

目录

第6章家庭作业-存储器层次结构	1. 1
6. 24	1. 2
6. 28	1. 3
6. 31	1. 4
6. 32	1. 5
6. 36	1. 6
6. 38	1. 7
6. 40	1. 8
6. 44	1. 9
cachelab	1. 10

李一鸣

1160300625

6.24

平均寻道时间 $T_{avg\ seek} = 4\ ms$

最大旋转延迟 $T_{max\ rotation} = 60\ s / 15000\ RPM * 1000\ ms/s = 4\ ms$

平均旋转延迟 $T_{avg\ rotation} = 1/2 * T_{max\ rotation} = 2\ ms$

平均传送时间 $T_{avg\ transfer} = T_{max\ rotation} / (1000\ 扇区/磁道) = 0.004\ ms$

A.

最好情况：块被映射到连续的扇区，只需在同一柱面上一块接一块地读。

$2MB \approx 2000\ KB = 4000\ 512\ B$ ；读这一连串数据磁盘旋转圈数为 $4000 / 1000 = 4$ 圈 因此读这个文件的总时间为： $T_{avg\ seek} + T_{avg\ rotation} + 4T_{max\ rotation} = 22\ ms$

随机情况：读4000块中的每一块都需要 $T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \approx 6\ ms$ ，读这个文件的总时间为 $4000 * 6\ ms = 24\ s$ 。

6.28

地址格式: CT(8位)+CI(3位)+CO(2位)

A.

组2中两行的有效位均为0, 不可能命中。

B.

组4中两行的有效位均为1

组索引CI = 0x4 标记CT = 0xC7 或 0x05 块偏移CO = 00-11

组合起来就是: 1 1000 1111 00xx 或者 0 0000 1011 00xx 在组4中命中的十六进制内存地址为: 0x18F0 0x18F1 0x18F2 0x18F3 0x00B0
0x00B1 0x00B2 0x00B3

C.

组5中行0有效位为1

组索引CI = 0x5 标记CT = 0x71 块偏移CO = 00-11

组合起来就是: 0 1110 0011 01xx

在组5中命中的十六进制内存地址为: 0x0E34 0x0E35 0x0E36 0x0E37

D.

组7中行1有效位为1

组索引CI = 0x7 标记CT = 0xDE 块偏移CO = 00-11

组合起来就是: 1 1011 1101 11xx

在组5中命中的十六进制内存地址为: 0x1BDC 0x1BDD 0x1BDE 0x1BDF

6.31

对于作业6.30中的高速缓存， $S = 8$, $s = 3$; $B = 4$, $b = 2$; $t = 13 - s - b = 8$

地址字段分配情况为：13位 = CT(8位) + CI(3位) + CO(2位)

A.

0x071A转换为二进制为：0 0111 0001 1010

B.

因此 $CT = 0011\ 1000 = 0x38$, $CI = 110 = 0x6$, $CO = 10 = 0x2$

组6中索引位0x38的元素标记为0, 因此不命中。

6.32

对于作业6.30中的高速缓存， $S = 8, s = 3; B = 4, b = 2; t = 13 - s - b = 8$

地址字段分配情况为：13位 = CT(8位) + CI(3位) + CO(2位)

A.

0x16E8转换为二进制为：1 0110 1110 1000

B.

因此 $CT = 1011\ 0111 = 0xB7, CI = 010 = 0x2, CO = 0 = 0x0$

组2中没有索引位为0xB7的元素，因此不命中。

参数	值
高速缓存块偏移(CO)	0x0
高速缓存组索引(CI)	0x2
高速缓存标记(CT)	0xB7
高速缓存命中？	否

6.36

```

int x[2][128];
int i;
int sum = 0;

for (i = 0; i < 128; i++) {
    sum += x[0][i] * x[1][i];
}

```

A.

容量C = 512; 由于是直接映射的，高速缓存行数E = 1; 高速缓存块数B = 16, b = 4; 组数S = C/E/B = 32, s = 5

总共需要读取数据 $2 * 128 = 256$ 次

元素	地址
x[0][0]	0
x[0][i]	4i
x[1][i]	sizeof(int)*(128+i) = 4i+512

$4i\%C = (4i+512)\%C$, 因此x[0][i]和x[1][i]会缓存到同一数据块中，这样会造成冲突不命中，不命中率为100%。

B.

C = 1024; E = 1; B = 16, b = 4; S = 64, s = 6

$\text{sizeof}(x) == 2 \cdot 128 \cdot \text{sizeof}(\text{int}) = 1024 = C$

整个数组都能被缓存下来。每B = 16字节共用一个组， $16/\text{sizeof}(\text{int}) = 4$ ，也就是说每有1次不命中，就会缓存4个int，这样后续3个int就都会命中，因此不命中率为25%。

C.

C = 512; E = 2; B = 16, b = 4; S = 16

与A的区别在于：x[0][i]和x[1][i]缓存到同一组的不同行中。

$C/\text{sizeof}(\text{int}) = 126$, 能存下126个int。

(1) 在读取前126个元素时

```

for (i = 0; i < 64; i++)
    sum += x[0][i] * x[1][i];

```

所有元素都能存下，类似于B，不命中率为25%。

(2) 在读取后126个元素时

```

for (i = 64; i < 128; i++)
    sum += x[0][i] * x[1][i];

```

x[0][i]不命中，根据Least-Recently-Used策略，缓存x[0][i]，替换x[0][i-64]，同时也会缓存x[1][i]，替换x[1][i-64]。后续跟(1)类似。

D.

