# Nebenläufigkeit
# und
# Asynchronizität
# in
# eingebetteten Systemen

*Klaus Kusche, September 2012*

# Inhalt

- **Definitionen und Begriffe**
- **Strategien zur Parallelisierung**
- **Kommunikation & Synchronisation**
- **Hardware-Voraussetzungen**
- **Probleme**

- (nicht: Theorie: Petri-Netze, ... )

# Was ist Nebenläufigkeit?

**Nebenläufigkeit**
(engl. „**Concurrency**“)

= *Mehrere Dinge passieren*
*(zumindest potentiell / gedanklich)*
*gleichzeitig bzw. parallel zueinander!*

Verwandte Begriffe:

*Multithreading, Multitasking,*
*Parallelprogrammierung, …*

Realer Vergleich:

Film mit *mehreren Handlungs-Strängen*

# Was ist Asynchronizität?

**Asynchron**
= Gegenteil von synchron
= *nicht starr im Gleichschritt*

- Die *zeitliche Relation* zueinander
  von gleichzeitig ablaufenden Vorgängen
  ist *nicht exakt fixiert / vorhersehbar!*
- Die Vorgänge laufen
  *zumindest zeitweise unabhängig voneinander*.

Realer Vergleich:
Nicht Fließbandfertigung
sondern *Polizeikommissariat*

# In eingebetteten Systemen? (1)

- Auch eingebettete Systeme können heute **Multicore- / Multiprozessor-Systeme** sein!

  Dual-Core MIPS / ARM / PowerPC CPU's:
  *‹ 20 Euro, ‹ 2 W*

  „Große" CPU's für Netzwerk / Telekom:
  *32 MIPS-Cores (schon im Jahr 2000: 4 Cores)*

- **Verteilte Systeme**
  (= mehrere *unabhängige, kommunizierende Rechner*)
  erfordern *ähnliche Konzepte* der Programmierung.

  „Fairlight CMI" (1. digitaler Synthesizer, ~1980):
  *Bis zu 10 Prozessoren MC6809 (8 Bit)*

# In eingebetteten Systemen? (2)

Für jedes **Echtzeit-Betriebssystem**
(= **preemptives Multitaskting-System**)
werden Anwendungen mit denselben Konzepten
*wie für ein Multicore-System programmiert*
*(auch bei nur einem Core!)*.

Das Betriebssystem kann *jederzeit* von sich aus
*zwischen Aufgaben umschalten*

==› Ausführ-Reihenfolge *nicht fix vorhersehbar*
==› Quasi-parallele („*verzahnte*“) Ausführung
==› Logisch äquivalent zu echter Parallelität
==› Gleiche Programmier-Mechanismen nötig!

# Verschiedene Ebenen

- **Multitasking** =

  Gleichzeitiger Ablauf
  **mehrerer verschiedener** *Aufgaben / Programme*

  Ergibt sich meist automatisch
  aus der *Gesamtarchitektur des Systems!*

- **Multithreading** =

  Parallelität **innerhalb** *einer einzelnen Aufgabe*
  (*eines* Programms)

  Zweck meist: *Erhöhung des Durchsatzes*

# Prozesse, Tasks & Threads (1)

- „**Task**“: Uneinheitlich, oft Synonym für „Prozess“
- „**Prozess**“ = „*Gestartetes Programm*“

*Aus Betriebssystem-Sicht:*
**Einheit der Speicher- und Rechte-Verwaltung**
(belegt Speicher, führt ein Programm aus, hat offene Files, hat Rechte, ...)

==> Prozess-Wechsel sind eher *aufwändig!*

Jeder Prozess enthält *mindestens einen Thread!*
*(bei einem parallelen Programm: Mehrere!)*

# Prozesse, Tasks & Threads (2)

- „**Thread**“ = „*1 Ablauf innerhalb eines Prozesses*“

  *Aus Betriebssystem-Sicht:*
  **Einheit der CPU-Zuteilung**
  (hat aktuellen Code- und Stackpointer,
  Zustand (wartet, rechnet, ...), Priorität, ...)

  „Lebt“ *innerhalb* eines Prozesses:
  Nutzt dessen Speicher, Files, Rechte usw.
  ==> Alle Threads eines Prozesses haben
  *dieselben globalen & dynamischen Variablen!*

  ==> Thread-Wechsel sind relativ *effizient!*

# Prozesse, Tasks & Threads (3)

*Läuft jeder Code als Thread in einem Prozess?*

**Nein:** Das *Betriebssystem* enthält

- *Betriebssystem-interne* Threads und
- **„Interrupt Handler“**
  = kurze Codestücke (Bestandteil von „Treibern“), die der Prozessor automatisch als Reaktion auf *Interrupts* (Hardware-I/O, Timer, ...) ausführt.

  Sie können *jeden anderen Code zu jedem beliebigen Zeitpunkt* unterbrechen!

  ==> Extremfall *asynchroner* Ausführung!

