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 $adr$:
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:

\[
\begin{align*}
\text{Mutex} & \text{ newMutex ();} & \quad \text{— creates a new mutex;} \\
\text{void} & \text{ lock (Mutex \*me);} & \quad \text{— tries to acquire the mutex;} \\
\text{void} & \text{ unlock (Mutex \*me);} & \quad \text{— releases the mutex;}
\end{align*}
\]

Caveat
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.

\[
\begin{array}{c}
1 \\
0 \\
\end{array}
\begin{array}{c}
\text{BQ} \\
\text{owner}
\end{array}
\]

Then we translate:

\[
\text{code: newMutex () \ p = newMutex}
\]

where:

\[
\text{newMutex}
\]
Then we translate:

\[
\text{code} \ e \ \text{newMutex}() \ \rho = \ \text{newMutex}
\]

where:

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

Accordingly, we translate:

\[
\text{code} \ e \ \text{lock}() \ \rho = \ \text{coder} \ e \ \rho
\]

where:

```
lock
```
Append the sequence:

```
438

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

439

Accordingly, we translate:

```
54 Waiting for Better Weather

code unlock (e); ρ = codeρ e ρ
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 of waiting threads.

0 WQ
```
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:

```plaintext
code newCondVar () $\rho = \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:

```plaintext
code wait ($e_2, e_1$); $\rho = \text{code}\ e_1\ \rho$
```

```plaintext
code lock
```

`$\rightarrow$`

```plaintext
code unlock
```

```plaintext
code wait
```

```plaintext
\text{next}
```

`$\rightarrow$`

```plaintext
\text{lock}
```

where ...
Accordingly, we translate:

\[
\text{code signal}(\rho) \quad \rho = \text{code}_{\text{R}}\rho \\
\text{signal}
\]

RQ \[17]

\[
\text{if (0 \leq \text{tid} = \text{enqueue (S[SP])})} \\
\text{enqueue (RQ, tid)}; \\
\text{SP}--;
\]

Accordingly, we translate:

\[
\text{code signal}(\rho) \quad \rho = \text{code}_{\text{R}}\rho \\
\text{signal}
\]

RQ \[17]

\[
\text{if (0 \leq \text{tid} = \text{enqueue (S[SP])})} \\
\text{enqueue (RQ, tid)}; \\
\text{SP}--;
\]
Analogously:

\[
\text{code broadcast \((e); \rho = \text{code}_{\rho} e \rho \\
\text{broadcast)}
\]

where the instruction \text{broadcast} enqueues all threads from the queue \text{WQ}
into the ready-queue \text{RQ}:

\[
\text{while } (0 \leq \text{tid} = \text{dequeue} (\text{S}[\text{SP}]))
\text{enqueue} (\text{RQ}, \text{tid});
\text{SP}--;
\]

Caveat
The re-animated threads are not blocked. !!!
When they become running, though, they first have to acquire their mutex.

Therefore, a semaphore consists of:

- a \textbf{counter} of type \text{int};
- a \textbf{mutex} for synchronizing the semaphore operations;
- a \textbf{condition variable}.

\[
\text{typedef struct } \\
\text{ 
   Mutex * } \text{me};  \\
\text{CondVar * } \text{cv};  \\
\text{int count;}  \\
\text{Sema;}
\]

55 Example: Semaphores

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

\textbf{Operations}

\[
\begin{align*}
\text{Sema + newSema (int } n \text{)} & \quad \text{— creates a new semaphore;} \\
\text{void Up (Sema * } s \text{)} & \quad \text{— increases the number of free resources;} \\
\text{void Down (Sema * } s \text{)} & \quad \text{— decreases the number of available resources.}
\end{align*}
\]

\[
\text{Sema * newSema (int } n \text{)} \\
\text{Sema + } s;  \\
\text{s = (Sema *) malloc (sizeof (Sema));}  \\
\text{s->me = newMutex ();}  \\
\text{s->cv = newCondVar ();}  \\
\text{s->count = } n;  \\
\text{return } (s);  \\
\]

445 446

447 448
The translation of the body amounts to:

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc 1</td>
<td>newMutex</td>
</tr>
<tr>
<td>loadc 3</td>
<td>loadr 1</td>
</tr>
<tr>
<td>new</td>
<td>store</td>
</tr>
<tr>
<td>storer 1</td>
<td>pop</td>
</tr>
<tr>
<td>pop</td>
<td>store</td>
</tr>
<tr>
<td>pop</td>
<td>pop</td>
</tr>
</tbody>
</table>

```c
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc 1</td>
<td>newMutex</td>
</tr>
<tr>
<td>loadc 3</td>
<td>loadr 1</td>
</tr>
<tr>
<td>loadc 2</td>
<td>loadr -2</td>
</tr>
<tr>
<td>new</td>
<td>store</td>
</tr>
<tr>
<td>storer 1</td>
<td>pop</td>
</tr>
<tr>
<td>pop</td>
<td>store</td>
</tr>
<tr>
<td>pop</td>
<td>pop</td>
</tr>
</tbody>
</table>
The function `Down()` **decrements** the counter.

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

```c
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 `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:

```c
void Up (Sema * s) {
    Mutex *me;
    me = s->me;
    lock (me);
    s->count++; // Corrected: `s->count++` instead of `s->count+i`.
    if (s->count ≤ 0) signal (s->cv);
    unlock (me);
}
```

The translation of the body amounts to:

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

Then we implement:

```c
void *newStack() { return malloc(N); }
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 ...
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 ...

Caveat

The de-allocated block may reside inside the stack!

We maintain a list of freed stack blocks.

<table>
<thead>
<tr>
<th>42</th>
<th>19</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>15</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

This list supports a function

```c
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.

We order the new block after entering the function body!

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

In particular, 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 ...