# 43 Die Erzeugung neuer Threads

Wir nehmen an, der Ausdruck: $s \equiv$ **create** $(e_0, e_1)$ wertet erst die Ausdrücke $e_i$ zu Werten $f, a$ aus und erzeugt einen neuen Thread, der $f\ (a)$ abarbeitet.

Scheitert die Thread-Erzeugung, liefert $s$ den Wert -1 zurück, andernfalls liefert $s$ die tid des neuen Prozesses.

## Aufgaben des erzeugten Codes:

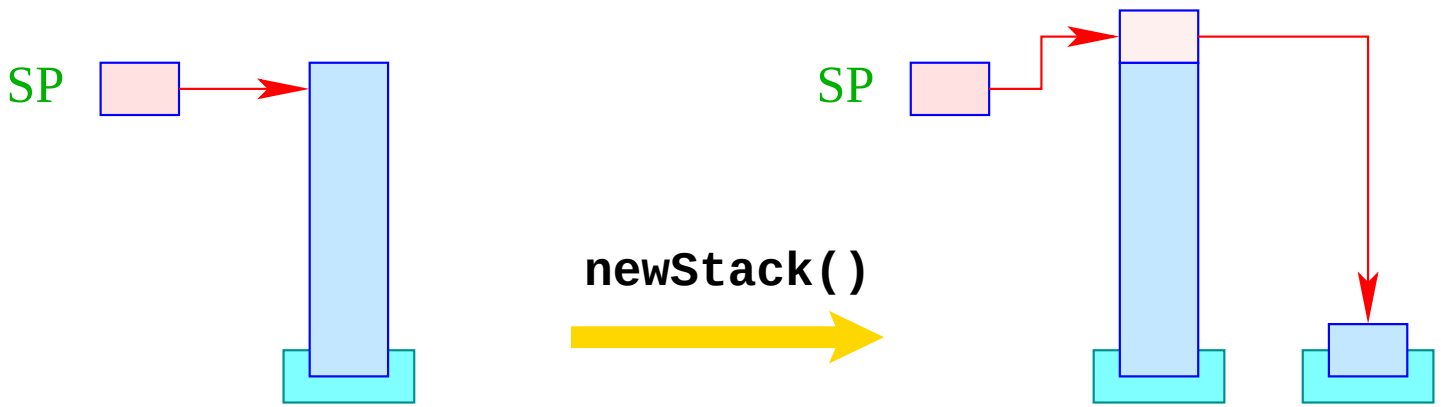
- Auswerten der $e_i$;
- Anlegen eines neuen Laufzeit-Stacks mit Keller-Rahmen zum Auswerten von $f\ (a)$;
- Erzeugen einer neuen tid;
- Anlegen eines neuen Eintrags in die TTab;
- Einfügen der neuen tid in die Ready-Schlange.

Die Übersetzung von $s$ ist dann ganz einfach:

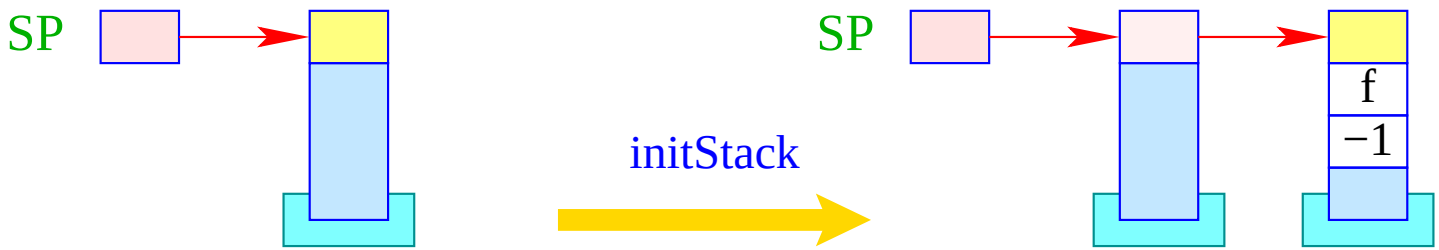
$$
\begin{array}{lcl}
\text{code}_\text{R}\ s\ \rho & = & \text{code}_\text{R}\ e_0\ \rho \\
 & & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{initStack} \\
 & & \text{initThread}
\end{array}
$$

wobei wir Platzbedarf 1 für den Wert des Arguments annehmen :-)

Zur Implementierung von initStack benötigen wir eine Laufzeit-Funktion newStack(), welche einen Pointer auf ein erstes Element eines neuen Stacks liefert:




Falls das Anlegen eines neuen Stacks scheitert, soll der Wert 0 zurück geliedert werden.




```
newStack();
if (S[SP]) {
        S[S[SP]+1] = -1;
        S[S[SP]+2] = f;
        S[S[SP]+3] = S[SP-1];
        S[SP-1] = S[SP]; SP--
    }
else S[SP = SP - 2] = -1;
```

