


# 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

November 9, 2021

TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington

Tacoma

1

## OBJECTIVES – 11/9

- **Questions from 11/2 & Midterm Review**
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021

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

L11.2

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

Quiz 0 - C background survey

November 9, 2021TCSS422: Computer Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - TacomaL11.4

4

### 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

November 9, 2021TCSS422: Computer Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - TacomaL11.5

5

## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (29 respondents):
  - 1-mostly review, 5-equal new/review, 10-mostly new
  - **Average - 5.81** (↓ - **previous 6.50**)
  
- Please rate the pace of today's class:
  - 1-slow, 5-just right, 10-fast
  - **Average - 5.46** (↓ - **previous 5.48**)

November 9, 2021	TCSS422: Computer Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.6
------------------	--	-------

6

## FEEDBACK

- **Comment on the practice midterm vs. actual midterm:**  
**Suggest posting a list of essay style questions to help practice for the midterm**
- There were no essay questions on the midterm
- I think what is meant is to post questions that have more “background” setting up the question.
- Practice midterm Q5 has a lot of narrative, and to a lesser extent Q4 and Q7. All offer similar examples of questions with some background/narrative to set up the question

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.7
------------------	---	-------

7

# MIDTERM REVIEW

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.9
------------------	---	-------

9

# OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- **Assignment 0 Grades Posted**
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.10
------------------	---	--------

10

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- **Assignment 1 – Nov 12**
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.11
------------------	---	--------

11

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- **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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.12
------------------	---	--------

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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.13
------------------	---	--------

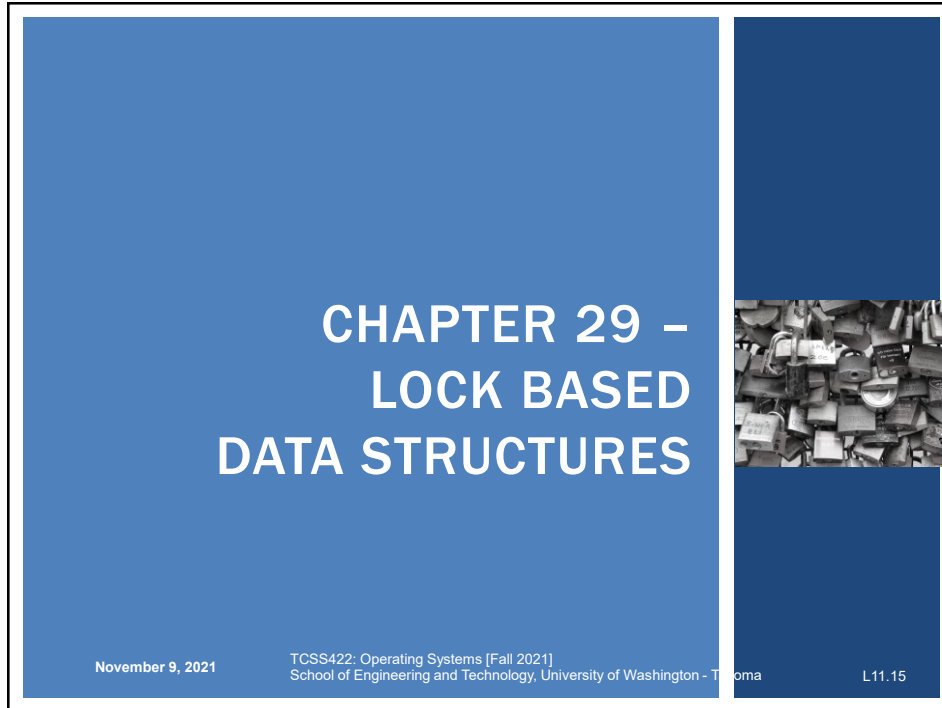
13

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.14
------------------	---	--------

14



CHAPTER 29 –  
LOCK BASED  
DATA STRUCTURES

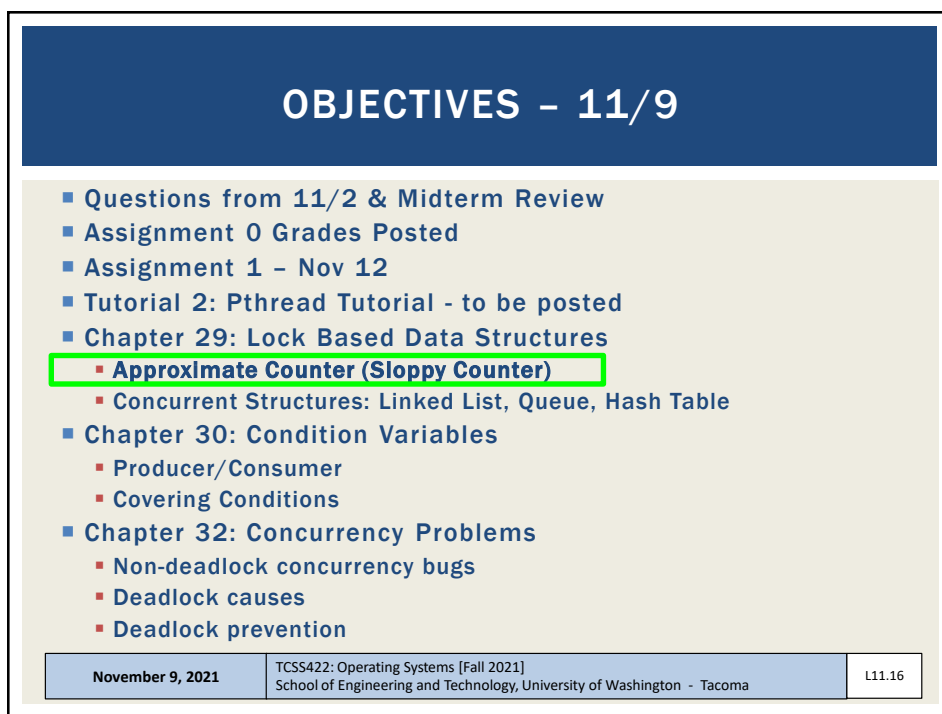
November 9, 2021

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

L11.15

The slide features a blue background with a vertical white line on the right side. The title is centered in white text. A small image of a pile of padlocks is visible on the right side of the slide.

15



OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021

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

L11.16

The slide has a dark blue header with the title in white. The main content is on a light beige background with a list of objectives. The 'Approximate Counter (Sloppy Counter)' item is highlighted with a green border. The footer is a light blue bar with white text.

16

## APPROXIMATE (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
    - Update threshold (S) – referred to as *sloppiness threshold*:  
How often to push local values to global counter
    - 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?

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.17
------------------	---	--------

17

## APPROXIMATE COUNTER – MAIN POINTS

- Idea of the Approximate 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
- Approximate counter: trade-off accuracy for speed
  - It's approximate because it's not so accurate (until the end)

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.18
------------------	---	--------

18



## OBJECTIVES - 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 - Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.19
------------------	---	--------

19

## 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
18 (Cont.)
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.20
------------------	---	--------

20

## 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      }
25      return -1; // fail
26      new->key = key;
27      new->next = L->head;
28      L->head = new;
29      pthread_mutex_unlock(&L->lock);
30      return 0; // success
31  }
(Cont.)
```

November 9, 2021

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

L11.21

21

## 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
45  }
```

November 9, 2021

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

L11.22

22

# 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 ...**

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.23
------------------	---	--------

23

# 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
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.24
------------------	---	--------

