APUVS, Blatt 6

Jan Fajerski and Kai Warncke and Magnus Müller

13. Dezember 2010

Aufgabe 6.1

Die Aufgabenstellung fragt, ob der *Chandy-Lamport-Algorithmus* auch dann einen konsistenten Schnitt erstellt, wenn er auf mehreren oder allen Prozessen gleichzeitig gestartet wird. Dies geschieht unter der Voraussetzung, dass alle beteiligten Prozesse bisher noch keine Markernachrichten bekommen haben.

Der Chandy-Lamport-Algorithmus funktioniert korrekt, auch wenn er gleichzeitig auf mehreren Prozessen gestartet wird.

Wir illustrieren das an dem durch Abbildung 1 gegebenen Beispiel. Wir beginnen in einem Zustand, der durch Abbildung 1(a) gegeben ist: Der Prozessgraph ist zusammenhängend und besteht aus 4 Prozessen und 4 Kommunikationskanälen. Nun wird der Chandy-Lamport-Algorithmus zeitgleich (zumindest ohne zu große Verzögerung dazwischen) auf den Prozessen 1 und 4 gestartet (vgl. Abbildung 1(b)). Der Start des Algorithmus funktioniert auch über Marker, welche diese beiden Prozesse bereits erhalten haben (nicht dargestellt). Diese aktivierten Prozesse senden an alle Kommunikationskanäle Marker, nachdem sie ihren internen Zustand gesichert haben (Abbildung 1(c)). Zudem starten sie danach die Aufzeichnung der eingehenden Nachrichten über die angeschlossenen Kommunikationskanäle. Nun folgt die in Abbildung 1(d) dargestelle Situation: Die Prozesse 2 und 3 erhalten ihrerseits ihre ersten Marker und beginnn somit, den Algorithmus auszuführen. Sie sichern also ihren internen Zustand und senden dann, wie in Abbildung 1(e) dargestellt, auch Marker an alle angeschlossenen Kommunikationskanäle. Wie an Abbildung 1(e) zu sehen ist, wurden aber noch nicht alle im System vorhanden Markernachrichten konsumiert. Dies ist eine wichtige Erkenntnis, denn dadurch sind die auf diesem Kanal benötigten Markernachrichten bereits vorhanden. Abbildung 1(f) beschreibt einen Ausschnitt aus dem System, nachdem die Prozesse 2 und 3 ihre Marker geschickt haben, diese aber noch nicht konsumiert wurden. Nun sind die für die Prozesse 1 und 4 nötigen Marker unterwegs und können konsumiert werden. Dadurch kommen 1 und 4 in den Endzustand und können den Algorithmus beenden. Die Prozesse 2 und 3 warten wiederrum noch auf eingehende Nachrichten. Wie an den Kanälen noch zu sehen, befinden sich immer noch Markernachrichten in den Puffern. Aufgrund der FIFO-Eigenschaft der Kanäle können diese Nachrichten nicht überholt werden. Das bedeutet, dass auch neue Nachrichten von den bereits mit dem Algorithmus fertigen Prozessen diese Marker nicht überholen werden. Somit werden die Marker später, wie in Abbildung 1(g) dargestellt, von den Prozessen 2 und 3 konsumiert und diese können wiederrum in den Endzustand übergehen.

Schlussfolgerung

Der Algorithmus ist also stabil gegenüber mehrfachem Start auf unterschiedlichen Prozessen, da die Marker nicht überholt werden können (FIFO-Eigenschaft). Es besteht also nicht die Gefahr, dass Nachrichten mit in den Schnitt aufgenommen werden, die zu spät abgesendet wurden – also solche Nachrichten, deren Sender bereits den Algorithmus abgeschlossen hat. In unserem Beispiel könnte der Prozess 1 in Abildung 1(f) bereits neue Nachrichten an Prozess 2 schicken, ohne dass dies den Schnitt kaputt machen würde, denn Prozess 2 wartet noch auf einen Marker und erhält diesen wegen der FIFO-Eigenschaft vor der neuen Nachricht von Prozess 1.
Damit besteht die *Grenze des Schnitts* (vergleiche Folie 35, Vorlesung 6) auch weiterhin nur aus dem Empfang von Markern auf allen eingehenden Kommunikationskanälen.

