Operating System (OS)
CS232

Concurrency: Lock-based Concurrent Data Structures

Dr. Muhammad Mobeen Movania
Dr Muhammad Saeed

Outlines

- What is thread safety of a data structure?
- Concurrent Counter
- Issues and resolution of Concurrent Counter
- Concurrent Linked Lists
- Concurrent Queues
- Concurrent Hash Tables
- Summary

What is thread safety?

- Adding locks to a data structure to make it usable by multiple threads concurrently makes the data structure **thread safe**
- How locks are added determines correctness and performance of the data structure
- Many different data structures exist, simplest is a concurrent counter

Concurrent Counter

- Simple counter without locks is not thread safe

```c
typedef struct __counter_t {
    int value;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
}

void increment(counter_t *c) {
    c->value++;
}

void decrement(counter_t *c) {
    c->value--;
}

int get(counter_t *c) {
    return c->value;
}
```

Figure 29.1: **A Counter Without Locks**

Concurrent Counter (2)

- Counter with lock is thread safe

```
typedef struct __counter_t {
    int             value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

Figure 29.2: **A Counter With Locks**

Concurrent Counter (3)

- Performance of concurrent counter scales poorly
  – Locks add additional overhead which reduces performance
- Solution
  – Use approximate counters which update counter value after some threshold S so the lock is accessed S times

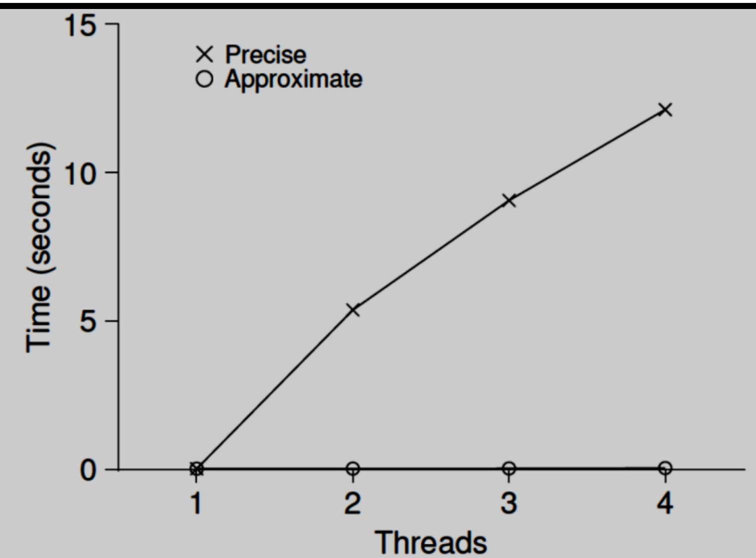

Figure 29.5: **Performance of Traditional vs. Approximate Counters**

Approximate Counter

- Performance of approximate counter with varying values of S

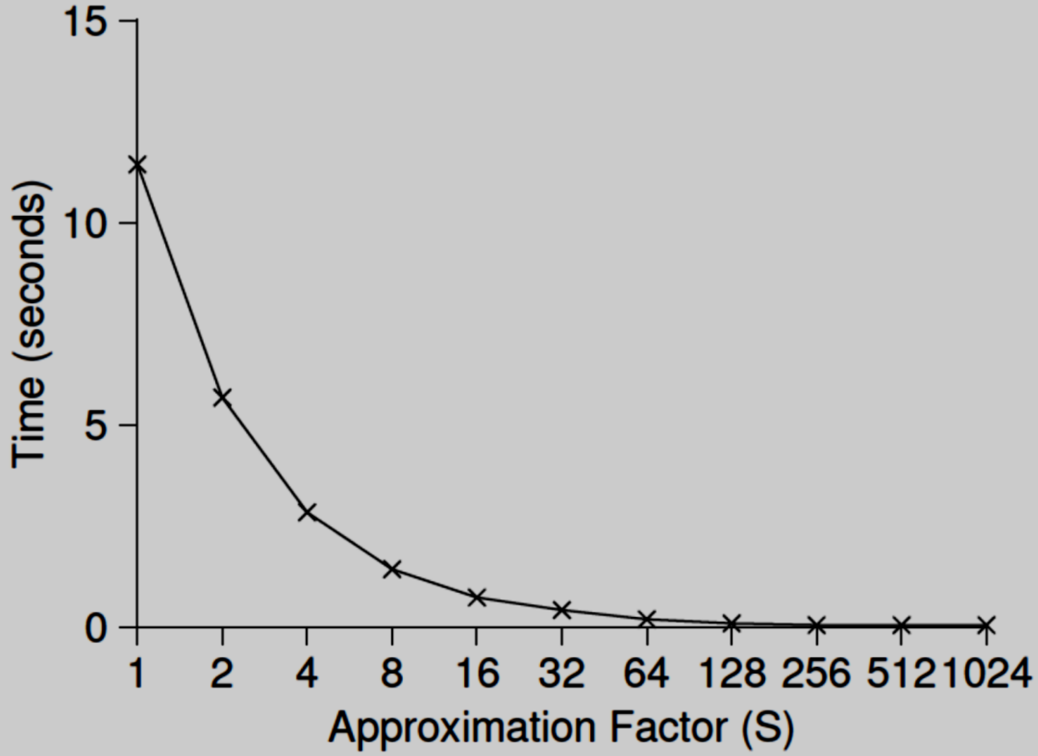

Figure 29.6: **Scaling Approximate Counters**

Concurrent Linked List

- Per-list lock which is acquired when needed

