CAREFULLY SCHEDULED SELECTION WITH ALT

Geraint Jones, Programming Research Group, 11 Keble Road, Oxford OX1 3QD

The Summer 1988 edition of the oug newsletter contains an article[2] by Alan Chalmers which devotes about forty lines of occam and six of transputer assembler code to an obscure (and, as it happens, unsuccessful) attempt to make a 'fair' selection with an ALT. This is a well-studied problem to which there are a number of very much tidier known solutions; the earliest reference to coding a fairish ALT in occam upon which I could readily lay a hand was [5].

What makes an ALT fair?

Briefly, the problem is that the process

```
WHILE busy
  ALT i = 0 FOR n
    c[i] ? x[i]
      P(i)
```

makes potentially 'unfair' selections. Suppose, for simplicity, that the process $P(i)$ uses only the corresponding $c[i]$, and no other $c[j]$. If one of the channels, say $c[0]$, is driven by a process which makes it ready again before $P(0)$ has completed execution, then it is permitted by the semantics that the input from $c[0]$ will be selected every time. (In fact, this happens to be guaranteed by the transputer implementation.)

For those who have not come across the problem before: a frequently asked question is 'why is the occam ALT not a fair one'? Roughly speaking, the problem is that fairness cannot be a static property of the ALT construct. An ALT makes just one selection, and the best that you could hope for in the way of fairness is that the selection is reasonably even-handed. A long sequence of independent executions of an even-handed randomised ALT might yet be unfair, although with a very low probability. Fairness is a dynamic property of a sequence of selections made by the same construct. That, incidentally, is why there have to be state variables *outside* the loop in each of the programs which are suggested in this article.

Chalmers codes up something like

```
INT favourite :
SEQ
  favourite := 0
  WHILE busy
    INT how.many :
    [n]INT available :
    [n] BOOL selected :
    SEQ
      how.many := 0
      SEQ i = 0 FOR n
        IF
          ... c[i] is ready -- this is not coded in occam
```

```
          SEQ
            available[how.many] := i
            how.many := how.many + 1
        ... c[i] is not ready
          SKIP
  --
  --  how.many is the number of channels that are ready
  --  [available FROM 0 FOR how.many] are the indexes,
  --          in increasing order, of those that are ready
  --
  IF
    how.many = 0            -- none ready, choose any one
      SEQ i = 0 FOR n
        selected[i] := TRUE
    how.many > 0
      SEQ
        IF
          (favourite + 1) < how.many
            favourite := favourite + 1
          (favourite + 1) >= how.many
            favourite := 0
        SEQ i = 0 FOR n
          selected[i] := (i = favourite)
  ALT
    selected[i] & c[i] ? x[i]
      P(i)
```

(I have left out the assembly code to make it clearer what happens.)

It is difficult to tell what he intended, but it is plain that something has gone wrong here. Suppose that n is at least three, and that the following sequence of events happens:

`favourite := 0`
$c[0]$, $c[1]$, $c[2]$ become ready
`how.many, [available FROM 0 FOR 3] := 3, [0, 1, 2]`
`favourite := 1`
$c[1]$ is selected, and is no longer ready
`how.many, [available FROM 0 FOR 2] := 2, [0, 2]`
`favourite := 0`
$c[0]$ is selected, and is no longer ready
$c[0]$, $c[1]$ happen to become ready again
`how.many, [available FROM 0 FOR 3] := 3, [0, 1, 2]`
`favourite := 1`
...

Guess when $c[2]$ is chosen. I would not like to swear that I have got Chalmers' code right: it certainly was not immediately obvious what it did, which is partly the point of this present article.

Coding a fairer ALT

So, how should one code up a fairer selection? A feasible strategy is to rotate the priority of the branches through a fixed sequence:

```
INT favourite :
SEQ
  favourite := 0
  WHILE busy
    SEQ
      PRI ALT index = favourite FOR n
        VAL shifted IS index \ n :
        c[shifted] ? x[shifted]
          P(shifted)
      favourite := (favourite + 1) \ n
```

which guarantees that any of the channels will be serviced after at most n passes through the loop. Even if some of the channels are saturated, this scheme will guarantee about $1/n$th of the bandwidth to any channel that needs it. (That is the solution suggested by [5].)

Of course, if you do not like all those modulo operators inside a loop, you could always do it by indirection through an array that keeps track of a permutation of the branches:

```
[n]INT perm:
SEQ
  SEQ i = 0 FOR n
    perm[i] := i
  WHILE busy
    SEQ
      PRI ALT index = 0 FOR n
        VAL shifted IS perm[index] :
        c[shifted] ? x[shifted]
          P(shifted)
      INT first:
      SEQ
        first := perm[0]
        SEQ i = 0 FOR n - 1
          perm[i] := perm[i + 1]
        perm[n - 1] := first
```

which has the same effect. Of course, this is rather silly, because if you were worried about how long a few modulo operations were going to take, you ought to be frantic about all that copying of the *perm* array. Roger Peel suggested the following altogether more sensible coding:

```
[2 * n] INT perm :
INT favourite :
SEQ
  SEQ i = 0 FOR n
    SEQ
      perm[i]     := i
```

```
      perm[n + i] := i
  --  0 <= i < 2n   ==>  perm[i] = i\n
  favourite := 0
  WHILE busy
    SEQ
      PRI ALT index = favourite FOR n
        VAL shifted IS perm[index] :
        c[shifted] ? x[shifted]
          P(shifted)
      favourite := perm[favourite + 1]
```

which simply precomputes the results of all of the modulo operations. The cost is in the extra storage required for the array.

