


TCSS 422: OPERATING SYSTEMS

Three Easy Pieces: Locks, Lock Based Data Structures



Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma


April 23, 2018 TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Quiz 2 – Scheduling Review
- Assignment 1 – MASH Shell
- Review: Proportional Share Scheduler – Ch. 9
- Review: Concurrency: Introduction – Ch. 26
- Review: Linux Thread API – Ch. 27
- Locks – Ch. 28
- Lock Based Data Structures – Ch. 29


April 23, 2018 TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma L5.2

CHAPTER 28 – LOCKS



April 23, 2018 TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma L5.3

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);
```

April 23, 2018 TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma L7.4

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

April 23, 2018 TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma L7.5

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.

April 23, 2018 TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma L7.6

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

April 23, 2018

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

L7.7

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);
```



April 23, 2018

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

L7.8

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 . . .
```



April 23, 2018

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

L7.9

EVALUATING LOCK IMPLEMENTATIONS

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



April 23, 2018

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

L7.10

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
 - CMPPXCHG
 - CMPPXCHG8B
 - CMPPXCHG16B

April 23, 2018

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

L7.11

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...

April 23, 2018

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

L7.12

## SPIN LOCK IMPLEMENTATION

- Operate without atomic-as a unit assembly instructions
- "Do-it-yourself" Locks
- Is this lock implementation: Correct? Fair? 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 }
```

April 23, 2018

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

L7.13

## DIY: CORRECT?

- Correctness requires luck... (e.g. *DIY 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

April 23, 2018

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

L7.14

## DIY: PERFORMANT?

```
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...
  - Generates heat...

April 23, 2018

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

L7.15

## TEST-AND-SET INSTRUCTION

- C implementation: not atomic
  - Adds a simple check to basic spin lock
  - One a single core CPU system with preemptive scheduler:
  - Try this...

```
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 }
```

- lock() method checks that TestAndSet doesn't return 1
- Comparison is in the caller
- Single core systems are becoming scarce
- Try on a one-core VM

April 23, 2018

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

L7.16

## DIY: TEST-AND-SET - 2

- Requires a preemptive scheduler on single CPU core system
- Lock is never released without a context switch
- 1-core VM: occasionally will deadlock, doesn't miscount

```
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 }
```

April 23, 2018

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

L7.17

## SPIN LOCK EVALUATION

- Correctness:**
  - Spin locks guarantee: 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...
- Performance:**
  - Spin locks perform "busy waiting"
  - Spin locks are best for short periods of waiting
  - Performance is slow when multiple threads share a CPU
    - Especially for long periods

April 23, 2018

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

L7.18

## 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
- Useful for wait-free synchronization
  - Supports implementation of shared data structures which can be updated atomically (as a *unit*) using the HW support CompareAndSwap instruction
  - Shared data structure updates become "wait-free"
  - Upcoming in Chapter 32

April 23, 2018

TCS5422: Operating Systems [Spring 2018]  
Institute of Technology, University of Washington - Tacoma

L7.19

## 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;
6 }
```

### Spin lock

**1-core VM:  
Count is correct, no deadlock**

```
2 while (lock == 1)
3 ; // spin
4 }
```

- X86 provides "cmpxchg1" compare-and-exchange instruction
  - cmpxchg8b
  - cmpxchg16b

April 23, 2018

TCS5422: Operating Systems [Spring 2018]  
Institute of Technology, University of Washington - Tacoma

L7.20

## TWO MORE "LOCK BUILDING" CPU INSTRUCTIONS

- 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

April 23, 2018

TCS5422: Operating Systems [Spring 2018]  
Institute of Technology, University of Washington - Tacoma

L7.21

## 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 pseudo code

April 23, 2018

TCS5422: Operating Systems [Spring 2018]  
Institute of Technology, University of Washington - Tacoma

L7.22

## 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

April 23, 2018

TCS5422: Operating Systems [Spring 2018]  
Institute of Technology, University of Washington - Tacoma

L7.23

## HARDWARE SPIN LOCKS - SUMMARY

- Simple, correct
- Slow
- With long locks, waiting threads spin for entire timeslice
  - Repeat comparison continuously
  - Busy waiting

**How To Avoid *Spinning*?**  
**Need both HW & OS Support !**

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.24

## FETCH-AND-ADD

- HW CPU Instruction
- Increment counter atomically-as a *unit* in one instruction

```
1 int FetchAndAdd(int *ptr) {
2 int old = *ptr;
3 *ptr = old + 1;
4 return old;
5 }
```

- Fetch and return value
- Increment by 1

April 23, 2018

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

L7.25

## TICKET LOCK

- Can build Ticket Lock using Fetch-and-Add
- Ensures progress of all threads (fairness)

```
1 typedef struct __lock_t {
2 int ticket;
3 int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7 lock->ticket = 0;
8 lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12 int myturn = FetchAndAdd(&lock->ticket);
13 while (lock->turn != myturn)
14 ; // spin
15 }
16 void unlock(lock_t *lock) {
17 FetchAndAdd(&lock->turn);
18 }
```