24

## 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  }
```

November 9, 2021

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

L11.25

25

## 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?



November 9, 2021

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

L11.26

26

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.27
------------------	---	--------

27

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.28
------------------	---	--------

28

## 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.)
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.29
------------------	---	--------

29

## 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
25        tmp->value = value;
26        tmp->next = NULL;
27
28        pthread_mutex_lock(&q->tailLock);
29        q->tail->next = tmp;
30        q->tail = tmp;
31        pthread_mutex_unlock(&q->tailLock);
32    }
(Cont.)
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.30
------------------	---	--------

30

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.31
------------------	---	--------

31

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.32
------------------	---	--------

32

## 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	~1.0	~0.1
20	~2.5	~0.1
30	~4.5	~0.1
40	~7.5	~0.1
50	~11.5	~0.1

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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.33
------------------	---	--------

33

## 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      }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.34
------------------	---	--------

34



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

- Locks are encapsulated within data structure code ensuring thread safety.
- Lock granularity tradeoff already optimized inside data structurew
- Multiple threads can more easily share data
- All of the above
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at [polllev.com/app](https://polllev.com/app)

35

## 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>

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.36
------------------	---	--------

36

**WE WILL RETURN AT  
2:50PM**



November 9, 2021 TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma L11.3  
7

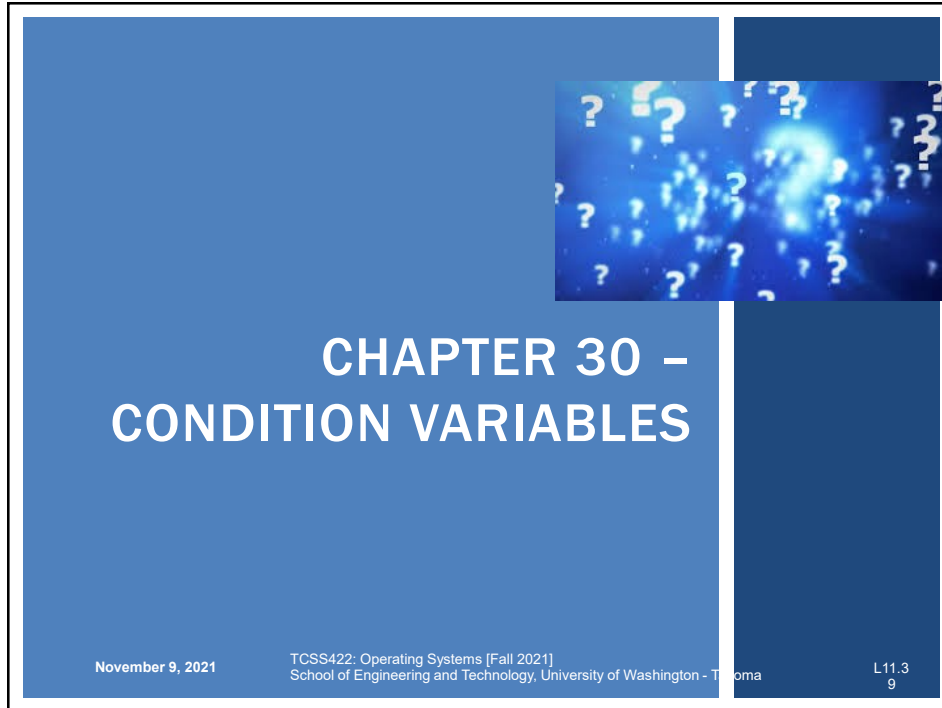
37

**OBJECTIVES – 11/9**

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021 TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma L11.38

38



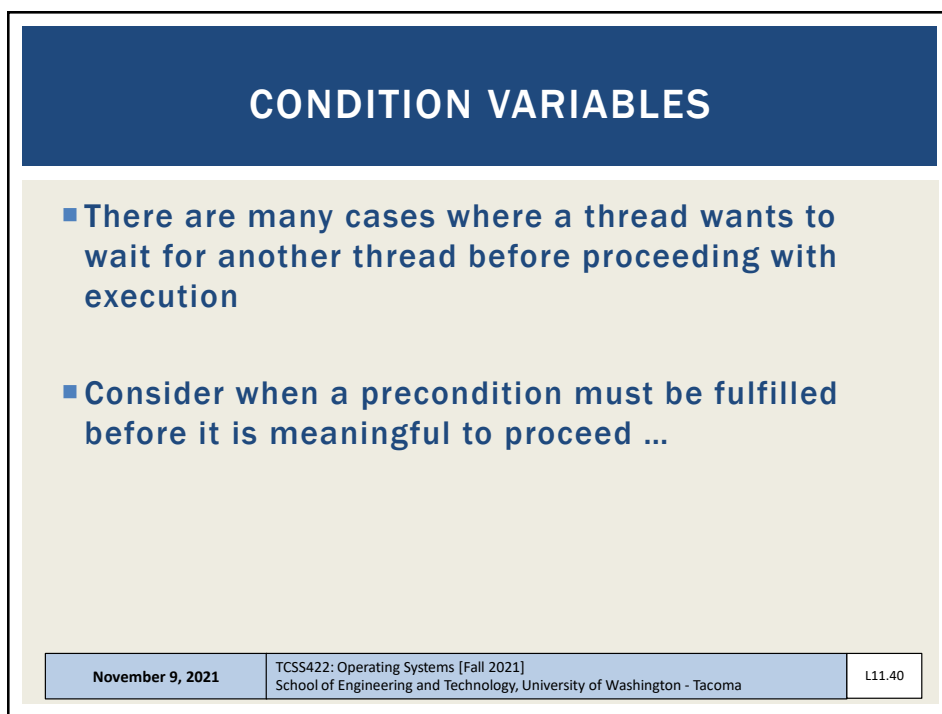
# CHAPTER 30 – CONDITION VARIABLES

November 9, 2021

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

L11.3  
9

39



## 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 ...


November 9, 2021

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

L11.40

40

## 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)

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.41
------------------	---	--------

41

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.42
------------------	---	--------

42

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.43
------------------	---	--------

43

## MATRIX GENERATOR

Matrix generation example

Chapter 30  
signal.c

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.44
------------------	---	--------

44

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.45
------------------	---	--------

45

## 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"

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.46
------------------	---	--------

46

## 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

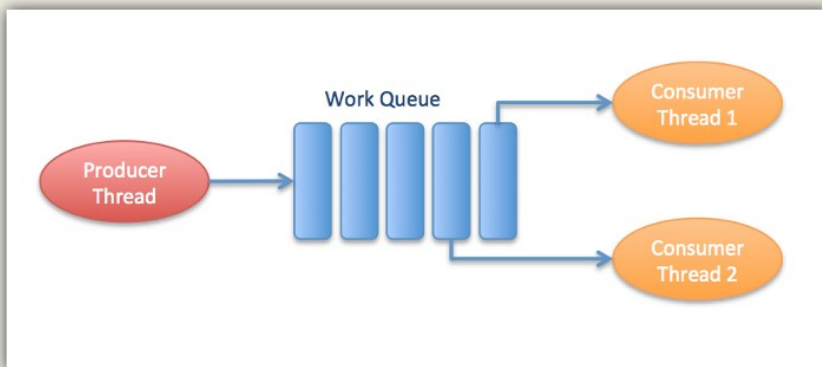
November 9, 2021

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

L11.47

47

## PRODUCER / CONSUMER



November 9, 2021

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

L11.48

48

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.49
------------------	---	--------

49

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.50
------------------	---	--------

50



## 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  }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.51
------------------	---	--------

51

## 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  }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.52
------------------	---	--------

