Lightning Talk: The C++11 Memory Model

Meeting C++ 2012
Neuss, Germany

Presented by Marc Mutz

Produced by Klarälvdalens Datakonsult AB

Material based on N3337, created on November 10, 2012

Module: The C++11 Memory Model

C++11 Multithreaded Execution Guarantees

- C++11 is the first C++ standard to mention multithreading.
- Only minimal progress guarantees are given:
  - Unblocked threads should “eventually make progress”.
  - Implementations should ensure that writes in one thread become visible to other threads “in a finite amount of time”.

The C++11 Memory^WConsistency Model

- Strict Consistency
  - requires global clock
  - too strict for the real world
- Sequential Consistency
  - L. Lamport, 1978: “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.”
  - IOW: Threads are executed as if thread steps were just interleaved.

Dekker's Example

- Scenario (initally $x = y = 0$):

Thread A	Thread B
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

- SC analysis:

```
x = 1; r1 = y;  y = 1; r2 = x; // r1==0; r2==1
x = 1;  y = 1; r1 = y; r2 = x; // r1==1; r2==1
x = 1;  y = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1; r2 = x;  x = 1; r1 = y; // r1==1; r2==0
y = 1;  x = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1;  x = 1; r1 = y; r2 = x; // r1==1; r2==1
```

$\Rightarrow \{r1 = 0\} \cap \{r2 = 0\} = \emptyset$

- But happens all the time on real hardware!
- Solution: "Sequential Consistency for Data-Race-Free Programs" (Boehm)

Dekker's Example

- Scenario (initally $x = y = 0$):

Thread *A*	Thread B
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

- SC analysis:

```
x = 1; r1 = y;  y = 1; r2 = x; // r1==0; r2==1
x = 1;  y = 1; r1 = y; r2 = x; // r1==1; r2==1
x = 1;  y = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1; r2 = x;  x = 1; r1 = y; // r1==1; r2==0
y = 1;  x = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1;  x = 1; r1 = y; r2 = x; // r1==1; r2==1
```

$\Rightarrow \{r1 = 0\} \cap \{r2 = 0\} = \emptyset$

- But happens all the time on real hardware!
- Solution: "Sequential Consistency for Data-Race-Free Programs" (Boehm)

Dekker's Example

- Scenario (initally $x = y = 0$):

Thread A	Thread B
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

- SC analysis:

```
x = 1; r1 = y;  y = 1; r2 = x; // r1==0; r2==1
x = 1;  y = 1; r1 = y; r2 = x; // r1==1; r2==1
x = 1;  y = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1; r2 = x;  x = 1; r1 = y; // r1==1; r2==0
y = 1;  x = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1;  x = 1; r1 = y; r2 = x; // r1==1; r2==1
```

$\Rightarrow \{r1 = 0\} \cap \{r2 = 0\} = \emptyset$

- But happens all the time on real hardware!
- Solution: "Sequential Consistency for Data-Race-Free Programs" (Boehm)

Dekker's Example

Thread *A*	Thread *B*
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

- Scenario (initally $x = y = 0$):
- SC analysis:

```
x = 1; r1 = y;  y = 1; r2 = x; // r1==0; r2==1
x = 1;  y = 1; r1 = y; r2 = x; // r1==1; r2==1
x = 1;  y = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1; r2 = x;  x = 1; r1 = y; // r1==1; r2==0
y = 1;  x = 1; r2 = x; r1 = y; // r1==1; r2==1
y = 1;  x = 1; r1 = y; r2 = x; // r1==1; r2==1
```

$\Rightarrow \{r1 = 0\} \cap \{r2 = 0\} = \emptyset$

- But happens all the time on real hardware!
- Solution: “Sequential Consistency for Data-Race-Free Programs” (Boehm)

Data Races in C++11

- Dekker's Example contains C++11 *Data Races* (on *x* and *y*):
  - "Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location." [intro.multithread]/3
  - "The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other." [intro.multithread]/14
  - And: "Any such data race results in undefined behavior."
- Under SC-for-DRF, Dekker's Example exhibits undefined behavior, so we cannot reason about it.

