Concurrent Processes for Icon*

Kelvin Nilsen

TR 88-11

ABSTRACT

Icon is well-known for its high-level approach to non-numeric programming. To increase the application domain of Icon to include communications and other real-time programming problems, concurrent processing capabilities have been added to an experimental version of Icon called Conicon. The concurrent processing language features added to Icon are natural extensions of the existing language. These new features add concurrency to Icon by augmenting Icon's existing goal-directed expression evaluation mechanisms. Sequential programs are, for the most part, unaffected by the new capabilities. Concurrent programs written in Conicon benefit from all of the same high-level language capabilities as sequential programs. Consistent with the design of Icon, the design of these new language features has focused on increasing programmer expressiveness. This document describes the new language features and provides several examples of how these capabilities are used.

February 12, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

Concurrent Processes for Icon

Introduction

The addition of concurrent processing capabilities to Icon was motivated by several different, but related, goals. One of these goals was to extend the high-level programming capabilities of Icon to the application domain of communications software. Computers that communicate with external systems, whether the external system represents another computer, a person speaking at a microphone, or a collection of sensors measuring the temperature and pressure of various critical locations inside a nuclear reactor, must conform to some degree with the scheduling constraints imposed by the remote system. A goal of this work is therefore to provide real-time response to sections of code for which response time is critical. Although it would be possible to satisfy this goal by requiring that every line of code in a large system of software execute in constant time, this places a large burden on the programmer, and significantly restricts the repertoire of Icon capabilities that might be employed to solve a particular problem. Detailed analysis and measurements are required of a programmer that claims to have implemented code that executes in real time. One approach toward minimizing the efforts required to justify claims of real-time performance is to focus attention only on portions of the system that are time critical. In these time-criticial sections of code, Icon capabilities that do not perform in constant time can be avoided. These capabilities include table lookups and goal-directed evaluation of expressions with an unlimited or very large number of alternatives.

For example, a network name server may require several data base lookups, some of which access data on remote machines. Users of this service do not expect an instantaneous turnaround. Likewise, a ping-pong-playing robot that collects statistics describing its opponent's weaknesses and expected behavior in order to formulate an effective strategy does its strategic planning only as time permits. The robot is not given an upper limit on the amount of time it can use to formulate a strategy. These are examples of applications that do not require real-time response. In both of these cases, the same software system that satisfies the requests described above must also be able to satisfy certain other demands in real time. For example, the network server to which name queries are directed might also be responsible for the exchange of data packets with other networked computers. If an arriving packet is not received and processed within a fixed amount of time, data buffers may overrun or communication protocols may be violated. Similarly, the robot's main priority is to move its paddle into the path of the opponent's shot in order to return the ball. It returns the shot in real time, using whatever strategy is available at the time it calculates its stroke, the default strategy being, for example, to return the ball to the position it came from.

The division of labor between activities that require real-time response and other activities that do not is easily made by dedicating different concurrent processes to each type of activity. Processes dealing with real-time activities execute with higher priority than the processes for which response time is less critical. The high-priority real-time processes generally spend the majority of their time blocked on I/O queues. However, when situations arise that call for their attention, these processes are granted access to the CPU until they once again become blocked waiting for I/O.

A second motivation for concurrent processes is that many real-time applications must monitor signals from a variety of different asynchronous sources. For example, a network server may simultaneously monitor conversations with several different remote computers. An aircraft flight control system must monitor velocity, altitude, and the condition of the aircraft's engines. It is possible to poll each possible source of information in a loop, dealing with whatever information becomes available when the device is polled. However, because each device may be polled many times without having information to report, time is wasted in this loop. More efficient utilization of CPU time is achieved by dedicating one process to every group of information sources. Such a process sleeps until information from one of the information sources in its group becomes available.

Concurrent processes are also useful in traditional applications that do not take advantage of the capabilities described above. For example, Icon evaluates expressions using a depth-first search strategy to traverse the tree of possible solutions to a compound goal. Certain problem domains benefit from a slightly different search strategy. For example, a depth-first search of all the computers connected by Unix[1] uucp connections for a path between two particular nodes might take an inordinately long time, even though a path of length three might be known to exist.

[1] Unix is a trademark of AT&T Bell Laboratories.

In this case, a breadth-first search is much more sensible. Concurrent processes are capable of simulating the most useful characteristics of coexpressions. Because of this, they provide most of the same capabilities offered by coexpressions for implementing alternative search strategies. In addition to the simulated capabilities of coexpressions, concurrent processes provide the ability to investigate different possible solution paths concurrently.

One final motivation for the addition of concurrent processing to Icon is that multi-processor hardware is becoming increasingly inexpensive and available. Providing programmers with linguistic mechanisms to describe concurrency is an important step toward making efficient use of this hardware. Being able to exploit multi-processor architectures is important because these architectures offer the potential for solving existing non-numeric problems faster, and because many problems that were previously considered economically unfeasible due to the limitations of current hardware might be solvable on more powerful multi-processor computers.

1. The Stream Data Type

An important decision in the design of any concurrent programming system is how the individual processes synchronize and communicate with each other. All global variables in a Conicon program are shared between processes. These global variables provide one method of communicating between processes.

Synchronization, though, is achieved by means of the stream data type [1]. Streams serve as message queues to be used by processes in exchanging information. Internal streams are created by the open function, which also serves to create streams of file data. Calling open with a null value in place of a file name creates an internal stream:

```
char_pipe := open()
```

Whenever the first argument to open is null, open's second argument specifies whether the internal stream represents arbitrary Icon values or only characters. Values are read from streams using the built-in functions probe and advance. These functions return strings when reading from streams of characters, and lists when reading from streams of arbitrary Icon values. The open modes are abbreviations for the types returned by probe and advance for each of these classes of streams respectively. To represent the idea that streams of arbitrary Icon values are referenced as lists or arrays, these streams are opened with an "a" mode. Streams of characters are opened using the "s" mode. The default mode is "s".

Messages are inserted into a stream using the write function. The following expression, for example, inserts nine characters into the stream char_pipe:

```
write(char_pipe, "some text")
```

With streams of characters, write returns the number of characters written to the stream. Characters may be read from the same pipe using the advance function, which advances the focus of interest for a stream to the position specified by its first argument returning whatever data is skipped over by advance as a string. Positions within a stream are numbered, as with strings, starting with one to represent the front position in the stream. The expression below advances the stream focus by three characters, returning the string of length 3 that represents the data between positions 1 and 4 within the stream.

```
s := advance(4, char_pipe)
```

open, if called with a null value as its first argument and the string "a" as its second argument, opens an internal stream of arbitrary Icon values. Values of any type can be written to or read from the stream. Each argument of write other than the first is appended to the end of the stream. When writing to streams of values, write returns the number of values actually written.

```
write(object_pipe, [1,2], "string", object_pipe)
```

The expression above, for example, writes 3 values to the end of the stream referenced by object_pipe. As with the other structured types of Icon, a stream value is a pointer to an internal data object. Assignment of this value does not make a copy. In the example above, object_pipe is written to itself, creating a pointer cycle.

Values are extracted from a pipe of this form using the advance function, which behaves as described above, except that it returns a list instead of a string. advance(1, object_pipe), for example, returns an empty list, leaving the current stream focus unchanged. The line below produces the next two items in the stream as a list, advancing

the stream focus to position 3 relative to its previous focus:

```
two_items := advance(3, object_pipe)
```

With either type of stream, an attempt to advance the focus beyond the end of the stream results in failure. This can be used, for example, to control looping:

```
while x := advance(2, object_pipe) do process(x[1])
```

This loop terminates when object_pipe is closed. Attempts to advance an open stream's focus beyond the end of the data that is currently available from that stream simply blocks the process until additional data becomes available. Because the size of the internal buffer that holds data piped between processes is fixed[2], writing processes are blocked whenever they attempt to write to a stream more data than the stream is capable of buffering. These characteristics are used in the following example to implement semaphore operations[3]:

```
procedure P(sem)
  advance(2, sem)
end

procedure V(sem)
  write(sem, &null)
end
```

It is possible that multiple processes might block waiting for a P operation to succeed. All internal process queues are ordered by process priority. This allows, for example, a higher priority process that arrives on the queue later than some other process to advance more quickly to the front of the queue. Processes of the same priority are queued in FIFO order.

Conicon's built-in functions for accessing the data of a stream are atomic with respect to other operations accessing the stream. This means, for example, that the two values written to the stream mailer by the following function call appear contiguously when read from the stream, even though other processes may be writing to the stream at the same time.

```
write(mailer, "robert", msg)
```

The same is true for advance and probe invocations. Once a process has gained mutually exclusive read or write access to a stream, that process retains read or write ownership of the stream until it is done accessing the stream. Processes waiting for mutual read or write access to a stream are queued in order of decreasing process priority.

Occassionally, processes that need to read from or write to a stream must be able to do so without becoming blocked on internal queues waiting to perform the requested I/O. Special non-blocking versions of the standard stream accessing routines provide these capabilities. cadvance, cprobe, and cwrite are conditional versions of advance, probe, and write respectively. cwrite returns the number of values written without blocking, or fails if no values can be written at the moment. cadvance and cprobe return as much of the requested stream as is available, or fail if none of the desired stream data can be returned without temporarily putting the process to sleep.

The close function, described in [1] as it relates to streams representing operating system files, behaves similarly for internal streams. Attempts to write to a closed internal stream result in a fatal run-time error. Reading from a closed stream succeeds only if the requested data was available from the stream at the time it was closed. If, at the moment the stream is closed, processes are blocked waiting to write to the stream, close terminates with a run-time error.

---

[2] The default buffer size for streams of either characters alone or of arbitrary Icon values is 256. This value can be modified using the bound function described below.

[3] This implementation of a semaphore is only valid as long as the number of V operations does not exceed the number of P operations by more than the size of the stream's internal buffer.

2. Explicit Process Creation

Icon's create operator, which creates co-expressions out of its expression argument, has been modified in Conicon to instead create a concurrent process out of its expression argument. The following program, for example, intertwines two sequences of integers, the first sequence increasing from 1, and the second decreasing from −1.

```
procedure main()
  p1 := create every write (&output, seq(1), "\n")      # write positive integers
  p2 := create every write (&output, -seq(1), "\n")     # write negative integers
  every wait(p1 | p2)
end
```

The wait function in the above program waits for its process argument to die. In this example, wait never returns because p1 and p2 never terminate.

Several of Icon's keywords have been redefined in Conicon. For example, &main and &current refer to the main and current processes respectively instead of referring to co-expressions, and there is no &source keyword. The meaning of &subject has also been changed. Each process has its own private &subject, which is set, at process creation time, to the value of &subject in the creating process. &subject for the main process is set initially to &input.

Concurrent processes often serve as filters. A filter might be used, for example, to replace in a stream of characters each pair of consecutive as with the single letter b. A second filter, reading the output of the first, might replace every pair of consecutive bs with a single c. A solution to this problem that uses co-expressions to implement each filter is described in §13.4.2 of [2]. The solution presented here uses instead an internal stream of characters to connect the two filters, and creates two processes to act as the filters.

```
pipe := open()
create compact("a", "b", &input, pipe)
create compact("b", "c", pipe, &output)
```

The first filter reads from &input and writes to the internal stream pipe. The second filter reads from pipe and writes to &output.

```
procedure compact(s1, s2, in, out)
  local c1
  c1 := cset(s1)
  s1 ||:= s1
  in ? {
    while write(out, advance(upto(c1))) do {
      if advance(match(s1)) then write(out, s2)
      else write(out, advance(2))
      }
      write(out, advance(0))
    }
end
```

In the above application, the two filters execute independently of each other. If pipe empties, the second filter is automatically blocked until new characters become available. If the pipe reaches its full capacity, the first filter is blocked until characters are extracted from the pipe, making room for more characters to be written.

