FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 2 – SUPPORT PRESENTATION (PART II)

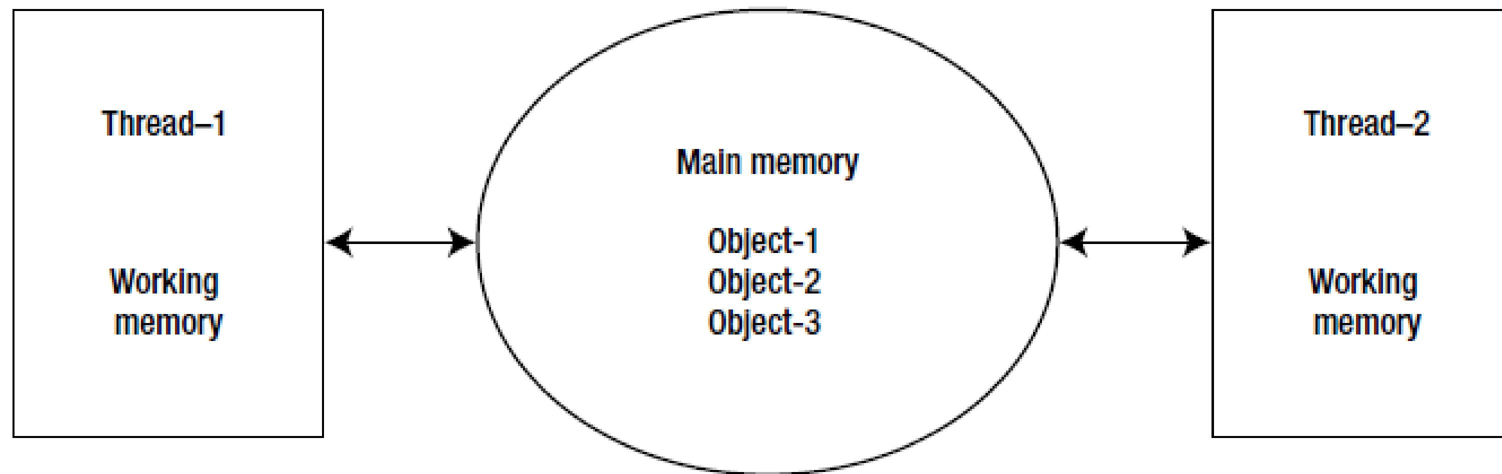
Main bibliographic sources

- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea, Java Concurrency in Practice, Addison Wesley, Pearson Education
- K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams 1st Edition, APRESS, 2014.

Agenda

- Java Memory Model
- Volatile Variables
- Synchronized Statements
- Synchronized Methods
- Atomicity
- Synchronized Collections
- Concurrent Collections

Java Memory Model



Atomicity

Visibility

Ordering

Volatile Variables

The Java volatile keyword guarantees visibility of changes to variables across threads.

For every read request for a volatile variable, a thread reads the value from the main memory.

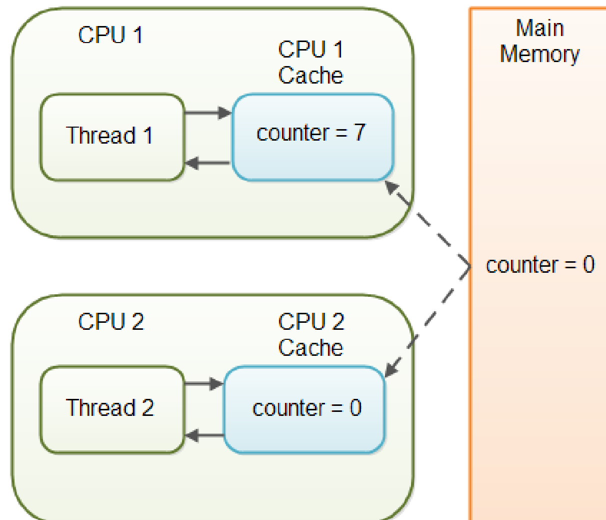
For every write request for a volatile variable, a thread writes the value to the main memory.

- A thread does not cache the value of a volatile variable in its working memory.
- Using a volatile variable is useful only in a multi-threaded environment for variables that are shared among threads.
- It is faster and cheaper than using a synchronized block.

Volatile Variables

Not using volatile variables

```
public class SharedObject {
    public int counter = 0;
}
```



“Visibility” Problem!

Source

Using volatile variables

```
public class SharedObject {
    public volatile int counter = 0;
}
```

If thread T1 modifies the counter, and thread T2 reads the counter (but never modifies it), declaring the counter variable volatile is enough to guarantee visibility for T2 of writes to the counter variable.

