Chapter 29: Lock-based Concurrent Data Structures

Overview

- **Main Idea**: Adding locks to data structures to make them thread-safe while balancing correctness and performance.
- **Key Concepts**: Scalability, lock granularity, approximate counters, hand-over-hand locking.

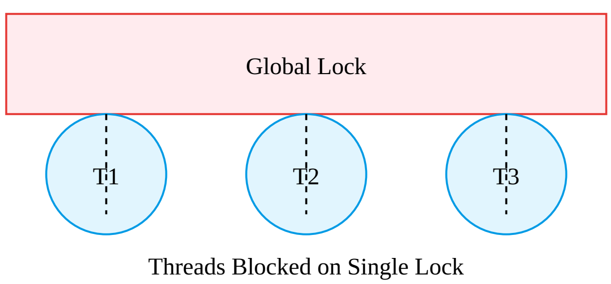
---

1. Concurrent Counters

1.1 Simple Counter with Single Lock

```
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}
```

Problem: Poor scalability due to serialized access.



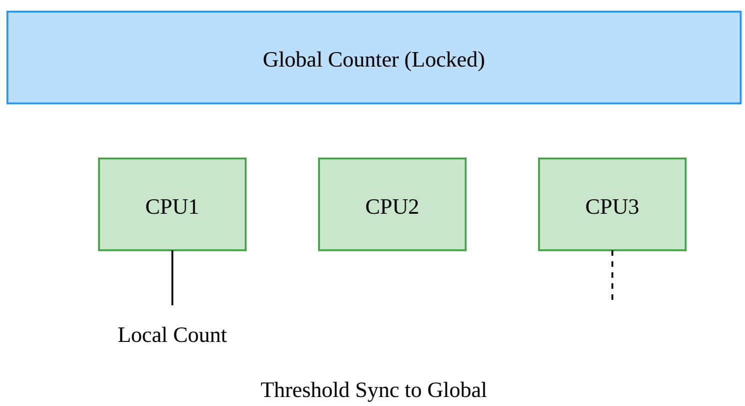
1.2 Approximate Counter (Scalable)

```
typedef struct __counter_t {
    int global;                  // Global count
    pthread_mutex_t glock;       // Global lock
```

```c
    int local[NUMCPUS];             // Per-CPU counts
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int threshold;                  // Update frequency
} counter_t;

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) {
        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]);
}
```

Advantages:

- Low contention for local counters.
- Global counter updated periodically (threshold `S`).

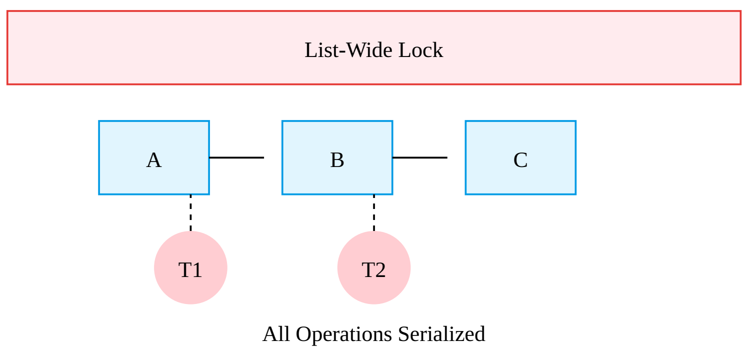


2. Concurrent Linked Lists

2.1 Basic Implementation

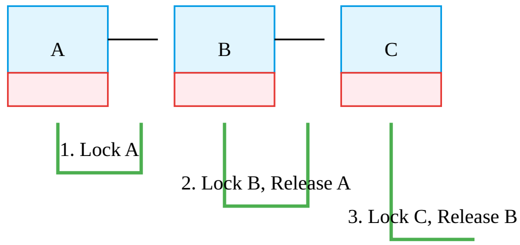
```
typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    new->key = key;
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
}
```

Optimization: Move `malloc()` outside critical section.



2.2 Hand-over-Hand Locking (Advanced)

- **Concept**: Each node has its own lock; traverse while acquiring next node's lock before releasing current.
- **Trade-off**: Higher overhead than single lock due to frequent lock acquisitions.



Hand-over-Hand Traversal

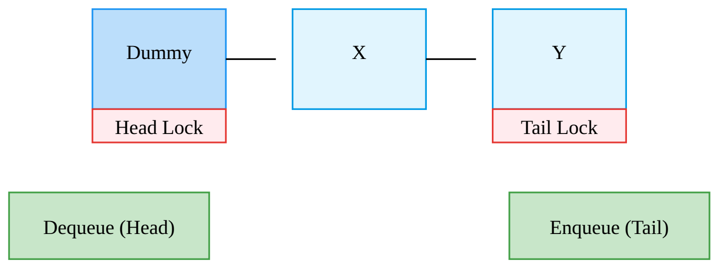
---

3. Concurrent Queues

Michael & Scott Queue

```
typedef struct __queue_t {
    node_t *head, *tail;
    pthread_mutex_t head_lock, tail_lock;
} queue_t;

void Queue_Enqueue(queue_t *q, int value) {
    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = new_node;
    q->tail = new_node;
    pthread_mutex_unlock(&q->tail_lock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    *value = q->head->next->value;
    q->head = q->head->next;
    pthread_mutex_unlock(&q->head_lock);
}
```

Key Idea: Separate locks for head/tail + dummy node for concurrency.



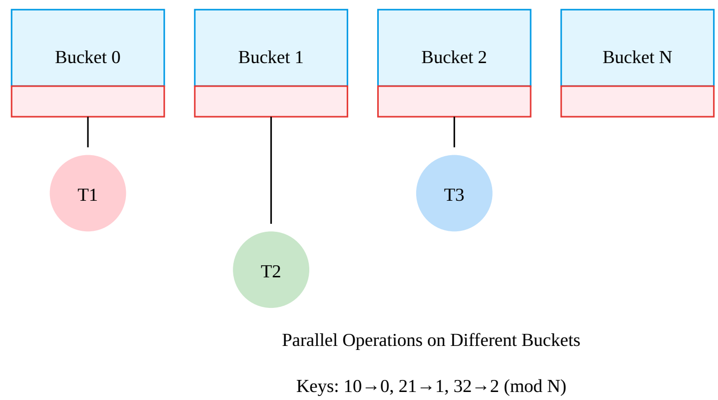
Independent Head/Tail Operations

---

4. Concurrent Hash Tables

```
#define BUCKETS 101
typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}
```

Performance: Scales well due to per-bucket locks.



---

5. Performance Comparison

Data Structure	Locking Strategy	Scalability
Counter	Single lock	Poor (serialized)
Approximate Counter	Per-CPU + global lock	High (threshold-based)
Linked List	Single lock	Moderate
Queue	Head/tail locks	High (enqueue/dequeue)
Hash Table	Per-bucket locks	Excellent

Key Takeaways

1. **Start Simple**: Use coarse-grained locks first, optimize only if needed (Knuth's Law).
2. **Reduce Contention**: Split data structures (e.g., per-bucket locks in hash tables).
3. **Avoid Premature Optimization**: Measure before optimizing (e.g., hand-over-hand locking often underperforms).