

# Concurrency Primitives and Bugs

Soham Chakraborty

15.02.2023

# Outline

Lock-based programming

Lock-free programming

Concurrency bugs

## Example

$X = 0;$

$X = X + 1; \parallel X = X + 1;$

What is the final value(s) of  $X$ ?

## Example

$$X = 0;$$

$$X = X + 1; \parallel X = X + 1;$$

What is the final value(s) of  $X$ ?

- Expected:  $X = 2$ .
- Reality:  $X \in \{1, 2\}$

## Example

$X = 0;$   
 $X = X + 1; \parallel X = X + 1;$

What is the final value(s) of  $X$ ?

- Expected:  $X = 2$ .
- Reality:  $X \in \{1, 2\}$

How do we fix it?

- With locks
- Without locks, other primitives

*Init*

|                               |  |                               |
|-------------------------------|--|-------------------------------|
| <code>lock(<i>m</i>)</code>   |  | <code>lock(<i>m</i>)</code>   |
| <code>critical section</code> |  | <code>critical section</code> |
| <code>unlock(<i>m</i>)</code> |  | <code>unlock(<i>m</i>)</code> |

*m*: lock object

critical section: code block between `lock(m)` and `unlock(m)`

# Properties of Lock and Unlock

**Mutual Exclusion** Critical sections of a lock object do not overlap.

- Safety property

# Properties of Lock and Unlock

**Mutual Exclusion** Critical sections of a lock object do not overlap.

- Safety property

**Deadlock Freedom** If one or multiple processes are trying to enter the critical section, then one process eventually will enter the critical section.

- Each lock is deadlock free  $\Rightarrow$  deadlock freedom



# Properties of Lock and Unlock

**Mutual Exclusion** Critical sections of a lock object do not overlap.

- Safety property

**Deadlock Freedom** If one or multiple processes are trying to enter the critical section, then one process eventually will enter the critical section.

- Each lock is deadlock free  $\Rightarrow$  deadlock freedom

**Starvation Freedom** Every thread that attempts to acquire the lock eventually succeeds.

- Requires fairness
- Starvation freedom implies deadlock freedom

# Properties of Lock and Unlock

**Mutual Exclusion** Critical sections of a lock object do not overlap.

- Safety property

**Deadlock Freedom** If one or multiple processes are trying to enter the critical section, then one process eventually will enter the critical section.

- Each lock is deadlock free  $\Rightarrow$  deadlock freedom

**Starvation Freedom** Every thread that attempts to acquire the lock eventually succeeds.

- Requires fairness
- Starvation freedom implies deadlock freedom

**Waiting** If one thread delays in the critical section then other threads also get delayed

- What if a thread acquires a lock and crashes?
- Requires fault tolerance

## Example of Lock-Unlock Implementation (Two Threads)

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true; // I'm interested  
    while(flag[j] == true) {} // wait loop  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false; // I'm not interested  
}
```

Is the lock implementation correct?

- Requires analysis

# Well-formedness

A thread is well-formed if:

- ① each critical section is associated with a unique lock object.
- ② the thread calls `lock` method for that object when it is trying to enter the critical section, and
- ③ the thread calls the `unlock` method for that object when it leaves the critical section.

Threads are state machines:  $a_0 \rightarrow a_1 \rightarrow \dots$

Thread transitions are events

$(a_0, a_1)$ : Interval between events  $a_0$  and  $a_1$

$I_A = (a_0, a_1)$ : interval between  $a_0$  and  $a_1$  in thread  $A$

$I_B = (b_0, b_1)$ : interval between  $a_0$  and  $a_1$  in thread  $B$

$I_A \rightarrow I_B$ : interval  $I_A$  precedes  $I_B$ ; when  $a_1 \rightarrow b_0$

$a_i^j$ :  $j^{th}$  occurrence of an event  $a_i$

$I_A^j$ :  $j^{th}$  occurrence of an interval  $I_A$

# Mutual Exclusion

Critical sections do not overlap.

Given thread  $A$  and  $B$

Given the intervals  $CS_A^i$  and  $CS_B^j$ :

either  $CS_A^i \rightarrow CS_B^j$  or  $CS_B^j \rightarrow CS_A^i$

## Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    while(flag[j] == true)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

## Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
     $i = tid();$   
     $j = 1 - i;$   
     $flag[i] = true;$   
    while( $flag[j] == true$ )  
        {}  
}
```

$CS_0^j \nrightarrow CS_1^k$  and  $CS_1^k \nrightarrow CS_0^j$

```
unlock() {  
     $i = tid();$   
     $flag[i] = false;$   
}
```



# Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    while(flag[j] == true)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

a:  $W_0(flag[0], true)$



b:  $R_0(flag[1], false)$



$CS_0$

# Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    while(flag[j] == true)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

a:  $W_0(flag[0], true)$

c:  $W_1(flag[1], true)$



b:  $R_0(flag[1], false)$

d:  $R_1(flag[0], false)$



$CS_0$

$CS_1$

# Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    while(flag[j] == true)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

a:  $W_0(flag[0], true)$

c:  $W_1(flag[1], true)$

b:  $R_0(flag[1], false)$

d:  $R_1(flag[0], false)$

$CS_0$

$CS_1$

# Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    while(flag[j] == true)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

a:  $W_0(flag[0], true)$

c:  $W_1(flag[1], true)$

b:  $R_0(flag[1], false)$

d:  $R_1(flag[0], false)$

$CS_0$

$CS_1$

Contradiction:  $a \rightarrow d$  and no intermediate  $W(flag[0], false)$  between  $a$  and  $b$ .

# Violation of Mutual Exclusion: Critical sections overlap!

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    while(flag[j] == true)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

a:  $W_0(flag[0], true)$

c:  $W_1(flag[1], true)$

b:  $R_0(flag[1], false)$

d:  $R_1(flag[0], false)$

$CS_0$

$CS_1$

Contradiction:  $a \rightarrow d$  and no intermediate  $W(flag[0], false)$  between  $a$  and  $b$ .

**Question:** What happens if  
 $flag[i] = true$  statements are executed before the wait loops?

## Alternative Implementation of Lock Unlock

```
void lock(){  
    i = tid();  
    victim = i; // let the other go first  
    while(victim == i) {} // wait  
}
```

```
unlock() {}
```

## Alternative Implementation of Lock Unlock

```
void lock(){  
    i = tid();  
    victim = i; // let the other go first  
    while(victim == i) {} // wait  
}
```

```
unlock() {}
```

Does the implementation ensures mutual exclusion?

# Mutual Exclusion

```
void lock(){  
    i = tid();  
    victim = i;  
    while(victim == i)  
        {}  
}
```

```
unlock() {}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

$W_0(victim, 0)$



$R_0(victim, 1)$



$CS_0$



# Mutual Exclusion

```
void lock(){  
    i = tid();  
    victim = i;  
    while(victim == i)  
        {}  
}
```

```
unlock() {}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

$W_0(victim, 0)$



$R_0(victim, 1)$



$CS_0$

$W_1(victim, 1)$



$R_1(victim, 0)$



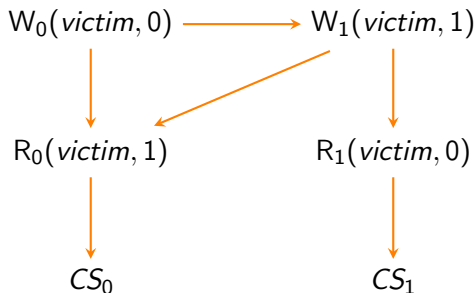
$CS_1$

# Mutual Exclusion

```
void lock(){  
    i = tid();  
    victim = i;  
    while(victim == i)  
        {}  
}
```

```
unlock() {}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

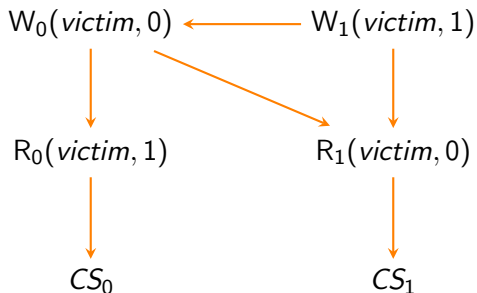


# Mutual Exclusion

```
void lock(){  
    i = tid();  
    victim = i;  
    while(victim == i)  
        {}  
}
```

```
unlock() {}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$

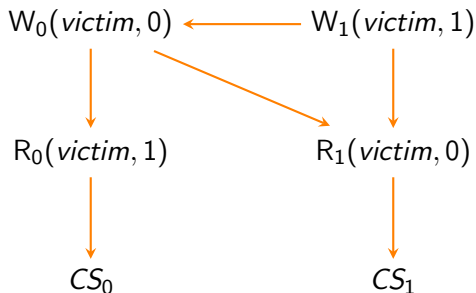


# Mutual Exclusion

```
void lock(){  
    i = tid();  
    victim = i;  
    while(victim == i)  
        {}  
}
```

```
unlock() {}
```

$CS_0^j \not\rightarrow CS_1^k$  and  $CS_1^k \not\rightarrow CS_0^j$



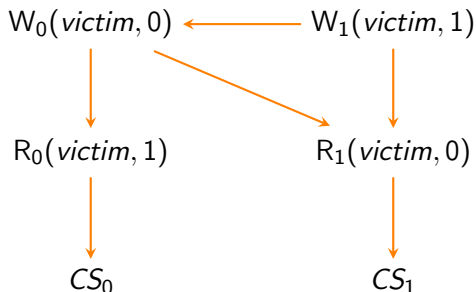
Problem: What if the threads are not running concurrently?

# Mutual Exclusion

```
void lock(){  
    i = tid();  
    victim = i;  
    while(victim == i)  
        {}  
}
```

```
unlock() {}
```

$$CS_0^j \not\rightarrow CS_1^k \text{ and } CS_1^k \not\rightarrow CS_0^j$$



Problem: What if the threads are not running concurrently?

It deadlocks if one thread runs completely before the other.

# Peterson's Lock Algorithm

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  // I am interested  
    victim = i;  // you go first  
    while(flag[j] && victim == i) {}  // I am waiting  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  // I am not interested  
}
```

# Peterson's Lock Algorithm

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    victim = i;  
    while(flag[j] &&  
        victim == i)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

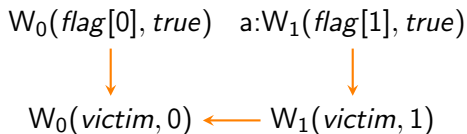
w.l.o.g assume thread 0  
writes to *victim*

# Peterson's Lock Algorithm

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    victim = i;  
    while(flag[j] &&  
          victim == i)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

w.l.o.g assume thread 0  
writes to *victim*



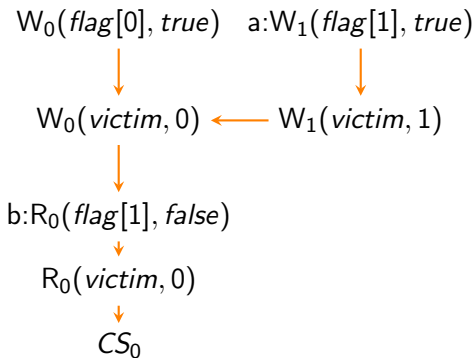


# Peterson's Lock Algorithm

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    victim = i;  
    while(flag[j] &&  
          victim == i)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

w.l.o.g assume thread 0  
writes to *victim*

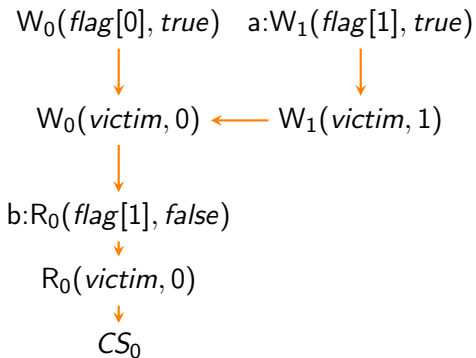


# Peterson's Lock Algorithm

```
void lock(){  
    i = tid();  
    j = 1 - i;  
    flag[i] = true;  
    victim = i;  
    while(flag[j] &&  
          victim == i)  
        {}  
}
```

```
unlock() {  
    i = tid();  
    flag[i] = false;  
}
```

w.l.o.g assume thread 0  
writes to *victim*



no intermediate write on *flag[1]*  
between *a* and *b*

# Use cases: Concurrent Data Structures

Multiple threads may access the data structure concurrently

Examples:

- Linked list
- Stack
- Queue
- ...

Often referred as concurrent objects (data structure+API methods)

Accessed by a set of methods

- LinkedList: `add()`, `search()`, `delete()`
- Queue: `enq()`, `deq()`
- Stack: `push()`, `pop()`

## Coarse-grained locking

- Synchronize every access to the object using a global lock
- Example: Lock the entire linked-list to add/delete a node

## fine-grained locking

- Partition the object into independent synchronized components
- Example: Lock relevant nodes in a linked-list to add/delete a node

## Nonblocking

- No use of lock/unlock
- Use special primitives for atomic update

# Lock Free/Nonblocking Data Structures

Multiple threads may access the object concurrently

Typically uses compare-and-exchange within a loop

```
CAS(X, old, new){  
  if (X  $\neq$  old)  
    return false;  
  X = new;  
  return true;  
}
```

Used in lock implementation

# Queue with Lock

```
Node {int data; Node next; ...}
```

```
Queue {Node head, tail; ...}
```

Enqueue:

```
void enq(int x) {  
    Node e = new Node(x);  
    enqLock.lock();  
    tail.next = e;  
    tail = e;  
    enqLock.unlock();  
}
```

Dequeue:

```
int deq() {  
    int result;  
    deqLock.lock();  
    if (head.next == null)  
        return ERROR;  
    result = head.next.value;  
    head = head.next;  
    deqLock.unlock();  
    return result;  
}
```

**Pros:** No deadlock as each tail and head has separate locks

**Cons:** performance penalty

## Lock Free Queue: Enqueue

```
1. void enq(int value) {  
2.     Node node = new Node(value);  
3.     while (true) {  
4.         Node last = tail;  
5.         Node next = last.next;  
  
6.         if (last == tail) {  
7.             if (next == null) {  
8.                 if (CAS(last.next, next, node)) {  
9.                     CAS(tail, last, node);  
10.                    return;  
11.                }  
12.            } else {  
  
13.                CAS(tail, last, next);  
14.            }  
15.        }  
16.    }  
17. }
```

# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail;
5.         Node next = last.next;

6.         if (last == tail) {
7.             if (next == null) {
8.                 if (CAS(last.next, next, node)) {
9.                     CAS(tail, last, node);
10.                    return;
11.                }
12.            } else {
13.                CAS(tail, last, next);
14.            }
15.        }
16.    }
17. }
```



# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;

6.         if (last == tail) {
7.             if (next == null) {
8.                 if (CAS(last.next, next, node)) {
9.                     CAS(tail, last, node);
10.                    return;
11.                }
12.            } else {
13.                CAS(tail, last, next);
14.            }
15.        }
16.    }
17. }
```

# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;
6.         // identify the position to append the new node
7.         if (last == tail) {
8.             if (next == null) {
9.                 if (CAS(last.next, next, node)) {
10.                    CAS(tail, last, node);
11.                    return;
12.                }
13.            } else {
14.                CAS(tail, last, next);
15.            }
16.        }
17.    }
```

# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;
6.         // identify the position to append the new node
7.         if (last == tail) {
8.             if (next == null) { // no successor
9.                 if (CAS(last.next, next, node)) {
10.                    CAS(tail, last, node);
11.                    return;
12.                }
13.            } else {
14.                CAS(tail, last, next);
15.            }
16.        }
17.    }
```

# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;
6.         // identify the position to append the new node
7.         if (last == tail) {
8.             if (next == null) { // no successor
9.                 if (CAS(last.next, next, node)) { // append the new node
10.                    CAS(tail, last, node);
11.                    return;
12.                }
13.            } else {
14.                CAS(tail, last, next);
15.            }
16.        }
17.    }
```

## Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;
6.         // identify the position to append the new node
7.         if (last == tail) {
8.             if (next == null) { // no successor
9.                 if (CAS(last.next, next, node)) { // append the new node
10.                    CAS(tail, last, node); // new node is the tail
11.                    return;
12.                }
13.            } else {
14.                CAS(tail, last, next);
15.            }
16.        }
17.    }
```

# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;
6.         // identify the position to append the new node
7.         if (last == tail) {
8.             if (next == null) { // no successor
9.                 if (CAS(last.next, next, node)) { // append the new node
10.                    CAS(tail, last, node); // new node is the tail
11.                    return;
12.                }
13.            } else {
14.                // tail has a successor; another thread is in between 8–9
15.                CAS(tail, last, next);
16.            }
17.        }
```

# Lock Free Queue: Enqueue

```
1. void enq(int value) {
2.     Node node = new Node(value); // create a new node
3.     while (true) {
4.         Node last = tail; // locate the last node
5.         Node next = last.next;
6.         // identify the position to append the new node
7.         if (last == tail) {
8.             if (next == null) { // no successor
9.                 if (CAS(last.next, next, node)) { // append the new node
10.                    CAS(tail, last, node); // new node is the tail
11.                    return;
12.                }
13.            } else {
14.                // tail has a successor; another thread is in between 8–9
15.                CAS(tail, last, next); // set tail to correct node
16.            }
17.        }
```

Lock-free algorithms are (usually) faster.

Subtle details

- Liveness
- Termination
- Shared memory reclamation

Difficult to reason about various properties



# Concurrency Bugs

Order violation

Atomicity violation

Deadlock

Data race

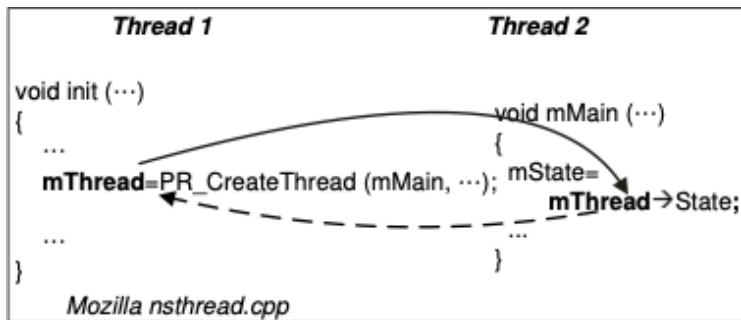
# Order Violation

Cause: Programmer assumes certain ordering of events

# Order Violation

Cause: Programmer assumes certain ordering of events

Example:

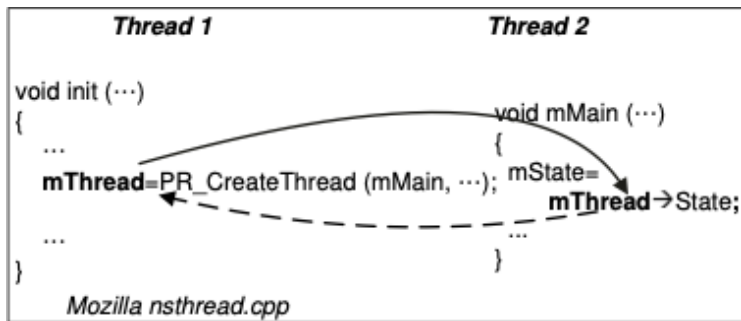


Thread 2 should not deref. **mThread** before Thread 1 initializes it

# Order Violation

Cause: Programmer assumes certain ordering of events

Example:



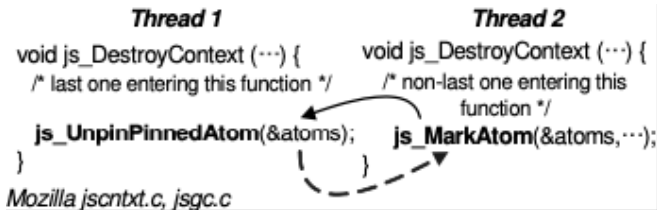
Thread 2 should not deref. **mThread** before Thread 1 initializes it  
Pattern:

```
X = 0;  
X = 1; || t = X; // 1
```

# Order Violation

Cause: Programmer assumes certain ordering of (W,R) events

Example:

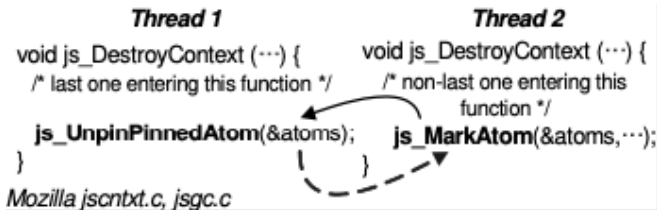


`js_UnpinPinnedAtom` should happen after `js_MarkAtom`.

# Order Violation

Cause: Programmer assumes certain ordering of (W,R) events

Example:



`js_UnpinPinnedAtom` should happen after `js_MarkAtom`.

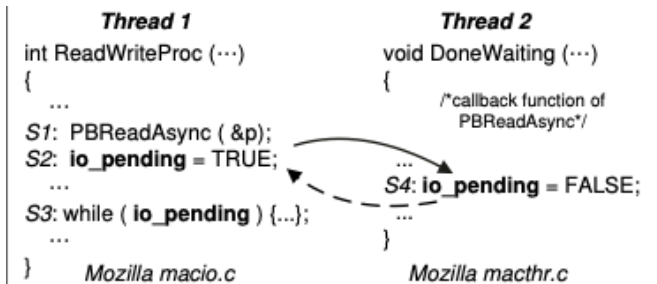
Pattern:

```
X = 0;
X = 1; || t = X; // 0
```

# Order Violation

Cause: Programmer assumes certain ordering of (W,W) events

Example:



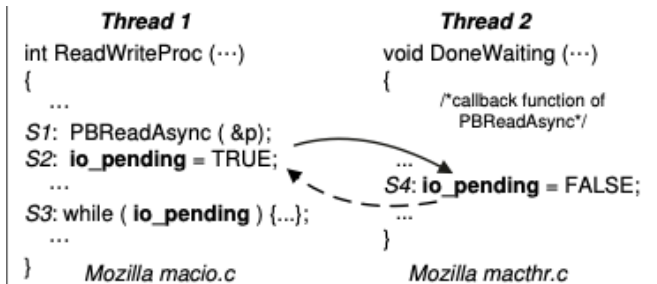
Assumption: S1 and S2 execute atomically

Unsafe ordering blocks thread 1

# Order Violation

Cause: Programmer assumes certain ordering of (W,W) events

Example:



Assumption: S1 and S2 execute atomically

Unsafe ordering blocks thread 1

Pattern:

```

X = 1;
while(X == 1) ; // 0 || X = 0;

```

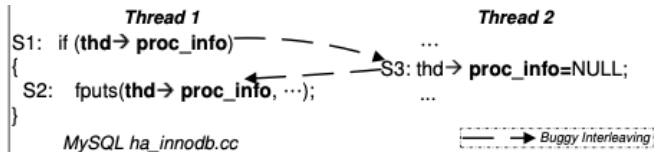




# Atomicity Violation

Cause: Programmer assumes atomicity of certain code regions

Example:



Assumption: S1;S2 are executed atomically

S2 access NULL value

Pattern:

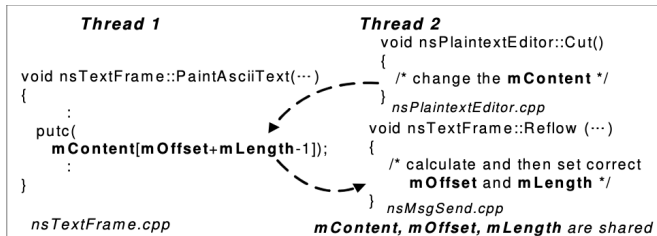
$$\begin{array}{l} X = 0; \\ a = X; \parallel \\ b = X; \parallel \end{array} \begin{array}{l} X = 1; \end{array}$$

Desired:  $a = b = 0$  or  $a = b = 1$

# Multi-Variable Atomicity Bugs

Cause: variables are semantically connected which is violated

Example:



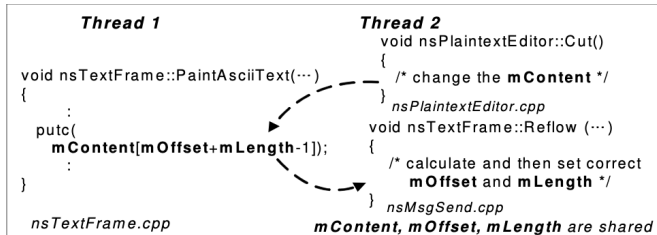
Assumption: *mOffset* and *mLength* are updated atomically wrt thread 1

Lack of synchronization  $\Rightarrow$  thread 1 read inconsistent value

# Multi-Variable Atomicity Bugs

Cause: variables are semantically connected which is violated

Example:



Assumption: *mOffset* and *mLength* are updated atomically wrt thread 1

Lack of synchronization  $\Rightarrow$  thread 1 read inconsistent value

$$Y = Z = 0;$$
$$t = X[Y + Z]; \parallel Y = 1; Z = 1;$$

Desired: access  $X[0]$  or  $X[2]$

# Timing Bugs

Cause: Programmer assumes the tasks would complete within certain time period

Example:

| <i>Thread 1</i>  | <i>Thread 2 ... Thread n</i>   | <i>Monitor thread</i>  |
|--|--------------------------------|--|
| <pre>void buf_flush_try_page() {<br/>    ...<br/>    rw_lock(&amp;lock);<br/>}</pre> | <pre>rw_lock(&amp;lock);</pre> | <pre>void error_monitor_thread() {<br/>    if(<b>lock_wait_time[i] &gt; fatal_timeout</b>)<br/>        assert(0, "We crash the server;<br/>        It seems to be hung.");<br/>}</pre> |
| <i>MySQL buf0flu.c</i>   |                                | <i>MySQL srv0srv.c</i>   |

Assumption:  $n$  tasks would complete before *fatal\_timeout*

Crash the server

# Fix Strategies

Understand the semantics

Add/modify locks

Add/modify synchronizations

Revisit the examples

# Deadlock

A thread holds a lock and wait for another lock held by another thread and vice versa

|                               |                               |
|-------------------------------|-------------------------------|
| <i>lock(m<sub>1</sub>);</i>   | <i>lock(m<sub>2</sub>);</i>   |
| <i>lock(m<sub>2</sub>);</i>   | <i>lock(m<sub>1</sub>);</i>   |
| <i>...</i>                    | <i>...</i>                    |
| <i>unlock(m<sub>2</sub>);</i> | <i>unlock(m<sub>1</sub>);</i> |
| <i>unlock(m<sub>1</sub>);</i> | <i>unlock(m<sub>2</sub>);</i> |

## Deadlock: Another Scenario

Another challenge: encapsulation

```
Vector v1, v2;  
v1.AddAll(v2); || v2.AddAll(v1);
```



# Conditions for Deadlock

All conditions must hold:

- **Mutual exclusion:** Threads claim exclusive control of resources (e.g. lock) that they require.
- **Hold-and-wait:** Threads hold allocated resources while waiting for additional resources
- **No preemption:** Held resources cannot be forcibly removed from threads
- **Circular wait:** There exists a circular chain of threads where each thread holds a resource that are being requested by the next thread in the chain.

**Prevent circular wait** Total ordering on acquiring lock

- Prone to mistakes
- Abstraction makes it difficult

**Prevent hold-and-wait** Acquire all locks at once

- Decreases concurrency significantly

**Prevent no-preemption**

- Problem: Livelock

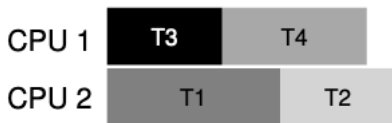
**No mutual-exclusion**

- Lock free programming

# Deadlock Avoidance

Schedule threads in order that access same resources

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



# Deadlock Recovery

Deadlock detector automatically detect deadlock

If deadlock is detected; restart system

# Data Race

Event  $a$  and  $b$  is in data race if:

- $a$  and  $b$  are concurrent/in conflict
- $a$  and  $b$  access same location
- At least one of  $a$  and  $b$  is a write

# C/C++ Concurrency Primitives

Introduced in 2011 C/C++ standard.

Provides platform independent abstraction

Consistency rules.

