Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T20

Lock-Based Concurrent Data Structures

Referência principal
Ch. 29 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 08 de outubro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Tonight's challenge

How to add locks to a data structure to enable many concurrent threads to access it correctly?

Can we do it without sacrificing performance?

Implementing a counter

- This is a simple implementation of a counter (`ch29-01.c`).
- Is it ***thread safe***, that is, can it be accessed concurrently by a number of threads?
  - No, there is a race condition.
- However, it can be made safe with the help of a lock, as shown in the next slide.

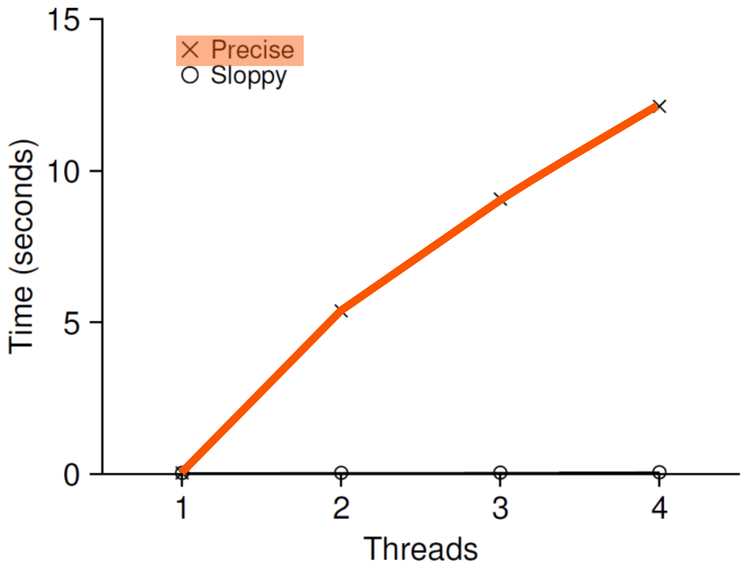
```c
typedef struct counter_t {
    int value;
} counter_t;

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;
}
```

A thread safe concurrent counter

```
typedef struct counter_t {
    int value;
    Mutex_t lock;
} counter_t;

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

int get(counter_t *c) {
    Mutex_lock(&c->lock);
    int rc = c->value;
    Mutex_unlock(&c->lock);
    return rc;
}

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

void decrement(counter_t *c) {
    Mutex_lock(&c->lock);
    c->value–;
    Mutex_unlock(&c->lock);
}
```

Performance of the concurrent counter

- It has been shown that our concurrent counter does not scale well.



- The chart shows the time taken by 1 to 4 threads to update the counter 1 million times on a machine with 4 CPUs.

- Ideally, we would like to get ***perfect scaling***, i.e. the threads should complete just as quickly on multiple processors as a single thread does on one.

A sloppy counter

- There have been many attempts at achieving a scalable counter.
- An interesting one is what is called a ***sloppy counter***.
  - There are n threads running on m CPUs (or cores).
  - There are m local counters (one per CPU) and just one global counter.
  - Each thread updates the local counter of the CPU it is running on.
  - Every time the local counter reaches a threshold value S, it is added to the global counter and zeroed again.
  - So, at any time the global counter has an approximation of the actual total of the local counters.
    - The smaller S is, the more the counter behaves like the previous non-scalable counter.
    - The bigger S is, the more scalable the sloppy counter, but the further off the global value might be from the actual count.

Sloppy counter (ch29-05.c)

