

CSAPP lab4 cachelab实验报告

实验日期: May.1-May.18

1实验解题思路

1.1 Part A: Writing a Cache Simulator

解题思路

1.读取并解析命令行参数

- 通过 `int main(int argc, char *argv[])` 读取命令行的参数数量(`argc`)和内容(`argv`)
- 利用 `getopt` 函数来解析命令行参数, 读取cache的基本参量 (`s,E,b`), 模式参数(`v`)和文件名。并利用 `switch case` 来给对应的全局变量赋值。
- `getopt` 每次调用会返回当前解析到的选项字符 (如 '`v`'、'`s`' 等) 到 `opt`, 如果没有更多选项可解析, 则返回 `-1`。对应代码如下:

```
void parseCommand(int argc, char* argv){
    int opt;
    while((opt=getopt(argc, argv, "vs:E:b:t:"))!=-1){
        switch (opt){
            case 'v':
                verbose=1;//verbose模式
                break;
            case 's':
                s=atoi(optarg);
                S=1<<s;
                break;
            case 'E':
                E=atoi(optarg);
                break;
            case 'b':
                b=atoi(optarg);
                break;
            case 't':
                strcpy(filename, optarg);
                break;
            default:
                break;
        }
    }
}
```

2.逐行读文件中的trace

- 通过 `FILE* fp=fopen(filename, "r")` 来读取文件; 通过 `fgets(buffer, 1000, fp)` 来以字符数组的形式读取每一行的trace; 通过 `n=sscanf(buffer, " %c %x,%d", &type, &address, &temp)` 来从每一行的字符数组中按指定格式提取 `type`, `address` 及 `temp` (`n` 为成功匹配的变量个数)。

- 输入的 trace 的格式为：[space]type address, size。size 在此题中**没有用处**；address 是**进行cache操作的参数**（决定在 cache 中是否命中）；操作的 type 有4种，决定**模拟cache需要进行操作的次数**。通过 switch case 进行选择。
 - I 表示加载指令（**不需要任何操作**）
 - L 加载数据（**一次访问cache操作**）
 - S 存储数据（**一次访问cache操作**）
 - M 修改数据（**两次访问cache操作**）

```
if(n<3){
    continue;
} //I加载指令的情况(不执行操作)
switch (type){
    case 'M':
        update(address); //刻意fall through 使M执行两次操作
    case 'L':
    case 'S':
        update(address); //L和S执行一次操作
        break;
}
```

3.定义并创建cache

- **定义set内的 cache_line 数据结构**。实际的 cache_line 分为 valid 位, tag 位, block 位，但在本题中 block 位没有用处，所以不需要。由于采用**LRU(最近最少使用)**的替换策略，所以需要 stamp 时间戳来记录每一行 cache_line 的数据距离上一次使用的时间。

```
typedef struct{
    int valid, tag, stamp; //stamp是时间戳 用于记录最近最少使用
}cache_line;
```

- 注：由于stamp时间戳是用于记录cache_line距离上一次使用的时间，所以注意在**每一行trace的操作后，所有非空的cache_line都要进行stamp的更新！**

```
void stamp_update(){
    for(int i=0; i<S; i++){
        for(int j=0; j<E; j++){
            if(cache[i][j].valid){ //所有非空的cache_line
                cache[i][j].stamp++;
            }
        }
    }
    return;
}
```

- 使用malloc动态分配cache所需空间并进行初始化。

```
cache_line** cache=NULL; //指向cache的指针 全局变量
void cachesimulate(char* filename){
    //开辟cache所需空间
    cache=(cache_line**)malloc(sizeof(cache_line*)*S); //为cache开辟S个set的空间
    for(int i=0; i<S; i++){
```

```

        cache[i]=(cache_line*)malloc(sizeof(cache_line)*E);
    }//为每一个set开辟E行cache_line的空间
    //对cache初始化
    for(int i=0;i<S;i++){
        for(int j=0;j<E;j++){
            cache[i][j].valid=0;
            cache[i][j].tag=cache[i][j].stamp=-1;
        }
    }
}

```

4.cache操作 (update函数, 用于判断是否hit/miss/eviction)

- 从 address 地址中提取用于**定位**的 set (定位组) 和 tag (定位行)。

```

unsigned set=(address>>b)&((1<<s)-1);
unsigned tag=address>>(s+b);

```

- 在对应的set中查找**是否命中**。若是tag匹配且valid位为1, 则命中, hit_count 加1, 同时更新时间戳stamp为0 (表示刚刚使用过); 否则未命中, miss_count 加1。
- miss的情况下, 再查找**是否需要替换**。若是存在空cache_line(即valid为0的cache_line), 则不需要替换, 将新的地址存入该行即可, 同时更新时间戳stamp为0 (表示刚刚使用过); 若是不存在空cache_line, 则需要替换, eviction_count 加1, 同时**查找最近最少使用的 (即时间戳最大的) 行进行替换, 并更新时间戳**。对应代码如下:

```

void update(unsigned address){
    unsigned set=(address>>b)&((1<<s)-1);
    unsigned tag=address>>(s+b);
    //命中的情况
    for(int i=0;i<E;i++){
        if(cache[set][i].tag==tag && cache[set][i].valid){//tag匹配且valid为1
            hit_count++;
            cache[set][i].stamp=0;//更新时间戳
            if(verbose) printf("hit "); //verbose模式要打印
            return;
        }
    }
    //未命中的情况
    miss_count++;
    if(verbose) printf("miss ");
    ///存在空line
    for(int i=0;i<E;i++){
        if(cache[set][i].valid==0){
            cache[set][i].valid=1;
            cache[set][i].tag=tag;
            cache[set][i].stamp=0;//更新时间戳
            return;
        }
    }
    ///没有空line 只能替换
    eviction_count++;
    if(verbose) printf("eviction ");
    int max_stamp_index=0;

```

```

        for(int i=1;i<E;i++){
            if(cache[set][i].stamp>cache[set][max_stamp_index].stamp){
                max_stamp_index=i;
            }
        }
        //找到最近最少使用（即搁置最久）
        cache[set][max_stamp_index].tag=tag;//进行替换
        cache[set][max_stamp_index].stamp=0;//更新时间戳
        return;
    }
}

```

5.整合（主函数）

- 总结下来，整体的步骤为：**解析命令行，读取trace，创建cache，更新cache（进行访问操作），释放内存，输出结果**。主函数代码如下：

```

int main(int argc,char *argv[]){
    parseCommand(argc,argv);//解析命令行
    cacheSimulate(filename);//模拟cache(包含读取trace，创建cache与更新cache)
    for(int i=0;i<S;i++){
        free(cache[i]);
    }
    free(cache);//释放内存
    printSummary(hit_count,miss_count,eviction_count);//输出结果
    return 0;
}

```

partA全部代码及注释：

```

#include "cachelab.h"
#include <getopt.h>//用于解析命令行参数(如optarg)
#include <stdlib.h>
#include <unistd.h>//getopt定义在这里
#include <string.h>
#include <stdio.h>

//定义一些需要用到的全局变量
static int verbose=0;
static int s;//set位的字节数
static int S;//总共有多少set
static int E;
static int b;//block位的字节数
static int hit_count=0;
static int miss_count=0;
static int eviction_count=0;
//定义cache_line的结构
typedef struct{
    int valid,tag,stamp;//stamp是时间戳 用于识别最近最少使用的cache_line
}cache_line;
cache_line** cache=NULL;//cache为全局变量
char filename[100];//输入的文件名

//解析命令行参数
void parseCommand(int argc,char* argv[]){
    int opt;
    while((opt=getopt(argc,argv,"vs:E:b:t:"))!=-1){

