Nichtsequentielle und Verteilte Programmierung SoSe 2018

Anton Oehler, Jona Rex

Abgabe am 11. Mai 2018 bis 10 Uhr

Aufgabe 1 Zustandsdiagramme *10 Punkte*

a)

b)

Der Zustand (a_2, b_2, b) gewährt durch den Zustand der geteilten Variable dem Prozess `b` den Zugang zum kritischen Abschnitt, während Prozess `a` in einer Warteschleife zwischen diesem und dem Zustand (a_3, b_2, b) *feststeckt*. Da Prozess `b` aufgrund schwacher Fairness irgendwann ausgeführt wird, verhindert diese Priorisierung Deadlocks.

Aufgabe 2 Der Algorithmus von Peterson

(a) Der Algorithmus führt eine zusätzliche globale Variable `letzter` ein, die festhält, welcher Prozess zuletzt den Eintritt in den kritischen Abschnitt gefordert hat. Der Prozess, der zuletzt gefordert hat, wartet nun, bis der andere Prozess seinen kritischen Abschnitt verlassen hat, signalisiert durch die globalen Variablen `adrin` und `bdrin`. Dadurch ist eindeutig geregelt, welcher der beiden Prozesse seinen kritischen Abschnitt betreten darf, falls beide fordern.

Der Algorithmus hat mit dem Dekkers Algorithmus gemein, dass beide Prozesse eine Variable besitzen, die anzeigt, dass die ihren kritischen Abschnitt ausführen möchten. Im Gegensatz zu Dekkers Algorithmus, nimmt der Prozess, der nicht im kritischen Abschnitt ist, nicht seine Forderung zurück um einen Deadlock zu verhindern. Stattdessen wird die Reihenfolge vor dem betreten der Schleifen festgelegt, als die Reihenfolge mit der das Betreten gefordert wurde.

Aufgabe 3 CMPXCHG

`common <- 0`

a	b
`U`	`U`
`until common != 1 do`	`until common != 2 do`
`    CMPXCHG(common, 0, 1)`	`     CMPXCHG(common, 0, 2)`
`K`	`K`
`common <- 0`	`common <- 0`

