

操作系统实验报告



**实验名称： 6.web服务器页面缓存及其替换评估方法**

**班级： 计算机18-5班**

**姓名： 冷亚静 学号： 2018010065**

**中国石油大学（北京）信息科学与工程学院**

**计算机科学与技术**

**题目一：**

1. **实验题目：**

根据在本节介绍的基于hash缓存结构和各种缓存替换算法，设计不同的缓存管理辅助结构（例如，队列，堆）等来实现LRU、LFU、ARC、MQ、GD和GDSF替换算法。

1. **实验设计：**
   1. **LRU算法：**

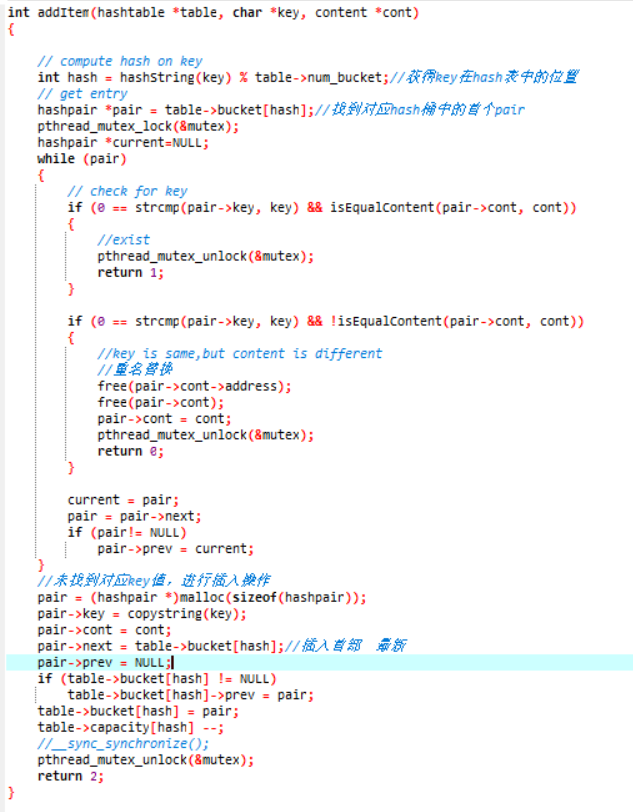
设计使用链表结构。

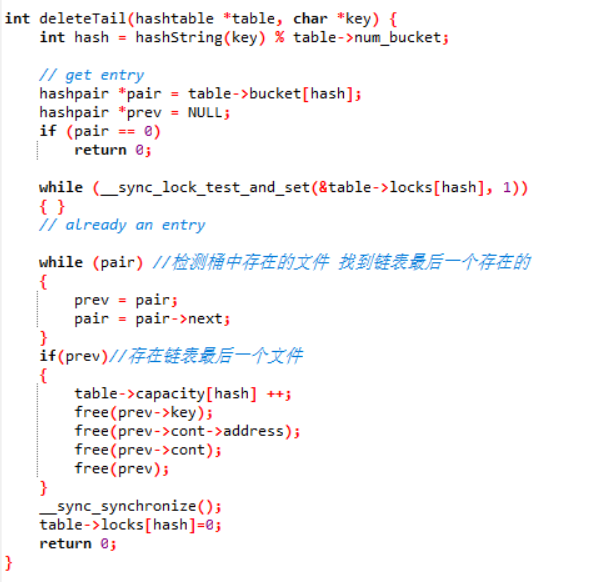
初始设置哈希表的长度，并确定每个哈希桶的大小。根据hash值来计算key（页面名称）在hash表中的位置。

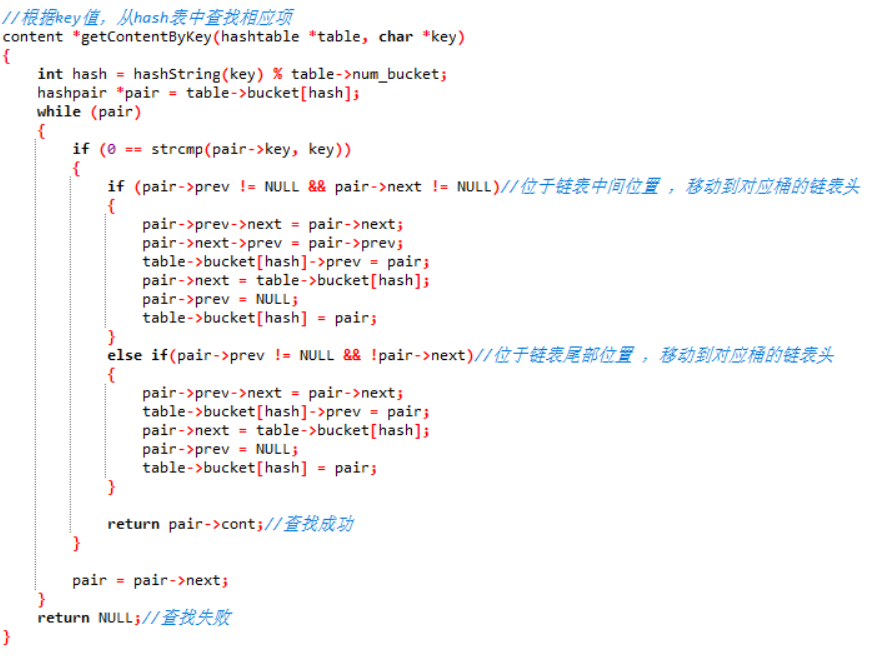


向hash桶内添加页面内容：返回1，表示要添加项已经在hash表中存在。返回0，如果仅是key相同，而content不同。返回2，如果不存在key，并且正常加入到hash table中。

LRU为最近最久未使用置换，设计思路，利用链表结构。每一次访问页面成功（getContentByKey函数）或向hash桶中添加页面（addItem函数）是将该页面放入对应hash桶链表的表头。这样，每次hash桶的表尾即为最近最久未使用的页面，发生缺页替换时，即将表尾的页面进行删除替换（deleteTail函数）。

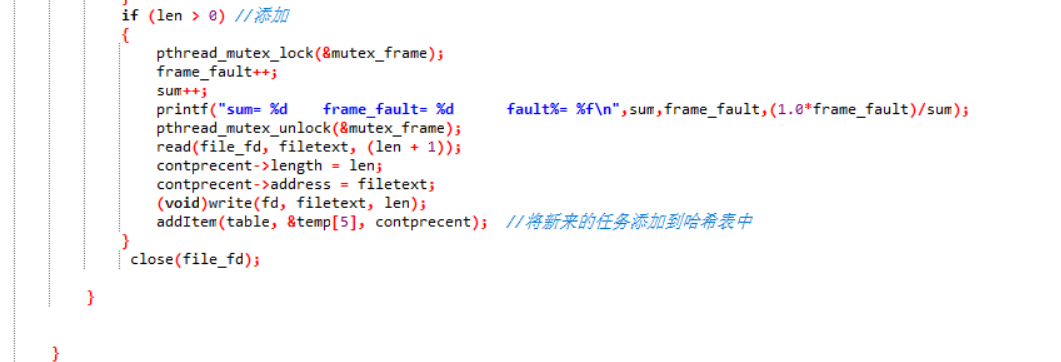






修改web函数：每次进行页面访问时，先判断对应的hash桶内是否存在请求的页面，若不存在，则判断hash桶是否已满：若已满，则按lru算法进行最近最久未访问页面的删除操作，并将请求访问页面加入到对应hash桶内。若不满，则直接将请求访问页面加入到对应hash桶内。若请求页面已经缓存在对应的hash桶内，则直接进行访问即可。





