Inter-process communication via sockets

Introduction
This is an introductory assignment in the practicum Codesign. It is also a module within the operation system course. Find available libraries, framework and demos in the course notes for the operating system practicum (in Dutch language).
After this assignment you will be familiar with:

1. sockets
2. client-server model
3. daemons

Preparation
See the book *O.S. Concepts*of Silberschatz (Edition 6):
§4.6.1
See the book Linux A-Z:

1. chapters17-18
2. chapters 22-23

Sockets

Sockets share a lot with pipes. Also sockets behave as files. E.g., you can read and write to a socket. The main difference, however, is, that an unnamed pipe (denoted by a |) only exists between two processes, created by a fork. Necessarily, such processes exist on the same systems. For a socket the processes do not have been created from each other by a fork. As long as there is network connection processes on different machines can communicate on different ways via sockets.

Sockets ? first introduced in BSD-UNIX ? form a general inter-process communication (IPC) mechanism that can be based on top of different network protocols. We will make use of the common TCP/IP protocol for a network connected to Internet. Each machine in the network has an Internet-address, e.g., within the cluster Informatica according to the pattern 130.89.xx.xx. Instead of address numbers is common to use host names or domain names according Internet agreements.

Under Linux the file */etc/hosts* contains a list of IP-numbers with related domain names and possibly *alias* host names. See this file and find out what internet number and hostname your own machine has. See also the file */etc/resolv.conf*. This file contains a list of the names of the *name-servers*. Name-servers are used to find IP-addresses for the domains of host names. The file can contain a local domain name as search area (in our case *cs.utwente.nl*) such that a host name alone, e.g., *xyz*, is sufficient to identify a machine met domain name *xyz.cs.utwente.nl*.

Different socket-connections from and to a single machine are distinguished by the machine identification as well as by a *port number*. There exist agreements for standard port numbers for certain Internet-services (see file */etc/services*). Client programs trying to reach a certain server on a remote host machine will give a specific port number. E.g., a *telnet*-connection as standard goes via the port number 23 and a *mail*-connection via port 25. When calling *telnet* a different port number can be given. Technically it is possible, e.g., to send mail using *telnet*via port 25.

In the following assignment we will implement a *bulletin-server*. The task of such a *bulletin-server* is to store text messages of client processes in a *bulletin-file*. Each time a client makes a socket connection to the *bulletin-server* the client sees the complete *bulletin-file*. Moreover, a textual message can be added to the *bulletin-file*. Via a *bulletin-server*client processes can see each others textual messages.

As client processes all over the world can make a socket connection via Internet, the *bulletin-server* could provide a world wide service. Of course, this depends on the accessibility of the server machine. The satellite machine is connected to a local network and not accessible via the practicum network. The *bulletin-server*should run by preference on the practicum host machine. From the satellite system a connection can be established with a client.

In the assignment we also will see how the *bulletin-server* as *daemon*(a permanently available system process) can be installed in the system.

We will use a simple socket-library (*sockLib.h*, *sockLib.c*) for TCP/IP connections.

This library contains functions with the following prototypes:

```
sockStruct *CreateSocket(void);
int ReturnSocket(sockStruct *handle);
int ServerConnectionToClient(sockStruct *handle,
int portNumber);
int ClientConnectionToServer(sockStruct *,
char *machine,
int portNumber);
```

A socket connection is established as a consequence of a "rendez-vous" between a call of *ServerConnectionToClient* and *ClientConnectionToServer*, where a socket-descriptor is created. See the socket-library code. The following system calls are used:

1. *socket*(in CreateSocket): to create a socket of the correct type
2. *bind*(in ServerConnectionToClient) to bind a socket in the server to an IP-address and port number.
3. *listen*(in ServerConnectionToClient) to open the socket in the server for possible clients that want to connect.
4. *accept* (in ServerConnectionToClient) to wait in de server for a client that wants to connect. With *accept* the *rendez-vous*is completed.
5. *connect* (in ClientConnectionToServer) to make a connection from the client.

See the manual pages for more details on these system calls. By the use of *accept* the server blocks until a client comes up. It is also possible to set a *non-blocking-flag*, such that a connection can be made only if a client comes up.

Using the socket-library functions a *server* program and a *client*program can be made with the functionality in mind. These programs are called as follows: First with:
server porta server-program is started that via *port*builds connections and provides service. The server-program continuously accepts new client-connections and works in principle for unbounded time. It must be ended by a kill command (e.g. Ctrl-C or the kill-command).

