Cache Lab实验报告

侯新铭 2021201651

Part A

任务是实现缓存存储器的相关功能，替换操作通过LRU策略来实现。

定义结构体

cache有多层结构，因此，我通过line，set，cache三个结构来嵌套定义，组织其变量，相关代码及注释如下：

```
// 定义Line结构体，用来组织cache模拟器基本结构中每个line的4个参数
typedef struct
{
    memory tag;      // 主存标记
    char *block; // 高速缓存块指针
    int valid;      // 有效位
    int age;      // 作为时间戳，通过比较，将最后的访问时间是距离现在最远的块被替换掉
} Line;

// 定义Set，作为Line构成的集合，组织形式为链表，即存储指向Line型实体的指针
typedef struct
{
    Line *lines;
} Set;

// 定义完整的Cache的结构
typedef struct
{
    // 基本属性
    int E;
    int b;
    int s;

    // 输出参数
    int hits;
    int misses;
    int evicts;

    // 由Set类型的变量以链表的形式存储cache内部的各个变量
    Set *sets;
} Cache;
```

初始化与free

即为上述定义的嵌套结构分配与free空间：

```
Cache *cacheInit(int s, int b, int E)
{
    Cache *cache;
    cache = (Cache *)calloc(1, sizeof(Cache));

    cache->s = s;
    cache->b = b;
    cache->E = E;

    cache->hits = 0;
    cache->misses = 0;
    cache->evicts = 0;

    int S = 2 << s;
    int B = 2 << b;

    cache->sets = (Set *)calloc(S, sizeof(Set));
    Set *sets = cache->sets;

    int i;
    // 外层循环为分配line分配空间，内层循环为line中block指针分配空间
    for (i = 0; i < S; i++)
    {
        Set set;
        set.lines = (Line *)calloc(E, sizeof(Line));
        int j;
        Line *lines = set.lines;
        for (j = 0; j < E; j++)
        {
            Line line;
            line.block = (char *)calloc(B, sizeof(char));
            line.valid = 0;
            lines[j] = line;
        }

        sets[i] = set;
    }
    return cache;
}
void cacheFree(Cache *cache)
{
    Set *sets = cache->sets;
    int S = 2 << cache->s;
    int E = cache->E;

    int i;
    for (i = 0; i < S; i++)
    {
        Line *lines = sets[i].lines;
        int j;
        for (j = 0; j < E; j++)
            free(lines[j].block);
        free(lines);
```

```
    }

    free(sets);
    free(cache);
    return;
}
```

需要注意的是，每层结构中的指针均需为其分配空间，一开始我写的时候就遗漏了为最内层的block指针分配空间的过程。

hit函数和miss函数

hit时，执行hit函数，实现修改时间戳和hits输出变量的功能：

```
// hit时执行函数
void hit(Cache *cache, Set *sp, Line *lp, int E)
{

    // 修改时间戳
    int i;
    for (i = 0; i < E; i++)
    {
        Line *curLp = &(sp->lines[i]);
        curLp->age++;
    }
    // 重置该line的时间戳为0
    lp->age = 0;
    cache->hits++;
    return;
}
```

miss时，执行miss函数，为LRU策略的主体部分，修改相关line的参数，以及输出变量misses和evicts。相关代码及对实现思路的注释如下：

```
// miss时执行函数
void miss(Cache *cache, Set *sp, int E, memory tag)
{
    cache->misses++;
    // 根据LRU策略，寻找到最后访问时间最早的line，修改其参数并替换它
    Line *TargetLine = NULL;

    // 修改时间戳
    int k;
    for (k = 0; k < E; k++)
    {
        Line *curLp = &(sp->lines[k]); // cur指针指向当前line
        curLp->age++;
    }
```

```
    // 遍历所有line，不断更新TargetLine
    int i;
    for (i = 0; i < E; i++)
    {
        Line *curLp = &(sp->lines[i]);
        // 找到空line，直接写入覆盖它，函数执行完毕
        if (!curLp->valid)
        {
            curLp->tag = tag;
            curLp->age = 0;
            curLp->valid = 1;
            return;
        }
        // 非空，则与前已检索lines中的最早line的时间戳比较，若更早，则更新TargetLine
        else if (TargetLine == NULL || curLp->age > TargetLine->age)
        {
            TargetLine = curLp;
        }
    }

    // 遍历完所有line仍为return，即均已被使用，即需替换所有lines中找到的最早line
    TargetLine->tag = tag;
    TargetLine->age = 0;
    cache->evicts++;
    return;
}
```

搜索cache的主体函数

调用上述hit和miss函数，为所给地址寻找最合适的cache，相关代码及对实现思路的注释如下：

```
// 以LRU策略为所给地址寻找最合适的cache
void search(Cache *cache, memory address)
{
    int b = cache->b;
    int s = cache->s;
    int E = cache->E;

    int setNum = (address >> b) & ~(~0 << s);                                    //
根据cache的s和b参数计算目标set的标号
    Set *sp = &(cache->sets[setNum]);                                            //
获得指向目标set的指针
    int tag = (address >> (s + b)) & ~(~0 << ((sizeof(memory) * 8) - (s + b))); //
根据cache的s和b参数计算tag

    Line *lp = search4Line(sp, tag, E); // 调用search4Line函数在上述set中找到给定tag
对应的有效line

    // hit的情形,执行hit函数
    if (lp != NULL)
    {
```

```
            hit(cache, sp, lp, E);
            return;
        }
        else // 未hit,执行miss函数
        {
            miss(cache, sp, E, tag);
            return;
        }
    }
```

