Verteilter gemeinsamer Speicher

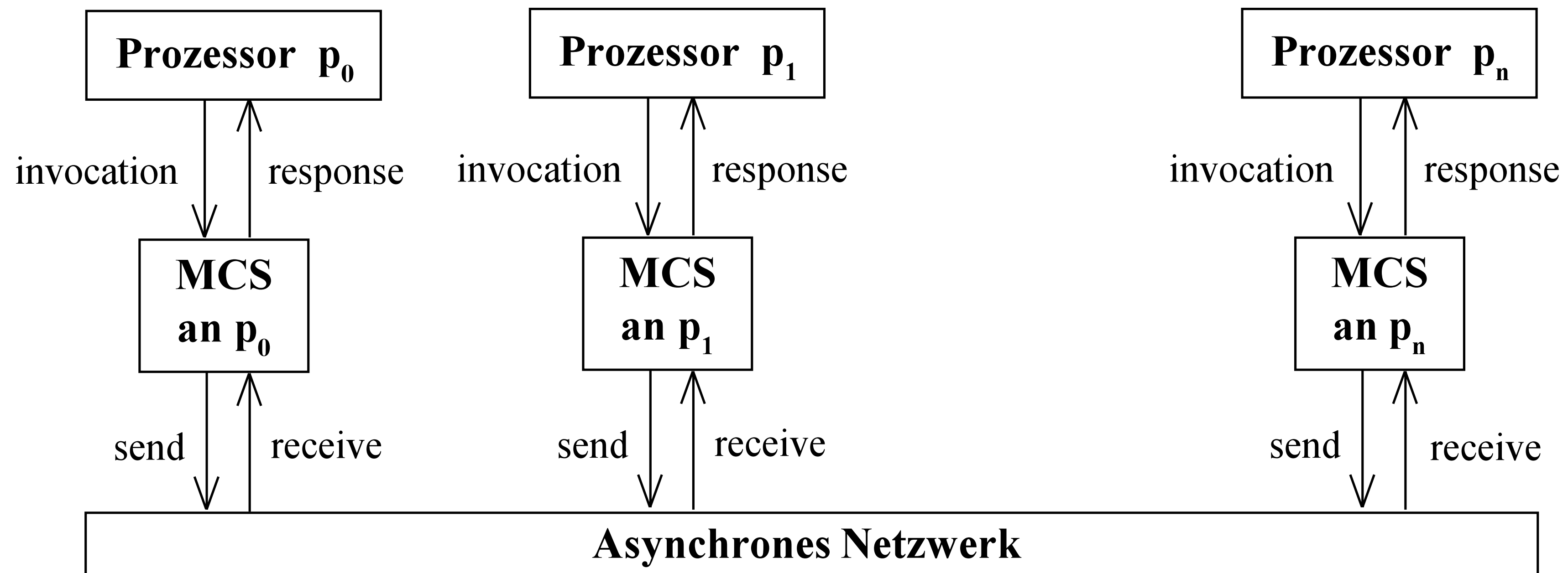
(Distributed Shared Memory)

Seminarvortag zu „Verteilte Algorithmen“
von Benjamin Bahnsen

Inhalt:

1.) Aufbau des System - was ist verteilter gemeinsamer Speicher?
2.) Linearisierbarkeit
3.) Sequentielle Konsistenz
4.) Vorhandensein einer globalen Uhr

1.Aufbau des Systems

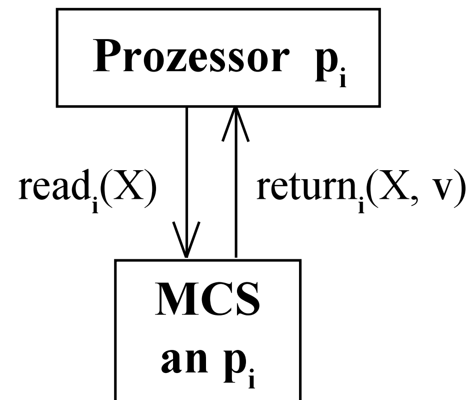


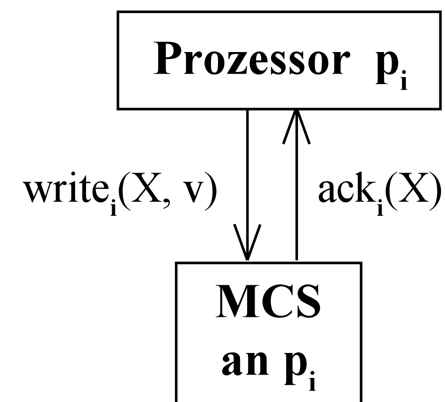
- Jeder Prozessor verfügt über ein MCS (memory consistency system), über das er auf den verteilten gemeinsamen Speicher zugreift
- Jedes MCS verfügt über die gleichen lokalen read/write-Objekte (im folgenden x, y und z); es existiert kein globaler Speicher
- MCS' können durch Nachrichten über ein Netzwerk kommunizieren, so daß Speicherveränderungen an andere MCS weitergegeben werden können

Für Nachrichten im asynchronen Netzwerk gilt:

- keine Verfälschung von Nachrichten
- kein Verlust/Duplizierung/Umsortierung von Nachrichten
- Nachrichtenverzögerung liegt im Bereich [d - u, d]
- Jedem Prozess ist d bekannt

Lesen von Objekt X	**Schreiben von Objekt X**
Prozessor p_i sendet $read_i(X)$	Prozessor p_i sendet $write_i(X, v)$
MCS_i antwortet mit $return_i(X, v)$	MCS_i antwortet mit $ack_i(X)$





Definition: Eine *Operation* ist eine Kombination von $read_i$ und $return_i$ oder $write_i$ und ack_i. Ersteres ist eine read-Operation, letzteres eine write-Operation.

Definition: Eine Sequenz f aus Operationen ist *legal*, wenn jede read-Operation eines Objekts den Wert der letztes write-Operationen dieses Objekts zurückgibt.

Beispiele:

f = **$write_0$(x, 1)ack_0(x)$read_1$(x)$return_1$(x, 1)** ist legale Sequenz

f = **$write_0$(x, 2)ack_0(x)$read_1$(x)$return_1$(x, 1)** ist keine legale Sequenz

Anforderungen an den gemeinsamen verteilten Speicher:

- ein Wert, den ein Prozesser p_i durch sein MCS in ein Objekt X schreibt, sollte an alle MCS' weitergegeben werden, so daß read-operationen anderer Prozessoren genau diesen Wert zurückliefern.

Wann arbeitet ein MCS korrekt? Zwei Konsistenzmodelle:

- Linearisierbarkeit
- Sequentielle Konsistenz

Eine Sequenz f ist linearisierbar, wenn folgenden Bedingungen erfüllt sind:

- *Correct interaction*
  Für jeden Prozessor p_i besteht f_i (die Ausführungssequenz f eingeschränkt auf p_i) aus abwechselnden *invocations* und *responses*

- *Liveness*
  Auf jedes $read_i$ bzw. $write_i$ folgt ein $return_i$ bzw. ack_i

- *Linearizability*
  Es existiert eine Permutation μ über alle Operationen in f, so daß
  1. für jedes Objekt X, μ_X legal ist
  2. wenn die *response* der Operation o_1 in f vor der *invocation* der Operation o_2 kommt, dann erscheint o_1 in μ vor o_2

Die zweite Permutationsbedingung der Linearisierbarkeit bedeutet anschaulich, dass einer abgeschlossenen Operation keine spätere Operation vorangestellt werden darf.

