CS 111 Discussion 1A/D

WEEK 6

Brian Roysar

Lab 2 Q&A

Lab 3: Hash Hash Hash

Project 3

In this lab, we will be creating an application that will handle hash-table insertions in a thread-safe manner

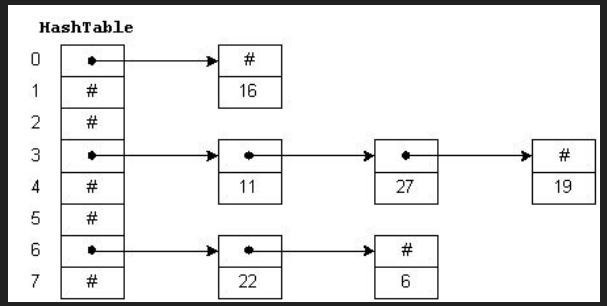
Key points from the spec:

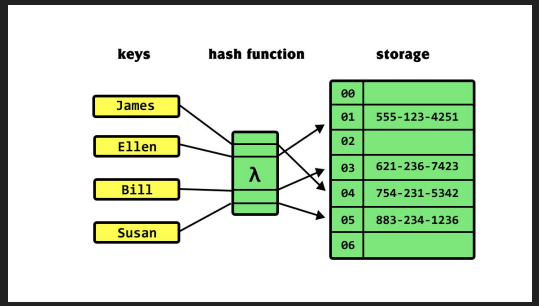
- Will be given a serial hash table implementation, and two additional hash table implementations to modify
  - And then perform comparisons between them
- Hash table uses separate chaining to resolve collisions
- Similar to Java concurrent hash tables implementation
- Will not being changing the algorithm, only adding mutex locks to make current implementation thread safe

Hash Tables

Abstract data structure, implements a dictionary (map), which maps keys → values

- Each key, we use a **hash function** to compute an index into the hash array
- On average, has O(1) insertion, deletion, and finding elements
- Ideally, we have a hash function that ensures each key will have a separate index, however, this is impossible. ***Why?***





Hash Tables

Assume our hash function, h, takes a string, and returns its remainder when divided by three. (i.e. h = k mod 3). Our hashmap aims to map ID to student name

Index	Value
0	
1	
2	

Hash Tables

We want to insert key-value pair (12301, “Adam”)

Index	Value
0	
1	
2	

Hash Tables

We want to insert key-value pair (12301, “Adam”)

12301 % 3 = 1

Index	Value
0	
1	(12301, “Adam”)
2	

Hash Tables

We want to insert key-value pair (12302, “Bella”)

12302 % 3 = 2

Index	Value
0	
1	(12301, “Adam”)
2	

Hash Tables

We want to insert key-value pair (12302, “Bella”)

12302 % 3 = 2

Index	Value
0	
1	(12301, “Adam”)
2	(12302, “Bella”)

Hash Tables

We want to insert key-value pair (12303, “Ivan”)

12303 % 3 = 0

Index	Value
0	
1	(12301, “Adam”)
2	(12302, “Bella”)

Hash Tables

We now want to insert key-value pair (12304, “Jane”)

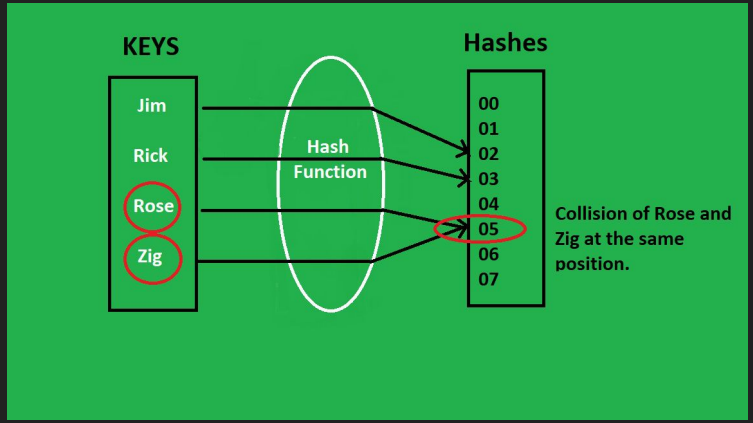
12304 % 3 = 1 – This index has already been used. What do we do?

