Threading: dos && don’ts

Bartek ’BaSz’ Szurgot

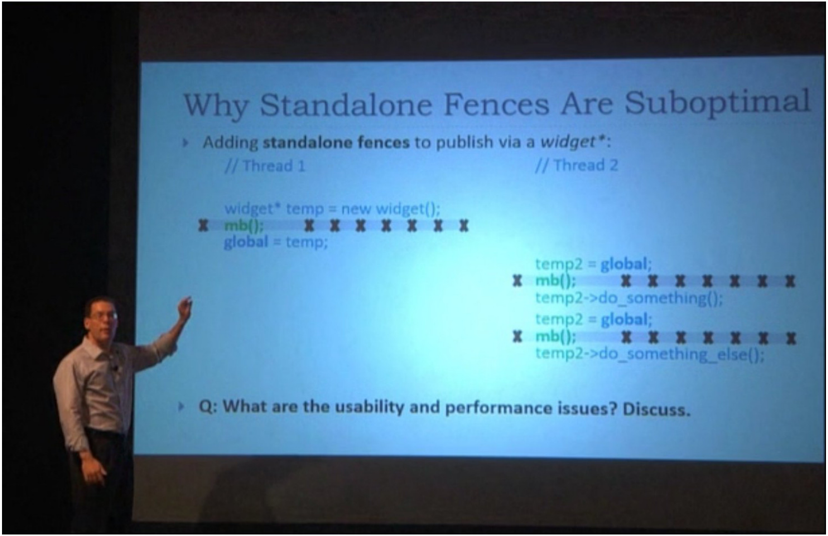
bartek.szurgot@baszerr.eu

2014-11-05


I Only Give
Negative

The problem

Atomic<> Weapons (Herb Sutter)



Threads and Shared Variables in C++11 (Hans Boehm)

Lazy initialization and DCL

- Assume x and initd are initially 0/false.
- Consider:

Thread 1

```
if (!initd) {
  lock_guard<mutex> _(m);
  x = 42;
  initd = true;
}
read x;
```

Thread 2

```
if (!initd) {
  lock_guard<mutex> _(m);
  x = 42;
  initd = true;
}
read x;
```

Data race on initd.

Often works in practice, but not reliable.

Threads and Shared Variables in C++11 (Hans Boehm)

Lazy initialization version 2

```
atomic<bool> initd; // initially false.
int x;
```

Thread 1

```
if (!initd) {
   lock_guard<mutex> _(m);
   x = 42;
   initd = true;
}
read x;
```

Thread 2

```
if (!initd) {
   lock_guard<mutex> _(m);
   x = 42;
   initd = true;
}
read x;
```

No data race.

2 February 2012

Every day coding (BaSz)

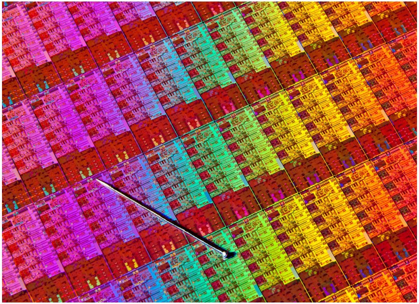
forgetting to lock mutex before accessing shared variable, resulting in non-obvious data-daces; inappropriate use of volatile variables, in pre-cpp11 test code, to synchronize threads; waking up conditional variable for just one thread, when multiple threads could be waiting; not adding assertion to ensure locks are in place, in object implementing lockable pattern; spawning threads for each task, instead of providing proper thread pool do do the work; forgetting to join running thread before program execution ends; implementing own threading proxy library, to cover POSIX API, instead of using already available at that time boost's threads; providing voodoo-like means to exercise stop conditions on a remote thread, sleeping on a queue access, instead of providing null-like element and make this one skipped in a thread's processing loop; arguing that incrementing volatile int is de-facto thread-safe on x86 (yes - this was a long time ago, but unfortunately in this very galaxy...); doing (mostly implicit) locking in interruption handlers; spawning new threads for each incoming client connection on simple instant messaging server; using POSIX threads in C++ without proper RAII-based wrappers; volatiles did appeared in my threaded test code for some period of time; doing mutex locking on counters, that could be instantiated on a per-thread basis, instead of making them local and just return final value at the end, or optionally separate atomics with release semantics, and accumulate logic in thread coordinator loop; performing long-lasting input-output operations while having a lock on a resource; using the same promise object from multiple threads, instead of moving its ownership to a final destination and not getting bothered about data races between set_value and promise's destructor; being happy that x86 has a pretty strong memory model (now can't wait ARMv8 with sequentially-consistent one!); forgetting to add a try-catch on the whole thread's body, to ensure (mostly) clean shutdown instead of nasty terminate/abort or even compiler-defined aborts (pre-cpp11 here); locking mutex for too long; checking if non-recursive mutex is locked by calling try_lock from the same thread, in assert;

And so. . .

- Concurrent programming
  - Hard ("*Small fixes to prevent blue-screens*")
  - Attention-demanding ("*Free lunch is over*")

And so. . .

- Concurrent programming
  - Hard ("*Small fixes to prevent blue-screens*")
  - Attention-demanding ("*Free lunch is over*")
- Concurrency and modern hardware



- Not that obvious
- How not to kill performance

PRNG

Sequential program

```
int count = 4*1000;
int sum   = 0;
for(int i=0; i<count; ++i)
  sum += simulateRandomEnv(); // heavy stuff...
cout << "average␣result:␣" << sum / count << endl;
```

Sequential program

```
int count = 4*1000;
int sum   = 0;
for(int i=0; i<count; ++i)
  sum += simulateRandomEnv(); // heavy stuff...
cout << "average␣result:␣" << sum / count << endl;
```

- How to speed it up?
- Make it parallel!

Sequential program

```
int count = 4*1000;
int sum   = 0;
for(int i=0; i<count; ++i)
  sum += simulateRandomEnv(); // heavy stuff...
cout << "average result: " << sum / count << endl;
```

- How to speed it up?
- Make it parallel!
- Each iteration:
  - Takes the same time
  - Is independent
- Perfect parallel problem!

Parallel program

- 4 cores – 4 threads
- $1/4$ iterations each
- 4x speedup!

Parallel program

- 4 cores – 4 threads
- 1/4 iterations each
- 4x speedup!

```
int count = 1*1000;
int sum   = 0;
for(int i=0; i<count; ++i)
  sum += simulateRandomEnv(); // heavy stuff...
// return sum from the thread
```

Results

- Timing:
  - Parallel way slower. . .
  - More cores == slower execution

Results

- Timing:
  - Parallel way slower. . .
  - More cores == slower execution
- Profiling:
  - Low CPUs load
  - Mostly waiting on a single mutex

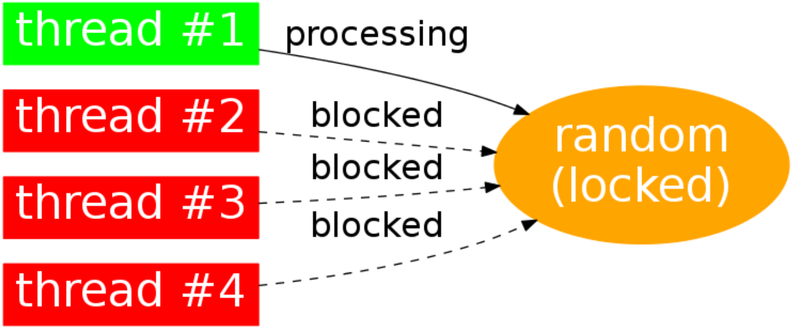
Results

- Timing:
  - Parallel way slower. . .
  - More cores == slower execution
- Profiling:
  - Low CPUs load
  - Mostly waiting on a single mutex
- Logic:
  - Come again?
  - What MUTEX?!

