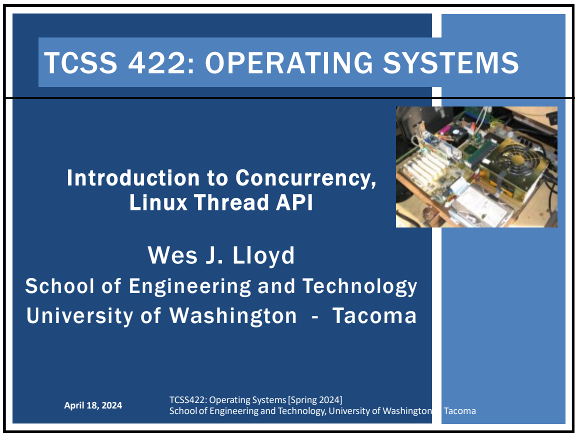


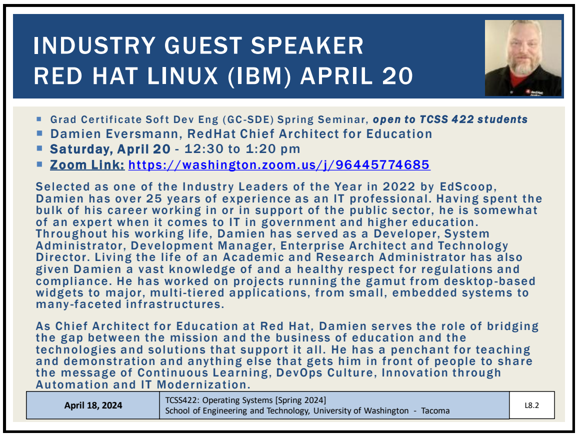
TCSS 422: OPERATING SYSTEMS

Introduction to Concurrency, Linux Thread API

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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma

1



INDUSTRY GUEST SPEAKER RED HAT LINUX (IBM) APRIL 20

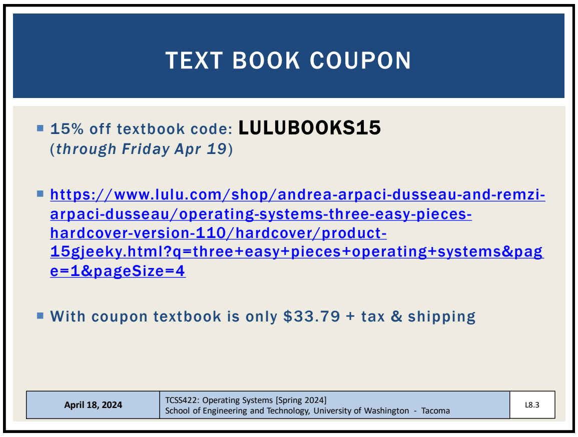
- Grad Certificate Soft Dev Eng (GC-SDE) Spring Seminar, ***open to TCSS 422 students***
- Damien Eversmann, RedHat Chief Architect for Education
- **Saturday, April 20** - 12:30 to 1:20 pm
- **Zoom Link:** https://washington.zoom.us/j/96445774685

Selected as one of the Industry Leaders of the Year in 2022 by EdScoop, Damien has over 25 years of experience as an IT professional. Having spent the bulk of his career working in or in support of the public sector, he is somewhat of an expert when it comes to IT in government and higher education. Throughout his working life, Damien has served as a Developer, System Administrator, Development Manager, Enterprise Architect and Technology Director. Living the life of an Academic and Research Administrator has also given Damien a vast knowledge of and a healthy respect for regulations and compliance. He has worked on projects running the gamut from desktop-based widgets to major, multi-tiered applications, from small, embedded systems to many-faceted infrastructures.

As Chief Architect for Education at Red Hat, Damien serves the role of bridging the gap between the mission and the business of education and the technologies and solutions that support it all. He has a penchant for teaching and demonstration and anything else that gets him in front of people to share the message of Continuous Learning, DevOps Culture, Innovation through Automation and IT Modernization.

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.2

2

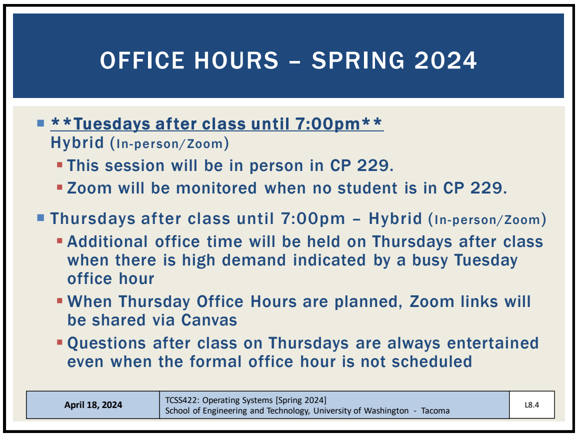


TEXT BOOK COUPON

- 15% off textbook code: **LULUBOOKS15** *(through Friday Apr 19)*
- https://www.lulu.com/shop/andrea-arpaci-dusseau-and-remzi-arpaci-dusseau/operating-systems-three-easy-pieces-hardcover-version-110/hardcover/product-15gjeeky.html?q=three+easy+pieces+operating+systems&page=1&pageSize=4
- With coupon textbook is only $33.79 + tax & shipping

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.3

3

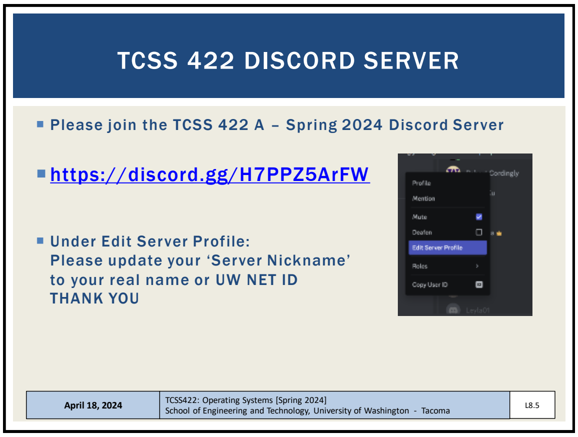


OFFICE HOURS – SPRING 2024

- ****Tuesdays after class until 7:00pm**** Hybrid (In-person/Zoom)
  - This session will be in person in CP 229.
  - Zoom will be monitored when no student is in CP 229.
- Thursdays after class until 7:00pm – Hybrid (In-person/Zoom)
  - Additional office time will be held on Thursdays after class when there is high demand indicated by a busy Tuesday office hour
  - When Thursday Office Hours are planned, Zoom links will be shared via Canvas
  - Questions after class on Thursdays are always entertained even when the formal office hour is not scheduled