```
// basic node structure
typedef struct __node_t {
    int                 key;
    struct __node_t         *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
    node_t                *head;
    pthread_mutex_t     lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}
```

Concurrent Linked List (2)

- Insert Function: Malloc call could be moved out of lock to avoid branching

```
int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; // fail
    }
    new->key  = key;
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}
```

```
void List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
}
```

Concurrent Queues

- Implemented using two locks: head and tail locks (to provide concurrency in enqueue and dequeuer operations)

```
typedef struct __node_t {
    int                  value;
    struct __node_t     *next;
} node_t;

typedef struct __queue_t {
    node_t               *head;
    node_t               *tail;
    pthread_mutex_t      head_lock, tail_lock;
}
```

Concurrent Queues (2)

- Enqueue and Dequeue operations

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    node_t *tmp = q->head;
    node_t *new_head = tmp->next;
    if (new_head == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return -1; // queue was empty
    }
    *value = new_head->value;
    q->head = new_head;
    pthread_mutex_unlock(&q->head_lock);
    free(tmp);
    return 0;
}
```

Concurrent Hash Table

- Made using concurrent linked list
- Uses one lock per hash bucket each represented by a list

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}

int Hash_Lookup(hash_t *H, int key) {
    return List_Lookup(&H->lists[key % BUCKETS], key);
}
```

Figure 29.10: **A Concurrent Hash Table**

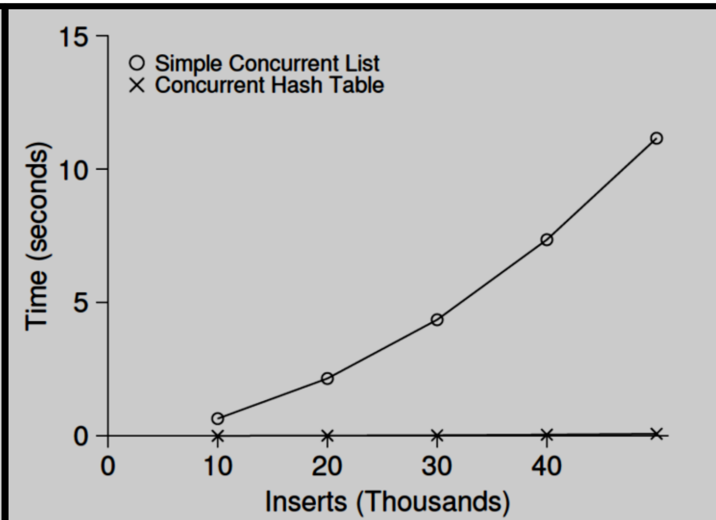

Figure 29.11: **Scaling Hash Tables**

Summary

- We introduced a sampling of concurrent data structures, from counters, to lists and queues, and finally to the ubiquitous and heavily used hash table
- We must be careful with acquisition/release of locks around conditional control flow
- Enabling more concurrency does not necessarily increase performance