不会。

C变大而B仍为16, $B/\text{sizeof}(\text{int}) = 4$, 最佳情况就是4次读取只有一次不命中，不命中率为25%。

E.

会。假设B = 32, $32/\text{sizeof}(\text{int}) = 8$, 8次有一次不命中，不命中率为12.5%。

6.38

A.

写总数为: $4 \cdot 16 \cdot 16 = 2^{10}$

B.

$C = 2048; E = 1; B = 32, b = 5; S = 64, s = 6$
 $\text{sizeof}(\text{point_color}) = 16, B / 16 = 2$

```
square[i][j].c = 0
```

不命中, 缓存两个结构体, 接下来

```
square[i][j].m = 0
square[i][j].y = 0
square[i][j].k = 0
square[i][j+1].c = 0
square[i][j+1].m = 0
square[i][j+1].y = 0
square[i][j+1].k = 0
```

全部命中。

因此不命中的写总数为 $2^{10} / 8 = 128$ 。

C.

1/8

6. 40

A.

写总数为: $4 * 16 * 16 = 2^{10}$

B.

(1) 第一个循环中

```
for(int i = 0; i < 16; i++){
    for(int j = 0; j < 16; j++){
        square[i][j].y = 1;
    }
}
```

```
square[0][0].y = 1
```

不命中, 缓存两个结构体

```
square[0][1].y = 1
```

命中

后续如此循环, 不命中的写总数为: $16 * 16 / 2 = 128$

(2) 第二个循环中

```
for(int i = 0; i < 16; i++){
    for(int j = 0; j < 16; j++){
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].k = 0;
    }
}
```

```
square[0][0].c = 0
```

不命中, 缓存两个结构体

```
square[0][0].m = 0
square[0][0].k = 0
square[0][1].c = 0
square[0][1].m = 0
square[0][1].k = 0
```

命中

后续如此循环, 不命中的写总数为: $16 * 16 * 3 / 6 = 128$

总的不命中的写总数为256。

C.

 $256/2^{10} = 1/4$

6.44

csapp在[这里](#)列出了书中的所有代码，下载链接在[这里](#)，我已经下载下来放到了本仓库code下。

在我的Ubuntu 16.04, Core i5 机器上运行结果为：

Clock frequency is approx. 500.0 MHz										
Memory mountain (MB/sec)										
s10	s11	s12	s13	s14	s15	s6	s7	s8	s9	
198	128m	2425	1264	777	589	460	376	313	306	211
198	64m	187	177	171	176	188	490	399	315	286
198	32m	2445	1296	802	609	184	414	325	270	244
263	16m	187	178	171	166	624	503	423	342	285
226	8m	2501	1312	810	168	187	423	342	280	243
269	4m	170	163	154	692	512	415	376	268	
355	2m	2630	1391	894	159	154	354			
783	1024k	199	187	181	627	507	448			
949	512k	2056	1774	1514	1043	803	773			
1020	1024	354	351	353	347	352	824			
1677	256k	2461	1972	1627	1390	1165	1016			
1943	1632	760	749	772	749	740	914			
1943	128k	1024k	3575	2884	2388	1915	1599	1374	1196	1060
1949	964	4498	940	921	900	1490	1295	1138	1108	1082
5280	512k	4736	3917	3827	3558	3123	2703	2268	2040	1996
4595	1040	2013	2216	3805	2567	2225	1953	1693	1525	1512
4595	256k	5789	5580	5527	5535	5388	5332	5088	5305	5384
4595	1632	5163	5043	4918	5417	5318	5170	4834	4830	4933
4595	128k	5834	5580	5482	5417	5318	4834	4830	4840	
4595	64k	4611	4701	4365	4200					

Cache Lab

李一鸣

1160300625

Part A: Writing a Cache Simulator 缓存模拟器

编写csim.c。

1. 定义缓存行结构体数据类型:

```
typedef struct cache_line{
    int valid;
    mem_addr_t tag;
    unsigned long long int times;
}cache_line_t;
```

|valid|有效位| |mem_addr_t| 标记位| |times| 访问次数|

`times` 用来实现Least-Recently-Used替换策略。每次访问某一行时，将这一行的 `times` 设为比所有其他行都大的最大值。
在全局变量里有 `times_counter`，记录着下一次将要赋给最新访问的行的 `times` 值。

2. 缓存有S组、E行，因此定义一个缓存行二维数组 `cache` :

```
typedef cache_line_t* cache_set_t;
typedef cache_set_t* cache_t;

/* The cache we are simulating */
cache_t cache;
```

3. `initCache()` 负责初始化 `cache`

4. `replayTrace()` 负责模拟缓存运行：

关键在于调用了 `accessData()`，`accessData` 负责处理缓存的hit、miss、eviction。

5. `freeCache` 释放 `chche` 申请的内存。

Part B: Optimizing Matrix Transpose 优化矩阵转置

编写trans.c。

缓存大小C = S E B = 32 1 32 Bytes = 1 KB

1. M = 32 N = 32

将3232的数组转换成16个88的块，数组能够完全缓存下来。只会出现冷不命中。

同时对角线元素访问时由于A、B的tag不同，会出现冲突不命中，用tmp暂存起来。

2. M = 64 N = 64

定义了8个临时变量，同时为了防止冲突不命中，使用了行跳转。

3. M = 61 N = 67

不能使用第一种方法，因为矩阵不是方的，可能会出现有些元素访问不到的错误、非法访问数据。

使用16*16的块，并进行越界检查。