TCSS 422: OPERATING SYSTEMS

Linux Thread API, Lock Implementations, Lock-based data structures,

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

1

TCSS 422 – OFFICE HRS – WINTER 2026

- <u>**Office Hours plan for Winter:**</u>
- **Tuesday 2:30 - 3:30 pm Instructor Wes, Zoom**
- **Tue/Thur 6:00 - 7:00 pm Instructor Wes, CP 229/Zoom**
- **Tue 6:00 – 7:00 pm GTA Robert, Zoom/MDS 302**
- **Wed 1:00 – 2:00 pm GTA Robert, Zoom/MDS 302**

- Instructor is available after class at 6pm in CP 229 each day

2

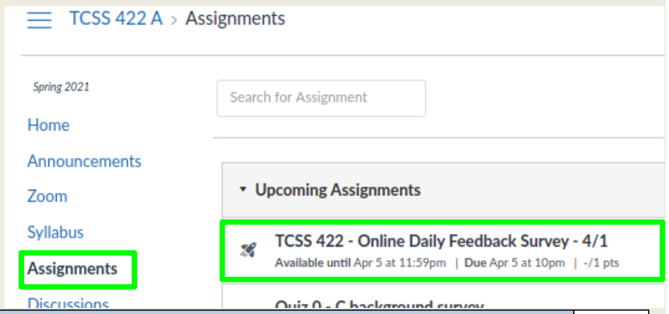
OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

3

ONLINE FEEDBACK SURVEY

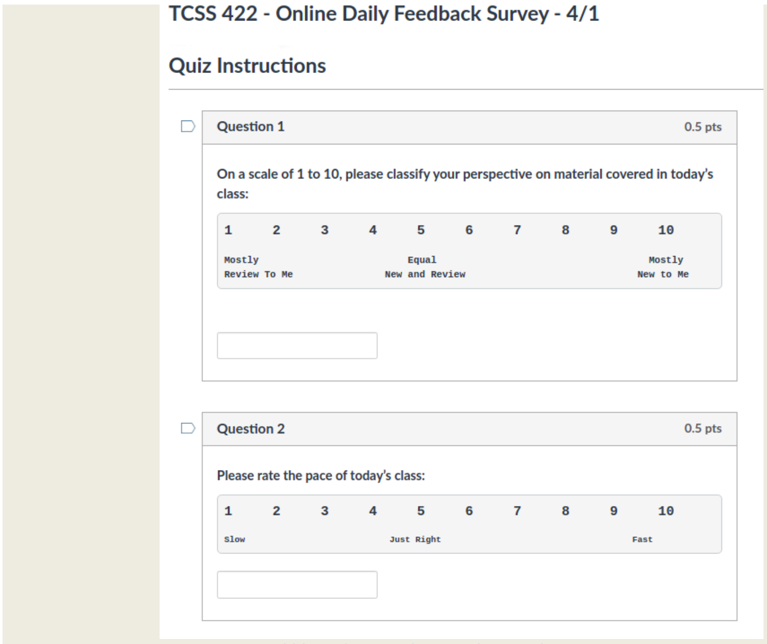
- Feedback Survey "Quiz" in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p



TCSS 422 - Online Daily Feedback Survey - 4/1
Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts

Quiz 0 - C background survey

4



TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 0.5 pts

On a scale of 1 to 10, please classify your perspective on material covered in today's class:

1	2	3	4	5	6	7	8	9	10
Mostly Review To Me				Equal New and Review					Mostly New to Me

Question 2 0.5 pts

Please rate the pace of today's class:

1	2	3	4	5	6	7	8	9	10
Slow				Just Right					Fast

February 10, 2026 TCSS422: Computer Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma L10.5

5

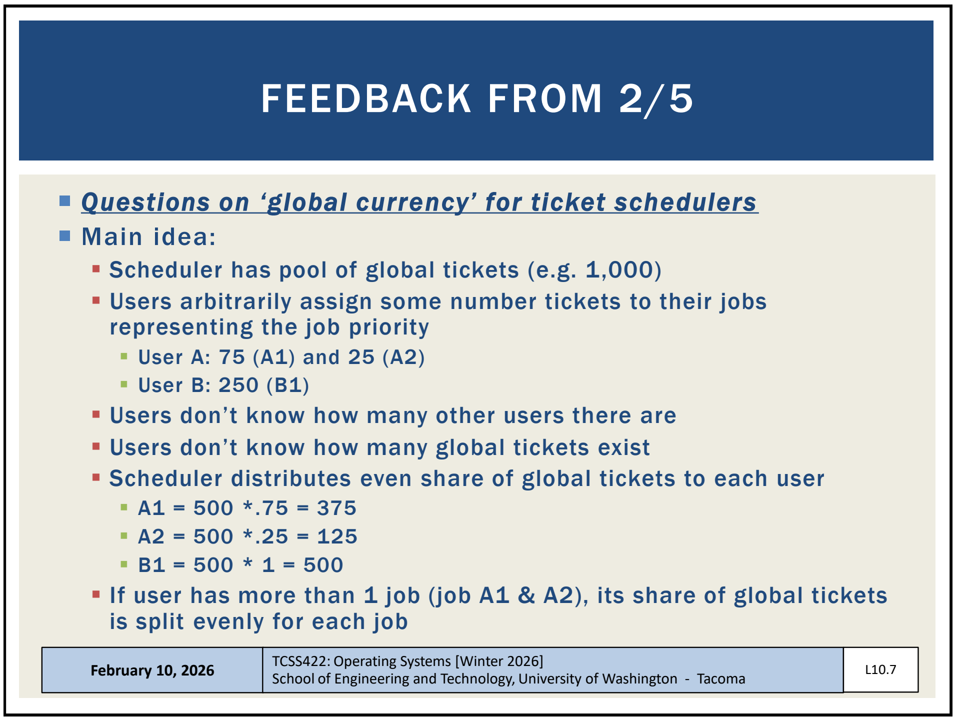
MATERIAL / PACE

- Please classify your perspective on material covered in today's class (34 of 46 respondents (8 online) – 73.9%):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.76 (↑ - previous 5.92)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.09 (↑ - previous 5.03)**

February 10, 2026	TCSS422: Computer Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L10.6

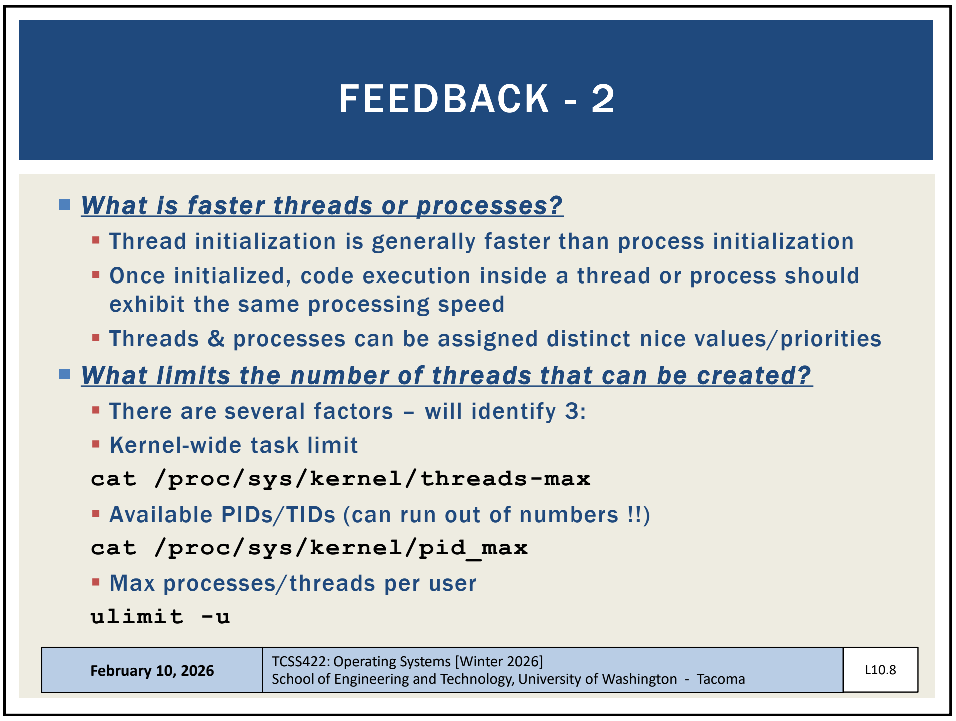
6



FEEDBACK FROM 2/5

- ***Questions on 'global currency' for ticket schedulers***
- Main idea:
  - Scheduler has pool of global tickets (e.g. 1,000)
  - Users arbitrarily assign some number tickets to their jobs representing the job priority
    - User A: 75 (A1) and 25 (A2)
    - User B: 250 (B1)
  - Users don't know how many other users there are
  - Users don't know how many global tickets exist
  - Scheduler distributes even share of global tickets to each user
    - A1 = 500 *.75 = 375
    - A2 = 500 *.25 = 125
    - B1 = 500 * 1 = 500
  - If user has more than 1 job (job A1 & A2), its share of global tickets is split evenly for each job

7



FEEDBACK - 2

- ***What is faster threads or processes?***
  - Thread initialization is generally faster than process initialization
  - Once initialized, code execution inside a thread or process should exhibit the same processing speed
  - Threads & processes can be assigned distinct nice values/priorities
- ***What limits the number of threads that can be created?***
  - There are several factors – will identify 3:
  - Kernel-wide task limit

```
cat /proc/sys/kernel/threads-max
```

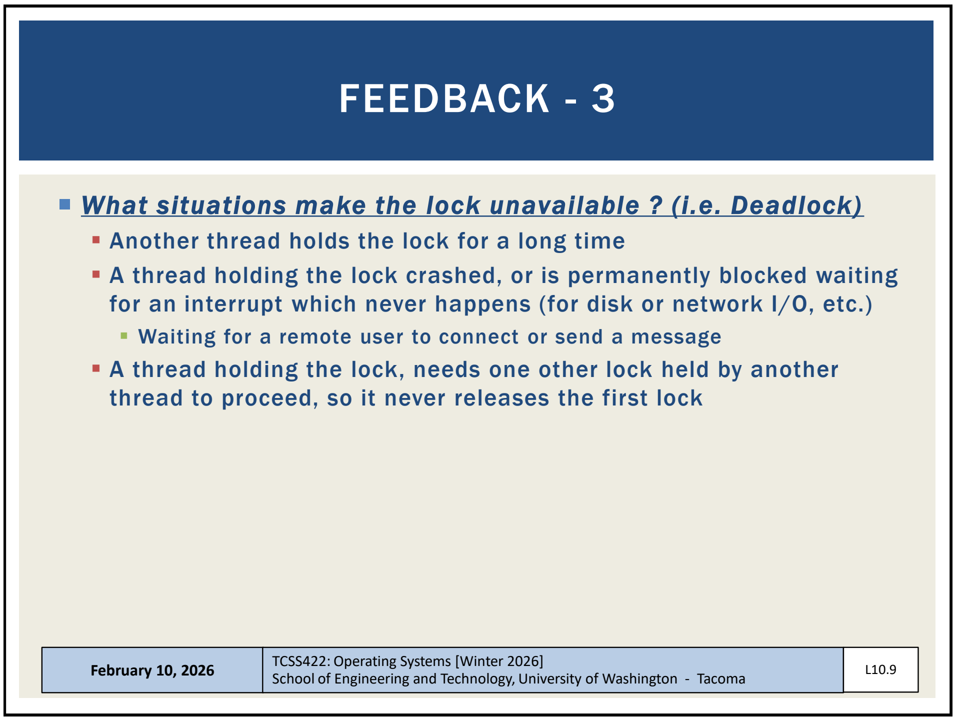
  - Available PIDs/TIDs (can run out of numbers !!)

```
cat /proc/sys/kernel/pid_max
```

  - Max processes/threads per user

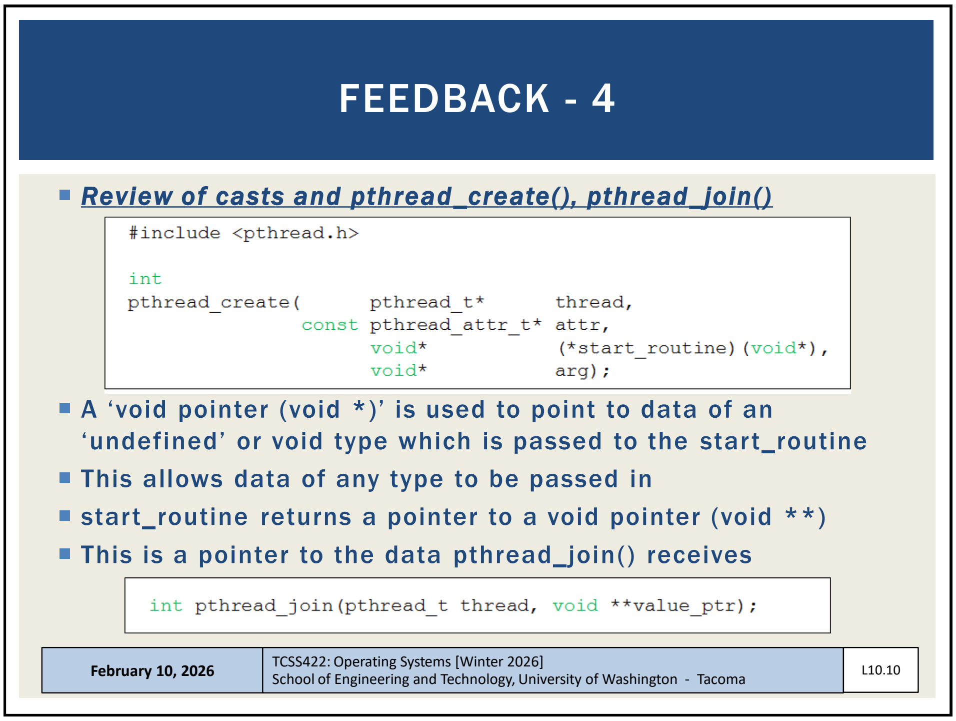
```
ulimit -u
```

8



FEEDBACK - 3

- ***What situations make the lock unavailable ? (i.e. Deadlock)***
  - Another thread holds the lock for a long time
  - A thread holding the lock crashed, or is permanently blocked waiting for an interrupt which never happens (for disk or network I/O, etc.)
    - Waiting for a remote user to connect or send a message
  - A thread holding the lock, needs one other lock held by another thread to proceed, so it never releases the first lock



