```
#include <string.h>
#include <sys/time.h>

#ifdef HASHTHREADED
    #include <pthread.h>
    #include <semaphore.h>
#endif

typedef struct content{
    int length;                     //内容长度
    void* address;                  //内容起始地址
} content;

typedef struct hashpair{
    char* key;                      //key 值为文件名
    content * cont;                 //内容项
    struct hashpair* next;          //在 hash 桶中，指向下一个 hashpair
} hashpair;

typedef struct hashtable{
    hashpair ** bucket;
    int num_bucket;
#ifdef HASHTHREAD
    volatile int * locks;           //对 hash 桶进行加锁
//  volatile int   lock;            //对 hashtable 进行加锁
#endif
} hashtable;

// 字符串的 hash 编码算法-djb2

static inline long int hashString(char * str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    return hash;
}

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，若相同，则返回 1；若不同，则返回 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;
}
//创建一个 hashtable
hashtable* createHashTable(int num_bucket){
    //创建一个 hashtable
    hashtable* table=(hashtable *) malloc(sizeof(hashtable));
    if(NULL==table){
        return NULL;
    }
    //根据 num_bucket，创建 hash 桶指针
    table->bucket=(hashpair**) malloc(num_bucket*sizeof(void*));
    if(!table->bucket){
        free(table);
        return NULL;
    }
    memset(table->bucket,0,num_bucket*sizeof(void*));
    table->num_bucket=num_bucket;
//初始化锁信号
#ifdef HASHTHREAD
    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));
#endif
    return table;
}
//释放 hashtable 中的资源
void freeHashTable(hashtable* table){
    if(table==NULL)
        return;
    hashpair* next;
    for (int i=0; i< table->num_bucket; i++) {
        //逐个释放 hash 桶
        hashpair* pair=table->bucket[i];
        while(pair){
            next=pair->next;
            //删除 pair，释放资源
```

```
            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);
}


//向 hashtable 中增加一个 item=<key,content>
//返回 1，表示要添加项已经在 hash 表中存在，
//返回 0，表示仅是 key 相同，而 content 不同
//返回 2，表示如果不存在 key，则正常加入 hashtable 中
int addItem(hashtable* table,char* key, content* cont){
    //根据 hash 值，计算 key 在 hash table 中的位置
    int hash=hashString(key)% table->num_bucket;
    //检索此项的 key 是否已经存在，如果已经存在，则在 hash 桶中寻找此项值，并对其进行替换
    hashpair* pair=table->bucket[hash];

#ifdef HASHTHREAD
    //利用 GCC 中的函数，加自旋锁
    while (__sync_lock_test_and_set(&table->locks[hash], 1)) {
        //GCC 内部函数，原子操作，将 table->locks[hash]中的值设置为 1，并返回原来的数值
//当第一次进入时，返回 0，而同时第二次进入则为 1，因此后面进入的线程获得值均为 1，导致在此处忙需等待

    }
#endif
    while(pair!=0){
        if(0==strcmp(pair->key,key) && isEqualContent(pair->cont, cont)) //已经存在
            return 1;
        if(0==strcmp(pair->key,key) && !isEqualContent(pair->cont, cont)) {
            //仅是 key 相同，需进行 content 替换
            free(pair->cont->address);
            free(pair->cont);

            pair->cont=cont;
            return 0;
        }
        pair=pair->next;
    }
    //否则在 hashtable 中不存在，在 hashtable 中新建一个项，并将其插入 hash 桶首部
```

```
    pair=(hashpair*) malloc(sizeof(hashpair));
    pair->key=copystring(key);
    pair->cont=cont;
    pair->next=table->bucket[hash];
    table->bucket[hash]=pair;

#ifdef HASHTHREAD
    //解锁
    __sync_synchronize(); // memory barrier
    table->locks[hash] = 0;
#endif
     return 2;
}

//从 hashtable 中删除指定 key 的对应项
//如果在 hashtable 中未发现此项，则返回 0
//如果在 hashtable 中发现并成功删除，则返回 1
int delItem(hashtable* table,char* key){
    //根据 hash 值计算 key 在 hash table 中的位置
    int hash=hashString(key)% table->num_bucket;
