Consistency models from a programmer's perspective

...

Presented by Alex Mononen and Jack Yu

Threads Cannot be Implemented as a Library

...

Hans-J. Boehm. 2005. Threads cannot be implemented as a library. *SIGPLAN Not.* 40, 6 (June 2005), 261–268. DOI:https://doi.org/10.1145/1064978.1065042

Pthread Concurrency Implementation

Pthread Concurrency Implementation

What causes concurrency issues?

- Hardware may reorder memory operations
- Compilers may reorder memory operations

"Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads"

Source: "Memory Synchronization" in *IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992)* , pp. 100, 6 Dec. 2001, doi: 10.1109/IEEESTD.2001.93364.

Pthread Concurrency Implementation

Problems with this specification

- Ambiguous - what happens when rules are violated? How big should a memory location be?
- Informal - doesn't say exactly what order memory operations take place
- Makes ensuring concurrency the responsibility of the application developer instead of language designer - more work for the application developer

Pthread Concurrency Implementation

Supported Synchronization Functions

- `fork()`
- `pthread_barrier_wait()`
- `pthread_cond_broadcast()`
- `pthread_cond_signal()`
- `pthread_cond_timedwait()`
- `pthread_cond_wait()`
- `pthread_create()`
- `pthread_join()`
- `pthread_mutex_lock()`
- `pthread_mutex_timedlock()`
- `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`
- `pthread_spin_lock()`
- `pthread_spin_trylock()`
- `pthread_spin_unlock()`
- `pthread_rwlock_rdlock()`
- `pthread_rwlock_timedrdlock()`
- `pthread_rwlock_timedwrlock()`
- `pthread_rwlock_tryrdlock()`
- `pthread_rwlock_trywrlock()`
- `pthread_rwlock_wrlock()`
- `pthread_rwlock_unlock()`
- `sem_post()`
- `sem_timedwait()`
- `sem_trywait()`
- `sem_wait()`
- `semctl()`
- `semop()`
- `wait()`
- `waitpid()`

Pthread Concurrency Implementation

The Pthreads solution to concurrency issues

- Memory barrier instructions in synchronization functions prevent reordering of memory operations out of the critical section
- Pthread functions are treated as opaque functions where anything is possible (i.e. read/write global value), so that memory operations cannot be moved around the call

Pthread Concurrency Implementation

The Pthreads solution to concurrency issues

- Memory barrier instructions in synchronization functions prevent reordering of memory operations out of the critical section
- Pthread functions are treated as opaque functions where anything is possible (i.e. read/write global value), so that memory operations cannot be moved around the call

Opaque function example

main.c:

```
#include “lib.h”

...

void f() {

  x = 0;

  b();

  printf(“%d\n”, x);

}

...
```

lib.h:

```
int x;

...

void b();

...
```

lib.c (hidden):

```
#include “lib.h”

...

void b() {

  x++;

}

...
```

Opaque function example

main.c:

```
#include “lib.h”

...

void f() {

  x = 0;

  b();

  printf(“%d\n”, x);

}

...
```

lib.h:

```
int x;

...

void b();

...
```

lib.c (hidden):

```
#include “lib.h”

...

void b() {

  x++;

}

...
```

Opaque function example

main.c:

```
#include “lib.h”

...

void f() {

  x = 0;

  b();

  printf(“%d\n”, x);

}

...
```

lib.h:

```
int x;

...

void b();

...
```

lib.c (hidden):

```
#include “lib.h”

...

void b() {

  x++;

}

...
```

Pthread Concurrency Implementation

Why are these “workarounds” needed?

- The C/C++ compiler does not see threads!
- “Just another runtime library”

Pthread Concurrency Implementation

Problems

- Compiler may introduce race conditions that the programmer does not expect
  - When is a race expected to occur? When does it ACTUALLY occur?
  - No formal specification
- Compiler cannot optimize program in accordance to best practice due to lack of information
  - Either accept poor performance, or intentionally/unintentionally break the rules
    - Programs are now more likely to have memory errors.

