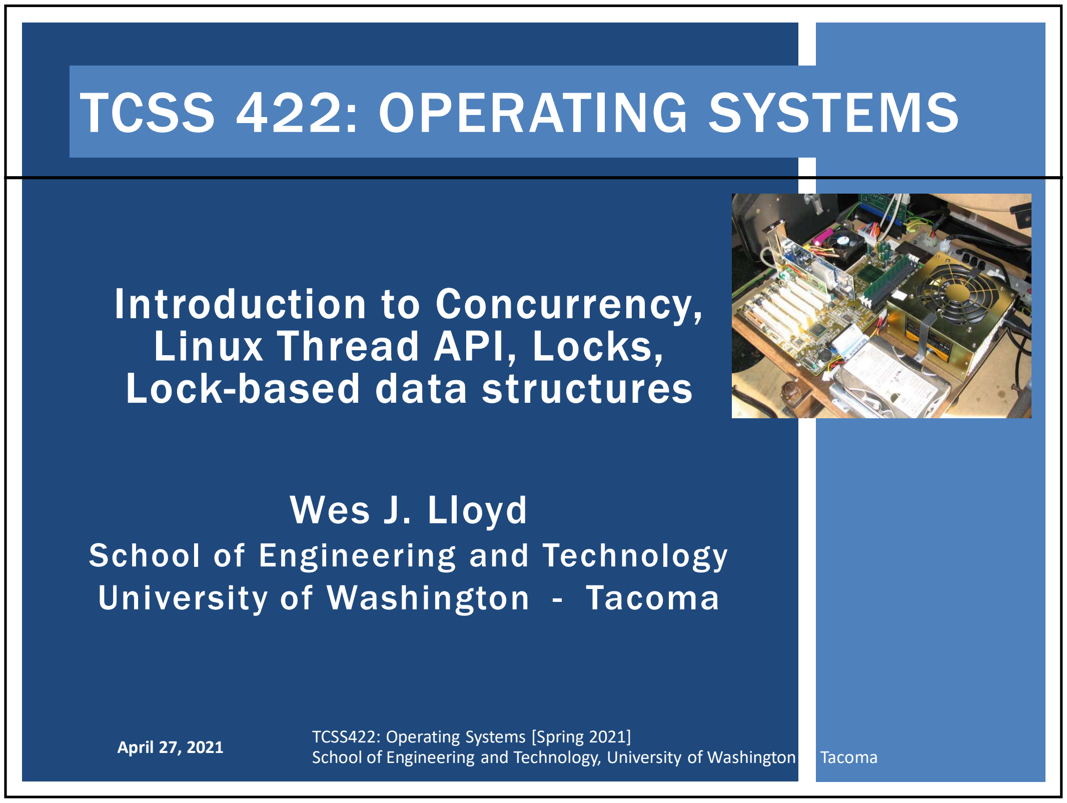


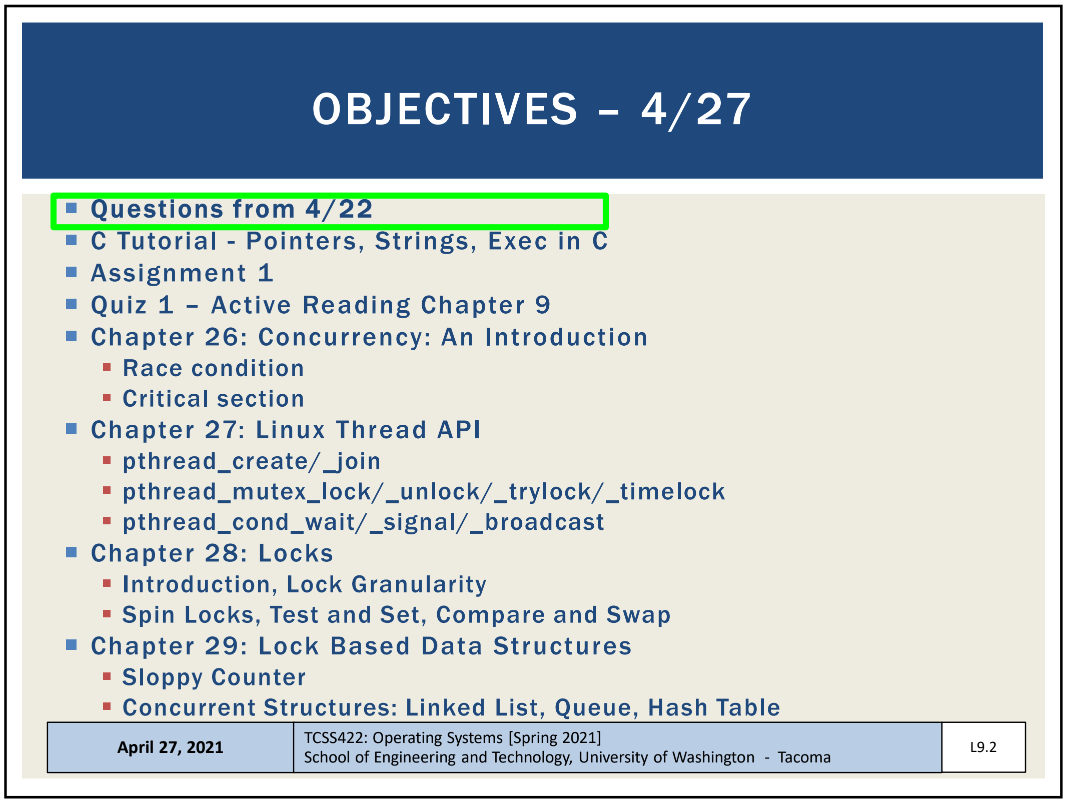
TCSS 422: OPERATING SYSTEMS

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

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

April 27, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma



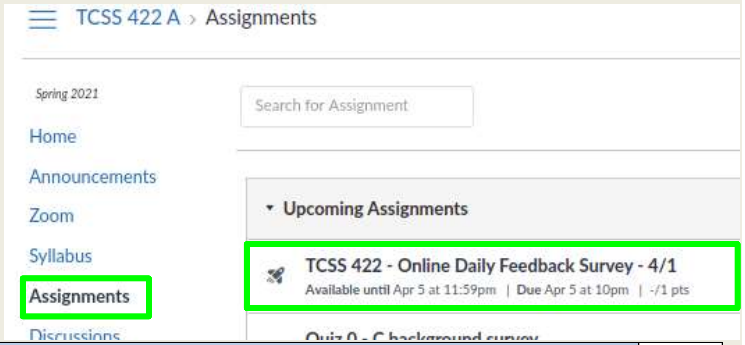
OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

April 27, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.2

ONLINE DAILY FEEDBACK SURVEY

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


TCSS 422 A › Assignments

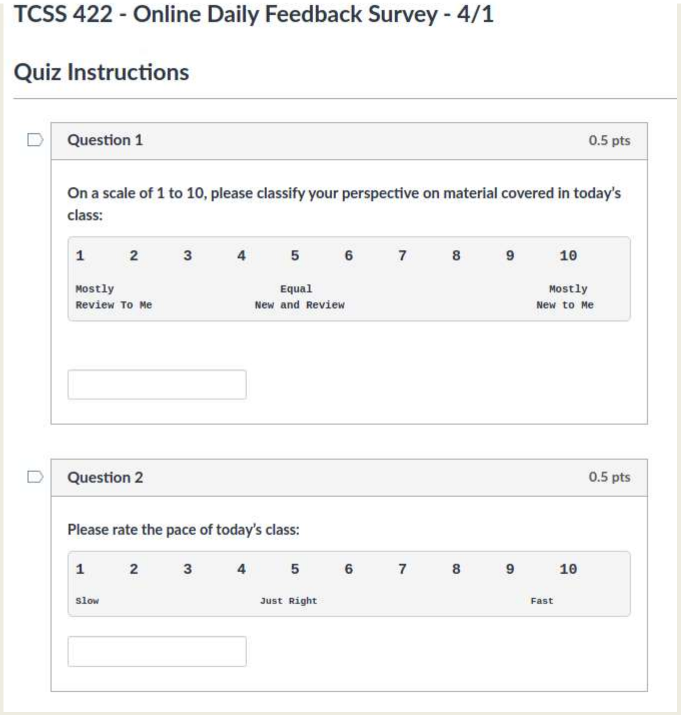
Spring 2021
Home
Announcements
Zoom
Syllabus
Assignments
Discussions

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
```

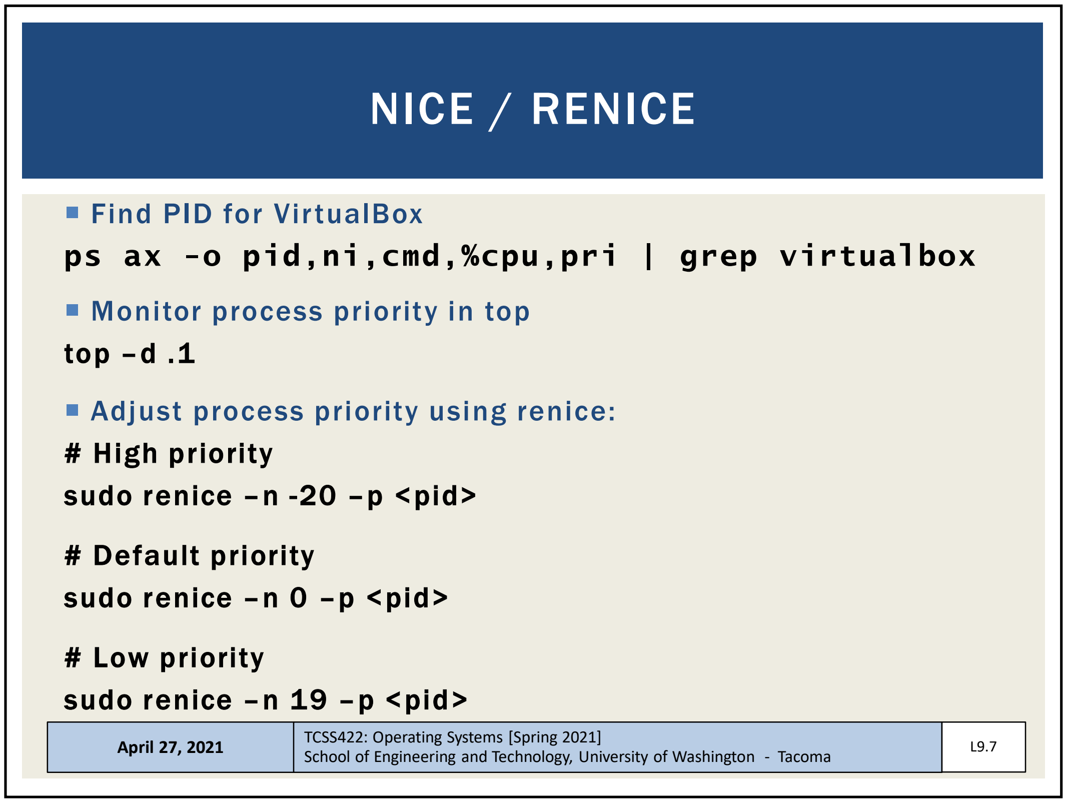
MATERIAL / PACE

- Please classify your perspective on material covered in today's class (57 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- <u>Average – 6.90 (↑ - previous 6.67)</u>

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- <u>Average – 5.52 (↓ - previous 5.53)</u>

FEEDBACK

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



NICE / RENICE

- Find PID for VirtualBox

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

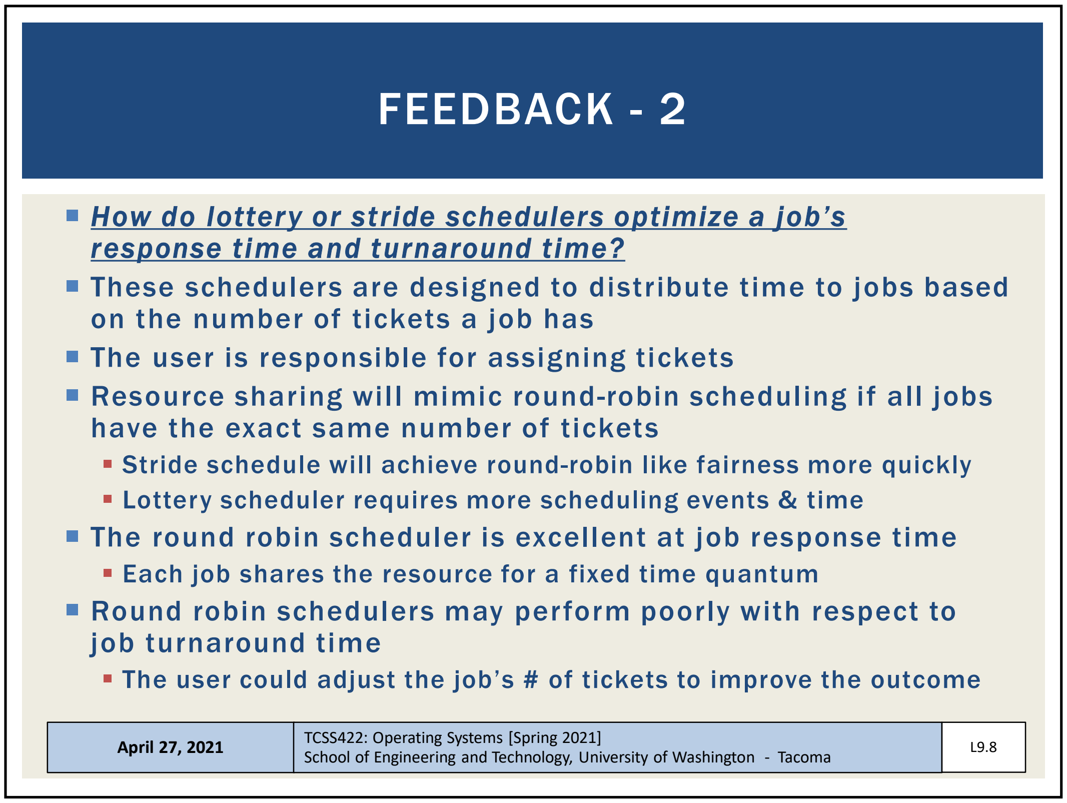
- Monitor process priority in top

```
top –d .1
```

- Adjust process priority using renice:

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

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

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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.7



FEEDBACK - 2

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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.8

FEEDBACK - 3

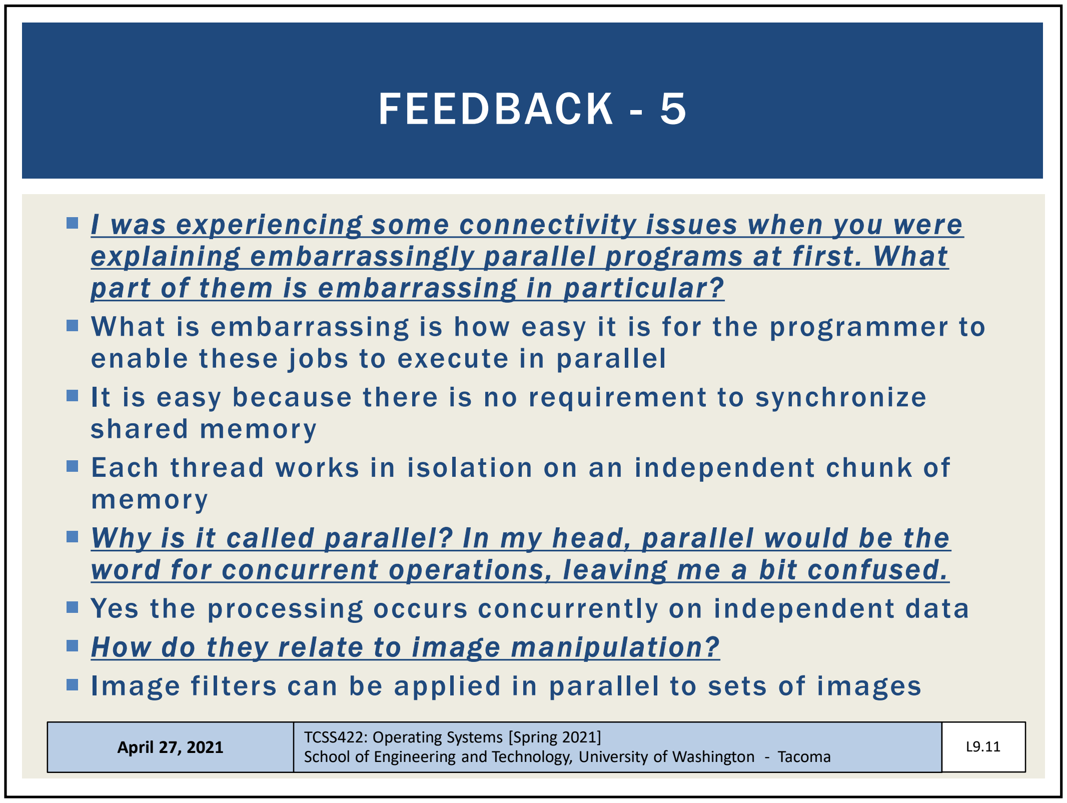
- ***I'm still a little confused on the difference between Stride and CFS scheduler and what situations where one would be better than the other.***
  - Unlike FIFO, SJF, STCF, and RR, the Stride scheduler enables a job's priority to be influenced dynamically based on the number of tickets it holds
  - The stride scheduler provides an algorithm to share any resource where there is a requirement to allow the user to influence a given job's priority
  - The Completely Fair Scheduler is the current process scheduler in Linux (Ubuntu 20.04 LTS)
  - It eliminates tickets in favor of a relative nice value
  - Users can "nudge" the priority of a job, but not set priority directly

FEEDBACK - 4

- ***How does the system decide which scheduling class a thread/process has?***
- By default jobs are of the "(TS) Time Sharing" class
- **Normal classes**: SCHED_OTHER (TS), SCHED_IDLE, SCHED_BATCH
  - TS = Time Sharing
- **Real-time classes**: SCHED_FIFO (FF), SCHED_RR (RR)
- How to show scheduling class and priority:

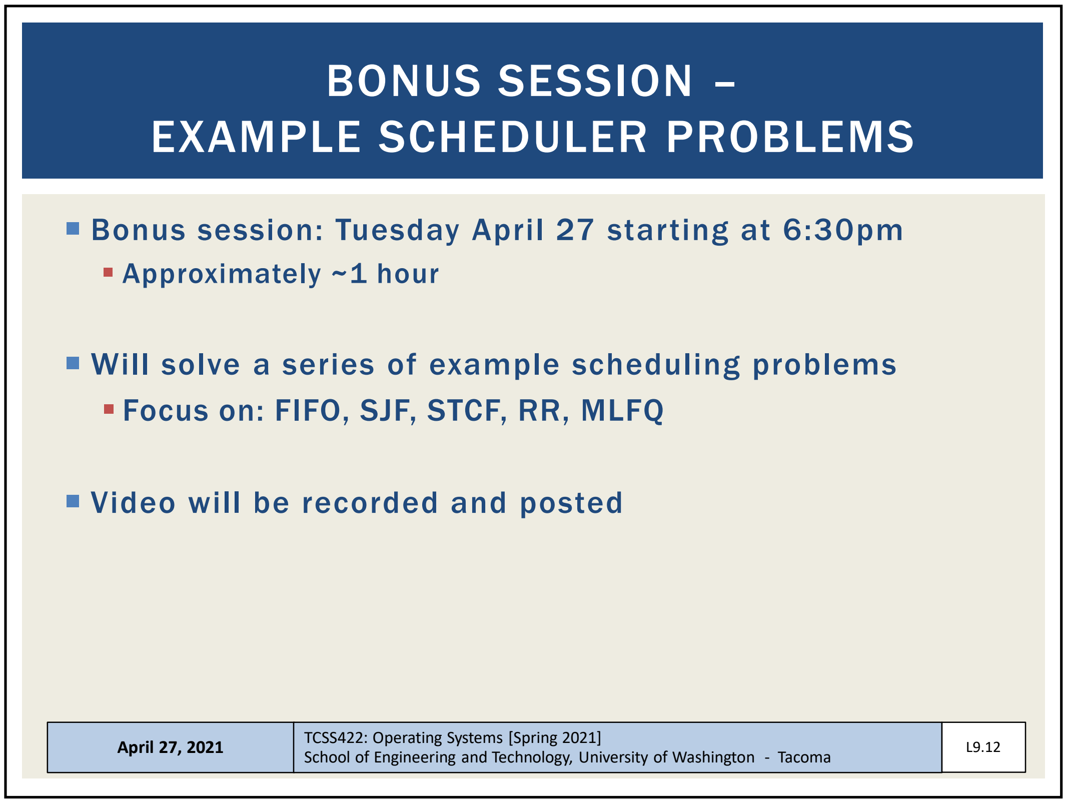
-
  ```
  #class
  ps -elfc
  ```

