


TCSS 422: OPERATING SYSTEMS

Lock Based Data Structures,
Condition Variables



Wes J. Lloyd

Institute of Technology

University of Washington - Tacoma

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Quiz 2 Review
- Tutorial 1 Questions
- Homework 1 Questions
- Feedback from 1/31
- Ch. 29
 - Lock Based Data Structures
- Ch. 30 (start)
 - Condition Variables
- Practice midterm

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.2

FEEDBACK FROM 1/31

- For lock implementation, do we want:
 - Correctness, fairness, OR performance
 - Correctness, fairness, AND performance
- Do we want at least two of the three for a good solution?
- Evaluation criteria apply to both:
 - Implementation of locks within a language (e.g. C)
 - Implementation of locking within a user program
- **Correctness:** locks **must be** correct to be usable.
Must avoid deadlock, race conditions
- Performance and fairness are never perfect
- Best solutions are correct, while offering some balance of performance and fairness.
 - Performance and fairness aren't directly related

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.3

FEEDBACK - 2


- How does lock granularity impact correctness, fairness, and/or performance of user programs?
- Fine grained locking increasing program complexity leading to greater potential for race conditions, dead lock from programmer error
- Fine grained locking potentially decreases performance by **increasing overhead** for obtaining a large number of locks
 - Coarse grained locking results in more blocked threads, less parallelism, and slower program performance
- With fine grained locking, there should be **less competition** for each individual lock, making fairness simpler to provide
 - Coarse grained locks increases competition for each lock
 - More opportunities for lock starvation

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.4

CHAPTER 29 –
LOCK BASED
DATA STRUCTURES



February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.5

LOCK-BASED
CONCURRENT DATA STRUCTURES

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

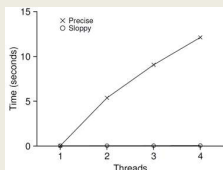
February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.6

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.

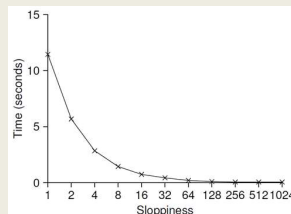
February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.7

SLOPPY COUNTER - THRESHOLD VALUE S

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



February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.8

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.)
    
```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.9

CONCURRENT LINKED LIST - 2

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

```

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.)
    
```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.10

CONCURRENT LINKED LIST - 3

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

```

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 }
    
```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.11

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 ...

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.12

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

```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.13

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 }

```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.14

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?



February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.15

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

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.16

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.)

```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.17

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     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
(Cont.)

```

February 5, 2018

TCCS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.18

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

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.19

INSERT PERFORMANCE – CONCURRENT HASH TABLE

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

Inserts (Thousands)	Simple Concurrent List (seconds)	Concurrent Hash Table (seconds)
10	~1.0	~0.5
20	~2.5	~0.8
30	~5.0	~1.0
40	~10.0	~1.2

The simple concurrent hash table scales magnificently.

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.20

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 }
```

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.21

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>

February 5, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L9.22

QUESTIONS