天津大学

计算机系统基础上机实验报告

实验题目 5：高速缓存 cache

学院名称 智能与计算学部

专　　业 计算机科学与技术（拔尖班）

学生姓名 牛天溳

学　　号 3024244288

年　　级 2024 级

班　　级 拔尖 1 班

时　　间 2025 年 6 月 28 日

实验 5：高速缓存

Cache

1. 实验目的

进一步理解高速缓存对于程序性能的影响。

2. 实验内容

这个实验包括以下两部分内容：你需要使用 C 语言编写一个小型程序（200-300 行）用来模拟高速缓存； 然后，对一个矩阵转置函数进行优化，以减少函数操作中的缓存未命中次数。详细内容请参考实验指导书：实验 5.pdf

No.	任务内容
1	任务 A：编写一个高速缓存模拟程序。在这部分任务中，你将在 csim.c 文件中编写一个高速缓存仿真程序。这个程序使用 valgrind 的内存跟踪记录 作为输入，模拟高速缓存的命中/未命中行为，然后输出总的命中次数，未命中次数和缓存块的替换次数。
2	任务 B：优化矩阵转置运算程序。在 trans.c 中编写一个矩阵转置函数，尽可能的减少程序对高速缓存访问的未命中次数。
3	

3. 实验要求

1） 在 Unbuntu18.04LTS 操作系统下，按照实验指导说明书，使用 gcc、make 和内存访问进行捕获和追踪的工具，完成本实验。

2） 本实验的具体要求：

a) 编译时不允许出现任何的 warning。

b) 转置函数中定义的 int 型局部变量总数不能超过 12 个。

c) 不允许使用 long 等数据类型，在一个变量中存储多个数组元素以减少内存访问。

d) 不允许使用递归。

e) 在程序中不能修改矩阵 A 中的内容，但是，你可以任意使用矩阵 B 中的空间，只要保证终的结果正确 即可。

f) 在函数中不能定义任何的数组，不能使用 malloc 分配额外的空间。

3） 需提交：csim.c 和 trans.c 源文件，电子版实验报告全文。

4. 实验结果

在实验 A 中，我们首先要想清楚如何表示一个高速缓存。最终，我采用了定义一个结构体作为一个高速缓存：

```
typedef struct cache_
{
    int S;
    int E;
    int B;
    Cache_line **line;
} Cache;
```

该结构体包含 S、E、B 等关键参数。Cache（缓存）的结构类似于二维数组，共分为 S = 2^s 个组（Set），每组包含 E 行（Line），每行存储的字节数固定。数组中的每个元素对应缓存中的一行，因此用 line 来表示这一存储单元。

接下来，我们又定义了一个结构体，包括了有效位 valid 和标记为 tag 两个参数。

由于该实验要求我们使用最近最少使用策略（LRU），因此我们还需要在该结构体里定义时间戳变量 timestamp，用于 LRU 的计算。时间戳越大，表示越久未访问。

```
typedef struct cache_
{
    int S;
    int E;
    int B;
    Cache_line **line;
} Cache;
```

接下来，我写了一个对 Cache 初始值设置的函数：

```
void Init_Cache(int s, int E, int b)
{
    int S = 1 << s;
    int B = 1 << b;
    cache = (Cache *)malloc(sizeof(Cache));
    cache->S = S;
    cache->E = E;
    cache->B = B;
    cache->line = (Cache_line **)malloc(sizeof(Cache_line *) * S);
    for (int i = 0; i < S; i++)
    {
        cache->line[i] = (Cache_line *)malloc(sizeof(Cache_line) * E);
        for (int j = 0; j < E; j++)
        {
            cache->line[i][j].valid = 0;
            cache->line[i][j].tag = -1;
            cache->line[i][j].timestamp = 0;
        }
    }
}
```