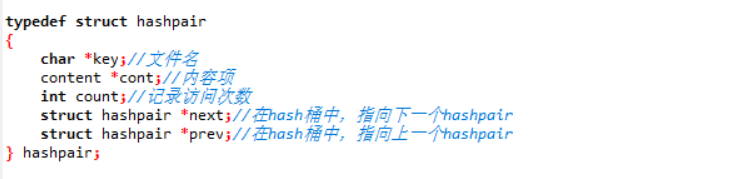


对访问页面总数、缺页数、缺页率进行统计并打印。

* 1. **LFU算法：**

通过记录缓存中每个页面内容被使用的次数来置换缓存中使用次数最少的页面内容。

类比LRU算法，利用链表结构。

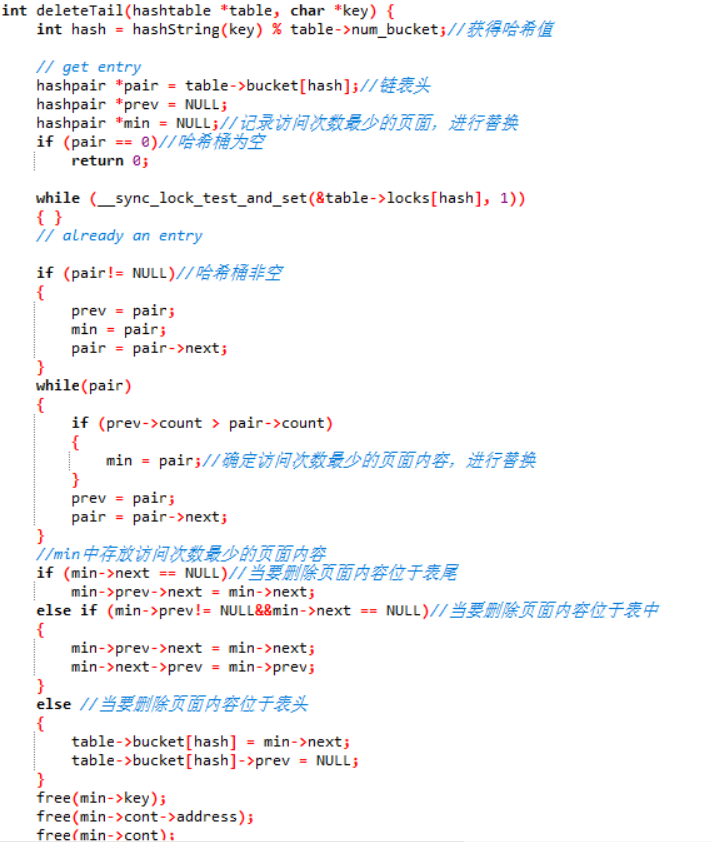


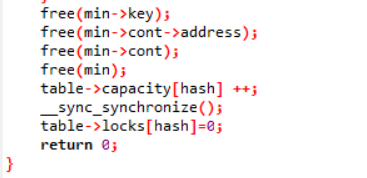
给hashpair添加一个新的变量count，用于记录每个页面被访问的次数。



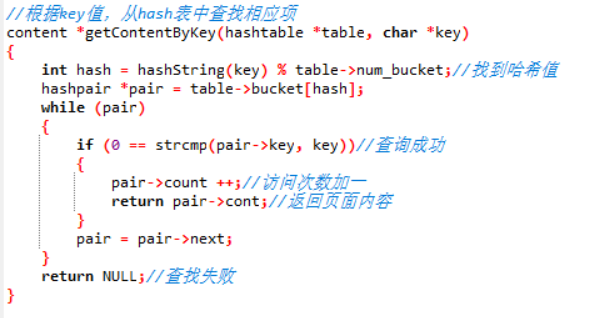


每次向哈希桶中放入新的页面时，将页面的访问次数count置为1。



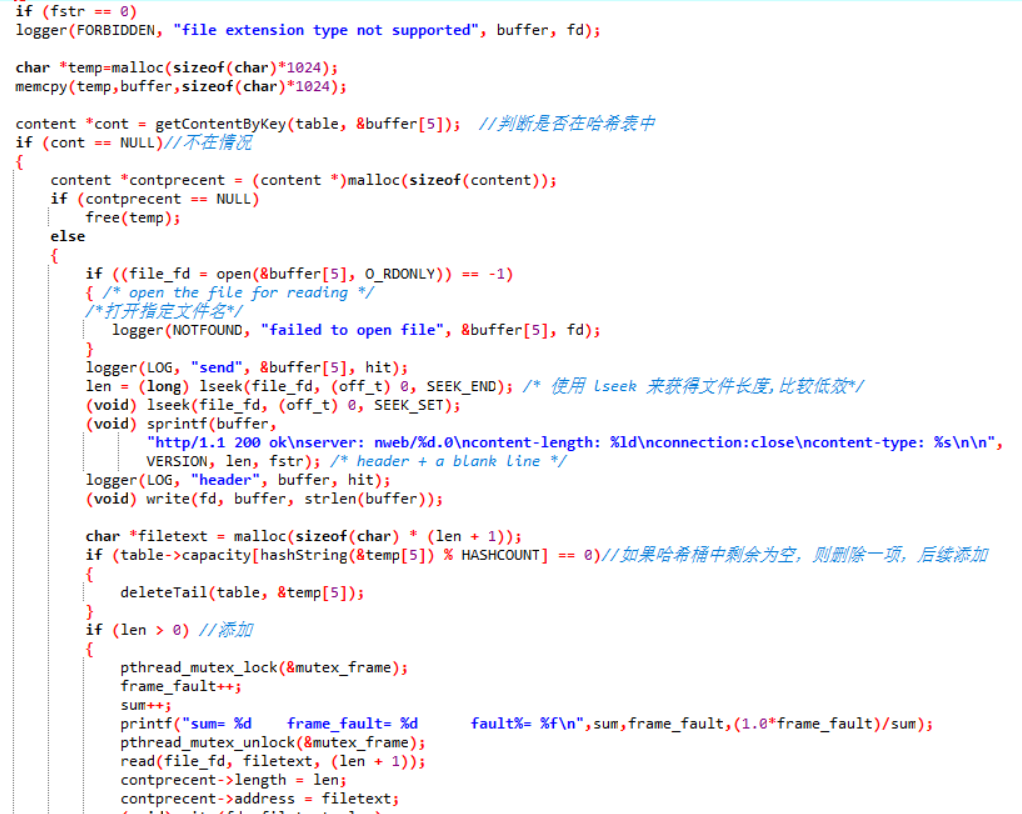


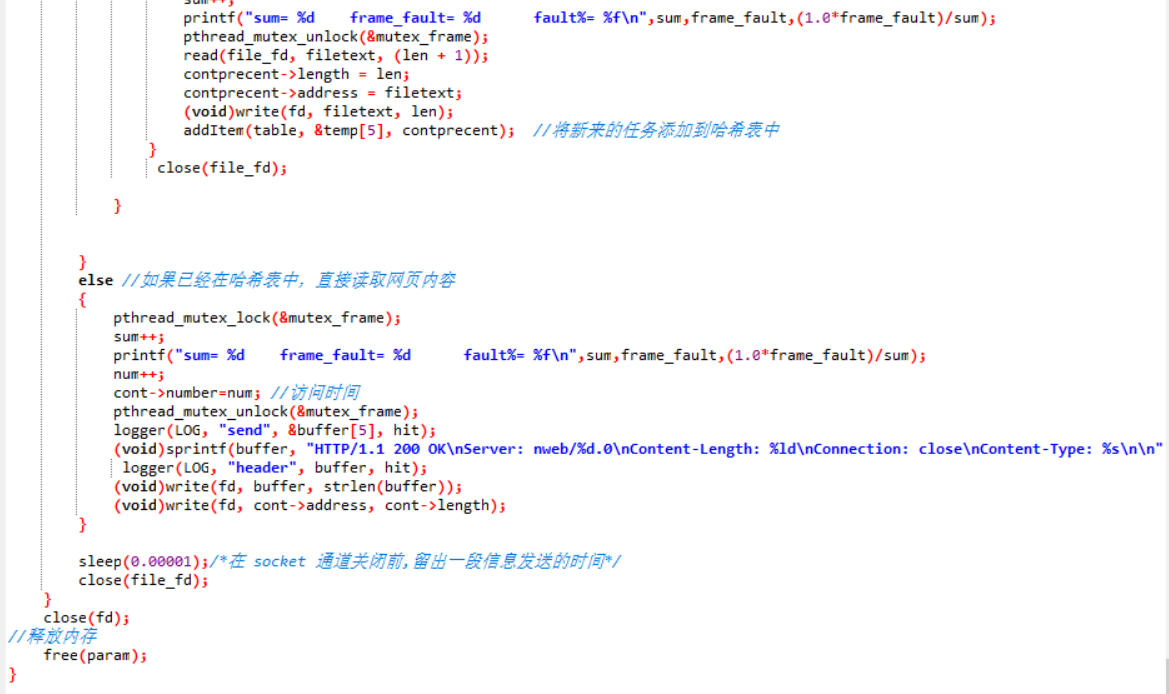
缺页替换时，进行遍历查询，找到访问次数最少的页面，将其进行删除替换。



每次访问页面，若页面在缓存中，则将其访问次数加一。

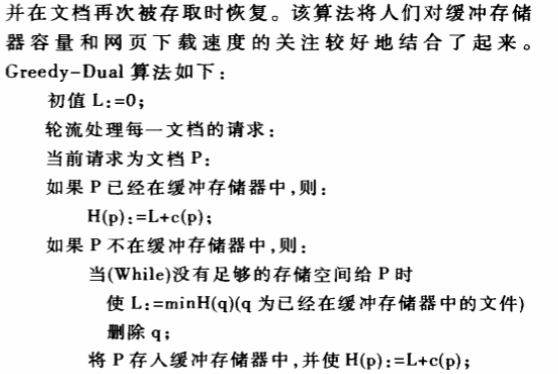
修改web函数：





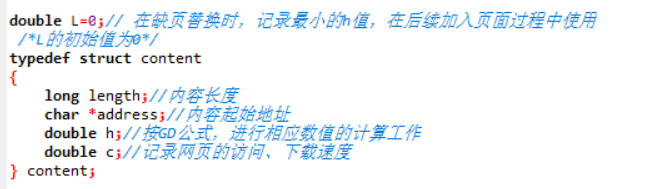
* 1. **GD算法：**

查询相关文献可知，GD相关算法如下图所示：

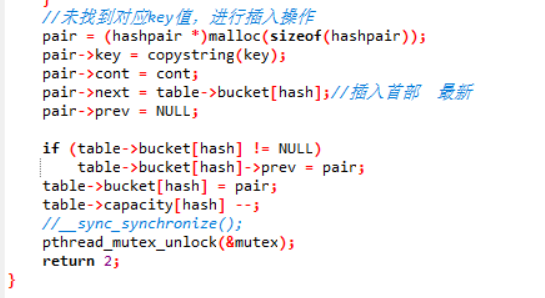


设置全局变量L，用于上图算法中L相关值的记录工作。

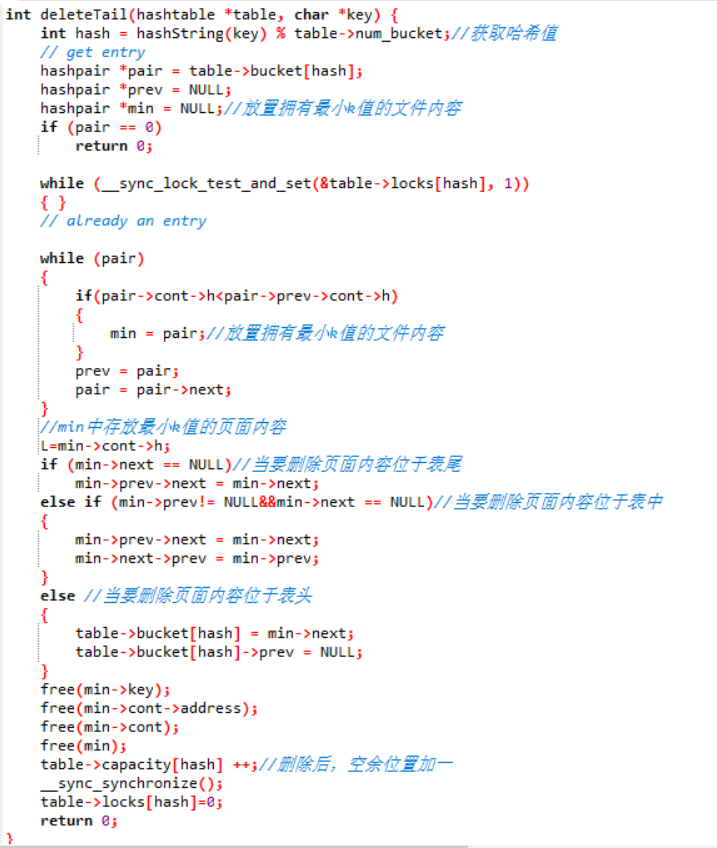
对页面内容进行相关变量的添加，用于相关值的计算操作。







缺页替换，添加相应页面时，将页面置入相应hash桶链表的表头。

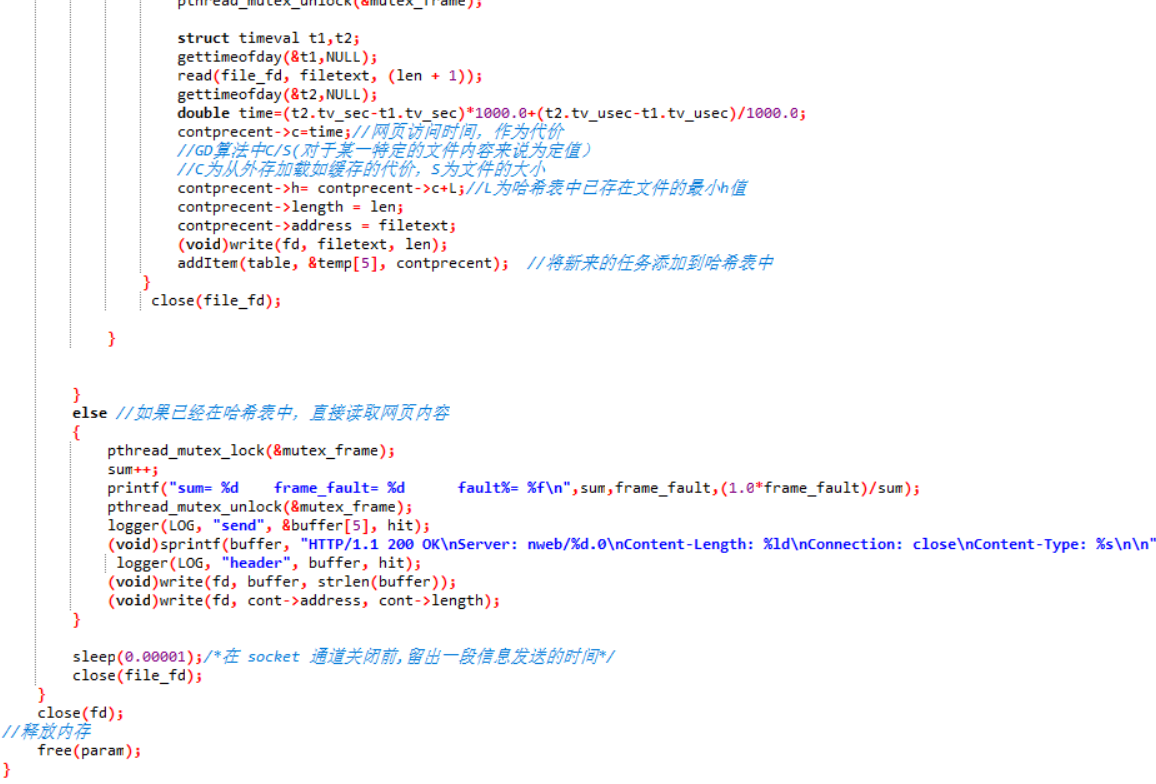


缺页替换，进行相关页面的删除替换时，选择根据GD公式计算得到的h值最小的页面进行删除替换。



修改web函数：



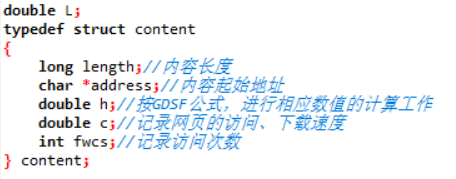


L为全局变量，用于记录缺页替换时所被替换的最小的h值，c（p）以网页的读取速度为代价，进行GD的计算。每发生缺页替换，便进行一次相应GD函数的计算工作。

* 1. **GDSF算法：**

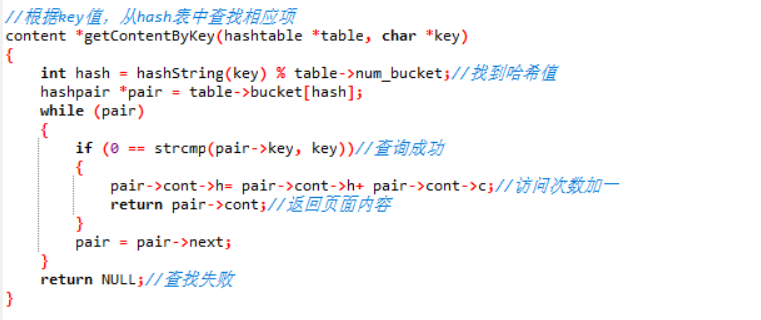
GD算法虽说是一个相当优秀的算法，但也存在不足，即它无法区分经常点击和很少点击的文档。一个常常被点击的文档在缓冲存储器列表中的位置和权重同一个新进入的文档完全相同。这一缺点是从LRU算法中继承来的。

