

## 第六章

6.30 假设我们有一个具有如下属性的系统：

- 存储器是字节寻址的。
- 存储器访问是对1字节字的（而不是4字节字）。
- 地址宽12位。
- 高速缓存是四路组相联的（ $E=2$ ），块大小为4字节（ $B=4$ ），有四个组（ $S=4$ ）。

高速缓存的内容如下，所有的地址、标记和值都以十六进制表示：

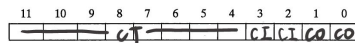
组索引	标记	有效位	字节0	字节1	字节2	字节3
0	00	1	40	41	42	43
	83	1	FE	97	CC	D0
1	000 83 1	1	44	45	46	47
	83	0	—	—	—	—
2	00	1	48	49	4A	4B
	40	0	—	—	—	—
3	0100 FF	1	9A	C0	03	FF
	00	0	—	—	—	—

www.TopSage.com

### 436 第一部分 程序结构和执行

A. 下面的图给出了一个地址的格式（每个小框表示一位）。指出用来确定下列信息的字段（在图中标号出来）：

- CO 高速缓存块偏移
- CI 高速缓存组索引
- CT 高速缓存标记



B. 对于下面每个存储器访问，当它们是按照列出来的顺序执行时，指出是高速缓存命中还是不命中。如果可以从小高速缓存中的信息推断出来的话，请给出读出的值。

0100 0000 1001  
0100 0000 1010  
1000 0011 0011

操作	地址	命中?	读出的值(或者未知)
读	0x409	X	
写	0x40A	✓	
读	0x83B	✓	0x00

cache中有

6.31 假设我们有一个具有如下属性的系统：

- 存储器是字节寻址的。
- 存储器访问是对1字节字的（而不是4字节字）。
- 地址宽13位。
- 高速缓存是四路组相联的（ $E=4$ ），块大小为4字节（ $B=4$ ），有八个组（ $S=8$ ）。

考虑下面的高速缓存状态。所有的地址、标记和值都以十六进制表示。每组有四行，索引列包含组索引。标记列包含每一行的标记值。V列包含每一行的有效位。字节0~3列包含每一行的数据，标号从左向右，字节0在左边。

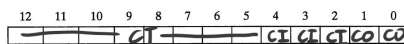
4路组相联高速缓存

索引	标记	V	字节 0~3	索引	标记	V	字节 0~3	索引	标记	V	字节 0~3	索引	标记	V	字节 0~3
0	FO	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A			
1	BC	0	03 3E CD 39	A0	0	16 7B ED 6A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02			
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E			
3	BE	0	2F 7E 3D A8	CO	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2			
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA			
5	98	0	A9 76 2B EE	54	0	BC 91 D6 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A			
6	8B	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	8B	1	FA 93 EB 48			
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30			

A. 这个高速缓存的大小（C）是多少字节？

B. 下面的图给出了一个地址的格式（每个小框表示一位）。指出用来确定下列信息的字段（在图中标号出来）：

- CO 高速缓存块偏移
- CI 高速缓存组索引
- CT 高速缓存标记



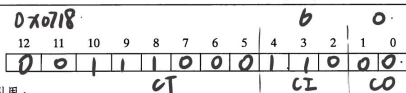
6.32 假设程序使用作业6.31中的高速缓存，引用位于地址0x0718处的1字节字。用十六进制表示出它所访问的高速缓存条目，以及返回的高速缓存字节值。指明是否发生了高速缓存不命中。如果有高速缓存不命中，对于“返回的高速缓存字节”输入“—”。提示：注意那些有效位！

A. 地址格式（每个小框表示一位）：

www.TopSage.com

3B: 0011 1000. 1100

### 第6章 存储器层次结构 437



B. 存储器引用：

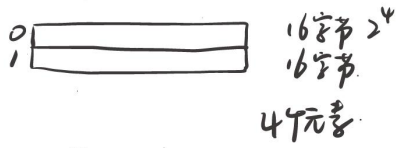
参数	值
块偏移量 (CO)	0x 00
索引 (CI)	0x 06
高速缓存标记 (CT)	0x 3B
高速缓存命中? (是/否)	是
返回的高速缓存值	0x FA

