TCSS 422: OPERATING SYSTEMS

Linux Thread API II, Locks, Lock-based data structures

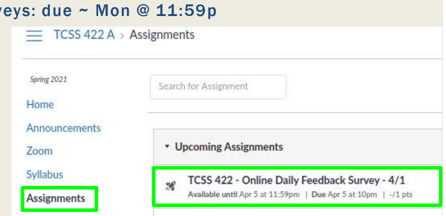
Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

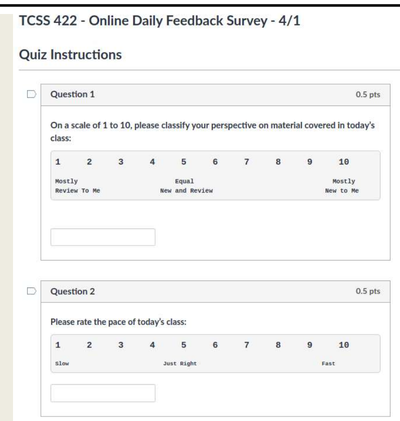
OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

ONLINE DAILY FEEDBACK SURVEY

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





MATERIAL / PACE

- Please classify your perspective on material covered in today's class (57 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.56 (↓ - previous 6.90)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.73 (↑ - previous 5.52)**

FEEDBACK

- ***Does using mutex cause problems with running things in parallel?***
  - In C, locks are called pthread_mutex
  - Using mutex (locks) in fact **SOLVES** problems running things in parallel
  - Locks synchronize access to critical sections of code that MODIFY shared variables
  - If these sections ARE NOT SYNCHRONIZED this leads to RACE CONDITIONS, and the intended changes to your variables may not be SAVED
  - These can lead to program errors and bugs are varying severity
  - In particular these errors can be hidden and hard to see: Realizing data is corrupted can often be hard
- ***I have to re-watch some of the lectures, some things were not making any sense at all.***
  - Please do ask any questions if/when they arise..

FEEDBACK - 2

Duplicate Question From Previous Class ...

- ***Would you review about Linux nice-value?***
  - `Nice/renice` command is used to influence a job's priority in Linux
  - `Nice` predates the CFS scheduler
  - Top shows `nice` values
  - `Nice` vals w/ ps: `ps ax -o pid,ni,cmd,%cpu, pri`
  - `Nice` values: -20 (HIGH priority) to 19 (LOW priority)
  - Default value is 0
  - Nice value influences the vruntime value of a job
  - `vruntime` is a weighted time measurement
  - Linux process priority weights the calculation of `vruntime` within a runqueue to impact the priority of a job (+ / -)
    - Influences job's position in rb-tree
  - Nice is used to launch a new job with a priority adjustment
  - Renice is used to adjust priority of an existing job

NICE / RENICE

- Find PID for VirtualBox

`ps ax -o pid,ni,cmd,%cpu,pri | grep virtualbox`

- Monitor process priority in top

top -d .1

- Adjust process priority using renice:

 # High priority
sudo renice –n -20 –p <pid>

 # Default priority
sudo renice –n 0 –p <pid>

 # Low priority
sudo renice –n 19 –p <pid>

FEEDBACK - 2

- ***How do lottery or stride schedulers optimize a job's response time and turnaround time?***
- These schedulers are designed to distribute time to jobs based on the number of tickets a job has
- The user is responsible for assigning tickets
- Resource sharing will mimic round-robin scheduling if all jobs have the exact same number of tickets
  - Stride scheduler will achieve round-robin like fairness more quickly
  - Lottery scheduler requires more scheduling events & time
- The round robin scheduler is excellent at job response time
  - Each job shares the resource for a fixed time quantum
- Round robin schedulers may perform poorly with respect to job turnaround time
  - The user could adjust the job's # of tickets to improve the outcome

OBJECTIVES – 4/29

- Questions from 4/27
- **C Tutorial - Pointers, Strings, Exec in C**
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- **Assignment 1**
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- **Quiz 1 – Active Reading Chapter 9**
- Quiz 2 – CPU Scheduling Algorithms
- 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

