Programmierparadigmen
Übung 11: Java Basic

Prof. Dr. Ralf Reussner / M.Sc. Larissa Schmid

01.02.2024

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS
dsis.kastel.kit.edu

Überblick heutige Übung

Karlsruhe Institute of Technology

- Amdahlsches Gesetz (Übungsblatt Aufgabe 1)
- Happens-before-Beziehung (Übungsblatt Aufgabe 4)
- Fortgeschrittene Parallelisierungsprinzipien (Übungsblatt Aufgabe 5)
- Beispiel Vektoraddition
- Java Advanced: Locks, Fork-Join

Amdahlsches Gesetz
(Übungsblatt Aufgabe 1)

- Programm mit mehreren Threads arbeitet lesen und schreibend auf einem Puffer
  - Lesen: Gleichzeitig möglich, beliebig viele Threads
  - Schreiben: Nur Schreib-Thread darf aktiv sein
- Gegeben: Thread-Pool, jeder Leser und Schreiber ein Thread
  - 90% Leser, 10% Schreiber
  - Lesevorgang: 2 Sekunden, Schreibvorgang: 3 Sekunden
- **Obere Grenze der Beschleunigung auf einem 4-Kern-Prozessor?**

Amdahlsches Gesetz
(Übungsblatt Aufgabe 1)

- Programm mit mehreren Threads arbeitet lesen und schreibend auf einem Puffer
  - Lesen: Gleichzeitig möglich, beliebig viele Threads
  - Schreiben: Nur Schreib-Thread darf aktiv sein
- Gegeben: Thread-Pool, jeder Leser und Schreiber ein Thread
  - 90% Leser, 10% Schreiber
  - Lesevorgang: 2 Sekunden, Schreibvorgang: 3 Sekunden
- **Obere Grenze der Beschleunigung auf einem 4-Kern-Prozessor?**

P: Anteil eines Programms, der parallelisiert werden kann
N: Anzahl der Prozessoren

$$S(P) = \frac{1}{(1-P)+\frac{P}{N}}$$

Amdahlsches Gesetz
(Übungsblatt Aufgabe 1)

- Gegeben: Thread-Pool, jeder Leser und Schreiber ein Thread
  - 90% Leser, 10% Schreiber
  - Lesevorgang: 2 Sekunden, Schreibvorgang: 3 Sekunden
- **Obere Grenze der Beschleunigung auf einem 4-Kern-Prozessor?**

P: Anteil eines Programms, der parallelisiert werden kann

N: Anzahl der Prozessoren

$$S(P) = \frac{1}{(1-P)+\frac{P}{N}}$$

$$P = \frac{(2*0.9)}{2*0.9+3*0.1} \approx 0.86$$

Mit $N = 4$ resultiert dies in: $S(P) = \frac{1}{(1-0.86)+\frac{0.86}{4}} \approx 2.82$

Happens-Before Beziehung
(Übungsblatt Aufgabe 4)

```java
public class HappensBefore {
    public static boolean ping = false;
    public static final int maxRuns = 100;

    public static void main(String[] args) {
        Thread pingThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(ping) {}

                ping = true;
                System.out.println("Ping␣-␣Round␣" + i);
            }
        });
        Thread pongThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(!ping) {}

                ping = false;
                System.out.println("Pong␣-␣Round␣" + i);
            }
        });

        pingThread.start();
        pongThread.start();
    }
}
```

- Warum terminiert das Programm manchmal nicht?
- Verwenden Sie Happens-Before Beziehungen als Grundlage für ihre Argumentation.

Happens-Before Beziehung
(Übungsblatt Aufgabe 4)

```
public class HappensBefore {
    public static boolean ping = false;
    public static final int maxRuns = 100;

    public static void main(String[] args) {
        Thread pingThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(ping) {}

                ping = true;
                System.out.println("Ping␣-␣Round␣" + i);
            }
        });
        Thread pongThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(!ping) {}

                ping = false;
                System.out.println("Pong␣-␣Round␣" + i);
            }
        });

        pingThread.start();
        pongThread.start();
    }
}
```

- Keine Happens-Before Beziehung zwischen Lesen und schreiben von *ping*
- Durch Lesen der alten Werte kein Verlassen der *while*-Schleifen

