Concurrent processing and procedure invocations

In this note we design a mechanism for concurrent execution of a collection of programs by a single-processor Von Neumann computer. As a special case of this we shall <u>derive</u> the mechanisms for procedure call and return.

The operation of the processor can, as a first approximation, be described by the following program —explanation follows— :

```
(0)     do true → IR, PI := S·PI , PI+1
                ; "execute the instruction in IR"
        od ,
```

where S is an array of "words" representing the store of the machine, IR is an internal register of the processor, and where variable PI — "program index" identifies the "instruction to be executed next". PI is part of the state of the computation; for (justified) reasons of efficiency PI is usually implemented as a register in the processor.

The computation that emerges when a program is executed is called a <u>process</u>; so, whereas a program is a piece of code a process is a computation. That the distinction is relevant follows from the observation that <u>different</u> processes can execute the <u>same</u> program —code sharing—. Consequently, every process needs its "own" program index, even when the process shares its code with other processes.

We now consider $n+1$ processes executing their code by the same processor. We assume the processes to be numbered $j: 0 \le j \le n$ and we introduce an array $pi(j: 0 \le j \le n)$ with the interpretation that:

$$pi.j = \text{"the program index of process } j\text{"} .$$

The processor is just an ordinary sequential processor: it can only execute instructions for one process at a time. Therefore, we introduce a system variable cp, with the invariant

$$0 \le cp \le n \quad ,$$

identifying the process for which the processor is executing instructions. The operation of the processor can now be described by the following program:

```
(1)   do true →  IR, pi·cp := S·(pi·cp) , pi·cp+1
               ; «execute the instruction in IR»
      od .
```

(Because cp occurs in this program it is reasonable to expect that cp be implemented as a register.)

In order to allow for <u>process switching</u> we require that the processor be equipped with a new instruction:

$$\text{switch}(p) \qquad \text{with effect:} \qquad cp := p \quad .$$

By means of switch and a (so-called) scheduling strategy (for selecting values p) concurrent processing can be effectively implemented, but how to do this is not the topic of this note.

The above program (1) differs from the earlier program (0) in that PI has been replaced by pi·cp . For large n it is not reasonable to require that all elements of pi be kept in processor registers; but, keeping pi·cp in the store S gives rise to at least 2 additional store accesses per instruction executed. Fortunately, a very acceptable compromise is possible, under the mild assumption that switch is invoked not too often: in that case cp is mainly constant and it suffices to keep only pi·cp in a register. Formally, with PI that register and with $q(j : 0 \leq j \leq n)$ an array in the store, pi can be represented by PI and q as follows:

$$Q: \quad pi{\cdot}cp = PI \ \wedge \ (\forall j : j \neq cp : pi{\cdot}j = q{\cdot}j) \quad .$$

Then, in program (1) we may substitute PI for pi·cp , as a result of which we obtain program (0) back again: so, the good old Von Neumann processor can still be used. All we need is that the machine has an instruction switch in its repertoire; in terms of the new representation, using the required invariance of Q , we can derive that switch now must be defined as follows:

switch(p) has effect:

```
   { Q }
   q·cp := PI
 ; { (∀j :: pi·j = q·j) }
   cp := p
 ; { (∀j :: pi·j = q·j) }
   PI := q·cp
   { Q }  .
```

The operation q·cp := PI is traditionally called "saving the program index" (of the suspended process) and PI := q·cp is traditionally called "restoring the program index" (of the resumed process). The above shows that the necessity of saving and restoring operations follows not from the process switching itself but from the sharing of a register among the processes.

The above transformation, via invariant Q, from program (1) back to program (0) can be applied to any shared use of a register: first, introduce for each process a private instance of the register, like pi·j for PI, and, second, introduce the register and the store representation by means of an invariant like Q; then the saving and restoring obligations for that register follow immediately from that invariant and the assignment cp := p . The only difference between PI and other registers is that the "instruction to be executed next" depends upon PI; therefore, the above process switch must, with saving and restoring PI included, be considered as a <u>single</u> instruction, whereas saving and restoring other registers may be encoded in different instructions. (That is to say: in the absence of interrupts, of course.)

* * *

We now consider a rather special case of the above. The case is special in the sense that we add:

S: $cp = n$

as an invariant to the system, and we study the effects of $n := n+1$ —extension of the collection of processes with a new process— and of $n := n-1$ —removal of process n from the collection—. Invariant S can then be interpreted as "the process with the highest number has the highest priority". We program this in terms of the variable pi first, thus postponing the implementation in terms of PI and q.

• <u>$n := n+1$</u> : with b for the address of the (first instruction of the) code corresponding to the new process, the addition of that new process to the collection amounts to —doing justice to S as well—:

$$n, cp := n+1, cp+1 \; ; \; pi.n := b \; .$$

• <u>$n := n-1$</u> : this is just the inverse of the above:

$$n, cp := n-1, cp-1 \quad .$$

Here, I have deliberately avoided the use of switch: the above only represents the required state change of the machine, and we are not (yet) interested in instructions encoding these operations.

Now we take into account the representation of pi, via Q, by means of PI and q; moreover, we eliminate the now superfluous variable n whose role is taken over by cp. Thus we obtain a version with saving and restoring operations (similar to the operation switch on p.3), as follows.

- <u>n := n+1</u> : this becomes

$$q \cdot cp := PI \ ; \ cp := cp+1 \ ; \ PI := b \quad .$$

- <u>n := n−1</u> : this becomes

$$cp := cp-1 \ ; \ PI := q \cdot cp \quad .$$

All modern machines contain dedicated instructions for these two operations; the former is usually called "call b" and the latter is usually called "return". For this special, nested arrangement of processes these two instructions are sufficient and switch has become superfluous. For this special case, "invocations" is the common term for "processes" and the pieces of code belonging to the invocations are usually called "procedures".

Thus, we have derived the call-and-return mechanism for procedure invocations by viewing the latter as a special case of concurrent processes. Notice that in this case code sharing amounts to <u>recursion</u>. Moreover, array q and variable cp together constitute what is known as the "stack of return addresses". In the above view, however, q does not contain return addresses but the program indices of suspended computations. The manipulations of q, cp, and PI in the instructions call and return are the special-case instances of process switching.

Eindhoven, 22 september 1993
Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven