Einleitung

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 2. November 2020

Übersicht

Warum ist Nebenläufige Programmierung interessant?

Unterschied: sequentiell – nebenläufig
Anwendungsbereiche nebenläufiger Programmierung
Schwierigkeiten bei nebenläufiger Programmierung

Sequentielle und nebenläufige Programmierung

Sequentielle Programme:

- Folge von (Maschinen-)Befehlen
- Befehle werden nacheinander ausgeführt
- Jeder Befehl ändert Hauptspeicher, Register, etc.
- Ausführung deterministisch

Sequentielle und nebenläufige Programmierung (2)

Nebenläufige (parallele) Programme:

- Mehrere Befehle werden gleichzeitig durchgeführt.
- gleichzeitig kann auch quasi-parallel bedeuten, d.h. in Realität sequentiell
- Ausführung i.a. nichtdeterministisch
- formalere Definition: später

Reale Systeme sind fast nie rein sequentiell

Beispiel Betriebssysteme:

- aus Benutzersicht: verschiedene Aufgaben werden gleichzeitig durchgeführt
- z.B. Drucken, Surfen, Mausbewegen, Musik hören (alles gleichzeitig)
- Umsetzung: ohne nebenläufige Programmierung undenkbar

Reale Systeme fast nie rein sequentiell (2)

Beispiel Web-Programming:

- Client-Server Modell: Client fordert Dienst beim Server an
- z.B. WWW-Server: Client fordert Webseite an
- Notwendig: Server bedient mehrere Clients gleichzeitig
- Absturz eines Clients, führt nicht zum Hängenbleiben anderer Clients

Nebenläufige Programmierung wird gebraucht

Auch auf Einprozessor-Systemen:

- Durch Nebenläufigkeit können Ressourcen manchmal besser genutzt werden
- Beispiel: 2 Aufgaben
  - Lange Berechnung
  - Große Datei schreiben
- Datei schreiben belastet nicht die CPU (Geschwindigkeit wird von der Festplatte / Bus bestimmt)
- Nebenläufigkeit: Wenn Festplatte busy, dann rechne.
- Nebenläufige Programmierung als Strukturierungsprinzip

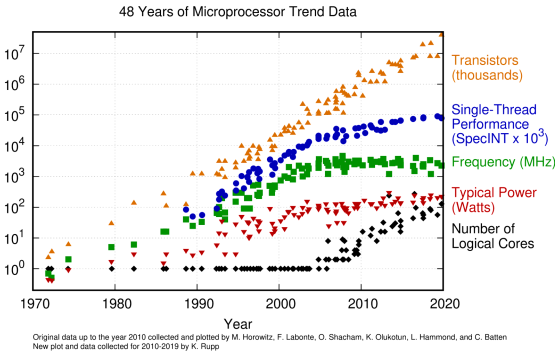
Nebenläufige Programmierung wird gebraucht (2)

Hartnäckige Probleme und große Probleminstanzen:

- Berechnung durch massive Parallelisierung / verteilte Berechnung
- Hochleistungsrechner, “GRID-Computing”

Nebenläufige Programmierung wird gebraucht (3)

Entwicklung der Hardware:



Quelle: https://github.com/karlrupp/microprocessor-trend-data

- Taktfrequenzen am Rande ihre physikalischen Möglichkeiten
- Statt Takterhöhung Erhöhung der Prozessoranzahl
- Multicore-Architekturen
- Architektur verlangt angepasste (parallelisierte) Programme
- Artikel von Herb Sutter: Multiprozessorsysteme verlangen einen Paradigmenwechsel für Programmiersprachen

Was macht nebenl. Programmierung so schwer?

- Parallelisierung sequentieller Algorithmen oft nicht offensichtlich.
- Hauptproblem: Die parallel ablaufenden Programme müssen korrekt zusammenarbeiten.
- Daten austauschen (Kommunikation)
- Aufeinander warten (Synchronisation)

Beispiel: Kontoführung

(Der Wert von `konto` vor der Ausführung sei 100)

Prozess P:	Prozess Q:
(P1) X:= konto;	(Q1) X := konto;
(P2) konto:= X-10;	(Q2) konto := X+10;

Beispiel: Kontoführung (2)

Prozess P:	Prozess Q:
(P1) X:= konto;	(Q1) X := konto;
(P2) konto:= X-10;	(Q2) konto := X+10;

Ausführung (P1),(P2),(Q1),(Q2)
konto = 100

Beispiel: Kontoführung (3)

Prozess P:	Prozess Q:
(P1) X:= konto;	(Q1) X := konto;
(P2) konto:= X-10;	(Q2) konto := X+10;

Ausführung (Q1),(P1),(P2),(Q2)
konto = 110

Beispiel: Kontoführung (4)

Prozess P:	Prozess Q:
(P1) X:= konto;	(Q1) X := konto;
(P2) konto:= X-10;	(Q2) konto := X+10;

Ausführung (Q1),(P1),(Q2),(P2)
konto = 90

Beispiel: Kontoführung (5)

Analyse ergibt:

Reihenfolge	Wert von konto danach
(P1),(P2),(Q1),(Q2)	100
(Q1),(Q2),(P1),(P2)	100
(P1),(Q1),(P2),(Q2)	110
(Q1),(P1),(P2),(Q2)	110
(P1),(Q1),(Q2),(P2)	90
(Q1),(P1),(Q2),(P2)	90

Fazit:

- Falsch programmiert.
- Traditionelles Debuggen funktioniert nicht
- Ergebnis kann von Ausführung zu Ausführung abweichen

Verbraucher-Erzeuger Probleme

Annahme: Es gibt (Daten) erzeugende Prozesse und (Daten) verbrauchende Prozesse

Verschiedene Varianten:

- Mehrere Verbraucher und ein Erzeuger
- Mehrere Erzeuger und ein Verbraucher
- Mehrere Erzeuger und mehrere Verbraucher

Gesucht:

Datenstrukturen um sicheren Austausch der Daten zwischen Verbrauchern und Erzeugern zu gewährleisten.

Deswegen: Nebenläufige Programmierung benötigt neue Datenstrukturen

Begriffe der nebenläufigen Programmierung

Definition und Abgrenzung wichtiger Begriffe

Begriffe der nebenläufigen Programmierung

Parallelität und Nebenläufigkeit

Paralleles Programm:

- Berechnung auf mehreren Prozessoren
- Gleichzeitig, überlappend

Nebenläufige Programmierung (engl. concurrent programming):

- Ausführung auf mehreren Prozessoren ein Szenario
- Potentiell sind alle Ausführungsreihenfolgen möglich.

Begriffe der nebenläufigen Programmierung (2)

Nebenläufige und verteilte Systeme

Verteiltes System:

- System aus mehreren Prozessoren (oft auch örtlich getrennt)
- Kein gemeinsamer Speicher
- Datenaustausch: Alleinig über Senden und Empfangen von Nachrichten.

Nebenläufiges System:

- Berechnung auf einem oder mehreren Prozessoren
- Gemeinsamer Speicher möglich

Begriffe der nebenläufigen Programmierung (3)

Prozesse und Threads

- **Prozess** wird in der Theorie verwendet (z.B. im π-Kalkül)
- Oft Unterscheidung anhand der Kontrolle:
  - Betriebssystem verwaltet **Prozesse**
  - Programme verwalten **Threads**
  - wobei Programm ist Betriebssystem-Prozess
- Wir trennen nicht strikt zwischen beiden Begriffen.
- **Multi-Threading** = Eigenschaft von Programmiersprachen: Konstrukte zur Verwaltung von Threads
- Verwandt: **Multi-Tasking** = Möglichkeit mehrere Aufgaben quasi-parallel durchzuführen

Begriffe der nebenläufigen Programmierung (4)

Message-Passing- und Shared-Memory Modell

Message-Passing-Modell:

- Prozesse haben keinen gemeinsamen Speicher.
- Kommunikation **ausschließlich** über Senden und Empfangen von Nachrichten.
- Passt zu verteilten Systemen

Shared-Memory-Modell:

- Prozesse verwenden auch gemeinsamen Speicher.
- Kommunikation findet direkt über den gemeinsamen Speicher statt.

Begriffe der nebenläufigen Programmierung (4)

Synchrone und asynchrone Kommunikation

Synchron:

- Kommunikation zwischen Sender und Empfänger geschieht "ohne Zeit"
- Beispiel: Telefon

Asynchron:

- Senden und Empfangen muss nicht gleichzeitig stattfinden.
- Oft natürlicher.
- Beispiel: Briefe oder Emails schreiben ist asynchron.

Modellannahmen

Wie sehen Programme aus, wie werden sie ausgeführt?

Modellannahmen: Nebenläufiges Programm

Nebenläufiges Programm

- Nebenläufiges Programm: *Endliche Menge von Prozessen*
- Prozess: *Sequentielles Programm aus atomaren Berechnungsschritten*

- Annahme für alle Modelle, die wir verwenden
- Definition von Berechnungsschritt modellabhängig!

