Concurrent programming

Agenda

- The Java memory model
- Threads and thread pools
- Thread synchronization
- Thread safety

Java memory model

Threads

- Threads are also referred to as **lightweight processes**
- Threads are separate execution flows within a single process
- They share the resources of the process including open files and memory

An even more "lightweight" version of threads exists called fibers that are parallel execution flows within a single thread. Currently native support for fibers in the JVM is being developed under project Amber.

Threads

- Java threads are typically bound to native OS threads that execute them on different OS processors
- Some JVM implementations also provide also so called **green threads** that are scheduled for execution by the JVM itself rather than the OS
- Threads in the JVM might be user or daemon threads

Threads

- A typical Java application contains multiple threads of execution
- The JVM itself once started starts different threads
- The Java application may also start a number of threads during its execution

Concurrency related utilities are provided by the java.util.concurrent package of the JDK.

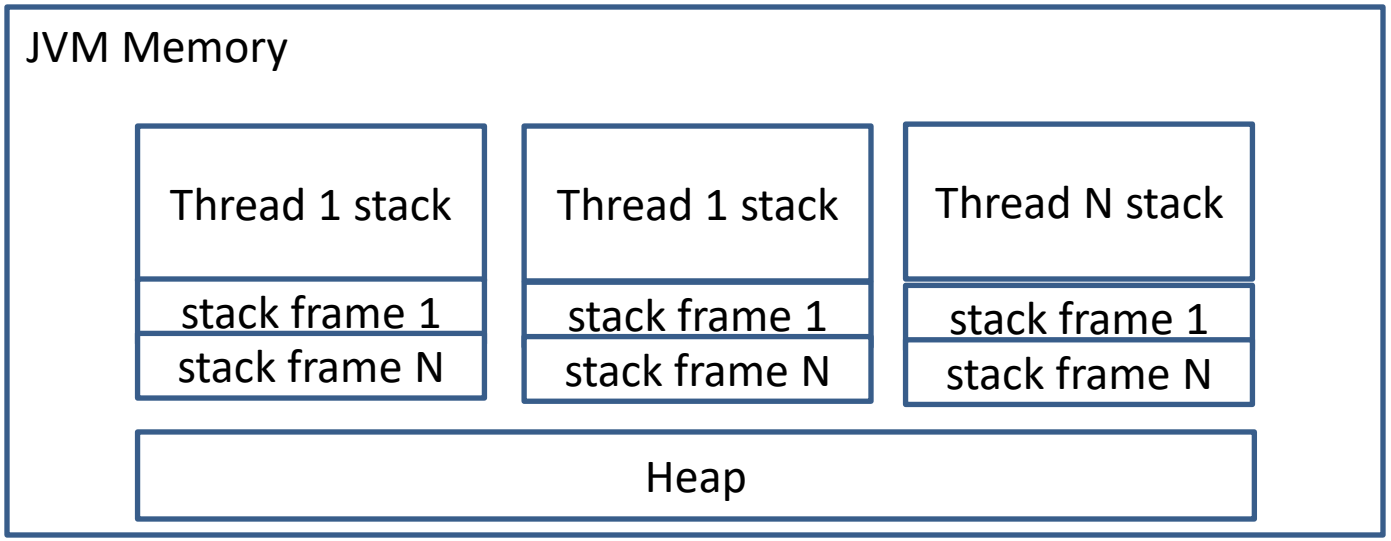
The Java memory model (JMM)

- The Java Memory Model describes how threads interact with JVM memory
- Different processors typically provide local caches that are synchronized with main memory
- Since Java threads may run on different processors they may see a different view of the same shared memory due to the caches

Keeping processors caches at all time in sync with main memory is costly and not feasible due to performance penalties.

The Java memory model (JMM)

