# 37 Extension: The Cut Operator

Realistic Prolog additionally provides an operator “!” (cut) which explicitly allows to prune the search space of backtracking.

Example:

$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

Once the queries before the cut have succeeded, the choice is committed:

Backtracking will return only to backtrack points preceding the call to the left-hand side ...

## The Basic Idea:

- We restore the oldBP from our current stack frame;
- We pop all stack frames on top of the local variables.

Accordingly, we translate the cut into the sequence:

```
prune
pushenv m
```

where m is the number of (still used) local variables of the clause.

## Example:

Consider our example:

$$\begin{aligned} \mathsf{branch}(X,Y) &\leftarrow \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) &\leftarrow \mathsf{q}_2(X,Y) \end{aligned}$$

We obtain:

```
setbtp    A:  pushenv 2    C:  prune       lastmark        B:  pushenv 2
try A         mark C           pushenv 2   putref 1            putref 2
delbtp        putref 1                     putref 2            putref 2
jump B        call p/1                     lastcall q1/2 2     move 2 2
                                                               jump q2/2
```

## Example:

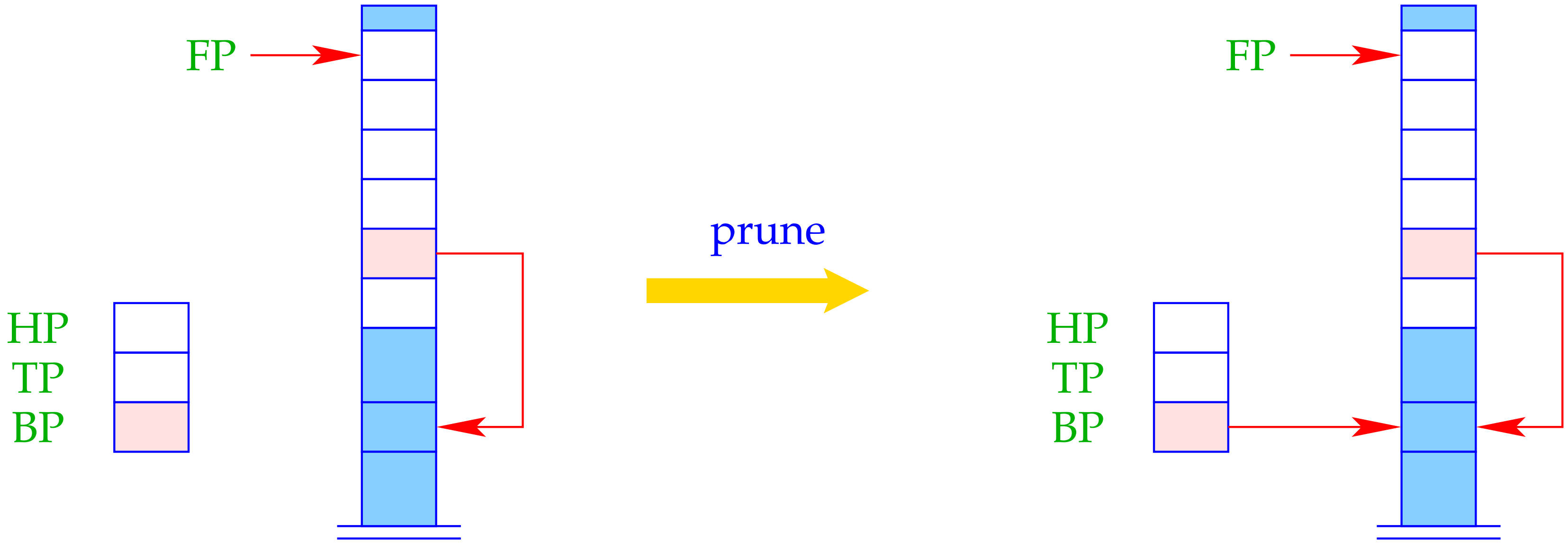
Consider our example:

$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

In fact, an optimized translation even yields here:

```
setbtp    A:   pushenv 2   C:   prune       putref 1     B:   pushenv 2
try A          mark C           pushenv 2   putref 2          putref 1
delbtp         putref 1                     move 2 2          putref 2
jump B         call p/1                     jump q1/2         move 2 2
                                                              jump q2/2
```

The new instruction prune simply restores the backtrack pointer:



BP = BPold;

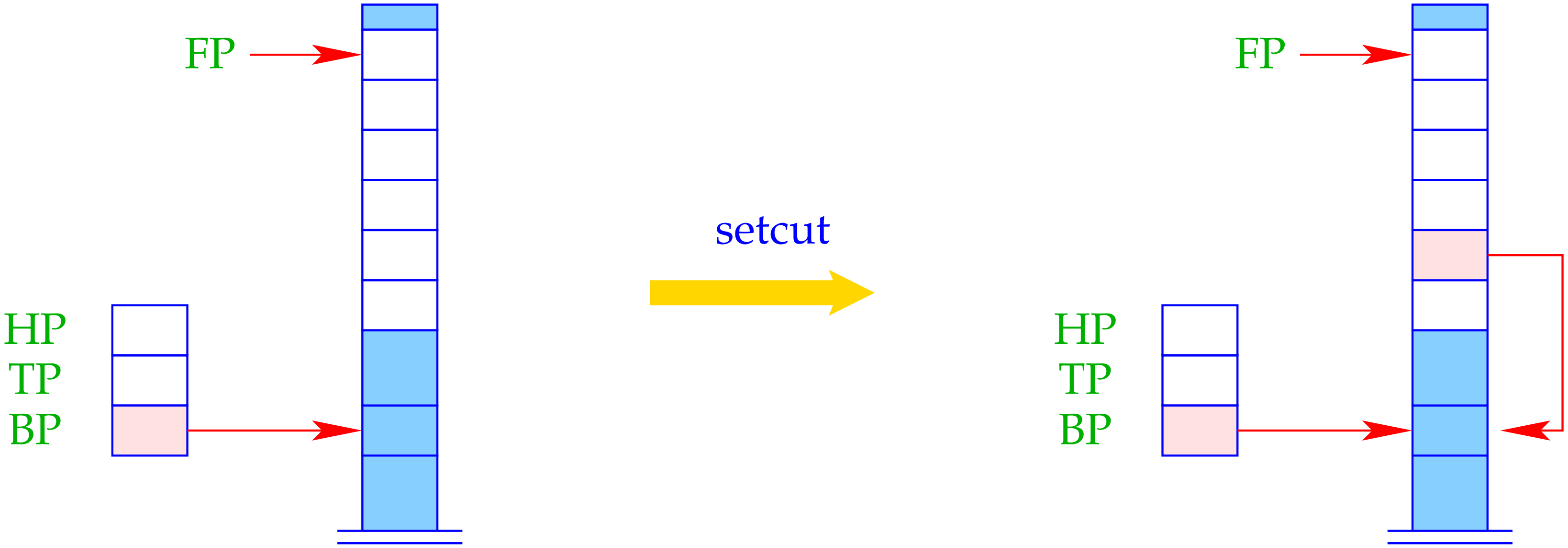
## Problem:

If a clause is single, then (at least so far ;-) we have not stored the old BP inside the stack frame :-(

$$\Longrightarrow$$

For the cut to work also with single-clause predicates or try chains of length 1, we insert an extra instruction setcut before the clausal code (or the jump):

The instruction setcut just stores the current value of BP:



BPold = BP;

## The Final Example: Negation by Failure

The predicate notP should succeed whenever p fails (and vice versa :-)

$$\begin{array}{lll} \mathsf{notP}(X) & \leftarrow & \mathsf{p}(X), !, \mathsf{fail} \\ \mathsf{notP}(X) & \leftarrow & \end{array}$$

where the goal fail never succeeds. Then we obtain for notP :

```
     setbtp     A:   pushenv 1   C:   prune        B:   pushenv 1
     try A           mark C           pushenv 1         popenv
     delbtp          putref 1         fail
     jump B          call p/1         popenv
```

# 38 Garbage Collection

- Both during execution of a MaMa- as well as a WiM-programs, it may happen that some objects can no longer be reached through references.
- Obviously, they cannot affect the further program execution. Therefore, these objects are called garbage.
- Their storage space should be freed and reused for the creation of other objects.

Warning:

The WiM provides some kind of heap de-allocation. This, however, only frees the storage of failed alternatives !!!

## Operation of a stop-and-copy-Collector:

- Division of the heap into two parts, the to-space and the from-space — which, after each collection flip their roles.
- Allocation with new in the current from-space.
- In case of memory exhaustion, call of the collector.

## The Phases of the Collection:

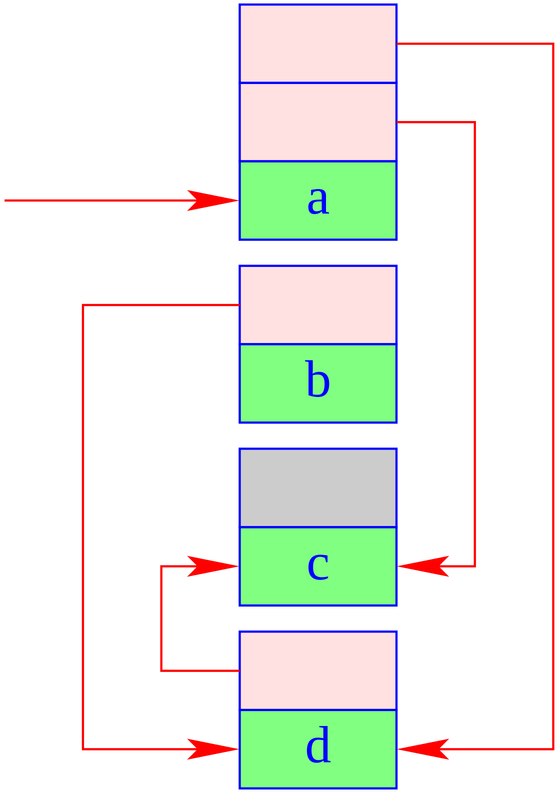
1. Marking of all reachable objects in the from-space.
2. Copying of all marked objects into the to-space.
3. Correction of references.
4. Exchange of from-space and to-space.

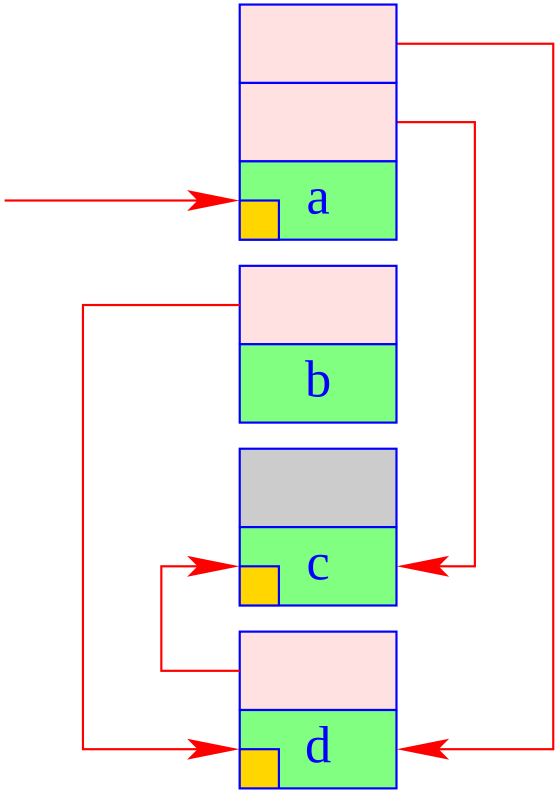
**(1)** **Mark:** Detection of live objects:

- all references in the stack point to live objects;
- every reference of a live object points to a live object.