Beispiel:

Programm mit 3 Prozessen

Prozess P	Prozess Q	Prozess R
(P1) $X := 10;$	(Q1) $Y := 3;$	(R1) $X := 0;$
(P2) `while` $X > 0$ `do`	(Q2) `while` $Y > 0$ `do`	(R2) $Y := 0;$
(P3) $X := X + 1;$	(Q3) $X := Y - 1;$	
	(Q4) $Y := X;$	

Modellannahmen: Interleaving-Annahme

Interleaving-Annahme

Ausführung eines nebenläufigen Programms:

Sequenz der atomaren Berechnungsschritte der Prozesse, die beliebig verzahnt sein können.

- Nur ein Berechnungsschritt gleichzeitig, d.h. zwei Schritte überlappen nie
- (Fast) beliebiges Verzahnen (Interleaving) von Berechnungsschritten der einzelnen Prozesse
- (Fast) alle Ausführungsreihenfolgen erlaubt
- „Fast“ wird später erklärt

Beispiel

Programm mit 3 Prozessen

Prozess P	Prozess Q	Prozess R
(P1) $X := 10;$	(Q1) $Y := 3;$	(R1) $X := 0;$
(P2) `while` $X > 0$ `do`	(Q2) `while` $Y > 0$ `do`	(R2) $Y := 0;$
(P3) $X := X + 1;$	(Q3) $X := Y - 1;$	
	(Q4) $Y := X;$	

Eine mögliche Ausführung:

Schritt	Prozess	ausgeführte Operation	Speicher
			$X \mapsto 0, Y \mapsto 0$
1	P1	$X := 10$	$X \mapsto 10, Y \mapsto 0$
2	P2	`while` $X > 0$?	$X \mapsto 10, Y \mapsto 0$
3	R1	$X := 0$	$X \mapsto 0, Y \mapsto 0$
4	P3	$X := \underline{X + 1}$	$X \mapsto 0, Y \mapsto 0$
5	P3	$X := 1$	$X \mapsto 1, Y \mapsto 0$
6	Q1	$Y := 3$	$X \mapsto 1, Y \mapsto 3$

...

Schreibweise oft einfach: $(P1), (P2), (R1), (P3), (P3), (Q1), \ldots$

Warum ist die Interleaving-Annahme sinnvoll?

Granularität

- Berechnungsschritte können beliebig definiert werden (d.h. beliebig klein!)

Multitasking-Systeme

- Einprozessor-Systeme mit Multitasking machen Interleaving
- Allerdings: Reale Implementierungen benutzen meist Verfahren (z.B. Zeitscheiben), die nicht jede Reihenfolge zulassen.
- “Prozess führt eine Zeit lang Schritte aus, dann wird gewechselt”
- Aber: “eine Zeit lang” problematisch:
  - Formal schwer zu fassen, kompliziert
  - Neue (schnellere) Hardware erfordert Modellanpassung
  - Berücksichtigung von Umwelteinflüssen (z.B. Phasenschwankungen)
- Alle Reihenfolgen umfassen auch die kleinere Menge der realen Reihenfolgen (Korrektheit aller impliziert Korrektheit eines Teils)

Warum ist die Interleaving-Annahme sinnvoll? (2)

Multiprozessor-Systeme

- Annahme scheint unrealistisch, da parallele und überlappende Berechnungsschritte hier möglich sind
- **Aber:** Gemeinsame Ressourcen (z.B. Speicher), sind auf Hardwareebene vor parallelen Zugriffen geschützt und erzwingen Sequentialisierung
- Ressourcen-autonome Schritte sind immer noch parallel
- Das ist aber nicht sichtbar!

Modellannahmen: Die Fairness-Annahme

- vorhin: . . . Fast beliebiges Interleaving . . .
- Die Einschränkung kommt durch die Fairness-Annahme
- Verschieden starke Annahme von Fairness
- Unser Begriff relativ schwach

Fairness-Annahme

Jeder Prozess für den ein Berechnungsschritt möglich ist, führt in der Gesamt-Auswertungssequenz diesen Schritt nach endlich vielen Berechnungsschritten durch.