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.4

4

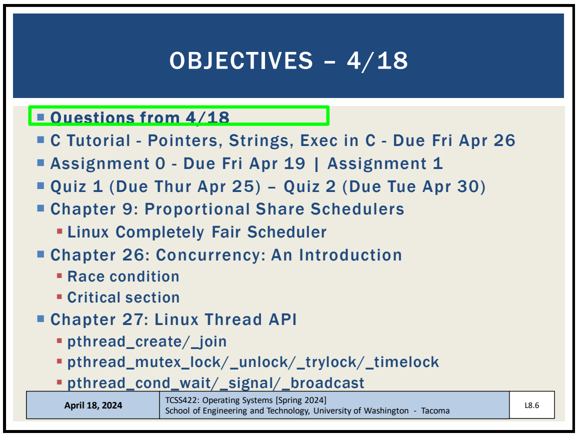


TCSS 422 DISCORD SERVER

- Please join the TCSS 422 A – Spring 2024 Discord Server
- https://discord.gg/H7PPZ5ArFW
- Under Edit Server Profile:
  Please update your 'Server Nickname' to your real name or UW NET ID
  THANK YOU

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.5

5



OBJECTIVES – 4/18

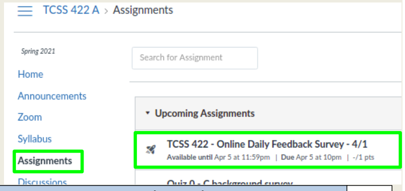
- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.6

6

ONLINE DAILY FEEDBACK SURVEY

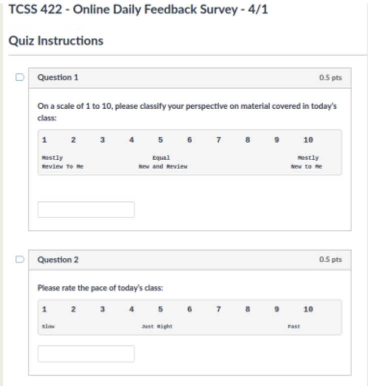
- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p



TCSS 422 - 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

7



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
```

8

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (28 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.93 (↑ - previous 6.81)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.21 (↑ - previous 5.42)**

9

FEEDBACK FROM 4/16

- ***How are the tickets and strides represented and used in the kernel?***

- Linux does not use the lottery or stride scheduler
- Linux uses the Completely Fair Scheduler (CFS)
- CFS tracks vruntime for each job, which capture a job's runtime
- CFS attempts to balance accumulative vruntime between jobs
- CFS does not use tickets or stride values

10

FEEDBACK - 2

- ***I'm not fully clear on the difference between normal and real-time CFS scheduling classes. What does real-time mean in this context?***

- "Real time" refers to a class of high priority processes which must response with minimal delay (latency)
- These "real-time" processes are special time-critical applications that need precise control over the way in which runnable threads are selected for execution.
- In general, these may be system processes which must respond to I/O or other critical operations
- These manual pages provide additional useful information:

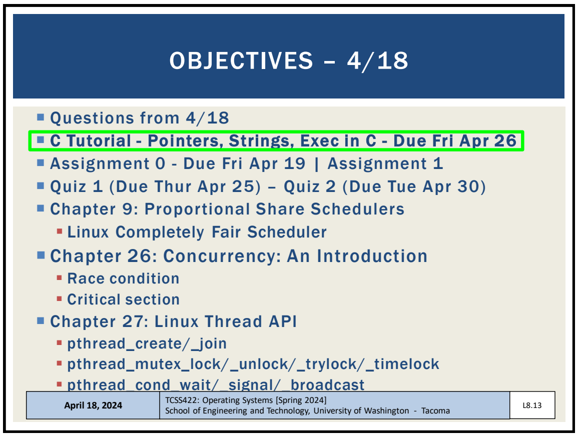
```
man sched_setscheduler
man 7 sched
```

11

FEEDBACK - 3

- ***Still confused with some of the schedule, only comfortable with Red Robin***

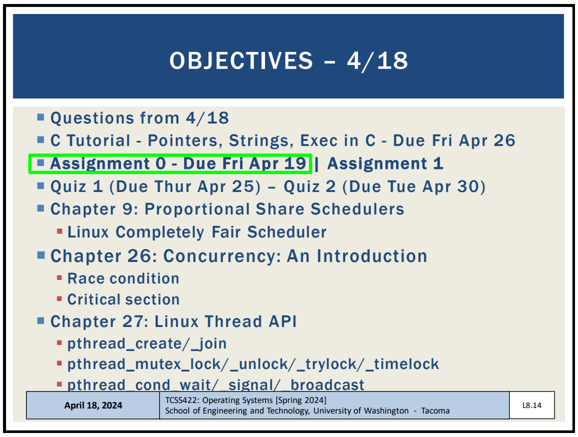
12



OBJECTIVES – 4/18

- Questions from 4/18
- **C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26**
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

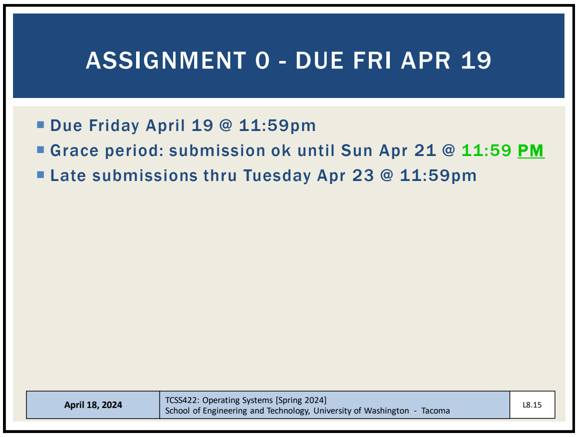
13



OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- **Assignment 0 - Due Fri Apr 19 | Assignment 1**
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

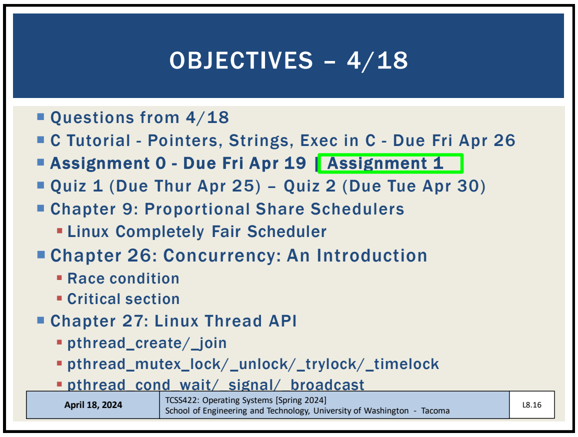
14



ASSIGNMENT 0 - DUE FRI APR 19

- Due Friday April 19 @ 11:59pm
- Grace period: submission ok until Sun Apr 21 @ 11:59 PM
- Late submissions thru Tuesday Apr 23 @ 11:59pm

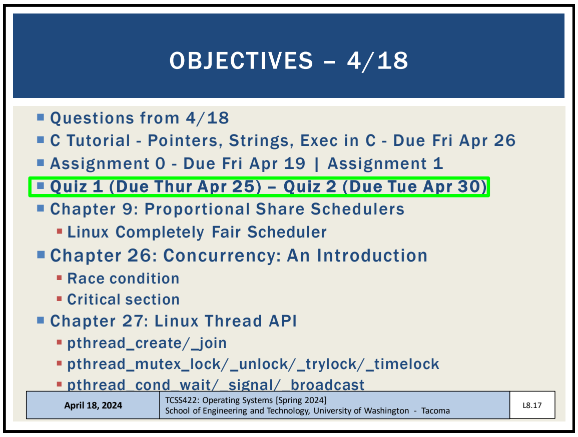
15



OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- **Assignment 0 - Due Fri Apr 19 | Assignment 1**
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

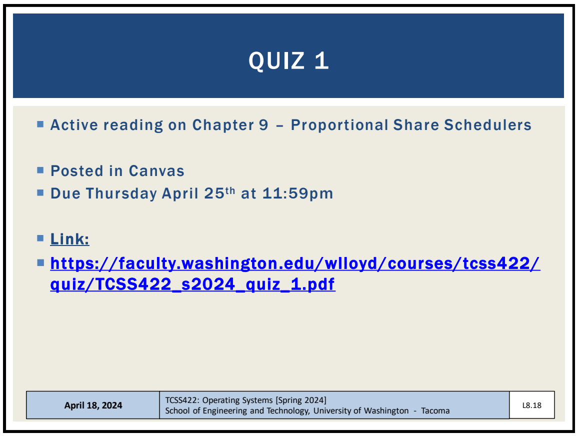
16



OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- **Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)**
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

17



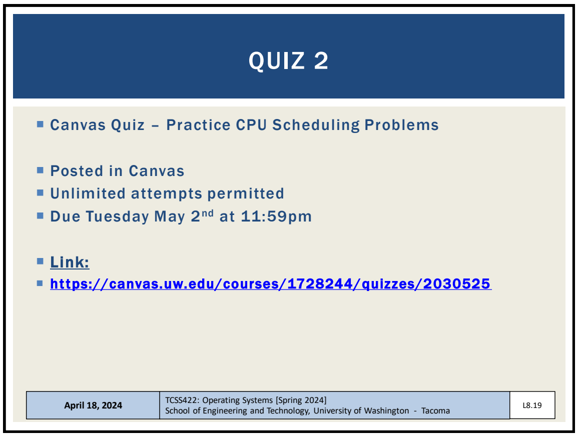
QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers

- Posted in Canvas
- Due Thursday April 25th at 11:59pm

- **Link:**
- **https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_s2024_quiz_1.pdf**

18



QUIZ 2

- Canvas Quiz – Practice CPU Scheduling Problems
- Posted in Canvas
- Unlimited attempts permitted
- Due Tuesday May 2nd at 11:59pm
- Link:
- https://canvas.uw.edu/courses/1728244/quizzes/2030525

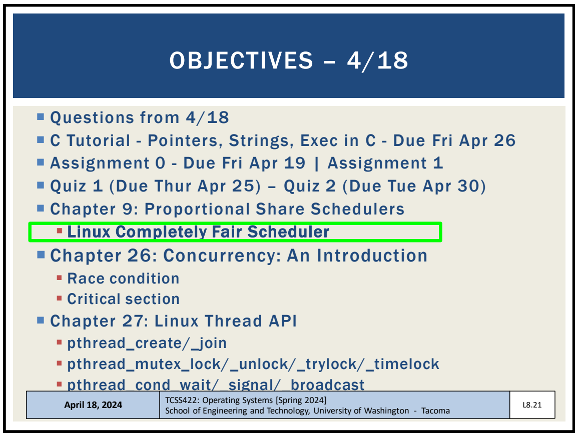
April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.19

19

WE WILL RETURN AT 5:00PM

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.20

20

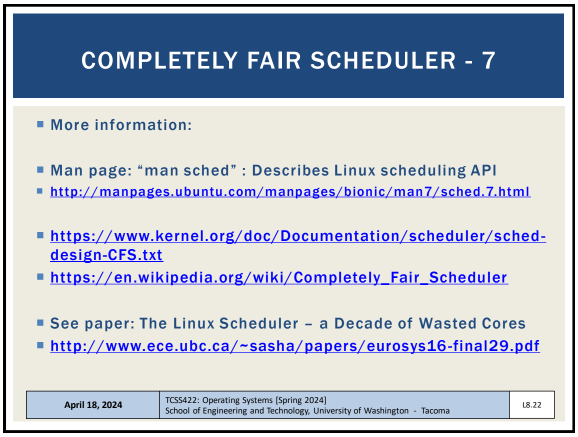


OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.21

21

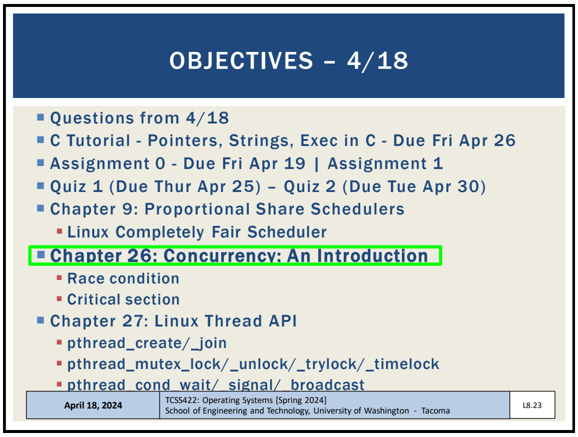


COMPLETELY FAIR SCHEDULER - 7

- More information:
- Man page: “man sched” : Describes Linux scheduling API
- http://manpages.ubuntu.com/manpages/bionic/man7/sched.7.html
- https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
- https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- See paper: The Linux Scheduler – a Decade of Wasted Cores
- http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.22

22

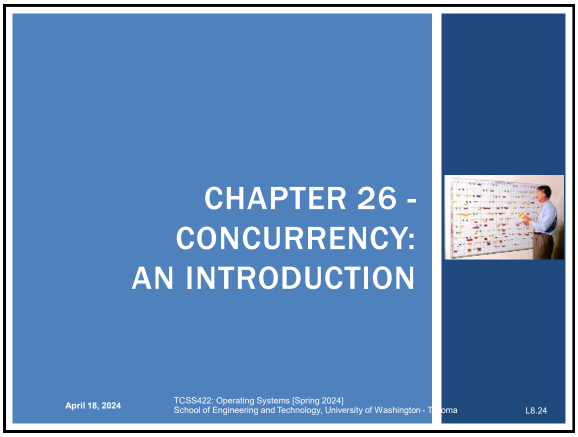


OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.23

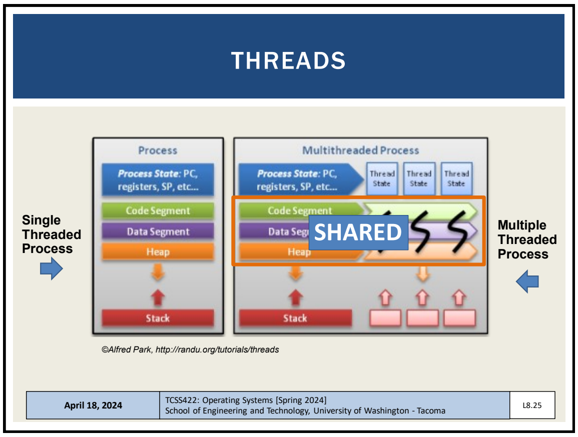
23



CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.24

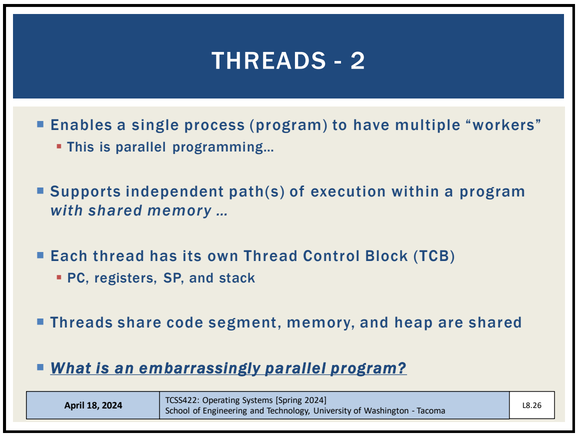
24



THREADS

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

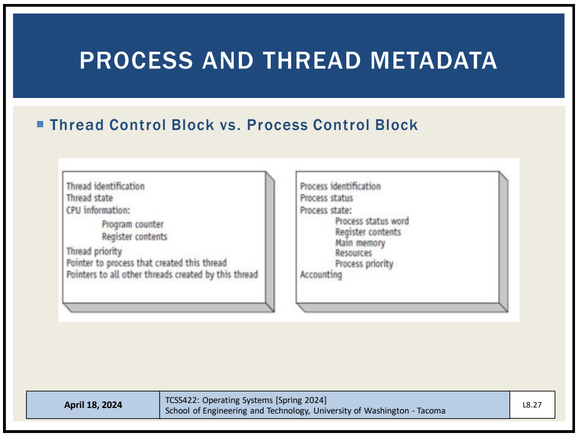
25



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?***

26



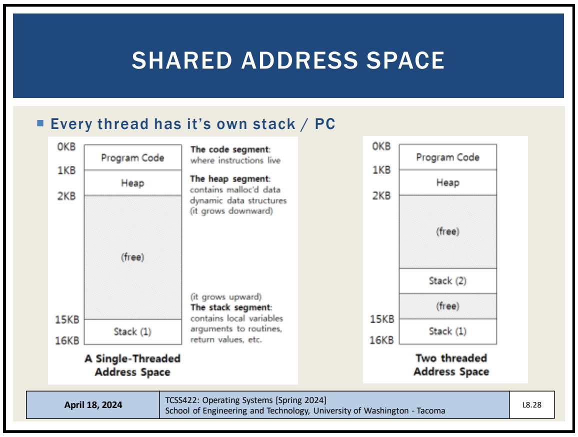
PROCESS AND THREAD METADATA

- Thread Control Block vs. Process Control Block

Thread identification
Thread state
CPU information:
    Program counter
    Register contents
Thread priority
Pointer to process that created this thread
Pointers to all other threads created by this thread

Process identification
Process status
Process state:
    Process status word
    Register contents
    Main memory
    Resources
    Process priority
Accounting

27



SHARED ADDRESS SPACE

- Every thread has it's own stack / PC

The code segment: where instructions live

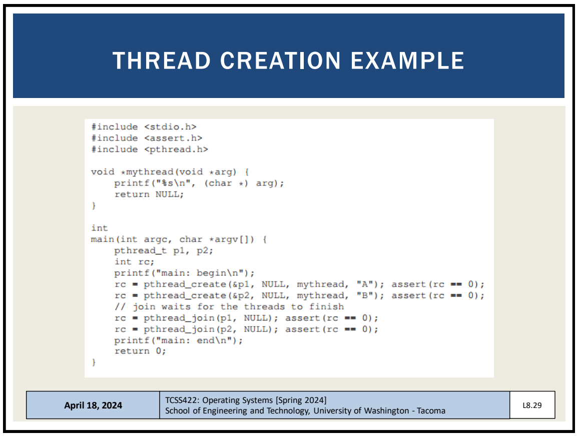
The heap segment: contains malloc'd data dynamic data structures (it grows downward)

(it grows upward) **The stack segment**: contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space

Two threaded Address Space

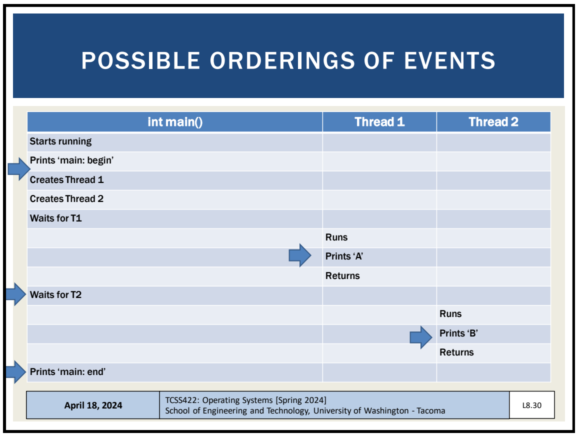
28



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;
}
```

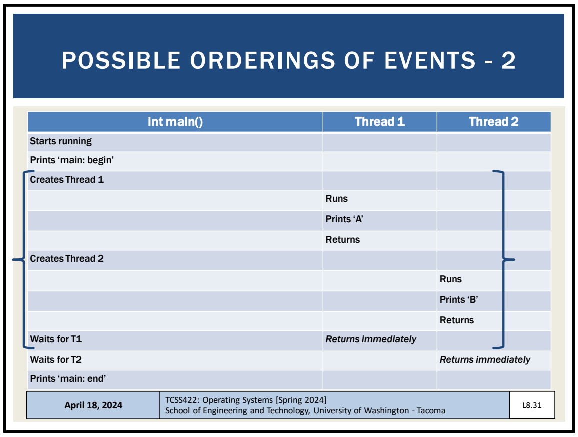
29



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'		

30

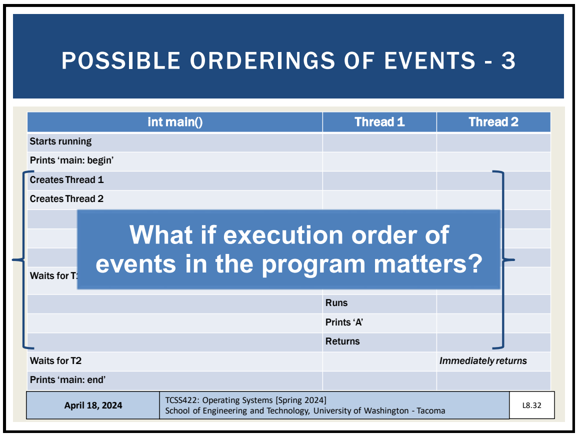


POSSIBLE ORDERINGS OF EVENTS - 2

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'		

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.31

31



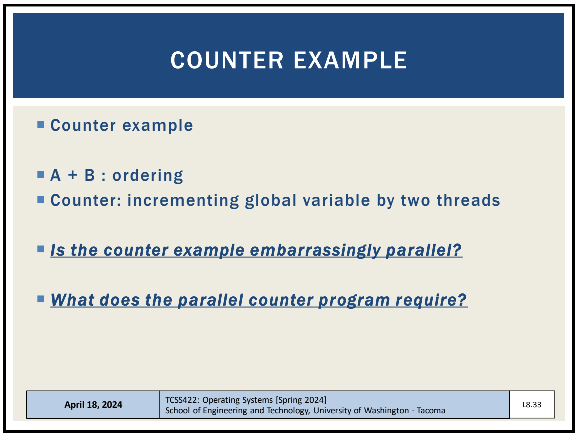
POSSIBLE ORDERINGS OF EVENTS - 3

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?

