CSCE 451/851 Operating Systems Principles

Fall 1999
Steve Goddard

Programming Assignment 2 (PA2), September 9
Due: 6:00pm September 22

____________________________________________________________

Did You Order this Product? revisited

In this assignment you will implement the product problem using a client/server paradigm. The assignment is constructed in two parts. The first part is required. The second part is required for CSCE 851 students and optional for CSCE 451 students (33% bonus). In the first part, you will implement the product problem using a client/server paradigm in which the two processes communicate using Unix sockets. The client and server may (but need not) be running on separate machines. In the second part, you will use multi-threaded I/O in the client.

Information on Sockets

I covered the basic socket calls, the role they play in the client and the server, and the order in which they are called in class. I will summarize the information here to give you a starting place, but you should also study the example code provided with the project, which is located in

http://www.cse.unl.edu/~goddard/Courses/CSCE451/StandardHandouts/IPCexample.tar.gz.

While both of these approaches will help, there is no substitute for reading the manual pages. They are easily found using the man or xman commands.

The first important point is that the socket interface assumes that the client-server relationship holds between two processes. One is the server, which waits for requests at a fixed port address. The fixed address is the equivalent of waiting at a known phone number. The client requests service from the server by calling its number from anywhere it likes. The telephone analogy is not perfect, though, since the processes involved also execute system calls which are the equivalent of building and destroying the telephones, as well as establishing their numbers!

The first system call of concern is the socket() call, which creates a file descriptor attached to a socket structure. This creates the communication point in the process that calls it. Both the client and the server call socket(). Next, the server uses the bind() call to establish its address, and listen() to define the number of pending calls that may wait for attention from the server. The server then calls accept() when it wishes to accept a request for service. Note that accept() returns a new socket, unrelated to the original socket created by the server, which provides a connection to the client making the request which has just been accepted.
After the client creates its socket using the `socket()` call, it needs to make a connection to the server. The client can first bind its socket to a specific address, although it need not do so. Whether it picks its port address or not, it requests a connection to the server using the `connect()` call. At this point the client and server are in communication. Each process reads and writes to the sockets as they would read and write to files. When finished, the client and server discard the sockets assuming the `close()` system call.

The example code available from the class Web page illustrates the use of sockets. The following pseudo code provides a summary of the system calls required to establish a connection.

```
Server
  fd = socket()
  bind(fd, server address)
  listen(fd, queue-length)
  nfd = accept(fd, ... )
  read requests
  write answers
  close(nfd)
  close(fd)

Client
  fd = socket()
  [bind(fd,optional explicit client address)]
  connect(fd, server-address);
  write requests
  read answers
  close(fd)
```

See the handouts on System Calls and IPC and Sockets for more information on how these calls actually work.

**Part I: Minimal Client-Server**

Use the client/sever paradigm to implement product problem. You should write two programs, a server program (process) and a client program (process); the programs should be compiled separately, so that they each have their own executable image. The two processes will communicate with each other in a connection-oriented manner using sockets.

The client process should first `connect()` to the server process (after creating the socket, of course). The client process should then read from standard input (`stdin`) and send the input data to the server (via the socket) without examining the input data. Each line of input data represents one input case to be sent to the server, which will process it using product/sort algorithm and send the results back.

The server process should `accept()` an incoming connection request (after creating, binding, and listening on the socket), read the input characters from the socket (two non-negative numbers represent a test case), execute product/sort algorithm, and write the line of results to the client via the socket (the client created).

The client should then receive characters from the server by reading from the socket (that the client created) and write the results to standard output (`stdout`).

As before, multiple test cases are read from standard input (i.e., the keyboard) and the results are written to standard output (i.e., the “screen”). For each case, the input contains a single line with two non-negative integers. The input test cases are terminated be either an End-Of-File (EOF) or a line that contains two negative integers.
Thus, your client simply reads from stdin, sends the data to the server, receives the formatted results from the server, and writes the formatted lines of text to stdout. The server is essentially the same program you wrote for PA1 except now it receives input from its socket and writes data to a socket the client creates, and it outputs lines of text as it processes them. As before,

- a “line of input” is defined as two integers plus a carriage return.
- your program should not output user prompts, debugging information, status messages, etc.

The client should terminate when it detects an EOF on stdin or from the socket. The server should terminate when the first pair of negative integers is received or an EOF is detected, after closing all sockets. (A more typical scenario would be for the server process to be in an infinite loop: accepting a connection, replying on that connection, and then closing the accepted connection on each pass through the loop.)

Additional Notes

1. Note that because of byte ordering conventions ("big-endian" versus "little endian") on different machines, your client and server may not correctly interpret the contents of an exchanged message. If your client and server are running on different machines with different byte ordering conventions (e.g., one is running on a DEC workstation and one is running on a SUN SPARC), they will not interpret the content of each others’ structures correctly. You should use the htonl() and ntohl() functions to insure that all machines interpret the byte orderings in the same manner.

2. In writing your code, make sure to check for an error return from all system calls. If there is an error, the system declared global variable, errno, will give you information about the type of error that occurred:

```
#include <errno.h>

......

if ( (sockid = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
   printf("error creating client socket, error%d\n",errno);
   perror("meaning:"); exit(0);
}
```

See the man pages errno(2) and perror(2) for a description of the error codes and the use of perror.

3. Make sure you close every socket (file descriptor) that you use in your program. If you abort your program, the socket may still hang around and the next time you try and bind a new socket to the port ID you previously used (but never closed), you may get an error. Also, please be aware that port ID’s, when bound to sockets, are system-wide values and thus other students may be using the port number you are trying to use. With this in mind, I suggest you use the last four digits of your SS # and add it to 10,000 to create your port number. The reason for the addition is that port numbers below 5000 are reserved for system use, and those between 5,000-10,000 are used by lots of local applications. If you
follow this rule the likelihood of colliding with another program using the same port at the same time is very low.

Part II: Multiplexed I/O

Required for CSCE 851 students. 33% bonus for CSCE 451 students!

Implement a modified version of the product problem using a multi-threaded client. In this case, the server ignores all cases in which the first integer is 1. The client is not allowed to examine the input before sending it to the server. Thus, the client does not know when it will receive results from the server.

Your client will need to multiplex input processing so that it can read from either stdin or the socket. The major problem here is not knowing when a formatted line of output will be ready for display. Thus, I/O must be multiplexed. Another problem is coordinating the shutdown of all of the processes.

Grading Policy for Programs

The programs you hand in should work correctly and be documented. When you hand in your programming assignment, you should include:

1. A program listing containing in-line documentation.

2. A separate (typed) document of approximately two pages describing the overall program design, a verbal description of "how it works" including the basics of what the system is doing underneath, and design tradeoffs considered and made. Also describe possible improvements and extensions to your program (and sketch how they might be made).

3. A separate description of the tests you ran on your program to convince yourself that it is indeed correct. Also describe any cases for which your program is known not to work correctly.

4. A make file that compiles your program(s).

Please hand in your source files for all parts of this project.

The program should be neatly formatted (i.e., easy to read) and structured and documented according to the guidelines distributed in class. Use the handin program to submit your program(s) for grading. This is assignment 2. Your grade will be determined as follows:

Program Listing
   works correctly 40%
   in-line documentation 15%
   quality of design 25%
Design Document 15%
Thoroughness of test cases 05%

START EARLY. THIS IS HARDER THAN IT LOOKS!