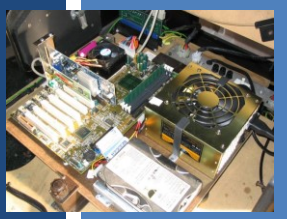
TCSS 422: OPERATING SYSTEMS



Intro to Concurrency, Linux Thread API, Locks, Lock-based data structures

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

1

TCSS 422 – OFFICE HRS – SPRING 2025

- **Office Hours plan for Spring (by Zoom):**
- **Monday 11:30am - 12:30p GTA Xinghan**
- **Tuesday 11:30am - 12:30p GTA Xinghan**
- **Wednesday 11:00am - 12:00p Instructor Wes**
- **Friday 12:00pm - 1:00p Instructor Wes or GTA Xinghan**
  - **THIS FRIDAY: Wes**
- **Instructor is available after class at 6pm in CP 229 each day**

2

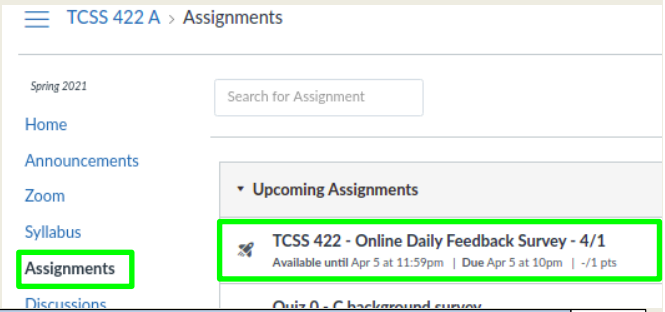
OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

3

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



4

TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 0.5 pts

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

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

Question 2 0.5 pts

Please rate the pace of today's class:

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

5

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (45 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.33 (↓ - previous 6.55)**

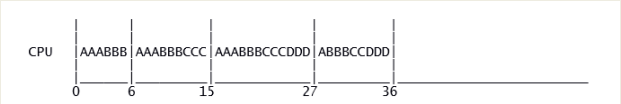
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.11 (↓ - previous 5.19)**

6

FEEDBACK FROM 4/24

- ***Two questions were received on Question 2 from the Quiz.***
- The starter graph should have been drawn as:

- For this RR queue, when jobs arrive, they are added at the back of the runqueue. This RR does not context switch when a new job arrives, but continues to run each job for the 3-sec time slice.
- Graph solution: *note the line after each RR-cycle*



- *All students received credit "CR" for attempting Q2 & Q3*

8

OBJECTIVES – 4/29

- Questions from 4/24
- **C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE**
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

9

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- **Assignment 0 - Closes Tue Apr 29 AOE** **Assignment 1**
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

10

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- **Assignment 0 - Closes Tue Apr 29 AOE** | **Assignment 1**
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

11

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- **Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)**
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

12

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Thursday May 1st AOE
- **Link:**
- **https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_s2025_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 6th AOE

- Link:
- https://canvas.uw.edu/courses/1809484/assignments/10329061

14

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- **Chapter 26: Concurrency: An Introduction**
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

15

CATCH UP FROM LECTURE 8

- Switch to Lecture 8 Slides
- Slides L8.27 to L8.38 (through Critical Section)

16

LOCKS

- To demonstrate how critical section(s) can be executed "atomically-*as a unit*" Chapter 27 & beyond introduce **LOCKS**

```
1    lock_t mutex;
2    . . .
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```

Critical section

- **(DEMO)** Counter example revisited

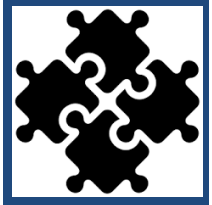
17

COUNTER EXAMPLE

- With locks
  - 2 threads count to 16 million
  - ~1.4 seconds
  - COUNT IS CORRECT – no data loss
- Without locks
  - 2 threads count to 16 million
  - ~0.03 seconds
  - COUNT IS INCORRECT - DATA IS LOST
- Correct version is 46.6 x slower
  - Cost is ~16 million Lock & Unlock API calls

18

CHAPTER 27 - LINUX THREAD API



19

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - **pthread_create/_join**
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

20

THREAD CREATION

- pthread_create

```
#include <pthread.h>

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

- thread: thread struct
- attr: stack size, scheduling priority… (*optional*)
- start_routine: function pointer to thread routine
- arg: argument to pass to thread routine (*optional*)

21

PTHREAD_CREATE – PASS ANY DATA

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```


22

PASSING A SINGLE VALUE

Using this approach on your Ubuntu VM, How large (in bytes) can the primitive data type be?

