) y, variables globales estáticas - debe ser encapsulado en uno o más objetos compartidos.

La programar con objetos compartidos se extiende de la programación tradicional orientada a objetos, en la cual los objetos ocultan sus detalles de implementación detrás de una interfaz limpia. De la misma manera, los objetos compartidos ocultan los detalles de la sincronización de las acciones de múltiples hilos detrás de una interfaz limpia. Los hilos que utilizan objetos compartidos sólo necesitan entender la interfaz; ellos no necesitan saber cómo el objeto compartido maneja internamente la sincronización.

Al igual que los objetos normales, los programadores pueden diseñar objetos compartidos para cualesquiera módulos, interfaces y semántica que necesita la aplicación. Cada clase de objeto compartido define un conjunto de métodos públicos en los que operan los hilos. Para ensamblar el programa general de estos objetos compartidos, cada hilo ejecuta un "bucle principal", escrito en términos de acciones sobre los métodos públicos de objetos compartidos.

Dado que los objetos compartidos encapsulan el estado compartido del programa, el código del bucle principal que define medidas de alto nivel de un hilo no tiene por qué preocuparse de los detalles de sincronización. Así, el modelo de programación se ve muy similar a la del código de un solo subproceso.

5.2.1 La implementación de objetos compartidos

Por supuesto, internamente los objetos compartidos deben manejar los detalles de sincronización. Los objetos compartidos se implementan en capas:

- **Capa de Objeto Compartido.** Al igual que en la programación orientada a objetos, los objetos compartidos definen la lógica específica de la aplicación y ocultan detalles de implementación internos. Externamente, parecen tener la misma interfaz que se debe definir para un programa de un solo subproceso.
- **Capa de Sincronización de Variable.** En lugar de implementar objetos compartidos directamente con cargas y almacenamientos atómicos cuidadosamente entrelazados, los objetos compartidos incluyen variables de sincronización como variables miembro. Las variables de sincronización, almacenadas en la memoria al igual que cualquier otro objeto, pueden incluirse en cualquier estructura de datos.

  Una ***variable de sincronización*** es una estructura de datos utilizada para coordinar el acceso simultáneo al estado compartido. Tanto la interfaz como la implementación de las variables de sincronización deben ser cuidadosamente diseñadas. En particular, podemos construir objetos compartidos utilizando dos tipos de variables de sincronización: Bloqueos (locks) y Variables de Condición. Las variables de sincronización coordinan el acceso a las variables de estado, que son sólo las variables de miembros normales de un objeto que está familiarizado con la programación a partir de un único subproceso (por ejemplo, enteros, cadenas, arrays y punteros).
- **Capa de Instrucción Atómica.** A pesar de que las capas superiores se benefician de un modelo de programación más simple, no es lento hasta el final. Internamente, las variables de sincronización deben gestionar el entrelazado de las acciones de diferentes hilos.

  En lugar de implementar las variables de sincronización, tales como bloqueos y variables de condición, utilizando carga y almacenamiento atómico como hemos tratado de hacer por el problema de demasiada leche, las implementaciones modernas construyen las variables de sincronización utilizando instrucciones de lectura-modificación-escritura atómicas.

  Estas instrucciones específicas del procesador dejan un hilo que tiene acceso de forma temporal y exclusiva a una ubicación de memoria mientras se ejecuta la instrucción. Típicamente, la instrucción atómicamente lee una posición de memoria, hace alguna operación aritmética simple para el valor, y almacena el resultado. El hardware garantiza que las instrucciones de cualquier otro hilo con el acceso a misma posición de memoria se producen ya sea por completo antes de, o en su totalidad después de, la instrucción de lectura-modificación-escritura atómica.

5.3 Bloqueos: Exclusión Mutua

Un bloqueo es una variable de sincronización que proporciona la exclusión mutua - cuando un hilo mantiene un bloqueo, ningún otro hilo puede mantenerlo (es decir, otros hilos están excluidos). Un programa asociado a cada bloqueo con algún subconjunto de estados compartidos y requiere un hilo para mantener el bloqueo al acceder a ese estado. Entonces, sólo un hilo puede acceder al estado compartido a la vez.

La exclusión mutua simplifica en gran medida el razonamiento acerca de los programas con un subproceso que puede realizar un conjunto arbitrario de operaciones, mientras que mantiene un bloqueo, y esas operaciones parecen ser atómicas a otros hilos. En particular, debido a un bloqueo, hace cumplir la exclusión mutua y los hilos deben sostener el bloqueo para acceder a estos recursos compartidos, ningún otro hilo puede observar un estado intermedio. Otros procesos únicamente pueden observar el estado de la izquierda después de la liberación del bloqueo.

Es mucho más fácil de razonar acerca de entrelazado de grupos atómicos de operaciones en lugar de entrelazado de las operaciones individuales por dos razones. En primer lugar, hay menos entrelazado a considerar. En segundo lugar, y más importante, que podemos hacer cada grupo atómico de las operaciones correspondientes a la estructura lógica del programa, lo que nos permite razonar sobre invariantes no intercalación específica.

