Computer Science 315
Laboratory 4 - Working with Sockets
Due: 11:00pm Monday February 27, 2023

In our last two labs, we saw how related processes (a parent and a child) could communicate using a pipe. Pipes are actually implemented using a pair of sockets (one at each end of the pipe). Sockets can be used to provide communication between two unrelated processes, as well. They can even be used for communication between processes on different machines.

When a socket is allocated in a process, it can be assigned a slot in the file descriptor table (just as we did with pipes). After it has been connected to another socket belonging to a different process, the sending process can use file commands to write to the receiving process (which, in turn, uses file commands to read whatever message is sent).

The connection between two sockets can be:

- on the same machine, in which case it is very similar to a pipe (reads and writes to a file descriptor)
- over a network, in which case the messages must be enclosed in a "packet" according to a network protocol (eg. TCP -- Transmission Control Protocol). Network protocols belong to "families" of protocols. They are listed in `/usr/include/sys/socket.h` (search for *Address families* in this file, if you want to see them listed).

There are also several different types of sockets:

- `SOCK_STREAM` -- byte-by-byte transmission of data (like pipes)
- `SOCK_DGRAM` -- datagrams, which are packets. These are not guaranteed to arrive at the other socket, and the order they arrive is indeterminate -- so the receiving process must sort out the packets and ask for retransmission of any that are missing! Processes that don't care about receiving all of the packets won't request retransmission. Can you think of such an application?
- `SOCK_RAW` -- even more work for the processes, since they must specify the path to use in a network.

There are some other types, as well. In this lab, we will implement the simplest set of sockets: `SOCK_STREAM` type sockets, using a protocol belonging to the `AF_UNIX` family (two processes on the same machine).

You will create two programs -- one to receive messages, and one to send messages.

Here's the `receive` program:

```
/* receive.c -- socket demo program, which acts as a message server */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <unistd.h>
#define TRUE 1

struct sockaddr saddr;
int s, rfd, ch;
FILE *rf;
char username[15];
char remString[50];

int main() {
  umask(0);
  strcpy(username, getenv("LOGNAME"));
  strcpy(saddr.sa_data, "/tmp/#");
  strcat(saddr.sa_data, username);

  sprintf(remString,"rm '/tmp/#%s' 2> /dev/null",username);
  system(remString);

  unlink(saddr.sa_data);
  saddr.sa_family = AF_UNIX;
  s = socket(AF_UNIX, SOCK_STREAM, 0);
  bind(s, &saddr, sizeof(saddr));
  listen(s, 1);
  while (TRUE) {
    rfd = accept(s, NULL, NULL);
    rf = fdopen(rfd, "r");
    while ((ch = getc(rf)) != EOF)
      putchar(ch);
    fclose(rf);
    fflush(stdout);
  }
  return 0;
}
```

Obviously, there's a lot of detail in this program.

- The socket requires a data structure (`struct sockaddr`) to identify its protocol family (`sa_family`), and give the socket a name (`sa_data`).
- The `umask(0)` call is to make sure that any file system items created by the process don't block any permissions. A file system item is created for the socket (specifically, `/tmp/#`*theUsersUsername*). Recall for COSC 109 that

the `umask` command is used to set the permissions that should be blocked by default.

- The purpose of the next three lines is to create the string that will be the directory and name of the file to represent to the socket. This file name is being stored in the `sa_data` attribute of the `saddr` structure.
- The following two lines (`sprintf()` and `system()` statements) remove that file from the file system before it is attempted to be made in the `bind()` function that follows a few lines below.
- A call of `unlink` is used to remove any previous instances of the same socket (`unlink` is the system call behind the `rm` program).
- The socket is created by the call of `socket()` .
- It is then *bound* to its name by the call of `bind`. This socket needs a name so that other sockets can identify it when requesting a connection.
- The call of `listen` states that the socket will be used to listen for messages.
- The accept function extracts the first connection on the queue of pending connections, and creates a new socket with the same socket type and address family as the specified socket (`s`, in this case). The `accept` call *blocks* until a message comes in from another process. It returns a file descriptor for the new socket that is created. Remember that a socket is one end of a connection.
- We then create a file handle (`rf`) so we can deal with the message using standard file operations (`getc`, `fclose`).
- The `putchar` call in the loop echoes whatever character is read by the `getc` call. The reading loop terminates when an `EOF` character is read (that's `CTRL-d`).
- The program then waits for another process to call.

To compile this program, we use the following command:

```
      gcc receive.c -o receive

On some unix versions may need the socets library so would use

      gcc receive.c -lsocket -o receive
```

which creates an executable program called `receive`, using the `socket` library, which is default.

And here's the `send` program:

```
/* send.c -- socket demo which acts as a message client */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
struct sockaddr saddr;
int s, ch;
FILE *sf;

int main(int argc, char *args[]) {
   char username[15];
   if (argc != 2) {
      fprintf(stderr, "Usage: %s recipient\n", args[0]);
      exit(1);
   }
   strcpy(username, getenv("LOGNAME"));
   strcpy(saddr.sa_data, "/tmp/#");
   strcat(saddr.sa_data, args[1]);
   s = socket(AF_UNIX, SOCK_STREAM, 0);
   saddr.sa_family = AF_UNIX;
   connect(s, &saddr, sizeof(saddr));
   sf = fdopen(s, "w");
   fprintf(sf, "Data from %s:\n", username);
   while ((ch = getchar()) != EOF)
      putc(ch, sf);
   fclose(sf);
   return 0;
}
```

This program is similar, except it expects an argument when it is used. It identifies the socket being connected with using a `struct sockaddr`, and initiates the communication with a call of `connect`. It then creates a file handle (`sf`) so that file commands can be used to send data (`fprintf`, `putc`, `fclose`). The call of `getchar` reads a character from the keyboard. The `while` loop terminates when the user enters the `CTRL-d` character. Then the process terminates, after closing its socket.

To compile this program:

```
      gcc send.c -o send
```

```
some installations of Linux may not include socket library in which case use
```

```
      gcc send.c -lsocket -o send
```

For more information on sockets see `man socket`. Basically, what we've got here is a cheap, one-machine, one-way `talk` program. A real `talk` program would require threads -- the program would have one thread to listen for messages, a second thread to send messages, and (usually) a third thread to schedule the first two. We will be looking at threads in later laboratories.

Experiments:

Below are a number of experiments and questions. Place your answers to items 3, 4, 5 & 6 in a file called `lab4.txt`.

1. Enter and compile the two programs, `receive` and `send`.
2. Open two terminal windows. Then, you run your `receive` program in one window and `send` program, in another window as follows:

   First:
   `$ receive`
   Second:
   `$ send` *`yourusername`*

   You should send a message (any number of lines), and terminate their message with `CTRL-d` (which means "end-of-file" in UNIX). Then send another message. When you've tried this out, kill your `receive` program using `CTRL-c`.

3. While your `receive program` is running, examine the contents of the directory `/tmp` (use `ls -l |more`) . Do you see anything familiar?

   - How are sockets distinguishable from regular files in a file listing? (1 mark)
   - What is the name of the socket? (1 mark)
   - What happens to the socket when you terminate the `receive` program? (1 mark)

   When you are finished experimenting, delete the socket.
4. What does the *getenv()* function do? This function is used with *`"LOGNAME"`* as an argument in `receive.c`. What other string could be passed in to accomplish the same result for any? (1 mark)
5. This question has three parts. To answer this question, examine the contents of the `socket.h` file that is included. The location of `#include` files on our system is `/usr/include/`. Linux maps `socket.h` to at `/usr/include/x86_64-linux-gnu/sys/socket.h`. This includes several other headers, including `/usr/include/x86_64-linux-gnu/bits/socket.h` and `/usr/include/x86_64-linux-gnu/bits/socket_type.h` . Examine these headers to answer th

   a. What are the values associated with `SOCK_STREAM` and `SOCK_DGRAM`? (1 mark)

   b. Find the name of another type of socket (other than those mentioned previously in this lab). (1 mark)

   c. Find the names of two other network protocol/address families (other than those mentioned previously in this lab). (2 marks)

6. Sockets in Java are much easier to understand and use. Slightly modified versions of the `DateServer.java` and `DateClient.java` files from your textbook are on Moodle for this step. These programs comprise a client-server application, where the client serves up the current date and time. Change the number 6013 in each program to some other number between 7700 and 7999, but be sure that each program shares the same number. Try running these two programs from two separate windows. Study the code for these two programs to determine what each is doing.

    a. What is the role of the number 7XXX? (1 mark)

    b. Do each of the client-side program and the server-side program use the same port number to receive messages from the other? (1 mark)

    c. Provide a line of code that you could add to the client program to print the port number on the client-side. Put that line of code in your `lab4.txt` file under question 6c. (1 mark)

    d. Provide a line of code that you could add to the server program that would print the message:

        `Host` *W.X.Y.Z* `has requested the date.`

        where *W.X.Y.Z* is the IP address of the client making the request. Put that line of code in your `lab4.txt` file under question 6 and indicate where in the DateServer.Java file you would put the line. (1 mark)

If you want to learn more about Java sockets, visit the Java documentations site (https://docs.oracle.com/javase/7/docs/api/).

Submit:

Submit your `lab4.txt` file to Moodle