Many string processing problems can be divided naturally into multiple filters connected to each other by internal streams. For example, a lexical analyzer might be connected to a parser by a stream of tokens. In this example, the lexical analyzer executes as a process that reads from &input and writes tokens to an internal pipe. For this application, the stream that connects the two processes represents arbitrary Icon values. It might be opened using the following expression:

```
tokens := open( ,"|")
```

Suppose a procedure named get_token has been implemented by the programmer. On each invocation, this procedure simply reads from &input to the end of the next available token and returns the token as a string. Given the existence of get_token, the following expression starts up a process to perform lexical analysis:

```
create while write(tokens, get_token())
```

A backtracking parser of the tokens stream is described briefly in [1]. Because the expression argument of the create operator is evaluated in a goal-directed fashion, the process is capable of producing a sequence of results. At the time of its creation, a special output stream is allocated for each process. This stream automatically receives the results produced by the expression. The following line, for example, creates a process that writes to its output stream each token returned by the lexical analyzer.

```
p := create |get_token()
```

The function yield produces the output stream of a process. For example, yield(p) represents the output stream of process p. To read the next token available from the lexical analyzer, execute the following:

```
tok := advance(2, yield(p))[1]
```

Created processes are automatically killed when the expressions they are evaluating fail to produce further results. When the process is killed, its output stream is automatically closed. Closing of the output stream does not discard buffered values. It simply prohibits the writing of additional data to the stream, and causes any attempt to read data that is not available from the stream to be treated as an end-of-stream condition instead of blocking the reading process.

These capabilities provide the ability to mimic the behavior of coexpressions. For example, the Icon expression:

```
every write(find(s1, s2))
```

can be rewritten as:

```
p := create find(s1, s2)
while write(advance(2, yield(p))[1])
```

Because it is common to ask for only one result at a time from a process created in this manner, Icon's coexpression activation operator has been redefined to provide similar functionality in Conicon. The meaning of

```
@p
```

is simply:

```
advance(2, yield(p))[1]
```

Thus, yet another notation for the every expression above is:

```
p := create find(s1, s2)
while write(@p)
```

Conicon does not support the binary form of the @ operator.

Note that executing an expression as a separate process is not necessarily equivalent to executing that expression in an every expression. For example, the code below writes the time at which each invocation of compute begins. This might be used to measure the performance of compute.

```
every write(|&time) do
  compute()
```

According to the pattern established above, this might be rewritten as:

```
p := create |&time
while write(@p) do compute()
```

However, the output of this program is somewhat deceptive. Because a stream of buffered results connects the two processes, it is possible that a considerable delay might occur between invocation of &time and execution of the @

operator that produces the time for printing.

3. Concurrent Alternation

Icon's alternation operator generates all results of its left-hand expression followed by all results of its right-hand expression. The concurrent alternation operator generates all results of both its left and right-hand expressions, but both expressions are evaluated concurrently, and results are produced in whatever order the expression arguments produce them. Concurrent alternation is represented by the binary ! operator, which has the same operator precedence as Icon's more traditional alternation operator.

