FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2
Memorias caché

[8637/6620] Organización de Computadoras
Curso 2
Segundo Cuatrimestre de 2020

Alumno:	Padrón
BIANCARDI, Julián	103945
CZOP, Santiago Nicolás	104057
OUTEIRO, Sebastián	92108

Índice

1. Introducción

En el siguiente informe se detallarán todos los pasos seguidos para la resolución del trabajo práctico 2 de la materia Organización de Computadoras (66.20). El mismo consiste en implementar una simulación de una memoria cache asociativa por conjuntos, en que se puedan pasar por parámetro el número de vıas, la capacidad y el tamaño de bloque. La politica de reemplazo será LRU y politica de escritura WB/WA. Se asume que el espacio de direcciones es de 16 bits, y hay entonces una memoria principal a simular con un tamaño de 64KB.

2. Desarrollo

Para el desarrollo del trabajo se dividio por bloques las tareas a realizar. Por un lado, detectar todos los comandos que el usuario ingresó para poder saber que operaciones ejecutar. Una vez que realizamos esto debemos leer los archivos que nos especificaron en la linea de comando y a medida que vamos leyendo debemos realiar las operaciones especificadas. Por último, los resultados que vayamos obteniendo debemos plasmarlos sobre un archivo de texto distinto del cual estabamos operando, o de no ser asi por la salida estandar (stdin).

2.1. Caché

La memoria caché cuenta con las siguientes primitivas:

- `int cache_init();`

  Inicializa la memoria caché, almacenado la memoria necesaria establecida por los parametros: cachesize, ways y blocksize pasados por la linea de comando.

- `unsigned int cache_find_set(int address);`

  Devuelve el set al que mapea la direccion de memoria.

- `unsigned int cache_find_lru(int setnum);`

  Devuelve el bloque menos recientemente usado dentro de un conjunto (o alguno de ellos si hay más de uno).

- `unsigned int cache_is_dirty(int way, int setnum);`

  Devuelve el estado del bit D(dirty) del bloque correspondiente.

- `void cache_read_block(int blocknum);`

  Lee el bloque blocknum de memoria guardandolo en el lugar que le corresponda en la memoria caché.

- `void cache_write_block(int way, int setnum);`

  Escribe en memoria los datos contenidos en el bloque setnum de la via way.

- `unsigned char cache_read_byte(int address);`

  Devuelve el valor correspondiente a la posición de memoria address, buscándolo primero en el caché.

- `void cache_write_byte(int address, unsigned char value);`

  Escribe el valor value en la posición correcta del bloque que corresponde a address.

- `float cache_get_miss_rate();`

  Devuelve el miss rate de la caché.

- `bool cache_hit();`

  Devuelve si el estado de la ultima operación fue hit o no.

- `void cache_uninit();`

  Destruye la memoria caché, liberando los recursos.

Para la implementecación de la memoria caché se ha optado por considerar los siguientes tipos de estructuras:

```
typedef struct block {
  unsigned int date;
  bool valid;
  bool dirty;
  unsigned char tag;
  unsigned char* data;
} block_t;
```

La siguiente estructura representa a un bloque de la memoria caché. Cuenta con un bit de validez, un bit de dirty, su correspondiente tag y la información que almacena. Además se consideró utilizar una variable date para determinar el intervalo de tiempo desde la ultima vez que fue accedido el mismo (esto nos servirá para la política LRU)

```
typedef struct set {
  block_t* blocks;
} set_t;
```

La siguiente estructura representa un set completo de la memoria caché.

```
typedef struct cache {
  set_t* sets;
  unsigned int hits;
  unsigned int misses;
  bool last_satuts;
  unsigned int setsnum;
  unsigned int bits_tag;
  unsigned int bits_set;
  unsigned int bits_offset;
} cache_t;
```

La siguiente estructura representa a la memoria caché asociativa por conjuntos. Contamos con la cantidad de bits tanto de tag, set y offset para una memoria principal de 64KB. Adicionalmente, se cuenta con la información de la cantidad de hits, cantidad de misses desde que se ha inicializado la caché más el estado de la última operación realizada sobre la misma.

