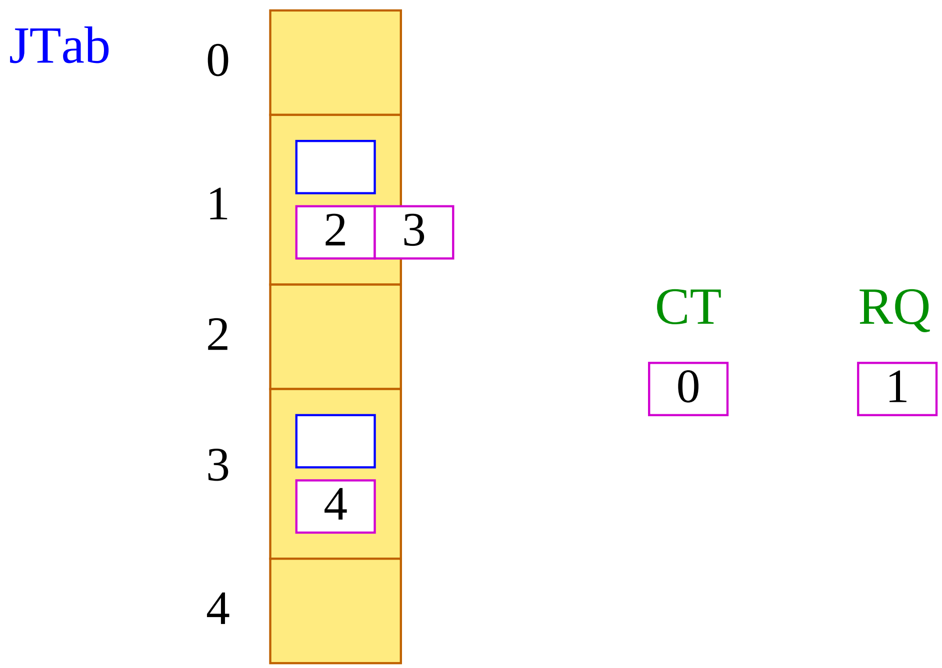
45 Waiting for Termination

Occaionally, a thread may only continue with its execution, if some other thread has terminated. For that, we have the expression **join** (e) where we assume that e evaluatges to a thread id tid.

- If the thread with the given tid is already terminated, we return its return value.
- If it is not yet terminated, we interrupt the current thread execution.
- We insert the current thread into the queue of treads already waiting for the termination.

  We save the current registers and switch to the next executable thread.
- Thread waiting for termination are maintained in the table JTab.
- There, we also store the return values of threads :-)

Example:



Thread 0 is running, thread 1 could run, threads 2 and 3 wait for the termination of 1, and thread 4 waits for the termination of 3.

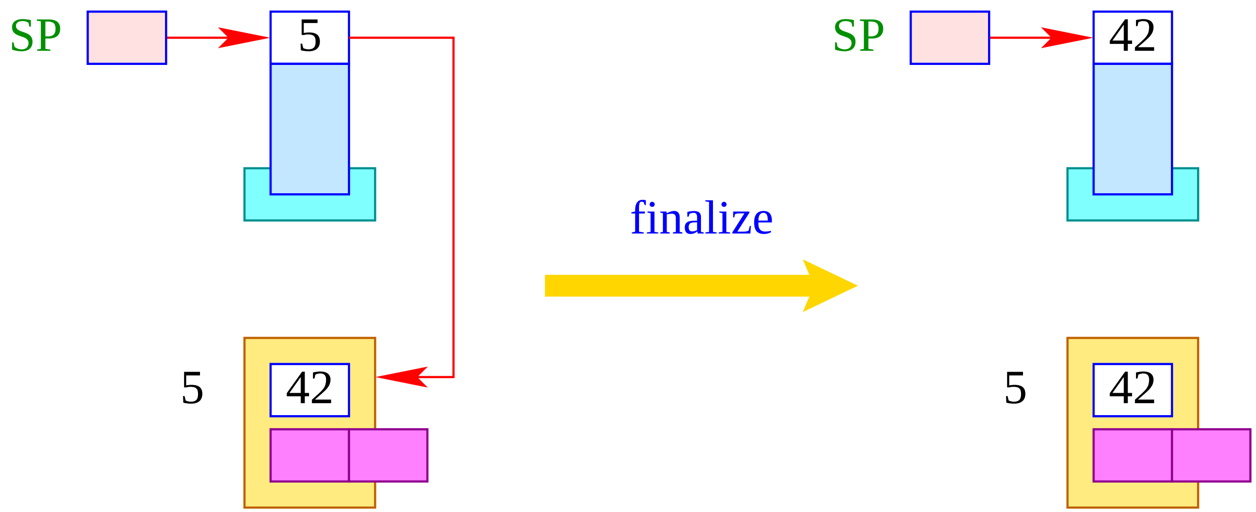
Thus, we translate:

$$\begin{array}{lll} \mathrm{code_R}\ \mathbf{join}\ (e)\ \rho & = & \mathrm{code_R}\ e\ \rho \\ & & \mathrm{join} \\ & & \mathrm{finalize} \end{array}$$

... where the instruction join is defined by:

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

... accordingly:



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

The instruction sequence:

term
next

is executed before a thread is terminated.
Therefore, we store them at the location f.

The instruction next switches to the next executable thread. Before that, though,

- ... the last stack frame must be popped and the result be stored in the table JTab ;
- ... the thread must be marked as terminated, e.g., by additionally setting the PC to -1;
- ... all threads must be notified which have waited for the termination.

