

**计算机系统结构实验报告**

|  |  |
| --- | --- |
| 姓 名： | 熊逸钦 |
| 学 院： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 班 级： | CS1701 |
| 学 号： | U201714501 |
| 指导教师： | 童薇、施展 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2020年4月18日

**目 录**

[1. Cache模拟器实验 3](#_Toc38130278)

[**1.1.** **实验目的** 3](#_Toc38130279)

[**1.2.** **实验环境** 3](#_Toc38130280)

[**1.3.** **实验思路** 3](#_Toc38130281)

[**1.4.** **实验结果和分析** 9](#_Toc38130282)

[2. 总结和体会 10](#_Toc38130283)

[3. 对实验课程的建议 11](#_Toc38130284)

# Cache模拟器实验

本次实验中需要通过编写一个200-300行的C程序来模拟Cache缓存的行为。

给出内存访问的轨迹，要求编写的Cache模拟器程序能够模拟缓存相对内存访问轨迹的命中/缺失行为。并最终输出该内存访问轨迹的命中、缺失和（缓存行）淘汰/驱逐的总数。

* 1. **实验目的**

（1）理解Cache的工作原理；

（2）掌握实现一个高效模拟器的方法。

* 1. **实验环境**

本次实验对环境的要求不高，只需要一个基础的C语言程序的编译运行环境以及64位的Linux操作系统即可。

实验中我的具体实验环境如下：

（1）操作系统：Ubuntu 18.04 LTS x64

（2）开发语言：C

（3）编译器：gcc 7.4.0

（4）编辑器：Visual Studio Code

* 1. **实验思路**

（1）分析需求

程序的输入是内存访问轨迹，以测试用例中的yi.trace为例，其访存轨迹如下：

L 10,1

M 20,1

L 22,1

S 18,1

L 110,1

L 210,1

M 12,1

本次实验只考虑数据Cache的性能，对指令Cache不进行考虑，因此不考虑行首字符为I的指令访存轨迹。对于上面的例子，其中每一行的第一个字符为空格，第二个字符的取值集合为{L, S, M}，代表操作类型operation。其中L表示Load读取内存，S表示Store写入内存，M表示Modify修改内存（可以看作是对于同一个地址先Load再Store）。后面以逗号隔开的两个数字分别是访存地址address（十六进制）、访问的内存字节数量size（十进制）。由于本次实验假设内存访问的地址总是正确对齐的，即一次内存访问从不跨越块的边界，因此可忽略访问轨迹中给出的访问请求大小。

实验规定程序执行时的命令行格式如下：

csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>

其中：

-h: 显示帮助信息（可选）

-v: 显示轨迹信息（可选）

-s <s>: 组索引位数

-E <E>: 关联度（每组包含的缓存行数）

-b <b>: 内存块内地址位数

-t <tracefile>: 内存访问轨迹文件名

根据执行模拟器程序时携带的命令行参数建立Cache后，逐行读入并分析操作类型和访存地址，对于L和S操作，若命中Cache则将LRU计数重置，否则查找Cache空行进行加载，若无空行则需要根据各行LRU计数的值找出被淘汰行实现替换。对于M操作，可以看作对同一地址依次执行一次L和S操作。

（2）理解代码框架

实验中需要完成的是csim.c文件，已给出代码框架，其中包含cache行的结构体定义，包含有效位、标识位和LRU计数。此外还有关键全局变量如下：

/\* Globals set by command line args \*/

int verbosity = 0; /\* 若为1则显示详细命中信息 \*/

int s = 0; /\* 组号位数 \*/

int b = 0; /\* 块内偏移位数 \*/

int E = 0; /\* 组内行数\*/

/\* Derived from command line args \*/

int S; /\* 组数 \*/

int B; /\* 块大小 (字节) \*/

/\* Counters used to record cache statistics \*/

int miss\_count = 0; /\* 缺失计数 \*/

int hit\_count = 0; /\* 命中计数 \*/

int eviction\_count = 0; /\* 驱逐计数 \*/

unsigned long long int lru\_counter = 1; /\* LRU计数 \*/

程序框架里已经给出的函数中，initCache()使用malloc函数根据命令行参数的值来动态申请Cache所需的内存空间；printUsage()展示程序用法；main()获取命令行参数，调用相关函数进行模拟。

