函钩 2025.6.9 CacheLab

一、实验目的

通过实现一个缓存模拟器，深入理解 CPU 缓存的工作原理，包括地址映射、缓存命中判断、替换策略等核心机制，掌握组相联缓存的组织结构和 LRU 替换策略的实现。

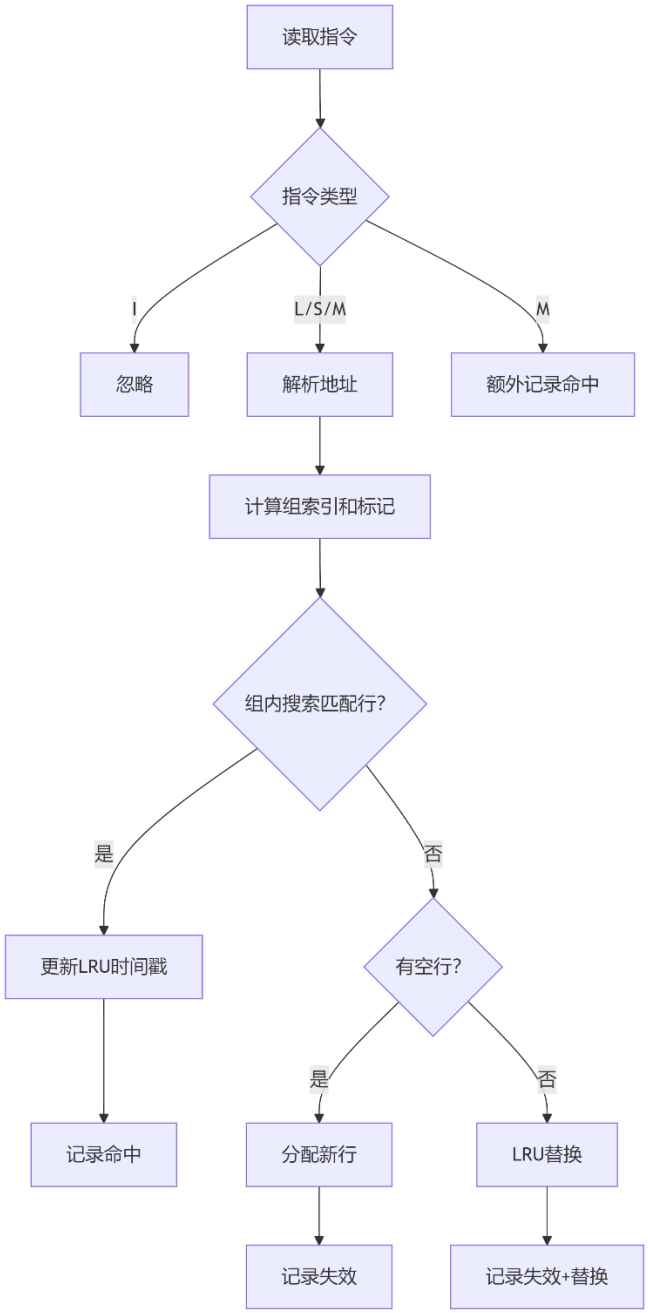
二、实验流程

附带全部代码：

```c
#include "cachelab.h"
#include <stdio.h>
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int s,E,b;
int v=0;
int hit=0,miss=0,eviction=0;
char t[100];
int time=1;
void help(){
    printf("Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>\n"
           "Options:\n"
           "  -h         Print this help message.\n"
           "  -v         Optional verbose flag.\n"
           "  -s <num>   Number of set index bits.\n"
           "  -E <num>   Number of lines per set.\n"
           "  -b <num>   Number of block offset bits.\n"
           "  -t <file>  Trace file.\n"
           "\n"
           "Examples:\n"
           "  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace\n"
           "  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace\n"
       );
    return;
}
typedef struct{
    int valid;//用于判断缓存块中是否有数据
    unsigned long tag;
    unsigned long data;
    int timestamp;//时间戳
}Cache;
int power(int n){
    int sum=1;
    for(int a=0;a<n;a++){
        sum*=2;
    }
    return sum;
}
void access1(char line[],Cache cache[][E]){
    if(line[0]=='I')return;
    unsigned long addr;
    char *p = line;
    // 动态定位地址起始位置
    while (*p && *p == ' ') p++;  // 跳过前导空格
    if (*p == 'I') return;        // 再次检查'I'
    p++;                          // 跳过操作符(L/S/M)
    while (*p && *p == ' ') p++;  // 跳过操作符后空格

    // 使用strtoul解析十六进制地址（参考源.cpp）
    char *endptr;
    addr = strtoul(p, &endptr, 16);
    if (endptr == p) {
        if(v==1) fprintf(stderr, "Address parse error: %s\n", line);
        return; // 解析失败时跳过
    }

    // 计算组索引和标记位（使用unsigned long防止截断）
    int set_index = (addr >> b) & ((1 << s) - 1);
    unsigned long set_tag = addr >> (b + s);  // 修改类型为unsigned long

    for(int x=0;x<E;x++){
        if(cache[set_index][x].tag==set_tag&&cache[set_index][x].valid==1){
            hit++;
            cache[set_index][x].timestamp=time;
            if(v==1)printf(" hit");
            return;
        }
    }
    miss++;
    for(int x=0;x<E;x++){
        if(!cache[set_index][x].valid){
            cache[set_index][x].tag=set_tag;
            cache[set_index][x].valid=1;
            cache[set_index][x].timestamp=time;
            if(v==1)printf(" miss");
            return;
        }
    }
    eviction++;
    int y=0,yy=0;
    y=cache[set_index][0].timestamp;
    for(int x=0;x<E;x++){
        if(y>=cache[set_index][x].timestamp){
            y=cache[set_index][x].timestamp;
            yy=x;
        }
    }//LRU
    cache[set_index][yy].tag=set_tag;
    cache[set_index][yy].timestamp=time;
    if(v==1)printf(" miss eviction");
    return;
}
void func(){
```

```c
void func(){
    Cache cache[power(s)][E];
    for(int d=0;d<power(s);d++){
        for(int c=0;c<E;c++){
            cache[d][c].valid=0;
            cache[d][c].tag=0;
            cache[d][c].data=0;
            cache[d][c].timestamp=0;
        }
    }

    FILE *f=fopen(t,"r");
    char line[100];
    if (f!=NULL) {
        while(fgets(line,100,f)!= NULL) {
            int len=strlen(line);
            line[len-1]='\0';
            if(v==1&&line[0]!='I')printf("%s ",line);
            time++;
            access1(line,cache);
            if(line[1]=='M'){
                hit++;
                if(v==1)printf(" hit");
            }
            if(v==1&&line[0]!='I')printf("\n");
        }
        fclose(f);
    }

}
int main(int argc,char *argv[])
{   int o;
    const char *optstring="hvs:E:b:t:";
    while((o=getopt(argc,argv,optstring))!=-1){
        switch(o){
            case 'h':
                help();
                break;
            case 'v':
                v=1;//v为1则输出详细过程
                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;
            case '?':
                help();
                break;
        }
    }
    func();
    printSummary(hit, miss, eviction);
    return 0;
}
```

缓存结构：
采用组相联映射（Set-Associative）
缓存大小由参数决定：S = 2^s（组数），E（每组行数），B = 2^b（块大小）
内存地址划分为三部分：
标记位（Tag）：高位比特，用于标识内存块
组索引（Set Index）：中间 s 比特，选择组
块偏移（Block Offset）：低位 b 比特，定位块内字节



首先是定义全局变量

```
int s,E,b;//存终端命令的参数
```

```
int v=0;//v 的值为命令行传入的参数，用于确定是否输出每次访问缓存的情况，v=1 输出，v=0 不输出，该值在检测中实际上没用用到，但是为了尽可能还原 csim-ref 程序，在此处定义了。
int hit=0,miss=0,eviction=0;//用于输出结果，命中，未命中，替换
char t[100];//存终端命令的文件
int time=1;//总的时间流动。后面 LRU（最近最少使用）会用
```

- **s, E, b**: 缓存参数，分别表示组索引位数、每组行数、块偏移位数
- **v**: 详细模式标志，控制是否输出每次访问的详细信息
- **hit, miss, eviction**: 统计命中、未命中和替换的次数
- **t**: 存储跟踪文件路径
- time: 全局时间戳，用于实现 LRU 替换策略

然后根据还原的要求写出 help 函数，实际上也没有用到，只要用 printf 把 csim-ref 输出的内容复制进去就好。

最重要的是定义 Cache 的结构体。使用二维数组 Cache cache[S][E]模拟缓存（S=2^s）

```
typedef struct{
    int valid;//用于判断缓存块中是否有数据
    unsigned long tag;//标记位
    int timestamp;//时间戳
}Cache;
```

根据 README.md 文件中的指导，我们可以使用<unistd.std>中的 getopt 函数对控制台传入的参数进行读取。然后在 main 函数中对控制台命令行的读取进行设置，如下所示（参数 h 或者 v 实际上在测试用例中没有用到）：

```
int main(int argc,char *argv[])
{   int o;
    const char *optstring="hvs:E:b:t:";
    while((o=getopt(argc,argv,optstring))!=-1){
        switch(o){
            case 'h':
                help();
                break;
            case 'v':
                v=1;//v 为 1 则输出详细过程
                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;
            case '?':
                help();
                break;
        }
    }
    func();
    printSummary(hit, miss, eviction);
    return 0;
}
```

optarg 是一个全局变量，它的类型是 char *。它用于存储当前选项（由 getopt 函数解析出的选项）对应的参数值，然后用于解析控制台输入的指令。所以此时要用 atoi 将字符转换为 int 类型。

```
void func(){
    Cache cache[power(s)][E];//power(s)表明 cache 的组数为 2 的 s 次方
    for(int d=0;d<power(s);d++){/*初始化 cache，将所有 valid 设置为 0，表示
空状态*/
        for(int c=0;c<E;c++){
            cache[d][c].valid=0;
            cache[d][c].tag=0;

            cache[d][c].timestamp=0;
        }
    }

    FILE *f=fopen(t,"r");
    char line[100];
    if (f!=NULL) {
        while(fgets(line,100,f)!= NULL) {
            int len=strlen(line);
            line[len-1]='\0';
            if(v==1&&line[0]!='I')printf("%s ",line);
            time++;
            access1(line,cache);//访问处理函数
            if(line[1]=='M'){
                hit++;
                if(v==1)printf(" hit");
            }
            if(v==1&&line[0]!='I')printf("\n");
        }
        fclose(f);
    }
```

```
}
```

在 main 函数读取命令行之后，调用了 func 函数，func 函数是缓存模拟器的核心执行引擎，负责：

1.初始化缓存数据结构

2.解析跟踪文件（trace file）

3.模拟所有内存访问操作

4.实现缓存替换策略（LRU）

5.统计命中/失效/替换次数

在 func 函数的指令处理循环中，不同指令作用如下：

I 指令：直接跳过（指令加载不涉及数据缓存）

L/S 指令：通过 access1()处理

M 指令：修改操作 = 加载(L) + 存储(S)

加载可能未命中 → 通过 access1()处理

存储必然命中 → 额外增加 hit 计数

特殊处理 M 指令的原理

```
if(line[1]=='M'){
                hit++;
                if(v==1)printf(" hit");
            }
```

内存修改操作包含两个步骤：1.从内存加载数据（可能未命中）。2.将修改后数据写回（必然命中刚加载的数据）。因此需要额外增加一次命中计数。

Func 函数的核心逻辑是通过全局变量 time 作为逻辑时钟，每次缓存访问不管是命中、分配还是替换都增加时间。并且在函数中调用 access1 函数。Func 函数设计过程中使用了 LRU 策略，替换组中时间戳最小的行。

```
void access1(char line[],Cache cache[][E]){
    if(line[0]=='I')return;
    unsigned long addr;
    char *p = line;
    // 动态定位地址起始位置
    while (*p && *p == ' ') p++;  // 跳过前导空格
    if (*p == 'I') return;        // 再次检查'I'
    p++;                          // 跳过操作符(L/S/M)
    while (*p && *p == ' ') p++;  // 跳过操作符后空格

    // 使用 strtoul 解析十六进制地址
    char *endptr;
    addr = strtoul(p, &endptr, 16);
    if (endptr == p) {
        if(v==1) fprintf(stderr, "Address parse error: %s\n", line);
        return; // 解析失败时跳过
    }
```

```
    // 计算组索引和标记位（使用 unsigned long 防止截断）
    int set_index = (addr >> b) & ((1 << s) - 1);
    unsigned long set_tag = addr >> (b + s);  // 修改类型为 unsigned long

    for(int x=0;x<E;x++){
        if(cache[set_index][x].tag==set_tag&&cache[set_index][x].valid==
1){
            hit++;
            cache[set_index][x].timestamp=time;
            if(v==1)printf(" hit");
            return;
        }
    }
    miss++;
    for(int x=0;x<E;x++){
        if(!cache[set_index][x].valid){
            cache[set_index][x].tag=set_tag;
            cache[set_index][x].valid=1;
            cache[set_index][x].timestamp=time;
            if(v==1)printf(" miss");
            return;
        }
    }
    eviction++;
    int y=0,yy=0;
    y=cache[set_index][0].timestamp;
    for(int x=0;x<E;x++){
        if(y>=cache[set_index][x].timestamp){
            y=cache[set_index][x].timestamp;
            yy=x;
        }
    }//LRU
    cache[set_index][yy].tag=set_tag;
    cache[set_index][yy].timestamp=time;
    if(v==1)printf(" miss eviction");
    return;
}
```

access1 函数是缓存模拟程序的核心访问逻辑，负责：

1. 解析内存访问指令
2. 计算地址映射关系（标记位 + 组索引）
3. 实现缓存命中判断
4. 处理未命中的两种情况：
    - 冷启动失效（分配空行）

- 冲突失效（LRU 替换）

5. 更新缓存状态和时间戳

```
// 计算组索引和标记位（使用 unsigned long 防止截断）
   int set_index = (addr >> b) & ((1 << s) - 1);
   unsigned long set_tag = addr >> (b + s);  // 修改类型为 unsigned
long
```

位操作原理：

组索引：提取地址中间 s 位

```
(addr >> b) & ((1 << s) - 1)
```

标记位：提取地址高位

```
addr >> (b + s)
```

示例：

设 s=2, b=3, 地址 0x19A (二进制 1100 11 010)

块偏移 = 低 3 位 010 (2)

组索引 = 中间 2 位 10 (组 2)

标记位 = 高 5 位 11001 (25)

对于命中和未命中的判断逻辑和处理方式如下：

```
    for(int x=0;x<E;x++){
        if(cache[set_index][x].tag==set_tag&&cache[set_index][x].valid==
1){
            hit++;
            cache[set_index][x].timestamp=time;
            if(v==1)printf(" hit");
            return;
        }
    }
```

命中条件要同时满足 `valid==1` 以及 `tag` 的正确匹配，在判定为命中后更新命中数和时间戳。

```
miss++;
    for(int x=0;x<E;x++){
        if(!cache[set_index][x].valid){
            cache[set_index][x].tag=set_tag;
            cache[set_index][x].valid=1;
            cache[set_index][x].timestamp=time;
            if(v==1)printf(" miss");
            return;
        }
    }
    eviction++;
    int y=0,yy=0;
    y=cache[set_index][0].timestamp;
    for(int x=0;x<E;x++){
        if(y>=cache[set_index][x].timestamp){
```

```
            y=cache[set_index][x].timestamp;
            yy=x;
        }
    }//LRU
    cache[set_index][yy].tag=set_tag;//执行替换
    cache[set_index][yy].timestamp=time; //执行替换
    if(v==1)printf(" miss eviction");
```

未命中时有两种情况，首先是冷未命中，出现在首次访问时缓存为空，valid==0 的情况。此时需要将 valid 更新为 1 并且更新 tag 和时间戳。另一种情况为冲突未命中，当组内无空行时触发，此时需要对缓存进行替换（eviction）。题目要求使用 LRU 策略对已有的行进行替换：遍历组内的所有行，然后找到时间戳最小的行，即为最久没访问的行，可以将这一行进行替换。

以上就是实验第一部分的完成思路。第一部分主要由命令行的读取和 LRU 替换策略两部分组成，加深了我们对高速缓存运行逻辑的理解。

接下来是实验第二部分，对缓存友好的矩阵转置：

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]) {
    if(M==32&&N==32){
        for (int i = 0; i < M; i += 8) {
            for (int j = 0; j < N; j += 8) {
                for (int k = i; k < i + 8; k++)//在矩阵的每一个分块中进行复
制
                {
                    int a_0 = A[k][j];
                    int a_1 = A[k][j + 1];//你好
                    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;
                }
            }
```

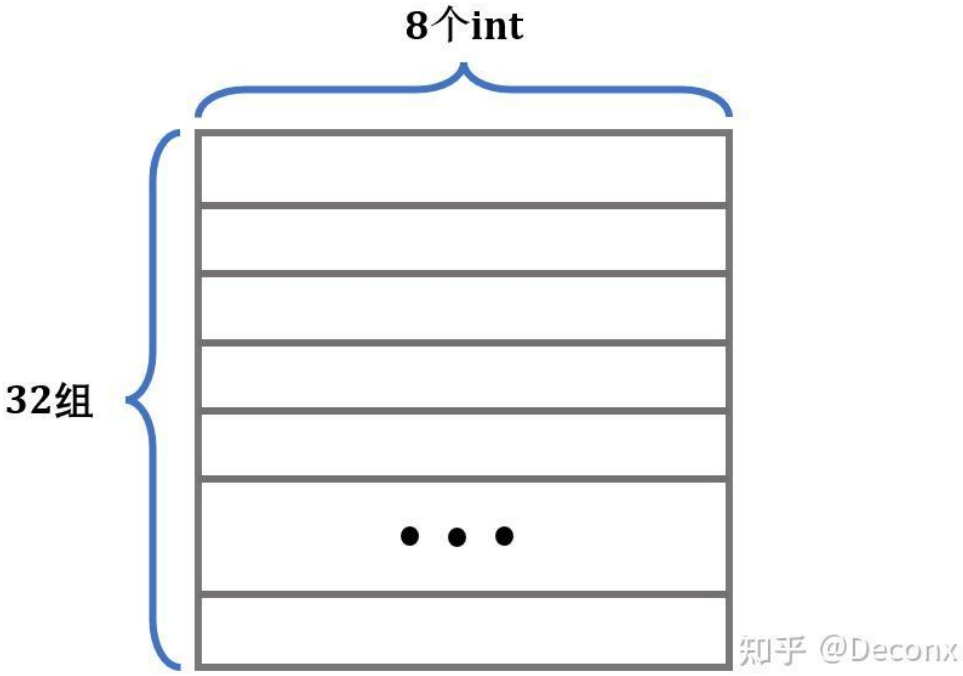
```
        }
    }else if(M==64&&N==64){
        int a0, a1, a2, a3, a4, a5, a6, a7;
    for (int i = 0; i < M; i += 8) {
        for (int j = 0; j < N; j += 8) {
            for (int k = i; k < i + 4; k++) {
                a0 = A[k][j + 0];//*将矩阵 A 的每一个 8*8 块分割成 4 个 4*4 的
小块，并使用变量存储第一行
                a1 = A[k][j + 1];//此时存储的是 4*4 块的左上和右上两个块的第
k 行，
                a2 = A[k][j + 2];
                a3 = A[k][j + 3];
                a4 = A[k][j + 4];
                a5 = A[k][j + 5];
                a6 = A[k][j + 6];
                a7 = A[k][j + 7];
                B[j + 0][k] = a0;//把方块复制到 B 的左上，按照列复制
                B[j + 1][k] = a1;
                B[j + 2][k] = a2;
                B[j + 3][k] = a3;
                B[j + 0][k + 4] = a4;//把方块复制到 B 的右上，按照列赋值
                B[j + 1][k + 4] = a5;
                B[j + 2][k + 4] = a6;
                B[j + 3][k + 4] = a7;
            }//第一部结束
            for (int k = j; k < j + 4; k++) {//*听到了 Redo，燃起来了*//
                //储存 B 的左上块
                a0 = B[k][i + 4];
                a1 = B[k][i + 5];
                a2 = B[k][i + 6];
                a3 = B[k][i + 7];
                //获取 A 的左下块
                a4 = A[i + 4][k];
                a5 = A[i + 5][k];
                a6 = A[i + 6][k];
                a7 = A[i + 7][k];
                //将 a 的左下块复制给 b 的右上块
                B[k][i + 4] = a4;
                B[k][i + 5] = a5;
                B[k][i + 6] = a6;
                B[k][i + 7] = a7;
                //将已经储存的 b 的右上块复制到 b 的左下块
                B[k + 4][i] = a0;
                B[k + 4][i + 1] = a1;
                B[k + 4][i + 2] = a2;
```