If both T1 and T2 were incrementing the counter variable, then declaring the counter variable volatile would not have been enough

Volatile Variables

- If two threads are **both reading and writing** to a shared variable, then using the volatile keyword for that is not enough!
  - Solutions
    1) Use the **synchronized keyword around critical sections** to guarantee that the reading and the writing of the volatile variable is atomic
    2) Use **atomic data types** found in **java.util.concurrent package** (e.g., **AtomicLong**, **AtomicReference**)
- Performance considerations
  - Reading and writing of volatile variables causes the variable to be read or written to main memory
  - *Reading from and writing to main memory is more expensive than accessing the CPU cache*

Use volatile variables when you really need to enforce visibility of variables!

Source

Synchronized Statements

- To preserve state consistency, update state variables in a single atomic operation!

Block of code to be guarded by the lock

```
synchronized (lock) {//Reference to an object
    // Access or modify shared state guarded by lock
    …
}
```

- Every Java object can implicitly act as a lock for purposes of synchronization
  - ***intrinsic locks*** or ***monitor locks***
  - Automatically acquired and released
  - A happens-before relationship is established

Synchronized Statements

- Examples of synchronized blocks

```
synchronized(MyClass.class){
   // some code
 }
Or
synchronized(this){
   // some code
 }
```

Locks can be used to create synchronized code

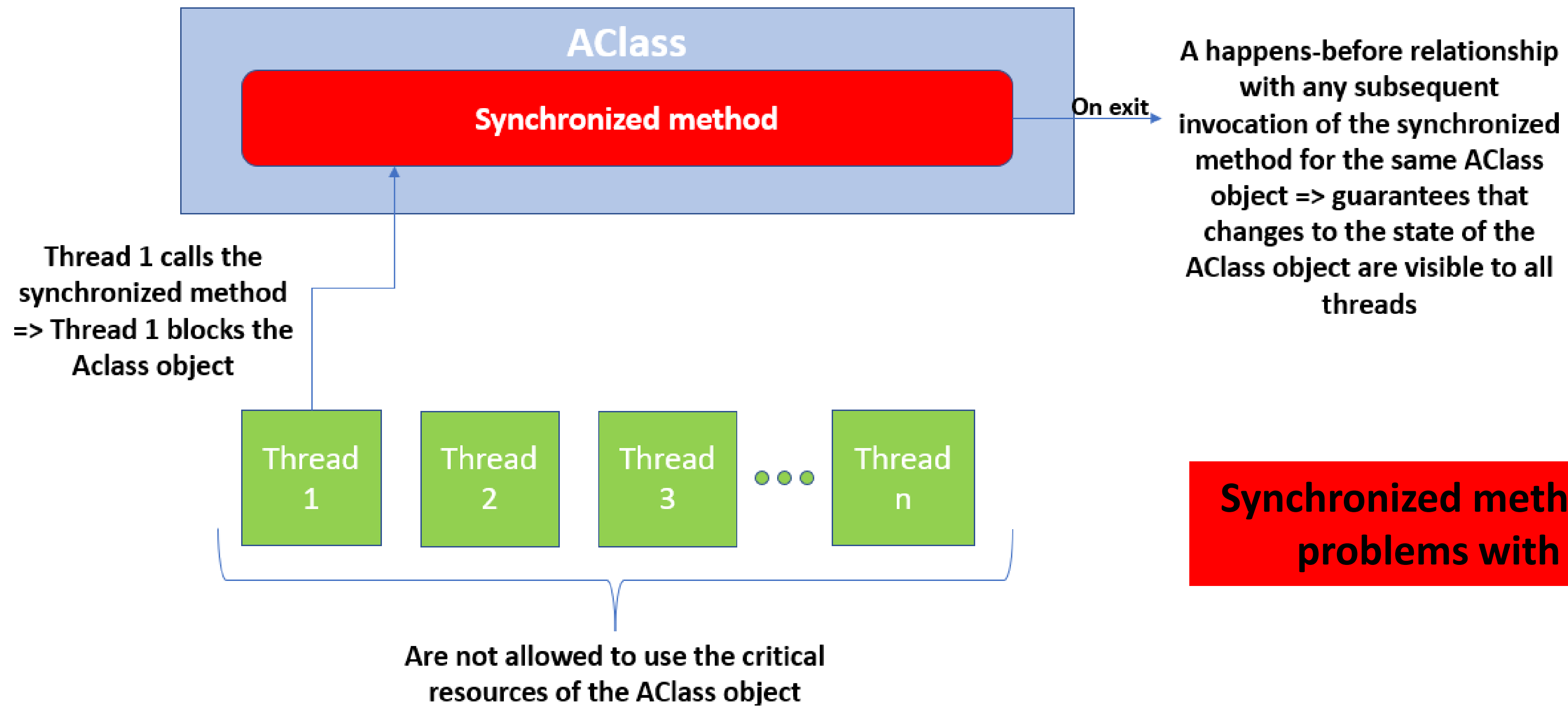
```
public void transfer(Account a,
             Account b, double sum){
  synchronized(a){// Th1 locks acc. a
     //Th2 locks account b
    synchronized(b){
    //transfer sum
    }
  }
}
…
Call function :
Th1: transfer(a,b,sum1);
Th2: transfer(b,a,sum1);
```

