


# TCSS 422: OPERATING SYSTEMS

**Condition Variables,  
Concurrency Problems,  
Address Spaces**

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



November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

## FEEDBACK FROM 11/7

- How can you display the thread ID for debugging purposes?
- Can associate an unique int when creating pthread

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.2

## FEEDBACK - 2

- Do you need multiple condition variables (*in program 2*) or can you use just one?
- Coordinating the end of program 2
  - Max LOOPS is reached
  - Consumer threads: run out of matrices

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.3

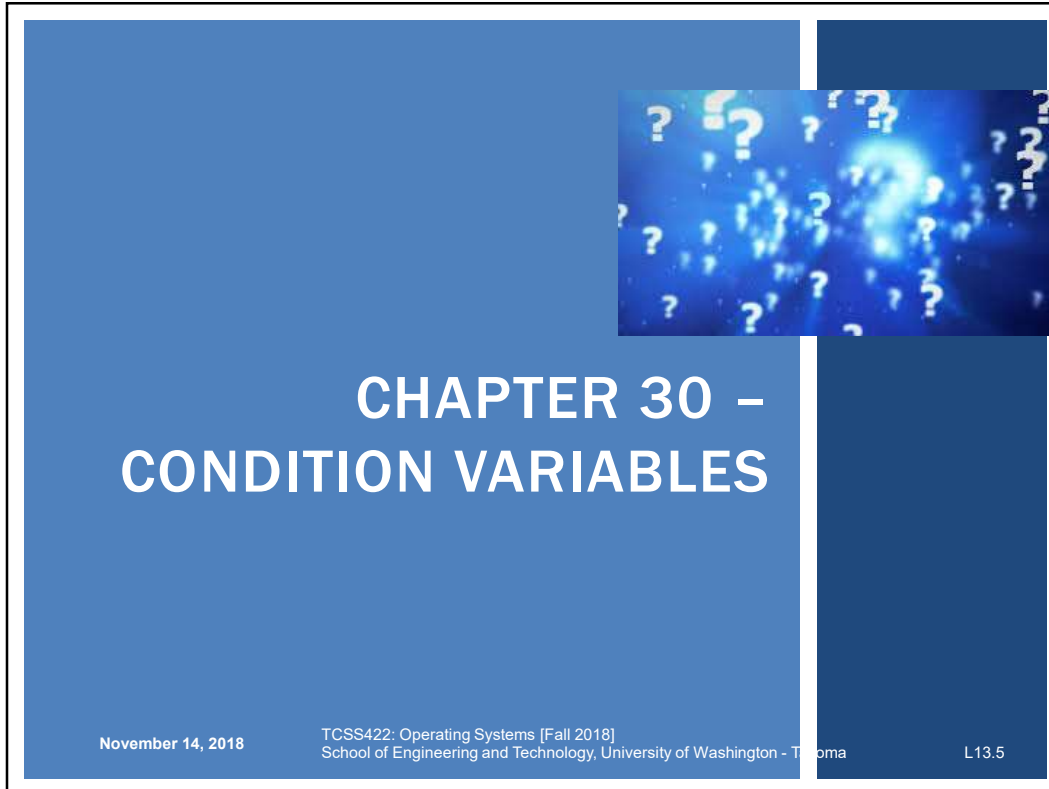
## OBJECTIVES

- Quiz 3 – Synchronized Array
- Multi-threaded Programming
  - Chapter 30 – Condition Variables
  - Chapter 32 – Concurrency Problems
- Memory Virtualization
  - Chapters 13, 14, 15, 16....

November 14, 2018

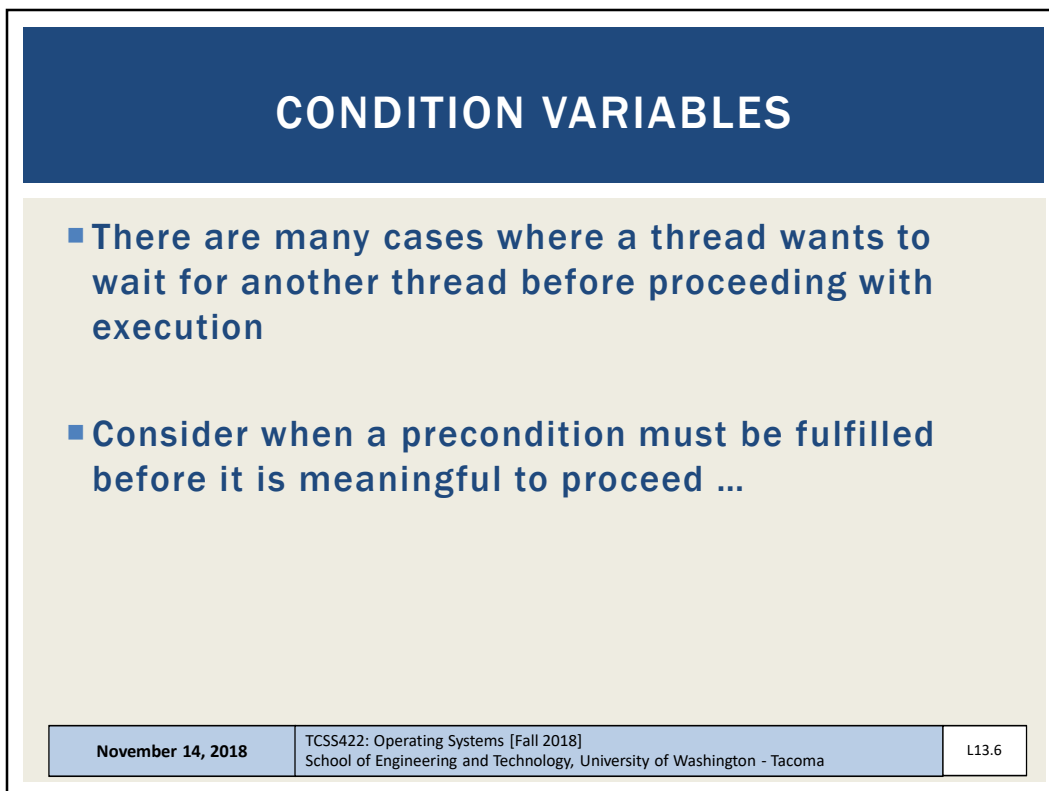
TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.4



# CHAPTER 30 – CONDITION VARIABLES

November 14, 2018 TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma L13.5



## 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 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.6
-------------------	---	-------

## 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 an **explicit queue (FIFO)** to wait for signals
- **Signal**: wakes one thread  
**broadcast** wakes all (ordering by the OS)

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.7

## 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
- signal()
  - Wakes up thread, awakening thread acquires lock

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.8

## CONDITION VARIABLES - QUESTIONS

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

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.9

## MATRIX GENERATOR

Matrix generation example

Chapter 30  
signal.c

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.10

## MATRIX GENERATOR

- The main thread, and worker thread (generates matrices) share a single matrix pointer.
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Let's try "nosignal.c"

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.11

## SUBTLE RACE CONDITION: WITHOUT A WHILE

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

- Parent thread calls thr\_join() and executes the comparison
- The context switches to the child
- The child runs thr\_exit() and signals the parent, but the parent is not waiting yet.
- The signal is lost
- The parent deadlocks

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.12

## PRODUCER / CONSUMER

November 14, 2018    TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma    L13.13

## PRODUCER / CONSUMER

- **Producer**
  - Produces items – consider the child matrix maker
  - Places them in a buffer
    - Example: the buffer 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 14, 2018    TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma    L13.14

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.15

## PUT/GET ROUTINES

- Buffer is a one element shared data structure (int)
- Producer “puts” data
- Consumer “gets” data
- 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.16



## PRODUCER / CONSUMER - 3

- Producer adds data
- Consumer removes data (busy waiting)
- Will this code work (spin locks) with 2-threads?
  1. Producer
  2. Consumer

```
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 14, 2018	TCCS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.17
-------------------	---	--------

## 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 14, 2018	TCCS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.18
-------------------	---	--------

## 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
  
- If we scale to (2+) consumer's it fails
  - How can it be fixed ?

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.19

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

	$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
■ Two threads	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- put()  
 p4- get()  
 c5/p5- signal  
 c6/p6- unlock

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.20

## PRODUCER/CONSUMER SYNCHRONIZATION

- When producer threads awake, they do not check if there is any data in the buffer...
  - Need while, not if
  
- 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 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.21
-------------------	---	--------

## 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	<b>Oops! Woke <math>T_{c2}</math></b>

**Legend**

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

November 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.22
-------------------	---	--------

## EXECUTION TRACE – 2

### WHILE, 1 CONDITION, 1 PRODUCER, 2 CONSUMERS

- $T_{c2}$  runs, no data to consume

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

$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
...	...	...	...	...	...	...	(cont.)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
		→ c2	Running		Sleep	0	
		→ c3	Sleep		Sleep	0	Everyone asleep ...

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.23

## TWO CONDITIONS

- 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.24

## FINAL PRODUCER/CONSUMER

- Change buffer from int, to int buffer[MAX]
- Add indexing variables

```
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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.25

## FINAL P/C - 2

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

```

November 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.26

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.27

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.28

## 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 }
```

Check available memory

Broadcast

November 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.29


## 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!
- Overhead
  - Many threads may be awoken which can't execute

November 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.30



# CHAPTER 32 – CONCURRENCY PROBLEMS

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.31

## OBJECTIVES

- Chapter 32:
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.32



## 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>

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.33

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.34

## ATOMICITY VIOLATION - MYSQL

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

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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.35

