Aalto University
School of Science

Java Concurrency - Part 2

CS-E4110 Concurrent Programming

Keijo Heljanko

Department of Computer Science
Aalto University School of Science

October 5th, 2016

Slides by Jan Lönnberg, Teemu Kiviniemi, Jaakko Harjuhahto and Keijo Heljanko

On the previous lecture... 1/2

Java threading and `synchronized`

- Java provides threading either by extending the class `Thread` or implementing the interface `Runnable`
- Each Java object has an intrinsic lock, inherited from the super class `Object`
- `synchronized` -blocks provide mutual exclusion with regards to a certain lock

On the previous lecture... 2/2

Java Memory Model

- The *happens-before* guarantees that the memory effects of statement A are visible to B if $hb(A, B)$
- Variables declared `volatile` are fully ordered and happen-before each other, but
- Releasing a lock at the end of a `synchronized` -block happens-before acquiring the lock again at the start of a another `synchronized` -block
- Starting threads, joining threads and interrupts establish happens-before
- The transitive nature of happens-before enables piggybacking

Section 1

Monitors

Monitors

Waiting for and delivering notifications

- Each object has a **wait set**, which is a set of threads that are waiting for notification on that object.
- The `wait()`, `notify()` and `notifyAll()` methods can be used to wait for and notify other threads.
  - All classes inherit the methods from the `Object` root class.
  - The thread that calls the methods must have ownership of the object's lock.
  - The methods make it possible to wait for and deliver generic notifications on an object instance. The reason for a notification is not known after `wait()`, and has to be checked separately.

Monitors

The `wait()` method

- When a thread executes the `wait()` method of an object, the thread:
  1. Adds itself to the object's wait set.
  2. Releases the object's lock (fully, even if multiply acquired), ensuring all variable values written will be visible to the thread that acquires the lock.
  3. If the thread holds multiple locks, only the one for which `wait()` is called is released

Monitors

Waking up from `wait()`

- A `waiting` thread will continue execution:
  - When another thread notifies the `waiting` thread with `notify()` or `notifyAll()` on the same object.
    - The `waiting` thread must then reacquire the object's lock, just like when entering a `synchronized` block (and restore re-entrancy state). Values written by the last holder of the lock are ensured to be visible.
    - The thread may have to compete with other threads to obtain the lock, because it doesn't receive any form of special priority.
  - When the thread is interrupted with the `Thread.interrupt()` method.
  - For no apparent reason (**spurious wakeup**).

Spurious Wakeups

- Java inherits the spurious wake-up behaviour from POSIX, which allows this because condition variables are hard to implement efficiently on some multiprocessor systems without the possibility of spurious wakeups in a few cases. Linux also has a habit of generating spurious wakeups when processes are signalled.
- From a performance point of view, you can usually assume that spurious wakeups are rare.
- The behaviour is easy to mitigate: **always** use a `while()` -loop to check the wait condition

Monitors

`wait()` example

```
2       public synchronized int getData() throws InterruptedException {
3           // wait until not empty or interrupted
4           while(empty) {
5               wait();
6           }
7           empty = true;
8           return value;
9       }
```

Monitors

The `notify()` method.

- The `notify()` method wakes up one thread `waiting` on an object's wait set.
- If there are no threads `waiting` at the moment, `notify()` does nothing.
- If there are multiple threads `waiting`, the thread is chosen arbitrarily.
- The notified thread is removed from the object's wait set.
- `notify()` doesn't give up ownership of the object's lock.

Monitors

Waking up after notified

- A notified thread can start execution only after it re-obtains ownership of the object's lock.
  - May happen only after the `notify`ing thread has given up the lock.
  - The notified thread may have to compete with other threads to obtain the lock.

Monitors

The `notifyAll()` method.

- The `notifyAll()` method works just like `notify()`, but instead of one thread, it wakes up all threads waiting on the object's wait set.
- If there are no threads `wait`ing at the moment, `notifyAll()` does nothing.
- The notified threads must compete to reobtain lock ownership.

