Hello!

CONCURRENCY: DATA STRUCTURES

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Midterm 1 - ?!? → P2, P3 & Midterm grades

Project 4 out soon?

Office hours: Friday 3-4pm (this week only)

AGENDA / LEARNING OUTCOMES

Concurrency

- How do we design locks?
- How to build concurrent data structures?

RECAP

LOCK IMPLEMENTATION WITH XCHG

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = ??;
}

void acquire(lock_t *lock) {
    while (xchg(&lock->flag, 1) == 1);
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = 0;
}
```

Atomic instructions

```
int xchg(int *addr, int newval)
```

TICKET LOCK IMPLEMENTATION

Fairness

```
typedef struct __lock_t {
    int ticket;
    int turn;
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    // spin
    while (lock->turn != myturn);
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}
```

fetch & add

spin waiting : CPU cycles wasted

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

(a) Why is **guard** used?

spin-lock around flag and queue ops

(b) Why okay to **spin** on guard?

time spent spinning is limited

(c) In release(), why not set lock=false when unpark?

thread woken up, as if it is returns from park

(d) Is there a race condition?

```
void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        park(); // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (XCHG(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
```

RACE CONDITION

Thread 1 (in lock)

```
if (l->lock) {
    qadd(l->q, tid);
    l->guard = false;
```

spin lock

```
    park();     // block
```

→ never be unblocked as its TID is removed from Q

Thread 2 (in unlock)

```
while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
```

called on a thread which is not blocked

BLOCK WHEN WAITING: FINAL CORRECT LOCK

```
typedef struct {
   bool lock = false;
   bool guard = false;
   queue_t q;
} LockT;
```

setpark() fixes race condition

```
void acquire(LockT *l) {
   while (TAS(&l->guard, true));
   if (l->lock) {
       qadd(l->q, tid);
       setpark(); // notify of plan
       l->guard = false;
       park(); // unless unpark()
   } else {
       l->lock = true;
       l->guard = false;
   }
}
void release(LockT *l) {
   while (TAS(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

tell OS that I will call park in future

while holding spin lock

SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances

Uniprocessor

- Waiting process is scheduled → Process holding lock isn't
- Waiting process should always relinquish processor
- Associate queue of waiters with each lock (as in previous implementation)

Multiprocessor

- Waiting process is scheduled → Process holding lock might be
- Spin or block depends on how long, t, before lock is released
  - Lock released quickly → Spin-wait (t << C)
  - Lock released slowly → Block (t >= C)
  - Quick and slow are relative to context-switch cost, C

QUIZ 10

https://tinyurl.com/cs537-fa24-q10

```
a = 1
int b = xchg(&a, 2)
int c = CompareAndSwap(&b, 2, 3)
int d = CompareAndSwap(&b, 1, 3)
```

A = 2

B = ~~1~~ ~~1~~ 3

C = 1

D = 1

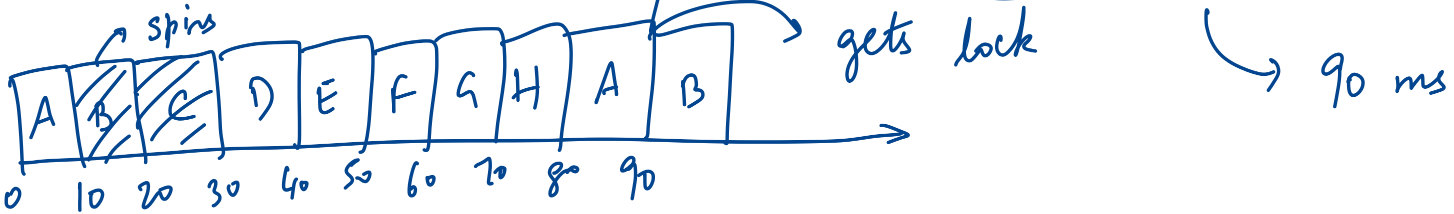
```
int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}

int CAS(int *addr, int ex, int n) {
  int actual = *addr;
  if (actual == ex)
    *addr = n;
  return actual;
}
```

→ ~~False~~ True

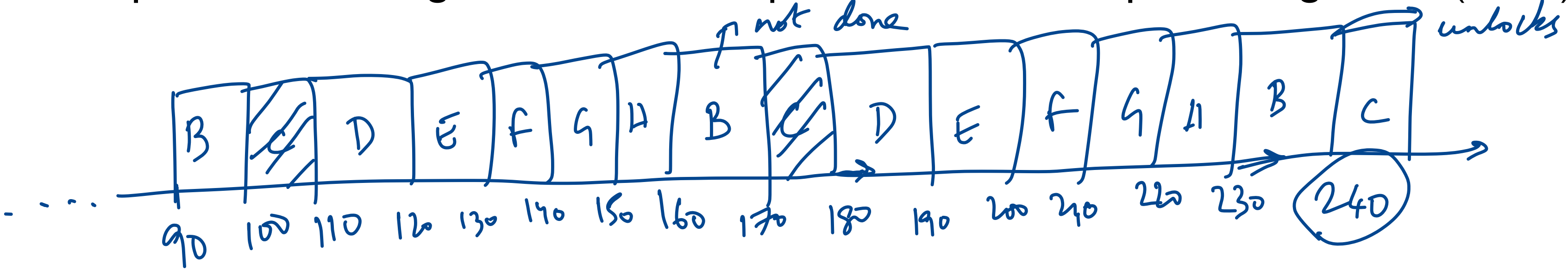
Assuming round-robin scheduling, 10ms time slice with processes A, B, C, D, E, F, G, H in a single CPU system, with processes C - H long-running jobs.

Timeline
A: lock() … compute … unlock()
B: lock() … compute … unlock()
C: lock()

A's compute is 20ms, starting at t = 0, when does B acquire the lock with spin waiting locks?



B's compute is 30ms long, when does C acquire the lock with spin waiting locks? (in ms)



CONCURRENT DATA STRUCTURES

CONCURRENT DATA STRUCTURES

- Counters
- Lists
- Hashtable
- Queues

→ data structures → correctly even if multiple threads use them

Start with a correct solution

Make it perform better!

WHAT IS SCALABILITY

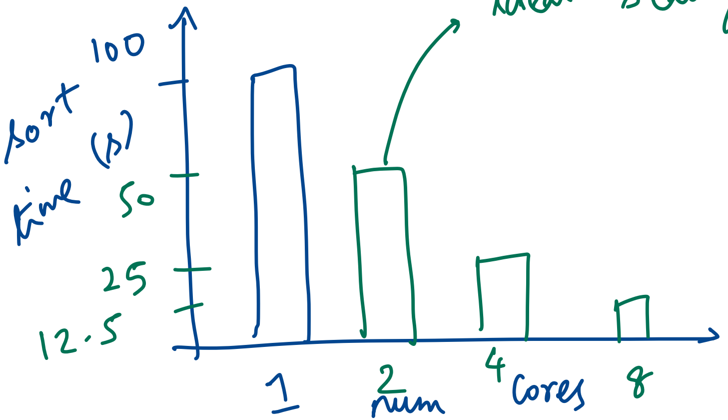
same number of integers

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

Strong scaling

Fix input size, increase number of cores

ideal scaling



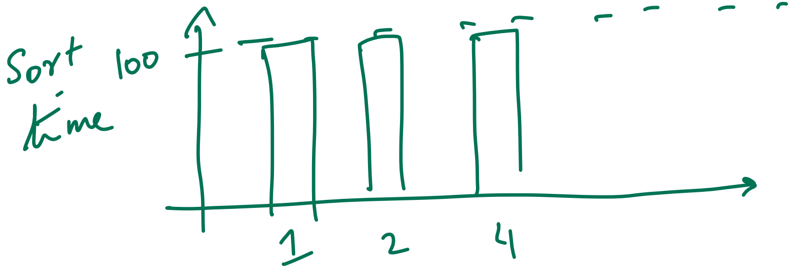
Weak scaling

Increase input size with number of cores

n = 1000 p = 1

= 2 × 1000 p = 2

= 4 × 1000 = 4



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;
}
```

T1 increment()

T2 increment()

correctness = increment 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);
}
```

T1 , T2

ensures correctness

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++;
}
```

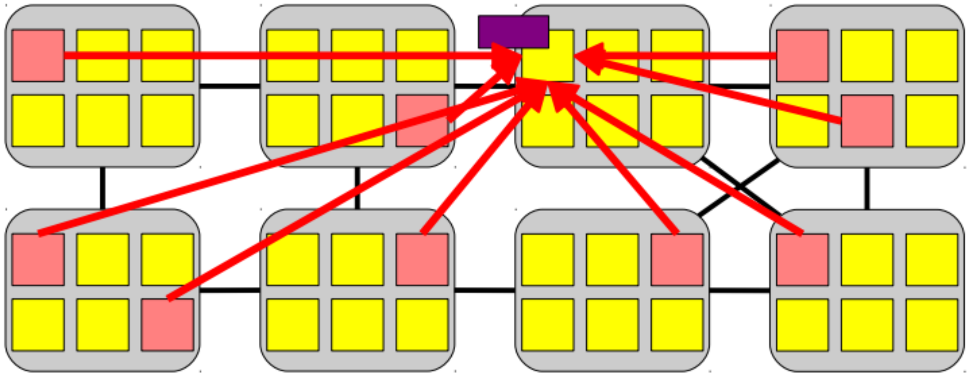
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

ticket lock

48 cores

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?

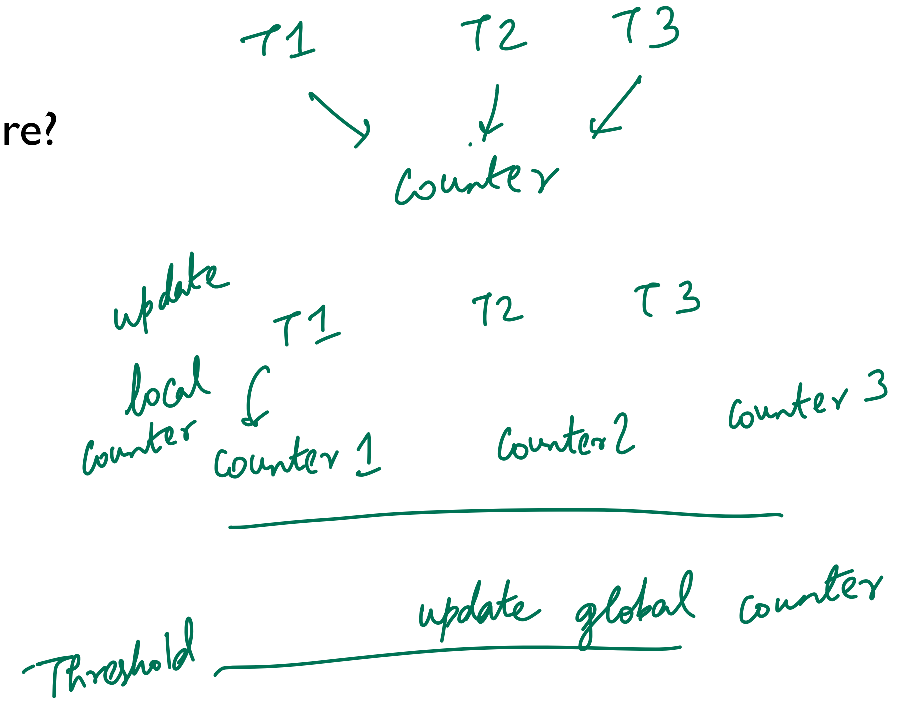
Increment:

update local counters

at threshold update global

Read:

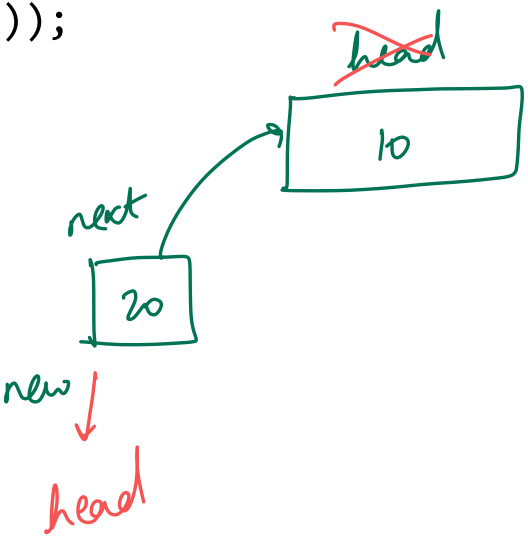
global counter (maybe inaccurate?)



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");
    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

→ takes some time

→ only 1 thread does this at a time

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);
  }

```

hash collisions are kept in a linked list

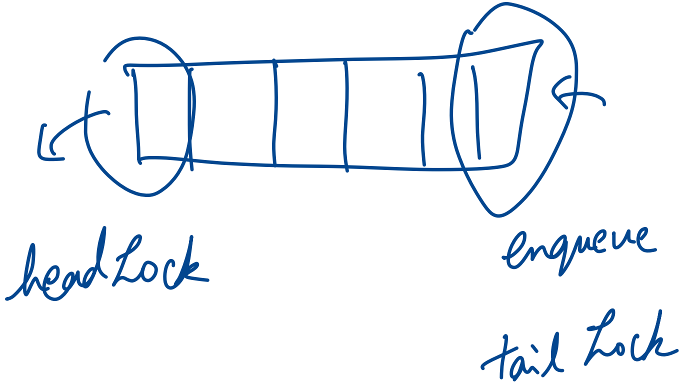
decreased lock contention!

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;
}
```

similar in spirit to hash table

Queue lock → single lock



CONCURRENT DATA STRUCTURES

Simple approach: Add a lock to each method?!

Check for scalability – weak scaling, strong scaling

Java

Avoid cross-thread, cross-core traffic

- Per-core counter → local
- Buckets in hashtable → decreased contention
- Keep critical sections small!

  ↳ linked list

NEXT STEPS

Condition Variables