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.32

32

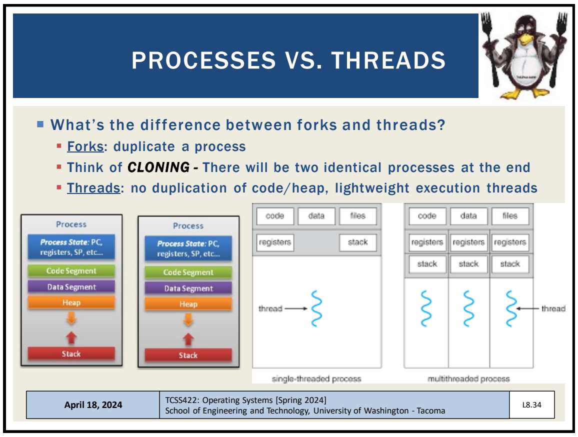


COUNTER EXAMPLE

- 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?***

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.33

33

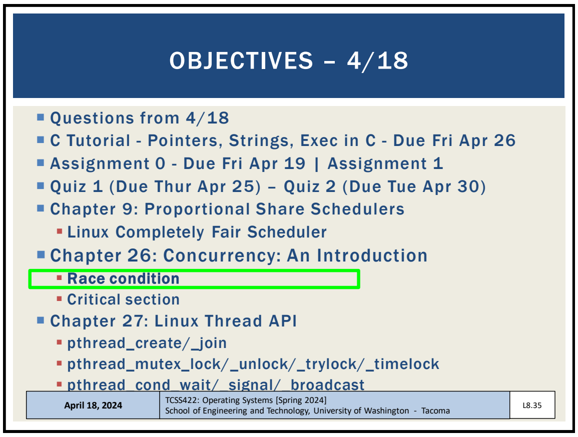


PROCESSES VS. THREADS

- 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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.34

