


# TCCS 422: OPERATING SYSTEMS

## Condition Variables, Concurrency Problems

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



May 13, 2021 TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma

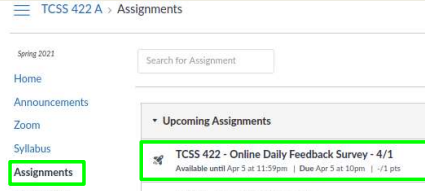
## OBJECTIVES – 5/13

- **Questions from 5/11**
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021 TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L13.2

## 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



May 13, 2021 TCCS422: Computer Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L13.3

### TCCS 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

May 13, 2021 TCCS422: Computer Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L13.4

## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (47 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 7.77 (↑ - previous 5.99)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.81 (↑ - previous 5.36)**

May 13, 2021 TCCS422: Computer Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L13.5

## FEEDBACK

- ***Could hand over hand locking strategy cause there to be a traffic jam of sorts? Where each thread has to wait for the one that's at the beginning since it seems like it wouldn't be able to skip ahead of a thread that was taking longer than needed.***
- "Thread" traffic with hand-over-hand locking should be less than with a single lock for the entire list
- A new thread could start traversing the list at each iteration as soon as another thread iterates to the next item
- For a list of **N** items, **N** threads could be traversing the list simultaneously
- Drawback is many calls to lock/unlock APIs

May 13, 2021 TCCS422: Operating Systems [Spring 2021]  
 School of Engineering and Technology, University of Washington - Tacoma L13.6

### OBJECTIVES – 5/13

- Questions from 5/11
- **Assignment 2**
- Quiz 3 – Synchronized Array
- 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

May 13, 2021 TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma L13.7

### OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- **Quiz 3 – Synchronized Array**
- 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

May 13, 2021 TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma L13.8

### Vote For Posting: Tutorial 2: Pthread Tutorial: Parallel Prime Number Generation with W pthread\_t, pthread\_mutex\_t, pthread\_mutex\_cond (two lowest quiz grades automatically dropped)

Yes, practice would be helpful before/with Assignment 2 **A**

No, I am swamped **B**


Either way - Yes or No is Fine **C**

May 13, 2021 TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma L13.9

### OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021 TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma L13.10



## CHAPTER 30 – CONDITION VARIABLES

May 13, 2021 TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma L13.11

## MATRIX GENERATOR

Matrix generation example

Chapter 30  
signal.c

May 13, 2021 TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma L13.12

### MATRIX GENERATOR

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.13

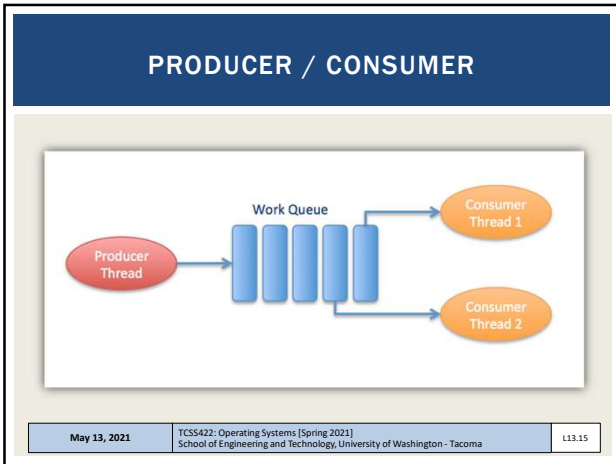
### ATTEMPT TO USE CONDITION VARIABLE WITHOUT A WHILE STATEMENT

```

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

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.14



### PRODUCER / CONSUMER

- **Producer**
  - Produces items – e.g. child the makes matrices
  - Places them in a buffer
    - Example: the buffer size is only 1 element (single array pointer)
- **Consumer**
  - Grabs data out of the buffer
  - Our example: parent thread receives dynamically generated matrices and performs an operation on them
    - Example: calculates average value of every element (integer)
- Multithreaded web server example
  - Http requests placed into work queue; threads process

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.16

### PRODUCER / CONSUMER - 2

- Producer / Consumer is also known as **Bounded Buffer**
- Bounded buffer
  - Similar to piping output from one Linux process to another
  - `grep pthread signal.c | wc -l`
  - Synchronized access:
    - sends output from `grep` → `wc` as it is produced
  - File stream

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.17

### PUT/GET ROUTINES

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

```

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.18

### 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(i);
13        printf("%d\n", tmp);
14    }
15 }
    
```

Producer

Consumer

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.19

### 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++) { // c1
19        // ...
    }
    }
    
```

