Trabajo Práctico 2
Memoria Caché

Sebastian Ignacio Penna, *Padrón 98752*
sebapenna19@gmail.com

Constanza M Frutos Ramos, *Padrón 96728*
constanza.frutos.ramos@gmail.com

1er. Cuatrimestre de 2019
86.37 / 66.20 Organización de Computadoras – Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

El trabajo práctico consta de la realización de una memoria caché con ciertas características permitiendo asentar los conocimientos vistos en clase.

Índice

1. Introducción

El objetivo principal de trabajo es poder entender el funcionamiento interno de la caché mediante su implementación en C.

En el enunciado se indican las primitivas a implementar. Haciendo uso de estas se debe hacer un programa que recibe comandos a aplicar sobre la caché en un archivo de texto.

Para esto hay cinco casos de prueba que se ejecutan y se chequea que la salida obtenida sea la esperada.

2. Diseño e Implementación

2.1. Análisis del problema

Se pide simular una memoria caché con las siguientes especificaciones:

- Asociativa por conjuntos de cuatro vías; (4WSA)
- 2KB de capacidad; (CS)
- Bloques de 64B; (BS)
- Política de reemplazo FIFO;
- Política de escritura WT/¬WA.

A su vez el espacio de direcciones se asume de 16 bits, teniendo así una memoria principal 64KB.

Lo primero que se hace es analizar el problema. Dado que la caché tiene un tamaño de 2KB, cada bloque es de 64B y esta separada en 4 vías, se obtiene la cantidad de bloques de cada caché mediante la siguiente operación:

$$\#Bloques = \frac{CS}{BS * \#Vias}$$

Donde:

- CS = 2 KB = $2 * 2^{10}$ B = 2^{11} B;
- BS = 64 B = 2^6 B;
- #Vías = 4.

Entonces:

$$\#Bloques = \frac{2^{11}}{2^6 * 2^2} = 8$$

Hay 8 bloques por caché, esto quiere decir que el índice va de 0 a 7, por lo cual se requieren de 3 bits para direccionarlo. El offset se obtiene conociendo el tamaño del bloque y cuantas direcciones hay en este. Se direcciona al Byte, y puesto que el bloque tiene 64B se requieren de 6 bits de offset para direccionarlo. Y dado que las direcciones de memoria son de 16 bits, los restantes son de tag, es decir 7 bits.

2.2. Primitivas de la caché

- **void init()**: inicializa toda la memoria necesaria para el espacio representado por la cache al igual que los distintos atributos de las estructuras antes mencionadas. Inicialmente todos los bloques tendrán su bit V seteado en false ya que no tienen datos;

- **unsigned int get_offset (unsigned int address)**: para el calculo del offset se realizar la operación *and* (&) entre la dirección brindada como parámetro y el valor hexa 0x3F, el cuál nos devuelve el valor a representar con los primeros 6 bits;
- **unsigned int find_set(unsigned int address)**: en este caso se realizá la operación *and* pero con el valor hexa 0x1C0 y luego se realizá un shift a derecha de 6 bits (los que representan al offset). De esta manera se tendrán en los primeros 3 bis el valor presente en en index;
- **unsigned int select_oldest(unsigned int setnum)**: teniendo el set sobre el cuál se quiere obtener el bloque más viejo se itera sobre los 4 bloques relativos al mismo y se compara sucesivamente el atributo *orden*. De ésta manera, al finalizar se tendrá el valor de la vía dónde se tuvo el primer bloque;
- **void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)**: en éste caso se sobreescribe un bloque de la caché con los datos provistos como párametro. Nuevamente se deberá pedir memoria al heap y liberar la ya existente;
- **void write_tocache(unsigned int address, unsigned char)**: ésta función consta tán solo de setear el valor indicado dentro de la dirección correspondiente en la memoria principal, es decir el vector en dicha posición;
- **unsigned char read_byte(unsigned int address)**: en primer lugar se obtienen tanto el index como el offset respectivo a la addres indicada y con el index se busca si alguno de los bloques del set indicado comparte el mismo tag de la address. Si así fuese se retorna el byte correspondiente al offset, vía e index que corresponda. De lo contrario se procederá a contabilizar un miss y almacenar el dato en caché. Algo a detallar también es que para seleccionar la vía donde almacenar el dato en caché se deberá evaluar si el set se encuentra lleno (todas las vías utilizadas) y si así fuese se debería aplicar el protocolo FIFO para reemplazar los datos, sino se utilizará una nueva vía del set;
- **void write_byte(unsigned int address, unsigned char value)**: al igual que en casos previos se calculan tanto el offset como el index de la address indicada. Luego se determina si alguna vía dentro del set coincide con el dato de la address y de así ser se almacena el nuevo valor en la cache. De no encontrar coincidencia se contabilizará un nuevo miss. En ambos casos por último se almacenará el dato en la memoria principal;
- **float get_miss_rate()**: ésta función se encarga de realizar el cálculo en base a los datos de los cuales se llevo régistro, siendo éste: $\frac{\#Misses}{\#Accesos}$;

