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

TCSS 422 – OFFICE HRS – WINTER 2026

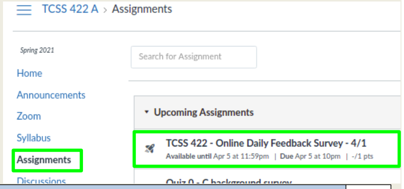
- **Office Hours plan for Winter:**
- Tuesday 2:30 - 3:30 pm Instructor Wes, Zoom
- Tue/Thur 6:00 - 7:00 pm Instructor Wes, CP 229/Zoom
- Tue 6:00 – 7:00 pm GTA Robert, Zoom/MDS 302
- Wed 1:00 – 2:00 pm GTA Robert, Zoom/MDS 302
- Instructor is available after class at 6pm in CP 229 each day

OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

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



TCSS 422 A › Assignments

Spring 2021

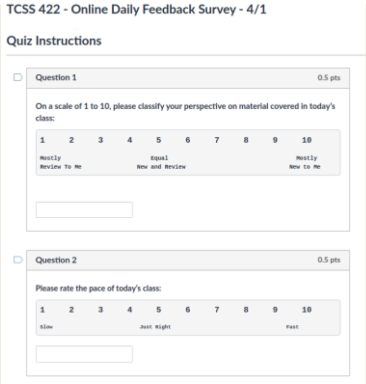
Home
Announcements
Zoom
Syllabus
Assignments
Discussions

Search for Assignment

▾ Upcoming Assignments

TCSS 422 - Online Daily Feedback Survey - 4/1
Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts

Quiz 0 - C background survey



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              Equal                 Mostly
Review To Me     New and Review          New to Me
```

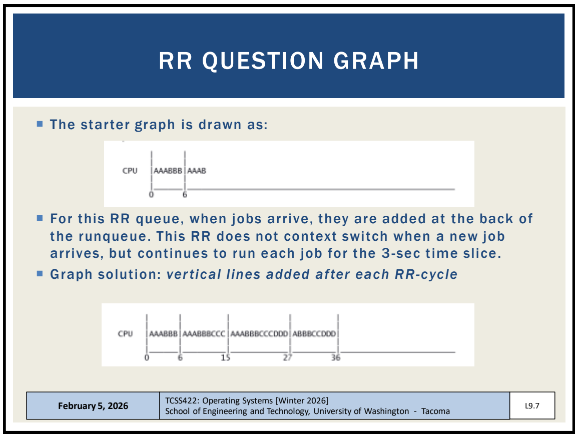
Question 2 0.5 pts

Please rate the pace of today's class:

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

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (36 of 46 respondents (8 online) – 78.3%):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 5.92 (↑ - previous 5.75)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.03 (*tie* - previous 5.03)**



RR QUESTION GRAPH

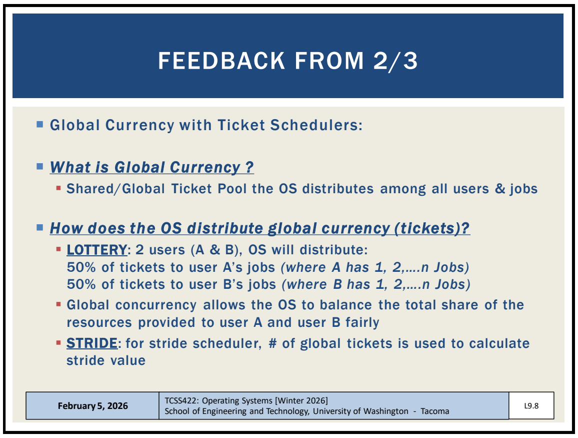
- The starter graph is drawn as:

CPU
AAABBB
AAAB
0
6

- 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: *vertical lines added after each RR-cycle*

CPU
AAABBB
AAABBBCCC
AAABBBCCCDDD
ABBBCCDDD
0
6
15
27
36

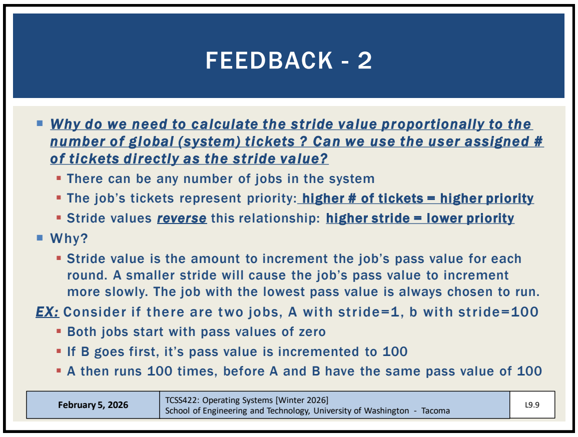
7



FEEDBACK FROM 2/3

- Global Currency with Ticket Schedulers:
- ***What is Global Currency ?***
  - Shared/Global Ticket Pool the OS distributes among all users & jobs
- ***How does the OS distribute global currency (tickets)?***
  - **LOTTERY**: 2 users (A & B), OS will distribute:
    50% of tickets to user A's jobs *(where A has 1, 2,….n Jobs)*
    50% of tickets to user B's jobs *(where B has 1, 2,….n Jobs)*
  - Global concurrency allows the OS to balance the total share of the resources provided to user A and user B fairly
  - **STRIDE**: for stride scheduler, # of global tickets is used to calculate stride value

8



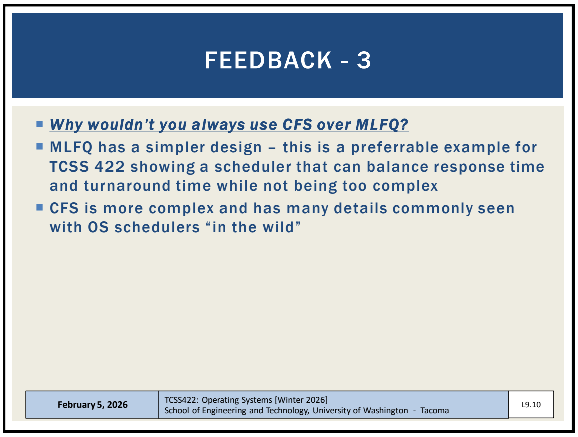
FEEDBACK - 2

- ***Why do we need to calculate the stride value proportionally to the number of global (system) tickets ? Can we use the user assigned # of tickets directly as the stride value?***
  - There can be any number of jobs in the system
  - The job's tickets represent priority: **higher # of tickets = higher priority**
  - Stride values ***reverse*** this relationship: **higher stride = lower priority**
- Why?
  - Stride value is the amount to increment the job's pass value for each round. A smaller stride will cause the job's pass value to increment more slowly. The job with the lowest pass value is always chosen to run.

EX: Consider if there are two jobs, A with stride=1, b with stride=100

  - Both jobs start with pass values of zero
  - If B goes first, it's pass value is incremented to 100
  - A then runs 100 times, before A and B have the same pass value of 100

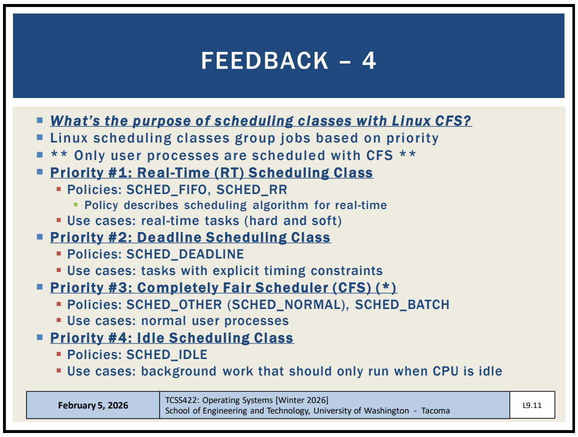
9



FEEDBACK - 3

- ***Why wouldn't you always use CFS over MLFQ?***
- MLFQ has a simpler design – this is a preferrable example for TCSS 422 showing a scheduler that can balance response time and turnaround time while not being too complex
- CFS is more complex and has many details commonly seen with OS schedulers "in the wild"

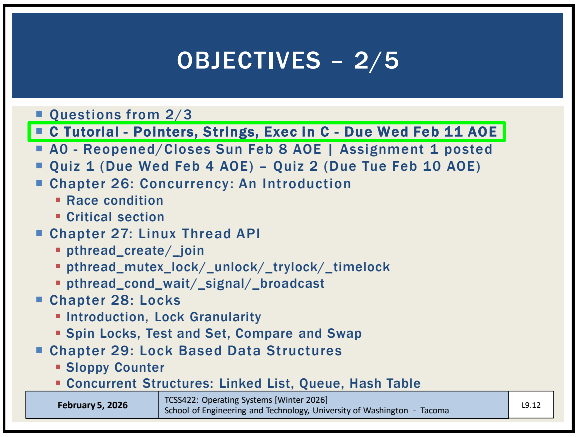
10



FEEDBACK – 4

- ***What's the purpose of scheduling classes with Linux CFS?***
- Linux scheduling classes group jobs based on priority
- ** Only user processes are scheduled with CFS **
- **Priority #1: Real-Time (RT) Scheduling Class**
  - Policies: SCHED_FIFO, SCHED_RR
    - Policy describes scheduling algorithm for real-time
  - Use cases: real-time tasks (hard and soft)
- **Priority #2: Deadline Scheduling Class**
  - Policies: SCHED_DEADLINE
  - Use cases: tasks with explicit timing constraints
- **Priority #3: Completely Fair Scheduler (CFS) (*)**
  - Policies: SCHED_OTHER (SCHED_NORMAL), SCHED_BATCH
  - Use cases: normal user processes
- **Priority #4: Idle Scheduling Class**
  - Policies: SCHED_IDLE
  - Use cases: background work that should only run when CPU is idle

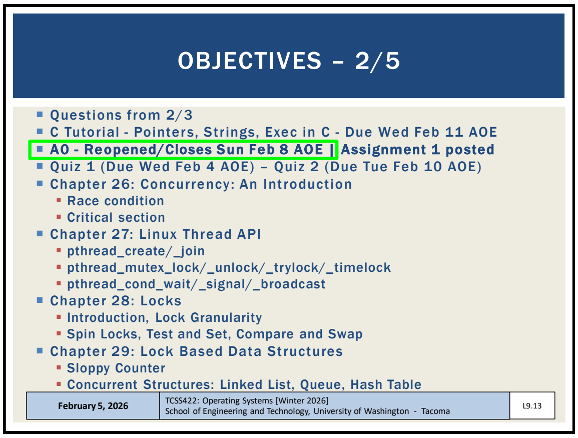
11



OBJECTIVES – 2/5

- Questions from 2/3
- **C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE**
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

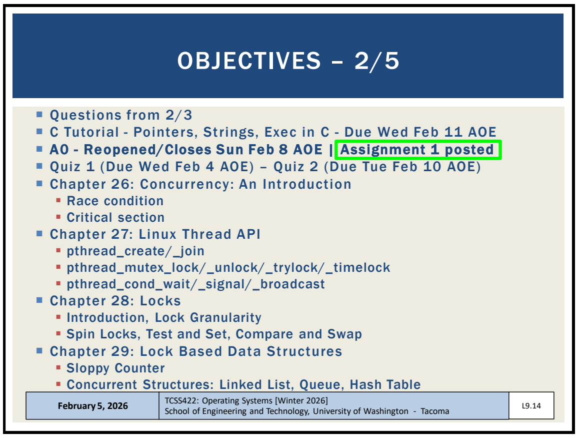


OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L9.13

13

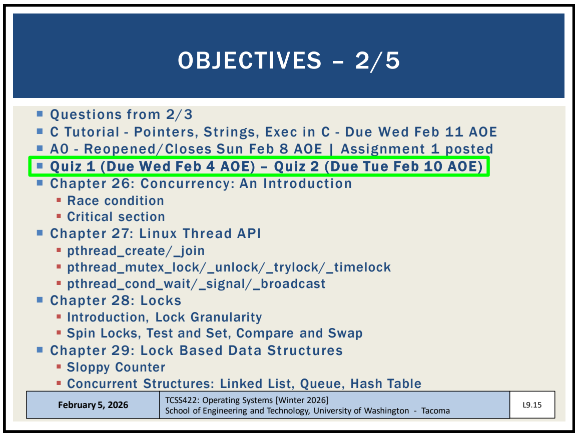


OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L9.14

14

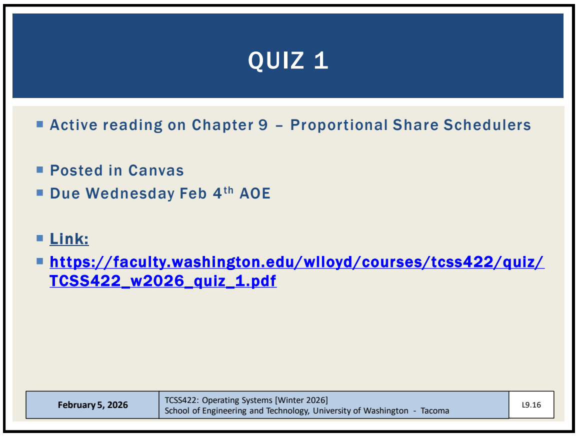


OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L9.15

15



QUIZ 1

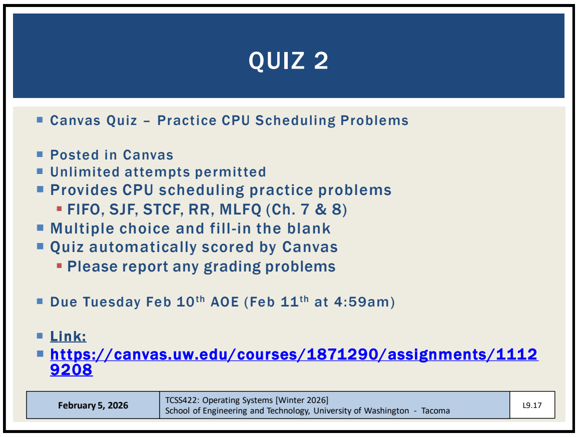
- Active reading on Chapter 9 – Proportional Share Schedulers

- Posted in Canvas
- Due Wednesday Feb 4th AOE

- **Link:**
- **https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_w2026_quiz_1.pdf**

February 5, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L9.16

16



QUIZ 2

- Canvas Quiz – Practice CPU Scheduling Problems

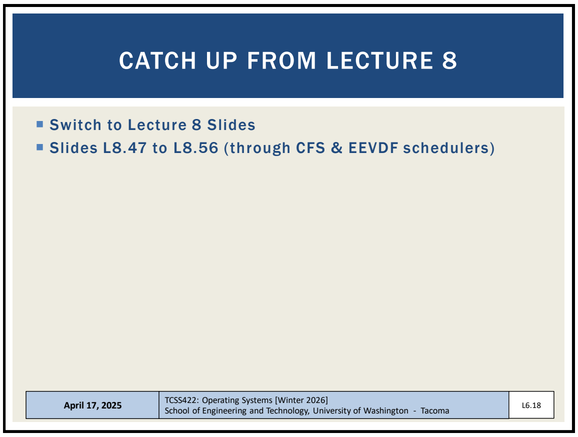
- Posted in Canvas
- Unlimited attempts permitted
- Provides CPU scheduling practice problems
  - FIFO, SJF, STCF, RR, MLFQ (Ch. 7 & 8)
- Multiple choice and fill-in the blank
- Quiz automatically scored by Canvas
  - Please report any grading problems

- Due Tuesday Feb 10th AOE (Feb 11th at 4:59am)

- **Link:**
- **https://canvas.uw.edu/courses/1871290/assignments/11129208**

February 5, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L9.17

17

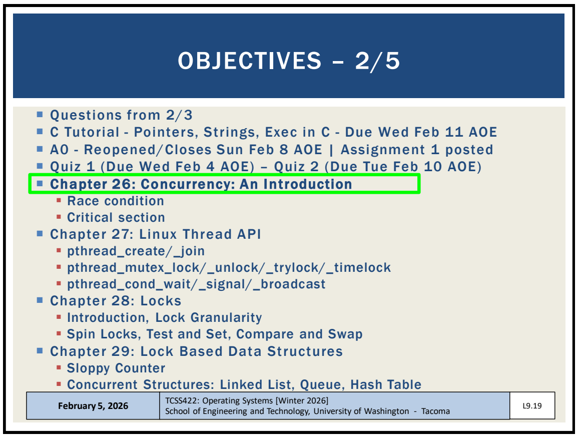


CATCH UP FROM LECTURE 8

- Switch to Lecture 8 Slides
- Slides L8.47 to L8.56 (through CFS & EEVDF schedulers)

April 17, 2025 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L6.18

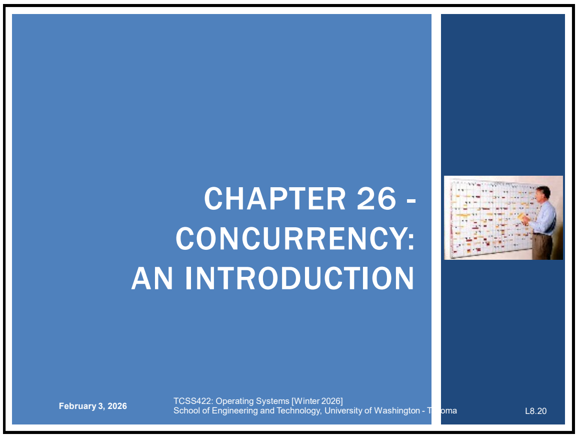
18



OBJECTIVES – 2/5

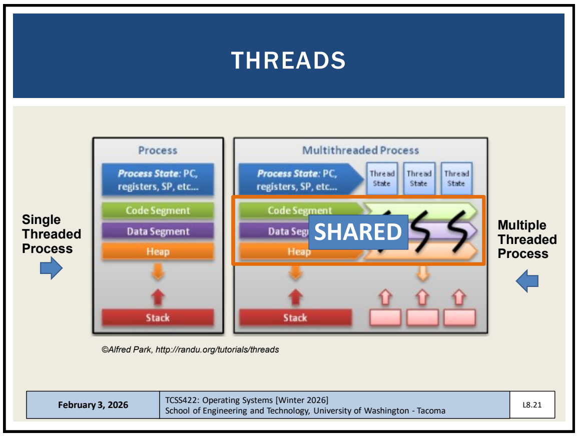
- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 AOE)
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: 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

19



CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

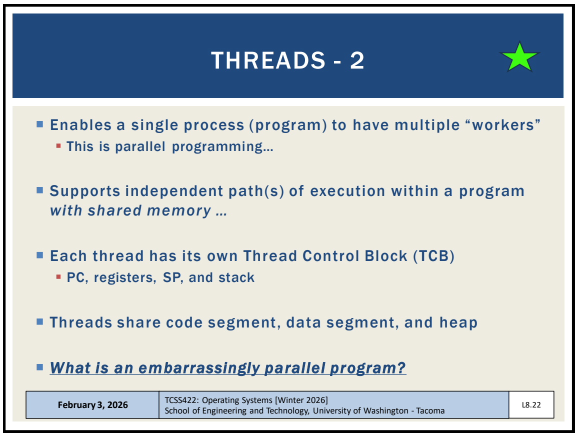
20



THREADS

©Alfred Park, http://randu.org/tutorials/threads

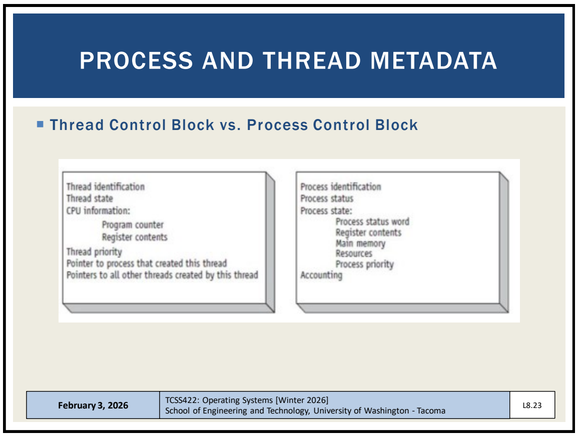
21



THREADS - 2

- Enables a single process (program) to have multiple "workers"
  - This is parallel programming...
- Supports independent path(s) of execution within a program *with shared memory ...*
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Threads share code segment, data segment, and heap
- ***What is an embarrassingly parallel program?***

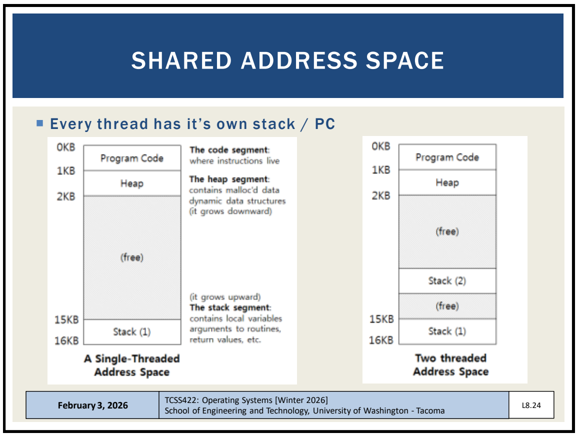
22



PROCESS AND THREAD METADATA

- Thread Control Block vs. Process Control Block

23

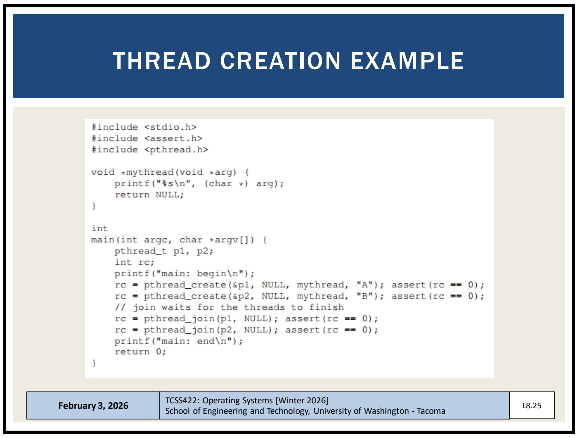


SHARED ADDRESS SPACE

- Every thread has it's own stack / PC

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (1)
The code segment: where instructions live
The heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.
A Single-Threaded Address Space
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)
Two threaded Address Space

24

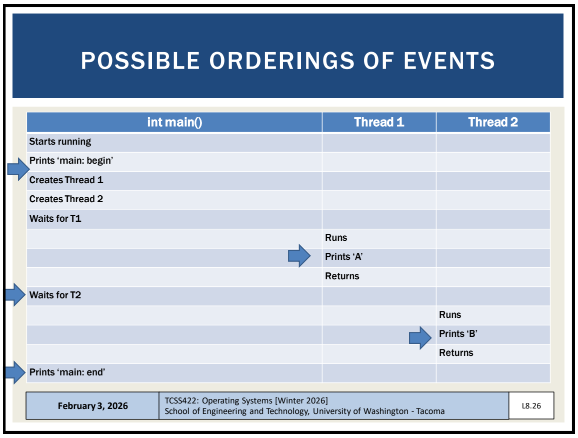


THREAD CREATION EXAMPLE

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

February 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L8.25

25

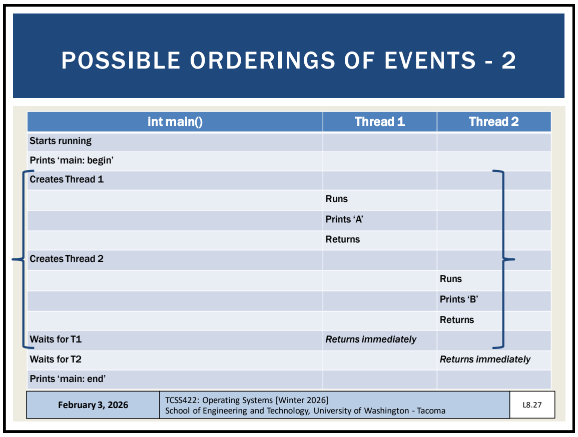


POSSIBLE ORDERINGS OF EVENTS

int main()	Thread 1	Thread 2
Starts running		
Prints ‘main: begin’		
Creates Thread 1		
Creates Thread 2		
Waits for T1		
	Runs	
	Prints ‘A’	
	Returns	
Waits for T2		
		Runs
		Prints ‘B’
		Returns
Prints ‘main: end’		

February 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L8.26

26

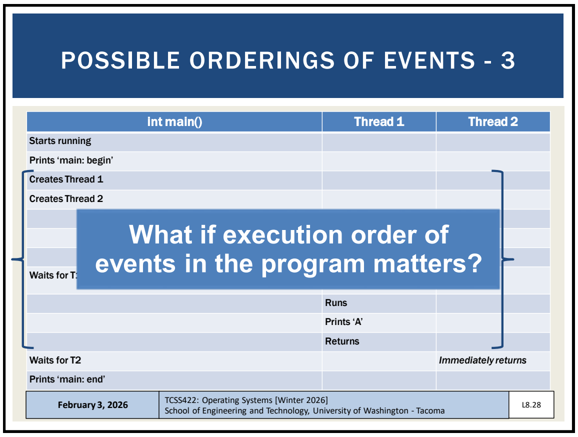


POSSIBLE ORDERINGS OF EVENTS - 2

int main()	Thread 1	Thread 2
Starts running		
Prints ‘main: begin’		
Creates Thread 1		
	Runs	
	Prints ‘A’	
	Returns	
Creates Thread 2		
		Runs
		Prints ‘B’
		Returns
Waits for T1	*Returns immediately*	
Waits for T2		*Returns immediately*
Prints ‘main: end’		

February 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L8.27

27



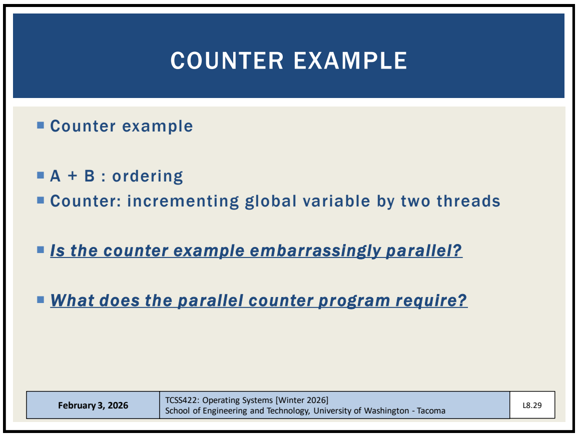
POSSIBLE ORDERINGS OF EVENTS - 3

int main()	Thread 1	Thread 2
Starts running		
Prints ‘main: begin’		
Creates Thread 1		
Creates Thread 2		
Waits for T		
	Runs	
	Prints ‘A’	
	Returns	
Waits for T2		*Immediately returns*
Prints ‘main: end’		

What if execution order of events in the program matters?

February 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L8.28

28

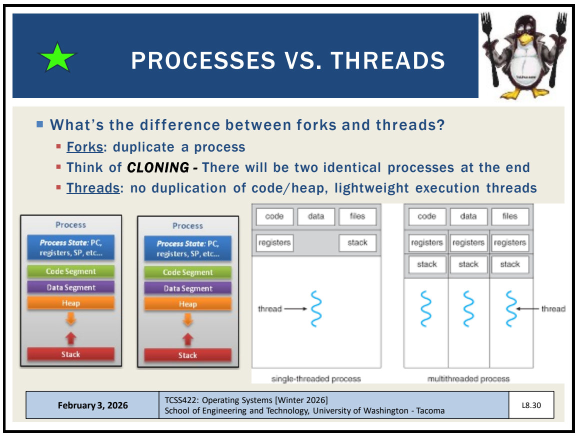


COUNTER EXAMPLE

- Counter example
- A + B : ordering
- Counter: incrementing global variable by two threads
- ***Is the counter example embarrassingly parallel?***
- ***What does the parallel counter program require?***

February 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L8.29

29

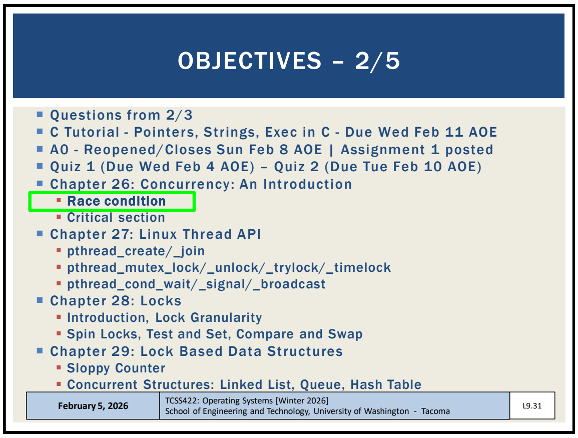


PROCESSES VS. THREADS

- What’s the difference between forks and threads?
  - Forks: duplicate a process
  - Think of ***CLONING -*** There will be two identical processes at the end
  - Threads: no duplication of code/heap, lightweight execution threads

February 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L8.30

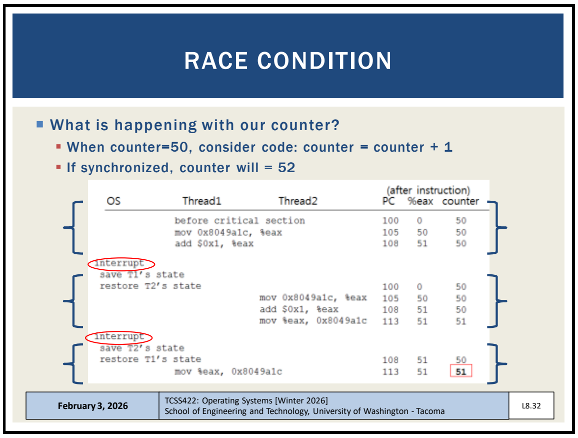
30



OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

31

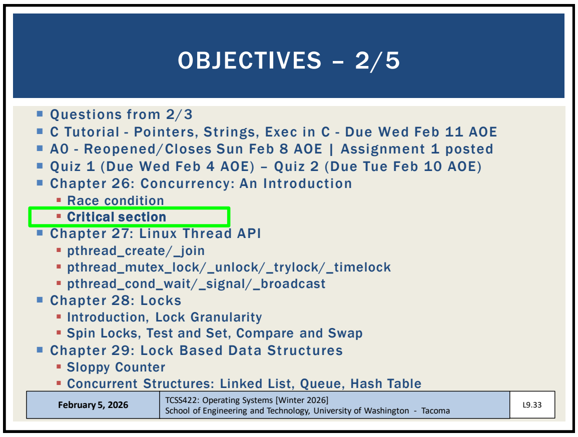


RACE CONDITION

- What is happening with our counter?
  - When counter=50, consider code: counter = counter + 1
  - If synchronized, counter will = 52

