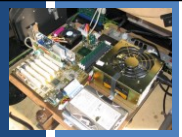
TCSS 422: OPERATING SYSTEMS



**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

- **Office Hours plan for Winter:**
- **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

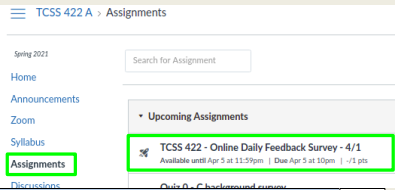
OBJECTIVES – 2/10

- **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

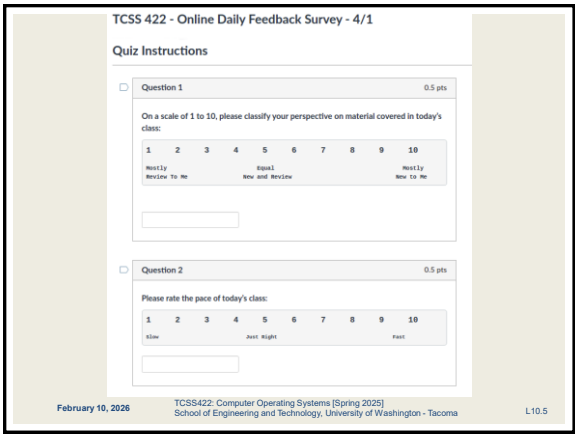
3

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



4



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)**

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

9

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);
```


10

FEEDBACK - 5

- See pthread_cast.c example (chapter 26)

11

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

12

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

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

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**

15

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**

16

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

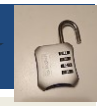
- Questions from 2/5
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- Assignment 1 - Due Sun Feb 15 AOE
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - **Introduction** Lock Granularity
  - 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":

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  …
3  lock(&mutex);
4  balance = balance + 1;
5  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

21

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.

22

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

23

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

24

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);
```

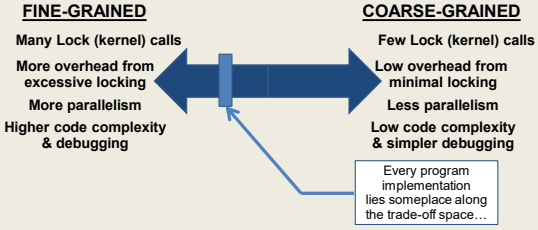
25

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 . . .
```

26

LOCK GRANULARITY TRADE-OFF SPACE



27

EVALUATING LOCK IMPLEMENTATIONS

- **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**

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

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

- 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

31

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?***

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 → lock is available, 1 → held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ;  // spin-wait (do nothing)
11      mutex->flag = 1;  // now SET it !
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```


32

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

35

TEST-AND-SET INSTRUCTION

- Hardware support required for working locks
- Book presents pseudo code of C implementation for TEST-AND-SET 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:

```
1  int TestAndSet(int *ptr, int new) {
2      int old = *ptr;   // fetch old value at ptr
3      *ptr = new;       // store 'new' into ptr
4      return old;       // return the old value
5  }
```


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

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 indicates that lock is available,
7       // 1 that it is held
8       lock->flag = 0;
9   }
10
11  void lock(lock_t *lock) {
12      while (TestAndSet(&lock->flag, 1) == 1)
13          ;       // spin-wait
14  }
15
16  void unlock(lock_t *lock) {
17      lock->flag = 0;
18  }
```


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

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

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

```
1   int CompareAndSwap(int *ptr, int expected, int new) {
2       int actual = *ptr;
3       if (actual == expected)
4           *ptr = new;
5       return actual;
```

- Spin l

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

```
3               ; // spin
4   }
```

- 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

44

LL/SC LOCK

```
1   int LoadLinked(int *ptr) {
2       return *ptr;
3   }
4
5   int StoreConditional(int *ptr, int value) {
6       if (no one has updated *ptr since the LoadLinked to this address) {
7           *ptr = value;
8           return 1; // success!
9       } else {
10          return 0; // failed to update
11      }
12  }
```

- 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

45

LL/SC LOCK - 2

```
1   void lock(lock_t *lock) {
2       while (1) {
3           while (LoadLinked(&lock->flag) == 1)
4               ; // spin until it's zero
5           if (StoreConditional(&lock->flag, 1) == 1)
6               return; // if set-it-to-1 was a success: all done
7                       otherwise: try it all over again
8       }
9   }
10
11  void unlock(lock_t *lock) {
12      lock->flag = 0;
13  }
```

- Two instruction lock

46

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

49

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      }
```


50

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

51

CONCURRENT COUNTER - 2

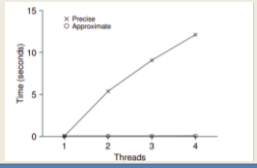
- Decrease counter
- Get value

```
(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      }
```


52

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.

53

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?***

54

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

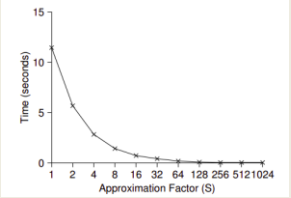
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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?



59

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'*

60

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

62

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
(Cont.)
```


63

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              new->key = key;
26              new->next = L->head;
27              L->head = new;
28              pthread_mutex_unlock(&L->lock);
29              return 0; // success
30      }
31
(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.)
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      }
```


65

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 …

66

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
```


67

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      }
```


68

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?

69

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

70

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

```
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.)
```


72

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.)
```


73

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**

74

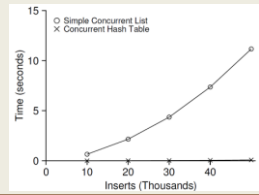
CONCURRENT HASH TABLE

- 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

75

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.

76

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      }
```


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**

79

QUESTIONS

80