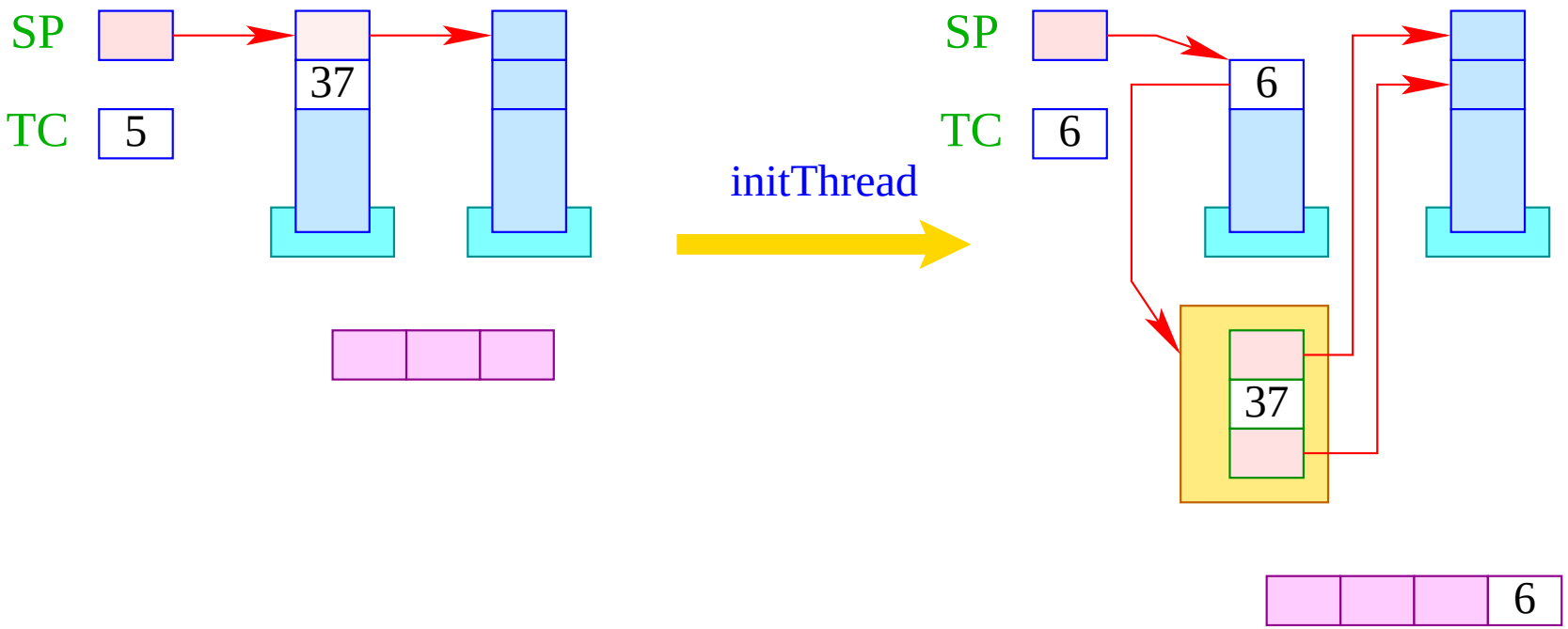
## Beachte:

- Die Fortsetzungs-Adresse f zeigt auf den (festen) Code zur Beendigung eines Thread.
- Im Kellerrahmen haben wir keinen Platz mehr für den EP allokiert $\Longrightarrow$ Der Rückgabe-Wert hat darum jetzt Relativ-Adresse -2.
- Den untersten Kellerrahmen erkennen wir daran, dass dort FPold = -1 ist.

Um neue Thread-Ids erzeugen zu können, spendieren wir uns ein neues Register TC (Thread Count).

Anfangs hat TC den Wert 0 (entspricht der tid des Start-Threads).

Vor Erzeugen eines neuen Threads, wird TC um eins erhöht.


SP
TC
5
37
initThread
SP
TC
6
6
37
6

```
if (S[SP] ≥ 0) {
        tid = ++TCount;
        TTab[tid][0] = S[SP]-1;
        TTab[tid][1] = S[SP-1];
        TTab[tid][2] = S[SP];
        S[--SP] = tid;
        enqueue( RQ, tid );
}
```

# 44 Die Beendigung von Threads

Die Beendigung eines Threads liefert (normalerweise :-) einen Wert zurück. Es gibt zwei (reguläre) Verfahren, um einen Thread zu beenden:

1. Der anfängliche Funktions-Aufruf terminiert. Der Rückgabe-Wert des Threads ist gleich des Aufrufs.
2. Der Thread führt das Statement **exit** $(e)$; aus. Der Rückgabe-Wert des Threads ist gleich dem Wert von $e$.

## Achtung:

- Den Rückgabe-Wert wollen wir in der untersten Stack-Zelle übergeben.
- **exit** kann tief geschachtelt in einer Rekursion vorkommen. Dann geben wir sämtliche Kellerrahmen des Threads frei.
- Anschließend springen wir die End-Behandlung von Threads an der Adresse f am Ende des Programms an.

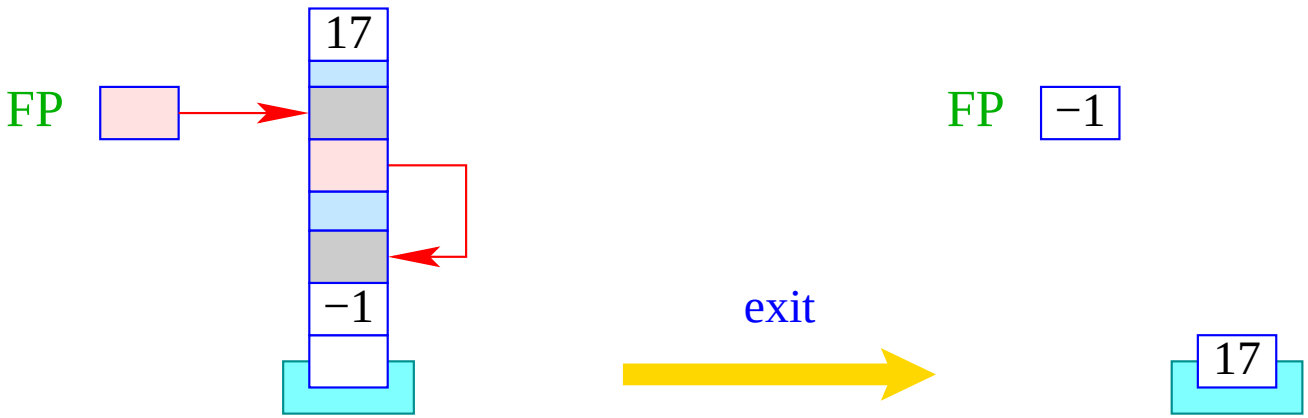
Damit übersetzen wir:

$$\begin{array}{lcl} \text{code } \textbf{exit } (e);\ \rho & = & \text{code}_{\text{R}}\ e\ \rho \\ & & \text{exit} \\ & & \text{term} \\ & & \text{next} \end{array}$$