Tabelle 1: Thread-Safer Algorithmus mit CMPXCHG als atomare Anweisung

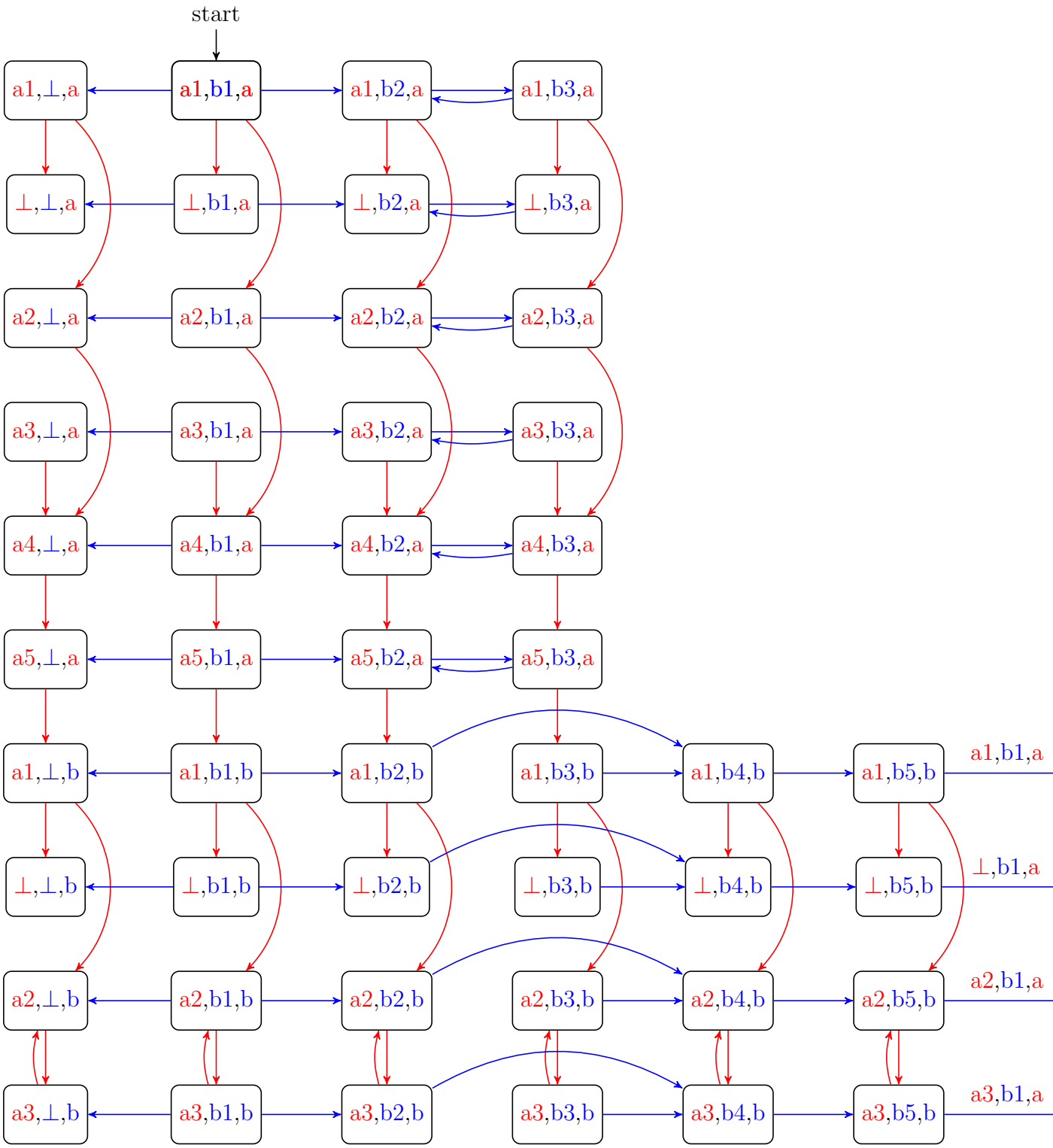

Abbildung 1: Teilaufgabe a)

common <- 0

a	b

`until common != 1 do`
`  CMPXCHG(common, 0, 1)`
`common <- 0`
	`until common != 2 do`
	`  CMPXCHG(common, 0, 2)`
	`common <- 0`

Tabelle 2: Abgekürzter Thread-Safer Algorithmus mit CMPXCHG als atomare Anweisung