```
                                                          (after instruction)
OS                 Thread1                 Thread2           PC   %eax  counter
              before critical section                        100   0     50
              mov 0x8049a1c, %eax                            105   50    50
              add $0x1, %eax                                 108   51    50
interrupt
 save T1's state
 restore T2's state                                          100   0     50
                             mov 0x8049a1c, %eax             105   50    50
                             add $0x1, %eax                  108   51    50
                             mov %eax, 0x8049a1c             113   51    51
interrupt
 save T2's state
 restore T1's state                                          108   51    50
              mov %eax, 0x8049a1c                            113   51    51
```

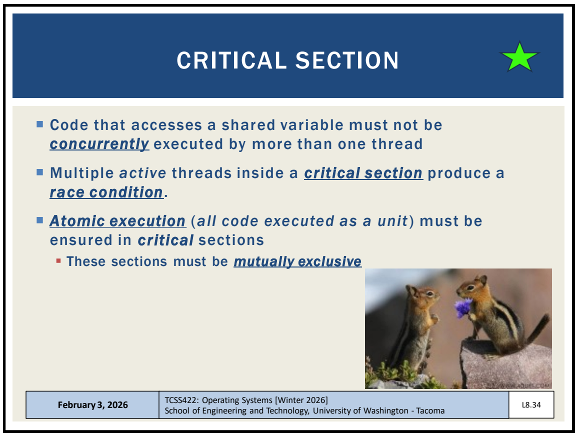
32



OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

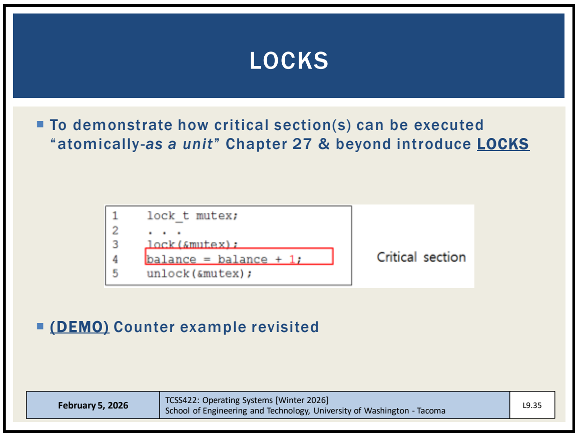
33



CRITICAL SECTION ★

- Code that accesses a shared variable must not be ***concurrently*** executed by more than one thread
- Multiple *active* threads inside a ***critical section*** produce a ***race condition***.
- ***Atomic execution*** (*all code executed as a unit*) must be ensured in ***critical*** sections
  - These sections must be ***mutually exclusive***

34



LOCKS

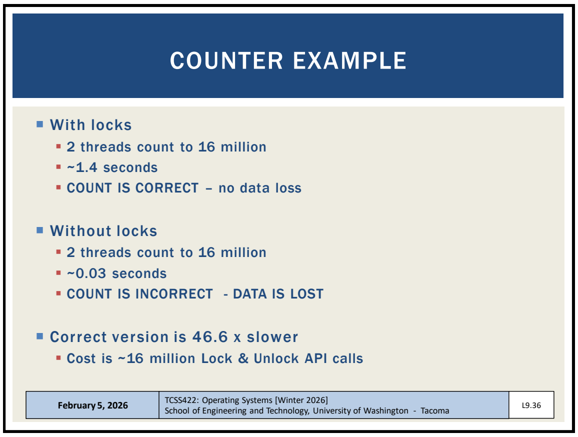
- To demonstrate how critical section(s) can be executed "atomically-*as a unit*" Chapter 27 & beyond introduce **LOCKS**

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

- **(DEMO)** Counter example revisited

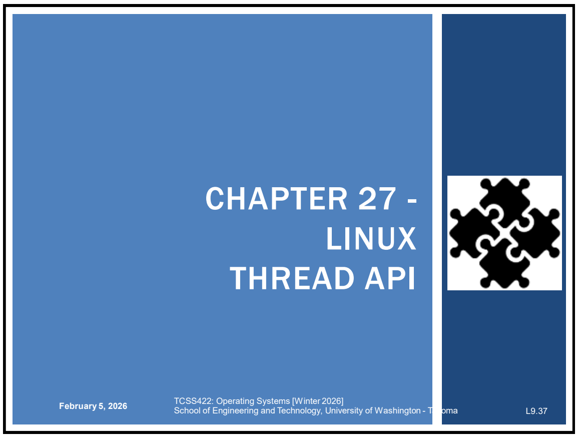
35



COUNTER EXAMPLE

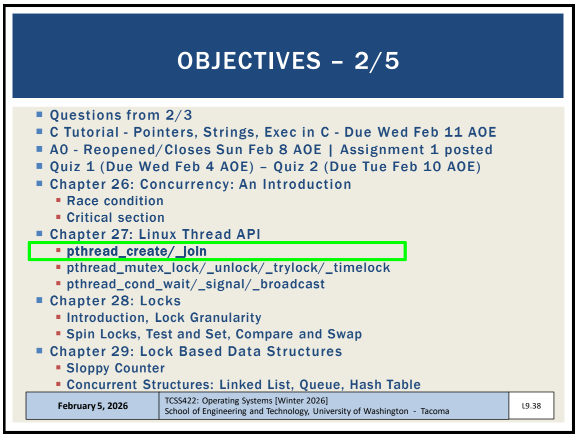
- With locks
  - 2 threads count to 16 million
  - ~1.4 seconds
  - COUNT IS CORRECT – no data loss
- Without locks
  - 2 threads count to 16 million
  - ~0.03 seconds
  - COUNT IS INCORRECT - DATA IS LOST
- Correct version is 46.6 x slower
  - Cost is ~16 million Lock & Unlock API calls

36



CHAPTER 27 - LINUX THREAD API

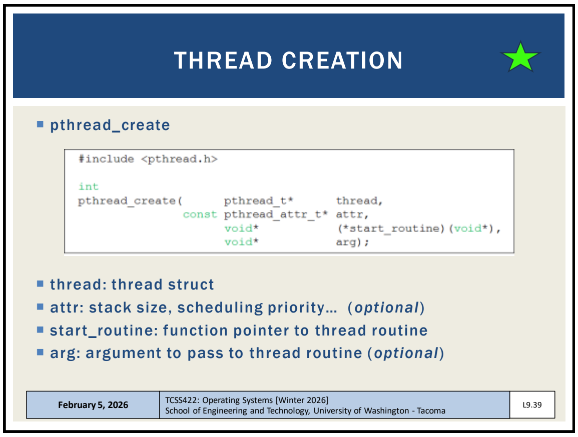
37



OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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



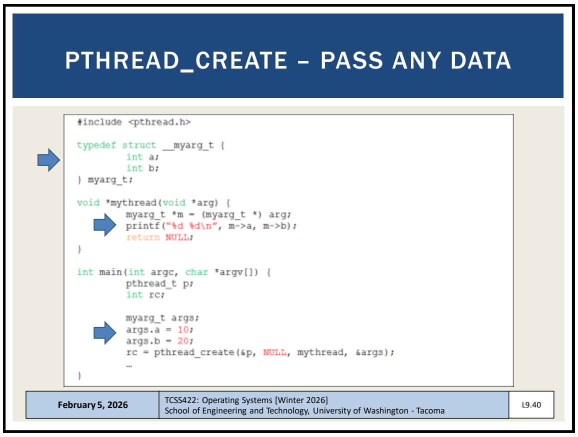
THREAD CREATION ★

- 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*)

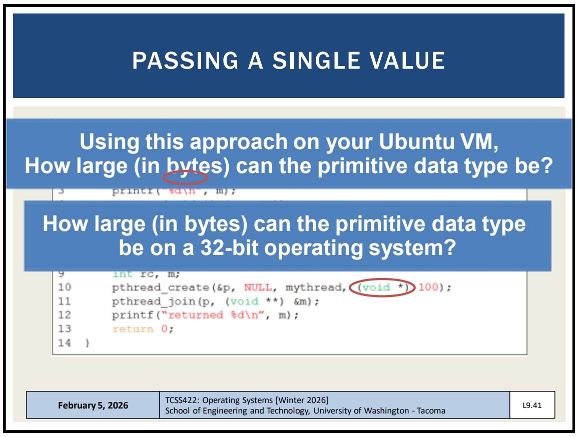
39



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);
        …
}
```