```
                B[k + 4][i + 3] = a3;
                //左下和右下复制结束
            }
            //复制右下角对角块
            for (int k = i + 4; k < i + 8; k++)
            {
                // 处理第 4 块
                a4 = A[k][j + 4];
                a5 = A[k][j + 5];
                a6 = A[k][j + 6];
                a7 = A[k][j + 7];
                B[j + 4][k] = a4;
                B[j + 5][k] = a5;
                B[j + 6][k] = a6;
                B[j + 7][k] = a7;
            }

        }
    }

    }
    else if(M==61&&N==67){
        for(int i=0;i<N;i+=17){
        for(int j=0;j<M;j+=17){
            for(int p=0;p<17&&i+p<N;p++){
                for(int q=0;q<17&&j+q<M;q++){
                    B[j+q][i+p]=A[i+p][j+q];
                }
            }
        }
    }
    }
}
```

代码中本应该是三个函数的，在测试程序中不知道为什么能够分别在三个函数测试出相应的正确答案，但是程序只会按照写在最前面的那个函数计分，被迫无奈，我把三个函数用 `if else` 搞成了一个函数。

计算机的高速缓存是有分块结构的，高速缓存访问同一块的速度更快。

例如 s = 5, E = 1, b = 5 的缓存有 32 组，每组一行，每行能够存 8 个 int。内存结构示意图

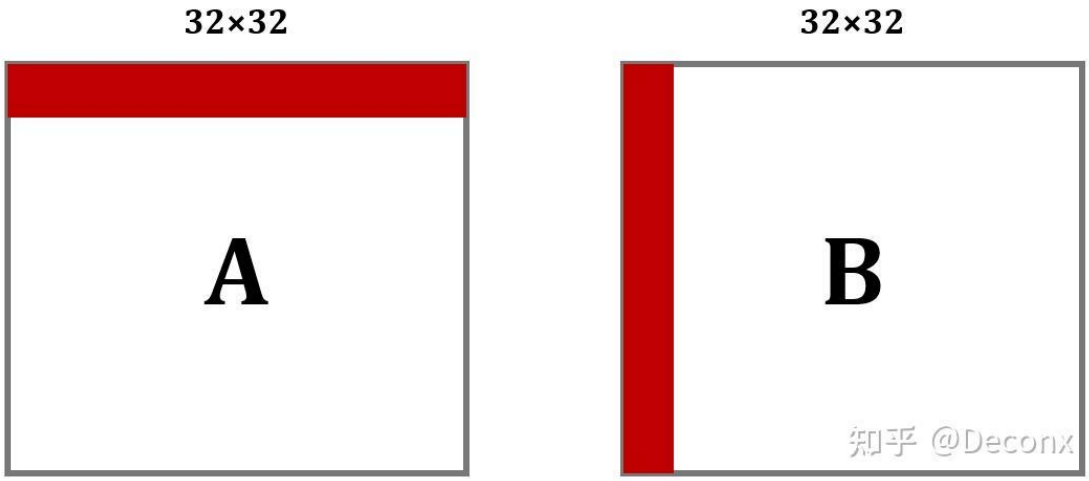


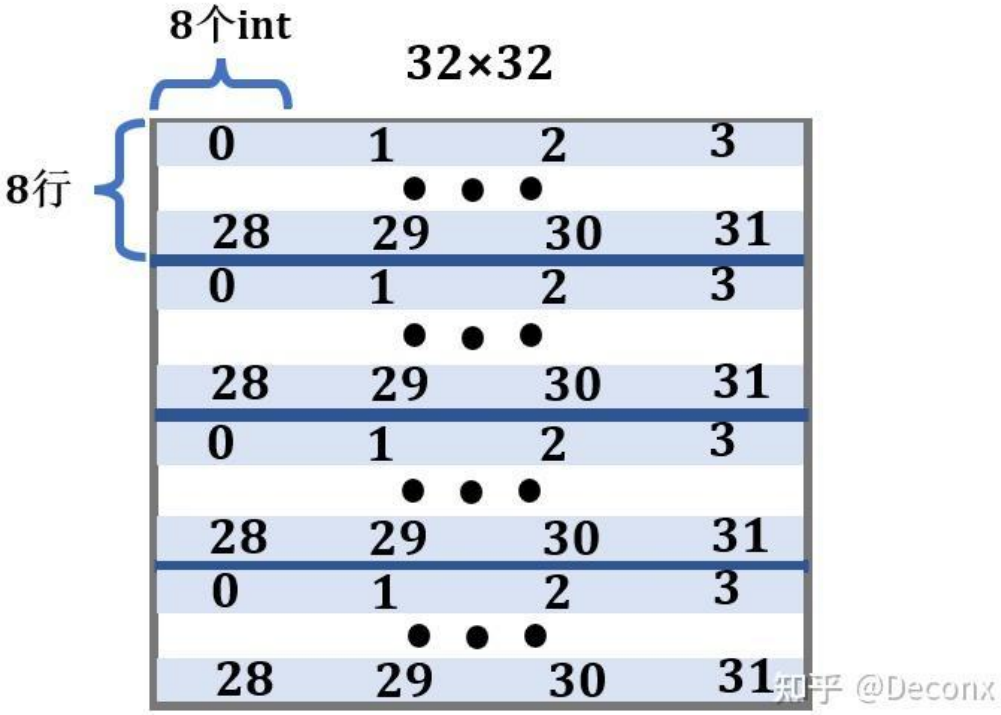

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, tmp;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
    }

}
```

