CSAPP-CacheLab Report

1. 评分

```
avalon@LAPTOP-8ABRBJH3:~/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
avalon@LAPTOP-8ABRBJH3:~/cachelab-handout$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     6 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     6 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     6 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     6 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     6 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     6 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     6 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     8 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    50

TEST_CSIM_RESULTS=50
```

2. 头文件

   接下来要用到的一些库函数，$getopt$和实验需要的函数的的头文件：

```c
#include "cachelab.h" // printSummary
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free, exit
#include <getopt.h> // getopt
#include <unistd.h>
```

3. 数据结构与全局变量

   数据结构方面，我主要定义了缓存行的结构体，包含有效位、标记位和LRU计数器，如下：

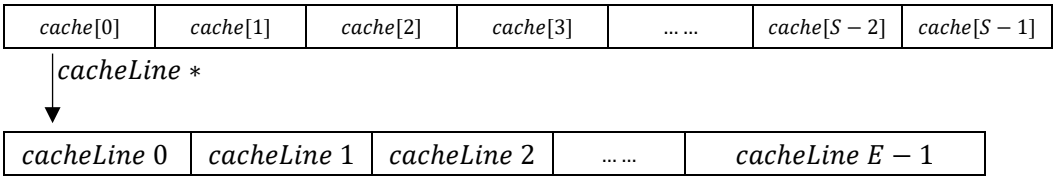
```c
// 定义缓存行的结构体，包含有效位、标记位和 LRU 计数器
typedef struct
{
    int valid;
    unsigned long tag;
    unsigned long lru;
} cacheLine;
```

   其中，lru用于给缓存行加时间戳。维护一个全局的$lruCount$，初始为0，每次访问一行，就把它的lru改为当前的$lruCount+1$，同时更新$lruCount$。这样，lru越小，就是越早被访问的，在实现LRU替换策略时，只要找到lru最小的缓存行即可。

   然后，我们把$cacheLine$ **作为$cache$，$cache$数组的每一个元素是一个组，也即一个$cacheLine$指针，指向缓存行构成的数组：

```c
typedef cacheLine **cache; //定义缓存数组
```

   $cache$指向这样一个数组：



全局变量方面，维护如下的全局变量用于存储命令行参数和缓存统计信息。由于$S = 2^s, B = 2^b$，维护s, E, b, S, B。

```
// 定义全局变量，用于存储命令行参数和缓存统计信息
int s = -1, E, b = -1; // 缓存的组数的指数、每组的行数和每行的块数的指数
int S, B; // 缓存的组数、每行的块数
```

以及其他需要的信息如下：

```
char *traceFile; // trace 文件的路径
int verbose; // 是否打印详细信息
int hitCount, missCount, evictionCount; // 缓存的命中、失效和替换次数
unsigned long lruCount; // 缓存的 LRU 计数器，用于实现替换策略
```

以及，我们的缓存本体$simCache$：

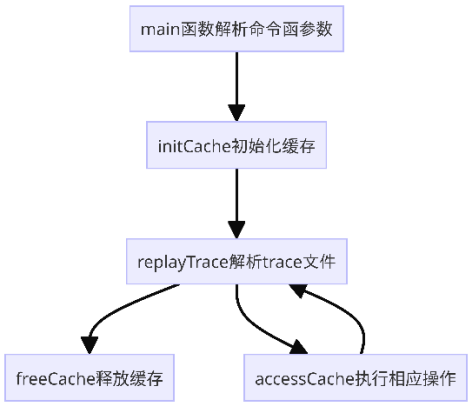
```
cache simCache; // 模拟的缓存
```

4. 函数

我在主函数之外，主要定义了这样一些函数：

```
// 初始化缓存，根据命令行参数分配内存空间，并将所有缓存行的有效位设为 0，LRU 计数
// 器设为 0
void initCache(void);
// 释放缓存，释放分配的内存空间
void freeCache(void);
// 访问缓存，根据给定的地址判断是否命中缓存，如果命中则更新 LRU 计数器，如果不命中
// 则从内存中加载数据，并可能发生替换
void accessCache(unsigned long address);
// 回放 trace 文件，根据给定的 trace 文件模拟缓存的访问过程，并记录缓存的统计信息
void replayTrace(void)
// 打印帮助信息，显示程序的用法和选项
void printUsage(char *argv[]);
```

他们有这样的调用关系：$replayTrace$内部会调用$accessCache$，然后，在主函数中依次调用$initCache, replayTrace, freeCache$。也就是说，总体的程序流程如下：



下面我简要介绍一下这些函数：

4.1 `void initCache(void);`

这个函数没有参数，也没有返回值，它是我们在主函数中最先调用的，用于初始化模拟缓存，根据命令行参数分配内存空间，并将所有缓存行的有效位和LRU计数器设为0。

具体实现如下：

首先分配内存空间，回顾我们定义的$cache$，这是一个嵌套数组，每一个元素都是$cacheLine$的指针，我们先分配S个指针的空间，再分别给每一个指针分配它指向的$cacheLine$数组的空间，每一个都是E个$cacheLine$。

```
// 分配数组的内存空间
simCache = (cacheLine **) malloc(sizeof(cacheLine *) * S);
for (int i = 0; i < S; i++)
{
    simCache[i] = (cacheLine *) malloc(sizeof(cacheLine) * E);
}
```

然后，把所有行的有效位和lru设为0，用二重循环实现：

```
// 将所有缓存行的有效位设为 0，LRU 计数器设为 0
for (int i = 0; i < S; i++)
{
    for (int j = 0; j < E; j++)
    {
        simCache[i][j].valid = 0;
        simCache[i][j].lru = 0;
    }
}
```

4.2 `void freeCache(void);`

这个函数没有参数，也没有返回值，它是我们在缓存工作结束后调用的，用于释放为缓存分配的内存空间。与分配对应的，我们先释放$cache$数组中每一个指针指向的空间，再释放$cache$本身。

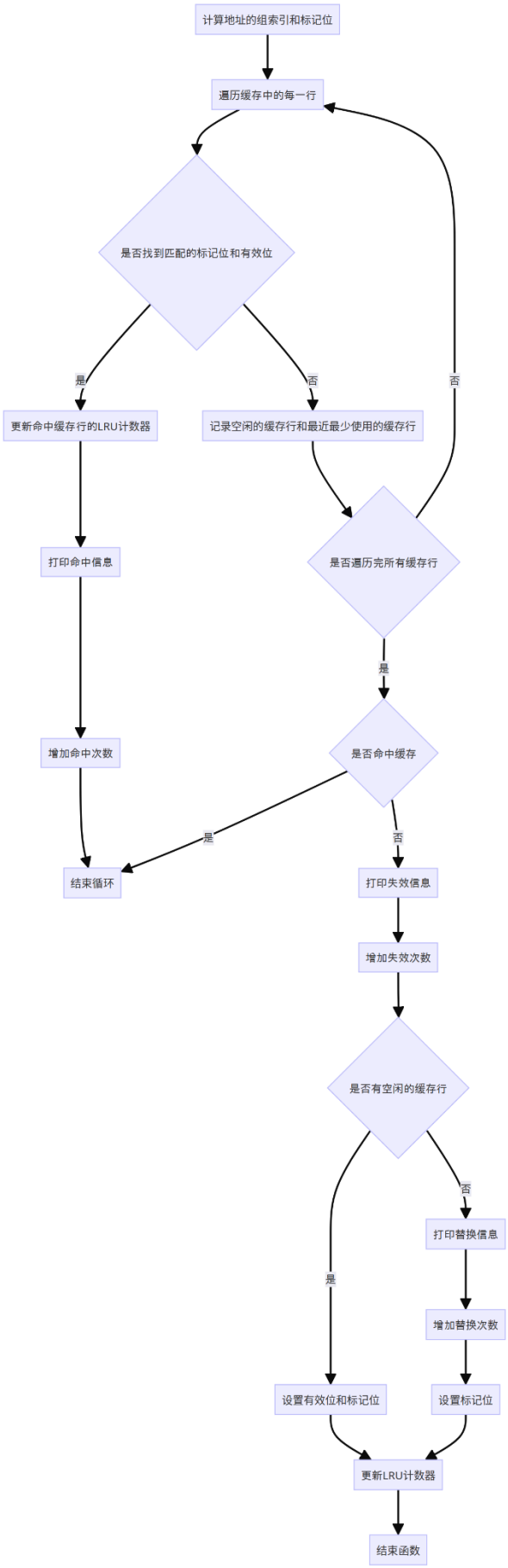
实现如下：

```
// 释放数组的内存空间
for (int i = 0; i < S; i++)
{
    free(simCache[i]);
}
free(simCache);
```

