9

Handling interrupts

It used to be that programmers met concurrently executing processes only if they had to code interrupt routines, or to write code which shared store with interrupt routines. An interrupt is a mechanism designed to make small amounts of processing power available at short notice to handle urgent tasks, when it would be unreasonably expensive to make that processing capability permanently available. To this extent, it separates two concerns: an applications programmer wanting to send characters to a lineprinter need only supply them to an interrupt handler; it is the responsibility of the interrupt handler to transmit them to the printer at the precise times that the printer indicates that it is ready for them. In this way, the programmer is relieved of the burden of making frequent checks on the state of the printer, and the structure of his program can be unaffected by the timing constraints imposed by the printer.

Interrupt routines are notoriously difficult to code and to use. In addition to assuming responsibilities of meeting real-time deadlines, the interrupt routine must maintain the programmer's illusion that the application program has exclusive use of the processor and store. This imposes rigorous discipline on the use of registers, and of store locations, both to avoid conflicts, and in the management of those shared variables by which program and interrupt routine communicate. Moreover, the interrupt routine has usually to be programmed as a 'subroutine' (rather than a 'coroutine') invoked once by each interrupt, which means that any state that is to persist from the handling of one interrupt to that of the next must be saved in store and reconstructed at the next interrupt. To make matters worse, high level programming languages are rarely able to offer convenient abstractions for coding interrupt handlers, which are inherently machine-dependent, and it is usual for interrupt routines to be written in machine code.

It is tempting to claim that the concurrently executed processes of occam are the right tools for writing interrupt routines. To do so would be misleading: concurrent processes are right for a task for which interrupt routines have always been inadequate! The task is in two parts: that of writing code to meet real-time deadlines; and that of isolating their effects, so as to keep the rest of the program simple.

In occam, sustaining the illusion that the application program has exclusive use of the processor and store is easily done, since each and every process of every occam program operates under this very illusion. The illusion is sufficiently strong that a programmer need never know whether or not any particular process is executed on its own dedicated processor.

Meeting real-time deadlines remains a problem that must be solved by ensuring that each processor is fast enough, and that the code is short enough. Apart from this concern with urgency, an interrupt handler coded in occam can be written in exactly the same way as any other process, and communicates with the application program in the same way as any other processes communicate with each other.

Managing terminal input

To take a concrete example, consider managing the traffic to and from a terminal. Every time a key is struck at the keyboard, there will be a corresponding event (traditionally an interrupt) in the computer, and some action must be taken to read information about the key before the next key is struck, lest the information be lost. Quite often the action taken will be to store the character corresponding to the key in a buffer, from which it will subsequently be read at the leisure of the program which is consuming the terminal input. The capacity of the buffer determines how many characters can be 'typed ahead' of the demand from the program.

In occam, an 'event in the computer' is represented by a communication on a special channel which PLACE locates at some implementation-dependent address (such as that of the relevant peripheral controller).

```
CHAN OF BYTE keystroke.in :
PLACE keystroke.in AT keyboard.interface :
```

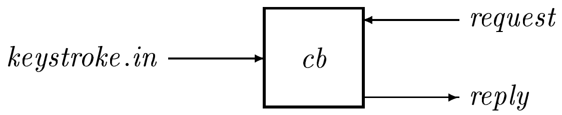
Programs use special channels just as they would use other channels, except that they use each channel only for input, or only for output, with the other half of the communication being performed by a process implemented in the hardware. On a transputer, this might be a 'link' to some other device.

