Trabajo Práctico 2
Memorias cache

Marco Rodrigo Albanesi, *Padrón Nro. 86.063*
fiuba@mrod.com.ar

1er. Cuatrimestre del 2020
66.20 Organización de Computadoras — Práctica: Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

26 de junio de 2020

Índice

1. Enunciado

66:20 Organización de Computadoras
Trabajo práctico 2: Memorias caché

1. Objetivos

Familiarizarse con el funcionamiento de la memoria caché implementando una simulación de una caché dada.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección 8), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido. Por este motivo, el día de la devolución deben concurrir todos los integrantes del grupo.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo, y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

4. Recursos

Este trabajo práctico debe ser implementado en C[2], y correr al menos en Linux.

5. Introducción

La memoria a simular es una caché[1] asociativa por conjuntos de cuatro vías, de 4KB de capacidad, bloques de 128 bytes, política de reemplazo LRU y política de escritura WT/¬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. Estas memorias pueden ser implementadas como variables globales. Cada bloque de la memoria caché deberá contar con su metadata, incluyendo el tag, el bit `V` y la información necesaria para implementar la política de reemplazo LRU.

6. Programa

Se deben implementar las siguientes primitivas:

```
void init()
unsigned int get_offset (unsigned int address)
unsigned int find_set(unsigned int address)
unsigned int select_oldest(unsigned int setnum)
int compare_tag (unsigned int tag, unsigned int set)
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)
void write_tocache(unsigned int address, unsigned char value)
unsigned char read_byte(unsigned int address)
void write_byte(unsigned int address, unsigned char value)
float get_miss_rate()
```

- La función `init()` debe inicializar la memoria principal simulada en 0, los bloques de la caché como inválidos y la tasa de misses a 0.
- La función `get_offset(unsigned int address)` debe devolver el *offset* del byte del bloque de memoria al que mapea la dirección `address`.
- La función `find_set(unsigned int address)` debe devolver el conjunto de caché al que mapea la dirección `address`.
- La función `select_oldest()` debe devolver la vía en la que está el bloque más "viejo" dentro de un conjunto, utilizando el campo correspondiente de los metadatos de los bloques del conjunto.
- La función `compare_tag(unsigned int tag, unsigned int set)` debe devolver la vía en la que se encuentra almacenado el bloque correspondiente a `tag` en el conjunto `index`, o -1 si ninguno de los *tags* coincide.
- La función `read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)` debe leer el bloque `blocknum` de memoria y guardarlo en el conjunto y vía indicados en la memoria caché.
- La función `read_byte(unsigned int address)` debe buscar el valor del byte correspondiente a la posición `address` en la caché; si éste no se encuentra en la caché debe cargar ese bloque. El valor de retorno siempre debe ser el valor del byte almacenado en la dirección indicada.

- La función `write_byte(unsigned int address, unsigned char value)` debe escribir 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.
- La función `get_miss_rate()` debe devolver el porcentaje de misses desde que se inicializó la caché.

Con estas primitivas, hacer un programa que llame a `init()` y luego lea de un archivo una serie de comandos y los ejecute. Los comandos tendrán la siguiente forma:

```
FLUSH
R ddddd
W ddddd, vvv
MR
```

- El comando "`FLUSH`" se ejecuta llamando a la función `init()`. Representa el vaciado del caché.
- Los comandos de la forma "`R ddddd`" se ejecutan llamando a la función `read_byte(ddddd)` e imprimiendo el resultado.
- Los comandos de la forma "`W ddddd, vvv`" se ejecutan llamando a la función `write_byte(unsigned int ddddd, char vvv)` e imprimiendo el resultado.
- El comando "`MR`" se ejecuta llamando a la función `get_miss_rate()` e imprimiendo el resultado.

El programa deberá chequear que las líneas del archivo correspondan a un comando con argumentos dentro del rango estipulado, o de lo contrario estar vacías. En caso de que una línea tenga otra cosa que espacios blancos y no tenga un comando válido, se deberá imprimir un mensaje de error informativo.

7. Mediciones

Se deberá incluir la salida que produzca el programa con los siguientes archivos de prueba:

- prueba1.mem
- prueba2.mem
- prueba3.mem

- prueba4.mem
- prueba5.mem
- prueba6.mem

7.1. Documentación

Es necesario que el informe incluya una descripción detallada de las técnicas y procesos de medición empleados, y de todos los pasos involucrados en el mismo, ya que forman parte de los objetivos principales del trabajo.

8. Informe

El informe deberá incluir:

- Este enunciado;
- Documentación relevante al diseño e implementación del programa, incluyendo las estructuras de datos;
- Instrucciones de compilación;
- Resultados de las corridas de prueba;
- El código fuente completo del programa, en formato digital.

9. Fecha de entrega

La fecha de entrega es el jueves 25 de junio de 2020.

Referencias

[1] Hennessy, John L. and Patterson, David A., Computer Architecture: A Quantitative Approach, Third Edition, 2002.

[2] Kernighan, Brian, and Ritchie, Dennis, The C Programming Language.

2. Desarrollo

Se realizó un programa C con un módulo llamado cache donde se implementan las primitivas propuestas por el enunciado y main que es responsable de interpretar y validar el archivo de entrada e invocar a las funciones de cache.

```
#define MEM_SIZE 65536
#define CACHE_SIZE 4096
#define LINE_SIZE 128
#define WAYS 4
```

Los parámetros de la memoria principal y caché están definidos en un solo lugar. Modificando estos valores se puede simular distintos tipos de memoria cache. En particular si WAYS es 1 se simula una direct mapped, y si WAYS es CACHE SIZE/LINE SIZE queda directamente mapeada.

```
struct Line {
   bool valid;
   unsigned int tag;
   unsigned int last;
   unsigned char data[LINE_SIZE];
};
```

Cada línea del caché está representada por este struct. Además del tag, bit de validez y el bloque de memoria tiene un contador utilizado por la política de remplazo.

```
struct Set {
   unsigned int last;
   struct Line ways[WAYS];
};
```

Los conjuntos son simplemente arreglos de líneas mas un campo para la política de remplazo.

```
struct Set sets[CACHE_SIZE / (LINE_SIZE * WAYS)];
```

El cache es entonces un arreglo de Set. Para los parámetros dados son 8 conjuntos.

El módulo emplea otras variables globales: Contadores de accesos a memoria y misses, máscaras de bits para aplicar a las direcciones de memoria y cantidades de bits para hacer corrimientos.

```
offsetMask = LINE_SIZE - 1;
address & offsetMask;
```

Para determinar el offset simplemente se usa como máscara LINE SIZE-1. Al aplicar esta máscara a la dirección se obtienen solo los bits necesarios para direccionar un byte dentro del bloque de memoria caché.

```
numOfSets = CACHE_SIZE / (LINE_SIZE * WAYS);
setMask = numOfSets * LINE_SIZE - 1;
offsetBits = log2int(LINE_SIZE);

(address & setMask) >> offsetBits
```

El número de conjunto se obtiene de manera similar. La máscara de bits es la cantidad de sets multiplicado por el tamaño de la línea -1. Luego se hace un corrimiento a derecha por la cantidad de bits necesarios para el offset. Para los parámetros dados, son los bits 0 a 10 y un corrimiento de 7. Lo que deja los tres bits para direccionar a los 8 conjuntos.

```
setBits = log2int(numOfSets);
address >> (offsetBits + setBits);
```

El tag es la dirección de memoria con un corrimiento a derecha de la cantidad de bits que ocupan el offset y el número de conjunto.

```
unsigned int oldest = set->last + 1;
int waynum = -1;
for (int i = 0; i < WAYS; i++) {
    struct Line* line = &set->ways[i];
    if ((waynum < 0 && ! line->valid) || line->last < oldest) {
        waynum = i;
        oldest = line->last;
```

```
    }
}
```

Para obtener el número de vía donde poder escribir un bloque de memoria que debe copiarse a la caché se busca la linea con menor valor en el campo last. Este campo se copia desde su homónimo del conjunto cada vez que se lee o escribe en la línea justo después de incrementar al de conjunto. La de menor valor es la menos usada recientemente. En caso que una vía no haya sido inicializada nunca entonces el valor del campo last es 0 y es seleccionada por sobre otra vía activa.

La inicialización de variables, separación de campos y la selección de vía para la política de remplazo es lo más relevante de la implementación. El resto de las primitivas se resuelven directamente una vez definido esto.

3. Compilación del programa

Para compilar el programa sencillamente tenemos utilizar su makefile desde el directorio donde se descomprimió el entregable.

```
$ make
cc src/main.c src/cache.c -o cache -lm -O0 -Wall -Werror -pedantic -pedantic-errors
```

4. Pruebas

4.1. prueba1.mem

```
W 0, 255
W 1024, 254
W 2048, 248
W 4096, 096
W 8192, 192
R 0
R 1024
R 2048
R 8192
MR
```

- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Lee de conjunto 0. Miss. Copia el bloque.
- Lee de conjunto 0. Miss. Copia el bloque.
- Lee de conjunto 0. Miss. Copia el bloque.
- Lee de conjunto 0. Miss. Copia el bloque.
- Muestra el miss rate: %100

```
$ ./cache prueba1.mem
Byte at 0 set to 255
Byte at 1024 set to 254
Byte at 2048 set to 248
Byte at 4096 set to 96
Byte at 8192 set to 192
Byte at 0 is 255
Byte at 1024 is 254
```

```
Byte at 2048 is 248
Byte at 8192 is 192
Miss rate: %100.000000
```

4.2. prueba2.mem

```
R 0
R 127
W 128, 10
R 128
W 128, 20
R 128
MR
```

- Lee de conjunto 0. Miss. Copia el bloque.
- Lee de conjunto 0. Hit.
- Escribe en conjunto 1. Miss.
- Lee de conjunto 1. Miss. Copia el bloque.
- Escribe en conjunto 1. Hit.
- Escribe en conjunto 1. Hit.
- Muestra el miss rate: %50

```
$ ./cache prueba2.mem
Byte at 0 is 0
Byte at 127 is 0
Byte at 128 set to 10
Byte at 128 is 10
Byte at 128 set to 20
Byte at 128 is 20
Miss rate: %50.000000
```

4.3. prueba3.mem

```
W 128, 1
W 129, 2
W 130, 3
W 131, 4
R 1152
R 2176
R 3200
R 4224
R 128
R 129
R 130
R 131
MR
```

- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.

- Lee de conjunto 1. Miss. Copia el bloque a vía 0 con tag 1.
- Lee de conjunto 1. Miss. Copia el bloque a vía 1 con tag 2.
- Lee de conjunto 1. Miss. Copia el bloque a vía 2 con tag 3.
- Lee de conjunto 1. Miss. Copia el bloque a vía 3 con tag 4.
- Lee de conjunto 0. Miss. Copia el bloque a vía 0 con tag 0.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Muestra el miss rate: %75

```
$ ./cache prueba3.mem
Byte at 128 set to 1
Byte at 129 set to 2
Byte at 130 set to 3
Byte at 131 set to 4
Byte at 1152 is 0
Byte at 2176 is 0
Byte at 3200 is 0
Byte at 4224 is 0
Byte at 128 is 1
Byte at 129 is 2
Byte at 130 is 3
Byte at 131 is 4
Miss rate: %75.000000
```

4.4. prueba4.mem

```
W 0, 255
W 1, 2
W 2, 3
W 3, 4
W 4, 5
R 0
R 1
R 2
R 3
R 4
R 4096
R 8192
R 2048
R 1024
R 0
R 1
R 2
R 3
R 4
MR
```

- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.
- Escribe en conjunto 0. Miss.

