CS 422 Fall 2014

Lab 1: System Programming Practice and Getting Started with Network Programming

Due: 09/21/2014 (Sun), 11:59 PM

Objective

The objective of this introductory lab is to do practice exercises involving system programming and getting started on simple examples of network programming that treat the network as a black box.

Reading

Read chapters 1, 2 and 3 from Comer (textbook).

Problems [240 pts]

Problem 1 (70 pts)

Write an app, myreminder, in C that takes as input a file name

% myreminder reminder-file

as command-line argument and prints reminders to standard output (i.e., terminal) as specified in

reminder-file. An example format of the file (ASCII) is

2 go to lunch
5 call joe
11 meeting

where the first line specifies that after 2 seconds from now the message "go to lunch" should be output to stdout. The second line indicates that "call joe" should be output after 5 seconds from now, and similarly for the third reminder instance. When reading the reminder file, use the read() system call to read input one byte at a time instead of user libraries (e.g., stdio). When writing a reminder to stdout, use the write() system call. The size/unit of write() need not be one byte at a time. Before printing the actual reminder message, call gettimeofday() to compute the time of day and print it using write(). Insert newlines where appropriate so that the output is easily readable. The exact format is up to you.

The first step of your myreminder app is to register a signal handler for SIGALRM with the Linux kernel that will be invoked whenever a SIGALRM signal is raised. Make sure to use sigaction() to register signals, i.e., do not use signal(). The second step of your app is to read the file containing reminder instances and set future SIGALRMS using the alarm() system call. While doing so, the reminder messages read in should be stored in a data structure so that they can be printed to stdout by the SIGALRM handler when their respective alarms are raised.

Write code that is modular with each function stored in its own .c file. Use Makefile to automate compilation. Your code must be adequately documented. Put your files in a subdirectory mylab1/p1. Additional submission instructions are specified under turn-in instructions below.

Problem 2 (70 pts)

Write a pair of checksum apps, mychecksum1 and mychecksum2, where

% mychecksum1 file1 file2

mychecksum1 takes two file names as command-line arguments. file1, an arbitrary file, is read one byte at a

time using read(). Each byte is treated as an unsigned integer and added to a 64-bit checksum variable (unsigned 64-bit word) that is initialized as 0. After the last byte has been read, file1 is copied to file2 and the 64-bit checksum is appended to file2 as eight additional bytes. When doing so, follow the big endian convention where the most significant byte is appended first and the least significant byte is appended last. These 8 bytes serve as a checksum signature of file1.

mychecksum2 takes an arbitrary file with its last 8 bytes interpreted as a checksum in big endian format

% mychecksum2 file2 file1

It reads file2 using read() one byte at a time and computes the 64-bit unsigned checksum as in mychecksum1. When doing so, the last 8 bytes of file2 must be excluded from the summation. The last 8 bytes are read in and converted to little endian format, and the two 64-bit unsigned integers are compared. If they are equal, it is concluded that file2 has not been corrupted, i.e., it passes an integrity test. Of course, it is possible for two dfferent files to yield the same 64-bit checksum, hence passing the checksum does not guarantee that there was no corruption. If the checksums do not match, then it is assured that the original file has been corrupted. mychecksum2 strips the last 8 bytes from file2 and saves the resultant file in file1. mychecksum2, along with a match/does not match verdict and generating file1 that is stripped of the checksum signature, should print the two 64-bit checksum values to stdout in hex format. You may use the stdio library for this last step.

Write code that is modular with each function stored in its own .c file. Use Makefile to automate compilation. Your code must be adequately documented. Put your files in a subdirectory mylab1/p2.

Problem 3 (100 pts)

Background At the present, we are treating the network to which the LWSN B158 machines are connected to as a black box. That is, without knowing how "the network" does its job, we are going to write apps that make system calls to send and receive messages. The simplest method is using sendto() and recvfrom() system calls, which are part of a network programming API, that allows sending/receiving of individual messages or packets. As noted in class, since process IDs are not used as identifiers of apps in network communication, a process that wishes to engage in network communication must get an alternate name, a 16-bit port number, from its operating system. Some well-known port numbers are reserved for widely used

network services (e.g., port number 80 for HTTP) and others may be registered with IANA (Internet Assigned Numbers Authority) hence best avoided when developing private apps. The port numbers 49152-65535 are available for this purpose.

