Trabajo Práctico 2 — Memorias caché

[66.20] Organización de Computadoras
Curso 2
Primer cuatrimestre de 2021

Alumnos	Padrón	Email
ARRACHEA, Tomás	104393	tarrachea@fi.uba.ar

Índice

1. Diseño y detalles de implementación

Se implementó la memoria caché como una estructura que contiene las dimensiones variables de la memoria (la cantidad de vías, capacidad y tamaño de bloques), un vector de bloques de tamaño igual a la cantidad de bloques, y un vector de topes de tamaño igual a la cantidad de conjuntos de la memoria. A cada conjunto corresponde un número que indica dónde esta el bloque tope, que será el último que ingresó. Esto permite implementar la política de reemplazo FIFO.

Además, se modeló la memoria principal como otra estructura que contiene el tamaño de la memoria (será de 65 KiB), y un vector de bloques, cuyo tamaño dependerá de la capacidad de la memoria y del tamaño de los bloques.

Por último, se modeló los bloques como estructuras que contienen un vector de bytes, que serán los datos almacenados en el bloque; un tag, que permitirá identificar a los bloques; y un byte de validez.

Para identificar los bloques y manejar sus índices se trabajó mucho con las direcciones en memoria. A continuación se muestra un diagrama de cómo se segmenta una dirección de 16 bits para identificar el bloque y conjunto correspondiente. La dificultad aparece porque las dimensiones de la caché son variables, así que cambia la segmentación. Para desplazar la dirección una cierta cantidad de bits, en lugar de usar los logaritmos se puede dividir o multiplicar por las variables. Así es como se implementó el manejo de direcciones.

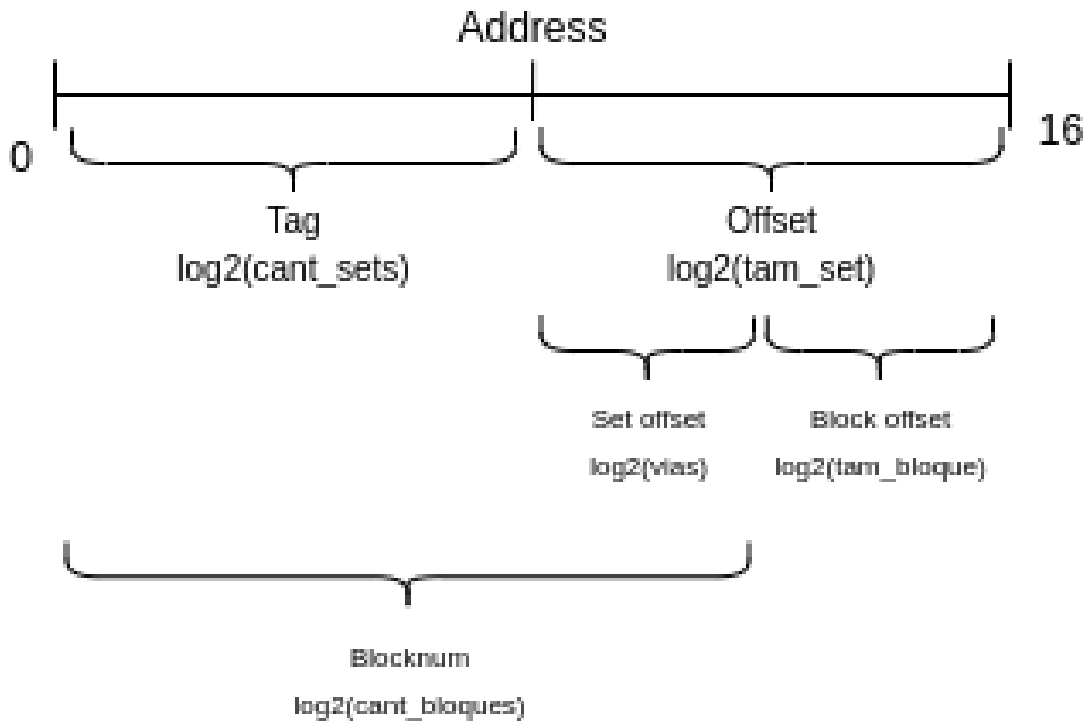

Figura 1: Segmentación de las direcciones de memoria.

2. Comportamiento deseado

El objetivo fue modelar una memoria caché que se comunica con una memoria principal. Se implementaron una función para leer y otra para escribir un byte. Estas funciones buscan en la caché si está el dato, y devuelven un byte que avisa si el acceso fue un miss o un hit. Además, se

lleva un conteo de cada acción, por lo que se puede saber también la tasa de miss final. Estas dos funciones no acceden directamente a la memoria, sino que delegan ese comportamiento en otras funciones que manejan el caché.

La memoria permite setear el tamaño de la caché, la cantidad de vías y el tamaño de los bloques.

El programa se maneja a partir de un archivo inicial de instrucciones, y se devuelve un archivo de log que muestra el resultado de todas las operaciones realizadas.

3. Ejemplos de ejecución

Se hicieron corridas de prueba con cinco archivos con instrucciones distintas ("prueba1.mem", "prueba2.mem", "prueba3.mem", "prueba4.mem", "prueba5.mem"), para dos configuraciones de cachés: 4 KB, 4WSA, 32bytes y 16KB, una vıa, 128 byte . A continuación, se muestra el contenido del log tras haberlos ejecutado:

3.1. [4 KB, 4WSA, 32bytes]

- prueba 1:

```
1 Write value 255 in address 0: hit = 0
2 Read value 255 from address 0: hit = 0
3 Write value 254 in address 16384: hit = 0
4 Read value 0 from address 16384: hit = 0
5 Write value 248 in address 32768: hit = 0
6 Read value 0 from address 32768: hit = 0
7 Write value 96 in address 49152: hit = 0
8 Read value 96 from address 49152: hit = 0
9 Read value 255 from address 0: hit = 1
10 Write value 1 in address 0: hit = 1
11 Read value 96 from address 16384: hit = 0
12 Write value 163 in address 16384: hit = 0
13 Read value 96 from address 32768: hit = 0
14 Write value 32 in address 32768: hit = 0
15 Read value 96 from address 49152: hit = 0
16 Write value 49 in address 49152: hit = 0
17 Miss rate: 87%.
```

pruebas_1/cache_log_prueba1.mem

- prueba 2:

```
1 Write value 123 in address 0: hit = 0
2 Write value 233 in address 1: hit = 0
3 Write value 34 in address 2: hit = 0
4 Read value 123 from address 0: hit = 0
5 Read value 233 from address 1: hit = 1
6 Read value 34 from address 2: hit = 1
7 Write value 1 in address 0: hit = 1
8 Write value 2 in address 1: hit = 1
9 Write value 3 in address 2: hit = 1
10 Read value 1 from address 0: hit = 1
11 Read value 2 from address 1: hit = 1
12 Read value 3 from address 2: hit = 1
13 Miss rate: 33%.
```

pruebas_1/cache_log_prueba2.mem

- prueba 3:

```
1 Write value 1 in address 128: hit = 0
2 Write value 2 in address 129: hit = 0
3 Write value 3 in address 130: hit = 0
4 Write value 4 in address 131: hit = 0
5 Read value 1 from address 128: hit = 0
6 Read value 2 from address 129: hit = 1
7 Read value 3 from address 130: hit = 1
8 Read value 4 from address 131: hit = 1
9 Read value 0 from address 1152: hit = 0
10 Read value 0 from address 2176: hit = 0
11 Read value 0 from address 3200: hit = 0
12 Read value 0 from address 4224: hit = 0
13 Read value 1 from address 128: hit = 0
14 Read value 2 from address 129: hit = 1
15 Read value 3 from address 130: hit = 1
16 Read value 4 from address 131: hit = 1
17 Miss rate: 62%.
```

pruebas_1/cache_log_prueba3.mem

- prueba 4:

```
1 Write value 0 in address 0: hit = 0
2 Write value 2 in address 1: hit = 0
3 Write value 3 in address 2: hit = 0
4 Write value 4 in address 3: hit = 0
5 Write value 5 in address 4: hit = 0
6 Read value 0 from address 0: hit = 0
7 Read value 2 from address 1: hit = 1
8 Read value 3 from address 2: hit = 1
9 Read value 4 from address 3: hit = 1
10 Read value 5 from address 4: hit = 1
11 Read value 0 from address 4096: hit = 0
12 Read value 0 from address 8192: hit = 0
13 Read value 0 from address 0: hit = 1
14 Read value 2 from address 1: hit = 1
15 Read value 3 from address 2: hit = 1
16 Read value 4 from address 3: hit = 1
17 Read value 5 from address 4: hit = 1
18 Miss rate: 47%.
```

pruebas_1/cache_log_prueba4.mem

- prueba 5:

```
1 SEGFAULT trying to Read address 131072.
2 Read value 0 from address 4096: hit = 0
3 Read value 0 from address 8192: hit = 0
4 Read value 0 from address 4096: hit = 1
5 Read value 0 from address 0: hit = 0
6 Read value 0 from address 4096: hit = 1
7 Miss rate: 60%.
```

pruebas_1/cache_log_prueba5.mem

3.2. [16KB, una vıa, 128 bytes

- prueba 1:

```
1 Write value 255 in address 0: hit = 0
2 Read value 255 from address 0: hit = 0
3 Write value 254 in address 16384: hit = 0
4 Read value 254 from address 16384: hit = 0
5 Write value 248 in address 32768: hit = 0
6 Read value 248 from address 32768: hit = 0
7 Write value 96 in address 49152: hit = 0
8 Read value 96 from address 49152: hit = 0
9 Read value 255 from address 0: hit = 0
10 Write value 1 in address 0: hit = 1
11 Read value 254 from address 16384: hit = 0
12 Write value 163 in address 16384: hit = 0
13 Read value 248 from address 32768: hit = 0
14 Write value 32 in address 32768: hit = 0
15 Read value 96 from address 49152: hit = 0
16 Write value 49 in address 49152: hit = 0
17 Miss rate: 93%.
```

prueba_2/cache_log_prueba1_2.mem

- prueba 2:

```
1 Write value 123 in address 0: hit = 0
2 Write value 233 in address 1: hit = 0
3 Write value 34 in address 2: hit = 0
4 Read value 123 from address 0: hit = 0
5 Read value 233 from address 1: hit = 1
6 Read value 34 from address 2: hit = 1
7 Write value 1 in address 0: hit = 1
8 Write value 2 in address 1: hit = 1
9 Write value 3 in address 2: hit = 1
10 Read value 1 from address 0: hit = 1
11 Read value 2 from address 1: hit = 1
12 Read value 3 from address 2: hit = 1
13 Miss rate: 33%.
```

prueba_2/cache_log_prueba2_2.mem

- prueba 3:

```
1 Write value 1 in address 128: hit = 0
2 Write value 2 in address 129: hit = 0
3 Write value 3 in address 130: hit = 0
4 Write value 4 in address 131: hit = 0
5 Read value 1 from address 128: hit = 0
6 Read value 2 from address 129: hit = 1
7 Read value 3 from address 130: hit = 1
8 Read value 4 from address 131: hit = 1
9 Read value 0 from address 1152: hit = 0
10 Read value 0 from address 2176: hit = 0
11 Read value 0 from address 3200: hit = 0
12 Read value 0 from address 4224: hit = 0
13 Read value 1 from address 128: hit = 1
```

```
14 Read value 2 from address 129: hit = 1
15 Read value 3 from address 130: hit = 1
16 Read value 4 from address 131: hit = 1
17 Miss rate: 56%.
```