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Friday April 30th at 11:59pm
- Grace period til Sunday May 2nd at 11:59 ** AM **
- Late submissions til Tuesday May 4th at 11:59pm
- **Link:**
- **http://faculty.washington.edu/wlloyd/courses/tcss422/TCSS422_s2021_quiz_1.pdf**

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- **Quiz 2 – CPU Scheduling Algorithms**
- 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

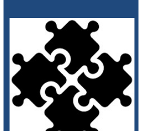
QUIZ 2 - CPU SCHEDULING ALGORITHMS

- Quiz posted on Canvas
- Due Wednesday May 5 @ 11:59p
- Provides CPU scheduling practice problems
  - FIFO, SJF, STCF, RR, MLFQ (Ch. 7 & 8)
- Unlimited attempts allowed
- Multiple choice and fill-in the blank
- Quiz automatically scored by Canvas
  - Please report any grading problems

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

CHAPTER 27 - LINUX THREAD API



CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- pthread_cont_t datatype
- pthread_cond_wait()
  - Puts thread to "sleep" (waits) (THREAD is BLOCKED)
  - Threads added to >**FIFO queue**<, lock is released
  - Waits ***(listens)*** for a "signal" (NON-BUSY WAITING, no polling)
  - When signal occurs, interrupt fires, wakes up first thread, (THREAD is RUNNING), lock is provided to thread

CONDITIONS AND SIGNALS - 2

```
int pthread_cond_signal(pthread_cond_t * cond);
int pthread_cond_broadcast(pthread_cond_t * cond);
```

- pthread_cond_signal()
  - Called to send a "signal" to wake-up first thread in **FIFO "wait" queue**
  - The goal is to unblock a thread to respond to the signal
- pthread_cond_broadcast()
  - Unblocks ***all*** threads in **FIFO "wait" queue**, currently blocked on the specified condition variable
  - Broadcast is used when all threads should wake-up for the signal
- Which thread is unblocked first?
  - Determined by OS scheduler (based on priority)
  - Thread(s) awoken based on placement order in **FIFO wait queue**
  - When awoken threads acquire lock as in pthread_mutex_lock()

CONDITIONS AND SIGNALS - 3

- Wait example:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&cond, &lock);
// Perform work that requires lock
a = a + b;
pthread_mutex_unlock(&lock);
```

- wait puts thread to sleep, releases lock
- when awoken, lock reacquired (but then released by this code)
- When initialized, another thread signals

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

State variable set, Enables other thread(s) to proceed above.

CONDITION AND SIGNALS - 4

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&cond, &lock);
// Perform work that requires lock
a = a + b;
pthread_mutex_unlock(&lock);
```

- Why do we wait inside a while loop?
- The while ensures upon awakening the condition is rechecked
  - A signal is raised, but the pre-conditions required to proceed may have not been met. **MUST CHECK STATE VARIABLE**
  - Without checking the state variable the thread may proceed to execute when it should not. (e.g. too early)

PTHREADS LIBRARY

- Compilation:
  gcc requires special option to require programs with pthreads:
  - gcc –pthread pthread.c –o pthread
  - Explicitly links library with compiler flag
  - RECOMMEND: using makefile to provide compiler arguments
- List of pthread manpages
  - man –k pthread

SAMPLE MAKEFILE

```
CC=gcc
CFLAGS=-pthread -I. -Wall

binaries=pthread pthread_int pthread_lock_cond pthread_struct

all: $(binaries)

pthread_mult: pthread.c pthread_int.c
    $(CC) $(CFLAGS) $^ -o $@

clean:
    $(RM) -f $(binaries) *.o
```

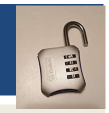
- Example builds multiple single file programs
  - All target
- pthread_mult
  - Example if multiple source files should produce a single executable
- clean target

CHAPTER 28 – LOCKS

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

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);
```


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 – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

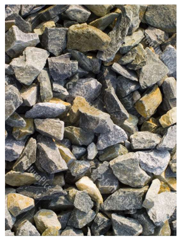
LOCKS - 4

- 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 . . .
```

