

Synchronisation Teil II: Stärkere Primitiven

Prof. Dr. David Sabel

LFE Theoretische Informatik



- 1 Stärkere Speicheroperationen
 - Übersicht über die Operationen
 - Mutual-Exclusion Algorithmen mit test-and-set-Bits
 - Ein Mutual-Exclusion Algorithmus mit RMW-Objekt
 - Der MCS Queue-basierte Algorithmus mit Compare and Swap

- 2 Konsensus und die Herlihy-Hierarchie

- Nur Lesen und Schreiben sind atomare (unteilbare) Operationen
- atomare Leseoperation: `if $x = 10$ then ...`
- atomare Schreiboperation: `$x := 10$`

- Hardware unterstützt verschiedene stärkere Speicheroperationen
- Formal stellen wir diese Operationen als Funktionen dar

```
function xyz( $p_1 : \mathbf{Typ}_1, \dots, p_n : \mathbf{Typ}_n \dots$ ) returns : Typ
```

“atomare Ausführung des Rumpfs”

```
end function
```

- Hardware unterstützt verschiedene stärkere Speicheroperationen
- Formal stellen wir diese Operationen als Funktionen dar

Funktionsname

Parameter mit Typen

Rückgabebetyp



```
function xyz(p1 : Typ1, ..., pn : Typn...) returns : Typ
```

“atomare Ausführung des Rumpfs”

```
end function
```

- **Register** = gemeinsames Speicherregister
- **Lokales Register** = lokales Speicherregister
- **Wert** = ein Wert wie True, 1, (keine weitere Unterscheidung)
- **Funktion** = eine Seiteneffekt-freie Funktion auf Werten

Read und Write in der Funktionsnotation

read(r)

```
function read( $r$  : Register) returns : Wert  
  return( $r$ ) :  
end function
```

write(r, v)

```
function write( $r$  : Register,  $v$  : Wert)  
   $r := v$   
end function
```

Read und Write in der Funktionsnotation

read(r)

```
function read( $r$  : Register) returns : Wert  
  return( $r$ ):  
end function
```

write(r, v)

```
function write( $r$  : Register,  $v$  : Wert)  
   $r := v$   
end function
```



kein returns, da kein Rückgabewert!

“Nebenläufiges Objekt”:

Nebenläufige Datenstruktur mit atomaren Operationen

“Nebenläufiges Objekt”:

Nebenläufige Datenstruktur mit atomaren Operationen

Beispiel:

Atomares Speicherregister (als Objekt)

Gemeinsames Speicherregister mit

- atomarer *read*-Operation
- atomarer *write*-Operation

test-and-set(r, v)

Wert v wird dem Register r zugewiesen, Rückgabewert: **alter** Wert von r .

```
function test-and-set( $r$  : Register,  $v$  : Wert) returns : Wert  
   $temp := r$ ;  
   $r := v$ ;  
  return( $temp$ );  
end function
```

test-and-set(r, v)

Wert v wird dem Register r zugewiesen, Rückgabewert: **alter** Wert von r .

```
function test-and-set( $r$  : Register,  $v$  : Wert) returns : Wert  
   $temp := r$ ;  
   $r := v$ ;  
  return( $temp$ );  
end function
```

Variante: Register, darf nur 0 oder 1 sein.

```
function test-and-set( $r$  : Register) returns : Wert  
   $temp := r$ ;  
   $r := 1$ ;  
  return( $temp$ ) :  
end function
```

Test-and-set-Objekt

Ein gemeinsames Speicherregister mit

- *write*-Operation
- *test-and-set*-Operationen

Test-and-set-Bit

Ein gemeinsames Speicherregister mit Werten 0 und 1

- alternative *test-and-set*-Operationen setzt Register auf 1
- *reset*-Operation: atomares Schreiben einer 0 in das Register,

Test-and-test-and-set-Objekt

Ein Test-and-set Objekt, das zusätzlich die atomare *read*-Operation unterstützt

swap

Wert von lokalem Register mit gemeinsamen Register tauschen:

```
function swap(r : Register, l : Lokales Register)  
    temp := r;  
    r := l;  
    l := temp;  
end function
```

wird manchmal auch als *fetch-and-store* bezeichnet.

Swap-Objekt

Ein gemeinsames Speicherregister mit

- *swap*-Operation zwischen dem Register und jedem beliebigen lokalen Register unterstützt.

fetch-and-add

Erhöhen des Registerwerts um v , Rückgabe: alter Wert

```
function fetch-and-add( $r$  : Register,  $v$  : Wert) returns : Wert  
   $temp := r$ ;  
   $r := temp + v$ ;  
  return( $temp$ );  
end function
```

Variante: *fetch-and-increment*: genau 1 wird dazu addiert

Fetch-and-increment-Objekt

Ein gemeinsames Speicherregister mit

- *fetch-and-increment*-Operation
- *write*-Operation
- *read*-Operation

Fetch-and-add-Objekt

Ein gemeinsames Speicherregister mit

- *fetch-and-add*-Operation
- *write*-Operation
- *read*-Operation

read-modify-write

Anwenden einer Funktion auf Registerwert. Rückgabe: alter Wert.

```
function read-modify-write(r : Register, f : Funktion)  
  returns : Wert  
  temp := r;  
  r := f(temp);  
  return(temp);  
end function
```

read-modify-write

Anwenden einer Funktion auf Registerwert. Rückgabe: alter Wert.

```
function read-modify-write(r : Register, f : Funktion)  
  returns : Wert  
    temp := r;  
    r := f(temp);  
    return(temp);  
end function
```

Beachte:

- *read*, *write*, *test-and-set* und *fetch-and-add* können alle mittels *read-modify-write* ausgedrückt werden
- Funktion *f* muss Seiteneffekt-frei sein.
Bsp.: *f* eine atomare Operation wie *fetch-and-increment*
⇒ unklar was atomar ausgeführt wird? Deswegen: Verboten!

Read-Modify-Write-Objekt

Ein gemeinsames Speicherregister mit

- *read-modify-write*-Operation
- *write*-Operation
- *read*-Operation

compare-and-swap

erwartet (r, old, new): Wenn $r=old$, dann neuer Wert für r und Rückgabe True, sonst Rückgabe False

```
function compare-and-swap( $r$  : Register,  $old$  : Wert,  $new$  : Wert)
  returns : Wert
  if  $r = old$  then
     $r := new$ ;
    return(True);
  else
    return(False);
end function
```

Compare-and-swap-Objekt

Ein gemeinsames Speicherregister mit

- *compare-and-swap*-Operation
- *write*-Operation
- *read*-Operation

sticky-write

- $sticky-write(r, v)$ erwartet ein Register und einen Wert.
- Initialwert von r : undefiniert (geschrieben als \perp).
- Wenn der Wert des Registers r gleich zu v oder \perp , dann Wert wird auf v gesetzt, True als Ergebnis geliefert.
- sonst: False als Ergebnis

Variante

sticky-bit-write-Operation

Spezialfall: r kann nur die Wert 0, 1 und \perp annehmen

Sticky-Bit

Ein gemeinsames Speicherregister mit

- *read*-Operation
- *sticky-bit-write*-Operation

move

Wert eines gemeinsamen Registers wird in weiteres gemeinsames Register „verschoben“

```
function move( $r_1$  : Register,  $r_2$  : Register)  
    temp :=  $r_2$ ;  
     $r_1$  := temp;  
end function
```

Move-Objekt

Gruppe von gemeinsamen Speicherregister, so dass

- paarweise die *move*-Operation unterstützt wird
- *write*-Operation für alle Register
- *read*-Operation für all Register

shared swap

Zwei gemeinsame Register tauschen den Wert

```
function shared-swap( $r_1$  : Register,  $r_2$  : Register)  
     $temp_1 := r_1$ ;  
     $temp_2 := r_2$ ;  
     $r_1 := temp_2$ ;  
     $r_2 := temp_1$ ;  
end function
```

Shared-swap-Objekt

Gruppe von gemeinsamen Speicherregistern, mit

- paarweise: die *shared-swap*-Operation
- *write*-Operation
- *read*-Operation

Mutual-Exclusion Algorithmen mit test-and-set-Bits

Test-and-set-Bit = Lock (Sperrung)

Ein gemeinsames Speicherregister mit Werten 0 und 1

- *test-and-set*-Operationen setzt Register auf 1
Operation wird auch *lock* genannt
- *reset*-Operation: atomares Schreiben einer 0
Operation wird auch *unlock* genannt

Spin-Lock

Algorithmen benutzen Locks oft so:

```
while test-and-set(lock)=1 do skip;
```

D.h. der Algorithmus "kreiselt" um den Lock.

Daher der Name **Spin-Lock**

Einfacher Algorithmus mit Test-and-Set-Bit

Initial: Test-and-set-Bit x hat Wert 0

Programm des i . Prozesses

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set( $x$ )=0);
(3)  Kritischer Abschnitt
(4)  reset( $x$ );
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x :=

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 1
return von t-a-s: 0

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 1

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 1
return von t-a-s: 1

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 1
return von t-a-s: 1

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 0

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 1
return von t-a-s: 0

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Beispiel: 4 Prozesse

Prozess 1

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 2

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

x := 1

Prozess 3

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Prozess 4

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Eigenschaften des einfachen Algorithmus

- Garantiert **wechselseitigen Ausschluss**:
 - Nur ein Prozess kann atomar x von 0 auf 1 setzen und dabei 0 als Ergebnis erhalten
 - Solange $x=1$ gilt, erhalten alle andern Prozesse das Ergebnis 1
 - Erst nach atomarem *reset* kann der nächste Prozess den Lock setzen
- ist **Deadlock-frei**
 - Wenn mehrere Prozesse im Initialisierungscode keiner im kritischen Abschnitt
 - Dann muss $x=0$ nach endlich vielen Schritten gelten
 - Danach: Einer muss nach endlichen vielen Schritten den Lock setzen
- ist **nicht** Starvation-frei

Eigenschaften des einfachen Algorithmus (2)

- Aus **praktischer** Sicht schlecht:
Bei vielen Prozessen: Lock wird ständig (von 1 auf 1) gesetzt
- Durch das ständige Schreiben müssen Prozessor-Caches auch ständig aktualisiert werden!
- Das führt zu (unnötigem) Kommunikationsaufwand!

Verbesserung mit Test-and-Test-and-Set-Bit

Initial: Test-and-test-and-set-Bit x hat Wert 0

Programm des i . Prozesses

```
loop forever
(1)  restlicher Code
(2)  await (x=0);
(3)  while (test-and-set(x)=1) do
(4)    await (x=0);
(5)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

- Er wird **lesend** gewartet
- Nur wenn Chance zum Setzen des Locks vorhanden:
test-and-set

(Praktische) Verbesserung mit Pause-Operation

Initial: Test-and-test-and-set-Bit x hat Wert 0, delay: lokale Variable, minDelay, maxDelay: Konstanten

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  delay=minDelay;
(3)  repeat
(4)    delay := min(2*delay,maxDelay)
(5)    while (x=1) do pause(delay);
(6)  until (test-and-set(x)=0)
(7)  Kritischer Abschnitt
(8)  reset(x);
end loop
```

- pause(x) wartet x ms
- Wartezeit wird exponentiell erhöht ("exponential backoff")
- Nur erhöhen, wenn Lock frei war, aber nicht geschnappt wurde

Starvation-freier Algorithmus mit Test-and-Set

- Veröffentlicht: R. Alur und G. Taubenfeld 1993
- Starvation-frei
- erfüllt die Eigenschaft n -Fairness ($n =$ Anzahl der Prozesse)

r -Fairness

Ein wartender Prozess hat die Möglichkeit den kritischen Abschnitt zu betreten bevor alle anderen Prozesse **gemeinsam** den kritischen Abschnitt $r + 1$ -mal betreten können.