$\Longrightarrow$

Graph Reachability


a
b
c
d

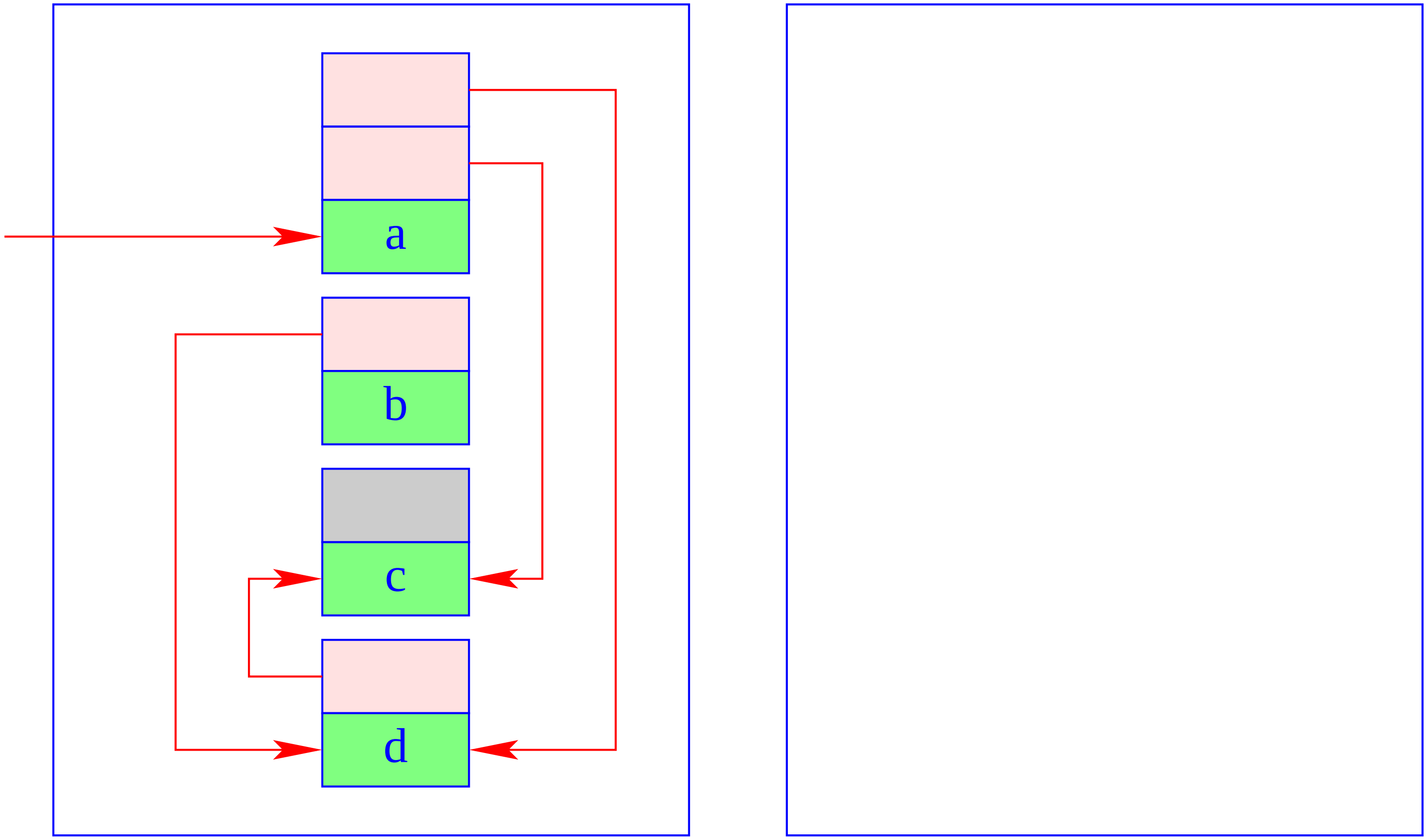

a
b
c
d

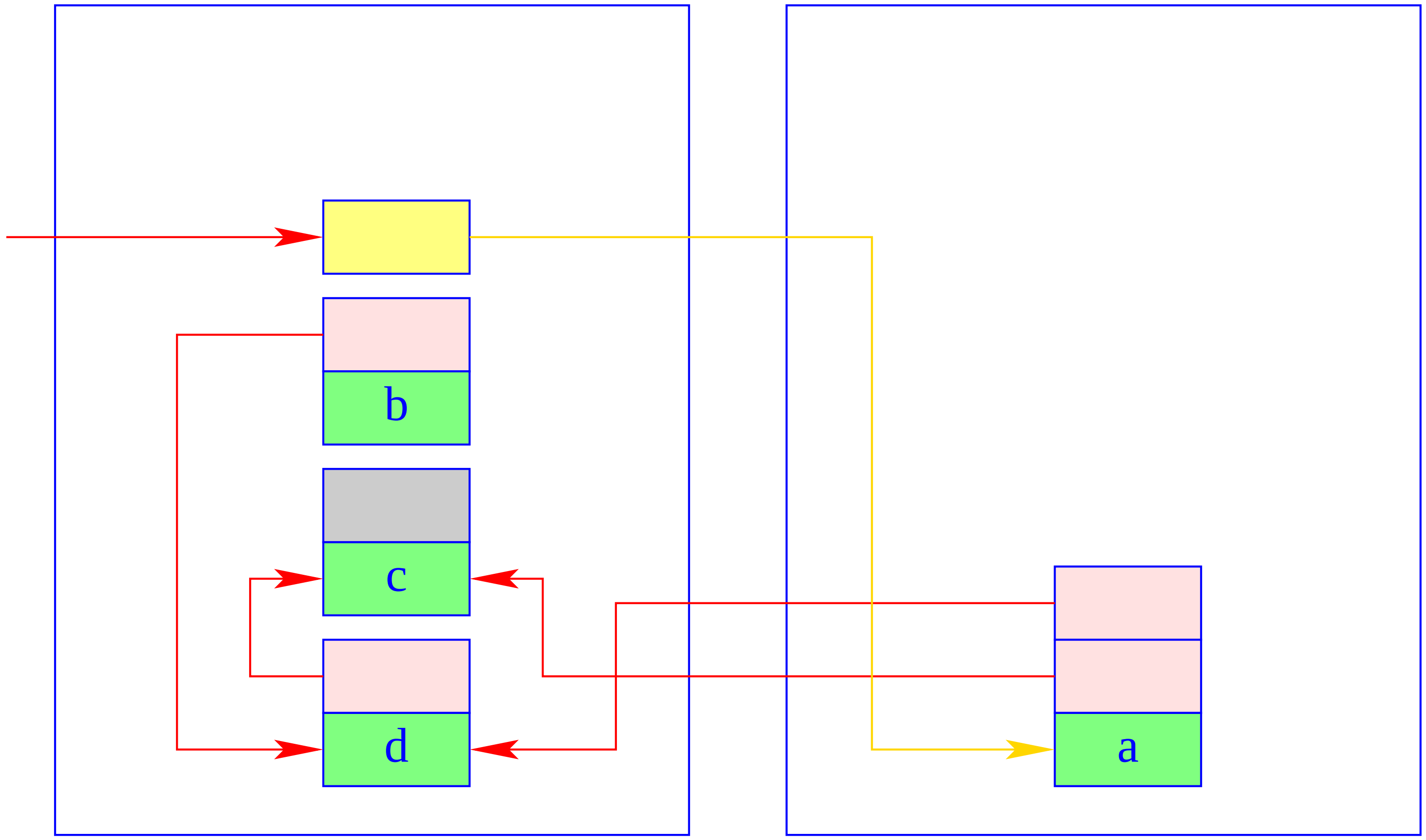
**(2) Copy:** Copying of all live objects from the current from-space into the current to-space. This means for every detected object:

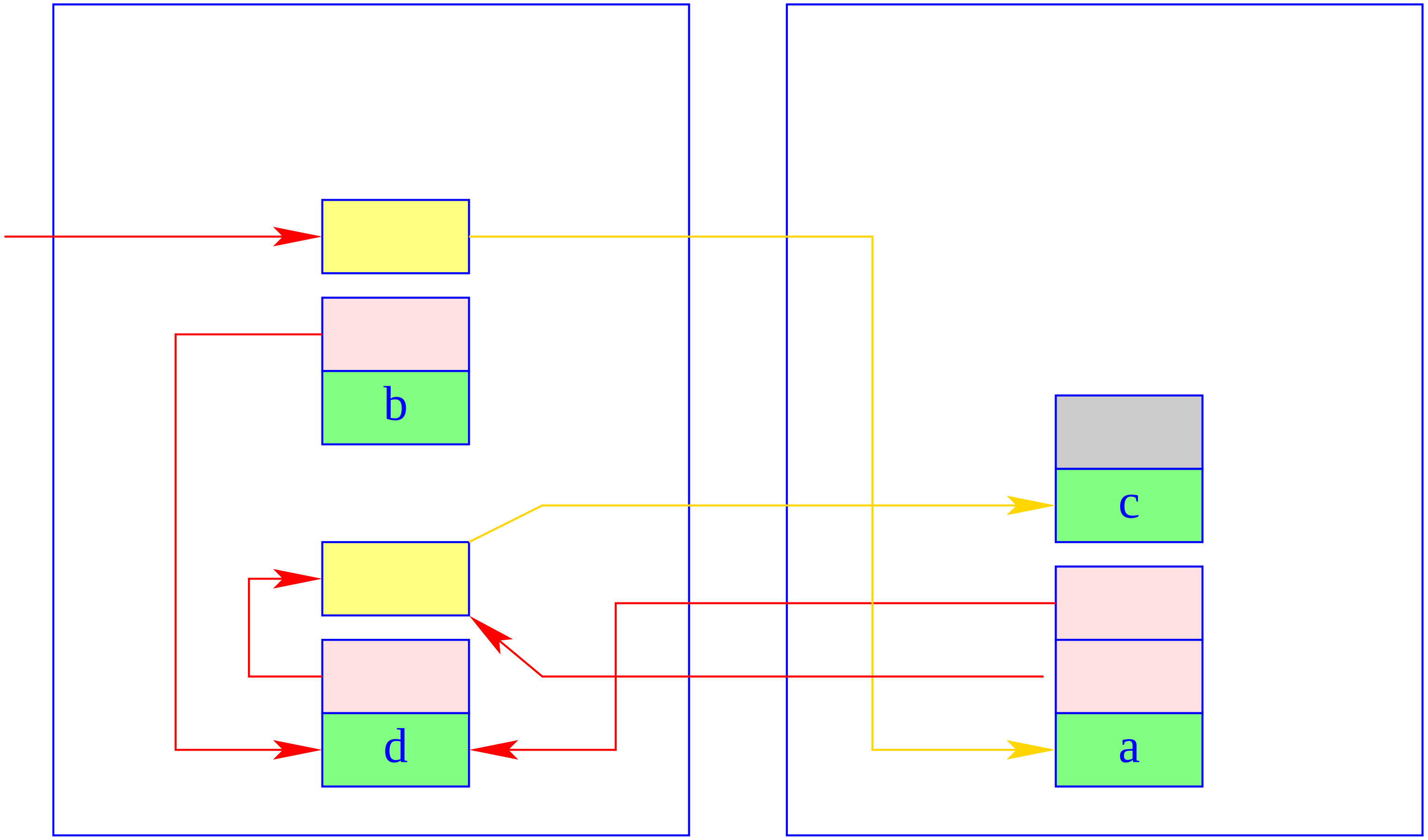
- Copying the object;
- Storing a forward reference to the new place at the old place :-)

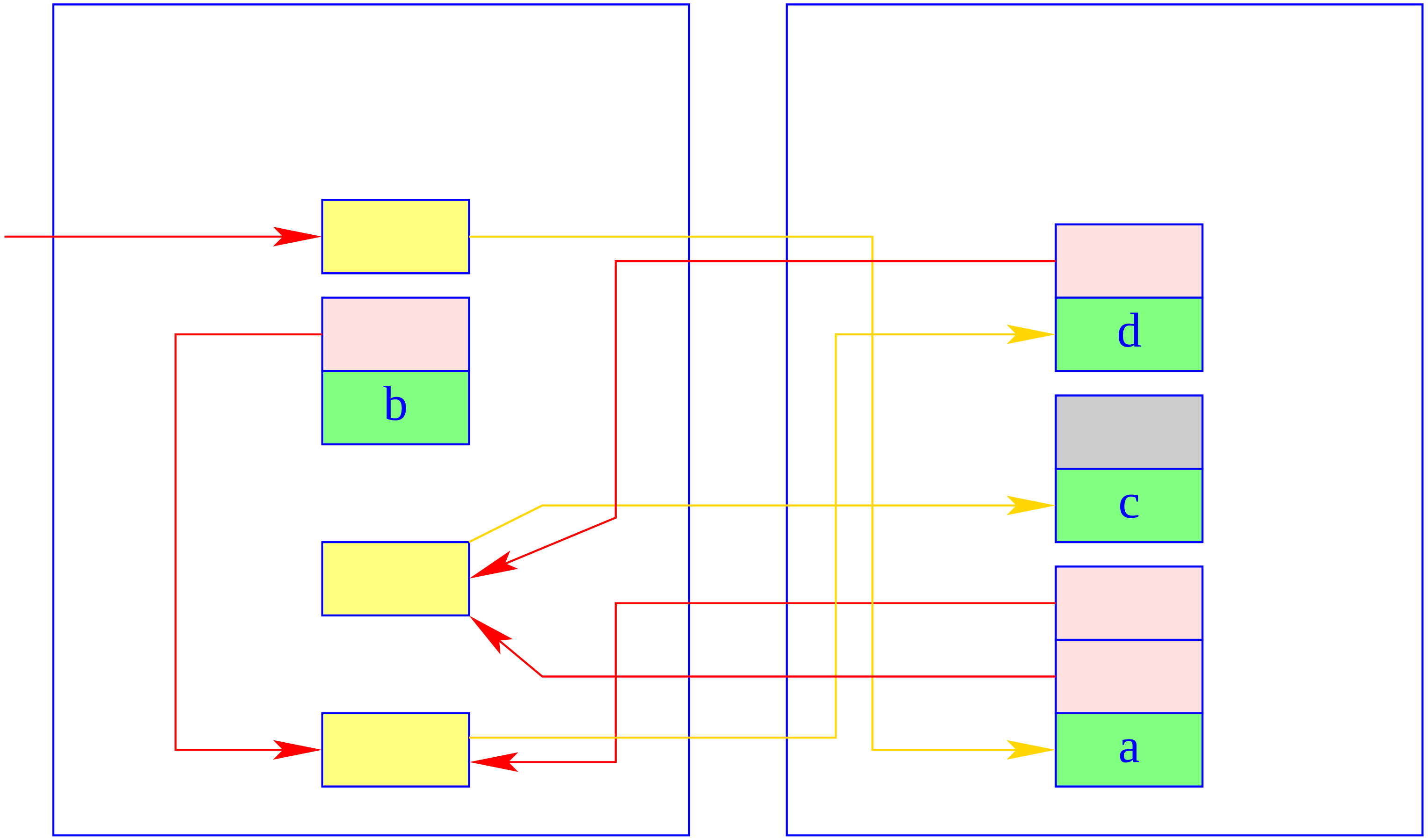
$$\Longrightarrow$$

all references of the copied objects point to the forward references in the from-space.

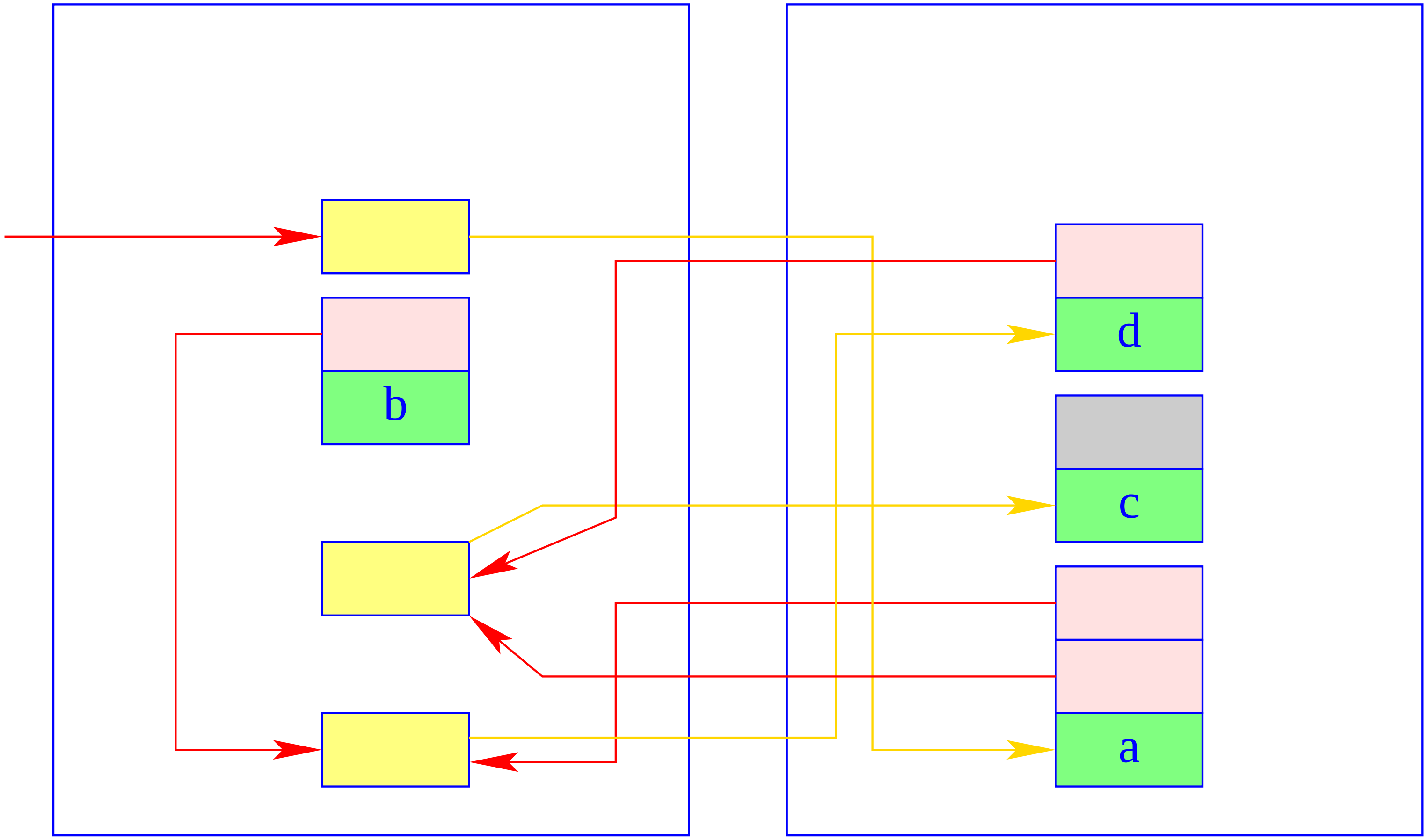

a
b
c
d

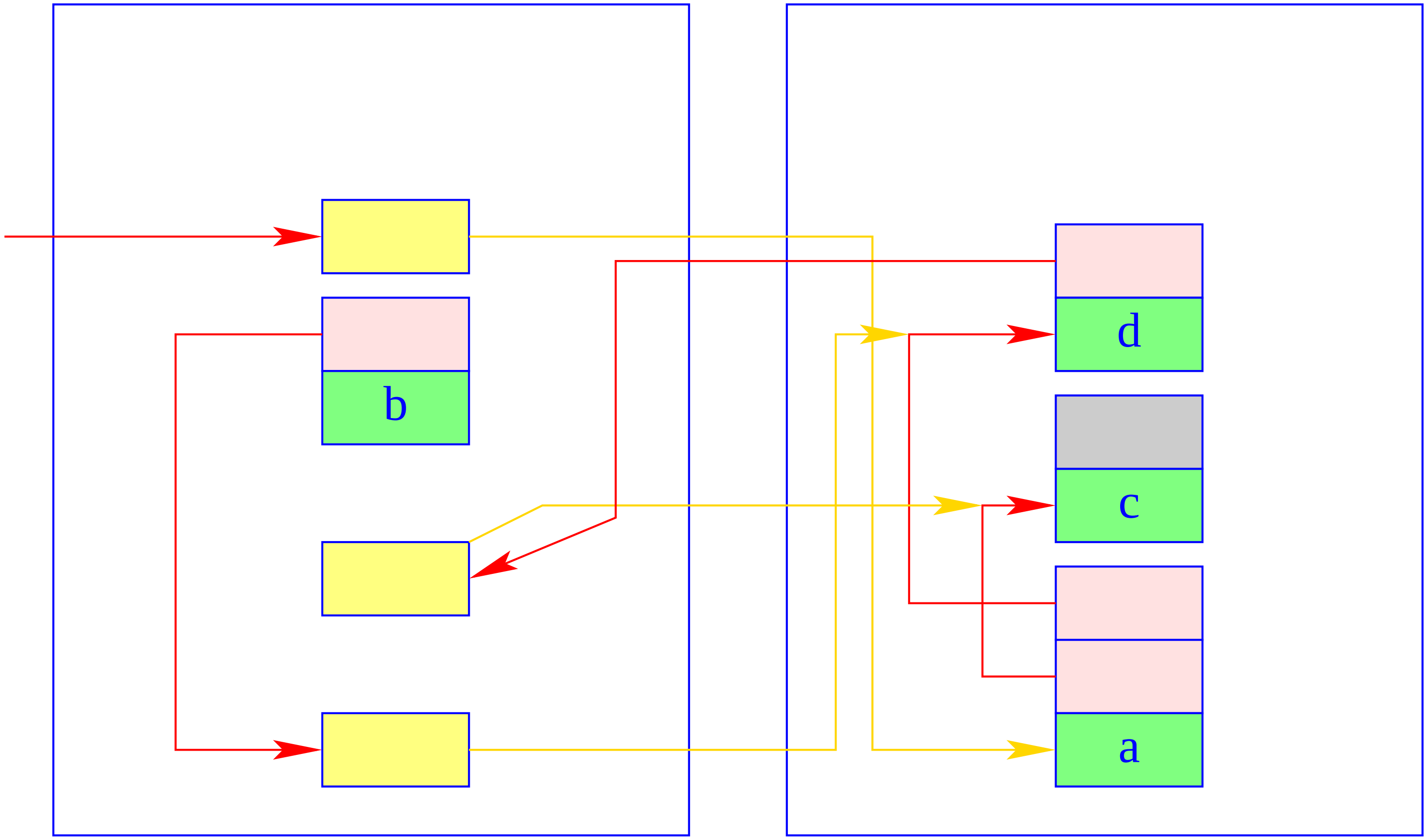

b
c
d
a

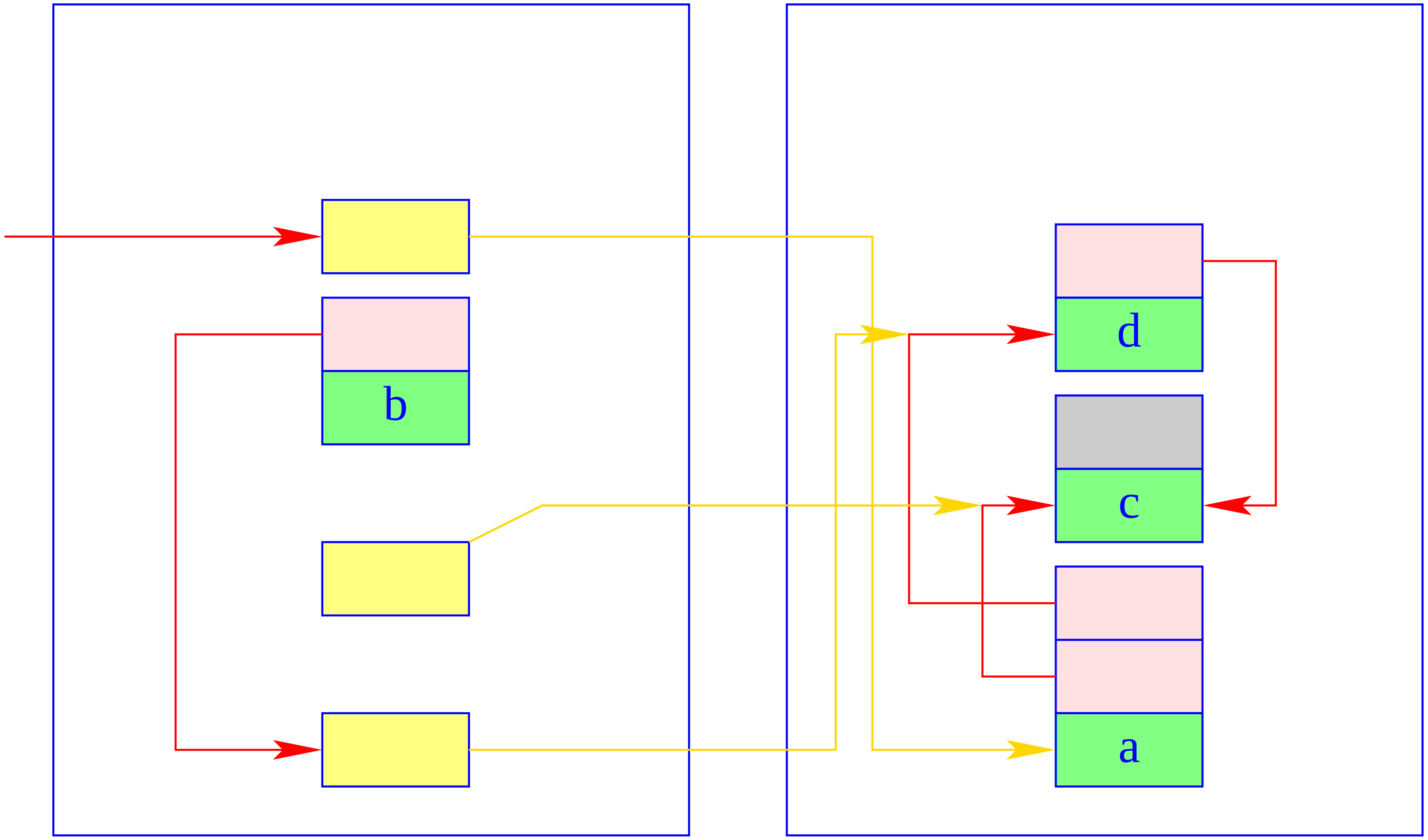

b
c
d
a

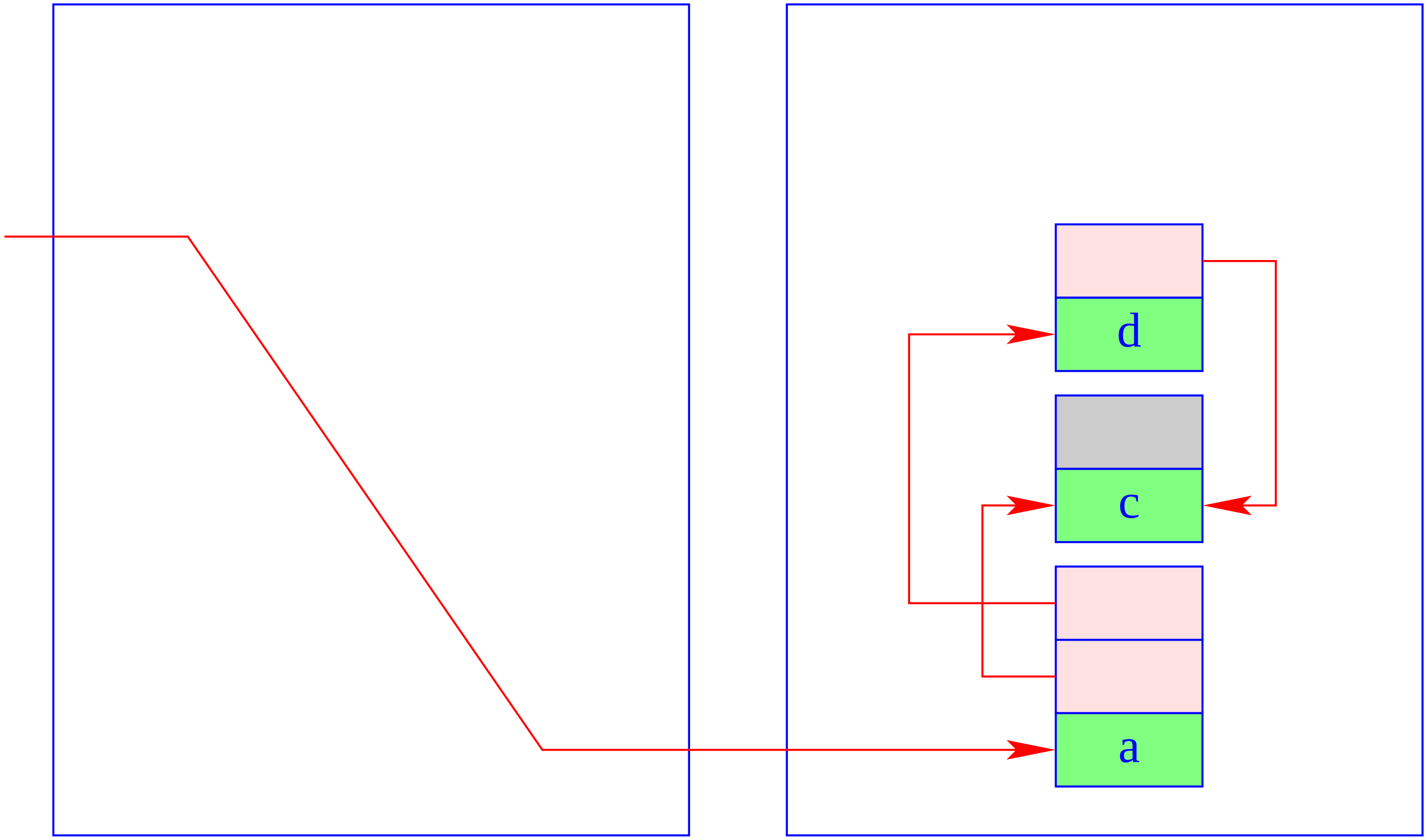

b
d
c
a

(3) Traversing of the to-space in order to correct the references.

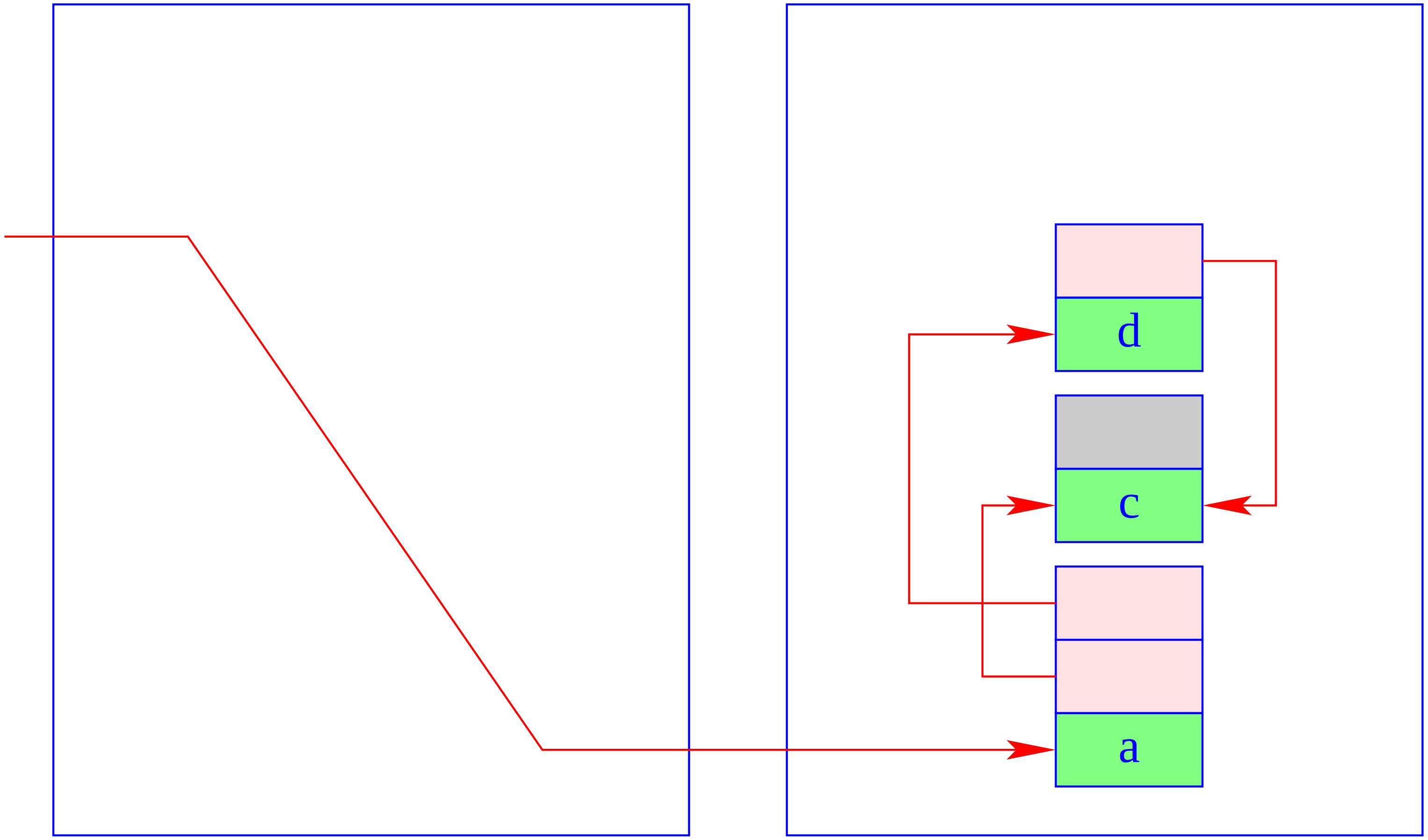


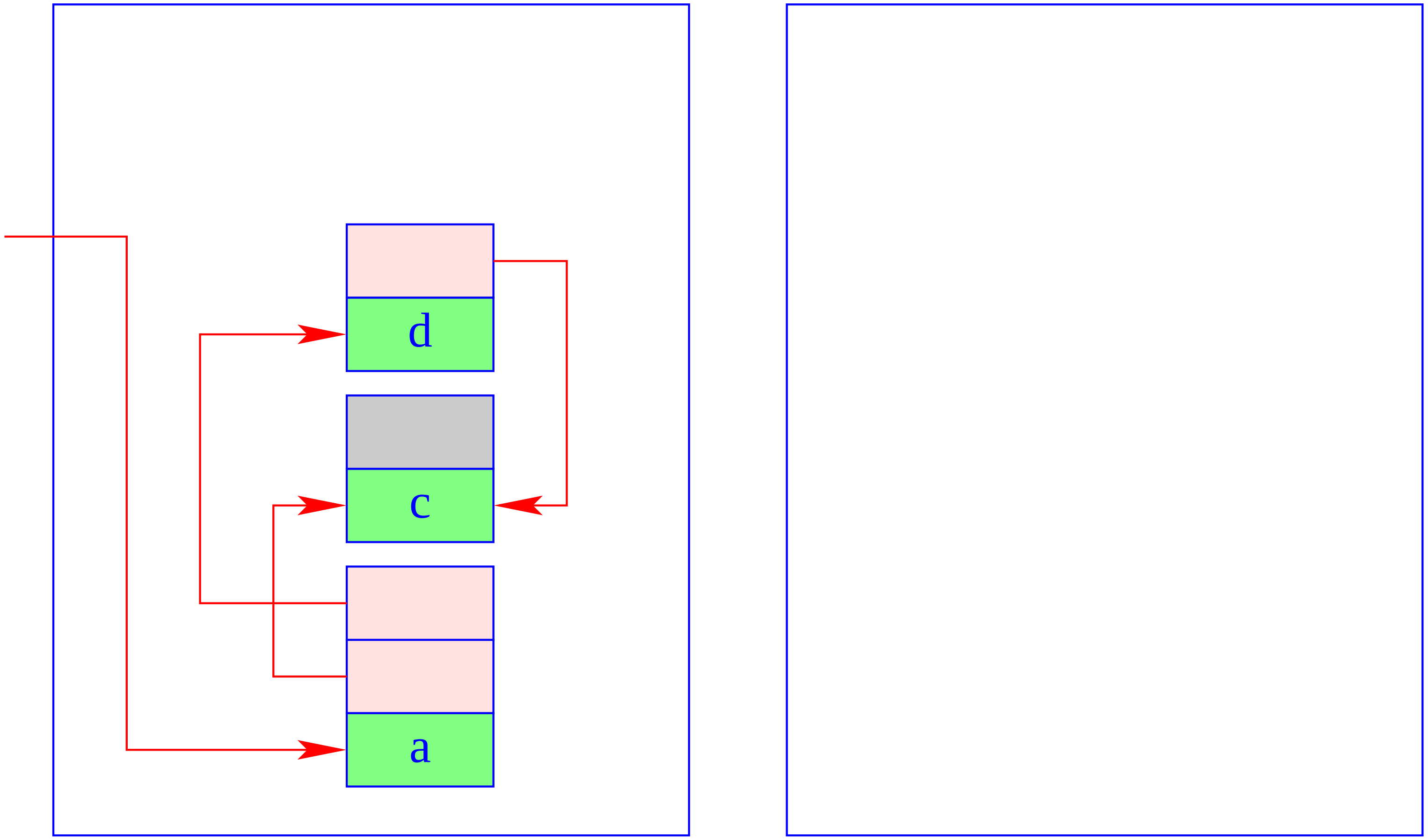

b
d
c
a


b
d
c
a


d
c
a

(4) Exchange of to-space and from-space.



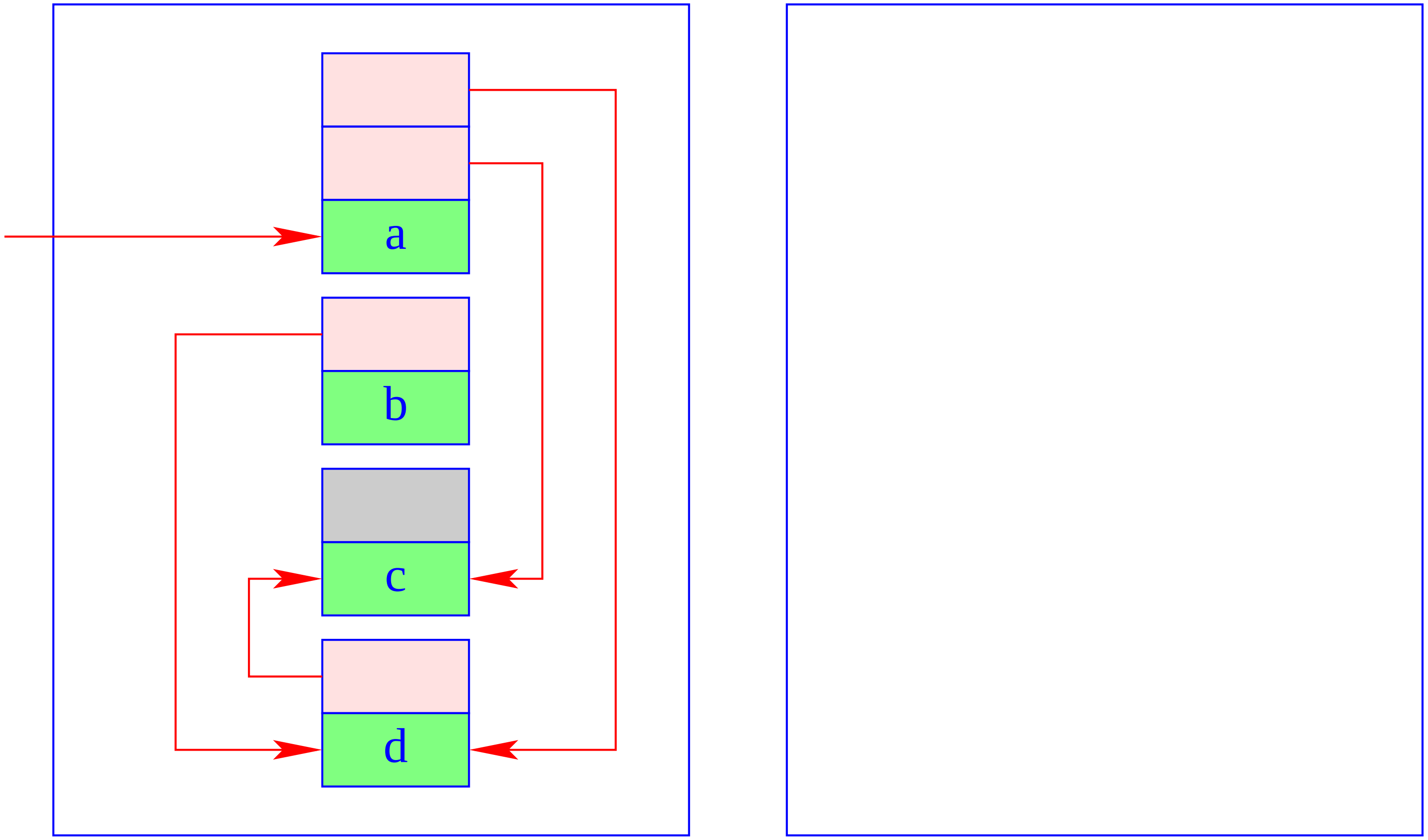

d
c
a

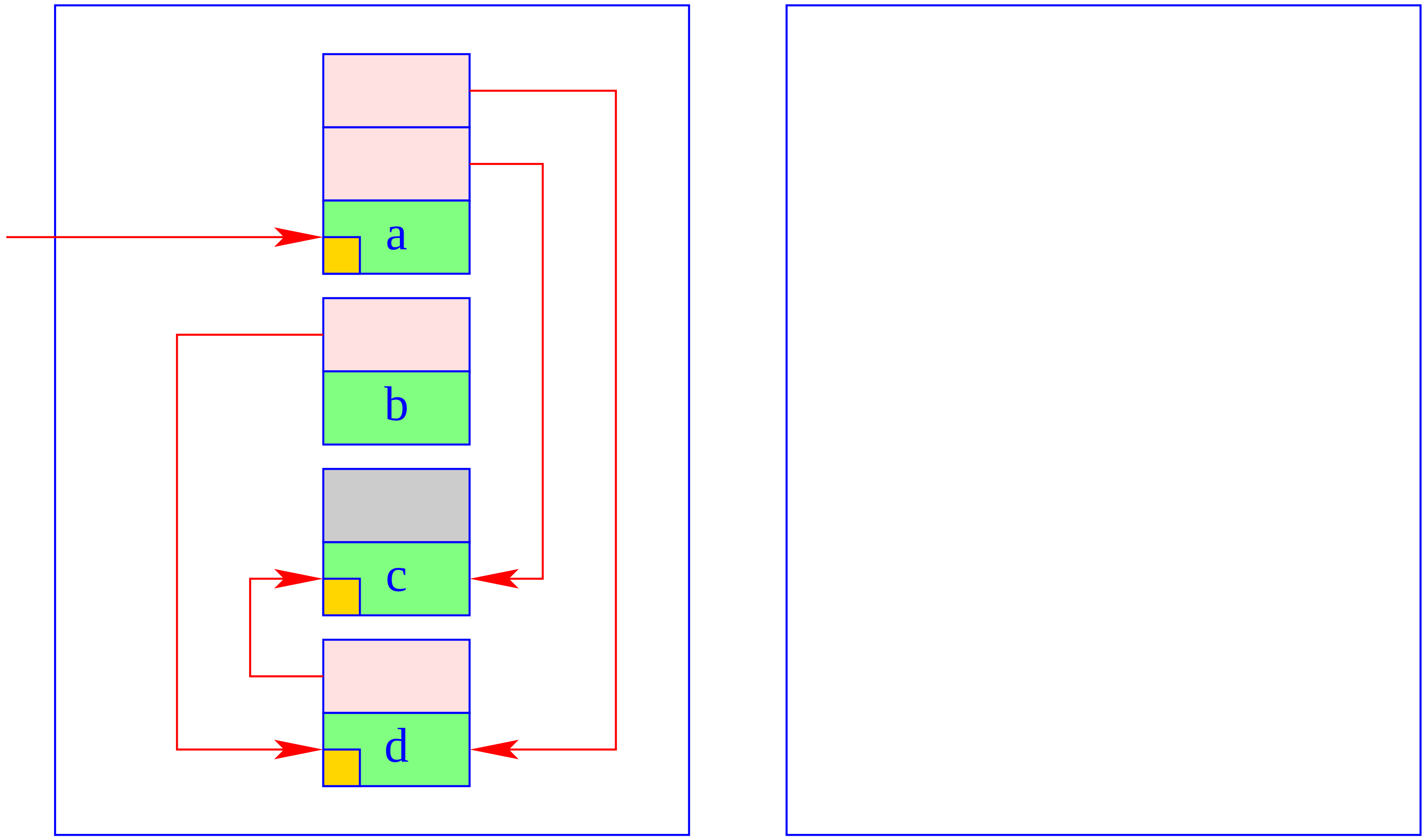
## Warning:

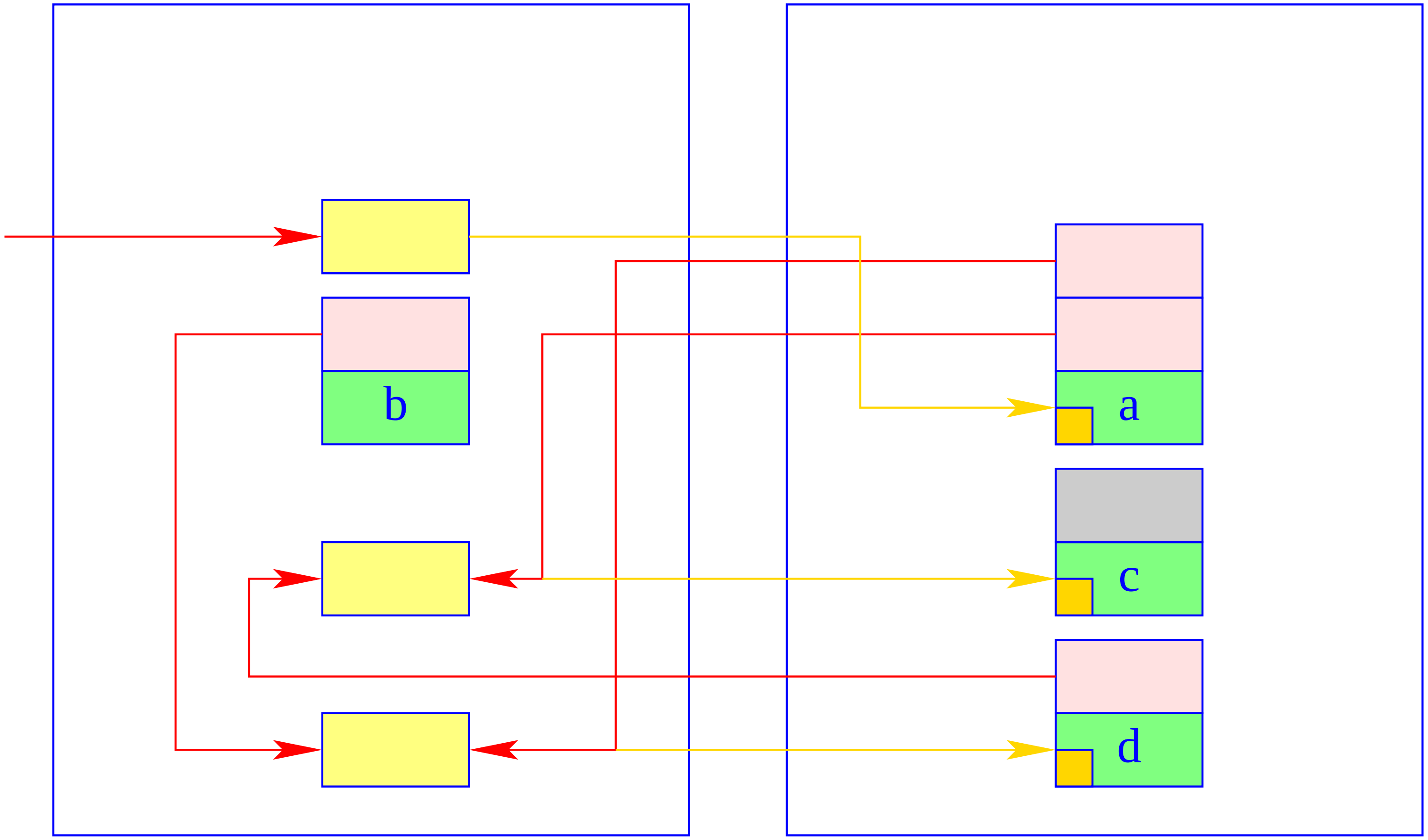
The garbage collection of the WiM must harmonize with backtracking.

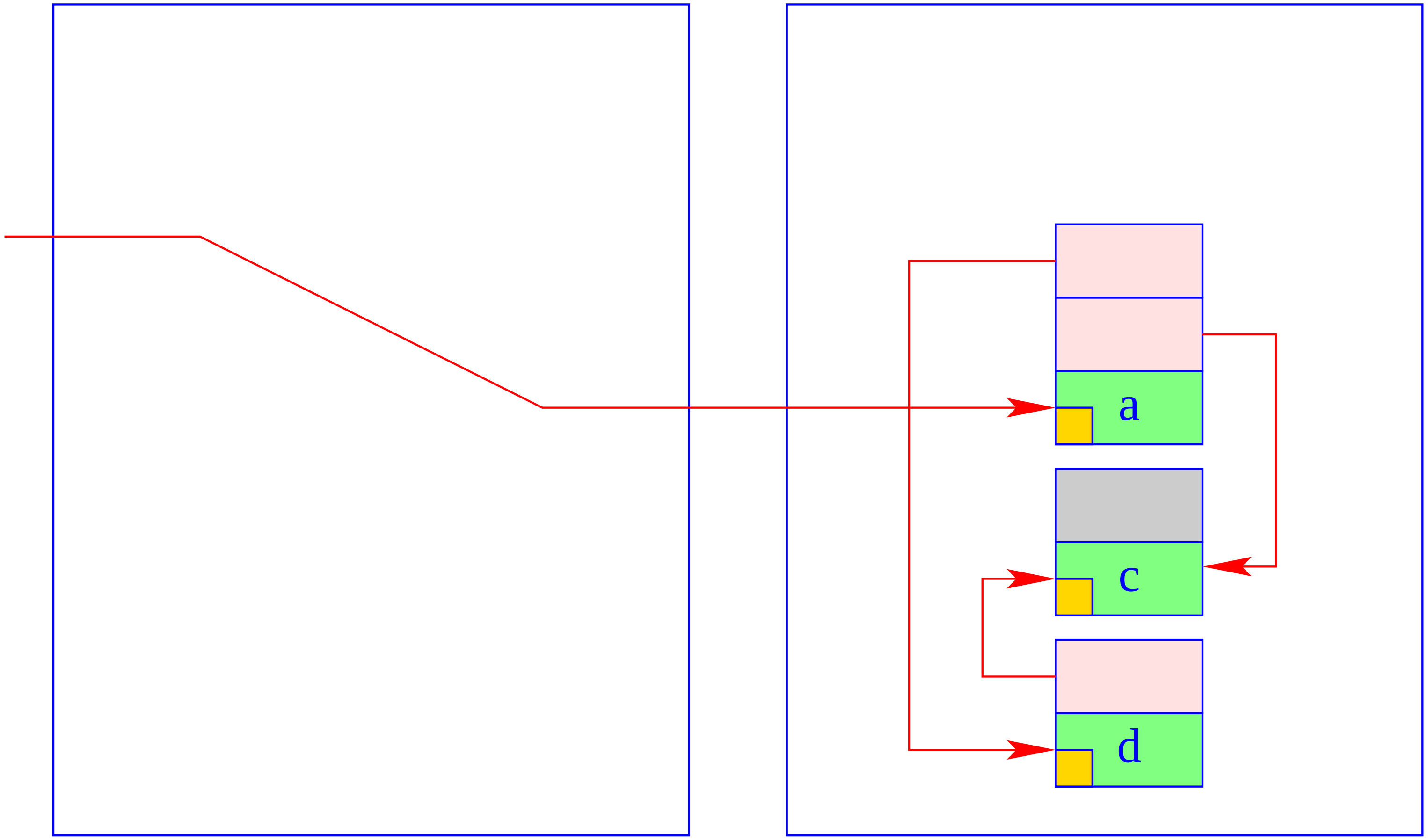
This means:

- The relative position of heap objects must not change during copying :-!!
- The heap references in the trail must be updated to the new positions.
- If heap objects are collected which have been created before the last backtrack point, then also the heap pointers in the stack must be updated.


a
b
c
d


a
b
c
d


b
a
c
d


a
c
d

# Threads

# 39 The Language ThreadedC

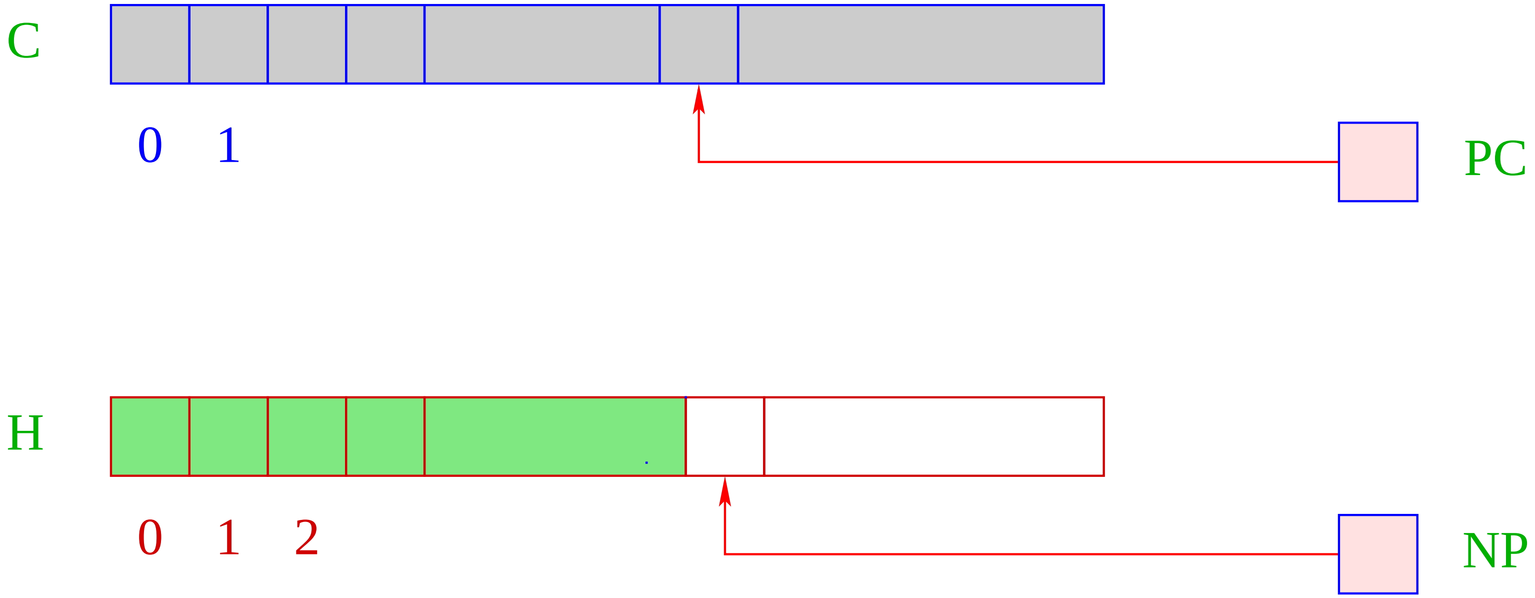
We extend C by a simple thread concept. In particular, we provide functions for:

- generating new threads: create();
- terminating a thread: exit();
- waiting for termination of a thread: join();
- mutual exclusion: lock(), unlock(); ...

In order to enable a parallel program execution, we extend the abstract machine (what else? :-)

# 40 Storage Organization

All threads share the same common code store and heap:

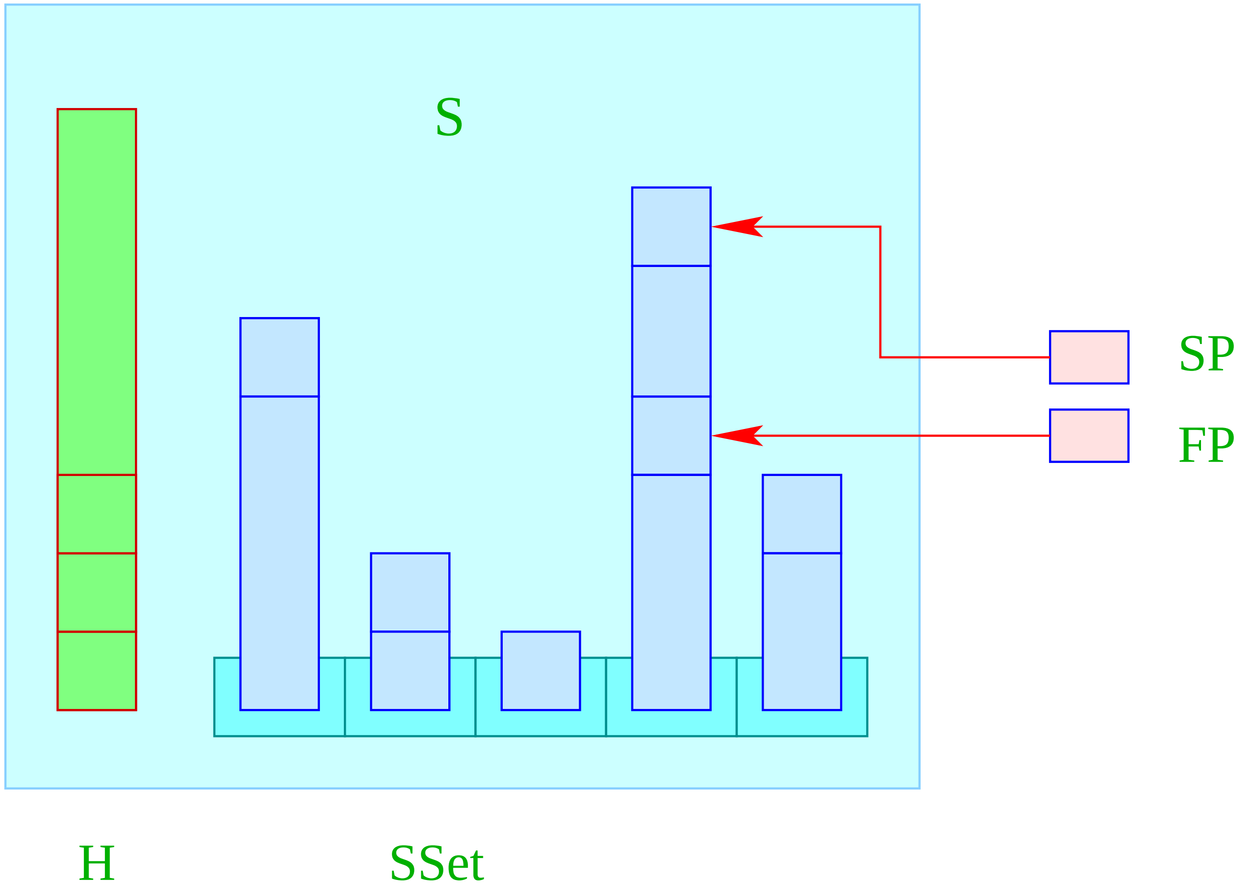


... similar to the CMa, we have:

C = Code Store – contains the CMa program;
every cell contains one instruction;
PC = Program-Counter – points to the next executable instruction;
H = Heap –
every cell may contain a base value or an address;
the globals are stored at the bottom;
NP = New-Pointer – points to the first free cell.

For a simplification, we assume that the heap is stored in a separate segment.
The function malloc() then fails whenever NP exceeds the topmost border.

Every thread on the other hand needs its own stack:



In constrast to the CMa, we have:

SSet = Set of Stacks – contains the stacks of the threads;
every cell may contain a base value of an address;
S = common address space for heap and the stacks;
SP = Stack-Pointer – points to the current topmost ocupied stack cell;
FP = Frame-Pointer – points to the current stack frame.

## Warning:

- If all references pointed into the heap, we could use separate address spaces for each stack.

  Besides SP and FP, we would have to record the number of the current stack :-)
- In the case of C, though, we must assume that all storage reagions live within the same address space — only at different locations :-)

  SP Und FP then uniquely identify storage locations.
- For simplicity, we omit the extreme-pointer EP.

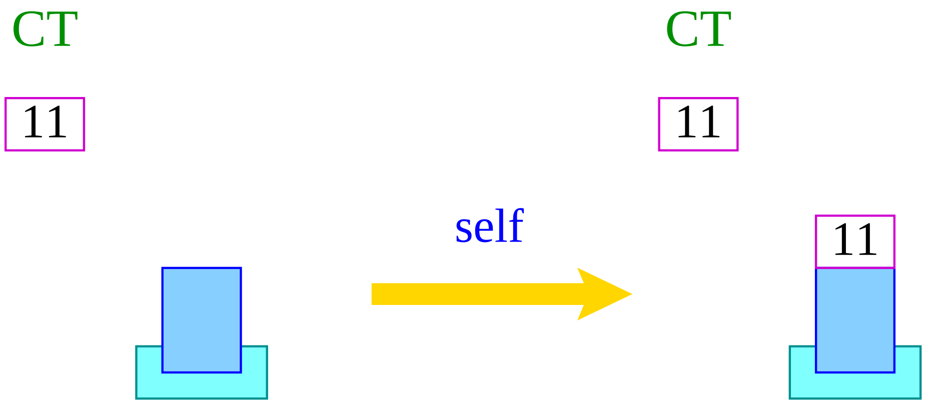
# 41 The Ready-Queue

Idea:

- Every thread has a unique number tid.
- A table TTab allows to determine for every tid the corresponding thread.
- At every point in time, there can be several executable threads, but only one running thread (per processor :-)
- the tid of the currently running thread is cept in the register CT (Current Thread).
- The function: **tid self ()** returns the tid of the current thread. Accordingly:

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

... where the instruction self pushes the content of the register CT onto the (current) stack:

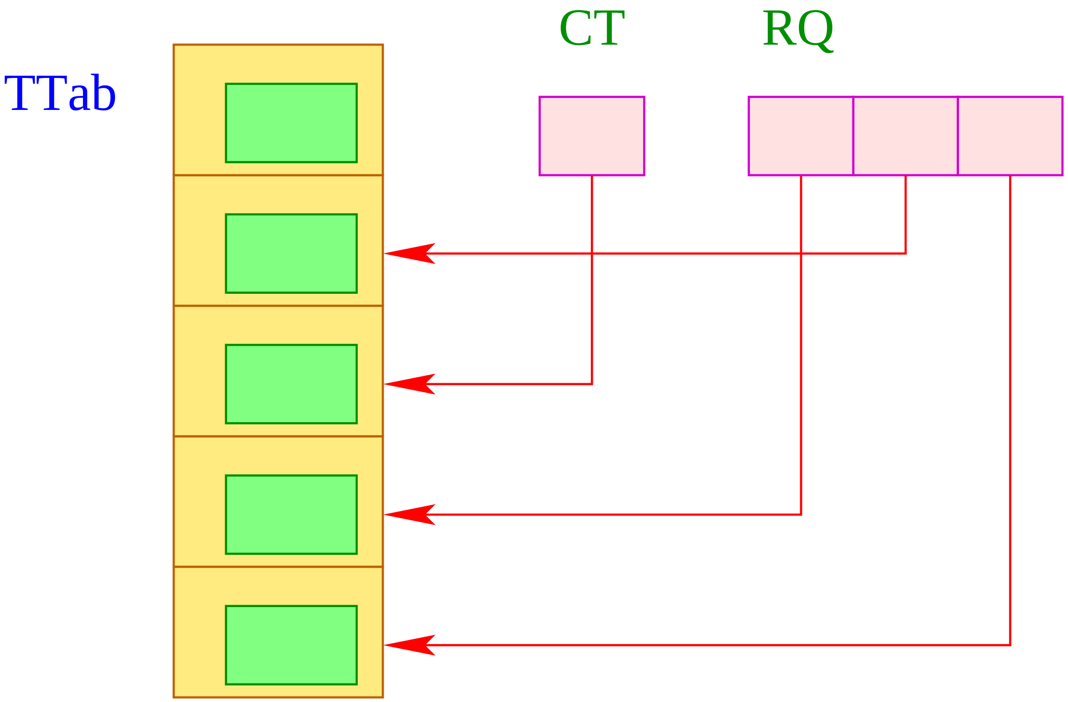


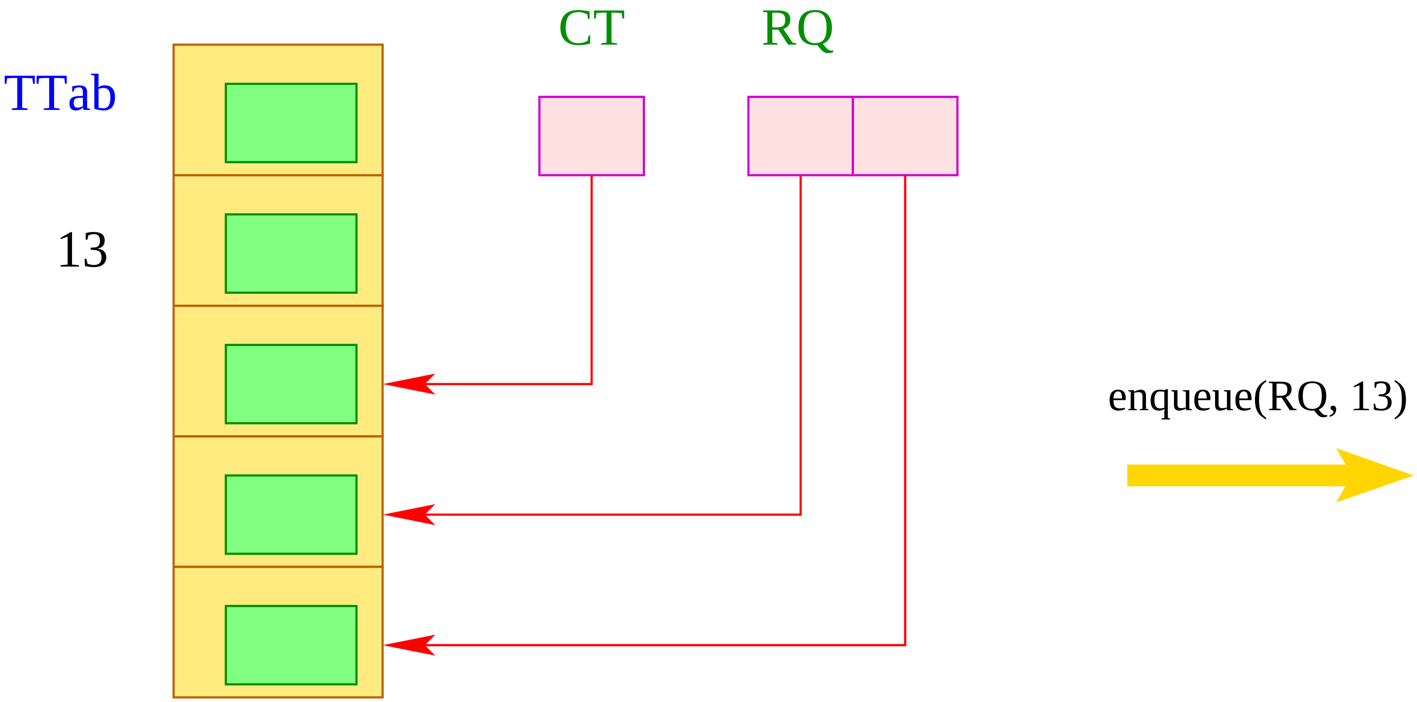
S[SP++] = CT;

