Threads

39 Die Sprache ThreadedC

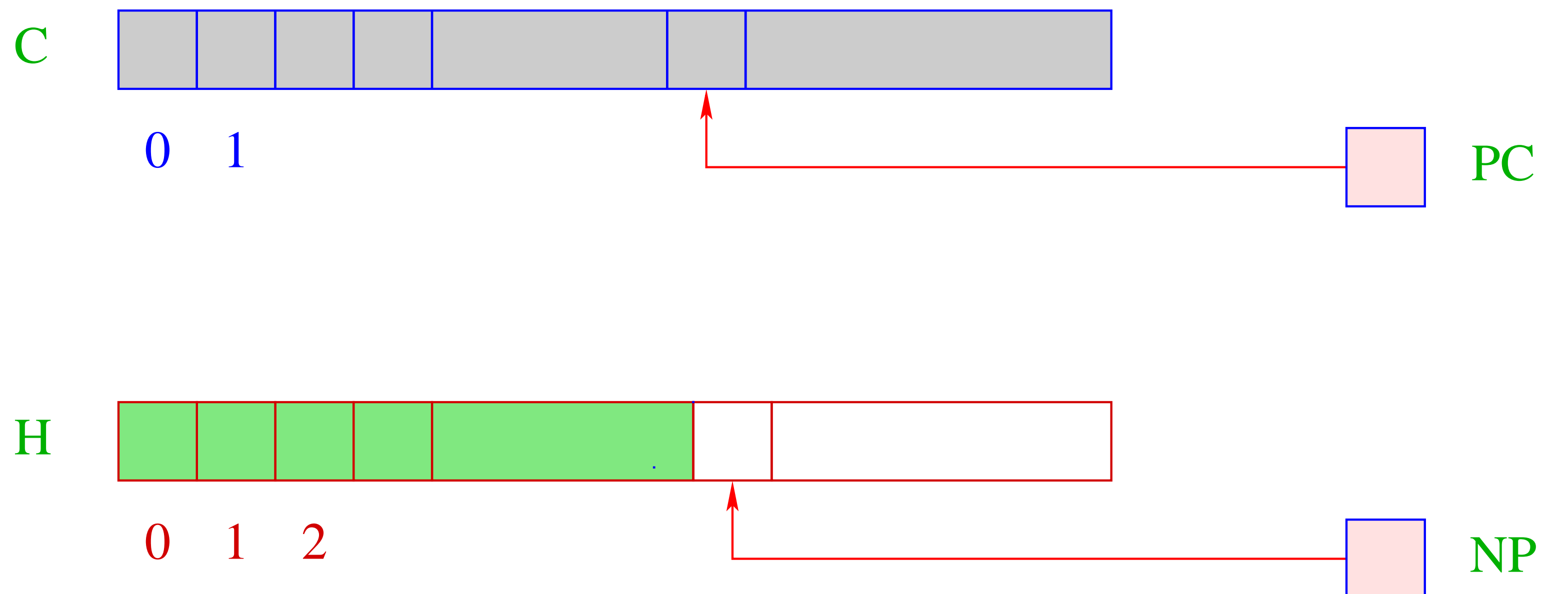
Wir erweitern C um ein einfaches Thread-Konzept. Insbesondere stellen wir Funktionen bereit, um:

- neue Threads zu erzeugen: create();
- einen Thread zu beenden: exit();
- auf die Terminierung eines Threads zu warten: join();
- wechselseitigen Ausschluss zu ermöglichen: lock(), unlock(); ...

Um eine parallele Programm-Ausführung zu ermöglichen, benötigen wir natürlich :-) eine Erweiterung der abstrakten Maschine ...

40 Speicher-Organisation

Allen Threads gemeinsam ist Code-Speicher und Halde:

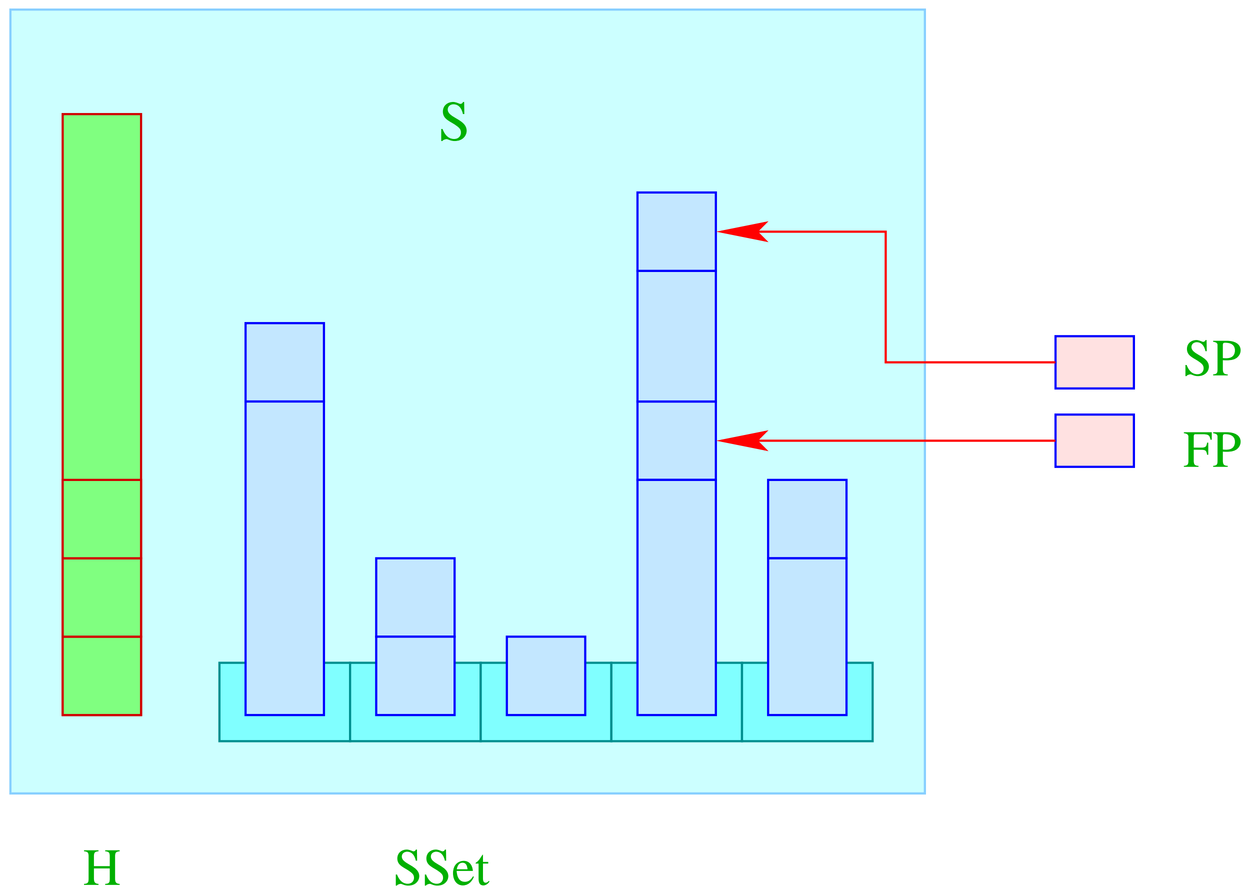


... wie bei der CMa haben wir:

C	=	Code-Speicher – enthält CMa-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter – zeigt auf nächsten auszuführenden Befehl;
H	=	Halde –
		jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
		am unteren Ende legen wir die globalen Variablen ab;
NP	=	New-Pointer – zeigt auf erste freie Zelle.

Nur nehmen wir zur Vereinfachung an, dass die Halde in einem eigenen Segment abgelegt ist. Die Funktion malloc() scheitert jetzt dann, wenn NP das obere Ende erreichte.

Jeder Thread benötigt andererseits seinen eigenen Stack:



Im Unterschied zur CMa haben wir:

SSet	=	Set of Stacks – enthält die Stacks der Threads;
		jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
S	=	gemeinsamer Adress-Raum für Halde und Stacks;
SP	=	Stack-Pointer – zeigt auf oberste belegte Zelle;
FP	=	Frame-Pointer – zeigt auf aktuellen Kellerrahmen.

Achtung:

- Zeigten alle Referenzen stets in die Halde, könnten wir für jeden Stack einen eigenen Adressraum verwenden.

  Dann müssten wir uns außer SP und FP auch merken, in welchem Stack wir uns befinden.

- Für C müssen wir allerdings annehmen, dass sämtliche Speicherbereiche im selben Adressraum liegen – jeweils an unterschiedlichen Stellen :-)

  SP und FP identifizieren damit eindeutige Stellen im Speicher.

- Der Einfachkeit halber verzichten wir auf Extreme-Pointer EP.

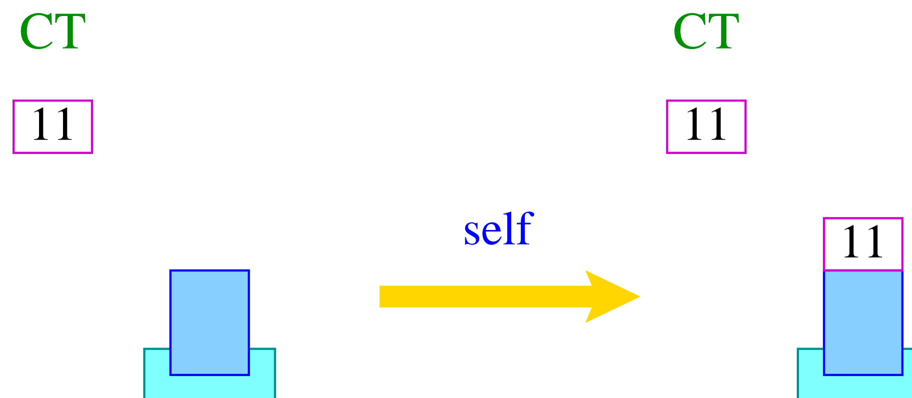
41 Die Ready-Schlange

Idee:

- Jeder Thread hat eine eindeutige Nummer tid.
- Eine Tabelle TTab ermöglicht es, zu einer tid den zugehörigen Thread zu ermitteln.
- Zu jedem Zeitpunkt kann es mehrere ausführbare Threads geben, aber nur einen laufenden (pro Prozessor) geben.
- Die tid des laufenden Threads steht im Register CT (Current Thread).
- Die Funktion: **tid self ()** liefert die tid des laufenden Threads. Folglich:

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

... wobei die Instruktion self den Inhalt des Registers CT auf den Stack lädt:

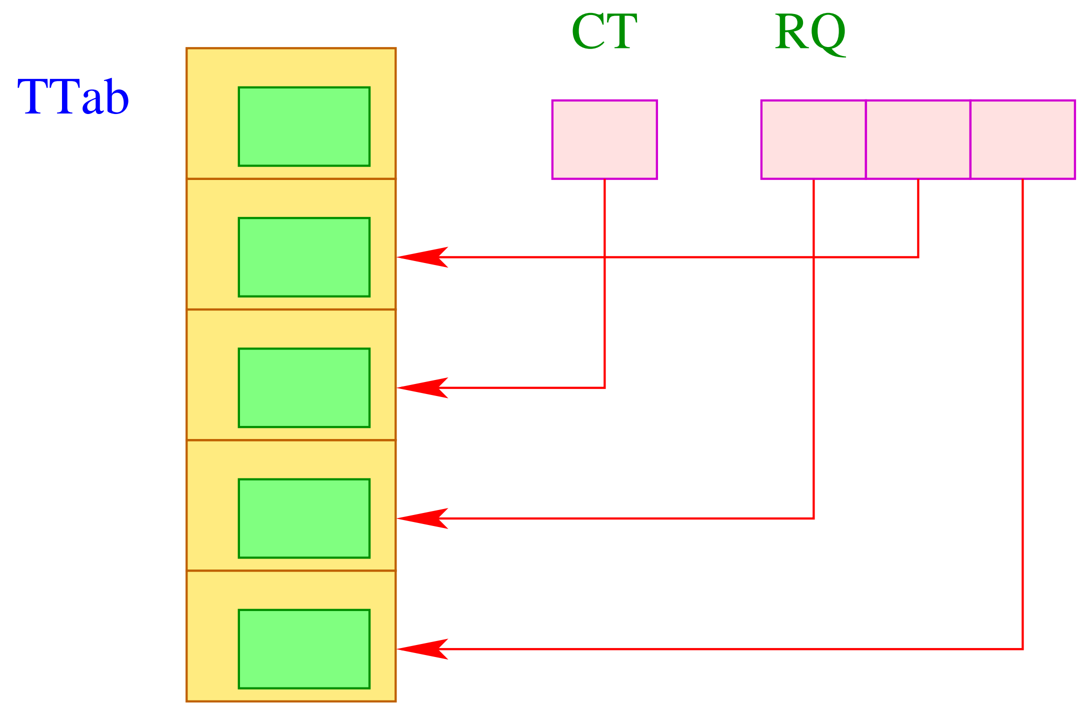


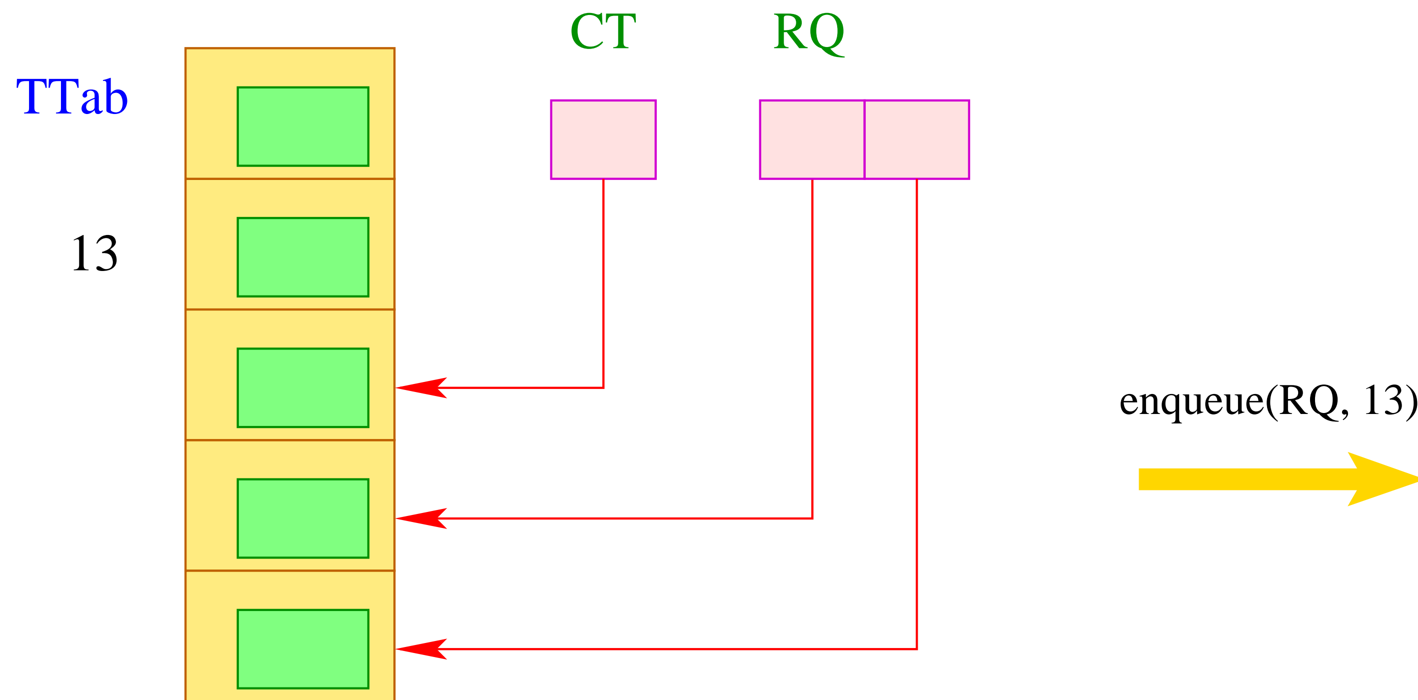
S[SP++] = CT;

- Die weiteren lauffähigen Threads (bzw. deren tid's) verwalten wir in einer Schlange RQ (Ready-Queue).
- Für Schlangen von Threads benötigen wir die Funktionen:

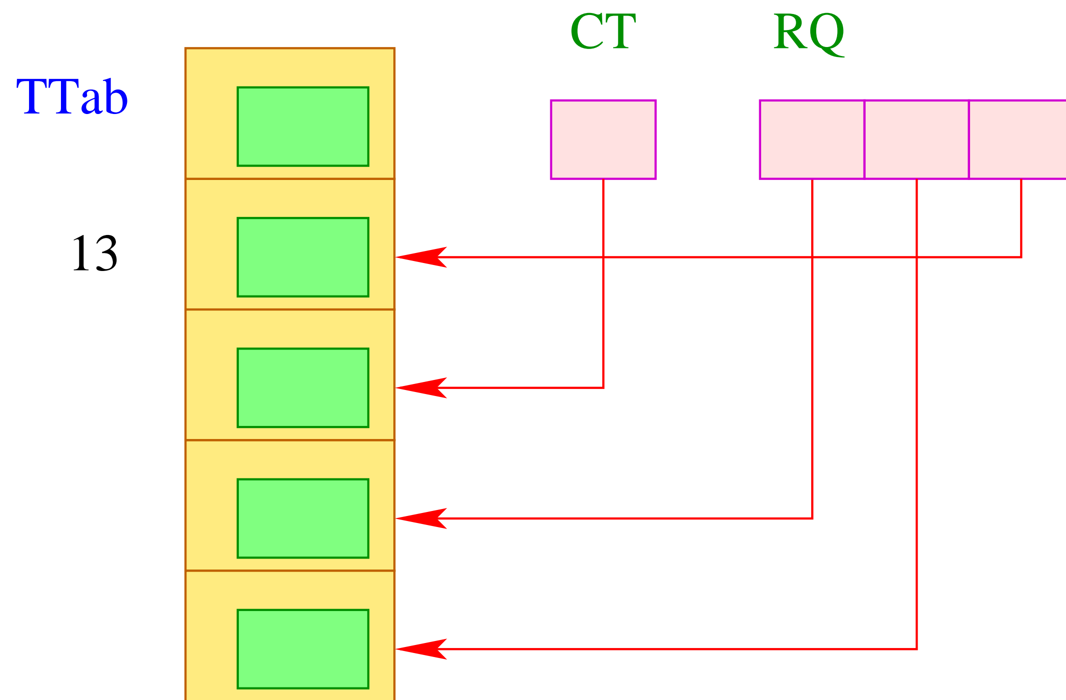
```
void enqueue (queue q, tid t),
tid dequeue (queue q)
```

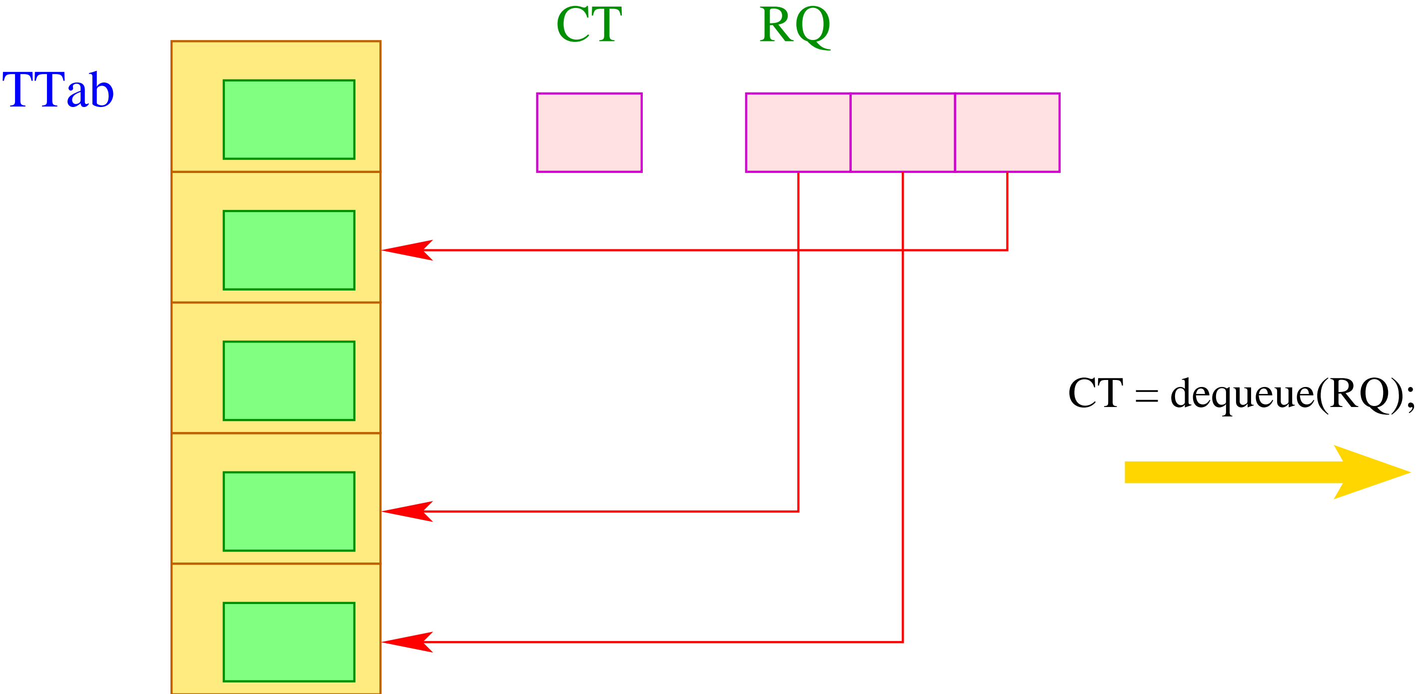
die eine neue tid in die Schlange einfügen bzw. die erste zurück liefern ...

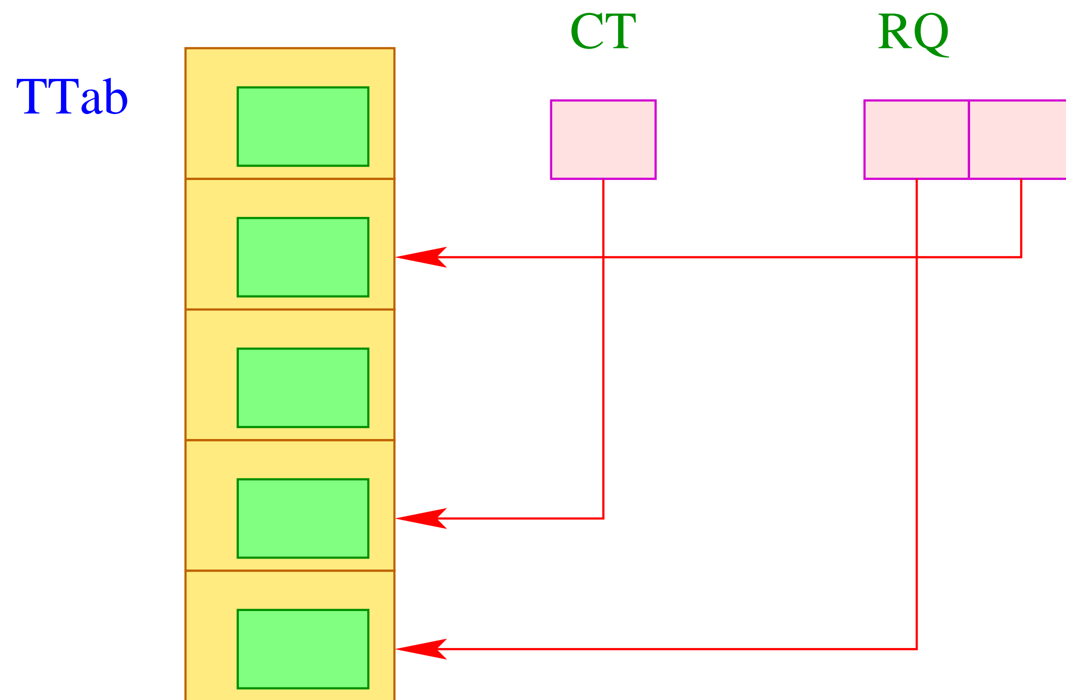

TTab
CT
RQ



TTab
13
CT
RQ
enqueue(RQ, 13)

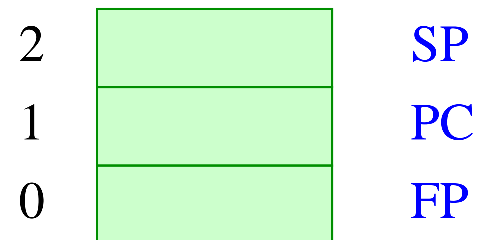

CT
RQ
TTab
13


CT
RQ
TTab
CT = dequeue(RQ);


TTab
CT
RQ

War der Aufruf von dequeue () nicht erfolgreich, liefert die Funktion einen Wert < 0 :-)

Die Thread-Tabelle muss für jeden Thread alle Informationen enthalten, die wir zu seiner Ausführung benötigen. Insbesondere sind das die aktuellen Register PC, SP und FP:



Unterbrechen des gegenwärtigen Threads erfordert darum das Retten dieser Register:

```
void save () {
    TTab[CT][0] = FP;
    TTab[CT][1] = PC;
    TTab[CT][2] = SP;
    }
```

Analog restauriert die Funktion `restore()` diese Register:

```
void restore () {
     FP = TTab[CT][0];
     PC = TTab[CT][1];
     SP = TTab[CT][2];
     }
```

Damit können wir nun eine Instruktion yield realisieren, die einen Thread-Wechsel herbei führt:

```
tid ct = dequeue ( RQ );
if (ct ≥ 0) {
      save (); enqueue ( RQ, CT );
      CT = ct;
      restore ();
      }
```

Nur wenn die Ready-Schlange nicht-leer ist, wird der aktuelle Thread ersetzt :-)

42 Thread-Wechsel

Problem:

Wir wollen alle lauffähigen Threads nach Möglichkeit fair abarbeiten.

- Jeder Thread muss früher oder später dran kommen.
- Jeder Thread muss früher oder später unterbrochen werden.

Mögliche Strategien:

- Thread-Wechsel nur bei explizitem Aufruf einer Funktion `yield()` :-(
- Thread-Wechsel nach jeder Instruktion $\Longrightarrow$ zu teuer :-(
- Thread-Wechsel nach einer festen Anzahl von Schritten $\Longrightarrow$ wir müssten einen Zähler mitführen und yield dynamisch einfügen :-(

Thread-Wechsel an ausgewählten Programm-Punkten ...

- am Anfang von Funktions-Rümpfen;
- vor jedem Sprung, dessen Ziel nicht größer ist als der aktuelle PC...

$\Longrightarrow$ selten :-))

Das modifizierte Schema für Schleifen $s \equiv \textbf{while}\ (e)\ s$ liefert dann etwa:

$$
\begin{array}{llll}
\text{code}\ s\ \rho & = & \text{A}: & \text{code}_\text{R}\ e\ \rho \\
 & & & \text{jumpz B} \\
 & & & \text{code}\ s\ \rho \\
 & & & \text{yield} \\
 & & & \text{jump A} \\
 & & \text{B}: & \ldots
\end{array}
$$

Beachte:

- **If-then-else**-Statements enthalten nicht notwendigerweise Thread-Wechsel.
- **do-while**-Schleifen erfordern einen Thread-Wechsel am Ende der Bedingung.
- Jede Schleife sollte (mindestens) einen Thread-Wechsel enthalten :-)
- Loop-Unroling verringert die Anzahl dieser Wechsel.
- Bei der Übersetzung von **switch**-Statements legten wir die Sprungtabelle hinter die Alternativen. Hier können wir trotzdem auf Thread-Wechsel verzichten.
- Bei frei programmierter Benutzung von jumpi wie auch jumpz sollte sicherheitshalber auch ein Thread-Wechsel vor dem Sprung (oder am Sprung-Ziel) eingefügt werden.
- Will man die Anzahl der Thead-Wechsel weiter reduzieren, kann man z.B. nur bei jedem 100. Aufruf von yield den Kontext wechseln ...

43 Die Erzeugung neuer Threads

Wir nehmen an, der Ausdruck: $s \equiv \textbf{create}\ (e_0, e_1)$ 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:

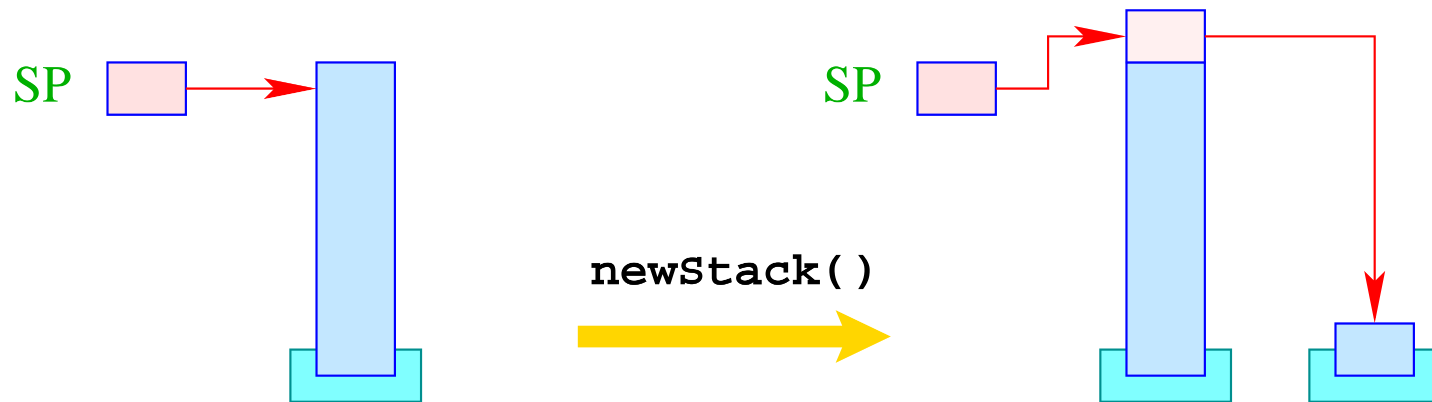
- Auswerten der e_i;
- Anlegen eines neuen Laufzeit-Stacks mit Keller-Rahmen zum Auswerten von $f\ (a)$;
- Erzeugen einer neuen tid;
- Anlegen eines neuen Eintrags in die TTab;
- Einfügen der neuen tid in die Ready-Schlange.

Die Übersetzung von s ist dann ganz einfach:

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

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;
```

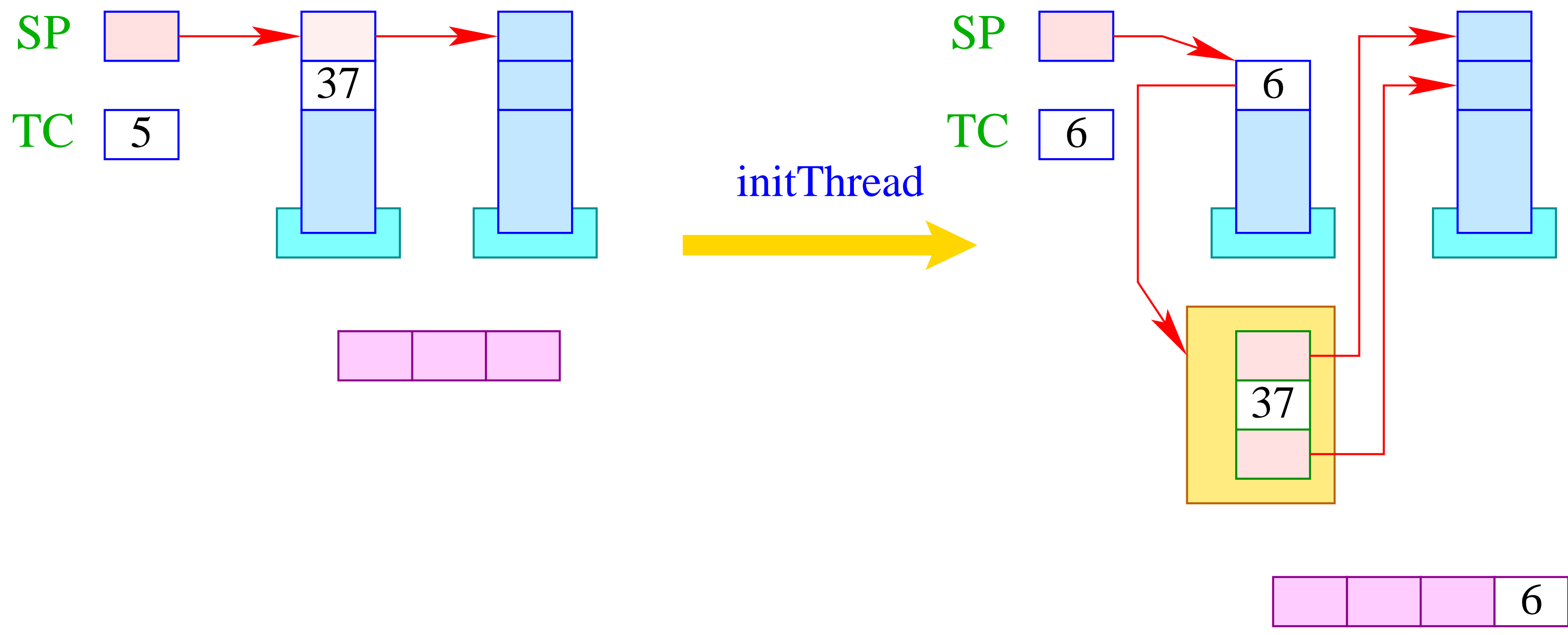
Beachte:

- 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:

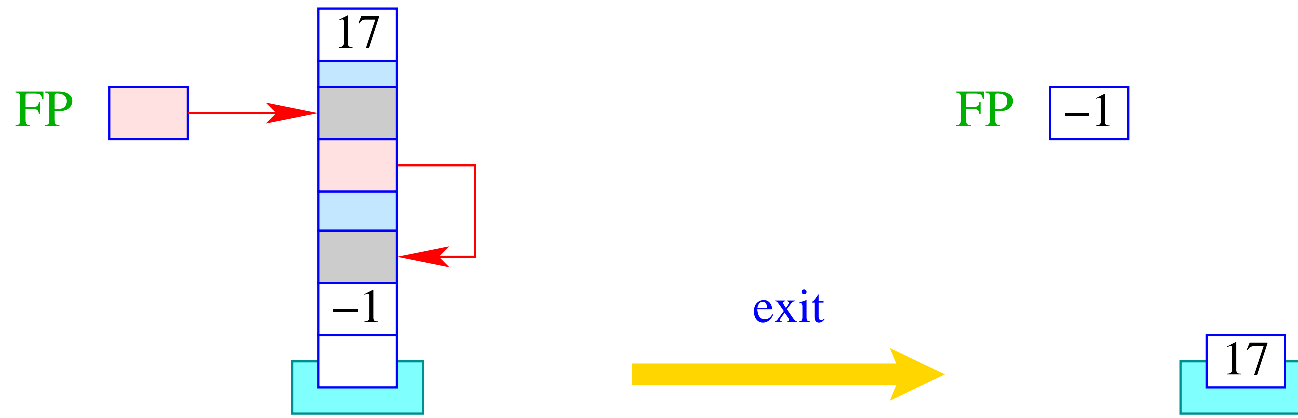
$$\text{code } \textbf{exit } (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$

$$\begin{array}{l} \text{exit} \\ \text{term} \\ \text{next} \end{array}$$

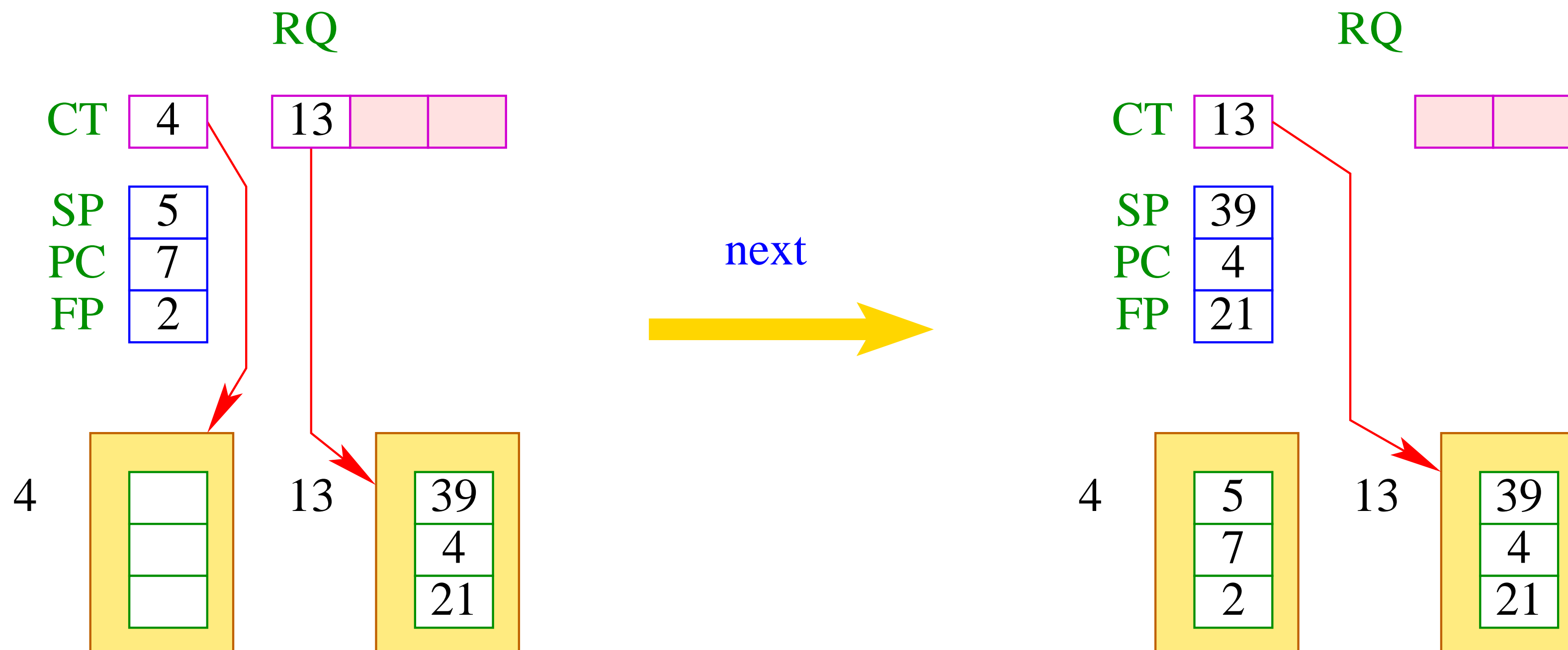
Die Instruktion term behandeln wir später :-)

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;
```


FP
17
−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:

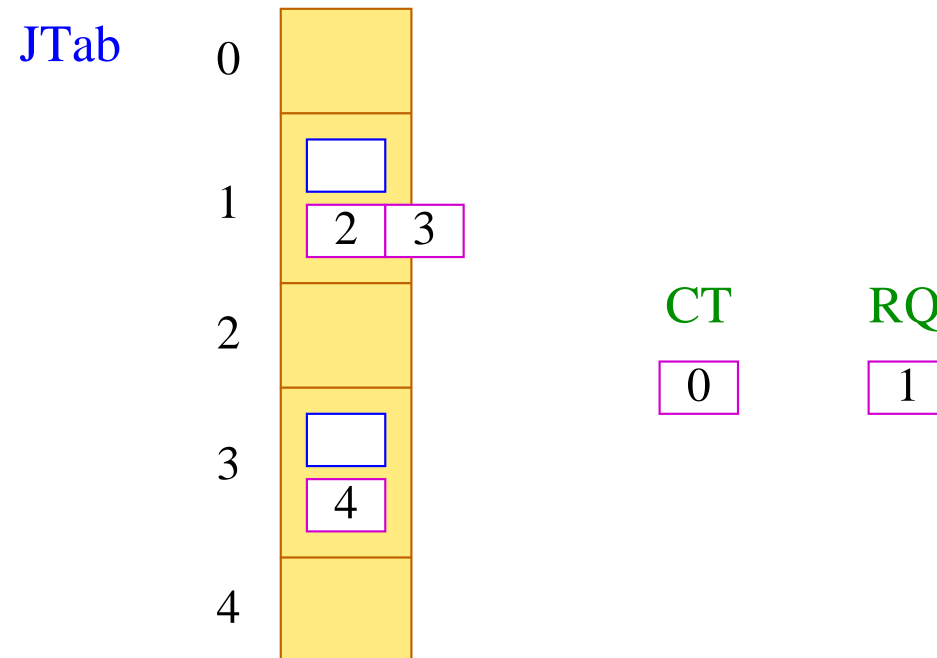
```
if (0 > ct = dequeue( RQ ))   halt;
else {
        save ();
        CT = ct;
        restore ();
    }
```

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.

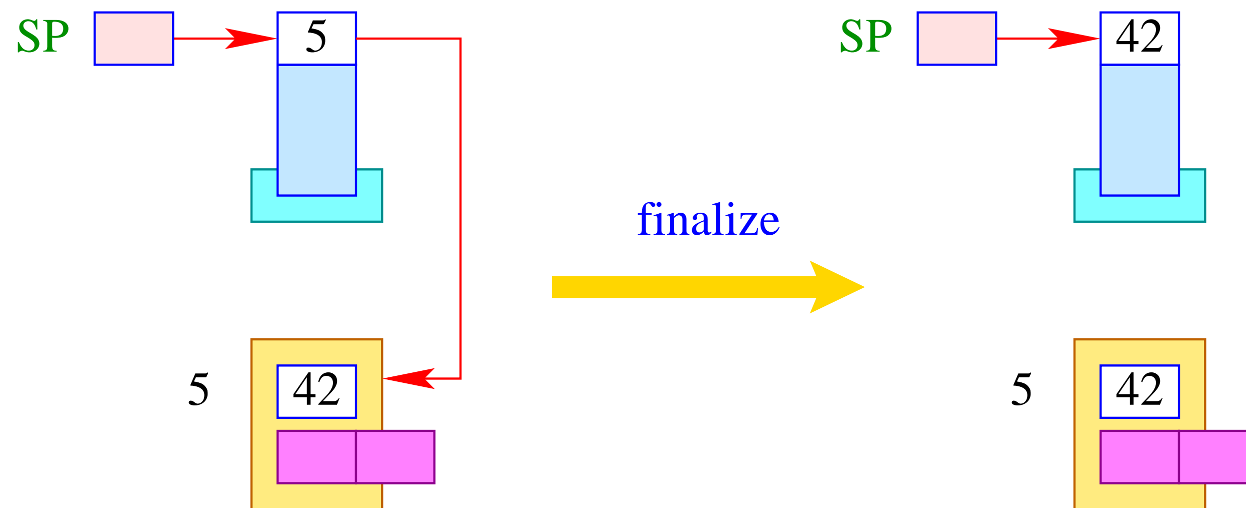
Damit übersetzen wir:

$$\text{code}_R\ \mathbf{join}\ (e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{join}$$
$$\text{finalize}$$

... 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 $(0 \leq$ 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 :-)

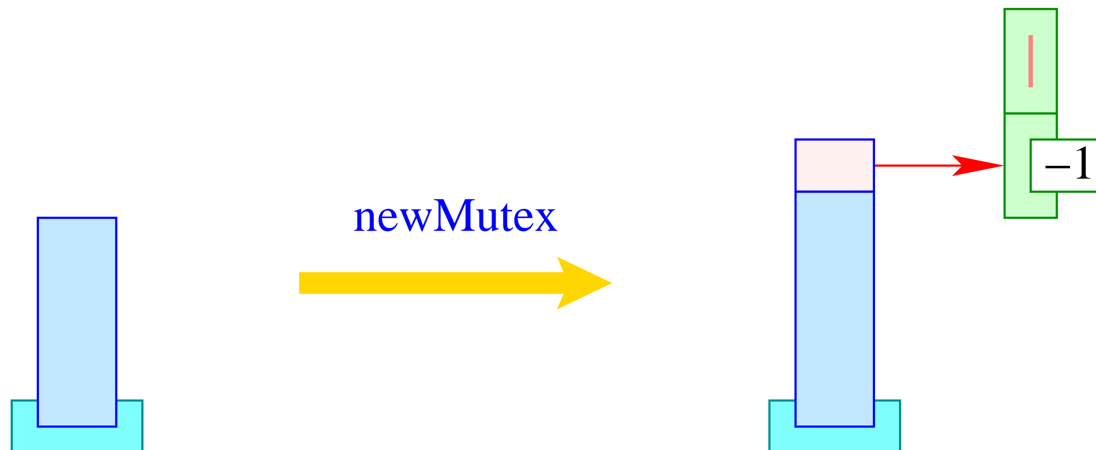
Ein Mutex me besteht aus:

- 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:

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

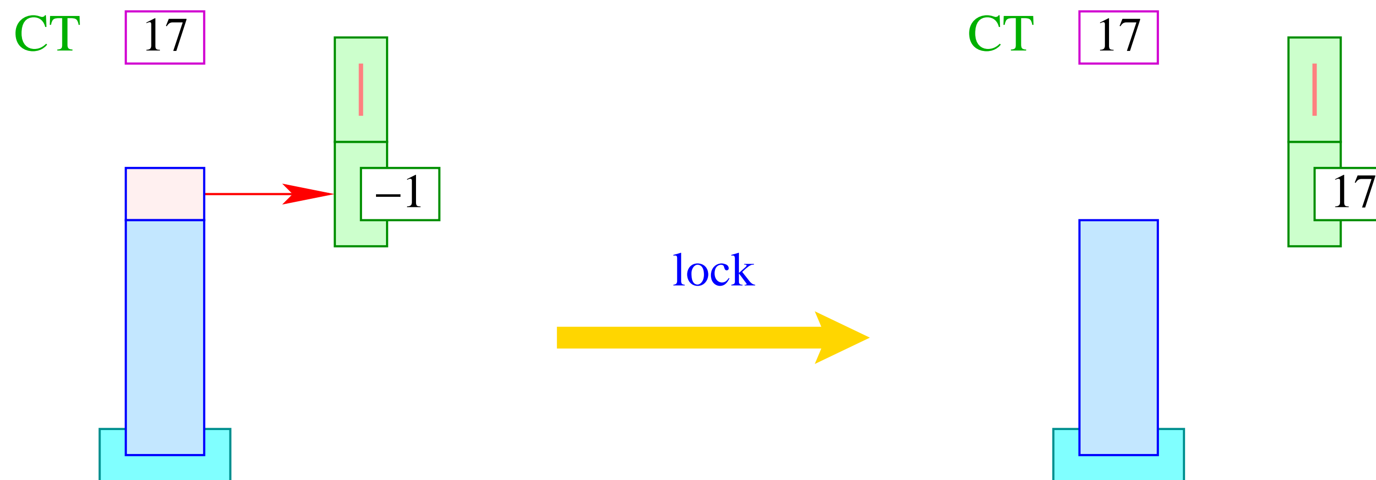
wobei:



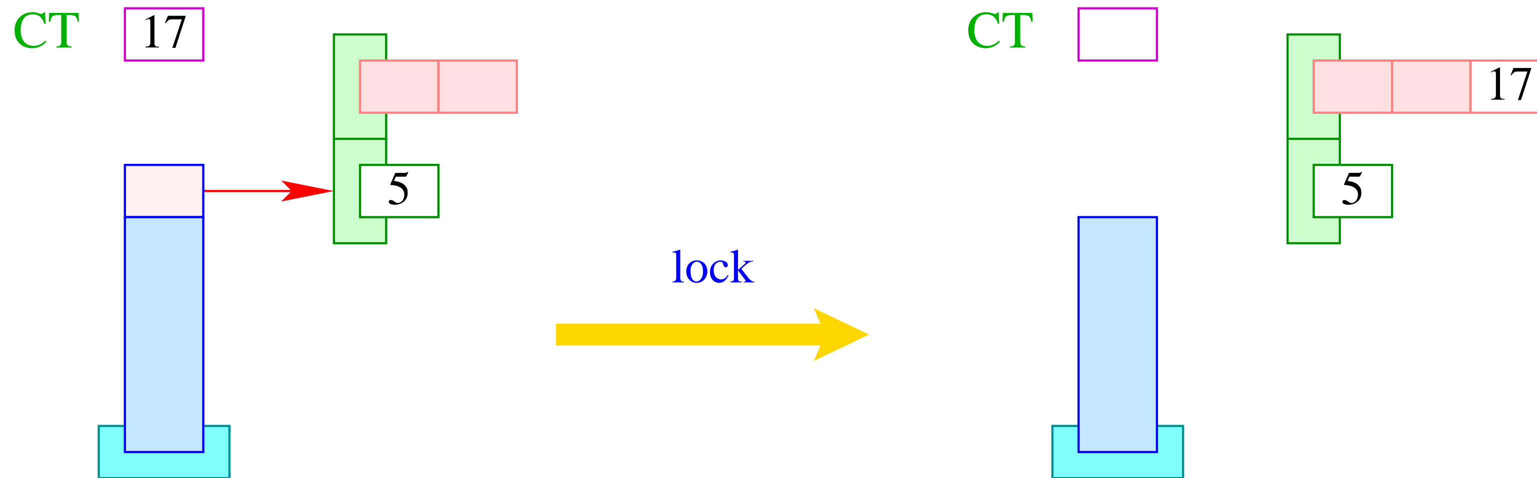
Dann übersetzen wir:

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

wobei:



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

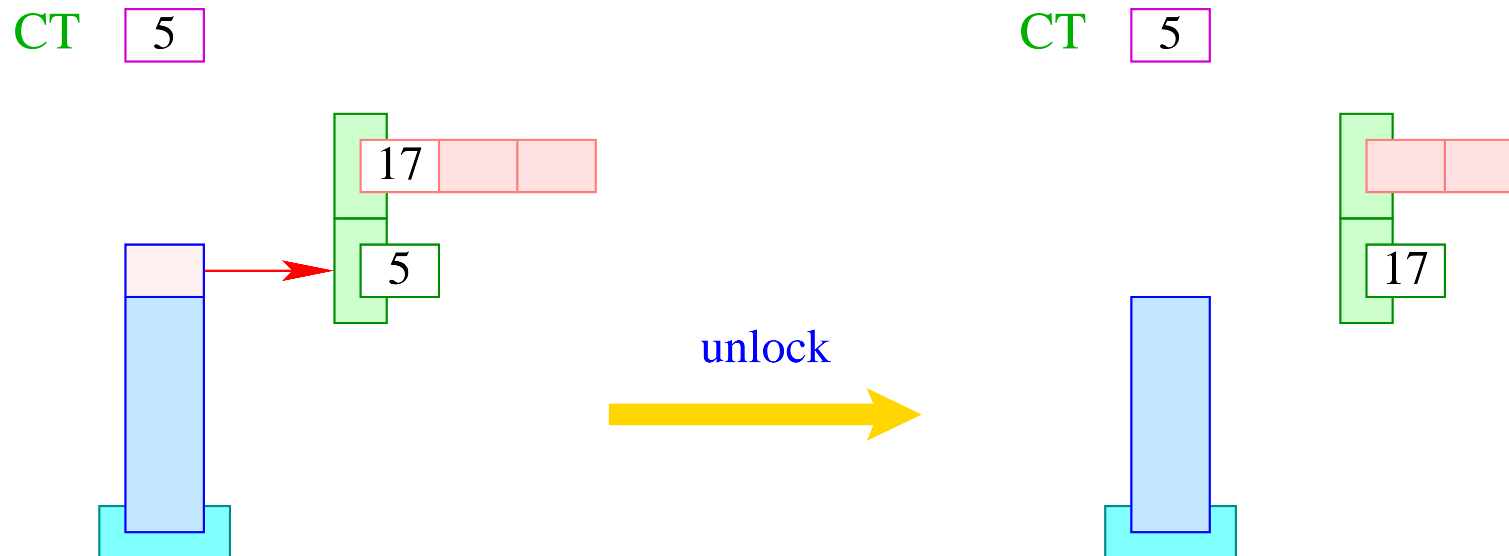


$$\begin{array}{l} \text{if } (S[S[SP]] < 0) \quad S[S[SP--]] = CT; \\ \text{else } \{ \\ \qquad \qquad \text{enqueue } (\, S[SP--]+1, CT\,); \\ \qquad \qquad \text{next}; \\ \qquad \} \end{array}$$

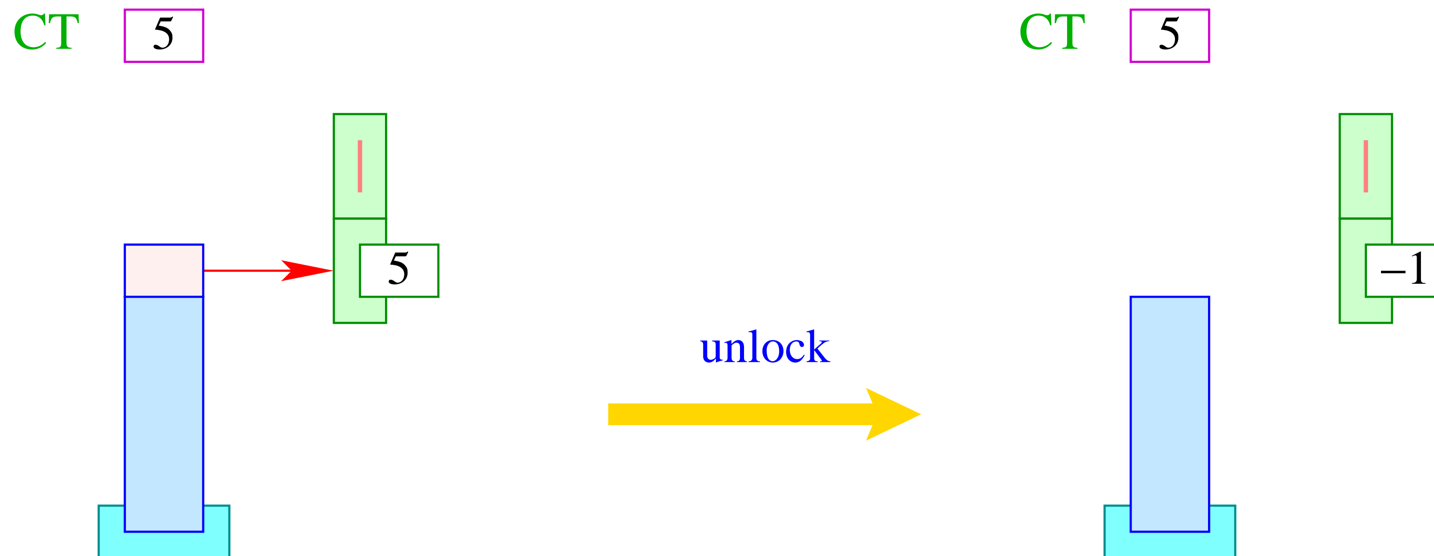
Entsprechend übersetzen wir:

$$\text{code } \mathbf{unlock}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\qquad\qquad\qquad\qquad \text{unlock}$$

wobei:



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

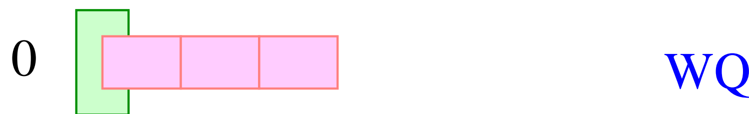


if (S[S[SP]] $\neq$ CT) Error (“Illegal unlock!”);
if ($0 > \text{tid} =$ dequeue (S[SP]+1)) S[S[SP– –]] = –1;
else {
S[S[SP--]] = tid;
enqueue (RQ, tid);
}

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 :-)



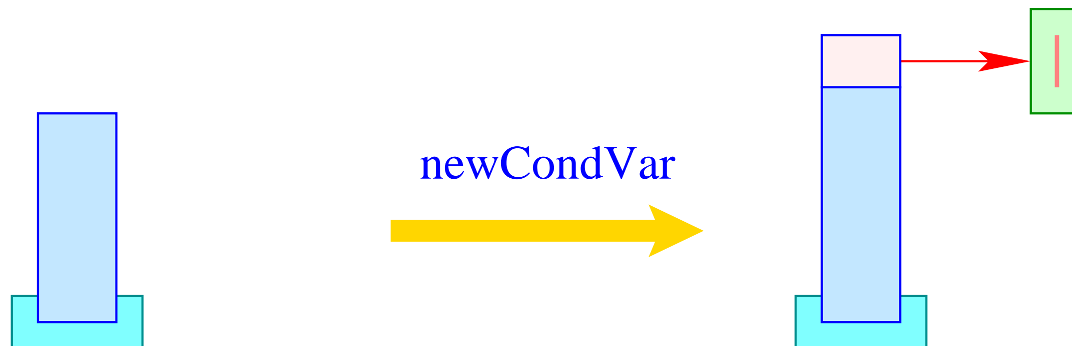
Für Bedingungsvariablen gibt es die Funktionen:

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:

$$\text{code}_R\ \mathbf{newCondVar}\ ()\ \rho \quad = \quad \text{newCondVar}$$

wobei:

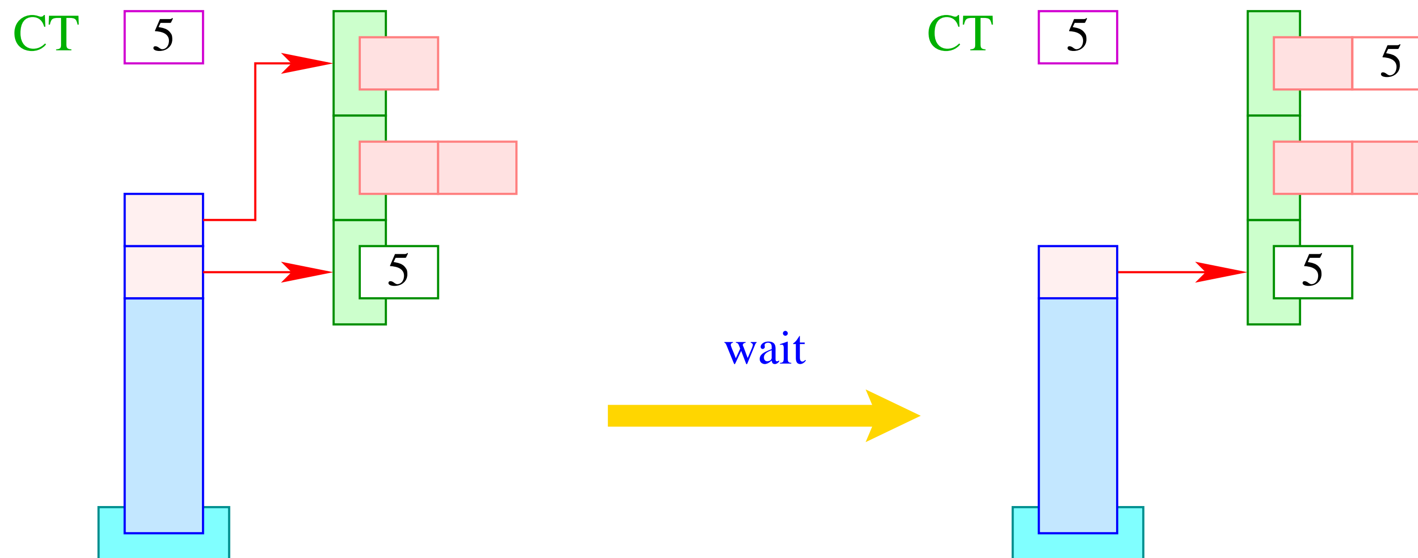


Nach Einreihen in die Warteschlange wird der Mutex wieder frei gegeben. Nach dem Aufwecken muss dieser allerdings neu erworben werden.

Darum übersetzen wir:

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

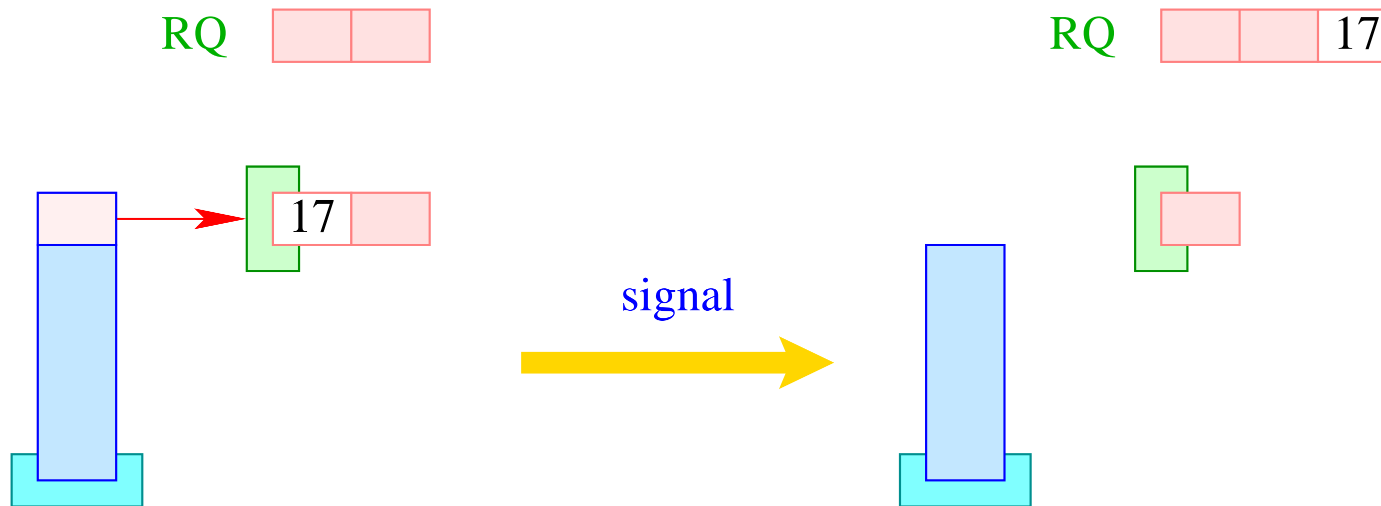
wobei ...



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
```

Die Funktion Down() dekrementiert den Zähler.

Rutscht dieser dadurch ins Negative, wird **wait** aufgerufen:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count– –;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
    }
```

Die Übersetzung liefert für den Rumpf:

```
    alloc 1
    loadr 1
    load
    storer 2
    lock

    loadr 1
    loadc 2
    add
    load
    loadc 1
    sub
    loadr 1
    loadc 2
    add
    store
    loadc 0
    le
    jumpz A
    loadr 2
    loadr 1
    loadc 1
    add
    load
    wait
A:  loadr 2
    unlock
    return
```

Die Funktion Up() inkrementiert den Zähler wieder.

Ist dieser danach noch nicht positiv, gibt es wartende Threads, von denen einer ein Signal erhält:

```
void Up (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
    }
```

Die Übersetzung liefert für den Rumpf:

```
alloc 1      loadc 2      add              loadc 1
loadr 1      add          store            add
load         load         loadc 0          load
storer 2     loadc 1      le               signal
lock         add          jumpz A     A:   loadr 2
             loadr 1                       unlock
loadr 1      loadc 2      loadr 1          return
```

49 Stack-Management

Problem:

- Alle Threads leben in einem gemeinsamen Speicher.
- Jeder Thread benötigt (konzeptuell) einen eigenen Stack.