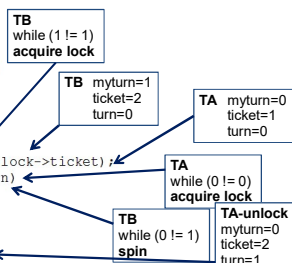
April 23, 2018

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

L7.26

## TICKET LOCK - 2

```
1 typedef struct __lock_t {
2 int ticket;
3 int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7 lock->ticket = 0;
8 lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12 int myturn = FetchAndAdd(&lock->ticket);
13 while (lock->turn != myturn)
14 ; // spin
15 }
16 void unlock(lock_t *lock) {
17 FetchAndAdd(&lock->turn);
18 }
```



April 23, 2018

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

L7.27

## YIELD() – SYSTEM CALL

```
1 void init() {
2 flag = 0;
3 }
4
5 void lock() {
6 while (TestAndSet(&flag, 1) == 1)
7 yield(); // give up the CPU
8 }
9
10 void unlock() {
11 flag = 0;
12 }
```

- Give up the CPU – instead of busy waiting...
  - running → ready
- Ready relinquishes the CPU for another thread (ctxt. switch)
- How does the thread get the CPU back?
  - OS must opportunistically reschedule it: ready → running

April 23, 2018

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

L7.28

## THREAD QUEUES

- Don't allow the OS to control your program
  - Use internal **Thread Queues**
- Allows programmer to maintain control
  - Ensure fairness, prevent starvation
  - Better for synchronizing large #'s of threads
- Require OS support to add/remove threads to/from queue(s)
- Solaris API:
  - `park()`: puts thread to sleep
  - `unpark(threadID)`: wakes specified thread
- Linux API: `futex()`

April 23, 2018

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

L7.29

## THREAD QUEUES - 2

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4 m->flag = 0;
5 m->guard = 0;
6 queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10 while (TestAndSet(&m->guard, 1) == 1)
11 ; // acquire guard lock by spinning
12 if (m->flag == 0) {
13 m->flag = 1; // lock is acquired
14 m->guard = 0;
15 }
16 queue_add(m->q, gettid());
17 m->guard = 0;
18 park();
19 }
20
21 ...
```

Guard uses a spin-lock to protect the critical sections in `lock()` and `unlock()`

Obtain guard lock

try to obtain actual lock

lock unavailable; add thread to queue potential wakeup/waiting race

April 23, 2018

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

L7.30

THREAD QUEUES - 3

- Unlock

```
22 void unlock(lock_t *m) {
23 while (TestAndSet(&m->guard, 1) == 1) // Obtain guard lock (spin)
24 ; // acquire guard lock by spinning
25 if (queue_empty(m->q))
26 m->flag = 0; // let go of lock; no one wants it
27 else
28 unpark(queue_remove(m->q)); // wake up thread from queue
29 m->guard = 0; // release guard lock
30 }
```

- Note: no change to m->flag if unparking a thread
- Lock is passed to the unparked thread “directly”

April 23, 2018

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

L7.31

WAKEUP/WAITING RACE

- Thread B: context switch occurs immediately before call to park()
- Thread A: releases lock, calls unpark, queue is empty
- Thread B: regains context, proceeds to lock itself forever


- Need new system call
  - **setpark()**- informs OS about soon to be parked thread
  - Subsequent calls to unpark() are aware that ThreadB is about to park
  - ThreadB's call to park() immediately returns

April 23, 2018

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

L7.32

FUTEX



- **Fast Userspace MuTEX**
- Linux futex system calls similar to park() and unpark()
- Linux uses an in-kernel queue
- Provides a futex() system call

- Provides atomic-as a unit compare-and-block operation
- **Futex is a lower-level construct**
- Used as **building blocks** for:  
**mutex, condition variables, semaphores**
- Objective: reduce the number of system calls

April 23, 2018

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

L7.33

FUTEX: WRITE YOUR OWN MUTEX LOCK

- futex\_wait(addr, expected)
  - Put calling thread to sleep
  - If value @ addr != expected → return immediately
- futex\_wake(addr)
  - Wake one thread that is waiting on the queue
- These are not exposed as C library calls directly
  - Call futex() with FUTEX\_WAIT or FUTEX\_WAKE
- Use a 32-bit integer
  - The leftmost bit (the +/- sign) tracks the lock state
    - 0 - free
    - 1 - locked
  - Remaining 31 bits: identifies thread

April 23, 2018

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

L7.34

HYBRID - TWO PHASE LOCKS

- Hybrid between spin-locks and yielding
- Useful if lock is about to be released


- First phase – spin lock
  - Spin for some time waiting for the lock to be released
  - If lock is not acquired after time expires enter phase two.
- Second phase - yield
  - Thread sleeps (yields)
  - Is awoken when the lock becomes free

April 23, 2018

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

L7.35

CHAPTER 29 –  
LOCK BASED  
DATA STRUCTURES



April 23, 2018

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

L7.36

OBJECTIVES

- Concurrent Data Structures
- Performance
- Lock Granularity

April 23, 2018

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

L7.37

LOCK-BASED  
CONCURRENT DATA STRUCTURES

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

April 23, 2018

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

L7.38

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 }
```

