Frage: Schränkt Linearisierbarkeit als Korrektheitseigenschaft die mögliche Nebenläufigkeit eines Programms ein?

↳ Nein!

Progress Conditions

Linearisierbarkeit ist eine nicht-blockierende (nonblocking) Korrektheitseigenschaft.
D.h. für jeden Methodenaufruf existiert ein passendes Methodenende, unabhängig davon, was die anderen Threads tun.
Dies setzt eine totale Spezifikation voraus.

Definition

Eine Spezifikation ist total, falls für alle sequentiellen und gültigen Historien S und alle Methodenaufrufe $\langle A, x.m(\bar{a}) \rangle$ ein passendes Methodenende $\langle A, x:\bar{r} \rangle$ existiert, sodass $S.\langle A, x.m(\bar{a}) \rangle.\langle A, x:\bar{r} \rangle$ gültig ist.

Satz: Linearisierbarkeit ist nicht-blockierend

Sei $\langle A, x.m(\bar{a}) \rangle$ ein offener Methodenaufruf in einer linearisierbaren History H bzgl. einer totalen Spezifikation.
Dann existiert ein Methodenende $\langle A, x:\bar{r} \rangle$ mit:

$H.\langle A, x:\bar{r} \rangle$ ist linearisierbar.

Beweis:

Sei H linearisierbar bzgl. S.
Nach Def. ex. eine Erweiterung $H.R$ von H mit:

$$\text{compl}_e(H.R) \text{ ist äquivalent zu } S.$$

Sei $\langle A, x.m(\bar{a}) \rangle$ offen in H.

- Falls ein passendes Methodenende in R ex., so ist $H.R$ auch eine Erweiterung von $H.\langle A, x:\bar{r} \rangle$. Ferner bleibt S eine Linearisierung von $H.\langle A, x:\bar{r} \rangle$.
- Andernfalls ist $\langle A, x.m(\bar{a}) \rangle$ nicht in $\text{compl}_e(H.R)$ enthalten, damit also auch nicht in S.
  Nach Totalität ex. $\langle A, x:\bar{r} \rangle$ mit $S' = S.\langle A, x.m(\bar{a}) \rangle.\langle A, x:\bar{r} \rangle$ gültig (und sequentiell).
  Ferner ist $\text{compl}_e(H.R.\langle A, x:\bar{r} \rangle)$ äquivalent zu S'.
  und $H.R.\langle A, x:\bar{r} \rangle$ ist Erweiterung von H.

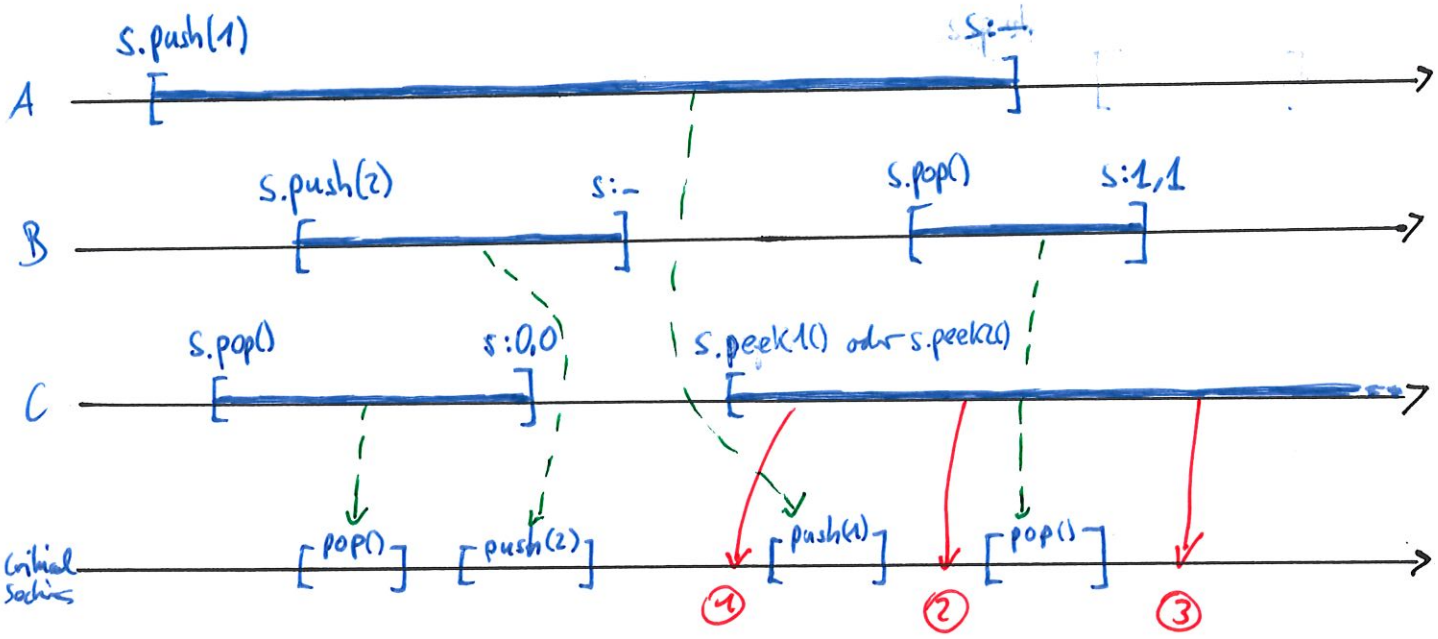
Also ist H linearisierbar bzgl. S'.