-
  ```
  #priority (nice value)
  ps ax -o pid,ni,cls,pri,cmd
  ```



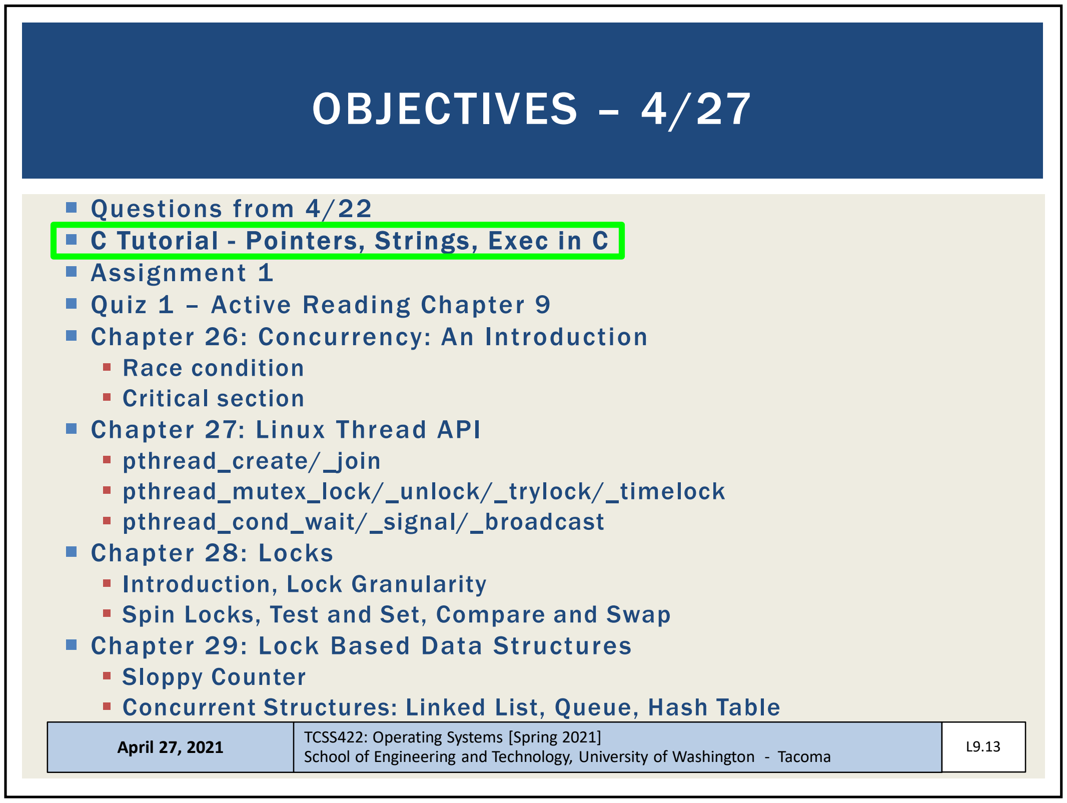
FEEDBACK - 5

- ***I was experiencing some connectivity issues when you were explaining embarrassingly parallel programs at first. What part of them is embarrassing in particular?***
- What is embarrassing is how easy it is for the programmer to enable these jobs to execute in parallel
- It is easy because there is no requirement to synchronize shared memory
- Each thread works in isolation on an independent chunk of memory
- ***Why is it called parallel? In my head, parallel would be the word for concurrent operations, leaving me a bit confused.***
- Yes the processing occurs concurrently on independent data
- ***How do they relate to image manipulation?***
- Image filters can be applied in parallel to sets of images



BONUS SESSION – EXAMPLE SCHEDULER PROBLEMS

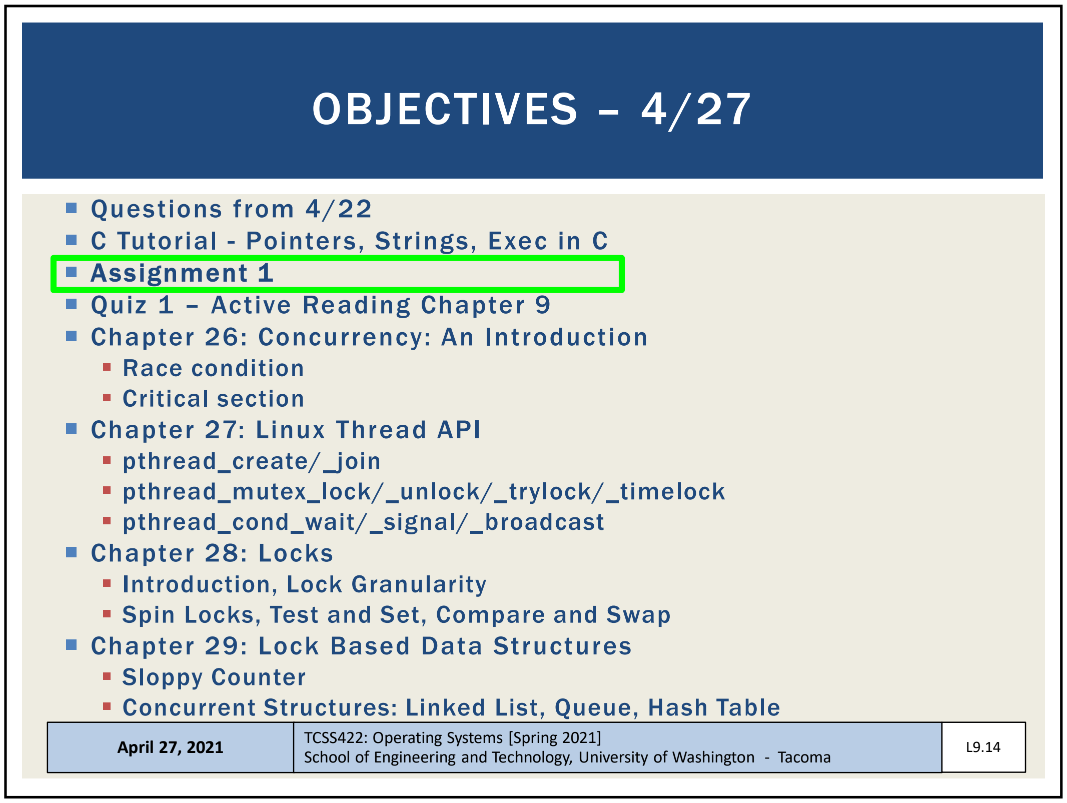
- Bonus session: Tuesday April 27 starting at 6:30pm
  - Approximately ~1 hour
- Will solve a series of example scheduling problems
  - Focus on: FIFO, SJF, STCF, RR, MLFQ
- Video will be recorded and posted



OBJECTIVES – 4/27

- Questions from 4/22
- **C Tutorial - Pointers, Strings, Exec in C**
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.13



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- **Assignment 1**
- Quiz 1 – Active Reading Chapter 9
- 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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.14

OBJECTIVES – 4/27

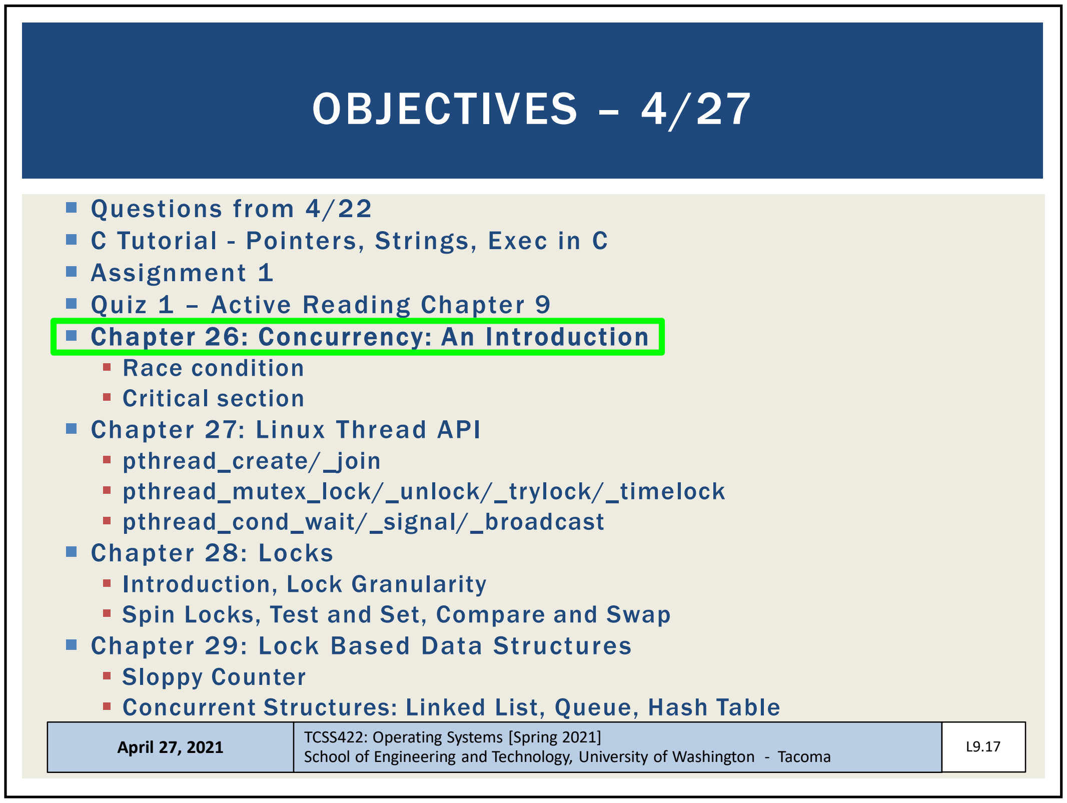
- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers

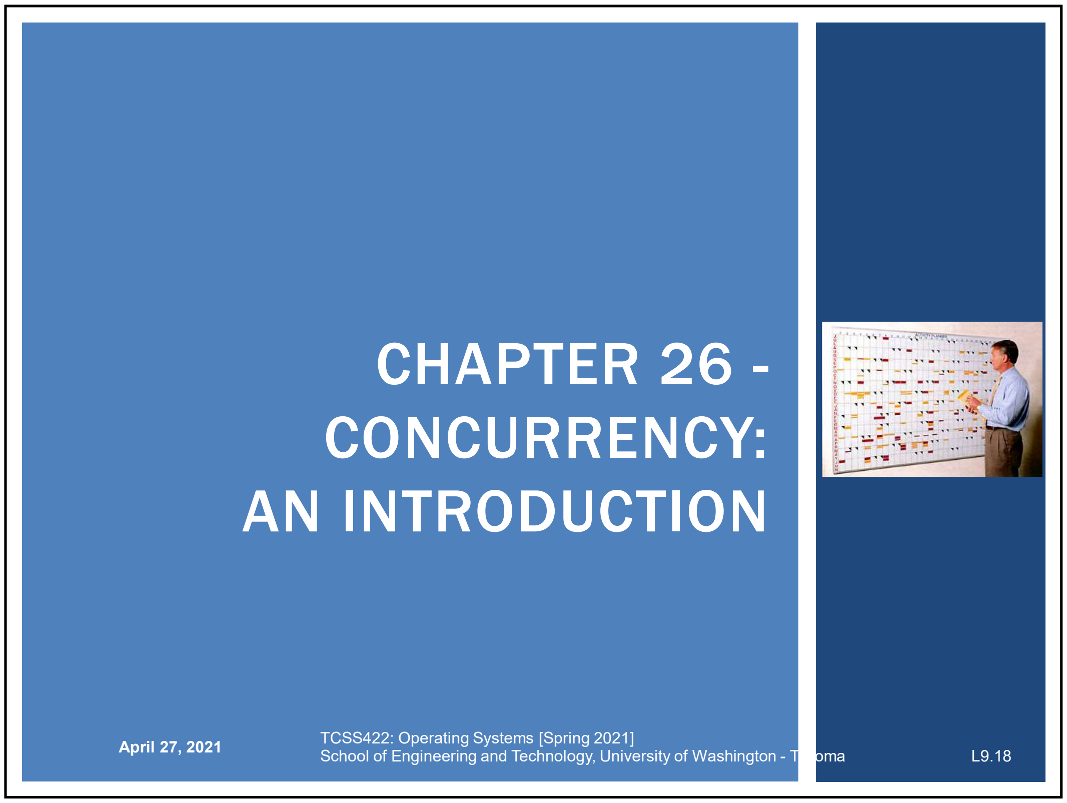
- Posted in Canvas
- Due Friday April 30th at 11:59pm
- Grace period til Sunday May 2nd at 11:59 ** AM **
- Late submissions til Tuesday May 4th at 11:59pm

- Link:
- http://faculty.washington.edu/wlloyd/courses/tcss422/TCSS422_s2021_quiz_1.pdf

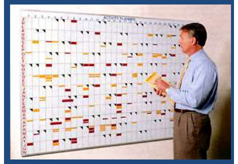


OBJECTIVES – 4/27

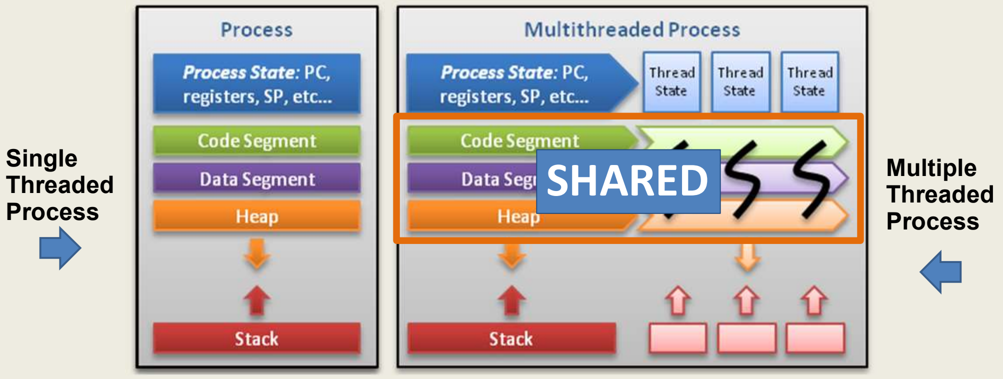
- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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



CHAPTER 26 - CONCURRENCY: AN INTRODUCTION



THREADS



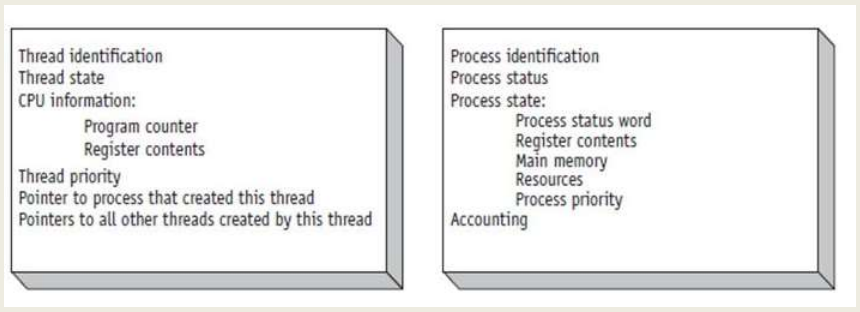
©Alfred Park, http://randu.org/tutorials/threads

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, memory, and heap are shared
- ***What is an embarrassingly parallel program?***

PROCESS AND THREAD METADATA

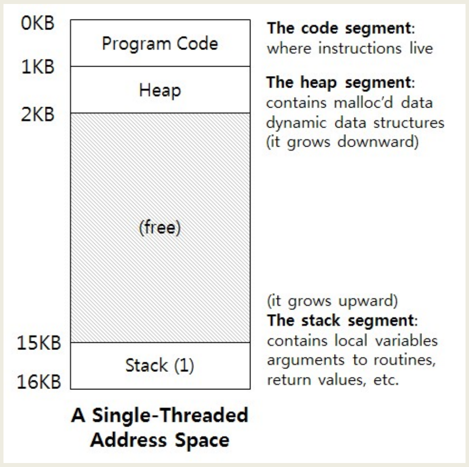
- Thread Control Block vs. Process Control Block

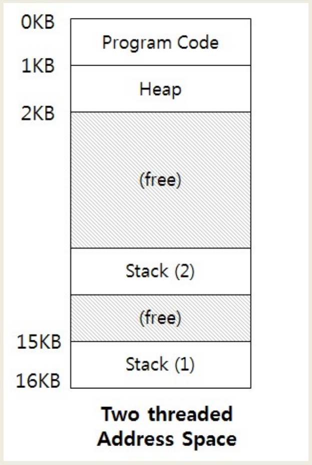


April 27, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.21

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



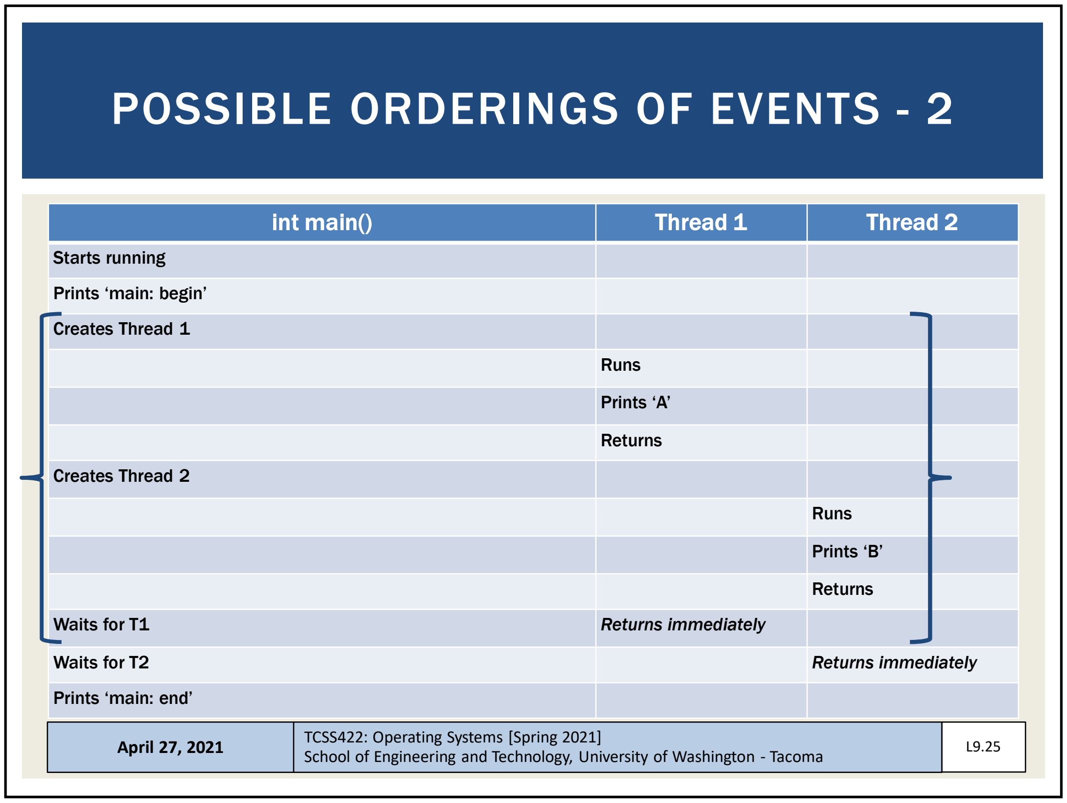
April 27, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.22

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;
}
```

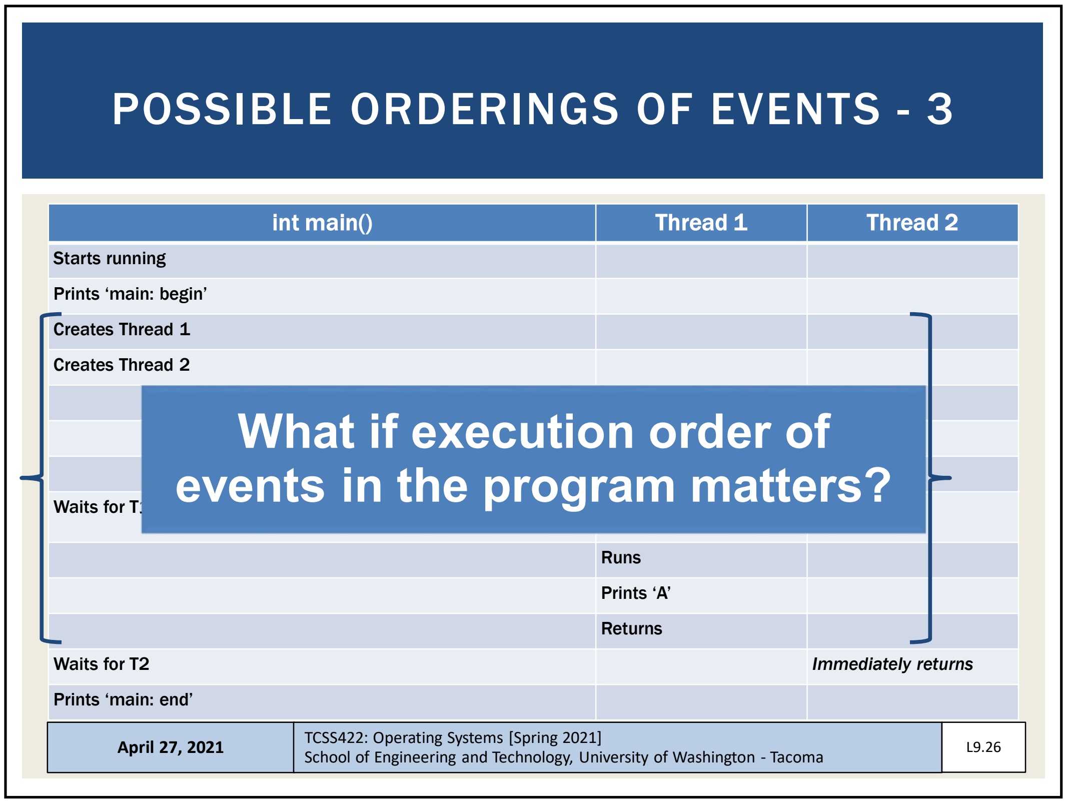
POSSIBLE ORDERINGS OF EVENTS

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’		



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'		



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?