For the instruction term this means:

$PC = -1;$
$JTab[CT][1] = S[SP];$
$freeStack(SP);$
while $(0 \leq tid =$ dequeue $(JTab[CT][0]))$
    enqueue $(RQ, tid);$

The run-time function freeStack (int adr) removes the (one-element) stack at the location adr :

46 Mutual Exclusion

A mutex is an (abstract) datatype (in the heap) which should allow the programmer to dedicate exclusive access to a shared resource (mutual exclusion).

The datatype supports the following operations:

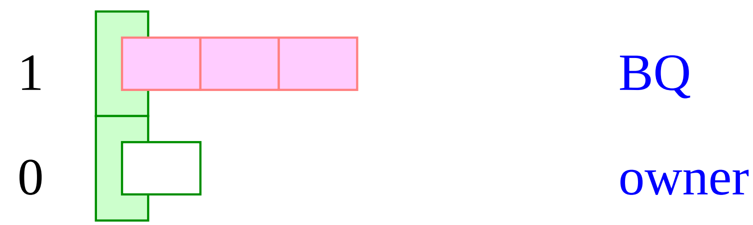
Mutex $*$ newMutex ();	—	creates a new mutex;
void lock (**Mutex** $*$me);	—	tries to acquire the mutex;
void unlock (**Mutex** $*$me);	—	releases the mutex;

Warning:

A thread is only allowed to release a mutex if it has owned it beforehand :-)

A mutex me consists of:

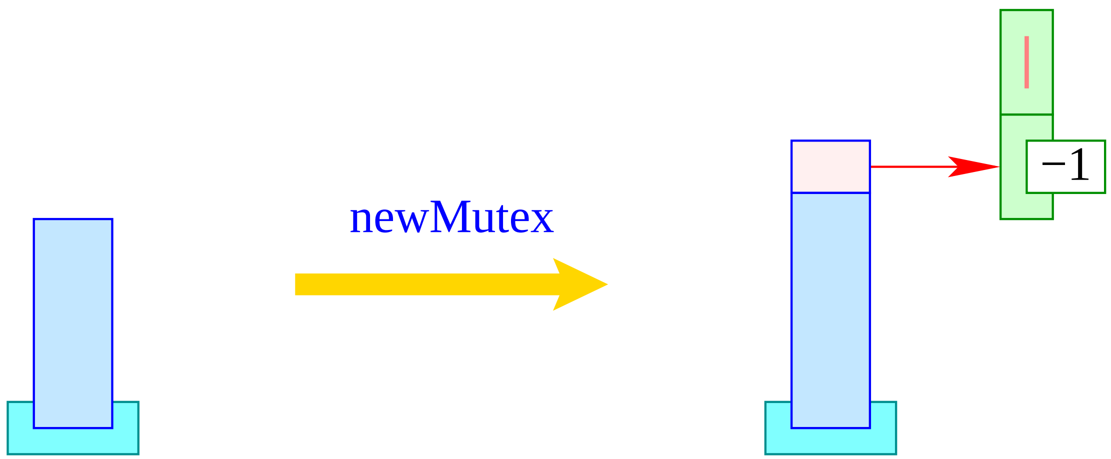
- the tid of the current owner (or -1 if there is no one);
- the queue BQ of blocked threads which want to acquire the mutex.



Then we translate:

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

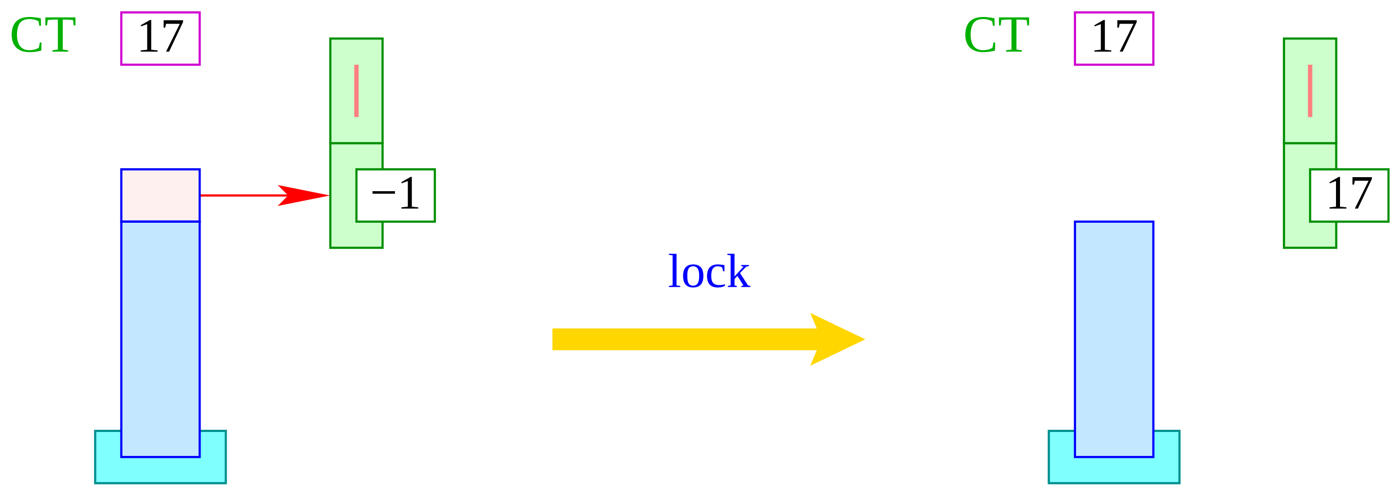
where:



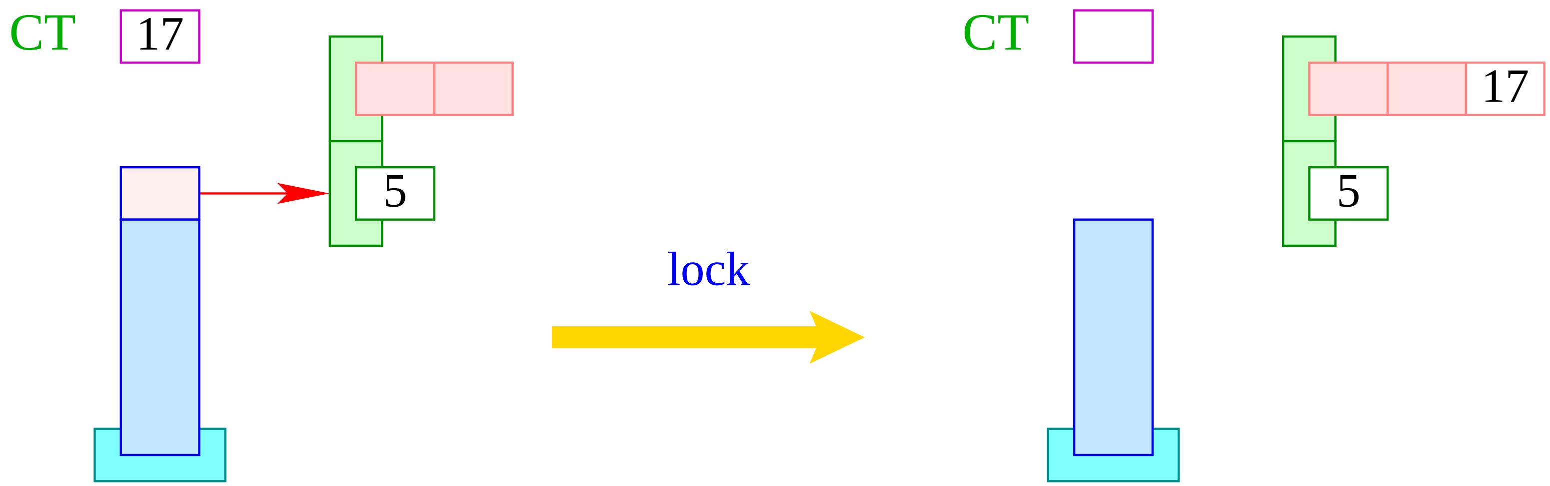
Then we translate:

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

where:



If the mutex is already owned by someone, the current thread is interrupted:

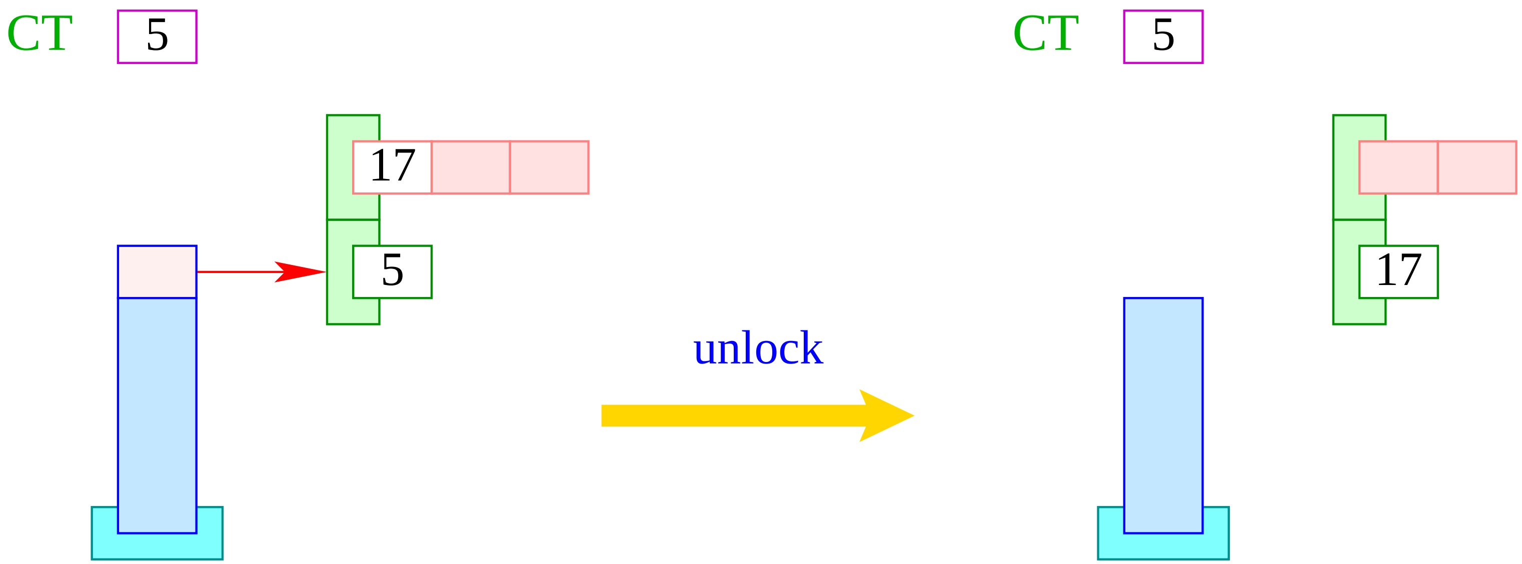


$$
\begin{array}{l}
\text{if } (S[S[SP]] < 0) \quad S[S[SP--]] = CT; \\
\text{else } \{ \\
\qquad \text{enqueue } (\, S[SP--]+1, CT \,); \\
\qquad \text{next}; \\
\quad \}
\end{array}
$$

