

**计算机系统结构实验报告**

|  |  |
| --- | --- |
| 姓 名： | 李一帆 |
| 学 院： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 班 级： | 校交1802班 |
| 学 号： | U201811495 |
| 指导教师： | 万继光 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2020年4月28日

**目 录**

[1. Cache模拟器实验 3](#_Toc23963)

[1.1. 实验目的 3](#_Toc3705)

[1.2. 实验环境 3](#_Toc865)

[1.3. 实验思路 3](#_Toc25045)

[1.4. 实验结果和分析 3](#_Toc21474)

[2. 总结和体会 3](#_Toc18091)

[3. 对实验课程的建议 3](#_Toc32753)

# Cache模拟器实验

## 实验目的

1. 理解Cache工作原理
2. 如何实现一个高效的模拟器

## 实验内容

编写一个200-300行的C程序来模拟Cache缓存的行为

## 实验环境

系统：Linux 64-bit

开发语言：c语言

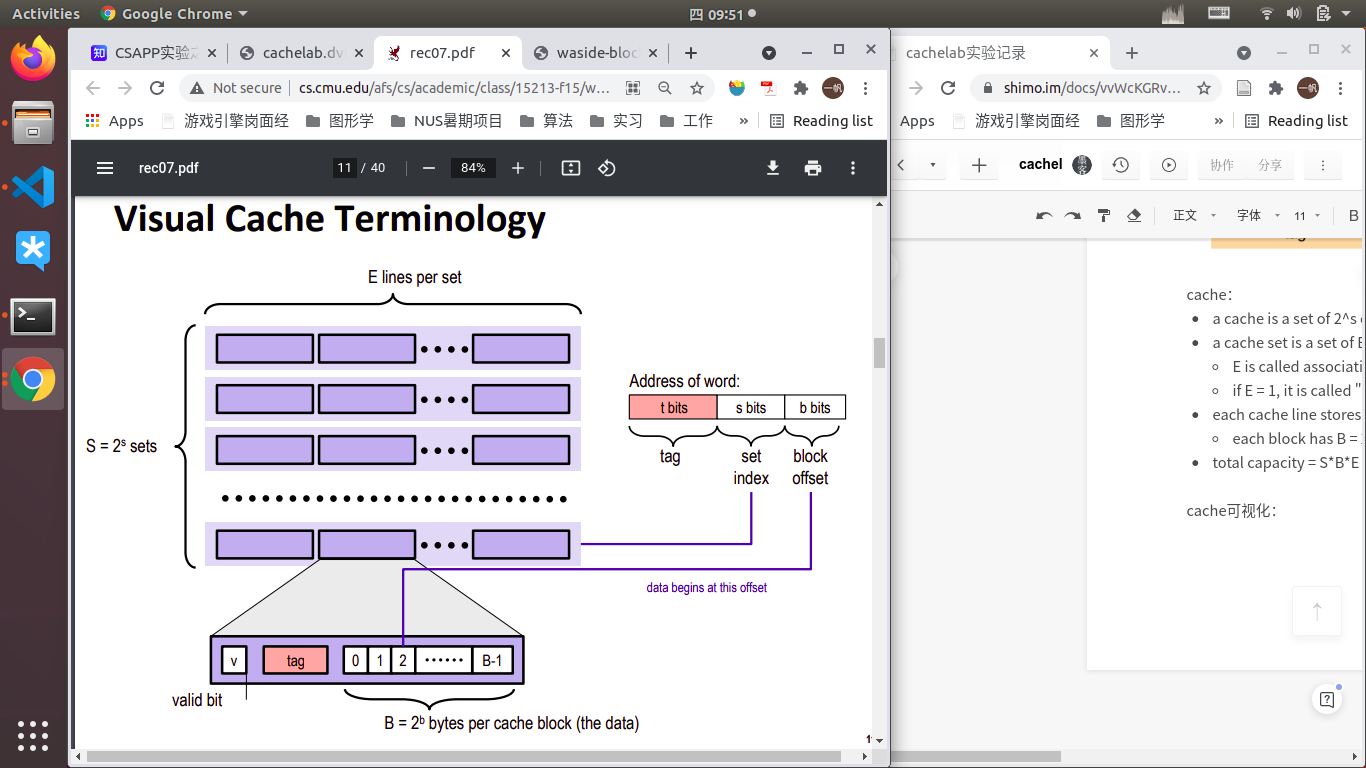
实验平台：阿里云服务器

## 实验思路

本次实验需要我们模拟一个Cache，首先需要理解Cache的结构、Cache访存及相关算法。

### 1.4.1 Cache的结构

Cache的结构：



这张图是我从CMU的PPT中找出的一个描述Cache的结构的图片。

块(block/cache line)是组成Cache的最小单位，许多（E个）块组成block set，许多（S个）block set组成Cache。每个block/cache line由 tag|set index|block offset组成。

因为本次实验要支持不同结构即不同S和E的Cache，需要通过malloc来动态分配cache的内存。实验中因只要模拟访存的路径，不需要真正的访问主存或者修改数据，因此不需要分配cache line的数据区。

定义结构体cache\_line，包含valid和tag，分别表示当前cache行是否有效和其标记：

typedef struct cache\_line {

char valid;

mem\_addr\_t tag;

} cache\_line\_t;

用typedef 定义cacle\_set\_t为指向一个cache set的指针；

用typedef 定义cache\_t 为指向cache的指针（指向一组cache set）。

### 1.4.2 Cache访存

Cache访存的策略主要有两种：

一、放置策略：决定block放在cache中哪个块；

本次实验设计中通过一个mask来决定block的放置位置。

首先通过set\_index\_mask 找到block对应的cache set，逐个判断其中的cache line 是否有效。

若有效且tag与要放置的block相同（访存命中），说明该block已经在cache line中了，表示访存命中，可以直接从cache中读取数据。

否则需要从该set中找到一个位置放置这个block（访存未命中）。首先遍历当前cache set查看是否有无效的cache line（即valid为0），若有，则将block放置在这个位置；若所有的行都是valid的，则需要根据替换策略替换某个cache行。

二、替换策略：决定哪个block要被替换。

本次实验中使用LRU算法作为cache的替换策略。

通过定义一个双向链表来记录每个cache set的访问时间。

与Cache的结构体定义类似，首先定义包含cache line下标的LRU\_line为LRU的结点：

typedef struct LRU\_line {

int index; // set中的block index

struct LRU\_line\* prev;

struct LRU\_line\* next;

} LRU\_line\_t;

用typedef定义LRU\_set\_t为指向LRU\_line\_t的指针；

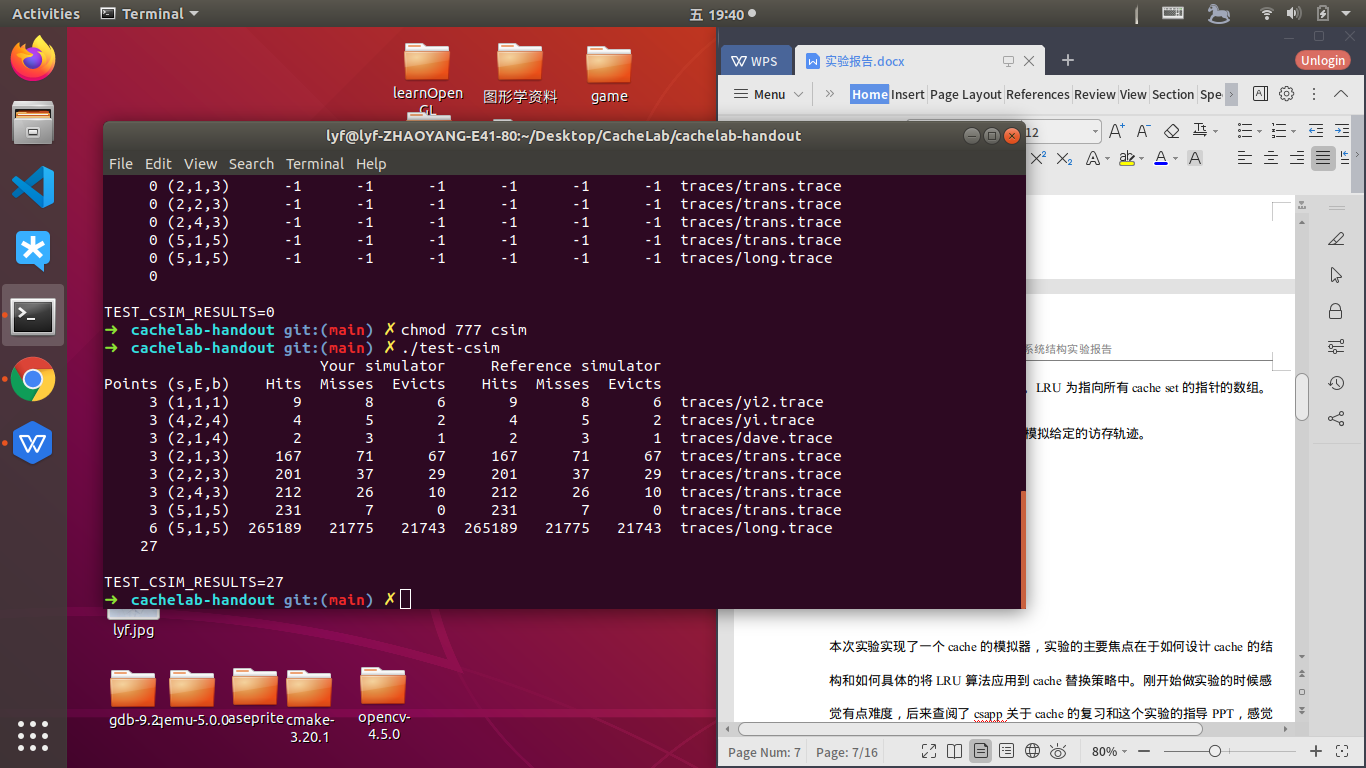
用typedef定义LRU\_t为指向LRU\_set\_t的指针。

每次访存，若命中某个cache set的某个cache行，将该cache行对应的LRU结点放入这个cache set的LRU链表的表头。若未命中并且找到了一个invalid的cache行，新建一个该cache行对应的LRU结点并放入LRU链表的表头。若未命中且所有的行都是valid的，则将LRU链表表尾的结点重新放入表头。

### 1.4.3 其他实验细节

1. 通过getopt函数处理命令行参数
2. 主函数逻辑：若所有必须的命令行参数已指定，初始化Cache和LRU链表，分析（模拟）trace file文件，释放分配的空间，输出命中和未命中相关信息。
3. 初始化cache：给cache和LRU分配内存，将cache line的valid和tag初始值都设置为0，计算set index的mask。LRU为指向所有cache set的指针的数组。
4. 解析trace file文件：在cache上模拟给定的访存轨迹。

## 实验结果和分析



实验测试成功！

# 总结和体会

本次实验实现了一个cache的模拟器，实验的主要焦点在于如何设计cache的结构和如何具体的将LRU算法应用到cache替换策略中。刚开始做实验的时候感觉有点难度，后来查阅了csapp关于cache的复习和这个实验的指导PPT，感觉茅塞顿开，还是相对比较好完成的。

# 对实验课程的建议

实验内容较少，与计算机系统结构本学期理论课程的联系感觉上不是特别紧密。感觉计算机系统基础或者组成原理课也可以做这个实验。

# 附录

核心代码：

#include <getopt.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <assert.h>

#include <math.h>

#include <limits.h>

#include <string.h>

#include <errno.h>

#include "cachelab.h"

//#define DEBUG\_ON

// 内存地址

typedef unsigned long long int mem\_addr\_t;

// Cache 行

typedef struct cache\_line {

char valid;

mem\_addr\_t tag;

} cache\_line\_t;

typedef cache\_line\_t\* cache\_set\_t; // 指向一个set的地址

typedef cache\_set\_t\* cache\_t; // 指向cache的地址

// cache 模拟器

cache\_t cache;

mem\_addr\_t set\_index\_mask;

// LRU node

typedef struct LRU\_line {

int index; // set中的block index

struct LRU\_line\* prev;

struct LRU\_line\* next;

} LRU\_line\_t;

typedef LRU\_line\_t\* LRU\_set\_t; // 指向一个set的LRU链表的地址

typedef LRU\_set\_t\* LRU\_t;

// LRU 模拟

LRU\_t LRU;

// 通过命令行设置的参数

int verbosity = 0; // 是否输出 trace

int s = 0; // set index bits

int b = 0; // block offset bits

int E = 0; // associativity

char\* trace\_file = NULL;

// 由命令行参数计算的参数

int S; // number of sets

int B; // block size (bytes)

// 保存 cache 记录的计数器

int miss\_count = 0;

int hit\_count = 0;

int eviction\_count = 0;

/\*

\* init 初始化cache，分配内存，初始值设为0；初始化lru链表

\* 同时计算 set\_index\_mask

\*/

void initCache() {

// init cache

cache = (cache\_set\_t\*) malloc(sizeof(cache\_set\_t) \* S);

for (int i = 0; i < S; i++){

cache[i] = (cache\_line\_t\*) malloc(sizeof(cache\_line\_t) \* E);

for (int j = 0; j < E; j++){

cache[i][j].valid = 0;

cache[i][j].tag = 0;

}

}

// 计算 set index mask

set\_index\_mask = (mem\_addr\_t) (pow(2, s) - 1);

// init LRU

LRU = (LRU\_set\_t\*) malloc(sizeof(LRU\_set\_t) \* S);

for (int i = 0; i < S; i++) {

LRU[i] = (LRU\_line\_t\*)malloc(sizeof(LRU\_line\_t)); // head

}

}

/\*

\* freeCache - 释放分配的内存

\*/

void freeCache() {

for(int i = 0; i < S; i++) {

free(cache[i]);

LRU\_line\_t\* p = LRU[i];

while(p != NULL) {

LRU\_line\_t\* tmp = p;

p = p->next;

free(tmp);

}

}

free(cache);

free(LRU);

}

/\*

\* 将第index个block对应的lru节点放入链表头部

\*/

void getToHead(int set\_index, int index) {

LRU\_line\_t\* lruP = LRU[set\_index];

if(lruP->next != NULL && lruP->next->index == index) { // 已经在第一个

return;

}

while(lruP->next != NULL) {

lruP = lruP->next;

if(lruP->index == index) { // 找到对应的lru节点

lruP->prev->next = lruP->next;

if(lruP->next != NULL) {

lruP->next->prev = lruP->prev;

}

lruP->next = LRU[set\_index]->next;

lruP->prev = LRU[set\_index];

LRU[set\_index]->next->prev = lruP;

LRU[set\_index]->next = lruP;

break;

}

}

}

/\*

\* 将LRU表尾的节点放到链表头部（即清除数据再放入新数据），返回这个节点对应的index

\*/

int popTailToHead(int set\_index) {

LRU\_line\_t\* lruP = LRU[set\_index];

while(lruP->next != NULL) { // 找到表尾节点

lruP = lruP->next;

}

// 放入头部

lruP->prev->next = NULL;

lruP->next = LRU[set\_index]->next;

lruP->prev = LRU[set\_index];

if(LRU[set\_index]->next != NULL) {

LRU[set\_index]->next->prev = lruP;

}

LRU[set\_index]->next = lruP;

return lruP->index;

}

/\*

\* 插入新的节点到链表头部

\*/

void insertToHead(int set\_index, int index) {

LRU\_line\_t\* newNode = (LRU\_line\_t\*)malloc(sizeof(LRU\_line\_t));

newNode->index = index;

if(LRU[set\_index]->next != NULL) {

LRU[set\_index]->next->prev = newNode;

}

newNode->next = LRU[set\_index]->next;

LRU[set\_index]->next = newNode;

newNode->prev = LRU[set\_index];

