

Concurrent Systems

Exercise 03 – Memory, Atomicity, Consistency

Stefan Reif

2017-11-27



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



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
- Analyzability

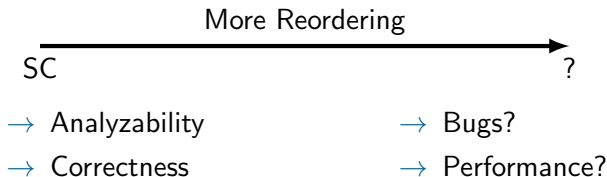
■ Problems of SC

- Even simple compiler optimizations violate SC



Why don't we always want SC?

- Modern Platforms are not WYSIWYG¹
 - 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 link breaks our code
- SC makes optimizations hard



¹What You See Is What You Get



■ SC-DRF

- Distinguish synchronizing actions from non-synchronizing actions
 - Guarantee SC for programs without data races
 - Undefined behavior in case of data race
 - Most code parts allow optimization
- Synchronizing actions provide SC
 - e.g. `mutex_lock`, `atomic_fetch_add`, ...
- Non-synchronizing actions can cause data races

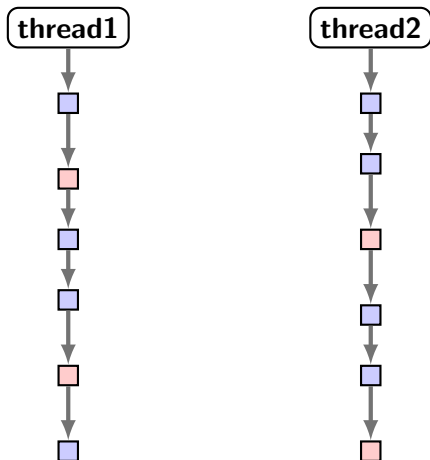
Def. *Data Race*

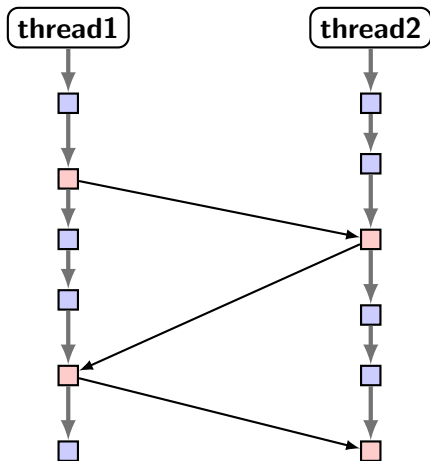
C11 Standard (Draft)

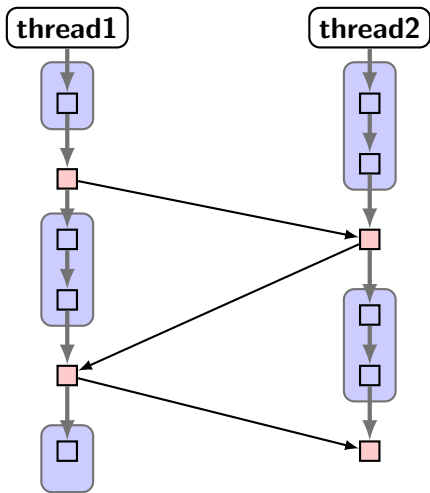
“Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location.”

“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.”









■ Modern C/C++ offers atomic types

- C $\Rightarrow$ `_Atomic` qualifier
- C++ $\Rightarrow$ `std::atomic<type>` template
- Intentionally compatible:

```
1  #ifdef __cplusplus
2  #define _Atomic(type) std::atomic<type>
3  #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

```
1  #include <stdatomic.h>
2  atomic_store(AT*, T);
3  atomic_load(AT*);
4  atomic_exchange(AT*, T);
5  atomic_compare_exchange_strong(AT*, T*, T);
6  atomic_compare_exchange_weak(AT*, T*, T);
7  atomic_fetch_add(AT*, T);
8  atomic_fetch_sub(AT*, T);
9  atomic_fetch_or(AT*, T);
10 atomic_fetch_xor(AT*, T);
11 atomic_fetch_and(AT*, T);
```



■ Atomic operations with explicit memory order parameter

```
1  #include <stdatomic.h>
2  atomic_store_explicit(AT*,T, MO);
3  atomic_load_explicit(AT*,MO);
4  atomic_exchange_explicit(AT*, T, MO);
5  atomic_compare_exchange_strong_explicit(AT*,
6      T*,T,MO,MO);
7  atomic_compare_exchange_weak_explicit(AT*,
8      T*,T,MO,MO);
9  atomic_fetch_add_explicit(AT*, T, MO);
10 atomic_fetch_sub_explicit(AT*, T, MO);
11 atomic_fetch_or_explicit(AT*, T, MO);
12 atomic_fetch_xor_explicit(AT*, T, MO);
13 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`

Consistency

Optimization

Performance?



- Strongest consistency model
- 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



■ 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
 - modification order
- Transitive relation

■ Acquire/Release consistency

- Each operation sees side effects of all operations that happened before

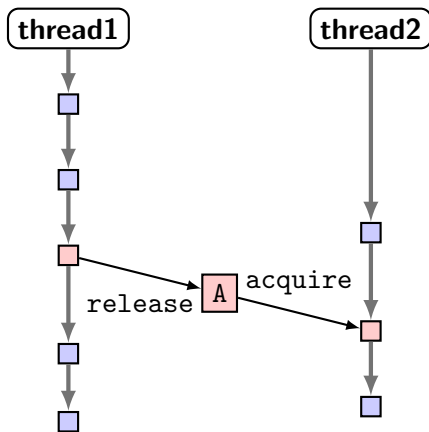
■ Partial Ordering

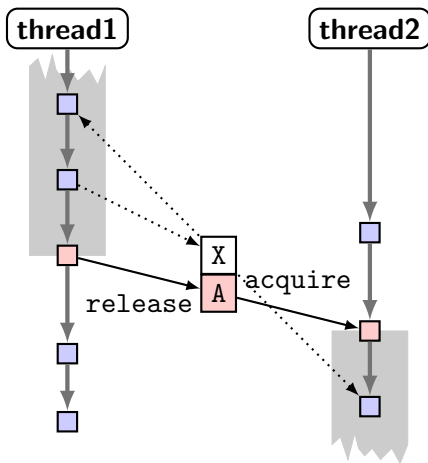
- No global sequence of operations
- Synchronization per variable
- Concurrent operations are possible

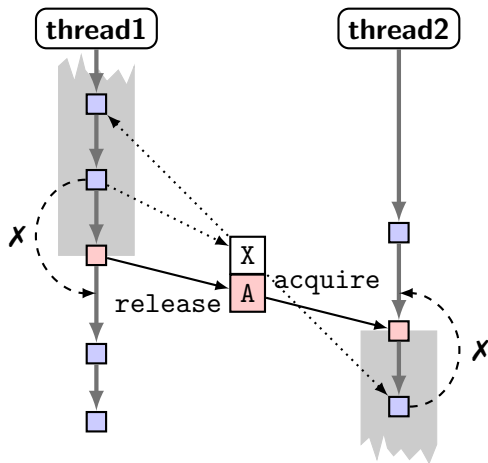


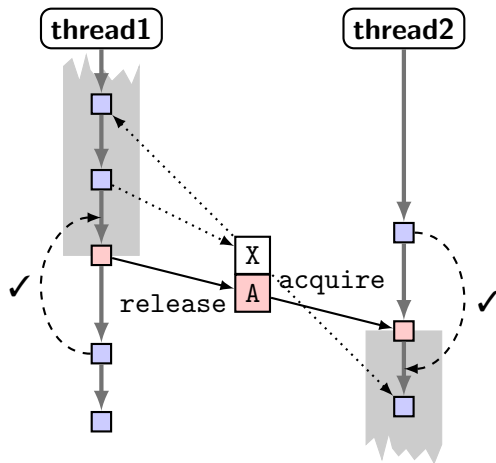
- `acquire` → `load`
 - *fetch-all-data*
 - Side effects after `load-acquire` remain afterwards
 - Implicit in `thrd_join`, `mtx_lock`
- `release` → `store`
 - *push-all-data*
 - Side effects before `store-release` remain before
 - Implicit in `thrd_exit`, `mtx_unlock`
- `acq_rel` → `load/store`
 - e.g. `atomic_fetch_and_*`, `atomic_compare_exchange_*`

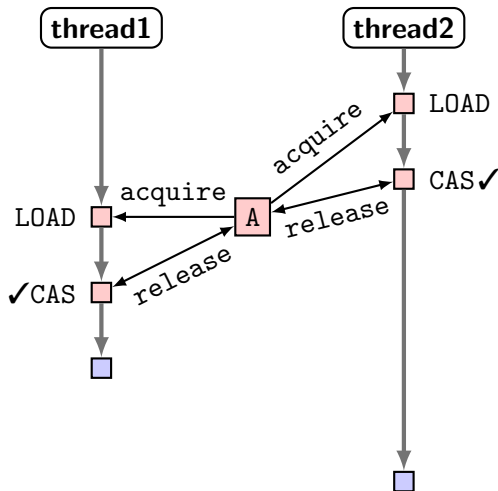


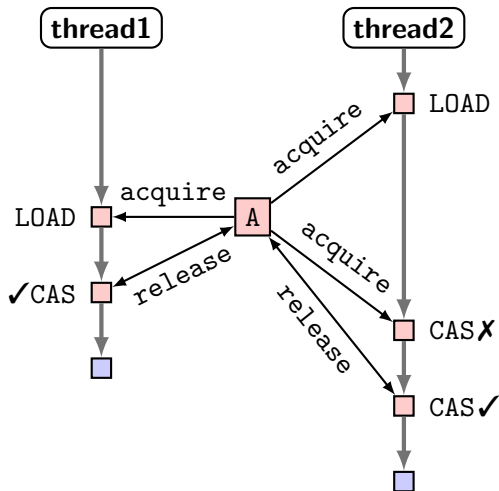












■ Example:

```
1  int data; _Atomic(bool) avail;
2  thread1() {
3      data = createdata(); // not atomic
4      atomic_store_explicit(&avail, 1,
5          memory_order_release);
6  }
7  thread2() {
8      if (atomic_load_explicit(&avail,
9          memory_order_acquire)) {
10         int mycopy = data; use(mycopy);
11     }
12 }
```



■ Example:

```
1  _Atomic(foo *) data;
2  thread1() {
3      foo *d = createdata(); // not atomic
4      atomic_store_explicit(&data, d,
5          memory_order_release);
6  }
7  thread2() {
8      foo *d = atomic_load_explicit(&data,
9          memory_order_acquire);
10     if (d)
11         use(d);
12 }
```



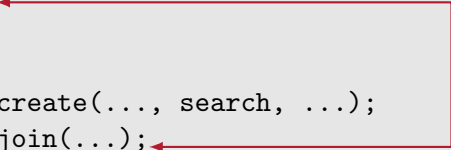
- Weakest memory order parameter
 - Meaningful when additional mechanisms synchronize
- Ensures atomicity
 - No synchronization
 - Lock-freedom not guaranteed
- No reordering constraints
 - Except for thread fences ...
 - Dangerous to use
- Performance improvement?



```
1  _Atomic(unsigned int) counter;
2  search() {
3      while (...) {
4          if (...)
5              atomic_fetch_add_explicit(&counter, 1,
6              memory_order_relaxed);
7      }
8      thrd_exit(...);
9  }
10 main() {
11     for (...) thrd_create(..., search, ...);
12     for (...) thrd_join(...);
13     unsigned int total = atomic_load_explicit(&counter,
14     memory_order_relaxed);
15 }
```

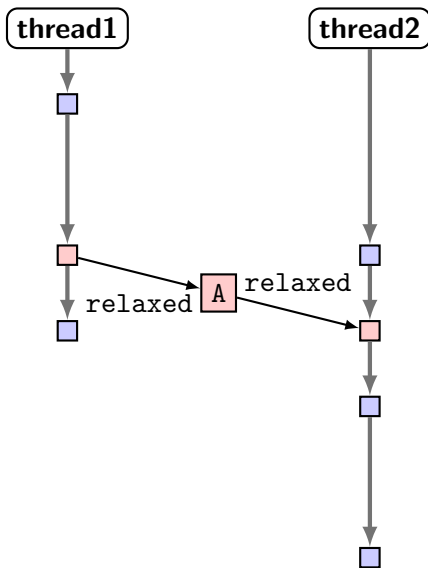


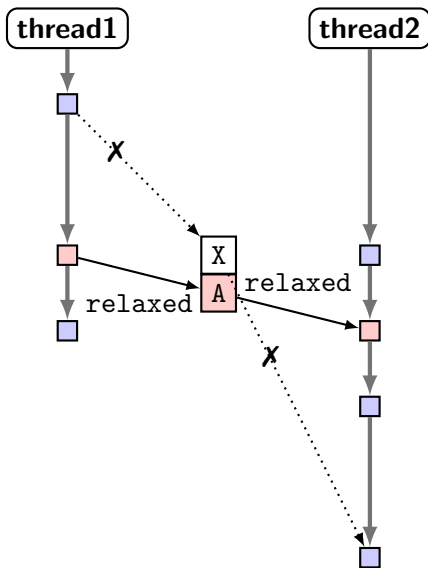
```
1  _Atomic(unsigned int) counter;
2  search() {
3      while (...) {
4          if (...)
5              atomic_fetch_add_explicit(&counter, 1,
6              memory_order_relaxed);
7      }
8      thrd_exit(...);
9  }
10 main() {
11     for (...) thrd_create(..., search, ...);
12     for (...) thrd_join(...);
13     unsigned int total = atomic_load_explicit(&counter,
14     memory_order_relaxed);
15 }
```

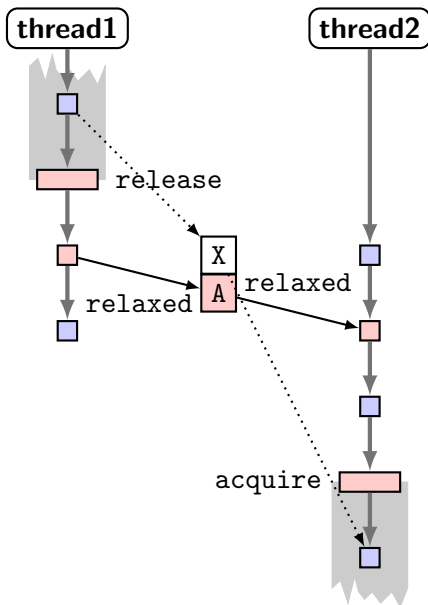


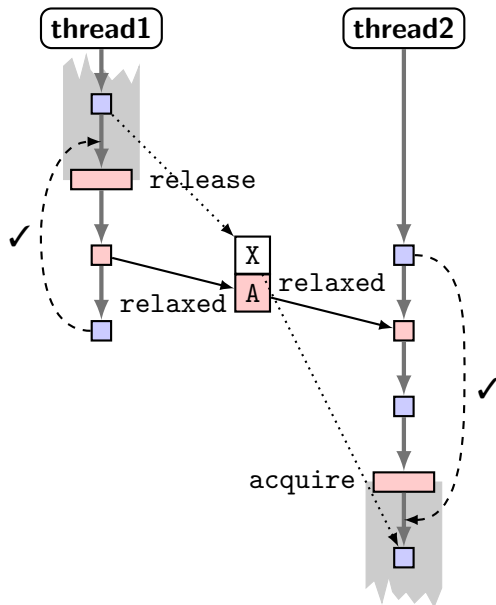
- Enforces additional memory ordering guarantees
 - Often better: use stronger memory order parameter at critical operation
- Parameter: memory order
 - memory_order_acquire
 - memory_order_release
 - memory_order_acq_rel
 - memory_order_seq_cst
- Improves consistency of relaxed-atomic operations
 - Useful for library functions

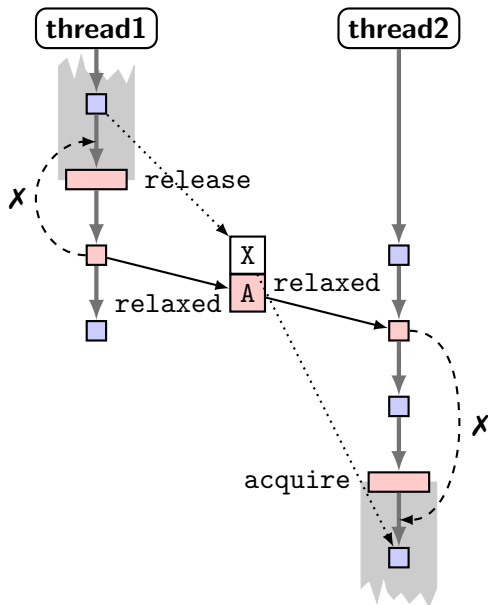












```
1  int data; atomic_bool avail;
2  thread1() {
3      data = createdata(); // not atomic
4
5      atomic_store_explicit(&avail, 1,
6          memory_order_relaxed);
7  }
8  thread2() {
9      if (atomic_load_explicit(&avail,
10         memory_order_relaxed)) {
11
12         int mydata = data; use(mydata);
13     }
14 }
```



```
1  int data; atomic_bool avail;
2  thread1() {
3      data = createdata(); // not atomic
4      atomic_thread_fence(memory_order_release);
5      atomic_store_explicit(&avail, 1,
6          memory_order_relaxed);
7  }
8  thread2() {
9      if (atomic_load_explicit(&avail,
10         memory_order_relaxed)) {
11         atomic_thread_fence(memory_order_acquire);
12         int mydata = data; use(mydata);
13     }
14 }
```



Assignment 3

- 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 sequentialized 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 synchronization
- Under which conditions are actors a better choice?



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

- Example:

```
1 int n, o = atomic_load(&shared);  
2 do {  
3     n = compute(o);  
4     if (atomic_compare_exchange_weak(&shared, &o, n))  
5         break;  
6  
7 } while (1);
```



- *weak* $\Rightarrow$ Operation may fail spuriously
- Efficient implementation with LL/SC possible
- maybe ABA-free
- Example:

```
1 int n, o = LL(&shared);
2 do {
3     n = compute(o);
4     if (SC(&shared, n))
5         break;
6     o = LL(&shared);
7 } while (1);
```

