Übersicht & Wiederholung

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 10. Februar 2021

1 Einleitung

2 Synchronisation

3 Programmierprimitiven

4 Zugriff auf mehrere Ressourcen

4.1 Deadlocks bei mehreren Ressourcen

4 notwendige Bedingungen

4.2 Deadlock-Verhinderung

2-Phasen Sperr-Protokoll (mit Timestamping), Total-Order Theorem

4,3 Deadlock-Vermeidung

Bankiers-Algorithmus

4.4 Transactional Memory

Basisprimitive, Atomare Blöcke, abort, retry, orElse, Eigenschaften von TM Systemen, Korrektheitskriterien (z.B. Sequentialisierbarkeit), TL2-Algorithmus

5 Nebenläufigkeit in der Programmiersprache Haskell

5.1 I/O in Haskell

5.2 Concurrent Haskell

MVars mit Operationen, forkIO, ...

5.3 Software Transactional Memory in Haskell

`>>=`, `return`, `retry`, `orElse`

6 Semantische Modelle nebenläufiger Programmiersprachen

6.1 Der Lambda-Kalkül

6.2 Ein Message-Passing-Modell: Der π-Kalkül

synchron / asynchron, Turing-mächtig, monadisch /polyadisch, Summen, Bisimulation

6.3 CHF-Kalkül

Im Folgenden: Auswahl wichtiger Themen

Modellannahmen

Interleaving-Annahme

Ausführung eines nebenläufigen Programms:

> *Sequenz der atomaren Berechnungsschritte der Prozesse, die beliebig durchmischt sein können.*

Fairness-Annahme

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

Das Mutual-Exclusion-Problem, formales Modell

Code-Struktur jedes Prozesses

```
loop forever
    restlicher Code
    Initialisierungscode
    Kritischer Abschnitt
    Abschlusscode
end loop,
```

Korrektheitskriterien

Lösung des Mutual-Exclusion-Problems

Fülle Initialisierungs- und Abschlusscode, so dass die folgenden Anforderungen erfüllt sind:

- Wechselseitiger Ausschluss: Es sind niemals zwei oder mehr Prozesse zugleich in ihrem kritischen Abschnitt.
- Deadlock-Freiheit: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann betritt irgendein Prozess schließlich den kritischen Abschnitt.

Starvation-Freiheit

Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann muss er ihn nach endlich vielen Berechnungsschritten betreten.

Algorithmus von Peterson

Initial: wantp = False, wantq = False, turn egal

Prozess P:

```
 loop forever
(P1)   restlicher Code
(P2)   wantp := True;
(P3)   turn := 1;
(P4)   await wantq = False or turn = 2
(P5)   Kritischer Abschnitt
(P6)   wantp := False;
 end loop
```

Prozess Q:

```
 loop forever
(Q1)   restlicher Code
(Q2)   wantq := True;
(Q3)   turn := 2;
(Q4)   await wantp = False or turn = 1
(Q5)   Kritischer Abschnitt
(Q6)   wantq := False;
 end loop
```

Komplexitätsresultate bei atomarem Lesen & Schreiben

Untere Schranke für den Platz:

Theorem

Jeder Deadlock-freie Mutual-Exclusion Algorithmus für n Prozesse benötigt mindestens n gemeinsam genutzte Speicherplätze.

Obere Schranke für den Platz:

Theorem

Es gibt einen Deadlock-freien Mutual-Exclusion Algorithmus für n Prozesse der n gemeinsame Bits verwendet.

Laufzeit lässt sich nicht beschränken:

Theorem

Es gibt keinen (Deadlock-freien) Mutual-Exclusion Algorithmus für 2 (oder auch n) Prozesse, der eine obere Schranke hat für die Anzahl an Speicherzugriffen (des gemeinsamen Speichers), die ein Prozess ausführen muss, bevor er den kritischen Abschnitt betreten darf.

Stärkere Speicheroperationen

,

test-and-set(r, v)

```
function test-and-set(r : Register, v : Wert) returns : Wert
    temp := r;
    r := v;
    return(temp);
end function
```

swap

```
function swap(r : Register, l : Lokales Register)
    temp := r;
    r := l;
    l := temp;
end function
```

Stärkere Speicheroperationen (2)

fetch-and-add

```
function fetch-and-add(r : Register, v : Wert) returns : Wert
    temp := r;
    r := temp + v;
    return(temp);
end function
```

read-modify-write

```
function read-modify-write(r : Register, f : Funktion)
  returns : Wert
    temp := r;
    r := f(temp);
    return(temp);
end function
```

Stärkere Speicheroperationen (3)

compare-and-swap

```
function compare-and-swap(r : Register, old : Wert, new : Wert)
 returns : Wert
  if r = old then
      r := new;
      return(True);
  else
      return(False);
end function
```

move

```
function move(r_1 : Register, r_2 : Register)
   temp := r_2;
   r_1 := temp;
end function
```

Stärkere Speicheroperationen (4)

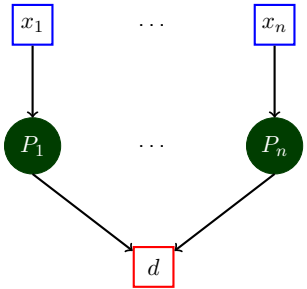
LL/SC-Objekt (war in der Übung)

`function` *LoadLink*(r : **Register**) `returns` : **Wert**
liest den Wert des Registers
`end-function`

`function` *StoreConditional*(r : **Register**, v : **Wert**) `returns` : **Bool**
Wenn Prozess P die Operation ausführt und r nicht beschrieben hat, seitdem P das Register r zuletzt (mit LoadLink) gelesen hat, dann wird v in das Register geschrieben und True geliefert. Anderenfalls findet keine Speicheränderung statt und False wird geliefert.
`end-function`

Das Konsensus Problem

- n Prozesse, die auch abstürzen können
- Prozess i erhält einen **Eingabewert** $x_i \in \{0, 1\}$
- Programmiere die Prozesse, so dass alle (nicht-abstürzenden) Prozesse sich für einen gemeinsamen **Entscheidungswert** $d \in \{0, 1\}$ entscheiden



- **Übereinstimmung**: Alle nicht-abgestürzten Prozesse entscheiden sich für den gleichen Wert d.
- **Gültigkeit**: $d \in \{x_1, \ldots, x_n\}$, d.h. d ist einer der Eingabewerte.

Lösung mit dreiwertigem RMW-Objekt

Objekte und Initialisierung:
x: RMW-Objekt mit den möglichen Werten $\perp, 0, 1$, initial $\perp$
x_i: Eingabewert von Prozess i
d_i: Entscheidungswert, den Prozess i trifft.

Programm des i. Prozesses

(1) $d_i :=$ *read-modify-write*(x,f_i);
(2) `if` $d_i = \perp$ `then`
$d_i := x_i$;

Funktion f_i des i. Prozesses

`function` f_i(v)
`if` v=$\perp$ `then return` x_i
`else return` v
`end function`

} erster Prozess setzt sein x_i als neuen Wert, alle anderen nicht-abstürzenden Prozesse lesen diesen Wert
$\Rightarrow$ alle d_i-Werte identisch

Die Konsensus-Zahl

Definition

Für ein nebenläufiges Objekt vom Typ o ist die Konsensus-Zahl $\mathbb{CN}(o)$ die größte Zahl an Prozessen n für die man das Konsensus-Problem für n Prozesse lösen kann, indem man beliebig viele Objekte vom Typ o und beliebig viele atomare Register (mit *read* und *write*) verwendet. Ist die Anzahl unbeschränkt, so sei $\mathbb{CN}(o) = \infty$.

