HOW UBISOFT MONTREAL DEVELOPS GAMES FOR MULTICORE

Before & After C++11

Jeff Preshing
Technical Architect
Ubisoft Montreal


UBISOFT
CAFÉ VIENNE
CAFÉ VIENNE
PETRO-CANADA


Watch Dogs


Assassin's Creed Unity

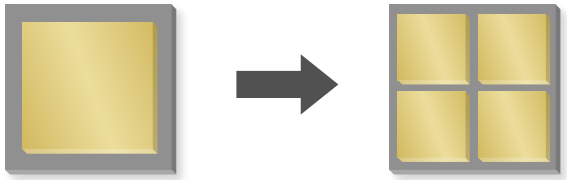
MULTICORE


Rainbow Six: Siege


Far Cry 4

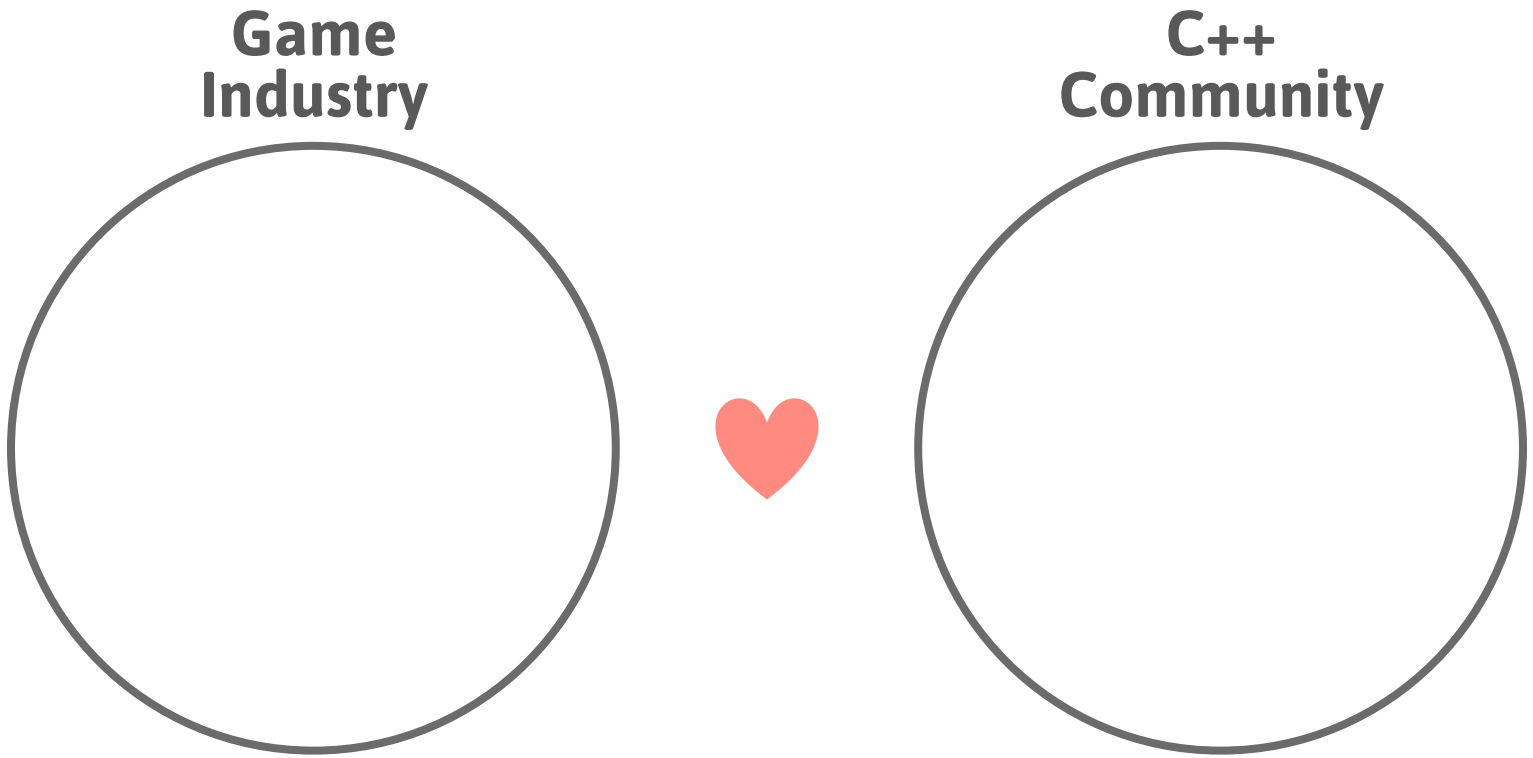
MULTICORE

several CPU cores on a single processor



- Same instruction set
- Same address space
- Existing threads can run on any core

*Popular way to offer **power***



We all want to exploit multicore!

PART ONE

Multicore Programming at Ubisoft

PART TWO

The C++11 Atomic Library

Part One

Multicore Programming at Ubisoft

GENERAL-PURPOSE HARDWARE THREADS

available for game use

2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014

PlayStation 2

1 MIPS

Xbox

1 x86

Xbox 360

6 PowerPC

PlayStation 3

2 PowerPC
+6 SPU

PlayStation 4

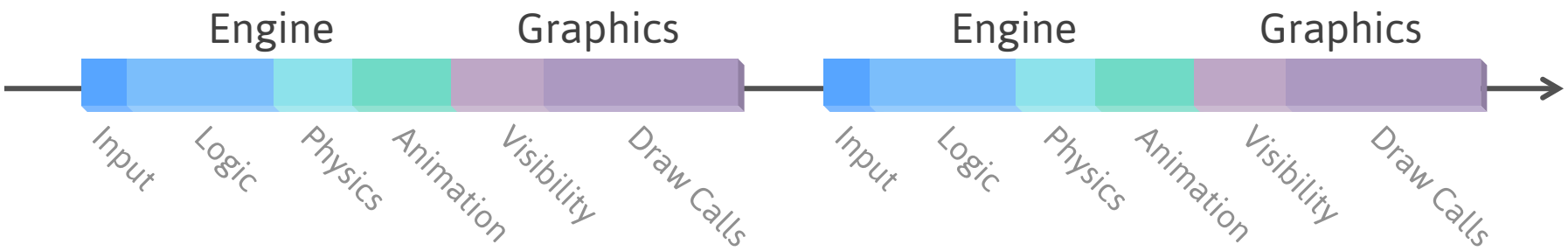
6 x64

Xbox One

6 x64

SINGLE-THREADED MAIN LOOP

in the early 2000s



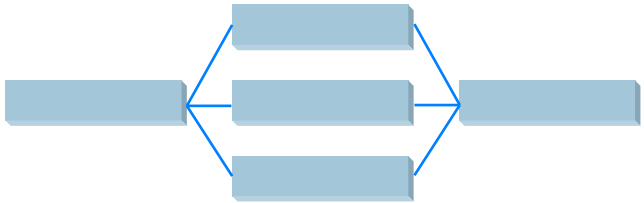
THREE THREADING PATTERNS

to exploit multicore

Pipelining Work

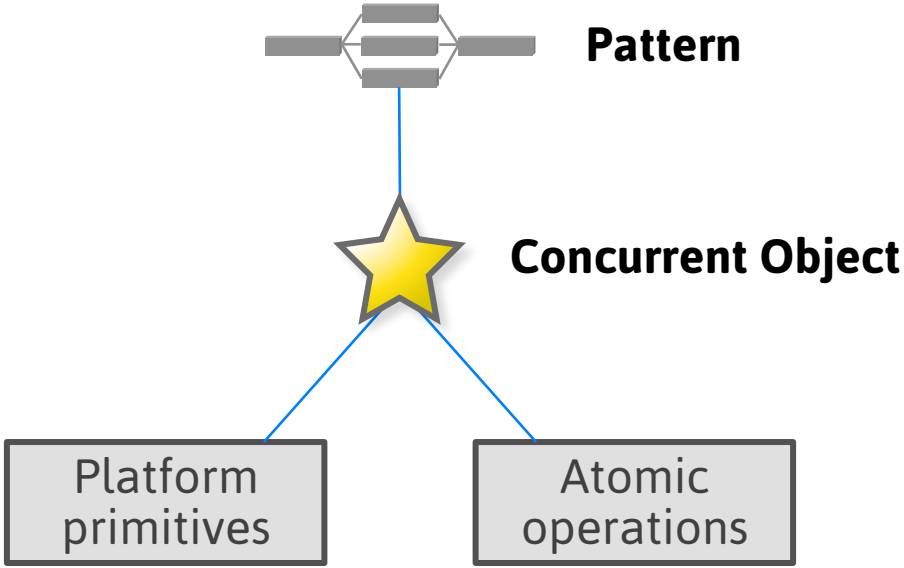
Dedicated Threads

Task Schedulers



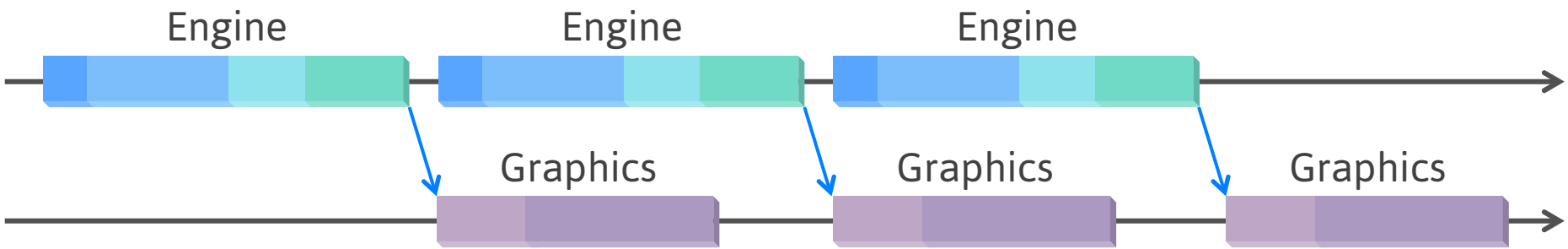
CONCURRENT OBJECTS

Multiple threads operate concurrently on the object with at least one thread modifying its state.

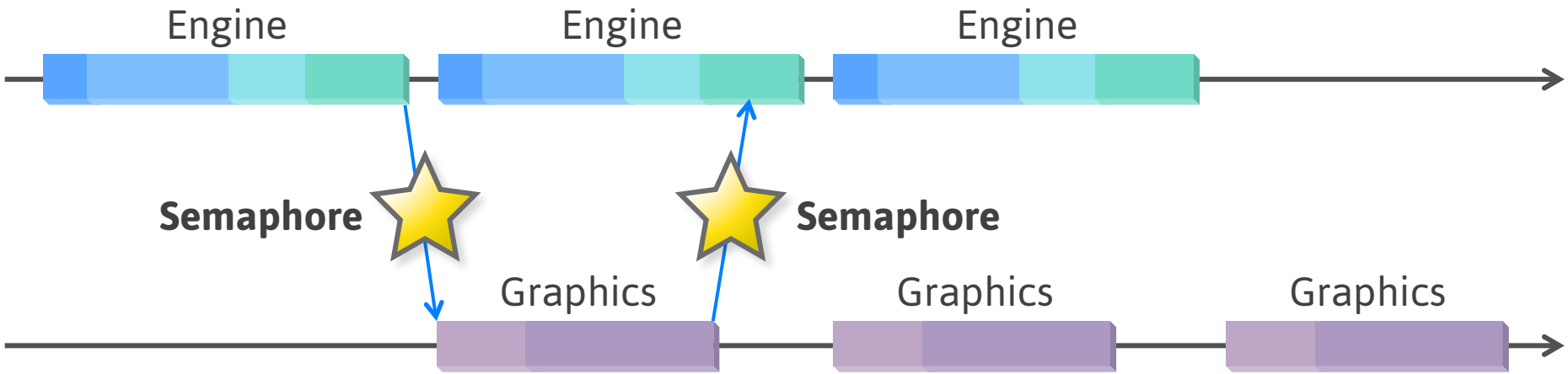


Pipelining Work

PIPELINED GRAPHICS

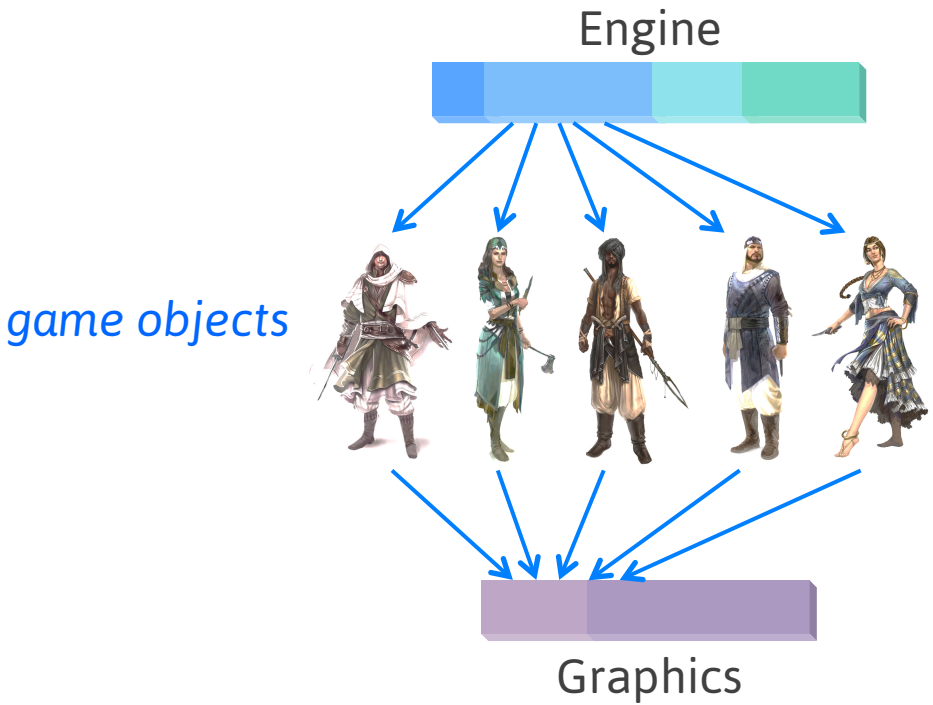


PIPELINED GRAPHICS



PIPELINED GRAPHICS

how to avoid concurrent modifications?



DOUBLE-BUFFERED GRAPHICS STATE

One approach

```
struct Object
{
    ...
    Matrix xform[2];
    ...
};
```

SEPARATE GRAPHIC OBJECTS

Another approach

```
struct Object
{
    ...
    Matrix xform;
    GraphicObject* gfxObject;
    ...
};
```

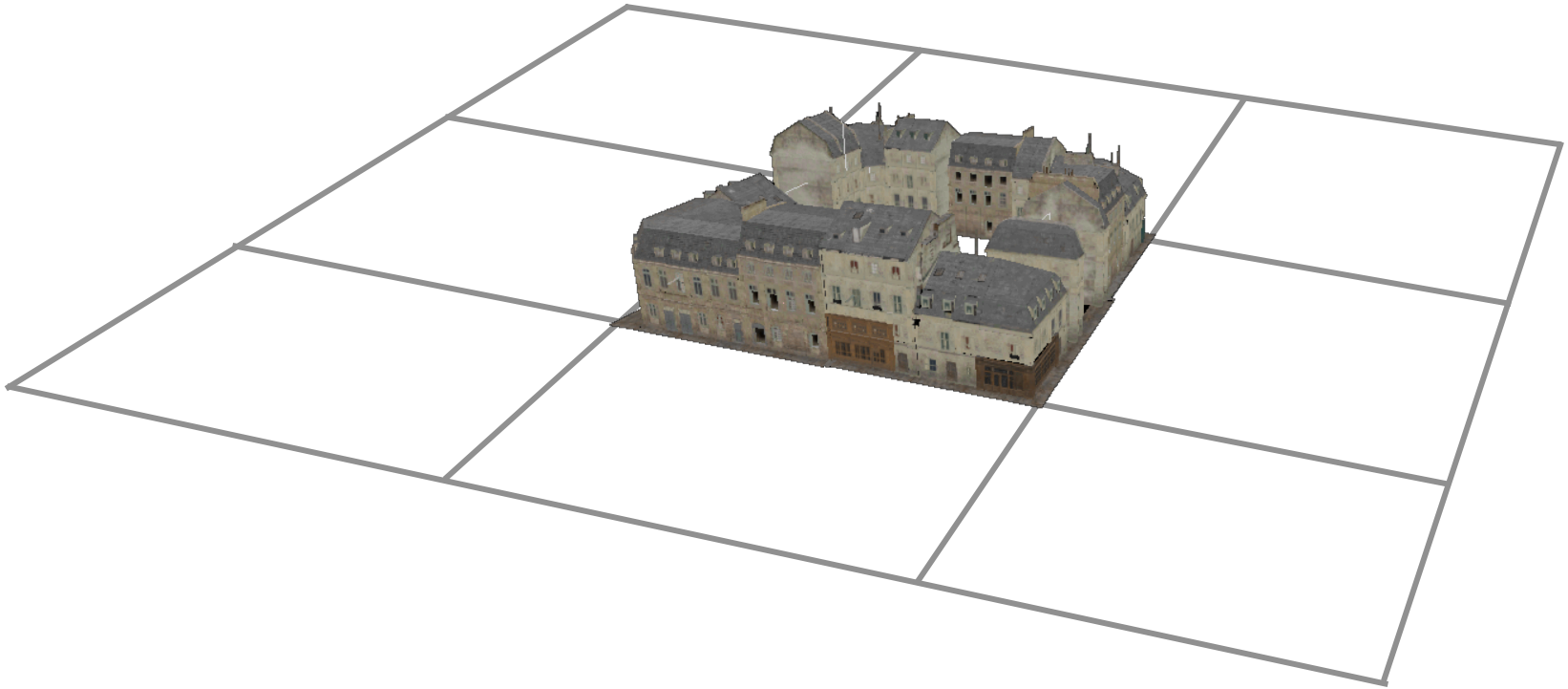
```
struct GraphicObject
{
    ...
    Matrix xform;
    ...
};
```

Copy at start of frame

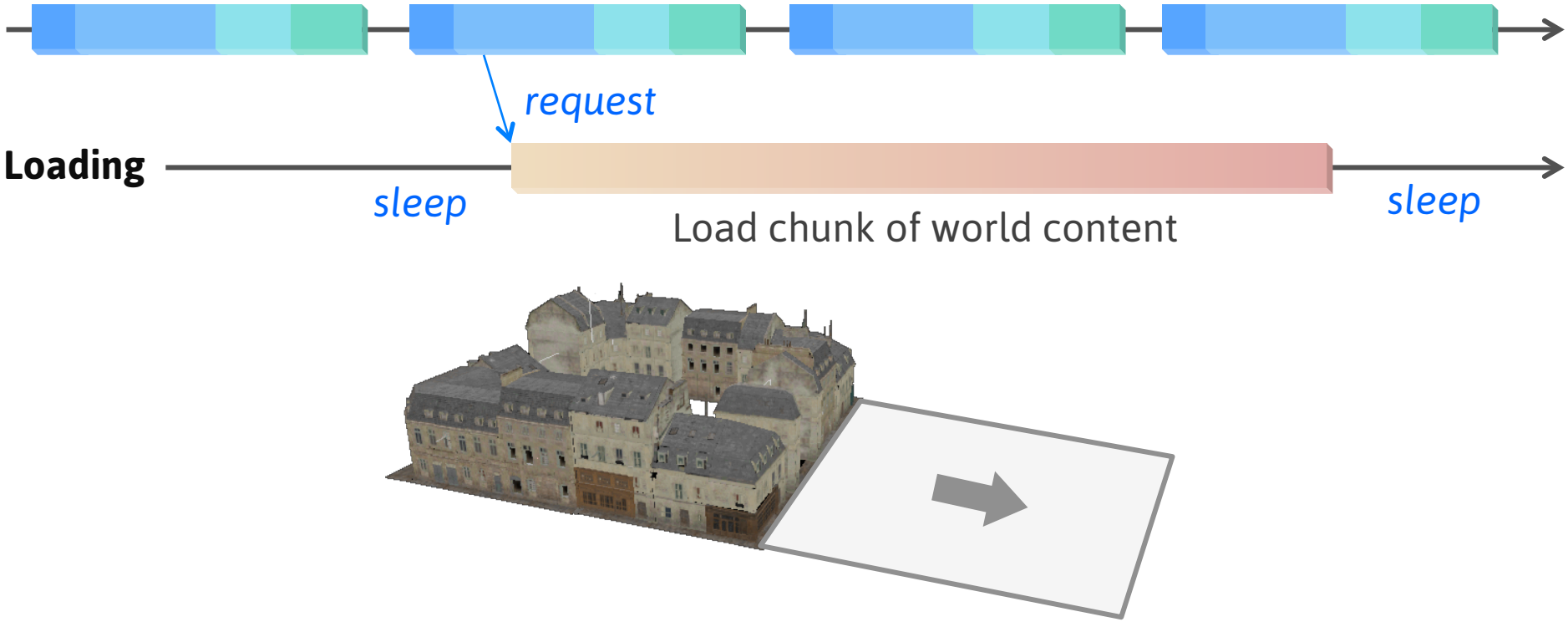
Dedicated Threads

CONTENT STREAMING

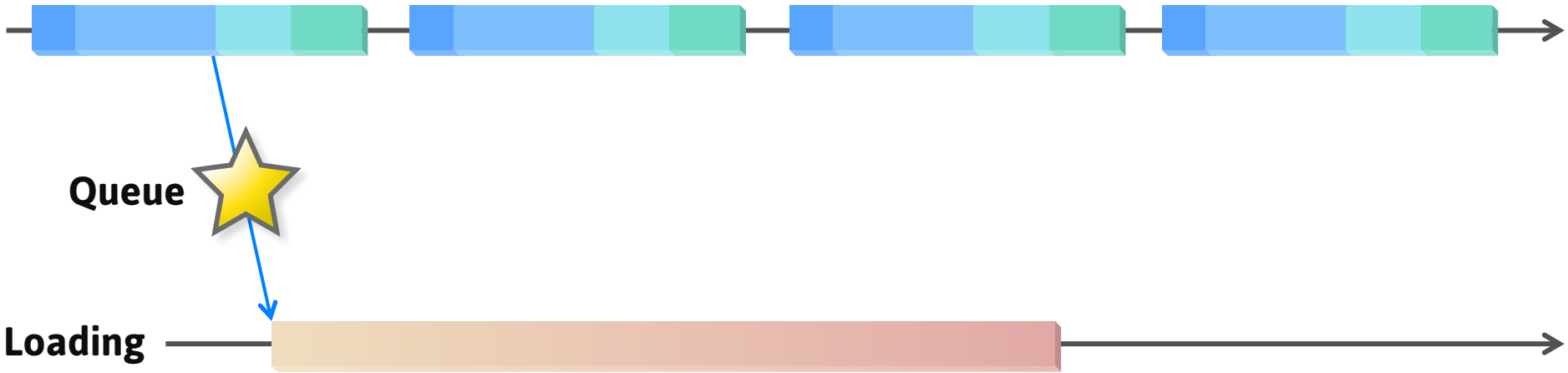
We don’t load the entire game environment in memory at once.



DEDICATED LOADING THREAD



DEDICATED LOADING THREAD



WAKING UP THE LOADING THREAD

```
ThreadSafeQueue<Request> requests;
Event workAvailable;
```

Engine Thread

```
requests.push(r);
workAvailable.signal();
```

Loading Thread

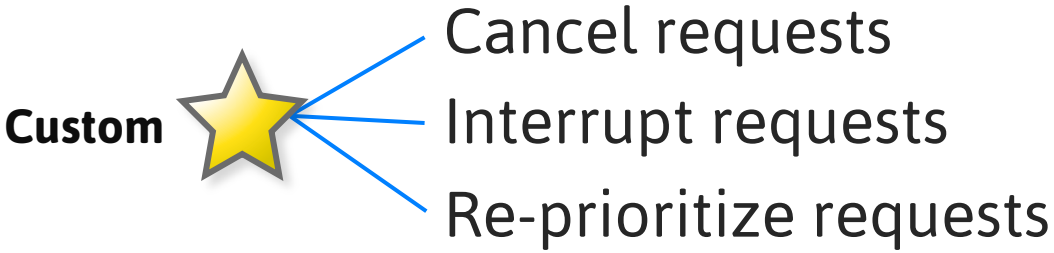
```
for (;;)
{
    workAvailable.waitAndReset();
    while (r = requests.tryPop())
    {
        processLoadRequest(r);
    }
}
```

Event signaled → threads pass through

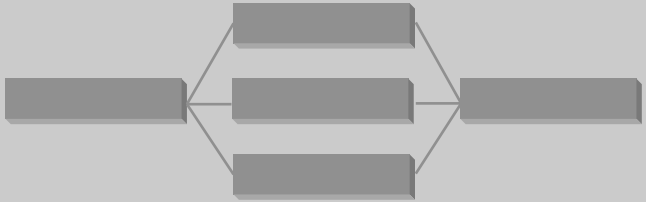
Event reset → threads wait

IMPROVING ON THE QUEUE

Many design choices

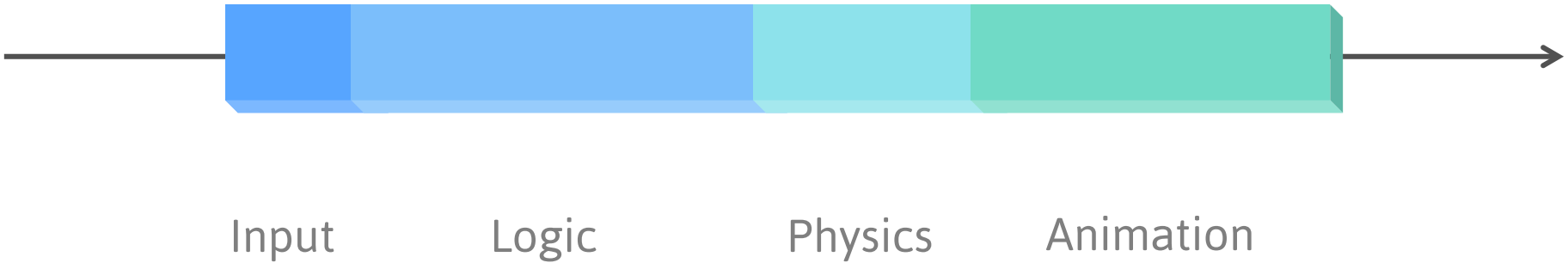


Task Schedulers



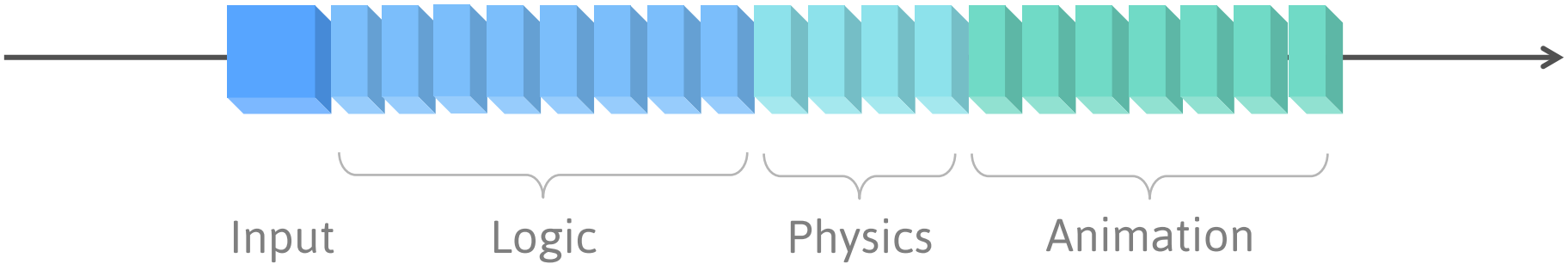
FINE-GRAINED PARALLELISM

Motivation for a task scheduler



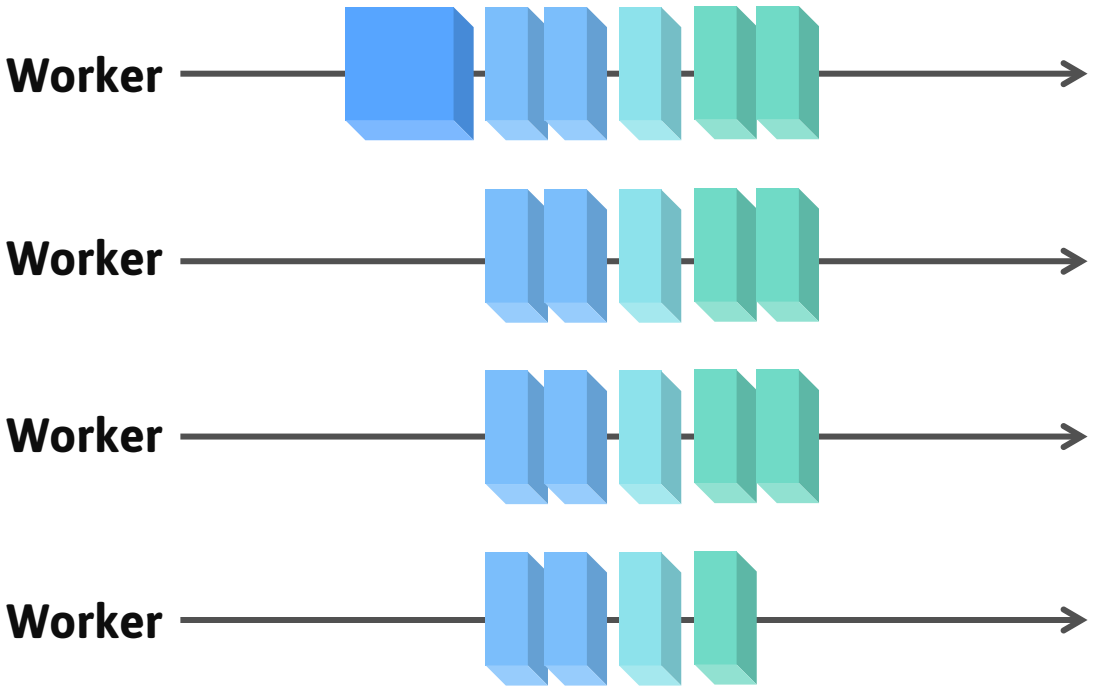
FINE-GRAINED PARALLELISM

Motivation for a task scheduler

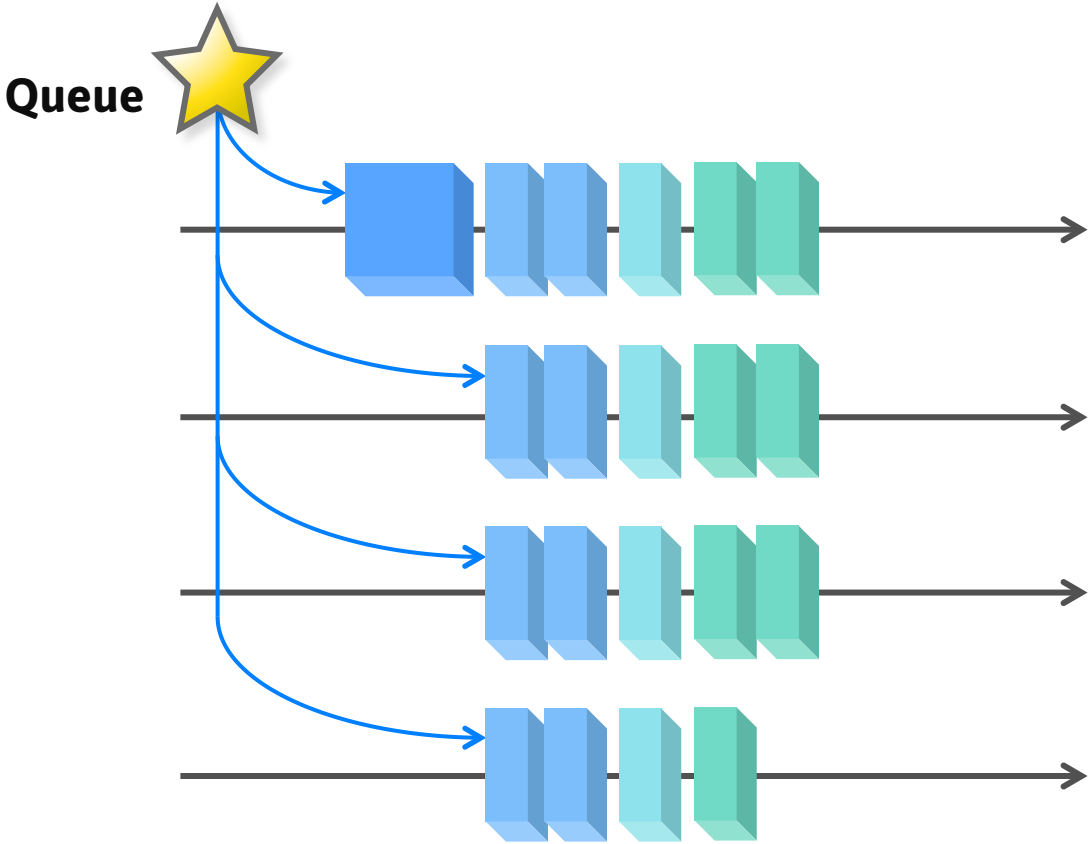


FINE-GRAINED PARALLELISM

Motivation for a task scheduler



SIMPLE TASK QUEUE



WAKING UP THE WORKER THREADS

```
ThreadSafeQueue<Task> tasks;
Event workAvailable[numThreads];
```

Submitting Thread

```
tasks.push(t);
for (int i = 0; i < numThreads; i++)
    workAvailable.signal();
```

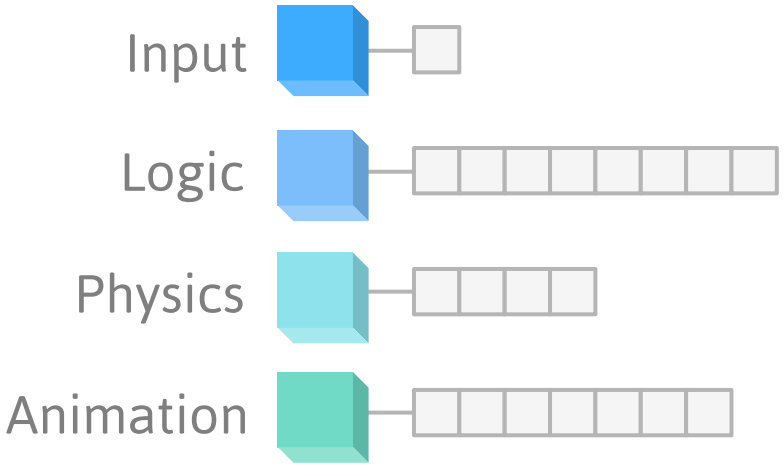
Worker Thread

```
for (;;)
{
    workAvailable[thread].waitAndReset();
    while (t = tasks.tryPop())
    {
        t->Run();
    }
}
```

One event for each worker thread

TASK GROUPS

Grouping work units together into larger tasks

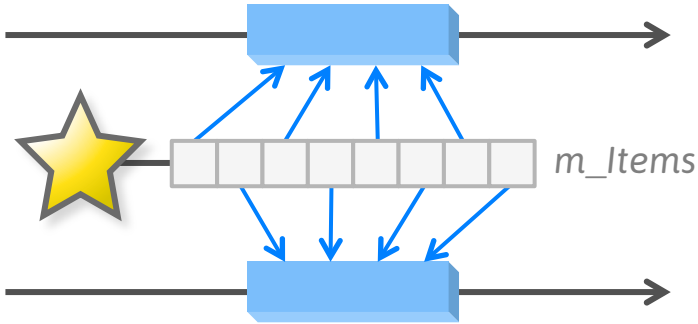



```
class TaskGroup
{
private:
    Array<Item*> m_Items;
    ...
};
```

Each TaskGroup keeps an array of items to update in parallel.

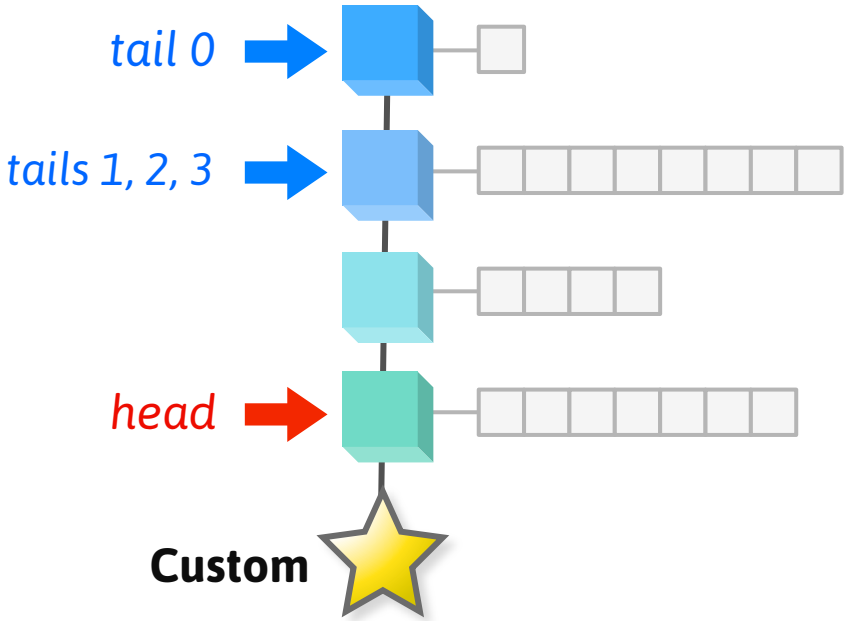
TASK GROUPS

```
class TaskGroup
{
private:
    vector<Item*> m_Items;
    volatile int m_Index;

public:
    void Run()
    {
        for (;;)
        {
            int index = AtomicIncrement(m_Index);
            if (index >= m_Items.size())
                break;
            m_Items[index]->Run();
        }
    }
};
```



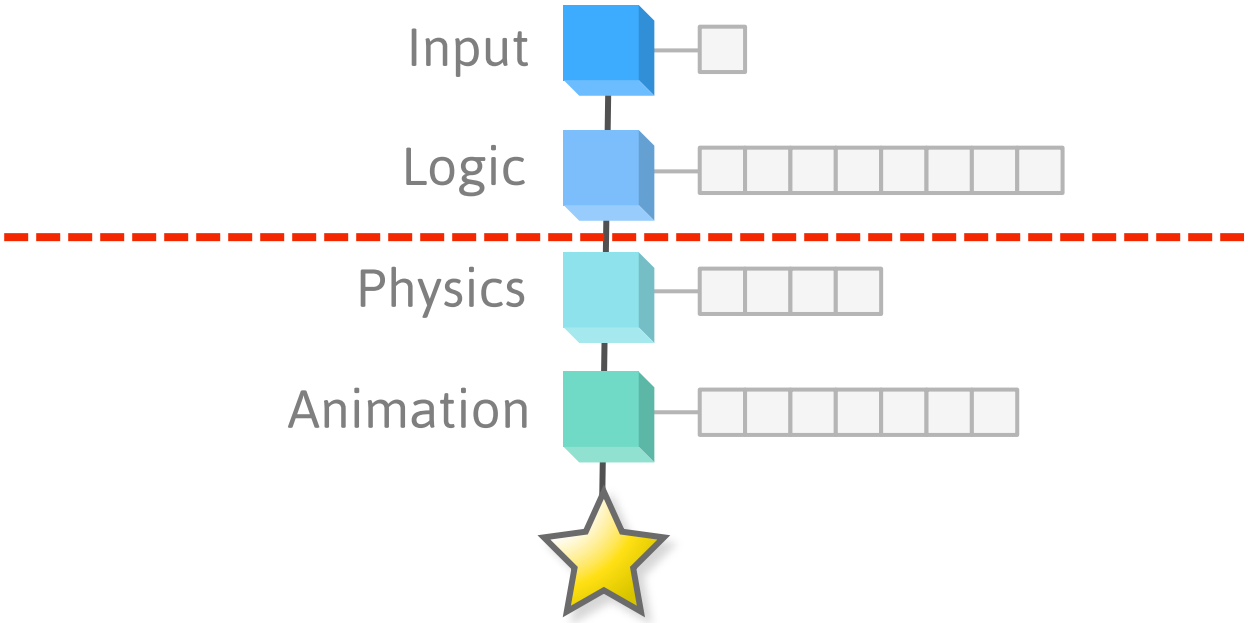
Multiple threads work on the same TaskGroup.

NOT A SIMPLE QUEUE ANYMORE



Could be a queue with separate tails for each worker.

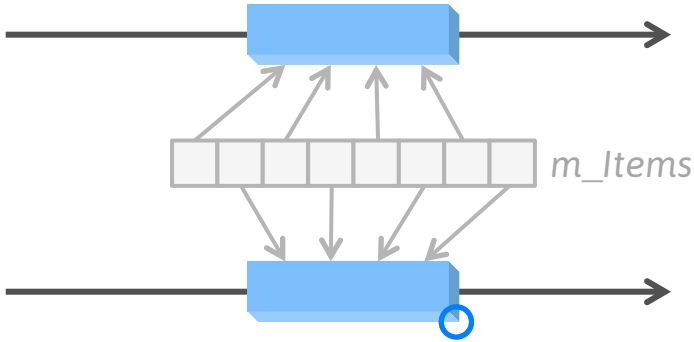
MANAGING DEPENDENCIES



No physics tasks before ***all*** logic tasks.

MANAGING DEPENDENCIES

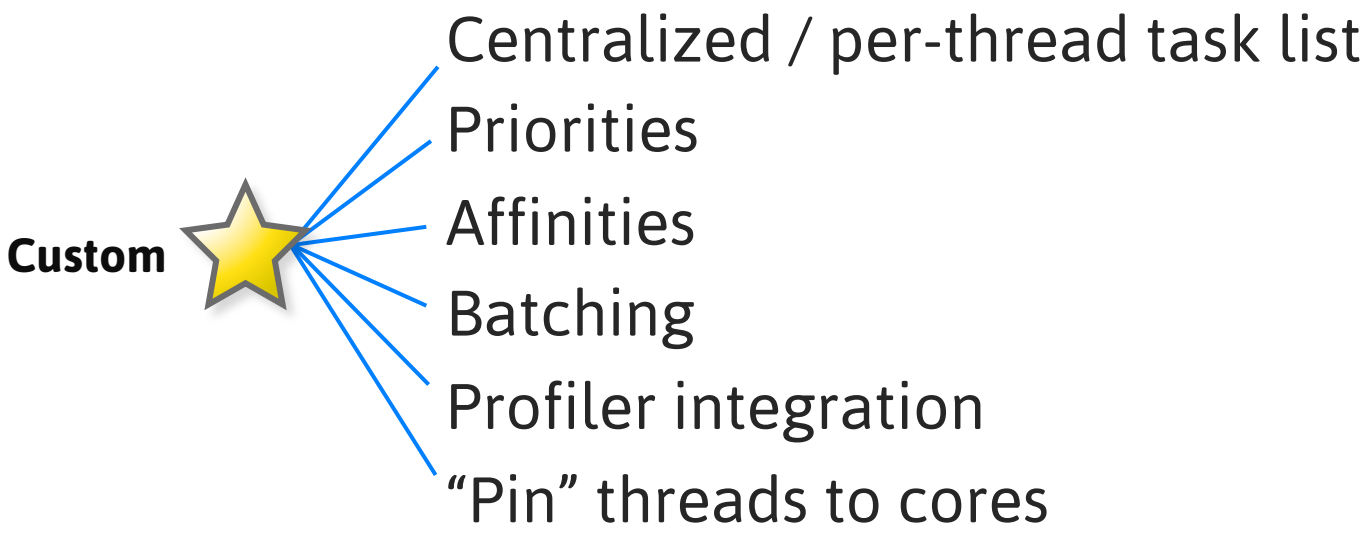
```
class TaskGroup
{
private:
    vector<Item*> m_Items;
    volatile int m_Index;
    volatile int m_RemainingCount;
    ...

public:
    void Run() {
        int count = 0;
        for (;;) {
            int index = AtomicIncrement(m_Index);
            if (index >= m_Items.size())
                break;
            m_Items[index]->Run();
            count++;
        }
        if (count > 0 && AtomicSubtract(m_Index, count) == 0)
            AddDependencies();
    }
};
```



The thread that finishes the last item schedules the next TaskGroup.

IMPROVING ON THE TASK SCHEDULER

Many design choices



Atomic Operations

GAME ATOMICS

Typical portable library

Declaration	`volatile int A;`
Load/Store	`A = 1;` `int a = A;`
Ordering	`LIGHTWEIGHT_FENCE();` `FULL_FENCE();`

Read-Modify-Write	`AtomicIncrement(A);`
	`AtomicCompareExchange(A, …, …);`
	`...`

FENCE MACROS

What's the difference?

Used more often

```
LIGHTWEIGHT_FENCE();
```

Orders loads from memory
Orders stores to memory

Does the job of:

```
atomic_thread_fence(memory_order_acquire);
atomic_thread_fence(memory_order_release);
atomic_thread_fence(memory_order_acq_rel);
```

```
FULL_FENCE();
```

... all that, plus:

Commits stores before next load

Does the job of:

```
atomic_thread_fence(memory_order_seq_cst);
```

HOW THEY'RE IMPLEMENTED

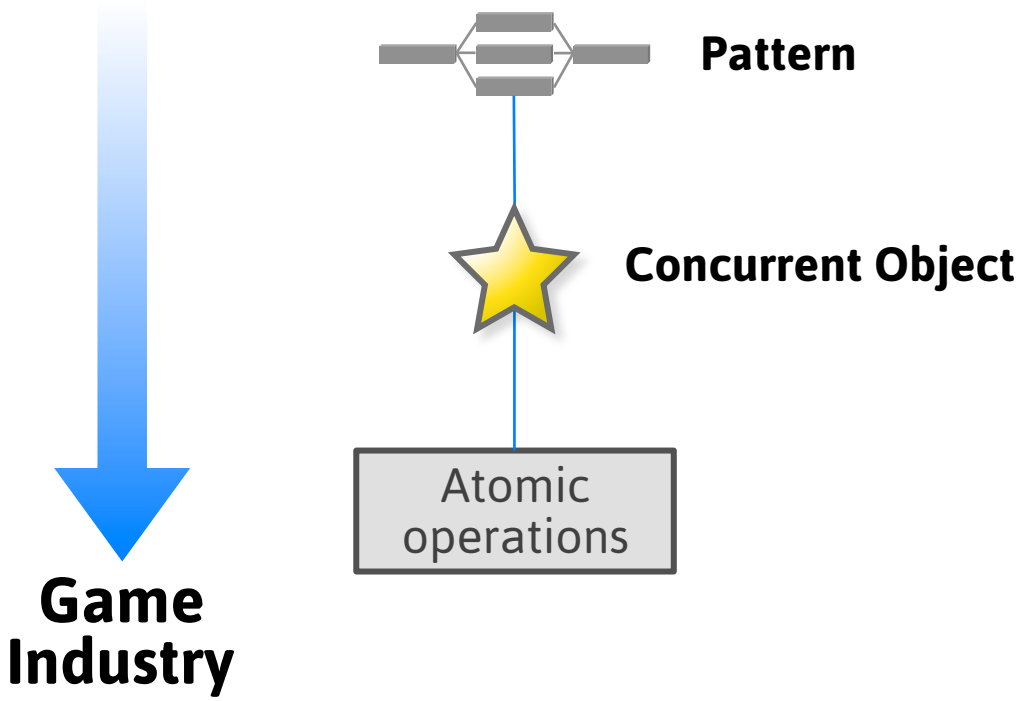
on processors we care about

		x86/64	PowerPC	ARMv7
Declaration	`volatile int A;`			
Load/Store	`A = 1;` `int a = A;`	`mov %, %` `mov %, %`	`ld %, %` `st %, %`	`ldr %, %` `str %, %`
Ordering	`LIGHTWEIGHT_FENCE();` `FULL_FENCE();`	*(compiler barrier)* `mfence`	`lwsync` `hwsync`	`dmb` `dmb`

Read-Modify-Write	`AtomicIncrement(A);`
	`AtomicCompareExchange(A, …, …);`
	`...`
		`lock inc`
		`lock cmpxchg`
			`lwarx`
			`...`
			`stwcx`
				`ldrex`
				`...`
				`strex`

ATOMIC OPERATIONS

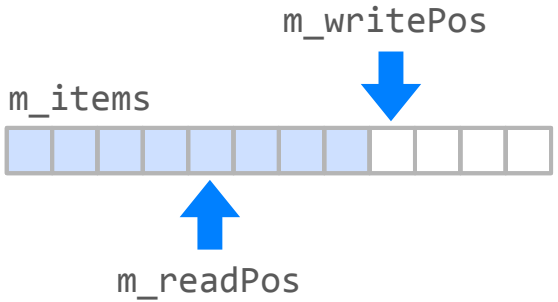
How we ended up using them



EXAMPLE

Capped wait-free queue

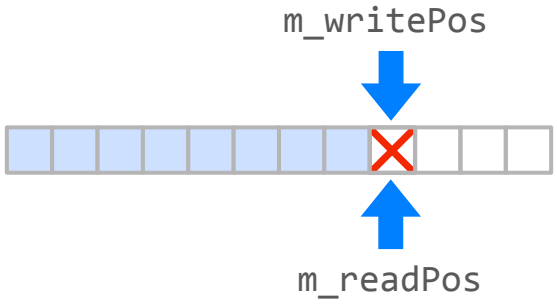
```
template <class T, int size>
class CappedSPSCQueue
{
private:
    T m_items[size];
    volatile int m_writePos;
    int m_readPos;

public:
    CappedSPSCQueue() : m_writePos(0), m_readPos(0) {}
    bool tryPush(const T& item) { ... }
    bool tryPop(T& item) { ... }
};
```



Single producer, single consumer

EXAMPLE

Capped wait-free queue




```
bool tryPush(const T& item)
{
    int w = m_writePos;
    if (w >= size)
        return false;
    m_items[w] = item;
    m_writePos = w + 1;
    return true;
}
```

reorder

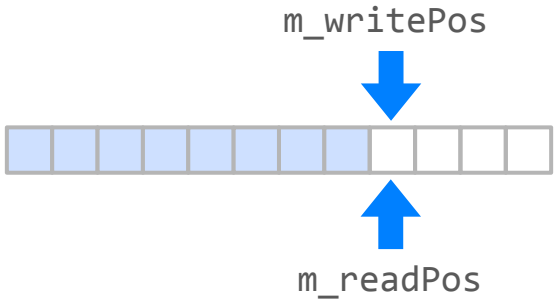
BROKEN

reorder

```
bool tryPop(T& item)
{
    int w = m_writePos;
    if (m_readPos >= w)
        return false;
    item = m_items[m_readPos];
    m_readPos++;
    return true;
}
```

EXAMPLE

Capped wait-free queue




```
bool tryPush(const T& item)
{
    int w = m_writePos;
    if (w >= size)
        return false;
    m_items[w] = item;
    LIGHTWEIGHT_FENCE();
    m_writePos = w + 1;
    return true;
}
```

```
bool tryPop(T& item)
{
    int w = m_writePos;
    if (m_readPos >= w)
        return false;
    LIGHTWEIGHT_FENCE();
    item = m_items[m_readPos];
    m_readPos++;
    return true;
}
```

RECAP:
Multicore programming at Ubisoft

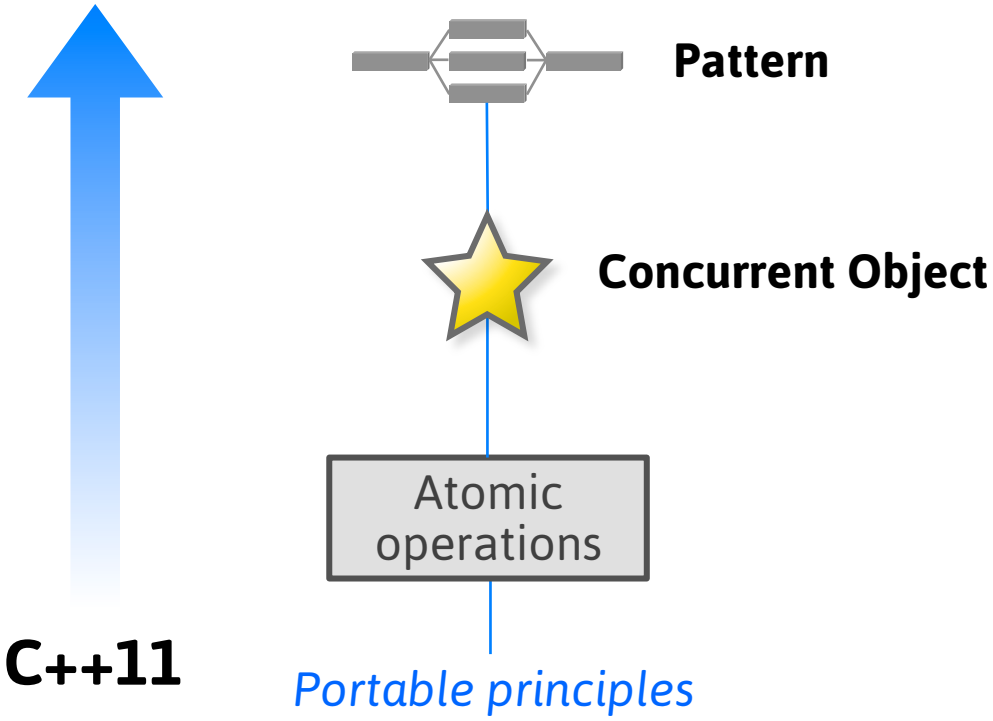
- Three threading patterns
- Lots of custom concurrent objects
- Atomic operations for high contention objects
- We learned by doing

Part Two

The C++11 Atomic Library

ATOMIC OPERATIONS

in C++11

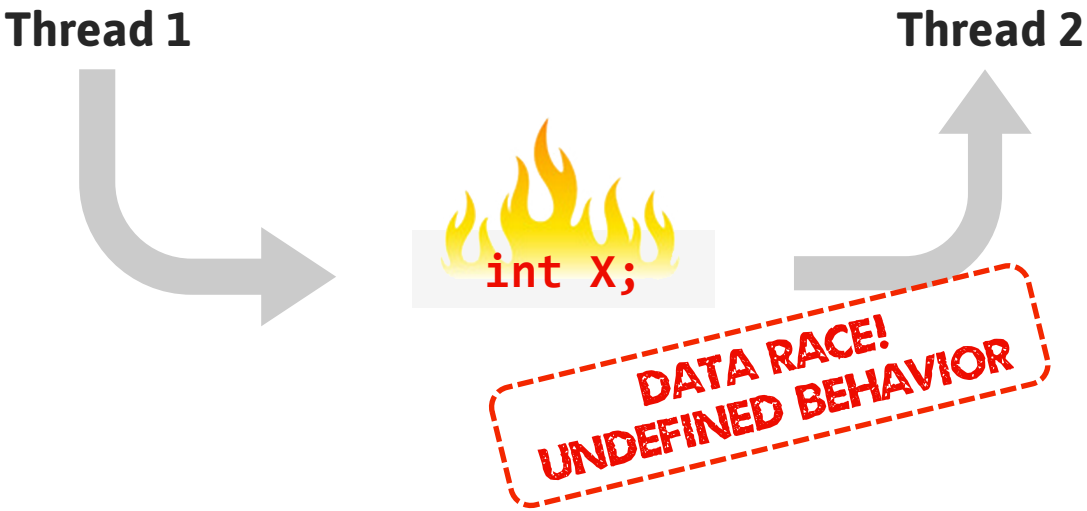


C++11 FORBIDS DATA RACES

If multiple threads access the same variable concurrently, and at least one thread modifies it, all threads must use C++11 atomic operations.

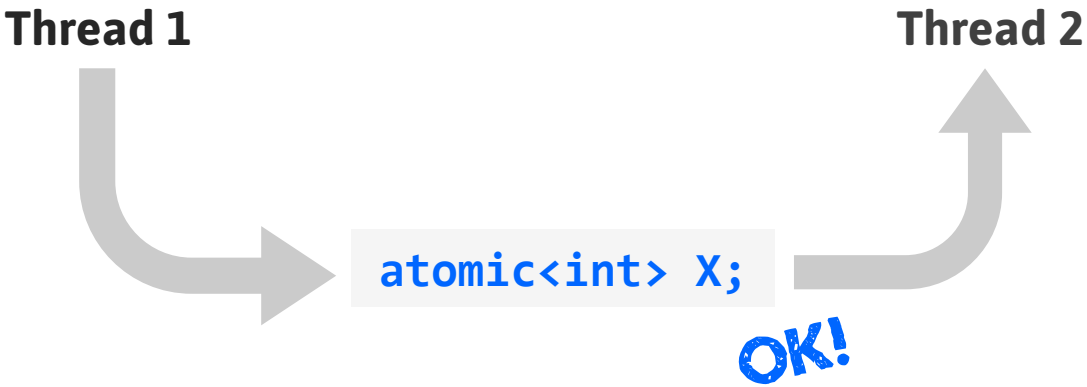
C++11 FORBIDS DATA RACES

If multiple threads access the same variable concurrently, and at least one thread modifies it, all threads must use C++11 atomic operations.



C++11 FORBIDS DATA RACES

If multiple threads access the same variable concurrently, and at least one thread modifies it, all threads must use C++11 atomic operations.



- That's how you know when you must use atomic<>.

C++11 FORBIDS DATA RACES

One reason they're bad

Thread 1

```
x = 0x80004;
```

Thread 2

```
c = x;
```

If machine can only write 16 bits...

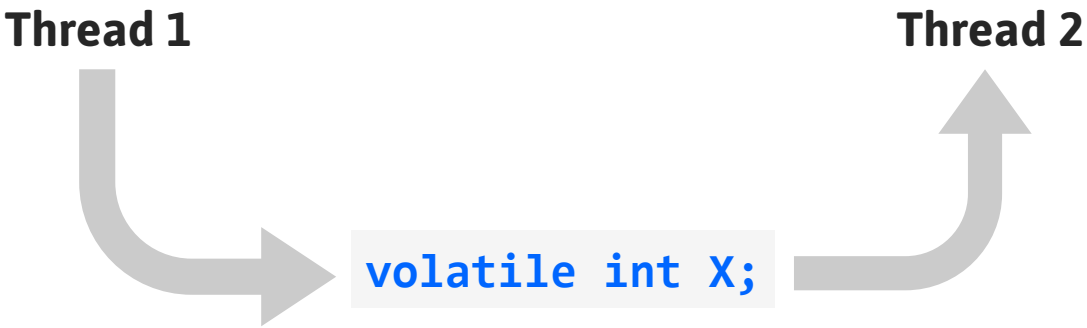
```
int x;
```

...and this is 32-bit...

...we get a "torn write". 0x80000

C++11 FORBIDS DATA RACES

If multiple threads access the same variable concurrently, and at least one thread modifies it, all threads must use C++11 atomic operations.



We break this rule all the time.

We know that `int` is atomic.

IT'S ACTUALLY TWO ATOMIC LIBRARIES

Masquerading under one API

Sequentially Consistent Atomics

- Similar to Java volatiles
- Used in literature/books

*All about **interleaving** statements*

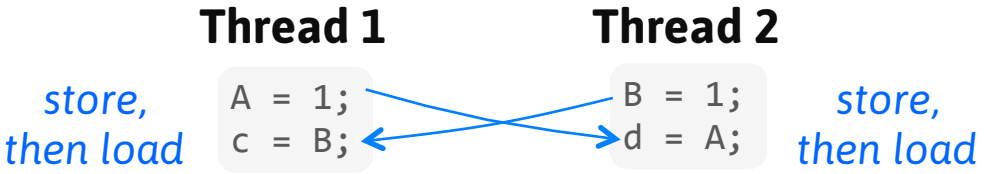
Low-Level Atomics

- Similar to C/C++ volatiles
- Much like game atomics

SEQUENTIALLY CONSISTENT ATOMICS

Example #1

```
atomic<int> A(0);
atomic<int> B(0);
```



Possible Interleavings:

```
A = 1;
c = B;
B = 1;
d = A;
```

```
A = 1;
B = 1;
c = B;
d = A;
```

```
B = 1;
d = A;
A = 1;
c = B;
```

c	d
~~0~~	~~0~~
0	1
1	0
1	1

Impossible!