En particular, los objetos compartidos por lo general tienen un bloqueo que guarda todos los estados de un objeto. Cada método público adquiere el bloqueo a la entrada y libera el bloqueo en la salida. Por lo tanto, el razonamiento sobre el código de una clase compartida es similar a razonar sobre el código de una clase tradicional: asumimos un conjunto de invariantes cuando se llama a un método público y restablecer esas invariantes ante un público devuelve el método. Si definimos así nuestros invariantes, podemos razonar acerca de cada método de forma independiente

5.3.1 Bloqueos: API y Propiedades

Un bloqueo permite la exclusión mutua, proporcionando dos métodos: Lock::acquiere() Bloqueo::adquirir() y Lock::Release() Bloqueo::liberar(). Estos métodos se definen como sigue:

- Un bloqueo puede estar en uno de dos estados: Busy o Free (ocupado o libre).
- Un bloqueo se encuentra inicialmente en el estado libre.
- Bloqueo::acquiere espera hasta que el bloqueo se libere y luego atómicamente hace que el bloqueo quede OCUPADO.
- Bloqueo::realase hace que el bloqueo quede libre. Si hay operaciones pendientes adquirir, este cambio de estado hace que uno de ellos para proceder.

Un Bloqueo se puede definir con más precisión como sigue. Un hilo mantiene un bloqueo si se ha de volver de método de adquirir un bloqueo de más a menudo de lo que ha de volver de método de liberación de un bloqueo. Un hilo está intentando adquirir un bloqueo si se ha invocado, pero todavía no regresar de una llamada a adquirir en la cerradura.

Un bloqueo debe garantizar las siguientes propiedades:

1. **Exclusión Mutua.** A lo sumo un hilo mantiene el bloqueo.
2. **Progreso.** Si hay hilo mantiene el bloqueo y cualquier tema intenta adquirir el bloqueo, entonces eventualmente un poco de hilo tiene éxito en la adquisición de la cerradura.
3. **Tiempo Limitado.** Si el hilo T intenta obtener un bloqueo, entonces existe un límite en el número de veces que otros hilos pueden adquirir con éxito la cerradura antes de T hace.

La exclusión mutua es una característica de seguridad porque los bloqueos impiden que más de un hilo accedan a estos recursos compartidos.

Progreso y espera acotada son propiedades LIVENESS. Si un bloqueo es GRATIS, un poco de hilo debe ser capaz de adquirirlo. Además, cualquier hilo particular que quiere adquirir el bloqueo debe finalmente tener éxito en hacerlo.

Si estas definiciones suenan poco naturales, es porque las hemos construido cuidadosamente para evitar la introducción de casos sutiles. Por ejemplo, si un hilo que mantiene un bloqueo no lo suelta, otros hilos no pueden avanzar, por lo que definimos la condición de espera limitada en términos de la operación acquiere (adquirir) con éxito.

5.3.2 Hilo de Seguridad de Cola Limitada

Al igual que en la programación orientada a objetos estándar, cada objeto compartido es una instancia de una clase que define el estado de la clase y los métodos que operan en ese estado.

El estado de la clase incluye dos variables, ***de estado*** (por ejemplo, enteros, flotadores, cuerdas, matrices y punteros) y las ***variables de sincronización*** (por ejemplo, bloqueos). Cada vez que un constructor de la clase produce otra instancia de un objeto compartido, asigna tanto una nueva cerradura y nuevas instancias del estado protegido por esa cerradura.

Una cola acotada es una cola con un límite de tamaño fijo de elementos almacenados en la cola. El kernel del sistema operativo utiliza colas acotadas para la gestión de la comunicación entre procesos, Zócalos TCP y UDP, y las solicitudes de E/S. Dado que el núcleo se ejecuta en una memoria física finita, el núcleo debe ser diseñado para

funcionar correctamente con recursos finitos. Por ejemplo, en lugar de un simple buffer infinito entre un generador de un hilo y un consumidor, el kernel utiliza un buffer de tamaño limitado, o cola limitada.

Una sección crítica es una secuencia de código que tiene acceso automático al estado compartido. Al asegurar que sostiene un hilo de bloqueo del objeto durante la ejecución de cualquiera de sus secciones críticas, nos aseguramos de que cada sección crítica aparece para ejecutar automáticamente su estado compartido.

Los objetos compartidos se asignan de la misma forma que otros objetos. Pueden asignarse dinámicamente desde el almacenamiento dinámico utilizando malloc y new, o pueden ser asignados estáticamente en la memoria global por la declaración de variables estáticas en el programa.

Múltiples hilos deben ser capaces de acceder a los objetos compartidos. Si los objetos compartidos son variables globales, a continuación, el código de un hilo puede referirse al nombre global de un objeto para hacer referencia a él; el compilador calcula la dirección correspondiente. Si los objetos compartidos se asignan dinámicamente, a continuación, cada hilo que utiliza un objeto necesita un puntero o referencia a ella.