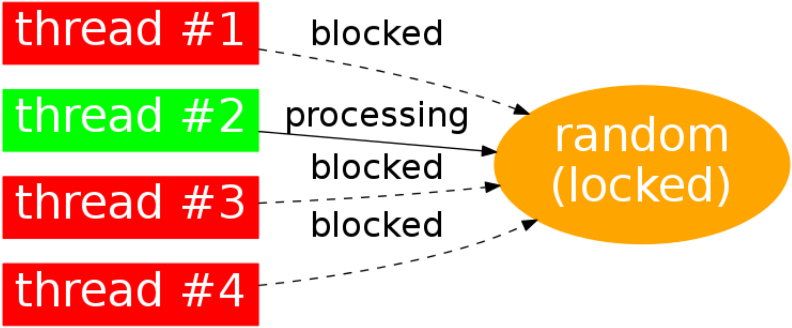
Results

- Timing:
  - Parallel way slower. . .
  - More cores == slower execution
- Profiling:
  - Low CPUs load
  - Mostly waiting on a single mutex
- Logic:
  - Come again?
  - What MUTEX?!
- Suspect:
  - *simulate**Random**Env()*
  - *random()*
  - POSIX: *random() is thread-safe*. . .
  - . . . via mutex

What is wrong?



What is wrong?



Fix

- Drop problematic *random()*
- Add per-thread PRNG

Fix

- Drop problematic *random()*
- Add per-thread PRNG

```
int simulateRandomEnv()
{
  using Distribution = uniform_int_distribution<long>;
  random_device rd;
  mt19937       gen{rd()};
  Distribution  dist{0, RAND_MAX};
  auto          random = [&]{ return dist(gen); };
  int           result = 0;
  //
  // rest of the code remains the same!
  //
  return result;
}
```

Lessons learned

- Measure:
  - Do it always
  - Also "the obvious"
  - Especially when "you are sure"
  - No excuse for not doing so

Lessons learned

- Measure:
  - Do it always
  - Also "the obvious"
  - Especially when "you are sure"
  - No excuse for not doing so
- Avoid shared state:
  - Requires synchronization
  - Locking means blocking
  - Often kills performance

Lessons learned

- Measure:
  - Do it always
  - Also "the obvious"
  - Especially when "you are sure"
  - No excuse for not doing so
- Avoid shared state:
  - Requires synchronization
  - Locking means blocking
  - Often kills performance
- Use state-of-art tools:
  - More powerful
  - Known issues addressed

(Not)shared

Source code

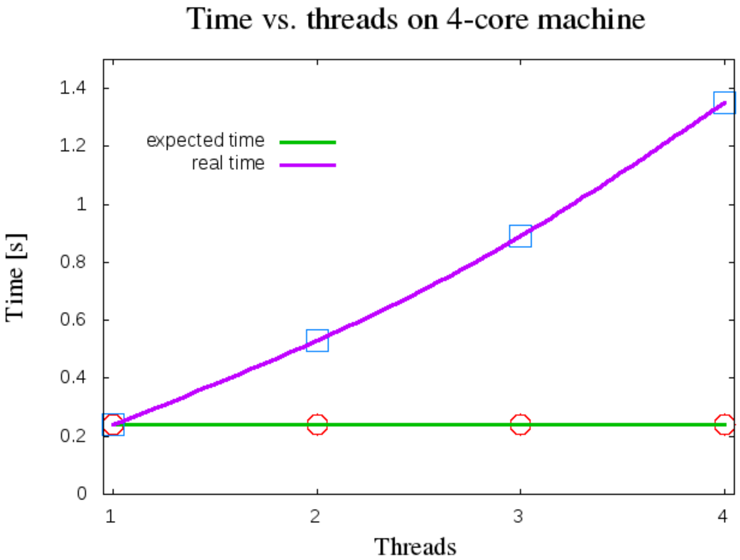
```
unsigned a = 0;
unsigned b = 0;

void threadOne()
{
  for(int i=0; i<10*1000*1000; ++i)
    a += computeSth(i);
}

void threadTwo()
{
  for(int i=0; i<10*1000*1000; ++i)
    b += computeSthElse(i);
}
```

