Application, Process and Thread

- Application

  is a combination of data and instructions which performs a specific task

- Process

  is loaded application in RAM or instance of a computer program

- Execution Thread

  is part of the process that executes machine instructions

Application, Process and Thread in Java

- Application

  application is a combination of JVM and compiled Java classes

- Process

  loaded JVM with loaded application’s classes in memory

- Execution Thread

  a Main thread that is started by JVM and execute code of the main method of a Java application and can create and start child threads

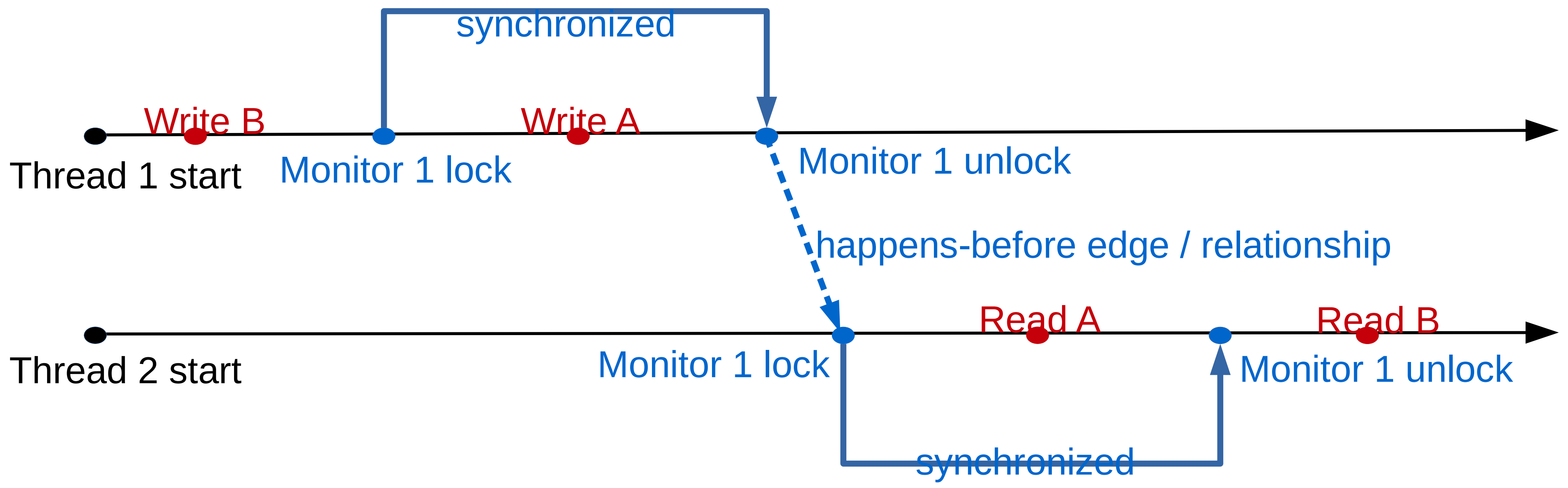
Java Basics

- Difference among class/static class/inner class/interface
- Methods declaration and static methods
- Final and static fields
- Functional interface and lambda functions
- Basics of Streams API

Environment

- Java 8 OracleJDK or OpenJDK
- IntelliJ IDEA or any other Java IDE
- Maven
- Any OS which supports Java (Windows/ Mac/ Linux)

Happens-Before Relationship



Atomic Variables

- **AtomicBoolean, AtomicInteger, AtomicLong** - a value that may be updated atomically.
- **AtomicIntegerArray, AtomicLongArray** - an int or long array in which elements may be updated atomically.
- **AtomicReference<V>** - an object reference that may be updated atomically.
- **AtomicStampedReference<V>** - An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
- **AtomicMarkableReference<V>** - an AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
- **AtomicReferenceArray<E>** - an array of object references in which elements may be updated atomically.
- **AtomicIntegerFieldUpdater<T>, AtomicLongFieldUpdater<T>** - a reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
- **AtomicReferenceFieldUpdater<T,V>** - a reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.