Data Races in C++11

- Dekker's Example contains C++11 *Data Races* (on *x* and *y*):
  - "Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location." [intro.multithread]/3
  - "The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other." [intro.multithread]/14
  - And: "Any such data race results in undefined behavior."
- Under SC-for-DRF, Dekker's Example exhibits undefined behavior, so we cannot reason about it.

Data Races in C++11

- Dekker's Example contains C++11 *Data Races* (on *x* and *y*):
  - "Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location." [intro.multithread]/3
  - "The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other." [intro.multithread]/14
  - And: "Any such data race results in undefined behavior."
- Under SC-for-DRF, Dekker's Example exhibits undefined behavior, so we cannot reason about it.

Fixing Dekker's Example

- Since
  - "The execution of a program contains a *data race* if it contains two conflicting actions in different threads, **at least one of which is not atomic**, and neither happens before the other."

  atomic operations don't participate in data races.
- $\Rightarrow$ declare *x*, *y* as atomic:

```
std::atomic<int> x, y;
std::atomic_int  x, y;
QAtomicInt       x, y;
```

Thread *A*

```
x.store(1);
r1 = y.load();
```

Thread B

```
y.store(1);
r2 = x.load();
```

- Warning: this is only true if you don't specify a custom *memory ordering*.

The C++11 Happens-Before Relation

- “An evaluation A *happens before* an evaluation B if:
  - A is sequenced before B, or
  - A inter-thread happens before B.”

  [intro.multithread]/11
- A *Inter-thread happens before* B $\equiv$ “there’s a synchronisation point between A and B”
- Almost exhaustive list:
  - Thread creation synchronizes with start of thread execution.
  - Thread completion synchronizes with the return of the join.
  - Unlocking a mutex synchronizes with locking the same mutex.
- That’s all!!

Fixing Dekker's Example II

- Remember:
  - "The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and **neither happens before the other**."
  - Unlocking a mutex synchronizes with locking the same mutex.
- $\Rightarrow$ Protect *x*, *y* with mutex(es):

Thread *A*

```
x_mutex.lock():
x = 1;
x_mutex.unlock();

y_mutex.lock();
r1 = y;
y_mutex.unlock();
```

Thread B

```
y_mutex.lock();
y = 1;
y_mutex.unlock();

x_mutex.lock();
r2 = x;
x_mutex.unlock();
```

SC-for-DRF Programmer's Recipe

1. Check for and eliminate *C++11 Data Races*
2. Divide your source into blocks separated by synchronization primitives
3. Do the classical SC analysis, assuming these blocks *execu atomically*

SC-for-DRF Programmer's Recipe

1. Check for and eliminate *C++11 Data Races*
2. Divide your source into blocks separated by synchronization primitives
3. Do the classical SC analysis, assuming these blocks *execu atomically*

SC-for-DRF Programmer's Recipe

1. Check for and eliminate *C++11 Data Races*
2. Divide your source into blocks separated by synchronization primitives
3. Do the classical SC analysis, assuming these blocks *execute atomically*

Memory Locations

- "A memory location is either
  - an object of scalar type
  - or a maximal sequence of adjacent bit-fields all having non-zero width."
- "Two threads of execution can update and access separate memory locations without interfering with each other."
- Example (contains four memory locations):

```
struct {
    char a;
    // ----
    int b:5, c:11, :0,
    // ----
    d:8;
    // ----
    struct { int ee:8; } e;
};
```

Memory Locations cont'd

- "Two threads of execution can update and access separate memory locations without interfering with each other."
- Example (Boehm): no data race under C++11, potentially under Posix:

```
struct { char a; char b; } x; /*T1*/ x.a = 1; /*T2*/ x.b = 1;
```

Memory Locations cont'd

- Example (Linux):

```
struct btrfs_block_rsv {
    u64 size;
    u64 reserved;
    struct btrfs_space_info *space_info;
    spinlock_t lock;
    unsigned int full:1;
};
```

"We actually spotted this race in practice in btrfs on structure fs/btrfs/ctree.h:struct btrfs_block_rsv where spinlock content got corrupted due to update of following bitfield and there seem to be other places in kernel where this could happen." (Jan Kara on LKML)