34

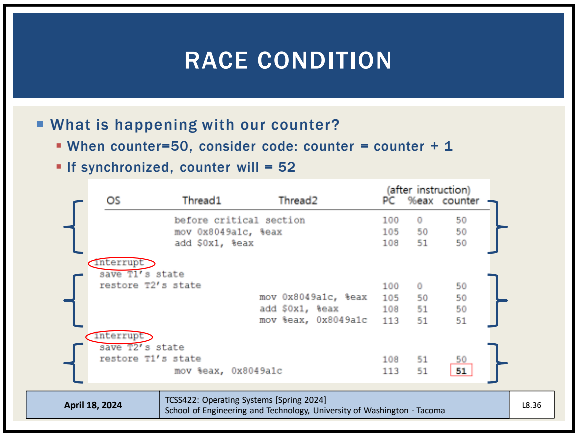


OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.35

35



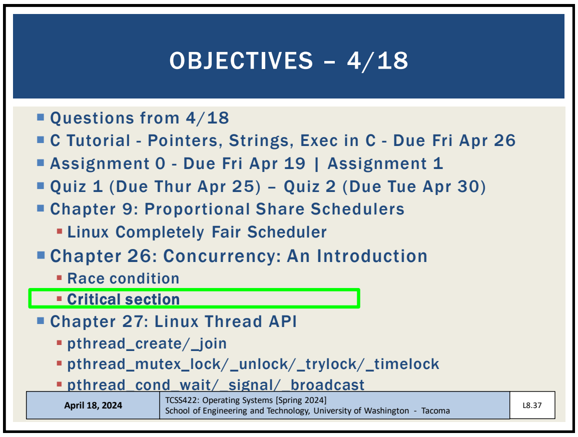
RACE CONDITION

- 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
```

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.36

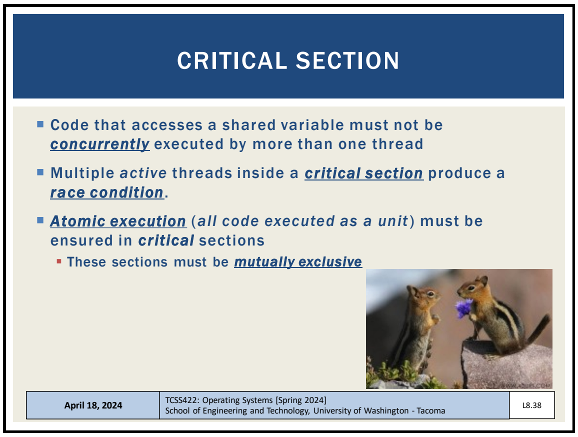
36



OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

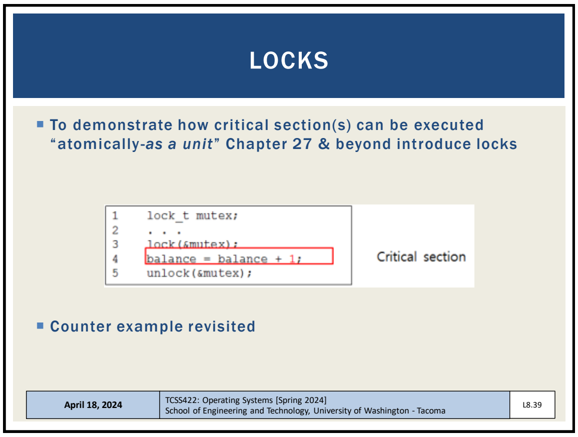
37



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***

38



LOCKS

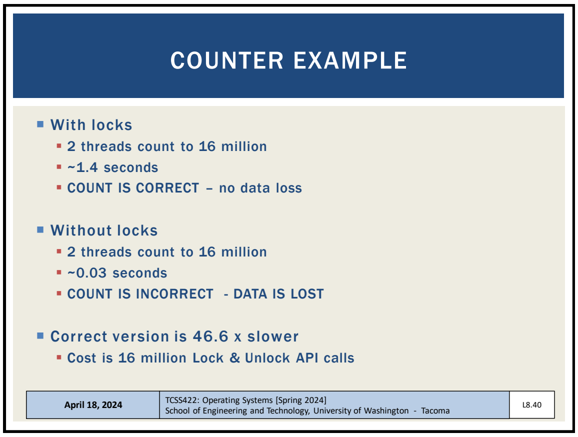
- To demonstrate how critical section(s) can be executed "atomically-*as a unit*" Chapter 27 & beyond introduce locks

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

Critical section

- Counter example revisited

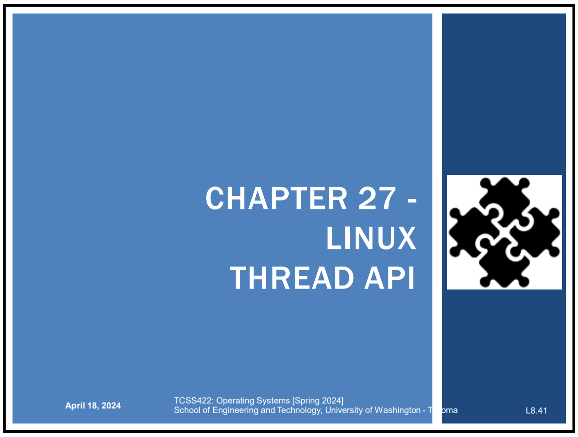
39



COUNTER EXAMPLE

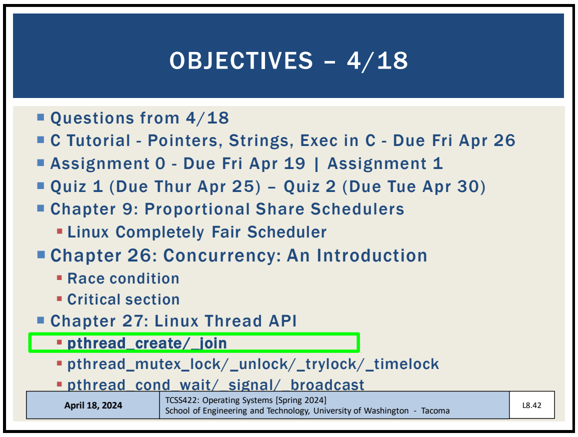
- With locks
  - 2 threads count to 16 million
  - ~1.4 seconds
  - COUNT IS CORRECT – no data loss
- Without locks
  - 2 threads count to 16 million
  - ~0.03 seconds
  - COUNT IS INCORRECT - DATA IS LOST
- Correct version is 46.6 x slower
  - Cost is 16 million Lock & Unlock API calls

40



CHAPTER 27 - LINUX THREAD API

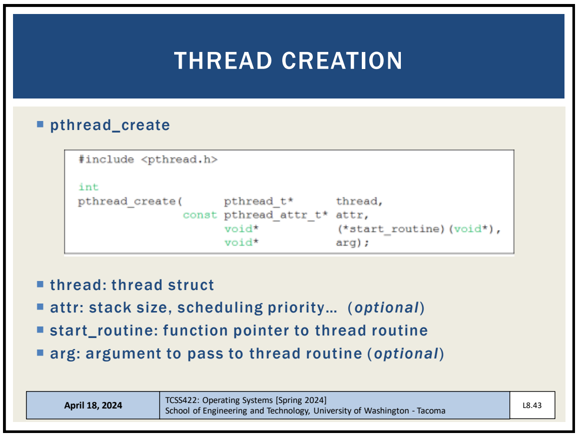
41



OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

42



THREAD CREATION

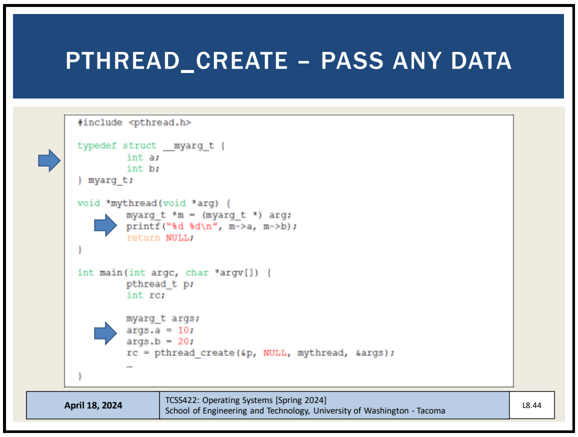
- pthread_create

```
#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*)

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.43

43

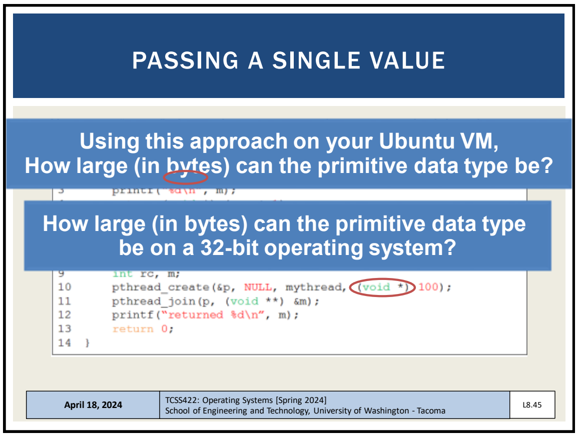


PTHREAD_CREATE – PASS ANY DATA

```
#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);
        …
}
```

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.44

44



PASSING A SINGLE VALUE

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

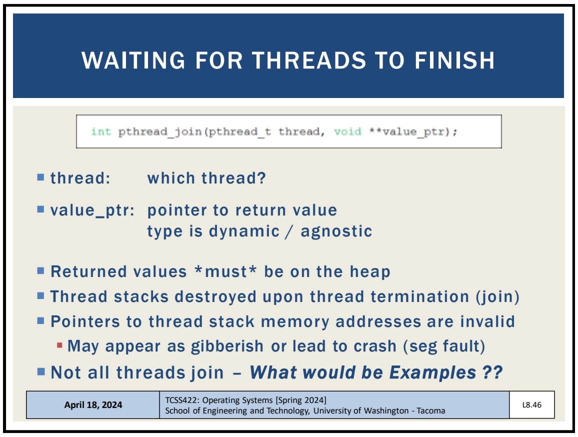
```
        printf("%d\n", m);
```

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 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.45

45



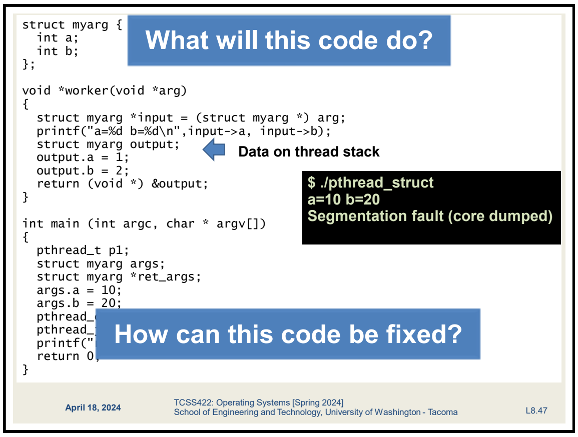
WAITING FOR THREADS TO FINISH

```
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 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.46

46



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
}
```

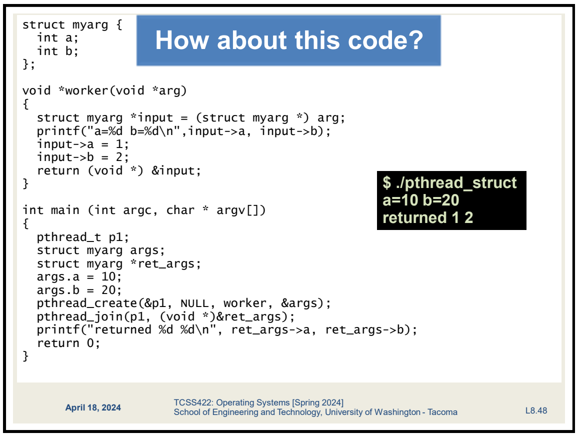
Data on thread stack

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

How can this code be fixed?