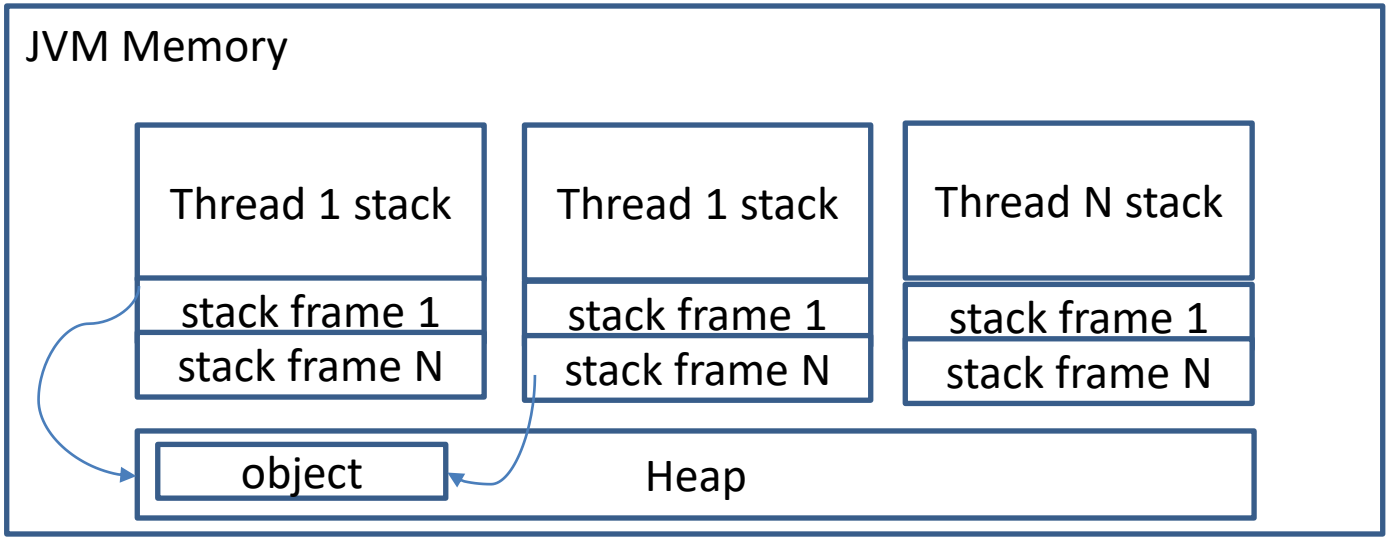
- Each thread has its own thread stack allocated
- The thread stack contains local variables that are not visible to other threads



The Java memory model (JMM)

- Since objects are stored on the heap they might be accessed by multiple threads

- Additional mechanisms should be used to ensure proper memory visibility and synchronization among threads



The JVM memory (stack and heap) is mapped to the RAM on the target machine

The Java memory model (JMM)

- Synchronization with main memory can be achieved as follows:
  - marking a variable as **volatile** makes reading from and writing to the variable directly from main memory
  - creating a synchronized block with the **synchronized** keyword may also perform a cache flush (in most cases partial using special instruction like memory barriers) once a thread exists the block

Synchronized blocks in that regard serve not only as a mechanism to provide order of execution but also memory visibility.

Threads and thread pools

Java threads

- There are two main ways to create a thread:
  - by implementing the **java.lang.Runnable** interface (preferred way)

```
new Thread(new Runnable() {

        @Override
        public void run() {
                // thread logic here
        }
}).start();
```

```
new Thread(() -> {
        // thread logic here
}).start();
```

Java threads

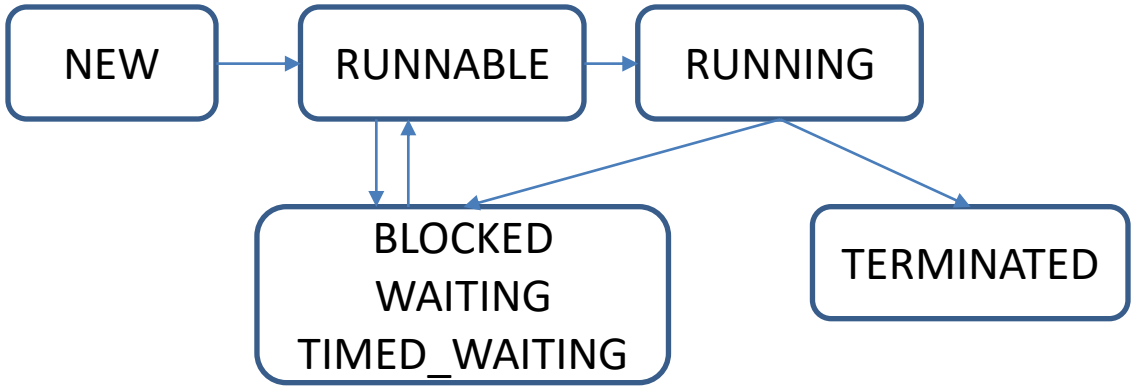
- There are two main ways to create a thread:
  - by extending the **java.lang.Thread** class

```
public class CustomThread extends Thread {
	@Override
	public void run() {
		// thread logic here
	}
}
```

```
new CustomThread().start();
```

Thread states

- A non-running thread might have any of the states defined in the Thread#State enum:
  - **NEW**
  - **RUNNABLE**
  - **BLOCKED**
  - **WAITING**
  - **TIMED_WAITING**
  - **TERMINATED**



