

TCCS 422: OPERATING SYSTEMS

Concurrency Problems

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma



FEEDBACK - 4/27/2017

- How does using a signal on a condition variable fire off a routine?
- **MANPAGE:** `man pthread_cond_signal`:
- **`pthread_cond_signal`** restarts one of the threads that are waiting on the condition variable. If no threads are waiting, nothing happens.
Waiting threads are tracked in a data structure associated with the condition variable.
If several threads are waiting, exactly one is restarted, but it is not specified which.
- Can you think of a way to coordinate exactly which thread to run when signaling? Hint: use `pthread_cond_broadcast`

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.2

FEEDBACK - 2

- Can you walk us through what a makefile is made of?
How to make one?
What is 'clean'?
- Can we do more in-class tutorials?
With concepts/code related to what we have to do in the assignments?
 - Tutorial #1 posted

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.3

OBJECTIVES

- **Chapter 32:**
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Mock midterm

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.4

CONCURRENCY BUGS IN OPEN SOURCE SOFTWARE

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

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

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.5

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.6

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 Thread2::
8 thd->proc_info = NULL;
```

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.7

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 pthread_mutex_unlock(&lock);
10
11 Thread2::
12 pthread_mutex_lock(&lock);
13 thd->proc_info = NULL;
14 pthread_mutex_unlock(&lock);
```

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.8

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.9

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    ...
```

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.10

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 }
```

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.11

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)

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.12

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 known desired order

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.13

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:
lock (L1);
lock (L2);

Thread 2:
lock (L2);
lock (L1);

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

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

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.14

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.15

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.16

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.17

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.18

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.19

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.20

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.21

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.22

PREVENTION – 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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.23

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.24

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   }
```

NO STOPPING ANY TIME

May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma


L10.25

NO PREEMPTION – LIVENLOCKS 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
- Add random delay
 - Allows one thread to win livelock race!



May 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.26

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.27

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.28

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.29

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 2, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.30

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 2, 2017

TCSS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.31

DETECT AND RECOVER


- Allow deadlock to occasionally occur and then take some action.
 - Example: When OS freezes, reboot...
- How often is this acceptable?
- Many database systems employ deadlock detection and recovery techniques.

May 2, 2017

TCSS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.32

QUESTIONS



May 2, 2017

TCSS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L10.33