COUNTER EXAMPLE

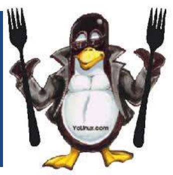
- Counter example

- A + B : ordering
- Counter: incrementing global variable by two threads

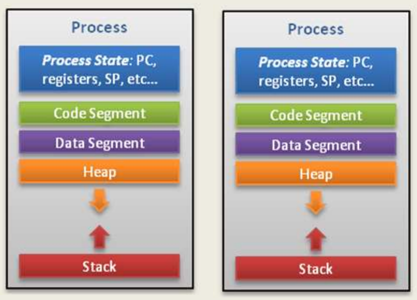
- ***Is the counter example embarrassingly parallel?***

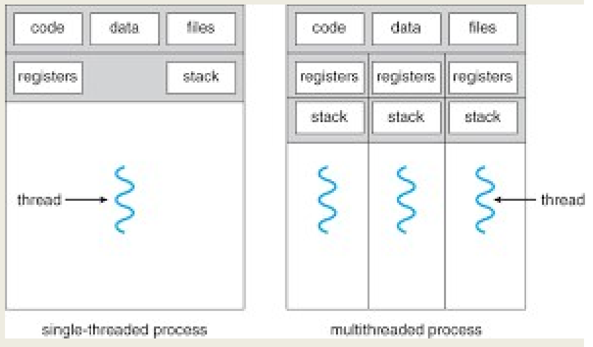
- ***What does the parallel counter program require?***

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


Process
Process State: PC, registers, SP, etc...
Code Segment
Data Segment
Heap
Stack
Process
Process State: PC, registers, SP, etc...
Code Segment
Data Segment
Heap
Stack



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

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**

OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

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***

LOCKS

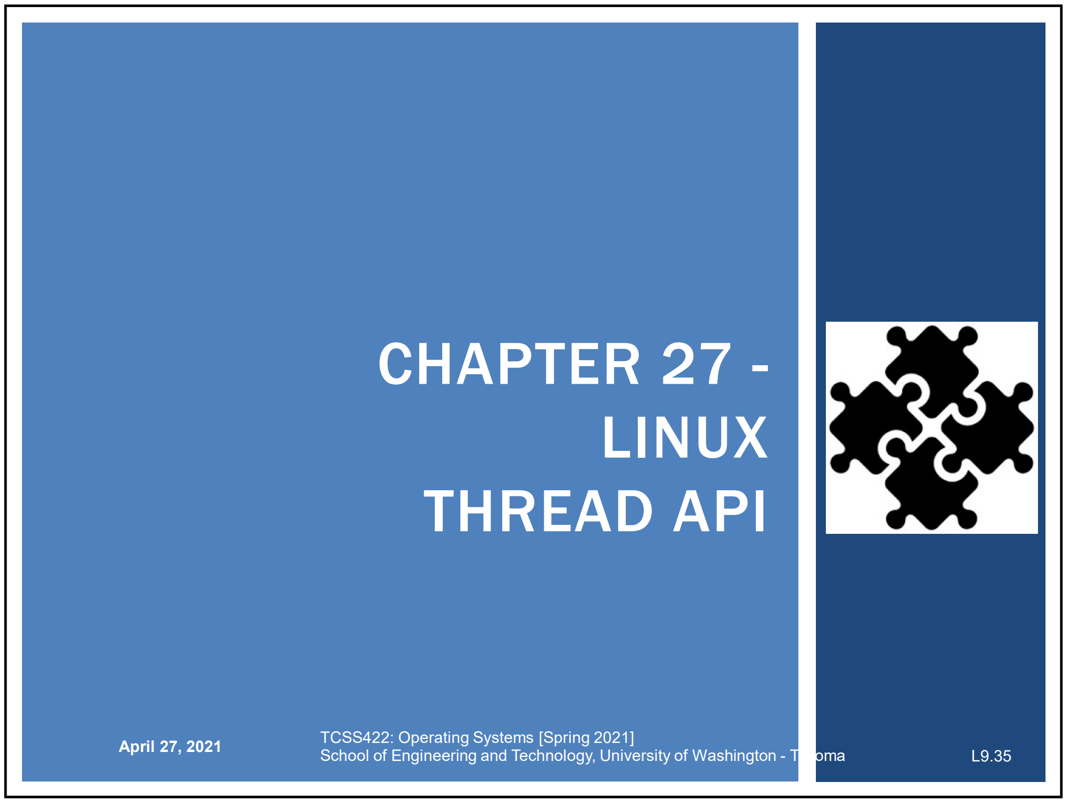
- To demonstrate how critical section(s) can be executed "atomically-*as a unit*" Chapter 27 & beyond introduce locks

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

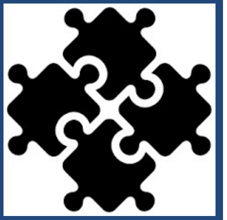
Critical section

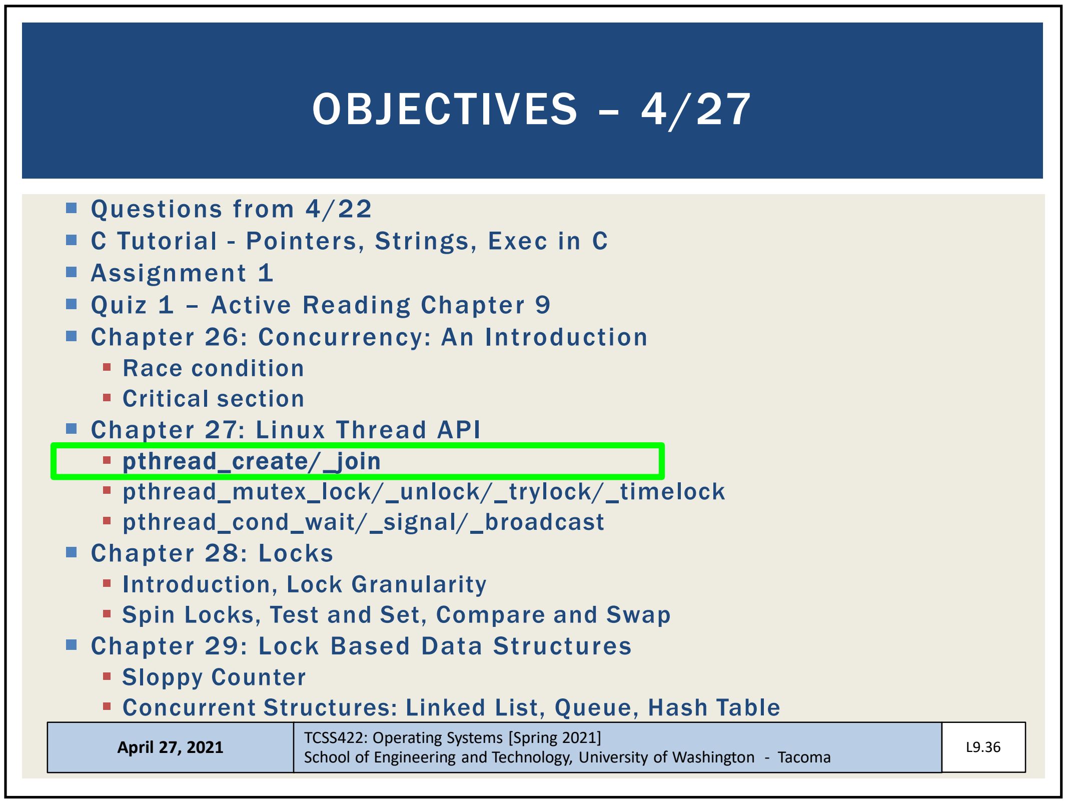
- Counter example revisited

WE WILL RETURN AT 4:53PM



CHAPTER 27 - LINUX THREAD API





OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

THREAD CREATION

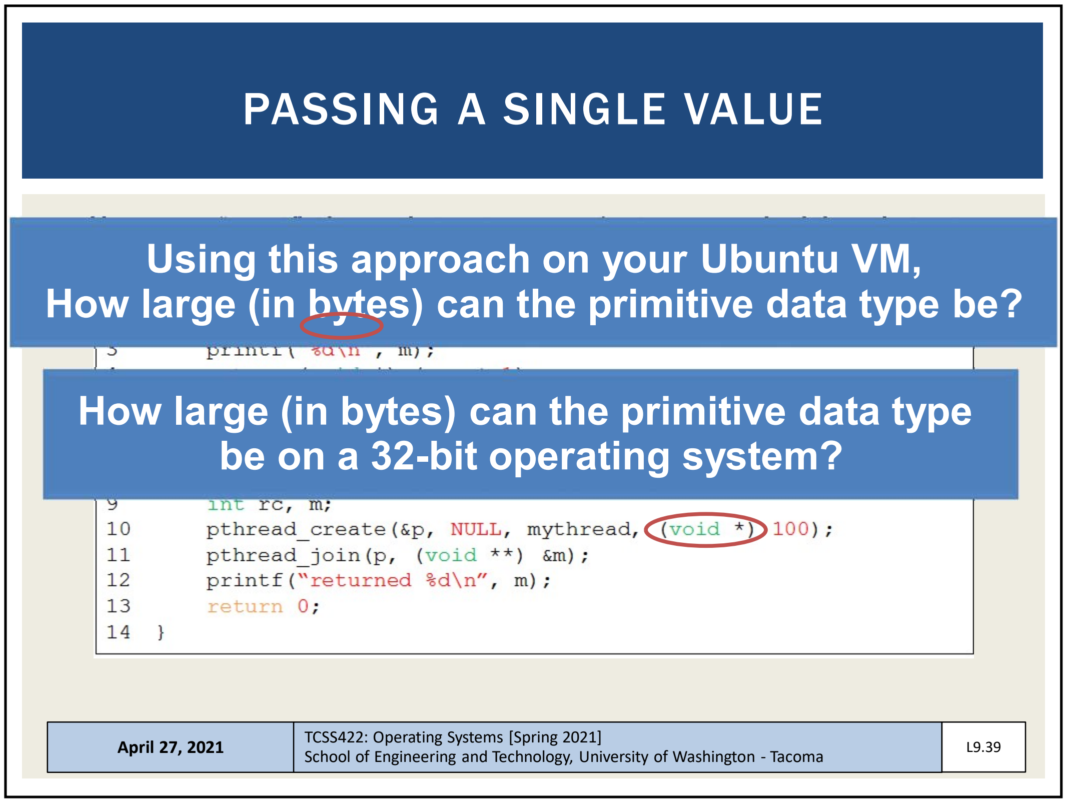
- pthread_create

```
#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*)

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);
        …
}
```



PASSING A SINGLE VALUE

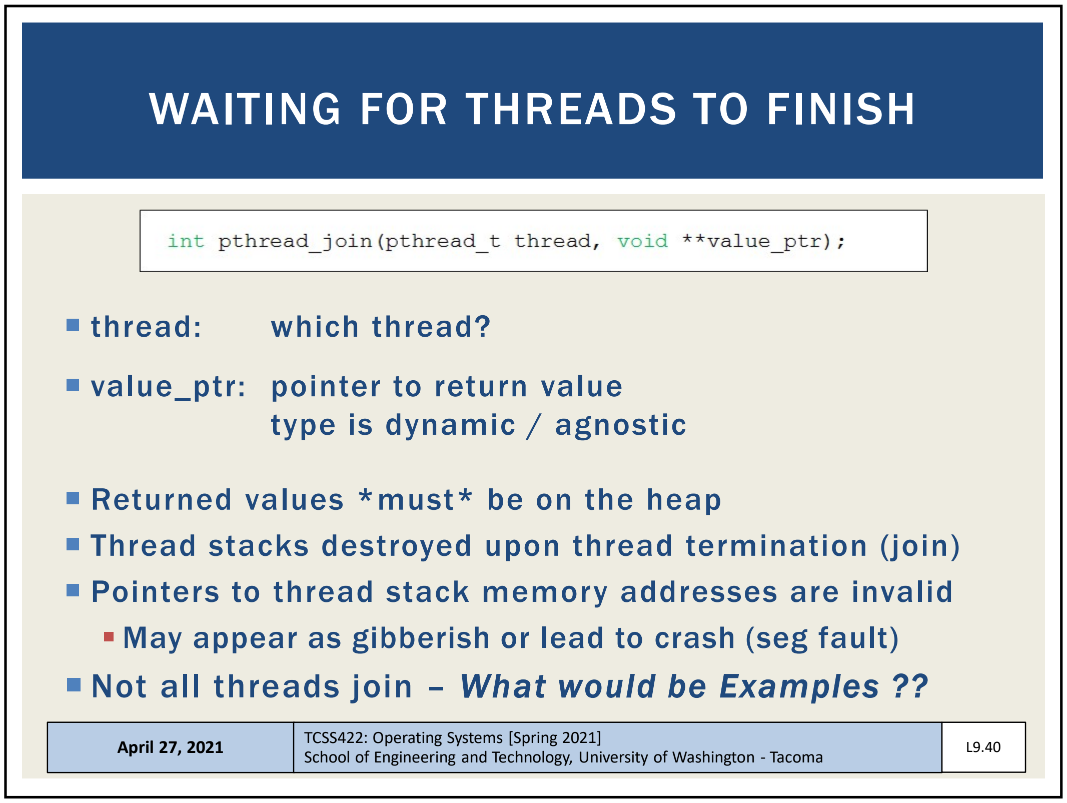
**Using this approach on your Ubuntu VM,
How large (in bytes) can the primitive data type be?**

```
        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;
}
```

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.39



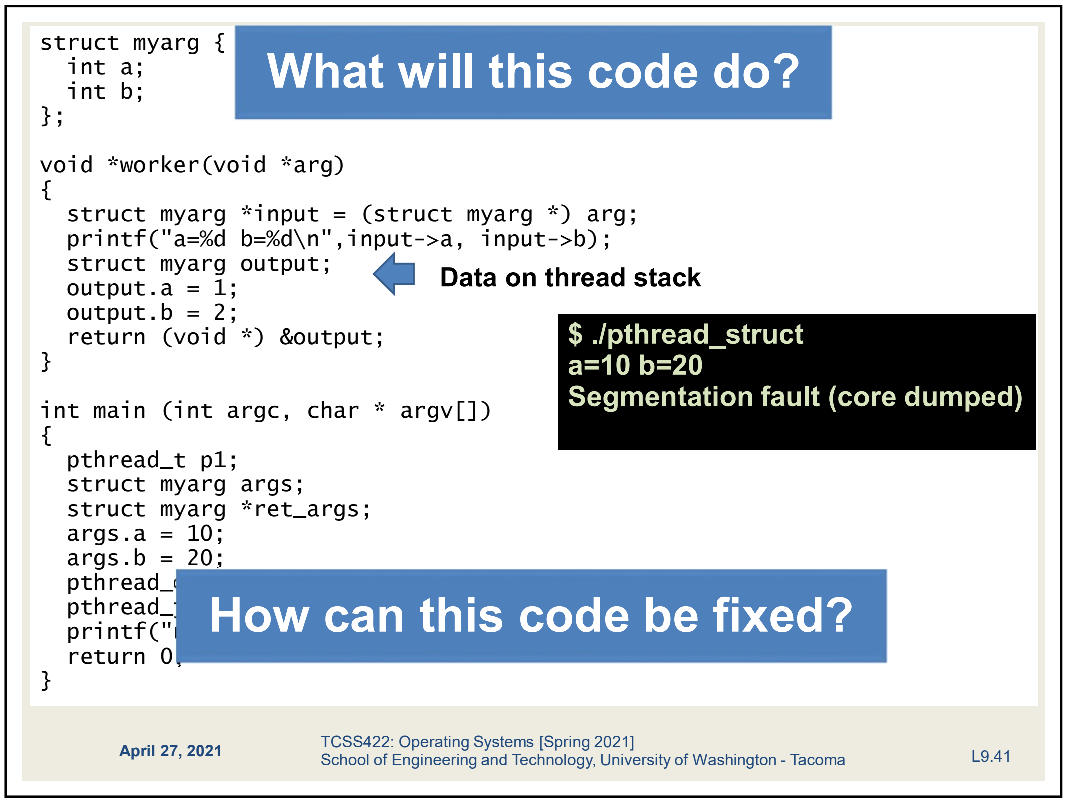
WAITING FOR THREADS TO FINISH

```
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 ??***

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.40



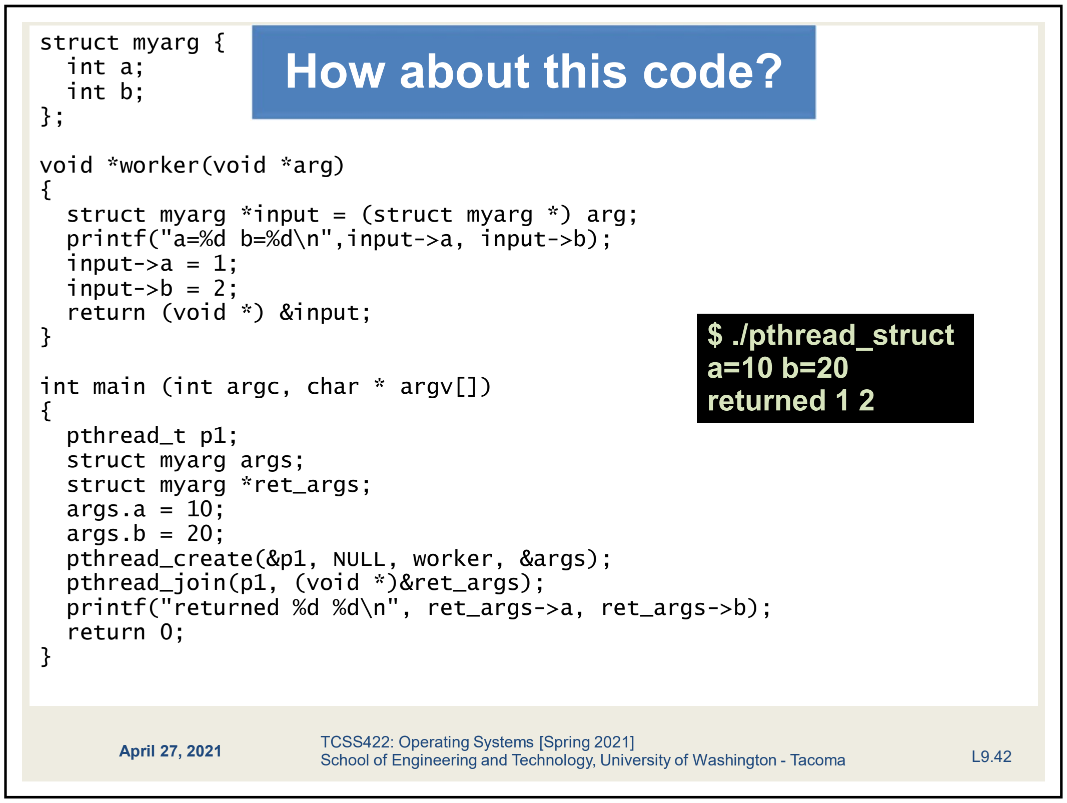
What will this code do?