4.3 `void accessCache(unsigned long address);`

这个函数没有返回值，参数是要访问的地址，用于访问缓存，根据给定的地址判断是否命中，如果命中则更新LRU计数器，如果不命中则从内存中加载数据，并可能发生替换。

流程图如下（$verbose$情况）：


计算地址的组索引和标记位
遍历缓存中的每一行
是否找到匹配的标记位和有效位
是
否
更新命中缓存行的LRU计数器
记录空闲的缓存行和最近最少使用的缓存行
否
打印命中信息
是否遍历完所有缓存行
是
增加命中次数
是否命中缓存
是
否
结束循环
打印失效信息
增加失效次数
是否有空闲的缓存行
否
打印替换信息
是
增加替换次数
设置有效位和标记位
设置标记位
更新LRU计数器
结束函数

首先，我们要解析地址，也就是计算地址的组索引和标记位。地址是这种结构：

$tag(m-s-b\ bits)$	$setIndex\ (s\ bits)$	$blockOffset\ (b\ bits)$

我们需要的是组索引和标记位。值得注意的是，我们把地址的类型设为了$unsigned\ long$，所以它的右移执行的是逻辑右移，左侧补0。对于最左边的标记位，我们只要把地址右移$(s+b)$位，就可以得到。对于中间的组索引，我们首先把地址右移b位，去掉块偏移，然后构造一个掩码，让我们能够取出右移后的低s位，这样的掩码是就是低s位都为1，其他都为0，也就是$(1\ <<\ s)-1$，然后按位取并即可：

```
// 计算地址的组索引和标记位
unsigned long setIndex = (address >> b) & ((1 << s) - 1);
unsigned long tag = address >> (s + b);
```

我们已经知道了在哪一组，然后，我们需要遍历该组缓存中的每一行，查找是否有匹配的标记位和有效位。我们维护一些标志如下：

```
// 遍历缓存中的每一行，查找是否有匹配的标记位和有效位
int hit = 0; // 是否命中的标志
int emptyLine = -1; // 空闲的缓存行的索引，初始为-1
int lruLine = 0; // 最近最少使用的缓存行的索引，初始为 0
unsigned long minLRU = simCache[setIndex][0].lru; // 最小的 LRU 计数器的
值，初始为第 0 行的值
```

我们用循环遍历每一行：

```
for (int i = 0; i < E; i++)
```

如果我们找到了标记位匹配，且有效位为1的行，说明缓存命中，我们把刚才定义的命中标志hit设为1，同时更新缓存行的lru，总体的LRU计数器以及总的命中次数计数器，同时结束遍历。如果开启了详细模式，也就是全局变量$verbose$非0 (我们会在主函数中实现)，我们打印"hit "。

```
// 如果找到匹配的标记位和有效位，说明命中缓存
if (simCache[setIndex][i].tag == tag && simCache[setIndex][i].valid
== 1)
{
    hit = 1;
    // 更新命中缓存行的 LRU 计数器
    simCache[setIndex][i].lru = ++lruCount;
    // 如果是详细模式，打印命中信息
    if (verbose)
    {
        printf("hit ");
    }
    // 增加命中次数
    hitCount++;
    // 结束循环
    break;
}
```

考虑到没有命中的情况，我们需要记录两个东西，一个是如果有空行的情况中，我们要找到第一个空行用于存放（为了找到第一个空行，我们最多更新$emptyLine$一次）；

另外一个是考虑到可能需要驱逐，我们要记录最小的lru以及它在哪一行，这里采用了简单的循环比较。

```
// 如果没有找到匹配的标记位和有效位，记录空闲的缓存行和最近最少使用的缓存行
else
{
    // 如果遇到有效位为 0 的缓存行，说明该行是空闲的
    if (simCache[setIndex][i].valid == 0)
    {
        // 记录第一个遇到的空闲缓存行的索引
        if (emptyLine == -1)
        {
            emptyLine = i;
        }
    }
    // 如果遇到 LRU 计数器小于当前最小值的缓存行，说明该行是最近最少使用的
    if (simCache[setIndex][i].lru < minLRU)
    {
        // 记录最近最少使用的缓存行的索引和 LRU 计数器的值
        lruLine = i;
        minLRU = simCache[setIndex][i].lru;
    }
}
```