First Section Results

- Process and Thread
- Thread start, stop and join methods
- Daemon Threads
- Synchronized keyword
- Happens-before relationship
- Volatile variables
- Atomic variables
- Compare-And-Set operations
- Wait/Notify construction
- ThreadLocal variables

Advantages of Multithreading

- Can split work among different threads
- Higher throughput
- Lower latency
- More responsive application
- Easy to create, start, interrupt and join a thread
- Simple constructions: synchronized, volatile, atomics and wait/notify methods

Disadvantages of Multithreading

- Understanding of happens-before relationship and visibility of operations
- Data-race, race-condition, live-lock, starvation and other concurrent bugs
- IDE doesn't highlight concurrent bugs
- Bugs which are not reproducible and hardly explainable
- Requires a lot of mental efforts and attention from a developer

java.util.concurrent package

- Atomics – AtomicInteger, AtomicLong, AtomicReference and others
- Locks – Reentranlock, ReentrantReadWriteLock, Condition variables
- Advanced synchronization primitives - Semaphores, CountDownLatch, CyclicBarrier, Exchanger
- Concurrent Collection
- Executors
- CompletableFuture

BlockingQueue implementations

ArrayBlockingQueue - a bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.

BlockingQueue implementations

LinkedBlockingQueue - an optionally-bounded blocking queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time.

The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.

BlockingQueue implementations

PriorityBlockingQueue - an unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations.

While this queue is logically unbounded, attempted additions may fail due to resource exhaustion (causing OutOfMemoryError).

This class does not permit null elements.

A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so results in ClassCastException).

BlockingQueue implementations

DelayQueue - an unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired.

The head of the queue is that Delayed element whose delay expired furthest in the past. If no delay has expired there is no head and poll will return null.

Expiration occurs when an element's getDelay(TimeUnit.NANOSECONDS) method returns a value less than or equal to zero. Even though unexpired elements cannot be removed using take or poll, they are otherwise treated as normal elements. For example, the size method returns the count of both expired and unexpired elements.

This queue does not permit null elements.

BlockingQueue implementations

SynchronousQueue - a blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

A synchronous queue does not have any internal capacity, not even a capacity of one. You cannot peek at a synchronous queue because an element is only present when you try to remove it; you cannot insert an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate.

The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null.

For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements.

BlockingQueue implementations

TransferQueue - a BlockingQueue in which producers may wait for consumers to receive elements.

A TransferQueue may be useful for example in message passing applications in which producers sometimes (using method transfer(E)) await receipt of elements by consumers invoking take or poll, while at other times enqueue elements (via method put) without waiting for receipt.

Non-blocking and time-out versions of tryTransfer are also available. A TransferQueue may also be queried, via hasWaitingConsumer(), whether there are any threads waiting for items, which is a converse analogy to a peek operation.

Like other blocking queues, a TransferQueue may be capacity bounded. If so, an attempted transfer operation may initially block waiting for available space, and/or subsequently block waiting for reception by a consumer.

Note that in a queue with zero capacity, such as SynchronousQueue, put and transfer are effectively synonymous.

BlockingQueue implementations

LinkedBlockingDeque - implements BlockingDeque interface.

An optionally-bounded blocking deque based on linked nodes.

The optional capacity bound constructor argument serves as a way to prevent excessive expansion. The capacity, if unspecified, is equal to Integer.MAX_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the deque above capacity.

Most operations run in constant time (ignoring time spent blocking). Exceptions include remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations, all of which run in linear time.

BlockingQueue implementations

LinkedTransferQueue - an unbounded TransferQueue based on linked nodes. This queue orders elements FIFO (first-in-first-out) with respect to any given producer.

The head of the queue is that element that has been on the queue the longest time for some producer. The tail of the queue is that element that has been on the queue the shortest time for some producer.

Beware that, unlike in most collections, the size method is NOT a constant-time operation. Because of the asynchronous nature of these queues, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal.

Additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.

