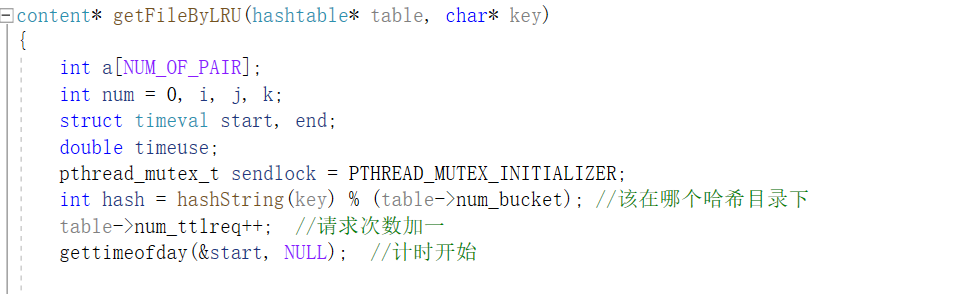
操作系统第四次课设报告

2021.12.20

题目一：（1）LRU算法

此算法思路是最近访问优先，基于哈希结构中将不同的文件存在不同的哈希目录下，之后每次访问到的页面都放在哈希表的第一个，依次往复，若哈希表满了，则将链表尾部的删除，再把最新一次访问的节点头插进来。核心代码如下:

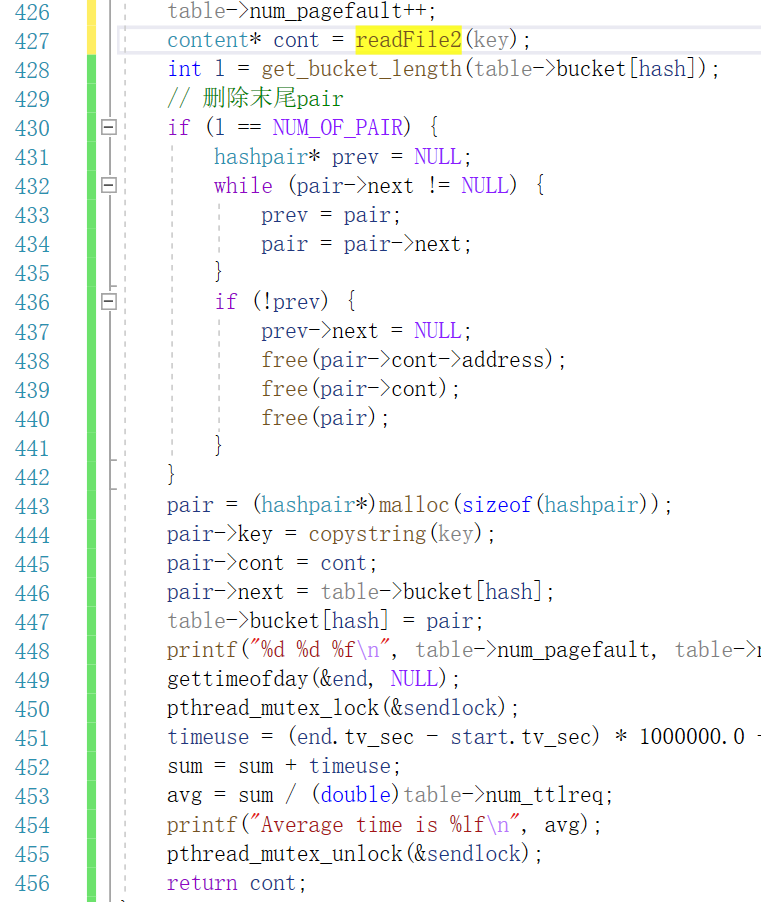


首先找到这个key所在的哈希目录是哪个。



若在哈希链表中找到该文件名，则直接在哈希表缓存中将该文件取出，返回，其中393行的if判断的是不是哈希表的第一个值，若是，则记下时间，返回；若不是i的一个，则用while循环往后找，找到是哪个，计时，返回。

若不在哈希表缓存中，用下面的代码：

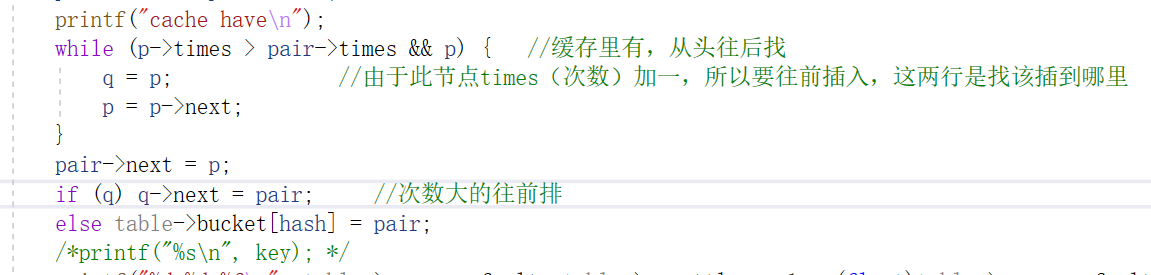


426行是将缺页数加一，427行根据当前key值获取页面内容，通过readfile2函数来获取（所有的函数详见附录源代码）。再之后的428行是获取当前哈希目录下的链表结点个数。若个数达到最多的十个，则删除尾部节点，之后在头部插入当前访问的页面节点。再之后计时结束。

（2）LFU算法

此算法比LRU复杂，要开辟一个特定的空间，用于存放所有访问过的页面，这是个链表，在全局变量上。LFU是根据页面被访问的次数来决定要不要放入缓存的，即页面访问被访问次数多的进入缓存，我采用的尾插法，即哈希表头部的访问次数多，尾部次数少，因此要删除节点的时候，直接删除尾部的就行，把尾部的从缓存里删除，存到开的全局变量链表上（所有访问过的历史记录都在这里，下称为垃圾堆链表），还要看该存到哪里，因为暂存链表也是按照访问次数从多到少排列的。其余的就是链表操作了，插入删除blabla……

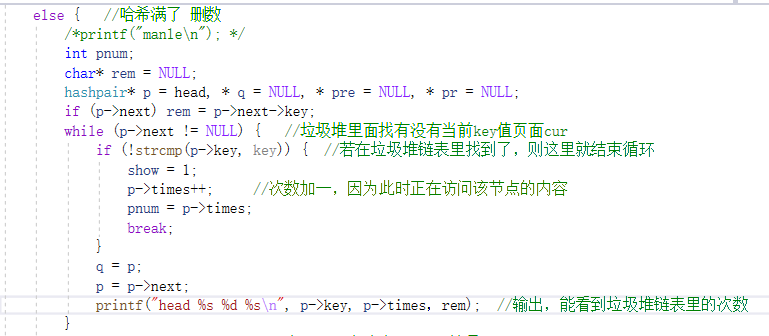
下面是关键代码介绍



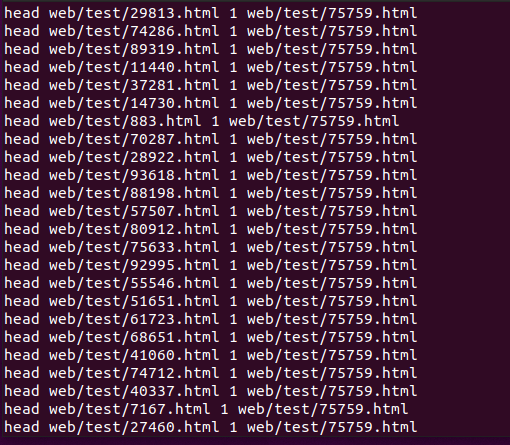
若在缓存中找到，此部分和LRU类似，不过多赘述。唯一区别是上图中的同在缓存中的节点顺序要变化，要按次数（times）降序排列。