- Escribe en conjunto 0. Miss.
- Lee de conjunto 0. Miss. Copia el bloque a vía 0 con tag 0.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Miss. Copia el bloque a vía 1 con tag 4.
- Lee de conjunto 0. Miss. Copia el bloque a vía 2 con tag 8.
- Lee de conjunto 0. Miss. Copia el bloque a vía 3 con tag 2.
- Lee de conjunto 0. Miss. Copia el bloque a vía 0 con tag 1.
- Lee de conjunto 0. Miss. Copia el bloque a vía 1 con tag 0.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Muestra el miss rate: 100*11/19 = %57.89

```
$ ./cache prueba4.mem
Byte at 0 set to 255
Byte at 1 set to 2
Byte at 2 set to 3
Byte at 3 set to 4
Byte at 4 set to 5
Byte at 0 is 255
Byte at 1 is 2
Byte at 2 is 3
Byte at 3 is 4
Byte at 4 is 5
Byte at 4096 is 0
Byte at 8192 is 0
Byte at 2048 is 0
Byte at 1024 is 0
Byte at 0 is 255
Byte at 1 is 2
Byte at 2 is 3
Byte at 3 is 4
Byte at 4 is 5
Miss rate: %57.894737
```

4.5. prueba5.mem

```
R 131072
R 4096
R 8192
R 4096
R 0
R 4096
MR
```

La dirección de memoria está fuera del espacio direccionable por 16 bits.

```
$ ./cache prueba5.mem
Direccion de memoria 131072 fuera de rango
```

4.6. prueba6.mem

```
W 0, 1
W 1024, 2
W 2048, 3
W 4096, 4
W 8192, 256
R 0
R 1024
R 2048
R 4096
R 8192
MR
```

En el comando W 8192, 256 el valor a escribir no es representable por un byte.

```
$ ./cache prueba6.mem
Byte at 0 set to 1
Byte at 1024 set to 2
Byte at 2048 set to 3
Byte at 4096 set to 4
256 No es un byte válido
```

4.7. pruebaFLUSH.mem

```
W 0, 1
R 0
W 32, 100
R 32
MR
FLUSH
R 0
R 32
MR
```

- Escribe en conjunto 0. Miss.
- Lee de conjunto 0. Miss. Copia el bloque a via 0 con tag 0.
- Escribe en conjunto 0. Hit.
- Lee de conjunto 0. Hit.
- Muestra el miss rate: %50
- Invalida todo el cache
- Lee de conjunto 0. Miss. Copia el bloque a vía 0 con tag 0.
- Lee de conjunto 0. Hit.
- Muestra el miss rate: %50

```
$ ./cache pruebaFLUSH.mem
Byte at 0 set to 1
Byte at 0 is 1
Byte at 32 set to 100
```

```
Byte at 32 is 100
Miss rate: %50.000000
Byte at 0 is 1
Byte at 32 is 100
Miss rate: %50.000000
```

4.8. pruebaMR.mem

```
MR
WRONG
```

Imprime el miss rate sin accesos a memoria. Muestra 0 en vez de el resultado de 0 / 0. El segundo comando es incorrecto.

```
$ ./cache pruebaMR.mem
Miss rate: %0.000000
Comando invalido: WRONG
```

5. Conclusiones

La implementación de simulación de memoria cache sirvió para profundizar los conocimientos de cómo funciona este sistema de memoria y cómo los parámetros con lo cuál fue diseñado junto con el comportamiento del programa que se ejecuta afecta la tasa de aciertos y por ende su desempeño.

6. Apéndice: Código fuente

6.0.1. cache.h

```
#ifndef CACHE_H_
#define CACHE_H_

#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>

#define MEM_SIZE 65536
#define CACHE_SIZE 4096
#define LINE_SIZE 128
#define WAYS 4

struct Line {
   bool valid;
   unsigned int tag;
   unsigned int last;
   unsigned char data[LINE_SIZE];
};

struct Set {
   unsigned int last;
   struct Line ways[WAYS];
};

void init();

void flush();

unsigned int get_offset (unsigned int address);

unsigned int find_set(unsigned int address);

unsigned int select_oldest(unsigned int setnum);

int compare_tag (unsigned int tag, unsigned int set);

void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set);

void write_tocache(unsigned int address, unsigned char value);

unsigned char read_byte(unsigned int address);

void write_byte(unsigned int address, unsigned char value);

float get_miss_rate();

#endif /* CACHE_H_ */
```