Correctness Issues

Correctness Issues

- Concurrent Modification (Speculation leading to invalid states)
- Rewriting of Adjacent Data (Bit-fields overwriting data in the same word)
- Register Promotion (Optimizing variables out of critical section)

Correctness Issues

Concurrent Modification

—

Optimization: Speculation

- Branch instructions can be expensive if it is the common case
- May be faster to speculate then “undo” the assignment for the rare cases

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

“Optimized”:

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

“Optimized”:

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y):

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

"Optimized":

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y):

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

“Optimized”:

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y):

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

“Optimized”:

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y): (0, 0)

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

"Optimized":

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y): (0, 0)

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

"Optimized":

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y): (0, 0)

Correctness Issues

Concurrent Modification

Original:

```
int x = y = 0;
T1:
    if (x == 1) ++y;
T2:
    if (y == 1) ++x;
```

Possible states of (x, y): (0, 0)

"Optimized":

```
int x = y = 0;
T1:
    ++y; if (x != 1) --y;
T2:
    ++x; if (y != 1) --x;
```

Possible states of (x, y): (0, 0), (1, 1)

Correctness Issues

Concurrent Modification

Optimization: speculation

- Branch instructions can be expensive if it is the common case
- May be faster to speculate then “undo” the assignment for the rare cases

Issue:

- Optimization results in new possible state in program
- Author mentions no known real-life bugs caused by this, but still violation of pthreads specification

Correctness Issues

Rewriting of Adjacent Data

—

Optimization: consolidating bit-fields

- Compiler combines variables into a single word for space efficiency
- Memory location of variables not transparent to application programmer

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};
```

What does the struct look like in memory?

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};
```

```
T1:
    x.b = 'b';
    x.c = 'c';
    x.d = 'd';

T2:
    x.a = 'a';
```

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};
```

```
T1:
    x = “dcb\0” | x.a;
```

```
T2:
    x.a = ‘a’;
```

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};

x.a: '\0'
x.b: '\0'
x.c: '\0'
x.d: '\0'
```

```
T1:
    x = "dcb\0" | x.a;

T2:
    x.a = 'a';
```

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};
```

```
x.a: 'a'
x.b: '\0'
x.c: '\0'
x.d: '\0'
```

```
T1:
    x = "dcb\0" | x.a;
```

```
T2:
    x.a = 'a';
```

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};

x.a: 'a'
x.b: '\0'
x.c: '\0'
x.d: '\0'
```

```
T1:
    x = "dcb\0" | x.a;

T2:
    x.a = 'a';
```

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};

x.a: 'a'
x.b: '\0'
x.c: '\0'
x.d: '\0'
```

```
T1:
    x = "dcb\0" | '\0';

T2:
    x.a = 'a';
```

Correctness Issues

Rewriting of Adjacent Data

```
struct x {
  char a,b,c,d;
};
```

```
x.a: '\0'
x.b: 'b'
x.c: 'c'
x.d: 'd'
```

```
T1:
    x = "dcb\0" | x.a;
```

```
T2:
    x.a = 'a';
```

Correctness Issues

Rewriting of Adjacent Data

—

Optimization: consolidating bit-fields

- Compiler combines variables into a single word for space efficiency
- Memory location of variables not transparent to application programmer

Issue:

- Threads modifying different variables update same memory location
- Modifications are not atomic, race condition occurs

Correctness Issues

Register Promotion

Optimization: register promotion of variables

- Variable stored in register instead of memory to improve performance
- Store to/read from memory around opaque functions for correct behavior

Correctness Issues

Register Promotion

- x is a global variable
- b(), c() are normal functions

```
int x;

void f() {

  b();

  x++;

  c();

}
```

Correctness Issues

Register Promotion

- x is a global variable
- b(), c() are normal functions

```
int x;

void f() {

  b();

  x++;

  c();

}
```

Correctness Issues

Register Promotion

- ~~x is a global variable~~
- y is a register variable
- b(), c() are normal functions

```
~~int x;~~

register int y

void f() {

  b();

  y++;

  c();

}
```

Correctness Issues

Register Promotion

- ~~x is a global variable~~
- y is a register variable
- b(), c() are opaque functions

```
~~int x;~~

register int y

void f() {

  b(); //opaque

  y++;

  c(); //opaque

}
```

Correctness Issues

Register Promotion

- ~~x is a global variable~~
- y is a register variable
- b(), c() are opaque functions

```
~~int x;~~

register int y

void f() {

  b(); //opaque

  y++;

  c(); //opaque

}
```

Correctness Issues

Register Promotion

- x is a global variable
- y is a register variable
- b(), c() are opaque functions

```
int x;

register int y

void f() {

  x = y;

  b(); //opaque

  y = x;

  y++;

  x = y;

  c(); //opaque

  y = x;

}
```

Correctness Issues

Register Promotion

- x is a global variable
- y is a register variable
- orange section is a critical section

```
int x;

register int y

void f() {

  x = y;

  pthread_mutex_lock(&mtx);

  y = x;

  y++;

  x = y;

  pthread_mutex_unlock(&mtx);

  y = x;

}
```

Correctness Issues

Register Promotion

- x is a global variable
- y is a register variable
- orange section is a critical section
- modify x outside of critical section!

```
int x;

register int y

void f() {

  x = y;

  pthread_mutex_lock(&mtx);

  y = x;

  y++;

  x = y;

  pthread_mutex_unlock(&mtx);

  y = x;

}
```

Correctness Issues

Register Promotion

Optimization: register promotion of variables

- Variable stored in register instead of memory to improve performance
- Store to/read from memory around opaque functions for correct behavior

Issue:

- Variable is originally shared across multiple threads protected by a lock
- Memory location of shared variable now modified outside of critical section

Performance Issues

Performance Issues

- Only parallel programming style sanctioned by pthreads: pthread synchronization primitives
  - No volatile, no atomic C++ data types
  - `pthread_mutex_lock()/pthread_mutex_unlock()` implemented using atomic instructions and memory barriers
  - Better performance using atomic instructions directly (C++ atomic types)
  - Prevents hardware instruction reordering, in some cases, in both directions when only one direction matters
- What if data races and specification violation do not affect program output?

Example: Sieve of Eratosthenes

