~~An Introduction to~~ Lock-free Programming
Continued

Tony Van Eerd
BoostCon 2011

1.
Summary of
Lockfree BoostCon 2010

2 hours of content that was almost presented in 90min, condensed to 30min

– or less –

“Use Locks!”

A Guide to Threaded Coding

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks
3. Measure

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks
3. Measure
4. Measure

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm
6. GOTO 1

A Guide to Threaded Coding

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm
6. GOTO 1

∞. Lock-free

A Guide to Threaded Coding

"Lock-free coding is the <u>last</u> thing you want to do."

1. Forget what you learned in Kindergarten

   (stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm
6. GOTO 1

∞. Lock-free

Example 1

Bad

Thread 1,2,3...:

```
static bool locked;

if ( !locked )
{
   locked = true;
   do_exclusive_stuff();
}
else...
```

Example 1

Bad

Thread 1,2,3...:

```
static bool locked;

if ( !locked )
{
   locked = true;
   do_exclusive_stuff();
}
else...
```

"Read between the lines"

Example 1 - Atomically

Bad

Thread 1,2,3...:

```
static bool locked;

if ( !locked )
{
   locked = true;
   do_exclusive_stuff();
}
else...
```

Atomically

How?

Example 1 - Atomically

Good - "CAS"

Thread 1,2,3...:

```
static std::atomic<bool> locked;

if (locked.compare_exchange(false, true))
{
   do_exclusive_stuff();
}
else...
```

(CAS = Compare-And-Set == Compare-And-Swap == Test-And-Set == compare_exchange)

Example 1 - Atomically

Bad

```
static bool locked;

if ( !locked )
{
   locked = true;
   do_exclusive_stuff();
}
else...
```

*Good - **CAS***

```
static std::atomic<bool> locked;

if (locked.compare_exchange(false, true))
{
   do_exclusive_stuff();
}
else...
```

Example 1 - Atomically

Bad

```
static bool locked;

if ( !locked )
{
   locked = true;
   do_exclusive_stuff();
}
else...
```

*Good - **CAS***

```
static std::atomic<bool> locked;

if (locked.compare_exchange(false, true))
{
   do_exclusive_stuff();
}
else...
```

Simple, Right?...

Example 2

Bad

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Thread 1:

Your Code:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Example 2

Thread 1:

Your Code:

Evil (or Smart?) CPU(s):

```
data.x = ....;
data.y = ....;
data_ready = true;
```


```
data_ready = true;
data.x = ....;
data.y = ....;
```

That was Then:

speed of CPU == speed of RAM

That was Then:

speed of CPU == speed of RAM

This is Now:

speed of CPU >>> speed of RAM

100x!

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.

Example 2

Thread 1:

Your Code:

Evil/Smart/Fast CPU(s):

```
data.x = ....;
data.y = ....;
data_ready = true;
```


```
data_ready = true;
data.x = ....;
data.y = ....;
```

Example 2

Thread 1:

Your Code:	*Evil/Smart/Fast CPU(s):*

`data.x = ....;`
`data.y = ....;`
`data_ready = true;`
	Shift Happens
	`data_ready = true;`
	`data.x = ....;`
	`data.y = ....;`

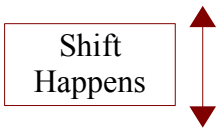
Example 2

Thread 1:

Your Code:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Evil/Smart/Fast CPU(s):




```
data_ready = true;
data.x = ....;
data.y = ....;
```

Thread 2:

Your Code:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Evil/Smart/Fast CPU(s):

```
Can't reorder an if!
        (?)
```

Example 2

Thread 1:

Your Code:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Evil/Smart/Fast CPU(s):

Shift Happens

```
data_ready = true;
data.x = ....;
data.y = ....;
```

Thread 2:

Your Code:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Evil/Smart/Fast CPU(s):

Shift Happens

```
tmp = read(data);
if (data_ready)
{
   a = tmp.x;
   b = tmp.y;
}
```

Example 2

Bad

```
static bool data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready = true;
q = 10;
```

Thread 2:

```
w = r;
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

release: *before means before*

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

release: *before means before*

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::before_means_before);
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

```
tmp = read(data);
if (data_ready)
{
   a = tmp.x;
   b = tmp.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready.load(std::memory_order_acquire))
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready.load(std::memory_order_acquire))
{
   a = data.x;
   b = data.y;
   ...
}
```

acquire: *after means after*

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

release: *before means before*

Thread 2:

```
if (data_ready.load(std::memory_order_acquire))
{
   a = data.x;
   b = data.y;
   ...
}
```

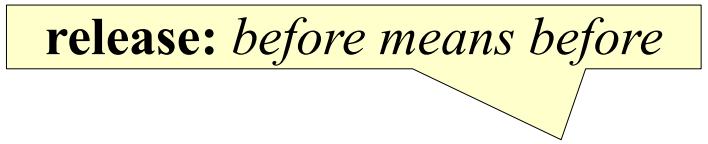
acquire: *after means after*

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

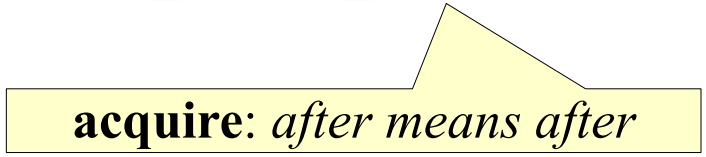
Thread 1:




```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready.load(std::memory_order_acquire))
{
   a = data.x;
   b = data.y;
   ...
}
```



Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

CPU 1

release: *before means before*

A

```
data.x = ....;
data.y = ....;
data_ready.store(true, std::memory_order_release);
```

Thread 2:

```
if (data_ready.load(std::memory_order_acquire))
{
```

B

```
   a = data.x;
   b = data.y;
   ...
}
```

acquire: *after means after*

CPU 2

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

A

```
data.x = ....;
data.y = ....;
data_ready = true;
```

Thread 2:

B

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2

Good - C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

A

```
data.x = ....;
data.y = ....;
data_ready = true;
```

```
data.x = ....;
data.y = ....;
data_ready.store(true,release);
```

Thread 2:

B

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

```
if(data_ready.load(acquire))
{
   a = data.x;
   b = data.y;
   ...
}
```

Example 2 – Happens Before

Good – C++0x

```
static std::atomic<bool> data_ready;
```

Thread 1:

A

```
data.x = ....;
data.y = ....;
data_ready = true;
```

```
data.x = ....;
data.y = ....;
data_ready.store(true,release);
```

Thread 2:

B

```
if (data_ready)
{
   a = data.x;
   b = data.y;
   ...
}
```

```
if(data_ready.load(acquire))
{
   a = data.x;
   b = data.y;
   ...
}
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

```
data.x = ....; // A
data.y = ....; // B                    // A
data_ready.store(true,release); // B
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

```
data.x = ....; // A                                    // A
data.y = ....; // B                  // A              // C
data_ready.store(true,release); // B                   // B
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

```
data.x = ....;
data.y = ....;
data_ready.store(true,release);
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

```
data.x = ....;
data.y = ....;
data_ready.store(true,release);
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a store to some atomic variable m, with an ordering of std::memory_order_release, or std::memory_order_seq_cst,

and

B is a load from the same variable m, with an ordering of std::memory_order_acquire or std::memory_order_seq_cst,

and

B reads the value stored by A.

```
data.x = ....;
data.y = ....;
data_ready.store(true,std::memory_order_release); //A
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a load from the same variable m, with an ordering of std::memory_order_acquire or std::memory_order_seq_cst,

and

B reads the value stored by A.

```
data.x = ....;
data.y = ....;
data_ready.store(1,release);
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a load from the same variable m, with an ordering of std::memory_order_acquire or std::memory_order_seq_cst,

and

B reads the value stored by A.

`data.x = ....;`
`data.y = ....;`
`data_ready.store(1,release);`
	`if(data_ready.load(acquire))`
	`{`
	`  a = data.x;`
	`  b = data.y;`
	`}`

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a **load** from the same variable m, with an ordering of std::**memory_order_acquire** or std::memory_order_seq_cst,

and

B reads the value stored by A.

```
data.x = ....;
data.y = ....;
data_ready.store(1,release);
```

```
if(data_ready.load(acquire))
{
    a = data.x;
    b = data.y;
}
```

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a **load from the same variable** m, with an ordering of std::**memory_order_acquire** or std::memory_order_seq_cst,

and

B reads the value stored by A.

`data.x = ....;`
`data.y = ....;`
`data_ready.store(1,release);`
	`if(data_ready.load(acquire))`
	`{`
	`  a = data.x;`
	`  b = data.y;`
	`}`

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a **load from the same variable** m, with an ordering of std::**memory_order_acquire** or std::memory_order_seq_cst,

and

B reads the value stored by A.

`data.x = ....;`
`data.y = ....;`
`data_ready.store(1,release);`
	`if(data_ready.load(acquire))`
	`{`
	`  a = data.x;`
	`  b = data.y;`
	`}`

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a **load from the same variable** m, with an ordering of std::**memory_order_acquire** or std::memory_order_seq_cst,

and

B reads the value stored by A.

`data.x = ....;`
`data.y = ....;`
`data_ready.store(1,release);`
	`if(data_ready.load(acquire))`
	`{`
	`  a = data.x;`
	`  b = data.y;`
	`}`

Happens-before

An operation A *happens-before* an operation B if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a **store** to some **atomic** variable m, with an ordering of std::**memory_order_release**, or std::memory_order_seq_cst,

and

B is a **load from the same variable** m, with an ordering of std::**memory_order_acquire** or std::memory_order_seq_cst,

and

B reads the value stored by A.

A

```
data.x = ....;
data.y = ....;
data_ready.store(1,release);
```

```
if(data_ready.load(acquire))
{
```

B

```
    a = data.x;
    b = data.y;
```

```
}
```

Happens-before

An operation **A *happens-before* an operation B** if:

A is performed on the same thread as B, and A is before B in program order,

or

A synchronizes-with B,

or

A happens-before some other operation C, and C happens-before B.

Synchronizes-with

An operation A *synchronizes-with* an operation B if

A is a store to some atomic variable m, with an ordering of std::memory_order_release, or std::memory_order_seq_cst,

and

B is a load from the same variable m, with an ordering of std::memory_order_acquire or std::memory_order_seq_cst,

and

B reads the value stored by A.

Summary

Atomic *(no reading/acting between the lines)*

```
if ( !locked )
{
   locked = true;
   do_exclusive();
}
```

```
if(locked.compare_exchange(false,true))
{
   do_exclusive();
}
```

Happens-before *(ensure order when needed)*

```
data = ...;
ready = true;

.........

if(ready)
{
   use(data);
}
```

```
data = ...;
ready.store(true,memory_order_release);

..........

if(ready.load(memory_order_acquire))
{
   use(data);
}
```

Atomic *(no reading/acting between the lines)*

```
if ( !locked )
{
   locked = true;
   do_exclusive();
}
```

```
if(locked.compare_exchange(false,true))
{
   do_exclusive();
}
```

Atomic *(no reading/acting between the lines)*

```
if(locked.compare_exchange(false,true))
{
   do_exclusive();
}
```

Atomic *(no reading/acting between the lines)*

```
if(locked.compare_exchange(false,true))
{
   do_exclusive(data);
}
```

Atomic + Happens-before

```
if(locked.compare_exchange(false,true,memory_order_acquire))
{
    do_exclusive(data);
}
```

Atomic + Happens-before

```
if(locked.compare_exchange(false,true,memory_order_acquire))
{
   do_exclusive(data);
   locked.store(false, memory_order_release);
}
```

Atomic + Happens-before

```
std::atomic<bool> locked;

if(locked.compare_exchange(false,true,memory_order_acquire))
{
   do_exclusive(data);
   locked.store(false, memory_order_release);
}
```

Atomic + Happens-before

```
std::atomic<bool> locked;

if(locked.compare_exchange(false,true,memory_order_acquire))
{
    do_exclusive(data);
    locked.store(false, memory_order_release);
}
```

No reordering

No "Reading between the lines"

Reading between the Tokens

Reading between the Tokens

```
count++;
```

Reading between the Tokens

```
    count++;
```

=>

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

Reading between the Tokens

```
    count++;

=>

    register reg = read_memory(&count);
    reg = reg + 1;
    write_memory(&count, reg);
```

Reading between the Tokens

```
    count++;
```

=>

```
    register reg = read_memory(&count);
    reg = reg + 1;
    write_memory(&count, reg);
```

"Read between the lines"

Reading between the Tokens

```
count++;
```

"Read between the... tokens"

=>

```
register reg = read_memory(&count);
reg = reg + 1;
write_memory(&count, reg);
```

"Read between the lines"

Reading between the Tokens

```
count++;
```

"Read between the... tokens"

=>

```
register reg = read_memory(&count);
reg = reg + 1;
write_memory(&count, reg);
```

"Read between the lines"

Solution?...

Atomic Increment

```
std::atomic<int> count;

count++;
```

Atomic Increment

```
std::atomic<int> count;

count++;
```

=>

????

Atomic Increment

```
std::atomic<int> count;

count++;
```

=>

```
// lock or TM magic?
atomically {
    register reg = read_memory(&count);
    reg = reg + 1;
    write_memory(&count, reg);
}
```

Atomic Increment

```
std::atomic<int> count;

count++;
```

=>

```
// processor magic?
__asm {
   LOCK inc @count
}
```

Atomic Increment

```
std::atomic<int> count;

count++;
```

=>

```
//?
register reg = read_memory(&count);
reg = reg + 1;
write_memory(&count, reg);
```

Thread2: **count++;**

Atomic Increment

```
    std::atomic<int> count;

    count++;

=>
    //?
    register old = read_memory(&count);
    register new = old + 1;
    count.compare_exchange(old, new);  //?
```

Atomic Increment

```
    std::atomic<int> count;

    count++;

=>
retry:
    register old = read_memory(&count);
    register new = old + 1;
    if ( ! count.compare_exchange(old, new) )
       goto retry;
```

Atomic Increment

```
std::atomic<int> count;

count++;
```

=>

```
do {
   register old = read_memory(&count);
   register new = old + 1;
} while ( ! count.compare_exchange(old, new) );
```

Atomic Increment

```
std::atomic<int> count;

count++;
```

=>

```
do {
   int old = count.load(std::memory_order_relaxed);
   int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

CAS Loop

```
do {
   int old = count.load(std::memory_order_relaxed);
   int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

CAS Loop

Read global

```
do {
   int old = count.load(std::memory_order_relaxed);
   int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

CAS Loop

Act local

```
do {
int old = count.load(std::memory_order_relaxed);
int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

CAS Loop

```
do {
   int old = count.load(std::memory_order_relaxed);
   int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

CAS global

CAS Loop

```
do {
   int old = count.load(std::memory_order_relaxed);
   int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

"Think Global
Act Local"

CAS Loop

```
do {
   int old = count.load(std::memory_order_relaxed);
   int newc = old + 1;
} while ( ! count.compare_exchange(old, newc));
```

"Act Local
CAS Global"

“CAS”

C++0x

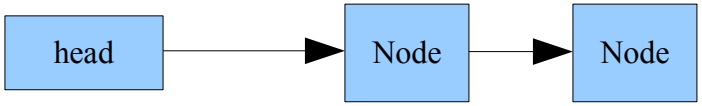
- std::memory_order_seq_cst
- std::memory_order_acq_rel
- std::memory_order_acquire
- std::memory_order_release
- std::memory_order_consume
- std::memory_order_relaxed

compare_exchange_strong(volatile type* x, type* expected, type desired, success_order, failure_order)
compare_exchange_weak(volatile type* x, type* expected, type desired, success_order, failure_order)

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

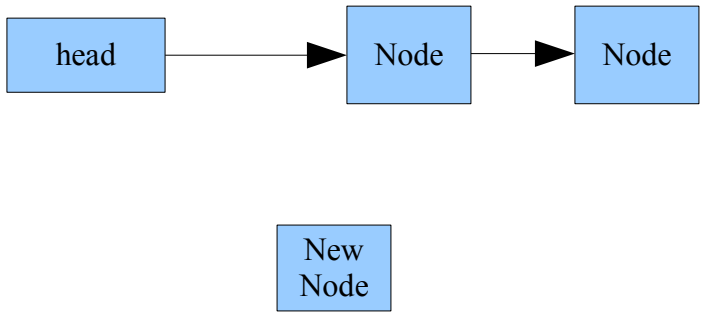
2.
Data Structures

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




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

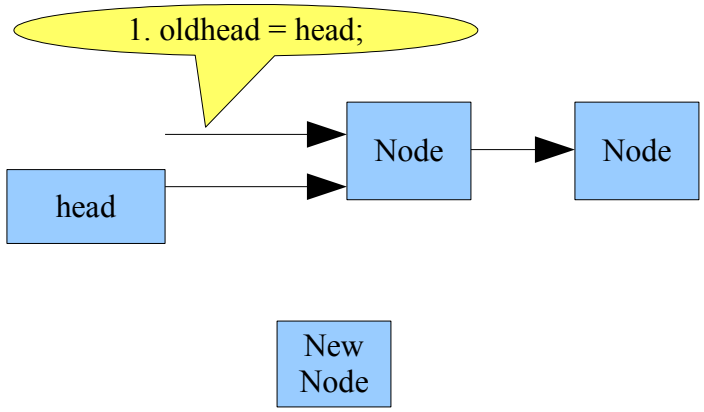
“Easy” - Push a Value onto a Lock-free Stack



0. newhead = new Node;

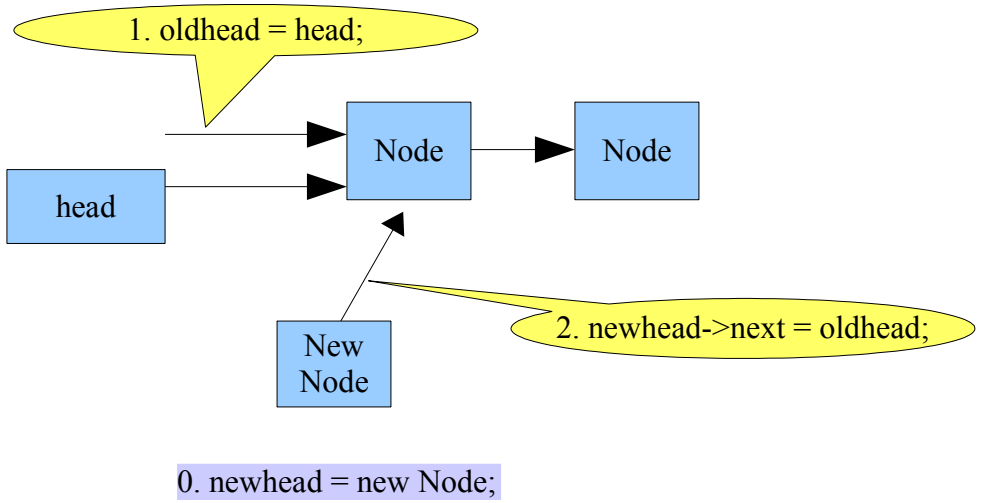
```
void push(Val val)
{
   Node * newhead = new Node(val);    // 0
}
```

"Easy" - Push a Value onto a Lock-free Stack



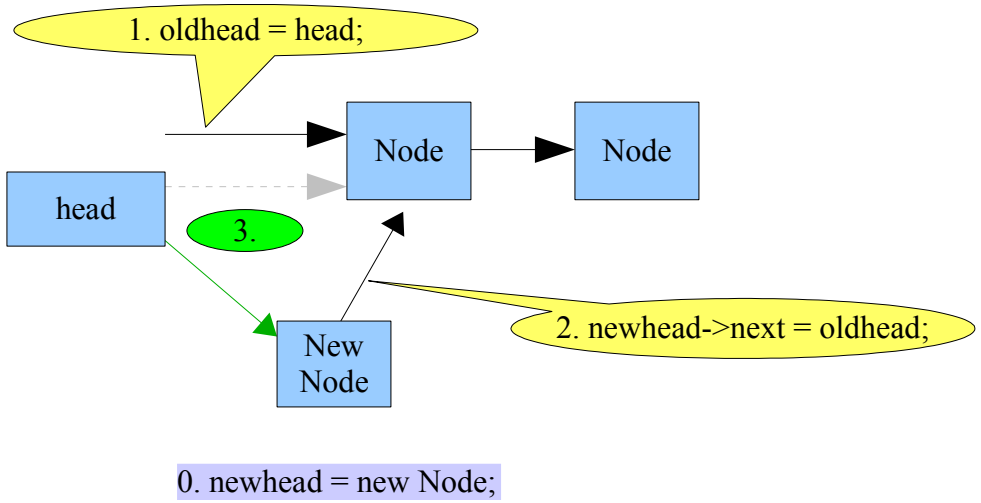

```
void push(Val val)
{
   Node * newhead = new Node(val);    // 0
   Node * oldhead = stack.head;       // 1 Read
}
```

"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 Act Locally
}
```

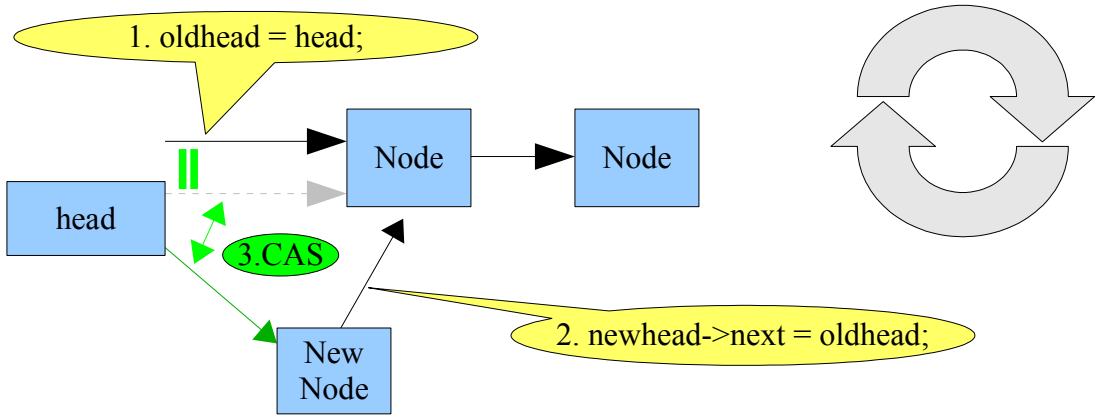
"Easy" - Push a Value onto a Lock-free Stack




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

Write, right?
Wrong.

"Easy" - Push a Value onto a Lock-free Stack



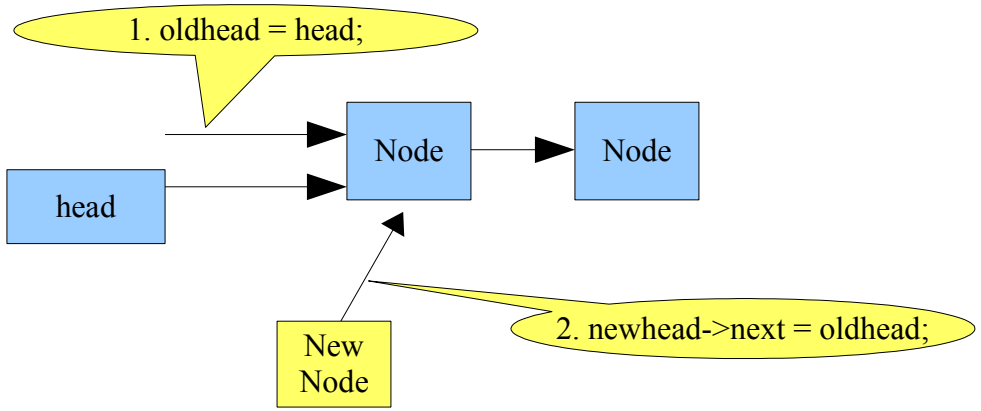
0. newhead = new Node;

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

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

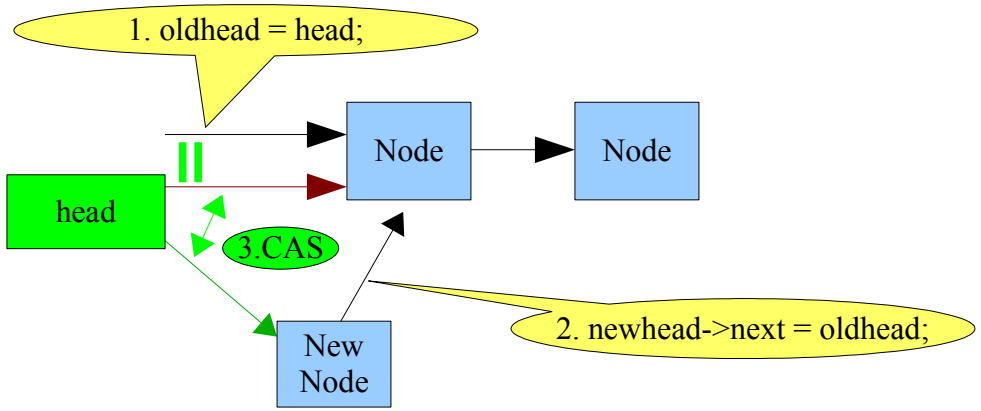
P.S. What memory ordering is required, and where? (and why?)

Act Local



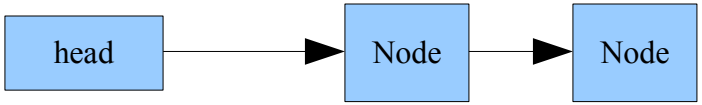
0. newhead = new Node;

CAS Global



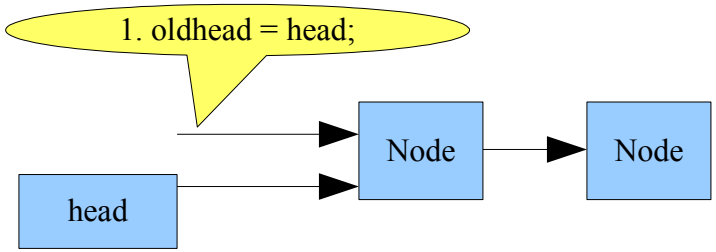
0. newhead = new Node;

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



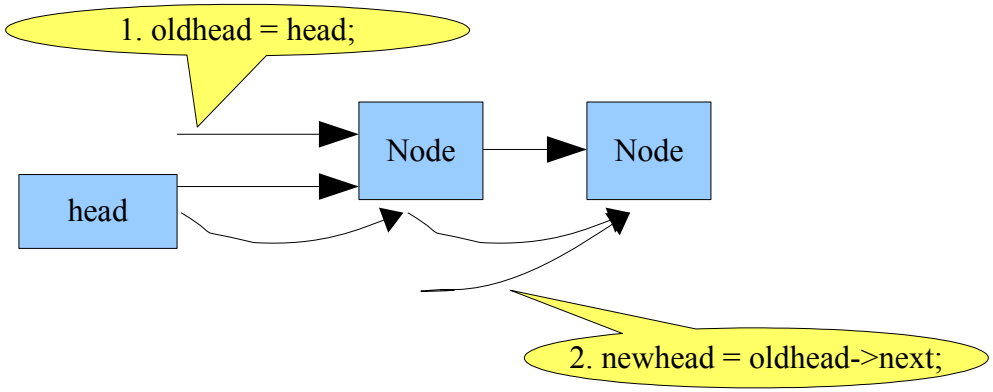

```
Val pop()
{
}
```

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



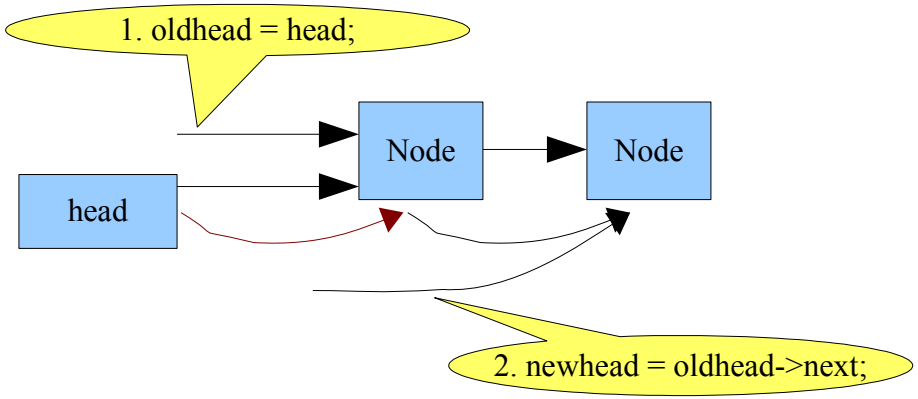

```
Val pop()
{
   Node * oldhead = stack.head;     // 1 Read
}
```

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



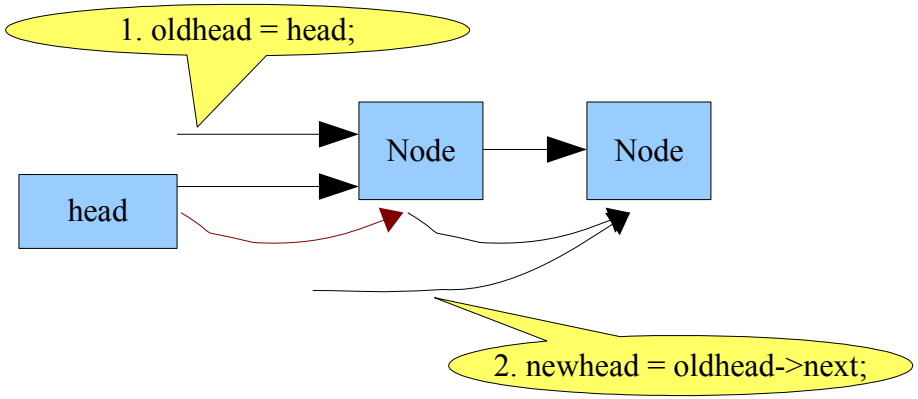

```
Val pop()
{
   Node * oldhead = stack.head;     // 1 Read
   Node * newhead = oldhead->next; // 2 Act Locally
}
```

"Hard" - Pop a Value from a Lock-free Stack



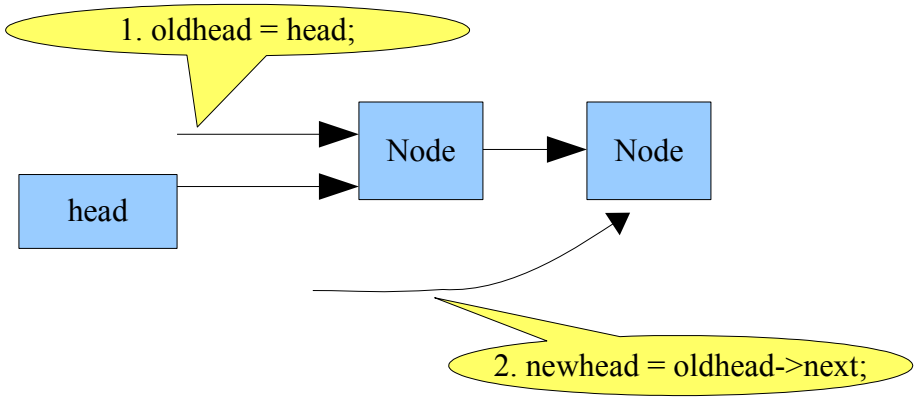

```
Val pop()
{
   Node * oldhead = stack.head;     // 1 Read
   if (!oldhead)
      throw StackEmpty(); // or return null, etc
   Node * newhead = oldhead->next; // 2 Act Locally
}
```

"Hard" - Pop a Value from a Lock-free Stack



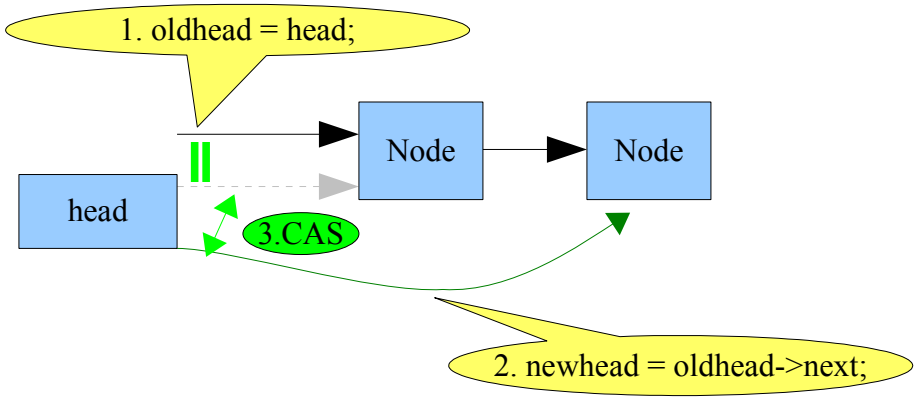

```
Val pop()
{
   Node * oldhead = stack.head;     // 1 Read
   if (!oldhead)
      throw StackEmpty(); // or return null, etc
   Node * newhead = oldhead->next; // 2 Act Locally
}
```

“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 Act Locally
}
```

"Hard" - Pop a Value from a Lock-free Stack



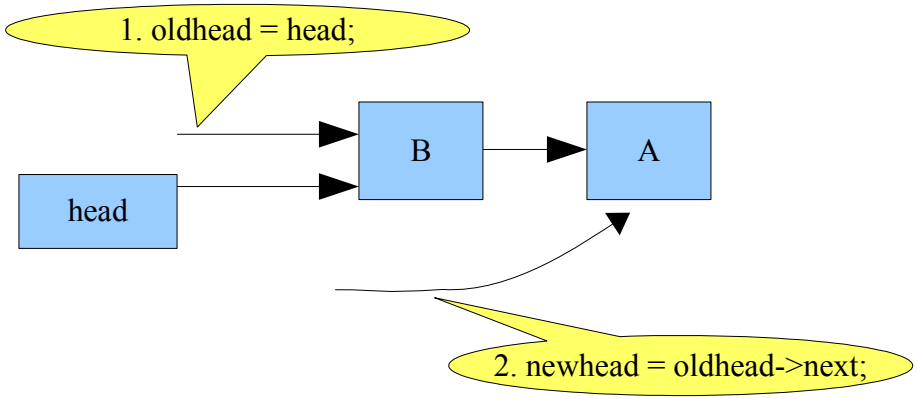

```
Val pop()
{
   Node * newhead;
   do
   {
      Node * oldhead = stack.head;    // 1 Read
      if (!oldhead)
         throw StackEmpty(); // or return null, etc
      newhead = oldhead->next;         // 2 Locally
   }
   while (!head.CAS(oldhead, newhead)); // 3 CAS Globally

   Val val = oldhead->val;
   recycle(oldhead);
   return val;
}
```

What's wrong with this code?

In a word:

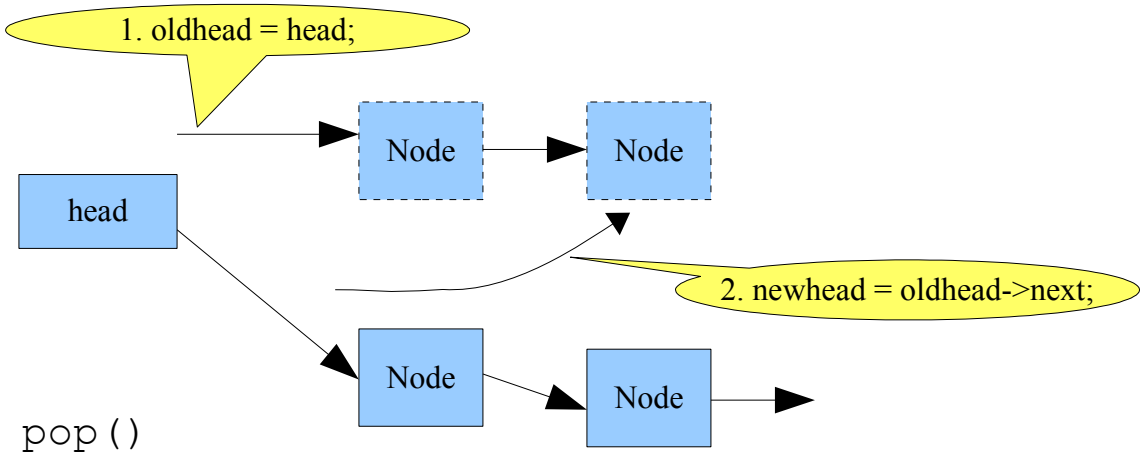
ABA

A – Pop at Step 2. Prepped for CAS:



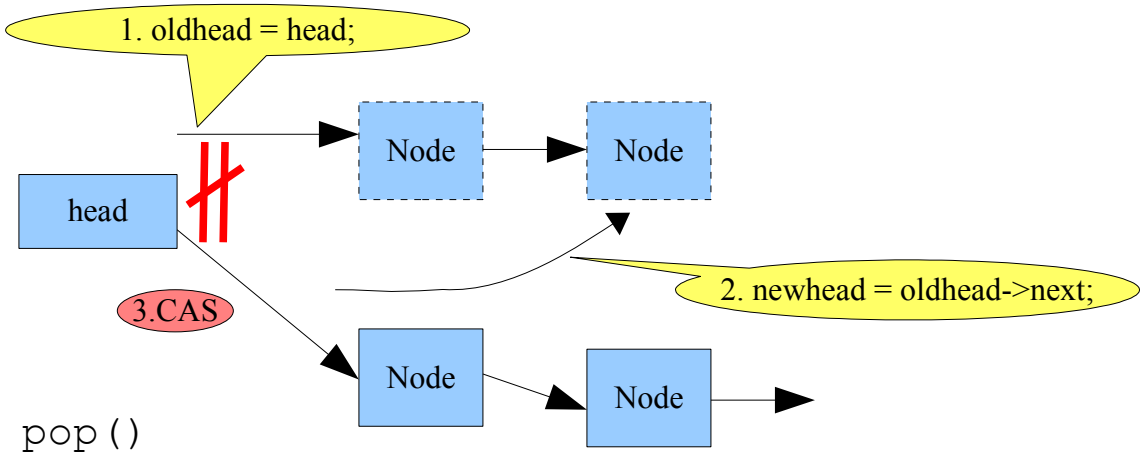

```
Val pop()
{
   Node * newhead;
   do
   {
      Node * oldhead = stack.head;
      if (!oldhead)
         throw StackEmpty();
      newhead = oldhead->next;
   }
   while (!head.CAS(oldhead, newhead));

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

AB – Now, another thread interrupts, leaves us with:



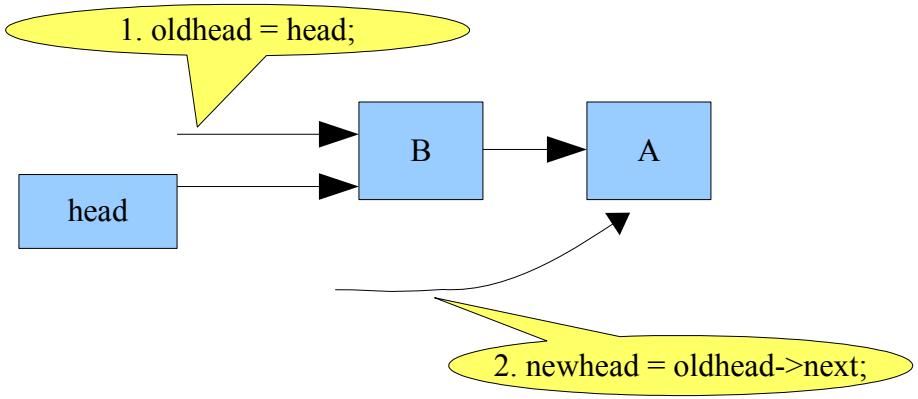

```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;
        if (!oldhead)
            throw StackEmpty();
        newhead = oldhead->next;
    }
    while (!head.CAS(oldhead, newhead));

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

AB – CAS will fail. Yeah!



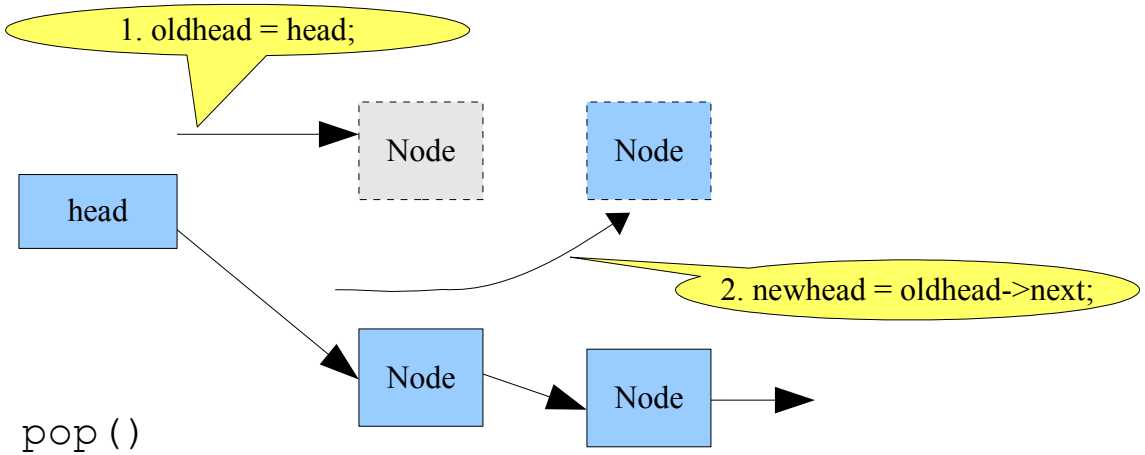

```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;
        if (!oldhead)
            throw StackEmpty();
        newhead = oldhead->next;
    }
    while (!head.CAS(oldhead, newhead));

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

A – Pop at Step 2. Prepped for CAS:



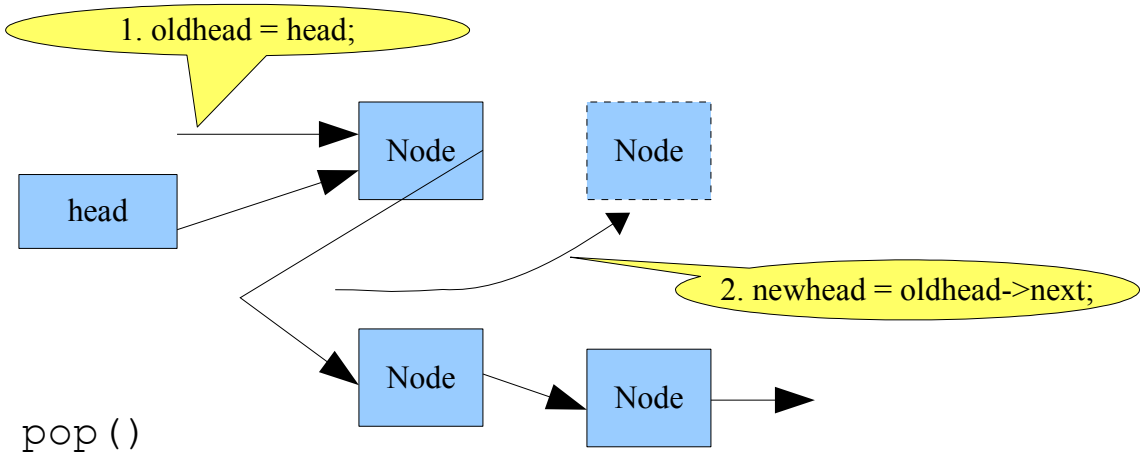

```
Val pop()
{
   Node * newhead;
   do
   {
      Node * oldhead = stack.head;
      if (!oldhead)
         throw StackEmpty();
      newhead = oldhead->next;
   }
   while (!head.CAS(oldhead, newhead));

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

AB – Maybe the first Node was deleted...



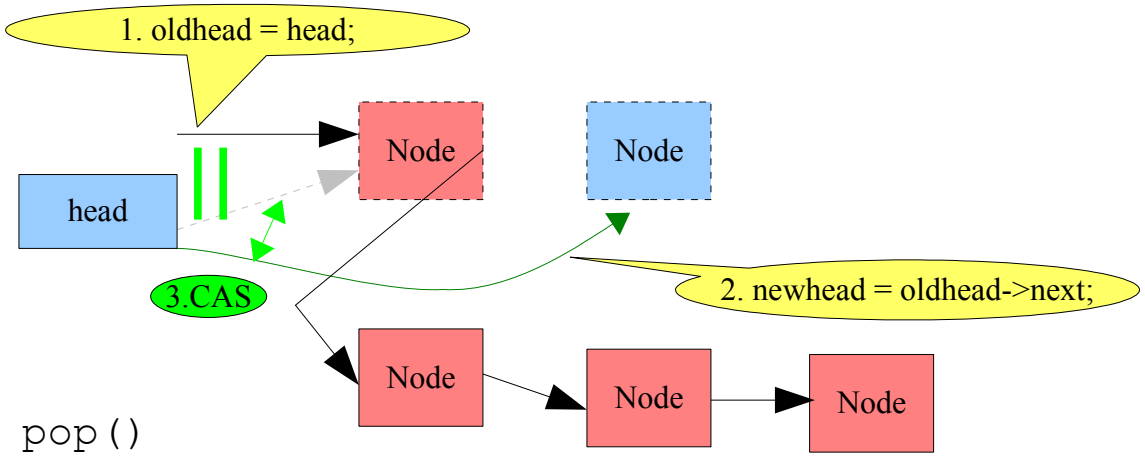

```
Val pop()
{
   Node * newhead;
   do
   {
      Node * oldhead = stack.head;
      if (!oldhead)
         throw StackEmpty();
      newhead = oldhead->next;
   }
   while (!head.CAS(oldhead, newhead));

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

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




```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;
        if (!oldhead)
            throw StackEmpty();
        newhead = oldhead->next;
    }
    while (!head.CAS(oldhead, newhead));

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

ABA – CAS Succeeds! (Yeah!?):




```
Val pop()
{
    Node * newhead;
    do
    {
        Node * oldhead = stack.head;
        if (!oldhead)
            throw StackEmpty();
        newhead = oldhead->next;
    }
    while (!head.CAS(oldhead, newhead));

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

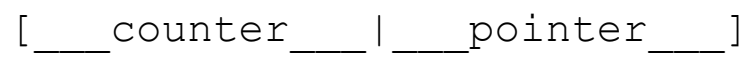
Our friend CAS has issues!

DWCAS to the Rescue!

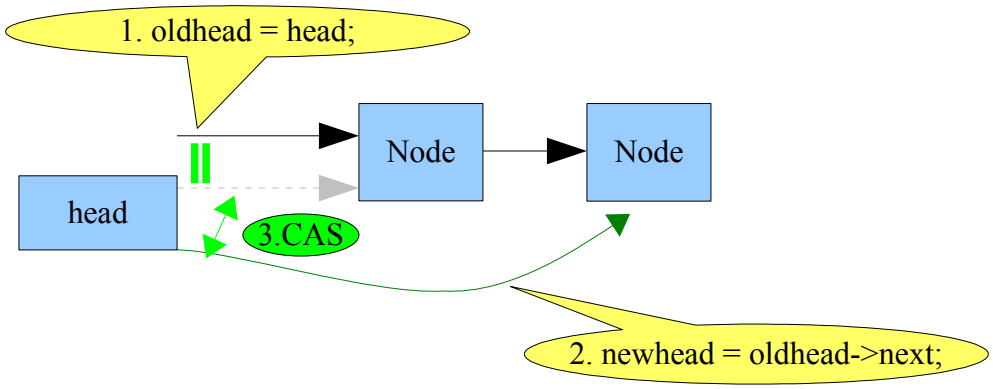
```
DWCAS([___word___|___word___], oldwide, newwide);
```

What should we do with that?

DWCAS to the Rescue!

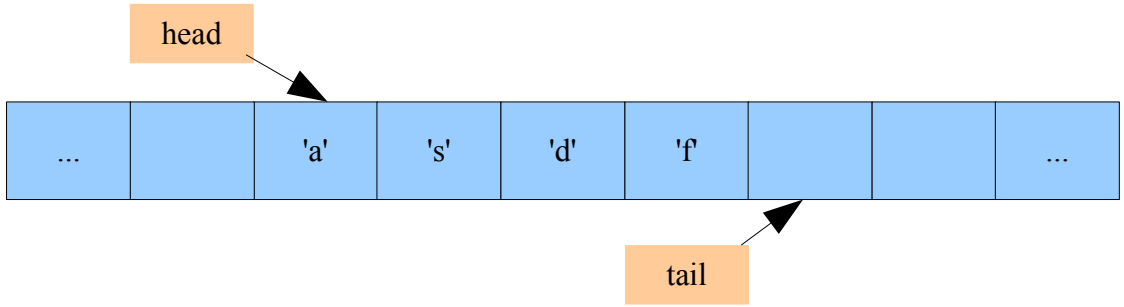


DWCAS – Pop from a lockfree stack

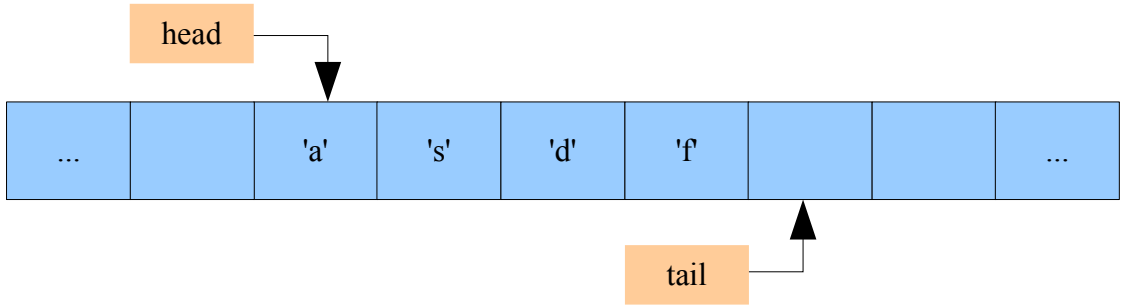



```
Val pop()
{
   NodePtr newhead;
   do
   {
      NodePtr oldhead = stack.head;
      if (!oldhead)
         throw StackEmpty();
      newhead = oldhead->next;
   }
   while (!DWCAS(&head, oldhead, newhead));

   Val val = oldhead->val;
   delete oldhead;
   return val;
}
NodePtr NodeAlloc(Val val)
{
    NodePtr ptr;
    ptr.ptr = new Node(val);
    ptr.count = atomic_inc(NodePtr::global_count);
    return NodePtr;
}
```

Circular Queue

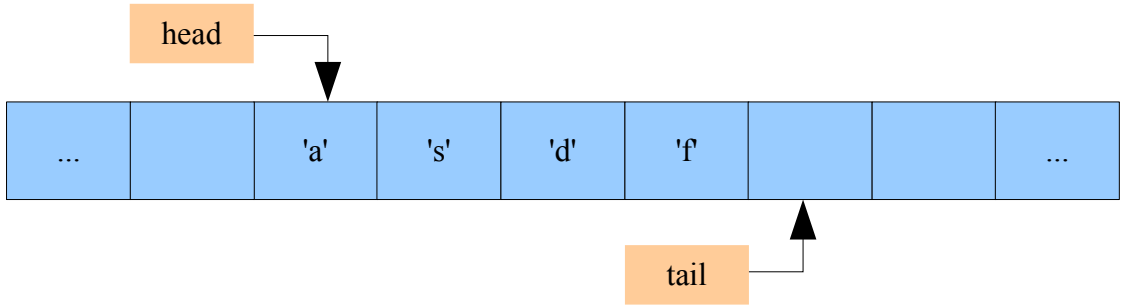


Single Threaded Circular Queue



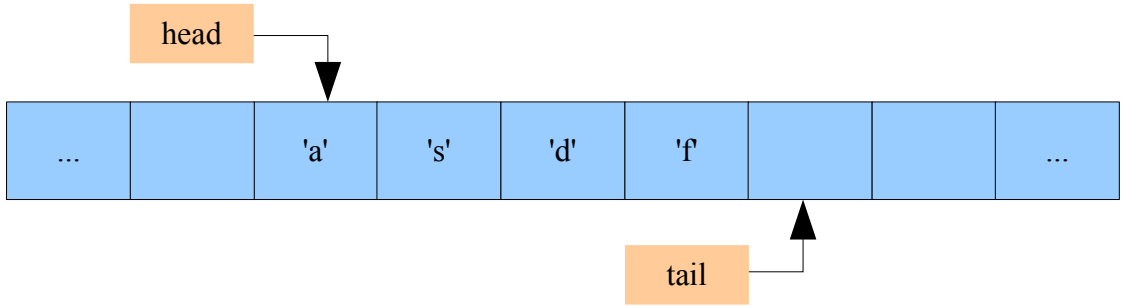

```
bool push(int item)
{
    if (tail == head - 1) // full?
        return false;
    array[tail] = item;
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item)
{
    if (head == tail) // empty?
        return false;
    item = array[head];
    head = (head + 1) % size;
    return true;
}
```

Single Threaded Circular Queue



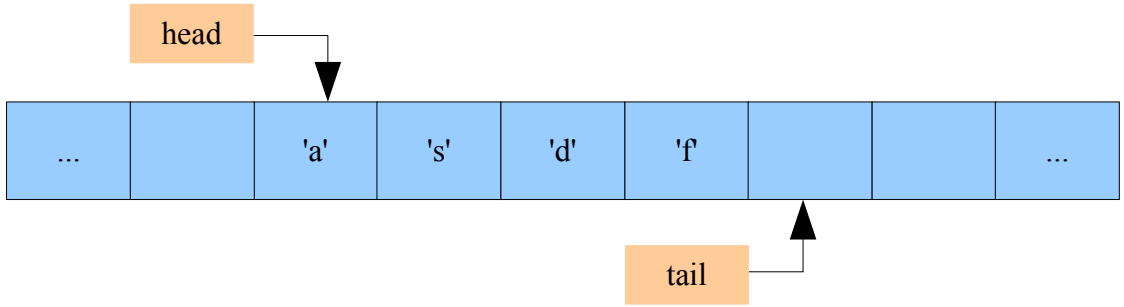

```
bool push(int item)
{
    if ((tail + 1) % size == head) // full?
        return false;
    array[tail] = item;
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item)
{
    if (head == tail) // empty?
        return false;
    item = array[head];
    head = (head + 1) % size;
    return true;
}
```

"Atomic"? Circular Queue



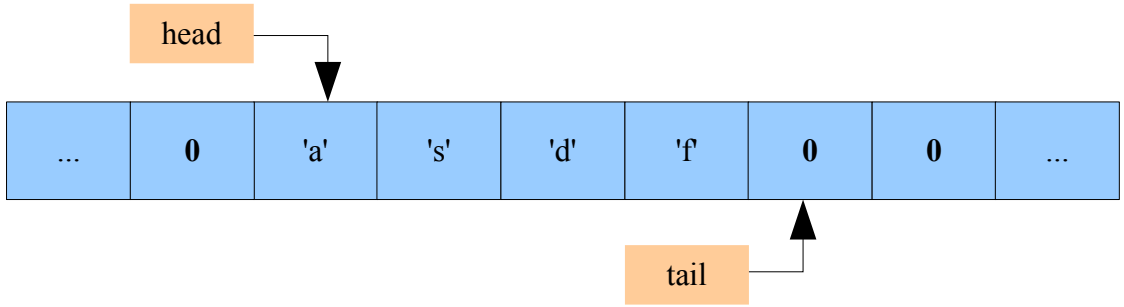

```
std::atomic<size_t> head;
std::atomic<size_t> tail;

bool push(int item) {
    if ((tail + 1) % size == head)
        return false;
    array[tail] = item;
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    if (head == tail)
        return false;
    item = array[head];
    head = (head + 1) % size;
    return true;
}
```

"Atomic"? Circular Queue



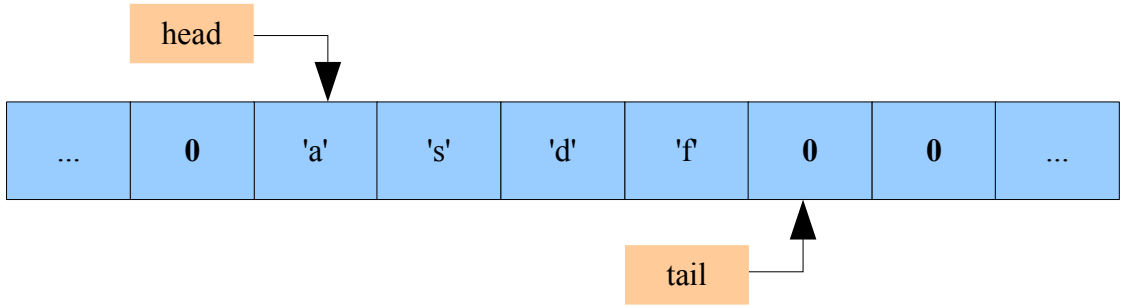

```
std::atomic<size_t> head;
std::atomic<size_t> tail;

bool push(int item) {
    if ((tail + 1) % size == head)
        return false;
    array[tail] = item;
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    if (head == tail)
        return false;
    item = array[head];
    head = (head + 1) % size;
    return true;
}
```

Single Threaded Circular Queue... of non-zero ints



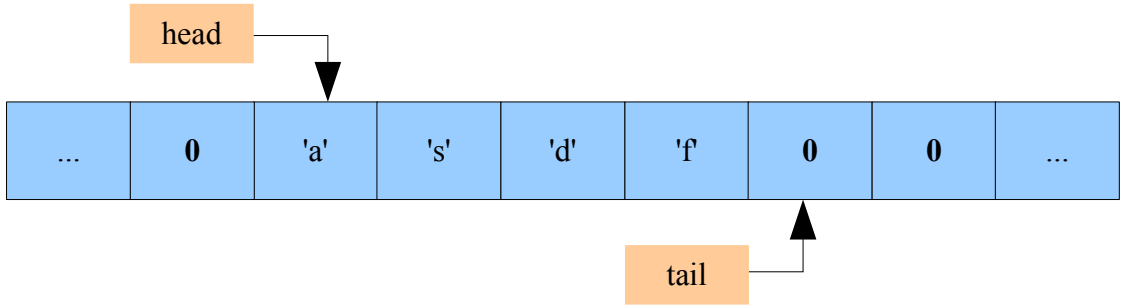

```
bool push(int item) {
    if (array[tail] != 0) //occupied => full
        return false;
    array[tail] = item;
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    if (array[head] == 0) // empty?
       return false;
    item = array[head];
    array[head] = 0;
    head = (head + 1) % size;
    return true;
}
```

Single Threaded Circular Queue... of non-zero ints



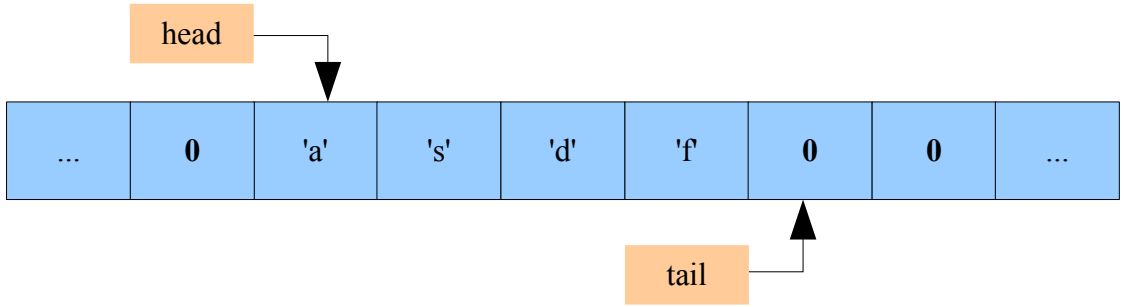

```
bool push(int item) {
    if (array[tail] != 0) //occupied => full
        return false;
    array[tail] = item;
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    item = array[head];
    if (item == 0) // empty?
       return false;
    array[head] = 0;
    head = (head + 1) % size;
    return true;
}
```

"Atomic"? Circular Queue... of non-zero ints



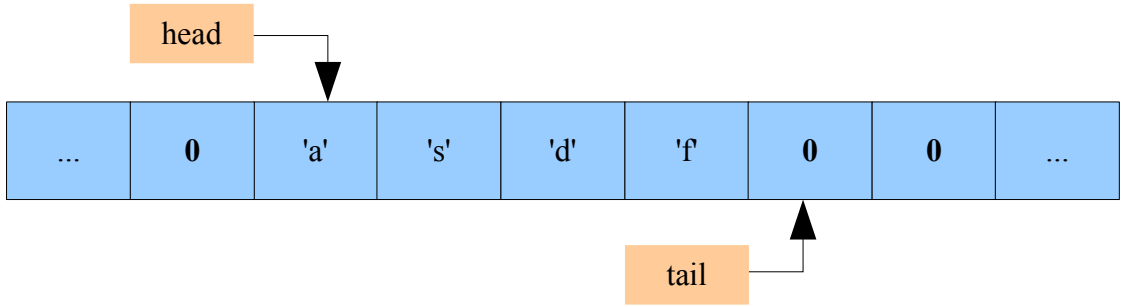

