Usage Considerations of Java Synchronizer Classes

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Know the key synchronizers defined in the Java class library

• Recognize synchronizer usage considerations

Performance

Productivity
Java Synchronizer Class
Usage Considerations
Java Synchronizer Class Usage Considerations

• Choosing between these synchronizers involve understanding tradeoffs between *performance* & *productivity*
Java Synchronizer Class Usage Considerations

• Choosing between these synchronizers involve understanding tradeoffs between *performance* & *productivity*

• Some synchronizers (or synchronizer methods) have more overhead

See [en.wikipedia.org/wiki/Overhead_(computing)](en.wikipedia.org/wiki/Overhead_(computing))
Choosing between these synchronizers involve understanding tradeoffs between *performance* & *productivity*

- Some synchronizers (or synchronizer methods) have more overhead, e.g.
  - Spin locks
    - A lock that causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available

See [en.wikipedia.org/wiki/Spinlock](en.wikipedia.org/wiki/Spinlock)
Choosing between these synchronizers involve understanding tradeoffs between performance & productivity.

Some synchronizers (or synchronizer methods) have more overhead, e.g.

- Spin locks
- Sleep locks
  - A lock that does not return until another thread has issued a notification that some event may have occurred

See docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
Java Synchronizer Class Usage Considerations

- Choosing between these synchronizers involve understanding tradeoffs between \textit{performance} & \textit{productivity}
- Some synchronizers (or synchronizer methods) have more overhead, e.g.
  - Spin locks
  - Sleep locks
  - Adaptive locks
    - A lock that spins when trying to access a resource locked by a currently-running thread, but sleeps if the locking thread is not currently running

See hackernoon.com/building-a-c-hybrid-spin-mutex-f98de535b4ac
Java Synchronizer Class Usage Considerations

- Choosing between these synchronizers involve understanding tradeoffs between *performance* & *productivity*
- Some synchronizers (or synchronizer methods) have more overhead
- Some synchronizers are harder to program correctly than others
Choosing between these synchronizers involve understanding tradeoffs between *performance* & *productivity*.

- Some synchronizers (or synchronizer methods) have more overhead.
- Some synchronizers are harder to program correctly than others.
  - e.g., risk of deadlock from non-reentrant locking semantics.

**Deadlocks are problematic in object-oriented frameworks due to callbacks & complex control flows**.

See [en.wikipedia.org/wiki/Deadlock](en.wikipedia.org/wiki/Deadlock)
Java Synchronizer Class Usage Considerations

• Java synchronizers differ from Java built-in monitor objects
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects, e.g.
- They are largely written in Java rather than C/C++
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects, e.g.
  - They are largely written in Java rather than C/C++
  - Some low-level methods written in native C/C++
    - e.g., `compareAndSwapInt()`, `park()`, `unpark()`, etc.

Concurrency

And few words about concurrency with Unsafe. `compareAndSwap` methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```java
interface Counter {
    void increment();
    long getCounter();
}
```

Then we define worker thread `CounterClient`, that uses `Counter`:

```java
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects, e.g.
  - They are largely written in Java rather than C/C++
  - They provide *many* more features & have more powerful semantics
End of Usage Considerations of Java Synchronizer Classes