

Practical Concurrency

Agenda

- Motivation
- Java Memory Model Basics
- Common Bug Patterns
- JDK Concurrency Utilities
- Patterns of Concurrent Processing
- Testing Concurrent Applications
- Concurrency in Java 7

Motivation

- Multi-core architecture changes everything
 - Parallel execution is more common
 - Memory model is more important

Understand program execution



- Know what the VM is doing with your application
- Understand how best to express concurrent semantics

Java Memory Model Basics

Core Principles



- The JMM defines:
 - Ordering
 - Atomicity
 - Visibility

Basic Guarantees



- Ordering
 - Within thread as-if-serial
- Atomicity
 - Read and writes to all fields except long and double
- Visibility
 - Within thread only

Improving these guarantees



- Synchronized
 - Define atomic blocks
 - Constrain ordering
- Volatile
 - Control cross-thread visibility
 - Constrain ordering
 - Atomic read/write to long and double
- Final

Understanding *happens-before*



- Program order rule
- Monitor lock rule
- Volatile variable rule
- Thread start rule
- Thread termination rule

Mystery of Visibility



Safe publication



- Final
- Volatile
- Locking

Common Bug Patterns

Common Bug Patterns



- Shared, non-volatile primitives
- One-sided synchronization
- Mixed synchronization
- Incorrect encapsulation

Demo

Common Bug Patterns

JDK Concurrency Utilities

Building Blocks



- Atomics
- Latches
- Semaphores
- Locks and Conditions

- Common operations on common types handled atomically
- Support for manipulating references and primitives
- Supports updates of fields reflectively

Semaphores

- Manage access to resources
- Provide some degree of ordering

Latches

- Provide control across task execution
- Await a condition

Locks and Conditions



- Lock interface as an alternative to synchronized
- Condition interface as an alternative to wait/notify
- Prefer inbuilt operations where possible
 - Use when read/write locking can offer a real benefit

Demo

Building Blocks

Concurrent Collections



- ConcurrentHashMap
- CopyOnWrite(List|Set)
- ConcurrentSkipList(Map|Set)
- Improvement over synchronized Collections
 - Permit interleaved read/write

- Ideal where read greatly outweighs write
 - Event listeners for example

ConcurrentHashMap



- Default ConcurrentHashMap
 - Supports a configurable level of throughput
 - Ideally match to expected number of threads
 - Provides 'weakly-consistent' views
 - Iteration does not throw ConcurrentModificationException

ConcurrentSkipList*



- Concurrent replacement for sorted Map and sorted Set
- Typically lower throughput than ConcurrentMap
- Some operations are not constant time
 - size() requires traversal of all elements

Weak Consistency



- Iterators on ConcurrentHashMap and ConcurrentSkipListMap
- Reflect the state of the Map at or since the creation of the iterator
- CopyOnWrite gives **fully consistent** iterators

ConcurrentMap interface



- putIfAbsent
 - Add an item if not present and return previously mapped value
- remove
 - Only if mapped to supplied value
- replace
 - Only if mapped to supplied value

Why composite operations?



Thread A

```
Object val = get("key");
```

```
someProcess(val); // invalidates "key"
```

```
remove("key"); // incorrect!
```

Thread B

```
set("key", new Object()); // key is now valid
```

Task Execution



- Abstraction around execution of concurrent tasks
- Based on a thread pool
- Highly configurable
- Supports
 - Delays
 - Prioritization
 - Concurrent result processing

Demo

Task Execution

Patterns of Concurrent Processing

Concurrent Caches



- Use `ConcurrentHashMap`

Updating Data Structures



- CopyOnWrite Collections for Lists/Sets
- Thread confinement for simple data structures

Encapsulation



- Encapsulate access to shared data
- Avoid synchronized methods
- Keep locking internal

Demo

Caches and Data Structures

Handling Interruption

- Use interruption to stop task execution
- Never ignore an InterruptedExecution
 - Propagate the state
 - Or handle the interrupt