2.3. Estructuras

2.3.1. Memoria

```
typedef struct memoria {
    unsigned char* direcciones[CANTIDAD_DIRECCIONES];
} memoria_t;
```

Se implementa la memoria como un arreglo de direcciones. La cantidad de direcciones es una constante y esta dada en el enunciado.

2.3.2. Bloque

```
typedef struct bloque {
    unsigned char* direcciones[TAMANIO_BLOQUE];
    unsigned int tag;
    bool v;
    unsigned int orden;
} bloque_t;
```

El bloque contiene sus datos en un arreglo que tiene:

- 64 elementos, uno por cada byte;
- el tag, el cual es propio del bloque;
- el bit de validez v para indicar si el bloque es válido o no;
- el orden de dicho bloque, este se usa para implementar la política de reemplazo (FIFO en este caso).

2.3.3. Set

```
typedef struct set {
    bloque_t* bloques[BLOQUES_POR_SET];
     int latest;
} set_t;
```

Cada set tiene 4 bloques, y el atributo latest que indica cual fue el último bloque en ser agregado al set.

2.3.4. Cache

```
typedef struct cache {
     set_t* sets[CANTIDAD_SETS];
     float total_accesos;
     float misses;
} cache_t;
```

La caché contiene sus 8 sets, cada uno con los atributos explicados anteriormente. A su vez acumula la cantidad de accesos totales que hubo a la misma, y la cantidad de misses. Estos dos atributos se usan para calcular la tasa de miss.

2.4. Comandos del programa

- FLUSH
- R ddddd
- W ddddd, vvv
- MR

2.5. Desarrollo del Código Fuente

Los archivos con el código fuente que se utilizarán para la compilación y ejecución del programa son los siguientes:

1. **cache.c**: Contiene la implementación de las primitivas mostradas anteriormente. Además de algunas funciones auxiliares que se usan en las primitivas y dos funciones para imprimir la caché y la memoria.
2. **main.c**: Contiene la implementación del programa principal, el cual recibe el archivo con los comandos y los ejecuta.

2.6. Análisis de la línea de comandos

La aplicación se ejecuta haciendo uso de la línea de comando mostrada a continuación:

```
./tp2 archivo_texto.mem
```

En el archivo se colocan los comandos a ser ejecutados separados por un salto de línea.

