APUVS, Blatt 6

Jan Fajerski and Kai Warncke and Magnus Müller

22. Februar 2011

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

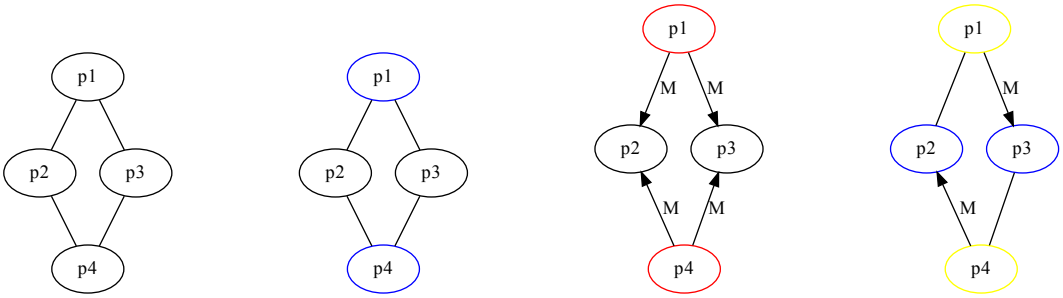
Der Algorithmus ist also stabil gegenüber mehrfachem Start auf unterschiedlichen Prozessen, da die Marker nicht überholt werden können (FIFO-Eigenschaft). Es besteht also nicht die Gefahr, dass Nachrichten mit in den Schnitt aufgenommen werden, die zu spät abgesendet wurden – also solche Nachrichten, 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

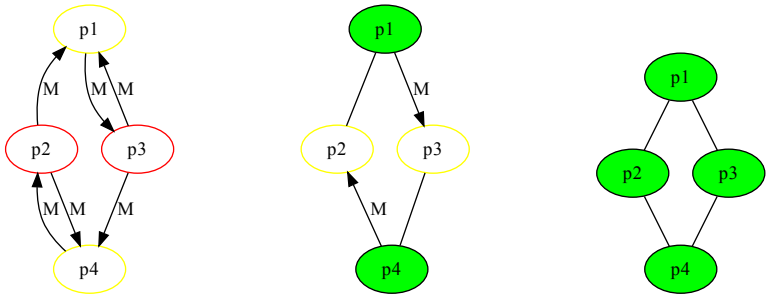
Siehe Anhang auf Seite 6. `lamdy.erl` enthält das Beispiel aus der Vorlesung, `snapshot.erl` die Implementation des Snapshotalgorithmus.

Anmerkungen

- Der Zustand der Prozesse muss vor Betreten der Snapshotfunktion gespeichert werden. Wir haben uns hierfür entschieden, da eine Abstraktion der Zustandsspeicherung nicht sinnvoll ist.
- Die Snapshotfunktion merkt sich alle empfangenen Nachrichten einer Liste. Ein aufrufender Prozess muss diese Liste nach Erstellung des Schnitts selbstständig wieder abarbeiten.



(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

› Der initiale Marker wird im Beispiel durch den `distributor` versendet, falls dessen Nachrichtenpuffer leer ist.

Aufgabe 6.3

Behauptung 1

Für zwei Ereignisse e, e' gilt:

$$e \to e' \iff V(e) < V(e')$$

Beweis 1

„$\Longrightarrow$“. Voraussetzung: $e \to e'$. Zu zeigen: $V(e) < V(e')$.
Wir zeigen dies per Induktion über die Eigenschaften der *Happens-Before Relation*.
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 $V_j(e) > V_j(e')$ 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 $e = send(m), e' = receive(m, t)$, 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

„$\Longleftarrow$“. 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 $e = send\ m$ und $e' = receive\ m$ 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 $e_n = send\ message$ und $e_m = receive\ message$ 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.

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

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 $e_a \rightarrow e_b$, weshalb auch $e \rightarrow e'$ folgt.