Beispiele:

f = **$write_0(x, 10)write_1(x, 20)ack_0(x)ack_1(x)read_2(x)return_2(x, 10)$** ist linearisierbar, denn

$write_1(x, 20)ack_1(x)write_0(x, 10)ack_0(x)read_2(x)return_2(x, 10)$ ist die gewünschte Permutation.

f = **$write_0(x, 10)ack_0(x)write_1(x, 20)ack_1(x)read_2(x)return_2(x,10)$** ist nicht linearisierbar.

f = **$write_0(x,10)write_1(x,20)ack_0(x)ack_1(x)read_2(x)read_3(x)return_2(x,10)return_3(x,20)$** ist ebefalls nicht linearisierbar.

2.Linearisierbarer gemeinsamer Speicher - Bedeutung

- Nach einer write-Operation auf ein Objekt X liest jeder Prozess genau diesen Wert
- nach zwei gleichzeitigen write-Operationen auf ein Objekt X, die „acknowledged“ werden, lesen alle Prozessoren nur einen der beiden Werte

Gesucht ist ein Algorithmus für die MCS, der linearisierbaren gemeinsamen Speicher implementiert. Dazu muß jede globale Sequenz von Anweisungen, die durch diesen Algorithmus möglich ist, in ihrer genauen zeitlichen Abfolge linearisierbar sein.

Erste Idee einer Implementierung:

Ein MCS an Prozessor p_i liefert bei einer read_i-*invocation* unmittelbar den Wert seiner Kopie dieses Objektes durch die return_i-*response* zurück.
Bei einer write_i-*invocation* aktualisiert das MCS seine Kopie des Objekts und sendet den neuen Wert an alle anderen MCS und läßt sich den Empfang bestätigen. Erst wenn eine Bestätigung jedes anderen MCS angekommen ist, sendet das MCS ein ack_i-*response* an Prozessor p_i.

Algorithmus für MCS_i, $0 \leq i \leq n-1$
copy[x] enthält den Wert v für jedes gemeinsame Objekt x
acked, a set of MCS', initially Ø