Beachte:

- r -Fairness impliziert r -bounded waiting
- r -bounded waiting impliziert $((n - 1) \cdot r)$ -Fairness

Starvation-freier Algorithmus mit Test-and-Set (2)

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  waiting[i]:=True;
(3)  key := 1;
(4)  while (waiting[i] and key=1) do
(5)    key := test-and-set(lock);
(6)  waiting[i]:=False;
(7)  Kritischer Abschnitt
(8)  if turn=i
(9)    then lturn := 1 + (turn mod n);
(10)   else lturn := turn
(11)  if waiting[lturn]
(12)    then turn := lturn;
(13)         waiting[lturn] := False;
(14)   else turn := 1+(lturn mod n);
(15)         reset(lock);
end loop
```

Initial:

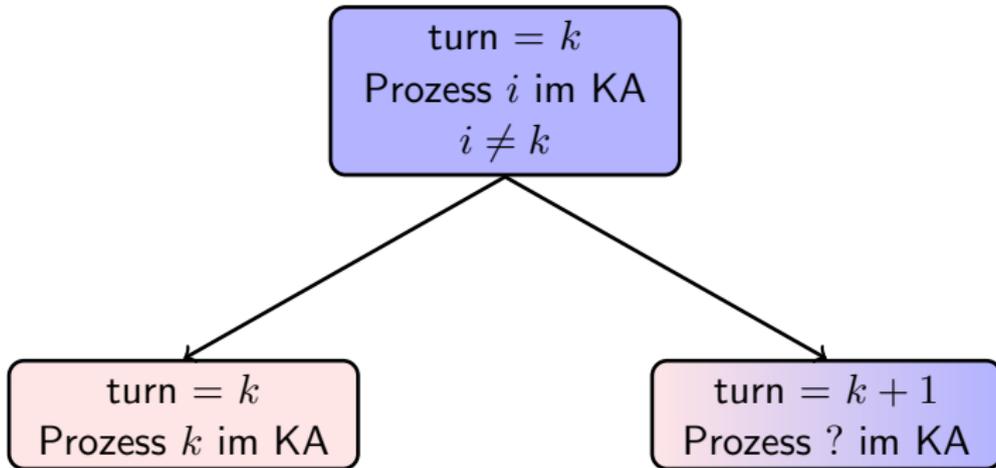
turn: atom. Register, Wert egal
(zwischen 1..n);
Test-and-set-Bit lock: 0;
waiting[i]: False;
lturn, key: lokale Register

Starvation-freier Algorithmus mit Test-and-Set (2)

- Variable `turn` hält Nummer eines Prozesses
- Trick: im Abschlusscode
- Lock wird „übergeben“ an Prozess mit der Nummer `turn` wenn er wartet
- das geschieht ohne *reset* sondern über *waiting*
- (Wenn `turn` aktueller Prozess, dann wird vorher inkrementiert)
- Wenn Prozess `turn` nicht wartet, dann `turn` erhöhen und *reset*
- Nach höchstens `n` mal Abschlusscode insgesamt gilt `turn = j` (nächste Folie)

Beweis der n -Fairness

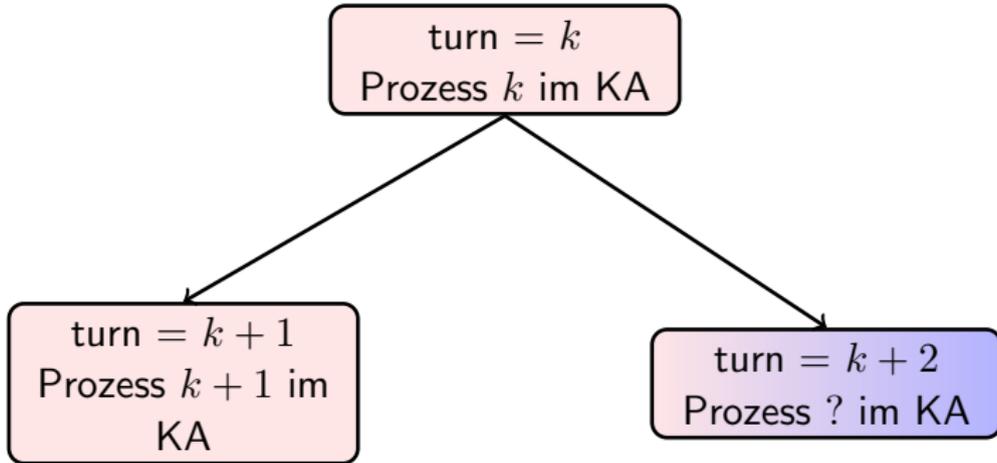
Zwei Fälle: Fall 1:



Rote Knoten = Prozess u im kritischen Abschnitt und $\text{turn} = u$
Blaue Knoten = Prozess u im kritischen Abschnitt und $\text{turn} \neq u$
Rot/Blau = Beides möglich

Beweis der n -Fairness (2)

Zwei Fälle: Fall 2:



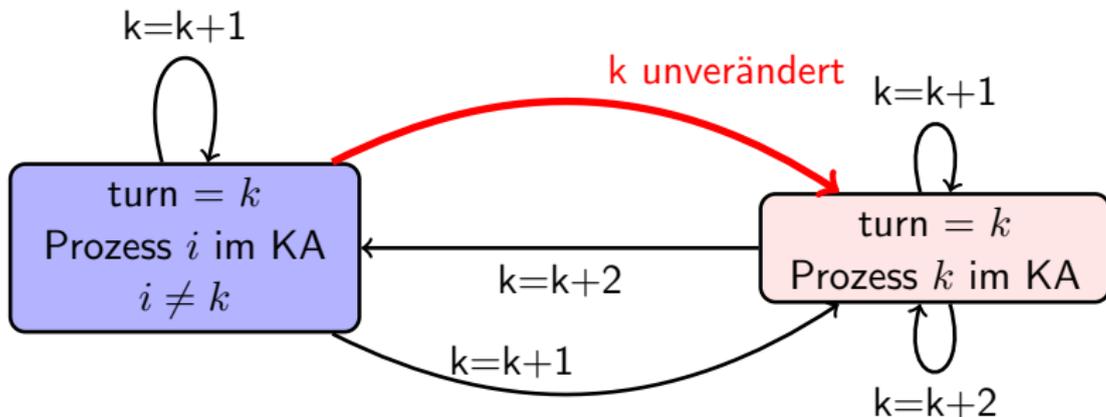
Rote Knoten = Prozess u im kritischen Abschnitt und $\text{turn} = u$

Blaue Knoten = Prozess u im kritischen Abschnitt und $\text{turn} \neq u$

Rot/Blau = Beides möglich

Beweis der n -Fairness (3)

Vereinigung der Fälle:



Roter Pfeil kann erst wieder gewählt werden, wenn turn um **2** erhöht wurde!

Mutual-Exclusion Algorithmus mit RMW-Objekt

Ticket-Algorithmus mit RMW-Objekt

- M.J. Fischer, N.A. Lynch, J.E. Burns, A. Borodin 1989
- Idee ähnlich zum Bakery-Algorithmus („Nummern ziehen“).
- Feste Anzahl an Nummern reicht jedoch aus.

Ticket-Algorithmus mit RMW-Objekt (2)

Initial: (ticket,valid): Read-Modify-Write Objekt für ein Paar von Zahlen im Bereich $1 \dots n$ wobei initial ticket = valid (ticket_i, valid_i) lokales Register

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
    // erhöhe ticket (lese alte Werte):
(2)  (ticketi,validi) := read-modify-write((ticket,valid), inc-fst)
(3)  while ticketi ≠ validi do
(4)      validi := valid;
(5)  Kritischer Abschnitt
(6)  read-modify-write((ticket,valid), inc-snd) // erhöhe valid
end loop
```

Funktionen

```
function inc-fst((a,b))
    return (1+(a mod n), b)
end function
```

```
function inc-snd((a,b))
    return (a, 1+(b mod n))
end function
```

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = ?$
 $v_1 = ?$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = ?$
 $v_2 = ?$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = ?$
 $v_3 = ?$

$t = 1$
 $v = 1$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = ?$
 $v_1 = ?$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = ?$
 $v_2 = ?$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

$t = 2$
 $v = 1$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r-m-w((t, v), inc-fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r-m-w((t, v), inc-snd)$
- $$t_1 = 2$$
- $$v_1 = 1$$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r-m-w((t, v), inc-fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r-m-w((t, v), inc-snd)$
- $$t_2 = ?$$
- $$v_2 = ?$$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r-m-w((t, v), inc-fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r-m-w((t, v), inc-snd)$
- $$t_3 = 1$$
- $$v_3 = 1$$

$$t = 3$$
$$v = 1$$

Beispiel

Prozess 1

- (1) restlicher Code
- (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) Kritischer Abschnitt
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
 $t_1 = 2$
 $v_1 = 1$

Prozess 2

- (1) restlicher Code
- (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) Kritischer Abschnitt
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
 $t_2 = 3$
 $v_2 = 1$

Prozess 3

- (1) restlicher Code
- (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) Kritischer Abschnitt
- (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
 $t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 1$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_1 = 2$$
- $$v_1 = 1$$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_2 = 3$$
- $$v_2 = 1$$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_3 = 1$$
- $$v_3 = 1$$

$$t = 1$$
$$v = 1$$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = 2$
 $v_1 = 1$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = 3$
 $v_2 = 1$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 1$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_1 = 2$$
- $$v_1 = 1$$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_2 = 3$$
- $$v_2 = 1$$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_3 = 1$$
- $$v_3 = 1$$

$$t = 1$$
$$v = 1$$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_1 = 2$$
- $$v_1 = 1$$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_2 = 3$$
- $$v_2 = 1$$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) **Kritischer Abschnitt**
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_3 = 1$$
- $$v_3 = 1$$

$$t = 1$$
$$v = 1$$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = 2$
 $v_1 = 1$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = 3$
 $v_2 = 1$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_1 \neq v_1$ **do**
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_1 = 2$$
- $$v_1 = 1$$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_2 \neq v_2$ **do**
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_2 = 3$$
- $$v_2 = 1$$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_3 \neq v_3$ **do**
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $$t_3 = 1$$
- $$v_3 = 1$$

$$t = 1$$
$$v = 2$$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_1 \neq v_1$ do
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = 2$
 $v_1 = 2$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_2 \neq v_2$ do
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = 3$
 $v_2 = 1$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) while $t_3 \neq v_3$ do
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Beispiel

Prozess 1

- (1) restlicher Code
 - (2) $(t_1, v_1) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_1 \neq v_1$ **do**
 - (4) $v_1 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_1 = 2$
 $v_1 = 2$

Prozess 2

- (1) restlicher Code
 - (2) $(t_2, v_2) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_2 \neq v_2$ **do**
 - (4) $v_2 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_2 = 3$
 $v_2 = 1$

Prozess 3

- (1) restlicher Code
 - (2) $(t_3, v_3) := r\text{-}m\text{-}w((t, v), \text{inc}\text{-}fst)$
 - (3) **while** $t_3 \neq v_3$ **do**
 - (4) $v_3 := v;$
 - (5) Kritischer Abschnitt
 - (6) $r\text{-}m\text{-}w((t, v), \text{inc}\text{-}snd)$
- $t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Beispiel

Prozess 1

- (1) restlicher Code
- (2) $(t_1, v_1) := r-m-w((t, v), inc-fst)$
- (3) while $t_1 \neq v_1$ do
- (4) $v_1 := v;$
- (5) **Kritischer Abschnitt**
- (6) $r-m-w((t, v), inc-snd)$
 $t_1 = 2$
 $v_1 = 2$

Prozess 2

- (1) restlicher Code
- (2) $(t_2, v_2) := r-m-w((t, v), inc-fst)$
- (3) while $t_2 \neq v_2$ do
- (4) $v_2 := v;$
- (5) **Kritischer Abschnitt**
- (6) $r-m-w((t, v), inc-snd)$
 $t_2 = 3$
 $v_2 = 1$

Prozess 3

- (1) restlicher Code
- (2) $(t_3, v_3) := r-m-w((t, v), inc-fst)$
- (3) while $t_3 \neq v_3$ do
- (4) $v_3 := v;$
- (5) **Kritischer Abschnitt**
- (6) $r-m-w((t, v), inc-snd)$
 $t_3 = 1$
 $v_3 = 1$

$t = 1$
 $v = 2$

Theorem

Der Ticket-Algorithmus garantiert wechselseitigen Ausschluss, Starvation-Freiheit und die FIFO-Eigenschaft.

Den Beweis ist einfach (Wir lassen ihn weg).

Eigenschaften des Ticket-Algorithmus (2)

Bemerkungen

- Der Ticket-Algorithmus verwendet ein RMW-Objekt mit n^2 unterschiedlichen Werten
- Bezüglich dieser Anzahl an Werten ist der Algorithmus der beste bekannte Algorithmus unter allen Algorithmen, die wechselseitigen Ausschluss, Deadlock-Freiheit und die starke FIFO-Eigenschaft garantieren.
starke FIFO-Eigenschaft: FIFO gilt auch Initialisierungscode und im Abschlusscode
- Untere Schranke: Jeder Algorithmus für $n \geq 3$ Prozesse benötigt mindestens $\frac{n^2-3n+2}{2}$ verschiedene Zustände des gemeinsamen Speichers.

Mutual-Exclusion Algorithmus mit Warteschlange und Compare-and-Swap-Objekt

Der MCS Queue-basierte Algorithmus

- Veröffentlicht von **Mellor-Crummey** und **Scott** im Jahr 1991
- Dijkstra Preis in Distributed Computing 2006
- Benutzt ein **Swap-Compare-and-Swap Objekt**, d.h. Objekt mit *swap*- und *compare-and-swap*-Operation
- Prozesse reihen sich in eine Warteschlange ein
- Besonderheit: Der Abschlusscode ist **nicht wait-frei**
- Eigenschaften: erfüllt Mutual-Exclusion, Starvation-Freiheit und **FIFO**-Eigenschaft
- Es wird nur gewartet an **lokalen** Registern

Für einen Zeiger p sei

- $*p$ der Indirektions-Operation, d.h. $*p$ liefert das Objekt, auf das der Zeiger zeigt.
- $\&p$ gibt die Speicheradresse eines Objekts (um damit Zeiger auf das Objekt zu erstellen)

Der MCS Queue-basierte Algorithmus (2)

Typen:

- Record Element: zwei Attribute: value:Bool, und next: Zeiger auf ein Element

Gemeinsame Variablen:

- nodes[i]: Feldeintrag: Inhalt ein Record vom Typ Element
- tail: Swap und Compare-and-Swap Objekt vom Typ: Zeiger auf ein Element
Wert am Anfang: Nil

Lokale Variablen:

- mynode: Zeiger auf ein Element, am Anfang auf nodes[i]
- prev, succ: Zeiger auf Elemente

Der MCS Queue-basierte Algorithmus (3)

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  *mynode.next := Nil;
(3)  prev := mynode;
(4)  swap(tail,prev);
(5)  if prev  $\neq$  Nil then
(6)    *mynode.value := 1;
(7)    *prev.next := mynode;
(8)    await *mynode.value = 0;
(9)  Kritischer Abschnitt
(10) if mynode.next = Nil then
(11)   if compare-and-swap(tail,mynode,Nil) = False then
(12)     await *mynode.next  $\neq$  Nil;
(13)     succ := *mynode.next;
(14)     *succ.value := 0;
(15)  else
(16)    succ := *mynode.next;
(17)    *succ.value := 0;
end loop
```

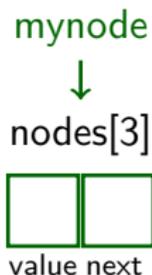
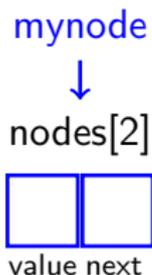
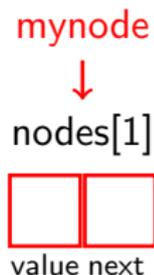
Der MCS Queue-basierte Algorithmus (4)

Idee:

- Prozesse reihen sich in die Warteschlange ein.
- Erster Prozess darf in den kritischen Abschnitt
- Beim Verlassen des kritischen Abschnitts gibt es mehrere Fälle:
 - kam in der Zwischenzeit ein weiterer Prozess, der sich anhängte, dann einfacher Fall: dessen Wert wird auf 0 gesetzt (* `mynode.next` \neq `nil`)
 - kam in der Zwischenzeit kein Prozess in den Initialisierungscode, dann wird `tail` auf `nil` gesetzt (Fall `compare-and-swap` liefert `True`)
 - kam in der Zwischenzeit ein Prozess, der aber noch nicht fertig ist mit anhängen, dann wird gewartet bis er sich angehängt hat (Fall `compare-and-swap` liefert `false`)

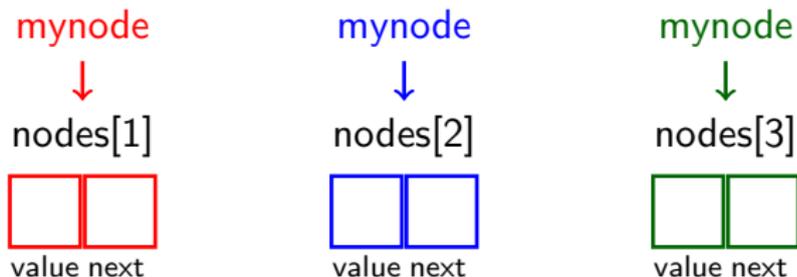
Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt
tail \longrightarrow Nil



Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt
tail \longrightarrow Nil

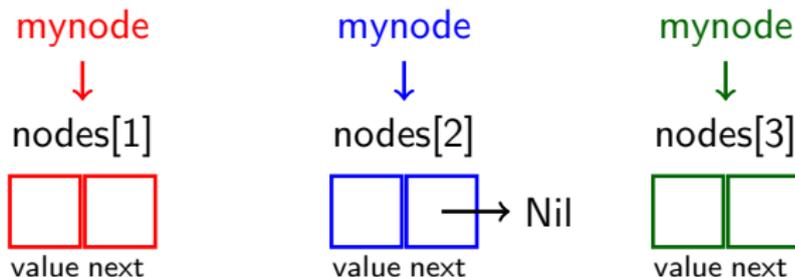


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt
tail \longrightarrow Nil

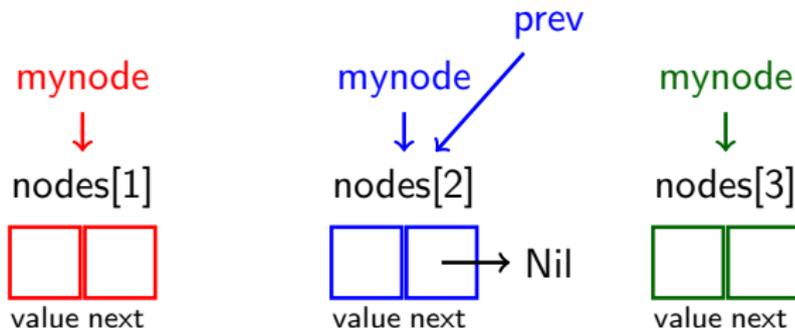


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt
tail \longrightarrow Nil

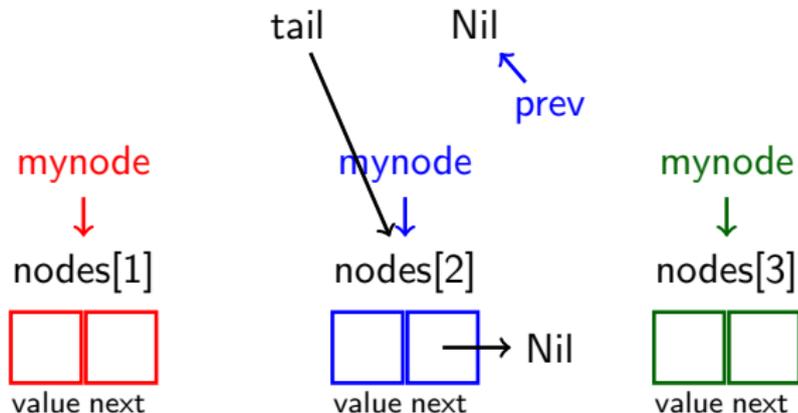


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt

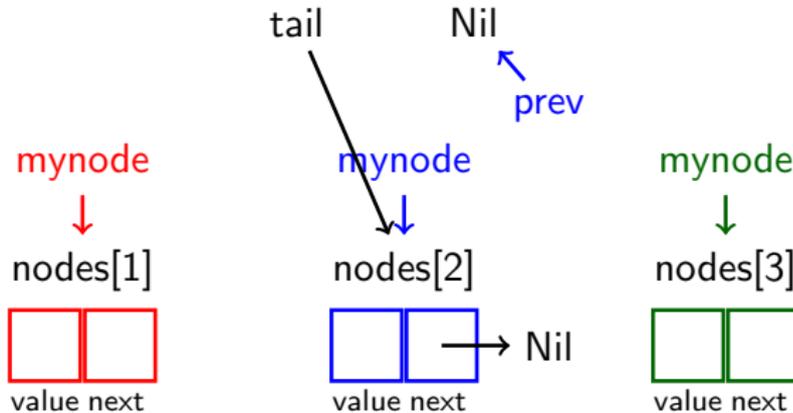


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt



Prozess 2:

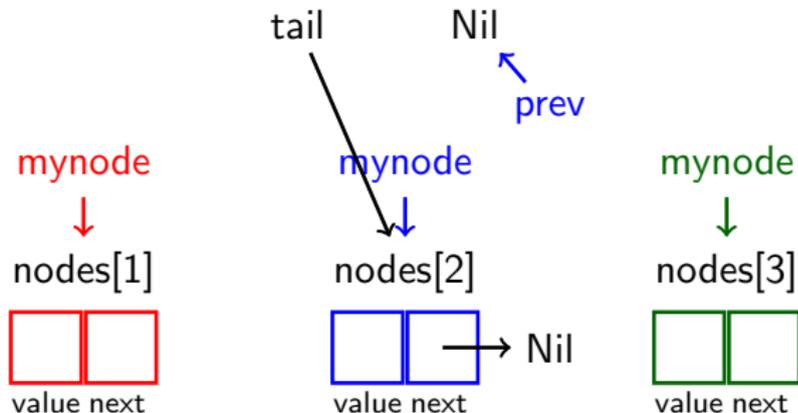
(5) if $prev \neq Nil$ then

...

(9) Kritischer Abschnitt

Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt



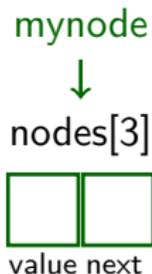
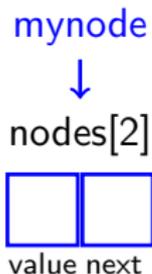
