Script generated by TTT

Title: Seidl: Virtual_Machines (30.06.2015)

Date: Tue Jun 30 10:16:31 CEST 2015

Duration: 89:57 min

Pages: 52

46 The Language ThreadedC

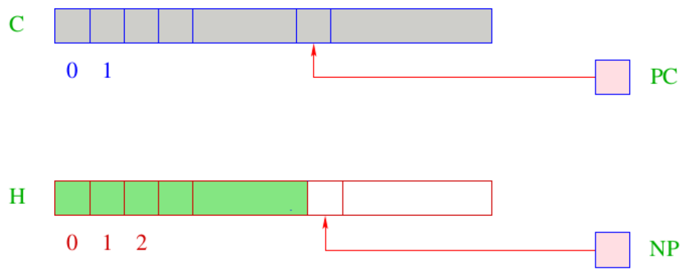
We extend C by a simple thread concept. In particular, we provide functions for:

- generating new threads: create();
- terminating a thread: exit();
- waiting for termination of a thread: join();
- mutual exclusion: lock(), unlock(); ...

In order to enable a parallel program execution, we extend the virtual machine (what else? :-)

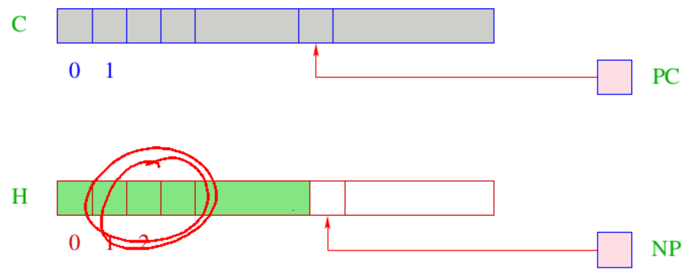
47 Storage Organization

All threads share the same common code store and heap:



47 Storage Organization

All threads share the same common code store and heap:

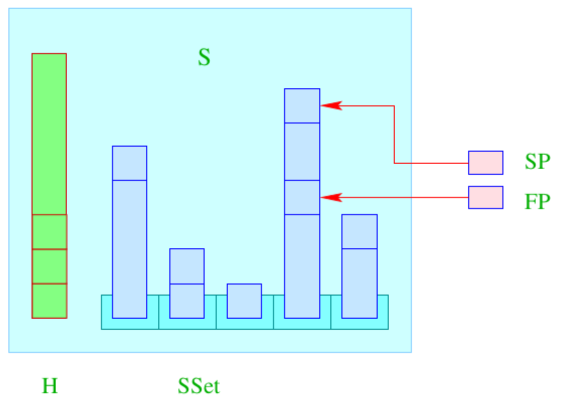


... similar to the CMa, we have:

C	=	Code Store – contains the CMa program; every cell contains one instruction;
PC	=	Program-Counter – points to the next executable instruction;
H	=	Heap – every cell may contain a base value or an address; the globals are stored at the bottom;
NP	=	New-Pointer – points to the first free cell.

For a simplification, we assume that the heap is stored in a separate segment. The function malloc() then fails whenever NP exceeds the topmost border.

Every thread on the other hand needs its own stack:



In constrast to the CMa, we have:

SSet	=	Set of Stacks – contains the stacks of the threads; every cell may contain a base value of an address;
S	=	common address space for heap and the stacks;
SP	=	Stack-Pointer – points to the current topmost ocupied stack cell;
FP	=	Frame-Pointer – points to the current stack frame.

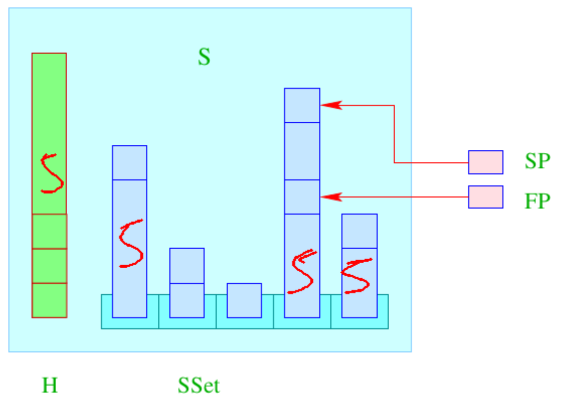
Warning:

- 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 Und FP then uniquely identify storage locations.
- For simplicity, we omit the extreme-pointer EP.

