


# 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

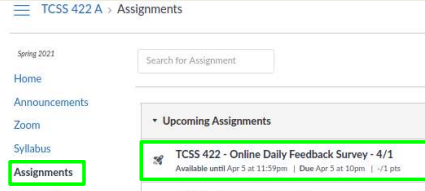
# OBJECTIVES – 4/27

- **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

# 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



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

# TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1    0.5 pts

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

1   2   3   4   5   6   7   8   9   10

Mostly Review To Me      Equal New and Review      Mostly New To Me

Question 2    0.5 pts

Please rate the pace of today's class:

1   2   3   4   5   6   7   8   9   10

slow      just right      fast

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

# 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
- **Average – 6.90 (↑ - previous 6.67)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.52 (↓ - previous 5.53)**

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

# 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

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

### 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	TCCS422: 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	TCCS422: 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

April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.9
----------------	---	------

### 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
```

April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.10
----------------	---	-------

### 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

April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.11
----------------	---	-------

### 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

April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.12
----------------	---	-------

### 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	TCCS422: 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	TCCS422: 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

April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.15
----------------	---	-------

### QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Friday April 30<sup>th</sup> at 11:59pm
- Grace period til Sunday May 2<sup>nd</sup> at 11:59 \*\* AM \*\*
- Late submissions til Tuesday May 4<sup>th</sup> at 11:59pm
- **Link:**
- [http://faculty.washington.edu/wlloyd/courses/tcss422/TCCS422\\_s2021\\_quiz\\_1.pdf](http://faculty.washington.edu/wlloyd/courses/tcss422/TCCS422_s2021_quiz_1.pdf)


April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.16
----------------	---	-------

### 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	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.17
----------------	---	-------

## CHAPTER 26 - CONCURRENCY: AN INTRODUCTION



April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.18
----------------	---	-------

## THREADS

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

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.19

## 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?**

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.20

## PROCESS AND THREAD METADATA

Thread Control Block vs. Process Control Block

Thread Identification	Process Identification
Thread state	Process status
CPU information:	Process state:
Program counter	Process status word
Register contents	Register contents
Thread priority	Main memory
Pointer to process that created this thread	Resources
Pointers to all other threads created by this thread	Process priority
	Accounting

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.21

## SHARED ADDRESS SPACE

Every thread has its own stack / PC

April 27, 2021 TCCS422: 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;
}
    
```

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.23

## 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'			

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.24

### 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 27, 2021    TCS5422: Operating Systems [Spring 2021]    School of Engineering and Technology, University of Washington - Tacoma    L9.25

### POSSIBLE ORDERINGS OF EVENTS - 3

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		
		Immediately returns
Prints 'main: end'		

What if execution order of events in the program matters?

April 27, 2021    TCS5422: Operating Systems [Spring 2021]    School of Engineering and Technology, University of Washington - Tacoma    L9.26

### 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 27, 2021    TCS5422: Operating Systems [Spring 2021]    School of Engineering and Technology, University of Washington - Tacoma    L9.27

### 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

single-threaded process

multithreaded process

April 27, 2021    TCS5422: Operating Systems [Spring 2021]    School of Engineering and Technology, University of Washington - Tacoma    L9.28

### 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    TCS5422: Operating Systems [Spring 2021]    School of Engineering and Technology, University of Washington - Tacoma    L9.29

### RACE CONDITION

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

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		100	0	50
	restore T2's state		105	50	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
	interrupt				
	save T2's state		108	51	50
	restore T1's state		113	51	51
	mov %eax, 0x8049a1c				51

April 27, 2021    TCS5422: Operating Systems [Spring 2021]    School of Engineering and Technology, University of Washington - Tacoma    L9.30


## 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	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.31
----------------	---	-------

## 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**



April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.32
----------------	---	-------

## LOCKS

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

```

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

Critical section

- Counter example revisited


April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.33
----------------	---	-------

## WE WILL RETURN AT 4:53PM



April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.34
----------------	---	-------

## CHAPTER 27 - LINUX THREAD API



April 27, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.35
----------------	---	-------

## 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	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L9.36
----------------	---	-------

## 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 27, 2021 TCCS422: Operating Systems [Spring 2021] L9.37  
 School of Engineering and Technology, University of Washington - Tacoma

## 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 27, 2021 TCCS422: Operating Systems [Spring 2021] L9.38  
 School of Engineering and Technology, University of Washington - Tacoma

## PASSING A SINGLE VALUE

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

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

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

April 27, 2021 TCCS422: Operating Systems [Spring 2021] L9.39  
 School of Engineering and Technology, University of Washington - Tacoma

## 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 TCCS422: Operating Systems [Spring 2021] L9.40  
 School of Engineering and Technology, University of Washington - Tacoma

## 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_create(&p1, NULL, worker, &args);
    pthread_join(p1, (void **)&ret_args);
    printf("returned %d %d\n", ret_args->a, ret_args->b);
    return 0;
}
```

← Data on thread stack

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

## How can this code be fixed?

April 27, 2021 TCCS422: Operating Systems [Spring 2021] L9.41  
 School of Engineering and Technology, University of Washington - Tacoma

## 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 27, 2021 TCCS422: Operating Systems [Spring 2021] L9.42  
 School of Engineering and Technology, University of Washington - Tacoma

### 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 TCCS422: 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 TCCS422: 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
  - Stuppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.45

### 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;
}
```

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.46

### 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

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.47

### 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 2<sup>nd</sup> argument
- If NULL, then default attributes are used
- Upon initialization, the mutex is initialized and unlocked

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.48



## 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 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.49

## 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
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.50

## CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

```

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



- pthread\_cond\_t datatype
- pthread\_cond\_wait()
  - Puts thread to "sleep" (waits) (THREAD is BLOCKED)
  - Threads added to >FIFO queues, 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 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.51

## 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 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.52

## 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 27, 2021
TCCS422: 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
TCCS422: 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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L9.55


## 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 27, 2021
TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L9.56

# CHAPTER 28 – LOCKS




April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L9.57

## 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
TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L9.58

## LOCKS



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

```
balance = balance + 1;
```
- A “critical section”:
 

```
1 lock_t mutex; // some globally-allocated lock 'mutex'
2 ...
3 lock (&mutex);
4 balance = balance + 1;
5 unlock (&mutex);
```

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L9.59

## LOCKS - 2

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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L9.60

### LOCKS - 3

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

April 27, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.61

### 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
L9.62

### LOCKS - 4

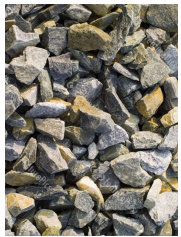
- Program can have many mutex (lock) variables to "serialize" many critical sections
- Locks are also used to protect data structures
  - Prevent multiple threads from changing the same data simultaneously
  - Programmer can make sections of code "granular"
    - **Fine grained** – means just one grain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

April 27, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.63

### 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 %d", str1, str2, str3, &e);
ListNode *node = mylist->head;
int i=0
while (node) {
    node->title = str1;
    node->subheading = str2;
    node->desc = str3;
    node->end = *e;
    node = node->next;
    i++;
}
e = e - i;
pthread_mutex_unlock(&lock);
```



April 27, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.64

### 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 %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

- **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**

What makes a good lock?



April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.67

## 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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.68

## HISTORICAL IMPLEMENTATION

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

```

1 void lock() {
2     DisableInterrupts();
3 }
4 void unlock() {
5     EnableInterrupts();
6 }
```

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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.69


## 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
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.70

## SPIN LOCK IMPLEMENTATION

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



```

1 typedef struct __lock_t { int flag; } lock_t;
2
3 void init(lock_t *mutex) {
4     // 0 → lock is available, 1 → held
5     mutex->flag = 0;
6 }
7
8 void lock(lock_t *mutex) {
9     while (mutex->flag == 1) // TEST the flag
10        ; // spin-wait (do nothing)
11     mutex->flag = 1; // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.71

## 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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.72

### 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...

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.73

### 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
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.74

### TEST-AND-SET INSTRUCTION

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

```

1 int TestAndSet(int *ptr, int new) {
2     int old = *ptr; // fetch old value at ptr
3     *ptr = new;    // store 'new' into ptr
4     return old;   // return the old value
5 }
    
```

- lock() method checks that TestAndSet doesn't return 1
- Comparison is in the caller
- Can implement the C version (non-atomic) and have some success on a single-core VM

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.75

### DIY: TEST-AND-SET - 2

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

```

1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available,
7     // 1 that it is held
8     lock->flag = 0;
9 }
10
11 void lock(lock_t *lock) {
12     while (TestAndSet(&lock->flag, 1) == 1)
13         ; // spin-wait
14 }
15
16 void unlock(lock_t *lock) {
17     lock->flag = 0;
18 }
    
```

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.76

### 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
TCCS422: 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
TCCS422: 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
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.79

## COMPARE AND SWAP

- Compare and Swap
 

```

            1 int CompareAndSwap(int *ptr, int expected, int new) {
            2   int actual = *ptr;
            3   if (actual == expected)
            4     *ptr = new;
            5   return actual;
            6 }
```
- Spin lock
 

C implementation 1-core VM:  
 Count is correct, no deadlock
- X86 provides “cmpxchg1” compare-and-exchange instruction
  - cmpxchg8b
  - cmpxchg16b

April 27, 2021
TCCS422: 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 [polliv.com/app](http://polliv.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
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.82

## LL/SC LOCK

```

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

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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.83

## LL/SC LOCK - 2


