Programmiersprachen
Vorlesung 7 vom 29.11.21
Nebenläufigkeit

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

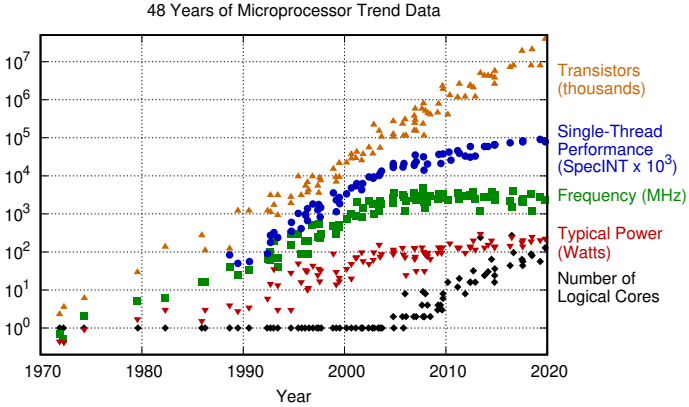
- Einführung
- Werte und Typen
- Anweisungen, Variablen und Zustand
- Kontrollabstraktion
- Datenabstraktion
- Fortgeschrittene Typsysteme
- Nebenläufigkeit
- Objektorientierung
- Skriptsprachen
- Beispielsprache II
- Ab Woche 11: Studentische Vorträge.

Nebenläufigkeit

Warum Nebenläufigkeit?

- Abstraktion:
  - Die Welt **ist** nebenläufig.
  - Viele Anwendungen reagieren
    - auf Eingaben in undefinierter Reihenfolge,
    - zu unvorgesehenen Zeitpunkten.
    - Beispiel: Webanwendungen
  - Architekur der Anwendung muss diese Struktur reflektieren

Warum Nebenläufigkeit?



- Performance:
  - Ausnutzung mehrerer Cores
  - Ausnutzung von **Latenz** (Warten auf externe Ereignisse)

Sequentielle Architektur.

- Die Von-Neumann Architektur:
  - Programme und Daten in einem Speicher
  - Berechnung in Zyklus fetch — compute — store
- Formale Beschreibung:
  - Turing-Maschine
  - Interner Zustand plus Arbeitsspeicher (Band)

Nicht-sequentielle Modelle:

- Nicht alle Berechnungsmodelle sind sequentiell.
- Beispiel: λ-Kalkül, Gleichungsreduktion

```
inc :: Int → Int
inc x = x+1
```

```
mdbl :: Int → Int → Int
mdbl x y = 2* x * y
```

- Im Ausdruck `inc (mdbl (inc 4) (inc 2))` sind mehrere Reduktionen möglich.
  - Innermost-first, outermost-first (s. vorletzte Vorlesung)
  - ... oder sogar parallel (`inc 4` und `inc 2`)

Implementationsaspekte

Prozesse vs. Threads

- Prozesse:
  - Schwergewichtig, Erzeugung teuer
  - Eigener Speicherbereich, eigene Ressourcen
  - Vom Betriebssystem verwaltet
- Threads:
  - Leichtgewichtig, Erzeugung billig
  - Geteilter Speicherbereich und Ressourcen
  - Von Programmierer/Programmiersprache verwaltet
- Uns interessieren hier **Threads**

Arten der Nebenläufigkeit:

- **true concurrency** vs. **interleaved concurrency**
  - true concurrency: Programme laufen **gleichzeitig**, z.B. threads auf einer Multi-Core-CPU
  - interleaved concurrency: Programme laufen auf einer zentralen Berechungseinheit
- **Kooperativ** vs. **präemptiv**
  - Kooperativ: Programme geben explizit Kontrolle ab
  - Präemptiv: Betriebssytem/Scheduler unterbricht Programme, verteilt Kontrolle.