Every thread on the other hand needs its own stack:



In constrast to the CMa, we have:

SSet	=	Set of Stacks – contains the stacks of the threads;
		every cell may contain a base value of an address;
S	=	common address space for heap and the stacks;
SP	=	Stack-Pointer – points to the current topmost ocupied stack cell;
FP	=	Frame-Pointer – points to the current stack frame.

Warning:

- If all references pointed into the heap, we could use separate address spaces for each stack.

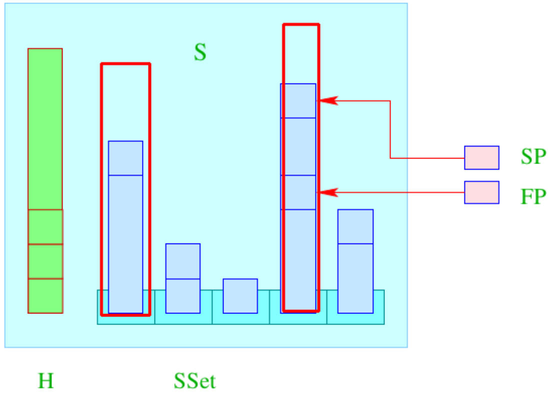
  Besides SP and FP, we would have to record the number of the current stack :-)

- In the case of C, though, we must assume that all storage regions live within the same address space — only at different locations :-)

  SP Und FP then uniquely identify storage locations.

- For simplicity, we omit the extreme-pointer EP.

Every thread on the other hand needs its own stack:



In constrast to the CMa, we have:

SSet	=	Set of Stacks – contains the stacks of the threads;
		every cell may contain a base value of an address;
S	=	common address space for heap and the stacks;
SP	=	Stack-Pointer – points to the current topmost ocupied stack cell;
FP	=	Frame-Pointer – points to the current stack frame.

Warning:

- 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 Und FP then uniquely identify storage locations.

- For simplicity, we omit the extreme-pointer EP.

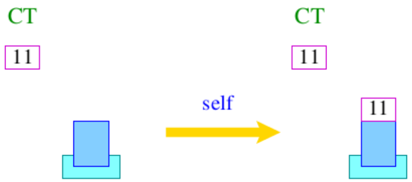
48 The Ready-Queue

Idea:

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

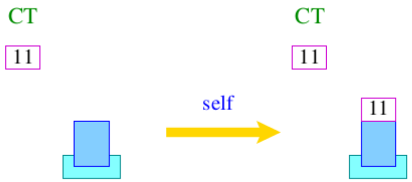
$$\text{code}_R\ \textbf{self}\ ()\ \rho \quad = \quad \text{self}$$

... where the instruction self pushes the content of the register CT onto the (current) stack:



S[SP++] = CT;

... where the instruction self pushes the content of the register CT onto the (current) stack:

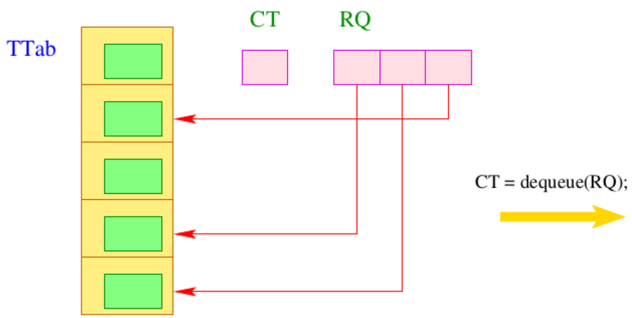


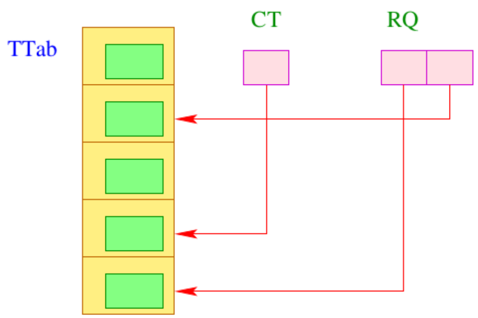
S[SP++] = CT;

- 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)
```

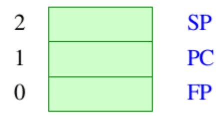
which insert a tid into a queue and return the first one, respectively ...





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 und 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 (); enqueue ( 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:

```
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 (); 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.

$\Longrightarrow$

- 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 $\Longrightarrow$ too expensive :-(
- Thread switch after a fixed number of steps $\Longrightarrow$ we must install a counter and execute yield at dynamically chosen points :-(

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 ...

