CONCURRENCY: DATA STRUCTURES

Shivaram Venkataraman
CS 537, Spring 2019
Spring break!
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
Fix input size, increase number of cores

E.g., \( A \times B \)

\( n \times n \times n \times d \)

\( = O(m \cdot n \cdot d) \)

Flops

Equal floating point ops per second

Weak scaling
Increase input size with number of cores

1 core: 100 x 100
2 cores: 100 x 200
4 cores: 100 x 400

Time to multiply

1 core
2 cores
4 cores
8 cores
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 int get(counter_t *c) {
14   return c->value;
15 }
16
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
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?)
CONCURRENT LINKED LIST

18 void 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; // fail
25     }
26     new->key = key;
27     new->next = L->head;
28     L->head = new;
29     pthread_mutex_unlock(&L->lock);
30     return; // success
31 }

head pointer
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
#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);
}
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;
}
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

Critical section small

Java

Class C

public synchronized get() {

3

< work >
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!