```

```

switch (opt){
    case 'v':
        verbose=1;//verbose模式
        break;
    case 's':
        s=atoi(optarg);
        S=1<<s;
        break;
    case 'E':
        E=atoi(optarg);
        break;
    case 'b':
        b=atoi(optarg);
        break;
    case 't':
        strcpy(filename,optarg);
        break;
    default:
        break;
}
}
}

```

//对cache进行操作(hit/miss/eviction)

```

void update(unsigned address){
    unsigned set=(address>>b)&((1<<s)-1);
    unsigned tag=address>>(s+b);
    for(int i=0;i<E;i++){
        if(cache[set][i].tag==tag && cache[set][i].valid){//tag匹配且valid为1才hit
            hit_count++;
            cache[set][i].stamp=0;//更新时间戳
            if(verbose) printf("hit ");//verbose模式要打印
            return;
        }
    }
    //hit
    miss_count++;//未hit都先miss
    if(verbose) printf("miss ");
    //接下来检查该组是否有空的cache_line
    for(int i=0;i<E;i++){
        if(cache[set][i].valid==0){
            cache[set][i].valid=1;
            cache[set][i].tag=tag;
            cache[set][i].stamp=0;
            return;
        }
    }
    //存在空line
    eviction_count++;//没有空line 只能eviction替换
    if(verbose) printf("eviction ");
    int max_stamp_index=0;
    for(int i=1;i<E;i++){
        if(cache[set][i].stamp>cache[set][max_stamp_index].stamp){
            max_stamp_index=i;
        }
    }
    //找到最近最少使用（即搁置最久）
    cache[set][max_stamp_index].tag=tag;
    cache[set][max_stamp_index].stamp=0;//进行替换
}

```

```

        return;
    }

    //时间戳更新
    void stamp_update(){
        for(int i=0;i<S;i++){
            for(int j=0;j<E;j++){
                if(cache[i][j].valid){//所有不为空的cache_line
                    cache[i][j].stamp++;//时间都增加
                }
            }
        }
        return;
    }

    //读文件 根据不同的操作进行模拟
    void cachesimulate(char* filename){
        //动态开辟cache的空间
        cache=(cache_line**)malloc(sizeof(cache_line*)*S);
        for(int i=0;i<S;i++){
            cache[i]=(cache_line*)malloc(sizeof(cache_line)*E);
        }
        //cache初始化
        for(int i=0;i<S;i++){
            for(int j=0;j<E;j++){
                cache[i][j].valid=0;
                cache[i][j].tag=cache[i][j].stamp=-1;
            }
        }
        //读文件
        FILE* fp=fopen(filename,"r");
        if(fp==NULL){
            printf("the file is wrong");
            exit(-1);//文件错误 立即终止当前程序
        }
        char buffer[1000];
        char type;//操作类型
        unsigned int address;//地址
        int temp;//block位 在本题中没用
        while(fgets(buffer,1000,fp)){//fgets成功时返回buffer,文件末尾返回null
            int n=sscanf(buffer," %c %x,%d",&type,&address,&temp);
            //从字符串中按指定格式提取数据 sscanf返回值为成功匹配的变量个数
            if(n<3){
                continue;
            }//处理type为'I'的情况
            if(verbose){
                printf("%c %x,%d ",type,address,temp);
            }//verbose模式 要打印
            switch (type)
            {
                case 'M':
                    update(address);//fall through 使M模式操作两次
                case 'L':
                case 'S':
                    update(address);
                    break;
            }
        }
    }
}

```

```

    } // M模式要操作两次 L/S操作一次
    if(verbose) printf("\n");
    stamp_update(); // 一次操作后时间戳update
}
fclose(fp);
return;
}

int main(int argc, char *argv[]){
    // 解析命令行
    parseCommand(argc, argv);
    // 模拟cache行为
    cacheSimulate(filename);
    // 释放内存
    for(int i=0; i<S; i++){
        free(cache[i]);
    }
    free(cache);
    // 打印结果
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}

```

1.2 Part B: Optimizing Matrix Transpose

题目给出的cache参数为 $s = 5$, $E = 1$, $b = 5$, 是一个直接映射高速缓存（每组只有一行），有32个组（2的 s 次方），每行32个bits（2的 b 次方），每行(组)可存8个 `int`。