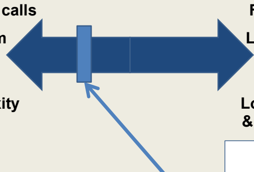
LOCK GRANULARITY TRADE-OFF SPACE

FINE-GRAINED

Many Lock (kernel) calls
More overhead from excessive locking
More parallelism
Higher code complexity & debugging

COARSE-GRAINED

Few Lock (kernel) calls
Low overhead from minimal locking
Less parallelism
Low code complexity & simpler debugging



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

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**

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

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…

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

SPIN LOCK IMPLEMENTATION

- Operate without atomic-*as a unit* assembly instructions
- "Do-it-yourself" Locks
- Is this lock implementation: ***(1)Correct? (2)Fair? (3)Performant?***

```
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;
}
```


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

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…

WE WILL RETURN AT 4:54PM

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

TEST-AND-SET INSTRUCTION

- Hardware support required for working locks
- Book presents pseudo code of C implementation
  - TEST-and-SET adds a simple check to the basic spin lock
  - Assumption is this "C code" runs atomically on CPU:

```
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
}
```

- lock() method checks that TestAndSet doesn't return 1
- Comparison is in the caller

- Can implement the C version (non-atomic) and have some success on a single-core VM

DIY: TEST-AND-SET - 2

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

```
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;
}
```


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…

- **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

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

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
  - 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 the HW support CompareAndSwap instruction
  - Shared data structure updates become "wait-free"
  - Upcoming in Chapter 32

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 loc

```
            ; // spin
}
```

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

- X86 provides "`cmpxchgl`" compare-and-exchange instruction
  - `cmpxchg8b`
  - `cmpxchg16b`

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

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

LL/SC LOCK

```
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

```
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

CHAPTER 29 – LOCK BASED DATA STRUCTURES

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

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

```
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

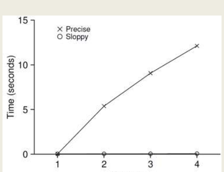
- Decrease counter
- Get value

```
(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

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



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

Synchronized counter scales poorly.

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)

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

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?

SLOPPY COUNTER – MAIN POINTS

- Idea of Sloppy 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
- Sloppy counter: trade-off accuracy for speed
  - It's sloppy because it's not so accurate (until the end)

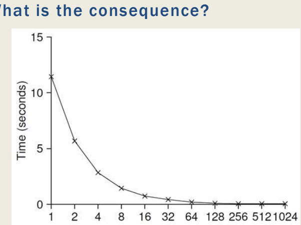
SLOPPY COUNTER - 2

- 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)

THRESHOLD VALUE S

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



SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

OBJECTIVES – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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 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.)
```


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.)
```


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 – 4/29

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- 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

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.)
```


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 – 4/29

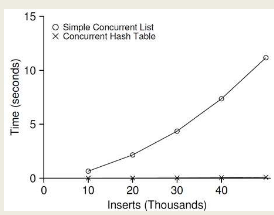
- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin Locks, Test 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);
}
```

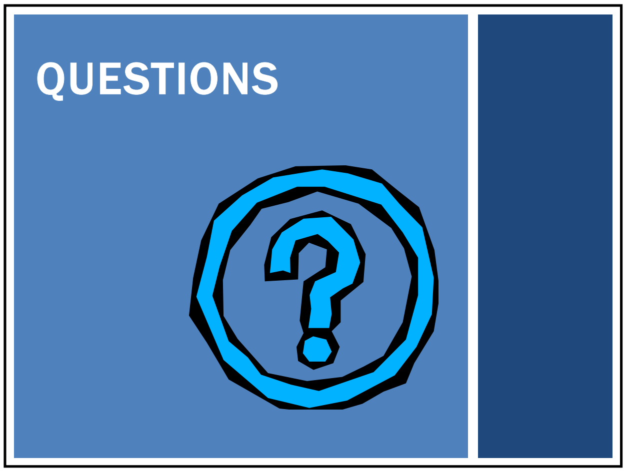

Which is a major advantage of using concurrent data structures in your programs?

- 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

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


QUESTIONS