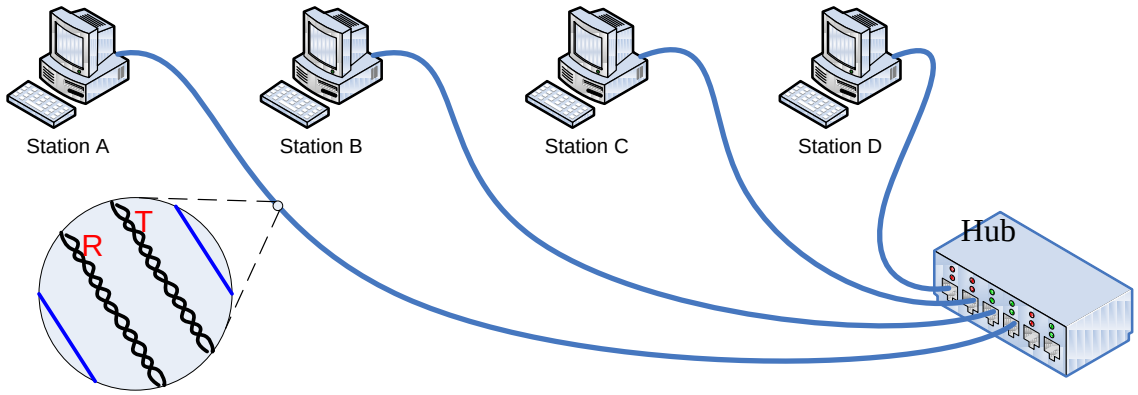
Ehternet/802.3 Simulation Software Design
CSI3131 – Operating Systems

Introduction

The Ethernet/802.3 communications hub is a device to create a LAN using twisted pair wiring as shown in the Figure below. Each station is connected to the hub using two twisted pair wires as shown; one pair (labelled T, call it the *T-pair*) is the **t**ransmission pair that is used by the station to transmit LAN frames (messages) to the hub and the other pair (labelled R, call it the *R-pair*) is the **r**eception pair used by the station to receive LAN frames from the hub.

The hub creates a LAN (where all stations see the transmissions from all other stations) by retransmitting any frame received from a station transmits on its T-pair to all R-pairs of the stations except for the station that transmitted the frame. For example if Station B transmits a frame via its T-pair to the hub, it will retransmit that frame back to the Station A, Station C and Station D via their R-pair.

Transmission frames on such a broadcast network contain two station addresses; a source address that identifies the source station and the destination address that identifies the destination of the frame. Stations ignore frames whose destination address does not correspond to their own.



The objective of this assignment is to create a set of processes, threads and pipes that simulate the above network. Processes shall represent each of the devices, that is, the four stations and the hub. Pipes shall represent twisted pair wires (two pipes per connection between a station and the hub). Threads shall be used within the hub process to monitor each of the "T-pair" pipes for retransmission across the "R-Pair" pipes.

Two programs are developed. The program "*stn*" is run by the station processes while "*hub*" is run by the hub process. The "*stn*" program is run with a configuration file that defines the following: a single character that serves as an address for the station, another character that serves as the destination address for sending messages, and a set of messages to send to the destination. See the section "Background Information" for details on the format of the configuration file.

Two C programs have been provided: `stn.c` and `hub.c`; as well as 4 configuration files to simulate 4 stations. Your task shall be to complete the `hub.c` programs such that the execution of the program `hub` creates the following processes:

1. Station A process: a process that runs the *stn* program using the file *stnA.cfg* to simulate Station A.
2. Station B process: a process that runs the *stn* program using the file *stnB.cfg* to simulate Station B.

3. Station C process: a process that runs the *stn* program using the file *stnC.cfg* to simulate Station C.
4. Station D process: a process that runs the *stn* program using the file *stnD.cfg* to simulate Station D.
5. Hub process: This process mimics the network hub and will retransmit any frames transmitted by a Station to all other stations. This program also sets up the network to create the station processes, threads and pipes to setup the network. Your task for the assignment is to provide the code to do the network setup.