∎

Beispiel:

```
class LStack {
    Node* head = NULL;

    void push(int x) {
        lock();
        Node* node = new Node();
        node->data = x;
        node->next = head;
        head = node;
        unlock();
    }

    (bool, int) pop() {
        lock();
        bool flag = false; int x = 0;
        if (head != NULL) {
            x = head->data;
            head = head->next;
            flag = true;
        }
        unlock();
        return flag, x;
    }

    (bool, int) peek1() {
        lock();
        bool flag = false; int x = 0;
        if (head != NULL) {
            flag = true;
            x = head->data;
        }
        unlock();
        return flag, x;
    }

    (bool, int) peek2() {
        Node* top = head;
        bool flag = top != NULL;
        int x = flag ? top->data : 0;
        return flag, x;
    }
}
```

Bemerkung: im Sequentiellen verhalten sich peek1 und peek2 gleich.



① s: 1,2

② s: 1,1

③ s: 1,2 Dieses Methodenende wird im obigen Beweis genutzt

Beobachtung:

Linearisierbarkeit erlaubt (mehrere) Methodenenden für das offene pop() von C.

Die Implementierung verbietet diese ggf., da ein Thread auf das globale Lock warten muss.

D.h. ein Thread wird blockiert, da er auf einen anderen Thread warten muss.

Falls ein Thread nicht durch andere Threads blockiert werden kann, so ist die Implementierung nicht-blockierend.

Diese Eigenschaft hat mehrere Ausprägungen.

Wir unterscheiden zwischen den folgenden Progress Guarantees:

1) Blocking (blockierend).

Ein Thread A kann u.U. unbeschränkt lange auf die Ausführung eines anderen Threads B warten.

Programme, die Locks enthalten, fallen in diese Kategorie.

2) Non-blocking (nicht-blockierend).

Threads müssen in bestimmten Situationen (!) nicht auf andre Threads warten.

a) Wait-free.

Eine Methode (ein Programm) ist wait-free, falls jeder Methodenaufruf nach endlich vielen Schritten beendet werden kann.

b) Lock-free (lock-frei).

Eine Methode (ein Program) ist lock-free, falls unendlich oft ein Methodenaufruf in endlich vielen Schritten beendet wird.

(Oft auch so beschreiben: ..., falls es zu jedem Zeitpunkt einen Thread gibt, der "Fortschritt" erzielt.)

c) Obstruction-free.

Eine Methode (ein Programm) ist obstruction-free, falls zu jedem Zeitpunkt jeder Thread in endlich vielen Schritten den Methodenaufruf in Isolation beenden kann.

↳ ohne, dass andre Thread ausgeführt wurde

"sequentielles Fortsetzen einer parallelen Ausführung"

Beobachtung: wait-free ⇒ lock-free ⇒ obstruction-free.

Achtung: Das Nichtvorhandensein von Locks impliziert nicht non-blocking! /5

Am Beispiel: LStack

- LStack ist blockierend
- push, pop, peek1 sind blockierend
- peek2 ist nicht-blockierend
- peek2 ist wait-free

↳ Oft bei Datenstrukturen: - Methoden, die die DS nicht verändern, sind wait-free.
- andere Methoden sind blockierend aber lock-frei.

Bemerkung zur Praxis:

- Wait-free Datenstrukturen sind selten. Stattdessen werden lock-freie DS verwendet, da sie eine bessere Performance zeigen.

Also: - lock-frei liefert bessere Performance, aber kann im Gegensatz zu wait-free DS keine Thread-Starvation verhindern.

Es kann Threads geben, die ihre Ausführung nie beenden.

- Thread-Starvation ist in lock-freien DS selten.

- Lock-freie Datenstrukturen können (in gewissen Fällen) in wait-free Datenstrukturen überführt werden.

[Timnat & Petrank '14]

Linearisierbarkeitsnachweis:

Schwierigkeit: Linearisierbarkeit verlangt das Betrachten aller Historien, also aller möglichen Ausführungen eines Programms.

Lösung:
- Wir wissen, dass Linearisierbarkeit die Existenz eines einzigen Zeitpunktes in jeder Methode verlangt, zu dem die Methoden ihren vollständigen Effekt ausübt.
- Identifiziere diesen Linearisierungspunkt.
  - Bei der Verwendung von Locks befindet sich der Linearisierungspunkt / ist der Linearisierungspunkt typischerweise die Critical Section.
  - Ansonsten sind Compare-And-Swap (CAS) Operationen ein guter Kandidat (reichen aber oft nicht aus).

