CONCURRENCY: DATA STRUCTURES

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

Spring break!

AGENDA / LEARNING OUTCOMES

Concurrency: How to build concurrent data structures?

Summary of virtualization, concurrency

RECAP

CONCURRENCY OBJECTIVES

Mutual exclusion (e.g., A and B don't run at same time)

solved with *locks*

Ordering (e.g., B runs after A does something)

solved with *condition variables* and *semaphores*

ABSTRACTIONS

Objects, Lists, Hashtable

Semaphores

Locks, Condition variables

Atomic Primitives

CONCURRENT DATA STRUCTURES

CONCURRENT DATA STRUCTURES

Counters

Lists

Hashtable

Queues

Start with a correct solution

Make it perform better!

WHAT IS SCALABILITY

N times as much work on N cores as done on 1 core

Strong scaling	Weak scaling
Fix input size, increase number of cores	Increase input size with number of cores

COUNTERS

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

void init(counter_t *c) {
  c->value = 0;
 }
 void increment(counter_t *c) {
  c->value++;
}
int get(counter_t *c) {
  return c->value; 19
}
```

THREAD SAFE COUNTER

```
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);
}
```

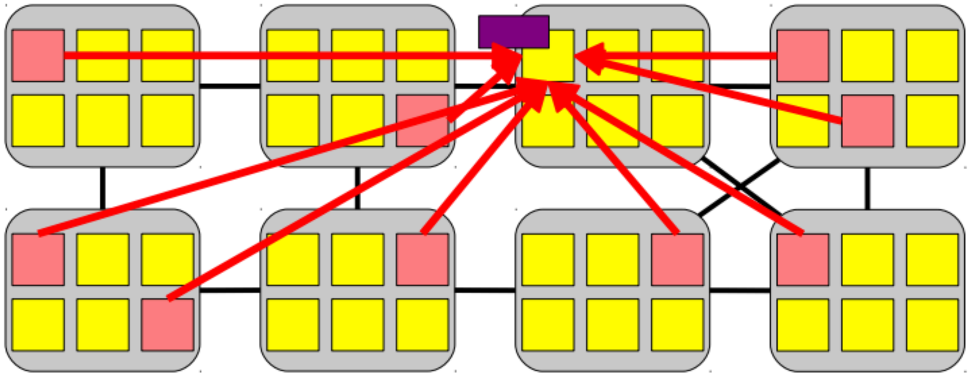
COUNTER SCALABILITY DEMO

UNDERLYING PROBLEM?

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



An Analysis of Linux Scalability
to Many Cores

Boyd-Wickizer et. al
OSDI 2010

APPROXIMATE COUNTERS

Maintain a counter per-core, global counter
Global counter lock
Per-core locks if more than 1 thread per-core?

Increment:
    update local counters
    at threshold update global

Read:
    global counter (maybe inaccurate?)

DEMO

CONCURRENT LINKED LIST

```
void 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; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return; // success
}
```

BETTER CONCURRENT LINKED LIST?

```
void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return; // fail
  }

  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return; // success
}
```

DEMO

HASH TABLE FROM LIST

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

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

```

DEMO

```c
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

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

Figure 29.9: Michael and Scott C

CONCURRENT DATA STRUCTURES

Simple approach: Add a lock to each method?!

Check for scalability – weak scaling, strong scaling

Avoid cross-thread, cross-core traffic

- Per-core counter
- Buckets in hashtable

OPERATING SYSTEMS: THREE EASY PIECES

Three conceptual pieces

1. Virtualization

2. Concurrency

3. Persistence

VIRTUALIZATION

Make each application believe it has each resource to itself

CPU and Memory

Abstraction: Process API, Address spaces

Mechanism:

Limited direct execution, CPU scheduling

Address translation (segmentation, paging, TLB)

Policy: MLFQ, LRU etc.

CONCURRENCY

Events occur simultaneously and may interact with one another

Need to

- Hide concurrency from independent processes
- Manage concurrency with interacting processes

Provide abstractions (locks, semaphores, condition variables etc.)

Correctness: mutual exclusion, ordering

Performance: scaling data structures, fairness

Common Bugs!

NEXT STEPS

Spring break!