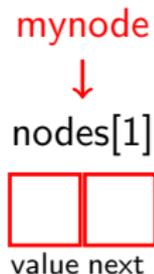
Prozess 2:

(10) if mynode.next = Nil then

(11) if *compare-and-swap*(tail, mynode, Nil) = False then

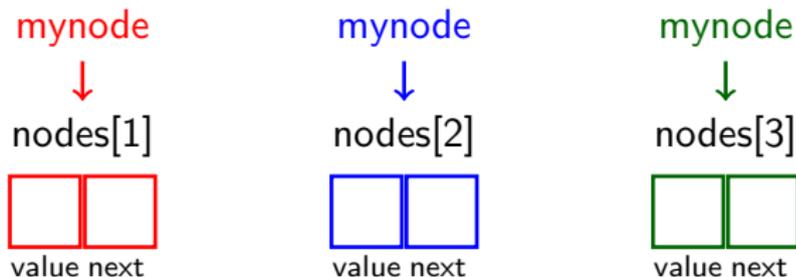
Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt
tail \longrightarrow Nil



Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt
tail \longrightarrow Nil

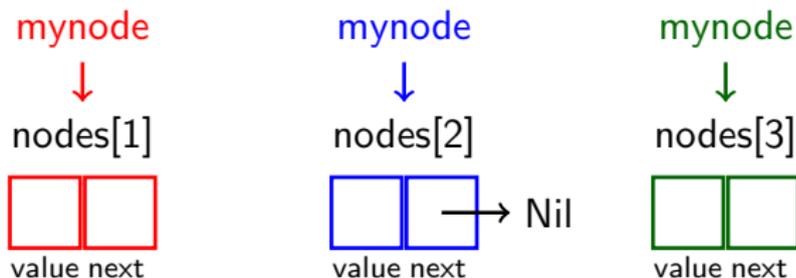


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt
tail \longrightarrow Nil

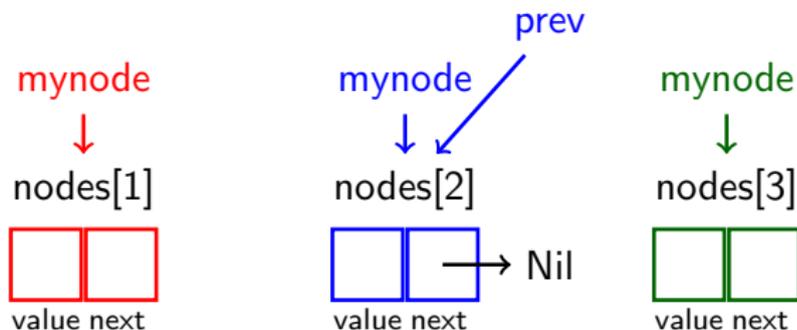


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt
tail \longrightarrow Nil

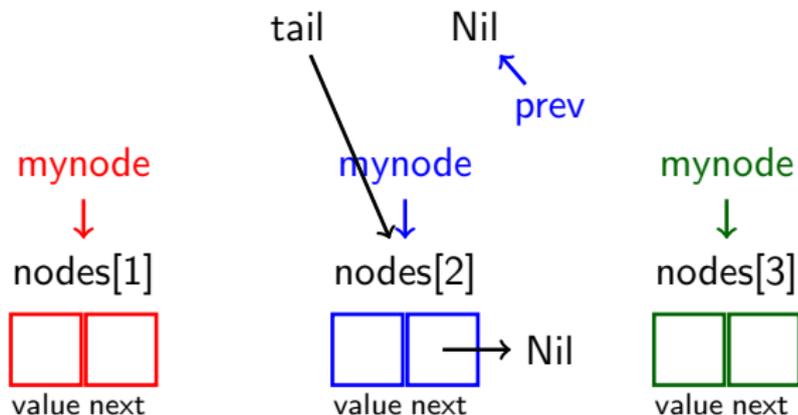


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

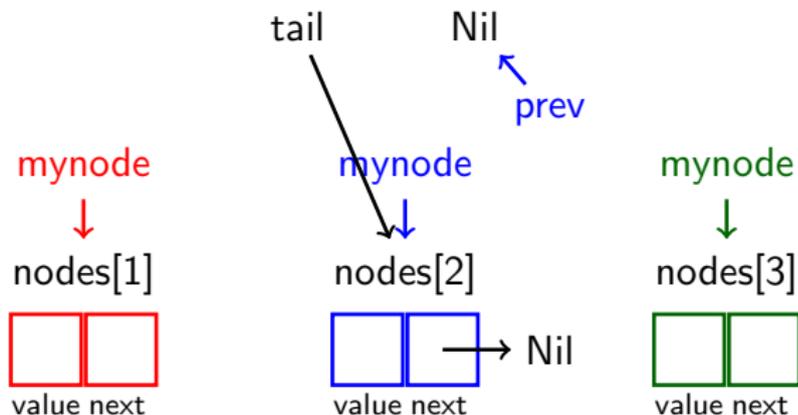


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt



Prozess 2:

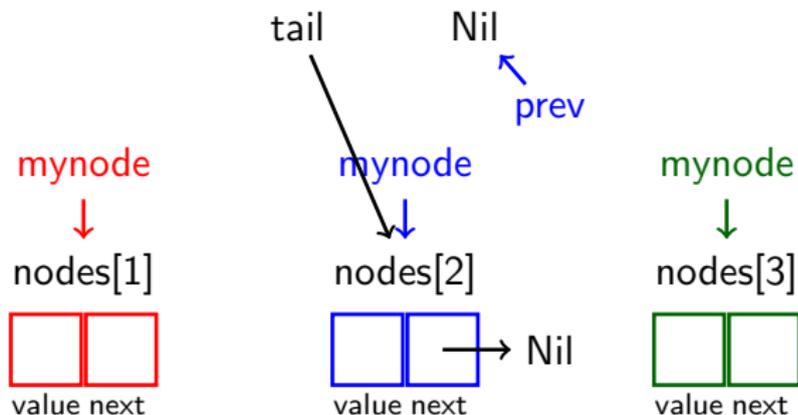
(5) if $prev \neq Nil$ then

...

(9) Kritischer Abschnitt

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

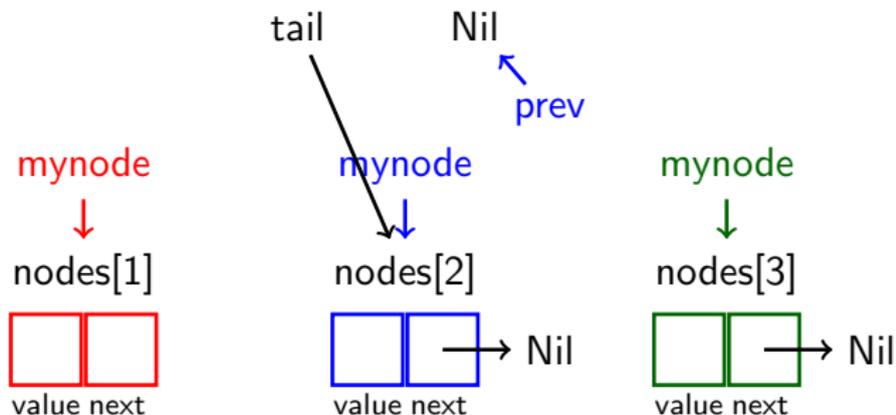


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

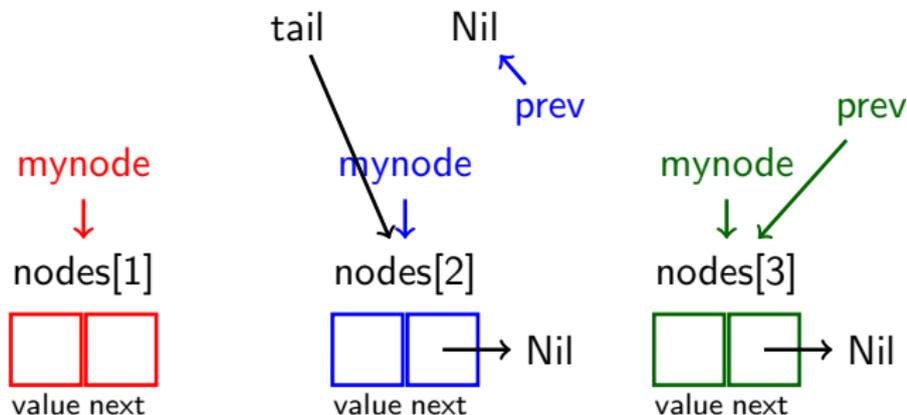


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

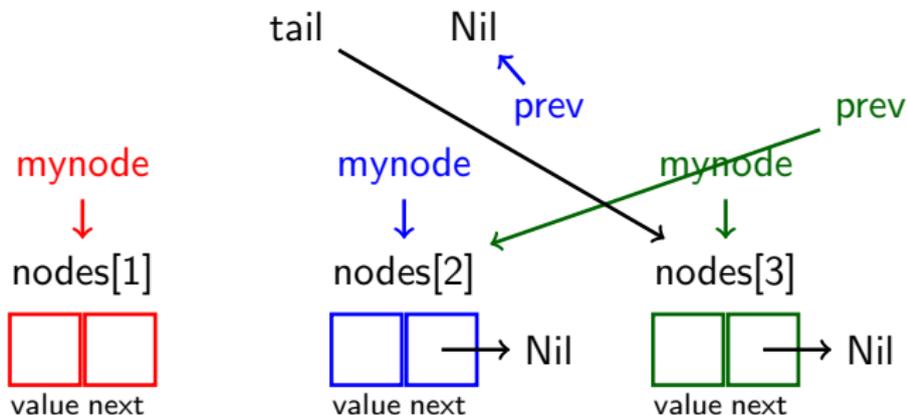


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

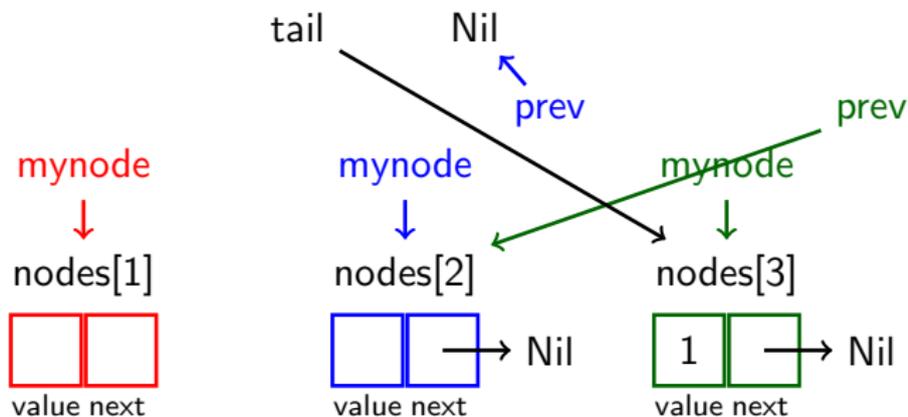


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

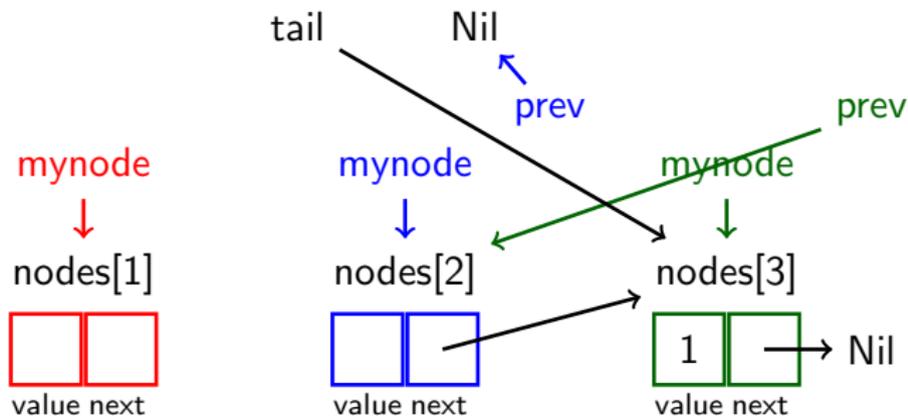


Prozess 3:

- (5) `if prev \neq Nil then`
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

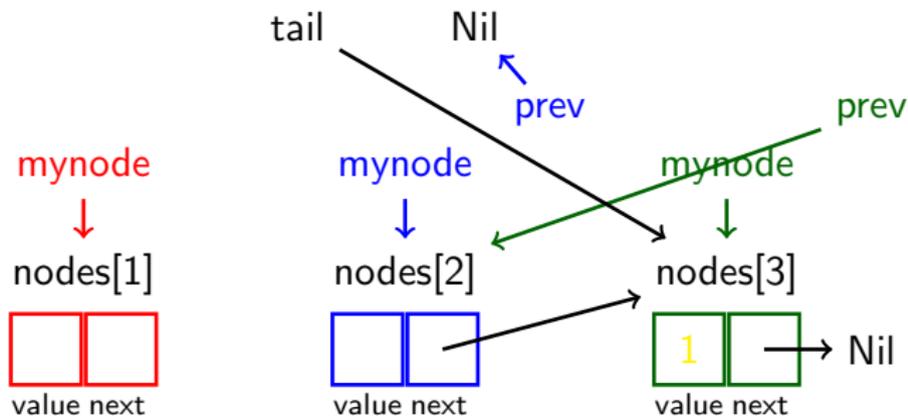


Prozess 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

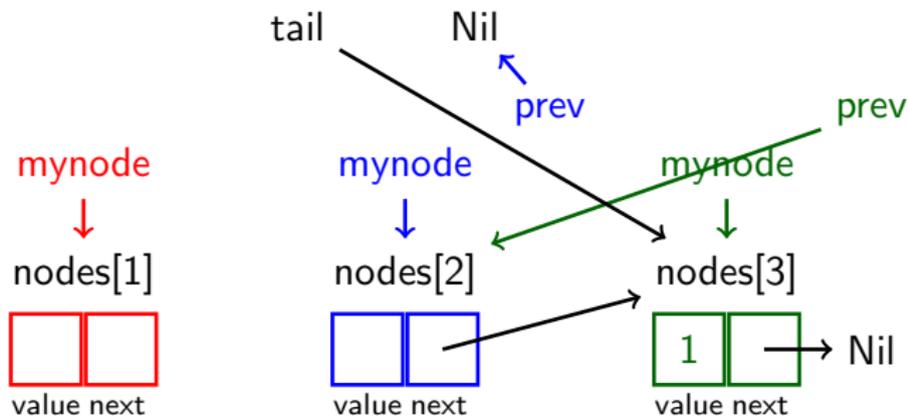


Prozess 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

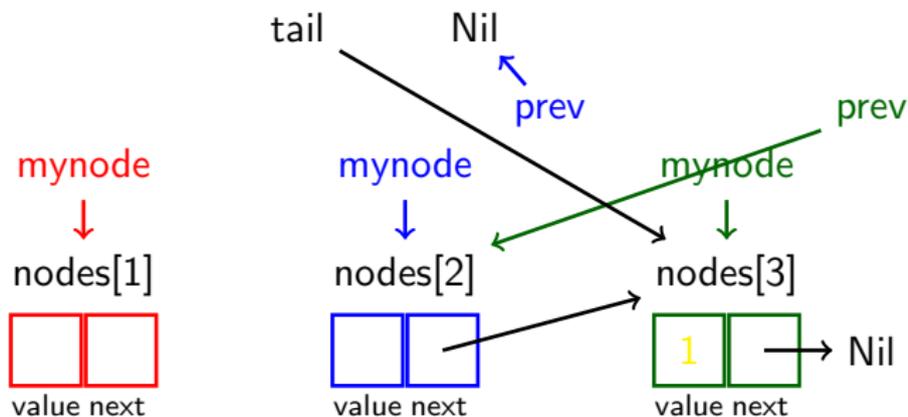


Prozess 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

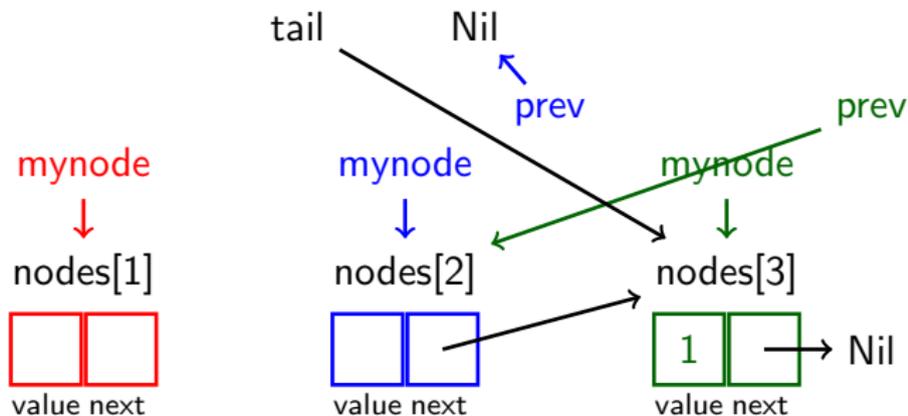


Prozess 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

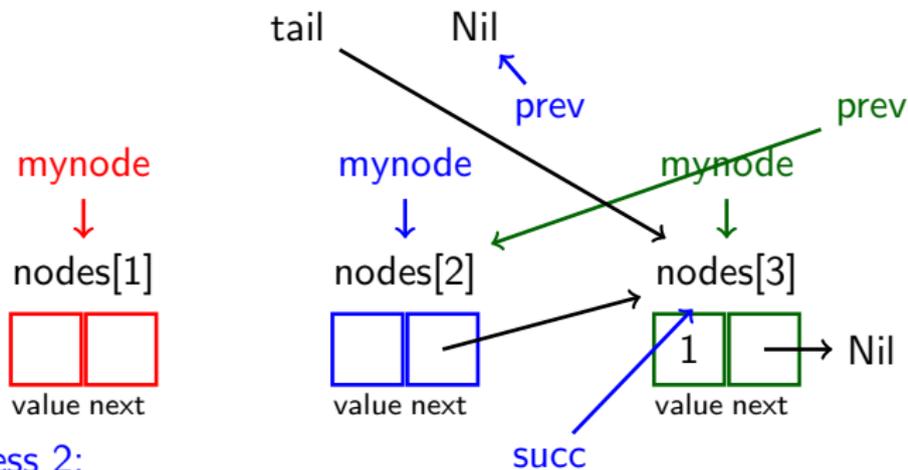


Prozess 3:

- (5) if `prev ≠ Nil` then
- (6) `*mynode.value := 1;`
- (7) `*prev.next := mynode;`
- (8) `await *mynode.value = 0;`

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt

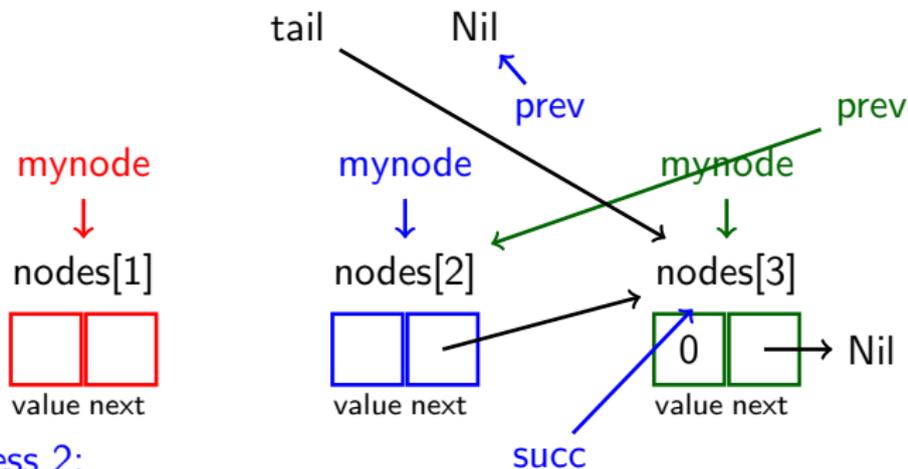


Prozess 2:

```
(10) if mynode.next = Nil then
(15) else
(16)   succ := *mynode.next;
(17)   *succ.value := 0;
```

Der MCS Queue-basierte Algorithmus: Beispiel (2)

Prozess 2 im KA, Prozess 3 will in den kritischen Abschnitt



Prozess 2:

```
(10) if mynode.next = Nil then  
(15) else  
(16)   succ := *mynode.next;  
(17)   *succ.value := 0;
```

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor
3 den next-Zeiger umbiegt

tail \longrightarrow Nil

mynode



nodes[1]



value next

mynode



nodes[2]



value next

mynode



nodes[3]

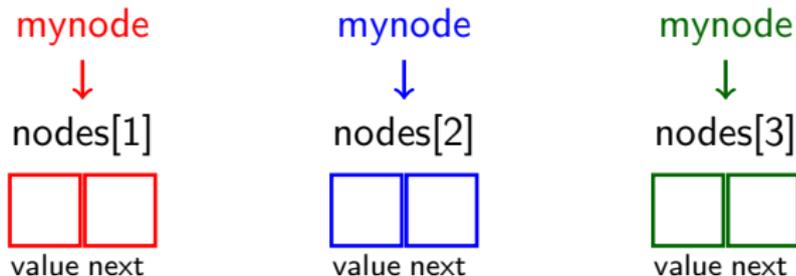


value next

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor
3 den next-Zeiger umbiegt

tail \longrightarrow Nil



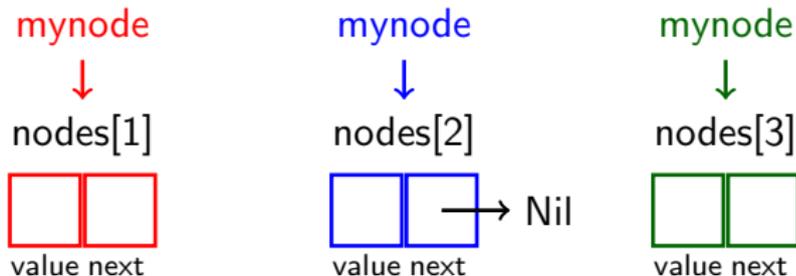
Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor
3 den next-Zeiger umbiegt

tail \longrightarrow Nil



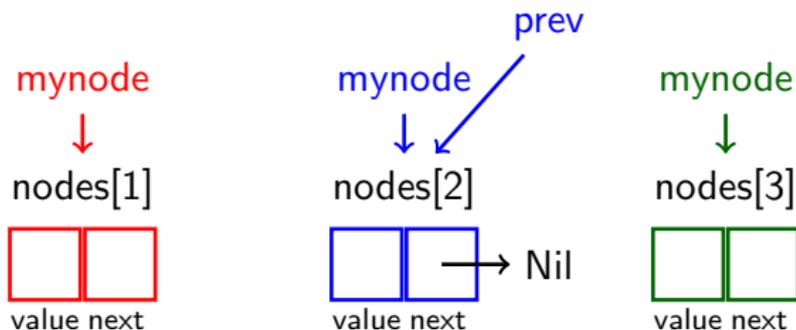
Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

tail \longrightarrow Nil

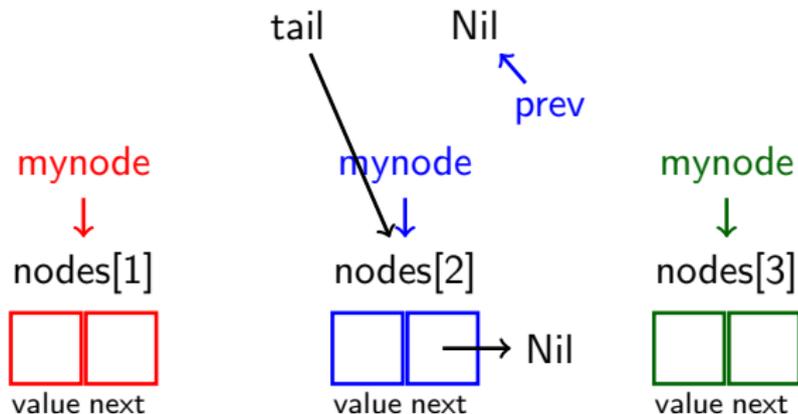


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

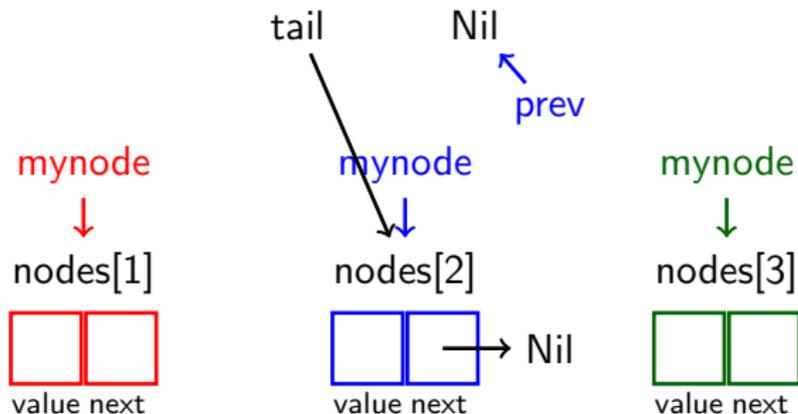


Prozess 2:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt



Prozess 2:

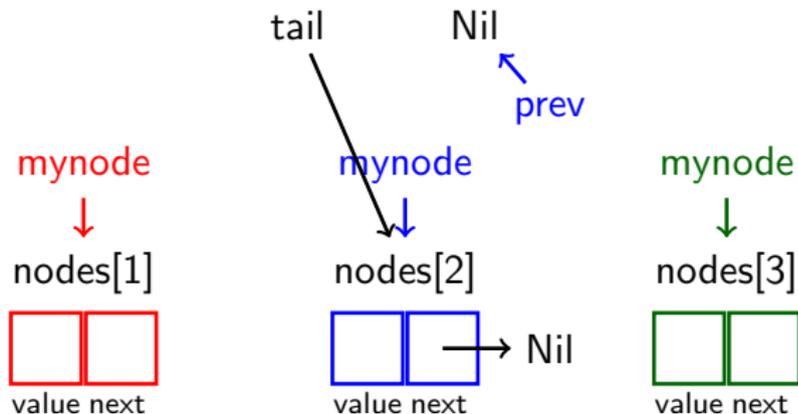
(5) if $prev \neq Nil$ then

...

(9) Kritischer Abschnitt

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

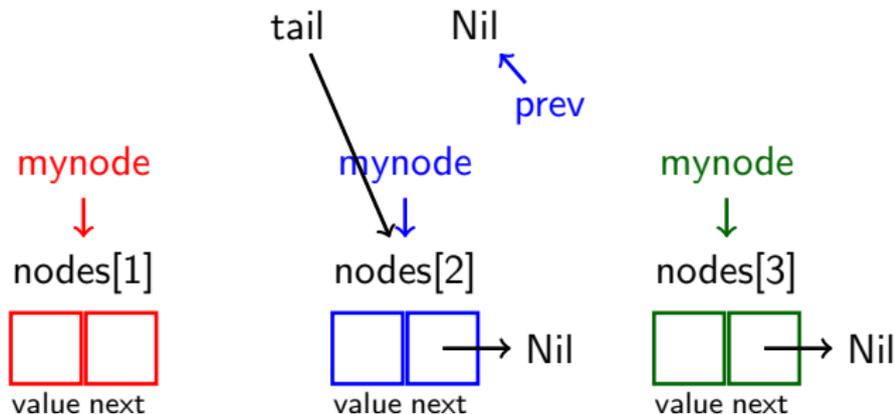


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

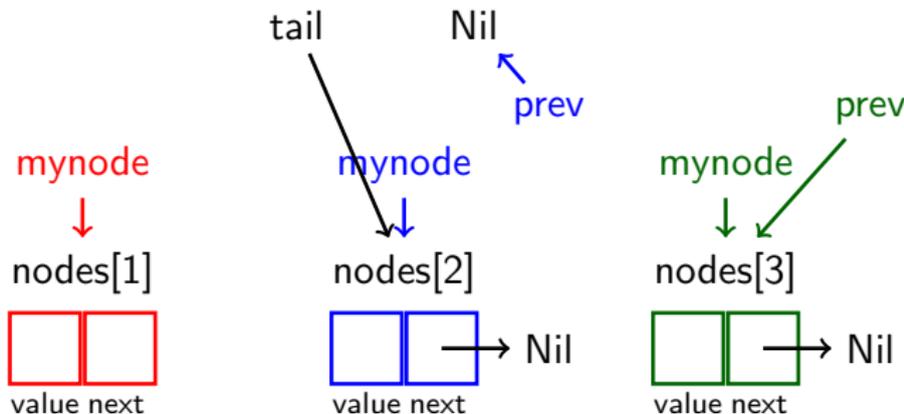


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

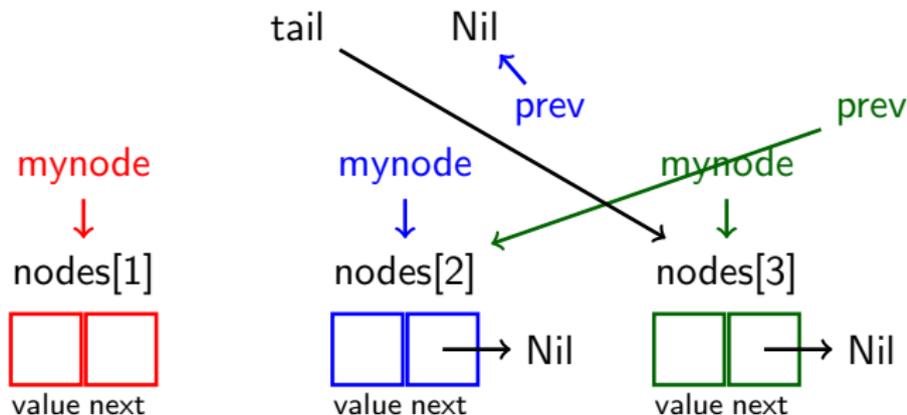


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

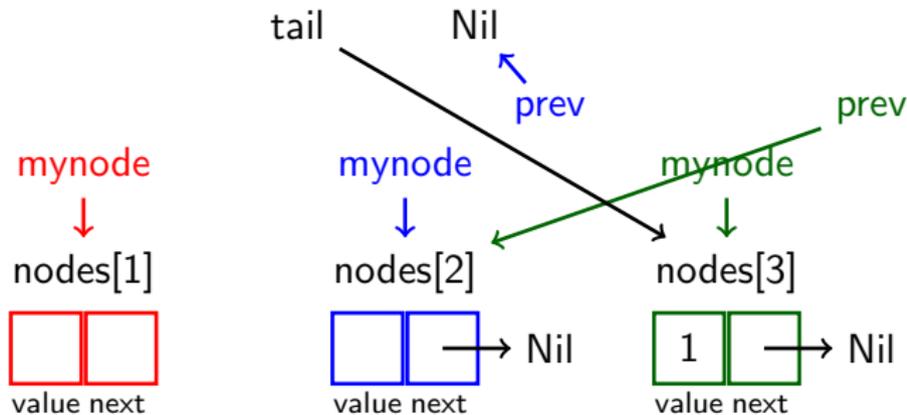


Prozess 3:

- (2) `*mynode.next := Nil;`
- (3) `prev := mynode;`
- (4) `swap(tail,prev);`

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

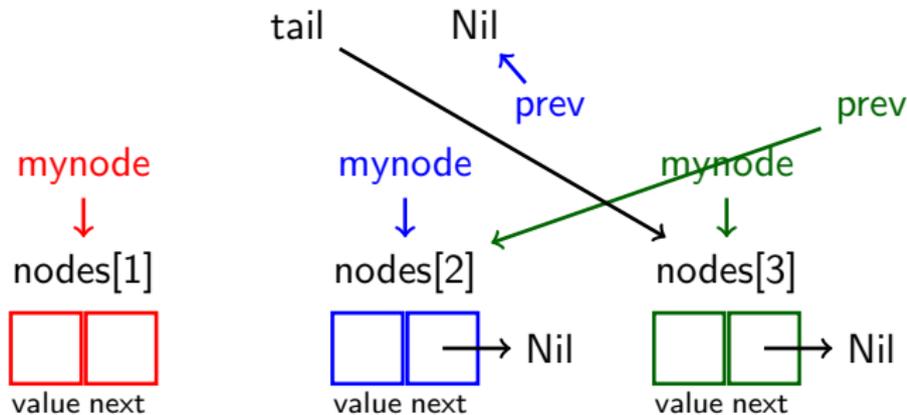


Prozess 3:

- (5) if $prev \neq Nil$ then
- (6) $*mynode.value := 1;$

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt

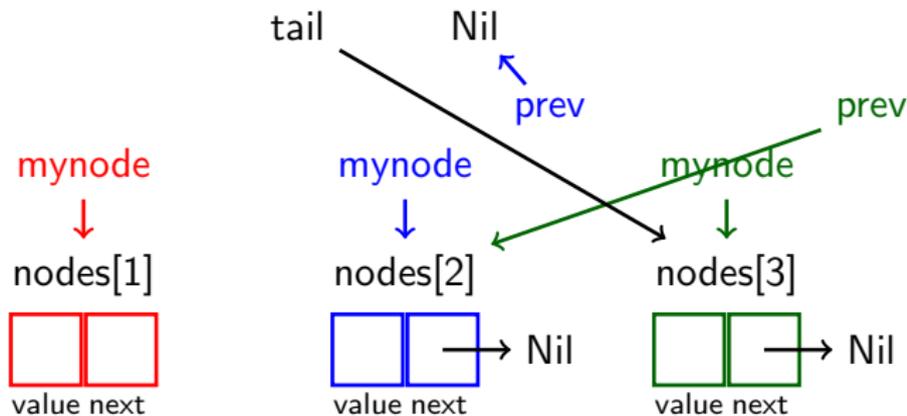


Prozess 3:

- (5) if $prev \neq Nil$ then
- (6) *mynode.value := 1;

Der MCS Queue-basierte Algorithmus: Beispiel (3)

Prozess 2 im KA, Prozess 3 will in KA, aber Prozess 2 fertig, bevor 3 den next-Zeiger umbiegt



Prozess 2:

- ```
(10) if mynode.next = Nil then
(11) if compare-and-swap(tail,mynode,Nil) = False then
(12) await *mynode.next ≠ Nil;
(13) succ := *mynode.next;
(14) *succ.value := 0;
```

# Konsensus und die Herlihy-Hierarchie

- Jede Menge Nebenl. Objekte: Atomares Register, RMW-Objekt, Test-and-set-Objekt, usw.
- Welche Objekte sind „besser“ als die anderen?
- Was bedeutet „besser“?

Intuitiv klar:

- Atomares Register ist eher schwach (z.B. da Mutual-Exklusion schwer)
- RMW-Objekt ziemlich stark, kann viele andere Objekte implementieren

# Herlihy's Modell

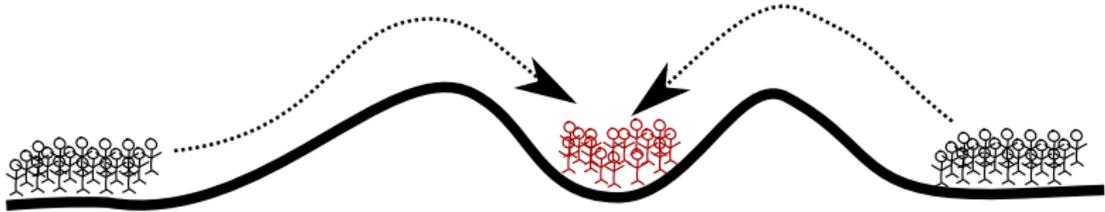
---

- Maurice Herlihy hat dies 1991 untersucht / formalisiert
- aber: Anderes Modell!
- Modell: Prozesse können jederzeit abstürzen
- Modellierung: Prozess bleibt vor einem Kommando hängen

Beachte: Alle gesehenen Mutual-Exklusion Algorithmen vertragen keine Abstürze!

# Ein Beispiel für abstürzende Prozesse

## Das Zwei-Generäle-Problem (auch Coordinated-Attack)



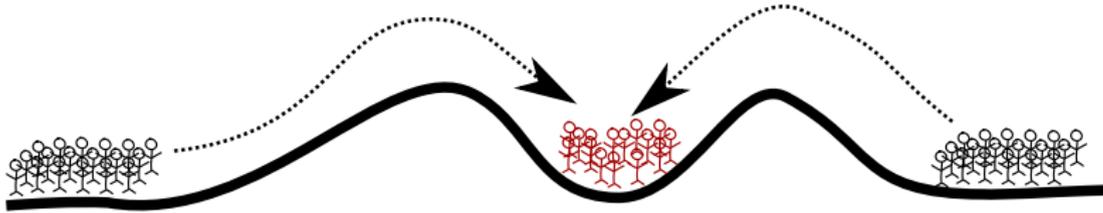
Fakten:

- Angriff nur erfolgreich, wenn beide Divisionen **zeitgleich** angreifen
- Kommunikation nur über **Boten**
- **Boten** müssen durch das Feindesland und kommen daher **nicht immer** an

**Problem:** Wie sprechen sich die Generäle ab?

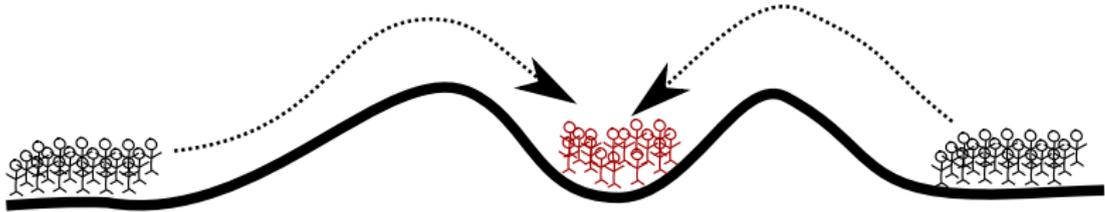
# Zwei Generäle Problem

---



- Rechter General schickt Boten los: "Angriff um 12:00 mittags"

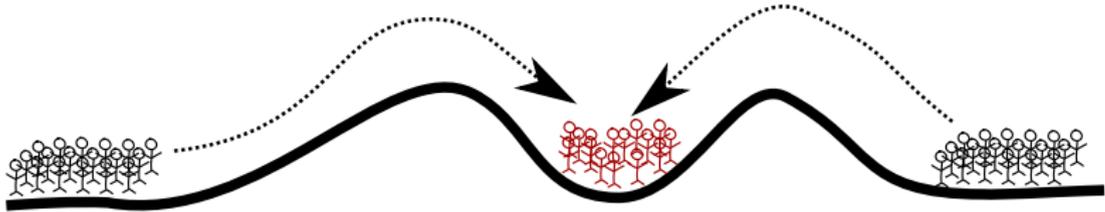
# Zwei Generäle Problem



- Rechter General schickt Boten los: “Angriff um 12:00 mittags”
- Rechter General kann nicht sicher sein, dass der Bote ankommt  
⇒ wartet auf Bestätigung vom linken General

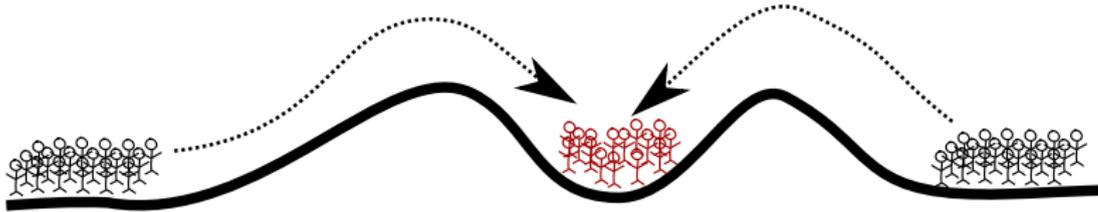
# Zwei Generäle Problem

---



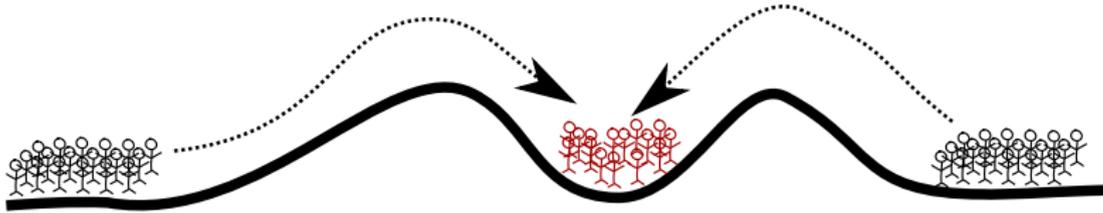
- Rechter General schickt Boten los: “Angriff um 12:00 mittags”
- Rechter General kann nicht sicher sein, dass der Bote ankommt  
⇒ wartet auf Bestätigung vom linken General
- Linker General schickt Boten los: “Nachricht erhalten”

# Zwei Generäle Problem



- Rechter General schickt Boten los: “Angriff um 12:00 mittags”
- Rechter General kann nicht sicher sein, dass der Bote ankommt  
⇒ wartet auf Bestätigung vom linken General
- Linker General schickt Boten los: “Nachricht erhalten”
- Linker General kann nicht sicher sein, dass der Bote ankommt  
⇒ wartet auf Bestätigung vom rechten General

# Zwei Generäle Problem

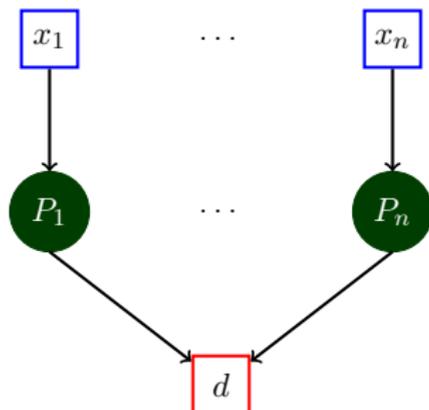


- Rechter General schickt Boten los: “Angriff um 12:00 mittags”
- Rechter General kann nicht sicher sein, dass der Bote ankommt  
⇒ wartet auf Bestätigung vom linken General
- Linker General schickt Boten los: “Nachricht erhalten”
- Linker General kann nicht sicher sein, dass der Bote ankommt  
⇒ wartet auf Bestätigung vom rechten General
- Rechter General schickt Boten los: “Nachricht erhalten”
- usw.

Tatsächlich: Problem ist unlösbar in diesem Modell!

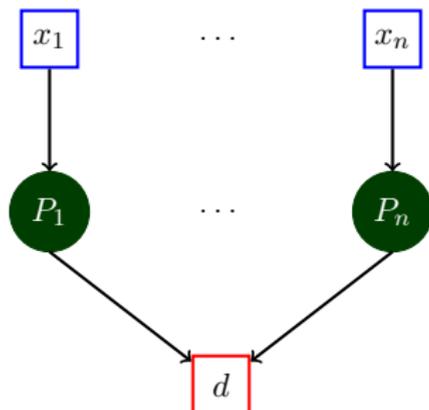
# Das Konsensus Problem

- $n$  Prozesse, die auch abstürzen können
- Prozess  $i$  erhält einen **Eingabewert**  $x_i \in \{0, 1\}$
- Programmier die Prozesse, so dass alle (nicht-abstürzenden) Prozesse sich für einen gemeinsamen **Entscheidungswert**  $d \in \{0, 1\}$  entscheiden
- **Übereinstimmung**: Alle nicht-abgestürzten Prozesse entscheiden sich für den gleichen Wert  $d$ .
- **Gültigkeit**:  $d \in \{x_1, \dots, x_n\}$ , d.h.  $d$  ist einer der Eingabewerte.



# Das Konsensus Problem

- $n$  Prozesse, die auch abstürzen können
- Prozess  $i$  erhält einen **Eingabewert**  $x_i \in \{0, 1\}$
- Programmier die Prozesse, so dass alle (nicht-abstürzenden) Prozesse sich für einen gemeinsamen **Entscheidungswert**  $d \in \{0, 1\}$  entscheiden



- **Übereinstimmung:** Alle nicht-abgestürzten Prozesse entscheiden sich für den gleichen Wert  $d$ .
- **Gültigkeit:**  $d \in \{x_1, \dots, x_n\}$ , d.h.  $d$  ist einer der Eingabewerte.

Beachte: Übereinstimmung  $\Rightarrow$  Algorithmus ist wait-free:

Jeder nicht-abstürzende Prozess entscheidet sich nach endlich vielen Schritten

## Das Konsensus Problem (2)

---

Bei nicht-abstürzenden Prozessen wäre das Problem einfach:

- Alle Prozesse teilen sich (über gemeinsamen Speicher) untereinander alle ihre Eingabewerte mit
- anschließend berechnet jeder Prozess  $d = f(x_1, \dots, x_n) = d$
- wobei  $f$  eine feste Funktion ist, die alle Prozesse kennen, z.B.  $f(x_1, \dots, x_n) = x_1$ .

# Lösung mit dreiwertigem RMW-Objekt

---

Objekte und Initialisierung:

$x$ : RMW-Objekt mit den möglichen Werten  $\perp, 0, 1$ , initial  $\perp$

$x_i$ : Eingabewert von Prozess  $i$

$d_i$ : Entscheidungswert, den Prozess  $i$  trifft.

Programm des  $i$ . Prozesses

- (1)  $d_i := \text{read-modify-write}(x, f_i);$
- (2) **if**  $d_i = \perp$  **then**  
     $d_i := x_i;$

Funktion  $f_i$  des  $i$ . Prozesses

```
function $f_i(v)$
 if $v = \perp$ then return x_i
 else return v
end function
```

# Lösung mit dreiwertigem RMW-Objekt

Objekte und Initialisierung:

$x$ : RMW-Objekt mit den möglichen Werten  $\perp, 0, 1$ , initial  $\perp$

$x_i$ : Eingabewert von Prozess  $i$

$d_i$ : Entscheidungswert, den Prozess  $i$  trifft.

## Programm des $i$ . Prozesses

- (1)  $d_i := \text{read-modify-write}(x, f_i);$
- (2) **if**  $d_i = \perp$  **then**  
     $d_i := x_i;$

## Funktion $f_i$ des $i$ . Prozesses