1. Idee:

Allokiere für jeden neuen Thread einen festen Speicherbereich auf der Halde!

Dann implementieren wir:

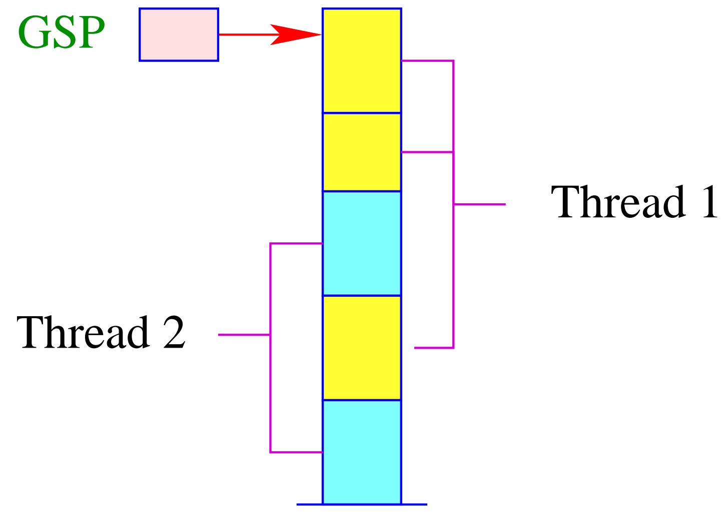
```
void *newStack() { return malloc(M); }
void freeStack(void *adr) { free(adr); }
```

Problem:

- Manche Threads brauchen viel, manche weniger Stack-Space.
- Evt. ist der nötige Platz statisch gar nicht bekannt :-(

2. Idee:

- Verwalte sämtliche Keller zusammen in einem Frame-Heap FH :-)
- Sorge dafür, dass der Platz im Rahmen zumindest ausreicht zum Abarbeiten des aktuellen Funktionsaufrufs.
- Ein globaler Stack-Pointer GSP gibt an, wieviel Platz bereits vergeben ist...



Allokation und De-Allokation eines Frames erfolgt mittels Laufzeitfunktionen:

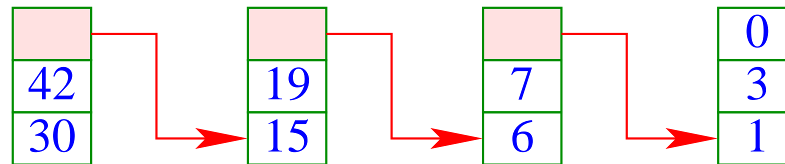
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

