Monitoring communication

```
--  This chapter contains the code of a simple sorting program
--  implemented by a tree-shaped array of processes, and the code
--  needed to monitor on a screen the execution of the sorter.
--

--
--  configuration constants
--

-- size and shape of the tree

VAL INT depth.of.tree       IS 3 :
VAL INT number.of.leaves    IS 1 << depth.of.tree :
VAL INT number.of.forks     IS number.of.leaves - 1 :
VAL INT number.of.processes IS number.of.forks + number.of.leaves :
VAL INT number.of.channels  IS number.of.processes :

-- indexing of the tree

VAL INT root       IS 0 :
VAL INT first.fork IS root :
VAL INT first.leaf IS first.fork + number.of.forks :

-- and for the monitoring code

VAL INT collect IS 0 :
VAL INT deliver IS 1 :

VAL INT number.of.probes IS number.of.channels + number.of.leaves :

VAL INT first.channel.probe IS 0 :
VAL INT first.leaf.probe    IS number.of.channels :
```

```
--
-- protocols
--

PROTOCOL INT.STREAM
  CASE
    another.int; INT
    no.more.ints
:

PROTOCOL DISPLAY.STREAM
  CASE
    display.number; INT
    display.empty
    display.stop
:

PROTOCOL MULTIPLEX.STREAM
  CASE
    multiplex.number; INT; INT                -- probe; data
    multiplex.empty; INT                      -- probe
    multiplex.stop                            -- all probes stopped
:

VAL INT field.width IS 3 :

PROTOCOL TERMINAL.STREAM
  CASE
    terminal.item; INT; INT; [field.width]BYTE  -- x; y; characters
    terminal.stop
:
```

```
--
-- leaf processes
--

PROC leaf(CHAN OF INT.STREAM from.parent, to.parent,
          CHAN OF DISPLAY.STREAM probe              )
  SEQ
    from.parent ? CASE

      INT number :
      another.int; number
        SEQ
          probe ! display.number; number

          from.parent ? CASE no.more.ints

          to.parent ! another.int; number
          probe ! display.empty

      no.more.ints
        SKIP

    to.parent ! no.more.ints
    probe ! display.stop
:

--
-- fork processes
--

VAL INT left IS 0 :
VAL INT right IS 1 :

PROC fork.distribute(CHAN OF INT.STREAM from.parent,
                     [2]CHAN OF INT.STREAM to.children )
  INT child :
  BOOL more :
  SEQ
    child, more := left, TRUE
    WHILE more
      from.parent ? CASE
        INT number :
        another.int; number
          SEQ
            to.children[child] ! another.int; number
            child := child >< (left >< right)
        no.more.ints
          more := FALSE
    PAR child = left FOR 2
      to.children[child] ! no.more.ints
:
```

```
PROC fork.gather(CHAN OF INT.STREAM to.parent,
                 [2]CHAN OF INT.STREAM from.children )
  [2]BOOL more :
  [2]INT minimum :
  SEQ
    PAR child = left FOR 2
      from.children[child] ? CASE
        another.int; minimum[child]
          more[child] := TRUE
        no.more.ints
          more[child] := FALSE
    WHILE more[left] OR more[right]
      IF child = left FOR 2
        VAL INT other IS child >< (left >< right) :
        more[child] AND
                ((NOT more[other]) OR
                 (minimum[child] <= minimum[other]))
          SEQ
            to.parent ! another.int; minimum[child]
            from.children[child] ? CASE
              another.int; minimum[child]
                SKIP
              no.more.ints
                more[child] := FALSE
    to.parent ! no.more.ints
:

PROC fork(CHAN OF INT.STREAM from.parent, to.parent,
          [2]CHAN OF INT.STREAM from.children, to.children )
  SEQ
    fork.distribute(from.parent, to.children)
    fork.gather(to.parent, from.children)
:
```

```
--
--  monitor process
--

PROC monitor(CHAN OF INT.STREAM up.collect, down.collect,
                                up.deliver, down.deliver,
             CHAN OF DISPLAY.STREAM probe                )

  PROC copy.monitoring(CHAN OF INT.STREAM collector, deliverer,
                       CHAN OF DISPLAY.STREAM probe            )
    BOOL more :
    SEQ
      more := TRUE
      WHILE more
        collector ? CASE
          INT number :
          another.int; number
            SEQ
              probe ! display.number; number
              deliverer ! another.int; number
              probe ! display.empty
          no.more.ints
            more := FALSE
      deliverer ! no.more.ints
  :

  SEQ
    copy.monitoring(up.collect, up.deliver, probe)
    copy.monitoring(down.collect, down.deliver, probe)
    probe ! display.stop
:
```

```
--
--  multiplex process
--

PROC multiplex([]CHAN OF DISPLAY.STREAM probe,
               CHAN OF MULTIPLEX.STREAM all.probes )
  INT active.probes :
  [number.of.probes]BOOL more.from :
  SEQ
    active.probes := number.of.probes
    SEQ i = 0 FOR number.of.probes
      more.from[i] := TRUE
    WHILE active.probes > 0
      ALT i = 0 FOR number.of.probes
        more.from[i] & probe[i] ? CASE
          INT number :
          display.number; number
            all.probes ! multiplex.number; i; number
          display.empty
            all.probes ! multiplex.empty; i
          display.stop
            active.probes, more.from[i] := active.probes - 1, FALSE
    all.probes ! multiplex.stop
:

--
--  terminal-independent display processes
--

INT, INT FUNCTION make.cartesian(VAL INT index)
  INT x, y :
  VALOF
    IF
      IF line = 1 FOR depth.of.tree + 1
        index < ((1 << line) - 1)
          VAL INT column IS index - ((1 << (line - 1)) - 1) :
          VAL INT spread IS number.of.leaves >> (line - 1) :
          x, y := ((2 * column) + 1) * spread, line
      index >= number.of.channels
        VAL INT column IS index - number.of.channels :
        x, y := (2 * column) + 1, depth.of.tree + 2
    RESULT x, y
:
```

```
PROC independent(CHAN OF MULTIPLEX.STREAM source,
                 CHAN OF TERMINAL.STREAM sink    )
  [field.width]BYTE blanks :
  BOOL running :
  SEQ
    SEQ i = 0 FOR field.width
      blanks[i] := '*s'
    running := TRUE

    WHILE running
      source ? CASE

        INT index, number :
        multiplex.number; index; number
          INT x, y :
          [field.width]BYTE buffer :
          SEQ
            x, y := make.cartesian(index)
            make.decimal.string.int(buffer, number)
            sink ! terminal.item; x; y; buffer

        INT index :
        multiplex.empty; index
          INT x, y :
          SEQ
            x, y := make.cartesian(index)
            sink ! terminal.item; x; y; blanks

        multiplex.stop
          running := FALSE

    sink ! terminal.stop
:

--
--  terminal-dependent display processes
--

VAL INT virtual.height IS depth.of.tree + 1 :
VAL INT virtual.width  IS (2 * number.of.leaves) - 1 :

VAL INT screen.height IS 24 :
VAL INT screen.width  IS 80 :

VAL INT height.scale  IS
               (screen.height - 1) / virtual.height :
VAL INT width.scale   IS
               (screen.width - field.width) / virtual.width :
```

```
PROC clear.screen(CHAN OF BYTE terminal)
  -- clear screen sequence for an ANSI terminal
  write.string(terminal, "*#1B[2J")
:

PROC move.cursor(CHAN OF BYTE terminal, VAL INT x, y)
  -- lefthanded co-ordinates, origin 0,0 at top left
  CHAN OF DATA.ITEM c :
  PAR
    write.formatted(terminal, "*#1B[%d;%dH", c)
    SEQ
      c ! data.int; y + 1
      c ! data.int; x + 1
:

PROC dependent(CHAN OF TERMINAL.STREAM source,
               CHAN OF BYTE terminal          )
  -- terminal-dependent code for driving ANSI terminal
  BOOL more :
  SEQ
    clear.screen(terminal)
    more := TRUE
    WHILE more
      source ? CASE
        INT x, y :
        [field.width]BYTE buffer :
        terminal.item; x; y; buffer
          SEQ
            move.cursor(terminal, (x - 1)  * width.scale,
                (virtual.height - (y - 1)) * height.scale )
            write.string(terminal, buffer)
        terminal.stop
          more := FALSE
    move.cursor(terminal, 0, screen.height - 1)
:

--
--  display process
--

PROC display(CHAN OF MULTIPLEX.STREAM source, CHAN OF BYTE sink)
  CHAN OF TERMINAL.STREAM internal :
  PAR
    independent(source, internal)
    dependent(internal, sink)
:
```

```
--
--  driver
--

VAL INT mask IS BITNOT ((BITNOT 0) << 9) :

PROC shift(INT state)
  SEQ i = 1 FOR 9
    state := ((state << 1) /\ mask) \/
                                (((state >> 4) >< (state >> 8)) /\ 1)
:

PROC driver(CHAN OF INT.STREAM up.to.tree, down.from.tree)
  VAL INT second IS ... :  -- number of clock ticks per second
  TIMER clock :
  SEQ
    INT event, number :
    SEQ
      clock ? event
      number :=  (event /\ mask) \/ 1
      SEQ i = 0 FOR number.of.leaves
        SEQ
          event := event PLUS second
          shift(number)
          up.to.tree ! another.int; number
          clock ? AFTER event
      up.to.tree ! no.more.ints
    INT event :
    SEQ
      clock ? event
      SEQ i = 0 FOR number.of.leaves
        SEQ
          event := event PLUS second
          INT discard :
          down.from.tree ? CASE another.int; discard
          clock ? AFTER event
      down.from.tree ? CASE no.more.ints
:
```

```
--
--  structure of the program
--

[2][number.of.channels]CHAN OF INT.STREAM up, down :
[number.of.probes]CHAN OF DISPLAY.STREAM probe :
CHAN OF MULTIPLEX.STREAM all.probes :

PAR
  --
  --  sequential driver process
  --
  driver(up[collect][root], down[deliver][root])
  --
  --  parallel sorter
  --
  PAR
    --
    --  fork nodes in the tree
    --
    PAR i = first.fork FOR number.of.forks
      VAL INT first.born IS (2 * i) + 1 :
      fork(up[deliver][i], down[collect][i],
           [down[deliver] FROM first.born FOR 2],
           [up[collect] FROM first.born FOR 2]   )
    --
    --  leaf nodes in the tree
    --
    []CHAN OF DISPLAY.STREAM leaf.probe IS
            [probe FROM first.leaf.probe FOR number.of.leaves] :
    PAR i = first.leaf FOR number.of.leaves
      leaf(up[deliver][i], down[collect][i],
                                   leaf.probe[i - first.leaf] )
    --
    --  monitoring buffers
    --
    []CHAN OF DISPLAY.STREAM channel.probe IS
            [probe FROM first.channel.probe FOR number.of.channels] :
    PAR i = root FOR number.of.channels
      monitor(up[collect][i], down[collect][i],
              up[deliver][i], down[deliver][i], channel.probe[i] )
  --
  --  display control processes
  --
  multiplex(probe, all.probes)
  display(all.probes, terminal.screen)
```