```
std::atomic<int> array[size];

bool push(int item) {
    if (array[tail].load(memory_order_???) != 0))
        return false;
    array[tail].store(item, memory_order_???);
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    item = array[head].load(memory_order_???);
    if (item == 0)
       return false;
    array[head].store(0, memory_order_???);
    head = (head + 1) % size;
    return true;
}
```

"Atomic"? Circular Queue... of non-zero ints



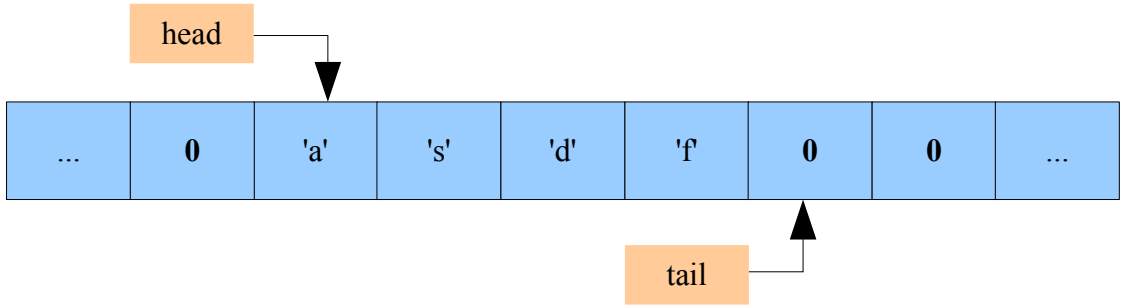

```
std::atomic<int> array[size];

bool push(int item) {
    if (array[tail].load(memory_order_relaxed) != 0))
        return false;
    array[tail].store(item, memory_order_relaxed);
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    item = array[head].load(memory_order_relaxed);
    if (item == 0)
       return false;
    array[head].store(0, memory_order_relaxed);
    head = (head + 1) % size;
    return true;
}
```

"Atomic"? Circular Queue... of non-zero ints



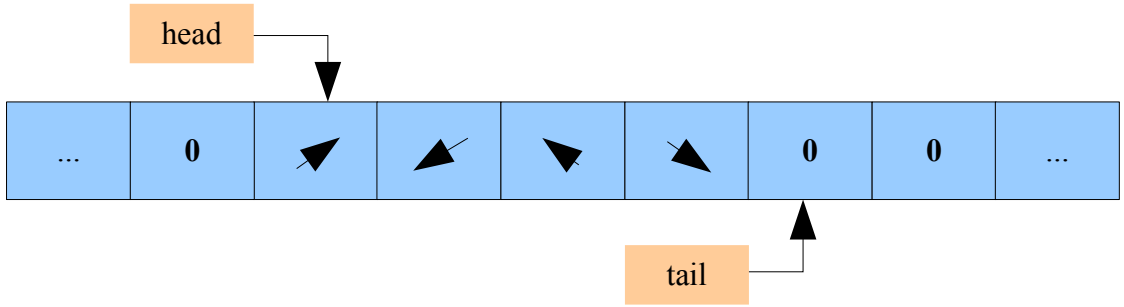

```
std::atomic<int> array[size];

bool push(int item) {
    if (array[tail].load(memory_order_relaxed) != 0))
        return false;
    array[tail].store(item, memory_order_relaxed);
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    item = array[head].load(memory_order_relaxed);
    if (item == 0)
       return false;
    array[head].store(0, memory_order_relaxed);
    head = (head + 1) % size;
    return true;
}
```

SPSC Circular Queue of non-zero ints



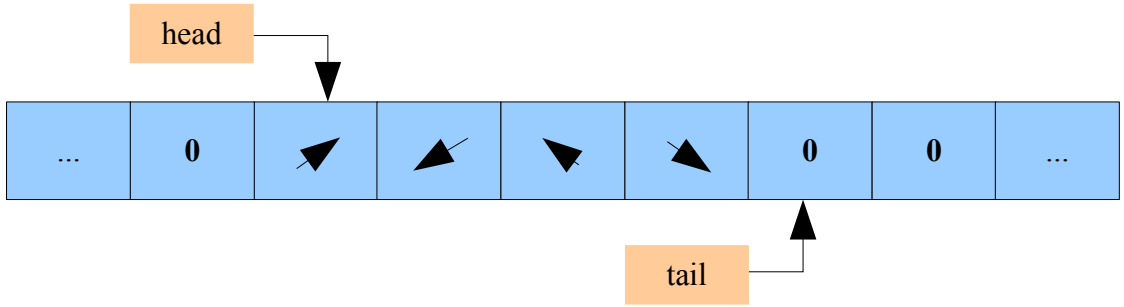

```
std::atomic<int> array[size];

bool push(int item) {
    if (array[tail].load(memory_order_relaxed) != 0))
        return false;
    array[tail].store(item, memory_order_relaxed);
    tail = (tail + 1) % size;
    return true;
}
bool pop(int & item) {
    item = array[head].load(memory_order_relaxed);
    if (item == 0)
       return false;
    array[head].store(0, memory_order_relaxed);
    head = (head + 1) % size;
    return true;
}
```

SPSC Circular Queue of non-null pointers



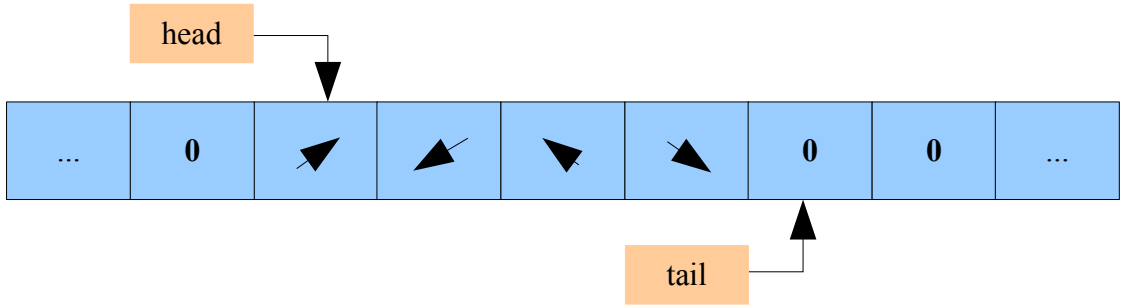

```
std::atomic<Item *> array[size];

bool push(Item * itemPtr) {
    if (array[tail].load(memory_order_relaxed) != 0))
        return false;
    array[tail].store(itemPtr, memory_order_relaxed);
    tail = (tail + 1) % size;
    return true;
}
bool pop(Item * & itemPtr) {
    itemPtr = array[head].load(memory_order_relaxed);
    if (itemPtr == 0)
       return false;
    array[head].store(0, memory_order_relaxed);
    head = (head + 1) % size;
    return true;
}
```

SPSC Circular Queue of non-null pointers




```
std::atomic<Item *> array[size];

bool push(Item * itemPtr) {
    if (array[tail].load(memory_order_relaxed) != 0))
        return false;
    array[tail].store(itemPtr, memory_order_relaxed);
    tail = (tail + 1) % size;
    return true;
}
bool pop(Item * & itemPtr) {
    itemPtr = array[head].load(memory_order_relaxed);
    if (itemPtr == 0)
       return false;
    array[head].store(0, memory_order_relaxed);
    head = (head + 1) % size;
    return true;
}
```

SPSC Circular Queue of non-null pointers




```
std::atomic<Item *> array[size];

bool push(Item * itemPtr) {
    if (array[tail].load(memory_order_relaxed) != 0))
        return false;
    array[tail].store(itemPtr, memory_order_release);
    tail = (tail + 1) % size;
    return true;
}
bool pop(Item * & itemPtr) {
    itemPtr = array[head].load(memory_order_acquire);
    if (itemPtr == 0)
       return false;
    array[head].store(0, memory_order_relaxed);
    head = (head + 1) % size;
    return true;
}
```

Conclusions...

A Guide to Threaded Coding

"Lock-free coding is the <u>last</u> thing you want to do."

1. Forget what you learned in Kindergarten
   (stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm
6. GOTO 1

∞. Lock-free

“Use Locks!”