```
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_c
  pthread_j
  printf("r
  return 0;
}
```

Data on thread stack

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

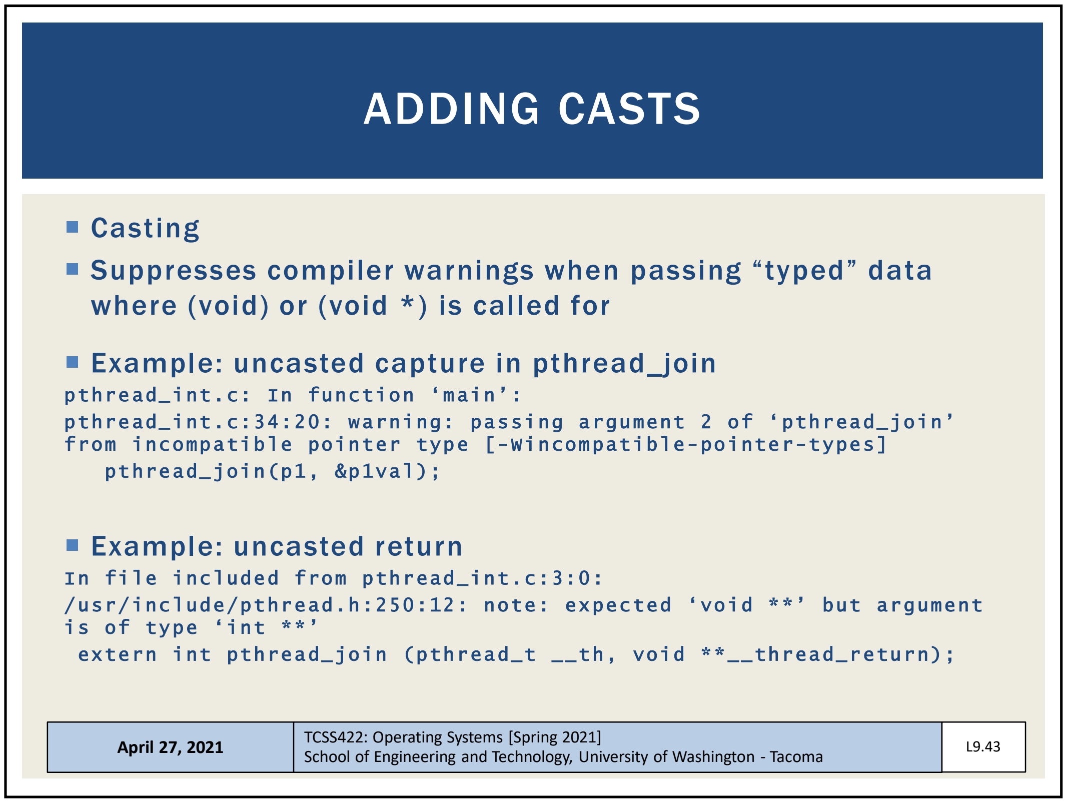
How can this code be fixed?



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

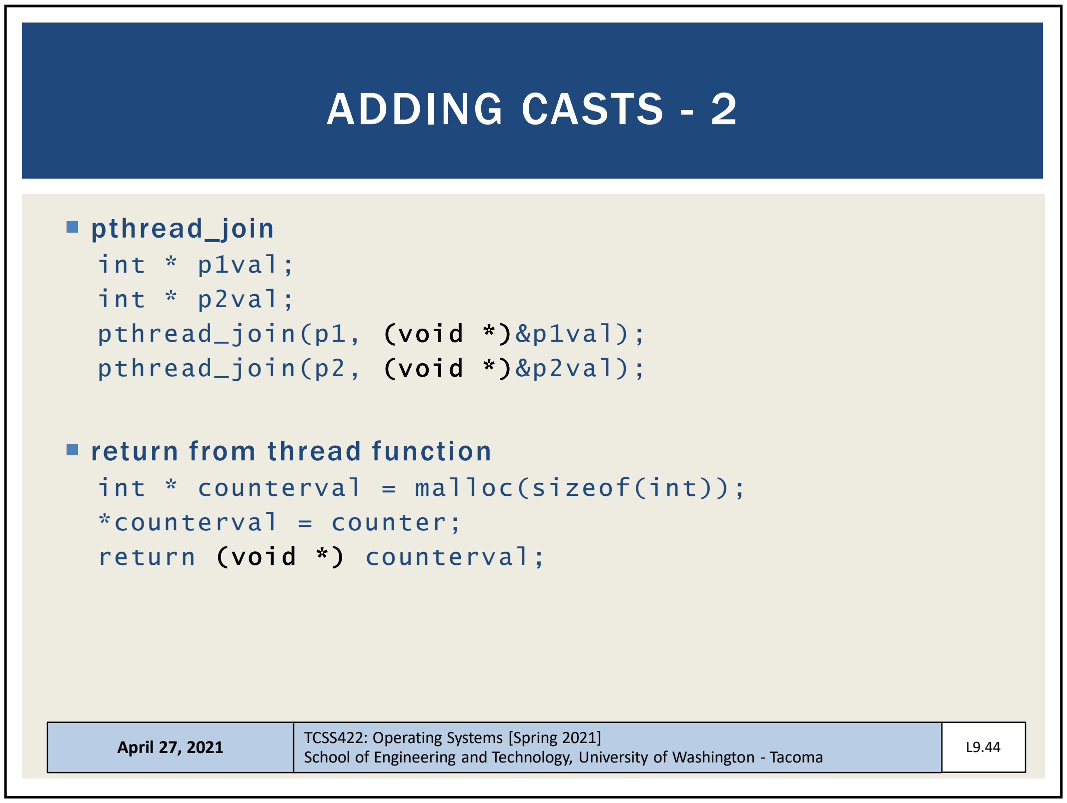
- Casting
- Suppresses compiler warnings when passing "typed" data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join

```
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);
```

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.43



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;
```

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.44

OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

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;
}
```

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

LOCK INITIALIZATION

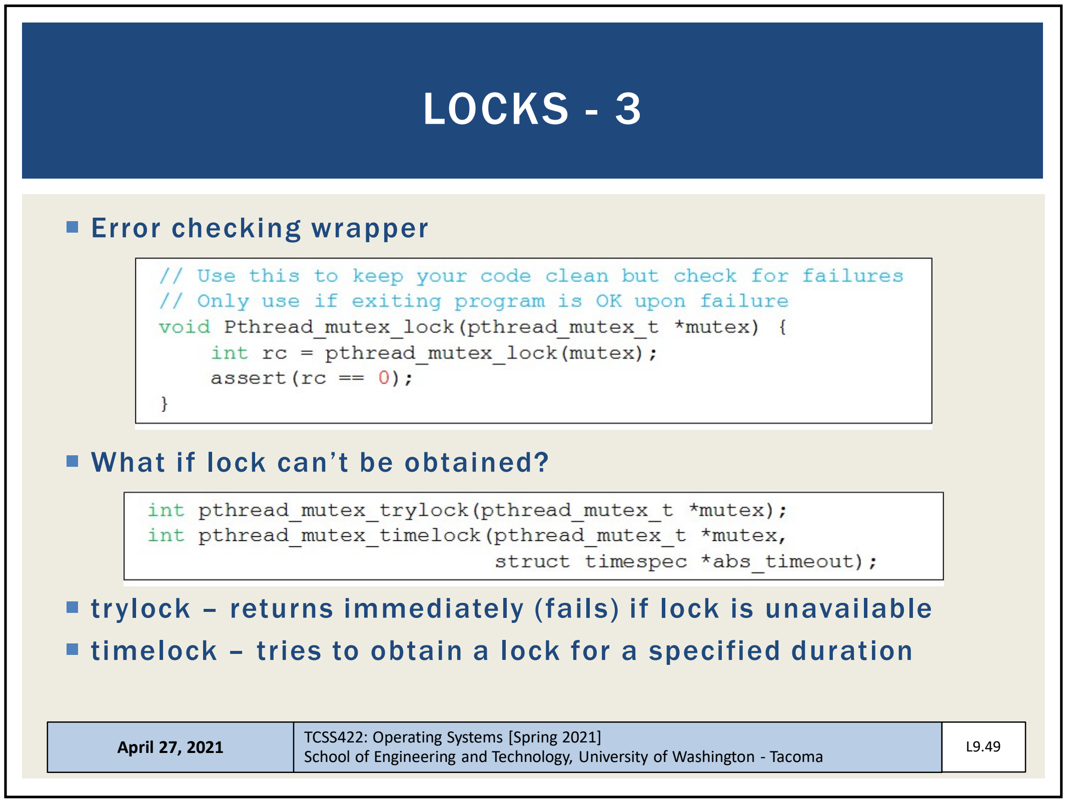
- Assigning the constant

```
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



LOCKS - 3

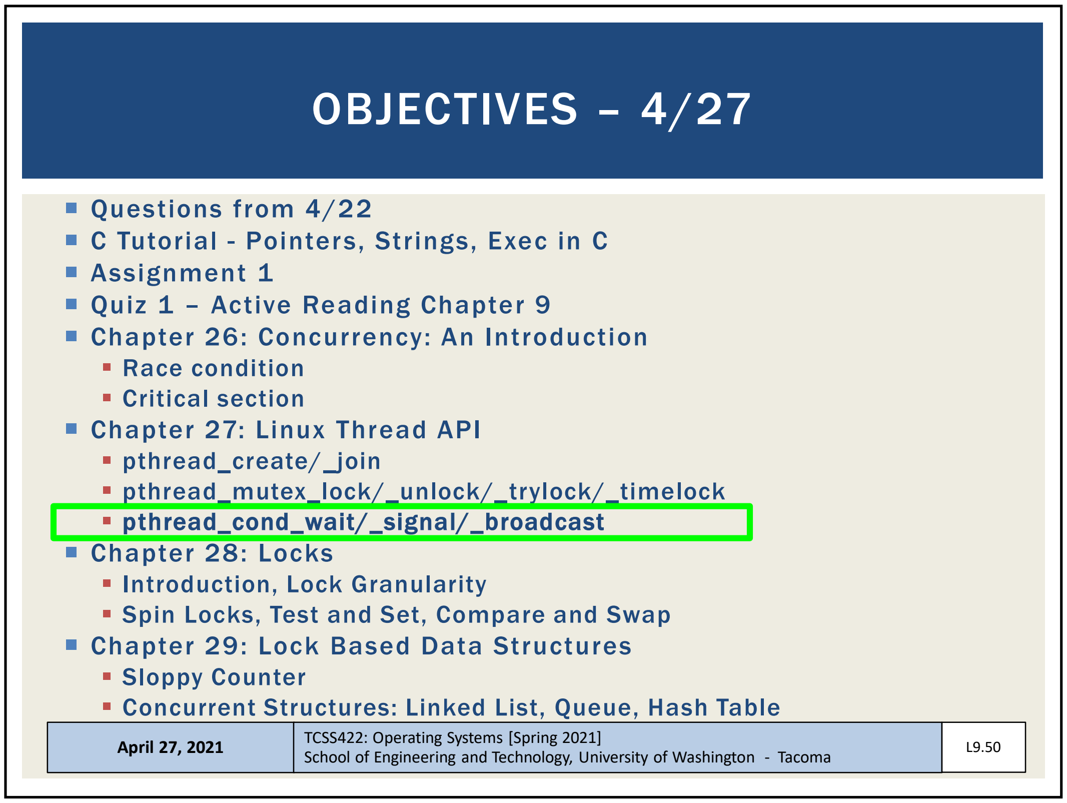
- Error checking wrapper

```
// 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



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

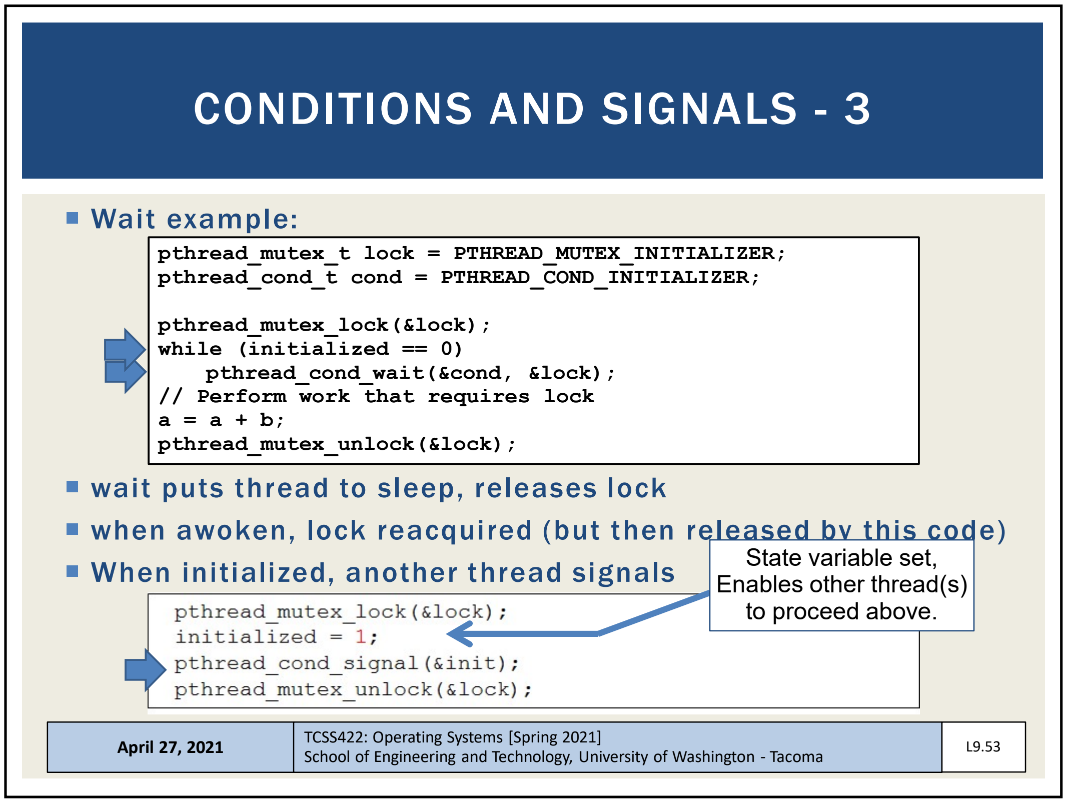
```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

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

CONDITIONS AND SIGNALS - 2

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

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



CONDITIONS AND SIGNALS - 3

- Wait example:

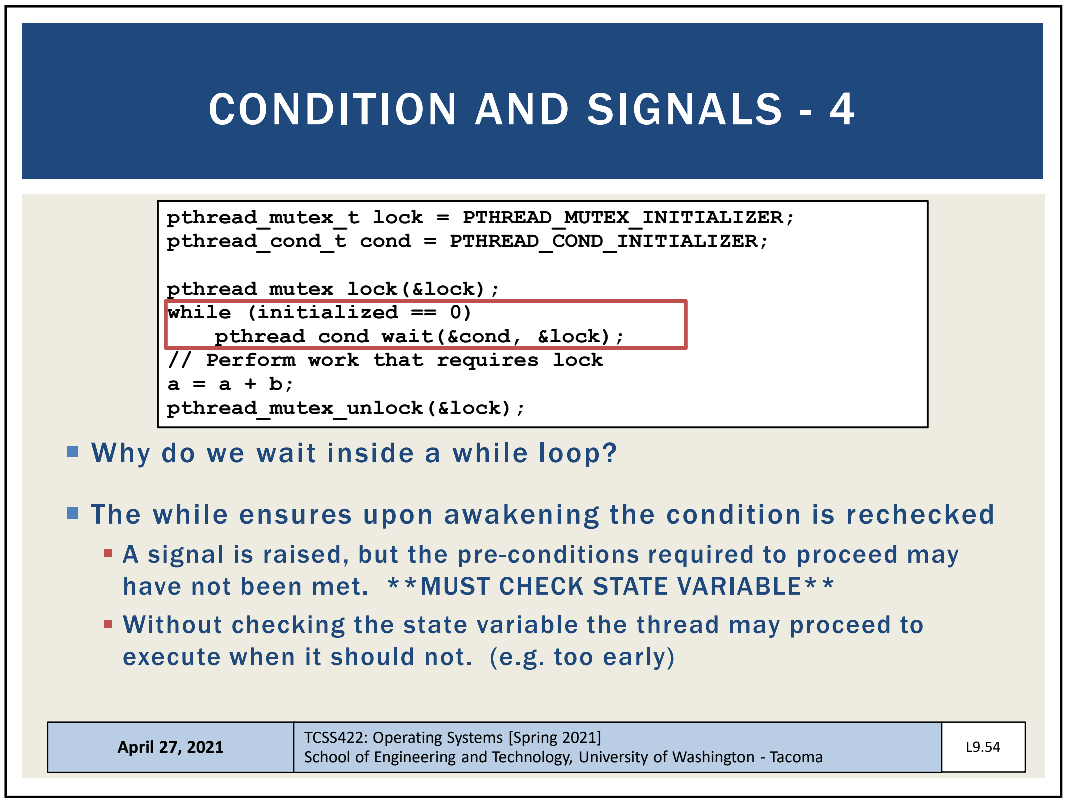
```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

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

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

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

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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.53



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)

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.54

PTHREADS LIBRARY

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

- List of pthread manpages
  - man –k pthread

SAMPLE MAKEFILE

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

binaries=pthread pthread_int pthread_lock_cond pthread_struct

all: $(binaries)

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

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

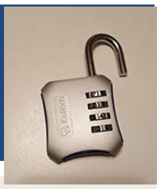
- Example builds multiple single file programs
  - All target
- pthread_mult
  - Example if multiple source files should produce a single executable
- clean target

CHAPTER 28 – LOCKS

OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

LOCKS



- Ensure critical section(s) are executed atomically-*as a unit*
  - Only one thread is allowed to execute a critical section at any given time
  - Ensures the code snippets are “mutually exclusive”
- Protect a global counter:

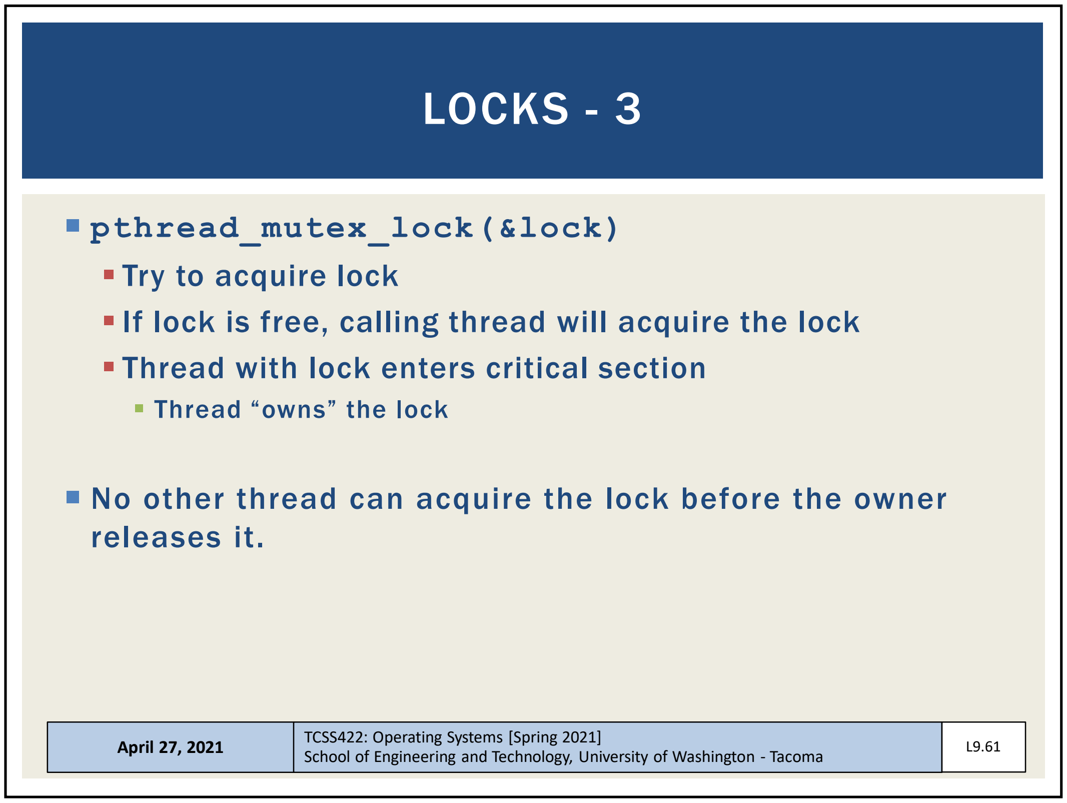
```
balance = balance + 1;
```

- A “critical section”:

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

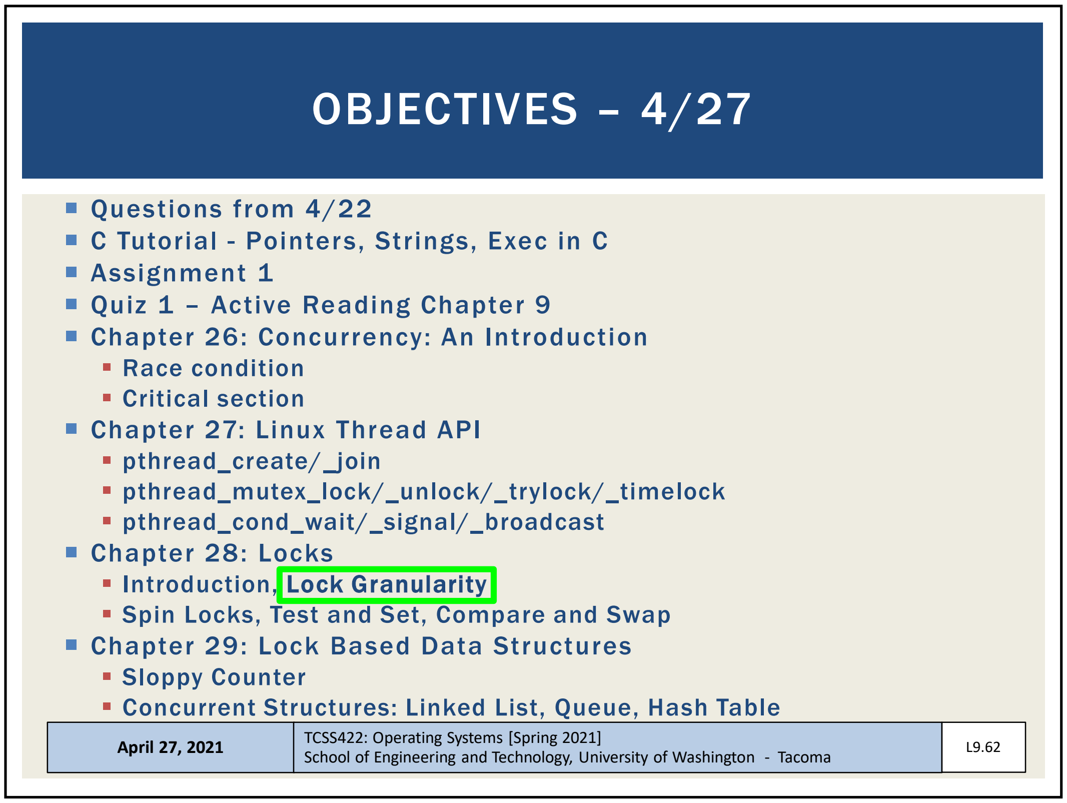
LOCKS - 2

- Lock variables are called “MUTEX”
  - Short for mutual exclusion (that’s what they guarantee)
- Lock variables store the state of the lock
- States
  - **Locked** (acquired or held)
  - **Unlocked** (available or free)
- Only 1 thread can hold a lock



LOCKS - 3

- `pthread_mutex_lock(&lock)`
  - Try to acquire lock
  - If lock is free, calling thread will acquire the lock
  - Thread with lock enters critical section
    - Thread “owns” the lock
- No other thread can acquire the lock before the owner releases it.



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

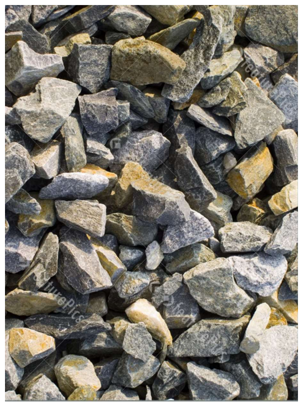
LOCKS - 4

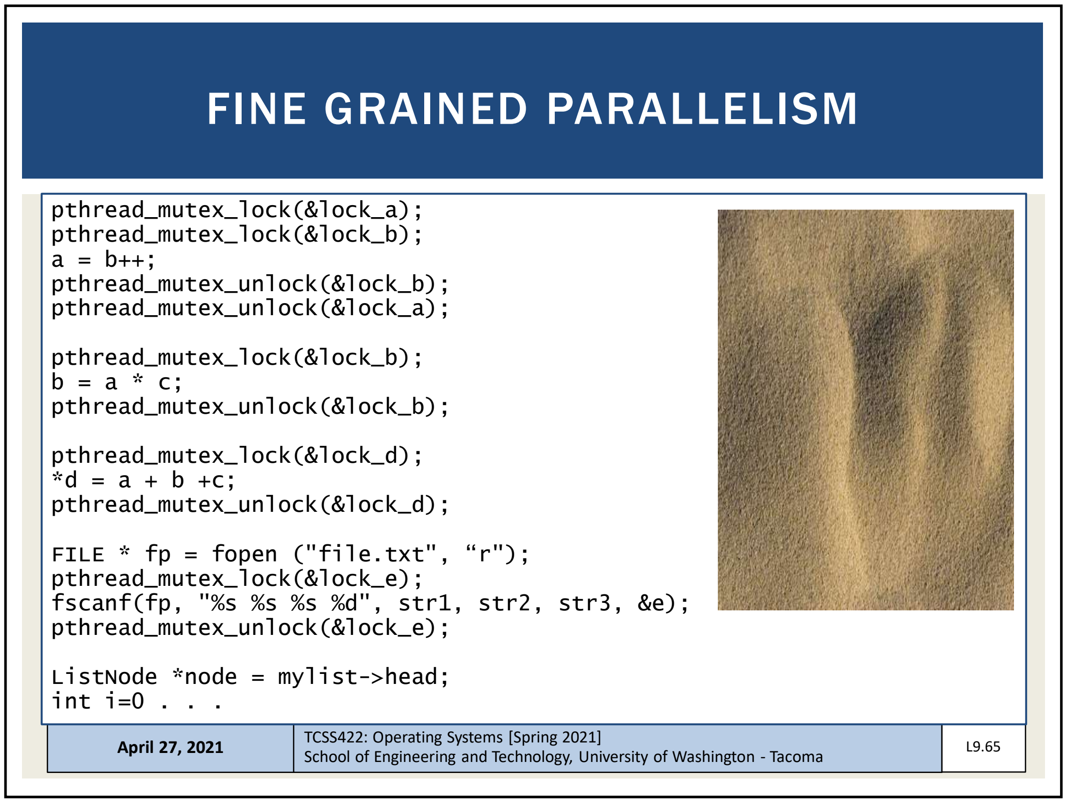
- Program can have many mutex (lock) variables to "serialize" many critical sections
- Locks are also used to protect data structures
  - Prevent multiple threads from changing the same data simultaneously
  - Programmer can make sections of code "granular"
    - ***Fine grained*** – means just one grain of sand at a time through an hour glass
  - Similar to relational database transactions
    - DB transactions prevent multiple users from modifying a table, row, field

FINE GRAINED?

- ***Is this code a good example of "fine grained parallelism"?***