$\mathbb{CN}(o)$	Objekt o
1	atomares Register mit *read* und *write*
2	test-and-set Objekt, fetch-and-increment Objekt, fetch-and-add Objekt, swap-Objekt, read-modify-write Bit
$\Theta(\sqrt{m})$	swap^m-Objekt
$2m-2$	m-Register mit m-facher Zuweisung ($m > 1$)
∞	(drei-wertiges) RMW-Objekt, Compare-and-swap-Objekt, Sticky-Bit

Semaphor S

Attribute (i.a.):

- $V =$ Nicht-negative Ganzzahl
- $M =$ Menge von Prozessen

Schreibweise für Semaphor S: $S.V$ und $S.M$

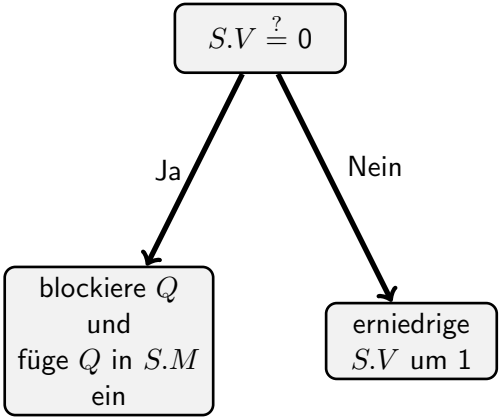
Operationen:

- `newSem`(k): Erzeugt neuen Semaphor mit $S.V = k$ und $S.M = \emptyset$
- `wait`(S)
- `signal`(S)

wait(S)

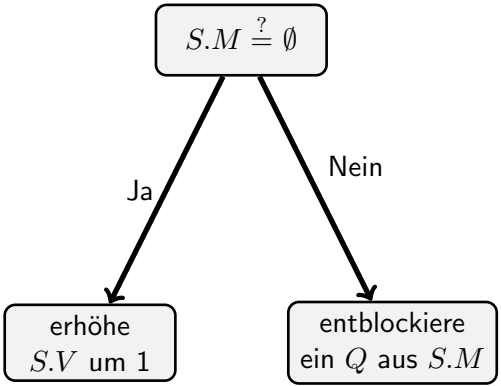
Sei Q der aufrufende Prozess:

```
procedure wait(S)
  if S.V > 0 then
     S.V := S.V − 1;
  else
     S.M := S.M ∪ {Q};
     Q.state := blocked;
```



signal(S)

```
procedure signal(S)
  if S.M = ∅ then
     S.V := S.V + 1;
  else
     wähle ein Element Q aus S.M;
     S.M := S.M \ {Q};
     Q.state := ready;
```



Monitore

```
monitor Konto {
  int Saldo;
  int Kontonummer;
  int KundenId

  abheben(int x) {
    Saldo := Saldo - x;
  }

  zubuchen(int x) {
    Saldo := Saldo + x;
  }
}
```

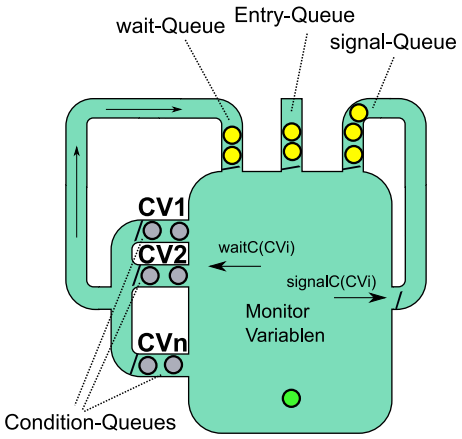
- Kapselung von Daten und Methoden
- Kein direkter Zugriff auf Attribute
- Zugriff nur über die Methoden
- Nur ein Prozess kann zu einer Zeit im Monitor sein
- D.h. nur eine Methode von einem Prozess zu einer Zeit am Ausführen
- Andere Prozesse werden blockiert

Monitore mit Condition Variables

- FIFO-Queue (meistens) mit Operationen
- Name der Condition Variables wird meistens so gewählt, dass er die wahr werdene Bedingung erläutert, aber
- Operationen für Condition Variable cond:
  `waitC`(cond) und `signalC`(cond)

Semaphore Sem	Monitore (Condition Variable cond)
`wait`(Sem) kann zum Blockieren führen, muss aber nicht	`waitC`(cond) blockiert den Prozess stets
`signal`(Sem) hat stets einen Effekt: Entblockieren eines Prozesses oder Erhöhen von Sem.V	`signalC`(cond) kann effektlos sein: Entweder Prozess in cond wird entblockiert, oder effektlos, wenn cond leer ist