FEEDBACK - 4

- ***Review of casts and pthread_create(), pthread_join()***

```
#include <pthread.h>

int
pthread_create(      pthread_t*       thread,
               const pthread_attr_t* attr,
                     void*            (*start_routine)(void*),
                     void*            arg);
```

- A ‘void pointer (void *)’ is used to point to data of an ‘undefined’ or void type which is passed to the start_routine
- This allows data of any type to be passed in
- start_routine returns a pointer to a void pointer (void **)
- This is a pointer to the data pthread_join() receives

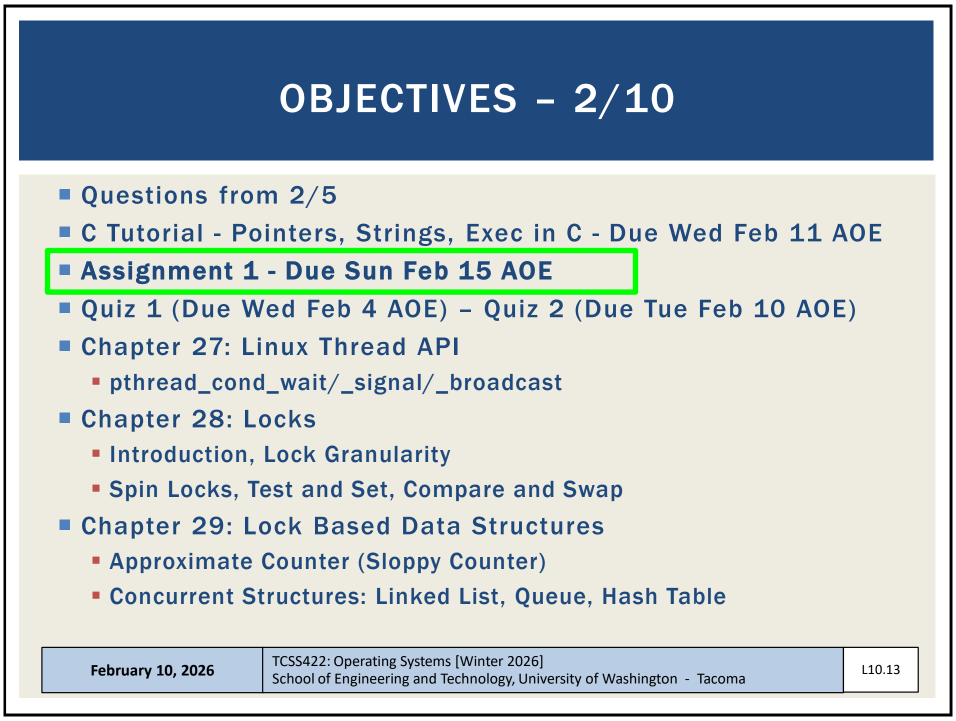
```
int pthread_join(pthread_t thread, void **value_ptr);
```

FEEDBACK - 5

- See pthread_cast.c example (chapter 26)

OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

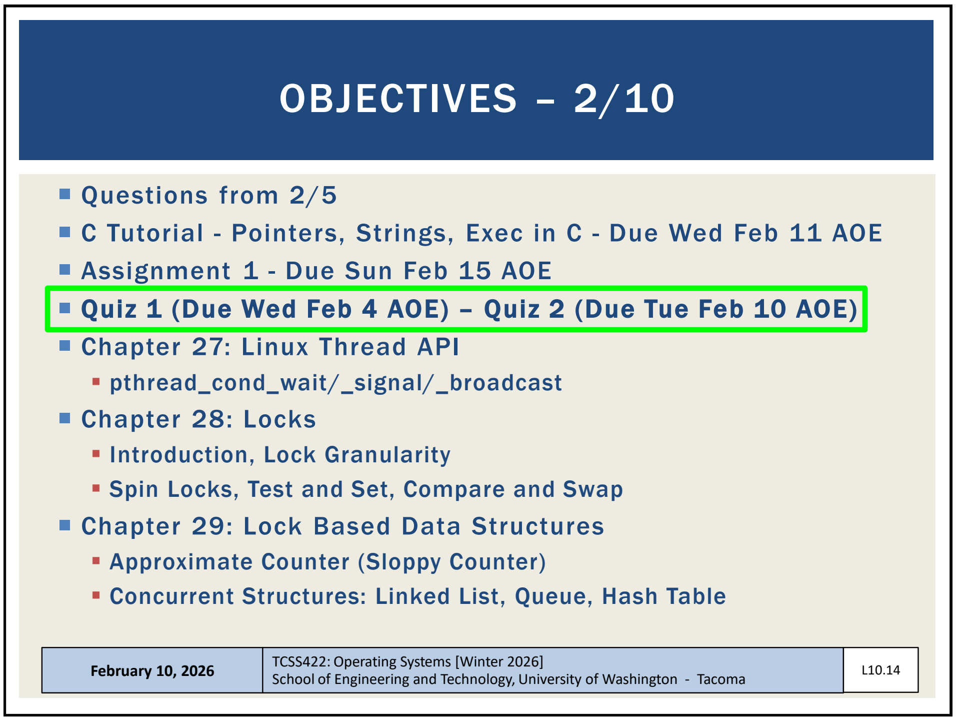


OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- **Assignment 1 - Due Sun Feb 15 AOE**
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

February 10, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L10.13

13



OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- **Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)**
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

February 10, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L10.14

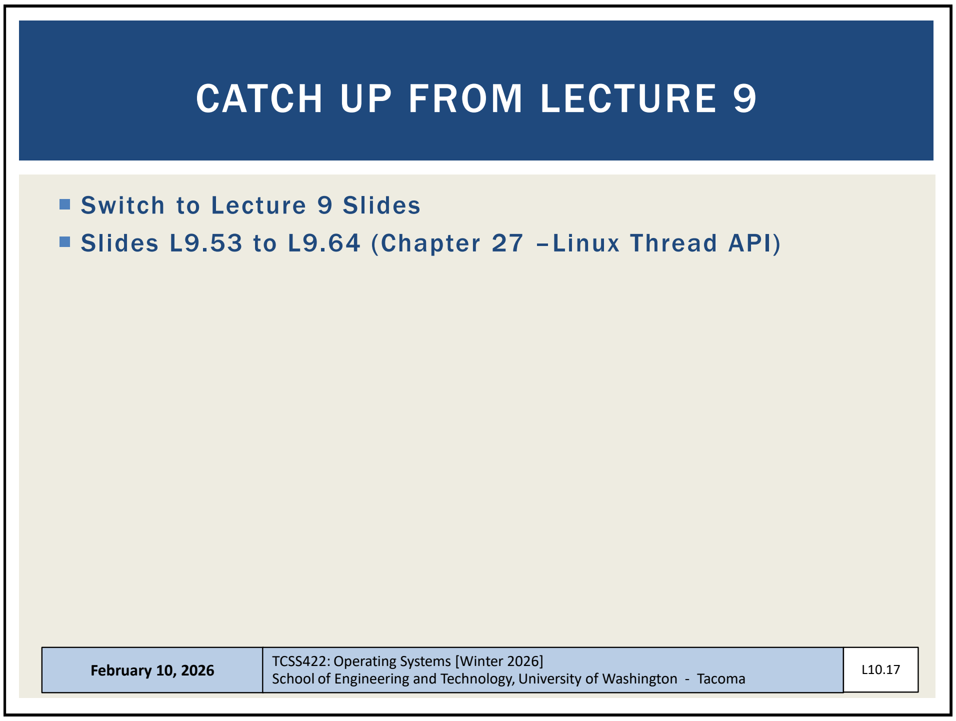
14

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Wednesday Feb 4th AOE
- Link:
- https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_w2026_quiz_1.pdf

QUIZ 2

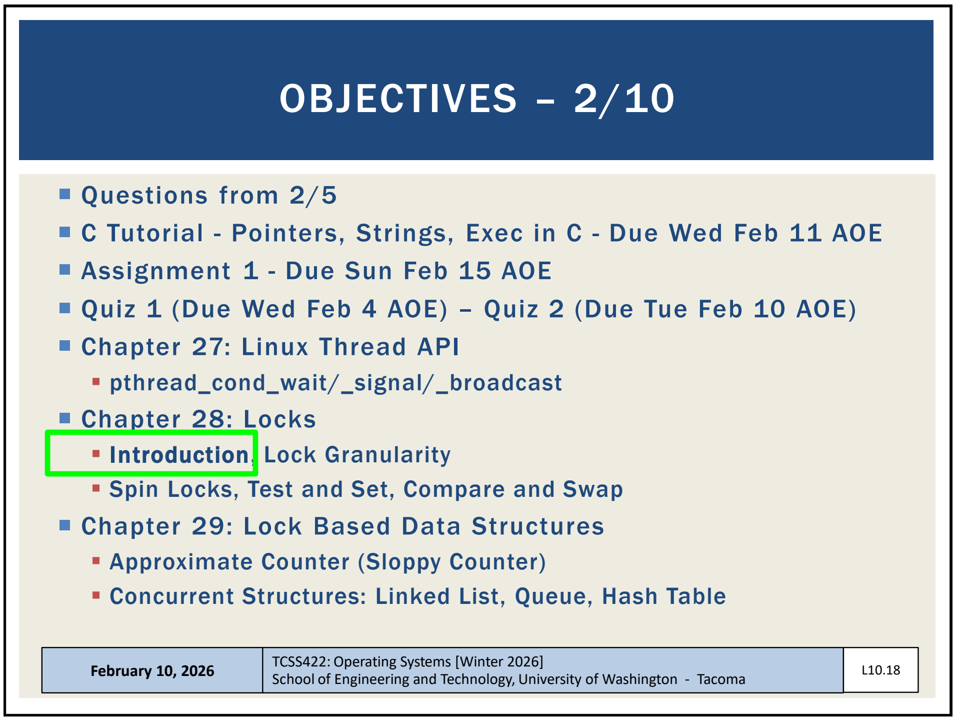
- Canvas Quiz – Practice CPU Scheduling Problems
- Posted in Canvas
- Unlimited attempts permitted
- Provides CPU scheduling practice problems
  - FIFO, SJF, STCF, RR, MLFQ (Ch. 7 & 8)
- Multiple choice and fill-in the blank
- Quiz automatically scored by Canvas
  - Please report any grading problems
- Due Tuesday Feb 10th AOE (Feb 11th at 4:59am)
- Link:
- https://canvas.uw.edu/courses/1871290/assignments/11129208



CATCH UP FROM LECTURE 9

- Switch to Lecture 9 Slides
- Slides L9.53 to L9.64 (Chapter 27 –Linux Thread API)

17



OBJECTIVES – 2/10

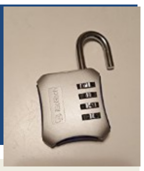
- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - **Introduction**, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

18

CHAPTER 28 – LOCKS

19

LOCKS



- Ensure critical section(s) are executed atomically-*as a unit*
  - Only one thread is allowed to execute a critical section at any given time
  - Ensures the code snippets are "mutually exclusive"
- Protect a global counter:

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

- A "critical section":