可以看到Cache为一个二维结构，Cache指针是S个Cache块头指针构成的数组的头指针，每个Cache块头指针为E个Cache行结构体组成的数组的起始地址。Cache的替换算法采用LRU算法，LRU计数器的初始值为unsigned long long类型的最大值，每次命中都将计数器重置为初始值，其他情况下计数器值递减1。

程序框架里需要完善的函数中，freeCache()释放initCache函数所申请的内存空间；accessData(addr)对地址addr在Cache中查找，对命中、缺失等情况作出相应处理并统计信息；replayTrace()读取内存访问轨迹文件内容，从每一行解析出操作类型、目标内存地址，若为L或S操作则调用一次accessData函数，若为M操作则对于同一地址连续调用两次accessData函数。

（3）编写程序

根据上述（2）中的理解和分析，完善freeCache函数、accessData函数和replayTrace函数。

1. freeCache函数

由于Cache是一个二维结构，一个Cache包含S个组，每组中有E个Cache行，因此在使用free函数释放内存时首先以组为单位，对Cache每一组进行free操作，然后再对Cache进行free操作，最后将Cache指针置为NULL。

完成后该函数代码如下：

/\*

\* freeCache - free allocated memory

\*/

void freeCache()

{

int i;

for (i = 0; i < S; i++)

free(cache[i]);

free(cache);

cache = NULL;

}

2. accessData函数

首先使用位运算从接收参数addr中提取出组号set\_index和标识tag。

然后对于set\_index指定的组中的E个Cache行进行遍历。对于有效位为1的每一行，首先将LRU计数-1（LRU计数的默认值为unsigned long long类型的最大值），然后维护当前组中LRU最小值对应的Cache行号，若发生驱逐，则作为被驱逐行的行号，最后若目标tag与当前tag匹配，说明命中，记录下命中行号。对于有效位为0的行，记录下第一次出现的有效位为0的行，作为发生数据载入时的目标行（空行），此后跳过有效位为0的行。

再对命中情况作出处理，若命中行号不为-1，说明Cache命中，则打印命中信息、增加命中计数、将命中行LRU计数重置、将命中行号重置为-1。

最后对缺失情况作出处理，首先增加缺失计数。若空闲行号不为-1，说明存在空行，则打印载入信息、将空闲行信息改写为目标行信息、重置空闲行号为-1。否则说明不存在空行，则对被驱逐行号对应的行进行驱逐：增加驱逐计数、把目标行信息覆盖到被驱逐行，最后重置最小LRU计数为最大值。

完成后该函数代码如下：

/\*

\* accessData - Access data at memory address addr.

\* If it is already in cache, increast hit\_count

\* If it is not in cache, bring it in cache, increase miss count.

\* Also increase eviction\_count if a line is evicted.

\*/

void accessData(mem\_addr\_t addr)

{

int i;

lru\_counter = ULONG\_MAX;

unsigned int eviction\_line = 0;

mem\_addr\_t set\_index = (addr >> b) & set\_index\_mask;

mem\_addr\_t tag = addr >> (s + b);

cache\_set\_t cache\_set = cache[set\_index];

int hit\_index = -1; //命中行号

int invalid\_index = -1; //空闲行号

/\* 在set中的每一行查找 \*/

for (i = 0; i < E; i++)

{

if (cache\_set[i].valid)

{

cache\_set[i].lru--; //无论是否命中，默认将LRU计数-1

if (cache\_set[i].lru < lru\_counter)

{

lru\_counter = cache\_set[i].lru;

eviction\_line = i; //找出set中LRU计数最小值，若须替换则为替换行

}

if (cache\_set[i].tag == tag)

{

/\* 命中 \*/

hit\_index = i;

}

}

else if (invalid\_index == -1)

invalid\_index = i; //如果有的话，记录下第一个空行

}

/\* 命中处理 \*/

if (hit\_index != -1)

{

if (verbosity)

printf("Hit! line:%d tag:%lld\n", hit\_index, cache\_set[hit\_index].tag);

hit\_count++;

cache\_set[hit\_index].lru = ULONG\_MAX; //LRU计数重置为最大值

hit\_index = -1; //重置命中行号为-1

}

/\* 缺失处理 \*/

else

{

miss\_count++;

/\* 尝试找空位填入 \*/

if (invalid\_index != -1)

