```
call writeread (process^fd, startup^message, 66); -- write startup message
while 1 do
  begin
  read data from terminal
  call writeread (process^fd,
                  data, data^length,              -- write data
                  reply, max^reply,               -- read data back
                  @reply^length);                 -- return real reply length
  if reply^length > 0
    write data back to terminal
  end;
```

The following shows the child process (server):

```
call open (receive, receive^fd);
do
  call read (receive^fd, startup^message, 66);
until startup^message = -1;            -- first word of startup message is -1.
while 1 do
  begin
  call readupdate (receive^fd, message, read^count, count^read);
  process message received, replacing buffer contents
  call reply (message, reply^length);
  end;
```

The first messages that the child receives are system messages: the parent `open` of the child sends an `open` message to the child, and then the first call to `writeread` sends the startup message. The child process handles these messages and replies to them. It can use the open message to keep track of requestors or receive information passed in the last 16 bytes of the file name. Only then does the process receive the normal message traffic from the parent. At this point, other processes can also communicate with the child. Similarly, when a requestor closes the server, the server receives a `close` system message.

Device I/O

It's important to remember that device I/O, including disk file I/O, is handled by I/O processes, so "opening a device" is really opening the I/O process. Still, I/O to devices and files is implemented in a slightly different manner, though the file system procedures are the same. In particular, the typical procedures used to access files are the more conventional `read` and `write`, and normally disk I/O is not no-wait.

Security

In keeping with the time, the T/16 is not an overly secure system. In practice, this hasn't caused any serious problems, but one issue is worth mentioning: the transition from nonprivileged to privileged procedures is based on the position of the procedure entry point in the PEP table and the value of the `priv` bit in the E register. Early on, exploits became apparent. If you could get a privileged procedure to return a value via a pointer and get it to overwrite the saved E register on the stack in such a way that the `priv` bit was set, the process would remain privileged