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

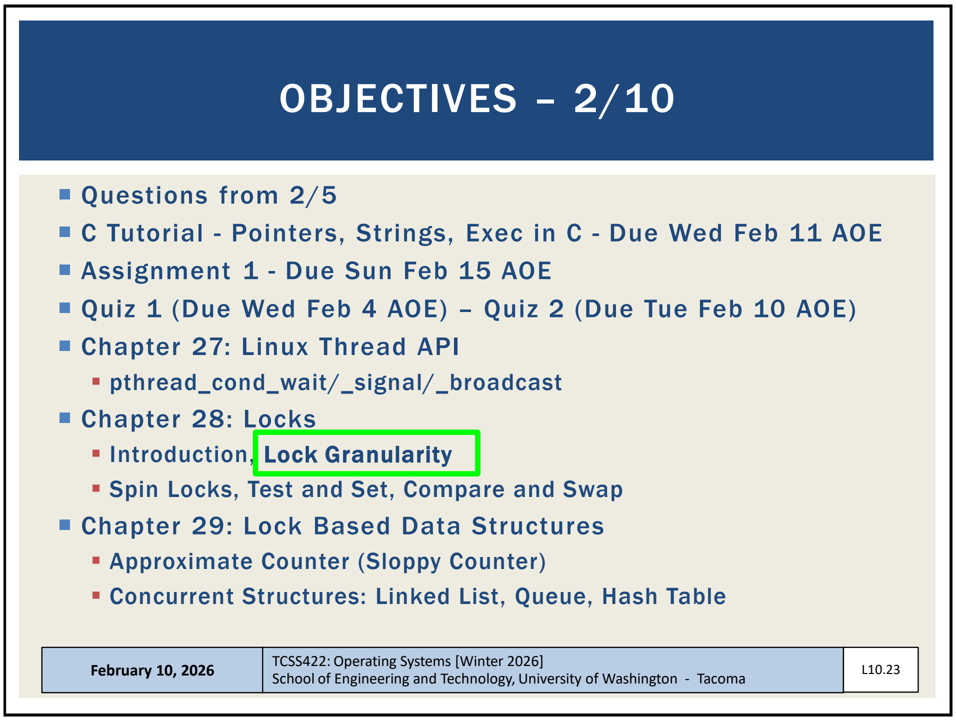
20

LOCKS - 2

- Lock variables are called "MUTEX"
  - Short for mutual exclusion (that's what they guarantee)
- Lock variables store the state of the lock
- States
  - **Locked** (acquired or held)
  - **Unlocked** (available or free)
- Only 1 thread can hold a lock

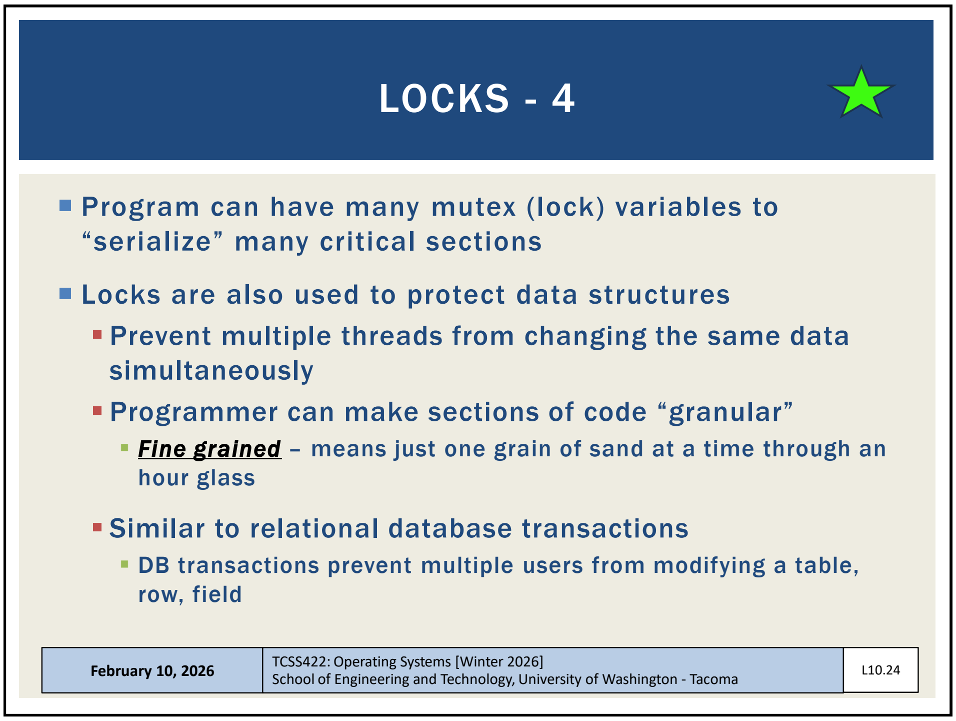
LOCKS - 3

- `pthread_mutex_lock(&lock)`
  - Try to acquire lock
  - If lock is free, calling thread will acquire the lock
  - Thread with lock enters critical section
    - Thread "owns" the lock
- No other thread can acquire the lock before the owner releases it.



OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, **Lock Granularity**
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table



LOCKS - 4

- Program can have many mutex (lock) variables to "serialize" many critical sections
- Locks are also used to protect data structures
  - Prevent multiple threads from changing the same data simultaneously
  - Programmer can make sections of code "granular"
    - ***Fine grained*** – means just one grain of sand at a time through an hour glass
  - Similar to relational database transactions
    - DB transactions prevent multiple users from modifying a table, row, field

FINE GRAINED?

- ***Is this code a good example of "fine grained parallelism"?***

```
pthread_mutex_lock(&lock);
a = b++;
b = a * c;
*d = a + b +c;
FILE * fp = fopen ("file.txt", "r");
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
ListNode *node = mylist->head;
Int i=0
while (node) {
  node->title = str1;
  node->subheading = str2;
  node->desc = str3;
  node->end = *e;
  node = node->next;
  i++
}
e = e - i;
pthread_mutex_unlock(&lock);
```

FINE GRAINED PARALLELISM

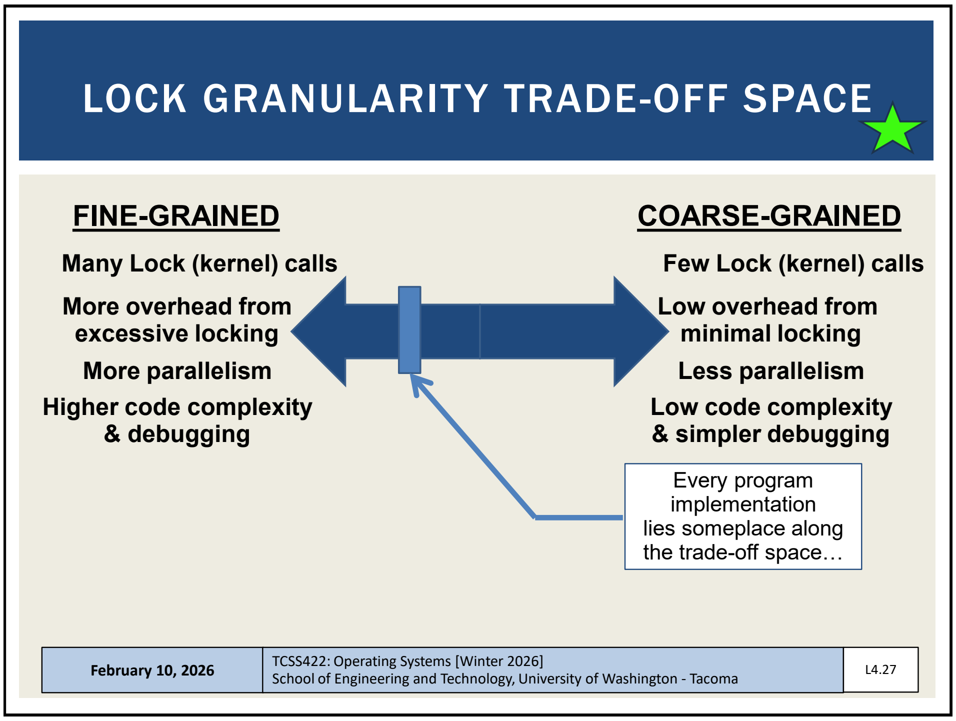
```
pthread_mutex_lock(&lock_a);
pthread_mutex_lock(&lock_b);
a = b++;
pthread_mutex_unlock(&lock_b);
pthread_mutex_unlock(&lock_a);

pthread_mutex_lock(&lock_b);
b = a * c;
pthread_mutex_unlock(&lock_b);

pthread_mutex_lock(&lock_d);
*d = a + b +c;
pthread_mutex_unlock(&lock_d);

FILE * fp = fopen ("file.txt", "r");
pthread_mutex_lock(&lock_e);
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
pthread_mutex_unlock(&lock_e);

ListNode *node = mylist->head;
int i=0 . . .
```



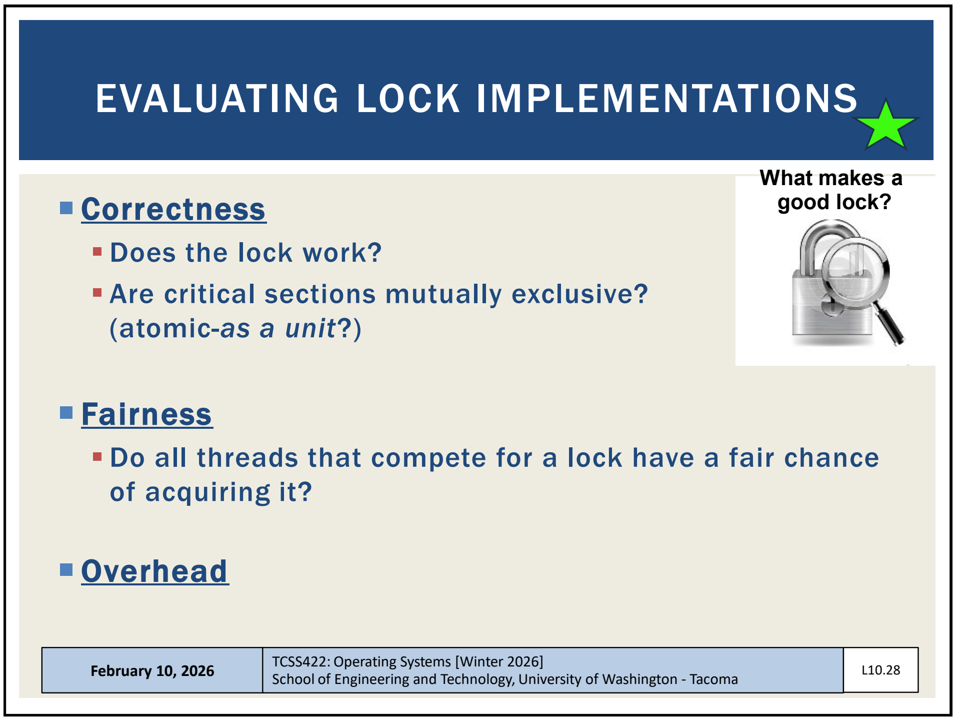
LOCK GRANULARITY TRADE-OFF SPACE

FINE-GRAINED	COARSE-GRAINED
Many Lock (kernel) calls	Few Lock (kernel) calls
More overhead from excessive locking	Low overhead from minimal locking
More parallelism	Less parallelism
Higher code complexity & debugging	Low code complexity & simpler debugging

Every program implementation lies someplace along the trade-off space…

February 10, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L4.27

27



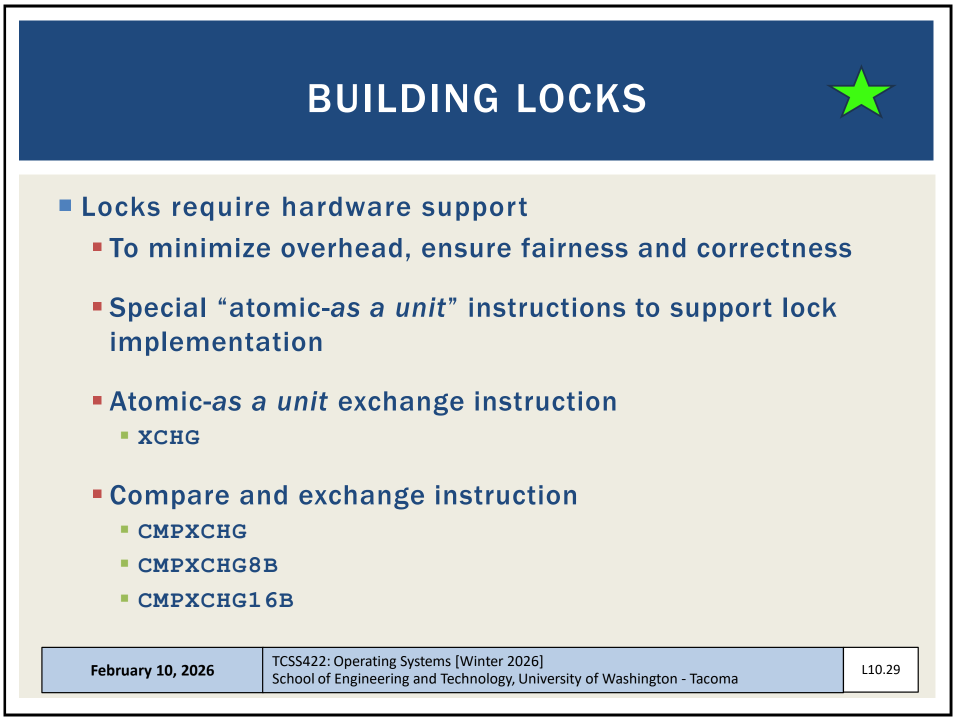
EVALUATING LOCK IMPLEMENTATIONS

What makes a good lock?

