Concurrency Without Locks

GURUTEAM
specialist ICT learning

Review - Basic Data Access Synchronization

- Every Java object has a lock
  - guarantees mutual exclusion
- Basis of protecting shared data
  - `synchronized` method acquires lock before proceeding
  - only one thread in any `synchronized` method of object at a time

```
...
public synchronized void makeDeposit ( int amount )
{
  int curBal = getBalance();
  curBal += amount;
  setBalance( curBal );
}
...
```

Synchronization and the Java Memory Model

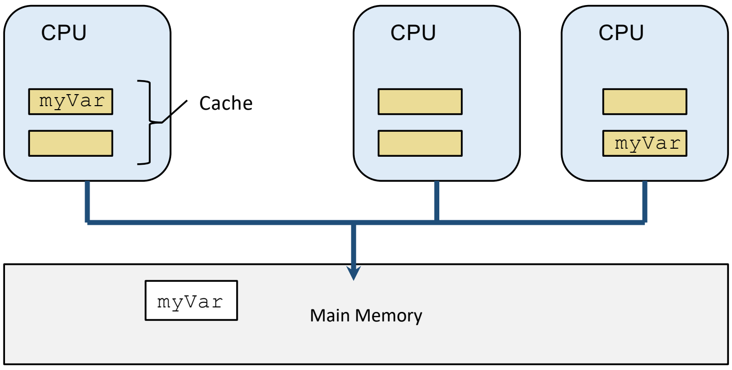
- A set of rules that describe the behaviour of read/write operations with respect to memory
  - if thread 1 executes

```
myVar = 42;
```

  - when does thread 2 see the value in the variable
  - does it ever?
- Memory model needs to be defined rigorously
  - shared memory multiprocessor machines with per-CPU caches
  - compiler introduced optimisations such as register storage and instruction reordering
- Java synchronisation mechanisms work safely because of the Memory Model

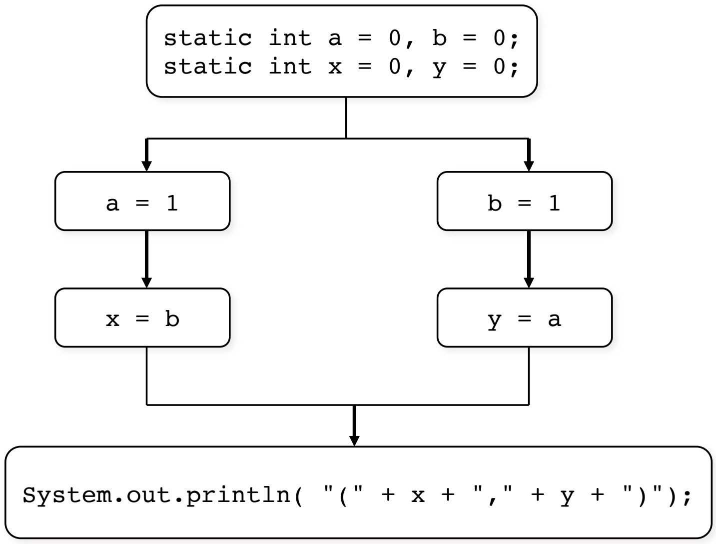
Why Do We Need a Memory Model?

- Multiple CPUs (cores) sharing memory
- Cached copies of shared data must be kept up to date



Why Do We Need a Memory Model?

- What gets printed?
  - (1,1)
  - (1,0)
  - (0,1)
  - (0,0) ???

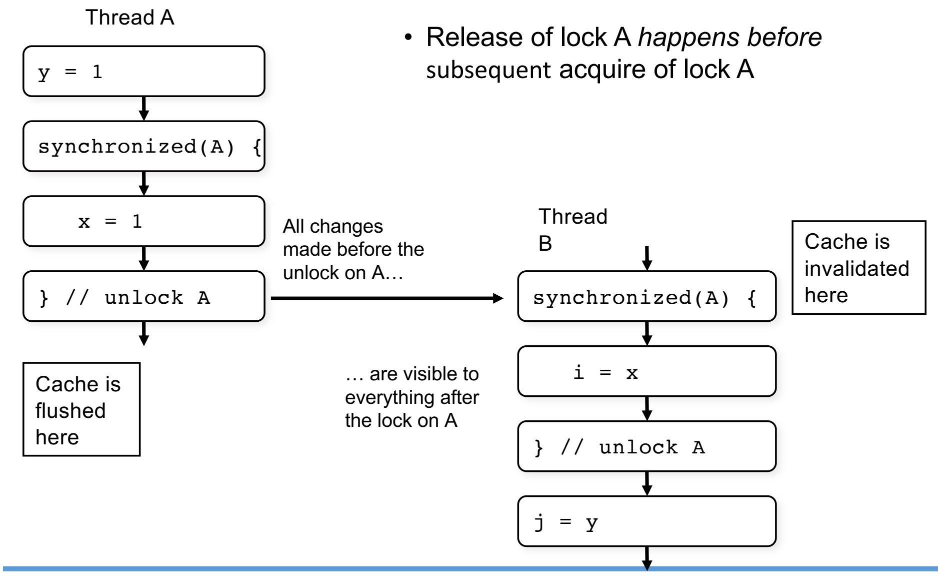


Synchronized is About More than Locks

- Java Memory Model defines a "partial ordering" on memory operations within a program
  - "Happens-before"
- If action A "Happens-before" action B then the results of A will be visible in B
  - even if the actions occur in different threads
- If not, then action A and action B may be reordered
- Synchronization introduces a "Happens-before" relationship
  - so does volatile
  - other scenarios as well

Synchronized and Happens-Before

- Release of lock A *happens before* subsequent acquire of lock A



About `volatile`

- `volatile` keyword signals compiler and JVM that a variable is shared
    - no caching in registers
    - cache flushes always follow writes
- Also prevents memory reordering
    - reads and writes to volatile variable cannot be ordered with respect to each other or to non-volatile variables
    - ensures changes made are visible at the right time to other threads

Thread A

```
volatile boolean finished = false;
…
while ( !finished ) {
  doMyWork();
}
```

Thread B

```
doStuff();
finished = true;
```

Initialization and Publication

- Publication is making an object visible outside the current thread
  - care needed to ensure other threads see consistent object state
- Particularly important during object initialization

```
// This is unsafe because there is no synchronization!!
public class ExpensiveResource {
  private static ExpensiveResource resource;

  // Lazy initialization - defer creation and initialization
  // until the resource is actually needed
  public static ExpensiveResource getInstance() {
    if ( resource == null ) {
      resource = new ExpensiveResource();
    }
    return resource;
  }
}
```


Thread-safe Lazy Initialization

- Synchronized keyword introduces happens-before relationship
  - most calls to method will result in uncontended synchronization since code path is short
  - possible performance degradation

```
public class ExpensiveResource {
  private static ExpensiveResource resource;

  public synchronized static ExpensiveResource getInstance() {
    if ( resource == null ) {
      resource = new ExpensiveResource();
    }
    return resource;
  }
}
```

Eager Initialization

- Alternative, thread-safe approach to initialization
- Static initialization does not require explicit synchronization
  - JVM uses internal lock to manage class loading
  - results in happens-before behaviour for static initializers

```
public class ExpensiveResource {
  private static ExpensiveResource resource =
                        new ExpensiveResource();
  public static ExpensiveResource getInstance() {
    return resource;
  }
}
```


Lazy Initialization with Holder Class

- Use nested class to hold instance of resource
- Initialize reference in class using static initialization
  - thread-safe
- Lazy class loading ensures instance created only when needed

```
public class ExpensiveResource {
  private static class ResourceHolder {
    public static ExpensiveResource resource =
                        new ExpensiveResource();
  }
  public static ExpensiveResource getInstance() {
    return ResourceHolder.resource;
  }
}
```

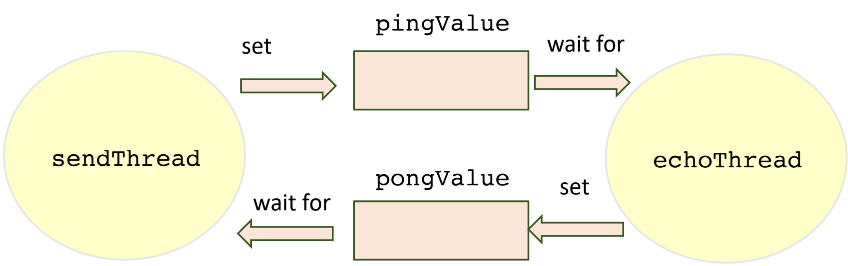
Initialized safely when the class is loaded

Causes loading of inner class if necessary

Concurrency without Locks

- Synchronisation carries an overhead
  - block/unblock thread
  - requires OS intervention
- No option on single core systems
  - multicore systems now make alternatives possible
  - spin rather than block
- Algorithms changing
  - require special low level support
  - require understanding of hardware, especially memory architectures
- "Safe memory operations"
  - volatile

Example: Ping Pong



Example: Ping Pong

```
public class PingPong {
 …
 private static final long REPETITIONS = 1L * 1000L * 1000L;
 public static void main(final String[] args) throws Exception {
  final Thread echoThread = new Thread(new EchoRunner());
  final Thread sendThread = new Thread(new SendRunner());
  echoThread.start();
  sendThread.start();
  final long start = System.nanoTime();
  echoThread.join();
  final long duration = System.nanoTime() – start;
  out.printf("duration %,d (ns)\n", duration);
  out.printf("%,d ns/op\n", duration / (REPETITIONS * 2L));
  out.printf("%,d ops/s\n",
             (REPETITIONS * 2L * 1000L * 1000L * 1000L) / duration);
  out.println("sendValue = "+sendValue+", echoValue = "+echoValue);
 }
}
```

- Driver program
  - start threads, measure time (1,000,000 iterations)

Example: Ping Pong

- Conventional synchronisation
  - condition variable and associated lock

```
public class CVPingPong {
 …
 private static final Lock sendLock = new ReentrantLock();
 private static final Condition sendCondition =
                                      sendLock.newCondition();
 private static final Lock echoLock = new ReentrantLock();
 private static final Condition echoCondition =
                                      echoLock.newCondition();
 private static long sendValue = -1;
 private static long echoValue = -1;
 …
}
```

Example: Ping Pong

```
public static class SendRunner implements Runnable {
 public void run() {
  for (long i = 0; i < REPETITIONS; i++) {
   sendLock.lock();
   try {
    sendValue = i;
    sendCondition.signal();
   } finally { sendLock.unlock(); }
   echoLock.lock();
   try {
    while (echoValue != i) {
     echoCondition.await();
    }
   } catch (final InterruptedException ex) {
    break;
   } finally { echoLock.unlock(); }
  }
 }
}
```

- Worker
  - `EchoRunner` class is similar

Example: Ping Pong

```
public class PingPong {
 …
 private static volatile long sendValue = -1;
 private static volatile long echoValue = -1;
 …
 public static class SendRunner implements Runnable {
  public void run() {
   for (long i = 0; i < REPETITIONS; i++) {
    sendValue = i;
    while (echoValue != i) {
     // busy spin
    }
   }
  }
 }
 // Similar for EchoRunner
}
```

- Lock free
  - volatile variable and spin threads

Example: Ping Pong

- Comparison of performance

```
$ time java CVPingPong
duration 14,255,025,000 (ns)
7,127 ns/op
140,301 ops/s
sendValue = 999999, echoValue = 999999

real    0m14.410s
user    0m5.383s
sys     0m11.443s
```

```
$ time java PingPong
duration 100,420,000 (ns)
50 ns/op
19,916,351 ops/s
sendValue = 999999, echoValue = 999999

real    0m0.227s
user    0m0.336s
sys     0m0.035s
```


Example: Ping Pong

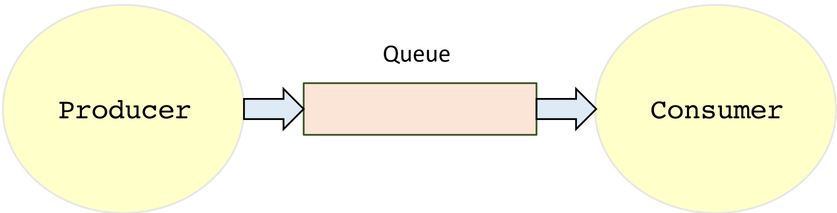
- Comparison of performance

```
$ time java CVPingPong
duration 14,255,025,000 (ns)
7,127 ns/op
140,301 ops/s
sendValue = 999999, echoValue = 999999

real    0m14.410s
user    0m5.383s
sys     0m11.443s
```

