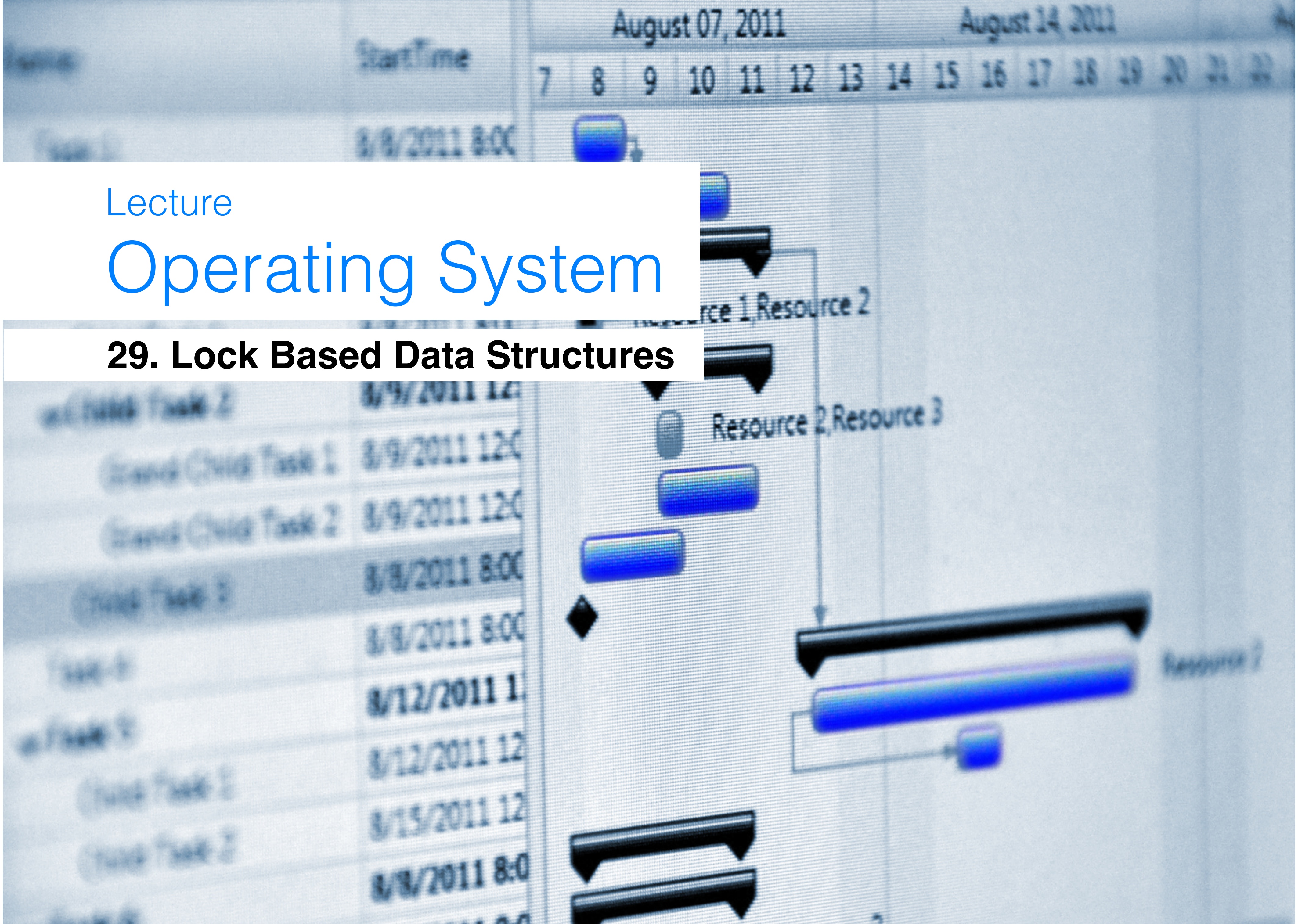


Lecture

Operating System

29. Lock Based Data Structures

Professor Dr. Michael Mächtel

29. Lock Based Data Structures

1. **Counter**
2. **Linked List**
3. **Queue**
4. **Hash Table**

29. Lock Based Data Structures

1. **Counter**
2. Linked List
3. Queue
4. Hash Table

Criteria

- Adding locks to a data structure makes the structure **thread safe**.
  - How locks are added determine both the correctness and performance of the data structure.

Example: Concurrent Counter

simple, but not scalable

```c
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) {
  c->value--;
}

int get(counter_t *c) {
  return c->value;
}
```

Example: Concurrent Counter

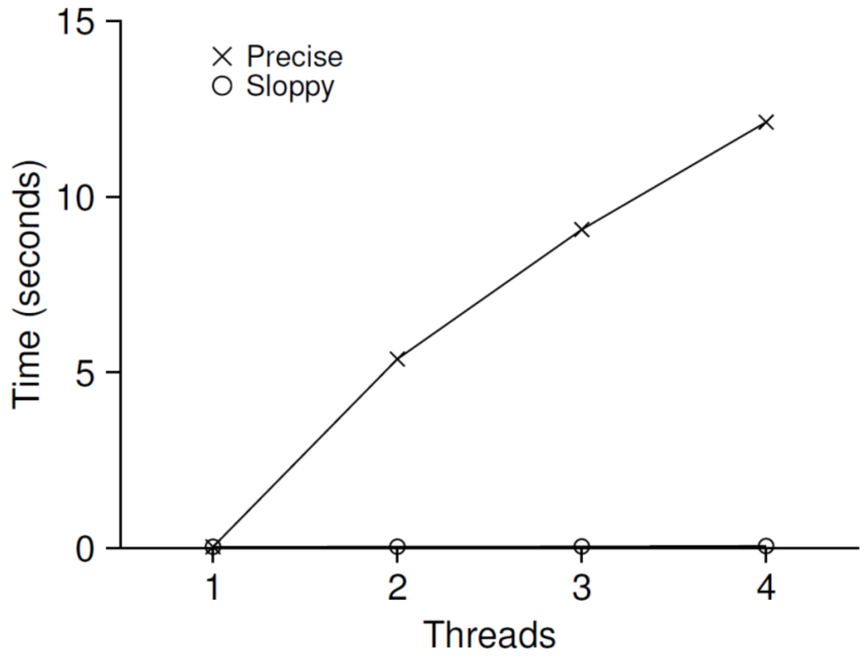
Add a **single lock**

```c
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) {
  Pthread_mutex_lock(&c->lock);
  c->value--;
  Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
   Pthread_mutex_lock(&c->lock);
   int rc = c->value;
   Pthread_mutex_unlock(&c->lock);
   return rc;
}
```

```c
typedef struct __counter_t {
  int value;
  pthread_lock_t lock;
} counter_t;
```

The performance costs

- Each thread updates a single shared counter.
  - Each thread updates the counter one million times.
  - iMac with four Intel 2.7GHz i5 CPUs.



Performance of
Traditional vs. Sloppy Counters
(Threshold of Sloppy, S, is set to 1024)

Perfect Scaling

- Even though more work is done, it is done in parallel.
- The time taken to complete the task **is not increased**.

Sloppy Counter

- The sloppy counter works by representing …
  - A single **logical counter** via numerous local physical counters, on per CPU core
  - A single **global counter**
  - There are **locks**:
    - One fore each local counter and one for the global counter
- Example: on a machine with four CPUs
  - Four local counters
  - One global counter

The basic idea of sloppy counting

When a thread running on a core wishes to increment the counter.

- It increment its local counter.
- Each CPU has its own local counter:
  - Threads across CPUs can update local counters *without contention*.
  - Thus counter updates are scalable.
- The local values are periodically transferred to the global counter.
  - Acquire the global lock
  - Increment it by the local counter's value
  - The local counter is then reset to zero.

The basic idea of sloppy counting

- How often the local-to-global transfer occurs is determined by a threshold, S (sloppiness).
  - The smaller S:
    - The more the counter behaves like the non-scalable counter.
  - The bigger S:
    - The more scalable the counter.
    - The further off the global value might be from the actual count.

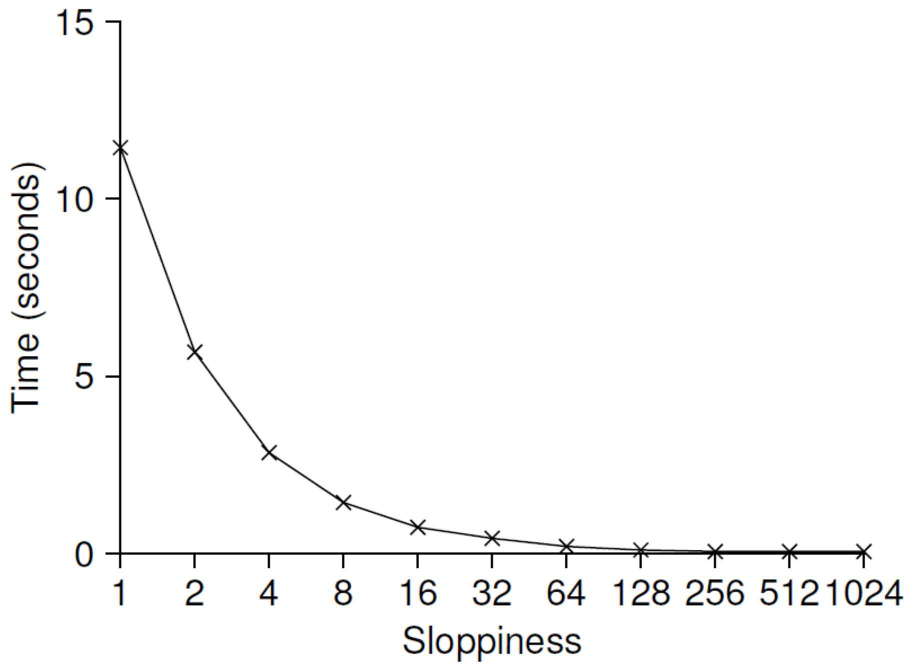
Sloppy counter example

- Tracing the Sloppy Counters
  - The threshold S is set to 5.
  - There are threads on each of four CPUs
  - Each thread updates their local counters $L_1 \ldots L_4$.

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 ➜ 0	1	3	4	5 (from L_1)
7	0	2	4	5 ➜ 0	10 (from L_4)

Importance of the threshold value S

- Each four threads increments a counter 1 million times on four CPUs.
- Low S ➙ Performance is **poor**, The global count is always quire **accurate**.
- High S ➙ Performance is **excellent**, The global count **lags**.



Scaling Sloppy Counters

Sloppy Counter Implementation

```c
typedef struct __counter_t {
    int global;                  // global count
    pthread_mutex_t glock;       // global lock
    int local[NUMCPUS];          // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS]; //  ... and locks
    int threshold;               // update frequency
} counter_t;

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->thres hold = 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);
    }
}
```

Sloppy Counter Implementation

```
// update: usually, just grab local lock and update local amount
//         once local count has risen by 'threshold', grab global
//         lock and transfer local values to it
void update(counter_t *c, int threadID, int amt) {
    pthread_mutex_lock(&c->llock[threadID]);
    c->local[threadID] += amt;    // assumes amt > 0
    if (c->local[threadID] >= c->threshold) { // transfer to global
        pthread_mutex_lock(&c->glock);
        c->global += c->local[threadID];
        pthread_mutex_unlock(&c->glock);
        c->local[threadID] = 0;
    }
    pthread_mutex_unlock(&c->llock[threadID]);
}

// get: just return global amount (which may not be perfect)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val;  // only approximate!
}
```

29. Lock Based Data Structures

1. Counter
2. **Linked List**
3. Queue
4. Hash Table

Concurrent Linked Lists

```c
// basic node structure
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

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

Concurrent Linked Lists

```c
int 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 -1; // fail
    }
    new->key = key;
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}
```

```c
int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *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
}
```

Discussion

- The code **acquires** a lock in the insert routine upon entry.
- The code **releases** the lock upon exit.
  - If `malloc()` happens to fail, the code must also release the lock before failing the insert.
  - This kind of exceptional control flow has been shown to be **quite error prone**.
- **Solution**: The lock and release *only surround* the actual critical section in the insert code

Concurrent Linked Lists rewritten

```c
void List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
}
```

```c
int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

Scaling Linked List

- How does the single lock linked list perform?
- Alternative: Hand-over-hand locking (lock coupling)
  - **Add a lock per node** of the list instead of having a single lock for the entire list.
  - When traversing the list,
    - First grabs the next node's lock.
    - And then releases the current node's lock.
  - Enable a high degree of concurrency in list operations.
  - However, in practice, **the overheads** of acquiring and releasing locks for each node of a list traversal is prohibitive.

29. Lock Based Data Structures

1. Counter
2. Linked List
3. **Queue**
4. Hash Table

Concurrent Queue

- Simple Concurrent Queue
  - Add one lock
  - Scaling?
- Alternative:
  - Michael and Scott Concurrent Queues

Michael and Scott Concurrent Queues

- There are two locks.
  - One for the **head** of the queue.
  - One for the **tail**.
  - The goal of these two locks is to enable concurrency of *enqueue* and *dequeue* operations.
- Add a dummy node
  - Allocated in the queue initialization code
  - Enable the separation of head and tail operations
- Run the code in the book (OSTEP) !