$\Longrightarrow$ rare :-))

The modified scheme for loops $s \equiv$ **while** $(e)\ s$ then yields:

```
code s ρ  =  A :  codeR e ρ
                  jumpz B
                  code s ρ
                  yield
                  jump A
             B :  ...
```


Note:

- **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-Unroling 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: $s \equiv$ **create** (e_0, e_1) first evaluates the expressions e_i 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_i;
- 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.

50 Generating New Threads

We assume that the expression: $s \equiv$ **create** (e_0, e_1) first evaluates the expressions e_i 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:

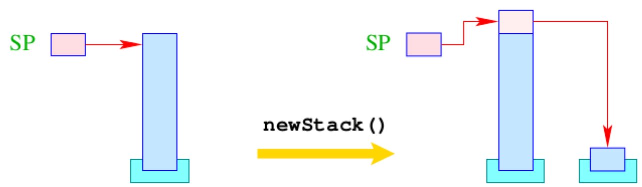
- Evaluation of the e_i;
- Allocation of a new run-time stack together with a stack frame for the evaluation of $f\ (a)$;
- Generation of a new tid;
- Allocation of a new entry in the TTab;
- Insertion of the new tid into the ready-queue.

The translation of s then is given by:

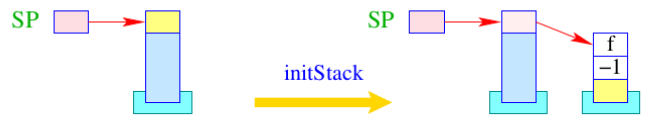
$$\begin{aligned} \text{code}_\text{R}\ s\ \rho \quad = \quad & \text{code}_\text{R}\ e_0\ \rho \\ & \text{code}_\text{R}\ e_1\ \rho \\ & \text{initStack} \\ & \text{initThread} \end{aligned}$$

where we assume the argument value occupies 1 cell :-)

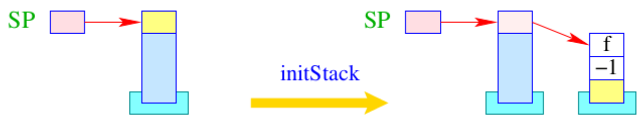
For the implementation of initStack we need a run-time function newStack() which returns a pointer onto the first element of a new stack:



If the creation of a new stack fails, the value 0 is returned.




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




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

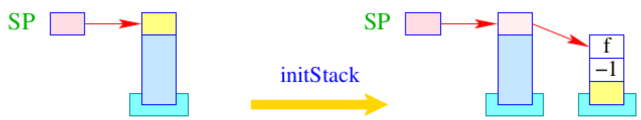

Remark

- 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 $\Longrightarrow$ 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.



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

418

Remark

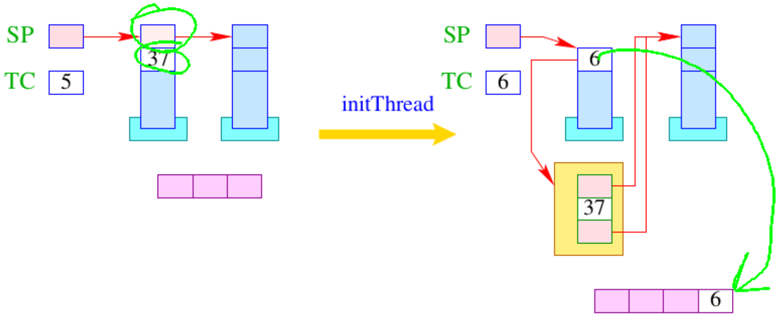
- The continuation address f points to the (fixed) code for the termination of threads.
- Inside the stack frame, we no longer allocate space for the EP $\Longrightarrow$ the return value has relative address -2.
- The bottom stack frame can be identified through FPold = -1 :-)

In order to create new thread ids, we introduce a new register TC (Thread Count).

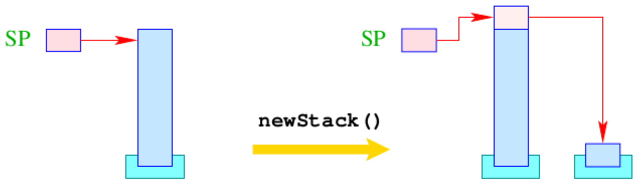
Initially, TC has the value 0 (corresponds to the tid of the initial thread).

Before thread creation, TC is incremented by 1.

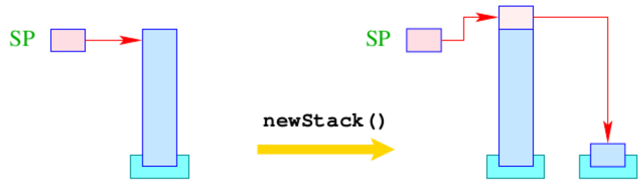
419




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

If the creation of a new stack fails, the value 0 is returned.



If the creation of a new stack fails, the value 0 is returned.

417

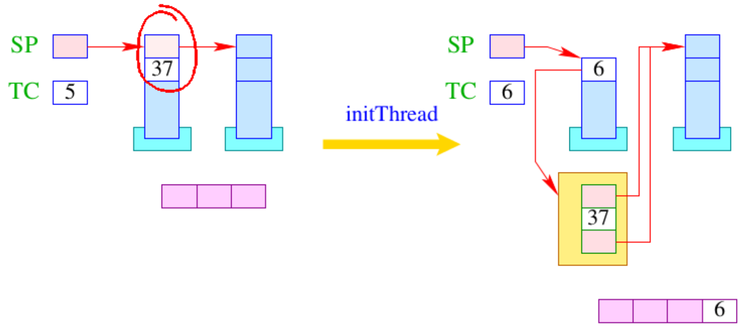
Remark

- 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 $\Longrightarrow$ 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.




```
if (S[SP] ≥ 0) {
    tid = ++TC;
    TTab[tid][0] = S[SP]-1;
    TTab[tid][1] = S[SP];
    TTab[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 **exit** (e); Then the return value equals the value of e.

Warning:

- We want to return the return value in the bottom stack cell.
- **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:

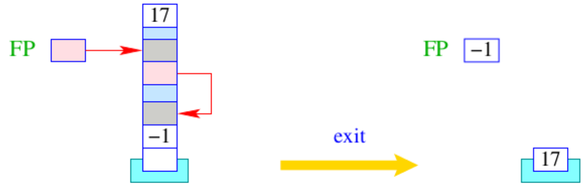
$$\text{code } \mathbf{exit}\ (e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$

exit
term
next

The instruction term is explained later :-)

The instruction exit successively pops all stack frames:

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

If the queue RQ is empty, we additionally terminate the whole program:

```
if (0 > ct = dequeue( RQ ))   halt;
else {
        save ();
        CT = ct;
        restore ();
     }
```


If the queue RQ is empty, we additionally terminate the whole program:

```
if (0 > ct = dequeue( RQ ))   halt;
else {
        save ();
        CT = ct;
        restore ();
     }
```

426

52 Waiting for Termination

Occationally, 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 treads 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:

$$\begin{array}{lll} \text{code}_R\ \textbf{join}\ (e)\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{join} \\ & & \text{finalize} \end{array}$$

... where the instruction join is defined by:

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


52 Waiting for Termination

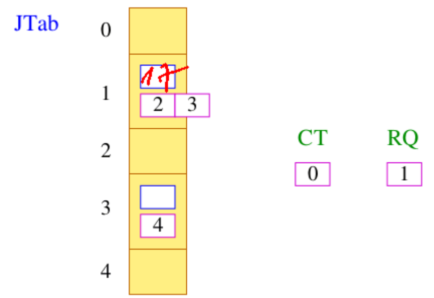
Occationally, a thread may only continue with its execution, if some other thread has terminated. For that, we have the expression **join** (e) where we assume that e evaluates to a thread id tid.

- 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 treads 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 :-)

427

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.

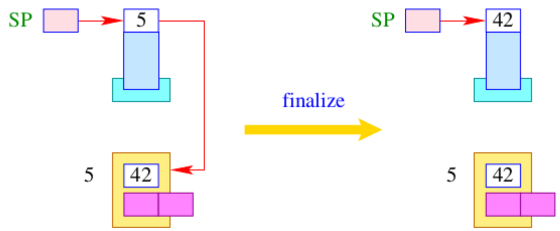
Thus, we translate:

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

... where the instruction join is defined by:

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


... accordingly:



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

Thus, we translate:

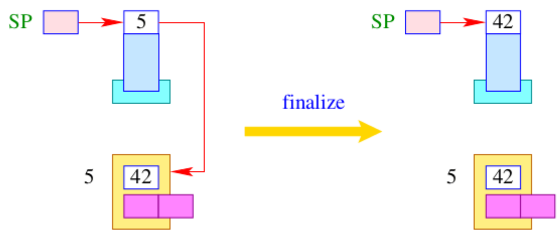
code_R join (e) ρ = code_R e ρ
join
finalize

... where the instruction join is defined by:

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

429

... accordingly:



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

S[SP]

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: