

# **An Introduction to Lock-free Programming**

**Tony Van Eerd**  
**BoostCon 2010**

**Forewarned is Forearmed...**

**``Premature optimization is the root of all evil."**

**-- Knuth  $\rightarrow$  Hoare  $\rightarrow$  Dijkstra**

``Premature optimization is the root of all evil."

-- Knuth → Hoare → Dijkstra

"What 99 percent of programmers need to know is  
not how to build components but how to use them."

-- Alexander Stepanov

“Lock-free programming is hard.”

...

“Lock-free programming is hard.”

...

“Lock-free programming is not that hard.”

...

“Lock-free programming is hard.”

...

“Lock-free programming is not that hard.”

...

“Lock-free programming is *hard*.”

- Typical progression of a programmer learning lock-free programming.

“Use Locks!”

- me, etc

*...And that concludes the talk. :-)*



*I Am Not An Expert.*

## ***Lock Calculus – An Alternative Definition of Lock-free Programming***

```
{
    scoped_lock lock(mutex);

    do();
    some();
    stuff();
    while();
    holding();
    lock();

} // mutex unlocked in scoped_lock destructor
```

*How long should the lock be held?*

## ***Lock Calculus – An Alternative Definition of Lock-free Programming***

$\lim_{\text{lock} \rightarrow \infty} P = \text{sequential}$ <ul style="list-style-type: none"><li>+ no Deadlocks!</li><li>+ no livelocks</li><li>+ minimizes logic errors</li><li>– poor use of resources/CPU's</li></ul>	$\lim_{\text{lock} \rightarrow 0} P = \text{lock-free}$ <ul style="list-style-type: none"><li>+ no Deadlocks!</li><li>– chance of livelocks</li><li>– <i>maximizes logic errors!</i></li><li>+ good use of resources/CPU's</li></ul>
---	--

*(Where  $P$  is your program, and 'infinity' is really the maximum lock-length – the complete length of the program.)*



**Are We There Yet?**  
**Are We There Now?**  
**What About *Now*?...**

*What's wrong with this code...*

```
if (b != 0)
{
    x = a / b;
}
else ...
```

*What's wrong with this code...*

```
if (b != 0)
{
    x = a / b;
}
else ...
```

*...in a multithreaded program?*

*What's wrong with this code, in a multithreaded program?*

```
if (b != 0)
{
    x = 10;
}
```



*Aha!*

```
temp = b;  
if (temp != 0)  
{  
    x = a / temp;  
}
```

*The Fundamental difference...*

```
if (b != 0)
{
    x = a / b;
}
```

```
temp = b;
if (temp != 0)
{
    x = a / temp;
}
```

*...that you will forget again and again.*

Kids: “Are we there yet?”

Dad: “No; Were here!”

*Try that with your kids next time.*

### *What's wrong with this code?*

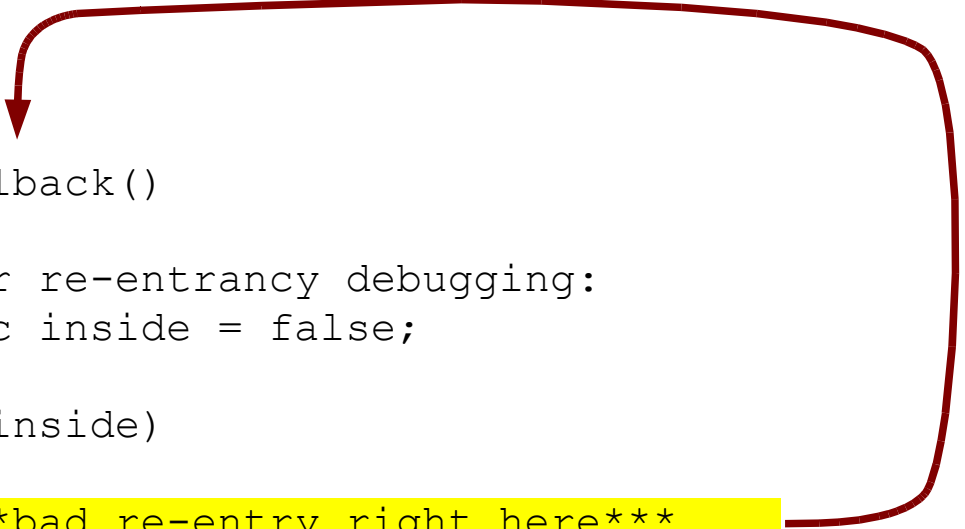
*(Based on actual code (by me) in a device driver, 15+ years ago.)*

```
void Callback()
{
    // for re-entrancy debugging:
    static inside = false;

    if (!inside)
    {
        inside = true;

        // non re-entrant
        do_stuff();
    }
    else
    {
        printf("oh my gosh, it's re-entrant!!!");
    }
}
```

*Read between the lines*



```
void Callback()
{
    // for re-entrancy debugging:
    static inside = false;

    if (!inside)
    {
        ***bad re-entry right here***
        inside = true;

        // non re-entrant
        do_stuff();
    }
    else
    {
        printf("oh my gosh, it's re-entrant!!!");
    }
}
```

*Need to close the gap between checking and setting...*

```
if (!inside)
{
    *** The Gap ***
    inside = true;

    do_stuff();
    ...
}
```

*The Atomic Solution:*

```
if (XCHG(&inside, 1) == 0)
{
    do_stuff();
    ...
}
```

**XCHG**(&target, value): // “exchange”

**Atomically** { set target = value; return old target}. Atomically – ie NO GAP.

***BFF: CAS...***



*Concurrent counting:*

*What's wrong with this code?*

```
int count = 0;
void main()
{
    start_threads(thread1, thread2);
}

// Thread 1
void thread1()
{
    int i = 100;
    while(i--)
        count++;
}

// Thread 2
void thread2()
{
    int i = 100;
    while(i--)
        count++;
}
```

*What is the end value of count? 200?*

*A closer look at “++”*

```
count++;
```

*A closer look at “++”*

```
count++;
```

=>

```
count = count + 1;
```

*A closer look at “++”*

```
count++;
```

=>

```
count = count + 1;
```

=>

```
register reg = read_memory(&count); // Read  
reg = reg + 1;                       // Modify  
write_memory(&count, reg);           // Write
```

*A closer look at “++”*

```
count++;
```

=>

```
count = count + 1;
```

=>

```
register reg = read_memory(&count); // Read  
reg = reg + 1;                       // Modify  
write_memory(&count, reg);           // Write
```

*Now, remember:*

- *Read between the lines!*
- *“Are we there yet?”*

*Concurrent counting:*

*What's wrong with this code?*

```
int count = 0;
void main()
{
    start_threads(thread1, thread2);
}

// Thread 1
void thread1()
{
    int i = 100;
    while(i--)
        count++;
}

// Thread 2
void thread2()
{
    int i = 100;
    while(i--)
        count++;
}
```

*Thread 1 can change count while thread 2 is incrementing, and vice-versa.*

*But at least the count is always **increasing**, right? Right!?*

*(based on a question from comp.programming.threads)*

```
int count = 0;
void main()
{
    start_threads(thread1, thread2);
}

// Thread 1
void thread1()
{
    int i = 100;
    while(i--)
        count++;
}

// Thread 2
void thread2()
{
    int i = 100;
    while(i--)
        count++;
}
```

``Premature optimization..."



``Use Locks"

*A closer look:*

```
// Thread 1
void thread1()
{
    int i = 100;
    while(i--)
        count++;
}
```

```
// Thread 2
void thread2()
{
    int i = 100;
    while(i--)
        count++;
}
```

```
...thread1,thread2 do some incrementing...
[count == 8]
thread1 reads count into reg1 [reg1 == 8]
    thread2 reads count [reg2 == 8]
thread1 increments reg1 [reg1 == 9]
thread1 writes count [count == 9]
thread1 repeats r.m.w. [count == 10]
thread1 repeats r.m.w. [count == 11]
    thread2 incr reg2 8->9 [reg2 == 9]
    thread2 writes count [count = 9]
thread1 reads count [count == 9]
```

Result: thread1 saw count == 11, then 9.

*Result: Yikes!*

*If only there was a way to know that the other thread modified count during our increment...*

```
register reg = read_memory(&count);    // Read  
reg = reg + 1;                        // Modify  
// OK to write ???  
write_memory(&count, reg);            // Write
```

*If only there was a way to know that the other thread modified count during our increment...*

*I know, we'll check to see if it changed, before setting it!!!*

```
oldcount = count;
register reg = read_memory(&count); // Read
reg = reg + 1;                      // Modify
if (count == oldcount) //OK to write?
    write_memory(&count, reg);      // Write
```

*If only there was a way to know that the other thread modified count during our increment...*

*I know, we'll check to see if it changed, before setting it!!!*

```
retry:
    oldcount = count;
    register reg = read_memory(&count);    // Read
    reg = reg + 1;                          // Modify
    if (count == oldcount) //OK to write?
        write_memory(&count, reg);        // Write
    else
        goto retry;
```

*Do I need to ask “What's wrong with this code?” (besides the goto)?*

*If only there was a way to know that the other thread modified count during our increment...*

*I know, we'll check to see if it changed, before setting it!!!*

```
retry:
    oldcount = count;
    register reg = read_memory(&count);    // Read
    reg = reg + 1;                          // Modify
    if (count == oldcount) //OK to write?
        --->>> *** here *** <<<---
        write_memory(&count, reg);        // Write
    else
        goto retry;
```

*“Are we there yet?”*

*Need to close the gap...*

*Meet your new **Best Friend Forever**, CAS:*

```
// "CompareAndSwap"
// "CompareAndSet"
// "TestAndSet"
// "CompareExchange"
// ...
bool CAS(word *target,
         word oldvalue,
         word newvalue)
{
    if (*target == oldvalue)
    {
        *target = newvalue;
        return true;
    }
    return false;
}
```

Meet your new **Best Friend Forever**, CAS:

```
// "CompareAndSwap"
// "CompareAndSet"
// "TestAndSet"
// "CompareExchange"
// ...
bool CAS(word *target,
         word oldvalue,
         word newvalue)
{
    if (*target == oldvalue)
    {
        *target = newvalue;
        return true;
    }
    return false;
}
```

\*\*\*Atomically\*\*\*



*count++ is getting complicated...*

```
count++;
```

=>

```
register reg = read_memory(&count); // Read  
reg = reg + 1;                      // Modify  
write_memory(&count, reg);          // Write
```

=>

```
word atomic_inc(word & target)  
{  
    word oldval;  
    word newval;  
    do  
    {  
        oldval = target;    // Read  
        newval = oldval + 1; // Modify  
    }                      // Write or Retry:  
    while ( ! CAS(&target, oldval, newval) );  
  
    return newval;  
}
```

*Every vote counts! - If you miss your chance, retry.*

*CAS to the rescue – Concurrent Counting with CAS:*

```
// Thread 1
void thread1()
{
    int i = 100;
    while(i-->0)
        atomic_inc(&count);
}

// Thread 2
void thread2()
{
    int i = 100;
    while(i-->0)
        atomic_inc(&count);
}
```

*Result: count == 200. Every time.*

*Other tricks that CAS can do:*

```
do
{
    oldval = target;
    newval = (oldval + 5);
    newval = newval % 17;
    //...
}
while ( !CAS(&target, oldval, newval) );
```

*Atomic add 5 modulo 17. Amazing! (?)*

**Wait, I thought polling was Bad!...**

*(pun intended)*

*“Polling”: What's wrong with this code?*

```
while (!done)
    yield();
```

```
// or worse
while (!done)
    ;
```

*(Answer: lots.)*

*CAS-loop vs Polling. What's the difference?*

A: Spin/Poll

```
// check done;  
// or try again
```

```
while (!done)  
    yield();
```

```
// or worse  
while (!done)  
    ;
```

B: CAS Loop

```
// update target;  
// or try again
```

```
do  
{  
    old = target;  
    new = old + 1;  
}  
while(!CAS(&target, old, new));
```

*Answer: one is “lock-free”, the other isn't.*

*CAS-loop vs Polling. What's the difference?*

A: Spin/Poll

```
// check done;  
// or try again
```

```
while (!done)  
    yield();
```

```
// or worse  
while (!done)  
    ;
```

B: CAS Loop

```
// update target;  
// or try again
```

```
do  
{  
    old = target;  
    new = old + 1;  
}  
while (!CAS(&target, old, new));
```

*Answer: one is “lock-free”, the other isn't.*

Definition:

An algorithm is ***lock-free*** if at all times **at least one thread** is guaranteed to be **making progress**.

*(Herlihy)*

*Polling has no locks. Is it lock-free?*

Thread A	Thread B
<pre>while (!done)     ;</pre>	<pre>do_some_stuff(); more_stuff(); etc(); done = true;</pre>

*What happens when Thread B is paused by the scheduler?*

Definition:

An algorithm is ***lock-free*** if at all times **at least one thread** is guaranteed to be **making progress**.  
(Herlihy)

*Polling – Not lock-free :-)*



*CAS loop?*

Thread A	Thread B
do	do
{	{
old = target;	old = target;
new = old + 1;	new = old + 1;
}	}
while (!CAS (&target, old, new));	while (!CAS (&target, old, new));

*What happens when Thread B is paused by the scheduler?*

Definition:

An algorithm is ***lock-free*** if at all times **at least one thread** is guaranteed to be **making progress**.  
(Herlihy)

*CAS loop – Lock-free! :-)*

```
do
{
    old = target;
    new = old + 1;
}
while (!CAS(&target, old, new));
```

## Definitions:

An algorithm is ***lock-free*** if at all times **at least one thread** is guaranteed to be **making progress**.

*(Herlihy)*

An algorithm is ***wait-free*** if at all times **all threads** are guaranteed to be **making progress**.

*(Herlihy)*

```
do
{
    old = target;
    new = old + 1;
}
while (!CAS(&target, old, new));
```

## Definitions:

An algorithm is ***lock-free*** if at all times **at least one thread** is guaranteed to be **making progress**.

*(Herlihy)*

An algorithm is ***wait-free*** if at all times **all threads** are guaranteed to be **making progress**.

*(Herlihy)*

An algorithm is ***obstruction-free*** if it guarantees **progress** for any thread that **eventually executes in isolation**.

*(Herlihy)*

*I Am Not An Expert.*

```
do
{
    old = target;
    new = old + 1;
}
while (!CAS(&target, old, new));
```

## Definitions:

An algorithm is *wait-free* if at all times **all threads** are guaranteed to be **making progress**.

(Herlihy)

An algorithm is *lock-free* if at all times **at least one thread** is guaranteed to be **making progress**.

(Herlihy)

An algorithm is *obstruction-free* if it guarantees **progress** for any thread that **eventually executes in isolation**.

(Herlihy)

*What happens when  $\text{priority}(\text{Thread } A) > \text{priority}(\text{Thread } B)$ ?*

Thread A	Thread B
<pre>while (!done)     ;</pre>	<pre>do_some_stuff(); more_stuff(); etc(); done = true;</pre>

Definition:

***Priority inversion*** occurs whenever a high-priority thread must wait for a low-priority thread.

*What happens when  $\text{priority}(\text{Thread } A) > \text{priority}(\text{Thread } B)$ ?*

Thread A	Thread B
do	do
{	{
old = target;	old = target;
new = old + 1;	new = old + 1;
}	}
while(!CAS(&target, old, new));	while(!CAS(&target, old, new));

*Lock-free => no priority inversion :-)*

*Spinning isn't always bad:*

```
void spin_lock(word & lock)
{
    while (XCHG(&lock, 1))
        ;
}

void spin_unlock(word & lock)
{
    XCHG(&lock, 0);
}
```

*Use sparingly - Everything in moderation.*



*Sneak Peek:*

```
void spin_lock(word & lock)
{
    while (lock || XCHG(&lock, 1))
        ;
}

void spin_unlock(word & lock)
{
    XCHG(&lock, 0);
}
```

*'Naked' reads are faster, but less useful.*

**Let's get *Started*...**

*What's wrong with this code?*

```
void someFunc()  
{  
    static MyCriticalSection cs();  
  
    cs.lock();  
  
    do_important_stuff();  
  
    cs.unlock();  
}
```

*The CriticalSection is initialized... when?*

“Use Locks!”

“Use ~~Locks~~ Boost!”

"What 99 percent of programmers need to know..."

*boost::threads::call\_once() to the rescue:*

```
void someFunc()
{
    static MyCriticalSection cs;

    static once_flag once = BOOST_ONCE_INIT;
    call_once(once, bind(cs::init, cs));

    cs.lock();

    do_important_stuff();

    cs.unlock();
}
```

*(Still not 100% C++)*

**Everything I need to know about lock-free I learned from `call_once()`...**

***vs***

**DCLP – The Double-Checked Locking Pattern**

**“Globals are Bad.”**



“Globals are Bad.”

“Singletons are Bad.”

“Globals are Bad.”

“Singletons are Bad.”

“Use Locks.”

“Globals are Bad.”

“Singletons are Bad.”

“Use Locks.”

“MACROS are EVIL.”

“Globals are Bad.”

“Singletons are Bad.”

“Use Locks.”

“MACROS are EVIL.”

“I'm overgeneralizing.”

*Lazy Singleton Initialization with DCLP:  
What's wrong with this code?*

```
static Singleton singleton; //POD
static Mutex mutex;
static bool done_yet = false;

void lazy_init()
{
    // locking/unlocking mutex is expensive***,
    // so check flag first before using mutex
    if (!done_yet)
    {
        // now use a mutex:
        mutex.lock();
        if (!done_yet) // double-check!
        {
            singleton.important_ptr = new Important();
            singleton.moredata = etc();
            done_yet = true;
        }
        mutex.unlock();
    }
}
// ***not really
```

*See Meyers' "Effective C++" book, or Alexandrescu's "Loki" library*

“Use Locks.” - close the 'if...then' gap ✓

“Use Locks.” - close the 'if...then' gap ✓

“Premature Optimizations...”

*Lazy Singleton Initialization with DCLP:  
What's wrong with this code?*

```
static Singleton singleton; //POD
static Mutex mutex;
static bool done_yet = false;

void lazy_init()
{
    // locking/unlocking mutex is expensive***,
    // so check flag first before using mutex
    if (!done_yet)
    {
        // now use a mutex:
        mutex.lock();
        if (!done_yet) // double-check!
        {
            singleton.important_ptr = new Important();
            singleton.moredata = etc();
            done_yet = true;
        }
        mutex.unlock();
    }
}
// ***not really
```

*See Meyers' "Effective C++" book, or Alexandrescu's "Loki" library*



*Speaking of Meyers and Alexandrescu...*

"C++ and the Perils of Double-Checked Locking"

Meyers, Scott and Alexandrescu, Andrei, Dr. Dobb's Journal,  
September 2004.

([http://www.aristeia.com/Papers/DDJ\\_Jul\\_Aug\\_2004\\_revised.pdf](http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf))

*The **Perils** of DCLP:*

*What's wrong with this code?*

```
static Singleton singleton; //POD
static Mutex mutex;
static bool done_yet = false;

void lazy_init()
{
    if (!done_yet)
    {
        mutex.lock();
        if (!done_yet) // double-check!
        {
            singleton.important_ptr = new Important();
            singleton.moredata = etc();
            done_yet = true;
        }
        mutex.unlock();
    }
}
```

*Answer: Memory Barriers – it needs some...*

*Imagine, if you will...*

Thread A:

```
    lazy_init();  
    int x = singleton.important_ptr->foo;
```

Thread B:

```
    lazy_init();  
    int y = singleton.important_ptr->foo;
```

*And even imagine that it happens in the above order. ie ThreadA completes before Thread B calls lazy\_init().*

*And lastly, imagine that*

*Thread B crashes because `important_ptr == null`.*

*Imagine, if you will...*

Thread A:

```
    lazy_init();  
    int x = singleton.important_ptr->foo;
```

Thread B:

```
    lazy_init();  
    int y = singleton.important_ptr->foo; // BOOM!!
```

*And even imagine that it happens in the above order. ie ThreadA completes before Thread B calls lazy\_init().*

*And lastly, imagine that*

*Thread B crashes because important\_ptr == null.*

*This is impossible. What's going on?*

*This is impossible. What's going on?*

*This is impossible. What's going on?*

Aha! It's the cache's fault. My variables are 'stale' and being read from the cache, when really the right value was in main memory all along!

*Cache Coherency:*

Aha! It's the cache's fault. My variables are 'stale' and being read from the cache, when really the right value was in main memory all along!

*Wrong. Most CPUs implement cache-coherency.  
Caches communicate with each other to know when a value is stale.  
(google “MESI protocol”)*

*This is impossible. What's going on?*

Thread A:

```
    lazy_init();  
    int x = singleton.important_ptr->foo;
```

Thread B:

```
    lazy_init();  
    int y = singleton.important_ptr->foo; // BOOM!!
```

=>



*A closer look:*

Thread A:

```
    singleton.important_ptr = new Important();  
    done_yet = true;  
    mutex.unlock();  
    int x = singleton.important_ptr->foo;
```

Thread B:

```
    read(done_yet);  
    int y = singleton.important_ptr->foo; // BOOM!!
```

=>

*A closer look:*

Thread A:

```
write(singleton.important_ptr);  
write(done_yet);  
mutex.unlock();  
int x = singleton.important_ptr->foo;
```

Thread B:

```
read(done_yet);  
read(singleton.important_ptr);           // BOOM!!
```

=>

*A closer look:*

Thread A:

```
    ↪ write(done_yet);  
    ↪ write singleton.important_ptr);  
    mutex.unlock();  
    int x = singleton.important_ptr->foo;
```



Thread B:

```
    ↪ read(singleton.important_ptr);  
    ↪ read(done_yet);
```



// **BOOM!!**

*A closer look:*

Thread A:

```
 write(done_yet) ;  
 write singleton.important_ptr ;  
mutex.unlock();  
int x = singleton.important_ptr->foo;
```

Thread B:

```
 read(singleton.important_ptr) ;           // BOOM!!  
 read(done_yet) ;
```

*The compiler is reordering instructions?*

*A closer look:*

Thread A:

```
    write(done_yet);  
    write.singleton.important_ptr);  
    mutex.unlock();  
    int x = singleton.important_ptr->foo;
```

Thread B:

```
    read.singleton.important_ptr);           // BOOM!!  
    read(done_yet);
```

*The compiler is reordering instructions?*

*The CPU is reordering instructions.*

*CPU and compiler optimization/reordering rule:*

Do any optimizations you'd like, as long as it doesn't change how the program works.

*CPU and compiler optimization/reordering rule:*

Do any optimizations you'd like, as long as it doesn't change how the program works.\*

\*.....assuming a single-threaded program.

**How CPUs Work...\***

**\* (sort of)**



*That was Then:*

speed of RAM == speed of CPU

*That was Then:*

speed of RAM == speed of CPU

*This is Now:*

speed of RAM <<< speed of CPU

*What's a CPU to do?*

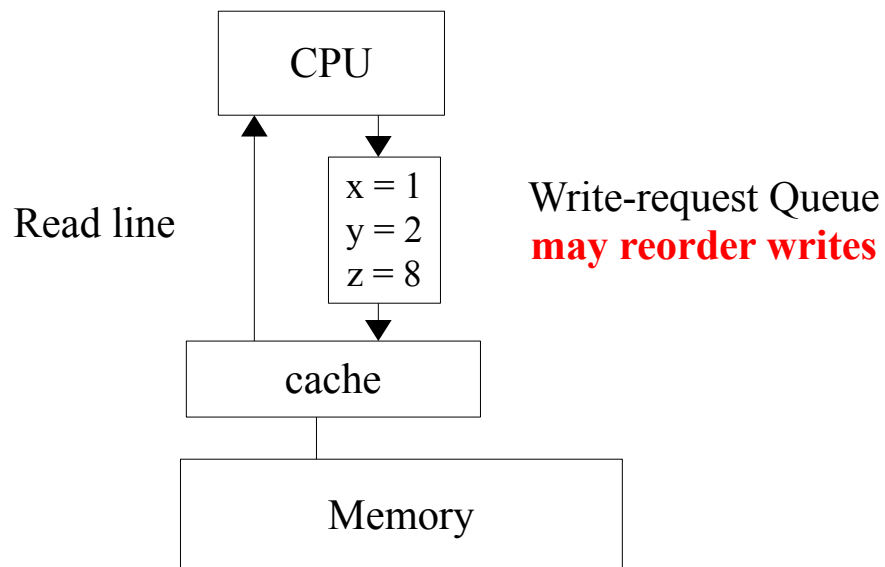
If a CPU waits for a memory operation to complete, **100s** of cycles can go by.

*What's a CPU to do?*

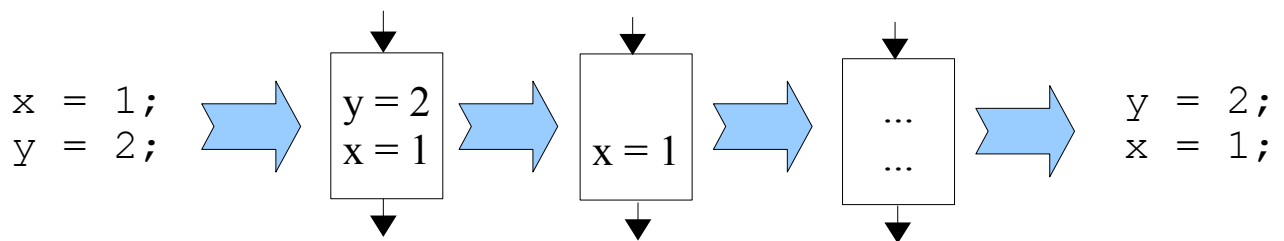
If a CPU waits for a memory operation to complete, **100s** of cycles can go by.

*Not wait.*

*The Write Request Queue:*



*Reordered Writes:*



*Writes “performed” out of order.*

## *What about Reading?*

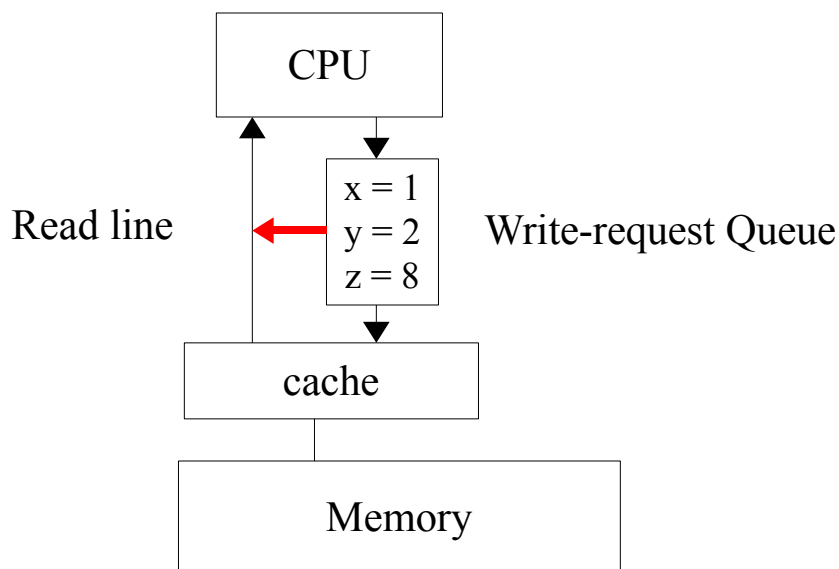
1. Read ahead:

```
x = 1; // why not start z here  
y = 2;  
r = z;
```

2. “Speculative Execution”

```
if (x != 0) // read z  
    r = z;    // before x
```

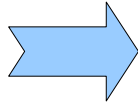
3. Write-queue peeking:



*Result: “Read-request Queue”  
Reads performed out of order.*

## *Intermix Reads and Writes*

```
x = 1;  
y = 2;  
r1 = z;  
r2 = w;
```



```
r2 = w;  
y = 2;  
r1 = z;  
x = 1;
```

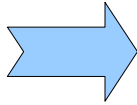
*In any order.*



## *Interdependent Operations*

"y = z"

```
r1 = z;  
y = r1;
```



```
r1 = z;  
y = r1;
```

*Cannot break Single-processor behaviour.*

*DCLP reminder:*

*As programmed:*

ThreadA

```
A1) ptr = new...;  
A2) done_yet = true;
```

ThreadB

```
B1) if (done_yet)  
B2)   read ptr
```

*“reordered” =>*

ThreadA

```
A2) done_yet = true;  
A1) ptr = new...;
```

ThreadB

```
B2)   read ptr  
B1) if (done_yet)
```

***Boom.***

*Why does DCLP work?*

*Why does DCLP work?*

1. Most singletons initialized early – and outside of threads.

*Why does DCLP work?*

1. Most singletons initialized early – and outside of threads.
2. Thread interleaving is course, not fine grained.

*Why does DCLP work?*

1. Most singletons initialized early – and outside of threads.
2. Thread interleaving is course, not fine grained.
3. Intel x86.

*Why does DCLP work?*

1. Most singletons initialized early – and outside of threads.
2. Thread interleaving is course, not fine grained.
3. Intel x86.

*It doesn't.*

*I “lied”. :-)*

*Imagine, if you will...*

Thread A:

```
    lazy_init();  
    int x = singleton.important_ptr->foo;
```

Thread B:

```
    lazy_init();  
    int y = singleton.important_ptr->foo; // BOOM!!
```

*And even imagine that it happens in the above order. ie ThreadA **completes before** Thread B calls lazy\_init().*

*And lastly, imagine that*

*Thread B crashes because important\_ptr == null.*

*...“completes”...*

*...“before”...*

*...”reodered”...*

*...“performed”...*



**What to do, What to do...**

“Use Locks!”

## *Prevent Reordering*

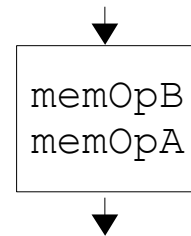
```
memOpA;  
memOpB;
```

## *Prevent Reordering*

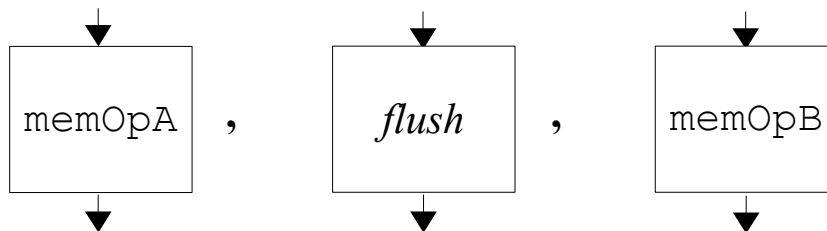
```
memOpA;  
----please-don't-reorder----  
memOpB;
```

*Prevent Reordering – eg Write Reordering*

memOpA;  
----*please-don't-reorder*----  
memOpB;

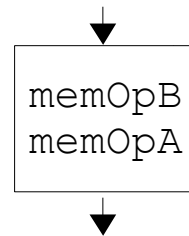


*Flush the Queue:*

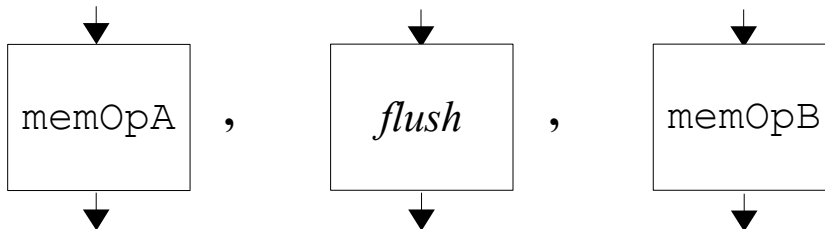


## *Prevent Reordering – eg Write Reordering*

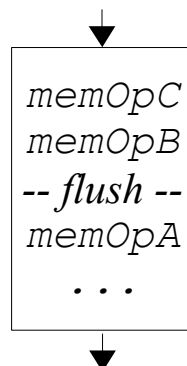
memOpA;  
----*please-don't-reorder*----  
memOpB;



## *Flush the Queue*



## *“Pseudo”-Flush the Queue*



*“Similarly” for Reads and Intermixing Reads and Writes.*

*If the processor lets us...*

“Use Locks!”



“Use Locks!”

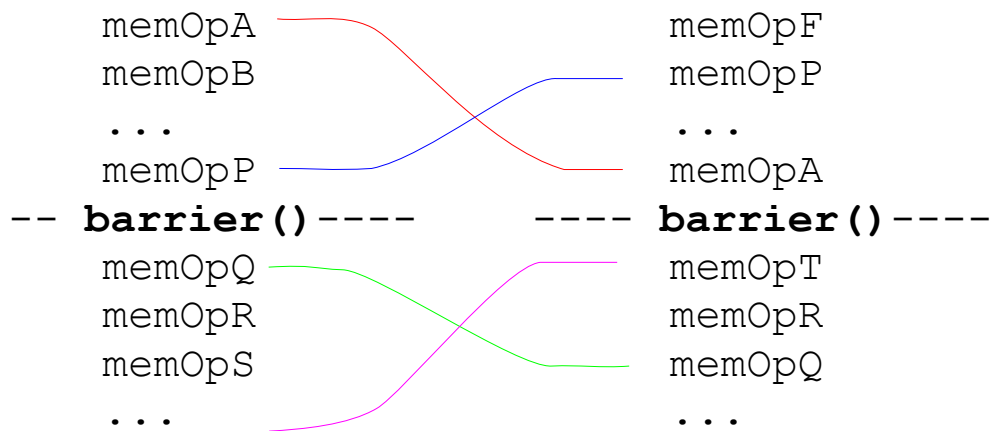
```
lock(mutex); // locked = true;  
    ptr = new Important();  
unlock(mutex); // locked = false;
```

*Does the locked flag get reordered with the ptr?  
What's the magic?*

*Memory Barriers - Yes the processor lets us.*

### General Concept of a Memory Barrier:

Instructions:      ->      Reordered by CPU:



*(It was the processor's fault in the first place.)*

*Memory Barriers – (at least) 2 types of Memory Operations...*

```
    memOpA    // a load (read) or a store (write)
---barrier()----
    memOpB    // a load (read) or a store (write)
```

*Memory Barriers – (at least) 2 types of Memory Operations...*

```
    memOpA    // a load (read) or a store (write)
---barrier()---
    memOpB    // a load (read) or a store (write)
```

*...means (at least) 4 combinations...*

loadOp	loadOp	storeOp	storeOp
-----	-----	-----	-----
loadOp	storeOp	loadOp	storeOp

*Memory Barriers – (at least) 2 types of Memory Operations...*

```
    memOpA    // a load (read) or a store (write)
---barrier()---
    memOpB    // a load (read) or a store (write)
```

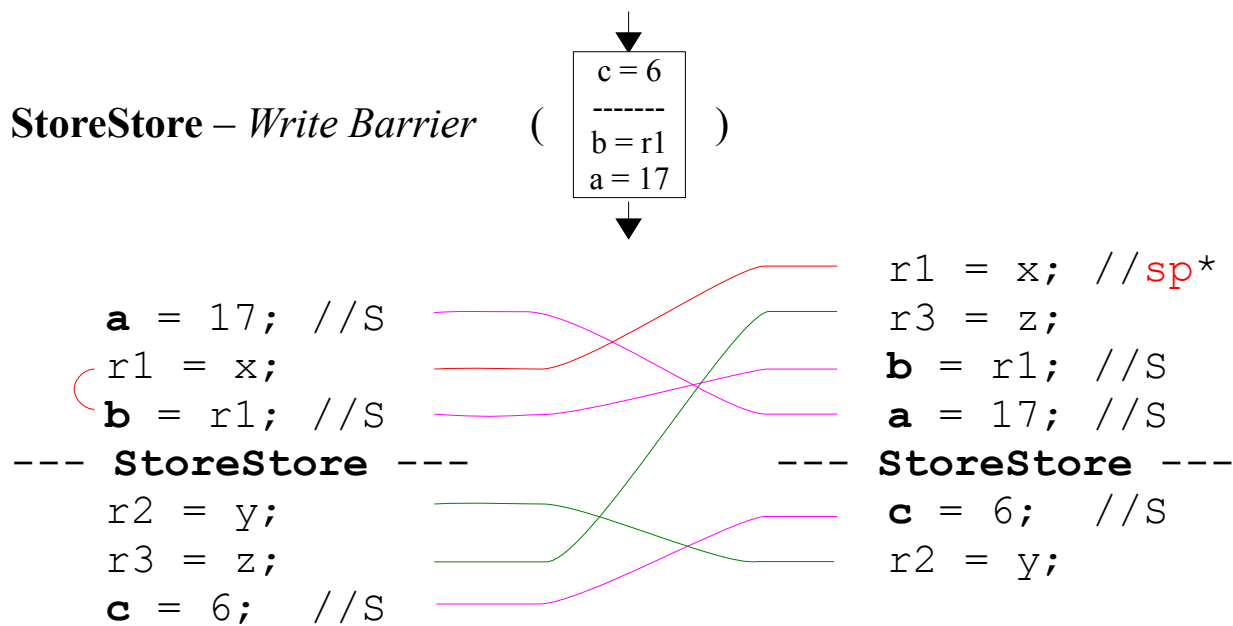
*...means (at least) 4 combinations...*

loadOp	loadOp	storeOp	storeOp
-----	-----	-----	-----
loadOp	storeOp	loadOp	storeOp

*...Means (at least) 4 types of (SPARC) Barriers:*

**Load|Load** - *prevents reordering of preceding loads with following loads*  
**Load|Store** - *prevents reordering of preceding loads with following stores*  
**Store|Store** - *prevents reordering of preceding stores with following stores*  
**Store|Load** - *prevents reordering of preceding stores with following loads*

(At least)



*“a and b must be written **before** c is written”*

- *Operations on a single side still free to reorder*
- *Loads can move freely*
- *Stores can't cross in either direction*
- *\*sp - single-processor correctness constraint*

## *StoreStore Uses*

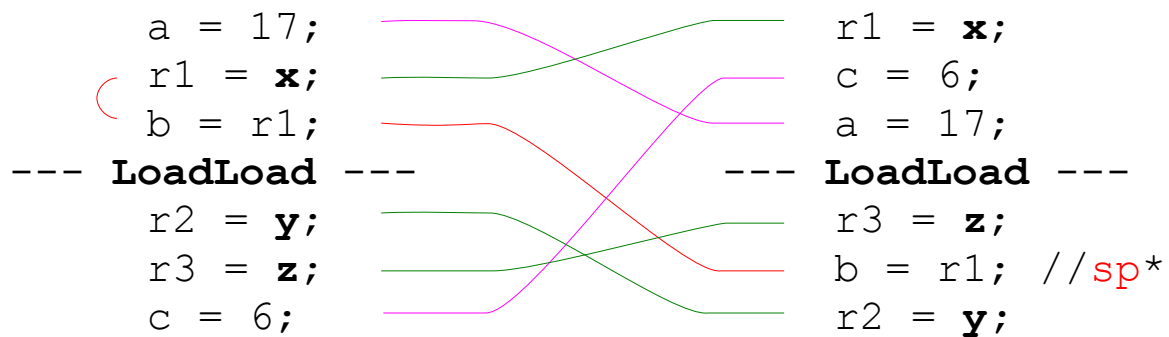
### enqueueing data

```
data.a = 17;  
data.b = x;  
// ensure data is written before queuing:  
#StoreStore  
queue.push(data); // <- does a write in push()
```

### initializing singletons

```
temp->a = 17;  
temp->b = x;  
// ensure data is written before being exposed:  
#StoreStore  
singleton.instance = temp;
```

## LoadLoad – Read Barrier



*“x must be read **before** y and z are read”*

- *Operations on a single side still free to reorder*
- *Stores can move freely*
- *Loads can't cross in either direction*
- *\*sp - single-processor correctness constraint*



## *LoadLoad Uses*

dequeuing data

```
if (queue.data_ready) //or wait until ready, etc
#LoadLoad
{
    data = queue.pop();    // read data!
}
```

reading from a singleton

```
if (singleton != null)
#LoadLoad
{
    read_from_singleton();
}
```

(Note that if you also plan to *write* to the singleton, then LoadLoad is insufficient. For now, let's just worry about reading...)

*LoadLoad mirrors StoreStore*

queues

*Publish*

```
data = ...;  
#StoreStore  
queue.push(data);
```

*Subscribe*

```
if (queue.data_ready)  
#LoadLoad  
{  
    data = queue.pop();  
}
```

singletons

*Publish*

```
Temp = new ...;  
#StoreStore  
singleton = temp;
```

*Subscribe*

```
if (singleton != null)  
#LoadLoad  
{  
    read_from_singleton();  
}
```

## *Publish / Subscribe*

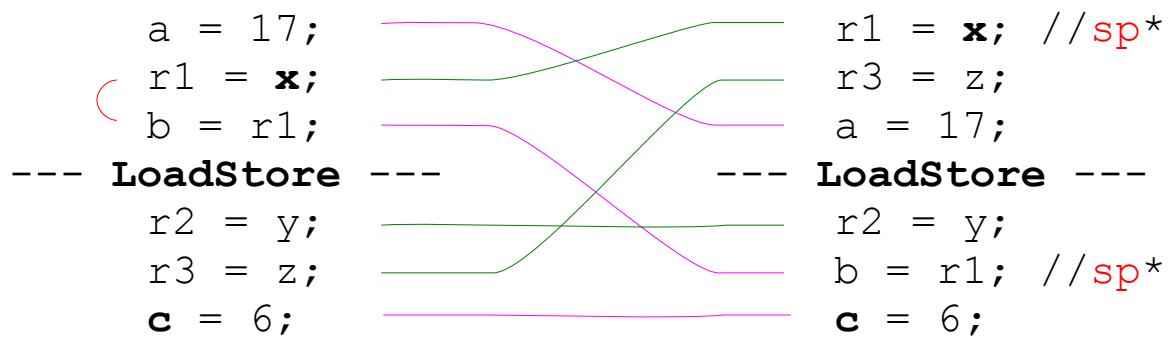
### *Publish:*

- Write Data
- WriteBarrier (StoreStore)
- Publish (ready\_flag)

### *Subscribe:*

- Check Subscription (ready\_flag)
- ReadBarrier (LoadLoad)
- Read Data

## LoadStore –



*“x must be **read** before c is **written**”*

- *Operations on a single side still free to reorder*
- *Each “trends” to its own side*
- *\*sp - single-processor correctness constraint*
- *Involves both Read and Write Queues - slower*

### *LoadStore Uses – not as common as Read/Write barriers*

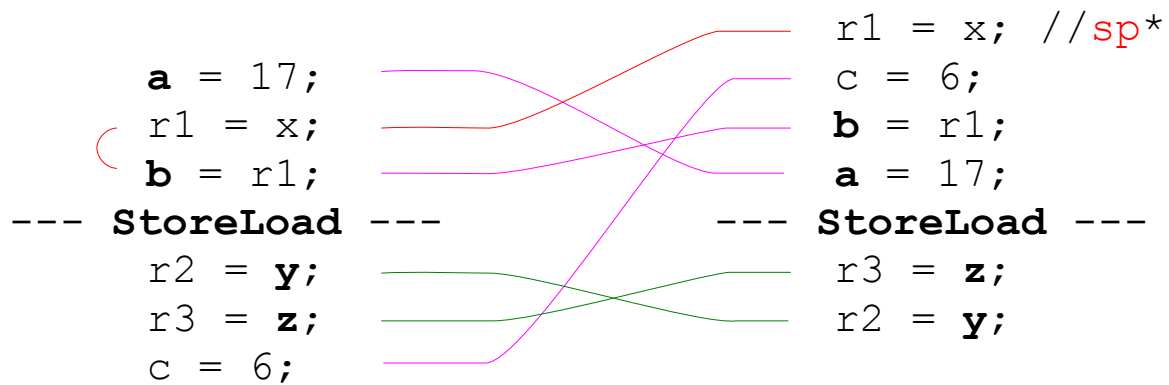
Make sure something is 'ready' before being written:

```
if (destination_ready)    // load
#LoadStore
{
    fill_destination();    // store
}
```

Or, make sure you finish reading before saying you are finished!

```
temp = important.data;    // Load
#LoadStore
important.in_use = false; // Store
```

## StoreLoad –



*“a and b must be **written** before y and z are **read**”*

- *Operations on a single side still free to reorder*
- *Each “trends” to its own side*
- *\*sp – note that r1 = x (load) is stuck on the store side*
- *Involves both Read and Write Queues - slower*

*StoreLoad Uses –*

*(useless)*

## *StoreLoad Uses – Rudeness*

Mirror of LoadStore 'in use'

(recall:

```
temp = important.data;    // Load
#LoadStore
important.in_use = false; // Store
```

)

```
important.in_use = true; // Store
#StoreLoad
temp = important.data;    // Load
```

*How Rude!*



“Use Locks!”

*Starting to look a lot like Locks*

```
LoadStore:    if (ready) read...  
LoadLoad:    if (ready) write...
```

*Combined:*

```
if (foo.ready)  
#LoadLoad  
#LoadStore  
{  
    // free to both read and write foo  
    int x = foo.x;  
    foo.a  = 10;  
    etc(foo);  
}
```

*Lock-ish,  
But only check, no set...*

## *Return of the CAS – Check **and** Set Flag*

```
if (CAS(foo.lock, 0, 1)) // "was 0, now 1" == ours
#LoadLoad
#LoadStore
{
    // free to both read and write foo
    int x = foo.x;
    foo.a  = 10;
    etc(foo);
}
else
    do_something_else();
```

*More Lockish.  
But wait, CAS is a Load and a Store...*

## *Return of the CAS – Check **and** Set Flag*

```
if (CAS(foo.lock, 0, 1)) // "was 0, now 1" == ours
#LoadLoad
#LoadStore
{
    // free to both read and write foo
    int x = foo.x;
    foo.a  = 10;
    etc(foo);
}
else
    do_something_else();
```

*More Lockish.*  
*But wait, CAS is a Load and a Store...*  
**\*\*\**Atomically*\*\*\***

## *Revenge of the Rudeness – Clear the Flag when done*

```
if (CAS(foo.lock, 0, 1)) // "was 0, now 1" == ours
#LoadLoad
#LoadStore
{
    // free to both read and write foo
    int x = foo.x;
    foo.a = 10;
    etc(foo);

    // OK, done with foo
    #LoadStore        // make sure all loads are done
    #StoreStore       // make sure all stores are done
    foo.lock = false; // and make sure this is last
}
```

***Very Lockish.***

*Where's StoreLoad? Where's the Mirror?*

*Where's StoreLoad? Where's the Mirror?*

Load|Load  
+ Load|Store == Load|XXX

Load|Store  
+ Store|Store == XXX|Store

*Mirror*

LoadXXX  $\diamond$  XXXStore

Check lock  
LoadXXX;  
 $\diamond$   
XXXStore;  
Clear lock

*Excellent!...*

StoreLoad is the slowest barrier.

*(correlation/causation?)*

*A Different Kind of Company, a Different Kind of Barrier...*

SPARC – LoadLoad, LoadStore, ...

Intel – **Acquire / Release**

From MSDN

([http://msdn.microsoft.com/en-us/library/ms684122\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms684122(VS.85).aspx)):

“Acquire memory semantics specify that the memory operation being performed by the current thread will be visible before any other memory operations are attempted. Release memory semantics specify that the memory operation being performed by the current thread will be visible after all other memory operations have been completed.”

“Visisble”  $\sim$  Completed. *Visible to other threads/processors.*

*Cannot open the computer case and look at the memory!*

*“...memory operation being performed...” - Lockish Acquire / Release*

```
if (CAS_acquire(foo.lock, 0, 1))
{
    // free to both read and write foo
    int x = foo.x;
    foo.a  = 10;
    etc(foo);

    // OK, done with foo
    Write_release(foo.lock, false);
}
```

Acquire – all CPUs have `foo.lock = true` available  
before *this CPU does any other instructions*

Release – *this CPU's* instructions will be completed before  
`foo.lock == false` is completed.

\*\*\* Must be paired and used consistently! \*\*\*



## *Hard Acquire / Release*

“Acquire memory semantics specify that the memory operation being performed by the current thread will be visible before any other memory operations are attempted. Release memory semantics specify that the memory operation being performed by the current thread will be visible after all other memory operations have been completed.”

## *Easy Acquire / Release*

Acquire – after means after

Release – before means before

```
Acquire(foo.lock);  
    int x = foo.x;      // These instructions  
    foo.a  = 10;        // can't escape.  
    etc(foo);           // Boxed in!  
Release(foo.lock);
```

*First time before a live audience:*

*Relating Acquire/Release to Load/Store:*

Acquire/Release semantics

As Load/Store barriers

`Read_WithAcquire(lock);`

`Read(lock);`

`#LoadLoad`

`#LoadStore`

`Write_WithRelease(lock,  
false)`

`#LoadStore`

`#StoreStore`

`Write(lock, false);`

*Are they Equivalent?*

## *Locks - An Invasion from Above and Below!?*

*Question: was `unshared.x` written **(2)** before `unshared.y` was read **(3)** ?  
Was `unshared.x` written **(2)** before `unshared.y` was written **(4)**?  
(And why not ask about **(1)**?)*

```
int x = unshared.x;    // 1
unshared.x = 10;       // 2
mutex.lock();
    do_stuff(shared_data);
mutex.unlock();
int y = unshared.y;    // 3
unshared.y = 20;       // 4
```

## *Locks - An Invasion from Above and Below!?*

*Question: was `unshared.x` written (2) **before** `unshared.y` was read (3) ?  
Was `unshared.x` written (2) **before** `unshared.y` was written (4)?*

```
int x = unshared.x;    // 1
unshared.x = 10;       // 2
mutex.lock();
    do_stuff(shared_data);
mutex.unlock();
int y = unshared.y;    // 3
unshared.y = 20;       // 4
```

*Answer: Who cares?*

*Translating...*

*Question: was unshared.x written (2) **before** unshared.y was read (3) ?  
Was unshared.x written (2) **before** unshared.y was written (4)?*

```
int x = LOAD(unshared.x);      // 1
STORE(unshared.x, 10);         // 2
CAS_or_wait(mutex.lock, 0, 1);
#LoadLoad
#LoadStore
    LOAD_andor_STORE(shared_data);
#LoadStore
#StoreStore
STORE(mutex.lock, 0);
int y = LOAD(unshared.y);      // 3
STORE(unshared.y, 20);         // 4
```

*Reorder...*

*Question: was unshared.x written (2) before unshared.y was read (3) ?  
Was unshared.x written (2) before unshared.y was written (4)?*

```
CAS_or_wait(mutex.lock, 0, 1);  
int x = LOAD(unshared.x);      // 1  
#LoadLoad  
int y = LOAD(unshared.y);      // 3  
#LoadStore  
    LOAD_andor_STORE(shared_data);  
#LoadStore  
STORE(unshared.x, 10);         // 2  
#StoreStore  
STORE(unshared.y, 20);         // 4  
STORE(mutex.lock, 0);
```

*Untranslate..*

*Question: was unshared.x written (2) **before** unshared.y was read (3) ?  
Was unshared.x written (2) **before** unshared.y was written (4)?*

```
mutex.lock();  
    int x = LOAD(unshared.x);      // 1  
    int y = LOAD(unshared.y);      // 3  
    LOAD_andor_STORE(shared_data);  
    STORE(unshared.x, 10);         // 2  
    STORE(unshared.y, 20);         // 4  
mutex.unlock();
```

*Instructions from above and below, moved into the lock!  
(But no instructions exited the lock, so we got that going for us,  
which is nice.)*



## *Locks – Acquire/Release version*

*Question: was unshared.x written (2) **before** unshared.y was read (3) ?  
Was unshared.x written (2) **before** unshared.y was written (4)?*

```
int x = unshared.x;    // 1
unshared.x = 10;       // 2
mutex.lock();          // Acquire - after means after
    do_stuff(shared_data);
mutex.unlock();        // Release - before means before
int y = unshared.y;    // 3
unshared.y = 20;       // 4
```

*Locks – Acquire/Release - allow greater reordering*

*Question: was unshared.x written (2) **before** unshared.y was read (3) ?  
Was unshared.x written (2) **before** unshared.y was written (4)?*

```
mutex.lock();    // Acquire - after means after
    int y = unshared.y; // 3
    unshared.y = 20;    // 4
    do_stuff(shared_data);
    int x = unshared.x; // 1
    unshared.x = 10;    // 2
mutex.unlock(); // Release - before means before
```

*What's was the Question?*

*Answer: DCLP...*

*The **Perils** of DCLP:*

*What's wrong with this code?*

```
static Singleton singleton; //POD
static Mutex mutex;
static bool done_yet = false;

void lazy_init()
{
    if (!done_yet)
    {
        mutex.lock();
        if (!done_yet) // double-check!
        {
            singleton.important_ptr = new Important();
            singleton.moredata = etc();
            done_yet = true;
        }
        mutex.unlock();
    }
}
```

*Answer: Memory Barriers – it needs some...*

*The **Perils** of DCLP:*

*What's wrong with this code?*

```
static Singleton singleton; //POD
static Mutex mutex;
static bool done_yet = false;

void lazy_init()
{
    // it's the done_yet == true case
    // that needs the acquire
    if (!read_acquire(done_yet))    // barrier
    {
        mutex.lock(); // (barrier)
        if (!done_yet)
        {
            singleton.important_ptr = new Important();
            singleton.moredata = etc();
            #StoreStore //those stores before this one
            done_yet = true;
        }
        mutex.unlock(); // another barrier!
    }
}
```

*Answer: Nothing.*

*Correction: Too complicated.*

“Premature optimization....”

“Use Locks!”

"What 99 percent of programmers need to know is not how to build components but how to use them."

*What about that 1% ?....*

```

static Singleton singleton; //POD
static Mutex mutex;
static bool done_yet = false;

void lazy_init()
{
    // it's the done_yet == true case
    // that needs the acquire
    if (!read_acquire(done_yet))    // barrier
    {
        mutex.lock(); // (barrier)
        if (!done_yet)
        {
            singleton.important_ptr = new Important();
            singleton.moredata = etc();
            #StoreStore //those stores before this one
            done_yet = true;
        }
        mutex.unlock(); // another barrier!
    }
}

```

*How about an lazy\_init() / call\_once() with:*

- minimum barriers (2 for 1<sup>st</sup> thread, 1 for rest if no contention)
- minimum resource usage:
  - single word for tracking state
  - no static mutex
  - no dynamic mutex unless contention
- no leaks – nothing leftover after init



```

void call_once(once_flag & flag, vcall & func, bool to_completion, bool
allow_recursion)
{
    once_node node;

    static_assert(sizeof(flag.flag) == sizeof(&node));

    // try to be first node in (ie winner)
    // if we fail, we still get an atomic read,
    // which we need anyway
    // must have *acquire* memory ordering to ensure that all data protected by the
once_flag is visible (particularly in the done case)
    once_flag::atomic_flag oldflag = atomic::CAS(flag.flag, 0,
add_winner_bit(&node));

    if (oldflag == once_done)
    {
        return;
    }

    while (oldflag != once_done)
    {
        if (oldflag == 0) // firsttime!
        {
            // we have a winner! we're first in, go for it

            try
            {
                func.call();
            }
            catch (...)
            {
                // we have some cleanup to do before rethrow
                post_winner_cleanup(flag, !to_completion); // called, but didn't
complete
                throw;
            }

            post_winner_cleanup(flag, true); // done, and complete
            return;
        }
        else
        {
            // flag/oldflag is already a pointer to a once_node.
            // get in line!
            if (has_winner_bit(oldflag))
            {
                node.count = 1; // preset this in case we are the first waiter,
otherwise it can be anything
                // oldflag is node from first thread in,

```

```

        // so we *might be* the second thread in, if we get in line soon
enough.
        // All waiters will then wait on our done event.
        // NOTE that we need to have the event created *before* we get in
line,
        // so that it is already there for additional waiters if they are
right behind us.
        node.done = CreateEvent(0, true, false, 0); // manual reset,
initial state is non-signaled
    }

    // this is just like the classic push onto a lockfree stack
do
{
    node.next = (once_node *)oldflag;
    oldflag = atomic::CAS(flag.flag, oldflag, &node); // if it hasn't
changed yet, swap us into the line
}
    while (oldflag != (once_flag::atomic_flag)node.next); // ie if another
thread pushed while we were trying to

    if (has_winner_bit(oldflag))
    {
        // we made it as the second thread in!

        // we now just need to wait for the done event to be signaled (by
the winner thread - probably)
        // note also that all other threads coming in behind us will wait
on this as well
        WaitForSingleObject(node.done, INFINITE);

        // OK, signalled. So now we are back...

        // if more than just this thread was waiting,
        // then all the other waiters were/are reading this node (on our
stack!) and/or waiting on *our* event
        // so we can't leave (trashing our stack) nor cleanup the done
event (while other threads are just about to wait on it).
        // Instead, rest assured that those other threads will let us
know when they are done:

        // we keep track of number of waiters left (which included us):
        atomic::uint_t count = atomic::decrement(node.count);
        if (count > 0)
        {
            // other threads still waiting on done!
            // we can't leave (nor clean up handles) until they are all
finished waiting.
            assert(node.cleanup != NULL); // a later waiter should have
set this up for us

```

```

        WaitForSingleObject(node.cleanup, INFINITE);
    }
}
else
{
    // we are not the first, nor the second thread in

    if (node.done)
    {
        // we had created this thinking we might become the first
waiter, but we lost the race :-(
        CloseHandle(node.done);
        node.done = 0;
    }

    // now, walk the chain to the end,
    // wait on the event you find there
    // Note that by this time, the call_once 'winner' thread might be
done,
    // and already be trying to walk the chain to wake us up,
    // even though we haven't got there yet.
    // Luckily it is careful not to allow anyone in the chain to
leave before we get to the end.
    once_node *endwaiter = node.next;
    while (!has_winner_bit(endwaiter->next))
        endwaiter = endwaiter->next;
    // now at the one-before-winner, or end-waiter

    // since we are the 2nd or nth thread waiting,
    // we need to tell the first-waiter (endwaiter, the owner of the
end of the chain)
    // that it can't leave until every other waiter leaves.
    // We do this with the endwaiter->cleanup event handle.
    // ** NOTE that we set up the cleanup event BEFORE incrementing
endwaiter->count (farther below),
    // so that endwaiter can't wake up on 'done' and find
endwaiter->cleanup missing. **

    // since we don't know whether we are 2nd or nth,
    // we attempt to Create the cleanup event and race to set end-
>cleanup
    // if we don't get there first, that's fine, just throw ours away
and use what is there.
    {
        HANDLE cleanup = atomic::read(endwaiter->cleanup);    //
this is atomic so we don't get a very stale read of our view of another thread's
leftover stack
        if (!cleanup)                                         // we
could avoid the first atomic::read, and just always attempt the CAS,
        cleanup = CreateEvent(0, false, false, 0);          //

```

but it is probably best to avoid the CreateEvent call at the expense of an additional atomic op

```
        HANDLE prev_cleanup = atomic::CAS(endwaiter->cleanup, 0,
cleanup);
        if (prev_cleanup != 0) // if we didn't get our handle in
there
            CloseHandle(cleanup);
    }

    // the 'winning' thread (that did the func call), when done
    // 1. removes the chain from the once_flag, so it is no longer
visible,
    //    and cannot be added to.
    // 2. walks the chain, counting the number of 'waiters' in the
chain.
    //
    // if it finds that count is the same as the number of nodes in
the chain,
    // it knows everyone is on the end of the chain (maybe not yet
waiting, but on the end)
    // so all OTHER chained threads except the last one can be
awakened and let go (just go, no need to 'pop' from the list);
    // only the end node needs to stick around to clean up (the end
node struct is in the STACK space of
    // the thread, so the thread can't leave until all other threads
are done with the node).
    //
    // Now, even worse, if a 'waiter' is still walking the chain when
'winner' finishes,
    // *none* of the threads in the chain can leave, until all
walkers get to the end.
    // We track this via endwaiter->count. Walker/waiters increment
the low-half of count,
    // the 'winner' thread sets the true count in the high-half of
the count.
    // This way either side can see the true state.

    atomic::uint_t count = atomic::increment(endwaiter->count); // we
only increment 'bottom-half' of count (ie assume number of waiters doesn't overflow
bottom half of an int)

    if (high_half(count) == low_half(count))
    {
        // 'winner' thread stopped by and saw how slow we are/were;
        // it set the true_count in the high-half (otherwise 0) and
then impatiently left - without signalling done!
        // Now that we've finally incremented the low-half to catch
up to the high-half,
        // that means we must be the last walker/waiter, and it is
up to us to flag the done event
```

```

        // we can write end->count NON-atomically because there are
no other threads left to touch it
        // and the SetEvent() does the proper memory barriers for
us (we assume - else most of Windows would be broken)

        endwaiter->count = low_half(count) - 1; // we don't need to
wait ourselves, so don't count this thread
        SetEvent(endwaiter->done); // do the job that 'winner'
thread left for us!
    }
    else
    {
        // well, we've added ourselves to the count,
        // looks like the winner thread hasn't been here (or there
are more slow-pokes yet to come);
        // nothing left to do but wait on the done event
        WaitForSingleObject(endwaiter->done, INFINITE);

        // and now we are back.
        // decr the count, see if we are last ones left,
        // if so, turn out the lights. Or, actually, notify the
waiting end-node thread that it can now turn out the lights (ie cleanup)
        count = atomic::decrement(endwaiter->count);
        if (count == 0)
        {
            // we were the last one waiting on done.
            // But end-node is still waiting to clean up!
            SetEvent(endwaiter->cleanup);
        }
    }
}
}
}
}
}
}
}
}
}
}

```

“Use Locks!”

*One more scare tactic – The DEC Alpha and dependent loads:*

```
int r1 = *p;
```

*This is 2 separate loads, the second depending on the first:*

```
int* ptr = read(p);  
int r1 = read_at(ptr);
```

In essence, *The Alpha processor can **reorder** these 2 reads.*

*Obviously this is “impossible”.*

- “address prediction” ?
- write-queue peeking ?
- ???
- 2 caches *per processor* – even lines / odd lines
- cache *update* queues
- loose definition of cache coherency

*Yikes*

“Use Locks!”

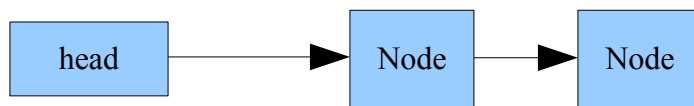


**And Now For Something *A Little Bit* Different...**

“Use Locks!”

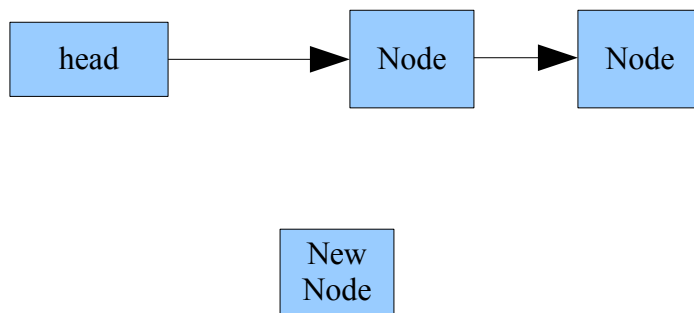
*CAS – with barriers assumed (store/load/release/acquire/whatever)*

*“Easy” - Push a Value onto a Lock-free Stack*



```
void push(Val val)
{
}
```

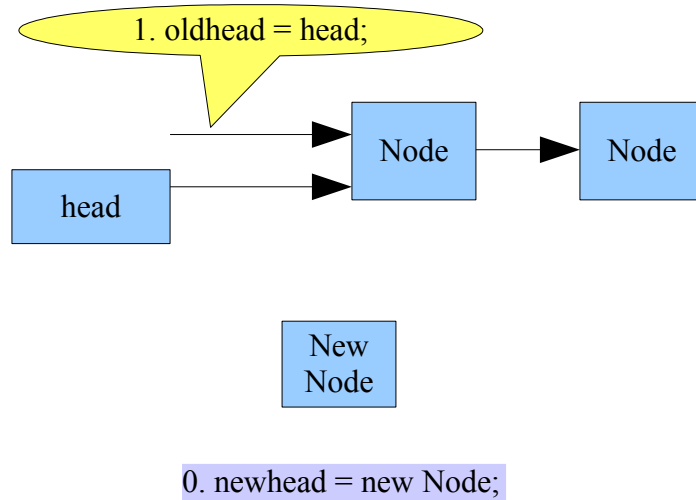
*“Easy” - Push a Value onto a Lock-free Stack*



```
0. newhead = new Node;
```

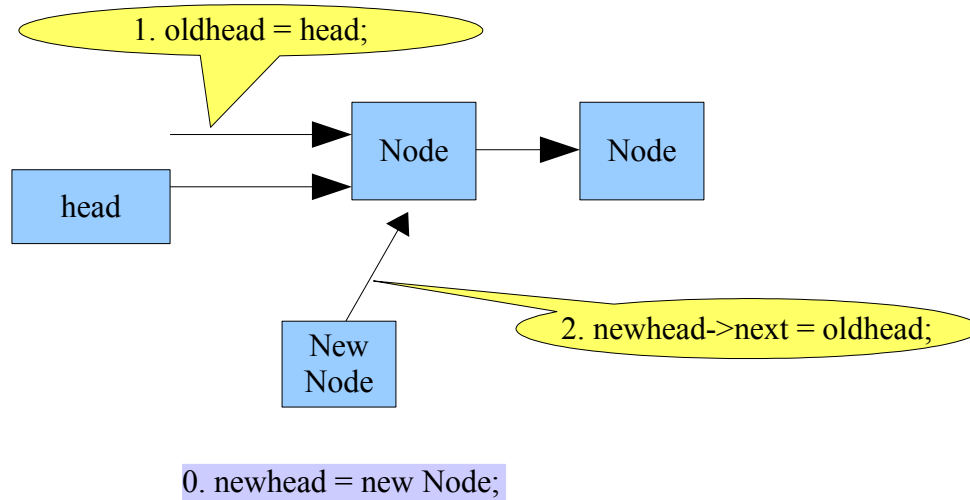
```
void push(Val val)
{
    Node * newhead = new Node(val);    // 0
}
```

*“Easy” - Push a Value onto a Lock-free Stack*



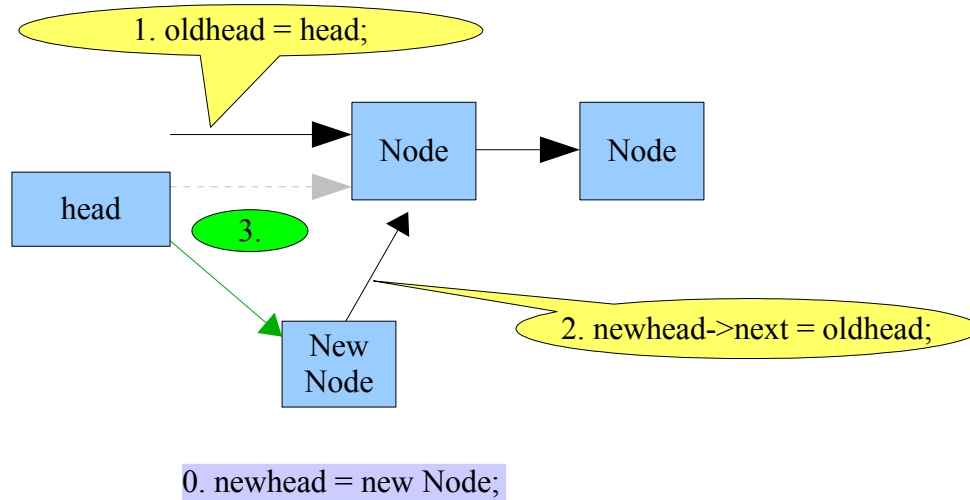
```
void push(Val val)
{
    Node * newhead = new Node(val);    // 0
    Node * oldhead = stack.head;       // 1 Read
}
```

*“Easy” - Push a Value onto a Lock-free Stack*



```
void push(Val val)
{
    Node * newhead = new Node(val);    // 0
    Node * oldhead = stack.head;       // 1 Read
    newhead->next = oldhead;            // 2 Modify
}
```

*“Easy” - Push a Value onto a Lock-free Stack*

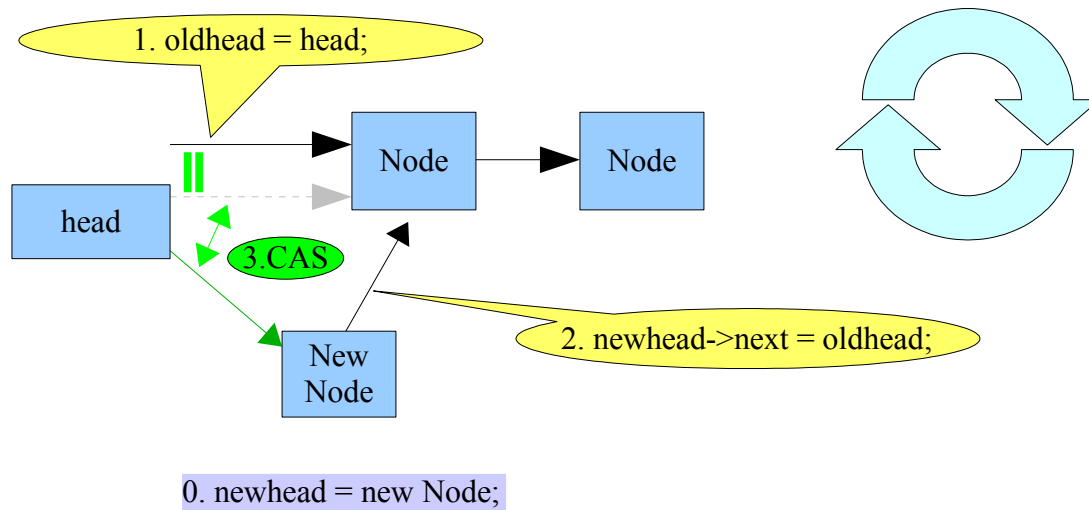


```
void push(Val val)
{
    Node * newhead = new Node(val);    // 0
    Node * oldhead = stack.head;        // 1 Read
    newhead->next = oldhead;            // 2 Modify
    stack.head = newhead;               // 3 Write
}
```

*Write, right?*  
*Wrong.*



*“Easy” - Push a Value onto a Lock-free Stack*



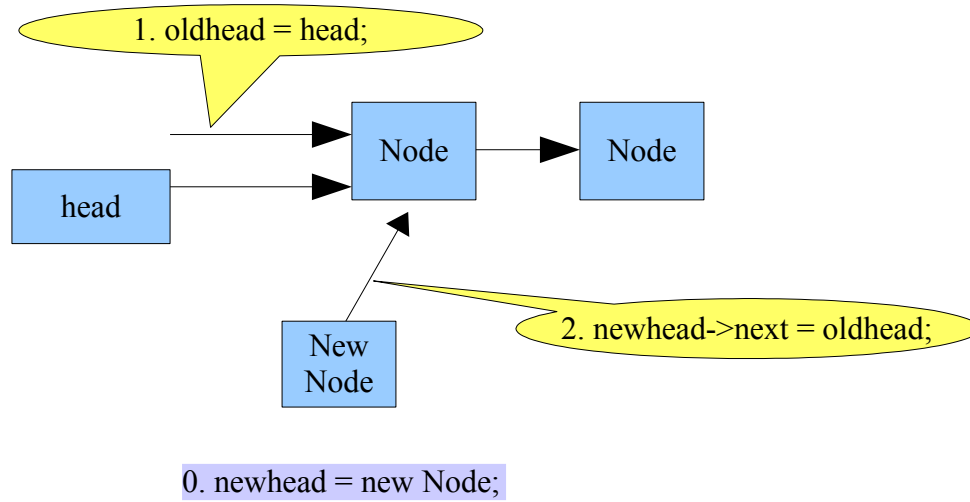
```
void push(Val val)
{
    Node * newhead = new Node(val); // 0
    do
    {
        Node * oldhead = stack.head; // 1 Read
        newhead->next = oldhead;      // 2 Modify
    }                                 // 3 Write/Retry:
    while(!CAS(&stack.head, oldhead, newhead));
}
```

*Take a good look – that's as easy as it gets!*

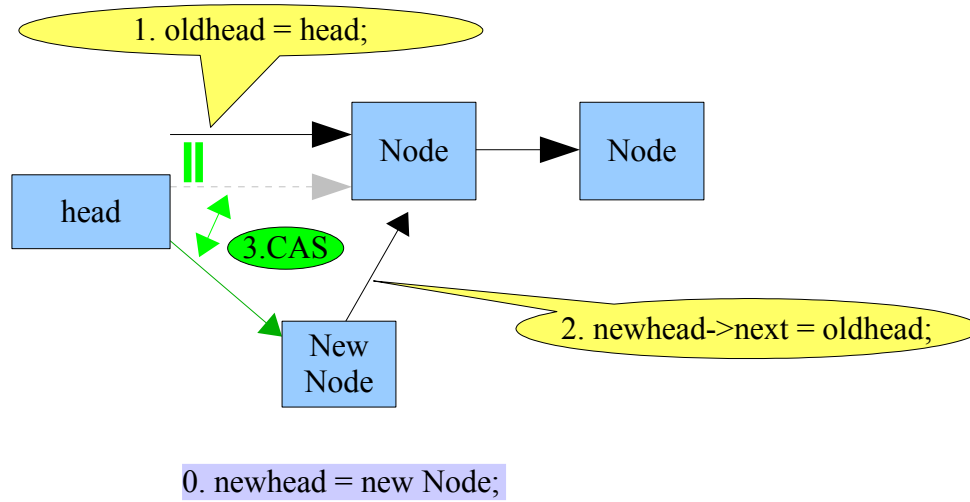
“Act Local; CAS Global”

- *me*

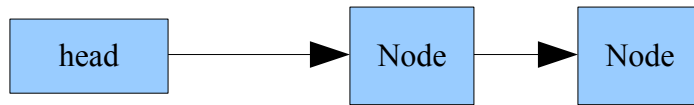
## *Act Local*



## CAS Global

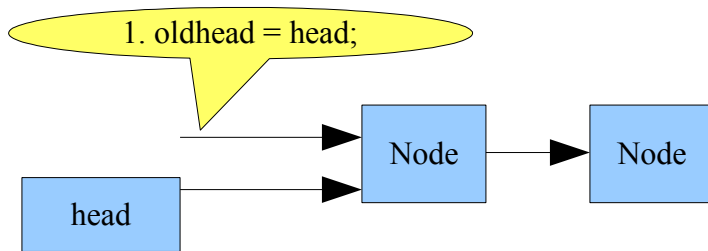


*“Hard” - Pop a Value from a Lock-free Stack*



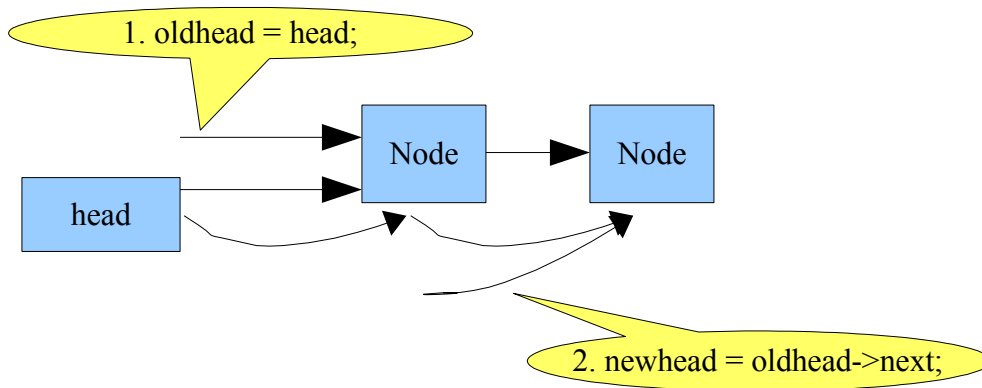
```
Val pop()  
{  
}
```

*“Hard” - Pop a Value from a Lock-free Stack*



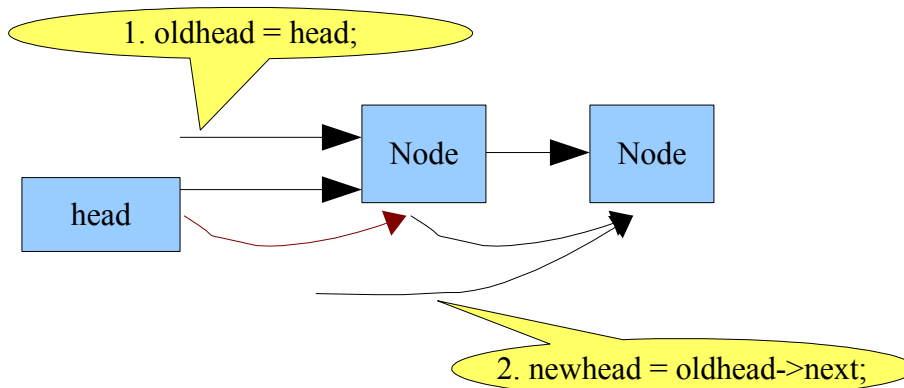
```
Val pop()  
{  
    Node * oldhead = stack.head;    // 1 Read  
}
```

*“Hard” - Pop a Value from a Lock-free Stack*



```
Val pop()  
{  
    Node * oldhead = stack.head;    // 1 Read  
    Node * newhead = oldhead->next; // 2 Modify  
}
```

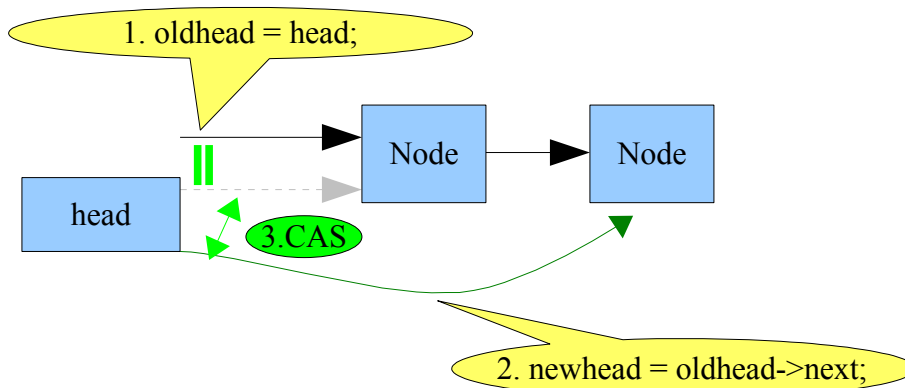
## *“Hard” - Pop a Value from a Lock-free Stack*



```
Val pop()
{
    Node * oldhead = stack.head;    // 1 Read
    if (!oldhead)
        throw StackEmpty(); // or return null, etc
    Node * newhead = oldhead->next; // 2 Modify
}
```



## *“Hard” - Pop a Value from a Lock-free Stack*

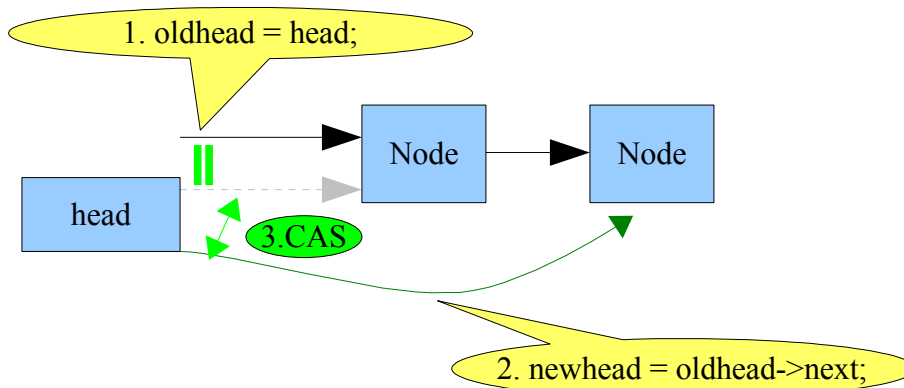


```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;    // 1 Read
        if (!oldhead)
            throw StackEmpty(); // or return null, etc
        newhead = oldhead->next;        // 2 Modify
    }
    while (!CAS(&head, oldhead, newhead)); // 3 W/R

    Val val = oldhead->val;
    delete oldhead;
    return val;
}
```

*What's wrong with this code?*

## *“Hard” - Pop a Value from a Lock-free Stack*



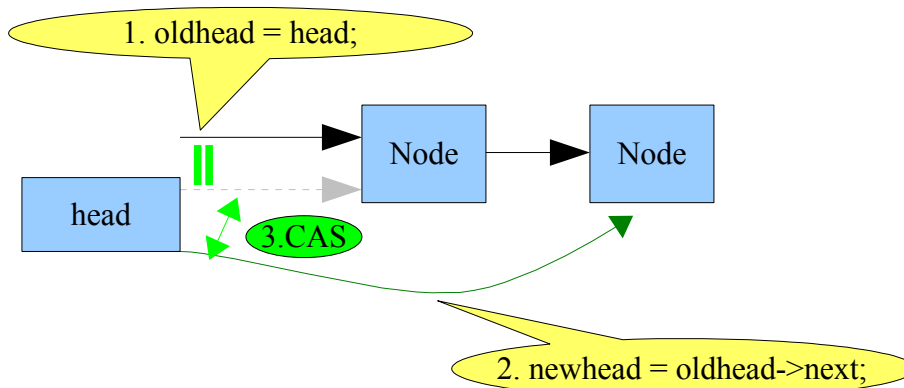
```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;    // 1 Read
        if (!oldhead)
            throw StackEmpty(); // or return null, etc
        newhead = oldhead->next;        // 2 Modify
    }
    while (!CAS(&head, oldhead, newhead)); // 3 W/R

    Val val = oldhead->val;
    delete oldhead;
    return val;
}
```

*What's wrong with this code?*

- “Are We There Yet?”

## *“Hard” - Pop a Value from a Lock-free Stack*



```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;    // 1 Read
        if (!oldhead)
            throw StackEmpty(); // or return null, etc
        newhead = oldhead->next;        // 2 Modify
    }
    while (!CAS(&head, oldhead, newhead)); // 3 W/R

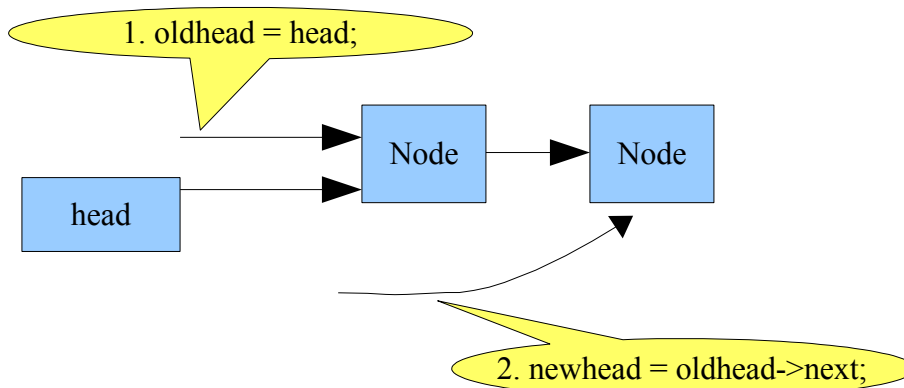
    Val val = oldhead->val;
    delete oldhead;
    return val;
}
```

*What's wrong with this code?*

- “Are We There Yet?”
- **ABA**

*Our friend CAS has issues...*

*ABA – Pop at Step 2. Prepped for CAS:*

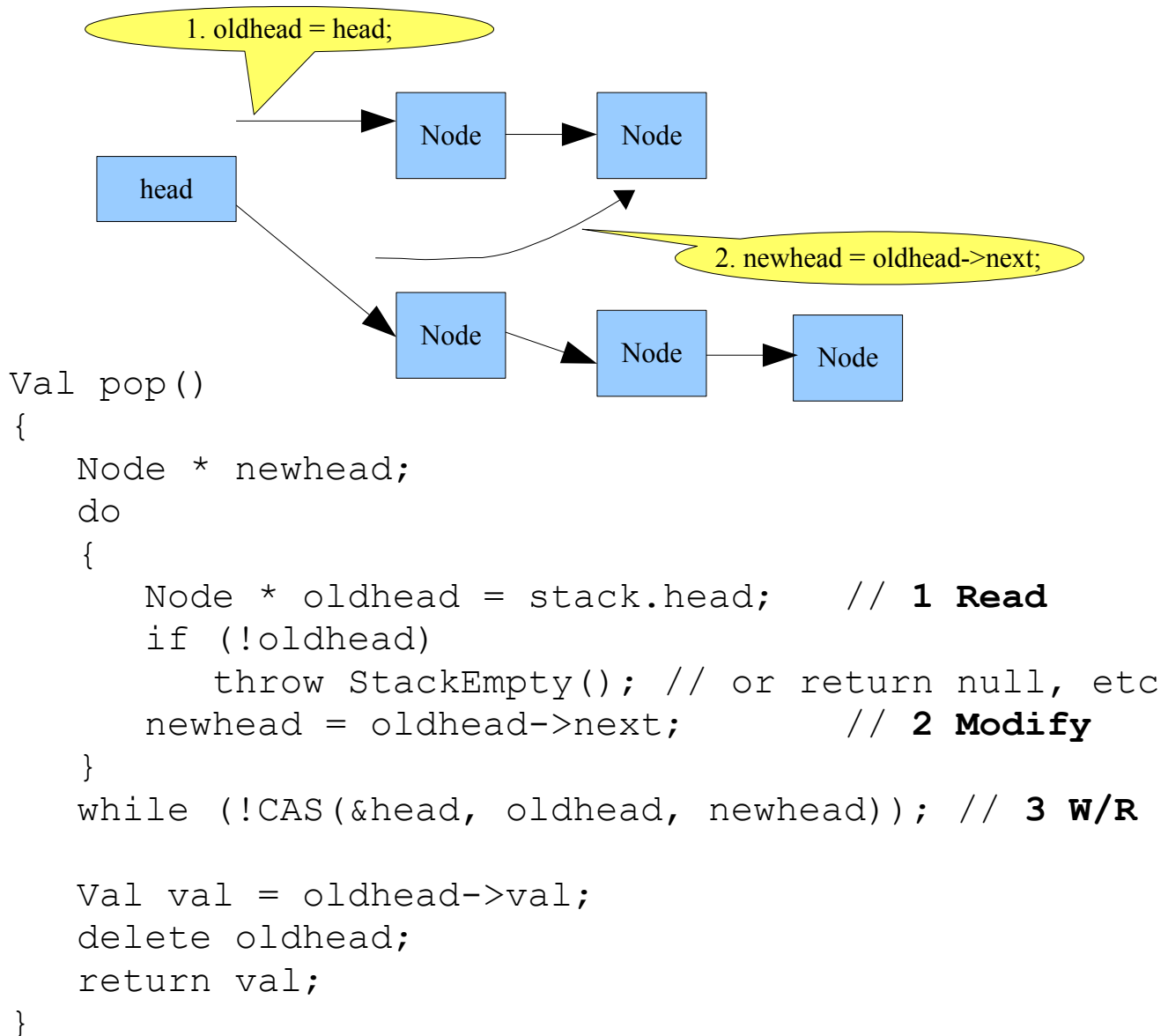


```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;    // 1 Read
        if (!oldhead)
            throw StackEmpty(); // or return null, etc
        newhead = oldhead->next;        // 2 Modify
    }
    while (!CAS(&head, oldhead, newhead)); // 3 W/R

    Val val = oldhead->val;
    delete oldhead;
    return val;
}
```

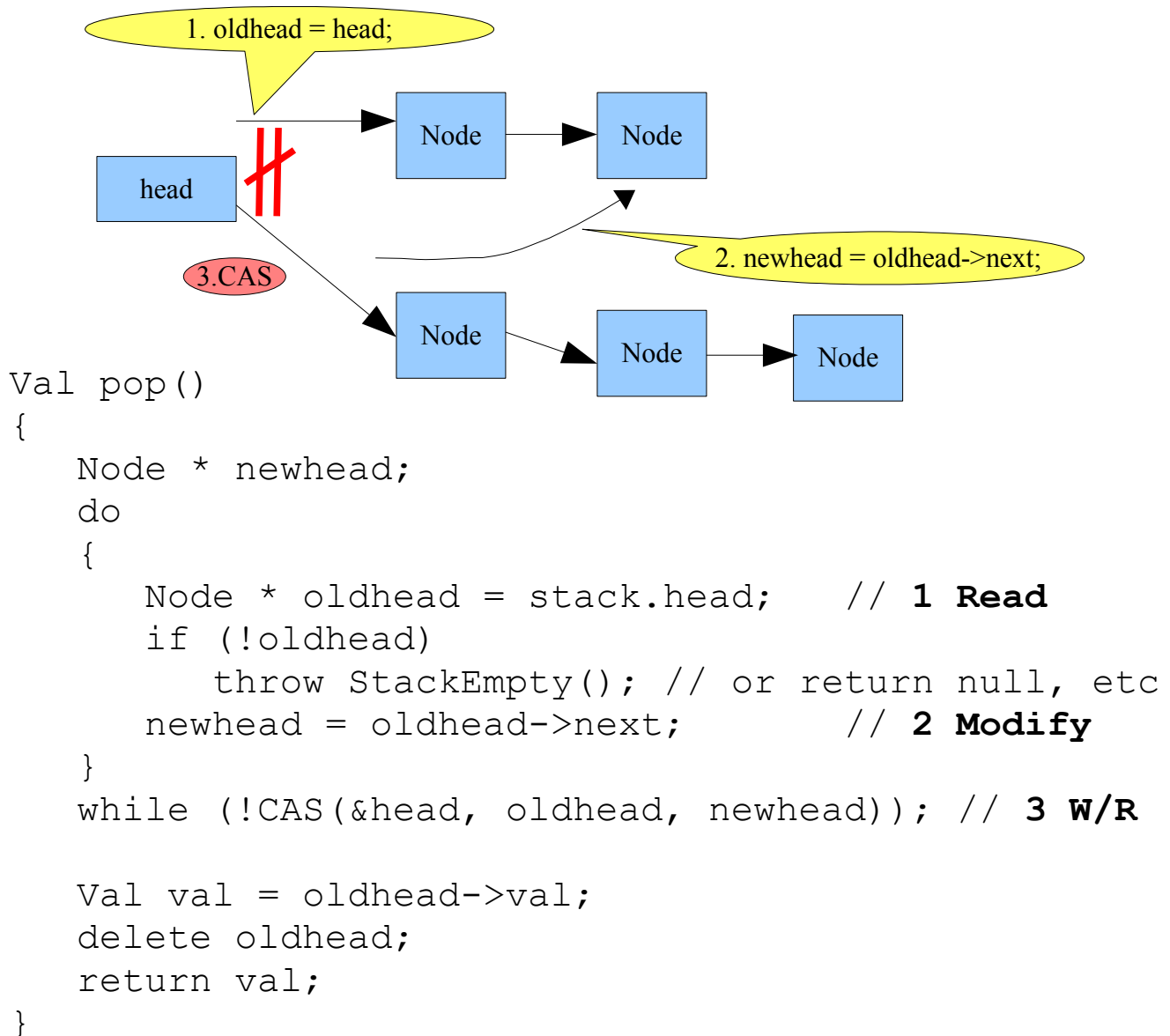
*Our friend CAS has issues...*

*ABA – Now, another thread interrupts, leaves us with:*



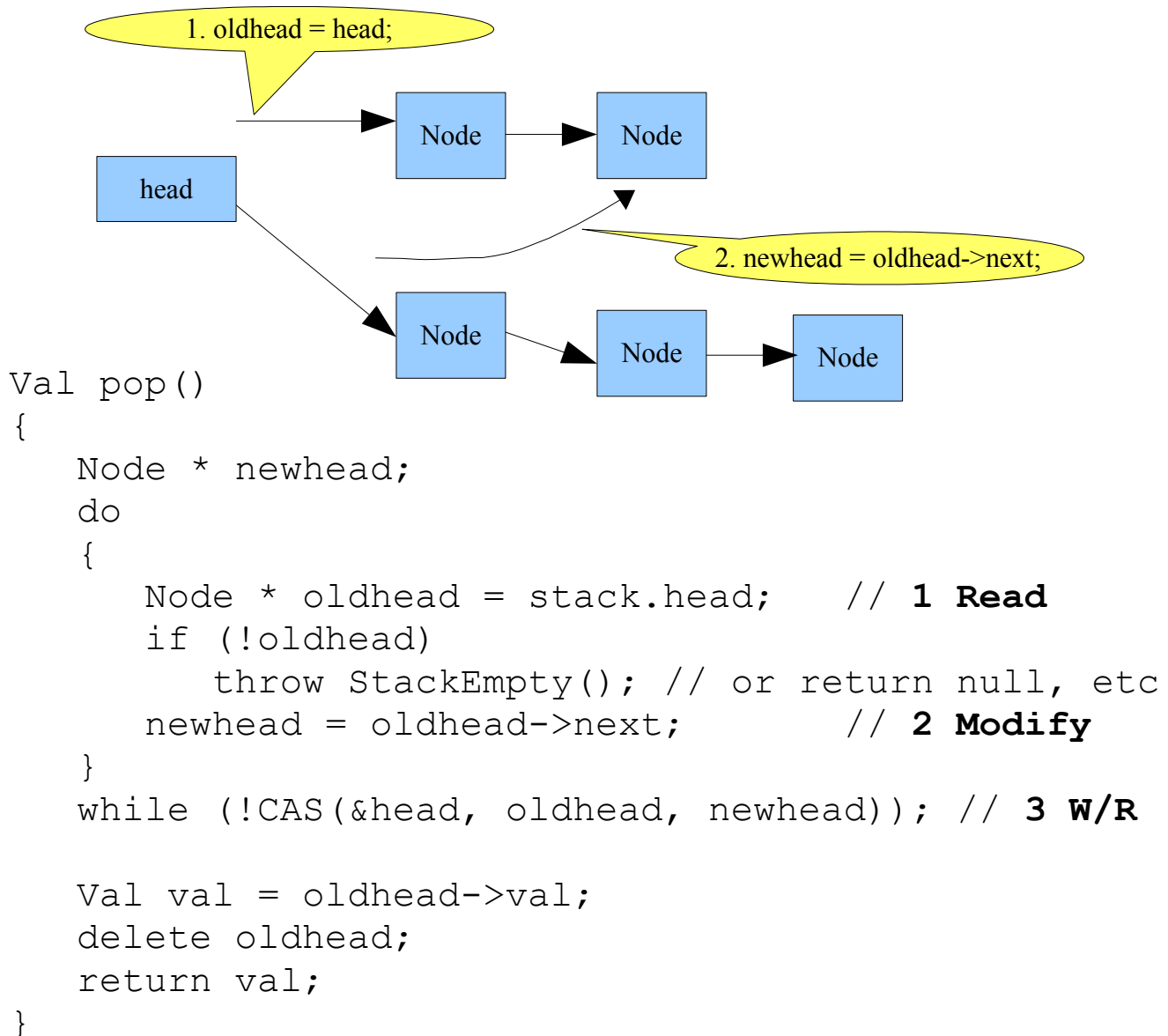
*Our friend CAS has issues...*

*ABA – CAS will fail. Yeah!*



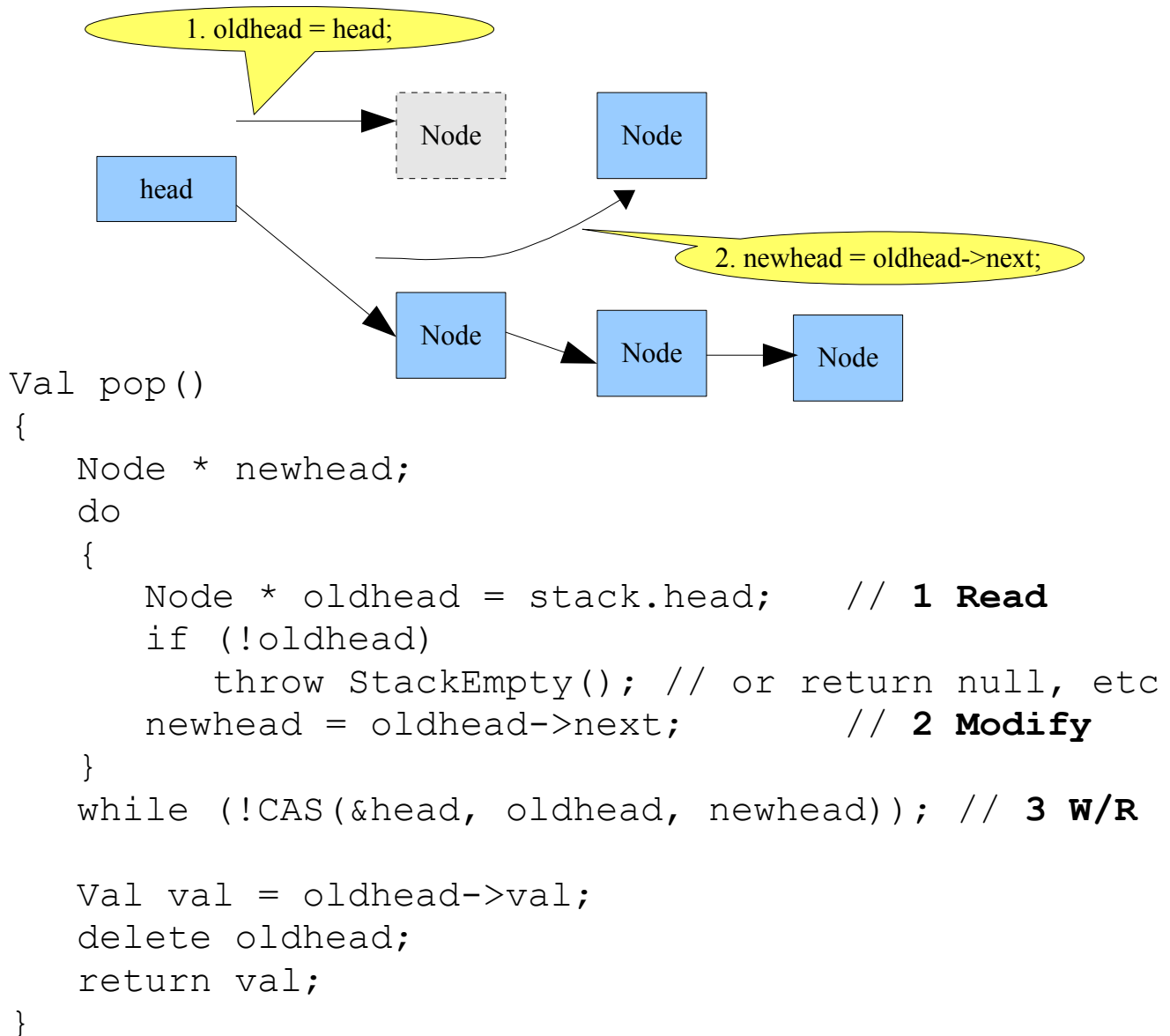
*Our friend CAS has issues...*

*ABA – Now, another thread interrupts, leaves us with:*



*Our friend CAS has issues...*

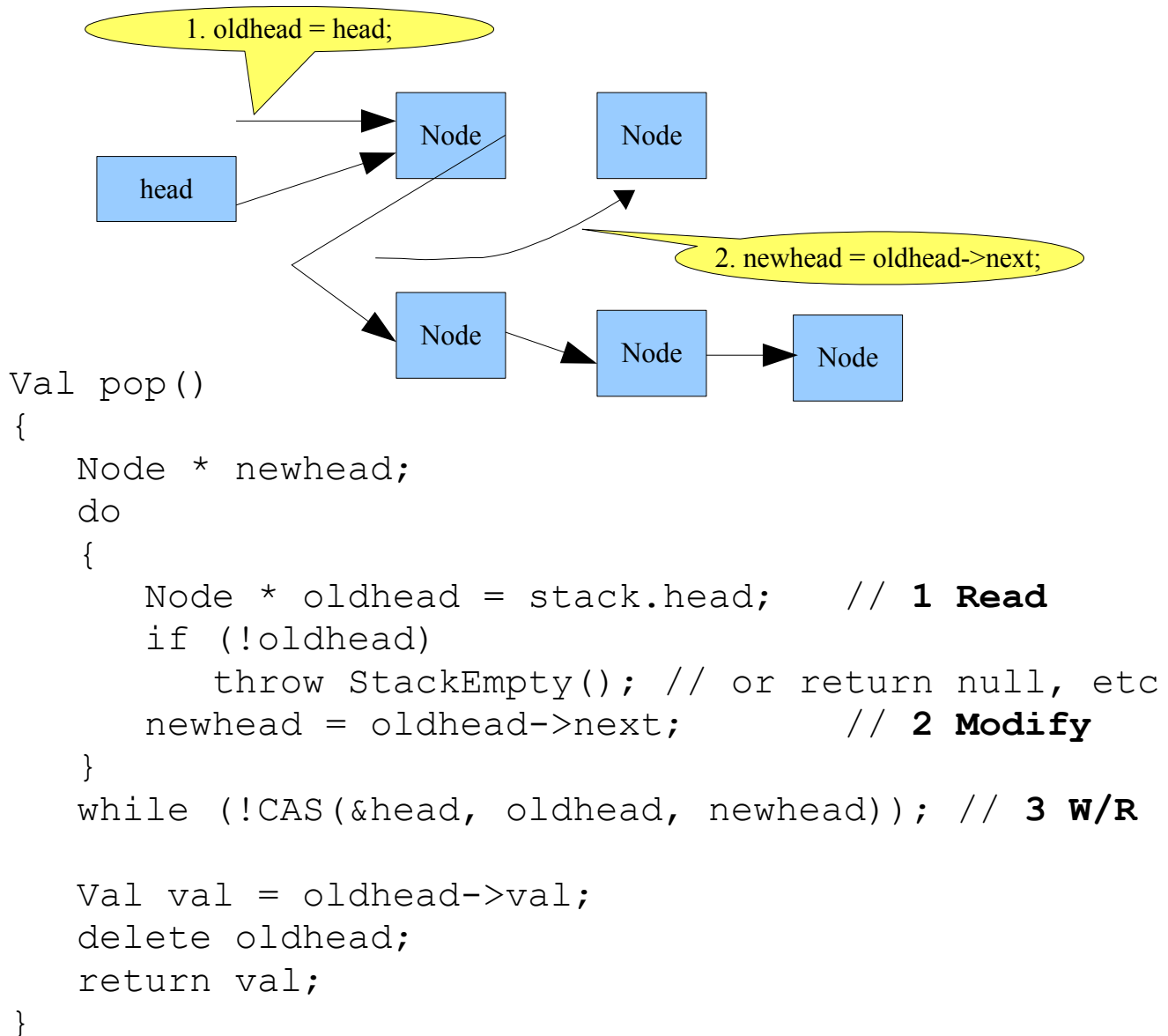
*ABA – Maybe the first Node was deleted...*



*Our friend CAS has issues...*

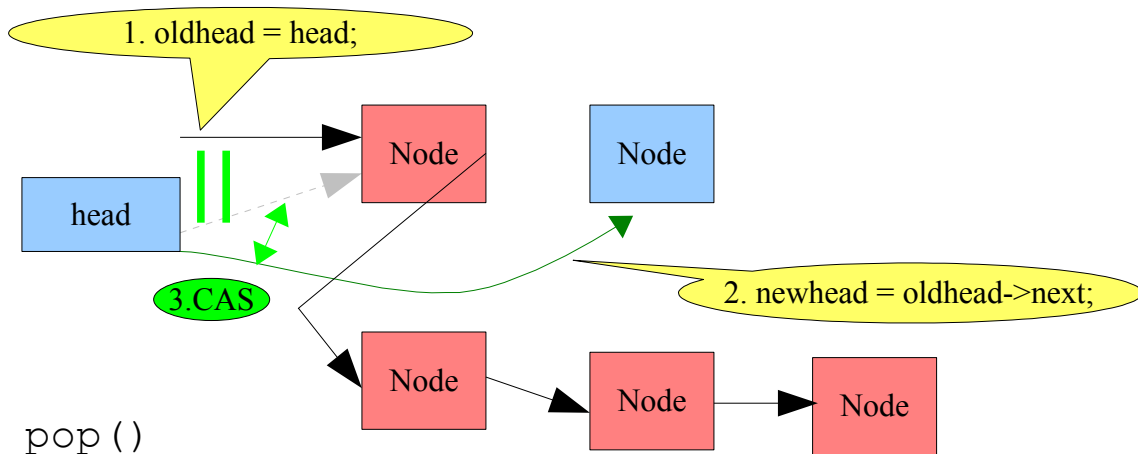


*ABA – ...And then reused (recycled by Allocator) and Pushed:*



*Our friend CAS has issues...*

*ABA – CAS Succeeds! (Yeah!?):*



```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;    // 1 Read
        if (!oldhead)
            throw StackEmpty(); // or return null, etc
        newhead = oldhead->next;        // 2 Modify
    }
    while (!CAS(&head, oldhead, newhead)); // 3 W/R

    Val val = oldhead->val;
    delete oldhead;
    return val;
}
```

*Our friend CAS has issues...*

*DWCAS to the Rescue!*

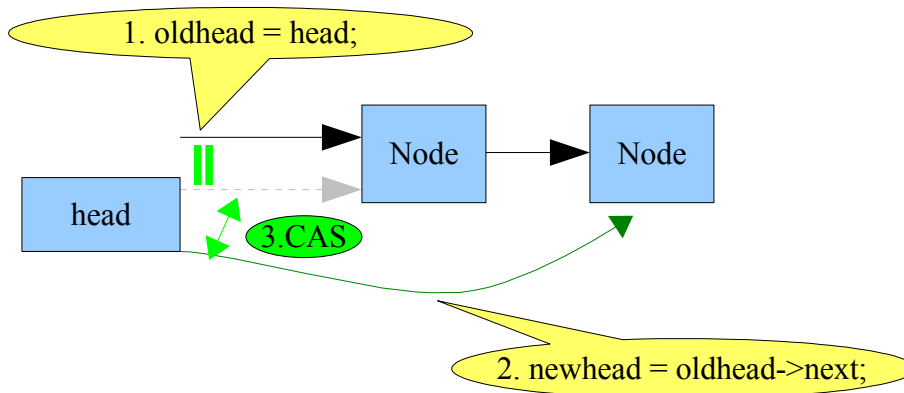
```
DWCAS ( [____word____|____word____], oldwide, newwide);
```

*What should we do with that?*

*DWCAS to the Rescue!*

[\_\_counter\_\_|\_\_pointer\_\_]

## DWCAS – Pop from a lockfree stack



```
Val pop()
{
    NodePtr newhead;
    do
    {
        NodePtr oldhead = stack.head; // 1 Read
        if (!oldhead)
            throw StackEmpty(); // or return null, etc
        newhead = oldhead->next; // 2 Modify
    }
    while (!DWCAS(&head, oldhead, newhead)); // 3

    Val val = oldhead->val;
    delete oldhead;
    return val;
}

NodePtr NodeAlloc(Val val)
{
    NodePtr ptr;
    ptr.ptr = new Node(val);
    ptr.count = atomic_inc(NodePtr::version);
    return NodePtr;
}
```

**Conclusions...**

“Use Locks!”

“Use Locks!”

FCD – Fear, Certainty, Doubt

- scary information
- correct information



“Use Locks!”

FCD – Fear, Certainty, Doubt

- scary information
- correct information

*I Am Not An Expert.*

## *Links*

[comp.programing.threads](#)

[gottlobfrege@gmail.com](#)

[lockfree@forecode.com](#)

[www.forecode.com](#)