3. Proceso de Compilación

Se usa gcc para compilar. Para compilar tán sólo con código c el comando utilizado es el siguiente:

```
gcc -Wall -Werror -std=c99 cache.c main.c -o tp2
```

4. Casos de Prueba

4.1. Prueba1

```
W 0, 255   //El bloque 0 no se encuentra en cache. Hay un miss. Se escribe 255 en la
    direccion 0 de memoria.
W 1024, 254 //El bloque 1024 no se encuentra en cache. Hay un miss. Se escribe 254 en la
    direccion 1024 de memoria.
W 2048, 248 //El bloque 2048 no se encuentra en cache. Hay un miss. Se escribe 248 en la
    direccion 2048 de memoria.
W 4096, 096 //El bloque 4096 no se encuentra en cache. Hay un miss. Se escribe 096 en la
    direccion 4096 de memoria.
W 8192, 192 //El bloque 8192 no se encuentra en cache. Hay un miss. Se escribe 192 en la
    direccion 8192 de memoria.
R 0     //El bloque 0 no se encuentra en cache. Hay un miss. Se trae el bloque a cache, el
    cual va al set 0, via 0 (dado que no hay nada en cache).
R 1024 //El bloque 1024 no se encuentra en cache. Hay un miss. Se trae el bloque a cache,
    el cual va al set 0, via 1 porque la 0 recien fue cargada.
R 2048 //El bloque 0 no se encuentra en cache. Hay un miss. Se trae el bloque a cache, el
    cual va al set 0, via 2 porque la 1 recien fue cargada.
R 8192 //El bloque 0 no se encuentra en cache. Hay un miss. Se trae el bloque a cache, el
    cual va al set 0, via 3, porque la 2 recien fue cargada.
MR      //miss_rate = 9/9 = 1
```

4.2. Prueba2

```
R 0     //El bloque 0 no se encuentra en cache. Hay un miss. Se trae a cache en el set 0,
    via 0.
R 31    //El bloque ya se encuentra en cache. Es un hit.
W 64, 10 //El bloque no se encuentra en cache. Hay un miss. Se escribe en memoria, no se
    trae el bloque(NWA).
R 64    //El bloque ya se encuentra en cache. Hay un hit.
W 64, 20 //El bloque esta en cache. Se escribe en cache y en memoria el valor 20 en la
    direccion 64.
R 64    //El bloque ya esta en cache.
MR      //miss_rate = 3/6 = 0.5
```

4.3. Prueba3

```
W 128, 1   //El bloque no se encuentra, escribe en memoria y no trae el bloque. Hay un
    miss. Se pone un 1 en la direccion 128.
W 129, 2   //El bloque no se encuentra, escribe en memoria y no trae el bloque. Hay un
    miss. Se pone un 2 en la direccion 129.
W 130, 3   //El bloque no se encuentra, escribe en memoria y no trae el bloque. Hay un
    miss. Se pone un 3 en la direccion 130.
W 131, 4   //El bloque no se encuentra, escribe en memoria y no trae el bloque. Hay un
    miss. Se pone un 4 en la direccion 131.
R 1152      //El bloque no se encuentra en cache, hay un miss. Lo carga en cache.
R 2176      //El bloque no se encuentra en cache, hay un miss. Lo carga en cache.
R 3200      //El bloque no se encuentra en cache, hay un miss. Lo carga en cache.
R 4224      //El bloque no se encuentra en cache, hay un miss. Lo carga en cache.
R 128       //El bloque no se encuentra en cache, hay un miss. Lo carga en cache.
R 129       //El bloque se encuentra en cache, es el que se cargo anteriormente.
R 130       //El bloque se encuentra en cache, es el que se cargo anteriormente.
```

```
R 131      //El bloque se encuentra en cache, es el que se cargo anteriormente.
MR         //miss_rate = 9/12 = 0.75
```

En las primeras 4 lecturas los bloques van al mismo set. Esto significa que cuando se lee el primer bloque, se coloca en la vía 0, el correspondiente a la dirección 2176 en la vía 1, el el correspondiente a la dirección 3280 en la vía 2, y el el correspondiente a la dirección 4224 en la vía 3. Luego, al leer el bloque que contiene a la dirección 128 se reemplaza el mas "viejo"(por ser la política de reemplazo FIFO) por el nuevo. Quedando así el nuevo bloque en la vía 0. Las próximas 3 lecturas dan hit dado que el bloque ya esta en caché.

4.4. Prueba 4

```
W 0, 256   //La direccion 0 no se encuentra en cache. Hay un miss. Se escribe el valor 256
     en la direccion 0 de memoria.
W 1, 2 //La direccion 1 no se encuentra en cache. Hay un miss. Se escribe el valor 2 en la
     direccion 1 de memoria.
W 2, 3 //La direccion 2 no se encuentra en cache. Hay un miss. Se escribe el valor 3 en la
     direccion 2 de memoria.
W 3, 4 //La direccion 3 no se encuentra en cache. Hay un miss. Se escribe el valor 4 en la
     direccion 3 de memoria.
W 4, 5 //La direccion 4 no se encuentra en cache. Hay un miss. Se escribe el valor 5 en la
     direccion 4 de memoria.
R 0    //La direccion 0 no esta en cache. Se lee el bloque en el set 0, via 0. Hay un miss.
R 1    //La direccion 1 esta en cache. Hay un hit.
R 2    //La direccion 2 esta en cache. Hay un hit.
R 3    //La direccion 3 esta en cache. Hay un hit.
R 4    //La direccion 4 esta en cache. Hay un hit.
R 4096 //La direccion 4096 no se encuentra en cache. Hay un miss. Se lee el bloque en el
    set 0, via 1.
R 8192 //La direccion 8192 no se encuentra en cache. Hay un miss. Se lee el bloque en el
    set 0, via 2.
R 0    //La direccion 0 esta en cache. Hay un hit.
R 1    //La direccion 1 esta en cache. Hay un hit.
R 2    //La direccion 2 esta en cache. Hay un hit.
R 3    //La direccion 3 esta en cache. Hay un hit.
R 4    //La direccion 4 esta en cache. Hay un hit.
MR     //miss_rate = 8/17 = 0.470
```

4.5. Prueba 5

```
R 131072   //La direccion a leer es invalida, esta fuera del rango de direcciones
    disponibles. Se emite mensaje de error.
R 4096     //La direccion 4096 no se encuentra en cache. Se lee el bloque en el set 0, via
     0. Hay un miss.
R 8192     //La direccion 8192 no se encuentra en cache. Se lee el bloque en el set 0, via
     1. Hay un miss.
R 4096     //La direccion 4096 se encuentra en cache. Hay un hit.
R 0        //La direccion 0 no se encuentra en cache. Se lee el bloque en el set 0, via 2.
     Hay un miss.
R 4096     //La direccion 4096 se encuentra en cache. Hay un hit.
MR         //miss_rate = 3/5 = 0.6
```

