

Title: Petter: Virtual Machines (25.06.2019)

Date: Tue Jun 25 10:17:07 CEST 2019

Duration: 71:30 min

Pages: 20

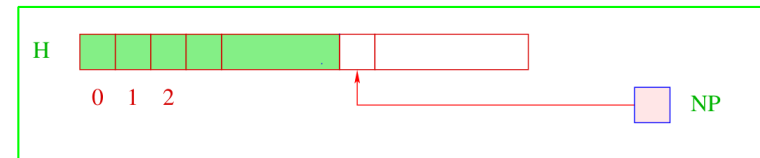
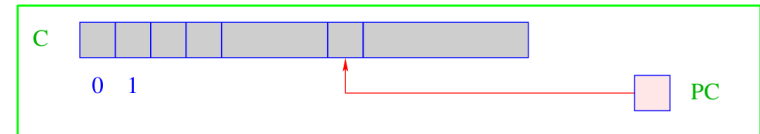
... similar to the **CMa**, we have:

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.

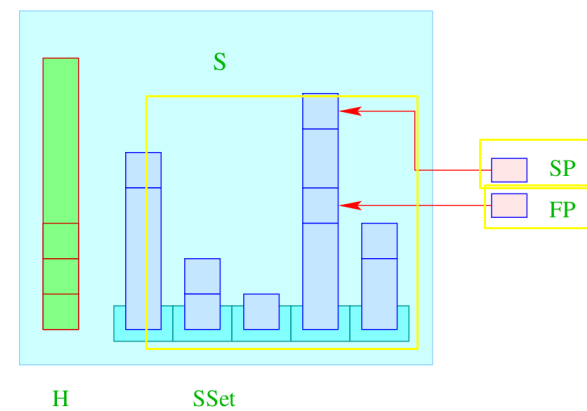
For a simplification, we assume that the heap is stored in a separate segment. The function `malloc()` then fails whenever **NP** exceeds the topmost border.

47 Storage Organization

All threads share the same common code store and heap:



Every thread on the other hand needs its **own stack**:



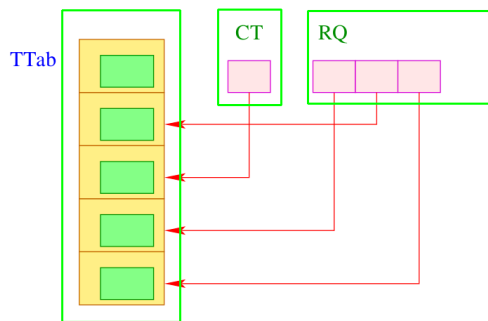
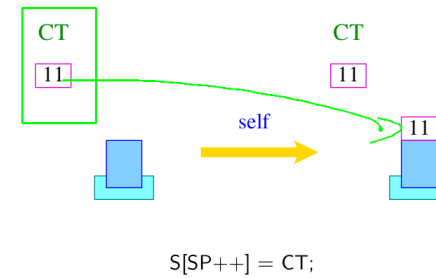
In contrast to the **CMa**, we have:

- S**Set = 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 occupied stack cell;
- FP** = Frame-Pointer – points to the **current** stack frame.

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 **EP**.

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



- 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 ...

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.

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

Possible Strategies

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

413

Remark

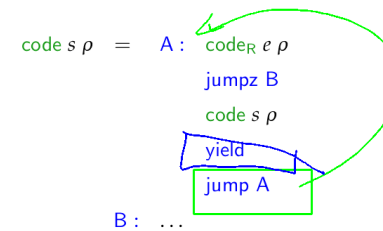
- **If-then-else**-Statements do not necessarily contain thread switches.
- **do-while**-Loops require a thread switch at the end of the condition.
- Every loop should contain (at least) one thread switch.
- **Loop-Unrolling** reduces the number of thread switches.
- At the translation of **switch**-statements, 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 ...`**

415

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 ...
- \implies rare!

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



414

50 Generating New Threads

We assume that the expression: $s \equiv \mathbf{create} (e_0, e_1)$ first evaluates the expressions e_i to the values $f_i(a)$ 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.

416

Remark

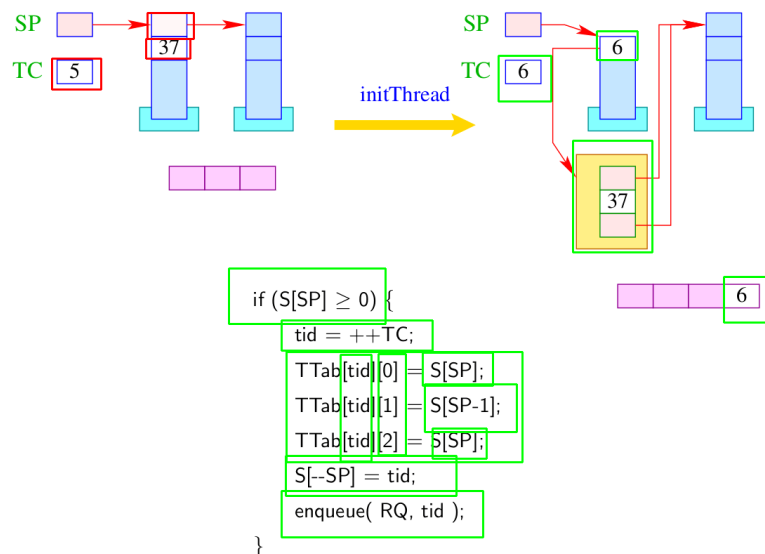
- The continuation address `f` points to the (fixed) code for the termination of threads.
- Inside the stack frame, we no longer allocate space for the `EP` \implies the return value has relative address `-2`.
- The bottom stack frame can be identified through `FPold = -1`.

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.

420



421

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` \implies 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.

420

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`.

Caveat

- 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`.

422

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

```

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

```

426

Thus, we translate:

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

join
finalize

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

```

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

```

429

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.

- Threads waiting for termination are maintained in the table `JTab`.
- There, we also store the return values of threads.

427

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 `FP` to `-1`;
- ... all threads must be notified which have waited for the termination.

For the instruction `term` this means:

431

```

FP = -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` :

