CSAPP 实验之 cache lab

预备内容

首先，了解 getopt（）函数的功能和使用方法。输入 linux> man 3 getopt，查看 getopt（）函数的介绍。它需要包含这三个头文件#include <getopt.h> 、#include <stdlib.h> 、#include <unistd.h> 。一般形式如下：

```
int getopt(int argc, char * const argv[], const char *optstring);
```

返回整型。如果选项成功找到，返回选项字母；如果所有命令行选项都解析完毕，返回 -1；如果遇到选项字符不在 optstring 中，返回字符 '?'；如果遇到丢失参数，那么返回值依赖于 optstring 中第一个字符，如果第一个字符是':' 则返回':'，否则返回'?'并提示出错误信息。

三个参数：argc、argv、optstring。其中 argc 和 argv 与 main 函数的两个参数是一致的。Optstring，表示我们短选项的字符串。带冒号的字符，表示选项后面带一个参数，不带冒号的字符，表示选项，带两个冒号的字符，表示选项可有可无。

```
extern char *optarg;extern int optind, opterr, optopt;
```

4 个全局变量 optarg，optind, opterr, optopt。Optarg，是当前选项对应的参数值。optind，是 argv 中要处理的下一个元素的索引。opterr，是在默认情况下，opterr 有一个非零值，opterr 设置为零，那么 getopt()不会打印错误消息。optopt，是表示错误选项字符。

了解 malloc()函数的功能和使用方法。输入 linux> man malloc，查看 malloc（）函数的介绍。

Part A: Writing a Cache Simulator

基本架构-接收命令行参数

首先，利用 getopt（）函数解析命令行。设置 optstring 为"sEbt:hv"，根据 getopt（）函数返回的字符，利用 Switch-case 语句进行分解。如果是 h，就输出提示信息，并正常退出；如果是 v，显示跟踪信息的可选详细标志，即设置一个标签表示；如果是 s，就是高速缓存组 S=2^s,argv[optind]里存储的就是字符 s，利用 atoi（）函数，进行字符转十进制，并存到变量 s 中；如果是 E，就是高速缓存行，同样利用 atoi（）函数，,argv[optind]里存储的进行字符转十进制，并存到变量 E 中；如果是 b，就是数据块 B=2^b，同样利用 atoi（）函数，,argv[optind]里存储的进行字符转十进制，并存到变量 b 中；如果是 t，就获取文件名 argv[optind-1]，并打开文件；如果是其他的，就输出提示信息并异常退出

```
//"sEbt:hv"：-s、-E、-b、-t 可接收字符，-h 和-v 不 x 接受
while ((opt = getopt(argc, argv, "sEbt:hv")) != -1) {
    switch (opt) {
```

```
            case 'h':
                fprintf(stderr, help,argv[0],argv[0],argv[0]);   //argv[0]是命令行参数
的第一个值，即运行的可执行文件名
                exit(0）;
            case 'v':
                tag_v = 1;
                break;
            case 's':
                s=atoi(argv[optind]);     //atoi 字符转十进制
                break;
            case 'E':
                E=atoi(argv[optind]);
                break;
            case 'b':
                b=atoi(argv[optind]);
                break;

            case 't':
                file_name=argv[optind-1];        //获取文件名

                file = fopen(file_name,"r");   //打开文件
                if(file == NULL){
                    perror(file_name);
                    exit(EXIT_FAILURE);
                }
                break;

            default:
                fprintf(stderr, help,argv[0],argv[0],argv[0]);
                exit(EXIT_FAILURE);
```

定义提示信息的字符串。

```
char help[]="Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n"
            "Options:\n"
            "-h                 Print this help message.\n"
            "-v                 Optional verbose flag.\n"
            "-s <num>       Number of set index bits.\n"
            "-E <num>       Number of lines per set.\n"
            "-b <num>       Number of block offset bits.\n"
            "-t <file>     Trace file.\n\n"
            "Examples:\n"
            "linux>         %s    -s 4 -E 1 -b 4 -t traces/yi.trace\n"
            "linux>         %s    -v -s 8 -E 2 -b 4 -t traces/yi.trace\n";
```

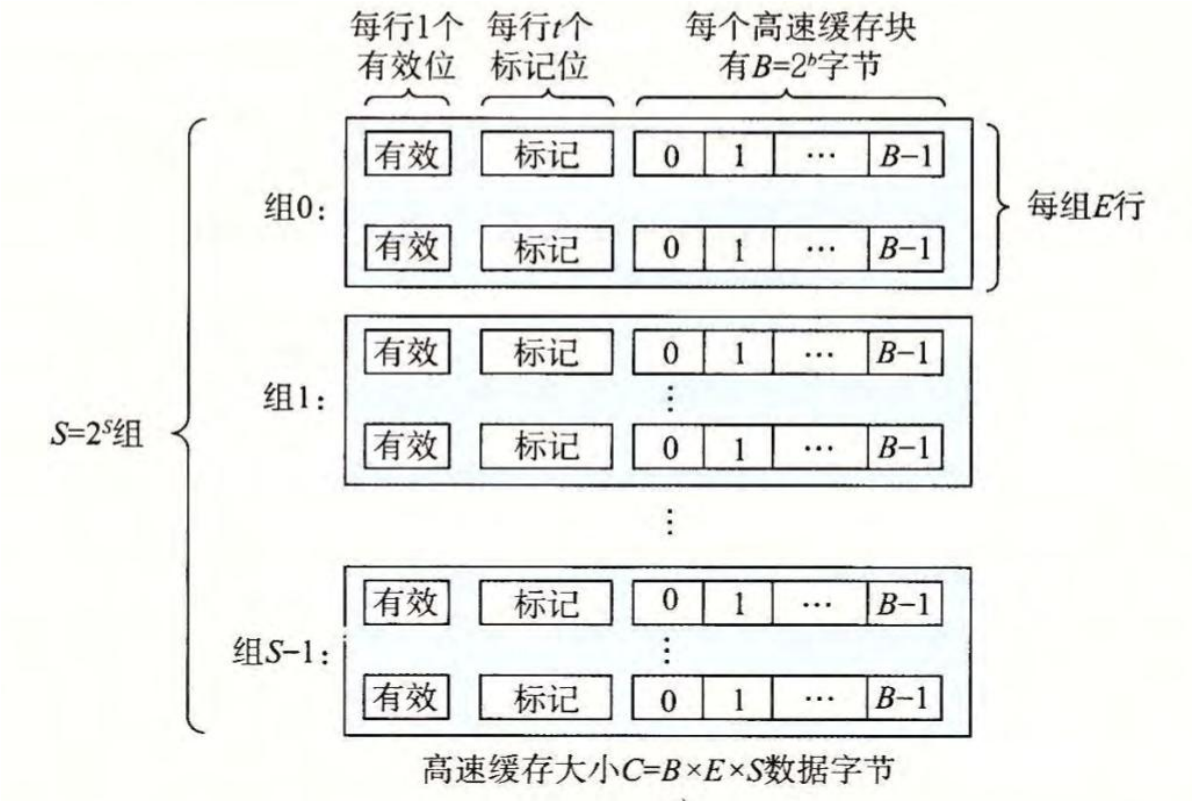
设计模拟 LRU 高速缓存结构

可以用一个链表来实现 LRU 替换策略。链头是最近访问的 Cache。链尾是最后访问的缓存区。当一个位置被命中之后，通过调整链表的指向，将该位置调整到链表头的位置，新加入 Cache 直接加到链表头中。这样，在多次进行 Cache 操作后，最近被命中的，就会被向链表头方向移动，而没有命中的，会向链表后面移动，链表尾则表示最近最少使用的 Cache。

```
struct LRU_set{
    struct list *head;     //链表头
    struct row *E_row;     //组
};

struct list{
    struct list *prior;    //指向直接前趋
    struct list *next;     //指向直接后继
    int tag;               //标记
};
```

模拟高速缓存结构



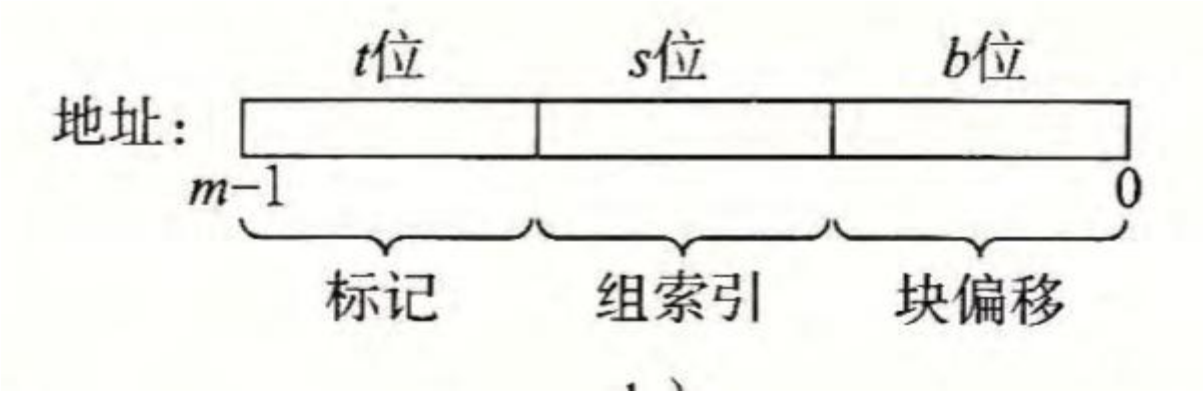
设计一个数据结构，使它能模仿高速缓存结构的行设计一个数据结构，使它能模仿高速缓存结构的行为。高速缓存是一个高速缓存组。每个组包括一个行或

多个行，每个行包含一个有效位，一些标记位，以及一个数据块。可以看出这是一个“组”里面有“行”，行里面有有效位，标记位，以及一个数据块。
可以把有效位，标记位，定义成一个结构体，然后创建该结构体的链表就可以模拟高速缓存组。
通过位运算，获得高速缓存组，并初始化结构。

```
struct row{
    int valid;        //有效位
    int tag;           //标记位
};

  int S = 1 << s;
struct LRU_set cache[S];
    for(int i = 0; i < S; i++){
        cache[i].head = (struct list*)malloc(sizeof(struct list));
        cache[i].head->tag = -1;
        cache[i].E_row = malloc(sizeof(struct row)*E);
        for(int j = 0; j < E; j++){
            cache[i].E_row[j].valid =0 ;
            cache[i].E_row[j].tag = -1;
        }
    }
```

解析地址



计算组索引和标记位。

```
  int set_index = (address >> b) % (1 << s);
        int tag = address >> (s + b);
```

解析文件

每一行的格式为[space]operation address,size

operation 字段表示内存访问的类型：“I” 表示指令加载，“L” 表示数据加载，“S” 表示数据存储，“M” 表示数据修改(即，数据加载后跟着一个数据存储)。在每个 “I” 之前从来没有空格。每个 “M”、“L” 和 “S” 前面总是有一个空格。address 字段指定一个 64 位的十六进制内存地址。size 字段指定操作访问的字节数。用 char *fgets(char *str, int n, FILE *stream);一行一行读取并进行处理。

用 char *strtok(char *str, const char *delim);分割字符串。Str，要被分解成一组小字符串的字符串。Delim，包含分隔符的 C 字符串。该函数返回被分解的第一个子字符串，如果没有可检索的字符串，则返回一个空指针。

