29. Lock-based Concurrent Data Structures

Operating System: Three Easy Pieces

Lock-based Concurrent Data structure

- 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.

- Just a succinct introduction to the multithreaded way-of-"thinking"
  - Thousands of research papers about it

Example: Concurrent Counters without Locks

- Not thread safe

```
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      }
```

Example: Concurrent Counters with Locks

- Add a **single lock**: thread safe but scalable?
  - The lock is acquired when calling a routine that manipulates the data structure.

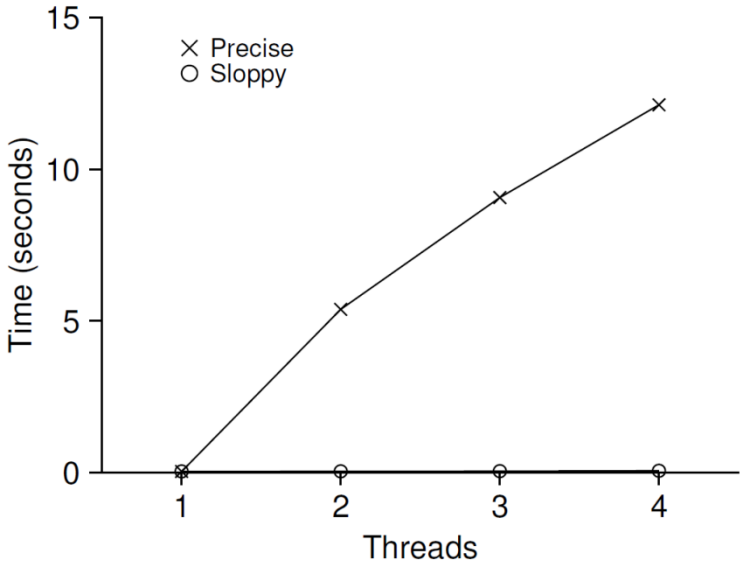
```
1       typedef struct __counter_t {
2               int value;
3               pthread_lock_t lock;
4       } counter_t;
5
6       void init(counter_t *c) {
7               c->value = 0;
8               Pthread_mutex_init(&c->lock, NULL);
9       }
10
11      void increment(counter_t *c) {
12              Pthread_mutex_lock(&c->lock);
13              c->value++;
14              Pthread_mutex_unlock(&c->lock);
15      }
16
```

Example: Concurrent Counters with Locks (Cont.)

```
(Cont.)
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      }
```

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 might scale poorly.

Perfect Scaling

- Even though more work is done, it is **done in parallel**.
- The time taken to complete the task is *not increased*.

Sloppy counter

- The sloppy counter works by representing …
  - A single **logical counter** via numerous local physical counters, on per CPU core
  - A single **global counter**
  - There are **lock**s:
    - One fore 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 increment its local counter.
  - Each CPU has its own local counter:
    - Threads across CPUs can update local counters *without contention*.
    - Thus 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.

The basic idea of sloppy counting (Cont.)

- 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*.

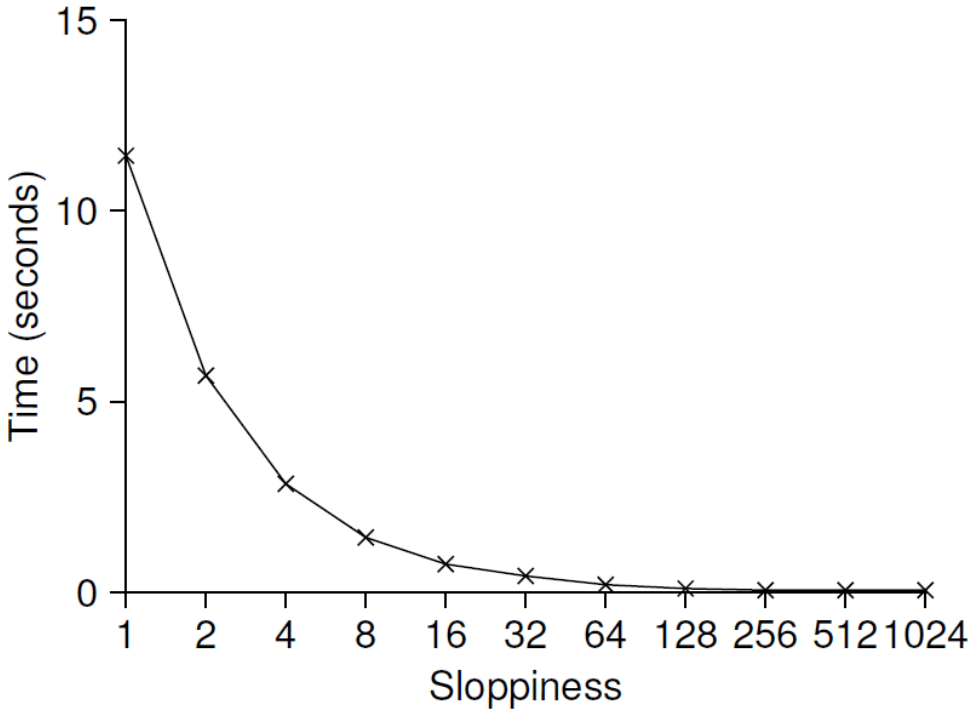
Sloppy counter example

- 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 $L_1 \ldots L_4$.

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 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

Importance of the threshold value S

- Each four threads increments a counter 1 million times on four CPUs.
  - Low S → Performance is **poor**, The global count is always quire **accurate**.
  - High S → Performance is **excellent**, The global count **lags**.



Scaling Sloppy Counters

Sloppy Counter Implementation

```
1       typedef struct __counter_t {
2           int global;             // global count
3           pthread_mutex_t glock;  // global lock
4           int local[NUMCPUS];     // local count (per cpu)
5           pthread_mutex_t llock[NUMCPUS]; // ... and locks
6           int threshold;          // update frequency
7       } counter_t;
8
9       // init: record threshold, init locks, init values
10      //       of all local counts and global count
11      void init(counter_t *c, int threshold) {
12          c->thres hold = threshold;
13
14          c->global = 0;
15          Pthread_mutex_init(&c->glock, NULL);
16
17          int i;
18          for (i = 0; i < NUMCPUS; i++) {
19              c->local[i] = 0;
20              Pthread_mutex_init(&c->llock[i], NULL);
21          }
22      }
23
```

Sloppy Counter Implementation (Cont.)

(Cont.)

```
24      // update: usually, just grab local lock and update local amount
25      //         once local count has risen by 'threshold', grab global
26      //         lock and transfer local values to it
27      void update(counter_t *c, int threadID, int amt) {
28          Pthread_mutex_lock(&c->llock[threadID]);
29          c->local[threadID] += amt;                  // assumes amt > 0
30          if (c->local[threadID] >= c->threshold) { // transfer to global
31              Pthread_mutex_lock(&c->glock);
32              c->global += c->local[threadID];
33              Pthread_mutex_unlock(&c->glock);
34              c->local[threadID] = 0;
35          }
36          Pthread_mutex_unlock(&c->llock[threadID]);
37      }
38
39      // get: just return global amount (which may not be perfect)
40      int get(counter_t *c) {
41          Pthread_mutex_lock(&c->glock);
42          int val = c->global;
43          Pthread_mutex_unlock(&c->glock);
44          return val;    // only approximate!
45      }
```

Concurrent Linked List: 1st try (and how easy is introducing bugs)

```
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
(Cont.)
```

Concurrent Linked Lists

```
(Cont.)
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      }
(Cont.)
```

Concurrent Linked Lists (Cont.)

```
(Cont.)
32
32      int List_Lookup(list_t *L, int key) {
33              Pthread_mutex_lock(&L->lock);
34              node_t *curr = L->head;
35              while (curr) {
36                      if (curr->key == key) {
37                              Pthread_mutex_unlock(&L->lock);
38                              return 0; // success
39                      }
40                      curr = curr->next;
41              }
42              Pthread_mutex_unlock(&L->lock);
43              return -1; // failure
44      }
```

Concurrent Linked Lists (Cont.)

- 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: Refactored Insert

```
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
```

Concurrent Linked List: Refactored(Cont.)

```
(Cont.)
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      }
```

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 can be *prohibitive*.
    - Perhaps an hybrid (where you grab a new lock every so many nodes) approach?