In standard Icon,

```
every write(1 | 2 | 3)
```

outputs the three lines:

```
1
2
3
```

Use of the concurrent alternation operator in place of Icon's standard alternation operator yields the same three lines, but in undefined order. For example, the output of:

```
every write(1 ! 2 ! 3)
```

might be:

```
3
1
2
```

The concurrent alternation operator creates a separate Icon process to evaluate each of its expression arguments. The processes created by this operator share a common output stream. Each expression is evaluated in a goal-directed fashion, producing all possible values and writing them to the shared output stream. The following expression, for example, writes all of the integers from 1 to 20, but the order of the written numbers depends on how the two processes are scheduled with respect to each other:

```
every write ((1 to 10) ! (11 to 20), "\n")
```

As processes exhaust the result sequences of the expressions they are evaluating, the processes kill themselves, decrementing a reference count on their shared output stream. The second process to decrement this reference count closes the stream. Remember that closing the stream only prevents other processes from writing additional data to the stream. It does not prevent other processes from reading the stream data that has already been buffered. Concurrent alternation therefore provides a concise notation for the creation and destruction of short-lived processes and communication with those processes while they are active.

In many contexts, only one result from an expression is desired. In standard Icon, if no more results are required from an expression capable of producing multiple results, evaluation of the expression simply aborts. For example, the following expression only obtains one result from the upto generator:

```
write(upto('aeiou', "a fool and his money are soon parted"))
```

However, the line below outputs each position at which a vowel appears in upto's string argument.

```
every write(upto('aeiou', "a fool and his money are soon parted"))
```

The difference between these two examples is that the first invocation of upto appears within a bounded expression that requires only one result. This same distinction is made for concurrent alternation. Whenever a bounded expression terminates, any processes created by a concurrent alternation operator within that bounded exrpession are automatically killed. For example, only one result is required from the expression below:

```
s := prompt_user() ! sleep(10000)
```

In this example, prompt_user displays some prompt to an interactive user and waits for a response. If this response comes within ten seconds (10000 milliseconds), it is returned as a string by prompt_user and assigned to s. Since only one result is required from this expression, sleep is automatically killed as control leaves this bounded expression. However, if sleep produces its result before prompt_user, then prompt_user is killed. This idiom provides a clear concise notation for exception handling. A similar construct might be used, for example, to monitor a modem's carrier signal while a concurrent process communicates via the modem.

```
if (watch_carrier() ! communicate()) = -1 then
  write("lost carrier\n")
```

The example above expects watch_carrier to sit quietly until it detects that carrier has been lost, at which time it returns -1. communicate, in this example, must return some value other than -1.

This paradigm can be carried even further. Because of the high computational complexity of many real-time problems, it is not possible to obtain solutions to these problems that work for all possible combinations of input data in a constant amount of time. However, it is possible to create solutions that work in constant time for some subset of the possible domain. Consider, for example, natural language parsing of voice phonemes. A procedure might be written to scan a stream of these phonemes, returning a parsed English sentence. If this procedure is able to find a sentence within, for example, 100 milliseconds, the real-time constraints imposed on the algorithm are satisfied. However, if no sentence can be found in that amount of time, the algorithm may be forced to simply ignore some initial sequence of the phonemes remaining to be parsed. This is modeled as:

```
if ((s := get_sentence()) ! sleep(100)) === &null then
  skip_to_pause()
```

In the above expression, skip_to_pause advances the focus within the stream of voice phonemes to the next period of silence exceeding some minimum duration. Before giving up entirely on parsing the phonemes, however, a quick check of the system status may reveal that computing resources are available for additional parsing. If, for example, after 100 milliseconds of unsuccessful parsing, few if any additional phonemes have arrived for processing, the decision might be made to continue parsing the input data that was previously available. This is achieved with the following:

```
every s := (get_sentence() ! |sleep(100)) do
  if \s | heavy_load() then
    break
```

This loop terminates only if get_sentence parses a complete sentence or if the system is perceived to be heavily loaded at one of the timeout points. Note that a result becomes available from the concurrent alternation operator at least once every 100 milliseconds.