该函数首先根据组索引位数 s 和块偏移位数 b 计算出缓存的组数 S = 2^s 和块大小 B = 2^b，虽然块大小在模拟中并不直接使用，但它作为参数的一部分也被记录在结构中。接着，运用 malloc 函数，程序在堆上为整个缓存分配空间，构造一个包含 S 个组的结构体，每组包含 E 条缓存行。每条缓存行被表示为一个结构体，包含有效位（valid）、标记位（tag）和时间戳（timestamp）三部分信息。在初始化时，所有缓存行的有效位被设为无效（0），标记位设为默认值（通常为 -1，表示未分配地址），时间戳设为 0，用于后续实现 LRU 替换策略。通过这一系列初始化操作，整个缓存的二维结构被完整构建，确保每一组和每一行都准备好接受模拟过程中的访问、命中和替换操作。

```
void free_Cache()
{
    int S = cache->S;
    for (int i = 0; i < S; i++)
    {
        free(cache->line[i]);
    }
    free(cache->line);
    free(cache);
}
```

free_Cache 函数用于在程序结束时释放缓存结构体中动态分配的内存。函数首先获取组数 S，然后利用 free 函数，依次释放每一组中缓存行数组的内存，接着释放存放组指针的数组，最后释放整个缓存结构体本身。该函数确保程序在运行结束后不会造成内存泄漏，是缓存模拟中资源管理的必要步骤。

```
int get_index(int set_index, int tag)
{
    for (int i = 0; i < cache->E; i++)
    {
        if (cache->line[set_index][i].valid && cache->line[set_index][i].tag == tag)
            return i;
    }
    return -1;
}
```

get_index 函数用于在指定的组内查找是否存在与给定 tag 匹配的有效缓存行。函数遍历该组中的所有行，若某一行的 valid 位为 1 且 tag 匹配，则返回该行的索引，表示命中；否则返回 -1，表示该组中没有对应的数据。这一函数是判断缓存命中与否的核心逻辑。

```
int find_LRU(int set_index)
{
    int max_index = 0;
    int max_stamp = 0;
    for(int i = 0; i < cache->E; i++){
        if(cache->line[set_index][i].timestamp > max_stamp){
            max_stamp = cache->line[set_index][i].timestamp;
            max_index = i;
        }
    }
    return max_index;
}
```

find_LRU 函数用于在指定的组内找到最近最少使用（LRU）的缓存行。函数遍历该组中所有有效缓存行，比较它们的时间戳，选出时间戳最大（即最久未被访问）的那一行，并返回其索引。该函数在发生替换（eviction）时被调用，是实现 LRU 替换策略的关键步骤。

```
int is_full(int set_index)
{
    for (int i = 0; i < cache->E; i++)
    {
        if (cache->line[set_index][i].valid == 0)
            return i;
    }
    return -1;
}
```

is_full 函数用于判断指定组内是否还有空闲的缓存行。函数遍历该组的所有行，若发现某一行的 valid 位为 0，说明该行尚未被使用，立即返回其索引；若所有行都已被占用，则返回 -1，表示该组已满。该函数在缓存未命中时用于判断是否需要执行替换操作。

```
void update(int i, int set_index, int tag){
    cache->line[set_index][i].valid=1;
    cache->line[set_index][i].tag = tag;
    for(int k = 0; k < cache->E; k++)
        if(cache->line[set_index][k].valid==1)
            cache->line[set_index][k].timestamp++;
    cache->line[set_index][i].timestamp = 0;
}
```

接下来，我写了一个 update 函数，用于更新指定组中某条缓存行的信息。当缓存行被新数据占用或命中时，函数将该行的有效位设为有效（1），并更新其标签为当前访问的 tag。为了实现 LRU 替换策略，函数还会遍历该组所有有效行，将它们的时间戳统一加一，表示它们变得“更旧”，而当前被访问的缓存行时间戳重置为 0，表示这是最新访问的缓存行。这样通过时间戳管理访问顺序，辅助后续的替换判断。

在 update 函数下面，就是该程序的核心部分（我认为的），我写了一个 update_info 函数，作用是模拟一次缓存访问，判断当前访问的地址在缓存中是命中、未命中，还是需要替换。它先通过 get_index 查找当前组里有没有与 tag 相同的缓存行。如果找到了，就是命中，命中次数加一，同时调用 update 更新时间戳。如果没找到，就是未命中，未命中次数加一。如果这时候组里还有空位（用 is_full 检查），就把新数据放进空位；如果组已经满了，就说明需要替换，把最久没用的那一行（用 find_LRU 找出）替换掉，同时替换次数加一。最后，无论是命中、填空位还是替换，都会调用 update 来更新该缓存行的 tag 和时间戳。如果启用了 verbose 模式，还会在屏幕上打印 "hit"、"miss" 或 "eviction" 来展示过程。具体代码如下：

```
void update_info(int tag, int set_index)
{
    int index = get_index(set_index, tag);
    if (index == -1)
    {
        miss_count++;
        if (verbose)
            printf("miss ");
        int i = is_full(set_index);
        if(i==-1){
            eviction_count++;
            if(verbose) printf("eviction");
            i = find_LRU(set_index);
        }
        update(i,set_index,tag);
    }
    else{
        hit_count++;
        if(verbose)
            printf("hit");
        update(index,set_index,tag);
    }
}
```

至此，该程序的主体部分大致就是这样。接下来要处理一下文件部分的方面，模拟模拟器的运行。这一部分的核心是 get_trace 函数。

get_trace 函数的作用是读取并模拟整个内存访问过程。它首先打开用户指定的轨迹文件，一行一行地读取里面的访存操作信息，包括操作类型（如加载、存储、修改）、内存地址和访问的数据大小。每读取一条操作后，程序会把内存地址拆分成组号和标记，用来确定这次访问映射到缓存中的哪个位置。然后，根据操作类型判断是访问一次还是两次（比如修改操作会先读再写，相当于两次访问），并调用函数去判断这次访问是命中、未命中还是需要替换。同时，如果设置了详细模式，还会在终端输出相应的信息。这个函数会不断循环，直到所有操作都处理完，最后关闭文件。它是整个模拟器运行的核心，负责驱动每一次缓存访问并更新命中、未命中和替换的统计数据。

```
void get_trace(int s, int E, int b)
{
    FILE *pFile;
    pFile = fopen(t, "r");
    if (pFile == NULL)
    {
        exit(-1);
    }
    char identifier;
    unsigned address;
    int size;
    // Reading lines like " M 20,1" or "L 19,3"
    while (fscanf(pFile, " %c %x,%d", &identifier, &address, &size) > 0)
    {
        int tag = address >> (s + b);
        int set_index = (address >> b) & ((unsigned)(-1) >> (8 * sizeof(unsigned) - s));
        switch (identifier)
        {
        case 'M':
            update_info(tag, set_index);
            update_info(tag, set_index);
            break;
        case 'L':
            update_info(tag, set_index);
            break;
        case 'S':
            update_info(tag, set_index);
            break;
        }
    }
    fclose(pFile);
}
```

最后，再设计一下主函数：

```c
int main(int argc, char *argv[])
{
    char opt;
    int s, E, b;
    while (-1 != (opt = getopt(argc, argv, "hvs:E:b:t:")))
    {
        switch (opt)
        {
        case 'h':
            print_help();
            exit(0);
        case 'v':
            verbose = 1;
            break;
        case 's':
            s = atoi(optarg);
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
            b = atoi(optarg);
            break;
        case 't':
            strcpy(t, optarg);
            break;
        default:
            print_help();
            exit(-1);
        }
    }
    Init_Cache(s, E, b);
    get_trace(s, E, b);
    free_Cache();
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}
```

主函数主要负责程序的整体流程控制和各模块的调用。首先通过 getopt 函数解析命令行参数，提取用户输入的缓存配置，包括组索引位数 s、每组缓存行数 E、块偏移位数 b 和轨迹文件路径 t，同时判断是否启用了详细输出模式（-v 选项）。在参数获取完成后，程序调用 Init_Cache 初始化缓存结构，为后续模拟做好准备。接着通过 get_trace 函数读取并处理轨迹文件中的访存操作，逐条模拟缓存的访问情况。模拟完成后，调用 free_Cache 释放动态分配的内存，最后通过 printSummary 输出缓存的命中、未命中和替换统计信息，完成整个模拟流程，确保了程序从输入到输出的完整闭环。