1.2.1 32x32

未优化前的暴力做法分析

- 未进行优化前，函数按行读取 `A` 矩阵，然后按列写入 `B` 矩阵。实际跑出的结果miss1184次，远超出题目要求的次数。

```

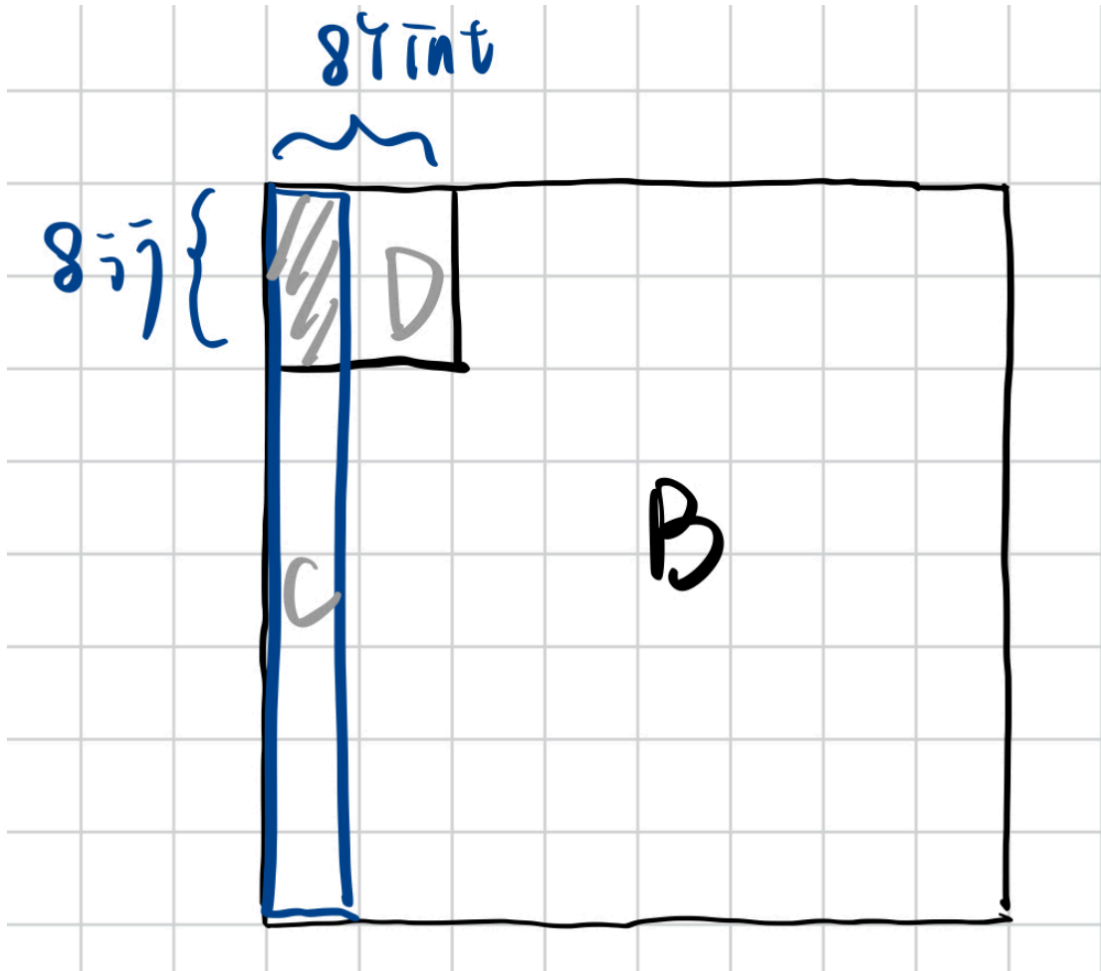
Function 1 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:869, misses:1184, evictions:1152

