In contrast to the CMs, we have:

\[
\begin{align*}
\text{SSet} & \quad \text{Set of Stacks} \quad \text{contains the stacks of the threads;} \\
& \quad \text{every cell may contain a base value of an address;} \\
\text{S} & \quad \text{common address space for heap and the stacks;} \\
\text{SP} & \quad \text{Stack-Pointer} \quad \text{points to the current topmost occupied stack cell;} \\
\text{FP} & \quad \text{Frame-Pointer} \quad \text{points to the current stack frame.}
\end{align*}
\]

**Caveat**
- If all references pointed into the heap, we could use separate address spaces for each stack. Besides SP and FP, we would have to record the number of the current stack.
- In the case of C, though, we must assume that all storage regions live within the same address space — only at different locations.
- SP and FP then uniquely identify storage locations.
- For simplicity, we omit the extreme-pointer \( \text{EP} \).

### 48 The Ready-Queue

**Idea**
- Every thread has a unique number \( \text{tid} \).
- A table \( \text{TTab} \) allows to determine for every \( \text{tid} \) the corresponding thread.
- At every point in time, there can be several executable threads, but only one running thread (per processor).
- the \( \text{tid} \) of the currently running thread is kept in the register \( \text{CT} \) (Current Thread).
- The function: \( \text{tid self}() \) returns the \( \text{tid} \) of the current thread. Accordingly:

\[
\text{codey self}() \rho = \text{self}
\]
... where the instruction `self` pushes the content of the register `CT` onto the (current) stack:

\[
S[SP++] = CT;
\]

- The remaining executable threads (more precisely, their `tid's`) are maintained in the queue `RQ` (Ready-Queue).
- For queues, we need the functions:

  - `void enqueue(queue q, tid t)`
  - `tid dequeue(queue q)`

  which insert a `tid` into a queue and return the first one, respectively ...
If a call to dequeue () failed, it returns a value < 0.
The thread table must contain for every thread, all information which is needed for its execution. In particular it consists of the registers PC, SP and FP:

```
2       SP
1       PC
0       FP
```

Interrupting the current thread therefore requires to save these registers:
```
void save () {
    TTab[CT][0] = FP;
    TTab[CT][1] = PC;
    TTab[CT][2] = SP;
}
```

Analogously, we restore these registers by calling the function:
```
void restore () {
    FP = TTab[CT][0];
    PC = TTab[CT][1];
    SP = TTab[CT][2];
}
```

Thus, we can realize an instruction `yield` which causes a thread-switch:
```
tid ct = dequeue ( RQ );
if (ct ≥ 0) {
    save (); dequeue ( RQ, CT );
    CT = ct;
    restore ();
}
```

Only if the ready-queue is non-empty, the current thread is replaced.
Analogously, we restore these registers by calling the function:

```c
void restore () {
    FP = TTab[CT][0];
    PC = TTab[CT][1];
    SP = TTab[CT][2];
}
```

Thus, we can realize an instruction `yield` which causes a thread-switch:

```c
tid ct = dequeue ( RQ );
if (ct ≥ 0) {
    save (); enqueue ( RQ, CT );
    CT = ct;
    restore ();
}
```

Only if the ready-queue is non-empty, the current thread is replaced.

---

49 Switching between Threads

Problem
We want to give each executable thread a fair chance to be completed.

- Every thread must former or later be scheduled for running.
- Every thread must former or later be interrupted.

Possible Strategies
- Thread switch only at explicit calls to a function `yield()`.
- Thread switch after every instruction → too expensive.
- Thread switch after a fixed number of steps → we must install a counter and execute `yield` at dynamically chosen points.

If a call to dequeue () failed, it returns a value < 0.
The thread table must contain for every thread, all information which is needed for its execution. In particular consists of the registers PC, SP and FP:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interrupting the current thread therefore requires to save these registers:

```c
void save () {
    TTab[CT][0] = FP;
    TTab[CT][1] = PC;
    TTab[CT][2] = SP;
}
```

We insert thread switches at selected program points ...
- at the beginning of function bodies;
- before every jump whose target does not exceed the current PC ...

→ rare

The modified scheme for loops \( s \equiv \text{while}(e) s \) then yields:

```c
code s p = A : code e p
         jump z B
         code s p
         yield
         jump A

B : ...
```
Remark

- If-then-else: Statements do not necessarily contain thread switches.
- do-while: Loops require a thread switch at the end of the condition.
- Every loop should contain (at least) one thread switch.
- Loop-Unrolling reduces the number of thread switches.
- At the translation of switch-statements, we created a jump table behind the code for the alternatives. Nonetheless, we can avoid thread switches here.
- At freely programmed uses of jumpi as well as jumpz we should also insert thread switches before the jump (or at the jump target).
- If we want to reduce the number of executed thread switches even further, we could switch threads, e.g., only at every 100th call of yield ...

50 Generating New Threads

We assume that the expression: \( s \equiv \text{create} (e_2, e_1) \) first evaluates the expressions \( e_1 \) to the values \( f, a \) and then creates a new thread which computes \( f(a) \).

If thread creation fails, \( s \) returns the value \(-1\).
Otherwise, \( s \) returns the new thread’s tid.

