Part-A

一、实验目的

在Part A，你要在scim.c文件中写一个cache模拟器，以valgrind内存trace作为输入，模拟这个trace在cache上的hit/miss行为，输出hit、miss和替换页（eviction）的总数。

二、实现分析

PartA部分的代码由于实验平台过于卡，我使用vscode进行代码抒写，并使用clipboard实现代码转换进虚拟实验平台。

首先提示以及题目要求导入库。

```
#include "cachelab.h"
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <stddef.h>
```

根据课上学的缓存器结构和初始化一些参数(hits,misses等)对其进行代码实现。

```
typedef struct {
    int valid;
    unsigned tag;
    int timestamp;
} CacheLine;

char* traceFilePath = NULL;
int setBits, lineCount, blockBits, numSets;
int hits = 0, misses = 0, evictions = 0;
CacheLine** cache = NULL;
```

根据要求可以得知需要实现初始化缓存器，对缓存器进行访问以及是实现LRU所需要的更新时间戳函数

```
// 函数声明
void initializeCache();
void accessCache(unsigned address);
void updateTimestamps();
```

主函数,所有代码操作解释已经写在注释

```
int main(int argc, char *argv[]) {
    // 首先透过getopt得到参数
    int opt;
    while ((opt = getopt(argc, argv, "s:E:b:t:")) != -1) {
        switch (opt) {
            case 's':
                setBits = atoi(optarg);
                break;
            case 'E':
```

```
                lineCount = atoi(optarg);
                break;
            case 'b':
                blockBits = atoi(optarg);
                break;
            case 't':
                traceFilePath = optarg;
                break;
        }
    }

    // 因为总共的大小是 2^s
    numSets = 1 << setBits;

    // 初始化缓存
    initializeCache();

    FILE* traceFile = fopen(traceFilePath, "r");
    if (traceFile == NULL) {
        printf("打开文件错误");
        exit(-1);
    }

    char operation;
    unsigned address;
    int size;
    while (fscanf(traceFile, " %c %x,%d", &operation, &address, &size) > 0) {
        switch (operation) {
            case 'L':
            case 'M':
            case 'S':
                accessCache(address);
                break;
        }
        updateTimestamps();
    }

    // 释放缓存
    for (int i = 0; i < numSets; i++)
        free(*(cache + i));
    free(cache);
    fclose(traceFile);
    printSummary(hits, misses, evictions);
    return 0;
}
```

初始化缓存函数

```
// 初始化缓存
void initializeCache() {
    cache = (CacheLine**)malloc(sizeof(CacheLine*) * numSets);
    for (int i = 0; i < numSets; i++) {
        *(cache + i) = (CacheLine*)malloc(sizeof(CacheLine) * lineCount);
        for (int j = 0; j < lineCount; j++) {
            cache[i][j].valid = 0;
            cache[i][j].tag = 0xffffffff;
            cache[i][j].timestamp = 0;
        }
    }
}
```

访问缓存函数-这个部分比较复杂需要考虑三个情况，命中，未命中的两个情况，实现LRU，这里是缓存的核心

```
// 访问缓存
// 这里要实现击中or没击中两种情况(包括LRU)去做考虑
void accessCache(unsigned address) {
    unsigned setIndex = (address >> blockBits) & ((0xffffffff) >> (32 -
setBits));
    unsigned tagIndex = address >> (setBits + blockBits);

    for (int i = 0; i < lineCount; i++) {
        CacheLine* currentLine = cache[setIndex] + i;
        if (currentLine->valid && currentLine->tag == tagIndex) {
            currentLine->timestamp = 0; // 命中，将时间戳设为0
            hits++;
            return;
        }
    }

    for (int i = 0; i < lineCount; i++) {
        CacheLine* currentLine = cache[setIndex] + i;
        if (!currentLine->valid) {
            currentLine->tag = tagIndex;
            currentLine->valid = 1;
            currentLine->timestamp = 0;
            misses++;
            return;
        }
    }

    int maxTimestamp = 0;
    int maxIndex = 0;
    for (int i = 0; i < lineCount; i++) {
        if (cache[setIndex][i].timestamp > maxTimestamp) {
            maxTimestamp = cache[setIndex][i].timestamp;
            maxIndex = i;
        }
    }
    evictions++;
    misses++;
    cache[setIndex][maxIndex].tag = tagIndex; // 实现LRU策略，替换
    cache[setIndex][maxIndex].timestamp = 0;
}
```

更新时间戳函数-实现对时间戳的更新，使LRU能够实现

```
// 更新时间戳
void updateTimestamps() {
    for (int i = 0; i < numSets; i++) {
        for (int j = 0; j < lineCount; j++) {
            if (cache[i][j].valid)
                cache[i][j].timestamp++;
        }
    }
}
```