52

## 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
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.53
------------------	---	--------

53

## 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 ?

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.54
------------------	---	--------

54

## EXECUTION TRACE: NO WHILE, 1 PRODUCER, 2 CONSUMERS

Two threads

**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
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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.55

55

## 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  $T_{c2}$

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.56

56

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

$T_{c1}$	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 $T_{c2}$

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

November 9, 2021      TCSS422: Operating Systems [Fall 2021]  
 School of Engineering and Technology, University of Washington - Tacoma      L11.57

57

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

▪  $T_{c2}$  runs, no data to consume

$T_{c1}$	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 ...

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

November 9, 2021      TCSS422: Operating Systems [Fall 2021]  
 School of Engineering and Technology, University of Washington - Tacoma      L11.58

58

## 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
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.59
------------------	---	--------

59

## 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    }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.60
------------------	---	--------

60

## 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);
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
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.61
------------------	---	--------

61

## 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**

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.62
------------------	---	--------

62

**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](https://pollev.com/app)

63

**Using one condition variable, with 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
- 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](https://pollev.com/app)

64

**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 [polllev.com/app](https://polllev.com/app)

65

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.66
------------------	---	--------

66



## 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?

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.67
------------------	---	--------

67

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.68
------------------	---	--------

68

## COVER CONDITIONS - 3

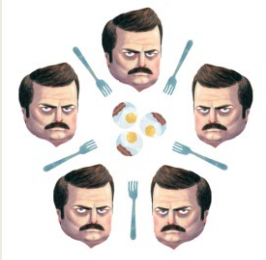
- Broadcast awakens all blocked threads requesting memory
- Each thread evaluates if there's enough memory: (bytesLeft < size)
  - Reject: requests that cannot be fulfilled- go back to sleep
    - *Insufficient memory*
  - Run: requests which **can** be fulfilled
    - with newly available memory!
- **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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.69
------------------	---	--------

69

## 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



November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.70
------------------	---	--------


70

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.71
------------------	---	--------

71



## CHAPTER 32 – CONCURRENCY PROBLEMS

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.72
------------------	---	--------

72

## 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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.73

73

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.74

74

## NON-DEADLOCK BUGS

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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.75
------------------	---	--------

75

## 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;
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.76
------------------	---	--------

76

## 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);
```