4.6. Funcionamiento de la aplicación

4.6.1. Valgrind

Al correr la aplicación con valgrind se demuestra que no hay perdida de memoria en la misma:

```
==17392== HEAP SUMMARY:
==17392==     in use at exit: 0 bytes in 0 blocks
==17392==   total heap usage: 35,121 allocs, 35,121 frees, 322,288 bytes allocated
==17392==
==17392== All heap blocks were freed -- no leaks are possible
==17392==
==17392== For counts of detected and suppressed errors, rerun with: -v
==17392== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 1: Ejecución con Valgrind.

5. Conclusión

Siendo el objetivo del trabajo el desarrollo e implementación de una memoria principal y caché simuladas, se pueden extraer ciertas conclusiones una vez resuelto el mismo y llegado al resultado deseado.

En primer lugar se trabajo en detalle con la serie de pasos existentes a la hora de trabajar con una memoria caché. Hasta previo al trabajo los conocimientos del uso de la misma fueron principalmente teóricos y prácticos en ciertos ejercicios, pero en ningún momento se observó el desarrolló y progreso de la memoria caché a medida que se sumaban una cantidad relevante de datos (aunque no de gran magnitud en las pruebas realizadas). Llevando a cabo las distintas primitivas que se indicaron en la consigna de éste trabajo se logro un entendimiento mayor de los distintos procesos que se llevan a cabo en el uso de una caché y la manera en que están serializados.

Tanto a la hora de escribir sobre caché como memoria principal se tuvo que implementar todos los pasos previos que éstas tareas requieren y en base a éstos se pueden observar las ventajas y diferencias que tiene una caché como la que se estableció en el enunciado, de 4 vías y la cantidad memoria indicada.

Dado que se trataba de una memoria relativamente chica y de bajo direccionamiento (tán sólo 16 bits) las distintas iteraciones que se llevaron a cabo tuvieron un bajo efecto en el tiempo, sin embargo a medida que se incremente el tamaño de la misma o se aumente el direccionamiento de las direcciones se verá también un aumento seguramente en el tiempo que necesitará el programa implementado para almacenar y obtener los distintos datos, mostrando la importancia de que la caché éste optimizada y adecuada al problema en particular con el que se traté.

Otro detalle relevante es que se tiene una caché de 4 vías, lo que afecto al trabajo, ya que si se hubiese tenido por ejemplo una caché direct mapped o fully associative el desarrollo del mismo tendría que haber sido distinto por supuesto y la forma de obtener los datos habría sido significativamente modificada. En una fully associative cada dirección podría haber utilizado cualquier bloque de caché, incrementando la cantidad de iteraciones a realizar, que ya no estarían limitadas a un cierto set o bloque.

En conclusión, el desarrollo de una caché (simulada en éste caso) es de suma relevancia para la eficiencia en el uso de datos, ya que a través de la misma se reducirán los tiempos de trabajo y accesos a memoria. Una mala implementación o diseño se verá reflejado a la hora de trabajar y, en nuestro caso, a la hora de ejecutar las distintas pruebas contra el modelo escogido.

Referencias

[1] Computer Arquitechture: A quantitative aproach. Capítulo 5. Memory-Hierarchy Design.

[2] Computer organization and design: The hardware/software interface. 3 edition. Capítulo 7. Large and Fast: Exploiting Memory Hierarchy.

[3] The C programming language, Kernigan and Ritchie,

[4] Valgrind, valgrind-3.6.0.SVN-Debian, http://valgrind.org/

6. Código

6.1. cache.h

```
#ifndef CACHE_H
#define CACHE_H

#define CANTIDAD_DIRECCIONES 32768 //32KB

//Post: La memoria principal simulada queda
//inicializada en 0, los bloques de caché como
//inválidos y la tasa de misses en 0.
void init();

//Pre: recibe una address como unsigned int
//Post: devuelve el offset del byte del bloque de memoria al que
//mapea la dirección address.
unsigned int get_offset(unsigned int address);

//Pre: recibe una address como unsigned int
//Post: devuelve el conjunto de caché al que
//mapea la dirección address.
unsigned int find_set(unsigned int address);

//Pre: Recibe setnum como unsigned int.
//Post: devuelve la vía en la que esta el bloque mas
//"viejo" dentro de un conjunto, utilizando el campo
//correspondiente de los metadatos de los bloques del conjunto.
unsigned int select_oldest(unsigned int setnum);

//Pre: Recine un blocknum, la vía y el conjunto.
//Lee el bloque blocknum de memoria y lo guarda en el conjunto y
//vía indicados en la memoria caché.
//Post: Deja a la caché en el estado indicado.
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set);

//Pre:
//Post:
void write_tocache(unsigned int address, unsigned char value);

//Pre: Recibe una dirección.
//Busca el valor del byte correspondiente a la posición
//address en la caché. Si no se encuentra carga el bloque.
//Post: Retorna el valor del byte almacenado en la
//dirección indicada.
unsigned char read_byte(unsigned int address);

//Pre: Recibe una dirección y un valor.
//Escribe el valor value en la posición address de memoria,
//y en la posición correcta del bloque que corresponde a address,
//si el bloque se encuentra en la caché. Si no se encuentra,
//debe escribir el valor solamente en la memoria.
void write_byte(unsigned int address, unsigned char value);

//Post: devuelve el porcentaje de misses desde que se inicializó la caché.
float get_miss_rate();

void free_resources();

void print_cache();

void print_memoria();

