Concurrent Systems
Exercise 03 – Memory, Atomicity, Consistency

Stefan Reif

2018-11-26
Agenda

Memory Models

Sequential Consistency

C11/C++11 Atomic Types

C/C++ Memory Consistency Models

Thread Fences

Assignment 3
What is a memory model?

- Formal definition of memory behavior
  - More or less trivial for sequential programs
  - Complex for parallel programs

- Crucial for application correctness
  - Particularly for parallel programs

- Memory models are created by ...
  - Language designers
  - Hardware developers
  - Service providers
Sequential Consistency

Def. *Sequential Consistency*  
Lamport, 1979

“[…] 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.”

**Benefits of SC**
- Result is equivalent to a sequential execution
  - Pseudo-parallelism?
- Global order of actions
- Analysability

**Problems of SC**
- Even simple compiler optimisations violate SC
- Strictly sequential memory becomes the system bottleneck
Why don’t we always want SC?

- Modern Platforms are not WYSIWYG\(^1\)
  - Optimizing Compiler, Assembler, Linker, Hardware
  - Every link in the tool-chain can reorder operations
    - We nearly always want that ...
    - ... but it can break parallel code
    - It does not matter which tool breaks our code

- SC makes optimisations hard

\[ \text{More Reordering} \]

\[ \text{SC} \rightarrow \text{Analysability} \rightarrow \text{Bugs?} \]

\[ \text{Correctness} \rightarrow \text{Performance?} \]

\(^1\)What You See Is What You Get
Sequential Consistency – Data Race Free

- **SC-DRF**
  - Distinguish *synchronising* actions from *non-synchronising* actions
    - Guarantee SC for programs without data races
    - Undefined behavior in case of data race
    - Most code parts allow optimisation
  - Synchronising actions provide SC
    - e.g. mutex.lock, atomic_fetch_add, ...
  - Non-synchronising actions can cause data races

---

**Def. Data Race**

“Two expression evaluations *conflict* if one of them modifies a memory location and the other one reads or modifies the same memory location.”

“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.”

---

C11 Standard (Draft)
Sequential Consistency – Data Race Free

thread1

thread2
Sequential Consistency – Data Race Free

thread1

thread2

Sequential Consistency (2018-11-26)
Sequential Consistency – Data Race Free

thread1

thread2
Modern C/C++ offers atomic types

- C ⇒ _Atomic qualifier
- C++ ⇒ std::atomic<type> template
- Intentionally compatible:

```cpp
#ifdef __cplusplus
#define _Atomic(type) std::atomic<type>
#endif
```

Well-defined semantics

- Compiler guarantees atomicity
- *unlike volatile* ...

Portable

- Actual run-time properties still depend on the hardware
- Lock-freedom is not guaranteed
# Atomic Operations

## Atomic operations

```c
#include <stdatomic.h>

atomic_store(AT*, T);
atomic_load(AT*);
atomic_exchange(AT*, T);
atomic_compare_exchange_strong(AT*, T*, T);
atomic_compare_exchange_weak(AT*, T*, T);
atomic_fetch_add(AT*, T);
atomic_fetch_sub(AT*, T);
atomic_fetch_or(AT*, T);
atomic_fetch_xor(AT*, T);
atomic_fetch_and(AT*, T);
```

Atomic operations with explicit memory order parameter

```c
#include <stdatomic.h>

atomic_store_explicit(AT*, T, MO);
atomic_load_explicit(AT*, MO);
atomic_exchange_explicit(AT*, T, MO);
atomic_compare_exchange_strong_explicit(AT*, T*, T, MO, MO);
atomic_compare_exchange_weak_explicit(AT*, T*, T, MO, MO);
atomic_fetch_add_explicit(AT*, T, MO);
atomic_fetch_sub_explicit(AT*, T, MO);
atomic_fetch_or_explicit(AT*, T, MO);
atomic_fetch_xor_explicit(AT*, T, MO);
atomic_fetch_and_explicit(AT*, T, MO);
```
C/C++ Memory Order Parameters

1. sequential-consistent
   - memory_order_seq_cst

2. acquire-release
   - memory_order_acquire
   - memory_order_release
   - memory_order_acq_rel

3. consume-release
   - memory_order_consume
   - memory_order_release

4. relaxed
   - memory_order_relaxed
C/C++ Memory Order Parameters

1. sequential-consistent
   - memory_order_seq_cst

2. acquire-release
   - memory_order_acquire
   - memory_order_release
   - memory_order_acq_rel

3. consume-release
   - memory_order_consume
   - memory_order_release

4. relaxed
   - memory_order_relaxed
memory_order_seq_cst

- Strongest consistency model available

- Implicit consistency model
  - All operations without _explicit

- Semantics: SC-DRF
  - All operations with `memory_order_seq_cst` are sequentially consistent
  - Non-atomic variables can cause data races
memory_order_{acquire,release,acq_rel}

- Happens-before relation
  - Intra-thread: trivial
  - Inter-thread: If $S_A$ release-stores a value $v$ in $A$ and $L_A$ acquire-loads that value, then $S_A$ happens before $L_A$.
  - $S_A$ also happens before $L_A$ if $L_A$ reads a later value $v'$ of $A$
    \[\rightarrow\] modification order
  - Transitive relation

- Acquire/Release consistency
  - Each operation sees side effects of all operations that happened before

- Partial Ordering
  - No global sequence of operations
  - Synchronisation per variable
  - Concurrent operations are possible
```plaintext
memory_order_{acquire,release,acq_rel}

- **acquire → load**
  - *fetch-all-data*
  - Side effects after load-acquire remain afterwards
  - Implicit in `thrd_join`, `mtx_lock`

- **release → store**
  - *push-all-data*
  - Side effects before store-release remain before
  - Implicit in `thrd_exit`, `mtx_unlock`

- **acq_rel → load/store**
  - e.g. `atomic_fetch_and_*`, `atomic_compare_exchange_*`
```
memory_order_{acquire,release,acq_rel} (3)
memory_order_{acquire,release,acq_rel}
memory_order_{acquire, release, acq_rel} (3)
memory_order_{acquire,release,acq_rel} (3)
memory_order_{acquire,release,acq_rel} (4)
memory_order_{acquire,release,acq_rel} (4)
memory_order_\{acquire,release,acq\_rel\}  (5)

Example:

```c
int data; _Atomic(bool) avail;
thread1() {
    data = createdata();  // not atomic
    atomic_store_explicit(&avail, 1,
                           memory_order_release);
}
thread2() {
    if (atomic_load_explicit(&avail,
                             memory_order_acquire)) {
        int mycopy = data; use(mycopy);
    }
}
```

Example:

```c
_Atomic(foo *) data;
thread1() {
    foo *d = createdata();  // not atomic
    atomic_store_explicit(&data, d,
                          memory_order_release);
}
thread2() {
    foo *d = atomic_load_explicit(&data,
                                  memory_order_acquire);
    if (d)
        use(d);
}
```
memory_order_relaxed

- Weakest memory order parameter
  - Meaningful when additional mechanisms synchronise

- Ensures atomicity
  - No synchronisation
  - Lock-freedom not guaranteed

- No reordering constraints
  - Except for thread fences ...
  - Dangerous to use

- Performance improvement?
```c
_Atomic(unsigned int) counter;

search() {
    while (...) {
        if (...)
        atomic_fetch_add_explicit(&counter, 1,
                                memory_order_relaxed);
    }
    thrd_exit(...);
}

main() {
    for (...) thrd_create(..., search, ...);
    for (...) thrd_join(...);
    unsigned int total = atomic_load_explicit(&counter,
                                             memory_order_relaxed);
}
```
```c
_Atomic(unsigned int) counter;

search() {
    while (...) {
        if (...) 
            atomic_fetch_add_explicit(&counter, 1,
                                      memory_order_relaxed);
    }
    thrd_exit(...);
}

main() {
    for (...) thrd_create(..., search, ...);
    for (...) thrd_join(...);
    unsigned int total = atomic_load_explicit(&counter,
                                              memory_order_relaxed);
}
Enforces additional memory ordering guarantees
- Often better: use stronger memory order parameter at critical operation

Parameter: memory order
- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_seq_cst

Improves consistency of relaxed-atomic operations
- Useful for library functions
atomic_thread_fence

thread1

relaxed

A

relaxed

thread2
atomic_thread_fence

(2)

thread1

thread2

relaxed

relaxed

A

X

X

7

7

acquire

release
atomic_thread_fence

thread1

thread2

release

relaxed

X

A

relaxed

acquire
atomic_thread_fence

thread1

thread2

release

relaxed

X

relaxed

A

acquire

(2)
atomic_thread_fence

- thread1
  - release
  - relaxed
- thread2
  - relaxed
  - acquire
```c
int data; atomic_bool avail;

thread1() {
    data = createdata(); // not atomic

    atomic_store_explicit(&avail, 1,
                           memory_order_relaxed);
}

thread2() {
    if (atomic_load_explicit(&avail,
                              memory_order_relaxed)) {

        int mydata = data; use(mydata);
    }
}
```
```c
int data; atomic_bool avail;
thread1() {
    data = createdata(); // not atomic
    atomic_thread_fence(memory_order_release);
    atomic_store_explicit(&avail, 1,
                          memory_order_relaxed);
}
thread2() {
    if (atomic_load_explicit(&avail,
                              memory_order_relaxed)) {
        atomic_thread_fence(memory_order_acquire);
        int mydata = data; use(mydata);
    }
}
```
Implement lock algorithms
- Lock algorithms are discussed in the lecture

Test all lock algorithms

Evaluate all lock algorithms
- Vary the degree of contention
- Check multiple back-off algorithms

Use a model checker for concurrent data structures
- CDSChecker ➔ plrg.eecs.uci.edu/software_page/42-2/
- Write a unit test for at least one lock algorithm
Implement a simple actor library
- Actors send requests asynchronously
- Each actor owns a worker thread
- Requests are sequentialised implicitly

Shortcomings
- No actor states and mutation
- No actor migration
- ...

Test your actor library
- Write test cases for your actor
- Can you re-use the lock test cases?

Evaluate your actor library
- Compare your actor implementation against lock-based synchronisation
- Under which conditions are actors a better choice?
**compare_exchange_weak**

- Weak ⇒ Operation may fail spuriously
- Efficient implementation with LL/SC possible
- Maybe ABA-free

**Example:**

```c
int n, o = atomic_load(&shared);
do {
    n = compute(o);
    if (atomic_compare_exchange_weak(&shared, &o, n))
        break;
} while (1);
```
**compare_exchange_weak**

- **weak** ⇒ Operation may fail spuriously
- Efficient implementation with LL/SC possible
- maybe ABA-free

**Example:**

```c
int n, o = LL(&shared);
do {
    n = compute(o);
    if (SC(&shared, n))
        break;
    o = LL(&shared);
} while (1);
```