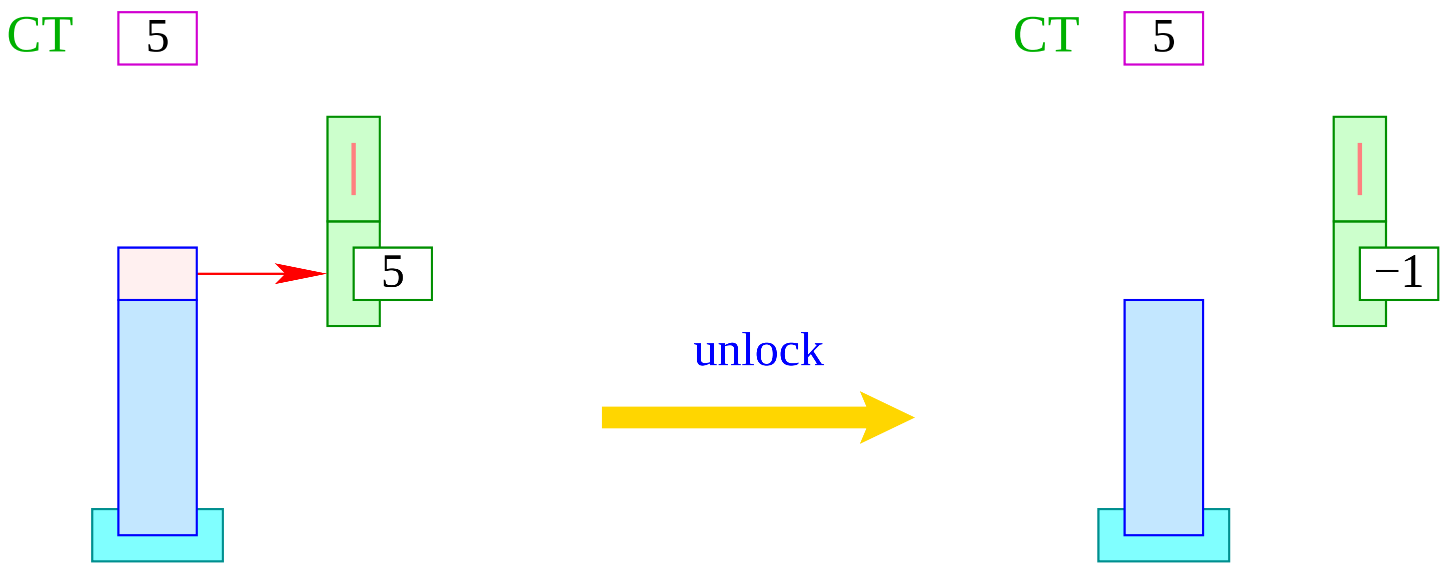
Accordingly, we translate:

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

where:



If the queue BQ is empty, we release the mutex:



if (S[S[SP]] $\neq$ CT) Error ("Illegal unlock!");

if ($0 >$ tid $=$ dequeue (S[SP]+1)) S[S[SP– –]] = –1;

else {

S[S[SP--]] = tid;

enqueue (RQ, tid);

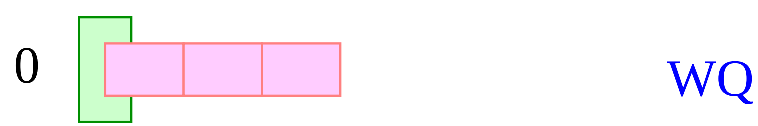
}

47 Waiting for Better Wheather

It may happen that a thread owns a mutex but must wait until some extra condition is true.

Then we want the thread to remain in-active until it is told otherwise.

For that, we use condition variables. A condition variable consists of a queue WQ of waiting threads :-)



For condition variables, we introduce the functions:

CondVar $*$ newCondVar ();	—	creates a new condition variable;
void wait (**CondVar** $*$ cv), **Mutex** $*$ me);	—	enqueues the current thread;
void signal (**CondVar** $*$ cv);	—	re-animates one waiting thread;
void broadcast (**CondVar** $*$ cv);	—	re-animates all waiting threads.

Then we translate:

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

where:

The translation of the body amounts to:

```
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
```

The function Down() decrements the counter.

If the counter becomes negative, **wait** is called:

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

The translation of the body amounts to:

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

The function Up() increments the counter again.

If it is afterwards not yet positive, there still must exist waiting threads. One of these is sent a signal:

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

The translation of the body amounts to:

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:

- All threads live within the same storage.
- Every thread requires its own stack (at least conceptually).

