Karlsruhe Institute of Technology

Programmierparadigmen

Übung 2 (Reussner) – 09.01.2026

Lars König, M.Sc.

Aufgabe 1

Gegeben ist ein Thread Pool, wobei jeder Leser und Schreiber durch einen Thread repräsentiert wird. Dabei sind 90% der Threads Leser und 10% Schreiber. Jeder Leser benötigt 2 Sekunden und jeder Schreiber 3 Sekunden zur Ausführung. Das Programm ist beendet, wenn alle Threads im Pool beendet sind.

Unter Berücksichtigung des Amdahlschen Gesetzes, wo liegt die obere Grenze der Beschleunigung (Speedup) der oben beschriebenen Implementierung auf einem 4-Kern-Prozessor?

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

Aufgabe 1

Gegeben ist ein Thread Pool, wobei jeder Leser und Schreiber durch einen Thread repräsentiert wird. Dabei sind 90% der Threads Leser und 10% Schreiber. Jeder Leser benötigt 2 Sekunden und jeder Schreiber 3 Sekunden zur Ausführung. Das Programm ist beendet, wenn alle Threads im Pool beendet sind.

Unter Berücksichtigung des Amdahlschen Gesetzes, wo liegt die obere Grenze der Beschleunigung (Speedup) der oben beschriebenen Implementierung auf einem 4-Kern-Prozessor?

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

Aufgabe 1

Parallelisierbarer Anteil:

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

Anzahl der Prozessoren:

$$N = 4$$

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

Aufgabe 1

Parallelisierbarer Anteil:

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

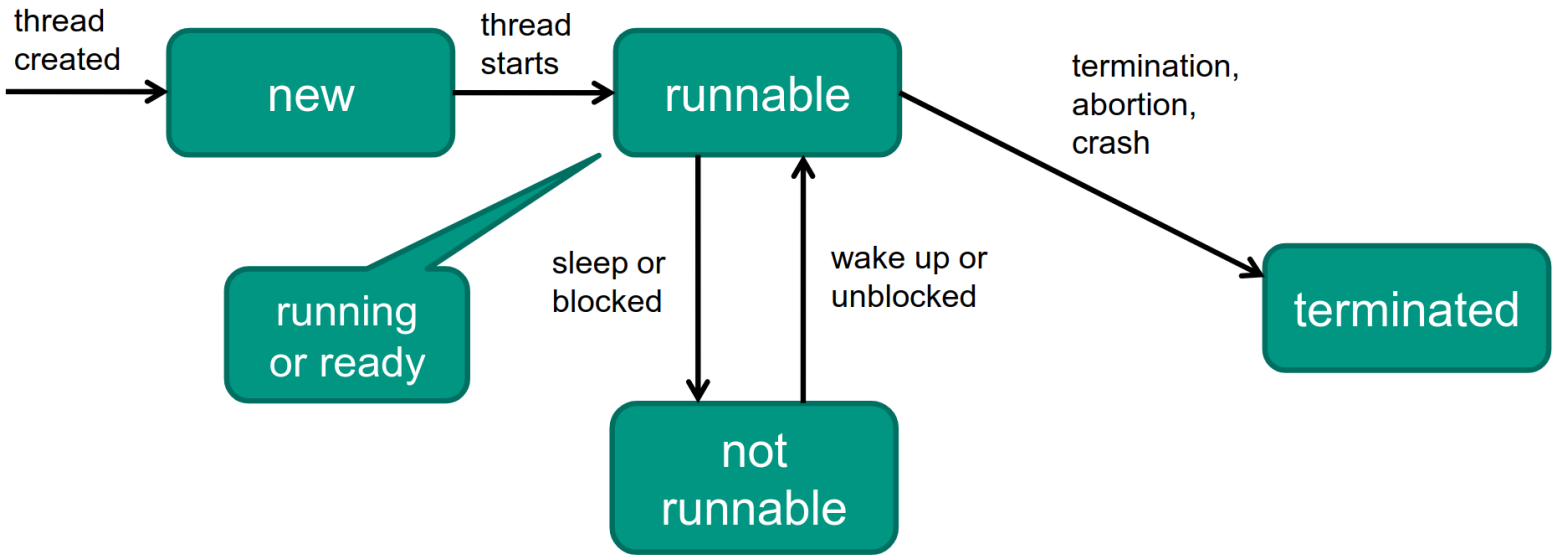
Anzahl der Prozessoren:

