Script **generated by TTT**

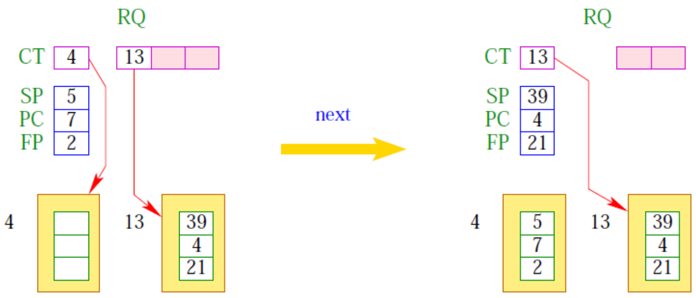
Title: Seidl: Virtual_Machines (26.06.2013)

Date: Wed Jun 26 16:03:20 CEST 2013

Duration: 87:05 min

Pages: 48

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



Therefore, we translate:

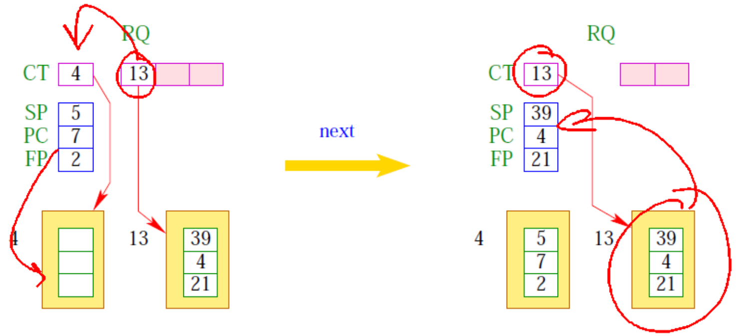
$$\text{code } \mathbf{exit}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
exit
term
next

The instruction term is explained later :-)

The instruction exit successively pops all stack frames:

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


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



If the queue RQ is empty, we additionally terminate the whole program:

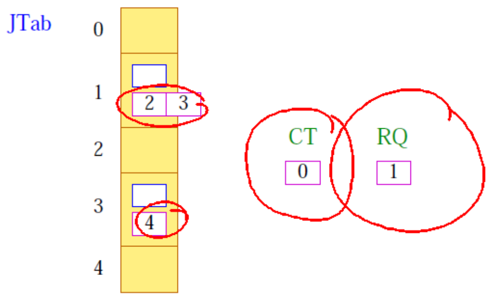
```
if (0 > ct = dequeue( RQ ))    halt;
else {
        save ();
        CT = ct;
        restore ();
    }
```


51 Waiting for Termination

Occationally, 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 evaluates 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.

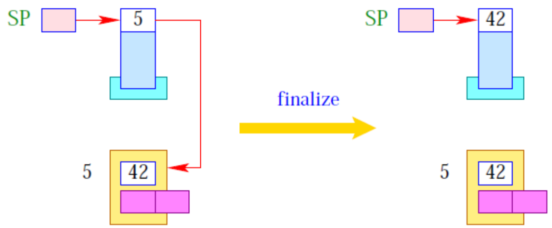
Thus, we translate:

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

... where the instruction join is defined by:

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

... accordingly:



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

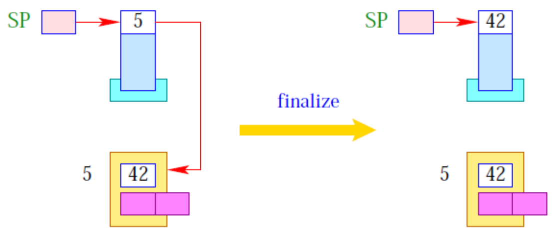
Thus, we translate:

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

... where the instruction join is defined by:

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


... accordingly:



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

416

Thus, we translate:

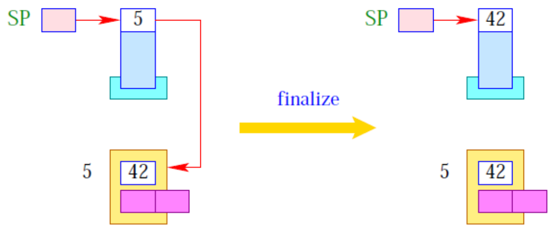
$$\begin{aligned} \text{code}_R\ \textbf{join}\ (e)\ \rho \quad = \quad & \text{code}_R\ e\ \rho \\ & \text{join} \\ & \text{finalize} \end{aligned}$$

... where the instruction join is defined by:

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

415

... 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:

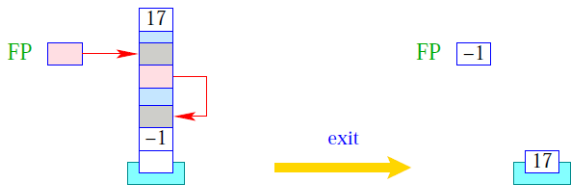
52 Mutual Exclusion

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:

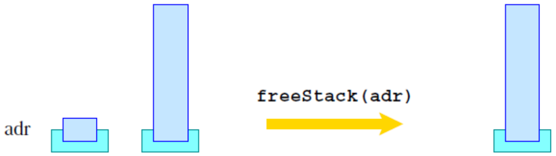
Mutex $*$ newMutex ();	—	creates a new mutex;
void lock (**Mutex** $*$me);	—	tries to acquire the mutex;
void unlock (**Mutex** $*$me);	—	releases the mutex;

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



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 :



The instruction sequence:

term
next

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

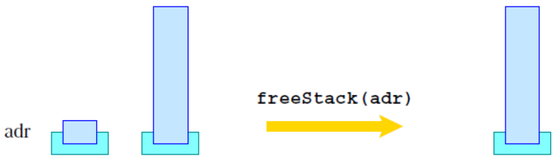
The instruction next switches to the next executable thread. Before that, though,

- ... 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 :



52 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 :-)

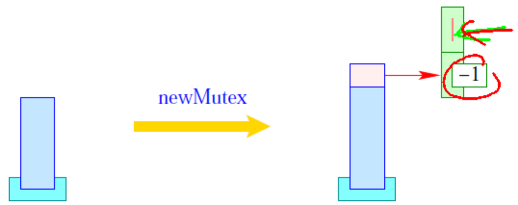
A mutex me consists of:

- 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:

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

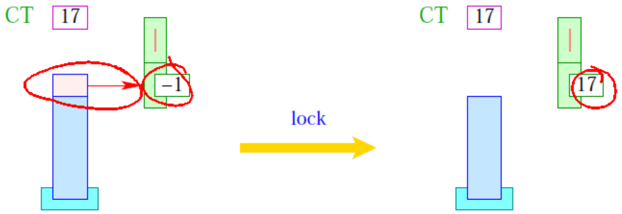
where:



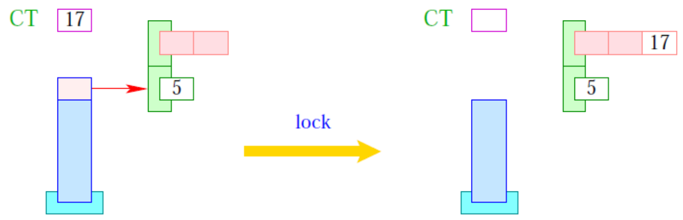
Then we translate:

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

where:



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

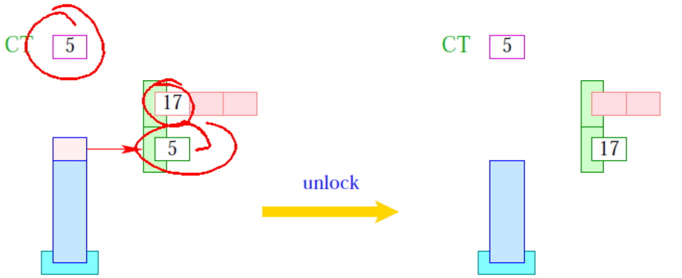



```
if (S[S[SP]] < 0)   S[S[SP--]] = CT;
else {
        enqueue ( S[SP--]+1, CT);
        next;
    }
```

Accordingly, we translate:

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

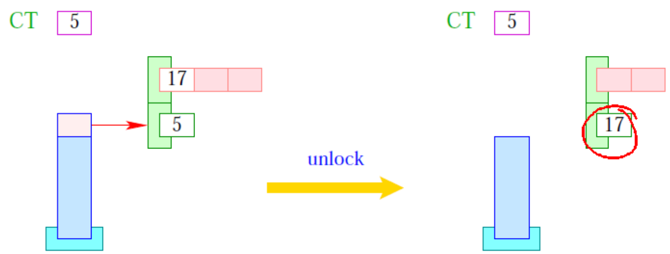
where:



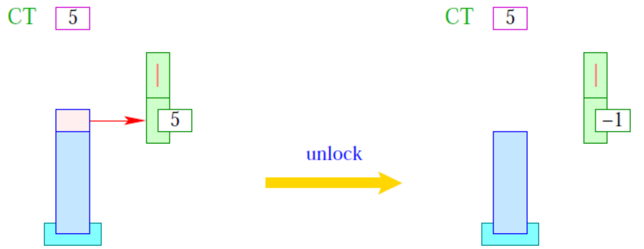
Accordingly, we translate:

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

where:

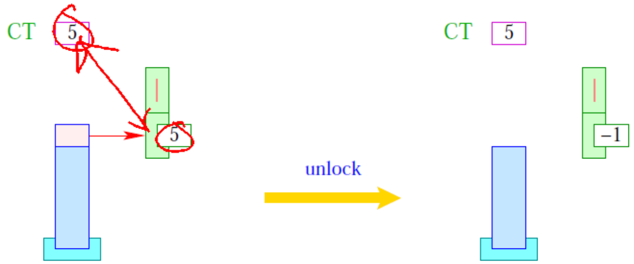


If the queue BQ is empty, we release the mutex:



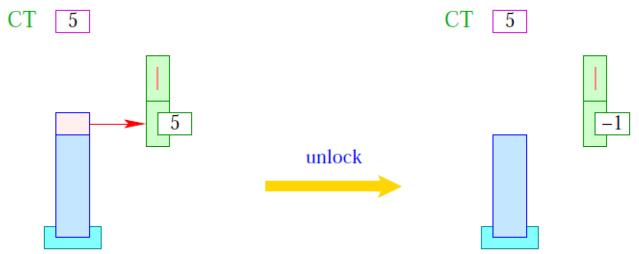

```
if (S[S[SP]] ≠ CT)   Error ("Illegal unlock!");
if (0 > tid = dequeue ( S[SP]+1))    S[S[SP−−]] = −1;
else {
        S[S[SP--]] = tid;
        enqueue ( RQ, tid );
    }
```


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




```
if (S[S[SP]] ≠ 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);
}

53 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:

code **wait** $(e_0, e_1);\ \rho$ = code$_R$ $e_1\ \rho$
code$_R$ $e_0\ \rho$
wait
dup
unlock
next
lock

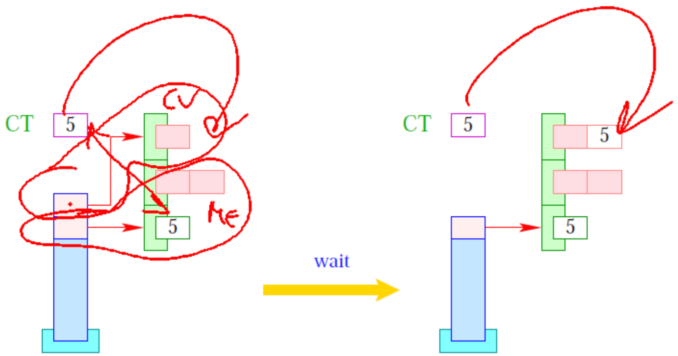
where ...

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

Therefore, we translate:

code **wait** $(e_0, e_1);\ \rho$ = code$_R$ $e_1\ \rho$
code$_R$ $e_0\ \rho$
wait
dup
unlock
next
lock

where ...



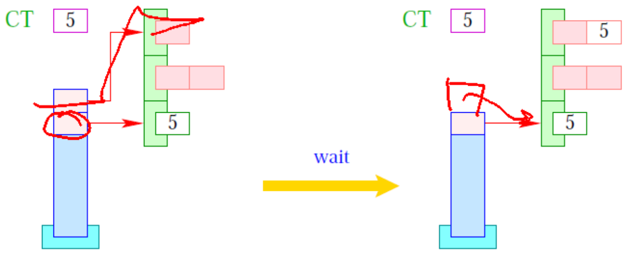
if (S[S[SP-1]] $\neq$ CT) Error ("Illegal wait!");
enqueue (S[SP], CT); SP--;

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

Therefore, we translate:

code **wait** $(e_0, e_1);\ \rho$ = code$_R$ $e_1\ \rho$
code$_R$ $e_0\ \rho$
wait
dup
unlock
next
lock

where ...



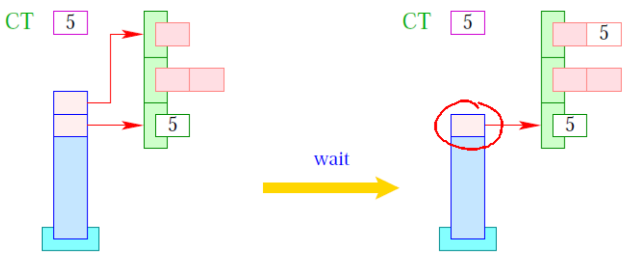
if (S[S[SP-1]] ≠ CT)   Error ("Illegal wait!");
enqueue (S[SP], CT); SP--;

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

Therefore, we translate:

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

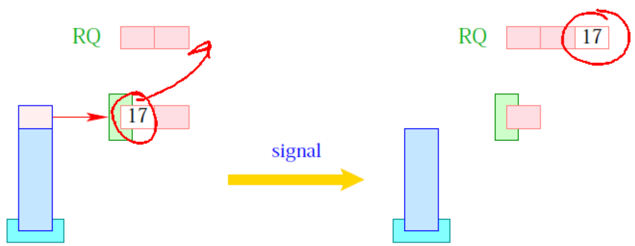
where ...



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

Accordingly, we translate:

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



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

Analogously:

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

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--;
```

Warning:

The re-animated threads are not blocked !!!

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

54 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);
    }
```

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

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;
```

The translation of the body amounts to:

```
alloc 1
loadc 3
new
storer 1
pop

newMutex
loadr 1
store
pop

newCondVar
loadr 1
loadc 1
add
store
pop

loadr -3
loadr 1
loadc 2
add
store
pop

loadr 1
storer -3
return
```