```
pthread_mutex_lock(&lock);
a = b++;
b = a * c;
*d = a + b +c;
FILE * fp = fopen ("file.txt", “r");
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
ListNode *node = mylist->head;
Int i=0
while (node) {
  node->title = str1;
  node->subheading = str2;
  node->desc = str3;
  node->end = *e;
  node = node->next;
  i++
}
e = e - i;
pthread_mutex_unlock(&lock);
```

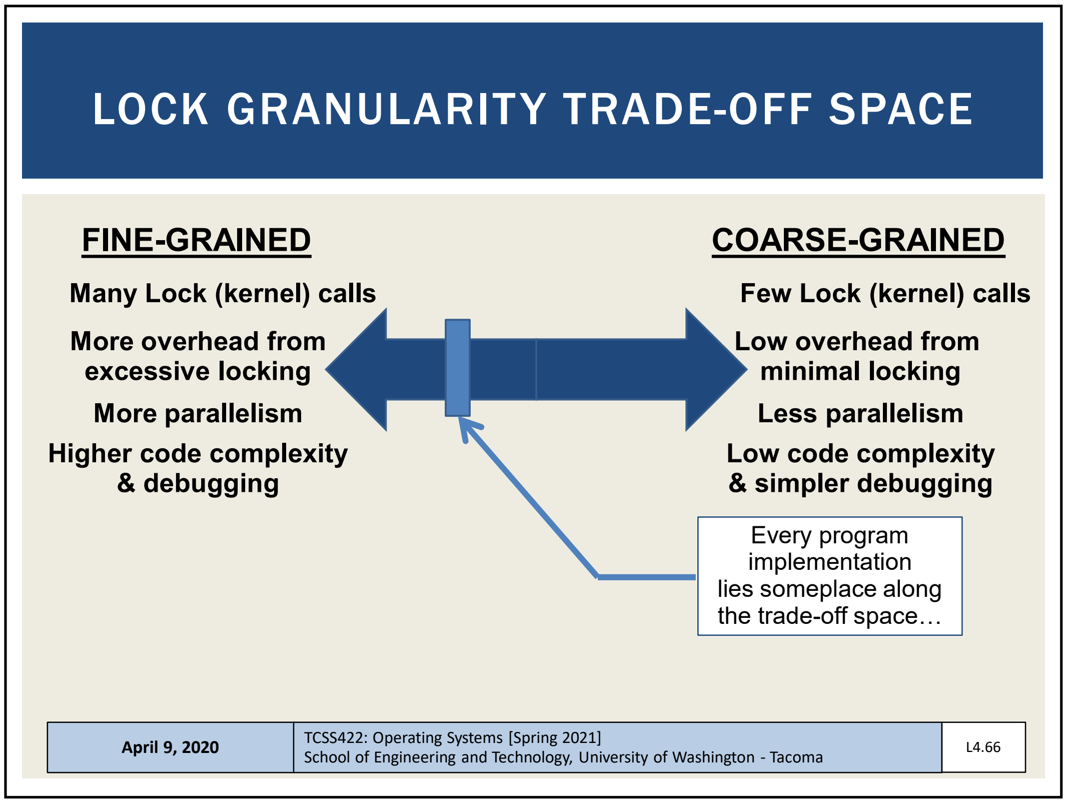




FINE GRAINED PARALLELISM

```
pthread_mutex_lock(&lock_a);
pthread_mutex_lock(&lock_b);
a = b++;
pthread_mutex_unlock(&lock_b);
pthread_mutex_unlock(&lock_a);

pthread_mutex_lock(&lock_b);
b = a * c;
pthread_mutex_unlock(&lock_b);

pthread_mutex_lock(&lock_d);
*d = a + b +c;
pthread_mutex_unlock(&lock_d);

FILE * fp = fopen ("file.txt", “r");
pthread_mutex_lock(&lock_e);
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
pthread_mutex_unlock(&lock_e);

ListNode *node = mylist->head;
int i=0 . . .
```

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.65



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…

April 9, 2020 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L4.66

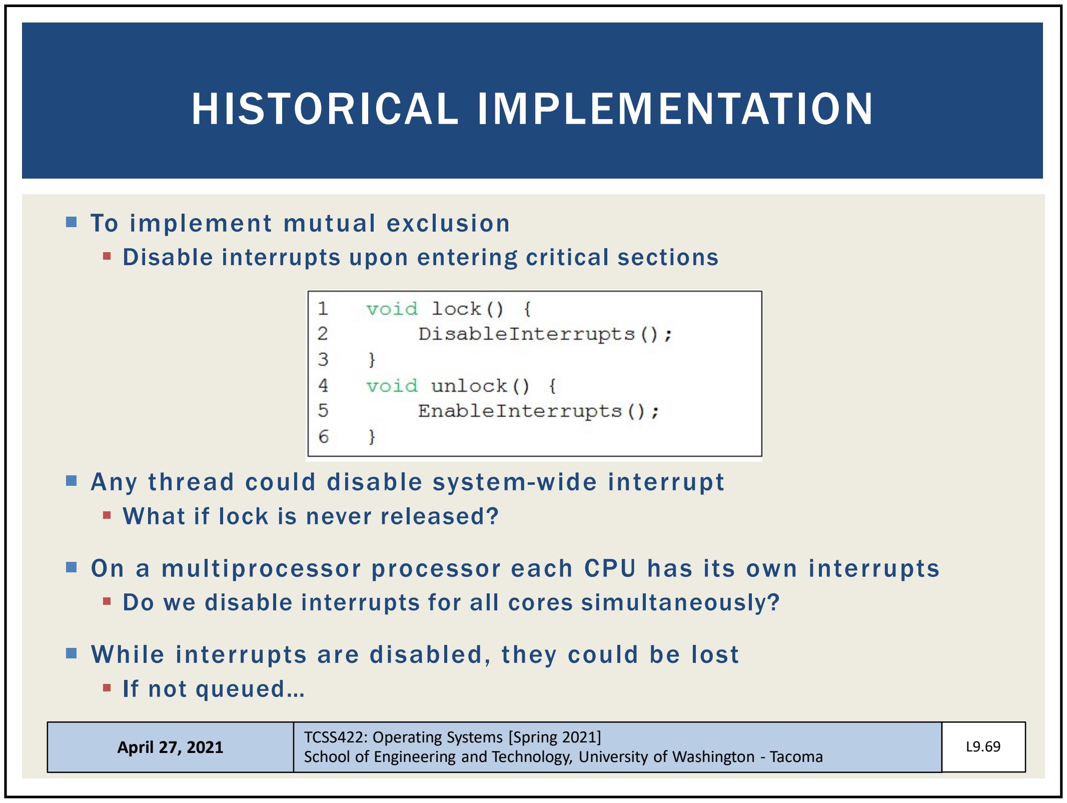
EVALUATING LOCK IMPLEMENTATIONS

What makes a good lock?

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

BUILDING LOCKS

- Locks require hardware support
  - To minimize overhead, ensure fairness and correctness
  - Special "atomic-*as a unit*" instructions to support lock implementation
  - Atomic-*as a unit* exchange instruction
    - `XCHG`
  - Compare and exchange instruction
    - `CMPXCHG`
    - `CMPXCHG8B`
    - `CMPXCHG16B`

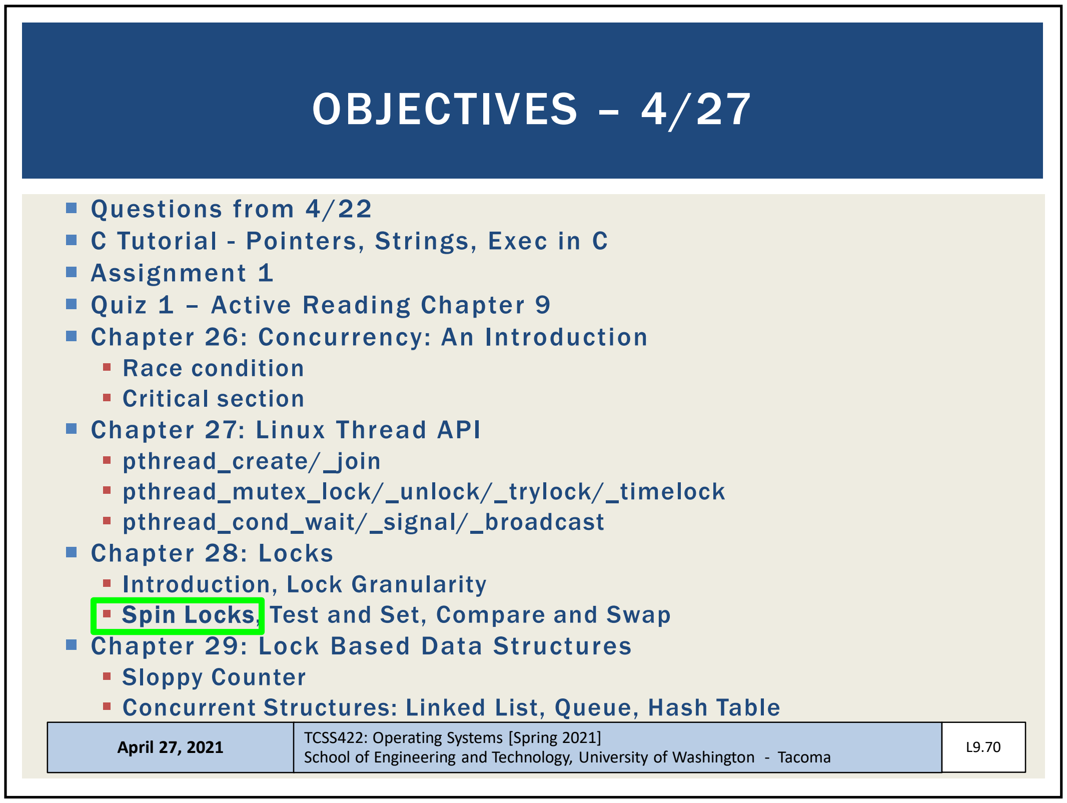


HISTORICAL IMPLEMENTATION

- To implement mutual exclusion
  - Disable interrupts upon entering critical sections

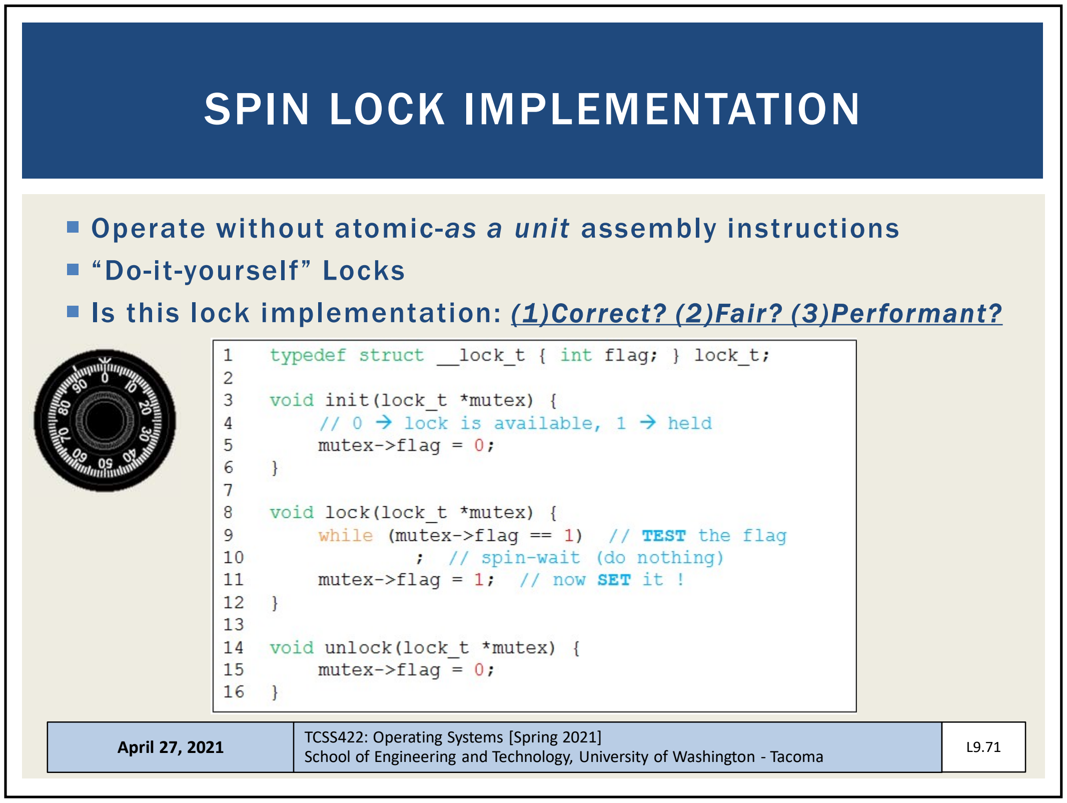
```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- Any thread could disable system-wide interrupt
  - What if lock is never released?
- On a multiprocessor processor each CPU has its own interrupts
  - Do we disable interrupts for all cores simultaneously?
- While interrupts are disabled, they could be lost
  - If not queued…



OBJECTIVES – 4/27

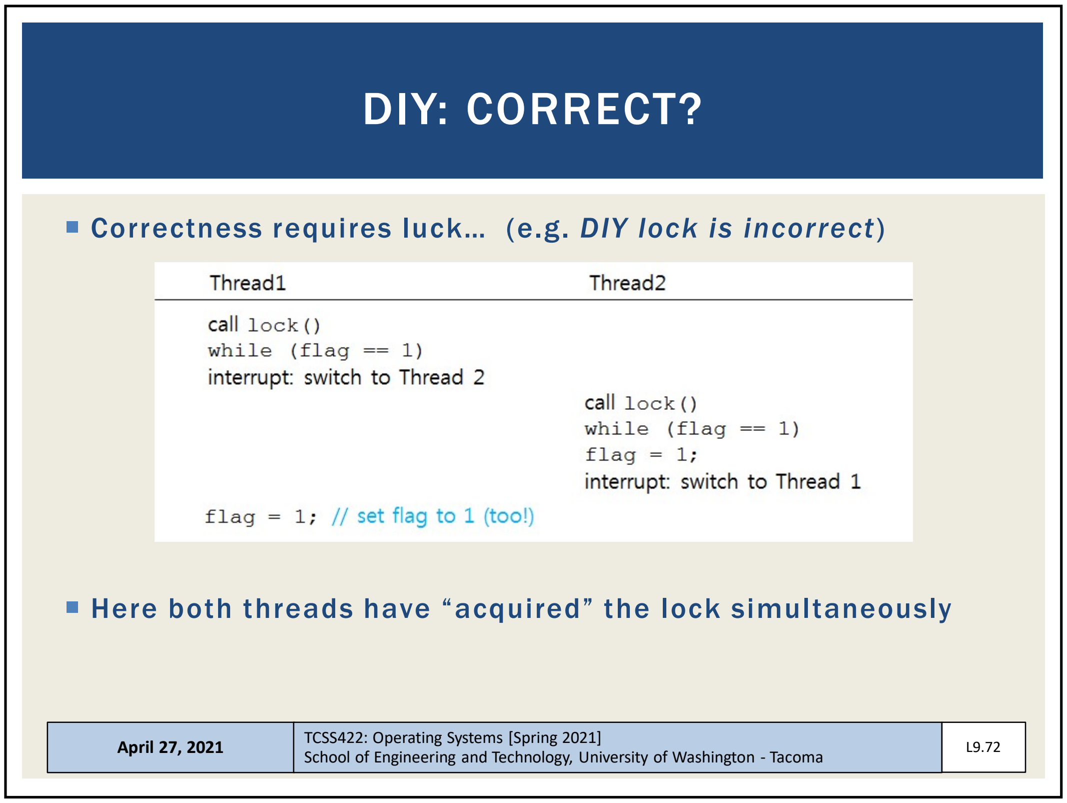
- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - 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



SPIN LOCK IMPLEMENTATION

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

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
            ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```



DIY: CORRECT?

- Correctness requires luck… (e.g. *DIY lock is incorrect*)

Thread1	Thread2
call `lock()` `while (flag == 1)` interrupt: switch to Thread 2	

	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1

`flag = 1;` // set flag to 1 (too!)	

- Here both threads have "acquired" the lock simultaneously

DIY: PERFORMANT?

```
void lock(lock_t *mutex)
{
  while (mutex->flag == 1);    // while lock is unavailable, wait…
  mutex->flag = 1;
}
```

- What is wrong with while(<cond>); ?

