HI!

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*

Bugs ↳ Deadlocks

ABSTRACTIONS

Concurrency

Objects, Lists, Hashtable

OS

C library

Spin locks

Semaphores

equally

Locks, Condition variables

xchg or Test And

Arch

Instructions

Atomic Primitives

CAS

CONCURRENT DATA STRUCTURES

CONCURRENT DATA STRUCTURES

Counters

Lists

Hashtable

Queues

Thread safety

Mutual exclusion

ordering

Start with a correct solution

Make it perform better!

Still correct

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

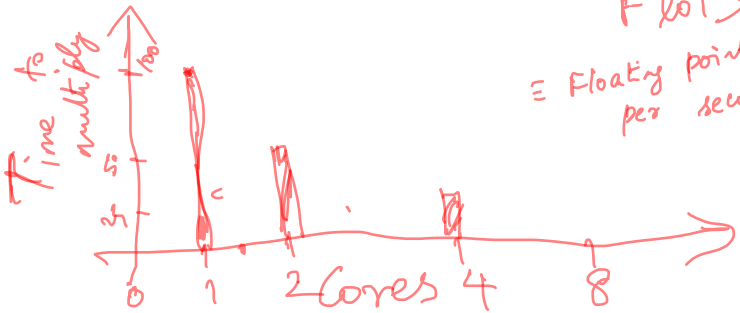
e.g. $A \times B$

$m \times n$ $n \times d$

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

= Floating point ops per second

"Linear scaling"



Weak scaling

Increase input size with number of cores

	A		B	
1 core :	100	× 100	100 × 100	10^6
2 cores :	100	× 200	200 × 100	2×10^6
4 cores :	100	× 400	400 × 100	

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
 }
```

initialize to zero

adds 1 value

return

If two threads increment value shld increase by 2

THREAD SAFE COUNTER

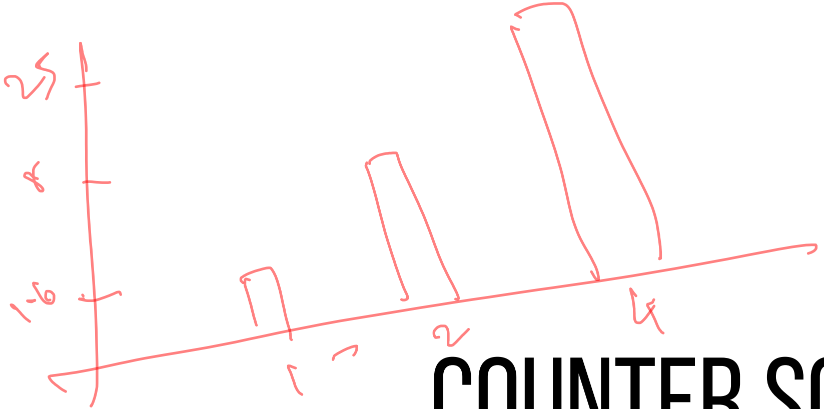
```
typedef struct __counter_t {
  int value;
  pthread_mutex_t lock;
} counter_t;

…

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

CORRECT → be How?

T1 T2 T3
inc inc (get)

LINEARIZABILITY

```
int get(counter_t *c) {
  pthread_mutex_lock(c->lock);
  c->value;
  pthread_mutex_unlock(c->lock);
}
```



COUNTER SCALABILITY DEMO

UNDERLYING PROBLEM?