Index	Value
0	(12303, “Ivan”)
1	(12301, “Adam”)
2	(12302, “Bella”)

Collisions

This is when two or more distinct keys produce the same hash value or index.

- We can’t evict the existing index since our goal is still to store all the information
  - How do we handle this?



Separate Chaining

In the lab, we will be using separate chaining as our collision handling approach

- Each index of the array will store the head of a linked list
- When we encounter a collision, we simply append the colliding insertion into the end of that buckets linked list
- When we want to remove from a hash table with separate chaining, we have to index into the appropriate location, then traverse LL to delete desired element

Hash Tables

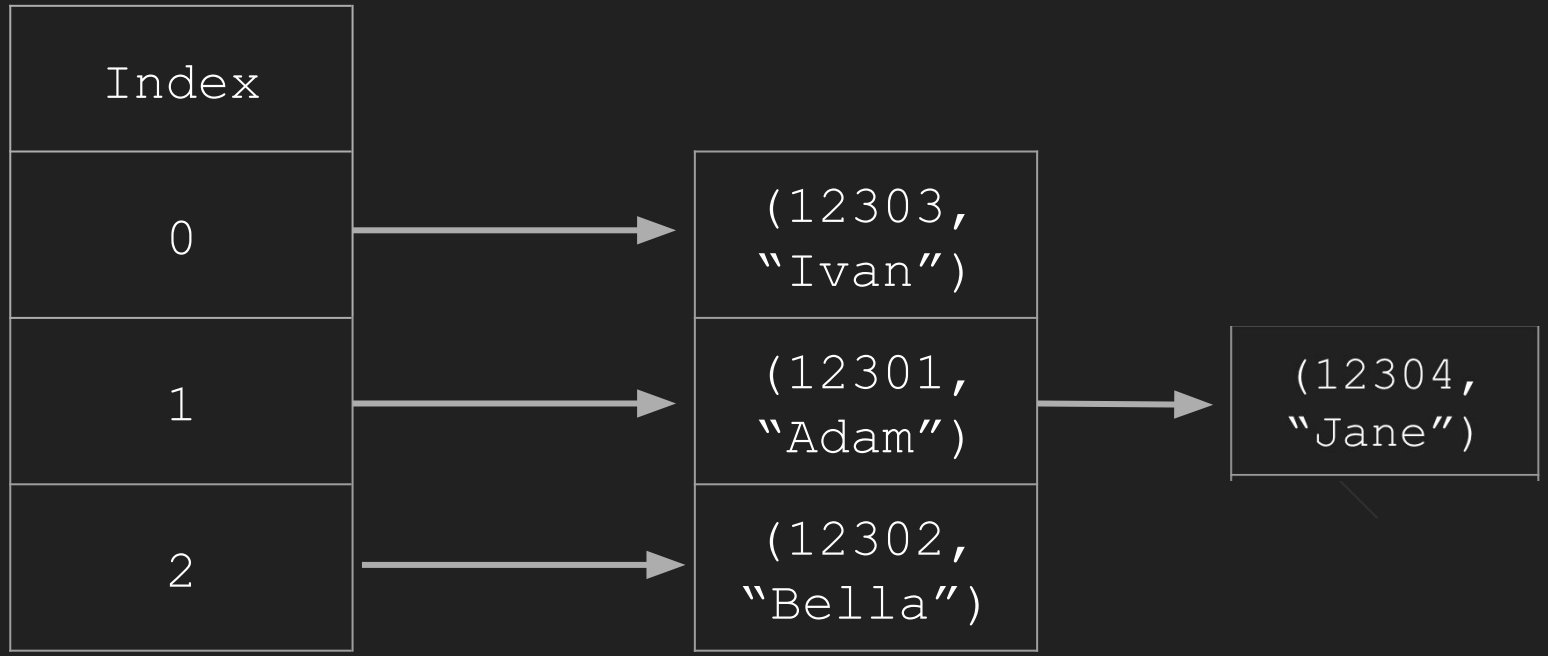
Back to our example, we still want to insert key-value pair (12304, “Jane”)

12304 % 3 = 1

Index	Value
0	(12303, “Ivan”)
1	(12301, “Adam”)
2	(12302, “Bella”)

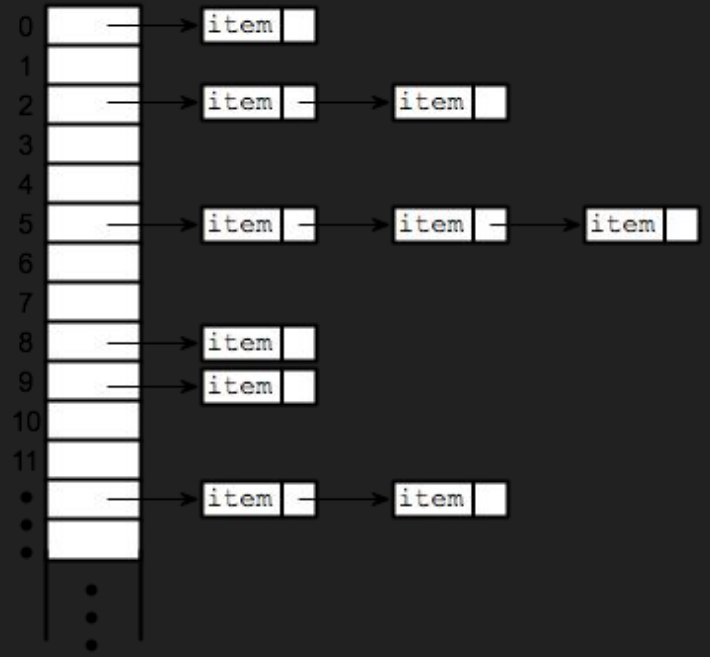
Hash Tables

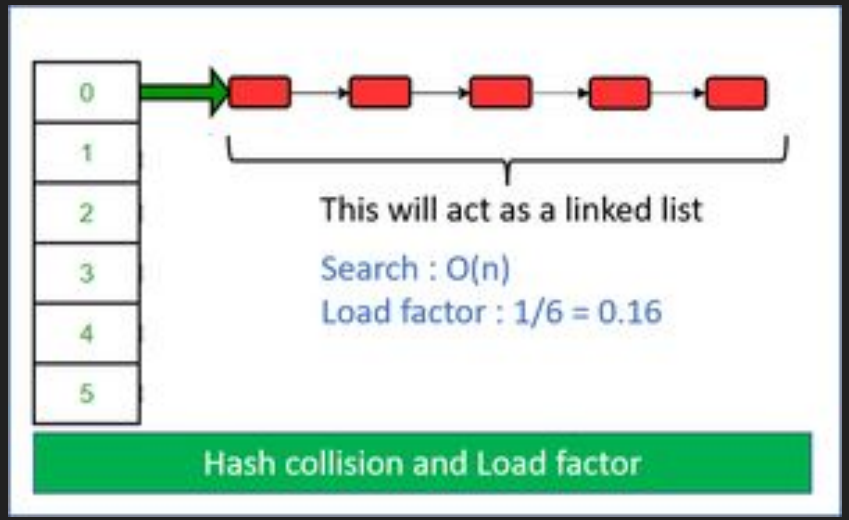
Back to our example, we still want to insert key-value pair (12304, “Jane”) 12304 % 3 = 1



Separate Chaining – is it optimal?

With this approach what issues can arise?





Thread-safe

Thread safe hash tables

In this project, multiple threads will be accessing our hash table at the same time

1. What are the issues that can arise from this
2. How do we prevent it from happening?

Race conditions

A condition where two or more threads/processes access the same data, and at least one of them needs to modify the data

This is due to the nature of threads being scheduled in a non-deterministic manner

Let’s take a look at an example

Race conditions

```c
int account_balance = 1000;

void *perform_transaction(void *thread_id) {
    int amount = 200;
    if (account_balance >= amount) {
        int temp = account_balance;
        temp -= amount;
        account_balance = temp;

    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[2];

    for (int t = 0; t < 2; t++) {
        pthread_create(&threads[t], NULL, perform_transaction, (void *)t);
    }

    for (int t = 0; t < 2; t++) {
        pthread_join(threads[t], NULL);
    }

    printf("Final account balance: %d\n", account_balance);

    return 0;
}
```

We have two threads, both trying to deduct money from the same account.

In a serial flow, we expect the final balance to be 600

How does this code lead to race conditions?

Race conditions

```
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 800
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 800
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 800
briansinambela@Brians-MBP-2 thread_demo % ./race
Final account balance: 600
```

Mutexes

To resolve this problem, we need to **lock** the **critical sections** of our code to ensure race conditions do not occured

- **Critical section:** “a portion of a program's code that accesses shared resources or variables that can be accessed by multiple threads or processes.”

This is where **mutexes** comes in

- Used to ensure exclusive access to the critical section (shared data) between threads (or processes).
- When the lock is set, no other thread can access the locked region of code.

Mutexes

- Suppose one thread has locked a region of code using mutex and is executing that piece of code.
- Now if scheduler decides to do a context switch
  - If the selected thread tries to execute the same region of code that is already locked then it will again go to sleep.
- Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
- Mutex lock will only be released by the thread who locked it.
- So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.
- Hence, this system ensures synchronization among the threads while working on shared resources

Mutexes

```c
int account_balance = 1000;
pthread_mutex_t account_mutex;


void *perform_transaction(void *thread_id) {
    int amount = 200;
    pthread_mutex_lock(&account_mutex);
    if (account_balance >= amount) {
        int temp = account_balance;
        temp -= amount;
        account_balance = temp;

    }
    pthread_mutex_unlock(&account_mutex);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[2];
    pthread_mutex_init(&account_mutex, NULL);

    for (int t = 0; t < 2; t++) {
        pthread_create(&threads[t], NULL, perform_transaction, (void *)t);
    }

    for (int t = 0; t < 2; t++) {
        pthread_join(threads[t], NULL);
    }


    pthread_mutex_destroy(&account_mutex);
    printf("Final account balance: %d\n", account_balance);

    return 0;
}
```

We changed our code to be thread safe by locking our critical section to ensure only one thread is working on it at a time

Lab 3

Lab3

In Lab 3, we will be attempting to use mutexes to make hash table **insertions** thread safe

Serial implementation is already given, so the only thing you need to modify are:

1. `hash-table-v1.c`: only one mutex, and cares about correctness
2. `hash-table-v2.c`: as many mutexes you want, but cares about correctness and performance
3. `README.md`: report (weighed more heavily this time; 30%)

Mutexes

```
void hash_table_v2_add_entry(struct hash_table_v1 *hash_table,const char *key, uint32_t value)
{
    struct hash_table_entry *hash_table_entry = get_hash_table_entry(hash_table, key);
    struct list_head *list_head = &hash_table_entry->list_head;
    struct list_entry *list_entry = get_list_entry(hash_table, key, list_head);

    /* Update the value if it already exists */
    if (list_entry != NULL) {
        list_entry->value = value;
        return;
    }
    list_entry = calloc(1, sizeof(struct list_entry));
    list_entry->key = key;
    list_entry->value = value;
    SLIST_INSERT_HEAD(list_head, list_entry, pointers);
}
```

Locating the head we will be inserting entry into

Within LL, get the node if already exists, else return NULL

Creating new node for a new entry

How can inserting into our hash table result in race conditions?

Mutex API (all under pthread library)

Declaring and initializing a mutex

```
static pthread_mutex_t foo_mutex;

pthread_mutex_init(&foo_mutex, NULL);
```

Destroying a mutex once finished to prevent memory leaks

```
pthread_mutex_destroy(&foo_mutex);
```

Locking the mutex

```
pthread_mutex_lock(&foo_mutex);
```

Unlocking the mutex

```
pthread_mutex_unlock(&foo_mutex);
```

v1 vs v2

How do these two versions differ?

- V1 we are only using one mutex, while v2 we are using as many as we want
  - With one mutex, what do we lock?
- We want to see a gain in performance, so we do not want to be locking unnecessarily
  - Play around and think about which lines really need to be locked, and what else can be left unlocked

References

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
https://www.eecs.umich.edu/courses/eecs380/ALG/niemann/s_fig31.gif
https://khalilstemmler.com/img/blog/data-structures/hash-tables/hash-table.png
https://media.geeksforgeeks.org/wp-content/uploads/20210108180437/Chaining2.jpg
Past TA slides (Can Aygun)