Script generated by TTT

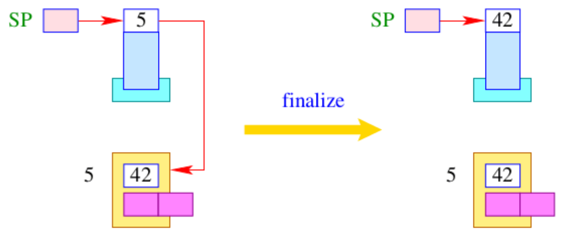
Title: Seidl: Virtual_Machines (06.07.2015)

Date: Mon Jul 06 10:17:04 CEST 2015

Duration: 89:38 min

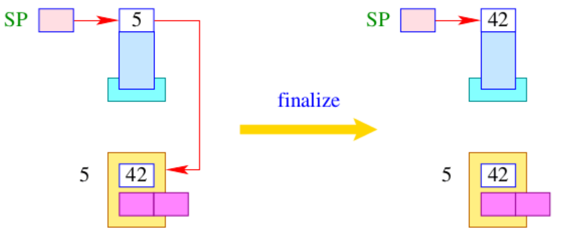
Pages: 62

... accordingly:



S[SP] = JTab[tid][0];

... accordingly:



S[SP] = JTab[tid][0];

430

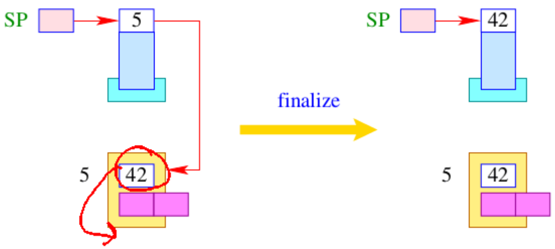
Thus, we translate:

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

... where the instruction join is defined by:

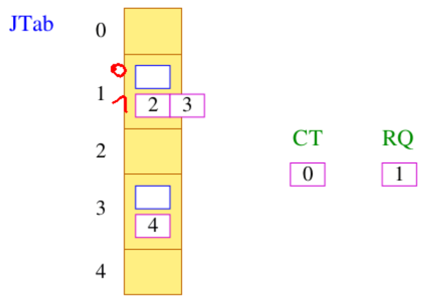
tid = S[SP];
if (TTab[tid][1] $\geq$ 0) {
  enqueue (JTab[tid][1], CT);
  next
}

... accordingly:



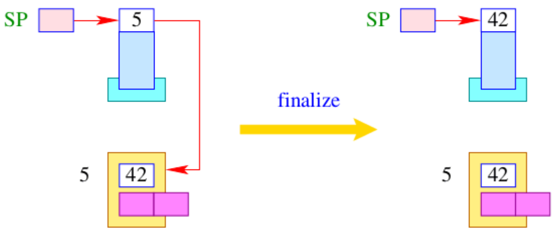
S[SP] = JTab[tid][0];

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.

... accordingly:



S[SP] = JTab[tid][0];

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

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

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

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

53 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 :-)

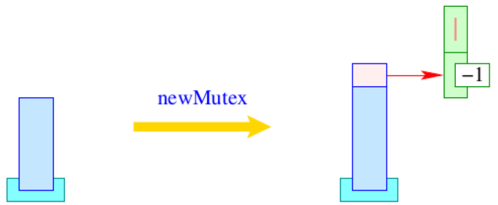
A mutex me consists of:

- 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:

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

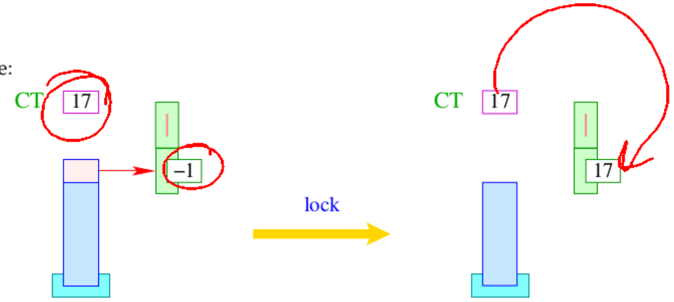
where:



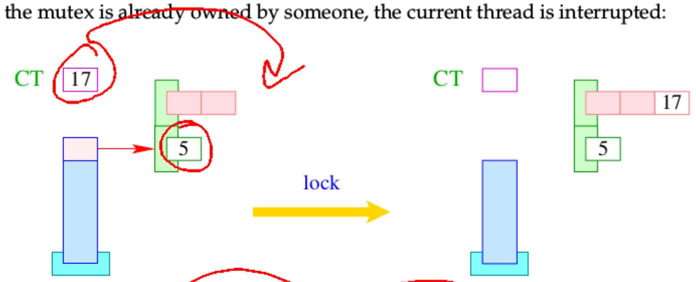
Then we translate:

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

where:



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

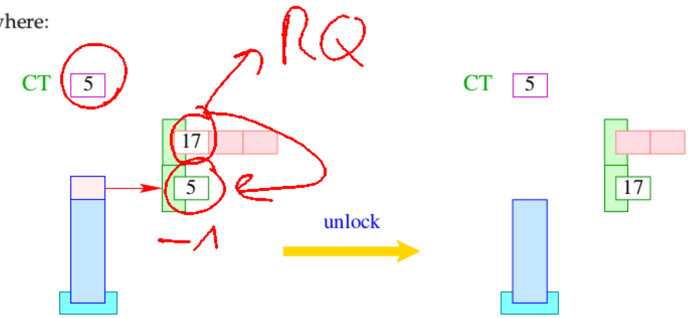



```
if (S[S[SP]] < 0) S[S[SP--]] = CT;
else {
        enqueue ( S[SP--]+1, CT );
        next;
}
```


Accordingly, we translate:

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

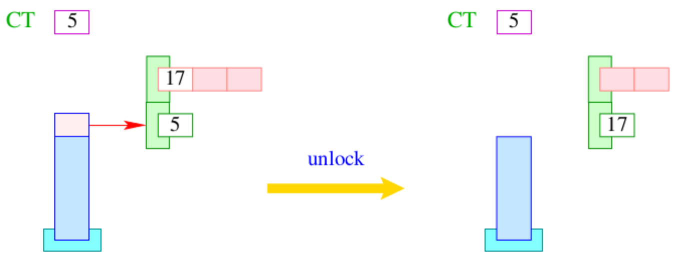
where:



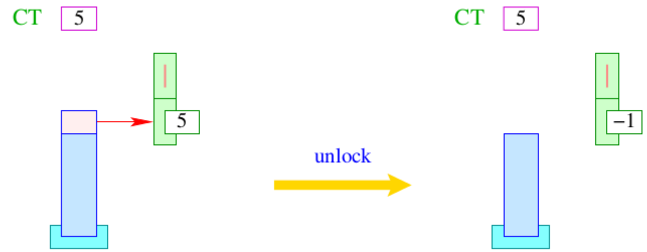
Accordingly, we translate:

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

where:

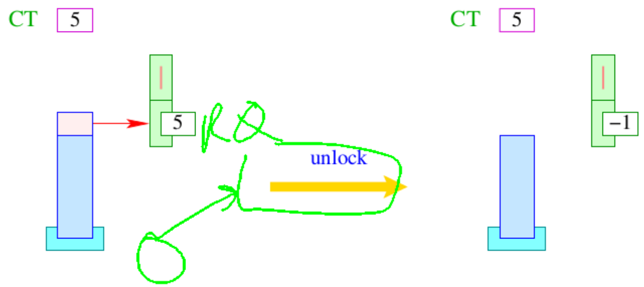


If the queue BQ is empty, we release the mutex:



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);
}

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);
}

54 Waiting for Better Weather

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:

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

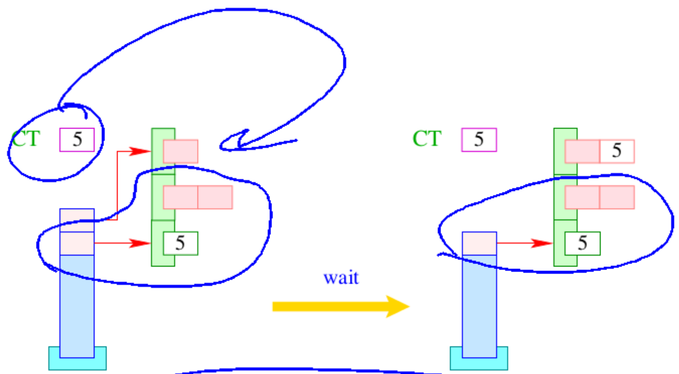
where:

After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

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

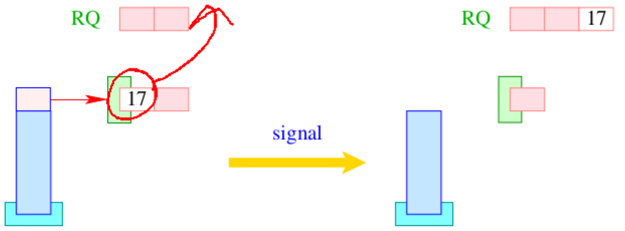
where ...



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

Accordingly, we translate:

$$\begin{array}{rcl} \text{code } \textbf{signal } (e);\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{signal} \end{array}$$




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


Analogously:

$$\begin{array}{rcl} \text{code } \textbf{broadcast } (e);\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{broadcast} \end{array}$$

where the instruction broadcast enqueues all threads from the queue WQ into the ready-queue RQ :

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

Warning:

The re-animated threads are not blocked !!!

When they become running, though, they first have to acquire their mutex :-)

55 Example: Semaphores

A semaphore is an abstract datatype which controls the access of a bounded number of (identical) resources.

Operations:

Sema ∗ newSema (**int** n)	—	creates a new semaphore;
void Up (Sema ∗ s)	—	increases the number of free resources;
void Down (Sema ∗ s)	—	decreases the number of available resources.

Therefore, a semaphore consists of:

- a counter of type **int**;
- a mutex for synchronizing the semaphore operations;
- a condition variable.

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

Therefore, a semaphore consists of:

- a counter of type **int**;
- a mutex for synchronizing the semaphore operations;
- a condition variable.

```
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);
}
```


The translation of the body amounts to:

alloc 1	newMutex	newCondVar	loadr -2	loadr 1
loadc 3	loadr 1	loadr 1	loadr 1	storer -2
new	store	loadc 1	loadc 2	return
storer 1	pop	add	add	
pop		store	store	
		pop	pop	

loadrc 1
loadc 0
add

The translation of the body amounts to:

alloc 1	newMutex	newCondVar	loadr -2	loadr 1
loadc 3	loadr 1	loadr 1	loadr 1	storer -2
new	store	loadc 1	loadc 2	return
storer 1	pop	add	add	
pop		store	store	
		pop	pop	

The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

The function Down() decrements the counter.

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

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


The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

452

The function Down() decrements the counter.

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

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

The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

The function Down() decrements the counter.

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

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


The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

452

The function Down() decrements the counter.

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

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

451

The translation of the body amounts to:

alloc 1	loadc 2	add	loadc 1
loadr -2	add	store	add
load	load	loadc 0	load
storer 1	loadc 1	leq	signal
lock	add	jumpz A	A: loadr 1
	loadr -2		unlock
loadr -2	loadc 2	loadr -2	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→me;
    lock (me);
    s→count++;
    if (s→count ≤ 0)   signal (s→cv);
    unlock (me);
}
```


56 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); }
```


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→me;
    lock (me);
    s→count++;
    if (s→count ≤ 0)   signal (s→cv);
    unlock (me);
}

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→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
    }
```


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→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
    }
```

453

The translation of the body amounts to:

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

The translation of the body amounts to:

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

454

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:

$$
\begin{array}{l}
\textbf{void}\ \mathsf{Up}\ (\mathsf{Sema} * \mathsf{s})\ \{ \\
\qquad \textbf{Mutex}\ {*}\mathsf{me}; \\
\qquad \mathsf{me} = \mathsf{s}{\rightarrow}\mathsf{me}; \\
\qquad \textbf{lock}\ (\mathsf{me}); \\
\qquad \mathsf{s}{\rightarrow}\mathsf{count}{+}{+}; \\
\qquad \textbf{if}\ (\mathsf{s}{\rightarrow}\mathsf{count} \leq 0)\quad \textbf{signal}\ (\mathsf{s}{\rightarrow}\mathsf{cv}); \\
\qquad \textbf{unlock}\ (\mathsf{me}); \\
\}
\end{array}
$$

56 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:

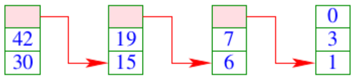
- Maintain all stacks in one joint Frame-Heap FH :-)
- Take care that the space inside the stack frame is sufficient at least for the current function call.
- A global stack-pointer GSP points to the overall topmost stack cell ...

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.

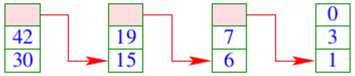
- If the block is on top of the stack, we pop the stack immediately;
- ... together with the blocks below – given that these have already been marked as de-allocated.
- If the block is inside the stack, we merge it with neighbored free blocks:

Warning:

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

$\Longrightarrow$

We maintain a list of freed stack blocks :-)

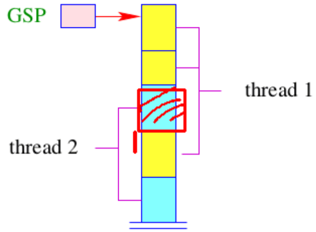


This list supports a function

```
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:



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

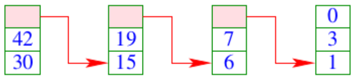
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

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 :-)

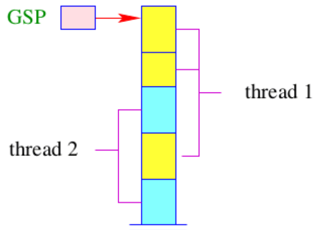


This list supports a function

```
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:



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

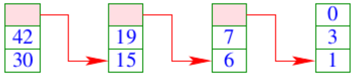
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

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:

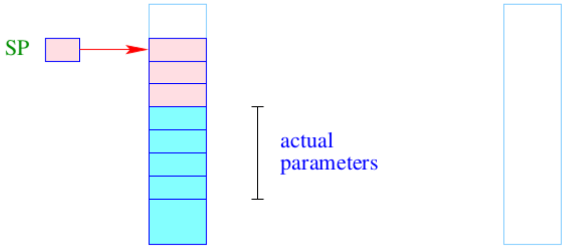
Approach:

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

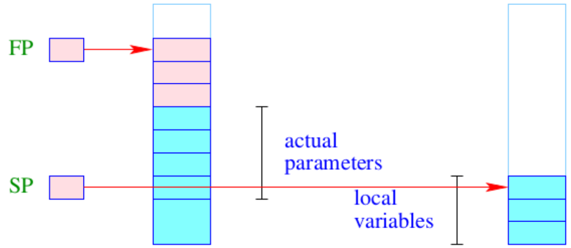
Problem:

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!



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 ... :-)

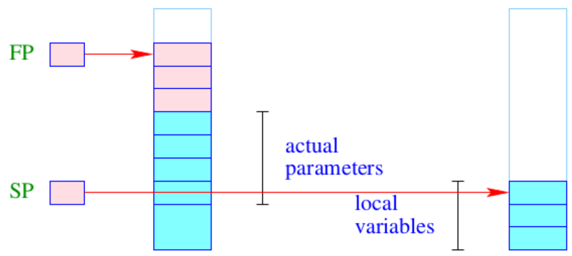
$\Longrightarrow$ We address ...

- 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 ... :-)

466



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