```
function $f_i(v)$
 if $v = \perp$ then return x_i
 else return v
end function
```

erster Prozess setzt sein  $x_i$  als neuen Wert,  
alle anderen nicht-abstürzenden Prozesse lesen diesen Wert  
⇒ alle  $d_i$ -Werte identisch

# Lösung des Konsensusproblem

---

Mit dreiwertigem RMW-Objekt:

- Der Algorithmus ist eine wait-freie Lösung
- für beliebig viele Prozesse

Resultat für **atomare Register mit *read* und *write***:

Selbst für 2 Prozesse und nur einen Absturz nicht lösbar:

## Theorem

Es gibt keinen Konsensus-Algorithmus für atomares *read* und *write*, der einen Absturz tolerieren kann.

**Folgerung:** Atomare Register können kein RMW-Objekt simulieren

- Statt direkt mit Abstürzen zu argumentieren zeigen wir:  
Ein wait-freier Konsensusalgorithmus für atomare Register und 2 Prozesse existiert nicht
- Wir verwenden Berechnungsbäume, um die Ausführung der Prozesse  $P_1$  und  $P_2$  zu betrachten
- Knoten: Zustände der Prozesse und des Speichers
- Kanten: Wesentliche Schritte, d.h. Lese- oder Schreiboperation auf gem. Speicher
- Kante nach links = Schritt von  $P_1$
- Kante nach rechts = Schritt von  $P_2$

## Beweisskizze (2)

---

- Blätter sind mit Entscheidungswert  $\in \{0, 1\}$  markiert.
- Innere Knoten mit den noch möglichen Entscheidungswerten.
- Bivalenter Knoten: Markierung 0 und 1
- Univalenter Knoten: Markierung 0 (0-valent) oder Markierung 1 (1-valent)
- Da Algorithms wait-frei, ist der Baum endlich!

## Beweisskizze (3)

---

Aussage: Jeder 2-Prozess Konsensusalgorithmus hat einen bivalenten Anfangszustand:

Beweis:

- Nehme an  $P_1$  hat Eingabewert  $x_1 = 0$  und  $P_2$  hat Eingabewert  $x_2 = 1$ .
- Wenn nur  $P_1$  Schritte macht (linkester Pfad im Baum), dann muss  $P_1$  als Entscheidungswert 0 berechnen
- Wenn nur  $P_2$  Schritte macht, dann muss als Entscheidungswert 1 berechnet werden.  $\square$

## Beweisskizze (4)

---

Ein Knoten ist **kritisch**, wenn er bivalent ist und jeder Nachfolger ist univalent.

Aussage: Jeder Berechnungsbaum zu einem Konsensusalgorithmus hat einen kritischen Knoten:

Beweis:

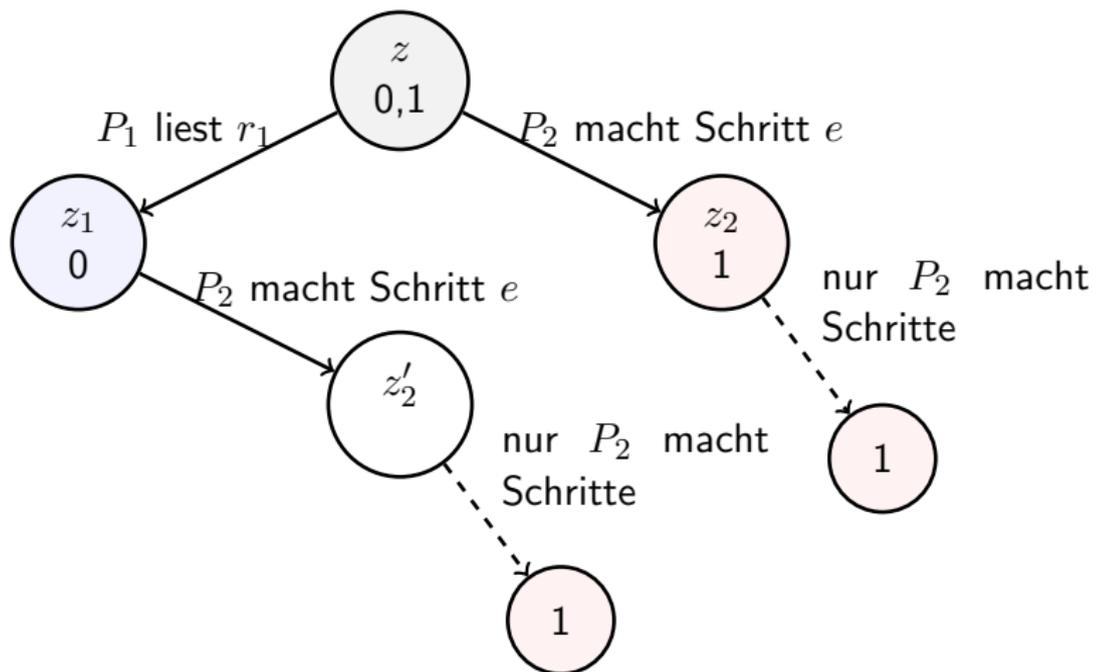
- Angenommen das gilt nicht.
- Starte mit einem bivalenten Anfangszustand.
- Solange es Schritte gibt, die zu einem bivalenten Zustand führen, führe diese Schritte aus.
- Wenn das unendlich lange möglich ist, dann ist der Algorithmus nicht wait-free. Widerspruch.

## Beweisskizze (5)

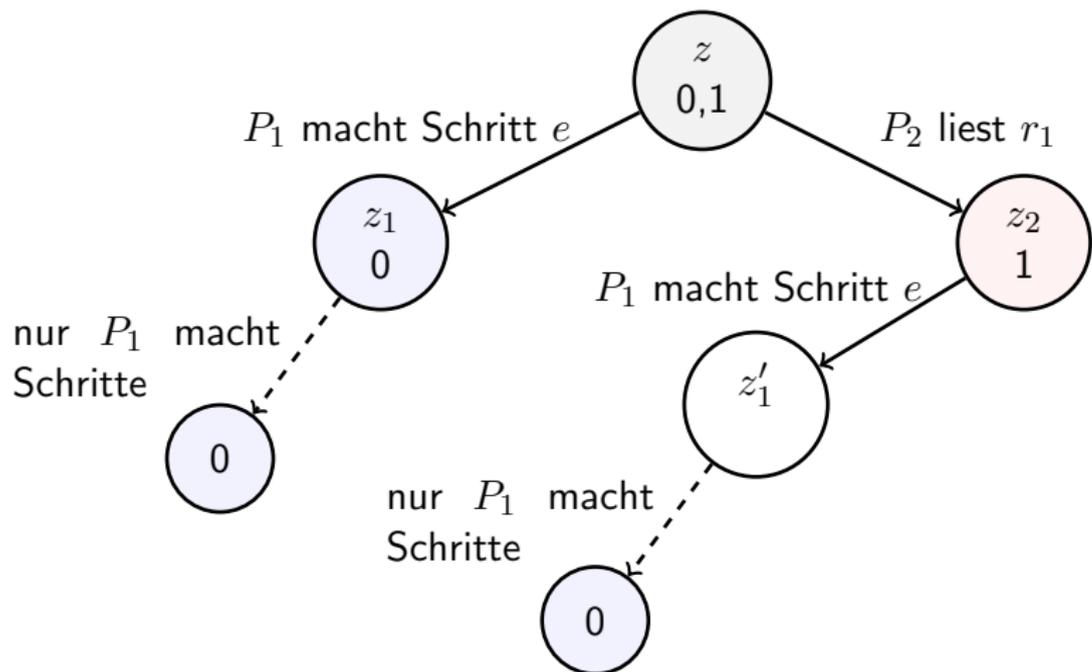
---

- Sei nun Konsensusalgorithmus für 2 Prozesse gegeben.
- Gehe zu kritischem bivalenten Zustand  $z$
- O.b.d.A. linkes Kind ist 0-valent, rechtes Kind ist 1-valent
- Untersuche alle Fälle.

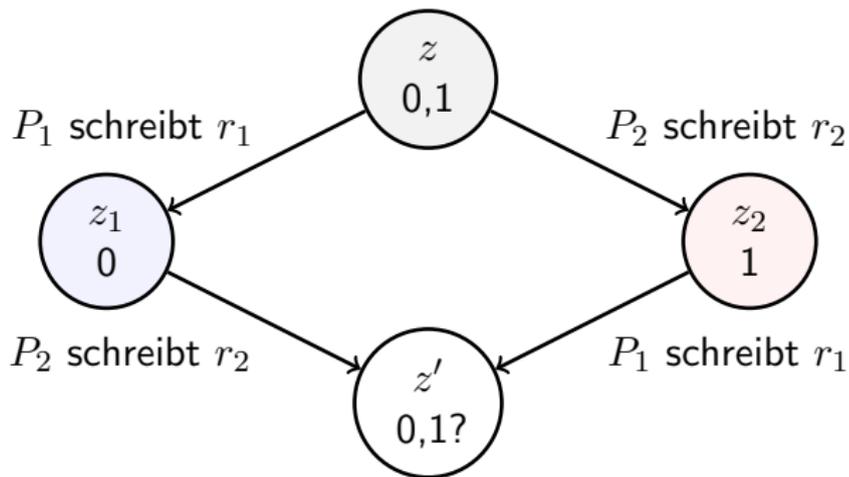
## Beweisskizze (6)



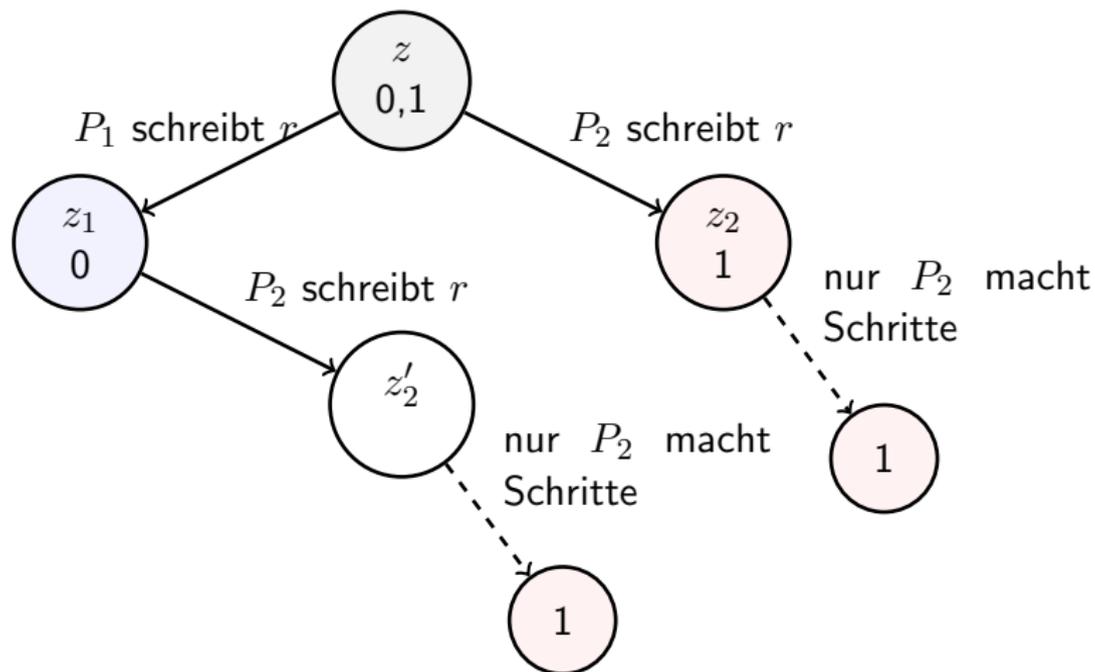
## Beweisskizze (7)



## Beweisskizze (8)



## Beweisskizze (9)



## Definition

Für ein nebenläufiges Objekt vom Typ  $o$  ist die **Konsensus-Zahl**  $CN(o)$  die größte Zahl an Prozessen  $n$  für die man das Konsensus-Problem für  $n$  Prozesse lösen kann, indem man beliebig viele Objekte vom Typ  $o$  und beliebig viele atomare Register (mit *read* und *write*) verwendet. Ist die Anzahl unbeschränkt, so sei  $CN(o) = \infty$ .

# Konsensus-Zahl Hierarchie

| $\mathbb{CN}(o)$   | Objekt $o$                                                                                                |
|--------------------|-----------------------------------------------------------------------------------------------------------|
| 1                  | atomares Register mit <i>read</i> und <i>write</i>                                                        |
| 2                  | test-and-set Objekt, fetch-and-increment Objekt, fetch-and-add Objekt, swap-Objekt, read-modify-write Bit |
| $\Theta(\sqrt{m})$ | swap <sup><math>m</math></sup> -Objekt                                                                    |
| $2m - 2$           | $m$ -Register mit $m$ -facher Zuweisung ( $m > 1$ )                                                       |
| $\infty$           | (drei-wertiges) RMW-Objekt, Compare-and-swap-Objekt, Sticky-Bit                                           |

## Theorem

Wenn  $o_1$  und  $o_2$  zwei Objekte sind mit  $\mathbb{CN}(o_1) < \mathbb{CN}(o_2)$ , dann gilt für ein System mit  $\mathbb{CN}(o_2)$  Prozessen:

- Es gibt keine wait-freie Implementierung von Objekt  $o_2$  ausschließlich mit Objekten vom Typ  $o_1$  und atomaren Registern.
- Es gibt eine wait-freie Implementierung von Objekt  $o_1$  ausschließlich mit Objekten vom Typ  $o_2$  und atomaren Registern.

## Theorem

Ein Objekt  $o$  mit  $\mathbb{CN}(o) \geq n$  ist **universell** in einem System mit  $n$  Prozessen. D.h. jedes wait-freie Objekt (mit einer sequentiellen Beschreibung) kann durch Objekte vom Typ  $o$  und atomaren Registern in einem System mit  $n$  Prozessen implementiert werden.

- Universelle Objekte können viele andere Objekte implementieren (unabhängig vom Konsensus-Problem)
- Objekte mit Konsensus-Zahl  $\geq n$  "reichen aus".