How large (in bytes) can the primitive data type be on a 32-bit operating system?

```
9       int rc, m;
10      pthread_create(&p, NULL, mythread, (void *) 100);
11      pthread_join(p, (void **) &m);
12      printf("returned %d\n", m);
13      return 0;
14  }
```


23

WAITING FOR THREADS TO FINISH

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

- thread: which thread?
- value_ptr: pointer to return value
  type is dynamic / agnostic

- Returned values *must* be on the heap
- Thread stacks destroyed upon thread termination (join)
- Pointers to thread stack memory addresses are invalid
  - May appear as gibberish or lead to crash (seg fault)
- Not all threads join – ***What would be Examples ??***

24

What will this code do?

```
struct myarg {
  int a;
  int b;
};

void *worker(void *arg)
{
  struct myarg *input = (struct myarg *) arg;
  printf("a=%d b=%d\n",input->a, input->b);
  struct myarg output;
  output.a = 1;
  output.b = 2;
  return (void *) &output;
}

int main (int argc, char * argv[])
{
  pthread_t p1;
  struct myarg args;
  struct myarg *ret_args;
  args.a = 10;
  args.b = 20;
  pthread_
  pthread_
  printf("
  return 0;
}
```

Data on thread stack

```
$ ./pthread_struct
a=10 b=20
Segmentation fault (core dumped)
```

How can this code be fixed?

25

How about this code?

```
struct myarg {
  int a;
  int b;
};

void *worker(void *arg)
{
  struct myarg *input = (struct myarg *) arg;
  printf("a=%d b=%d\n",input->a, input->b);
  input->a = 1;
  input->b = 2;
  return (void *) &input;
}

int main (int argc, char * argv[])
{
  pthread_t p1;
  struct myarg args;
  struct myarg *ret_args;
  args.a = 10;
  args.b = 20;
  pthread_create(&p1, NULL, worker, &args);
  pthread_join(p1, (void *)&ret_args);
  printf("returned %d %d\n", ret_args->a, ret_args->b);
  return 0;
}
```

```
$ ./pthread_struct
a=10 b=20
returned 1 2
```


ADDING CASTS

- Casting
- Suppresses compiler warnings when passing "typed" data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join

```
pthread_int.c: In function 'main':
pthread_int.c:34:20: warning: passing argument 2 of 'pthread_join'
from incompatible pointer type [-Wincompatible-pointer-types]
   pthread_join(p1, &p1val);
```

- Example: uncasted return

```
In file included from pthread_int.c:3:0:
/usr/include/pthread.h:250:12: note: expected 'void **' but argument
is of type 'int **'
 extern int pthread_join (pthread_t __th, void **__thread_return);
```

ADDING CASTS - 2

- pthread_join

```
int * p1val;
int * p2val;
pthread_join(p1, (void *)&p1val);
pthread_join(p2, (void *)&p2val);
```

- return from thread function

```
int * counterval = malloc(sizeof(int));
*counterval = counter;
return (void *) counterval;
```


28

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - **pthread_mutex_lock/_unlock/_trylock/_timelock**
  - 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

