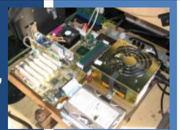
TCSS 422: OPERATING SYSTEMS



Lock-based data structures, Midterm Review

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

February 17, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma

1

OBJECTIVES – 2/17

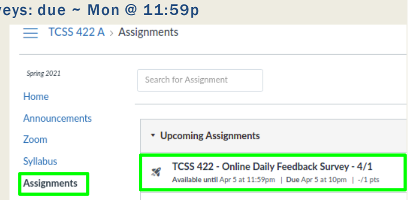
- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- Assignment 1 - Due Tue Feb 17 AOE
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

February 17, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L11.2

2

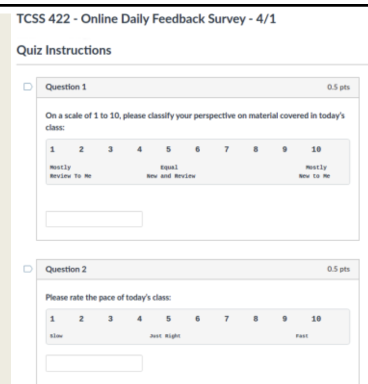
ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p



February 17, 2026 | TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L11.3

3



February 17, 2026 | TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L11.4

4

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (37 of 46 respondents (8 online) – 80.4%):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.81 (↑ - previous 6.76)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.24 (↑ - previous 5.09)**

February 17, 2026 | TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L11.5

5

FEEDBACK FROM 2/10

- ***What does the term 'mutually exclusive' mean relative to locking ?***
- Adding a **lock** around a **critical section** of code makes the code **mutually exclusive**, so only one thread holding the lock can execute the critical section at any given time
  - Makes critical section 'sequential' in that only one thread in the whole program can run it at any given time
- ***What is the point of having a lock call inside a critical section protected by another lock call ?***
  - This is nested locking
  - Nested locks are used if a critical section modifies 2 separate variables (A and B) protected with distinct locks (lock_A, lock_B) at the same
  - With separate locks (lock_A, lock_B) these variables can be modified separately elsewhere in the code
  - The two locks (lock_A, lock_B) could be replaced by a single lock (lock_AB), but this may reduce parallelism in the program – any change to A requires lock_AB, and any change to B requires lock_AB

February 17, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L11.7

7

FEEDBACK - 2

- ***How does fairness work in locks?***
- ***WITH LOCKS***
  - With pthread_mutex_lock() and pthread_mutex_unlock(), the thread that receives the lock first is the thread scheduled next to run by the OS scheduler
  - The programmer has no control over this
  - If you repeat the same unlock many times, the distribution may appear random, or potentially unfavorable
- ***WITH CONDITION VARIABLES***
  - Using condition variables the programmer can explicitly control which thread(s) receive the lock
    - pthread_cond_signal(): The thread waiting the longest on a FIFO queue receives the lock
    - Pthread_cond_broadcast(): all threads are awoken. A state variable is checked. The thread with the desired state (i.e. THREADID=7) runs

8

FEEDBACK - 3

- ***When one of the threads waiting in the FIFO wait queue for the signal wakes up, does the thread get the lock (mutex) right away ?***
- YES
- If the state variable is not satisfied the while loop will call pthread_cond_wait() again, which implicitly releases the lock (i.e. pthread_mutex_unlock)

9

OBJECTIVES – 2/17

- Questions from 2/10
- **C Tutorial - Pointers, Strings, Exec in C - Closed**
- Assignment 1 - Due Tue Feb 17 AOE
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

10

OBJECTIVES – 2/17

- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- **Assignment 1 - Due Tue Feb 17 AOE**
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

11

OBJECTIVES – 2/17

- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- Assignment 1 - Due Tue Feb 17 AOE
- **Tutorial 2 - Pthread Tutorial (TO BE POSTED)**
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

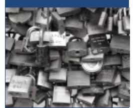
12

CATCH UP FROM LECTURE 10

- Switch to Lecture 10 Slides
- Slides L10.31 to L10.46 (Chapter 28 –Locks)