Source code

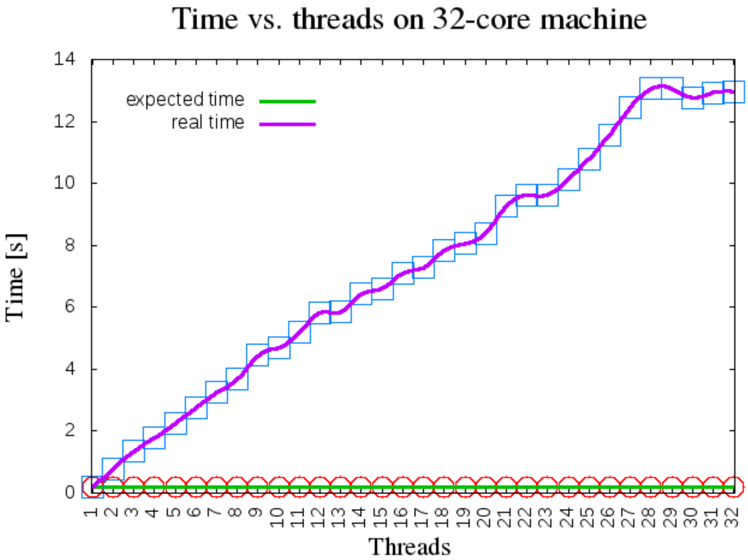
```
unsigned a = 0;
unsigned b = 0;

void threadOne()
{
  for(int i=0; i<10*1000*1000; ++i)
    a += computeSth(i);
}

void threadTwo()
{
  for(int i=0; i<10*1000*1000; ++i)
    b += computeSthElse(i);
}
```

- Data-race free
- Returns expected results

Measurement results: 4-core



Measurement results: 32-core



Variables in memory

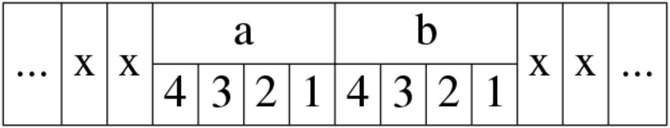
```
unsigned a; // used by thread #1
unsigned b; // used by thread #2
```

- Remember Scott’s presentation?

Variables in memory

```
unsigned a; // used by thread #1
unsigned b; // used by thread #2
```

- Remember Scott’s presentation?
- False sharing is back!



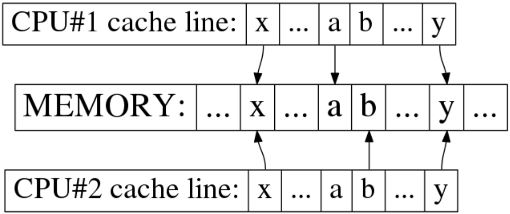
- Assume 32-bit
- Most likely:
  - Consecutive addresses
  - Same cache line

Line-wise

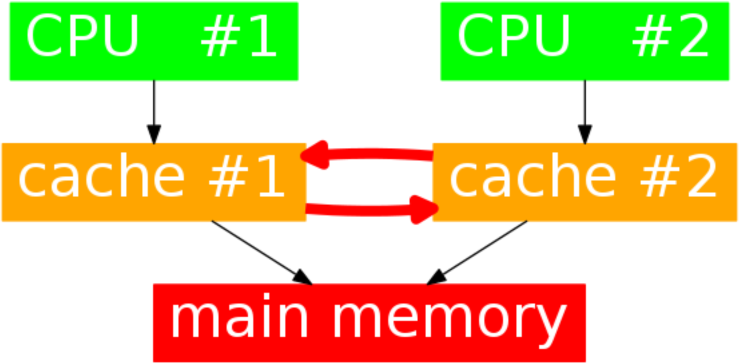
- Caches are not byte-wise
- Operate on lines
- Tens-hundreds of bytes
- Eg. $64B$ in my case
- Operate on aligned addresses

Line-wise

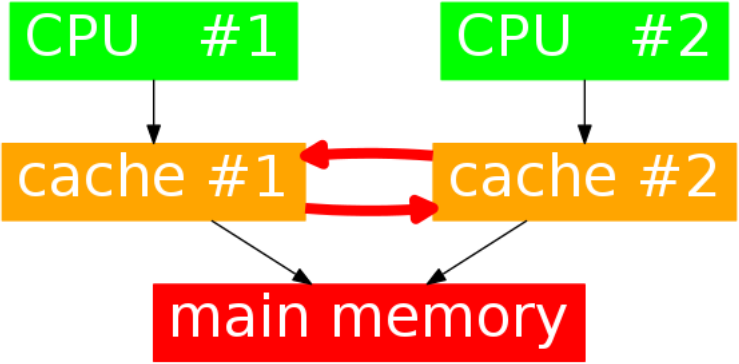
- Caches are not byte-wise
- Operate on lines
- Tens-hundreds of bytes
- Eg. 64*B* in my case
- Operate on aligned addresses



HOT-line



HOT-line



- What can we do?

