Concurrency Analysis for Multithreaded Programs

Soham Chakraborty

20.02.2023

Outline

Lock-based programming

Lock-free programming

Concurrency bugs

Example

$$X = 0;$$

$$X = X + 1; \;\Big\|\; X = X + 1;$$

What is the final value(s) of X?

Example

$$X = 0;$$
$$X = X + 1; \;\Big\|\; X = X + 1;$$

What is the final value(s) of X?

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

Example

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

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

Lock-based Programming

Init

lock(m)	‖	lock(m)
critical section	‖	critical section
unlock(m)	‖	unlock(m)

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 $\not\Longrightarrow$ 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 $\not\Longrightarrow$ 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 $\not\Longrightarrow$ 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)

$$
\begin{array}{l}
\mathit{void\ lock}()\{ \\
\quad i = \mathit{tid}(); \\
\quad j = 1 - i; \\
\quad \mathit{flag}[i] = \mathit{true}; \quad // \ I'm\ interested \\
\quad \mathit{while}(\mathit{flag}[j] == \mathit{true})\ \{\} \quad // \ wait\ loop \\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{unlock}()\ \{ \\
\quad i = \mathit{tid}(); \\
\quad \mathit{flag}[i] = \mathit{false}; \quad // \ I'm\ not\ interested \\
\}
\end{array}
$$

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: $a_0 \rightarrow a_1 \rightarrow \ldots$

Thread transitions are events

(a_0, a_1): Interval betwwen 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!

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

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

Violation of Mutual Exclusion: Critical sections overlap!

$$
\begin{array}{l}
void\ lock()\{ \\
\quad i = tid(); \\
\quad j = 1 - i; \\
\quad flag[i] = true; \\
\quad while(flag[j] == true) \\
\qquad \{\} \\
\}
\end{array}
$$

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

$$
\begin{array}{l}
unlock()\ \{ \\
\quad i = tid(); \\
\quad flag[i] = false; \\
\}
\end{array}
$$

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: $\mathrm{W}_0(\mathit{flag}[0], \mathit{true})$

$\downarrow$

b: $\mathrm{R}_0(\mathit{flag}[1], \mathit{false})$

$\downarrow$

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: $\mathrm{W}_0(flag[0], true)$ → b: $\mathrm{R}_0(flag[1], false)$ → CS_0

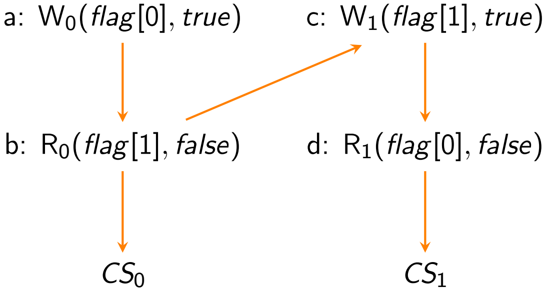
c: $\mathrm{W}_1(flag[1], true)$ → d: $\mathrm{R}_1(flag[0], false)$ → CS_1

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;
}
```

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



Violation of Mutual Exclusion: Critical sections overlap!

$$
\begin{array}{l}
\textit{void lock}()\{\\
\quad i = \textit{tid}();\\
\quad j = 1 - i;\\
\quad \textit{flag}[i] = \textit{true};\\
\quad \textit{while}(\textit{flag}[j] == \textit{true})\\
\qquad \{\}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textit{unlock}()\ \{\\
\quad i = \textit{tid}();\\
\quad \textit{flag}[i] = \textit{false};\\
\}
\end{array}
$$

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

$$
\begin{tikzcd}
\text{a: } \mathrm{W}_0(\textit{flag}[0], \textit{true}) \arrow[d] & \text{c: } \mathrm{W}_1(\textit{flag}[1], \textit{true}) \arrow[d] \\
\text{b: } \mathrm{R}_0(\textit{flag}[1], \textit{false}) \arrow[d] \arrow[ur] & \text{d: } \mathrm{R}_1(\textit{flag}[0], \textit{false}) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}
$$

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

Violation of Mutual Exclusion: Critical sections overlap!

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

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

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

$$
\begin{tikzcd}
\text{a: } \mathrm{W}_0(flag[0], true) \arrow[d] & \text{c: } \mathrm{W}_1(flag[1], true) \arrow[d] \\
\text{b: } \mathrm{R}_0(flag[1], false) \arrow[d] \arrow[ur] & \text{d: } \mathrm{R}_1(flag[0], false) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}
$$

Contradiction: $a \to d$ 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?

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

unlock() {}
```

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

$$\begin{array}{c} \mathrm{W}_0(victim, 0) \\ \downarrow \\ \mathrm{R}_0(victim, 1) \\ \downarrow \\ CS_0 \end{array}$$

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

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


unlock() {}
```

$$\begin{tikzcd} \mathrm{W}_0(victim, 0) \arrow[d] & & \mathrm{W}_1(victim, 1) \arrow[d] \\ \mathrm{R}_0(victim, 1) \arrow[d] & & \mathrm{R}_1(victim, 0) \arrow[d] \\ CS_0 & & CS_1 \end{tikzcd}$$

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

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

unlock() {}
```

$$\begin{tikzcd}
\mathrm{W}_0(victim, 0) \arrow[r] \arrow[d] & \mathrm{W}_1(victim, 1) \arrow[dl] \arrow[d] \\
\mathrm{R}_0(victim, 1) \arrow[d] & \mathrm{R}_1(victim, 0) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}$$

Mutual Exclusion

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


unlock() {}
```

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

$$\begin{tikzcd}
\mathrm{W}_0(victim, 0) \arrow[d] \arrow[dr] & \mathrm{W}_1(victim, 1) \arrow[l] \arrow[d] \\
\mathrm{R}_0(victim, 1) \arrow[d] & \mathrm{R}_1(victim, 0) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}$$

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

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


unlock() {}
```

$$\begin{tikzcd}
\mathrm{W}_0(victim, 0) \arrow[d] \arrow[dr] & \mathrm{W}_1(victim, 1) \arrow[l] \arrow[d] \\
\mathrm{R}_0(victim, 1) \arrow[d] & \mathrm{R}_1(victim, 0) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}$$

Problem: What if the threads are not running concurrently?

Mutual Exclusion

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

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

unlock() {}

$$\begin{tikzcd}
\mathrm{W}_0(\mathit{victim}, 0) \arrow[d] \arrow[dr] & \mathrm{W}_1(\mathit{victim}, 1) \arrow[l] \arrow[d] \\
\mathrm{R}_0(\mathit{victim}, 1) \arrow[d] & \mathrm{R}_1(\mathit{victim}, 0) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}$$

Problem: What if the threads are not running concurrently?

It deadlocks if one thread runs completely before the other.

Peterson's Lock Algorithm

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

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

Peterson's Lock Algorithm

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

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

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

Peterson's Lock Algorithm

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

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

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

$$\begin{tikzcd}
W_0(flag[0], true) \arrow[d] & a{:}W_1(flag[1], true) \arrow[d] \\
W_0(victim, 0) & W_1(victim, 1) \arrow[l]
\end{tikzcd}$$

```
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

$$
\begin{tikzcd}
W_0(\mathit{flag}[0], \mathit{true}) \arrow[d] & a{:}W_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \\
W_0(\mathit{victim}, 0) \arrow[d] & W_1(\mathit{victim}, 1) \arrow[l] \\
b{:}R_0(\mathit{flag}[1], \mathit{false}) \arrow[d] & \\
R_0(\mathit{victim}, 0) \arrow[d] & \\
\mathit{CS}_0 &
\end{tikzcd}
$$

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

$$\begin{tikzcd}
W_0(\mathit{flag}[0], \mathit{true}) \arrow[d] & \text{a:}W_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \\
W_0(\mathit{victim}, 0) \arrow[d] & W_1(\mathit{victim}, 1) \arrow[l] \\
\text{b:}R_0(\mathit{flag}[1], \mathit{false}) \arrow[d] & \\
R_0(\mathit{victim}, 0) \arrow[d] & \\
\mathit{CS}_0 &
\end{tikzcd}$$

no intermediate write on $\mathit{flag}[1]$ 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()

Categorization

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

$$
\begin{aligned}
&CAS(X, old, new)\{\\
&\quad if\,(X \neq old)\\
&\qquad return\ \ false;\\
&\quad X = new;\\
&\quad return\ \ true;\\
&\}
\end{aligned}
$$

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

```
void enq(int value) {
    Node node = new Node(value);
    while (true) {
        Node last = tail;
        Node next = last.next;

        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

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

        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

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

        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value); // create a new node
    while (true) {
        Node last = tail; // locate the last node
        Node next = last.next;
// identify the position to append the new node
        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value);  // create a new node
    while (true) {
        Node last = tail;  // locate the last node
        Node next = last.next;
// identify the position to append the new node
        if (last == tail) {
            if (next == null) {  // no successor
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
   Node node = new Node(value);  // create a new node
   while (true) {
      Node last = tail;  // locate the last node
      Node next = last.next;
// identify the position to append the new node
      if (last == tail) {
         if (next == null) {  // no successor
            if (CAS(last.next, next, node)) {  // append the new node
               CAS(tail, last, node);
                 return;
            }
         } else {

            CAS(tail, last, next);
         }
      }
   }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
   Node node = new Node(value);  // create a new node
   while (true) {
      Node last = tail;  // locate the last node
      Node next = last.next;
 // identify the position to append the new node
      if (last == tail) {
         if (next == null) {  // no successor
            if (CAS(last.next, next, node)) {  // append the new node
               CAS(tail, last, node);   // new node is the tail
                return;
            }
         } else {

            CAS(tail, last, next);
         }
      }
   }
}
```

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;
 //  identify the position to append the new node
6.        if (last == tail) {
7.           if (next == null) {  //  no successor
8.              if (CAS(last.next, next, node)) {  // append the new node
9.                 CAS(tail, last, node);   // new node is the tail
10.                 return;
11.              }
12.           } else {
 // tail has a successor; another thread is in between 8–9
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;
 //  identify the position to append the new node
6.          if (last == tail) {
7.              if (next == null) {  //  no successor
8.                  if (CAS(last.next, next, node)) {  // append the new node
9.                      CAS(tail, last, node);   // new node is the tail
10.                     return;
11.                 }
12.             } else {
 // tail has a successor; another thread is in between 8–9
13.                 CAS(tail, last, next);   // set tail to correct node
14.             }
15.         }
16.     }
17. }
```

Observations

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

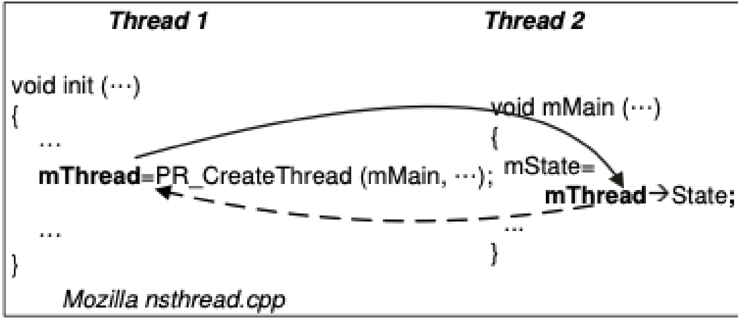
Order Violation

Cause: Programmer assumes certain ordering of events

Order Violation

Cause: Programmer assumes certain ordering of events

Example:

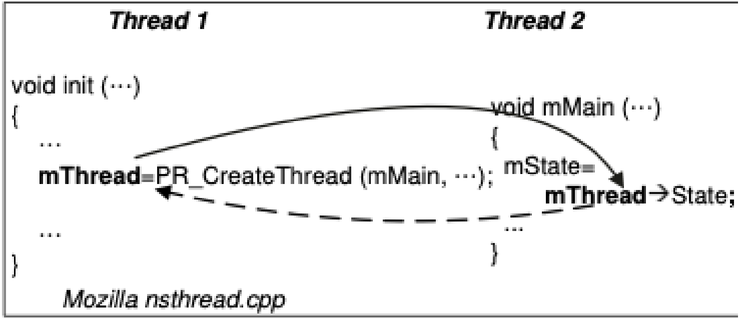


Thread 2 should not deref. **mThread** before Thread 1 initializes it

Order Violation

Cause: Programmer assumes certain ordering of events

Example:



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

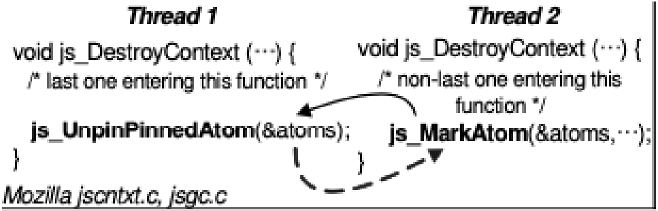
Pattern:

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

Order Violation

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

Example:

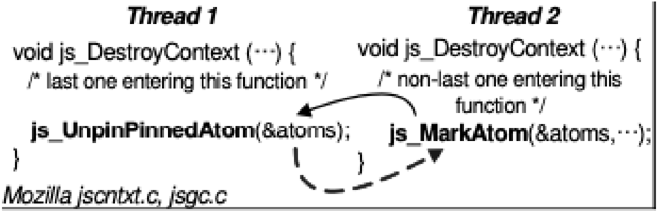


js_UnpinPinnedAtom should happen after js_MarkAtom.

Order Violation

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

Example:



js_UnpinPinnedAtom should happen after js_MarkAtom.

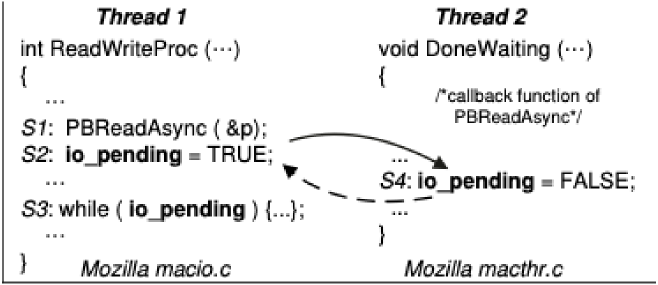
Pattern:

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

Order Violation

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

Example:

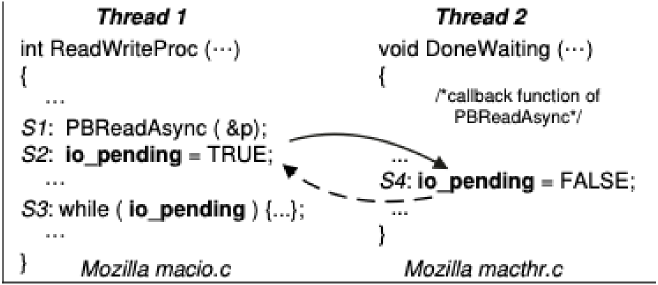


Assumption: S1 and S2 execute atomically

Unsafe ordering blocks thread 1

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

Example:



Assumption: S1 and S2 execute atomically

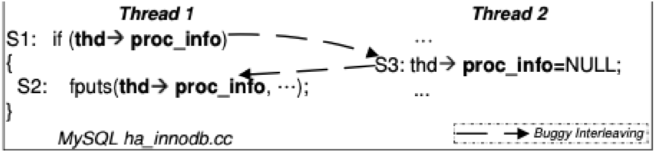
Unsafe ordering blocks thread 1

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:



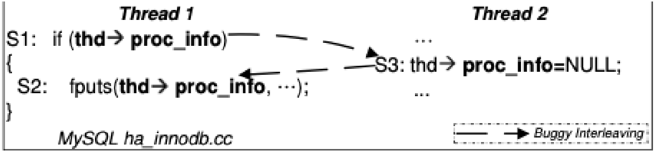
Assumption: S1;S2 are executed atomically