- **Correctness**
  - Does the lock work?
  - Are critical sections mutually exclusive? (atomic-*as a unit*?)
- **Fairness**
  - Do all threads that compete for a lock have a fair chance of acquiring it?
- **Overhead**

February 10, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L10.28

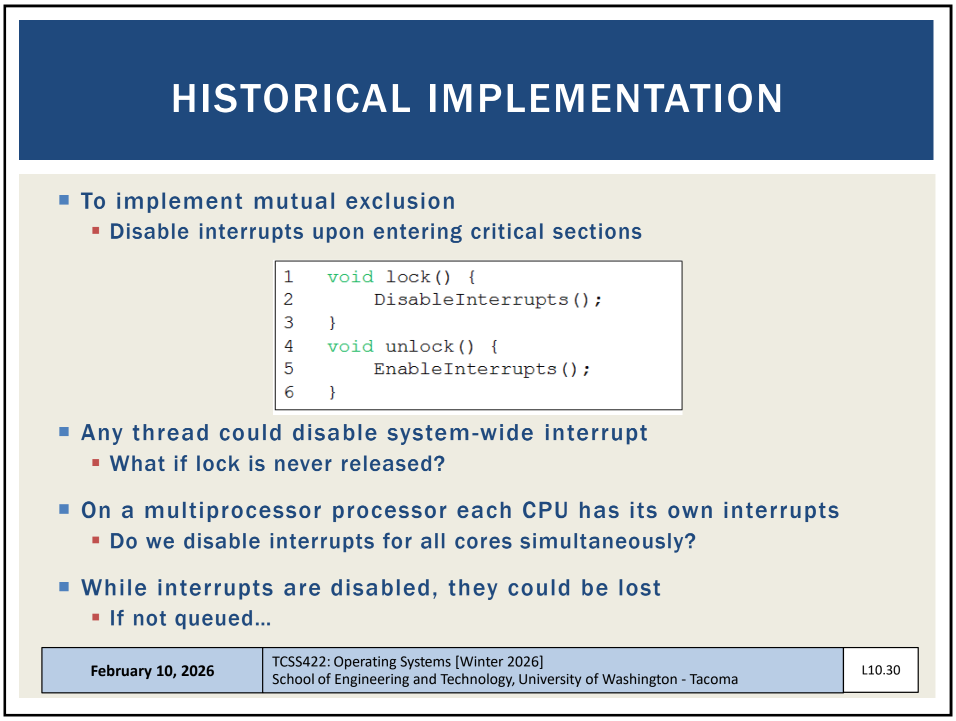
28



BUILDING LOCKS

- Locks require hardware support
  - To minimize overhead, ensure fairness and correctness
  - Special "atomic-*as a unit*" instructions to support lock implementation
  - Atomic-*as a unit* exchange instruction
    - XCHG
  - Compare and exchange instruction
    - CMPXCHG
    - CMPXCHG8B
    - CMPXCHG16B

29



HISTORICAL IMPLEMENTATION

- To implement mutual exclusion
  - Disable interrupts upon entering critical sections

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- Any thread could disable system-wide interrupt
  - What if lock is never released?
- On a multiprocessor processor each CPU has its own interrupts
  - Do we disable interrupts for all cores simultaneously?
- While interrupts are disabled, they could be lost
  - If not queued…

30

OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

BASIC SPIN LOCK IMPLEMENTATION

- Demonstration of lock implementation using C code
- C code is compiled to assembly, instructions are not atomic
- Idea is to imagine "what if" the lock code were atomic

- Is this lock implementation:
  (1)Correct?
  (2)Fair?
  (3)Performant?

```c
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;
}
```

BASIC SPIN LOCK: CORRECT?

- If both threads can run at the same time, then correctness requires luck... (e.g. *basic spin lock is incorrect*)

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!)	

- Here both threads have "acquired" the lock simultaneously

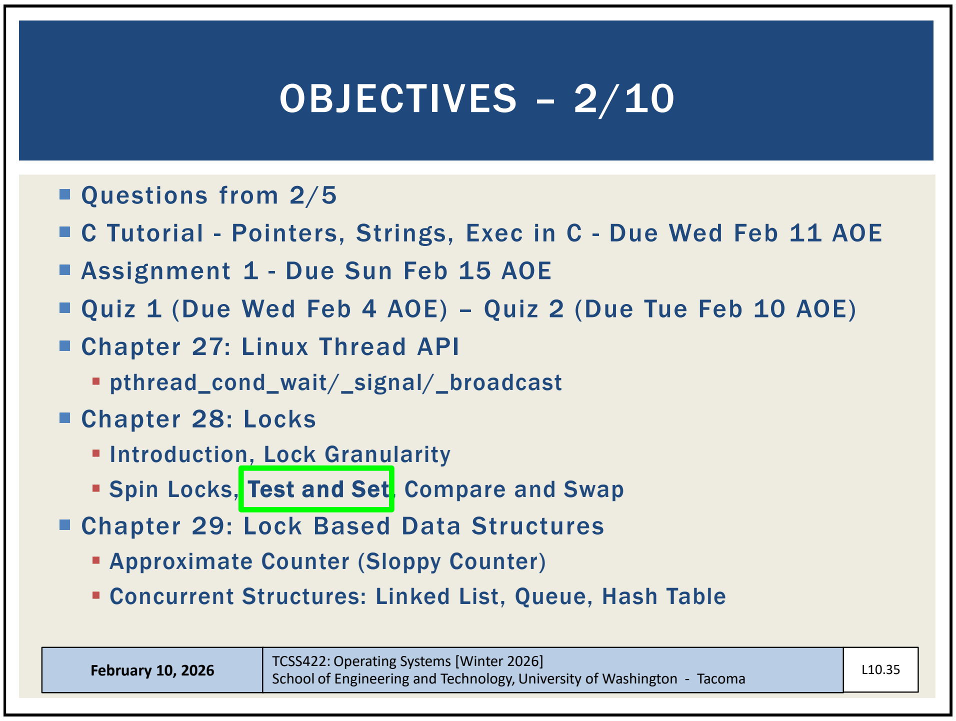
33

BASIC SPIN LOCK: PERFORMANCE ?

```
void lock(lock_t *mutex)
{
  while (mutex->flag == 1);    // while lock is unavailable, wait…
  mutex->flag = 1;
}
```

- What is wrong with while(<cond>); ?
- Spin-waiting wastes time actively waiting for another thread
- while (1); will "peg" a CPU core at 100%
  - Continuously loops, and evaluates mutex->flag value...
  - If multiple threads wait for the CPU, more CPU capacity is wasted
  - Generates heat...

34

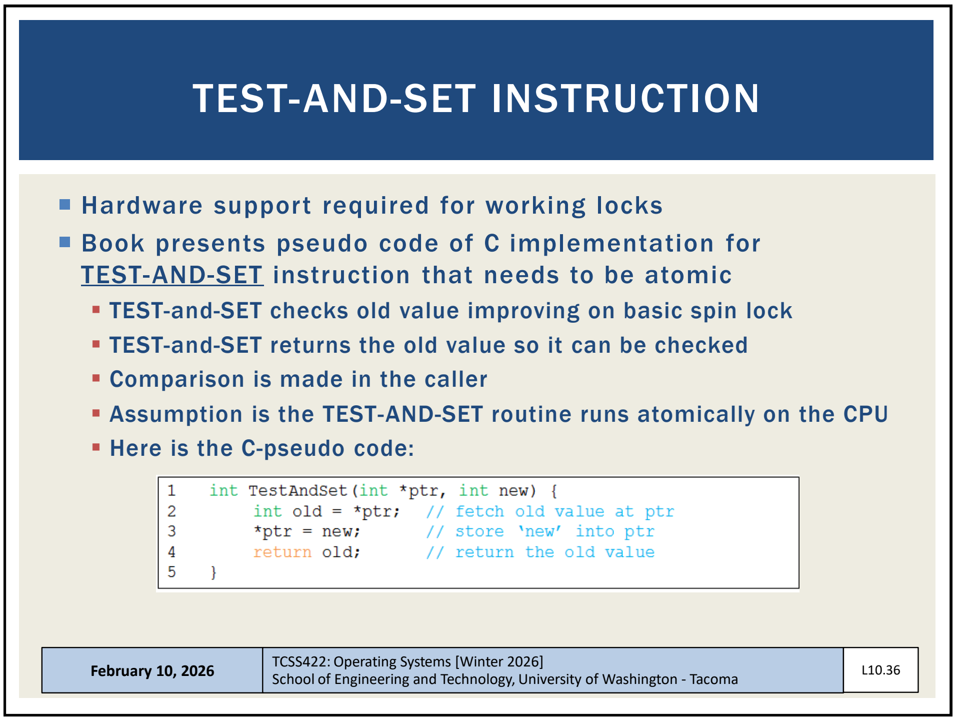


OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, **Test and Set**, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

February 10, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L10.35

35



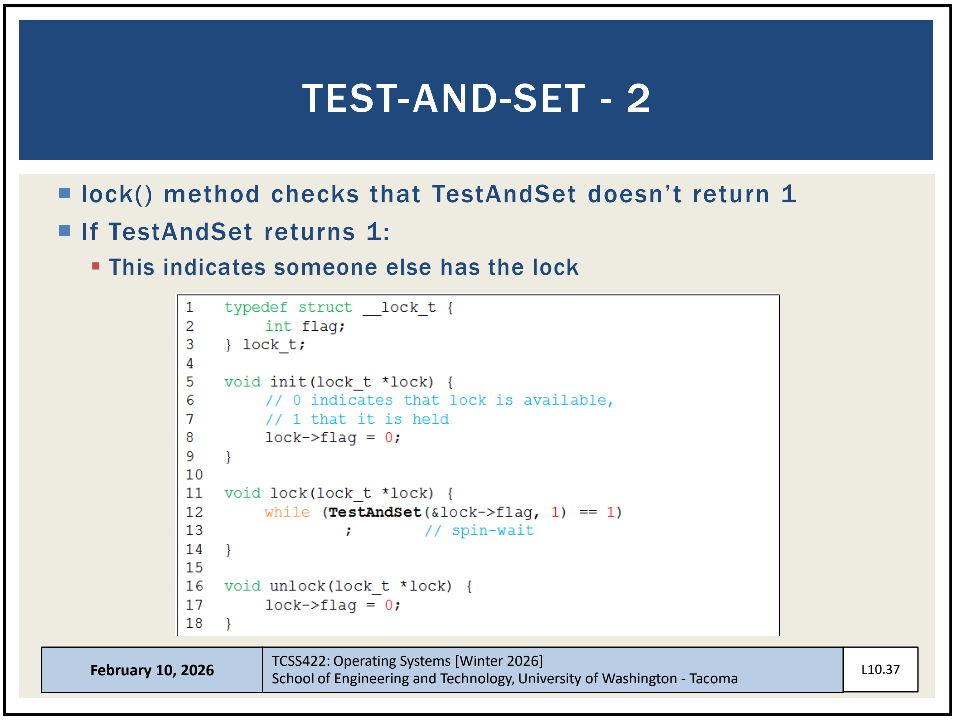
TEST-AND-SET INSTRUCTION

- Hardware support required for working locks
- Book presents pseudo code of C implementation for <u>TEST-AND-SET</u> instruction that needs to be atomic
  - TEST-and-SET checks old value improving on basic spin lock
  - TEST-and-SET returns the old value so it can be checked
  - Comparison is made in the caller
  - Assumption is the TEST-AND-SET routine runs atomically on the CPU
  - Here is the C-pseudo code:

```
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
}
```

February 10, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L10.36

36



TEST-AND-SET - 2

- lock() method checks that TestAndSet doesn't return 1
- If TestAndSet returns 1:
  - This indicates someone else has the lock

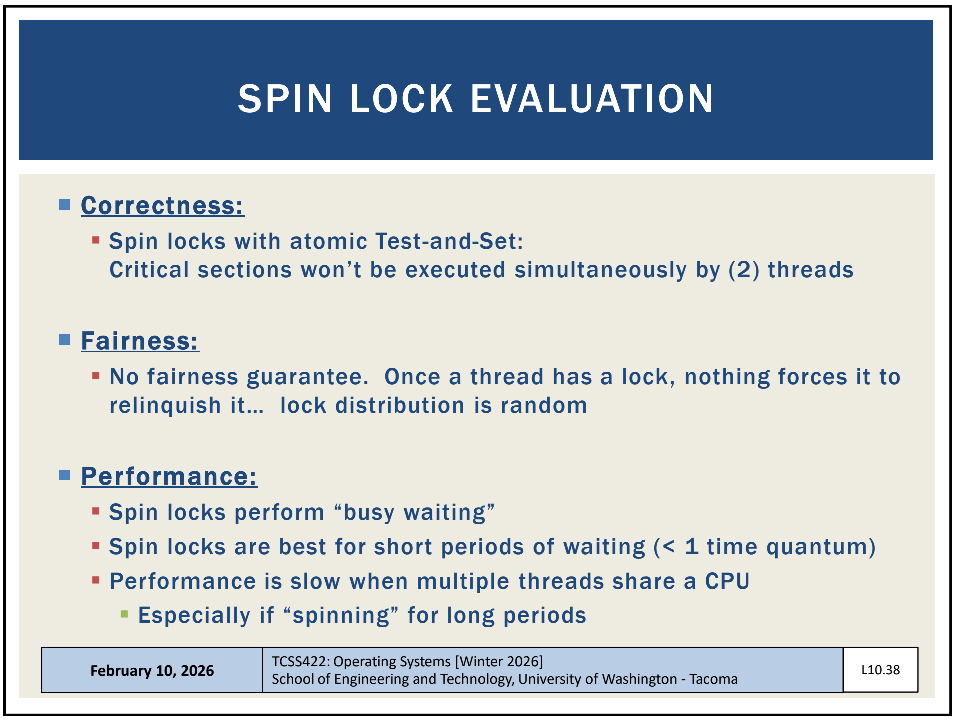
```
typedef struct __lock_t {
    int flag;
} lock_t;

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;
}
```