Probleme der Nebenläufigkeit

- Nicht-Determinismus
- **Deadlock**
  - Nichts geht mehr — gegenseitige Blockade
  - Beispiel: Straßenkreuzung, Bewerber aus dem Ausland
- **Starvation**
  - Ein Thread wird “ausgehungert”
  - Beispiel: Auto in Nebenstraße zu Hauptverkehrsstraße

Operationen für Threads

- Erzeugung — neuen Thread starten
  - Erhält auszuführenden Rumpf als Argument
  - Jeder Thread hat eine eindeutige `id`
- Beendigung
  - Beendet **laufenden** Thread
- Ggf. Kontrollübergabe
  - Für kooperative Nebenläufigkeit
  - Meist implizit in I/O-Operationen
- **Synchronisation**
  - Auf andere Threads warten, andere Threads stoppen, Nachrichten senden und empfangen
  - Hier wird es interessant...

Konzepte der Nebenläufigkeit

Konzepte

- Konzepte zur Nebenläufigkeit in Programmiersprachen dienen zwei Zwecken:
  1. Verhinderung von Interferenz
  2. Kommunikation

Definition: Kritischer Abschnitt

Ein **kritischer Abschnitt** (in Bezug auf eine Resource r) ist ein Teil des Programmes, in kein anderer Thread/Prozess auf r zugreifen darf.
In Bezug auf die Resource CPU: in dem kein anderer Thread/Prozess laufen darf.

Kritische Abschnitte I: Spin-Locks.

- Spin-Locks sind ein Beispiel für **Mutexe**
  - Sicherstellung des gegenseitigen Ausschluß (mutual exclusion)
- Spin-Locks sind "busy-waiting loops"
- Benutzung mit `acquire(r)` und `release(r)`:

```
... non-critical code ...
acquire(r);
// CRITICAL SECTION
release(r);
... more non-critical code ...
```

- `r` ist hier der Parameter, der die Resource identifiziert.
  - Ohne `r`: kein anderer Thread darf laufen

Dekker's Algorithmus

- Implementiert `acquire(r)` und `release(r)`
- Setup:
  - Zwei Threads, jeweils `self` und `other`.
- Implementation in vier Versuchen nach Djikstra (1968a)
- Erster Versuch:

```
acquire(r) =
  while turn = other loop null: end loop;

release(r) =
  turn := other;
```

- Stellt Ausschluss sicher
- Problem: Prozesse **müssen** alternieren

Dekker's Algorithmus: Zweiter Versuch

- Zweiter Versuch:
  - Nutzt Array ein `claimed[2]`

```
acquire(r) =
  while claimed[other] loop null; end loop;
  claimed[self]:= true;

release(r) =
  claimed[self]:= false;
```

- Problem: Gegenseitiger Ausschluss nicht garantiert:
  - Thread 1 findet `claimed[2] == false` und will gerade `claimed[1]` auf `true` setzen...
  - ... als Thread 2 `claimed[1] == false` findet und den kritischen Abschnitt betritt ....
  - ... Thread 1 ist wieder dran, setzt `claimed[1]` auf `true` aber zu spät.

Dekker's Algorithmus: Dritter Versuch

- Dritter Versuch:

```
acquired(r) =
  claimed[self]:= true;
  while claimed[other] loop
    claimed[self] := false;
    while  claimed[other] loop null; end loop;
    claimed[self] := true;
  end loop;
```

- Problem: Deadlock bei **gleichzeitiger** Ausführung
  - Kontextwechsel nach jeweils einer Anweisung

Dekker's Algorithmus: Korrekte Fassung

- Korrekte Fassung:

```
acquired(r) =
  claimed[self]:= true;
  while claimed[other] loop
    if turn = other then
      claimed[self] := false;
      while  claimed[other] loop null; end loop;
      claimed[self] := true;
    end if;
  end loop;

release(r) =
  turn := other;
  claimed[self]:= false;
```

- Problem: korrekt, aber schlecht auf n Threads zu verallgemeinern.

