A simple shell—I/O redirection

UNIX shells allow standard input and standard output to be *redirected* with the < and > symbols.

Here is a shell command that runs `wc` on the class mailing list mailbox and redirects the output to the file `wc.output`

```
% wc < /home/cs451/mail > wc.output
```

The result:

```
% cat wc.output
   6257   31501  268989
```

IO redirection, continued

For reference:

```
% wc < /home/cs451/mail > wc.output
```

A cornerstone of redirection is that the `exec()` call replaces the current process with the execution of another program, but file descriptors are unaffected. For example, standard input (`&input`) in the original process is standard input in the replacement process.

Here is a program (`redir1`) that takes advantage of this carryover of file descriptors to mimic the shell command above:

```
procedure main()

   infile := open("/home/cs451/mail")
   fdup(infile, &input)  # like &input := infile

   outfile := open("wc.output", "w")
   fdup(outfile, &output)

   exec("wc", "wc")
end
```

Execution:

```
% redir1
% cat wc.output
   6257   31501  268989
```

The `fdup(`*`from`*`, `*`to`*`)` function replaces the file descriptor associated with the file value `to` with the file descriptor associated with the file value `from`.

`ish`: A shell in Unicon

Here is `ish`, a rudimentary shell that provides I/O redirection and background processes (via `&`):

```
procedure main()
   while line := (writes("ish -- "), read()) do {
      if *line = 0 then next
      w := split(line)
      cmd := get(w)

      background := if w[-1] == "&" then { pull(w); 1} else &null

      if fork() = 0 then {
         pgmargs := [ ]
         stdin := stdout := &null
         while arg := get(w) do {
            case arg[1] of {
               "<": stdin := arg[2:0]    # assume no space after '<' and '>'
               ">": stdout:= arg[2:0]
               default: put(pgmargs, arg)
               }
            }
         if \stdin then {
            stdin := open(stdin) | stop(stdin, ": Can't open")
            fdup(stdin, &input)
            }
         if \stdout then {
            stdout := open(stdout, "w") | stop(stdin, ": Can't open")
            fdup(stdout, &output)
            }
         exec!([cmd, cmd] ||| pgmargs)
         }
      else {
         if /background then wait()
         }
      }
end
```

`ish` in operation

```
% ish
ish -- date
Wed Apr 23 23:33:48 MST 2003
ish -- date >out
ish -- cat out
Wed Apr 23 23:33:54 MST 2003
ish -- wc <ish.icn
     38     115     832
ish --
ish -- du /usr >du.out &
ish -- wc du.out
    562    1124   21711 du.out
ish -- who
whm      tty1     Apr  6 23:50
whm      pts/0    Apr  6 23:52 (:0)
ish -- date
Wed Apr 23 23:42:56 MST 2003
ish -- wc du.out
   1644    3288   64077 du.out
ish -- ^D
%
```

Some work remains:

```
ish --
ish -- ls *.icn
ls: *.icn: No such file or directory
ish --
ish -- ls | wc
ls: |: No such file or directory
ls: wc: No such file or directory
```

Pipes

A standard feature of UNIX shells is the ability to send the output from one program into the input of another program.

```
ls | wc
ls -t | grep -v ".icn$" | head -1
```

The supporting mechanism for this is called a *pipe*.

A pipe is an operating system mechanism that arranges for output written to a file descriptor to be available as input on another file descriptor.

Reads from a pipe will block until something is written to the other end. Writes to a pipe will block if a sufficient amount of already written data is still unread.

Unicon's `pipe()` function creates a pipe and returns a list of two file values: the first for reading and the second for writing:

```
][ pipe();
   r := [file(pipe), file(pipe)]  (list)
```

A trivial example:

```
procedure main(args)
    pipes := pipe()
    write(pipes[2], "Testing...")
    write(reverse(read(pipes[1])))
end
```

Output:

```
...gnitseT
```

Pipes, continued

In most cases a pipe is used to send data between two processes.

In the following program a child process writes to a parent process at random intervals.

```
procedure main()
    pipe_pair := pipe()
    if fork() = 0 then
        every i := 1 to 5 do {
            delay(?5000)
            write(pipe_pair[2], i)
            }

    while line := read(pipe_pair[1]) do
        write("My child wrote to me! (",
            line, " at ", &clock, ")")
end
```

Output:

```
My child wrote to me! (1 at 21:14:04)
My child wrote to me! (2 at 21:14:06)
My child wrote to me! (3 at 21:14:08)
My child wrote to me! (4 at 21:14:10)
My child wrote to me! (5 at 21:14:13)
```

Pipes, continued

Consider a program that prompts for two commands and uses a pipe to connect the output of the first to the output of the second: (blank lines have been added...)

```
Pipe from? ls
Pipe to? wc
    118     118     917

Pipe from? ls
Pipe to? grep fork
fork
fork.icn
fork0
fork0.icn

Pipe from? who
Pipe to? wc
     71     355    2201

Pipe from? iota 3
Pipe to? cat
1
2
3

Pipe from? iota 3
Pipe to? tac
3
2
1
```

Pipes, continued

Here is the from/to piper:

```
procedure main()
    repeat {
        writes("Pipe from? ")
        from_cmd := split(read())
        writes("Pipe to? ")
        to_cmd := split(read())

        pipe_pair := pipe()
        if fork() = 0 then {
            fdup(pipe_pair[2], &output)
            close(pipe_pair[1])
            exec!([from_cmd[1]]|||from_cmd)
            write("exec failed! (from)")
            }

        if fork() = 0 then {
            fdup(pipe_pair[1], &input)
            close(pipe_pair[2])
            exec!([to_cmd[1]]|||to_cmd)
            write("exec failed! (to)")
            }

        close(pipe_pair[1])
        close(pipe_pair[2])

        wait()  # wait for both children to
        wait()  # terminate
        }
end
```

Note that the `close()`s are needed to make it work.

How could we add piping to `ish`?

The `select()` function

The `select()` function allows a program to wait on input from any one of several input sources and, optionally, a delay time. It looks like this:

```
select(file1, file2, ..., wait_time)
```

It returns when input is available on at least one of the files and/or the wait time (in milliseconds) has elapsed. The return value is a list of files on which input is available. If the list is empty, the wait time was exceeded.

In what situations is something like `select()` necessary?

`select()` allegedly works on files, network connections, pipes, and windows. The following example required a patch to the Unicon runtime system.

The `select()` function, continued

In this program a parent process forks three children and then waits to hear from each via a pipe.

```
procedure main()
    cpipes :=[] # input side of pipes from children
    every c := !"ABC" do {
        pipe_pair := pipe()
        put(cpipes, pipe_pair[1])
        if fork() = 0 then {
            randomize()
            repeat {
                delay(?15000) # 15 seconds
                write(pipe_pair[2], c)
                }
            }
        }
    while files := select(cpipes[1], cpipes[2],
      cpipes[3], 3500) do { # should use select!...
        if *files ~= 0 then
            every f := !files do {
                line := read(f)
                write(line, " wrote to me at ",
                      &clock)
                }
        else
             write("My kids never write...")
        }
end
```

Output:

C wrote to me at 02:22:58 My kids never write... B wrote to me at 02:23:04 A wrote to me at 02:23:04 My kids never write... C wrote to me at 02:23:08	A wrote to me at 02:23:11 My kids never write... B wrote to me at 02:23:18 C wrote to me at 02:23:19 C wrote to me at 02:23:22