Operating Systems

Lock (Chapter 28 ~ 29)

Dr. Young-Woo Kwon

Definitions

- **Race condition**: output of a concurrent program depends on the order of operations between threads
- **Mutual exclusion**: only one thread does a particular thing at a time
- **Critical section**: piece of code that only one thread can execute at once
- **Lock**: prevent someone from doing something

Locks: The Basic Idea

- Ensure that any **critical section** executes as if it were a **single atomic instruction.**
  - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

  - Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks: The Basic Idea

- Lock variable holds <u>the state of </u>the lock.
  - **available** (or **unlocked** or **free**)
    - No thread holds the lock.
  - **acquired** (or **locked** or **held**)
    - Exactly one thread holds the lock and presumably is in a critical section.

The semantics of the lock()

- `lock()`
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the *critical section*.
    - This thread is said to be **the owner of** the lock.
  - Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

Pthread Locks - mutex

- The name that the POSIX library uses for a lock.
  - Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

  - We may be using *different locks* to protect *different variables* → Increase **concurrency** (a more **fine-grained** approach).

Building A Lock

- Efficient locks provided mutual exclusion at **low cost**.
- Building a lock need some help from the **hardware** and the **OS**.

Evaluating locks – Basic criteria

- **Mutual exclusion**
  - Does the lock work, preventing multiple threads from entering *a critical section*?
- **Fairness**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)
- **Performance**
  - The time overheads added by using the lock

Why hardware support needed?

- **First attempt**: Using a *flag* denoting whether the lock is held or not.
  - The code below has problems.

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

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
            ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}`
```

Why hardware support needed? (Cont.)

- **Problem 1**: **No Mutual Exclusion** (assume `flag=0` to begin)

Thread1	Thread2
call `lock()` `while (flag == 1)` interrupt: switch to Thread 2	

	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1

`flag = 1;` // set flag to 1 (too!)	

- **Problem 2**: **Spin-waiting** wastes time waiting for another thread.

• So, we need an **atomic instruction** supported by **Hardware**!

- *test-and-set* instruction, also known as *atomic exchange*

Test And Set (Atomic Exchange)

- An instruction to support the creation of simple locks

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;   // fetch old value at ptr
    *ptr = new;       // store 'new' into ptr
    return old;       // return the old value
}
```

  - **return**(testing) old value pointed to by the `ptr`.
  - *Simultaneously* **update**(setting) said value to `new`.
  - This sequence of operations is **performed atomically.**

A Simple Spin Lock using test-and-set

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

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;        // spin-wait
}

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

– **Note**: To work correctly on a single CPU, it requires a preemptive scheduler.

- **Why?**

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section.
- **Fairness**: no
  - Spin locks don’t provide any fairness guarantees.
  - Indeed, a thread spinning may spin *forever*.
- **Performance**:
  - In the single CPU, performance overheads can be quire *painful*.
  - If **the number of threads roughly equals the number of CPUs**, spin locks work *reasonably well*.

Compare-And-Swap (SPARC)

- Test whether the value at the address(`ptr`) is equal to `expected`.
  - *If so*, update the memory location pointed to by `ptr` with the `new` value.
  - *In either case*, return the actual value at that memory location.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Compare-and-Swap hardware atomic instruction (C-style)

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Spin lock with compare-and-swap

Compare-And-Exchange (x86)

- C-callable x86-version of compare-and-swap

```
char CompareAndSwap(int *ptr, int old, int new) {
    unsigned char ret;

    // Note that sete sets a 'byte' not the word
    __asm__ __volatile__ (
            " lock\n"
            " cmpxchgl %2,%1\n"
            " sete %0\n"
            : "=q" (ret), "=m" (*ptr)
            : "r" (new), "m" (*ptr), "a" (old)
            : "memory");
    return ret;
}
```

Fetch-And-Add

- Atomically increment a value while returning the old value at a particular address.

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- **Ticket lock** can be built with fetch-and add.
  - Ensure progress for all threads. → fairness

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

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

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

So Much Spinning

- Hardware-based spin locks are simple and they work.

- In some cases, these solutions can be quite inefficient.
  - Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value.

How To Avoid *Spinning*?
We'll need OS Support too!

A Simple Approach: Just Yield

- When you are going to spin, **give up the CPU** to another thread.
  - OS system call moves the caller from the ***running state to the ready state***.
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- **Queue** to keep track of which threads are waiting to enter the lock.
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

Using Queues: Sleeping Instead of Spinning

```
 typedef struct __lock_t {
   int flag;          // lock is acquired or not
   int guard;         // to protect the queue
   queue_t *q;
} lock_t;
```

Using Queues: Sleeping Instead of Spinning

```c
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

Waiting is short

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Wakeup/waiting race

```
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

What if a lock holder releases the lock just before the thread B executes "park()"?

- In case of releasing the lock (*thread A*) just before the call to `park()` (*thread B*) → Thread B would **sleep forever** (potentially).
- **Solaris** solves this problem by adding a third system call: `setpark()`.
  - By calling this routine, a thread can indicate it *is about to* `park`.
  - If it happens to be interrupted and another thread calls `unpark` before `park` is actually called, the subsequent `park` returns immediately instead of sleeping.

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

Code modification inside of `lock()`

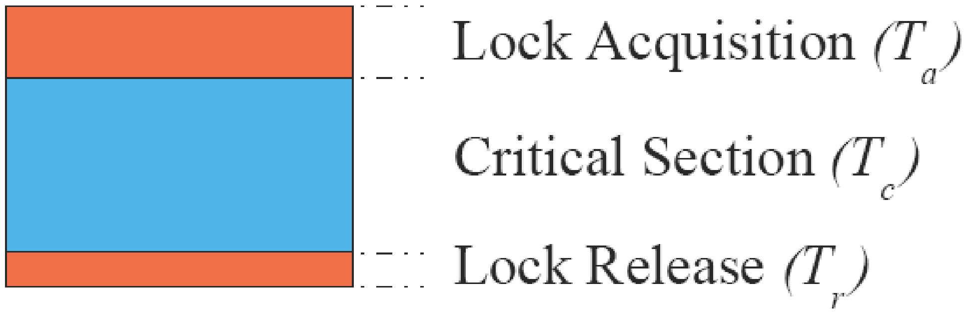
Spinning vs Blocking

- Some lock implementations combine spinning and blocking locks
- Blocking has a cost
  - Shouldn’t block if lock becomes available in less time than it takes to block
- Strategy: spin for time it would take to block
  - Even in worst case, total cost for lock() is less than 2*block time

Two-Phase Locks

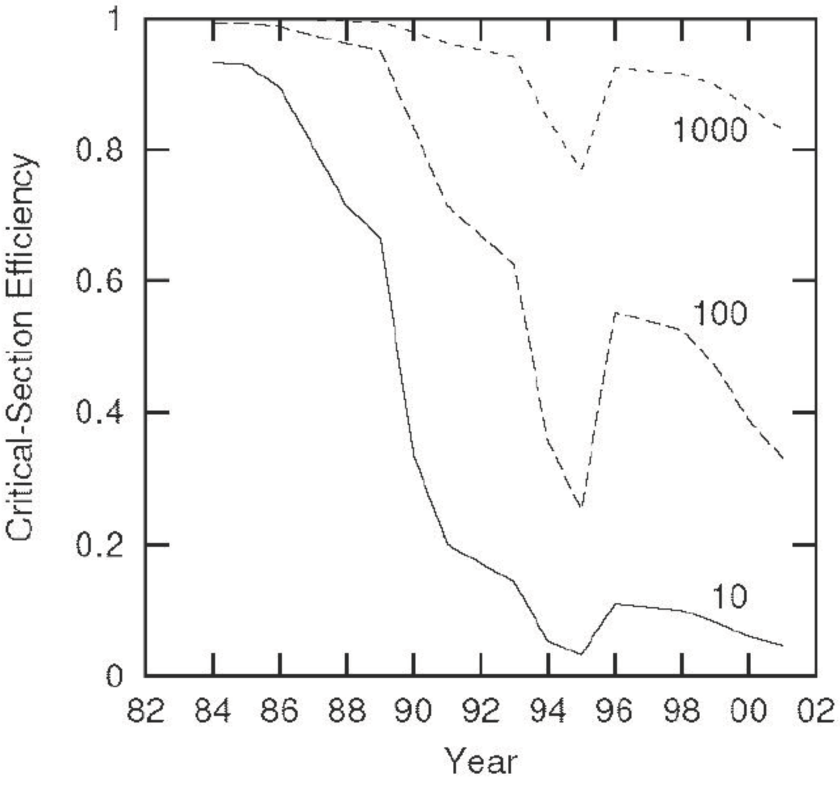
- A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.
  - **First phase**
    - The lock spins for a while, *hoping that* it can acquire the lock.
    - If the lock is not acquired during the first spin phase, <u>a second phase</u> is entered,
  - **Second phase**
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.

Critical Section Efficiency



$$Efficiency = \frac{T_c}{T_c + T_a + T_r}$$

- As processors get faster, CSE decreases because atomic instructions become relatively more expensive



Source: McKenney, 2005

29. Lock-based Concurrent Data Structures

Lock-based Concurrent Data structure

- 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 Counters without Locks

- 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 Counters with Locks

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

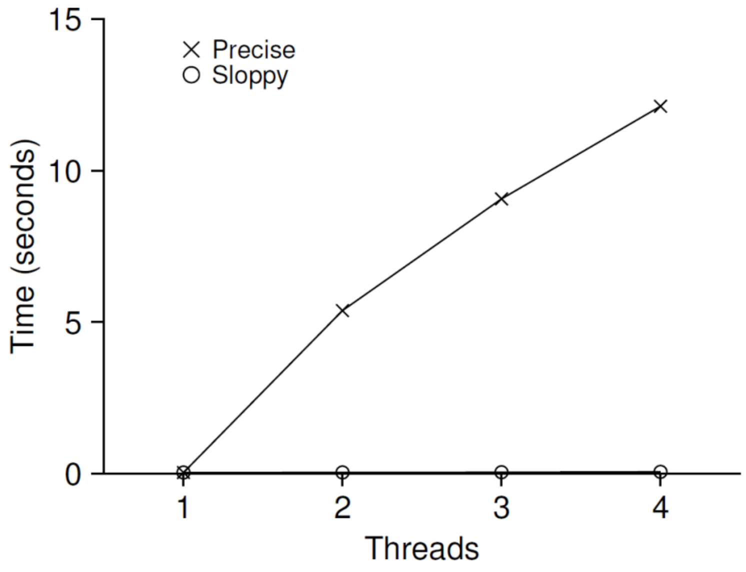
```c
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

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);
}

```

Example: Concurrent Counters with Locks

```
(Cont.)
    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;
    }
```

The performance costs of the simple approach

- Each thread updates a single shared counter.
  - Each thread updates the counter one million times.



Performance of Traditional vs. Sloppy Counters

Synchronized counter scales poorly.

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, <u>on per CPU core</u>
  - A single **global counter**
  - There are **lock**s:
    - One for 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 (Cont.)

- 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*.