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:

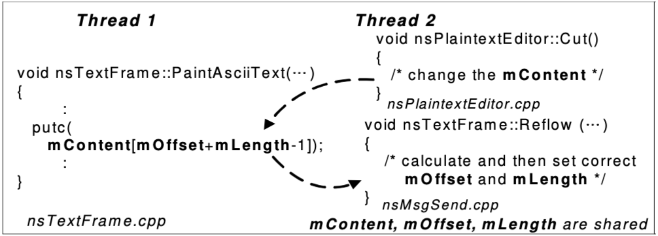
$$X = 0;$$
$$\begin{array}{l} a = X; \\ b = X; \end{array} \;\Bigg\|\; X = 1;$$

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

Multi-Variable Atomicity Bugs

Cause: variables are semantically connected which is violated

Example:



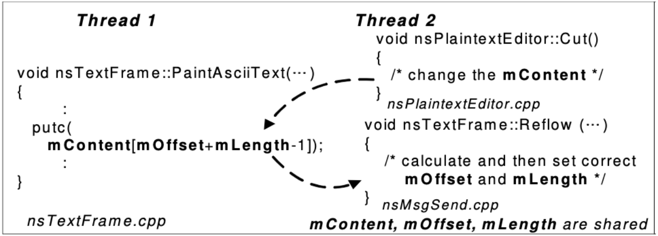
Assumption: *mOffset* and *mLength* are updated atomically wrt thread 1

Lack of synchronization $\Rightarrow$ 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

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

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

$lock(m_1);$	$lock(m_2);$
$\quad lock(m_2);$	$\quad lock(m_1);$
$\ldots$	$\ldots$
$\quad unlock(m_2);$	$\quad unlock(m_1);$
$unlock(m_1);$	$unlock(m_2);$

Deadlock: Another Scenario

Another challenge: encapsulation

$$\mathit{Vector}\ v1,\ v2;$$
$$v1.\mathit{AddAll}(v2);\ \|\ v2.\mathit{AddAll}(v1);$$

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.

Deadlock Prevention

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

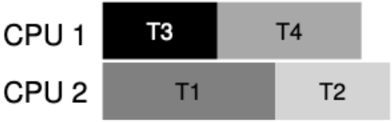
No mutual-exclusion

- 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 concflict
- *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. $CAS(X, v, v', o_s, o_f)$
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

Reordeing Rules

$a(\ell) \Downarrow / b(\ell') \Rightarrow$	$\mathsf{Ld_{na}}/\mathsf{St_{na}}$	$\mathsf{St_{sc}}$	$\mathsf{Ld_{sc}}$
$\mathsf{Ld_{na}}/\mathsf{St_{na}}$	✓	✗	✓
$\mathsf{St_{sc}}$	✓	✗	✗
$\mathsf{Ld_{sc}}$	✗	✗	✗

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

$$X = NULL, flag = 0;$$

$$\begin{array}{l} X = new\ Obj(); \\ flag = 1; \end{array} \quad \left\| \quad \begin{array}{l} while(flag \neq 1) \text{ // } \textit{waiting....} \\ \quad ; \\ t = X; \text{ // must be non-NULL} \end{array} \right.$$

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

Thread Communication & Synchronization

$$
\begin{array}{c}
X = NULL, flag = 0; \\
\begin{array}{l} X = new\ Obj(); \\ flag = 1; \end{array}
\;\Bigg\|\;
\begin{array}{l} while(flag \neq 1) \\ \quad ; \\ t = X; \end{array}
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{c}
X = NULL, Y = 0; \\
\begin{array}{l} flag = 1; \\ X = new\ Obj(); \end{array}
\;\Bigg\|\;
\begin{array}{l} while(flag \neq 1) \\ \quad ; \\ t = X; \end{array}
\end{array}
$$

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

Examples: $X = 0$ initially.

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

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/