Station Software

The station software is complete and found in the C module stored in the file `stn.c` and functions as follows:

- Transmit each of the messages in the configuration file to the destination station (also identified in the configuration file). The station waits for an acknowledgement (special message) back from the destination between each message sent.
- When it receives a message from another station, it responds by transmitting back an acknowledgement message. It uses the source identifier (address) in the received frame to create the acknowledge frame.
- Ignore any frame (and message) that contains a destination address that does not correspond to its own station identifier (address).
- A frame consists of a character string as show below. All characters are printable characters (STX and ETX are normally non-printable characters) to facilitate debugging. You will be able to display frames seen by the hub. The stations look for the STX before parsing a frame. If it does not find an STX it will ignore all subsequent characters until it finds the ETX or the end of the characters received. When such errors occur, a message shall be displayed.

STX	D	S	<message>	ETX

  where STX is the character '@', ETX is the character '~', D is the character that gives the identifier (address) of the destination station (one of A, B, C, or D), and S is the identifier (address) of the source of the message/frame (one of A, B, C, or D), and <message> is a message string that contain neither the STX nor the ETX characters.

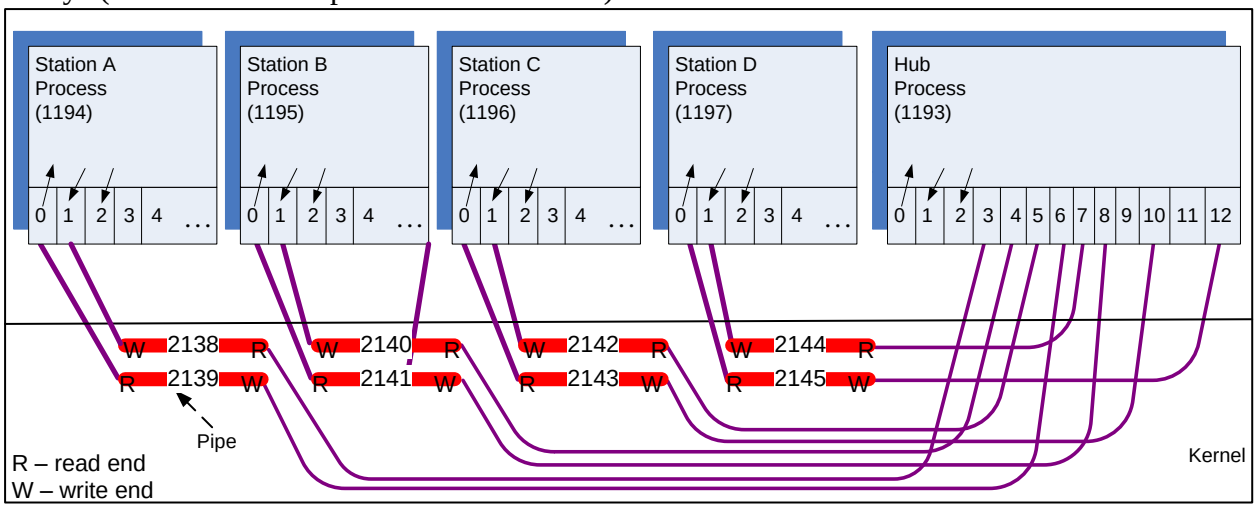
The functions provided in the module are:

- `main`: The function main provides the overall control of the program. It extracts the name of the configuration file from the command line, reads the content of the file which is used to initialise the pointer array `messages`, and the message buffer `msgsBuffer`. The function `communication` is then called to transmit messages to the destination station and process received messages destined for the station.
- `readFile` – used to parse the configuration file.
- `communication` – This function controls the station communications. It transmits the messages read from the configuration file and displays messages received from other stations. This function returns when the write end of the R-pair pipe is closed by the hub process.
- `readMsg` – This function is called by `communication` to obtain the message of the next frame received over the network. It is responsible for reading from the R-pair pipe and can block when this pipe is empty. All data is read from the pipe (note that multiple frames can be read at the same time) and stored in the buffer, `allFrames`. The function `extractMsg` is used to read messages one at a time from this buffer.
- `extractMsg` – This function is used to extract frame information (addresses and message content) from a buffer containing one or more frames.

Hub Software

The hub software consists of a single module which has two main roles. The first role is to create the station processes and the pipes used to interconnect the stations processes with the hub process (running the hub software).