Demo

Handling Interruption

Client Locking Protocols



- Allowing clients to participate in your locking protocol
- Avoid where possible
 - Can lead to deadlock
- Expose composite operations or consider concurrent structures

Alien Calls

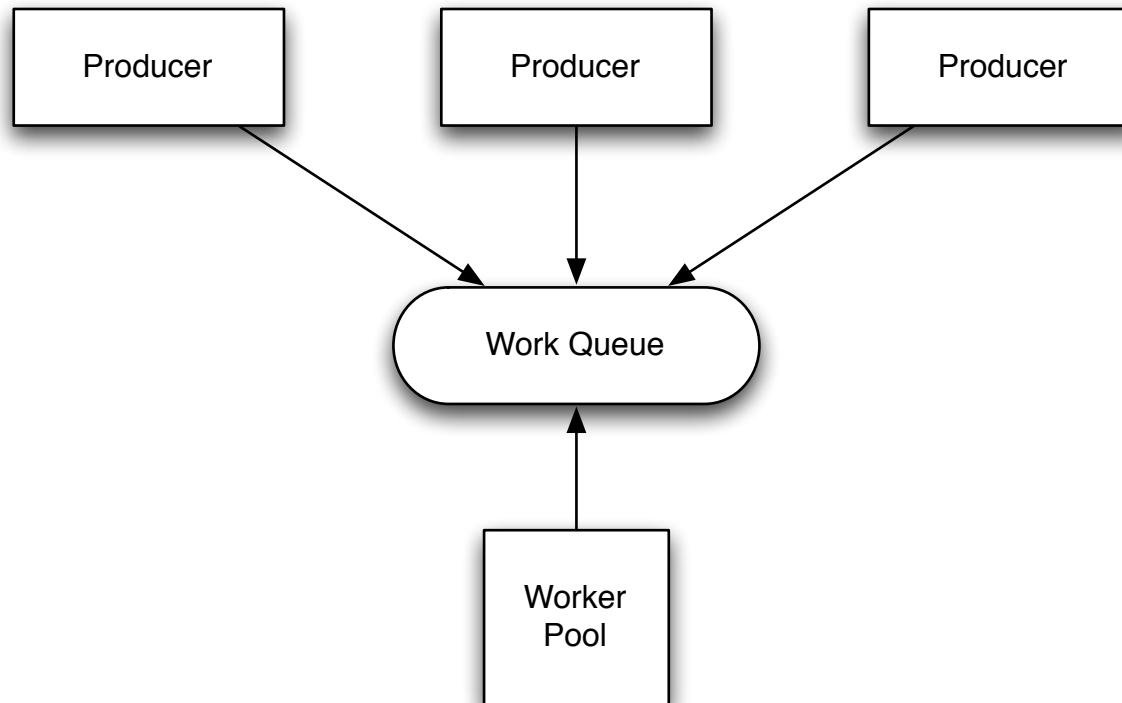
- Calling methods outside your control whilst holding a lock
- Avoid at all costs
 - Easily leads to deadlock
 - Can block indefinitely

Producer/Consumer



- Decouple the production of work items from their processing
- Build on ExecutorService and/or BlockingQueue
- Basic building block for a simple SEDA architecture

Pattern



Demo

Producer/Consumer

Testing Concurrent Applications

- MultithreadTC
 - <http://www.cs.umd.edu/projects/PL/multithreadedtc/>

Demo

Testing

Concurrency in Java 7

Fork and Join

- Framework for parallel processing
- Recursively split processing into smaller chunks
- Details can be found at: <http://g.oswego.edu/dl/concurrency-interest/>

Basic Pattern

```
Result solve(Problem problem) {  
    if (problem is small)  
        directly solve problem  
    else {  
        split problem into independent parts  
        fork new subtasks to solve each part  
        join all subtasks  
        compose result from subresults  
    }  
}
```

Source: A Java Fork/Join Framework (Doug Lea)

Summary

Summary



- Concurrency is essential part of today's applications
- Java has **excellent** supporting libraries
 - Use them
- Encapsulate your own shared data
- Keep an eye on Java 7

Q&A