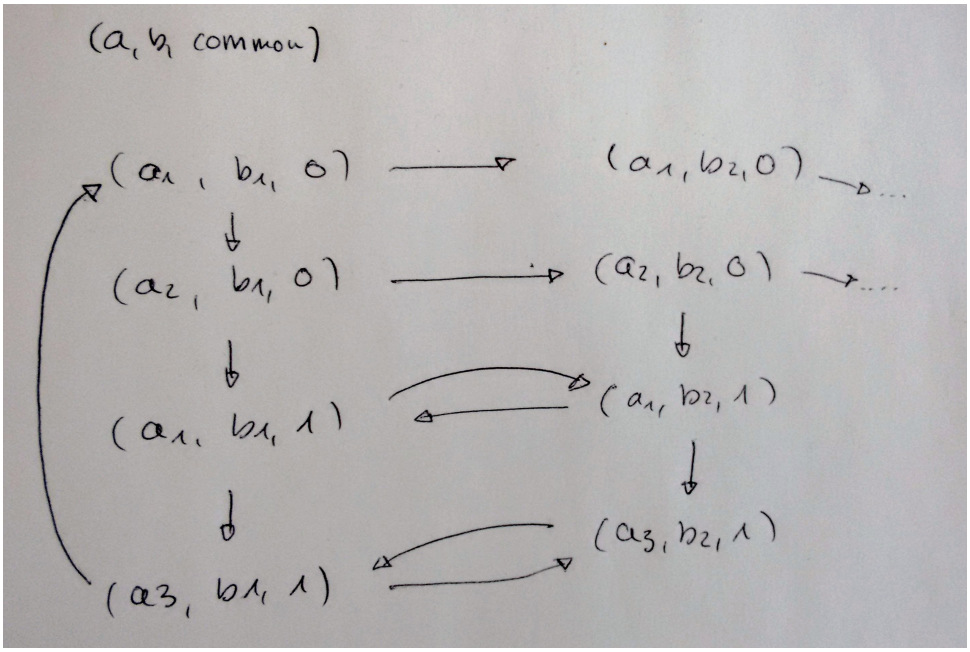

Abbildung 2: Zustandsdiagramm des vereinfachten Algorithmus

Aus dem in Abbildung 1 dargestellten Zustandsdiagramm können wir die Korrektheit des Algorithmus nachvollziehen. Der Einfachheit halber ist in dem Diagramm nur der Fall dargestellt, in dem Prozess `a` den kritischen Bereich betritt. Der andere Fall ist symmetrisch.

Zunächst einmal lässt sich feststellen, dass nie der Zustand $(a_3, b_3, 1)$, $(a_3, b_3, 0)$ oder $(a_3, b_3, 2)$ angenommen wird. *Mutual exclution* ist somit erfüllt.

Da die Variable `common` im kritischen Abschnitt zurück auf 0 gesetzt wird und damit früher oder später (auf Grund schwacher Fairness) der Anfangszustand erreicht wird, wird irgendwann jeder Prozess die Möglichkeit haben, in seinen kritischen Abschnitt zu gelangen. Damit ist die *Freedom from Starvation* sichergestellt.

Zu guter Letzt muss noch gezeigt werden, dass es keine Deadlocks gibt. Dies ist gewährleistet durch die `CMPXCHG` Funktion, welche in diesem Kontext nur das Überschreiben der geteilten Variable durch den Prozess, von dem sie zuerst aufgerufen wird, zulässt.

Aufgabe 4 Der Algorithmus von Peterson *10 Punkte*

a1: U	b1: U
a2: adrin ← true	b2: bdrin ← true
a3: letzter ← a	b3: letzter ← b
a4: while bdrin && letzter = a do	b4: while adrin && letzter = b do
a5: NOP	b5: NOP
a6: K	b6: K
a7: adrin ← false	b7: bdrin ← false

Der Algorithmus von Peterson löst das Problem des wechselseitigen Ausschlusses, indem beide Prozesse zuerst ihr *Interesse am Ausführen des kritischen Abschnittes* bekunden, indem sie *adrin* bzw *bdrin* auf *true* setzen. Haben beide Prozesse gleichzeitig ihr Interesse bekundet, darf der Prozess seinen kritischen Abschnitt betreten, welcher zuletzt *letzter* auf a/b gesetzt hat, der andere muss warten bis dieser Prozess mit $adrin/bdrin \leftarrow false$ sein Interesse wieder zurückzieht und seinen unkritischen Abschnitt ausführt.

Stürzt einer der Prozesse in U ab, so ist seine *drin* Variable auf $false$ gesetzt und der andere Prozess kann ungehindert seinen kritischen Abschnitt betreten. Somit ist *Freedom from starvation* auch gelöst.

Die Ähnlichkeit zum Algorithmus von Decker wird besonders deutlich, wenn man eine etwas andere Variante des Algorithmus von Peterson betrachtet, in der die Variable *letzter* wieder durch *dran* ersetzt wird und a und b bei der Zuweisung vertauscht werden ($a3 : letzter \leftarrow b$ und $a4 : \ldots letzter = b$). Durch die Umbenennung/ das vertauschen von a und b ändert sich die allgemeine Funktionsweise natürlich nicht, es wird jedoch von der Bedeutung her bei gleichzeitigem Interesse dem anderen Prozess der Vortritt gelassen, wenn nur einer der Prozesse Interesse bekundet kann dieser ungehindert K ausführen. Besonders wurde die verschachtelte while-Schleife in der if-Abfrage in der while-schleife zu einer einzigen while-Schleife mit 2 Bedingungen gemacht, wodurch der Algorithmus kürzer ist.

Das Grundprinzip der beiden Algorigmen ist aber das gleiche: es wird vor Betreten des kritischen Abschnittes *Interesse bekundet* und bei gleichzeitigem Interesse einem Prozess der Vortritt gegeben.