6.32

对于作业 0.31 中的问题继续讨论，列出包含组 0 中所有八个字节地址（以下表格表示）。

考虑下面的矩阵转置函数：

```
1 typedef int array[4][4];
2
3 void transpose2(array dst, array src)
4 {
5     int i, j;
6
7     for (i = 0; i < 4; i++) {
8         for (j = 0; j < 4; j++) {
9             dst[i][j] = src[j][i];
10        }
11    }
12 }
```



假设这段代码运行在一台具有如下属性的机器上：

- `sizeof(int) == 4`。
- 数组 `src` 从地址 0 开始，而数组 `dst` 从地址 64 开始（十进制）。
- 只有一个 L1 数据高速缓存，它是直接映射、直写、写分配的，块大小为 16 字节。
- 这个高速缓存总共有 32 个数据字节，初始为空。
- 对 `src` 和 `dst` 数组的访问分别是读和写不命中的唯一来源。

对于每个 `row` 和 `col`，指明对 `src[row][col]` 和 `dst[row][col]` 的访问是命中 (h) 还是不命中 (m)。

`src[col], src[col]` 对应组 0 | `src[col][row]` 到 `dst[col][row]`  
`col, col` 对应组 1 | 对应组 1 次

例如，读 `src[0][0]` 会不命中，而写 `dst[0][0]` 也会不命中。  
 组 0 组 1 组 0 组 1 组 0 组 1 组 0 组 1

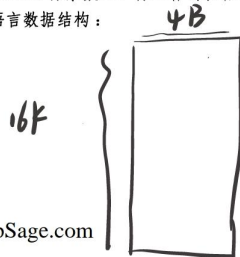
		dst 数组				src 数组			
		列 0	列 1	列 2	列 3	列 0	列 1	列 2	列 3
行 0	组 0 <code>src[0][0]</code>	m	h	m	h	行 0	m	m	m
行 1	组 1 <code>dst[1][0]</code>	m	m	h	m	行 1	m	m	m
行 2	组 0 <code>src[2][0]</code>	m	h	m	h	行 2	m	m	m
行 3	组 1 <code>dst[3][0]</code>	m	m	h	m	行 3	m	m	m

←6.35

6.35 C 个命中需要多少？

你正在编写一个新的 3D 游戏，希望能名利双收。你正在写一个函数，使得在画下一帧之前先清空屏幕缓冲区。你工作的屏幕是 640×480 像素数组。你工作的机器有一个 64KB 直接映射高速缓存，每行 4 个字节。你使用下面的 C 语言数据结构：

```
1 struct pixel {
2     char r;
3     char g;
4     char b;
5     char a;
6 };
7
```



$2^4 \times 2^{10} = 2^{14}$  2

```
8 struct pixel buffer[480][640];
9 int i, j;
10 char *cptr;
11 int *iptr;
```

有如下假设：

- `sizeof(char) == 1` 和 `sizeof(int) == 4`。
- `buffer` 起始于存储器地址 0。
- 高速缓存初始为空。
- 唯一的存储器访问是对于 `buffer` 数组中元素的访问。变量 `i`、`j`、`cptr` 和 `iptr` 被存放在寄存器中。

下面代码中百分之多少的写会在高速缓存中不命中？

```
1 for (j = 0; j < 640; j++) {
2     for (i = 0; i < 480; i++) {
3         buffer[i][j].r = 0;
4         buffer[i][j].g = 0;
5         buffer[i][j].b = 0;
6         buffer[i][j].a = 0;
7     }
8 }
```

第一个不命中 100% 加载 cache  
 后续命中 25%

←6.42

## 第九章

9.13:

### 9.6.4 综合：端到端的地址翻译

在这一节里，我们通过一个具体的端到端的地址翻译示例，来总结一下我们刚学过的这些内容。这个示例运行在一个有 TLB 和 L1 d-cache 的小系统上。为了保证可管理性，我们做出如下假设：

- 存储器是按字节寻址的。
- 存储器访问是针对 1 字节的字的（不是 4 字节的字）。
- 虚拟地址是 14 位长的 ( $n=14$ )。
- 物理地址是 12 位长的 ( $m=12$ )。
- 页面大小是 64 字节 ( $P=64$ )。
- TLB 是四路组相连的，总共有 16 个条目。
- L1 d-cache 是物理地址、直接映射的，行大小为 4 字节，而总共有 16 个组。