April 23, 2018

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

L7.39

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

April 23, 2018

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

L7.40

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 }
```

April 23, 2018

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

L7.41

CONCURRENT COUNTERS - PERFORMANCE

- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times

| Threads | Precise (seconds) | Sloppy (seconds) |
|---------|-------------------|------------------|
| 1       | ~0.5              | ~0.2             |
| 2       | ~5.5              | ~0.3             |
| 3       | ~9.5              | ~0.4             |
| 4       | ~12.5             | ~0.5             |

Traditional vs. sloppy counter  
Sloppy Threshold (S) = 1024

**Synchronized counter scales poorly.**

April 23, 2018

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

L7.42

PERFECT SCALING

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

April 23, 2018

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

L7.43

SLOPPY COUNTER

- Provides single logical shared counter
  - Implemented using local counters for each ~CPU core
    - 4 CPU cores = 4 local counters & 1 global counter
    - Local counters are synchronized via local locks
  - Global counter is updated periodically
    - Global counter has lock to protect global counter value
    - Sloppiness threshold (S):
      - Update threshold of global counter with local values
      - Small (S): more updates, more overhead
      - Large (S): fewer updates, more performant, less synchronized
- Why this implementation?  
Why do we want counters local to each CPU Core?

April 23, 2018

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

L7.44

SLOPPY COUNTER - 2

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

| Time | L <sub>1</sub> | L <sub>2</sub> | L <sub>3</sub> | L <sub>4</sub> | 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 <sub>1</sub> )  |
| 7    | 0              | 2              | 4              | 5 → 0          | 10 (from L <sub>4</sub> ) |

April 23, 2018

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

L7.45

THRESHOLD VALUE S

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

April 23, 2018

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

L7.46

SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

April 23, 2018

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

L7.47

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.)
```

April 23, 2018

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

L7.48



## CONCURRENT LINKED LIST - 2

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

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

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.49

## 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 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 }
```

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.50

## 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 ...

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.51

## 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
```

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.52

## 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 }
```

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.53

## 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?



April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.54

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

April 23, 2018

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

L7.55

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.)
```

April 23, 2018

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

L7.56

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.)
```

April 23, 2018

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

L7.57

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

April 23, 2018

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

L7.58

INSERT PERFORMANCE -  
CONCURRENT HASH TABLE

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

| Inserts (Thousands) | Simple Concurrent List (seconds) | Concurrent Hash Table (seconds) |
|---------------------|----------------------------------|---------------------------------|
| 10                  | 1.0                              | 0.5                             |
| 20                  | 2.5                              | 0.5                             |
| 30                  | 5.0                              | 0.5                             |
| 40                  | 12.0                             | 0.5                             |

The simple concurrent hash table scales magnificently.

April 23, 2018

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

L7.59

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 }
```

April 23, 2018

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

L7.60

## LOCK-FREE DATA STRUCTURES

- Lock-free data structures in Java
- `Java.util.concurrent.atomic` package
- Classes:
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicIntegerArray`
  - `AtomicIntegerFieldUpdater`
  - `AtomicLong`
  - `AtomicLongArray`
  - `AtomicLongFieldUpdater`
  - `AtomicReference`
- See: <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html>

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.61

## QUESTIONS



## FUTEX: MUTEX\_LOCK PSUEDO CODE

```
void mutex_lock(int *mutex) {
 int v;
 /* Bit 31 was clear, we got the mutex (this is a fast lock!)
 if (atomic_bit_test_set (mutex, 31) == 0)
 return;
 // "adds" mutex to queue
 atomic_increment (mutex);
 while (1) {
 // is lock available?
 if (atomic_bit_test_set (mutex, 31) == 0 {
 // remove mutex from queue - it has the lock now
 atomic_decrement (mutex);
 return;
 }
 // Have to wait. Make sure futex value is locked (negative)
 v = *mutex;
 if (v >= 0)
 continue;
 // wait to be woken up when lock is available
 // this is not a spin lock... (signal)
 futex_wait (mutex, v);
 }
}
```

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.63

## FUTEX: MUTEX\_UNLOCK PSUEDO CODE

```
void mutex_unlock(int *mutex) {
 // Adding 0x80000000 to counter results in 0 if and only if
 // there are no other interested threads
 if (atomic_add_zero (mutex, 0x80000000))
 return;
 // There are other threads waiting for this lock (mutex)
 // wake one of them up..
 // (e.g. dequeue it)
 futex_wake (mutex);
}
```

- Interesting note: Futex bug in Redhat Linux
- <https://www.infoq.com/news/2015/05/redhat-futex>

April 23, 2018

TCS5422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L7.64