Tasks of the Generated Code

- Evaluation of the \( e_1 \);
- Allocation of a new run-time stack together with a stack frame for the evaluation of \( f(a) \);
- Generation of a new tid;
- Allocation of a new entry in the TTab;
- Insertion of the new tid into the ready-queue.

The translation of \( s \) then is given by:

\[
\text{code}_{\mathcal{R}} \ s \rho \ = \ \text{code}_{\mathcal{R}} \ e_2 \rho \\
\quad \quad \text{code}_{\mathcal{R}} \ e_1 \rho \\
\quad \quad \text{initStack} \\
\quad \quad \text{initThread}
\]

where we assume the argument value occupies 1 cell.

For the implementation of \( \text{initStack} \) we need a run-time function \( \text{newStack()} \) which returns a pointer onto the first element of a new stack.

If the creation of a new stack fails, the value 0 is returned.
newStack();
if (S[SP]) {
  S[SP] = S[SP-1];
  S[SP+1] = 1;
  S[SP+2] = f;
  S[SP-1] = S[SP] + 2; SP--
}
else S[SP = SP - 2] = -1;

newStack();
if (S[SP]) {
  S[SP] = S[SP-1];
  S[SP+1] = -1;
  S[SP+2] = f;
  S[SP-1] = S[SP] + 2; SP--
}
else S[SP = SP - 2] = -1;
Remark

- The continuation address \( f \) points to the (fixed) code for the termination of threads.
- Inside the stack frame, we no longer allocate space for the \( EP \) the return value has relative address -2.
- The bottom stack frame can be identified through \( FPold = -1 \).

In order to create new thread ids, we introduce a new register \( TC \) (Thread Count).
Initially, \( TC \) has the value 0 (corresponds to the \( tid \) of the initial thread).
Before thread creation, \( TC \) is incremented by 1.

```c
if (S[SP] ≥ 0) {
    tid = ++TC;
    TTns[tid][0] = S[SP];
    TTns[tid][1] = S[SP-1];
    TTns[tid][2] = S[SP];
    S[SP] = tid;
    enqueue( RQ, tid );
}
```

51 Terminating Threads

Termination of a thread (usually) returns a value. There are two (regular) ways to terminate a thread:
1. The initial function call has terminated. Then the return value is the return value of the call.
2. The thread executes the statement \( \text{exit} (e) \); Then the return value equals the value of \( e \).

Caveat

- We want to return the return value in the bottom stack cell.
- \( \text{exit} \) may occur arbitrarily deeply nested inside a recursion. Then we de-allocate all stack frames ...
- ... and jump to the terminal treatment of threads at address \( f \).
Therefore, we translate:

\[
\text{code exit (c); } \rho = \text{code}\_c \rho
\]

\[
\text{exit}
\]

\[
\text{term}
\]

\[
\text{next}
\]

The instruction `term` is explained later ...

The instruction `exit` successively pops all stack frames:

```c
result = S[SP];
while (FP ≠ -1) {
    SP = FP−2;
    FP = S[FP−1];
}
S[SP] = result;
```

The instruction `next` activates the next executable thread:

in contrast to `yield` the current thread is not inserted into `RQ`.

The instruction `next` activates the next executable thread:

in contrast to `yield` the current thread is not inserted into `RQ`.
The instruction next activates the next executable thread: in contrast to yield the current thread is not inserted into RQ.

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 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:

\[
\text{code} \_\text{R} \text{ join} (e) \rho = \text{code} \_\text{R} e \rho \\
\text{join} \\
\text{finalize}
\]

... where the instruction join is defined by:

\[
tid = S[SP]; \\
\text{if} (\text{TTab}[tid][1] \geq 0) \\
\quad \text{enqueue} \{ \text{JTab}[tid][1], \text{CT} \} \\
\text{next}
\]
Thus, we translate:

\[
\begin{align*}
\text{code}_{\text{join}}(e; \rho) & = \text{code}_{\text{join}}(e; \rho) \\
\text{join} & \\
\text{finalize} & \\
\end{align*}
\]

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

\[
\begin{align*}
tid & = S[SP]; \\
\text{if } (\text{JTab[tid][1]} \geq 0) \{ \\
& \quad \text{enqueue } (\text{JTab[tid][1]}, \text{CT}); \\
& \quad \text{next} \\
\}
\end{align*}
\]

\[
S[SP] = \text{JTab}[S[SP]][0];
\]

The instruction sequence:

\[
\begin{align*}
\text{term} & \\
\text{next} & \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{code exit}(e; \rho) & = \text{code}_{\text{exit}}(e; \rho) \\
\text{exit} & \\
\text{term} & \\
\text{next} & \\
\end{align*}
\]

Therefore, we translate:

\[
\begin{align*}
\text{result} & = S[SP]; \\
\text{while } (\text{FP} \neq -1) \{ \\
& \quad \text{SP} = \text{FP} - 2; \\
& \quad \text{FP} = S[\text{FP} - 1]; \\
\}
\end{align*}
\]

\[
S[SP] = \text{result};
\]
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:

```cpp
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`:

![Free Stack Diagram]