GDSF算法中引入访问次数因子。



在GD算法基础上，再次引入一个新的变量，fwcs（访问次数）。

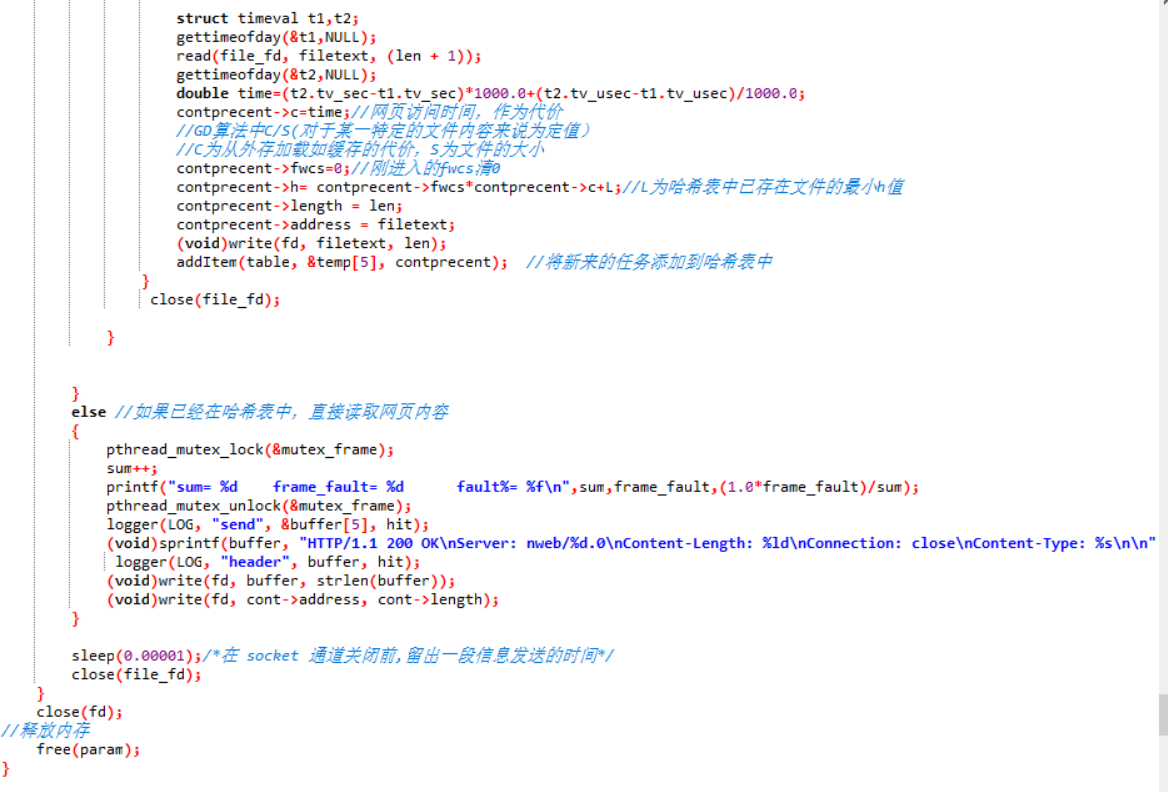
其addItem函数与deleteTail函数与GD大致相同。



引入访问次数，因为访问次数作为访问代价的倍数，故每在缓存区中访问一次，即加一次对应的访问代价。

修改web函数：





**题目二：**

1. **实验题目：**

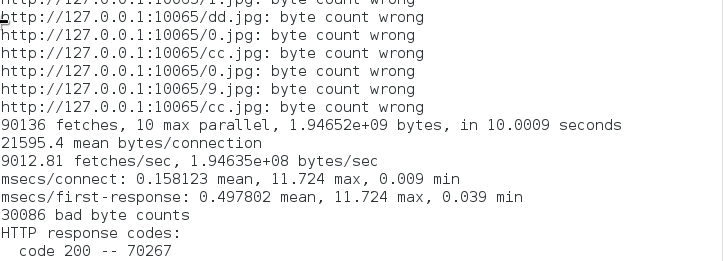
通过实验来评估各个替换算法的好坏，通过服务器缓存命中率、客户端获得请求内容的平均时间等参数，来说明有无Web文件缓存对Web服务的影响。

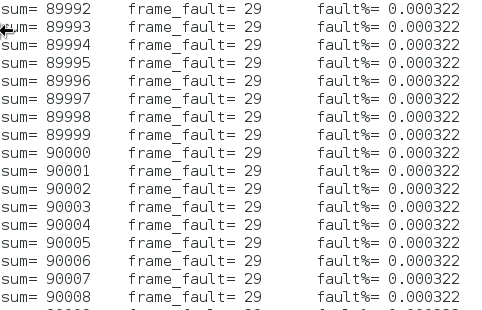
1. **测试结果：**

对15个页面进行测试抓取工作。

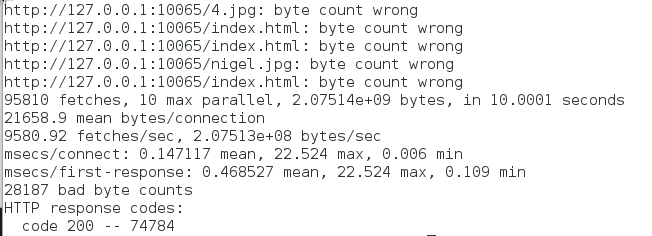
测试命令：./http\_load -parallel 10 -fetches 50 -seconds 10 urls

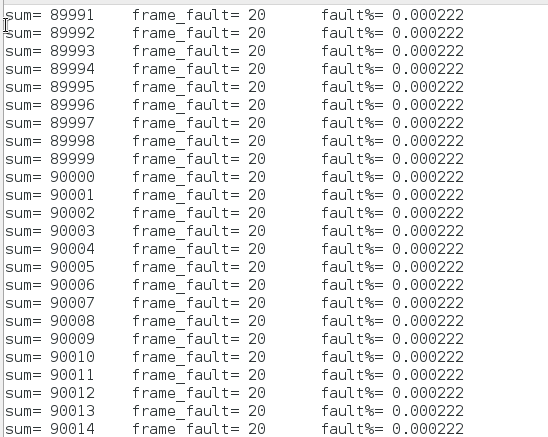
**LRU：**



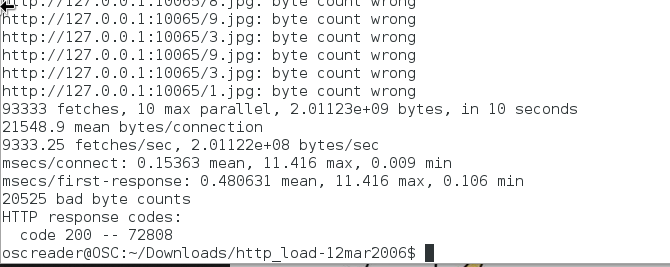


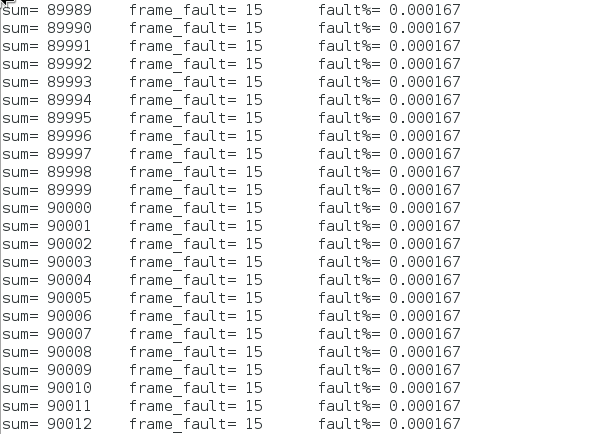
**LFU:**

****

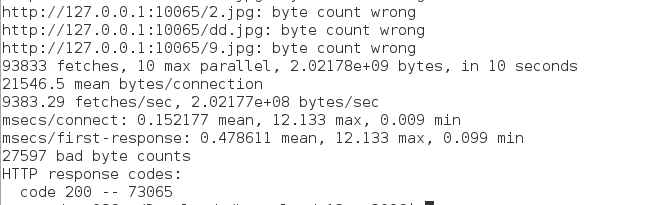
****

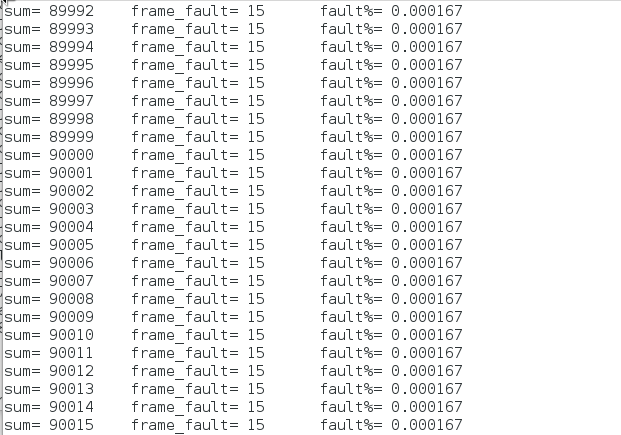
**GD:**

****

****

**GDSF:**

****

****

1. **结果分析：**

通过实验结果分析可以得到，增添web文件按缓存可以提升web服务的性能，客户请求文件内容花费的时间减少。

在测试中，gd与gdsf的缺页率要少于lru与lsu。由于实验中访问页面较少，四种算法缺页率相差并不是很大。从理论分析进行，gdsf应在一定程度上要优于gd算法，因为它引入了文件访问次数因子，与函数策略结合使得替换的时候更加精准。

在实验中出现字节错误的问题，导致部分网页抓取失败。初步认为为网页中图片文件过大，超出所设定的hash桶所能储存字节。因实验时间原因，未能进行进一步问题的改进。

**题目三：**

1. **实验题目：**

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？）。

**答：**LRU算法适用于管理有热度区分的网页服务器。这样，服务器缓存中可以存放当前被用户访问最近即近期最具热度的网页。LFU算法适用于管理网页首页的服务器，网页首页一般被访问次数最多，故放于缓存中，有利于用户访问。GDSF更为适应本实验环境，，因为它引入了文件访问次数因子，与函数策略结合使得替换的时候更加精准。

**附：（lru算法代码）**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/time.h>

#include <time.h>

#include <mqueue.h>

#include <fcntl.h>

#include <sys/msg.h>

#include <semaphore.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <stdbool.h>

#include <pthread.h>

#include <sys/prctl.h>

#define TRUE 1

#define FALSE 0

#define VERSION 23

#define ERROR 42

#define LOG 44

#define FORBIDDEN 403

#define NOTFOUND 404

#ifndef SIGCLD

#define SIGCLD SIGCHLD

#endif

#define HASHCOUNT 20

#define SIZE 40

int num=0;

#define BUFSIZE 8096

pthread\_mutex\_t mutex;

pthread\_mutex\_t mutex\_frame;

typedef struct content

{

long length;//内容长度

char \*address;//内容起始地址

int number;

//此变量用于记录文件进入哈希桶的时间，将其设为全局变量。

//每当有新文件进入桶内时，加一记录，并将其赋值给文件，便于判断文件的进入时间。

} content;

typedef struct hashpair

{

char \*key;//文件名

content \*cont;//内容项

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

struct hashpair \*prev;//在hash桶中，指向上一个hashpair

} hashpair;

typedef struct hashtable

{

hashpair \*\*bucket;

int num\_bucket;//哈希桶的个数

int \*capacity;//每个哈希桶的个数

volatile int\* locks;

} hashtable;

static inline long int hashString(char \*str)//字符串转化为哈希号

{

int hash = 5381;

int c;

while (c = \*str++)

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

while(hash < 0 || hash >= HASHCOUNT)

hash = (hash + HASHCOUNT) % HASHCOUNT;//防止哈希值过大，进行取余

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)

{

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

return 0;

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

return 0;

return 1;

}

int frame\_fault;//全局变量，缺页数量

int sum;//全局变量，总访问数量

hashtable \*table;

// Create hash table

hashtable \*create\_hash(int buckets)

{

// allocate space

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

if (!table)

{

// unable to allocate

return NULL;

}

// setup

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

if (!table->bucket)

{

free(table);

return NULL;

}

memset(table->bucket, 0, buckets \* sizeof(void \*));//清0

table->num\_bucket = buckets;

//初始化锁信号

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

if (!table->locks)

{

free(table);

return NULL;

}

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

table->capacity = (int \*)malloc(buckets \* sizeof(int));//哈希表的大小

int i;

for ( i = 0; i < buckets; i++)

table->capacity[i] = SIZE;//每个哈希桶大小，初始值设为40

return table;

}

//释放hash table中的资源

void freeHashTable(hashtable \*table)

{

if (table == NULL)

{

return;

}

hashpair \*next;

int i;

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

//逐个桶释放

{

hashpair \*pair = table->bucket[i];//每个桶内进行逐个释放

while (pair)

{

next=pair->next;

free(pair->key);

free(pair->cont->address);

free(pair->cont);

free(pair);

pair = next;

}

}

free(table->bucket);

free(table);

}

//add hashtable item

int addItem(hashtable \*table, char \*key, content \*cont)

{

// compute hash on key

int hash = hashString(key) % table->num\_bucket;//获得key在hash表中的位置

// get entry

hashpair \*pair = table->bucket[hash];//找到对应hash桶中的首个pair

pthread\_mutex\_lock(&mutex);

hashpair \*current=NULL;

while (pair)

{

// check for key

if (0 == strcmp(pair->key, key) && isEqualContent(pair->cont, cont))

{

//exist

pthread\_mutex\_unlock(&mutex);

return 1;

}

if (0 == strcmp(pair->key, key) && !isEqualContent(pair->cont, cont))

{

//key is same,but content is different

//重名替换

free(pair->cont->address);

free(pair->cont);

pair->cont = cont;

pthread\_mutex\_unlock(&mutex);

return 0;

}

current = pair;

pair = pair->next;

if (pair!= NULL)

pair->prev = current;

}

//未找到对应key值，进行插入操作

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

pair->key = copystring(key);

pair->cont = cont;

pair->next = table->bucket[hash];//插入首部 最新

pair->prev = NULL;

if (table->bucket[hash] != NULL)

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

table->bucket[hash] = pair;

table->capacity[hash] --;

//\_\_sync\_synchronize();

pthread\_mutex\_unlock(&mutex);

return 2;

}

int deleteTail(hashtable \*table, char \*key) {

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

// get entry

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

hashpair \*prev = NULL;

if (pair == 0)

return 0;

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

{ }

// already an entry

while (pair) //检测桶中存在的文件 找到链表最后一个存在的

{

prev = pair;

pair = pair->next;

}

if(prev)//存在链表最后一个文件

{

table->capacity[hash] ++;

free(prev->key);

free(prev->cont->address);

free(prev->cont);

free(prev);

}

\_\_sync\_synchronize();

table->locks[hash]=0;

return 0;

}

//根据key值，从hash表中查找相应项

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

{

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

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

while (pair)

{

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

{

if (pair->prev != NULL && pair->next != NULL)//位于链表中间位置 ，移动到对应桶的链表头

{

pair->prev->next = pair->next;

pair->next->prev = pair->prev;

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

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

pair->prev = NULL;

table->bucket[hash] = pair;

}

else if(pair->prev != NULL && !pair->next)//位于链表尾部位置 ，移动到对应桶的链表头

{

pair->prev->next = pair->next;

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

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

pair->prev = NULL;

table->bucket[hash] = pair;

}

return pair->cont;//查找成功

}

pair = pair->next;

}

return NULL;//查找失败

}

/\*queue status and conditional variable\*/

typedef struct staconv {

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;/\*用于阻塞和唤醒线程池中线程\*/

bool status;/\*表示任务队列状态\*/

} staconv;

/\*Task\*/

typedef struct task {

struct task \*next;/\*指向下一任务\*/

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

void \*arg;/\*函数参数指针\*/

} task;

/\*Task Queue\*/

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;/\*队列中任务个数\*/

} taskqueue;

/\*Thread\*/

typedef struct thread {

int id;/\*线程id\*/

pthread\_t pthread;/\*封装的POSIX线程\*/

struct threadpool \*pool;/\*与线程池绑定\*/

} thread;

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;/\*表示线程池是否还存活\*/

} threadpool;

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

void\* thread\_do(struct thread\* pthread);

/\*初始化任务队列\*/

void init\_taskqueue(taskqueue\* queue)

{

queue->len = 0;

queue->front =NULL;

queue->rear=NULL;

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

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

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

pthread\_cond\_init(&(queue->has\_jobs->cond), NULL);

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

}

/\*线程池初始化函数\*/

struct threadpool\* initThreadPool(int num\_threads) {

//创建线程空间

threadpool\* pool;

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

pool->num\_threads=0;

pool->num\_working=0;

//初始化互斥量和条件变量

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

pthread\_cond\_init(&(pool->threads\_all\_idle),NULL);

pool->is\_alive=TRUE;

//初始化任务队列

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

//创建线程数组

pool->threads=(struct thread\*\*)malloc(num\_threads \* sizeof(struct thread\*));

//创建线程

int i;

for(i=0;i<num\_threads;i++)

{

create\_thread(pool,pool->threads[i],i);//i为线程id

}

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

return pool;

}

/\*将任务加入队列\*/

void push\_taskqueue(taskqueue\* queue,task\* curtask) //任务队列 当前任务

{

pthread\_mutex\_lock(&(queue->mutex));//用于互斥读/写任务队列

if(queue->len==0)//任务队列中任务数为0

{

queue->front=curtask;

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

queue->rear=curtask;

}

else

{

queue->rear->next=curtask;

queue->rear=queue->rear->next;

}

queue->len++;

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

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

pthread\_cond\_signal(&(queue->has\_jobs->cond));//用于唤醒线程池中的线程

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

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

}

/\*向线程池中添加任务\*/

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

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

}

/\*等待当前任务全部运行完\*/

void waitThreadPool(threadpool\* pool) {

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);

}

/\*销毁任务队列\*/

void destory\_taskqueue(taskqueue\* queue)

{

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

task\* curtask;

while(queue->front != NULL)

{

curtask = queue->front;

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

free(curtask);

}

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

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

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

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

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

pthread\_cond\_destroy(&(queue->has\_jobs->cond));

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

}

/\*销毁线程池\*/

void destoryThreadPool(threadpool\* pool){

//如果当前任务队列中有任务，需等待任务队列为空，并且运行线程执行完任务后

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

waitThreadPool(pool);//等待当前任务全部运行完

destory\_taskqueue(&pool->queue);//销毁任务队列

//销毁线程指针数组，并释放所有为线程池分配的内存

int i;

for (i = 0; i < pool->num\_threads; ++i)

free(pool->threads[i]);

free(pool->threads);

pool->is\_alive = FALSE;

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

pthread\_mutex\_destroy(&(pool-> thcount\_lock));//销毁锁

pthread\_cond\_destroy(&(pool->threads\_all\_idle));//销毁条件变量

}

/\*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;

}

/\*从任务队列头部提取任务，并在队列中删除此任务\*/

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

{

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

if (queue->len != 0)//队列中存在任务

{

task\* curtask = queue->front;

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

queue->len--;

if (queue->len == 0)

{

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

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

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

}

else

{

pthread\_cond\_signal(&queue->has\_jobs->cond);//唤醒线程

}

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

return curtask;

}

else

{

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

return NULL;

}

}

/\*创建线程\*/

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

{

pthread=(struct thread\*)malloc(sizeof(struct thread));

if(pthread==NULL)

{

perror("creat\_thread(): Could not allocate memory for thread\n");

return -1;

}

pthread->pool=pool;

pthread->id=id;

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

pthread\_detach(pthread->pthread);

return 0;

}

/\*线程运行的逻辑函数\*/

void\* thread\_do(struct thread\* pthread){

/\*设置线程名字\*/

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

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

prctl(PR\_SET\_NAME,thread\_name);

/\*获得线程池\*/

threadpool\* pool=pthread->pool;

/\*在线程池初始化时，用于已经创建线程的计数，执行pool->num\_thread++\*/

pthread\_mutex\_lock(&(pool->thcount\_lock));//线程池锁，用于修改线程池中线程数量变量

pool->num\_threads++;

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

/\*线程一直循环往复运行，直到pool->is\_alive变为false\*/

while(pool->is\_alive){

/\*如果任务队列中还有任务，则继续运行，否则阻塞\*/

pthread\_mutex\_lock (&(pool->queue.has\_jobs->mutex));//用于互斥读写任务队列

while ((pool->queue.has\_jobs->status == 0) &&pool->is\_alive)//无任务状态

{

pthread\_cond\_wait (&(pool->queue.has\_jobs->cond), &(pool->queue.has\_jobs->mutex));//阻塞

}

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

if(pool->is\_alive){

/\*执行到此位置，表明线程在工作，需要对工作线程数量进行计数\*/

//pool->num\_working++

pthread\_mutex\_lock (&(pool->thcount\_lock));//线程池锁，用于修改线程池中正在工作线程数量变量

pool->num\_working++;

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

/\*从任务队列的队首提取任务并执行\*/

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

void\* arg;

//take\_taskqueue从任务队列头部提取任务，并在队列中删除此任务

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

if(curtask)

{

func=curtask->function;

arg=curtask->arg;

//执行任务

func(arg);

//释放任务

free(curtask);

}

/\* 执行到此位置，表明线程已经将任务执行完成，需要改工作线程数量\*/

/\*此处还需注意，当工作线程数量为0，表示任务全部完成，要让阻塞在waitThreadPool()函数上的线程继续运行\*/

pthread\_mutex\_lock (&(pool->thcount\_lock));//线程池锁，用于修改线程池中正在工作线程数量变量

pool->num\_working--;

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

if(pool->num\_working == 0)//任务全部完成

pthread\_cond\_signal(&(pool->threads\_all\_idle));//唤醒线程

}

}

/\*运行到此位置表明，线程将要退出，需更改当前线程池中的线程数量\*/

//pool->num\_threads--;

pthread\_mutex\_lock(&(pool->thcount\_lock));//线程池锁，用于修改线程池中线程数量变量

pool->num\_threads--;

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

return NULL;

}

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}};

typedef struct

{

int hit;

int fd;

} webparam;

unsigned long get\_file\_size(const char \*path)

{

unsigned long filesize = -1;

struct stat statbuff;

if (stat(path, &statbuff) < 0)

{

return filesize;

}

else

{

filesize = statbuff.st\_size;

}

return filesize;

}

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

{

int fd;

char logbuffer[BUFSIZE \* 2];

switch (type) {

case ERROR:

(void) sprintf(logbuffer, "ERROR: %s:%s Errno=%d exiting pid=%d", 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, "FORBIDDEN: %s:%s", s1, s2);

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, "NOT FOUND: %s:%s", s1, s2);

break;

case LOG:

(void) sprintf(logbuffer, " INFO: %s:%s:%d", s1, s2, socket\_fd);

break;

}

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

if ((fd = open("webserver.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 == ERROR || type == NOTFOUND || type == FORBIDDEN) exit(3);

}

/\* this is a web thread, so we can exit on errors \*/

/\*此函数完成了webserver主要功能，它首先解析客户端发送的消息，然后从中获取客户端请求的文件名，

然后根据文件名从本地将此文件读入缓存，并生成相应的HTTP响应消息；最后通过服务器与客户端的socket

通道想客户端返回HTTP响应消息\*/

void web(void \*data) {

int fd;

int hit;

int j, file\_fd, buflen;

long i, ret, len;

char \*fstr;

char buffer[BUFSIZE + 1]; /\* static so zero filled \*//\*设置静态缓冲区\*/

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

fd = param->fd;

hit = param->hit;

ret = read(fd, buffer, BUFSIZE); /\* read web request in one go \*//\*从连接通道中读取客户端的请求消息\*/

if (ret == 0 || ret == -1)

{ /\* read failure stop now \*/

/\*如果读取客户端消息失败，则向客户端发送HTTP失败响应信息\*/

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

}

else

{

if (ret > 0 && ret < BUFSIZE) /\* return code is valid chars \*//\*设置有效字符串，即将字符串尾部表示为0\*/

buffer[ret] = 0;/\* terminate the buffer \*/

else buffer[0] = 0;

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

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

buffer[i] = '\*';

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

/\*判断客户端HTTP请求消息是否为GET类型，如果不是则给出响应的响应消息\*/

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

{

logger(FORBIDDEN, "only simple get operation supported", buffer, fd);

}

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;

}

}

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);

}

if (!strncmp(&buffer[0], "GET /\0", 6) || !strncmp(&buffer[0], "GET /\0", 6)) /\* convert no filename to index file \*/

/\*如果请求消息中没有包含有效的文件名，则使用默认文件名index.html\*/

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

/\* work out the file type and check we support it \*/

/\*检查请求服务器类型是否支持\*/

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);

char \*temp=malloc(sizeof(char)\*1024);

memcpy(temp,buffer,sizeof(char)\*1024);

content \*cont = getContentByKey(table, &buffer[5]); //判断是否在哈希表中

if (cont == NULL)//不在情况

{

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

if (contprecent == NULL)

free(temp);

else

{

if ((file\_fd = open(&buffer[5], O\_RDONLY)) == -1)

{ /\* open the file for reading \*/

/\*打开指定文件名\*/

logger(NOTFOUND, "failed to open file", &buffer[5], fd);

}

logger(LOG, "send", &buffer[5], hit);

len = (long) lseek(file\_fd, (off\_t) 0, SEEK\_END); /\* 使用 lseek 来获得文件长度,比较低效\*/

(void) lseek(file\_fd, (off\_t) 0, SEEK\_SET);

(void) sprintf(buffer,

"http/1.1 200 ok\nserver: nweb/%d.0\ncontent-length: %ld\nconnection:close\ncontent-type: %s\n\n",

VERSION, len, fstr); /\* header + a blank line \*/

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

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

char \*filetext = malloc(sizeof(char) \* (len + 1));

if (table->capacity[hashString(&temp[5]) % HASHCOUNT] == 0)//如果哈希桶中剩余为空，则删除一项，后续添加

{

deleteTail(table, &temp[5]);

}

if (len > 0) //添加

{

pthread\_mutex\_lock(&mutex\_frame);

frame\_fault++;

sum++;

printf("sum= %d frame\_fault= %d fault%= %f\n",sum,frame\_fault,(1.0\*frame\_fault)/sum);

pthread\_mutex\_unlock(&mutex\_frame);

read(file\_fd, filetext, (len + 1));

contprecent->length = len;

contprecent->address = filetext;

(void)write(fd, filetext, len);

addItem(table, &temp[5], contprecent); //将新来的任务添加到哈希表中

}

close(file\_fd);

}

}

else //如果已经在哈希表中，直接读取网页内容

{

pthread\_mutex\_lock(&mutex\_frame);

sum++;

printf("sum= %d frame\_fault= %d fault%= %f\n",sum,frame\_fault,(1.0\*frame\_fault)/sum);

num++;

cont->number=num; //访问时间

pthread\_mutex\_unlock(&mutex\_frame);

logger(LOG, "send", &buffer[5], hit);

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

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

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

(void)write(fd, cont->address, cont->length);

}

sleep(0.00001);/\*在 socket 通道关闭前,留出一段信息发送的时间\*/

close(file\_fd);

}

close(fd);

//释放内存

free(param);

}

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

{

signal(SIGPIPE,SIG\_IGN);

frame\_fault=0;

sum=0;

table=create\_hash(HASHCOUNT);

struct threadpool\* pool;

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 \*/

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);

}

pool=initThreadPool(40);//设置线程池中线程个数为40

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

/\* setup the network 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);

//初始化线程属性,为分离状态

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

pthread\_t pth;

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);

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

length = sizeof(cli\_addr);

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

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

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

param->hit=hit;

param->fd=socketfd;

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

curtask->next=NULL;

curtask->function=web;//函数指针

curtask->arg=(void\*)param;//函数参数指针

addTask2ThreadPool(pool,curtask);

}

destoryThreadPool(pool);

freeHashTable(table);

}