Reasoning About The Implementations Of Concurrency Abstractions On x86-TSO

By Scott Owens,

University of Cambridge.

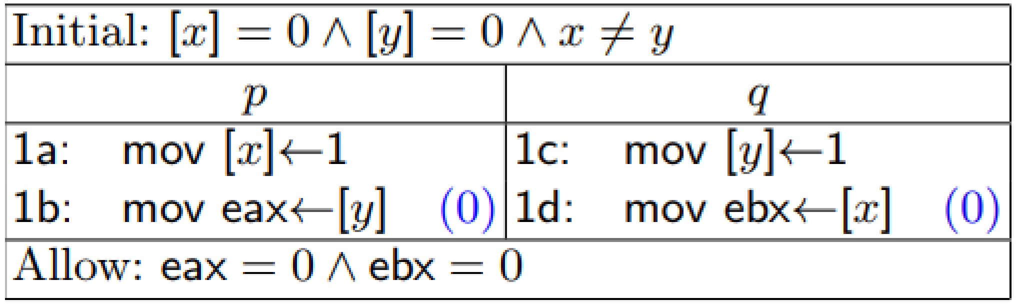
Plan

- **Intro**
- Data Races And Triangular Races
- Examples

sequential consistency

- The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

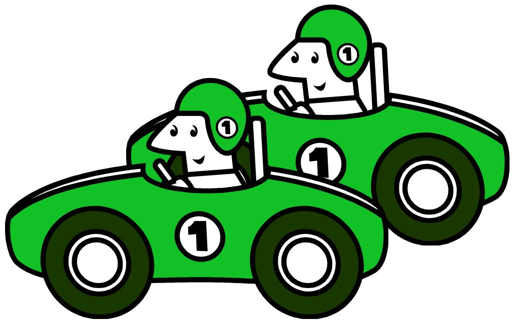
Simple Example



Initial: $[x] = 0 \wedge [y] = 0 \wedge x \neq y$	
p	q
1a: mov $[x]$←1	1c: mov $[y]$←1
1b: mov eax←$[y]$ (0)	1d: mov ebx←$[x]$ (0)
Allow: eax $= 0 \wedge$ ebx $= 0$	

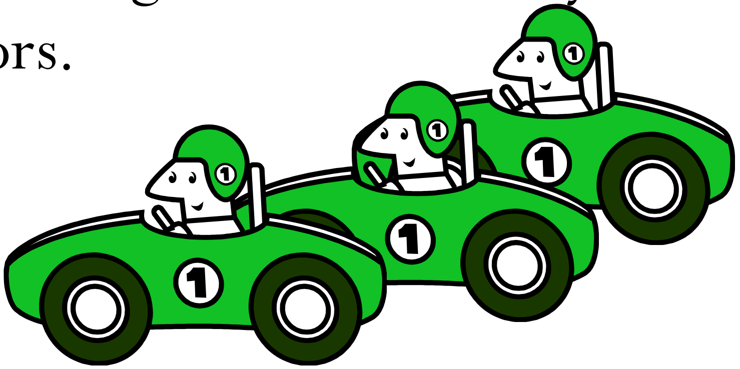
Data Races

- Intuitively – 2 threads that access the same memory address, and at least one of them is writing.
- If none of a program execution can encounter a data race, then the program is data-race free
- The fundamental property of memory model: DRF programs have no observable non SC behaviors.

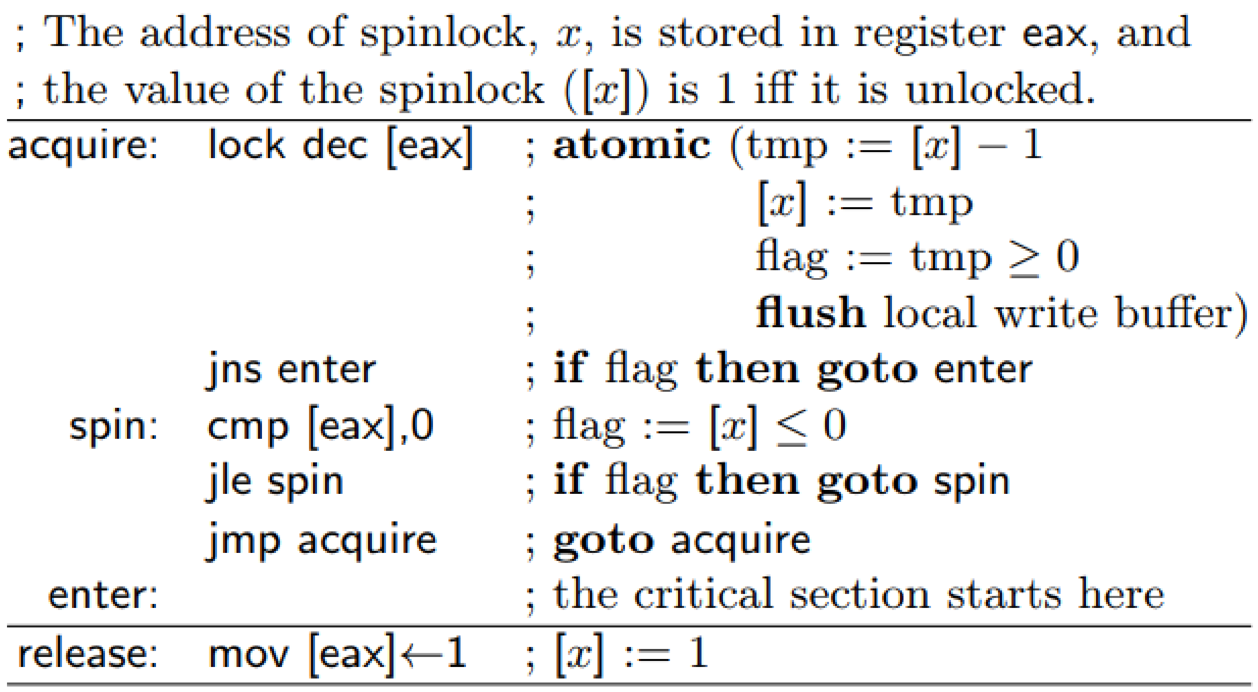


Triangular Races

- Triangular race – a data race between a read and a write operation, where the read is preceded by another write operation on the same thread, and there are no intervening hardware synchronization primitives.
- TRF – Triangular Race Free programs
- Main Result - TRF programs running on TSO memory model has only SC observable behaviors.



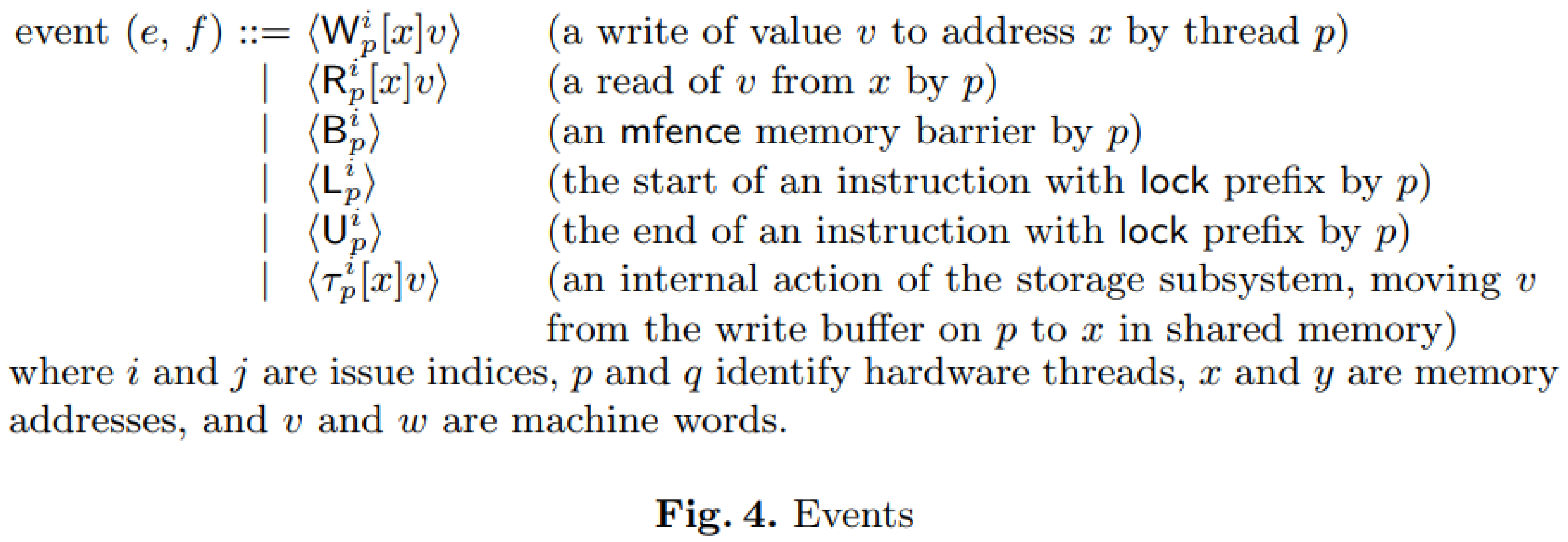
Example



```
; The address of spinlock, $x$, is stored in register eax, and
; the value of the spinlock ([$x$]) is 1 iff it is unlocked.
----------------------------------------------------------------------
acquire:  lock dec [eax]   ; atomic (tmp := [$x$] − 1
                           ;         [$x$] := tmp
                           ;         flag := tmp $\geq$ 0
                           ;         flush local write buffer)
          jns enter        ; if flag then goto enter
   spin:  cmp [eax],0      ; flag := [$x$] $\leq$ 0
          jle spin         ; if flag then goto spin
          jmp acquire      ; goto acquire
  enter:                   ; the critical section starts here
----------------------------------------------------------------------
release:  mov [eax]←1      ; [$x$] := 1
----------------------------------------------------------------------
```

Fig. 2. An x86 spinlock from Linux v2.6.24.7 (pseudocode to the right of ;)

