CSE 380: Extra Credit: Distributed Leader Election

Due : Wednesday, December 10, 2003

Submit the code, executable, and documentation using the `turnin` command to the account `cse380@eniac` by 11.59pm on Dec 10. **Reminder:** Copying the solution from a fellow student, a website, or some other source, is a violation of the University policies on academic integrity. On *eniac*, you can submit the files by using the command:

```
turnin -c cse380 [list of files to turnin]
```

This assignment requires you to implement a leader election protocol in the context of Unix processes talking to each other over TCP sockets. You have to write a program `leader` that takes 3 integers as arguments. First is to be used as the unique identifier to be used for leader election, the second is the port number of the socket that the process should receive messages from, and the third is the port number of the socket that the process should be sending messages to. So an invocation will be `leader id ls rs`. Multiple copies of this program are started to form a communication ring. For example, the following should form a ring of 3 processes:

```
leader 45 54325 23330 &
leader 21 23330 10010 &
leader 86 10010 54325 &
```

The program `leader` should do the following steps:

- Open a server socket, bind the port number `ls` to this socket, and get ready to listen to messages over this socket. This would require a sequence of `socket`, `bind`, and `listen` calls.
- Open a client socket, and connect it to the port `rs`. This would require a sequence of `socket` and `connect` calls.
- Messages should be sent over client socket using `send`, and received from the server socket using `recv`. The goal of the message exchange is to elect a unique leader.

  A simple ring based election algorithm that you can use is given here:

  > In this algorithm, each processor sends a message with its identifier to its right neighbor, and then waits for messages from its left neighbor. When it receives such a message, it checks the identifier in this message. If the identifier is greater than its own identifier, it forwards the message to the right; otherwise, it 'swallows' the message and does not forward it. If a processor receives a message with its own identifier, it declares itself a leader by sending a termination message to its right neighbor, and terminating as a leader. A processor that receives a termination message forwards it to the right, and terminates as a non-leader.

  You can experiment with your own protocol as long as it utilizes the ring structure.

  Whenever the program decides on the id of the leader, it should print a message mentioning the id. The protocol should satisfy two properties: eventually all processes should print such an election message, and there should be no disagreement about who the leader is.
- Once the leader is decided, the program can close the connections and terminate.

To learn about socket programming, consult a network programming book, or Unix man pages. There is a wealth of information available online. For example, see
`http://www.ecst.csuchico.edu/~beej/guide/` You should focus on the network programming portion of the site, not the IPC portion.

The course site also has a demo program `socktest.c` which shows how to open server and client connections and exchange messages.

Here is what you should submit:

1. A file `leader.c`. Include a `README` file if compilation requires unusual flags. It is important that this file compiles on `eniac` without errors.

2. The code should include documentation including an argument about why this implements the leader election correctly.

3. The program should print a statement whenever an interesting event happens. Interesting events include opening and closing connections, sending and receiving of messages, and deciding on the leader. Each such statement should include the id of the current instance. For example, if we run the script for the 3 process ring from last page, a possible sequence of statements printed could be

```
I am 45 : Server connection established for port 54325
I am 21 : Server connection established for port 23330
I am 86 : Server connection established for port 10010
I am 45 : Client connection established for port 23330
I am 86 : Client connection established for port 54325
I am 21 : Client connection established for port 10010
I am 45 : Sending message 45
I am 21 : Sending message 21
I am 86 : Sending message 86
I am 45 : Receiving message 86
I am 86 : Receiving message 21
I am 21 : Receiving message 45
I am 45 : Sending message 86
I am 21 : Receiving message 86
I am 21 : Sending message 86
I am 86 : Receiving message 86
I am 86 : Leader id is 86
I am 86 : Sending leader message 86
I am 45 : Receiving leader message 86
I am 45 : Leader id is 86
I am 45 : Sending leader message 86
I am 21 : Receiving leader message 86
I am 21 : Leader id is 86
```

   You should use the exact syntax above for your output. You may include other messages in your output as well.

Finally, it is worth noting that we are using sockets for one-way FIFO communication between a single sender (client) and a single receiver (server), but sockets actually allow more powerful communication (e.g. multiple clients). Second, the ports are global across the system. That is, in the above example, if some other process is already binding its socket to the port 54325, `bind()` will return an error. Since there are 64000 ports (of which some are already used by the kernel), if you pick port numbers randomly, or try a couple of times, things should work. If not, devise your solution (e.g. you can write a parent process that opens sockets, acquires port numbers for them, and then spawns off different copies of `leader` as child processes). Also, note that the hostname for identifying ports may need to include `red` or `blue` system on `eniac`.