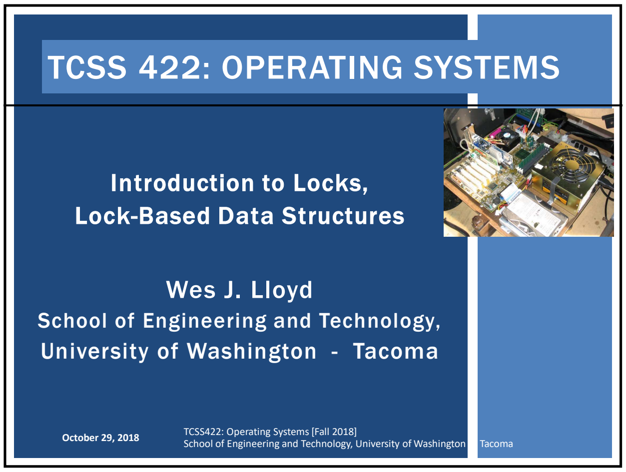
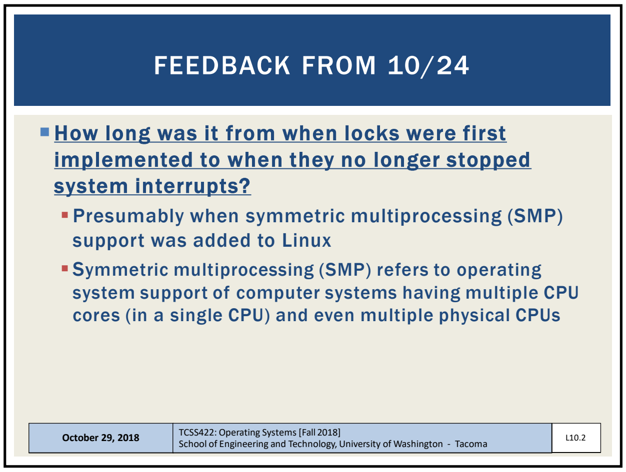


TCSS 422: OPERATING SYSTEMS

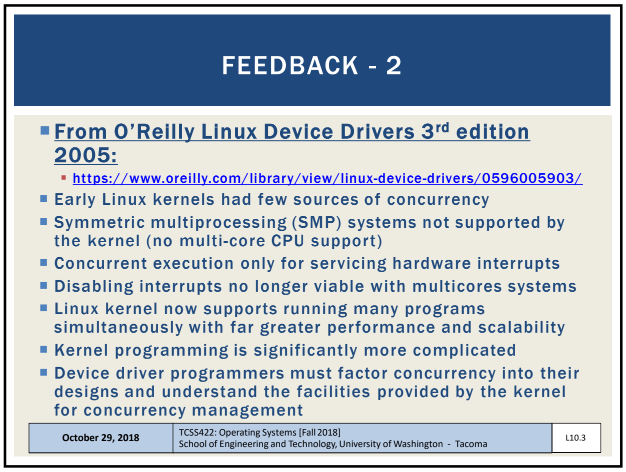
Introduction to Locks, Lock-Based Data Structures

Wes J. Lloyd
School of Engineering and Technology,
University of Washington - Tacoma



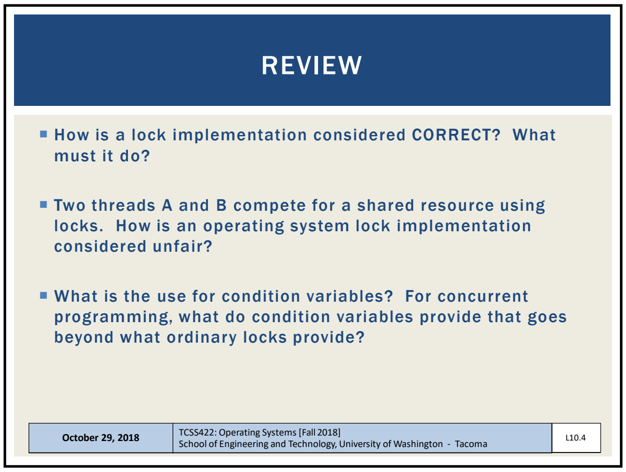
FEEDBACK FROM 10/24

- **How long was it from when locks were first implemented to when they no longer stopped system interrupts?**
  - Presumably when symmetric multiprocessing (SMP) support was added to Linux
  - Symmetric multiprocessing (SMP) refers to operating system support of computer systems having multiple CPU cores (in a single CPU) and even multiple physical CPUs



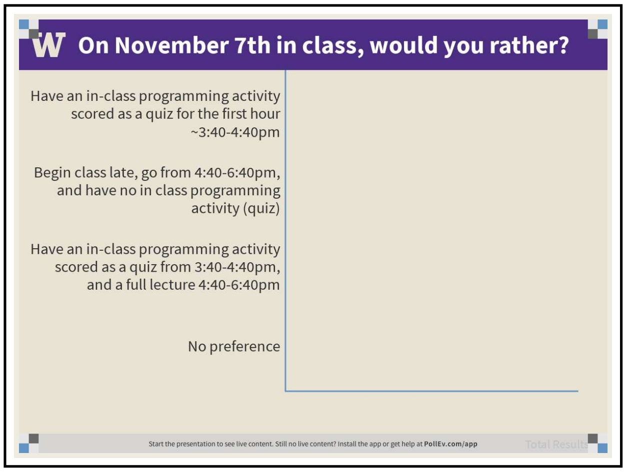
FEEDBACK - 2

- **From O'Reilly Linux Device Drivers 3rd edition 2005:**
  - https://www.oreilly.com/library/view/linux-device-drivers/0596005903/
- Early Linux kernels had few sources of concurrency
- Symmetric multiprocessing (SMP) systems not supported by the kernel (no multi-core CPU support)
- Concurrent execution only for servicing hardware interrupts
- Disabling interrupts no longer viable with multicores systems
- Linux kernel now supports running many programs simultaneously with far greater performance and scalability
- Kernel programming is significantly more complicated
- Device driver programmers must factor concurrency into their designs and understand the facilities provided by the kernel for concurrency management



REVIEW

- How is a lock implementation considered CORRECT? What must it do?
- Two threads A and B compete for a shared resource using locks. How is an operating system lock implementation considered unfair?
- What is the use for condition variables? For concurrent programming, what do condition variables provide that goes beyond what ordinary locks provide?



On November 7th in class, would you rather?

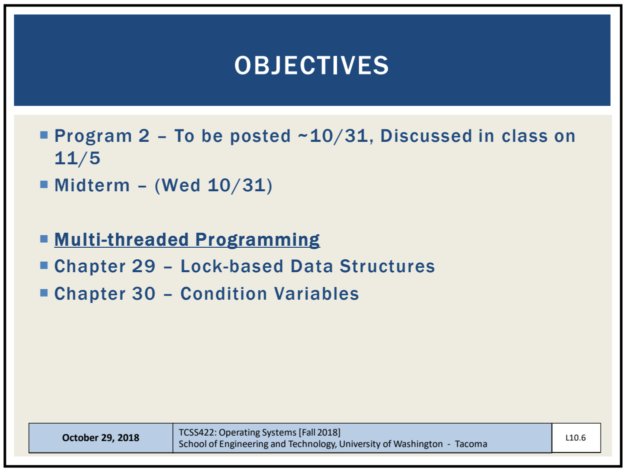
Have an in-class programming activity scored as a quiz for the first hour ~3:40-4:40pm

Begin class late, go from 4:40-6:40pm, and have no in class programming activity (quiz)

Have an in-class programming activity scored as a quiz from 3:40-4:40pm, and a full lecture 4:40-6:40pm