`notifyAll()` example

```
11      public synchronized void writeData(int v) {
12          value = v;
13          empty = false;
14          notifyAll();
15      }
```

Interrupting

`interrupt()`

- Waiting threads can be interrupted using by calling `interrupt()` on the thread.
- `wait()`, `join()` and such throw an `InterruptedException` when interrupted.
- If the thread is not waiting, its **interrupt flag** is set.
  - The methods above also check the interrupt flag before they start waiting and are interrupted if it is set.
  - The interrupt flag is read and cleared (atomically) by the `interrupted()` method.
  - The interrupt flag is also cleared when throwing an `InterruptedException` above.

Interrupting

`interrupt()`

- `interrupt()` is useful for stopping worker threads that may be waiting for something.
- It is usually a good idea to simply leave the interrupt flag and `InterruptedException`s alone unless you know what you're doing with them.
  - In practice, this means that general-purpose code should let the calling application handle these exceptions.
  - It's usually a good idea to have an **interruption policy** for each (type of) thread and only use `interrupt` on threads for which you know it (i.e. ones you've created).

Section 2

Problems with concurrency

Problems with concurrency

Background

- Concurrent programs often behave in unexpected ways.
- Programming assignments have, on average 1-2 concurrency errors
- Studying common mistakes and applying practices that avoid them makes it easier to produce correct code.

Protecting groups of variables

Problem

An object (or data structure consisting of many variables) is accessed from two or more threads without `synchronized`.

Avoidance

- Look for **all** objects that can be accessed by many threads.
- Access each shared object only inside `synchronized` blocks or methods with the same lock.
- Make sure state is consistent before you release the lock.
- **Immutable** objects are also safe (all fields **final**) if fully constructed before being exposed.

Releasing locks allows concurrent modification

Problem

A `synchronized` block is split into two blocks, because a time-consuming task is executed in between.

```
1  synchronized(this) {
2          while(bufferFull) {
3                  try { wait(); } catch(InterruptedException e) {}
4          }
5  }
6  DataType data = generateData();
7  synchronized(this) {
8          buffer = data;
9          bufferFull = true;
10 }
```

Releasing locks allows concurrent modification

Problem

The value of `bufferFull` may change between the `synchronized` blocks.

Avoidance

- Don't make any assumptions on the state of the system when entering a `synchronized` block. Any previously checked states might have changed.
- Assume the scheduler is **evil**. If a thread enters the first `synchronized` block before another thread, the thread might not enter the second block before it.

Releasing locks allows concurrent modification

Avoidance

Now the buffer state is checked in the beginning of the `synchronized` block.

```
1  DataType data = generateData();
2  synchronized(this) {
3          while(bufferFull) {
4                  try {
5                          wait();
6                  } catch(InterruptedException e) {}
7          }
8          buffer = data;
9          bufferFull = true;
10 }
```

Using `wait()` unconditionally

Problem

`wait()` is called unconditionally, even if the condition is already true.

```
2       synchronized void waitUntilZero() throws InterruptedException {
3           wait();
4       }
5
6       synchronized void decrement() {
7           value--;
8           if (value == 0)
9               notifyAll();
10      }
```

`wait()` with `if`

Problem

Checking for a condition is done with an `if` statement:

- **The `wait()`ing thread may wake up spuriously.**
- **The condition may become false again before the woken thread gets the lock.**

```
2       synchronized void waitUntilZero() throws InterruptedException {
3           if (value == 0) {
4               try {
5                   wait();
6               } catch(InterruptedException e) {}
7           }
8       }
```

wait() with while

Avoidance

The condition **must** be checked in a `while` loop (or equivalent).

```
2   synchronized void waitUntilZero() throws InterruptedException {
3       while(value != 0)
4           wait();
5   }
6   synchronized void decrement() {
7       value--;
8       if (value == 0)
9           notifyAll();
10  }
```

Missignaling with `notify()`

`notify()` may wake up any waiting thread

- Example:
  - Threads waiting for conditions A and B under lock L
  - Actions by thread P changes condition A to true
  - P calls `notify()` and expects a waiting thread on A to wake up and proceed
  - The signal wakes up a thread waiting for B, it checks that the condition is still not true and goes back to waiting
  - The program remains in an invalid state, often a deadlock
- This is the "signal being lost"
- The problem is avoided by using `notifyAll()`, which may in turn degrade performance

Section 3

Performance issues

Busy waiting and polling

Problem

Busy waiting or polling is used to wait for a condition.

- **Using `wait()` with the optional time-out argument is a form of polling**
- **Busy waiting wastes lots of CPU time.**
- **Adding a delay saves CPU time but adds latency.**
- **`wait()` uses practically no time and responds quickly.**

synchronized vs. volatile

Comparison

- Entering and exiting a `synchronized` section is slow due to locking (especially if the lock is not free).
- Once a thread has entered a `synchronized` section, the code inside it is executed quickly.
- Avoid a lot of separate `synchronized` blocks, but remember that long `synchronized` blocks may block other threads longer.
- If accessing several shared variables, or accessing the same variable several times in the same place, use a single `synchronized` section.
- If accessing a single variable once, you can declare the variable `volatile` for a gain in speed and simplicity.

notify vs. notifyAll

Comparison

- `notifyAll()` wakes **all** threads waiting on an object.
- `notify()` wakes **one** thread waiting on an object.
- Indiscriminate use of `notifyAll()` can waste a lot of CPU time by waking up many threads to check whether they can proceed.
- If you use `notify()`, you just have to be careful that whatever thread is woken, it is one that can proceed.

Section 4

java.util.concurrent

Concurrent data structures

General

- Concurrent equivalents in `java.util.concurrent` of `java.util` collections.
- Most operations are atomic and putting an object in the structure happens-before reading or removing it, allowing your code to piggyback on the structures' synchronisation.

Blocking queue structures

`BlockingQueue` `Queue` with many atomic operations and operations that block until the queue is not empty/full

`SynchronousQueue` 0-size queue; `put` and `take` wait for each other.

`BlockingDeque` Like `BlockingQueue` and `Deque`

Concurrent data structures

Other collections

`ConcurrentMap` Like `Map`, but with additional atomic methods to test and set (`putIfAbsent`, `remove` and `replace`).

`CopyOnWriteArrayList/-Set` Variants of `ArrayList` and `ArraySet` that avoid synchronisation by copying their contents on modification

- This is only useful if traversal is much more common than modification.

Locks and condition variables

Locks

- In the `java.util.concurrent.locks` package
- `Lock` interface and implementations
- Locking without block structure (e.g. chain locking)
- Implementations (e.g. `ReentrantLock`) may provide functionality such as:
  - Fairness
  - Reentrancy
  - `interrupt()`ing operations
- Same happens-before effects as normal locking/unlocking

Locks and condition variables

Condition variables

- Associated with `Locks`
  - Each `Lock` can have several `Conditions`.
  - Created through `Lock.newCondition()` method
- `await()`, `signal()` and `signalAll()` have similar semantics to equivalents in `Object`:
  - Lock must be held to use condition variable operations.
  - Lock is automatically released and reacquired when waiting.
  - Operations can be `interrupted`.
- Waiting threads are signalled in FIFO order, spurious wakeups can occur.
- Do not use the `notify` and `wait` methods on these!

Locks and condition variables

Example

```
1  import java.util.concurrent.locks.*;
2
3  public class LockCounter {
4      public void decrement() {
5          l.lock();
6          value--;
7          if (value == 0)
8              c.signalAll();
9          l.unlock();
10     }
12     public void waitForZero()
13         throws InterruptedException {
14         l.lock();
15         while (value != 0)
16             c.await();
17         l.unlock();
18     }
19
20     private int value = 10;
21     private Lock l = new ReentrantLock();
22     private Condition c = l.newCondition();
23 }
```

Futures and executors

Futures

- A **future** represents the result of an asynchronous computation. `Futures` in Java can:
  - `isDone()` Check if the method has completed.
  - `get()` Wait for and get the result (or throw an exception if the computation threw one).
  - `cancel()` Cancel the computation (if it has not yet completed or been cancelled).
- The computation happens-before getting the result.
- A `RunnableFuture` is a `Runnable` `Future`.

Futures and executors

`Callables`

- A `Callable` is similar to a `Runnable`, but returns a result or throws an exception.

`Executors`

- An `Executor` can `execute(Runnable)` asynchronously.
  - This allows you to have one object that coordinates multiple asynchronous tasks instead of explicitly creating new threads for each task.
  - The executor may execute all `Runnables` concurrently or limit the number that execute simultaneously.

Futures and executors

Example

```
1  class AddCallable implements Callable<Long> {
2      private long start, end;
3
4      AddCallable(long min, long max) {
5          start = min;
6          end = max;
7      }
```

Futures and executors

Example

```
12        public Long call() {
13            long r = 0; /* start − (start + 1) + ... − end. */
14            for(long i = start; i <= end; i += 2) {
15                r += i;
16                r −= i + 1;
17            }
18            return r;
19        }
20    }
```

Futures and executors

ExecutorServices

- An `ExecutorService` is an `Executor` that:

  `submit(Callable)` Executes the `Callable` asynchronously, returns a `Future` representing the result.

  `shutdown()` Stop accepting new tasks.

  `awaitTermination()` Wait for shutdown to complete by allowing remaining tasks to finish.

- A `ThreadPoolExecutor` is an `ExecutorService` that maintains a pool of threads and reuses these threads to execute tasks submitted to it.

Futures and executors

Queuing

- `ThreadPoolExecutors` use a `BlockingQueue` to track tasks to perform.
- You can use an unbounded queue (and try not to run out of memory).
- A `SynchronousQueue` blocks the submitting thread until a thread is available to process the request.
- Using a positive-size bounded queue may be better for performance.
- If you have a bounded queue, you need to specify a **saturation policy**; what to do when the queue is full, e.g.:

  `CallerRunsPolicy` Use the submitting thread to run the queued task.
  `AbortPolicy` Throw an exception when submitting a task that cannot be executed or queued.

Futures and executors

Example

```
23      public static final int THREADS = 8;
24      public static final long SIZE = 10000000, BLOCKS = 10000;
25
26      public static void main(String[] args) throws Exception {
27          ExecutorService es =
28              new ThreadPoolExecutor(THREADS, THREADS, 0,
29                                     TimeUnit.SECONDS,
30                                     new SynchronousQueue<Runnable>(),
31                                     new ThreadPoolExecutor.CallerRunsPolicy());
```

Futures and executors

Example

```
33          List<Future<Long>> blocks = new ArrayList<Future<Long>>();
34          for(long i = 0; i < BLOCKS; i++)
35              blocks.add(es.submit(new AddCallable(i * SIZE, i * SIZE + SIZE - 1)));
36
37          long result = 0;
38          for(Future<Long> b : blocks)
39              result += b.get();
40
41          System.out.println(result);
42
43          es.shutdown();
44      }
```

Section 5

Conclusion

Conclusion

References

- The Java Language Specification, Java SE 7 Edition
  - Especially Chapter 17.
  - `http://java.sun.com/docs/books/jls/`
- Java™ 2 Platform, Standard Edition 7 API Specification
  - Especially classes `java.lang.Object` and `java.lang.Thread` and package `java.util.concurrent` and its subpackages.
  - `http://docs.oracle.com/javase/7/docs/api/`
- Goetz et al.: Java Concurrency in Practice
  - Especially Chapters 2, 3, 7, 8, 14 and 16.