Events



$$\begin{array}{rcll}
\text{event } (e,\ f) & ::= & \langle \mathsf{W}_p^i[x]v \rangle & \text{(a write of value } v \text{ to address } x \text{ by thread } p\text{)} \\
 & | & \langle \mathsf{R}_p^i[x]v \rangle & \text{(a read of } v \text{ from } x \text{ by } p\text{)} \\
 & | & \langle \mathsf{B}_p^i \rangle & \text{(an } \mathsf{mfence} \text{ memory barrier by } p\text{)} \\
 & | & \langle \mathsf{L}_p^i \rangle & \text{(the start of an instruction with } \mathsf{lock} \text{ prefix by } p\text{)} \\
 & | & \langle \mathsf{U}_p^i \rangle & \text{(the end of an instruction with } \mathsf{lock} \text{ prefix by } p\text{)} \\
 & | & \langle \tau_p^i[x]v \rangle & \text{(an internal action of the storage subsystem, moving } v \\
 & & & \text{from the write buffer on } p \text{ to } x \text{ in shared memory)}
\end{array}$$

where i and j are issue indices, p and q identify hardware threads, x and y are memory addresses, and v and w are machine words.

Fig. 4. Events

X86-SC

- A simple model of SC memory behavior
- Same as x86-TSO, but every write is immediately flushed to shared memory, and every read consults main storage
- Each $< W_p^i[x]v >$ event must be immediately followed by a $< \tau_p^i[x]v >$ event

resultSC

- A program is resultSC iff for every x86-TSO execution there exists an x86-SC execution with the same result.

- TRF is not powerful enough to detect resultSC programs

Memory equivalence

- Two memory equivalent traces must have the same memory writes in the same order
- Corresponding read events must have the same value and the values must have been put in place by the same write event
- Reads From – a read event reads from the last write event

MemorySC

- A program is memory SC iff for each of its possible execution on x86-TSO, there exists a memory equivalent execution on x86-SC

Plan

- Intro
- Data Races And Triangular Races
- Examples

Data Race

- A data race is a prefix of an x86-SC execution with either of the following shapes:

$$e_1 \ldots e_n \langle \mathsf{R}_q[x]v \rangle \langle \mathsf{W}_p[x]w \rangle \qquad or \qquad e_1 \ldots e_n \langle \mathsf{R}_q[x]v \rangle \langle \mathsf{L}_p \rangle f_1 \ldots f_m \langle \mathsf{W}_p[x]w \rangle$$

where p ≠ q and none of the f are unlocks

- DRF Theorem – every DRF x86 program is memorySC (follows from the main theorem)

Triangular Races

- A triangular race is a prefix of an x86-SC execution with either one of the 2 shapes:

$$e_1 \dots e_m \langle \mathsf{W}_q[y]v_1\rangle\langle \mathsf{R}_q[z_1]w_1\rangle \dots \langle \mathsf{R}_q[z_n]w_n\rangle\langle \mathsf{R}_q[x]v_2\rangle\langle \mathsf{W}_p[x]v_3\rangle \;\; or$$
$$e_1 \dots e_m \langle \mathsf{W}_q[y]v_1\rangle\langle \mathsf{R}_q[z_1]w_1\rangle \dots \langle \mathsf{R}_q[z_n]w_n\rangle\langle \mathsf{R}_q[x]v_2\rangle\langle \mathsf{L}_p\rangle f_1 \dots f_o\langle \mathsf{W}_p[x]v_3\rangle$$

  where x ≠ y and p ≠ q and x ∉ $\{Z_1, \dots, Z_n\}$ and none of the f are unlocks

- TRF Theorem – An x86 program is memorySC iff it is TRF

Example

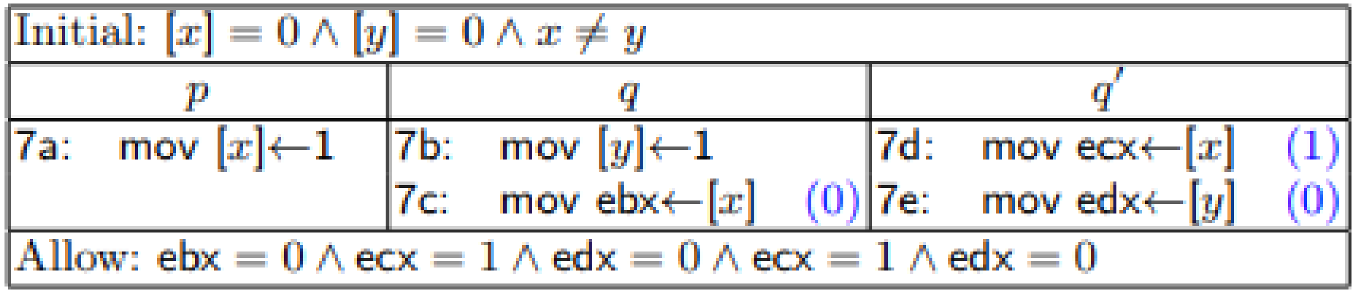
Initial: $[x] = 0 \wedge [y] = 0 \wedge x \neq y$	
p	q
6a: mov $[x] \leftarrow 1$	6b: mov $[y] \leftarrow 1$
	6c: mov eax$\leftarrow [x]$ (0)

Fig. 6. A simple triangular race

Consider the following sequence of actions on an x86-TSO. Can we find a memory equivalent x86-SC trace?

1.		$\langle \mathsf{W}_q^1[y]1 \rangle$	buffer the write of 1 to y (6b)
2.		$\langle \mathsf{R}_q^2[x]0 \rangle$	read x from main memory (6c)
3.	$\langle \mathsf{W}_p[x]1 \rangle$		buffer the write of 1 to x (6a)
4.	$\langle \tau_p[x]1 \rangle$		write 1 to x in shared memory (6a)
5.		$\langle \tau_q[y]1 \rangle$	write 1 to y in shared memory (6b)

Example Continued



Initial: $[x]=0 \wedge [y]=0 \wedge x \neq y$		
p	q	q'
7a: mov $[x]$←1	7b: mov $[y]$←1	7d: mov ecx←$[x]$ (1)
	7c: mov ebx←$[x]$ (0)	7e: mov edx←$[y]$ (0)
Allow: ebx $=0 \wedge$ ecx $=1 \wedge$ edx $=0 \wedge$ ecx $=1 \wedge$ edx $=0$		

Fig. 7. Observing write ordering leads to relaxed behavior

Plan

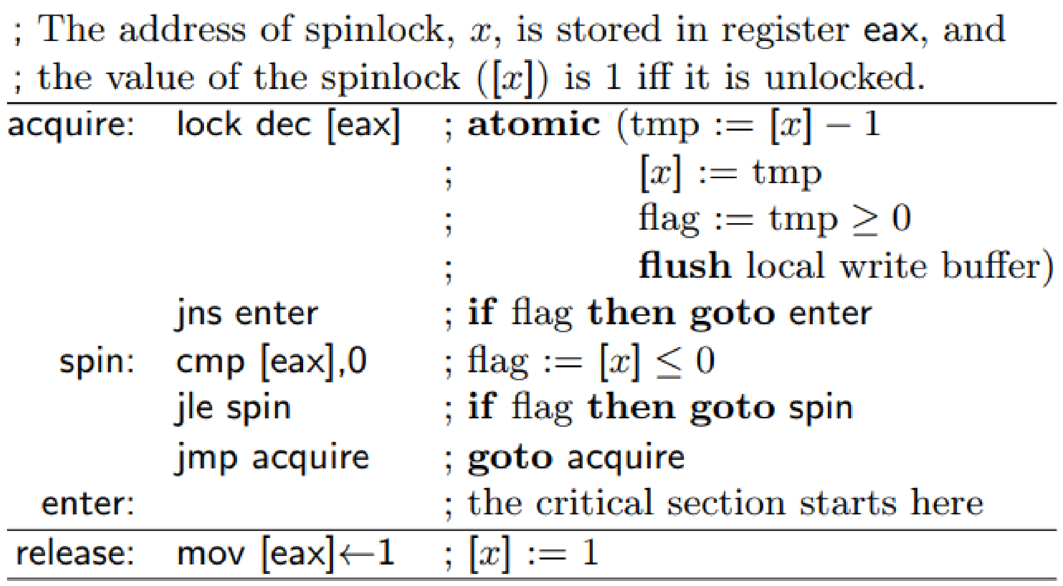
- Intro
- Data Races And Triangular Races
- **Examples**

Back To The SpinLock

- We say a program is Spinlock well-synchronized iff for every x86-SC execution, and for every pair of competing events that are not on a spinlock, there is a spinlock that is released and then acquired between them
- Lemma – In a Spinlock well-synchronized program any data race is on a spinlock address.

Back To The SpinLock

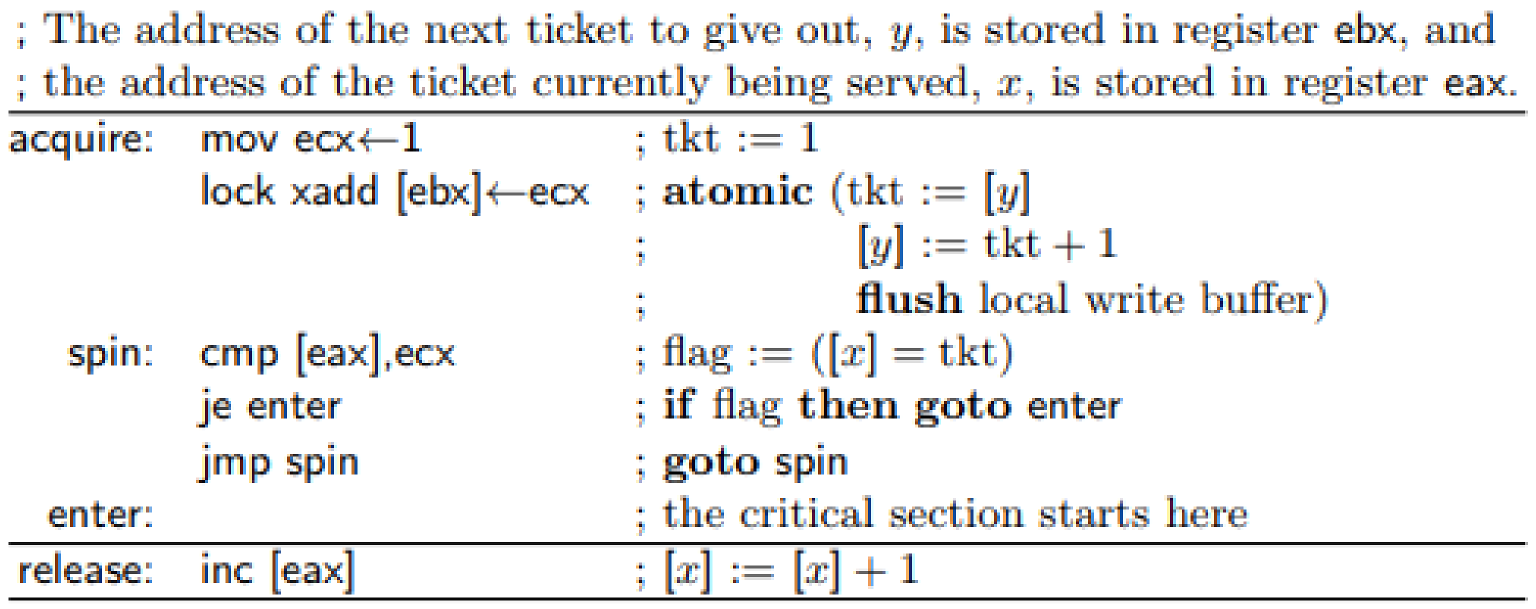
- Theorem – if an x86 program is Spinlock well-synchronized and the locations of the spinlocks ae only accessed by the spinlock code, then it is memorySC.



```
; The address of spinlock, $x$, is stored in register eax, and
; the value of the spinlock ($[x]$) is 1 iff it is unlocked.
-----------------------------------------------------------------
acquire:  lock dec [eax]   ; atomic (tmp := $[x] - 1$
                           ;          $[x]$ := tmp
                           ;          flag := tmp $\geq 0$
                           ;          flush local write buffer)
          jns enter        ; if flag then goto enter
   spin:  cmp [eax],0      ; flag := $[x] \leq 0$
          jle spin         ; if flag then goto spin
          jmp acquire      ; goto acquire
  enter:                   ; the critical section starts here
-----------------------------------------------------------------
release:  mov [eax]←1      ; $[x]$ := 1
-----------------------------------------------------------------
```

Fig. 2. An x86 spinlock from Linux v2.6.24.7 (pseudocode to the right of ;)

Ticketed Spinlock



```
; The address of the next ticket to give out, y, is stored in register ebx, and
; the address of the ticket currently being served, x, is stored in register eax.
acquire:  mov ecx←1             ; tkt := 1
          lock xadd [ebx]←ecx   ; atomic (tkt := [y]
                                ;            [y] := tkt + 1
                                ;            flush local write buffer)
   spin:  cmp [eax],ecx         ; flag := ([x] = tkt)
          je enter              ; if flag then goto enter
          jmp spin              ; goto spin
  enter:                        ; the critical section starts here
release:  inc [eax]             ; [x] := [x] + 1
```

Fig. 8. A ticketed x86 spinlock inspired by Linux v2.6.31

Correctly Locked

- A program is correctly locked if each of its x86-SC execution traces satisfies:
    1. The locations of the spinlocks are only accessed by the spinlock code
    2. Threads only release lock they hold.

Spinlock mutual exclusion

- Lemma – In a correctly locked x86 program, if a hardware thread reaches the enter line of a spinlock, no other thread can reach the enter line until the first thread completes the increment from release.

Ticketed Spinlock Is SC

- Theorem – if a correctly locked x86 program is spinlock well-synchronized with respect to the ticketed spinlock then it is memorySC

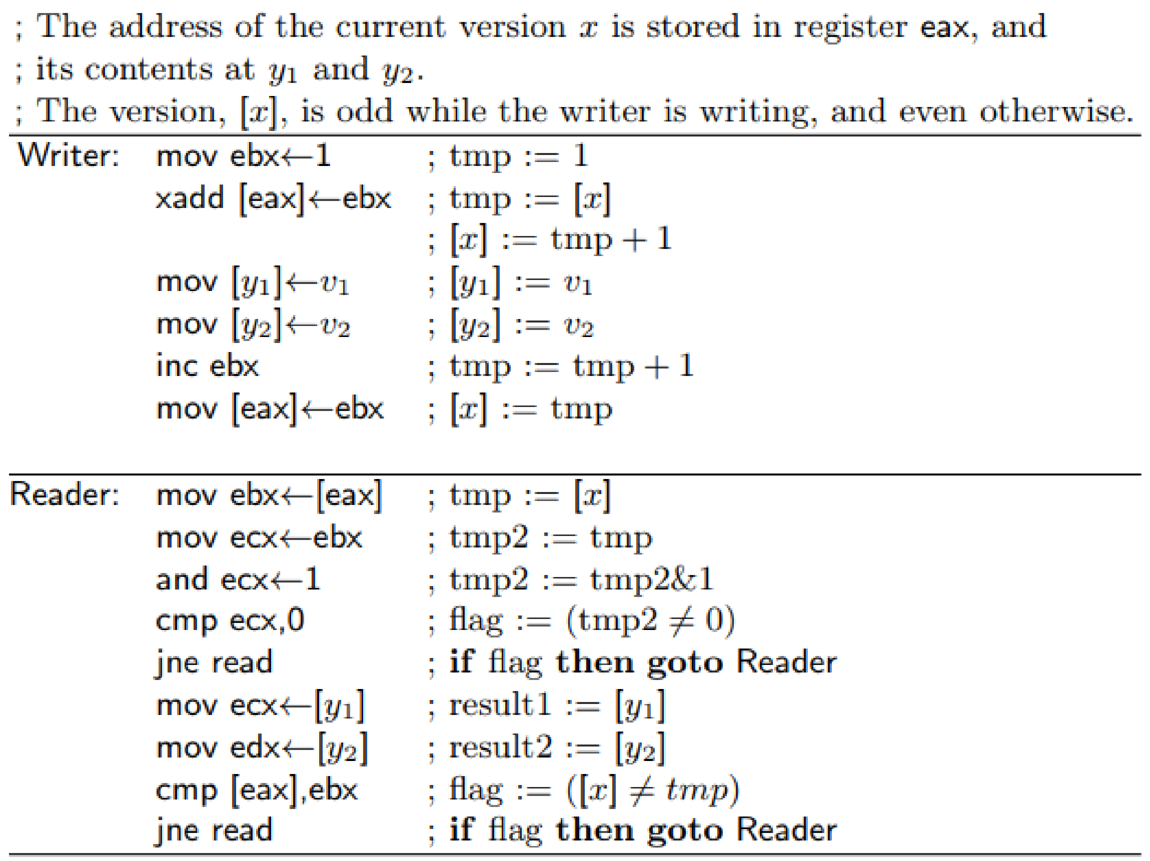
Proof

- Analyze the possible data races:

acquire: $\langle \mathsf{L}_p \rangle \langle \mathsf{R}_p[y]w_1 \rangle \langle \mathsf{W}_p[y]w_2 \rangle \langle \mathsf{U}_p \rangle \langle \mathsf{R}_p[x]v_1 \rangle \dots \langle \mathsf{R}_p[x]v_n \rangle$
release: $\langle \mathsf{R}_p[x]v_1 \rangle \langle \mathsf{W}_p[x]v_2 \rangle$

- 3 possible races:
  - Reading y in acquire
  - Reading x in acquire
  - Reading x in release

Non Blocking Write Protocol



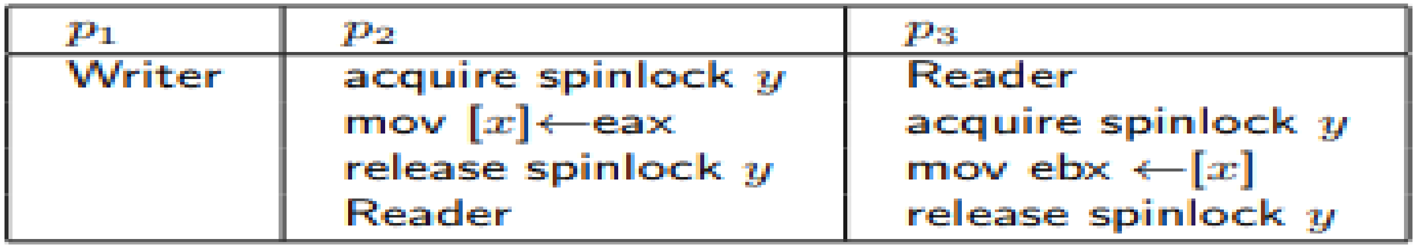
```
; The address of the current version $x$ is stored in register eax, and
; its contents at $y_1$ and $y_2$.
; The version, $[x]$, is odd while the writer is writing, and even otherwise.
-------------------------------------------------------------------------------
Writer:  mov ebx←1          ; tmp := 1
         xadd [eax]←ebx     ; tmp := $[x]$
                            ; $[x]$ := tmp + 1
         mov $[y_1]$←$v_1$        ; $[y_1]$ := $v_1$
         mov $[y_2]$←$v_2$        ; $[y_2]$ := $v_2$
         inc ebx            ; tmp := tmp + 1
         mov [eax]←ebx      ; $[x]$ := tmp

-------------------------------------------------------------------------------
Reader:  mov ebx←[eax]      ; tmp := $[x]$
         mov ecx←ebx        ; tmp2 := tmp
         and ecx←1          ; tmp2 := tmp2&1
         cmp ecx,0          ; flag := (tmp2 $\neq$ 0)
         jne read           ; if flag then goto Reader
         mov ecx←$[y_1]$         ; result1 := $[y_1]$
         mov edx←$[y_2]$         ; result2 := $[y_2]$
         cmp [eax],ebx      ; flag := ($[x] \neq tmp$)
         jne read           ; if flag then goto Reader
-------------------------------------------------------------------------------
```

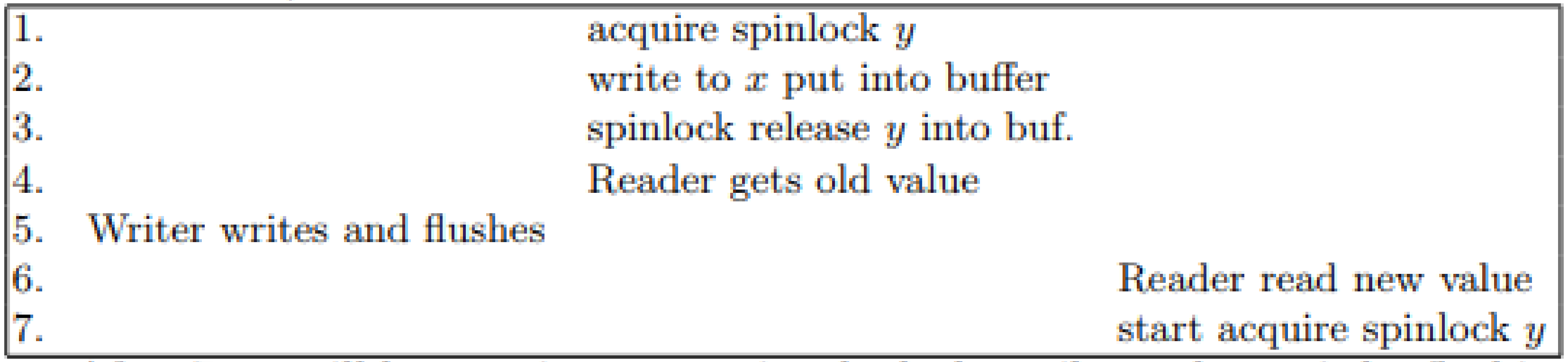
Fig. 9. A versioning non-blocking write protocol

Non Blocking Write Protocol

- If the reader also writes to memory, then the code is not TRF and therefor not memorySC, but it can be resultSC



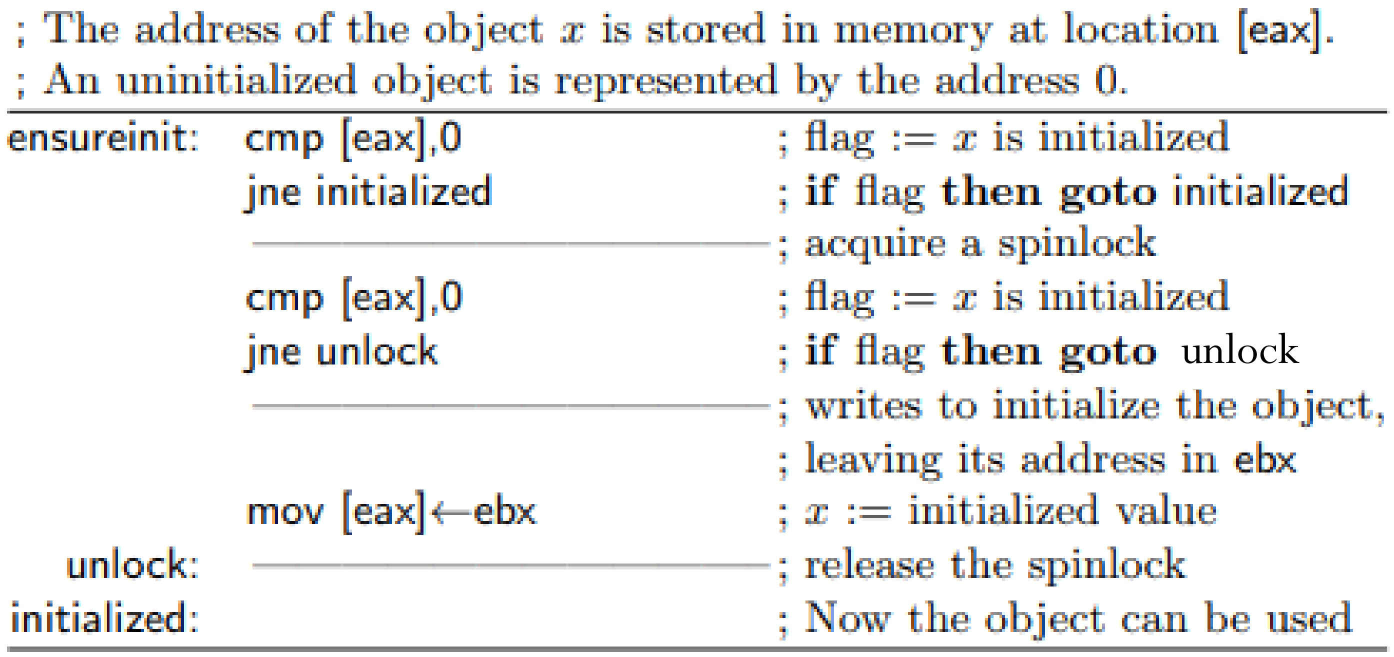
p_1	p_2	p_3
Writer	acquire spinlock y mov $[x]\leftarrow$eax release spinlock y Reader	Reader acquire spinlock y mov ebx $\leftarrow[x]$ release spinlock y



1.		acquire spinlock y	
2.		write to x put into buffer	
3.		spinlock release y into buf.	
4.		Reader gets old value	
5.	Writer writes and flushes		
6.			Reader read new value
7.			start acquire spinlock y

Double Checekd Locking

- An object x is never accessed without first ensuring it has been initialized using ensureinit



```
; The address of the object x is stored in memory at location [eax].
; An uninitialized object is represented by the address 0.
ensureinit:   cmp [eax],0                  ; flag := x is initialized
              jne initialized              ; if flag then goto initialized
              ____________________________ ; acquire a spinlock
              cmp [eax],0                  ; flag := x is initialized
              jne unlock                   ; if flag then goto unlock
              ____________________________ ; writes to initialize the object,
                                           ; leaving its address in ebx
              mov [eax]←ebx                ; x := initialized value
     unlock:  ____________________________ ; release the spinlock
initialized:                               ; Now the object can be used
```

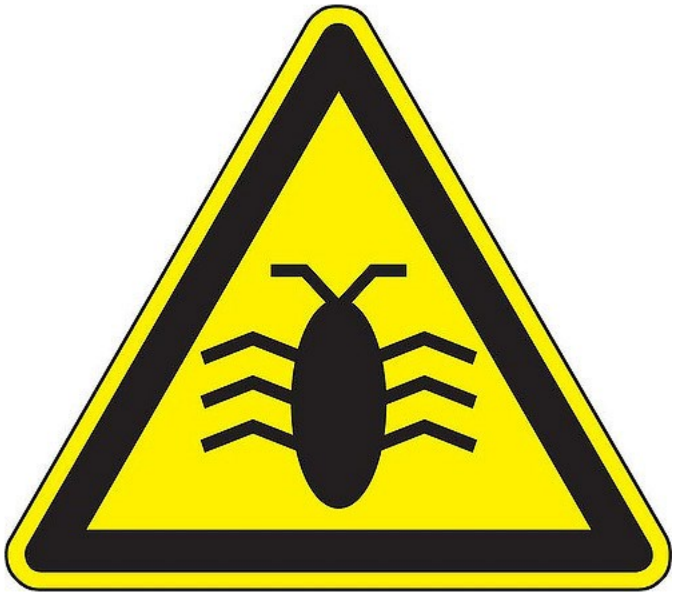
Fig. 10. Double-checked Locking

Double Checked Locking

- On x86-SC, one of 3 things can happen while ensuring initaliazition:
  1. Read x, find it initialized, use x
  2. Read x, find it uninitialized, lock, read x, find it initialized, unlock, use x
  3. Read x, find it uninitialized, lock, read x, find it uninitialized, init x, unlock, use x

A JVM Bug Due To TRF

- A concurrency bug in jvm implementation of blocking sychronization.
- A cretain excutaion using Parker::park could lose a wake-up call due to a missing mfence



```
class Parker {
  volatile int _counter = 0;
  pthread_mutex_t _mutex [1];    pthread_cond_t _cond [1];
};

void Parker::park() {
  if (_counter > 0) {
    _counter = 0;
    // mfence needed here
    return;
  }
  if (pthread_mutex_trylock(_mutex) != 0) return;
  if (_counter > 0) { // no wait needed
    _counter = 0;
    pthread_mutex_unlock(_mutex);
    return;
  }
  pthread_cond_wait(_cond, _mutex);
  _counter = 0;
  pthread_mutex_unlock(_mutex);
}

void Parker::unpark() {
  pthread_mutex_lock(_mutex);
  int s = _counter;
  _counter = 1;
  pthread_mutex_unlock(_mutex);
  if (s < 1) pthread_cond_signal(_cond);
}
```

Fig. 11. A simplified `Parker` from HotSpot (written in C++) taken from [12]

Usage Example

- Thread p want to wait for condition x==0

```
while !(x == 0) pk.park();
```

- Thread q wants to signal the first thread the condition is met

```
x = 0;  mfence();  pk.unpark();
```

Parker Is Not TRF

- P events: $\langle \mathsf{R}_p^i[x]1\rangle\langle \mathsf{R}_p[y]1\rangle\langle \mathsf{W}_p[y]0\rangle\langle \mathsf{R}_p^j[x]1\rangle.$
- Q events: $\langle \mathsf{W}_q[x]0\rangle\langle \mathsf{B}_q\rangle\ldots$
- If all of q events happens after all of p events we get a triangular race

The Bug In Action

- T1 check the condition, calls park, fetches _counter=1, set _counter=0 (in local buffer) and returns
- T1 checks the condition, calls park again
- T2 satisfies the condition, calls unpark, set _counter=1 (in shared memory) and returns
- T1 flushes _counter=0 to memory
- T1 fetches _counter=0, and blocks
- T1 hangs

Proof Sketch 1

- Triangular race causes non memorySC behaviour:
  - Suppose we have $< W_p^i[y]v >$ $< R_p^i[x]u >$ $< W_q^i[x]z >$
  - If read returns old value, $< R_p^i[x]u >$ happened before $< W_q^i[x]z >$ happened, we conclude $< W_p^i[y]v >$ happened before $< W_q^i[x]z >$
  - In TSO this is not the case, and we cannot conclude this.

Proof Sketch 2

- No triangular race guarantees memorySC
  - Suppose our program is TRF, take a trace and move all $<\tau_p^i[x]v>$ events immediately after the corresponding write event.
  - Suppose the new trace is not an equivalent SC trace then there was a race between a read and a $<W_p^i[x]v>$ and $<\tau_p^i[x]v>$
  - Since the program is TRF we can move the read event before\after the write event