图 9-19 展示了虚拟地址和物理地址的格式。因为每个页面是  $2^6 = 64$  字节，所以虚拟地址和物理地址的低 6 位分别作为 VPO 和 PPO，虚拟地址的高 8 位作为 VPN，物理地址的高 6 位作为 PPN。

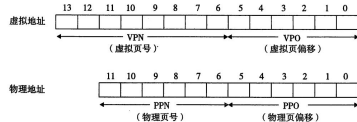


图 9-19 小存储系统的寻址。假设 14 位的虚拟地址 ( $n=14$ )，12 位的物理地址 ( $m=12$ ) 和 64 字节的页面 ( $P=64$ )。

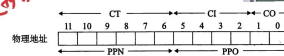
图 9-20 展示了我们小存储系统的一个快照，包括 TLB (见图 9-20a)、页表的一部分 (见图 9-20b) 和 L1 高速缓存 (见图 9-20c)。在 TLB 和高速缓存的图上面，我们还展示了访问这些设备时硬件是如何划分虚拟地址和物理地址的位的。



a) TLB: 四组, 16 个条目, 四路组相联

VPN	PPN	有效位	VPN	PPN	有效位
00	28	1	08	13	1
01	—	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	—	0
04	—	0	0C	—	0
05	16	1	0D	2D	1
06	—	0	0E	11	1
07	—	0	0F	0D	1

b) 页表: 只展示了前 16 个 PTE



索引	标记位	有效位	块 0	块 1	块 2	块 3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	54	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

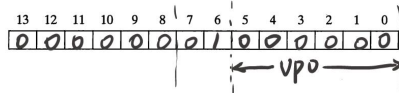
c) 高速缓存: 16 个组, 4 字节的块, 直接映射

图 9-20 小存储系统的 TLB、页表以及缓存。TLB、页表和缓存中所有的值都是十六进制表示的

9.13 对于下面的地址，重复习题 9.11：

虚拟地址：0x0040

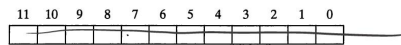
A. 虚拟地址格式



B. 地址翻译

参数	值
VPN	0x01
TLB 索引	0x01
TLB 标记	0x00
TLB 命中? (是/否)	否
缺页? (是/否)	是
PPN	—

C. 物理地址格式



D. 物理地址引用

参数	值
字节偏移	—
缓存索引	—
缓存标记	—
缓存命中? (是/否)	—
返回的缓存字节	—

9.14:

9.14 假设有一个输入文件 hello.txt，由字符串“Hello, world!\n”组成，编写一个 C 程序，使用 mmap 将 hello.txt 的内容改变为“Jello, world!\n”。

#### 9.8.4 使用 mmap 函数的用户级存储器映射

Unix 进程可以使用 mmap 函数来创建新的虚拟存储器区域，并将对象映射到这些区域中。

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

返回：若成功时则为指向映射区域的指针，若出错则为 MAP\_FAILED (-1)。

mmap 函数要求内核创建一个新的虚拟存储器区域，最好是从地址 start 开始的一个区域，并将文件描述符 fd 指定的对象的一个连续的片 (chunk) 映射到这个新的区域。连续的对象片大小为 length 字节，从距文件开始处偏移量为 offset 字节的地方开始。start 地址仅仅是一个暗示，通常被定义为 NULL。为了我们的目的，我们总是假设起始地址为 NULL。图 9-32 描述了这些参数的意义。

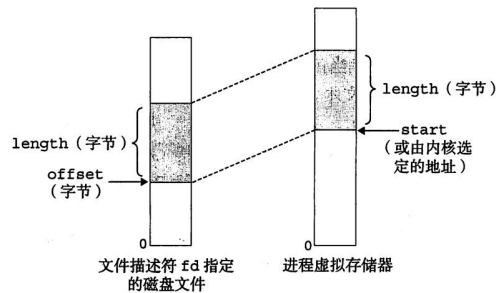


图 9-32 mmap 参数的可视觉解释

参数 prot 包含描述新映射的虚拟存储器区域的访问权限位 (在相应区域结构中的 vm\_prot 位)。

- PROT\_EXEC：这个区域内的页面由可以被 CPU 执行的指令组成。未与任何文件关联
- PROT\_READ：这个区域内的页面可读。
- PROT\_WRITE：这个区域内的页面可写。
- PROT\_NONE：这个区域内的页面不能被访问。

参数 flags 由描述被映射对象类型的位组成。如果设置了 MAP\_ANON 标记位，那么被映射的对象就是一个匿名对象，而相应的虚拟页面是请求二进制零的。MAP\_PRIVATE 表示被映射的对象是一个私有的、写时拷贝的对象，而 MAP\_SHARED 表示是一个共享对象。例如

```
bufp = Mmap(-1, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

让内核创建一个新的包含 size 字节的只读、私有、请求二进制零的虚拟存储器区域。如果调用成功，那么 bufp 包含新区域的地址。

munmap 函数删除虚拟存储器的区域：

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

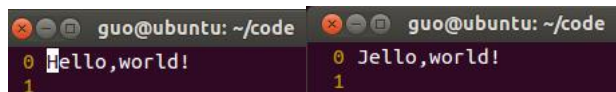
返回：若成功则为 0，若出错则为 -1。

munmap 函数删除从虚拟地址 start 开始的，由接下来 length 字节组成的区域。接下来对已删除区域的引用会导致段错误。

代码：

```
guo@ubuntu: ~/code
0 #include <sys/types.h>
1 #include <sys/stat.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <sys/mman.h>
5 int main()
6 {
7     int fd;
8     char *start;
9     fd = open("hello.txt", O_RDWR, 0); //打开文件
10    start = mmap(NULL, 1, PROT_WRITE, MAP_SHARED, fd, 0);
11    close(fd);
12    if(start == MAP_FAILED) return -1; //判断是否映射成功
13    (*start) = 'J';
14    munmap(start, 1);
15    return 0;
16 }
```

执行后:



mmap 允许用户将文件或其他对象映射到进程的虚拟内存空间中, 它创建了一个文件到内存的直接映射, 使用户能够像操作内存一样直接操作文件的内容。多个进程使用 mmap 映射同一个文件时, 它们共享的是同一份页缓存。

使用 mmap 进行文件读写时, 由于文件内容已经被映射到进程的地址空间中, 可以直接操作内存, 避免了不必要的数据拷贝, 从而提高效率。

## 第五章

### 5.19:

**\*\*5.19** 库函数 `memset` 的原型如下:

```
void *memset(void *s, int c, size_t n);
```

这个函数将从 `s` 开始的 `n` 个字节的存储器区域都填充为 `c` 的低位字节。例如, 通过将参数 `c` 设置为 0, 可以用这个函数来对一个存储器区域清零, 不过用其他值也是可以的。

下面是 `memset` 最直接的实现:

```
1  /* Basic implementation of memset */
2  void *basic_memset(void *s, int c, size_t n)
3  {
4      size_t cnt = 0;
5      unsigned char *schar = s;
6      while (cnt < n) {
7          *schar++ = (unsigned char) c;
8          cnt++;
9      }
10     return s;
11 }
```

实现该函数一个更有效的版本, 使用数据类型为 `unsigned long` 的字来装下 4 个 (对于 IA32) 或者 8 个 (对于 x86-64) 个 `c`, 然后用字级的写遍历目标存储器区域。你可能发现增加额外的循环展开会有所帮助。在 Intel Core i7 机器上, 我们能够把 CPE 从直接实现的 2.00 降低到对于 IA32 为 0.25, 和对 x86-64 为 0.125, 每个周期写 4 个或者 8 个字节。这里是一些额外的指导原则。在此, 假设  $K$  表示你运行程序的机器上的 `sizeof(unsigned long)` 的值。

- 你不可以调用任何库函数。
- 你的代码应该对任意  $n$  的值都能工作, 包括当它不是  $K$  的倍数的时候。你可以用类似于使用循环展开时完成最后几次迭代的方法做到这一点。
- 你写的代码应该做到无论  $K$  的值是多少, 都能够正确编译和运行。使用操作 `sizeof` 来做到这一点。
- 在某些机器上, 未对齐的写可能比对齐的写慢很多。(在某些非 x86 机器上, 未对齐的写甚至可能会导致错误。) 写出这样的代码, 开始时直到目的地址是  $K$  的倍数时, 使用字节级的写, 然后进行字级的写, (如果需要) 最后采用用字节级的写。
- 注意 `cnt` 足够小以至于一些循环上界变成负数的情况。对于涉及 `sizeof` 运算符的表达式, 可以用无符号运算来执行测试。(参见 2.2.8 节和家庭作业 2.72。)

具体解释可见注释。