Monitor-Modellierung mit drei Queues für die Condition Variable



- **Wartende Prozesse: Verwaltung durch Monitor.lock**
- **Wartende Prozesse: Verwaltung durch Condition**
- **Prozess im Monitor**

Prioritäten

1	$E = W = S$	
2	$E = W < S$	Wait and Notify
3	$E = S < W$	Signal and Wait
4	$E < W = S$	
5	$E < W < S$	Signal and Continue
6	$E < S < W$	Klassische Definition
7	$E > W = S$	nicht sinnvoll
8	$E = S > W$	nicht sinnvoll
9	$S > E > W$	nicht sinnvoll
10	$E = W > S$	nicht sinnvoll
11	$W > E > S$	nicht sinnvoll
12	$E > S > W$	nicht sinnvoll
13	$E > W > S$	nicht sinnvoll

Monitore in Java

```
class MonitoredClass {
  ... private Attribute ...
  synchronized method1 {...}
  synchronized method2 {...}
```

Statt Condition Variables

- Operationen `wait`, `notify`, `notifyAll`
- Nur eine Queue pro Objekt
- `wait`(): Thread wartet an der Queue des Objekts
- `notify`(): Ein wartender Thread wird entblockiert, aber:
  Aufrufender Prozess behält Lock!
- `notifyAll`(): Alle wartende Threads werden entblockiert, aber:
  Aufrufender Prozess behält Lock!
- Wartende Threads haben gleiche Priorität wie neue!
- Entspricht $W = E < S$

Kanäle: Operationen

- ch $\Leftarrow$ w
  - entspricht: "sende w über den Kanal ch"
  - dabei ist w ein Wert vom passenden Typ oder Programmvariable
  - in Go: `ch <- w`
- ch $\Rightarrow$ x
  - entspricht "empfange über den Kanal ch und setze Variable x auf den empfangenen Wert"
  - In Go: `x := <- ch`

Tuple Spaces: Wesentliche Operationen

- `out`$(N, v_1, \ldots, v_n)$:

  Einfügen eines Tupels in den Tuple Space, v_i können Werte oder Programmvariablen sein
- `in`$(N, x_1, \ldots, x_n)$:

  Entfernen eines Matching Tupels, binden der Werte an Variablen x_i.
  Erweiterung $x_i =$ statt x_i bedeutet:
  Wert muss gleich zum Wert der Programmvariablen x_i sein
- `read`$(N, x_1, \ldots, x_n)$:

  Wie in aber ohne Entfernen des Tupels

Erzeuger / Verbraucher

Problemstellung

- Erzeuger: Produziert Daten
- Verbraucher: Konsumiert Daten

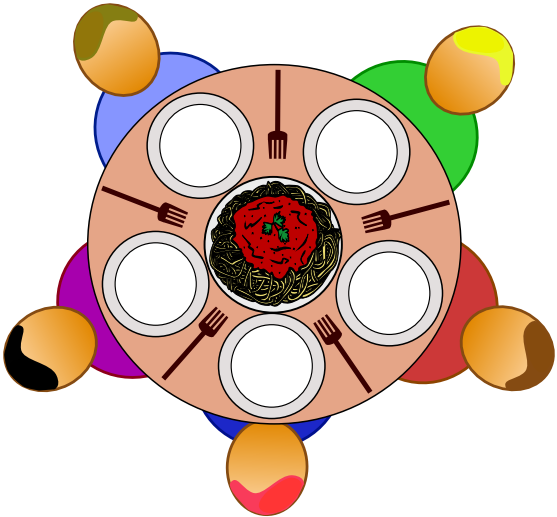
Erzeuger / Verbraucher mit infinite Buffer:

- Lesen / Schreiben auf den Puffer sicher (atomar)
- Verbraucher braucht Schutz für den Fall, dass der Puffer leer ist