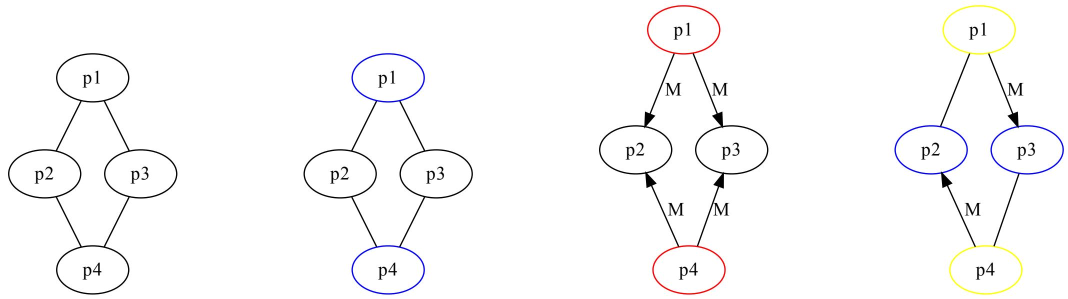
Aufgabe 6.2

Siehe Anhang auf Seite 5

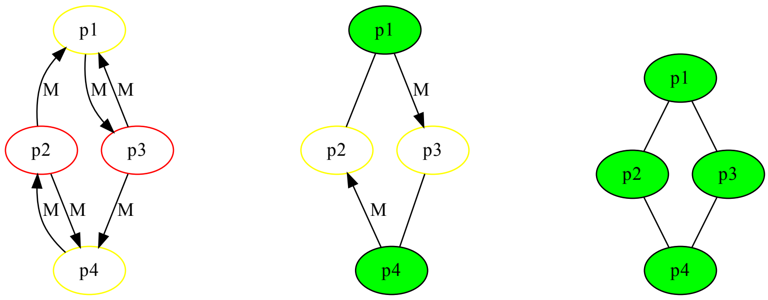
Aufgabe 6.3

Behauptung 1
$e \to e' \Leftrightarrow V(e) < V(e')$

Beweis 1
„$\Rightarrow$“. Voraussetzung: $e \to e'$. Zu zeigen: $V(e) < V(e')$.
Wir zeigen dies per Induktion über die Eigenschaften der *Happens-Before Relation*.



(a) Ausgangssituation (b) Algorithmus startet (c) Marker auf alle Kanäle (d) Erste Marker konsumiert



(e) Marker auf alle Kanäle (f) Einige sind fertig (g) Alle sind fertig

Abbildung 1: Beispiel zu Chandy-Lamport

Induktionsanfang.

Gelte $e \to e'$ wegen (HB1), also $e \to_i e'$. Offensichtlich gilt, genau wie bei der Lamportuhr, dass $V_i(e) < V_i(e')$ ist. Es gilt also noch zu zeigen, dass ebenso $V_j(e) \leq V_j(e'), \forall j \neq i$ zutrifft. Nehmen wir an, dies würde nicht zutreffen. Dann muss der entsprechende Prozess j, der diese Bedingung nicht erfüllt, mit Prozess i „synchronisiert“ worden sein[1] Das Ereignis e muss nach Voraussetzung vor Ereignis e' in i passiert sein. Wenn nun aber $V_j(e) > V_j(e')$

[1] Diese zwei Prozesse müssen also einmal in der Zwischenzeit ihre Zeitstempel über *send-receive* angepasst haben.

gilt, dann muss die Synchronisation von i und j nach e' passiert sein, jedoch vor e. e' kann also nicht vor e passiert sein, was im Widerspruch zur Voraussetzung steht.

Sei nun $e = \mathit{send}(m), e' = \mathit{receive}(m, t)$, wobei m eine Nachricht und t der Zeitstempel bei Versand von M sind. Es gilt also (HB2). Dann folgt offensichtlich $V(e) < V(e')$, da $V(e')$ als komponentenweises Maximum von $V(e)$ und dem im Empfängerprozess j aktuellen Vektor V' gebildet wird. In V' wird vor Empfang der Nachricht V_j inkrementiert, weshalb $V(e) \neq V(e')$ sein muss.

Induktionsschritt Es gelte $e \to e' \to e''$, also (HB2). Nach Induktionsvoraussetzung gilt $V(e) < V(e')$ und $V(e') < V(e'')$. Da die Relation $<$ aber transitiv ist folgt direkt $V(e) < V(e'')$, was zu zeigen war.
Somit ist die Vorwärtsrichtung erfolgreich bewiesen.

