Lock-based Concurrent Data Structures

Chapter 29

Previously in CS212…

- We’ve talked about threads and how we can coordinate and control concurrent thread execution with locks

- We discussed some of the metrics that are important to us regarding locking solutions
  - Mutual exclusion
  - Fairness
  - Performance

- Now we’ll look a bit at the interplay between locking and common data structures

Controlled Chaos

- We know that we are mostly at the mercy of the scheduler when it comes to what, when, and for how long a process or its threads may run

- Integrating locks with data structures and operations, can help us make them **thread safe.**
    - Locks provide mutually exclusive access to code that should not be altered concurrently

- What does it look like to make a concurrent data structure and what must be considered?

Concurrent Counters

- Might want to keep track of operation counts, resource availability status, indices into other data structures, etc. while using threads
- Straight forward solution, lock the increment, decrement, and read ops
- Note that the caller doesn't have to worry about the lock (similar concept to a **Monitor**)
- Performance hit!
  - Single Thread: 0.03 seconds
  - Two Threads: 5 seconds

```
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;
}
```

Figure 29.2: A Counter With Locks

Scaling Counting

- Simple solution won't do
  - One option is to approximate it
- Each CPU gets a local counter
  - No concurrency issue there
- Add another counter that is shared globally among all the CPUS
- At a given update value for the local counters add that value to the global counter then set the value back to 0
  - Only need to lock global counter read/write ops

Example with 4 CPUs

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**

Concurrent Linked Lists

- Similarly, to the counter we could just focus on the functions that change the linked list
- Lock at the top of the function, and unlock before we leave
- What happens if the malloc fails?
  - If we forgot the lock, this would be **quite bad**™

```
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
}
```

Concurrent Linked Lists - Fixed

- Similarly, to the counter we could just focus on the functions that change the linked list
- Lock at the top of the function, and unlock before we leave
- What happens if the malloc fails?
  - If we forgot the lock, this would be **quite bad**™
- Let’s fix it…

```
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
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
}
```

Scaling Linked Lists

- Locking the whole list means that no other thread can do concurrent operations (even if it just to read the list)

- We could have a lock for each node

- As we traverse the list, we acquire the next node’s lock and release the previous one
  - **Hand-over-hand locking** or **lock coupling**

- While concurrency goes up, the performance is (roughly) the same as locking the entire list

Scaling Concurrent Queues

- Again, we could just use a big lock around the whole queue

- A better idea is to focus just on two nodes, the head and the tail of the queue
  - Head for dequeue operations
  - Tail for enqueue operations

- We provide a fake starting node so that queue has one node initialized for setting up the queue and the locking

Scaling Concurrent Hash Tables

- RECAP: Hash tables store data using a key that is run through a function to locate the data in the structure
- The example simply uses integer keys and a mod function based on the number of “buckets” to hold data to determine where the values are
- Instead of locking the entire hash table, we can use the concurrent linked list to hold each “bucket” of data when we have hash collisions

Gotchas

- Be careful of control structures and locks
  - Conditional paths, early returns, exits, etc. can make for edge cases where concurrency fails

- Avoid premature optimization
  - Consider the case of the linked list
  - We could implement the hand-over-hand approach which is more complicated
  - However, the performance gains are negligible
  - Hold off until you see a need to improve performance before you try to solve a problem you may not have

Next Time

- We look at condition variables