【OS】Day35

Class	Operating System: Three Easy Pieces
Date	@February 15, 2022

【Ch29】Locked-based Concurrent Data Structures

29.1 Concurrent Counters

One of the simplest data structures is a counter. It is a structure that is commonly used and has a simple interface.

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: **A Counter Without Locks**

Simple But Not Scalable

How can we make this code thread safe?

```
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**

We simply add a single lock, which is acquired when calling a routine that manipulates the data structure, and is released when returning from the call.

The following diagram shows its performance:

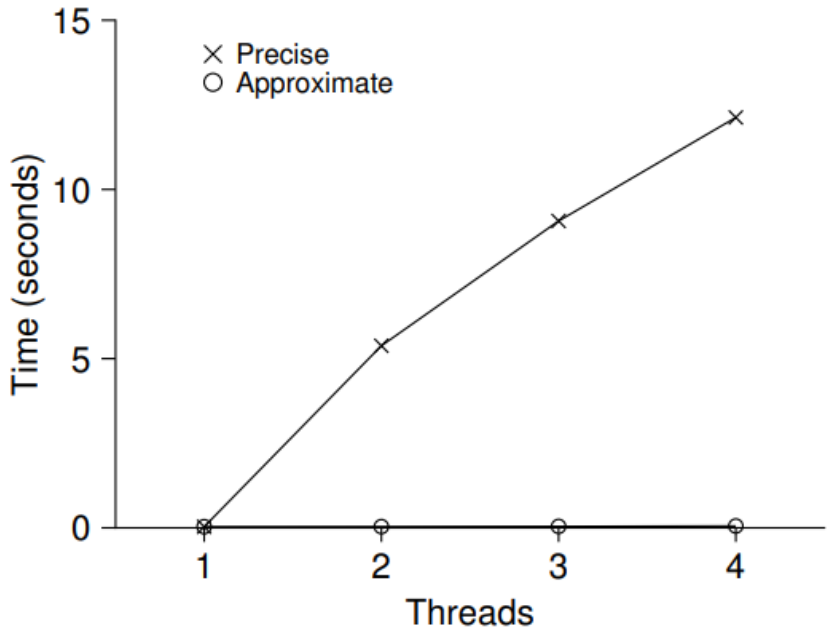

Figure 29.5: **Performance of Traditional vs. Approximate Counters**

Ideally, we would like to see the threads complete just as quickly on multiple processors as the single thread does on one. Achieving this end is called perfect scaling; even though more work is done, it is done in parallel.

Scalable Counting

We attack the problem of scaling is by one approach known as an approximate counter.

The approximate counter works by representing a single logical counter via numerous local physical counters, one per CPU core, as well as a single global counter.

Specifically, on a machine with four CPUs, there are four local counters and one global one. In addition to these counters, there are also locks: one for each local counter, and one for the global counter.

The basic idea of approximate counting is as follows. When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corresponding local lock. Because each CPU has its

own local counter, threads across CPUs can update local counters without contention, and thus updates to the counter are scalable.

However, to keep the global counter up to date(in case a thread wishes to read its value), the local values are periodically transferred to the global counter, by acquiring the global lock and incrementing it by the local counter's value; the local counter is then reset to zero.

How often this local-to-global transfer occurs is determined by a threshold S. The smaller S is, the more the counter behaves like the non-scalable counter above; the bigger S is, the more scalable the counter, but the further off the global value might be from the actual count.

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**

Approximate Counter Implementation

```
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 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 '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**

29.2 Concurrent Linked Lists

Here is a rudimentary implementation of the linked list:

```
1  // basic node structure
2  typedef struct __node_t {
3      int                 key;
4      struct __node_t         *next;
5  } node_t;
6
7  // basic list structure (one used per list)
8  typedef struct __list_t {
9      node_t                *head;
10     pthread_mutex_t     lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key  = key;
27     new->next = L->head;
28     L->head   = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }
```

Figure 29.7: **Concurrent Linked List**

One thing to notice: If `malloc()` fails when allocating a new node fails, the code must also release the lock before failing the insert.

In fact, this path is quite error prone.

We can rearrange the code a bit so that the lock and release only surround the actual critical section in the insert code, and that a common exit path is used in the lookup code. The former works because part of the insert actually need not be locked; assuming that `malloc()` itself is thread-safe, each thread can call into it without worry of race conditions or other concurrency bugs. Only when updating the shared list does a lock need to be held.

The following code fixes this issue:

```
1  void List_Init(list_t *L) {
2      L->head = NULL;
3      pthread_mutex_init(&L->lock, NULL);
4  }
5
6  void 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;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head   = new;
19     pthread_mutex_unlock(&L->lock);
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**