CONSISTENCY MODELS FROM A PROGRAMMER'S PERSPECTIVE

Presented by: Beakal Lemeneh and Zeyu Xu

1

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

2

IMPLEMENTATION OF MULTI-THREADED PROGRAMS

- Threads provided as a part of language specification.
  - Java
  - C#
  - Ada
- Thread support provided as an add-on library.
  - C
  - C++

3

PTHREAD IMPLEMENTATION

What causes concurrency issues?

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

4

INFORMAL DEFINITION OF THE MEMORY MODEL FOR PTHREAD

"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"

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.

5

SOME OF THE 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()

6

REASON FOR USING THIS DEFINITION

"Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective"

7

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

8

THE PTHREAD'S 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

9

WHAT ARE OPAQUE FUNCTIONS?

- A call to a function that the compiler has no prior information about.
- This implies that the compiler can make no assumptions about the side effects of this call except what is guaranteed by the language definition..
- Further, the compiler can never skip making a call to this function even when calling it looks useless since there is no way for the compiler to know this for certain.

Source: https://stackoverflow.com/questions/48078415/what-does-an-opaque-function-call-mean-in-compiler-optimization

10

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++;
}
...
```

11

WHY ARE THESE "WORKAROUNDS" NEEDED?

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

12

PROBLEMATIC EXAMPLE:

```
if(globals.hasData) {
        int prelock_value = globals.foo;
        pthread_mutex_lock(&m);
        if(prelock_value != globals.foo) {
        // value changed before we could lock it, do something different
                DoSpecialStuffSinceValueChangedWhileWaiting();
                pthread_mutex_unlock(&m);
                return;
}
DoOtherStuff();
 ...
```

Source: https://stackoverflow.com/questions/26476139/why-do-global-variables-cause-trouble-for-compiler-optimizations-in-function-cal

13

PROBLEMATIC EXAMPLE:

```
if(globals.hasData) {
        int prelock_value = globals.foo;
        pthread_mutex_lock(&m);
        if(false /* always false: prelock_value != globals.foo */) {
        // value changed before we could lock it, do something different
                DoSpecialStuffSinceValueChangedWhileWaiting();
                pthread_mutex_unlock(&m);
                return;
}
DoOtherStuff();
 ...
```

Source: https://stackoverflow.com/questions/26476139/why-do-global-variables-cause-trouble-for-compiler-optimizations-in-function-cal

14

PROBLEMATIC EXAMPLE AFTER COMPILATION:

```
if(globals.hasData) {
        pthread_mutex_lock(&m);
        // everything was removed.
        DoOtherStuff();
 ...
```

Source: https://stackoverflow.com/questions/26476139/why-do-global-variables-cause-trouble-for-compiler-optimizations-in-function-cal

15

NOT ALL PROBLEMS ARE SOLVED

- 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.

16

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)

17

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

18

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;
```

19

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):

20

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):

21

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)

22

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)

23

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)

24

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)

25

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
- Authors mentioned no known real-life bugs caused by this, but still violation of pthreads specification

26

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

27

REWRITING OF ADJACENT DATA

Sample Code:

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

What does the struct look like in memory?

d	c	b	a

28

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'
```

29

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';
```

30

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';
```

31

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';
```

32

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';
```

33

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';
```

34

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

35

REWRITING OF ADJACENT DATA

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

36

REGISTER PROMOTION

- x is a global variable
- Lock is acquired conditionally

```
for (...) {
    ...
    if (mt) pthread_mutex_lock(...);
        x = ... x ...
    if (mt) pthread_mutex_unlock(...);
}
```

37

REGISTER PROMOTION

- x is a global variable
- Lock is acquired conditionally
- Assume the compiler determines that the conditionals are usually not taken

```
for (...) {
    ...
    if (mt) pthread_mutex_lock(...);
        x = ... x ...
    if (mt) pthread_mutex_unlock(...);
}
```

38

REGISTER PROMOTION

- x is a global variable
- Lock is acquired conditionally
- Assume the compiler determines that the conditionals are usually not taken
- r is a register variable

```
r=x;
for (...) {
    ...
    if (mt) {
        x = r; pthread_mutex_lock(...); r = x;
    }
    r = ... r ...
    if (mt) {
        x = r; pthread_mutex_unlock(...); r = x;
    }
}
x=r;
```

39

REGISTER PROMOTION

- x is a global variable
- Lock is acquired conditionally
- Assume the compiler determines that the conditionals are usually not taken
- r is a register variable
- modify x outside of critical section!

```
r=x;
for (...) {
    ...
    if (mt) {
        x = r; pthread_mutex_lock(...); r = x;
    }
    r = ... r ...
    if (mt) {
        x = r; pthread_mutex_unlock(...); r = x;
    }
}
x=r;
```

