C++ 11 Memory Consistency Model

Sebastian Gerstenberg

NUMA Seminar

07.01.2015

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Agenda

1. Sequential Consistency
2. Violation of Sequential Consistency
   - Non-Atomic Operations
   - Instruction Reordering
3. C++ 11 Memory Consistency Model
4. Trade-Off - Examples
5. Conclusion

Agenda

1. **Sequential Consistency**
2. Violation of Sequential Consistency
   - Non-Atomic Operations
   - Instruction Reordering
3. C++ 11 Memory Consistency Model
4. Trade-Off - Examples
5. Conclusion

Sequential Consistency

"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

-Leslie Lamport

Sequential Consistency - Ordering

Maintaining program order among operations on individual processors

Dekkers Algorithm:

P1

Flag1 = 1;

If(Flag2 == 0)

… critical section

P2

Flag2 = 1;

if(Flag1 == 0)

… critical section

Sequential Consistency - Atomicity

Maintaining a single sequential order among operations of all processors

A = B = C = D = 0

P1	P2	P3	P4
A = 1;	A = 2;	while(B!=1){;}	while(B!=1){;}
B = 1;	C = 1;	while(C!=1){;}	while(C!=1){;}
		print(A);	print(A);

Agenda

1. Sequential Consistency
2. **Violation of Sequential Consistency**
   - Non-Atomic Operations
   - Instruction Reordering
3. C++ 11 Memory Consistency Model
4. Trade-Off - Examples
5. Conclusion

Violation in UMA Systems

P1

Flag1 = 1;

If(Flag2 == 0)

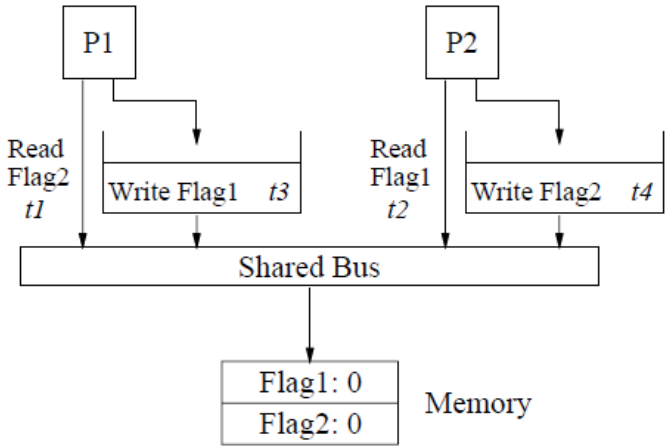
… critical section

P2

Flag2 = 1;

if(Flag1 == 0)

… critical section



http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

Violation in NUMA Systems

P1

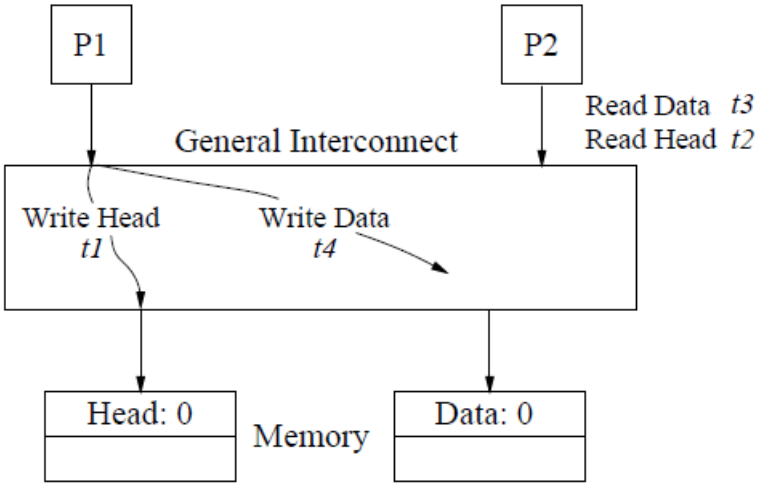
Data = 1000;