```

- 对于 `A`，以读 `A` 的第1行为例，在读 `A[0][0]` 时，除了 `A[0][0]` 被加载到cache中，它之后的 `A[0][1]---A[0][7]` 也被加载进cache。因为 `A` 刚好是按行读取的，所以在接下来读 `A[0][1]---A[0][7]` 时不需要重新加载，每一行会有 $32/8 = 4$ 次不命中，所以读 `A` 大概共 4×32 次miss。
- 然而对于 `B`，由于 `B` 是按列写入，而在列上相邻的元素不在一个内存块上，所以每次写入都不命中缓存。并且一列写完读下一列时，原来的缓存可能被覆盖了，这样就会又不命中。考虑最坏情况，`B` 的每一列都有 32 次不命中，所以写 `B` 大概共 32×32 次miss。
- 未优化前理论的miss为 $4 \times 32 + 32 \times 32 = 1152$ 次，比实际的miss(1184)少了一点，这是由于 `A` 和 `B` 的地址由于取余关系，每个元素对应的地址是相同的，对角线部分两者会冲突，导致了更多的miss。

优化的做法

- 由上面暴力做法导致miss的原因分析可知，主要需要优化的部分在于**写B时缓存的利用**。如图，在写入B的前8行后，B的D区域就进入了缓存，而未优化解法接下来操作的是C，每一个元素的写都要驱逐之前的缓存区，当来到第2列继续写D时，它对应的缓存行很可能已经被驱逐了，于是又要miss。此时如果能对D进行操作，那么就能利用上缓存的内容，不会miss。



- 于是考虑**分块**，由于A中的元素对应的缓存行每隔8行就会重复，尝试**8x8分块**，代码如下：

```
void trans_88(int M,int N,int A[N][M],int B[M][N]){
    for(int n=0;n<N;n+=8){
        for(int m=0;m<M;m+=8){//分成8*8的小块
            for(int i=n;i<n+8;i++){
                for(int j=m;j<m+8;j++){
                    B[j][i]=A[i][j];
                }
            }
        }
    }
}
```

结果如下：

```
Function 2 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (8*8 blocking): hits:1709, misses:344, evictions:312
```

miss344次，仍然未达到题目的要求，比预期的每8次miss一次算出的 $4 \times 2 \times 32 = 256$ 多了很多。

- 考虑**对角线上的块**。A 与 B 对角线上的块在缓存中对应的位置是相同的，而它们在转置过程中位置**不变**，所以复制过程中会发生**相互冲突**。以 A 的一个对角线块 p，B 与 p 相应的对角线块 q 为例，复制前，p 在缓存中。复制时，q 会驱逐 p。下一个开始复制 p 又被重新加载进入缓存驱逐 q，这样就会**多产生两次 miss**。
- 于是使用**局部变量**一次性存下 A 的一行再复制给 B，减少重复交替访问造成的 miss，代码如下（代码中 3 个循环局部变量，8 个局部变量，共 11 个局部变量，符合题目至多 12 个局部变量的要求）：

```
for(int n=0;n<N;n+=8){
    for(int m=0;m<M;m+=8){//分成8*8的小块
        for(int i=n;i<n+8;i++){
            a=A[i][m];
            b=A[i][m+1];
            c=A[i][m+2];
            d=A[i][m+3];
            e=A[i][m+4];
            f=A[i][m+5];
            g=A[i][m+6];
            h=A[i][m+7];//使用8个局部变量，暂存A的一行

            B[m][i]=a;
            B[m+1][i]=b;
            B[m+2][i]=c;
            B[m+3][i]=d;
            B[m+4][i]=e;
            B[m+5][i]=f;
            B[m+6][i]=g;
            B[m+7][i]=h;
        }
    }
}
```

结果如下：

```
Function 0 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1765, misses:288, evictions:256
```

满足要求！

1.2.2 64x64

尝试8x8分块

```
Function 1 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3473, misses:4724, evictions:4692

Function 2 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (8*8 blocking) hits:3473, misses:4724, evictions:4692
```

