

## Stärkere Speicherprimitive

Prof. Dr. David Sabel

LFE Theoretische Informatik



Letzte Änderung der Folien: 2. November 2020

## Bisher

- Nur Lesen und Schreiben sind atomare (unteilbare) Operationen
- atomare Leseoperation: `if x = 10 then ...`  
Meinte eigentlich `if read(x) = 10 then ...`
- atomare Schreiboperation: `x := 10`  
Meint eigentlich `write(x, 10)`

## Übersicht: Stärkere Speicheroperationen

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

## Jetzt

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

Funktionsname      Parameter mit Typen      Rückgabetyt

```
function xyz(p1 : Typ1, ..., pn : Typn...) returns : Typ
    "atomare Ausführung des Rumpfs"
end function
```

## Typen

- **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
```

kein returns, da kein Rückgabewert!

## Nebenläufige Objekte

“Nebenläufiges Objekt”:

Nebenläufige Datenstruktur mit atomaren Operationen

Beispiel:

**Atomares Speicherregister (als Objekt)**

Gemeinsames Speicherregister mit

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

## Test-and-set

**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

### 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, dass zusätzlich die atomare *read*-Operation unterstützt

## Swap

### 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

### Swap-Objekt

Ein gemeinsames Speicherregister mit

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

## Fetch-and-add

### 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-add Objekt

### 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

### 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.  
Z.B.: *f* eine atomare Operation wie *fetch-and-increment*  
⇒ unklar was atomar ausgeführt wird? Deswegen: Verboten!

## RMW-Objekt

### Read-Modify-Write-Objekt

Ein gemeinsames Speicherregister mit

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

## Compare-and-swap

### compare-and-swap

Operation 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

### Compare-and-swap-Objekt

Ein gemeinsames Speicherregister mit

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

## Sticky-Write

### 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 (Objekt)

### Sticky-Bit

Ein gemeinsames Speicherregister mit

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

## Move

### 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

### Move-Objekt

Gruppe von gemeinsamen Speicherregistern, so dass

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

## Shared swap

### 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

### 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-Bits

### Test-and-set-Bit = Lock (Sperre)

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
```

$x := 01$   
return von t-a-s: 0  
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
```

## 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 anderen Prozesse das Ergebnis 1
  - Erst nach atomarem *reset* kann der nächste Prozess den Lock setzen
- ist **Deadlock-frei**
  - Wenn mehrere Prozesse im Initialisierungscode und keiner im kritischen Abschnitt, dann muss  $x=0$  nach endlich vielen Schritten gelten
  - Danach: Einer muss nach endlich 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
    
```

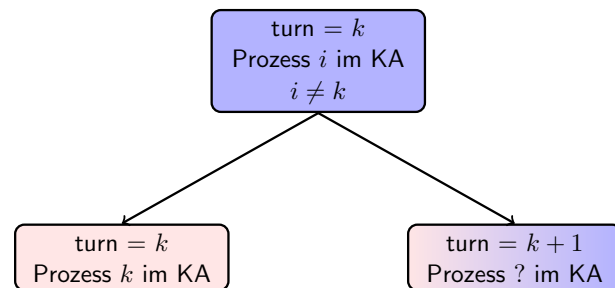
Initial:  
 turn: atom. Register, Wert egal (zwischen 1..n);  
 Test-and-set-Bit lock: 0;  
 waiting[i]: False;  
 lturn, key: lokale Register

## 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

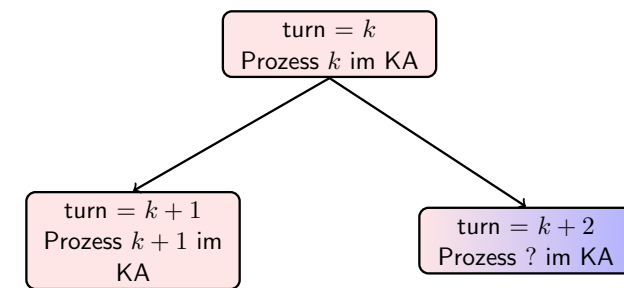
Zwei Fälle: Fall 1:



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

## Beweis der $n$ -Fairness (2)

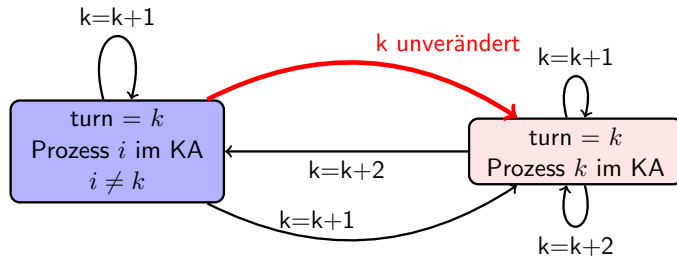
Zwei Fälle: Fall 2:



Rote Knoten = Prozess  $u$  im kritischen Abschnitt und turn =  $u$   
 Blaue Knoten = Prozess  $u$  im kritischen Abschnitt und 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<sub>*i*</sub>, valid<sub>*i*</sub>) 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)  $(t_1, v_1) := r-m-w((t, v), inc-fst)$ 
(3) while  $t_1 \neq v_1$  do
(4)    $v_1 := v;$ 
(5) Kritischer Abschnitt
(6)  $r-m-w((t, v), inc-snd)$ 
       $t_1 = ?2$ 
       $v_1 = ?12$ 
```

### 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)  $r-m-w((t, v), inc-snd)$ 
       $t_3 = ?1$ 
       $v_3 = ?1$ 
```

### 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)  $r-m-w((t, v), inc-snd)$ 
       $t_2 = ?3$ 
       $v_2 = ?1$ 
```

```
t = 1231
v = 12
```

## Eigenschaften des Ticket-Algorithmus

### 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

## Notation zu Zeigern

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 ≠ 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 ≠ Nil;
(13)     succ := *mynode.next;
(14)     *succ.value := 0;
(15) else
(16)   succ := *mynode.next;
(17)   *succ.value := 0;
```

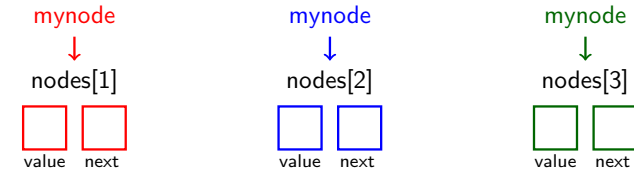
## Der MCS-Queue-basierte Algorithmus (4)

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 ≠ 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)

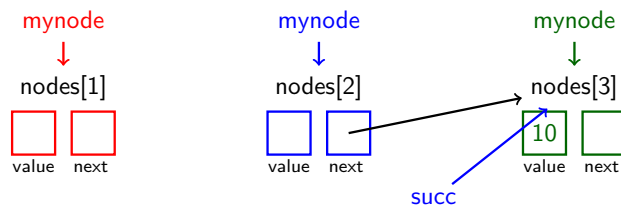
## Der MCS Queue-basierte Algorithmus: Beispiel

Nur ein Prozess will in den kritischen Abschnitt  
tail → Nil



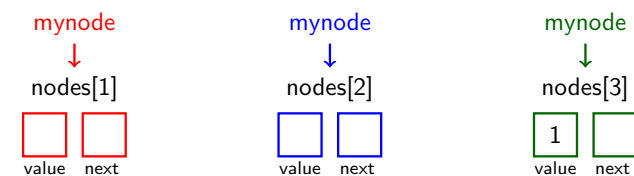
## Der MCS-Queue-basierte Algorithmus: Beispiel (2)

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



## Der MCS-Queue-basierte Algorithmus: Beispiel (3)

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



- Stärkere Speicheroperationen als atomares Read- und Write.
- Einige verschiedene Operationen.
- Mutual-Exclusion-Algorithmen mit diesen Operationen einfach, aber für gute Eigenschaften (Starvation-Freiheit, FIFO-Eigenschaften) auch nicht-trivial
- Ausblick:
  - Welche der Operationen braucht man?
  - Welche der Operationen sind stärker als andere?