Therefore, we translate:

\[
\text{code exit (c); } \rho = \text{code}\_e \rho \\
\text{exit} \\
\text{term} \\
\text{next}
\]

The instruction `term` is explained later. 

The instruction `exit` successively pops all stack frames:

\[
\text{result} = S[\text{SP}]; \\
\text{while} (\text{FP} \neq -1) \{ \\
\quad \text{SP} = \text{FP} - 2; \\
\quad \text{FP} = S[\text{FP} - 1]; \\
\} \\
S[\text{SP}] = \text{result};
\]
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:

```c
if (0 > ct = dequeue(RQ))
  halt;
else {
  sleep(1);  // ct = ct
  restore();
}
```
52 Waiting for Termination

Occasionally, a thread may only continue with its execution, if some other thread has terminated. For that, we have the expression \texttt{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 threads 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 \(\texttt{JTab}\).
- There, we also store the return values of threads \(\therefore\)

Thus, we translate:

\[
\text{code}_0: \text{join}(e) = \text{code}_1: e \rho
\]

\[
\text{join}
\]

\[
\text{finalize}
\]

... where the instruction \texttt{join} is defined by:

\[
\text{tid} = \text{SP} + l_1
\]

\[
\text{if (JTab[tid][1] } \geq 0 \{
\text{enque} (\text{JTab[tid][1], CT });
\text{next}
\}
\]

\[
S[\text{SP}] = \text{JTab[tid][0]};
\]
Thus, we translate:

\[
\text{code} \Join \ (e \rho) = \text{code} (e \rho) \\
\begin{array}{l}
\text{join} \\
\text{finalize}
\end{array}
\]

... where the instruction \text{join} is defined by:

\[
\text{tid} = S[SFP] \\
\text{if} (TTab[tid][1] \geq 0) \\
\text{enqueu}e (TTab[tid][1], CT) \\
\text{next}
\]

52 Waiting for Termination

Occasionally, a thread may only continue with its execution, if some other thread has terminated. For that, we have the expression \text{join} \ (e) where we assume that \( e \) evaluates to a thread id \( \text{tid} \).

- 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 threads 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 \( TTab \).
- There, we also store the return values of threads \( \Rightarrow \)

The instruction sequence:

\[
\text{term} \\
\text{next}
\]

is executed before a thread is terminated.

Therefore, we store them at the location \( f \).

The instruction \text{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 \( TTab \) 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 \text{term} this means:

\[
\text{PC} = -1; \\
\text{TTab}[CT][0] = S[SFP]; \\
\text{freeStack}(SP); \\
\text{while}(0 \leq \text{tid} = \text{enqueu}e(\text{TTab}[CT][1])) \\
\text{enqueu}e(\text{RQ}, \text{tid})
\]

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

```
kode2 newMutex () ρ = newMutex
```
where:

```
newMutex
```

Then we translate:

```
code lock (e); ρ = code[ e ρ ]
```
where:

```
CT 17
```

```
lock
```

```
CT 17
```
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:

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

where:

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

\[
\text{if } S(SP[0] < 0) \text{ and } S(SP[-1]) = CT; \text{ then continue; else return.}
\]

432

434

435
Accordingly, we translate:

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

where:

Accordingly, we translate:

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

where:

Then we translate:

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

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

```c
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 inactive until it is told otherwise.
For that, we use condition variables. A condition variable consists of a queue `WQ` of waiting threads.

Then we translate:

```c
code: newCondVar ()  \rho = newCondVar
```
where:

- `newCondVar` creates a new condition variable;
- `wait` enqueues the current thread;
- `signal` re-animates one waiting thread;
- `broadcast` re-animates all waiting threads.
After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

\[
\text{code } \text{wait } (e_0, e_1); \rho = \text{decode } e_1 \rho \\
\text{decode } e_0 \rho \\
\text{wait} \\
\text{dup} \\
\text{unlock} \\
\text{next} \\
\text{lock}
\]

where ...

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

Therefore, we translate:

\[
\text{code } \text{wait } (e_0, e_1); \rho = \text{decode } e_1 \rho \\
\text{decode } e_0 \rho \\
\text{wait} \\
\text{dup} \\
\text{unlock} \\
\text{next} \\
\text{lock}
\]

where ...

if (S[S[SP-1]] \neq \text{CT}) Error (“Illegal wait!”); enqueue ( S[SP], \text{CT} ); SP--;
After enqueuing the current thread, we release the mutex. After re-animation, though, we must acquire the mutex again.

Therefore, we translate:

\[
\text{code } \text{wait} (e_0, e_1); \rho = \text{codeg } e_1 \rho \\
\text{codeg } e_2 \rho \\
\text{wait} \\
\text{dup} \\
\text{unlock} \\
\text{next} \\
\text{lock}
\]

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

Accordingly, we translate:

\[ \text{code signal}(e); \ r = \text{code}_{\text{e}} \ e \ r \text{ signal} \]

\[ \text{RQ} \]

\[ \text{signal} \]

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

443

Analogously:

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

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

\[ \text{broadcast} \]

\[ \text{WQ} \]

\[ \text{RQ} : \]

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

\[ \text{Warning:} \]

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

444

Analogously:

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

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

\[ \text{broadcast} \]

\[ \text{WQ} \]

\[ \text{RQ} : \]

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

\[ \text{Warning:} \]

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.

```c
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.

```c
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.

```c
typedef struct {
    Mutex * me;
    CondVar * cv;
    int count;
} Sema;
```
The translation of the body amounts to:

```
alloc 1  newMutex  newCondVar  loadr-2  loadr 1
loadr 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        store
pop        pop        store
```

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

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);
}
```