要处理 address，因为它是字符的表示形式要转成整数才可以对它的位进行处理。写个 htoi 方法：十六进制字符转成十进制整数。

```
long htoi(char *s){
    long n = 0;
    for(int i = 0; (s[i] >= '0' && s[i] <= '9') || (s[i] >= 'a' && s[i] <= 'f'); i++){
        if(s[i] > '9')
            n = 16 * n + (10 + s[i] - 'a');
        else n = 16 * n + (s[i] - '0');
    }
    return n;
}
```

```
 char ch[16];                              //刚好可以容纳一行数据的大小
    const char Separator[] = " ,";       //分隔符
    char *token;                              //子串
    while(fgets(ch, 16, file) != NULL){
        token = strtok(ch, Separator);
        char *instruction = token;           //内存访问的类型
        if (*instruction == 'I')              //I 类型不用处理
            continue;
        token = strtok(NULL, Separator);
        char *address_char = token;           //地址的字符表示
        long address = htoi(address_char); //地址的十进制表示
        token = strtok(NULL,Separator);
        int size = atoi(token);
        if(v) printf("%s %s,%d ", instruction, address_char, size);
```

模拟缓存行为

遍历行是否有效，在判断标记位是否相同，标记位相同就命中。将其移到表头。没有命中。遍历行是否有无效(空)的缓存区，有就写入，并移到表头。也没有。驱逐表尾一个缓存区，写入并移到表头。

"I"表示指令加载：不用处理。
"L"表示数据加载和"S"表示数据存储：一样处理
"M"数据修改(即，数据加载后跟着一个数据存储)：“L"然后"S”

```
int M = *instruction == 'M' ? 2 : 1;                //'M'  循环两遍,'S''I'一遍
    while(M--){
        int flag = 0;
        struct LRU_set *lru_row = &cache[set_index];     //获取该组
        struct list *head = lru_row->head;         //获取该组的双链表头
        for(int i = 0; i < E; i++){                          //遍历一组中的所有行
            struct row *arow = &lru_row->E_row[i]; //获取一行
            if(arow->valid && arow->tag == tag){        //有效且标记相同则命中
                //找到双链表中命中的缓存，准备放到链表头
                while(head->tag != tag)
                    head = head->next;
                //放到链表头，要判断，是不是本身就是头，是头就跳过
                if (head != lru_row->head){
                    head->prior->next = head->next;
                    //判断是不是链表尾，是尾就跳过
                    if(head->next != NULL)
                        head->next->prior = head->prior;
                    head->prior = NULL;
                    head->next = lru_row->head;
                    lru_row->head->prior = head;
                    lru_row->head = head;
                }
                hit_count++;                                    //命中
                if(v) printf("hit ");
                flag = 1;
                break;
            }
        }
        if (flag){
            if (v && !M) printf("\n");
            continue;
        }

        for(int i = 0;i < E;i++){          //无效：未命中，要写入无效的缓存区
            struct row *arow = &lru_row->E_row[i]; //获取一行
            if(!arow->valid){
                //准备放入链表的新结点
                struct list * temp = (struct list*)malloc(sizeof(struct list));
                temp->prior = NULL;
                temp->next = NULL;
```

```
            temp->tag = tag;
            //判断是不是链表中的第一个结点，是就跳过
            if(head->tag != -1){
                temp->next = head;
                head->prior = temp;
            }
            //设置尾头结点
            lru_row->head = temp;
            miss_count++;                                    //未命中
            arow->valid = 1;
            arow->tag = tag;
            if(v) printf("miss ");
            flag = 1;
                break;
        }
    }
    if (flag){
          if (v && !M) printf("\n");
          continue;
    }
    /*
      又没命中，又没有多余的缓存区，则驱逐。驱逐的是双链表的最后一个
       驱逐就是把双链表最后一个移到开头并重新设置 tag
     */
    while(head->next != NULL)
        head = head->next;
    //判断链表中是不是只有一个结点，是就跳过
    if (head != lru_row->head){
        head->prior->next = NULL;
        head->prior = NULL;
        head->next = lru_row->head;
        lru_row->head->prior = head;
        lru_row->head = head;
    }
    //找到在缓存中对应与双链表最后一个的缓存，移除，写入新的缓存。
    struct row *arow;
    for(int j = 0; j < E; j++){
        arow = &cache[set_index].E_row[j];
        if(head->tag == arow->tag)
            break;
    }
    head->tag = tag;
    arow->tag = tag;
```

```
            if(v) printf("miss eviction");
            miss_count++;              //未命中
            eviction_count++;       //驱逐
            if (v && !M) printf("\n");                    //不是'M'就要换行
        }
    }
```