2.2. Línea de comando

Para parsear la línea de comando se optó por usar la librería "getopt.h" (ver manual: `https://man7.org/linux/man-pages/man3/getopt.3.html`). En específico, se utilizó la función *getopt_long* que nos simplifica el procesamiento de los argumentos, pudiendo así definir una lógica distinta a partir de cada parámetro. El comportamiento se definió en su totalidad en el archivo *args_parser.c*. En caso de que este módulo, que fue diseñado específicamente para resolver este problema, devuelva un error entonces el programa frenará su ejecución.

2.3. Lectura de Datos

Para una lectura de los archivos de input de forma ordenada y que nos permitiera leer todo su contenido, fuimos por un diseño general que permite leer archivos línea por línea y procesarla una a la vez. Esta operación la concentra el TDA llamado *FileReader* el cual se inicializa para leer archivos con las siguientes dos funciones:

```
int file_reader_init(file_reader_t* self);
int file_reader_open(char* file_name);
```

Donde *file_ name* es el nombre del archivo del cual queremos leer. Si este parámetro es "stdin", se leerá de entrada estándar. En caso de no abrir un archivo, se leerá de entrada estándar por default. A través de la siguiente función podemos procesar el archivo:

```
int file_reader_process(file_reader_t* self, callback_t callback, void* context);
```

Esta última función mencionada nos será de gran utilidad puesto que podremos realizar distintas operaciones sin acoplarnos al FileReader. Esto se debe a que por cada lectura exitosa la función *callback* es invocada delegando así en el usuario qué hacer con cada línea leído.

2.4. Instrucciones

En este apartado mencionameremos los distintos formatos de instrucciones posibles para interactuar con la memoria caché:

- Las instrucciones de la forma "init" se ejecutan llamando a la función init para inicializar la caché y la memoria simulada.
- Las instrucciones de la forma "R ddddd" se ejecutan llamando a la función read byte(ddddd) e imprimiendo el resultado y si es hit o miss.
- Las instrucciones de la forma "W ddddd, vvv" se ejecutan llamando a la función write byte(int ddddd, char vvv) e imprimiendo si es hit o miss.
- Las instrucciones de la forma "MR" se ejecutan llamando a la función get miss rate() e imprimiendo el resultado.

En caso de no respetar el formato de ninguna de las anteriores o bien haber colocado datos invalidos, se infromará al usuario el error correspondiente.

Cabe mencionar que la instruccion init debe estar presente en el archivo a procesar. En caso contrario se le informará al usuario el error correspondiente.

2.5. Proceso de Instrucciones

Las instrucciones antes mencionadas las procesaremos mediante un TDA llamado *Processor*. Mediante la siguiente función podemos procesar cada instruccion que vayamos leyendo del archivo y por cada una escribiendo su resultado esperado:

```
void process_and_output(char* instr, size_t instr_len, file_writer_t* file_writer)
```

Cabe destacar que esta función cumple el rol de callback en los llamados a file reader. Esto es asi debido a que fue necesario crear funciones que permitieran hacer de "puente" entre los distintos módulos: *main*, *file reader*, *file writer*y *procesor*.

2.6. Salida

La salida de las operaciones antes mencionadas serán escritas en un archivo especificado por el comando "-o" (o en su ausencia, en la salida estándar). Para esta operación se optó por implementar un TDA llamado *FileWriter* el cual controla las operaciones de escritura, apertura y cierre de archivos. Los posibles formatos de salida son:

- READ address ->value - [HIT/MISS]
- WRITE value ->address - [HIT/MISS]
- MISS-RATE: mr %

3. Ejecución de Pruebas

En el siguiente apartado se mostrarán las pruebas realizadas para probar el codigo del trabajo y garantizar así el correcto funcionamiento en todos los escenarios posibles.