{

if (verbosity)

printf("Miss\_Insert! line:%d tag:%lld\n", invalid\_index, cache\_set[invalid\_index].tag);

cache\_set[invalid\_index].tag = tag;

cache\_set[invalid\_index].lru = ULONG\_MAX;

cache\_set[invalid\_index].valid = 1;

invalid\_index = -1; //重置空闲行号为-1

}

/\* 没有空位则替换 \*/

else

{

if (verbosity)

printf("Miss\_Evict! line:%d tag:%lld\n", eviction\_line, cache\_set[eviction\_line].tag);

eviction\_count++;

cache\_set[eviction\_line].tag = tag;

cache\_set[eviction\_line].lru = ULONG\_MAX;

lru\_counter = ULONG\_MAX; //重置最小LRU计数

}

}

}

3. replayTrace函数

在replayTrace函数中，将文件中满足首字符为空格的内容按行读取到buf中，同时读addr和len参数。从buf[0]获取操作类型，若为L或S操作则调用一次accessData函数，若为M操作则对同一地址连续调用两次accessData函数。

完成后该函数代码如下：

/\*

\* replayTrace - replays the given trace file against the cache

\*/

void replayTrace(char \*trace\_fn)

{

char buf[1000];

mem\_addr\_t addr = 0;

unsigned int len = 0;

FILE \*trace\_fp = fopen(trace\_fn, "r");

/\* 跳过空格，读取行内容 \*/

while (fscanf(trace\_fp, " %s %llx,%u", buf, &addr, &len) != EOF)

{

if (verbosity)

printf("%s %llx,%u\n", buf, addr, len);

switch (buf[0])

{

case 'L':

/\* Load \*/

accessData(addr);

break;

case 'S':

/\* Store \*/

accessData(addr);

break;

case 'M':

/\* Modify \*/

accessData(addr);

accessData(addr);

break;

default:

break;

}

}

fclose(trace\_fp);

}

* 1. **实验结果和分析**

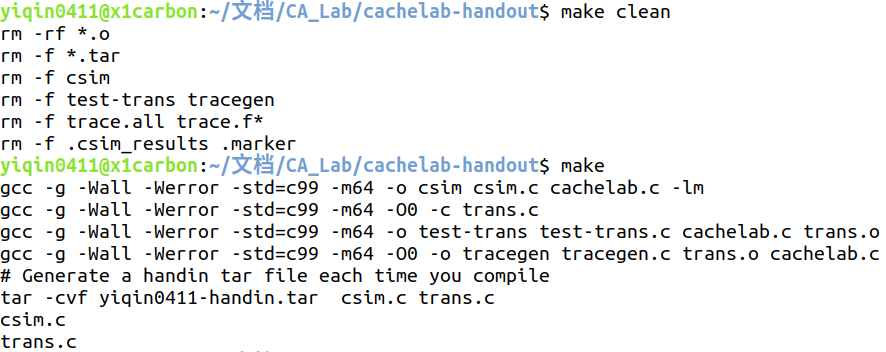
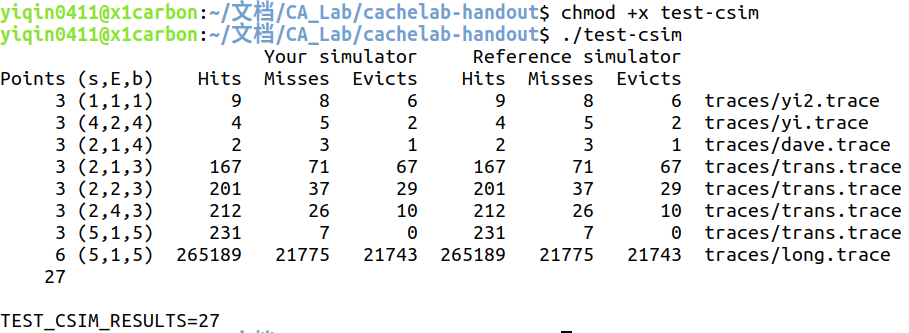
完成代码之后在源码所在根目录打开终端，使用老师提供的已经编写好的MakeFile进行编译。首先输入make clean命令清除上一次编译生成的文件，然后输入make命令进行编译，并且自动将源代码打包为tar压缩文件，如下图1.1所示。

图1.1 生成目标程序

编译完成后就可以使用老师提供的test-csim测试程序对程序正确性进行验证，验证结果如下图1.2所示：

图1.2 测试程序运行结果

从上述结果可知，本次实验中的Cache模拟器的模拟行为与参考模拟器的行为一致，正确实现了Cache的模拟功能。

再使用带-v参数的命令行对模拟器行为进行分析，选用yi.trace作为用例，构造一个4组、每组2行，块内偏移位数为4的Cache。

首先使用老师提供的手工演算Excel工具预测程序行为，以yi.trace为例的手工演算过程如下图1.3所示：

图1.3 手工演算的程序行为

从上图可以看到，第一步L操作访问第1块，此时Cache为空故缺失，将第1块装入Cache第1组第0行。第二步M操作访问第2块，先缺失，将第2块装入Cache第2组第0行，然后下一次访问命中。第三步L操作访问第2块，命中。第四步S操作访问第1块，命中。第五步L操作访问第17块，缺失，将第17块装入Cache第1组第1行。第六步L操作访问第33块，缺失，将第33块装入Cache第1组，但相联数为2，每组只有2行，Cache第1组已满，根据LRU策略，应该淘汰第1组第0行。第七步M操作访问第1块，先缺失，将第1块装入Cache，但Cache第1组已满，根据LRU策略，应该淘汰第1组第1行，然后下一次访问命中。

使用如下命令进行实际操作：

./csim -v -s 4 -E 2 -b 4 -t traces/yi.trace

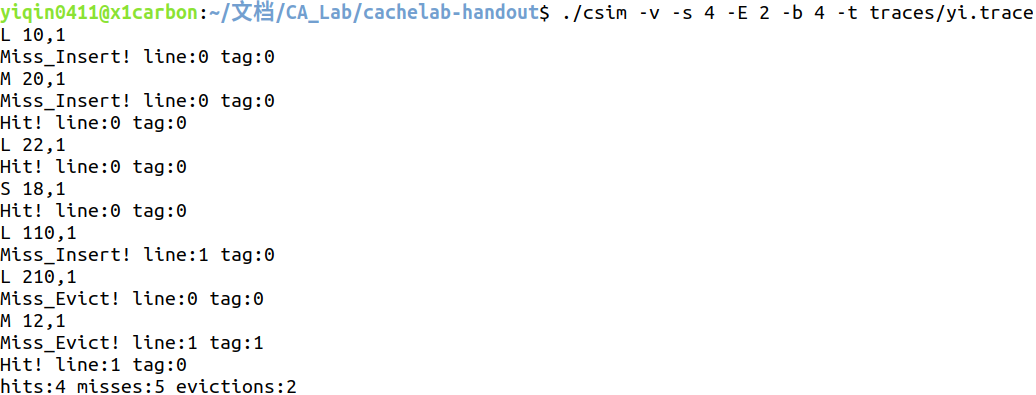
实际过程如下图1.4所示。

图1.4 实际程序行为

# 总结和体会

在计算机系统中对存储器的访问是非常频繁的操作，但CPU的速度和主存的速度相差几个量级，根据加快经常性事件的原理和程序的时间、空间局部性原理，为了提高整个系统的速度，在CPU和主存之间设置了Cache，将经常使用的数据存放在容量小但速度快的Cache中。在本次实验中，使用C语言实现了一个自定义规格Cache的模拟程序，模拟了Cache的构造和处理命中、缺失和冲突的过程。实验的代码框架中主要需要完成的就是访问Cache的程序段。在组内遍历Cache行，当有效位为1且tag匹配时命中，否则不命中，不命中时还需要区分载入新行还是替换已有行。

通过完成本次实验，整个过程使我对Cache的结构、Cache命中和缺失时的行为、Cache的LRU替换算法的规则有了更加深刻的理解和认识，实际的模拟器编写过程也让我对书本上的Cache模型有了实践层面的理解。

实验的指导书、代码框架、测试样例、手工演算工具，甚至实验环境的搭建教程都由老师提供完备，实验当天老师还开了直播进行实操讲解，真的非常感谢老师们的精心准备和耐心指导！

# 对实验课程的建议

实验本身的内容很不错，就是只有一个实验感觉内容有点单薄。我个人感觉系统结构课程里面的互联网络这一部分内容有点抽象，或许可以增加一点实验课时，模拟一下互联网络，加深对概念的理解。