6.0.2. cache.c

```
#include "cache.h"

//Memoria principal
unsigned char mem[MEM_SIZE];
```

```
//Conjuntos del cache. Para los valores dados son 8.
struct Set sets[CACHE_SIZE / (LINE_SIZE * WAYS)];

//Cantidad de accesos a memoria y misses
unsigned int accessed;
unsigned int missed;

//Mask para obtener el numero de set y el offset desde la dirección
unsigned int offsetMask;
unsigned int setMask;

//Cantidad de sets que tiene el cache
unsigned int numOfSets; //8

//Cantidad de bits para el offset y para obtener el numero de set.
unsigned int offsetBits; //7
unsigned int setBits; //3

//logaritmo base dos para obtener cantidad de bits
unsigned int log2int(unsigned int n) {
    int r = 0;
    while (n >>= 1) ++r;
    return r;
}

void init() {
  //Inicialidar la memoria en 0
    for (int i = 0; i < MEM_SIZE; i++) {
        mem[i] = 0;
    }

    numOfSets = CACHE_SIZE / (LINE_SIZE * WAYS);
    offsetBits = log2int(LINE_SIZE);
    setBits = log2int(numOfSets);

    offsetMask = LINE_SIZE - 1; //0000000001111111
    setMask = numOfSets * LINE_SIZE - 1; //0000001111111111

    flush();
}

//Resetea el bit valid en las vias y el campo last usado para la pilitica de remplazo
void flush() {
    for (int i = 0; i < numOfSets; i++) {
        sets[i].last = 0;
        for (int j = 0; j < WAYS; j++) {
            sets[i].ways[j].valid = false;
            sets[i].ways[j].last = 0;
        }
    }
}

unsigned int get_offset(unsigned int address) {
    return address & offsetMask;
}

unsigned int find_set(unsigned int address) {
    return (address & setMask) >> offsetBits;
}
```

```
unsigned int get_tag(unsigned int address) {
    return address >> (offsetBits + setBits);
}

//Devuelve el numero de vía más antiguo o el primero no valido
unsigned int select_oldest(unsigned int setnum) {
    struct Set* set = &sets[setnum];

    unsigned int oldest = set->last + 1;
    int waynum = -1;
    for (int i = 0; i < WAYS; i++) {
        struct Line* line = &set->ways[i];
        if ((waynum < 0 && ! line->valid) || line->last < oldest) {
            waynum = i;
            oldest = line->last;
        }
    }
    return waynum;
}

//Devuelve el numero de vía con mismo tag del parámetro para un conjunto
int compare_tag(unsigned int tag, unsigned int setnum) {
    struct Set* set = &sets[setnum];
    for (int i = 0; i < WAYS; i++) {
        if (set->ways[i].valid && set->ways[i].tag == tag)
            return i;
    }
    return -1;
}

//Copia un bloque de memoria a una linea de cache, seteando el tag y bit de validez
void read_tocache(unsigned int blocknum, unsigned int wayNum, unsigned int setnum) {
    struct Set* set = &sets[setnum];
    struct Line* line = &set->ways[wayNum];

    memcpy(&line->data, &mem[blocknum * LINE_SIZE],LINE_SIZE);

    line->tag = get_tag(blocknum * LINE_SIZE);
    line->valid = true;
}

//Escribe un byte al bloque de cache correspondiente según address
void write_tocache(unsigned int address, unsigned char value) {
    unsigned int setnum = find_set(address);
    unsigned int tag = get_tag(address);
    unsigned int offset = get_offset(address);
    int wayNum = compare_tag(tag, setnum);

    struct Set* set = &sets[setnum];
    struct Line* line = &set->ways[wayNum];

    line->last = ++set->last;
    line->data[offset] = value;
}

//Lee un byte desde el cache, copiando un bloque desde memoria si hace falta
unsigned char read_byte(unsigned int address) {
    accessed++;

    unsigned int setnum = find_set(address);
```

```
    unsigned int tag = get_tag(address);
    int wayNum = compare_tag(tag, setnum);

    struct Set* set = &sets[setnum];

    if (wayNum < 0) {
        missed++;
        wayNum = select_oldest(setnum);
        read_tocache(address / LINE_SIZE, wayNum, setnum);
    }

    set->ways[wayNum].last = ++set->last;

    int offset = get_offset(address);
    return set->ways[wayNum].data[offset];
}

//Escribe un byte a memoria y al cache si el bloque está presente
void write_byte(unsigned int address, unsigned char value) {
    accessed++;

    unsigned int setnum = find_set(address);
    unsigned int tag = get_tag(address);
    int wayNum = compare_tag(tag, setnum);

    if (wayNum < 0) {
        missed++;
    } else {
        write_tocache(address, value);
    }

    mem[address] = value;
}

//Devuelve el miss rate como porcentaje
float get_miss_rate() {
    return accessed > 0 ? (missed / (float) accessed) : 0.0;
}
```