#endif //CACHE_H
```

6.2. cache.c

```
#include "cache.h"

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#define TAMANIO_BLOQUE 64 //[B]
#define CANTIDAD_SETS 8
#define BLOQUES_POR_SET 4
#define BITS_OFFSET 6
#define BITS_INDEX 3
#define ESPACIO_DIRECCIONES 16 //[b]

#define RESET_COLOR  "\x1b[0m"
#define ROJO_T   "\x1b[31m"
#define VERDE_T      "\x1b[32m"

typedef struct memoria {
       unsigned char* direcciones[CANTIDAD_DIRECCIONES];
} memoria_t;

typedef struct bloque {
       unsigned char* direcciones[TAMANIO_BLOQUE];
       unsigned int tag;
       bool v;
       unsigned int orden;
} bloque_t;

typedef struct set {
       bloque_t* bloques[BLOQUES_POR_SET];
       int latest;
} set_t;

typedef struct cache {
       set_t* sets[CANTIDAD_SETS];
       float total_accesos;
       float misses;
} cache_t;

/* Memorias */
memoria_t* memoria;
cache_t* cache;

//Post: La memoria principal simulada queda inicializada en 0, los bloques de
// caché como inválidos y la tasa de misses en 0.
void init() {
       memoria = calloc(1, sizeof(memoria_t));
       for (unsigned int i=0; i<CANTIDAD_DIRECCIONES; ++i)
              memoria->direcciones[i] = calloc(1, sizeof(unsigned char));

       cache = calloc(1, sizeof(cache_t));
       for (unsigned int i=0; i<CANTIDAD_SETS; ++i) {
              cache->sets[i] = calloc(1, sizeof(set_t));
              cache->sets[i]->latest = -1;
              for (unsigned int j=0; j<BLOQUES_POR_SET; ++j){
                     cache->sets[i]->bloques[j] = calloc(1, sizeof(bloque_t));
                     for (int k=0; k<TAMANIO_BLOQUE; k++)
                            cache->sets[i]->bloques[j]->direcciones[k] = calloc(1, sizeof
                                (unsigned char));
                     cache->sets[i]->bloques[j]->v = false;
                     cache->sets[i]->bloques[j]->orden = 0;
              }
       }
       cache->total_accesos = 0;
```

```
        cache->misses = 0;
}

//Pre: recibe una address como unsigned int
//Post: devuelve el offset del byte del bloque de memoria al que
//mapea la dirección address.
unsigned int get_offset(unsigned int address) {
        return (address & 0x3f);
}

//Pre: recibe una address como unsigned int
//Post: devuelve el conjunto de caché al que mapea la dirección address.
unsigned int find_set(unsigned int address) {
        return ((address & 0x1c0) >> BITS_OFFSET);
}

unsigned int get_tag(unsigned int address) {
        return ((address & 0xfe00) >> (BITS_OFFSET + BITS_INDEX));
}

//Pre: Recibe setnum como unsigned int.
//Post: devuelve la vía en la que esta el bloque mas
//"viejo" dentro de un conjunto, utilizando el campo
//correspondiente de los metadatos de los bloques del conjunto.
unsigned int select_oldest(unsigned int setnum) {
        unsigned int way = 0;
        unsigned int orden_mas_viejo = 1000000;
        for (int i=0; i<BLOQUES_POR_SET; ++i)
                if (cache->sets[setnum]->bloques[i]->orden <= orden_mas_viejo) {
                        orden_mas_viejo = cache->sets[setnum]->bloques[i]->orden;
                        way = i;
                }
        return way;
}