$$N = 4$$

$$S(P) = \frac{1}{(1 - 0.86) + \frac{0.86}{4}} \approx 2.82$$

Threads

- unabhängig ausgeführter Code
- eigener Stack
- teilt Heap mit anderen Threads *(shared memory)*



Was kann schon passieren?

- Threads können sich gegenseitig beeinflussen
  - gemeinsam genutzter Speicher
  - Koordination mit Signalen

- Welche Probleme ergeben sich daraus?
- Welche Lösungsstrategien haben Sie in der Vorlesung kennengelernt?

Mögliche Probleme

- Race Conditions
- Deadlocks
- Inkonsistente Daten (z.B. durch Reorderings oder Memory Inconsistencies)

Lösungsstrategien

- Atomare Operationen
- Synchronisation von kritischen Abschnitten
- Vorbeugen von Deadlocks
- Herstellen von happens-before-Beziehungen

Mögliche Probleme

- Race Conditions
- Deadlocks
- Inkonsistente Daten (z.B. durch Reorderings oder Memory Inconsistencies)

Lösungsstrategien

- Atomare Operationen
- Synchronisation von kritischen Abschnitten
- Vorbeugen von Deadlocks
- Herstellen von happens-before-Beziehungen

Race Conditions

```
private int c = 0;

public void increment() {
      int _c = c;  // get
      _c = _c + 1; // increment
      c = _c;      // write
}
```

Korrektes Ergebnis ist abhängig von der Ausführungsreihenfolge

Pingo

```
private int c = 0;

public void increment() {
      int _c = c;
      _c = _c + 1;
      c = _c;
}
```

Angenommen 3 Threads rufen überschneidend die Methode „increment“ auf. Welche Werte kann „c“ danach haben?

Pingo

```
private int c = 0;

public void increment() {
      int _c = c;
      _c = _c + 1;
      c = _c;
}
```

Lösung: c kann Werte zwischen 1 (jeder Thread liest den ursprünglichen Wert) und 3 (Lesen erst nachdem der vorherige Thread geschrieben hat) annehmen.

Atomare Operationen

- werden zusammenhängend ausgeführt
- keine Seiteneffekte vor der Fertigstellung

in Java:

- Lesen/Schreiben von Referenzvariablen
- Lesen/Schreiben von manchen primitiven Variablen
- Lesen/Schreiben von allen `volatile`-Variablen

- Java stellt Typen mit atomaren Operationen zur Verfügung (z.B. `AtomicInteger`)

Pingo

```
private volatile int c = 0;

public void inc() {
      c = c + 1;
}
```

Können bei Ausführung der Methode „foo“ noch race conditions auftreten?

Pingo

```
private volatile int c = 0;

public void inc() {
      c = c + 1;
}
```

Können bei Ausführung der Methode „foo“ noch race conditions auftreten?

Lösung: Ja, da volatile nur garantiert, dass die einzelnen Lese-/ Schreiboperationen atomar sind. Hier finden aber mehrere Operationen nacheinander statt.

Critical Sections

- Code-Abschnitt, der nur von einem Thread gleichzeitig betreten werden darf
- andere Threads warten am Beginn des Abschnitts
- ein Thread darf den von ihm belegten Bereich auch erneut betreten *(Reentrance)*

```
monitor.lock()

int _c = c;
_c = _c + 1;
c = _c;

monitor.unlock()
```

Critical Sections

Zu beachten

- beschränken den parallelisierbaren Teil des Programms
- können zum Bottleneck werden

```
monitor.lock()

if (valid)
      doSomething();
else
      throw new Exception();

monitor.unlock();
```

Synchronized

```
public void increment() {
      synchronized (monitor) {
            c++;
      }
}

public synchronized void increment() {
      c++;
}
```

- jedes Objekt kann als Monitor verwendet werden
- bei `synchronized`-Methoden wird `this` als Monitor verwendet

Koordination von Threads

```
while (!done) {
      synchronized (monitor) {
            if (dataAvailable) {
                  // do something
                  done = true;
            }
      }
}
```

Busy waiting führt zu hoher Auslastung und hohem Synchronisationsaufwand.