此部分的实验结果在最后

Part-B

一、实验目的

本实验目的是帮助你理解cache对C语言程序性能的影响。

二、报告要求

本报告要求学生解释说明所提交的转置函数是如何针对三种矩阵大小进行了何种优化的，优化效果如何。欢迎你写出你尝试过的所有优化策略，比较它们的效果以及分析效果优劣的原因。

对于每组矩阵大小，你的性能分根据miss数量m，在两个阈值之间线性计分：

- 32×32: 若m<300，得15分，若m>600，则为0分
- 64×64: 若m<1,300，得15分, 若m>2,000 ，则为0分
- 61×67: 若m<2,000，得15分，若m>3,000，则为0分

三、实现分析

这里使用分支语句分割三种情况

首先经过我自己的测试以及分析发现到miss过多的原因是访问两个数组的过程中存在太多的冲突不命中，其原因是B数组和A数组下标相同的情况下会映射到同一个cache块，造成不断地发生了冲突不命中。分析可以得知本实验要求就是解决冲突不命中问题得以优化。

首先初步测试运行顺序，这里使用的是最原始的矩阵转置方式

```
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;
        }
    }

}
```

运行结果：

可以看到misses达到了1183，远远高于hits的数量

1. 针对 **32X32 (M=32,N=32)** 的矩阵

   根据我们以上的分析，这里尝试用8个元素作为同一块进行访问跟转置也就是使用分块的思想。

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, bi, bj;
    for(bi = 0;bi < M;bi += 8){
        for(bj = 0;bj < N;bj+=8){
            for(i = bi;i < bi+8;i++){
                for(j = bj; j < bj+8;j++){
                    B[j][i] = A[i][j];
                }
            }
        }
    }
}
```

   得到结果

   可以发现到misses优化有了很大的进展，但是还是没有低于300。

   接下来优化的思想就只适用于对称矩阵，做法是将A中缓存的值用变量保存起来。

   代码改进：

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, bi, bj;
    int tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
    for(bi = 0;bi < M;bi += 8){
        for(bj = 0;bj < N;bj+=8){
            for(i = bi;i < bi+8;i++){
                tmp1 = A[i][0+bj];
                tmp2 = A[i][1+bj];
                tmp3 = A[i][2+bj];
                tmp4 = A[i][3+bj];
                tmp5 = A[i][4+bj];
                 tmp6 = A[i][5+bj];
                 tmp7 = A[i][6+bj];
                 tmp8 = A[i][7+bj];

                B[0+bj][i] = tmp1;
                B[1+bj][i] = tmp2;
                B[2+bj][i] = tmp3;
                B[3+bj][i] = tmp4;
                B[4+bj][i] = tmp5;
                B[5+bj][i] = tmp6;
                B[6+bj][i] = tmp7;
                B[7+bj][i] = tmp8;
            }
        }
```

```
    }
}
```

得到结果

成功达到要求，**misses<300**！！！

2. 针对 **64X64 (M=64,N=64)** 的矩阵

使用类似32X32使用的优化方法，得到的效果非常不好，过后上网看了几篇思路文才发现可以在**八分块再分四块**的方法达到要求。

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, bi, bj;
    int tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
    for (bi = 0; bi < N; bi += 8) {
            for (bj = 0; bj < M; bj += 8) {
                for (i = bi; i < bi + 4; i++) {
                    tmp1 = A[i][0+bj];
                    tmp2 = A[i][1+bj];
                    tmp3 = A[i][2+bj];
                    tmp4 = A[i][3+bj];
                    tmp5 = A[i][4+bj];
                    tmp6 = A[i][5+bj];
                    tmp7 = A[i][6+bj];
                    tmp8 = A[i][7+bj];

                    B[0+bj][i] = tmp1;
                    B[1+bj][i] = tmp2;
                    B[2+bj][i] = tmp3;
                    B[3+bj][i] = tmp4;
                    B[0+bj][4+i] = tmp5;
                    B[1+bj][4+i] = tmp6;
                    B[2+bj][4+i] = tmp7;
                    B[3+bj][4+i] = tmp8;
                }
                for (j = bj; j < bj + 4; j++) {
                    tmp1 = A[4+bi][j];
                    tmp2 = A[5+bi][j];
                    tmp3 = A[6+bi][j];
                    tmp4 = A[7+bi][j];
                    tmp5 = B[j][4+bi];
                    tmp6 = B[j][5+bi];
                    tmp7 = B[j][6+bi];
                    tmp8 = B[j][7+bi];

                    B[j][4+bi] = tmp1;
                    B[j][5+bi] = tmp2;
                    B[j][6+bi] = tmp3;
                    B[j][7+bi] = tmp4;
                    B[4+j][0+bi] = tmp5;
                    B[4+j][1+bi] = tmp6;
                    B[4+j][2+bi] = tmp7;
```

```
                    B[4+j][3+bi] = tmp8;
                }
                for (i = bi + 4; i < bi + 8; i++) {
                     tmp1 = A[i][4+bj];
                    tmp2 = A[i][5+bj];
                    tmp3 = A[i][6+bj];
                    tmp4 = A[i][7+bj];