以上函数是实验程序中附带的简单转置程序，他使用了最简单暴力的方法，直接将矩阵的每个值对应进行转置。虽然代码简单，但是它对缓存并不友好。

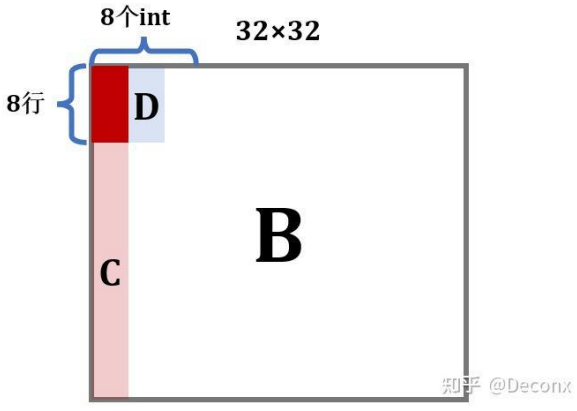
图片转载自 https://zhuanlan.zhihu.com/p/484657229

对于矩阵 A，我们会一行一行地读取，然后按列赋值到矩阵 B。当我们需要读取 A[0][0]时，从 A[0][0]到 A[0][7]的同一行的八个元素都会被加载到缓存中。同理，写入时 B 矩阵也是按行加载进缓存，但是简单转置函数是按照列写入的，列相邻的元素并不在同一个缓存块中，所以每给 B 写入一个元素就会产生一次未命中，而且当写完一列之后，本来加载过的元素可能已经被其他元素覆盖了，又要重新加载，严重降低了缓存运行的效率。



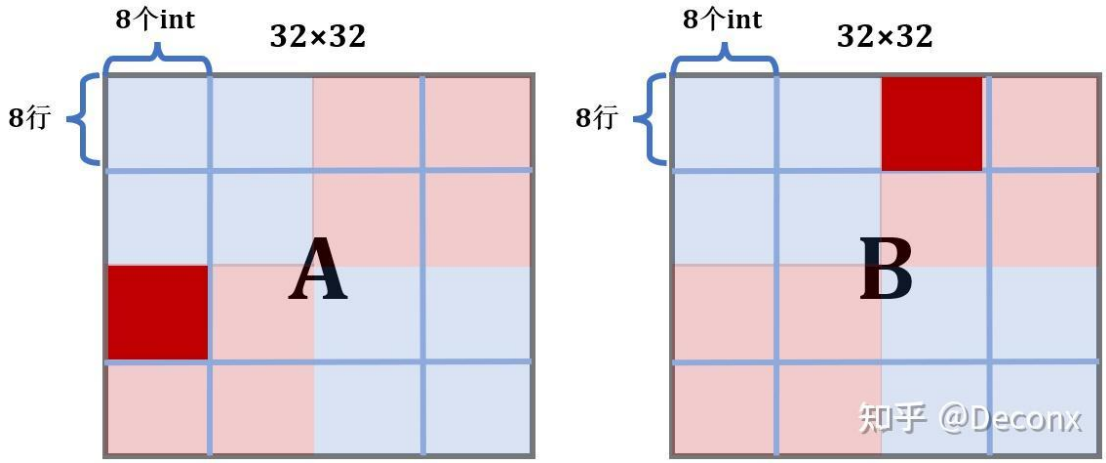


首先应对 32*32 的矩阵，简单转置算法肯定不能用，因为矩阵一行有 32 个元素，缓存块是装不下的。对于 A，每 8 个 int 就会占满缓存的一个组，A 的每行就会有 32/8=4 次不命中。对于 B，则每一列都会有 32 次不命中，计算得 4*32+32*32=1152 次不命中。加上对角线部分的冲突，不命中次数会更多，为 1152+32=1184。所以要考虑进行矩阵分块的算法。



在矩阵 B 的前 8 行数据写入完成后，其 D 区域已完整载入缓存。若能立即处理 D 区域，便可充分利用缓存内容避免未命中。然而暴力解法转而操作区域 C，每次元素写入都会驱逐现有缓存内容。当处理第 2 列 D 区域时，对应的缓存行极可能已被覆盖，导致再次发生未命中。这表明暴力解法的核心缺陷在于未能有效复用已加载的缓存数据。

分块技术正是针对同一矩阵内部缓存块相互覆盖问题提出的解决方案。基于上述由上述分析，显然应考虑 8 × 8 分块，这样在块的内部不会冲突，接下来判断 A 与 B 之间会不会冲突。



A 中标红的块占用的是缓存的第 0，4，8，12，16，20，24，28 组，而 B 中标红的块占用的是缓存的第 2，6，10，14，18，16，30 组，刚好不会冲突。而且我们还发现，除了对角线用一样的缓存块可能会冲突外，其他的都不会冲突。我们可以尝试写一个分块转置的代码：

```
void test_transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    for(int i=0;i<M;i+=8){//每 8*8 个元素为一组
        for(int j=0;j<N;j+=8){
            for(int p=0;p<8;p++){//分别在每个小块中进行转置
                for(int q=0;q<8;q++){
                    B[j+q][i+p]=A[i+p][j+q];
                }
            }
        }
    }
}
```

进行测试

```
Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (just a test): hits:1710, misses:343, evictions:311
```

结果是 misses 为 343，与正确答案仍然有差距。根据分析可以得到，一般情况下 A 和 B 使用不同的缓存块，不会产生冲突，但是矩阵转置后对角线的相对位置是不变的，所以两者会占用相同的缓存块，产生了冲突。

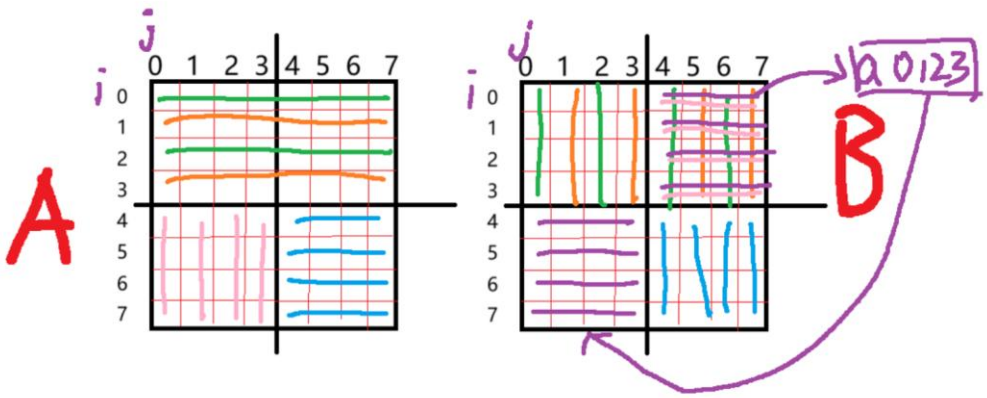
在 README.md 中作者提示：“You are allowed to define at most 12 local variables of type int per transpose function”说明我们还可以额外定义 12 个局部变量。所以我们可以额外定义 8 个变量，我们用 8 个变量把 a0~a7 储存起来，这样缓存块中的元素就得到了充分利用，之后把 8 个变量给 B。B 会用第 0，4，8，12，16，20，24，28 组缓存存储 8 个变量。

```
if(M==32&&N==32){
        for (int i = 0; i < M; i += 8) {
            for (int j = 0; j < N; j += 8) {
                for (int k = i; k < i + 8; k++)//在矩阵的每一个分块中进行复
制
                {
                    int a_0 = A[k][j];
                    int a_1 = A[k][j + 1];//你好
                    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;
                }
            }
        }
```

此时测试的结果是满分

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
```

第二题，64*64 矩阵的转置



若同样使用 8*8 分块矩阵，64 的一行就要用 8 个矩阵块，那么只能用 4 行矩阵块，缓存就会被占满。为了避免冲突，可以使用 4*4 的缓存块。我们可以将 8*8 分成 4 个 4*4 矩阵块。

如何将 8*8 的块拆分成四部分复制给 B 呢？根据上图的颜色，我们可以大致表示转换的顺序。首先是使用黄色和绿色的条，对 A 的行复制，并竖方向复制到 B，但是按照矩阵转置的规律，黄绿条有一块会被复制到 B 的右上，而它的正确位置应该是左下。所以可以先看紫色条（黄绿条），通过定义新变量的方式，将 B 的紫色条储存进变量。定义 4 个变量储存 B 紫色的一行，再定义另外 4 个变量储存 A 的粉色的一列，将左下的粉色复制到 B 的右上，与此同时再将 B 右上的紫色复制到左下。最后就是蓝色部分，使用变量简单转置即可。

代码：

```
else if(M==64&&N==64){
        int a0, a1, a2, a3, a4, a5, a6, a7;
    for (int i = 0; i < M; i += 8) {
        for (int j = 0; j < N; j += 8) {
            for (int k = i; k < i + 4; k++) {
                a0 = A[k][j + 0];//*将矩阵 A 的每一个 8*8 块分割成 4 个 4*4 的小块，并使用变量存储第一行
                a1 = A[k][j + 1];//此时存储的是 4*4 块的左上和右上两个块的第 k 行，
                a2 = A[k][j + 2];
                a3 = A[k][j + 3];
                a4 = A[k][j + 4];
                a5 = A[k][j + 5];
                a6 = A[k][j + 6];
                a7 = A[k][j + 7];
                B[j + 0][k] = a0;//把方块复制到 B 的左上，按照列复制
                B[j + 1][k] = a1;
                B[j + 2][k] = a2;
                B[j + 3][k] = a3;
                B[j + 0][k + 4] = a4;//把方块复制到 B 的右上，按照列赋值
                B[j + 1][k + 4] = a5;
                B[j + 2][k + 4] = a6;
                B[j + 3][k + 4] = a7;
            }//第一部结束
            for (int k = j; k < j + 4; k++) {//*听到了 Redo，燃起来了*//
                //储存 B 的左上块
                a0 = B[k][i + 4];
```

```
            a1 = B[k][i + 5];
            a2 = B[k][i + 6];
            a3 = B[k][i + 7];
            //获取 A 的左下块
            a4 = A[i + 4][k];
            a5 = A[i + 5][k];
            a6 = A[i + 6][k];
            a7 = A[i + 7][k];
            //将 a 的左下块复制给 b 的右上块
            B[k][i + 4] = a4;
            B[k][i + 5] = a5;
            B[k][i + 6] = a6;
            B[k][i + 7] = a7;
            //将已经储存的 b 的右上块复制到 b 的左下块
            B[k + 4][i] = a0;
            B[k + 4][i + 1] = a1;
            B[k + 4][i + 2] = a2;
            B[k + 4][i + 3] = a3;
            //左下和右下复制结束
        }
        //复制右下角对角块
        for (int k = i + 4; k < i + 8; k++)
        {
            // 处理第 4 块
            a4 = A[k][j + 4];
            a5 = A[k][j + 5];
            a6 = A[k][j + 6];
            a7 = A[k][j + 7];
            B[j + 4][k] = a4;
            B[j + 5][k] = a5;
            B[j + 6][k] = a6;
            B[j + 7][k] = a7;
        }

    }
}

}
```

测试结果：

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9018, misses:1227, evictions:1195
```

Misses 为 1227，满分。

第三题，61*67 矩阵的转置。这个矩阵不是对称的。题目要求的通过分数为 2000 以下。测试过几次，16*16 分块或者 17*17 的分块都能通过。

因为要求比较松，所以不需要十分优化。

代码如下：

```
else if(M==61&&N==67){
        for(int i=0;i<N;i+=16){
        for(int j=0;j<M;j+=16){
            for(int p=0;p<16&&i+p<N;p++){
                for(int q=0;q<16&&j+q<M;q++){
```

```
                    B[j+q][i+p]=A[i+p][j+q];
                }
            }
        }
    }
    }
```

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6187, misses:1992, evictions:1960
```

16分块

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6229, misses:1950, evictions:1918
```

17分块

似乎 **17*17** 的效果会稍好一些。

实验总结及心得体会

在实验的第一部分，我在探索的过程中能够更加深入的了解 LRU 策略的运作原理以及软件实现方式：通过对每个高速缓存的时间戳进行比对，选择最久没有使用过的缓存块进行替换，这样可以保证最新或地址最靠近的数据不会被替换。第一部分的命令行读取部分也是非常让人头疼的，但是通过实验我们也能够大体了解在 Linux 类系统的程序应该如何编写来通过控制台的命令行进行交互。

在实验的第二部分，矩阵的转置，让我能够更好地了解高速缓存的物理结构和读取写入方式，对我编写缓存友好程序有一定启发。

深入探索 Cache Lab 收获颇丰！