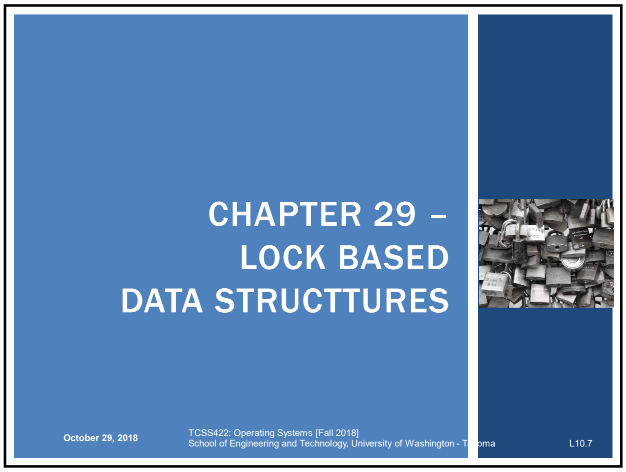
No preference

Start the presentation to see live content. Still no live content? Install the app or get help at PollEv.com/app

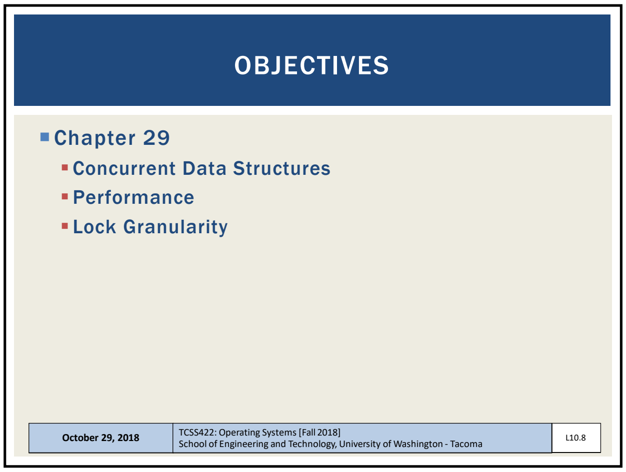


OBJECTIVES

- Program 2 – To be posted ~10/31, Discussed in class on 11/5
- Midterm – (Wed 10/31)
- **Multi-threaded Programming**
- Chapter 29 – Lock-based Data Structures
- Chapter 30 – Condition Variables

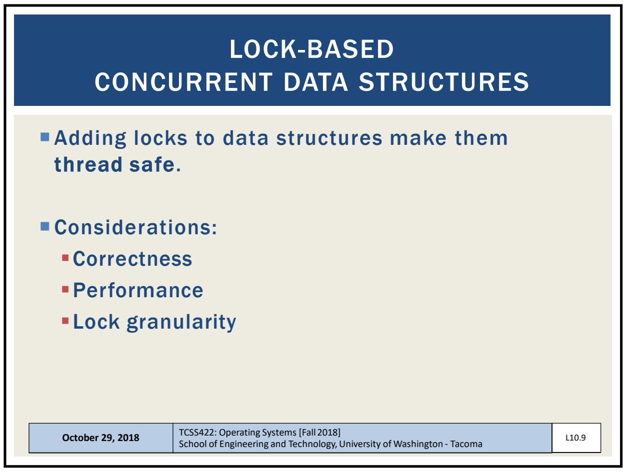


CHAPTER 29 – LOCK BASED DATA STRUCTTURES



OBJECTIVES

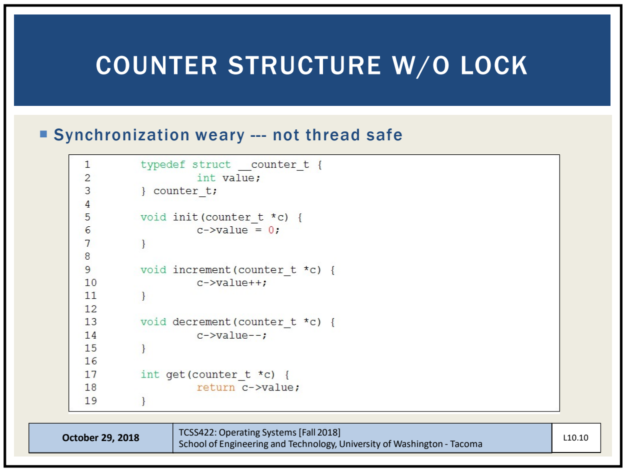
- Chapter 29
  - Concurrent Data Structures
  - Performance
  - Lock Granularity



LOCK-BASED CONCURRENT DATA STRUCTURES

- **Adding locks to data structures make them thread safe.**

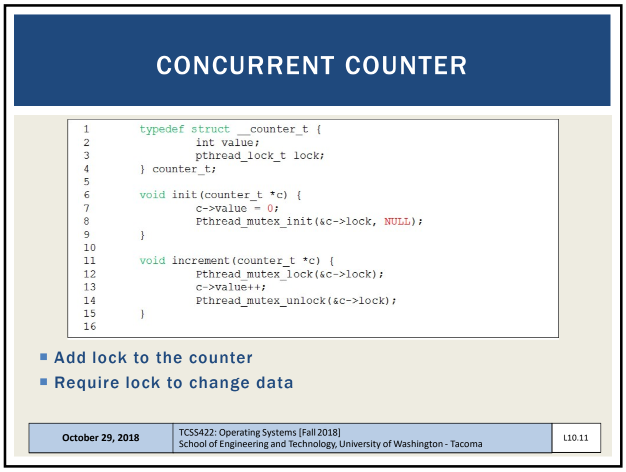
- Considerations:
  - Correctness
  - Performance
  - Lock granularity



COUNTER STRUCTURE W/O LOCK

- Synchronization weary --- not thread safe

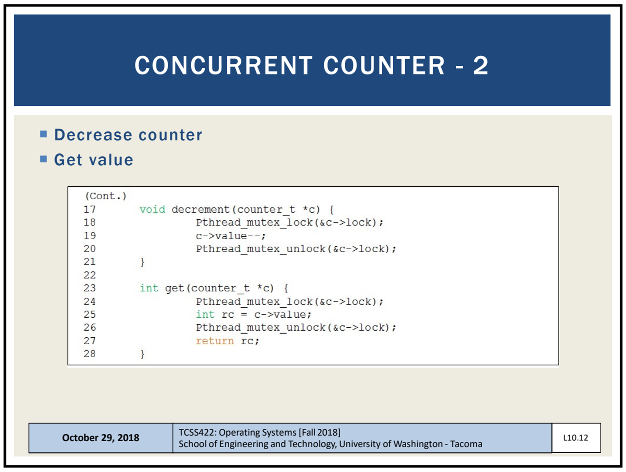
```
typedef struct __counter_t {
        int value;
} counter_t;

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;
}
```



CONCURRENT COUNTER

```
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);
}

```

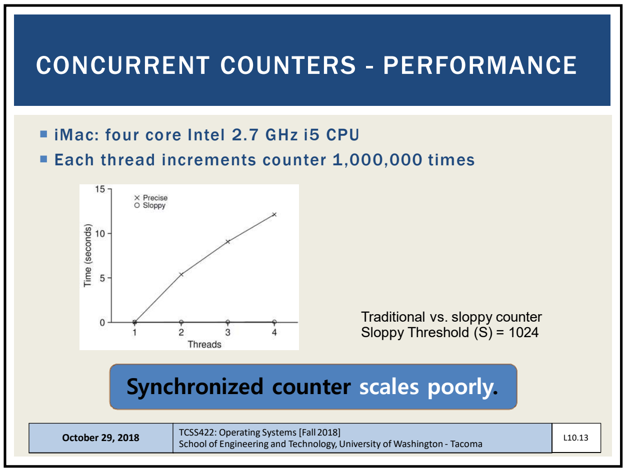
- Add lock to the counter
- Require lock to change data



CONCURRENT COUNTER - 2

- Decrease counter
- Get value

```
(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;
}
```

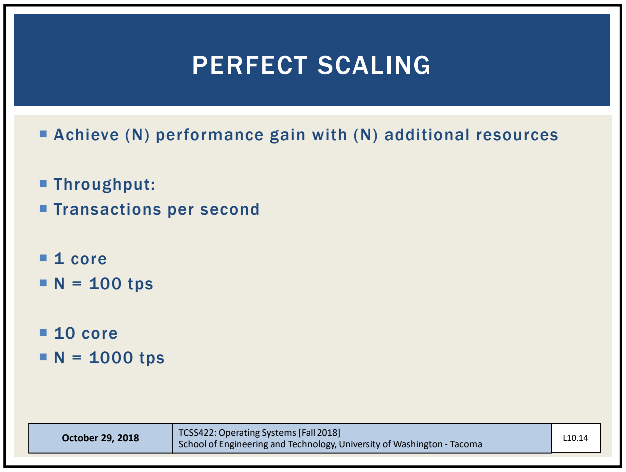


CONCURRENT COUNTERS - PERFORMANCE

- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times

Traditional vs. sloppy counter
Sloppy Threshold (S) = 1024

Synchronized counter scales poorly.



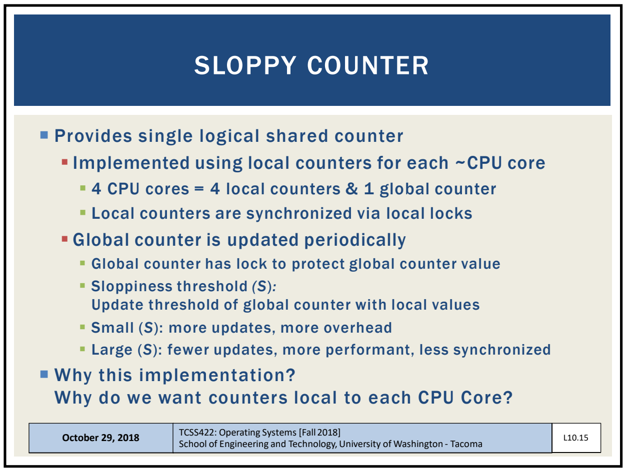
PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources

- Throughput:
- Transactions per second

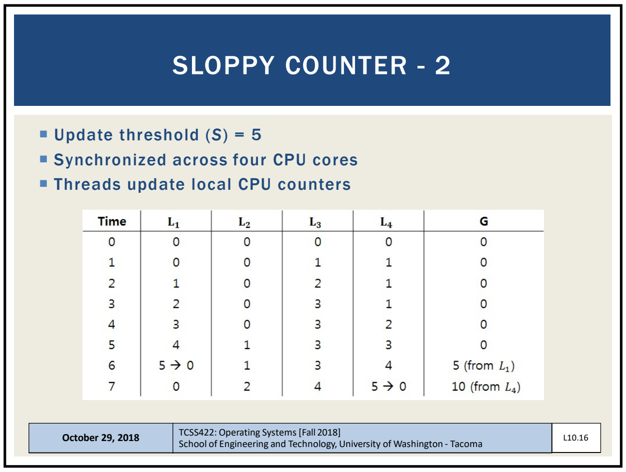
- 1 core
- N = 100 tps

- 10 core
- N = 1000 tps



SLOPPY COUNTER

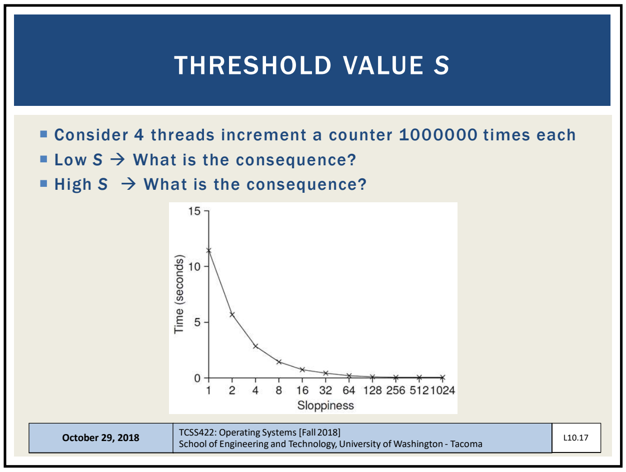
- Provides single logical shared counter
  - Implemented using local counters for each ~CPU core
    - 4 CPU cores = 4 local counters & 1 global counter
    - Local counters are synchronized via local locks
  - Global counter is updated periodically
    - Global counter has lock to protect global counter value
    - Sloppiness threshold (S):
      Update threshold of global counter with local values
    - Small (S): more updates, more overhead
    - Large (S): fewer updates, more performant, less synchronized
- Why this implementation?
  Why do we want counters local to each CPU Core?



SLOPPY COUNTER - 2

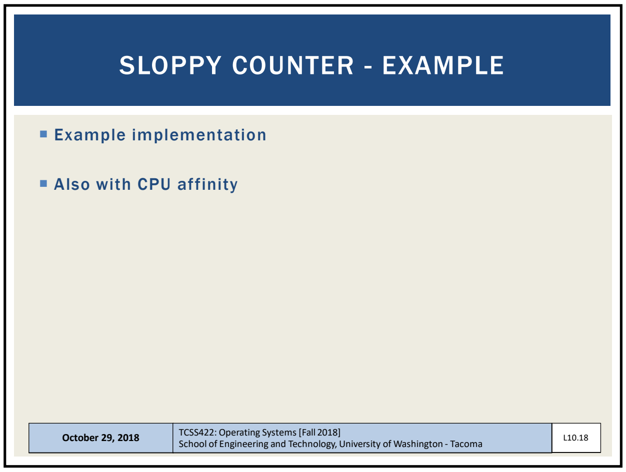
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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)



THRESHOLD VALUE S

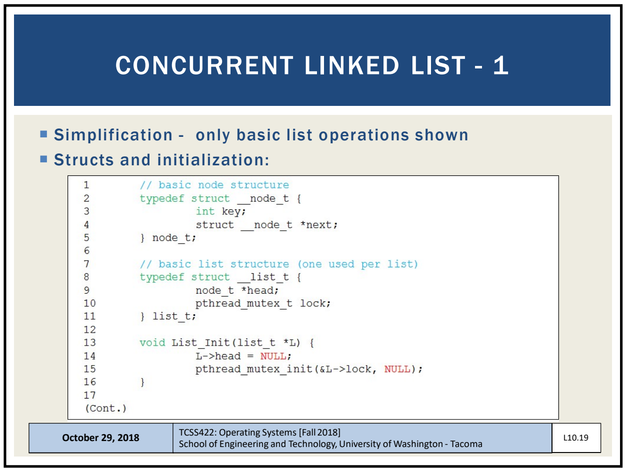
- Consider 4 threads increment a counter 1000000 times each
- Low S → What is the consequence?
- High S → What is the consequence?



SLOPPY COUNTER - EXAMPLE

- Example implementation

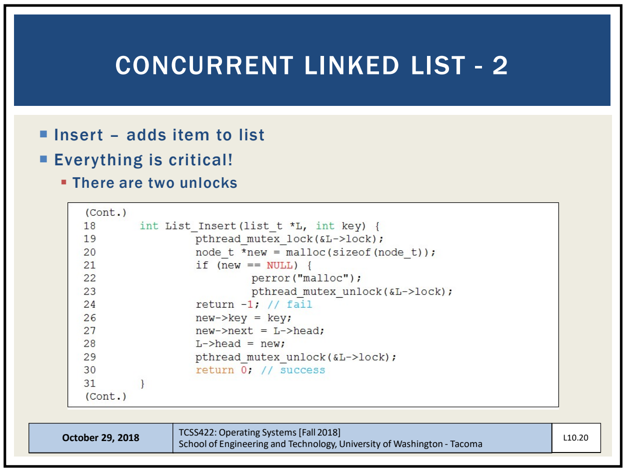
- Also with CPU affinity



CONCURRENT LINKED LIST - 1

- Simplification - only basic list operations shown
- Structs and initialization:

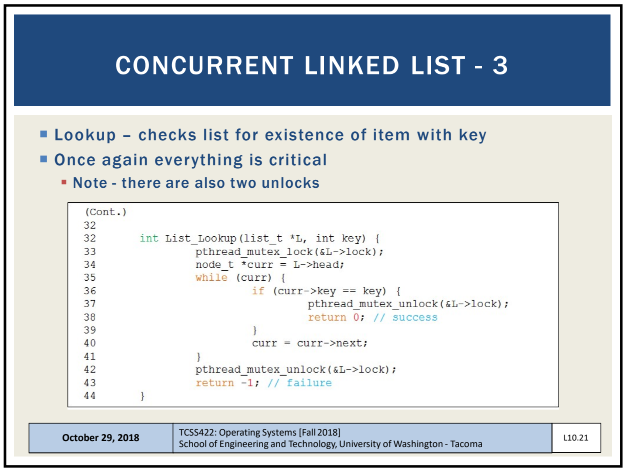
```
        // basic node structure
        typedef struct __node_t {
                int key;
                struct __node_t *next;
        } node_t;

        // 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);
        }

(Cont.)
```



CONCURRENT LINKED LIST - 2

- Insert – adds item to list
- Everything is critical!
  - There are two unlocks

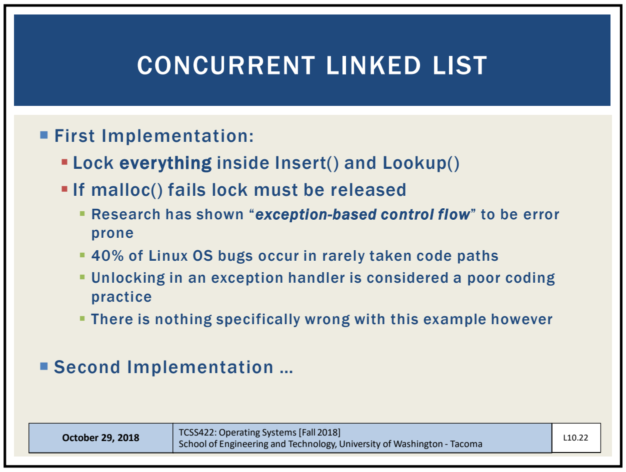
```
(Cont.)
        int List_Insert(list_t *L, int key) {
                pthread_mutex_lock(&L->lock);
                node_t *new = malloc(sizeof(node_t));
                if (new == NULL) {
                        perror("malloc");
                        pthread_mutex_unlock(&L->lock);
                return -1; // fail
                new->key = key;
                new->next = L->head;
                L->head = new;
                pthread_mutex_unlock(&L->lock);
                return 0; // success
        }
(Cont.)
```



