**Script** generated by TTT

Title: Seidl: Virtual Machines (24.06.2014)

Date: Tue Jun 24 10:15:35 CEST 2014

Duration: 89:02 min

Pages: 55

Therefore, we translate:

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

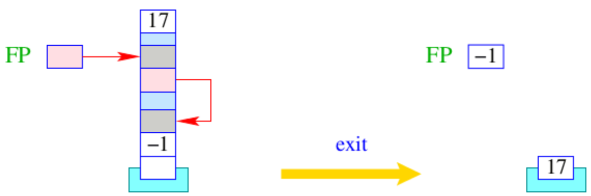
exit
term
next

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;
```








Therefore, we translate:

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

exit
term
next

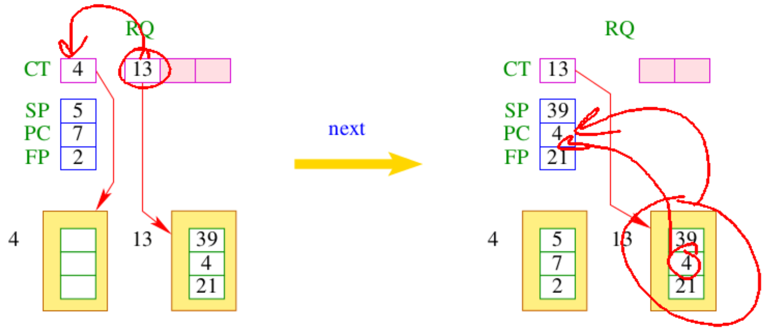
The instruction term is explained later :-)

The instruction exit successively pops all stack frames:

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

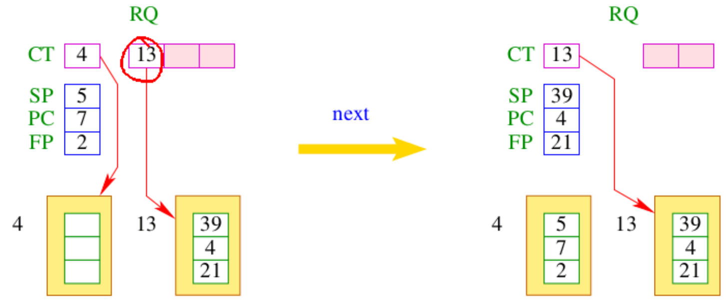
421

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






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

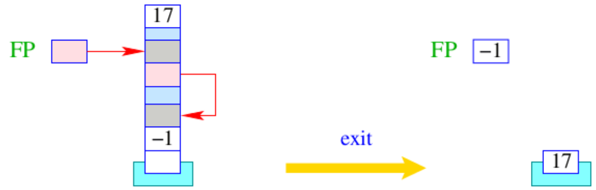





If the queue RQ is empty, we additionally terminate the whole program:

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





# 52 Waiting for Termination

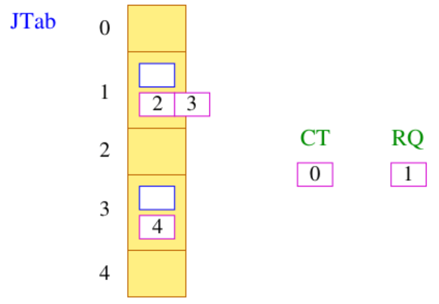
Occationally, a thread may only continue with its execution, if some other thread has terminated. For that, we have the expression **join** $(e)$ where we assume that $e$ evaluates to a thread id tid.

- 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:

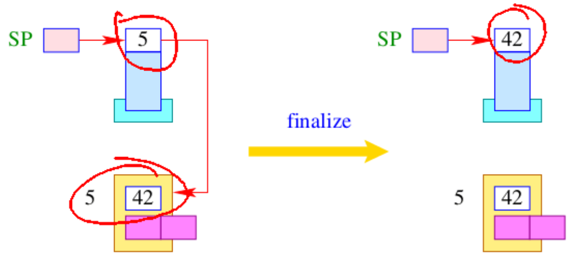
$$\text{code}_R\ \textbf{join}\ (e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{join}$$
$$\text{finalize}$$

... 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];

## 52 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 :-)



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
    }
```



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; |

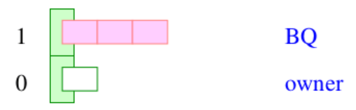
Warning:

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.