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.36

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.37

## ORDER VIOLATION - SOLUTION

- Use condition variable 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.38

## ORDER VIOLATION – SOLUTION 2

```
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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.39

## NON-DEADLOCK BUGS - 1

- 97% of Non-Deadlock Bugs were
  - Atomicity
  - Order violations
- Consider what is involved in “spotting” these bugs in code
- Desire for automated tool support (IDE)

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.40

## 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?
  - Before threads are created, after threads exit
  - Must verify the scope

### ■ Order violation

- Must consider all variable accesses
- Must know desired order

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.41

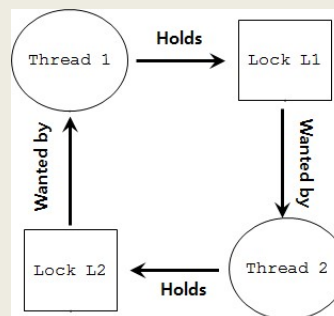
## DEADLOCK BUGS



- 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.42

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.43

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.44

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.45

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.46

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.47

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.48



## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.49

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.50

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.51

## 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 14, 2018

TCCS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.52

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.53

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.54

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.55

## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.56

## 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 14, 2018
TCSS422: Operating Systems [Fall 2018]  
 School of Engineering and Technology, University of Washington - Tacoma
L13.57

## INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:

CPU 1	T3	T4
CPU 2	T1	T2

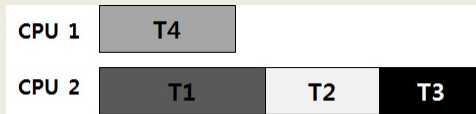
- No deadlock can occur
- Consider:

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

November 14, 2018
TCSS422: Operating Systems [Fall 2018]  
 School of Engineering and Technology, University of Washington - Tacoma
L13.58

## INTELLIGENT SCHEDULING - 3

- Scheduler produces schedule



- 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.59


## 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 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.60



# CHAPTER 13: ADDRESS SPACES

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.61

## OBJECTIVES – MEMORY VIRTUALIATION

- Chapter 13
  - Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14
  - Memory API
  - Common memory errors
- Chapter 15
  - Address translation
  - Base and bounds
  - HW and OS Support
- Chapter 16
  - Memory segments, fragmentation

November 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.62
-------------------	---	--------

## 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

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.63

## 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**



November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.64



## 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)

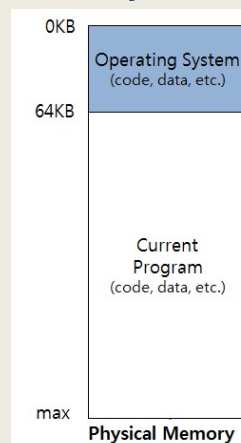
November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.65

## EARLY MEMORY MANAGEMENT

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



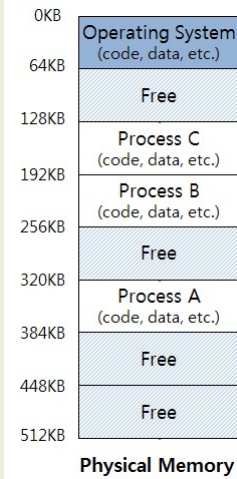
November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.66

## 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



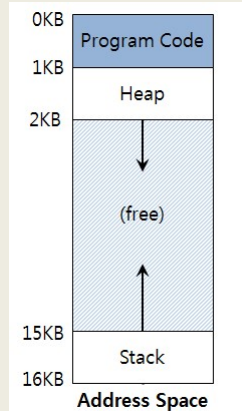
November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.67

## ADDRESS SPACE

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



November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.68

## ADDRESS SPACE - 2

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

0KB  
1KB  
2KB  
15KB  
16KB  
Address Space

November 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.69
-------------------	---	--------

## 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

0KB  
1KB  
2KB  
15KB  
16KB  
Address Space

November 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.70
-------------------	---	--------

# 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**

November 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.71
-------------------	---	--------

# 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. It is divided into several segments: Code (Text) at 0x400000, Data at 0x401000, Heap at 0xcf2000, a free region between 0xd13000 and 0x7fff9ca28000, stack at 0x7fff9ca28000, and Stack at 0x7fff9ca49000. Arrows indicate the direction of growth: the heap grows downwards from 0xcf2000, and the stack grows upwards from 0x7fff9ca49000.

Address Space

0x400000 Code (Text)

0x401000 Data

0xcf2000 Heap

0xd13000

↓ heap

(free)

↑ stack

0x7fff9ca28000 Stack

0x7fff9ca49000

November 14, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L13.72
-------------------	---	--------

## 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)

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.73

## 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*

November 14, 2018

TCSS422: Operating Systems [Fall 2018]  
School of Engineering and Technology, University of Washington - Tacoma

L13.74