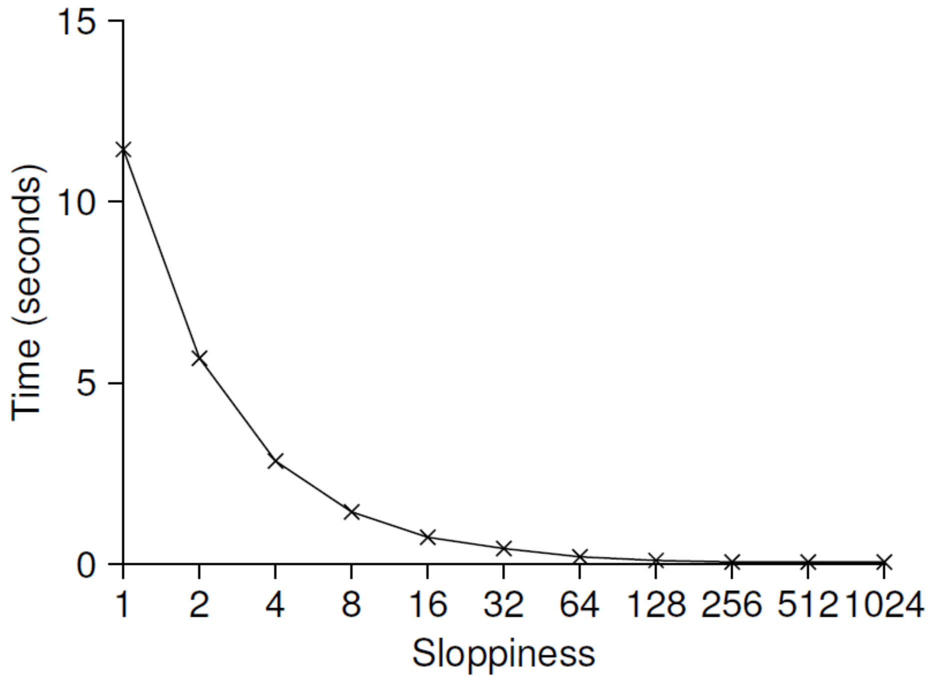
Sloppy counter example

- 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	L1	L2	L3	L4	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)
7	0	2	4	5 → 0	10 (from)

Importance of the threshold value S

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



Scaling Sloppy Counters

Sloppy Counter Implementation

```
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 (Cont.)

```
(Cont.)
    // 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!
    }
```

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);
}
```

Concurrent Linked Lists (Cont.)

```
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
}

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
}
```

Concurrent Linked Lists (Cont.)

- 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 List: Rewritten

```
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);
}

```

Concurrent Linked List: Rewritten (Cont.)

```
(Cont.)
    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

- 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***.

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**

Concurrent Queues (Cont.)

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}
```

Concurrent Queues (Cont.)

```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;
}
```

Concurrent Hash Table

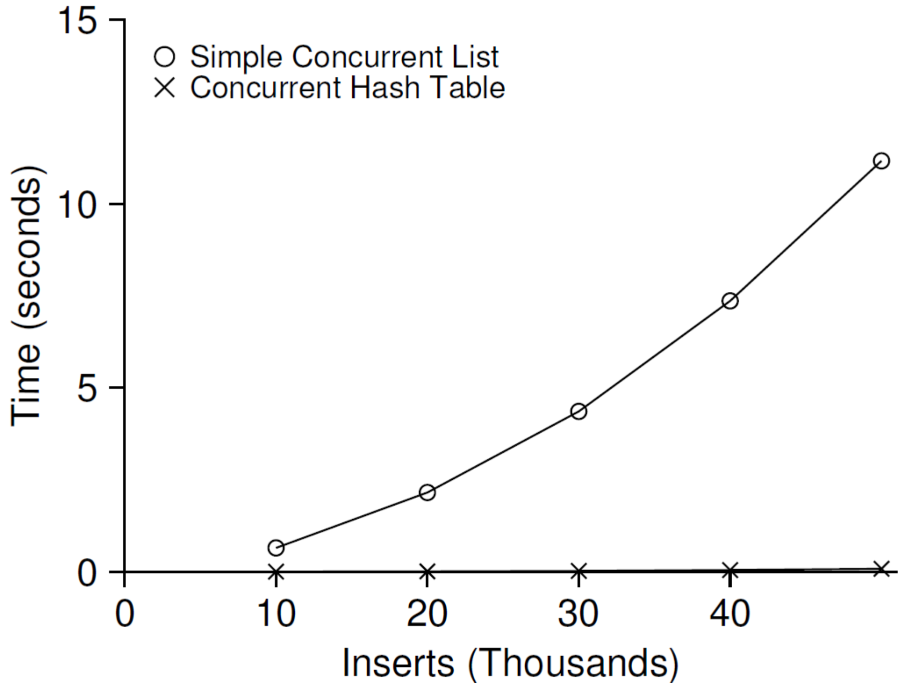
- Focus on a simple hash table
  - The hash table does not resize.
  - Built using the concurrent lists
  - It uses a **lock per hash bucket** each of which is represented by *a list*.

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);
}
```

Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates from each of four threads.



The simple concurrent hash table scales magnificently.