29. Lock Based Data Structures

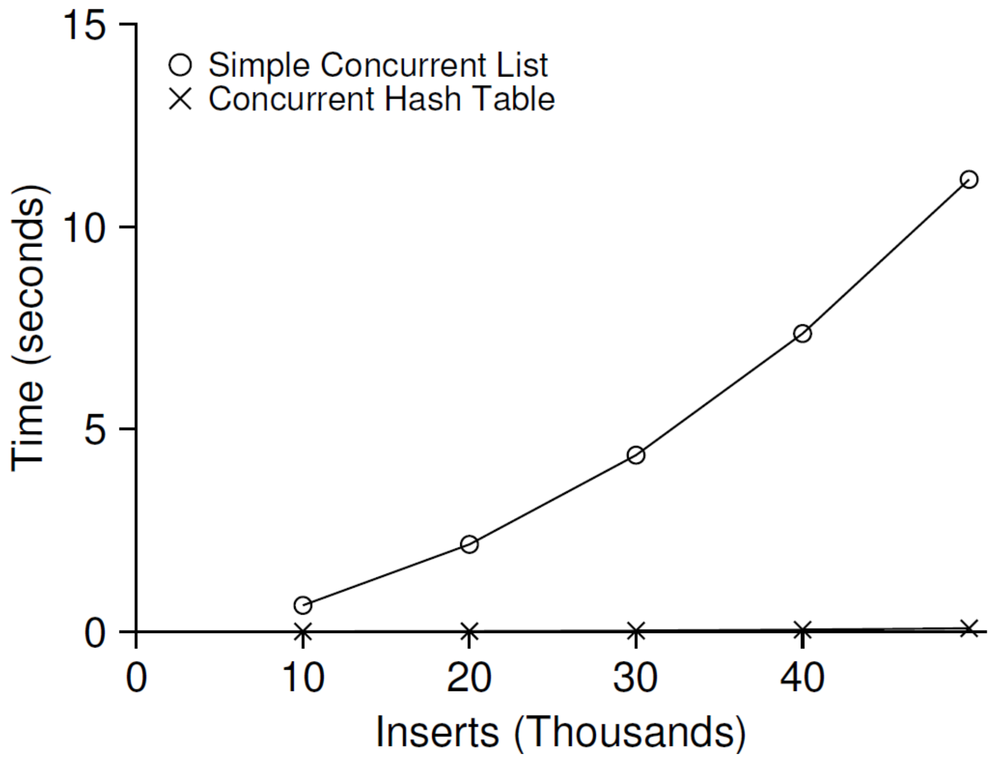
1. Counter
2. Linked List
3. Queue
4. **Hash Table**

Concurrent Hash Table

- Focus on a simple hash table
  - The hash table does not resize.
  - Built using the concurrent lists
  - It uses a lock per hash bucket each of which is represented by a *list*.

Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates from each of four threads.
  - iMac with four Intel 2.7GHz i5 CPUs.



The simple concurrent hash table scales magnificently.

Concurrent Hash Table

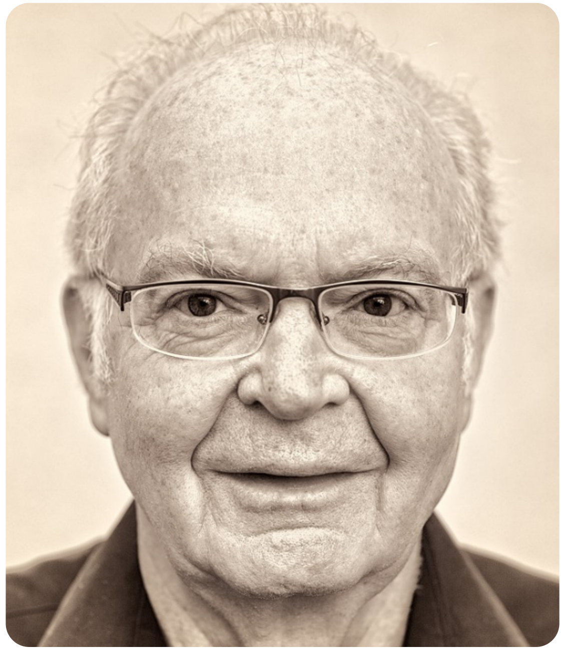
```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    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);
}
```

Learned important lessons

- **be careful** with acquisition and release of locks around control flow changes
- **enabling more** concurrency does not **necessarily increase** performance
- performance problems should only be remedied once they **exist**
- avoid premature optimization
    - there is no value in making **something faster** if doing so **will not improve** the overall **performance** of the **application**

“Premature optimization is the root of all evil.”

— Donald Knuth [2]



Thanks

Questions?