


TCSS 422: OPERATING SYSTEMS

Introduction to Locks, Lock-Based Data Structures



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

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
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

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.2

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

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.3

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?

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.4

W 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 Total Results

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

October 29, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L10.6
------------------	---	-------

<h1>CHAPTER 29 – LOCK BASED DATA STRUCTURES</h1>		
October 29, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L10.7

<h2>OBJECTIVES</h2>		
<ul style="list-style-type: none">■ Chapter 29<ul style="list-style-type: none">■ Concurrent Data Structures■ Performance■ Lock Granularity		
October 29, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L10.8

LOCK-BASED CONCURRENT DATA STRUCTURES

- Adding locks to data structures make them thread safe.
- Considerations:
 - Correctness
 - Performance
 - Lock granularity

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.9

COUNTER STRUCTURE W/O LOCK

- Synchronization weary --- not thread safe

```
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    }
```

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.10

CONCURRENT COUNTER

```
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
```

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

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.11

CONCURRENT COUNTER - 2

- Decrease counter
- Get value

```
(Cont.)
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    }
```

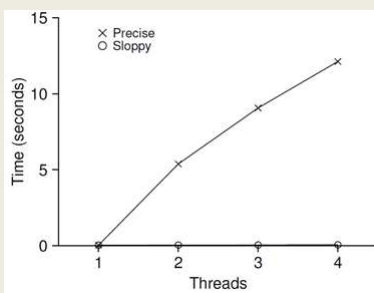
October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.12

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.

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.13

PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources
- Throughput:
 - Transactions per second
 - 1 core
 - N = 100 tps
 - 10 core
 - N = 1000 tps

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.14

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?

October 29, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L10.15
------------------	---	--------

SLOPPY COUNTER - 2

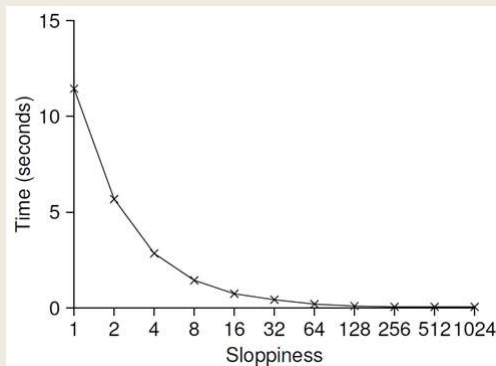
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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)

October 29, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L10.16
------------------	---	--------

THRESHOLD VALUE S

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



October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.17

SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.18

CONCURRENT LINKED LIST - 1

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

```
1 // basic node structure
2 typedef struct __node_t {
3     int key;
4     struct __node_t *next;
5 } node_t;
6
7 // basic list structure (one used per list)
8 typedef struct __list_t {
9     node_t *head;
10    pthread_mutex_t lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 (Cont.)
```

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.19

CONCURRENT LINKED LIST - 2

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

```
(Cont.)
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key = key;
27     new->next = L->head;
28     L->head = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 (Cont.)
```

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.20

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.)
32
32     int List_Lookup(list_t *L, int key) {
33         pthread_mutex_lock(&L->lock);
34         node_t *curr = L->head;
35         while (curr) {
36             if (curr->key == key) {
37                 pthread_mutex_unlock(&L->lock);
38                 return 0; // success
39             }
40             curr = curr->next;
41         }
42         pthread_mutex_unlock(&L->lock);
43         return -1; // failure
44     }
```

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.21

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 ...

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.22

CCL – SECOND IMPLEMENTATION

■ Init and Insert

```
1 void List_Init(list_t *L) {
2     L->head = NULL;
3     pthread_mutex_init(&L->lock, NULL);
4 }
5
6 void List_Insert(list_t *L, int key) {
7     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10        perror("malloc");
11        return;
12    }
13    new->key = key;
14
15    // just lock critical section
16    pthread_mutex_lock(&L->lock);
17    new->next = L->head;
18    L->head = new;
19    pthread_mutex_unlock(&L->lock);
20 }
21
```

October 29, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.23

CCL – SECOND IMPLEMENTATION - 2

■ Lookup

```
(Cont.)
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
```

October 29, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.24

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?



October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.25

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

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.26

CONCURRENT QUEUE

Remove from queue

```
1  typedef struct __node_t {
2      int value;
3      struct __node_t *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t *head;
8      node_t *tail;
9      pthread_mutex_t headLock;
10     pthread_mutex_t tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
(Cont.)
```

October 29, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.27

CONCURRENT QUEUE - 2

Add to queue

```
(Cont.)
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24
25     tmp->value = value;
26     tmp->next = NULL;
27
28     pthread_mutex_lock(&q->tailLock);
29     q->tail->next = tmp;
30     q->tail = tmp;
31     pthread_mutex_unlock(&q->tailLock);
32 }
(Cont.)
```

October 29, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.28

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

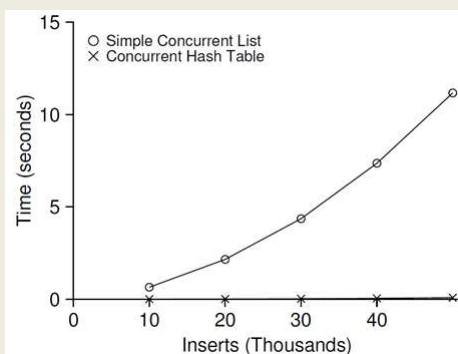
October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.29

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.**

October 29, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.30

CONCURRENT HASH TABLE

```
1      #define BUCKETS (101)
2
3      typedef struct __hash_t {
4          list_t lists[BUCKETS];
5      } hash_t;
6
7      void Hash_Init(hash_t *H) {
8          int i;
9          for (i = 0; i < BUCKETS; i++) {
10             List_Init(&H->lists[i]);
11         }
12     }
13
14     int Hash_Insert(hash_t *H, int key) {
15         int bucket = key % BUCKETS;
16         return List_Insert(&H->lists[bucket], key);
17     }
18
19     int Hash_Lookup(hash_t *H, int key) {
20         int bucket = key % BUCKETS;
21         return List_Lookup(&H->lists[bucket], key);
22     }
```

October 29, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.31

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>

October 29, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L10.32