- Spin-waiting wastes time actively waiting for another thread
- while (1); will "peg" a CPU core at 100%
  - Continuously loops, and evaluates mutex->flag value…
  - Generates heat…

OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

TEST-AND-SET INSTRUCTION

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

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

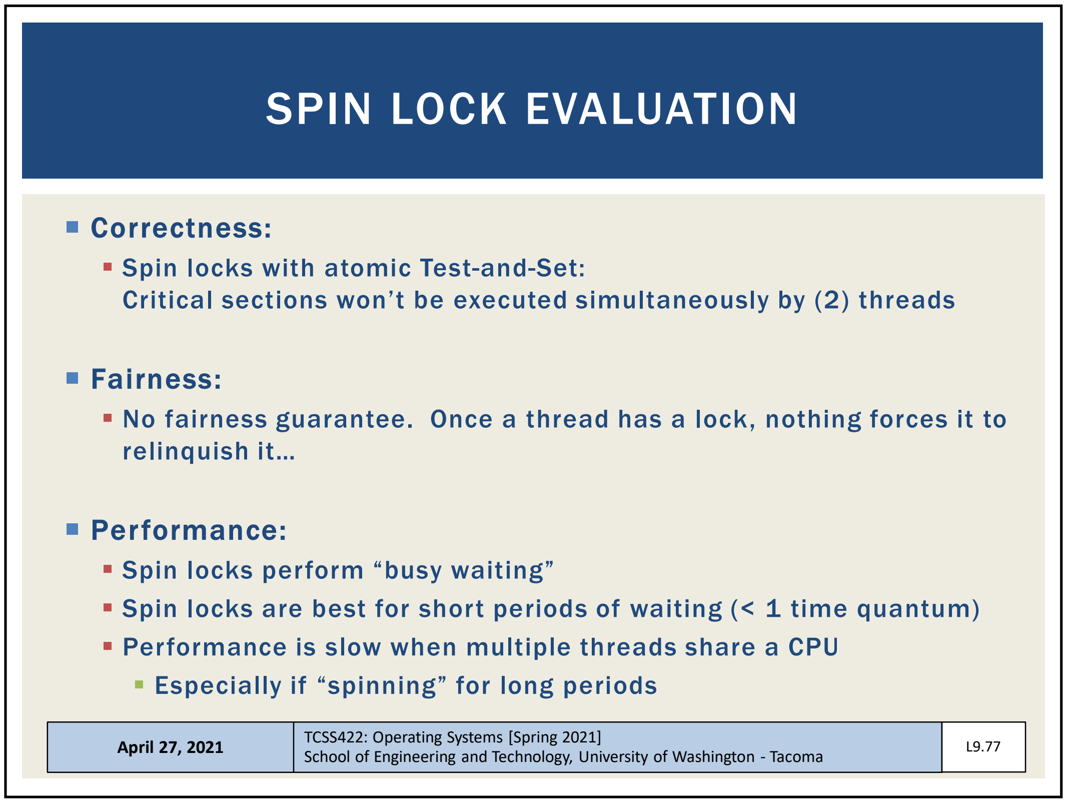
- lock() method checks that TestAndSet doesn't return 1
- Comparison is in the caller

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

DIY: TEST-AND-SET - 2

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

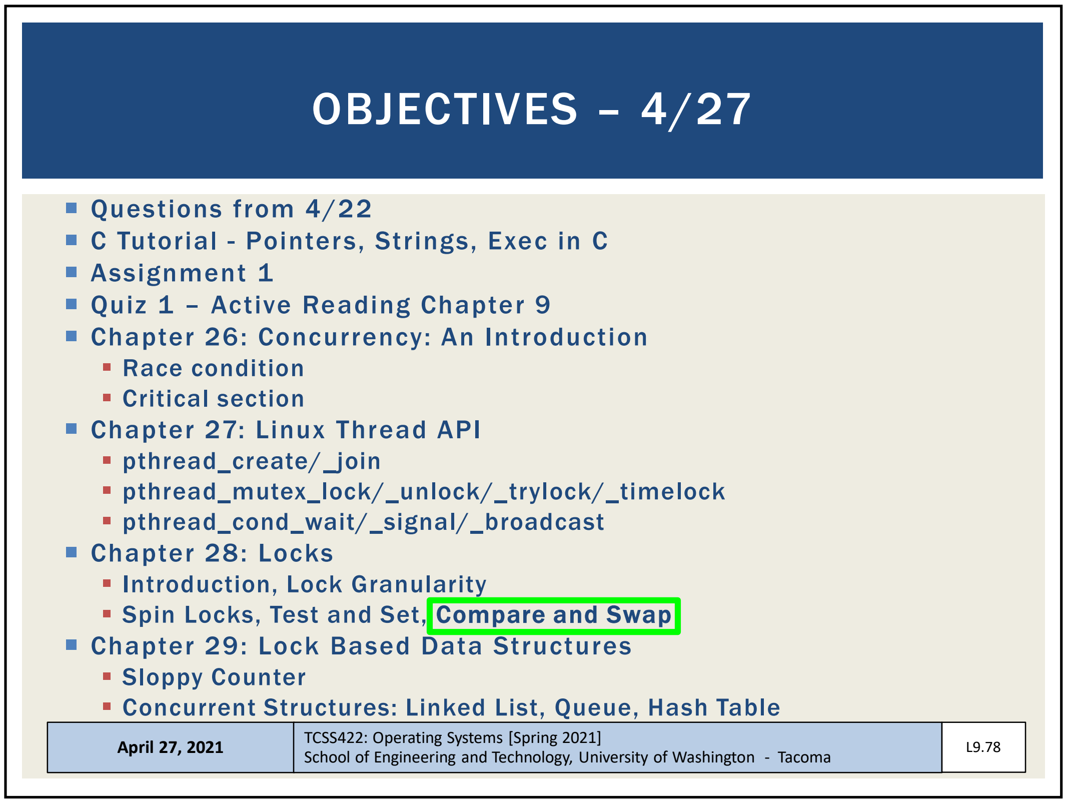
```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
            ;        // spin-wait
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```



SPIN LOCK EVALUATION

- Correctness:
  - Spin locks with atomic Test-and-Set:
    Critical sections won't be executed simultaneously by (2) threads
- Fairness:
  - No fairness guarantee. Once a thread has a lock, nothing forces it to relinquish it…
- Performance:
  - Spin locks perform "busy waiting"
  - Spin locks are best for short periods of waiting (< 1 time quantum)
  - Performance is slow when multiple threads share a CPU
    - Especially if "spinning" for long periods

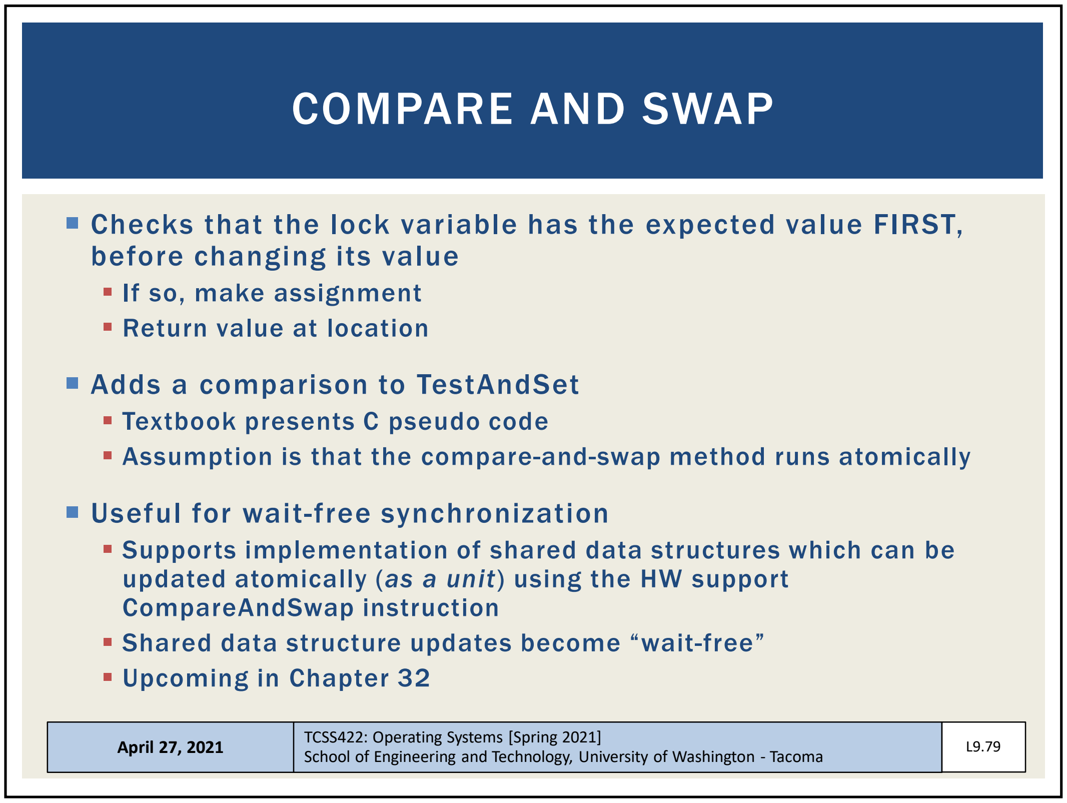
April 27, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.77



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

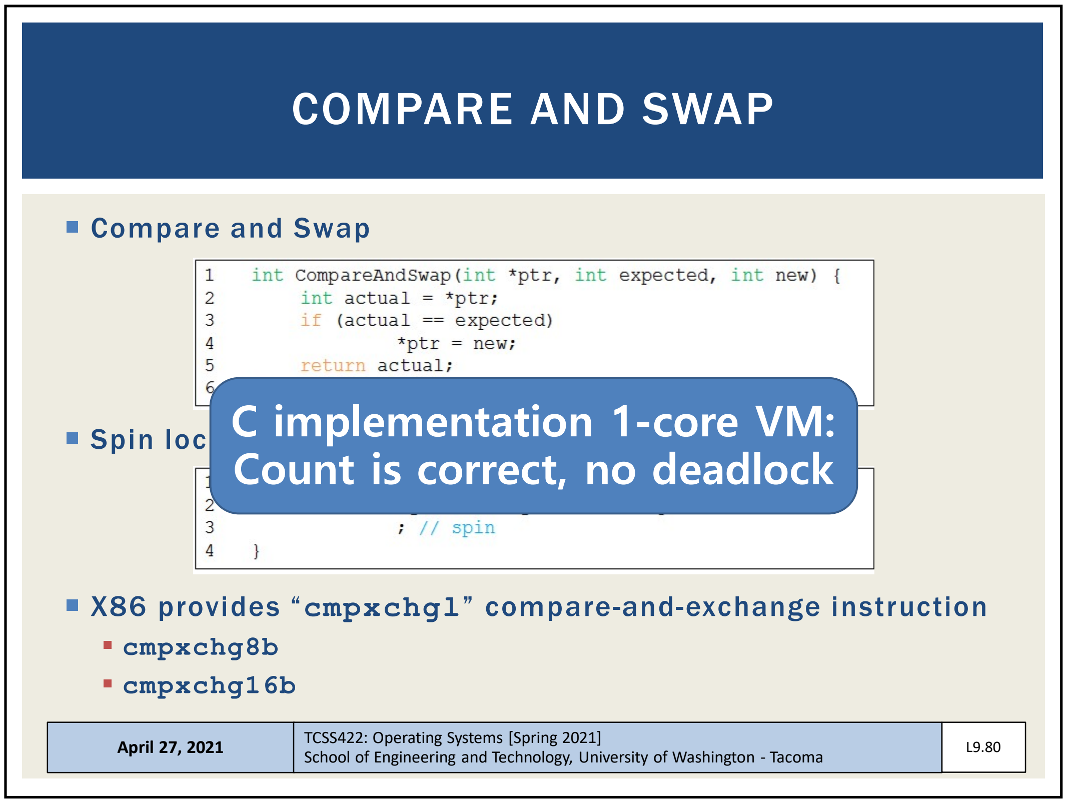
April 27, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.78



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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.79



COMPARE AND SWAP

- Compare and Swap

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

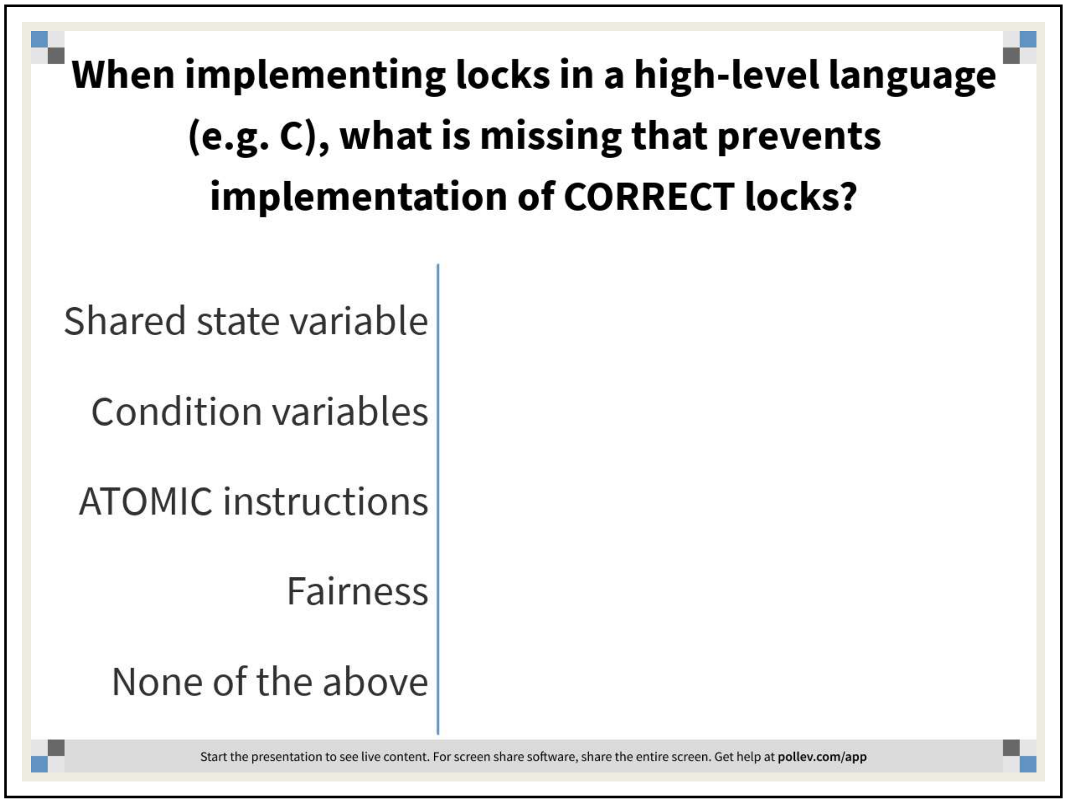
- Spin loc

```
            ; // spin
}
```

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

- X86 provides “`cmpxchgl`” compare-and-exchange instruction
  - `cmpxchg8b`
  - `cmpxchg16b`

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.80



When implementing locks in a high-level language (e.g. C), what is missing that prevents implementation of CORRECT locks?

Shared state variable

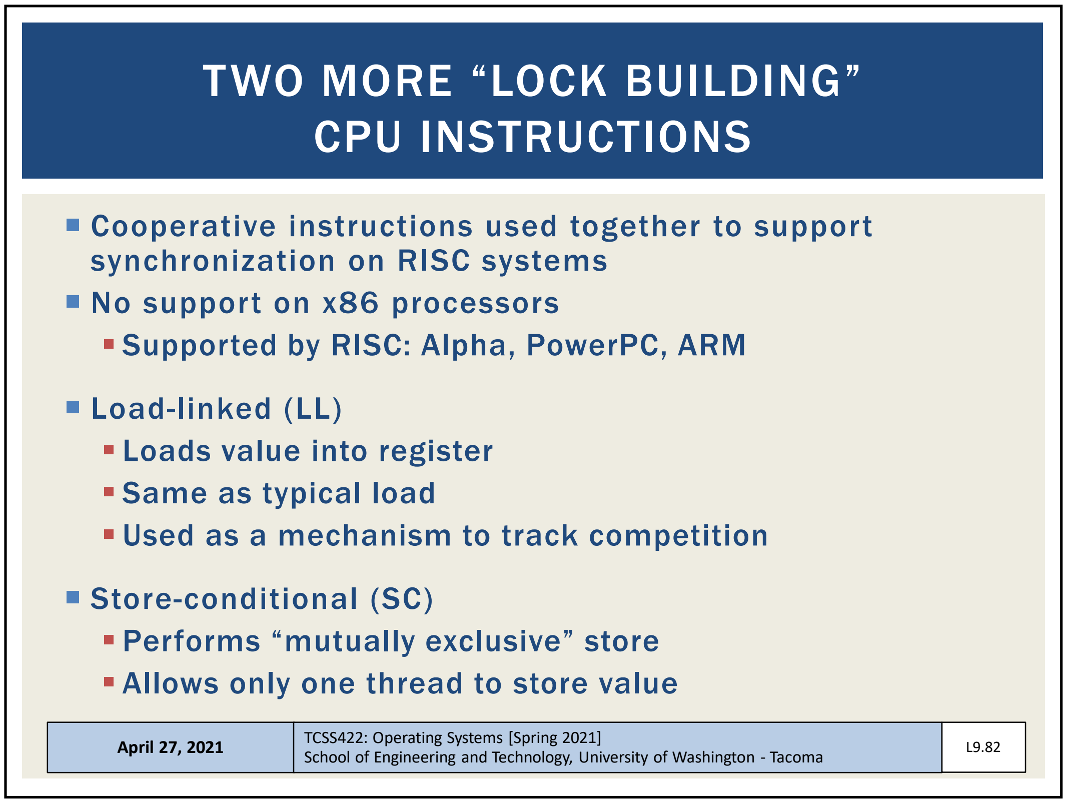
Condition variables

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**



TWO MORE "LOCK BUILDING" CPU INSTRUCTIONS

- Cooperative instructions used together to support synchronization on RISC systems
- No support on x86 processors
  - Supported by RISC: Alpha, PowerPC, ARM
- Load-linked (LL)
  - Loads value into register
  - Same as typical load
  - Used as a mechanism to track competition
- Store-conditional (SC)
  - Performs "mutually exclusive" store
  - Allows only one thread to store value

April 27, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.82

LL/SC LOCK

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
            *ptr = value;
            return 1; // success!
    } else {
            return 0; // failed to update
    }
}
```

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

LL/SC LOCK - 2

```
void lock(lock_t *lock) {
    while (1) {
            while (LoadLinked(&lock->flag) == 1)
                    ; // spin until it's zero
            if (StoreConditional(&lock->flag, 1) == 1)
                    return; // if set-it-to-1 was a success: all done
                            otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- Two instruction lock

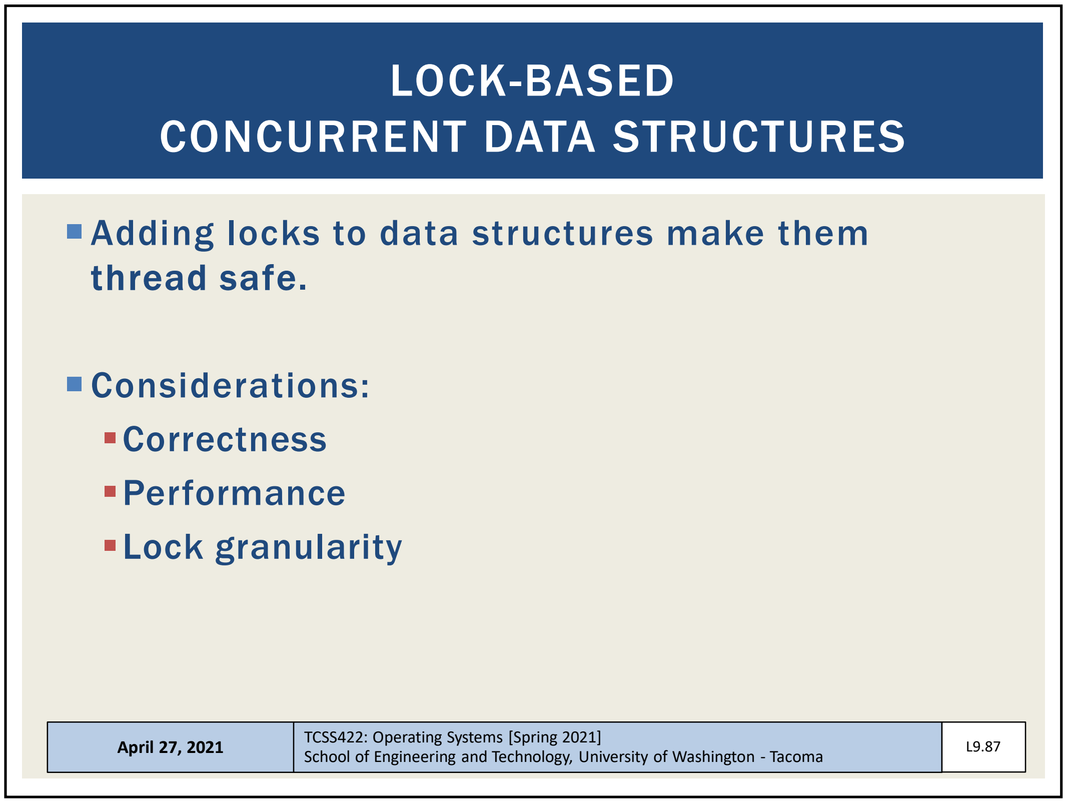
CHAPTER 29 – LOCK BASED DATA STRUCTURES

April 27, 2021 TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L9.85

OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

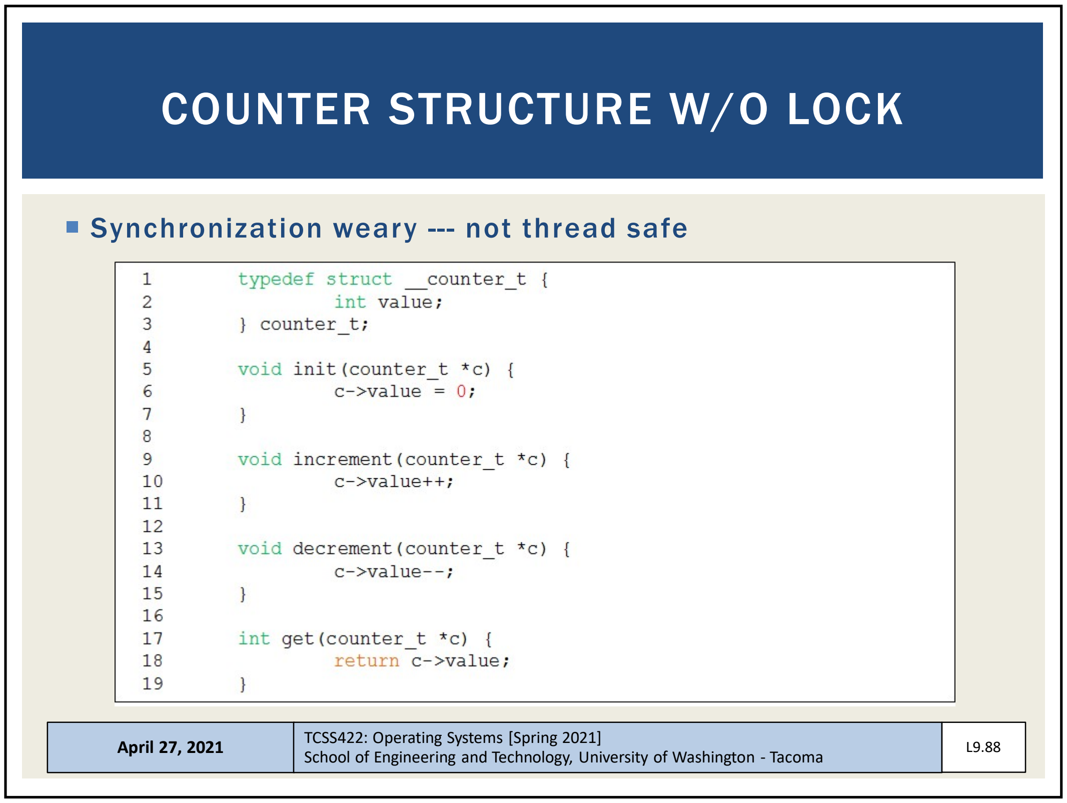
April 27, 2021 TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L9.86



LOCK-BASED CONCURRENT DATA STRUCTURES

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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.87



COUNTER STRUCTURE W/O LOCK

- Synchronization weary --- not thread safe