37



SPIN LOCK EVALUATION

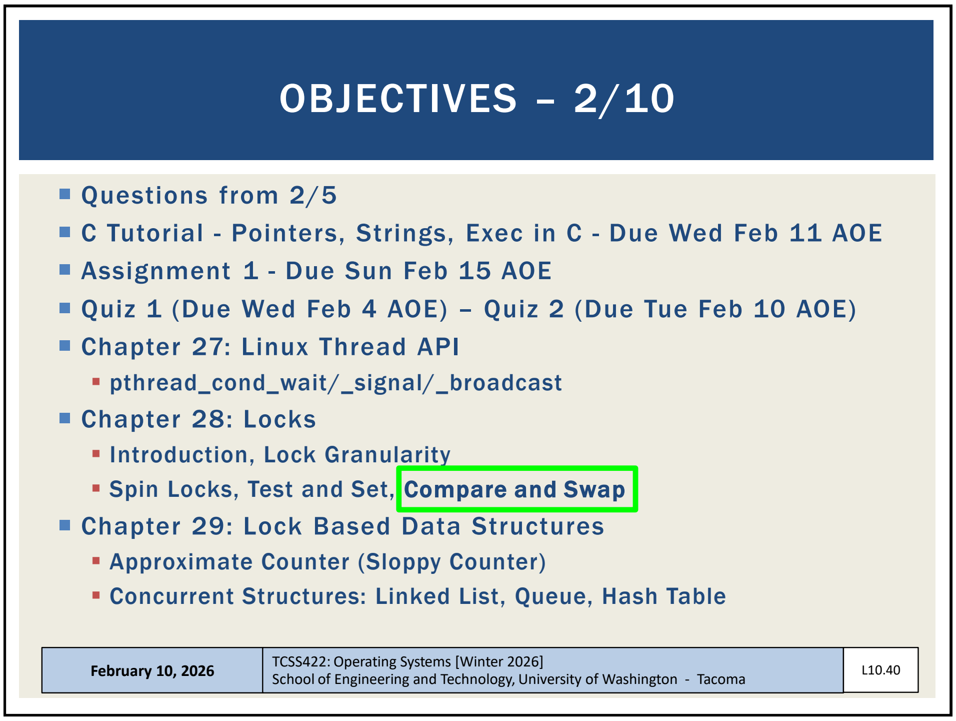
- **Correctness:**
  - Spin locks with atomic Test-and-Set:
    Critical sections won't be executed simultaneously by (2) threads
- **Fairness:**
  - No fairness guarantee. Once a thread has a lock, nothing forces it to relinquish it… lock distribution is random
- **Performance:**
  - Spin locks perform "busy waiting"
  - Spin locks are best for short periods of waiting (< 1 time quantum)
  - Performance is slow when multiple threads share a CPU
    - Especially if "spinning" for long periods

38

WE WILL RETURN AT 5:10PM

February 10, 2026 TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - coma L10.39

39

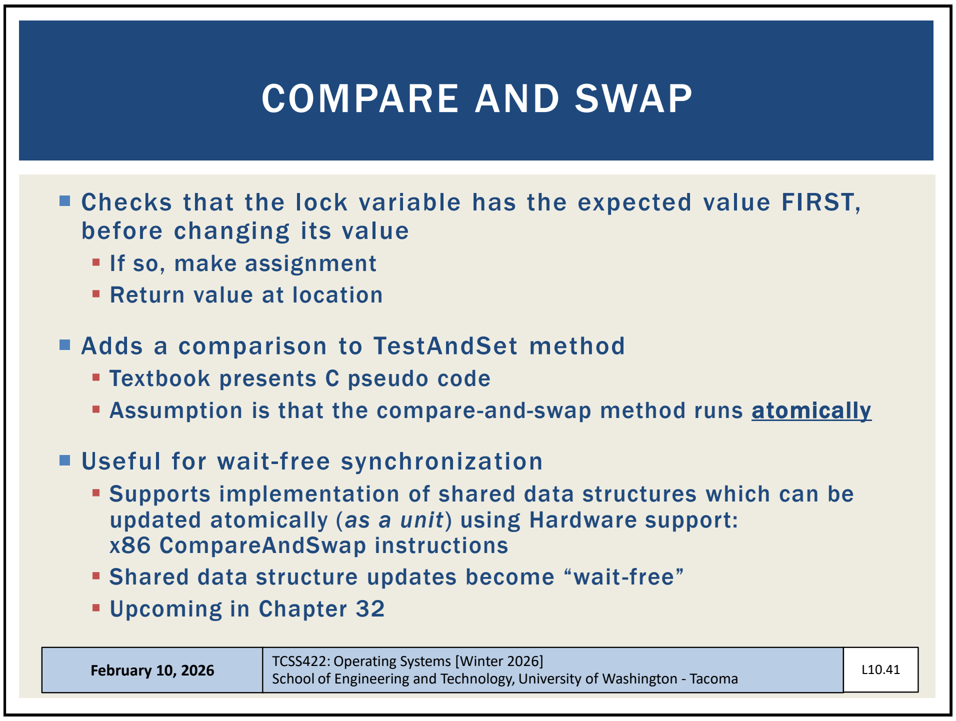


OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, **Compare and Swap**
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

February 10, 2026 TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma L10.40

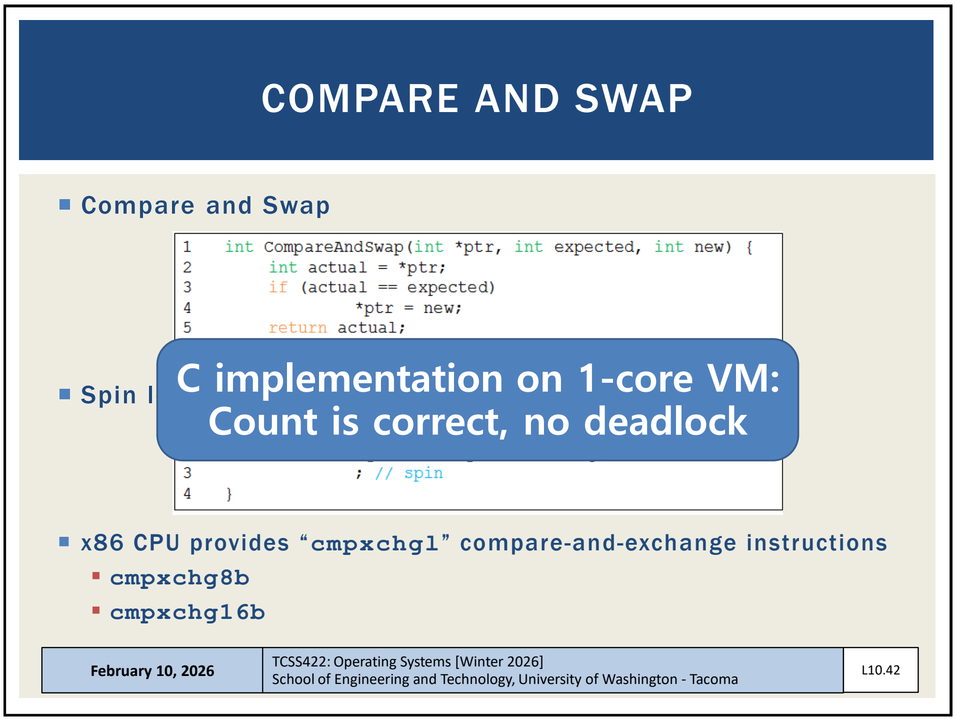
40



COMPARE AND SWAP

- Checks that the lock variable has the expected value FIRST, before changing its value
  - If so, make assignment
  - Return value at location
- Adds a comparison to TestAndSet method
  - Textbook presents C pseudo code
  - Assumption is that the compare-and-swap method runs **atomically**
- Useful for wait-free synchronization
  - Supports implementation of shared data structures which can be updated atomically (*as a unit*) using Hardware support: x86 CompareAndSwap instructions
  - Shared data structure updates become "wait-free"
  - Upcoming in Chapter 32

41



COMPARE AND SWAP

- Compare and Swap

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

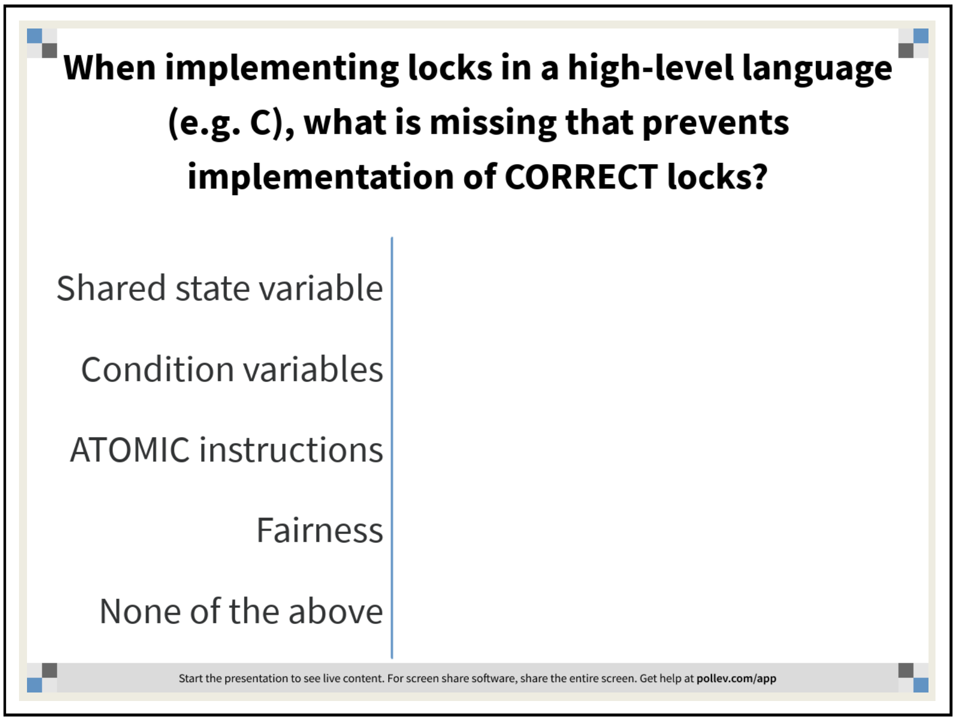
- Spin l

C implementation on 1-core VM:
Count is correct, no deadlock

```
            ; // spin
}
```

- x86 CPU provides "`cmpxchgl`" compare-and-exchange instructions
  - `cmpxchg8b`
  - `cmpxchg16b`

42

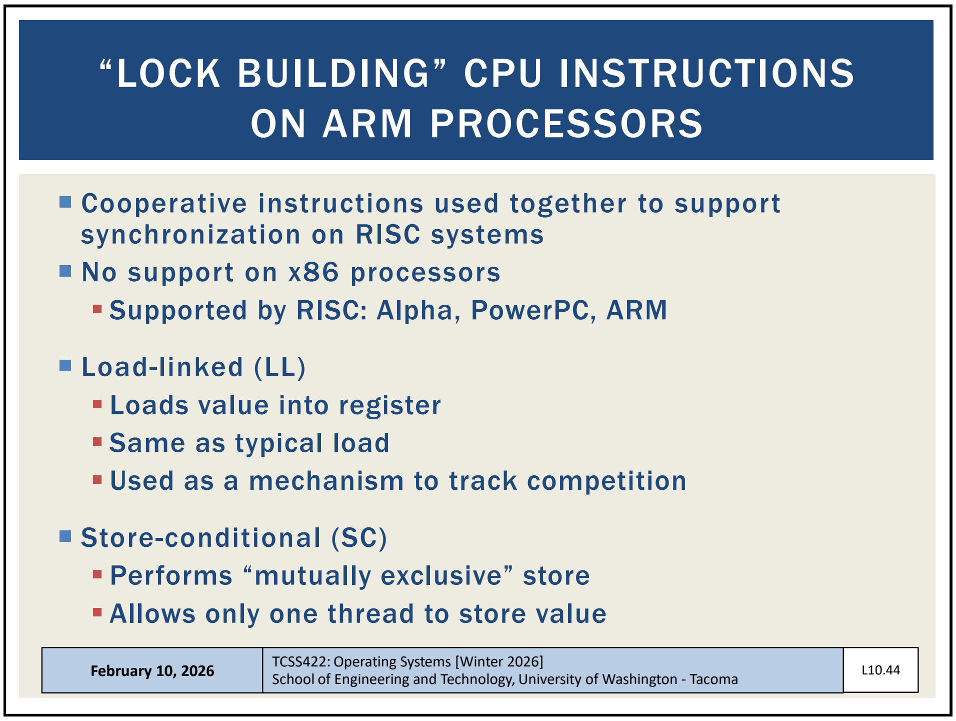


When implementing locks in a high-level language (e.g. C), what is missing that prevents implementation of CORRECT locks?