Peterson's Algorithmus

- Peterson's Algorithmus: einfacher, verallgemeinert Dekker:

```
acquire(r) =
  claimed[self] := true;
  turn := other;
  while claimed[other] and turn = other
    loop null; end loop;

release(r) =
  claimed[self]:= false;
```

Fazit: Dekker und Peterson

- Algorithmen zeigen, wie kompliziert Nebenläufigkeit ist...
- Spin-Locks verschwenden CPU-Zeit.
- Algorithmen haben Voraussetzungen:
  - Compiler optimiert keine Zugriffe
  - Änderungen werden **sofort** für **alle** Prozessoren sichtbar
- Weiteres: Simpson's Algorithmus

Semaphoren

- Verallgemeinerung von Spin-Locks nach Dijkstra
- Drei Operationen:
  - Initialisierung mit Wert n – gibt an, wieviele Prozesse **maximal** im kritschen Abschnitt sein dürfen.
  - Warten (`wait`, `p`) — Betritt kritischen Abschnitt, kann blockieren wenn gewartet werden muss
  - Freigeben (`signal`, `v`) — Verläßt kritischen Abschnitt, kann anderen Thread freigeben
- Invariante (für Semaphore s):

$$0 \leq s.waits \leq s.signals + s.initial$$

Semaphoren: Beispiel

- Ein geteilter Buffer (e.g. Speicherbereich)
- Gemeinsamer Teil:

```
Semaphore full;
Semaphore empty;
char buf[BUFFERSIZE]; // Actual type irrelevant

sema_init(full, 0);
sema_init(empty, 1);
```

Sender:

```
for (;;) {
  sema_wait(empty);
  write(buf);
  sema_signal(full);
}
```

Empfänger:

```
for (;;) {
  sema_wait(full);
  read(buf);
  sema_signal(empty);
}
```

Kontrollabstraktionen

Events und Messages

- Prozesse (oder Threads) senden **Nachrichten** auf **Kanälen**
- Grundlegende Funktionen:
  - Kanal erstellen (wenn möglich getypt, ggf. mit initialer Kapazität)
  - Auf Kanal Nachricht **senden** (blockiert wenn Kanal voll, oder Fehler)
  - Aus Kanal Nachricht **empfangen** (blockiert wenn Kanal leer)
- Damit fortgeschrittene Funktionen möglich:
  - Testen ob Nachricht empfangen werden kann
  - Aus mehreren Kanälen lesen
- Abstraktion über zugrundeliegenden Transportmechanismus:
  - Shared memory, pipes, Sockets (im Filesystem oder über das Netz),...
- Diverse Implementierungen

Monitore

- Zuerst in Concurrent Pascal und Modula-2 (Niklaus Wirth)
- Monitore kombinieren
  - gegenseitigen Ausschluss mit
  - Kommunikation und
  - Verkapselung.
- Modern: `synchronized` in Java

```
DEFINITION MODULE Processes;
  TYPE SIGNAL;

  PROCEDURE StartProcess(P: PROC; n: INTEGER);
    (* startet einen nebenläufigen Prozeß mit Programm P
       und einem Arbeitsspeicher der Größe n. PROC ist
       ein Standardtyp, definiert durch PROC = PROCEDURE *)

  PROCEDURE SEND(VAR s: SIGNAL);
    (* startet einen auf s wartenden Prozeß wieder *)

  PROCEDURE WAIT(VAR s: SIGNAL);
    (* wartet auf einen anderen Prozeß, der s sendet *)

  PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
    (* Awaited(s) = "mindestens ein Prozeß wartet auf s" *)

  PROCEDURE Init(VAR s: SIGNAL);
    (* zwingende Initialisierung *)
END Processes.
```

Aus: Niklaus Wirth, Programmieren in Modula-2, Springer 1991.

Aktoren

- Grundlegendes Berechnungsmodell nach Hewitt, Bishop, Steiger
- Hier: Kontrollabstraktion für Nebenläufigkeit
- Aktoren verarbeiten Nachrichten

Während ein Aktor eine Nachricht verarbeitet, kann er

- neue Aktoren erzeugen,
- Nachrichten an bekannte Aktor-Referenzen versenden,
- festlegen, wie die nächste Nachricht verarbeitet werden soll.

Ein Aktor darf nicht

- auf einen **globalen** Zustand zugreifen,
- **veränderliche** Nachrichten versenden,
- irgendetwas tun, während er keine Nachricht verarbeitet.