In the case of the terminal example it would be possible, every time a key was struck, for the program to perform an input

```
keystroke.in ? char
```

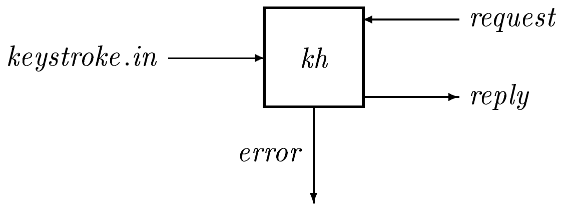
so a reasonable interrupt handler might be

```
circular.buffer(keystroke.in, request, reply)
```



using the circular buffer, Process 8.8 from the preceding chapter. This process has the disadvantage that, were the buffer to become full through the coincidence of a fast typist and a slow program, the process would no longer be prepared to accept input from *keystroke.in*. Since there is (fortunately) no mechanism built into present-day terminals to suspend the execution of the typist while the computer is busy, this would mean that keystrokes made whilst the buffer was full would be lost, without warning.

An improved scheme would be to code the interrupt handler in such a way that it was always prepared to acknowledge the keystroke, and to take some remedial action in case there were no room left in the buffer.



Process 9.1 will signal on the *error* channel if an attempt is made to overfill the type-ahead buffer; later, that signal will be used to cause the bell on the terminal to be rung. Notice that the *keyboard.handler* is written in such a way that, provided

- the outputs to *error* and *reply* are never delayed for more than a fixed time
- this process executes at a known rate within a known short time of becoming ready

then it is possible to put a bound on the length of time which can pass before the process next becomes ready to accept an input from *keystroke.in*. Bounds of this kind are what you would need to demonstrate that no interrupts were lost without trace.

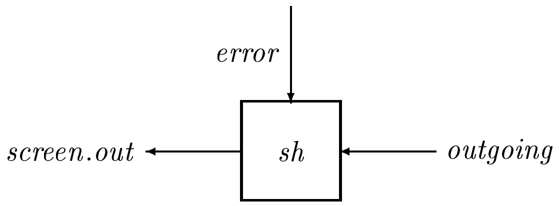
Managing terminal output

For the purpose of this example, suppose that the outgoing traffic to the terminal screen consists of a sequence of bytes passing along the special channel

Process 9.1

```
VAL INT keyboard.interface IS ... :
                          -- some hardware address
VAL INT type.ahead IS ... :
                          -- number of characters that can
                          -- be typed ahead of being wanted
PROTOCOL SIGNAL
  CASE
    signal
:

PROC keyboard.handler(CHAN OF SIGNAL request,
                      CHAN OF BYTE reply,
                      CHAN OF SIGNAL error   )

  -- Characters typed at the keyboard can be read from
  -- reply. A signal is required on request before each
  -- item is read. If more than type.ahead are typed
  -- ahead, an error is signalled.

  CHAN OF BYTE keystroke.in :
  PLACE keystroke.in AT keyboard.interface :
  INT reader, writer, count :
  SEQ
    reader, writer, count := 0, 0, type.ahead

    [type.ahead]BYTE datum :
    WHILE TRUE
      ALT

        BYTE lost :
        count = 0 & keystroke.in ? lost
          error ! signal

        count > 0 & keystroke.in ? datum[writer]
          count, writer := count - 1, (writer + 1) \ type.ahead

        count < type.ahead & request ? CASE signal
          SEQ
            reply ! datum[reader]
            count, reader := count + 1, (reader + 1) \ type.ahead
:
```

screen.out to be displayed as characters on a screen, or acted upon in some other way by the terminal. The terminal may then become busy for some short time, before again being ready to accept output. The screen handling process has to accept characters from the user's program, and to pass them on; additionally, it must accept urgent error signals from the process handling the type-ahead buffer, and send a 'bell' character to the terminal when an error is flagged.



Process 9.2

```
PROC screen.handler(CHAN OF BYTE outgoing,
                    CHAN OF SIGNAL error  )
  CHAN OF BYTE screen.out :
  PLACE screen.out AT vdu.interface :
  VAL BYTE bell.character IS '*#07' :
  WHILE TRUE
    ALT
      BYTE char :
      outgoing ? char
        screen.out ! char
      error ? CASE signal
        screen.out ! bell.character
:
```