prueba_2/cache_log_prueba3_2.mem

- prueba 4:

```
1 Write value 0 in address 0: hit = 0
2 Write value 2 in address 1: hit = 0
3 Write value 3 in address 2: hit = 0
4 Write value 4 in address 3: hit = 0
5 Write value 5 in address 4: hit = 0
6 Read value 0 from address 0: hit = 0
7 Read value 2 from address 1: hit = 1
8 Read value 3 from address 2: hit = 1
9 Read value 4 from address 3: hit = 1
10 Read value 5 from address 4: hit = 1
11 Read value 0 from address 4096: hit = 0
12 Read value 0 from address 8192: hit = 0
13 Read value 0 from address 0: hit = 1
14 Read value 2 from address 1: hit = 1
15 Read value 3 from address 2: hit = 1
16 Read value 4 from address 3: hit = 1
17 Read value 5 from address 4: hit = 1
18 Miss rate: 47%.
```

prueba_2/cache_log_prueba4_2.mem

- prueba 5:

```
1 SEGFAULT trying to Read address 131072.
2 Read value 0 from address 4096: hit = 0
3 Read value 0 from address 8192: hit = 0
4 Read value 0 from address 4096: hit = 1
5 Read value 0 from address 0: hit = 0
6 Read value 0 from address 4096: hit = 1
7 Miss rate: 60%.
```

prueba_2/cache_log_prueba5_2.mem

4. Conclusiones

La modelización e implementación de una memoria caché permitió entrar en detalle sobre el funcionamiento de una memoria asociativa por conjuntos, entendiendo como es internamente la lógica del funcionamiento y cómo simularlo.

5. Apéndice

Código fuente.

5.1. tp2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <ctype.h>
5 #include <getopt.h>
6 #include <string.h>
7 #include "cache.h"
8
9 const char* VERSION_ACTUAL = "1.0";
10
11 void imprimir_ayuda();
12 void imprimir_version();
13 void leer_archivo(char* nombre);
14 void ejecutar_instrucciones(char* archivo_instrucciones, char*
      archivo_salida);
15 char parse_arguments(int argc, char** argv, int* ways, int*
      cache_size, int* block_size, char* archivo_salida, char*
      archivo_instrucciones);
16
17
18 int main(int argc, char* argv[]) {
19     int ways = 4, cache_size = 4096, block_size = 256;
20     char* archivo_salida = "cache_log.mem", *archivo_instrucciones;
21
22     if (parse_arguments(argc, argv, &ways, &cache_size, &block_size,
          archivo_salida, archivo_instrucciones) != 0) {
23         return 1;
24     }
25
26     init_memoria(block_size);
27     init_cache(ways, cache_size);
28     ejecutar_instrucciones(archivo_instrucciones, archivo_salida);
29     delete_memoria();
30     delete_cache();
31     return 0;
32 }
33
34
35 char parse_arguments(int argc, char** argv, int* ways, int*
      cache_size, int* block_size, char* archivo_salida, char*
      archivo_instrucciones) {
36     int c;
37     static struct option long_options[] = {
38         {"help",      no_argument,       0,  'h' },
39         {"version",   no_argument,       0,  'V' },
40         {"output",    required_argument, 0,  'o' },
41         {"ways",      required_argument, 0,  'w' },
42         {"cachesize", required_argument, 0,  'c' },
43         {"blocksize", required_argument, 0,  'b' },
44         {"cs", required_argument, 0,  'c' },
45         {"bs", required_argument, 0,  'b' },
46         {0,           0,                 0,  0   }
```