LOW-LEVEL ATOMICS

Example #1

```
atomic<int> A(0);
atomic<int> B(0);
```

Thread 1

```
A.store(1, memory_order_relaxed);
c = B.load(memory_order_relaxed);
```

Thread 2

```
B.store(1, memory_order_relaxed);
d = A.load(memory_order_relaxed);
```

Doing the same thing

You can prevent it with “full memory fences”:

```
atomic_thread_fence(memory_order_seq_cst);
```

c	d
0	0
0	1
1	0
1	1

Possible!

SEQUENTIALLY CONSISTENT ATOMICS

How to write them

```
atomic<int> A;
```

```
A.store(1, memory_order_seq_cst);
c = A.load(memory_order_seq_cst);
```

All other constraints are low-level

...is the same as:

```
A.store(1);
c = A.load();
```

Default argument

...and the same as:

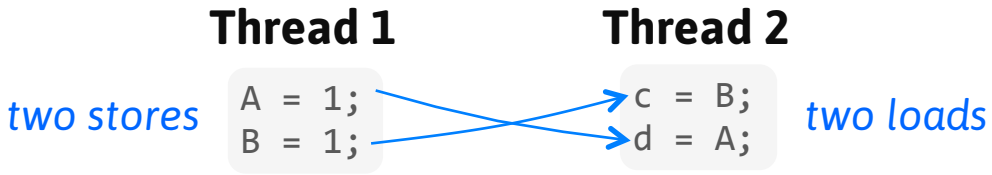
```
A = 1;
c = A;
```

Operator overloading

SEQUENTIALLY CONSISTENT ATOMICS

Example #2

```
atomic<int> A(0);
atomic<int> B(0);
```



Possible Interleavings:

```
A = 1;
B = 1;
c = B;
d = A;
```

```
A = 1;
c = B;
B = 1;
d = A;
```

```
c = B;
d = A;
A = 1;
B = 1;
```

c	d
0	0
0	1
1	0
1	1

Impossible!

LOW-LEVEL ATOMICS

Example #2

```
atomic<int> A(0);
atomic<int> B(0);
```

Thread 1

```
A.store(1, memory_order_relaxed);
B.store(1, memory_order_relaxed);
```

Thread 2

```
c = B.load(memory_order_relaxed);
d = A.load(memory_order_relaxed);
```

Doing the same thing

This is the bug from Section One!

You can fix it with “lightweight fences”:

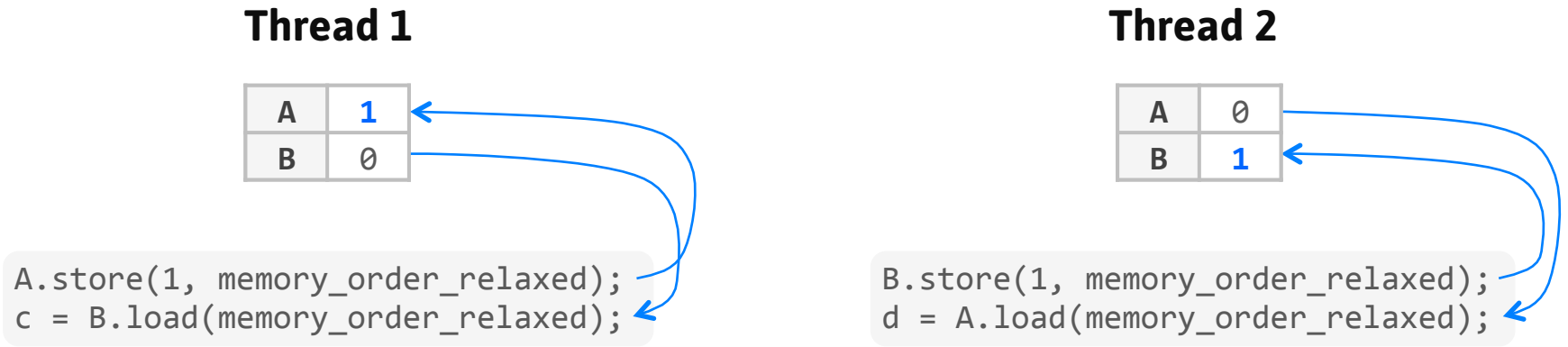
```
atomic_thread_fence(memory_order_acquire);
atomic_thread_fence(memory_order_release);
```

c	d
0	0
0	1
1	0
1	1

Possible!

VISUALIZING LOW-LEVEL ATOMICS

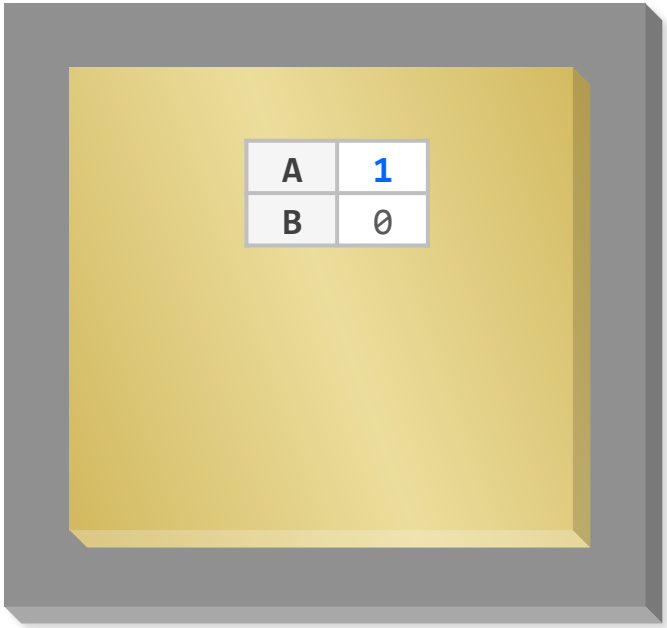
Imagine each thread having its own private copy of memory.

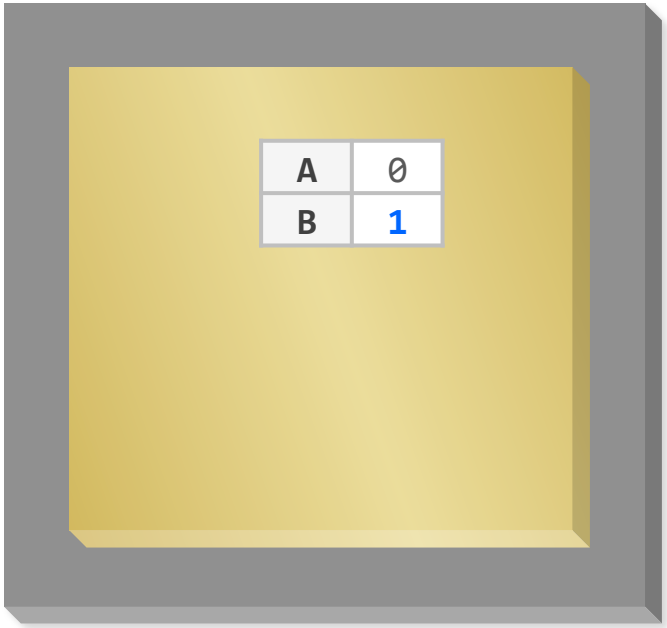


Now c = 0, d = 0 is trivial.

VISUALIZING LOW-LEVEL ATOMICS

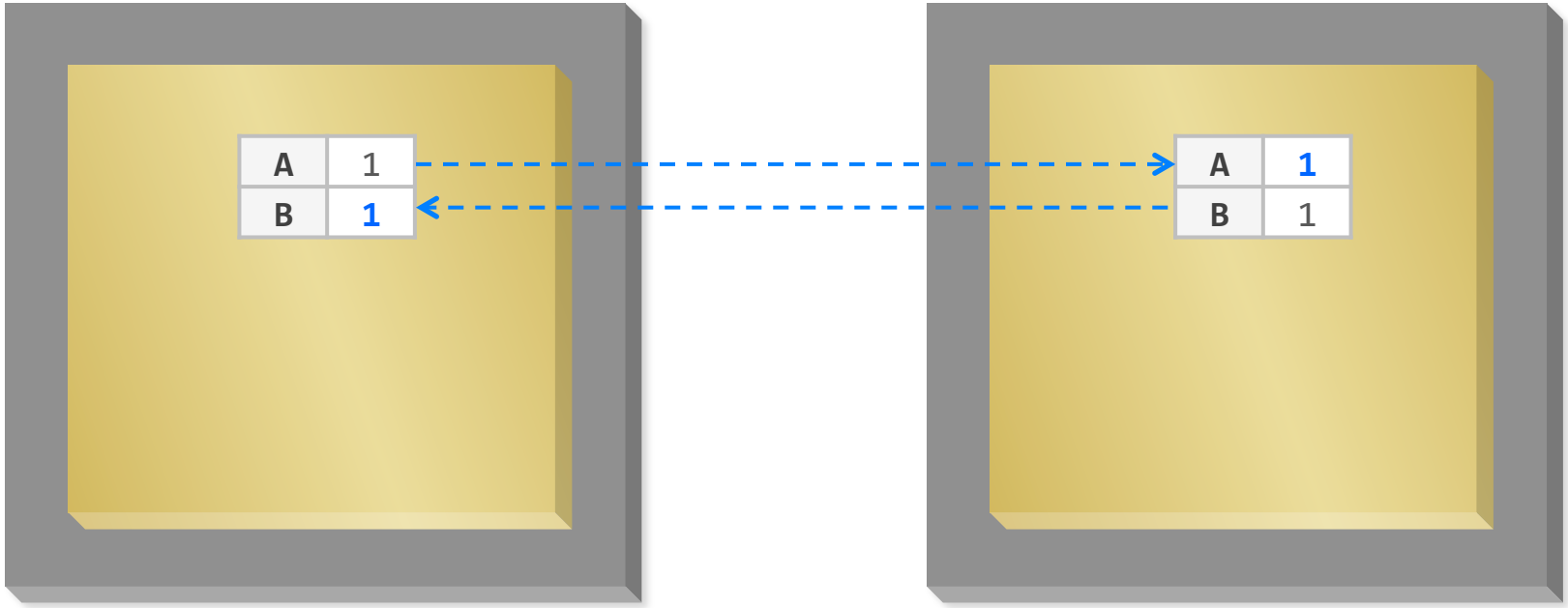
This analogy corresponds to each CPU core having its own cache.