40

REWRITING OF ADJACENT DATA

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

41

PERFORMANCE ISSUES

- Pthreads restricts data contentions among threads
  - Potential benefit from allowing data races
    - Hardware atomic instructions are typically expensive
    - Hardware atomic instructions might implicitly prevent hardware reordering of memory references
- Synchronization is done through pthread primitives
  - Dynamic library calling overhead
  - Less flexibility than direct calls to hardware primitives

42

EXAMPLE: SIEVE OF ERATOSTHENES

```
for (my_prime = 2; my_prime < 10000; ++my_prime)
    if (!get(my_prime)) {
        for (multiple = my_prime; multiple < 100000000; multiple += my_prime)
        if (!get(multiple)) set(multiple);
    }
bool A[99999999] = {false, …, false};
get(i) {return A[i]; }
set(i) {A[i] = true; }
```

43

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

44

COLANDER 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

45

COLANDER 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

46

GRATER 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

47

GRATER 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

48

GAUZE 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

49

GAUZE 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

50

CENTRIFUGE 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

51

CENTRIFUGE 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

52

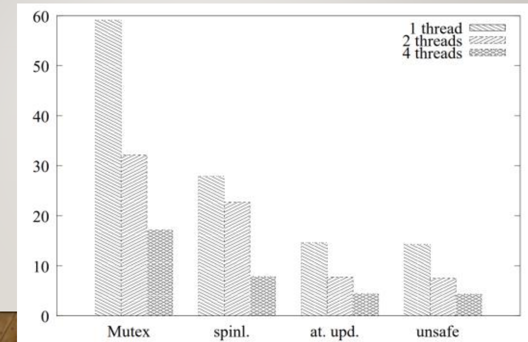
SIEVE OF ERATOSTHENES

	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

53

PERFORMANCE

- Mutex
- Spinlock
- unsafe



54

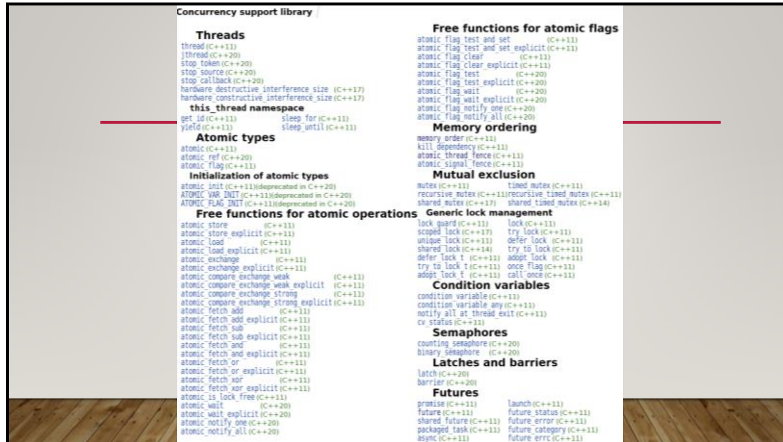
CONCLUSION

- 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 complier

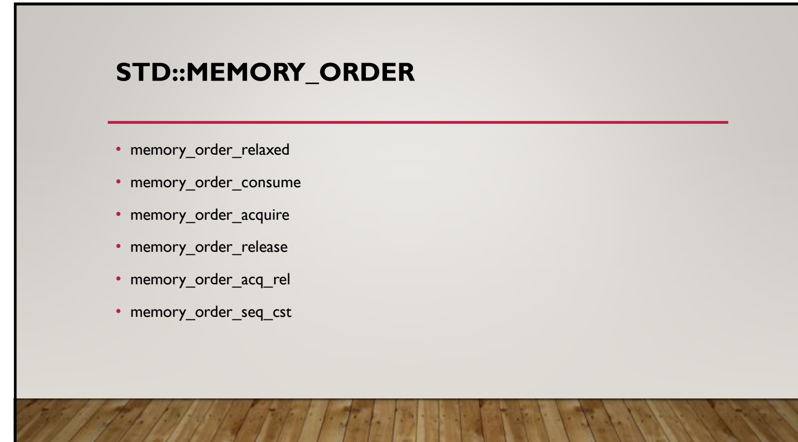
55

C++ MEMORY ORDER

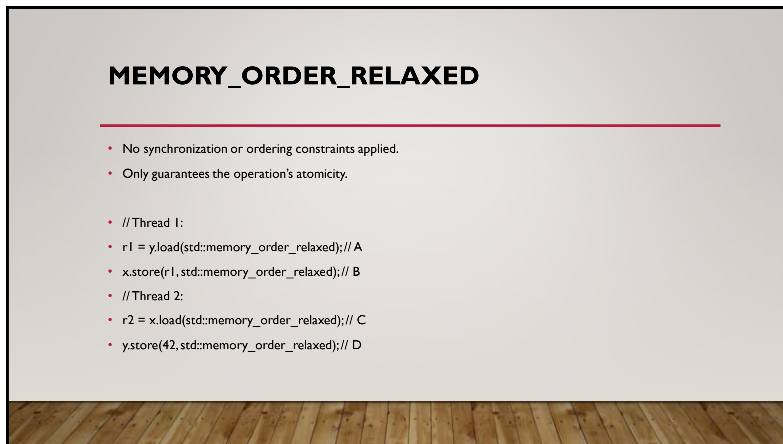
56



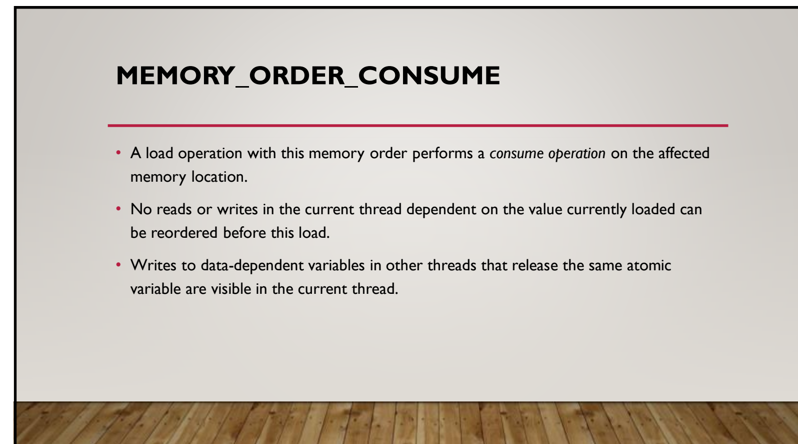
57



58



59



60

MEMORY_ORDER_ACQUIRE

- A load operation with this memory order performs the *acquire operation* on the affected memory location.
- No reads or writes in the current thread can be reordered before this load.
- All writes in other threads that release the same atomic variable are visible in the current thread.

61

MEMORY_ORDER_RELEASE

- A store operation with this memory order performs the release operation.
- No reads or writes in the current thread can be reordered after this store.
- All writes in the current thread are visible in other threads that acquire the same atomic variable.
- All writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.

62

MEMORY_ORDER_ACQ_REL