1. Idea:

Allocate for each new thread a fixed amount of storage space.

$$\Longrightarrow$$

Then we implement:

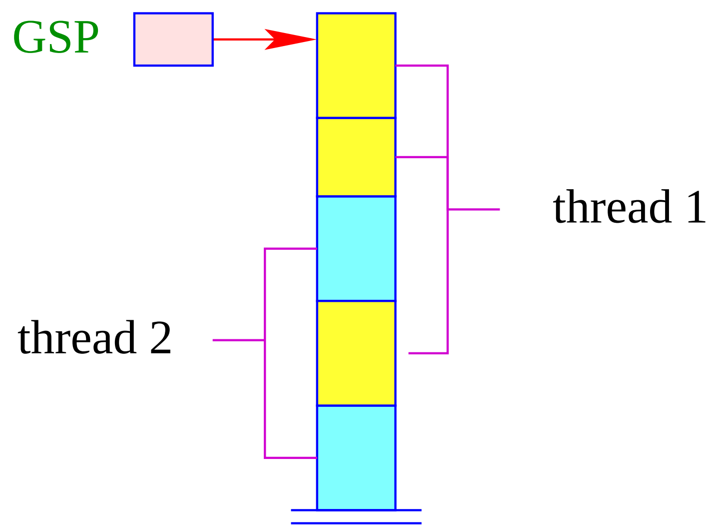
```
void *newStack() { return malloc(M); }
void freeStack(void *adr) { free(adr); }
```

Problem:

- Some threads consume much, some only little stack space.
- The necessary space is statically typically unknown :-(

2. Idea:

- Maintain all stacks in one joint Frame-Heap FH :-)
- Take care that the space inside the stack frame is sufficient at least for the current function call.
- A global stack-pointer GSP points to the overall topmost stack cell ...



Allocation and de-allocation of a stack frame makes use of the run-time functions:

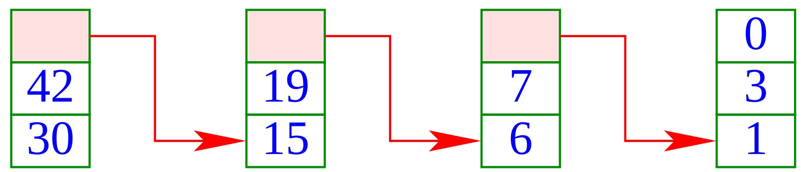
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

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

Warning:

The de-allocated block may reside inside the stack :-(

$$\Longrightarrow$$

We maintain a list of freed stack blocks :-)