Similar constructs might be used in voice synthesis. In order to give proper duration, intonation, volume, and accents to the synthesized reading of English text, some natural language parsing is required. However, if some sentence is particularly difficult to parse, or some word is not found in the system's dictionary, then the program might attempt a simpler, less capable algorithm. Suppose the simple algorithm is known to execute in constant time. The following code might be used to obtain a list of parameterized phonemes representing a single English sentence:

```
phonemes := compile_sentence(s) ! (sleep(75), simple_compile(s))
```

In this code, the system vocalizes the list of phonemes produced by whichever parser terminates first. Because, for example, the output from compile_sentence is preferred over the output from simple_compile, it is given a 75 millisecond head start. Because simple_compile is known to execute in constant time, no additional timing restrictions need be imposed on this expression.

4. Process Manipulation

Several new functions are provided for accessing and manipulating information about processes and internal streams. bound, for example, allows programmers to set an upper limit on the number of messages that may be stored in a stream. bound takes as arguments a stream and an integer limit. Following execution of bound, a process that attempts to place more than the specified number of values into the internal buffer is blocked. Once a message has been viewed by a recipient process, that message no longer occupies a slot in the bounded buffer representing the stream. If backtracking occurs, the message is restored to a special buffer whose size is not affected by bound. If the buffer, at the time that bound is called holds more than the specified number of values, all of these values are retained in the buffer until they are extracted by a reading process. If the limit is set to zero, future message passing requires a rendezvous between the reading and writing processes.

The kill function destroys its process argument. Any processes spawned by the specified process in order to evaluate a concurrent alternation expression are likewise killed. Explicitly created child processes (processes created by the create operator) are not killed. If a killed process owns mutually exclusive read or write access to a stream at the time it is killed, this access is relinquished when the process is killed.

Processes are prioritized from 0, being the highest priority, to 15. When processes are created, they inherit the priority of the process that creates them. &main initially has priority 0. The priority function allows programmers to set the priority of a process. priority takes two arguments: an integer priority and a process. The default process is &current.

The sleep function blocks the current process for the number of milliseconds specified by its single integer argument and returns &null.

The wait function blocks the current process, delaying until the process specified as wait's single argument dies. If the specified process is already dead, wait returns immediately.

Each process has associated with it a special output stream which accumulates the results produced by the process. yield, which expects a single process argument, returns the output stream of that process.

5. Examples

In addition to the examples discussed above, two classical concurrency problems are presented and solved here. These programs demonstrate, among other things, that even though Conicon offers only a small number of simple concurrent processing mechanisms, the language is expressive enough to allow elegant solutions to many traditional concurrent programming problems.

One famous concurrency problem, selected because it represents a variety of real scenarios in which multiple processes share access to a limited number of resources, is the "Dining Philosophers Problem" [3]. In this problem, N philosophers sit at a round table with a bowl of rice in the center of the table. To eat rice, each philosopher needs two chopsticks. However, there are only a total of N chopsticks on the table, so not all of the philosophers can eat at the same time. Each chopstick is placed on the table between a pair of philosophers, and is shared only between the two adjacent philosophers.

A solution to this problem must implement some protocol that allows each philosopher to repeatedly eat, then think. While one philosopher is thinking, other philosophers are given the opportunity to eat. A solution should ensure that every philosopher is given equal opportunity to eat, and it should prevent deadlock from occurring.

Following is a solution to this problem written in Conicon. This solution uses a stream to represent each chopstick. Below is the main procedure:

```
procedure main()
  local i, n, chopsticks

  write("how many philosophers? ")
  n := read()
```

```
  chopsticks := []
  every 1 to n do {
    put(chopsticks, open(,"l"))
    write(chopsticks[-1], &null)
    }

  every i := 1 to n - 1 do
    create philosopher(i, chopsticks[i], chopsticks[i+1])
  wait(create philosopher(n, chopsticks[1], chopsticks[n]))
end
```

The main procedure simply creates one stream for each of the n chopsticks, and sends to each of those streams a single value. Then main creates the n philosopher processes. The philosopher procedure takes as parameters its own identification number, and the streams representing each of the two chopsticks that it must eat with. philosopher's second argument represents the first chopstick that the philosopher must pick up. It is important to specify for each philosopher the order in which it should pick up its two chopsticks. By requiring that the nth process pick its chopsticks up in reversed order from the other processes, the possibility of deadlock is avoided.

```
procedure philosopher(id, firstchopstick, secondchopstick)

  repeat {
    getchopstick(firstchopstick)
    getchopstick(secondchopstick)

    write("philosopher ", id, " is eating\n")

    putchopstick(firstchopstick)
    putchopstick(secondchopstick)

    write("philosopher ", id, " is thinking\n")
    }
end
```

putchopstick and getchopstick are simple stream accessing procedures:

```
procedure getchopstick(queue)
  advance(2, queue)
end

procedure putchopstick(queue)
  write(queue, &null)
end
```

The dining philosophers problem represents situations where multiple processes compete for limited resources. In another large class of concurrent problems, a single server process satisfies the needs of many client processes. These problems are represented by the "Sleeping Barber Analogy" [4]. In this analogy, a small barbershop with a single barber and a single barber's chair serves customers as they arrive. Within the barbershop, a special waiting room is provided where customers generally take short naps, waiting until the barber is ready to cut their hair. If the barber finds his waiting room empty after finishing a hair cut, he likewise sleeps in the waiting room, awaiting arrival of a new customer. A program that models this situation is described below. In this program, the barber and each customer are represented by individual processes.

The barber repeatedly waits for a customer, cuts the customer's hair, opens the exit door for the customer, and waits for the customer to leave. This is implemented by the following procedure:

```
procedure barber()

  repeat {
    # get a customer
    signal(barber_ready)
    wait_til(chair_occupied)

    cut_hair()

    # show the customer out the door
    signal(door_opened)
    wait_til(customer_gone)
    }
end
```

Meanwhile, each client to the barber executes the following procedure:

```
procedure client()

  wait_til(barber_ready)
  signal(chair_occupied)

  # sit quietly while barber cuts hair

  wait_til(door_opened)
  signal(customer_gone)
end
```

In this program, wait_til blocks the current process until the condition supplied as its argument is signaled by some other process. Processes blocked on these conditions are queued in first-come, first-served order. Each time a condition is signaled, only the first process on the queue is wakened. Using streams to represent each of the special conditions, signal and wait_til are implemented as shown below:

```
procedure signal(s)
  write(s, " ")
end

procedure wait_til(s)
  advance(2, s)
end
```

The main procedure shown below initializes shared variables and creates processes to represent the barber and each of the clients.

```
global barber_ready, chair_occupied, door_opened, customer_gone

procedure main()
  local n, clients

  # create condition queues
  barber_ready := open()
  chair_occupied := open()
  door_opened := open()
  customer_gone := open()

  create barber()

  write("how many clients?")
  n := read()
```

```
    # create client processes
    clients := []
    every 1 to n do
      put(clients, create client())

    # wait for client processes to die
    while wait(get(clients))
  end
```

6. Conclusions

Concurrent processes bring new expressive capabilities to Icon. These capabilities extend the application domain of Icon to include real-time problems such as data communication and voice recognition. The unique combination of Icon's high-level goal-directed evaluation mechanisms with real-time concurrent processing capabilities provides new opportunities for investigating the effects of novel language mechanisms on the process of program development.

7. Acknowledgements

The design of these concurrent processing capabilities has benefitted from discussion with members of the Icon research group at the University of Arizona. Members of this group are Ralph Griswold, Dave Gudeman, Janalee O'Bagy, and Ken Walker. Ralph Griswold has also read several drafts of this document, and suggested a variety of improvements.

References

1. K. Nilsen, "A Stream Data Type for Icon", The Univ. of Arizona Tech. Rep. 88-10, Tucson, Arizona, 1988.
2. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
3. R. C. Holt, G. S. Graham, E. D. Lazowska and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, 1978.
4. E. W. Dijkstra, "Cooperating sequential processes", in *Programming Languages*, F. Genuys (ed.), Academic Press, 1968.