```
for (my_prime = start;
     my_prime < 10000; ++my_prime)
  if (!get(my_prime)) {
    for (multiple = my_prime;
         multiple < 100000000;
         multiple += my_prime)
    if (!get(multiple)) set(multiple);
  }
```

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

→	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Performance Issue: Sieve of Eratosthenes

Performance Issues

Multiple Sieve of Eratosthenes on same array

- Multiple threads mark and sweep what is not a prime
- Check unmarked numbers for primes
- What if a mark doesn't show up on a different thread?
  - Redo the work, produce the same results
- Safe to have data races!
  - Doesn't change output

Performance Issues

Multithreaded Sieve of Eratosthenes Synchronization

—

- Mutex
- Spinlock
- Atomics - Provided by C++, not pthreads
- None - Should produce correct results

Only Mutex and Spinlock methods are sanctioned by pthreads!

Performance Issues

Multithreaded Sieve of Eratosthenes Performance

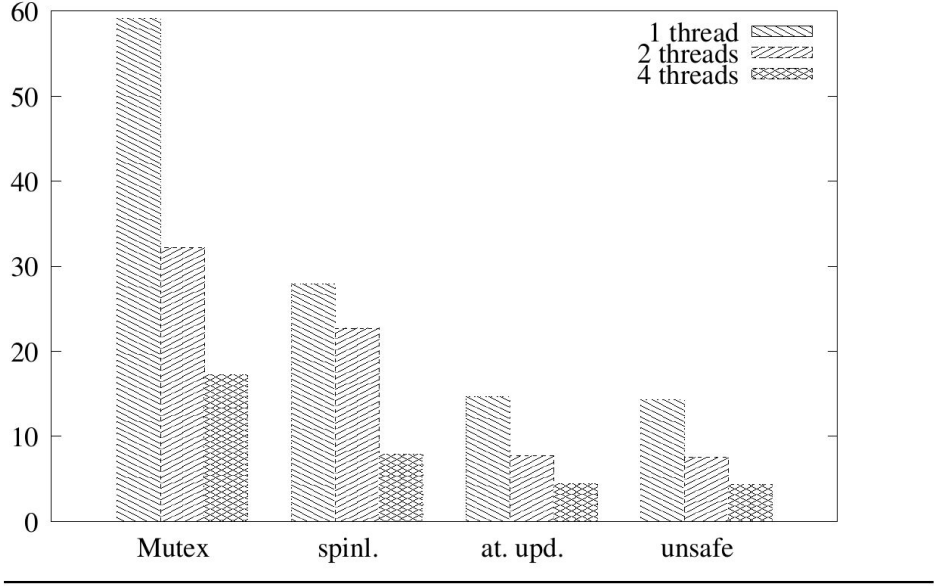


Figure 1. Sieve execution time for byte array (secs)

Performance Issues

Multithreaded Sieve of Eratosthenes Performance

- Unsanctioned methods are significantly faster!
- Better to break the rules to achieve better performance?

Conclusion of the Paper

Conclusion of the Paper

- Compiler optimizations can lead to correctness issues due to not understanding difference between a regular runtime library and a threading library
- Compiler cannot perform certain optimizations due to threading library information hiding
- Threads need to visible to the compiler

C++11 Memory Model

...

A. Alexandrescu, H.-J. Boehm, K. Henney, D. Lea, and B. Pugh. Memory model for multithreaded C++. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1680.pdf.

Memory Models

Memory Models

What is a memory model?

- A ‘contract’ between the programmer and the system
- Requires the programmer to obey certain rules

Memory Models

What was it like previously?

- Either used direct calls to OS-provided threading library (pthreads) or used an intervening layer (boost threads)
- Multithreading was very vague and unstandardized
- Lack of standardization led to everybody doing things differently

Memory Models

Well what's wrong with that?

- Nothing was truly abstract
- Multithreading wasn't officially supported by C++

Memory Models

Well what's wrong with that?

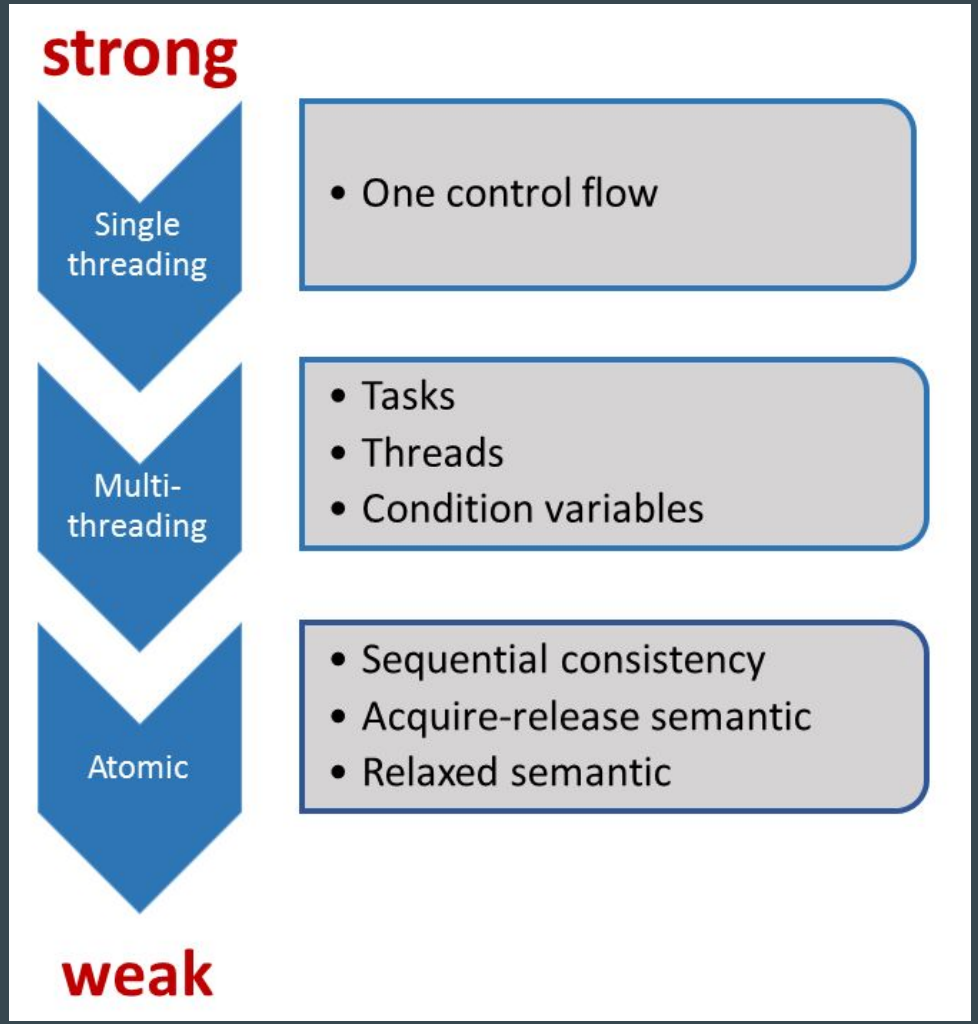
```
Global
int x, y;