29

LOCKS

- pthread_mutex_t data type
- /usr/include/bits/pthread_types.h

```
// Global Address Space
static volatile int counter = 0;
pthread_mutex_t lock;

void *worker(void *arg)
{
  int i;
  for (i=0;i<10000000;i++)  {
    int rc = pthread_mutex_lock(&lock);
    assert(rc==0);
    counter = counter + 1;
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```


30

LOCKS - 2

- Ensure critical sections are executed atomically-*as a unit*
  - Provides implementation of "***Mutual Exclusion***"
- API

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Example w/o initialization & error checking

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

  - Blocks forever until lock can be obtained
  - Enters critical section once lock is obtained
  - Releases lock

31

LOCK INITIALIZATION

- Assigning the constant

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- API call:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

- Initializes mutex with attributes specified by 2nd argument
- If NULL, then default attributes are used
- Upon initialization, the mutex is initialized and unlocked

32

LOCKS - 3

- Error checking wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- What if lock can't be obtained?

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- trylock – returns immediately (fails) if lock is unavailable
- timelock – tries to obtain a lock for a specified duration

33

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

Which NON-BLOCKING API call can be used to obtain a lock without BLOCKING the calling thread?

- pthread_mutex_lock()
- pthread_mutex_unlock()
- pthread_join()
- pthread_mutex_trylock()
- 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**

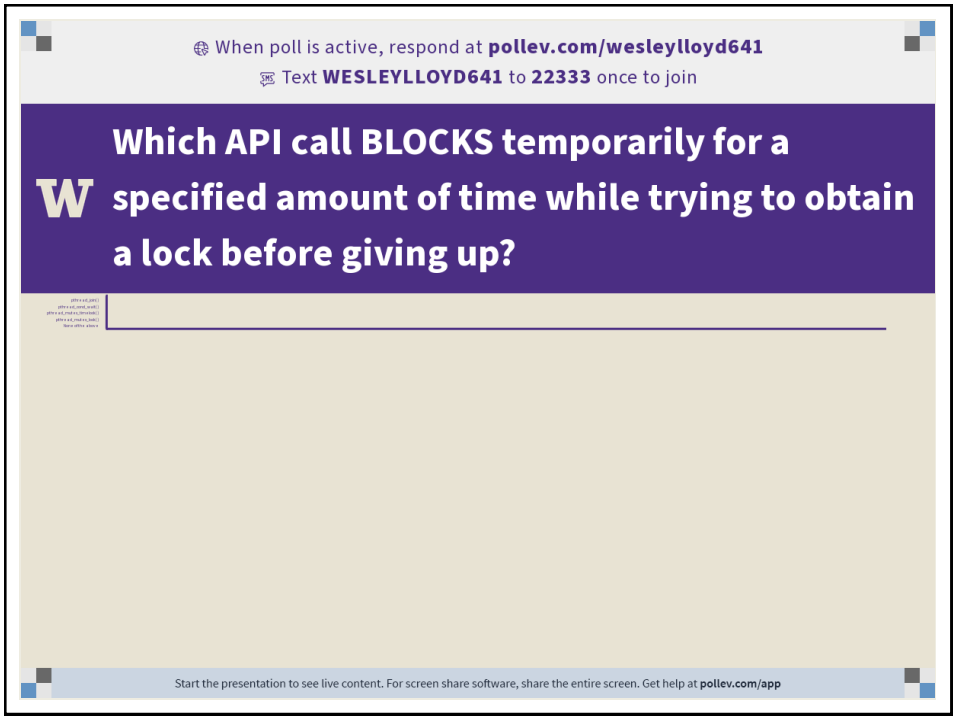
34

POLL EV

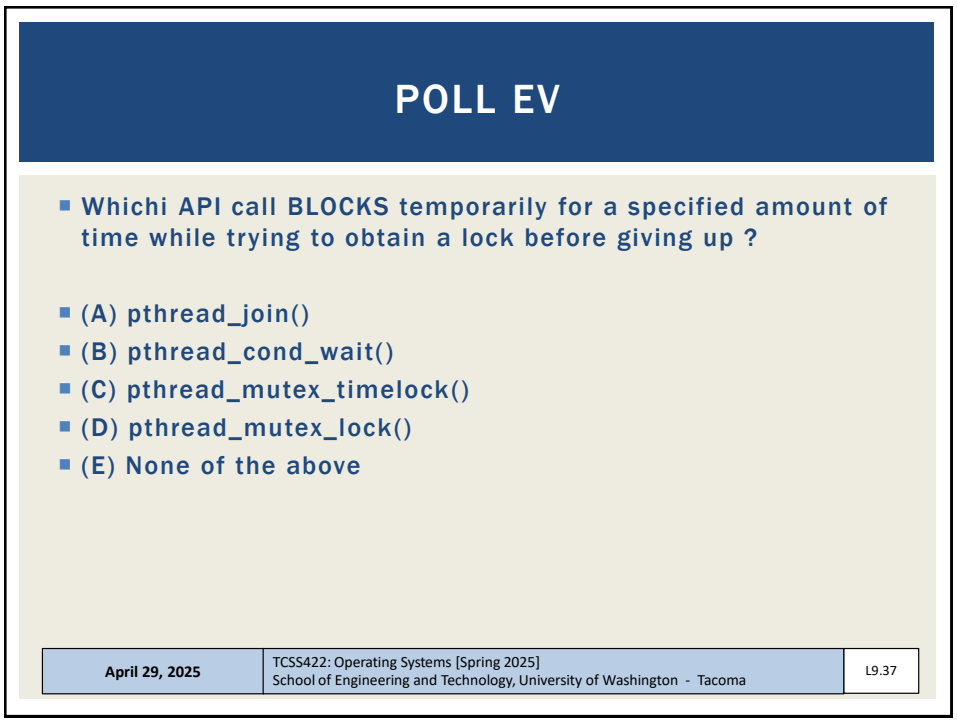
- Which NON-BLOCKING API call can be used to obtain a lock without BLOCKING the calling thread ?

- (A) pthread_mutex_lock()
- (B) pthread_mutex_unlock()
- (C) pthread_join()
- (D) pthread_mutex_trylock()
- (E) None of the above

35


When poll is active, respond at pollev.com/wesleylloyd641
Text WESLEYLLOYD641 to 22333 once to join
Which API call BLOCKS temporarily for a specified amount of time while trying to obtain a lock before giving up?
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

36


POLL EV
Whichi API call BLOCKS temporarily for a specified amount of time while trying to obtain a lock before giving up ?
(A) pthread_join()
(B) pthread_cond_wait()
(C) pthread_mutex_timelock()
(D) pthread_mutex_lock()
(E) None of the above
April 29, 2025
TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L9.37

37

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - **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

38

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_cond_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

39

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()

40

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);
```


41

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)

42

WE WILL RETURN AT 5:00PM

43

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

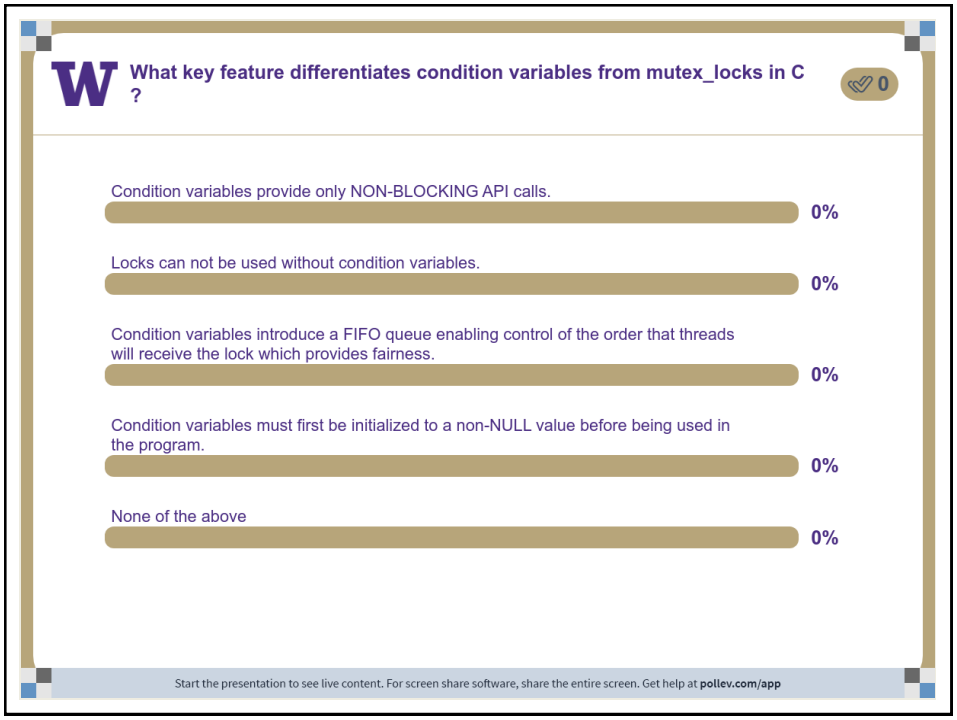
44

SAMPLE MAKEFILE

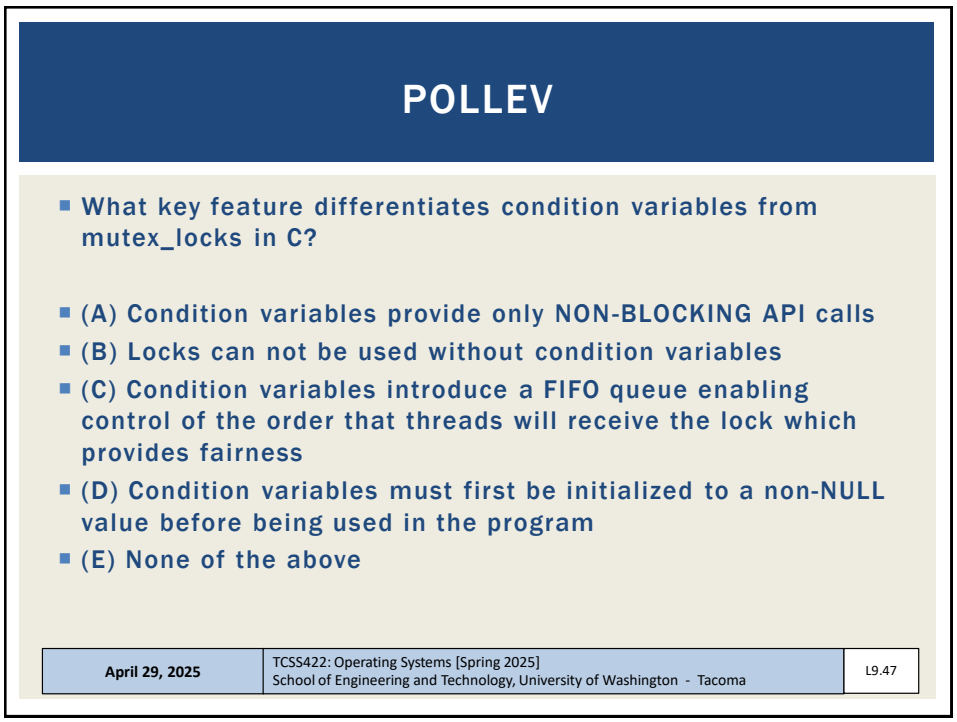
```
CC=gcc
CFLAGS=-pthread -I. -Wall

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

45


What key feature differentiates condition variables from mutex_locks in C ?
0
Condition variables provide only NON-BLOCKING API calls.
0%
Locks can not be used without condition variables.
0%
Condition variables introduce a FIFO queue enabling control of the order that threads will receive the lock which provides fairness.
0%
Condition variables must first be initialized to a non-NULL value before being used in the program.
0%
None of the above
0%
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

46


POLLEV
What key feature differentiates condition variables from mutex_locks in C?
(A) Condition variables provide only NON-BLOCKING API calls
(B) Locks can not be used without condition variables
(C) Condition variables introduce a FIFO queue enabling control of the order that threads will receive the lock which provides fairness
(D) Condition variables must first be initialized to a non-NULL value before being used in the program
(E) None of the above
April 29, 2025
TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L9.47

47

CHAPTER 28 – LOCKS

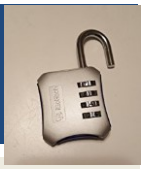
48

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

49

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);
```


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.

