COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 32: Volatile variables and Java memory model

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

- Combined handout for Lectures 27-32 (to be updated)
- "Introduction to Concurrent Programming in Java", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - —Contributing authors: Doug Lea, Brian Goetz
  - —Contributing authors: Doug Lea, Tim Peierls, Brian Goetz
- "Engineering Fine-Grained Parallelism Support for Java 7", Doug Lea, July 2010
- "Java Concurrency in Practice", Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea. Addison-Wesley, 2006.

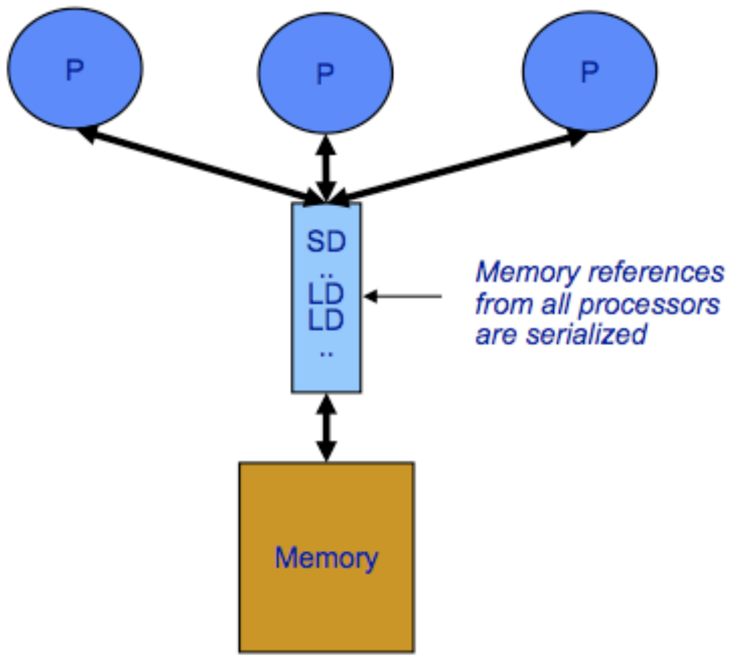
Program Order != Reality

- Programmers view:
  - Everything happens in the order I indicate through the code statements that I write
- Reality (JVM/compiler & processor):
  - Everything happens in whatever order yields best performance, so long as the program(mer) can't tell the difference
- For single-threaded systems:
  - Program order can't be distinguished from actual order
- For multi-threaded systems:
  - Without correct use of synchronization different threads can see different actions in memory
    - At different times
    - In different orders
- The Memory Model defines the rules

Memory Models

- A memory consistency model, or *memory model*, is the part of a programming language specification that defines what write values a read may see in the presence of data races.
- We will briefly discuss three memory models
  - Sequential Consistency (SC)
  - Weak Ordering (WO)
  - Java Memory Model (JMM)

Sequential Consistency (Lecture 6)

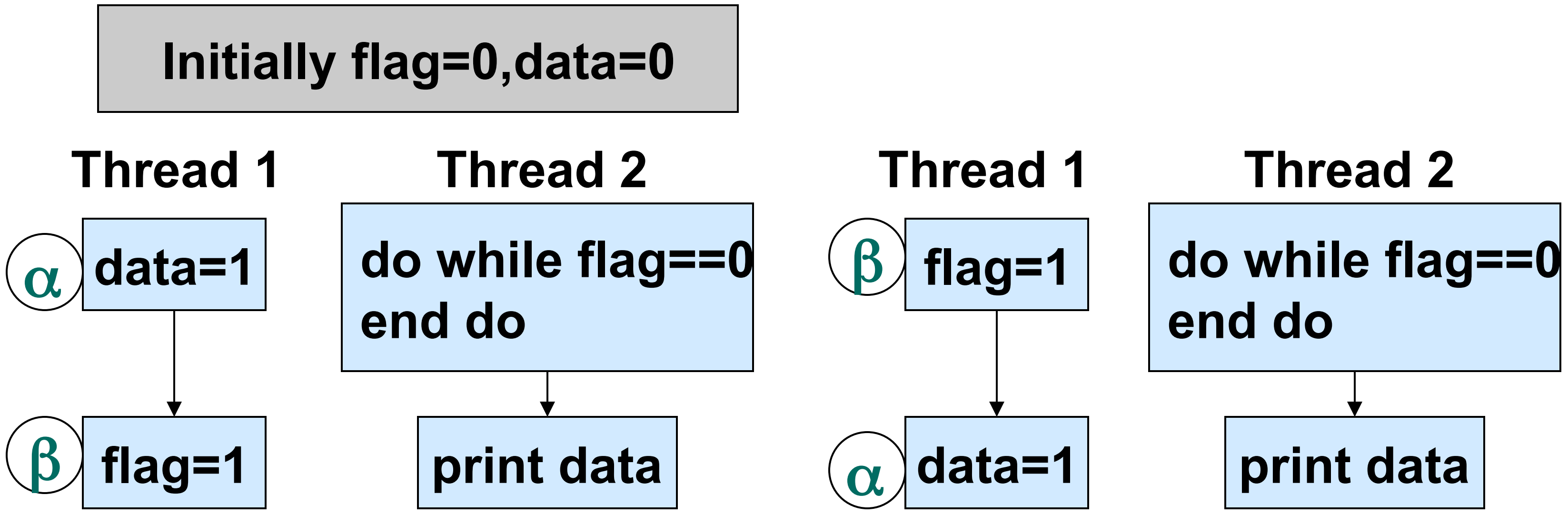


[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

Sequential Consistency

- SC constrains all memory operations:
    - Write → Read
    - Write → Write
    - Read → Read, Write
- Simple model for reasoning about parallel programs
- But, intuitively reasonable reordering of memory operations in a uniprocessor may violate sequential consistency model
    - Modern microprocessors reorder operations all the time to obtain performance e.g., write buffers, overlapped writes,non-blocking reads…
    - Optimizing compilers perform code transformations that have the effect of reordering memory operations e.g., scalar replacement, register allocation, instruction scheduling, …
    - A programmer may perform similar code transformations for software engineering reasons without realizing that they are changing the program's semantics

Rolling your own Synchronization Primitives using Sequential Consistency



Print Example (Lecture 6)

- SC model will not permit Task T3 to print “0, 1, 2” and Task T4 to print “0, 2, 1”

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p.x);
}
 async { // Task T4
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x);
}
```

Print Example (contd)

- What if the programmer transformed the body of Task T3?

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
   int p_x = p.x;
  System.out.println("First read = " + p_x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p_x);
}
 async { // Task T4
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x);
}
```

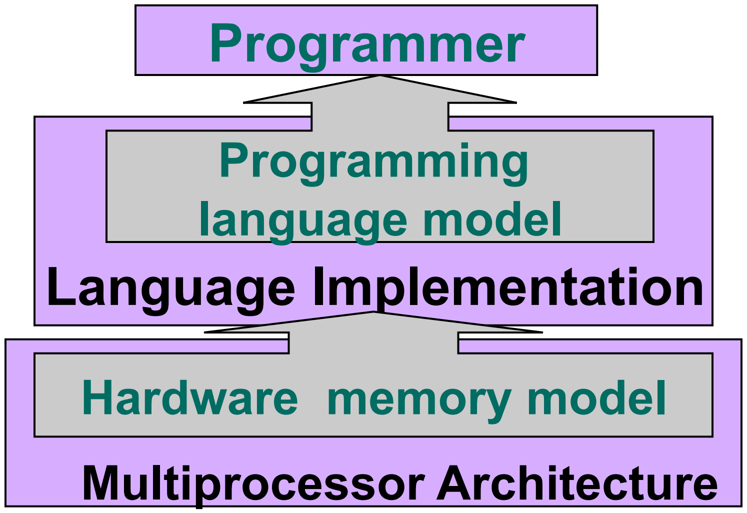
Weak Ordering

- Weak ordering:
  - Divide memory operations into data operations and synchronization operations
  - Synchronization operations act like a "fence":
    - All data operations before synch in program order must complete before synch is executed
    - All data operations after synch in program order must wait for synch to complete
    - Synchs are performed in program order
  - Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed

But all these examples have races

- What if the programmers properly synchronizes the application to avoid races?
  - Program will have same semantics under all memory models!
  - Programmer and implementation can assume a weak consistency model (which will be equivalent to SC for data-race-free programs)

The Language Implementation's task



- **Compiler must enforce programming language memory model**
  - **Hardware and software model often differ**
  - **Compiler may need to insert *fences* to make hardware model stricter**