    //检索此项的 key 是否已经存在，如果已经存在，则在 hash 桶中寻找此项值，并将其替换
    hashpair* pair=table->bucket[hash];
    hashpair* prev=NULL; //记录 hash 桶中的前一项数值
    if(pair==0) //此 key 不存在
        return 0;
#ifdef HASHTHREAD
    //利用 GCC 中的函数加自旋锁
    while (__sync_lock_test_and_set(&table->locks[hash], 1)) {
        //GCC 内部函数，原子操作，将 table->locks[hash]中的值设置为 1，并返回原来的数值
//当第一次进入时，返回 0，而同时第二次进入则为 1，因此后面进入的线程获得值均为 1，导致在此处忙需等待
    }
#endif
    //遍历 hash 桶
    while(pair!=0){
        if(0==strcmp(pair->key, key)){
            //在 hash 桶中找到匹配的 key，更改 hash 桶链表
            if(!prev)//在 hash 桶中的第一项
                table->bucket[hash]=pair->next;
            else //在 hash 桶中的其他位置
                prev->next=pair->next;
            //删除 pair，释放资源
            free(pair->key);
            free(pair->cont->address);
            free(pair->cont);
            free(pair);
            return 1;
        }
```

```
        //运动到hash桶的下一项
        prev=pair;
        pair=pair->next;
    }
#ifdef HASHTHREAD
    //解锁
    __sync_synchronize(); // memory barrier
    table->locks[hash] = 0;
#endif
    return 0;
}

//根据key值，则从hash table中查找相应项
//如果没有找到，则返回NULL
content* getContentByKey(hashtable* table, char* key){
    //根据hash值计算key在hash table中的位置
    int hash=hashString(key)% table->num_bucket;
    //检索此项的key是否已经存在，如果已经存在，则在hash桶中寻找此项值
    hashpair* pair=table->bucket[hash];

    while(pair){
        if(0==strcmp(pair->key, key))
            return pair->cont;
        pair=pair->next;
    }
    return NULL;
}

#define NUMTHREADS 8
#define HASHCOUNT 1000000

typedef struct threadinfo {hashtable *table; int start;} threadinfo;

void * thread_func(void *arg){
    threadinfo *info = arg;
    char buffer[512];
    int i = info->start;
    hashtable *table = info->table;
    free(info);
    for(;i<HASHCOUNT;i+=NUMTHREADS) {
        sprintf(buffer,"%d",i);
        content* cont=malloc(sizeof(content));
        cont->length=rand()% 2048;
        cont->address=malloc(cont->length);
        addItem(table, buffer, cont);
    }
}
```

```
int main(void){
    hashtable* table=createHashTable(HASHCOUNT);
    srand((unsigned)time(NULL)); // 初始化随机种子
    // hash a million strings into various sizes of table
     struct timeval tval_before, tval_done1, tval_done2, tval_writehash, tval_
readhash;
    gettimeofday(&tval_before, NULL);
    int t;
    pthread_t * threads[NUMTHREADS];
    for(t=0;t<NUMTHREADS;++t) {
        pthread_t * pth = malloc(sizeof(pthread_t));
        threads[t] = pth;
        threadinfo *info = (threadinfo*)malloc(sizeof(threadinfo));
        info->table = table; info->start = t;
        pthread_create(pth,NULL,thread_func,info);
    }
    for(t=0;t<NUMTHREADS;++t) {
        pthread_join(*threads[t], NULL);
    }
    gettimeofday(&tval_done1, NULL);
    int i,j;
    int error = 0;
    char buffer[512];
    for(i=0;i<HASHCOUNT;++i) {
        sprintf(buffer,"%d",i);
        getContentByKey(table,buffer);

    }

    gettimeofday(&tval_done2, NULL);
    timersub(&tval_done1, &tval_before, &tval_writehash);
    timersub(&tval_done2, &tval_done1, &tval_readhash);
    printf("\n%d threads.\n",NUMTHREADS);
    printf("Store %d ints by string: %ld.%06ld sec, read %d ints: %ld.%06ld
sec\n",HASHCOUNT,
        (long int)tval_writehash.tv_sec, (long int)tval_writehash.tv_usec, HASHCOUNT,
        (long int)tval_readhash.tv_sec, (long int)tval_readhash.tv_usec);

    freeHashTable(table);

    return 0;
}
```

3.3　Web 页面的缓存置换算法

3.1 节中的第（2）个问题（由于内存有限，因此不可能把所有文件都缓存到内存中，那么如何设计缓存中的页面替换策略？是缓存内容置换问题。即需使经常使用的文件内容