Beweis 2
„$\Leftarrow$“. Voraussetzung: $V(e) < V(e')$. Zu zeigen: $e \to e'$.
Falls e und e' Ereignisse des gleichen Prozesses i sind ist dies offensichtlich. Ebenso gilt dies offensichtlich, falls $e = \mathit{send}\ m$ und $e' = \mathit{receive}\ m$ ist. Zudem gilt für e, e', dass e'' existiert, so dass $V(e) < V(e'') <= V(e')$. Auf dieser Grundlage kann man nun folgendermaßen einen gerichteten Graph G mit folgenden Eigenschaften erstellen: Zwei beliebige Ereignisse e_n, e_m (oder auch Knoten) werden durch eine Kante (e_n, e_m) verbunden, falls

(a) Beide Ereignisse des gleichen Prozess i sind und $V_i(e_n) + 1 = V_i(e_m)$ (also direkt aufeinander folgende Ereignisse)

(b) Falls $e_n = \mathit{send\ message}$ und $e_m = \mathit{receive\ message}$ für eine Nachricht.

Der so konstruierte Graph ist also konsistent mit der *happens before* Relation, denn zwei Ereignisse sind nur direkt verbunden, falls das eine Ereignis vor dem anderen passiert.
Für diesen Graph gilt nun aber folgende Eigenschaft: Falls für zwei Ereignisse e und e' die Voraussetzung $V(e) < V(e')$ gilt, dann existiert ein Weg $w = (e, \ldots, e_a, \ldots e_b, \ldots, e')$ innerhalb des Graphen von e nach e'. Für zwei Ereignisse e_a, e_b des Weges w gilt aber $e_a \to e_b$, weshalb auch $e \to e'$ folgt.

Anhang

```
-module(lamdy).
-export([run/4, distributor/3, buyer/3]).


%%%%%%%%%%
%
% both distributor and buyer can use the following helper functions
%

record_state(Processname, Acc, Storage) ->
    io:format ("~s <Konto: ~B , Schrauben: ~B>\n", [Processname, Acc, Storage]).
send_marker(Receiver) ->
    case catch Receiver ! {snapshot, self()} of
        {'EXIT',_} ->
            io:format("I lost the connection to the receiver. Aborting.\n"),
            exit("Lost connection")
            ;
        _ -> true
    end.


%%%%%%%%%%
%
% Distributor
%

distributor(_, Storage, _) when Storage < 10 ->
    io:format("DIST: I don't have enough screws .... bye bye\n"),
    exit("distributor finishes");

distributor(Acc, Storage, false) ->
    receive
        {Number, Price} when is_integer(Number) and is_integer(Price) ->
            %send screws to buyer
            buy ! {Number},
            distributor(Acc + Price, Storage - Number, false);
        {snapshot, Sender} -> % received marker
            record_state ("Distributor", Acc, Storage),
            send_marker (buy),
            case Sender of
                system -> distributor(Acc, Storage, true); % initiate snapshot
                _ -> distributor(Acc, Storage, false) % received message on all incoming channels
            end
    end;

% record incmoing messages
distributor(Acc, Storage, true) ->
    receive
        {Number, Price} when is_integer(Number) and is_integer(Price) ->
            io:format("Distributor: received ~B screws for ~B money\n", [Number, Price]),
            buy ! {Number},
            distributor(Acc + Price, Storage - Number, true);
        {snapshot, _} -> % finished - we can only be here if the initial message was from the system
            distributor(Acc, Storage, false)
    end.

%%%%%%%%%%
%
% Buyer
%

buyer(Acc, _, _) when Acc < 10 ->
    io:format("BUYER: Darn... i need a dollar..dollar..dollar is what i need\n"),
    exit("Buyer finishes");

% no snapshot
buyer (Acc, Storage, false) ->
    distrib ! {10, 50},
    Newacc = Acc - 50,
    receive
        {Number} when is_integer(Number) ->
            buyer(Newacc, Storage + Number, false);
        {snapshot, Sender} -> % initiate snapshot
            record_state ("Buyer", Newacc, Storage),
            send_marker (distrib),
            case Sender of
```

```
                    system -> buyer(Newacc, Storage, true);
                    _ -> buyer(Newacc, Storage, false) % received message on all incoming channels
               end
     end;

% record incoming messages
buyer(Acc, Storage, true) ->
     % keep on sending messages
     distrib ! {10, 50},
     Newacc = Acc - 50,
     receive
          % record incoming messages
          {Number} when is_integer (Number) ->
               io:format("Buyer - received ~B screws\n", [Number]),
               buyer(Newacc, Storage + Number, true);
          {snapshot, _} -> % finished snapshot
               buyer(Acc, Storage, false)
     end.

run(Dacc, Dstore, Bacc, Bstore) ->
     Distributor = spawn(lamdy, distributor, [Dacc, Dstore, false]),
     register(distrib, Distributor),
     Buyer = spawn (lamdy, buyer, [Bacc, Bstore, false]),
     register(buy, Buyer),
     link(Distributor),
     link(Buyer),
     % create a snapshot
     Buyer ! {snapshot, system}.
```