40



PASSING A SINGLE VALUE

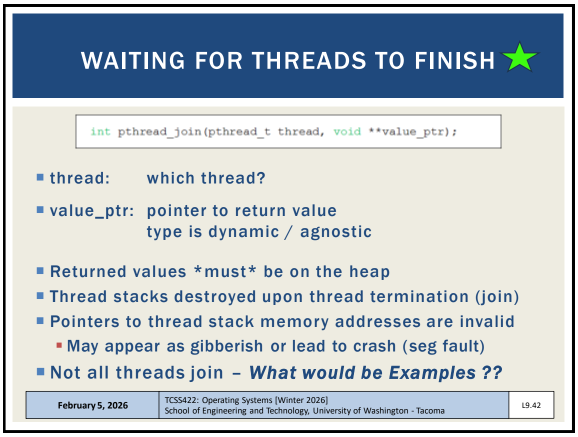
**Using this approach on your Ubuntu VM,
How large (in bytes) can the primitive data type be?**

```
        printf("%d\n", m);
```

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

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

41

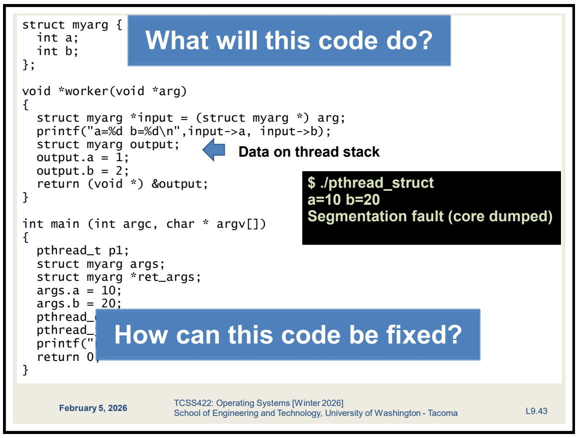


WAITING FOR THREADS TO FINISH ★

```
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 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 ??***

42



What will this code do?

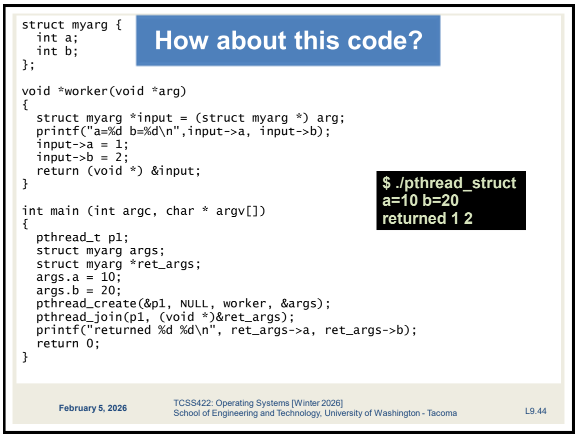
```
struct myarg {
  int a;
  int b;
};

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?

43

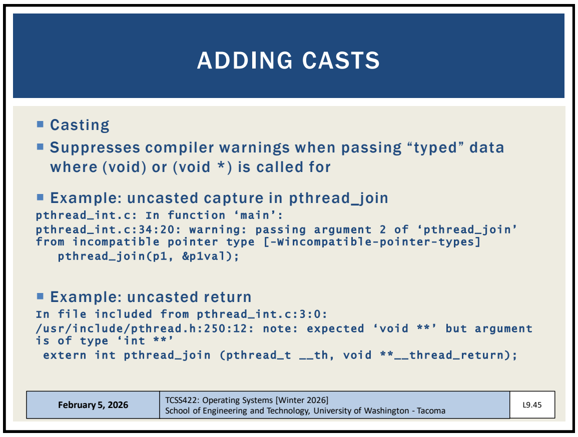


How about this code?

```
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
```