```

1 void lock(lock_t *lock) {
2   while (1) {
3     while (LoadLinked(&lock->flag) == 1)
4       ; // spin until it's zero
5     if (StoreConditional(&lock->flag, 1) == 1)
6       return; // if set-it-to-1 was a success; all done
7     ; // otherwise: try it all over again
8   }
9 }
10
11 void unlock(lock_t *lock) {
12   lock->flag = 0;
13 }
```

- Two instruction lock

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.84

# CHAPTER 29 – LOCK BASED DATA STRUCTURES



April 27, 2021 TCCS422: 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 TCCS422: 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 TCCS422: 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

```

1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

April 27, 2021 TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L9.88

## CONCURRENT COUNTER

```

1  typedef struct __counter_t {
2      int value;
3      pthread_lock_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     pthread_mutex_lock(&c->lock);
13     c->value++;
14     pthread_mutex_unlock(&c->lock);
15 }
16
```

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

April 27, 2021 TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L9.89

## CONCURRENT COUNTER - 2

- Decrease counter
- Get value

```

(Cont.)
17 void decrement(counter_t *c) {
18     pthread_mutex_lock(&c->lock);
19     c->value--;
20     pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

April 27, 2021 TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L9.90

### 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.

April 27, 2021
TCS5422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.91

### 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)

April 27, 2021
TCS5422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.92

### 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
TCS5422: 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
    - Stoppiness 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
TCS5422: 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
TCS5422: 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 <sub>1</sub>	L <sub>2</sub>	L <sub>3</sub>	L <sub>4</sub>	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 <sub>1</sub> )
7	0	2	4	5 → 0	10 (from L <sub>4</sub> )

April 27, 2021
TCS5422: 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?

Sloppiness	Time (seconds)
1	12
2	6
4	3
8	1.5
16	0.7
32	0.4
64	0.25
128	0.15
256	0.1
512	0.07
1024	0.05

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.97

## SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.98

## 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    TCCS422: 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:

```

1 // basic node structure
2 typedef struct __node_t {
3     int key;
4     struct __node_t *next;
5 } node_t;
6
7 // basic list structure (one used per list)
8 typedef struct __list_t {
9     node_t *head;
10    pthread_mutex_t lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
    
```

April 27, 2021    TCCS422: 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.)
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key = key;
27     new->next = L->head;
28     L->head = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
    
```

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.101

## CONCURRENT LINKED LIST - 3

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

```

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

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.102

## 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 ...

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.103

## CCL – SECOND IMPLEMENTATION

- Init and Insert

```

1 void List_Init(list_t *L) {
2     L->head = NULL;
3     pthread_mutex_init(&L->lock, NULL);
4 }
5
6 void List_Insert(list_t *L, int key) {
7     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10        perror("malloc");
11        return;
12    }
13    new->key = key;
14
15    // just lock critical section
16    pthread_mutex_lock(&L->lock);
17    new->next = L->head;
18    L->head = new;
19    pthread_mutex_unlock(&L->lock);
20 }
21
    
```

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.104

## CCL – SECOND IMPLEMENTATION - 2

- Lookup


```

(cont.)
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
    
```

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.105

## 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?



April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.106

## 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
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.107

## 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

April 27, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L9.108

## CONCURRENT QUEUE

- Remove from queue

```

1  typedef struct __node_t {
2      int value;
3      struct __node_t *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t *head;
8      node_t *tail;
9      pthread_mutex_t headLock;
10     pthread_mutex_t tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
(Cont.)
    
```

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.109

## CONCURRENT QUEUE - 2

- Add to queue

```

(Cont.)
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
(Cont.)
    
```

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.110

## 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    TCCS422: 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    TCCS422: 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

Inserts (Thousands)	Simple Concurrent List (seconds)	Concurrent Hash Table (seconds)
10	~1.5	~1.5
20	~3.5	~1.5
30	~6.5	~1.5
40	~11.5	~1.5

**The simple concurrent hash table scales magnificently.**

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.113

## CONCURRENT HASH TABLE

```

1  #define BUCKETS (101)
2
3  typedef struct __hash_t {
4      list_t lists[BUCKETS];
5  } hash_t;
6
7  void Hash_Init(hash_t *H) {
8      int i;
9      for (i = 0; i < BUCKETS; i++) {
10         List_Init(&H->lists[i]);
11     }
12 }
13
14 int Hash_Insert(hash_t *H, int key) {
15     int bucket = key % BUCKETS;
16     return List_Insert(&H->lists[bucket], key);
17 }
18
19 int Hash_Lookup(hash_t *H, int key) {
20     int bucket = key % BUCKETS;
21     return List_Lookup(&H->lists[bucket], key);
22 }
    
```

April 27, 2021    TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma    L9.114

### 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 structure
- 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 [polllev.com/app](https://polllev.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>

April 27, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L9.116

# QUESTIONS