Solution #1

- Sun Tzu: Art of war. . .
- . . . avoid situations like this! :)

Not that simple

- Almost sure:
  - Arrays
  - Globals in a single file

Not that simple

- Almost sure:
  - Arrays
  - Globals in a single file
- Probable:
  - Globals from different compilation units
  - Dynamically allocated memory

Not that simple

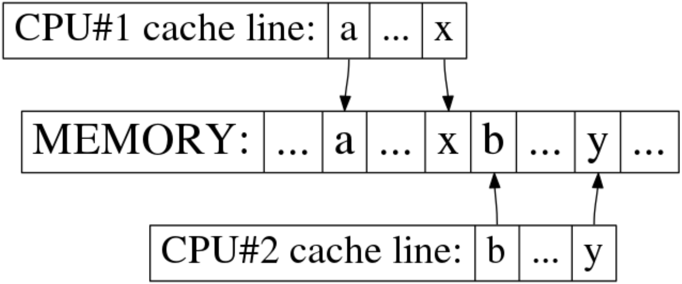
- Almost sure:
  - Arrays
  - Globals in a single file
- Probable:
  - Globals from different compilation units
  - Dynamically allocated memory
- Risky business...

Solution #2

- Ensure won't happen
- One variable – one cache line:
  - Alignment
  - Padding

Solution #2

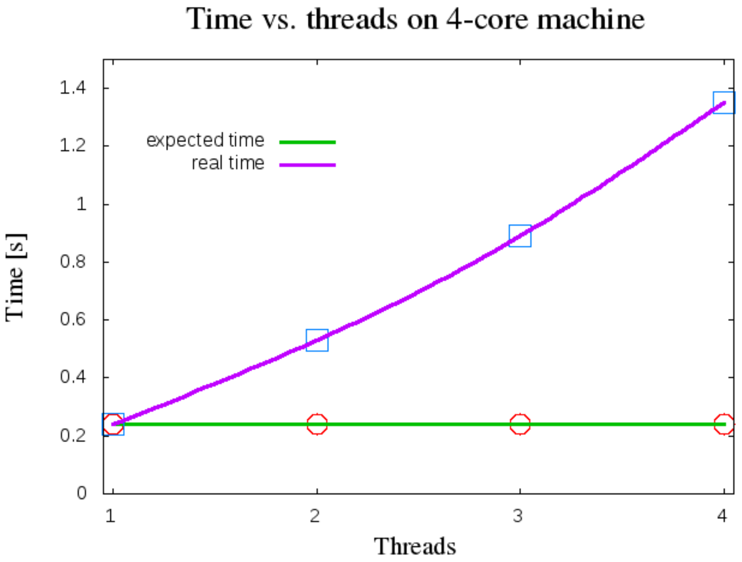
- Ensure won't happen
- One variable – one cache line:
  - Alignment
  - Padding



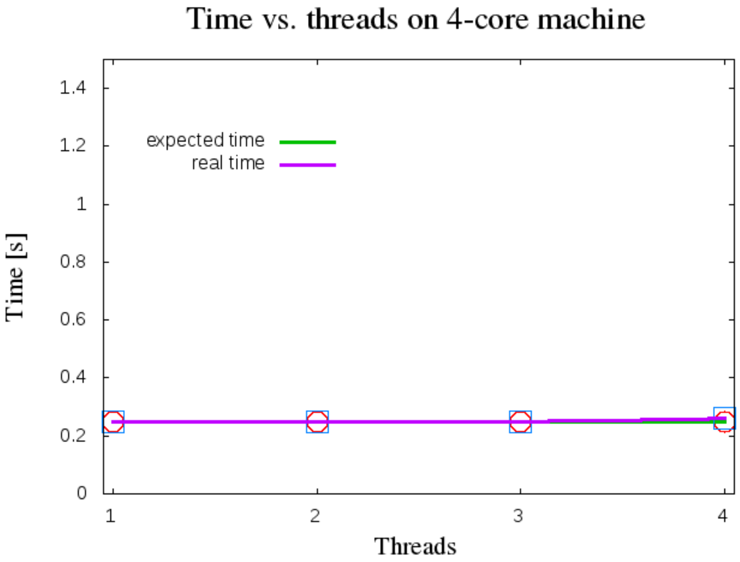
Helper template

```
template<typename T, unsigned Align=64>
struct alignas(Align) CacheLine
{
  static_assert( std::is_pod<T>::value,
      "cannot guarantee layout for non-PODs" );
  T data_;
  // NOTE: auto-padded due to alignment!
};
```

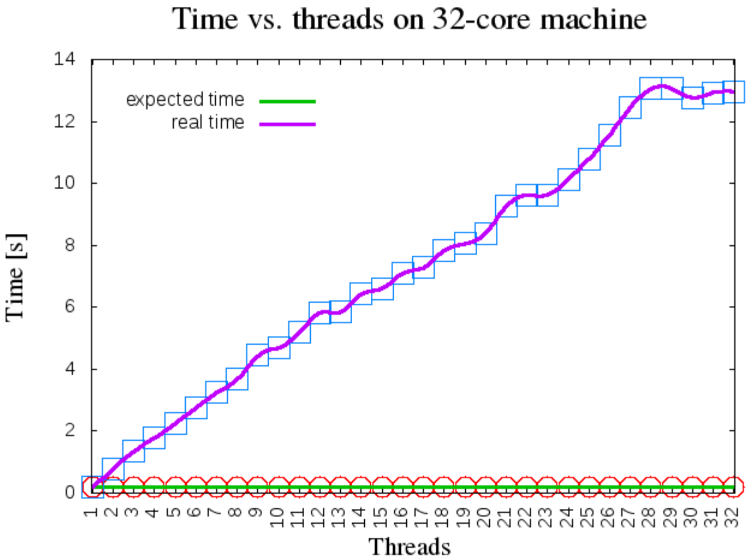
Before the fix: 4-core



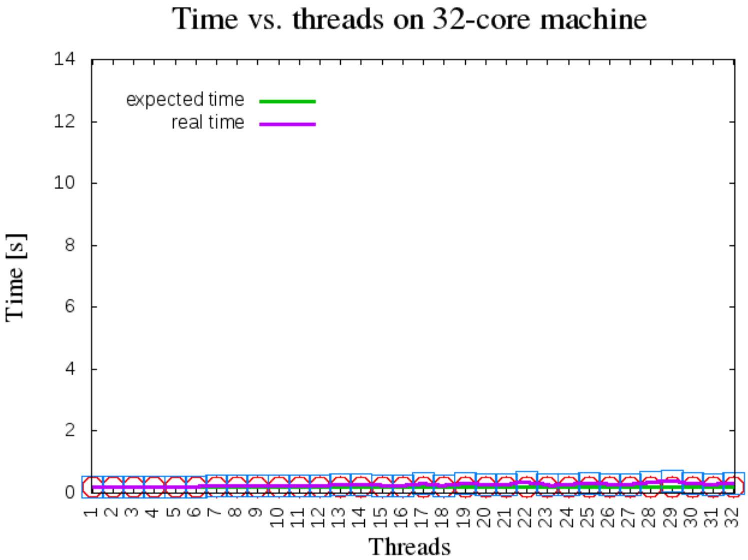
Measuring fix: 4-core



Before the fix: 32-core



Measuring fix: 32-core



Lessons learned

- Measure:
  - Do it always
  - Also "the obvious"
  - Especially when "you are sure"
  - No excuse for not doing so

Lessons learned

- Measure:
  - Do it always
  - Also "the obvious"
  - Especially when "you are sure"
  - No excuse for not doing so
- Think about caches:
  - Great speed improvement
  - Worst-case scenario – cache miss
  - Not fully transparent
  - In-CPU cache coherency protocols
  - More CPUs == bigger impact

Atomics

"I know my hardware!" case

```
int g_counter = 0;

void threadCall() // called from multiple threads
{
  for(int i=0; i<1*1000*1000; ++i)
    ++g_counter;
}
```

- Good. . . ?
- Bad. . . ?
- Ugly. . . ?

"I know my hardware!" case

```
int g_counter = 0;

void threadCall() // called from multiple threads
{
  for(int i=0; i<1*1000*1000; ++i)
    ++g_counter;
}
```

- Good...?
- Bad...?
- Ugly...?

- Results for 4 threads:
  - 2000000 – 232/10000
  - 4000000 – 526/10000
  - 3000000 – 9242/10000
- Right 5% of times
- Smells like a bug!

"I know my compiler!" case

```
volatile int g_counter = 0; // fix

void threadCall() // called from multiple threads
{
  for(int i=0; i<1*1000*1000; ++i)
    ++g_counter;
}
```

- Good...?
- Bad...?
- Ugly...?

"I know my compiler!" case

```
volatile int g_counter = 0; // fix

void threadCall() // called from multiple threads
{
  for(int i=0; i<1*1000*1000; ++i)
    ++g_counter;
}
```

- Good. . . ?
- Bad. . . ?
- Ugly. . . ?

- Some results on 4 threads:
  - 1000002 – 4/1000
  - 1000060 – 8/1000
  - 1000000 – 69/1000
- Right 0 (% of) times
- Closest: 1871882/4000000

The Only Way(tm)

```
std::atomic<int> g_counter(0);

void threadCall() // called from multiple threads
{
  for(int i=0; i<1*1000*1000; ++i)
    ++g_counter;
}
```

- Valid C++11

The Only Way(tm)

```
std::atomic<int> g_counter(0);

void threadCall() // called from multiple threads
{
  for(int i=0; i<1*1000*1000; ++i)
    ++g_counter;
}
```

- Valid C++11
- Using 4 threads
- Result: 4000000
- Always – lol!!!11

Think before you code

```
std::atomic<int> g_counter(0);  // shared

void threadCall() // called from multiple threads
{
  int counter = 0;              // local
  for(int i=0; i<1*1000*1000; ++i)
    ++counter;
  g_counter += counter;         // single write
}
```

- Can be an option?
- WAY faster

Volatile rescue mission

- Volatile and lost writes
- Missed optimizations

- Single-instruction summing failed
- How about signaling?

Volatile flags maybe?

```
volatile bool started1 = false;
volatile bool started2 = false;

void thread1()
{
  started1 = true;
  if(not started2)
    std::cout << "thread␣1␣was␣first\n";
}

void thread2()
{
  started2 = true;
  if(not started1)
    std::cout << "thread␣2␣was␣first\n";
}
```

Results

- Most of the time – fine
- $6/10000$ times:

Results

- Most of the time – fine
- 6/10000 times:

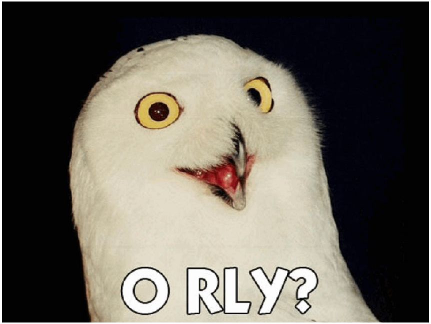
```
thread 1 was first
thread 2 was first
```

Results

- Most of the time – fine
- 6/10000 times:

```
thread 1 was first
thread 2 was first
```

- 0.06% error rate
- What really happened?



Zoom in – original code

```
volatile bool started1 = false;
volatile bool started2 = false;

void thread1()
{
  started1 = true;  // memory write
  if(not started2)  // memory read
  { /* some action */ }
}
```

Zoom in – reordered

```
volatile bool started1 = false;
volatile bool started2 = false;

void thread1()
{
  if(not started2)  // memory read
  { /* some action */ }
  started1 = true;  // memory write (!)
}
```

Sequential consistency

- Reordering by:
  - Compiler
  - Hardware
- Do they break programs?

Sequential consistency

- Reordering by:
  - Compiler
  - Hardware
- Do they break programs?
- Restrictions:
  - No observables effects
  - As-if single-threaded

Sequential consistency

- Reordering by:
  - Compiler
  - Hardware
- Do they break programs?
- Restrictions:
  - No observables effects
  - As-if single-threaded
- Explicit marking shared elements:
  - Atomics
  - Mutex-protected sections

Sequential consistency

- Reordering by:
  - Compiler
  - Hardware
- Do they break programs?
- Restrictions:
  - No observables effects
  - As-if single-threaded
- Explicit marking shared elements:
  - Atomics
  - Mutex-protected sections
- Data-race free (DRF) code is a must!
- SC for DRF