```

1  #include <bits/stdc++.h>
2  void my_memset(void *s, unsigned char c, size_t n)
3  {
4      size_t K = sizeof(unsigned long); // 1. 获取平台上的unsigned long类型的大小 (字节数)
5      size_t cnt = 0; // 2. 计数器, 用于跟踪已设置的字节数
6
7      // 3. 开始部分进行字节级的写直到对齐
8      unsigned char *schar = (unsigned char *)s; // 转换指针s为unsigned char*, 以便按字节操作
9      while ((size_t)schar % K != 0 && cnt < n) // 检查schar是否对齐到unsigned long的边界, 同时确保不超过要设置的字节数
10     {
11         *schar++ = c; // 设置当前字节为c
12         cnt++; // 计数器加1
13     }
14
15     // 4. 合成longc满足K个字节, 每个字节都是c的值
16     unsigned long longc = 0; // 使用unsigned long, 因为我们只填充到unsigned long的大小
17     unsigned char *temp = (unsigned char *)&longc; // 转换指针以按字节操作longc
18     for (size_t i = 0; i < K; i++) // 循环K次来填充longc的每个字节
19     {
20         *temp++ = c; // 设置longc的每个字节为c
21     }
22
23     // 5. 对齐后进行字级的写
24     unsigned long *slong = (unsigned long *)schar; // 转换指针schar为unsigned long*, 以便按字操作
25     while (cnt + K <= n) // 确保剩余要设置的字节数足以填充一个完整的unsigned long
26     {
27         *slong++ = longc; // 设置当前unsigned long为longc的值
28         cnt += K; // 计数器加上unsigned long的大小
29     }
30
31     // 6. 结尾部分可能的未成字部分进行字节级的写
32     schar = (unsigned char *)slong; // 将指针slong转换回unsigned char*以继续按字节操作
33     while (cnt < n)
34     {
35         *schar++ = c; // 设置剩余的字节为c

```

```

6         cnt++; // 计数器加1
7     }
8
9     return s; // 7. 返回原始指针s
10 }
11
12 int main()
13 {
14     char m[10]; // 8. 定义一个包含10个字符的数组m
15     my_memset(m, 0x11, sizeof(m)); // 9. 调用my_memset来设置数组m的每个字节为0x11
16
17     for (size_t i = 0; i < sizeof(m) / sizeof(char); i++) // 10. 遍历数组m的每个元素
18     {
19         std::cout << "count " << i << ": " << std::hex << static_cast<int>(m[i]) << "\n"; // 11. 打印当前元素的索引和值
20     }
21     return 0;
22 }

```

结果:


```

guoruilong@guoruilong-virtual-machine:~/cs/hw3$ ./main
count 1: 11
count 2: 11
count 3: 11
count 4: 11
count 5: 11
count 6: 11
count 7: 11
count 8: 11
count 9: 11

```

## 5.21:

- 5.21 在练习题 5.12 中, 我们能够把前置和计算的 CPE 减少到 3.00, 这是由该机器上浮点加法的延迟决定的。简单的循环展开没有改进什么。
- 使用循环展开和重新结合的组合, 写出求前置和的代码, 能够得到一个小于你机器上浮点加法延迟的 CPE。例如, 我们使用 2 次循环展开的版本每次迭代需要 3 个加法, 而使用 3 次循环展开的版本需要 5 个。

 练习题 5.12 重写 psum1 (图 5-1) 的代码, 使之不需要反复地从存储器中读取  $p[i]$  的值。不需要使用循环展开。得到的代码测试出的 CPE 等于 3.00, 受浮点加法延迟的限制。

```

1  /* Compute prefix sum of vector a */
2  void psum1(float a[], float p[], long int n)
3  {
4      long int i;
5      p[0] = a[0];
6      for (i = 1; i < n; i++)
7          p[i] = p[i-1] + a[i];
8  }

```

优化后:

```

1  void psum(float a[], float p[], long int n)
2  {
3      long int i;
4      int v = 0;
5      for(i=0; i<n-1; i+=2)
6      {
7          int v1 = a[i];
8          int v2 = a[i+1];
9          v2 = v1 + v2;
10         p[i] = v + v1;
11         p[i+1] = v + v2;
12         v = v + v2;
13     }
14     for(; i<n; i++)
15     {
16         v = v + a[i];
17         p[i] = v;
18     }
19 }

```

使用了 2 次循环展开。