It might appear that there are no timing constraints on the behaviour of the screen handling process (Process 9.2), but recall that the performance of the *keyboard.handler* depends on its *error* signals not being unduly delayed. As written, the *error*-guard in the *screen.handler* might indeed be delayed indefinitely, even were it guaranteed that the *screen.handler* was executed immediately either of the guards became ready. It might be that the process that sends characters along the *outgoing* channel is able to send a new character in less time than it takes the *screen.handler* to execute the body of its `WHILE` loop once. In that case, the *outgoing*-guard would always be ready every time the alternative was executed, and since an alternative can choose any one of the ready guards, it is possible that the *error*-guard might be ignored indefinitely, even were it ready. Notice, particularly, that this behaviour is not caused by writing the *outgoing*-guard first: the order of the components of an alternative is immaterial to its meaning.

It is for just this reason that occam has the additional `PRI ALT` construct (see Chapter 7), which breaks the symmetry between its branches. Using this, the *screen.handler* can be re-written as Process 9.3 and then an *error* signal could not be delayed for longer than it takes to execute the body of the `WHILE` loop once. Discharging the responsibility to accept these signals in a fixed time reduces to showing that

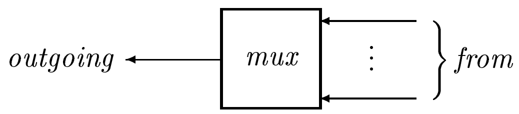
- the outputs to *screen.out* are never delayed for more than a fixed time
- this process executes at a known rate within a known short time of becoming ready

The first requirement is met by the terminal, by assumption; the second will be discussed later. Notice that there is no constraint on the timing of transactions on the *outgoing* channel; the aim is to build a firewall around the terminal, beyond which meeting real-time deadlines will no longer be a concern.

Process 9.3

```
PROC screen.handler(CHAN OF BYTE outgoing,
                    CHAN OF SIGNAL error  )
  CHAN OF BYTE screen.out :
  PLACE screen.out AT vdu.interface :
  VAL BYTE bell.character IS '*#07' :
  WHILE TRUE
    PRI ALT
      error ? CASE signal
        screen.out ! bell.character
      BYTE char :
      outgoing ? char
        screen.out ! char
:
```

A particular program that uses the terminal may contain a large number of processes, each needing to send characters to the terminal screen. Since the *outgoing* channel is now the only way out to the terminal, and since only one process is able to send along that channel, a process must be written to interleave the many output streams, and send their interleaving along *outgoing*.



Process 9.4 interleaves messages (sequences of characters) from each of the *from* channels, in an arbitrary order, each message being terminated by the *release* value. (One could use on each *from* channel a `BYTE.STREAM` protocol, like `INT.STREAM` except for bytes, or an integer type into which the data bytes can be coerced, using -1, say, for the *release* value. Our solution avoids the extra complexity and overheads at the expense of the – probably correct – assumption that nobody is going to want to output an ASCII null character.) The most interesting property of this process, for our present purpose, is that

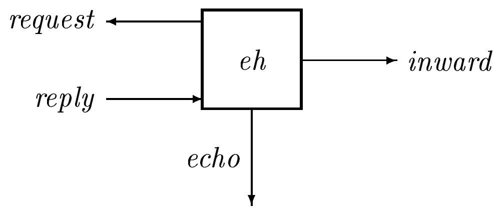
Process 9.4

```
VAL BYTE release IS 0(BYTE) :

PROC output.multiplexer([]CHAN OF BYTE from,
                        CHAN OF BYTE outgoing )
  WHILE TRUE
    ALT selected.process = 0 FOR SIZE from
      BYTE char :
      from[selected.process] ? char
        WHILE char <> release
          SEQ
            outgoing ! char
            from[selected.process] ? char
:
```

it is outside the firewall: there are no constraints on the speed with which it executes, nor on the times at which other processes communicate with it.

Managing echoing

The time-dependency firewall is not yet complete: there remains the problem of reading from the type-ahead buffer. Recall that, having issued a *request* signal, the reader assumes a responsibility to accept the reply from the *reply* channel within a fixed time. That means the reader must be within the firewall.