# Multitasking (1)

... ergibt sich bei eingeb. Systemen meist implizit:

- **Low-Level I/O**
- **Hochpriore** Berechnungen
  (Berechnen der nächsten Ausgangs-Werte,
  Behandeln von Not-Situationen & Störungen)
- **Langfristige** Berechnungen
  (z.B. Wegberechnung, math. Modelle, ...)
- **Benutzerführung**, GUI
- **Nicht zeitkritisches I/O** (Netz, Platte),
  **Protokollierung**, Statistiken, ...

# Multitasking (2)

==> Die einzelnen Aufgaben müssen
*intern nicht parallel* programmiert werden!

==> Das Betriebssystem (genauer: Der „**Scheduler**“)
**verteilt die Ausführung der Threads auf ein oder mehrere CPU-Cores**, je nach

- *Externen Ereignissen*
  = **Interrupts** von I/O-Geräten, Timern, ...
- **Priorität** (Wichtigkeit) der Threads
- **Kommunikation** der Threads untereinander
- Ev. **Zeitscheiben** (bei eingeb. Systemen selten!)

# Varianten des Multithreadings (1)

**Datenparallelität:**

- *Unterteilung der Daten*
- Jeder Thread macht *dieselbe Operation* auf einem *anderen Teil der Daten*

==> Die Threads sind *ident* und arbeiten „*nebeneinander*“

Anwendung:

Bei *großen Datenmengen* und auf allen Teildaten *ident & unabhängig auszuführenden Operationen*

# Beispiele Datenparallelität

- **Bild- oder Signalverarbeitung**
  (Filterung, Codierung, Mustererkennung, …):
  - Entweder (geometrische) Unterteilung jedes einzelnen Frames / Samples
  - Oder parallele Verarbeitung („reihum“) aufeinanderfolgender Frames / Samples
- **Netzwerkkomponenten** (Firewall, Content Filter, NAS-Server, VPN-Konzentrator, …)

  Pool von *x unabhängigen, identen Threads*:
  *x* Pakete / Verbindungen *zugleich* verarbeitbar

# Varianten des Multithreadings (2)

**Pipelining:**

- *Unterteilung des Lösungsverfahrens*
- Jeder Thread rechnet
  *einen Teilschritt* für *alle Daten*

==> Die Threads sind *verschieden*
und arbeiten „*nacheinander*“

==> Die Daten „fließen“
*sequentiell* (Schritt für Schritt)
durch *alle* Threads

# Asynchrones I/O

= (einfachster) *Sonderfall des Pipelinings:*

- Ein Thread *rechnet*
- Ein Thread *liest* die *nächsten* Eingabe-Daten
- Ein Thread *schreibt* die *vorigen* Ausgabe-Daten

Wichtig bei:

- *Langsamem* / aufwändigem I/O
  (Disk, Netz, ...)
- *Interrupt-getriebenem* / asynchronem I/O
  (Parallelität Anwendung / Interrupt Handler)

# Datenaustausch Möglichkeit 1

**Queues, Pipes, Message Passing, FIFO, ...:**

Thread A *schickt* Thread B Daten, meist „*paketweise*“

2 Operationen: **send** & **receive**

Wie ein *Brief* oder *Förderband*

==> Daten werden *kopiert* (ev. zwischengespeichert)
==> *Implizite Synchonisation:*
**B wartet automatisch, bis Daten von A da sind**

==> Meist via System-Aufruf & mit Prozesswechsel
==> Langsam & aufwändig!

Aber: *Einfach vorstellbar, weniger Fehler-anfällig!*

# Datenaustausch Möglichkeit 2

**Shared Memory** (gemeinsamer Speicher):

A und B greifen *„ganz normal“* auf *dieselben Variablen* (denselben RAM-Bereich) zu.

Wie *„schwarzes Brett“* oder *Zentrallager*

==> *Keine speziellen Operationen*
==> *Kein zusätzlicher Aufwand* oder Platzbedarf
==> Betriebssystem nicht involviert
==> Einfach & schnell!

Aber:

In *verteilten* Systemen *nicht* (oder nur simuliert)!

# Probleme von Shared Memory (1)

- *Zeitlich unkoordinierte Zugriffe* verursachen

  **inkonsistente Daten**

  Beispiele: Zähler erhöhen, an Liste anhängen, ...

  ==> *Wer darf wann worauf zugreifen???*

- *Wie erkennt B, dass A fertig ist*
  oder Daten geliefert hat?

  1. Idee: „**Polling**“ bzw. „**Busy Waiting**“
     (= in einer Schleife *laufend prüfen*)

  ==> *Meist ganz schlecht, braucht sinnlos viel CPU!*

# Probleme von Shared Memory (2)

Daher:

Datenaustausch via Shared Memory braucht

**zusätzliche Mechanismen**

- zur *Synchronisation* und *Signalisierung*
- zum *gegenseitigen Ausschluss*
  („**Mutual Exclusion**“)

  d.h. zum *exklusiven Zugriff*
  auf gemeinsame Daten

# Locking (1)

Ein „**Lock**“ (~ „**Semaphore**“, „**Mutex**“, „**Monitor**“)
ist wie ein „*Staffel-Stab*“:

*Zu jedem Zeitpunkt darf es*
*maximal einer „haben“!*

2 Operationen:

**Nehmen & Freigeben**

*(„lock / unlock“, „acquire / release“, „wait / signal“)*

Wer „**lock**“ macht, wird *automatisch angehalten*,
bis der *aktuelle Lock-Besitzer* „**unlock**“ aufruft!