测试一下：

```
root@ToddyN:~/lab5# ./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
```

通过！

在实验 B 中，实验要求通过程序完成矩阵的转置，并在模拟 1KB 直接映射缓存、32 字节块大小 的条件下，减少缓存未命中（miss）次数。实验要求尽可能减少缓存未命中次数。

在对 32×32 矩阵进行转置操作时，若采用最直观的双重循环方式，即按行读取 A[i][j] 并将其赋值到 B[j][i]，虽然读取 A 时是顺序访问，符合缓存的空间局部性，但写入 B 时却是按列写入，内存地址跳跃较大，容易频繁造成缓存未命中，特别是在直接映射缓存结构下。这种情况下，为了提升缓存命中率，必须设计一种更加“缓存友好”的算法。

```
void transpose_32x32(int M, int N, int A[N][M], int B[M][N])
{
    for (int i = 0; i < 32; i += 8)
    {
        for (int j = 0; j < 32; j += 8)
        {
            for (int k = i; k < i + 8; k++)
            {
                int a_0 = A[k][j];
                int a_1 = A[k][j + 1];
                int a_2 = A[k][j + 2];
                int a_3 = A[k][j + 3];
                int a_4 = A[k][j + 4];
                int a_5 = A[k][j + 5];
                int a_6 = A[k][j + 6];
                int a_7 = A[k][j + 7];
                B[j][k] = a_0;
                B[j + 1][k] = a_1;
                B[j + 2][k] = a_2;
                B[j + 3][k] = a_3;
                B[j + 4][k] = a_4;
                B[j + 5][k] = a_5;
                B[j + 6][k] = a_6;
                B[j + 7][k] = a_7;
            }
        }
    }
}
```

优化的关键在于“分块”技术。具体地，将原始矩阵按 8×8 的子块划分，每次仅处理一个小块内部的转置操作。由于一个 cache block 大小为 32 字节，而一个 int 为 4 字节，因此每个 block 能容纳正好 8 个 int，即 8 个连续的整数。因此处理 8×8 小块在缓存中非常合适，可以有效利用一个 cache block 存储整行或整列的数据。

实现过程中，外层两个循环控制块的起点，即 i 和 j，以步长 8 遍历矩阵的行和列。内层循环则固定行 k，在当前 8×8 小块中顺序读取 A[k][j] 到 A[k][j+7] 的 8 个元素，并分别保存在临时变量 a_0 到 a_7 中。接着，将这 8 个变量写入转置后的 B 矩阵中对应位置，即写入 B[j][k]到 B[j+7][k]。由于这一过程先按行读取再按行写入，最大程度地利用了缓存的空间局部性，避免了频繁的跨行访问所带来的缓存冲突。

这种优化方式的本质在于：通过块内局部化访问和使用寄存器变量暂存数据，避免了原始写法中“读取顺序、写入跳跃”导致的大量缓存未命中，从而极大提升了缓存效率与程序性能。