3.1. Pruebas Unitarias

Se cuenta con un total de 18 pruebas unitarias que en conjunto garantizan el correcto funcionamiento de la caché.

```
CACHE

The cache gets its properties set correctly: OK
The cache is initialized correctly: OK
The cache is unitialized correctly: OK
The cache is initially fully clean: OK
The cache finds a set from an address: OK
The cache misses when first reading a byte: OK
The cache hits when rereading a byte: OK
The cache correctly measures reading miss-rate: OK
The cache misses when set and offset are equal but not the tag: OK
The cache doesn't replace a block if another way is free: OK
The cache finds the least recently used block correctly: OK
The cache replaces the least recently used block if the set is full: OK
The cache hits all the block if it was loaded for reading: OK
The cache reads the correct value after writing: OK
The cache misses when first writing a byte: OK
The cache hits when rewriting a byte: OK
The cache correctly measures writing miss-rate: OK
The cache hits all the block if it was allocated for writing: OK
```

3.2. Pruebas de Enunciado

Para cumplir con las pautas especificadas en el enunciado, se presenta a continuación 4 pruebas solicitadas en el mismo. Las pruebas son:

1. Cache 4KB, 4WSA, 32B blocks - Archivo prueba1.mem
2. Cache 4KB, 4WSA, 32B blocks - Archivo prueba2.mem
3. Cache 16KB, 1 vía, 128B blocks - Archivo prueba1.mem

4. Cache 16KB, 1 vía, 128B blocks - Archivo prueba2.mem

Los archivos *prueba1.mem* y *prueba2.mem* se detallan a continuación:

prueba1.mem

```
W 0, 255
W 16384, 254
W 32768, 248
W 49152, 096
R 0
R 16384
R 32768
R 49152
MR
```

prueba2.mem

```
W 0, 123
W 1024, 234
W 2048, 33
W 3072, 44
W 4096, 55
R 0
R 1024
R 2048
R 3072
R 4096
MR
```

A continuación se presentan las 4 salidas de las pruebas, según la enumeración antes indicada.

Prueba 1

```
WRITE FF ->0 - MISS
WRITE FE ->16384 - MISS
WRITE F8 ->32768 - MISS
WRITE 60 ->49152 - MISS
READ 0 ->FF - HIT
READ 16384 ->FE - HIT
READ 32768 ->F8 - HIT
READ 49152 ->60 - HIT
MISS-RATE: 50.0 %
```

Prueba 2

WRITE 7B ->0 - MISS
WRITE EA ->1024 - MISS
WRITE EA ->1025 - HIT
WRITE 21 ->2048 - MISS
WRITE 2C ->3072 - MISS
WRITE 37 ->4096 - MISS
READ 0 ->7B - MISS
READ 1024 ->EA - MISS
READ 2048 ->21 - MISS
READ 3072 ->2C - MISS
READ 4096 ->37 - MISS
MISS-RATE: 90.9 %

Prueba 3

WRITE FF ->0 - MISS
WRITE FE ->16384 - MISS
WRITE F8 ->32768 - MISS
WRITE 60 ->49152 - MISS
READ 0 ->FF - MISS
READ 16384 ->FE - MISS
READ 32768 ->F8 - MISS
READ 49152 ->60 - MISS
MISS-RATE: 100.0 %

Prueba 4

WRITE 7B ->0 - MISS
WRITE EA ->1024 - MISS
WRITE EA ->1025 - HIT
WRITE 21 ->2048 - MISS
WRITE 2C ->3072 - MISS
WRITE 37 ->4096 - MISS
READ 0 ->7B - HIT
READ 1024 ->EA - HIT
READ 2048 ->21 - HIT
READ 3072 ->2C - HIT
READ 4096 ->37 - HIT
MISS-RATE: 45.5 %

4. Código Fuente

Se puede observar el código implementado en la siguiente dirección, en la carpeta TP2. Se proporciona un archivo Makefile para compilar el ejecutable.

`https://github.com/JulianBiancardi/Organizacion-de-Computadoras-66.20`

El código fuente del presente informe se encuentra disponible para lectura en:
`https://www.overleaf.com/read/cjdvgvmmwhbh`

5. Conclusiones

En conclusión, se considera que se ha logrado implementar una solución elegante y que cumple con las indicaciones. A lo largo del proceso de diseño, programación y testeo se incorporó mucho conocimiento tanto sobre la politica de escritura WB/WA (write back and write allocate), como a la política de reemplazo LRU (least recently used).

Referencias

[1] Linux.die.net. 2020. *Getopt_Long(3): Parse Options - Linux Man Page.* [online] Disponible en: <https://linux.die.net/man/3/getopt_long>(Accedido 21 Octubre 2020).

[2] Gnu.org. n.d. *GNU Make.* [online] Disponible en:
<https://www.gnu.org/software/make/manual/make.html>(Accedido 12 Noviembre 2020).