52

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

53

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

54

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);
```

55

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 . . .
```

56

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…

57

EVALUATING LOCK IMPLEMENTATIONS

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

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

```
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…

60

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

61

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?***

```
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  }
```


62

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

63

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…

64

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

65

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:

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

66

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

```
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  }
```


67

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

68

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

69

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

70

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 loc

```
1
2
3               ; // spin
4   }
```

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

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

71

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**

72

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

73

LL/SC LOCK

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

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

74

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

75

CHAPTER 29 – LOCK BASED DATA STRUCTURES

76

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

77

LOCK-BASED CONCURRENT DATA STRUCTURES

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

78

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  }
```


79

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

80

CONCURRENT COUNTER - 2

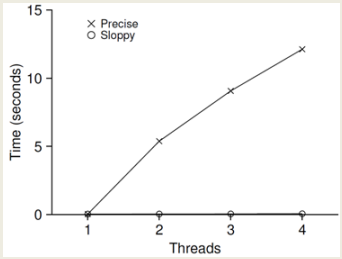
- Decrease counter
- Get value

```
(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      }
```


81

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 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

84

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?

85

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)

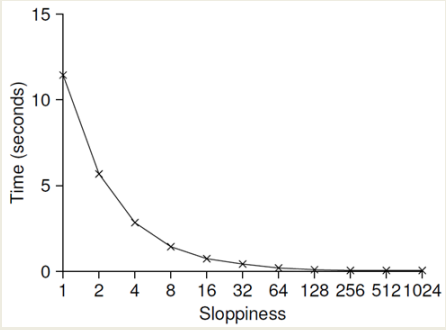
SLOPPY COUNTER - 2

- 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?



88

SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

89

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

90

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.)
```


91

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 }
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.)
```


CONCURRENT LINKED LIST - 3

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

```
(Cont.)
32
32      int List_Lookup(list_t *L, int key) {
33              pthread_mutex_lock(&L->lock);
34              node_t *curr = L->head;
35              while (curr) {
36                      if (curr->key == key) {
37                              pthread_mutex_unlock(&L->lock);
38                              return 0; // success
39                      }
40                      curr = curr->next;
41              }
42              pthread_mutex_unlock(&L->lock);
43              return -1; // failure
44      }
```

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 …

94

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
```


95

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      }
```


96

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?

97

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

```
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.)
```


