华东师范大学软件工程学院实验报告

姓　　名:	李鹏达	学　　号:	10225101460
实验编号:	Lab 03	实验名称:	Building A Cache Simulator

1　实验目的

1) 深入了解高速缓存的原理与结构

2) 学习通过利用高速缓存的结构优化效率

2　实验内容与实验步骤

2.1　实验内容

本练习将帮助您了解缓存内存对 C 语言性能的影响程序。

实验室由两部分组成。在第一部分中，您将编写一个小的 C 程序（大约 200-300 行）

模拟高速缓存的行为。在第二部分中，您将优化一个小矩阵转置函数，目的是最大限度地减少缓存未命中次数。

2.1.1　Part A:Writing a Cache Simulator

在 A 部分中，您将在 csim.c 中编写一个缓存模拟器，该模拟器将 valgrind 内存跟踪作为输入，模拟此跟踪上缓存的命中/未命中行为，并输出命中，未命中和替换总数。

我们先对原理进行分析。在输入中会提供高速缓存的 s、E、b，从而确定该高速缓存有 $S = 2^s$ 组，每组中有 E 行，每行的高速缓存块大小为 $B = 2^b$ 字节，通过 malloc 函数进行动态内存分配即可构造出模拟的高速缓存。依据存储器层次结构，来模拟高速缓存块映射的命中、不命中以及替换过程，替换策略为 LRU(最近最少使用)。

首先，我们先定义行结构体和所需的全局变量。

```
1 typedef struct {
2     int valid; // valid
3     long tag;  // tag
4     long time; // time
5 } line;
6
```

```
line** cache;

int hits = 0;
int misses = 0;
int evictions = 0;

int s, E, b;
char trace_file[1000];
int verbose = 0; // false
```

接下来，我们需要在 main 函数中对命令行输入的参数进行解析。

getopt 是一个在 UNIX 类操作系统中常用的库函数，它用于解析命令行参数。该函数会返回在命令行中找到的选项字符，如果选项需要一个参数，那么 optarg 全局变量就会被设置为该参数。

函数通过字符串”hvs:E:b:t:” 定义了可以接受的命令行选项。在这个字符串中，如果选项后面有冒号，那么它就需要一个参数。

处理每个选项：函数使用了一个 switch 语句来处理找到的每个选项。case ’h’ 是显示帮助信息，case ’v’ 是决定是否输出详细信息，case ’s’、’E’ 和’b’ 是设置缓存参数，case ’t’ 指定输入文件。

如果 getopt 找到了一个未知的选项，那么函数会显示使用说明，并退出程序。这个行为可以帮助用户了解如何正确使用程序。

总的来说编写思路是：使用 getopt 函数解析命令行参数，并针对每个可接受的选项进行处理。如果遇到了未知的选项，就显示使用说明并退出程序。

```
char opt;
const char* usage = "usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <
    tracefile>";
while ((opt = getopt(argc, argv, "hvs:E:b:t:")) != EOF) {
    switch (opt) {
        case 'h':
            fprintf(stdout, "%s", usage);
            exit(0);
            break;
        case 'v':
            verbose = 1;
            break;
        case 's':
            s = atoi(optarg);
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
```

```
19              b = atoi(optarg);
20              break;
21          case 't':
22              strcpy(trace_file, optarg);
23              break;
24          default:
25              fprintf(stdout, "%s", usage);
26              exit(-1);
27              break;
28      }
29  }
```

接下来，我们需要对高速缓存进行初始化. 我们首先计算 $S = 2^s$，然后使用 malloc 函数动态分配内存。

```
1  void init() {
2      int S = 1 << s; // 2 ^ s
3      cache = (line**)malloc(sizeof(line*) * S);
4      for (int i = 0; i < S; i++) {
5          cache[i] = (line*)malloc(sizeof(line) * E);
6          for (int j = 0; j < E; j++) {
7              cache[i][j].valid = 0;
8              cache[i][j].tag = -1;
9              cache[i][j].time = 0;
10         }
11     }
12 }
```

然后，我们需要读取处理文件中的命令。

处理输入文件：接着，函数开始读取输入文件。对于文件中的每一行，函数首先判断是否是'I'开头，如果是，就忽略这一行。否则，函数就解析这一行，得到操作类型（'S'、'L' 或'M'），地址和大小。然后，根据操作类型，函数调用 update 函数模拟一次 CPU 访问缓存的过程。

处理详细输出：在每次调用 update 之后，如果 verbose 变量被设置为 1，那么函数就会输出一些详细信息，包括操作类型、地址、大小，以及这次访问的结果（命中、未命中或替换）。