完整代码

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

extern char *optarg;
extern int optind, opterr, optopt;

//十六进制字符转成十进制整数
long htoi(char *s){
    long n = 0;
    for(int i = 0; (s[i] >= '0' && s[i] <= '9') || (s[i] >= 'a' && s[i] <= 'f'); i++){
        if(s[i] > '9')
            n = 16 * n + (10 + s[i] - 'a');
        else n = 16 * n + (s[i] - '0');
    }
    return n;
}

struct row{
    int valid;          //有效位
    int tag;             //标记位
};

struct LRU_set{
    struct list *head;     //链表头
    struct row *E_row;      //组
};
```

```
struct list{
    struct list *prior;   //指向直接前趋
    struct list *next;    //指向直接后继
    int tag;              //标记
};


int main(int argc, char *argv[]){
    int hit_count = 0, miss_count = 0, eviction_count = 0;   //命中，未命中，驱逐
    int opt;
    int s = 0;          //高速缓存组 S=2^s
    int E = 0;          //高速缓存行 E
    int b = 0;          //数据块 B=2^b
    int v = 0;   //是否查看过程信息
    char *file_name;
    FILE *file;
    //提示(帮助)的信息
    char help[] = "Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n"
                  "Options:\n"
                  "-h              Print this help message.\n"
                  "-v              Optional verbose flag.\n"
                  "-s <num>        Number of set index bits.\n"
                  "-E <num>        Number of lines per set.\n"
                  "-b <num>        Number of block offset bits.\n"
                  "-t <file>     Trace file.\n\n"
                  "Examples:\n"
                  "linux>          %s    -s 4 -E 1 -b 4 -t traces/yi.trace\n"
                  "linux>          %s    -v -s 8 -E 2 -b 4 -t traces/yi.trace\n";
    //"sEbt:hv"：-s、-E、-b、-t 可接收字符，-h 和-v 不接收
    while ((opt = getopt(argc, argv, "sEbt:hv")) != -1) {
        switch (opt) {
            case 'h':
                fprintf(stderr, help, argv[0], argv[0], argv[0]);   //argv[0]是命令行参
数的第一个值，即运行的可执行文件名
                exit(0);
            case 'v':
                v = 1;
                break;
            case 's':
                s = atoi(argv[optind]);     //atoi 字符转十进制
                break;
            case 'E':
                E = atoi(argv[optind]);
                break;
```

```
            case 'b':
                b = atoi(argv[optind]);
                break;
            case 't':
                file_name = argv[optind-1];         //获取文件名
                file = fopen(file_name,"r");    //打开文件
                if(file == NULL){
                    perror(file_name);
                    exit(EXIT_FAILURE);
                }
                break;
            default:
                fprintf(stderr, help, argv[0], argv[0], argv[0]);
                exit(EXIT_FAILURE);
        }
    }
    //设置模拟缓存。初始化。S=2^s 组，即 1<<s。
    int S = 1 << s;
    struct LRU_set cache[S];
    for(int i = 0; i < S; i++){
        cache[i].head = (struct list*)malloc(sizeof(struct list));
        cache[i].head->tag = -1;
        cache[i].E_row = malloc(sizeof(struct row)*E);
        for(int j = 0; j < E; j++){
            cache[i].E_row[j].valid =0 ;
            cache[i].E_row[j].tag = -1;
        }
    }
    char ch[16];                                //刚好可以容纳一行数据的大小
    const char Separator[] = " ,";      //分隔符
    char *token;                                //子串
    while(fgets(ch, 16, file) != NULL){
        token = strtok(ch, Separator);
        char *instruction = token;          //内存访问的类型
        if (*instruction == 'I')              //I 类型不用处理
            continue;
        token = strtok(NULL, Separator);
        char *address_char = token;           //地址的字符表示
        long address = htoi(address_char); //地址的十进制表示
        token = strtok(NULL,Separator);
        int size = atoi(token);
        int set_index = (address >> b) % (1 << s);    //组索引
        int tag = address >> (s + b);                                //标记位
        if(v) printf("%s %s,%d ", instruction, address_char, size);
```