void freeFrame(int sp, int size);
```

Achtung:

Der frei zu gebende Block kann im Innern des Stacks liegen :-(

Wir verwalten eine Liste der freigegebenen Abschnitte des Stacks :-)

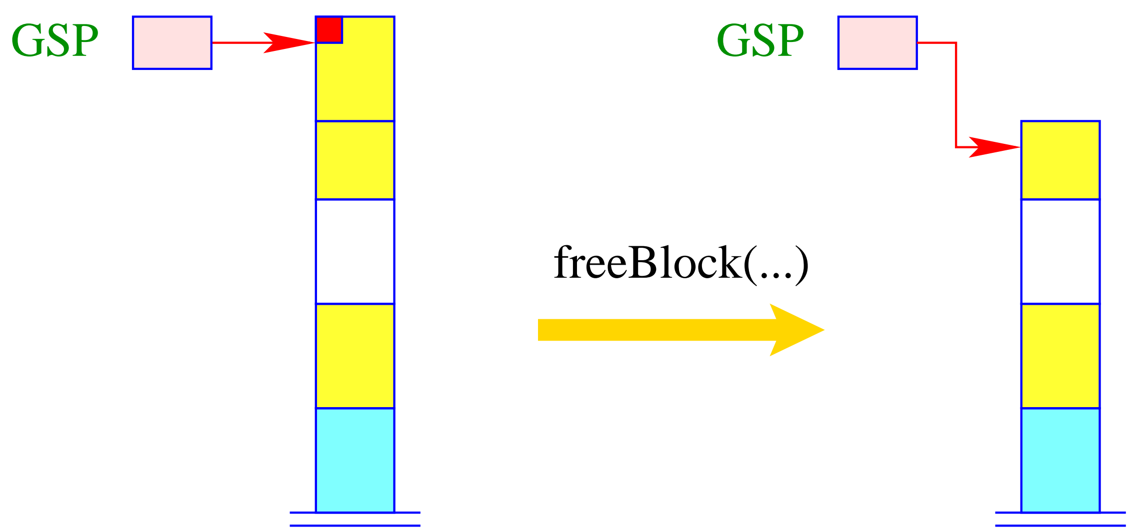


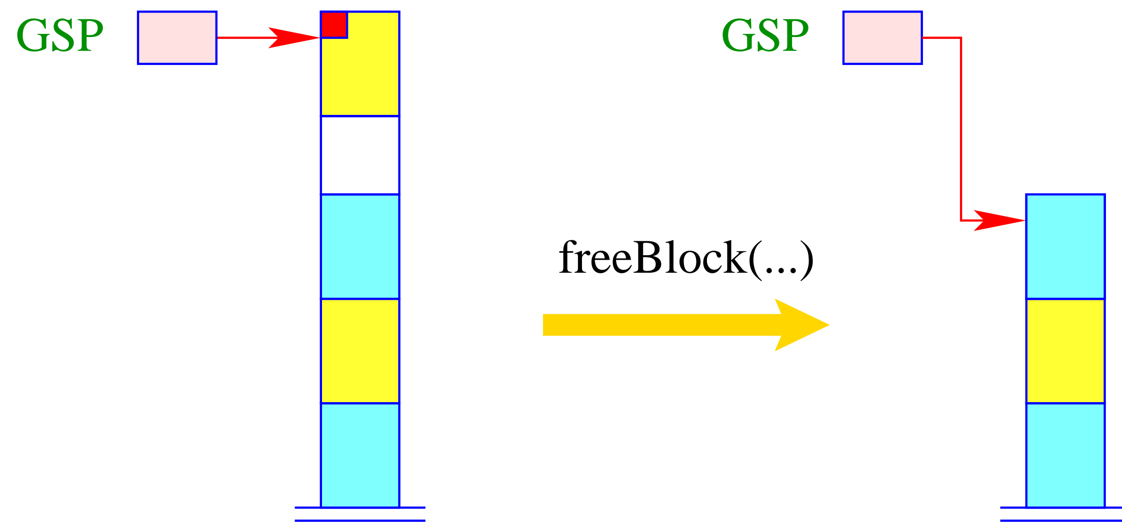
Diese Liste unterstützt eine Funktion

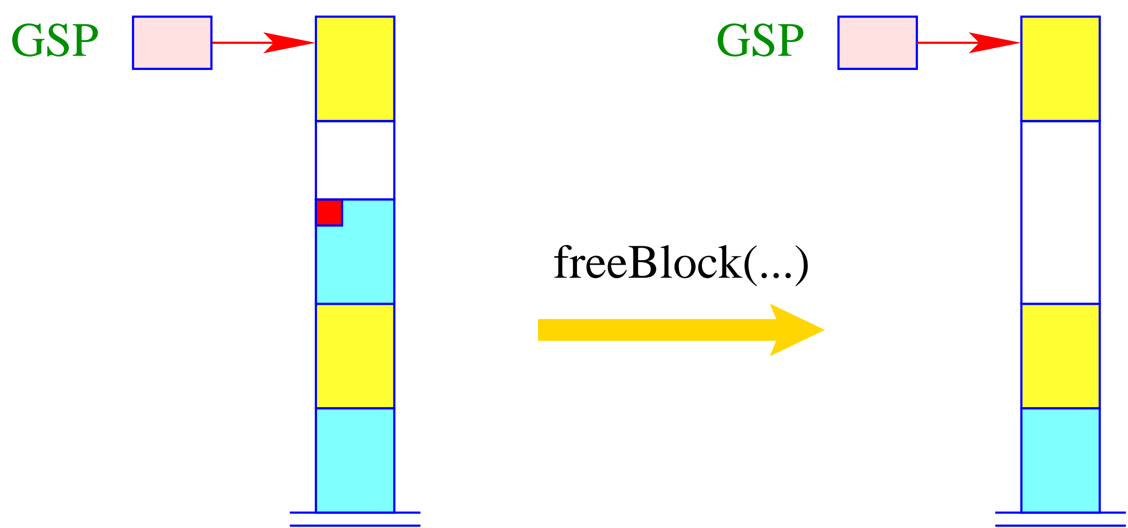
```
void insertBlock(int max, int min)
```

die es gestattet, einzelne Blocks frei zugeben.

- Liegt der Block am oberen Ende des Stacks geben wir ihn sofort frei;
- ... Wie den darunter liegenden Abschnitt – falls dieser bereits de-allokiert ist.
- Liegt er im Innern, verschmelzen wir ihn mit angrenzenden freien Blöcken:


GSP
GSP
freeBlock(...)


GSP
GSP
freeBlock(...)


GSP
GSP
freeBlock(...)

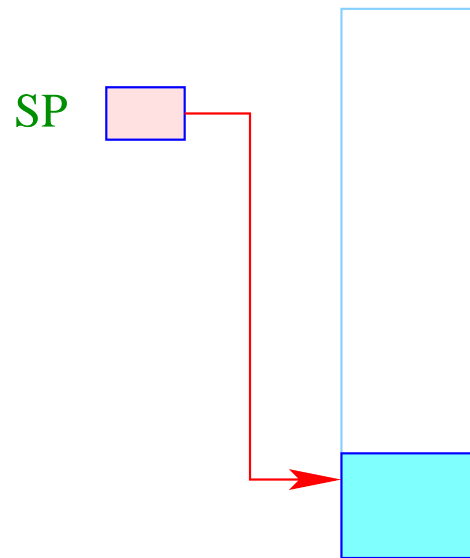
Ansatz:

Wir allokieren einen neuen Block für jeden Funktions-Aufruf ...

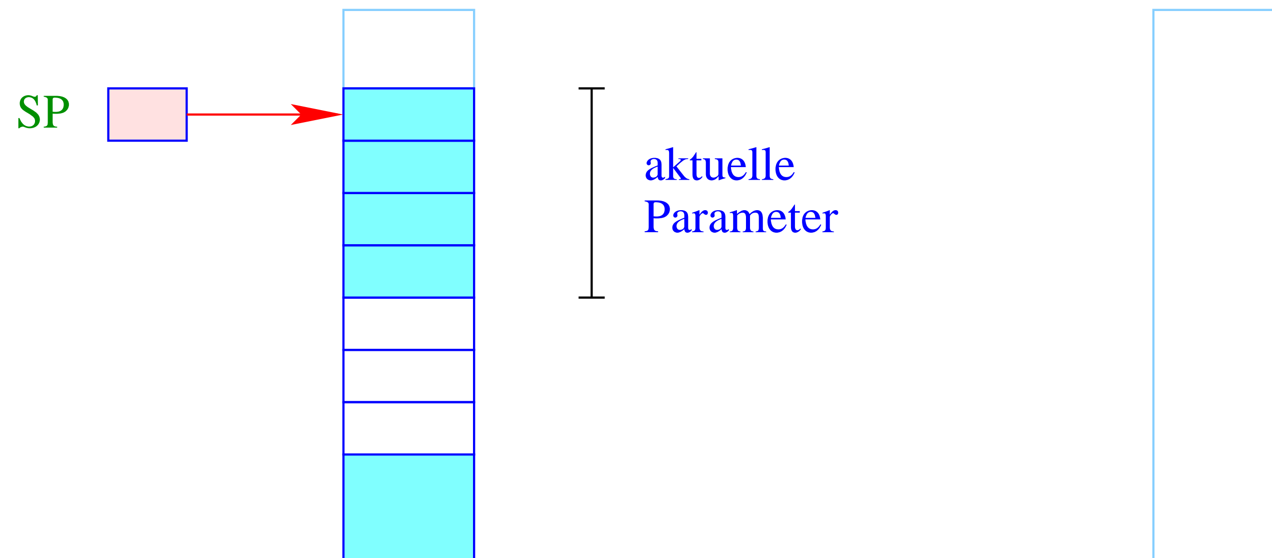
Problem:

Bei Anforderung des neuen Blocks vor dem Aufruf ist der Speicherbedarf der aufgerufenen Funktion noch gar nicht bekannt :-(

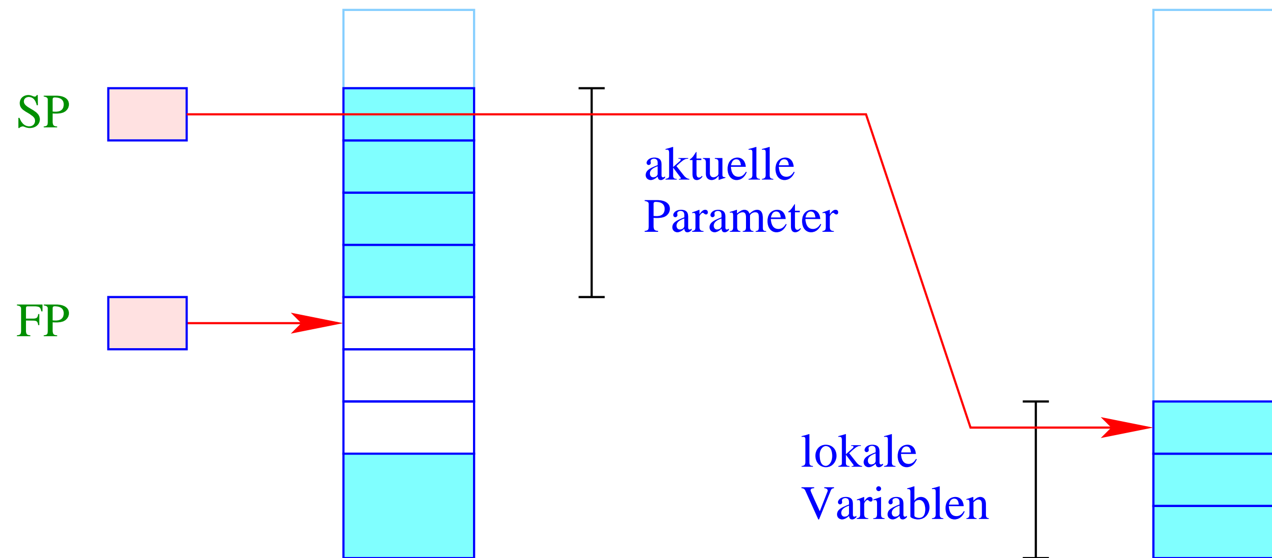
$\Longrightarrow$ Wir können den neuen Block erst bei Betreten des Funktions-Rumpfs anfordern!



Organisatorische Zellen wie aktuelle Parameter müssen noch im alten Block angelegt werden ...



Bei Betreten der neuen Funktion allokieren wir auch den neuen Block ...



Insbesondere liegen jetzt die lokalen Variablen im neuen Block ...

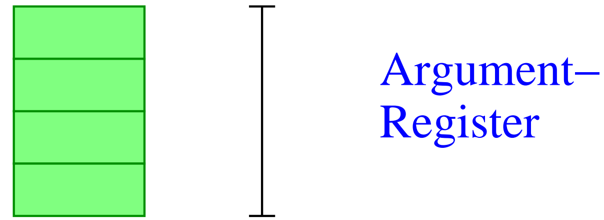
Wir adressieren ...

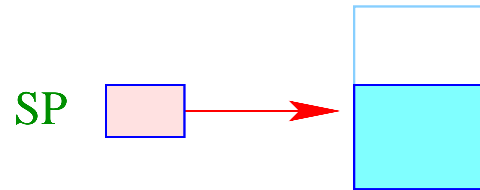
- die formalen Parameter relativ zum Frame-Pointer;
- die lokalen Variablen relativ zum Stack-Pointer :-)

Wir müssen die gesamte Code-Erzeugung umstellen ... :-(

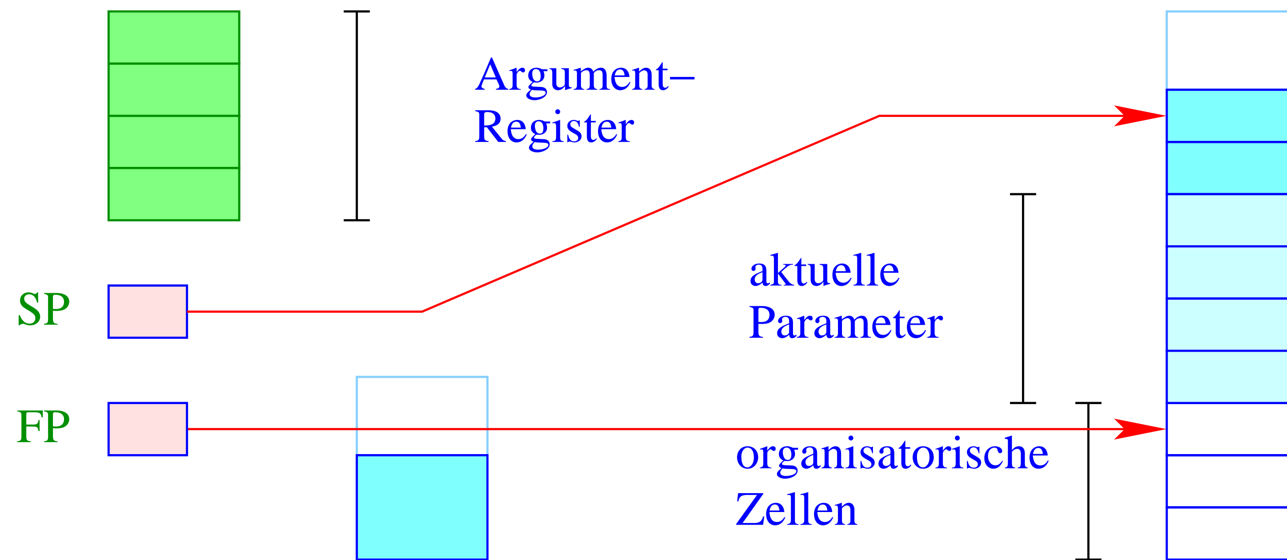
Ausweg:

Übergabe von Parametern in Registern ... :-)

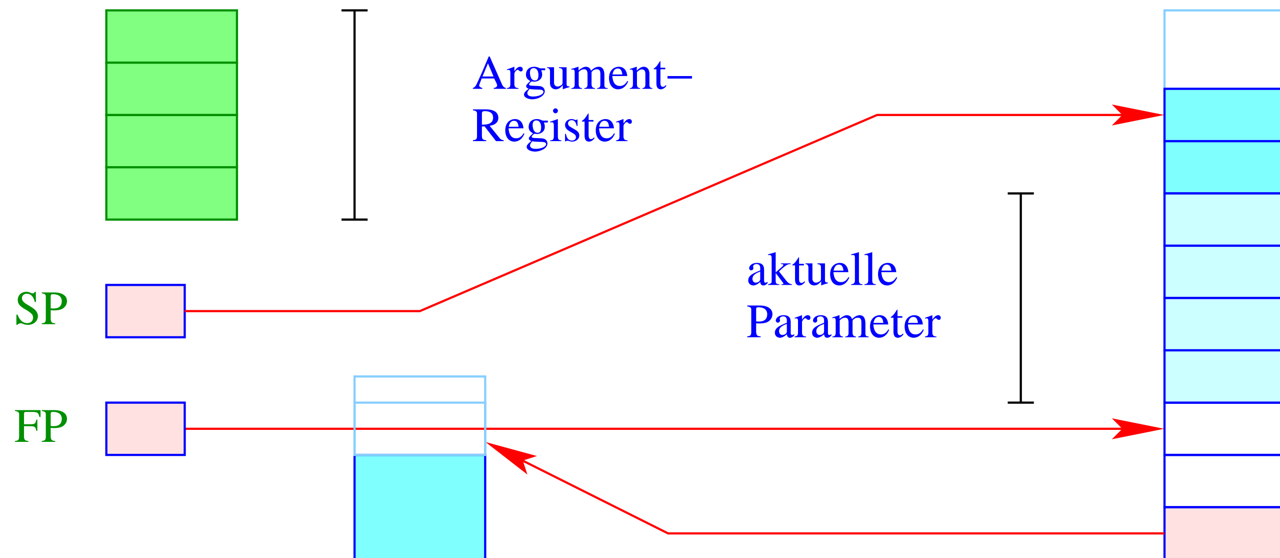




Die Werte der aktuellen Parameter werden vor Anlegen des neuen Keller-Rahmens ermittelt.



Der gesamte Rahmen wird im neuen Block angelegt – inclusive Platz für die aktuellen Parameter.



Im neuen Block müssen wir allerdings uns auch den alten SP (evt. +1) merken, damit das Ergebnis korrekt zurück geliefert werden kann ...

3. Idee: Hybrid-Lösung

- Für die ersten k Threads lege jeweils einen eigenen Speicherbereich an!
- Für alle weiteren benutze reihum einen der bereits vorhandenen ...

- Für wenige Threads extrem einfach und effizient;
- Für viele Threads amortisierte Speicher-Ausnutzung ...