Head = 1;

P2

while(!Head) {;}

… work on data



http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

Compiler

- Dekkers Algorithm, g++ -O2, read and write are switched

```
owner@vmTest:~/src$ cat dekker.cpp

int flag;
int flag2;
int data;

int main() {
        if(flag2 == 0)
                data = 42;
        flag = 1;
        if(flag2 == 0)
                return 0;
}
```

```
main:
.LFB0:
        .cfi_startproc
        movl    flag2, %eax     read flag2
        testl   %eax, %eax      comp flag2
        jne     .L2
        movl    $42, data
.L2:
        movl    $1, flag        write flag
        xorl    %eax, %eax
        ret
        .cfi_endproc
```

Out of Order Execution

Processor avoids being idle by executing instructions out of order

- Weak Memory Model (PowerPC, ARM)
  - may reorder any instructions
  - exception: data dependency ordering:

```
x = 1;
y = 2;
```

may be reordered

```
x = 1;
y = x;
```

may not be reordered

- Strong Memory Model (X86, SPARC)
  - stricter rules apply to reordering (x86 allows only store-load reordering)

Agenda

1. Sequential Consistency
2. Violation of Sequential Consistency
   - Non-Atomic Operations
   - Instruction Reordering
3. **C++ 11 Memory Consistency Model**
4. Trade-Off - Examples
5. Conclusion

C++ 11 std::atomic

Strictly enforces *Sequential Consistency* (*default*) by giving three guarantees:

- Operations on std::atomic is atomic
- No instruction reordering past std::atomic operations
- No out-of-order execution of std::atomic operations

Similar to Java & C# volatile keyword (not similar to C++ volatile!)

C++ 11 std::atomic

header <atomic>

- Template
- load, store, compare_exchange
- operations allow a specific memory order
  - sequential consistency by default
- Specialization for integral types (int, char, bool …)
- specialized instructions (and operator overloading) for integral types
  - fetch_add/sub (+= , -=)
  - fetch_and/or/xor (&= , |= , ^=)
  - operator++/--

C++ 11 std::atomic - assembler

```
#include <atomic>

std::atomic_int flag;
std::atomic_int flag2;
int data;

int main() {
    if(flag2 == 0)


        data = 42;

    flag = 1;

    if(flag2 == 0)

        return 0;
}
```

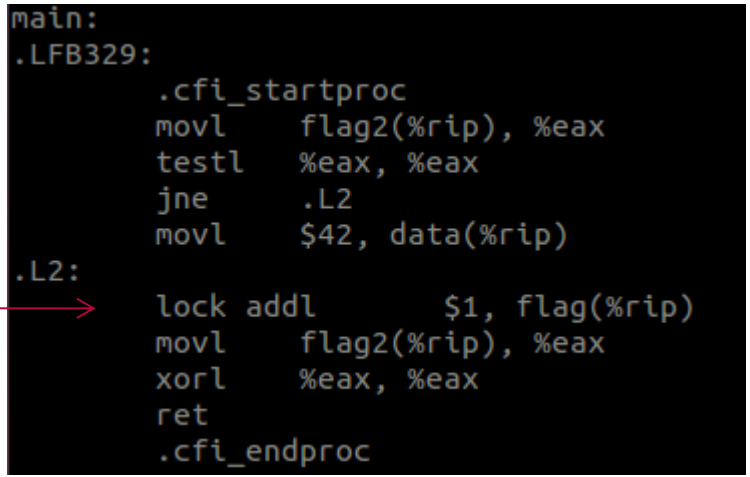
C++ 11 std::atomic - assembler

```
#include <atomic>

std::atomic_int flag;
std::atomic_int flag2;
int data;

int main() {
    if(flag2 == 0)


        data = 42;

    flag = 1;

    if(flag2 == 0)

        return 0;
}
```

```
main:
.LFB329:
        .cfi_startproc
        movl    flag2(%rip), %eax
        testl   %eax, %eax
        jne     .L2
        movl    $42, data(%rip)
.L2:
        movl    $1, flag(%rip)
        mfence
        movl    flag2(%rip), %eax
        xorl    %eax, %eax
        ret
        .cfi_endproc
```