# Locking (2)

Grundsätzliche **Anwendung** von Locks:

- *Ein Lock pro Datenbereich*,
  auf den der Zugriff koordiniert werden soll.
- „**Lock**“ steht *unmittelbar vor* jedem Code-Stück,
  das auf die *gemeinsamen Daten* zugreift,
  und „**unlock**“ *gleich danach*.

==> *Es kann immer nur einer den Code ausführen,*
*der auf die gemeinsamen Daten zugreift.*

Ein solches Codestück heißt „**Critical Region**“
(einige Sprachen machen das *Lock* dafür *intern*).

# Hardware-Voraussetzungen

Synchronisation und Kommunikation
können auf unterster Ebene
*nur mit Hardware-Hilfe implementiert* werden!

- **Single**-Core-Systeme: Befehl „**Interrupt-Sperre**“

  Threadwechsel verhindert ==› es reichen „normale“ Befehle

- **Multi**-Core-Systeme: „**Atomic Instructions**“
  *(Atomic Increment, atomic exchange, ...)*

  Ein Core kann eine Speicherstelle
  **exklusiv** *lesen und gleich wieder zurückschreiben!*

  Technisch *sehr* aufwändig ==› relativ langsam!

# Probleme (1)

Die *„gedankliche Simulation“ paralleler Abläufe* überfordert das menschliche Gehirn

==>

*Parallele / nebenläufige Programmierung ist schwierig und sehr fehleranfällig!*

# Probleme (2)

**„Race Conditions“:**

*Korrektheit / Ergebnis / Verhalten*
des Programms
hängen von **zeitlichen Zufällen** ab.

Ursache: *Fehlende explizite Synchronisation!*

==> Zugriffe auf *gemeinsamen Speicher*
je nach zufälliger zeitlicher Verzahnung
in *verschiedener Reihenfolge*.

==> *Unterschiedliche Werte* im Speicher,
unterschiedliche Lese-Ergebnisse!

# Probleme (3)

**„Deadlock“** (**„Verklemmung“**):

**Zyklisches Warten** mehrerer Threads aufeinander:

*A wartet auf B, B wartet auf A*
*(„Dining Philosophers“)*

==> Keiner kann mehr weiterrechnen, *System steht!*
==> *Gewaltsamer Eingriff* von außen nötig!

Ursache: *„Zu viel“ Synchronisation:*
Schlecht entworfene Kommunikations- und Synchronisations-*Abhängigkeiten.*
Aber: Theoretisch gut untersucht & *gelöst!*

# Probleme (4)

**„Livelock“:**

Zwei oder mehr Prozesse kommen *nicht voran*, weil sie sich *nur mehr mit gegenseitiger Synchronisation / Kommunikation beschäftigen*.

Anschauliches Beispiel:

*Zwei Leute begegnen einander auf derselben Seite des Gehsteigs und weichen synchron hin und her aus.*

Kann z.B. bei *optimistischen Strategien* auftreten:

==> Nur mehr *Retries* (z.B. altes Ethernet)!

# Probleme (5)

**„Priority Inversion“**:

- **Niedrig-priorer Thread C**
  hält gemeinsame Daten *gesperrt*.
- **Mittel-priorer Thread B** nimmt C die *CPU weg*
  ==> C wird nicht fertig, sperrt das Lock „ewig“!
- **Hoch-priorer Thread A**
  *wartet* auf die von C gesperrten Daten.

==> Mittel-priorer B kann hoch-prioren A
*beliebig lange blockieren!*

In der Praxis: *Häufiges Problem in Steuerungen!*

# Probleme (6)

**Interrupt Handler**
(= jene Teile des Codes, die unmittelbar durch Interrupts aktiviert werden)
**dürfen nie blockiert (wartend gestellt) werden!**

Daher kann man in Interrupt Handlern
*weder normales Send / Receive*
*noch Lock / Unlock* verwenden.

==> Sie erfordern *spezielle Techniken*
zum Datenaustausch mit dem Rest des Systems!

# Probleme (7)

**„Amdahl's Law“:**

*Der nicht parallelisierbare Code-Anteil*
*beschränkt die erreichbare Beschleunigung*
*des Gesamtsystems.*

Anschaulich: **„50 % Regel“**

„Wenn (zeitlich) die *Hälfte* des Problems
*rein sequentiell* berechnet werden muss,
wird es auch mit unendlich vielen Prozessoren
*höchstens doppelt so schnell*“

==> *Ist das Problem überhaupt gut parallelisierbar?*