Happens-Before Beziehung
(Übungsblatt Aufgabe 4)

```
public class HappensBefore {
    public static boolean ping = false;
    public static final int maxRuns = 100;

    public static void main(String[] args) {
        Thread pingThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(ping) {}

                ping = true;
                System.out.println("Ping␣-␣Round␣" + i);
            }
        });
        Thread pongThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(!ping) {}

                ping = false;
                System.out.println("Pong␣-␣Round␣" + i);
            }
        });

        pingThread.start();
        pongThread.start();
    }
}
```

- Problemlösung:
  - Happens-Before Beziehung herstellen
  - z.B. *ping* als *volatile* deklarieren

Fortgeschrittene Parallelisierungsprinzipien
(Übungsblatt Aufgabe 5)

- *calculateMax* soll den größten Wert einer Sequenz von Integer-Zahlen parallel berechnen
  - Parameter: disjunkte Blöcke *blocksOfNumbers*
  - Rückgabewert: größter Wert aller Blöcke
- *findMax* berechnet größten Wert einer Folge von Integerzahlen

Fortgeschrittene Parallelisierungsprinzipien
(Übungsblatt Aufgabe 5)

```
public class MaxOfMax {
    public int calculateMax(Collection<List<Integer>>
        blocksOfNumbers,
                            int numberThreads) throws
                                ExecutionException,
                                InterruptedException {

        if (blocksOfNumbers.size() == 0) {
            return Integer.MIN_VALUE;
        }

        List<Integer> results = new ArrayList<>();
```

… Implementierung Lösung …

```
        return findMax(results);
    }
```

```
    private Integer findMax(Collection<Integer> numbers) {
        Integer maxValue = Integer.MIN_VALUE;
        for (Integer number : numbers) {
            if (number > maxValue) {
                maxValue = number;
            }
        }
        return maxValue;
    }
```

Fortgeschrittene Parallelisierungsprinzipien
(Übungsblatt Aufgabe 5)

```java
        List<Integer> results = new ArrayList<>();
        List<Future<Integer>> futures = new ArrayList<>();
        ExecutorService executor =
           Executors.newFixedThreadPool(numberThreads);
        for (List<Integer> numberBlock : blocksOfNumbers) {

            futures.add(executor.submit(() -> {
                return findMax(numberBlock);
            }));
        }

        for (Future<Integer> future : futures) {
            results.add(future.get());
        }

        executor.shutdown();
        return findMax(results);
    }
```

Fortgeschrittene Parallelisierungsprinzipien
(Übungsblatt Aufgabe 5)

- Implementieren Sie die Methode *findMax* mittels parallelen Java-Streams und der Stream-Operation *max*.
- Hinweis: Rückgabetyp der Stream-Operation beachten! *Stream::max* gibt *Optional*<*T*> zurück.

Fortgeschrittene Parallelisierungsprinzipien
(Übungsblatt Aufgabe 5)

- Implementieren Sie die Methode *findMax* mittels parallelen Java-Streams und der Stream-Operation *max*.
- Hinweis: Rückgabetyp der Stream-Operation beachten! *Stream::max* gibt *Optional*<*T*> zurück.

```
public Integer findMax(Collection<Integer> numbers){
        return numbers.parallelStream()
                        .mapToInt(x -> x.intValue())
                        .max()
                        .orElse(Integer.MIN_VALUE);
}
```

Example: Vector addition (1)

- We want to perform a parallel vector addition.
- The most important component is the worker class, which adds two (partial) vectors element by element and saves the result in a third vector.

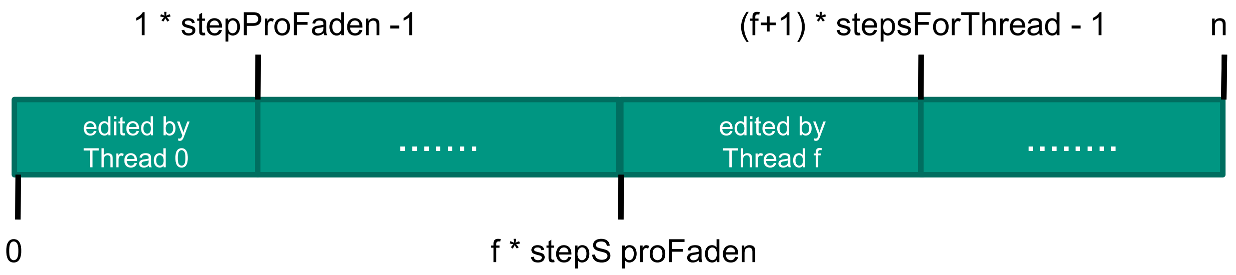
```
class Worker implements Runnable {
  private int[] a, b, c;
  private int left, right;

  public worker (int[] a, int[] b, int[] c, int left, int right) {
    this.a = a; this.b = b; this.c = c;
    this.left = left; this.right = right;
  }

  public void run() { // adds subvectors in the segment [left..right)
    for (int i = left; i < right; i++) {
      c[i] = a[i] + b[i];
    }
  }
}
```