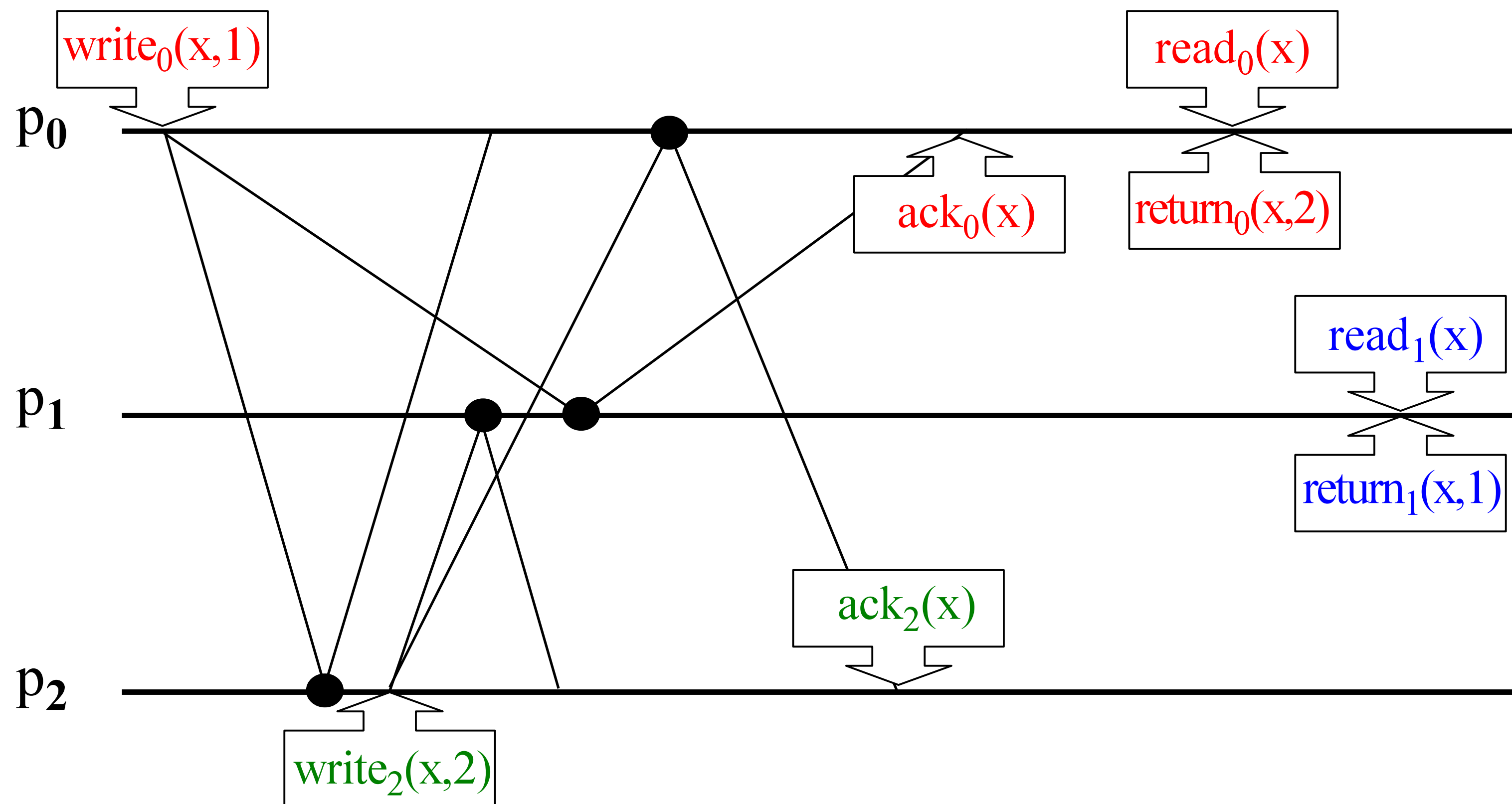
when $read_i$(x) occurs:
    $return_i$(x, *copy*[x])

when $write_i$(x, v) occurs:
    copy[x] = v
    $send_{i,j}$(<x, v>) to all MCS_j, $j \neq i$

when $recv_{i,j}$(x, v) occurs:
    copy[x] = v;
    $send_{i,j}$(„acknowledged“);

when $recv_{i,j}$(„acknowledged“) occurs:
    add j to *acked*;
    if | ack | = n - 1 then ack_i(x);

2.Linearisierbarer gemeinsamer Speicher - Beispielausführung des Implementierungsversuches



f = **write$_0$(x,1)write$_2$(x,2)ack$_2$(x)ack$_0$(x)read$_0$(x)return$_0$(x,2)read$_1$(x)return$_1$(x,1)**

ist <u>nicht</u> linearisierbar.

Der erste Implementierungsversuch garantiert somit keinen linearisierbaren gemeinsamen Speicher.

Problem:

Ein MCS weiß nicht, in welcher Reihenfolge die bei ihm ankommenden Nachrichten verschickt worden sind.

Lösung:

Definition einer Ordnung über alle verschickten Nachrichten, die ohne globale Uhr auskommt.

Totale Ordnung über alle Nachrichten:

- mit jeder Nachricht verschickt ein MCS seinen Timestamp T (anfang 0)
- anschließend wird der Timestamp inkrementiert

Für Nachrichten mit gleichen Timestamp T gilt:

- $T_i < T_j$, wenn $i < j$
- $T_i > T_j$, wenn $i > j$

Weiter:

- ein MCS aktualisiert seinen Timestamp mit jeder Nachricht, die er bekommt
- ändert sich der Timestamp eines MCS, werden alle anderen MCS benachrichtigt

Wann erst darf ein MCS_i eine Nachricht mit Timestamp T verarbeiten?

