Concurrent Data Structures

CMPU 224 – Computer Organization

Jason Waterman

Critical section

- In the previous example, there are three actions that take place to increase the counter
  - **Read** the current value from memory
  - **Modify** the value
  - **Write** the new value to memory
- These three operations must happen **atomically**
  - Once the read takes place, the thread should not be interrupted by another thread that uses the shared variable until after the modify and write occur
- A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread is called a **critical section**
  - Multiple threads executing critical section can result in a **race condition**
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

- Locks ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**)

```
1     lock_t mutex;
2     . . .
3     lock(&mutex);
4     counter = counter + 1;
5     unlock(&mutex);
```

Critical section

Locks

- Provide **mutual exclusion** to a critical section
  - Interface – two operations **lock** and **unlock**

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (without lock initialization and error checking)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - If no other thread holds the lock → the thread will acquire the lock and **enter the critical section**
    - If another thread holds the lock → the thread will **not return from the call** until it has acquired the lock

Lock Initialization

- All locks must be **properly initialized**
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Lock Error Checking

- Always check the return value for errors when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the **pthreads library**, by adding the `-pthread` flag

```
prompt> gcc -o main main.c -Wall -pthread
```

  - For more information

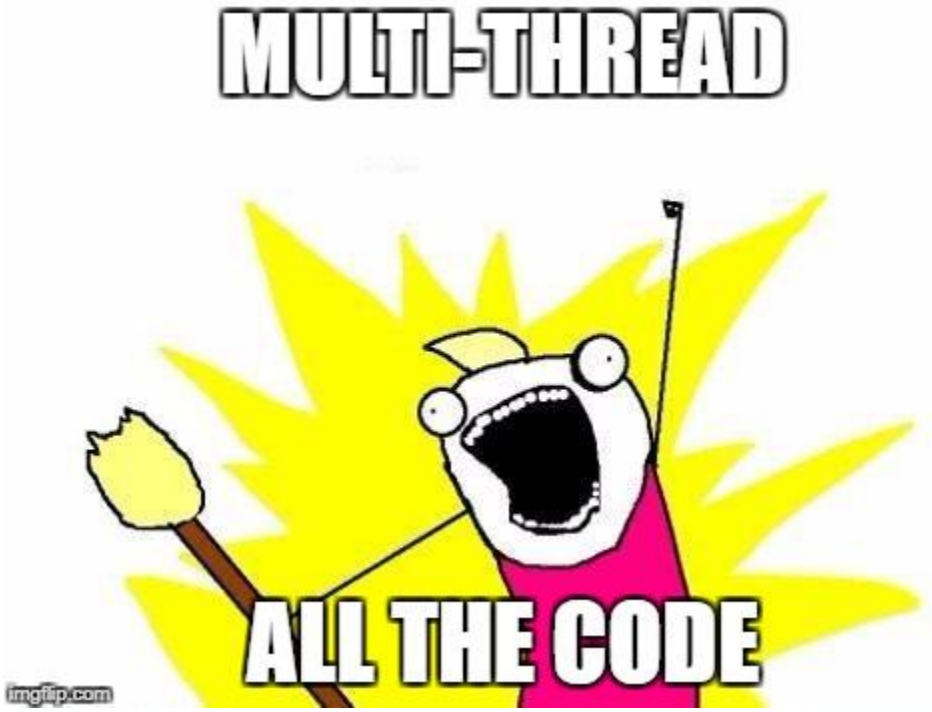
```
man -k pthread
```

Thread API Guidelines

- **Keep it simple**
  - Tricky thread interactions lead to bugs
- **Minimize thread interactions**
  - Each interaction should be carefully thought out and constructed with known design patterns (which we will learn about)
- **Check your return codes**
  - Failure to do so will lead to bizarre and hard to understand behavior
- **Be careful how you pass arguments and return values**
  - If you returning a reference to a variable on the stack, you are going to have a bad time
- **Each thread has its own stack**
  - To share data between threads, the values must be in the heap or a global variable
- **Use the man pages**
  - Highly informative with good examples

Lock-based Concurrent Data structures

- Adding threads is a good way to parallelize your program
- Must be done correctly however
  - 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
  - Adding threads may actually slow down your code



Example: Concurrent Counters without Locks

- Single threaded

```
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 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Example: Concurrent Counters with Locks

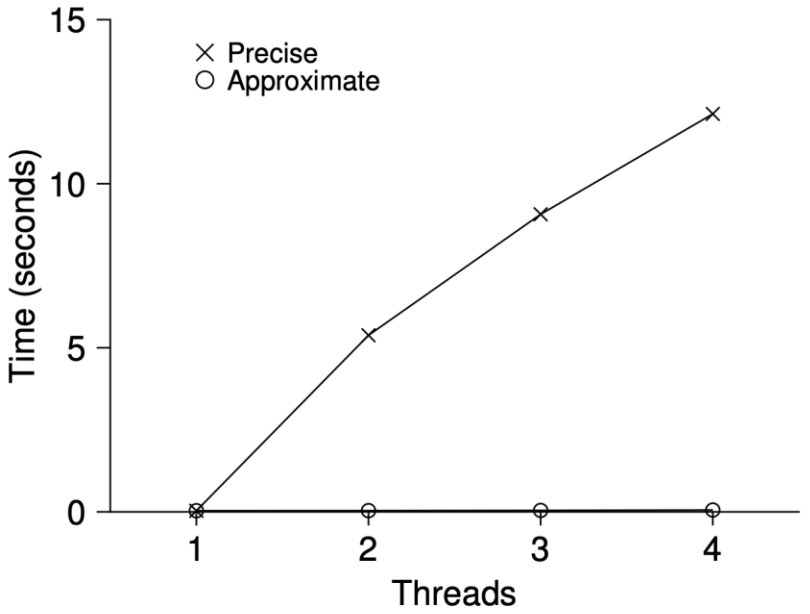
- Add a **single lock**
  - The lock is acquired when calling a routine that manipulates the data structure

```
1    typedef struct __counter_t {
2        int value;
3        pthread_lock_t lock;
4    } counter_t;
5
6    void init(counter_t *c) {
7        c->value = 0;
8        Pthread_mutex_init(&c->lock, NULL);
9    }
10
11   void increment(counter_t *c) {
12       Pthread_mutex_lock(&c->lock);
13       c->value++;
14       Pthread_mutex_unlock(&c->lock);
15   }
16
```

```
17   void decrement(counter_t *c) {
18       Pthread_mutex_lock(&c->lock);
19       c->value--;
20       Pthread_mutex_unlock(&c->lock);
21   }
22
23   int get(counter_t *c) {
24       Pthread_mutex_lock(&c->lock);
25       int rc = c->value;
26       Pthread_mutex_unlock(&c->lock);
27       return rc;
28   }
```

The performance costs of the simple approach

- Each thread updates a single shared counter
  - Each thread updates the counter one million times
  - iMac with four Intel 2.7GHz i5 CPUs
- Ideally threads complete just as quickly on multiple processors as a single thread does on one
  - Even though more work is done, it is **done in parallel**
  - The time taken to complete the task on each core is *not increased*
- For our example:
  - Single thread on one core: about 0.03 seconds
  - Two threads running concurrently: little over 5 seconds



Synchronized counter scales poorly

Approximate counter

- The approximate counter works by representing:
  - A single **logical counter,** via numerous local physical counters, **one per CPU core**
  - A single **global counter**
  - There are multiple **locks**
    - One for each local counter and one for the global counter

- Example: on a machine with four CPUs
  - Four local counters
  - One global counter

Basic idea of approximate counting

- When a thread running on a core wishes to increment the counter
  - It increments its local counter
  - Each CPU has its own local counter
    - Threads across CPUs can update local counters *without contention*
    - Therefore, 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

Approximation Threshold

- How often the local-to-global transfer occurs is determined by threshold S
  - 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*
      - Worst case: `S * NUMCPUS`

Approximate counter example

- Tracing the Approximate 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)

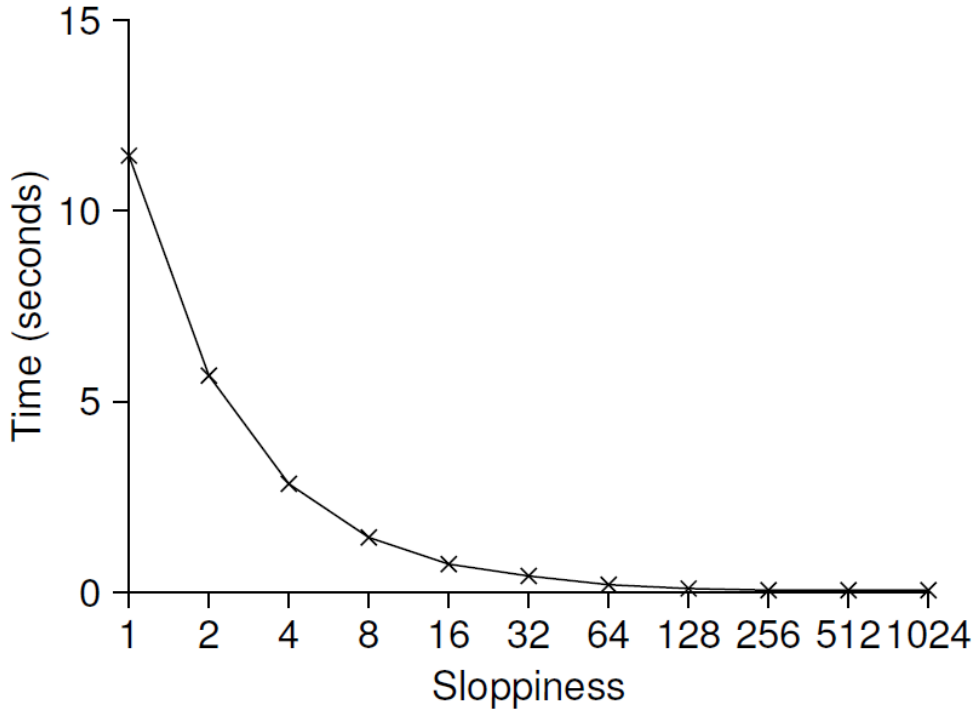
Approximate Counter Implementation

```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->threshold = 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);
  }
}
```

```c
// 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) {
  int cpu = threadID % NUMCPUS;
  pthread_mutex_lock(&c->llock[cpu]);
  c->local[cpu] += amt;  // assumes amt > 0
  // transfer to global
  if (c->local[cpu] >= c->threshold) {
    pthread_mutex_lock(&c->glock);
    c->global += c->local[cpu];
    pthread_mutex_unlock(&c->glock);
    c->local[cpu] = 0;
  }
  pthread_mutex_unlock(&c->llock[cpu]);
}
// 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!
}
```

Importance of the threshold value *S*

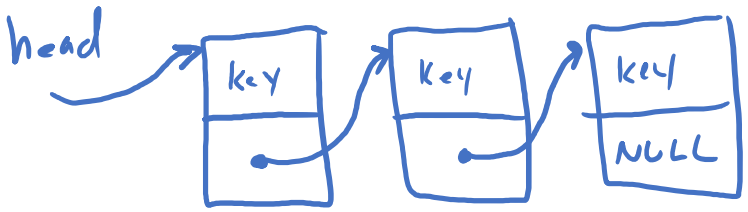
- Four threads each increment a counter 1 million times on four CPUs
  - Low S → Performance is **poor**, the global count is always quire **accurate**
  - High S → Performance is **excellent**, the global count **lags**



Scaling of Approximate Counters

Concurrent Linked Lists

```
// 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);
}
```




```
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
}
```

```
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
}
```

Concurrent Linked Lists

- 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

malloc is thread safe

```
man malloc
...
ATTRIBUTES
       For an explanation of the terms used in this section, see attributes(7).

       +----------------------+---------------+---------+
       |Interface             | Attribute     | Value   |
       +----------------------+---------------+---------+
       |malloc(), free(),     | Thread safety | MT-Safe |
       |calloc(), realloc()   |               |         |
       +----------------------+---------------+---------+

man 7 attributes
...
 MT-Safe
            MT-Safe  or  Thread-Safe functions are safe to call in the presence of other threads.  MT, in MT-Safe, stands for
            Multi Thread.

            Being MT-Safe does not imply a function is atomic, nor that it uses any of the memory synchronization  mechanisms
            POSIX exposes to users.  It is even possible that calling MT-Safe functions in sequence does not yield an MT-Safe
            combination.  For example, having a thread call two MT-Safe functions one right after the other does not  guaran-
            tee  behavior  equivalent to atomic execution of a combination of both functions, since concurrent calls in other
            threads may interfere in a destructive way.

            Whole-program optimizations that could inline functions across library interfaces may expose  unsafe  reordering,
            and  so performing inlining across the GNU C Library interface is not recommended.  The documented MT-Safety sta-
            tus is not guaranteed under whole-program optimization.  However, functions defined in user-visible  headers  are
            designed to be safe for inlining.
```

Concurrent Linked List: Rewritten

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

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
}
```

Concurrent Queues

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t lock;
} Queue;

void queue_init(Queue *q) {
    q->head = q->tail = NULL;
    pthread_mutex_init(&q->lock, NULL);
}
```

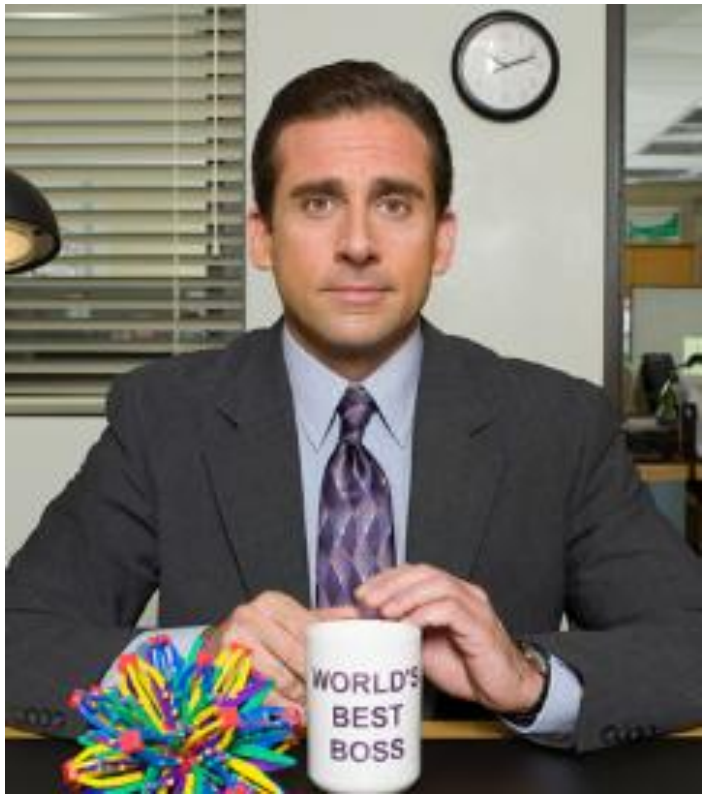
- Single lock
- Enqueue/Dequeue operations are serialized

```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->lock);
    if (q->tail == NULL) {
        q->head = q->tail = node;
    } else {
        q->tail->next = node;
        q->tail = node;
    }
    pthread_mutex_unlock(&q->lock);
}

int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->lock);
    if (q->head == NULL) {
        pthread_mutex_unlock(&q->lock);
        return 0;
    }
    Node *front = q->head;
    *value = front->value;
    q->head = front->next;
    if (q->head == NULL) {
        q->tail = NULL;
    }
    pthread_mutex_unlock(&q->lock);

    free(front);
    return 1;
}
```

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

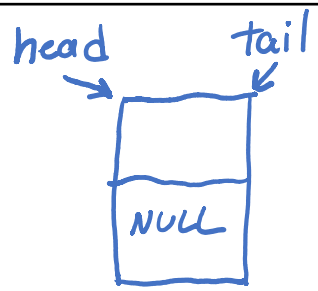
- Add a dummy node
  - Allocated in the queue initialization code
  - Enable the separation of head and tail operations



Not this guy

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

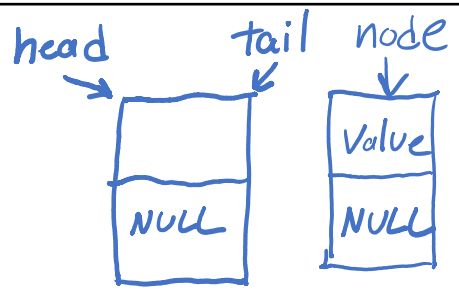



```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

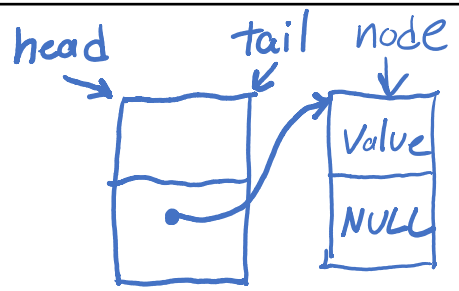



```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

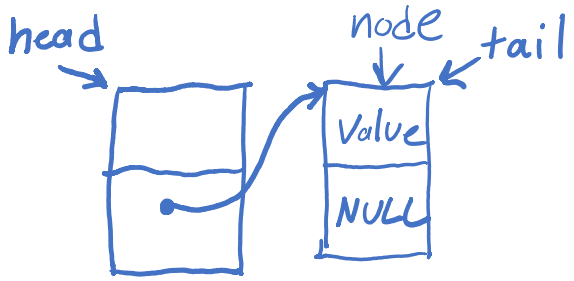



```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

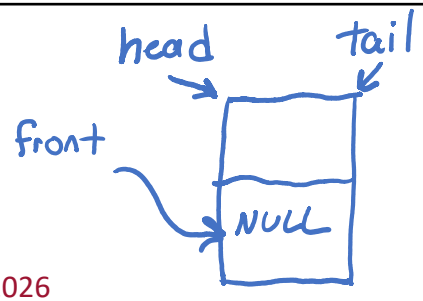



```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```



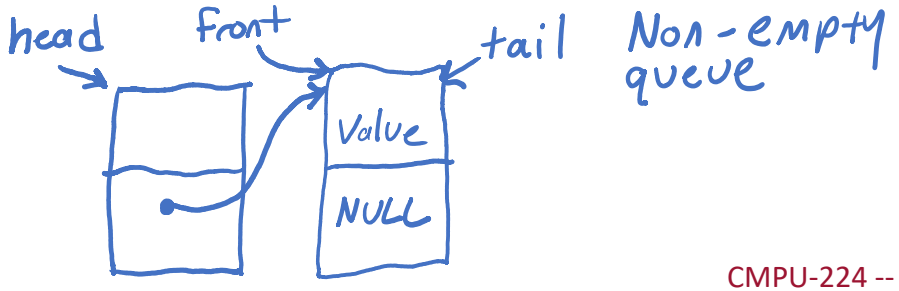
Empty queue case:
front is NULL
return 0

```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

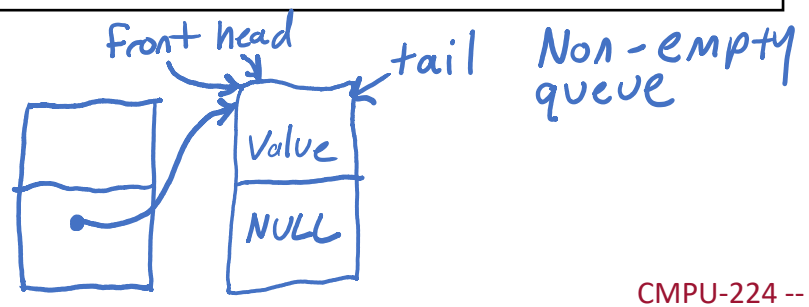



```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

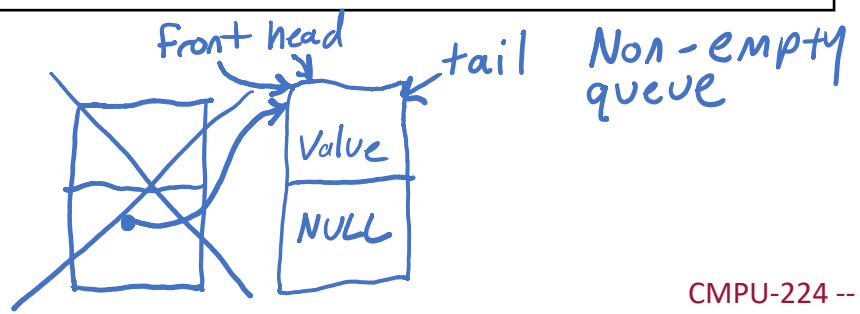



```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```



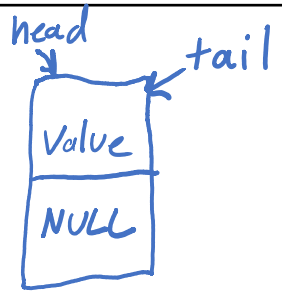

```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

Concurrent Queues (Cont.)

```
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    pthread_mutex_t head_lock;
    pthread_mutex_t tail_lock;
} Queue;

void queue_init(Queue *q) {
    Node *dummy = malloc(sizeof(Node));
    dummy->next = NULL;
    q->head = q->tail = dummy;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}
```

Empty queue




```
void enqueue(Queue *q, int value) {
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = node;
    q->tail = node;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```
int dequeue(Queue *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    Node *head = q->head;
    Node *front = head->next;

    if (front == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return 0;
    }

    *value = front->value;
    q->head = front;
    pthread_mutex_unlock(&q->head_lock);

    free(head);
    return 1;
}
```

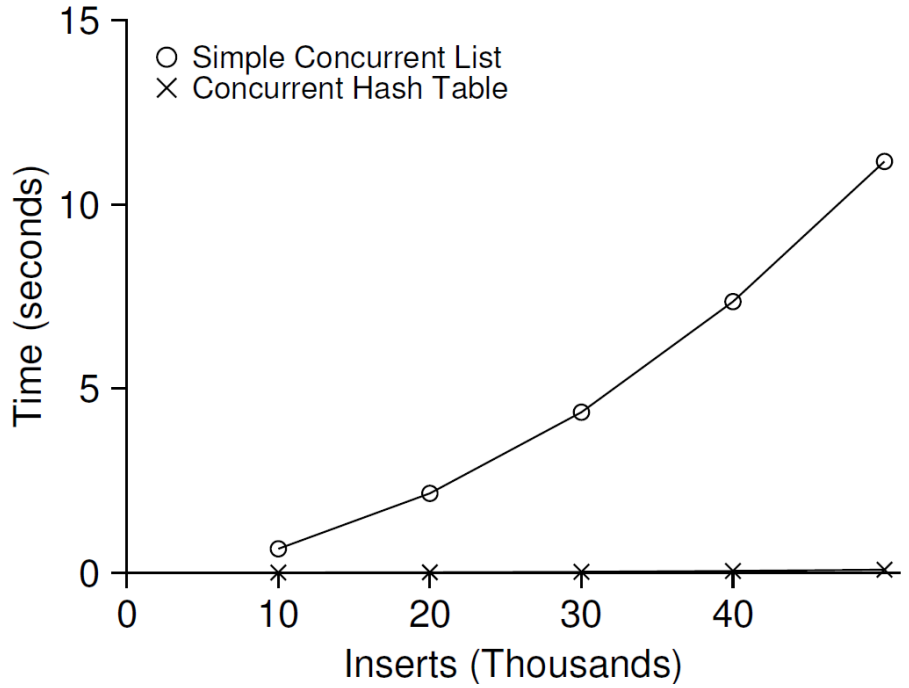
Concurrent Hash Table

- Simple 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*

```
#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);
}
```

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 very well

Summary

- We looked at a few of the concurrent data structures out there
  - Counters
  - Lists
  - Queues
  - Hash Tables
- Tips
  - Be careful with acquiring and releasing locks around control flow changes
  - Enabling more concurrency does not necessarily increase performance
  - Premature optimization is the root of all evil! (Knuth's Law)