# Shared Memory Accesses

Non-atomic accesses: Read (Ld), Write (St)

Atomic accesses = operation + memory order

Operations:

- Read (Ld)
- Write (St)
- Atomic update (U)
- Fence (F)

Memory orders:

- Relaxed (rlx)
- Release (rel)
- Acquire (acq)
- Acquire-Release (acq\_rel)
- Sequentially consistent (sc)

# Shared Memory Accesses

Non-atomic accesses: Read (Ld), Write (St)

Atomic accesses = operation + memory order

Operations:

- Read (Ld)
- Write (St)
- Atomic update (U)
- Fence (F)

Memory orders:

- Relaxed (rlx)
- Release (rel)
- Acquire (acq)
- Acquire-Release (acq\_rel)
- Sequentially consistent (sc)

Example:

- X.load(memory order)
- X.store(val, memory order)
- X.CAS(oldval, nwval, success mem order, failure mem order)
- atomic\_thread\_fence(memory order)



# Access Types

Read.  $t = X_o$   
where  $o \in \{\text{na}, \text{rlx}, \text{acq}, \text{sc}\}$

Write.  $X_o = v$   
where  $o \in \{\text{na}, \text{rlx}, \text{acq}, \text{sc}\}$

Update.  $\text{CAS}(X, v, v', o_s, o_f)$   
where  $o_s, o_f \in \{\text{rlx}, \text{rel}, \text{acq}, \text{acq\_rel}, \text{U}_{\text{sc}}\}$

Fence  $F_o$   
where  $o \in \{\text{rel}, \text{acq}, \text{acq\_rel}, \text{sc}\}$

For now we consider only sc accesses

# Reordering Rules

| $a(\ell) \Downarrow / b(\ell') \Rightarrow$   | $\text{Ld}_{\text{na}}/\text{St}_{\text{na}}$ | $\text{St}_{\text{sc}}$ | $\text{Ld}_{\text{sc}}$ |
|---|---|-------------------------|-------------------------|
| $\text{Ld}_{\text{na}}/\text{St}_{\text{na}}$ | ✓   | ✗                       | ✓                       |
| $\text{St}_{\text{sc}}$                       | ✓   | ✗                       | ✗                       |
| $\text{Ld}_{\text{sc}}$                       | ✗   | ✗                       | ✗                       |

$a; b \rightsquigarrow b; a$  where  $\ell \neq \ell'$  and are independent

```
X = NULL, flag = 0;  
  
X = new Obj(); || while(flag ≠ 1) // waiting....  
flag = 1;      || ;  
              || t = X; // must be non-NULL
```

What happens if we reorder the statements in the first thread?

# Thread Communication & Synchronization

$X = NULL, flag = 0;$

$X = NULL, Y = 0;$

|                                  |             |                                       |            |                                  |             |                                       |
|----------------------------------|-------------|---------------------------------------|------------|----------------------------------|-------------|---------------------------------------|
| $X = new\ Obj();$<br>$flag = 1;$ | $\parallel$ | $while(flag \neq 1)$<br>;<br>$t = X;$ | $\leadsto$ | $flag = 1;$<br>$X = new\ Obj();$ | $\parallel$ | $while(flag \neq 1)$<br>;<br>$t = X;$ |
|----------------------------------|-------------|---------------------------------------|------------|----------------------------------|-------------|---------------------------------------|

$t = NULL$  is NOT possible

$t = NULL$  is possible

# Varieties of Data Races

Event  $a$  and  $b$  is in data race if:

- $a$  and  $b$  are concurrent/in conflict
- $a$  and  $b$  access same location
- At least one of  $a$  and  $b$  is a write

**Examples:**  $X = 0$  initially.

$$X_{sc} = 1 \quad \left\| \begin{array}{ll} a = X_{na}; & // 0 \quad - \text{NA-race} \\ b = X_{rlx}; & // 0 \quad - \text{Relaxed-race} \\ c = \text{acq}; & // 0 \quad - \text{RA-race} \end{array} \right.$$

## References

The Art of Multiprocessor Programming (chapter 2, 10)

2nd Edition - September 8, 2020

Authors: Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear

Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics.

Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou  
ASPLOS 2008.

Common Concurrency Problems (chapter 32)

Operating Systems: Three Easy Pieces

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

<https://en.cppreference.com/>