Interconnection of the processes with pipes is shown in the diagram below. The standard input (fd 0) and standard output (fd 1) are connected to the pipes representing the R-pair and T-pair connections respectively for the stations. The hub maintains a set of file descriptors of the other ends of the pipes in two arrays (see details in the provided source code).



The pipe identifiers and process identifiers (PIDS) shown in the above diagram are specific to a run and correspond to the identifiers shown in the output of item 4 in the section "Background Information". All standard error file descriptors (2) the Station processes are connected to the terminal.

The file descriptor arrays, `fdsRec` and `fdsTran`, are initialised during the creation of the station processes and pipes with the file descriptors to the pipe ends managed by the hub process. The same index into both arrays associates the T-pair pipe and R-pair pipe of one station. For example at index 0, the file descriptor of the T-pair pipe of Station A and the file descriptor of the R-pair pipe of Station A are stored respectively into `fdsTran` and `fdsRec`.

The second role of the module is to monitor frames transmitted across the T-pair pipes of all stations. For each station process, a thread is created to monitor its T-pair pipe. Any frame received across the pipe is copied to all R-pair pipes of the other stations.

The functions used in the hub module are summarized below.

- `main` – Calls createStation four times to create each of the four stations. The function createStation adds file descriptors to the arrays `fdsRec` and `fdsTran`. It then shifts all entries in the `fdsRec` array by 1 position (first entry is moved to the last entry). Then it calls hubThreads that creates the hub threads.
- `createStation` – Creates a station process (running the stn program). It appends the file descriptors to be monitored at the end of the `fdsTran` and `fdsRec` arrays.
- `hubThreads` – For each entry in the fdsTran array, this function creates a thread to monitor the pipe. Thus four threads are created, one for each station to monitor its T-pair pipe. After 30 seconds, the threads are cancelled.
- `listenTran` – this function is run by the created threads. It will copy any data seen on the T-pair pipe from a station to all R-pair pipes of the other stations.

Annex A – Background

1. An open file descriptor is an integer, used as a handle to identify an open file. Such descriptors are used in library functions or system calls such as `read()` and `write()` to perform I/O operations.
2. In Unix/Linux, each process has by default three open file descriptors:
    a. Standard input (file descriptor 0, i.e. read(0,buf, 4) reads 4 characters from the standard input to the buffer buf). Typically, the standard input for a program launched from the command line is the keyboard input.
    b. Standard output (file descriptor 1).
    c. Standard error (file descriptor 2).
    d. When a command is run from the shell, the standard input, standard output and standard error are connected to the shell tty (terminal). So reading the standard input reads from the keyboard and writing to the standard output or standard error writes to the display.
    e. Note that many library functions used these file descriptors by default. For example *printf("String")* writes "String" to the standard output.
    f. From the shell it is possible to connect the standard output from one process to the standard input of another process using the pipe character "|". For example, the command "who | wc" connects the standard output from the who process to the standard input of the wc process such that any data written to the standard output by the who process is written (via a pipe) to the standard input of the wc process.
3. You will need the following C library functions:
    g. fork() – should be familiar from lectures
    h. pthread_create() – should be familiar from lectures
    i. pthread_cancel(), pthread_testcancel() – for cancelling a thread (termination by another thread).
    j. pthread_join() – waits for a thread to terminate
    k. pthread_exit() – exits from a thread
    l. pipe()
        - should be familiar from lectures
        - note that multiple process can be attached to each end of the pipes, which means that a pipe is maintained until no processes are connected at either end of the pipe
    m. execvp(const char * program, const char *args[]) (or execlp)
        - replaces the current process with the program from the file specified in the first argument
        - the second argument is a NULL terminated array of strings representing the command line arguments
        - by convention, args[0] is the file name of the file to be executed
    n. execlp(const char * program, const char *arg1, const char *arg2, … NULL)
        - replaces the current process with the program from the file specified in the first argument
        - the subsequent arguments are strings representing the command line arguments.
        - by convention, arg1 is the file name of the file to be executed

o. dup2(int newfd, int oldfd) – duplicates the oldfd by the newfd and closes the oldfd. See http://mkssoftware.com/docs/man3/dup2.3.asp for more information. For example, the following program:

```
int main(int argc, char *argv[]) {
      int fd;
      printf("Hello, world!")
      fd = open("outFile.dat", "w");
      if (fd != -1) dup2(fd, 1);
      printf("Hello, world!");
                    close(fd);
}
```

will redirect the standard output of the program into the file outFile.dat, i.e. the first "Hello, world!" will go into the console, the second into the file "outFile.dat".