```
typedef struct __counter_t {
        int value;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
}

void increment(counter_t *c) {
        c->value++;
}

void decrement(counter_t *c) {
        c->value--;
}

int get(counter_t *c) {
        return c->value;
}
```

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.88

CONCURRENT COUNTER

```
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
        Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value++;
        Pthread_mutex_unlock(&c->lock);
}

```

- Add lock to the counter
- Require lock to change data

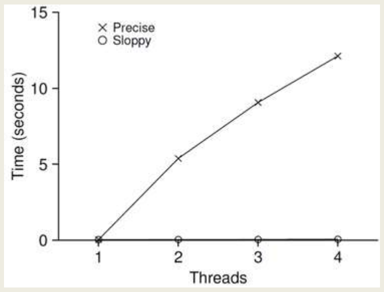
CONCURRENT COUNTER - 2

- Decrease counter
- Get value

```
(Cont.)
void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
}
```

CONCURRENT COUNTERS - PERFORMANCE

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

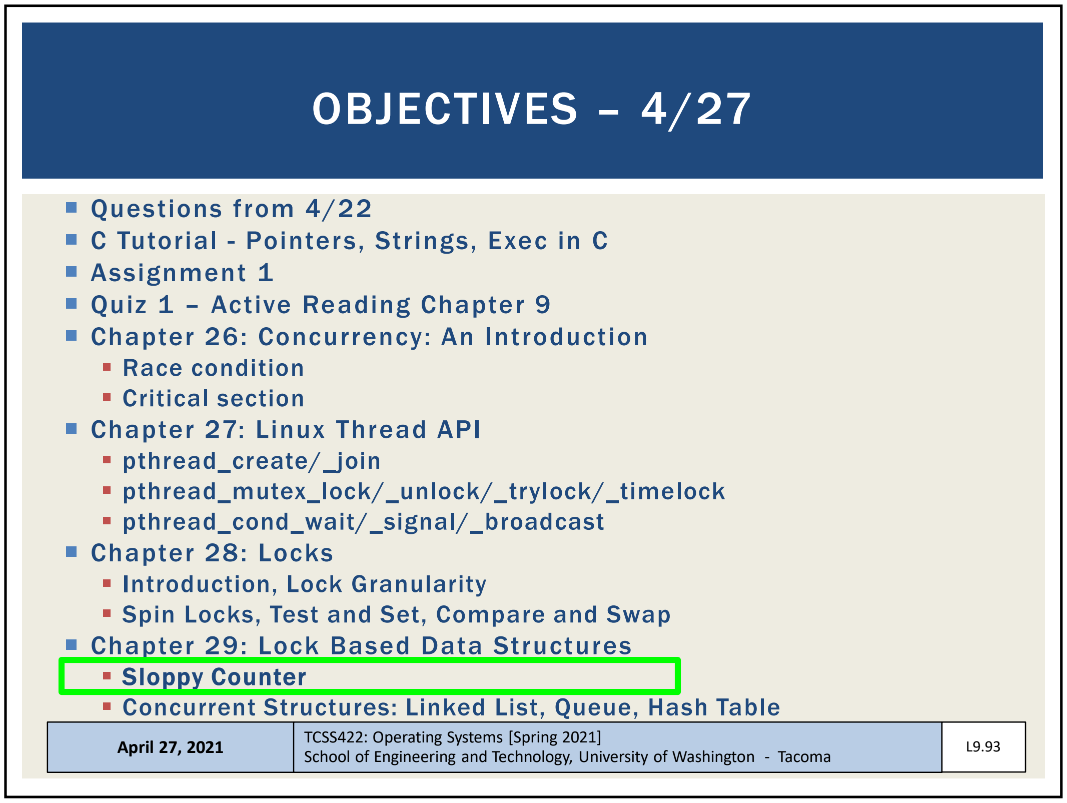


Traditional vs. sloppy counter
Sloppy Threshold (S) = 1024

Synchronized counter scales poorly.

PERFECT SCALING

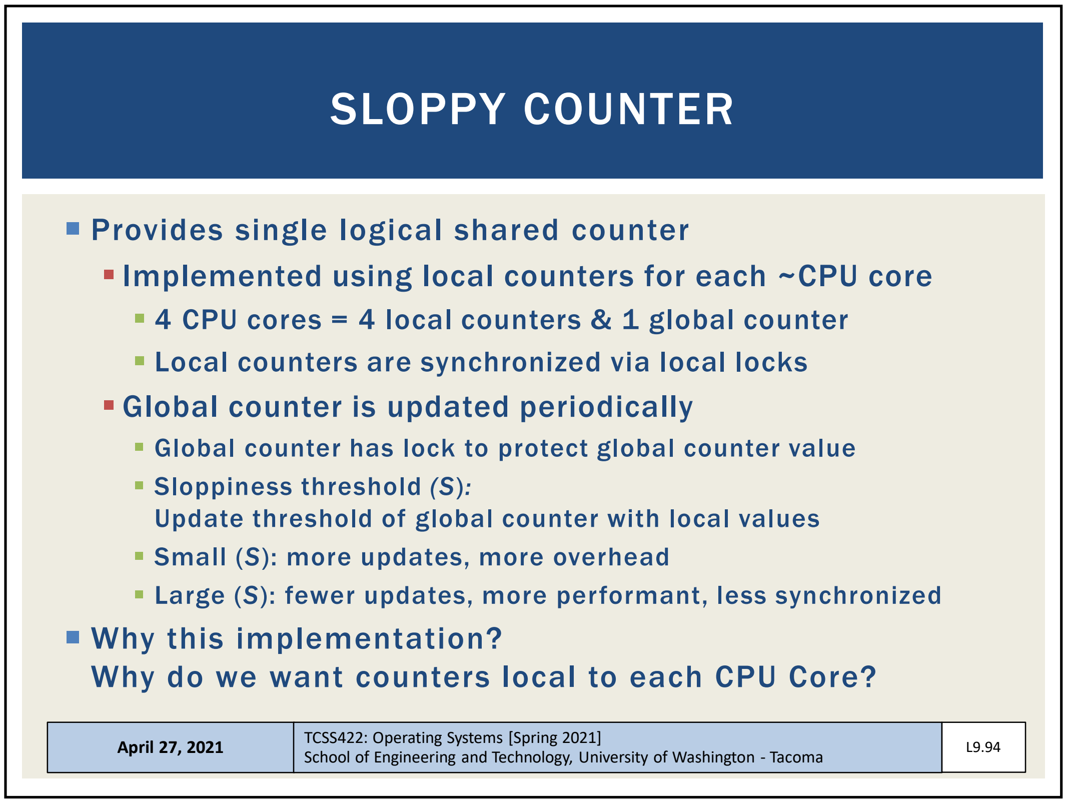
- Achieve (N) performance gain with (N) additional resources
- Throughput:
- Transactions per second (tps)
- 1 core
- N = 100 tps
- 10 cores (x10)
- N = 1000 tps (x10)



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

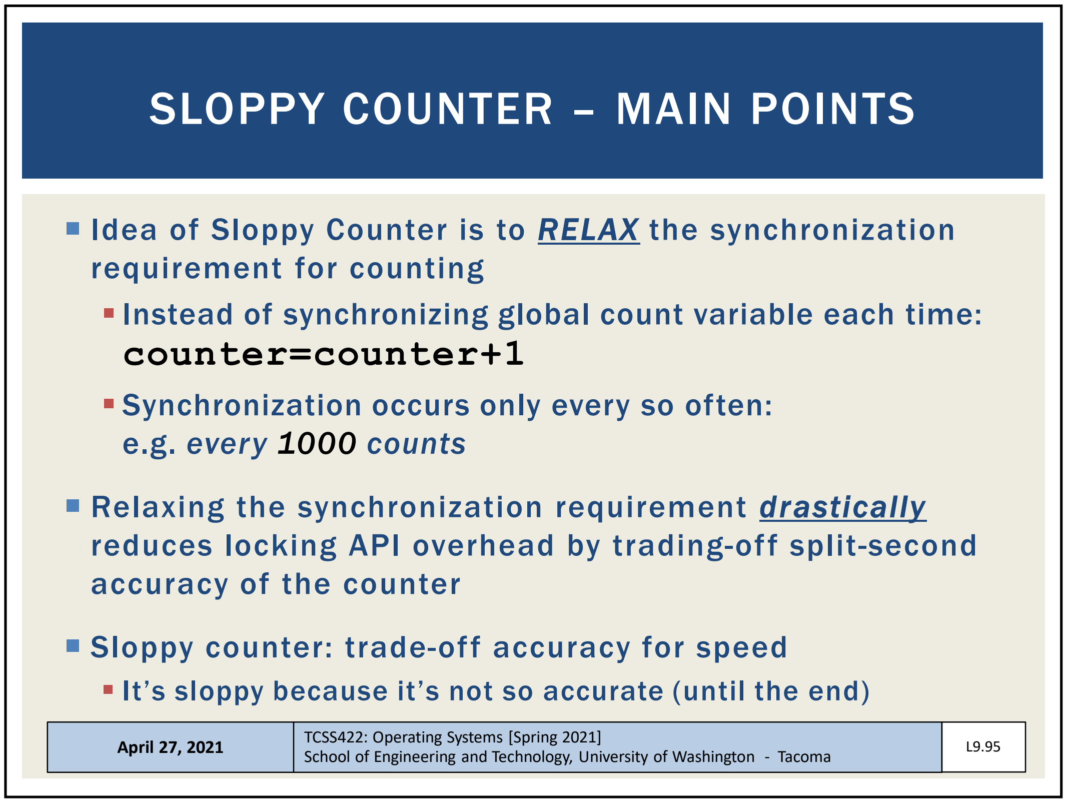
April 27, 2021 | TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma | L9.93



SLOPPY COUNTER

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

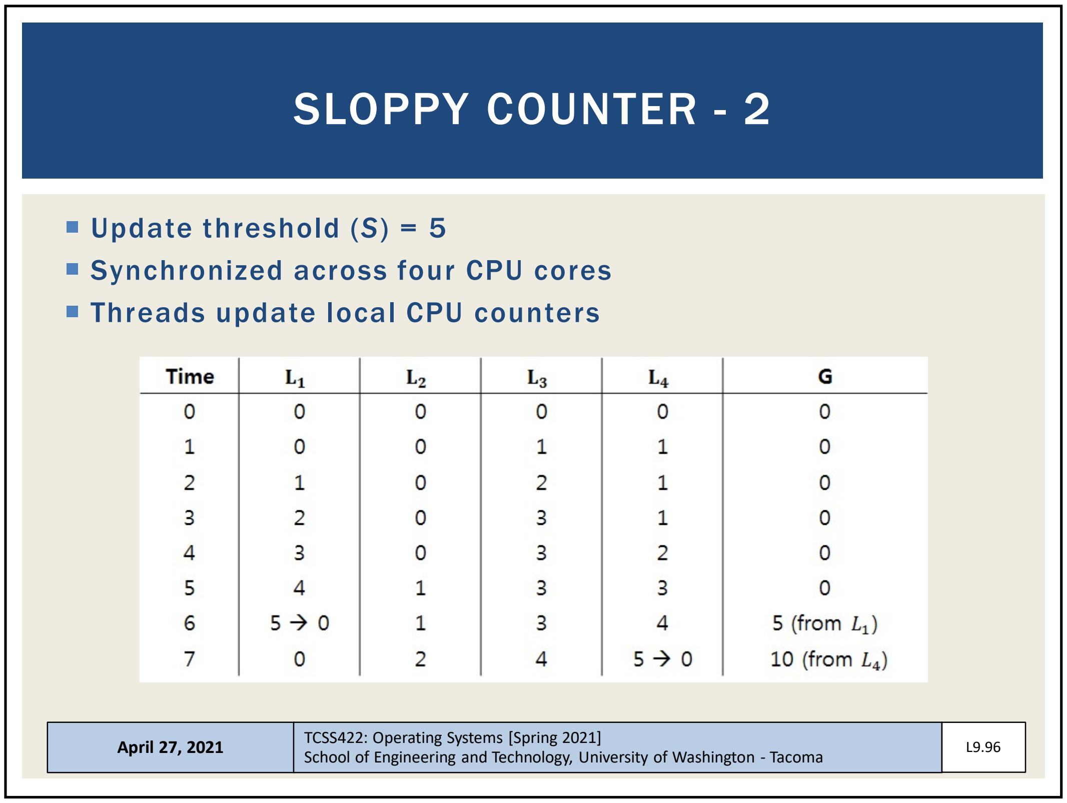
April 27, 2021 | TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma | L9.94



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)

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.95



SLOPPY COUNTER - 2

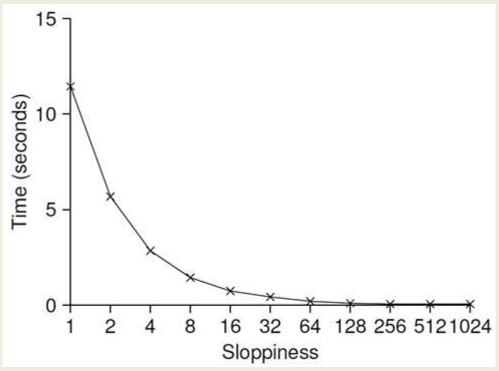
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

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)

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.96

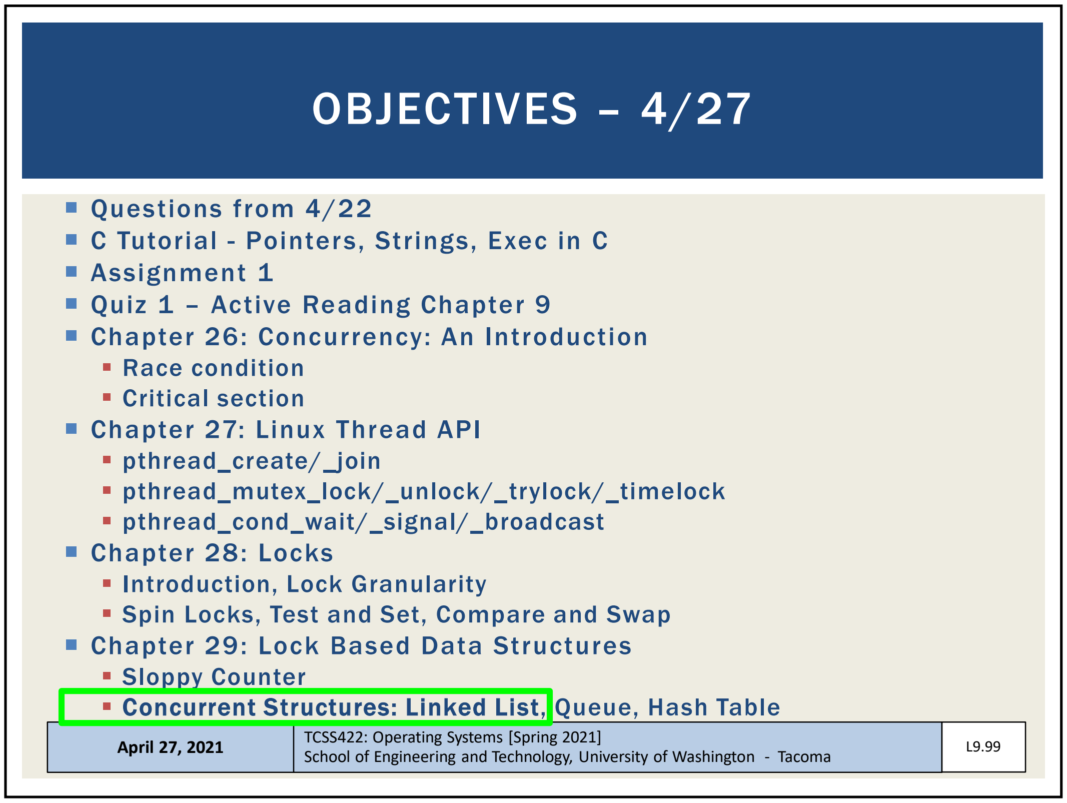
THRESHOLD VALUE *S*

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



SLOPPY COUNTER - EXAMPLE

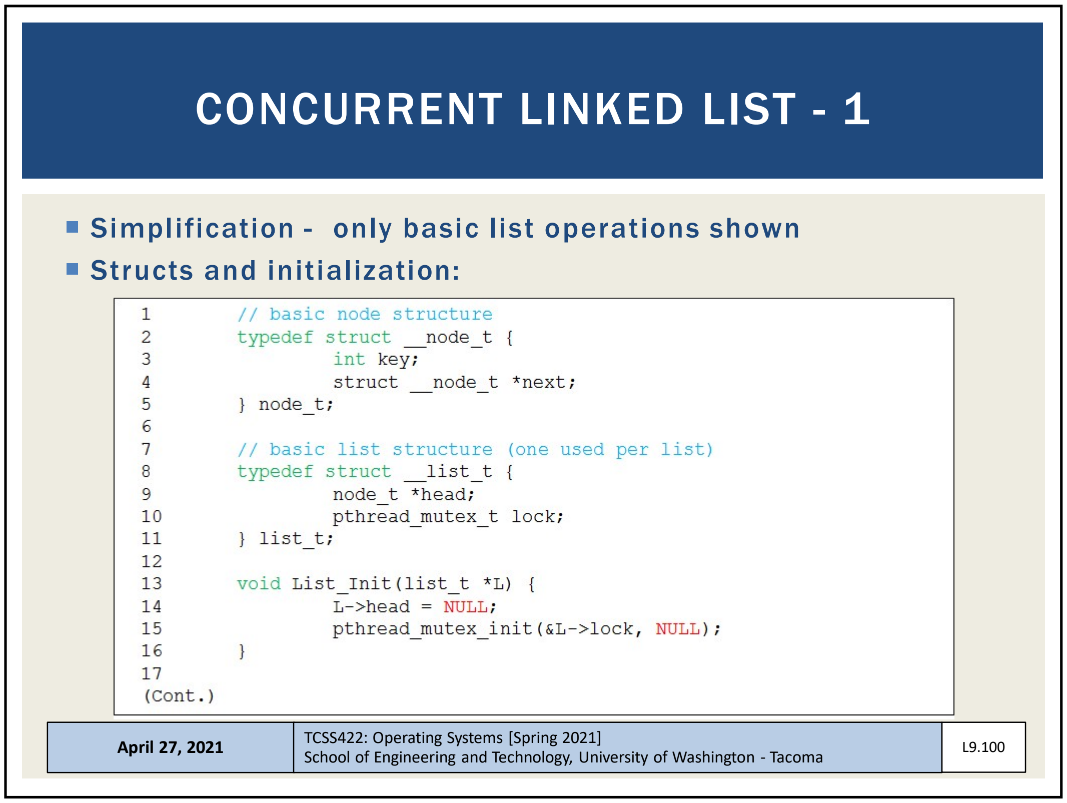
- Example implementation
- Also with CPU affinity



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma | L9.99

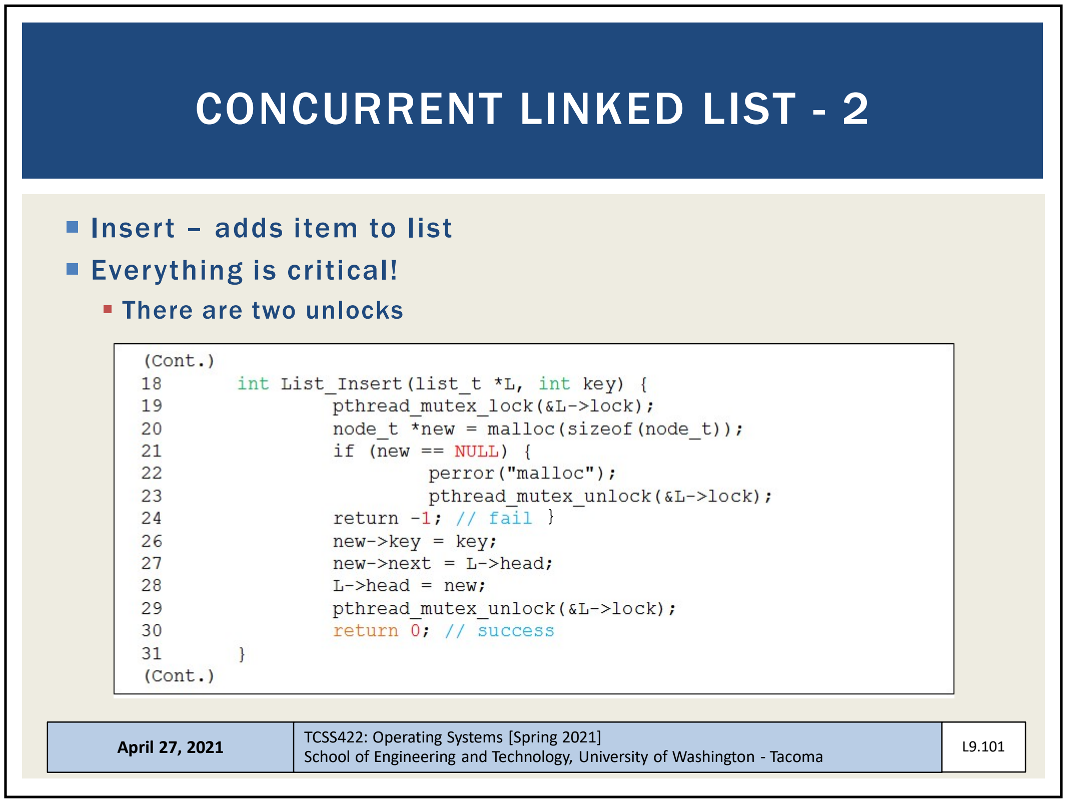


CONCURRENT LINKED LIST - 1

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