//Pre: Recibe un blocknum, la vía y el conjunto.
//Lee el bloque blocknum de memoria y lo guarda en el conjunto y
//vía indicados en la memoria caché.
//Post: Deja a la caché en el estado indicado.
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set) {
        int inicio = blocknum - blocknum % BLOQUES_POR_SET;
        bloque_t* bloque = calloc(1, sizeof(bloque_t));
        for (int i=0; i<TAMANIO_BLOQUE; ++i) {
                bloque->direcciones[i] = calloc(1, sizeof(unsigned char));
                *bloque->direcciones[i] = *memoria->direcciones[inicio + i];
        }
        bloque->v = true;
        bloque->orden = cache->sets[set]->latest + 1;
        bloque->tag = get_tag(blocknum);

        bloque_t* aux = cache->sets[set]->bloques[way];
    for (int k=0; k<TAMANIO_BLOQUE; ++k)
        free(aux->direcciones[k]);
    free(aux);

        cache->sets[set]->bloques[way] = bloque;
        cache->sets[set]->latest = bloque->orden;
}

//Devuelve -1 si no lo encuentra
int get_way(unsigned int address, unsigned int idx) {
        unsigned int tag = get_tag(address);
        for (int i = 0; i < BLOQUES_POR_SET; ++i)
                if (cache->sets[idx]->bloques[i]->v){
                        unsigned int other_tag = cache->sets[idx]->bloques[i]->tag;
                        if (tag == other_tag)
```

```
                        return i;
                }
        return -1;
}

bool set_is_full(unsigned int set) {
        for (int i=0; i<BLOQUES_POR_SET; ++i)
                if (!cache->sets[set]->bloques[i]->v)
                        return false;
        return true;
}

//Pre: Recibe una dirección.
//Busca el valor del byte correspondiente a la posición
//address en la caché. Si no se encuentra carga el bloque.
//Post: Retorna el valor del byte almacenado en la
//dirección indicada.
unsigned char read_byte(unsigned int address) {
        unsigned int offset = get_offset(address);
        unsigned int idx = find_set(address);
        cache->total_accesos ++;
        int i = get_way(address, idx);
        if (i != -1)
                return *cache->sets[idx]->bloques[i]->direcciones[offset];
        else
                cache->misses ++;

        unsigned int way;
        if (set_is_full(idx))
                way = select_oldest(idx);
        else
                way = (cache->sets[idx]->latest + 1) % BLOQUES_POR_SET;

        read_tocache(address, way, idx);

        return *cache->sets[idx]->bloques[way]->direcciones[offset];
}

//Escribe en memoria.
void write_tocache(unsigned int address, unsigned char value) {
        *memoria->direcciones[address] = value;
}

//Pre: Recibe una dirección y un valor.
//Escribe el valor value en la posición address de memoria,
//y en la posición correcta del bloque que corresponde a address,
//si el bloque se encuentra en la caché. Si no se encuentra,
//debe escribir el valor solamente en la memoria.
void write_byte(unsigned int address, unsigned char value) {
        unsigned int offset = get_offset(address);
        unsigned int idx = find_set(address);

        cache->total_accesos ++;
        int i = get_way(address, idx);
        if (i != -1)
                *cache->sets[idx]->bloques[i]->direcciones[offset] = value;
        else
                cache->misses ++;

        write_tocache(address, value);
}

//Post: devuelve el porcentaje de misses desde que se inicializó la caché.
float get_miss_rate() {
        float miss_rate = cache->misses / cache->total_accesos;
        return miss_rate;
```

```
}

void free_memory() {
       for (unsigned int i=0; i<CANTIDAD_DIRECCIONES; ++i){
              free(memoria->direcciones[i]);
       }
       free(memoria);
}

void free_cache() {
       for (unsigned int i=0; i<CANTIDAD_SETS; ++i) {
              for (unsigned int j=0; j<BLOQUES_POR_SET; ++j) {
                     for (int k=0; k<TAMANIO_BLOQUE; ++k)
                            free(cache->sets[i]->bloques[j]->direcciones[k]);
                     free(cache->sets[i]->bloques[j]);
              }
              free(cache->sets[i]);
       }
       free(cache);
}