```
47      };
48
49      while ((c = getopt_long_only(argc, argv, "hVo:w:c:b:?",
            long_options, NULL)) != -1)
50          switch (c) {
51              case 'h':
52                  imprimir_ayuda();
53                  return 1;
54              case 'V':
55                  imprimir_version();
56                  return 1;
57              case 'o':
58                  strcpy(archivo_salida, optarg);
59                  break;
60              case 'w':
61                  *ways = atoi(optarg);
62                  break;
63              case 'c':
64                  *cache_size = atoi(optarg);
65                  break;
66              case 'b':
67                  *block_size = atoi(optarg);
68                  break;
69              case '?':
70                  return 1;
71              default:
72                  fprintf (stderr, "opcion default.");
73                  return 1;
74      }
75      int index = optind;
76      if (argc - index != 1) {
77          fprintf (stderr, "Error: se debe ingresar el nombre de un
                archivo con instrucciones. Ingrese -h para ver la ayuda.");
78          return 1;
79      }
80      strcpy(archivo_instrucciones, argv[index++]);
81      return 0;
82  }
83
84  void ejecutar_instrucciones(char* archivo_instrucciones, char*
        archivo_salida) {
85      FILE* instrucciones = fopen(archivo_instrucciones, "r");
86      if (!instrucciones){
87          fprintf(stderr, "Error al abrir el archivo de
                instrucciones.");
88          return;
89      }
90      FILE* salida = fopen(archivo_salida, "w");
91      if (!salida){
92          fprintf(stderr, "Error al abrir el archivo de resultados.");
93          return;
94      }
95
96      char comando;
```

```
97      while(!feof(instrucciones)) {
98          fscanf(instrucciones, "%c", &comando);
99          switch (comando) {
100             case 'W' : {
101                 int address;
102                 char value;
103                 fscanf(instrucciones, "%i, %hhu\n", &address, &value);
104                 char hit = write_byte(address, value);
105                 if (hit == -1)
106                     fprintf(salida, "SEGFAULT trying to Write address
                            %i.\n", address);
107                 else
108                     fprintf(salida, "Write value %hhu in address %i:
                            hit = %hhu\n", value, address, hit);
109                 break;
110             }
111             case 'R': {
112                 int address;
113                 fscanf(instrucciones, "%i\n", &address);
114                 char hit;
115                 char value = read_byte(address, &hit);
116                 if (hit == -1)
117                     fprintf(salida, "SEGFAULT trying to Read address
                            %i.\n", address);
118                 else
119                     fprintf(salida, "Read value %hhu from address %i:
                            hit = %hhu\n", value, address, hit);
120                 break;
121             }
122             case 'M':
123                 fscanf(instrucciones, "R\n");
124                 fprintf(salida, "Miss rate: %hhu%%\n",
                        get_miss_rate());
125                 break;
126             case 'i':
127                 fscanf(instrucciones, "nit\n");
128                 init();
129                 fprintf(salida, "init\n");
130                 break;
131             default:
132                 return;
133         }
134     }
135
136     fclose(instrucciones);
137     fclose(salida);
138 }
139
140 void imprimir_ayuda(){
141     printf(
142     "    Uso:\n"
143     "        tp2 -h\n"
144     "        tp2 -V\n"
145     "        tp2 options archivo\n"
```

```
146     "\n Opciones:\n"
147     "        -h  Imprime este mensaje.\n"
148     "        -V  Da la versi n del programa.\n"
149     "        -s  Imprime por consola la salida del programa, en lugar
            de guardarlo en el archivo de salida.\n"
150     "        -o  Prefijo de los archivos de salida.\n"
151     "        -w  Cantidad de v   as.\n"
152     "        -cs Tama o del cach  en kilobytes.\n"
153     "        -bs Tama o de bloque en bytes.\n"
154     "\n Ejemplos:\n"
155     "   -> tp2 -w 4 -cs 8 -bs 16 prueba1.mem\n");
156 }
157
158 void imprimir_version(){
159     printf("Versi n actual: %s\n", VERSION_ACTUAL);
160 }
```

src/tp2.c

5.2. cache.h

```
1 #ifndef __CACHE_H__
2 #define __CACHE_H__
3
4
5 typedef struct bloque {
6     char* datos;
7     char valido;
8     char tag;
9 } bloque_t;
10
11
12 typedef struct cache {
13     unsigned short int vias;
14     unsigned short int capacidad;
15
16     bloque_t* bloques;
17     unsigned short int* topes;
18
19     unsigned short int accesos;
20     unsigned short int misses;
21 } cache_t;
22
23
24 typedef struct memoria_16_bits {
25     bloque_t* bloques;
26     unsigned short int capacidad;
27     unsigned short int tam_bloque;
28 } memoria_t;
29
30 //inicializar la memoria principal con capacidad ed 65 KiB, con su
       tama o de bloque, y reservar el espacio de memoria para los
       bloques
31 void init_memoria(short unsigned int tam_bloque);
32
```

