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 \texttt{join (e)} where we assume that \texttt{e} evaluates to a thread id \texttt{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 :-)

413
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:

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

join

finalize

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

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

\[ S[SP] = JTab[tid][0]; \]
The instruction sequence:

```
term
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 \( J\text{Tab} \) 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:
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:

```
freeStack(adr)
```
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 :-)

419
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:

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

where:

![Diagram showing the transformation of newMutex]
Then we translate:

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

where:

\[ \text{CT} \ 17 \]

\[ \text{CT} \ 17 \]

\[ \text{CT} \ 17 \]
If the mutex is already owned by someone, the current thread is interrupted:

\[
\text{if } (S[S[SP]] < 0) \quad S[S[SP-\_\_>] = CT;
\text{else } \{
\text{enqueue } (S[SP-\_\_>] + 1, CT);
\text{next;}
\}
\]
Accordingly, we translate:

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

\[
\text{unlock}
\]

where:

\[
\begin{array}{c|c|c}
\text{CT} & 5 & \text{CT} \\
\hline
5 & | & 17 \\
\end{array}
\]
If the queue \( BQ \) is empty, we release the mutex:

\[
\text{CT } 5 \rightarrow \text{unlock} \rightarrow \text{CT } 5
\]

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

\[
\text{if } (0 > \text{tid} = \text{dequeue ( S[SP]+1)}) \quad S[SP-] = -1;
\]

\[
\text{else } \{
\]

\[
S[SP--] = \text{tid};
\]

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

\[
\}
\]
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 :-)

\[0\]
For condition variables, we introduce the functions:

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

$$\text{code}_R \ new\text{CondVar} \ () \ \rho \ = \ \text{newCondVar}$$

where:

```
newCondVar
```

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

Therefore, we translate:

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

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

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

if \((0 \leq \text{tid} = \text{dequeue ( S[SP])})\)

\[
\text{enqueue ( RQ, tid );} \\
\text{SP--;}
\]
Analogously:

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

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

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

**Warning:**
The re-animated threads are not blocked !!!
When they become running, though, they first have to acquire their mutex :-)
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 * 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    storer -2
new        store     loadc 1      loadc 2    return
storer 1   pop        add          add        
pop        store      store        pop        pop
```