而在对 64×64 矩阵进行转置时，如果还使用处理 32×32 矩阵的方法来处理，并不能有效减少缓存未命中。这是因为在 1KB 的直接映射缓存和 32 字节块大小的设定下，一个 int 占 4 个字节，一个 cache block 容纳 8 个 int，而 64×64 矩阵每行占 256 个字节，分布在多个 cache block 中。如果多个行/列映射到同一 cache 行，就会出现严重的缓存冲突未命中，降低程序性能。

```
void transpose_64x64(int M, int N, int A[N][M], int B[M][N])
{
    int a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7;
    for (int i = 0; i < 64; i += 8)
    {
        for (int j = 0; j < 64; j += 8)
        {
            for (int k = i; k < i + 4; k++)
            {
                a_0 = A[k][j];
                a_1 = A[k][j + 1];
                a_2 = A[k][j + 2];
                a_3 = A[k][j + 3];
                a_4 = A[k][j + 4];
                a_5 = A[k][j + 5];
                a_6 = A[k][j + 6];
                a_7 = A[k][j + 7];
                B[j + 0][k] = a_0;
                B[j + 1][k] = a_1;
                B[j + 2][k] = a_2;
                B[j + 3][k] = a_3;
                B[j + 0][k + 4] = a_4;
                B[j + 1][k + 4] = a_5;
                B[j + 2][k + 4] = a_6;
                B[j + 3][k + 4] = a_7;
            }
            for (int k = j; k < j + 4; k++)
            {
                a_0 = B[k][i + 4];
                a_1 = B[k][i + 5];
                a_2 = B[k][i + 6];
                a_3 = B[k][i + 7];

                a_4 = A[i + 4][k];
                a_5 = A[i + 5][k];
                a_6 = A[i + 6][k];
                a_7 = A[i + 7][k];

                B[k][i + 4] = a_4;
                B[k][i + 5] = a_5;
                B[k][i + 6] = a_6;
                B[k][i + 7] = a_7;

                B[k + 4][i + 0] = a_0;
                B[k + 4][i + 1] = a_1;
                B[k + 4][i + 2] = a_2;
                B[k + 4][i + 3] = a_3;
            }
            for (int k = i + 4; k < i + 8; k++)
            {
                a_4 = A[k][j + 4];
                a_5 = A[k][j + 5];
                a_6 = A[k][j + 6];
                a_7 = A[k][j + 7];

                B[j + 4][k] = a_4;
                B[j + 5][k] = a_5;
                B[j + 6][k] = a_6;
                B[j + 7][k] = a_7;
            }

        }
    }
}
```

为了解决这一问题，该函数对每个 8×8 小块进行了更精细的处理。整个算法将 8×8 块再划分为 4×4 的四个子块，并将转置过程拆解为三个阶段，合理调度读写顺序，以规避缓存冲突。首先处理 A 块左上角的前 4 行，将每行前 4 个元素直接转置写入 B 的左上角，后 4 个元素则暂存至 B 的左下角（并非最终目标

位置），这是为了暂时避开直接映射缓存中可能的冲突。随后，在第二阶段，对 B 中暂存的数据与 A 的右上角数据进行交叉处理：一方面，将 A 的右上角数据转置写入 B 的右上角，另一方面，把 B 中左下角的暂存数据“交换”到其真正应在的右下角位置。最后处理 A 块右下角的 4 行 4 列区域，此时由于之前的调度已经避开冲突，可直接完成转置写入。

对于 61×67 的矩阵，实验的要求比较宽松，未命中次数小于 2000 次即可得到满分。

我们尝试用处理 32×32 矩阵的策略来处理该矩阵，具体代码如下：

```
void transpose_61x67(int M, int N, int A[N][M], int B[M][N])
{
    for (int i = 0; i < N; i += 16)
    {
        for (int j = 0; j < M; j += 16)
        {
            for (int k = i; k < i + 16 && k < N; k++)
            {
                for (int s = j; s < j + 16 && s < M; s++)
                {
                    B[s][k] = A[k][s];
                }
            }
        }
    }
}
```

最后，我们测试一下：

```
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        1227
Trans perf 61x67          10.0        10        1992
Total points              53.0        53
```

通过！（最后一个函数卡线过……）

5. 实验总结及心得体会

这次的实验难度可以说是出乎我的预料。本以为 lab3 和 lab4 已经是整个课程中最具挑战性的部分，没想到 lab5 又带来了一次“惊喜”——不仅在思维深度上提出了更高要求，也极大考验了我的代码实现能力。为了顺利完成任务，我查阅了大量资料，也参考了部分优秀的思路，最终逐步攻克了这一实验。

总体而言，这次 lab 的收获非常大。一方面，我在编码技巧和调试能力上得到了明显提升；另一方面，我对高速缓存的工作机制有了更加深入的理解。最让我感到震撼的是：在时间复杂度相同的前提下，仅仅通过优化访问模式，程序在性能上竟然可以表现出如此显著的差异——这让我第一次真正体会到了“写出高性能代码”的意义。