p. read(int fd, char *buff, int bufSize) – reads from the file (or pipe) identified by the file descriptor *fd* a number of bytes into the memory buffer whose address is passed in the argument *buff*.. The parameter *bufSize* defines the maximum number of bytes to read. For example, if a pipe (referenced by *fd*) only contains 50, and *bufSize* is set to 100, all 50 bytes will be read. On the other hand, if the pipe contains 200 bytes and *bufSize* is set to 100, then only the first 100 bytes are read from the pipe. The function returns the number of bytes read, or -1 on error or 0 if the end of file has been reached (or the write end of the pipe has been closed and all data read).

q. write(int fd, char *buff, int bufSize) – writes into the file/pipe referenced by the file descriptor *fd* the a number of bytes found in the buffer whose address is given in the parameter *buff*. The parameter *bufSize* defines the number of bytes to write. The function returns the number of bytes actually written or -1 on error.

r. close(int fd) – closes an open file descriptor

s. printf(): You may want to use the printf() function to format output. This function writes to the standard output (fd 1). But be careful since this function buffers output and does not write immediately to the standard output. To force an immediate write, use fflush(stdout). Alternatively, you may used sprintf(), to format the output into a buffer and use write() to write to the standard output.

t. fprintf(): this is a version of *printf()* that provides the means to specify where output should be send. Use it to write to the standard error with *fprintf(stderr,"a message", arg, arg,…)*. This function is useful for debugging as it will write to the terminal in processes where the standard output has been redirected to a pipe.

u. getpid(): this function returns the PID of the current process. It is useful in creating messages printed on the screen to identify the source of the message.

v. Consult the manual pages (by typing 'man function_name', i.e. 'man fork') and/or web resources for more information.

4. Here is a hint at how you can observe processes, threads and pipes
    w. Insert long delays using the standard library function `sleep` (e.g. `sleep(300)`) to allow observation of processes, threads and pipes at different points in the execution of the `hub.c` program during the creation and termination of processes and threads.
    x. To see the processes and threads created use the command "`ps -Hmu test1`". The option H has `ps` print out a tree of processes/threads and the option m includes all threads. In fact, Linux treats all processes and threads as tasks assigning each a PID. See below for expected output.

```
[test1@sitedev proc]$ ps -Hmu test1
  PID TTY          TIME CMD
 1117 ?        00:00:00 sshd
 1118 pts/0    00:00:00   bash
 1193 pts/0    00:00:00     hub
 1194 pts/0    00:00:00       stn
 1195 pts/0    00:00:00       stn
 1196 pts/0    00:00:00       stn
 1197 pts/0    00:00:00       stn
 1198 pts/0    00:00:00       hub
 1199 pts/0    00:00:00       hub
 1200 pts/0    00:00:00       hub
 1201 pts/0    00:00:00       hub
 1159 pts/1    00:00:00   bash
 1202 pts/1    00:00:00     ps
```

Hub process (pid 1193) that forks the four station processes and the four threads

Four Station processes (pids 1194, 1195, 1196, 1197)

1198, 1199, 1200, and 1201 are threads created by 1193 (using pthreads)

See the next page for a hint on observing file descriptors and pipes.

y. To see how pipes and standard input, standard output, and standard error are set up for the various processes, use the command "`ls -l /proc/xxx/fd`" where `xxx` is replaced with the PID of a process. This will display how the various file descriptors of the identified process are connected. See below for expected output from the programs of this assignment.

