Lösung Übungsblatt 11

Max Wisniewski

Aufgabe 42) *Rollercoaster Problem*

Nehmen Sie an, dass es n Passagierprozesse und einen Wagenprozess gibt. Die Passagiere warten immer wieder auf den Wagen, um Achterbahn zu fahren. Der Wagen kann $0 < c < n$ Passagiere aufnehmen. Der Wagen kann nur abfahren, wenn er voll ist, also c Passagiere in ihm sitzen.

1. Entwickeln Sie Pseudocode für die Aktionen der Passagiere und des Wagens, sowie einen Monitor um diese zu synchronisieren. Der Monitor soll drei Operationen haben: takeRide wird vom Passagier aufgerufen und load und unload werden vom Wagenprozess aufgerufen. Geben Sie eine Invariante für Ihren Monitor an und benutzen Sie signal-and-continue.
2. Verallgemeinern Sie Ihre Antwort zu 1 zu m Wagenprozesse ($m > 1$). Da es nur eine Spur gibt, können sichWagen nicht überholen, d.h. sie müssen in der Reihenfolge die Rundfahrt beenden, in der sie abgefahren sind. Ein Wagen darf wieder nur dann abfahren, wenn er voll ist.

Lösung:

1. Als erstes formulieren wir die folgende Invariante:

$$INV \equiv (riding \Rightarrow passagner = c) \wedge (\neg riding \Rightarrow 0 \leq passanger \leq c)$$

Wenn der Zug fährt, müssen genau c Passagiere anwesend sein. Sollte der Zug nicht fahren, können zwischen 0 und c Passagieren anwesend sein. Daraus ergibt sich das folgende Programm.

```
monitor Rollercoaster
  passanger := 0 : Nat;
  c : Nat;
  full, empty, finished : Condition
  riding := false : Boolean;

  procedure takeRide()
    do
      [] passanger > c -> wait(empty);
    od
    passanger++;
    if
      [] passanger = c -> signal(full);
      [] otherwise -> skip;
    fi

     wait(finished);

  procedure load()
    if
      [] passanger < c -> wait(full);
    fi
    riding := true;

  procedure unload()
    riding := false;
    passanger := passanger - c;
    signal_all(finished);
    signal_all(empty);
```

Wir beobachten als erstes, dass takeRide() zunächst jeden Passagier warten lässt, solange schon c Leute auf einen Zug warten. Daher ist der zweite Teil der Invariante immer erfüllt, da je ein Passagier den Zähler nur um eins inkrementieren kann. Sollte der Passagier der letzte anwesende sein, so benachrichtigt er den Zug, dass er nun losfahren kann.
In *load* gehen wir zunächst sicher, dass wir genau c Passagiere haben (beachte das nur takeRide den Zähler erhöht, aber wie eben genannt nicht auf einen Wert größer als c). Sobald dies der Fall ist, setzen wir riding auf wahr. Nun bleibt der erste Teil der Invariante erhalten, da wir c Passagiere haben. Der zweite Teil bleibt trivialer weise wahr.
Ruft der Zug dann *unload* auf, setzt er riding auf falsch, nimmt den counter um c herunter (setzt ihn also auf 0) und signalisiert alle zum Aussteigen und alle die Warten, dass nun eine neue Runde los geht. Riding ist wieder falsch, daher gilt der erste Teil trivialer weise und die Passagieranzahl liegt in der Begrenzung der zweiten Bedingung.
Unser Programm ist daher sicher, solange *unload* immer nach einem *load* aufgerufen wird.

Unser Programm ist darüber hinaus Deadlock frei. Wir haben insgesammt drei Stellen, an denen ein Prozess warten kann. Das *wait(finished)* wird immer terminieren, wenn wir zu jedem Zeitpunkt mindestens c Prozesse haben, die mit dem Zug fahren wollen und ein Zug vorhanden ist. Dies ist der Fall, weil ein Zug solange wartet, bis c Kunden da sind und vom letzten Signalisiert wird. Kommt der zug später, fährt er gleich los.
Beim ankommen werden alle Prozesse an *wait(finished)* frei gelassen.

Die letzte Stelle ist *wait(empty)*. Hier kommen die Prozesse nur nicht weiter, wenn sie von Prozessen außerhalb überholt werden. Da das überholen, aber nur möglich ist, wenn gerade Passagiere in den Zug passen und wir beim entladen des Zuges diese Prozesse signalisiert haben und darüber hinaus bei signal&continue entry<wait gilt, können wir nicht überholt werden.

2. Um uns den Verwaltungsaufwand kleiner zu machen, beobachten wir, dass die Züge einer Achterbahn in fester Reihenfolge auf den Schienen stehen. Wir weisen also jedem Zug eine Zahl von 1 bis m zu und führen Zähler *next* und *last* ein, die Speicher, welcher Zug als nächste beladen und entladen werden soll. Die Annahmen die wir treffen ist, dass alle Züge immer wieder ankommen und dass es unendlich viele Kunden gibt.

```
monitor Rollercoaster

  passanger := (0,...,0) : Nat[m];
  finished := (false,...,false) : Boolean[m];
  next, last := 0 , 0 : Nat;

  empty, toUnload, full, finished : Condition;

  takeRide()
    do
      [] passanger[next] > c -> wait(empty);
    od
    passanger[next]++;
    if
      [] passanger[next] = c -> signal_all(full);
      [] otherwise -> skip;
    fi

    myID = next;
    do
      [] finished[myId]  -> wait(finished);
    od
    if
      [] passanger[last] = 0 ->
        last := (last + 1) % m;
        finished[myId] := false;
        signal_all(toUnlad);
```

```
        signal_all(empty);
      [] otherwise -> skip
    fi

  load(id : Nat)
    do
      [] passanger[id] -> wait(full);
    od
    next := (next + 1) % m;
    signal_all(empty);

  unload(id : Nat)
    do
      [] id not= last -> wait(toUnload);
    od
    finished[id] := true;
    passanger[id] := 0;
    signal_all(finished);
```

Als Passagier warten wir nun, wenn im nächsten Zug kein Platz ist. Sollte einer frei sein, setzten wir uns signalisieren den Zug, falls wir wieder die letzten waren. Nun fahren wir so lange, bis unser Zug erstens fertig ist und zweitens an der Reihe zum aussteigen ist.
Der Zug wartet, bis er voll besetzt ist (da die Kunden wissen in welchen Zug sie einsteigen müssen, ist es dem Zug egal, ob er dran ist). Danach fährt er los, setzt den nächsten um eins weiter und signalisiert alle, dass sie zum nächsten Zug können.
Beim Aussteigen, wartet der Zug nun, ob er der ist, der entladen darf. Ist dies der Fall sagt er, dass er fertig ist setzt seine Mitfahrer wieder auf 0 und gibt das signal, dass er entladen werden darf.
Die Kunden die auf diesen Zug warten, werden nun sehen, dass der Zug da ist und der letzte setzt den Zähler eins weiter, sagt das der Zug nicht fertig ist und gibt nun zum einen den nächsten Zug zum ankommen frei und zum anderen sagt er den Kunden, dass ein neuer Zug da ist, falls gerade alle in Benutzung waren und er selber nun wieder der erste ist.

Diese Lösung noch eine einfache Lösung im SignalAll Java Stil. Will man das ganze anders Lösung, kann man mit PriorityConditions arbeiten um immer genau den Richtigen zu und richtigen Kunden zu bekommen oder man führt für jeden Wagen eine neue Condition ein.

Aufgabe 43) *Quicksort*

Betrachten Sie einen Filterprozess Partition mit der folgenden Spezifikation. Der Prozess Partition empfängt nicht sortierteWerte auf einem Eingangskanal in und sendet die Werte, die er empfängt, auf zwei Ausgangskanälen out1 und out2. Der Prozess Partition benutzt den ersten Wert v, den er empfängt, um die Eingabe in zwei Ströme zu partitionieren. Er sendet alleWerte, die nicht größer als v sind auf dem Kanal out1, einschließlich des ersten empfangenen Wertes. Er sendet alle Werte, die größer als v sind auf dem Kanal out2. Wenn schließlich der Wächter EOS empfangen wird, sendet er diesen auf beiden Kanälen out1 und out2.

1. Entwickeln Sie eine Implememtierung von Partition. Geben Sie zuerst Prädikate an, die den Inhalt der Kanäle beschreiben. Entwickeln Sie danach den Rumpf von Partition.

2. Zeigen Sie, wie man ein Sortierneztwerk von Prozessen Partition erzeugt. Nehmen Sie an, dass es N verschiedene Eingabewerte gibt und dass

Lösung:

1. Wir speichern uns zunächst das erste Element (Pivot) in einer Queue. Nun schicken wir jedes Element das kleiner ist in out1 und jedes Element das größer ist nach out2. Das Pivot und alle gleichen werden wir erst versenden, wenn wir das erste mal ein kleineres Element sehen, so werden Folgen von gleichen Elementen nicht sinnlos hin und her geschickt.

$$\begin{aligned} in &: \forall x \in in \ : \ x \in Int \\ out1 &: \forall x \in out1 \ : \ x \leq fst(in) \\ out2 &: \forall x \in out2 \ : \ x > fst(in) \end{aligned}$$

   Eine leichte verallgemeinerung bei random wahl des Pivot Elements, würde gelten

$$\forall p \in out1 \ \forall q \in out2 \ : \ p < q.$$

```
process Partition(in, out1, out2 : Channel)
  pivot,value : Int;
  pQueue : Queue<Int>;
  other := false : Bolean

  in?pivot;
  pQueue.add(pivot);
  do
    [] in?value AND value < pivot -> out1!value; other := true;
    [] in?value AND value > pivot -> out2!value;
    [] in?value AND value = pivot -> pQueue.add(pivot);
  od
  do
    [] ¬ pQueue.empty() AND other-> out1!pQueue.front();
  od
  out1!EOS;
  out2!EOS;
```

   Die Abfrage mit der Queue und other ist notwendig, da wir unter umständen keine Partition mehr erreichen, sondern nur die Liste umsortieren würden und dann komplett nach out1 zu schieben.

2. Mithilfe von Partition können wir nun ein Sortiernetzwerk aufbauen.
   Es gibt konkrete Berechnungsformeln, mit denen man die Form des resultierenden Baumes ermitteln kann, aber in unserem Fall reicht der leichteste Ansatz. Wir spawnen an einem Channel immer genau dann einen Prozess, wenn er uns neue Daten schickt.
   Sollten wir nur unesere Queue haben, wissen wir dass alles sortiert ist und können bei richtiger Organisation (Ablage des Baumes in einem Array) uns Prozess für Prozess die Wert aus den Buckets

holen.

Wollten wir alle Prozesse zuerst spawnen, würden wir einen Baum mit 2^n Blättern der Höhe n erhalten, da Quicksort keine gute Partition generiert.

Aufgabe 44) *Hammings Probelm*

Entwickeln Sie einen Algorithmus dessen Ausgabe eine aufsteigende Folge von vielfachen von 2, 3 und 5 ist. Die ersten Elemente dieser Folge sind 0, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... Es soll vier Prozesse geben: für jeden Faktor einen Prozess, der die Multiplikation durchführt und einen vierten Prozess, der die Ergebnisse verschmelzt. Benutzen Sie Kanäle zur Koordination und Kommunikation.