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.20

### PRODUCER/CONSUMER - 4

```

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

Consumer

- This code as-is works with just:
  - (1) Producer
  - (1) Consumer
- PROBLEM:** no while. If thread wakes up it **MUST** execute
- If we scale to (2+) consumer's it fails
  - How can it be fixed ?

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.21

### EXECUTION TRACE: ISSUE: NO WHILE

1 PRODUCER, 2 CONSUMERS, 1 CONDITION

- Two threads

$T_{c1}$	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

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

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.22

### PRODUCER/CONSUMER SYNCHRONIZATION

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

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.23

### EXECUTION TRACE: ISSUE: 1 CONDITION

WHILE, 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- get()  
 p4- put()  
 c5/p5- signal  
 c6/p6- unlock

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.24

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

- T<sub>c2</sub> runs, no data to consume

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

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	State	Count	Comment
...	...	...	...	...	...	...	(cont.)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	→ c2	Running		Sleep	0	
	Sleep	→ c3	Sleep		Sleep	0	Everyone asleep ...

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.25

### TWO CONDITIONS

- Required w/ multiple producer and consumer threads
- Use two condition variables: **empty & full**
  - One condition handles the producer
  - the other the consumer

```

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.26

### FINAL PRODUCER/CONSUMER

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

```

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

```

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.27

### 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)
9              pthread_cond_wait(&empty, &mutex); // p3
10         put(i);
11         pthread_cond_signal(&full); // p4
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)
21             pthread_cond_wait(&full, &mutex); // c3
22         int tmp = get(); // c4

```

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.28

### FINAL P/C - 3

```

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

```

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.29

### 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 poller.com/app

**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 [polliv.com/app](http://polliv.com/app)

**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 [polliv.com/app](http://polliv.com/app)

### OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.33

### COVERING CONDITIONS

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

**PREVENT:** Out of memory:  
 - queue requests until memory is free

- Which thread should be woken up?

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.34

### 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); // Broadcast
23     pthread_mutex_unlock(&m);
24 }
    
```

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.35

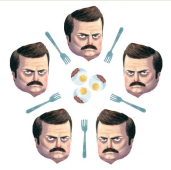
### COVER CONDITIONS - 3

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

May 13, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L13.36

## CHAPTER 31: SEMAPHORES

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



May 13, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.37

## WE WILL RETURN AT 5:14PM




May 13, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.38

## OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.39

## CHAPTER 32 – CONCURRENCY PROBLEMS



May 13, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.40

## 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
<b>Total</b>		<b>74</b>	<b>31</b>

May 13, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.41

## OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021
TCSS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.42

### NON-DEADLOCK BUGS

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.43

### ATOMICITY VIOLATION - MYSQL

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

Programmer intended variable to be accessed atomically... →

```

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.44

### ATOMICITY VIOLATION - SOLUTION

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

```

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.45

### ORDER VIOLATION BUGS

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

```

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

- What if `mThread` is not initialized?

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.46

### 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 mInit = 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  mInit = 1;
13  pthread_cond_signal(&mtCond);
14  pthread_mutex_unlock(&mtLock);
15  ...
16 }
17
18 Thread2::
19 void mMain(){
20  ...
            
```

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.47

### 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(mInit == 0)
24   pthread_cond_wait(&mtCond, &mtLock);
25 pthread_mutex_unlock(&mtLock);
26
27 mState = mThread->State;
28 ...
29 }
            
```

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.48



### NON-DEADLOCK BUGS - 1

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.49

### NON-DEADLOCK BUGS - 2

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.50

### 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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.51

### OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.52

### REASONS FOR DEADLOCKS

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

```

1 Vector v1, v2;
2 v1.AddAll(v2);
    
```

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.53

### CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.54

### OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021    TCCS422: Operating Systems [Spring 2021]    L13.55  
 School of Engineering and Technology, University of Washington - Tacoma

### PREVENTION – MUTUAL EXCLUSION

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

```

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

May 13, 2021    TCCS422: Operating Systems [Spring 2021]    L13.56  
 School of Engineering and Technology, University of Washington - Tacoma

### PREVENTION – MUTUAL EXCLUSION - 2

- Recall atomic increment

```

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

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

May 13, 2021    TCCS422: Operating Systems [Spring 2021]    L13.57  
 School of Engineering and Technology, University of Washington - Tacoma

### MUTUAL EXCLUSION: LIST INSERTION

- Consider list insertion

```

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

May 13, 2021    TCCS422: Operating Systems [Spring 2021]    L13.58  
 School of Engineering and Technology, University of Washington - Tacoma

### MUTUAL EXCLUSION – LIST INSERTION - 2

- Lock based implementation

```

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

May 13, 2021    TCCS422: Operating Systems [Spring 2021]    L13.59  
 School of Engineering and Technology, University of Washington - Tacoma

### MUTUAL EXCLUSION – LIST INSERTION - 3

- Wait free (no lock) implementation

```

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

- Assign &head to n (new node ptr)
- Only when head = n->next

May 13, 2021    TCCS422: Operating Systems [Spring 2021]    L13.60  
 School of Engineering and Technology, University of Washington - Tacoma

### CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.61

### PREVENTION LOCK – HOLD AND WAIT

- Problem: acquire all locks atomically
- Solution: use a “lock” “lock”... (like a *guard lock*)

```

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

- Effective solution – guarantees no race conditions while acquiring L1, L2, etc.
- Order doesn't matter for L1, L2
- Prevention (GLOBAL) lock decreases concurrency of code
  - Acts Lowers lock granularity
- Encapsulation: consider the Java Vector class...

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.62

### CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.63

### PREVENTION – NO PREEMPTION

- When acquiring locks, don't BLOCK forever if unavailable...
- pthread\_mutex\_trylock() - try once
- pthread\_mutex\_timedlock() - try and wait awhile

```

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

- Eliminates deadlocks

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.64

### NO PREEMPTION – LIVELOCKS PROBLEM

- Can lead to livelock

```

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

- Two threads execute code in parallel → always fail to obtain both locks
- Fix: add random delay
  - Allows one thread to win the livelock race!

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.65

### CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.66

### PREVENTION – CIRCULAR WAIT

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.67

### CONDITIONS FOR DEADLOCK

■ **If any of the following conditions DOES NOT EXIST, describe why deadlock can not occur?**

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.68

### 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 [polliv.com/app](http://polliv.com/app)

### 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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.70

### INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:

CPU 1	T3	T4
CPU 2	T1	T2

- No deadlock can occur
- Consider:

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

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.71

### INTELLIGENT SCHEDULING - 3

- Scheduler produces schedule

CPU 1	T4	
CPU 2	T1	T2
		T3

- Scheduler must be conservative and not take risks
- Slows down execution – many threads
- There has been limited use of these approaches given the difficulty having intimate lock knowledge about every thread

May 13, 2021
TCCS422: Operating Systems [Spring 2021]  
School of Engineering and Technology, University of Washington - Tacoma
L13.72

## DETECT AND RECOVER

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


May 13, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L13.73
--------------	---	--------

## OBJECTIVES – 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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

May 13, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L13.74
--------------	---	--------

# CHAPTER 13: ADDRESS SPACES



May 13, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L13.75
--------------	---	--------

## OBJECTIVES – 5/13

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

May 13, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L13.76
--------------	---	--------

## 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

May 13, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L13.77
--------------	---	--------


## 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**



May 13, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L13.78
--------------	---	--------

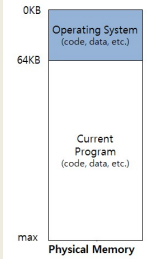
### MOTIVATION FOR MEMORY VIRTUALIZATION

- 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)

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.79

### EARLY MEMORY MANAGEMENT

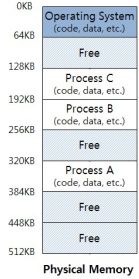
- Load one process at a time into memory
- Poor memory utilization
- Little abstraction



May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.80

### MULTIPROGRAMMING WITH SHARED MEMORY

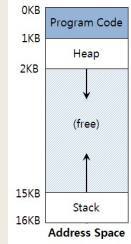
- Later machines supported running multiple processes
- Swap out processes during I/O waits to increase system utilization and efficiency
- Swap entire memory of a process to disk for context switch
- Too slow, especially for large processes
- Solution →
  - Leave processes in memory
- Need to protect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



May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.81

### ADDRESS SPACE

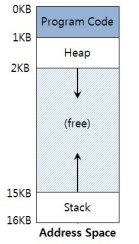
- Easy-to-use abstraction of physical memory for a process
- Main elements:
  - Program code
  - Stack
  - Heap
- Example: 16KB address space



May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.82

### ADDRESS SPACE - 2

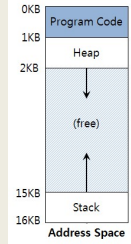
- Code
  - Program code
- Stack
  - Program counter (PC)
  - Local variables
  - Parameter variables
  - Return values (for functions)
- Heap
  - Dynamic storage
  - Malloc() new()



May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.83

### 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



May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.84

## 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

May 13, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L13.85

## VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:
  - location of code: 0x400686
  - location of heap: 0x1129420
  - location of stack: 0x7ffe040d77e4

May 13, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L13.86

## 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)

May 13, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L13.87

## 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

May 13, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L13.88

## OBJECTIVES - 5/13

- Questions from 5/11
- Assignment 2
- Quiz 3 – Synchronized Array
- 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**

May 13, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L13.89

## CHAPTER 14: THE MEMORY API

May 13, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L13.90

## OBJECTIVES – 5/12

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

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.91

## 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

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.92

## sizeof()

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

<pre>int *x = malloc(10 * sizeof(int)); printf("%d\n", sizeof(x));</pre>	4
<pre>int x[10]; printf("%d\n", sizeof(x));</pre>	40

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.93

## FREE()

```

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

- Free memory allocated with malloc()
- Provide: (void \*) ptr to malloc'd memory

- Returns: nothing

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.94

```

#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;
}
    
```

What will this code do?

95

```

#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;
}
    
```

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?**

96



### 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).

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.97

### 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”

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.98

### 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
```

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.99

### 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

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.100

### 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

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.101

### 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

May 13, 2021    TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma    L13.102