Example: Vector addition (2)

- **Illustration:** How parallel vector addition works
- Each thread `f` processes a segment of the length `stepsProFaden`.



Example: Vector addition (2)

- The main method first defines
  - The vectors,
  - constants to divide the index range of the vectors.

```
public static void main(String[] args) {

  int[] a = ... // fill
  int[] b = ... // fill

  assert a.length == b.length;
  int[] c = new int[a.length];

  final int numberOfThreads = 10;
  final int n = a.length;
  final int stepsPerThread = (int) Math.ceil((double) n / numberOfThreads);
```

Example: Vector addition (3)

- A field is created that contains references to the threads to be started.
- In the loop, the threads are each created and started with the index range that they are to process.
  - Explain what the minimum function is used for when calculating from the `right`.
  - Can `left >= right`? Is that a problem? What do the threads do for which this is the case?

```
Thread[] team = new Thread[numberOfThreads];

for (int f = 0; f < numberOfThreads; ++f) {
  int links = f * stepsPerThreads;
  int right = Math.min((f + 1) * stepsPerThread, n);

  team[f] = new Thread(new worker(a, b, c, left, right));
  team[f].start(); //thread f is now running
}
```

Example: Vector addition (4)

- Now you have to wait for the threads to finish. This is done with `join()`
  - Note: `join()` can also be `interrupted` via `InterruptedException.`

```
for (Thread f : team) {
  try {
    f.join(); /* waits until thread ends */
  } catch (InterruptedException ex) {
    System.err.println("Unexpected interruption" +
                       "while waiting for workers");
  }
}

//Now use the result in c[]
```

Locks

- An option to realize critical sections are locks
- Java provides different implementations of a `Lock` interface [1]
  - `ReentrantLock`
  - `ReentrantReadWriteLock`
- A constructor parameter allows to specify if a lock shall be fair
  - “Unfair” locks are more performant
- Reentrant: Locked section can be entered multiple times by the same thread
  - E.g. via recursion

```java
public class Fibonacci {
  private int n1 = 0, n2 = 0, n3 = 0;
  Lock lock = new ReentrantLock(false);
  public void fibonacci(int count) {
    lock.lock();
    n3 = n1 + n2; n1 = n2; n2 = n3;
    fibonacci(count - 1);
    lock.unlock();
  }
}
```

[1] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ Lock.html

Locks: Pitfalls

```java
public void doSomething() {
        lock.lock();
        // do something
        lock.unlock();
}
```

What if this section includes a return or throws an exception?

```java
public void doSomething() {
  lock.lock();
  try {
    // do something
  } finally {
    lock.unlock();
  }
}
```

Locks: Attempted Acquisition

- A simple kind of lock in Java we have already seen are monitors
  - A `synchronized` block can be seen as a pair of lock and unlock operations on the monitor object
  - Drawback is that there is no possibility to back out of an attempt to acquire a lock
- Locks provide a `tryLock()` method, which acquires the lock if possible and returns whether it was successful
  - Useful to acquire several locks without blocking
  - Allows to avoid deadlocks by not fulfilling *hold and wait*
  - Be sure to correctly unlock the acquired locks (and none else)

Locks: Deadlock-free Locking Principle

```java
public class DeadlockFreeLock {
        Lock lock = new ReentrantLock();

        private boolean getLocks(DeadlockFreeLock other) {
                boolean myLock = false;
                boolean theirLock = false;
                try {
                        myLock = lock
                        theirLock = other.lock.tryL
                } finally {
                        if (!(myLock && theirLock)) {
                                if

                                }
                                if

                                }
                        }
                }
                return myLock && th
        }
```

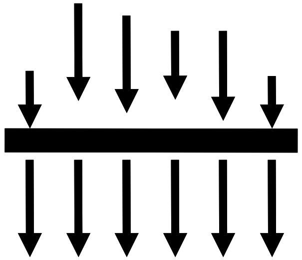
```java
        …
private void use(DeadlockFreeLock
other) {
  if (getLocks(other)) {
    // Do something
    lock.unlock();
    other.lock.unlock();
    }
  }
}
```

If tryLock() returns true, acquisition of the lock was successful

Barriers

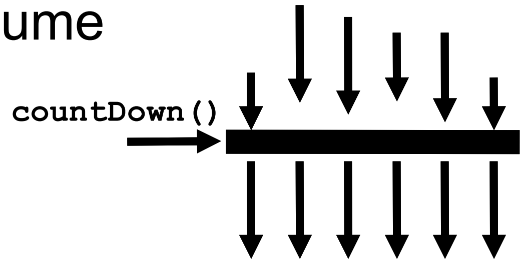
`CyclicBarrier(int n)`

- `await()` blocks the calling thread
- If `await()` was called *n* times, all threads resume
- Barrier can be reused afterwards



`CountDownLatch(int n)`

- `await()` blocks the calling thread
- If `countdown()` was called *n* times, all threads resume
- Latch cannot be restarted afterwards
- Further calls to `await()` return immediately



`CyclicBarrier` Example

```
public class BarrierDemo {
        static CyclicBarrier barrier;

        public static void main(String[] args) {
             barrier = new CyclicBarrier(2);
             new Thread(BarrierDemo::runSingleThread).start();
             new Thread(BarrierDemo::runSingleThread).start();
        }

        public static void runSingleThread() {
             try {
                         // Do first task
                         System.out.println("Reached first barrier");
                         barrier.await();
                         // Do second task
                         System.out.println("Reached second barrier");
                         barrier.await();
             } catch (InterruptedException | BrokenBarrierException e) {}
        }
}
```

CountDownLatch Example

```java
public class BarrierDemo {
	static CountDownLatch latch;

	public static void main(String[] args) {
		latch = new CountDownLatch(2);
		new Thread(BarrierDemo::runSingleThrea
		new Thread(BarrierDemo::runSingleThrea
		try {
			latch.await();
		} catch (InterruptedException e) {}
		// Do tasks that require completion of the threads
	}

	public static void runSingleThread() {
		try {
			// Do some task
		} catch (InterruptedException e) {}
		latch.countDown();
	}
}
```

Latch cannot be reused after await() has returned

Fork-Join (1)

- Fork-Join is a pattern for effectively computing divide-and-conquer algorithms in parallel
  - Problems are solved by splitting them into subtasks, solving them in parallel and finally composing the results
  - General algorithm in pseudocode:

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

Fork-Join (2)

- Java provides a `ForkJoinPool` on which `ForkJoinTasks` can be executed
- Implementations of `ForkJoinTask` must override the `compute()` method
  - `RecursiveAction` (no result) and `RecursiveTask` (returns result) are concretizations of such tasks
  - They can be executed by a `ForkJoinPool` calling its `invoke()` method

- If `MyTask` is a `ForkJoinTask`, it can be invoked as follows:

```
…
ForkJoinPool fjPool = new ForkJoinPool();
MyTask myTask = new MyTask(…);
fjPool.invoke(myTask);
…
```

Fork-Join (3)

- A task returning a value has to override `RecursiveTask`:

```
public class MyTask extends RecursiveTask<Integer> {
  private int[] array;
  private static final int THRESHOLD = 20;

  public MyTask(int[] arr) {this.arr = arr;}

  @Override
  public Integer compute() {
    if (arr.length > THRESHOLD) {
      return ForkJoinTask.invokeAll(createSubtasks())
                   .stream().mapToInt(ForkJoinTask::join).sum();
    } else {
      return processing(arr);
    }
  }

  private Collection<CustomRecursiveTask> createSubtasks() {
    //divide array into smaller parts, e.g., two halves
  }

  private Integer processing(int[] arr) {
    //do actually interesting computation, e.g., calculate average of array
} }
```

Split task if it is bigger than the threshold

Join subtasks

Adapted from: https://www.baeldung.com/java-fork-join