                     B[4+bj][i] = tmp1;
                    B[5+bj][i] = tmp2;
                    B[6+bj][i] = tmp3;
                    B[7+bj][i] = tmp4;
                }
            }
    }
}
```

结果

比起原本的矩阵转置方式，成功优化使**misses<1300**

3. 针对 **61X67 (M=61,N=67)** 的矩阵

   由于本体是不对称的矩阵，所以我们直接使用最开始的分块方法，进行测试得到当块为16时，成功得到**misses<2000**的结果。

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
        int bi, bj, i, j, tmp;
        int block_size = 16;
        for (bi = 0; bi < N; bi += block_size) {
            for (bj = 0; bj < M; bj += block_size) {
                for (i = bi; i < N && i < bi + block_size; i++) {
                        for (j = bj; j < M && j < bj + block_size; j++) {
                            tmp = A[i][j];
                            B[j][i] = tmp;
                        }
                }
            }
        }
}
```

结果

full code

```
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){
    int i, bi, bj;
    int tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
    for(bi = 0;bi < M;bi += 8){
```

```
        for(bj = 0;bj < N;bj+=8){
            for(i = bi;i < bi+8;i++){
                tmp1 = A[i][0+bj];
                tmp2 = A[i][1+bj];
                tmp3 = A[i][2+bj];
                tmp4 = A[i][3+bj];
                tmp5 = A[i][4+bj];
                    tmp6 = A[i][5+bj];
                    tmp7 = A[i][6+bj];
                    tmp8 = A[i][7+bj];

                B[0+bj][i] = tmp1;
                    B[1+bj][i] = tmp2;
                    B[2+bj][i] = tmp3;
                    B[3+bj][i] = tmp4;
                B[4+bj][i] = tmp5;
                    B[5+bj][i] = tmp6;
                    B[6+bj][i] = tmp7;
                    B[7+bj][i] = tmp8;
            }
        }
    }
    }else if(M == 64 && N == 64){
     int i, j, bi, bj;
     int tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
     for (bi = 0; bi < N; bi += 8) {
             for (bj = 0; bj < M; bj += 8) {
                 for (i = bi; i < bi + 4; i++) {
                         tmp1 = A[i][0+bj];
                     tmp2 = A[i][1+bj];
                     tmp3 = A[i][2+bj];
                     tmp4 = A[i][3+bj];
                 tmp5 = A[i][4+bj];
                     tmp6 = A[i][5+bj];
                     tmp7 = A[i][6+bj];
                     tmp8 = A[i][7+bj];

                         B[0+bj][i] = tmp1;
                     B[1+bj][i] = tmp2;
                     B[2+bj][i] = tmp3;
                     B[3+bj][i] = tmp4;
                         B[0+bj][4+i] = tmp5;
                     B[1+bj][4+i] = tmp6;
                     B[2+bj][4+i] = tmp7;
                     B[3+bj][4+i] = tmp8;
                 }
                 for (j = bj; j < bj + 4; j++) {
                         tmp1 = A[4+bi][j];
                     tmp2 = A[5+bi][j];
                     tmp3 = A[6+bi][j];
                     tmp4 = A[7+bi][j];
                         tmp5 = B[j][4+bi];
                     tmp6 = B[j][5+bi];
                     tmp7 = B[j][6+bi];
                     tmp8 = B[j][7+bi];

                         B[j][4+bi] = tmp1;
                     B[j][5+bi] = tmp2;
```

```
                    B[j][6+bi] = tmp3;
                    B[j][7+bi] = tmp4;
                        B[4+j][0+bi] = tmp5;
                    B[4+j][1+bi] = tmp6;
                    B[4+j][2+bi] = tmp7;
                    B[4+j][3+bi] = tmp8;
                }
                for (i = bi + 4; i < bi + 8; i++) {
                     tmp1 = A[i][4+bj];
                 tmp2 = A[i][5+bj];
                 tmp3 = A[i][6+bj];
                 tmp4 = A[i][7+bj];

                         B[4+bj][i] = tmp1;
                 B[5+bj][i] = tmp2;
                 B[6+bj][i] = tmp3;
                 B[7+bj][i] = tmp4;
                }
            }
     }
    }else if(M == 61 && N == 67){
         int bi, bj, i, j, tmp;
         int block_size = 16;
         for (bi = 0; bi < N; bi += block_size) {
             for (bj = 0; bj < M; bj += block_size) {
                 for (i = bi; i < N && i < bi + block_size; i++) {
                         for (j = bj; j < M && j < bj + block_size; j++) {
                             tmp = A[i][j];
                             B[j][i] = tmp;
                         }
                 }
             }
         }
    }
}
```

实验总结果

四、实验总结

本次的实验是真的困难，如果不上网寻找思路，partB部分根本做不出来，很庆幸又学会了新的优化方法，也成功体会到缓存的作用以及其模拟代码。