Aktoren: Nachrichten

- Nachrichten sind **unveränderliche** Daten, **reine** Funktionen oder **Futures**
- Die Zustellung von Nachrichten passiert höchstens einmal (Best-effort)
  - Wenn z.B. die Netzwerkverbindung abbricht, wird gewartet, bis der Versand wieder möglich ist
  - Wenn aber z.B. der Computer direkt nach Versand der Nachricht explodiert (oder der Speicher voll läuft), kommt die Nachricht möglicherweise niemals an.
- Über den Zeitpunkt des Empfangs kann keine Aussage getroffen werden (Unbounded indeterminacy)
- Über die Reihenfolge der Empfangenen Nachrichten wird im Aktorenmodell keine Aussage gemacht (In vielen Implementierungen allerdings schon)
- Nachrichtenversand $\neq$ (Queue | Lock | Channel | ...)

Beispiel: Aktorenmodell in Erlang

- Aktorenmodell implementiert in Erlang und Akka (Bücherei für Scala/Java).
- Erlang:
  - Jede Funktion ein Aktor
  - Schwach getypt
  - Nachrichten senden mit `!`
  - Nachrichten empfangen mit `receive`
- Beispiel: `pingpong.erl`

Weitere Abstraktionsmechanismen

- Rendezvous (z.B. CSP)
- Remote Procedure Call (implemetationsnah)
- Futures (Promises)
- Software Transactional Memory (sehr abstrakt)

Implementationen

C

- Sprache selber streng sequentiell, Unterstützung durch externe Büchereien (i.e. nicht Teil des Standards)
- Ausnahme: `volatile`
  - C-Standard (C-90), 6.7.3.5 Type qualifiers:
    An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine [...].
    Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.

Volatile

- Warum unterstützt `volatile` Nebenläufigkeit?
- Ohne `volatile` könnte der Compiler Optimierungen vornehmen.
- Siehe Peterson's Algorithmus:

```
acquire(int r) {
  claimed[self]= 1;
  turn = other;
  while (claimed[other] && turn == other);
}
```

- Compiler könnte `turn = other` aus Schleifenbedingung entfernen
- `volatile` verhindert diese Optimierung
- Auch in Java (mit der gleichen Bedeutung)

Python

- Sprache selber unterstützt keine Nebenläufigkeit
- Global Interpreter Lock (für CPython)
- Nebenläufigkeit nur durch Büchereien:
  - Threads `threading`
  - Asynchrone IO `asyncio` (Coroutinen)

Java

- Per design nebeneläufig: Threads werden durch JVM unterstützt
- Threads relativ schwergewichtig
- Synchronisation durch Monitore (`synchronized`)
- Beispiel: `threads.java`

Haskell

- **Sequentielles** Haskell: Reduktion eines Ausdrucks
- **Nebenläufiges** Haskell: Reduktion eines Ausdrucks an **mehreren Stellen**
  - `ghc` implementiert Haskell-Threads
  - Zeitscheiben (Default 20ms), Kontextwechsel bei Heapallokation
  - Threaderzeugung und Kontextswitch sind **billig**
- Wenige Basisprimitive, darauf aufbauend Abstraktionen
- Synchronisation mit `MVars`
- Erstes Beispiel: `SimpleConcurrent1.hs`

Synchronisationsmechanismus: MVars

- `MVar a` ist **polymorph** über dem Inhalt
- Entweder **leer** oder **gefüllt** mit Wert vom Typ `a`
- Verhalten beim Lesen und Schreiben:

Zustand vorher:	leer	gefüllt
Lesen	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)

- **Aufwecken** blockierter Prozesse **einzeln** in **FIFO**

Zusammenfassung

Zusammenfassung

- Nebenläufigkeit tut not weil:
  - die Welt ist nebenläufig (Abstraktionsaspekt),
  - es geht schneller (Performanceaspekt).
- Nebenläufigkeitsabstraktionen:
  - Spin-Locks
  - Semaphoren
  - Monitore
  - Aktoren
- Unterstützung in Programmiersprachen: dürftig (C, Python), sehr gut (Java, Haskell)