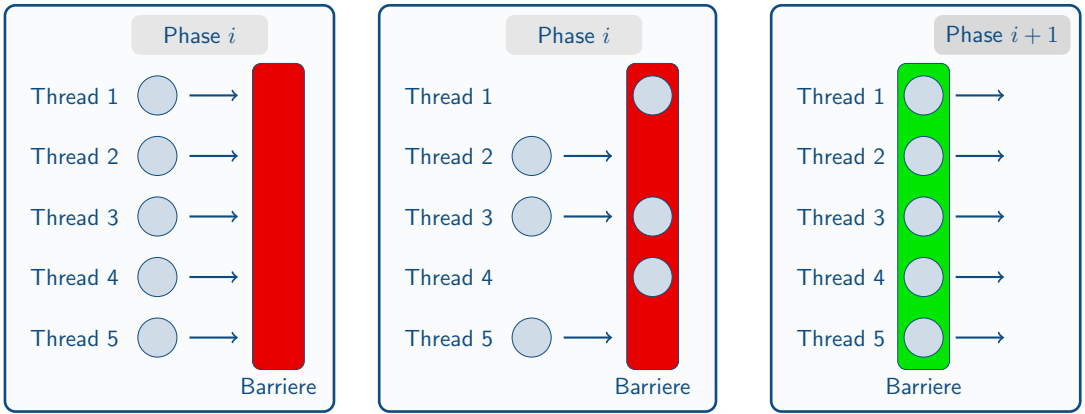
Erzeuger / Verbraucher mit bounded Buffer:

- Lesen / Schreiben auf den Puffer sicher (atomar)
- Verbraucher braucht Schutz für den Fall, dass der Puffer leer ist
- Erzeuger braucht Schutz für den Fall, dass der Puffer voll ist

Speisende Philosophen



Barrieren


Phase i
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Barriere
Phase i
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Barriere
Phase $i+1$
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Barriere

Readers & Writers

Gruppierung der Prozesse in

- Readers: Prozesse, die auf eine gemeinsame Ressource lesend zugreifen
- Writers: Prozesse, die auf die gemeinsame Ressource schreibend zugreifen

Erlaubt / Nicht erlaubt

- Mehrere lesende Prozesse gleichzeitig, aber
- Nur ein Prozess schreibt gleichzeitig

Problem:

- Löse den Zugriff so, dass viele gleichzeitig lesen, aber nie mehrere gleichzeitig schreiben.

Verschiedene Lösungen:

- Priorität für Readers
- Priotität für Writers

Wann tritt globaler Deadlock auf?

Vier notwendige Bedingungen (alle gleichzeitig erfüllt):

1. Wechselseitiger Ausschluss (Mutual-Exclusion): Nur ein Prozess kann gleichzeitig auf eine Ressource zugreifen.
2. Halten und Warten (Hold and Wait): Ein Prozess kann eine Ressource anfordern (auf eine Ressource warten), während er eine andere Ressource bereits belegt hat.
3. Keine Bevorzugung/Unterbrechung (No Preemption): Jede Ressource kann nur durch den Prozess freigegeben (entsperrt) werden, der sie belegt hat.
4. Zirkuläres Warten: Es gibt zyklische Abhängigkeit zwischen wartenden Prozessen: Jeder wartende Prozess möchte Zugriff auf die Ressource, die der nächste Prozesse im Zyklus belegt hat.

Deadlock-Verhinderung

Verhindern von Hold and Wait

- 2-Phasen Sperrprotokoll, Nachteil: Live-lock möglich
- 2-Phasen Sperrprotokoll mit Timestamping: Prozesse in Phase 1 erhalten Ressourcen, wenn sie den kleinsten Zeitstempel haben

Verhindern der Zyklischen Abhängigkeit:

- Total-Order Theorem: Sind alle gemeinsamen Ressourcen durch eine totale Ordnung geordnet und jeder Prozess belegt seine benötigten Ressourcen in aufsteigender Reihenfolge bezüglich der totalen Ordnung, dann ist ein Deadlock unmöglich.