- Shared state variable
- Condition variables
- ATOMIC instructions
- Fairness
- 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**

43



"LOCK BUILDING" CPU INSTRUCTIONS ON ARM PROCESSORS

- Cooperative instructions used together to support synchronization on RISC systems
- No support on x86 processors
  - Supported by RISC: Alpha, PowerPC, ARM
- Load-linked (LL)
  - Loads value into register
  - Same as typical load
  - Used as a mechanism to track competition
- Store-conditional (SC)
  - Performs "mutually exclusive" store
  - Allows only one thread to store value

February 10, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L10.44

44

LL/SC LOCK

```c
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
            *ptr = value;
            return 1; // success!
    } else {
            return 0; // failed to update
    }
}
```

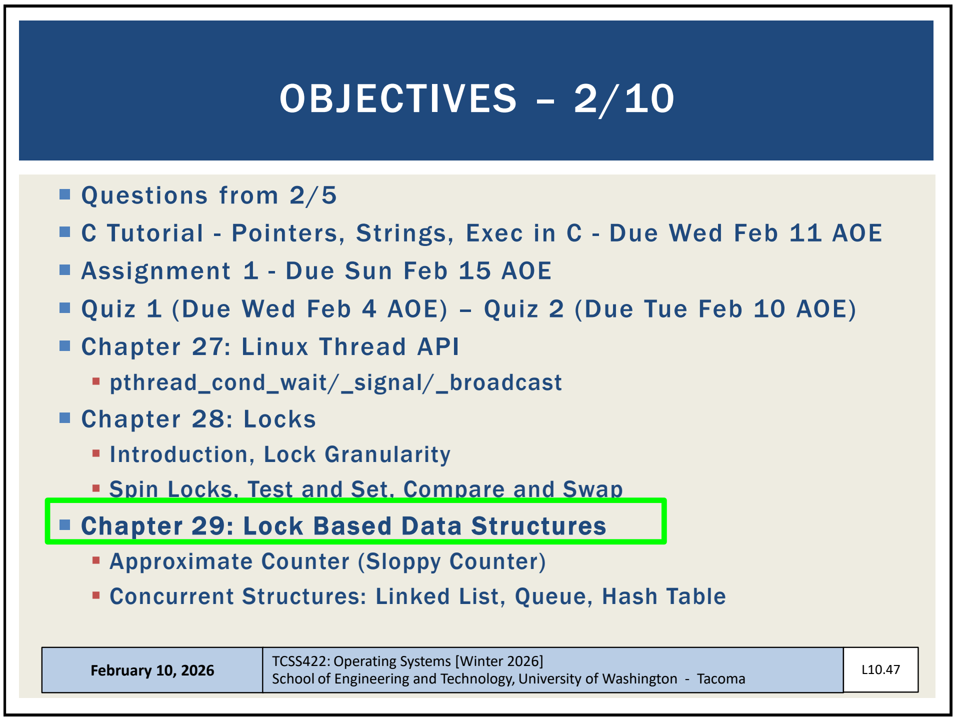
- LL instruction loads pointer value (ptr)
- SC only stores if the load link pointer has not changed
- Requires HW support
  - C code is psuedo code

LL/SC LOCK - 2

```c
void lock(lock_t *lock) {
    while (1) {
            while (LoadLinked(&lock->flag) == 1)
                    ; // spin until it's zero
            if (StoreConditional(&lock->flag, 1) == 1)
                    return; // if set-it-to-1 was a success: all done
                            otherwise: try it all over again
    }
}

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

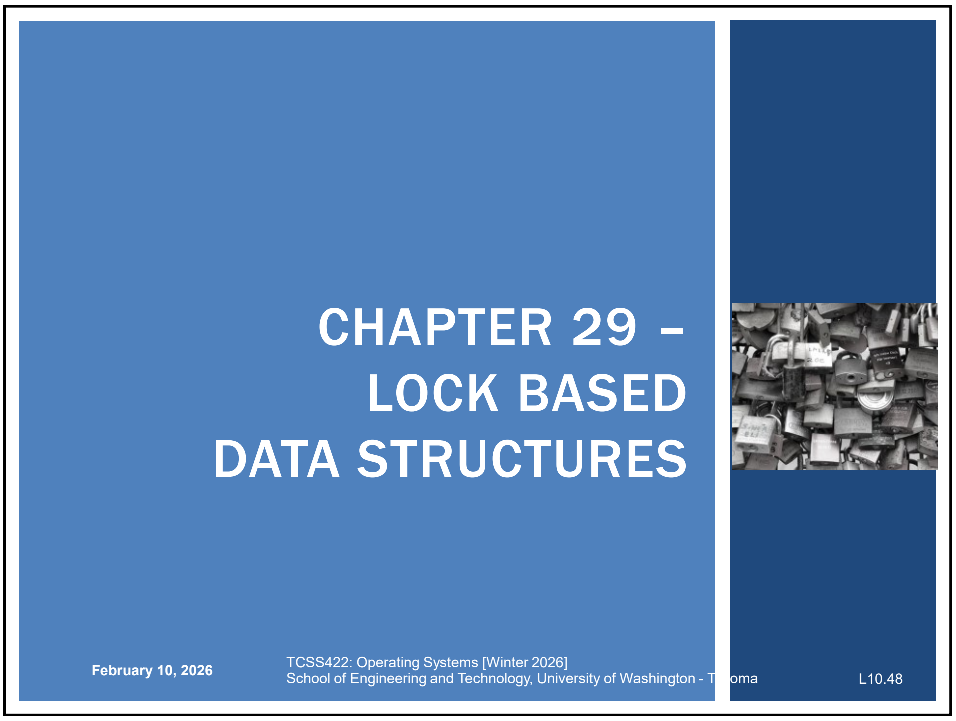
- Two instruction lock



OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- **Chapter 29: Lock Based Data Structures**
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table

47



CHAPTER 29 – LOCK BASED DATA STRUCTURES

48

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

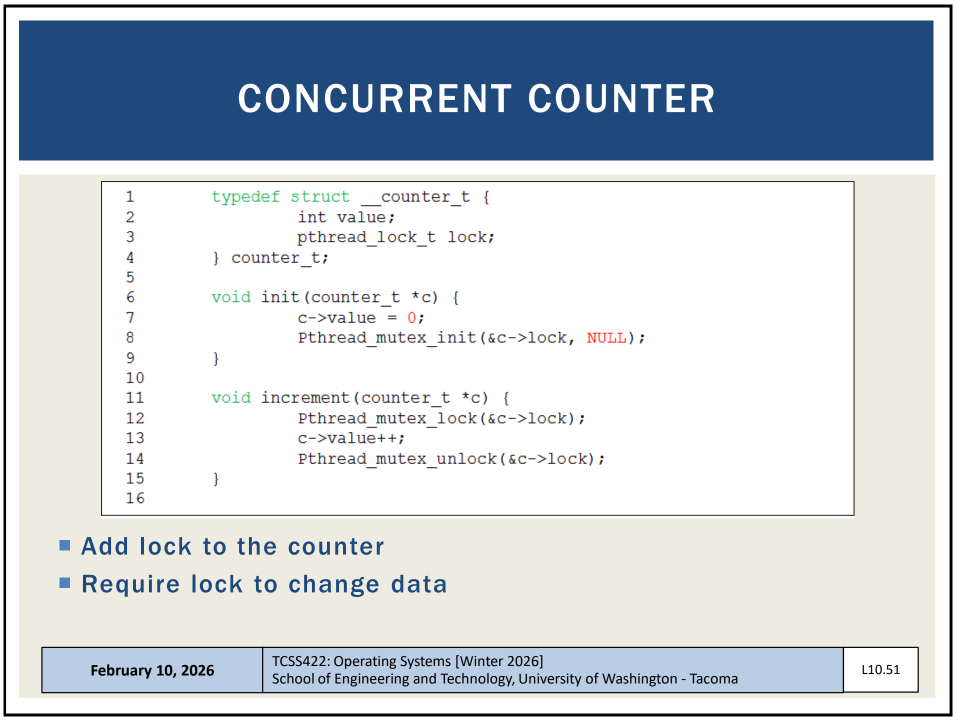
```
typedef struct __counter_t {
        int value;
} counter_t;

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

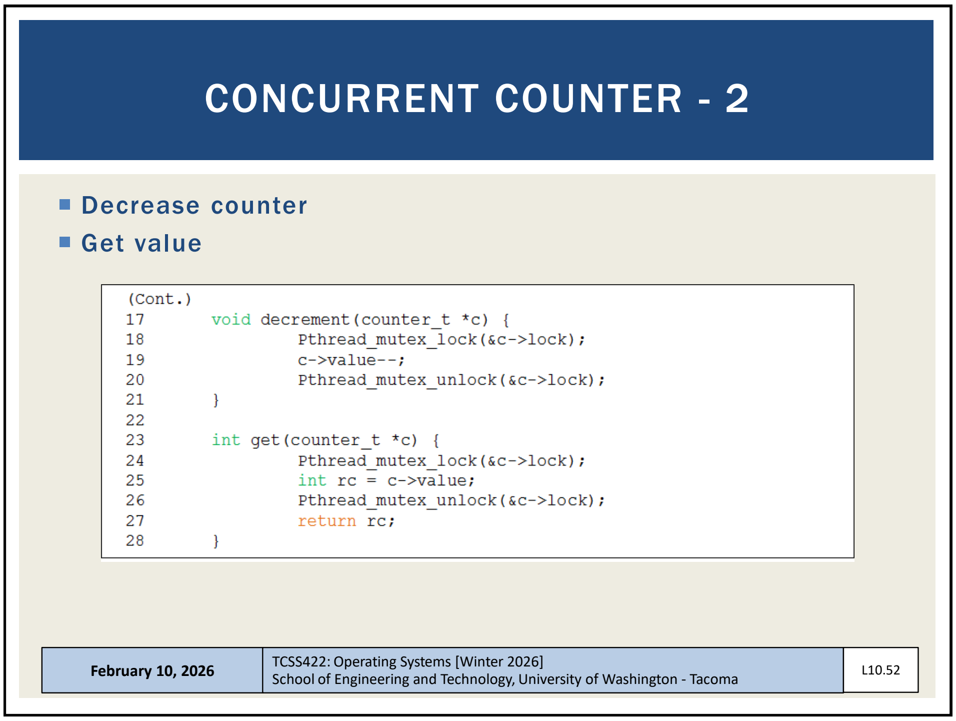
```c
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

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);
}