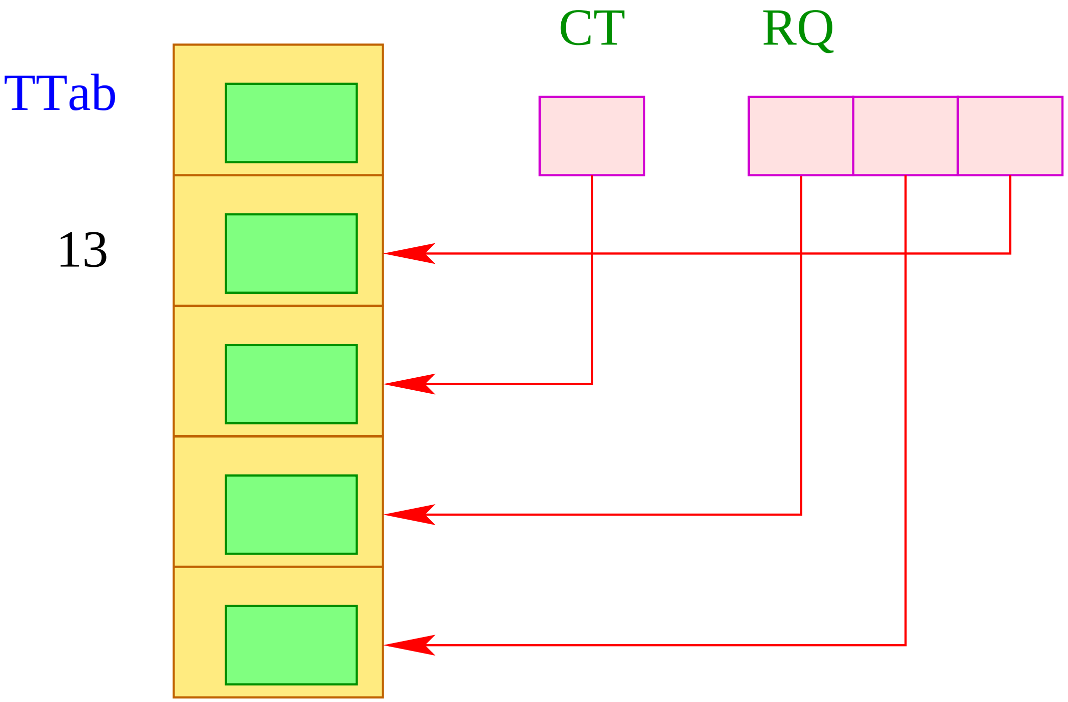
- The remaining executable threads (more precisely, their tid's) are maintained in the queue RQ (Ready-Queue).
- For queues, we need the functions:

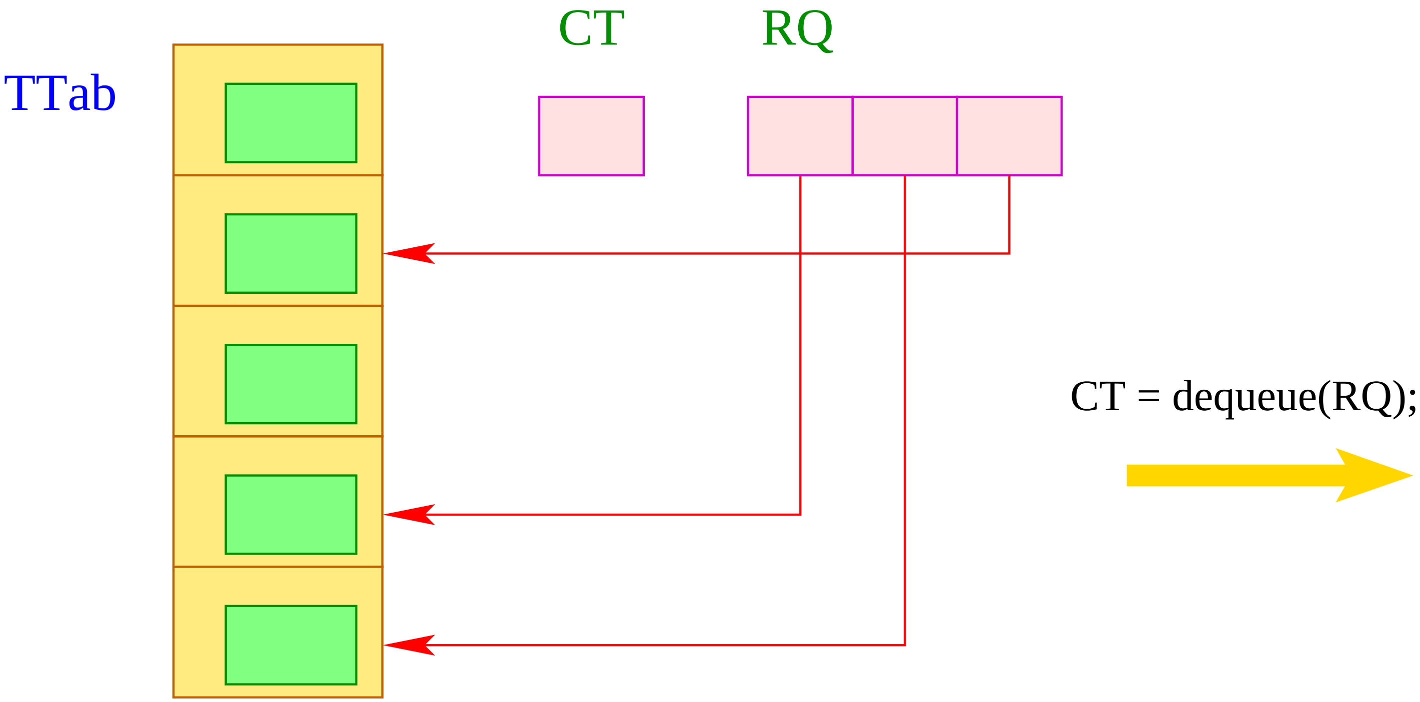
```
void enqueue (queue q, tid t),
tid dequeue (queue q)
```

which insert a tid into a queue and return the first one, respectively ...

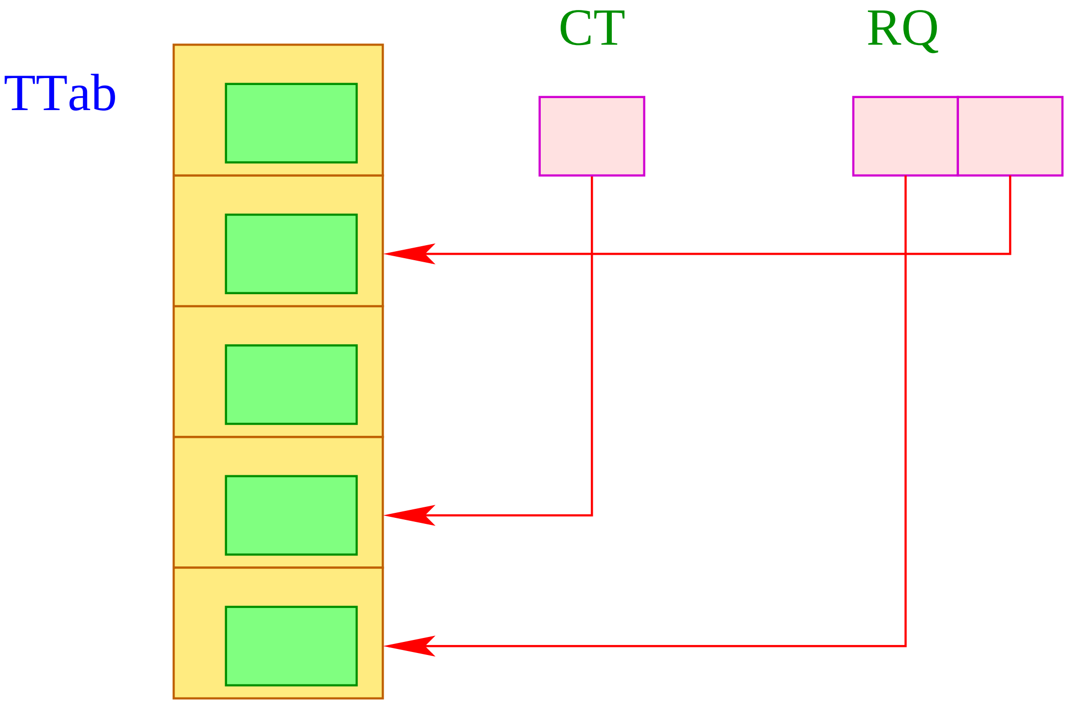

TTab
CT
RQ


TTab
13
CT
RQ
enqueue(RQ, 13)


CT
RQ
TTab
13

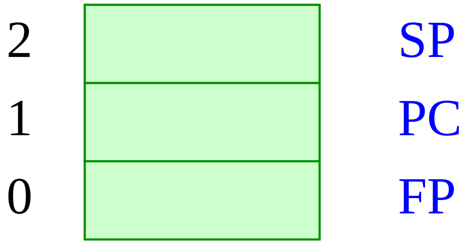


CT = dequeue(RQ);


CT
RQ
TTab

If a call to dequeue () failed, it returns a value $< \ 0$ :-)

The thread table must contain for every thread, all information which is needed for its execution. In particular it consists of the registers PC, SP und FP:



Interrupting the current thread therefore requires to save these registers:

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

Analogously, we restore these registers by calling the function:

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

Thus, we can realize an instruction yield which causes a thread-switch:

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

Only if the ready-queue is non-empty, the current thread is replaced :-)

# 42 Switching between Threads

## Problem:

We want to give each executable thread a fair chance to be completed.

$\Longrightarrow$

- Every thread must former or later be scheduled for running.
- Every thread must former or later be interrupted.

## Possible Strategies:

- Thread switch only at explicit calls to a function `yield()` :-(
- Thread switch after every instruction $\Longrightarrow$ too expensive :-(
- Thread switch after a fixed number of steps $\Longrightarrow$ we must install a counter and execute yield at dynamically chosen points :-(

We insert thread switches at selected program points ...

- at the beginning of function bodies;
- before every jump whose target does not exceed the current PC ...

$\Longrightarrow$ rare :-))

The modified scheme for loops $s \equiv \textbf{while}\ (e)\ s$ then yields:

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

## Note:

- **If-then-else**-Statements do not necessarily contain thread switches.
- **do-while**-Loops require a thread switch at the end of the condition.
- Every loop should contain (at least) one thread switch :-)
- Loop-Unroling reduces the number of thread switches.
- At the translation of **switch**-statements, we created a jump table behind the code for the alternatives. Nonetheless, we can avoid thread switches here.
- At freely programmed uses of jumpi as well as jumpz we should also insert thread switches before the jump (or at the jump target).
- If we want to reduce the number of executed thread switches even further, we could switch threads, e.g., only at every 100th call of yield ...

# 43 Generating New Threads

We assume that the expression: $s \equiv \textbf{create}\ (e_0, e_1)$ first evaluates the expressions $e_i$ to the values $f, a$ and then creates a new thread which computes $f\ (a)$ .

If thread creation fails, $s$ returns the value $-1$.

Otherwise, $s$ returns the new thread's tid.

