

TCSS 422: OPERATING SYSTEMS


Lock-based data structures II, Condition Variables, Concurrency Problems

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

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington

Tacoma



1

OBJECTIVES – 5/13

- **Questions from 5/6 & Midterm Distribution**
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Approximate Counter (Sloppy Counter)
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.2

2

ONLINE DAILY FEEDBACK SURVEY

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

TCSS 422 A > Assignments

Spring 2021

Search for Assignment

Home

Announcements

Zoom

Syllabus

Assignments

Discussions

Upcoming Assignments

TCSS 422 - Online Daily Feedback Survey - 4/1

Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts

May 13, 2025

TCSS422: Computer Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.3

3

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:

12345678910

Mostly Review To MeEqual New and ReviewMostly New to Me

Question 2

0.5 pts

Please rate the pace of today's class:

12345678910

SlowJust RightFast

May 13, 2025

TCSS422: Computer Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.4

4

MATERIAL / PACE

- Please classify your perspective on material covered in today’s class (48 of 63 respondents – 76.2%):
 - 1-mostly review, 5-equal new/review, 10-mostly new
 - **Average – 6.08** (↓ - **previous 6.70**)
- Please rate the pace of today’s class:
 - 1-slow, 5-just right, 10-fast
 - **Average – 4.90** (↓ - **previous 5.24**)

May 13, 2025

TCSS422: Computer Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.5

5

FEEDBACK FROM 5/6

- **What exactly is the difference between the terms concurrency and parallelism?**
 - This depends on if the computer has 1 or more CPU cores
 - On a single CPU core computer:
Concurrency is when multiple tasks can run in overlapping periods. It's an illusion of multiple tasks running in parallel because of a very fast switching by the CPU. The two tasks don't actually run at the same time on a single-core CPU.
 - **Parallelism** is when tasks actually run in parallel in multiple CPUs (or hyperthreads)

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.6

6

FEEDBACK - 2

- What would we say the Ideal sloppy threshold is?
- It is subject to our needs?
- Or is there a way to find a mid point between accuracy and efficiency?
- The ideal sloppy threshold depends on the goal.
- If your goal is accuracy, chose a low number
- If your goal is performance, chose a high number
- If your goal is both, you'll need to pick a number in-between to balance the trade-off between accuracy and performance

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.7

7

MIDTERM RESULTS

- Statistics
- Average: 78.81
- Mode: 84.0
- Median: 80.0
- Min score: 45
- Lower quartile: 77.00
- 2nd quartile: 80.0
- 3rd quartile: 84
- Max score: 91
- Standard deviation: 8.93
- Curve: +7
- Question 2 Correction: +4

Score Range	# of students
100-91	0
91-84	1
84-76	7
76-68	6
68-60	11
60-52	14
52-44	9
44-36	4
36-28	4
28-20	3
20-12	0
12-4	1
4-0	1
0-8	0
8-16	0
16-24	0
24-32	0
32-40	1
40-31	1

May 13, 2025

TCSS562: Software Engineering for Cloud Computing [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.8

8

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- **Assignment 0 Grades Posted**
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Approximate Counter (Sloppy Counter)
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.9

9

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- **Assignment 1 – May 13 --> May 16**
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Approximate Counter (Sloppy Counter)
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.10

10

GOOGLE BIG TEXTFILE

- <https://faculty.washington.edu/wlloyd/courses/tcss422/assignments/googlebig.txt.gz>

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.11
--------------	---	--------

11

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted**
- Chapter 29: Lock Based Data Structures
 - Approximate Counter (Sloppy Counter)
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.12
--------------	---	--------

12

TUTORIAL 2

- Pthread Tutorial
- Practice using:
 - pthreads
 - Locks
 - Condition variables
- Generate and visualize prime number generation in parallel
- To be posted in next couple of days

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.13
--------------	---	--------

13

QUIZ 3

- Build a synchronized array thread-safe data structure
- As a class activity (~30 min allocated)
- Thursday May 15
- Bring Laptops
- Groups of 1 or 2

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.14
--------------	---	--------

14

CATCH UP FROM LECTURE 11

- Switch to Lecture 11 Slides
- Slides L11.20 to L11.40
(Chapter 29 –Lock Based Data Structures)

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.15
--------------	---	--------


15

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- **Chapter 30: Condition Variables**
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.16
--------------	---	--------

16



CHAPTER 30 – CONDITION VARIABLES

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.1
7

17

CONDITION VARIABLES

- There are many cases where a thread wants to wait for another thread before proceeding with execution
- Consider when a precondition must be fulfilled before it is meaningful to proceed ...

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.18

18

CONDITION VARIABLES - 2



- Support a signaling mechanism to alert threads when preconditions have been satisfied
- Eliminate busy waiting
- Alert one or more threads to “consume” a result, or respond to state changes in the application
- Threads are placed on **(FIFO) queue** to **WAIT** for signals
- **Signal**: wakes one thread (thread waiting longest)
broadcast wakes all threads (ordering by the OS)

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.19

19

CONDITION VARIABLES - 3

▪ Condition variable

```
pthread_cond_t c;
```

- Requires initialization

▪ Condition API calls

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

- **wait()** accepts a mutex parameter
 - Releases lock, puts thread to sleep, thread added to FIFO queue
- **signal()**
 - Wakes up thread, awakening thread acquires lock

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.20

20

CONDITION VARIABLES - QUESTIONS

- Why would we want to put waiting threads on a queue? why not use a stack?
 - Queue (FIFO), Stack (LIFO)
- Why do we want to not busily wait for the lock to become available?
 - Using condition variables eliminates busy waiting by putting threads to “sleep” and yielding the CPU.
- A program has 10-threads, where 9 threads are waiting. The working thread finishes and broadcasts that the lock is available. **What happens next?**
 - All threads woken up in FIFO order - based on when started to wait

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.21

21

MATRIX GENERATOR

Matrix generation example

Chapter 30
signal.c

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.22

22

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - **Producer/Consumer**
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.23

23

MATRIX GENERATOR

- The worker thread produces a matrix
 - Matrix stored using shared global pointer
- The main thread consumes the matrix
 - Calculates the average element
 - Display the matrix
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Example program: "nosignal.c"

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.24

24

ATTEMPT TO USE CONDITION VARIABLE WITHOUT A WHILE STATEMENT

```
1  void thr_exit() {           ← Child calls
2      done = 1;
3      pthread_cond_signal(&c);
4  }
5
6  void thr_join() {           ← Parent calls
7      if (done == 0)
8          pthread_cond_wait(&c);
9  }
```

- Subtle race condition introduced
- **Parent** thread calls **thr_join()** and executes comparison (line 7)
- Context switches to the child
- The **child** runs **thr_exit()** and signals the parent, but the parent is not waiting yet. (*parent has not reached line 8*)
- **The signal is lost !**
- The parent deadlocks

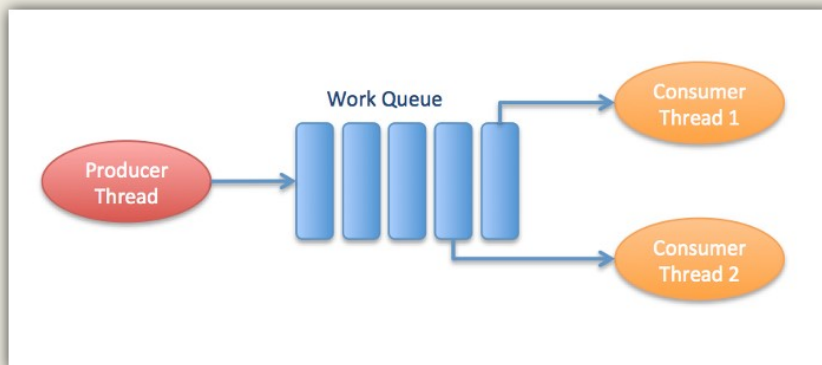
May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.25

25

PRODUCER / CONSUMER



May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.26

26

PRODUCER / CONSUMER

■ Producer

- Produces items – e.g. child the makes matrices
- Places them in a buffer
 - Example: the buffer size is only 1 element (single array pointer)

■ Consumer

- Grabs data out of the buffer
- Our example: parent thread receives dynamically generated matrices and performs an operation on them
 - Example: calculates average value of every element (integer)

■ Multithreaded web server example

- Http requests placed into work queue; threads process

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.27

27

PRODUCER / CONSUMER - 2

■ Producer / Consumer is also known as **Bounded Buffer**

■ Bounded buffer

- Similar to piping output from one Linux process to another
- `grep pthread signal.c | wc -l`
- Synchronized access:
sends output from `grep` → `wc` as it is produced
- File stream

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.28

28

WE WILL RETURN AT 5:00PM

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma



L12.2
9

29

PUT/GET ROUTINES

- Buffer is a one element shared data structure (int)
- Producer “puts” data, Consumer “gets” data
- “Bounded Buffer” shared data structure requires synchronization

```
1  int buffer;
2  int count = 0;  // initially, empty
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.30

30

PRODUCER / CONSUMER - 3

- Producer adds data
- Consumer removes data (busy waiting)
- Without synchronization:

1. Producer Function

2. Consumer Function

```
1 void *producer(void *arg) {
2     int i;
3     int loops = (int) arg;
4     for (i = 0; i < loops; i++) {
5         put(i);
6     }
7 }
8
9 void *consumer(void *arg) {
10    int i;
11    while (1) {
12        int tmp = get();
13        printf("%d\n", tmp);
14    }
15 }
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.31

31

PRODUCER / CONSUMER - 3

- The shared data structure needs synchronization!

```
1 cond_t cond;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex); // p1
8         if (count == 1) // p2
9             Pthread_cond_wait(&cond, &mutex); // p3
10        put(i); // p4
11        Pthread_cond_signal(&cond); // p5
12        Pthread_mutex_unlock(&mutex); // p6
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex); // c1
20     }
21 }
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.32

32

PRODUCER/CONSUMER - 4

```
20         if (count == 0)                // c2
21             Pthread_cond_wait(&cond, &mutex);    // c3
22         int tmp = get();                // c4
23             Pthread_cond_signal(&cond);        // c5
24             Pthread_mutex_unlock(&mutex);      // c6
25             printf("%d\n", tmp);
26         }
27     }
```

Consumer

■ This code as-is works with just:

(1) Producer
(1) Consumer

■ **PROBLEM:** no while. If thread wakes up it **MUST** execute

■ If we scale to (2+) consumer's it fails

■ How can it be fixed ?

May 13, 2025

TCCS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.33

33

EXECUTION TRACE:
NO WHILE, 1 PRODUCER, 2 CONSUMERS

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

No while can result in second consumer having no data to consume

→ c3

→ c4

	T _{c1}	State	T _{c2}	State	T _p	State	Count	Comment
	c1	Running		Ready		Ready	0	
	c2	Running		Ready		Ready	0	
	c3	Sleep		Ready		Ready	0	Nothing to get
		Sleep		Ready	p1	Running	0	
		Sleep		Ready	p2	Running	0	
		Sleep		Ready	p4	Running	1	Buffer now full
		Ready		Ready	p5	Running	1	T _{c1} awoken
		Ready		Ready	p6	Running	1	
		Ready		Ready	p1	Running	1	
		Ready		Ready	p2	Running	1	
		Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	→ c1	Running		Sleep		1	T _{c2} sneaks in ...
	Ready	→ c2	Running		Sleep		1	
	Ready	→ c4	Running		Sleep		0	... and grabs data
	Ready	→ c5	Running		Ready		0	T _p awoken
	Ready	→ c6	Running		Ready		0	
	→ c4	Running		Ready		Ready	0	Oh oh! No data

May 13, 2025

TCCS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.34

34

PRODUCER/CONSUMER
SYNCHRONIZATION

- When producer threads awake, they do not check if there is any data in the buffer...
 - Need “while” statement, “if” statement is *Insufficient ...*
- What if T_p puts a value, wakes T_{c1} whom consumes the value
- Then T_p has a value to put, but T_{c1} ’s signal on &cond wakes T_{c2}
- There is nothing for T_{c2} consume, so T_{c2} sleeps
- T_{c1} , T_{c2} , and T_p all sleep forever
- T_{c1} needs to wake T_p to T_{c2}

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.35

35

EXECUTION TRACE:
WHILE, 1 CONDITION, 1 PRODUCER, 2 CONSUMERS

Legend

c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

One condition
can result in
waking up wrong
thread (consumer
instead of producer)

→

→

→

	T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
	c1	Running		Ready		Ready	0	
	c2	Running		Ready		Ready	0	
	c3	Sleep		Ready		Ready	0	Nothing to get
		Sleep	c1	Running		Ready	0	
		Sleep	c2	Running		Ready	0	
		Sleep	c3	Sleep		Ready	0	Nothing to get
		Sleep		Sleep	p1	Running	0	
		Sleep		Sleep	p2	Running	0	
		Sleep		Sleep	p4	Running	1	Buffer now full
		Ready		Sleep	p5	Running	1	T_{c1} awoken
		Ready		Sleep	p6	Running	1	
		Ready		Sleep	p1	Running	1	
		Ready		Sleep	p2	Running	1	
		Ready		Sleep	p3	Sleep	1	Must sleep (full)
	c2	Running		Sleep		Sleep	1	Recheck condition
	c4	Running		Sleep		Sleep	0	T_{c1} grabs data
	c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.36

36

EXECUTION TRACE – 2
WHILE, 1 CONDITION, 1 PRODUCER, 2 CONSUMERS

■ T_{c2} runs, no data to consume

Legend

c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T _{c1}	State	T _{c2}	State	T _p	State	Count	Comment
...	(cont.)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	→ c2	Running		Sleep	0	
	Sleep	→ c3	Sleep		Sleep	0	Everyone asleep ...

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.37

37

TWO CONDITIONS

■ Required w/ multiple producer and consumer threads

■ Use two condition variables: empty & full

- One condition handles the producer
- the other the consumer

```
1      cond_t empty, full;
2      mutex_t mutex;
3
4      void *producer(void *arg) {
5          int i;
6          for (i = 0; i < loops; i++) {
7              pthread_mutex_lock(&mutex);
8              while (count == 1)
9                  pthread_cond_wait(&empty, &mutex);
10             put(i);
11             pthread_cond_signal(&full);
12             pthread_mutex_unlock(&mutex);
13         }
14     }
15
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.38

38

FINAL PRODUCER/CONSUMER

- Change buffer from int, to int buffer[MAX]
- Add indexing variables
- >> Becomes **BOUNDED BUFFER**, can store multiple matrices

```

1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }

```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.39

39

FINAL P/C - 2

```

1  cond_t empty, full;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          pthread_mutex_lock(&mutex);           // p1
8          while (count == MAX)                  // p2
9              pthread_cond_wait(&empty, &mutex); // p3
10         put(i);                                // p4
11         pthread_cond_signal(&full);            // p5
12         pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         pthread_mutex_lock(&mutex);             // c1
20         while (count == 0)                      // c2
21             pthread_cond_wait(&full, &mutex);   // c3
22         int tmp = get();                        // c4

```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.40

40

FINAL P/C - 3

```
(Cont.)
23      Pthread_cond_signal(&empty);           // c5
24      Pthread_mutex_unlock(&mutex);          // c6
25      printf("%d\n", tmp);
26      }
27      }
```

- **Producer: only sleeps when buffer is full**
- **Consumer: only sleeps if buffers are empty**

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.41
--------------	---	--------

41

Using one condition variable, and no while loop is sufficient to synchronize access to a bounded buffer shared by:

- 1 Producer, 1 Consumer Thread
- 2 Consumers, 1 Producer Thread
- 2+ Producers, 2+ Consumer Threads
- All of the above
- None of the above

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

42

Using two condition variables, and a while loop is sufficient to synchronize access to a bounded buffer shared by:

- 1 Producer, 1 Consumer Thread
- 2 Consumers, 1 Producer Thread
- 2+ Producers, 2+ Consumer Threads
- All of the above
- None of the above

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

43

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - **Covering Conditions**
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.44
--------------	---	--------

44

COVERING CONDITIONS

- A condition that covers all cases (conditions):
- Excellent use case for `pthread_cond_broadcast`
- Consider memory allocation:
 - When a program deals with huge memory allocation/deallocation on the heap
 - Access to the heap must be managed when memory is scarce

PREVENT: Out of memory:

- queue requests until memory is free

- Which thread should be woken up?

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.45

45

COVERING CONDITIONS - 2

```
1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8 void *
9 allocate(int size) {
10     pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         pthread_cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     pthread_mutex_lock(&m);
21     bytesLeft += size;
22     pthread_cond_signal(&c);
23     pthread_mutex_unlock(&m);
24 }
```

Check available memory

Broadcast

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.46

46

COVER CONDITIONS - 3

- Broadcast awakens all blocked threads requesting memory
- Each thread evaluates if there's enough memory: (bytesLeft < size)
 - Reject: requests that cannot be fulfilled- go back to sleep
 - *Insufficient memory*
 - Run: requests which can be fulfilled
 - with newly available memory!
- **Another use case:** coordinate a group of busy threads to gracefully end, to EXIT the program
- **Overhead**
 - Many threads may be awoken which can't execute

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.47

47

CHAPTER 31: SEMAPHORES

- Offers a combined C language construct that can assume the role of a lock or a condition variable depending on usage
 - Allows fewer concurrency related variables in your code
 - Potentially makes code more ambiguous
 - For this reason, with limited time in a 10-week quarter, we do not cover
- **Ch. 31.6 – Dining Philosophers Problem**
 - Classic computer science problem about sharing eating utensils
 - Each philosopher tries to obtain two forks in order to eat
 - Mimics deadlock as there are not enough forks
 - Solution is to have one left-handed philosopher that grabs forks in opposite order



May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.48

48

OBJECTIVES – 5/13

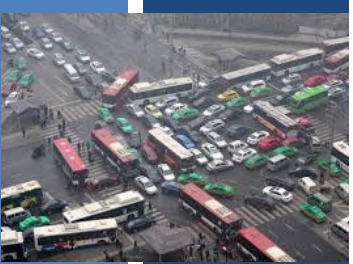
- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.49

49



CHAPTER 32 –
CONCURRENCY
PROBLEMS

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.50

50

CONCURRENCY BUGS IN
OPEN SOURCE SOFTWARE

- “Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics”
 - Shan Lu et al.
 - Architectural Support For Programming Languages and Operating Systems (ASPLOS 2008), Seattle WA

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		74	31

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.51

51

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.52

52

NON-DEADLOCK BUGS

- Majority of concurrency bugs
- Most common:
 - Atomicity violation: forget to use locks
 - Order violation: failure to initialize lock/condition before use

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.53

53

ATOMICITY VIOLATION - MYSQL

- Two threads access the `proc_info` field in `struct thd`
- `NULL` is 0 in C
- Mutually exclusive access to shared memory among separate threads is not enforced (e.g. non-atomic)
- Simple example: ***proc_info* deleted**

Programmer intended
variable to be accessed
atomically...



```
1  Thread1::  
2  if(thd->proc_info){  
3  ...  
4      fputs(thd->proc_info , ...);  
5  ...  
6  }  
7  
8  Thread2::  
9  thd->proc_info = NULL;
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.54

54

ATOMICITY VIOLATION - SOLUTION

- Add locks for all uses of: `thd->proc_info`

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread1::
4 pthread_mutex_lock(&lock);
5 if(thd->proc_info){
6     ...
7     fputs(thd->proc_info , ...);
8     ...
9 }
10 pthread_mutex_unlock(&lock);
11
12 Thread2::
13 pthread_mutex_lock(&lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&lock);
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.55

55

ORDER VIOLATION BUGS

- Desired order between memory accesses is flipped
- E.g. something is checked before it is set
- Example:

```
1 Thread1::
2 void init(){
3     mThread = PR_CreateThread(mMain, ...);
4 }
5
6 Thread2::
7 void mMain(...){
8     mState = mThread->State
9 }
```

- What if `mThread` is not initialized?

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.56

56

ORDER VIOLATION - SOLUTION

■ Use condition & signal to enforce order

```

1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit = 0;
4
5  Thread 1::
6  void init(){
7      ...
8      mThread = PR_CreateThread(mMain,...);
9
10     // signal that the thread has been created.
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread2::
19 void mMain(...) {
20     ...

```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.57

57

ORDER VIOLATION - SOLUTION - 2

■ Use condition & signal to enforce order

```

21 // wait for the thread to be initialized ...
22 pthread_mutex_lock(&mtLock);
23 while(mtInit == 0)
24     pthread_cond_wait(&mtCond, &mtLock);
25 pthread_mutex_unlock(&mtLock);
26
27 mState = mThread->State;
28 ...
29 }

```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.58

58

NON-DEADLOCK BUGS - 1

- 97% of Non-Deadlock Bugs were
 - Atomicity
 - Order violations
- Consider what is involved in “spotting” these bugs in code
 - >> *no use of locking constructs to search for*
- Desire for automated tool support (IDE)

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.59

59

NON-DEADLOCK BUGS - 2

- Atomicity
 - How can we tell if a given variable is shared?
 - Can search the code for uses
 - How do we know if all instances of its use are shared?
 - Can some non-synchronized, non-atomic uses be legal?
 - Legal uses: before threads are created, after threads exit
 - Must verify the scope
- Order violation
 - Must consider all variable accesses
 - Must know desired order


May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.60

60

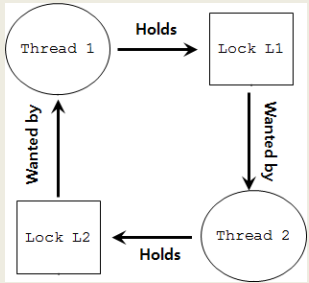
DEADLOCK BUGS



- Presence of a cycle in code
- Thread 1 acquires lock L1, waits for lock L2
- Thread 2 acquires lock L2, waits for lock L1

Thread 1:	Thread 2:
<code>lock (L1);</code>	<code>lock (L2);</code>
<code>lock (L2);</code>	<code>lock (L1);</code>

- Both threads can block, unless one manages to acquire both locks



```
graph TD; T1((Thread 1)) -- Holds --> L1[Lock L1]; L1 -- "Wanted by" --> T2((Thread 2)); T2 -- Holds --> L2[Lock L2]; L2 -- "Wanted by" --> T1;
```

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.61
--------------	---	--------

61

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - **Deadlock causes**
 - Deadlock prevention

May 13, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L12.62
--------------	---	--------

62

REASONS FOR DEADLOCKS

- **Complex code**
 - Must avoid circular dependencies – can be hard to find...
- **Encapsulation hides potential locking conflicts**
 - Easy-to-use APIs embed locks inside
 - Programmer doesn't know they are there
 - Consider the Java Vector class:

```
1  Vector v1,v2;  
2  v1.AddAll(v2);
```
- **Vector is thread safe (synchronized) by design**
- **If there is a v2.AddAll(v1); call at nearly the same time deadlock could result**

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.63

63

CONDITIONS FOR DEADLOCK

- Four conditions are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.64

64

OBJECTIVES – 5/13

- Questions from 5/6 & Midterm Distribution
- Assignment 0 Grades Posted
- Assignment 1 – May 13 --> May 16
- Tutorial 2: Pthread Tutorial - to be posted
- Chapter 29: Lock Based Data Structures
 - Sloppy Counter
 - Concurrent Structures: Linked List, Queue, Hash Table
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - **Deadlock prevention**

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.65

65

PREVENTION – MUTUAL EXCLUSION

- Build wait-free data structures
 - Eliminate locks altogether
 - Build structures using CompareAndSwap atomic CPU (HW) instruction
- C pseudo code for CompareAndSwap
- Hardware executes this code atomically

```
1  int CompareAndSwap(int *address, int expected, int new){
2      if(*address == expected){
3          *address = new;
4          return 1; // success
5      }
6      return 0;
7  }
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.66

66

PREVENTION – MUTUAL EXCLUSION - 2

■ Recall atomic increment

```
1 void AtomicIncrement(int *value, int amount){
2     do{
3         int old = *value;
4     }while( CompareAndSwap(value, old, old+amount)!=0);
5 }
```

- Compare and Swap tries over and over until successful
- CompareAndSwap is guaranteed to be atomic
- When it runs it is **ALWAYS** atomic (at HW level)

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.67

67

MUTUAL EXCLUSION: LIST INSERTION

■ Consider list insertion

```
1 void insert(int value){
2     node_t * n = malloc(sizeof(node_t));
3     assert( n != NULL );
4     n->value = value ;
5     n->next = head;
6     head = n;
7 }
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.68

68

MUTUAL EXCLUSION – LIST INSERTION - 2

■ Lock based implementation