```
33 //inicializar la cach  con su cantidad de vias y capacidad, los
       bloques como inv lidos, los topes, misses y accesos en cero, y
       reservar memoria para los bloques.
34 void init_cache(short unsigned int vias, short unsigned int
       capacidad);
35
36 // inicializar los bloques de la cach  como inv lidos, la memoria
       simulada en 0 y la tasa de misses a 0
37 void init();
38
39 // liberar la memoria reservada por la memoria
40 void delete_memoria();
41
42 // liberar la memoria reservada por el cache
43 void delete_cache();
44
45 //devolver el conjunto de cach  al que mapea la direcci n address.
46 unsigned int find_set(int address);
47
48 //devolver el bloque m s     antiguo     dentro de un conjunto,
       utilizando el campo correspondiente de los metadatos de los
       bloques del conjunto.
49 unsigned int find_earliest(int setnum);
50
51 //leer el bloque blocknum de memoria y guardarlo en el lugar que le
       corresponda en la memoria cach
52 void read_block(int blocknum);
53
54 // busca un bloque en la cache. Se fija en el conjunto
       correspondiente si algun bloque coincide con el tag.
55 // devuelve el indice del bloque en el array de bloques (la posicion
       en cache) si es un hit, si no devuelve basura.
56 unsigned short int find_block(int setnum, char* hit);
57
58 //retornar el valor correspondiente a la posici n de memoria
       address, busc ndolo primero en el cach
59 //escribir 1 en *hit si es un hit y 0 si es un miss
60 char read_byte(int address, char *hit);
61
62 // escribir el valor value en la posici n correcta del bloque que
       corresponde a address, si est  en el cach , y en la memoria en
       todos los casos, y
63 //devolver 1 si es un hit y 0 si es un miss.
64 char write_byte(int address, char value);
65
66 //devolver el porcentaje de misses desde que se inicializ  el cache
67 char get_miss_rate();
68
69 #endif /* __CACHE_H__ */
```

src/cache.h

5.3. cache.c

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

```
2 #include <stdlib.h>
3
4 cache_t cache;
5 memoria_t memoria;
6
7 bloque_t new_bloque(int tamanio) {
8     bloque_t bloque;
9     bloque.valido = 0;
10    bloque.tag = 0;
11    bloque.datos = malloc(tamanio * sizeof(char));
12    return bloque;
13 }
14
15 void init_memoria(short unsigned int tam_bloque) {
16    memoria.capacidad = 0xffff;
17    memoria.tam_bloque = tam_bloque;
18
19    unsigned short int cant_bloques_memoria = memoria.capacidad /
          memoria.tam_bloque;
20    memoria.bloques = malloc(cant_bloques_memoria * sizeof(bloque_t));
21
22    for (int i = 0; i < cant_bloques_memoria; i++) {
23        memoria.bloques[i] = new_bloque(memoria.tam_bloque);
24    }
25 }
26
27 void init_cache(short unsigned int vias, short unsigned int
      capacidad) {
28    cache.vias = vias;
29    cache.capacidad = capacidad;
30    cache.misses = 0;
31    cache.accesos = 0;
32
33    unsigned short int cant_bloques = cache.capacidad /
          memoria.tam_bloque;
34    cache.bloques = malloc(cant_bloques * sizeof(bloque_t));
35    for (int i = 0; i < cant_bloques; i++) {
36        cache.bloques[i] = new_bloque(memoria.tam_bloque);
37    }
38
39    unsigned short int cant_sets = capacidad / (memoria.tam_bloque *
          vias);
40    cache.topes = malloc(cant_sets * sizeof(short int));
41    for (int i = 0; i < cant_sets; i++) {
42        cache.topes[i] = 0;
43    }
44 }
45
46 void init() {
47    unsigned short int cant_bloques_cache = cache.capacidad /
          memoria.tam_bloque;
48    for (int i = 0; i < cant_bloques_cache; i++) {
49        cache.bloques[i].valido = 0;
50    }
```