## Tasks of the Generated Code:

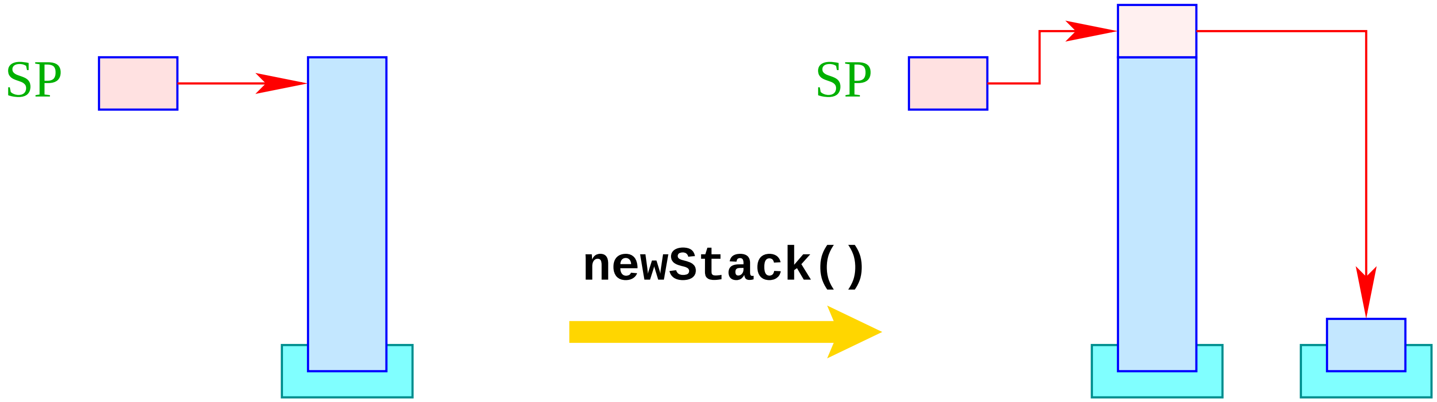
- Evaluation of the $e_i$;
- Allocation of a new run-time stack together with a stack frame for the evaluation of $f\ (a)$;
- Generation of a new tid;
- Allocation of a new entry in the TTab;
- Insertion of the new tid into the ready-queue.

The translation of $s$ then is quite simple:

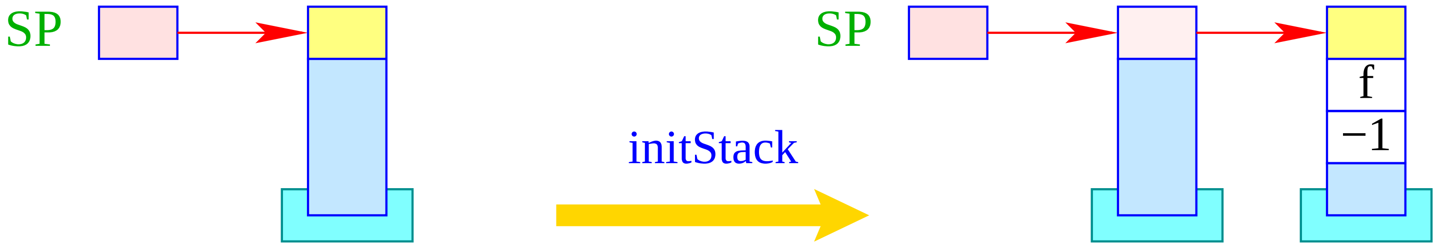
$$
\begin{array}{lll}
\mathrm{code}_{\mathrm{R}}\ s\ \rho & = & \mathrm{code}_{\mathrm{R}}\ e_0\ \rho \\
 & & \mathrm{code}_{\mathrm{R}}\ e_1\ \rho \\
 & & \mathrm{initStack} \\
 & & \mathrm{initThread}
\end{array}
$$

where we assume the argument value occupies 1 cell :-)

For the implementation of initStack we need a run-time function newStack() which returns a pointer onto the first element of a new stack:



If the creation of a new stack fails, the value 0 is returned.



```
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;
```

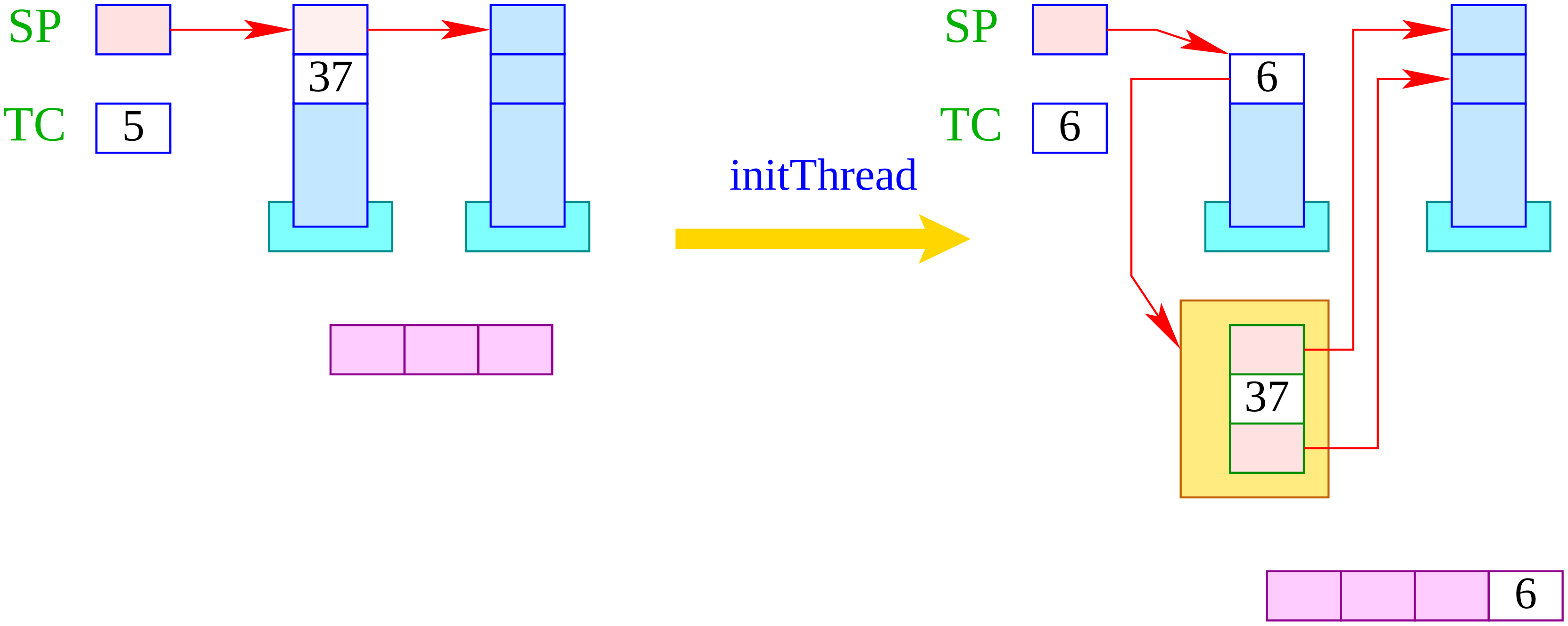
## Note:

- The continuation address f points to the (fixed) code for the termination of threads.
- Inside the stack frame, we no longer allocate space for the EP $\Longrightarrow$ the return value has relative address $-2$.
- The bottom stack frame can be identified through FPold = -1 :-)

In order to create new thread ids, we introduce a new register TC (Thread Count).

Initially, TC has the value 0 (corresponds to the tid of the initial thread).

Before thread creation, TC is incremented by 1.


SP
TC
5
37
initThread
SP
TC
6
6
37
6

```
if (S[SP] $\geq$ 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 Terminating Threads

Termination of a thread (usually :-) returns a value. There are two (regular) ways to terminate a thread:

1. The initial function call has terminated. Then the return value is the return value of the call.
2. The thread executes the statement **exit** $(e)$; Then the return value equals the value of $e$.

Warning:

- We want to return the return value in the bottom stack cell.
- **exit** may occur arbitrarily deeply nested inside a recursion. Then we de-allocate all stack frames ...
- ... and jump to the terminal treatment of threads at address f .

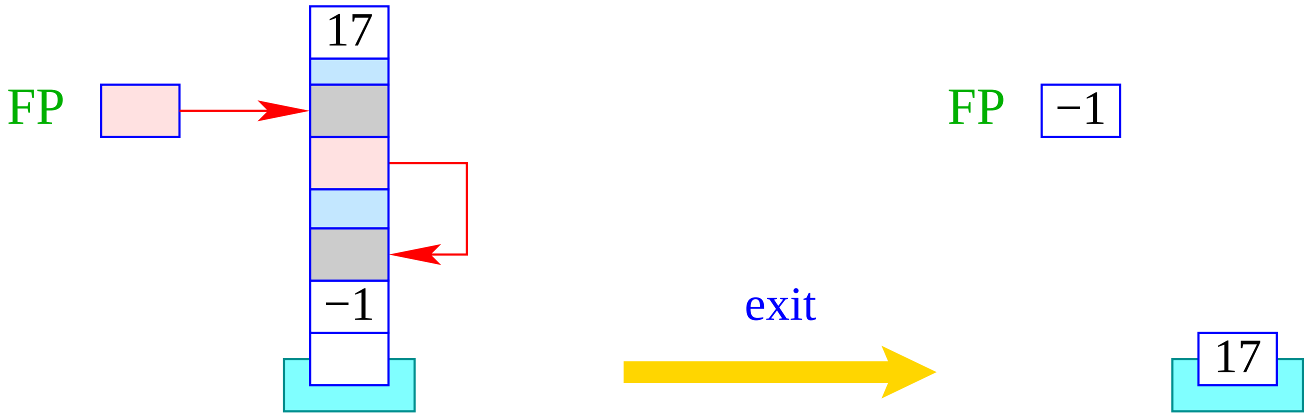
Therefore, we translate:

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

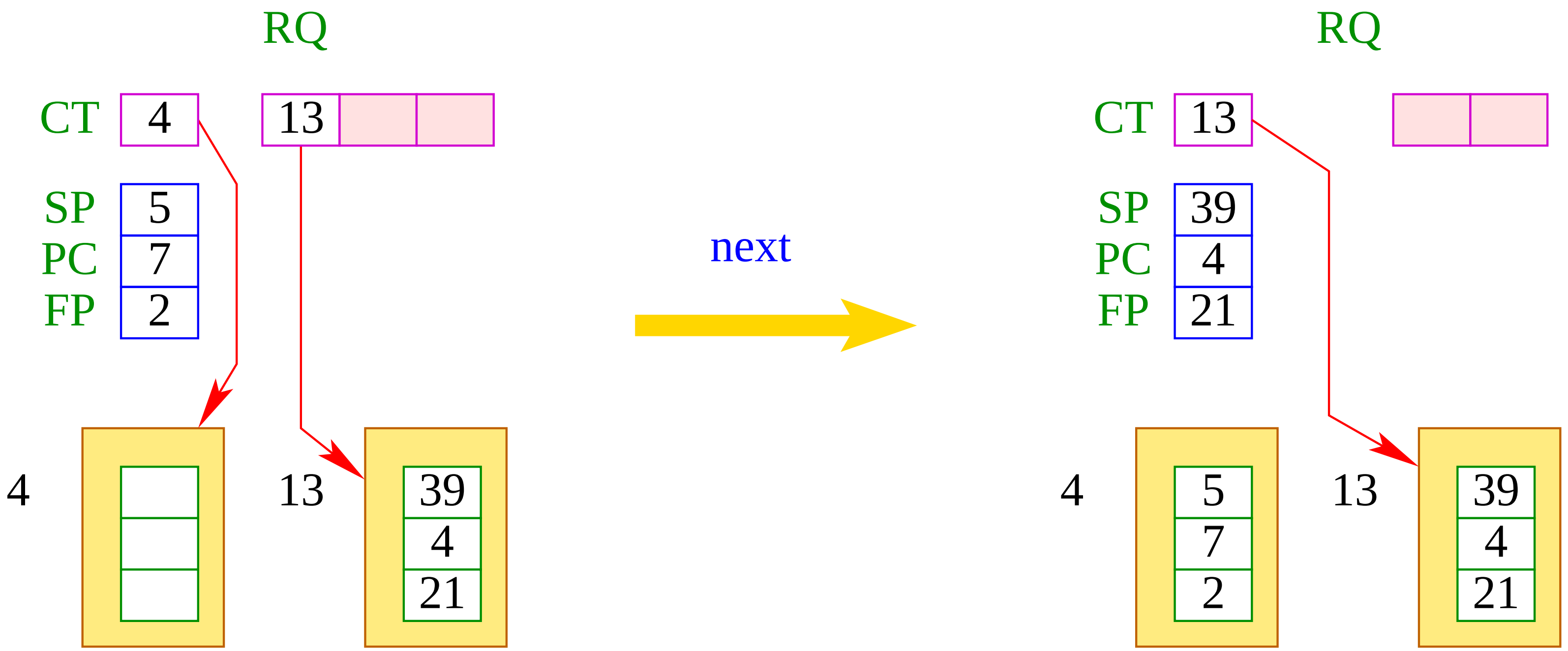
The instruction term is explained later :-)

The instruction exit successively pops all stack frames:

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


FP
17
−1
exit
FP
−1
17

The instruction next activates the next executable thread:
in contrast to yield the current thread is not inserted into RQ .



Ist die Schlange RQ leer, wird zusätzlich If the queue RQ is empty, we additionally terminate the whole program:

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