```
int M = *instruction == 'M' ? 2 : 1;                    //'M'  循环两遍,'S''I'一遍
while(M--){
   int flag = 0;
    struct LRU_set *lru_row = &cache[set_index];     //获取该组
    struct list *head = lru_row->head;         //获取该组的双链表头
    for(int i = 0; i < E; i++){                              //遍历一组中的所有行
        struct row *arow = &lru_row->E_row[i]; //获取一行
        if(arow->valid && arow->tag == tag){         //有效且标记相同则命中
                //找到双链表中命中的缓存，准备放到链表头
                while(head->tag != tag)
                    head = head->next;
                //放到链表头，要判断，是不是本身就是头，是头就跳过
                if (head != lru_row->head){
                    head->prior->next = head->next;
                    //判断是不是链表尾，是尾就跳过
                    if(head->next != NULL)
                        head->next->prior = head->prior;
                    head->prior = NULL;
                    head->next = lru_row->head;
                    lru_row->head->prior = head;
                    lru_row->head = head;
                }
                hit_count++;                                    //命中
                if(v) printf("hit ");
                flag = 1;
                break;
        }
    }
    if (flag){
          if (v && !M) printf("\n");
          continue;
    }

    for(int i = 0;i < E;i++){          //无效：未命中，要写入无效的缓存区
        struct row *arow = &lru_row->E_row[i]; //获取一行
        if(!arow->valid){
            //准备放入链表的新结点
            struct list * temp = (struct list*)malloc(sizeof(struct list));
            temp->prior = NULL;
            temp->next = NULL;
            temp->tag = tag;
            //判断是不是链表中的第一个结点，是就跳过
            if(head->tag != -1){
                temp->next = head;
```

```
                head->prior = temp;
            }
            //设置尾头结点
            lru_row->head = temp;
            miss_count++;                                    //未命中
            arow->valid = 1;
            arow->tag = tag;
            if(v) printf("miss ");
            flag = 1;
                break;
        }
    }
    if (flag){
        if (v && !M) printf("\n");
        continue;
    }
    /*
      又没命中，又没有多余的缓存区，则驱逐。驱逐的是双链表的最后一个
       驱逐就是把双链表最后一个移到开头并重新设置 tag
     */
    while(head->next != NULL)
        head = head->next;
    //判断链表中是不是只有一个结点，是就跳过
    if (head != lru_row->head){
        head->prior->next = NULL;
        head->prior = NULL;
        head->next = lru_row->head;
        lru_row->head->prior = head;
        lru_row->head = head;
    }
    //找到在缓存中对应与双链表最后一个的缓存，移除，写入新的缓存。
    struct row *arow;
    for(int j = 0; j < E; j++){
        arow = &cache[set_index].E_row[j];
        if(head->tag == arow->tag)
            break;
    }
    head->tag = tag;
    arow->tag = tag;
    if(v) printf("miss eviction");
    miss_count++;              //未命中
    eviction_count++;       //驱逐
    if (v && !M) printf("\n");                    //不是'M'就要换行
```

```
            }
        }
        //释放分配的空间
        for (int i = 0; i < S; i++){
            free(cache[i].head);
            free(cache[i].E_row);
        }
        printSummary(hit_count, miss_count, eviction_count);
        return 0;
}
```