void free_resources() {
       free_memory();
       free_cache();
}

void print_cache() {
       for (int i=0; i<CANTIDAD_SETS; ++i){
              printf("set␣%i\n", i);
              for (int j=0; j<BLOQUES_POR_SET; ++j){
                     printf("Bloque␣%i\n", j);
                     printf("t:␣%u|v:␣%i|o:␣%u␣", cache->sets[i]->bloques[j]->tag,
                                                  cache->sets[i]->bloques[j]->v,
                                                  cache->sets[i]->bloques[j]->orden);
                     for (int k=0; k<TAMANIO_BLOQUE; ++k){
                            if (cache->sets[i]->latest == cache->sets[i]->bloques[j]->
                                orden
                                   && cache->sets[i]->bloques[j]->orden != 0)
                                   printf(VERDE_T "|%i|" RESET_COLOR, *cache->sets[i]->
                                       bloques[j]->direcciones[k]);
                            else if (cache->sets[i]->bloques[j]->v)
                                   printf(ROJO_T "|%i|" RESET_COLOR, *cache->sets[i]->
                                       bloques[j]->direcciones[k]);
                            else
                                   printf("|%i|", *cache->sets[i]->bloques[j]->
                                       direcciones[k]);
                     }
                     printf("\n");
              }
              printf("\n\n");
       }
}

void print_memoria() {
       for (int i=0; i<CANTIDAD_DIRECCIONES; ++i){
              if (*memoria->direcciones[i] != 0)
                     printf(ROJO_T "|%i|" RESET_COLOR, *memoria->direcciones[i]);
              else
                     printf("|%i|", *memoria->direcciones[i]);
       }
       printf("\n\n");
}
```

6.3. main.c

```
#include "cache.h"
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define MAX_LEN 13
#define FLUSH "FLUSH"
#define R "R␣"
#define W "W"
#define MR "MR"

bool is_address_valid(unsigned int address){
       return (address < CANTIDAD_DIRECCIONES);
}

bool is_flush_command(char* command){
       return (strstr(command, FLUSH));
}

bool is_r_command(char* command){
       return (strstr(command, R));
}

bool is_w_command(char* command){
       return (strstr(command, W));
}

bool is_mr_command(char* command){
   return (strstr(command, MR));
}

void parse_w_command(char* command, unsigned int *address, unsigned char *value){
       unsigned int n = 0;
       sscanf(command, "%*s␣%u,␣%u", address, &n);
       *value = n;
}

void parse_r_command(char* command, unsigned int *address){
       sscanf(command, "%*s␣%u", address);
}

int main(int argc, char* argv[]){
       init();

       if (argc < 2)
              printf("Falta␣el␣nombre␣del␣archivo.\n");

       FILE* file;
       file = fopen(argv[1], "r");
       if (!file) {
              perror("Error␣al␣abrir␣el␣archivo.");
              exit (1);
       }

       char command[MAX_LEN];
       while (!feof(file)) {
              if (fgets(command, MAX_LEN, file)) {
                     printf("%s\n", command);
                     if (is_flush_command(command)) {
                            init();
                     } else if (is_r_command(command)) {
                            unsigned int address;
                            parse_r_command(command, &address);
                            if (is_address_valid(address)) {
                                   unsigned char value = read_byte(address);
                                   printf("Value:␣%i\n", value);
```

```
                                print_cache();
                            } else {
                                perror("Comando␣invalido.\n");
                            }
                        } else if (is_w_command(command)) {
                            unsigned int address;
                            unsigned char value;
                            parse_w_command(command, &address, &value);
                            if (is_address_valid(address)) {
                                write_byte(address, value);
                                print_cache();
                                print_memoria();
                            } else {
                                perror("Comando␣invalido.\n");
                            }
                        } else if (is_mr_command(command)) {
                    printf("Miss␣rate:␣%f\n", get_miss_rate());
                } else {
                    perror("Comando␣invalido.\n");
                }
                    }
        }
        print_cache();
//      print_memoria();
        free_resources();
        fclose(file);
        return 0;
}
```