这样，我们就已经处理好了hit的情况，并且已经为$miss, eviction$做好了准备，即空行（如果有）和最小lru行。那么下面我们处理未hit的情况。这种情况下，我们刚才建立的hit标志应该为0。首先，我们更新$miss$计数器，如果是详细模式，则打印"$miss$ "。

```
if (hit == 0)
{
    // 如果是详细模式，打印失效信息
    if (verbose)
    {
        printf("miss ");
    }
    // TODO: 对两种情况采取相应操作
    // 增加失效次数
    missCount++;
}
```

然后，我们需要存放这次的数据，那么首先我们先判断有没有空行，根据上一步操作，如果$emptyLine$仍然是初始的-1，就代表没有空行，否则有空行。有空行的情况下，我们把这次的tag放到空行里面，并把有效位设为1，更新lru和$lruCount$。

```
// 如果有空闲的缓存行，将数据加载到该行
if (emptyLine != -1)
{
    // 设置有效位和标记位
    simCache[setIndex][emptyLine].valid = 1;
```

```
    simCache[setIndex][emptyLine].tag = tag;
    // 更新 LRU 计数器
    simCache[setIndex][emptyLine].lru = ++lruCount;
}
```

如果没有空行，那我们就要根据LRU策略进行驱逐。我们刚才已经找到了lru最小的一行，只要在这一行放入当前的tag，同时更新$lru, lruCount, evictionCount$。如果是详细模式，打印"$eviction$ "。

```
// 如果没有空闲的缓存行，说明需要替换最近最少使用的缓存行
else
{
    // 如果是详细模式，打印替换信息
    if (verbose)
    {
        printf("eviction ");
    }
    // 增加替换次数
    evictionCount++;
    // 设置标记位
    simCache[setIndex][lruLine].tag = tag;
    // 更新 LRU 计数器
    simCache[setIndex][lruLine].lru = ++lruCount;
}
```

到这里，我们已经处理好了所有情况。

4.4 void replayTrace(void)

这个函数没有返回值，没有参数，用于回放$trace$文件，根据给定的$trace$文件模拟缓存的访问过程，并记录缓存的统计信息。

我们首先用只读模式打开$trace$文件并检查是否正确打开，如果不正确，打印错误信息并退出。

```
// 打开 trace 文件
FILE *trace = fopen(traceFile, "r");
// 检查文件是否打开成功
if (trace == NULL)
{
    fprintf(stderr, "%s: %s\n", traceFile, "No such file or
directory");
    exit(1);
}
```

根据$trace$文件的内容，我们维护这样一些变量存储它一行的内容:

```
// 定义 trace 文件中的操作类型、地址和大小
char operation;
unsigned long address;
int size;
```

然后我们用$fscanf$逐行读取$trace$文件，根据$trace$文件的格式，例如某一行可能是

L 10,4，操作类型是一个字符，它前面可能有空格，我们在%c前面加上空格。之后按它的格式读取地址和大小即可。$fscanf$的返回值是它成功修改了几个变量，我们用这一点判断是否读完。

```
// 逐行读取 trace 文件中的内容
while (fscanf(trace, " %c %lx,%d", &operation, &address, &size) == 3)
```

如果是详细模式，我们需要先打印这一行的信息，并且操作结束后换行。而这一行的$hit, miss, evition$由$accessCache$打印。

```
// 如果是详细模式，打印操作类型、地址和大小
if (verbose)
{
    printf("%c %lx,%d ", operation, address, size);
}
// TODO: 执行某一行对应的操作
// 如果是详细模式，换行
if (verbose)
{
    printf("\n");
}
```

然后，我们解析操作类型，根据要求，我们要处理的是L, S, M，忽略I。其中，数据加载L和数据存储S都是访问一次缓存，而数据修改M相当于先加载再存储，需要访问两次缓存。

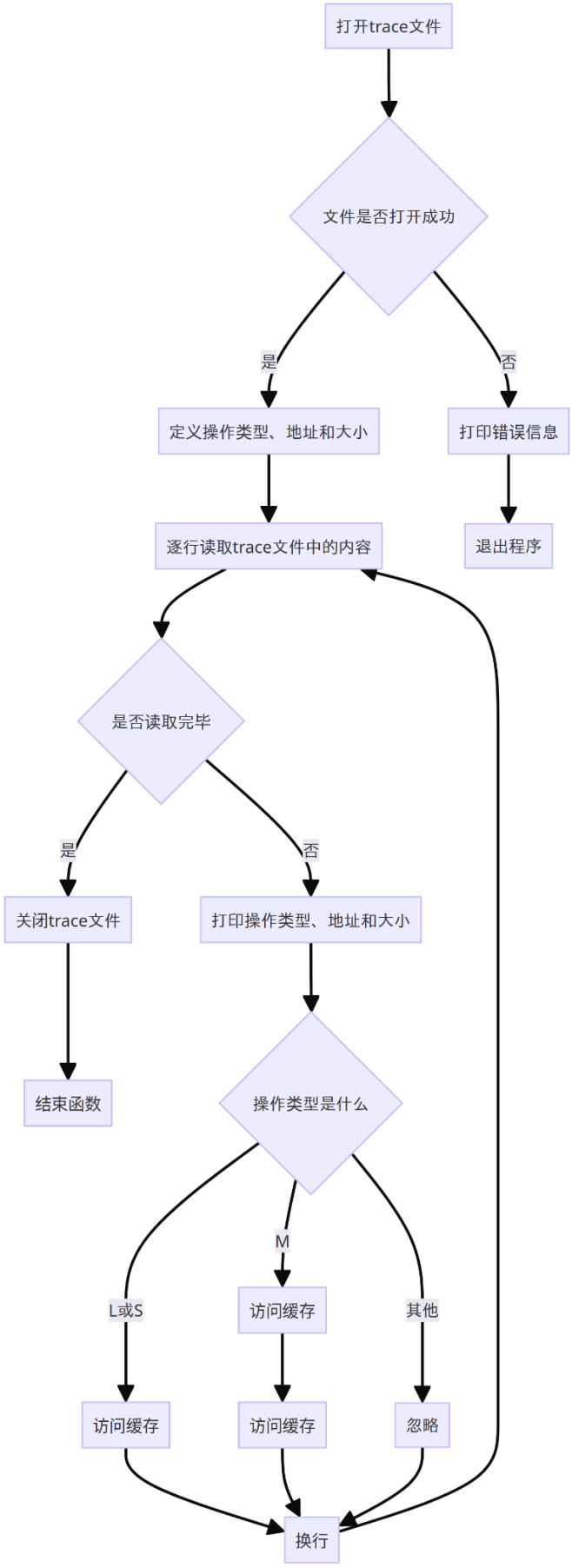
```
// 根据操作类型，执行相应的缓存访问
switch (operation)
{
    case 'L': // 数据加载
    case 'S': // 数据存储
        accessCache(address);
        break;
    case 'M': // 数据修改，相当于先加载后存储
        accessCache(address);
        accessCache(address);
        break;
    default: // 其他操作，忽略
        break;
}
```

最后需要关闭我们打开的$trace$文件：

```
// 关闭 trace 文件
fclose(trace);
```

到这里，我们完成了对$trace$文件的解析和操作。

这个函数的流程图如下（$verbose$情况）：


打开trace文件
文件是否打开成功
是
否
定义操作类型、地址和大小
打印错误信息
逐行读取trace文件中的内容
退出程序
是否读取完毕
是
否
关闭trace文件
打印操作类型、地址和大小
结束函数
操作类型是什么
M
L或S
其他
访问缓存
访问缓存
忽略
访问缓存
换行