```
51      cache.misses = 0;
52      cache.accesos = 0;
53
54      unsigned short int cant_bloques_memoria = memoria.capacidad /
            memoria.tam_bloque;
55      for (int i = 0; i < cant_bloques_memoria; i++) {
56          memoria.bloques[i].valido = 0;
57          for (int j = 0; j < memoria.tam_bloque; j++) {
58              memoria.bloques[i].datos[j] = 0;
59          }
60      }
61  }
62
63  void delete_memoria() {
64      unsigned short int cant_bloques_memoria = memoria.capacidad /
            memoria.tam_bloque;
65      for (int i = 0; i < cant_bloques_memoria; i++) {
66          free(memoria.bloques[i].datos);
67      }
68      free(memoria.bloques);
69  }
70
71  void delete_cache() {
72      free(cache.bloques);
73      free(cache.topes);
74  }
75
76  unsigned int find_set(int address) {
77      unsigned short int cant_sets = cache.capacidad /
            (memoria.tam_bloque * cache.vias);
78      return (address/memoria.tam_bloque) % cant_sets;
79  }
80
81  unsigned int find_earliest(int setnum) {
82      return setnum*cache.vias + cache.topes[setnum];
83  }
84
85  void read_block(int blocknum) {
86      unsigned short int setnum = find_set(blocknum *
            memoria.tam_bloque);
87      unsigned short int pos = find_earliest(setnum);
88
89      cache.bloques[pos] = memoria.bloques[blocknum];
90      cache.bloques[pos].valido = 1;
91      cache.bloques[pos].tag = blocknum / cache.vias;
92
93      if (cache.topes[setnum] >= cache.vias - 1)
94          cache.topes[setnum] = 0;
95      else
96          cache.topes[setnum]++;
97  }
98
99  void write_byte_tomem(int address, char value) {
100     unsigned short int block_offset = address % memoria.tam_bloque;
```

```
101     memoria.bloques[address / memoria.tam_bloque].datos[block_offset]
            = value;
102 }
103
104 unsigned short int find_block(int address, char* hit){
105     unsigned short int tag = address /
            (cache.vias*memoria.tam_bloque);
106     unsigned short int setnum = find_set(address);
107     unsigned short int i = 0, pos;
108     bloque_t bloque;
109     *hit = 0;
110
111     while (!(*hit) && i < cache.vias) {
112         pos = setnum*cache.vias + i;
113         bloque = cache.bloques[pos];
114         if (bloque.tag == tag && bloque.valido) {
115             *hit = 1;
116         }
117         i++;
118     }
119     return pos;
120 }
121
122 char read_byte(int address, char *hit) {
123     if (address >= memoria.capacidad || address < 0) { // si no
            existe la direccion en memoria, SEGFAULT
124         *hit = -1;
125         return 0;
126     }
127
128     unsigned short int pos = find_block(address, hit);
129     if (*hit == 0) {
130         read_block(address / memoria.tam_bloque);
131         pos = find_block(address, hit);
132         *hit = 0;
133         cache.misses++;
134     }
135     cache.accesos++;
136     unsigned int block_offset = address % memoria.tam_bloque;
137     return cache.bloques[pos].datos[block_offset];
138 }
139
140 char write_byte(int address, char value){
141     if (address >= memoria.capacidad || address < 0) // si no existe
            la direccion en memoria, SEGFAULT
142         return -1;
143
144     char hit;
145     unsigned short int pos = find_block(address, &hit);
146     cache.accesos++;
147
148     if (hit == 1) {
149         unsigned int block_offset = address % memoria.tam_bloque;
150         cache.bloques[pos].datos[block_offset] = value;
```

```
151     } else
152         cache.misses++;
153
154     write_byte_tomem(address, value);
155     return hit;
156 }
157
158 char get_miss_rate() {
159     return (cache.misses*100 / cache.accesos);
160 }
```

src/cache.c