}

/\*

\* accessData - 访问给定主存地址的数据

\* 如果已经在 cache 中，命中，增加 hit\_count

\* 如果不在 cache 中，未命中，放入 cache，增加 miss\_count

\* 如果这行 evicted，增加eviction\_count

\*/

void accessData(mem\_addr\_t addr) {

mem\_addr\_t set\_index = (addr >> b) & set\_index\_mask;

mem\_addr\_t tag = addr >> (s + b);

cache\_set\_t cache\_set = cache[set\_index]; // 取出对应的 set

// 判断是否命中

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

// 命中

if (cache\_set[i].valid && cache\_set[i].tag == tag) {

if (verbosity)

printf("hit ");

++hit\_count;

getToHead(set\_index, i); // 将其lru节点放入表头

return;

}

}

// 未命中

if (verbosity)

printf("miss ");

++miss\_count;

// 找出最大lru的block 或者第一个非法行

int j;

for (j = 0; j < E; ++j) {

if(cache\_set[j].valid != 1) { // 找到第一个invalid的行

cache\_set[j].valid = 1;

cache\_set[j].tag = tag;

insertToHead(set\_index, j); // 新建lru节点放入表头

break;

}

}

if (j == E) { // 所有的行都是valid的，需要替换 lru 最大的，即 lru 表尾的

if (verbosity)

printf("eviction ");

++eviction\_count;

int index = popTailToHead(set\_index);

cache\_set[index].tag = tag;

cache\_set[index].valid = 1;

}

}

/\*

\* parseTrace - 在cache上模拟给定的访存轨迹

\*/

void parseTrace(char\* trace\_fn) {

char ch;

mem\_addr\_t addr = 0;

unsigned int size = 0;

FILE\* trace\_fp = fopen(trace\_fn, "r"); // 打开 trace 文件

if(trace\_fp == NULL) {

printf("file open error");

exit(1);

}

while (fscanf(trace\_fp, " %c %llx,%d", &ch, &addr, &size) > 0) {

if (verbosity && ch != 'I')

printf("%c %llx,%d ", ch, addr, size);

switch (ch) {

case 'I':

break;

case 'L':

case 'S':

accessData(addr);

break;

case 'M': // 两次访存

accessData(addr);

accessData(addr);

break;

default:

break;

}

if (verbosity && ch != 'I')

putchar('\n');

}

fclose(trace\_fp);

}

/\*

\* printUsage - 打印 usage 信息

\*/

void printUsage(char\* argv[]) {

printf("Usage: %s [-hv] -s <num> -E <num> -b <num> -t <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/yi.trace\n", argv[0]);

printf(" linux> %s -v -s 8 -E 2 -b 4 -t traces/yi.trace\n", argv[0]);

exit(0);

}

/\*

\* main

\*/

int main(int argc, char\* argv[]) {

char opt;

while( (opt = getopt(argc, argv, "s:E:b:t:vh")) != -1){

switch(opt){

case 's':

s = atoi(optarg);

break;

case 'E':

E = atoi(optarg);

break;

case 'b':

b = atoi(optarg);

break;

case 't':

trace\_file = optarg;

break;

case 'v':

verbosity = 1;

break;

case 'h':

printUsage(argv);

exit(0);

default:

printUsage(argv);

exit(1);

}

}

// 保证所有必须的命令行参数已指定

if (s == 0 || E == 0 || b == 0 || trace\_file == NULL) {

printf("%s: Missing required command line argument\n", argv[0]);

printUsage(argv);

exit(1);

}

// 根据命令行输入的参数计算 S 和 B

S = (unsigned int) pow(2, s);

B = (unsigned int) pow(2, b);

// 初始化 cache

initCache();

#ifdef DEBUG\_ON

printf("DEBUG: S:%u E:%u B:%u trace:%s\n", S, E, B, trace\_file);

printf("DEBUG: set\_index\_mask: %llu\n", set\_index\_mask);

#endif

parseTrace(trace\_file);

// // 释放分配的空间

freeCache();

// 输出命中和未命中相关信息

printSummary(hit\_count, miss\_count, eviction\_count);

return 0;

}