Lessons learned

- Never use volatiles for threading
- I mean it!
- Synchronize using:
  - Atomics
  - Mutexes
  - Conditionals

Lessons learned

- Never use volatiles for threading
- I mean it!
- Synchronize using:
  - Atomics
  - Mutexes
  - Conditionals
- Mind sequential consistency (SC)
  - Write data-race free (DRF) code
  - Weird problems if you don't
  - Reproducibility issues otherwise

Lessons learned

- Never use volatiles for threading
- I mean it!
- Synchronize using:
  - Atomics
  - Mutexes
  - Conditionals
- Mind sequential consistency (SC)
  - Write data-race free (DRF) code
  - Weird problems if you don't
  - Reproducibility issues otherwise
- Mind the efficiency:
  - Prefer local over shared
  - Synchronize when must

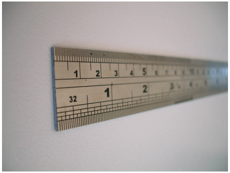
Lessons learned

- Never use volatiles for threading
- I mean it!
- Synchronize using:
  - Atomics
  - Mutexes
  - Conditionals
- Mind sequential consistency (SC)
  - Write data-race free (DRF) code
  - Weird problems if you don't
  - Reproducibility issues otherwise
- Mind the efficiency:
  - Prefer local over shared
  - Synchronize when must
- Experiment and verify
- Do the code review

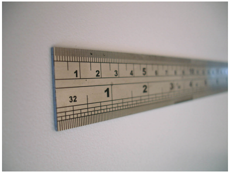
Summary

Dos && don'ts

- Rule No.1:
  - measure
  - MEASURE
  - M-E-A-S-U-R-E

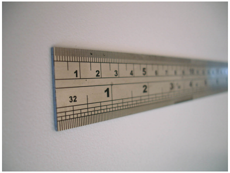


Dos && don'ts



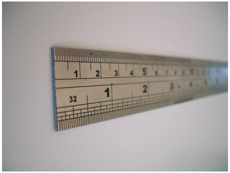
- Rule No.1:
  - measure
  - MEASURE
  - M-E-A-S-U-R-E
- Be cache-aware:
  - Keep non-shared data in separate cache lines
  - Prefer local over shared

Dos && don'ts



- Rule No.1:
  - measure
  - MEASURE
  - M-E-A-S-U-R-E
- Be cache-aware:
  - Keep non-shared data in separate cache lines
  - Prefer local over shared
- Synchronize properly:
  - Ensure code is data-race free (DRF)
  - NEVER use volatiles for sharing
  - Benefit from sequential consistency (SC)

Dos && don’ts



- Rule No.1:
  - measure
  - MEASURE
  - M-E-A-S-U-R-E
- Be cache-aware:
  - Keep non-shared data in separate cache lines
  - Prefer local over shared
- Synchronize properly:
  - Ensure code is data-race free (DRF)
  - NEVER use volatiles for sharing
  - Benefit from sequential consistency (SC)
- Homework:
  - Read C++11 memory model
  - Read *multi-threaded executions and data races*
  - x86* vs. ARM and IA-64

More materials

- Something to watch:
  - *Threads and shared variables in C++11*,
    Hans Boehm,
    available on Channel9
  - *Atomic<> Weapons*,
    Herb Sutter,
    available on Channel9
  - *CPU Caches and Why You care*,
    Scott Meyers,
    available on Code::Dive :)

More materials

- Something to watch:
  - *Threads and shared variables in C++11*,
    Hans Boehm,
    available on Channel9
  - *Atomic<> Weapons*,
    Herb Sutter,
    available on Channel9
  - *CPU Caches and Why You care*,
    Scott Meyers,
    available on Code::Dive :)
- Something to read:
  - *C++ Concurrency in action: practical multithreading*,
    Anthony Williams

More materials

- Something to watch:
  - *Threads and shared variables in C++11*,
    Hans Boehm,
    available on Channel9
  - *Atomic<> Weapons*,
    Herb Sutter,
    available on Channel9
  - *CPU Caches and Why You care*,
    Scott Meyers,
    available on Code::Dive :)
- Something to read:
  - *C++ Concurrency in action: practical multithreading*,
    Anthony Williams
  - *The Hitchhiker's Guide to the Galaxy*,
    Douglas Adams

Questions?

?