C++ 11 std::atomic - assembler

```
#include <atomic>

std::atomic_int flag(0);
std::atomic_int flag2;
int data;

int main() {
    if(flag2 == 0)


        data = 42;

    flag++;
    if(flag2 == 0)

        return 0;
}
```



```
main:
.LFB329:
        .cfi_startproc
        movl    flag2(%rip), %eax
        testl   %eax, %eax
        jne     .L2
        movl    $42, data(%rip)
.L2:
        lock addl       $1, flag(%rip)
        movl    flag2(%rip), %eax
        xorl    %eax, %eax
        ret
        .cfi_endproc
```

C++ 11 Atomics Memory Ordering

Different memory models can be applied to specific operations

- memory_order_seq_cst: default
  enforces sequential consistency
- memory_order_acquire: load only (needs associated release)
  all writes before release are visible side effects after this operation
- memory_order_release: store only (needs associated acquire)
  preceding writes are visible after associated acquire operation
- memory_order_acq_rel: combination of both acquire and release
- memory_order_relaxed: no memory ordering, atomicity only

Agenda

1. Sequential Consistency
2. Violation of Sequential Consistency
   - Non-Atomic Operations
   - Instruction Reordering
3. C++ 11 Memory Consistency Model
4. **Trade-Off - Examples**
5. Conclusion

Memory Barriers

```
void produce() {
    payload = 42;
    guard.store(1, std::memory_order_release)
}

void consume(int iterations) {
    for(int i = 0; i < iterations; i++){
        if(guard.load(std::memory_order_acquire))
            result[i] = payload;
    }
}
```

Memory Barriers

Intel x86

```
mov     ecx, dword ptr [rip + _Guard]
test    ecx, ecx
cmovne  eax, dword ptr [rip + _Payload]
```

load from Guard

load from Payload

ARM V7

```
add     r3, pc
ldr.w   r4, [r9]
dmb     ish
ldr     r5, [r3]
cmp     r4, #0
it      ne
movne   r2, r5
```

load from Guard

memory barrier

load from Payload

PowerPC

```
lis     r8, Guard@ha
addi    r8, r8, Guard@l
lis     r7, Payload@ha
lwz     r9, 0(r8)
cmpw    cr7, r9, r9
bne-    cr7, $+4
isync
cmpwi   cr7, r9, 0
beq-    cr7, .L0
lwz     r10, Payload@l(r7)
```

load from Guard

memory barrier

load from Payload

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

Memory Barries

1000 iterations:

Intel x86: strong memory model
implicit acquire-release consistency

~~#LoadLoad~~	~~#LoadStore~~
#StoreLoad	~~#StoreStore~~

ARM v7, PowerPC: weak memory model
casual consistency
needs memory barriers for acquire-release consistency

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

Memory Models – CPU Architecture

http://preshing.com/20120930/weak-vs-strong-memory-models/

Agenda

1. Sequential Consistency
2. Violation of Sequential Consistency
   - Non-Atomic Operations
   - Instruction Reordering
3. C++ 11 Memory Consistency Model
4. Trade-Off - Examples
5. **Conclusion**

Conclusion

std::atomics provide
simple,
multiplatform,
lock-free thread synchronization

at the cost of runtime performance through
enforcing atomicity of longrunning operations
locally disabling compiler optimization
locally disabling out-of-order execution

the performance impact can be reduced by
using atomics sparcely (obviously)
specifying special memory ordering when ever possible.

Sources

- http://en.cppreference.com/w/cpp/atomic/atomic
- http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf
- http://preshing.com
- https://peeterjoot.wordpress.com/tag/memory-barrier/
- http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html
- http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

Image sources as listed below each image

Thank you
for your attention!

Sebastian Gerstenberg

NUMA Seminar

07.01.2015