Ticket lock

up to 48 cores

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

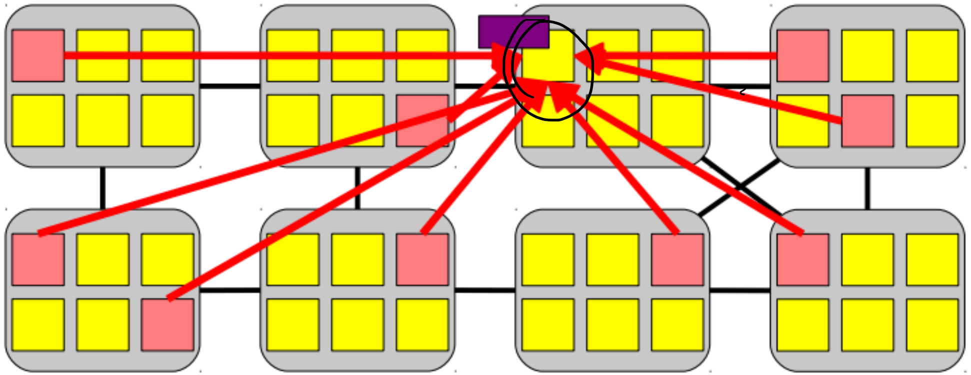
TURN

```
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

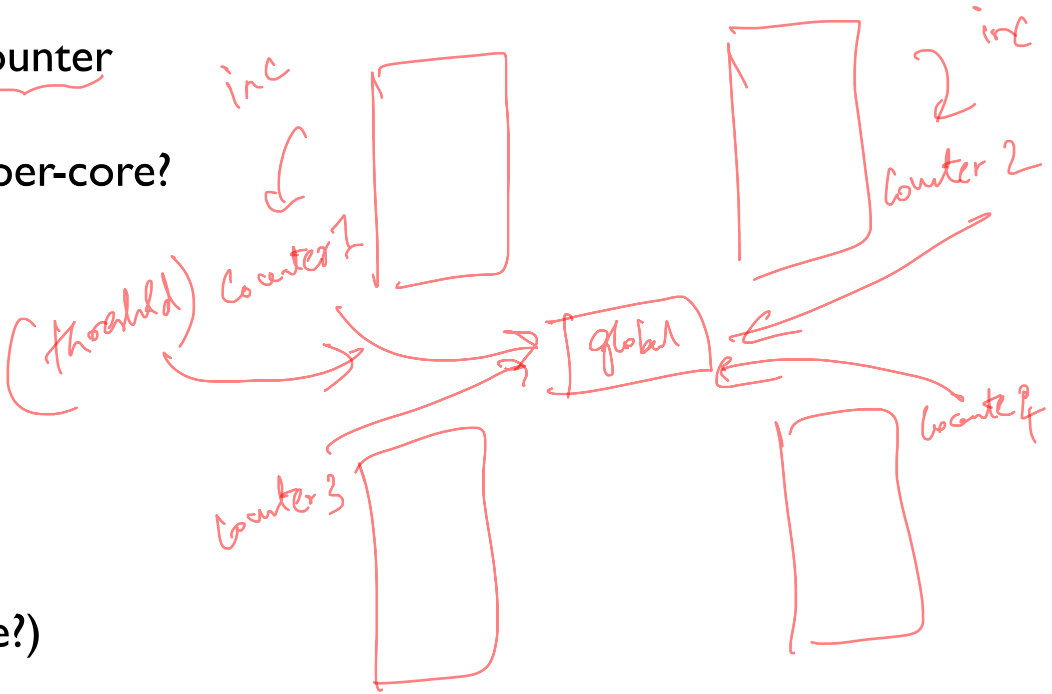


APPROXIMATE COUNTERS

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?)

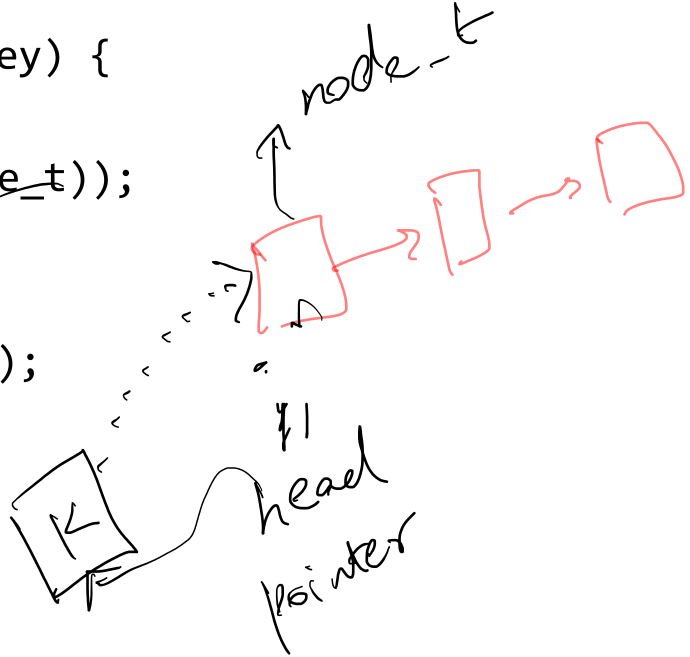


DEMO

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
}
```

critical



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

Thread safe

Keep your critical section small

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);
  }

```

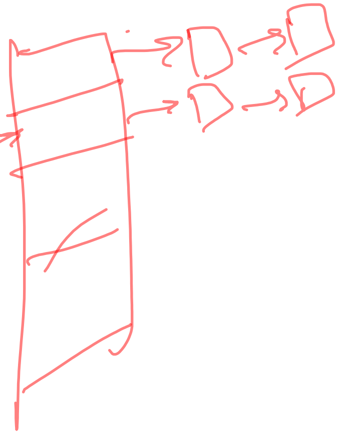
hashing?

hash(key) → hashed key

lookup

value

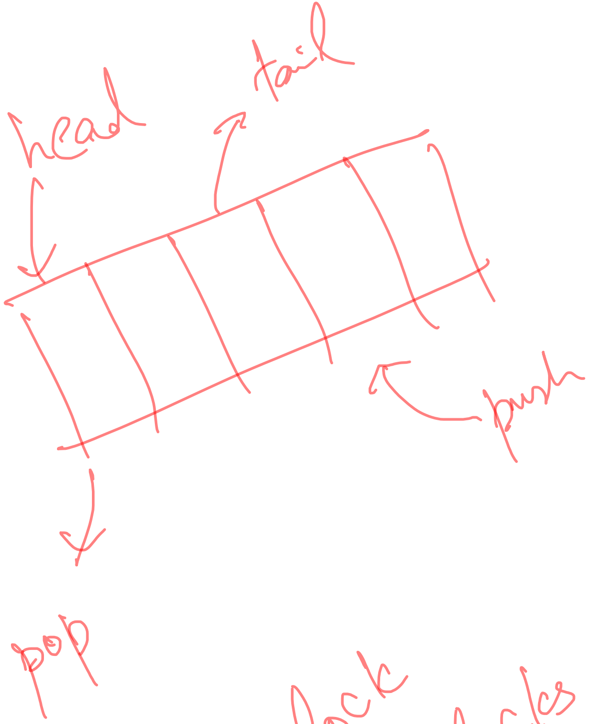
101



If you can avoid contention by using different locks, ⇒ better scaling

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;
}
```



1 lock to 2 locks

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 → Relax consistency

Buckets in hashtable → locks more fine grained

Critical sections small

Java

```
Class Queue {
    public synchronized get() {
        < 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 → Scheduler Memory allocation Stack growth

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!