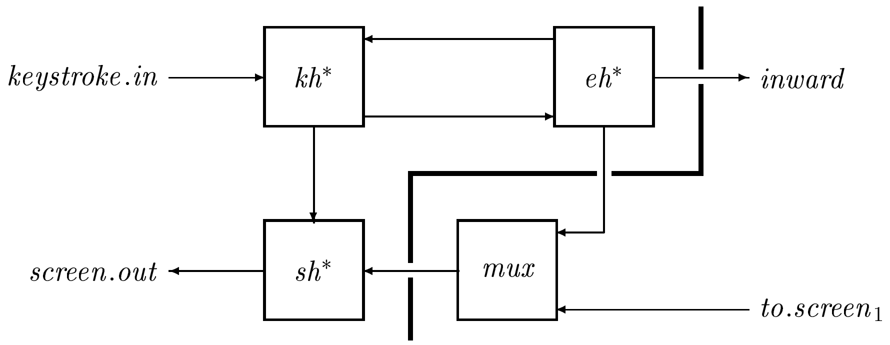
Process 9.5 reads each character from the type-ahead buffer by signalling a *request* and accepting a *reply*. The character is passed on to the *inward* channel to the user. This process is given the additional job of 'echoing' the printable characters to the terminal screen as and when they are read by the program using the keyboard input. The only timing constraint on this process is that it execute sufficiently rapidly that the input from *reply* is accepted within a permissible time of acceptance by the *keyboard.handler* of the preceding *request*. There being no constraints on communication on the *echo* and *inward* channels, these may cross the firewall: the *echo* channel is to be one of the array *from* going to the *output.multiplexer*, and the *inward* channel can be used directly by the process that consumes keyboard input.

Process 9.5

```
PROC echo.handler(CHAN OF SIGNAL request,
                  CHAN OF BYTE reply, echo, inward )
  VAL BYTE enter IS '*c' :
  WHILE TRUE
    BYTE char :
    SEQ
      request ! signal
      reply ? char
      inward ! char    -- Transmit character to user
      IF
        ('*s' <= char) AND (char <= '~')
          echo ! char       -- Echo visible input to screen
        char = enter
          SEQ
            echo ! '*c'     -- Echo carriage return ...
            echo ! '*n'     -- ... and newline
            echo ! release  -- Release screen at end of line
        TRUE
          SKIP              -- No action on other characters
:
```

Thus Figure 9.1 represents the completed scheme for the terminal handling processes, showing the firewall. The starred processes are those that have to execute at a known rate within a known time of being ready.

Figure 9.1 The firewall in place



Notice that the screen sharing strategy is implemented by an 'ordinary' process not subject to any timing constraints. Since each line of echoed characters from the type-ahead buffer is sent to the screen as an indivisible message, there is no problem about input characters being mixed in with output, but neither is there any need for the *echo.handler* to be concerned with screen allocation. The user process must ensure that it releases its channels to the multiplexer before accepting keyboard input, and that it consumes whole lines of input before attempting further output. If the program that used the terminal were written as the procedure *user*, then the whole could be put together with the terminal handler to form Process 9.6.

Process 9.6

```
VAL INT keyboard.interface IS ... : -- hardware address
VAL INT vdu.interface      IS ... : -- hardware address

VAL INT type.ahead IS ... :  -- size of buffer
VAL BYTE release IS 0(BYTE) :

PROC keyboard.handler(CHAN OF SIGNAL request,
                      CHAN OF BYTE reply,
                      CHAN OF SIGNAL error   )
  ...
:
PROC echo.handler(CHAN OF SIGNAL request,
                  CHAN OF BYTE reply, echo, inward )
  ...
:
PROC output.multiplexer([]CHAN OF BYTE from,
                        CHAN OF BYTE outgoing )
  ...
:
PROC screen.handler(CHAN OF BYTE outgoing,
                    CHAN OF SIGNAL error  )
  ...
:
PROC user(CHAN OF BYTE terminal.keyboard, terminal.screen)
  ...
:

CHAN OF BYTE reply, outgoing, from.keyboard :
CHAN OF SIGNAL request, error :
VAL INT from.echo.handler IS 0 :
VAL INT from.user         IS 1 :
VAL INT number.of.outputs IS 2 :
[number.of.outputs]CHAN OF BYTE to.screen :

PAR
  keyboard.handler(request, reply, error)     --*
  echo.handler(request, reply,
               to.screen[from.echo.handler],
               from.keyboard                ) --*
  output.multiplexer(to.screen, outgoing)
  screen.handler(outgoing, error)             --*
  user(from.keyboard, to.screen[from.user])
```

Configuring the program

Ignoring, for the moment, the timing constraints imposed by the proper handling of the terminal interrupts, checking the correctness of this program can be done in two parts. First of all, there are properties of individual processes that can be checked in isolation from the other processes: for example, that the *echo.handler* performs a cycle of communications in a fixed order, behaviour that is unaffected by the other processes. Secondly, there are some properties that are inherently global, notably freedom from deadlock, which may depend on the behaviour of every one of the processes.

The same is the case with the timing constraints: the argument thus far has been about each of the component processes, more or less in isolation. If we had in mind a particular implementation of occam on a particular computer, and with a particular set of terminal characteristics, then we could calculate the 'fixed times' within which actions must occur as so many seconds of processor time, so much communication time, and so on. It remains, however, to be demonstrated that there will always be sufficient processor time available when it is required.

One way of achieving this would be to dedicate a processor to the execution of each of the five components of the program. This is where the `PLACED PAR` variant of the parallel constructor, introduced in Chapter 7, would be useful. To indicate such a division of labour, a placed parallel could be used instead of the `PAR` in the present program, together with other housekeeping annotations such as assigning processor numbers and fixing the inter-processor channels onto hardware communication links with `PLACE`. The result would be that the processor occupancy times calculated for the three starred processes would be actual elapsed times, each independent of the processor loading of the other processes. In this particular case, such a solution seems excessive, since the tasks are each fairly simple, and the traffic is light. It would be a more reasonable way of dealing with, say, the traffic to and from a fast disk, where a whole transputer might be allocated to managing the large volumes of data, and the potentially intricate calculations required to make efficient accesses to the disk.

More realistically, this particular program would probably be run on a single processor, say one transputer. As it stands, in order to be able to guarantee sufficient speed of execution in the starred processes, it is necessary to know details of the behaviour of the unstarred processes: for example, that the *user* process does not require more than a known proportion of the processor's time. This being unsatisfactory, we can use the other `PAR` variant from Chapter 7: `PRI PAR`, which distinguishes more and less urgent tasks. In occam programs to be executed on current transputers, asymmetric parallel constructs can have no more than two components, corresponding to the two process-queues in the transputer. For that machine, the right way to organize the terminal handler would be Process 9.7.

Process 9.7

```
PRI PAR

  PAR                          -- High priority process
    keyboard.handler(request, reply, error)
    echo.handler(request, reply,
                 to.screen[from.echo.handler],
                 from.keyboard                   )
    screen.handler(outgoing, error)

  PAR                          -- Low priority process
    output.multiplexer(to.screen, outgoing)
    user(from.keyboard, to.screen[from.user])
```

Now it cannot matter what the *user* or *output.multiplexer* processes do: if any of the urgent processes is able to execute, then one of them will do so within a very short time. This latency will be determined and guaranteed by the implementation, so again, with a particular implementation in mind, this would be known. The total waiting time, for any of the 'interrupt' processes, between becoming ready and beginning to execute, is bounded by the sum of one latency time and the sum of the longest execution time of each of the other interrupt processes.

That completes the analysis of the timing of the program. All that is needed in the case of a particular implementation is to calculate the times, a matter of counting instructions, which task could and should be delegated to the compiler. Substituting the figures for the waiting and execution times allows a check to be made that the required response times are achieved.