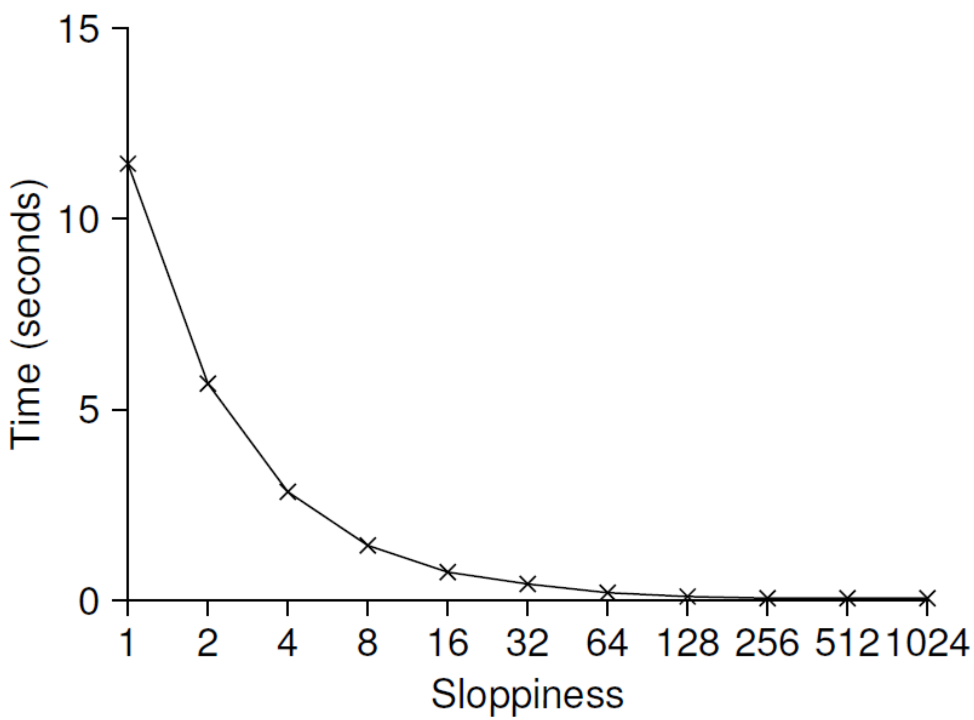
```c
typedef struct counter_t {
    int global;              // global count
    Mutex_t glock;           // global lock
    int local[NUM_CPUS];     // local count (per cpu)
    Mutex_t llock[NUM_CPUS]; // ... and locks
    int threshold;           // update frequency
} counter_t;

// 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;
    Mutex_init(&c->glock);
    for (int i = 0; i < NUM_CPUS; i++) {
        c->local[i] = 0;
        Mutex_init(&c->llock[i]);
    }
}
```

Sloppy counter (ch29-05.c)

```
// 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 % NUM_CPUS;
   Mutex_lock(&c->llock[cpu]);
   c->local[cpu] += amt;              // assumes amt > 0
   if (c->local[cpu] >= c->threshold) { // transfer to global
      Mutex_lock(&c->glock);
      c->global += c->local[cpu];
      Mutex_unlock(&c->glock);
      c->local[cpu] = 0;
   }
   Mutex_unlock(&c->llock[cpu]);
}
```

Sloppy counter (`ch29-05.c`)

```c
// get: just return global amount (which may not be perfect)
int get(counter_t *c) {
    Mutex_lock(&c->glock);
    int val = c->global;
    Mutex_unlock(&c->glock) ;
    return val; // only approximate!
}
```



- The chart shows the increase in scalability of the sloppy counter, as S grows, at the expense of its accuracy.

Concurrent Linked Lists

- We will omit most of the usual linked list routines and just focus on concurrent insert.
- As you can see in the next slide, the insert code simply acquires a lock at the beginning and releases it at the end of the routine.
  - Since the call to `malloc` may fail, the code must release the lock before raising the exception.
- This sort of *ad hoc* control flow is a major source of bugs.
  - A study of the Linux kernel patches has shown that 40% of the bugs occur on rarely-taken paths.

Concurrent Linked List (ch29-07.c)

```
// 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;
    Mutex_t lock;
} list_t;

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

Concurrent Linked List (ch29-07.c)

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

Concurrent Linked List (ch29-07.c)