上图中的get\_bucket\_length()是统计这个hash值指向的哈希缓存中，节点个数满没满（到十个是满）。若没满，先往缓存里插入，这个hash位置的若没有节点，采用头插法；若有节点，把它插到末尾。这里 要说明一下，我设置的缓存链表是按照times(次数)降序排列的，由于是新进入的节点，times肯定为1，所以直接插到末尾即可。



这是哈希表满了的情况，可能要删除数据，其中删除不删除主要由这个key值的页面被访问次数多少来决定。如果此页面times小于该哈希值对应的哈希表的末尾节点的times，那么意味着它不算最频繁访问的页面，因为连缓存中最低频的次数都达不到，则不把它插入缓存中。但要实现这个功能，首先要在额外队列里找这个key值的页面有没有被访问过，如上图所示方法。



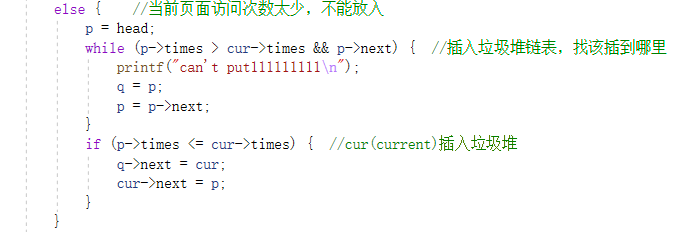
（此图是垃圾堆链表输出的一部分，可以看到每个节点的访问次数全是1,右面那个是rem节点的key，rem是垃圾堆链表的第一个，输出它用于检测垃圾堆链表存储结果是否正确。通过有无头插来看头节点和后面节点的关联来判断。）

若有记录下这个点的访问次数（pnum）,然后show是标志，结束循环，没有，继续在垃圾堆链表里找。

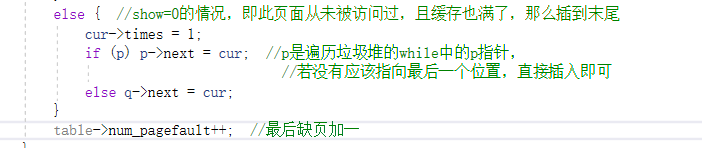
Show=1的执行如下图所示



这是show=1且当前页面访问次数足够多，可以插入缓存的情况，要执行的操作是在哈希表中将访问次数最少的节点拿掉，放入垃圾堆链表里，然后从垃圾堆链表里把该页面插到缓存中该插的地方。



如果当前页面访问次数太少，则不能进入缓存，那么还继续在垃圾堆里，但由于访问次数增加了一次，则要重新寻找插入垃圾堆链表的哪个位置。这里说明，垃圾堆链表之所以也是降序排列，为了方便每次的寻找操作。



这是垃圾堆和缓存中都没有此页面的情况，那么将其插入垃圾堆的最后位置。

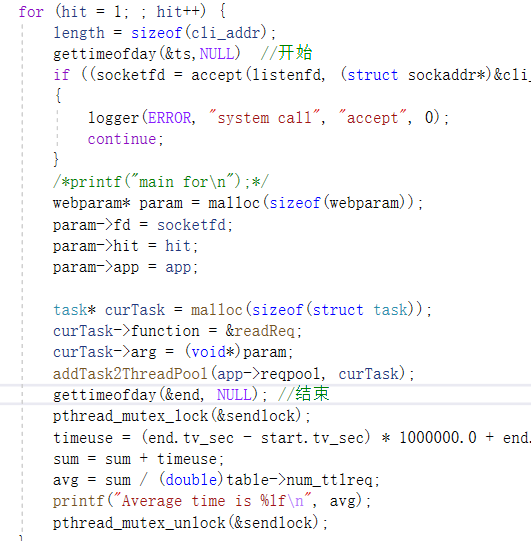
以上是LRU和LFU的实现。

题目二：通过实验来评估各个替换算法的好坏，通过服务器缓存命中率、客户端获得请求内容的平均时间等参数，来说明有无Web文件缓存对Web服务的影响。

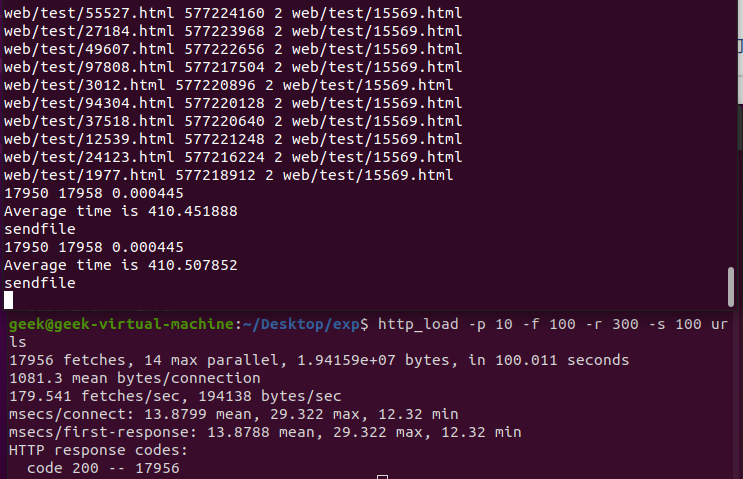
我测试时间测了两个方面，一是总的客户请求的时间，这个开始结束都在main函数里，二是两种缓存调度策略的执行时间，这个在LRU和LFU的算法函数里做测试。



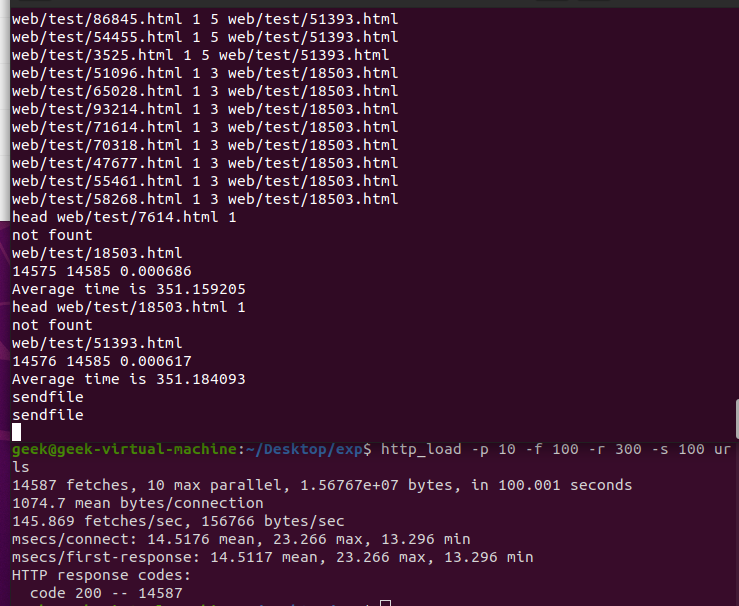
此为LRU的函数内计时，LFU同理。



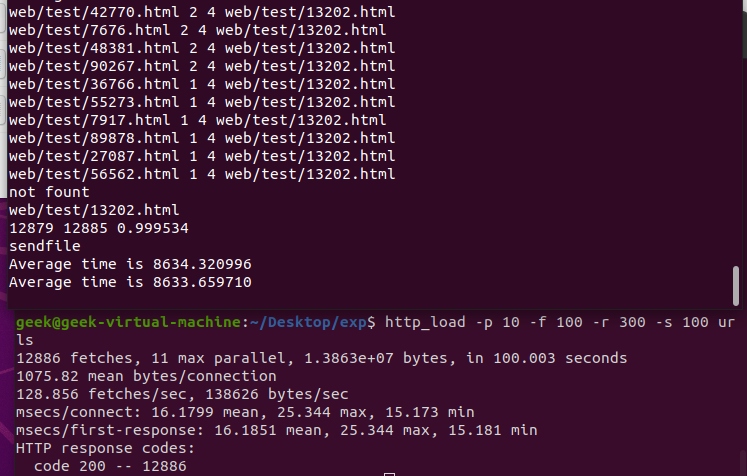
这是测试的总的客户端获得请求的时间，从accept接收到那一刻起，通过各种线程池和消息队列，找到要执行的函数体，经过解析请求，读文件和发送文件三个步骤，最后又回到main里，这是总的客户端获得相应内容的时间。



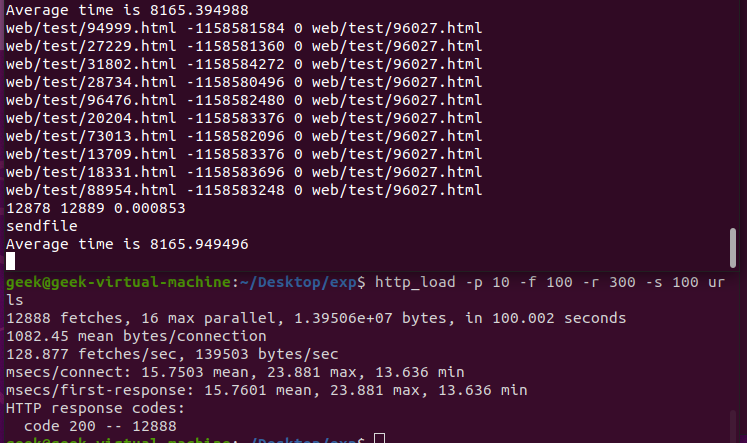
上图是LRU算法的平均执行时间和命中率，17950是缺页次数，17958是总的访问次数，命中率为0.000445。平均执行时间410us.



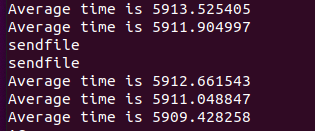
上图是LFU算法的平均执行时间和命中率，14576是缺页次数，14585是总的访问次数，命中率为0.000617。平均执行时间351us.



上图是LFU总的访问平均时间，在main里测试的，8633us左右。



LRU的总访问时间，同理。



没有缓存的情况，平均总运行时间反而更长了。可能因为有缓存但命中率极低，因为十万个页面，而我的缓存数量只有100个，并且缓存没找到的话要进行大量的链表操作，进而影响时间。若把缓存容量开大来提升命中率，可能会有性能提升。

题目三：根据以上实验数据来说明这些替换算法在实验环境中的应用效果，从中找到更为适合此实验环境的替换算法，并说明原因（为什么这个替换算法好？与其它置换算法相比，好在何处？）。

此实验环境更适合LFU算法，因为测试数据有十万个，而缓存大小只有一百个，差距很大，但根据LFU规则，访问次数多的留在缓存中，且还有垃圾堆链表作为备用文件。相比LRU每次都重新插入值，LFU根据访问频率来决定点是否留在缓存，更精确的来调度缓存存储。这一点从题目二的两个算法运行时间也能看出来，LFU比LRU快了60us左右。

附 源代码

#include <sys/prctl.h>

#include <stdbool.h>

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#include <semaphore.h>

#define VERSION 23

#define BUFSIZE 8096

#define ERROR 42

#define LOG 44

#define FORBIDDEN 403

#define NOTFOUND 404

#ifndef SIGCLD

# define SIGCLD SIGCHLD

#endif

#define NUM\_THREADS 40

#define NUM\_REQ 20

#define NUM\_READ 2

#define NUM\_SEND 20

#define NUM\_OF\_PAIR 10 // 桶长度

#define NUM\_OF\_BUCKET 10 // 桶数

struct {

char\* ext;

char\* filetype;

} extensions[] = {

{"gif", "image/gif" },

{"jpg", "image/jpg" },

{"jpeg","image/jpeg"},

{"png", "image/png" },

{"ico", "image/ico" },

{"zip", "image/zip" },

{"gz", "image/gz" },

{"tar", "image/tar" },

{"htm", "text/html" },

{"html","text/html" },

{0,0}

};

/\* 文件内容（长度 + 内容） \*/

typedef struct content

{

int length; // 内容长度

char\* address; // 内容起始地址

} content;

/\* 文件信息对 （文件名 + 文件内容） \*/

typedef struct hashpair {

char\* key; // 文件名

content\* cont; // 内容项

int times;

struct hashpair\* next; // hash桶中指向下一个 hashpair

} hashpair;

/\* hashtable，文件系统载体 \*/

typedef struct hashtable {

hashpair\*\* bucket;

int num\_bucket; // 桶数量

int num\_pagefault; // 缺页次数

int num\_ttlreq; // 所有请求次数

volatile int\* locks; // 对hash桶加锁

volatile int lock; // 对hash table加锁

} hashtable;

/\* 队列状态和条件变量 \*/

typedef struct staconv {

pthread\_mutex\_t mutex; /\* 辅助条件变量 cond \*/

pthread\_cond\_t cond; /\* 等待和激发线程池中的线程 \*/

int status;

}staconv;

/\* 任务 \*/

typedef struct task {

struct task\* next; /\* 指向下一个任务 \*/

void (\*function)(void\* arg); /\* 函数指针 \*/

void\* arg; /\* 函数参数指针 \*/

} task;

/\* 任务队列 \*/

typedef struct taskqueue {

pthread\_mutex\_t mutex; /\* 用于互斥读/写任务队列 \*/

task\* front; /\* 指向队首 \*/

task\* rear; /\* 指向队尾 \*/

staconv\* has\_jobs; /\* 根据状态阻塞线程 \*/

int len; /\* 任务队列中任务个数 \*/

} taskqueue;

/\* 线程 \*/

typedef struct thread {

int type;

int id; /\* 线程 id \*/

pthread\_t pthread; /\* 封装的 POSIX 线程 \*/

struct threadpool\* pool; /\* 与线程池进行绑定 \*/

} thread;

/\* 线程池 \*/

typedef struct threadpool {

thread\*\* threads; /\* 线程指针数组 \*/

volatile int num\_threads; /\* 线程池中线程数量 \*/

volatile int num\_working; /\* 目前正在工作的线程个数 \*/

pthread\_mutex\_t thcount\_lock;/\* 线程池锁，用于修改上面两个变量 \*/

pthread\_cond\_t threads\_all\_idle;/\* 用于销毁线程的条件变量 \*/

taskqueue queue; /\* 任务队列 \*/

volatile bool is\_alive; /\* 线程池是否存活 \*/

}threadpool;

typedef struct application {

threadpool\* reqpool;

threadpool\* readpool;

threadpool\* sendpool;

hashtable\* table; // file cache

} application;

/\* web()参数: 网络请求info \*/

typedef struct webparam {

int hit;

int fd;

application\* app;

} webparam;

typedef struct filename {

int hit;

char\* fn;// filename

char\* fstr;

int fd;// socket key word

application\* app;

} filename;

typedef struct msg {

char\* fc;//文件内容

int fd;

long len;

} msg;

/\* nweb.log \*/

void logger(int type, char\* s1, char\* s2, int socket\_fd);

hashpair\* head;

static inline unsigned long hashString(unsigned char\* str) {

unsigned long hash = 5381;

int c;

while (c = \*str++)

hash = ((hash << 5) + hash) + c; /\* hash \* 33 + c \*/

return hash;

}

// helper for copying string keys and values

static inline char\* copystring(char\* value)

{

char\* copy = (char\*)malloc(strlen(value) + 1);

if (!copy) {

printf("Unable to allocate string value %s\n", value);

abort();

}

strcpy(copy, value);

return copy;

}

// 判断两个 content是否相同，相同返回1，不同返回0

static inline int isEqualContent(content\* cont1, content\* cont2)

{

/\*printf("isequalcontent\n");\*/

if (cont1->length != cont2->length)

return 0;

if (cont1->address != cont2->address)

return 0;

return 1;

}

// Create hash table

hashtable\* createHashTable(int num\_bucket) {

/\*printf("creathashtable\n");\*/

hashtable\* table = (hashtable\*)malloc(sizeof(hashtable));

if (NULL == table) {

return NULL;

}

/\* 创建 hash桶指针 \*/

table->bucket = (hashpair\*\*)malloc(num\_bucket \* sizeof(void\*));

if (NULL == table->bucket) {

free(table);

return NULL;

}

memset(table->bucket, 0, num\_bucket \* sizeof(void\*));

table->num\_bucket = num\_bucket;

table->num\_pagefault = 0;

table->num\_ttlreq = 0;

/\*初始化信号锁 \*/

table->locks = (int\*)malloc(num\_bucket \* sizeof(int));

if (!table->locks) {

free(table);

return NULL;

}

memset((int\*)&table->locks[0], 0, num\_bucket \* sizeof(int));

/\*printf("completehashtable\n");\*/

return table;

}

void freeHashTable(hashtable\* table) //释放哈希表

{

/\*printf("freehashtable\n");\*/

if (!table) return;

hashpair\* next;

int i;

for (i = 0; i < table->num\_bucket; i++) {

// 逐个桶释放

hashpair\* pair = table->bucket[i];

while (pair) {

next = pair->next;

free(pair->key);

free(pair->cont->address);

free(pair->cont);

free(pair);

pair = next;

}

}

free(table->bucket);

/\*#ifdef HASHTHREAD\*/

free(table->locks);

/\*#endif\*/

free(table);

}

/\* 前提：调用者已获得锁 \*/

content\* getContentByKey(hashtable\* table, char\* key)

{

/\*printf("getcontentbykey\n"); \*/

int hash = hashString(key) % (table->num\_bucket);

int num = 0;

hashpair\* pair = table->bucket[hash];

while (pair && num < 10)

{

printf("%s %d %d %s\n", pair->key, pair->times, hash, key);

if (0 == strcmp(pair->key, key))

return pair->cont;

pair = pair->next;

num++;

}

return NULL;

}

int get\_file\_size(char\* key)

{

/\*printf("get file size\n"); \*/

struct stat statbuff;

if (stat(key, &statbuff) < 0) {

return 0;

}

else {

return (int)statbuff.st\_size;

}

}

/\* 读取文件key，包装后返回 \*/

content\* readFile2(char\* key) {

int l = get\_file\_size(key);

/\*printf("readfile2\n");\*/

content\* tmp = malloc(sizeof(content));

if (!tmp)

return NULL;

tmp->length = l + 1;

tmp->address = calloc(l + 1, sizeof(char));

if (!tmp->address) {

free(tmp);

return NULL;

}

int fd = open(key, O\_RDONLY);

if (fd == -1) // 未打开文件

{

free(tmp->address);

free(tmp);

printf("return -1\n");

return NULL;

}

else {

read(fd, tmp->address, l);

close(fd);

/\*printf("completely\n"); \*/

return tmp;

}

}

/\* 前提：调用者已获得锁 \*/

int get\_bucket\_length(hashpair\* pair) {

int l = 0;

while (pair && l < 10) {

l++;

pair = pair->next;

}

return l;

}

double sum = 0, avg;

/\* FIFO算法实现的获取文件函数 \*/

content\* getFileByFIFO(hashtable\* table, char\* key) //key==f->fn

{

int hash = hashString(key) % (table->num\_bucket); //找到哈希表头节点

table->num\_ttlreq++; //请求次数加一

content\* tmp = getContentByKey(table, key); //找到和key一致的哈希节点

// 缓存中存在文件

if (tmp)

return tmp;

// 缺页

hashpair\* pair = table->bucket[hash];

content\* cont = readFile2(key); //读取key文件的内容

// 统计

int l = get\_bucket\_length(table->bucket[hash]); //哈希表长

printf("daozhelile\n");

// 删除末尾pair

if (l == NUM\_OF\_PAIR) {

hashpair\* prev = NULL;

while (pair->next != NULL) {

prev = pair;

pair = pair->next;

}

if (!prev) {

prev->next = NULL;

free(pair->cont->address);

free(pair->cont);

free(pair);

}

// 记录缺页置换次数

table->num\_pagefault++;

}

// 头插

pair = (hashpair\*)malloc(sizeof(hashpair));

pair->key = copystring(key);

pair->cont = cont;

pair->next = table->bucket[hash]; // 头插法

table->bucket[hash] = pair;

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, (float)table->num\_pagefault / (float)table->num\_ttlreq);

return cont;

}

/\* 使用前提：调用者已加锁 \*/

void movePairToHead(hashtable\* table, char\* key)

{

int hash = hashString(key) % (table->num\_bucket);

hashpair\* pair = table->bucket[hash];

// 不用移动

if (!strcmp(key, pair->key)) {

return;

}

// 找到位置、连接断裂处、头插

hashpair\* prev = NULL;

while (pair->next != NULL)

{

if (!strcmp(pair->key, key)) {

break;

}

prev = pair;

pair = pair->next;

}

prev->next = pair->next;

pair->next = table->bucket[hash];

table->bucket[hash] = pair;

}

content\* getFileByLRU(hashtable\* table, char\* key)

{

int a[NUM\_OF\_PAIR];

int num = 0, i, j, k;

struct timeval start, end;

double timeuse;

pthread\_mutex\_t sendlock = PTHREAD\_MUTEX\_INITIALIZER;

int hash = hashString(key) % (table->num\_bucket); //该在哪个哈希目录下

table->num\_ttlreq++; //请求次数加一

gettimeofday(&start, NULL); //计时开始

content\* tmp = getContentByKey(table, key); //找缓存中有没有一样的文件

// 缓存中存在文件

hashpair\* pair = table->bucket[hash];

if (tmp) { //有

printf("cache has\n");

if (!strcmp(key, pair->key)) { //第一个就是

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, 1 - (float)table->num\_pagefault / (float)table->num\_ttlreq);

gettimeofday(&end, NULL); //要退出，计时结束

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg);//存取平均时间

pthread\_mutex\_unlock(&sendlock);

return tmp;

}

hashpair\* prev = NULL;

while (pair->next != NULL) //不是第一个，往后找

{

if (!strcmp(pair->key, key)) {

break;

}

prev = pair;

pair = pair->next;

}

prev->next = pair->next;

pair->next = table->bucket[hash];

table->bucket[hash] = pair;

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, 1 - (float)table->num\_pagefault / (float)table->num\_ttlreq);

gettimeofday(&end, NULL); //计时结束

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg); //存取平均时间

pthread\_mutex\_unlock(&sendlock);

return tmp;

}

table->num\_pagefault++;

content\* cont = readFile2(key);

int l = get\_bucket\_length(table->bucket[hash]);

// 删除末尾pair

if (l == NUM\_OF\_PAIR) {

hashpair\* prev = NULL;

while (pair->next != NULL) {

prev = pair;

pair = pair->next;

}

if (!prev) {

prev->next = NULL;

free(pair->cont->address);

free(pair->cont);

free(pair);

}

}

pair = (hashpair\*)malloc(sizeof(hashpair));

pair->key = copystring(key);

pair->cont = cont;

pair->next = table->bucket[hash];

table->bucket[hash] = pair;

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, 1 - (float)table->num\_pagefault / (float)table->num\_ttlreq);

gettimeofday(&end, NULL); //计时结束

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg);

pthread\_mutex\_unlock(&sendlock);

return cont;

}

content\* getFileByLFU(hashtable\* table, char\* key)

{

int a[NUM\_OF\_PAIR];

int num = 0, i, j, k, show = 0;

struct timeval start, end;

double timeuse;

pthread\_mutex\_t sendlock = PTHREAD\_MUTEX\_INITIALIZER;

int hash = hashString(key) % (table->num\_bucket); //该在哪

/\*while (\_\_sync\_lock\_test\_and\_set(&table->locks[hash], 1)); \*/

table->num\_ttlreq++; //请求次数加一

gettimeofday(&start, NULL);

content\* tmp = getContentByKey(table, key);

hashpair\* pair = table->bucket[hash];

hashpair\* cur = NULL;

if (tmp) { //找到了

if (!strcmp(key, pair->key)) {

pair->times++;

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, 1 - (float)table->num\_pagefault / (float)table->num\_ttlreq);

printf("%s\n", key);

/\*\_\_sync\_synchronize();

table->locks[hash] = 0;\*/

gettimeofday(&end, NULL);

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg);

pthread\_mutex\_unlock(&sendlock);

return tmp;

}

hashpair\* prev = NULL, \* p = table->bucket[hash], \* q = NULL;

while (pair->next != NULL)

{

if (!strcmp(pair->key, key)) {

pair->times++;

break;

}

prev = pair;

pair = pair->next;

}

prev->next = pair->next;

printf("cache have\n");

while (p->times > pair->times && p) { //缓存里有，从头往后找

q = p; //由于此节点times（次数）加一，所以要往前插入，这两行是找该插到哪里

p = p->next;

}

pair->next = p;

if (q) q->next = pair; //次数大的往前排

else table->bucket[hash] = pair;

/\*printf("%s\n", key); \*/

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, 1 - (float)table->num\_pagefault / (float)table->num\_ttlreq);

/\*\_\_sync\_synchronize();

table->locks[hash] = 0;\*/

gettimeofday(&end, NULL);

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg);

pthread\_mutex\_unlock(&sendlock);

return tmp;

}

// 缺页

content\* cont = readFile2(key);

cur = (hashpair\*)malloc(sizeof(hashpair));

cur->key = copystring(key);

cur->cont = cont;

cur->next = NULL;

// 统计

int l = get\_bucket\_length(table->bucket[hash]);

if (l < NUM\_OF\_PAIR) { //哈希没满

/\*printf("meiman\n"); \*/

cur->times = 1;

if (!pair) { //该hash位置没有节点时，头插

pair = (hashpair\*)malloc(sizeof(hashpair));

pair = table->bucket[hash];

pair = cur;

pair->next = table->bucket[hash];

table->bucket[hash] = pair;

}

else { //有节点了，尾插

while (pair->next != NULL && num <= 10) {

printf("meimanelse %s %d %s\n", pair->key, pair->times, key);

pair = pair->next;

num++;

}

pair->next = cur;

}

table->num\_pagefault++; //缺页加一

/\*printf("insert directly completed\n"); \*/

}

else { //哈希满了 删数

/\*printf("manle\n"); \*/

int pnum;

char\* rem = NULL;

hashpair\* p = head, \* q = NULL, \* pre = NULL, \* pr = NULL;

if (p->next) rem = p->next->key;

while (p->next != NULL) { //垃圾堆里面找有没有当前key值页面cur

if (!strcmp(p->key, key)) { //若在垃圾堆链表里找到了，则这里就结束循环

show = 1;

p->times++; //次数加一，因为此时正在访问该节点的内容

pnum = p->times;

break;

}

q = p;

p = p->next;

printf("head %s %d %s\n", p->key, p->times，rem); //输出，能看到垃圾堆链表里的次数

}

if (show) { //历史记录里有这个页面，p就是cur

printf("youyemian\n");

while (pair->next != NULL && num < 10) { //在哈希链表里找到最后一个点

pr = pair;

pair = pair->next;

num++;

}

//pr->next = pair->next;

//q->next = p->next;

if (pair->times <= pnum) {//哈希表最少的访问页面小于当前访问页面，可以放入,删除pair

q = head->next;

while (q->times > pair->times && q->next != NULL) { //找到原哈希表中的值插入垃圾堆链表该插到哪里，

//按次数降序排列

pre = q;

q = q->next;

}

//将pair(原哈希表中的）插入垃圾堆列表

/\*pr->next = NULL; \*/

pair->next = q;

if (pre) pre->next = pair;

else if (!pre) head->next = pair;

pair = table->bucket[hash]->next;

num = 0;

while (pair->times > p->times && pair->next && num < 10) { //找到垃圾堆链表中的点（当前访问页）

//该插到哈希表中的哪里，按次数降序排解

pr = pair;

pair = pair->next;

num++;

/\*printf("manleshow %s %d %s\n", pair->key, pair->times, key);\*/

}

/\*printf("found2222222222\n"); \*/

pr->next = p; //将当前页面放入哈希表，当作缓存里的元素

p->next = pair;

}

else { //当前页面访问次数太少，不能放入

p = head;

while (p->times > cur->times && p->next) { //插入垃圾堆链表，找该插到哪里

printf("can't put111111111\n");

q = p;

p = p->next;

}

if (p->times <= cur->times) { //cur(current)插入垃圾堆

q->next = cur;

cur->next = p;

}

}

}

else { //show=0的情况，即此页面从未被访问过，且缓存也满了，那么插到末尾

cur->times = 1;

if (p) p->next = cur; //p是遍历垃圾堆的while中的p指针，

//若没有应该指向最后一个位置，直接插入即可

else q->next = cur;

}

table->num\_pagefault++; //最后缺页加一

}

printf("%s\n", key);

printf("%d %d %f\n", table->num\_pagefault, table->num\_ttlreq, 1 - (float)table->num\_pagefault / (float)table->num\_ttlreq);

gettimeofday(&end, NULL);

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg);

pthread\_mutex\_unlock(&sendlock);

/\*\_\_sync\_synchronize();

table->locks[hash] = 0;\*/

return cont;

}

void push\_taskqueue(taskqueue\* queue, task\* curTask) {

if (NULL == curTask) return;

pthread\_mutex\_lock(&queue->mutex);//写taskqueue

if (queue->front == NULL && queue->rear == NULL) {

curTask->next = NULL;

queue->front = queue->rear = curTask;

}

else {

curTask->next = NULL;

queue->rear->next = curTask;

queue->rear = curTask;

}

queue->len++;

pthread\_mutex\_lock(&queue->has\_jobs->mutex);//条件变量加锁

queue->has\_jobs->status = 1;

pthread\_cond\_broadcast(&queue->has\_jobs->cond); //激发阻塞的线程等待条件

pthread\_mutex\_unlock(&queue->has\_jobs->mutex);

pthread\_mutex\_unlock(&queue->mutex);

}

void sendFile(void\* msg2) {

printf("sendfile\n");

msg\* m = (msg\*)msg2;

write(m->fd, m->fc, m->len);

usleep(10000);

close(m->fd);

free(m);

}

void readFile(void\* file)

{

/\*printf("readfile\n"); \*/

filename\* f = (filename\*)file;

// 发送信息写入log

logger(LOG, "SEND", f->fn, f->hit);

// 获取文件长度

int len = get\_file\_size(f->fn);

// 发送响应消息

char buffer[BUFSIZE + 1];

(void)sprintf(buffer, "HTTP/1.1 200 OK\nServer: nweb/%d.0\nContent-Length: %d\nConnection: close\nContent-Type: %s\n\n", VERSION, len, f->fstr);

// 写入log

logger(LOG, "Header", buffer, f->hit);

(void)write(f->fd, buffer, strlen(buffer));

msg\* m = malloc(sizeof(msg));

m->fd = f->fd;

/\*printf("before LFU\n"); \*/

m->fc = getFileByLFU(f->app->table, f->fn)->address;

m->len = len;

free(f->fn);

free(f->fstr);

free(f);

task\* t = malloc(sizeof(task));

t->function = &sendFile;

t->arg = (void\*)m;

/\*printf("qwq\n");\*/

push\_taskqueue(&f->app->sendpool->queue, t);

}

void readReq(void\* data)

{

//1.读取文件信息

//2.将 filename 和 fd 打包成filename 添加到 filename-queue

int j, file\_fd, buflen;

long i, ret, len;

char\* fstr;

char buffer[BUFSIZE + 1]; /\* !!!!!!!! 不能是静态缓冲区 \*/

webparam\* param = (webparam\*)data;

int fd = param->fd;

int hit = param->hit;

ret = read(fd, buffer, BUFSIZE); /\* 从客户端读取请求消息 \*/

if (ret == 0 || ret == -1) { /\* 读消息失败 \*/

logger(FORBIDDEN, "failed to read browser request", "", fd);

return;

}

else {

if (ret > 0 && ret < BUFSIZE) /\* 设置有效字符串 \*/

buffer[ret] = 0;

else buffer[0] = 0;

for (i = 0; i < ret; i++) /\* remove CF and LF characters \*/

if (buffer[i] == '\r' || buffer[i] == '\n')

buffer[i] = '\*';

// 请求信息l写入log

logger(LOG, "request", buffer, hit);

// method

if (strncmp(buffer, "GET ", 4) && strncmp(buffer, "get ", 4)) {

logger(FORBIDDEN, "Only simple GET operation supported", buffer, fd);

close(fd);

return;

}

for (i = 4; i < BUFSIZE; i++) { /\* null terminate after the second space to ignore extra stuff \*/

if (buffer[i] == ' ') { /\* string is "GET URL " +lots of other stuff \*/

buffer[i] = 0;

break;

}

}

// 不能使用 ..

for (j = 0; j < i - 1; j++) /\* check for illegal parent directory use .. \*/

if (buffer[j] == '.' && buffer[j + 1] == '.') {

logger(FORBIDDEN, "Parent directory (..) path names not supported", buffer, fd);

close(fd);

return;

}

// 不包含有效文件名有效文件名，使用默认 /index.html

if (!strncmp(&buffer[0], "GET /\0", 6) || !strncmp(&buffer[0], "get /\0", 6))

(void)strcpy(buffer, "GET /index.html");

// 根据预定义在extensionsh中的文件类型，检查请求的文件类型是否本服务器支持

buflen = strlen(buffer);

fstr = (char\*)0;

for (i = 0; extensions[i].ext != 0; i++) {

len = strlen(extensions[i].ext);

if (!strncmp(&buffer[buflen - len], extensions[i].ext, len)) {

fstr = extensions[i].filetype;

break;

}

}

if (fstr == 0) {

logger(FORBIDDEN, "file extension type not supported", buffer, fd);

close(fd);

return;

}

filename\* f = (filename\*)malloc(sizeof(filename));

f->hit = hit;

f->fd = fd;

int l = strlen(&buffer[5]);

f->fn = (char\*)malloc(sizeof(char) \* (l + 1));

strncpy(f->fn, &buffer[5], l);

f->fn[l] = '\0';

f->app = param->app;

int l2 = strlen(fstr);

f->fstr = (char\*)malloc(sizeof(char) \* (l2 + 1)); //扩展名

strncpy(f->fstr, fstr, l2);

f->fstr[l2] = '\0';

task\* t = (task\*)malloc(sizeof(task));

t->function = &readFile;

t->arg = (void\*)f;

push\_taskqueue(&param->app->readpool->queue, t);

free(param);

}

}

/\* 初始化任务队列 \*/

void init\_taskqueue(taskqueue\* queue) {

/\*printf("init taskqueue\n");\*/

queue->front = NULL;

queue->rear = NULL;

queue->len = 0;

pthread\_mutex\_init(&queue->mutex, NULL);//互斥读写任务队列

/\* staconv \* has\_jobs; \*/

queue->has\_jobs = (staconv\*)malloc(sizeof(struct staconv));

pthread\_mutex\_init(&queue->has\_jobs->mutex, NULL);

pthread\_cond\_init(&queue->has\_jobs->cond, NULL);//阻塞和唤醒线程池中线程

queue->has\_jobs->status = 0;//任务队列任务状态，0表示无任务

}

// 从哪个队列取任务

task\* take\_taskqueue(taskqueue\* queue) {

//从头部提取任务

pthread\_mutex\_lock(&queue->mutex);

task\* t = NULL;

if (queue->front == NULL && queue->rear == NULL) {

pthread\_mutex\_unlock(&queue->mutex);

return t;

}

else if (queue->front == queue->rear) {

t = queue->front;

queue->front = queue->rear = NULL;

}

else {

t = queue->front;

queue->front = queue->front->next;

}

queue->len--;

if (queue->len == 0) {

pthread\_mutex\_lock(&queue->has\_jobs->mutex);//条件变量加锁

queue->has\_jobs->status = 0;

pthread\_mutex\_unlock(&queue->has\_jobs->mutex);

}

pthread\_mutex\_unlock(&queue->mutex);

return t;

}

//销毁任务队列

void destory\_taskqueue(taskqueue\* queue) {

/\*printf("destroy taskqueue\n");\*/

pthread\_mutex\_lock(&queue->mutex);

queue->len = 0;

queue->has\_jobs->status = 0;

while (queue->front != queue->rear) {

task\* t = queue->front;

queue->front = queue->front->next;

free(t);

}

if (queue->front != NULL) free(queue->front);

queue->front = queue->rear = NULL;

free(queue->has\_jobs);

queue->has\_jobs = NULL;

pthread\_mutex\_unlock(&queue->mutex);

free(queue);

}

void\* thread\_do(void\* pth)

{

/\*printf("thread do\n");\*/

struct thread\* pthread = (struct thread\*)pth;

char thread\_name[128] = { 0 };

sprintf(thread\_name, "%d-thread-%d", pthread->type, pthread->id);

prctl(PR\_SET\_NAME, thread\_name);// 设置进程名

/\* 获得线程池 \*/

threadpool\* pool = pthread->pool;

pthread\_mutex\_lock(&pool->thcount\_lock);

/\* 在线程池初始化时，用于已经创建线程的计数 \*/

pool->num\_threads++;

pthread\_mutex\_unlock(&pool->thcount\_lock);

/\* 线程一直往返运行，直到 pool->is\_alive=false \*/

while (pool->is\_alive) {

/\* 任务队列没有任务则阻塞 \*/

// 使用 staconv \* has\_jobs 等待和激发

pthread\_mutex\_lock(&pool->queue.has\_jobs->mutex);

while (pool->queue.has\_jobs->status == 0) {

pthread\_cond\_wait(&pool->queue.has\_jobs->cond, &pool->queue.has\_jobs->mutex); // 队列无任务，等待被激发

}

pthread\_mutex\_unlock(&pool->queue.has\_jobs->mutex);

if (pool->is\_alive) {

/\* 对工作线程数量进行计数 \*/

pthread\_mutex\_lock(&pool->thcount\_lock);

pool->num\_working++;

pthread\_mutex\_unlock(&pool->thcount\_lock);

void (\*func)(void\*);

void\* arg;

task\* curtask = take\_taskqueue(&pool->queue); /\* take task from taskqueue \*/

if (NULL != curtask) {

// logger(LOG,"task\_taskqueue\_successful",thread\_name,0);

func = curtask->function;

arg = curtask->arg;

func(arg);//执行任务 web(void\*)

free(curtask);//释放任务

}

/\* 线程已经将任务执行完成，需要更改工作线程数量 \*/

pthread\_mutex\_lock(&pool->thcount\_lock);

pool->num\_working--;

pthread\_mutex\_lock(&pool->queue.mutex);

if (0 == pool->num\_working && pool->queue.len == 0) { /\* 任务全部完成，让阻塞在waitThreadPool上的线程继续运行 \*/

pthread\_cond\_broadcast(&pool->threads\_all\_idle); //条件激发，此时工作线程数=0，任务数=0 ==> 等待所有线程停止，进行线程池资源释放

}

pthread\_mutex\_unlock(&pool->queue.mutex);

pthread\_mutex\_unlock(&pool->thcount\_lock);

}

}

pthread\_mutex\_lock(&pool->thcount\_lock);

/\* 线程退出，要修改线程池中的数量 \*/

pool->num\_threads--;

pthread\_mutex\_unlock(&pool->thcount\_lock);

return NULL;

}

/\* 向线程池加入任务 \*/

void addTask2ThreadPool(threadpool\* pool, task\* curtask)

{

//将任务加入队列

//logger(LOG,"addTask2ThreadPool","",0);

push\_taskqueue(&pool->queue, curtask);

}

/\* 等待当前任务全部运行完 \*/

void waitThreadPool(threadpool\* pool)

{

/\*printf("wait thread pool\n");\*/

pthread\_mutex\_lock(&pool->thcount\_lock);

while (pool->queue.len || pool->num\_working) {

pthread\_cond\_wait(&pool->threads\_all\_idle, &pool->thcount\_lock);

}

pthread\_mutex\_unlock(&pool->thcount\_lock);

}

/\* 销毁线程池 \*/

void destoryThreadPool(threadpool\* pool, int n)

{

/\*printf("destroythreadpool\n");\*/

//1.如果当前任务队列中有任务，需等待任务队列为空，并且运行线程执行任务后

waitThreadPool(pool); //len=0 && num\_working=0

//2.销毁任务队列

pthread\_mutex\_lock(&pool->thcount\_lock);

pool->is\_alive = false;

pthread\_mutex\_unlock(&pool->thcount\_lock);

destory\_taskqueue(&pool->queue);

//3.销毁线程指针数组，并释放所有为线程池分配的内存

int i = 0;

for (; i < n; i++) free(pool->threads[i]);//free every thread

free(pool->threads);//free array

free(pool);//free pool

}

void destoryApp(application\* app)

{

/\*printf("destroyapp\n"); \*/

destoryThreadPool(app->reqpool, NUM\_REQ);

destoryThreadPool(app->readpool, NUM\_READ);

destoryThreadPool(app->sendpool, NUM\_SEND);

freeHashTable(app->table);

}

/\* 获得当前线程池中正在运行线程的数量 \*/

int getNumofThreadWorking(threadpool\* pool) {

return pool->num\_working;

}

/\* 创建线程 \*/

int create\_thread(struct threadpool\* pool, struct thread\* pthread, int id, int type)

{

/\*printf("create thread\n");\*/

pthread->pool = pool;

pthread->id = id;

pthread->type = type;

pthread\_create(&pthread->pthread, NULL, (void\*)thread\_do, (void\*)pthread);//例程 thread\_do, 参数 thread\*

pthread\_detach(pthread->pthread);

return 0;

}

/\* 线程池初始化函数 \*/

struct threadpool\* initThreadPool(int threads\_num, int type) {

/\*printf("initthreadpool\n");\*/

// 创建线程池空间

threadpool\* pool = (threadpool\*)malloc(sizeof(struct threadpool));

pool->num\_threads = 0;

pool->num\_working = 0;

pthread\_mutex\_init(&(pool->thcount\_lock), NULL);//互斥量,用来修改 num\_threads和num\_working

pthread\_cond\_init(&(pool->threads\_all\_idle), NULL);//条件变量,用来销毁线程

init\_taskqueue(&(pool->queue));

pool->is\_alive = true;