- A read-modify-write operation with this memory order is both an *acquire operation* and a *release operation*.

63

MEMORY_ORDER_SEQ_CST

- A load operation with this memory order performs an *acquire operation*
- A store performs a *release operation.*
- A read-modify-write performs both an *acquire operation* and a *release operation.*

64

RELAXED ORDERING

```
1 #include <atomic>
2 #include <thread>
3 #include <vector>
4 #include <iostream>
5 #include <cstdio>
6 #include <string>
7
8 using namespace std;
9
10 int rx(1), ry(1);
11 atomic<int> x, y;
12
13
14 void f1(){
15     ry = y.load(memory_order_relaxed);
16     x.store(ry, memory_order_relaxed);
17 }
18
19 void f2(){
20     rx = x.load(memory_order_relaxed);
21     y.store(42, memory_order_relaxed);
22 }
23
24 int main(){
25     thread t1(f1);
26     thread t2(f2);
27     t1.join();
28     t2.join();
29
30     cout << "x = " << rx << "\ny = " << ry << endl;
31 }
```

```
[zxu44@node01 experiments]$ taskset -c 0,1 ./relaxed
x = 0
y = 0
[zxu44@node01 experiments]$ taskset -c 0,60 ./relaxed
x = 0
y = 42
```

X == 42, y == 42 also allowed

65

RELEASE-CONSUME ORDERING

```
1 #include <thread>
2 #include <atomic>
3 #include <cassert>
4 #include <string>
5
6 std::atomic<std::string*> ptr;
7 int data;
8
9 void producer()
10 {
11     std::string* p  = new std::string("Hello");
12     data = 42;
13     ptr.store(p, std::memory_order_release);
14 }
15
16 void consumer()
17 {
18     std::string* p2;
19     while (!(p2 = ptr.load(std::memory_order_consume)))
20         ;
21     //assert(*p2 == "Hello"); // never fires: *p2 carries dependency from ptr
22     assert(data == 42); // may or may not fire: data does not carry dependency from ptr
23 }
24
25 int main()
26 {
27     std::thread t1(producer);
28     std::thread t2(consumer);
29     t1.join(); t2.join();
30 }
```

66

VOLATILE

- Within a single thread of execution, a volatile access cannot be optimized out or reordered relative to another visible side effect that is separated by a sequence point from the volatile access.
- Volatile access does not establish inter-thread synchronization.
- Volatile accesses are not atomic (concurrent read and write is a data race) and do not order memory (non-volatile memory accesses may be freely reordered around the volatile access).

67

BIBLOGRAPHY

- 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
- "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.
- https://stackoverflow.com/questions/26476139/why-do-global-variables-cause-trouble-for-compiler-optimizations-in-function-cal
- Alex Mononen and Jack Yu's Slide from the previous year

68