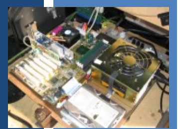
TCSS 422: OPERATING SYSTEMS



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

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

1

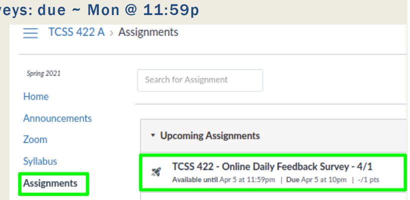
OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

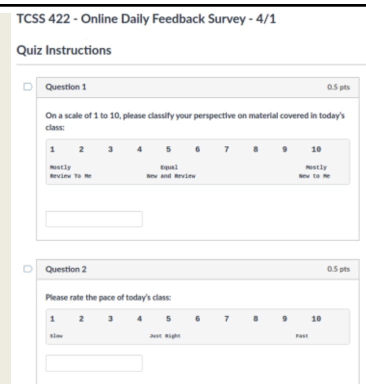
2

ONLINE DAILY FEEDBACK SURVEY

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



3



4

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (40 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 7.30 (↓ - previous 7.44)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.70 (↓ - previous 5.84)**

5

FEEDBACK FROM 4/25

- ***In what situations would it be useful to provide attributes different from the defaults for pthread_create?***
- pthread_attr_init() initializes a pthread attribute type
- The type is pthread_attr_t
- Once the type is initialized, there is an API to configure the attr record
- API functions:
- pthread_attr_setaffinity_np(), pthread_attr_setdetachstate(), pthread_attr_setguardsize(), pthread_attr_setinheritsched(), pthread_attr_setschedparam(), pthread_attr_setschedpolicy(), pthread_attr_setscope(), pthread_attr_setstack(), pthread_attr_setstackaddr(), pthread_attr_setstacksize(), pthread_getattr_np(), pthread_setattr_default_np()
- See man pages for more info on this API

6

FEEDBACK - 2

- ***In what situations would it be useful to provide attributes different from the defaults for pthread_mutex_init?***
- *From the man pages:*
- pthread_mutexattr_init() initializes a mutex attribute type
- The type is pthread_mutexattr_t
- LinuxThreads supports only one mutex attribute: the mutex **kind**
  - PTHREAD_MUTEX_FAST_NP for 'fast' mutexes
  - PTHREAD_MUTEX_RECURSIVE_NP for 'recursive' mutexes
  - PTHREAD_MUTEX_ERRORCHECK_NP for 'error checking' mutexes
  - NP indicates these are non-portable extensions to the standard.
- The kind determines what happens if a thread attempts to lock a mutex it already owns with pthread_mutex_lock().
- The default mutex kind is PTHREAD_MUTEX_FAST_NP ('fast').

7

FEEDBACK - 3

- ***(CONTINUED) In what situations would it be useful to provide attributes different from the defaults for pthread_mutex_init?***
- If the mutex is of the '**fast**' kind, pthread_mutex_lock() simply suspends the calling thread forever.
- If the mutex is of the '**recursive**' kind, the call to pthread_mutex_lock() returns immediately with a success return code. The number of times the thread owning the mutex has locked it is recorded in the mutex. The owning thread must call pthread_mutex_unlock() the same number of times before the mutex returns to the unlocked state.
- If the mutex is of the '**error checking**' kind, pthread_mutex_lock() returns immediately with the error code EDEADLK.

8

FEEDBACK - 4

- ***With different purposes, each type of thread will be used in different scenarios, isn't it?***
- ***REINTERPRETATION :***
- ***For what different purposes will each type of thread be used in?***
  - Not clear what is meant by "each type of thread"
  - Is this referring to threads vs. processes?
  - Is this referring to threads with different INIT attributes?

9

OBJECTIVES – 4/27

- Questions from 4/25
- **C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28**
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

10

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- **Assignment 1 - Due Tue May 9**
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

11

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- **Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)**
- 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

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Thursday April 27th at 11:59pm
- **Link:**
- **https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_s2023_quiz_1.pdf**

13

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 May 2nd at 11:59pm
- **Link:**
- **https://canvas.uw.edu/courses/1642522/assignments/8316759**

14

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

15

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

16

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()

17

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

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

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


18

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)

19

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
  - May need to install Linux package for pthread API documentation:
  - `sudo apt install glibc-doc`

20

SAMPLE MAKEFILE

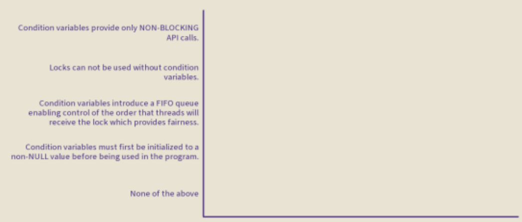
```
CC=gcc
CFLAGS=-pthread -I. -Wall

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

21

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

What key feature differentiates condition variables from mutex_locks in C ?



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

22

OBJECTIVES – 4/27

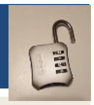
- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 27: Linux Thread API
  - pthread_cond_wait/_signal/_broadcast
- Chapter 28: Locks
  - Introduction, Lock Granularity
  - Spin 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

CHAPTER 28 – LOCKS

24

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);
```

April 27, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L10.25

25

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 27, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L10.26

26

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 27, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L10.27

27

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

April 27, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L10.28

28

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 27, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L10.29

29

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 27, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L10.30

30

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 . . .
```

31

LOCK GRANULARITY TRADE-OFF SPACE

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

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

32

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**

33

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

34

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…

35

WE WILL RETURN AT 4:55PM

36

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

37

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?***

```
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;
}
```


38

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

39

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…

40

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

41

TEST-AND-SET INSTRUCTION

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

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


42

TEST-AND-SET - 2

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

```c
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;
}
```


43

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

44

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

45

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

46

COMPARE AND SWAP

- Compare and Swap

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

- Spin l

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

```c
        ; // spin
}
```

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

47

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

48

"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

49

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

50

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

51

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

52

CHAPTER 29 – LOCK BASED DATA STRUCTURES

53

LOCK-BASED CONCURRENT DATA STRUCTURES

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

54

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;
}
```


55

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

56

CONCURRENT COUNTER - 2

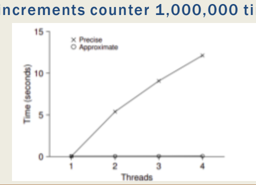
- Decrease counter
- Get value

```
(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;
        }
```


57

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.

58

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?***

59

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

60

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?

61

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)

62

APPROXIMATE COUNTER - 2

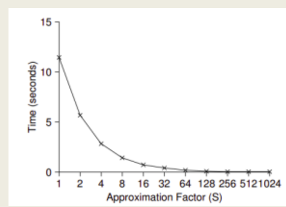
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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)

63

THRESHOLD VALUE S

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



64

APPROXIMATE COUNTER - EXAMPLE

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

65

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?

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

66

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

67

CONCURRENT LINKED LIST - 1

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

```c
// 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.)
```


68

CONCURRENT LINKED LIST - 2

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

```c
(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.)
```


69

CONCURRENT LINKED LIST - 3

- Lookup – checks list for existence of item with key
- Once again everything is critical
  - Note - there are also two unlocks

```c
(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
}
```


70

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 …

71

CCL – SECOND IMPLEMENTATION

- Init and Insert

```c
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);
}
```


72

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
      }
```


73

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?

74

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

75

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

76

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.)
```


77

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.)
```


78

OBJECTIVES – 4/27

- Questions from 4/25
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- 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

79

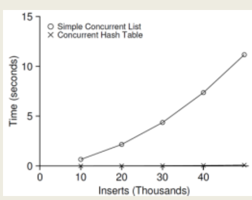
CONCURRENT HASH TABLE

- 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

80

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.

81

CONCURRENT HASH TABLE

```c
#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);
}
```


82

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

83

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

84

QUESTIONS

85