Therefore, we translate:

\[
\text{code exit (c); } \rho \begin{array}{c} \text{exit} \\
\text{term} \\
\text{next} \end{array}
\]

The instruction \text{term} is explained later.

The instruction \text{exit} successively pops all stack frames:

\[
\text{result} = S[SP];
\text{while } (\text{FP} \neq -1) \{
\text{SP} = \text{FP}-2;
\text{FP} = S[\text{FP}-1];
\}
\text{S[SP] = result;
\]
51 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 \( tid \).

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

Thus, we translate:

\[
\text{codeg \ join}(e) \rho = \text{codeg \ } e \rho
\]

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

\[
tid = S[SP];
\]
\[
\text{if } (JTab[tid][1] \geq 0) \{
\text{enqueue} \{ JTab[tid][1], CT \};
\text{next}
\}
\]

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:

```
codeg join (e) \rho = codeg e \rho
    join
    finalize
```

... where the instruction `join` is defined by:

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

Thus, we translate:

```
codeg join (e) \rho = codeg e \rho
    join
    finalize
```

... where the instruction `join` is defined by:

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

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

\[
\text{PC} = -1; \\
\text{JTab}[\text{CT}][0] = S[\text{SP}]; \\
\text{freeStack}(\text{SP}); \\
\text{while}(0 \leq \text{tid} = \text{dequeue}(\text{JTab}[\text{CT}][1])) \\
\text{enqueue}(\text{RQ}, \text{tid});
\]

The instruction sequence:

1. \( \text{term} \)
2. \( \text{next} \)

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 \( \text{JTab} \) at offset 0;
- the thread must be marked as terminated, e.g., by additionally setting the \( \text{PC} \) to \(-1\);
- all threads must be notified which have waited for the termination.

For the instruction \( \text{term} \) this means:

52 Mutual Exclusion

A \text{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:

- \( \text{Mutex} + \text{newMutex}() \); — creates a new mutex;
- \( \text{void lock (Mutex *me)} \); — tries to acquire the mutex;
- \( \text{void unlock (Mutex *me)} \); — releases the mutex;

Warning:
A thread is only allowed to release a mutex if it has owned it beforehand. 😃
A mutex consists of:

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

![](image1)

Then we translate:

\[
\text{cdecl newMutex}() \rho = \text{newMutex}
\]

where:

![](image2)

Then we translate:

\[
\text{cdecl lock}() \rho = \text{lock}
\]

where:

![](image3)

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

\[
\text{if} (S[S[SP]] < 0) \quad S[S[SP-1]] = CT;
\text{else}
\{\text{enqueue } S[SP-1] = CT;\}
\]

where:

![](image4)
Accordingly, we translate:

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

\[
\text{unlock}
\]

where:

\[
\begin{align*}
&\text{CT } 5 \\
&\text{17} \\
&\text{5}
\end{align*}
\]

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

\[
\begin{align*}
&\text{CT } 5 \\
&\text{1} \\
&\text{5} \\
&\text{unlock}
\end{align*}
\]

\[
\begin{align*}
&\text{CT } 5 \\
&\text{1} \\
&\text{1} \\
&\text{unlock}
\end{align*}
\]

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

\[
\begin{align*}
&\text{S[SP−]} = \text{tid}; \\
&\text{enqueue (RQ, tid)};
\end{align*}
\]

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

Therefore, we translate:

\[
\text{code wait } (p, c_1); p = \text{codeg } e_3 p \\
\text{codeg } e_2 p \\
\text{wait} \\
\text{dup} \\
\text{unlock} \\
\text{next} \\
\text{lock}
\]

where ...

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

Therefore, we translate:

\[
\text{code wait } (p_2, c_2); p = \text{codeg } e_3 p \\
\text{codeg } e_2 p \\
\text{wait} \\
\text{dup} \\
\text{unlock} \\
\text{next} \\
\text{lock}
\]

where ...

if \(\text{S}[\text{SP-1}] \neq \text{CT}\)  Error ("Illegal wait!");

enqueue \((\text{S}[\text{SP}], \text{CT}); \text{SP}:=\)

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

Therefore, we translate:

\[ \text{code wait } (e_0, e_1); \rho = \text{code } e_1 \rho \]

\[ \text{code } e_0 \rho \]

\[ \text{wait} \]

\[ \text{dup} \]

\[ \text{unlock} \]

\[ \text{next} \]

\[ \text{lock} \]

where ...

Accordingly, we translate:

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

\[ \text{signal} \]

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

enqueue (S[SP], CT); SP~;

if (0 ≤ tid = dequeue (S[SP]))

enqueue (RQ, tid);

SP~;
Analogously:

\[
\text{code broadcast } (x); \rho = \text{code } \tau \rho
\]

\[
\text{broadcast}
\]

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

\[
\text{while } (0 \leq \text{tid} \leq \text{dequeue } (\text{SP}))
\]

\[
\text{enqueue } (\text{RQ}, \text{tid});
\]

\[
\text{SP}--;
\]

\textbf{Warning:}
The re-animated threads are not \texttt{blocked} !!!

When they become running, though, they first have to acquire their \texttt{mutex} :-)

\section{Example: Semaphores}

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

\textbf{Operations:}

\[
\text{Sema * newSema (int n)} \quad \text{--- creates a new semaphore;}
\]

\[
\text{void Up (Sema * s)} \quad \text{--- increases the number of free resources;}
\]

\[
\text{void Down (Sema * s)} \quad \text{--- decreases the number of available resources.}
\]

Therefore, a semaphore consists of:

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

\[
\text{typedef struct }
\]

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

\[
\text{Sema * newSema (int n) }
\]

\[
\{
\text{Sema = s;}
\text{s = (Sema *) malloc (sizeof (Sema));}
\text{s->me = newMutex ();}
\text{s->cv = newCondVar ();}
\text{s->count = n;}
\text{return (s);}
\}
\]
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  loadc 1  loadr 1  store -3
new  store  add  add  return
storer 1  pop  add  store  pop
pop  store  pop  pop

The translation of the body amounts to:

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  newMutex  newCondVar  leaddr -3  leaddr 1
leaddr 3  leaddr 1  leaddr 1  leaddr 1  storer -3
new      store     add       store     return
storer 1  pop       store     store     pop
pop       pop
```