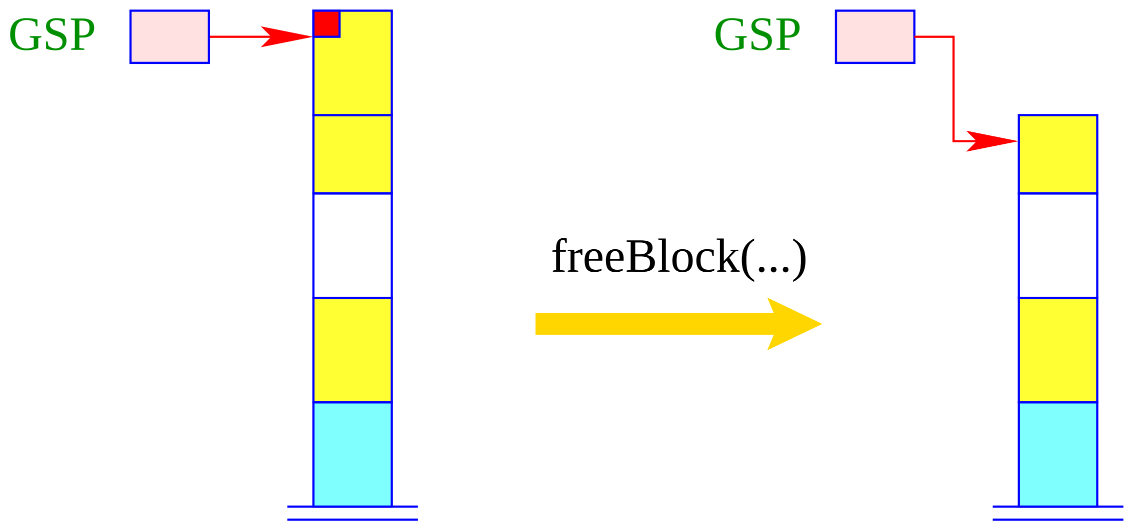


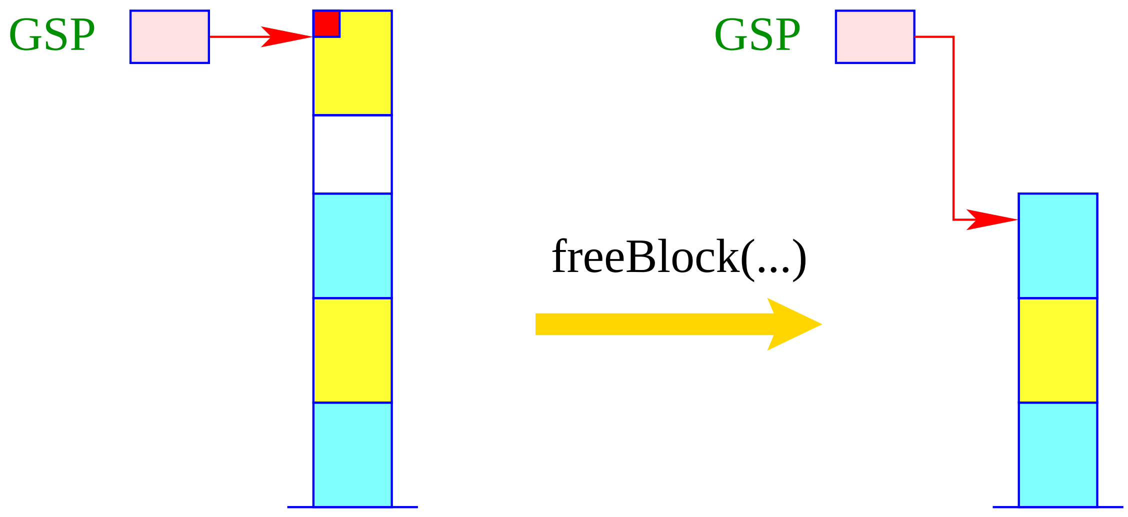
This list supports a function

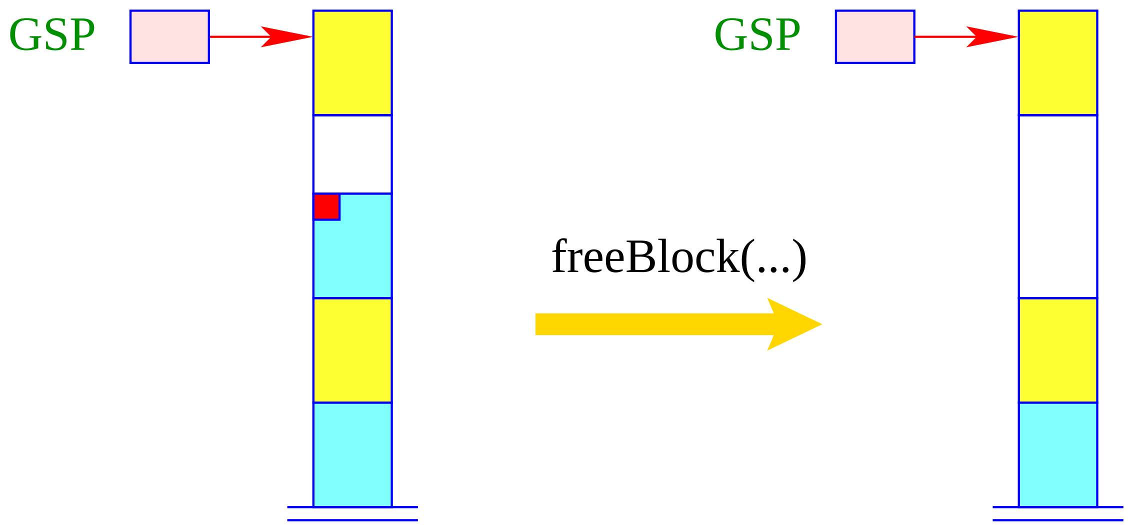
```
void insertBlock(int max, int min)
```

which allows to free single blocks.

- If the block is on top of the stack, we pop the stack immediately;
- ... together with the blocks below – given that these have already been marked as de-allocated.
- If the block is inside the stack, we merge it with neighbored free blocks:


GSP
GSP
freeBlock(...)


GSP
GSP
freeBlock(...)


GSP
GSP
freeBlock(...)

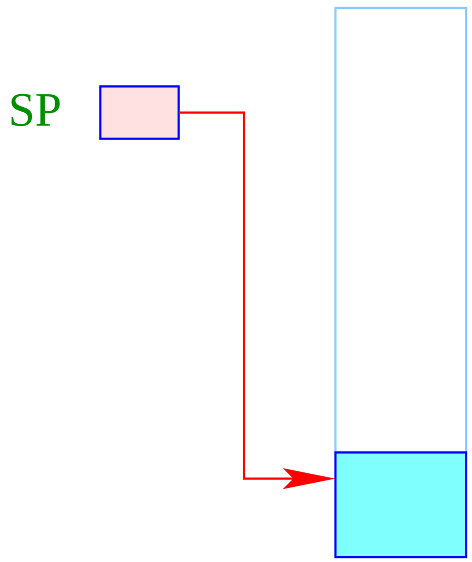
Approach:

We allocate a fresh block for every function call ...

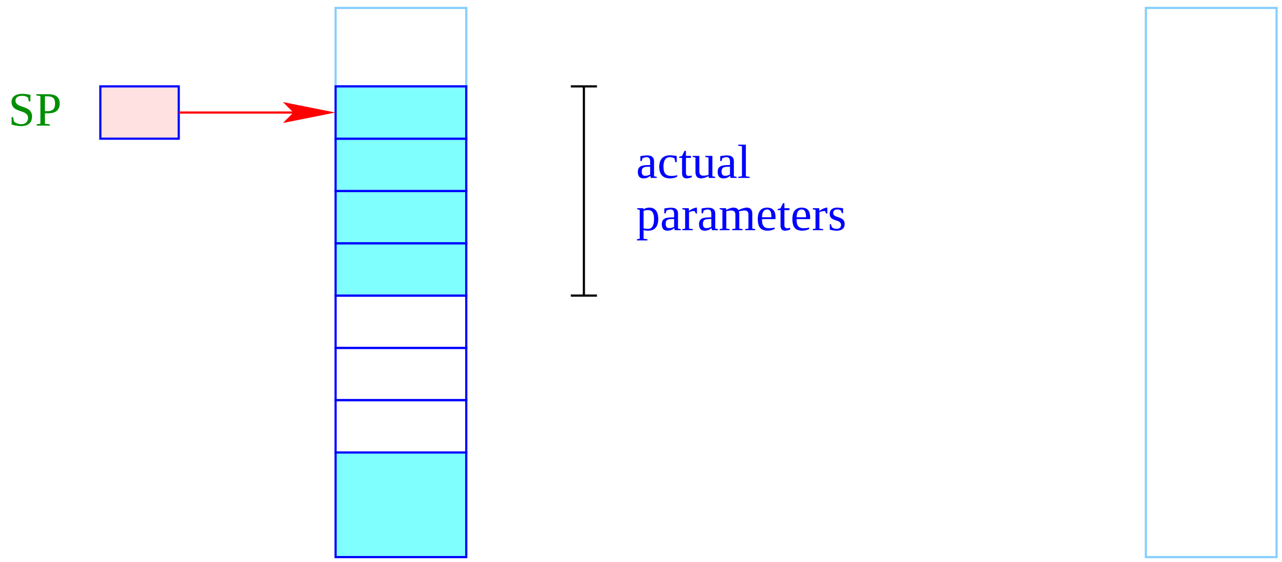
Problem:

When ordering the block before the call, we do not yet know the space consumption of the called function :-(

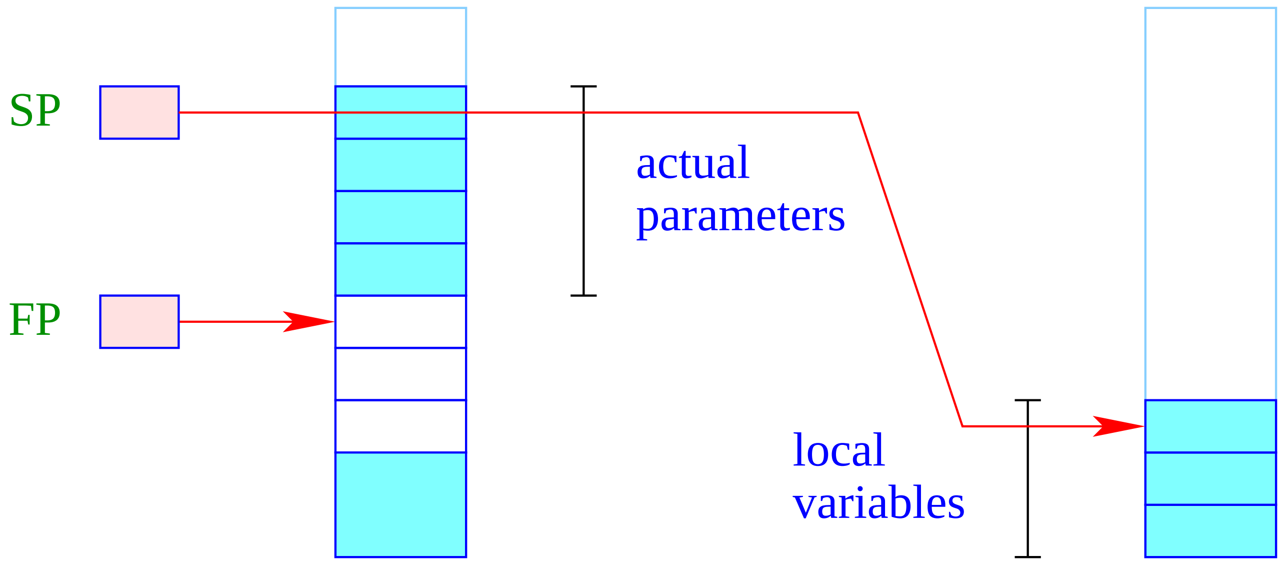
$\Longrightarrow$ We order the new block after entering the function body!



Organisational cells as well as actual parameters must be allocated inside the old block ...



When entering the new function, we now allocate the new block ...



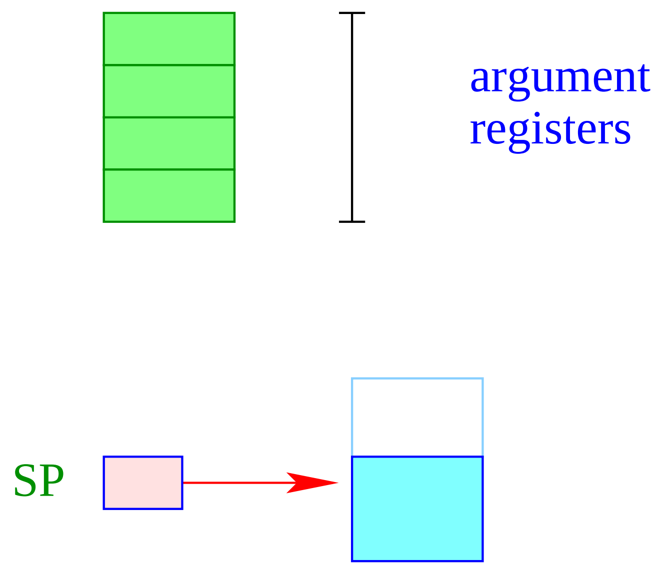
Inparticular, the local variables reside in the new block ...

$\Longrightarrow$ We address ...

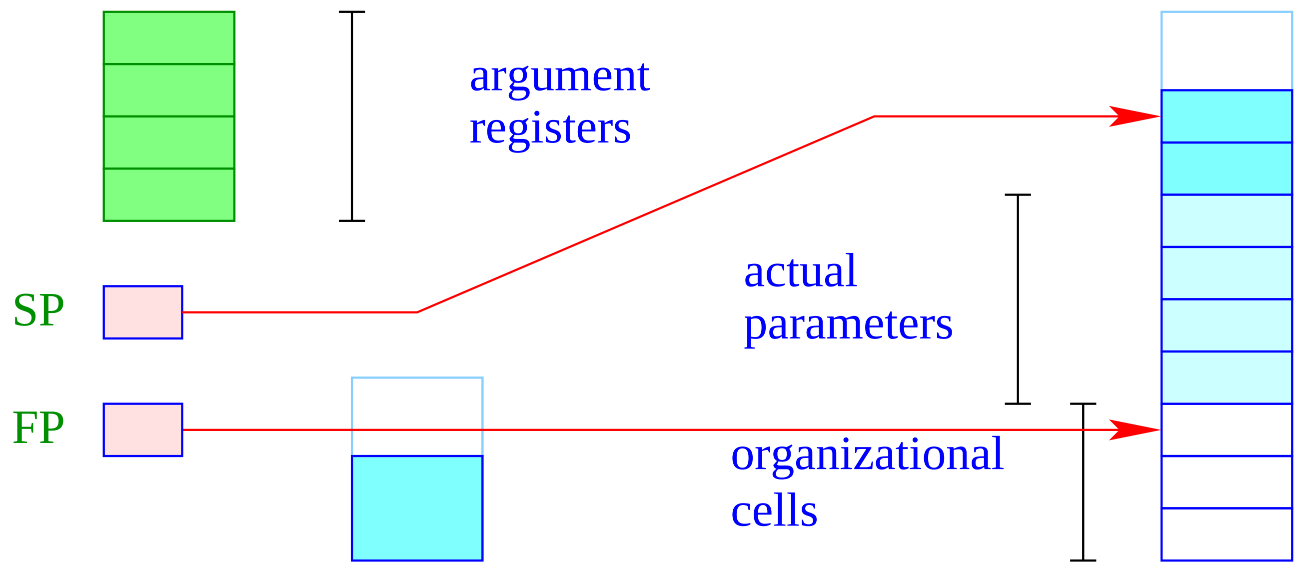
- the formal parameters relatively to the frame-pointer;
- the local variables relatively to the stack-pointer :-)

$\Longrightarrow$ We must re-organize the complete code generation ... :-(

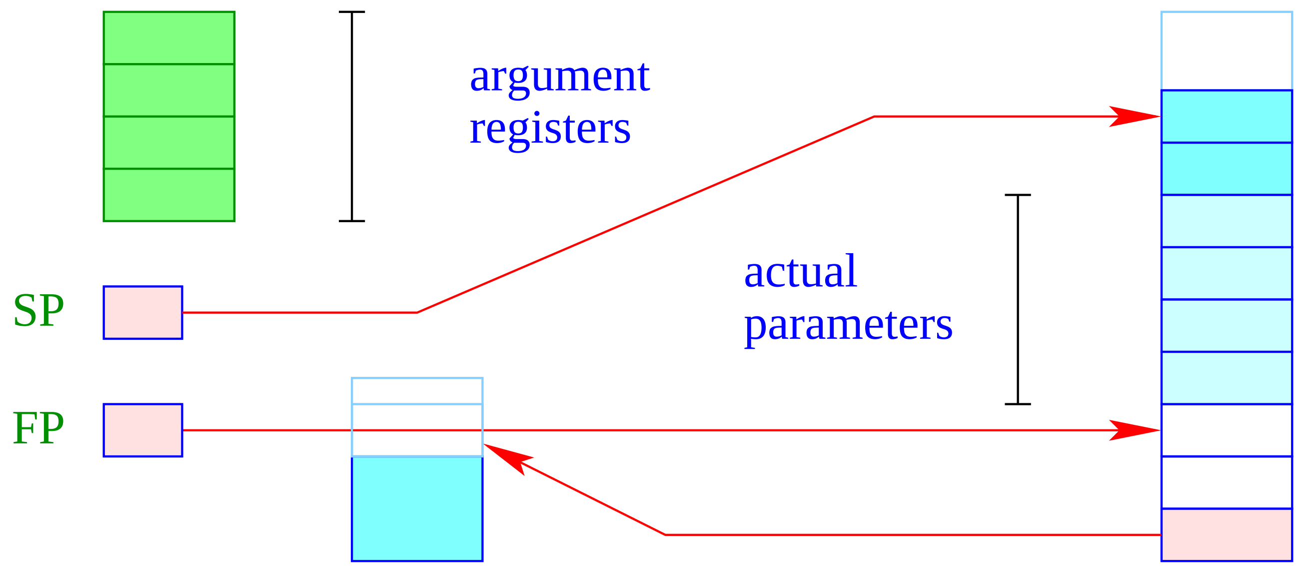
Alternative: Passing of parameters in registers ... :-)



The values of the actual parameters are determined before allocation of the new stack frame.



The complete frame is allocated inside the new block – plus the space for the current parameters.



Inside the new block, though, we must store the old SP (possibly +1) in order to correctly return the result ... :-)

3. Idea: Hybrid Solution

- For the first k threads, we allocate a separate stack area.
- For all further threads, we successively use one of the existing ones !!!

$$\Longrightarrow$$

- For few threads extremely simple and efficient;
- For many threads amortized storage usage :-))