- wenn gilt $T \geq T_j$, $0 < j \leq n\text{-}1$, $j \neq i$

2.Linearisierbarer gemeinsamer Speicher - Totally ordered broadcast

ts[j], enthält den Timestamp für jeden Prozess $0 \leq j \leq n-1$, bei Start 0 für alle j
pending, eine Liste von Nachrichten, bei Start Ø

when tob-send$_i$(m) occurs:
    ts[i] = *ts*[i] +1;
    add < m, *ts*[i], i > to *pending*
    send$_i$(<m, *ts*[i], i>) to all MCS$_j$, $0 \leq j \leq n-1$

when recv$_i$(<m, T, j>) occurs:
    ts[j] = T;
    add <m, T, j> to *pending*
    if T > *ts*[i] then
        ts[i] = T;
        send$_i$(<timestamp-update, T, i>) to all MCS$_j$, $0 \leq j \leq n-1$

when recv$_i$(<timestamp-update, T, j>), $j \neq i$ occurs:
    ts[j] = T;

tasks:
if <m, T, j> is entry in *pending* with the smallest (T, j) and T ≤ *ts*[k] = k for all $0 \leq k \leq n-1$ then
    enable tob-recv$_i$(m, j);
    remove <m, T, j> from *pending*

Algorithmus für MCS_i, $0 \leq i \leq n-1$

copy[x] enthält den Wert v für jedes gemeinsame Objekt x

when $read_i$(x) occurs:
    enable tob-$send_i$(<„read“>);

when $write_i$(x, v) occurs:
    enable tob-$send_i$(<„write“, x, v>);

when tob-$recv_i$(<„read“>) from MCS_j occurs:
    if j = i then $return_i$(x, copy[x]);

when tob-$recv_i$(<„write“, x, v) from MCS_j occurs:
    copy[x] = v;
    if j = i then ack_i(x);

<u>Behauptung:</u> Der vorgestellte Algorithmus implementiert linearisierbaren gemeinsamen Speicher.

<u>Beweis:</u> z.Z.: jede zulässige Sequenz f ist linearisierbar.

Ordne die Operationen in f entsprechend der totalen Ordnung der Nachrichten zu einer Permutation μ. Zu zeigen:

a) μ_x ist legal

   μ_x, x beliebiges read/write-Objekt ist legal, denn jedes MCS erhält die Operationen auf x in dergleichen Reihenfolge

b) Permutationsbedingung wird nicht verletzt

   Sei Operation o_1 in f beendet, bevor o_2 beginnt

   => das MCS, das o_1 bearbeitet, hat seine eigene Antwort bekommen, bevor das MCS, das o_2 bearbeitet, seine Anfrage gesendet hat

   => o_2 ist in der totalen Ordnung der Nachrichten hinter o_1

   => o_2 ist auch in μ hinter o_1

Effizienz:

- jede read/write-Operation benötigt eine konstante Anzahl von Nachrichten
- für praktische Anwendungen dennoch zu ineffizient, weil read und write globale Operationen sind

In der Praxis daher häufiger verwendet wird **sequentiell konsistenter Speicher**, an den schwächere Bedingungen gestellt werden.

Eine Sequenz f ist sequentiell konsistent, wenn folgenden Bedingungen erfüllt sind:

- *Correct interaction*
  Für jeden Prozessor p_i besteht f_i (die Ausführungssequenz f eingeschränkt auf p_i) aus abwechselnden *invocations* und *responses*

- *Liveness*
  Auf jedes $read_i$ bzw. $write_i$ folgt ein $return_i$ bzw. ack_i

- *Sequential consistency*
  Es existiert eine Permutation μ über alle Operationen in f, so daß
  1. für jedes Objekt X, μ_X legal ist
  2. wenn die *response* der Operation o_1 in f an p_i vor der *invocation* der Operation o_2 an p_i kommt, dann erscheint o_1 in μ vor o_2

Im Gegensatz zur Linearisierbarkeit darf einer abgeschlossenen Operation eine spätere vorangestellt werden, sofern diese von unterschiedlichen Prozessoren stammen.

