1. Get a copy of the sample program pipesend.c. Compile and link it under Unix (or Linux) by typing:
   ```
   make pipe < cr >
   ```
   If you have a Windows NT system available you should try compiling it with the Microsoft `nmake` (for "new make"). To do this, you will have had to enabled the command line compiler under Microsoft Visual C++ version 5 or higher. This makes a batch file named `vcvars32.bat` which you will have to run before compilation to set your environment for a (32-bit) MSOS command prompt session. Make a subdirectory for the files, copy over the (sub)directories `.\arpa`, `.\netinet`, and `.\netns`, and do
   ```
   copy makefile.msv Makefile < cr >
   ```
   `nmake pipe < cr >`
   which will make an executable `pipesend.exe` which you can then run.

2. The program `pipesend` is our first example of a program whose structure depends strongly on the design of the operating system. Under Unix (and Linux), it opens a pipe and forks an offspring process. Since the offspring has a full copy of the parent’s data segment and descriptor table, and since the offspring starts at the line after the `fork()` communication through the pipe is straightforward using `read()` and `write()`. But, if you have not seen the `fork()` system call before you should study this code closely. Under Windows NT the parent externalizes the pipe read descriptor and calls the `spawnl` command to create a duplicate offspring process and invoke it as if
   ```
   pipesend.exe -child < pipe read descriptor >
   ```
   were manually typed. The child must parse its command line to
   1. find out that it is the child, and
   2. get the pertinent descriptor information.

   Thereafter, the parent closes the receive end of the pipe and the offspring closes the send end of the pipe. The parent prompts the user for a line of information to send to the offspring (through the pipe), however, line boundaries may not preserved in the transmission. Note that the offspring process blocks in the `read()` until at least one character is available (since `O_NDELAY` is not set) or until there are no processes left writing the pipe, in which case 0 is returned. If more characters are available than requested the read will return the number requested (and subsequent `read(s)` will be needed to empty the pipe). Normally, the processes writing the pipe do not block unless the pipe is full (the system buffers a certain amount of characters but it is limited). Try executing the program.
   ```
   pipesend < cr >
   ```
   (if possible, under both Unix and Windows NT). The drawbacks of using pipes are
   i. they are uni-directional (whereas sockets are bi-directional), so two pipes are needed for bi-directional communication, and
   ii. two processes must share some common ancestor to communicate via a pipe.