Anhang

```
-module(lamdy).
-export([run/0, run/4, distributor/2, buyer/2]).

%%%%%%%%%%%
%
% example
% - call lamdy:run() to start the example
%
% Note: the distributor initiates the snapshot if his message box is empty.
%

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

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

distributor(Acc, Storage) ->
    receive
        {Number, Price} when is_integer(Number) and is_integer(Price) ->
            %send screws to buyer
            buy ! {Number},
            distributor(Acc + Price, Storage - Number);
        {marker} ->
            UnprocessedMessages = snapshot:snapshot([buy], [buy]),
            distributor(Acc, Storage, UnprocessedMessages)
    after
        % if no message is buffered, do a snapshot
        0 -> io:format ("Distributor: Account: ~B\t\t Storage: ~B\n",[Acc, Storage]),
             UnprocessedMessages = snapshot:snapshot([buy], [distrib, buy]),
             io:format("\n"),
             distributor (Acc, Storage, UnprocessedMessages)
    end.

% Handle all messages which where left unprocessed in the snapshort algorithm
distributor (Acc,Storage,[]) ->
    distributor (Acc, Storage);

distributor (Acc,Storage, [{Order, Money}|T]) ->
    distributor (Acc + Money, Storage - Order, T).

%%%%%%%%%%%
%
% 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) ->
    distrib ! {10, 50},
    Newacc = Acc - 50,
    receive
        {Number} when is_integer(Number) ->
            buyer(Newacc, Storage + Number);
        {marker} ->
            io:format("BUYER: Account: ~B \t\t Storage: ~B\n", [Newacc, Storage]),
            UnprocessedMessages = snapshot:snapshot([distrib], [distrib]),
            buyer(Newacc, Storage, UnprocessedMessages)
    end.

% Handle all messages which where left unprocessed in the snapshort algorithm
buyer (Acc, Storage, []) ->
    buyer (Acc, Storage);

buyer (Acc, Storage, [{H}|T]) ->
    % we know how head looks as we know the structure of the messages
    buyer (Acc, Storage+H, T).


%%%%%%%%%%%
```

```
77 %
78 % Start methods
79 %
80 %
81 run () ->
82     run (1000,1000,1000,1000).
83
84 run(Dacc, Dstore, Bacc, Bstore) ->
85     Distributor = spawn(lamdy, distributor, [Dacc, Dstore]),
86     Buyer = spawn (lamdy, buyer, [Bacc, Bstore]),
87     link(Distributor),
88     link(Buyer),
89     register(distrib, Distributor),
90     register(buy, Buyer)
91     .
```

```
1  -module(snapshot).
2  -export([snapshot/2]).
3
4  %%%%%%%%
5  %
6  % snapshot
7  %
8  %  Channels are lists of processes
9
10 snapshot(OutgoingChannels, IncomingChannels) ->
11     snapshot(OutgoingChannels, IncomingChannels, 1).
12
13 snapshot(OutgoingChannels, IncomingChannels, NumberOfMarkers) ->
14     % send marker to each outgoing channel
15     sendMarkerToOutgoing (OutgoingChannels),
16     recordMessages (IncomingChannels, NumberOfMarkers, [])
17 .
18
19 % received marker on all incoming channels
20 recordMessages (IncomingChannels, NumberOfMarkers, ListOfSavedMessages) when length(IncomingChannels) ==
   NumberOfMarkers ->
21     lists:reverse (ListOfSavedMessages)
22     ;
23
24 recordMessages (IncomingChannels, NumberOfMarkers, ListOfSavedMessages) ->
25     receive
26         {marker} ->
27             recordMessages (IncomingChannels, NumberOfMarkers + 1, ListOfSavedMessages);
28         Msg ->
29             % record message
30             io:format ("Recorded ~w\n", [Msg]),
31             recordMessages (IncomingChannels, NumberOfMarkers, [Msg | ListOfSavedMessages])
32     end .
33
34 %%%%%%%%
35 %
36 % Send markers
37 %
38 sendMarkerToOutgoing ([]) -> true;
39 sendMarkerToOutgoing ([H|T]) ->
40     case catch H ! {marker} of
41             {'EXIT',_} ->
42                 io:format ("Lost my connection to ~w. Aborting.\n", [H]);
43             _ -> true
44     end,
45     sendMarkerToOutgoing(T).
```