

TCSS 422: OPERATING SYSTEMS

Concurrency Problems, Intro to Memory Virtualization



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

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington Tacoma

1

TCSS 422 – OFFICE HRS – WINTER 2026

- **Office Hours plan for Winter:**
- **Tuesday 2:30 - 3:30 pm Instructor Wes, Zoom**
- **Tue/Thur 6:00 - 7:00 pm Instructor Wes, CP 229/Zoom**
- **Tue 7:00 – 8:00 pm GTA Robert, Zoom Only This Week**
- **Wed 7:00 – 8:00 pm GTA Robert, Zoom Only This Week**

- **Instructor is available after class at 6pm in CP 229 each day**

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.2

2

OBJECTIVES – 2/24

- **Questions from 2/19**
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.3
-------------------	---	-------

3

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

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

February 24, 2026 TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma L13.5

5

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (25 of 63 respondents (5 online) – 54.35%):
 - 1-mostly review, 5-equal new/review, 10-mostly new
 - **Average – 6.92 (↑ - previous 6.72)**
- Please rate the pace of today's class:
 - 1-slow, 5-just right, 10-fast
 - **Average – 5.08 (↑ - previous 4.90)**

February 24, 2026 TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma L13.6

6

FEEDBACK FROM 2/19

- **What's a bounded buffer ?**
- A bounded buffer is a concurrent data structure
- Bounded buffer is like a concurrent queue
- Unlike a concurrent queue which is boundless, the bounded buffer only stores a fixed number of elements
 - Adding data must halt when out of capacity
 - Whereas a concurrent queue keeps growing

- **Why a bounded buffer ?**
- Memory is not infinite
- For most problems, there is a maximum limit on the number of items that can be produced before consumption - having a limitless queue is not necessary
- It is more efficient to have a fixed buffer size
- Resizing the structure (queue, list) involves extra work → rewriting the head, tail, etc.
- A bounded buffer is just an array with fixed indexing → simpler

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.7
-------------------	---	-------

7

FEEDBACK - 2

- **Why does the Bounded Buffer need to have synchronization?**
- Any bounded buffer shared between 1-to-n producer and 1-to-n consumer threads requires synchronization
- Producer threads and consumer threads both add and remove data from the shared buffer
- If unsynchronized:
 - a consumer can consume too much or unready data: CONSUME NULL
 - a producer can produce too much: BUFFER OVERFLOW

- **Is the bounded buffer considered embarrassingly parallel?**
- No. Bounded buffer shares state, has contention, ordering constraints (FIFO), blocking potential (full/empty) → threads must coordinate

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.8
-------------------	---	-------

8

OBJECTIVES – 2/24

- Questions from 2/19
- **Pthread Tutorial-March 5 / Assignment 2 posted**
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.9
-------------------	---	-------

9

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / **Assignment 2 posted**
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.10
-------------------	---	--------

10

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- **Quiz 3 – Synchronized Array (activity-Thursday)**
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.11
-------------------	---	--------

11

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - **Covering Conditions**
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.12
-------------------	---	--------

12

CATCH UP FROM LECTURE 12

- Switch to Lecture 12 Slides
- Slides L12.45 to L12.49
(Chapter 30 –Condition Variables)
(Chapter 31 – Semaphores)

May 13, 2025	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L12.13
--------------	---	--------

13

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Producer/Consumer
 - Covering Conditions
- **Chapter 32: Concurrency Problems**
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.14
-------------------	---	--------

14



CHAPTER 32 – CONCURRENCY PROBLEMS

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.15

15

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

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.16

16

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - **Non-deadlock concurrency bugs**
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.17
-------------------	---	--------

17

NON-DEADLOCK BUGS

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.18
-------------------	---	--------

18

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;
            
```

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.19
-------------------	---	--------

19

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);
            
```

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.20
-------------------	---	--------

20

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?

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.21
-------------------	---	--------

21

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    ...
```

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.22
-------------------	---	--------

22

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 }
```

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.23
-------------------	---	--------

23

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)

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.24
-------------------	---	--------

24

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.25
-------------------	---	--------

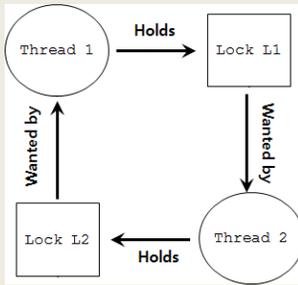
25

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 block leading to deadlock, 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;
```

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.26
-------------------	---	--------

26

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - **Deadlock causes**
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.27
-------------------	---	--------

27

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.28
-------------------	---	--------

28

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 resource that are being requested by the next thread in the chain

February 24, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L13.29

29

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

February 24, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L13.30

30

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 }
```

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.31

31

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)

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.32

32

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 }
```

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.33

33

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 }
```

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.34

34

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.35
-------------------	---	--------

35

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.36
-------------------	---	--------

36

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...

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.37

37

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

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.38

38

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.39
-------------------	---	--------

39

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!



February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.40
-------------------	---	--------

40

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

February 24, 2026TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - TacomaL13.41

41

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**

February 24, 2026TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - TacomaL13.42

42

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

February 24, 2026TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - TacomaL13.43

43

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

44

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

February 24, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L13.45

45

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

February 24, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L13.46

46

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

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.47
-------------------	---	--------

47

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.

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.48
-------------------	---	--------

48

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- **Chapter 13: Address Spaces**
- Chapter 14: The Memory API

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.49
-------------------	---	--------

49

CHAPTER 13: ADDRESS SPACES



February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.50
-------------------	---	--------

50

OBJECTIVES - 2/24

- **Chapter 13: Introduction to memory virtualization**
 - The address space
 - Goals of OS memory virtualization
- **Chapter 14: Memory API**
 - Common memory errors

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.51
-------------------	---	--------

51

MEMORY VIRTUALIZATION

- What is memory virtualization?
- This is not “virtual” memory,
 - Classic use of disk space as additional RAM
 - When available RAM was low
 - Less common recently

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.52
-------------------	---	--------

52

MEMORY VIRTUALIZATION - 2

- Presentation of system memory to each process
- Appears as if each process can access the entire machine's address space
- Each process's view of memory is isolated from others
- Everyone has their own sandbox

Process A



Process B



Process C



February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.53
-------------------	---	--------

53

MOTIVATION FOR MEMORY VIRTUALIZATION

- Easier to program
 - Programs don't need to understand special memory models
- Abstraction enables sophisticated approaches to manage and share memory among processes
- Isolation
 - From other processes: easier to code
- Protection
 - From other processes
 - From programmer error (segmentation fault)

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.54
-------------------	---	--------

54

**WE WILL RETURN AT
5:02PM**

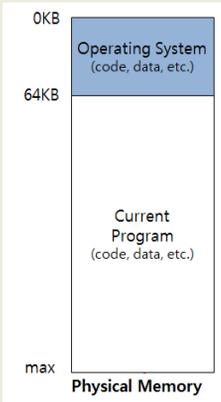


February 24, 2026 TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma L13.5
5

55

EARLY MEMORY MANAGEMENT

- Load one process at a time into memory
- Poor memory utilization
- Little abstraction



February 24, 2026 TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma L13.56

56

MULTIPROGRAMMING WITH SHARED MEMORY

- Later machines supported running multiple processes
- Swap out processes during I/O waits to increase system utilization and efficiency
- Swap entire memory of a process to disk for context switch
- Too slow, especially for large processes
- Solution →
 - Leave processes in memory
- Need to protect from errant memory accesses in a multiprocessing environment

Physical Memory

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.57
-------------------	---	--------

57

ADDRESS SPACE ★

- Easy-to-use abstraction of physical memory for a process
- Main elements:
 - Program code
 - Stack
 - Heap
- Example: 16KB address space

Address Space

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.58
-------------------	---	--------

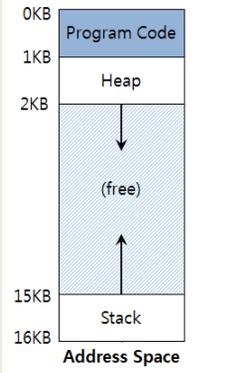
58

ADDRESS SPACE - 2 ★

- **Code**
 - Program code

- **Stack**
 - Program counter (PC)
 - Local variables
 - Parameter variables
 - Return values (for functions)

- **Heap**
 - Dynamic storage
 - Malloc() new()



February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.59
-------------------	---	--------

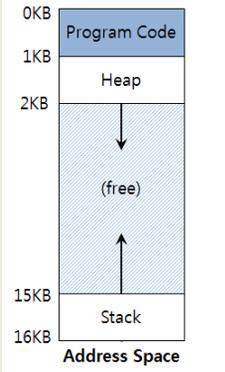
59

ADDRESS SPACE - 3 ★

- **Program code**
 - Static size

- **Heap and stack**
 - Dynamic size
 - Grow and shrink during program execution
 - Placed at opposite ends

- **Addresses are virtual**
 - They must be physically mapped by the OS



February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.60
-------------------	---	--------

60

VIRTUAL ADDRESSING

- Every address is virtual
 - OS translates virtual to physical addresses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code : %p\n", (void *) main);
    printf("location of heap : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

- EXAMPLE: virtual.c**

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.61
-------------------	---	--------

61

VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:

location of code: 0x400686
location of heap: 0x1129420
location of stack: 0x7ffe040d77e4

The diagram illustrates the virtual address space layout. At the top, the 'Code (Text)' region is located between 0x400000 and 0x401000. Below it is the 'Data' region between 0x401000 and 0xc2000. The 'Heap' region is between 0xc2000 and 0xd13000, with a downward arrow labeled 'heap' indicating its growth direction. A large shaded area labeled '(free)' represents the unallocated memory between 0xd13000 and 0x7ff9ca28000. The 'stack' region is between 0x7ff9ca28000 and 0x7ff9ca49000, with an upward arrow labeled 'stack' indicating its growth direction. The 'Stack' region is shown at the bottom of the stack area.

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.62
-------------------	---	--------

62

GOALS OF OS MEMORY VIRTUALIZATION

- **Transparency**
 - Memory shouldn't appear virtualized to the program
 - OS multiplexes memory among different jobs behind the scenes

- **Protection**
 - Isolation among processes
 - OS itself must be isolated
 - One program should not be able to affect another (or the OS)

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.63
-------------------	---	--------

63

GOALS - 2

- **Efficiency**
 - **Time**
 - Performance: virtualization must be fast

 - **Space**
 - Virtualization must not waste space
 - Consider data structures for organizing memory
 - Hardware support TLB: Translation Lookaside Buffer

- *Goals considered when evaluating memory virtualization schemes*

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.64
-------------------	---	--------

64

OBJECTIVES – 2/24

- Questions from 2/19
- Pthread Tutorial-March 5 / Assignment 2 posted
- Quiz 3 – Synchronized Array (activity-Thursday)
- Chapter 30: Condition Variables
 - Covering Conditions
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- **Chapter 14: The Memory API**

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.65
-------------------	---	--------

65

CHAPTER 14: THE MEMORY API



February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.66
-------------------	---	--------

66

OBJECTIVES – 2/24

- **Chapter 13: Introduction to memory virtualization**
 - The address space
 - Goals of OS memory virtualization
- **Chapter 14: Memory API**
 - Common memory errors

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.67
-------------------	---	--------

67

MALLOC()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- **Allocates memory on the heap**
- **size_t** unsigned integer (must be +)
- **size** size of memory allocation in bytes

- **Returns**
- **SUCCESS:** A void * to a memory address
- **FAIL:** NULL

- **sizeof()** often used to ask the system how large a given datatype or struct is

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.68
-------------------	---	--------

68

SIZEOF()

- Not safe to assume data type sizes using different compilers, systems
- Dynamic array of 10 ints
- Static array of 10 ints

```
int *x = malloc(10 * sizeof(int));  
printf("%d\n", sizeof(x));
```

4

```
int x[10];  
printf("%d\n", sizeof(x));
```

40

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.69
-------------------	---	--------

69

FREE()

```
#include <stdlib.h>  
  
void free(void* ptr)
```

- Free memory allocated with malloc()
- Provide: (void *) ptr to malloc'd memory
- Returns: nothing

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.70
-------------------	---	--------

70

```
#include<stdio.h>

int * set_magic_number_a()
{
    int a =53247;
    return &a;
}

void set_magic_number_b()
{
    int b = 11111;
}

int main()
{
    int * x = NULL;
    x = set_magic_number_a();
    printf("The magic number is=%d\n",*x);
    set_magic_number_b();
    printf("The magic number is=%d\n",*x);
    return 0;
}
```

71

What will this code do?

71

```
#include<stdio.h>

int * set_magic_number_a()
{
    int a =53247;
    return &a;
}

void set_magic_number_b()
{
    int b = 11111;
}

int main()
{
    int * x = NULL;
    x = set_magic_number_a();
    printf("The magic number is=%d\n",*x);
    set_magic_number_b();
    printf("The magic number is=%d\n",*x);
    return 0;
}
```

72

What will this code do?

Output:
\$./pointer_error
The magic number is=53247
The magic number is=11111

We have not changed *x but
the value has changed!!
Why?

72

DANGLING POINTER (1/2)

- Dangling pointers arise when a variable referred (a) goes “out of scope”, and it’s memory is destroyed/overwritten (by b) without modifying the value of the pointer (*x).
- The pointer still points to the original memory location of the deallocated memory (a), which has now been reclaimed for (b).

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.73

73

DANGLING POINTER (2/2)

- Fortunately in the case, a compiler warning is generated:

```
$ g++ -o pointer_error -std=c++0x pointer_error.cpp
```

```
pointer_error.cpp: In function ‘int*  
set_magic_number_a()’:  
pointer_error.cpp:6:7: warning: address of local  
variable ‘a’ returned [enabled by default]
```

- This is a common mistake - - -
accidentally referring to addresses that have gone “out of scope”

February 24, 2026

TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma

L13.74

74

CALLOC()



```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate “C”lear memory on the heap
- Calloc wipes memory in advance of use...
- `size_t num` : number of blocks to allocate
- `size_t size` : size of each block(in bytes)

■ Calloc() prevents...

```
char *dest = malloc(20);
printf("dest string=%s\n", dest);
```

dest string=□□F

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.75
-------------------	---	--------

75

REALLOC()



```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Resize an existing memory allocation
- Returned pointer may be same address, or a new address
 - New if memory allocation must move
- `void *ptr`: Pointer to memory block allocated with malloc, calloc, or realloc
- `size_t size`: New size for the memory block(in bytes)

■ EXAMPLE: realloc.c

■ EXAMPLE: nom.c

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.76
-------------------	---	--------

76

DOUBLE FREE

```
int *x = (int *)malloc(sizeof(int)); // allocated  
free(x); // free memory  
free(x); // free repeatedly
```

- Can't deallocate twice
- Second call core dumps

Undefined Error

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.77
-------------------	---	--------

77

SYSTEM CALLS

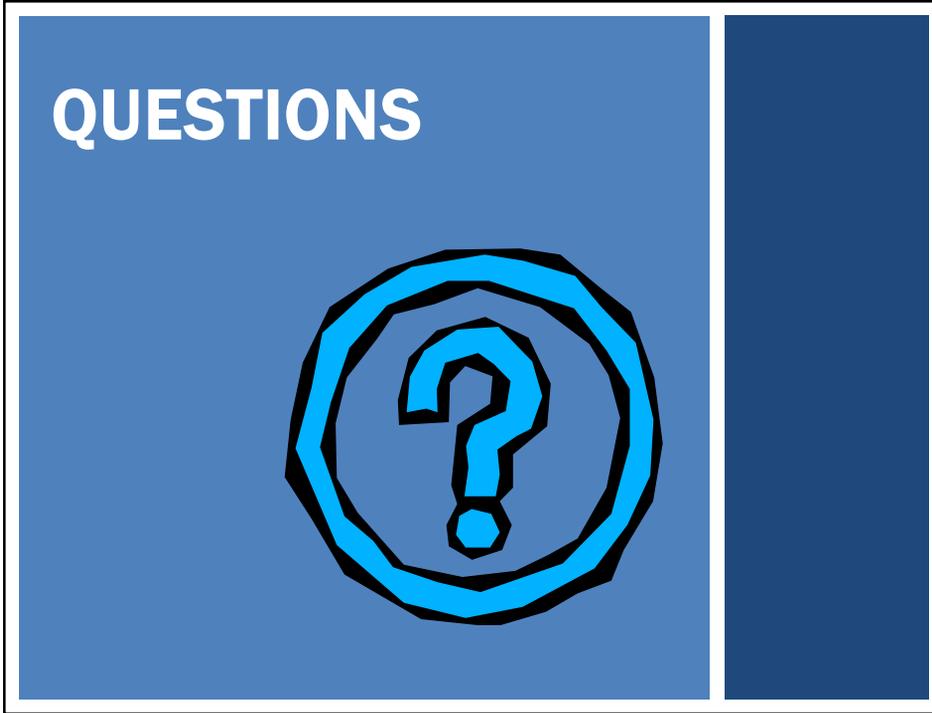
- **brk(), sbrk()**
 - Used to change data segment size (the end of the heap)
 - Don't use these

- **Mmap(), munmap()**
 - Can be used to create an extra independent "heap" of memory for a user program

- See man page

February 24, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L13.78
-------------------	---	--------

78



79