Deadlock-Vermeidung: Bankier-Algorithmus

```
function testeZustand(P, A⃗):
  if P = ∅ then
      return "sicher"
  else
      if ∃P ∈ P mit M⃗_P − C⃗_P ≤ A⃗ then
          A⃗ := A⃗ + C⃗_P;
          P := P \ {P};
          testeZustand(P, A⃗)
      else
        return "unsicher"
```

Transactional Memory

`atomic`-Blöcke:

```
atomic {
    Code der Transaktion
}
```

Der `retry`-Befehl

- Ermöglicht es Transaktionen zu koordinieren
- `retry`: Transaktion wird abgebrochen (Roll-back) und erneut gestartet

Der `orElse`-Befehl

- Gibt Alternativen vor, wenn Transaktionen in retry laufen
- T_1 `orElse` T_2.

Korrektheitskriterien für STM-Systeme

Historie= Folge von Ereignissen, wobei Ereignis:

- Aufrufe & Rückgaben v. READ(x), WRITE(x,v), COMMIT, ABORT
- Spezialwert A_T = Transaktion T ist abgebrochen.

(Strikte) Sequentialisierbarkeit

- Für jede Historie H des Systems ist comm(H) (=committed Transaktionen in H) **äquivalent** zu einer sequentiellen, legalen Historie
- Strikte Sequentialisierbarkeit: die sequentielle, legale Historie erhält die Realzeitordnung

Äquivalenz:

Zwei Historien sind äquivalent, wenn die Ereignissefolge pro Transaktion dieselbe ist (d.h. gleiche Reihenfolge innerhalb einer Transaktion und gleiche Rückgaben).

Concurrent Haskell

- `forkIO :: IO () -> IO ThreadId`
- Terminierung des main-Threads, beendet alle Threads
- `newEmptyMVar :: IO (MVar a)`
  erzeugt leere `MVar`
- `takeMVar :: MVar a -> IO a`
  - liest Wert aus `MVar`, danach ist die `MVar` leer
  - falls `MVar` vorher leer: Thread wartet
  - Bei mehreren Threads: FIFO-Warteschlange
- `putMVar :: MVar a -> a -> IO ()`
  - speichert Wert in der `MVar`, wenn diese leer ist
  - Falls belegt: Thread wartet
  - Bei mehreren Threads: FIFO-Warteschlange

STM-Haskell

- `atomically :: STM a -> IO a` überführt eine STM-Aktion in eine IO-Operation
- `data TVar a = ...`
- `newTVar :: a -> STM (TVar a)` erzeugt eine neue `TVar` mit Inhalt
- `readTVar :: TVar a -> STM a` liest den den momentanen Wert einer `TVar`
- `writeTVar :: TVar a -> a -> STM ()` schreibt einen neuen Wert in die `TVar`
- `retry :: STM a`
- `>>=`, `>>`, do erlaubt sequentielle Komposition
- `orElse :: STM a -> STM a -> STM a`

Synchroner π-Kalkül ohne Summe mit Replikation

Syntax

- $\mathcal{N}$ abzählbar unendliche Menge von Namen
- Syntax für π-Kalkül-Prozesse ($x \in \mathcal{N}$)

$$
\begin{array}{rcll}
P & ::= & \pi.P & \text{(Aktion)} \\
 & | & P_1 \mid P_2 & \text{(Parallele Komposition)} \\
 & | & !\,P & \text{(Replikation)} \\
 & | & \mathbf{0} & \text{(Inaktiver Prozess)} \\
 & | & \nu x.P & \text{(Restriktion)}
\end{array}
$$

- Syntax für Aktionspräfixe wobei $x, y \in \mathcal{N}$

$$
\begin{array}{rcll}
\pi & ::= & x(y) & \text{Input} \\
 & | & \overline{x}y & \text{Output}
\end{array}
$$

Operationale Semantik

Reduktionsregeln

(INTERACT)	$x(y).P \mid \overline{x}v.Q \to P[v/y] \mid Q$
(PAR)	$P \mid Q \to P' \mid Q$, falls $P \to P'$
(NEW)	$\nu x.P \to \nu x.P'$, falls $P \to P'$
(STRUCTCONGR)	$P \to P'$, falls $Q \to Q'$, $P \equiv Q$ und $P' \equiv Q'$

Strukturelle Kongruenz $\equiv$ erlaubt α-Umbenennung und Umordnen bez. $\mid$ und ν, Entfalten von !