main函数

涉及命令行输入、文件输入等操作，主体为两个`switch`，使用上述封装好的结构和函数即可：

```
int main(int argc, char *argv[])
{
    // 待输入参数
    int E;
    int s;
    int b;
    char *t;

    // 从命令行读取flag及相应地址处的字符串对应数组，存储到对应变量中
    char flag = getopt(argc, argv, "s:E:b:t:");
    do
    {
        switch (flag)
        {
        case 'E':
            E = atoi(optarg);
            break;

        case 's':
            s = atoi(optarg);
            break;

        case 'b':
            b = atoi(optarg);
            break;

        case 't':
            t = optarg;
            break;

        case 'v':
            break;

        default:
            return -1;
        }
    } while ((flag = getopt(argc, argv, "s:E:b:t:")) != -1);
```

```
    Cache *cache = cacheInit(s, b, E); // 初始化cache

    // 读取给定文件
    FILE *f;
    f = fopen(t, "r");
    if (f != NULL)
    {
        char operation;
        memory address;
        int size;

        while (fscanf(f, " %c %llx,%d", &operation, &address, &size) == 3)
        {
            switch (operation)
            {
            case 'S':
                search(cache, address);
                break;
            case 'L':
                search(cache, address);
                break;
            case 'M':
                search(cache, address); // 数据加载过程
                search(cache, address); // 数据存储过程
                break;

            default:
                break;
            }
        }
    }
    fclose(f);

    printSummary(cache->hits, cache->misses, cache->evicts);

    cacheFree(cache);
    return 0;
}
```

Part B

任务是使用分块技术对三种不同维度的矩阵转置操作进行优化。我将32*32和64*64两种维度整合起来通过相同一段代码实现优化（分块及再分块思路完全相同）；详细的实现过程及思路已通过代码注释的形式呈现：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{

    int interval = 8;   // 间隔初始化为8，对32*32和64*64规模的矩阵均按8分块
    int nBegin, mBegin; // 用来当前时刻记录两个维度下将要移动存储的初始位置
```

```
    if (M == 61) // 61*67作为不规则规模的矩阵，单独处理
    {

        interval = 16; // 经尝试发现用16来分块很合适

        for (nBegin = 0; nBegin < N; nBegin += interval) // 外两层循环遍历整个矩阵
            for (mBegin = 0; mBegin < M; mBegin += interval)
            {
                int i, j;
                for (i = 0; i < interval; i++) // 内两层循环遍历单个分块
                    for (j = 0; j < interval; j++)
                    {

                        if (nBegin + i >= N)
                            continue;
                        if (mBegin + j >= M)
                            continue;

                        B[mBegin + j][nBegin + i] = A[nBegin + i][mBegin + j];
                    }
            }
        return;
    }

    // 下述代码处理32*32和64*64的情形，均按照8分块来操作

    // 外两层循环遍历整个矩阵
    for (mBegin = 0; mBegin < M; mBegin += interval)
        for (nBegin = 0; nBegin < N; nBegin += interval)
        {
            // 在进行内部循环调整之前，记录当前遍历到的分块的转置后的位置
            int mPos = nBegin;
            int nPos = mBegin;

            // 初始化目标待调整位置变量
            int mTar = mPos;
            int nTar = nPos;

            // “找位置”：从当前位置开始，调整m维度上的位置到转置后分块起始位置，并在调整过程中对应调整n维度
            do
            {
                mTar += interval;
                if (mTar >= N) // m维度上位置越界后，n维度调整一位，mTar对N取模
                {
                    nTar += interval;
                    mTar -= N;
                }
            } while (mTar == mBegin);

            // 初始化下一个的目标待调整位置变量
            int mTarNext = mTar;
            int nTarNext = nTar;
```

```
            // “找下一个位置”：从刚刚找到的位置开始，调整m维度上的位置到转置后下一个分块的起始位置，并在调整过程中对应调整n维度
            do
            {
                mTarNext += interval;
                if (mTarNext >= N) // m维度上位置越界后，n维度调整一位，mTar对N取模
                {
                    nTarNext += interval;
                    mTarNext -= N;
                }

            } while (mTarNext == mBegin);

            int i, j;
            // 若nTar已经越界了，则转置位置仍为“整块”，则直接把8*8分块挪到转置后位置即可
            if (nTar >= M)
            {

                for (i = 0; i < interval; i++)
                    for (j = 0; j < interval; j++)
                        B[nPos + j][mPos + i] = A[nBegin + i][mBegin + j];
                continue;
            }
            //nTar没有越界，转置后的位置“分散”，需作出精细化处理，画图后总结规律发现，可再分成4分块的子块部分，对应到下面的4个循环，分别挪过去
            for (i = 0; i < SPACE; i++)
                for (j = 0; j < interval; j++)
                    B[nTar + i][mTar + j] = A[nBegin + i][mBegin + j];

            for (i = 0; i < SPACE; i++)
                for (j = 0; j < interval; j++)
                    B[nTarNext + i][mTarNext + j] = A[nBegin + SPACE + i][mBegin + j];

            for (i = 0; i < SPACE; i++)
                for (j = 0; j < SPACE; j++)
                {
                    B[nPos + j][mPos + i] = B[nTar + i][mTar + j];
                    B[nPos + j][mPos + SPACE + i] = B[nTarNext + i][mTarNext + j];
                }

            for (i = 0; i < SPACE; i++)
                for (j = 0; j < SPACE; j++)
                {
                    B[nPos + SPACE + j][mPos + i] = B[nTar + i][mTar + SPACE + j];
                    B[nPos + SPACE + j][mPos + SPACE + i] = B[nTarNext + i][mTarNext + SPACE + j];
                }
        }
}
```