Concurrency Bugs

Soham Chakraborty

16.02.2022

Outline

Order violation

Atomicity violation

Deadlock

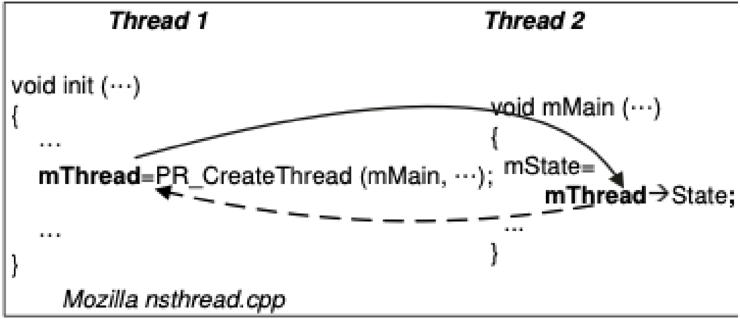
Data race

Order Violation

Cause: Programmer assumes certain ordering of events

Cause: Programmer assumes certain ordering of events

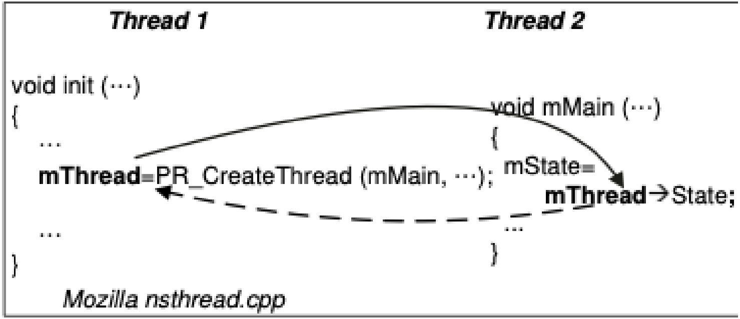
Example:



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

Cause: Programmer assumes certain ordering of events

Example:



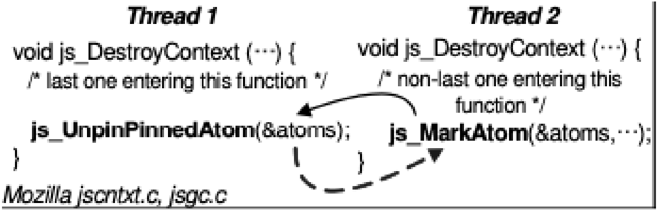
Thread 2 should not deref. **mThread** before Thread 1 initializes it

Pattern:

$$X = 0;$$
$$X = 1; \;\|\; t = X; \;\; // \; 1$$

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

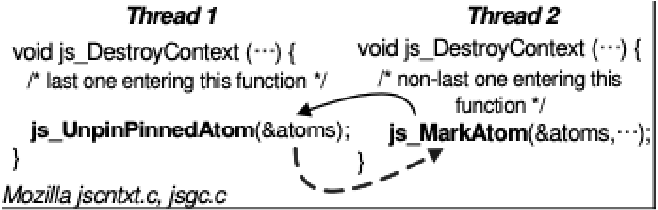
Example:



js_UnpinPinnedAtom should happen after js_MarkAtom.

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

Example:



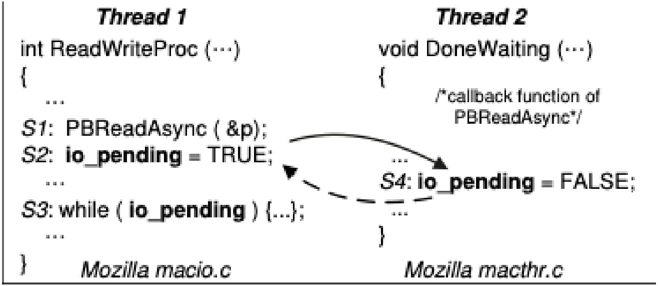
js_UnpinPinnedAtom should happen after js_MarkAtom.

Pattern:

$$X = 0;$$
$$X = 1; \;\Big\|\; t = X; \; \text{// 0}$$

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

Example:

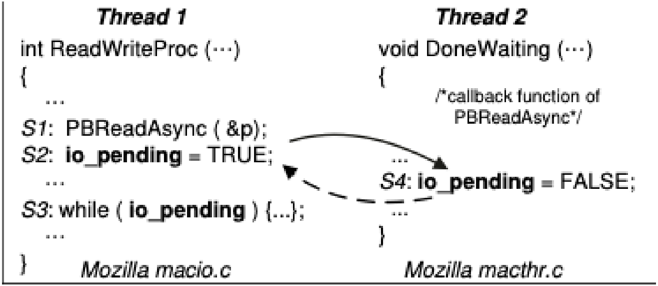


Assumption: S1 and S2 execute atomically

Unsafe ordering blocks thread 1

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

Example:



Assumption: S1 and S2 execute atomically

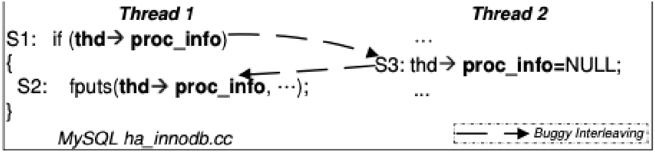
Unsafe ordering blocks thread 1

Pattern:

$$\begin{array}{l} X = 1; \\ \text{while}(X == 1)\ ; \quad // \ 0 \end{array} \ \Bigg\| \ X = 0;$$

Cause: Programmer assumes atomicity of certain code regions

Example:



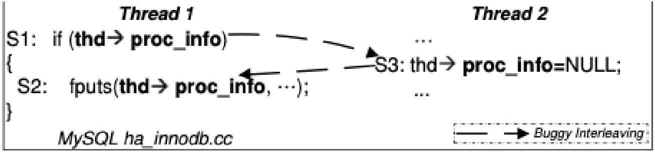
Assumption: S1;S2 are executed atomically

S2 access NULL value

Atomicity Violation

Cause: Programmer assumes atomicity of certain code regions

Example:



Assumption: S1;S2 are executed atomically

S2 access NULL value

Pattern:

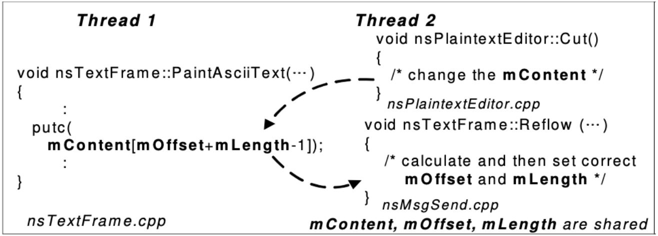
$$X = 0;$$

$$\begin{array}{l} a = X; \\ b = X; \end{array} \;\Bigg\|\; X = 1;$$

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

Multi-Variable Atomicity Bugs

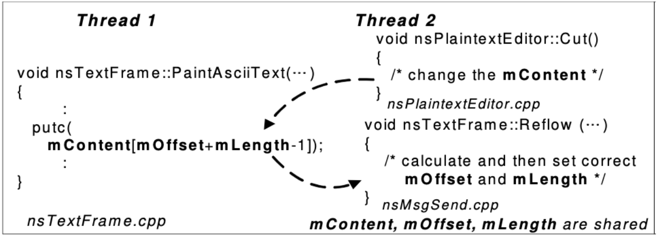
Cause: variables are semantically connected which is violated
Example:



Assumption: *mOffset* and *mLength* are updated atomically wrt thread 1
Lack of synchronization ⇒ 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 ⇒ thread 1 read inconsistent value

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

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

Timing Bugs

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

Example:

Thread 1	***Thread 2 ··· Thread n***	***Monitor thread***
`void buf_flush_try_page() {`		`void error_monitor_thread() {`
`...`		
`rw_lock(&lock);`	`rw_lock(&lock);`	`if(`**`lock_wait_time[i]`**` > `**`fatal_timeout`**`)`
		`assert(0, "We crash the server; It seems to be hung.");`
`}`		`}`
MySQL buf0flu.c		*MySQL srv0srv.c*

Assumption: n taks 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

$$
\begin{array}{l||l}
lock(m_1); & lock(m_2); \\
\quad lock(m_2); & \quad lock(m_1); \\
\ldots & \ldots \\
 & \\
\quad unlock(m_2); & \quad unlock(m_1); \\
unlock(m_1); & unlock(m_2);
\end{array}
$$

Another challenge: encapsulation

$$\begin{gathered} Vector\ v1,\ v2; \\ v1.AddAll(v2); \ \Big\|\ v2.AddAll(v1); \end{gathered}$$

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.

Deadlock Prevention

Prevent circular wait Programming convention: total ordering on acquiring lock

- Prone to mistakes

Prevent hold-and-wait Acquire all locks at once

- Decreases concurrency significantly

Deadlock Prevention

Prevent no-preemption

Hold locks only when all the locks are available

Challenge: encapsulation prevents the 'top' loop implementation

$$
\begin{array}{l}
top: \\
lock(L1); \\
\text{if}(trylock(L2) == -1)\ \{ \\
\quad unlock(L1); \\
\quad goto\ top; \\
\}
\end{array}
$$

Deadlock Prevention

Prevent no-preemption

Hold locks only when all the locks are available

Challenge: encapsulation prevents the 'top' loop implementation

top :	top :
$lock(L1)$;	$lock(L2)$;
if$(trylock(L2) == -1)$ {	if$(trylock(L1) == -1)$ {
$unlock(L1)$;	$unlock(L2)$;
$goto\ top$;	$goto\ top$;
}	}

Deadlock Prevention

Prevent no-preemption

Hold locks only when all the locks are available

Challenge: encapsulation prevents the 'top' loop implementation

Thread 1	Thread 2
$top:$	$top:$
$lock(L1);$	$lock(L2);$
$\text{if}(trylock(L2) == -1)\ \{$	$\text{if}(trylock(L1) == -1)\ \{$
$\quad unlock(L1);$	$\quad unlock(L2);$
$\quad goto\ top;$	$\quad goto\ top;$
$\}$	$\}$

Problem: Livelock

Deadlock Prevention

Prevent circular wait Total ordering on acquiring lock

- Prone to mistakes

Prevent hold-and-wait Acquire all locks at once

- Decreases concurrency significantly

Prevent no-preemption

- Problem: Livelock
- Challenge: encapsulation prevents the 'top' loop implementation

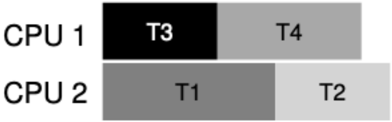
No mutual-exclusion

- Lock free programming

Deadlock Avoidance

Schedule threads 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 concflict
- *a* and *b* access same location
- At least one of *a* and *b* is a write

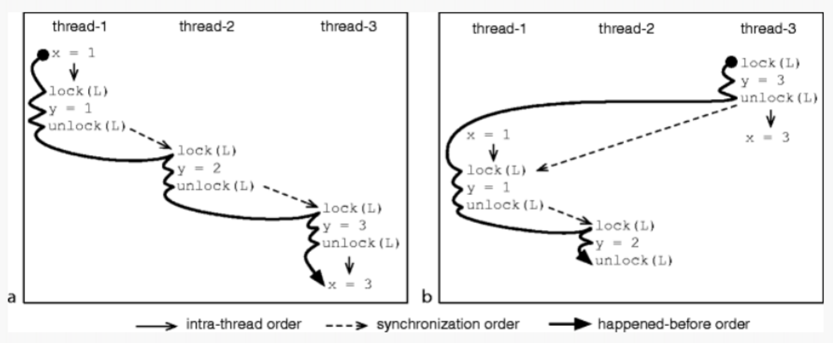
```
                    start       end
event e1 -------------|----------|---->

           start        end
event e2 ----|-----------|------------>

             start  end
event e3 ------|-----|---------------->
```

Concurrent: (e_1, e_2), (e_2, e_3)

e_3 happens-before e_1

- $\mathit{end}(e_3) \rightarrow \mathit{start}(e_1)$

concurrent/conflict $\Rightarrow$ Not in happens-before (HB) order



Execution 1: No data race
Execution 2: data race on x

Lockset algorithm

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{ \}$, then issue a warning.

Lockset algorithm

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

Example:

Program	*locks_held*	*C(v)*
	`{}`	`{mu1,mu2}`
`lock(mu1);`		
	`{mu1}`	
`v := v+1;`		
		`{mu1}`
`unlock(mu1);`		
	`{}`	
`lock(mu2);`		
	`{mu2}`	
`v := v+1;`		
		`{}`
`unlock(mu2);`		
	`{}`	

References

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://pages.cs.wisc.edu/ remzi/OSTEP/threads-bugs.pdf

Race Detection Techniques
Christoph von Praun
https://doi.org/10.1007/978-0-387-09766-4_38

Eraser: A Dynamic Data Race Detector for Multithreaded Programs
Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson. ACM TOCS 1997.