Beispiel zur Fairness

(X am Anfang 0)

Prozess P:	Prozess Q:
(P1) `if` $X \neq 10$ `then goto` (P1);	(Q1) X:=10;

Auswertungssequenz die keine Fairness beachtet:

$(P1), (P1), (P1), (P1), (P1), (P1),$ . . . unendlich so weiter

Beachte: Die Fairness-Annahme macht nur eine Aussage über unendliche Auswertungen

Auswertungssequenzen unter Beachtung von Fairness

$\underbrace{(P1), (P1), \ldots, (P1)}_{n\text{-mal}}(Q1)(P1)$ für beliebiges $n \in \mathbb{N}_0$

Beispiel zur Fairness (2)

(X am Anfang 0)

Prozess P:	Prozess Q:
(P1) `if` $X \neq 10$ `then goto` (P1);	(Q1) X:=10;

Folgerung:

Unter Einhaltung der Fairness-Annahme gilt:
Obiges Programm terminiert immer
(Da (Q1) irgendwann ausgeführt werden muss)

Noch ein Beispiel zur Fairness

(Der Wert von X vor der Ausführung sei 0)

Prozess P:	Prozess Q:
(P1) `if` X $\neq$ 10 `then goto` (P1); (P2) `goto` (P1)	(Q1) X:=10;

Erlaubte Sequenzen

$\underbrace{(P1), (P1), \ldots, (P1)}_{n\text{-mal}}, (Q1), \underbrace{(P1), (P2), (P1), (P2), \ldots}_{\text{unendlich oft}}$ mit $n \in \mathbb{N}_0$

Verbotene Sequenzen

$(P1), (P1), (P1), \ldots$ unendlich lange

Weitere Annahmen

Bekannte Prozesse

- Ein Programm besteht aus einer festen Anzahl von Prozessen
- Es werden keine Prozesse vom Programm neu erzeugt
- Prozesse sind identifizierbar

Beachte:

Diese Annahme werden wir nicht für die ganze Vorlesung beibehalten!

Weitere Annahmen (2)

Programmiersprache

- Zunächst gehen wir von einer kleinen Pseudo-Programmiersprache aus, die imperativ ist.
- Befehle: `if-then`, `goto`, Zuweisungen $X := 10$, `while`-Schleifen usw.
- Jede nummerierte Zeile wird atomar ausgeführt.
- erlaubte Operationen auf dem Speicher werden wir festlegen

Einschub: Threads in Java

- Leichtgewichtige Threads nativ eingebaut (Klasse `Thread`)
- Zwei Ansätze zum Erzeugen von Threads:
  - Unterklasse von `Thread`
  - Über das Interface `Runnable`

Unterklasse von Thread

- Wesentliche Methode: `run`
- Wird beim Thread-Start ausgeführt
- Analog zur `main`-Methode in Java

Beispiel:

```
class EinThread extends Thread {
    public void run() {
        System.out.println("Hallo vom Thread " + this.getId());
    }

}

public class Main {
     public static void main(String args[]) {
          for (int k = 1; k <= 10; k++) {
               (new EinThread()).start();
             }
     }
}
```


Interface Runnable

- Methode `run` muss implementiert werden
- Aber keine Unterklasse von `Thread`
- stattdessen: Objekt dem Konstruktor von `Thread` übergeben

```
class EinThread implements Runnable {
  public void run() {
    System.out.println("Hallo vom Thread " +
                       (Thread.currentThread()).getId());
    }
}

public class Main {
  public static void main(String args[]) {
    for (int k = 1; k <= 10; k++) {
      (new Thread(new EinThread())).start();
    }
  }
}
```


Warten

- Methode der Klasse `Thread`: `sleep`(Millisekunden)
- Muss InterruptedException abfangen

```
class EinThread implements Runnable {
  public void run() {
    long myThreadId = (Thread.currentThread()).getId();
      try { (Thread.currentThread()).sleep(myThreadId*100);}
      catch (InterruptedException e) { };
        System.out.println("Hallo vom Thread " + myThreadId);
    }
}
public class Main {
public static void main(String args[]) {
  for (int k = 1; k <= 10; k++) {
     (new Thread(new EinThread())).start();
   }
  }
}
```


Warten auf Thread-Ende

Methode `join`: Warten auf die Terminierung eines Threads.

Wenn t_1 einen Aufruf t_2`.join()` macht, dann wartet t_1 solange bis, t_2 terminiert ist.

Interrupted-Exceptions muss abgefangen werden mit `try-catch`-Block.

Volatile Variablen

- Der Qualifier `volatile` für Variablen kennzeichnet eine Variable auf die verschiedene Threads zugreifen
- Die Java VM sichert dann zu, dass Werte der Variablen nicht gecached werden, sondern es „einen Wert“ im Hauptspeicher gibt
- Kein Synchronisationsmechansimus oder Schutz vor gleichzeitigem Zugriff!