1. Then, with:

client *machinename port*
One or more client processes are started. Via a client process text can be typed in, that is sent to the server in order to be added to the *bulletin-file.* A client process is finished by ending the input with Ctrl-D). In this way, client processes can be active on any machine at any moment. Also *telnet* (started with telnet *machinename port*) must be available for use as client process.

Above, *machinename* means the name of the machine, where the server is started. In most cases this is the machine where the work is done (*localhost*), but it also may be any (practicum) machine in the network

On the system demo versions of *clientDemo* and *serverDemo* are available showing how it works. There are frameworks available (*sock*C*lient.c* and *sockServer.c*) that can serve as basis for the implementation. In the server-implementation for each socket-connection established a child process is split off that is the service process for the client process. The frameworks contain hints for what has to be added at which place. The intention is that the service process forked in the server first sends the content of the *bulletin-file* to the client via the socket connection, and then receives the text that has to be added via the socket. In principle, a socket is a *full-duplex* connection, i.e. it provides the possibility to communicate in two directions. By the system call *shutdown* (see *man 2 shutdown*) one of the communication partners can make the socket *half-duplex* by finishing reading from a socket or writing to a socket. The (partial) finishing a socket by *shutdown*can be important for the correct ending of the communication process. If a write process is finished, but the socket remains open for writing, a read process gets stuck. Therefore, take care that a (partial) socket shutdown is performed, if the bulletin file has been sent by the server, or the text message has been sent by the client, if this is necessary for the correct ending of the process that can read from the socket.

It is the best strategy to start with the client program and test it using the demo server.

Bulletin client

1. Fill in the framework given for *sockClient.c*. For information on i/o-functions *(f)open*, *read*, *write* and *(f)close* see the manual pages. The client must receive the *bulletin-file* and send the txt message. It can make the implementation easier to parallelize these client activities, in place of executing them sequentially. As a consequence the client is split into a read and a write process by *fork*.
2. Translate the program using the make mechanism.
3. Test the bulletin client in combination with the program *serverDemo*. The effect of the client program must be similar effect of the c*lientDemo*.
4. Care for a well working client program before you continue. Follow the check list below before you show it to the student assistants.

0.0.1 Checklist for bulletin client

1. Assume, the bulletin-client performs reading and writing in *parallel*using an extra forked process. Explain why in this case it is not necessary to know when reading from the socket has ended before writing starts. How do we care that such an extra read process finishes properly?
2. Control the correct finishing of the processes via the socket communication.

Bulletin-server

1. Complete the given framework of *sockServer.c*.
2. Test the socket connection between your own *sockClient* and *sockServer*within the same system. Do this using different virtual consoles or x-terminals. The effect of the client/server-programs must be similar to the ones of the demo-versions. Make sure that a socket-connection terminates correctly and no client or server processes get stuck on the socket.
3. Test the *bulletin-server*on your system by making contact from the satellite system with the server using a client process.
4. Show this assignment together with the bulletin daemon.

The goal of the following assignment is to let the *bulletin-server* take the role of a *daemon* in the system. A daemon is a system process running in the background independent of a user, i.e. even when the user logs out it continues. A daemon is not linked to a terminal, where error messages can be sent to. Moreover, a daemon has to be protected against signals that could stop the process by accident. The most important steps to make a daemon from a *bulletin-server*are:

By a twofold fork a grandchild process is generated that works as server process independent from its start environment. The starting parent process and the first child process are terminated. Error messages have to be avoided, and have to be written to a file otherwise.

Bulletin daemon

1. See the program *sockDaemon.c*. This program is the daemon-version of the framework program *sockServer.c*. In this version we assume a fixed port number for the socket connections.
2. Adapt the framework *sockDaemon.c* similar to the earlier assignment, such that the daemon functions as *bulletin-server*. Take care that all test and/or error messages are written to a file.
3. Test the *bulletin-server-daemon*using client processes as in the previous assignment.
4. See the presence and the status of the daemon using the command *ps -x*. Check that the daemon is still present after logout. The daemon can be killed with the command *killall*followed by the name of the daemon program.
5. Show the assignments to a student assistant.

0.0.2 Checklist for bulletin server and daemon

1. Try to communicate via *telnet*with the bulletin-server.
2. Did you test the *bulletin-server* and/or *bulletin-daemon*on your system with a client-process from the satellite system?
3. Think of the reasons why in the bulletin-daemon contains no *printf*-statements and all error messages are written to a logfile.
4. Before terminating yourself clean up the bulletin files if not saved in your own .home directory.