44



ADDING CASTS

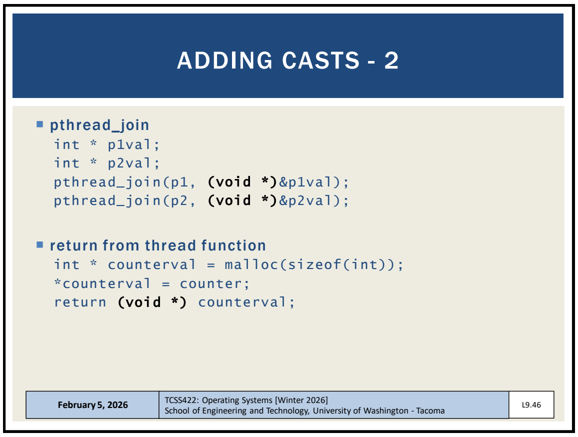
- Casting
- Suppresses compiler warnings when passing “typed” data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join

```
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);
```

45



ADDING CASTS - 2

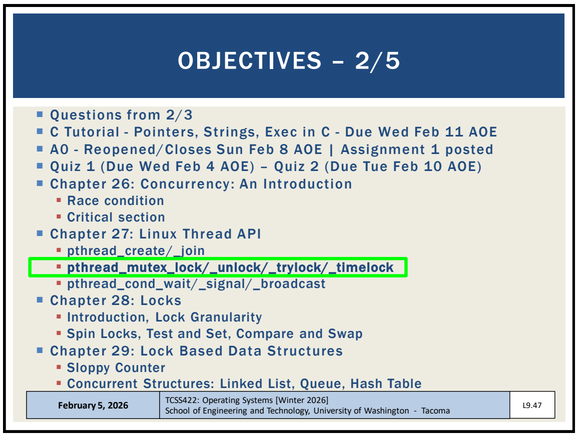
- pthread_join

```
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;
```

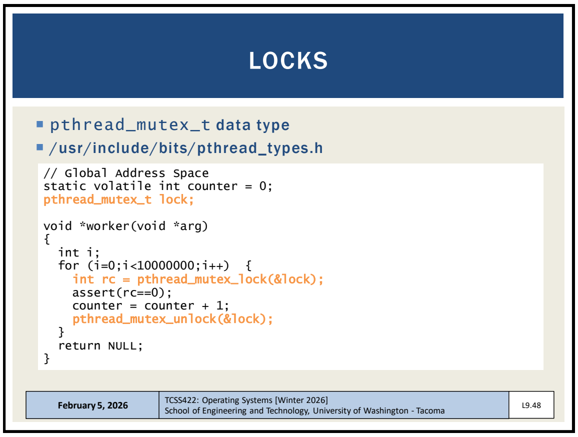
46



OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

47

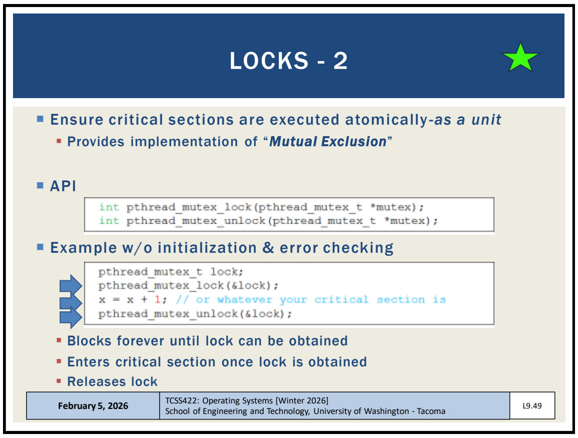


LOCKS

- 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;
}
```

48



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);
```

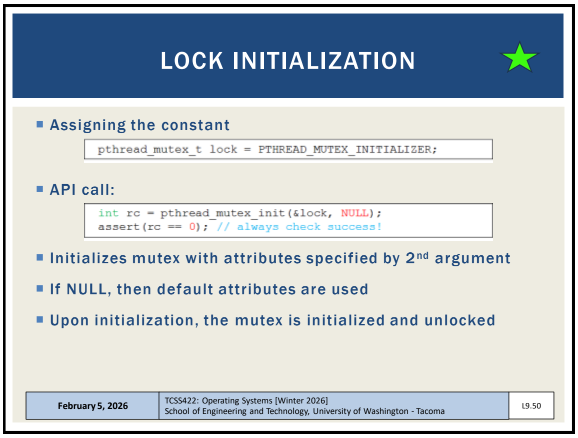
- Example w/o initialization & error checking

```
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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma | L9.49

49



LOCK INITIALIZATION

- Assigning the constant

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

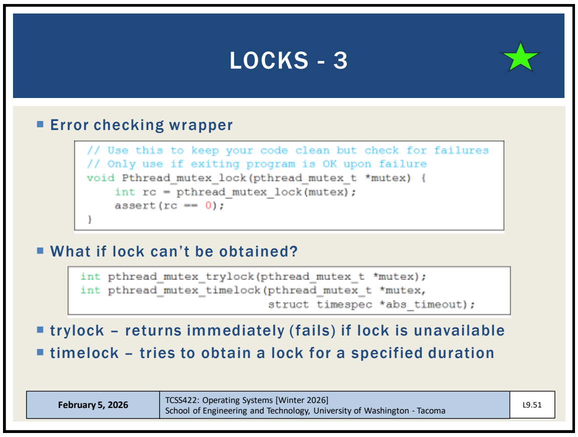
- API call:

```
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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma | L9.50

50



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);
}
```

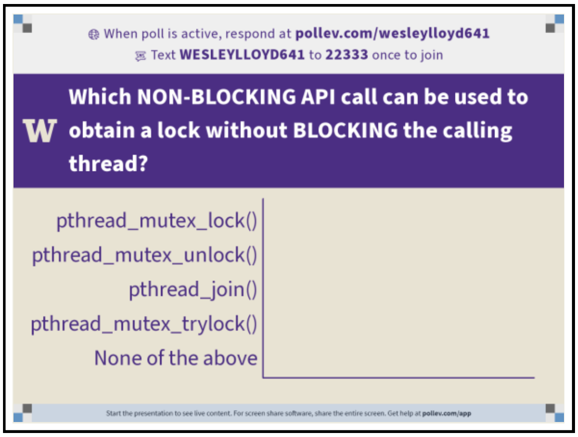
- What if lock can't be obtained?

```
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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma | L9.51

51

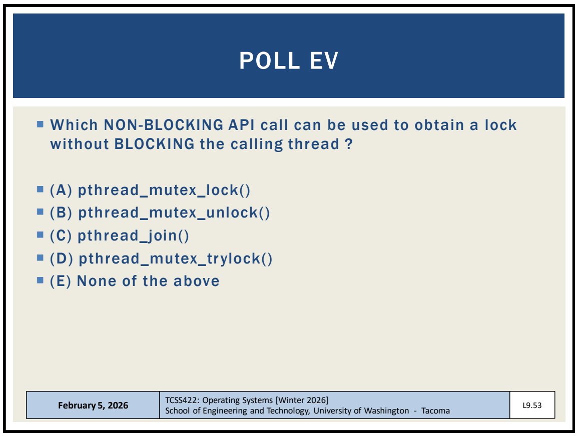


When poll is active, respond at **pollev.com/wesleylloyd641**
Text **WESLEYLLOYD641** to **22333** once to join

