Thus, we translate:

\[
\text{code}_k \text{ join } (e) \rho = \text{code}_k e \rho \\
\text{join} \\
\text{finalise}
\]

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

\[
tid = S[SP]; \\
\text{if} (T[Tab[tid][1]] \geq 0) \{ \\
\text{enqueue} (T[Tab[tid][1], CT); \\
\text{next}
\}
\]
... accordingly:

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

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.

The instruction sequence

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:

<table>
<thead>
<tr>
<th>adr</th>
<th>freeStack(adr)</th>
</tr>
</thead>
</table>

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 * me = 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.

| 1 | BQ |
| 0 | owner |
Then we translate:

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

where:

Then we translate:

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

\[ \text{lock} \]

where:

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

\[ \text{CT} \]

\[ \text{lock} \]

\[ \text{CT} \]

\[ \text{CT} \]

\[ \text{CT} \]

Accordingly, we translate:

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

\[ \text{unlock} \]

where:
Accordingly, we translate:

```plaintext
code unlock (e); \rho = codeunlock \rho
unlock
```

where:

```
CT 5
```

If the queue \( BQ \) is empty, we release the mutex:

```
CT 5
```

```plaintext
If \( S[S[SP]] \neq CT \) Error ("Illegal unlock!");
if \( 0 > tid = dequeque ( S[SP]+1) \) \( S[S[SP-1]] = -1 \);
else {
    \( S[S[SP-1]] = tid \);
    enqueue ( RQ, tid );
}
```

```plaintext
unlock
```

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.

```
0 WQ
```
For condition variables, we introduce the functions:

\[
\begin{align*}
\text{CondVar} & \text{ newCondVar ();} & \quad \text{creates a new condition variable;} \\
\text{void wait (CondVar + cv), Mutex + me);} & \quad \text{enqueues the current thread;} \\
\text{void signal (CondVar + cv);} & \quad \text{re-animates one waiting thread;} \\
\text{void broadcast (CondVar + cv);} & \quad \text{re-animates all waiting threads.}
\end{align*}
\]

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

\[
\begin{align*}
\text{code wait (}e_2, e_1;\text{)} \cdot \rho &= \text{code}_e e_1 \cdot \rho \\
\text{code}_e e_0 \cdot \rho \\
\text{wait} \\
\text{dup} \\
\text{unlock} \\
\text{next} \\
\text{lock}
\end{align*}
\]

where ...

Then we translate:

\[
\text{codeg newCondVar () } \rho = \text{newCondVar}
\]

where:

\[
\begin{align*}
\text{newCondVar} & \quad \text{newCondVar}
\end{align*}
\]
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.

```c
typedef struct {
    Mutex *mutex;
    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;

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    store -2
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    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 \texttt{Down()} \textit{decrements} the counter. If the counter becomes negative, \texttt{wait} is called:

\begin{verbatim}
void Down (Sema * s) {
    Mutex *me;
    me = s\rightarrow me;
    lock (me);
    s\rightarrow count--;
    if (s\rightarrow count < 0) wait (s\rightarrow cv.me);
    unlock (me);
}
\end{verbatim}
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 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);
}
The translation of the body amounts to:

```
alloc 1  loadc 2  add  loadc 1
loadr -2  add  store  add
load  load  loadc 0  load
storerr 1  loadc 1  leq  signal
lock  add  jumpz A  A:  loadr 1
loadr -2  loadr -2  unlock
loadr -2  loadc 2  loadr -2  return
```

The function \( \text{Up}() \) increments the counter again.
If it is afterwards not yet positive, there still must exist waiting threads. One of these is sent a signal:

```
void Up (Sema *s) {
    Mutex *me;
    me = s->me;
    lock (me);
    s->count++;
    if (s->count \leq 0)  signal (s->cv);
    unlock (me);
}
```

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.

   \( \rightarrow \)

Then we implement:
```
void *newStack() { return malloc(N); }
void freeStack(void *adr) { free(adr); }
```
The function \texttt{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++;
    if (s->count <= 0) signal (s->cv);
    unlock (me);
}
```

The translation of the body amounts to:

- \texttt{alloc 1 loadc 2 add loadc 1}
- \texttt{loaddr -2 add store add load}
- \texttt{load storer 1 load loadc 0 load}
- \texttt{lock add jumpz A: loaddr 1}
- \texttt{loaddr -2 unlock}
- \texttt{loaddr -2 loadc 2 loadc -2 return}

The function \texttt{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++;
    if (s->count <= 0) signal (s->cv);
    unlock (me);
}
```

The translation of the body amounts to:

- \texttt{alloc 1 loadc 2 add loadc 1}
- \texttt{loaddr -2 add store add load}
- \texttt{load loaddr 1 leq signal}
- \texttt{loaddr -2 unlock}
- \texttt{loaddr -2 loadc 2 loaddr -2 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++;
    if (s->count <= 0) signal (s->cv);
    unlock (me);
}
```

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
evoid *newStack() { return malloc(M); }
evoid freeStack(void *adr) { free(adr); }
```

Warning:
The de-allocated block may reside inside the stack ✎

We maintain a list of freed stack blocks ✎

```plaintext
   42  19  30  15  7  6  0  3  1
```

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:

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 ...
Warning:
The de-allocated block may reside inside the stack :-(

We maintain a list of freed stack blocks =>

This list supports a function

\[
\text{void insertBlock(int max, int min)}
\]

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:

```
488
```

```
487
```

Allocation and de-allocation of a stack frame makes use of the run-time functions:

```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP + size;
    return result;
}
```

```
void freeFrame(int sp, int size);
```

```
488
```

```
487
```
Warning:
The de-allocated block may reside inside the stack :(

We maintain a list of freed stack blocks :-)

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 ...

In particular, the local variables reside in the new block ...