Synchronized Collections

- Collections.synchronizedCollection
- Collections.synchronizedList
- Collections.synchronizedSet
- Collections.synchronizedMap

Copy-On-Write Collections

- CopyOnWriteArrayList
- CopyOnWriteArraySet

Non-Blocking Queues

- ConcurrentLinkedQueue
- ConcurrentLinkedDeque

Blocking Queues

- BlockingQueue - interface
- ArrayBlockingQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- DelayQueue
- SynchronousQueue
- TransferQueue
- LinkedTransferQueue
- LinkedBlockingDeque - implements BlockingQueue and BlockingDeque interface

Maps and Sets

- ConcurrentMap – interface
- ConcurrentHashMap
- ConcurrentHashMap.newKeySet()

Maps and Sets

- ConcurrentNavigableMap – interface, extends NavigableMap and SortedMap interfaces
- ConcurrentSkipListMap
- ConcurrentSkipListSet

Thread Pool’s Advantages

- Simple thread management
- Don’t need to call the start, join, interrupts methods of a thread
- Dynamic threads creation if needed
- Convenient result of an asynchronous task – Future object
- It’s possible to schedule a task for execution with a given delay or periodically
- ForkJoinPool for recursive tasks

Thread Pool’s Disadvantages

- It’s needed to choice thread pool size correctly
- Deadlocks are possible
- Long-running tasks can exhaust a thread pool
- Should not forget to call the shutdown or shutdownNow method for a thread pool
- ForkJoinPool is effective only for number crunching algorithms

Performance

- Performance defines how much work an application can process for some period of time and how fast it can process a single unit of work
- Latency is time required to perform some action or to produce some result
- Throughput is the number of executed actions or results produced per unit of time

Testing

Unit Testing

- Write a test for a single method without multithreading
- Check thread safety of thread safe classes
- Test a multithreaded application with stages with barriers CountDownLatch or Phaser

Testing

Stress Testing

- Checking potential limits of an application
- Can check correctness of an application
- Can help to avoid complicated Unit tests

Testing

Performance Testing

- Measure Throughput and Latency with given count of requests per time unit
- A testing environment should be close in configuration to production
- Should help you to figure out the maximum number of clients which can serve you system

Actors with Akka

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala

- Homepage

  https://akka.io

- Hello World Example

  https://developer.lightbend.com/guides/akka-quickstart-java/define-actors.html

Reactive Style

RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.

- Homepage

  https://github.com/ReactiveX/RxJava
- Reactive programming style explanation

  http://reactivex.io

Transactional Memory

Multiverse Software Transactional Memory

A software transactional memory implementation for the JVM. Access (read and writes) to shared memory is done through transactional references, that can be compared to the AtomicReferences of Java. Access to these references will be done under A (atomicity), C (consistency), I (isolation) semantics.

- Homepage

  https://github.com/pveentjer/Multiverse

- Example

  https://www.baeldung.com/java-multiverse-stm

Other Libraries

- Guava is a set of core Java libraries from Google that includes new collection types

  https://guava.dev/releases/18.0/api/docs/com/google/common/cache/package-summary.html

  https://guava.dev/releases/snapshot-jre/api/docs/
- JCTools - Java Concurrency Tools for the JVM. This project aims to offer some concurrent data structures currently missing from the JDK

  https://github.com/JCTools/JCTools
- Disruptor is a library for the Java programming language that provides a concurrent ring buffer data structure of the same name

  https://lmax-exchange.github.io/disruptor/

  https://www.baeldung.com/lmax-disruptor-concurrency
- Comparing different concurrent frameworks

  https://medium.com/@vijay.vk/a-birds-eye-view-on-java-concurrency-frameworks-5072fd3a8759

Additional Materials

- Java source code and documentation
- Java Concurrency in Practice

  https://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601
- Java Memory Model

  https://shipilev.net/blog/2014/jmm-pragmatics/

  https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/
- What Every Programmer Should Know About Memory

  https://people.freebsd.org/~lstewart/articles/cpumemory.pdf
- The Art of Multiprocessor Programming

  https://www.amazon.com/Art-Multiprocessor-Programming-Revised-Reprint/dp/0123973376