Michael Goldsmith suggests yet another way of implementing the same selection strategy, guaranteeing $1/n$th of the bandwidth to any saturated channel. It is essentially the same as Roger Peel's solution, but without the need to keep the precalculated remainders:

```
INT split :
SEQ
  split := 0
  WHILE busy
    SEQ
      PRI ALT
        ALT index = split FOR n - split
          c[index] ? x[index]
            P(index)
        ALT index = 0 FOR split
          c[index] ? x[index]
            P(index)
      split := (split + 1) \ n
```

These schemes can be adapted to implement different scheduling strategies, for example:

```
INT favourite :
SEQ
  favourite := 0
  WHILE busy
    PRI ALT index = favourite FOR n
      VAL shifted IS index \ n :
      c[shifted] ? x[shifted]
        SEQ
          P(shifted)
          favourite := (shifted + 1) \ n
```

which gives the lowest priority next time around to the branch which was selected this time. That has the advantage over the previous scheme that if p of the n channels are carrying high volumes of traffic, it will tend to give $1/p$th of the bandwidth to each of them.

Harold Curnow suggested another more traditional looking way of getting this stronger guarantee:

```
[n]BOOL mask :
BOOL reset :
SEQ
  SEQ i = 0 FOR n
    mask[i] := TRUE
  reset := FALSE
  WHILE busy
    PRI ALT
      ALT index = 0 FOR n
        mask[index] & c[index] ? x[index]
          SEQ
            P(index)
            mask[index] := FALSE
            reset := TRUE
      reset & SKIP
        SEQ
          reset := FALSE
          SEQ i = 0 FOR n
            mask[i] := TRUE
```

which is much more in the spirit of what Chalmers was trying to do. It guarantees no more than $p-1$ missed opportunities between successive selections of a given one of p saturated channels.

If all you want is non-starvation, and the $1/n$th bandwidth guarantee will do, my favourite coding is simpler:

```
INT favourite :
SEQ
  favourite := 0
  WHILE busy
    SEQ
      PRI ALT
        c[favourite] ? x[favourite]
          P(favourite)
        ALT index = 0 FOR n
          c[index] ? x[index]
            P(index)
      favourite := (favourite + 1) \ n
```

This keeps the time overheads down to a single modulo operation, and an extra branch in the ALT; and the space to a single INT variable. It relies on the idempotence of ALT – the fact that it does not matter much if a branch appears twice. (Bill Roscoe[6] uses this one, but attributes it to a message from me to an electronic mailing list; you will also find it in [4].)

The moral of the tale

Well, first of all, do not start from the assumption that you need to write your own scheduler; start from the required behaviour of the program, and you will probably find that it can be done cleanly with the existing scheduler.

Secondly, the longer and more complicated your program and the woollier your idea of what it is meant to do, the less likely you are to get it right. This is particularly true of schedulers, as witness the embarrassing confusion which is the transputer's own scheduler, a sorry tale related in part in [1].

One of my colleagues (a year or two older than me, perhaps) said that there was a time so far back I would hardly remember it (he meant the sixties, I think) when the people building compilers and operating systems would insist on writing their systems in assembler. It was one thing for computing scientists to write toy systems in high level languages like BCPL (well, this was the sixties) but real systems had to be written in assembler, or even better in signed binary. My colleague thinks that that battle has been won, but then he does not know about Helios, and transputer-code schedulers.

Acknowledgement

I am grateful to a number of electronic-mail correspondents who contributed to this article, although since they have not all had an opportunity to correct my mistakes I lay a personal claim to those.

References

[1] Geoff Barrett, Michael Goldsmith, Geraint Jones, Andrew Kay, *The meaning and implementation of PRI ALT in occam*, in "occam and the transputer – research and applications" (Proc 9th technical meeting of the occam user group), ed. Charlie Askew, Southampton, 1988.

[2] Alan Chalmers, *Useful Titbits*, oug newsletter, pp.15–19, Summer 1988.

[3] INMOS Ltd., *Communicating Process Architecture*, Prentice Hall International, 1988.

[4] Geraint Jones, Michael Goldsmith, *Programming in occam2*, Prentice Hall International Series in Computer Science, 1988.

[5] David May, *Communicating Processes and occam*, draft of April 1986, subsequently INMOS technical note 20, subsequently in [3].

[6] Bill Roscoe, *Routing Messages Through Networks: An Exercise in Deadlock Avoidance*, in "Parallel Programming of Transputer Based Machines" (Proc 7th technical meeting of the occam user group), ed. Traian Muntean, Grenoble, 1987.