Lösung:
Wir konstruieren eine Filterkette. Jeder Prozess prüft, ob der die eingehende Zahl teilen kann. Als Nachricht schicken wir immer die Zahl und das bisherige Ergebnis einer Teilung, solange es möglich war. Die Zahl ist eine Hamming Zahl, wenn am Ende eine 1 der zweiten Komponente steht.

```
process teilbar(n : Nat, in , out : Channal)
  value,div : Nat;
  do
    [] in?(value,div) AND div % n = 0 -> out ! (value, div / n);
    [] in?(value,div) AND div % n > 0 -> out ! (value, div);
    [] in ?EOS -> out!EOS
  od

process Hamming
  two, three, five, erg : Channel;
  next := 2 : Nat;
  print(0);
  spawn(2, two.out, three.in);
  spawn(3, three.out, five.in);
  spawn(5, five.out, erg.in);

  do
    [] true -> two.in ! next; next++;
    [] erg.out?(value,div) AND div = 1 -> print(value);
    [] erg.out?(value,div) AND div /= 1 -> skip;
  od
```

Der Ergebnisprozess legt zunächst 4 Channel an (ein Channel mit je Ein- und Ausgang). Danach spawned er die Prüfprozesse und verbindet sie mit der erzeugten Channels. Die 0 gibt er zunächst aus und wird dann mit der 2 fortfahren.

Bekommt er eine Nachricht und der dividierte Teil ist 1, so kann die Zahl nur aus den genanten Primfaktoren bestehen. Sollte der geteilte Rest nicht 1 sein, muss es noch einen anderen Primteiler geben und es ist keine Hammingzahl.
Immer zwischendurch werden wir neue Zahl auf den Weg schicken. Auf diese weise blockieren wir uns nicht und bekommen eine unendliche Ausgabe von Hammingzahlen. (Ob es nun print ist, oder eine andere Ausgabe sollte egal sein).

Aufgabe 45) *Saving Account Problem*

Ein Sparkonto wird von verschiedenen Personen benutzt. Eine Person kann Geld auf das Konto einzahlen oder Geld vom Konto abbuchen. Der aktuelle Kontostand ist die Summe aller Einzahlungen minus der Summe aller Auszahlungen. Der Kontostand darf niemals Negativ sein. Entwickeln sie einen Dienstleister (Server), um dieses Problem zu lösen, und zeigen Sie die Schnittstelle des Dienstleisters zum Dienstnutzer (Client). Dienstnutzer können Anfragen zum Einzahlen von amount Geldeinheiten und Anfragen zum Abheben von amount Geldeinheiten senden. Die Anfrage zum Abheben muss solange verzögert werden, bis ausreichend Mittel vorhanden sind. Nehmen Sie an, dass amount immer positiv ist.

Lösung:

In dieser Lösung machen wir uns die Eigenschaft zu nutze, dass Nachrichten im Channel verbleiben können. So spart man in der Lösung viel Platz. Wir nehmen an dieser Stelle an, dass wir n Kunden haben und jeder eine eindeutige Zahl zwischen 1 und n hat. Der initiale Geldbestand wird dem Server mit savings übergeben.

```
process SavingAccountServer (request : Channel, answers : Channel[n],
   savings : Int)
  do
    [] request?(pid, "deposit", x) ->
      savings + x;
    [] request?(pid, "withdraw", x) AND x ≤ savings ->
      savings - x;
      answers[pid]!SUCCESS;
  od

procedure deposit(pid : Nat, x : Nat)
  request!(pid, "deposit", x);

procedure withdraw(pid : Nat, x : nat)
  request!(pid, "withdraw", x);
␣␣answers[pid]?SUCCESS;
```

Das Protokoll / Interface zwischen Server und Klienten sieht wie folgt aus. Auf dem *request* Kanal wird eine Nachricht verschickt, die die Form hat

1: Die eigene ID, auf der die Antwort zurück kommen soll (pid).

2: Die Aktion die mit dem Betrag ausgeführt werden soll, zum einen "depositßum anderen "withdraw".

3: Der Betrag, für die Aktion.

Deposit ist eine nicht blockierende Methode, wohingegen withdraw blockiert, bis der Server das Geld hat und mit einer einfache Nachricht SUCCESS antwortet.