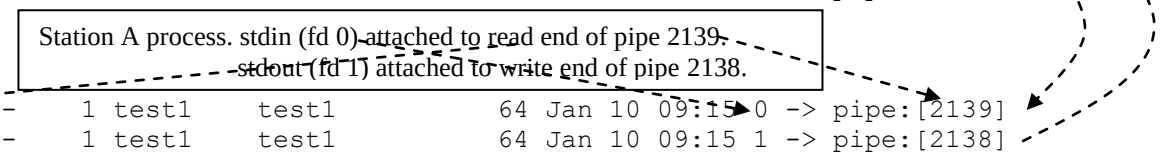
```
[test1@sitedev proc]$ ls -l 1193/fd 1194/fd 1195/fd 1196/fd 1197/fd
1193/fd:
total 11
lrwx------    1 test1    test1          64 Jan 10 09:15 0 -> /dev/pts/0
lrwx------    1 test1    test1          64 Jan 10 09:15 1 -> /dev/pts/0
l-wx------    1 test1    test1          64 Jan 10 09:15 10 -> pipe:[2143]
l-wx------    1 test1    test1          64 Jan 10 09:15 12 -> pipe:[2145]
lrwx------    1 test1    test1          64 Jan 10 09:15 2 -> /dev/pts/0
lr-x------    1 test1    test1          64 Jan 10 09:15 3 -> pipe:[2138]
lr-x------    1 test1    test1          64 Jan 10 09:15 4 -> pipe:[2140]
lr-x------    1 test1    test1          64 Jan 10 09:15 5 -> pipe:[2142]
l-wx------    1 test1    test1          64 Jan 10 09:15 6 -> pipe:[2139]
lr-x------    1 test1    test1          64 Jan 10 09:15 7 -> pipe:[2144]
l-wx------    1 test1    test1          64 Jan 10 09:15 8 -> pipe:[2141]

1194/fd:
total 3
lr-x------    1 test1    test1          64 Jan 10 09:15 0 -> pipe:[2139]
l-wx------    1 test1    test1          64 Jan 10 09:15 1 -> pipe:[2138]
lrwx------    1 test1    test1          64 Jan 10 09:15 2 -> /dev/pts/0

1195/fd:
total 3
lr-x------    1 test1    test1          64 Jan 10 09:15 0 -> pipe:[2141]
l-wx------    1 test1    test1          64 Jan 10 09:15 1 -> pipe:[2140]
lrwx------    1 test1    test1          64 Jan 10 09:15 2 -> /dev/pts/0

1196/fd:
total 3
lr-x------    1 test1    test1          64 Jan 10 09:15 0 -> pipe:[2143]
l-wx------    1 test1    test1          64 Jan 10 09:15 1 -> pipe:[2142]
lrwx------    1 test1    test1          64 Jan 10 09:15 2 -> /dev/pts/0

1197/fd:
total 3
lr-x------    1 test1    test1          64 Jan 10 09:15 0 -> pipe:[2145]
l-wx------    1 test1    test1          64 Jan 10 09:15 1 -> pipe:[2144]
lrwx------    1 test1    test1          64 Jan 10 09:15 2 -> /dev/pts/0
```



The setup shown for Station A process is repeated for each of the other three station processes. Notice how a pipe is used to represent an R-pair and a second to represent a T-pair between the hub process and a station process. Each station process uses its standard input and standard output to communicate with the hub process, and still can print to the display using the standard error (file descriptor 2) attached to `/dev/pts/0` that corresponds to a terminal (actually a pseudo-terminal connected to an ssh client).

5. Configuration file format:
    z. The configuration file for the *stn* program contains a set of lines that are processed when the program is started.
    aa. All lines that start with the character "#" are ignore as are empty lines (careful about having spaces in a line – this will not be ignored). Such lines can be used for commenting the configuration file.
    bb. The first line with data is used to set the station identifier (address). Only the first character on the line is used (again do not start with a space character) – so the rest of the line can be used for documentation as it will be ignored.
    cc. The second line with data is used to set the destination identifier (address). Only the first character on the line is used (again do not start with a space character) – so the rest of the line can be used for documentation as it will be ignored.
    dd. All other lines with data are treated as separated messages to be sent to the destination.
    ee. The following is an example configuration file that sets the station identifier to A and has the station transmit three messages to station C.

```
# Messages sent from station A to station C

A  - station identifier
C  - destination identifier

# the messages are:
Hello station C
Received your acknowledgement to first message
Have a good day - see you next time
```