Lock-based Concurrent Data Structures

CMFU 334 – Operating Systems
Jason Waterman
Lock-based Concurrent Data structures

• Adding threads is a good way to parallelize your program

• Must be done correctly however
  • Adding locks to a data structure makes the structure **thread safe**
  • How locks are added determine both the **correctness** and **performance** of the data structure
  • Adding threads may actually slow down your code

3/19/2019
Example: Concurrent Counters without Locks

• Simple but not scalable (single threaded)

```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;
}
```
Example: Concurrent Counters with Locks

- Add a **single lock**
  - The lock is acquired when calling a routine that manipulates the data structure

```c
typedef struct __counter_t {
    int value;
    pthread_lock_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;
}
```
The performance costs of the simple approach

- Each thread updates a single shared counter
  - Each thread updates the counter one million times
  - iMac with four Intel 2.7GHz i5 CPUs

Performance of Traditional vs. Sloppy Counters
(Threshold of Sloppy, $S$, is set to 1024)

Synchronized counter scales poorly
Perfect Scaling

• Example: four threads running on four cores

• Even though more work is done, it is **done in parallel**

• The time taken to complete the task on each core is **not increased**
Sloppy counter

• The sloppy counter works by representing:
  • A single logical counter via numerous local physical counters, one per CPU core
  • A single global counter
  • There are multiple locks
    • One for each local counter and one for the global counter

• Example: on a machine with four CPUs
  • Four local counters
  • One global counter
The basic idea of sloppy counting

• When a thread running on a core wishes to increment the counter
  • It increments its local counter
  • Each CPU has its own local counter
    • Threads across CPUs can update local counters without contention
    • Therefore counter updates are scalable
  • The local values are periodically transferred to the global counter
    • Acquire the global lock
    • Increment it by the local counter’s value
    • The local counter is then reset to zero
Sloppiness Threshold

• How often the local-to-global transfer occurs is determined by a threshold, \( S \) (sloppiness)
  • The smaller \( S \):
    • The more the counter behaves like the non-scalable counter
  • The bigger \( S \):
    • The more scalable the counter
    • The further off the global value might be from the actual count
      • Worst case: \( S \times \text{NUMCPUS} \)
Sloppy counter example

- Tracing the Sloppy Counters
  - The threshold $S$ is set to 5
  - There are threads on each of four CPUs
  - Each thread updates their local counters $L_1 \ldots L_4$

<table>
<thead>
<tr>
<th>Time</th>
<th>$L_1$</th>
<th>$L_2$</th>
<th>$L_3$</th>
<th>$L_4$</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5 → 0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5 (from $L_1$)</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5 → 0</td>
<td>10 (from $L_4$)</td>
</tr>
</tbody>
</table>
Importance of the threshold value $S$

- Each four threads increments a counter 1 million times on four CPUs
  - Low $S \rightarrow$ Performance is poor, the global count is always quite accurate
  - High $S \rightarrow$ Performance is excellent, the global count lags
typedef struct __counter_t {
    int global; // global count
    pthread_mutex_t glock; // global lock
    int local[NUMCPUS]; // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS]; // and locks
    int threshold; // update frequency
} counter_t;

// init: record threshold, init locks, init
// values of all local counts and global count
void init(counter_t *c, int threshold) {
    c->threshold = threshold;

    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);

    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

// update: usually, just grab local lock and update
// local amount once local count has risen by
// ’threshold’, grab global lock and transfer local
// values to it
void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt; // assumes amt > 0
    // transfer to global
    if (c->local[cpu] >= c->threshold) {
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}

// get: just return global amount
// (which may not be perfect)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
Concurrent Linked Lists

• The code **acquires** a lock in the insert routine upon entry
• The code **releases** the lock upon exit
  • If `malloc()` happens to *fail*, the code must also **release the lock** before failing the insert
  • This kind of exceptional control flow has been shown to be **quite error prone**
Concurrent Linked Lists

// 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);  
}

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  
}

int List_Lookup(list_t *L, int key) {  
  pthread_mutex_lock(&L->lock);  
  node_t *curr = L->head;  
  while (curr) {  
    if (curr->key == key) {  
      pthread_mutex_unlock(&L->lock);  
      return 0; // success  
    }  
    curr = curr->next;  
  }  
  pthread_mutex_unlock(&L->lock);  
  return -1; // failure  
}
Concurrent Linked Lists

• The code **acquires** a lock in the insert routine upon entry
• The code **releases** the lock upon exit
  • If `malloc()` happens to *fail*, the code must also **release the lock** before failing the *insert*
  • This kind of exceptional control flow has been shown to be **quite error prone**
• **Solution:** The lock and release *only surround* the actual critical section in the insert code
Concurrent Linked List: Rewritten

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

void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
}

int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```
Scaling Linked List

• Hand-over-hand locking (lock coupling)
  • Add a lock per node of the list instead of having a single lock for the entire list
  • When traversing the list:
    • First grabs the next node’s lock
    • And then releases the current node’s lock

• Enable a high degree of concurrency in list operations
  • However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive
Michael and Scott Concurrent Queues

• There are two locks
  • One for the head of the queue
  • One for the tail
  • The goal of these two locks is to enable concurrency of enqueue and dequeue operations

• Add a dummy node
  • Allocated in the queue initialization code
  • Enable the separation of head and tail operations
Concurrent Queues (Cont.)

```c
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 headLock;
    pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next = NULL;
    pthread_mutex_lock(&q->headLock);
    q->head = q->tail = tmp;
    pthread_mutex_unlock(&q->headLock);
}

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->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

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

- Focus on a simple hash table
  - The hash table does not resize
  - Built using the concurrent lists
  - It uses a **lock per hash bucket** each of which is represented by a list

```c
#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) {
  int bucket = key % BUCKETS;
  return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
  int bucket = key % BUCKETS;
  return List_Lookup(&H->lists[bucket], key);
}
```
Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates from each of four threads
  - iMac with four Intel 2.7GHz i5 CPUs

![Graph showing the performance of concurrent hash table](image)
Summary

• We looked at a few of the concurrent data structures out there
  • Counters
  • Lists
  • Queues
  • Hash Tables

• Tips
  • Be careful with acquiring and releasing locks around control flow changes
  • Enabling more concurrency does not necessarily increase performance
  • Premature optimization is the root of all evil!