Thread-Safe Data Structures

Crux: How To Add Locks To Data Structures

When given a particular data structure, how should we add locks to it, in order to make it work correctly? Further, how do we add locks such that the data structure yields high performance, enabling many threads to access the structure at once, i.e., **concurrently**?

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf

Thread-Safe Counters

not thread safe

```
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;
}
```

thread safe, using locks

```
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;
}
```

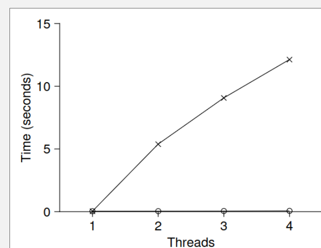

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf

Performance

- sources of performance impacts
  - overhead of acquiring and releasing locks
  - loss of concurrency due to mutual exclusion

ideally, *n* threads running simultaneously on *n* CPUs and each performing a task shouldn't take any longer than a single thread on a single CPU performing the same task

problem: scalability on multicore machines is really important and the simple solution is often not efficient enough



https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf

Making Counting Scalable

- the bottleneck is the update of a single shared counter
  - threads have to wait on each other in order to get access
- approximate counting
  - threads update a local counter (one per CPU) and local counts are periodically transferred to a shared global counter
    - there is little competition for local locks because only threads on a single CPU are trying to update that counter
    - all threads compete for the global lock but the reduced frequency of global updates means reduced likelihood of having to wait when updating the global counter
  - the less frequent the global updates, the more scalable – but the global counter also lags more behind the local counts

Approximate Counting

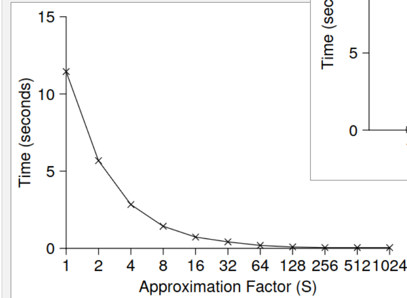
```
typedef struct __counter_t {
    int             global;          // global count
    pthread_mutex_t glock;           // global lock
    int             local[NUMCPUS]; // per-CPU count
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int             threshold;       // update freq
} counter_t;

// 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 it has risen '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;
    if (c->local[cpu] >= c->threshold) {
        // transfer to global (assumes amt>0)
        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 (approximate)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```


Performance

approximate counting scales well (threshold 1024)

Precise
Approximate
Time (seconds)
15
10
5
0
1
2
3
4
Threads



tradeoff between update threshold and performance

Concurrent Linked Lists

other operations (such as delete) are similar

```
// 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
}
```

another challenge: multiple exit paths complicate ensuring that the lock is always released before returning

many bugs result from error handling and similar control flow complications – try to avoid!


```
int List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return -1;
    }
    new->key = key;
    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

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
}
```

- reducing exit paths

`malloc` should be thread-safe already – only the actual relinking of nodes needs to be protected

structure for a single return path

Scaling Linked Lists

- hand-over-hand locking (lock coupling)
  - have a lock for each node instead of a single lock for the whole list
  - when traversing the list, acquire the lock for the next node before releasing the lock for the current node
- advantages
  - allows concurrent access to different parts of the list
- drawbacks
  - adds a lot of overhead for list traversal – must acquire and release *n* locks instead of one

Concurrent Queues

```
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;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

```
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;
}
```

queues are only modified at the ends (enqueue at the tail, dequeue at the head) → only need head and tail locks

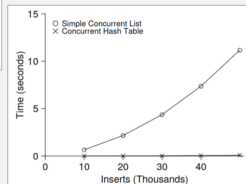
dummy node is used to keep head and tail separate

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf

Concurrent Hashtables

```
#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);
}
```

performance scales well – only access the same list when multiple things hash to the same slot at the same time



uses separate chaining – each slot has a linked list of the elements hashing to that slot

uses the simple one-lock-per-list linked list – only a critical section when accesses involve the same slot

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf

Lessons for Concurrency

Tip: Be Wary Of Locks And Control Flow

A general design tip, which is useful in concurrent code as well as elsewhere, is to be wary of control flow changes that lead to function returns, exits, or other similar error conditions that halt the execution of a function. Because many functions will begin by acquiring a lock, allocating some memory, or doing other similar stateful operations, when errors arise, the code has to undo all of the state before returning, which is error-prone. Thus, it is best to structure code to minimize this pattern.

Tip: More Concurrency Isn't Necessarily Faster

If the scheme you design adds a lot of overhead (for example, by acquiring and releasing locks frequently, instead of once), the fact that it is more concurrent may not be important. Simple schemes tend to work well, especially if they use costly routines rarely. Adding more locks and complexity can be your downfall. All of that said, there is one way to really know: build both alternatives (simple but less concurrent, and complex but more concurrent) and measure how they do. In the end, you can't cheat on performance; your idea is either faster, or it isn't.

Tip: Avoid Premature Optimization (Knuth's Law)

When building a concurrent data structure, start with the most basic approach, which is to add a single big lock to provide synchronized access. By doing so, you are likely to build a *correct* lock; if you then find that it suffers from performance problems, you can refine it, thus only making it fast if need be. As **Knuth** famously stated, "Premature optimization is the root of all evil."

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf