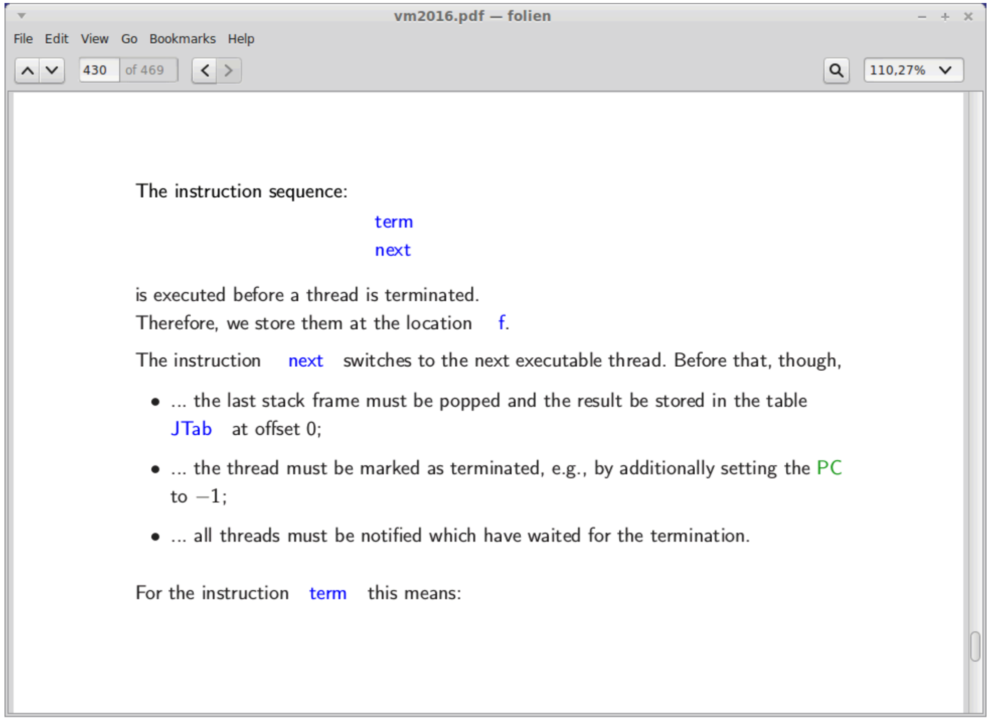
Script **generated by TTT**

Title: Seidl: Virtual_Machines (27.06.2016)

Date: Mon Jun 27 10:22:47 CEST 2016

Duration: 90:13 min

Pages: 53



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:

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 ≤ 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;

Caveat

A thread is only allowed to release a mutex if it has owned it beforehand.

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;

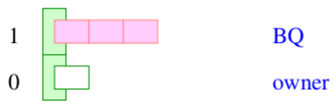
Caveat

A thread is only allowed to release a mutex if it has owned it beforehand.

432

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:

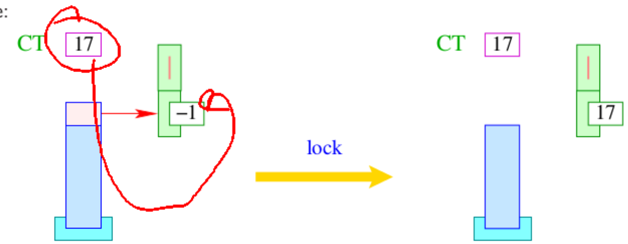
$$\text{code}_R\ \textbf{newMutex}\ ()\ \rho \quad = \quad \text{newMutex}$$

where:

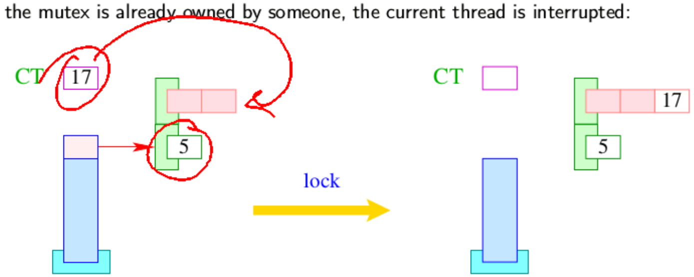
Then we translate:

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

where:



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

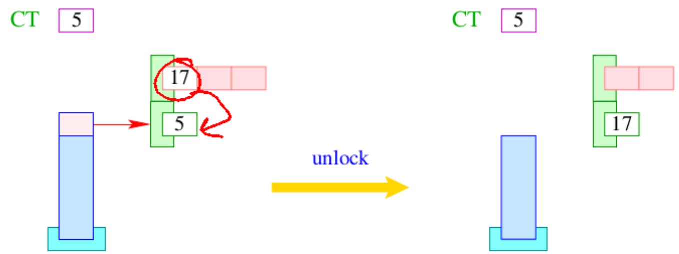



```
if (S[S[SP]] < 0)   S[S[SP– –]] = CT;
else {
        enqueue ( S[SP– –]+1, CT );
        next;
     }
```

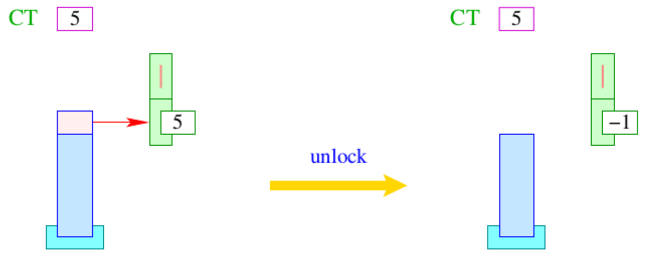

Accordingly, we translate:

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

where:

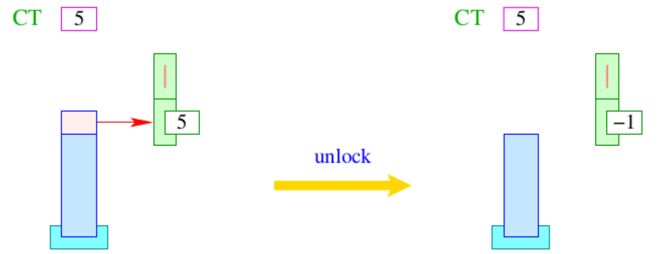


If the queue BQ is empty, we release the mutex:



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:

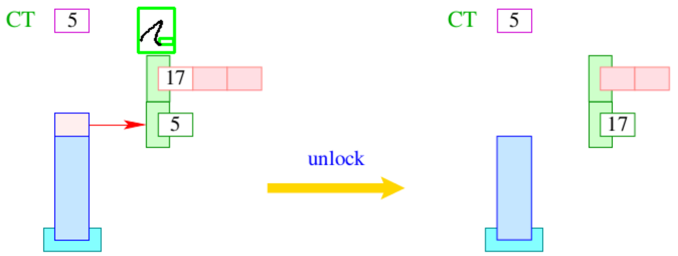


if (S[S[SP]] $\neq$ CT) Error ("Illegal unlock!");
if (0 > tid = dequeue (S[SP]+1)) S[S[SP– –]] = –1;
else {
S[S[SP--]] = tid;
enqueue (RQ, tid);
}

Accordingly, we translate:

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

where:



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.

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.

440

Then we translate:

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

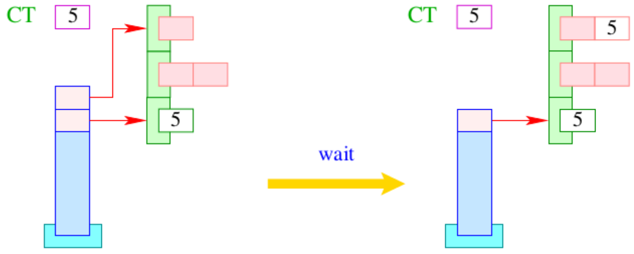
where:

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

Therefore, we translate:

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

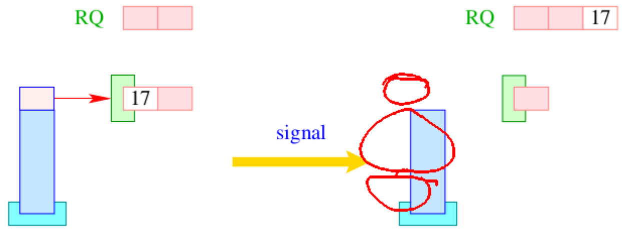
where ...



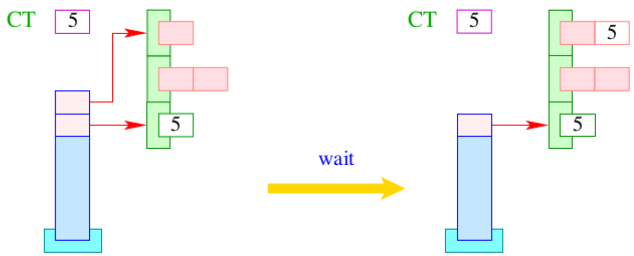
if (S[S[SP-1]] $\neq$ CT) Error ("Illegal wait!");
enqueue (S[SP], CT); SP--;

Accordingly, we translate:

$$\textsf{code } \textbf{signal } (e);\ \rho \quad = \quad \textsf{code}_{\textsf{R}}\ e\ \rho$$
$$\textbf{signal}$$



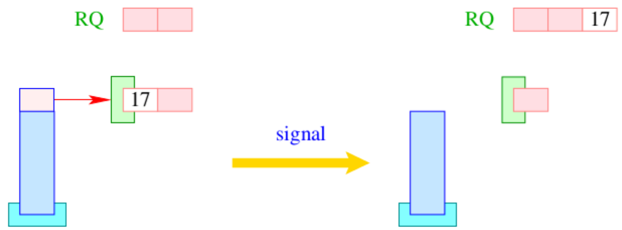
if (0 $\leq$ tid = dequeue (S[SP]))
    enqueue (RQ, tid);
SP--;



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

Accordingly, we translate:

$$\textsf{code } \textbf{signal } (e);\ \rho \quad = \quad \textsf{code}_{\textsf{R}}\ e\ \rho$$
$$\textbf{signal}$$



if (0 $\leq$ tid = dequeue (S[SP]))
    enqueue (RQ, tid);
SP--;

Analogously:

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

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--;
```

Caveat

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;
```



```
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	


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



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

448

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	

449

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
```


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);
    }
```

450

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
```

451

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 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);
}

452

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

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

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); }
```


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); }

454

Problem

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

2. Idea

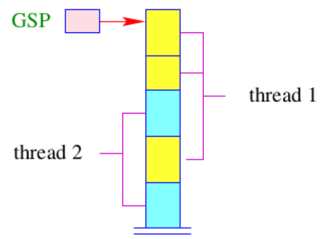
- Maintain all stacks in one joint Frame-Heap FH.
- Take care that the space inside the stack frame is sufficient at least for the current function call.
- A global stack-pointer GSP points to the overall topmost stack cell ...

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:

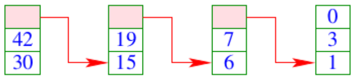
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

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


Caveat

The de-allocated block may reside inside the stack!

⟹

We maintain a list of freed stack blocks.

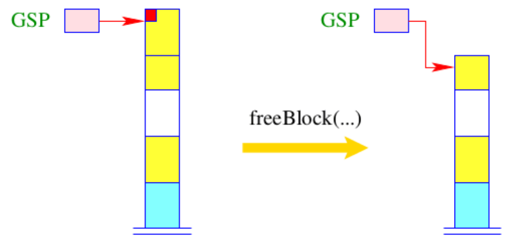


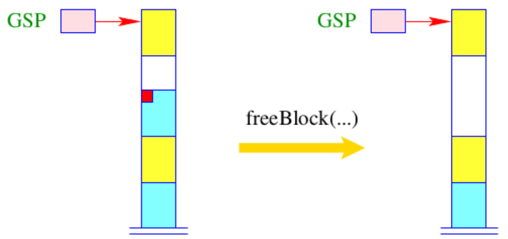
This list supports a function

```
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:





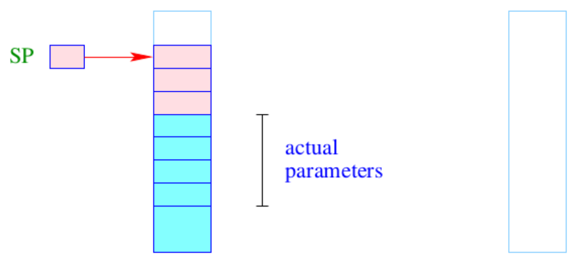
Approach

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

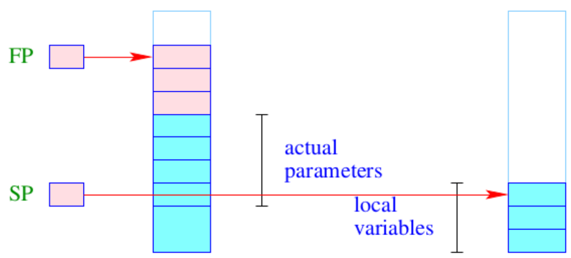
Problem

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 ...

⟹ We address ...

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

⟹ We must re-organize the complete code generation ...

Alternative: Passing of parameters in registers ...

⟹ We address ...

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

⟹ We must re-organize the complete code generation ...

Alternative: Passing of parameters in registers ...

465