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.47

47



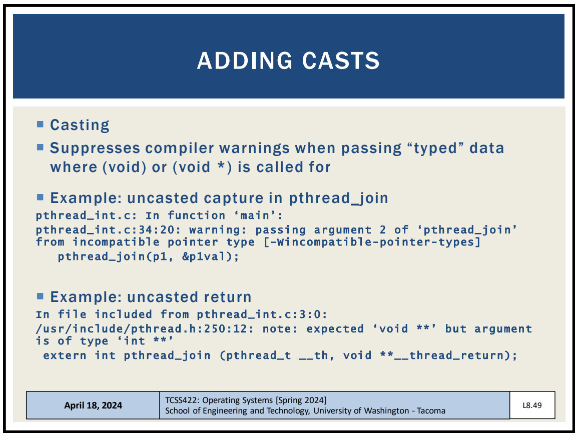
How about this code?

```
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
```

April 18, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L8.48

48



ADDING CASTS

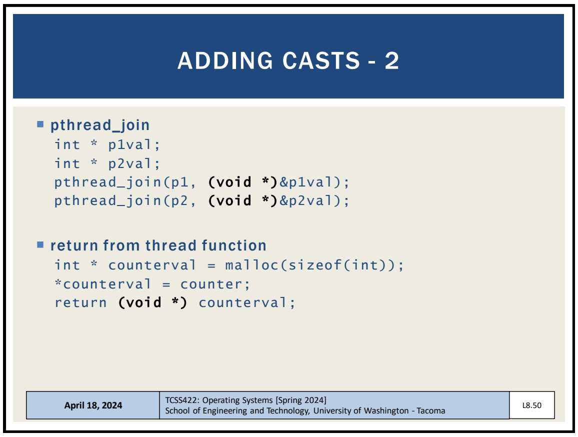
- Casting
- Suppresses compiler warnings when passing “typed” data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join

```
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);
```

49



ADDING CASTS - 2

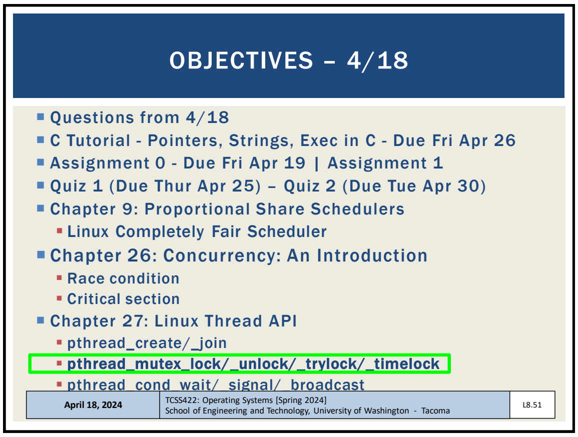
- pthread_join

```
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;
```

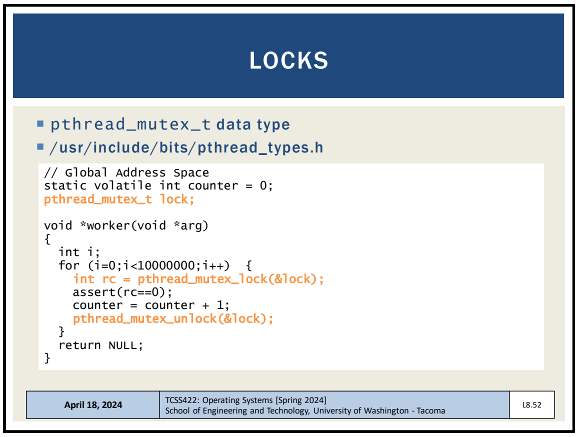
50



OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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

51

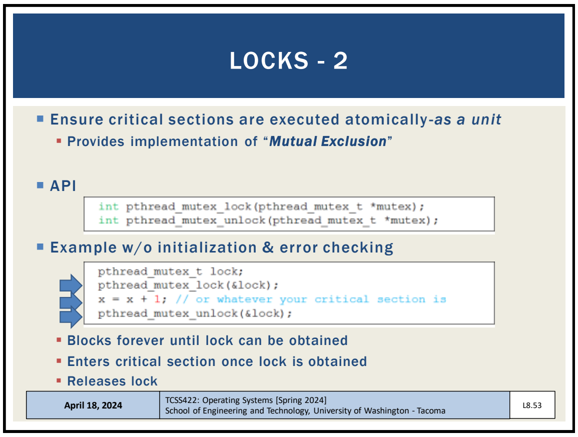


LOCKS

- 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;
}
```

52



LOCKS - 2

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

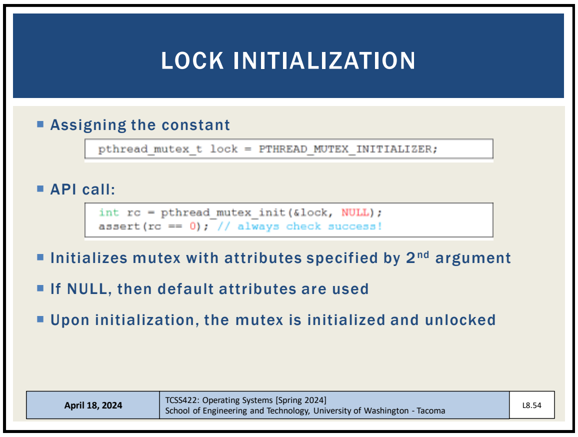
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- 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

53



LOCK INITIALIZATION

- Assigning the constant

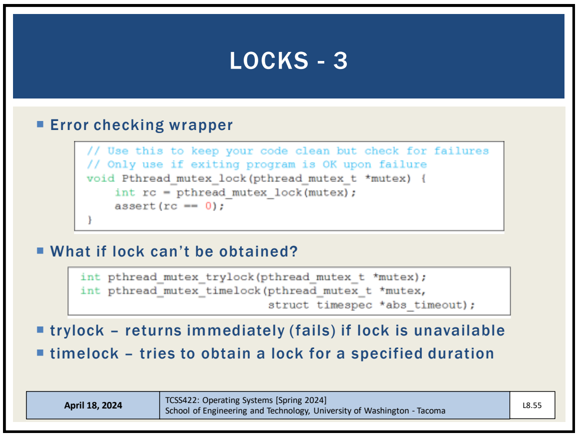
```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- 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

54



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);
}
```

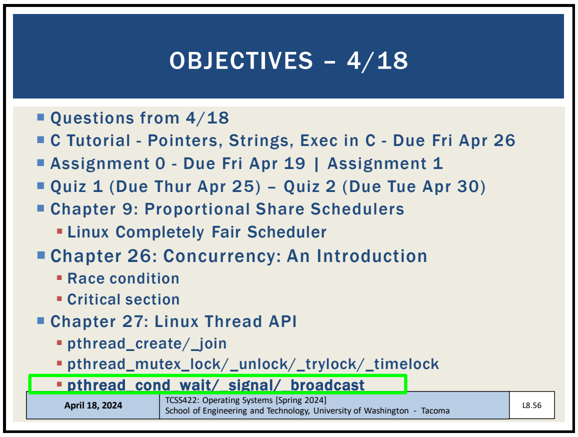
- What if lock can't be obtained?