Beispiel:

```
class LStack {
    Node* head = NULL;

    void push(int x){
        lock();
        Node* node = new Node();
        node->data = x;
        node->next = head;
        head = node;
        unlock();
    }

    (bool,int) peek1() {
        lock();
        bool flag = false;
        int x = 0;
        if (head != NULL) {
            flag = true;
            x = head->data;
        } else { }
        unlock();
        return flag,x;
    }

    (bool,int) pop() {
        lock();
        bool flag = false;
        int x = 0
        if (head != NULL) {
            x = head->data;
            head = head->next;
            flag = true;
        } else { }
        unlock();
        return flag,x;
    }

    (bool,int) peek2() {
        Node* top = head;
        bool flag = top != NULL;
        int x = flag ? top->data:0;
        return flag, x;
    }
}
```

Linearisierungspunkte:

① neuer Knoten mit Wert x eingefügt: ⟨A, s.push(x)⟩⟨A, s:-⟩

② Knoten mit Wert x entfernt: ⟨A, s.pop()⟩⟨A, s:1,x⟩

③ Kein Knoten entfernt, Stack ist leer: ⟨A, s.pop()⟩⟨A, s:0,0⟩

④ Oberster Knoten enthält Wert x: ⟨A, s.peek1()⟩⟨A, s:1,x⟩

⑤ Stack leer: ⟨A, s.peek1()⟩⟨A, s:0,0⟩

↝ Linearisierungspunkte schreiben die Linearisierung ("history s") vor. Es verbleibt zu prüfen, ob die Methoden die vorgeschriebenen Rückgabewerte tatsächlich zurück liefern.

Linearisierungspunkte ①-⑤ geben genau vor was passiert muss. Wenn der Stack z.B. leer ist, muss peek1() liefern: 1,1. vgl ⑤.

Man sagt, dass diese Linearisierungspunkte keine Nebenbedingung haben (unconditional).

Linearisierungspunkt ⑥ hingegen ist conditional. Das passende Methodenende hängt vom tatsächlichen Wert von head und head→data ab.

Also:

⑥ Oberster Wert ist x: ⟨A, s.peek2()⟩.⟨A, s:1,x⟩
falls: head != NULL und x = head→data

⑥ Stack ist leer: ⟨A, s.peek2()⟩.⟨A, s:0,0⟩
falls head == NULL

Berechnen von Linearisierungen

Um mit Linearisierungspunkten die Linearisierbarkeit eines Objekts x nachzuweisen, führen wir dieses aus und generieren eine Linearisierung S.

Dazu:

- Starte mit $H = \varepsilon = S$ und $R = \emptyset$
- Führe x aus // berechne alle Ausführungen
  - ↳ Falls Thread A die Methode $m(\bar{a})$ aufruft, setze $H := H.\langle A, x.m(\bar{a})\rangle$
  - ↳ Falls Thread A einen Linearisierungspunkt in $m(\bar{a})$ erreicht, konstruiere $S' := S.\langle A, x.m(\bar{a})\rangle\langle A, x:\bar{r}\rangle$ so, dass S' gültig ist.
    Hier gehen wir davon aus, dass die Spezifikation total ist. Demnach existiert ein solches $\langle A, x:\bar{r}\rangle$.
    Vereinfachend nehmen wir an, das $\langle A, x:\bar{r}\rangle$ eindeutig determiniert ist.

    Setze: $S := S'$
    $R := R \cup \{\langle A, x:\bar{r}\rangle\}$
  - ↳ Falls Thread A eine Methode mit Rückgabewerten $\bar{r}$ beendet:
    - ↳ Falls $\langle A, x:\bar{r}\rangle \notin R$: Abbruch ⇝ nicht linearisierbar!
      - ↳ "falsche" Rückgabe oder kein Linearisierungspunkt
    - ↳ Sonst setze: $R := R \setminus \{\langle A, x:\bar{r}\rangle\}$
      $H := H.\langle A, x:\bar{r}\rangle$
- Falls kein Abbruch und fertig, so ist H linearisierbar, S ist Zeuge dafür.

Bemerkung:

- Das Verfahren wird erfolgreich beendet wenn x linearisierbar ist.
- Das Verfahren benötigt korrekt plazierte Linearisierungspunkte, um erfolgreich zu sein (Linearisierbarkeit zu zeigen).
- Falls die Linearisierungspunkte falsch gewählt oder gar nicht gewählt sind, so bricht das Verfahren ab. Es wird kein falsches Ergebnis geliefert.
- Also können wir Linearisierungspunkte raten und damit verifizieren.

Linearisierungspunkte in der Praxis:

- Für nicht-blockierende Programme sind Linearisierungspunkte schwer zu finden (und schwer „sinnvoll" zu raten).
- Bei komplexeren Datenstrukturen wie z.B. Harris' Set treten zwei Typen von Linearisierungspunkten auf:

  1) fixed: wie bisher. Linearisierungspunkt befindet sich an einem Command und wird zusammen mit diesem „ausgeführt". Ggf. ist der Linearisierungspunkt conditional.

  2) non-fixed: der Linearisierungspunkt befindet sich in einem anderen Thread. D.h. es hängt von anderen Threads $A_1, \ldots, A_n$ ab, wann/ob Thread B ein Linearisierungspunkt erfährt.
  => eine einfache Annotation im Code ist nicht mehr möglich