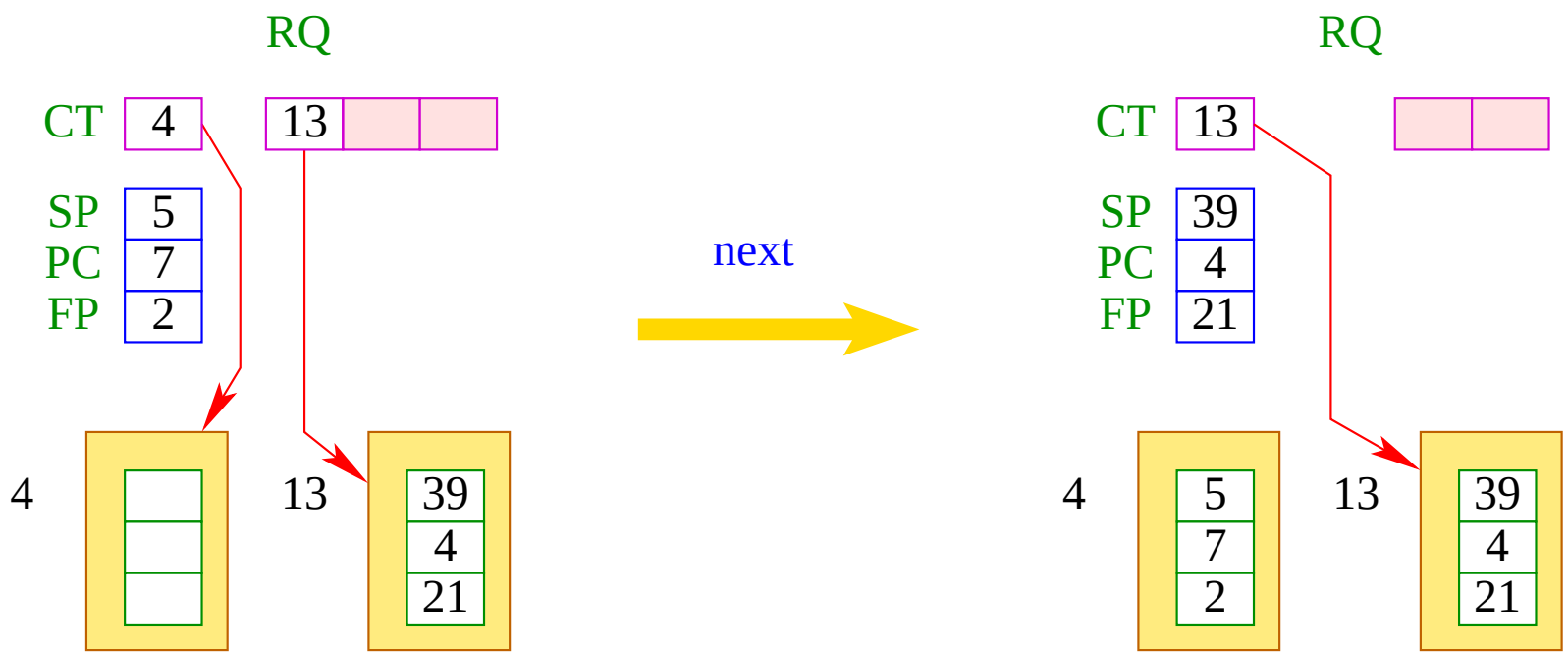
Die Instruktion term behandeln wir später :-)

Die Instruktion exit muss sukzessive sämtliche Keller-Rahmen des Threads aufgeben:

```
result = S[SP];
while (FP ≠ –1) {
        SP = FP–2;
        FP = S[FP–1];
        }
S[SP] = result;
```


17
FP
−1
exit
FP
−1
17

Die Instruktion next aktiviert den nächsten lauffähigen Thread:
im Gegensatz zu yield wird jedoch der aktuelle Thread nicht wieder in RQ eingefügt.



Ist die Schlange RQ leer, wird zusätzlich das Programm beendet:

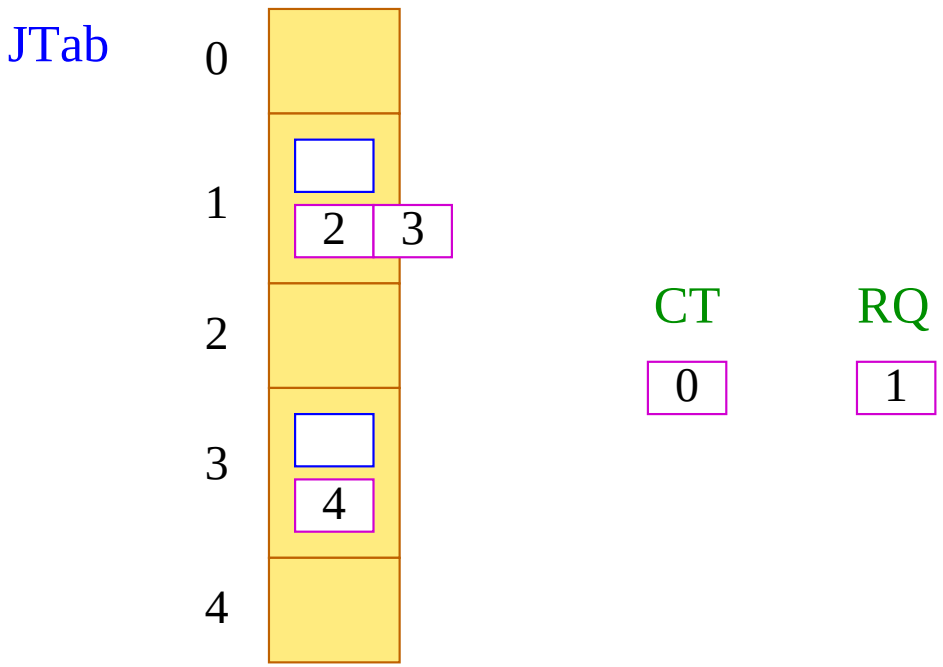
```
if (0 > ct = dequeue( RQ ))  halt;
else {
        save ();
        CT = ct;
        restore ();
     }
```

