23. Locked Data Structures

adding locks to a data structure to make it usable by threads makes the structure **thread safe**

HOW TO ADD LOCKS TO DATA STRUCTURES?

Concurrent Counter

Simply adds a single lock,

- is acquired when calling a routine that manipulates the data structure
- is released when returning from the call

```
1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: **A Counter With Locks**

Performance can be a problem (we add overheads by adding a lock)

perfect scaling

threads complete just as quickly on multiple processors as the single thread does on one.

- even though more work is done, it is done in parallel
  => hence the time taken to complete the task is not increased

Without scalable counting, some workloads running on Linux suffer from serious scalability problems on multicore machines

=> **approximate counter**

represent a single logical counter

via numerous local physical counters, one per CPU core,

as well as a single global counter.

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.3: **Tracing the Approximate Counters**

threshold (S) :

- the smaller : counter behaves like the non-scalable counter
- the bigger : more scalable the counter / further off the global value from actual count. performance / accuracy trade off

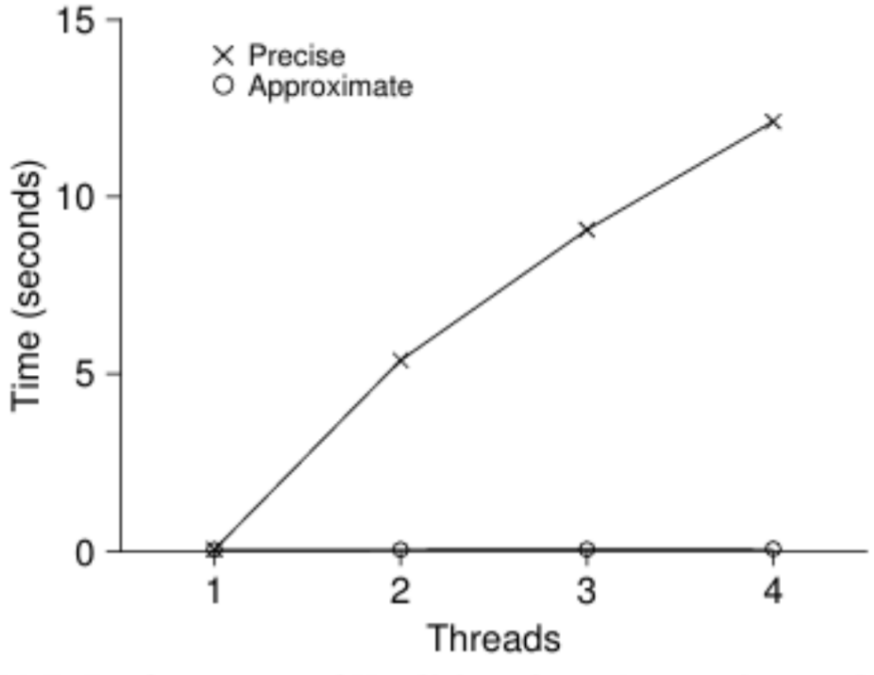

Figure 29.5: **Performance of Traditional vs. Approximate Counters**

```
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!
}
```

Figure 29.4: **Approximate Counter Implementation**

Concurrent Linked List

the lock and relase only surround the actual critical section.

```
1  void List_Init(list_t *L) {
2      L->head = NULL;
3      pthread_mutex_init(&L->lock, NULL);
4  }
5
6  int List_Insert(list_t *L, int key) {
7      // synchronization not needed
8      node_t *new = malloc(sizeof(node_t));
9      if (new == NULL) {
10         perror("malloc");
11         return -1;
12     }
13     new->key = key;
14     // just lock critical section
15     pthread_mutex_lock(&L->lock);
16     new->next = L->head;
17     L->head   = new;
18     pthread_mutex_unlock(&L->lock);
19     return 0; // success
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
```

Figure 29.8: **Concurrent Linked List: Rewritten**

: it does not scale particularly well.

=> **Hand-over-hand locking (lock coupling)**
instead of having a single lock for the entire list => add a lock per node when traversing the list,

the code first grabs the next node's lock and then releases the current node's lock (which inspires the name hand-over-hand)

- enables a high degree of concurrency in list operations
- it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquring and releasing locks for each node of a list traversal is prohibitive. (Even with very large lists)

=> a standard method to make a concurrent data structure : add a big lock

Be Wary of Locks and Control Flow : minimize complexity

as less code as you can in stateful operations (such as lock required)

More Concurrency isn't Necessarily Faster

if the scheme you design adds a lot of overhead, 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.
=> 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.

Concurrent Queue

```
1  typedef struct __node_t {
2      int                 value;
3      struct __node_t     *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t              *head;
8      node_t              *tail;
9      pthread_mutex_t     head_lock, tail_lock;
10 } queue_t;
11
12 void Queue_Init(queue_t *q) {
13     node_t *tmp = malloc(sizeof(node_t));
14     tmp->next = NULL;
15     q->head = q->tail = tmp;
16     pthread_mutex_init(&q->head_lock, NULL);
17     pthread_mutex_init(&q->tail_lock, NULL);
18 }
19
20 void Queue_Enqueue(queue_t *q, int value) {
21     node_t *tmp = malloc(sizeof(node_t));
22     assert(tmp != NULL);
23     tmp->value = value;
24     tmp->next  = NULL;
25
26     pthread_mutex_lock(&q->tail_lock);
27     q->tail->next = tmp;
28     q->tail = tmp;
29     pthread_mutex_unlock(&q->tail_lock);
30 }
31
32 int Queue_Dequeue(queue_t *q, int *value) {
33     pthread_mutex_lock(&q->head_lock);
34     node_t *tmp = q->head;
35     node_t *new_head = tmp->next;
36     if (new_head == NULL) {
37         pthread_mutex_unlock(&q->head_lock);
38         return -1; // queue was empty
39     }
40     *value = new_head->value;
41     q->head = new_head;
42     pthread_mutex_unlock(&q->head_lock);
43     free(tmp);
44     return 0;
45 }
```

Figure 29.9: **Michael and Scott Concurrent Queue**

head works as just a pointer not valid value container.

Concurrent Hash Table

uses a lock per hash bucket (each of which is represented by a list)
Doing so enables many concurrent operations to take place

```
1  #define BUCKETS (101)
2
3  typedef struct __hash_t {
4      list_t lists[BUCKETS];
5  } hash_t;
6
7  void Hash_Init(hash_t *H) {
8      int i;
9      for (i = 0; i < BUCKETS; i++)
10         List_Init(&H->lists[i]);
11 }
12
13 int Hash_Insert(hash_t *H, int key) {
14     return List_Insert(&H->lists[key % BUCKETS], key);
15 }
16
17 int Hash_Lookup(hash_t *H, int key) {
18     return List_Lookup(&H->lists[key % BUCKETS], key);
19 }
```

Figure 29.10: **A Concurrent Hash Table**

the performance of per-bucket locked concurrent hash table is better than a linked list with a single lock
Q. is it about granuality of locking? : NO.
Even with the same level of locking in linked list (per node), performance get worse than single lock linked list.

the point is consider the difference of data structure characteristic.

- hash table : index-like, direct access => better with per bucket lock
- linked list : non direct access (traverse) => better with single lock
  - in per-node lock, it needs to access multiple locks anyway to traverse

Summary

be careful with acquisition and release of locks around control flow changes; **that enabling more concurrency does not necessarily increase performance**

there is no value in making something faster if doing so will not improve the overall performance of the application.

Avoid Premature Optimization

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.**

"Premature optimization is the root of all evil"