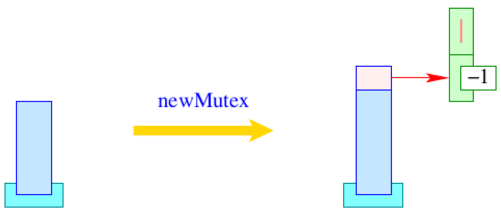





Then we translate:

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

where:

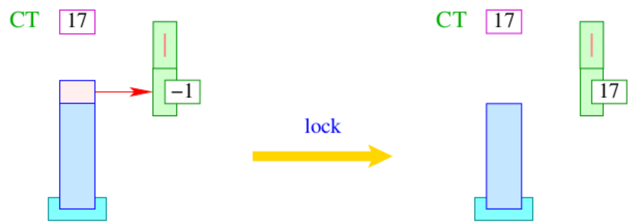





Then we translate:

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

where:



A mutex me consists of:

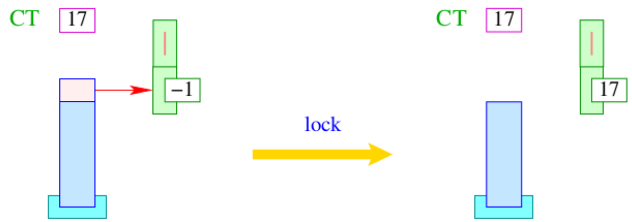
- the tid of the current owner (or −1 if there is no one);
- the queue BQ of blocked threads which want to acquire the mutex.




Then we translate:

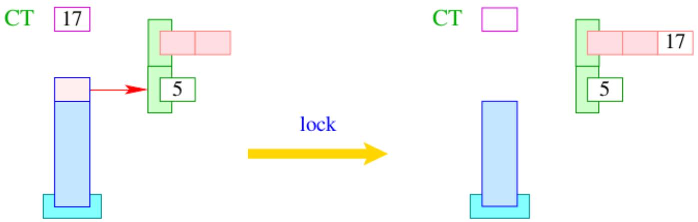
$$\text{code } \mathbf{lock}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\qquad\qquad\qquad\qquad \text{lock}$$

where:





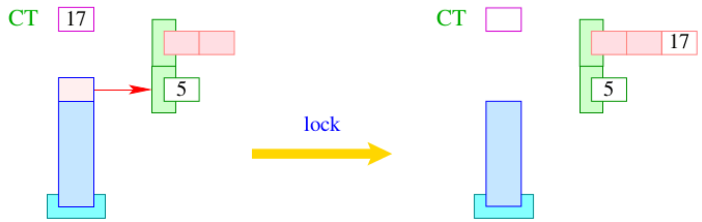

If the mutex is already owned by someone, the current thread is interrupted:




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



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



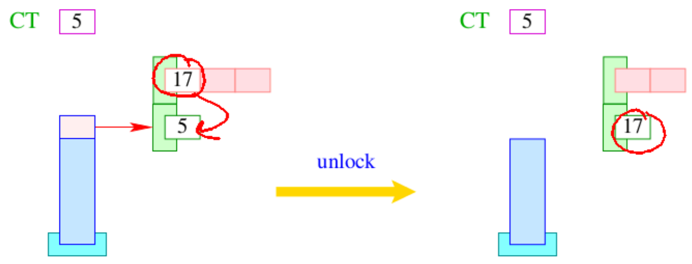

```
if (S[S[SP]] < 0)   S[S[SP−−]] = CT;
else {
        enqueue ( S[SP−−]+1, CT );
        next;
     }
```

435

Accordingly, we translate:

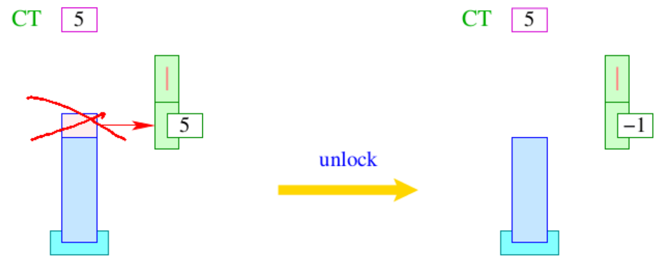
code **unlock** $(e)$; $\rho$ = $\text{code}_R\ e\ \rho$
unlock

where:






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



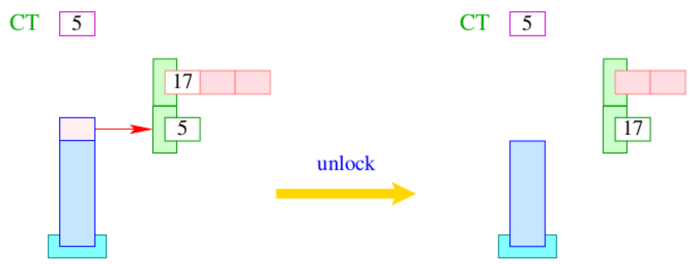

```
if (S[S[SP]] ≠ CT)   Error ("Illegal unlock!");
if (0 > tid = dequeue ( S[SP]+1))   S[S[SP--]] = –1;
else {
        S[S[SP--]] = tid;
        enqueue ( RQ, tid );
}
```



Accordingly, we translate:

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

where:

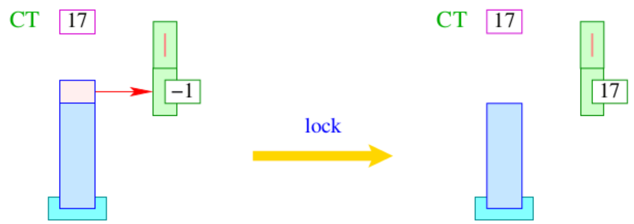



436

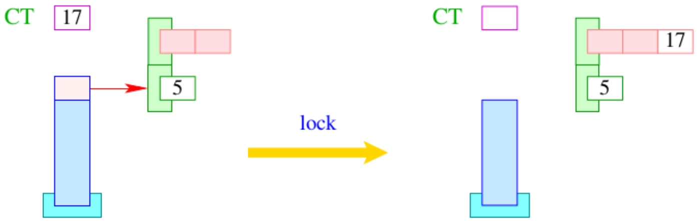
Then we translate:

code **lock** $(e)$; $\rho$ = $\text{code}_R\ e\ \rho$
lock

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;
    }
```



## 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:

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

where ...



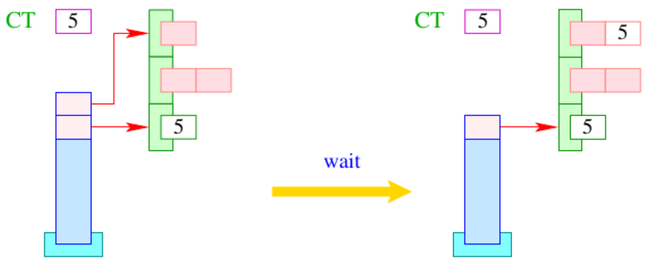
After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

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

where ...

441




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



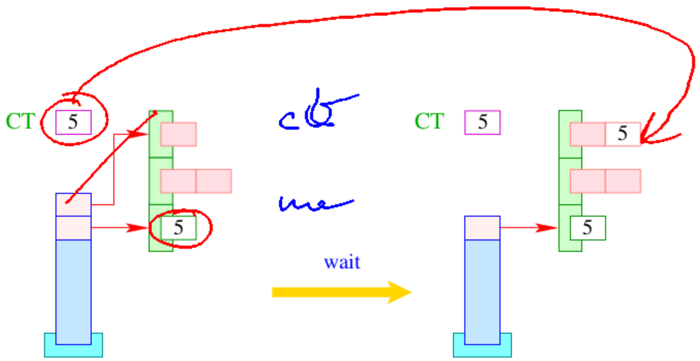
After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

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

where ...

441




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



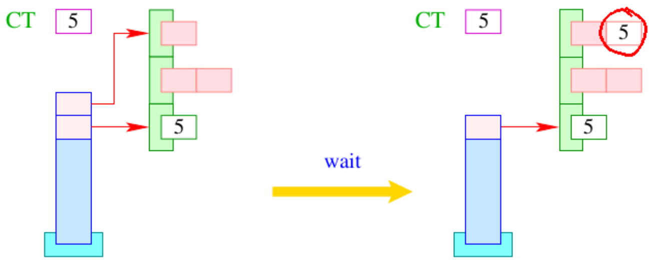
After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

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

where ...






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

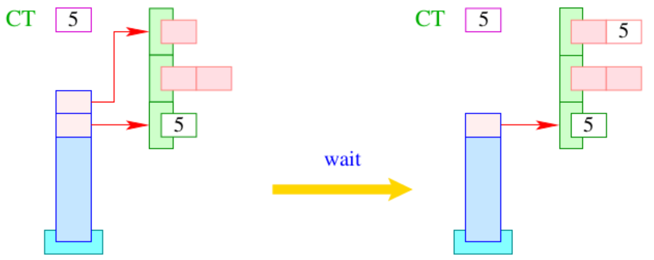


After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

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

where ...

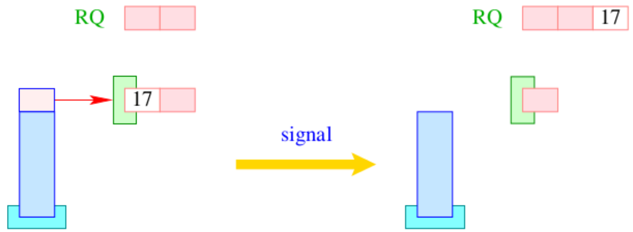



if (S[S[SP-1]] $\neq$ CT) Error ("Illegal wait!");
enqueue ( S[SP], CT ); SP--;



Accordingly, we translate:

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




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 ($0 \leq$ 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 :-)



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 ($0 \leq$ 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 :-)

444

## 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);
    }
```



Therefore, a semaphore consists of:

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

```
typedef struct {
    0   Mutex * me;
    1   CondVar * cv;
    2   int count;
    } Sema;
```

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);
    }
```



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 | |

448

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

447

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);
    }
```



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 | |

448

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);
    }
```