On a Renewed Visit to the Banker and a Remarkable Analogy

Coen Bron

In the mid-1960's I was in the fortunate circumstances to exchange my ill-chosen future in chemistry for an apprenticeship in Computing Science by joining Edsger Dijkstra's group which was — at that time — deeply involved with the design and implementation of the T.H.E. Operating System. I deliberately put in the dots where Edsger may have just as deliberately left them out in his original publication[4], to suggest that the E. stands for Eindhoven and there was no attempt to suggest that this system was to be the final word in this area (a position that now seems to be claimed by Unix, whether we like it or not).

The main impact on my personal career was the recognition that — in spite of the T.H.E. System’s beautiful structure — the tools that we had used in its implementation were far from adequate. Operating systems are (also) just programs, and so any higher level approach to writing programs may be applied to the construction of systems programs as well. Knowledge of assembly language should be necessary only for those responsible for the code generating part of a compiler, and high level languages should provide the loop-holes to escape from the (sometimes too) strictly enforced rules to enable the writing of those parts of an Operating System that otherwise would necessitate an overall escape to assembly language.

* * *

Before I digress any further from my intended visit to the Banker, let me return to those memorable years of the T.H.E. System. These were the years of the invention of concepts and paradigms that have already immortalized Edsger Dijkstra: Co-operating Sequential Processes, Semaphores with $P$- and $V$-operations[2] (they are the only concept for which a one letter mnemonic to me seems acceptable[5]), the Banker’s Algorithm[6], the Dining (quintuple of) Philosophers[3].

The Banker’s Algorithm models the decision by an Operating System
when a process requests non-preemptable resources, whether allocating the request can be done without the risk of an ensuing deadlock. If more than one resource type is involved, the problem is modelled by a banker who can lend his customers units of several non-exchangeable currencies. The Banker must decide if, when granting a loan, he can be sure to get all of his money back eventually. The customers are well behaved in the sense that they are accepted as such when they can give an upper limit for the number of units of different currencies they will want to borrow at any moment, and they will—given their maximal loan—return everything in due time.

For several years, teaching a course in Operating Systems at the University of Twente, I have presented the algorithm in the following form:

```plaintext
TYPE process = 1..??;
   units = ...;  {Say: some vector, each element denoting the number of units of one particular currency}

VAR claim, loan: ARRAY [process] OF units;
   {claim[i] "+" loan[i] = max[i]}

FUNCTION safe({if} p: process
   ;{given} sop: SET OF process
      {having loan > 0}
   ;{wanting} request: units
   ;{while available} cash: units
);
{Determines if, given that the initial state of the Banker is safe, a safe state would result if the request were granted}
}
VAR otherp: process;

BEGIN {assert: request "<=" claim[p]}
   DO ( NOT claim[p] "<=" cash) AND
      exists( otherp, {in} sop
         , {satisfying}
            claim[otherp] "<=" cash - request
      )
   -> cash := loan[otherp]; sop := [otherp]
   OD
;   safe := claim[p] "<=" cash
END;
```