Memory Models

Java and C++

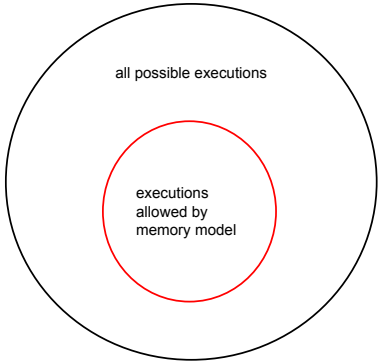
Jingwei Xu Ran Pang

http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf
http://en.cppreference.com/w/cpp/atomic/memory_order

Outline

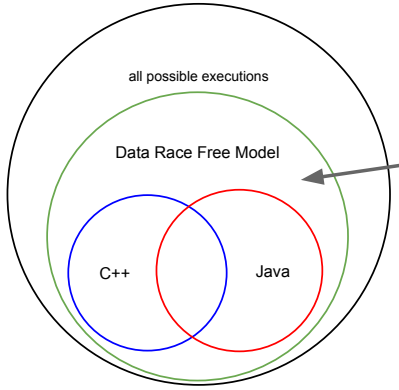
- Data Race Free Model
- Happens-Before Model
- Java Memory Model
- C++ memory ordering
- How to implement synchronization in C++

What does a memory model do?



- Programmer
  Easy to use
- Compiler & Hardware
  Optimizations

Java vs. C++ Memory Models



Guarantee **sequential consistency** for **data race** free programs

Data Race Free Model

Sequential Consistency:

Multi-threaded program run as if multi-programs on uniprocessor

Data Race:

If a program is executed in a sequentially consistent way

- two or more threads access the same memory location concurrently
- at least one of the accesses is for writing
- the threads are not using any exclusive locks to control their accesses to that memory

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 1;	x = 1;

In any sequentially consistent execution
r1 == r2 == 0
thus no write of x nor y
thus no data race

A possible sequential consistent execution:

1. read(x, 0)
2. write(r1, 0)
3. read(y, 0)
4. write(r2, 0)
5. read(r1, 0)
6. read(r2, 0)

Happens-Before Model

Action A happens-before(→) action B if

- A B in same thread and A occurs before B in the program or
- B is the receipt of the message sent by A or
- A → X, X → B

A read R(x) is allowed to see the result of a write W(x) if

- W(x) is the last write to x before R (x) along the path in the happens-before order, or
- W(x) and R(x) has no happens-before relationship

message sending (synchronize) actions

Java:

- Language support
  - lock / unlock
  - read / write volatile
  - thread start / in-thread action
  - in-thread action / thread join

C++11:

- Thread support library
  - mutex
  - lock
  - condition_variable
- Atomic operations library
  - memory_order

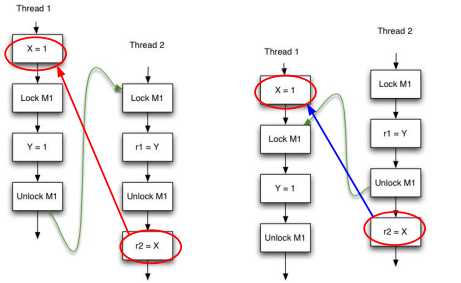
Happens-Before Model

Action A happens-before(→) action B if

- A B in same thread and A occurs before B in the program or
- B is the receipt of the message sent by A or
- A → X, X → B

A read R(x) is allowed to see the result of a write W(x) if

- W(x) is the last write to x before R (x) along the path in the happens-before order, or
- W(x) and R(x) has no happens-before relationship



(a) Thread 1 acquires lock first; Accesses to X are ordered by happens-before

(b) Thread 2 acquires lock first; Accesses to X not ordered by happens-before

Happens-Before Model

Action A happens-before(→) action B if

- A B in same thread and A occurs before B in the program or
- B is the receipt of the message sent by A or
- A → X, X → B