Signals

- Koordination von Threads auf einem Monitor
- Threads werden inaktiv, solange sie warten
- Ressourcen werden während des Wartens freigegeben

- `wait()` - Thread wird inaktiv
- `notify()` - weckt einen Thread auf dem aktuellen Monitor auf
- `notifyAll()` - weckt alle Threads auf dem aktuellen Monitor auf

Koordination von Threads

Busy Waiting

```
while (!done) {
        synchronized (monitor) {
                if (dataAvailable) {
                        // do something
                        done = true;
                }
        }
}
```

Signals

```
synchronized (monitor) {
        while (!dataAvailable) {
                monitor.wait();
        }
        // do something
}
```

Koordination von Threads

```
synchronized (monitor) {
	while (!dataAvailable) {
		try {
			monitor.wait();
		} catch (InterruptedException e) {
			Thread.currentThread().interrupt();
			return;
		}
	}
	// do something
}
```

Signals – Zu Beachten

- `notify()` und `notifyAll()` werden nicht gespeichert
- `wait()` sollte immer eine umgebende Schleife haben, die den Grund für das Aufwecken überprüft
- das Betriebssystem kann Threads ”zufällig” aufwecken
- `notifyAll()` ist meist sicherer als `notify()`
- keine Fairness beim Aufwecken der wartenden Threads

Aufgabe 3

Verschachtelte Kritische Abschnitte

```
public void copy() {
	synchronized (read) {
		synchronized (write) {
			// do something
		}
	}
}

public void log() {
	synchronized (write) {
		// do something
		copy();
	}
}
```

Wodurch kann hier ein Problem entstehen?

Mögliche Probleme

- Race Conditions
- Deadlocks
- Inkonsistente Daten (z.B. durch Reorderings oder Memory Inconsistencies)

Lösungsstrategien

- Atomare Operationen
- Synchronisation von kritischen Abschnitten
- Vorbeugen von Deadlocks
- Herstellen von happens-before-Beziehungen

Deadlocks

- mehrere Threads warten aufeinander
- dauerhaft kein Fortschritt mehr

Coffman-Bedingungen:

- Mutual exclusion: Ressourcen werden nicht geteilt
- Hold and wait: ein Thread der eine Ressource hält kann auf eine weitere warten
- No preemption: nur der Thread selbst kann seine Ressourcen freigeben
- Circular wait: Thread *a* wartet auf Thread *b* ... wartet auf Thread *a*

Deadlocks

Welche der Coffman-Bedingungen können bei der Benutzung von `synchronized`-Blöcken erfüllt sein? Welche sind immer erfüllt?

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlocks

Welche der Coffman-Bedingungen können bei der Benutzung von `synchronized`-Blöcken erfüllt sein? Welche sind immer erfüllt?

- Mutual exclusion — immer erfüllt
- Hold and wait — kann erfüllt sein
- No preemption — immer erfüllt
- Circular wait — kann erfüllt sein

Zusammenfassung

Mögliche Probleme

- Race Conditions
- Deadlocks
- Inkonsistente Daten (z.B. durch Reorderings oder Memory Inconsistencies)

Lösungsstrategien

- Atomare Operationen
- Synchronisation von kritischen Abschnitten
- Vorbeugen von Deadlocks
- Herstellen von happens-before-Beziehungen

Zusammenfassung

Fortgeschrittene Strategien

- Synchronisationskonstrukte (Barrieren, Semaphoren, …)
- Futures
- Streams
- Aktoren