The Javadoc on each of these states provides clear overview of when it can occur:
https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Thread.State.html

Thread interruption

- A thread can be interrupted by calling the **Thread.interrupt** method on the thread object

- The **Thread.interrupt** method sets a special flag that indicates the thread is interrupted and that can be checked with the **Thread.isInterrupted** method on the thread object

- Some methods such as a **Thread.sleep** throw a **java.lang.InterrupteException** if the calling thread is interrupted

Thread pools

- Threads might be reused since thread creation is an expensive process
- In the JDK different thread pools are provided as an implementation of the **java.util.concurrent.Executor** interface
- The **java.util.concurrent.Executors** class provides static methods for the creation of different types of thread pools

Thread pools

- The **java.util.concurrent.Executor** is extended by an **ExecutorService** interface that provides more operations for thread pools

- The ExecutorService interface if further implemented by the **ScheduledExecutorService** interface that provides the possibility to schedule threads for execution

Thread pools

```
final int numThreads = 10;
ExecutorService executor =
        Executors.newFixedThreadPool(numThreads);

for (int i = 0; i < 100; i++) {
        Runnable task = new Task(i);
        executor.execute(task);
}

executor.shutdown();
// waits for all threads to finish
executor.awaitTermination();
System.out.println("Finished all threads");
```

Thread locals

- Thread locals are special types of variables that can only be written and read from a single thread

```
private ThreadLocal threadLocal =
        new ThreadLocal();
```

```
threadLocal.set("some value");
```

```
String value =
        (String) threadLocal.get();
```

Futures

- Futures are used to represent the result of a future execution
- In Java futures are represented by the **java.util.concurrent.Future** interface
- Futures provide the possibility to cancel a task, check if the task is done or get the result (blocks until computation is done)

Futures

```
final int numThreads = 10;
ExecutorService executor =
       Executors.newFixedThreadPool(numThreads);

Runnable task = new Task();
Future<String> future = executor.submit(task);

// blocks until task is completed
String result = future.get();
```

Thread synchronization

Thread synchronization

- Since different threads can access shared data this may result in data consistency issues also known as **race condition**

- To solve issues related to race conditions a mechanism called thread synchronization can be used

Thread synchronization

- Java uses object monitors to perform thread synchronization
- A monitor is associated with an object and conducts a region that can be accessed by one thread at a time
- A monitor is acquired by a thread using a **lock**
- A lock can be either **intrinsic** or **extrinsic**

Intrinsic locks

- An intrinsic block is created using the **synchronized** keyword that:
  - can be applied as a method attribute that makes the method block a synchronized region using the method's object as a monitor

```
public synchronized void someMethod() { … }
```

  - can be specified as a separate block within a method using a target object as a monitor

```
synchronized (object) {
        // block of code
}
```

Extrinsic locks

- Extrinsic locks are provided as different implementations of the **java.util.concurrent.locks.Lock** interface:
  - ReentrantLock
  - ReadWriteLock
  - StampedLock
- Extrinsic locks provide more flexibility than intrinsic locks such as:
  - the possibility to create the synchronized block across multiple methods
  - the possibility to check if a lock is already acquired with the **tryLock** method

Extrinsic locks

```
public void someMethod() {
    reentrantLock.lock();
    try {
        // block of code
    }finally {
        reentrantLock.unlock();
    }
}
```

Joining threads

- The **Thread.join** method provides a mechanism whereby one thread can wait for the completion of another thread

```
// called by thread A
public void someMethod() {

        // some logic ...

        // wait for thread B to complete
        threadB.join();
}
```

Synchronization issues

- Problems may occur when synchronization is not applied properly such as:
  - **deadlock**: two or more threads are locked indefinitely waiting for each other
  - **starvation**: a thread is not able to gain access to shared resource thus not being able to make progress
  - **livelock**: similar to deadlock but occurs when two or more threads act on each other as a response and are not blocked but continue indefinitely without being able to make progress

Thread safety

Thread safety

- Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly

- Immutable objects are thread safe

- To make a class immutable the following can be done:
  - mark all class fields final or declare the class as final
  - ensure **this** reference is not allowed to escape during construction
  - make any fields which refer to mutable data objects private
  - don’t provide setter method

Synchronized collections

- **StringBuffer** is a thread-safe alternative of **StringBuilder**
- The **java.util.Collections** class provides methods to retrieve thread-safe collections (i.e. Collections.synchronizedSet())
- **java.util.concurrent.ConcurrentHashMap** provides a thread-safe hash map

Questions ?