Beispiele:

f = **$write_0(x, 10)ack_0(x)write_1(x, 20)ack_1(x)read_2(x)return_2(x,10)$** war nicht linearisierbar, ist aber sequentiell konsistend, denn

$write_1(x, 20)ack_1(x)write_0(x, 10)ack_0(x)read_2(x)return_2(x, 10)$

ist die gewünschte Permutation.

f = **$write_0(x, 1)ack_0(x)write_1(y, 1)ack_1(y)read_0(y)return_0(y, 0)read_1(x)return_1(x, 0)$**

ist <u>nicht</u> sequentiell konsistent.

3.Sequentiell konsistenter Speicher - Bedeutung

- es existiert eine zeitliche Abfolge aller Operation, die aus der Sicht jedes MCS korrekt erscheint

3.Sequentiell konsistenter Speicher - Eigenschaften

- schwächer als Linearisierbarkeit
- linearisierbarer gemeinsamer Speicher ist auch sequentiell konsistent, die Umkehr gilt (wie gesehen) nicht
- gilt dennoch als strenge Konsistenzbedingung
- Algorithmen für einen MCS mit lokalem Lesen/Schreiben ist möglich

Algorithmus für MCS_i, $0 \le i \le n-1$

copy[x] enthält den Wert v für jedes gemeinsame Objekt x

when $read_i$(x) occurs:
    $return_i$(x, copy[x]);

when $write_i$(x, v) occurs:
    enable tob-$send_i$(<„write“, x, v>);

when tob-$recv_i$(<„write“, x, v) from MCS_j occurs:
    copy[x] = v;
    if j = i then ack_i(x);

Algorithmus für MCS_i, $0 \leq i \leq n-1$

copy[x] enthält den Wert v für jedes gemeinsame Objekt x
num = 0

when $read_i$(x) occurs:
  wait until *num* = 0;
  $return_i$(x, copy[x]);

when $write_i$(x, v) occurs:
  num = *num* + 1;
  enable tob-$send_i$(<„write“, x, v>);
  ack_i(x);

when tob-$recv_i$(<„write“, x, v) from MCS_j occurs:
  copy[x] = v;
  if j = i then *num* = *num* - 1;

Ein Algorithmus für sequentielle Konsistenz mit lokalem Schreiben **und** Lesen ist nicht möglich; eine Operation muß immer auf die Antwort aller anderen MCS warten.

Theorem 1: Für jedes sequentiell konsistente MCS, das zwei read/write-Objekte zur Verfügung stellt, gilt $t_{read} + t_{write} \geq d$. ($t_{read}$ und t_{write} sind worst-case-Zeiten für eine Lese- und Schreiboperation)

Beweis: Tafel, Anhang

Effizienz:

- entweder read- oder write-Operation benötigt eine konstante Anzahl von Nachrichten, eine Operation kann lokal durchgeführt werden
- bei gleicher Anzahl von read’s und write’s doppelt so schnell wie linearisierbarer Speicher
- in der Praxis werden manchmal Modelle mit noch schwächeren Konsistenzbedingungen eingesetzt

Theorem 2: Wenn $u = 0$ existiert eine linearisierbare Implementation von read/write-Objekten mit $t_{read} = 0$ und $t_{write} = d$.

Beweis:

when $read_i$(x) occurs:
    $return_i$(x, copy[x]);

when $write_i$(x, v) occurs:
    $send_{i,j}$(<x, v>) to all MCS_j, $j \neq i$
    sleep(d);
    copy[x] = v;
    ack_i(x);

when $recv_i$(<x, v>) from MCS_j occurs:
    copy[x] = v;

Weiter: Tafel, Anhang

Theorem 3: Im Vorhandensein einer globalen Uhr existiert eine linearisierbare Implementation von read/write-Objekten mit $t_{read} = 0$ und $t_{write} = d$.