```
1 void insert(int value){
2     node_t * n = malloc(sizeof(node_t));
3     assert( n != NULL );
4     n->value = value ;
5     lock(listlock); // begin critical section
6     n->next = head;
7     head = n;
8     unlock(listlock) ; //end critical section
9 }
```

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.69

69

MUTUAL EXCLUSION – LIST INSERTION - 3

■ Wait free (no lock) implementation

```
1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     do {
6         n->next = head;
7     } while (!CompareAndSwap(&head, n->next, n));
8 }
```

■ Assign &head to n (new node ptr)

■ Only when head = n->next

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.70

70

CONDITIONS FOR DEADLOCK

■ Four conditions are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.71

71

PREVENTION LOCK – HOLD AND WAIT

■ Problem: acquire all locks atomically

■ Solution: use a “lock” “lock”... (like a guard lock)

```
1 lock(prevention);
2 lock(L1);
3 lock(L2);
4 ...
5 unlock(prevention);
```

■ Effective solution – guarantees no race conditions while acquiring L1, L2, etc.

■ Order doesn't matter for L1, L2

■ Prevention (GLOBAL) lock decreases concurrency of code

- Acts Lowers lock granularity

■ Encapsulation: consider the Java Vector class...

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.72

72

CONDITIONS FOR DEADLOCK

■ Four conditions are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
➡ No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.73

73

PREVENTION – NO PREEMPTION

■ When acquiring locks, don't BLOCK forever if unavailable...

■ pthread_mutex_trylock() - try once

■ pthread_mutex_timedlock() - try and wait awhile

```
1 top:
2   lock(L1);
3   if( tryLock(L2) == -1 ){
4       unlock(L1);
5       goto top;
6   }
```

■ Eliminates deadlocks

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.74

74

NO PREEMPTION – LIVELOCKS PROBLEM


■ Can lead to livelock

```
1  top:
2      lock(L1);
3      if( tryLock(L2) == -1 ){
4          unlock(L1);
5          goto top;
6      }
```

■ Two threads execute code in parallel →
always fail to obtain both locks

■ Fix: add random delay

- Allows one thread to win the livelock race!



May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.75

75

CONDITIONS FOR DEADLOCK

■ Four conditions are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
→ Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.76

76

PREVENTION – CIRCULAR WAIT

- Provide total ordering of lock acquisition throughout code
 - Always acquire locks in same order
 - L1, L2, L3, ...
 - Never mix: L2, L1, L3; L2, L3, L1; L3, L1, L2....
- Must carry out same ordering through entire program

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.77

77

CONDITIONS FOR DEADLOCK

- If any of the following conditions DOES NOT EXSIST, describe why deadlock can not occur?

Condition	Description
➡ Mutual Exclusion	Threads claim exclusive control of resources that they require.
➡ Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
➡ No preemption	Resources cannot be forcibly removed from threads that are holding them.
➡ Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.78

78

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

DEADLOCK AVOIDANCE VIA INTELLIGENT SCHEDULING

- | | T1 | T2 | T3 | T4 |
|----|-----|-----|-----|----|
| L1 | yes | yes | no | no |
| L2 | yes | yes | yes | no |

Slides by Wes J. Lloyd

INTELLIGENT SCHEDULING - 2

■ Scheduler produces schedule:

CPU 1

T3

T4

CPU 2

T1

T2

■ No deadlock can occur

■ Consider:

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.81

81

INTELLIGENT SCHEDULING - 3

■ Scheduler produces schedule

CPU 1

T4

CPU 2

T1

T2

T3

■ Scheduler must be conservative and not take risks

■ Slows down execution – many threads

■ There has been limited use of these approaches given the difficulty having intimate lock knowledge about every thread

May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.82

82

DETECT AND RECOVER

- Allow deadlock to occasionally occur and then take some action.
 - Example: When OS freezes, reboot...
- How often is this acceptable?
 - Once per year
 - Once per month
 - Once per day
 - *Consider the effort tradeoff of finding every deadlock bug*
- Many database systems employ deadlock detection and recovery techniques.


May 13, 2025

TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma

L12.83

83

QUESTIONS



84