A read R(x) is allowed to see the result of a write W(x) if

- W(x) is the last write to x before R (x) along the path in the happens-before order, or
- W(x) and R(x) has no happens-before relationship

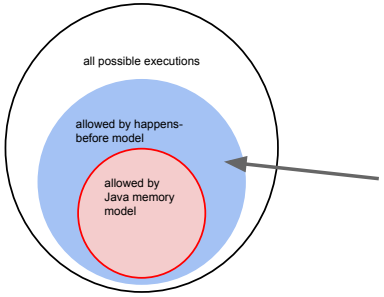
Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 1;	x = 1;

visibility under happens-before model

Happens-Before Model

Happens-Before Model provides necessary but not sufficient constraints.



Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 1;	x = 1;

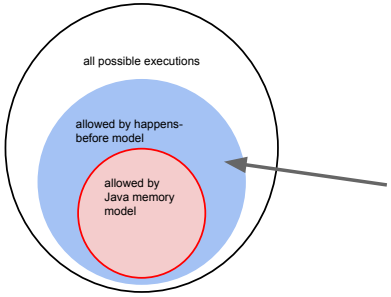
- no data race
- no happens-before relationships between threads
- only allowed result by sequentially consistent execution r1 = r2 = 0

Execution allowed by Happens-Before Model

1. write(y, 1)
2. read(y, 1)
3. write(r2, 1)
4. write(x, 1)
5. read(x, 1)
6. write(r1, 1)
7. read(r1, 1)
8. read(r2, 1)

Happens-Before Model

Happens-Before Model provides necessary but not sufficient constraints.



Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

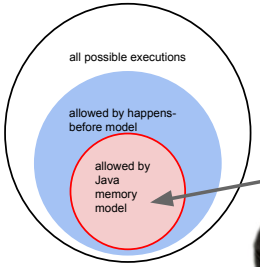
- has data race
- no happens-before relationships between threads

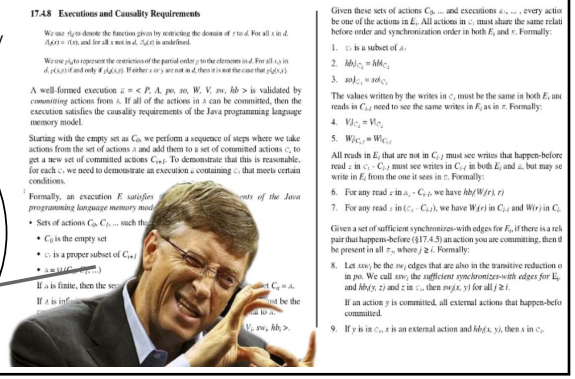
Execution allowed by Happens-Before model:

1. speculated write(r1, 42)
2. speculated write(r2, 42)
3. read(r1, 42)
4. write(y , 42)
5. read(r2, 42)
6. write(x, 42)
7. check(x == r1)
8. check(y == r2)

Java Memory Model

Java Memory Model enforces Happens-Before model in a very complex way





17.4.8 Executions and Causality Requirements

We use $x|_S$ to denote the function given by restricting the domain of x to A. For all x in A, $A(x) = \Omega$, and for all x not in A, $\preceq_A(x)$ is undefined.

We use $\preceq_A|_S$ to represent the restriction of the partial order $\preceq$ to the elements in A. For all x, y in A, $\preceq_A(x,y)$ if and only if $\preceq_A(x,y)$. If either x or y are not in A, then it is not the case that $\preceq_A(x,y)$.

A well-formed execution $E = \langle P, A, po, so, W, V, sw, hb \rangle$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i, we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution E *satisfies* [illegible] *of the Java programming language memory model* if:

- Sets of actions $C_0, C_1, \ldots$ such that:
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \bigcup_i (C_0, C_1, \ldots)$

If A is finite, then the se[illegible] $C_n = A$.

If A is infi[illegible] must be the [illegible] to A.

V_x, sw_x, hb_x >.

Given these sets of actions $C_0, \ldots$ and executions $x_1, \ldots$, every action be one of the actions in E_i. All actions in C_i must share the same relati before order and synchronization order in both E_i and π. Formally:

1. C_i is a subset of A_i
2. $hb|_{C_i} = hb|_{C_i}$
3. $so|_{C_i} = so_{C_i}$

The values written by the writes in C_i must be the same in both E_i and reads in C_{i+1} need to see the same writes in E_i as in π. Formally:

4. $V|_{C_i} = V_{C_i}$
5. $W|_{C_i} = W_{C_i}$

All reads in E_i that are not in C_{i+1} must see writes that happen-before read r in $C_i - C_{i+1}$ must see writes in C_{i+1} in both E_i and π, but may se write in E_i from the one it sees in π. Formally:

6. For any read r in $A_i - C_{i+1}$, we have $hb(W(r), r)$
7. For any read r in $(C_i - C_{i+1})$, we have $W(r)$ in C_{i+1} and $W(r)$ in C_i

Given a set of sufficient synchronizers-with edges for E_0, if there is a rel pair that happen-before (§17.4.5) an action you are committing, then t be present in all π_i, where $j \geq i$. Formally:

8. Let sw_i be the sw_i edges that are also in the transitive reduction o in po. We call sw_i the *sufficient synchronizes-with edges for* E_0 and $hb(y, z)$ and z in C_i, then $sw_i(x, y)$ for all $j \geq i$.

If an action y is committed, all external actions that happen-befo committed.

9. If y is in C_i, x is an external action and $hb(x, y)$, then x in C_i.

Java Memory Model

- Guarantee Sequential Consistency for Data Race Free Programs
- Based on Happens-Before Model
- Deal with Data Race
- Language Support Synchronize Actions
  - synchronized()
  - monitor lock / unlock
  - volatile read / write
  - thread start() / in-thread action
  - in-thread action / thread join()
- Other features
  - Forbid word tearing
  - Implement the final variable
  - Non-atomic 64-bit variable
  - volatile or other synchronize encouraged

```
class Counter{

    long count = 0;

    synchronized void add(long value){ // synchronization
      this.count += value;
    }
  }

class CounterThread extends Thread{

    Counter counter = null;

    CounterThread(Counter counter){
      this.counter = counter;
    }

    public void run() { // thread function
        for(int i=0; i<10; i++)
          counter.add(i);
    }
  }

class Example {

    public static void main(String[] args){
      Counter counter = new Counter();

      Thread  threadA = new CounterThread(counter); // thread creatio
      Thread  threadB = new CounterThread(counter);

      threadA.start();
      threadB.start();
    }
  }
```

C++11 Memory Model

Thread support library: (similar with Pthreads)

- **Thread:** manage a separate thread
  - thread
- **Mutual exclusion:** prevent multiple threads from simultaneously accessing shared resources
  - mutax
  - lock
- **Condition variables:** allows some number of threads to wait (possibly with a timeout) for notification from another thread that they may proceed
  - condition_variable

C++11 Memory Model

Atomic operations library : support lock free algorithm

memory_order : defines memory ordering constraints for the given atomic operation

Four types of ordering:

- **weak ordering (Default)**
  - memory_order_seq_cst
- **Release-Acquire Ordering**
  - memory_order_acquire
  - memore_order_release
- **Release-Consume Ordering**
  - memory_order_consume
  - memore_order_release
- **Relaxed Ordering**
  - memory_order_relaxed

Memory ordering principle of in C++

Memory Ordering: (How operations are ordered between different threads)

- **Sequenced-before :**
  - Carries dependency : A carries a dependency into B
    - A writes to M, B read from M
- **Modification Order :** All modifications to any particular atomic variable occur in a total order that is specific to this one atomic variable
  - Write-Write; Read-Read; Read-Write; Write-Read
- **Release sequence :** After a release operation A on M
  - Writes performed by the same thread that performed A
  - Atomic read-modify-write operations made to M by any thread

Memory ordering principle of in C++

Memory Ordering: (what we should know to write lock free program)

- **Dependency-ordered before :** Between threads, evaluation A is *dependency-ordered before* evaluation B
  - release operation A on M, consume operation B on M
- **Inter-thread happens before :**
  - synchronized with
  - dependency-ordered before
- **Happens before :**
  - sequenced-before
  - inter-thread happens before

C++ Memory Model

Sequentially-Consistent Ordering(weak ordering):

- Obey the happens before ordering
- Establish a single total modification order of all atomic operations that are so tagged

```
void write_x()
    x.store(true, std::
memory_order_seq_cst);

void write_y()
    y.store(true, std::
memory_order_seq_cst);

void read_x_then_y()
    while (!x.load(std::
memory_order_seq_cst))
    if (y.load(std::memory_order_seq_cst))
        ++z;

void read_y_then_x()
    while (!y.load(std::
memory_order_seq_cst))
    if (x.load(std::memory_order_seq_cst))
        ++z;
```

```
int main()
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join(); b.join(); c.join(); d.join();
    assert(z.load() != 0);  // will never happen
```

Release-Acquire Ordering

prior writes made to other memory locations by the thread that did the release become visible in this thread

```
producer(){
    std::string* p = new std::string("hello");   //A
    data = 42;     //B
    ptr.store(p,std::memory_order_release);      //C
}
consumer(){
    std::string* p2;
    while(!(p2 = ptr.load(std::memory_order_acquire)));    //D
    assert(*p2 == "hello");       //E
    assert(data == 42);           //F
}
```

Release-Consume Ordering

prior writes to data-dependent memory locations made by the thread that did a release operation become visible to this thread's dependency chain

```
producer(){
    std::string* p = new std::string("hello");   //A
    data = 42;     //B
    ptr.store(p,std::memory_order_release);      //C
}
consumer(){
    std::string* p2;
    while(!(p2 = ptr.load(std::memory_order_consume)));    //D
    assert(*p2 == "hello");       //E
    assert(data == 42);           //F
}
```

C++ Memory Model

Relaxed Ordering: only guarantee atomicity and modification order consistency.

```
Thread1()
    r1 = y.load(memory_order_relaxed);      // A
    x.store(r1, memory_order_relaxed);      // B
Thread2()
    r2 = x.load(memory_order_relaxed);      // C
    y.store(42, memory_order_relaxed);      // D

r1 == r2 == 42 is allowed.
```

```
std::atomic<int> num = {0};

void f()
{
    for (int n = 0; n < 1000; ++n) {
        num.fetch_add(1, std::memory_order_relaxed);
    }
}
int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(f);
    }
    for (auto& t : v) {
        t.join();
    }
}
```

atomic<int> num will become 10000.

C++ Memory Model

Relationship with volatile

- **Volatile :**
  - Volatile access orders are only guaranteed within the thread
  - Volatile accesses are not atomic
- **Exception :** Visual Studio
  - release write & require read

Java Memory Model

- Guarantee sequential consistency for data race free programs
- Amendment on Happens-Before Model
- Language support synchronize actions
  - synchronized()
  - monitor lock / unlock
  - volatile read / write
  - thread start() / in-thread action
  - in-thread action / thread join()
- Other features
  - Forbid word tearing
  - Implement the final variable
  - Non-atomic 64-bit variable
  - volatile or other synchronize encouraged

C++ Memory Model

- Guarantee sequential consistency for data race free programs
- Thread support library
  - Thread support library
  - Atomic operations library
- Atomic operations library (support lock-free algorithms)
  - Sequential consistency ordering
  - Release-Acquire ordering
  - Release-Consume ordering
  - Relaxed ordering

Questions?