- 发现**8x8分块**的效果和**暴力做法**一致，都是4724次miss

- 原因是64x64的矩阵4行即可占满cache，所以8x8分块后，上半块和下半块的数据会发生冲突不命中，导致miss增加

改为4x4分块

代码如下：

```
void trans_64(int M,int N,int A[N][M],int B[M][N]){
    for(int n=0;n<N;n+=4){
        for(int m=0;m<M;m+=4){ //分成4*4的小块
            for(int i=n;i<n+4;i++){
                for(int j=m;j<m+4;j++){
                    B[j][i]=A[i][j];
                }
            }
        }
    }
}
```

结果如下：

```
Function 4 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 4 (simply 4*4 blocking for 64*64): hits:6305, misses:1892, evictions:1860
```

miss的数量明显减少了，为1892，但仍然未达到题目满分的要求。

先 8x8 分大块，再在每个大块内 4x4 分小块

- 读完A的第一行（8个int）后，先将前4个正常转置并写入B，然后将后4个先放到B的右上角暂时存储，这样就可以避免写下半块时的冲突不命中。
- 于是具体步骤及对应的miss情况为：（以下A表示A矩阵的一个8x8分块，B表示B矩阵的一个8x8分块）

1. 读A的前四行同时转置放入B的前四行（放入后B的左上角已为最终结果，B的右上角暂存了最终的左下角）

- 无论是否为对角线上元素，读A每行会miss一次，共miss四次
- 若非对角线上的块，B在写入第一列时miss四次，之后无miss，共miss四次
- 若为对角线上的块，B在第二/三/四次循环时都需要覆盖对应的A的第二/三/四行，多miss三次，共miss七次

执行完这一步后，缓冲区中是A的上半部分及B的上半部分

2. 将A的左下角转置到B的右上角，同时把B右上角暂存的复制到B的左下角

读时的miss：

- 读A的左下角的一列，会miss四次。若为对角线块，还会因为循环间读B，读后三列时共多三次miss
- 若为非对角线上的块，再读B时因为B在上一步时已存于缓存区，不会造成miss
- 若为对角线上的块，再读B时每次都会因为之前读A造成miss，共四次miss

写时的miss：

- 在读完B的右上角后写B的右上角，不会造成miss
- 每次写B的左下角造成一次miss，共四次miss

此步中先读A再读B效果一致，执行完后缓冲区中有B的下半部分及A的下半部分（若为非对角线块），或A的下半部分除最后一行及B的最后一行（若为对角线元素）

3. 将A的右下角先复制再转置到B的右下角

复制:

- 若为非对角线元素，复制前所需部分均在缓冲区中，**没有miss**
- 若为对角线元素，因为上一部分最后是写 B 的最后一行，但是其他行都是正常的，所以读 A 时会出现 1 次冲突不命中。读完 A 写 B 时，必定出现冲突不命中，每次循环 1 次 miss，所以总计造成**五次miss**

转置:

- B的右小角均在缓冲区中，**自身转置没有miss**

- 完整代码如下:

```
int a,b,c,d,e,f,g,h;
int k;
for(int m=0;m<64;m+=8){//行
    for(int n=0;n<64;n+=8){//列
        for(k=0;k<4;k++){
            a=A[m+k][n];
            b=A[m+k][n+1];
            c=A[m+k][n+2];
            d=A[m+k][n+3];
            e=A[m+k][n+4];
            f=A[m+k][n+5];
            g=A[m+k][n+6];
            h=A[m+k][n+7];//从A中取前四行(循环下来共会有4次miss)

            B[n][m+k]=a;
            B[n+1][m+k]=b;
            B[n+2][m+k]=c;
            B[n+3][m+k]=d;
            B[n][m+k+4]=e;
            B[n+1][m+k+4]=f;
            B[n+2][m+k+4]=g;/*!注意A的行号是B的列号! 写矩阵索引的时候注意!
            B[n+3][m+k+4]=h;*/转置后读到B的上半部分(共4次miss)
        }
        for(k=0;k<4;k++){
            a=A[m+4][n+k];
            b=A[m+5][n+k];
            c=A[m+6][n+k];
            d=A[m+7][n+k];//存A
            e=B[n+k][m+4];
            f=B[n+k][m+5];
            g=B[n+k][m+6];
            h=B[n+k][m+7];//存B

            B[n+k][m+4]=a;
            B[n+k][m+5]=b;
            B[n+k][m+6]=c;
            B[n+k][m+7]=d;//读完B右上角后写B右上角 减少miss
            B[n+k+4][m]=e;
            B[n+k+4][m+1]=f;
            B[n+k+4][m+2]=g;
```

```

        B[n+k+4][m+3]=h;//写B左下角
    }//把A的左下角转置到B的右上角 同时把B右上角复制到B的左下角
    for(k=4;k<8;k++){
        a=A[m+k][n+4];
        b=A[m+k][n+5];
        c=A[m+k][n+6];
        d=A[m+k][n+7];
        B[n+k][m+4]=a;
        B[n+k][m+5]=b;
        B[n+k][m+6]=c;
        B[n+k][m+7]=d;
    }//把A的右下角先复制到B的右下角
    for(k=4;k<8;k++){
        for(int i=k+1;i<8;i++){
            a=B[n+k][m+i];
            B[n+k][m+i]=B[n+i][m+k];//!
            B[n+i][m+k]=a;
        }
    }//在B处进行转置 防止出现多余的miss
}
}

```

- 运行结果如下:

```

Function 0 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:10633, misses:1148, evictions:1116

```

miss数为1148, 满足要求!

1.2.3 61x67

- 本题对于miss数目的要求较低, 尝试不同的分块, 发现分块大小为16x16, 17x17, 18x18时均小于2000
- 分块大小为17x17时, 代码如下:

```

for(int n=0;n<N;n+=17){
    for(int m=0;m<M;m+=17){//分成17*17的小块
        for(int i=n;i<n+17&& i<N;i++){
            for(int j=m;j<m+17&& j<M;j++){
                B[j][i]=A[i][j];
            }
        }
    }
}
}

```

结果如下:

```

Function 0 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6228, misses:1951, evictions:1919

```

miss数为1951, 满足要求!

2实验中遇到和解决的问题

2.1对角线上A和B的地址重合问题

问题:

发现32x32矩阵在8x8分块后miss344次，仍然未达到题目的要求，**比预期算出的 $4 \times 2 \times 32 = 256$ 多了很多**

解决:

- 查看 `tracegen.c` 文件，发现无论 `M`，`N` 设定什么值，`A` 和 `B` 都是存在 `256x256` 大小的静态数组中的，`256x256` 是题目给定的cache大小的整数倍，因此由于取余的关系，**A和B在相同位置上的元素将会缓存在相同的 `cache_line` 中。**
- 原来计算的时候只考虑了非对角线上的元素（地址不重合的情况），没有考虑**对角线上的元素**。而对角线上的元素会因为地址冲突而造成**缓存抖动**。以 `A` 的一个对角线块 `p`，`B` 与 `p` 相应的对角线块 `q` 为例，复制前，`p` 在缓存中。复制时，`q` 会驱逐 `p`。下一个开始复制 `p` 又被重新加载进入缓存驱逐 `q`，**这样的重复替换就会多产生两次 miss。**
- 具体的解决策略为:使用**局部变量**一次性存下A的一行再复制给B，从而减少重复交替访问造成的 miss。

3实验最终结果截图

```
wcx@LAPTOP-OMTCB5PG:~/lab4/cacheLab-handout$ python2 driver.py
Part A: Testing cache simulator
Running ./test-csim
```

Points (s,E,b)	Your simulator			Reference simulator			
	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	288
Trans perf 64x64	8.0	8	1148
Trans perf 61x67	10.0	10	1951
Total points	53.0	53	