6.0.3. main.c

```
#include "cache.h"
#include <stdio.h>

#define ERROR -1
#define OK 0

int abrirArchivo(FILE** file, const char *nombre, const char *modo) {
    *file = fopen(nombre, modo);
    if (*file == NULL) {
        fprintf(stderr, "Error␣abriendo␣el␣archivo␣'%s':␣", nombre);
        perror(NULL);
        return ERROR;
    }
    return OK;
}

int cerrarArchivo(FILE* file) {
    if (file == NULL)
        return ERROR;
```

```
    if (fclose(file) == ERROR) {
        perror("Error␣cerrando␣archivo");
        return ERROR;
    }
    return OK;
}

int procesarArchivo(FILE* archivo) {
    int status = OK;

    do {
        char command[7];
        unsigned int address;
        unsigned int value;

        int read = fscanf(archivo, "%6s␣%u,␣%u\n", command, &address, &value);

        if (read > 0) {
            if (strncmp(command, "FLUSH", 6) == 0) {
                flush();
            } else if (strncmp(command, "MR", 3) == 0) {
                printf("Miss␣rate:␣%%%f\n", 100 * get_miss_rate());
            } else if (strncmp(command, "R", 2) == 0 && read == 2) {
                if (address >= MEM_SIZE) {
                    fprintf(stderr, "Direccion␣de␣memoria␣%u␣fuera␣de␣rango\n", address);
                    status = ERROR;
                    break;
                }
                value = read_byte(address);
                printf("Byte␣at␣%u␣is␣%hhu\n", address, value);
            } else if (strncmp(command, "W", 2) == 0 && read == 3) {
                if (address >= MEM_SIZE) {
                    fprintf(stderr, "Direccion␣de␣memoria␣%u␣fuera␣de␣rango\n", address);
                    status = ERROR;
                    break;
                }

                if (value > 255) {
                    fprintf(stderr, "%u␣No␣es␣un␣byte␣válido\n", value);
                    status = ERROR;
                    break;
                }

                write_byte(address, value);
                printf("Byte␣at␣%u␣set␣to␣%hhu\n", address, value);
            } else {
                fprintf(stderr, "Comando␣invalido:␣%s", command);
                if (read > 1)
                    fprintf(stderr, "␣%u", address);
                if (read > 2)
                    fprintf(stderr, ",␣%hhu", value);
                fprintf(stderr, "\n");

                status = ERROR;
            }
        }
    } while (status != ERROR && ! feof(archivo));

    return status;
```

```
}

int main(int argc, char *argv[]) {
    int status = OK;
    FILE* archivo;

    if (argc == 2) {
        status = abrirArchivo(&archivo, argv[1], "r");
        if (status == OK) {
            init();
            procesarArchivo(archivo);
            cerrarArchivo(archivo);
        }
    } else {
        status = ERROR;
        fprintf(stderr, "Error␣en␣argumentos:␣Se␣debe␣proporcionar␣la␣ruta␣de␣UN␣archivo␣
    para␣procesar\n");
    }

    return status;
}
```

6.0.4. Makefile

```
CFLAGS = -O0 -Wall -Werror -pedantic -pedantic-errors

SRC := src
SOURCES := $(SRC)/main.c $(SRC)/cache.c

all: cache

cache:
  $(CC) $(SOURCES) -o cache $(CFLAGS)

clean:
  $(RM) -f cache
```