Dos formas comunes de proporcionar un hilo de un puntero a un objeto compartido son: (1) proporcionar un puntero al objeto compartido cuando se crea el hilo, y (2) almacenar referencias a objetos compartidos en otros objetos compartidos (por ejemplo, contenedores). Por ejemplo, un programa podría tener una tabla hash compartida y global (y sincronizada!) Que los hilos se pueden utilizar para almacenar y recuperar referencias a otros objetos compartidos.

5.8 Los semáforos consideran perjudiciales

Este libro se centra en la construcción de objetos compartidos utilizando bloqueos y variables de condición para la sincronización. Sin embargo, en los últimos años, se han propuesto muchas diferentes primitivas de sincronización, incluyendo procesos secuenciales que se comunican, la entrega de sucesos, el paso de mensajes, y así sucesivamente. Es importante darse cuenta de que ninguno de ellos es más poderoso que el uso de bloqueos y variables de condición; un programa utilizando cualquiera de estos paradigmas se pueden asignar a los monitores que utilizan transformaciones sencillas.

Un tipo de sincronización, un semáforo, vale la pena discutir en detalle, ya que sigue siendo ampliamente utilizado. Los semáforos fueron introducidos por Dijkstra para proporcionar sincronización en el sistema operativo LA, que (entre otros avances) exploró las formas estructuradas de uso concurrencia en el diseño del sistema operativo

Los semáforos se definen como sigue:

- Un semáforo tiene un valor no negativo.
- Cuando se crea un semáforo, su valor puede ser inicializado a cualquier número entero no negativo.
- Semáforo::P() espera hasta que el valor es positivo. A continuación, se atómicamente disminuye el valor en 1 y regresa.
- Semáforo::V() atómicamente incrementa el valor de 1. Si todos los hilos están esperando en P, uno está habilitada, por lo que su llamada a P tiene éxito en decrementar el valor y rentabilidad.
- No hay otras operaciones se permiten en un semáforo; en particular, ningún hilo puede leer directamente el valor actual del semáforo.

Tenga en cuenta que el semáforo::P es una operación atómica: la lectura que se observa el valor positivo es atómica con la actualización que lo decrementa. Como resultado, los semáforos no pueden tener un valor negativo, aun cuando varios subprocesos llaman P al mismo tiempo.

Del mismo modo, si V se produce cuando hay un hilo de espera en P, entonces la subasta de P y disminución del valor de V son atómicos: ningún otro hilo puede observar el valor incrementado, y del interior de P está garantizado para disminuir el valor y volver.

Teniendo en cuenta esta definición, los semáforos pueden ser usados para exclusión mutua (como los bloqueos) o de espera general para otro hilo para hacer algo (un poco como variables de condición). Para utilizar un semáforo como un bloqueo de exclusión mutua, inicializa a 1. Entonces, Semáforo::P es equivalente a Lock::acquiere y semáforo::V es equivalente a Lock::realase.

Semáforo P y V se pueden configurar para comportarse de manera similar. Por lo general (pero no siempre), inicializar el semáforo a 0. Entonces, cada llamada al semáforo::P espera a que la rosca correspondiente para llamar V. Si la V se llama en primer lugar, entonces P vuelve inmediatamente.

La dificultad viene cuando se trata de coordinar estado compartido (que necesita la exclusión mutua) con la espera general. Desde la distancia, Semáforo::P es similar a la CV::espera (y la cerradura) y semáforo :: V es similar a la CV :: señal. Sin embargo, hay diferencias importantes. En primer lugar, CV::esperar (y la cerradura) atómicamente libera el bloqueo del monitor, por lo que se puede comprobar con seguridad el estado del objeto compartido y luego suspender atómicamente ejecución.

Los semáforos consideran dañinos. Nuestro punto de vista es que la programación con las cerraduras y las variables de condición es superior a la programación con semáforos. Te recomendamos siempre a escribir el código usando las variables de sincronización por dos razones.

En primer lugar, el uso de clases de bloqueo y variables independientes condición hace que el código sea más auto-documentado y fácil de leer. A medida que la cita de notas Dijkstra, se necesitan dos abstracciones diferentes, y el código es más claro que el papel de cada variable de sincronización se pone de manifiesto a través de la tipificación explícita. Por ejemplo, es mucho más fácil de verificar que cada adquieren bloqueo está emparejado con un abrepuertas, si no se mezclan con otras llamadas a P y V de espera general.

En segundo lugar, un estado variable sin estado unido a un bloqueo es una mejor abstracción de espera generalizada de un semáforo. Mediante la unión de una variable de condición a una cerradura, podemos esperar cómodamente en cualquier predicado arbitrario en el estado de un objeto. Por el contrario, los semáforos se basan en el programador para trazar cuidadosamente el estado del objeto al valor del semáforo por lo que la decisión de esperar o proceder en P puede ser hecho enteramente basado en el valor, sin que mantiene un bloqueo o examinando el resto del estado del objeto compartido.

Aunque no se recomienda escribir el nuevo código con semáforos, código basado en los semáforos no es infrecuente, especialmente en los sistemas operativos. Por lo tanto, es importante entender la semántica de los semáforos y ser capaz de leer y entender el código basado semáforo-escrito por otros.