Assignment 2

Jason Medcoff

March 6, 2018

1 Problem 1

To begin, we would like to implement a threaded hash table with chaining. To perform chaining, we first must implement a list. In pure C, without classes, we resort to structs and pointers. The list is implemented to be generic, using void pointers to store data.

```
/* ------------------------------- */
/* ----------- List ops ----------- */
/* ------------------------------- */

typedef struct node {
    void *data; //generic data
    struct node *next; //successor node
} node;

node* node_create(void *data, node *next) {
    node* newnode = (node*)malloc(sizeof(node));
    newnode->data = data;
    newnode->next = next;
    return newnode;
}

node* node_push(node* head, void *data) {
    node* newnode = node_create(data, head);
    head = newnode;
    return head;
}

node* node_append(node* head, void *data) {
    node* current = head;
    while (current->next != NULL)
        current = current->next;
    node* newnode = node_create(data, NULL);
```

```
    current->next = newnode;
    return head;
}

node* node_search(node* head, void *data) {
    node* current = head;
    while (current != NULL) {
        if (current->data == data)
            return current;
        current = current->next;
    }
    return NULL;
}

void node_dispose(node* head) {
    node *current, *temp;
    if (head != NULL) {
        current = head->next;
        head->next = NULL;
        while (current != NULL) {
            temp = current->next;
            free(current);
            current = temp;
        }
    }
}
```

The typical list interface is found, with an additional memory management operation, `node_dispose`, to free the list.

Looking ahead to problem 2, we know we will need to place instances of a 15-puzzle into our hash table. Thus, to simplify implementation, we will construct the hash table specifically for the data structure chosen for the 15-puzzle. For simplicity, a puzzle state will be represented as a 4×4 array, containing the numbers 0 through 15. The address occupied by 0 represents the empty tile, whereas the remaining numbers represent the tiles in order, i.e., 1 resides in the upper left corner, 4 in the upper right, 13 in the bottom left, etc. This arrangement allows operations on puzzle states to perform very fast, nearly indistinguishable from constant time. This will become relevant when we are moving tiles around to solve the game.

In the hash table implementation, we will consider a set of entries and a set of pthread locks, with each lock associated with a size `k` block of entries.

```
typedef struct entry {
    unsigned int key;
    node* values;
```

```
} entry;

typedef struct hashtable { // of size P
    int num_entries;
    pthread_rwlock_t locks[NLOCKS];
    //pthread_mutex_t locks[NLOCKS];
    entry** table;
} hashtable;
```

To create a new hashtable, we will allocate the appropriate memory for the entries and initialize all entry keys to zero, and all values to `NULL`. The values will eventually be a list of all states associated with the respective keys. As with the list, we must create a routine to free a created table from memory. However, here we must also take care to dispose of each pthread lock we had created.

```
// init table with keys=0 and no values
hashtable* create_table() {
    entry** tab = malloc(P*sizeof(entry*));
    hashtable* newtable = malloc(sizeof(hashtable));
    newtable->table = tab;
    for (int i=0; i<P; i++) {
        newtable->table[i]->key = 0;
        newtable->table[i]->values = NULL;
    }
    for (int j=0; j<NLOCKS; j++) {
        //pthread_mutex_init(&newtable->locks[j], NULL);
        pthread_rwlock_init(&newtable->locks[j], NULL);
    }
    return newtable;
}

void free_table(hashtable* h) {
    for (int i=0; i<P; i++) {
        node_dispose(h->table[i]->values);
    }
    for (int j=0; j<NLOCKS; j++) {
        //pthread_mutex_destroy(&h->locks[j]);
        pthread_rwlock_destroy(&h->locks[j]);
    }
    free(h->table);
    free(h);
}
```

Among the most challenging tasks for problem 1 is designing a somewhat intelligent hash function for gamestates. While chaining removes the need for extreme cleverness, a

good hash function will still try to evenly distribute values among the table. Efficiency will also be of great importance. Here we will take the product of the first row of the puzzle, and modulo by a number `P`. Namely, the maximum possible product is $32670 = 12 \cdot 13 \cdot 14 \cdot 15$, and we will modulo by `P` $= 10007$, a prime number.

```
unsigned int hash_state(int a[4][4]) {
    unsigned int key = 1;
    int i;
    for (i=0; i<4; i++) {
        key *= a[0][i];
    }
    return key % P;
}
```

Next, we can now design functions to add values to the table, as well as check the table for existence of a particular value.

```
void add_to_table(hashtable* h, int a[4][4]) {
    unsigned int key = hash_state(a);
    int tn = (int) floor((key/P)*NLOCKS);
    pthread_rwlock_wrlock(&h->locks[tn]);
    //pthread_mutex_lock(&h->locks[tn]);
    h->table[key]->key = key;
    node_append(h->table[key]->values, a);
    h->num_entries++;
    pthread_rwlock_unlock(&h->locks[tn]);
    //pthread_mutex_unlock(&h->locks[tn]);
}

int check_table(hashtable* h, int a[4][4]) {
    unsigned int key = hash_state(a);
    int tn = (int) floor((key/P)*NLOCKS);
    pthread_rwlock_rdlock(&h->locks[tn]);
    //pthread_mutex_lock(&h->locks[tn]);
    node* res = node_search(h->table[key]->values, a);
    pthread_rwlock_unlock(&h->locks[tn]);
    //pthread_mutex_unlock(&h->locks[tn]);
    if (res)
        return 1;
    return 0;
}
```

Note that in all code above, mutex and rwlocks are present, with one or the other commented out to allow switching between the two used in implementation.

For testing purposes, we want to try hashing different objects and observe the running time. To do this, we must create objects the table can hash. While the puzzle states will be as described above, we can be more lenient with regards to testing, since the hash function will operate on a 2D array by taking the product of the first row. So, our function to create states for testing need only worry about putting numbers in the first row of a 2D array; we will do so randomly.

```
int** rand_state() {
    // for testing purposes, we need only care about
    // the rop row
    int a[4][4];
    srand(time(NULL));
    for (int i=0; i<4; i++) {
        a[0][i] = rand() % 20; // this doesn't matter
    }
    return a;
}
```

The driver for problem 1 can be written. We create 10,000 entries, then hash them and record the time taken.

```
void driver1() {
    hashtable* ht = create_table();
    time_t now = time(NULL);
    for (int i=0; i<10000; i++) {
        add_to_table(ht, rand_state());
    }
    int n = time(NULL) - now;
    printf("time: %d\n", n);
}
```

2 Problem 2

As described above, puzzle states are represented as 4×4 arrays holding the numbers 0 through 15 once.