释放内存和关闭文件：在处理完输入文件的所有行之后，释放了申请的内存，关闭输入文件。

```
1  void get_trace() {
2      char operation;
3      unsigned long address;
4      int size;
5      FILE* fp = fopen(trace_file, "r");
6      if (fp == NULL) exit(-1);
```

```
7       while (fscanf(fp, " %c %lx,%d\n", &operation, &address, &size) > 0) {
8
9           // do operation
10          switch (operation) {
11              case 'L':
12                  update(address, operation, size);
13                  break;
14              case 'S':
15                  update(address, operation, size);
16                  break;
17              case 'M':
18                  update(address, operation, size);
19                  update(address, operation, size);
20                  break;
21          }
22
23          // update time
24          for (int i = 0; i < (1 << s); i++) {
25              for (int j = 0; j < E; j++) {
26                  if (cache[i][j].valid) {
27                      cache[i][j].time++;
28                  }
29              }
30          }
31      }
32      fclose(fp);
33  }
34
35  void free_cache() {
36      for (int i = 0; i < (1 << s); i++) {
37          free(cache[i]);
38      }
39      free(cache);
40  }
```

接下来，我们应该具体模拟高速缓存的工作过程。

update 函数的目的是模拟 CPU 访问缓存的过程。给定一个内存地址，这个函数会在缓存中查找对应的行，然后根据查找结果更新命中次数 (hits)、未命中次数 (misses) 和替换次数 (evictions)。以下是实现这个目的的详细步骤：

计算标签和集合索引：首先，函数计算了给定地址的标签 (tag) 和集合索引 (set)。这是通过移位操作和位掩码完成的。对于给定的内存地址，标签是地址的高位部分，集合索引是地址的中间部分。这些部分的大小由缓存的参数决定。

访问对应的缓存集合：然后，函数访问了缓存中对应的集合。这个集合是一个 line 结构体的数组，每个元素代表一行。

在集合中查找标签：接着，函数在集合中查找标签。如果找到了对应的标签，那么就发生了一次命中，函数就会增加命中次数，并更新对应行的时间戳。

处理未命中的情况：如果在集合中没有找到标签，那么就发生了一次未命中。此时，函数会查找一个空行或者使用 LRU 策略找到一个要被替换的行。如果找到了空行，那么就将新标签放入这个行；如果所有的行都不为空，那么就替换最久未使用的行。在这个过程中，函数可能会增加未命中次数和替换次数。

通过这些步骤，update 函数实现了 CPU 访问缓存的模拟。这个函数的实现思路是基于缓存的工作原理和 LRU 替换策略的。

```
1  void update(unsigned long address, char operation, int size) {
2      int set = (address >> b) & ((-1U) >> (64 - s));
3      int tag = address >> (b + s);
4
5      // hit
6      for (int i = 0; i < E; i++) {
7          if (cache[set][i].tag == tag) {
8              cache[set][i].time = 0;
9              hits++;
10             if (verbose) {
11                 printf("%c %lx,%d hit\n", operation, address, size);
12             }
13             return;
14         }
15     }
16
17     // miss
18     for (int i = 0; i < E; i++) {
19         if (cache[set][i].valid == 0) {
20             cache[set][i].valid = 1;
21             cache[set][i].tag = tag;
22             cache[set][i].time = 0;
23             misses++;
24             if (verbose) {
25                 printf("%c %lx,%d miss\n", operation, address, size);
26             }
27             return;
28         }
29     }
30
```

```
31      // miss eviction
32      evictions++;
33      misses++;
34      int max_time = -1;
35      int max_time_index = -1;
36      for (int i = 0; i < E; i++) {
37          if (cache[set][i].time > max_time) {
38              max_time = cache[set][i].time;
39              max_time_index = i;
40          }
41      }
42      cache[set][max_time_index].tag = tag;
43      cache[set][max_time_index].time = 0;
44      if (verbose) {
45          printf("%c %lx,%d miss eviction\n", operation, address, size);
46      }
47  }
```