November 9, 2021

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

L11.77

77

## 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?

November 9, 2021

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

L11.78

78

## 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    ...
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.79
------------------	---	--------

79

## 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 }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.80
------------------	---	--------

80

## 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)

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.81
------------------	---	--------

81

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.82
------------------	---	--------

82

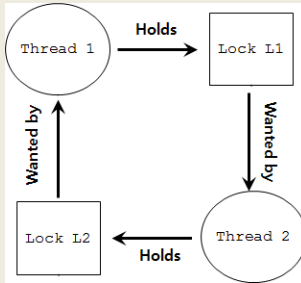


## DEADLOCK BUGS



- 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:  
lock (L1);     lock (L2);  
lock (L2);     lock (L1);
```



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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.83
------------------	---	--------

83

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.84
------------------	---	--------

84

## 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**

November 9, 2021TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - TacomaL11.85

85

## 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

November 9, 2021TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - TacomaL11.86

86

## OBJECTIVES – 11/9

- Questions from 11/2 & Midterm Review
- Assignment 0 Grades Posted
- Assignment 1 – Nov 12
- 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**

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.87
------------------	---	--------

87

## 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  }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.88
------------------	---	--------

88

## 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)

November 9, 2021

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

L11.89

89

## 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 }
```

November 9, 2021

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

L11.90

90

## 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 }
```

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.91
------------------	---	--------

91

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.92
------------------	---	--------

92

## 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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.93

93

## 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...

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.94

94

## 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


November 9, 2021    TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma    L11.95

95

## 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

November 9, 2021    TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma    L11.96


96

## 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!



November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.97

97

## 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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L11.98

98



## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.99
------------------	---	--------

99

## 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

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.100
------------------	---	---------

100

**The dining philosophers problem where 5 philosophers compete for 5 forks, and where a philosopher must hold two forks to eat involves which deadlock condition(s)?**

Mutual Exclusion  
Hold-and-wait  
No preemption  
Circular wait  
All 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)

101

## DEADLOCK AVOIDANCE VIA INTELLIGENT SCHEDULING

- Consider a smart scheduler
  - Scheduler knows which locks threads use
- Consider this scenario:
  - 4 Threads (T1, T2, T3, T4)
  - 2 Locks (L1, L2)
- Lock requirements of threads:

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

November 9, 2021      TCSS422: Operating Systems [Fall 2021]  
School of Engineering and Technology, University of Washington - Tacoma      L11.102

102

## 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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L11.103

103

## 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

November 9, 2021
TCSS422: Operating Systems [Fall 2021]  
 School of Engineering and Technology, University of Washington - Tacoma
L11.104

104


## 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.

November 9, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L11.105
------------------	---	---------

105

# QUESTIONS



106