```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

(Cont.)
```

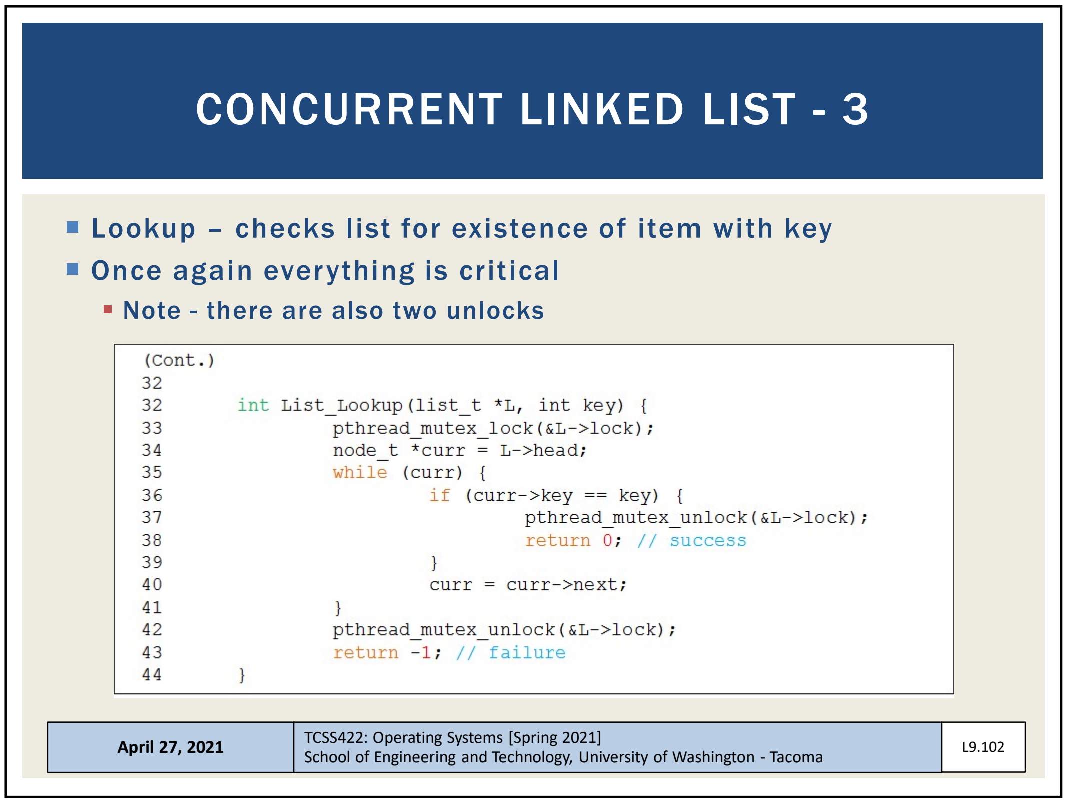
April 27, 2021 | TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma | L9.100



CONCURRENT LINKED LIST - 2

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

```
(Cont.)
        int List_Insert(list_t *L, int key) {
                pthread_mutex_lock(&L->lock);
                node_t *new = malloc(sizeof(node_t));
                if (new == NULL) {
                        perror("malloc");
                        pthread_mutex_unlock(&L->lock);
                return -1; // fail }
                new->key = key;
                new->next = L->head;
                L->head = new;
                pthread_mutex_unlock(&L->lock);
                return 0; // success
        }
(Cont.)
```



CONCURRENT LINKED LIST - 3

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

```
(Cont.)

        int List_Lookup(list_t *L, int key) {
                pthread_mutex_lock(&L->lock);
                node_t *curr = L->head;
                while (curr) {
                        if (curr->key == key) {
                                pthread_mutex_unlock(&L->lock);
                                return 0; // success
                        }
                        curr = curr->next;
                }
                pthread_mutex_unlock(&L->lock);
                return -1; // failure
        }
```

CONCURRENT LINKED LIST

- First Implementation:
  - Lock everything inside Insert() and Lookup()
  - If malloc() fails lock must be released
    - Research has shown "***exception-based control flow***" to be error prone
    - 40% of Linux OS bugs occur in rarely taken code paths
    - Unlocking in an exception handler is considered a poor coding practice
    - There is nothing specifically wrong with this example however
- Second Implementation …

CCL – SECOND IMPLEMENTATION

- Init and Insert

```
void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}

```

CCL – SECOND IMPLEMENTATION - 2

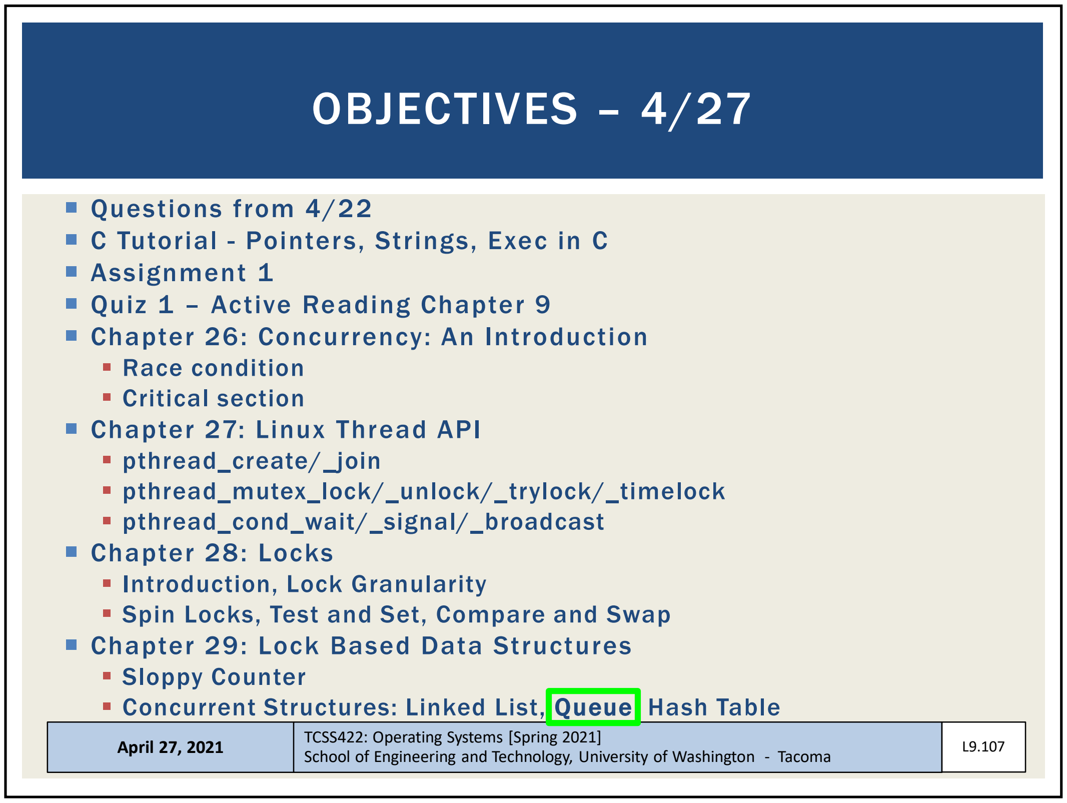
- Lookup

```
(Cont.)
        int List_Lookup(list_t *L, int key) {
                int rv = -1;
                pthread_mutex_lock(&L->lock);
                node_t *curr = L->head;
                while (curr) {
                        if (curr->key == key) {
                                rv = 0;
                                break;
                        }
                        curr = curr->next;
                }
                pthread_mutex_unlock(&L->lock);
                return rv; // now both success and failure
        }
```

CONCURRENT LINKED LIST PERFORMANCE

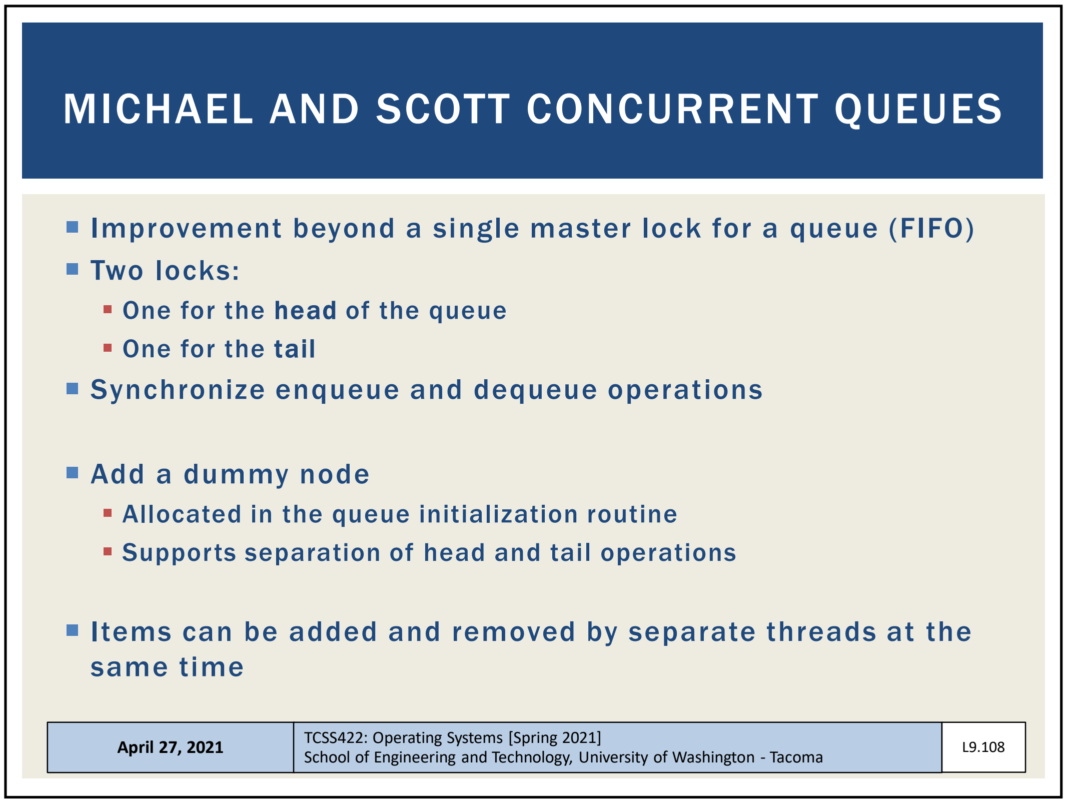
- Using a single lock for entire list is not very performant
- Users must “wait” in line for a single lock to access/modify any item
- Hand-over-hand-locking (lock coupling)
  - Introduce a lock for each node of a list
  - Traversal involves handing over previous node’s lock, acquiring the next node’s lock…
  - Improves lock granularity
  - Degrades traversal performance

- Consider hybrid approach
  - Fewer locks, but more than 1
  - Best lock-to-node distribution?



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

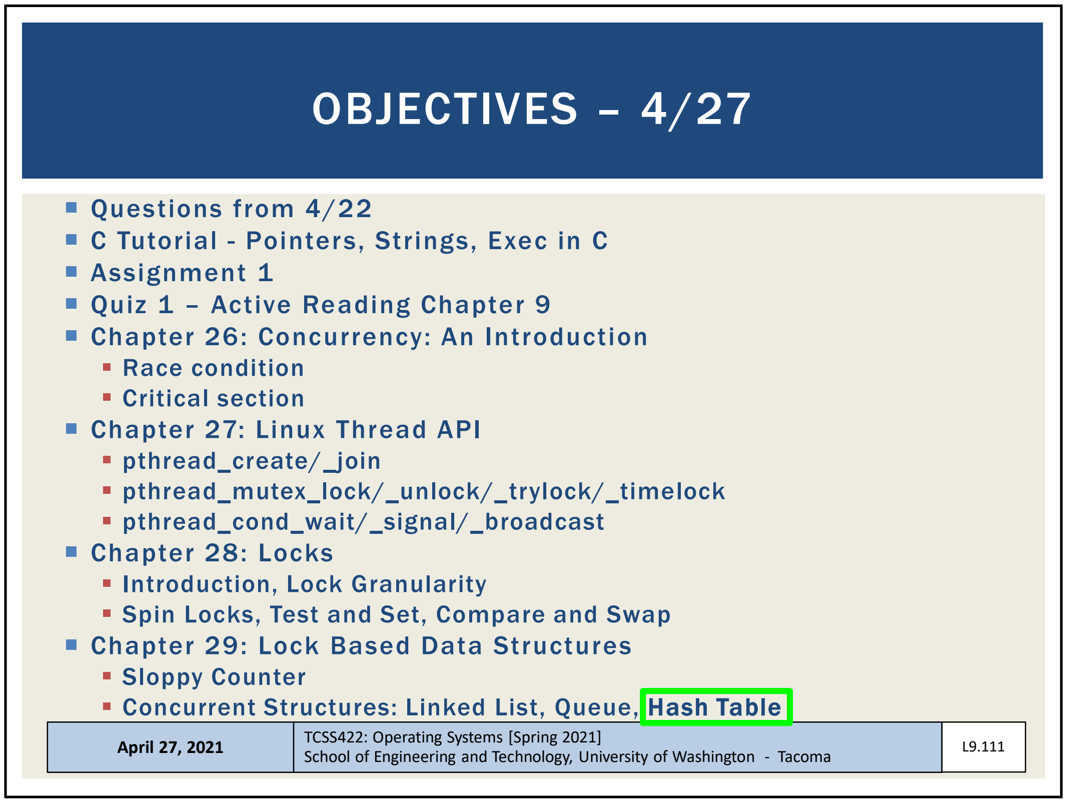
- Remove from queue

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```

CONCURRENT QUEUE - 2

- Add to queue

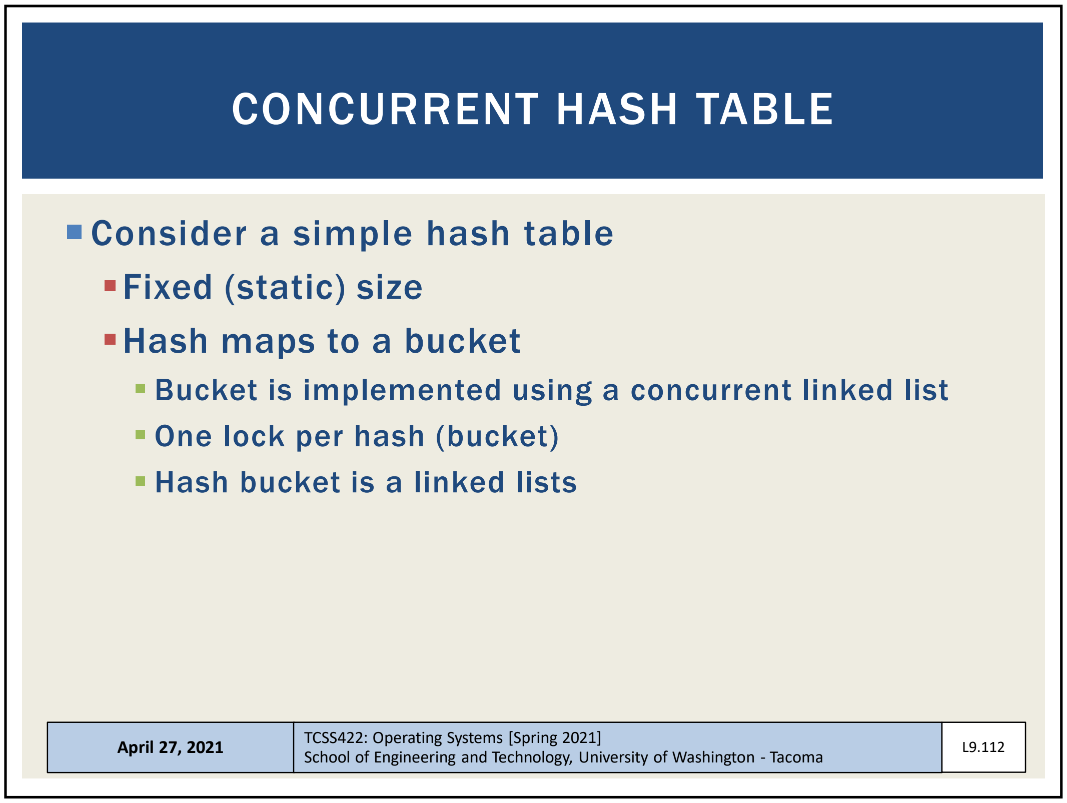
```
(Cont.)
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
}
(Cont.)
```



OBJECTIVES – 4/27

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Quiz 1 – Active Reading Chapter 9
- 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

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.111



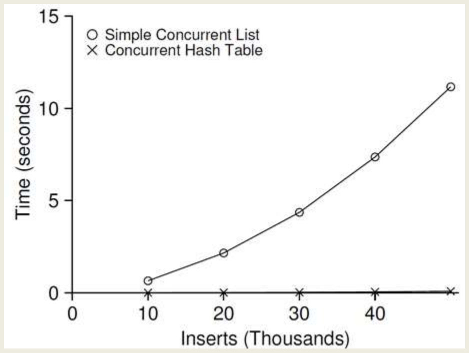
CONCURRENT HASH TABLE

- Consider a simple hash table
  - Fixed (static) size
  - Hash maps to a bucket
    - Bucket is implemented using a concurrent linked list
    - One lock per hash (bucket)
    - Hash bucket is a linked lists

April 27, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L9.112

INSERT PERFORMANCE – CONCURRENT HASH TABLE

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



The simple concurrent hash table scales magnificently.

CONCURRENT HASH TABLE

```
#define BUCKETS (101)

typedef struct __hash_t {
        list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
        int i;
        for (i = 0; i < BUCKETS; i++) {
                List_Init(&H->lists[i]);
        }
}

int Hash_Insert(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Lookup(&H->lists[bucket], key);
}
```

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

Locks are encapsulated within data structure code ensuring thread safety.

Lock granularity tradeoff already optimized inside data structurew

Multiple threads can more easily share data

All of the above

None of the above

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

LOCK-FREE DATA STRUCTURES

- Lock-free data structures in Java
- Java.util.concurrent.atomic package
- Classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicIntegerFieldUpdater
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
  - AtomicReference
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

QUESTIONS