# 45 Warten auf Terminierung

Manchmal darf ein Thread erst mit seiner Ausführung fortfahren, wenn ein anderer Thread terminierte. Dafür gibt es den Ausdruck **join** $(e)$. Dabei erwarten wir, dass sich $e$ zu einer Thread-Id tid auswerten lässt.

- Ist der Thread mit dieser Kennung bereits beendet, soll dessen Rückgabe-Wert geliefert werden.
- Ist er noch nicht beendet, müssen wir die aktuelle Programm-Ausführung unterbrechen.
- Wir fügen den aktuellen Thread in die Schlange der anderen bereits auf Terminierung wartenden Threads ein, retten die aktuellen Register und schalten auf den nächsten ausführbaren Thread um.
- Auf Terminierung wartende Threads verwalten wir in der Tabelle JTab.
- Dort legen wir auch den Rückgabe-Wert der Threads ab :-)

# Beispiel:




Thread 0 ist am Laufen, Thread 1 könnte laufen, Threads 2 und 3 warten auf Terminierung von 1, und Thread 4 wartet auf Terminierung von 3.

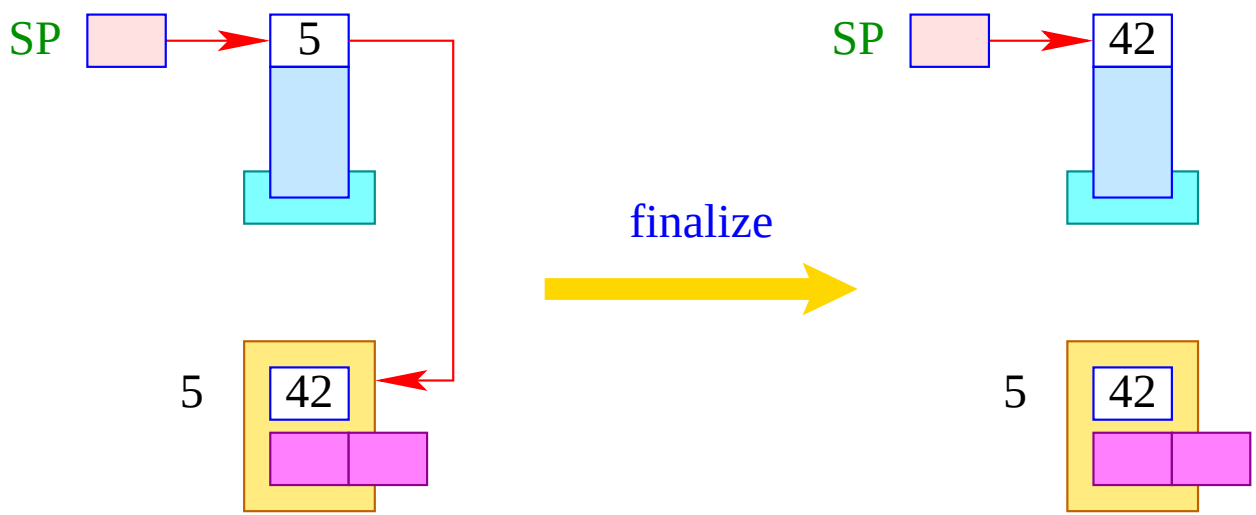
Damit übersetzen wir:

$$\begin{array}{lcl} \text{code}_R\ \textbf{join}\ (e)\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{join} \\ & & \text{finalize} \end{array}$$

... wobei die Instruktion join definiert ist als:

```
tid = S[SP];
if (TTab[tid][1] ≥ 0) {
        enqueue ( JTab[tid], CT );
        next
    }
```

... sowie:




S[SP] = JTab[tid][1];

Die Instruktions-Folge:

```
term
next
```

soll zuletzt ausgeführt werden, bevor ein Thread terminiert.
Deshalb schreiben wir sie auch an die Stelle f.

Die Instruktion next schaltet zum nächsten lauffähigen Thread weiter.
Vorher muss allerdings noch:

- ... der letzte Kellerrahmen aufgegeben und das Resultat in der Tabelle JTab abgelegt werden;
- ... kenntlich gemacht werden, dass der Thread terminiert ist, z.B. indem der PC auf -1 gesetzt wird;
- ... sämtliche Threads aufgeweckt werden, die auf Beendigung des Threads gewartet haben.

Für die Instruktion term heißt das:

```
PC = –1;
JTab[CT][1] = S[SP];
freeStack(SP);
while (0 ≤ tid = dequeue ( JTab[CT][0] ))
        enqueue ( RQ, tid );
```

Die Laufzeit-Funktion `freeStack (int adr)` beseitigt den (ein-elementigen) Stack an der Stelle adr :

# 46 Wechselseitiger Ausschluss

Ein Mutex ist ein (abstrakter) Datentyp (in der Halde), die es der Programmiererin gestatten soll, gemeinsame Resourcen für einen Thread exklusiv zu reservieren (wechselseitiger Ausschluss / mutual exclusion).