/\* 存放`thread\*`的数组 \*/

pool->threads = (struct thread\*\*)malloc(threads\_num \* sizeof(struct thread));//总共创建NUM\_THREADS个线程

for (int i = 0; i < threads\_num; i++) {

pool->threads[i] = (struct thread\*)malloc(sizeof(struct thread));

create\_thread(pool, pool->threads[i], i, type);// i为线程id，循序执行，不必上锁

}

// 忙等待，直到所有线程创建完毕

while (pool->num\_threads != threads\_num) {}

//logger(LOG,"initThreadPool","",0);

return pool;

}

struct application\* initApp() {

struct application\* app = (struct application\*)malloc(sizeof(struct application));

app->reqpool = initThreadPool(NUM\_REQ, 1);

app->readpool = initThreadPool(NUM\_READ, 2);

app->sendpool = initThreadPool(NUM\_SEND, 3);

app->table = createHashTable(NUM\_OF\_BUCKET);

return app;

}

void inithashpair()

{

head = (struct hashpair\*)malloc(sizeof(struct hashpair));

if (!head) {

printf("error head\n");

exit(1);

}

head->cont = "aaaaa";

head->key = "aaaaaaa";

head->times = 0;

head->next = NULL;

}

int main(int argc, char\*\* argv)

{

int i, port, pid, listenfd, socketfd, hit;

socklen\_t length;

static struct sockaddr\_in cli\_addr; /\* static = initialised to zeros \*/

static struct sockaddr\_in serv\_addr; /\* static = initialised to zeros \*/

// 解析b命令行参数

if (argc < 3 || argc > 3 || !strcmp(argv[1], "-?")) {

(void)printf("hint: nweb Port-Number Top-Directory\t\tversion %d\n\n"

"\tnweb is a small and very safe mini web server\n"

"\tnweb only servers out file/web pages with extensions named below\n"

"\t and only from the named directory or its sub-directories.\n"

"\tThere is no fancy features = safe and secure.\n\n"

"\tExample: nweb 8181 /home/nwebdir &\n\n"

"\tOnly Supports:", VERSION);

for (i = 0; extensions[i].ext != 0; i++)

(void)printf(" %s", extensions[i].ext);

(void)printf("\n\tNot Supported: URLs including \"..\", Java, Javascript, CGI\n"

"\tNot Supported: directories / /etc /bin /lib /tmp /usr /dev /sbin \n"

"\tNo warranty given or implied\n\tNigel Griffiths nag@uk.ibm.com\n");

exit(0);

}

if (!strncmp(argv[2], "/", 2) || !strncmp(argv[2], "/etc", 5) ||

!strncmp(argv[2], "/bin", 5) || !strncmp(argv[2], "/lib", 5) ||

!strncmp(argv[2], "/tmp", 5) || !strncmp(argv[2], "/usr", 5) ||

!strncmp(argv[2], "/dev", 5) || !strncmp(argv[2], "/sbin", 6))

{

(void)printf("ERROR: Bad top directory %s, see nweb -?\n", argv[2]);

exit(3);

}

if (chdir(argv[2]) == -1) {

(void)printf("ERROR: Can't Change to directory %s\n", argv[2]);

exit(4);

}

(void)signal(SIGCLD, SIG\_IGN); // ignore child death

(void)signal(SIGHUP, SIG\_IGN); // ignore terminal hangups

logger(LOG, "nweb starting", argv[1], getpid());

// 建立服务端侦听socket

if ((listenfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

logger(ERROR, "system call", "socket", 0);

port = atoi(argv[1]);

if (port < 0 || port >60000)

logger(ERROR, "Invalid port number (try 1->60000)", argv[1], 0);

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

serv\_addr.sin\_port = htons(port);

if (bind(listenfd, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0)

logger(ERROR, "system call", "bind", 0);

if (listen(listenfd, 64) < 0)

logger(ERROR, "system call", "listen", 0);

//struct threadpool \* pool = initThreadPool();//不传递参数，用宏常量

struct application\* app = initApp();

inithashpair();

for (hit = 1; ; hit++) {

length = sizeof(cli\_addr);

gettimeofday(&ts,NULL) //开始

if ((socketfd = accept(listenfd, (struct sockaddr\*)&cli\_addr, &length)) < 0)

{

logger(ERROR, "system call", "accept", 0);

continue;

}

/\*printf("main for\n");\*/

webparam\* param = malloc(sizeof(webparam));

param->fd = socketfd;

param->hit = hit;

param->app = app;

task\* curTask = malloc(sizeof(struct task));

curTask->function = &readReq;

curTask->arg = (void\*)param;

addTask2ThreadPool(app->reqpool, curTask);

gettimeofday(&end, NULL); //结束

pthread\_mutex\_lock(&sendlock);

timeuse = (end.tv\_sec - start.tv\_sec) \* 1000000.0 + end.tv\_usec - start.tv\_usec;

sum = sum + timeuse;

avg = sum / (double)table->num\_ttlreq;

printf("Average time is %lf\n", avg);

pthread\_mutex\_unlock(&sendlock);

}

destoryApp(app);

}

void logger(int type, char\* s1, char\* s2, int socket\_fd)

{

int fd;

char logbuffer[BUFSIZE \* 2];

// 获取时间信息

struct tm\* ptr;

time\_t lt;

lt = time(NULL);

ptr = localtime(&lt);

char timebuf[48] = { 0 };

strftime(timebuf, sizeof(timebuf), "%H:%M:%S %a %d/%m/%Y", ptr);

/\*将消息写入logger，或直接将消息通过socket通道返回给客户端\*/

switch (type) {

case ERROR: (void)sprintf(logbuffer, "[%s] ERROR: %s:%s Errno=%d exiting pid=%d\n", timebuf, s1, s2, errno, getpid());

break;

case FORBIDDEN:

(void)write(socket\_fd, "HTTP/1.1 403 Forbidden\nContent-Length: 185\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>403 Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\nThe requested URL, file type or operation is not allowed on this simple static file webserver.\n</body></html>\n", 271);

(void)sprintf(logbuffer, "[%s] FORBIDDEN: %s:%s:%d\n", timebuf, s1, s2, socket\_fd);

break;

case NOTFOUND:

(void)write(socket\_fd, "HTTP/1.1 404 Not Found\nContent-Length: 136\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>404 Not Found</title>\n</head><body>\n<h1>Not Found</h1>\nThe requested URL was not found on this server.\n</body></html>\n", 224);

(void)sprintf(logbuffer, "[%s] NOT FOUND: %s:%s:%d\n", timebuf, s1, s2, socket\_fd);

break;

case LOG: (void)sprintf(logbuffer, "[%s] INFO: %s:%s:%d", timebuf, s1, s2, socket\_fd); break;

}

/\* No checks here, nothing can be done with a failure anyway \*/

if ((fd = open("nweb.log", O\_CREAT | O\_WRONLY | O\_APPEND, 0644)) >= 0) {

(void)write(fd, logbuffer, strlen(logbuffer));

(void)write(fd, "\n", 1);

(void)close(fd);

}

//if(type == NOTFOUND || type == FORBIDDEN) pthread\_exit((void\*)3);

//if(type == ERROR) exit(3);

}