© J&G Services Ltd, 2017

Example – Single Writer/Reader Queue



Example – Single Writer/Reader Queue

- A driver program for the test

```
public class QueuePerfTest {
 public static final Integer TEST_ELEMENT = Integer.valueOf(777);
 public static final int REPS = 10 * 1000 * 1000;
 public static final int QUEUE_SIZE = 64 * 1024;
 public static void main(final String[] args) throws Exception {
  final Queue<Integer> queue = new
     java.util.concurrent.ArrayBlockingQueue<Integer>(QUEUE_SIZE);
  for (int i  = 0; i < 5; i++) {
   System.gc();
   Thread.sleep(1000);
   runTest(i, queue);
  }
 }
 …
}
```

Example – Single Writer/Reader Queue

```
private static void runTest( final int runNumber,
                final Queue<Integer> queue) throws Exception {
 final CyclicBarrier barrier = new CyclicBarrier(2);
 final Runnable runner = new Producer(barrier, queue);
 final Thread t = new Thread(runner);
 t.start();
 barrier.await();
 final long start = System.nanoTime();
 int i = REPETITIONS + 1;
 while (0 != --i) {
  while (null == queue.poll()) {
   Thread.yield();
  }
 }
 final long finish = System.nanoTime();
 final long duration = finish - start;
 final long ops = (REPS * 1000L * 1000L * 1000L) / duration;
 System.out.format("%d - ops/sec = %,d\n",
           Integer.valueOf(runNumber), Long.valueOf(ops) );
}
```

- The test

Example – Single Writer/Reader Queue

```
private static class Producer implements Runnable {
 private final CyclicBarrier barrier;
 private final Queue<Integer> queue;
 public Producer( final CyclicBarrier barrier,
                  final Queue<Integer> queue ) {
  this.barrier = barrier; this.queue = queue;
 }
 public void run() {
  try {
   barrier.await();
  } catch (final Exception ex) { ex.printStackTrace(); }
  try {
   int i = REPETITIONS + 1;
   while (0 != --i) {
    while (!queue.offer(TEST_ELEMENT)) { Thread.yield(); }
   }
  } catch (final Exception ex) { ex.printStackTrace(); }
 }
}
```

- The test - Producer

Example – Single Writer/Reader Queue

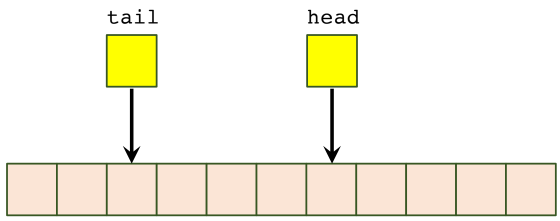
- Results from `ArrayBlockingQueue<T>`

```
$ java QueuePerfTest
0 – ops/sec = 4,133,628
1 – ops/sec = 4,103,580
2 – ops/sec = 4,767,521
3 – ops/sec = 4,367,569
4 – ops/sec = 4,324,988
```


Alternative `Queue<T>` Implementation

- Lamport Queue

- Use array, and `volatile long` values for `head/tail`
  - elements added at `tail` index
  - elements removed from `head` index
  - use mod (`%`) to ensure values can operate within array length
  - `tail – head == 0` => empty
  - `tail – head >=` *`arraylength`* => full



Alternative `Queue<T>` Implementation

- Results from `NonBlockingQueue<T>`

```
public class QueuePerfTest {
 … // As before
 public static void main(final String[] args) throws Exception {
  final Queue<Integer> queue = new
                        NonBlockingQueue<Integer>(QUEUE_SIZE);
 … // As before
}
```

```
$ java QueuePerfTest
0 – ops/sec = 14,257,260
1 – ops/sec = 13,256,744
2 – ops/sec = 15,750,710
3 – ops/sec = 14,330,219
4 – ops/sec = 13,932,020
```