```

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



CONCURRENT COUNTER - 2

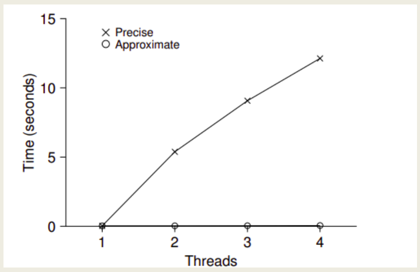
- Decrease counter
- Get value

```c
(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

- 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.

PERFECT SCALING

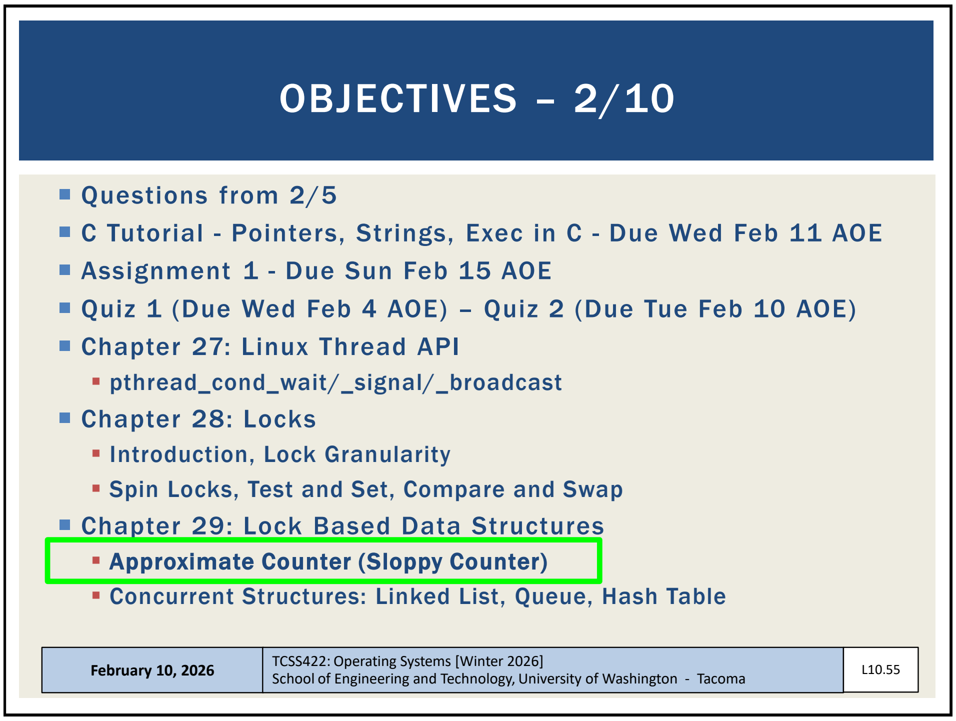
- Achieve (N) performance gain with (N) additional resources

- Throughput:
- Transactions per second (tps)

- 1 core
- N = 100 tps

- 10 cores (x10)
- N = 1000 tps (x10)

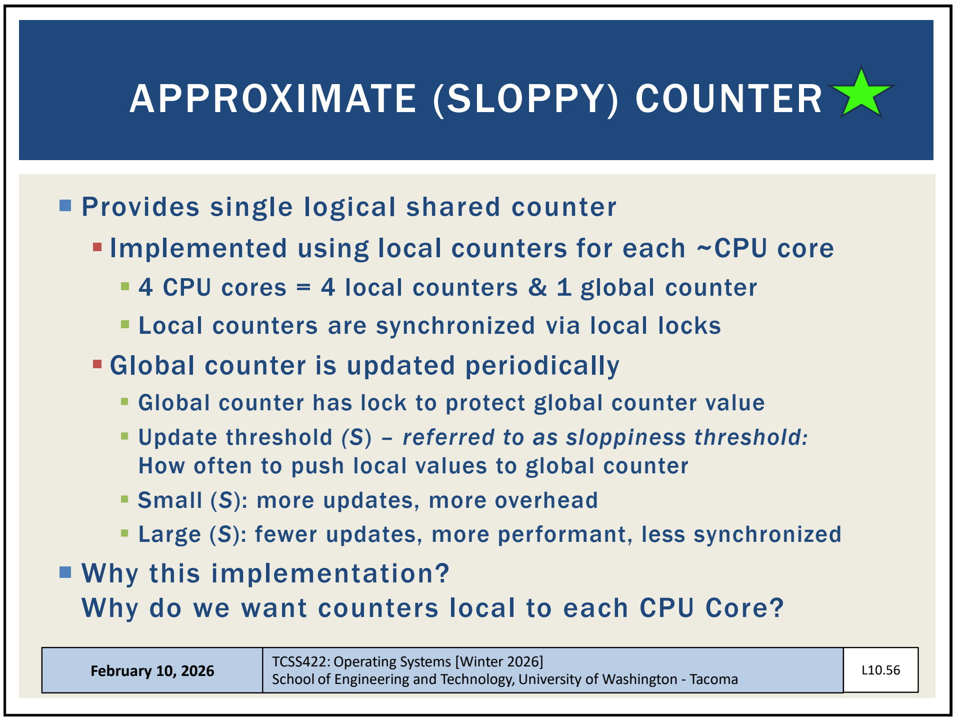
- ***Is parallel counting with a shared counter an embarrassingly parallel problem?***



OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - **Approximate Counter (Sloppy Counter)**
  - Concurrent Structures: Linked List, Queue, Hash Table

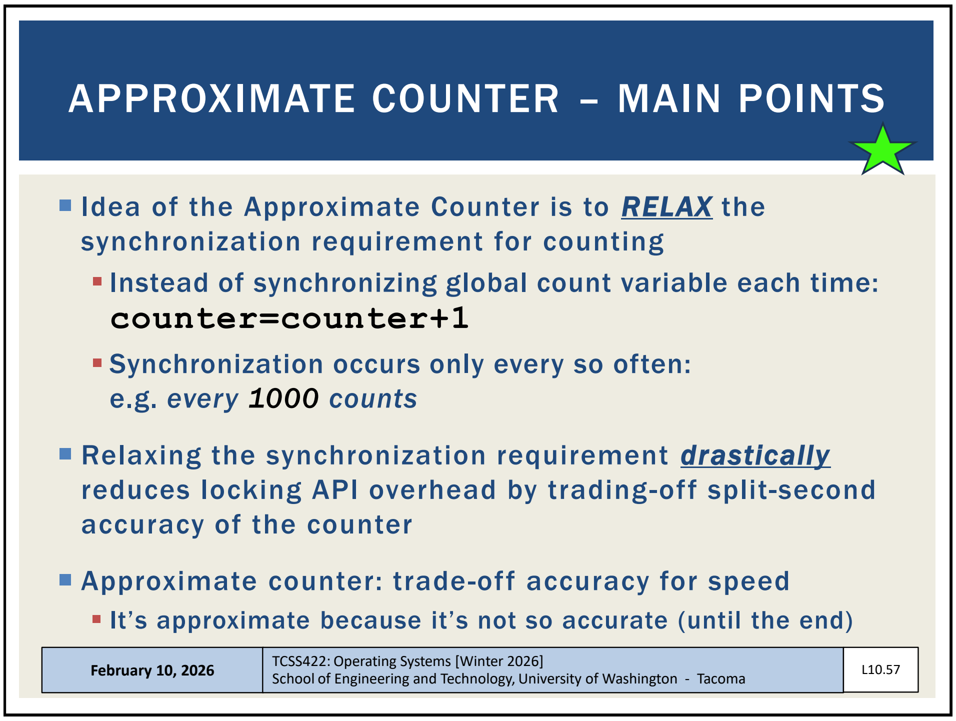
55



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?

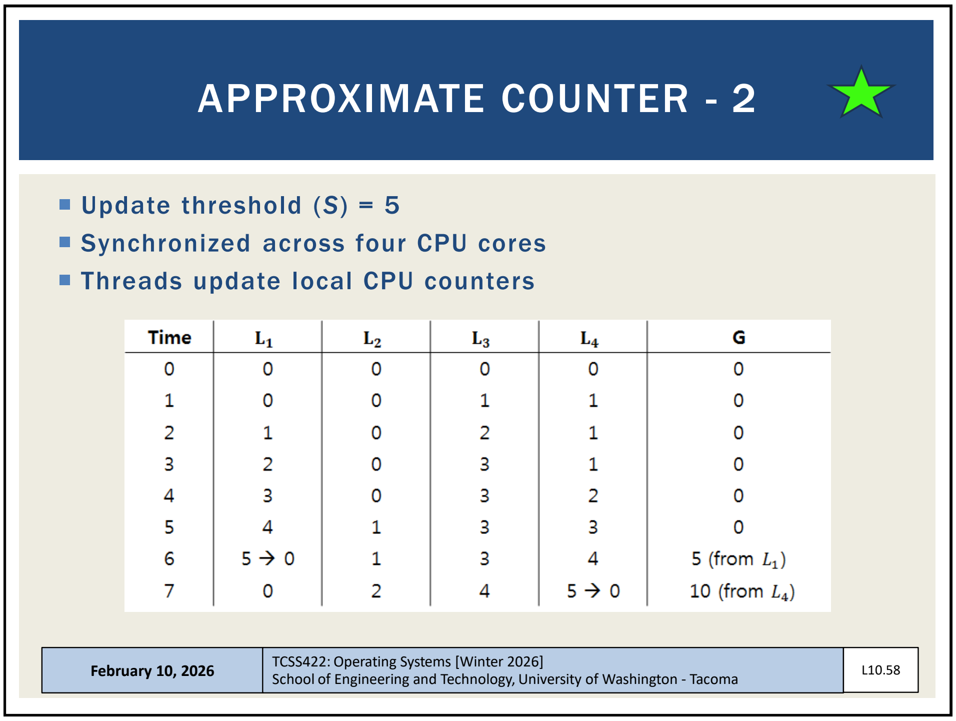
56



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)

57



APPROXIMATE COUNTER - 2

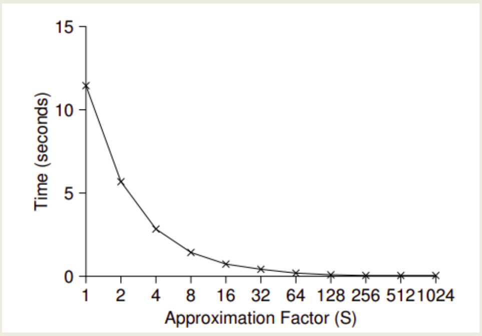
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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)

58

THRESHOLD VALUE *S*

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

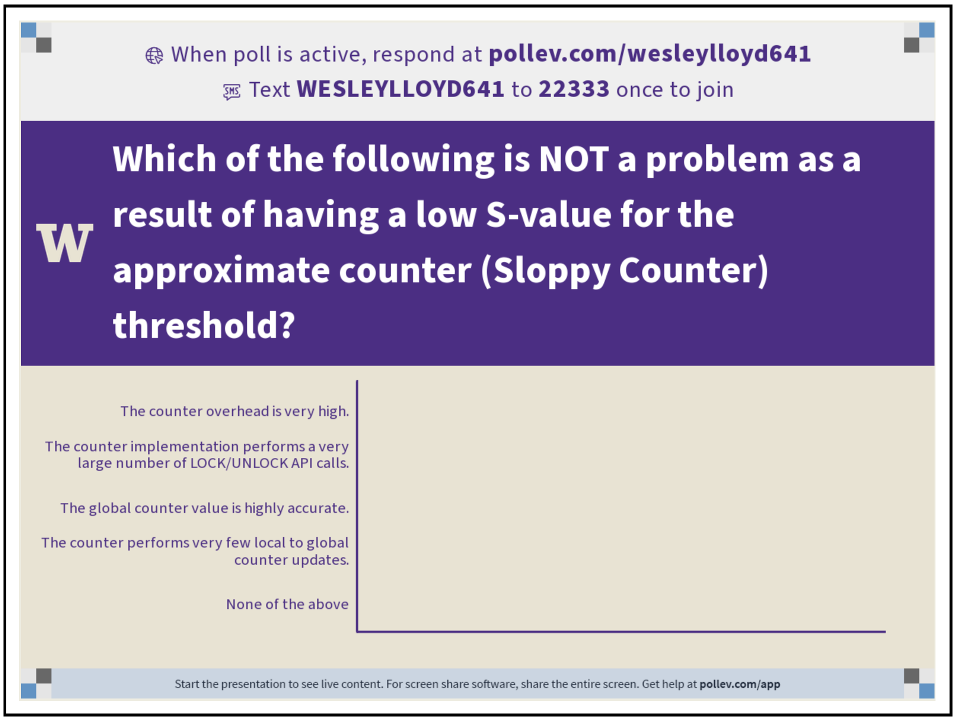


APPROXIMATE COUNTER - EXAMPLE

- Example implementation – sloppybasic.c

- Example features optional CPU affinity

→*Fix threads to run on pre-specified CPU cores*
→*As opposed to allowing for opportunistic scheduling by the OS*
→*Known as 'core pinning'*



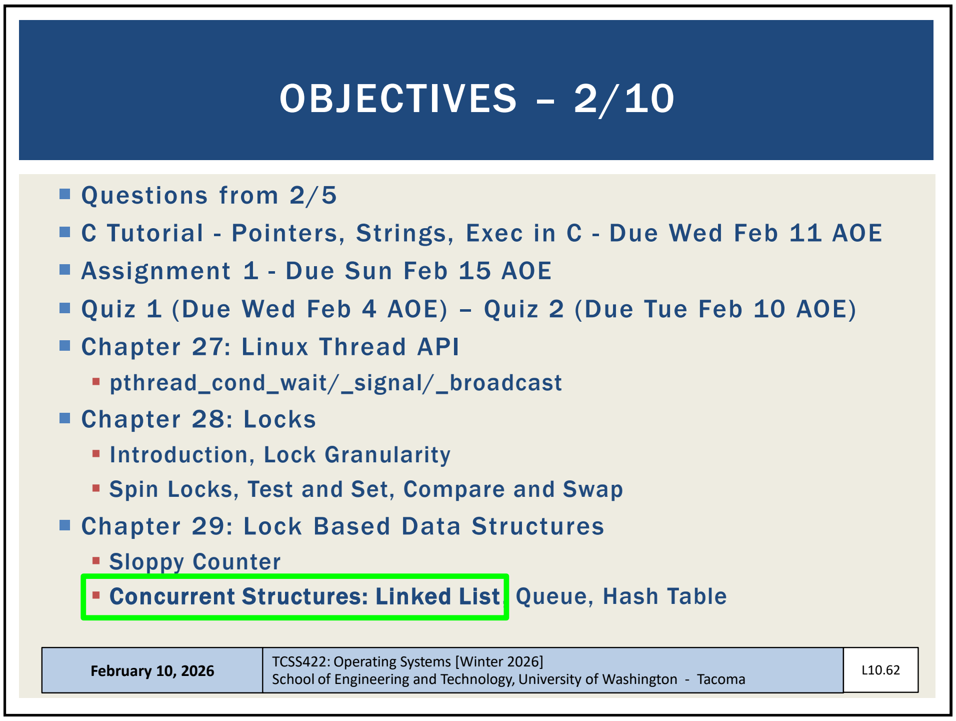
When poll is active, respond at **pollev.com/wesleylloyd641**
Text **WESLEYLLOYD641** to **22333** once to join

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**

61

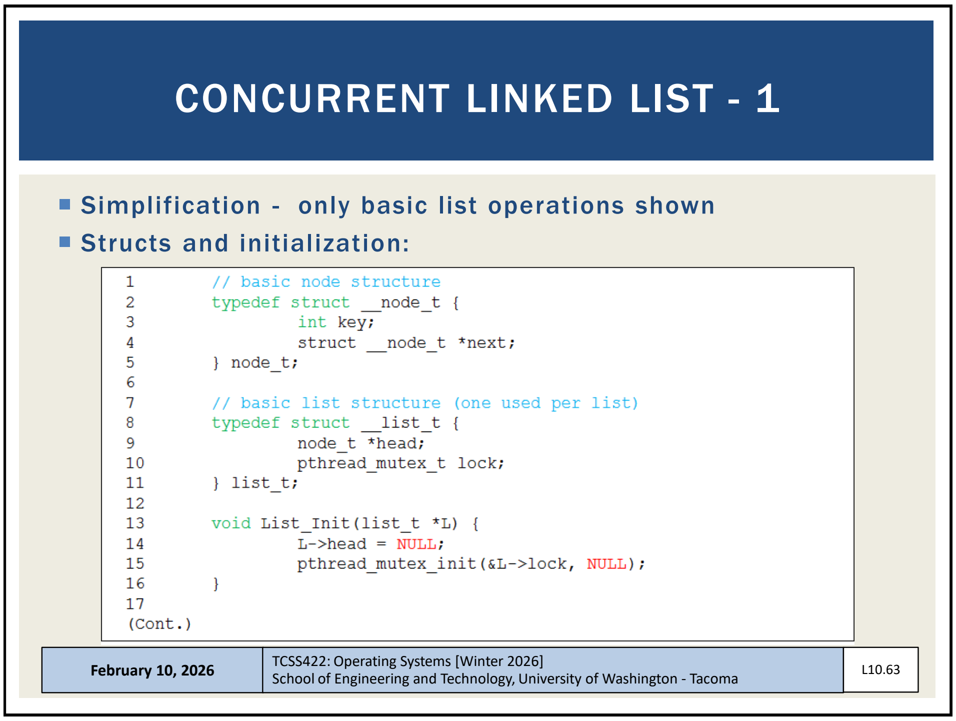


OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - **Concurrent Structures: Linked List** Queue, Hash Table

February 10, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L10.62

62



CONCURRENT LINKED LIST - 1

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

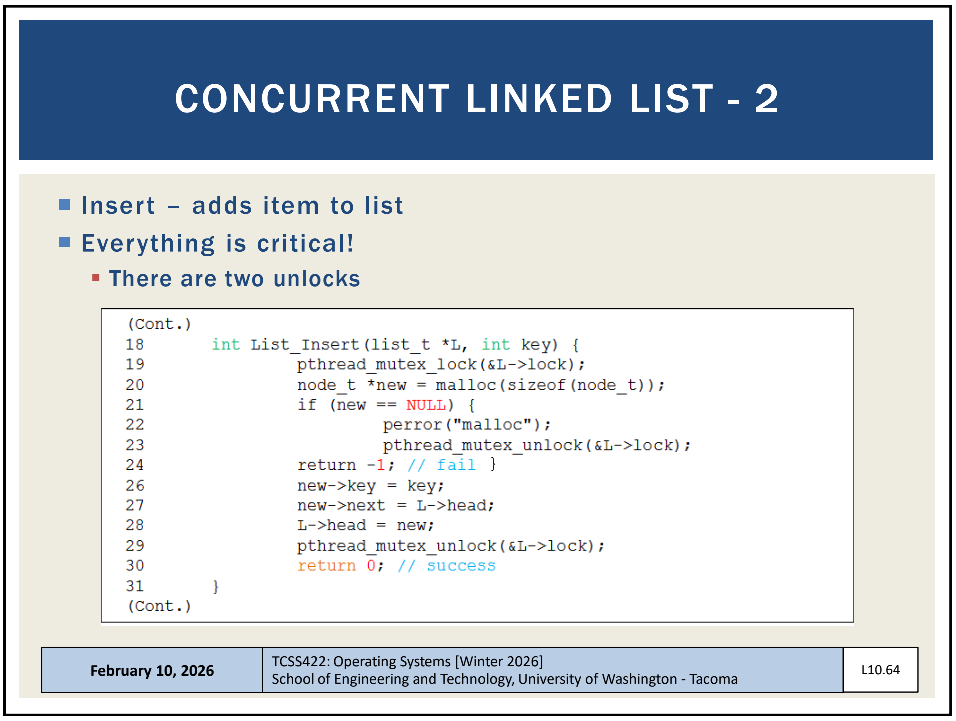
```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// 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.)
```

63



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.)
```