这样，我们便完成了模拟高速缓存的全部过程。

2.1.2 Part B: Optimizing Matrix Transpose

在 B 部分中，您将在 trans.c 中编写一个转置函数，该函数会导致尽可能少的缓存未命中。

1）32×32 在本部分，我们需要转置一个 32×32 的矩阵。

因为一行有 32 个 bytes，也就是能一次保存 8 个 int，我们可以将矩阵拆分成多个 8×8 分块矩阵来进行转置。那么可以得到代码如下：

```
1  for (int i = 0; i < 32; i += 8) {
2      for (int j = 0; j < 32; j += 8) {
3          for (int k = i; k < (i + 8); k++) {
4              int temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
5              temp1 = A[k][j];
6              temp2 = A[k][j + 1];
7              temp3 = A[k][j + 2];
8              temp4 = A[k][j + 3];
9              temp5 = A[k][j + 4];
10             temp6 = A[k][j + 5];
11             temp7 = A[k][j + 6];
12             temp8 = A[k][j + 7];
13             B[j][k] = temp1;
14             B[j + 1][k] = temp2;
15             B[j + 2][k] = temp3;
```

```
            B[j + 3][k] = temp4;
            B[j + 4][k] = temp5;
            B[j + 5][k] = temp6;
            B[j + 6][k] = temp7;
            B[j + 7][k] = temp8;
        }
    }
}
```

2） 64×64　在本部分，我们需要转置一个 64×64 的矩阵。

我们可以通过将矩阵先进行 8 划分，再进行 4 划分，由于我们通过 4 划分后的分块矩阵进行多次转置操作，实现整个矩阵的转置。

首先，我们将每个 8×8 的大块中上面的两个 4×4 小块进行内部转置。然后，还需要将下面两个小块进行内部转置。最后，需要对 8×8 的大块中左下角和右上角两个块进行转置。代码如下：

```
for (int i = 0; i < N; i += 8) {
    for (int j = 0; j < M; j += 8) {
        int temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
        for (int k = i; k < i + 4; k++) {
            temp1 = A[k][j];
            temp2 = A[k][j + 1];
            temp3 = A[k][j + 2];
            temp4 = A[k][j + 3];
            temp5 = A[k][j + 4];
            temp6 = A[k][j + 5];
            temp7 = A[k][j + 6];
            temp8 = A[k][j + 7];

            B[j][k] = temp1;
            B[j + 1][k] = temp2;
            B[j + 2][k] = temp3;
            B[j + 3][k] = temp4;
            B[j][k + 4] = temp5;
            B[j + 1][k + 4] = temp6;
            B[j + 2][k + 4] = temp7;
            B[j + 3][k + 4] = temp8;
        }
        for (int k = j; k < j + 4; k++) {
            temp1 = A[i + 4][k];
            temp2 = A[i + 5][k];
            temp3 = A[i + 6][k];
            temp4 = A[i + 7][k];
```

```
28              temp5 = B[k][i + 4];
29              temp6 = B[k][i + 5];
30              temp7 = B[k][i + 6];
31              temp8 = B[k][i + 7];
32
33              B[k][i + 4] = temp1;
34              B[k][i + 5] = temp2;
35              B[k][i + 6] = temp3;
36              B[k][i + 7] = temp4;
37              B[k + 4][i] = temp5;
38              B[k + 4][i + 1] = temp6;
39              B[k + 4][i + 2] = temp7;
40              B[k + 4][i + 3] = temp8;
41          }
42          for (int k = i + 4; k < i + 8; k++) {
43              temp1 = A[k][j + 4];
44              temp2 = A[k][j + 5];
45              temp3 = A[k][j + 6];
46              temp4 = A[k][j + 7];
47
48              B[j + 4][k] = temp1;
49              B[j + 5][k] = temp2;
50              B[j + 6][k] = temp3;
51              B[j + 7][k] = temp4;
52          }
53      }
}54
```

1) 61×67　在本部分，我们需要转置一个 61×67 的矩阵。

考虑分成 4×17 的矩阵。代码如下：

```
1 for (int i = 0; i < 67; i += 17) {
2     for (int j = 0; j < 61; j += 4) {
3         for (int k = i; k < (i + 17 > 67 ? 67 : i + 17); k++) {
4             for (int l = j; l < (j + 4 > 61 ? 61 : j + 4); l++) {
5                 B[l][k] = A[k][l];
6             }
7         }
8     }
9 }
```

2.2 实验步骤

1) 解打包 cachelab-handout.tar

```
1 linux> tar -xvf cachelab-handout.tar
```

2) 阅读要求，编写 csim.c 和 trans.C

3) 编译

```
1 linux> make
```

4) 评测

```
1 linux> ./driver.py
```

3 实验过程与分析

实验的运行结果如下：

```
~/De/lab4/cachelab-handout ./driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1179
Trans perf 61x67          10.0        10        1848
          Total points    53.0        53
```

图 1: 运行结果

4 实验结果总结

在本次实验中，我学习到了高速缓存的基本结构，同时也学会了利用高速缓存的结构优化效率。