4.5 `void printUsage(char *argv[]);`

这个函数没有返回值，参数是命令行参数$argv$，用于打印帮助信息，显示程序的用法和选项，我们用$argv[0]$得到当前程序的位置显示在输出信息中。

```
printf("Usage: %s [-hv] -s <s> -E <E> -b <b> -t <trace_file>\n",
argv[0]);
printf("Options:\n");
printf("  -h        Print this help message.\n");
printf("  -v        Optional verbose flag.\n");
printf("  -s <num>   Number of set index bits.\n");
printf("  -E <num>   Number of lines per set.\n");
printf("  -b <num>   Number of block offset bits.\n");
printf("  -t <file>  Trace file.\n");
printf("\nExamples:\n");
printf("  linux>  %s -s 4 -E 1 -b 4 -t traces/dave.trace\n",
argv[0]);
printf("  linux>  %s -v -s 8 -E 2 -b 4 -t traces/dave.trace\n",
argv[0]);
```

5. 主函数

有了前面的五个函数，我们下面可以开始编写主函数。我们需要解析命令行参数，初始化缓存，回放$trace$文件，释放缓存，打印缓存的统计信息。

首先，我们需要用$getopt$函数解析命令行参数。我们定义$option$用于存储$getopt$函数的返回值，表示找到的选项字符。使用$while$循环调用$getopt$函数，传入$main$函数接收的参数$argc$和$argv$，表示命令行参数的个数和内容，以及"$s:E:b:t:hv$"，表示程序支持的选项字符，每个字符代表一个选项。S,E,b,t后面必须有参数，所以我们加上冒号，h和v没有参数，我们放在最后。如果没有更多的选项要处理，$getopt$函数会返回-1，循环结束。

在循环中，使用$switch$执行相应的操作。每个$case$分支对应一个选项，如下：

‘s’,‘E’,‘b’：用$strtol$函数，把$getopt$函数提供的参数$optarg$转换成整数并赋值给相应的全局变量。

‘t’：设置$trace$文件的路径，将$optarg$赋给全局变量$trace_file$。

‘v’：设置详细模式，将全局变量$verbose$设为1，表示打印缓存的详细信息。

‘h’：打印帮助信息，调用$printUsage$函数显示程序的用法和选项，然后使用$exit$函数退出程序，返回0表示正常结束。

$default$：处理未知的选项，调用$printUsage$函数，用错误码1异常终止程序。

```
int option;
while ((option = getopt(argc, argv, "s:E:b:t:hv")) != -1)
{
    switch (option)
    {
        case 's':
            s = strtol(optarg, NULL, 10);
            break;
        case 'E':
            E = strtol(optarg, NULL, 10);
```

```
            break;
        case 'b':
            b = strtol(optarg, NULL, 10);
            break;
        case 't':
            traceFile = optarg;
            break;
        case 'v':
            verbose = 1;
            break;
        case 'h':
            printUsage(argv);
            exit(0);
        default:
            printUsage(argv);
            exit(1);
    }
}
```

在解析完命令行参数后，我们需要判断是否合法，如果不合法，我们打印提示信息并且用错误码1退出程序。理论上来说，s和b可以是0，表示只有一组或一块，而E必须是一个正整数，我们把s和b初始化为-1，E则是0。

```
// 检查命令行参数是否完整
if (s == -1 || E == 0 || b == -1 || traceFile == NULL)
{
    printf("%s: Missing required command line argument\n", argv[0]);
    printUsage(argv);
    exit(1);
}
```

然后根据$S = 2^s, B = 2^b$，计算S, B

```
// 计算 S 和 B 的值
S = 1 << s;
B = 1 << b;
```

下面依次调用我们定义的函数:

```
// 初始化缓存
initCache();

// 回放 trace 文件
replayTrace();

// 释放缓存
freeCache();
```

最后按要求输出$hit, miss, eviction$的数量

```
// 打印缓存的统计信息
printSummary(hitCount, missCount, evictionCount);
```

到此，我们完全完成了任务。

6. 总结

这个实验极大强化了我对缓存的认识，包括缓存的结构(S,E,B),地址的结构$(tag,setIndex,blockOffset)$，缓存的匹配方式，$miss$情况的处理，$LRU$的驱逐策略等。在术的层面，学习了如何在程序中利用命令行参数以及$getopt$的函数的使用，复习了$fopen,fclose,fscanf$等文件操作函数，特别是$fscanf$的返回值。在道的层面，体会了面对一个工程项目，如何统筹规划，如何分解问题，面对不知道的东西怎么处理。总的来说，这次实验是系统编程的简单初体验，让我收获颇丰。