Correspondence

Termination Indicators in Programming

Dear Sir,

M. C. Er's bubble sort algorithm[1] can be further improved by using the termination indicator to hold the index of the last swap on any pass. Elements above this in the array are already in order and need not be scanned again.

```
t:= N-1;
  while t > 0 do begin
  i:= t; j:= 0;
  t:= 0;
  while j < > i do begin
    if a[j] > a[j+1] then begin
      swap (a[j], a[j+1]);
      t:= j
      end;
    j:= j+1
  end
end
```

In this form the algorithm is directly comparable with Er's. However, t is now more than a termination indicator; it holds the number of comparisons to be made on the current iteration. Therefore the algorithm is more elegantly written as follows, where s, like j, is local to the current scan and simply holds the index of the latest swap.

```
t:= N-1;
  while t > 0 do begin
  j:= 0;
  s:= 0;
  while j < t do begin
    if a[j] > a[j+1] then begin
      swap (a[j], a[j+1]);
      s:= j
      end;
    j:= j+1
    end;
  t:= s
end
```

A similar version is given by Gonnet.[2]

In view of the simplicity of this algorithm it is rather remarkable that other books on advanced programming and data structures are still using boolean flags in their bubble sorts.[3,4,5]

Of course it could be argued that the bubble sort is so inefficient that it should not be included in such a text. My experience is that a naïve student asked to write a sort without any leads from the tutor will invariably produce a bubble. It is then helpful to discuss ways in which this could be improved before pointing out that better sort algorithms exist.

Yours faithfully

ANN V. BOWKER (Mrs)
Department of Computing,
Trent Polytechnic,
Nottingham

References

1. M. C. Er, The Use of Termination Indicators in Computer Programming. *The Computer Journal* **29** (5), 430 (1986).
2. G. H. Gonnet, *Handbook of Algorithms and Data Structures*. Addison-Wesley, London (1984).
3. Y. Langsam *et al.*, *Data Structures for Personal Computers*. Englewood Cliffs, N.J., Prentice-Hall (1985).
4. T. L. Naps and B. Singh, *Introduction to Data Structures with Pascal*. West (1986).
5. J. F. Korsh, *Data Structures, Algorithms, and Program Style*, PWS (1986).

An implementation of parallel processing

Dear Sir,

Considering the current interest in languages and implementations for parallel processing it might be of interest to readers to draw attention to some little-known work carried out by Dr A. G. Hill and myself in the 1970s.[1-3]

Our parallel implementation was fairly coarse-grained and was based, like CSP/80, on a number of separately compiled processes. These however were linked dynamically by the operating system. Co-operating processes were structured as a tree, with each process having at most one 'owner' but each able to have zero, one or more 'slaves'.

Synchronisation was dependent on an escapement mechanism implemented by the OS.

(i) The 'owner' process issues PUT (⟨slave⟩).
(ii) The 'slave' process issues GETM, and is held up if necessary until the corresponding PUT has been issued.
(iii) The 'slave' process issues PUTM when it has completed some processing
(iv) The owner process issues GET (⟨slave⟩), and is held up if necessary until the corresponding PUTM has been issued.

If processes are written in a structured language (we used a modified CORAL) the synchronisation requests can be validated very simply at a syntactic level by the compiler, and it is easy to show that deadlock conditions cannot occur. PUT and GET calls to different slaves may be interleaved in any desired manner so long as they alternate properly in execution.

For communicating data between processes the very simple convention was adopted that the data structures of a 'slave' (procedures can of course be considered as data structures) are accessible to the 'owner', but only before GETM-PUT(⟨s⟩) or after PUTM-GET(⟨s⟩). These constraints also can easily be checked at compile time.

In general therefore a running process will

(i) Obtain a 'slave' from the Operating System, for example REQUEST ('SORT').
(ii) Copy parameters – perhaps an array or pointer – to the 'slave' workspace.
(iii) Issue PUT('SORT'); and perhaps continue computation in parallel with 'slave' operation
(iv) Issue GET('SORT'); and perhaps be held up for a while until the sorting operation is complete.
(v) Copy the results if necessary.
(vi) Signal to the Operating System DISCARD('SORT').

A variety of other schemes are possible, including the degenerate cases where

(*a*) 'slave' is set up and DETACHed to run autonomously. In a multi-user system a user may be given a shell, or in a real-time system an autonomous controller with appropriate priority may be set up; or

(*b*) The 'slave' may contain no code and never 'run' but simply provide data structures for use by the owner. This mechanism provides a convenient way of allowing for dynamic store allocation like the PASCAL 'heap'.

Finally non-determinism was accommodated by an explicit PEEK(⟨slave⟩) or PEEKM which determined if a process could issue a GET (or GETM) without being held up, thus:

```
if PEEK(proc1)
  then
    begin
      GET(proc1);
      . . .
    end
elseif PEEK(proc1)
  then
    begin
      GET(proc1);
      . . .
    end
else . .
```

The system may have been rather naïve, but it does have a certain simplicity and might perhaps still be useful in practice.

Yours sincerely

H. R. A. TOWNSEND
The National Hospitals for
Nervous Diseases
Queen Square
London WC1

References

1. A. G. Hill and H. R. A. Townsend, *Deadlock-free parallel processing*. Advance paper, 4th International Joint Conference on Artificial Intelligence, Tbilisi, pp. 534–537 (1975).
2. H. R. A. Townsend, *In the Footsteps of the Amoeba, or multiprocessing without tears*. Machine Intelligence Research Unit Report, University of Edinburgh (MIP-R-97) (1972).
3. H. R. A. Townsend, A real-time programming system. *International Journal of Bio-Medical Computing*, **10**, 129–143 (1979).