

Concurrency: Locked Data Structures

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

Administrivia

- Project 4 Due Tonight, Oct 24th @ 11:59pm
- Project 5 released today, due Nov 7th @ 11:59pm
- Exam 1 grades posted

Review: Locks

- A correctly implemented lock is a way to ensure **mutual exclusion**, should not cause **deadlock** or **starvation**. Locks are judged on correctness, fairness, and performance.
- Locks typically require hardware support:
 - Disable Interrupts
 - `int TestAndSet(int *old_ptr, int new) or xchg()`
 - `int CompareAndSwap(int *addr, int expected, int new)`
 - `int LoadLinked(int *ptr) and int StoreConditional(int *ptr, int value)`
- Different types of locks: Spin-locks, Ticket-locks, and blocking-locks
 - Utilize `yield()` and `park()` and `unpark()`

Quiz: Locks

<https://tinyurl.com/cs537-fa23-q10>



Lock-based Concurrent Data Structures

Creating **thread-safe** (and fast) data structures:

- Counters
- Linked Lists
- Queues
- Hash Tables

Counter Without Locks

```
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) ...
int get(counter_t *c) ...
```

Counter With Locks

```
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) ...
int get(counter_t *c) ...
```

Locked version works correctly, but **doesn't scale**

Approximate Counter

```
typedef struct __counter_t {
    int          global;
    pthread_mutex_t glock;
    int          local[NUMCPUS];
    pthread_mutex_t llock[NUMCPUS];
    int          threshold;
} counter_t;

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);
    }
}
```

Approximate Counter (cont.)

```
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]);
}

int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val;
}
```

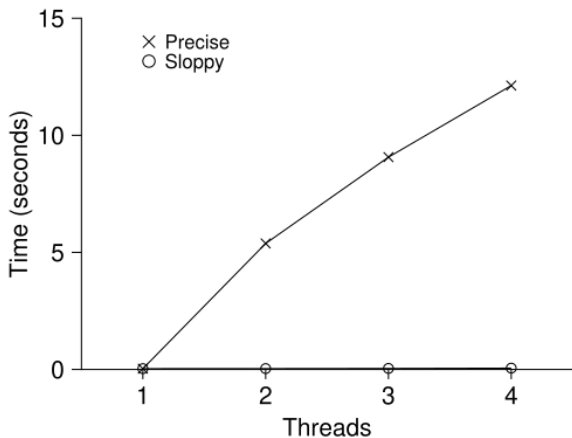

Approximate Counter Trace with Threshold = 5

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)

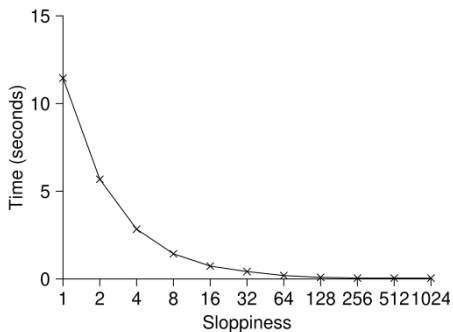
Figure 29.3: Tracing the Approximate Counters

Comparing Exact vs Approx. Counter

4 threads each incrementing counter 1 million times on 4 CPUs



Approximate Counter Scaling



Changing Threshold:

- Low – global count accurate
- High – improved performance

Concurrent Linked Lists

```
typedef struct __node_t {
    int          key;
    struct __node_t *next;
} node;

typedef struct __list_t {
    node          *head;
    pthread_mutex_t lock;
} list;

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

Concurrent Linked Lists (cont.)

```
int List_Insert(list *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *n = malloc(sizeof(node));
    if (n == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; //fail
    }
    n->key = key;
    n->next = L->head;
    L->head = n;
    pthread_mutex_unlock(&L->lock);
    return 0; //success
}
```

```
int List_Lookup(list *L, int key) {
    pthread_mutex_lock(&L->lock);
    node *curr = L->head;
    while(curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; //success
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; //failure
}
```

- One Big Lock – Does Not Scale Well – how else could we lock?
- Exceptional Control Flow – Remember to release lock!

Improved Exceptional Control Flow

```
void List_Insert(list *L, int key) {
    node_t *n = malloc(sizeof(node));
    if (n == NULL) {
        perror("malloc");
        return;
    }
    n->key = key;
    pthread_mutex_lock(&L->lock);
    n->next = L->head;
    L->head = n;
    pthread_mutex_unlock(&L->lock);
    return;
}
```

```
int List_Lookup(list *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node *curr = L->head;
    while(curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv;
}
```

- **Hand-over-hand** locking, having one lock per node. When traversing the list, acquire lock of next node then release lock of current node. In practice, so much locking/unlocking, difficult to make faster than one lock
- Perhaps hybrid approach, one lock for blocks of nodes

Concurrent Queues

- One standard approach – add a big lock around the entire data structure
 - Accurate but not very efficient
- Improved approach – add a lock around the head and another around the tail
 - Allows concurrent enqueue and dequeue operations
- Add a dummy node to separate the head from the tail
- More fully developed bounded queue discussed with condition variables

Concurrent Queue Implementation

```
typedef struct __node_t {
    int          value;
    struct __node_t  *next;
} node;

typedef struct __queue_t {
    node          *head;
    node          *tail;
    pthread_mutex_t  hLock;
    pthread_mutex_t  tLock;
} queue;

void Queue_Init(queue *q) {
    node *tmp = malloc(sizeof(node));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->hLock, NULL);
    pthread_mutex_init(&q->tLock, NULL);
}
```


Concurrent Queue Implementation (cont.)

```
void Enqueue(queue *q, int value) {
    node *tmp = malloc(sizeof(node));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next = NULL;
    pthread_mutex_lock(&q->tLock);
    q->tail->next=tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tLock);
}
```

```
int Dequeue(queue *q, int *value) {
    pthread_mutex_lock(&q->hLock);
    node *tmp = q->head;
    node *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->hLock);
        return -1; //empty queue
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->hLock);
    free(tmp);
    return 0;
}
```

Concurrent Hash Table

- Utilizes a concurrent list for each hash bucket
- Rather than having a single lock, each bucket is locked independently
 - Allows many concurrent operations

Concurrent Hash Table Implementation

```
#define BUCKETS (101)
typedef struct __hash_t {
    list    lists[BUCKET];
} hash_t;

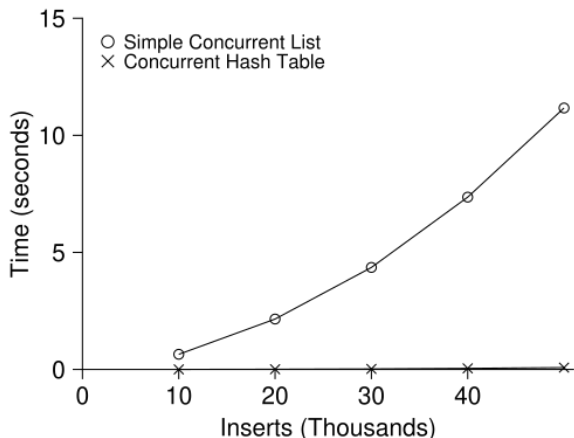
void Hash_Init(hash_t *H) {
    int i=0;
    for(i=0;i<BUCKETS;i++) {
        List_Init(&H->lists[i]);
    }
}

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);
}
```

Hashtable Performance

- Single List doesn't scale, but hashtable does



Summary

- Be careful with acquisition and release of locks
- Enabling more concurrency does not necessarily increase performance
- Performance problems should only be remedied once they exist
 - **Start with single big lock (likely correct)**
 - Only if it doesn't meet needs, refine it

Premature Optimization is the root of all evil. – Donald Knuth