64

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

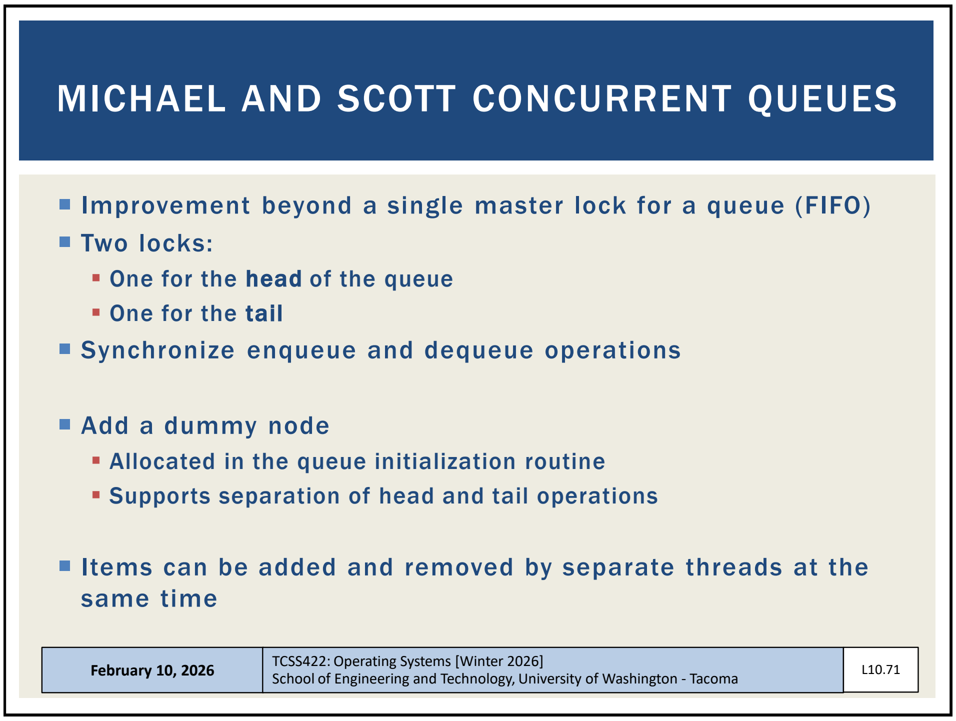
```
(Cont.)
int List_Lookup(list_t *L, int key) {
        int rv = -1;
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
                if (curr->key == key) {
                        rv = 0;
                        break;
                }
                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?

OBJECTIVES – 2/10

- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, **Queue**, Hash Table



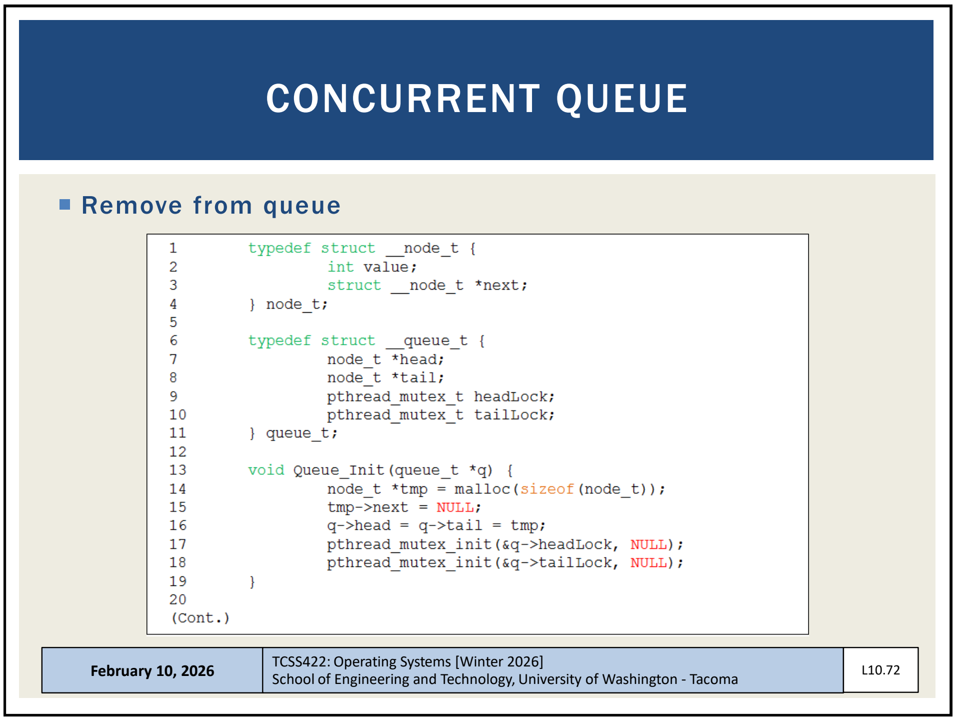
MICHAEL AND SCOTT CONCURRENT QUEUES

- 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

71



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.)
```

72

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.)
```

OBJECTIVES – 2/10

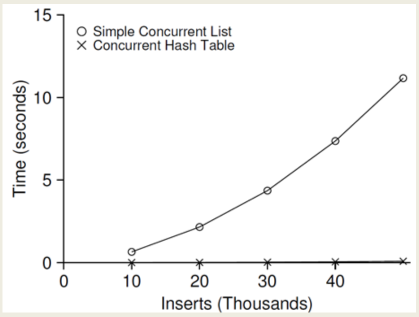
- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test and Set, Compare and Swap
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, **Hash Table**

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);
}
```

February 10, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L10.77

77

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

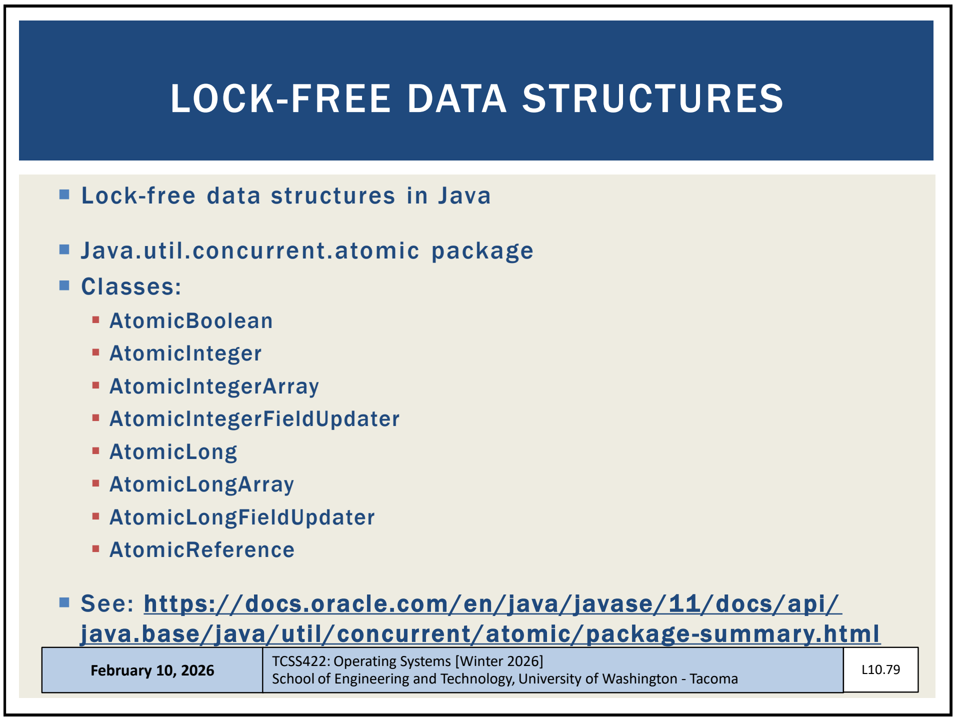
Multiple threads can more easily share data

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**

78

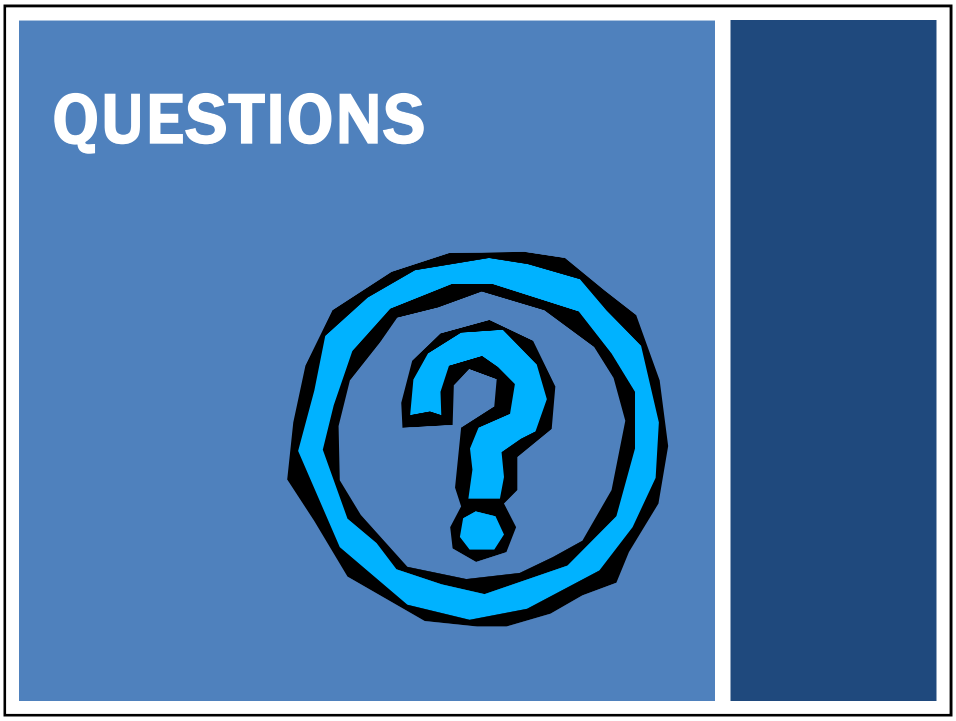


LOCK-FREE DATA STRUCTURES

- 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 10, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L10.79

79



QUESTIONS

80