TP2: Simulador de Cache

Santiago Alvarez Juliá, *Padrón Nro. 99522*
75.42 Taller de Programación
Facultad de Ingeniería, Universidad de Buenos Aires

Septiembre 2018

Índice general

0.1. Introducción 1
0.2. Temas Claves 1
0.2.1. Programación Orientada a Objetos 1
0.2.2. Multi Threading 3
0.3. Diagramas 4

0.1. Introducción

En las siguientes secciones abordaré algunos temas claves para la resolucion del programa Simulador de cache. Respecto a la base de programación de ejercicios anteriores, este difiere en 2 puntos claves :

- Programación Orientada a Objetos (POO)
- Multi Threading

0.2. Temas Claves

0.2.1. Programación Orientada a Objetos

Excepto la funcion main, todo el resto del programa fue diseñado en base a objetos que se relacionan entre si. A continuación haré una breve descripción de cada objeto de la aplicación.

- Thread: este objecto encapsula la clase std::thread (que representa un hilo de ejecución). Dentro del método start llamo al constructor de inicialización de std::thread que recibe como parámetros un puntero al método run de Thread y un puntero al objecto para el cual esta definido el método run. Dicho método run es virtual puro, por lo tanto Thread es una clase abstracta. Esto permite que cualquier objeto pueda correr en su propio hilo mientras herede de Thread e implemente el método run.
- FunctorCache: como lo indica su nombre este objecto es un functor, su objetivo es encapsular la función que debe ejecutarse cuando se lanza un hilo. Esta clase hereda de Thread e implementa el método run. Dentro de dicho método se parsean los archivos binarios que contienen las direcciones de memoria (1 archivo por hilo). Para cada dirección de memoria se le pregunta al atributo CacheProtected si la direcciones es valida y en caso contrario imprime que es invalida. En el caso positivo, se le vuelve a enviar la dirección a CacheProtected para que la procese. En el caso negativo, se finaliza la ejecución del hilo.
- CacheProtected: este objecto encapsula al objecto compartido por todos los hilos Cache, su mutex protector y un mutex mas que protege al std::cerr. El mutex protector de cache es utilizado en un unico metodo, que a su vez es el unico metodo en el cual se podria imprimir por std::cout durante la ejecucion multihilo, por lo tanto prescindo de un

mutex que proteja a std::cout en particular. Su objetivo es lockear mutex cuando sea necesario, su comportamiento lo delega completamente al objeto Cache (excepto la validación de direcciones de memoria).

- Cache: es una clase abstracta que tiene como clases derivadas CacheDirect, CacheFifo y CacheLru. Tiene como método virtual puro 'proccesMemoryAddress' cuya implementación depende de las clases derivadas. Esta clase tiene como objetivo permitir el polimorfismo para que las distintas clases de cache hagan su propia implementación de métodos que dependan del tipo de clase derivada. Para el resto de los métodos cuya implementación es común a todas las caches, esta se escribe en el .cpp de Cache y así evitar repetición de código. Análogamente sucede lo mismo con los atributos del objeto, si sus comportamientos son comunes a todas las clases derivadas de Cache, estos se escribe en el .cpp de Cache y se declaren como protegidos para que solo puedan acceder clases hijas.

- CacheDirect: esta clase es hija de la clase Cache. Utilizo un std::map para simular una cache directa ya que la única condición que se debe cumplir es que la búsqueda de tags sea de orden de complejidad menor a n (siendo n la cantidad de elementos que contiene), el método 'find' de std::map tiene orden de complejidad log (n).

- CacheFifo y CacheLru: ambas clases son hijas de la clase Cache. En ambas utilizo un std::map para la búsqueda de tags (al igual que en CacheDirect utilizo el método 'find'). Para simular la cache asociativa utilizo el std::deque. En ambos utilizo std::deque xq me permite agregar tags al cache en O(1) con el método 'pushBack'. En el caso de la CacheFifo también es O(1) reemplazar un bloque cuando se llena la cache con el método 'popFront'.

  En el caso de CacheLru gracias a std::deque agregar un bloque al cache cuando hay un miss y este no esta lleno es O(1) y borrar de la cache el bloque menos usado recientemente también es O(1) con el método 'popFront'. Cuando hay un hit debe borrarse el bloque viejo que contiene el mismo tag y agregar al principio el bloque nuevo para que tenga sentido usar 'popFront', para realizar ese reemplazo anteriormente almaceno un iterador a cada elemento que agrego al std::deque y lo amaceno en el std::map donde la clave es el tag asociado a dicho iterador. Por lo tanto quitar un elemento de std::deque con un iterador es O(1), lo que mas cuesta es almacenar el iterador en el map, lo realizo con el operador[], el cual tiene orden de complejidad log (n).

En ambas implementaciones también utilizo el método 'size' de std::map para saber si la cache esta llena que según documentación es O(1).

Me parece importante aclarar que sería mejor implementar a CacheFifo y CacheLru como clases derivadas de una nueva clase llamada CacheAsociativa que sea abstracta y sea hija de Cache. Como lo aclara su nombre, ambas caches fifo y lru se comportan de la misma manera excepto en la política de reemplazo. No lo implemente de tal manera por falta de tiempo.

- Lock: es una encapsulación RAII de la toma y la liberación del recurso mutex.

0.2.2. Multi Threading

Dentro del main thread se lanzan el resto de los hilos, 1 por cada archivo de direcciones de memoria ya que la información que almacenan puede ser procesada independientemente. El programa tiene una critical section que es verificar si dado un tag, este se encuentra en la cache y actuar en función de eso. También pueden ocurrir race conditions cuando se quiere imprimir por pantalla, ya sea por std::cout o std::cerr porque ambos archivos son compartidos por todos los hilos.

- Cache : existe una única instancia de Cache y tiene su propio mutex. Ambos elementos están encapsulados en el objeto CacheProtected. Dicho mutex es lockeado solamente cuando hay que procesar un tag, justo después de verificar si el tag representa una dirección de memoria válida, es decir cuando quiero verificar si dicho tag produce un hit o miss en la cache. Dentro de CacheProtected es lockeado dicho mutex justo antes de llamar al método 'proccesMemoryAddress'.
- Std::cout y std::cerr : antes de lanzar los hilos se abre el archivo de configuración y se imprime por pantalla estándar los datos de la cache y luego de "joinear"los hilos se imprime también por salida estándar el informe de hits y misses que generaron los archivos de direcciones de memoria en la cache. Ninguna de las anteriores impresiones podría generar una race condition porque siempre van a suceder cuando solamente el main thread está ejecutándose.

  En cambio cuando se ejecutan los hilos pueden suceder 2 cosas que podrían producir race conditions: una dirección de memoria inválida y

en el caso de que la cache tenga el modo debug en true, cada vez que se produce un hit o miss se tendría que imprimir sin problemas por salida estándar lo sucedido. El primer caso tiene como recurso compartido la salida de error estándar std::cerr por lo tanto para evitar la race condition CacheProtected también almacena un mutex asociado unicamente con std::cerr que es lockeado antes de verificar la validez de la dirección de memoria que se esta procesando. Para el segundo caso se da la situación particular de que el recurso compartido std::cout esta protegido por el mutex de la cache ya que este es lockeado solamente cuando hay que procesar un tag, justamente la única situación en la que se podría producir una race condition (es decir en el método 'proccesMemoryAddress' de cache).

0.3. Diagramas