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

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

52

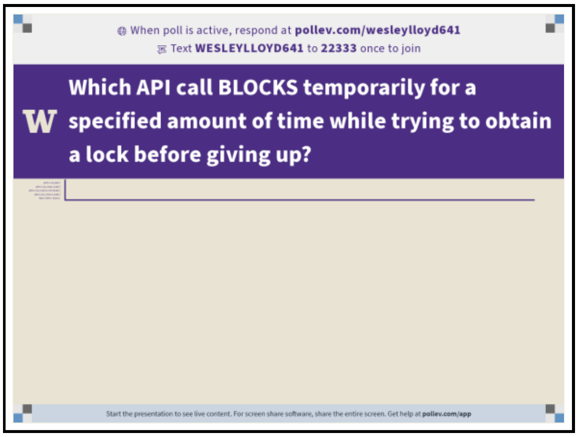


POLL EV

- 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

February 5, 2026 | TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma | L9.53

53

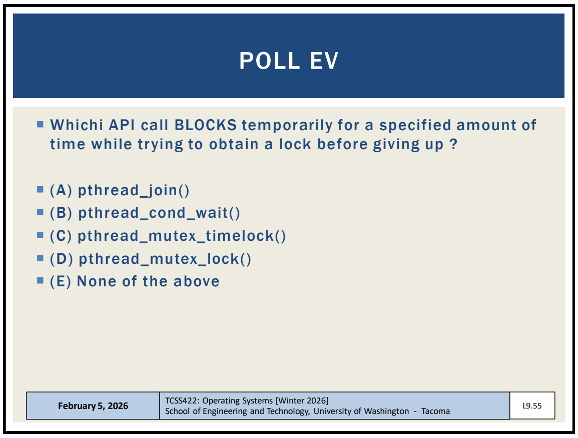


When poll is active, respond at **pollev.com/wesleylloyd641**
Text **WESLEYLLOYD641** to **22333** once to join

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**

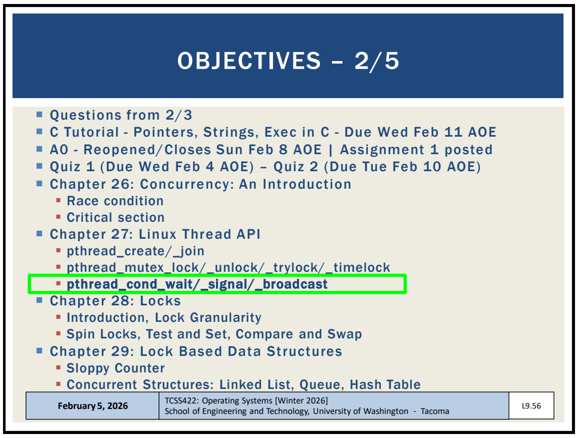
54



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

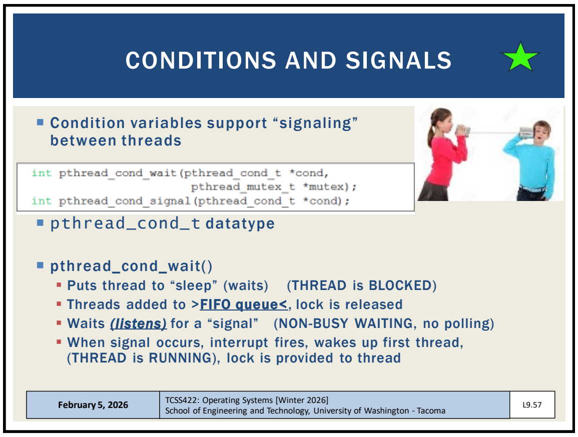
55



OBJECTIVES – 2/5

- Questions from 2/3
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Feb 11 AOE
- A0 - Reopened/Closes Sun Feb 8 AOE | Assignment 1 posted
- Quiz 1 (Due Wed Feb 4 AOE) – Quiz 2 (Due Tue Feb 10 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

56



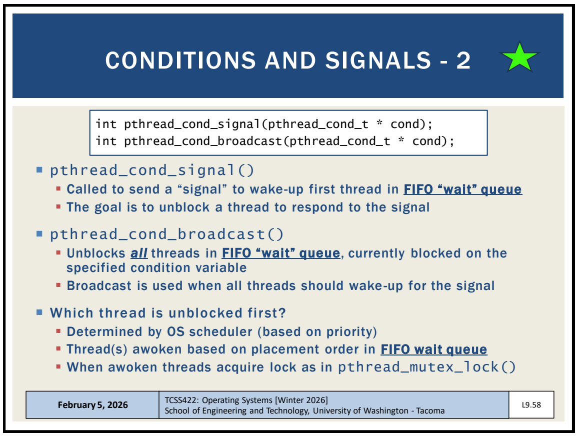
CONDITIONS AND SIGNALS

- 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

57

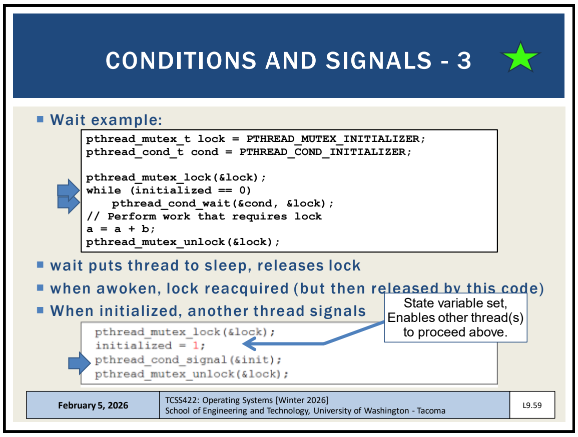


CONDITIONS AND SIGNALS - 2

```
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()

58



CONDITIONS AND SIGNALS - 3

- Wait example:

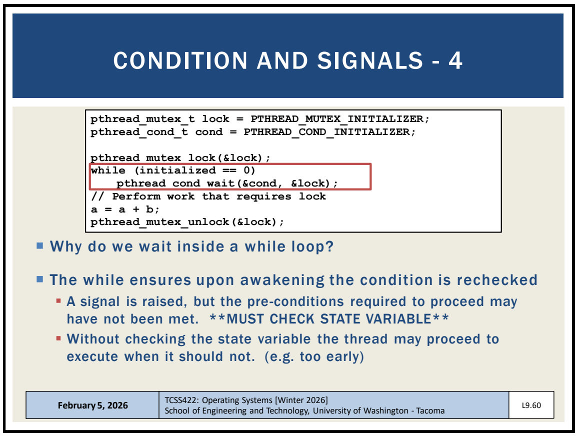
```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

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.

59



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)

60

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

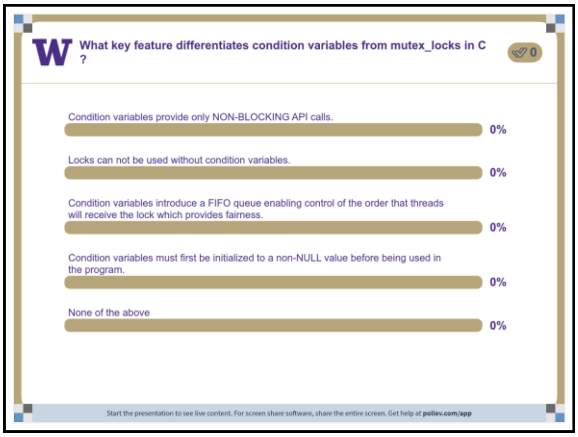
61

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

62



What key feature differentiates condition variables from mutex_locks in C ?

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

63

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

64

QUESTIONS

127