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.)
```

OBJECTIVES – 4/29

- Questions from 4/24
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Closes Tue Apr 29 AOE | Assignment 1
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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**

102

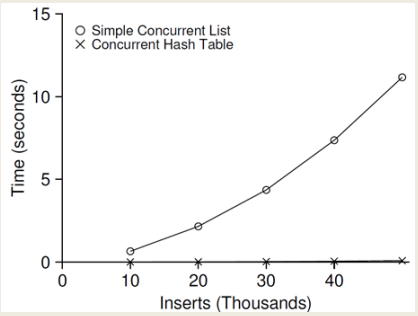
CONCURRENT HASH TABLE

- 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

103

INSERT PERFORMANCE – CONCURRENT HASH TABLE

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



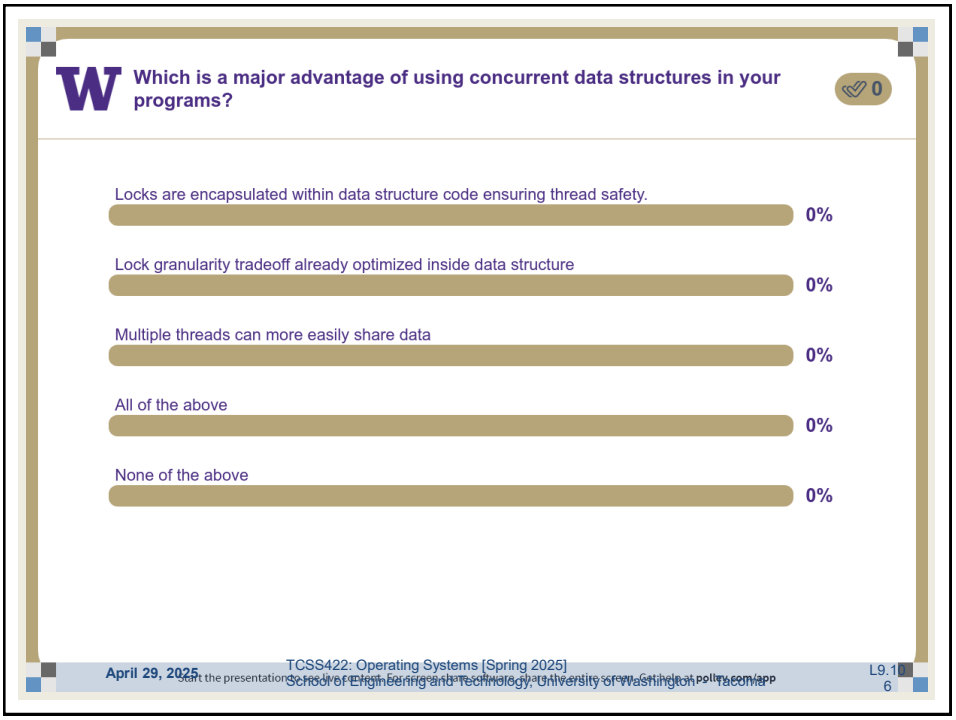
The simple concurrent hash table scales magnificently.

104

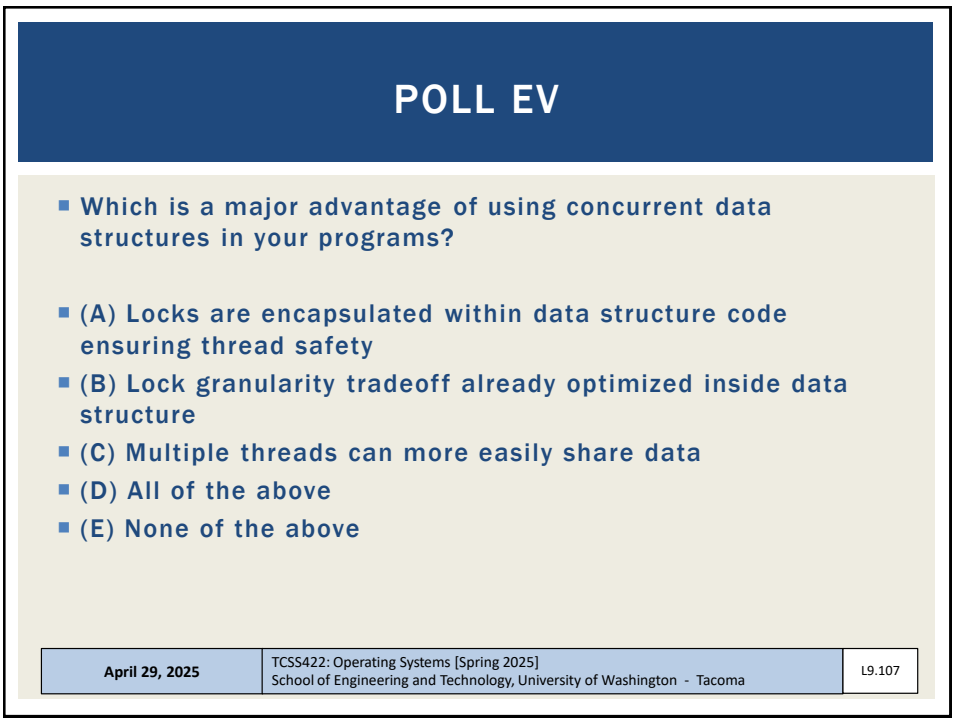
CONCURRENT HASH TABLE

```
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      }
```


105


Which is a major advantage of using concurrent data structures in your programs?
0
Locks are encapsulated within data structure code ensuring thread safety.
0%
Lock granularity tradeoff already optimized inside data structure
0%
Multiple threads can more easily share data
0%
All of the above
0%
None of the above
0%
April 29, 2025
TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L9.106

106


POLL EV
Which is a major advantage of using concurrent data structures in your programs?
(A) Locks are encapsulated within data structure code ensuring thread safety
(B) Lock granularity tradeoff already optimized inside data structure
(C) Multiple threads can more easily share data
(D) All of the above
(E) None of the above
April 29, 2025
TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L9.107

107

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

108

QUESTIONS

109