13

CHAPTER 29 – LOCK BASED DATA STRUCTURES



14

LOCK-BASED CONCURRENT DATA STRUCTURES

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

15

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;
}
```


16

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

17

CONCURRENT COUNTER - 2

- Decrease counter
- Get value

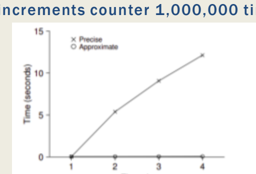
```
(Cont.)
void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
}
```


18

CONCURRENT COUNTERS - PERFORMANCE

- Concurrent counter is considered a "precise counter"
- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times



Precise counter scales poorly.

19

PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources
- Throughput:
- Transactions per second (tps)
- 1 core
- N = 100 tps
- 10 cores (x10)
- N = 1000 tps (x10)
- ***Is parallel counting with a shared counter an embarrassingly parallel problem?***

20

OBJECTIVES – 2/17

- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- Assignment 1 - Due Tue Feb 17 AOE
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

21

APPROXIMATE (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
    - Update threshold (*S*) – *referred to as sloppiness threshold:* How often to push local values to global counter
    - 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?

22

APPROXIMATE COUNTER – MAIN POINTS

- Idea of the Approximate Counter is to ***RELAX*** the synchronization requirement for counting
  - Instead of synchronizing global count variable each time: `counter=counter+1`
  - Synchronization occurs only every so often: e.g. *every 1000 counts*
- Relaxing the synchronization requirement ***drastically*** reduces locking API overhead by trading-off split-second accuracy of the counter
- Approximate counter: trade-off accuracy for speed
  - It's approximate because it's not so accurate (until the end)

23

APPROXIMATE COUNTER - 2

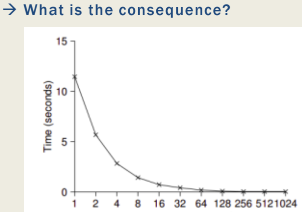
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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)

24

THRESHOLD VALUE S

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



25

APPROXIMATE COUNTER - EXAMPLE

- Example implementation – sloppybasic.c
- Also with CPU affinity

26

When poll is active, respond at **pollev.com/wesleylloyd641**
Text **WESLEYLLOYD641** to **22333** once to join

Which of the following is NOT a problem as a result of having a low S-value for the approximate counter (Sloppy Counter) threshold?

- The counter overhead is very high.
- The counter implementation performs a very large number of LOCK/UNLOCK API calls.
- The global counter value is highly accurate.
- The counter performs very few local to global counter updates.
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

27

OBJECTIVES – 2/17

- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- Assignment 1 - Due Tue Feb 17 AOE
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - **Concurrent Structures: Linked List**, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

28

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.)
```


29

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.)
```


30

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
}
```


31

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 …

32

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);
}
```


33

CCL – SECOND IMPLEMENTATION - 2

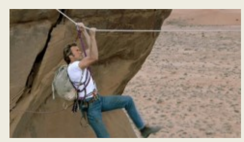
- Lookup

```
(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
}
```


34

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?



35

OBJECTIVES – 2/17

- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- Assignment 1 - Due Tue Feb 17 AOE
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, **Queue**, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

36

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

37

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.)
```


38

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.)
```


39

OBJECTIVES – 2/17

- Questions from 2/10
- C Tutorial - Pointers, Strings, Exec in C - Closed
- Assignment 1 - Due Tue Feb 17 AOE
- Tutorial 2 - Pthread Tutorial (TO BE POSTED)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions

40

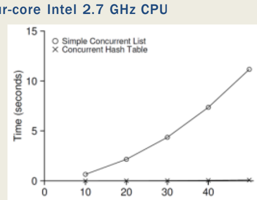
CONCURRENT HASH TABLE

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

41

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.

42

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);
}
```


43

Which is a major advantage of using concurrent data structures in your programs?

Locks are encapsulated within data structure code ensuring thread safety.

Lock granularity tradeoff already optimized inside data structurew

Multiple threads can more easily share data

All of the above

None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

44

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/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

February 17, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L11.45

45

QUESTIONS

78