Michael and Scott Concurrent Queues

- Queues uses enqueue/dequeue operations only

- 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.)

```
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 headLock;
10                pthread_mutex_t tailLock;
11        } queue_t;
12
13        void Queue_Init(queue_t *q) {
14                node_t *tmp = malloc(sizeof(node_t));
15                tmp->next = NULL;
16                q->head = q->tail = tmp;
17                Pthread_mutex_init(&q->headLock, NULL);
18                Pthread_mutex_init(&q->tailLock, NULL);
19        }
20
(Cont.)
```

Concurrent Queues (Cont.)

```
(Cont.)
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL && "Malloc failed");
24
25              tmp->value = value;
26              tmp->next = NULL;
27
28              Pthread_mutex_lock(&q->tailLock);
29              q->tail->next = tmp;
30              q->tail = tmp;
31              Pthread_mutex_unlock(&q->tailLock);
32      }
(Cont.)
```

Concurrent Queues (Cont.)

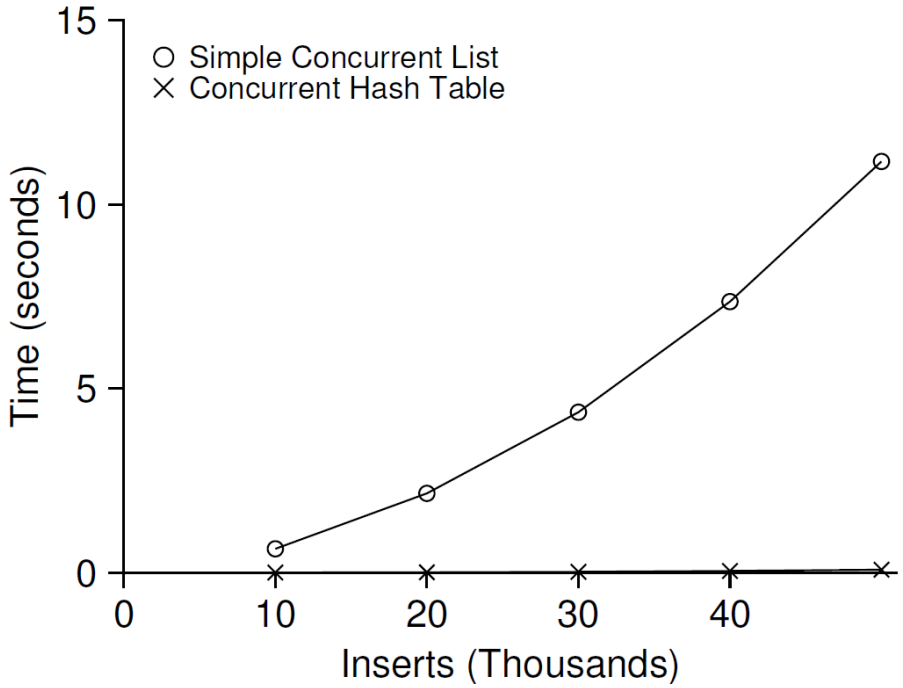
```
(Cont.)
33      int Queue_Dequeue(queue_t *q, int *value) {
34              Pthread_mutex_lock(&q->headLock);
35              node_t *tmp = q->head;
36              node_t *newHead = tmp->next;
37              if (newHead == NULL) {
38                      Pthread_mutex_unlock(&q->headLock);
39                      return -1; // queue was empty
40              }
41              *value = newHead->value;
42              q->head = newHead;
43              Pthread_mutex_unlock(&q->headLock);
44              free(tmp);
45              return 0;
46      }
```

Concurrent Hash Table

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

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.



The simple concurrent hash table scales magnificently.

Concurrent Hash Table

```
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
14      int Hash_Insert(hash_t *H, int key) {
15              int bucket = key % BUCKETS;
16              return List_Insert(&H->lists[bucket], key);
17      }
18
19      int Hash_Lookup(hash_t *H, int key) {
20              int bucket = key % BUCKETS;
21              return List_Lookup(&H->lists[bucket], key);
22      }
```

- Disclaimer: This lecture slide set has been adapted for AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)