Simulador de cache

Ejercicio N° 2

Objetivos	● Diseño y construcción de sistemas orientados a objetos ● Diseño y construcción de sistemas con procesamiento concurrente ● Encapsulación de Threads en clases ● Protección de los recursos compartidos
Instancias de Entrega	**Entrega 1:** clase 6 (25/09/2018). **Entrega 2:** clase 8 (09/10/2018).
Temas de Repaso	● Threads en C++11 ● Clases en C++11
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Orientación a objetos del sistema ● Empleo de estructuras comunes C++ (string, fstreams, etc) en reemplazo de su contrapartida en C (char*, FILE*, etc) ● Uso de **const** en la definición de métodos y parámetros ● Empleo de constructores y destructores de forma simétrica ● Buen uso del stack para construcción de objetos automáticos ● Ausencia de condiciones de carrera e interbloqueo en el acceso a recursos ● Buen uso de Mutex y Monitores para el acceso a recursos compartidos

Índice

Trabajo práctico N2

Simulador de cache de memoria

Introducción

En una computadora moderna, la CPU trabaja a muy alta velocidad ejecutando instrucciones sencillas como operaciones aritméticas y lógicas, accesos a memoria, etc. Los accesos a memoria son muy frecuentes y variados, por ejemplo, cuando iteramos sobre una estructura de datos como un vector o una lista.
La memoria principal trabaja a distinta frecuencia que la CPU, y sumando protocolos de transferencia y otras validaciones, puede tardar cientos de ciclos de CPU en acceder a la memoria principal.
Para mitigar estos problemas de performance, se utiliza una memoria intermedia de menor capacidad pero mucho mayor velocidad de acceso, llamada **Cache**
La *memoria cache* suele tener tamaños entre unos pocos KB a algunos pocos MB, pero a diferencia de la RAM principal, el tiempo que se tarda en acceder suele estar cerca de los 10 ciclos de reloj.

Se pide implementar un simulador de memoria cache que permita estudiar distintas implementaciones de memoria cache, para distintos escenarios prearmados.

Descripción

En este simulador se ingresará por comando el nombre de un archivo de configuración, con formato clave=valor, y archivos de texto plano con las direcciones de memoria a las que accede cada cpu. El archivo de configuración es mandatorio, y los archivos de direcciones pueden ser uno o varios.

En caso de recibir más de un archivo de direcciones de memoria se deberán simular los accesos de forma *concurrente*, trabajando con una *única* instancia de cache siendo procesado cada archivo por un hilo.

Acceso a la cache

La memoria *cache* se organiza en *bloques* de memoria de 2^n bytes. A su vez, estos bloques se pueden agrupar en *vías*. Para este simulador solo implementaremos caches de una única vía (direccionamiento directo), o totalmente asociativos, donde cada vía tiene un único bloque.

Cuando queremos acceder a un dato en la memoria principal, utilizamos una *dirección de memoria*. Una dirección de memoria es un número de N bits con el que identificamos una línea de datos.

Para utilizar la cache, se "interceptan" estos accesos a memoria y según el tipo de cache que utilizamos se realiza una búsqueda del dato en cuestión, utilizando algunos bits de la dirección de memoria.

Si el dato que busco está en el cache, se dice que ocurre un *hit*, caso contrario se lo llama *miss*

Cache asociativo

Cuando usamos un cache totalmente asociativo, dividimos la dirección en 2 partes: tag y offset.

Tag es la parte de la dirección de memoria que identifica a un bloque, y está compuesta por los bits más significativos. Offset identifica una ubicación dentro del bloque, y está compuesta por los menos significativos.

Para este trabajo práctico ignoraremos el contenido de los bloques, ergo ignoramos el offset.

Ejemplo 1: Tenemos una memoria principal con direcciones de 32bits, y una memoria cache de 64KB, dividida en bloques de 16 bytes. Esto significa que voy a tener 64KB/16B: 4096 bloques, y que además, como cada bloque es de 16B, voy a usar log2(16) bits para indicar el offset, es decir, 4 bits de offset.

Si usamos 4 bits para indicar el offset, los 28 restantes son bits del tag, con los que identificamos cada bloque dentro del cache.

Ejemplo 2: tenemos la dirección de memoria 0xDEADBEEF, y la quiero buscar en un cache asociativo de 64KB, con bloques de **256B** de tamaño.

> log2(256) = 8
> 8 bits para el offset, 24 para el tag
> Los 8 bits menos significativos, los del offset, son 0xEF
> Los 24 bits más significativos, los del tag, son 0XDEADBE

Cache directo

Cuando utilizamos una memoria cache, dividimos la dirección en 3 partes: tag, index y offset.

El índice nos dice en qué bloque dentro del cache, el offset nos dice en qué posición dentro del bloque está, y el tag se utiliza para comparar y validar si el bloque encontrado corresponde a la memoria principal solicitada.

Para este trabajo práctico, con los caches directos, nos centraremos en el uso del índice y del tag e ignoraremos el offset.

Ejemplo: Tenemos una memoria principal de 32bits memoria cache de 64KB, dividida en bloques de 16 bytes.

> Voy a tener 64KB/16B: 4k bloques.
> Si cada bloque es de 16B, entonces 4 bits se utilizan como offset.
> Si tengo 4k bloques, entonces utilizo 12 bits para indexar el bloque.
> Los 16 bits remanentes (32 - 12 - 4) son utilizados como tags.
> Dada una dirección de memoria como la siguiente: 0x11223344, lo separamos en:
>
> 1. 16 bits de tag: 0x1122
> 2. 12 bits de índice: 0x334
> 3. 4 bits de offset: 0x4

Registrar un acceso a la memoria

Cache asociativo

Cuando arranca el sistema, el cache inicia vacío. A medida que se solicitan accesos a memoria, se va poblando el cache con pequeños fragmentos de la memoria principal que fueron solicitados. En este trabajo práctico ignoraremos el tipo de acceso (lectura o escritura) ya que ignoramos el contenido del cache.

Ejemplo: tenemos una memoria cache de 1KB, bloques de 256B, entonces log2(64) -> 8 bits para el offset, 26 para el tag.

Tenemos 1024/256 -> 4 bloques

Supongamos que solicito acceso a las direcciones

```
0x00001130
0xFF00113C
0x0000113C
0x0000123C
```

En un cache totalmente asociativo registro los siguientes tags:

```
0x00001130: Tag: 0x000011
0xFF00113C: Tag: 0xFF0011
0x0000113C: Tag: 0x000011
0x0000123C: Tag: 0x000012
```

Ingreso el primer tag, a la izquierda (lo mostrado a continuación seria el cache con sus 4 bloques):

```
[0x000011 -------- -------- --------]
```

Ingreso el segundo tag.

```
[0xFF0011 0x000011 -------- --------]
```

El tercer tag ya se encuentra en el cache, por lo cuál no hay necesidad de agregarlo.

```
[0xFF0011 0x000011 -------- --------]
```

Por último agrego el cuarto tag.

```
[0x000012 0xFF0011 0x000011 --------]
```

Políticas de reemplazo

Cuando el cache se satura, hay que adoptar políticas de reemplazo para ver qué línea quitar del cache. Implementaremos dos políticas:

- **FIFO**: *First in, first out*, las entradas del cache se van eliminando en el mismo orden que ingresaron
- **LRU**: *Least recently used*, se elimina la entrada cuyo último uso sea el más viejo de todos.

Ejemplo de FIFO

Tomamos la configuración del ejemplo anterior, ampliándolo

```
0x00001130
0xFF00113C
0x0000113C
0x0000123C
0x00001338
0x00002538
```

```
Ingreso tag por tag
      [0x000011 -------- -------- --------]
      [0xFF0011 0x000011 -------- --------]
```

El tercer tag ya se encuentra en el cache, por lo cuál no hay necesidad de agregarlo.

```
      [0xFF0011 0x000011 -------- --------]
      [0x000012 0xFF0011 0x000011 --------]
      [0x000013 0x000012 0xFF0011 0x000011]
Acá intentamos agregar el tag 0x000025, pero como no entra en el cache, saco el primero
que se usó (First in, first out)
      [0x000025 0x000013 0x000012 0xFF0011]
```

Ejemplo de LRU

Tomamos el ejemplo anterior, pero con política LRU.

```
0x00001130
0xFF00113C
0x0000113C
0x0000123C
0x00001338
0x00002538
```

Ingreso tag por tag, a continuación los anoto según que tan recientemente fueron usados.

```
      [0x000011 -------- -------- --------]
      [0xFF0011 0x000011 -------- --------]
```

El tercer tag, `0x000011`, ya se encuentra en el cache, por lo cuál no hay necesidad de agregarlo. Sin embargo, lo tengo que “refrescar”

```
      [0x000011 0xFF0011 -------- --------]
      [0x000012 0x000011 0xFF0011 --------]
      [0x000013 0x000012 0x000011 0xFF0011]
Acá intentamos agregar el tag 0x000025, pero como no entra en el cache, saco el primero
que se usó (First in, first out)
      [0x000025 0x000013 0x000012 0x000011]
```

Cache directo

Registrar un acceso a memoria en un cache directo es más fácil: tenemos 2^b bloques (donde *b* es la cantidad de bits utilizados para identificar un bloque). Utilizamos los *b* bits correspondientes al índice de la memoria, y buscamos en el cache el bloque correspondiente a ese índice. Si el bloque estaba en uso, comparamos los bits de tag para saber si fue un hit o un miss.

Ejemplo de Directo

Tomamos el ejemplo anterior, pero con política Directo. Cada bloque es de 256B, son 8 bits de offset, ergo 24 de tag. Como son 4 bloques, los 2 bits (calculado a partir de log2(4)) menos significativos del tag serán usados para determinar el bloque que se utilizará.

```
0x00001130
0xFF00113C
0x0000113C
0x0000123C
0x00001338
0x00002538
```

Ingreso tag por tag.

```
      [Bloque00 Bloque01 Bloque02 Bloque03]
      ------------------------------------
Al Tag 0x000011 le corresponde (0x000011 mod 4 = 1)
      [-------- 0x000011 -------- --------]
Al Tag 0xFF0011 le corresponde el mismo bloque (0xFF0011 mod 4 = 1)
      [-------- 0xFF0011 -------- --------]
      [-------- 0x000011 -------- --------]
Al Tag 0x000012 le corresponde (0x000012 mod 4 = 2)
      [-------- 0x000011 0x000012 --------]
      [-------- 0x000011 0x000012 0x000013]
Al Tag 0x000025 le corresponde (0x000025 mod 4 = 1)
Acá intentamos agregar el tag 0x000025
      [-------- 0x000025 0x000012 0x000013]
```

Formato de archivos

Archivos de accesos

Los archivos de accesos a memoria constarán de texto plano con la representación hexadecimal de las direcciones de memoria a la que accederán la o las cpus.

Archivos de configuración

Habrá un archivo de configuración con formato clave valor. Estos valores serán cargados en memoria y utilizados para configurar la cache.
Los valores mandatarios son:

- **cache type**: Puede tomar los valores associative-lru, associative-fifo, o direct. Se utiliza para determinar el tipo de cache y política de reemplazo.
- **cache size**: Tamaño de la cache en bytes. Se asume que siempre será potencia de 2.
- **line size**: Tamaño de la línea en bytes. También se asume que siempre es potencia de 2.

Además pueden estar los valores vendor_id, model name, cpu MHz con descripción de la CPU simulada, y la opción debug para indicar el nivel de verbosidad de la salida.

Formato de línea de comandos

La línea de comandos para ejecutar la aplicación es

```
./tp <archivo.cfg> <cpu-01.bin> [<cpu-NN.bin>]
```

Entrada y salida estándar

La aplicación no recibe datos mediante la entrada standard.

La salida standard será utilizada para informar los siguientes eventos:

- Cuando la cache fue inicializada

```
# Cache inicializada

Fabricante:
Modelo:
Cpu MHz:
```

  Nota: en caso de faltar uno de estos datos, rellenar con N/A
- Cuando todas las CPUs terminan de operar, la cache informa una estadística de accesos:

```
# Informe:
* Total de hits:
* Total de misses:
```

La salida de error estándar será utilizada para informar los siguientes eventos:

- El simulador no admite direcciones no alineadas, es decir, que no sean múltiplo de 4. En caso de detectar una se imprime el siguiente mensaje, y se termina la ejecución de la cpu ofensiva:

```
Dirección inválida: <Dirección en hexadecimal>
Abortando
```

Debug

Si exista la clave **debug** y tiene el valor true se imprimirá una línea hit: <dirección> o miss <dirección> por cada acceso a cache ocurrido.

Ejemplos de ejecución

Dada la siguiente configuración

```
vendor_id=Taller
model name=Example
cpu MHz=1600.02
cache type=associative-lru
```

```
cache size=64
line size=16
debug=true
```

Y el siguiente archivo de accesos cpu-00.bin

```
0x00000000
0x00000010
0x00000008
0x00000028
0x00000040
0x00000020
0x00000080
0x00000000
```

La salida esperada es

```
# Cache inicializada

* Fabricante: Taller
* Modelo: Example
* Cpu MHz: 1600.02

Miss: 0x00000000
Miss: 0x00000010
Hit: 0x00000008
Miss: 0x00000028
Miss: 0x00000040
Hit: 0x00000020
Miss: 0x00000080
Hit: 0x00000000

# Informe

* Total de hits: 3
* Total de misses: 5
```

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C++11.
2. Está prohibido el uso de variables globales.
3. Debe haber uso de al menos una clase abstracta, jerarquía de clases y polimorfismo.
4. La búsqueda de tags en los distintos tipos de cache no puede ser de orden lineal. Se recomienda estructuras estándar como **std::set** y **std::map**.[3]
5. Procesar cada archivo de **<cpu-NN.bin>** en un hilo propio (en paralelo) sobre una única cache (los hilos comparten esta cache)
6. La validación de alineación de memoria no puede bloquear la cache (mientras un hilo esta validando la alineacion otros pueden estar interactuando con la cache).
7. Todos los recursos compartidos deben ser protegidos contra las **race-conditions**. Piense en todos los recursos que los hilos comparten, la cache es uno de ellos pero no el único.

Recomendaciones

Se sugiere considerar el uso de las siguientes clases de la biblioteca STL para facilitar la implementación del trabajo práctico:

- std::map
- std::set
- std::deque

También se sugiere buscar qué prestaciones hay en el header `<iomanip>`

Referencias

[1] https://es.wikipedia.org/wiki/Cach%C3%A9_(inform%C3%A1tica)
[2] https://es.wikipedia.org/wiki/Algoritmo_de_cach%C3%A9
[3] http://www.cplusplus.com/reference/stl/