Beweis:
- verschicke mit jede Nachricht die Zeit t, zu der sie verschickt wird
- ein MCS wartet genau $d - | t - t_{Nachricht} |$ bis er sie verarbeitet
- dadurch entsteht eine konstante Nachrichtenverzögerung d

Im Vorhandensein einer globalen Uhr ist der Zeitaufwand für sequentiell konsistenten gemeinsamen Speicher und linearisierbaren gemeinsamen Speicher gleich.

5.Anhang - Beweis Theorem 1

Theorem 1: Für jedes sequentiell konsistente MCS, das zwei read/write-Objekte zur Verfügung stellt, gilt $t_{read} + t_{write} \geq d$.

Beweis: Gegeben sei ein sequentiell konsistentes MCS, das zwei read/write-Objekte zur Verfügung stellt, beide anfangs 0.

Annahme: $t_{read} + t_{write} < d$.

Weiter sei die Verzögerung für jede Nachricht d.

Es existiert zulässige Ausführung $f' = write_0(x, 1)ack_0(x)read_0(y)return_0(y, 0)$.

Dann hat kein MCS eine Nachricht erreicht, bevor $return_0(y, 0)$ zurückgegeben wird.

Weiter existiert eine zulässige Ausführung $f'' = write_1(y, 1)ack_1(y)read_1(x)return_1(x, 0)$.

Wieder hat kein MCS eine Nachricht erreicht, bevor $return_1(x, 0)$ zurückgegeben wird.

Kombiniere nun f' und f'' zu f.

Die Operationen jedes Prozessors sind vor der Zeit d beendet, also weiß p_1 nicht von den Operationen an p_0 und umgekehrt. Also geben die MCS' in f die gleichen Werte zurück wie in f' und f''.

Damit ist f ist aber nicht sequentiell konsistent. Widerspruch!

5.Anhang - Beweis Theorem 2

Theorem 2: Wenn $u = 0$ existiert eine linearisierbare Implementation von read/write-Objekten mit $t_{read} = 0$ und $t_{write} = d$.

Beweis: Betrachte den Algorithmus aus Theorem 2. $t_{read} = 0$ und $t_{write} = d$.
Sei f eine zulässige Ausführung des Algorithmus. Ordne die Operationen in f zu einer Permutation μ, wobei die *response* an ihrem Platz bleibt, die *invocation* dieser Operation aber genau vor die *response* gestellt wird. Die Permutationsbedingung ist dadurch nicht verletzt.
Bleibt zu zeigen: μ ist legal. Oder, μ_X, für jedes read/write-Objekt X, ist legal.
Wähle ein beliebiges X. Dann ist $\mu_X = op_1 op_2$..... Sei $op_i = read_i(X) return_i(X, v)$

Fall 1: Keine write-Operation steht vor op_i in μ.
Nach Definition von μ wird kein write „acknowledged“ bevor op_i startet. Da das „acknowledge“ für eine write-Operation genau dann zurückgegeben wird, wenn jeder Prozess seine lokale Kopie von X aktualisiert, gibt op_i den initialen Wert von X zurück.

Fall 2: Ein $write_i(X, v)$ ist das letzte write, was vor op_i steht.Nach Definition von μ wird dieses write „acknowledged“ bevor op_i startet, aber kein anderes write wird „acknowledged“ bevor op_i startet. Da das „acknowledge“ für eine write-Operation genau dann gegeben wird, wenn jedes MCS seine lokale Kopie von X aktualisiert, gibt op_i genau diesen Wert zurück.

Literaturverzeichnis:

Attiya, Welch:

Distributed Computing, Fundamentals, Simulations and Advanced Topis
(Chapter 7, 8, 9)

Attiya, Welch:

Sequential Consistency versus Linearizability