Thread 1                Thread 2
x = 17;                 cout << y << " ";
y = 37;                 cout << x << endl;
```

Memory Models

How was this fixed?

- Specification of an abstract memory model describing the interactions between threads and memory
- Introduction of a small number of standard library classes providing standardized access to atomic update operations
- Definition of a standard thread library that provides similar functionality to pthreads and Win32 threads, but meshes with the rest of the C++ standard



https://www.modernescpp.com/images/blog/Speichermodell/Speichermodell/Overview.png

Memory Models

Fixes actualized?

```
Global

int x, y;
```

```
Thread 1

x = 17;

y = 37;
```

```
Thread 2

cout << y << " ";

cout << x << endl;
```

```
Global

atomic<int> x, y;
```

```
Thread 1

x.store(17);

y.store(37);
```

```
Thread 2

cout << y.load() << " ";

cout << x.load() << endl;
```

Memory Models

What's in the contract?

- Atomic Operations
- Visible Effects of Operations
- Memory Order

Atomic Operations

What are they?

```
namespace std {
    class atomic_int {
    public:
        int get();
        int set(int v);
        bool compare_and_set(int expected_value, int new_value);

        int weak_get();
        int weak_set(int v);
        bool weak_compare_and_set(int expected_value, int new_value);

        // other minor convenience functions, including:
        int get_and_increment();
        int get_and_add(int v);
        // ...
    };
}
```

Visible Effects of Operations

What are they?

- Atomicity
- Volatile data
- Opaque calls

Memory Order

What is it?

- Sequentially consistent
- Relaxed
- Acquire/release
- Release/consume

Memory Models

Sequential Consistency

```
Global

atomic<int> x, y;


Thread 1                Thread 2

x.store(17);            cout << y.load() << " ";

y.store(37);            cout << x.load() << endl;
```

Memory Order

Relaxed Ordering & Speedups

```
Global

atomic<int> x, y;
```

```
Thread 1

x.store(17, memory_order_relaxed);

y.store(37, memory_order_relaxed);
```

```
Thread 2

cout << y.load(memory_order_relaxed) << " ";

cout << x.load(memory_order_relaxed) <<
endl;
```

Memory Models

Acquire/Release Ordering

```
Global

atomic<int> x, y;
```

```
Thread 1

x.store(17, memory_order_release);

y.store(37, memory_order_release);
```

```
Thread 2

cout << y.load(memory_order_acquire) << " ";

cout << x.load(memory_order_acquire) <<
endl;
```

Memory Models

Release/Consume Ordering

```
Global

atomic<int> x, y;
```

```
Thread 1

x.store(17, memory_order_release);

y.store(37, memory_order_release);
```

```
Thread 2

cout << y.load(memory_order_consume) << " ";

cout << x.load(memory_order_consume) <<
endl;
```

What does this mean for us?

Why should I care?

- You don’t really need to manage threads unless seeking additional performance or control
- The weaker the contract, the faster the program

Problems that C++ didn’t have to think about

Java is good?

- Java enforces sequential consistency
- x86 isn’t natively sequentially consistent
- Barriers and fences

Any Questions?