When developing a server app that needs to be contacted by a client, a process can request a specific port number from its OS that is then made known to clients through separate channels (e.g., phone call, published advertisement, well-known address/name servers). When developing a client, its port number is communicated to a server as part of its packet (hence known to the server upon receipt of the client packet), therefore any port number in the 49152-65535 range will do. When a client asks its OS to pick an unused port number, we call it an ephemeral port number. To reach a server, along with the server process's port number, we need to know an IP address of a network interface that is attached to the computing system (e.g., server, labtop, smartphone) where the server process runs. Many devices are multihomed, i.e., have multiple network interfaces. For example, a laptop may have network interfaces for Ethernet, WiFi, Bluetooth, among others. Not all network interfaces are configured with IP address (e.g., Bluetooth is typically used for device-to-device direct communication).

After obtaining an IP address of a network device on a computing system that a service process is running, we are in business. The client can craft a packet containing the message (called payload) and bookkeeping information such as the server's port number and IP address. The client process's port number and IP address (the client itself may be multihomed) are, by default, inscribed as part of the bookkeeping info in the header portion of a packet. Hence a packet consists of a payload and header. Some packets, as we will see later in the course, can be more complex.

Application layer ping app Write an application layer (i.e., an app running as a user process) ping-like app where the ping server, mypingd, runs as a process on one of our lab machines using port number, say, 50000. The ping server or daemon must run before a client can contact it

```
% mypingd 50000
```

where the command-line argument specifies its port number. The client ping-like app, myping, runs on a different lab machine using an ephemeral port assigned by its operating system. myping takes two command-line arguments where the first is the IP address of the ping server and the second its port number.

For example,

% myping 128.10.25.101 50000

specifies that mypingd uses port number 50000 and runs on sslab01.cs.purdue.edu whose Ethernet interface is configured with IP address 128.10.25.101.

The client myping uses sendto() to transmit a packet of size 1024 bytes whose payload is all 0's (the ASCII character of number 0). Just before calling sendto(), use gettimeofday() to record the send time. When a response from the server arrives, call gettimeofday() again to record the receive time. Take the difference of the receive and send times, and print to stdout (you may use the stdio library) the time taken to receive an acknowledgment (i.e., ping) from the server in unit of millisecond. Also, print the IP address and port number of the server, as inscribed in the packet header which should match the IP address and port number used by the client to contact the server.

In terms of client/server architecture, the server is an iterative server that handles the client request itself. In contrast, a concurrent server would delegate the processing of a task to a worker process. We will build concurrent server apps in subsequent labs. Thus mypingd issues a recvfrom() system call (by default, blocking), and when a packet from a client arrives sends a packet back to the client (whose IP address and port number are accessible from the received packet header) whose payload is the same as what the client sent. The client, after calling sendto(), waits on a response from the server by calling recvfrom().

sendto() and recvfrom() are system calls to a network protocol called UDP (User Datagram Protocol) which, along with TCP (Transmission Control Protocol), is one of the two major means for apps to communicate over the Internet (which includes our sslab network). A desirable feature of UDP is that its software overhead is small. Its drawback is that when packets go missing (the Internet is "leaky"), they are lost forever. Thus if the client message transmitted through sendto() did not reach the server, or the server's response packet did not make it to the client, myping may hang forever waiting for a server response to arrive. This issue is addressed in the Bonus Problem.

Write code that is modular with each function stored in its own .c file. Use Makefile to automate compilation. Your code must be adequately documented. Put your files in a subdirectory mylab1/p3. Additional submission

instructions are specified under turn-in instructions below.

Bonus Problem (50 pts)

As an extension of Problem 3, use SIGALRM to set a timeout at the client so that if a response from the server does not arrive within 5 seconds, the client terminates with a message indicating that the server is not reachable. Test that this feature works correctly by modifying the server so that it does not respond to client requests every other time (it's a temperamental server). Put your files in a subdirectory mylab1/p4. *Note: The Bonus Problem is optional. That is, bonus points count toward the 50% that lab assignments contribute to the course grade.*

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Create a directory named mylab1 under which the subdirectories p1, p2, p3 (and p4) and code submissions are organized. Go to the parent directory of mylab1 and type the command

turnin -c cs422 -p lab1 mylab1

Please note the assignment submission policy specified in the course home page.

Back to the CS 422 web page