Deadlock situation!!!

Locks can lead to deadlocks

Synchronized Methods

- **synchronized** keyword in the methods' declaration
- **Constructors cannot be synchronized**



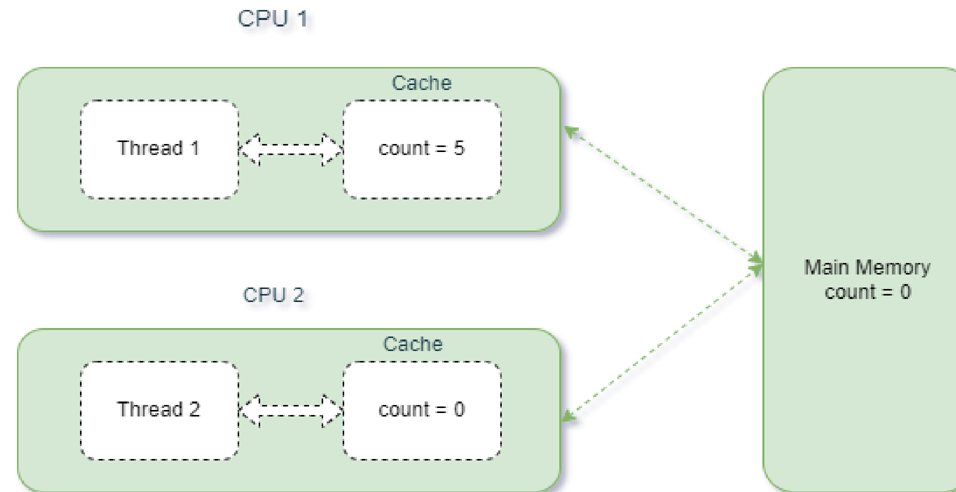
Atomicity

- Atomic action cannot be interleaved => avoids thread interference
- java.util.concurrent.atomic package

Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).

Reads and writes are atomic for all variables declared **volatile** (including long and double variables).

Working with memory doesn't happen instantly



Source

Atomicity

Compound Operations

```
int i=0;
i++; // Get I value & add one to it
/*Accessed simultaneously by both Th1 and Th2
Can lead to inconsistencies:
-  result can be 1(both threads got 0 and
   incremented to 1)
-  result can be 2(second thread got the value
   1 incremented by the first thread)
*/
```

i++ is not atomic!

- *read-modify-write* operation
- not stateless and is not thread-safe due to instance variable

Race conditions

Atomic Operations

```
AtomicInteger i= new AtomicInteger();
i.getAndIncrement ();
```

Volatile variables

- Changes are always visible to other threads
- Establishes a happens-before relationship with subsequent reads of that same variable
- Sees also the side effects of the code that led up the change

Synchronized Collections

- Synchronization wrappers which create synchronized views of collections
  - *syncronizedCollection*, *synchronizedList*, *synchronizedMap*, etc.

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
```

  - Achieve thread-safety through intrinsic locks
  - Synchronized collections are thread safe

Must manually synchronize on the returned collection when iterating over it

```
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized (c) {
  Iterator i = c.iterator(); // Must be in the
                             // synchronized block
  while (i.hasNext()) foo(i.next());
}
```

Concurrent Collections

- Designed for concurrent accesses from multiple threads
  - *java.util.concurrent package: BlockingQueue, ConcurrentHashMap, ConcurrentNavigableMap, CopyOnWriteArrayList*
- Achieve thread-safety
  - **BlockingQueue** – provides blocking put and take methods
    - Support the producer-consumer design patterns
  - **ConcurrentHashMap** - divides its data into segments
    - Different threads can acquire locks on each segment
    - Multiple threads can access the map at the same time
  - **CopyOnWriteArrayList** - creates a separate copy of List for each write operation

Are much more performant than synchronized collections