Der Datentyp unterstützt folgende Operationen:

| | | |
|---|---|---|
| **Mutex** $*$ newMutex (); | — | legt neuen Mutex an; |
| **void** lock (**Mutex** $*$me); | — | versucht, den Mutex zu erwerben; |
| **void** unlock (**Mutex** $*$me); | — | versucht, den Mutex frei zu geben. |

Achtung:

Ein Thread darf einen Mutex nur frei geben, wenn es über diesen verfügt :-)

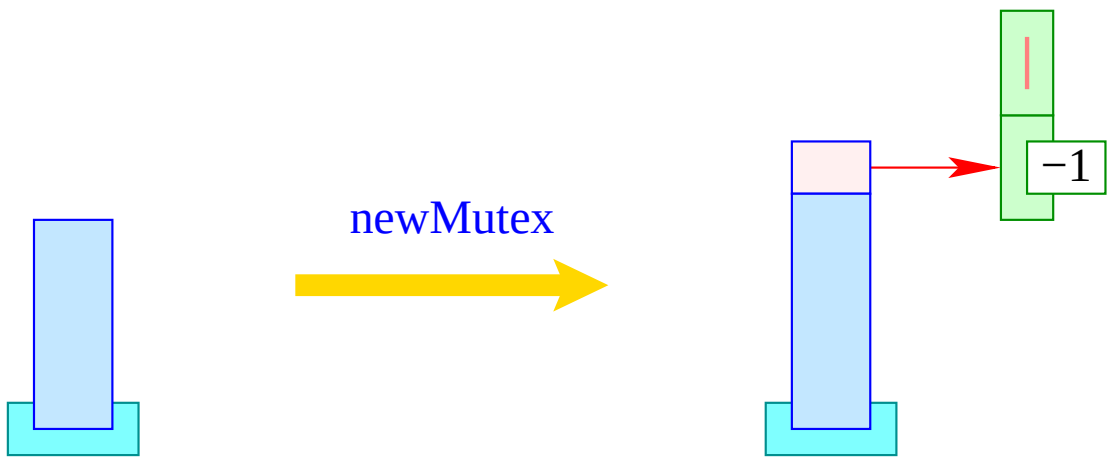
Ein Mutex me besteht aus:

- der tid des gegenwärtigen Besitzers (bzw. -1 falls es keinen gibt);
- der Schlange BQ der blockierten Threads, die den Mutex erwerben wollen.

Dann übersetzen wir:

$$\text{code}_R\ \textbf{newMutex}\ ()\ \rho \quad = \quad \text{newMutex}$$

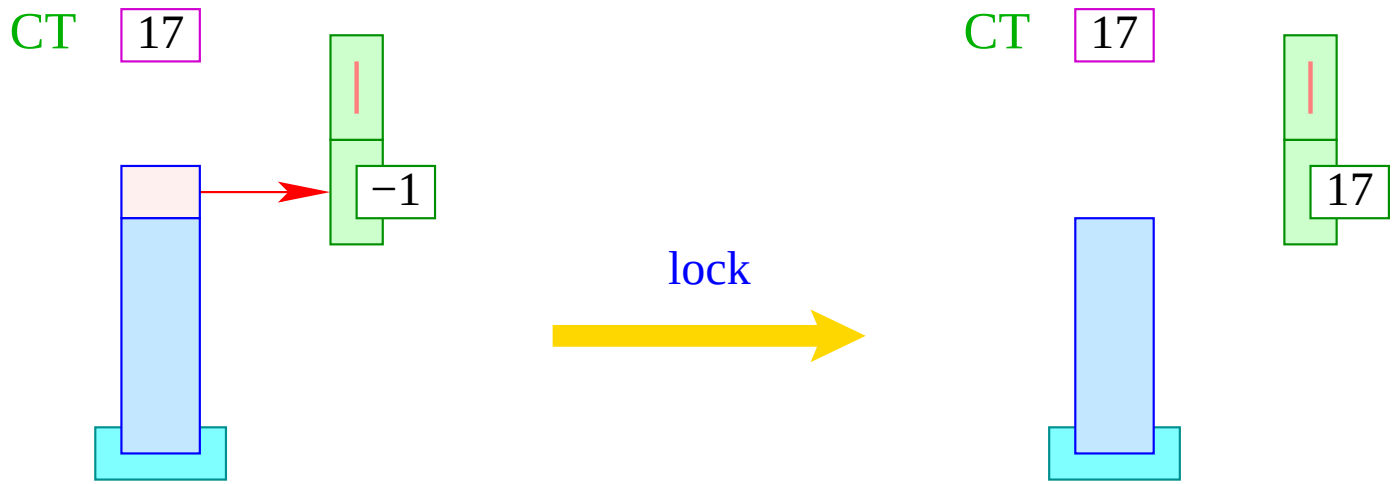
wobei:



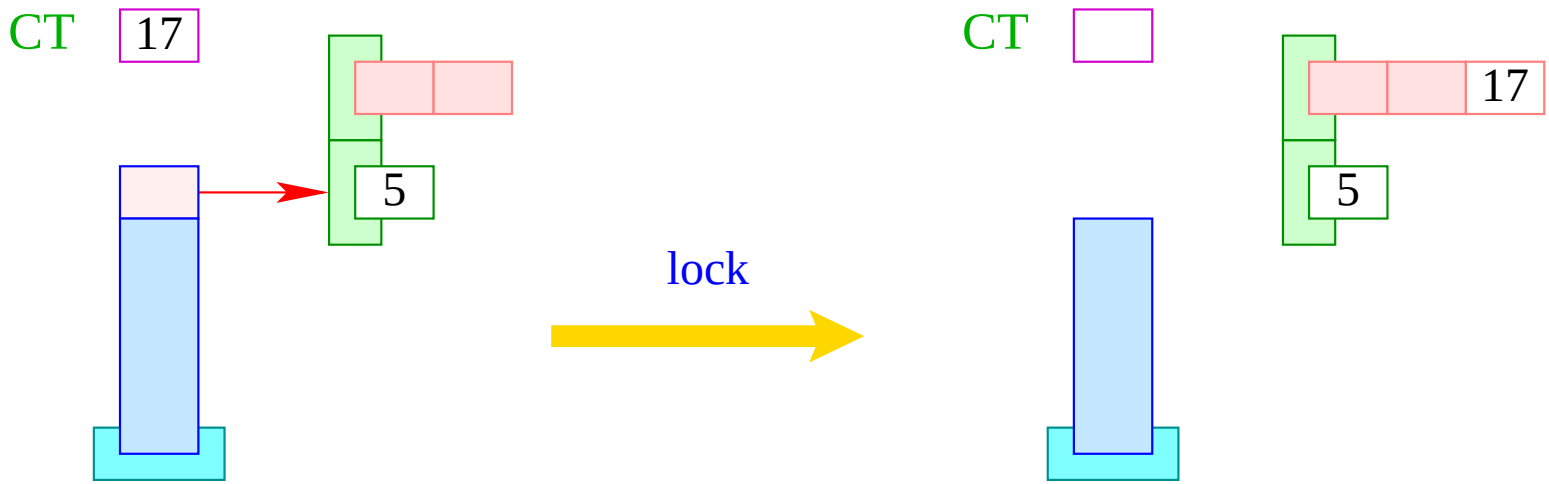
Dann übersetzen wir:

$$\text{code } \textbf{lock } (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{lock}$$

wobei:



Ist der Mutex bereits vergeben, wird der aktuelle Thread unterbrochen:

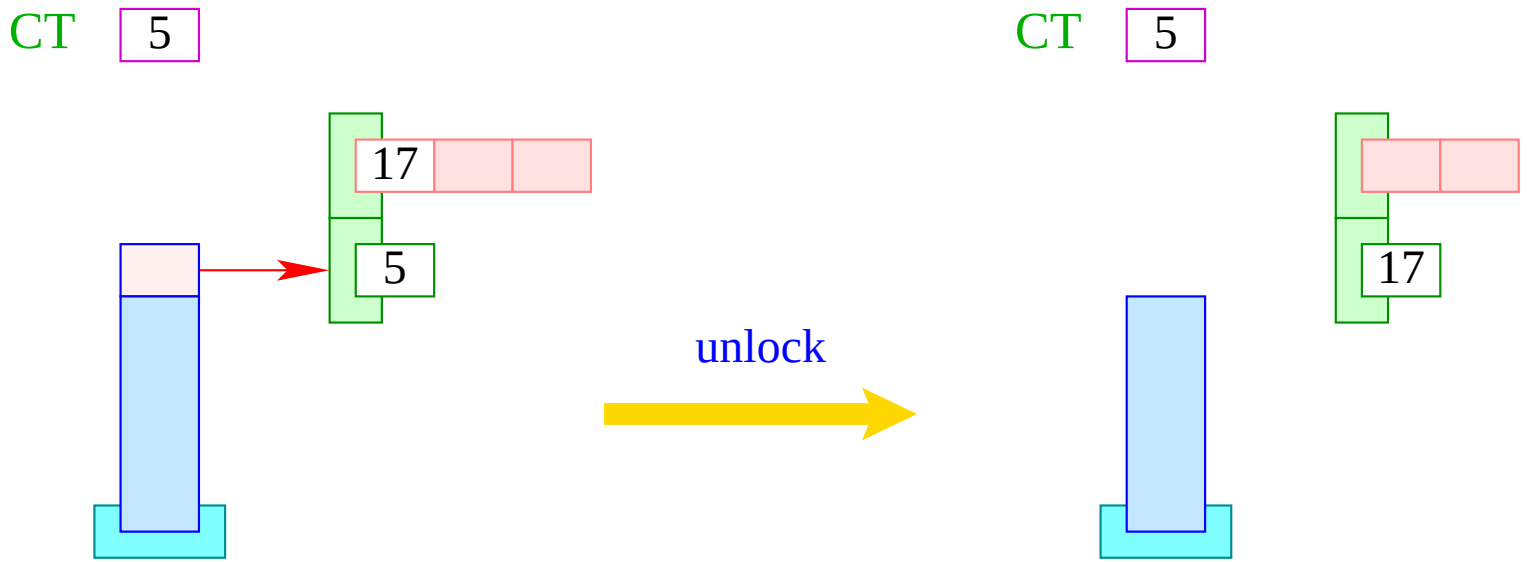



```
if (S[S[SP]] < 0)    S[S[SP−−]] = CT;
else {
        enqueue ( S[SP−−]+1, CT );
        next;
    }
```

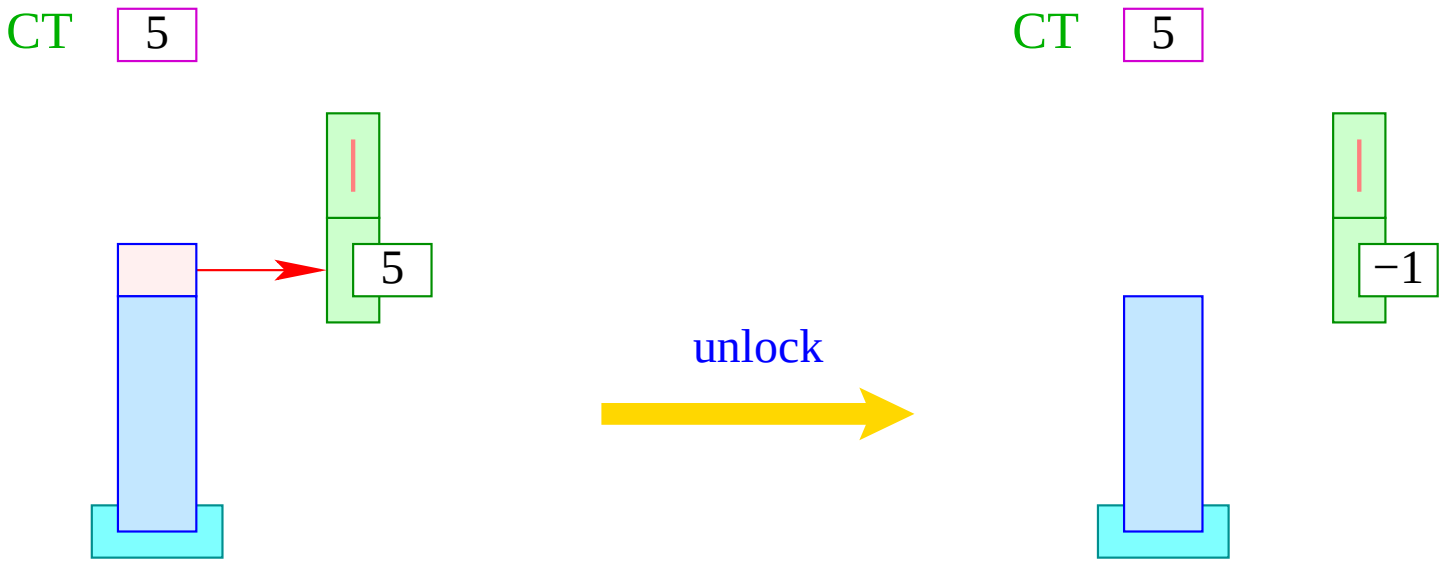
Entsprechend übersetzen wir:

$$\text{code } \textbf{unlock } (e);\ \rho \quad = \quad \begin{array}{l} \text{code}_{\text{R}}\ e\ \rho \\ \text{unlock} \end{array}$$

wobei:



Ist die Schlange BQ leer, geben wir den Mutex ganz frei:




```
if (S[S[SP]] ≠ CT)    Error ("Illegal unlock!");
if (0 > tid = dequeue ( S[SP]+1))    S[S[SP– –]] = –1;
else {
        S[S[SP--]] = tid;
        enqueue ( RQ, tid );
    }
```

# 47 Warten auf den Frühling

Es kann vorkommen, dass ein Thread zwar über einen Mutex verfügt, nun aber warten muss, bis eine Bedingung eingetreten ist.

Dann soll der Thread sich selbst blockieren, um später reaktiviert zu werden. Dazu dienen Bedingungsvariablen. Eine Bedingungsvariable besteht aus einer einer Schlange WQ wartender Threads :-)

Für Bedingungsvariablen gibt es die Funktionen:

| | | |
|---|---|---|
| **CondVar** $*$ newCondVar (); | — | legt eine Bedingungsvariable an; |
| **void** wait (**CondVar** $*$ cv), **Mutex** $*$ me); | — | legt aktuellen Thread schlafen; |
| **void** signal (**CondVar** $*$ cv); | — | weckt einen wartenden Thread; |
| **void** broadcast (**CondVar** $*$ cv); | — | weckt alle wartenden Threads. |

Dann übersetzen wir:

$$\text{code}_{\text{R}}\ \textbf{newCondVar}\ ()\ \rho \quad = \quad \text{newCondVar}$$

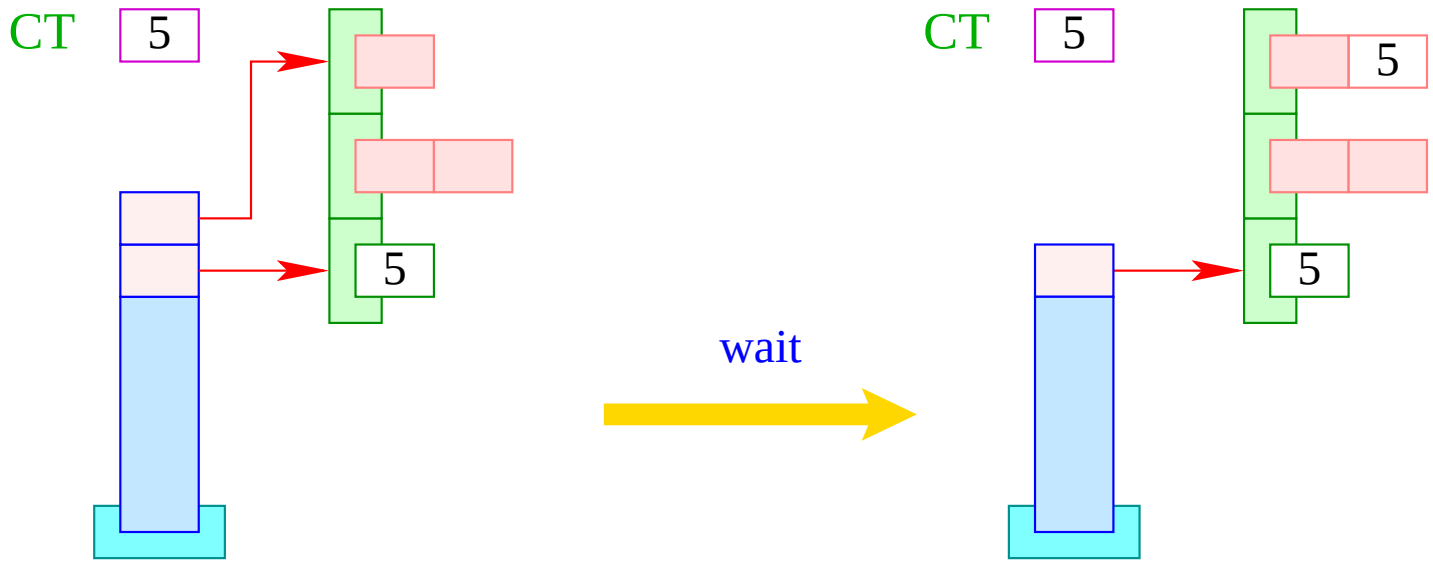
wobei:

Nach Einreihen in die Warteschlange wird der Mutex wieder frei gegeben. Nach dem Aufwecken muss dieser allerdings neu erworben werden.

Darum übersetzen wir:

$$
\begin{aligned}
\text{code } \textbf{wait } (e_0, e_1);\ \rho \quad = \quad & \text{code}_\text{R}\ e_1\ \rho \\
& \text{code}_\text{R}\ e_0\ \rho \\
& \text{wait} \\
& \text{dup} \\
& \text{unlock} \\
& \text{next} \\
& \text{lock}
\end{aligned}
$$

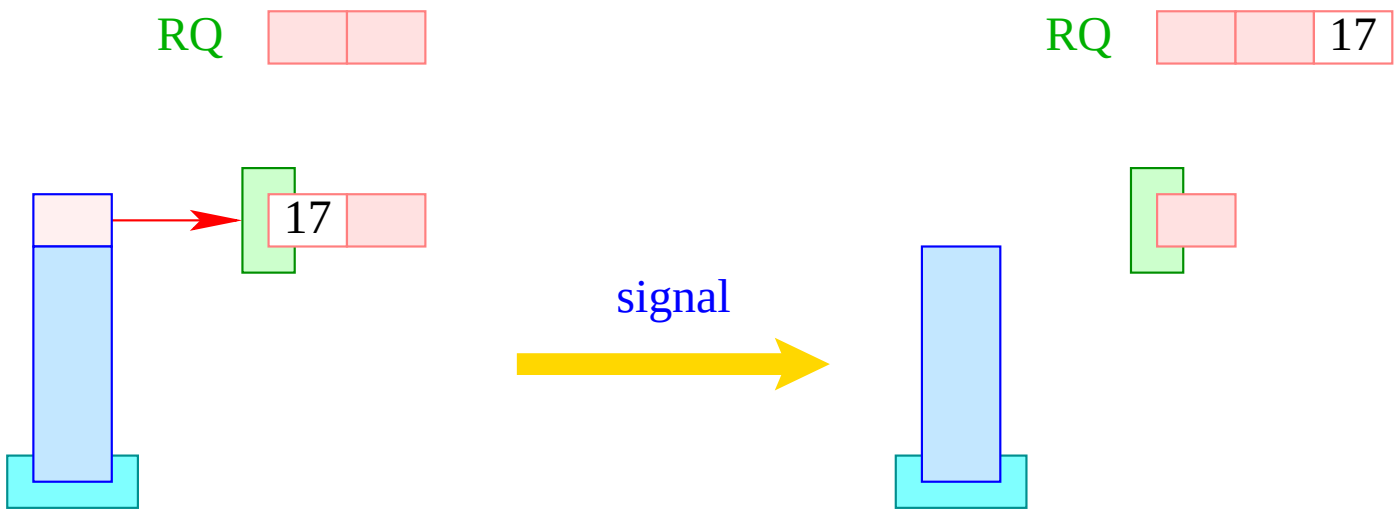
wobei ...




if (S[S[SP-1]] $\neq$ CT)  Error ("Illegal wait!");
enqueue ( S[SP], CT ); SP--;

Entsprechend übersetzen wir:

$$\text{code } \textbf{signal } (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{signal}$$




```
if (0 ≤ tid = dequeue ( S[SP]))
        enqueue ( RQ, tid );
SP--;
```

Analog:

$$\text{code } \textbf{broadcast}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{broadcast}$$

wobei die Instruktion broadcast sämtliche Threads der Schlange WQ in die Schlange RQ einfügt:

```
while (0 ≤ tid = dequeue ( S[SP]))
        enqueue ( RQ, tid );
SP--;
```

## Achtung:

Die aufgeweckten Threads sind nicht blockiert !!!

Wenn sie aktiv werden, benötigen sie jedoch als erstes das Lock ihres Mutex :-)

# 48 Beispiel: Semaphore

Ein Semaphor ist ein abstrakter Datentyp, der den Zugang zu einer festen Anzahl (identischer) Resourcen regeln soll.

Operationen:

| | | |
|---|---|---|
| Sema $*$ newSema (**int** n ) | — | liefert einen Semaphor; |
| **void** Up (Sema $*$ s) | — | gibt eine Resource frei; |
| **void** Down (Sema $*$ s) | — | allokiert eine Resource. |

Ein Semaphor besteht darum aus:

- einem Zähler vom Typ **int**;
- einem Mutex zur Synchronisation der Semaphor-Operationen;
- einer Bedingungsvariablen.

```
typedef struct {
        Mutex * me;
        CondVar * cv;
        int count;
} Sema;
```

```
Sema * newSema (int n) {
        Sema * s;
        s = (Sema *) malloc (sizeof (Sema));
        s→me = newMutex ();
        s→cv = newCondVar ();
        s→count = n;
        return (s);
    }
```

Die Übersetzung liefert für den Rumpf:

```
alloc 1
loadc 3
new
storer 2
pop
```

```
newMutex
loadr 2
store
pop
```

```
newCondVar
loadr 2
loadc 1
add
store
pop
```

```
loadr 1
loadr 2
loadc 2
add
store
pop
```

```
loadr 2
storer -2
return
```