VISUALIZING LOW-LEVEL ATOMICS

Eventually, changes propagate between threads, but the timing is unpredictable.



SEQUENTIALLY CONSISTENT ATOMICS

The magic compilers use to implement them

	Load	Store
x86/64	`mov %, %`	`lock xchg %, %`

PowerPC	`hwsync`
	`ld %, %`
	`cmp %, 0`
	`bc #`
	`isync`
		`hwsync`
		`st %, %`

ARMv7	`ldr %, %`
	`dmb`
		`dmb`
		`str %, %`
		`dmb`

ARMv8	`ldar %, %`	`stlr %, %`
Itanium	`ld.acq %, %`	`st.rel %, %` `mf`

http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

HOW TO CONVERT GAME ATOMICS

to low-level C++11 atomics

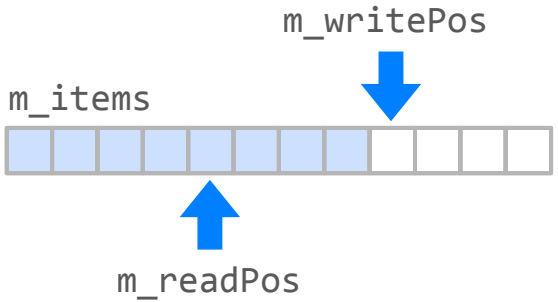
Game Atomics	Low-Level C++11 Atomics
`volatile int A;`	`atomic<int> A;`
`A = 1;` `int a = A;`	`A.store(1, memory_order_relaxed);` `int a = A.load(memory_order_relaxed);`
`LIGHTWEIGHT_FENCE();` `FULL_FENCE();`	`atomic_thread_fence(memory_order_acquire/release);` `atomic_thread_fence(memory_order_seq_cst);`

`AtomicIncrement(A);`
`AtomicCompareExchange(A, …, …);`
`...`
	`A.fetch_add(1, memory_order_relaxed);`
	`A.compare_exchange_strong(…, …, memory_order_relaxed);`
	`...`

EXAMPLE

Capped wait-free queue in C++11

```
template <class T, int size>
class CappedSPSCQueue
{
private:
    T m_items[size];
    atomic<int> m_writePos;
    int m_readPos;

public:
    CappedSPSCQueue() : m_writePos(0), m_readPos(0) {}
    bool tryPush(const T& item) { ... }
    bool tryPop(T& item) { ... }
};
```



EXAMPLE

Low-level with standalone fences

```
bool tryPush(const T& item)
{
    int w = m_writePos.load(memory_order_relaxed);
    if (w >= size)
        return false;
    m_items[w] = item;
    atomic_thread_fence(memory_order_release);
    m_writePos.store(w + 1, memory_order_relaxed);
    return true;
}
```

Could even be non-atomic

```
bool tryPop(T& item)
{
    int w = m_writePos.load(memory_order_relaxed);
    if (m_readPos >= w)
        return false;
    atomic_thread_fence(memory_order_acquire);
    item = m_items[m_readPos];
    m_readPos++;
    return true;
}
```

EXAMPLE

Low-level with standalone fences

```
bool tryPush(const T& item)
{
    int w = m_writePos.load(memory_order_relaxed);
    if (w >= size)
        return false;
    m_items[w] = item;
    atomic_thread_fence(memory_order_release);
    m_writePos.store(w + 1, memory_order_relaxed);
    return true;
}
```

When ***this*** load sees the value written by ***this*** store...

... the fences *synchronize-with* each other (§29.8.2, N3337).

```
bool tryPop(T& item)
{
    int w = m_writePos.load(memory_order_relaxed);
    if (m_readPos >= w)
        return false;
    atomic_thread_fence(memory_order_acquire);
    item = m_items[m_readPos];
    m_readPos++;
    return true;
}
```

EXAMPLE

Low-level ordering constraints

```
bool tryPush(const T& item)
{
    int w = m_writePos.load(memory_order_relaxed);
    if (w >= size)
        return false;
    m_items[w] = item;
    m_writePos.store(w + 1, memory_order_release);
    return true;
}
```

When the load sees the value written by the store...

... the store *synchronizes-with* the load (§29.3.2).

```
bool tryPop(T& item)
{
    int w = m_writePos.load(memory_order_acquire);
    if (m_readPos >= w)
        return false;
    item = m_items[m_readPos];
    m_readPos++;
    return true;
}
```

EXAMPLE

Using sequentially consistent atomics

```
bool tryPush(const T& item)
{
    int w = m_writePos;
    if (w >= size)
        return false;
    m_items[w] = item;
    m_writePos = w + 1;
    return true;
}
```

When the load reads from the store, they *synchronize-with* each other (§29.3.1).

```
bool tryPop(T& item)
{
    int w = m_writePos;
    if (m_readPos >= w)
        return false;
    item = m_items[m_readPos];
    m_readPos++;
    return true;
}
```

EXAMPLE

Capped wait-free queue in C++11

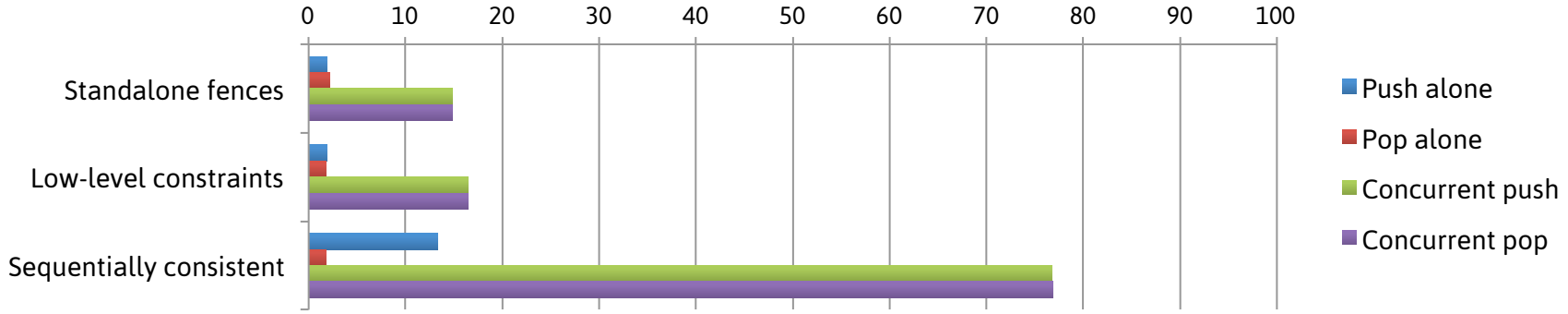
```
template <class T, int size>
class CappedSPSCQueue
{
private:
    T m_items[size];
    atomic<int> m_writePos;
    alignas(64) int m_readPos;

public:
    CappedSPSCQueue() : m_writePos(0), m_readPos(0) {}
    bool tryPush(const T& item) { ... }
    bool tryPop(T& item) { ... }
};
```

All other variables can remain non-atomic because there is no data race.

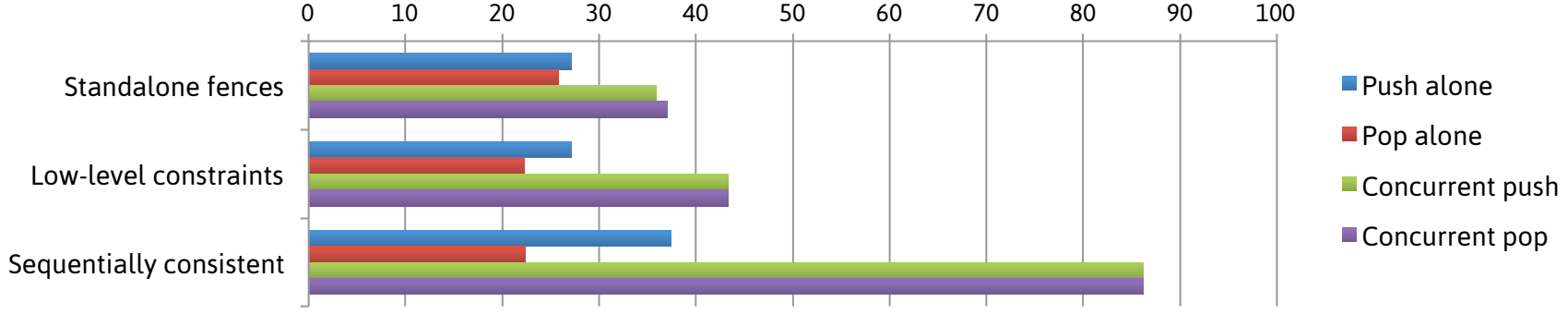
BENCHMARKS

nanoseconds per operation

Intel Core-i7 Quad-core / 2.3 GHz / Xcode 5.1.1 Release



ARM Cortex-A9 Dual-core / 800 MHz / Xcode 5.1.1 Release



RECAP:
The C++11 Atomic Library

- C++11 forbids “data races”
- Two atomic libraries
- Pass non-atomic information by *synchronizing-with*

THANKS

Hugo Allaire
Dominic Couture
Jean-François Dubé
Dominique Duvivier
Michael Lavaire
Jean-Sébastien Pelletier
Rémi Quenin
James Therien

Charles Bloom
Hans Boehm
Bruce Dawson
Peter Dimov
Maurice Herlihy
Paul McKenney
Peter Sewell
Herb Sutter
Anthony Williams
Dmitry Vyukov

Jeff Preshing

 @preshing

 jeff@preshing.com

Preshing on Programming

 http://preshing.com/