```
int List_Lookup(list_t *L, int key) {
    Mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            Mutex_unlock(&L->lock)
            return 0; // success
        }
        curr = curr->next;
    }
    Mutex_unlock(&L->lock);
    return -1; // failure
}
```

Warning and new challenge

It is difficult to ensure that unbalanced lock-unlock calls match correctly.

Could we rewrite the insert and lookup routines so they remain correct under concurrent access but avoid the odd call to unlock on the failure path?

Concurrent Linked List with Paired Lock-Unlock (ch29-08.c)

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

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
   Mutex_lock(&L->lock);
   new->next = L->head;
   L->head = new;
   Mutex_unlock(&L->lock);
}

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

Scaling linked lists

- Even the rewritten linked list algorithm does not scale well due to the existence of a single lock to take care of the whole list.
- One of the techniques used to enable greater concurrency within a list is called ***hand-over-hand locking*** or ***lock coupling***.
  - The idea is to create a lock for each node of a list, instead of a single one.
  - When traversing a list, the code first acquires the lock for the next node before releasing the lock on the current one.
  - Although this method enables a high degree of concurrency, the overhead of acquiring and releasing locks at each step may be prohibitive.

Concurrent Queues

- To increase concurrency of enqueue and dequeue operations, we will use two locks: one for the head of the queue, one for the tail.
  - Normally, enqueue will access the tail lock and dequeue the head lock.
  - A clever trick of the creators of this design was to add a dummy node, to separate head and tail operations.
- Queues are often used in multi-threaded applications, whose needs will not be met by the type of queue used here.
  - A more elaborate bounded queue will be developed in the next class.

A concurrent queue using two locks (`ch29-09.c`)

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

typedef struct queue_t {
     node_t *head;
     node_t *tail;
     Mutex_t headLock;
     Mutex_t tailLock;
} queue_t;

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

A concurrent queue using two locks (`ch29-09.c`)

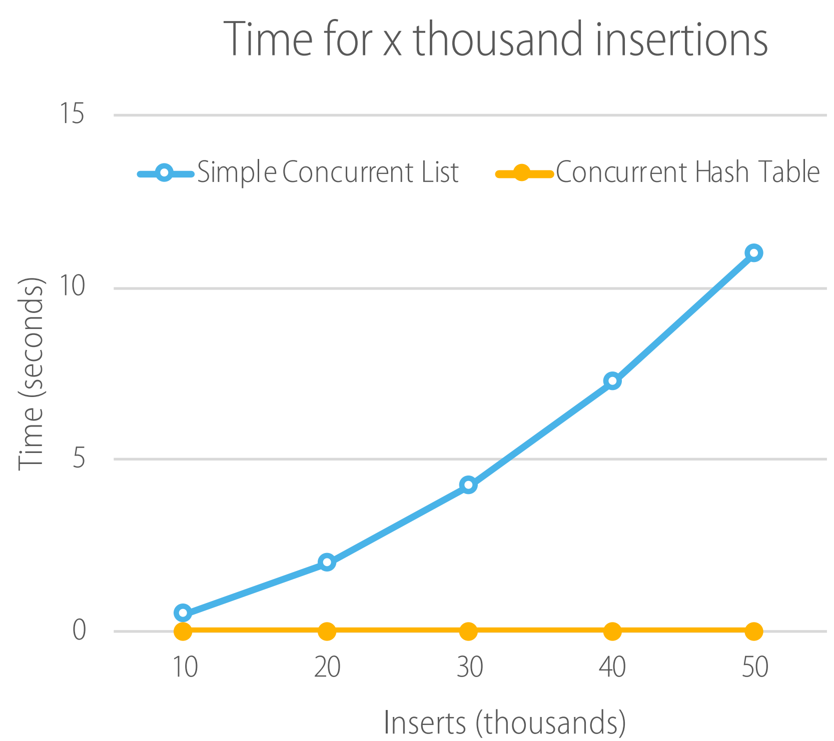
```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = Malloc(sizeof(node_t));
    tmp->value = value;
    tmp->next = NULL;
    Mutex_lock(&q->tailLock) ;
    q->tail->next = tmp;
    q->tail = tmp;
    Mutex_unlock(&q->tailLock) ;
}
```

A concurrent queue using two locks (`ch29-09.c`)

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

Concurrent Hash Table

- We will create a simple hash table that does not resize.
- The hash table will be built using our previous concurrent list model and works pretty well.
  - The main reason for its good performance is that, instead of having a single lock for the entire structure, it uses one lock per hash bucket.
  - Thus, hash buckets, which will be represented by lists, can be accessed concurrently.



A Concurrent Hash Table (`ch29-10.c`)

```c
#define BUCKETS (101)

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

void Hash_Init(hash_t *H) {
   for (int 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);
}
```

Three Tips

More concurrency isn't necessarily faster.

Beware of locks and control flow.

Avoid premature optimization.