CONCURRENT LINKED LIST - 3

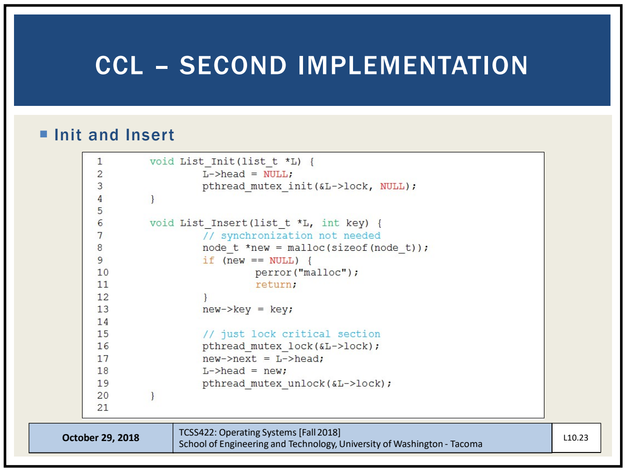
- Lookup – checks list for existence of item with key
- Once again everything is critical
  - Note - there are also two unlocks

```
(Cont.)

        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 LIST

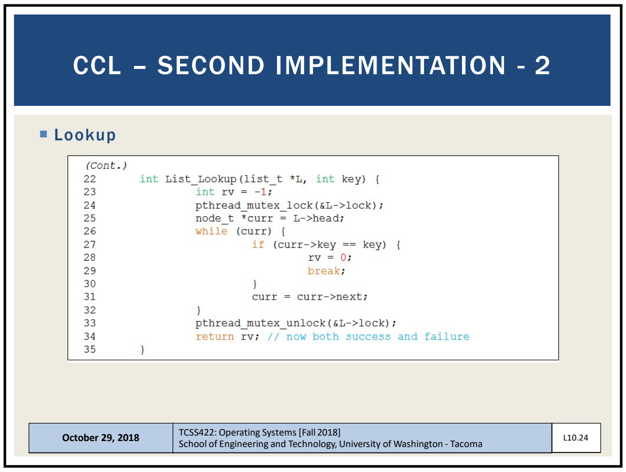
- First Implementation:
  - Lock **everything** inside Insert() and Lookup()
  - If malloc() fails lock must be released
    - Research has shown "***exception-based control flow***" to be error prone
    - 40% of Linux OS bugs occur in rarely taken code paths
    - Unlocking in an exception handler is considered a poor coding practice
    - There is nothing specifically wrong with this example however
- Second Implementation …



CCL – SECOND IMPLEMENTATION

- Init and Insert

```
        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);
        }

```



CCL – SECOND IMPLEMENTATION - 2

- Lookup

```
(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
        }
```

CONCURRENT LINKED LIST PERFORMANCE

- Using a single lock for entire list is not very performant
- Users must "wait" in line for a single lock to access/modify any item
- Hand-over-hand-locking (lock coupling)
  - Introduce a lock for each node of a list
  - Traversal involves handing over previous node's lock, acquiring the next node's lock…
  - Improves lock granularity
  - Degrades traversal performance
- Consider hybrid approach
  - Fewer locks, but more than 1
  - Best lock-to-node distribution?

MICHAEL AND SCOTT CONCURRENT QUEUES

- Improvement beyond a single master lock for a queue (FIFO)
- Two locks:
  - One for the **head** of the queue
  - One for the **tail**
- Synchronize enqueue and dequeue operations
- Add a dummy node
  - Allocated in the queue initialization routine
  - Supports separation of head and tail operations
- Items can be added and removed by separate threads at the same time

CONCURRENT QUEUE

- Remove from queue

```
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);
}

(Cont.)
```

CONCURRENT QUEUE - 2

- Add to queue

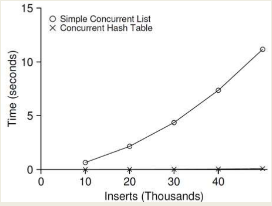
```
(Cont.)
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
}
(Cont.)
```

CONCURRENT HASH TABLE

- Consider a simple hash table
  - Fixed (static) size
  - Hash maps to a bucket
    - Bucket is implemented using a concurrent linked list
    - One lock per hash (bucket)
    - Hash bucket is a linked lists

INSERT PERFORMANCE – CONCURRENT HASH TABLE

- Four threads – 10,000 to 50,000 inserts
  - iMac with four-core Intel 2.7 GHz CPU



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);
}
```

LOCK-FREE DATA STRUCTURES

- Lock-free data structures in Java
- Java.util.concurrent.atomic package
- Classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicIntegerFieldUpdater
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
  - AtomicReference
- See: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html

QUESTIONS