Volatile Variables

- Java provides a "light" form of synchronization/fence operations in the form of `volatile` variables
- Volatile variables guarantee *visibility*
  - An access of a volatile variable is like an access of a synchronization variable in the Weak Ordering model
  - Adds serialization edges to computation graph due to isolated read/write operations
- Incrementing a volatile variable (++v) is <u>not thread-safe</u>
  - Increment operation looks atomic, but isn't (read and write are two separate operations)
- Volatile variables are best suited for flags that have no dependencies

```
–volatile boolean asleep
–while (! asleep)
    ++sheep;
```

•Warning: a volatile declaration on an array variable may not give you the semantics you expect

The Java Memory Model (JMM)

- Conceptually simple:
  - Every time a variable is written, the value so written is added to the set of all values the variable has had
  - A read of a variable is allowed to return ANY value from the set of written values
- The JMM defines the rules by which values in the set are removed
  - Synchronization actions and happens-before relationship
- Programmers goal: through proper use of synchronization
  - Ensure the written set consists of only one value—that most recently written by some thread
- Basic safety guarantee: No "out-of-thin-air" values
  - A read always returns a value written by some thread, some time
  - Reads and writes of all basic data types are atomic

    Except for `long` and `double`

Synchronization Actions

- Program order: The order in which statements appear in a program, as executed by a single thread
  - Continue edges in a computation graph
- Synchronization order: The order in which synchronization actions are executed
  - Always consistent with program order
  - Determines a partial order across actions in different threads
  - Spawn, join, and serialization edges in a computation graph
- Example synchronization actions:
  - Starting a thread; joining on a thread
  - Acquiring a lock; releasing a lock
- Some actions synchronize-with another action. Eg:
  - Starting a thread synchronizes-with the first action in that thread
  - Releasing a lock synchronizes-with all subsequent acquires of that lock

Happens-Before Relationship

- Formal relationship between reads and writes of variables
  - Controls the possible values that a read of a variable may return
- For a given variable:
  - If a write of the value v1 happens-before the write of a value v2, and the write of v2 happens-before a read, then that read may not return v1
  - Properly ordered reads and writes ensure a read can only return the most recently written value
- If an action A synchronizes-with an action B then A happens-before B
  - So correct use of synchronization ensures a read can only return the most recently written value

Example: Unsynchronized Data Holder

```
-class DataHolder {
  int data = 0;
  boolean dataReady = false;
  boolean isReady() { return dataReady; }
  void setData(int newData) {
    data = newData;
    dataReady = true;
  }
  int getData() { return data; }
}
              DataHolder h = ...;

// ... Thread-1                     // Thread-2
while (!h.isReady())               h.setData(42);
   doOtherWork();
int goodData = getData();
```

- No synchronization actions between threads, so no happens-before
  - Thread-1 need never have `isReady()` return true
  - Even if `isReady()` returns true, `getData()` may return 0 not 42

Example: Synchronized Data Holder

```
-class SyncDataHolder {
  int data = 0;
  boolean dataReady = false;
  synchronized boolean isReady() {
    return dataReady; }
  synchronized void setData(int newData) {
    data = newData;
    dataReady = true;
  }
  synchronized int getData() { return data; }
}
            SyncDataHolder h = ...;

// ... Thread-1                      // Thread-2
while (!h.isReady())               h.setData(42);
   doOtherWork();
int goodData = getData();
```

- Synchronization ensures if `isReady` is true then `getData` returns 42
  - Note: synchronization on `getData` not needed <u>iff</u> `isReady` always invoked first

`volatile` Variables

- Reads/writes of variables declared `volatile` are synchronization actions
  - A write to a `volatile` variable synchronizes-with all subsequent reads of that `volatile` variable
  - So `volatile` variables provide ordering and visibility guarantees for themselves and any data they "protect"
- Reads/writes of variables declared `volatile` are also atomic
  - Extends the basic atomicity guarantee from the 32-bit types to the 64-bit types: `long` and `double`
  - NOTE: compound actions are NOT atomic, Eg:

    ++ requires: read, increment, write

Example: `volatile` Data Holder

```
-class VolatileDataHolder {
  volatile int data = 0;
  volatile boolean dataReady = false;
  boolean isReady() { return dataReady; }
  void setData(int newData) {
    data = newData;
    dataReady = true;
  }
  int getData() { return data; }
}
           VolatileDataHolder h = ...;

// ... Thread-1                      // Thread-2
while (!h.isReady())               h.setData(42);
   doOtherWork();
int goodData = getData();
```

- Use of `volatile` ensures if `isReady` is true then `getData` returns 42
  - Note: `data` field need not be `volatile` iff `isReady` always invoked first AND `dataReady` field written after `data` field