```
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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.55

55

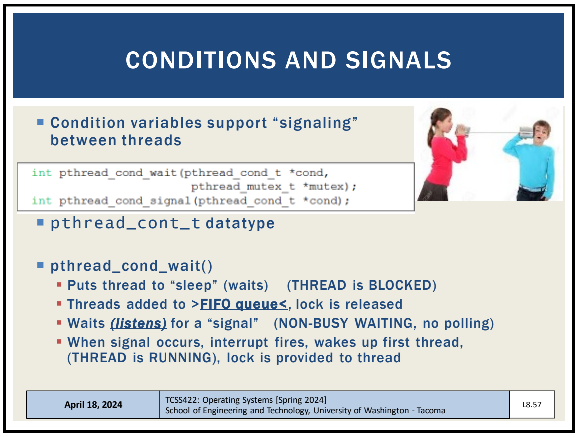


OBJECTIVES – 4/18

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 26
- Assignment 0 - Due Fri Apr 19 | Assignment 1
- Quiz 1 (Due Thur Apr 25) – Quiz 2 (Due Tue Apr 30)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- 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**

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.56

56



CONDITIONS AND SIGNALS

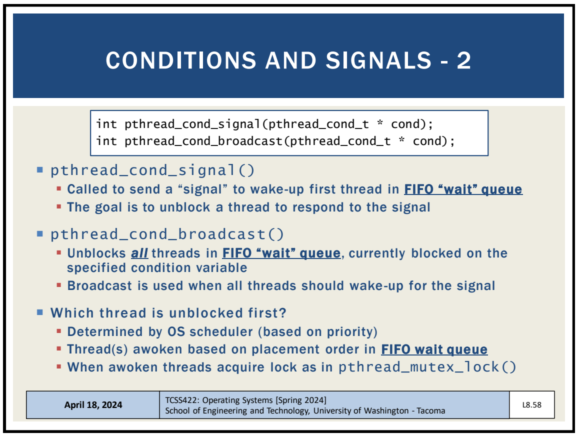
- Condition variables support "signaling" between threads

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

- pthread_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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.57

57



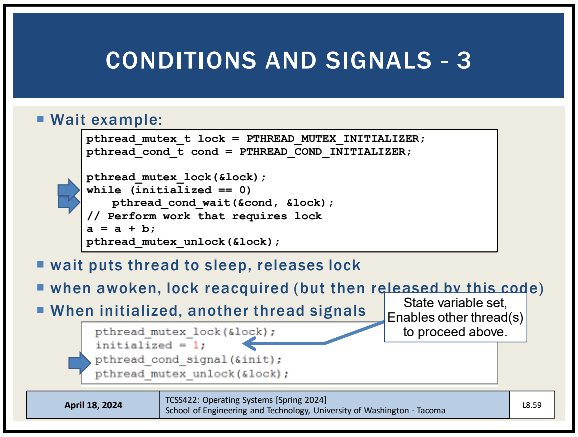
CONDITIONS AND SIGNALS - 2

```
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()

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.58

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);
```

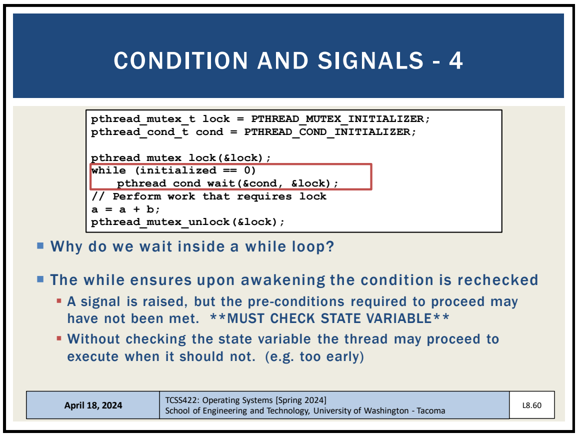
- wait puts thread to sleep, releases lock
- when awoken, lock reacquired (but then released by this code)
- When initialized, another thread signals

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

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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.59

59



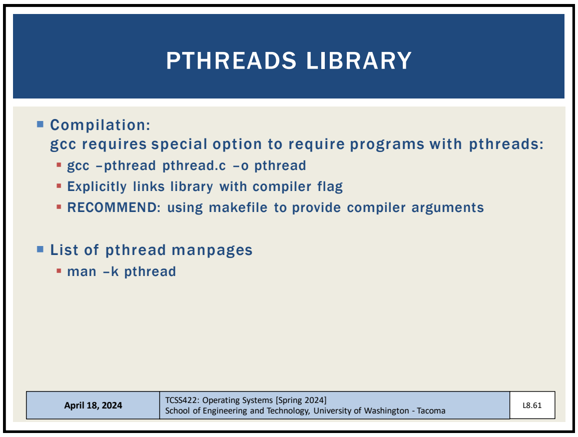
CONDITION AND SIGNALS - 4

```
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)

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.60

60

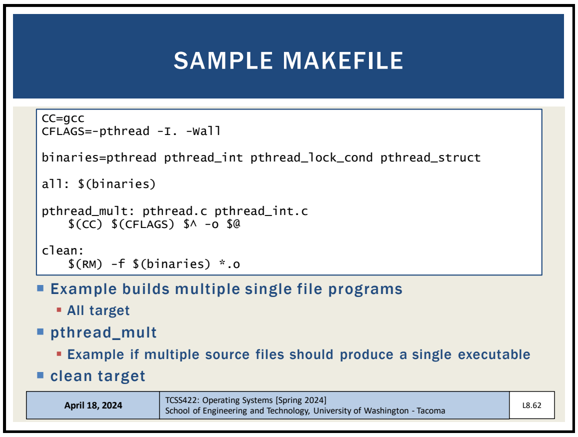


PTHREADS LIBRARY

- 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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.61

61



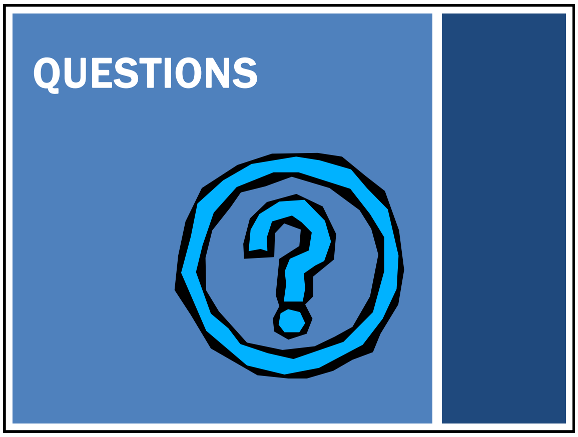
SAMPLE MAKEFILE

```
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

April 18, 2024 | TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma | L8.62

62



QUESTIONS

63